Data Structures in Java: Final Exam Capstone Project
Common Word Finder

1. Objective

Your goal is to use three of the data structures you developed this semester to create a program that finds the n most common words in a document. BSTMap, AVLTreeMap, and MyHashMap implement the MyMap interface. So, with proper object-oriented programming technique, specifically the use of polymorphism, you can create any one of these data structures and refer to it from a MyMap reference. Though all three classes implement the same interface, their methods are implemented very differently, leading to different execution times on the computer. You will time your program's execution to see how each of the data structures performs.

2. Problem

You will develop a command line Java application to find the n most common words in a document. The name of the public class must be CommonWordFinder, and it must reside in a file named CommonWordFinder.java.

Parsing Command Line Arguments

The program will take in either two or three command line arguments. If the number of arguments supplied is incorrect, the program will print the usage message

```
"Usage: java CommonWordFinder <filename> <bst|avl|hash> [limit]"
```

to stderr and exit in failure.

The program will first verify that the input file exists. If it does not, the program will print the message

```
"Error: Cannot open file '" + args[0] + "' for input."
```

to stderr, where args[0] contains the name of the file the user is attempting to process, and exit in failure.

The program will then try to parse the command line for which data structure the user wishes to use. Valid strings are "bst", "avl", and "hash". Any other string including those with different capitalization is considered invalid, causing the program to print the message

```
"Error: Invalid data structure '" + args[1] + "' received."
```

to stderr, where args[1] contains the data structure string name, and exit in failure.

At this point, the program will now attempt to process the limit command line argument, if it exists. The limit must be a positive integer. If the command line argument is invalid, the program will print the message

```
"Error: Invalid limit '" + args[2] + "' received."
```

to stderr, where args[2] contains the string form of limit, and exit in failure. Since limit is an optional command line argument, it may not be supplied. When the limit is not specified on the command line, the limit will be set to 10.

Parsing the Input File

After validating all command line arguments, you'll need to instantiate either a BSTMap, AVLTreeMap, or MyHashMap. You **must** create a reference to the interface rather than the class itself, as in:

```
MyMap<String, Integer> map = new BSTMap<>();
```

No credit will be given if your map variable is not of type MyMap. The key-value pairs are String-to-Integer, where String is the word and Integer is the number of times the word is found in the document.

When parsing the input file, follow these simple rules for breaking up the text into words. Lowercase letters (a-z) and single quotes (') are legal characters for words. Hyphens (-) are legal too, as long as they are not the first character in a word. Uppercase letters (A-Z) are converted to lowercase letters before putting them into a word. Words are separated by end-of-line characters and spaces. Every time your program parses a word for the first time, it is inserted into the map with a count of 1. If it has been seen before, the count associated with it is incremented by 1.

If an I/O error occurs as the reading takes place, the program will print the message

```
"Error: An I/O error occurred reading '" + args[0] + "'."
```

to stderr, where args[0] contains the name of the file the user is attempting to process, and exit in failure.

Displaying the Output

The first line of output will display the following:

```
Total unique words: <some positive integer>
```

Then up to <limit> words and their counts will be displayed. If the limit exceeds the total number of unique words in the file, then all the words in the file should be displayed. The output must be printed meticulously. Starting with 1, each line begins with the number of the word and a period. The number must be right-aligned to the width of the largest number. For example, if 1000 words are do be displayed, the first line will have 3 spaces, the number 1, and then a period. Line 10 will have two, line 100 will have one, and line 1000 will have no leading spaces.

The word in all lowercase letters appears after the period, left-aligned with one space between the period and the word. If two lowercase words have the same count, the words should be alphabetized in the output. Finally, the count of the words appears at the end of each line, with one space between the longest word and the count. Use System.lineSeparator() to properly get the system-dependent String for new line characters. Every line of output ends with a visible character, not whitespace.

3. Sample Input / Output

Copy the text below into a file called LoremIpsum.txt.

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc ac diam ac
massa sodales vulputate eget vel velit. Nam pellentesque nulla sit amet
ipsum fringilla, at dapibus sapien convallis. Sed lectus est, maximus nec
```

```
euismod dictum, venenatis eu justo. Ut vestibulum mattis nulla vel
rhoncus. In placerat pulvinar orci sed facilisis. Nullam facilisis velit
lacus, vel mollis mauris placerat in. Sed sit amet neque non magna
convallis commodo vel nec enim. Maecenas fermentum, enim ut ultricies
pellentesque, quam nulla convallis tellus, non eleifend arcu urna in
sapien. Maecenas sed fermentum nisi. Maecenas id maximus est. Aliquam
pellentesque dapibus ullamcorper. Duis pulvinar, quam vitae euismod
congue, mi felis imperdiet nisl, placerat vulputate lectus erat in tellus.
```

When the command below is run from the command line

```
java CommonWordFinder LoremIpsum.txt hash 200
```

the output is as follows:

```
Total unique words: 73
 1. in           4
 2. sed          4
 3. vel          4
 4. amet         3
 5. convallis    3
 6. maecenas     3
 7. nulla        3
 8. pellentesque 3
 9. placerat     3
10. sit          3
11. ac           2
12. dapibus      2
13. enim         2
14. est          2
15. euismod      2
16. facilisis    2
17. fermentum    2
18. ipsum        2
19. lectus       2
20. maximus      2
21. nec          2
22. non          2
23. pulvinar     2
24. quam         2
25. sapien       2
26. tellus       2
27. ut           2
28. velit        2
29. vulputate    2
30. adipiscing   1
31. aliquam      1
32. arcu         1
33. at           1
34. commodo      1
35. congue       1
36. consectetur  1
37. diam         1
38. dictum       1
39. dolor        1
40. duis         1
41. eget         1
42. eleifend     1
43. elit         1
44. erat         1
45. eu           1
46. felis        1
47. fringilla    1
48. id           1
49. imperdiet    1
50. justo        1
51. lacus        1
```

```
52. lorem        1
53. magna        1
54. massa        1
55. mattis       1
56. mauris       1
57. mi           1
58. mollis       1
59. nam          1
60. neque        1
61. nisi         1
62. nisl         1
63. nullam       1
64. nunc         1
65. orci         1
66. rhoncus      1
67. sodales      1
68. ullamcorper  1
69. ultricies    1
70. urna         1
71. venenatis    1
72. vestibulum   1
73. vitae        1
```

4. Requirements

You must use the supplied implementations of the BST, AVL tree, and hashmap. You **must not modify** these files in any way. You must write all your code in CommonWordFinder.java. We will copy the source for the data structures into same the folder as your source code when grading your submission. If you wish to write additional classes to help you encapsulate the data you process, put the non-public class inside CommonWordFinder.java.

You may and likely need to import from the following packages:
java.io.*;
java.util.Arrays;
java.util.Iterator;
You may not use any of the Java Collections including ArrayList, LinkedList, HashMap, TreeMap, etc. Basic arrays, however, are perfectly fine. Again, do not modify any of the existing code given to you. It is possible to complete this project without any additional methods or fields.

You are expected to comment your code. Use Javadoc for methods and inline comments where appropriate.

You may use the Internet to help with various parts of this assignment, including reading from a file, using Iterators, etc. Be sure to cite the source in an inline comment above the lines of code you reference. You may not consult with another human. You may not use openai.com. Since the project is open-ended, there are many different ways you can approach solving the problem. If your code is too similar to someone else's in the class, you will earn an automatic F in the course. Let's do everything we can to avoid this scenario!

5. Timing Your Work and Evaluating the Results

After you are confident you are getting the correct output for smaller files, try processing the Bible.txt supplied in Canvas. It should have 13680 unique words. Time it from the command line in Cygwin or the Linux/Mac terminal as follows:

```
time java CommonWordFinder Bible.txt bst 20000
time java CommonWordFinder Bible.txt avl 20000
time java CommonWordFinder Bible.txt hash 20000
```

Answer the following questions in a readme.txt file.

What data structure do you expect to have the best (fastest) performance? Which one do you expect to be the slowest? Do the results of timing your program's execution match your expectations? If so, briefly explain the correlation. If not, what run times deviated and briefly explain why you think this is the case.

6. Submission

Create a zip file containing only CommonWordFinder.java and readme.txt and upload it to GradeScope.