Assignment no 9

Problem statement: 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, meaning;
    Node* left;
    Node* right;
    int height;

    Node(string key, string mean) {
        keyword = key;
        meaning = mean;
        left = right = nullptr;
        height = 1;
    }
};

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

int getBalanceFactor(Node* node) {
    return (node == nullptr) ? 0 : getHeight(node->left) - getHeight(node->right);
}
```

```
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, string keyword, string meaning) {
    if (node == nullptr)
        return new Node(keyword, meaning);
```

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

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

int balance = getBalanceFactor(node);

if (balance > 1 && keyword < node->left->keyword)
    return rightRotate(node);

if (balance < -1 && keyword > node->right->keyword)
    return leftRotate(node);

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

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

```
    return node;
}

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

Node* deleteNode(Node* root, string keyword) {
    if (root == nullptr)
        return root;

    if (keyword < root->keyword)
        root->left = deleteNode(root->left, keyword);
    else if (keyword > root->keyword)
        root->right = deleteNode(root->right, keyword);
    else {
        if (root->left == nullptr || root->right == nullptr) {
            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);
        }
    }
```

```
    if (root == nullptr) return root;

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

    int balance = getBalanceFactor(root);

    if (balance > 1 && getBalanceFactor(root->left) >= 0)
        return rightRotate(root);

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

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

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

    return root;
}

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

```
    if (keyword < root->keyword)
        return search(root->left, keyword, comparisons);

    return search(root->right, keyword, comparisons);
}

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

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

int main() {
    Node* root = nullptr;
    root = insert(root, "Apple", "A fruit");
    root = insert(root, "Ball", "A round object");
    root = insert(root, "Cat", "A small pet animal");
    root = insert(root, "Dog", "A loyal pet");

    cout << "Dictionary in Ascending Order:\n";
```

```
    inorder(root);

    cout << "\nDictionary in Descending Order:\n";
    reverseInorder(root);

    int comparisons = 0;
    string searchKey = "Ball";
    Node* found = search(root, searchKey, comparisons);

    if (found)
        cout << "\nFound: " << found->keyword << " -> " << found->meaning << "
(Comparisons: " << comparisons << ")\n";
    else
        cout << "\nKeyword not found!\n";

    root = deleteNode(root, "Ball");
    cout << "\nAfter Deleting 'Ball':\n";
    inorder(root);

    return 0;
}
```

Output:

```
Dictionary in Ascending Order:
Apple : A fruit
Ball : A round object
Cat : A small pet animal
Dog : A loyal pet
Dictionary in Descending Order:
Dog : A loyal pet
Cat : A small pet animal
Ball : A round object
```

Apple : A fruit

Found: Ball -> A round object (Comparisons: 1)

After Deleting 'Ball':

Apple : A fruit

Cat : A small pet animal

Dog : A loyal pet