## Advanced Data Structures and Algorithm Analysis

# Projects 1: Roll Your Own Mini Search Engine


Date: 2024/03/14

2024 Fall Semester

# Table of Contents

# Chapter 1: Introduction

Nowadays, the large amount of digital information has necessitated the development of efficient search engines that can quickly index large amounts of data and retrive relevent content from vast dataset.

This project aims to create a mini searc engine that capable of handling inquiries over "The Complete Works of William Shakespeare" **and beyond**.

By employing an **inverted file index**, this search engine facilitates efficient querying and retrieval of text passages based on user-defined terms.

## 1.1) Inverted File Index

The inverted file index is the underlying data structure used in my implementation. In computer science, an inverted file index is a database index storing a mapping from content (words in this case) to the location of the documents.

The inverted index data structure is a central component of a typical search engine indexing algorithm. A goal of a search engine implementation is to optimize the speed of the query: find the documents where word occurs.

Compareing to forward file index, inverted file index can be used to quickly locate the documents that contain a specific word.

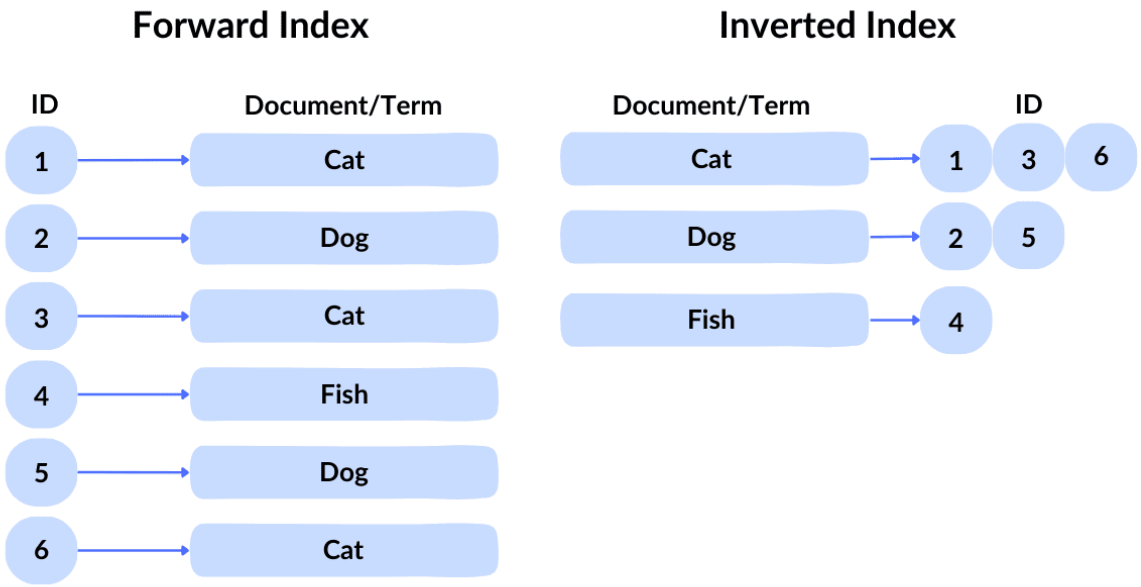


Figure 1: Forward Index v.s. Inverted Index

## 1.2) Word Stemming

Word stemming is an important technique in processing natural language text, aiming to reduce words to their base forms (stems). For example, the words "running," "ran," and "runner" can all be simplified to "run." This simplification helps improve the re-

trieval efficiency of the search engine, allowing different forms of a word to be recognized uniformly.

In this project, I will not implement a ward stemming algorithm myself. As the project requirements said, I will use a third-party library from the Internet, which actually used the **Porter Stemming Algorithm (Porter Stemmer)** to achieve word stemming.

### 1.3) Stop Filter

Stop words are common words in text processing that are considered to have little substantive meaning, such as "the", "is", and "at". These words can often interfere with search results in information retrieval, so they need to be filtered out when constructing the inverted index.

In this project, we will first perform word frequency analysis on "The Complete Works of William Shakespeare" to identify the most common stop words and create a stop word list. Which will be detailed explained in Chapter 2.

## Chapter 2: Algorithm Specification

### 2.1) Task 1: Identify the Stop Words

In this task, I need to count the words over the Shakespeare dataset and identify the most common words. Draw the line between the noisy stop words and the interesting words.

I downloaded "The Complete Works of William Shakespeare" from MIT's Github reposiroty https://github.com/TheMITTech/shakespeare/.

Those documents are in HTML format. In order to count the words over them, I implemented a tokenize method and them use the Porter Stemmer provided by third party to stem the words.

```
1  Function Tokenize(Stream Input)
2      String Result ← ""
3      While the input is not empty
4          Char C ← take one character from Input
5          If C is a letter or a number
6              add C to the end of Result
7          Else If C is '<'
8              take and discard from Input util C is '>' # remove HTML tags
9          Else discard C
10     Return Result
11 End
```

And then I can use the tokenize function to count the words.

```
Function CountWords({Document} Docs)
    Map[String→Interger] M ← {map any string to 0}
    For Doc In Docs
        While the Doc is not empty
            String Word ← Tokenize(Doc)
            M[Word] ← M[Word] + 1
    Return M
End
```

Countering all documents and sort then by most frequent words, I can get the stop words list in `code/word_count.txt`.

```
a: 139065
b: 91350
i: 87263
the: 75794
and: 68794
to: 51440
...
```

I choosed the top 109 words as the stop words (in `code/test/wtop_words.txt`). Because most words in the top 109 are meaningless words, such as “a”, “the”, “and”, “to”. And the words after this line are a mixture of meaningful and meaningless words.

## 2.2) Task 2: Create Inverted File Index

In this task, I create an inverted file index from the Shakespeare dataset. The inverted file index is a data structure that maps each unique word in the dataset to a list of documents that contain that word.

```
Function GenerateIndex({Documents} Docs, {String} StopWords)
    Map[String→{Document}] Index ← {map any string to {}}
    For Doc In Docs
        For Word In Tokenize(Doc)
        If Word Not In StopWords
            Index[Word] ← Index[Word] ∪ [Doc]
    Return Index
End
```

I implemented this in `FileIndex::add_dir()` function in `code/src/FIleIndex.cpp`.

## 2.3) Task 3: Process Query Over Generated Index

In this task, I wirte a program processing a query over the generated inverted file index. The query is a user-specified word of phrase. Our program should return the list of documents that contain that word or the words in the phrase.

```
1 Function Search(String Query, Index Index, {String} StopWords)
2 |  {Document} Docs = {all the documents}
3 |  For Word In Tokenize(Query)
4 |  |  If Word Not In StopWords
5 |  |  |  Docs ← Docs ∩ Index[Word]
6 |  Return Docs
7 End
```

## 2.4) Task 4: Threshold Strategy on Query

In all the words that use input to the search engine, those with less frequency are considered more important. If a threshold $t$ is set, and only take the top $t \times 100\%$ terms in query, we can get the result faster and even more accurate.

```
1 Function SearchWithThreshold(String Query, Index Index, {String} Stop-
    Words, Float Threshold)
2 |  {Document} Docs = {all the documents}
3 |  For Word In Tokenize(Query) If is the t × 100% less frequent of the words
4 |  |  If Word Not In StopWords
5 |  |  |  Docs ← Docs ∩ Index[Word]
6 |  Return Docs
7 End
```

## 2.5) Bonus: On Disk Index Merging

In the requirements of bonus, the program needs to process 500,000 files and 400,000,000 distinct words. This is a huge amount of data. If we store all the data in memory, it will consume a lot of memory and even run out of memory.

So I choose to implement the large scale indexing like this:

1. **Generate Index**: For each document, the program will tokenize it and generate an index file (in binary, to save space) **only contains this single word**.
2. **Merge**: After all documents are indexed, the algorithm will merge all the index files into a single index file. And this operation is **on disk**, the program **will not** load all data to memory.
3. **Search**: When search, the program will only load the words from disk, and record the **offset** of the list of realted documents IDs. During searching, the program will

find the offset by the word, and load the list of related documents IDs from disk. In this way, we only need to load the data that we need, and not all the data.

In this part, we need three functions **Serialize, OnDiskMerge, OnDiskSearch**:

```
Function Serialize(Index Index, {Document} Docs)
    Index Index = GenerateIndex(Docs StopWords)
    open a file F
    save the number of words to F
    For Word In Index
        save the length of the word, the word to F
        For Doc In Index[Word]
            save the size of the list, and the list of related documents IDs to F
    Return
End
```

The merge is much like the algorithm **merge sort**. This algorithm is implemented in `FIleIndex::gen_index_large`

```
Function OnDiskMerge2(File f1, File f2) Return the result file
    put a threshold for number of words, which will be updated later.
    For two entries in the two files
        String Word1, Word2
        get one word from left file, and one from right file
        if the words are not the same, put the smaller word (by alphabet order) to the
         result file
        get the next word, repeat untill Word1 = Word2
        {Document} Docs1, Docs2
        get the documents IDs from left file, and right file
        merge them (they are all ordered, and the result is ordered)
        this step is same as the merge in merge sort
        put the result document IDs to the result file
    move the file pointer to the beginning
    write number of words to the placeholder
End
Function OnDiskMerge(Interger Left, Interger Right)
    If Left = Right
        Return get the file stored
    If Left + 1 = Right
        Return OnDiskMerge2(get the file stored, get the file stored)
    Interger Mid = (Left + Right) / 2
```

```
22 |  File LeftFile = OnDiskMerge(Left, Mid)
23 |  File RightFile = OnDiskMerge(Mid, Right)
24 |  File ResultFile = OnDiskMerge2(LeftFile, RightFile)
25 |  Return ResultFile
26 End
```

The search step:

1. Scan the generated index file, get all the words, and its corresponding offset relative to the file's beginning.
2. For each word, insert the word and the offset to a hash table.
3. This step is the same as function SearchWithThreshold discussed above, but only store the offset and retrive from disk instead.

# Chapter 3: Testing Results

## 3.1) Unit Tests

I designed 7 unit tests, in `code/test/`:

```
word_counting_test
stop_filter_test
build_and_print_index_test
save_and_read_index_test
merge_and_print_index_file_test
search_engine_gen_index_test
search_engine_load_and_search_test
```

They are focused on different parts of the program, and they are all passed. But they are just basic use cases.

```
$ ctest
Test project /mnt/d/courses/ADS/proj/proj1/build
    Start 1: word_count
1/7 Test #1: word_count ......................   Passed    0.22 sec
    Start 2: stop_filter
2/7 Test #2: stop_filter .....................   Passed    0.03 sec
    Start 3: build_and_print_index
3/7 Test #3: build_and_print_index ...........   Passed    1.11 sec
    Start 4: save_and_read_index
4/7 Test #4: save_and_read_index .............   Passed    2.04 sec
    Start 5: merge_and_print_index_file
5/7 Test #5: merge_and_print_index_file .......   Passed    1.50 sec
    Start 6: search_engine_gen_index
6/7 Test #6: search_engine_gen_index ..........   Passed    0.50 sec
    Start 7: search_engine_load_and_search
7/7 Test #7: search_engine_load_and_search ....   Passed    0.04 sec

100% tests passed, 0 tests failed out of 7
```

```
Total Test time (real) =   5.51 sec
```

## 3.2) Further Tests

### 3.2.1) Correctness of the Inverted Index

The correctness of the inverted index can be checked by comparing the output of the search results and the output of `grep` program.

The `grep` program is a powerful tool for searching text using patterns. It can search for patterns in files and output the lines that match the pattern. It's correctness is guaranteed.

```
$ ./ADS_search_engine search ../test/shakespeare/
Enter query (or '/q' to quit): flask
./lll/full.html
./lll/lll.5.2.html
./romeo_juliet/full.html
./romeo_juliet/romeo_juliet.3.3.html
./Tragedy/romeoandjuliet/full.html
./Tragedy/romeoandjuliet/romeo_juliet.3.3.html
```

```
$ grep -rl "flask" ../test/shakespeare
../test/shakespeare/.ADS_search_engine/index.dat # this is the index file
../test/shakespeare/lll/full.html
../test/shakespeare/lll/lll.5.2.html
../test/shakespeare/romeo_juliet/full.html
../test/shakespeare/romeo_juliet/romeo_juliet.3.3.html
../test/shakespeare/Tragedy/romeoandjuliet/full.html
../test/shakespeare/Tragedy/romeoandjuliet/romeo_juliet.3.3.html
```

And we can apply this to any words, not only "flask", I tested many words, the output are all same as `grep`.

### 3.2.2) Thresholding for Query

Given different threshold, the program will choose different terms in the user input.

We use the input query "As we do trace this alley up and down", and threshold 0.2, 0.4, 0.6, 0.8, 1.0 using this script `code/test/test_threshold.sh`. You can check the output in `code/test/output/threshold` folder.

```
#!/bin/bash

thresholds=(0.2 0.4 0.6 0.8 1.0)

query="Then go we near her, that her ear lose nothing"
input_path="../test/shakespeare"

for threshold in "${thresholds[@]}"; do
    output_file="output/threshold/${threshold}.txt"
```

```
    echo "Running with threshold $threshold, output to $output_file"
    ../build/ADS_search_engine search "$input_path" -q "$query" -t "$threshold"
> "$output_file"
done
```

| Threshold | New Used Terms | Number of Results |
|---|---|---|
| 0.2 | near, ear, lose | 65 |
| 0.4 | go, nothing | 58 |
| 0.6 | then, we | 58 |
| 0.8 | her | 58 |
| 1.0 | that | 58 |

The result shows that for this query, we can use **threshold 0.4** to get same results as no using threshold strategy, but more efficiently.

# Chapter 4: Analysis and Comments

## 4.1) Word Counting

The word counting function is implemented using a hash table. And the scanning process is liner, so the overall time complexity is $O(N)$, where $N$ is the input scale.

## 4.2) Stop Filter

The stop filter is also implemented using a hash table, so the process is $O(N)$.

## 4.3) Index Generation

The index generation is implemented using `map`, which is actually red-black tree in C++.

For the native version, the program scan all text once and build the index file, the overall time complexity is $O(N \log M)$, where $N$ is the number of words, $M$ is the number of different words.

For the version using on-dick merge, we should first compute the time complexity of the merge process.

The merge process do not use any special data structure. The time complexity is $O(N_1 + N_2)$, where $N_1$ and $N_2$ are the size of two files. If we have $M$ files to merge, suppose every index file has same size $N$, then we have:

$$T(M) = 2T\left(\frac{M}{2}\right) + M \times N$$

Using the **Master Theorem**, we can get $T(M) = O(NM \log M)$.

### 4.4) Searching

Using hash table, the query's time complexity is $O(N)$, where $N$ is number of used words in the user input.

But, before doing actual searching, we need to load all words from the index file to the hash table, this step is the bottleneck. The time complexity is $O(M)$, where $M$ is the size of the index file.

Loading the index file only need to be done once, so the overall time complexity is $O(M + QN)$, where $Q$ is the number of queries.

## Appendix: Source Code (in C++)

### 5.1) `code/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)
project(ADS_project1)

set(CMAKE_CXX_STANDARD 17)

# set(CMAKE_BUILD_TYPE Debug)

# set executable output directory
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR})
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY_DEBUG ${CMAKE_BINARY_DIR})
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY_RELEASE ${CMAKE_BINARY_DIR})

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include)

# Third party
add_library(stmr STATIC stmr/stmr.c)
target_include_directories(stmr PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/stmr)

# Executable
file(GLOB SOURCES "src/*.cpp")

add_executable(ADS_search_engine ADS_search_engine.cpp ${SOURCES})
target_link_libraries(ADS_search_engine PRIVATE stmr)

# add_custom_command(TARGET ADS_search_engine POST_BUILD
#        COMMAND ${CMAKE_COMMAND} -E copy $<TARGET_FILE:ADS_search_engine>
${CMAKE_CURRENT_SOURCE_DIR}/bin/
# )

# Testing
enable_testing()

file(GLOB TESTS_SRC "test/*.cpp")
```

```
add_executable(tests test/tests.cpp ${SOURCES} ${TESTS_SRC})
target_link_libraries(tests PRIVATE stmr)

add_test(NAME word_count COMMAND tests word_count)
add_test(NAME stop_filter COMMAND tests stop_filter)
add_test(NAME build_and_print_index COMMAND tests build_and_print_index)
add_test(NAME save_and_read_index COMMAND tests save_and_read_index)
add_test(NAME merge_and_print_index_file COMMAND tests
merge_and_print_index_file)
add_test(NAME search_engine_gen_index COMMAND tests search_engine_gen_index)
add_test(NAME search_engine_load_and_search COMMAND tests
search_engine_load_and_search)
```

### 5.2) code/include/FileIndex.h

```
#pragma once

#include <map>
#include <vector>
#include <fstream>
#include <string>
#include <cstdint>
#include <iostream>
#include <filesystem>

#include "StopFilter.h"

/**
 * @class FileIndex
 * @brief This class implements an inverted file index teached in the ADS
course for a mini search engine.
 *
 * The inverted index allows for efficient storage and retrieval of documents
based on keywords.
 * It supports adding files and directories, serializing to and from binary
files, and **merging
 * multiple indexes without loading all data into memory**, making it suitable
for **large datasets**.
 */
class FileIndex {
public:
    /**
     * @brief Adds the content of a file to the index.
     *
      * This function reads tokens from the specified file and updates the
index accordingly.
    * It increments the frequency count of each token and records the document
```

```
ID in which
     * the token appears.
     *
     * @param filename The name of the file to be added to the index.
     * @param id The unique identifier for the document being indexed.
     * @param filter An optional pointer to a StopFilter instance to filter
out stop words.
     */
       void  add_file(const  std::filesystem::path&  filename,  uint32_t  id,
StopFilter* filter = nullptr);

    /**
     * @brief Adds all files from a specified directory to the index.
     *
     * This function retrieves all files in the specified directory and adds
each file's content
     * to the index using a unique document ID starting from id_start. It
supports large datasets
     * by processing files sequentially instead of loading all file contents
into memory.
     *
     * @param dir The directory containing files to be indexed.
     * @param id_start The starting unique identifier for documents. Defaults
to 0.
     * @param filter An optional pointer to a StopFilter instance to filter
out stop words.
     * @return The next available document ID after indexing the files in the
directory.
     */
    uint32_t add_dir(const std::filesystem::path& dir, uint32_t id_start = 0,
StopFilter* filter = nullptr);

    /**
     * @brief Clears the index.
     *
     * This function resets the index by clearing all stored data, preparing
the index for
     * new data to be added. It is essential for memory management, especially
when handling
     * large datasets.
     */
    void clear();

    /**
     * @brief Serializes the index to a binary output stream.
     *
     * This function converts the in-memory index into a binary format for
storage.
```

```
     * The binary format consists of a size header followed by entries for
each word.
     * Each entry includes the frequency of the word, the length of the word,
     * the word itself, the number of documents that contain the word,
     * and the list of document IDs.
     *
     * The binary format of the index is as follows:
     * - uint32_t size: The number of entries in the index.
     * - Entry[] entries: An array of Entry structures, each containing the
serialized data for a word.
     * - For the binary format of Entry, see the `write_entry` function.
     *
     * @param output The output stream to write the serialized index to.
     */
    void serialize(std::ostream& output);

    /**
     * @brief Saves the serialized index to a file.
     *
      * This function opens a binary output file stream for the specified
filename
     * and calls the serialize method to write the index to the file.
     * It ensures that the output file is properly closed after writing.
     *
     * @param filename The name of the file where the index will be saved.
     */
    void save(const std::filesystem::path& filename);

    /**
     * @brief Reads a serialized index from a file.
     *
      * This function opens a binary input file stream for the specified
filename,
     * calls the deserialize method to read and reconstruct the index,
     * and ensures the input file is properly closed after reading.
     *
    * @param filename The name of the file from which the index will be read.
     * @return The FileIndex object reconstructed from the file.
     */
    static FileIndex deserialize(std::istream& input);

    /**
     * @brief Reads a serialized index from a file.
     *
      * This function opens a binary input file stream for the specified
filename,
     * calls the deserialize method to read and reconstruct the index,
     * and ensures the input file is properly closed after reading.
```

```
     *
     * @param filename The name of the file from which the index will be read.
     * @return The FileIndex object reconstructed from the file.
     */
    static FileIndex read(const std::string& filename);

    /**
     * @brief Prints the index to an output stream.
      * This function iterates over the index and prints each word and its
corresponding entry.
     * The entry is printed in the format "word(freq): doc1 doc2 ...", where
"word" is the word, "freq" is the frequency of the word, and "doc1 doc2 ..."
is the list of document IDs that contain the word.
     * @param output The output stream to which the index will be printed.
     */
    void print(std::ostream& output);

    /**
     * @overload
     * @brief Prints the index to a file.
     * This function opens a file stream for the specified filename and calls
the print method to print the index to the file.
     * It ensures that the file is properly closed after printing.
     * @param filename The name of the file where the index will be printed.
     */
    void print(const std::filesystem::path& filename);

    /**
     * @brief Prints the index to a file.
     * This function opens a file stream for the specified filename and calls
the print method to print the index to the file.
     * It ensures that the file is properly closed after printing.
     * @param filename The name of the file where the index will be printed.
     * @param output The output stream to which the index will be printed.
     */
   static void print_file(const std::string& filename, std::ostream& output);

    /**
     * @brief Merge two indexes from two streams into one stream.
     * Merge two indexes from two streams into one stream.
     * This function avoid read all the data into memory.
     * So it can handle large files.
     * @param input1 The first input stream.
     * @param input2 The second input stream.
     * @param output The output stream.
     */
    static void merge_files(
        const std::filesystem::path& filename1,
```

```
        const std::filesystem::path& filename2,
        const std::filesystem::path& output_filename
    );

    /**
     * @brief Read an entry from the input stream.
     * Read a word and an entry from the input stream.
     * @param input The input stream.
     * @param word The word of the entry.
     * @param entry The entry to be read.
     */
    struct Entry {
        uint32_t freq;
        std::vector<uint32_t> docs;
    };

    /**
     * @brief Write an entry to the output stream.
     * Write a word and an entry to the output stream.
     * The binary format is:
     * - word_len (uint32_t): length of word
     * - word (char[word_len]): the word
     * - freq (uint32_t): frequency of the word
     * - num_doc (uint32_t): number of documents
     * - docs (uint32_t[num_doc]): the documents
     */
     static bool read_entry(std::istream& input, std::string& word, Entry&
entry);

    /**
     * @brief Print an entry to the output stream.
     * Print a word and an entry to the output stream.
     * The format is:
     * word(freq): doc1 doc2 ...
     * For example:
     * hello(3): 1 2 3
     * hello is the word, 3 is the frequency, 1 2 3 are the documents.
     *
     * @param output The output stream.
     * @param word The word of the entry.
     * @param entry The entry to be printed.
     */
     static void write_entry(std::ostream& output, const std::string& word,
const Entry& entry);

    /**
     * @brief Merge two entries.
     * Merge two entries into one.
```

```
     * The merged entry has the sum of the frequencies and the union of the
documents.
     * The documents are sorted in ascending order.
     * @param entry1 The first entry.
     * @param entry2 The second entry.
     * @return The merged entry.
     */
     static void print_entry(std::ostream& output, const std::string& word,
const Entry& entry);

    // Helper function to merge two vectors of file IDs
    static Entry merge_entries(const Entry& entry1, const Entry& entry2);
private:
     std::map<std::string, Entry> index; ///< The index of words and their
frequencies and documents.
};
```

### 5.3) code/src/FileIndex.cpp

```
#include "FileIndex.h"
#include "StopFilter.h"
#include "utils.h"

#include <iostream>
#include <fstream>

using namespace std;

/**
 * @brief Adds the content of a file to the index.
 *
 * This function reads tokens from the specified file and updates the index
accordingly.
 * It increments the frequency count of each token and records the document
ID in which
 * the token appears.
 *
 * @param filename The name of the file to be added to the index.
 * @param id The unique identifier for the document being indexed.
 * @param filter An optional pointer to a StopFilter instance to filter out
stop words.
 */
void FileIndex::add_file(const std::filesystem::path& filename, uint32_t id,
StopFilter* filter) {
    ifstream file(filename);
    string token;
    while (file) {
        token = tokenize(file);
```

```
        if (token.empty()) {
            continue;
        }
        // Skip stop words
        if (filter && filter->is_stop(token)) {
            continue;
        }
        auto& entry = index[token];
        // Add document ID to the list of documents containing the token
        if (entry.docs.empty() || entry.docs.back() != id) {
            entry.docs.push_back(id);
        }
        entry.freq++;
    }
}

/**
 * @brief Adds all files from a specified directory to the index.
 *
 * This function retrieves all files in the specified directory and adds each
file's content
 * to the index using a unique document ID starting from id_start. It supports
large datasets
 * by processing files sequentially instead of loading all file contents into
memory.
 *
 * @param dir The directory containing files to be indexed.
 * @param id_start The starting unique identifier for documents. Defaults to
0.
 * @param filter An optional pointer to a StopFilter instance to filter out
stop words.
 * @return The next available document ID after indexing the files in the
directory.
 */
uint32_t FileIndex::add_dir(const std::filesystem::path& dir, uint32_t
id_start, StopFilter* filter) {
    vector<string> files = get_files(dir);
    for (uint32_t i = 0; i < files.size(); i++) {
        // Process files sequentially
        this->add_file(files[i], id_start + i, filter);
    }
    // Return the next available document ID
    return static_cast<uint32_t>(files.size()) + id_start;
}

/**
 * @brief Clears the index.
 *
```

```
 * This function resets the index by clearing all stored data, preparing the
index for
 * new data to be added. It is essential for memory management, especially
when handling
 * large datasets.
 */
void FileIndex::clear() {
    index.clear();
}

/**
 * @brief Serializes the index to a binary output stream.
 *
 * This function converts the in-memory index into a binary format for storage.
 * The binary format consists of a size header followed by entries for each
word.
 * Each entry includes the frequency of the word, the length of the word,
 * the word itself, the number of documents that contain the word,
 * and the list of document IDs.
 *
 * The binary format of the index is as follows:
 * - uint32_t size: The number of entries in the index.
 * - Entry[] entries: An array of Entry structures, each containing the
serialized data for a word.
 * - For the binary format of Entry, see the `write_entry` function.
 *
 * @param output The output stream to write the serialized index to.
 */
void FileIndex::serialize(ostream& output) {
    uint32_t size = static_cast<uint32_t>(index.size()); // Get the size of
the index
   output.write(reinterpret_cast<const char*>(&size), sizeof(size)); // Write
size header
    for (const auto& [word, entry] : index) {
        write_entry(output, word, entry); // Serialize each entry
    }
}

/**
 * @brief Saves the serialized index to a file.
 *
 * This function opens a binary output file stream for the specified filename
 * and calls the serialize method to write the index to the file.
 * It ensures that the output file is properly closed after writing.
 *
 * @param filename The name of the file where the index will be saved.
 */
void FileIndex::save(const std::filesystem::path& filename) {
```

```
    ofstream output(filename, ios::binary); // Open output file in binary mode
    serialize(output); // Serialize the index to the file
    output.close(); // Close the file
}

/**
 * @brief Reads a serialized index from a file.
 *
 * This function opens a binary input file stream for the specified filename,
 * calls the deserialize method to read and reconstruct the index,
 * and ensures the input file is properly closed after reading.
 *
 * @param filename The name of the file from which the index will be read.
 * @return The FileIndex object reconstructed from the file.
 */
FileIndex FileIndex::deserialize(istream& input) {
    FileIndex index;
    uint32_t size; // Variable to store the number of entries in the index
    input.read(reinterpret_cast<char*>(&size), sizeof(size)); // Read size
header
    for (uint32_t i = 0; i < size; i++) {
        string word;
        Entry entry;
        read_entry(input, word, entry); // Deserialize each entry
        index.index[word] = entry;
    }
    return index;
}

/**
 * @brief Reads a serialized index from a file.
 *
 * This function opens a binary input file stream for the specified filename,
 * calls the deserialize method to read and reconstruct the index,
 * and ensures the input file is properly closed after reading.
 *
 * @param filename The name of the file from which the index will be read.
 * @return The FileIndex object reconstructed from the file.
 */
FileIndex FileIndex::read(const std::string& filename) {
    FileIndex index;
    ifstream input(filename, ios::binary);
    index = deserialize(input); // Deserialize the index from the file
    input.close();
    return index;
}

/**
```

```
 * @brief Prints the index to an output stream.
 * This function iterates over the index and prints each word and its corresponding entry.
 * The entry is printed in the format "word(freq): doc1 doc2 ...", where "word" is the word, "freq" is the frequency of the word, and "doc1 doc2 ..." is the list of document IDs that contain the word.
 * @param output The output stream to which the index will be printed.
 */
void FileIndex::print(ostream& output) {
    for (const auto& [word, entry] : index) {
        print_entry(output, word, entry);
    }
}

/**
 * @overload
 * @brief Prints the index to a file.
 * This function opens a file stream for the specified filename and calls the print method to print the index to the file.
 * It ensures that the file is properly closed after printing.
 * @param filename The name of the file where the index will be printed.
 */
void FileIndex::print(const std::filesystem::path& filename) {
    ofstream output(filename);
    print(output);
    output.close();
}

/**
 * @brief Prints the index to a file.
 * This function opens a file stream for the specified filename and calls the print method to print the index to the file.
 * It ensures that the file is properly closed after printing.
 * @param filename The name of the file where the index will be printed.
 * @param output The output stream to which the index will be printed.
 */
void FileIndex::print_file(const std::string& filename, std::ostream& output)
{
    uint32_t size;
    ifstream input(filename, ios::binary);
     input.read(reinterpret_cast<char*>(&size), sizeof(size)); // Read size header
    for (uint32_t i = 0; i < size; i++) {
        string word;
        Entry entry;
        read_entry(input, word, entry); // Deserialize each entry
        output << word << ":";
        for (uint32_t doc : entry.docs) {
```

```
            output << " " << doc;
        }
        output << endl;
    }
}

/**
 * @brief Merge two indexes from two streams into one stream.
 * Merge two indexes from two streams into one stream.
 * This function avoid read all the data into memory.
 * So it can handle large files.
 * @param input1 The first input stream.
 * @param input2 The second input stream.
 * @param output The output stream.
 */
void FileIndex::merge_files(
    const std::filesystem::path& filename1,
    const std::filesystem::path& filename2,
    const std::filesystem::path& output_filename
) {
    ifstream input1(filename1, ios::binary);
    ifstream input2(filename2, ios::binary);
    ofstream output(output_filename, ios::binary);

    uint32_t size1, size2; // size of index
    input1.read(reinterpret_cast<char*>(&size1), sizeof(size1));
    input2.read(reinterpret_cast<char*>(&size2), sizeof(size2));

    string word1, word2;
    Entry entry1, entry2;
    if (size1 > 0) { // if index1 is not empty
        read_entry(input1, word1, entry1);
    }
    if (size2 > 0) { // if index2 is not empty
        read_entry(input2, word2, entry2);
    }

    uint32_t size_merged = 0;
    std::streampos pos_size = output.tellp(); // save position for size
                output.write(reinterpret_cast<const    char*>(&size_merged),
sizeof(size_merged)); // placeholder for size, update later

    while (size1 > 0 || size2 > 0) { // merge until one of the index is empty
        if (size1 > 0 && (size2 == 0 || word1 < word2)) {
            // if index1 is not empty and index2 is empty or word1 < word2,
write word1 to output
            write_entry(output, word1, entry1);
            size_merged++;
```

```
            read_entry(input1, word1, entry1); // read next word
            size1--;
        }
        else if (size2 > 0 && (size1 == 0 || word2 < word1)) {
             // if index2 is not empty and index1 is empty or word2 < word1,
write word2 to output
            write_entry(output, word2, entry2);
            size_merged++;
            read_entry(input2, word2, entry2); // read next word
            size2--;
        }
        else { // both not empty and have same word
            Entry merged = merge_entries(entry1, entry2);
            write_entry(output, word1, merged);
            size_merged++;
            read_entry(input1, word1, entry1); // read next word
            read_entry(input2, word2, entry2); // read next word
            size1--;
            size2--;
        }
    }

    output.seekp(pos_size); // go back to position for size
                output.write(reinterpret_cast<const    char*>(&size_merged),
sizeof(size_merged)); // update size
}

/**
 * @brief Read an entry from the input stream.
 * Read a word and an entry from the input stream.
 * @param input The input stream.
 * @param word The word of the entry.
 * @param entry The entry to be read.
 */
bool FileIndex::read_entry(istream& input, string& word, Entry& entry) {
    uint32_t word_len; // length of word
    if (!input.read(reinterpret_cast<char*>(&word_len), sizeof(word_len))) {
        return false;
    }

    word.resize(word_len); // resize word to word_len
    input.read(&word[0], word_len);

    uint32_t freq;
    input.read(reinterpret_cast<char*>(&freq), sizeof(freq));
    entry.freq = freq;

    uint32_t num_doc;
```

```
    input.read(reinterpret_cast<char*>(&num_doc), sizeof(num_doc)); // read
number of docs
    entry.docs.resize(num_doc); // resize docs to num_doc
         input.read(reinterpret_cast<char*>(entry.docs.data()),  num_doc  *
sizeof(uint32_t));
    return true;
}

/**
 * @brief Write an entry to the output stream.
 * Write a word and an entry to the output stream.
 * The binary format is:
 * - word_len (uint32_t): length of word
 * - word (char[word_len]): the word
 * - freq (uint32_t): frequency of the word
 * - num_doc (uint32_t): number of documents
 * - docs (uint32_t[num_doc]): the documents
 */
void FileIndex::write_entry(ostream& output, const string& word, const Entry&
entry) {
    // write word
    uint32_t word_len = static_cast<uint32_t>(word.size());
   output.write(reinterpret_cast<const char*>(&word_len), sizeof(word_len));
    output.write(word.c_str(), word_len);

    // write freq
    uint32_t freq = entry.freq;
    output.write(reinterpret_cast<const char*>(&freq), sizeof(freq));

    // write docs
    uint32_t num_doc = static_cast<uint32_t>(entry.docs.size());
    output.write(reinterpret_cast<const char*>(&num_doc), sizeof(num_doc));
    output.write(reinterpret_cast<const char*>(entry.docs.data()), num_doc *
sizeof(uint32_t));
}

/**
 * @brief Print an entry to the output stream.
 * Print a word and an entry to the output stream.
 * The format is:
 * word(freq): doc1 doc2 ...
 * For example:
 * hello(3): 1 2 3
 * hello is the word, 3 is the frequency, 1 2 3 are the documents.
 *
 * @param output The output stream.
 * @param word The word of the entry.
 * @param entry The entry to be printed.
```

```
 */
void FileIndex::print_entry(ostream& output, const string& word, const Entry&
entry) {
    output << word << "(" << entry.freq << ")" << ":";
    for (auto& doc : entry.docs) {
        output << " " << doc;
    }
    output << endl;
}

/**
 * @brief Merge two entries.
 * Merge two entries into one.
 * The merged entry has the sum of the frequencies and the union of the
documents.
 * The documents are sorted in ascending order.
 * @param entry1 The first entry.
 * @param entry2 The second entry.
 * @return The merged entry.
 */
FileIndex::Entry FileIndex::merge_entries(const Entry& entry1, const Entry&
entry2) {
    Entry merged;
    merged.freq = entry1.freq + entry2.freq; // sum of frequencies
    auto& docs1 = entry1.docs;
    auto& docs2 = entry2.docs;
    size_t i = 0, j = 0;
    while (i < docs1.size() || j < docs2.size()) { // merge docs1 and docs2
        if (i < docs1.size() && (j >= docs2.size() || docs1[i] < docs2[j])) {
            // if docs1[i] < docs2[j], add docs1[i] to merged
            merged.docs.push_back(docs1[i]);
            i++;
        }
      else if (j < docs2.size() && (i >= docs1.size() || docs2[j] < docs1[i]))
{
            // if docs2[j] < docs1[i], add docs2[j] to merged
            merged.docs.push_back(docs2[j]);
            j++;
        }
        else { // docs1[i] == docs2[j]
            merged.docs.push_back(docs1[i]);
            i++;
            j++;
        }
    }
    return merged;
}
```

### 5.4) code/include/SearchEngine.h

```
#include <string>
#include <cstdint>
#include <unordered_map>
#include <vector>
#include <filesystem>

#include "FileIndex.h"
#include "StopFilter.h"

class SearchEngine {
public:
    using Offset = uint32_t;

    /**
     * @brief Construct a new Search Engine:: Search Engine object
     * @param dir The target directory to search in.
     *
         * This directory should contain a index folder built using
SearchEngine::gen_index(_large).
     * The index folder's name is specified by macro BASE_DIR in utils.h.
     */
    SearchEngine(const std::filesystem::path& dir);

     ~SearchEngine() { delete stop_filter; } // delete stop_filter to avoid
memory leak

    /**
     * @brief Search for a word in the index.
     * @param word The word to search for.
     * @param output The output stream to write the result to.
     * @param threshold The threshold for the search result.
     *
     * Threshold is a ratio from 0.0 to 1.0. It represents the percentage of
terms that should be used in searching. Default value is 1.0
     * For example, if threshold is 0.8, only the top 80% less frequent terms
will be used in searching.
     */
      void search(const std::string& query, std::ostream& output, double
threshold = 1.0) const;

    /**
     * @brief Search for a word in the index.
     * @param word The word to search for.
     * @param output The output stream to write the result to.
     * @return The entry of the word in the index. Retrived from the file.
     */
      FileIndex::Entry search_word(const std::string& word, std::ostream&
```

```
output) const;

    /**
     * @brief Generate an index for the target directory.
     * @param dir The target directory to index.
     * @param stop_filter The stop filter to use. nullptr if no stop filter
is needed.
     * @param quiet If true, do not print any output to stdout.
     */
      static void gen_index(const std::filesystem::path& dir, StopFilter*
stop_filter = nullptr, bool quiet = false);

    /**
     * @brief BONUS: Generate an index for the target directory, but do most
operations on dick to prevent running out of memory.
     * @param dir The target directory to index.
     * @param stop_filter The stop filter to use. nullptr if no stop filter
is needed.
     * @param quiet If true, do not print any output to stdout.
     */
   static void gen_index_large(const std::filesystem::path& dir, StopFilter*
stop_filter = nullptr, bool quiet = false);
private:
    /**
     * @brief Merge the index files generated by gen_index_large.
     * @param dir The target directory to index.
     * @param l The left index of the range to merge.
     * @param r The right index of the range to merge.
     * @param quiet If true, do not print any output to stdout.
     *
     * Merge a series of index file to one. The algorithm is similar to merge
sort.
     * It uses recursion to split the range into two halves and merge them.
     */
    static void merge_index(const std::filesystem::path& dir, std::size_t l,
std::size_t r, bool quiet = false);

    std::filesystem::path dir; ///< The target directory to search in.
    std::vector<std::string> file_list; ///< The list of files in the target
directory.
    std::unordered_map <std::string, Offset> words; ///< The map of words to
their offsets in the index file.
    StopFilter* stop_filter; ///< The stop filter to use.
};
```

### 5.5) code/src/SearchEngine.cpp

```
#include "SearchEngine.h"
```

```cpp
#include <fstream>
#include <iostream>
#include <filesystem>
#include <sstream>
#include <algorithm>
#include <cstdint>

#include "FileIndex.h"
#include "utils.h"

namespace fs = std::filesystem;

/**
 * @brief Construct a new Search Engine:: Search Engine object
 * @param dir The target directory to search in.
 *
  *  This directory should contain a index folder built using SearchEngine::gen_index(_large).
 * The index folder's name is specified by macro BASE_DIR in utils.h.
 */
SearchEngine::SearchEngine(const std::filesystem::path& dir) {
    this->dir = dir;
    std::string line;
     std::ifstream list_fs(dir / BASE_DIR / LIST_FILE_NAME); // this file contains a list of indexed files
    while (std::getline(list_fs, line)) {
        if (line.empty()) continue;
        file_list.push_back(line);
    }
    list_fs.close();

    if (fs::exists(dir / BASE_DIR / STOP_FILE_NAME)) {
      this->stop_filter = new StopFilter(dir / BASE_DIR / STOP_FILE_NAME); // load stop words list from file
    }
    else {
        this->stop_filter = nullptr; // fix bug on 9.29, if not initialized to nullptr, it will crash
    }

    uint32_t size;
    std::ifstream input(dir / BASE_DIR / INDEX_FILE_NAME, std::ios::binary);
     input.read(reinterpret_cast<char*>(&size), sizeof(size)); // read the number of indexed words
    for (uint32_t i = 0; i < size; i++) {
        std::string word;
        FileIndex::Entry entry;
```

```
        std::streampos entry_pos = input.tellg(); // record the offset of the
entry in the index file
        FileIndex::read_entry(input, word, entry);
        words.insert({ word, static_cast<Offset>(entry_pos) }); // insert the
word and its offset into the map
    }
    input.close();
}

/**
 * @brief Generate an index for the target directory.
 * @param dir The target directory to index.
 * @param stop_filter The stop filter to use. nullptr if no stop filter is
needed.
 * @param quiet If true, do not print any output to stdout.
 */
void SearchEngine::gen_index(const std::filesystem::path& dir, StopFilter*
stop_filter, bool quiet) {
   fs::path prev = fs::current_path(); // store the current working directory
    fs::current_path(dir); // change to the target directory
    fs::create_directory(BASE_DIR); // make sure the index folder exists
    fs::path base(BASE_DIR);
    std::vector<std::string> files = get_files("."); // get all files in the
directory
    std::ofstream list_fs(base / LIST_FILE_NAME);
    for (auto& file : files) {
        list_fs << file << std::endl;
    }
    list_fs.close();

    if (stop_filter) {
        std::ofstream stop_fs(base / STOP_FILE_NAME);
        stop_filter->print(stop_fs); // print stop words list to file
        stop_fs.close();
    }

    FileIndex index;
    for (uint32_t i = 0; i < files.size(); i++) {
         if (!quiet) std::cout << "Indexing " << fs::canonical(files[i]) <<
std::endl;
         // canonical() returns the absolute path of the file. For prettier
printing.
        index.add_file(files[i], i, stop_filter);
    }
    index.save(base / INDEX_FILE_NAME); // save the index to file
    fs::current_path(prev); // return to the original directory
}
```

```cpp
/**
 * @brief BNUNS: Generate an index for the target directory, but do most operations on dick to prevent running out of memory.
 * @param dir The target directory to index.
 * @param stop_filter The stop filter to use. nullptr if no stop filter is needed.
 * @param quiet If true, do not print any output to stdout.
 */
void SearchEngine::gen_index_large(const std::filesystem::path& dir, StopFilter* stop_filter, bool quiet) {
   fs::path prev = fs::current_path(); // store the current working directory
    fs::current_path(dir); // change to the target directory
    fs::create_directory(BASE_DIR); // make sure the index folder exists
    fs::path base(BASE_DIR);
    std::vector<std::string> files = get_files("."); // get all files in the directory
    std::ofstream list_fs(base / LIST_FILE_NAME); // write file list to file
    for (auto& file : files) {
        list_fs << file << std::endl;
    }
    list_fs.close();

    FileIndex index;
    for (uint32_t i = 0; i < files.size(); i++) {
         if (!quiet) std::cout << "Indexing " << fs::canonical(files[i]) << std::endl;
         // canonical() returns the absolute path of the file. For prettier printing.
        index.add_file(files[i], i, stop_filter); // add file to index
        std::string name;
                name = std::string("index_part_") + std::to_string(i) + std::string("to") + std::to_string(i) + ".tmp"; // generate file name
        // e.g. index_part_3to3.tmp
        index.save(base / name);
        index.clear();
    }
   merge_index(dir, 0, files.size() - 1, quiet); // merge all the .tmp files *on disk*
      std::string name = std::string("index_part_") + std::to_string(0) + std::string("to") + std::to_string(files.size() - 1) + std::string(".tmp"); // generate file name
    std::filesystem::rename(base / name, base / INDEX_FILE_NAME); // rename the merged file to index
    fs::current_path(prev); // return to the original directory
}

/**
 * @brief Merge the index files generated by gen_index_large.
```

```
 * @param dir The target directory to index.
 * @param l The left index of the range to merge.
 * @param r The right index of the range to merge.
 * @param quiet If true, do not print any output to stdout.
 *
 * Merge a series of index file to one. The algorithm is similar to merge
sort.
 * It uses recursion to split the range into two halves and merge them.
 */
void SearchEngine::merge_index(const std::filesystem::path& dir, std::size_t
l, std::size_t r, bool quiet) {
    if (l == r) return;
    std::size_t m = (l + r) / 2; // find the middle index
    merge_index(dir, l, m, quiet); // merge the left half
    merge_index(dir, m + 1, r, quiet); // merge the right half
      std::string name1 = std::string("index_part_") + std::to_string(l) +
std::string("to") + std::to_string(m) + std::string(".tmp");
    std::string name2 = std::string("index_part_") + std::to_string(m + 1) +
std::string("to") + std::to_string(r) + std::string(".tmp");
      std::string name3 = std::string("index_part_") + std::to_string(l) +
std::string("to") + std::to_string(r) + std::string(".tmp");
    fs::path base(BASE_DIR);
    FileIndex::merge_files(base / name1, base / name2, base / name3); // do
the actual merge
   if (!quiet) std::cout << "Merging " << name1 << " and " << name2 << " into
" << name3 << std::endl; // print the merge operation
    std::filesystem::remove(base / name1); // remove the temporary files
    std::filesystem::remove(base / name2); // remove the temporary files
}

/**
 * @brief Search for a word in the index.
 * @param word The word to search for.
 * @param output The output stream to write the result to.
 * @param threshold The threshold for the search result.
 *
 * Threshold is a ratio from 0.0 to 1.0. It represents the percentage of terms
that should be used in searching. Default value is 1.0
 * For example, if threshold is 0.8, only the top 80% less frequent terms will
be used in searching.
 */
void SearchEngine::search(const std::string& query, std::ostream& output,
double threshold) const {
    std::stringstream ss(query);
    std::vector<std::string> words;
    std::string token;

    while (ss) {
```

```
        token = tokenize(ss); // tokenize the query
        if (token.empty()) continue; // ignore empty tokens
        token = stem_word(token); // stem the word
         if (stop_filter && stop_filter->is_stop(token)) { // if stop word
filter is enabled, ignore stop words
           output << "Stop word \"" << token << "\" is ignored." << std::endl;
            continue;
        }
        words.push_back(token);
    }
    std::vector<std::pair<std::string, FileIndex::Entry>> entries;

    // search each word separetely and then intersect the results
    for (auto& word : words) {
         FileIndex::Entry entry = search_word(word, output); //  search the
word
        entries.push_back({ word, entry });
    }
    std::sort(entries.begin(), entries.end(), [](
        const std::pair<std::string, FileIndex::Entry>& e1,
        const std::pair<std::string, FileIndex::Entry>& e2
        ) {
            return e1.second.freq < e2.second.freq;
      }); // sort by frequency, *ascending*. It is for the querry thresholding
pollicy
    std::vector<uint32_t> res;
    bool first = true; // true if it is processing the first set of results.
    for (std::size_t i = 0; i < entries.size(); i++) {
        auto& entry = entries[i];
       if (i > entries.size() * threshold) { // if the threshold is reached,
ignore the rest of the words
           output << "\"" << entry.first << "\" is ignored due to threshold."
<< std::endl;
        }
        else {
            if (first) { // if it is the first set of results, just assign it
to the result
                res = entry.second.docs;
                first = false; // set the flag to false
            }
            else {
                  res = intersect(res, entry.second.docs); // intersect the
results
            }
        }
    }
    if (res.empty()) { // if the result is empty, print "No results found."
        output << "No results found." << std::endl;
```

```
    }
    else for (auto& doc : res) {
        output << file_list[doc] << std::endl; // print the result
    }
}

/**
 * @brief Search for a word in the index.
 * @param word The word to search for.
 * @param output The output stream to write the result to.
 * @return The entry of the word in the index. Retrived from the file.
 */
FileIndex::Entry    SearchEngine::search_word(const    std::string&    word,
std::ostream& output) const {
    auto it = words.find(word); // find the word in the index
    if (it == words.end()) return FileIndex::Entry(); // if the word is not
found, return an empty entry
    Offset offset = it->second; // get the offset of the word in the index

    std::ifstream index(dir / BASE_DIR / INDEX_FILE_NAME, std::ios::binary);
    index.seekg(offset); // seek to the offset of the word in the index
    FileIndex::Entry entry; // create an entry to store the result
    std::string index_word; // create a string to store the word in the index
    FileIndex::read_entry(index, index_word, entry); // read the entry from
the index
    index.close();

    return entry;
}
```

### 5.6) code/include/StopFilter.h

```
#pragma once

#include <unordered_set>
#include <string>
#include <iostream>
#include <filesystem>

/**
 * @class StopFilter
 * @brief A class to filter out stop words from a text.
 *
 * The StopFilter class provides functionality to load a list of stop words
 * from a file and check whether a given word is a stop word.
 */
class StopFilter {
private:
```

```
    std::unordered_set<std::string> stop_words; ///< A set to store stop
words.

public:
    /**
     * @brief Constructor to load stop words from a file.
     * @param stop_words_file The path to the file containing stop words.
     *
     * This constructor reads the stop words from the specified file and
     * stores them in a set for quick lookup.
     */
    StopFilter(const std::filesystem::path& stop_words_file);

    /**
     * @brief Check if a word is a stop word.
     * @param word The word to check.
     * @return true if the word is a stop word, false otherwise.
     *
     * This method returns true if the word is found in the stop words set
     * or if the word's length is less than 3 characters.
     */
    bool is_stop(const std::string& word) const;

    /**
     * @brief Print the stop words set.
     * @param output The output stream to print to.
     * This method prints the stop words set to the specified output stream.
     */
    void print(std::ostream& output) const;
};
```

### 5.7) code/src/StopFilter.cpp

```
#include "StopFilter.h"

#include <fstream>
#include <iostream>
#include <string>

/**
 * @brief Constructor to load stop words from a file.
 * @param stop_words_file The path to the file containing stop words.
 *
 * This constructor reads the stop words from the specified file and inserts
 * them into the stop_words set for efficient checking.
 */
StopFilter::StopFilter(const std::filesystem::path& stop_words_file) {
    std::ifstream input(stop_words_file);
```

```
    std::string word;

    // Read words from the file and insert them into the stop_words set
    while (input >> word) {
        if (!word.empty()) {
            stop_words.insert(word);
        }
    }
}

/**
 * @brief Check if a word is a stop word.
 * @param word The word to check.
 * @return true if the word is a stop word, false otherwise.
 *
 * This method checks if the word is present in the stop_words set.
 * It also considers words with less than 3 characters as stop words.
 */
bool StopFilter::is_stop(const std::string& word) const {
    if (word.size() < 3) return true; // Treat short words as stop words
    return stop_words.find(word) != stop_words.end(); // Check set for stop
word
}

/**
 * @brief Print the stop words set.
 * @param output The output stream to print to.
 * This method prints the stop words set to the specified output stream.
 */
void StopFilter::print(std::ostream& output) const {
    for (const auto& word : stop_words) {
        output << word << std::endl;
    }
}
```

## 5.8) code/include/utils.h

```
#pragma once

#include <string>
#include <vector>
#include <iostream>
#include <filesystem>

// Define constants for directory and file names
#define BASE_DIR (".ADS_search_engine") ///< Base directory for the search
engine, e.g. the index for `target_dir` will be stored in `target_dir/
<BASE_DIR>`
```

```
#define INDEX_FILE_NAME ("index.dat")    ///< Index file name
#define LIST_FILE_NAME ("list.txt")      ///< List file name
#define STOP_FILE_NAME ("stop_wrods.txt") ///< Stop words file name

/**
 * @brief Get all files from a specified directory with a given extension.
 *
 * This function retrieves all files with the specified extension from
 * the provided directory and its subdirectories.
 *
 * @param directory The path to the directory to search.
 * @param extension The file extension to filter by (default is ".html").
 * @return A vector containing the paths of the found files.
 */
std::vector<std::string> get_files(
    const std::filesystem::path& directory,
    const std::string& extension = ".html"
);

/**
 * @brief Stem a word to its base form.
 *
 * This function takes a word as input and converts it to its
 * lowercase stemmed form.
 * It uses the Porter stemming algorithm to achieve this.
 *
 * @param word The word to stem.
 * @return The stemmed version of the word.
 */
std::string stem_word(const std::string& word);

/**
 * @brief Tokenize a string into words.
 *
 * This function reads input from a stream and extracts a single
 * token (word) while skipping over HTML tags.
 *
 * @param input The input stream to read from.
 * @return The tokenized word in stemmed form.
 */
std::string tokenize(std::istream& input);

/**
 * @brief Intersect two **aescending** vectors of unsigned 32-bit integers.
 *
 * This function finds the intersection of two **aescending** vectors of
unsigned
 * 32-bit integers and returns the result as a new vector.
```

```
 *
 * @param vec1 The first vector to intersect, must be **aescending**.
 * @param vec2 The second vector to intersect, must be **aescending**.
 * @return A new aescending vector containing the intersection of vec1 and
vec2.
 */
std::vector<uint32_t> intersect(const std::vector<uint32_t>& vec1, const
std::vector<uint32_t>& vec2);
```

### 5.9) code/src/utils.cpp

```
#include "utils.h"

#include <filesystem>
#include <iostream>

extern "C" {
#include "stmr.h" // Include the stemmer header from the third-party library
}

/**
 * @brief Get all files from a specified directory with a given extension.
 *
 * This function retrieves all files with the specified extension from
 * the provided directory and its subdirectories. If the directory
 * does not exist, it prints an error message and returns an empty vector.
 *
 * @param directory The path to the directory to search.
 * @param extension The file extension to filter by.
 * @return A vector containing the paths of the found files.
 */
std::vector<std::string> get_files(
    const std::filesystem::path& directory,
    const std::string& extension
) {
    std::vector<std::string> files;
    if (!std::filesystem::exists(directory)) {
         std::cerr << "目录不存在: " << directory << std::endl; // Directory
does not exist
        return {};
    }
                            for       (const       auto&       entry       :
std::filesystem::recursive_directory_iterator(directory)) {
       if (entry.is_regular_file() && entry.path().extension() == extension)
{
           files.push_back(entry.path().string()); // Add file path to vector
        }
    }
```

```
    return files;
}

/**
 * @brief Stem a word to its base form.
 *
 * This function takes a word as input, converts it to lowercase,
 * and applies the stemming algorithm.
 * It uses the Porter stemming algorithm to achieve this.
 *
 * @param word The word to stem.
 * @return The stemmed version of the word.
 */
std::string stem_word(const std::string& word) {
    std::string s = word;
    for (char& p : s) {
        p = tolower(p); // Convert to lowercase
    }
     stem(s.data(), 0, static_cast<int>(s.size() - 1)); // Apply stemming
algorithm, this is a third-party library
    return s;
}

/**
 * @brief Tokenize a string into words.
 *
 * This function reads input from a stream and extracts a single
 * token (word) while skipping over HTML tags. It returns the
 * token in stemmed form.
 *
 * @param input The input stream to read from.
 * @return The tokenized word in stemmed form.
 */
std::string tokenize(std::istream& input) {
    std::string token;
    char ch;

    while (input.get(ch)) {
        if (isalnum(ch)) {
            token += ch; // Start of a token
            break;
        }
        else if (ch == '<') { // Skip HTML tags
            while (input.get(ch)) {
                if (ch == '>') {
                    break; // End of the HTML tag
                }
            }
```

```
        }
    }

    while (input.get(ch)) {
        if (!isalnum(ch)) {
            break; // End of token
        }
        token += ch; // Continue adding characters to token
    }

    return stem_word(token); // Return the stemmed token
}

/**
 * @brief Intersect two **aescending** vectors of unsigned 32-bit integers.
 *
 * This function finds the intersection of two **aescending** vectors of
unsigned
 * 32-bit integers and returns the result as a new vector.
 *
 * @param vec1 The first vector to intersect, must be **aescending**.
 * @param vec2 The second vector to intersect, must be **aescending**.
 * @return A new aescending vector containing the intersection of vec1 and
vec2.
 */
std::vector<uint32_t> intersect(const std::vector<uint32_t>& vec1, const
std::vector<uint32_t>& vec2) {
    std::vector<uint32_t> result;
    auto it1 = vec1.begin();
    auto it2 = vec2.begin();

    while (it1 != vec1.end() && it2 != vec2.end()) {
        if (*it1 == *it2) {
            result.push_back(*it1); // Add common element to result
            ++it1;
            ++it2;
        }
        else if (*it1 < *it2) {
            ++it1;
        }
        else {
            ++it2;
        }
    }

    return result;
}
```

### 5.10) code/include/WordCounter.h

```h
#pragma once

#include <string>
#include <unordered_map>
#include <iostream>

/**
 * @class WordCounter
 * @brief A class to count occurrences of words.
 *
 * The WordCounter class provides methods to add words and print their counts.
 */
class WordCounter {
public:
    using Entry = std::pair<std::string, int>; ///< Type alias for a word and
its count.

    /**
     * @brief Add a word to the counter.
     * @param word The word to be added to the counter.
     *
     * This method increments the count of the specified word.
     */
    void add_word(const std::string& word);


    /**
     * @brief Print the word count result.
     * @param output The output stream to print the results.
     *
     * This method prints the words and their corresponding counts in
     * descending order of frequency.
     */
    void print(std::ostream& output);
private:
   std::unordered_map<std::string, int> word_count;  ///< A map to store word
counts.
};
```

### 5.11) code/src/WordCounter.cpp

```cpp
#include "WordCounter.h"

#include <string>
#include <vector>
#include <algorithm>
```

```
/**
 * @brief Add a word to the counter.
 * @param word The word to be added to the counter.
 *
 * This method increments the count of the specified word.
 */
void WordCounter::add_word(const std::string& word) {
    word_count[word]++;
}


/**
 * @brief Print the word count result.
 * @param output The output stream to print the results.
 *
 * This method prints the words and their corresponding counts in
 * descending order of frequency.
 */
void WordCounter::print(std::ostream& output) {
    std::vector<Entry> word_list(word_count.begin(), word_count.end());

    // Sort the words by frequency in descending order
    std::sort(word_list.begin(), word_list.end(), [](const Entry& a, const
Entry& b) {
        return b.second < a.second;
        });

    // Print each word and its count
    for (const auto& entry : word_list) {
        output << entry.first << ": " << entry.second << std::endl;
    }
}
```

### 5.12) code/test/file_index_test.cpp

```
#include <fstream>
#include <cassert>

#include "FileIndex.h"
#include "tests.h"

int build_and_print_index_test() {
    FileIndex index;
    index.add_dir("shakespeare/merchant");
    std::ofstream file("output/build_and_print_index_test.txt");
    index.print(file);
    file.close();
    return 0;
```

```
}

int save_and_read_index_test() {
    FileIndex index;
    index.add_dir("shakespeare/cymbeline");
    index.save("output/save_and_read_index_test.dat");
    std::ofstream file1("../");
    index.print("output/save_and_read_index_test1.txt");
   FileIndex index2 = FileIndex::read("output/save_and_read_index_test.dat");
    index2.print("output/save_and_read_index_test2.txt");
     assert(files_identical("output/save_and_read_index_test1.txt", "output/
save_and_read_index_test2.txt"));
    return 0;
}

int merge_and_print_index_file_test() {
    std::string prefix = "output/merge_and_print_index_file_test";
    FileIndex index;
    uint32_t id_curr;

    // save index separately
    id_curr = 0;
    id_curr += index.add_dir("shakespeare/richardii", id_curr);
    index.save(prefix + "1.dat");
    index.clear();
    id_curr += index.add_dir("shakespeare/richardiii", id_curr);
    index.save(prefix + "2.dat");
    index.clear();

    // save index together
    id_curr = 0;
    id_curr += index.add_dir("shakespeare/richardii", id_curr);
    id_curr += index.add_dir("shakespeare/richardiii", id_curr);
    index.save(prefix + "_direct.dat");
    index.clear();

    // merge index
      FileIndex::merge_files(prefix + "1.dat", prefix + "2.dat", prefix +
"_merged.dat");

    // compare merged index with direct index
    assert(files_identical(prefix + "_direct.dat", prefix + "_merged.dat"));
    return 0;
}
```

### 5.13) code/test/search_engine_test.cpp

```
#include "SearchEngine.h"
```

```cpp
#include <filesystem>
#include <cassert>

#include "utils.h"

namespace fs = std::filesystem;

int search_engine_gen_index_test() {
    fs::path dir = fs::current_path() / "shakespeare/asyoulikeit";

    if (fs::exists(dir / BASE_DIR)) {
        fs::remove_all(dir / BASE_DIR);
    }

    SearchEngine::gen_index(dir, nullptr, true);
    assert(fs::exists(dir / BASE_DIR / INDEX_FILE_NAME));
    assert(fs::exists(dir / BASE_DIR / LIST_FILE_NAME));
    return 0;
}

int search_engine_load_and_search_test() {
    fs::path dir = fs::current_path() / "shakespeare/asyoulikeit";
    SearchEngine se(dir);
          std::ofstream   output(fs::current_path()   /   "output/
search_engine_load_and_search.txt");
    output << "searching for 'allow'..." << std::endl;
    se.search("allow", output);
    output << "searching for 'love'..." << std::endl;
    se.search("love", output);
    return 0;
}
```

### 5.14) code/test/stop_filter_test.cpp

```cpp
#include <cassert>

#include "StopFilter.h"

int stop_filter_test() {
    StopFilter stop_filter("stop_words.txt");
    assert(stop_filter.is_stop("a"));
    assert(stop_filter.is_stop("the"));
    assert(stop_filter.is_stop("a"));
    assert(!stop_filter.is_stop("hello"));
    assert(!stop_filter.is_stop("world"));
    assert(!stop_filter.is_stop("stop"));
```

```
    return 0;
}
```

### 5.15) code/test/word_counter_test.cpp

```
#include <iostream>
#include <fstream>

#include "WordCounter.h"
#include "utils.h"

using namespace std;

int word_counting_test() {
    WordCounter counter;
    ifstream input("../test/shakespeare/allswell/allswell.1.1.html");
    while (input) {
        string token = tokenize(input);
        if (token.empty()) break;
        counter.add_word(token);
    }
    input.close();
    ofstream output("../test/output/word_count_test.txt");
    counter.print(output);
    return 0;
}
```

### 5.16) code/test/tests.h

```
#pragma once
#include <iostream>
#include <string>
#include <unordered_map>
#include <functional>

int word_counting_test();
int stop_filter_test();
int build_and_print_index_test();
int save_and_read_index_test();
int merge_and_print_index_file_test();
int search_engine_gen_index_test();
int search_engine_load_and_search_test();
bool files_identical(const std::string& file1, const std::string& file2);
```

### 5.17) code/test/tests.cpp

```
#include <filesystem>
#include <fstream>
```

```cpp
#include "tests.h"



int main(int argc, char* argv[]) {
    if (argc == 1) {
        std::cerr << "No test specified.\n\nUseage: tests <test_case_name>\n
- see `CMakeList.txt for the test case names`" << std::endl;
        return 1;
    }

     std::filesystem::path testdir = std::filesystem::current_path() / "../
test";
    std::filesystem::current_path(testdir);

    std::string testname = argv[1];

    if (testname == "word_count") {
        return word_counting_test();
    }
    else if (testname == "stop_filter") {
        return stop_filter_test();
    }
    else if (testname == "build_and_print_index") {
        return build_and_print_index_test();
    }
    else if (testname == "save_and_read_index") {
        return save_and_read_index_test();
    }
    else if (testname == "merge_and_print_index_file") {
        return merge_and_print_index_file_test();
    }
    else if (testname == "search_engine_gen_index") {
        return search_engine_gen_index_test();
    }
    else if (testname == "search_engine_load_and_search") {
        return search_engine_load_and_search_test();
    }

    std::cerr << "Unknown test: " << testname << std::endl;
    return 1;
}

// Utils
bool files_identical(const std::string& file1, const std::string& file2) {
    std::ifstream f1(file1, std::ios::binary);
    std::ifstream f2(file2, std::ios::binary);
```

```
    char ch1, ch2;
    bool has1 = static_cast<bool>(f1.get(ch1));
    bool has2 = static_cast<bool>(f2.get(ch2));
    while (has1 && has2) {
        if (ch1 != ch2) {
            return false;
        }
        has1 = static_cast<bool>(f1.get(ch1));
        has2 = static_cast<bool>(f2.get(ch2));
    }
    return !has1 && !has2;
}
```

## References


- wooorm, "stmr library". https://github.com/wooorm/stmr.c
- William Shakespeare, "The Complete Works of William Shakespeare" http://shakespeare.mit.edu/


## Declaration

I hereby declare that all the work done in this project titled "Roll Your Own Mini Search Engine" is of my independent effort.