Ali Sultan Sikandar
5/10/2020
Project 3 Report

Introduction

The project basically investigates how different data containers like **ArrayList, HashMap, and TreeMap** work. This is done by reading from **text format** books and creating an **Index** (containing all the words followed by the list of line numbers they appear in) of these books. We measure the time it takes to create the index and the time it takes to write it to the output file for each of these data structures through analysis of multiple runtime experiments.

Initial hypothesis suggests that in terms of creating the **index, HashMap** should be most efficient because its **add and contains** method is of constant time complexity (big-O(1)). This should be followed by either **TreeMap** or **ArrayList,** because both of them take time complexity of big-O(lgn), where n is the number of items in the data structure, for the addition method. Note that while by default, an **ArrayList** add method has time complexity of big-O(1)**,** in our case we are conducting **binarysearch** before every addition, so time complexity subsequently degrades to big-O(lgn) for **add.**

On the other hand, hypotheses suggest that writing the index to the output file should represent a different story. **TreeMap and ArrayList** should prove to be the fastest because one just needs to read from the index and write it to the file without worrying about the order. **HashIndex,** however, should be the slowest because its entries must first be sorted before being printed out to the output file.

Approach

To maintain an object oriented design for the project, we use an **Interface** called **Index** which represents the generic **Index** class. Three concrete classes, **ListIndex, HashIndex and TreeIndex** implement the **Index** interface.

ListIndex, to represent every entry in order to create an Index entry, uses a subclass called **Entry** which holds the **String** word and **TreeSet** of its corresponding line numbers in the book. Therefore**, ListIndex** holds the ArrayList of these **Entries**. We also use another ArrayList containing just the words, since this helps determine the index position for addition of a new entry more conveniently.

TreeIndex and HashIndex, are similar classes, and use **TreeMap** and **HashMap** respectively. These data structures are convenient because we can use the **key** in Map to represent the **String word** and **TreeSet<Integer>** to represent the list of line numbers. In both of these classes, we also use an **ArrayList** to store all the words that are added into the map in the end. This helps the **printing** the index to the output file conveniently, and is also essential for sorting the **entries** in case of the **HashIndex.**

Finally, the **IndexTester** class is used to carry out the functionality and runtime experiments for the three Indexes. This class is built **BufferReader** to read from the **text file** which is our book to be analysed. Afterwards, inbuilt **PrintWriter** is used to write to the output text file called **myout.txt** the book index which has been created.

Methods

All the three Index classes use **four common methods : searchDict, addIndex, SizeSort** and **toString.**

The method **searchDict** is identical in all the three Index classes. This method is basically used to search the dictionary (provided as **English.txt**) for a word using the inbuilt **Collections.binarysearch**. **True or False** are returned upon the success or failure of the word in question.

The method **addIndex** only varies in the **ListIndex.** The method takes in two parameters: the word to be added and its line number in the book. In **ListIndex**, first we search for the word using **Collections.binarysearch** in the **ArrayList** of already added words that it maintains. If the word already exists, then we just add the **line number to its TreeSet** in the **Entry** class. **TreeSet** is convenient because it disallows the **duplicates** and keeps the set **ordered.** On the other hand, if the word is not found, then we create a new object **Entry** and add it to our ArrayList of Entries, using the modified **int** returned by the **BinarySearch.** This ensures that the ArrayList is always ordered after new entries have been added to their position and the subsequent entries shifted accordingly. In **HashIndex** and **TreeIndex,** things are way simpler. We just employ the **contains** method of the respective **Map** to check if the **key (Word we want to add)** exists or not. If the word exists, we just add to the **TreeSet** of line numbers associated with it as its **values.** If the word is not found, we add it to the **Map** using the **put** method.

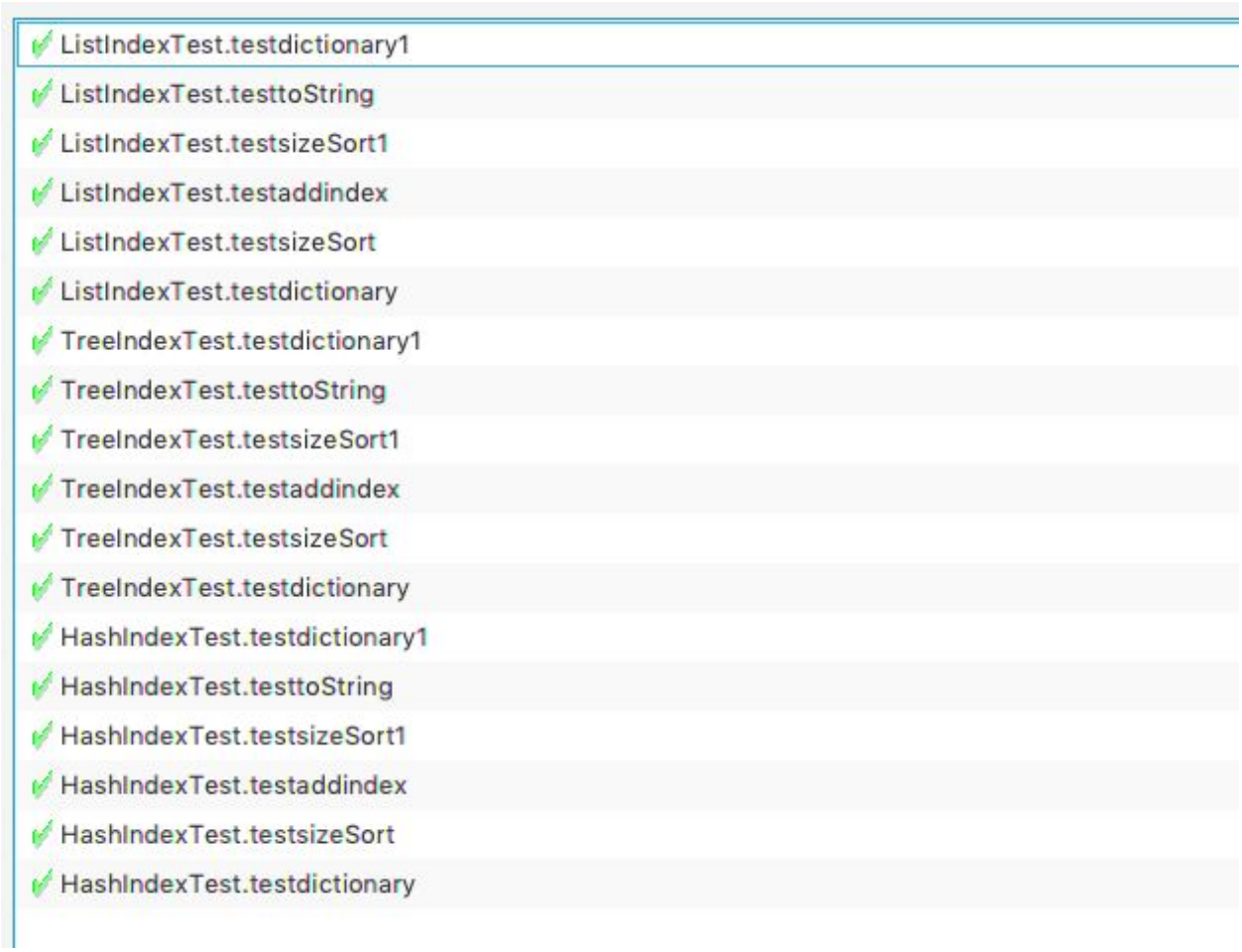
The **SizeSort** method returns the size of the Book Index i.e the number of total elements in the book Index. For the **HashIndex,** however, the method also sorts the

index alphabetically by calling the **Collections.sort** method on the **array list** of words derived from the **HashMap**. ListIndex and TreeIndex do not require this step.

The **toString** method just returns a string at **nth position** of the respective book index. The word and the **TreeSet** of lines associated with that word are returned as a **string.**

Lastly the main method in the **IndexTester** class does the job of reading from a book and using the specific index specified to build the index. The book is read line by line, and every line is converted into a single **String word**. This word is first searched in the dictionary provided in the package named **English.txt** using **BinarySearch.** If found successfully, it is added into the index with the line number which is tracked by an **int variable linecount.** Time in milliseconds, **using the inbuilt System.currentTimeMillis** , of creating the Index is recorded. Afterwards, the process of writing to index to the output file begins. This is done by just running a **for-loop** and writing the index entries into the output file using the **println** method of the **PrintWriter.** Time for this in milliseconds is also recorded. The entire main method is run five times, and the two times we are concerned with are **averaged out** and printed on the console for analysis. This ensures that the values of time are reliable as there can be some difference in them everytime.

All the methods are unit tested and their results are as follows:



Data & Analysis

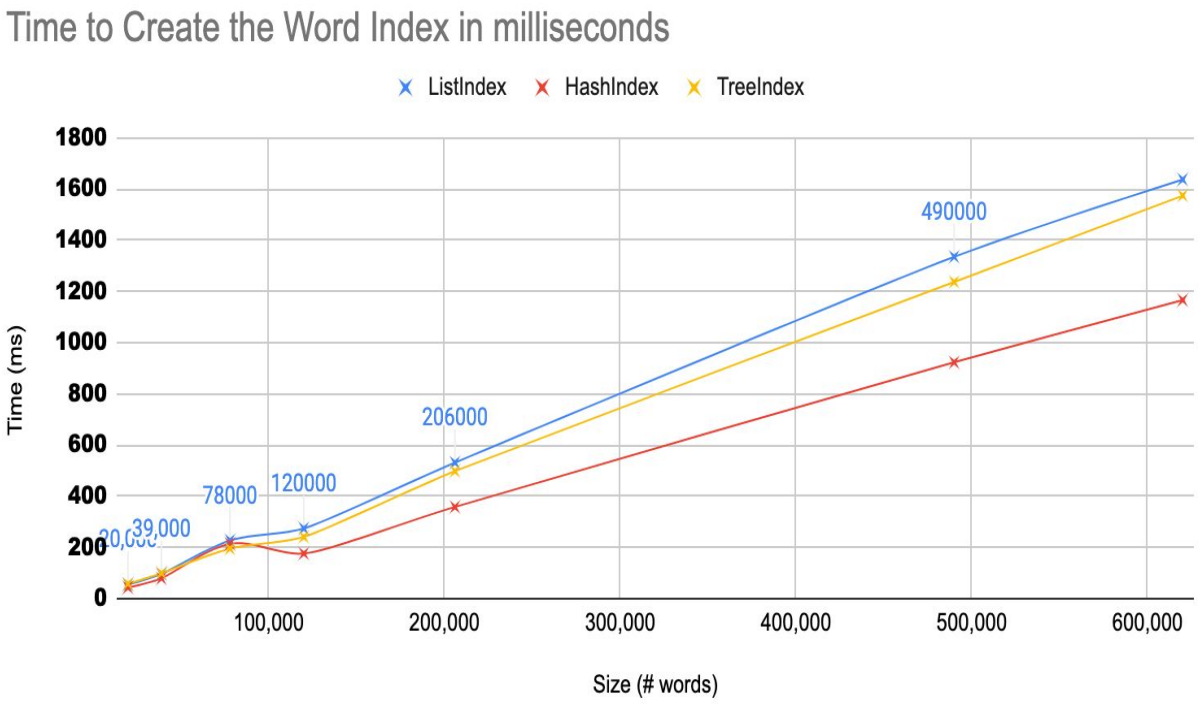


Fig 1. Time to create the word Index vs The book size

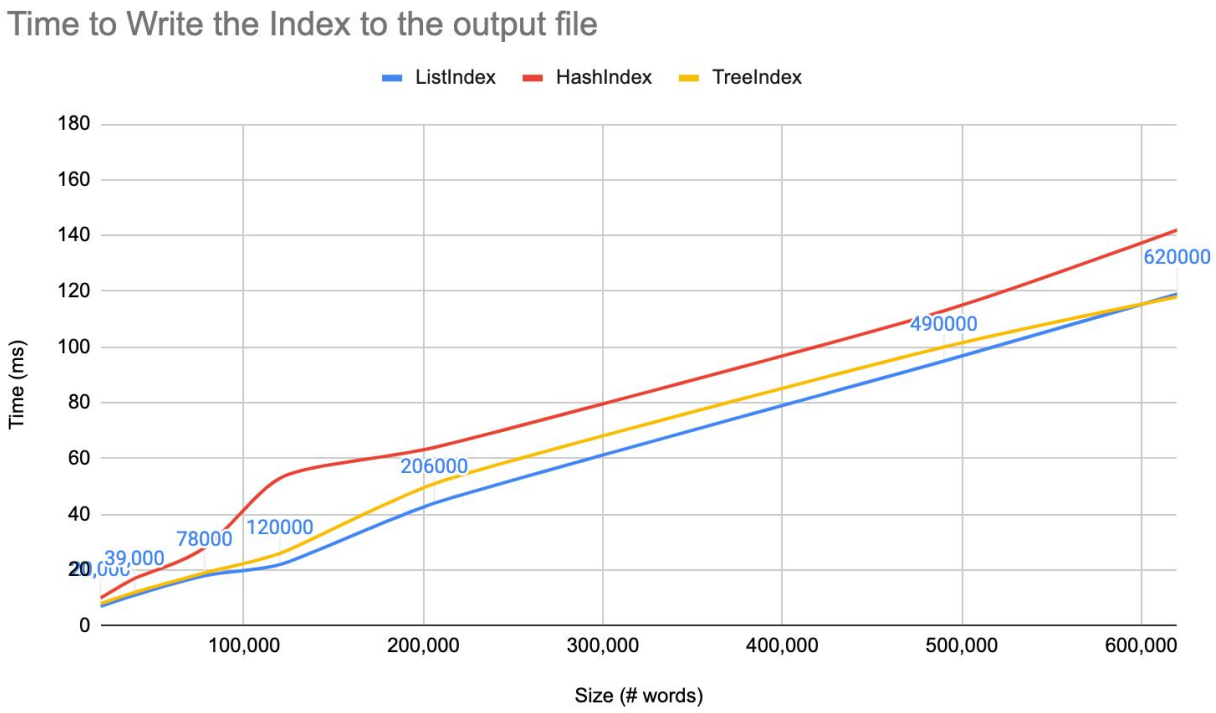


Fig 2. Time to output the word Index vs The book size

For both of the data graphs above, every data point represents the size of book and time it takes in milliseconds to create the specific index. We use books of approximate word count of **20k, 39k, 78k, 120k, 206k, 490k and 620k** for our analysis.

Conclusion

All the hypotheses prove to be right once we run the experiments in the **Controller** class.

In terms of creating the Index**, HashIndex** proves to be the most efficient. It performs almost two times better than the **TreeIndex and List Index** which seem to take more or less the same amount of time in creating the index. However, because hashing does not consider the order while adding, writing to the output file using **HashIndex** takes slightly more time than its competitors because before writing it has to sort the entire index.

To create the **ListIndex**, the time taken is the most amongst all the three options as hypothesized. While it is slightly taking more time than **TreeIndex**, it is twice as

inefficient as **HashIndex.** On the other hand, outputting the **Index using ListIndex** proves to be the fastest of all the options.

TreeIndex, proves to be the most efficient option overall. Its time performance sits between **HashIndex and ListIndex** in both creating the Index and outputting it. Hence we conclude that for overall performance, where both creating the **BookIndex** and outputting it is required, one must use **TreeMap** data structure since it is balanced, from books ranging in size from small to medium to large.

References

- System. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/7/docs/api/java/lang/System.html

- Weiss, M. A. (2010). *Data structures & problem solving using Java*. Boston, MA: Pearson.

- Ge Xia. (N.d.) *Lecture Note 10.* Moodle. Retrieved March 4, from https://moodle.lafayette.edu/pluginfile.php/571532/mod_resource/content/1/setsmaps.pdf

- ArrayList. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

- HashMap<K,V>. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

- TreeMap<K,V>. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

- TreeSet<E>. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

- BufferReader. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

- PrintWriter. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html

- Collections. (2018, October 6). Retrieved May 10, 2020, from https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html