**Purpose:**

The purpose of this lab experiment was to add three more lists to our previous projects. The three lists are all variations of hash tables. All of the new lists we will be adding are essentially identical, except for the hash function method (hashes the string to determine which linked list the string will be added to). The first hash table I created implemented the hash function that would take the sum of the numeric values of each of the characters in the specified string and then use modulus 256 (in the form String % 256). The second hash table used the hash function that would only take the numerical value of the very first character in the string. Finally, the third hash table implemented the hash function that is very similar to the way Java’s hash function works. However, instead of producing the normal 32-bit hash, our function produces an 8-bit hash. The third hash table is similar to the first one, but there are minor differences: the hash function for the third hash table is sensitive to the order of characters in the string, where in the hash function for the first hash table is not. So, “saw” and “was” hash the same way for the first hash table, but they are hashed differently for the third because the order in which the characters appear is different for the particular words.

**Hypothesis:**

There are only a few things we need to observe for this project: how our three new lists perform as compared to our previous ones (through benchmarking the creation of the lists and then tearing them down), comparing the statistics for the number of words that appear on each of the 256 collision lists for all three of our new lists (we will be using the King James Bible text for this comparison), and comparing the min, max, average number, and standard deviation of words for each of the three new hash table lists.

I believe that all three of the hash tables that we are creating should out-perform (benchmarking and tear down) all of the other lists up to this point (even the Skip List). I think this because there should be considerably less comparisons made. Once the string is hashed, it will go to one of the 256 linked lists. Since there is so many linked lists, each one should be relatively short and easy to traverse.

For the number of words present on each of the 256 collision lists for each of the three hash tables, hash table 2 should only have linked lists on 26 of the 256 possible lists. This is because we are only taking the hash of the first character of the string and there are only 26 possible options (26 letters in the alphabet). Thus, hash table 2 should perform a little worse than the other two hash tables because its strings will not be dispersed over a wide range of linked lists. For hash tables 1 and 3, I believe they should perform about the same (slight variances depending on the input text file).

For the min number of words on a linked list, I know that hash table 2 will have the least (or at least tied for the least). I know this because there will always be 230 lists that have zero words in them. Thus, the min will always be zero for hash table 2. For the max number of words on a linked list, again hash table 2 should have the most. This is because there are only 26 possible linked lists to add strings to, while hash tables 1 and 3 have 256 possible linked lists. For the average number of words, this should be the exact same value for each hash table (because there should be the exact same number of words to place into linked lists for each hash table). For standard deviation, hash table 2 should have the highest value every time. This is because there will be a smaller data set to work with and thus slight changes in the number of words on each linked list will make the standard deviation become larger. Hash tables 1 and 3 should have a mean that is close to the value for each linked list and thus a small standard deviation.

**Summary:**

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

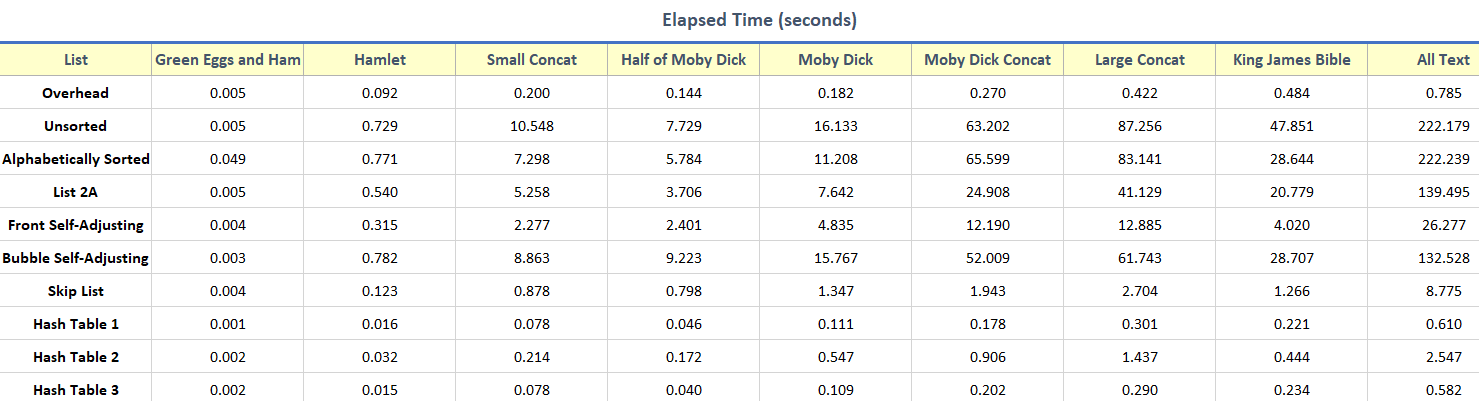


Figure 1. Elapsed Time it took each list to run all text files during the benchmarks.

As you can see from Figure 1 above, my assumption that the hash tables will out-perform all of the other lists was correct. For every test that I ran, hash tables 1 and 3 ran the benchmark in under a second (even for my All.txt file that contains 1.6 million words). As I stated in my hypothesis, hash table 2 performed slightly worse than hash tables 1 and 3 (we will examine as to why briefly). Now, we need to see how many comparisons each hash table made to give us a better idea of what is going on. The table for number of comparisons is below:

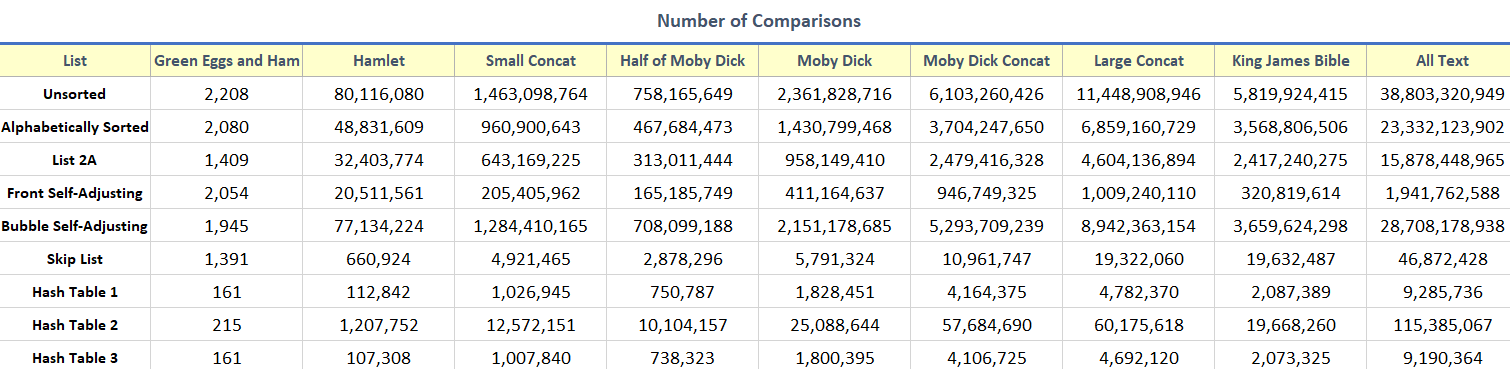


Figure 2. Number of comparisons made for each list during benchmarks.

From Figure 2, you can see that hash tables 1 and 3 made the least number of comparisons in every case (which is the main reason they are so fast). Also, hash table 2 made more comparisons than both hash table 1 and 3 in every case (why it takes slightly longer to run). This supports my notion in previous lab reports that the number of comparisons is directly related to how long it takes the particular list to parse the text file.

The number of reference changes that each list made is listed below:

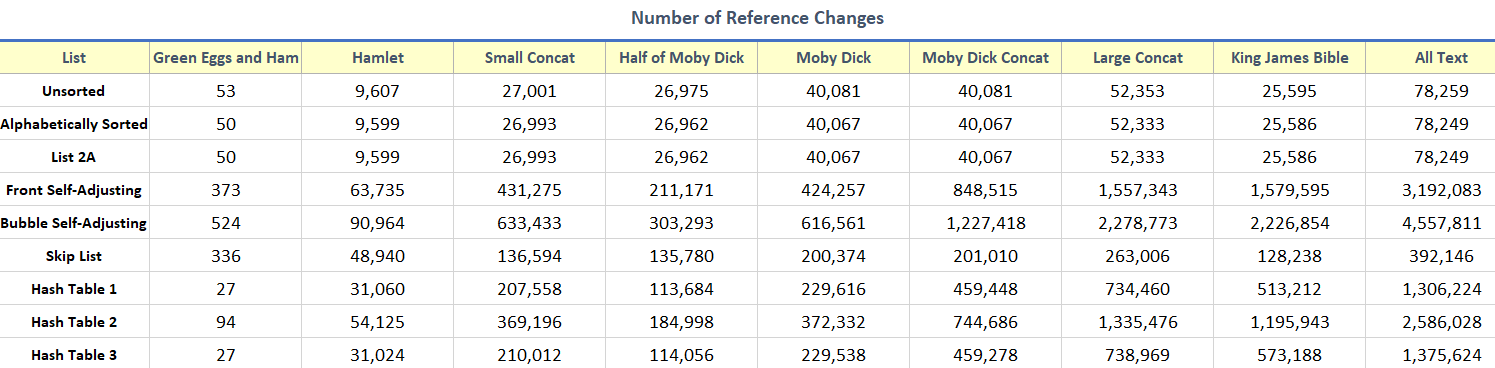


Figure 3. Number of reference changes made for each list during benchmarks

As seen from Figure 3, my hypothesis from Lab 2 & 3 still stands that number of reference changes does not affect the overall elapsed time it takes a list to process a text file. This is because for some lists (Unsorted and Alphabetically Sorted), we have a small number of reference changes and a large elapsed time, but for other lists (Skip List), we have a small number of reference changes and a small elapsed time. Thus, it can be concluded that the number of reference changes does not correlate to the elapsed time it takes for a list to parse a text file.

For the tear down procedure, I did not observe anything that was out of the ordinary. My hypothesis was correct that the hash tables still performed the best during tear down of the lists as well. The hash tables performed about the same, except hash tables 1 and 3 performed slightly better than hash table 2. You can view the data in greater detail below:

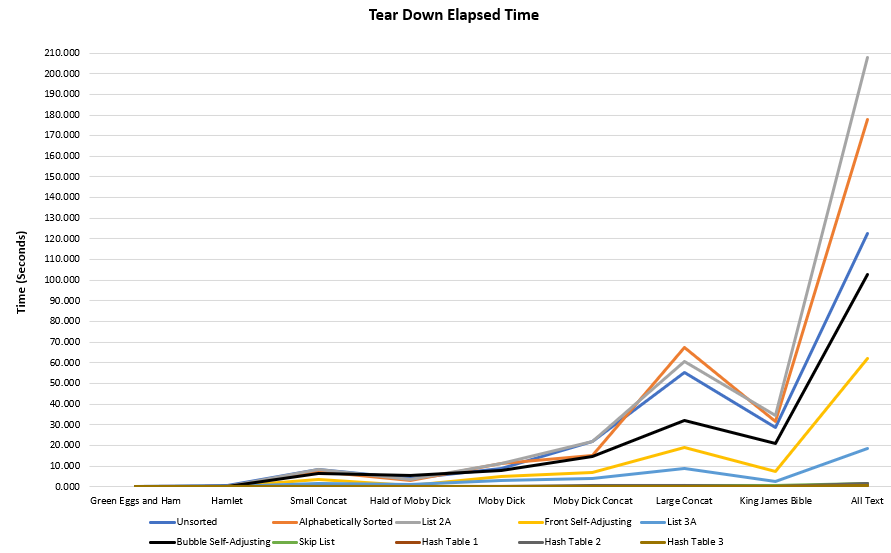


Figure 4. Elapsed time for the tear down of each list.

As seen from Figure 4, the hash tables all performed extremely well as compared to all of the other lists. It is hard to see the graph lines at the bottom (because they all overlap each other), but from the data I collected, the hash tables performed very well in regard to the elapsed time it took to tear down the lists.

For the number of words present on each of the 256 collision lists, I recorded the data for the King James Bible text file. The data I captured can be viewed below:

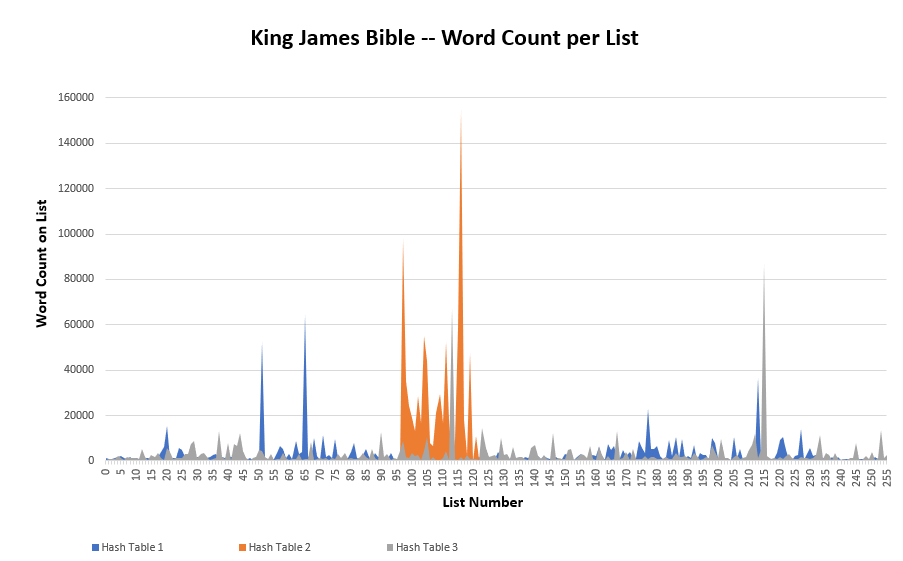


Figure . Word count per each of the 256 collision lists for King James Bible.

As seen Figure 5, my initial hypothesis was correct that hash table 2 only stores words on 26 of the 256 possible linked lists. Also, it can be seen that hash table 1 has more linked lists that are longer in length than hash table 3. I have concluded that this is because hash table 1 does not consider the order of characters in which they appear in the string. Thus, multiple words that are different, but contain the same characters will be placed into the same linked list. For example, this would happen with words like “cat” and “act”, “saw” and “was”, or “thaw” and “what”.

For comparing the min, max, average number, and standard deviation of words for the hash tables in each of the text file tests, I have created a table that contains all of the values for each of them that can be viewed in greater detail below:

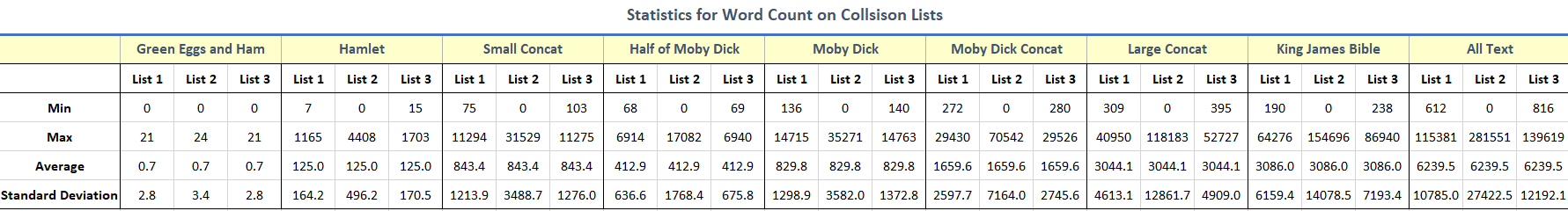


Figure . Statistics for the three hash tables tested on all of the text files.

From Figure 6, it can be seen that the min value for hash table 2 was zero in every case. This comes from what I have previously stated in my hypothesis: there will always be 230 linked lists in the hash table that have the min value of zero. The min value for hash tables 1 and 3 were similar in each case. It can also be seen that hash table 2 also contains the largest max value in every case. This also comes from my hypothesis that there are only 26 possible linked lists to store each string. Thus, the 26 linked lists will be much longer than the average linked list from hash tables 1 and 3 (which have 256 possible linked lists). For hash tables 1 and 3, the max value was usually slightly higher for hash table 3.

From my hypothesis, it was also correct that the average number of words was the same for all three hash tables on each of the text files. This is because the same number of strings were read-in for each of the hash tables. Also, the standard deviation had the highest value for hash table 2 in every test. My hypothesis was correct for this as well. In my hypothesis, I stated that hash table 2 should have the highest standard deviation because it has the smallest data set (26 linked lists of strings compared to 256 linked lists for hash tables 1 and 3). From Figure 6, the standard deviations for hash tables 1 and 3 were very similar in each case.

All went as expected for this lab experiment. I knew that the hash tables would out-perform the other lists (and they did by a large margin). I also found that hash table 2 performed slightly worse that hash tables 1 and 3. This is because hash table 2 is essentially a hash table of only 26 linked lists (thus the lists will be longer, and more comparisons will have to be made). I also found that hash tables 1 and 3 were very similar in terms of performance. There were only slight variations in their min values, max values, and standard deviations (because hash table 1 did not care about the order of characters in the string, while hash table 3 did). Overall, this lab was a good experience to create similar, very efficient lists. This project helped me realize that even the slightest of changes in your code (the hash functions for each hash table) can make a big difference in how the program runs and its optimization.