Project 2 – Search

CS 251, Fall 2023

Collaboration Policy

By submitting this assignment, you are acknowledging you have read the course collaboration policy. This project should be done individually. You may not receive any assistance outside of the CS 251 instructional team. ***Do not post your work for this class publicly, now or after class.***

Late Policy

You are given a grace period of 24-hours. You do not need to let anyone know you are using this grace period, you simply continue to work and submit during the time period. Beyond this period, no late submissions will be accepted. Please note that if you are continually using the grace period, then you are off track from our anticipated pace that facilitates spacing of assessments.

What & Where to Submit

1. Lab Check-In (marked in lab 9/21 - 9/22) - inverted index image or diagram
    a. **Do not wait until the lab to do this.** As a "lab check," you may ask questions in the lab or during office hours, and the TAs are allowed to be more direct with help.
2. zyBooks - functionality testing, similar harnesses will be utilized by graders
    a. We use multiple tests when officially grading your assessment, these tests may be the same as the public tests or different but testing the same functionality.
3. Gradescope - main.cpp (with your test cases) & search.h for final grading

Table of Contents

Project Summary
Program and Coding Restrictions
The Data
Using an Inverted Index for Searching
Milestone 1 - cleanToken
Testing Implementations
Milestone 2 - gatherTokens
Milestone 3 - buildIndex
Milestone 4 - findQueryMatches
Milestone 5 - searchEngine
Grading Breakdown

Citations & Resources

Julie Zelenski - Stanford University; Joe Hummel, PhD - Northwestern University; Shannon Reckinger, PhD - University of Illinois Chicago

Copyright Notice

Project 2 – Search

CS 251, Fall 2023

Project Summary

Search engines are one of the most influential developments of the modern internet age, having completely revolutionized the way that people use and interact with the web. What was once an intractable jumble of data with no semblance of organization has been transformed into an easily searchable repository of information. In this project, you will recreate this phenomenal technological development by using the **map** and **set** abstractions to build a document search engine that can find matching pages for a user's query with lightning-fast response time. This is a simplified version of the technology underpinning Spotlight, Google, Bing, and every other modern search engine that you encounter in your day-to-day internet use.

In our version of the search engine, each web page has a *URL* ("Uniform Resource Locator") that serves as its unique id and a string containing the body text of the page. The magic of search is enabled by pre-processing the body text of each page and storing the contents into a data structure optimized for fast retrieval of pages matching the search query.

You will first write functions that process the body text and populate the data structure. Next you'll implement the function to search for pages matching a search query. Finally, you will write a console program that allows the user to enter many search queries and retrieve the matching web pages. Put all together, you will have built your own mini search engine!

Primary Project Topics

- String Operations
- File IO
- Maps
- Sets

Given Files (Starter Code)

We have provided a single C++ file, a header file (.h), a makefile, and several text files for testing. The C++ file is for your testing file where all your self-written test cases should be implemented. The header file is where your implementation of the project's milestones will go. The makefile is for compilation and execution, and the text files are data files. You can generate additional testing files following similar patterns to the provided example.

Do not copy-paste starter code, download the files and place them into an appropriate folder.

Executing and Using the Solution

The executable solution can be found within zyBooks.

Additional Reading for After Reading Project Description

- Inverted Index on GeeksForGeeks
- Wikipedia article on Inverted Indexes
- Stanford Natural Processing Group on Tokenization

Project 2 – Search

CS 251, Fall 2023

Program and Coding Restrictions

- Any hard coding, especially to pass test cases, will result in a zero for the entire project.
- You cannot change the function signatures in the starter code. All the functions included are tested in unit tests in the autograder. You will not be able to pass the autograder if these are changed.
- You are allowed to use and add other libraries (make sure to **include** them at the top of your file). However, you do not need many. Any standard C++ library is allowed for this project. You either must find them and figure out how to use them, or write your own functionality.
- Each input file may be opened and read exactly once.
- Your two code files (main.cpp and search.h) must have a header comment with your name and a program overview. Each function must have a header comment above the function, explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate; comments such as *"declares variable"* or *"increments counter"* are useless. Comments that explain non-obvious assumptions or behavior **are** appropriate.
- No global variables; use parameter passing and function return. No heap allocation. No pointers.
- The **cyclomatic complexity** (CCN) of any function should be minimized. In short, cyclomatic complexity is a representation of code complexity --- e.g. nesting of loops, nesting of if-statements, etc. You should strive to keep code as simple as possible, which generally means encapsulating complexity within functions (and calling those functions) instead of explicitly nesting code.

  As a general principle, if we see code that has **more than 2 levels** of explicit looping, you are likely to receive grade penalties. The solution is to move one or more loops into a function, and call the function.

The Data

The format of each data file is as follows:

- The lines of the file are grouped into pairs with the following structure:
  - The first line of a pair is a page URL.
  - The second line of a pair is the body text of that page, with all newlines removed (basically text of the page in a single string).
- The first two lines in the file form the first pair. The third and fourth lines form another pair, the fifth and sixth another, and so on, with alternating lines of page URL and page content.

To view an example data file, open the file tiny.txt (or cplusplus.txt) in the starter code.

Project 2 – Search

CS 251, Fall 2023

Using an Inverted Index for Searching

The key to enabling efficient search of a large data structure comes down to how we structure and store the data. A poor choice in data structures would make search painfully slow, while a wise arrangement can allow search to be near instantaneous.

To begin with, let's consider the index of a book. For example, when you look in the index of a textbook. Let's say you look up **"Internet"** and find it has two page numbers listed: 18 and 821. The word internet occurs on page number 18 and again on page number 821. A book's index is an example of an *inverted index*, where you have a word in mind and you can find the page number it is listed on (a book's index does not usually include *every* instance of the word on all pages but is instead curated to give you the best matches). In other words, an inverted index creates a mapping from content to locations in a document (or table, etc.).

This is in contrast to a *forward index*, which, for the book example, would be a list of page numbers with all the words listed on that page. A search engine uses an inverted index to allow for fast retrieval of web pages – thus, we will first figure out how to build one of these inverted indexes to efficiently store the data that we want to be able to search through.

Efficiency note: To create an inverted index, you must process the entire document, or set of documents, and catalog where each word is located. This may not be a fast process, but once the inverted index is complete, searching for words and their corresponding locations is extremely fast.

Milestone Layout

This problem will be broken up into milestones. You should test exhaustively between milestones. All of the functions will be implemented in the file ***search.h***.

You should call and test those functions in *main.cpp*. You have to write your own tests and will be assessed on their coverage, so test all your functionalities in many different ways.

When we use hidden test cases the names should give you a clue about what's missing.

Milestone 1 - cleanToken

The function `cleanToken` takes in a whitespace-separated string of characters that appears in the body text and returns a "cleaned" version of that token, ready to be stored in the index. The cleaned version has trimmed off any leading or trailing punctuation characters and has been converted to lowercase. If the token contains no letters whatsoever, cleanToken returns an empty string to indicate this token is to be discarded entirely.

More precisely, `cleanToken` should:

- **Remove all punctuation from the beginning and end of a token, but not from inside a token.** The tokens `section` and `section.` and `"section"` are each trimmed to the same token `section`, which makes searching more effective. Tokens such as `doesn't` or `flastname@example.com` are unchanged, since the punctuation occurs in the middle of these tokens.

    - By punctuation, we mean any character for which `ispunct` returns true.
    - There are a few oddball characters (curly quotes, bullets, and the like) that are not recognized as punctuation by `ispunct`. Do not make a special case of this; just trim according to `ispunct`.
- **Return the empty string if the token does not contain at least one letter**, i.e. `isalpha` must be true for at least one character. By returning the empty string, `cleanToken` indicates this token should be discarded as it is not a word.
- **Convert the token to lowercase.** Standardizing on a canonical form for the tokens allows search queries to operate case-insensitively.

The return value from `cleanToken` is the trimmed, lowercase version (or empty string if the token is to be discarded).

Testing Implementations

After and while writing each implementation and before moving on, you should stop and write test cases to confirm your function works correctly on all inputs. You should write several tests to make sure your implementations work. More testing is better, and you will be graded manually by your test case coverage. We will learn how to use unit testing frameworks in C++ at a later point in the term, but for this project, you should utilize and invoke well-named predicate functions within your `main.cpp` file. Here are two sample tests for `cleanToken` that can help you with the idea of how to test. Note this function uses the ternary operator to increase testing counters. These counters could be output or used to present relevant testing results or output could occur per test comparison.

```
1  bool testCleanToken() {
2    string ans = "hello";
3    int pass = 0, fail = 0;
4
5    ans == cleanToken(".hello") ? ++pass : ++fail;
6    ans == cleanToken("...hello") ? ++pass : ++fail;
7
8    return 0 == fail;
9  }
```

Milestone 2 - gatherTokens

The function `gatherTokens` extracts the set of unique tokens from the body text. The single argument to `gatherTokens` is a string containing the body text from a single web page. The function returns a set of the unique cleaned tokens that appear in that body text.

The function must first *tokenize* the body text – in other words, divide into individual tokens, which will be strings separated by spaces. Do this division using string parsing (`string` library functions like `substr` and `find`) or using the `stringstream` library (stringstream example lecture).

Each token is then cleaned using your `cleanToken` helper function. The cleaned tokens are stored in a set. Even if the body text contains a thousand occurrences of the word `"llama,"` the set of unique tokens contains only one copy, so gathering the unique words in a set is perfect for avoiding unnecessary duplication.

Time to test! Add test cases that confirm the output from `gatherTokens`, so you will later be able to call on this function with confidence that it does its job correctly. Here are a couple of sample tests to give you an idea of how to test:

```
1   set<string> tokens = gatherTokens("to be or not to be");
2   set<string> answers = {"to", "be", "or", "not"};
3   tokens.size() == answers.size() ? ++pass : ++fail;
4   tokens == answers ? ++pass : ++fail;
```

Milestone 3 - buildIndex

`buildIndex` reads the content from the file and processes it into an inverted index.

The first argument to `buildIndex` is the name of the database file of the web page data, the second argument is the map to be populated with data for the inverted index. The return value of `buildIndex` is the number of documents processed from the database file.

Before starting to write code, first work through a small example on paper to ensure you understand what your function is trying to build. Write/draw out what the inverted index is for `tiny.txt`. This will be graded as a project progress "check-in" during your lab, so you can do it now before you do any coding!

Review the section on "Data Files" to remind yourself of the format of each data file. Additionally, review any lecture materials on strings and streams to remind yourself of the process as well as the templates.

Once you have read the line containing the body text of the page, call your `gatherTokens` to extract the set of unique tokens. For each token in the set, update the inverted index to indicate that this token has a match to this page's URL.

The action of `buildIndex` is to populate an *index* with entries where each key *word* is associated with a set of *URLs* where that word can be found. The function returns the count of web pages that were processed and added to the index.

The `map` and `set` classes are the right abstractions for storing the inverted index. Sets are especially handy when evaluating a complex query because of the high-level set operations that combine sets and are built into the algorithm library.

Make sure you test your implementation thoroughly in main before moving on.

Error handling for invalid file: If an invalid filename is provided, your code should exit `buildIndex` as soon as possible and return 0.

Project 2 – Search

CS 251, Fall 2023

Milestone 4 - findQueryMatches

The inverted index you built is exactly the data structure needed to enable quickly finding those pages that match a search query. The function `findQueryMatches` implements our search functionality.

The sentence string argument can either be a single search term or a compound sequence of multiple terms. A search term is a single word, and a sequence of search terms is multiple consecutive words, each of which (besides the first one) may or may not be preceded by a modifier like + or – (see below for details). You will need to decide how to combine streams and strings to make this work.

When finding the matches for a given query, you should follow these rules:

- For a single search term, matches are the URLs of the web pages that contain the specified term.
- A sequence of terms is handled as a compound query, where the matches from the individual terms are synthesized into one combined result.
- A search term has a slightly altered meaning when the term is prefaced by certain modifiers:
    - By default when not prefaced with a + or –, the matches are **unioned** across search terms. (any result matching either term is included).
    - If the user prefaces a search term with +, then matches for this term are **intersected** with the existing results. (results must match both terms)
    - If the user prefaces a search term with –, then the **difference** is taken between the current result and the result from the term. Effectively matches for this term are removed from the existing result. (results must match one term without matching the other)
- **The same token cleaning applied to the body text is also applied to query terms.** Call the `cleanToken` function to process each search term to strip punctuation, convert to lowercase, and discard non-words before performing the search for matches.

If you need to review set operations from CS 151, this resource might be helpful. The C++ algorithm library provides union, intersection, and difference functionality. You may use them. Alternatively, you are welcome to write your own union, intersect, and difference functions.

Note that searching is case-insensitive, that is, a search for "apply" should return the same results as "Apply" or "APPLY". Be sure to consider what implications this has for how you create and search the index.

Here are some example queries and how they are interpreted:

- `pointer`
    - matches all pages containing the term `"pointer"`
- `simple cheap`
    - means `simple` OR `cheap`
    - matches pages that contain either `"simple"` or `"cheap"` or both

- `tasty +healthy`
  - means `tasty AND healthy`
  - matches pages that contain both `"tasty"` and `"healthy"`
- `tasty -mushrooms`
  - means `tasty WITHOUT mushrooms`
  - matches pages that contain `"tasty"` but do not contain `"mushrooms"`
- `tasty -mushrooms simple +cheap`
  - means `tasty WITHOUT mushrooms OR simple AND cheap`
  - matches pages that match `((("tasty" without "mushrooms") or "simple") and "cheap")`

There is no precedence for the operators, the query is simply processed from left to right. The matches for the first term are combined with matches for second, then combined with matches for third term and so on. In the last query shown above, the matches for `tasty` are first filtered to remove all pages containing `mushrooms`, then unioned with all matches for `simple` and lastly intersected with all matches for `cheap`. When implementing this logic, you will find the set functions (algorithm library or personally implemented) for union, intersection, and difference to be very handy!

You may assume that the query sentence is well-formed, which means:

- The query sentence is non-empty and contains at least one search term.
- If a search term has a modifier, it will be the first character.
  - A modifier will not appear on its own as a search term
  - A + or – character within a search term that is not the first character is not considered a modifier.
- The first search term in the query sentence will never have a modifier.
- No search term will clean to the empty string (i.e. has at least one letter).

There is a lot of functionality to test in query processing, be sure you add an appropriate amount of tests to be sure you're catching all the cases.

Milestone 5 - searchEngine

Thus far, your amazing code has rearranged a mass of unstructured text data into a highly-organized and quickly-searchable inverted index with fancy query-matching capability. Now take it over the finish line to build your own search engine! The final function `searchEngine` implements the console program that works as follows:

- It first constructs an inverted index from the contents of the database file.
- It prints how many web pages were processed to build the index and how many distinct words were found across all pages.
- It then enters a loop that prompts the user to enter a query
- For each query entered by the user, it finds the matching pages and prints the URLs.
- The user presses enter (empty string) to indicate they are done and the program finishes executing.

Project 2 – Search

CS 251, Fall 2023

After you have completed this function, your program should behave as shown in the transcript shown below. Executed using `cplusplus.txt` as the file to load.

Here is sample output (**red is standard input**):

```
cplusplus.txt
Stand by while building index...
Indexed 86 pages containing 1498 unique terms

Enter query sentence (press enter to quit): vector
Found 8 matching pages
https://www.cplusplus.com/reference/array/array/
https://www.cplusplus.com/reference/bitset/bitset/
https://www.cplusplus.com/reference/forward_list/forward_list/
https://www.cplusplus.com/reference/list/list/
https://www.cplusplus.com/reference/queue/priority_queue/
https://www.cplusplus.com/reference/stack/stack/
https://www.cplusplus.com/reference/vector/vector-bool/
https://www.cplusplus.com/reference/vector/vector/

Enter query sentence (press enter to quit): vector +container
Found 7 matching pages
https://www.cplusplus.com/reference/array/array/
https://www.cplusplus.com/reference/forward_list/forward_list/
https://www.cplusplus.com/reference/list/list/
https://www.cplusplus.com/reference/queue/priority_queue/
https://www.cplusplus.com/reference/stack/stack/
https://www.cplusplus.com/reference/vector/vector-bool/
https://www.cplusplus.com/reference/vector/vector/

Enter query sentence (press enter to quit): vector +container -pointer
Found 6 matching pages
https://www.cplusplus.com/reference/array/array/
https://www.cplusplus.com/reference/forward_list/forward_list/
https://www.cplusplus.com/reference/list/list/
https://www.cplusplus.com/reference/queue/priority_queue/
https://www.cplusplus.com/reference/stack/stack/
https://www.cplusplus.com/reference/vector/vector/

Enter query sentence (press enter to quit):
Thank you for searching!
```

Way to go, 👍 you're well on your way to becoming the next internet search pioneer!

Project 2 – Search

CS 251, Fall 2023

Don't forget to submit your code to all the required places when you are finished.
Failure to submit to Gradescope will result in a zero on the project.

Grading Breakdown

When the test harness goes live this section will be updated with the scoring breakdown. Part of the manual grading will be to assess your testing coverage.

Penalties may occur when manually grading for things like hard-coding or code restriction violations, but here is the breakdown for positive point acquisition. Note that style is not listed, but will be graded and is a "penalty" category that could deduct up to 5 points (5%).

Functionality Score	Milestone 1 - cleanToken	21
	Milestone 2 - gatherTokens	22
	Milestone 3 - buildIndex	18
	Milestone 4 - findQueryMatches	19
	Milestone 5 - searchEngine	5
	Sub Total	**85**
Manual Score	Function Decomposition	5
	Test Cases & Testing Coverage	10
	Sub Total	**15**
	Total	**100**