/\*

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;

};

int height(Node\* n) {

return n ? n->height : 0;

}

int getBalance(Node\* n) {

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

}

int max(int a, int b) {

return (a > b) ? a : b;

}

Node\* createNode(string keyword, string meaning) {

Node\* node = new Node();

node->keyword = keyword;

node->meaning = meaning;

node->left = node->right = nullptr;

node->height = 1;

return node;

}

// Right Rotation

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;

}

// Left Rotation

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;ff

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

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

return y;

}

// Insert node

Node\* insert(Node\* root, string key, string meaning) {

if (!root)

return createNode(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 {

cout << "Duplicate keyword not allowed.\n";

return root;

}

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

int balance = getBalance(root);

// Balancing cases

if (balance > 1 && key < root->left->keyword)// LL Case

return rightRotate(root);

if (balance < -1 && key > root->right->keyword)// RR Case

return leftRotate(root);

if (balance > 1 && key > root->left->keyword) {// LR Case

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && key < root->right->keyword) {// RL Case

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

// Find min value node

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current && current->left)

current = current->left;

return current;

}

// Delete node

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 {

// Node with one or no child

if (!root->left) {

Node\* temp = root->right;

delete root;

return temp;

}

if (!root->right) {

Node\* temp = root->left;

delete root;

return temp;

}

Node\* temp = minValueNode(root->right);

root->keyword = temp->keyword;

root->meaning = temp->meaning;

root->right = deleteNode(root->right, temp->keyword);

}

if (!root) return root;

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

int balance = getBalance(root);

// Balancing

if (balance > 1 && getBalance(root->left) >= 0)//LL Case

return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {//LR Case

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0)//RR Case

return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {// RL Case

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

// Update meaning of a keyword

void update(Node\* root, string key, string newMeaning) {

if (!root) {

cout << "Keyword not found.\n";

return;

}

if (key < root->keyword)

update(root->left, key, newMeaning);

else if (key > root->keyword)

update(root->right, key, newMeaning);

else {

root->meaning = newMeaning;

cout << "Meaning updated successfully.\n";

}

}

// Display in ascending order

void inorder(Node\* root) {

if (!root) return;

inorder(root->left);

cout << root->keyword << " : " << root->meaning << endl;

inorder(root->right);

}

// Display in descending order

void reverseInorder(Node\* root) {

if (!root) return;

reverseInorder(root->right);

cout << root->keyword << " : " << root->meaning << endl;

reverseInorder(root->left);

}

// Search keyword and count comparisons

int search(Node\* root, string key, int& comparisons) {

if (!root) return 0;

comparisons++;

if (key == root->keyword)

return 1;

else if (key < root->keyword)

return search(root->left, key, comparisons);

else

return search(root->right, key, comparisons);

}

int main() {

Node\* root = nullptr;

int choice;

string keyword, meaning;

do {

cout << "\n--- Dictionary Menu ---\n";

cout << "1. Add Keyword\n2. Delete Keyword\n3. Update Meaning\n";

cout << "4. Display Ascending\n5. Display Descending\n";

cout << "6. Search Keyword\n7. Exit\nEnter choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter keyword: "; cin >> keyword;

cout << "Enter meaning: "; cin.ignore(); getline(cin, meaning);

root = insert(root, keyword, meaning);

break;

case 2:

cout << "Enter keyword to delete: "; cin >> keyword;

root = deleteNode(root, keyword);

break;

case 3:

cout << "Enter keyword to update: "; cin >> keyword;

cout << "Enter new meaning: "; cin.ignore(); getline(cin, meaning);

update(root, keyword, meaning);

break;

case 4:

cout << "--- Ascending Order ---\n";

inorder(root);

break;

case 5:

cout << "--- Descending Order ---\n";

reverseInorder(root);

break;

case 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 after " << comparisons << " comparisons.\n”;

break;

case 7:

cout<<”exiting the program”<<endl;

}

}while(choice!=7);

Return 0;

}

/\*

Balancing in deletion

| **Case** | **Condition** | **Insertion Position** | **Rotation Needed** |
| --- | --- | --- | --- |
| LL | balance > 1 && key < root->left->keyword | Left of Left Child | Right Rotation |
| RR | balance < -1 && key > root->right->keyword | Right of Right Child | Left Rotation |
| LR | balance > 1 && key > root->left->keyword | Right of Left Child | Left + Right Rotation |
| RL | balance < -1 && key < root->right->keyword | Left of Right Child | Right + Left Rotation |

Balancing in deletion

| **Case** | **Condition** | **Meaning** | **Rotation** |
| --- | --- | --- | --- |
| LL | balance > 1 && getBalance(root->left) >= 0 | Left-heavy and left child also left-heavy | Right Rotate |
| LR | balance > 1 && getBalance(root->left) < 0 | Left-heavy and left child right-heavy | Left + Right |
| RR | balance < -1 && getBalance(root->right) <= 0 | Right-heavy and right child also right-heavy | Left Rotate |
| RL | balance < -1 && getBalance(root->right) > 0 | Right-heavy and right child left-heavy | Right + Left |

\*/