**Introduction:**

In this lab, it asked us to build off our previous lab where we implemented the four different types of list. First, we are going to implement a new list called “Skip List” where it will do the same as the other four list and process the various text files. Next, we are to add a remove method to each of the list and collect the time it takes to remove all of the words. For the heavy handed list we are going to implement two remove methods where remove method 2 will move it to the front of the list once the node is decremented making it faster to remove words from the list. As well we are to implement a new sorted list that hold the reference to the last node location that was put into the list that compares the word there to see if it is after that location or before which will speed up data processing. Back to skip list, the way it works is it will add a node to the list, and put it on level 0 or lane 0. Once the word has been added to the list, a coin toss is made in order to decided it the word should go up to the next level. If the coin toss decides to have the word go to the next level then it will go up, if a new level needs to be created then we will create a new one and then add the word to the new lane. The word will continue to go up through the levels as long as the coin toss tells it to do so, otherwise once the coin toss returns false then it will stop and continue to the next word.

**Predictions:**

My predictions for how the skip list will work is that it should be much faster than the other list by a long shot. It will be able to find words much faster since one, the list is sorted and two it has different levels with various words in it that it can climb down in order to find the desired word. It can easily check if the word is in the list since it is sorted, say there are three levels in the list and the top level contains the words “a” and “d” and the target word is “d”, well we will only make a total of 2 comparisons and realize it’s not in the list since we know that the word “b” does not come after the word “d”. Next, I believe that the remove methods will take a little bit longer than adding it to the list except, the removing will have more comparisons because the first word in the list will be the last word in the file, and when we check the first word in the file it will be towards the end of the list and therefore making a lot of comparisons when trying to find the word. For the removing for the sorted, heavy-handed list, and skip list, I believe that those will of course remove the words from the list much faster since the words that occur more are towards the front.

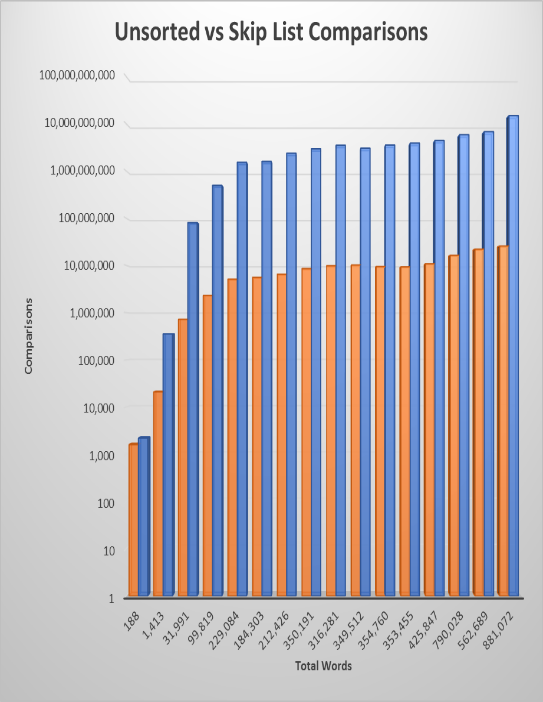
**Results:**

Results collected were as predicted which is good. First, is I decided to create a graph that compared all of the list together plus the skip list now included in the graph that compared their times with the total words of a certain text file. Shown below is the graph of the 5 different list:

The graph is similar to the graph collect in lab 2, but now it includes the skip list. Clearly the skip list runs significantly faster than the other list. The skip list it seems that the list is almost constant time for all of the list, which isn’t entirely true, it just seems like that compared to the other list because it is running and creating the list for the text file so much faster. Shown below is what the skip list looks like on as graph by itself. It can be seen that the list is not constant even when the text file gets larger, it just runs the files significantly faster than the other list do.

The skip List is very superior compared to the other list when it comes to time when processing a text file. The skip list was able to process the ALL.txt file in under 5 seconds, which is insane considering the text file consist of 5,716,639 words, it was leaving the other list in the dust, the next fastest list was the heavy-handed list processing the text file in about 159 seconds.

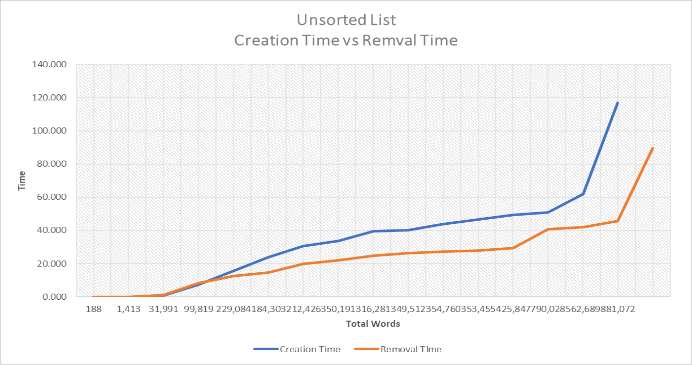
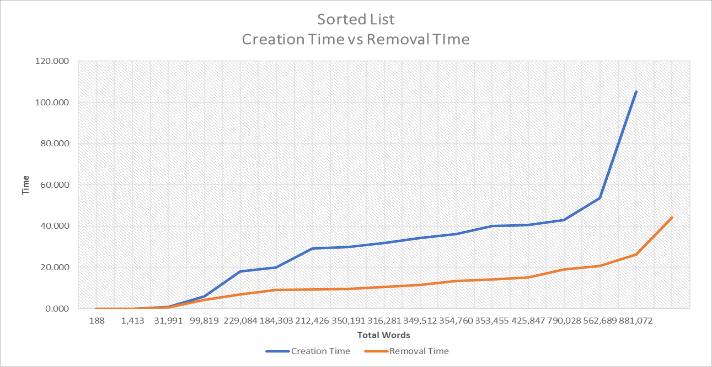
Next, I decided to observe the comparisons of all the list together comparing how well skip list did compared to them, maybe that is why it did everything so fast is because it didn’t make as many comparisons as the other list. Once again as shown in the graph below is skip list blowing the other list out of the water with the least amount of comparisons made.

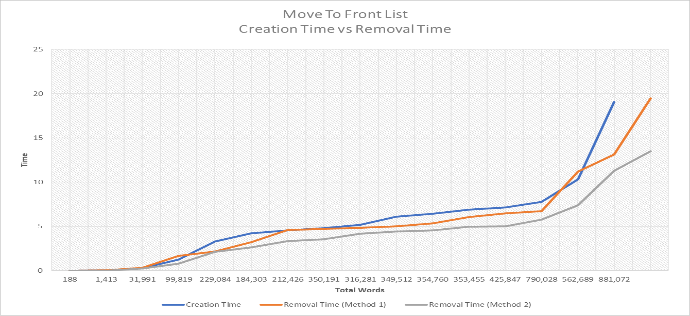
 It is insane that the skip list does so well in everything so far, it process the files much faster than the other list and has way less comparisons than the other list as well. It makes sense that the fewer comparisons the list makes, the faster the list should process. To the right I decided to make a bar graph of the unsorted list since it makes the most amount of comparisons and the skip list since it makes the least amount of comparisons. You can see the skip list barley goes over 10,000,000 comparisons while unsorted is touching 10,000,000,000 which is a lot more comparisons!

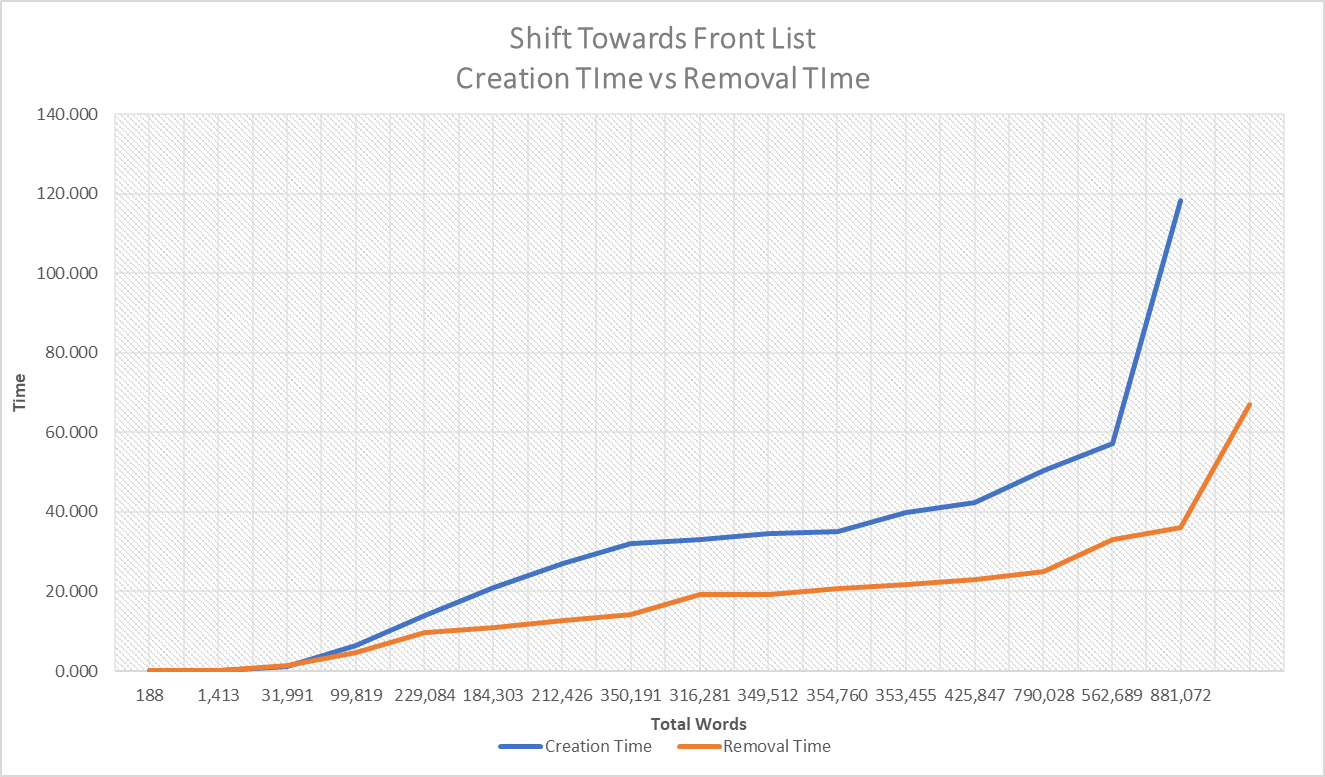
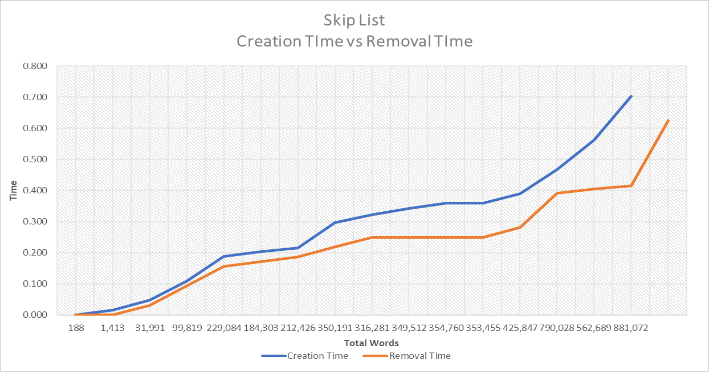
Next, I decided to compare the data on how well the list were able to remove the words from the list, how many comparisons the made, and the reference changes they made. The reference changes for unsorted, sorted, and shift will all be the same as the total word count since it goes to each node that many times and either decrements the count or removes the node from the list. The heavy handed list and skip list reference changes varies depending on how many times it has to move it to the front or with the skip list how many references it has to change in order to remove a word from the list. Below are the time vs total words for removing the words of the 5 different types of list:

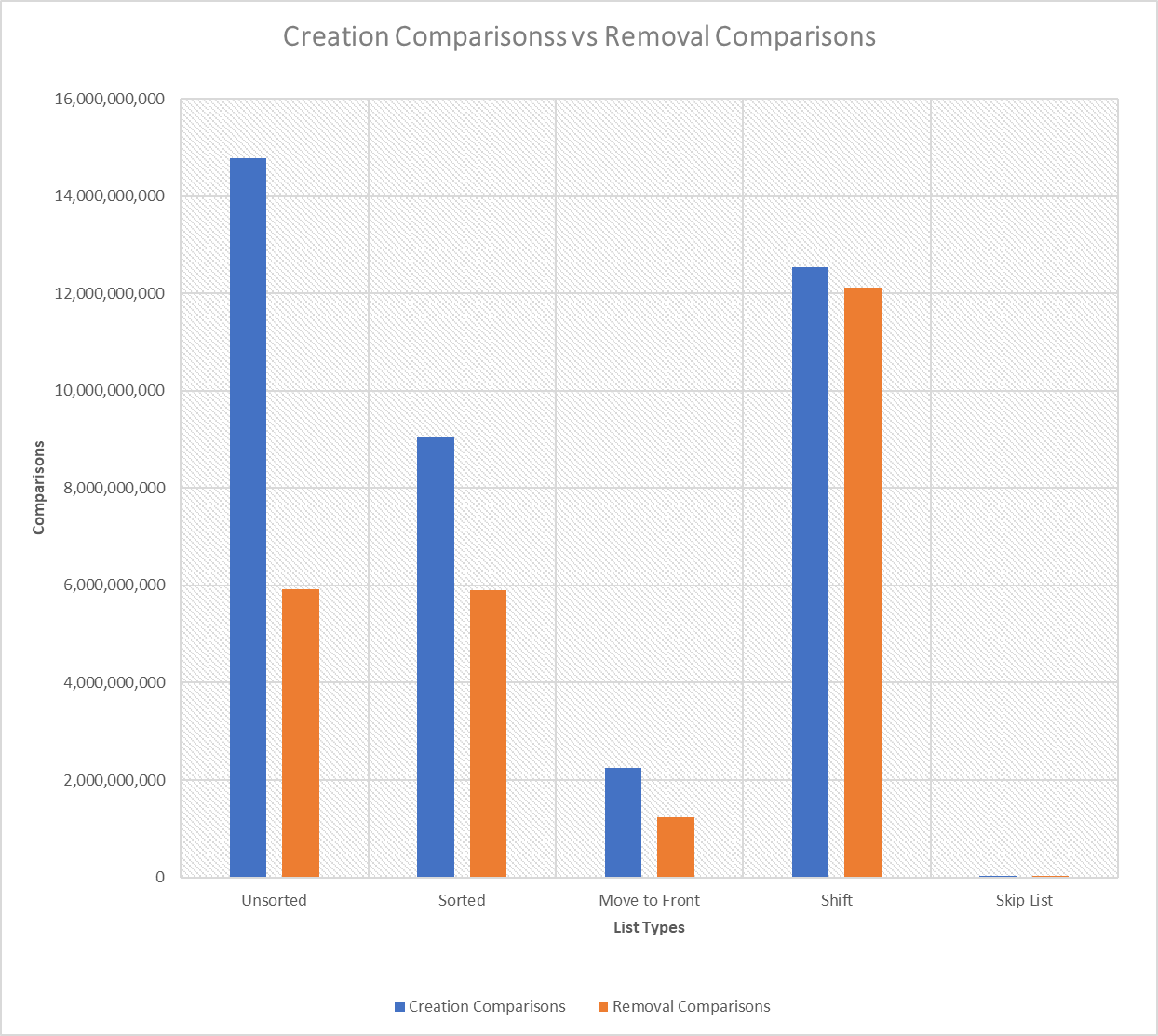
The graph shows the data for all the list including both the removal methods for the heavy-handed list. It makes sense that the skip list removes it the fastest since it is able to find a target word pretty fast. As well the removal 2 method for the heavy-handed list processes faster than the normal remove since it is moving the word to the front of the list and has a good chance that the word just moved to the front will occur again very soon.

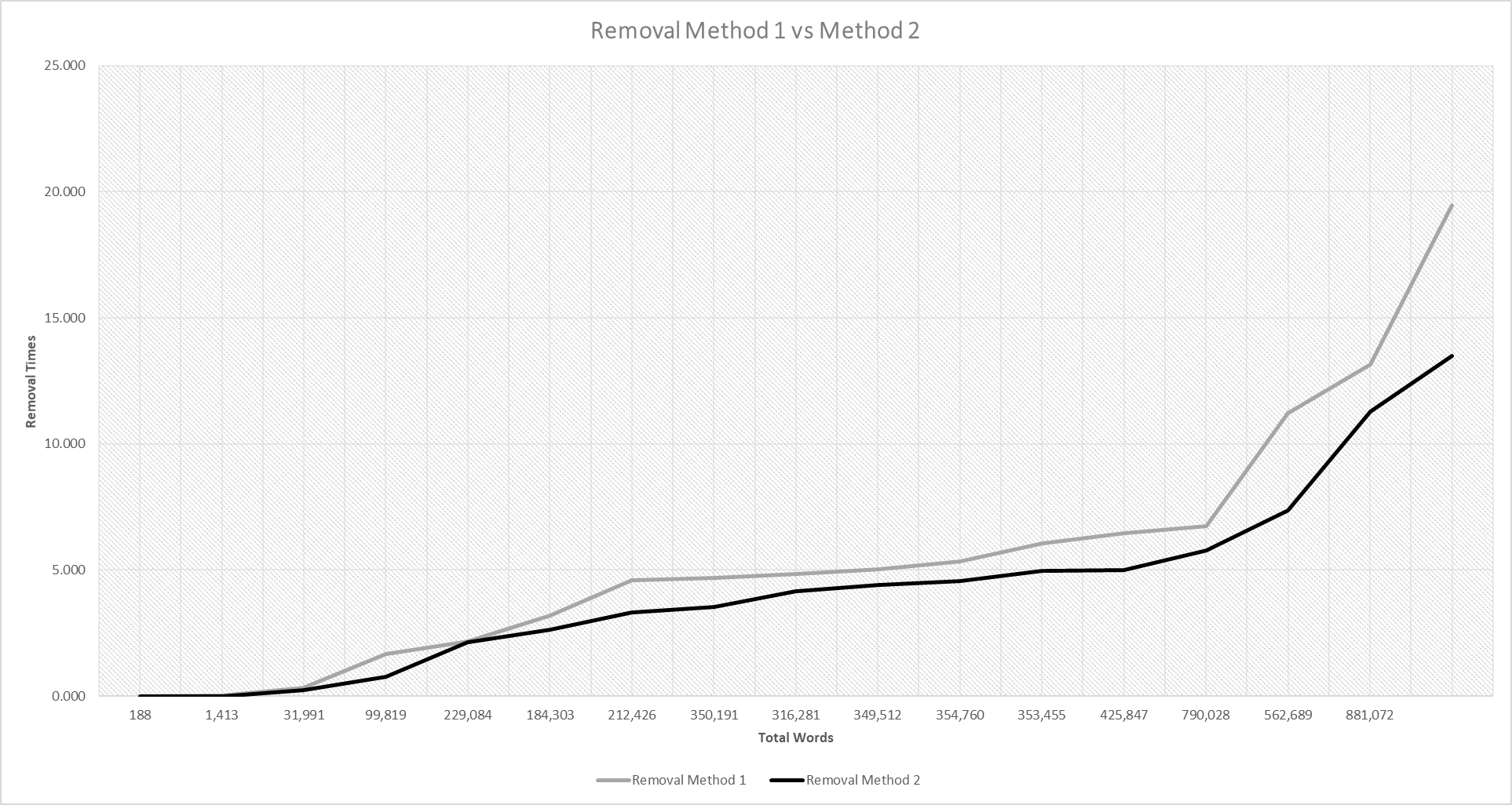
Looking at the graph more closely and observing that the max time on the graph is under 100 secs. With creating the list, the max time was 140 seconds as shown on page one of this report. Clearly, we can see that removing words from the list processes faster than adding them, why is that? The comparisons for removing are lower, of course the time will be faster since it doesn’t have to make as many comparisons when trying to find the word. As well it takes more time when trying to add a node to the list, for removing all you must do is one line of code to remove a node by setting previous to previous.getLink().

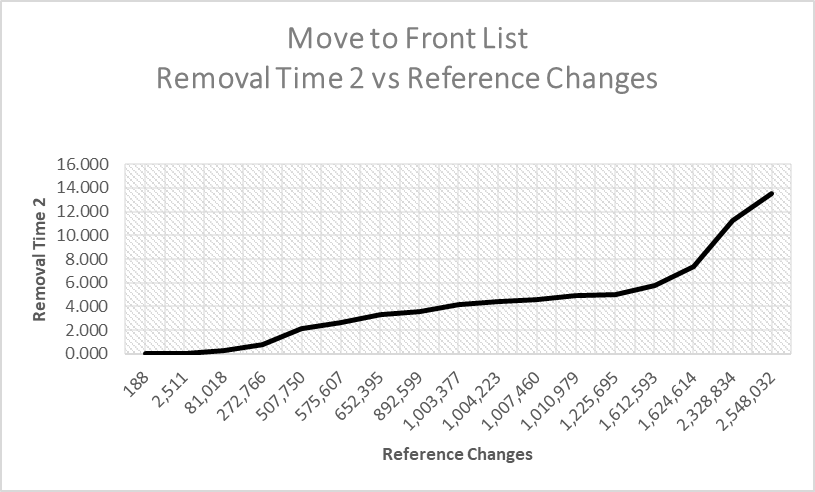
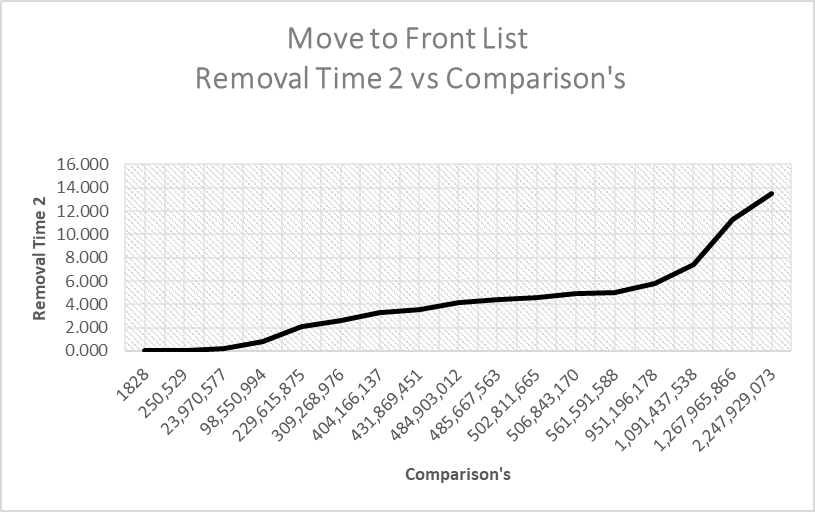
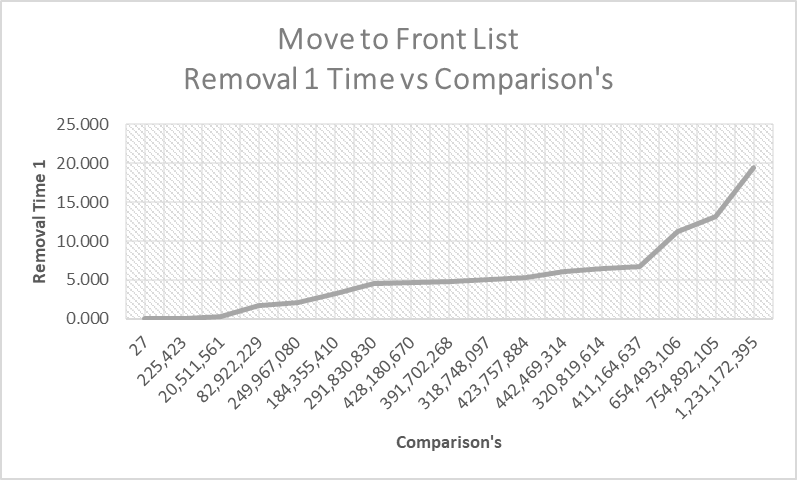
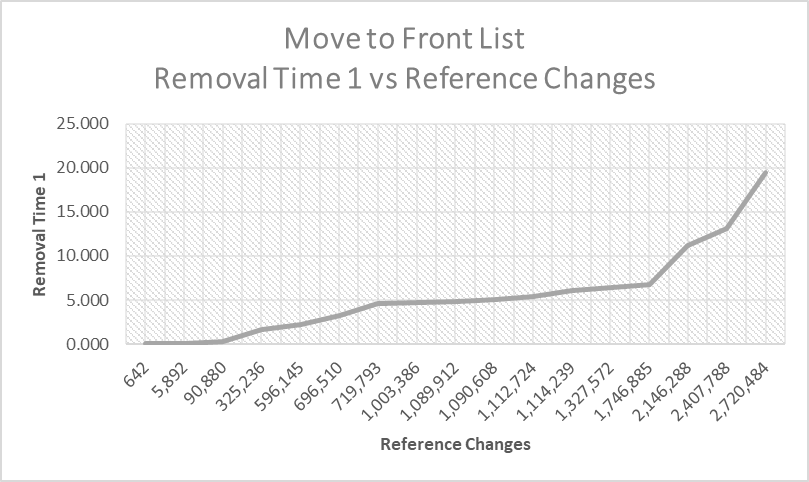
Now comparing the creation time for a list and the removing list times for each of the individual list would also show that removing a node from the list does process faster rather than adding a node to the list. The creation times vs removal time graph for each of the five list are shown below based on the total words of the text files:

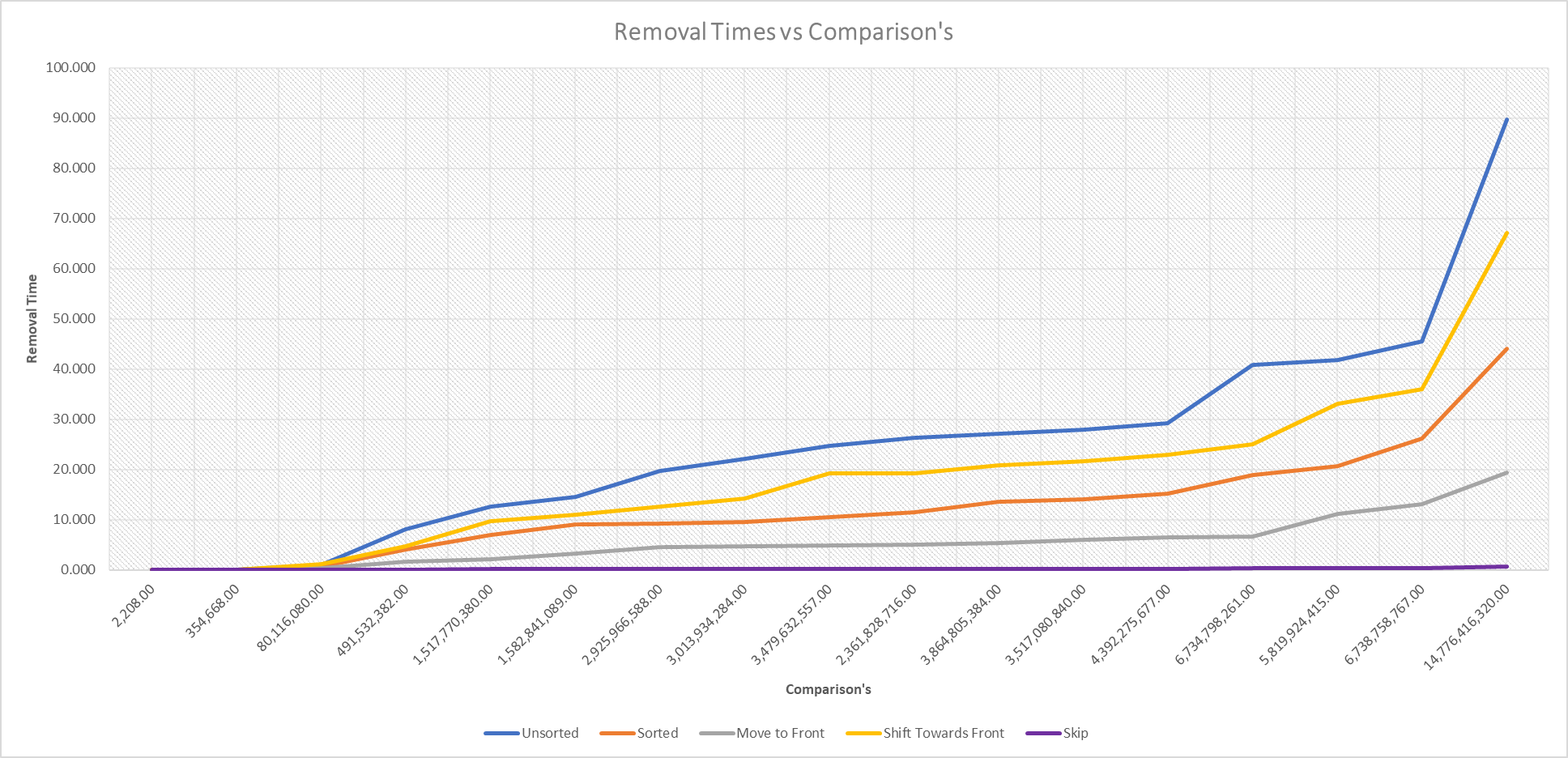


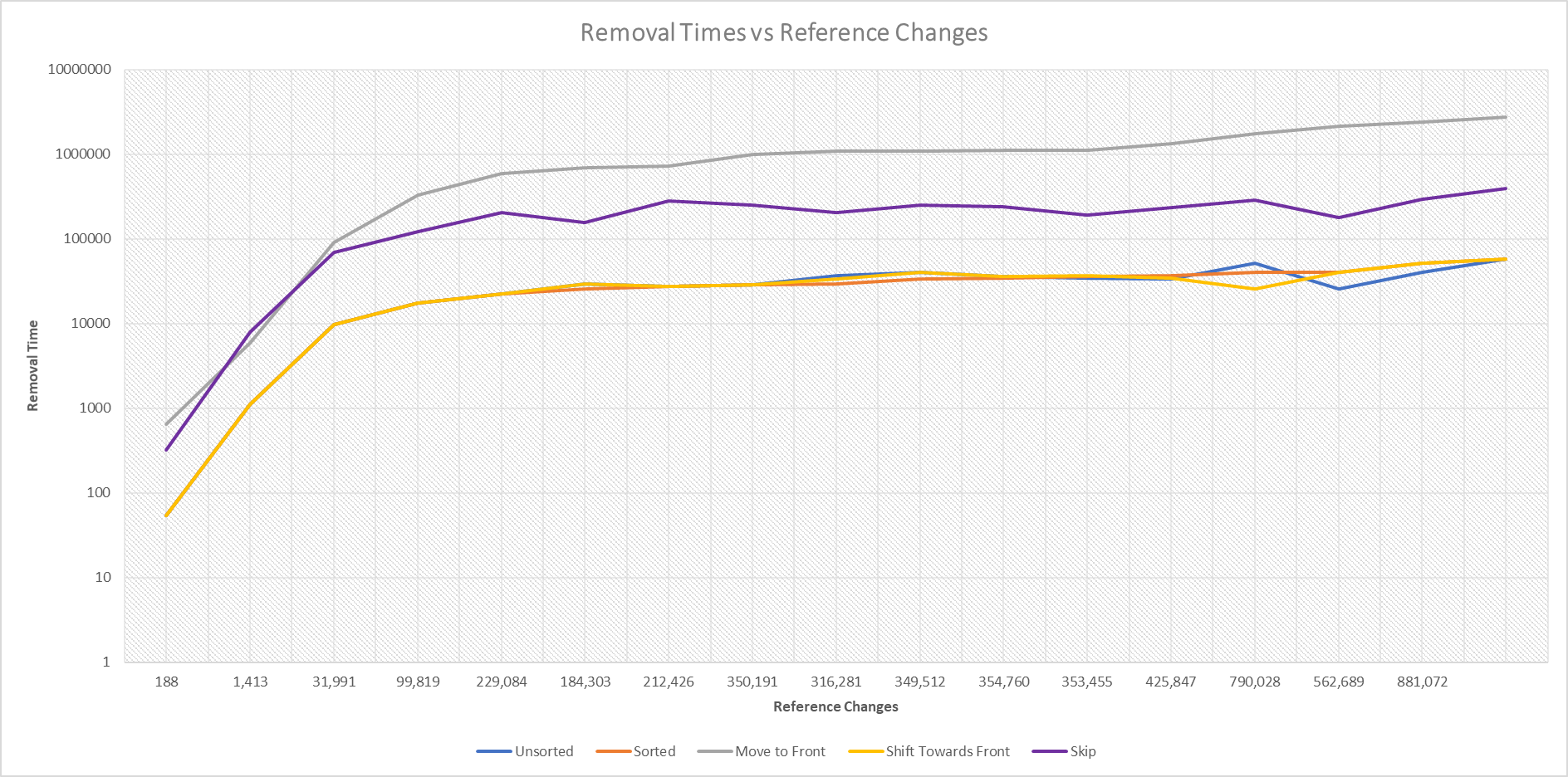


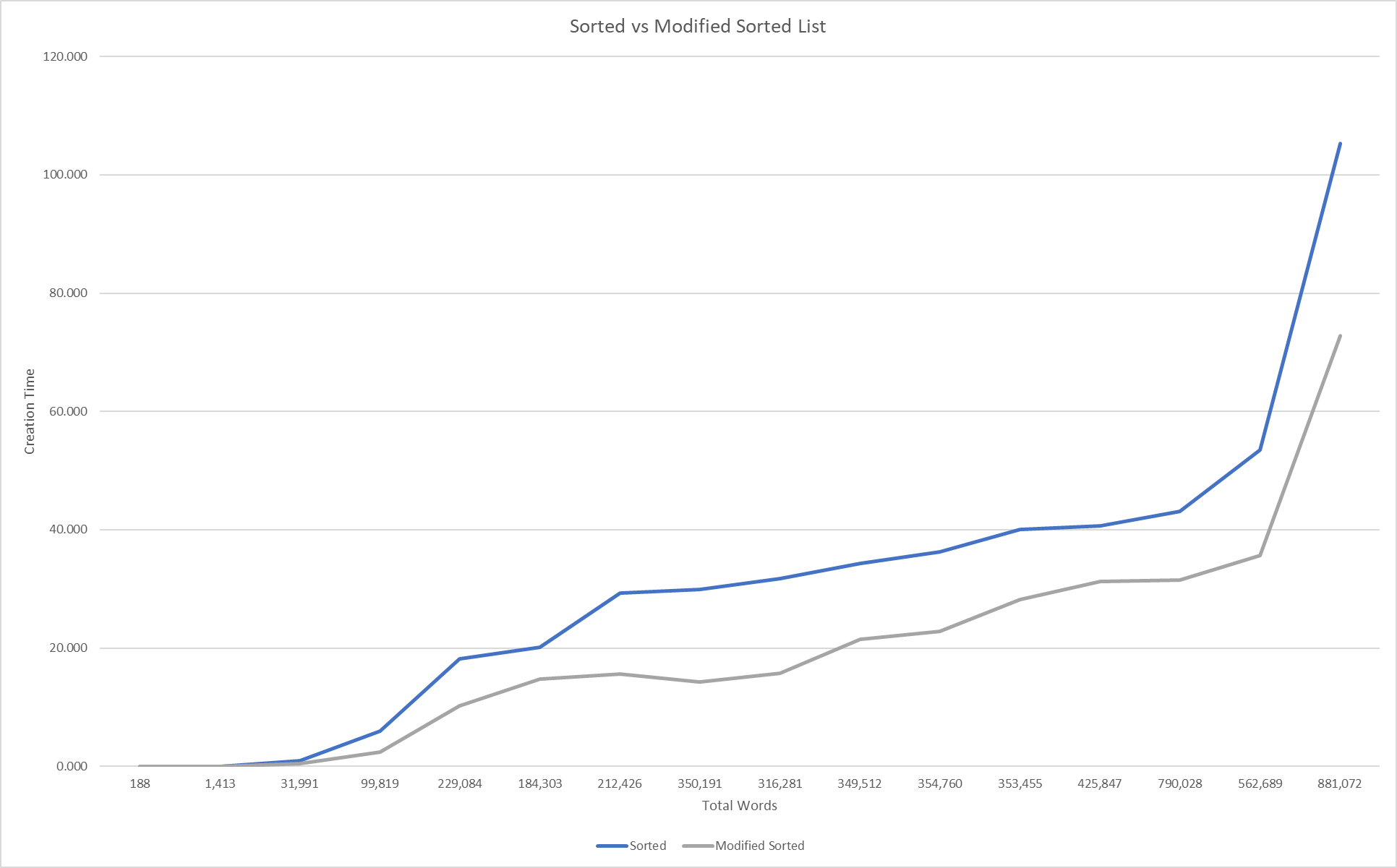
 Graphs prove that removing a node from the list performs faster than adding nodes to the list. Since we implemented two remove methods for the heavy-handed list, the orange line is the typical remove method and the gray line is method 2 moving the word to the front every time the word is decremented. Since knowing the comparisons for both creating the node and removing the node it supports our statement of why the remove method has a faster time. Shown to the right is a bar graph of creation time comparisons vs removal time comparisons for each of the five list, blue is creation and orange is removal:

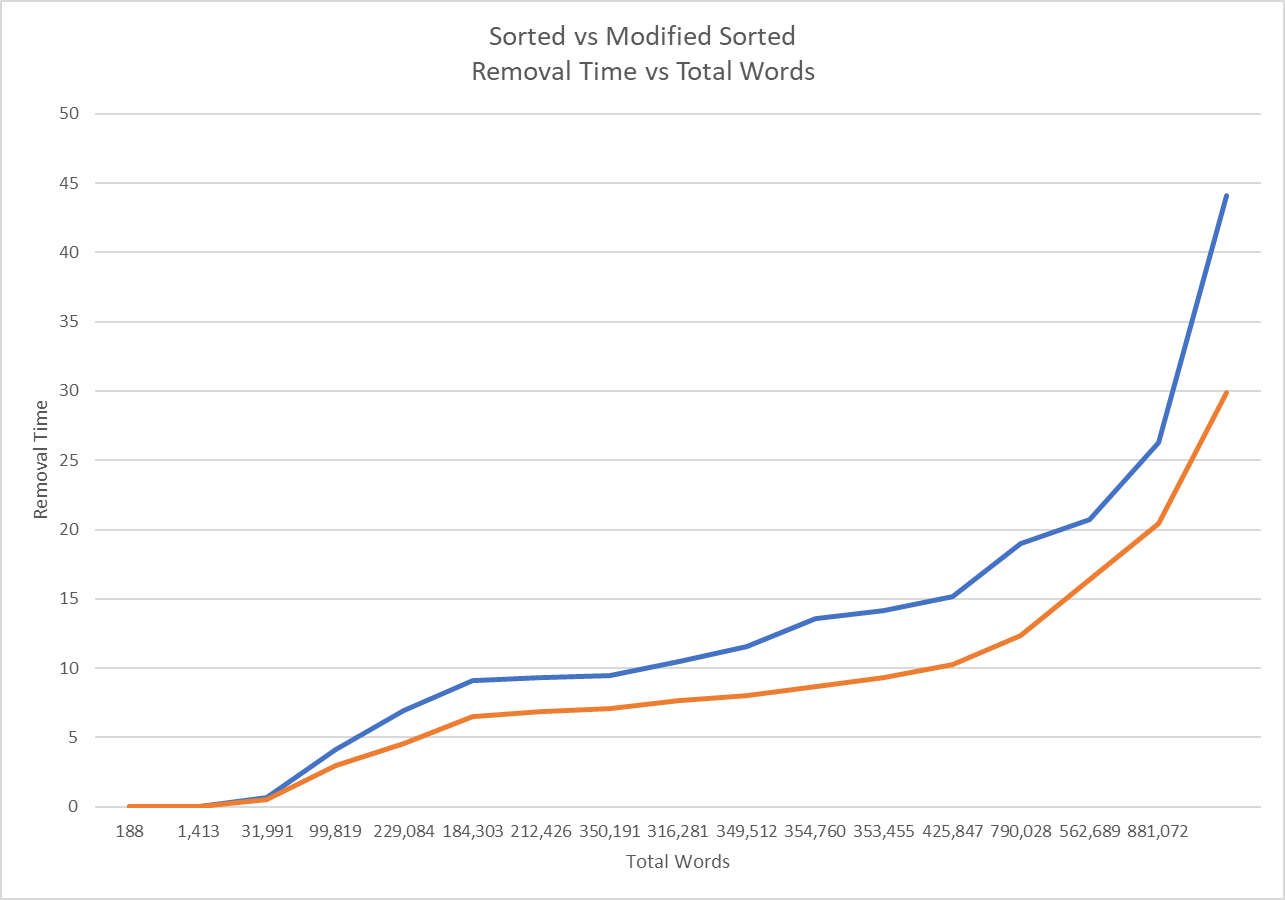
 Next, I did a graph comparing the times for the two remove methods for the heavy-handed list. Thinking about the two remove methods and how they work it should make sense that remove method two should run faster than the typical remove method all the other list are using. Remove method 2 works good as the size of the file gets bigger, for the smaller files the two remove methods for about the same. Below is the graph comparing the times based on the size of the text file:

 The graph proves that the two remove methods are just about the same for the smaller types of files, but as the various text files got bigger remove method 2 started pulling away and removing the nodes faster. It’s cool to see how implementing different tactics to remove nodes from a list can affect everything when it comes to time, comparisons, reference changes when processing text files of different size. The remove method 2 will have more reference changes the remove method 1 since it is moving the node to the front of the list almost every time. On the other hand, it is making way less comparisons when finding a target word in the list since remove 2 is moving the word to the front and it may occur again very soon and may not have to search the whole list.

Then, I decided to compare all the list based on both their comparisons and their reference changes they made on each of the files. As discussed earlier in the report, comparisons are directly proportional to the time that it takes to process the file, makes sense since fewer comparisons means it found the word a lot faster most of the time. It would make sense that unsorted list would have the most amount of comparisons since it has no particular order and just throws the words in the list however they come from the file. So, the word it may be looking for could be at the end of the list, well how many times would traverse almost the whole list? That’s the kind of thing that makes unsorted list so slow. Below is the graph for the comparisons vs. the removal time:

Next, the references that each of the list made for a specified file. I saw a trend that the more reference changes the list made, the faster it ran. Which makes sense because it is changing references a lot with the heavy-handed list and the skip list. Since it has to take the time to move the list to the front when trying to remove the node it has to hold a lot of reference in order to make that happen. As well with the skip list, it changes reference like crazy when setting all the links to null, and if it has to remove the lane that takes a lot as well. As show below is the reference changes vs the size of the file for each of the list and it can be seen that skip list and heavy-handed list have the most reference changes made:

**** Finally, I compared the two different types of sorted list, the sorted list in lab 2 and the modified sorted list implemented in lab 3. The main function of the modified sorted is was to hold a reference to the location of the previous incremented or inserted word. When the next word being studied was checked for in the list, it word first check at the location we decided to hold a reference to, this checks to see is I before or after that word. If the word is before that word then we know that we start checking at the front of the list, if it comes after then we check at the second part of the list. This helps this list add data much faster since maybe half of the time depending where the last location of the word is, it only has to check half of the list. Shown below is the old sorted list compared to the modified sorted list based on total words in the file:

**** As shown, it can be seen that as the file gets bigger the modified sorted list will be the way to go. It makes sense that the data will be added to the list faster because the find method will process quicker since it might know half of the time that it is either in the front half of the list or the last half of the list, figuratively speaking. Removing the words from the list works in a similar way, the modified sorted list moves faster because it processes the same way when finding the word just this time it is removing it, the graph below is the remove time vs total words:

**Conclusion:**

This project was my favorite of this semester and learning how to implement list in different ways to improve a variety of things. Seeing that skip list was the number one contender when it came to processing the larger files such as Shakespeare, King James Bible, and ALL files. When it comes to processing smaller data such as Hamlet and green Eggs and Ham the time difference was but a tenth of a millisecond. Therefore, implementing list is based on certain factors for processing the data such as, How many words? How many distinct words? Is there a lot of reoccurring words? How many words are towards the beginning of the alphabet? Questions like that are what need to be asked when implementing a list for processing data.

The removing of a word of course processed different for each of the list, but the times for all of them were all faster than the creation of the list itself. The comparisons made by the removing methods were less, but the reference changes were more. Which as studied in the report, both of those cases cause a faster in the list removal. As well the remove method only has one needed step in order to remove a word from the list, but adding it has a lot of task in order to add a word to as list. The two remove methods for the heavy-handed list acted as expected with remove 2 being the fastest since the word may be towards the front of the list and may not have to traverse the whole list in order to find the word it is looking for.

Next, the implementation of the modified sorted list was interesting to watch process. The modified sorted list of course processed the data faster but also had a lot less comparisons made in order to make that happen. It is making less comparisons when trying to see if theoretically it is in the first half of the list or the second half of the list. Also, most of the comparisons could also be due to if the last location was in the front of the list, well that doesn’t really help because it will make a pointless check since they both refer to the front of the list. It is still shown though that regardless the modified sorted list processed faster.

In conclusion, the data collected proved all the predictions made in the beginning of the report. We saw that the skip list indeed processed the data faster for all of the files compared to the other list. As well when comparing the removal times to the creation time of the list, it processed much faster and we saw that it was due to the comparisons made or the total reference changes the list had or both. Finally, we saw that the modified list was a better implementation of the old sorted list when it came to processing the data and based on the size of the text file.