```
#include <iostream>
using namespace std;
class Node {
public:
string keyword;
string meaning;
Node* left;
Node* right;
int height;
Node(string a, string b) : keyword(a), meaning(b), left(NULL), right(NULL),
height(1) {}
};
class AVLTree {
public:
Node* root;
int comp;
AVLTree() : root(NULL), comp(0) {}
int getHeight(Node* node) {
if (node == nullptr)return 0;
else
return max(getHeight(node->left), getHeight(node->right)) + 1;
}
int getBalance(Node* node) {
if (node == nullptr)
return 0;
return getHeight(node->left) - getHeight(node->right);
}
Node* rightRotate(Node* y) {
Node* x = y->left;
Node* T2 = x->right;
x->right = y;
y->left = T2;
y->height = getHeight(y);
x->height = getHeight(x);
return x;
}
Node* leftRotate(Node* x) {
Node* y = x->right;
Node* T2 = y->left;
y->left = x;
x->right = T2;
x->height = getHeight(x);y->height = getHeight(y);
return y;
}
Node* findmin(Node* node) {
Node* current = node;
while (current->left != nullptr)
current = current->left;
return current;
}
Node* insert(Node* node, string keyword, string meaning) {
if (node == nullptr) {
```

```
return new Node(keyword, meaning);
}
if (keyword < node->keyword)
node->left = insert(node->left, keyword, meaning);
else if (keyword > node->keyword)
node->right = insert(node->right, keyword, meaning);
else
return node;
node->height = 1 + max(getHeight(node->left), getHeight(node->right));
int balance = getBalance(node);
// LLCase
if (balance > 1 && keyword < node->left->keyword)
return rightRotate(node);// RRCase
if (balance < -1 && keyword > node->right->keyword)
return leftRotate(node);
// LRCase
if (balance > 1 && keyword > node->left->keyword) {
node->left = leftRotate(node->left);
return rightRotate(node);
}
// RL Case
if (balance < -1 && keyword < node->right->keyword) {
node->right = rightRotate(node->right);
return leftRotate(node);
}
return node;
}
Node* deletenode(Node* root, string keyword) {
if (root == NULL) {
return NULL;
}
if (keyword < root->keyword) {
root->left = deletenode(root->left, keyword);
}
else if (keyword > root->keyword) {
root->right = deletenode(root->right, keyword);
}
else {
if (root->left == NULL || root->right == NULL) {Node* temp = root->left != NULL ? root->left :
root->right;
if (temp == NULL) {
temp = root;
root = NULL;
}
else
*root = *temp;
delete temp;
}
else {
Node* find = findmin(root->right);
root->keyword = find->keyword;
root->meaning = find->meaning;
```

```
root->right = deletenode(root->right, find->keyword);
}
}
if (root == NULL)
return root;
root->height = 1 + max(getHeight(root->left), getHeight(root->right));
int balance = getBalance(root);
if (balance > 1 && getBalance(root->left) >= 0) {
return rightRotate(root);
}
if (balance < -1 && getBalance(root->right) <= 0) {
return leftRotate(root);
}
if (balance > 1 && getBalance(root->left) < 0) {
root->left = leftRotate(root->left);
return rightRotate(root);}
if (balance < -1 && getBalance(root->right) > 0) {
root->right = rightRotate(root->right);
return leftRotate(root);
}
return root;
}
void inorder(Node* root) {
if (root != NULL) {
inorder(root->left);
cout << root->keyword << ": " << root->meaning << endl;
inorder(root->right);
}
}
void reverseInorder(Node* root) {
if (root == nullptr) return;
reverseInorder(root->right);
cout << root->keyword << ": " << root->meaning << endl;
reverseInorder(root->left);
}
Node* search(Node* root, string target) {
comp = 0;
while (root != NULL) {
comp++;
if (root == NULL || root->keyword == target) {
return root;
}if (target < root->keyword) {
root = root->left;
}
else if (target > root->keyword) {
root = root->right;
}
else {
return root;
}
}
return NULL;
```

```
}
Node* update(Node* root, string keyword, string newKeyword) {
Node* targetnode = search(root, keyword);
if (targetnode != NULL) {
string prev;
cout << "Enter previous meaning: ";
cin >> prev;
root = insert(root, newKeyword, prev);
root = deletenode(root, keyword);
}
return root;
}
};
int main() {
AVLTree dict;
int choice;
string keyword, meaning, newKeyword;
Node* R = nullptr;do {
cout << "We are performing AVL tree operations" << endl;
cout << "1. Insert" << endl;
cout << "2. Update" << endl;
cout << "3. Search" << endl;
cout << "4. Display" << endl;
cout << "5. Reverse" << endl;
cout << "6. Delete" << endl;
cout << "7. Exit" << endl;
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
case 1:
cout << "Enter keyword: ";
cin >> keyword;
cout << "Enter meaning: ";
cin >> meaning;
dict.root = dict.insert(dict.root, keyword, meaning);
break;
case 2:
cout << "Enter the keyword you want to update: ";
cin >> keyword;
cout << "Enter the new keyword: ";
cin >> newKeyword;
dict.root = dict.update(dict.root, keyword, newKeyword);
break;
case 3:
cout << "Enter the keyword to search: ";
cin >> keyword;R = dict.search(dict.root, keyword);
if (R != NULL) {
cout << "Meaning: " << R->meaning << endl;
cout << "Total comparisons required: " << dict.comp << endl;
}
else {
cout << "Keyword not found" << endl;
```

```
}
break;
case 4:
cout << "Inorder traversal:" << endl;
dict.inorder(dict.root);
break;
case 5:
cout << "Reverse inorder traversal:" << endl;
dict.reverseInorder(dict.root);
break;
case 6:
cout << "Enter keyword to delete: ";
cin >> keyword;
dict.root = dict.deletenode(dict.root, keyword);
cout<<"keyword deleted"<<endl;
break;
case 7:
cout << "Exiting..." << endl;
break;
default:cout << "Invalid choice" << endl;
break;
}
} while (choice != 7);
}
```