**Purpose:**

The purpose of this lab experiment was to add more functionality to our previous project. Things we added include: a second list for list two (Alphabetically Sorted); called List 2a, a Skip List, a remove method to each of our previous four lists and to are two new ones, and a second remove method (that is utilized with a new list called List 3a where the method is called remove2) for list three (Front Self-Adjusting). After adding all of the new functionality, we want to compare our new (hopefully) improved List 2a to the Alphabetically Sorted list and see whether List 2a is an improvement. Also, we want to analyze the performance of our Skip List to the rest of the lists. Once this is done, we want to make a general overview of how each list is tore down and make comparisons to see if the second remove method (remove2) for List 3a is more efficient than the first (remove1) for the Front Self-Adjusting list.

**Hypothesis:**

For the comparison of the Alphabetically Sorted list and List 2a, I expect that List 2a should perform more efficiently. I believe this because I think that List 2a will should be making less comparisons (because the list is essentially cut in half, instead of having to traverse the whole list to find a matching node). For the comparison of the Skip List to the rest of our list, I believe that the Skip List will out-perform all of the other lists. I am making this assumption because the Skip List does exactly what it is in its name, it skips nodes to quickly find or insert the desired node (which means there are less comparisons to be made). The only speculations I can make about the tear down procedure is that Skip List should perform the best again. As for the rest, the self-adjusting lists should come behind the Skip List in terms of efficiency and the normal sorted and unsorted lists should perform the worst. For the comparison between the remove1 and remove2 methods for the Front Self-Adjusting list, the remove2 method should perform better because the number of comparisons should be less as more frequently deleted words transfer to the top of the list. I believe that it will become more apparent that it does as the size of the text files grow.

**Summary:**

After running all of the tests for the lists with each text file, I have obtained the following data for the elapsed time:

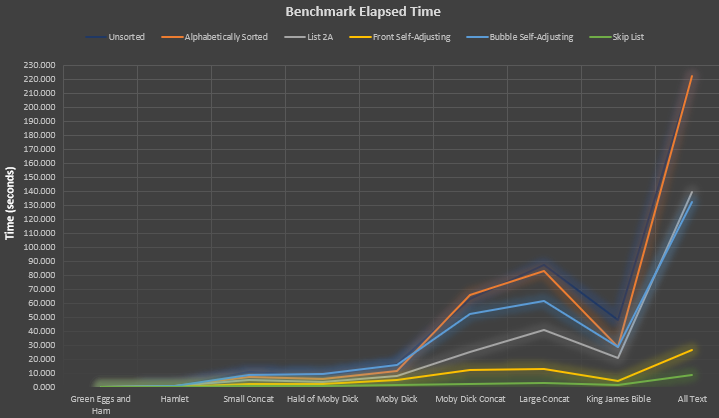


Figure 1. Elapsed Time it took each list to run all text files during the benchmarks.

As you can see from Figure 1 above, my assumption that List 2a would out-perform its predecessor, the Alphabetically Sorted list, was correct. The difference in performance becomes more obvious as the size of the file grows. Now, we need to see how many comparisons each list made to give us a better idea of what is going on. The table for number of comparisons is below:

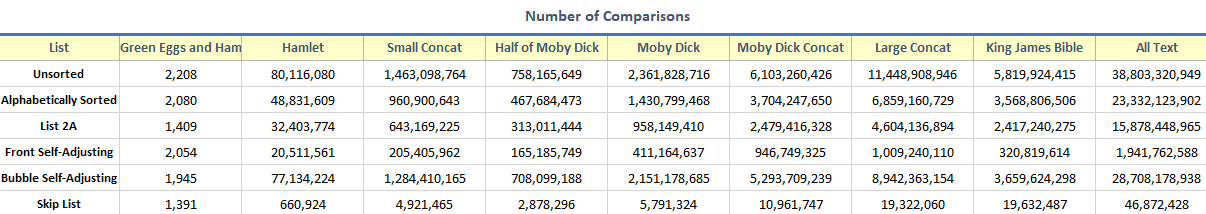


Figure 2. Number of comparisons made for each list during benchmarks.

From Figure 2, you can see that List 2a made less comparisons than the Alphabetically Sorted list in every case.

Now, you can also observe that in Figure 1, the Skip List was *substantially* more efficient at processing the text files than any other list. This statement holds true for every text file. Also, from Figure 2, you can see that the Skip List made the least number of comparisons for every text file as well. This furthers my notion from my previous lab report for Lab 2 that the time it takes one of the lists to process a text file is directly corelated to how many comparisons it needs to make. The number of reference changes that each list made is listed below:

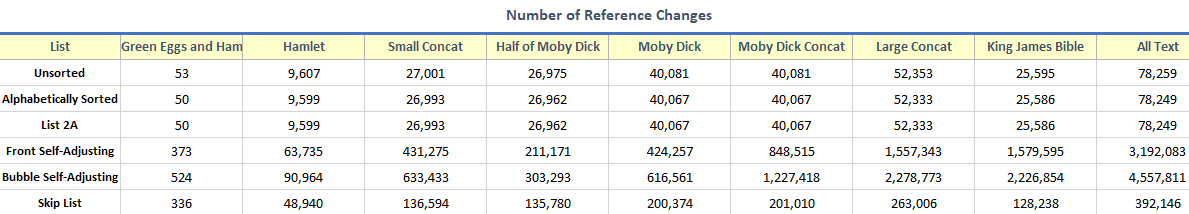


Figure 3. Number of reference changes made for each list during benchmarks

As seen from Figure 2, my hypothesis from Lab 2 still stands that number of reference changes does not affect the overall elapsed time it takes a list to process a text file.

For the tear down procedure, I did not observe anything that was out of the ordinary. My hypothesis was correct for most cases that the Skip List was the fastest, the Self-Adjusting lists were second fastest, and the normal Sorted and Unsorted lists were the slowest. You can view the data in greater detail below:

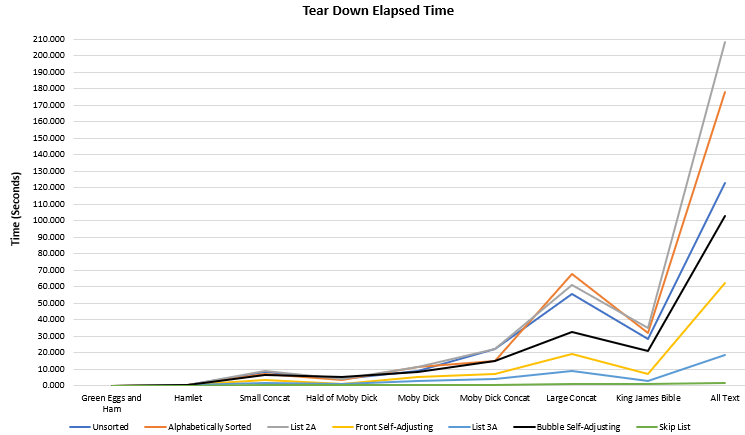


Figure 4. Elapsed time for the tear down of each list.

As seen from Figure 4, the remove2 method on List 3a out-performed the remove1 method on the Front Self-Adjusting list. This, again, had to do with the number of comparisons that were made. You can view the comparison data below:

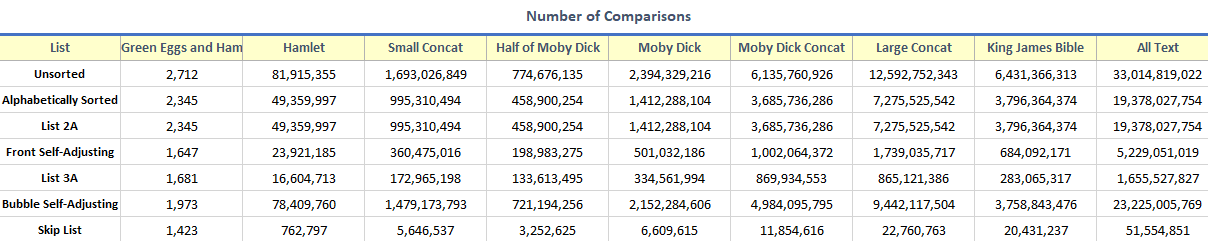


Figure 5. Number of comparisons made during tear down.

From Figure 5, you can see that the number of comparisons made were less in every case for the remove2 method in List 3a as compared to the remove1 method in the Front Self-Adjusting list. I also noticed that again, the number of reference changes did not correlate to the elapsed time it took to tear down the lists. The reference change data is below:

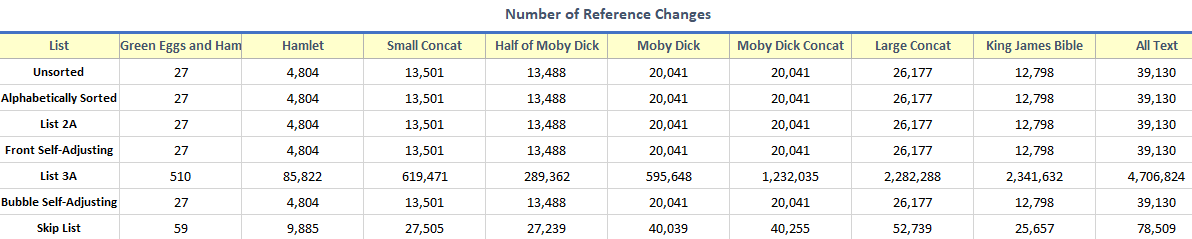


Figure 6. Number of reference changes made during tear down.

You can see from figure 6 that even though list 3a made more reference changes, it still out-performed every list except the Skip List. This data proves that the number of reference changes does not tell us anything about how efficient the particular list is.

Overall, all went as expected for this lab experiment. I knew that the Skip List would be very efficient, but I was surprised to how well it performed as compared to the other lists. This lab taught me a lot as to how more complex data structures wildly out-perform simple ones. Also, I learned that making minor changes to an already existing list can greatly improve its performance. I believe that this experience will always make me remember that no matter how efficient you think your program (or code) is, it can always be improved upon.