Layla Oesper
Fall 2017

This assignment is due by **10pm on Friday November 10** and is worth 24 points.

1 Goals

The goal of this assignment is to implement a particular tree structure and to use it to store words and their counts.

2 Setup and Instructions

You may work with a partner of your choice on this assignment or by yourself. If you would like a partner, but have not found someone to work with, send me an email no later than 5pm on Sunday and I will work on finding you a partner. You can also use Piazza to try to find a partner as well. Lastly, if you are working with a partner, remember that Pair Programming is about learning to write better code, it's not about trying to make the process as efficient as possible.

Download `HW11.zip` and unzip it into the directory that you've created for your work on this assignment. There is a Java class called `WordCloudMaker.java` (originally written by Sherri Goings, with a few modifications by Jeff Ondich and me - thanks for sharing!). There is also a sample book (`Robbers.txt` - a play by Friedrich Schiller) and a stopwords file called `Stopwords.txt`.

3 Building a Word Cloud

You've probably seen word clouds like the one below before which I generated from the text from "Anne of Green Gables" by Lucy Maud Montgomery. The neat thing about word clouds is that words are displayed in a size proportional to the number of times they are used in the text on which the cloud is based. (Note that very common words, also known as stopwords, are typically not included in the cloud. Otherwise, all English word clouds would be dominated by "the", "and", "a", "in", etc.) In this assignment, **you'll implement binary search tree** to count words and their counts in text, allowing you to make your own word clouds.

3.1 Overall Program Structure

Your program will come in three main pieces:

- A class called `WordCountMap` consisting of a binary search tree that maintains a list of words and their counts. (`WordCountMap` will make use of two very small classes called `Node` and `WordCount` described below.)
- A class called `WordCounter` containing your main program and any support methods it needs. The main program will be in charge of opening a text file, counting all of the words it contains, and producing one of three types of output.
- A class `WordCloudMaker.java` (provided to you) that will transform a list of word counts into HTML that shows a simple word cloud.

3.2 WordCountMap

First you'll build a class `WordCountMap` where each object of this class will represent a **binary search tree** where each node of the tree contains a word and the count of the number of times that word appears. The tree should be organized according to the alphabetical ordering of the words. What instance variables might you want for a binary search tree? Specifically, this class will need to make use of two other classes. The first is a `Node` class which should look something like the following:

```
private class Node {
    public String word;
    public int count;
    public Node left;
    public Node right;
}
```

The exact details of the `Node` class are up to you, since it's a class that will be invisible to users of `WordCountMap`. You may add other (private) methods to this class, but it should be a nested class within your `WordCountMap` class.

The other class you will need is a `WordCount` class. This small class will allow you to create an object that contains a word and its associated count. Please make this class in a separate file `WordCount.java`. This is a relatively simple class that should look something like the following (plus whatever methods/constructors you choose to add):

```
public class WordCount implements Comparable<WordCount>{
    public String word;
    public int count;
}
```

In particular, you'll want to use these exact instance variables and please make them **public**, as the `WordCloudMaker` class will depend on accessing them directly. Since this class implements the `Comparable` interface, you'll have to make sure to implement a `compareTo` method. In particular, you should use the instance variable `count` to do this comparison. If you aren't sure what this method should do, take a closer look at the Javadoc description of it in the `Comparable` interface.

Your `WordCountMap` class will need to implement the following three methods:

```
/**
 * If the specified word is already in this WordCountMap, then its
 * count is increased by one. Otherwise, the word is added to this map
 * with a count of 1.
 */
public void incrementCount(String word) {
    ...
}

 /**
 * Returns an array list of WordCount objects, one per word stored in this
 * WordCountMap, sorted in decreasing order by count.
 */
public ArrayList<WordCount> getWordCountsByCount() {
    ...
}

 /**
 * Returns a list of WordCount objects, one per word stored in this
 * WordCountMap, sorted alphabetically by word.
 */
public ArrayList<WordCount> getWordCountsByWord() {
    ...
}
```

You may certainly create other methods to help you implement these methods in your class. Please do make sure that you implement these methods exactly as described as the grader and I will likely build some testing scripts that use these methods. Lastly, note that `ArrayList` should be the built-in Java implementation of a List that uses an array to store the data.

3.3 WordCounter

This class will contain your main program and any support methods it needs. The main program will be in charge of opening a text file, counting all of the words it contains, and producing one of three types of output. Specifically, your main method here will parse the command-line arguments to support the following three ways of running the program:

1. You should be able to run the program using the following commands:

   ```
   java WordCounter alphabetical [textFileName]
   ```

   This will print out a list of words and their occurrence counts, one word per line, each line consisting of a word, a colon, and the word's count. This list will be sorted alphabetically by word. For example, the output might look like the following:

```
loesper$ java WordCounter alphabetical alice.txt

abide:1
above:1
accept:1
accepted:2
accepting:1
access:10
accessed:1
...
```

2. You should be able to run the program using the following commands:

   ```
   java WordCounter frequency [textFileName]
   ```

   This is the same as the the alphabetical case, except the words will be sorted in decreasing order by frequency (or count). For example, the output might look like the following:

   ```
   loesper$ java WordCounter frequency alice.txt

   alice:168
   project:87
   little:59
   gutenbergtm:56
   out:53
   down:51
   work:50
   up:44
   very:42
   its:41
   ...
   ```

3. You should be able to run the program using the following commands:

   ```
   java WordCounter cloud [textFileName] [numberOfWordsToInclude]
   ```

   This one will print HTML to the screen (standard output) containing a word cloud based on the code I've given you (read through the code and comments in `WordCloudMaker.java` to figure out what methods you should call to do this). A typical invocation of the cloud generator would be:

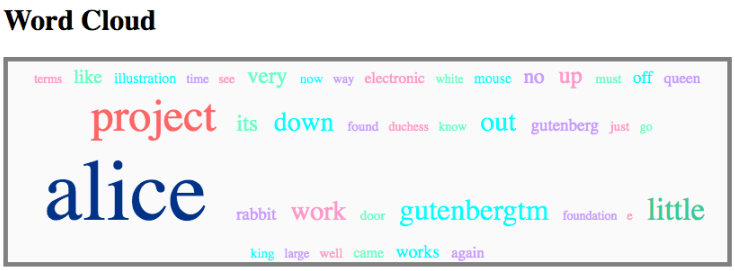
   ```
   loesper$ java WordCounter cloud alice.txt 40
   ```

   This would generate the word cloud based on the 40 most common non-stopwords in `alice.txt`. (If `alice.txt` contains fewer than 40 non-stopwords, then the cloud will just use all the words.)

   Note, since HTML is hard for us as humans to interpret, you might want to redirect the output into a text file which you can open using a browser. For example,

```
loesper$ java WordCounter cloud alice.txt 40 > AliceOutput.html
```

will create an html file `AliceOutput.html` that when you open in a browser looks like the following:



You must make sure to remove any extraneous print statements from your code if you do this! Otherwise, your word cloud may not display correctly.

Other Notes, Requirements and Suggestions

- Make sure that `WordCountMap` implements a binary search tree based on the alphabetical ordering of the words. You should not use other data structures included with Java to do this. You must implement the tree structure directly yourself.
- I've provided you a file called `Stopwords.txt` that contains a list of common English stop words. You should write your code so that any word which appears in this list is not included in the results that you produce. How exactly you choose to do this is up to you, but you should think about the efficiency of your code for this portion
- At least one of your `getWordCountsBy` methods in your `WordCountMap` should use a tree traversal to get the list of words and counts in the correct order. For the other method, you can use built in Java methods if you'd like.
- This is the first assignment where I'm asking you to use command line arguments. We talked briefly about this at the beginning of the term, but this is a really useful think to know how to do, which is why I'm requiring it here. If you don't remember this well, look back at the code I gave you for HW2 or in section A.10 of your book for more information.
- I do not care if you treat "alacrity" or "Alacrity" as the same word or different words for the purpose of this assignment. However, note that `stopwords.txt` only contain words in lower case. As such, when determining whether a word is a stopword, case should be ignored.
- When reading from a text file you should strip punctuations from words. One way to do this is to use the String method `replaceAll`. In particular, this method can use something called a regular expression to identify and then remove certain patterns of characters. If you want to find and remove any character in a string `s` that is NOT a letter, you could use the following line of code:

```
s = s.replaceAll("[^a-zA-Z]","")
```

- You do NOT need to make any changes to `WordCloudMaker.java`. But if you want to make changes; go ahead. The same applies to `stopwords.txt`.
- I included `Robbers.txt` a play by Friedrich Schiller obtained from Project Gutenberg. You may want to download other books to test.
- Keep modularity and encapsulation in mind as you design your code. Make sure your comments indicate what the instance variables that you choose represent.

4 Submission and Grading

Create a zip file of the following, to be handed in via Moodle:

- `WordCountMap.java`.
- `WordCounter.java`
- `WordCloudMaker.java`
- Any other files used by your code.

Only one partner from each pair needs to submit the assignment. Specifically, put these files into a directory named `[your_last_name_your_partner's_last_name]HW11`, zip this directory, upload it to Moodle. For example, if my partner was Schiller my directory would be named `OesperSchillerHW11` and the resulting file would be `OesperSchillerHW11.zip`.

This assignment is worth 24 points.

- 18 points for the assignment requirements.
- 6 points for comments, style and design.

5 Extra Challenge?

There are a number of things you can do with this assignment if you are looking for an extra challenge. Here are a few suggestions.

- Implement a self balancing tree (although if you do this, please do submit your code for your regular binary search tree as well). We'll talk about 2-3 trees in class, but if you want to learn about something new AVL trees are a good choice.
- Add some additional functionality to your WordCountMap class. Maybe a `decremenCount()` method. Maybe add a different kind of tree traversal.
- Play around with the `WrodCloudMaker.java` code to change the look of the cloud you create.

Start early, ask lots of questions, and have fun! Layla, the lab assistants, and the prefect are all here to help you succeed - don't hesitate to ask for help if you're struggling!