```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class DictNode {
public:
    string key;
    string meaning;
    DictNode* left;
    DictNode* right;
    DictNode(const string& k, const string& m) : key(k), meaning(m), left(nullptr), right(nullptr) {}
};
class Dictionary {
private:
    int comparisons;
    DictNode* root;
    vector<DictNode*> insertionOrder;
    public:
    Dictionary() : root(nullptr) {}
    void add(const string& key, const string& meaning) {
        root = sum(root, key, meaning);
    }
    void remove(const string& key) {
        root = removeNode(root, key);
    }
    void find(const string& key) {
        comparisons = 0;
        DictNode* result = searching(root, key);
        if (result != nullptr) {
            cout << "Keyword found.\n";cout << "Keyword: " << result->key << ", Meaning: " << result->meaning << "\n";
            cout << "Number of comparisons: " << comparisons << "\n";
```

```
        } else {
            cout << "Keyword not found.\n";
        }
    }

    void display() {
        if (!insertionOrder.empty()) {
            for (DictNode* node : insertionOrder) {
                cout << "Keyword: " << node->key << ", Meaning: " << node->meaning << "\n";
            }
        } else {
            cout << "\nDictionary is empty.\n";
        }
    }
    void ascending() {
        if (root != nullptr) {
            Inorder(root);
        } else {
            cout << "\nDictionary is empty.\n";
        }
    }
    void descending() {
        if (root != nullptr) {
            inverseInorder(root);
        } else {
            cout << "\nDictionary is empty.\n";
        }
    }
    int Maxcompare() {
       return Maxheight(root);
    }
```

```
    private:
        DictNode* sum(DictNode* node, const string& key, const string& meaning)
{
        if (node == nullptr) {
            DictNode* newNode = new DictNode(key, meaning);
            insertionOrder.push_back(newNode);
            return newNode;
            }
            int result = key.compare(node->key);
            if (result < 0) {
                node->left = sum(node->left, key, meaning);
            } else if (result > 0) {
                node->right = sum(node->right, key, meaning);
            } else {
                cout << "Keyword already exists.\n";
            }
            return node;
        }
        void Inorder(DictNode* node) {
        if (node != nullptr) {
        Inorder(node->left);
        cout << "Keyword: " << node->key << ", Meaning: " << node->meaning
<< "\n";
        Inorder(node->right);
        }
    }
    void inverseInorder(DictNode* node) {
        if (node != nullptr) {
            inverseInorder(node->right);
            cout << "Keyword: " << node->key << ", Meaning: " << node-
>meaning << "\n";
            inverseInorder(node->left);
        }
```

```
}
DictNode* removeNode(DictNode* node, const string& key) {
    if (node == nullptr) {
        return nullptr;
    }
    int result = key.compare(node->key);
    if (result < 0) {
        node->left = removeNode(node->left, key);
    } else if (result > 0) {
        node->right = removeNode(node->right, key);
    } else {
    if (node->left == nullptr) {
        DictNode* temp = node->right;
        delete node;
        return temp;
    } else if (node->right == nullptr) {
        DictNode* temp = node->left;
        delete node;
        return temp;
    }
    DictNode* temp = findMin(node->right);
    node->key = temp->key;
    node->meaning = temp->meaning;
    node->right = removeNode(node->right, temp->key);
    }
    return node;
}
DictNode* findMin(DictNode* node) {
    while (node->left != nullptr) {
        node = node->left;
        }
    return node;
```

```
        }
    DictNode* searching(DictNode* node, const string& key) {
        comparisons++;
        if (node == nullptr || key == node->key) {
            return node;
        }
        if (key < node->key) {
            return searching(node->left, key);
        } else {
            return searching(node->right, key);
        }
    }
    int Maxheight(DictNode* node) {
        if (node == nullptr) {
            return 0;
        }
        int leftHeight = Maxheight(node->left);
        int rightHeight = Maxheight(node->right);return max(leftHeight,
rightHeight) + 1;
    }
};
int main() {
    Dictionary dictionary;
    int choice;
    string key, meaning;
    do {
        cout << "\nMenu:\n";
        cout << "1. Add keyword\n";
        cout << "2. Display dictionary\n";
        cout << "3. Max comparisons\n";
        cout << "4. Display dictionary (ascending/descending)\n";
        cout << "5. Exit\n";
```

```
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Enter keyword: ";
            cin >> key;
            cout << "Enter meaning: ";
            cin.ignore();
            getline(cin, meaning);
            dictionary.add(key, meaning);
            break;
        case 2:
            cout << "Displaying dictionary:\n";
            dictionary.display();
            break;
        case 3:
            cout << "Maximum comparisons required: " << dictionary.Maxcompare()
<< "\n";
            break;
        case 4:
        int displayChoice;
            cout << "Choose display order:\n";
            cout << "1. Ascending order\n";
            cout << "2. Descending order\n";
            cin >> displayChoice;
            if (displayChoice == 1) {
                cout << "Displaying dictionary in ascending order:\n";
                dictionary.ascending();
            }
            else if (displayChoice == 2) {
                cout << "Displaying dictionary in descending order:\n";
                dictionary.descending();
```

```
            } else {
                cout << "Invalid choice for display order.\n";
            }
            break;
        case 5:
            cout << "Exiting program.\n";
            break;
        default:
            cout << "Invalid choice. Please enter a number between 1 and 8.\n";
            break;
        }
    } while (choice != 5);
    return 0;
}
```