CS201 - Assignment 6, Part 2
Due: Wednesday April 9, , at the beginning of class

OH, HI; I'M HERE FROM THE INTERNET.
WHAT ARE YOU DOING!?
GLUING CAPTIONS TO YOUR CATS.
RRRR

`https://xkcd.com/262/`

For this assignment we're going to build a basic search engine. I've provided you with some starter code, but you're going to build an inverted index, which does most of the work behind the scenes for modern search engines. **Read through this entire handout before you start working on the assignment**. Make sure you understand what you're supposed to be doing, etc. If you have questions, come talk to me sooner than later.

You may (and I'd encourage you to) work with a partner on this assignment.

Search engine basics

Inverted Index

The key data structure that allows search engines (like google) to identify pages that match a query quickly is called an inverted index. An inverted index is a mapping from words to the documents that contain those words, which is called a "postings list". For example, take the following three documents:

```
Document 0: a b c d
Document 1: a b d
Document 2: d f g
```

For simplicity I'm using letters for word, so the first document has four words in it, 'a', 'b', 'c' and 'd'. The inverted index for this set of documents would be something like:

```
a: 0, 1
b: 0, 1
c: 0
d: 0, 1, 2
f: 2
g: 2
```

Notice that now, if we did a search for the word 'd', we can very quickly go to the entry for 'd', get the postings list, which gives us the documents that 'd' occurs in (in this case 0 and 2).

Executing queries

Most real search engines allow you to enter more than just one work and would then return the documents that contain *all* of those words. For example, take the query "a d". What we'd like to return documents 0 and 1, since both those documents have both words. We can calculate this from the inverted index by first getting the postings list for 'a'. Next, we get the postings list for 'd' and AND it with the postings list for 'a. To AND two postings lists, we iterate through the lists and generate a new postings list that has *only those documents that occur in both lists.* In this case, documents 0 and 1.

This can be generalized to handle multiple words easily. For example, take the query "a b c d". We start by first ANDing the postings lists for 'a' and 'b' together. This will give us the postings list 0, 1. We then can take this postings list and AND it with the postings list for 'c', which would give us just 0. We can continue to merge the current postings list with the postings list for the next word in the query. In this case, it would be 'd', which would still give us just the document 0 and we're done.

If the postings list are stored in sorted order, the AND operation can be performed fairly efficiently with just a single iteration through each of the postings list simultaneously (Hint: Think about the `merge` method from merge sort).

Implementation Details

For this assignment you're going to be implementing two classes, the `PostingsList` class and the `Index` class to support postings lists and an inverted index. I've provided you with starter code at:

`http://www.cs.middlebury.edu/~dkauchak/classes/cs201/assignments/assign6/part2/`

Below are the details of what you need to implement for each class.

`PostingsList`

The `PostingsList` class will be used to store a postings list. A postings list will be stored as a singly-linked list of integers (holding the document ids, like above) with both a head and a tail reference. The class will only have three public non-static methods:

- `addDoc`: takes an `int` as a parameter and adds that integer to the *back* of the postings list. You can assume that the `addDoc` method will be called with increasing document ids so the result postings list will be in sorted order. You cannot, however, assume that the document IDs passed in will be unique and must make sure not to add them multiple time. Since they will be called in increasing order, however, this will just require you to check against the last thing in the post list, which should be fast (and easy). This must be an $O(1)$ run-time method.
- `getIDS`: returns an `ArrayList<Integer>` of the ids (i.e. integers) stored in this postings list. This will be useful for debugging and for printing out the documents that match a query.
- `size`: returns the number of ids stored in this postings list. This must be an $O(1)$ run-time method.

In addition to these three methods, you must also implement two `static` methods for merging postings list. Putting these methods in the `PostingsList` class will allow better access to postings list data and will make your life much easier.

- `andMerge`: takes two `PostingsList`s as parameters and returns a new `PostingsList` that represents the AND of these two postings lists (as described above). Neither of the input `PostingsList` should be modified.
- `orMerge`: Sometimes, when you're querying a search engine you're interested in articles that might contain one word OR another word. This is called an OR query. This function takes two `PostingsList`s as parameters and returns a new `PostingsList` that represents the OR of these two postings lists, that is, all of the documents that occur in *either* posting list.

  For example, if we called `orMerge` with the posting list for 'a' and 'd' above, we would get back: 0, 1, 2, since 'a' occurs in documents 0 and 1 and 'd' occurs in 0, 1 and 2. As another example, if we called `orMerge` with the postings list for 'c' and 'f' we would get back a postings list with 0 and 2 in it.

You may also need to implement some `private` helper methods and you will very likely need to implement another class to represent the nodes in your linked list (something like `LinkedListNode`).

`Index`

The `Index` class will store the inverted index, i.e. a mapping from words (`String`s) to postings lists. The class only has two public methods:

- `addOccurrence`: which takes a word and a document ID as parameters and adds that pair to the inverted index. If the word is already in the inverted index, it can just add that document ID to the end of the postings list for that word. If the word doesn't occur yet in the index, you need to make a new postings list and then add the document ID.
- `getPostingsList`: given a word, return the postings list associated with that word. *If a word does not exist in the index, return a new postings list that is empty.*

There are many ways that we can store the inverted index. Given what we've talked about so far, my advice is to store it as two `ArrayList`s. In the first `ArrayList` store all of the words and in the second, the associated postings list for that word. For example, if you use this setup, to get the postings list for a given word you first figure out what index the word occurs at in your first `ArrayList`. You then return the postings list that occurs at that index in the second `ArrayList`.

To write these methods, make sure that you look at the documentation for `ArrayList`. Most of the things you want to do will be already implemented and the class should not require a lot of code.

Build Your Own Search Engine

If you've done all of this correctly, then you should be able to run the `SearchEngine` code to interact with your own search engine! In the starter code I've included two test dataset you can use to try out your search engine:

- `simple.txt`: This file contains the same three "documents" used in the examples above.
- `quotes.50k.txt`: This file contains 50K quotes from various well-known people.

In the `main` method of the `SearchEngine` class there is a local variable called `quotesFile`. Set this to be the name and location of the file you would like to index. Once you've done this, you should be able to run the main method and it will load the specified file into the inverted index and the prompt you for queries.

For example, here is the output from an example run on `simple.txt`:

```
Enter a query (blank to exit): a
------------------------------
a b c d
  --alphabet guy
------------------------------
a b d
  --elmo


Enter a query (blank to exit): a b c d
------------------------------
a b c d
  --alphabet guy


Enter a query (blank to exit): a b c d e
No documents had all those words


Enter a query (blank to exit): e
No documents had all those words


Enter a query (blank to exit): a b c
------------------------------
a b c d
  --alphabet guy


Enter a query (blank to exit): a b d
------------------------------
a b c d
  --alphabet guy
------------------------------
a b d
  --elmo
```

and here is an example query from the quotes data set:

```
Enter a query (blank to exit): computer science
--------------------------------------------------
My background was computer science and business school
so eventually I worked my way up where I was running
product groups - development testing marketing user
education.
```

```
  --Melinda Gates
```

Note that it will take a little while for the index to load (30 second or so) for this full data set, but the queries themselves should execute almost instantaneously. As the index is loading it will count off the number of documents loaded in increments of 1000.

One path to implementation

There are many ways to go about coding this all up. As always, I strongly, strongly suggest an incremental approach, where you work on a single method and then test to make sure it works. If you try and code all of it up and then debug that way, you will have a very, very hard time tracking down all your issues.

Here's one way to go:

1. Write your "node" class and test the basic functionality.
2. Write the `addDoc`, `size` and `getIDs` methods. Test these methods! I've provided you with some `JUnit` tests in the `PostingsListTest`. The first four of these tests should pass. You should also consider writing a few other tests of your own since these tests may not consider all corner cases.
3. Write the `andMerge` method and test with the associated `JUnit` tests (and your own tests).
4. Write the `orMerge` method and test with the associated `JUnit` tests (and your own tests). The `orMerge` should feel fairly similar to the `andMerge` except you have to do a bit more work.
5. Implement the `Index` class methods. Make sure that you understand exactly how you're going to be representing/storing the inverted index and what methods are available for the `ArrayList` class. A few minutes of research and thinking about this class can save you *a lot* of headache. I haven't provided any test cases, but I'd encourage you to write a couple of your own.
6. If everything is working properly, you should now be able to run the `SearchEngine` code. First try out the simple example. Test a bunch of different cases to make sure it works correctly and that you don't get any errors. If that works, move on to the bigger file and try it out.

Extra credit

Right now, our search engine only performs AND queries. If you'd like, you can change the code in the `SearchEngine` class to also support OR queries. To specify that a word should be OR merged instead of AND merged, the user should be allowed to enter the keyword OR before a word. For example:

- `computer OR science`: would find all documents with the word "computer" or the word "science"
- `a b OR c`: would find all documents with the words 'a' AND 'b' in it OR the word 'c'.
- `a OR b OR c`: would find all documents that had at least one of 'a', 'b', or 'c' in it.

There are obviously more complicated types of queries we might want to support like verb1(a AND b) OR (c and d)1, however, we'll stick with the basics for this one.

When You're Done

Make sure you have comments at the top of any file/class that you wrote with with your name and the assignment number.

JavaDoc

All of your methods MUST have JavaDoc style headers. I've included it for many of the methods, but make sure you include it for any additional ones you write.

To group these files into a single file, you need to export your project. To do this:

1. Right-click on the project (on Mac, ctrl+click) and select `Export`.
2. You'll see a number of options. Open up the `Java` folder and select `JAR file` and click `Next`.
3. You should just see your project selected. Below, make sure **only** the following two options are checked:
    - "Export Java source files and resources"
    - "Compress the contents of the JAR File"
4. Click on the `Browse...` button and pick a location to save the output file. Give the file a name like "kauchak1.jar", where "kauchak" is your last name and "1" is the assignment number.
5. Click `Finish`.

If all went well this will generate a single `.jar` file wherever you picked to save it.

Submit this JAR file as assignment number "5.2" via the online submission mechanism on the course web page.