CS160 - Assignment 2
Due: Friday Sept. 25, 6pm

For the next step in our IR system we're going to be adding functionality to do boolean queries. For our purposes a boolean query consists of an expression. An expression can be:

- a term (i.e. any non-whitspace character sequence). For example *test*, *calpurnia*, *the*
- a negated term: any term preceeded by a !. For example, *!term*, *!calpurnia*, *!the*
- <expression> AND <expression>. For example, *brutus AND calpurnia*, *brutus AND calpurnia AND !caeser*
- <expression> OR <expression>. For example, *brutus AND calpurnia OR caeser*, *!caeser OR brutus AND banana*

Any multiterm query MUST be connected with either an AND or an OR, so "pomona college" is not a valid query, but could be expressed "pomona AND college".

We will be building on top of the code that we used last time. For simplicity, I'm asking that you all start from the same base code that I've provided at "/common/cs/cs160/assign2/". This will greatly simplify the headaches on your side and the grading on my side. You should be able to swap out your code at some later point for the code I'm providing you. If you forgot how to import the code into eclipse, see the write-up for assignment 1.

You may work with a partner if you'd like. You **MUST** both be there when you are working on the assignment (either coding or writeup) and you may only use one computer, i.e. I want you to do pair programming. When you

submit your write-up, make sure both people's names are in there.

Before starting this assignment read through the **entire** document. Although I've given you some freedom, for certain aspects I have been explicit about how you should implement them. At the end I've included some helpful hints and tools that may be useful. If there is any ambiguity or question about what you are being asked to do, ask the me to clarify.

For this assignment, I'm giving you a bit more flexibility about how you develop the code. In the skeleton code, I've provided a few classes that you'll need to fill in. Details about those classes are below. I've also provided a class "BooleanSearch" that will allow you to run queries against your system once it's built.

1. Data

   Your implementation will be expected to run semi-efficiently on the entire TDT corpus that we used for the first assignment ("/common/cs/cs160/data/tdt-corpus.text_only"). However, for testing purposes, I suggest you use the sample data set from the first assignment ("/common/cs/cs160/tdt-corpus.text_only").

2. What to implement

   Below are descriptions of the classes you must implement. In each case, I have included a skeleton of the class in the code I provided you and have defined which public methods you must define.

   (a) Posting list representation: PostingsList.java

   Each term in our dictionary will have an associated postings list which stores the document IDs for all the documents that word occurs in. The PostingsList class will store these. You will have one PostingsList object for each dictionary entry.

   Your implementation must use a singly linked list to store the docIDs. You will have to implement this yourselves, but this should be a review from data structures :) I'm asking you to do this both as an exercise, but more importantly, this makes our life much easier when performing merges.

Besides storing the data, the PostingsList class also provides functionality to andMerge (AND), orMerge (OR) and not (!) postings lists. Make sure that the postings lists you return from these operations are valid postings lists, that is, they contain unique docIDs that are in sorted order.

(b) Index generation and query processing: Index.java

The Index class will build the index, store the actual index and provides an interface to query the index.

Recall that the index provides a mapping from terms to postings lists, i.e. the dictionary. For now, you may use a hashtable to store this mapping.

- index construction
  To construct a new index, you create a new Index object and pass along a DocumentReader. You **must** use sort-based index construction to build your index, where you collect term/docID pairs, sort them, then make a final pass to generate the index/postings lists.
- query processing
  Given a text query, you should return a postings list with docIDs corresponding to the answer to the query (which may be an empty set). Query processing has two stages. First, you must process the query and figure out what it's trying to say. For our assignment, we will **not** be doing any token processing/normalization on the query and will query whatever terms the user provides. You may do this processing however you like, but one approach is to make a new class that represents the possible query term entries (i.e. a term, a negated term, an AND and an OR).

  Once you've parsed the query, the second step is to generate the result set. For this assignment, we won't worry about query processing ordering, so you may process the query in whatever order you like. I suggest starting at the end of the query and moving your way forward in term pairs. Use the methods you wrote for PostingsList (e.g. andMerge) to accomplish this task.

3. Hints/Comment

- You're code should be fairly efficient and the running times should be on the order of those discussed in class. For the entire corpus, it should only take a few minutes to tokenize and build the index. Once the index is built, queries should be answered almost immediately (I timed my implementation and they're on the order of a 5ms).
- You may test this on whatever data you like, but I'd suggest playing with the sample corpus to start with and then when you feel like it's working, try it on the entire TDT corpus.
- Some of these things can be tricky to get right. The best way to debug is to test each part individually. Once you're sure one method/class is working then you can test the larger system.
- Watch the corner cases! It's very easy to leave off the last item on a postings list or similar such issues. Think about these situations and make sure to test for them.
- You can change your code to just handle single term queries or queries with a single AND (or OR), which again can be useful for checking functionality one piece at a time.
- Look at the java.util.Comparable interface when you're thinking about sorting
- If you break up the sample file into 5 individual files named 0 through 4, then you can check your answers using grep.
- It's likely you'll run into memory issues and may have to increase your memory beyond 512M. If you find things are running slow you can use "top" or a similar program to see what your memory footprint is and then increase the VM arguments as in the first assignment.
- I added a toString() method to the Document class that prints out documents in a nice format.

4. What to turn in and how to turn it in

   - What to turn in:
     - A "jar" file of your code, which should contain all classes required to get your code working, including the original files I provided, the two new classes noted above as well as any supporting classes you need. See the assignment 1 writeup for details on creating a jar file.

- A text file with the following information:
  (a) Name(s)
  (b) What was the most challenging part of this assignment?
  (c) How long did it take you?
  (d) When did you start?
  (e) We used a hashtable for implementing the mapping from terms to postings list. Using lowercasing, how many empty entries are there in the hashtable after you read in the index? Roughly how much wasted memory does this correspond to?

- How to turn it in

  See the course web side for details (it's the same procedure as last time). Make sure to make a separate folder that contains the jar file and your text file and then copy this whole folder over (some of you didn't do this or accidentally copied the contents instead of the folder).