```
#include<iostream>
using namespace std;
class node{
public:
    string word, meaning;
    node* left;
    node* right;
    int ht;
};
class AVL_Tree{
public:
    node* root;
    AVL_Tree(){
    root = NULL;
    }
    node* createNode(string word, string meaning){
        node* newNode = new node;
        newNode->word = word;
        newNode->meaning = meaning;
        newNode->left = nullptr;
        newNode->right = nullptr;
        newNode->ht = 1;
        return newNode;
    }
    node* insert(node* root, string word, string meaning){
        if(root == NULL){
            root = createNode(word, meaning);
        }
        else if(word < root->word){
            root->left = insert(root->left, word, meaning);
        }
        else if(word > root->word){
```

```
        root->right = insert(root->right, word, meaning);
    }
    else{
        return root;
    }
    root->ht = height(root);
    int BF = get_BF(root);
    if((BF > 1) && (word < root->left->word)){
        return RR(root);
    }
    if((BF > 1) && (word > root->left->word)){
        root->left = RL(root->left);
        return RR(root);
    }
    if((BF < -1) && (word > root->right->word)){
        return RL(root);
    }
    if((BF < -1) && (word < root->right->word)){
        root->right = RR(root->right);
        return RL(root);
    }
    return root;
}
int height(node* root){
    if (root == NULL){
        return 0;
    }
    else{
        return(max(height(root->left),height(root->right)) + 1);
    }
}
int get_BF(node* root){
```

```
    if (root == NULL){
        return 0;
    }
    else{
        return (height(root->left) - height(root->right));
    }
}
node* RR(node* y){
    node*x = y->left;
    node* T = x->right;
    x->right = y;
    y->left = T;
    y->ht = height(y);
    x->ht = height(x);
    return x;
}
node* RL(node* y){
    node*x = y->right ;
    node*T = x->left ;
    x->left = y;
    y->right = T;
    y->ht = height(y);
    x->ht = height(x);
    return x;
}
void newkword(node *root,string word, string meaning){
    if(root->word > word){
        return newkword(root->left,word, meaning);
    }
    else if(root->word < word){
        return newkword(root->right,word, meaning);
    }
```

```
    else if(root->word == word){
        cout<<"\n Key is already present..."<<endl;
        cout<<"\n Updating meaning..."<<endl;
        root->meaning = meaning;
        cout<<"\n meaning updated succesfully"<<endl;
        return;
    }
}
void InorderA(node *root){
    if(root == NULL){
        return;
    }
    else{
        cout<<root->word<<" = "<<root->meaning<<" ";
        InorderA(root->left);
        InorderA(root->right);
    }
}
void InorderD(node *root){
    if(root == NULL){
        return;
    }
    else{
        InorderD(root->right);
        cout<<root->word<<" = "<<root->meaning<<" ";
        InorderD(root->left);
    }
}
int Search(node*root,string user_key,int comparision){
    if(root->word == user_key || root == NULL){
        cout<<"\n Key is present in tree"<<endl;
        comparision ++;
```

```
        return comparision;
    }
    else if(root->word > user_key){
        comparision ++;
        return Search(root->left,user_key,comparision);
    }
    else {
        comparision ++;
        return Search(root->right,user_key,comparision);
    }
}
node *MaxDataValue(node *root){
    if (root == NULL){
        return root;
    }
    else if(root->right == NULL){
        return root;
    }
    else{
        return MaxDataValue(root->right);
    }
}
node* deleteNode(node* root, string key) {
    if (root == NULL){
        return root;
    }
    if (key < root->word){
        root->left = deleteNode(root->left, key);
    }
    else if (key > root->word){
        root->right = deleteNode(root->right, key);
    }
```

```
        else{
            if (root->left == NULL) {
                node* temp = root->right;
                delete root;
                return temp;
            } else if (root->right == NULL) {
                node* temp = root->left;
                delete root;
                return temp;
            }
            node* temp = MaxDataValue(root->left);
            root->word = temp->word;
            root->meaning = temp->meaning;
            root->left = deleteNode(root->left, temp->word);
        }
        root->ht = height(root);
        int BF = get_BF(root);
        if (BF > 1 && get_BF(root->left) >= 0)
            return RR(root);
        if (BF > 1 && get_BF(root->left) < 0) {
            root->left = RL(root->left);
            return RR(root);
        }
        if (BF < -1 && get_BF(root->right) <= 0)
            return RL(root);
        if (BF < -1 && get_BF(root->right) > 0) {
            root->right = RR(root->right);
            return RL(root);
        }
        return root;
    }
};
```

```
int main(){
    AVL_Tree MyTree;
    int ch,comp;
    string word,meaning,T;
    string key;
    string temp_key;
    cout<<"Menu"<<endl;
    while (true){
        cout<<"1.Creating Tree."<<endl;
        cout<<"2.Insert Element."<<endl;
        cout<<"3.Update keyword meaning."<<endl;
        cout<<"4.Ascending traversing"<<endl;
        cout<<"5.Descending traversing"<<endl;
        cout<<"6.Search Value."<<endl;
        cout<<"7.Exit."<<endl;
        cout<<endl<<"\n Enter your choice: ";
        cin>>ch;
        switch(ch){
            case 1:
                cout<<"\nEnter data of root node of tree (key): ";
                cin>>word;
                cout<<"Enter data of root node of tree (meaning): ";
                cin>>meaning;
                MyTree.root = MyTree.insert(MyTree.root, word, meaning);
                cout<<"Tree created successfully"<<endl;
                break;
            case 2:
                cout<<"\nEnter data of new node of tree (key): ";
                cin>>word;
                cout<<"Enter data of new node of tree (meaning): ";
                cin>>meaning;
                MyTree.root = MyTree.insert(MyTree.root, word, meaning);
```

```
            cout<<"Node added successfully"<<endl;
            break;
        case 3:
            cout<<"\nEnter data of node of tree you want to update (new key): ";
            cin>>word;
            cout<<"Enter data of node of tree you want to update  (new meaning):
";
            cin>>meaning;
            MyTree.newkword(MyTree.root, word, meaning);
            break;
        case 4:
            cout<<"\nAscending traversing:"<<endl;
            MyTree.InorderA(MyTree.root);
            break;
        case 5:
            cout<<"\nDescending traversing:"<<endl;
            MyTree.InorderD(MyTree.root);
            break;
        case 6:
            cout<<"\nEnter the key you want to find: ";
            cin>>word;
            comp = MyTree.Search(MyTree.root, word, 0);
            cout<<"\nNumber of comparisons required: "<<comp<<endl;
            break;
        case 7:
            cout<<"\nExiting..."<<endl;
            exit(0);
            break;
      }
   }
   re
```