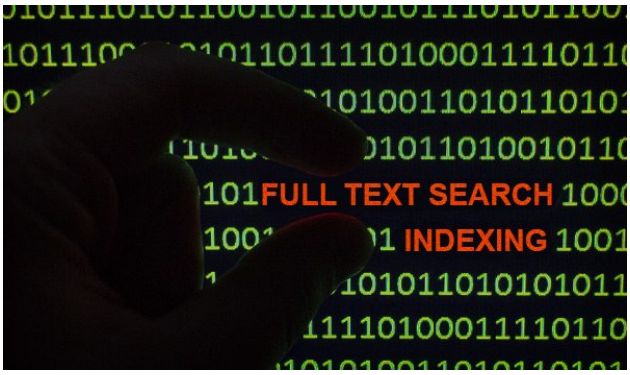
Object Oriented Programming (67125), 2019

Exercise 6: Full Text Indexing and Searching



Caption here

Contents

1 Objectives

By the end of this exercise, you should:

1. Familiarize yourself with the basics of Full-Text indexing
2. Get acquainted with Natural Language Processing (NLP), a fascinating field of Computer Science, and practice simple NLP methodologies
3. Work with multiple packages and gain better understanding of the mechanism
4. Become less afraid of Regular Expressions
5. Learn how to work with `RandomAccessFile` and seamlessly migrate between various representations of a text file
6. Practice implementing and working with serialization in Java.
7. Practice working as a team on a single project!

2 Some Theoretical Background

Full-Text-Searching

In text retrieval, full-text search[1] refers to techniques for searching a single computer-stored document or a collection[2] in a full-text database. Full-text search is distinguished from searches based on metadata or on parts of the original texts represented in databases (such as titles, abstracts, selected sections, or bibliographical references).

Indexing

When dealing with a small number of documents, it is possible for the full-text-search engine to directly scan the contents of the documents with each query, a strategy called "serial scanning". This is what some tools, such as grep, do when searching.

However, when the number of documents to search is potentially large, or the quantity of search queries to perform is substantial, the problem of full-text search is often divided into two tasks: indexing and searching. The indexing stage will scan the text of all the documents and build a list of search terms (often called an index, but more correctly named a concordance). In the search stage, when performing a specific query, only the index is referenced, rather than the text of the original documents.

The indexer will make an entry in the index for each term or word found in a document, and possibly note its relative position within the document. Usually the indexer will ignore stop words (such as "the" and "and") that are both common and insufficiently meaningful to be useful in searching. Some indexers also employ language-specific *stemming* on the words being indexed. A word stem is the basic form of that word- for example, the words "drives", "drove", and "driven" will be recorded in the index under the single concept word "drive".

[1] Adapted from WikiPedia

[2] A collection of related documents like all the writings of Shakespeare or a season of a TV show is often called a "Corpus", or a "Body" of work. In this exercise we also refer to single documents in this manner (a corpus of 1)

Unique Data-Structures in Full Text Indexing:

Building an index may seem like a trivial task - you simply hold a dictionary of all words and their locations as keys. But such a solution can be extremely memory hungry! Storing 1GB of text will undoubtedly take up even more than 1GB, due to the overhead of the data-structure (DS) itself, storing the exact indices (locations) of each result, and other fields used for searching.

So it would appear there's an inherent trade-off between speed of indexing and querying, and the memory footprint of the index DS. Lucky for us, some very smart people came up with efficient algorithms for both searching and indexing. In this exercise we will introduce you to several such solutions, the most interesting of which is the Generalized Suffix Tree. A regular suffix tree is a compressed trie containing all the suffixes of the given string as their keys and their positions in the string as their values. Suffix trees allow particularly fast implementations of many important string operations. A Generalized Suffix Tree (GST) is an extension of that concept, but this time - storing all the suffixes of a **set** of strings. Given the set of strings $D = S_1, S_2, \ldots S_d$ of total length $\sum_{i=1}^{d} |S_i|$, it is a compact suffix tree containing all n suffixes of the strings. Of course, we don't expect your to implement the entire thing! But you would be required to complete some missing parts in the code :-).

Question 1: Why are there n suffixes for a set of strings with total length n? Without formal proof, explain your intuition

3 Exercise Definition

Background

It's ComiCon-2020, and here at the Star Trek (ST) arena, contestants are sharpening their mind grapes for the annual *quote identification* contest. The rules are simple - each contestant has a buzzer, and after hearing a series of words, the first one to buzz and correctly identify the origin of the quote, wins the round!

Being the smart programmers that you are, you never rely on your memory - and you decide to write a program that will aid you in this contest. You managed to bag yourself a set of movies and TV scripts for various ST titles, but you need to find a way to quickly scan through them for the answer.

Since your OOP staff knew you were going to attend this contest instead of studying for your exam, they decided to help out and provide you with the basic necessities for an indexing program. In a News-Forum message, they wrote down the main stages of a good text indexing program, and those are:

1. **Parsing the text** - Each data-set (TV shows and Movies) has it's own specific format which you can exploit to quickly convert a big 'ol chunk of text into bite-sized pieces (or blocks, from now on), that are self contained.

2. **Indexing the text** - A good search scheme doesn't just go over the text willy-nilly. It uses some computer smarts to pre-process the text and store important bits about it to make the searches quicker and allow for "fuzzy searching" (more on that later).

3. **Searching the text** - Once you have your text all parsed and indexed, it's time to search! You're probably familiar with how google takes those 3-4 random words you put in the search field and automagically find the most relevant web page that relates to them. Well, we're not going to do anything that sophisticated, but we'll do more than exact string matching!

4. **Caching** - Finally, what's the point of doing all the hard work if the next time you run your program, you'll have to do it all over again? Well hopefully you wont! As long as the set of documents you processed hasn't changed (how do we know that?) then you can save the indexing results into a cache file and reload them the next time you execute you search program, saving lots of time in the crucial stages of the contest!

3.1 *TextSearcher* Program

The main part of your program would be the *TextSearcher*. This program would have a `main` class that would be responsible for parsing the input arguments file. An input arguments file is similar to the input files you saw in Ex5. It has 3 mandatory fields: `CORPUS` (see 2), `INDEXER` and `PARSE_RULE`; and an optional `QUERY` field. The `main` program would then need to *index* the *corpus* using the *parsing rule* according to the input provided in the relevant fields. An execution of the program would then look something like:

```
student@aqua-05:~%[3] java TextSearcher input.args
```

Where the "input.args" file looks like this:

```
CORPUS
/cs/usr/zivben/Documents/OOP2019/ex6/Corpuses/Scripts_ST_Movies
INDEXER
DICT
PARSE_RULE
ST_MOVIE
QUERY
to boldly go
```

This arguments file tells our program all it needs to do:

- `CORPUS` tells us the path to the *folder* or *file* where the texts are
- `INDEXER` denotes the indexing strategy we will use. The values in this field must be one of the existing constants in our `Aindexer` abstract class, namely:
  - ***DICT*** - Dictionary indexer, backed by HashMap
  - ***NAIVE*** - A naive "indexing" approach, no real index
  - ***NAIVE_RK*** - An extension of the naive approach with faster searching
  - ***SUFFIX_TREE*** - Bonus! A compact data-structure that works like a dictionary but is much smaller in footprint, and allows for partial matches!
  - ***CUSTOM*** - Bonus! more info later.
- `PARSE_RULE` denotes the parsing rule that will be used when indexing the corpus. The values here must be one of the constants defined in the `IparsingRule` interface (SIMPLE, ST_MOVIE or ST_TV). The simple parsing rule is provided for you, the rest you will have to implement.
- `QUERY` is an optional field. If present, it must be followed by a **single** line containing 1 or more words to search for.

[3] Don't be confused by this part, it's simply the linux shell prompt!

Any deviation from the structure of the arguments file should result in graceful termination of the program (i.e. - don't just let it crash with a stacktrace), and an informative printing. You should print out a message stating "Arguments file parsing failed", followed by the name of the invalid field(s) that were encountered. This can be:

- Missing or misformatted fields or field values
- Bad corpus (file or folder do not exist, an error occurs when reading one of the files)

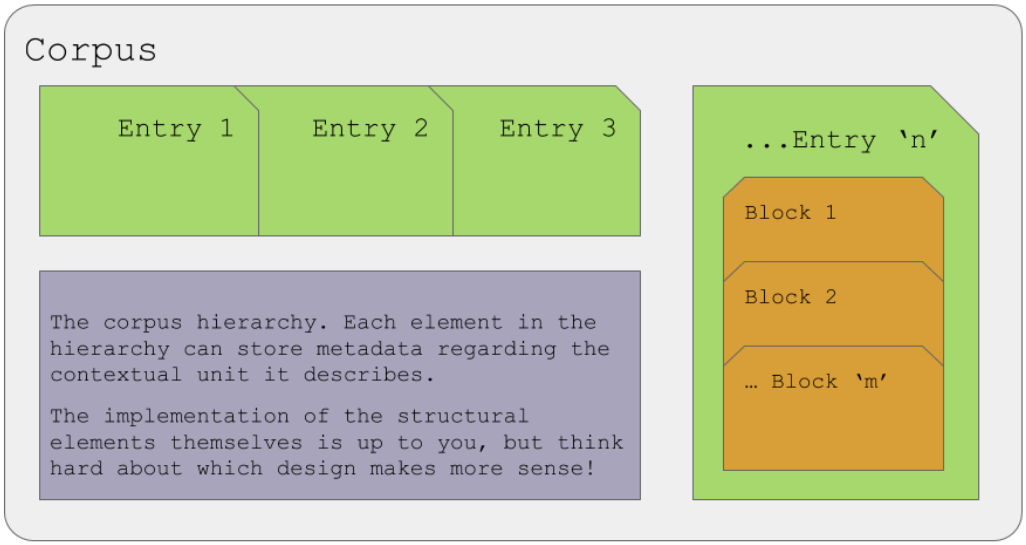
Attention!

Each corpus may have a different predefined format. In this exercise we will describe two such formats that you will have to support, but the program should be extendable to support any other format, as defined by the "IparsingRule" interface to be shown later.

3.2 Structural Hierarchy

Part of this exercise deals with the processing of files within a corpus. We therefore define the following object hierarchy when dealing with text files:

- Corpus - A body of work. Any number of separate files between 1 and 2,147,483,647, provided as a folder or file path.
- Entry - A single Entry is a single file within a Corpus.
- Block - A block is an abstract unit within an Entry. It is represented by the triad (file, start index, end index), meaning that when reading the *bytes* in the file between *start index* and *end index* and converting those bytes to a string, we will get some contextual unit from the file. A block can represent a scene, a sentence, any block of text separated by at least one empty line, or any other format defined by the *parsing rules*.



The Structural Hierarchy of the Text Corpus

Skeleton files for all the classes have been provided for you, but the implementation of some of their methods is empty. You are free to implement any method as you see fit as long as you do not change the method signature. You can add private, package private or protected (where relevant) fields and methods. Overriding or overloading public methods is allowed.

3.3 Text Searching

In this section we will implement two approaches to *"Simple Text Searching"*. Simple text searching (often called *"serial scanning"* involves the querying of a complete text without any pre-processing, or without saving any index to disk.

Starting our journey of *Text Searching*

In it's most basic form, *text searching* or *pattern matching* can seem like a very trivial process - you can simply take your search word (usually noted `pattern`) and start going over full text in overlapping windows, to see whether one of them is a complete `match` for the pattern.

> **Question 2:** Given a pattern `P` of length `m` and a text string `S` of length `n`, what is the time complexity of such a naive search algorithm? Write a pseudo-code for the algorithm and explain your time complexity calculation in the README.

We will introduce several other more efficient algorithms for pattern matching in this exercise, and you will have the pleasure of timing them to see exactly how efficient each one can be.

3.3.1 NaiveIndexer

The first searching algorithm you will implement is indeed the naive (or brute-force) searching algorithm. The way our program will work is by by parsing the arguments file, and creating new **Index** (possibly empty in case of the naive-indexer) and **Parsing rule** instances to be used for future queries. Each indexer in our program would not be very useful if it didn't have a search strategy to back it up. So apart from the Indexer itself, You would also have to write your first strategy- NaiveSearch. To get you going, we have provided a skeleton of the NaiveIndexer class. Our `NaiveIndexer` extends the `Aindexer<NaiveSearch>` abstract class, implementing all it's abstract methods. Check out the API for a complete description of the methods themselves. You may extend the package-private (default) or protected API (not public!) of the abstract class if you feel it would better serve your needs for this exercise. You **may not** modify existing method signatures.

But as we said, the "naive" search strategy is basically just sequential reading of bits of text and checking if they contain the given search string. So our NaiveIndexer doesn't really index anything, it just holds the necessary data members to back our `NaiveSearch`.

The `NaiveSearch` class you will write implements the `IsearchStrategy` interface. This interface defines the contract any search strategy must adhere to so it can be used with our indexers (remember that Aindexer is a generic class that can be initialized with any `T extends IsearchStrategy` implementing class!). `IsearchStrategy` is a functional interface defining a Single Abstract Method.

The method:

```
List<? extends WordResult> search(String query);
```

Is then the only method that you **need** to implement for your Naive Search strategy. The method receives a query string (not necessarily a single word!) and returns a list of `WordResult` objects. You may now implement the pseudo-code you dreamt up earlier in the exercise to search for the query within the text corpus. Again, a skeleton file for the class is provided. You may extend it as you see fit as long as you do not change the public API or any method signature. You are required to implement the search algorithm **without** using the String class methods like equals, substring, etc. You may use "charAt" to access a specific char in a string. You can also implement the search directly on the bytes you read from the RandomAccessFile object if you see fit.

The `WordResult` class is also implemented. You must only implement it's missing method but you may add to it (not public) if you want.

Summary!

Now that we described, in great GREAT length, all that you need to do, let's sum up:

- You have 3 main parts to implement (at least partially!):
  1. The NaiveIndexer class is already completely implemented, but you may add to it if you see the need to.
  2. The NaiveSearch class in which you must (at least) implement the search method which will use to corpus it holds as a data member to access its files, and go over each file (Entry), and for each file go over each Block, loading the text from the file and using whatever string matching algorithm you thought of the search for occurrences of **the exact input string** from the arguments file (treat multiple query words as one single string and match them to the text in the file).

3. Since you need to iterate over a corpus, you need to implement the hierarchy! The skeleton files and partial implementations of the *Corpus, Entry, Block* and *WordResult* classes are all in the supplied exercise files. You must finish the implementations of all of them, and as usual - you can extend private and package-private API, but not public! and do not change any method signatures.

- Since the creation of the corpus involves breaking text files into blocks, you can use the supplied `SimpleParsingRule` class to iterate over an entry and generate blocks from it. The implementation here is complete, and you can view this as a tutorial of how to work with the `RandomAccessFile` class. You will have to write your own parsing rules later!
- After you have created your Corpus and divided each file into blocks, you should write your main program in the `TextSearcher` class. This main program, same as with Ex5, must parse the arguments file and execute the relevant tasks. For now, you can focus on a simple arguments file which uses only the NaiveIndexer and SimpleParsingRule (provided in the exercise files) and make sure that your code runs on simple one word queries, or treat multiple word queries as a single string (i.e. - for the query "*set phasers to stun*", don't break into words, but search for exact match for that string).

3.3.2 The Rabin-Karp method

Now that your most basic implementation is done, we'll want to be smarter about how we search the text. You may have noticed that the naive search complexity is not linear with the length of the text. A smarter way to go about this would be to use something called a "Rolling Hash". A rolling hash is a hash function where the input is hashed in a window that moves through the input, meaning that calculating the hash of a substring takes into account parts of the substring that were already hashed! This means that scanning a text of size n for a pattern of size m is done in $O(n + m)$ time!

Of course the algorithm itself is quite complicated, but we encourage you to read it and try to see how it works. You may consult the pseudo code in the WiKi page for extra explanations. Since the algorithm itself is already implemented, all you need to do is implement the search method found in the `NaiveSearchRK` class to leverage the new search algorithm for querying your Corpus!

This means to implement the missing "public List¡WordResult¿ search(String query) " method from the interface using the "searchBlock" method provided.

Bonus! - The Rabin-Karp method we provided you with runs on char arrays, which requires reading the bytes from the file and converting them to chars. Since both the *chars* that are present in an ASCII text file and *bytes* we read directly from it have the same number of possible values (ASCII range is 0-255 and bytes are -127 to 128), try to think of a way to use the same RK method on byte arrays directly. Modify the exisitng `RabinKarpMethod(char[] pattern, char[] text)` to work on byte arrays, and make any changes needed in the `NaiveSearchRK` class to use this soution.

> **Question 3:** If you decided to implement the Byte version of the RK method, describe in the readme what changes were necessary and whether you saw a speed increase in searching when using this method as opposed to the regular one.

3.3.3 The Dictionary approach

By now you must already be aware of how complicated and slow the previous approaches are when running over very large texts. Enter- text indexing. This is a similar mechanism to what search engines use to index the web so you can get to your kitty videos in a flash! The basic Idea is to preprocess the entire text only once, which could be a costly process, but ensures very fast queries can be performed over the entire corpus. Here you will implement a very basic Dictionary style index.

As usual, the skeleton of the DictionaryIndexer and the DictionarySearch are already provided. But you will still have to implement all the methods from the abstract class and Interface these implementations derive from. This time, however, you will actually have to do the indexing! So:

- When creating a DictionaryIndexer you must pass the constructor a Corpus. This means that the indexer is always aware of the corpus it is going to index.
- Next part is the indexing! you may notice that the Aindexer abstract class already has an implementation of the `index()` method, which calls the `readIndexedFile()` method that is not yet implemented. Lets leave the reading for last, and focus on what comes if we can't read from a cache file.
- The `indexCorpus()` method is what does all the heavy lifting. After you have used your simple parsing rule to break the text into blocks, iterate over each block and read it's content. Read the blocks into strings and iterate over their words to store into the dictionary.
- Make sure to keep track of the indices within the file when using the `RandomAccessFile` methods. You need these indices to correctly identify the location of each word.
- Create a HashMap in you DictionaryIndexer class that will store a list of `Word` objects as values and some key to identify the words. The key can be the word string or a hashcode of the string (or any other unique identifier you may consider). You will use the `Word` objects as values and not just the strings as you want a wrapper class to hold extra data for each word, such as it's index within the block or entry, the file it was taken from, and any other data you feel should can be helpful for displaying results.
- Search queries are a funny thing. Sometimes people write one word but mean another. A very common misuse of words is using the wrong word form, for instance writing "drove" instead of "driven" or "drive". A helpful way to minimize the size of the dictionary and maximize the accuracy of a search is to use word stemming (as described before). Use the Stemmer class to stem words before putting them in the dictionary. For instance, if when scanning the file you reach the word "drove", stem it and get the word "drive", and store the index under that key instead. This will allow for a greater chance in finding the right result even if the query is miswritten!
- Implement the rest of the Indexer methods that are required by the API. You should be familiar with them already as they are mostly the same as with the NaiveIndexer.
- Reading and Writing the index - Once you have finished indexing a new Corpus, you should write it to a cache file using serialization (see Section 4.2). The cache file should hold all the data necessary to restore your program to working order. This means storing the dictionary (one dictionary for all the files in the corpus!) and probably the corpus as well, since you don't want to reread it. Any other data members or data structures you may have created to back your design should also be serialized to file.
  - Implement the writeIndexFile method. Using the path for the corpus, find the parent folder (if it's a single file) or use the folder that was provided (if indeed the input was a folder) to write the cache to.

* Like you saw in the recitation, open a new `FileOutputStream` to a file called

$$< IndexerType > \_ < ParseRule > \_ < CorpusName > .cache$$

and a new `ObjectOutputStream` composed onto it. For example, a NaiveIndexer with a SimpleParsingRule on a folder called "ST_Movies" will be called:

$$NAIVE\_SIMPLE\_ST\_Movies.cache$$

You don't have to use the actual enum values of the parsing rule or indexer, it just needs to be consistent so multiple runs with different arguments can be made, and separate caches saved for each one.

* into this stream you should write the MD5 hashcode for the entire corpus (see Section 4.3) so you can identify in the future if the corpus has changed, as well as the Corpus itself (to prevent scanning it in the future) and finally any other datamembers you would need to restore your indexer. This would obviously mean the hashmap itself, but if you use any other data members you must serialize them as well.
* make sure all the objects you write to the stream are serializeable! And don't forget to add a SerialVersionUID.
* In a similar fashion, implement the readIndexFile method. Look for a file with the same name you have used before (this is a convention for all indexers!) and start reading from it using a `FileInputStream` composed onto an `ObjectInputStream`. Remember that you are reading objects out of the stream in the same order you have written them into it.
* Remember to cast each object you read into the relevant type.

As for the DictionarySearch, you may notice that a Constructor is already present in the skeleton file which receives a hashmap object. You may change the type of the map keys to whatever you have used in your indexer implementation.

- Implement the search for words but this time you must break the query into single words.
- Use the stemmer on the input words to find the correct keys that you have used when indexing.
- Since you will get a list of words to search for, you will also get a list of lists of locations where all of those words appear! You will have to aggregate this list of lists into a single list of `MultyWordResult` objects. A MultiWordResult is a container representing a single match for multiple words.
- You will have to figure out for yourself how to convert a list of lists of words into a single list of matches. This could be a very complex task so here are two pointers:
  1. w.l.o.g, if three words are given as a query, a match will see all three in the same block.
  2. You need to implement a method for calculating the confidence about a list of locations in the MultiWordResult class, this means that the closer all query words are, the more likely it is that they are a good result. So three words in the same block, that are spread across less than, lets say, 20 charachters, are probably a relevant match. and the closer they are, the more confident you will be of the match. So calculate the sum of distances (or average distance if you want, this won't be auto-tested) between all words in a result.
- there is no efficiency requirement for this task (finding a complex result out of multiple single word results), but as always, try to find an elegant way to solve this task. A recursive solution would probably be best.

3.3.4 Parsing Rules

Until now, we have only used the simple parsing rule which breaks texts up to uneven sized groups of lines. But we if we have some prior information about format of our files which we can leverage to get more accurate results? Well we do! and you can!

We have provided you with two sample files for each type of corpus, *TV episodes* and *Movies*. Both look similar but have a slightly different format. You should complete the implementation of both the ST.*ParsingRule implementations to create blocks according to the rules below, and store all relevant metadata information in the relevant Entry/Block objects.

TV episodes

Each script for the *Star-Trek: The Next Generation* show has the following format:

- Any number of blank lines, followed by a line containing the string "STAR TREK: THE NEXT GENERATION"
- The next (non-blank line) will be the *episode title*. This is one of the fields you would want to store as an `Entry` metadata item.
- The next lines may or may not hold credits for "Story by", "Teleplay by" and "Directed by". Sometimes simply "by". Try and match these as well!
- After the credits there are multiple lines of information that is not constant among scripts, and so you may ignore it. The next interesting line is the first scene:
- Each scene in the scripts starts with a header line that looks like this:

```
107  INT. TRANSPORTER ROOM
```

  With the scene number at the very beginning of the line, followed by a scene description. A single scene is defined as all rows of text between two consecutive occurrences (If a scene starts with 107, the next scene will only start when you read 108)
- Each scene can be separated into specific row types by the indentation. So for the scene:

```
103  FULL ON BRIDGE (OPTICAL)

    BLINDING BLUE LIGHT from all panels, ENERGY FORMS
    (lightning-like) flings some of the bridge crew to the
    floor...

                WORF
        Captain! I'll help you if you'll
        let me...
```

  * We have the title row with scene number un-inddented
  * The scene description itself are the rows indented by one `tab` charachter
  * A character dialog is indented by three `tab`s and must immediately follow a character name, which is indented by five `tab`s.

    * If a conversation is specifically between two characters, then an optional line indented by 4 `tab`s can appear stating who the main character is addressing.
  - You should make not of all the metadata a scene holds to allow for quicker querying (remember the #QUICK modifier on the queries? now is where it comes in handy!)

Movies

The movie scripts follow a very similar convention to the TV episodes, with some slight variations.

- Each script starts with the writing credits at the very first line.
- These may be either "Screenplay by", "Story by", "Written By". We don't care about participating writers as they contribute NOTHING to the trope.
- Each scene in a movie script starts with a line that looks like:

```
                    29   EXT. VULCAN LEDGE - SPOCK ( M & F.P.)                    29
```

  Notice how the scene number is at both the beginning of the line as well as the end.
- Character names are indented by **43** spaces
- Scene descriptions are indented by **19** spaces
- Dialogue is indented by **29** spaces

In general, there are no requirements about what metadata you save, how you save it or how you print it out at the end. Try to design your program to be the most elegant in your eyes and describe in the README any design choices you made.

USE REGEX!!!

Printing Results

When your program ends up with a list of possible matches, you should print them out to screen:

- Only print the "top 10" (or up to 10 if there are less) results. For the simple parsing rule, this means the first 10 according to the order of the file names in the corpus (lexicographical) regardless of folders. For the other parsing rules, sort them by the confidence score you gave each result.
- For each result, print the following:
  * "The top 10 results for query " $\langle query \rangle$ " are:"
  * followed by an entry for each result. Each entry starts with a separator line of 256 "=" signs
  * Each entry will then print the line(s) where the match was found. If all query words match in a single line, print the substring that starts with the first word and ends with the first "line break" after the last word. If the query words span multiple lines, the same rules apply, only you should also grab the line breaks between the first and last words.
  * Below each result print out the metadata of the result, if applicable. For instance, after matching the query "emptiness vessel" to the text:

```
emptiness of this
vessel: most of our trainee crew
```

The following line will be printed:

```
Appearing in scene 5, titled "INT. BRIDGE - USS ENTERPRISE - CLOSE - JAMES T. KIRK"
Taken out of  of the entry "StarTrek_Movie_TSFS.txt"
With the characters: "KIRK"
Written by: "HARVE BENNETT"
```

3.3.5 Bonus - Suffix Tree

The "Crown Jewl" of text indexing, a Suffix Tree is an optimized data structure that allows searching for all substrings within a string in very efficient time, while maintaining good balance regarding storage requirements. Here you will receive a complete implementation of a Generalized Suffix Tree, following a linear-time construction algorithm first described by Esko Ukkonen way back in 1995.

Using the same conventions as before, write a `SuffixTreeIndexer` under the `dictionary` package. The complete documented code is available for you, you just need to figure out how to use it!

3.3.5.1 Bonus - The Final Frontier

Now that you have seen and implemented multiple strategies for full text indexing, we encourage you to try and implement your own strategy to compete in the *battle of the nerds*! Create a new class called CustomIndexer, similar to those you have written before. Your implementation is completely up to you! We recommend thinking about various requirements of the competition such as speed and space requirements, and how each method works with regards to each requirement. you can mix and match various parts of the other algorithms to perfect your strategy and claim your crown!

4 Appendix

4.1 `RandomAccessFile` class

From the *Java 7 API JavaDoc*:

> Instances of this class support both reading and writing to a random access file. A random access file behaves like a **large array of bytes** stored in the file system. There is a kind of cursor, or index into the implied array, called the **file pointer**; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; ... The file pointer can be read by the **getFilePointer** method and set by the **seek** method.
> It is generally true of all the reading routines in this class that if end-of-file is reached before the desired number of bytes has been read, an **EOFException** (which is a kind of IOException) is thrown. If any byte cannot be read for any reason other than end-of-file, an IOException other than EOFException is thrown. In particular, an IOException may be thrown if the stream has been closed.

- You will spend a lot of time working with text files in this exercise, and in order to speed up searching in large text files, you *need* to use the `RandomAccessFile` class for that.
- This class provides you with the ability to start reading a file at any random point(er) within the file, without having to scan through all previous lines.
- While this class allows you to read a text file "line-by-line", it is important to follow the file pointer (using the `getFilePointer` method) so you would be able to return to the same exact location in the file in a future time.
- Be sure to peruse the class API so you are better familiar with the available methods. Don't pay too much attention to the methods pertaining to writing, as we will be using a different way to write our index files. Of particular importance are the methods:
  * `public void seek(long pos)` which moves the file pointer to a specific offset in the file (measured in bytes)
  * The various read methods:
    · `public int read(byte[] b, int off, int len)`
    · `public int read(byte[] b)`
    · `public final void readFully(byte[] b)`
    · `public final String readLine()`

    and others, which allow you to read parts of the file according to the file pointer location and the size of the output byte array.

  But of course, if you wish to use other ways to read the files, you are free to do so.

4.2 Serializations

From the wonderful Java Developers blog Mkyong:

> Java object `Serialization` is an API provided by Java Library stack as a means to serialize Java objects. Serialization is a process to convert objects into a writable byte stream. Once converted into a byte-stream, these objects can be written to a file. The reverse process of this is called de-serialization.
> A Java object is *serializable* if its class or any of its superclasses implement either the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`.

- Part of the exercise requires creating a data-structure for storing the index and then writing it to disk
- In order for an object to be serializeable, it requires a specific datamember to be created in the object class called `serialVersionUID`. You have already seen this member in TA7
- The proper syntax is:

```
private static final long serialVersionUID = 1L;
```

- The actual `long` value assigned is not important, what's important is that if you make any changes to the class you are writing to disk (and you will need to save data to disk if you want your exercise to be efficient!), you must increment the value. Think of it like a software version

- Changing the *serial Version UID* allows java to know that a change in the DS has been made, and that it should not try to load a version (stored in file) into a new object with a different version.
- A single file can hold many objects. Objects written to the file (serialized) are written and read in a FIFO manner - this means that if you write Obj1 to file, and then write Obj2, when reading from the file you must first read to an object the same type as Obj1 and then you can read again to a second object the same type as Obj2.

4.3 More helper classes

We provide you with a partial implementation of the *Suffix-Tree* DS, as well as some helper classes to use in your exercise. Be sure to read the API of each class and understand the functionality each brings to the table in the context of this exercise. Making smart use of these packages is crucial to implementing an efficient solution. You are also encouraged to read the code of these classes to see some examples of different writing styles and some interesting algorithms.

These classes are:

- `MD5.java` - This class allows you to calculate an MD5 Checksum for some input file. This is sort of an elaborate hash function that allows you to check if some file has changed since last you accessed it.
- `Stemmer.java` - This class provides you with tools for stemming (see also Section 2) word into it's *word stem* - allowing for less precise (fuzzy) searching, and also reduces the index size greatly, since many different words are reduced to the same word stem.
- `Stopwords.java` - This class provides you with simple methods to check whether a word belongs to the list of *stop words* (again see Section 2). It is also used by the `Stemmer` so make sure you are not using it twice!
- The `suffixTree` package - Here you will get **most** of the code "for free", but will have to implement certain code snippets within the package.

5 Further Guidelines

- Remember the helper classes from Section 4.3? well use them! when reading and writing your indices to disk, you should make sure that the Corpus you are given matches the Corpus that was previously stored (what if someone added a file to it?) This can be easily done by calculating an MD5 Checksum string for each file (the checksum should be calculated over the content of the file!), concatenating them together, then calculating an MD5 checksum on the long string. This "fingerprint" for the Corpus can be serialized and written to the cache file along with your index, so when you try to read the index you can compare the checksum of the stored index with that of the given corpus. Mismatched checksums would mean that your index is outdated and needs to be remade!
- Other useful tools are the Stemmer and StopWords classes! when a user puts in a query, he may not remember correctly some of the words, writing down "He drove to san jose" instead of "he was driven to san jose". of course, words like "he" "was" and "to" are so common that they aren't very useful for searching. But only searching for "drove san jose" may not match the correct occurrence of "driven san jose" which appears in the text. Use the StopWords class to filter out

useless words for both indexing and searching, and use the Stemmer class to convert words to their base form both when indexing and searching. This will save space for the index file, and will help you create a smarter search scheme!

- We have provided you with a zip file containing the basic implementation and skeleton classes. Apart from that, the zip contains the following:
  * `Resources` folder - this folder contains two corpuses, each containing two files. Test your program on these files individually and as folders. We will test your program on more files, but they will have the same format as these.
  * `conf` folder - this folder contains examples of arguments files. You may need to change the path to the corpuses in them according to how you run your program.
  * `simple` folder - this folder contains extremely short files with random strings in them so you have a small input to test your program on. You may of course use any text files you want.
- The program should terminate if an arguments file is not found, or contains invalid format. The arguments file only contains one "block" of instructions, unlike ex5.

6 General Instructions

- **Start early**. In fact, start **today**. We know we keep saying this, but trust us, we really mean it.
- Read this document, write yourself notes, and repeat at least 3 times. Exercises descriptions, like *Product Requirements Documents* (PRDs) you'll see when you start working, are hard to understand and easy to misinterpret.
- Document your code using the Javadoc documentation style, as described in the coding style guidelines and exampled in the code. You should check yourself by invoking: `javadoc -private -d doc *.java` within your project directory or by running the designated tool in your IDE.
- Remember encapsulation.
- Write your code according to the styling guidelines.

7 Submission

1. Deadline: **Tuesday, June 25 2019, 23:55**
2. Make sure you do not submit any .class or Javadoc by mistake.
3. Create a JAR file named ex6.jar containing your implementation. This will be the entire skeleton project you received (only the SRC directory!) including any new classes you may have added.
4. Make sure that the JAR file you submit passes the presubmit script by **carefully** reading the response file generated by your submission. **Exercises failing this presubmit script will get an automatic 0!**
5. Submission will open on Monday next week.
6. Since this is your first exercise in pairs for this course, and since we know exams are coming up, we have given you some extra time for this exercise so plan your time carefully and work as a team!