#include <iostream>

#include <string>

using namespace std;

// Define the dict class for managing the dictionary

class dict {

dict \*root, \*node, \*left, \*right, \*tree1; // Pointers for tree structure

string s1, s2; // s1 = keyword, s2 = meaning of the keyword

int flag, flag1, flag2, flag3, cmp; // Flags for various checks

public:

dict() : flag(0), flag1(0), flag2(0), flag3(0), cmp(0), root(NULL) {} // Constructor initializing flags and root pointer

void input(); // Function to input a new keyword and meaning

void create\_root(dict\*, dict\*); // Function to create root node

void check\_same(dict\*, dict\*); // Function to check if the keyword already exists

void input\_display(); // Function to display all the words in the dictionary

void display(dict\*); // Function to display words in the dictionary

void input\_remove(); // Function to take input for deleting a keyword

dict\* remove(dict\*, string); // Function to remove a keyword from the dictionary

dict\* findmin(dict\*); // Function to find the minimum value in the tree (used for deletion)

void input\_find(); // Function to take input and find a keyword

dict\* find(dict\*, string); // Function to search for a keyword

void input\_update(); // Function to input a keyword for updating its meaning

dict\* update(dict\*, string); // Function to update the meaning of a keyword

};

// Function to input a new keyword and its meaning

void dict::input() {

node = new dict;

cout << "\nEnter the keyword:\n";

cin >> node->s1;

cout << "Enter the meaning of the keyword:\n";

cin.ignore();

getline(cin, node->s2); // Take the meaning of the keyword as input

create\_root(root, node); // Add the new node to the tree

}

// Function to create the root node or insert the keyword in the correct place in the tree

void dict::create\_root(dict \*tree, dict \*node1) {

if (root == NULL) { // If the tree is empty, set the root

root = node1;

root->left = root->right = NULL; // No children initially

cout << "\nRoot node created successfully" << endl;

return;

}

int result = tree->s1.compare(node1->s1); // Compare the keyword with the current node's keyword

check\_same(tree, node1); // Check if the keyword already exists

if (flag == 1) { // If keyword already exists, print an error message

cout << "The word you entered already exists.\n";

flag = 0;

} else {

if (result > 0) { // If the new keyword is lexicographically smaller, go left

if (tree->left != NULL) {

create\_root(tree->left, node1); // Recursively insert in the left subtree

} else {

tree->left = node1; // Insert as the left child

node1->left = node1->right = NULL;

cout << "Node added to left of " << tree->s1 << "\n";

}

} else { // If the new keyword is lexicographically larger, go right

if (tree->right != NULL) {

create\_root(tree->right, node1); // Recursively insert in the right subtree

} else {

tree->right = node1; // Insert as the right child

node1->left = node1->right = NULL;

cout << "Node added to right of " << tree->s1 << "\n";

}

}

}

}

// Function to check if the keyword already exists in the tree

void dict::check\_same(dict \*tree, dict \*node1) {

if (tree->s1 == node1->s1) { // If found, set the flag

flag = 1;

return;

} else if (tree->s1 > node1->s1) {

if (tree->left != NULL)

check\_same(tree->left, node1); // Search in the left subtree

} else {

if (tree->right != NULL)

check\_same(tree->right, node1); // Search in the right subtree

}

}

// Function to display the dictionary

void dict::input\_display() {

if (root != NULL) {

cout << "The words entered in the dictionary are:\n\n";

display(root); // Call the recursive display function

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

// Recursive function to display the dictionary in-order (sorted order)

void dict::display(dict \*tree) {

if (tree != NULL) {

display(tree->left); // Visit left child first (lexicographically smaller words)

cout << tree->s1 << " = " << tree->s2 << "\n\n"; // Print the keyword and its meaning

display(tree->right); // Visit right child (lexicographically larger words)

}

}

// Function to find the minimum value (leftmost node) in the tree

dict\* dict::findmin(dict \*tree) {

while (tree->left != NULL)

tree = tree->left; // Keep going left to find the smallest node

return tree;

}

// Function to remove a keyword from the dictionary

dict\* dict::remove(dict \*tree, string s3) {

if (tree == NULL) {

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

flag1 = 1;

return tree; // Word not found, return tree unchanged

}

if (tree->s1 > s3) { // If keyword is smaller, go left

tree->left = remove(tree->left, s3);

} else if (tree->s1 < s3) { // If keyword is larger, go right

tree->right = remove(tree->right, s3);

} else { // Found the node to delete

if (tree->left == NULL && tree->right == NULL) { // No children, just delete the node

delete tree;

return NULL;

} else if (tree->left == NULL) { // One child (right), replace with the right child

dict \*temp = tree->right;

delete tree;

return temp;

} else if (tree->right == NULL) { // One child (left), replace with the left child

dict \*temp = tree->left;

delete tree;

return temp;

} else { // Two children, find the in-order successor (smallest in the right subtree)

dict \*temp = findmin(tree->right);

tree->s1 = temp->s1;

tree->s2 = temp->s2;

tree->right = remove(tree->right, temp->s1); // Recursively remove the successor node

}

}

return tree;

}

// Function to take input and remove a keyword

void dict::input\_remove() {

if (root != NULL) {

cout << "\nEnter a keyword to be deleted:\n";

cin >> s1;

root = remove(root, s1); // Call the remove function

if (flag1 == 0) {

cout << "\nThe word '" << s1 << "' has been deleted.\n";

}

flag1 = 0;

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

// Function to take input and find a keyword in the dictionary

void dict::input\_find() {

if (root != NULL) {

cout << "\nEnter the keyword to be searched:\n";

cin >> s1;

find(root, s1); // Call the find function

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

// Function to search for a keyword in the tree

dict\* dict::find(dict \*tree, string s3) {

if (tree == NULL) {

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

return NULL;

}

if (tree->s1 == s3) { // If found, print the keyword and its meaning

cout << "\nWord found: " << tree->s1 << " - " << tree->s2 << "\n";

return tree;

}

return (tree->s1 > s3) ? find(tree->left, s3) : find(tree->right, s3); // Recursively search left or right

}

// Function to update the meaning of a keyword

void dict::input\_update() {

if (root != NULL) {

cout << "\nEnter the keyword to be updated:\n";

cin >> s1;

dict \*res = find(root, s1);

if (res) {

cout << "\nEnter the updated meaning:\n";

cin.ignore();

getline(cin, res->s2); // Update the meaning

cout << "\nThe meaning of '" << s1 << "' has been updated.\n";

}

} else {

cout << "\nThere are no words in the dictionary.\n";

}

}

// Main function to display the menu and execute operations

int main() {

int ch;

dict d;

do {

cout << "\n===== DICTIONARY MENU =====\n"

<< "1. Add new keyword\n"

<< "2. Display dictionary\n"

<< "3. Delete keyword\n"

<< "4. Find keyword\n"

<< "5. Update meaning\n"

<< "6. Exit\n"

<< "===========================\n"

<< "Enter your choice: ";

cin >> ch;

switch (ch) {

case 1: d.input(); break; // Add new keyword

case 2: d.input\_display(); break; // Display dictionary

case 3: d.input\_remove(); break; // Delete a keyword

case 4: d.input\_find(); break; // Find a keyword

case 5: d.input\_update(); break; // Update the meaning of a keyword

case 6: break; // Exit the program

default: cout << "\nInvalid option!\n"; // Handle invalid input

}

} while (ch != 6); // Repeat the menu until the user chooses to exit

return 0;

}