Compuer Science 62
Assignment 6

Due at noon on Friday, March 12, 2010

This assignment is a variation of Assignment 2, however this time will be making a few additions to our frequency counting procedure to make it more efficient. The assignment is designed to give you experience in three areas: implementing a "filtering iterator," working with linked structures, and adapting data structures.

1 Getting started

Open a new Eclipse project and copy the starter files from `/common/cs/cs062/assignments/assignment6/` into your source directory.

2 Filtering Iterator

For Assignment 2, we used the `WordScanner` class to iterate through words in a text document. Recall that this class returns in order all of the sequences of contiguous characters (roughly words) in the file. In the second assignment, we had to post-filter these to meet our specification. Another option for doing this is to filter them in a new iterating class, called a "filtering iterator."

Write a class `LongWordScanner` that `extends WordScannner` that is a filtering iterator. You can read more about filtering iterators in Section 8.5 of Bailey's book. The new class should have the header

```
class LongWordScanner extends WordScanner
                      implements Iterator<String>,
                                 Iterable<String>
```

and should support the same four constructors from the `WordScanner` class (either via inheritence or overridden in your class).

The behavior of `LongWordScanner` will be the same as `WordScanner` *except* that it will filter out one-letter words *and* it will lowercase all of the words. The `LongWordScanner` will need to override the methods `hasNext` and `next` to conform to the filtering behavior of `LongWordScanner`. The `iterator` method will just return a reference to the current object; be sure you understand why.

For inheritance, remember that you can call the original methods from the base class. To call the constructors use `super(...)` and to access methods of the inherited class use `super.method(...)`. This is important when you want to modify the behavior of a method, but you need the original method to accomplish your task.

Be careful... that your class works correctly "at the boundaries," including situations in which there is white space or a one-letter word at the very end of the source text. One way to deal with these challenges is to keep track of what the next *valid* word to be returned should, in essence looking ahead one word, assuming it's available.

When you're done, test out your solution on a number of different files before moving on to the next step. You can write your own "Tester" class with a `main` method and some simple test files to test it out.

3 Frequency Counting

As before, we want to maintain a list of word-integer pairs, in which the integer represents the number of times the word has appeared in the source text. We will be making two changes to the first assignment. First, we will be keeping the counts in a linked list (what benefit does this have?). Second, rather than just keeping one list of word-integer pairs will keep 26 such lists, each one holding all the words beginning with a specific letter of the alphabet. The multiple lists are analogous to the alphabetized tabs on a printed dictionary. For example, if we wanted to find the frequency count for the word "dog" we'd look in the linked list associated with the letter 'd', i.e. the list at index 3. (what benefit does this storage system have?).

We have written the class `WFNode` which will be the "node" of your linked lists. It has instance variables of type `String`, `int`, and `WFNode`. The idea is that a node is an element of a linked list that holds the frequency count for a particular word. It has accessor and mutator methods for each instance variable. The `toString` method is customized for the kind of output required below.

- Create a class `WFTable` which will store all of our counts as an array of 26 singly-linked lists, where the lists consist of `WFNode` objects. The ith array element will be an *alphabetized* list of the nodes of all the words that begin with the ith letter of the alphabet. For this exercise, you are to maintain the linked lists directly. Do **not** use any of the classes from `structure5` or the Java API to store the table. To accomplish this, you may either implement your own `WFLinkedList` class and then store this in your array or only use the `WFNode` class and implement all the appropriate functionality within the `WFTable` class. To make it explicit, you *must* store the `WFNode`s in your linked list in sorted order.

  Your `WFTable` class must support the following methods

  ```
  public WFTable()
  public add(String wd)
  ```

  A few useful notes:

  - Remember that `char` is a numeric type. If `ch` is a lowercase character, then `ch-'a'` is the (zero-based) position of `ch` among the lowercase letters.
  - One place where you may have difficulty is in maintaining the linked lists correctly. A possible approach is to maintain an "empty node" on the front of each list, which will alleviate having to keep worrying about if the head is null. Keep it simple. Whatever your problem, the solution is almost certainly "better code" and not "more code."

- Once you have your linked list construction working appropriately make your `WFTable` class `implement Iterable<WFNode>`, which will require you adding one additional method. The iterator returned by your class should traverse the `WFNode`s by decreasing `frequency` with ties being broken by alphabetical order. Fortunately, the `compareTo` method in `WFNode` is customized for exactly this—by large integers in decreasing order and secondarily by strings in increasing order.

  There area number of ways to tackle this problem:

- Iterate through all of your data in the table and put it in an ArrayList or similar structure. You can try and keep it in sorted order or an easier way is to just add everything and then sort the data. `Collections.sort` may be useful for this strategy.
- Implement a method like `removeGreatest` to extract the elements in the desired order in `WFTable` and implement the `Iterator` interface within `WFTable`. We don't want to modify our underlying data, so you will also nee do implement a `clone` method for `WFTable` that creates a complete copy of the table then returns this copy for the iterator.

4 Generating the data

Create a class `WordFrequency` which contains a `main` method that takes a single filename as a command-line argument (we'll talk about this in class) and then uses `LongWordScanner` and `WFTable` to determine the frequencies of words in the specified file. The code will be a close parallel to the `main` method in Assignment 2. Do not modify the `WFTable` after all the words have been inserted. Print out the words and their frequencies, one to a line, in order of decreasing frequency (ties broken by alphabetical order), as shown below. The words should be left-justified in a column with twelve characters, and the integers should be right-justified in a column with five characters (the `toString` method of WFNode should be useful).

```
the            14
mary           13
lamb           12
and             9
 ...
snow            1
white           1
```

Submission

Write your code carefully and clearly. Follow the specifications precisely, because your classes may be tested in ways that you do not suspect. As always, we will look at your Javadoc documentation. Submit the final versions of these files, with the names specified, in the usual way.

Grading

You will be graded based on the following criteria:

criterion	points
LongWordScanner	4
overall functionality	10
used linked lists appropriately	1
appropriate comments (including JavaDoc)	3
style and formatting	2
submitted correctly	1