Project Report

SearchSift

Kabir Yadav Shrey Goel

E22CSEU0255 E22CSEU0262

A report submitted in part fulfillment of the degree of BTech in Computer Science

Supervisor: Dr. Priyanka

Department of Computer Science

Project Report

Abstract:

This project involves the development of a C++ program that performs keyword search in multiple text files within a specified directory. The program tokenizes the content of each file, counts the occurrences of a user-specified keyword, ranks the files based on occurrences, and allows the user to open a selected file. This report outlines the implementation details, challenges faced, and the overall project outcome.

Table of Contents:

Section 1: Introduction

1.1 Background

In the contemporary era, vast amounts of textual data are generated daily across various domains, including business, research, and personal activities. Managing and extracting meaningful information from this abundance of textual content pose significant challenges. As a result, efficient methods for searching and analyzing text files have become imperative.

The motivation behind the project stems from the need to develop a tool that can facilitate effective keyword searches within a collection of text files. This need arises from the following considerations:

1. **Information Retrieval**: As users accumulate a large number of text documents, the ability to quickly locate specific information within these files becomes crucial. Traditional methods of manual searching are time-consuming and impractical when dealing with extensive datasets.

2. **Data Organization**: Many users store textual information across multiple files or directories, making it challenging to locate relevant content efficiently. A tool that can systematically search for keywords across these files aids in organizing and accessing data.

3. **Research and Analysis**: Researchers and analysts often work with extensive datasets comprising textual information. A tool capable of identifying occurrences of specific keywords can streamline the process of extracting meaningful insights from these datasets.

1.2 Objectives

The primary objectives of this project are as follows:

- **Efficient Keyword Search**: Develop a C++ program capable of efficiently searching for user-specified keywords within a directory containing multiple text files.

- **Tokenization and Analysis**: Implement a tokenization process to break down the content of each file into individual words, allowing for a granular analysis of the textual data.

- **Ranking and User Interaction**: Rank the files based on the occurrences of the specified keyword and provide a user-friendly interface for selecting and opening the most relevant files.

- **Usability and Accessibility**: Create a tool that is easy to use, even for users with minimal programming knowledge, and ensure compatibility with standard file systems.

This project aims to address the aforementioned challenges by providing a practical solution for efficient keyword searches in text files, promoting better information retrieval, data organization, and facilitating research and analysis processes.

Section 2: Problem Definition & Objectives

2.1 Problem Statement

The problem at hand revolves around the challenge of efficiently searching for specific keywords within a diverse collection of text files stored in a directory. As the volume of textual data continues to grow across various domains, users face difficulties in quickly locating relevant information. Manual methods of searching prove to be cumbersome and time-consuming, especially when dealing with a large number of text documents scattered across directories.

The specific issues addressed by this project include:

- **Ineffective Manual Search**: Traditional manual searches within text files are inefficient, especially when dealing with extensive datasets. Users require a more streamlined and automated approach to locate information.

- **Lack of Organization**: With textual information distributed across multiple files, users often struggle with organizing and accessing relevant content. There is a need for a tool that can systematically search and organize data based on user-specified keywords.

- **Time-Consuming Analysis**: Researchers and analysts working with substantial textual datasets encounter challenges in extracting meaningful insights. The absence of a tool capable of quickly identifying occurrences of specific keywords hinders the efficiency of research and analysis processes.

2.2 Project Objectives

To address the identified problem, the project aims to achieve the following specific objectives:

1. **Efficient Keyword Search**:

   - Develop a C++ program that can efficiently search for user-specified keywords within a given directory.

2. **Tokenization and Analysis**:

   - Implement a tokenization process to break down the content of each text file into individual words, enabling a detailed analysis of the textual data.

3. **Ranking and User Interaction**:

   - Establish a ranking mechanism for files based on the occurrences of the specified keyword, providing a user-friendly interface for selecting and opening relevant files.

4. **Usability and Accessibility**:

   - Create a user-friendly tool that is accessible to individuals with varying levels of programming knowledge, ensuring compatibility with standard file systems.

5. **Optimized Performance**:

   - Design the program to be resource-efficient and capable of handling large datasets without compromising performance.

By achieving these objectives, the project endeavors to offer a practical solution that enhances the efficiency of keyword searches in text files, ultimately addressing the challenges associated with information retrieval, data organization, and research and analysis processes.

Section 3: Proposed Work/Methodology

3.1 Tokenization Process

The tokenization process plays a pivotal role in breaking down the content of each text file into individual words, facilitating subsequent analysis. The process is implemented as follows:

- **Input**: The content of a text file is read as a string.

- **Tokenization**: The string is processed using an `istringstream` to extract individual words. A `Token` structure is utilized to store each word along with a pointer to the next token.

- **Data Structure**:

```
struct Token {
    string word;
    Token* next;
};
```

- **Algorithm**:

```
Token* tokenizeString(const string& content) {
    Token* tokens = nullptr;
    istringstream iss(content);
    string word;

    while (iss >> word) {
        Token* token = new Token{word, tokens};
        tokens = token;
    }

    return tokens;
}
```

3.2 File Reading and Tokenization

The program iterates through each text file in the specified directory, reads its content, and tokenizes it using the aforementioned process. The key steps include:

- **Directory Iteration**: The program utilizes the **`<filesystem>`** library to iterate through each file in the specified directory.

- **File Reading**: For each regular file encountered, the program opens and reads its content using an `ifstream`.

- **Tokenization**: The tokenization process is applied to the file's content, and the resulting tokens are stored along with the filename in a `FileContent` structure.

3.3 Keyword Search Algorithm

The keyword search algorithm involves traversing the list of tokens for each file and counting occurrences of the user-specified keyword. The steps include:

- **User Input**: The program prompts the user to input a keyword for the search.

- **Search Algorithm**:

```
for (FileContent& fileContent : fileContents) {
    Token* currentToken = fileContent.tokens;
    while (currentToken) {
        if (currentToken->word == keyword) {
            fileContent.occurrences++;
        }
        currentToken = currentToken->next;
    }
}
```

3.4 **File Ranking**

After counting occurrences, the program filters files with at least one occurrence, ranks them based on occurrences, and presents the results to the user. The ranking process involves:

- **Filtering Files**:

```
for (const FileContent& result : fileContents) {
    if (result.occurrences >= 1) {
        filteredFiles.push_back(result);
    }
}
```

- **Sorting Files**:

```
sort(
    filteredFiles.begin(),
    filteredFiles.end(),
    [](const FileContent& a, const FileContent& b) {
        return a.occurrences > b.occurrences;
    }
);
```

The resulting ranked list is then presented to the user, allowing them to select and open the most relevant file based on the keyword occurrences.

This methodology ensures an organized and efficient approach to tokenization, file reading, keyword search, and file ranking within the scope of the project.

Section 4: Data Structure Used

4.1 Token Structure

The `Token` structure serves as a fundamental building block for the program's tokenization process. It encapsulates the information related to each token, specifically:

- **Attributes**:

  - `**word**`: A string representing an individual word/token.

  - `**next**`: A pointer to the next token in the list.

- **Definition**:

```
struct Token {
    string word;
    Token* next;
};
```

- **Explanation**:

  - The `**word**` attribute holds the actual content of the token, representing an individual word extracted during the tokenization process.

  - The `**next**` attribute is a pointer to the next token in the linked list, facilitating the construction of a linked list of tokens for a given text file.

This structure is instrumental in forming a linked list of tokens for each text file, allowing for efficient storage and retrieval during subsequent stages of the program's execution.

4.2 FileContent Structure

The `FileContent` structure is designed to encapsulate information about each text file processed by the program. It stores details such as the file name, a linked list of tokens, and the count of occurrences of a specified keyword. Key attributes include:

- **Attributes**:

  - `**fileName**`: A string representing the name of the text file.

  - `**tokens**`: A pointer to the linked list of tokens obtained from tokenizing the file's content.

  - `**occurrences**`: An integer representing the count of occurrences of the specified keyword in the file.

- **Definition**:

```
struct FileContent {
    string fileName;
    Token* tokens;
    int occurrences;
};
```

- **Explanation**:

  - The `**fileName**` attribute stores the name of the text file, allowing for easy identification and retrieval.

  - The `**tokens**` attribute is a pointer to the linked list of tokens associated with the file, enabling subsequent analysis and processing.

  - The `**occurrences**` attribute represents the count of occurrences of the specified keyword within the file, which is crucial for ranking files based on relevance.

This structure facilitates the organization and storage of essential information about each processed text file, enabling efficient keyword search, file ranking, and user interaction within the program.

Section 5: Language and Tools

5.1 C++ Programming Language

Justification for Choosing C++:

The selection of the C++ programming language for this project is driven by several key considerations:

- **Performance**: C++ is renowned for its high-performance capabilities, making it well-suited for projects that involve resource-intensive tasks, such as text processing and file manipulation. This ensures that the program can efficiently handle large datasets.

- **Memory Management**: C++ provides explicit control over memory management, allowing for efficient allocation and deallocation of resources. This level of control is particularly beneficial for projects dealing with data structures and dynamic memory allocation, as is the case in this keyword search application.

- **Standard Template Library (STL):** The C++ STL offers a rich set of pre-built classes and functions that expedite the implementation of complex algorithms. Leveraging the STL can enhance the efficiency and readability of the code.

- **Cross-Platform Compatibility:** C++ is known for its portability across different platforms. This characteristic ensures that the program can be executed seamlessly on various operating systems without major modifications.

- **Mature Ecosystem**: C++ boasts a mature and well-established ecosystem with extensive documentation, a vibrant community, and a plethora of libraries. This contributes to the ease of development and troubleshooting.

- **Object-Oriented Paradigm**: The object-oriented features of C++ facilitate the creation of modular and scalable code structures. This is advantageous when dealing with multiple components and functionalities, as is the case in this keyword search project.

In summary, the choice of C++ aligns with the project's requirements for performance, memory management, cross-platform compatibility, and the availability of robust libraries.

5.2 Libraries Used

External Libraries:

The project utilizes the following external libraries to enhance its functionality:

- **Filesystem Library (`<filesystem>`):**

  - This library, introduced in C++17, provides convenient facilities for performing operations on file systems, including iterating through directories and checking file types. It simplifies the process of reading text files from a specified directory.

5.3 Development Environment

Integrated Development Environment (IDE):

The development of this C++ project was undertaken using a popular integrated development environment:

Section 6: Source Code

Github link: SHR3YGO3L/cpp_project_directory_search (github.com)

Section 7: Results

7.1 Sample Output

```
Enter a keyword to search: analysis
Rank 1: file1.txt (Occurrences: 3 times)
Rank 2: file3.txt (Occurrences: 2 times)
Rank 3: file2.txt (Occurrences: 1 times)

Enter the rank of the file you want to open: 1
Opening file: file1.txt
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Rank 78: Data1002.txt (Occurrences: 1 times)
Rank 79: Data978.txt (Occurrences: 1 times)
Rank 80: Data949.txt (Occurrences: 1 times)
Rank 81: Data930.txt (Occurrences: 1 times)
Rank 82: Data923.txt (Occurrences: 1 times)
Rank 83: Data906.txt (Occurrences: 1 times)
Rank 84: Group03_06.txt (Occurrences: 1 times)
Rank 85: Data880.txt (Occurrences: 1 times)
Rank 86: Data871.txt (Occurrences: 1 times)
Rank 87: Data868.txt (Occurrences: 1 times)
Rank 88: Data1574.txt (Occurrences: 1 times)
Rank 89: Data1461.txt (Occurrences: 1 times)
Rank 90: Data1423.txt (Occurrences: 1 times)
Rank 91: Data1358.txt (Occurrences: 1 times)
Rank 92: Data1337.txt (Occurrences: 1 times)
Enter the rank of the file you want to open: 12
Opening file: Data882.txt
```

```
Data882
File  Edit  View

Computational Complexity: Section 1
In this article I'll try to introduce you to the area of computation complexity. The article will be a bit
long before we get to the actual formal definitions because I feel that the rationale behind these
definitions needs to be explained as well - and that understanding the rationale is even more important
than the definitions alone.

Why is it important?

Example 1. Suppose you were assigned to write a program to process some records your company receives from
time to time. You implemented two different algorithms and tested them on several sets of test data. The
processing times you obtained are in Table 1.

# of records10205010010005000
algorithm 10.00s0.01s0.05s0.47s23.92s47min
algorithm 20.05s00.05s000.06s0.11s0.78s14.22s
Table 1. Runtimes of two fictional algorithms.
In praxis, we probably could tell which of the two implementations is better for us (as we usually can
estimate the amount of data we will have to process). For the company this solution may be fine. But from
the programmer's point of view, it would be much better if he could estimate the values in Table 1 before
writing the actual code - then he could only implement the better algorithm.

The same situation occurs during programming challenges: The size of the input data is given in the problem
statement. Suppose I found an algorithm. Questions I have to answer before I start to type should be: Is my
algorithm worth implementing? Will it solve the largest test cases in time? If I know more algorithms
solving the problem, which of them shall I implement?

This leads us to the question: How to compare algorithms? Before we answer this question in general, let's

Ln 1, Col 1  100%  Windows (CRLF)  UTF-8
```

Section 8: Conclusion

8.1 Summary of Achievements

The project has successfully addressed the need for an efficient keyword search tool in a collection of text files. Key achievements include:

- **Efficient Keyword Search**: The C++ program efficiently searches for user-specified keywords within a specified directory, promoting quick and accurate information retrieval.

- **Tokenization and Analysis**: The tokenization process effectively breaks down the content of each text file into individual words, enabling detailed analysis of textual data.

- **File Ranking and User Interaction**: The program ranks files based on keyword occurrences, providing a user-friendly interface for selecting and opening the most relevant files.

- **Usability and Accessibility**: A user-friendly tool has been created, ensuring accessibility for users with varying levels of programming knowledge. The program is compatible with standard file systems.

- **Optimized Performance**: The program is designed to be resource-efficient, capable of handling large datasets without compromising performance.

8.2 Challenges Faced

During the development process, some challenges were encountered, including:

- **Filesystem Library Compatibility**: Ensuring compatibility with the `<filesystem>` library across different compilers and environments posed challenges due to variations in library support. The adoption of C++17 features required attention to compiler versions.

- **Memory Management**: Implementing effective memory management for tokenized words and ensuring proper deallocation posed challenges, especially when handling large datasets.

8.3 Future Enhancements

For future development, the following enhancements could be considered:

- **User Interface Improvements**: Enhance the user interface to provide a more interactive and visually appealing experience. Consider incorporating graphical elements for a more intuitive user interaction.

- **Advanced Search Features**: Implement advanced search features, such as support for regular expressions or the ability to search for multiple keywords simultaneously, to expand the tool's capabilities.

- **Error Handling and Logging:** Strengthen error handling mechanisms and implement logging functionalities to provide users with informative messages in case of errors or unexpected behaviors.

- **Integration with Version Control:** Consider integrating the tool with version control systems, allowing users to analyze changes in text files over time.

Section 9: Bibliography

1. Bjarne Stroustrup. (2013). "The C++ Programming Language." Addison-Wesley.

2. C++ Standard Library - Filesystem.
[cppreference.com](https://en.cppreference.com/w/cpp/filesystem)

These references were instrumental in the development of the project, providing insights into C++ programming practices and the usage of relevant libraries.