Assignment No. 8

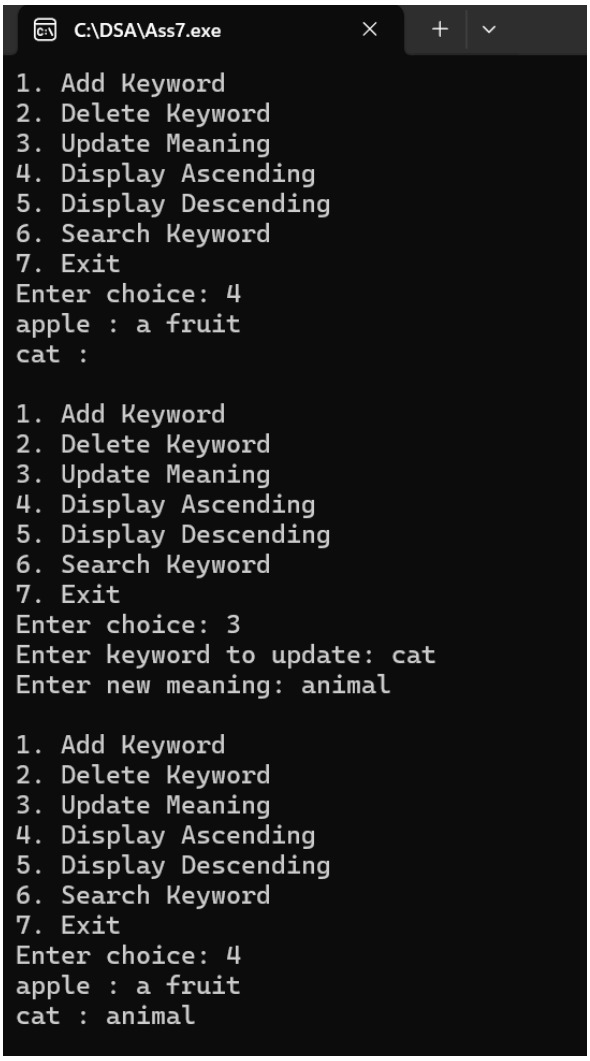
Q. A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

```
#include <iostream>
#include <string>
using namespace std;

struct Node {
    string keyword;
    string meaning;
    Node* left;
    Node* right;
    int height;

    Node(string k, string m) {
        keyword = k;
        meaning = m;
        left = right = NULL;
        height = 1;
    }
};

int height(Node* n) {
    return n ? n->height : 0;
}

int balanceFactor(Node* n) {
    return height(n->left) - height(n->right);
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

Node* insert(Node* root, string key, string meaning) {
    if (!root)
        return new Node(key, meaning);

    if (key < root->keyword)
```

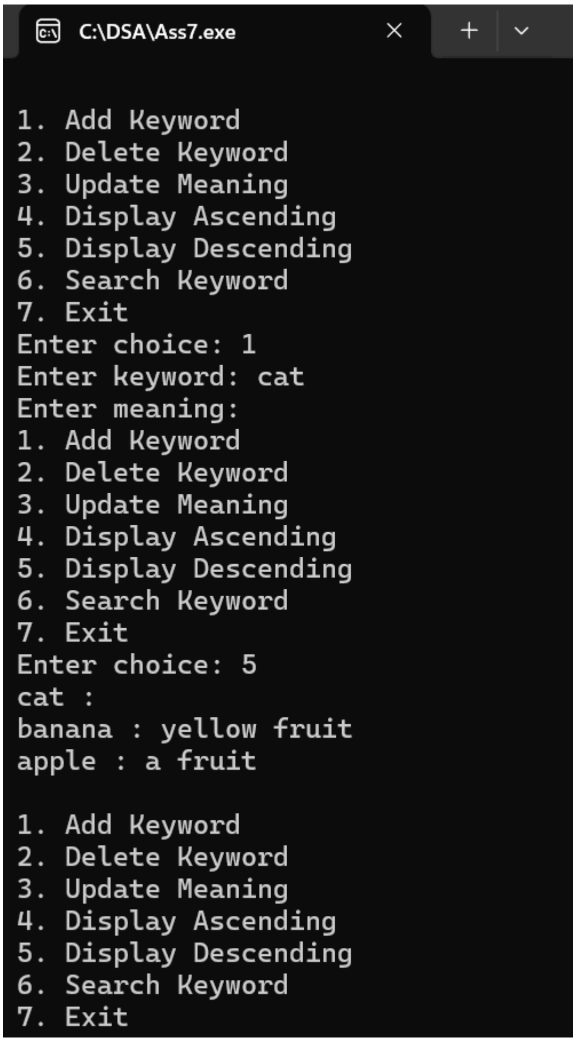
```
        root->left = insert(root->left, key, meaning);
    else if (key > root->keyword)
        root->right = insert(root->right, key, meaning);
    else {
        root->meaning = meaning;
        return root;
    }

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

    int balance = balanceFactor(root);

    if (balance > 1 && key < root->left->keyword)
        return rightRotate(root);
    if (balance < -1 && key > root->right->keyword)
        return leftRotate(root);
    if (balance > 1 && key > root->left->keyword) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && key < root->right->keyword) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left)
        current = current->left;
    return current;
}

Node* deleteNode(Node* root, string key) {
    if (!root)
        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 || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            delete root;
            return temp;
        } else {
            Node* temp = minValueNode(root->right);
            root->keyword = temp->keyword;
            root->meaning = temp->meaning;
            root->right = deleteNode(root->right, temp->keyword);
        }
    }

    root->height = 1 + max(height(root->left), height(root->right));
    int balance = balanceFactor(root);

    if (balance > 1 && balanceFactor(root->left) >= 0)
        return rightRotate(root);
    if (balance > 1 && balanceFactor(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && balanceFactor(root->right) <= 0)
        return leftRotate(root);
    if (balance < -1 && balanceFactor(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

void displayAscending(Node* root) {
    if (root) {
        displayAscending(root->left);
        cout << root->keyword << " : " << root->meaning << endl;
        displayAscending(root->right);
    }
}

void displayDescending(Node* root) {
    if (root) {
        displayDescending(root->right);
        cout << root->keyword << " : " << root->meaning << endl;
        displayDescending(root->left);
    }
}

int search(Node* root, string key, int& comparisons) {
    comparisons++;
    if (!root)
        return 0;
    if (root->keyword == key)
```

```
        return 1;
    if (key < root->keyword)
        return search(root->left, key, comparisons);
    else
        return search(root->right, key, comparisons);
}

int main() {
    Node* root = NULL;
    int choice;
    string keyword, meaning;

    while (true) {
        cout << "\n1. Add Keyword\n2. Delete Keyword\n3. Update Meaning\n4. Display
Ascending\n5. Display Descending\n6. Search Keyword\n7. Exit\nEnter choice: ";
        cin >> choice;

        if (choice == 1) {
            cout << "Enter keyword: ";
            cin >> keyword;
            cout << "Enter meaning: ";
            cin.ignore();
            getline(cin, meaning);
            root = insert(root, keyword, meaning);
        }
        else if (choice == 2) {
            cout << "Enter keyword to delete: ";
            cin >> keyword;
            root = deleteNode(root, keyword);
        }
        else if (choice == 3) {
            cout << "Enter keyword to update: ";
            cin >> keyword;
            cout << "Enter new meaning: ";
            cin.ignore();
            getline(cin, meaning);
            root = insert(root, keyword, meaning);
        }
        else if (choice == 4) {
            displayAscending(root);
        }
        else if (choice == 5) {
            displayDescending(root);
        }
        else if (choice == 6) {
            cout << "Enter keyword to search: ";
            cin >> keyword;
            int comparisons = 0;
            if (search(root, keyword, comparisons))
                cout << keyword << " found with " << comparisons << " comparisons.\n";
            else
                cout << keyword << " not found. Comparisons made: " << comparisons << endl;
        }
        else if (choice == 7) {
            break;
        }
        else {
            cout << "Invalid choice.\n";
        }
    }

    return 0;
```

Output :



```
1. Add Keyword
2. Delete Keyword
3. Update Meaning
4. Display Ascending
5. Display Descending
6. Search Keyword
7. Exit
Enter choice: 4
apple : a fruit
cat :

1. Add Keyword
2. Delete Keyword
3. Update Meaning
4. Display Ascending
5. Display Descending
6. Search Keyword
7. Exit
Enter choice: 3
Enter keyword to update: cat
Enter new meaning: animal

1. Add Keyword
2. Delete Keyword
3. Update Meaning
4. Display Ascending
5. Display Descending
6. Search Keyword
7. Exit
Enter choice: 4
apple : a fruit
cat : animal
```



```

1. Add Keyword
2. Delete Keyword
3. Update Meaning
4. Display Ascending
5. Display Descending
6. Search Keyword
7. Exit
Enter choice: 1
Enter keyword: cat
Enter meaning:
1. Add Keyword
2. Delete Keyword
3. Update Meaning
4. Display Ascending
5. Display Descending
6. Search Keyword
7. Exit
Enter choice: 5
cat :
banana : yellow fruit
apple : a fruit

1. Add Keyword
2. Delete Keyword
3. Update Meaning
4. Display Ascending
5. Display Descending
6. Search Keyword
7. Exit
```