Claremont McKenna College
Computer Science

CS 62 **Handout 4: PS 5** **October 15, 2015**

This problem set is due **11:55pm on Sunday, October 25**.

Be sure to include a comment at the top of each file submitted that gives your name and email address.

Submit the following files to the **Assignments** area on **Sakai**. Multiple submissions are allowed within the time limit. Please do **not** submit `.class` files or Eclipse project files.

```
ArrayList.java,
PageScanner.java,
Readme, and
other files that make up your solution.
```

Problem 1

A partial implementation of `ArrayList.java` is given in the lecture notes area. Let's use the one that uses an inner class. Your task for this problem is to complete the parts left in blank (Look for the string `Fill_in` in the file.). Submit your completed implementation.

Problem 2

A solution for this problem may be considered a starting point for building an integrated web browser and search engine.

Create a `PageScanner` class as an ADT that does the following:

- It reads a file (web page) and provides a mechanism by which the tokens (words) in the file can be accessed one toekn at a time. That is, it provides an iterator over the tokens in the file. It skips over common words (defined later) and provides only the words that are not common. That is, the tokens iterated over are only the non-common words in the file.
- It has an `iterator()` method that returns an iterator object that can be used to iterate over the tokens in the file.
- The constructor of this class takes a file name for the web page document that we want to process.
- Using this ADT a use program would read a file and print only the tokens (words) that are considered worth keeping for further processing, e.g., as a file to use for a search engine. Your main for testing this program would read in an input file name (for the web page that you want to process) and an output file name to keep only the valuable words from the file.

Let's see these in more detail.

1. The words we get by using the `next()` method in an iterator should all be a lowercase version of the words from the input file. For example, if the next token is "Apple", the next token returned should be "apple".

2. The net result of the iterator should be such that it should skip over common words such as "a", "and", and "in". The input file `common.in1` will contain the common words. The `common.in1` file provided contains one word per line, but a line could contain multiple words separated by whitespace. Here is the (sample) contents of `common.in1`:

   ```
   a
   and
   are
   been
   an
   did
   down
   in
   its
   have
   of
   the
   we
   to
   where
   ```

   You should assume that the file `common.in1` is large enough that it should be read in only once, and stored without a lot of unused space. I recommend that you store the common words in a structure that is efficient for search because it will be searched a lot. Each search of the common words should take $\Theta(log_2 n)$ time in the worst case. The file `common.in1` may not be in alphabetical order, but the stored common words should be in alphabetical order. As a choice of data structure for this consider using `java.util.TreeSet`. `java.util.TreeSet` keeps its elements in sorted order and returns each element when accessed in $\Theta(log_2 n)$ time in the worst case. That is, if you use this ADT, it will satisfy all the requirements that I mentioned for the common words. We will study this ADT soon, but learning how to use it is very simple because the interface for this ADT is quite similar to that of array lists.

3. The net effect of the iterator should be such that the `next()` method in the iterator should skip over all characters in an HTML tag, that is, all characters between $<$ and $>$. For example, suppose a line in the input file reads:

   ```
   Apples are <a href=appleref.in>delicious apple</a> something we eat.
   ```

   Then, the tokens returned as final output in the output file would be:

   ```
   apples
   delicious
   apple
   something
   eat
   ```

   You may want to filter out the HTML tags from the string containing the input file content before doing anything else to the string. You can read the entire file into a string and start processing the string. To read the entire file named "test.in" into a string do one of the following depending on which operating system you are using.

```
text = new Scanner(new File("test.in")).useDelimiter("\Z").next();
text = new Scanner(new File("test.in")).useDelimiter("/Z").next();
```

4. Some parts such as punctuations in the input file will be ignored. In fact, you may use the punctuation characters along with whitespace characters as delimiters for the `split` method in `String` class as you split a string into an array of substrings. For this problem, as delimiters you may use one like this:

   ```
   someString.split("[ ,.;:?!\n\f\r\t\"]+");
   ```

5. Your program needs two file names: one for the input file that contains a web page and another for the output file that will contain the output tokens, one word per line. Your program may read in the file names from the keyboard or hard-wire them in your main. In either case, provide a readme file describing how to run your program.

6. In what data structure would you store the words (tokens) that are worth keeping, namely the non-common words? Let's think about how you might process the input file (web page file) in what steps. Here is a possible approach. You might read the entire input file (web page file) into a gigantic string. You can then remove tags from the string. You can then split the resulting string using delimiters into an array. You can then go through the array and add the non-common words into a data structure while skipping the common words. What should that data structure be? If that data structure is one that stores the words in an efficient manner for access while providing an iterator, that would be good! Hmm... I think we know one that does all that!

Incidentally, a scanner like `PageScanner` is essential for a search engine because the relevance of a document is based on the words the document contains.

System Test 1: If the two input file names for this problem are `test1.in` and `test1.out` and if the file `test1.in` consists of:

```
Apples are <a href=apple.in>delicious apple</a> something we eat.
```

then the contents of `test1.out` will as shown as before.

System Test 2: If the two input file names for this problem are `test2.in` and `test2.out` and if the file `test2.in` consists of:

```
In the long history of the world,
Only a few generations have been <a href=granted.in>granted</a> the role
Of defending <a href=freedom.in>freedom</a> in its hour
of maximum danger.
```

then the contents of `test2.out` will be:

```
long
history
world
only
few
generations
granted
role
defending
freedom
hour
maximum
danger
```

Notes: Include a file named `Readme` which explains how to run your program and any other information that a reader (grader) should know about your solution.

Note that the `PageScanner` class here happens to have a similar name as the `java.util.Scanner` class, but they are certainly different classes.

I have included test input files and `common.in1` in the [given] folder.

What to hand in: Hand in `PageScanner.java` and `Readme` along with any other files that make up your solution. That should include test input files. Use the two I used in the handout and create another of your own (a somewhat large one, I suggest) and include the new one too in your submitted files.