Project 3 Report

12/4/20
Kyle Chmielewski
Andrew Heare
Christian Hollar

Introduction

Our goal of this project was to create an index, like you would see in the back of a textbook, for a given text file. Where the index mapped out every word to occur and at what line numbers throughout the text, since our data lacked page numbers. To accomplish this we utilized three different data structures: ArrayList, TreeMap, and HashMap. With the purpose of comparing their performance of handling such a large data set, in order to be able to determine each data structure's strengths and weaknesses.

Approach

First, the group needed to make a hypothesis based on what theoretically should happen when testing and comparing each data structure for efficiency.

Hypothesis: The Hashmap will run faster than the TreeMap when the input file contains smaller amounts of duplicates. HashMap has an add method of complexity O(1). TreeMap has an add method of complexity O(logn). The ArrayList will run the slowest because its add method has a complexity of O(n). For larger amounts of frequency, the HashMap will run slower than the ArrayList and TreeMap because it is not sorted. The HashMap will have a difficult time tracking down keys to add new lines to their individual TreeSets.

Each member of the group was given a designated data structure to test. Each data structure was responsible for scanning in the desired text file. For the ArrayList, each line of the

text file was split into individual words. Then each word in the line was checked to make sure it wasn't a duplicate before creating an Entry object and adding that to the arrayList in sorted order. For duplicate words their line number was added to their existing Entry object. This process was repeated for every line of text read in.

For the TreeMap, each line of the text file was split into individual words. Each word in the line was added to an array of type String. All words were converted to lowercase and punctuation was removed. The words were added to the TreeMap. The addition function checked whether the word existed in the map already. If the word was not in the map, it was added to the map and a TreeSet was created containing the line numbers where the word appeared. If the word was already in the map, the new line number was added to the TreeSet. Each addition of a word to the map was timed individually and a total time was calculated after all words were added.

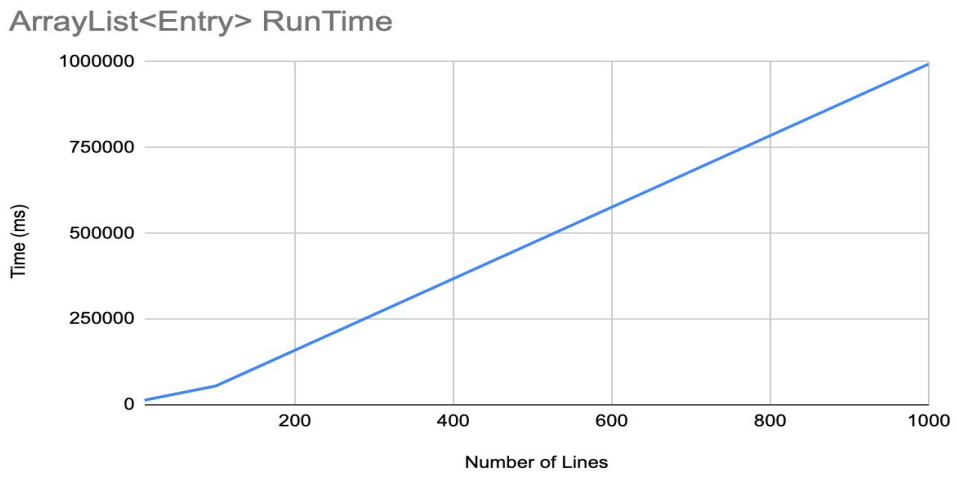
For the HashMap, each line of the text file would be sent to an ArrayList of type String. This ArrayList would be looped and each string would be split. The punctuation would be removed and the casing would be set to lower. Then it would be added to the HashMap. The HashMap would check if the key was already in the Map. If so, it just added the line number to the TreeSet in the value position. If it was not in the HashMap, a new key and TreeSet would be created and added to the HashMap. The add function would be timed for each element. This time would be added to a total time variable which would track the total time of the indexing with the specific data structure. The add function would include checking for the key within the hashmap and the manipulation or creation of the TreeSets.

Methods

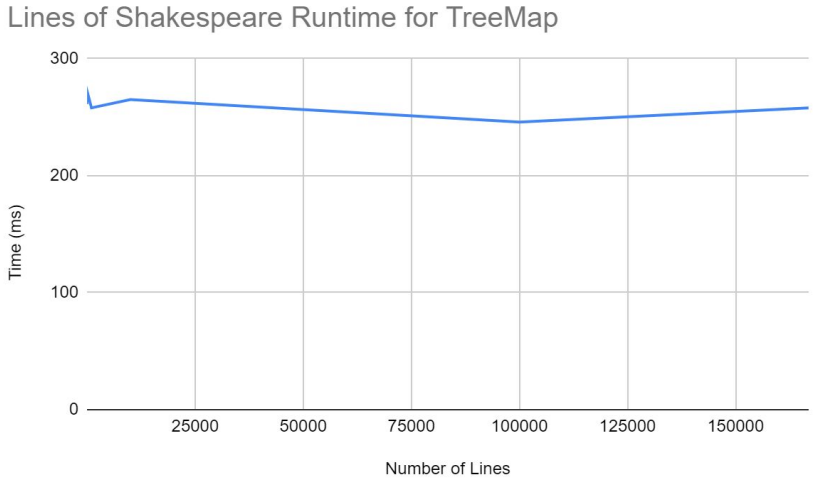
- **ArrayList:** First the **IndexUsingSortedArray** class is initialized with the name of the text file as its parameter, then to make it very simple the constructor initializes the ArrayList, then calls the **createIndex()** method which builds the index for the text file using the file name passed through the constructor as its parameter. Within the **creatIndex()** method the file is read in line by line using a buffered reader, and split into words, which are then all added to an ArrayList. The point of the ArrayList which is named allTheWords is to make it easier to process each word through my if-else complex to determine how I should create the corresponding Entry object. Nevertheless, the **Entry** class is a simple container for data, it holds each word and every line number it occurs on. After the arrayList of the line of words completes through the if-else complex, I clear the ArrayList and move to the next line to start the process over again. This process runs as long as the while condition is met.
- **TreeMap:** The **TreeMap** class reads in a dictionary file when initialized. Each word is copied into an ArrayList of type String. The class contains a **run()** method which reads in the file to be indexed. Each line is split into words and converted to lowercase. Words are added to the TreeMap using the method **addToTreeMap()**. This process is repeated for each line in the file. The **addToTreeMap()** method takes the word and the line number as parameters. The method checks whether an instance of the word already exists in the TreeMap. If so, the line number is added to a TreeSet containing the line numbers for the word. If not, the word is added to the **TreeMap**.
- **HashMap:** The **ReadHash class** is initialized with a string parameter representing the designated file to index. A try-catch block and Scanner tool is used to go through each

line of the .txt file. Each line is added to an ArrayList named “a” of type String. The first method to run is **readInHash()**. **readInHash()** for loops through each String value in “a.” It splits each string into an array of strings. The punctuation is removed, and all casing is set to lowercase. For each array of strings, they are sent to the **addToHashMap()** method along with their line index from the first for loop. **addToHashMap()** checks to see if the key-value exists. If it does, the value TreeSet is retrieved, and the line number is added. Then the replace method is called to replace the previous key and value set with the updated TreeSet. The final method is **sortHashMap()** which sorts the HashMap “h.” This class requires the inner class hashObject. Each hashObject holds a key word and value TreeSet<Integer>. Within this class, the **compareTo** and **toString** methods are utilized. compareTo is used to order hashObjects by their keys. toString prints the key and then cycles through each value of the TreeSet. This is useful in the **sortHashMap()** method. Two lists are created. The first is of type String that contains each of the key values from the hashMap. The second is of type hashObject. MapKeys is looped through. For each loop, a new HashObject is created with this MapKey String and the corresponding TreeSet. Finally, the list of HashObjects is printed to an outside text file named A.

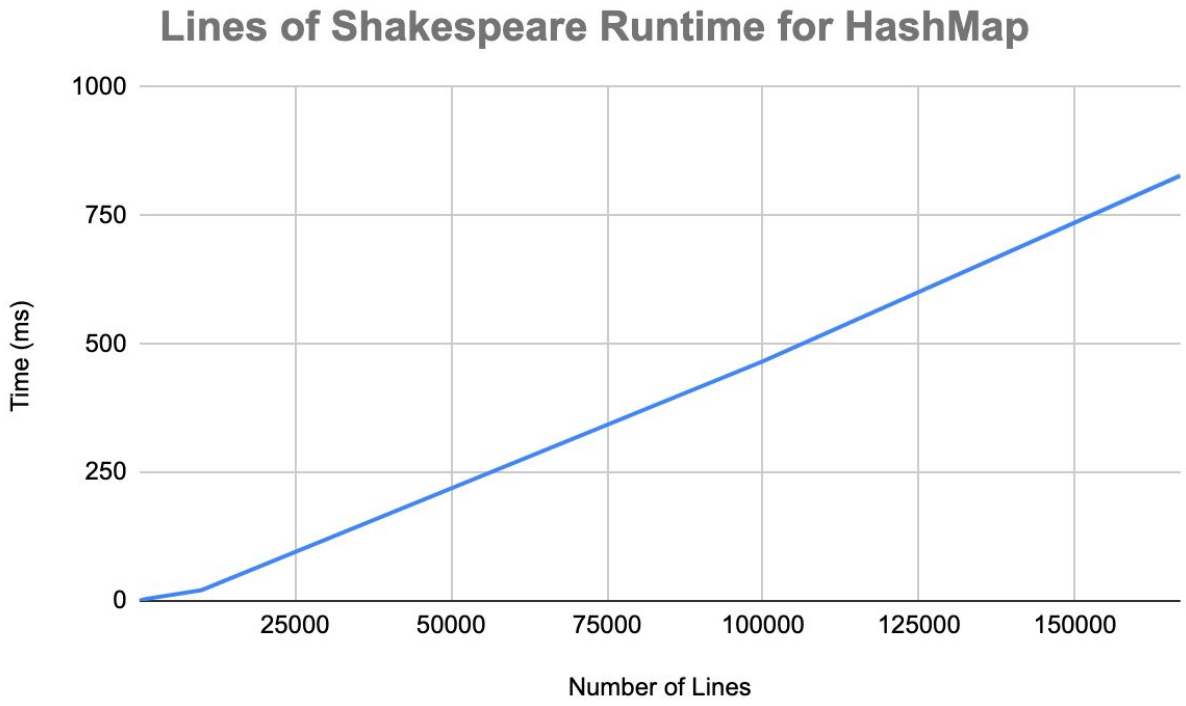
Data & Analysis



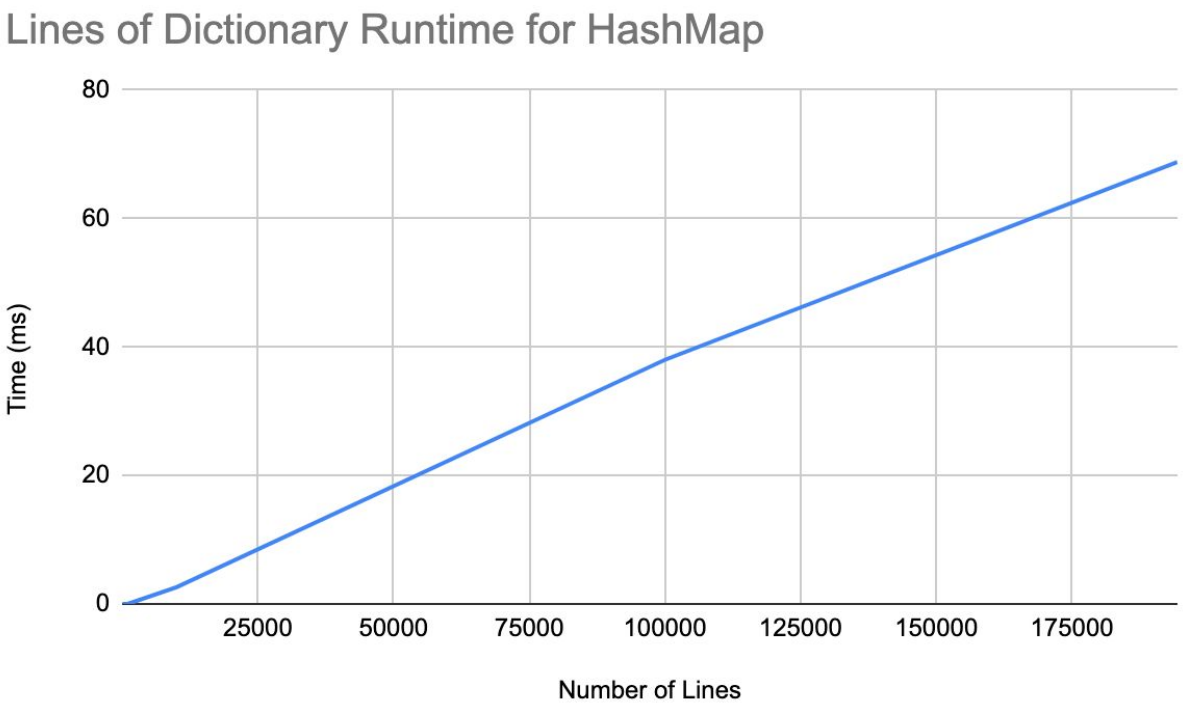
As you can see in the graph using an ArrayList to create an index is incredibly inefficient, as to index only 10 lines of text it took over 13,000 milliseconds. The performance of this data structure got exponentially worse as the size of the data set increased, with an average performance of 55415.2 ms at 100 lines of text, and close to 100,000 ms at 1000 lines. Making it impractical to test for any amount of lines greater than 1000. All in all, for searching through large amounts of data and assigning each of them specific values, the ArrayList is a data structure you should stay away from.



TreeMap offers consistent performance with an average of 259.7 milliseconds required to index the whole file, regardless of the amount of lines processed. This makes it useful for processing very large files but bad for processing very short files. This difference in performance is partially due to the line numbers that are added to a TreeSet when words are repeated, which is slower than adding to a HashSet. Generally, TreeMap will outperform HashMap only for files that are longer than 10,000 lines.



Based on the data, HashMap returned an almost linear curve. Prior to 1000 lines of code, the HashMap performed within 5 milliseconds. At 10000 lines of code, it jumped to an average of 20.8 milliseconds. It grew from roughly a factor of ten after that. There are 166902 lines within the shakespeare.txt file. At the maximum number of lines, HashMap had an average performance of 827.6 milliseconds.



Based on the data, HashMap returned an almost linear curve. Prior to 100000 lines of code, the HashMap performed within 5 milliseconds. At 100000 lines of code, it jumped to an average of 38 milliseconds. It grew to about double the amount of time after that. There are 194434 lines within the English.txt file. At the maximum number of lines, HashMap had an average performance of 68.8 milliseconds.

As the graphs reflect, the HashMap was much more efficient when using the dictionary. The dictionary did not contain duplicates. As a result, the HashMap did not have to keep using replace and editing TreeSets. It simply had to create a new String and new TreeSet<Integer> with a single integer.

Conclusion

While the TreeMap was more efficient in the end, HashMap was quicker for less than 10,000 line inputs. ArrayList was the least efficient regardless of the input or number of lines being evaluated. Both ArrayList and TreeMap produced constant data meaning the runtime was similar regardless of the number of input lines. HashMap produced linearly increasing data meaning the data increased in a straight positive line. TreeMap and HashMap both performed more efficiently with the dictionary ebook. This is because there were not repeating words so the TreeSets did not have to be replaced and edited. In conclusion, when evaluating ebooks holding repetitive words with less than 10,000 total lines, the HashMap should be utilized. For all other situations, TreeMap is the correct option for indexing.

References

ArrayList (Java Platform SE 8), 9 July 2020, https://docs.oracle.com/javase/7/docs/api/java/lang/ArrayList.html

TreeSet (Java Platform SE 8), 9 July 2020, https://docs.oracle.com/javase/7/docs/api/java/lang/TreeSet.html

String (Java Platform SE 8), 9 July 2020, https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

HashMap (Java Platform SE 8), 9 July 2020, https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

TreeMap (Java Platform SE 8), 9 July 2020, https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

Weiss, Mark Allen. *Data Structures and Problem Solving Using Java, Fourth Edition.*

Pearson Education, 2010.