Algorithm and Analysis – Assignment 1

Bryan Hong (s3679989) and Vincent Daniele (s3780417)

Introduction

For this experiment, three data structures were tested: 1) Hash table, 2) List and 3) Ternary Search Tree (TST). Three use case scenarios were introduced to evaluate performance of each data structure to determine the differences and efficiencies of each type of data structure for the task of word completion.

Data Generation and Experimental Setup

In this report, the dataset used was generated from a base 200,000 word-frequency txt file. This file contained already randomised words and frequencies. For Scenario 1 (Growing dictionary) and Scenario 2 (Shrinking dictionary), we had split the file into four 50,000 word-frequency txt files. For Scenario 3, to perform empirical analysis on Search and Auto-Completion, we split the 200k file into a further 20k and 10k file along with the SampleData.txt file containing 5k words. Scenario 3 used datasets of dictionary sizes: 5k, 10k, 20k, 50k and 200k.

In Scenario 1, for the four 50k datasets, we had set up a start timer and end timer for each "For" loop within our code to test the adding of words into a list/dictionary. Our experiment was set up so what we would obtain start and end times every 5000 words in a growing dictionary. We would always restart start timer for each result of every 5000 words so that the data results would reflect how long it takes for 5000 words intervals to be added in as the list/dictionary grows.

This parameter setting was also duplicated for the case of Scenario 2 but in opposite order so that we would get the start and end times every 5000 words as the list/dictionary shrinks as we delete words from it. We had also randomised the list another time before performing Scenario 2 deletion.

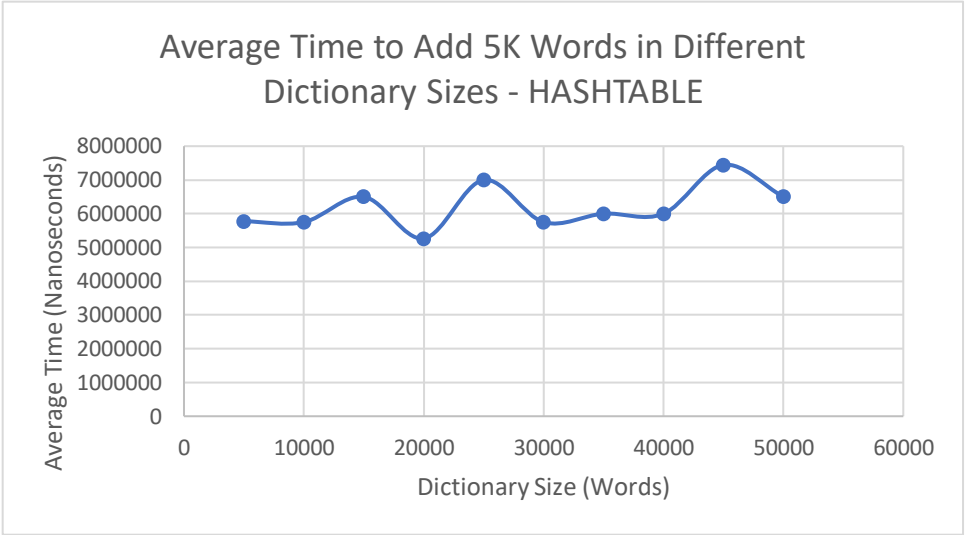
For our empirical analysis, we were aiming to obtain the averages of the four datasets which were labelled (A,B,C,D). We averaged the gained data for each level (e.g 5000, 10000…) for A,B,C,D and plotted it onto a graph. This was repeated for all data structures and in both Scenario 1 and 2.

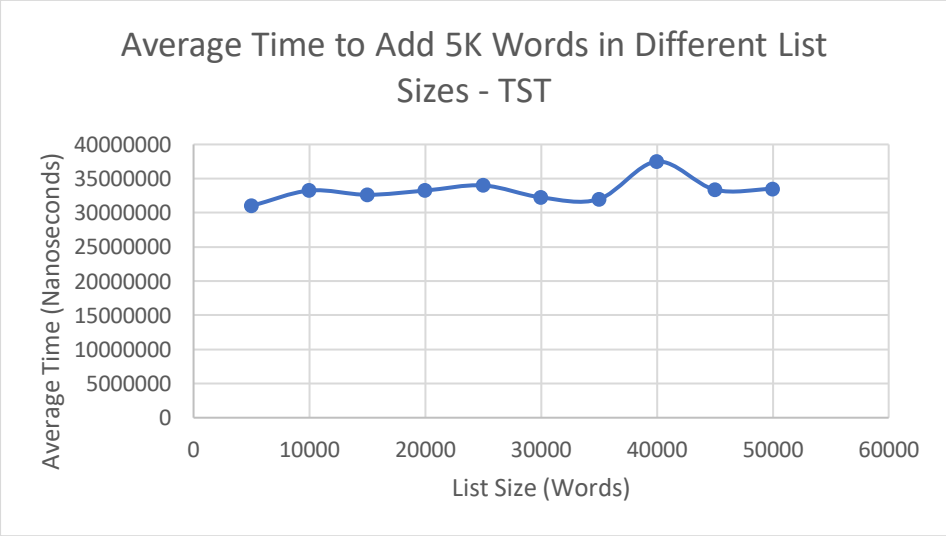
To gain results for Scenario 3, we set up the experiment such that we would get 100 random words out of the dataset that was being tested. The running times were measured after the 100 random words were already selected and the start timer would be just before the first "For" loop instance and end timer was set just after the "For" loop had completed. This was repeated for all sizes datasets (5k, 10k, 20k, 50k and 200k) multiple times and average result was plotted onto a graph for both Auto Complete and Search functions.

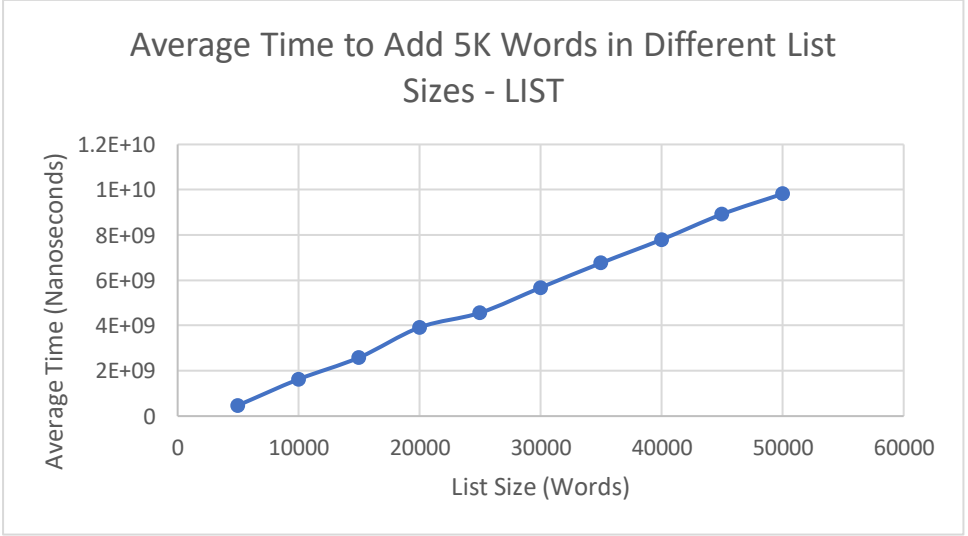
Data Observations

Scenario 1 (Growing Dictionary - ADD)

For this section, we have obtained points for every 5k words and recorded the intervals in nanoseconds to determine the effect of whether a larger or smaller size of dictionary/list would affect the time taken to add a word into the dictionary/list.

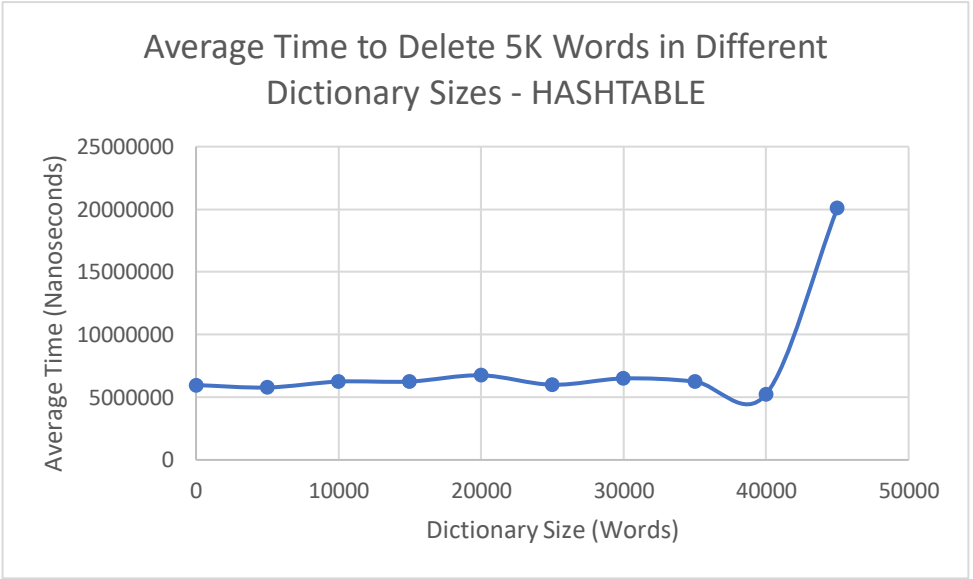


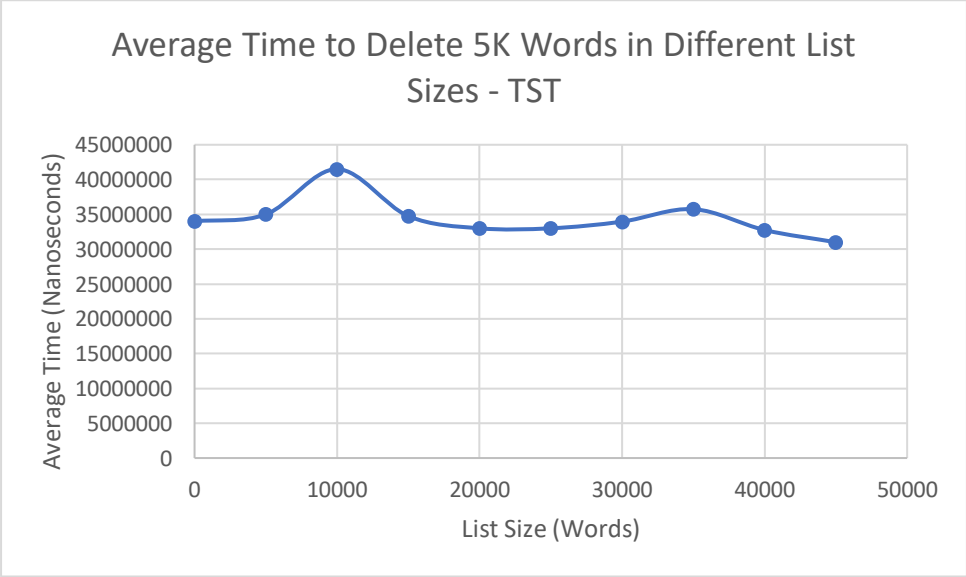


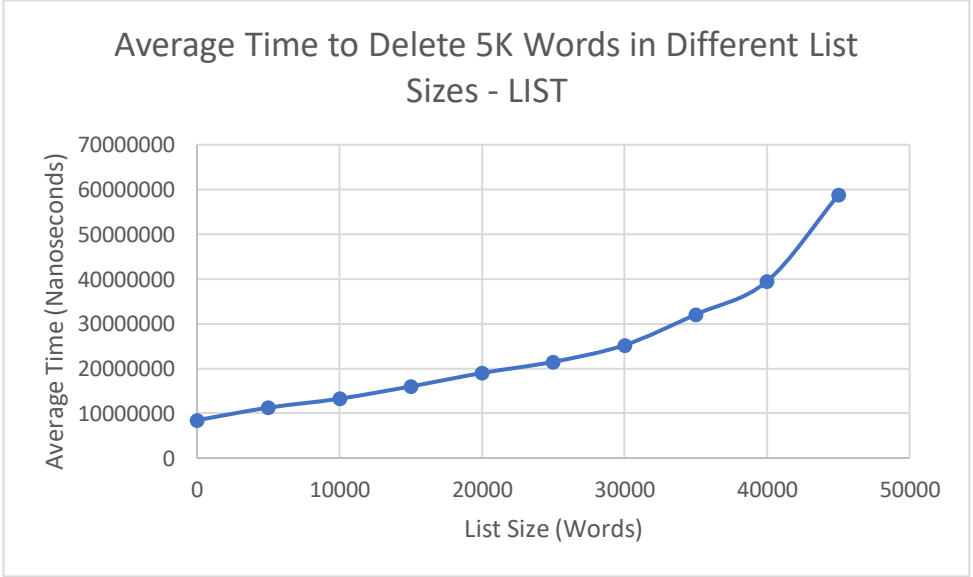


Scenario 2 (Shrinking Dictionary - DELETE)

For this section, we have obtained points for every 5k words as the words are deleted from the dictionary/list and recorded the intervals in nanoseconds to determine the effect of whether a larger or smaller size of dictionary/list would affect the time taken to add a word into the dictionary/list.



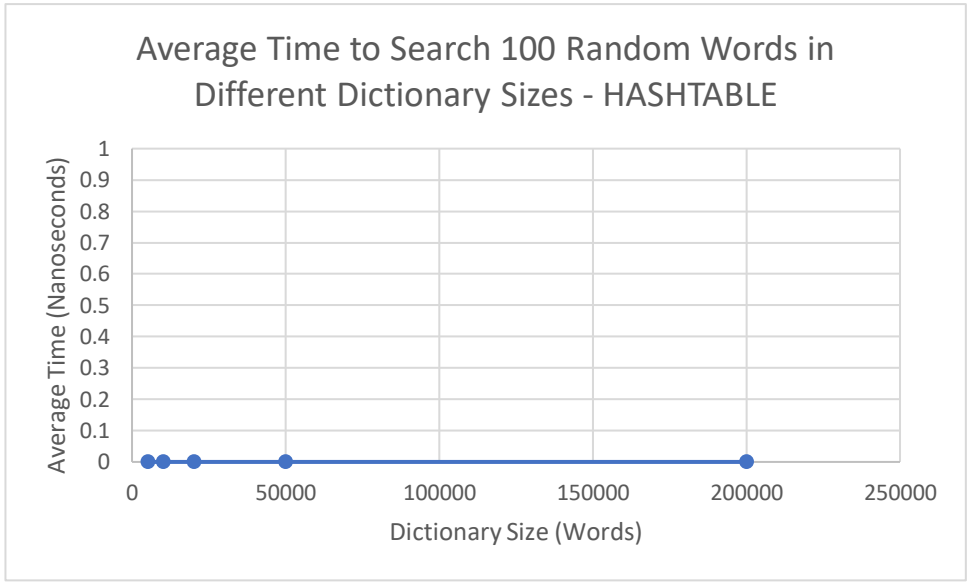


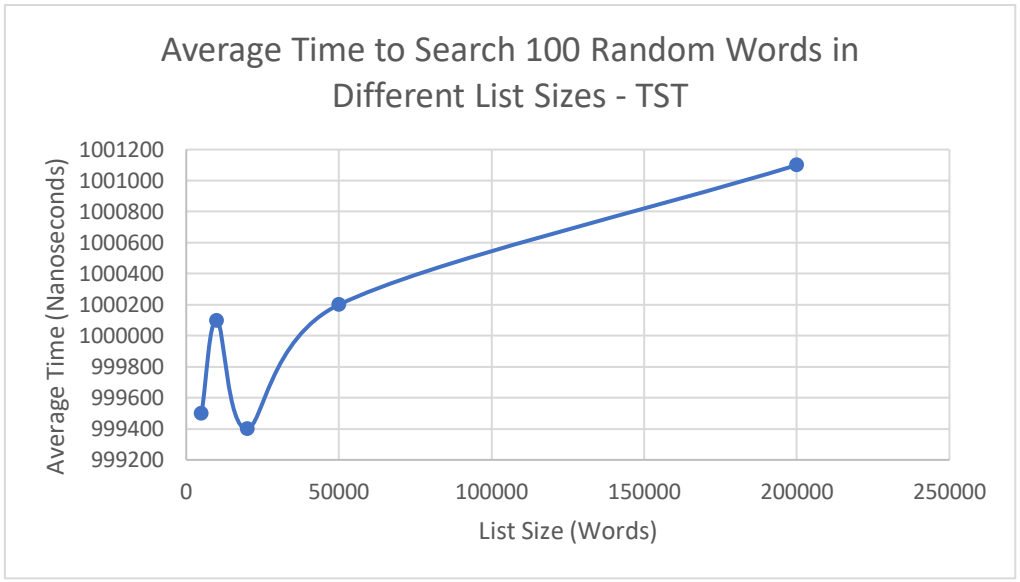


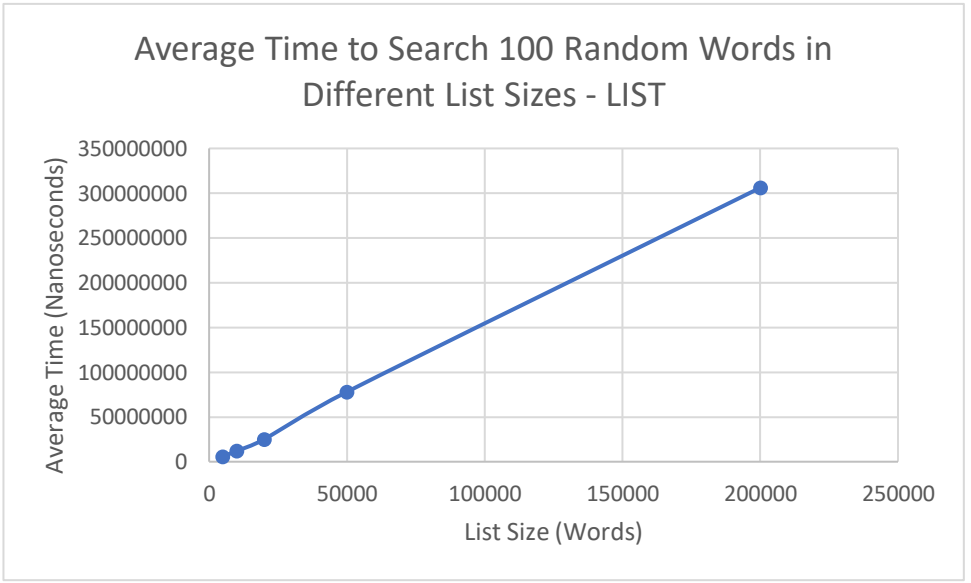
Discussion

Scenario 3 (Static Dictionary - SEARCH)

For this section, we have used 5 different dictionary/list sizes: 5k, 10k, 20k, 50k and 200k. The measured time in nanoseconds was repeated a few times for each point so that we can get the average time taken to search 100 random words in each different dictionary/list size.

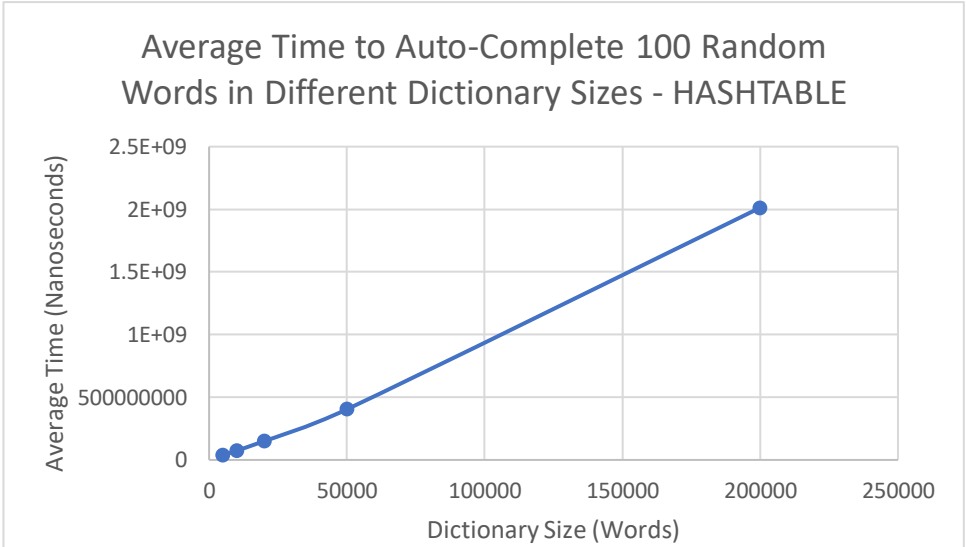


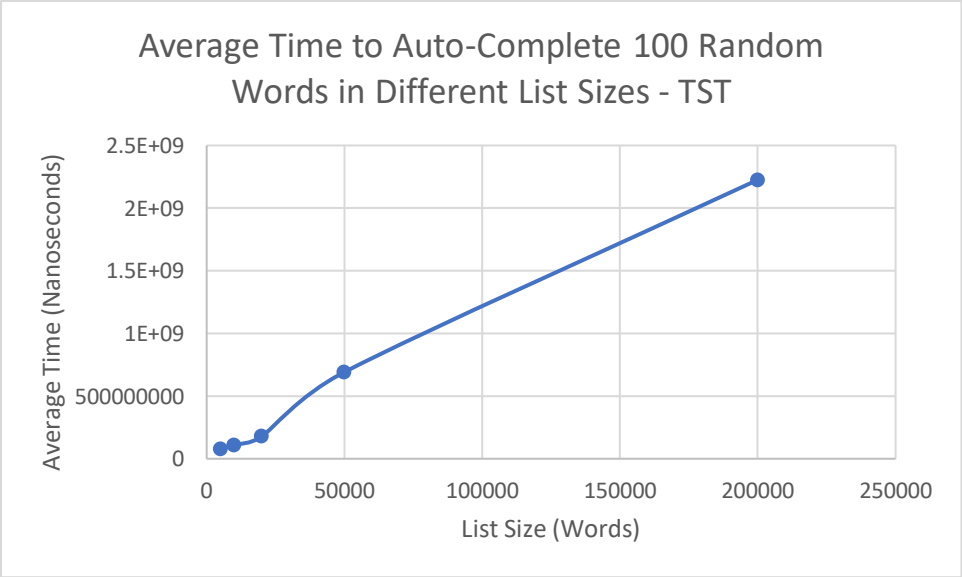


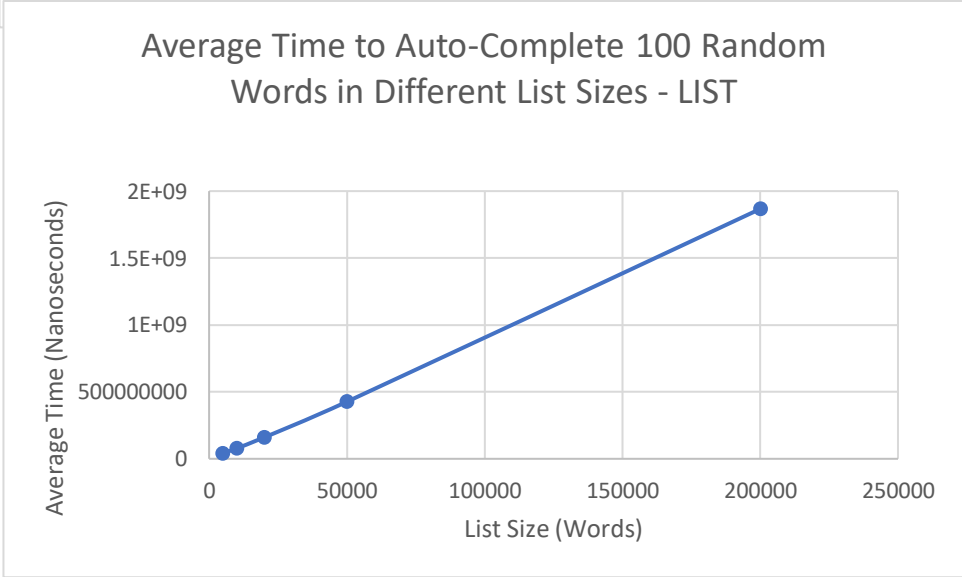


Scenario 3 (Static Dictionary – AUTO COMPLETE)

For this section, we have used 5 different dictionary/list sizes: 5k, 10k, 20k, 50k and 200k. The measured time in nanoseconds was repeated a few times for each point so that we can get the average time taken to return auto-complete list of 3 words based on the 100 random words in each different dictionary/list size.







Discussion

In this section, we will discuss the observations and compare with the theoretical time complexities expected based on the data structures tested for each scenario. According to Table 1 in Appendices, we can expect that for Hash Table, best case is O(1) and worst case is O(n). TST can be referenced as a binary search tree, and as such, theoretical best case is O(log(n)) while worst case is O(n). For List, the best case and worst case is O(n) for Search, Insertion and Deletion.

For the purposes of this discussion, we are acting on the hypothesis that there are some elements of system interference by the computer this result is tested on due to background applications. Minor differences in nanoseconds will be evaluated to be constant and this has been mitigated by using average values over multiple tests.

In our observed data, for Scenario 1, 2 and 3, we can see that the graphs produced adhere similarly to the expected theoretical behaviour. For Scenario 1, the Hash Table and TST data structures are fairly constant across all sizes and can be considered to be O(1) for Hash and arguably O(log(n)) due to the slight increase for TST on average as the data size increased, while List is linear and O(n). TST results didn't return as clear as expected for the O(log(n)) theory, but this could be due to the data size not being large enough to generate a clearer graph closer to theory.

Scenario 2 also results in a similar situations where Hash Table and TST looks like O(1) being a straight constant line, while List is O(n). It is noted that in Scenario 2, for Hash, there is an anomaly that could at 45000, this could be due to it being the worst-case O(n) due to having to search through the entire list. In general, the theory still holds in comparison to the observed data. Again, it can be observed that the theoretical O(log(n)) is not clearly upheld in the case of TST, but it can be argued that the data set size is not large enough to demonstrate the theoretical time complexity.

In Scenario 3, for Search, Hash and TST were operating on O(1) time complexity. In our data, Hash was seemingly 0 for all results, but this could be due to Hash being extremely fast and quicker than a nanosecond. TST was shown to be in a very tight range of nanoseconds. List however, was linear again and thus O(n).

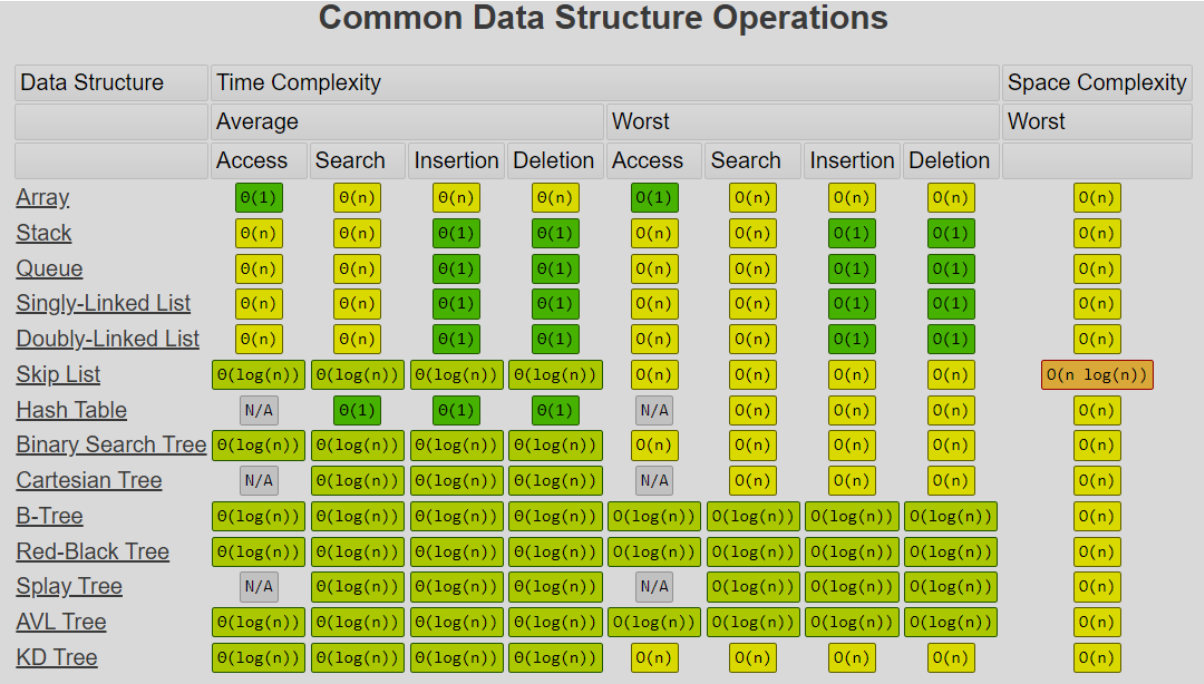
However, it is noted that for Scenario 3, in the Auto Complete case, all data structures produced linear results at O(n) time complexity. This is reasonable because for our code the word must loop through every single word in the list to find all the prefixes and then choosing the top 3 words with the most frequency based on the prefix. This would hold true to the theory of O(n) as it must loop through n times in a "for" loop for all data structures.

Conclusion

Overall, the data in this experiment shows that Hash Tables are superior in aspects of Search, Insert and Deletion with regards to average times returned and overall performance. In terms of Auto Complete every data structure leaned towards a linear model. For the majority, Hash and List followed theory, but it wasn't extremely clear that TST adhered the theory of O(log(n)) being the best case as the data size may have been too small.

Appendices

Table 1: Time Complexity, Source: https://www.bigocheatsheet.com/



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

References

***https://www.bigocheatsheet.com/* Accessed 23/04/2022**