Let's Search

Scenario

Help! Stack Overflow's search system is broken and they need a new one quick! You're all familiar with StackOverflow, the Q&A platform that has probably helped you many times throughout this semester when you encounter a weird error. For this project, you'll be building a search engine for a large collection of questions that have been extracted from Stack Overflow (abbreviated as SO sometimes). Users will be able to enter a search query and you'll return results that are ranked by a relevancy scheme that you devise.

Search Engine Architecture

Search engines are designed to allow users to quickly locate the data they want or need. Input to a search engine is a set of documents commonly referred to as the **corpus**. Your corpus for this project will be the Stack Overflow questions. Typically, the user will enter a search query, and any documents (questions from SO, in this case) that satisfy that query are returned to the user. Another task of a search engine is to rank the results based upon relevancy.

The four major components[1] of a typical search engine are the following:

1. Document parser/processor,
2. Query processor,
3. Search processor, and
4. Ranking processor.

Figure 1 provides a general overview of a potential system architecture:

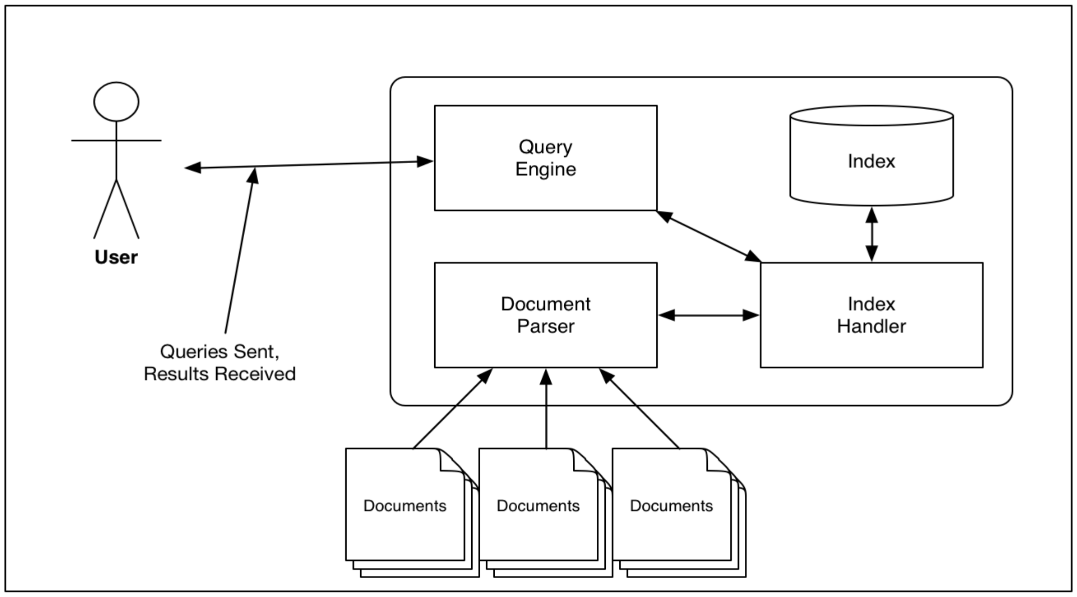

Figure 1 – Sample Search Engine System Architecture

[1]http://www.infotoday.com/searcher/may01/liddy.htm

The fundamental "document" for this project is one question from SO with it's associated information such as date posted and tags.

Below is an overview of the major tasks/responsibilities of each of the components of the search engine.

The **index handler**, the workhorse of the search engine, is responsible for the following:

- *Reading from and writing to the main index*. You'll be creating an **inverted file index** which stores references from each element to be indexed to the corresponding document(s) in which those elements exist. This is essentially the same data structure you created for Programming Assignment 2.
- *Searching the inverted file index based on a request from the query processor.*
- *Storing other data with each word.*

The **document parser/processor** is responsible for the following tasks:

- *Processing each question in the corpus*. The dataset we provide will be a number of comma-separated values files, or CSV for short. There are 2 CSV files for each year since 2008 - one contains the question information and the other contains tags that have been assigned to each post.
- *Removing stopwords from the questions*. Stopwords are common words that appear in text but that provide little discriminatory power with respect to the value of a document relative to a query because of the commonality of the words. Example stop words include "a", "the", and "if". One possible list of stop words to use for this project can be found at http://www.webconfs.com/stop-words.php. You may use other stop word lists you find online.
- *Stemming words*. Stemming[2] refers to removing certain grammatical modifications to words. For instance, the stemmed version of "running" may be "run". For this project, you may make use of any previously implemented stemming algorithm that you can find online. One such algorithm is the Porter Stemming algorithm. More information as well as implementations can be found at http://tartarus.org/~martin/PorterStemmer/. Another option is http://www.oleandersolutions.com/stemming/stemming.html. You may use others.
    - C ++ implementation of Porter 2: https://bitbucket.org/smassung/porter2_stemmer/src
- *Computing/maintaining information for relevancy ranking.* You'll have to design and implement some algorithm to determine how to rank the results that will be returned from the execution of a query. You can make use of tags, important words in the questions (look up term-frequency/inverse document frequency metric), and/or a combination of several metrics.

The **query processor** is responsible for:

- *Parsing of queries entered by the user of the search engine.* For this project, you'll implement functionality to handle ***simple*** prefix Boolean queries entered by the user. The Boolean expression will be prefixed with a Boolean operator of either AND or OR if there more than one word is of interest. Trailing search terms may be preceded with NOT to indicate questions including that term should be removed from the resultset. For simple one-term searches, an operator is not required. Here are some examples:
    - `java`
        - This query should return all questions that contain the word java.
    - `AND programming computer java`
        - This query should return all questions that contain the words programming ***and*** computer ***and*** java
    - `OR java javascript`
        - This query should return all questions that contain either java ***OR*** javascript ***OR*** both.
    - `AND book c++ NOT java`
        - This query should return all questions that contain book and c++, but not java.
    - `Java NOT c++`

[2]See https://en.wikipedia.org/wiki/Stemming for more information.

        - This query should return all questions that contain Java, but not c++.
- *Ranking the Results.* **Relevancy ranking** refers to organizing the results of a query so that "more relevant" documents are higher in the result set than less relevant documents. The difficulty here is determining what the concept of "more relevant" means. One way of handing relevancy is by using a basic **term frequency – inverse document frequency** (tf/idf) statistic[3]. tf/idf is used to determine how important a particular word is to a question from the corpus. If a word appears frequently in question d_t but infrequently in other questions, then question d_t would be ranked higher than another question d_s in which a query term appears frequently, but it also appears frequently in other question as well. There is quite a bit of other information that you can use to do relevancy ranking as well such as date posted, tags associated with each question, etc.

The **user interface** is responsible for:

- Receiving queries from the user
- Communicating with the Search Engine
- Formatting and displaying results in an organized, logical fashion

More info on the UI later.

The Index

The **inverted file index**[4] is a data structure that relates each unique word to the document(s) in which it appears. It allows for efficient execution of a query to quickly determine in which documents a particular query term appears. For instance, let's assume we have the following documents with ascribed contents:

- d1 = `Computer network security`
- d2 = `network cryptography`
- d3 = `database security`

The inverted file index for these documents would contain, at a very minimum, the following:

- computer = d1
- network = d1, d2
- security = d1, d3
- cryptography = d2
- database = d3

The query "AND computer security" would find the intersection of the documents that contained *computer* and the documents that contained *security*.

- set of documents containing computer = d1
- set of documents containing security = d1, d3
- the intersection of the set of documents containing computer AND security = d1

[3]http://en.wikipedia.org/wiki/Tf-idf or http://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html for more information
[4]See http://en.wikipedia.org/wiki/Inverted_index for more information.

Inverted File Index Implementation Details

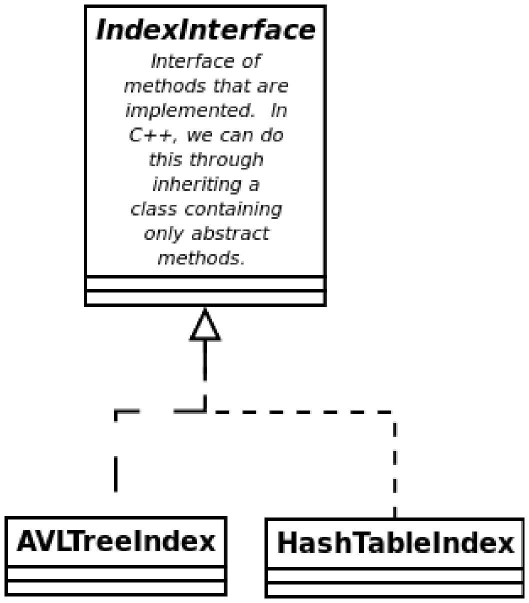
IndexInterface
Interface of methods that are implemented. In C++, we can do this through inheriting a class containing only abstract methods.
AVLTreeIndex
HashTableIndex

Figure 2: Index Interface Class Diagram

The heart of this project is the **inverted file index**. Notice from the example above that what is being stored is essentially words and a list of documents in which each appears (accompanied by some bookkeeping information). Once the index is created, retrieving the set of documents for various words will be the central task of the index. Therefore, we should use a data structure that will allow for efficient searching.

You will implement at least two different data structures to store the index.

- AVL Tree
- Hash Table with collisions handled by separate chaining

You may also include a list implementation of the index. This will provide a good starting place.

In your implementation, you should strive to abstract the index data structure idea from the underlying storage data structure implementation. What this means is that the AVL tree and hash table implementation should share a common interface. In Figure 2, IndexInterface may contain abstract methods such as addWord(), getDocsForWord(word:string), etc.

Each class that inherits from IndexInterface would be required to implement those methods. So that would mean that we could do something like:

```
IndexInterface* if = new AVLTreeIndex;
        or
IndexInterface* if = new HashTableIndex;
```

Index Persistence

The index must also be persistent once it is created. This means that the contents of the index should be written to disk when the program ends and read when the program starts. The user should have the option of clearing the index and starting over.

User Interface

The user interface of the application should provide the following options:

- The program should have **two modes** (controlled by menu or command line parameters)
    - maintenance mode –
        - allows the user to add documents to the index by supplying the path to a new file containing CSV files
        - allows the user to clear the index completely
    - interactive mode –
        - allow the user to indicate if they want the index loaded into an AVL structure or a hash table structure (if a persisted index exists).
        - allow the user to enter a properly formatted Boolean query (as described above).
            - The results should display the question, contributor's ID, date associated with the article, and the relevancy ranking value. The result set shown to the user need not contain any more than 15 articles. You may paginate all postings if you wish.
            - The user should be allowed to choose one of the questions from the result set and have its contents displayed.
        - Note that the query terms should have stop words removed and stemmed before querying the index.
        - Upon request, print basic statistics of the search engine including:
            - Total number of questions indexed
            - Total number of words indexed (after removal of stop words)
            - Top 50 most frequent words
        - Any other options you deem appropriate and useful.

Document Data Set

The CSV files will be available from the Course Webpage on Prof. Fontenot's website.

Mechanics of Implementation

Some things to note:

- This project may be done individually, in teams of two students, or in teams of three students.
    - Individually: Finish all work on your own.
    - Team of 2 students:
        - Each team member must contribute to both the design AND implementation of the project.
        - Each class in the design must have an "owner". The owner is a group member that is principally responsible for its design, implementation and integration into the overall project.
    - Team of 3 students:
        - Complete all work for this project and, additionally, the following feature:
            - Implement 2-word phrase searching. A 2-word phrase search will be indicated by square brackets (e.g. []) around the 2-word phrase. Reject any query that contains more than 2 words in the brackets. Multiple 2-word phrases may be found in one query (e.g. AND [color printer] coffee [digital camera]).
- This project must be implemented using an object-oriented design methodology.
- You are free to use as much of the C++ standard library as you would like. In fact, I encourage you to make generous use of it. You may use other libraries as well except for the caveat below.
    - You must implement your own version of an AVL tree and Hash Table (the storage data structures for the index). You may, of course, refer to other implementations for guidance, but you MAY NOT incorporate the total implementation from another source.

- All of your code must be properly documented and formatted
- Each class should be separated into interface and implementation (.h and .cpp) files unless templated.
- Each file should have appropriate header comments to include owner of the class and a history of updates/modifications to the class

Submission Schedule

You must submit the following:

- **Teams:** Due Monday Nov 6 @ 5pm submitted to TAJake (jrcarlson@smu.edu)
- **Design Documents:** Due Monday November 13, 2017 at 8am (Submit to Canvas)
    - Class diagram indicating
        - Interface for each class
        - Role and responsibilities of each class
        - Owner for each class (if done in groups)
- Implementation Milestone 1: Due In Lab – Week of November 13.
    - Demonstrate to your TA that you've made substantial progress on the document processor and one of the index data structures.
- Implementation Milestone 2: Nov 20/21.
    - Set up a meeting over these two days with your TA to do a quick check in on your progress.
- Parsing Speed Check with Prof. Fontenot - Monday, Nov 27. Sign up sheet will be made available soon.
- Lab time during week of Nov 27 for final polishing and tweaks.
- **Final Project: Due Monday Dec 4, 2017 @ 6:00am (no extensions!)**
    - Complete project with full user interface
        - Your goal for parsing the entire SO export file is 4 minutes. You should be able to implement parsing that can rip through all 200,000 pages in 4 minutes or under.
    - User's Manual to include information on management piece as well as regular user piece
    - Documentation
        - updated UML diagrams
        - documentation about each class in your project. Consider using Doxygen[5] for this.
    - Report that compares the underlying functionality of the AVL implementation vs the Hash Table implementation.
        - Which one is better?
        - How do you know?
        - Can you quantify this?
        - Is one better for small data sets compared to large data sets?
- Demonstration of functionality to Professor Fontenot and TAs on Monday Dec 4, 2017 (sign up sheet to be distributed).

Thoughts and Suggestions

- If you wait even 1 week to start this project, you will very likely not finish.
- A significant portion of your grade will come from your demonstration of the project to Prof. Fontenot and the TAs. Be ready for this.
- Take an hour to read about the various parts of the C++ STL, particularly the container classes. They can help you immensely in the project.
- As mentioned previously, beware of code that you find on the Internet. It isn't always as good as it it may seem upon initial inspection. Make sure that any code you use in the project is cited/referenced in the header comments of the project.

[5]See http://www.stack.nl/~dimitri/doxygen/ for more information.

- Don't take the Thanksgiving break off from the project. Work on it every day. Make use of a variety of options to communicate with your team. You can have virtual meetings through Google Hangout or use GroupMe to keep in touch with text messages.
- Take the SO questions file and examine it. Data is rarely beautiful and nicely formatted. Use the file to extract some sample test files of various sizes (10 pages, 100 pages, 1000 pages) to use during testing. Don't start out trying to index the whole thing.

Grading:
This project is worth ***25% of your final grade*** in this course (all other implementation projects are worth 35% percent of your final grade).

	Points possible	***Points awarded***
Early Design Documents	10	
Milestone 1	10	
Milestone 2	10	
Completed Project		
Completed Project	50	
Documentation	25	
Demonstration	25	