Assignment 3
Avenger Favor
(with Binary Search Trees)

Due Date: Wednesday March 22, before midnight
Weight: 7.5%
This assignment may be completed in groups. *Each group must submit a statement of contribution that clearly states the contribution from each team member.*

Description

This assignment is a reworking of the task you had to complete for assignments 1 and 2. Once again, you will be using what you have learned in your COMP2503 to help Jon with data analysis on popularity of Avengers. You should be able to reuse a good portion of the code you wrote for assignments 1 and 2. The behaviour of your program is very similar to the previous assignments, with the following differences:

- You cannot use `ArrayList` or `Linked List` to store your Word objects, you must use a Binary Search Tree of your own implementation, with the methods to enable your program to produce the required output.
- In this assignment, there are some more statistics expected in the output. You need to report on the some statistics on the binary tree data structure build by your program (see the instruction below).

Your finished programs will be run like this:

```
cat input.txt | java -jar A3.jar > output1.txt
```

`input.txt` is any text file containing English text. Your program `A3` will read the standard input, and writes to standard output. 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.

Your program reads from the standard input one word at a time, cleans it up, and uses a `binary search tree` to keep track of the mentioned avengers and how many times their names or their aliases have occurred in the text. The roster of avengers' aliases and last names is the same as in assignment 1. You can use your own implementation, or the example solution posted for A1 as the starting point, and make sure you follow the implementation details below.

Implementation Details

- Use the drive class called `A3` provided in the starter code package. It will have the `main()` method, in which you should instantiate an `A3` object and then call run, and the `A3.run()` that does the bulk of the processing.
- Input is all from standard input. Create a `Scanner` object and use that for all input:

  ```
  Scanner input = new Scanner(System.in);
  ```

  All output should be to standard output. Use `System.out.print();` and `System.out.println();` for all output.
- You have an `Avenger` class from `A1`. You will need to augment the `Avenger` class with an additional member that stores the "mention index" for each avenger, which keeps track of the order in which avengers are mentioned. You can reuse the implementation for the `Comparable` interface, the `equals` method, and the two `Comparator` from `A1`. You will need to implement an additional new `Comparator` to order objects based on their "mention index".
- **Creating your Binary Search Tree (BST):** You should create a `binary search tree` class called `BST`. Your `BST` class must use a generic type for the data that it stores. However, similar to A2, you should restrict the generic type to those that implement the `Comparable` interface.
  - Your BST must implement all the required method for adding and finding items in the data structure. It should also implement a public `size` method that returns the number of items stored in the tree, and a public `height` method that returns the height of the tree. The main class, `A3`, should have no knowledge of the inner workings of the BST. In particular it should have no references to a BSTNode object.
  - Your BST must implement the `Iterable` interface and provide an in-order traversal. For example 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 print the items in a tree in the required ordering, or to process items the tree in the required ordering. You will

    find that a Queue will come in handy when implementing the iterator for your BST.
- After you have created your `BST` class, you should store the avenger data in different instantiations of the `BST` class for each required ordering.

Hint on creating a BST class with alternative ordering: Recall that in a binary search tree, all items *smaller* than the root are stored in the left subtree, and all the items *greater* than the root are stored in the right subtree. In-order traversal of a tree will produce the items ordered from *smallest* to the *largest*.

To create a `BST` class with alternative orderings, you can extend the same idea you used for creating `SLL` objects with different orderings in `A2`. You can create a private `Comparator` object and a private `compare` method as part of your `BST` class, and two different constructors, one without any input arguments for the case when the `BST` is to be ordered based on the natural ordering of the objects, and another constructor with a `Comparator` object as input argument to create alternative orderings.

- You should create four `BST` objects.
  - The first `BST` object you create should be ordered by the natural (alphabetical) ordering of the avengers' alias. You should add new `Avenger` objects as they appear in the input stream, or update their frequency in the existing object in the tree if they have appeared before.
  - Barton wants to disappear and attend to his family, so he has asked Jon to remove all appearances of his name or his alias from this data processing task. Therefore, after you have read all the input stream and constructed the alphabetical ordered BST, you are asked to delete the `BSTNode` that contains the `Avenger` object for Barton, before constructing the other ordered threes or before printing the output.
  - After you have removed Barton from the alphabetical tree, you can start creating the remaining three BSTs by adding Avengers via an in-order traversal of the alphabetical tree. The second `BST` you create should use a new `Comparator` for ordering the `Avenger` objects in the order they appeared in the input stream.
  - The third and the fourth `BST` objects should be ordered by the most frequent `Comparator` and the least frequent `Comparator` respectively. To create the these trees, `Avenger` objects must be added from an in-order traversal of the `BST` that stores them in alphabetical order.

    Important note: I want the `Avenger` objects to be added to the last three trees in alphabetical order (i.e., based on an in-order traversal of the alphabetical tree). If you add them based on the mention order or any other order, your trees will potentially end up with a different shape, and therefore, will have a different height.

Important Note: *You should only ever have one instance of each `Avenger` object, therefor, each `Avenger` object will be part of all four lists.*

- Use these four lists to print out your results as in Assignment 1.
  - The total number of words in the input. This total should not include words that are all digits or punctuation.
  - The total number of avengers mentioned in the input.
  - The list of avengers in the order they appeared in the input stream.
  - The four *most* frequently mentioned avengers ordered from most frequent to least. In case of ties, then in ascending alphabetical order of alias.
  - The four *least* frequent avengers ordered from least to most. In case of ties, then in ascending order of last name length, then again in case of ties, in ascending alphabetical order of last name.
  - All mentioned avengers ordered in alphabetical order of their alias (ascending order).
  - The height of the tree ordered by the order avengers were mentioned in the original input, with the following format:
    `Height of the mention order tree is : x (Optimal height for this tree is : y)`
  - The height of the alphabetical tree with the following format:
    `Height of the alphabetical tree is : x (Optimal height for this tree is : y)`
  - The height of the most frequent tree with the following format:
    `Height of the most frequent tree is : x (Optimal height for this tree is : y)`
  - The height of the least frequent tree with the following format:
    `Height of the least frequent tree is : x (Optimal height for this tree is : y)`
    Hint: Remember that the optimum height of a BST is $\log_2(n)$ where n is the number items in the tree.
- **Testing and example files:** The input files are the same as for Assignment 1. But the output files have the additional statistics about the height of the trees. There are sample input and output pairs named `input1.txt`, `output1.txt` etc. Use these to determine if your program is running correctly. You should use the `diff` command to compare your output to the samples.

  ```
  diff (cat sampleout.txt) (cat myout.txt)
  ```

  If `diff` gives no output, your program is working correctly.

  You should also devise a test plan and create a series of test files to fully exercise your program. For example, what about an empty file? or one with only some punctuation in it? Or a single word repeated 100 times? or ...?

  I will be evaluating the program against files you have not seen yet that will be meant to *test* your code.

Documentation and Coding Standards

Your program should follow all of the coding rules and guidelines outlined in the document provided. In particular, include `Javadoc` style comments for all classes and substantial methods.

What to Hand In

Submit a single file, `A3.zip` to D2L. The package must include the following:

1. An executable jar file, that I will use as follows to test your program:

   ```
   cat input.txt | java -jar A3.jar > output1.txt
   ```

   Make sure the jar file is executable, you can test it with the sample input and outputs provided, otherwise you will loose the mark on testing.

2. All of your source files (`.java` files). Your source must include a class called `A3` which has a `main()` method, and a class called `Avenger` which implements the Comparable interface, and all supporting classes that you have implemented, including your implementation of the Binary Search Tree class.
   Make sure you include your java source files, otherwise I cannot mark your code and you will lose the mark associated with the documentation and style components.

3. A statement of contribution that reflects on the team work and clearly states the level of contribution from each team member. Submissions without a statement of contribution cannot be graded.

Grading

A detailed grading rubric is provided at the end of this document.

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 with iterator and alternative orderings.
- Compared a BST to a SLL.
- Understood the value of a BST and the importance of balancing the BST.

Rubric

1. Documentation
   (a) Java doc standards 5
   (b) Format of Code 5
   (c) Meets other rules/guidelines, including submission of detailed statement of contributions 5
2. Testing
   (a) Test file1 10
   (b) Test file2 10
   (c) Test file3 10
3. Follows implementation specifications 30
   - Has the following classes with the required functionality: A3, Avenger, BST, Iterator for BST. (4)
   - Implements and uses a Comparable generic BST class. (7)
   - Implements BST class with alternative orderings. (8)
   - Implements the Iterable interfaces for the BST class (7)
   - Code is compact and efficient. (4)

If you would like to explore a little further: 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/se12/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.