Programming Assignment 2
A Baby Search Engine[1]

Assignment spec may change depending upon discussion in the class, this is a working doc

Release Time	Due Date
September 21, 2023	**October 5, 2023**

Objectives

- Tokenization
- Creating forward and inverted index
- Search engine basics
- Practice developing high-performance solutions

Problem Specification

Search engines play an integral part in today's tech-savvy society. Their usage is pervasive and ubiquitous. In this assignment, you will experience this technological development by building a baby document search engine that can find matching pages for a user's query with fast response time. This is a simplified version of the technology underpinning Google, Bing, Yahoo and every other modern search engine that you encounter in your daily internet use.

The program you write for this assignment task will produce a custom data structure for a particular set of webpages, represented in the form of a *URL* ("Uniform Resource Locator") and an associated string of text representing the body of the webpage. All five of the functions and tests we'll ask you to implement will be in `search.py`.

Your program will scan the body text of each page to populate the data structure with a mapping from words in the body text to page URLs. Once you have built the data structure, you will then write a function that can search for a target query, given the data structure of webpage information. Finally, you will write a console program that enables a user to enter many different search queries and get back the webpages on which those search queries can be found. Put all together, you will have built your own mini search engine!

Understanding the webpage data

The format of the database 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.
  - <pagebody>

[1] Assignment idea courtesy Bowman and Jue @ Stanford University

    - The third and subsequent lines are the body text of that page until end page marker is encountered.
    - <endPageBody>
- The above key-value repeats for each URL, until the end-of-file marker.

To see what the database file looks like, check out **sampleWebsiteData`.txt`**.

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 can cause search to be very slow, while a wise arrangement can allow search to be near instantaneous. We will use a data storage scheme that can be used to enable a reasonably-fast lookup. You can be creative and improve further.

To begin with, let's consider the index of a book. For example, when you look in the index of a book, one of the entries may be the keyword "**`Internet`**" and two page numbers, 08 and 214. The word internet occurs on page number 08 and again on page number 214. 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 (note: 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 our 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 webpages – 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.

Pause: What is the efficiency of this step (i.e., building inverted index) in comparison to others?

1) Write a word-processing helper `cleanToken()`

We'll start by writing a reusable helper function that will make our lives much easier later down the line. The following function will take in a "token" from a webpage and process it into a trimmed down word that we want to store in our index:

```
string cleanToken(string token)
```

Tokens in our webpage database are whitespace-separated words that appear in the body text of each webpage. To create and return "clean" tokens, your **`cleanToken`** function should meet the following requirements:

- Trim away leading and trailing punctuation marks (using string.punctuation and string.translate()) from each token. Specifically, this means to **remove all punctuation characters from the *beginning* and *end* of a token, but not from inside a token**. The input tokens **`section`** and **`section.`** and **`"section"`** are each trimmed to become the same token (**`section`**), which makes searching more effective. It also means that **`doesn't`** should be stored

as-is, (and likewise for **`as-is`**), since for both of these words the punctuation occurs in the middle of the word.
- "Discard" any non-word tokens by just **returning the empty string if the token does not contain at least one letter**.
- **All tokens should be converted to lowercase format** to make for easier search eventually. Good idea to write a function or use a built-in function to convert.

Once you've edited the inputted token to adhere to the above guidelines, you should return it from your function.

2) Process webpages in `readDocs()`

A good SE practice is to write a structured code and break it up into small functions, i.e., not write a "spaghetti" code. Therefore, we will break down building the index into a couple of different steps. In this first step, we'll process the consolidated database text file into a map of URLs (**`string`**s) to **`Set<string>`**s representing the unique words contained on the given page. You will complete the following function:

```
dict<string, Set<string>> readDocs(string dbfile)
```

This function opens the named file (passed in as **`dbfile`**), reads it line by line, and builds a map from a URL to a **`Set<string>`** representing the unique words contained on that page (i.e., builds the forward index).

To process the database file, your code will need to:

- First, read in the contents of the file into a record: a key (URL) and the content of the webpage.
- For each page URL, *tokenize* its contents without using any NLP library – in other words, divide the body text into individual tokens, which will be strings separated by white spaces. Do this division using the string's split() method.
- Clean each token using your **`cleanToken()`** helper function.
- Store the tokens in forward index.

Note: Even if the body text contains a thousand occurrences of the word "**`teaching`**" the set of unique tokens contains only one copy, so gathering the unique words in a set is perfect for avoiding unnecessary duplication.

Given the following sample file (found in say `sampleWebsiteData.txt`)

```
www.shoppinglist.com
<pageBody> EGGS! milk, fish,      @  bread cheese <endPageBody>
www.rainbow.org
<pageBody> red ~green~ orange yellow blue indigo violet <endPageBody>
www.dr.seuss.net
<pageBody> One Fish Two Fish Red fish Blue fish !!! <endPageBody>
```

```
www.bigbadwolf.com
<pageBody> I'm not trying to eat you <endPageBody>
```

your function should return the following map:

```
{
    "www.bigbadwolf.com" : {"eat", "i'm", "not", "to", "trying", "you"},
    "www.dr.seuss.net" : { "blue", "fish", "one", "red", "two"},
    "www.rainbow.org" : { "blue", "green", "indigo", "orange", "red", "violet", "yellow"},
    "www.shoppinglist.com" :{ "bread", "cheese", "eggs", "fish", "milk"}
}
```

We may provide some unit tests to get you started. Make sure all the provided tests pass, but that won't be enough to fully vet the function. You should add tests of your own to ensure you have comprehensive coverage. Don't move on until all tests pass.

3) Create the inverted index with `buildInvertedIndex()`

Now that we have the document data in a well-structured form, it is time to build our inverted index! Your next task is to complete the following function:

```
dict<string, Set<string>> buildInvertedIndex(dict<string, Set<string>> docs)
```

Given the map of document data (**docs**) that was created by the **readDocs** function, your job is to build the inverted index data structure. The output of **buildInvertedIndex** will be a mapping from *word* (valid token as specified previously) to a *set of URLs* where that word can be found. In other words, this function will invert the map created by **readDocs**!

Before starting to implement this function, take some time to work through the example mappings shown above.

Pause: *What would the sample inverted index for the above example document map in milestone 2) look like?*

Once you've worked through this example, go ahead and implement the function. Make sure to follow the typical cycle of testing and debugging your code. Start with the provided tests (if any given) and then extend with student tests of your own. Don't move on until all tests pass.

4) Search webpages using `findQueryMatches()`

Now that we have built an inverted index, we can turn our focus to figuring out how to give users the ability to query our database and get search results back!

For this part of the assignment, you will be implementing the following function:

```
Set<string> findQueryMatches(dict<string, Set<string>> index, string query)
```

The **query** 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).

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

- For a single search term, the search matches are the URLs of the webpages 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 single search can take on one of three forms:
    - 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 matches for this term are **removed** from the existing result. (results must match one term without matching the other)
- **The same punctuation stripping rules that apply to webpage contents apply to query terms.** Before looking for matches in the inverted index, make sure you strip all punctuation from the *beginning* and *end* of the individual query terms. As before, you should reject any search query terms that don't have at least one alphabetic character.

Here are some example queries and how they are interpreted

- `pakka`
    - matches all pages containing the term "pakka"
- `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"
- `golgappa -sweetchutney`
    - means `golgappa WITHOUT sweetchutney`
    - matches pages that contain "golgappa" but do not contain "sweetchutney"
- `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**. In implementing this logic, you will find the `Set` operators for union, intersection, and difference to be very handy!

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

5) Put it all together with `mySearchEngine()`

So far, we've built capability to turn a mass of unstructured text data into a highly-organized and quickly-searchable inverted index, as well as written code that allows us to find matches in this database given specific user queries. Now, let's put it all together and build our own search engine!

Your final task will be to implement the function below:

```
void mySearchEngine(string dbfile)
```

This function should implement a console program that should implement the following logic:

- Using functions you have already written, construct an inverted index from the contents of the specified file.
- Display to the user how many website URLs were processed to build the index and how many distinct words were found across all website content.
- Once you have the index constructed, your program should go into a loop that allows the user to enter queries that your program will evaluate.
- Given a user query, you should calculate the appropriate matches from the inverted index.
- Once you have computed the resulting set of matching URLs, you should print it to the screen.
- Repeat this process until the user enters the empty string (“”) as their query. At this point, your program should stop running.

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

After you have completed this function, you should be able to have the following interaction flow with the user.

Example program run (executed by running **`mySearchEngine("`**sampleWebsiteOData.txt**`")`** in **`main.py`**):

```
Stand by while building index...
Indexed 50 pages containing 5595 unique terms.[2]


Enter query sentence (RETURN/ENTER to quit): golgappa
```

[2] Note: The number of pages and the number of unique terms here is just an example, it may not correspond to sample data files provided to you.

```
Found 1 matching pages

{"http://cs5950.wmich.edu/homeworks/hw2/searchengine.html"}


Enter query sentence (RETURN/ENTER to quit): basic +tokenization

Found 2 matching pages

{"http://cs5950.wmich.edu/homeworks/hw2/searchengine.html", " https://www.cs.wmich.edu/gupta/teachi
ng/cs5950/5950F23PGSweb/TopicsCovered%20ProgGradStu.html"}


Enter query sentence (RETURN/ENTER to quit): Mac linux -windows

Found 3 matching pages

{"https://cs.wmich.edu/~alfuqaha/Spring06/cs5550/projects.html", "https://www.cs.wmich.edu/~gupta/t
eaching/cs603/wsnSp04/ClassPolicies.html", "https://cs.wmich.edu/elise/courses/cs531/assignments-SI
19.html"}


Enter query sentence (RETURN/ENTER to quit): as-is wow!

Found 3 matching pages

{"http://cs5950.wmich.edu/homeworks/hw2/searchengine.html", "http://timbuktu.edu/foo.html", "https:
//wmich.edu/you.html"}


Enter query sentence (RETURN/ENTER to quit):


All done!
```

Congratulations! You're well on your way to becoming the next internet search pioneer! 🔍

Notes

References

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

If you have creative ideas for extensions or making it more interesting, run them by the course staff, and we'd be happy to give you guidance!

Design Requirements

Code Documentation

For this assignment, you must properly document your code and use good software development practices.

Github

Use github to store your repository. Use good revision-cotrol-system practices as you develop various pieces of the search engine.

Testing

Make sure you test your application with several different values capturing different cases, to make sure it works.

Assignment Submission

- Generate a .zip file that contains all your files, including:
  - Source code files
  - any input or output files
- Don't forget to follow the naming convention specified for submitting assignments
- You will also show execution of your application to grader / instructor. They may give you a test case or two on the spot.