**INTRODUCTION**

This project is about reading and processing multiple text files using ten ways: an unsorted linked list, an alphabetically sorted linked list, a modified alphabetically sorted linked list, a self-adjusting list (Heavy-weight Adjust), a self-adjusting list that moves the frequent words by one position (Lightweight Adjust), a Skip List, three different hash lists and a binary search tree.

The unsorted linked list reads each word, then removes the punctuation and adds it to the list in a normal way. While reading the text files, leading and trailing punctuations are removed from each word. The sorted linked list does the same but searches the list for where the word must be placed alphabetically in the list. Once the “correct” place is found, the word is added to that specific place. This way the list would have shorter average search times because once you’ve determined where a word should be and you do not find it, there is no use in searching the rest of the list. The modified sorted list changes a few things in the search process. During the search, the new word compares with the last word that is added. If the new word is alphabetically before the last word, start the search from the beginning. Otherwise, if the new word is alphabetically after the last word, start the search from the end of the list. This small change helps the search tremendously, especially if the new word is close to the end of the list.

The next two lists are a little different. The heavy-weight adjust linked list moves the frequent words to the top every time a repeated word is read. The goal for this is to minimize search time, by predicting that a repeated word would appear more frequently. Next, Lightweight adjust the linked list. This linked list is similar to the heavy-weight linked list, with the only difference of moving the repeated word, one word up instead of moving it all the way to the top. Additionally, the skip list creates different lanes with various amounts of nodes to decrease the number of nodes the list needs to traverse through. Each time a word is added to the list, the program flips a coin until it lands a tail. The number of flips that result in a head, will be the number of layers of the new word node. This will create an unpredictable amount of layers of different nodes. The top layer has small amounts of nodes, while the bottom layer has all the words. During the search, the program starts from the top of a layer of the list, and each time a word is alphabetically before the node, it moves down a layer. When it reaches the bottom, we either get the word itself or the word alphabetically before the new word.

The three hash lists are similar, with one major difference; how they convert the words into hash. Each of the lists takes in a word, turns it to a hash which ranges from 0 to 256. The hash is then used to locate one of the 256 references. From there, a search would began from the first node of that particular lists to see it the word in repeated. If the word is repeated, increment the count and make the word the first node of the list. Otherwise, insert the node in the correct position of the list, alphabetically. As for the differences, Hash 1 creates the hash by finding the sum of the integer values of each of the characters in the word. Hash 2 creates the hash by using the integer value of the first character in the word. Hash 3 varies from Hash 1 slightly, each time a character value is added, a multiplier is added to it too. Therefore, two words with the same characters won’t have the same hash.

Lastly, Binary search tree. A binary tree is a non-linear data structure in the shape of a tree that can have up to two children for each parent. In addition to the data element, each node in a binary tree also carries a left and right reference. The root node of a tree is the node that is at the top of the hierarchy. The parent nodes are the nodes that house the sub-nodes. Each time a new word is introduced, the word would be added one of the nodes at the bottom of the tree. To determine which node it lands on, the word traverses through the whole tree starting from the root, comparing if the word is alphabetically bigger or smaller. If it’s smaller, we would move on and compare it to the left child, if it’s bigger, we would compare it to the right child instead. This would go on and on till there is no left or right pointers.

In order for analysis, data such as run time, vocabulary, total words, key comparisons, reference changes, and height are collected. Additionally, for the hash lists, the maximum, minimum, average and standard deviations of each lists are recorded. Through the analysis, we sought to know which linked lists perform better; which are more efficient. The hypothesis for this project is that the Binary Search Tree is the fastest and most efficient.

**GRAPHS**

The first 6 graphs show how vocabulary and total words correspond to the run time, key comparisons, and reference changes. Each of the 6 graphs have two versions, a zoomed in version and an extended zoomed out version; where you could see the relationship of the two axis’s from a more general, accurate perspective. The extended version also includes the hash1, hash2, hash3, and binary search tree list. In Graphs 1-6, data from the 10 mil, 20 mil and 50 mil text files are excluded. This is because it takes an enormous amount of time to run inefficient lists like List 1-4. Graph 13 includes data from the large text files but only using the skip list, hash 3 and the binary search tree.

|  |
| --- |
| **1**    “Figure 1 - Vocabulary vs Run Time”    “Figure 1a - Vocabulary vs Run Time” (Extended)  Figure 1 above shows how the run time of the different linked lists generally increases when the vocabulary is increased. However, the amount of increase is different for each list. The Light-weight Adjust linked list increases the fastest while the Heavy-weight Adjust linked list and the skip list increases gradually. This means that the Lightweight Adjust linked list is affected by vocabulary the most and runs much slower with a large vocabulary.  In figure 1a, unsorted has the most positive gradient. On the other hand, the bst list has the least gradient, close to 0. This means that the bst list is affected by vocabulary the least and runs much faster with an increased in vocabulary. This beats out the current assumption of Skip list being efficient. |

|  |
| --- |
| **2**    “Figure 2 - Vocabulary vs Key Comparisons”    “Figure 2a - Vocabulary vs Key Comparisons” (Extended)  Figure 2 above shows how the run time of the different linked lists generally increases when the vocabulary is increased. Although the amount varies for each list, the sorted, modified sorted, unsorted, and Light-weight adjusted linked lists have a similar amount of increase. The Sorted linked list increases the fastest while the Skip list increases gradually. This means that the Sorted linked list is affected by vocabulary the most and has to use much more key comparisons which reduce its efficiency.  In figure 2a, sorted has the most positive gradient. On the other hand, the bst list has the least gradient, close to 0. This means that the bst list is affected by vocabulary the least and runs more efficiently with an increased in vocabulary. This beats out the current assumption of Skip list being efficient. |

|  |
| --- |
| **3**    “Figure 3 - Vocabulary vs Reference Changes”    “Figure 3a - Vocabulary vs Reference Changes” (Extended)  Figure 3 above shows how the reference changes of two lists increase while the other two lists remain relatively stagnant as the vocabulary increases. Although it may be hard to see on the graph, the Heavy-weight Adjust list matches the Light-weight Adjust lists trajectory. On the other hand, the Unsorted list matches the Modified sorted list and Sorted list trajectory. The skip list hangs in between the two. This suggests that both of the adjusted linked lists require more reference changes as more words are inputted which makes it less efficient.  In figure 3a, self-adjusted linked list has the most positive gradient. On the other hand, the bst list has the least gradient, close to 0. This means that the bst list is affected by vocabulary the least and runs more efficiently with an increased in vocabulary. This beats out the current assumption of sorted being efficient. |

|  |
| --- |
| **4**    “Figure 4 - Total Words vs Run Time”    “Figure 4a - Total Words vs Run Time” (Extended)  Figure 4 above shows how the run time of the different linked lists generally increases when the total words are increased. The amount of increase varies between lists. The Light-weight adjust linked list has a drastic increase in run time between 350000 and 400000 words, then stagnant. Both sorted and unsorted linked lists increase gradually fast, as a smooth line graph is seen in figure 4. Unlike the other lists, the heavy-weight adjust linked list and skip list maintains an average slight increase in run time. Though, the skip list trumps all lists. This suggests that comparatively the skip list is the best when dealing with an increase in total words.  In figure 4a, self-adjusted linked list has the most positive gradient. On the other hand, the bst list has the least gradient, close to 0. This means that the bst list is affected by total words the least and runs much faster with an increased in total words. This beats out the current assumption of Skip list being efficient. |

|  |
| --- |
| **5**    “Figure 5 - Total Words vs Key Comparisons”    “Figure 5a - Total Words vs Key Comparisons” (Extended)  Figure 5 above shows how the key comparisons of the different linked lists generally increase when the total words are increased. The amount of increase varies between lists. The lightweight adjusted linked list increases rapidly and stagnates around the 110-run-time mark.  Although the sorted, unsorted, and modified sorted linked lists increase at a similar rate till around 570000. There the sorted linked list increases at an even faster rate while the other two lists stagnate. Comparatively, the heavyweight adjusted and skip list have a much lesser increase in run time when the total words are increased; with the skip list seemingly remaining flat. This suggests that comparatively, the skip list is the best when dealing with an increase in total words as higher key comparisons reduces efficiency.  In figure 5a, sorted has the most positive gradient. On the other hand, the bst list has the least gradient, close to 0. This means that the bst list is affected by total words the least and runs more efficiently with an increased in total words. This beats out the current assumption of Skip list being efficient. |

|  |
| --- |
| **6**    “Figure 6 - Total Words vs Reference Changes”    “Figure 6a - Total Words vs Reference Changes” (Extended)  Figure 6 above shows how the reference changes of two lists increase while the other two lists remain relatively stagnant as the total words increase. Similar to figure 3, it may be hard to see on the graph, the Heavy-weight Adjust list matches the Light-weight Adjust list trajectory. For both of the adjusted lists, there is a small dip before the eventual increase.  On the other hand, the Unsorted list matches the Modified sort and Sorted list trajectory. This suggests that both of the adjusted linked lists require more reference changes as more words are inputted which makes it less efficient. The unadjusted lists seem to be unaffected by the total words. The skip list hovers just over the unadjusted lists.  In figure 6a, self-adjusted linked list has the most positive gradient. On the other hand, the bst list has the least gradient, close to 0. This means that the bst list is affected by total words the least and runs more efficiently with an increased in total words. This beats out the current assumption of sorted being efficient. |

|  |
| --- |
| **7**    “Figure 7 - RunTime vs Sum of Key Comparisons and Reference Changes”    “Figure 7a - RunTime vs Sum of Key Comparisons and Reference Changes” (Extended)  Figure 7 and 7a above both show how the sum of key comparisons and reference changes of the general increase when the run time is increased. This shows that the sum and the run time are correlated. Hence, it proves that the more key comparisons or reference changes made, the slower the process of finding the word is, reducing its efficiency. |

|  |
| --- |
| **8**    “Figure 8 - Word Frequency of heavy-weight adjust linked list”  Figure 8 above shows the frequency of the first 100 words that are left after running through the Shakespeare text file using the heavy-weight adjust linked list. Unlike figure 9, there are many spikes in the graph. This shows that the heavy-weight adjust linked list achieves its goal by placing more frequent words on the top of the list, hence making it easier to search for the word. |

|  |
| --- |
| **9**    “Figure 9 - Word Frequency of light-weight adjust linked list”  Figure 9 above shows the frequency of the first 100 words that are left after running through the Shakespeare text file using the light-weight adjust linked list. There is a spike at word 61 with a frequency of 230 words. There is only one spike in the graph because not all the frequent words are sent to the top of the list like in figure 8. |

|  |
| --- |
| **10**    “Figure 10 - Average Height of the different text files  Figure 10 illustrates all the heights (highest layer) of the 16 text files. We can see that most of the dots hover around the 15 mark. This shows that with this amount of words, chances are the maximum amount of heads flipped in a row is 15. |

|  |
| --- |
| **11**    “Figure 11 - Average Height of the Skip List different text files  Figure 11 shows the average number of key comparisons of each file compared to each of the linked lists. The sorted list has the highest number of comparisons while the skip list generates the least amount. This proves that the skip lists have the most efficient search process. |

|  |
| --- |
| **12**    “Figure 12 - Average Height of Binary Search Tree of the different text files  Figure 12 illustrates all the heights of the 24 text files. We can see that most of the dots hover around the 35 mark. This shows that with the variety of words, the binary search tree still manages to maintain in a height of 35 with a few outliers. |

|  |
| --- |
| **13**    “Figure 13 - Total Words against Sum of KeyComparisons and Reference Changes  Figure 13 illustrates how efficient the Binary search tree and Skip List is in terms of key comparisons and reference changes. Although its hard to see ont he graph, the bst and skip list have almost equal amounts of key comparisons and reference changes. It is also way below then hash 3. This shows that binary search tree and skip list is superior in efficiency. |

**CONCLUSION**

An interesting observation about the reference changes from the graphs is that both self-adjusted lists have the same amount of reference changes, while both not self-adjusted lists share the same amount of reference changes.

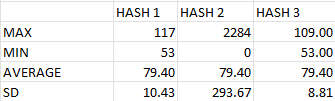


Illustration 1 (Max, min, average, and standard deviation of text file “war and peace”)

In my observation, Hash 1, Hash 2, and Hash3 behave differently due to their different ways of finding the hash. Hash 1 creates the hash by finding the sum of the integer values of each of the characters in the word. Hash 2 creates the hash by using the integer value of the first character in the word. Hash 3 varies from Hash 1 slightly, each time a character value is added, a multiplier is added to it too. Therefore, two words with the same characters won’t have the same hash. When we look at the data comparing it to the graphs, we can conclude that hash 3 runs the fastest, followed by hash 1 then hash 2. It is evident if we look at the total key comparisons and reference changes. Hash 3 has the least amount of both key comparisons and reference changes, while Hash 2 has the most. This means that overall, Hash 3 is most efficient, and Hash 1 is not.

If we look at illustration 1, we can see that hash 3 has the least amount in the maximum nodes category. Although the averages of the 3 hash lists are the same, looking at the standard deviation, we can analyze that most of hash 1 and 3’s data is closer to the mean. On the other hand hash 2’s data is significantly far away from the mean. A high standard deviation also indicates that the data is not normalized. That suggests that although overall they have the same amount of nodes, hash 2 has an inconsistent amount of nodes in each list. Some lists has alot more nodes than others; with a maximum of 2284 and a minimum of 0. This proved by the drastic amount of maximum node in hash 2.

This makes the search process inefficient. It takes a short amount of time to find the list according to the hash while it takes significantly longer to search through a list with more words. The inefficiency of hash 2 can be explained by how it creates it’s hash. Hash 2 groups all of the words by their first character, so all the a’s are in one list, while the z’s are in another. From there, it’s obvious that only 35 (26 alphabets and 9 numbers) in a list are filled with words while the rest are empty. This is inefficient as we only use 35 out of the 256 lists. Additionally, with more words having the same hash, more collisions occur, so when it traverses through each of the 35 it will have more words in them, which leads to more comparisons, longer runtimes. Hash 3 has a slight advantage over Hash 1 because the nodes are more spread out. This is because. two words with the same characters won’t have the same hash. Lesser nodes in each list means lesser nodes to traverse through, which reduces search time, producing faster run times.

Although Hash 3 is fast and efficient, the binary search tree proves itself to be the fastest. A binary search is a search in which the middle element is calculated to check whether it is smaller or larger than the element which is to be searched. The main advantage of using binary search is that it does not scan each element in the list. Instead of scanning each element, it performs the searching to the half of the list. So, the binary search takes less time to search an element as compared to a linear search. The first 100 words of the binary tree was printed and it matched the results of list 2 and list 2a.

An index of the data elements in a binary search tree is not kept. Instead, it keeps track of where each element is by using its implicit structure (the left or right of each node). This structure makes it possible to insert and remove nodes relatively rapidly. We only need to traverse half the tree, then half of half the tree, and then half of half of half the tree, as opposed to sequentially going through each element in an array until the proper one is identified.

When comparing to hash tables, here’s why the binary search tree is superior. By performing an inorder traversal of BST, we may obtain all keys in sorted order. This is not a typical Hash Table action and calls for additional work. Also, it is simple to perform range queries, identify the closest lower and greater components, and perform order statistics. These operations don't come naturally to Hash Tables, just like sorting.

Lastly, all operations with bst are ensured to complete in O(Logn) time. However, the average time for hashing operations is (1), and some specific operations may be expensive, taking O(n2), especially when the size increases. According to figure 12 the average height of the binary tree is 35, therefore the most comparisons we need to make in a binary tree is only 35. Comparing to the Hash 3, where the maximum amount of nodes we need to go through is 109.

The first 3 graphs show how the vocabulary size affects the run time, key comparisons, and reference changes. After analyzing those graphs, we can conclude that the binary has triumph the skip list and have the lowest average run time and key comparisons. The next 3 graphs show how the total words affect the run time, key comparisons, and reference changes. Unsurprisingly, the same results show, with a binary search tree having the lowest run time and key comparisons. That is incomparable to the enormous difference between the key comparisons and run time. Although, the skip list has a pyramid structure of nodes, the binary search tree is essentially a perfect skip list. Therefore through this project, we conclude and prove the initial hypothesis true, that the skip list is the fastest and most efficient.