Algorithms and Analysis Assessment 1: Word Completion

Anthony Salvatore s3920301

Simon Kaumbi s3455453

Data Generation

In all scenarios, data was generated from the provided file of 200,000 words and their associated frequencies. This data had been previously sourced from ‘kaggle.com’, having been shuffled to contain no clear pattern. The dataset was split in four, with the first three quarters reserved for adding and deleting words. The final quarter was used in ‘scenario 3’ to evaluate the search and autocomplete operations. The first three sets of fifty thousand words were each used to build dictionaries for the three different data structures examined throughout this report. This allowed for a different dataset to be used to build each of the three data structures to evaluate their corresponding add and delete operations. This is important as certain data structures may perform better on one dataset over another, so selecting three different ones can help eliminate this bias. For each data structure in ‘scenario 1’, a dictionary was grown from 0 words to 50, then to 500, 1000, 2000, 5000, 10000, 20000, 30000, 40000 and finally to the full size of the dataset, 50000. The time taken to grow the dictionary over each period was noted. This allowed for a nice evaluation from a very small dictionary to a much larger one. In ‘scenario 2’, the reverse was applied, starting from the previously built dictionary of 50000 words, each dictionary was reduced to a size of 40000, 30000, 20000, 10000, 5000, 2000, 1000, 500, 50 and finally to that of a completely empty dictionary. This provided a robust analysis, showcasing how long each delete operation took on dictionaries of differing sizes. Before deletion occurred, each dataset used to originally build the respective dictionaries was shuffled so that words were not deleted in the order they were just previously added in. Not shuffling would have provided the best case scenario for many of the delete operations, rather than a more desired average case. Only one output was noted for each test in scenario 1 and 2, as when multiple samples were drawn there was little difference between the time of each and more importantly, the shape of the resulting graph showed little change. In the interest of allowing for further testing and refinement of the analysis (as well as overall time) only one output was used rather than the average. Finally, in ‘scenario 3’, the final quarter from the original ‘kaggle’ dataset was used to build the same dictionary across the three different structures. Using the same dataset to complete the search and autocomplete operations allows for comparison across each data structure, as it can be determined how long it took each to search for a particular word or prefix. From this dataset, 100 words were selected to be searched for in each of the three dictionaries. Using these 100 words, the first two letters of each were selected to be used as a prefix for the autocomplete operation. This provides a nice method of producing an average time by performing the same operation several times. The time it took each data structure to complete a search for 100 words/prefixes was separately noted. The dataset was then reduced in size to

20000, 5000 and 1000 words, with the same search and autocomplete criteria applied to each. This allowed testing for search and autocomplete operation across datasets of differing sizes, which was then finally used to compare amongst each other in a line graph.

Scenario 1: Growing dictionary

Hashtable (Python Dict)

Adding words to a hashtable, implemented through the python dictionary has a constant time complexity of 1 in average and best case scenarios, approaching a time complexity of n in worst case scenarios. This can be seen in figure 1, where smaller corpus values result in a shorter time to add words to the data structure. A nuance of python dictionaries and hashtables in general is that they sometimes have limited space complexity. This results in collisions when adding and searching key values due to the same resulting hash function, adding up to half the size of the dictionary in time complexity, and finding an available hash key. This is overcome in hashtables by doubling the size of the data structures allocated space, so that fewer collisions are encountered. We would expect a generally constant/ linear time complexity as the size of the dictionary increases, with some dips explained by the allocated hashtable space increasing.

Using hashtables is recommended when the easy retrieval organisation of objects is desired, particularly when reading and extracting values from this data structure. It is also a good way to ensure that repeated values are handled correctly and efficiently, where the same word (resulting in the same hashed output) will be located at the same place where a new value is to be inserted.

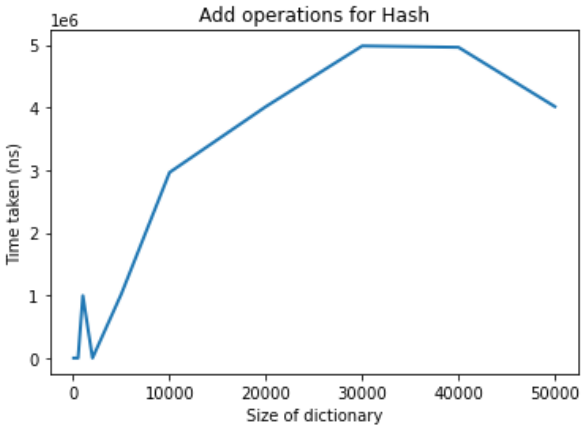


Figure 1.

List

List or array data structures have a best, average and worst case constant time complexity scenario of *1*. When adding objects to a list, the only requirement is to know the location of the end of the list, no searching or collision detection is required. The downside to adding values to a list is that repeated values are difficult to handle, and the values remain unsorted. We can see this in figure 3, time complexity grows linearly for increasing corpus sizes, due to the need to check the existence of the word before adding it the the end of the list *n * 1*.

Ternary Search Tree

Ternary search trees, a special type of Trie data structure, are an efficient way of storing word data, and are better than prefix trees in terms of space complexity, only containing three children nodes rather than 26 (for every letter of the alphabet that may follow). Adding words is as simple as traversing through the tree and inserting nodes for each letter where necessary, however, one major downside like any tree data structure is that the order of the insertions may cause an imbalanced and inefficient tree. In best and average case scenarios, the time complexity is *log(n) + k*, where k is the number of letters in the string. The worst case time complexity scenario can occur with a diverse, unusual and long letter corpus where the time complexity approaches *n*.

Insertion Comparison

Figure 3 shows the comparison in time complexity of the three data structures explored, where the list data structure is clearly the worst, at *n* time complexity.

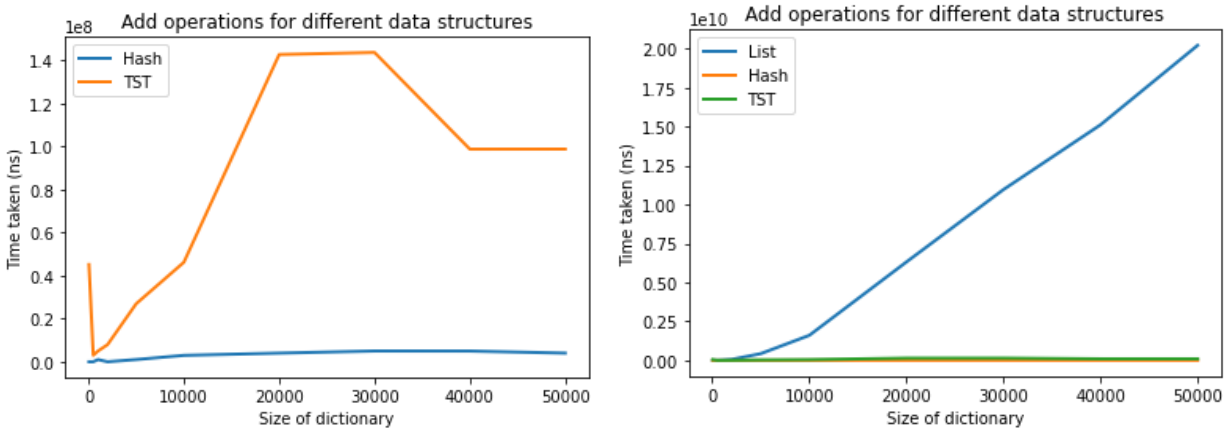


Figure 2, Figure 3.

When comparing the hashtable and ternary search tree, we can see that there is a slight difference in time complexity explained by the extra *k* described for the length of the string to be added in the tree structure. The benefit to using ternary search trees for storing corpus data is that the space complexity is better when compared to hashtables. The downside could occur when a complex corpus with unusual or lengthy words are required, resulting in a degraded and imbalanced tree.

Scenario 2: Shrinking dictionary

Hashtable (Python Dict)

Figure X shows the time complexity for deleting keys in a hashtable or dictionary data structure. We can see that delete operations have a very good time complexity, where best and average case is constant and worst case approaches *n*. This is due to the search functionality of hashtables discussed earlier, where the hash function outputs a result that allows us to find the correct key nearly all the time. The downside of hashtables is their size complexity which has to be at least double the size of the corpus in order to optimally avoid collisions.

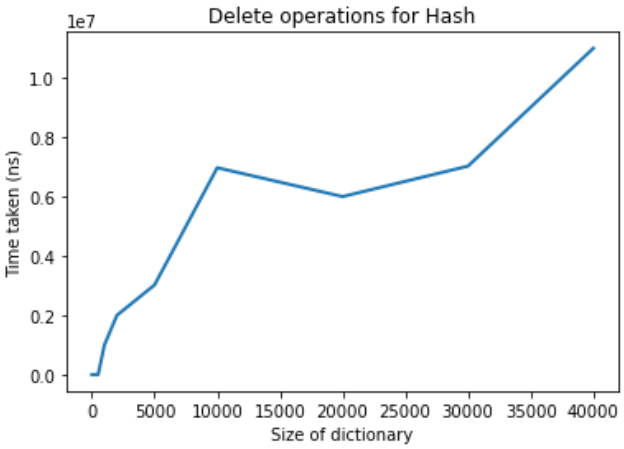


Figure 4.

List

List data structures have a delete time complexity of the size of the data structure, so increasingly large corpus' will result in increasingly longer times. The best case scenario occurs when the word to be deleted is the first word in the list, resulting in constant time complexity, however, average and worst case scenarios are *n*. If the list was sorted, this would improve the time complexity somewhat, where we could reach up to *log(n)* time complexity. If regular

lookups are required for the data structure, lists and especially unsorted lists are not recommended.

Ternary Search Tree

Ternary search trees have a delete time complexity of *log(n) + k*, where k is the length of the strings in the data structure. This can be seen in the figure below, where an increase in the corpus size doesn't affect the running time of the algorithm much. This may also be to our implementation of the code, where we were unable to achieve the removal of unique suffixes as node chains. In saying that, the space complexity should still be better than hashtables.

Deletion Comparison

As with insertion, the list data structure performed the worst with a linear time complexity. When dealing with larger corpus', it is not recommended to use lists.

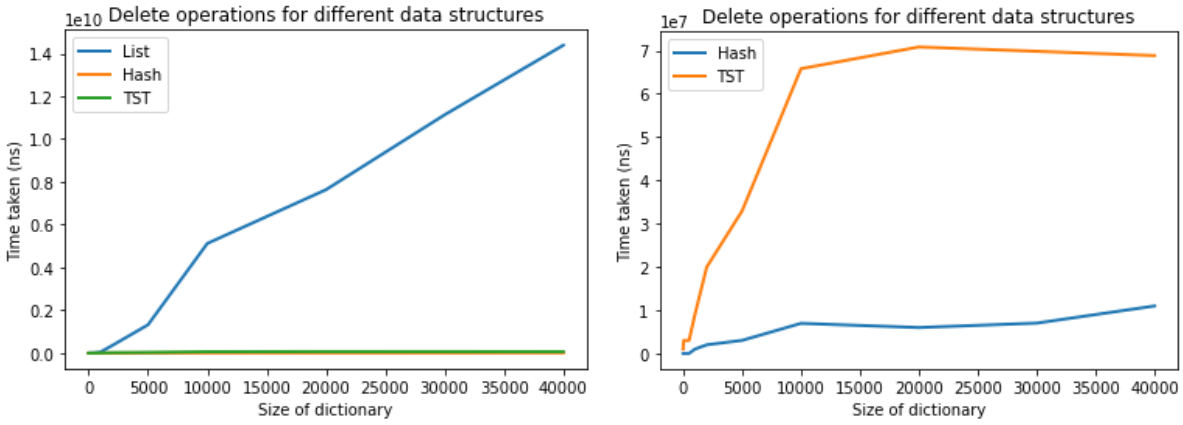
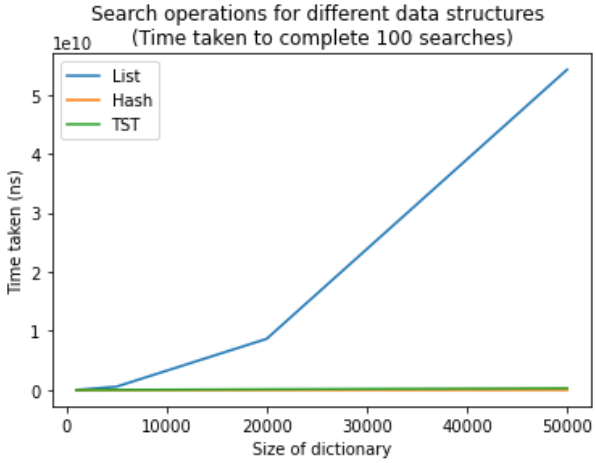


Figure 5, Figure 6.

The hashtable and ternary search tree data structure both performed exceptionally, where in smaller to medium sized corpus' the ternary search tree performed a little worse, due to the *k* metric for string length described earlier.

Scenario 3: Static dictionary

The search and autocomplete operations were assessed across dictionaries of different fixed sizes. Starting from a large dictionary of 50,000 words to a much smaller one of only 1000, the time taken to complete 100 searches and autocompletes across each of the fixed datasets for each data structure was evaluated. The search operation was analysed first, with running time comparisons across each data structure resulting in the line graph shown in figure 7. The list structure again performed the worst out of the three evaluated.



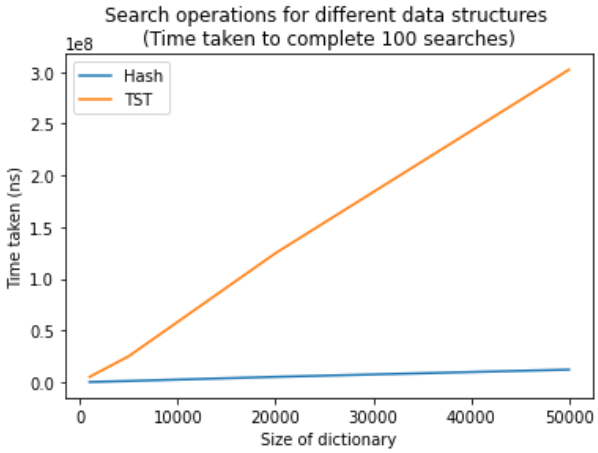


Figure 7, Figure 8.

Graphing just the hashtable and ternary search tree resulted in figure 8. It can be seen both visually and through the data generation that the hashtable performed around 10 times better than the ternary search tree, with the ternary search tree performing around 100 times better than the list. For simple search operations, the list is clearly superior when concerns are strictly over time to complete an operation.

The autocomplete function of each was then assessed, using the same fixed size datasets as the search. The first two letters of 100 randomly selected words found within each of the selected datasets were used as evaluation. Comparing the running times across the three resulted in figure 9. While all three structures exhibited linear time complexity, due to the structure of the ternary search tree, it was able to outperform both the list and the hashtable. If autocompletions are of primary concern, the ternary search tree should be implemented as the data structure of choice.

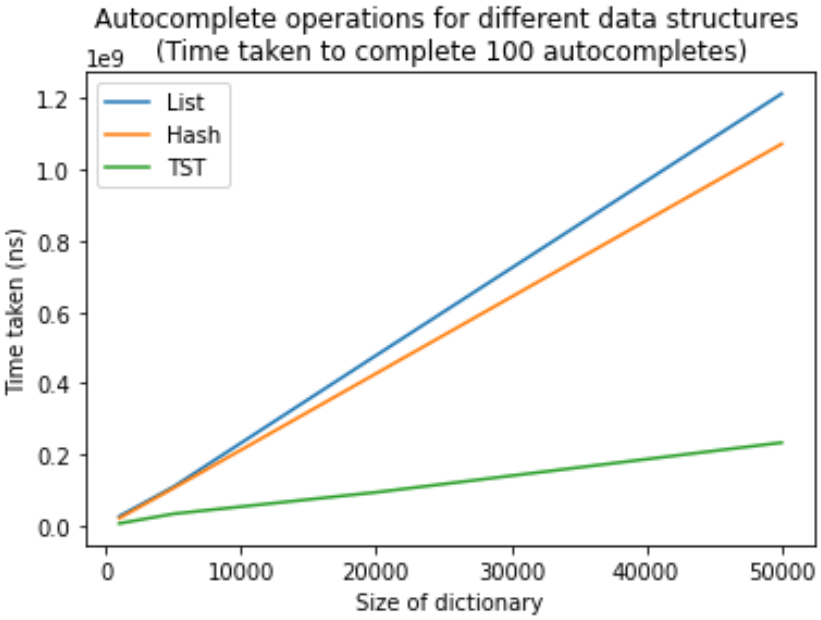

Autocomplete operations for different data structures
(Time taken to complete 100 autocompletes)
1e9
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Time taken (ns)
List
Hash
TST
0
10000
20000
30000
40000
50000
Size of dictionary

Figure 9.