Assignment 9

```
#include <iostream>

#include <string>

using namespace std;

class avlnode {
public:
    string keyword;
    string meaning;
    avlnode *left, *right;
    int height;

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

class dictionary {
private:
    avlnode *root;

    int getHeight(avlnode *node) {
        return node ? node->height : 0;
    }

    int getBalanceFactor(avlnode *node) {
        return node ? getHeight(node->left) - getHeight(node->right) : 0;
    }

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

    avlnode *leftRotate(avlnode *x) {
        avlnode *y = x->right;
        avlnode *T2 = y->left;
        y->left = x;
        x->right = T2;
        x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
        y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
```

```
    return y;
}

avlnode *insert(avlnode *node, string key, string meaning) {
    if (!node) return new avlnode(key, meaning);

    if (key < node->keyword)
        node->left = insert(node->left, key, meaning);
    else if (key > node->keyword)
        node->right = insert(node->right, key, meaning);
    else {
        cout << "Keyword already exists.\n";
        return node;
    }

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int balance = getBalanceFactor(node);

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

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

avlnode *deleteNode(avlnode *node, string key) {
    if (!node) return node;

    if (key < node->keyword)
        node->left = deleteNode(node->left, key);
    else if (key > node->keyword)
        node->right = deleteNode(node->right, key);
    else {
        if (!node->left || !node->right) {
            avlnode *temp = node->left ? node->left : node->right;
            if (!temp) {
                temp = node;
                node = NULL;
            } else
                *node = *temp;
            delete temp;
        } else {
            avlnode *temp = minValueNode(node->right);
            node->keyword = temp->keyword;
            node->meaning = temp->meaning;
```

```
            node->right = deleteNode(node->right, temp->keyword);
        }
    }

    if (!node) return node;

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int balance = getBalanceFactor(node);

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

void inorder(avlnode *node) {
    if (node) {
        inorder(node->left);
        cout << node->keyword << ": " << node->meaning << endl;
        inorder(node->right);
    }
}

void descending(avlnode *node) {
    if (node) {
        descending(node->right);
        cout << node->keyword << ": " << node->meaning << endl;
        descending(node->left);
    }
}

avlnode *search(avlnode *node, string key) {
    if (!node || node->keyword == key) return node;
    if (key < node->keyword) return search(node->left, key);
    return search(node->right, key);
}

public:
dictionary() { root = NULL; }

void insert(string key, string meaning) {
    root = insert(root, key, meaning);
}

void deleteKey(string key) {
    root = deleteNode(root, key);
}

void update(string key, string newMeaning) {
    avlnode *node = search(root, key);
```

```
        if (node)
            node->meaning = newMeaning;
        else
            cout << "Keyword not found.\n";
    }

    void displayAscending() {
        inorder(root);
    }

    void displayDescending() {
        descending(root);
    }

    void searchWord(string key) {
        avlnode *node = search(root, key);
        if (node)
            cout << "Found: " << node->keyword << " -> " << node->meaning << endl;
        else
            cout << "Not found.\n";
    }
};

int main() {
    dictionary dict;
    int choice;
    string key, meaning;

    do {
        cout << "\n1. Insert\n2. Update\n3. Delete\n4. Display Ascending\n5. Display Descending\n6.
Search\n7. Quit\n";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter keyword: ";
                cin >> key;
                cout << "Enter meaning: ";
                cin.ignore();
                getline(cin, meaning);
                dict.insert(key, meaning);
                break;
            case 2:
                cout << "Enter keyword to update: ";
                cin >> key;
                cout << "Enter new meaning: ";
                cin.ignore();
                getline(cin, meaning);
                dict.update(key, meaning);
                break;
            case 3:
                cout << "Enter keyword to delete: ";
                cin >> key;
                dict.deleteKey(key);
                break;
            case 4:
                dict.displayAscending();
                break;
            case 5:
                dict.displayDescending();
```

```
            break;
        case 6:
            cout << "Enter keyword to search: ";
            cin >> key;
            dict.searchWord(key);
            break;
        case 7:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice.\n";
      }
   } while (choice != 7);

   return 0;
}
```

Output:

```
1. Insert
2. Update
3. Delete
4. Display Ascending
5. Display Descending
6. Search
7. Quit
Your choice: 1
Enter keyword: apple
Enter meaning: fruit

Your choice: 1
Enter keyword: cat
Enter meaning: animal

Your choice: 4
apple: fruit
cat: animal

Your choice: 2
Enter keyword to update: cat
Enter new meaning: pet
Meaning updated successfully.

Your choice: 6
Enter keyword to search: cat
Found: cat -> pet

Your choice: 3
Enter keyword to delete: apple

Your choice: 4
cat: pet

Your choice: 7
Exiting...
```