**Introduction:**

The assignment of this lab experiment was to read in a text file word for word and compare how our program processed them. I tested a total of nine text files of various sizes and made several passes through each file. I will highlight the different passes later in the Procedure section of this lab report. I created several data structures composed of linked lists. Each node in the lists contains two pieces of information: a string value of the word it stores and an int value of how many times it has appeared. The data structures that I constructed are an unsorted link list, an alphabetically linked, a front self-adjusting linked list (where if a word is already in the list, then its corresponding node is moved to the front of the list), and a bubble self-adjusting linked list (where if a word is already in the list, then its node Is bubbled-up on spot closer to the front of the list).

For the two self-adjusting data structures, if a new word is encountered, then its node is added to the front of the list. I tested the four separate data structures one at a time with the chosen text file (this is where the passes come into play). The main goal of this the experiment was to compare the data structures when tested with the same text file. The properties that are of interest to me are the number of words the file contains, the number of distinct words, how many comparisons were made between the current word and each node in the list, the number of reference changes of the head node pointer, and the elapsed time it took to make the pass for each specific data structure.

**Hypothesis:**

In this lab experiment, it is rather simple to predict some metrics of our comparisons. For example, I believe it would be safe to say that files of shorter lengths will run much faster than those that are of much greater lengths. For this lab, I chose nine different text files of various lengths, I have them ordered from smallest to largest: Green Eggs and Ham (extra-small), Hamlet (small), half of Moby Dick (medium-small), a concatenation of To Kill a Mockingbird and half of The Three Musketeers (medium), Moby Dick (medium), Moby Dick concatenated with itself (medium-large), a concatenation of Don Quixote and Bleak House (large), the King James Bible (large), and all of the text files compiled together (except for the concatenated files) plus the addition of War and Peace(extra-large). I also believe that the shorter files should have fewer comparisons and reference changes because there is less data to work with.

For the four separate data structures, I think that the unordered list will have the longest elapsed time (because it may take a long time to traverse the list for every word) and the front self-adjusting list will have the shortest (because the most frequent word will always be at the op of the list, this traversal may not take long for each word). In terms of the number of reference changes, I speculate that unordered list should make the fewest and the front self-adjusting will make the most (because if the word is new or previously found, it will always change the head node pointer). For the number of comparisons made, I believe that the front self-adjusting will make the fewest (because reoccurring words should stay near the top of the list) and the alphabetically ordered list may make the most. However, this is all just speculation. I will later see how my predictions compare to the data I collect from various tests in the Observations section of the report.

**Procedure:**

I began the experiment by creating classes that the four list classes would implement. These classes include a linked list, sorted linked list, node, node iterator, word, and word counter. Once I established my base classes that I would use for this project, I constructed the four lists. The first was the unsorted list. The unsorted list was meant to function in the way that it would add a new node to the front of the linked list if the word had not been encountered before. If a word had be previously encountered, then the count of the node the word belongs to would be increased by one. I then constructed the alphabetically sorted list. This list is would add a new node to the list if the word had not been encountered and would place the node according to its alphabetical value. If the word had already been found, the count of the node was increased by one. The front self-adjusting list would add a new node to the front of the list if the word had not yet been found. If the word was already found, then the node would be moved to the very front of the list. The final list I made was the bubble self-adjusting list. This list would bubble-up the node one spot closer to the front of the list if the word had already been encountered. If the word was new, then add a new node to the front of the list.

Once I had all of the classes I needed to run the experiments, I gathered multiple text files that I could use to conduct trials of my lists. I previously mentioned the text files in the Hypothesis section. These text files will be used to test the behavior of the four lists and of their properties (elapsed time, comparisons, reference changes) and I will be reviewing my results later under the Observations section.

**Observations:**

I began my observations by recording the first 100 words of the front self-adjusting list and the bubble self-adjusting list for the Hamlet text file. The resulting graph for first 100 words for the front self-adjusting list can be viewed below:

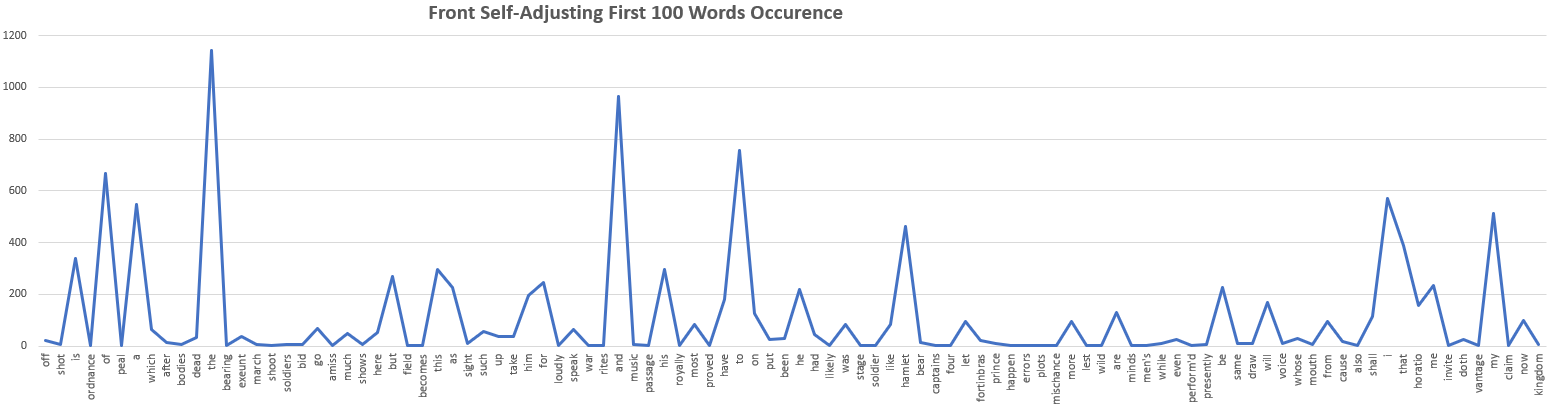


Figure 1. Graph displaying the first 100 words of the front self-adjusting list and their occurrences.

The top three words with the highest occurrences from the graph above listed from greatest to least is ‘the’ at 1142, ‘and’ at 966, and ‘to’ at 755. The graph’s occurrence value goes from 0-1200. There is a wide variety of word occurrences in the first 100 words of the front self-adjusting list. Now, lets take a look at the graph for the first 100 words for the bubble self-adjusting list below:

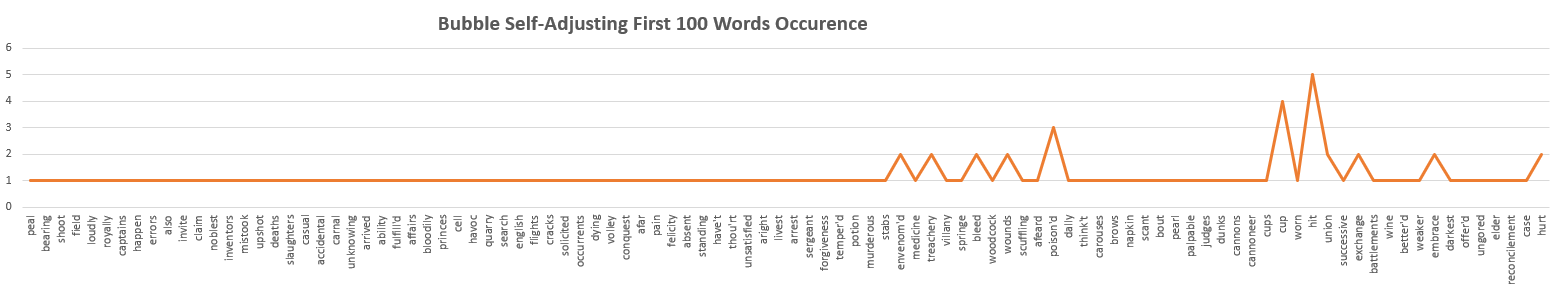


Figure 2. Graph displaying the first 100 words of the bubble self-adjusting list and their occurrences.

The top three words with the highest occurrences from the bubble self-adjusting graph from greatest to least is ‘hit’ at 5, ‘cup’ at 4, and ‘poison’d’ at 3. For this graph, the occurrence value goes from 0 to 6. The first 100 words for the bubble self-adjusting graph do not have much to any variety in regard to occurrences. It is obviously apparent that the two graphs look nothing alike, and I believe this has to do with the nature in which the two self-adjusting lists are constructed. The front self-adjusting list has a lot of variety because every time a high occurring word appears, it is always moved to the very front of the list. This gives a greater amount of diversity in the list. The bubble self-adjusting list has little diversity because the high occurring words can only move up the list one spot at a time. Thus, the words that only appear once will always be placed at the front of the list.

Once I finished with comparing the first 100 words from both of the self-adjusting lists, I compiled the elapsed time it took the four lists to run all nine of the text files separately. After I ran all of the separate tests in the Eclipse IDE, I compiled the data into Excel and created the following graph:

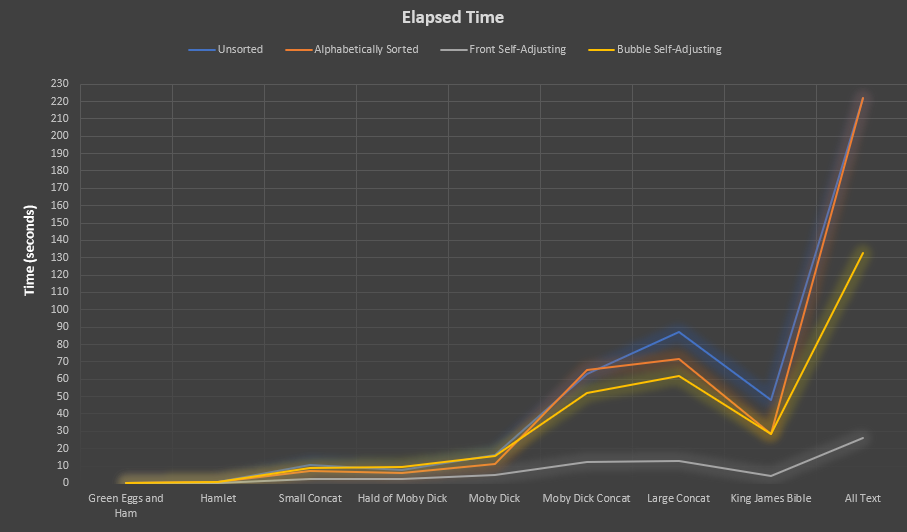


Figure 3. Visual display of the elapsed time it took each list to process the separate text files.

As seen from the graph above, on average the unsorted list took the longest to process the text files and the front self-adjusting list performed much more efficiently than all other three lists. However, there were cases when the alphabetically sorted list and the bubble self-adjusting list took slightly longer to run than the unsorted list. As the text files grew larger though, it became apparent that the unsorted list is the slowest of the four lists.

We have established that the unsorted list performs the worst, but now we need to analyze further into why it behaves in this way. From the data that I collected, I noticed an interesting trend: the unsorted list always had the highest number of comparisons made for every text file. Thus, it can be deduced that the as the number of comparisons increase, so does the elapsed time it takes to process the file. This phenomenon can be seen in the graph below:

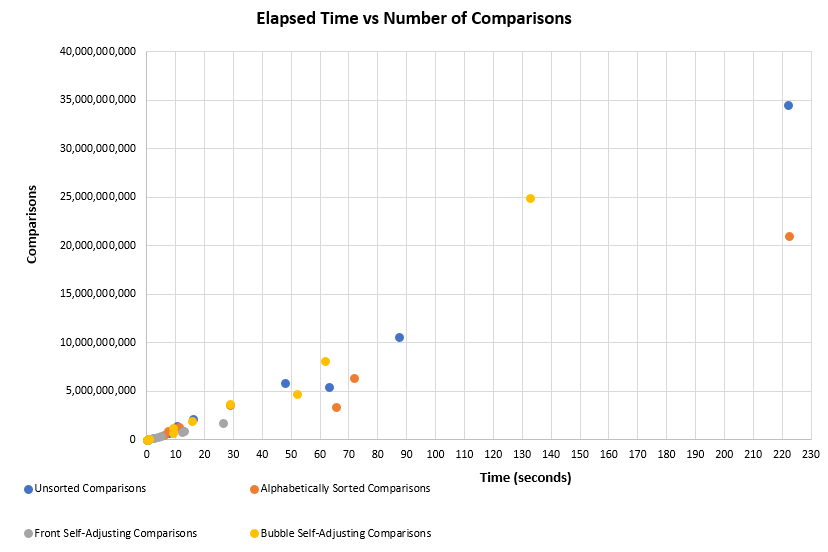


Figure 4. Elapsed time vs the number of comparisons made.

From the graph above, you can easily see the linear relationship between the elapsed time and the number of comparisons. In every case the time increases as the number of comparisons increase. It can also be noted that for each text file, the list with the most comparisons had the greatest elapsed time to process the text file.

After testing the relationship between elapsed time and the number of comparisons, I began to conduct an investigation on the notion of whether or not the number of reference changes had any effect on the elapsed time. From what I was able to find out, it appears that the elapsed time and number of reference changes have no correlation or relationship. The visual below helps to further this claim:

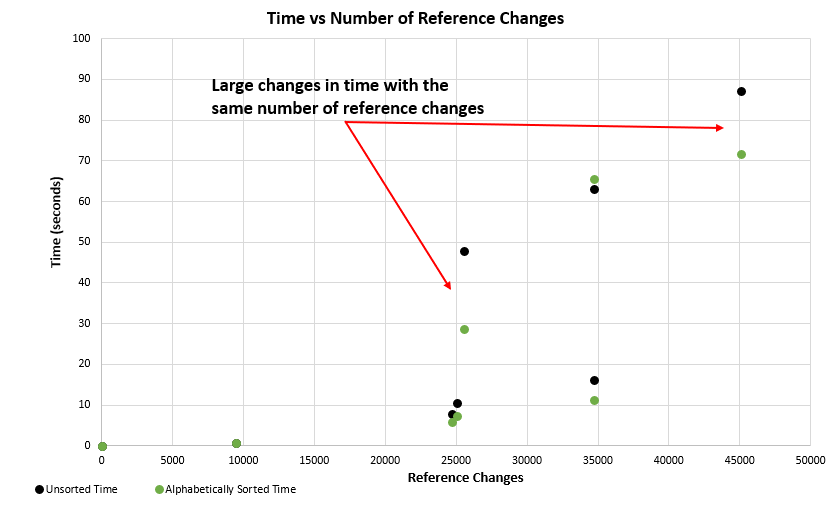


Figure . Elapsed time vs the number of reference changes.

As stated in the graph, there are many points where there if a substantial difference in elapsed time for the same number of reference changes. This seems to be happening consistently throughout all of the data that I recorded. Thus, it can be inferred that there is no relationship between elapsed time and number of reference changes.

The final phenomenon I wanted to take a look at is whether or not the number of words and/or distinct words correlated to time. To conduct this test, I have two very informative cases that I will be able to help us understand. In the first case, I compared the data from the concatenation of To Kill a Mockingbird and half of The Three Musketeers with the data from Moby Dick. These two text files have virtually the same number of total words, but a Moby dick has roughly 30% more distinct words than the concatenation text file. The data can be viewed below:

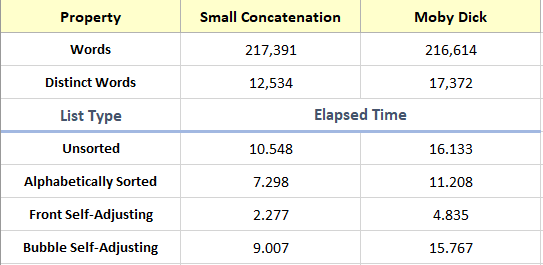


Figure . Data overview of the concatenation file (To Kill a Mockingbird and half of The Three Musketeers) and Moby Dick.

From the data above, you can see that both files are very similar in terms of number of words, but Moby Dick has a greater number of distinct words. Also, it can be seen that the elapsed time it took for each of the lists to run the files is much greater for Moby Dick. How can this be when they have basically the same length (in terms of total number of words)? I will be answering this question momentarily, after my overview of the data form larger text files than these.

Along with the comparison between these smaller text files, I also tested the same ideas with files that are almost four times larger than these. This data comes from a large concatenation text file (Don Quixote and Bleak House) and the King James Bible. My results that I recorded can be viewed in the table below:

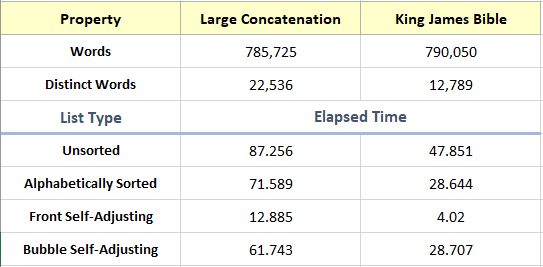


Figure . Data overview between a large concatenation file (Don Quixote and Bleak House) and the King James Bible.

It can be seen from the table that the large concatenation file has less words, but almost twice the number of distinct words than the King James Bible. This is mainly because I constructed the experiment in this way so that I had a file with a much more extensive vocabulary. Even though the large concatenation file has less words than the King James Bible, it takes over twice as long to run for each of the four lists (three times as long in some cases). It is now that I will be answer the question as to how this can occur. This is made possible because the large concatenation file has so many more distinct words (i.e. a greater vocabulary). The number of distinct words determines how long the linked list will be for each of our four lists (or how many nodes you have to traverse in order to find the correct comparison for your word). Thus, as the number of distinct words increases, so does the elapsed time it will take to process a text file.

**Conclusion:**

There has surely been a lot of data represented in this lab experiment and I will briefly summarize everything that I have gone over so that nothing is left out. I began this experiment by constructing all of the necessary code run the tests on the nine different text files. There were four lists I created: an unsorted linked list, and alphabetically sorted linked list, a front self-adjusting linked list, and a bubble self-adjusting linked list. Once all of the lists were constructed, it was time to start testing against the text files.

The first tests were on analyzing the first 100 words of the front self-adjusting list and the bubble self-adjusting list. It was found that the front self-adjusting list had more diversity than the bubble self-adjusting list. This was due to the fact that the high occurring words in the bubble self-adjusting list could only move up one node at a time, while the front self-adjusting list moved the node to the very front of the list when it occurred.

The second study that was conducted was recording the elapsed time it took for all four lists to run each of the nine text files. It was found that on average the unsorted list performed the worst, while the front self-adjusting list out-performed the other three lists in every case. After I had found this out, I tried to dive deeper into the data as to why the four lists behaved this way. The reason had become obvious: the unsorted list made the most comparisons in every case, while the front self-adjusting list made the least. A graph was produced to further this reasoning and it was found that the number of comparisons had a linear relationship with the elapsed time. As the number of comparisons grew, so did the elapsed time. This is because the more comparisons that are made, the more traversing you have to do through your lists, which will inherently increase the processing time.

The third study that was made was between the number of reference changes and elapsed time. I knew that the number of comparisons were affecting the processing time, but I was curious if there any other factors at play. From a few simple tests and with the help of Excel, it was found that there seemed to be no correlation between the number of reference changes and the elapsed time. There were many cases when the number of reference changes stayed the same, but the elapsed time varied immensely. This led me to verify that the was no direct relationship between the two.

The fourth and final study was conducted on the number of words, distinct words, and elapsed time. I created two cases: a small concatenated file (To Kill a Mockingbird and half of The Three Musketeers) and Moby Dick, while the other was between a large concatenated file (Don Quixote and Bleak House) and the King James Bible. I chose a small and a large case to see if any factors changed when the size of the text file increased. From the two cases, it was found that while concatenated files had the same word count as their respective counterpart, the elapsed time it took the four different lists to run them varied. This occurred from the number of distinct words. In the two cases, it was file with the larger number of distinct words that took longer to run. Thus, I concluded that file with more distinct words means that it has more nodes in its linked list, which in-turn increases the overall processing time.

Overall, this project was challenging, but at the same time satisfying to see it all come together in the end. I was able to learn a lot as to how data structures differ from each other in terms of optimization and efficiency. It was good practice to be able to compare and analyze the data from the four lists I constructed. There were certainly aspects of this experiment that I can use on future projects and that will help me to become a better programmer.