CS 314H Honors Data Structures — Fall 2017	Programming Assignment #7
Web Crawler and Search Engine	Due December 11, 2017

In this assignment you will work *individually* to build a simple web crawler and search engine. Your main tasks are to crawl a portion of the web, to build an index that allows you to quickly access portions of this web, and to respond to various types of queries—or web searches—much like Google queries. For example, you should be able to type "hoojiedoober" and get a list of pages that contain the word "hoojiedoober."

This assignment asks you to choose appropriate data structures to support various query operations. We ask you to consider three different types of queries, and we allow you to use different data structures to support each type of query.

1 Words of Wisdom

This is a large assignment with what may seem like a distant deadline. In other words, there is plenty of rope with which to hang yourself. We strongly encourage you to (1) start early, (2) think carefully about your design, (3) think about testing early and often, and (4) bite off and debug small pieces of the assignment before writing more code.

This assignment represents the culmination of your semester, as we're confident that you can now handle an open-ended assignment of this scope. If you need more incentive to start early, realize that this assignment will be weighted more heavily than any of the others.

2 Web Queries and Search Engines

A typical search engine supports some form of a query language. For example, a simple query might be a string, for which the search engine would return a list of URL's of web pages that contain that string. More advanced features might include a logical OR operator, the ability to search for synonyms, or the ability to restrict a search to a particular Internet domain. To satisfy such queries, a special HTTP request is sent to the server, which parses the query and then returns the result in the form of web page that contains a set of URL's that satisfy the query.

Given the enormous size of the web, how do search engines return their results so quickly? These search engines crawl the web and build indices of subsets of the web. Good engines have clever ways of identifying what to index and how to index them.

3 Your Assignment

For this assignment, you will crawl a small self-contained portion of the web. The code that we provide **should not be used to crawl the real web** because our code does not follow established guidelines for crawling. Moreover, if you try to crawl the real web, you will end up building enormous indices. (If you're curious to understand the guidelines for web crawling, the following URL provides a nice set of guidelines: `http://www.robotstxt.org/guidelines.html`. You should also understand the repercussions of not following such guidelines: `http://xxx.lanl.gov/RobotsBeware.html`.)

More specifically, your assignment is to implement three components: WebCrawler, WebIndex, and WebQueryEngine. **WebCrawler** is a stand-alone application that crawls the web, starting at a URL specified on the command line. The information gathered by the crawler is stored in a **WebIndex** object, which is saved to disk when WebCrawler terminates. Once a WebIndex has been built, **WebQueryEngine**, which is driven by a server that we provide, loads a previously created WebIndex object and is then able to perform queries on the information stored in the WebIndex.

This assignment is structured as a series of increasingly complex types of queries. Before describing these types of queries, we first explain the three components that you will implement. To simplify your task, we provide pieces of each class, along with other support classes that interface with the web server, that act as the GUI for your query engine, etc. In addition to the provided code, you are free to use any of the data structures in the Java Collections Framework.

3.1 The WebCrawler class

The **WebCrawler** can be invoked from the command line on a URL. See the Java documentation on the `URL` class for the various formats that work. For example, the TA can point the crawler at the copy of the class website on his hard drive with the command `java WebCrawler file://localhost/home/jyq777/cs314h/www/index.html`. The procedure for specifying files on your own machine may differ depending on how your operating system handles hostnames and paths.

The WebCrawler works by giving all of its work to another class, the **CrawlingMarkupHandler** which builds off of an HTML parsing library called attoparser (`http://www.attoparser.org/`). The library decomposes HTML documents into their different pieces and calls individual methods in the handler for each piece. To make your handler useful, you will need to modify the following methods:

- `public void handleDocumentStart(long startTimeNanos, int line, int col)`
  This method is called at the start of each page being parsed.
- `public void handleDocumentEnd(long endTimeNanos, long totalTimeNanos, int line, int col)`
  This method is called when the end of the page being parsed has been reached.
- `public void handleOpenElement(String elementName, Map<String, String> attributes, int line, int col)`
  This method is called at the start of each tag found on the page. Any tag, no matter if it is considered a one-part tag like `<img>` or a two-part tag like `<a>` will cause this method to be called. Any additional information inside of the tag will be parsed and stored in the `attributes` map for your parser to use.
- `public void handleCloseElement(String elementName, int line, int col)`
  This method is called either when the second part of a two-part tag is encountered or when the parser believes a tag should end. Note that tags that are one-part tags in HTML like `<img>` are automatically closed by the parser and this method will still be called.
- `public void handleText(char ch[], int start, int length, int line, int col)`
  This method is called whenever the parser encounters characters that are not part of the attributes for a tag. That means that these characters will most likely be displayed on the webpage. Note that the parser may not return all of the characters associated with a single tag at the same time as it is allowed to chunk the data in any way it sees fit.
- `public Index getIndex()`
  This method is used to get a WebIndex object out of the Crawler once all of the parsing is done.
- `public List<URL> newURLs()`
  This method is used to inform the WebCrawler that more URLs have been found that should be parsed later. The WebCrawler's `main` method calls it after each page is parsed in order to update its own list.

To show you how the parser will call these methods, the partially implemented CrawlingMarkupHandler class that we provide includes code that parses web pages and prints the sequence of callback method calls and encountered HTML elements. It also includes some javadoc comments that should be helpful, but if you want to learn more about the parser you can always look online for the library's API. You will want to change these methods to actually build your index and not to print anything to the console while parsing.

3.2 The WebIndex Class

We provide very little of the WebIndex class. The methods in this class will only be called by the code that you write in WebCrawler and WebQueryEngine, so you can build whatever indexing structures you wish. However, this class should implement the Serializable interface, so any data members that you use must be Serializable. Serializable objects can be easily saved to and restored from disk.

You have tremendous freedom to design the index. Therefore, documentation is extremely important. Be sure to include in your documentation a detailed description of your design and *why* you chose your design, including such factors as runtime and space considerations. Analyze and discuss the performance of your design. Of course, your code should be well-commented.

3.3 The WebQueryEngine Class

For the WebQueryEngine class, you should implement the following methods:

- `public static WebQueryEngine fromIndex(WebIndex index)`
  This method is a factory constructor used to get a WebQueryEngine that is backed by the given WebIndex object.

- `public Collection<Page> query(String query)`
  This method takes a query expression as an argument, parses the query, and returns a list of URL's to pages that match the query (represented as Page objects). Additional details about parsing can be found in the next section.

The Page class we provide is simply a wrapper around the URL class, which is what our tests will use to evaluate the quality of your `query` results. If you want to pass additional information between your index and the WebServer for any of the karma on this project, feel free to add any additional fields or methods to the Page class, but be sure to always populate the URL.

You may (and probably will) add additional methods to the query engine, but you must support the above two methods and not alter their semantics. We will test these two methods using automated techniques, so be sure that there are no hidden assumptions that would cause such a program to fail.

4 The Query Language

We are now ready to explain the various types of queries that your search engine should support. For each type of query, you have two tasks: (1) represent these queries and (2) efficiently perform these queries. Before you start your implementation, think carefully about both of these aspects of the problem. You may use different indexing structures and different strategies to implement different types of queries. In your report, be sure to explain why you chose your various data structures.

While the parser for the query language is not the most important part of the assignment, it does require a fair amount of explanation. When designing your index, you should also keep in mind the kinds of queries that you want to support, as they will have a great impact on your design decisions.

4.1 Basic Queries

The first part of your assignment is to support simple queries, which consist of individual words, the logical AND (&) operator, the logical OR (|) operator, and parentheses. To simplify the parsing, your language will have to fully parenthesize each query, ie, any use of AND or OR requires a set of parentheses. If you would like to relax these constraints, feel free to do so. Here are some examples of basic queries:

- `snufflelupagus`
  Find pages that contain the word "snufflelupagus."

- `(rosencrantz & guildenstern)`
  Find pages that contain both "rosencrantz" and "guildenstern."

- `(naughty | bear)`
  Find pages that contain either "naughty" or "bear."

- `((wealth & fame) | happiness)`
  Find pages that contain both "wealth" and "fame" or pages that contain "happiness."

4.2 Parsing

You might find it useful for your query engine to parse the String that represents the query into some internal representation before you perform your search. For example, you might represent the above query as the tree shown in Figure 1.

You do not need an explicit representation of the parse tree, but having one will likely make it *much* easier for you to optimize your search strategies. For example, to satisfy the above query, you could independently search for all pages

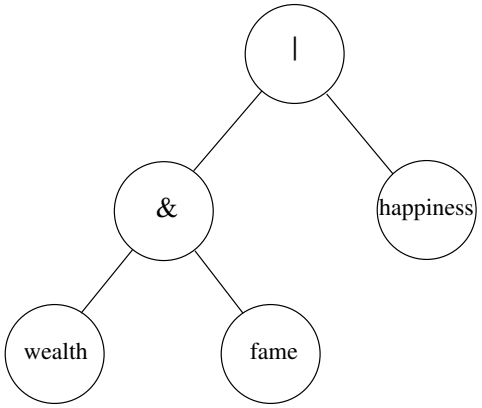

Figure 1: Example of a parse tree.

that contain the word "wealth," find all pages that contain the word "fame," and then take the intersection of these two sets. However, it'd probably be much faster to find all pages that contain "wealth" and of these pages search for those that also contain the word "fame." Having a parse tree can help you do this.

You will learn much more about parsing if you take a compiler course, but for now just use the following two-level approach. First, identify *tokens*, which in our case will be either words or one of the operators. To be more precise, we will treat the left parenthesis as a separate token from the right parenthesis. Second, parse these tokens into a tree.

We have not formally defined what is allowed as a word in our search engine. At a minimum, it should be any combination of letters and numbers. If you want to define words more liberally, you may do so. Be sure to provide your definition of a word in your code and report. Since most search engines perform case-insensitive searches, you should also do so.

Given our definition of a word, we can identify tokens using the following pseudocode:

```
Token GetToken(String stream)
{
  c = first character in stream;
  if (c == "&")
      return (AndToken);
  else if (c == "|")
      return (OrToken);
  else if (c == "(")
      return (LeftParenToken);
  else if (c == ")")
      return (RightParenToken);
  else
  {
      read until blank or operator;
      rewind the stream one character if it was an operator
      return a Token that contains a reference to the word;
  }
}
```

Notice that you won't always know when a token ends until you have already read the first character of the next token, so you'll have to be careful to make sure you do not accidentally lose characters. You should choose an internal representation for tokens that is easy to work with and easy to understand.

This step is known as *lexical analysis* (or just *lexing* for short) and allows you to iterate through the tokens in a string of input in order.

Once you can identify tokens, you can create the parse tree using the following pseudocode (this type of parser is known as a *recursive descent parser*).

```
Tree parseQuery()
{
  t = getToken();
  if (isLeftParens(t))
  {
     left = parseQuery();              // recursively build the left subtree
     op = getOperator();               // get the binary operator: AND or OR
     right = parseQuery();             // recursively build the right subtree
     op = getToken();                  // read the remaining Right Parens
     return makeBinaryNode(op, left, right);
  }
  else if (isWord(t))
  {
     return makeWord(t);               // return a simple word as a query
  }
  else
  {
     // a parse error has occurred; do something
  }
}
```

To understand this parser, it might help you to realize that this parser corresponds to the following set of rules. Starting with *Query*, this set of rules, collectively known as a grammar, can generate the set of all legal queries.

Query → (*Query* & *Query*)
Query → (*Query* | *Query*)
Query → word

The left hand side of each arrow is called a *non-terminal* and the right hand side is called a *production*. For example, the first line can be read as "A *Query* is an open paren, a *Query*, an ampersand, a *Query*, and a close paren."

It is easy for an undisciplined programmer to produce a parser that almost works but contains difficult to fix bugs. To avoid this, you should completely understand the grammar and sketch out your design *before* you start coding. There should be a very clear correspondence in your parser between the rules of the grammar and the structure of your code.

One way to structure your code is to have a method for each nonterminal that contains conditions that correspond to each production. This will make your code correspond very cleanly to the grammar and will assist in clarity and debugging.

4.3 Negative Queries

The second part of your assignment will add the ability to make negative queries. The NOT operator (`!`) matches pages that do not contain the specified word.

Our grammar for this second part includes the following rules.

Query → (*Query* & *Query*)
Query → (*Query* | *Query*)
Query → word
Query → !word

The pseudocode from above would be extended with another clause that might look like this:

```
// earlier clauses
if (isNegation(t))
{
  word = getToken();
  return makeNegation(makeWord(word));
}
// later clauses
```

For extra karma, you can support negation of arbitrary queries, not just words, which changes the last grammar rule to *Query* → ! *Query*, with corresponding changes in your code.

4.4 Phrase Queries

The third type of query is a phrase query, which searches for a contiguous sequence of words. The phrase is indicated by surrounding a sequence of words in double quotation marks, for example, `"john paul george"`. The new grammar is shown below:

Query → (*Query* & *Query*)
Query → (*Query* | *Query*)
Query → word
Query → !word
Query → "*Words*"
Words → word *Words*
Words → word

The two productions for *Words* define a list of one or more words as a word or as a word followed by more words.

4.5 Implicit AND Queries

Most search engines support implicit logical AND operators: If a query consists of consecutive words (not in quotation marks), the engine searches for pages that contain both words. Modify the parser to support implicit ANDs, matching the following grammar, which uses *Query'* for the old Query symbol and uses a non-terminal Query to allow for the implicit AND.

Query → *Query' Query*
Query → *Query'*
Query' → (*Query'* & *Query'*)
Query' → (*Query'* | *Query'*)
Query' → word
Query' → !word
Query' → "*Words*"
Words → word *Words*
Words → word

Notice that you do not have to worry about precedence in cases like `foo bar | baz`. This isn't a valid query, since the OR is not parenthesized.

5 Testing

We may provide some sample webs for you to play with. You can also download other websites using some tool and crawl the copy on your hard drive. These webs will let you see how your search engine works, but they are no substitute for rigorous testing.

One way to help test your code is to write a program that generates large random graphs representing a "web" with randomly chosen words and randomly chosen links. The program can write this web out to a collection of HTML files for use by your engine. Although the pages generated in this manner will be gibberish, you can in mere seconds create huge webs to test against.

You will want to check that you are returning the correct URLs in a search. On Unix-based operating systems (like Linux and Mac OS X) there are command line tools like `grep` that allow you to search files for certain patterns. This will let you easily check which pages in your web contain certain words. Other operating systems may have similar tools; ask around.

Lastly, you should keep an eye on efficiency. Obviously, you won't be able to index the billions of pages that Google does, but you should be able to handle tens or hundreds of thousands of pages. Comment on any scalability issues in your report.

6 Karma

There are numerous things you can do to make your project more fun and entertaining. You can do the following or come up with your own ideas.

- The existing search engine page is a bare-bones implementation. Modify it to be more functional or application-like. Google is a good example of an attractive and functional interface, and they provide a front-end framework called AngularJS which simplifies the creation of web applications that have good interfaces.

  If you implement highlighted excerpts in the results page or a highlighted "Google cache," you will need to add your own methods to support such functionality. Feel free to do so, but be sure not to break the required `useWebIndex` and `query` methods.

- Modify the parser so that it excludes common words such as "where," "what," "how," "and," "or," "a," "an," "of," and "I". If a common word is essential to a search, the plus sign (+) can be used to explicitly include it in the search. For example, `Star Wars Episode +I` will include the "I" in the search. When common words are excluded, Google provides an explanation, so you can determine what Google's common words are through trial and error. More information about Google's query language can be found at the following URL: `http://www.google.com/support/websearch/`. This page also describes Google's Advanced Search options, which provide additional ideas.

  Note that this requires changes to the parser and lexer. Be sure that you do not break anything in the process.

- Support full negative queries. There are a few ways to do this. You can choose to support them directly in your search engine; this will have some impact on your design choices. You can also remember DeMorgan's Law and other logic principles and convert full negative queries into something you can handle. Think about how you want to handle precedence of negation if you do this.

- Remove the restriction on parentheses. You can modify the query language so that parentheses are no longer required, and the operators have the appropriate precedence. This requires more challenging modifications to the parser. Again, be sure that you do not break support for the required query language in the process; as long as your language is a superset of ours, it will be ok.

- Rank your results in some meaningful manner. Many search engines rank pages by frequency of access, but since you will not have such information, you could try to order them by categories (like Yahoo) or by connectedness (like Google). Alternatively, you could provide some other metric of goodness, perhaps by looking at various HTML tags to get clues about the contents of the page; for example, a hit in the title of a web page might be ranked higher than a hit in the body of a web page, or any combination of these methods.

- Learn a bit about database query optimization. While we are doing full text searches and not relational database searches, you might find some inspiration for techniques to make your searches more efficient here.

- Explore the notion of *data mining* to produce a list of related pages even when those related pages do not match the search string. You can use text data mining, also known as *text analytics*, to support such a feature. Be warned that data mining is an open research topic that can consume arbitrary amounts of time, but the basics should be approachable.

  The easiest way to support this *related pages* query is to use *cosine similarity*, which is a common distance metric used in classic information retrieval. The idea is that a page can be represented as a high dimension vector with each *meaningful* word representing a different dimension and the value of the dimension being the frequency of that word. To find related pages, simply look for the distances among all the vectors and choose the k nearest vectors using a distance metric; cosine similarity is often used as the distance metric, but any distance metric can work with varying results.

  When determining similarity, however, we are relying on the rarity of certain words to provide differentiating abilities. Therefore, it is critical to magnify these differences to provide meaningful separation. Popular techniques include stop-lists, which remove common words such as *the*, *and*, *a*, *I*, etc., and frequency analysis, which removes words that are common in the corpus of documents being considered. You may also want to stress certain types of words, such as verbs and nouns, or you may wish to explore more advanced natural language processing (NLP) or information theoretic ideas.

In selecting a method to enhance the performance of the queries, be careful to account for the efficiency of these queries, which is usually the most difficult part. It's often too expensive to find the *best* match, so consider investigating fast approximate schemes and the use of acceleration structures.

If you do attempt to do text mining, you may also find it helpful to build at least a basic visualization tool to map the vectors from R^n to R^2 or R^3 space. These tools can be invaluable for providing insight into the structure of the documents.

There is also the closely related, but more advanced notion of *concept mining*, which would allow your system to accept a query and find pages related by the concept of the query, rather than its text or boolean operation. The idea is that the web pages that you have crawled are full of information—they consist of far more than the corpus of their words—but the difficulty is in extracting this information in a reasonable way.

For example, a query for American Mustang, should be able to provide at least two groups of pages: those related to the horse and those related to the car. In the former case, the system would ideally find all pages surrounding the concept of Mustang horses—issues such as land management, populations, history, etc. In the latter case, the system would ideally find information about the Ford Motor Company, the car's history, current sales trends, industry reviews, etc. Identifying and differentiating between such concepts is an active research topic, though a few basic pioneering commercial applications exist.

The general approach is to first annotate the page according to word families (sets of words with similar meanings, as would be found in a thesaurus); this step usually works by clustering word groups together using a tool such as Princeton's WordNet. This step reduces the number of words in the language to a smaller (but still large) number of clusters. You can then attempt to use techniques analogous to text data mining over these clusters. For better accuracy, *Bayesian models* are often utilized.

7 What to turn in

You will do this assignment individually. Submit you report and program in the usual manner.

Source Code. Turn in any source files relevant to your project, including, at a minimum: `WebCrawler.java`, `WebIndex.java`, `WebQueryEngine.java`, and `CrawlingMarkupHandler.java`. You are allowed to modify any and all files as long as you have the required `useWebIndex` and `query` methods, and your `WebCrawler.main()` saves an index to "index.db" when given valid command line arguments. Make sure all project source code (including tests) is in the `src` directory in the appropriate package directory.

Report. Your report will be very important. Since we give you tremendous freedom in design, you should explain your design and analyze your design decisions, as well as meeting all of the usual expectations of a report.

Acknowledgments. This assignment was inspired by a similar assignment in Cornell's equivalent course, CS312. Many thanks to Walter Chang, Andrew Dreher, Arthur Peters, Ashlie Martinez, and Josh Eversmann for their improvements to this assignment.