Text Analyzer

Abstract:

The idea for this project is to come up with a simple text analyzer. The project is intended to replicate various functions such as generating word count, a list of distinct words, etc. The main purpose of this exercise is to solidify my understanding of various data structures such as sets and maps, as well as use some basic functionalities of input/output file manipulations.

Goal:

1. Obtain total word count
2. Obtain total unique word count and corresponding frequencies of each word
3. Check for membership of a word using a HashSet of Strings
4. Filter out “non-words” from the text to get more accurate results

Method:

1. Copy arbitrary text from the internet into a text file and put it in an appropriate directory.
2. Generate a HashSet for searching purposes. The size of the HashSet provides the number of unique words.
3. Generate a TreeMap that will map unique words to their corresponding frequencies.
4. Use a filter to eliminate words that contain any symbol other than letters.
5. Using an iterator, print the TreeMap on a separate text file using PrintStream.

Structure:

I came up with a main class that contains the main method to be run, and a separate class that contains methods for filtering out tokens read from the scanner that are not true words. My initial approach was to put all the methods as static methods in the same class as the main method. The filtering method was put in a separate class out of need for organization, and not because it served a specific purpose. The structure will be explored further in the *Edits* section.

New Material:

1. Generic methods:

To combine the uppercase, lowercase and numbers array in the WordTrim class, instead of simply combining them using constants, I decided to explore writing a method that uses generics. I soon found out however, that generics and arrays don’t mix well together, but through some online resources I managed to create a method that utilizes generics to combine the three arrays together. I am not quite sure whether such a method has room for further generalization. The method that sorts TreeMaps by value (which I modified slightly from the URL in the reference section) was my first encounter with generic methods, and was the inspiration for my unnecessarily complicated approach to combining the arrays.

1. Anonymous classes:

While exploring the method from the reference section, I saw the use of an anonymous class for the instantiation of a comparator class. This approach allowed me to be more concise in my code.

Edits:

I realized that my initial approach to the project was very narrow, and did not make use of the object oriented programming nature of Java. Although I learned how to clean up the main method by writing new methods, they are still simply static methods within the main and thus lack the potential for further generalization. The Following are changes made to the first version.

* To utilize the OOP concept of Java, I created a class *Text* that has a TreepMap, HashSet, ArrayList etc. as parameters. In addition, the clutter of methods that used to be in the main class is now transferred into the *Text* class, allowing for a much cleaner version of the main method.
* New sets of basic methods such as getCount(), remove(), contains() and variations of them have been added.
* To practice more OOP concepts, I created an abstract class Filter, as well as various children classes that allow for various filters for the Trim method.
* A critical bug that has completely ruined the functionality of the Trim method has been fixed. It turned out that after removing a word in an ArrayList, I forgot to decrease the index by 1, thereby ruining the entire indexing and processing of the text.
* Used method overloading on the Trim method in order to make the method call more concise. In addition, the Trim method that deals with a single string has been modified to use recursion rather than a while loop. I think this method is slightly more effective than reassigning the object word using the substring method. However, because the loop iterates through a constant number, making the method O(1), the difference in efficiency may be negligible.
* I added a method that compares one Text object to another, as well as a method that removes commonly occurring words for more accurate comparison. The method iterates through the maps of both Text object and prints on a separate text file various data on how one objects compares to the other.

Efficiency:

The *test* class contains the main method that runs some analyses based on arbitrary texts. The steps that the main method include, instantiation of object, filtering words, generating a set and a map, and finally, printing it. Each step is analyzed using the big-oh notation, considering N as the number of words in the text.

* Instatiation of *Text* object: O(N), since we are iterating through the text to create an ArrayList.
* Filtering words: O(N), since we are iterating through every word in the text.
* Generating a Set and a Map: O(N), since we are iterating through every word in the text.
* Printing: O(N), since we are iterating through every word in the Map.

The program works overall on O(N) efficiency.

Room for improvement:

The main method runs the code on a Wikipedia article and the resulting analysis file, even with filtering, contains numerous non-words such as “Empire.[n”, meaning a more sophisticated version of a filter can be developed. By combining the filtering and instantiation of the Set and Map, the code could become more efficient as well. I also initially thought of using a HashTable in order to develop a search function, but did not follow through with it.

Reference:

https://beginnersbook.com/2014/07/how-to-sort-a-treemap-by-value-in-java/