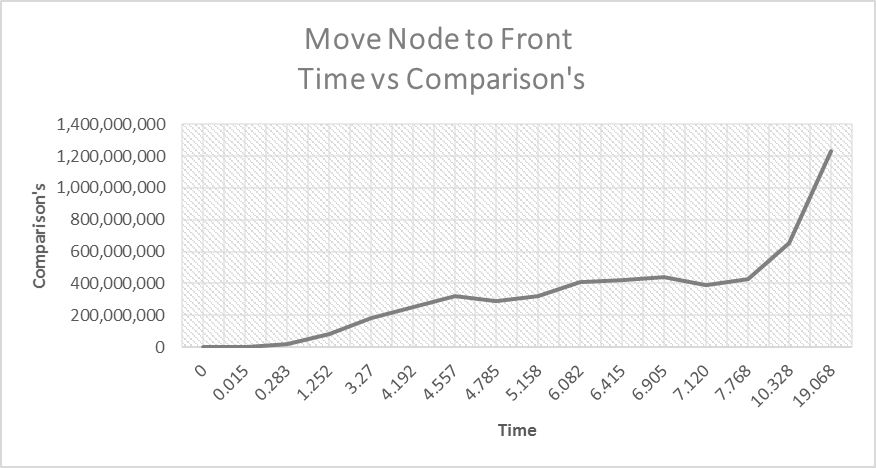
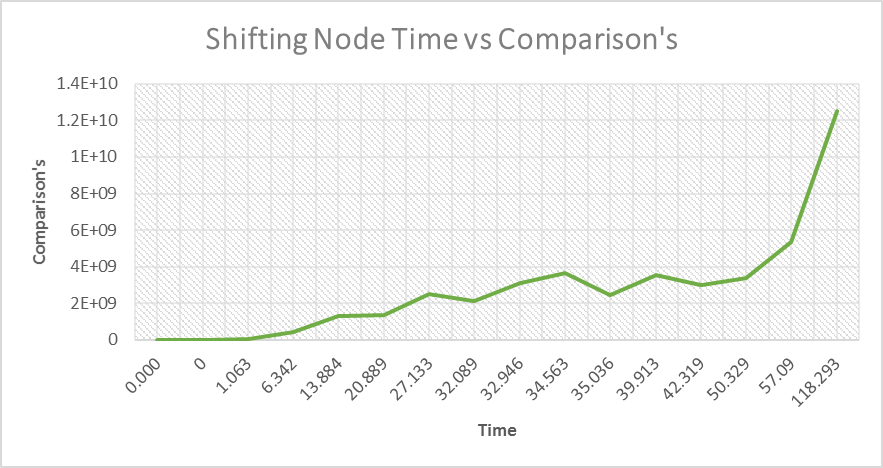
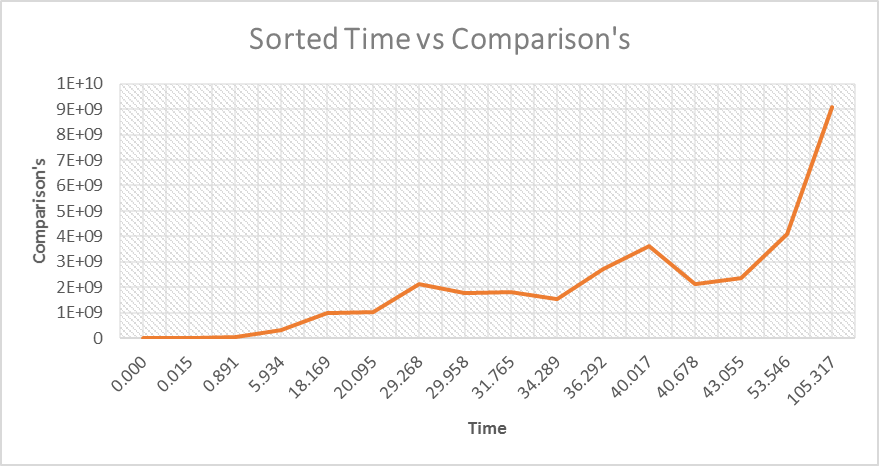
**Introduction:**

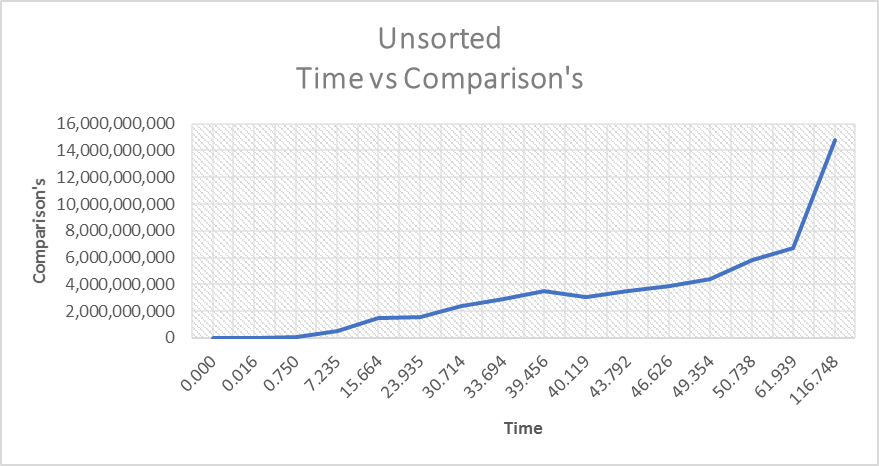
This lab asked us to test four different types of list against various text files that differed in both the size of the file, total words in the file, and distinct words that were contained in the total word count. The first, thing that was asked of us was to first read in the file and parse it to see both the total words, and the time It took to just parse the file, we then took a similar approach but with four different types of list, a unsorted, sorted (alphabetically), a list were every time an occurrence of a word happened it was moved to the front and the count was incremented, finally we implemented the final list by shifting a node up one towards the front every time it occurred in the file. For the four list types we collected many statistics consisting the time it took to put the words in the list, total words, distinct words, total comparisons it made, and how many times the reference was changed. Finally, we are asked to collect the data for the first 100 nodes put into the list of the last two types of sorting algorithms (moving node to front and shifting node to front) and put them into excel and create a graph.

**Prediction’s:**

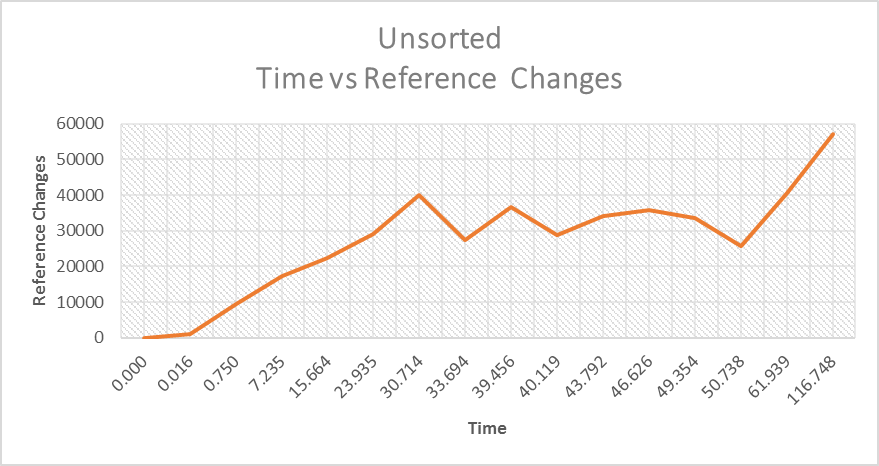
My prediction for the four types of list are that they will all act in a similar fashion with time, comparisons, and reference changes for the smaller types of files like “Green Eggs and Ham” and “Hamlet”, etc. As the files grow based on total number of words, the list that moves the node to the front and sorted list I feel will be the fastest out of the four list. Giving that the list moves the node all the way to the front every time it occurs saves a lot, it covers a lot of ground, even if a new node is added it is only pushed back one spot, saving a lot of time when it comes to trying to find a specific word in the list. For the last list, shifting a node up one doesn’t help a whole lot unless the file is small, if the file has a lot of distinct words then it is just adding new nodes to the front of the list, pushing the ones that occur the most to the back. With unsorted it is just hard to predict, my opinion is that it will of course run slower based on the size of the file, but if the words in that file are in such an order that it doesn’t have to traverse the whole list a lot, then you will have a good time, on the other hand all the words you need could be at the end of the list and you have to traverse the whole list each time giving a longer time.

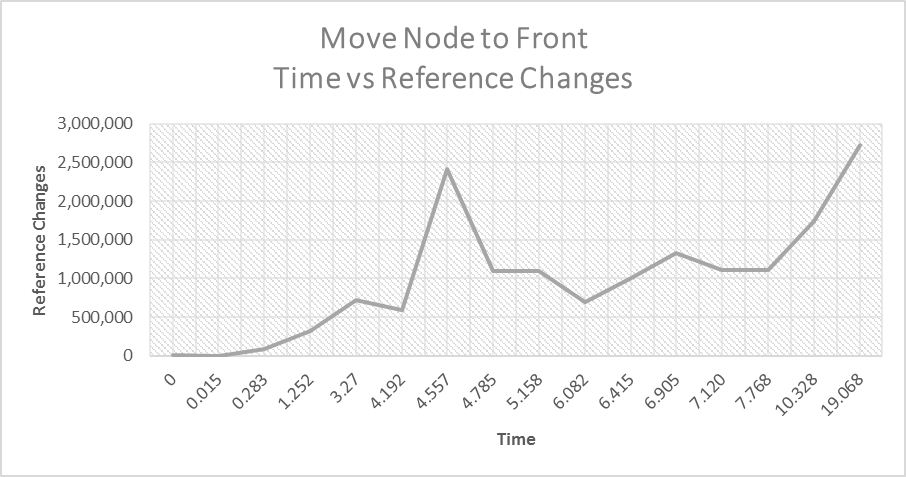
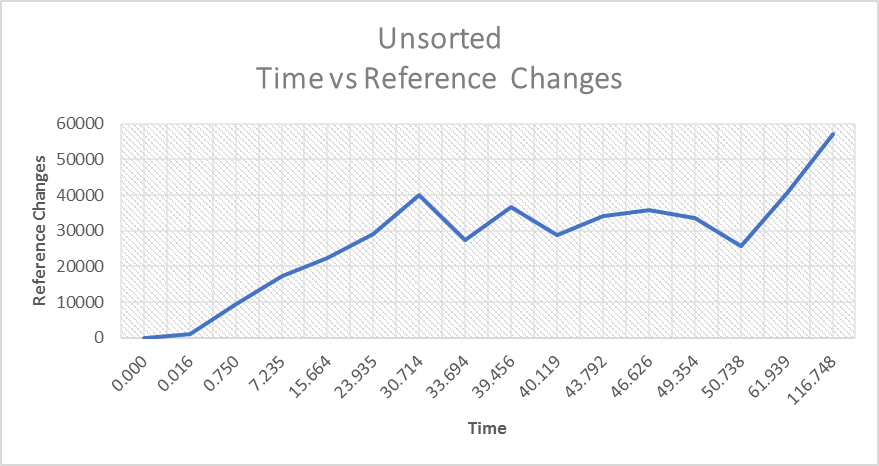
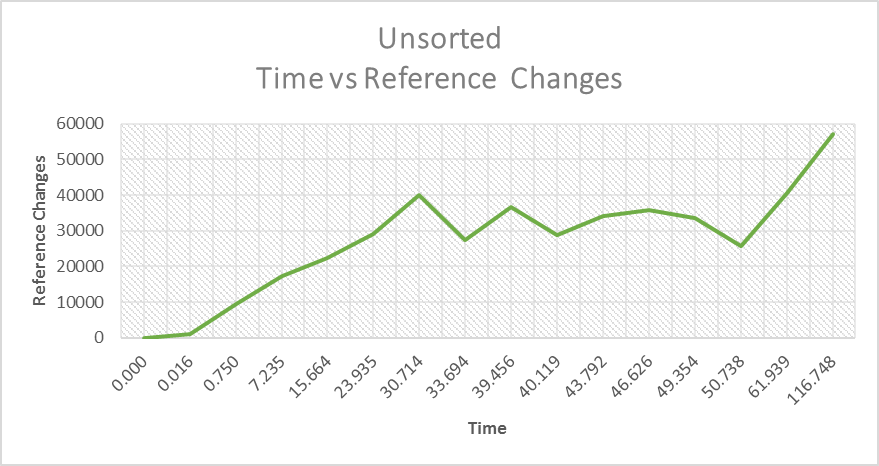
**Results:**

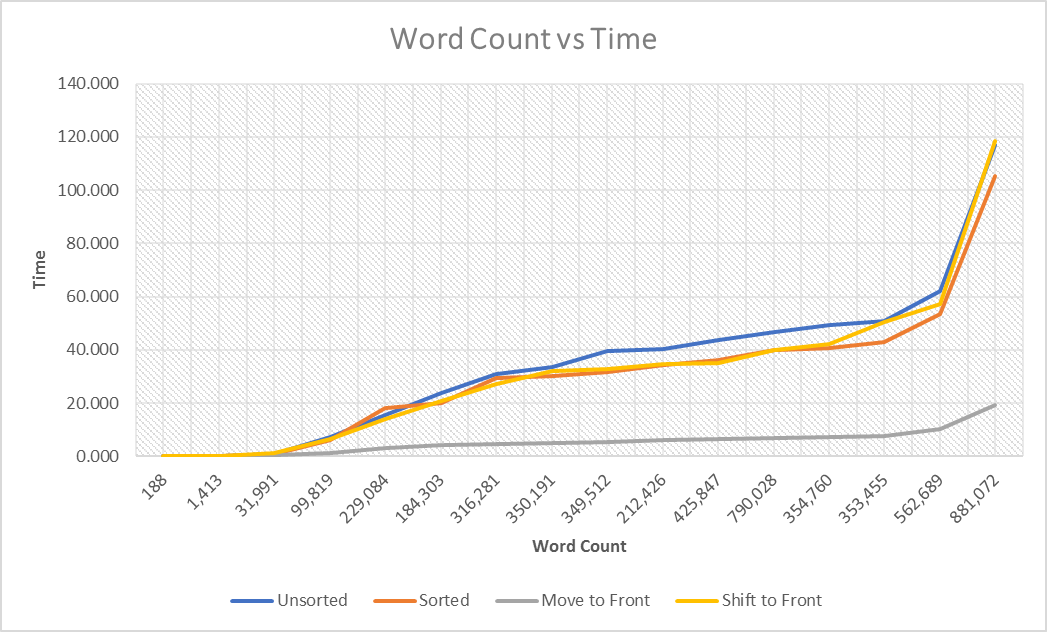
****The results from this project were very interesting and mind blowing. First I decided to compare the time it took for a file to process and the comparison’s that it made it made on that specific file. Shown below is the four graphs for the four list showing a Time vs Comparison graph:

****

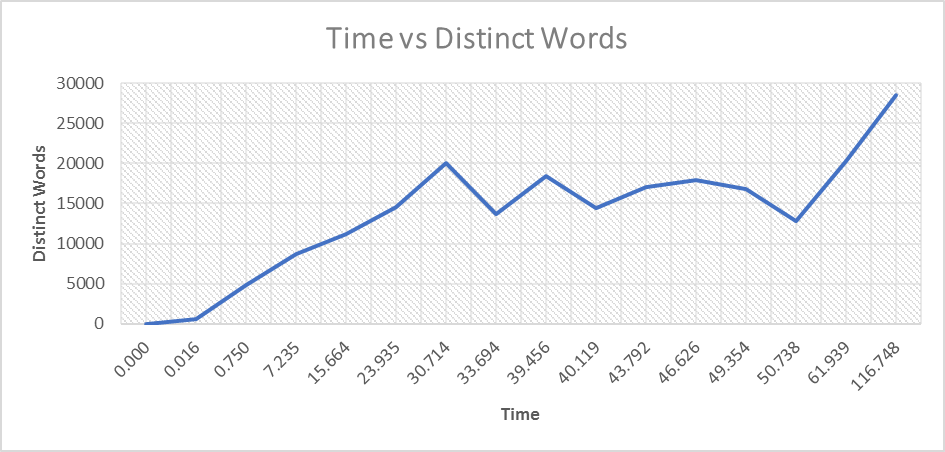
As shown in the graphs, it makes sense that as the time goes up the comparison’s for that file will go up as well because there are more words in the file to be processed and parsed. It also makes sense that the Move to Front list has the least amount of comparison’s out of the other 3 list, because since the node being put to the front is the one that is already on the list and occurs again, meaning that it may not have to go through the whole list to find the word, it might be at the first couple nodes and not have to compare that many times.

**** Next, I compared the time it took to process a file with the total reference changes that list did. It would make sense that the move to front list would have a ton of reference changes because it has to change the front of list pointer and current and previous location many times in order to move the specified word to the front of the list. Sorted, unsorted, and Shift should be the same.

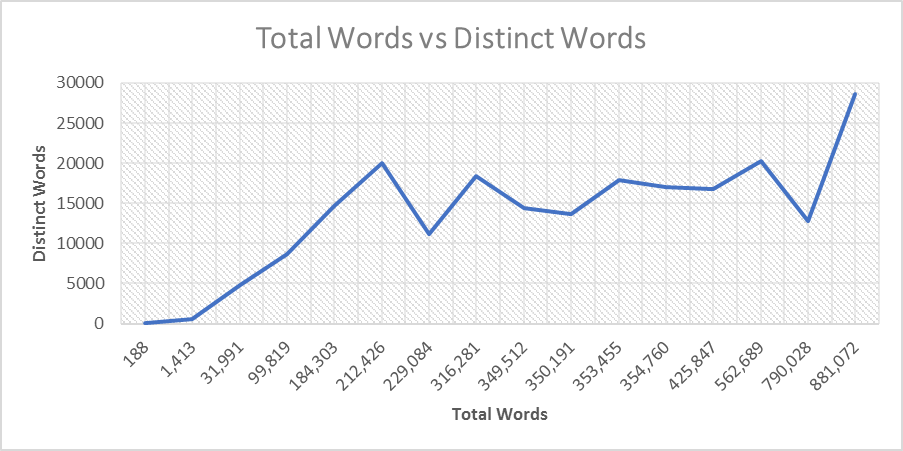
****

****It then makes sense to compare the graphs now based on their word count and the time it took to process that word count. Logically it made sense as the file got bigger the time it took to process the file would get bigger as well.

The list that moved the node to the front of course ran the fastest since it is very handy that the node is in the front of the list. Other than that the other three list ran similar except a couple of files. The sorted makes sense that it would run faster for some files, I saw kind of a trend with the lists comparing the number of distinct words with the time. The more distinct words that were in a file, would cause the lists to run a lot slower because there are more words to add to the list, there are some files that processed faster since they have limited vocabulary to work with but other than that the graph followed as expected.



Next, I compared the total words and the distinct words together to see how they affected the list. A lot of the files make sense, whereas the size of the total words gets bigger the distinct words will grow as well. Again, the bible is an outcast, it has a ton of words in it but not that may distinct words in the file. Shakespeare as less total words than the bible but it has double the distinct words the bible does which could affect how the list will process the data.



I decided to show the results for the Shakespeare file based on the recommendation from Dr. Thomas himself and because it is such a good read and have come to memorize the word count from so many test runs. I collected the first 100 nodes from the two-final list (Moving to front and Shifting up one) and put the results into a graph showing where the words are in the list and how many times a specific word came up in the text file.

**List #3: “Shifting Nodes to Front of List”**

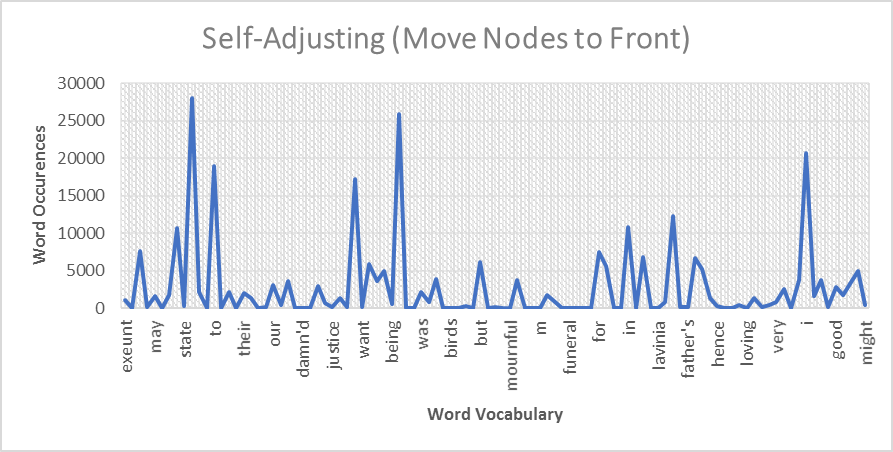
**Total Words:** 881,072

**Distinct Words:** 33,961

**Comparison’s:** 16,069,686,829

**Reference Changes:** 2,205,202

|  |  |
| --- | --- |
| **Test Run’s on Shakespeare** | **Time Elapsed** |
| Run 1 | 24.226 sec |
| Run 2 | 23.978 sec |
| Run 3 | 24.345 sec |
| Run 4 | 24.256 sec |
| Run 5 | 23.968 sec |
| Average Time | 24.155 sec |



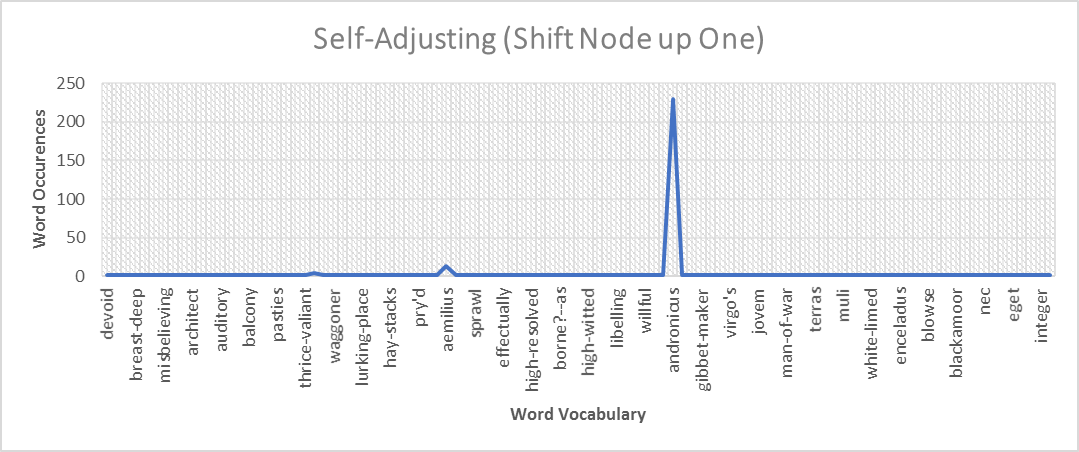
The graph proves why the send to front method works so well is because you can see that the words that occur the most are at the front of the list. This helps not having to traverse the whole list as much trying to find the word and see if it needs to add a new node or not to the list.

**List #4: “Shifting Node up One towards Front of The List”**

|  |  |
| --- | --- |
| **Test Run’s on Shakespeare** | **Time Elapsed** |
| Run 1 | 109.568 sec |
| Run 2 | 110.235 sec |
| Run 3 | 109.974 sec |
| Run 4 | 111.012 sec |
| Run 5 | 109.471 sec |
| Average Time | 110.052 sec |

**Total Words:** 881,072 **Reference Changes:** 57,136

**Distinct Words:** 28,568 **Comparisons:** 12,535,075,897

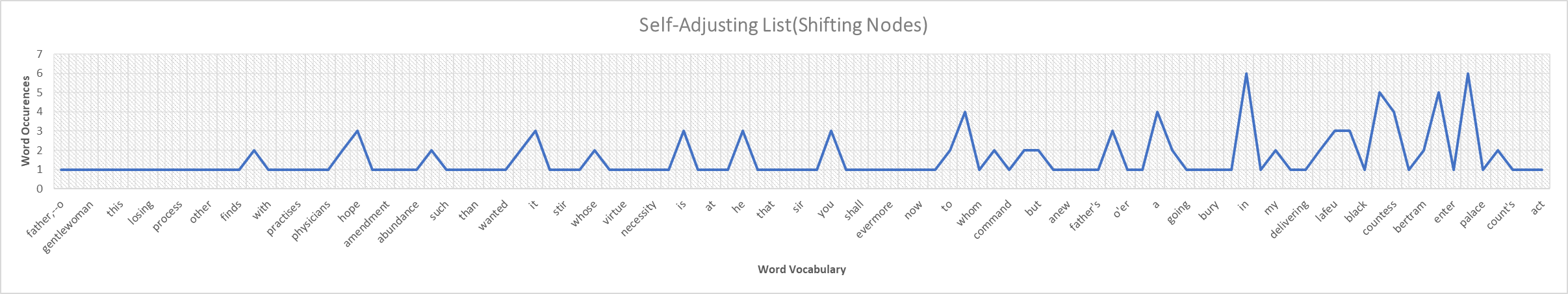
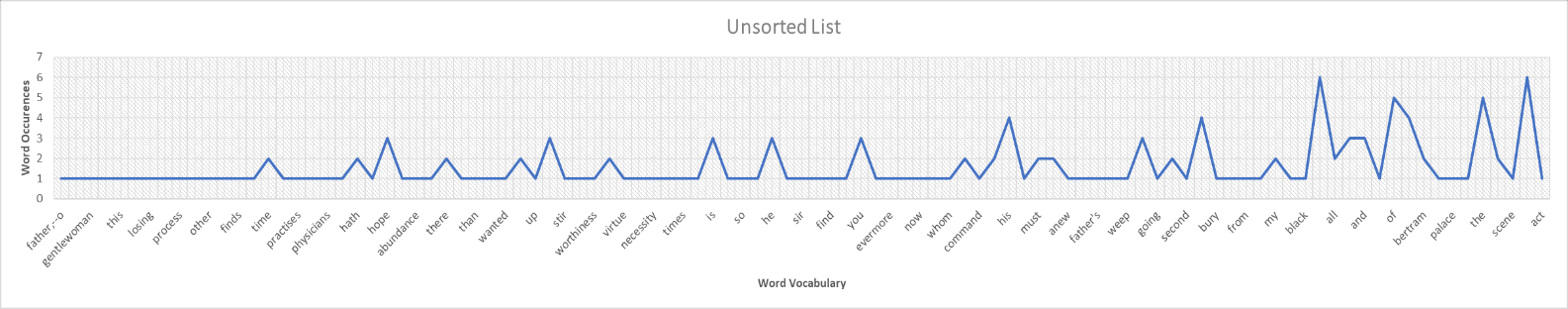


The graph of this one looks crazy at first glance, but it makes sense considering that one of the words occurs 230 times in the first 100 nodes of the list. Towards the end of the list there could be a lot of words that haven’t been added to the list yet and would be added to the front pushing the words that may occur a lot back towards the end of the list.

**Conclusion:**

The results from this project was very cool, it was interesting watching the times changes based on the text file size and how the different types of comparisons varied based on the list being studied. It was very interesting because when I ran into rough spots when coding, it was nice to debug and go step by step and seeing what the program is doing and see if the code I had written was working properly and to try and find where I may have screwed up.

Starting with the first graph (Moving node to front) the graph behaved as expected, with a lot of the words that occurred the most in the text file towards the front and the words that occurred not so often towards the back of the list. You can see that towards the back of the list there was a word that occurred a lot in the text but it’s in the back, the word “ward”. This could happen, where the word happens to come up a lot towards the beginning but may not come up again for a while and get pushed to the back of the list by a bunch of new words that have occurred. This is what makes this list so extremely fast compared to the other three list because it moves it to the front and if the word is already in the list, it has a very good chance that it’s in the front and won’t have to traverse the whole list trying to find it.

 The next graph (Shifting node up one) also behaved as I predicted. We can see the shifting the node up one in the list doesn’t really do a whole lot if you are looking for speed. When moving a node up one in the list, it doesn’t cover a whole lot of ground, and may or may not have 1 or 2 words push it back towards the list. Other than the function of shifting it up one node, it almost acts like an unsorted list. Of course, on bigger files this list would be better than an unsorted list if you had to choose, but on smaller files, you could compare graphs and see the would get similar results. A shown below is a comparison between the unsorted list and Shifting List and can see that the produce similar results.

The unsorted list acted as expected when it came to the size of the text file getting bigger, it took longer to process. My prediction for the sorted list was off though, based on the number of distinct words In the file, made the sorted list run slower, which makes sense, because one you are adding more nodes to the list and also you don’t know if you have a lot of “a” , “b”, or “c” type words that would be at the front of the list, if the word starts with a “z” then it has to traverse almost the whole list.