CSc 225 Assignment 4:
Heaps, Trees, and Word Frequency

Due date:

The submission deadline is 11:55pm on Wednesday, July 1st, 2020.

How to hand it in:

Submit your `WordFrequencyHeap.java`, `WordFrequencyBST.java`, and `WordFrequencyReport.java` files through the Assignment 4 link on the CSC225 conneX page.

Assignment Description

For this assignment, you will be counting words in text files and keeping track of the frequency (number of times) each word is found in a text file.

To do this, you will be populating a heap and a binary search tree with entries. An entry consists of a **word** (string) and the number of times that word is found in the input file, the **frequency** (int).

In Part 1, you will be implementing a max-heap, prioritized by word frequency.

In Part 2, you will be implementing a binary search tree, sorted alphabetically.

In Part 3, you will be use your implementations from Part 1 and Part2 to explore the word frequencies of different input files.

Starter files and usage

Please download the a4.zip file containing all of the starter files for this assignment. The file can be found in the Resources > Assignments > a4 section on ConneX.

The `A4Tester.java` file has tests for each part in separate methods. I've provided some sample output based on my solution for Part 3. Further instructions on how to use your implementation to analyze any text file are provided in further detail in the Part 3 section.

Part 1: Max-Heap Implementation

For Part 1, you will need to implement the methods specified in the `PriorityQueue.java` interface.

You will be implementing of max-heap of Entry objects. Read the **`Entry.java`** file to get an idea of what an entry is, and the `testHeapOperations` method in **`A4Tester.java`**

In order to complete the implementation, I suggest writing some helper methods. Remember the Heap Properties must be maintained; refer to the lecture videos, slides, and in-class worksheet if necessary.

For Part 1, the only file you need to add implementation to is **`MaxFrequencyHeap.java`**. I suggest adding some further tests in the `testHeapOperations` method in **`A4Tester.java`** until you are satisfied your implementation is correct.

Part 2: Binary Search Tree Implementation

For Part 2, you will need to complete the implemention of the `handleWord` and `getFrequency` methods specified in **`WordFrequencyBST.java`**. Notice that these two methods have a lot in common with some common BST operations.

For `handleWord`, the idea is that whenever a word is seen for the first time when reading the contents of a dataset, it should be added to the BST (in the correct location). At this point, the frequency associated with that word should be one (as it has only been seen once). Any subsequent time that word is seen in the dataset, its frequency value should be incremented. The `testBSTOperations` method in **`A4Tester.java`** might help you get a better idea what is expected from this method.

The `getFrequency` method just searches through the BST for a given word, and returns its associated frequency. This would be useful to see how many times a worst was found in a dataset.

For Part 2, the only file you need to add implementation to is **`WordFrequencyBST.java`**. I suggest adding some further tests in the `testBSTOperations` method in **`A4Tester.java`** until you are satisfied your implementation is correct.

Part 3: Word Frequency Report

For Part 3, you will use your completed BST and Heap implementations to perform a word frequency analysis on a text file. A Scanner has been setup for you to read through each word in the file. As each word is scanned, it will be "handled" in the BST (by either inserting or updating an entry). After reading through the input file, the BST should contain entries for each word that include its number of occurrences in the input file.

The Heap is then be populated with the tree entries, and then used to determine which words were found the highest number of times in the input file.

I have provided three sample input files to test with (sample1.txt, sample2.txt, and sample3.txt). For each input file, you will determine the top 5 most common words (overall), the top 5 most common words with at most n letters, and the top 5 most common words that begin with a given letter.

Specify the input file to examine through the command-line when executing the program. For example, to examine sample2.txt, we would type: **`java A4Tester.java sample2.txt.`**

Here is my sample output for **sample3.txt**:

```
Overall most frequent:
{"the", 40575}
{"of", 19656}
{"and", 14789}
{"a", 14400}
{"to", 13777}

Most frequent words with 6 or more characters:
{"marius", 1358}
{"valjean", 1112}
{"himself", 1085}
{"cosette", 1013}
{"little", 973}

Most frequent words starting with a c
{"cosette", 1013}
{"could", 676}
{"come", 556}
{"child", 470}
{"can", 433}
```

For Part 3, the only file you need to add implementation to is **`WordFrequencyReport.java`**.
I suggest adding some further tests in the `testFrequencyReport` method in **`A4Tester.java`** until you are satisfied your implementation is correct. In particular, I would be sure you solution works for other minimum word lengths and also on words that begin with a letter different than ‘c’.

If fewer than five words meet the given criteria, the locations in the `top5` array should be left as `null`.

Good luck!