**Ain Shams University**
**Faculty of Engineering**
**Department of Computer and Artificial Intelligence Engineering**

---

**Data Structures and Problem Solving**

**Course Code: CSE243**

---

**Submitted by:**

**Karim Samer Mostafa 23p0439**

**Youssef Atef Elnaggar 23p0341**

**Omar Gamil Anwar 23p0438**

**Ahmed Mohamed Elian 23p0446**

**Paula Hanna Naguib 23P0440**

---

**Submitted on:**
24/12/2024

---

**Instructor:**

**Dr. Ashraf Ahmed Abdel Raouf Hamdi**

---

# Table of Contents :

## 1. Introduction

The Text Editor Program serves as a simple, yet powerful text manipulation tool that integrates spell checking, word suggestions, and text editing functionalities. Its core innovation lies in the use of the Trie data structure to efficiently manage a dictionary of words, enabling quick spell-checking and auto-suggestion of misspelled words. The dictionary is stored in an external text file, english2.txt, which is loaded at the start of the program. The program also provides essential text editing features such as find-and-replace, undo, and redo functionalities. This report provides an in-depth analysis of the design and functionality of the program, focusing on the underlying Trie data structure, its interaction with the external dictionary file, and the overall flow of the text editing operations. The following sections will explore the individual components of the program, explaining how they contribute to the text editor's overall behavior and efficiency.

---

## 2. Program Structure and Design

The Text Editor Program is structured around three core components:

- **TrieNode**: The building block of the Trie, responsible for storing individual characters and linking to subsequent characters in the tree.
- Trie: The data structure responsible for managing the dictionary and providing functionalities such as inserting words, searching for words, and offering suggestions.
- **TextEditor**: The class that handles the text manipulation operations and user interactions, including spell checking, word suggestions, find-and-replace, and undo/redo functionalities.

Each of these components is designed to perform specific tasks, with clear boundaries between them. This modular design ensures that the program is both maintainable and scalable. Below, we delve into the specifics of each component and its responsibilities.

---

## 3. Trie Data Structure

The Trie is a tree-like data structure that is particularly effective for handling operations on strings, such as word lookup, prefix search, and auto-suggestions. In the context of this program, the Trie is used to store the dictionary of valid words and perform spell-checking operations.

### 3.1. TrieNode :

Each TrieNode represents a character in a word. It contains two main attributes:

- isEnd: A boolean flag indicating whether the node marks the end of a valid word in the dictionary.
- children: A vector of pointers to child TrieNodes. Each node has 26 children (one for each letter of the alphabet).

The TrieNode is fundamental to building the Trie structure. Every node represents a possible character in a word, and together, they form a complete word when traversed from the root to a leaf node. The isEnd flag allows the program to differentiate between partial words (prefixes) and complete words.

### 3.2. Trie Operations :

The Trie class encapsulates the logic for managing the dictionary and offers several key functionalities:

- Insert: The insert method takes a word and adds it to the Trie by traversing the tree and creating new nodes where necessary. This operation is efficient, with a time complexity of O(m), where m is the length of the word being inserted.
- Search: The search method checks if a given word exists in the Trie. It follows the path of nodes corresponding to each character in the word. If all characters are found and the isEnd flag is true at the last node, the word exists in the dictionary.
- Suggestions: The getSuggestions method is used to find words that start with the same prefix. When the user types part of a word, the Trie can efficiently return a list of words that match the prefix, offering word suggestions. This feature is particularly useful when dealing with incomplete or misspelled words.

3.3. Advantages of Using a Trie

The Trie structure offers several advantages:

- Efficient Word Lookup: Searching for a word or prefix in a Trie takes O(m) time, where m is the length of the word. This makes it faster than other data structures like arrays or linked lists.
- Prefix Matching: Tries are ideal for matching prefixes, making them perfect for suggesting words as the user types.

- Scalability: The Trie can handle large dictionaries efficiently, and its structure naturally supports dynamic updates, such as adding new words.

---

## 4. Text Editor Features

The TextEditor class is responsible for handling the core text manipulation features, including spell checking, word suggestions, and editing functionalities. Here, we'll explore each of the main features in detail.

### 4.1. Spell Checking

One of the most important features of the program is spell checking. When the user enters text, the program checks each word against the dictionary stored in the Trie. If a word is found to be invalid (not present in the dictionary), the program flags it as a misspelled word and provides suggestions based on the prefix of the word.

To accomplish this, the program leverages the getSuggestions method of the Trie class, which returns words that share a common prefix with the misspelled word. For instance, if the user types "helo," the program may suggest "hello" as a correction.

### 4.2. Word Suggestions

The word suggestion feature enhances the user experience by providing alternatives to misspelled words. If the program identifies a misspelled word, it will suggest corrections by matching prefixes between the input word and dictionary words.

The suggestions are displayed to the user, and they are given the option to add the word to the dictionary if it's not already present. This feature improves the usability of the text editor, especially for users who may be working with specialized or uncommon words.

### 4.3. Dictionary Management

The dictionary of valid words is stored in an external text file, english2.txt, which contains one word per line. When the program starts, it reads the file and loads the words into the Trie. This approach ensures that the dictionary persists between sessions, so users don't have to re-enter words every time they run the program.

The program allows users to add new words to the dictionary. If a word is flagged as misspelled, the user can choose to add it to the dictionary, ensuring that the word is available for future use.

### 4.4. Text Manipulation Features

In addition to spell checking and word suggestions, the TextEditor class provides several basic text manipulation features:

- Find and Replace: This feature allows users to search for a specific word in the text and replace it with a new word. The program performs this operation throughout the entire text, ensuring that all instances of the word are replaced.
- Undo and Redo: The undo and redo functionalities allow users to revert or reapply changes to the text. The program keeps a history of the text, enabling users to move backward and forward through their modifications.
- Displaying Current Text: The user can display the current text to review the changes made, providing a simple way to track progress.

---

## 5. Program Flow and User Interaction

The program interacts with the user through a simple command-line interface. Upon launching the program, the user is presented with a menu of options. The user can choose one of the following actions:

1. Check Spelling: The user can input a block of text, and the program will check each word for spelling errors, suggesting corrections where necessary.
2. Find and Replace: The user can specify a word to search for and a word to replace it with throughout the text.
3. Display Current Text: The user can view the current state of the text being edited.
4. Undo: This option allows the user to undo the most recent change made to the text.
5. Redo: If an action has been undone, the user can redo the action to restore the previous state.
6. Exit: This option exits the program.

   The program uses standard input and output streams to interact with the user, making it easy to operate via the command line.

---

## 6. Error Handling and Limitations

The program includes basic error handling, especially for file operations. For instance, if the dictionary file (english2.txt) is missing or cannot be accessed, the program will notify the user and proceed with an empty dictionary.

However, there are several limitations that could be addressed in future versions:

- File Integrity: The program currently only appends new words to the dictionary file. It could benefit from additional checks to ensure the integrity and sorting of the dictionary.
- Handling Large Dictionaries: While the Trie is efficient, larger dictionaries could pose challenges in terms of memory usage. Optimizations could be made to compress the Trie or use more memory-efficient data structures.
- User Interface: A graphical user interface (GUI) would improve usability, especially for less technical users. The current command-line interface is functional but could be made more intuitive with a GUI.

---

## 7. Conclusion

The Text Editor Program offers a robust solution for basic text editing with advanced spell-checking and word suggestion features. By utilizing a Trie data structure, the program ensures efficient dictionary management and fast lookups, making it an ideal tool for users who need to edit text while ensuring correct spelling. The interaction with the external dictionary file provides persistence, and the program's design allows for easy extensibility and potential future improvements.

In conclusion, this program serves as a practical and efficient tool for text editing, spelling correction, and dictionary management, with plenty of room for enhancements to improve its functionality and user experience.

# 8. APPENDIX

## 1) TrieNode.h

```
#ifndef TRIENODE_H
#define TRIENODE_H

#include <vector>
#include <string>

using namespace std;

struct TrieNode {
    bool isEnd;
    vector<TrieNode*> children;

    TrieNode();
};

#endif
```

## 2) TrieNode.cpp

```
#include "TrieNode.h"

TrieNode::TrieNode() : isEnd(false), children(26, nullptr) {}
```

**3)Trie.h**

```
#ifndef TRIE_H
#define TRIE_H

#include "TrieNode.h"
#include <vector>
#include <string>

using namespace std;

class Trie {
private:
    TrieNode* root;

public:
    Trie();
    void insert(const string& word);
    bool search(const string& word);
    vector<string> getSuggestions(const string& prefix, int
limit);

    TrieNode* getRoot() const {
        return root;
    }

private:
    void gatherSuggestions(TrieNode* node, const string&
prefix, vector<string>& suggestions, int& limit);
};

#endif
```

# 4)Trie.cpp

```
#include "Trie.h"
#include "TrieNode.h"
#include <vector>
#include <string>
#include <iostream>

using namespace std;

Trie::Trie() {
    root = new TrieNode();
}

void Trie::insert(const string& word) {
    TrieNode* node = root;
    for (char ch : word) {
        int index = ch - 'a';
        if (node->children[index] == nullptr) {
            node->children[index] = new TrieNode();
        }
        node = node->children[index];
    }
    node->isEnd = true;
}

bool Trie::search(const string& word) {
    TrieNode* node = root;
    for (char ch : word) {
        int index = ch - 'a';
        if (node->children[index] == nullptr) {
            return false;
        }
        node = node->children[index];
    }
    return node->isEnd;
}

vector<string> Trie::getSuggestions(const string& prefix, int limit) {
    vector<string> suggestions;
    TrieNode* node = root;

    for (char ch : prefix) {
        int index = ch - 'a';
        if (node->children[index] == nullptr) {
            return suggestions;
        }
```

```
            node = node->children[index];
        }

        gatherSuggestions(node, prefix, suggestions, limit);

        return suggestions;
    }

    void Trie::gatherSuggestions(TrieNode* node, const string& prefix, vector<string>&
    suggestions, int& limit) {
        if (node->isEnd) {
            suggestions.push_back(prefix);
            limit++;
        }

        for (int i = 0; i < 26; i++) {
            if (node->children[i] != nullptr) {
                gatherSuggestions(node->children[i], prefix + char(i + 'a'), suggestions, limit);
            }
        }
    }
```

# 5)TextEditor.h

```
    #ifndef TEXTEDITOR_H
    #define TEXTEDITOR_H

    #include <string>
    #include <set>
    #include <vector>
    using namespace std;

    class TextEditor {
    public:
        TextEditor();

        string toLowerCase(const string& str);

        void suggestWord(const string& word);

        void addWordToDictionary(const string& word);

        void checkSpelling(const string& text);

        void displayCurrentText();

        void findAndReplace(const string& oldWord, const string& newWord);
```

```
    void undo();

    void redo();

private:
    string currentText;
    set<string> dictionary;
    vector<string> history;
    vector<string> redoHistory;
};

#endif
```

# 6)TextEditor.cpp

```
#include "TextEditor.h"
#include <algorithm>
#include <sstream>
#include <iostream>
#include <fstream>

using namespace std;

TextEditor::TextEditor() {
    ifstream inFile("english2.txt");
    if (inFile) {
        string word;
        while (getline(inFile, word)) {
            dictionary.insert(toLowerCase(word));
        }
        inFile.close();
    }
    else {
        cout << "Dictionary file not found. Starting with an empty dictionary.\n";
    }
}

string TextEditor::toLowerCase(const string& str) {
    string lowerStr = str;
    transform(lowerStr.begin(), lowerStr.end(), lowerStr.begin(), ::tolower);
    return lowerStr;
}

void TextEditor::suggestWord(const string& word) {
    string lowerWord = toLowerCase(word);
    vector<string> suggestions;

    for (size_t prefixLength = 2; prefixLength <= lowerWord.size(); ++prefixLength) {
        for (const auto& dictWord : dictionary) {
```

```
            if (lowerWord.substr(0, prefixLength) == dictWord.substr(0, prefixLength)) {
                suggestions.push_back(dictWord);
            }
        }

        if (!suggestions.empty()) {
            break;
        }
    }

    if (!suggestions.empty()) {
        cout << "Suggestions for '" << word << "':\n";
        for (const auto& suggestion : suggestions) {
            cout << "- " << suggestion << "\n";
        }
    }
    else {
        cout << "No suggestions available for '" << word << "'.\n";
    }

    cout << "Would you like to add this word to the dictionary? (y/n): ";
    char choice;
    cin >> choice;
    if (choice == 'y' || choice == 'Y') {
        addWordToDictionary(word);
    }
}

void TextEditor::addWordToDictionary(const string& word) {
    dictionary.insert(word);
    ofstream outFile("english2.txt", ios::app);
    if (outFile) {
        outFile << word << "\n";
        outFile.close();
    }
    else {
        cout << "Error saving dictionary to file.\n";
    }
    cout << "'" << word << "' has been added to the dictionary.\n";
}

void TextEditor::checkSpelling(const string& text) {
    stringstream ss(text);
    string word;
    bool misspelled = false;

    history.push_back(currentText);

    while (ss >> word) {
        string lowerWord = toLowerCase(word);

        if (dictionary.find(lowerWord) == dictionary.end()) {
```

```
            cout << "Misspelled word: " << word << "\n";
            suggestWord(word);
            misspelled = true;
        }
    }

    if (!misspelled) {
        cout << "No spelling mistakes found.\n";
    }

    currentText = text;
}

void TextEditor::displayCurrentText() {
    cout << "Current text: " << currentText << "\n";
}

void TextEditor::findAndReplace(const string& oldWord, const string& newWord) {
    history.push_back(currentText);

    size_t pos = 0;
    while ((pos = currentText.find(oldWord, pos)) != string::npos) {
        currentText.replace(pos, oldWord.length(), newWord);
        pos += newWord.length();
    }

    redoHistory.clear();

    displayCurrentText();
}

void TextEditor::undo() {
    if (!history.empty()) {
        redoHistory.push_back(currentText);
        currentText = history.back();
        history.pop_back();

        cout << "Undo successful! Current text: " << currentText << "\n";
    }
    else {
        cout << "No changes to undo.\n";
    }
}

void TextEditor::redo() {
    if (!redoHistory.empty()) {
        history.push_back(currentText);
        currentText = redoHistory.back();
        redoHistory.pop_back();

        cout << "Redo successful! Current text: " << currentText << "\n";
    }
```

```
    else {
        cout << "No changes to redo.\n";
    }
}
```

# 7)project.cpp

```
#include <iostream>
#include "TextEditor.h"

using namespace std;

int main() {
    TextEditor editor;

    int choice;
    string text, findWord, replaceWord;

    while (true) {
        cout << "\nText Editor Menu:\n";
        cout << "1. Check Spelling\n";
        cout << "2. Find and Replace\n";
        cout << "3. Display Current Text\n";
        cout << "4. Undo\n";
        cout << "5. Redo\n";
        cout << "6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter text to check spelling: ";
            cin.ignore();
            getline(cin, text);
            editor.checkSpelling(text);
            break;
        case 2:
            cout << "Enter word to find: ";
            cin >> findWord;
            cout << "Enter word to replace with: ";
            cin >> replaceWord;
            editor.findAndReplace(findWord, replaceWord);
            break;
        case 3:
```

```
            editor.displayCurrentText();
            break;
        case 4:
            editor.undo();
            break;
        case 5:
            editor.redo();
            break;
        case 6:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Invalid choice. Try again.\n";
        }
    }

    return 0;
}
```