CS 150 - Project 3 Report

Yazdan Basir

May 10th, 2020

1 Introduction

The purpose of this project was to find the ideal data structure when it came to building an index of words for a certain book/text file. As books contain many thousands of words and phrases, the need to find out where in the book specific words appear is important for research and/or analysis purposes. Hence, the demand for indexes. Although indexes are usually situated at the end of a book, for the purposes of this project, a separate .txt file was created in the project folder for each data structure.

For this project, three different data structures were used as the foundations for an index and put against each other and tested for time: an ArrayList (ListIndex), a TreeMap (TreeIndex), and a HashMap (HashIndex).

2 Approach

The approach for testing each data structure was pretty straightforward here. The intention was to build an abstract class that housed the common methods between the 3 different indexes and then set up distinct methods in each index to be able to clearly test each of the steps involved in the process. As shown below, an abstract class named Index was created, housing a few common methods between the three indexes.

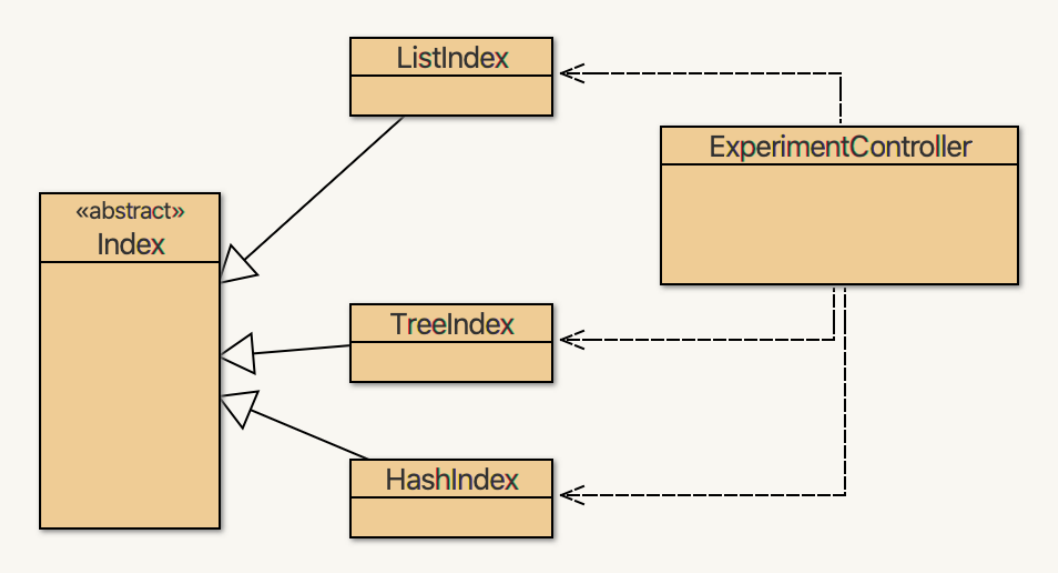

Figure 1: Structure and organization of the classes (without Test Classes)

The first two methods in Index are the abstract methods addEntry() and createOutputFile(). These two need be overridden in each separate index class due to their innate differences. The addEntry() method needs to be overridden because the way to add to an ArrayList, a TreeMap, and HashMap is different. Similarly, the createOutputFile() needs to be overridden because accessing elements the fastest way possible is different for all the structures.

The first common/shared method in Index is readFile(). It reads the given file line by line and calls addEntry() on each word and its corresponding line. The last two methods in Index are for building the dictionary. The createDictionary() method builds an ArrayList and adds each word from the Dictionary.txt file to it. The next method, isWordInDictionary(), checks if a particular word is in the Dictionary ArrayList. It gets called on each word in addEntry() and returns a boolean based on the result. An ArrayList of the Dictionary was built so that a list could be accessed each time instead of having to go through the entire file repeatedly. Binary Search is used to find the index of the word in the Dictionary and this technique of using an ArrayList becomes much faster than accessing the file again and again.

To settle this, both of these variations were tested for time. An experiment was conducted using 15000 words - accessing the file for the word took around 20 milliseconds on average whereas accessing the Dictionary ArrayList for the word took 5-6 milliseconds.

The addEntry() and createOutputFile() methods were overridden in ListIndex, TreeIndex, and HashIndex and they contain other minor helper methods. ListIndex contains an inner class called Entry, which holds a String for the word and a TreeSet for the set of lines it appears on. The Entry class is comparable in terms of String. ListIndex also contains a helper method called find() which performs a Binary Search on the ArrayList and returns whether or not the list contains an Entry of that particular word. HashIndex contains a helper method called sort() which creates a Set from the keySet() of the HashMap and adds all the keys to an ArrayList. The ArrayList is then sorted and the method returns.

For the purposes of demonstration, the Experiment Controller class, where the main() method is, can be run to create a ListIndex file, a TreeIndex file, and a HashIndex file for The Complete Works of William Shakespeare. However, as detailed in the Methods section below, the Experiment Controller was primarily used to test each class and its methods for the time they took.

3 Methods

To get the best possible understanding of the performance of each index, it was decided that three separate experiments would be conducted for the time of reading the input file, creating the output file, and one for both reading the input file and then creating the output file.

At first, a single run was conducted of each process and the time taken for each was observed to get a ballpark figure for the number. Many single runs were tried but the numbers seemed to fluctuate, starting from a high initial count and eventually approaching a constant lower value. This meant that factors like system load were causing the first experiment to return an inaccurate number for the time taken for the process. Thus, 20-30 runs were conducted of each process to figure out the time taken. Even though this was a better experiment and the numbers were much lower now, there was still some inconsistency. This was because the load on the system increased and to avoid this, it was decided that each experiment would be run 180 times to get the absolute, definitive value for the average time taken.

Another aspect of testing was the grouping of tests. As the project required us to test the program on books of different sizes, 8 additional books were selected to test on the program. Along with their approximate word counts, they were as follows: The Divine Comedy (40k), White Fang (75k), The Odyssey (150k), The Republic (250k), Middlemarch (350k), Don Quixote (450k), War and Peace (570k), Les Miserables (800k), and The Complete Works of William Shakespeare (1m). These books would give an accurate representation of the performance of the program. An array of the names of the books was created to be iterated through and for each index, the test was run with the expected output being 9 numbers in increasing order. All these free eBooks were downloaded from the Project Gutenberg website.

However, another inconsistency was noted here. For the provided book, The Complete Works of William Shakespeare, one test was taking around 300 milliseconds whereas it was taking 80 milliseconds when conducted individually and not in a sequence. It was understood that running so many tests consecutively enhanced the system load and obstructed the program from reaching its optimal performance level. For this reason, each test for each book for

each index was done conducted separately and independently of each other with small gaps of time between them to allow the system to cool down. This resulted in a significant decrease in the time taken for the experiment, even though it took longer to conduct, and paved the way for the true performance of the program to be seen.

4 Theoretical Analysis

Before delving into the Data Analysis and having the most efficient index-building process revealed, it is important to understand the 3 data structures being tested are even being tested in the first place. As mentioned previously, they are an ArrayList, a TreeMap, and a HashMap.

The ArrayList inside ListIndex uses the methods add(), get(), and remove(). Both add() and get() have a time complexity of O(1) while remove() has a time complexity of O(n). The TreeMap inside TreeIndex uses the methods put(), get(), containsKey(), all of which have a time of O(log n). Last but not least, the HashMap inside HashIndex uses the methods put(), get(), containsKey(), and the Arraylist used here uses add() and sort(). All the HashMap methods have a time complexity of O(1). As for the usage of memory, a HashMap uses significantly more memory than a TreeMap.

With the information above, we cannot really say what effect moving everything from the HashMap to an ArrayList will have on the final result but without considering that aspect of HashIndex, it is reasonable to expect the fastest performance from it as compared to other two indexes.

5 Data Analysis

Before moving on to the main results of the experiments, we'll look at the output of running the program once. The main() method in Experiment Controller runs the methods in each index 5 times and returns the average time taken for reading the input file and the average time taken for creating the output file. This is shown below.

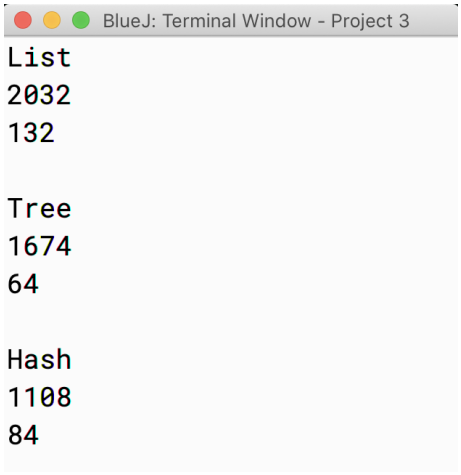

Figure 2: Average Time Taken For Each Index

The 3 output files were then checked against the Expected Output.txt file to see if the results matched. This test was successful and can be seen below:

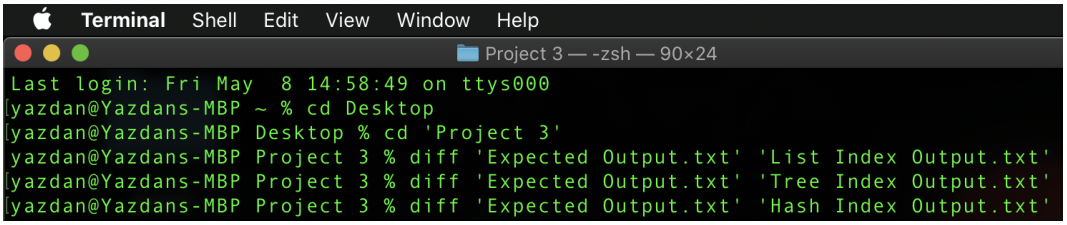

Figure 3: Testing If Each Index Produced The Correct File

As described in the section above, 3 different experiments were conducted on each index for each book. The first was to time the creation of the output file - createOutputFile(). This was the method that created a blank output

file, accessed elements from its corresponding data structure, wrote to that file, and ultimately saved it once it was done.

One thing to point here is that while ListIndex and TreeIndex were adding elements to their data structures in a sorted manner, HashIndex was not. This is why there was a separate sort() method present. However, this sort() method was not factored into the time taken by the experiment because it was tested for time independently and it was found that it was taking a negligible amount of time to run. Not only that, but it would've caused an unnecessary inconsistency in the experiment. So, for this reason, for the ListIndex and TreeIndex, the readFile() method was called and then the createOutputFile() method was called (which was timed). The same was done for the HashIndex but sort() was also called before timing the createOutputFile() method.

The results of all the experiments are as follows:

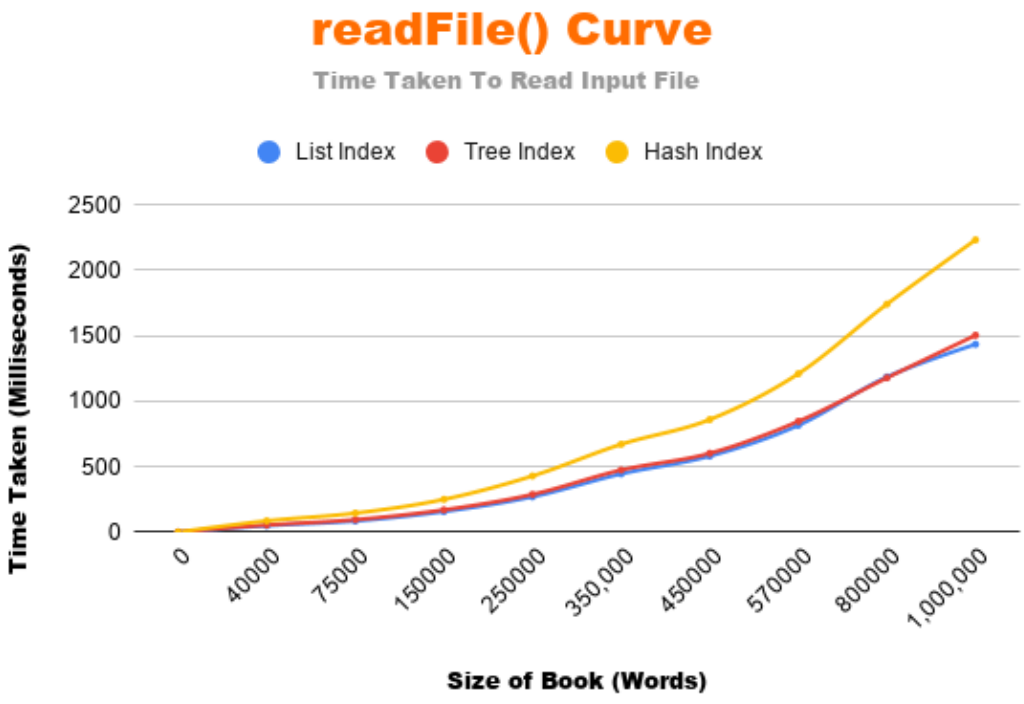

Figure 4: Time Taken For Reading The Input File

The first experiment was about reading the entire file and adding it to the specific data structure. This process involved creating a Buffered Reader, going through every line in the text, splitting it into its separate words, converting them to lowercase, and calling addEntry() on each word to add it to the corresponding data structure.

For ListIndex, the elements are added in a sorted manner (as an instance of the Entry class) using BinarySearch. Adding to the TreeIndex and HashIndex involved adding the word as a key and the line number it appeared on as an element of a set (the value). This is done until all words in the file have been read. The first experiment's result show that the TreeIndex and ListIndex take nearly identical times to read the file. As we can also see, the HashIndex takes slightly longer than both the ListIndex and TreeIndex. Due to the fact that the readFile() method is the same for all 3 indexes, this suggests that adding the elements themselves to a HashMap takes more time than adding elements to an ArrayList or TreeMap.

Another thing to note is that the difference in the time taken isn't consistent throughout. For smaller books, the HashIndex takes nearly the same time as the other indexes but as book size increases, the difference increases as well. While this difference is miniscule at first, it rises to around half a second, three-fourths of a second for the largest book. This data suggests that the TreeIndex and ListIndex perform at a steady rate as book size increases whereas HashIndex does not. This could be down to either the system load affecting the performance of the program as book size increases but it could also be a factor of the very nature of the data structure itself.

The next experiment involves creating the output file itself. This process of writing to the file involves creating a new .txt file, accessing the elements from the data structure, printing it to the file, and closing/saving the file.

For ListIndex, the elements are retrieved through an iterator and printed to the file. For TreeIndex, the elements are sorted as they are added and so, an iterator is created for the entrySet() of the TreeMap. The Map is then iterated through and the file is saved after all key-value pairs have been printed to it. For the HashIndex, a Set of the key is created using keySet() and sorted into an ArrayList. An iterator is then created over this ArrayList and the key-value pairs are printed to the file.

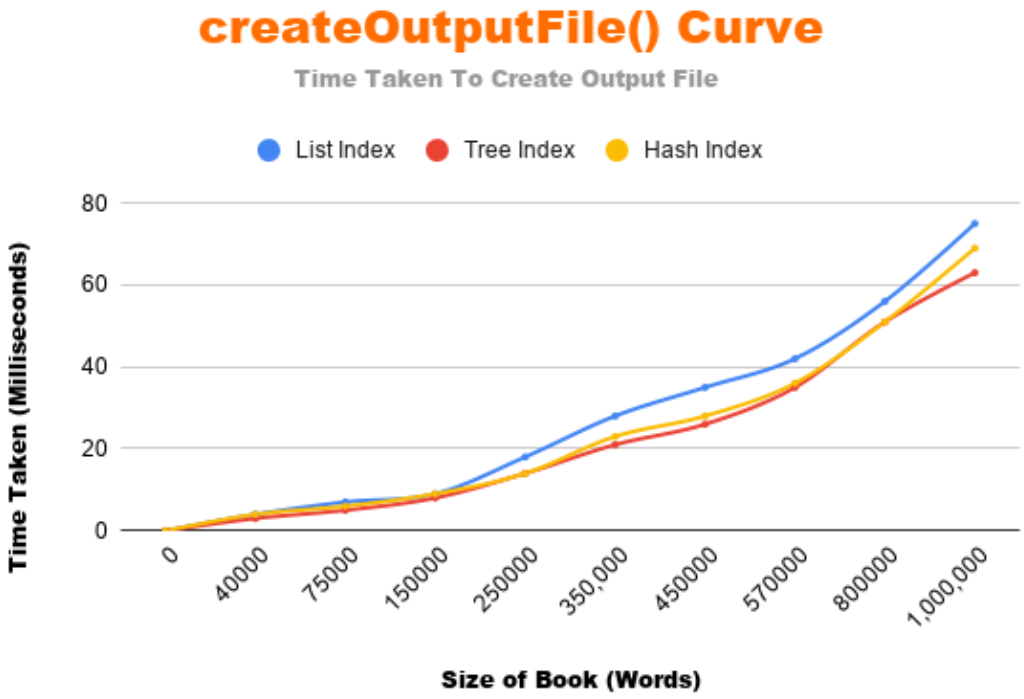

Figure 5: Time Taken For Creating The Output File

From the data above, we can see that while the TreeIndex and HashIndex essentially take the same time to create the output file for any book size, the ListIndex takes a bit longer. While it may seem like there is a bit of a gap in the time the ListIndex takes versus the other indexes, it should be noted that this difference is a few milliseconds, no matter what the size of the book is. This is further highlighted in the fact that the difference between the time taken for the ListIndex versus the other indexes is consistent for every book

size. This is indicator that the performance is stable and the methods are working just like they're supposed to. The consistency in performance is a good sign as it shows the program performs at the same rate no matter how large the input data is.

The next and final experiment involves the entire index-building process: reading the input files and then creating the output files.

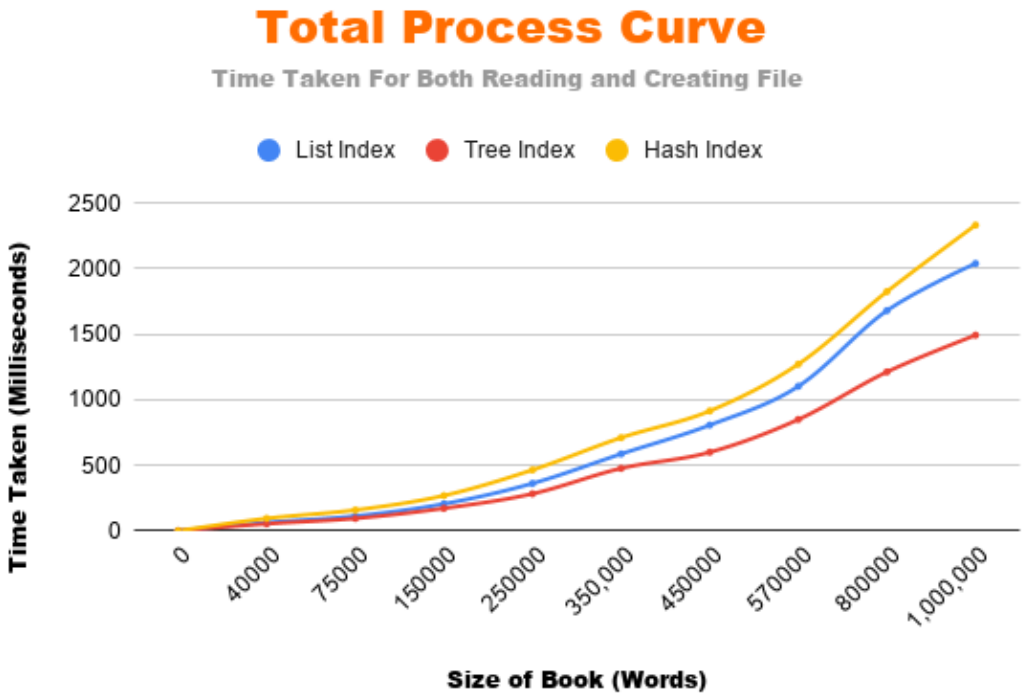

Figure 6: Time Taken For Both Reading and Creating The File

Some separate tests were conducted to find out if the time taken for the total process could be calculated by adding the time taken for reading the file and creating the output file. However, this was not the case. It was observed that the time taken for the whole process was greater than the sum of its parts. This could be down to system load and performance slightly slowing down as many experiments were being run one after the one.

The graph for the entire process shows that TreeIndex takes the least amount of time overall, whereas HashIndex takes the most time overall. This follows logically from the two previous graphs. TreeIndex took the least amount of time for both individual processes (was tied with ListIndex once and was tied with HashIndex once) and it makes sense for it the take the least amount of time overall for the two processes combined. While the HashIndex took the same time as the TreeIndex when it came to creating the output file, it took significantly longer for reading the file than the other two indexes. That is why, in the overall rankings, the HashIndex takes the most time. In the same graph, the ListIndex is sandwiched between the HashIndex and TreeIndex because in one earlier experiment, it takes the least amount of time alongside TreeIndex, whereas for the other it takes more than the other two indexes. However, unlike the HashIndex, the difference between the time the ListIndex took as compared to the other two in that experiment isn't very significant, which keeps it right under the HashIndex curve.

It can be seen that the HashIndex and TreeIndex perform the most consistently despite being opposites in performance. ListIndex seems to be the only one with minor inconsistencies in the time taken. And alongside the slight inconsistency, ListIndex seems to be closer to HashIndex (the most time taken) rather than the TreeIndex (the least time taken).

This reinforces the idea that the TreeMap/TreeIndex is the best performer in this project and should be the preferred data structure for building an index for any book size. Another observation that goes in TreeIndex's favor is the fact that as the size of the book increases, the difference between the TreeIndex curve and the other two also increases. This suggests the performance of the TreeIndex is the most stable no matter what the size of the input file being tested is.

6 Unit Testing

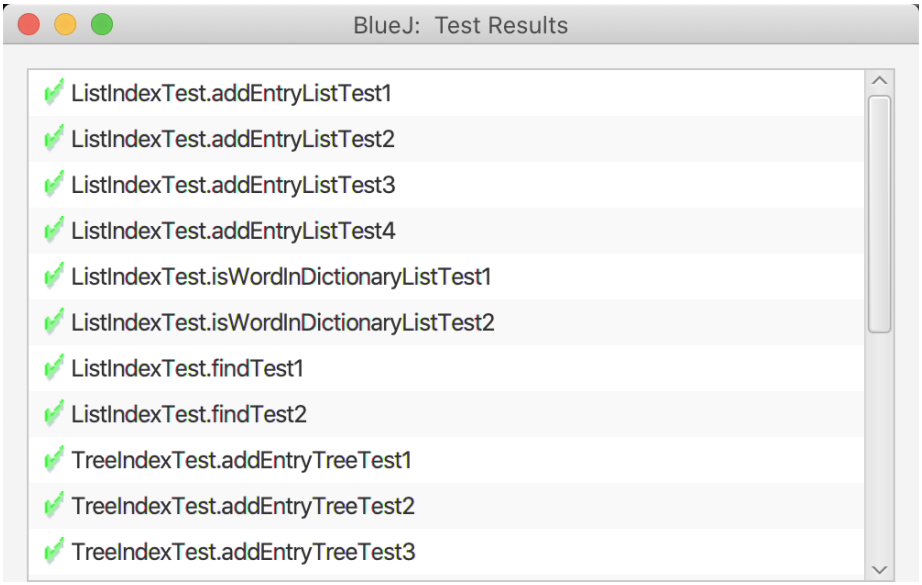

Figure 7: Output from the first half of the Unit Tests

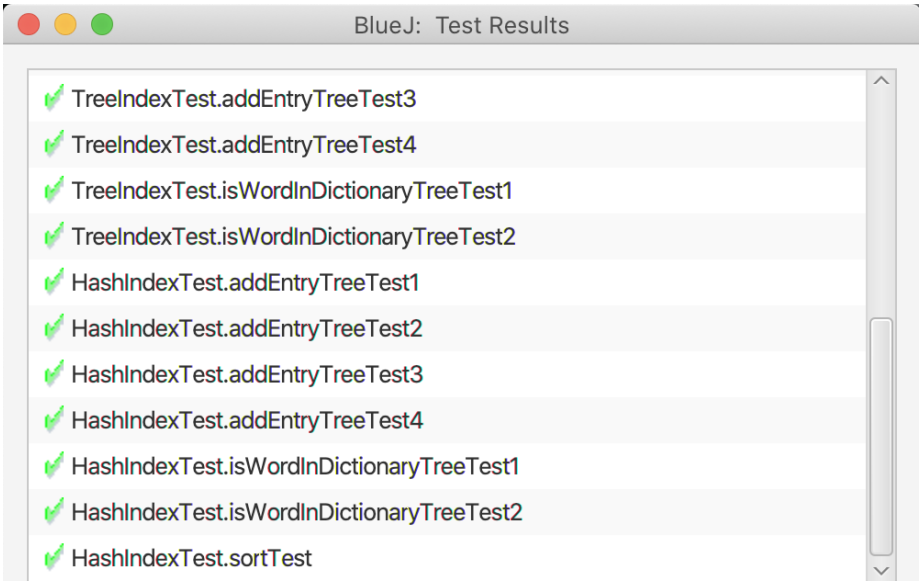

Figure 8: Output from the second half of the Unit Tests

7 Troubleshooting

Initially, the search method find() had been implemented without Binary Search. This resulted in abnormally large outputs for the time taken to read the file in ListIndex. This was because the find() method was using linear search which isn't efficient at all when it comes to dealing with such a high volume of data. Once this had been changed to Binary Search, the time for reading the file in ListIndex had been significantly reduced and was now comparable to the time taken by TreeIndex and HashIndex.

Another issue faced was deciding how to iterate through each data structure as there were many ways to do so. After conducting many tests on each data structure - such as sorting the ArrayList in reverse order and returning the last element or not using an iterator on it, not using an iterator on the TreeMap or accessing elements from the keySet(), and whether or not an iterator should be used on the HashMap - the most efficient techniques were implemented. Therefore, it was decided that the elements would be added to the ArrayList in a sorted manner and an iterator would be used, an entry set iterator would be used on the TreeMap, and the HashMap would be sorted using its key set and an iterator would be used on it for the most efficient performance possible.

The comparisons in these variations can be seen in the graphs that follow:

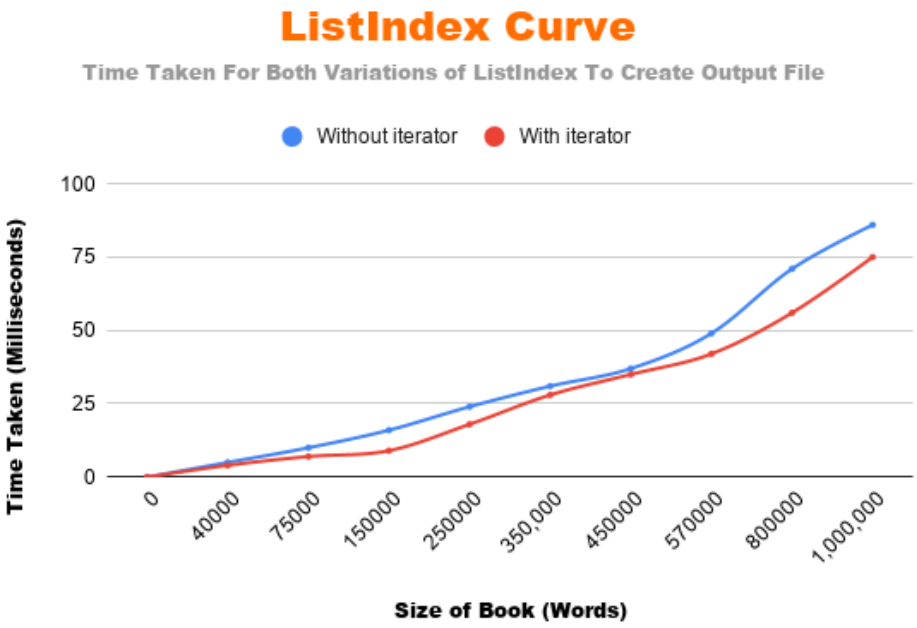

Figure 9: Output from the first half of the Unit Tests

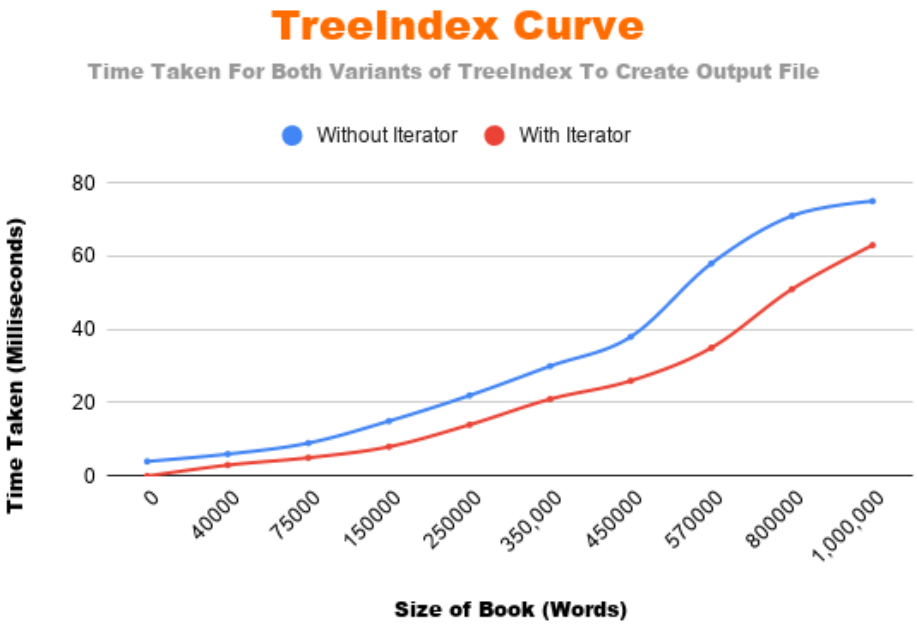

Figure 10: Output from the first half of the Unit Tests

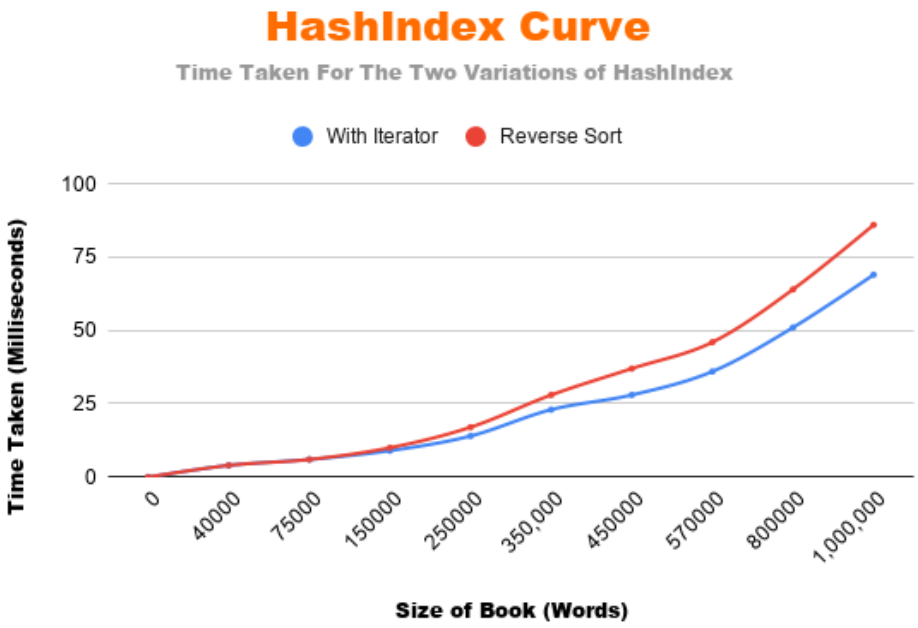

Figure 11: Output from the first half of the Unit Tests

8 Conclusion

Book indexes are a much needed feature in today's world and a very relevant practical application. Due to the fact that books contain many hundreds of thousands of words, there arises the need for finding the fastest way to go through the book and build an accurate index. And that was exactly the purpose behind this project. Conducting the experiments above and analyzing the data allowed us to reach conclusions about each different data structure.

By putting the ListIndex, TreeIndex, and HashIndex up against each other, it was observed that TreeIndex (and the TreeMap) consistently performed in a stable manner and at a steady rate. Even though it was expected that the HashIndex would be the fastest, it can be said that the whole process of sorting it and then having to access elements from an ArrayList used up a bit

more time. However, even before that, it was mentioned that a HashMap took significantly more memory than a TreeMap. Therefore, it could be reasonable to say that this extra memory usage over many, many tests slowed down the program to an extent and caused it to become slower than the TreeIndex.

Whatever the case may be, overall, the TreeIndex was the best performer and the most efficient as it took the least amount of time to create the entire index. Hence, in a comparison between an ArrayList, a TreeMap, and a HashMap where the output needs to be in some order (such as book indexes), it seems as if a TreeMap is the right way to go.

9 References

Baeldung. "Time Complexity of Java Collections." Baeldung, 19 July 2019, www.baeldung.com/java-collections-complexity.

Baeldung. "TreeMap vs HashMap." Baeldung, 9 Apr. 2020, www.baeldung.com/java-treemap-vs-hashmap.

Hart, Michael. "Popular Books." Project Gutenberg, www.gutenberg.org/catalog/

Oracle. "HashMap API." Java Platform SE 8 API Documentation, 11 Mar. 2020, docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

Oracle. "TreeMap API." Java Platform SE 8 API Documentation, 11 Mar. 2020, docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html.

Oracle. "Set API." Java Platform SE 11 API Documentation, 7 Sept. 2018, docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html.

Wizard, Info Tech. "Java Collections Performance." Information Technology Gems, 6 Nov. 2011, infotechgems.blogspot.com/2011/11/java-collections-performance-time.html.

Xia, Ge. "Lecture Notes 11 - Sets and Maps." Moodle, 24 Apr. 2020, moodle.lafayette.edu/pluginfile.php/571532/mod$_r$*esource*/*content*/1/*setsmaps.pdf*.