```
#include <iostream>

#include <cstring>

#include <algorithm>

using namespace std;


struct Node {

   char key[50];

   char meaning[100];

   Node* left;

   Node* right;

   int height;


   Node(const char* k, const char* m) {

      strcpy(key, k);

      strcpy(meaning, m);

      left = right = nullptr;

      height = 1;

   }

};


int getHeight(Node* n) {

   return n ? n->height : 0;

}


int getBalance(Node* n) {

   return n ? getHeight(n->left) - getHeight(n->right) : 0;

}
```

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* 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;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* 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;
}

Node* insert(Node* node, const char* key, const char* meaning) {
    if (!node) return new Node(key, meaning);

    if (strcmp(key, node->key) < 0)
        node->left = insert(node->left, key, meaning);
    else if (strcmp(key, node->key) > 0)
        node->right = insert(node->right, key, meaning);
    else
```

```
        return node; // Duplicates not allowed

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

    int balance = getBalance(node);

    if (balance > 1 && strcmp(key, node->left->key) < 0)
        return rightRotate(node);

    if (balance < -1 && strcmp(key, node->right->key) > 0)
        return leftRotate(node);

    if (balance > 1 && strcmp(key, node->left->key) > 0) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && strcmp(key, node->right->key) < 0) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

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


Node* deleteNode(Node* root, const char* key) {
    if (!root) return root;


    if (strcmp(key, root->key) < 0)
        root->left = deleteNode(root->left, key);
    else if (strcmp(key, root->key) > 0)
        root->right = deleteNode(root->right, key);
    else {
        if (!root->left || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            if (!temp) {
                temp = root;
                root = nullptr;
            } else
                *root = *temp;
            delete temp;
        } else {
            Node* temp = minValueNode(root->right);
            strcpy(root->key, temp->key);
            strcpy(root->meaning, temp->meaning);
            root->right = deleteNode(root->right, temp->key);
        }
    }


    if (!root) 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->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

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

    return root;
}

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

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

Node* search(Node* root, const char* key, int &comparisons) {
    comparisons++;
    if (!root) return nullptr;

    if (strcmp(key, root->key) == 0)
        return root;
    else if (strcmp(key, root->key) < 0)
        return search(root->left, key, comparisons);
    else
        return search(root->right, key, comparisons);
}

int main() {
    Node* root = nullptr;
    int choice;
    char key[50], meaning[100];
```

```
do {
  cout << "\nMenu:\n";
  cout << "1. Add new keyword\n";
  cout << "2. Delete keyword\n";
  cout << "3. Update meaning\n";
  cout << "4. Display in Ascending order\n";
  cout << "5. Display in Descending order\n";
  cout << "6. Search for a keyword\n";
  cout << "7. Max comparisons for search (Tree Height)\n";
  cout << "8. Exit\n";
  cout << "Enter your choice: ";
  cin >> choice;
  cin.ignore();

  switch(choice) {
    case 1:
      cout << "Enter keyword: ";
      cin.getline(key, 50);
      cout << "Enter meaning: ";
      cin.getline(meaning, 100);
      root = insert(root, key, meaning);
      break;

    case 2:
      cout << "Enter keyword to delete: ";
      cin.getline(key, 50);
      root = deleteNode(root, key);
      break;
```

```
case 3:
  cout << "Enter keyword to update: ";
  cin.getline(key, 50);
  {
    int cmp = 0;
    Node* node = search(root, key, cmp);
    if (node) {
      cout << "Enter new meaning: ";
      cin.getline(meaning, 100);
      strcpy(node->meaning, meaning);
      cout << "Updated successfully.\n";
    } else {
      cout << "Keyword not found.\n";
    }
  }
  break;

case 4:
  cout << "Dictionary in Ascending order:\n";
  inorder(root);
  break;

case 5:
  cout << "Dictionary in Descending order:\n";
  reverseInorder(root);
  break;
```

```
        case 6:
            cout << "Enter keyword to search: ";
            cin.getline(key, 50);
            {
                int cmp = 0;
                Node* node = search(root, key, cmp);
                if (node) {
                    cout << "Found: " << node->key << " : " << node->meaning << endl;
                    cout << "Comparisons made: " << cmp << endl;
                } else {
                    cout << "Keyword not found. Comparisons made: " << cmp << endl;
                }
            }
            break;

        case 7:
            cout << "Maximum comparisons for search = Tree Height = " << getHeight(root)
<< endl;
            break;

        case 8:
            cout << "Exiting.\n";
            break;

        default:
            cout << "Invalid choice.\n";
      }
    } while (choice != 8);
    return 0;
}
```