Word Autocomplete Using AVL Tree

Introduction

This task aims to develop a word autocomplete program that suggests dictionary words based on a prefix typed by the user. The program uses an AVL tree to store words efficiently, ensuring that both insertions and prefix searches remain fast even for large datasets.

Implementation Explanation

1. Data Structure

Each word from the dictionary is stored in a node of an AVL tree. Each node contains:

- The string key
- Pointers to left and right child nodes
- The node height used to maintain AVL balance

2. Insertion Function

When each word is read from the dictionary file, it is inserted into the AVL tree using the insert() function. After each insertion, the program computes the balance factor and applies rotations if needed to maintain tree balance.

3. Prefix Search Function

The findPrefix() function traverses the AVL tree and collects all words that start with the prefix entered by the user. It uses rfind(prefix, 0) to verify whether a word begins with the prefix and avoids searching down branches that cannot contain matching words.

4. Main Program Flow

1. Reads words from dictionary.txt
2. Inserts each word into the AVL tree
3. Prompts the user to enter a prefix
4. Displays all matching suggestions
5. Repeats until the user types 'exit'

Program Code

Below is the implementation of the autocomplete program:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>
#include <filesystem>
using namespace std;
```

```
struct Node {
    string key;
    Node* left;
    Node* right;
    int hight;
    Node(string o)
        : key(o), left(nullptr), right(nullptr), hight(1) {}
};

int height_node(Node* n) {
    return (n != nullptr ? n->hight : 0);
}

int balance(Node* n) {
    return (n != nullptr ? height_node(n->left) - height_node(n->right) : 0);
}

Node* right_rotate(Node* y) {
    Node* x = y->left;
    Node* w = x->right;
    x->right = y;
    y->left = w;
    y->hight = 1 + max(height_node(y->left), height_node(y->right));
    x->hight = 1 + max(height_node(x->left), height_node(x->right));
    return x;
}

Node* Left_rotate(Node* x) {
    Node* y = x->right;
    Node* w = y->left;
    y->left = x;
    x->right = w;
    x->hight = 1 + max(height_node(x->left), height_node(x->right));
    y->hight = 1 + max(height_node(y->left), height_node(y->right));
    return y;
}

Node* insert(Node* node, string key) {
    if (node == nullptr)
        return new Node(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;

    node->hight = 1 + max(height_node(node->left), height_node(node->right));
    int bal = balance(node);

    if (bal > 1 && key < node->left->key)
        return right_rotate(node);
```

```
    if (bal < -1 && key > node->right->key)
        return Left_rotate(node);
    if (bal > 1 && key > node->left->key) {
        node->left = Left_rotate(node->left);
        return right_rotate(node);
    }
    if (bal < -1 && key < node->right->key) {
        node->right = right_rotate(node->right);
        return Left_rotate(node);
    }

    return node;
}

void findPrefix(Node* node, const string& prefix, vector<string>& results) {
    if (node == nullptr)
        return;

    if (node->key < prefix) {
        findPrefix(node->right, prefix, results);
    }
    else if (node->key.rfind(prefix, 0) == 0) {
        results.push_back(node->key);
        findPrefix(node->left, prefix, results);
        findPrefix(node->right, prefix, results);
    }
    else {
        findPrefix(node->left, prefix, results);
    }
}

int main() {
    Node* root = nullptr;

    ifstream file("dictionary.txt");
    if (!file) {
        cout << "Dictionary file not found!" << endl;
        return 1;
    }

    string word;
    while (file >> word) {
        transform(word.begin(), word.end(), word.begin(), ::tolower);
        root = insert(root, word);
    }
    file.close();

    cout << "Dictionary loaded successfully.\n";

    while (true) {
        cout << "\nsearch: ";
        string prefix;
        cin >> prefix;
```

```
        if (prefix == "exit")
            break;

        transform(prefix.begin(), prefix.end(), prefix.begin(), ::tolower);

        vector<string> results;
        findPrefix(root, prefix, results);

        if (results.empty()) {
            cout << "No suggestions found for '" << prefix << "'.\n";
        }
        else {
            cout << "Suggestions for '" << prefix << "':\n";
       for (int i = 0; i < results.size(); i++) {

              cout << " - " << results[i] << endl;

       }
        }
    }

    return 0;
}
```

Complexity Analysis

Operation	**Complexity**
Insertion	O(log n)
Search	O(k + m), where k ≈ log n and m is the number of matches
Reason	AVL balancing ensures minimal height and efficient searching