Assignment 4 Binary Search Trees

Due Date: Friday March 30, before 4:00pm
Weight: 6%
This assignment should be completed individually.

Description

This assignment is a reworking of the task you had to complete for assignments 1 and 2.

Essentially you have the same task: read file of English text and print out various statistics and lists on the frequency and length of words in the text. You should be able to use much of the code you wrote for assignments 1 and 2.

The finished program will be run like this:

```
cat input1.txt | java A4.jar > myoutput1.txt
```

I will provide sample input and output files for the program. The output from your program must be *exactly* as shown in the examples. It can be very difficult to eyeball your code and see if it matches the required output exactly, so you need to let the computer do the comparison for you.

If you were given `input1.txt` and `output1.txt`, and you ran your program as above,

```
diff (cat output1.txt) (cat myoutput1.txt)
```

in Powershell or

```
diff output1.txt  myoutput1.ttxt
```

in Linux or MacOS will tell you if your progam is producing the correct result. No output from `diff` means all is well.

The specific requirements for this assignment are as follows.

1. Instead of a linked list you must use a BST of your own implementation. Call the class `BST`.
2. Your program should operate in five general steps.

(a) Read the words from the file and create a BST ordered by the natural (alphabetical) ordering of the words. Stop words will be dealt with differently in this exercise. Rather than skip them, you will add them into the tree just like any other word.

(b) Delete the stop words from the tree. Note that there is a new list of stopwords. I have provided the Java array definition for the list. This is contrived but it means you have to implement `BST.delete()`!

(c) Create another tree ordered by word frequency, from most to least. Only include words that occur more than twice.

This requirement means that your BST must be able to handle different orderings, other than just the natural ordering. Think carefully about how you will implement this!

(d) Create another tree ordered by word length from longest to shortest.

This will require the creation of another `Comparator` for the `Word` class. Remember that a BST does not have duplicate entries so your `Comparator` must distinguish between different words of the same length. e.g. *peace* and *faith* are both 5 letter words but the `Comparator` must see them as not equal.

(e) Print out the required results. See the sample output files for what is required.

- The number of stop words is the number of stop words from the list that occured in the text.
- Whenever you print a `Word`, print the word, its length and the number of times it occured, separated by colons.
- For the 20 least frequent words, DO NOT create another tree. Use the frequency tree that you have already created. You should only have three trees in your system.
- Remember that the optimum height of a BST is $log_2(n)$ where n is the number elements in the tree.

3. When printing items from your BST, you cannot have any print statements in the BST class. You must provide a method `public Iterator<T> iterator()` that returns an iterator over the tree. Use that to do any output or processing of elements in the tree.

   The main class, `A4`, should have no knowledge of the inner workings of the `BST`. In particular it should have no references to `BSTNode`.

4. You will find that a `Stack` or `Queue` will come in handy when implementing your `BST`. Use the ones we implemented for `A3`.

5. The `BST` you write must be generic. That is, it could be used in another application without change.

Stack Size

With larger text files you may find yourself running out of stack space.

When the Java virtual machine starts it allocates two memory areas: a stack and a heap. The stack is a typical program stack and is used to store frames — method invocation information such as parameters, local variables, and return values. Each time you perform a method call, recursive or not, a frame is placed (pushed) on the stack. It is popped when the method returns.

If your program is multi-threaded, a stack is created for each thread.

The heap is a run-time pool of memory and it is where object instances are created as the program runs. Whenever you execute a `new` the memory required to store that object is allocated from the heap. The garbage collector will periodically go through the heap and deallocate any space that is being used by an object that no longer has a reference to it.

All kinds of detail about the Java Virtual machine is available from https://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

Usually the default sizes for the stack and heap are sufficient but if you are getting errors you can increase them with the -X options when you start the Java virtual machine.

```
java -Xms100M -Xmx200M -Xss50M A5
```

The example above will set the inital heap size (`Xms100M`) to 100 megabytes, the maximum heap size to 200 megabytes (`Xmx200M`) and the stack size to 50 megabytes (`Xss50M`).

Warning:

Before messing around with stack or heap sizes be sure that you don't have an infinite loop or a recursive method without a base case to terminate it. Running out of memory is more likely from that than a legitimate need for more stack or heap space.

All of the test files ran on my ancient machine in less than 10 seconds each with no change to the stack or heap size.

Testing

I will supply some test input and output files for you to use. These files do not constitute extensive testing. You will need to do more testing to ensure all cases are covered.

I will be evaluating your program with test data that you have not yet seen.

What to Hand In

Hand in a single file, `A4.jar`.

Submit this to the Blackboard drop box provided. The jar should contain:

1. All of your `.class` and `.java` files.
2. A class called `A4`.

Do *Not* include your test input or output files or any other files.

Grading

The assignment will be graded according to this rubric:

```
Total (54)

1. Documentation (15)
(a) Java doc standards           0/3
(b) Format of Code               0/3
(c) Meets other rules/guidelines 0/3

2. Testing (30)
(a) Test file1 0/10
(b) Test file2 0/10
(c) Test file3 0/10


3. Follows implementation specifications (15)
This includes;
-- Has correct classes: A4, BST, Word, (Stack, Queue, SLL if needed).
-- Code is compact and efficient.
-- BST class is generic and encapsulated.
```

I will run your program with the command line given above on three text files and compare your output to the specifications.

Outcomes

Once you have completed this assignment you will have:

- Implemented a BST.
- Compared a BST to a SLL.
- Understood the value of a BST and the importance of balancing the BST.