#include <iostream>

#include <string>

using namespace std;

class Node {

public:

string keyword;

string meaning;

int height;

Node\* left;

Node\* right;

Node(string key, string value) {

keyword = key;

meaning = value;

height = 1;

left = NULL;

right = NULL;

}

};

class AVLTree {

private:

Node\* root;

public:

AVLTree() {

root = NULL;

}

int getHeight(Node\* node) {

if (node == NULL)

return 0;

return node->height;

}

int getBalanceFactor(Node\* node) {

if (node == NULL)

return 0;

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

}

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

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

if (node == NULL)

return new Node(key, value);

if (key < node->keyword)

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

else if (key > node->keyword)

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

else {

// Update the meaning if the keyword already exists

node->meaning = value;

return node;

}

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

int balance = getBalanceFactor(node);

// Left-Left case

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

return rotateRight(node);

// Right-Right case

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

return rotateLeft(node);

// Left-Right case

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

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

return rotateRight(node);

}

// Right-Left case

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 != NULL)

current = current->left;

return current;

}

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

if (root == NULL)

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 == NULL || root->right == NULL) {

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

if (temp == NULL) {

temp = root;

root = NULL;

} 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 == NULL)

return root;

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

int balance = getBalanceFactor(root);

// Left-Left case

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

return rotateRight(root);

// Right-Right case

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

return rotateLeft(root);

// Left-Right case

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

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

return rotateRight(root);

}

// Right-Left case

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

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

return rotateLeft(root);

}

return root;

}

void displayAscendingOrder(Node\* node) {

if (node != NULL) {

displayAscendingOrder(node->left);

cout << "Keyword: " << node->keyword << ", Meaning: " << node->meaning << endl;

displayAscendingOrder(node->right);

}

}

void displayDescendingOrder(Node\* node) {

if (node != NULL) {

displayDescendingOrder(node->right);

cout << "Keyword: " << node->keyword << ", Meaning: " << node->meaning << endl;

displayDescendingOrder(node->left);

}

}

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

if (node == NULL || node->keyword == key) {

comparisons++;

return node;

}

if (key < node->keyword) {

comparisons++;

return search(node->left, key, comparisons);

}

comparisons++;

return search(node->right, key, comparisons);

}

int findKeyword(string key) {

int comparisons = 0;

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

if (result != NULL) {

cout << "Keyword: " << result->keyword << ", Meaning: " << result->meaning << endl;

cout << "Total Comparisons: " << comparisons << endl;

} else {

cout << "Keyword not found." << endl;

}

return comparisons;

}

void addKeyword(string key, string value) {

root = insert(root, key, value);

cout << "Keyword added successfully." << endl;

}

void deleteKeyword(string key) {

root = deleteNode(root, key);

cout << "Keyword deleted successfully." << endl;

}

void displayAscending() {

if (root == NULL) {

cout << "The dictionary is empty." << endl;

return;

}

cout << "Dictionary in Ascending Order:" << endl;

displayAscendingOrder(root);

}

void displayDescending() {

if (root == NULL) {

cout << "The dictionary is empty." << endl;

return;

}

cout << "Dictionary in Descending Order:" << endl;

displayDescendingOrder(root);

}

};

int main() {

AVLTree dictionary;

dictionary.addKeyword("Apple", "A fruit.");

dictionary.addKeyword("Banana", "A curved fruit.");

dictionary.addKeyword("Cat", "A furry animal.");

dictionary.displayAscending();

cout << endl;

dictionary.displayDescending();

cout << endl;

dictionary.findKeyword("Apple");

cout << endl;

dictionary.deleteKeyword("Banana");

cout << endl;

dictionary.displayAscending();

cout << endl;

return 0;

}