Claire Liu, Renee Soika, and Yuqi Wang

December 1, 2020

CS 150-01

# Project Report: Book Index

## 1. Introduction

The purpose of this project is to implement a program that creates a book index of any book given in txt format in three ways—using a sorted list, a tree map, and a hash map. The book index lists all the English words that appear in the book and the line numbers where each word appears (Xia, "Project III").

A few assumptions were made. First, the cases of the words were ignored. This could change the meaning of certain words. For example, "Frank" is a name, while "frank" means honest, but these were considered to be the same word. In addition, only words that contained letters from A-Z were considered. Words with accented letters and numbers were not included. Furthermore, in our theoretical analysis, we assumed that the majority of the words in the file would be English words. We also assume the number of words in a line number are roughly the same for all input files. Also, it is a safe assumption that the number of lines in a file is directly proportional to the size of the file in KB.

## Theoretical Analysis

For all three implementations, words were first checked in an array of words from the English dictionary through binary search. A binary search has a theoretical time complexity of `O(lg n)` because an array of length `n` is divided in half `k` times until the length of the array becomes 1. This is mathematically expressed as $n/2^k = 1$. After some rearrangement, we find that the `k = lg(n)`. So, `O(lg n)` where n is the size of the array being searched should be added to the runtime of an insertion whenever a binary search is performed.

In addition, a TreeSet is used in all three implementations. Under the hood, a TreeSet is a red-black tree where the elements are ordered using their natural ordering or as specified by an override of the Comparable class. The TreeSet has guaranteed `O(lg n)` performance for the add method and the contains method where `n` is the number of nodes in the TreeSet because of the definition of a red-black tree. First, at least half the nodes in a red black tree are black. So, the black height is greater than or equal to height divided by 2. Also, a red-black tree balances itself, the best case of a binary tree, so no path from any node x to a NULL is more than twice as long as any other path from X to any other NULL. Every path from a node to any NULL contains the same number of black nodes by definition. Lastly, since $bh(x) >= h/2$, then

$$n >= 2^{bh(x)} - 1$$

$n + 1 >= 2^{h/2}$ *substituting in bh(x) for h/2.*

$lg(n + 1) >= h/2$ *simplify*

$$2lg(n + 1) >= h$$

So, the height of a red black tree with n nodes is $h <= 2\ lg(n + 1)$. An asymptotic analysis of the above yields $O(lg\ n)$ performance for adding to a TreeSet where $n$ is the number of nodes in the TreeSet (Xia 2020).

For the rest of our analysis, $n$ refers to the number of lines in an input file. We assume that there is an approximately constant number of words per line.

It is important to note that $O(n)$ can represent the number of lines in the input file, the number of words and the number of line numbers associated with each unique word (i.e. the number of times a word is repeated in the file). This because there is a relatively constant number of words per line in the input file, so the number of words in the file could be expressed as `O(n * (# of words per line))`, but the constant coefficient simplifies to $O(n)$ in asymptotic analysis.

The number of unique words can also be expressed as $O(n)$ because a proportion of each line is made up of newly-encountered words, and a proportion of each line is made up of words that have already been encountered (repeat words). So, the number of unique words in a file can be expressed as `O(n * (# of unique words per line) / (# of words per line))` which equals $O(n)$ via asymptotic analysis. The complement of that is `O(n * (# of repeat words per line) / (#  of words per line))`.

The number of line numbers for each word is also $O(n)$ because there will likely be more occurrences of a word as the total number of lines increases. However, many words will only occur a handful of times. Therefore, the coefficient of the $O(n)$ is quite small, so the $O(n)$ number of line numbers for a word behaves more like $O(1)$ in general. The number of line numbers would be closer to O(n) than O(1) is for an extremely common word. In a file with a size approaching infinity, each word would have many line numbers corresponding to it because the number of words in the English dictionary is finite, though the number of line numbers for each word would still be a relatively small fraction of the total number of words. **Therefore, in this theoretical analysis, we will treat the average number of line numbers for one word as O(n) but mention that it behaves more like O(1) when used in a file of a practical size**.

**Sorted List Implementation**

For the sorted list, an ArrayList<Entry> was created where an Entry object contained a String for the word and a TreeSet<Integer> for the set of line numbers (Oracle "ArrayList"; Oracle "TreeSet"). For each of the O(n) words in the file, the English dictionary array is first binary searched, which takes O(lg (size of dictionary array)). Since the array must remain sorted, the program performs a binary search on the ArrayList of Entries, which takes O(lg n) each time. If the word is new (Case A), then it is added to the ArrayList of Entries, which takes approximately O(n) each time. Then an associated TreeSet is created, which takes O(1) each time. So, in total for O(n) new words it would take O(n) * (O lg (size of dictionary array)) + O (lg n) + O (n) + O(1)) = D + O(n lg n) + O($n^2$) + O(n) = O($n^2$). However, if the word is already in the index (Case B), then it is added to the associated TreeSet, which takes (O lg n) time. So, in total for the O(n) words that are already in the index, it would take O(n)*(O lg (size of dictionary array)) + O (lg n) + O (lg n) = O(n lg (size of dictionary array) + O(n lg n) + O(n lg n) = O(n lg n). Since a proportion of the words are new words to the index and the complementary proportion is the number of new words, then the runtime for adding words from the file is a combination of the two cases, which takes somewhere between O($n^2$) and O(n lg n) time.

After all the words are inserted, the ArrayList has to be written to an external file. Converting one Entry to a String requires O(n) time when there are O(n) line numbers for each unique word. Since the elements are already in sorted order, the ArrayList can be iterated over directly, which takes O(n) time. So, it takes O($n^2$) to iterate over O(n) words and iterate over

`O(n)` line numbers for each. It is important to note that the `O(n)` number of line numbers for each word will actually behave more like `O(n)` when used in a file of a practical size because the `O(n)` number of line numbers is small compared to the `O(n)` number of words in the file. So writing to a file takes $O(n^2)$ theoretically, but will perform more like `O(n)`.

The whole process for creating the book index (inserting words and writing to file) takes between $O(n^2) + O(n^2)$ and $O(n^2) + O(n \lg n)$ time. Technically, an asymptotic analysis would yield $O(n^2)$ in total, but inserting words takes considerably more time than writing to a file, so the runtime for inserting words to a file will be more influential in the total runtime.

The advantages for using a sorted list over other implementations are that the sorted list maintains alphabetical order, it is easy to write to a file, and it is a simple implementation that does not require as much overhead space as other implementations because it is a simple ArrayList. The disadvantage is that it has a $O(n^2)$ and `O(n lg n)` insertion runtime so it may be slower than other implementations.

**Tree Map Implementation**

The tree map implementation is a TreeMap<String, TreeSet<Integer>>, which is essentially a red-black tree of red-black trees (Oracle "TreeMap"). For each word in the file, first the dictionary array is binary searched which takes `O(lg (size of dict array)),` then the TreeMap is searched for the word, which takes `O(lg n)` time where `n` is the `O(n)` number of words already in the TreeMap. If the word is new to the index (Case a), then it takes `O(lg n)` time to put a new node into the TreeMap and `O(1)` time to create an associated TreeSet. So, the total insertion process for `O(n)` new words to the index is `O(n)* (O(lg (size of dict array)) + O(lg n) + O(lg n) + O(1) = O(n lg (size of dict array) + O(n lg n) + O(n lg n) + O(n) = O(n lg n).` If the word is already in the TreeMap (Case B), then only the line number is added to the associated TreeSet, which would take `O(lg n)` time where `n` is the `O(n)` number of line numbers of a certain word. So, the total insertion process for `O(n)` words that are already in the index is `O(n)* (O(lg (size of dict array)) + O(lg n) + O(lg n) = O(n lg (size of dict` `array) + O(n lg n) + O(n lg n) = O(n lg n)`. Theoretically, the runtime for insertions to Tree Map would take between `O(n lg n)` and `O(n lg n)` which is just `O(n lg n).`

Next, the TreeMap needs to be written to an external file. Since the red-black trees under the hood of TreeMap are binary search trees by definition, an inorder traversal of the TreeMap and each corresponding inner TreeSet maintains the sorted order of the data. Iterating over each of the `O(n)` unique words takes `O(n)` time and iterating over each corresponding TreeSet of `O(n)` line numbers takes an additional `O(n)` time for each O(n) word, so the total runtime for printing to a file is $O(n^2)$. It is important to note that the `O(n)` number of line numbers for each word will actually behave more like `O(n)` when used in a file of a practical size because the `O(n)` number of line numbers is small compared to the `O(n)` number of words in the file. So writing to a file takes $O(n^2)$ theoretically, but may perform more like `O(n).`

The whole process for creating the book index (inserting words and writing to file) takes $O(n^2)$ `+ O(n lg n)` time. Technically, an asymptotic analysis would yield $O(n^2)$ in total, but inserting words takes considerably more time than writing to a file, so the runtime for inserting words to a file will be more influential in the total runtime.

A major advantage of tree map approach is that an inorder iteration of a TreeMap will preserve the ordering of the key making it easier to write to a file, also it is faster than the Sorted List implementation because it has `O(n lg n)` runtime for insertion. A disadvantage is that the elements being inserted into the TreeMap must be comparable. Even if it does not make sense for it to be comparable, a compareTo method must be imposed on it. However, for the purposes of the book index, this is not a disadvantage. Also, the space requirements for the TreeMap are slightly more than the space requirements for a sorted list because each node in the TreeMap must also store the right and left children nodes and the color of the node.

**Hash Map Implementation**

The hash map implementation of the book index is a HashMap<String, TreeSet<Integer>> (Oracle "HashMap"). The Java implementation of a HashMap has a default load factor of 0.75, which means that every space in the HashMap has 0.75 elements, on average. This is a good trade off between time and space costs because the load factor of 0.75 minimizes collisions, improving the performance of the HashMap so that it is `O(1)` runtime for the contains and put methods, on average. When the load factor gets too large, the runtime approaches `O(n)` for the contains and put methods without rehashing because the HashMap has to search through the list of collisions. However, when the load factor is too large, the Java implementation rehashes all the objects in the HashMap, which takes `O(n)` time, but keeps the runtime to `O(1)` for future additions. Java's HashMap starts with a capacity of 16 and doubles when the load factor exceeds 0.75 (Oracle "HashMap"). Doubling the size of the HashMap each time means that the rehashing process will occur O(lg n) times. In the context of creating a book index, it takes `O(1)` time to search the index for a word's associated TreeSet. If the word collides with many other words, it takes `O(n)` time to search all the collisions to find the correct TreeSet. Next, if the word is new to the index, then it takes constant time to create the set and insert a single element. Otherwise, the line number is added to the associated TreeSet, which takes `O(lg n)` time. The complete insertion process for one word takes the most time when there are many collisions, `O(n) + O(n) + O(lg n) = O(n)`. The average case, when there are few collisions, is `O(1) + O(1) + O(lg n) = O(lg n)`.

The sorted order of the elements inserted into the HashMap is not maintained because the indices are decided by the hashing process. So, when the contents of the HashMap are being written to the file, the elements must first be sorted via Arrays.sort() which uses merge sort for objects and has a `O(n lg n)` runtime (Oracle "Arrays"). In addition, creating the array copy takes `O(n)` time, but creating the List copy via Arrays.asList takes `O(1)` time because a List contains an array (Oracle "List"). The retrieval of `O(n)` line numbers for `O(n)` words takes $O(n^2)$ time. Overall, this means the retrieval of all words and line numbers in the index in order will take $O(n^2)$ time. It is important to note that the O(n) number of line numbers for each word will actually behave more like O(n) when used in a file of a practical size because the O(n) number of line numbers is small compared to the O(n) number of words in the file. So writing to a file takes $O(n^2)$ theoretically, but may perform more like O(n).

The advantages for the HashMap is that the insertion of new words is fast, having `O(lg n)` performance. However, completing the book index by writing to the file takes more time than the other implementations because of the additional `O(n lg n)` time required to sort, even if all implementations take `O(n²)` time overall. Furthermore, hashing and rehashing is expensive because each string must be iterated by each character to calculate the hash code. HashMaps tend to take up more memory space than other implementations when the load factor is low, because even though it performs efficiently, there are many empty buckets in the HashTable. MergeSort used to sort the contents of the HashMap via Arrays.sort also requires `O(n)` additional space to sort.

| **Symbol** | **Definition** | **Notes** |
|---|---|---|
| `D` | time to binary search the words in the English dictionary for every word in the file | `O(words in file * lg (size of dictionary)) = O(n * lg (size of dictionary))`<br><br>Same for all three implementations. |
| `T` | time to insert the line numbers into a TreeSet for every new word in the file | `O(unique words * lg (average line numbers)) = O(n lg n)`<br><br>`O(n lg n)` is according to asymptotic analysis, but it will behave more like `O(n)` when used in a file of a practical size because many words occur infrequently.<br><br>Same for all three implementations. |
| `S` | time to make a new TreeSet for every word in the file that is already in the index | `O((unique words) * 1) = O(n)`<br><br>Same for all three implementations. |

*Figure 1: Definitions and justifications for common situations in our theoretical analysis.*

Figure 1 contains symbols and the analyses of different variables and processes that are the same for all three implementations: binary searching the dictionary, inserting line numbers to a TreeSet and creating a new TreeSet.

| Runtime case | Sorted List Index (ListIndex) | Tree Map Index (TreeMapIndex) | Hash Map Index (HashMapIndex) |
|---|---|---|---|
| Case A<br><br>Need to insert `O(n)` new words into index | **`D + O(n lg n)`** (search ArrayList) + **`O(n²)`** (add elements to middle of ArrayList) + **`S`** | **`D + O(n lg n)`** (contains) + **`O(n lg n)`** (put) + **`S`** | **`D + O(n)`** (hashing)**+ `S`**<br><br>Rehash<br>**`D + O(n²)`** (rehashing) **`+ S`** |
| Case B<br><br>Need to insert `O(n)` words already in the index | **`D + O(n lg n)`** (search ArrayList) + **`T`** | **`D + O(n lg n)`** (contains) + **`T`** | **`D + O(n)`** (hashing) + **`T`** |
| Write to file | **`O(n²)`** *(will act more like `O(n)`)* | **`O(n²)`** *(will act more like `O(n)`)* | **`O(n lg n) + O(n²)`** *(will act more like `O(n)`)* |
| Combined = a(Case A) + b(Case B) + Write to file<br><br>where:<br><br>a = proportion of insertions that introduce a new word<br><br>b = proportion of insertions that do not introduce a new word<br><br>a + b = 1.0,<br>assuming b > a | **`D + O(n lg n) + a * O(n²) + b * T + O(n²)`** | **`D + O(n lg n) + a * O(n lg n) + a * S + b * T + O(n²)`** | **`D + O(n) + a * S + b * T + O(n lg n) + O(n²)`**<br><br>When rehashing is needed, it takes:<br>**`D + O(n²) + a * S + b * T + O(n lg n) + O(n²)`** |

*Figure 2: A summary of the theoretical analysis of the three implementations for a book index. The leading terms (what an asymptotic analysis would yield) are highlighted in yellow.*

**Hypotheses**

For most books, the proportions of insertions that do not introduce a new word to the index (Case B) will greatly exceed the proportion of insertions that introduce a new word (Case A). So, the terms multiplied by b will be more influential in the total runtime than the terms multiplied by a. Therefore, for the insertion algorithm, we predict that the Hash Map data structure will be the most efficient for creating a book index because it has `b*O(n) + a *O(n)` (or sometimes `a*O(n`$^2$`)` for rehashing) runtime. Tree Map will be the second fastest because it has `b*O(n lg n) + a*O(n lg n)` runtime, which is worse than Hash Map's performance. We hypothesize that the Sorted List data structure will have the worst performance because it has `b*O(n lg n) + a* O(n`$^2$`)` runtime, which is worse than Hash Map and Tree Map's performance.

However, for writing to the file, we predict that the Sorted List will have the fastest performance because it takes `O(n`$^2$`)` which will act more like `O(n)`. In second place will be the Tree Map which also has `O(n`$^2$`)` performance which will also act more like `O(n)`, but might be slightly worse than Sorted List because it must perform an inorder traversal of a binary search tree, which is more complex than a simple traversal of an ArrayList. Writing a Hash Map to the a file will be the worst performance because it also has a `O(n`$^2$`)` performance which will act like `O(n)` but it is trailed by a `O(n lg n)` which will increase the runtime.

Combining the insertion algorithm and writing to the file, we predict that in general, HashMap may be the best implementation for a book index because it is the fastest for the insertion process. It is the slowest for the write to file process, but writing to a file is relatively faster than the insertion process, so the runtime of the insertion process should be given precedence. We hypothesize that TreeMap will be the second fastest. Sorted List will be the slowest.

However, different implementations may be the best under different scenarios. For short books, it may be best to use the ListIndex because when the input size is small, the performances of all three implementations are relatively the same. ListIndex is a simple implementation which has the smallest space requirements out of all the three implementations. However, for medium books, either a TreeMapIndex or a HashMapIndex could work because their performances will be relatively similar. For larger books however, a HashMapIndex would almost definitely be the best option.

Another factor to consider is the ratio of unique words to repeat words in a file. The first would be Case A and the latter would be Case B. Most of the time, and especially for longer books, the number of words in total will exceed the number of unique words. So, Case B is likely more influential in the total runtime than Case A. However, in files where Case A would exceed Case B and there are many words, a TreeMapIndex would be the best option because it has a Case A runtime of `O(n lg n)` where as Sorted List has a Case A runtime of $O(n^2)$ and HashMapIndex has a `O(n)` runtime and sometimes for rehashing, a $O(n^2)$ runtime. In files where Case A would exceed Case B and there are fewer words, then a HashMapIndex might be the best because it would not have to rehash and have $O(n^2)$ runtime often.

## 2. Approach

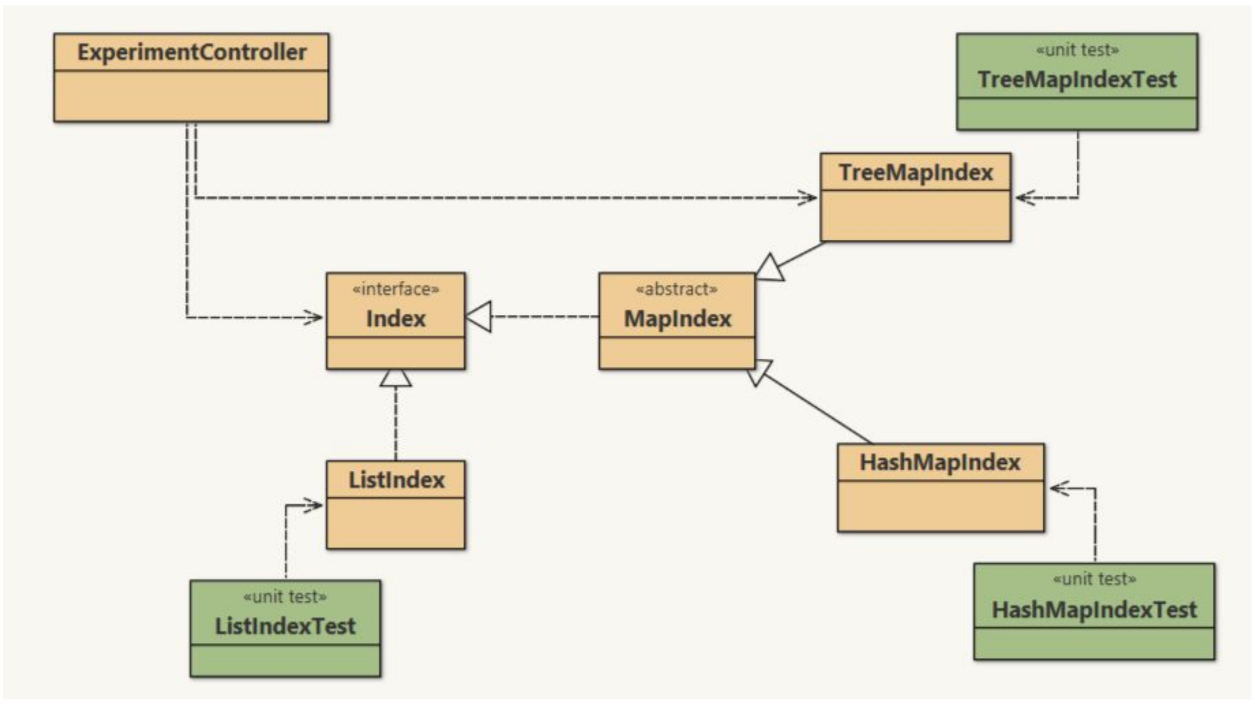

*Figure 3: The class diagram for our program.*

We started our program by creating the Index interface because all three indices should have the same external behavior; that way, we could compare the efficiency of their internal implementations and use the same methods to test every index. This interface only prescribes an insert method and extends the Iterator interface, which ensures that the entries in the index are accessible (Oracle “Iterator”).

Then we created the ListIndex class, which implements the Index interface. It accepts a String array dictionary as a parameter so that the index is independent of how the dictionary is read or generated. Since we do not know how many words we will add to the index ahead of time, the ListIndex stores an ArrayList. The ArrayList contains instances of the ListIndex's internal Entry class, which holds a word and an Integer TreeSet of its line numbers. An Entry is comparable based on the word it stores so that the index can put all the Entries in alphabetical order. ListIndex determines if a word is to be inserted by searching for the word in a dictionary of English words through a binary search. If the word is valid, a temporary Entry is created and using a binary search from the Collections class, the program finds either the position of an existing Entry for the same word or the position where the new Entry should be inserted (Oracle “Collections”). If the word already has an Entry, the line number is added to that Entry, and the temporary Entry is discarded. Otherwise, the temporary Entry is kept and inserted into the index. This allows the index to always store the Entries in alphabetical order, and the index's iterator can simply return the String form of each Entry in their current order.

Before we implemented the TreeMapIndex and HashMapIndex, we created an abstract MapIndex base class for both of them to share. Both TreeMap and HashMap implement the Map interface, so we avoided code duplication by taking advantage of their shared functionalities (Oracle “Map”). Their iterators also use the Set interface (Oracle “Set”). The abstract base class takes a Map as a parameter, which it uses to store the words and TreeSets of their line numbers. Using TreeSets like we did for ListIndex allows us to compare the map-like implementations of each index directly, instead of conflating the data with changes in both map and set implementations. Like ListIndex, the MapIndex class takes a dictionary as a parameter and binary searches the dictionary to determine if the word to insert is valid. Then it determines if the word is already present in the index. If it is present, the line number is added to the TreeSet that already exists for that word. If it is not, a new TreeSet is created for that word with the line number.

Both TreeMapIndex and HashMapIndex take in a dictionary like ListIndex and pass their respective map and the dictionary to the abstract base class. TreeMap's entries are already sorted, so the class’s iterator can just use in the order the words are already stored and write the index to a file. However, HashMap's entries are not sorted, so the words have to be sorted alphabetically before they can be written to the file.

ListIndex, TreeMapIndex, and HashMapIndex all had corresponding unit test classes as well (The J Unit Team).

ExperimentController is where the experiment is run. The user must specify which index implementation to use inside the loops the controller uses to control the current trial so that a fresh index is created for every trial to ensure accurate results. First, the dictionary is read from an external file into an array using the File and Scanner classes (Oracle "File"; Oracle "Scanner"). This occurs only once when the program is run because the dictionary never changes. Then, for five trials for each input file, an index is created and words from the input file are inserted into the index. Next, the index for each trial is written to the output file using the PrintWriter class (Oracle "PrintWriter"). When the experiment is done, the output file will only show the index of the last file that was read. The ExperimentController outputs the average runtimes for inserting to the index, the average runtimes for writing the index to a file, and the average combined (total) runtime for creating the book index.

## 3. Methods

The experiments were run through the ExperimentController's main method, the runtimes were outputted to the terminal window, and indices were outputted to txt files for the user's convenience. For one experiment, the ExperimentController must be run three times—once for each implementation. The user must change the type of index being used each time. Each implementation was tested on a different run of the program to minimize any behavior from the JVM that could interfere with the experiments.

Within each run, the insertion algorithms were timed first. Each data point was the average of 5 insertions. Then, the writing to file algorithms were timed. Each data point was again the average of 5 runs of the algorithm. The total runtime for creating an index (insertion algorithm plus writing to file) was calculated by adding the average runtimes of the  insertion algorithm and writing to files.

The parameter values for the input file sizes were chosen based on their sizes. We chose files ranging from 139 KB to 5642 KB. Because we assume that every line has approximately the same length, comparing file sizes is analogous to comparing how many lines they have. We chose a variety of files with mostly high school and above reading difficulties so that we could identify clear trends from our data; the books would not be too short and would use a typical diversity of words. When a new file is added, the name of the file must be added to the `inputFiles` array as a String and the `fileSize` must be added to the corresponding position in the `fileSize` array as an `int`.

We used the data from experiments run on one laptop, and the experiment was also run on two other laptops to ensure the general trends were consistent.

## 4. Data and Analysis

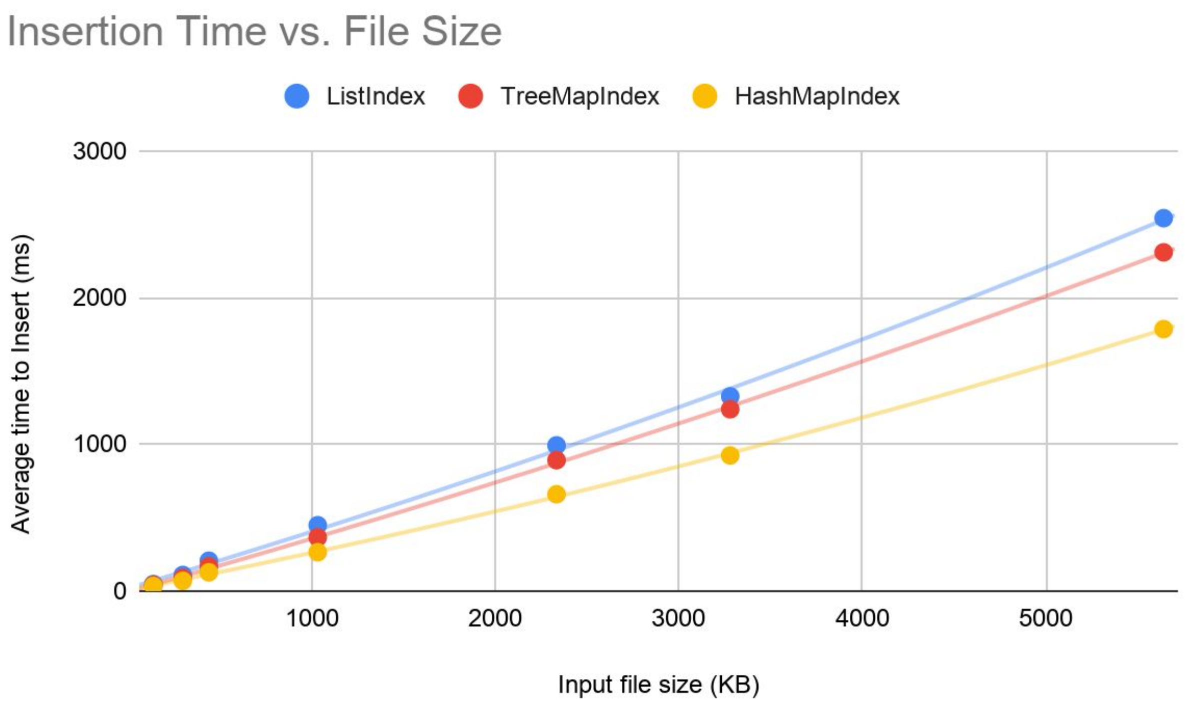

*Figure 4: The runtimes of the insertion process as a function of the input size.*

According to Figure 4, HashMapIndex is generally the fastest, followed by TreeMapIndex and lastly ListIndex. This matches our hypotheses on the performance of the three implementations. The hash map takes approximately `O(n)` time, on average, to insert `O(n)` new or existing words because rehashing occurs infrequently. However, TreeMapIndex takes guaranteed `O(n lg n)` time to insert `O(n)` new or existing words. Finally, ListIndex takes $O(n^2)$ for new words and `O(n lg n)` for existing words. ListIndex's graph appears more like `O(n lg n)` for larger input file sizes as the difference between TreeMapIndex and ListIndex barely increases, showing that mostly existing words are inserted into the index. However, all three of these curves appear to be relatively linear, and there is no significant difference in insertion time among the three data structures when the size of the files is small. That gap in time becomes slightly wider as the size of files becomes larger, but for the vast majority of books, any of these three data structures would perform efficiently for insertion.

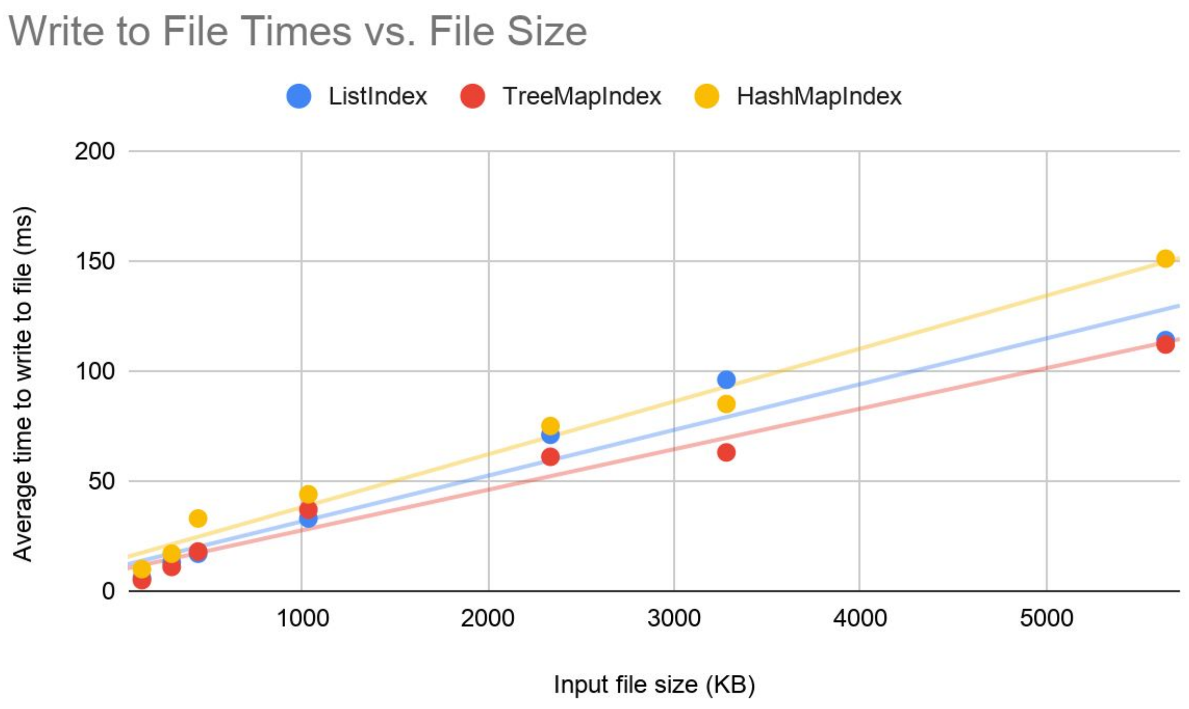

*Figure 5: The runtimes of the write to file process as a function of the input size.*

According to Figure 5, the TreeMapIndex is the fastest for writing to a file, the ListIndex is second fastest, and the HashMapIndex is the slowest. Our hypothesis that HashMap Index would be the slowest was correct. However, we predicted incorrectly that ListIndex would be faster than TreeMapIndex. In fact, the runtimes are quite close for many of the data points; in some instances, ListIndex was even faster than TreeMapIndex. This confirms our reasoning that HashMap is slower mainly because its entries have to be sorted, but we underestimated how slight the difference between an optimized map data structure and an ArrayList would be for this purpose. Even though we predicted that writing the indices to a file would take $O(n^2)$ time, the data points in this graph appear to have linear trends. This is likely because many words in each file would be unique or occur infrequently, so reading the set of line numbers for most words would take approximately constant time. Only a few outliers, such as common words like "a," "an," and "the," would require linear time to read all of their line numbers.

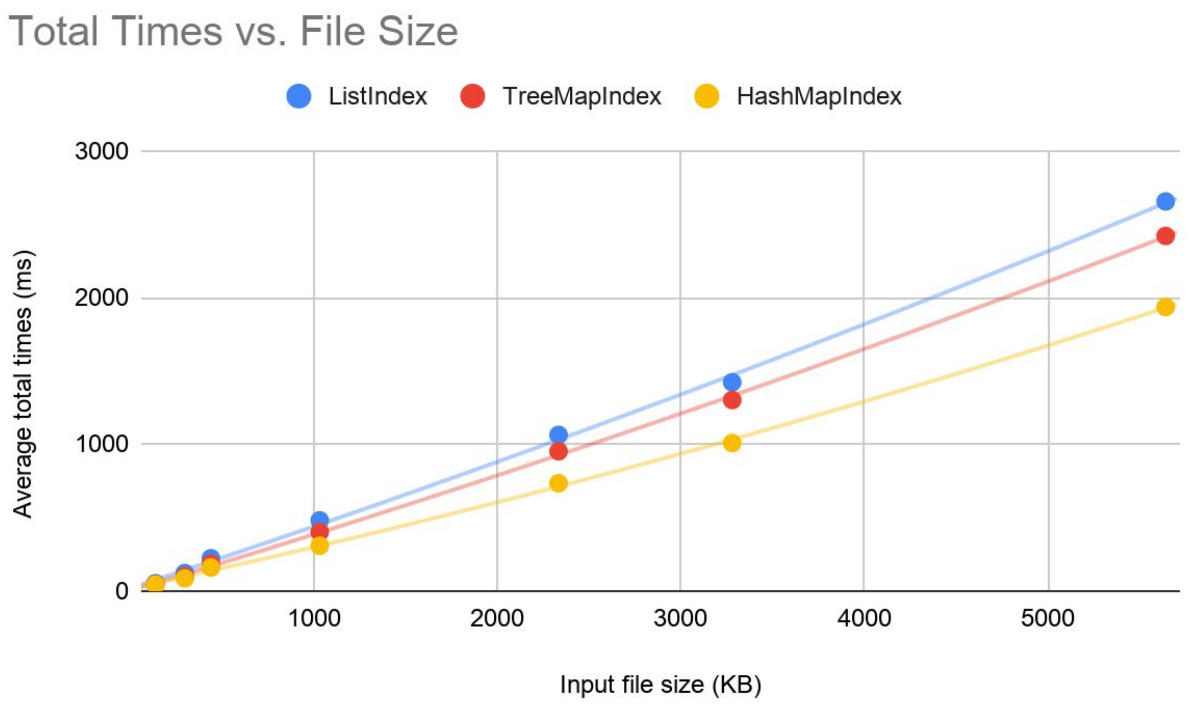

*Figure 6: The total time it takes to create the book index (insertion and writing to a file), as a function of the input size.*

According to Figure 6, HashMapIndex has the fastest runtime and hence the best performance. TreeMapIndex has the second best performance, followed closely by ListIndex. This is in line with the hypotheses that we made in the introduction. The O(n lg n) time to sort HashMap's keys is only required once, whereas ListIndex's `O(n)` insertion time for new entries occurred every time a new word was encountered, and TreeMapIndex's `O(lg n)` insertion time was required for every word. Many small additions to the runtime for TreeMapIndex and ListIndex outweighed the single, larger addition from sorting HashMapIndex's keys. In contrast, TreeMapIndex's additional `O(lg n)` time required for every insertion to find the correct TreeSet did not outweigh the $O(n^2)$ time occasionally required to insert a new word into ListIndex. Despite these differences, all three indices have fairly close runtimes; even for the largest file we tested, HashMapIndex and ListIndex only differed by about half a second. For book indices, the differences between lists, trees, and hash tables are not as pronounced as big-O notation would make them seem.

## 5. Conclusion

Between all three types of indices, HashMapIndex had the lowest total runtime, on average. TreeMapIndex was the second-fastest, while ListIndex was the slowest. Even though this order was reversed for writing to a file because HashMapIndex had to be sorted, the time required to insert all words and their line numbers into the index dominated the runtime. In addition, the potential for rehashing and collisions in a hash map did not outweigh its average $O(1)$ lookup time. However, the differences between all three data structures for this purpose were slight, even for large files. The graphs for all three indices' total runtime appeared almost linear, and ListIndex's total time for the largest book we tested did not even exceed three seconds. For a typical book, any of these data structures would be an efficient choice.

In highly specific cases that we did not cover in this project, certain index implementations may be a poor selection. If memory is a major concern, then the $O(n)$ additional storage space required to merge sort the words in HashMapIndex could outweigh its efficient performance time-wise. Books with many extremely long words would also be a poor situation to use a hash map because the time to hash words would approach $O(n)$ instead of $O(1)$. Similarly, books that create many collisions in a hash map would also make HashMapIndex a poor choice, but that would be difficult to predict from the outset. In books with many unique words, ListIndex would be inefficient because it takes $O(n)$ time to insert a newly-encountered word. However, tree maps have guaranteed $O(\lg n)$ lookup and insertion times for one word and already store words in alphabetical order. TreeMapIndex would be the best choice when an index being inefficient for a fraction of books or using extra memory would be unacceptable.

Although asymptotic analysis is an effective estimation tool, the leading terms did not tell the full story in this project. Even though we predicted that retrieving and writing an index to a file would take $O(n^2)$ time because $O(n)$ words would each have $O(n)$ line numbers, the trendlines appeared linear. Many words would only occur infrequently in a book, so reading line numbers would take closer to $O(1)$ time. Reading line numbers is also only done once for each word in the index, whereas insertion is performed multiple times for many words, so the insertion time dominated the total runtime, even though its leading big-O terms were smaller. In addition, we noted that adding newly-encountered words to ListIndex would take $O(n^2)$ time, but the trendline for its insertion time was barely curved and appeared nearly linear; quadratic operations were performed infrequently enough that they did not dramatically affect ListIndex's runtime. Similarly, rehashing and collisions in HashMapIndex were not prevalent enough to prevent it from being the fastest of the three indices.

The book index that we implemented is a good starting point to create a real index for any file. If we were to implement this for real world and practical use, we could improve our book index program by having the program omit commonly used words (like “a”, “the”, “he”, etc. ) and focus on the important words that the user would want to find. This could be done by removing common words from the dictionary. We would need to use a database of common words to remove them from a real dictionary before generating any indices. Omitting the most common words would decrease the number of total words and likely get rid of the words with the largest number of repeats, thus decreasing the write-to-file time and making the book index more useful to the user. Another way to improve the program would be to have the program only print a user specified number of most common words in the book index. We could also let the user specify words to omit in the command line and remove those from the dictionary. Conversely, we could let the user specify words to include in the index in the command line and have the index only be of those words. Furthermore, if we did not want the entries as a string or wanted to distribute the program to the public, we may have had the iterators for each index return instances of a custom Entry class, which would make each word and its line numbers accessible.

Implementing the book index was an excellent way to explore the performances of different data structures like ArrayLists, TreeSets, TreeMaps, and HashMaps. Knowing the pros and cons of each kind of data structure will help us to make better design decisions when there are certain space or time constraints in other real life applications.

## 6. References

Oracle. (2020). *Class ArrayList<E>*. Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Oracle. (2020). *Class Arrays.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

Oracle. (2020). *Class Collections.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Oracle. (2020). *Class File.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/io/File.html

Oracle. (2020). *Class HashMap<K,V>.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Oracle. (2020). *Class PrintWriter.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html

Oracle. (2020). *Class Scanner.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

Oracle. (2020). *Class TreeMap<K, V>.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

Oracle. (2020). *Class TreeSet<E>.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

Oracle. (2020). *Interface Iterator<E>.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Oracle. (2020). *Interface List<E>.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Oracle. (2020). *Interface Map<K, V>.* Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Oracle. (2020). *Interface Set<E>.* Java Platform, Java SE 11 & JDK 11 API Specification. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html

The JUnit Team. (2020). *Packages.* Junit4 Javadoc 4.8. https://junit.org/junit4/javadoc/4.8/

Xia, Ge. (2020). Project III [PDF].

Xia, Ge. (2020). Red-Black Tree 1 [Microsoft PowerPoint].