Algorithms and Analysis

Assignment 1: Algorithm Implementation and Empirical Analysis

By Lachlan Van Der Klift & Matthew Yamen

Abstract

In this paper, three distinct approaches are tested and evaluated to determine which approach provides the fastest runtime for each of the four distinct algorithms. Nearing the conclusion, a recommendation is made of the best approach of the three. Each approach is dependent on its underlying data structure (if any), and as a result, its efficiency is also dependent on the efficiency of the underlying data structure.

As of the current technological epoch, theoretical time complexity reigns as the dominant methodology of analysing and evaluating the runtime of an algorithm, wherein decisions are formulated, typically, solely on the determined order of growth of said algorithm. However, there are certain pitfalls of theoretical time complexity, such as its negligence of constants in the total amount of operations performed calculation, hardware specifications and processor operation speed. As a result, its effectiveness may diminish, in regards to determining which algorithm is the most efficient.

Therefore, in this paper, we will implicitly juxtapose, while simultaneously combining the two efficiency methodologies; theoretical time complexity and empirical analysis, with the hope of obtaining a more complete, and accurate picture of each of the approaches' runtimes. Thus, our recommendation can be more accurately substantiated.

Experimental Setup

Our approach, utilising the list, *[50, 500, 1k, 2k, 5k, 10k, 50k, 100k] (k = thousand)*, as input sizes, is as follows. A **new** dictionary is created, and utilising the *build_dictionary()* method of that dictionary, we pass in 50 elements (so the dictionary now contains 50 elements), and we then time n **add** [add_word_frequency()] operations (delete is not dissimilar, but the input and their size is reversed, so going from 100k to 0), for example, then take the average of all of those n operations, and then save that averaged time. So now, the dictionary will contain 50 + n elements all together. Next, we create a **second**

new dictionary (the first will be eligible for garbage collection), and this time, we fill it with 500 elements, again, utilising thebuild_dictionary()method of that dictionary. Then, we perform n **add** operations (could also be delete, but would require the input to be reversed, as specified above), for example, once more and time each one, then we take the average of all the n operations' times, and save that averaged time. So, now, the second new dictionary will contain 500 + n elements. We then repeat this process until we reach the last of our input sizes, 100k (at this iteration, the **8th new dictionary** will contain 100k + n elements), and then process further to produce some numerical statistics, graphical representations, etc.

As you can see, we are not sticking with just one dictionary, growing that overtime, and then timing the ith operation (i being the index of the loop from 0->100k) at a certain interval (i.e., 50, 500, 1k, ..., 100k); I believe this is the approach explained in FAQ Q7 and in the assignment spec. Instead, we are creating **8 new dictionaries**, (so, it's not technically one growing dictionary, but instead, it is **many growing dictionaries**) where each one is fed the input beforehand via the build_dictionary()method, and then timing our chosen operation/command (add or delete) by running it n times, averaging all those n runs at a particular input size and then saving that time, then proceeding onto the next input size (i.e., if we are at 1k, then we will transition to 2k, and repeat the filling and timing process described above).

The reason we interpreted it this way is because the alternative proved to be unreasonably slow while utilising our benchmarking methodology, which prioritises accuracy over speed of benchmarking completion (to a reasonable degree obviously). We have ensured that our code for Task A is as optimised as possible, so that every algorithm has the lowest possible theoretical time complexity, plus further optimisations to minimise the degree and number of constants (as the theoretical complexity can only tell us so much, due to constants being ignored in the equivalence classes). We believe that the problem is due to the add and delete methods containing the additional overhead of having to search beforehand and a few other operations to satisfy the criteria of Task A. However, the build_dictionary()method does not contain this additional overhead, and it can process word frequency objects in bulk (the additional overhead is also unnecessary for the un-timed operations, since all we want to do is just grow the dictionary to a certain point, then time the entire algorithm), as opposed to only being able to handle singular units at a time (add method). Consequently, the former performs much faster than the latter.

Therefore, we have managed to achieve a higher degree of accuracy in our analysis due to utilising our approach over the FAQ Q7 one, because we can now drastically increase the amount of repetitions, and repeat the entire process 10 times and take yet another average (among other accuracy-driven optimisations). We chose this approach over the FAQ Q7 one due to the computational limitations of our machines and again, respecting the reality that we cannot wait for hours on end to produce one set of results and still uphold the maxim of accuracy being held to the highest regard.

Data Generation

All of the data used in the experiment is taken from the file provided named **‘sampleData200k.txt’** in order to ensure data collection is unbiased.

Firstly, for measuring the runtime of each method invocation, we opted for the inbuilt Python ***timeit*** library. The rationale behind this decision is as follows:

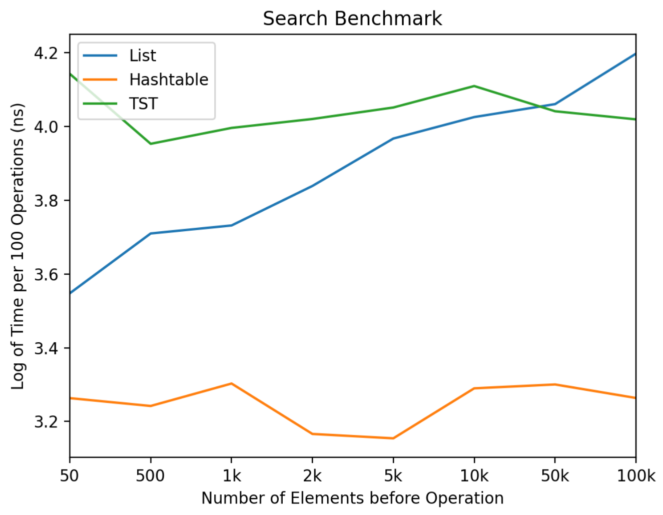
- It ensures that most of the compiler optimisations are mitigated, so as not to tamper with the actual accuracy of the algorithm.
- It disables Garbage Collection during the invocation, ensuring that a stop-the-world event (i.e., a collection) is not called in the midst of the invocation. If such an event occurred, the resulting time would be very different to the other results (likely far slower).
- It ensures disk flushing and other OS scheduling related activities do not add bias to the results.
- It ensures numerous under-the-hood operations are taken into account, ensuring that the results are actually a product of the algorithm’s runtime, and not a mixture of unintended, non-related operation/tasks plus the algorithm’s actual runtime.

Secondly, to initialise the data used in our dictionaries for all 3 scenarios for each data structure, we ensured that each dictionary **always** contained the data of the dictionaries that were below it in size, and that for **all 3** data structures, the dictionaries of corresponding sizes shared the **exact same** data. This pattern is continued until all dictionaries are satisfied (the data in each sized dictionary is a subset of all the dictionaries with sizes above it), where the total data for each dictionary is **always equal to the size**. The data for each individual dictionary will be the same throughout every run, as making it different would lead to different results and lead to a lack of accuracy and reliability in the methodology.

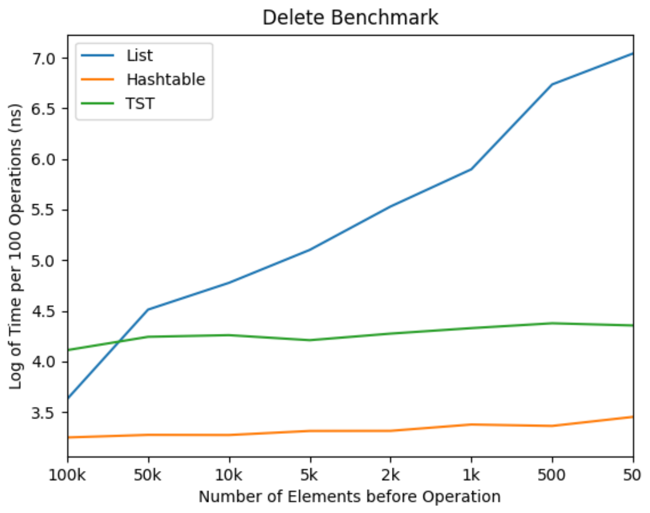
Data Structure	**Size 50 Unique Data**	**Size 500 Unique Data**	**Size 1000 Unique Data**
List Hashtable TST	Has data ‘x’ num(x) = 50	Has data ‘x’, ‘y’ num(y) = 500-num(x), x != y	Has data ‘x’, ‘y’, ‘z’ num(z) = 1000-num(y)-num(x), z != y != x

FInally, the sizes of our datasets are defined by a list of **input sizes** (as mentioned earlier), which we set as **[50, 500, 1k, 2k, 5k, 10k, 50k, 100k]**. These sizes ensure the graphed data is spread out across a variety of different sizes, enabling a better grasp at how efficient the algorithms are at certain sizes. Our **'n'** value (the amount of times each algorithm is run) has been set to **100**. If the add algorithm is run, all possible word frequency objects are taken from a separate file, **'input_adds'**, which contains objects not present in any of the other dictionaries.Whereas, if the autocomplete algorithm is run, a word is selected and a random number of letters from the start from 1-5 are taken. To ensure the results are realistic, it is entirely possible that the objects in these operations could be repeated. There may be occasional cases, where, for example, the add is already adding an object that has just been added to the dictionary. To account for this potential outlier, as well as any other overhead that might occur in one run, we go through and store the **average times** for the 100 operations done, per sized dictionary on **'k'** entirely separate runs, and **average all the average times** per sized dictionary for the **final average time** per sized dictionary per run. In our case, we have made **'k' 10**. This way, all our results are ensured to be consistent, unbiased, and as accurate as possible in terms of our own implementation. As a consequence of said approach, the results will not differ much between program runs. This provides us with reassurance that our results, collected from one run, are representative of the population of results.

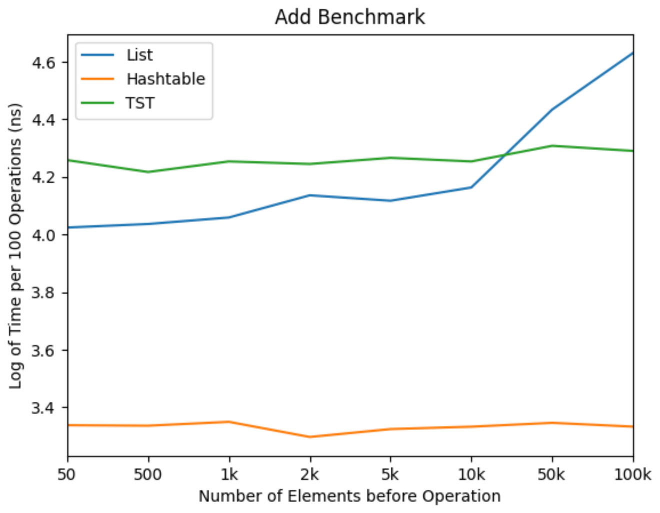
Empirical Analysis



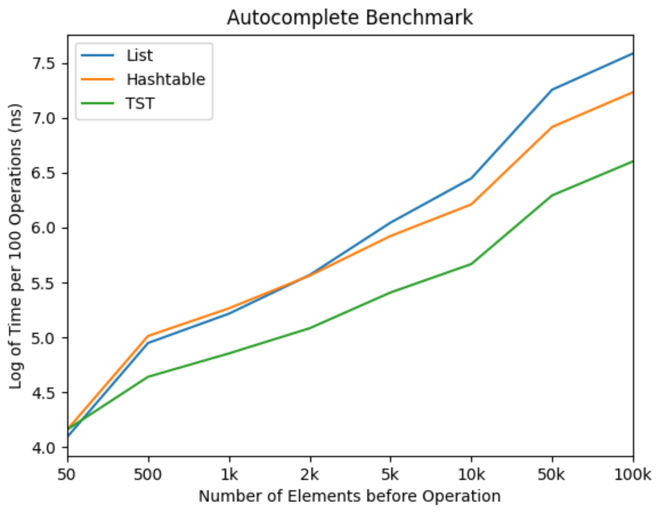
As depicted in this graph, the superior search algorithm implementation is the hashtable's, backed by its average case time complexity O(1). In addition, the list and TST's performance is backed by their average case time complexities of O(log(n)). The difference between the list and tst runtimes is the degree of constants and the cost of each operation. The TST's recursive base is far more expensive than a non-recursive binary search. That being said, the list's higher degree constants eventually overcome the TST's runtime at around 50k elements.



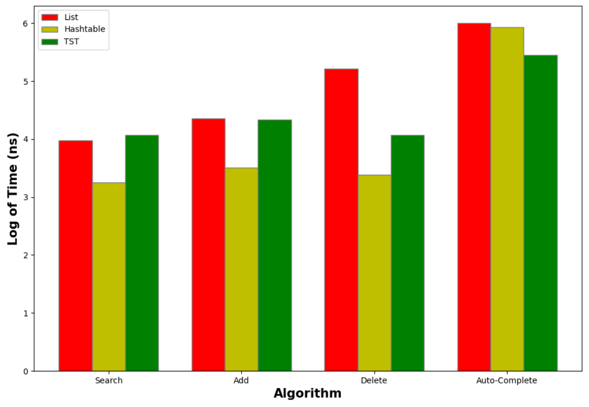
Once again, the hashtable's delete algorithm is the superior implementation. This is also backed by its average case time complexity O(1), compared to the list's O(n) and the TST's O(log(n)). The time complexities support the graph's results, as the list appears to be linearly increasing, the hashtable appears to be constant, whilst the TST appears to be only slightly increasing. The list's overhead stems from the underlying list's remove method having to perform two consecutive linear scans.



As per usual, the hashtable is the superior implementation for the add algorithm. This is also backed by its average case time complexity of O(1), with the list's being O(n) and the TST's being O(log(n) + k). The hashtable's growth rate is constant, meaning it is independent of the input size. Whereas the TST's growth rate is quite shy, but it is visible, and the list is very clearly linearly driven.



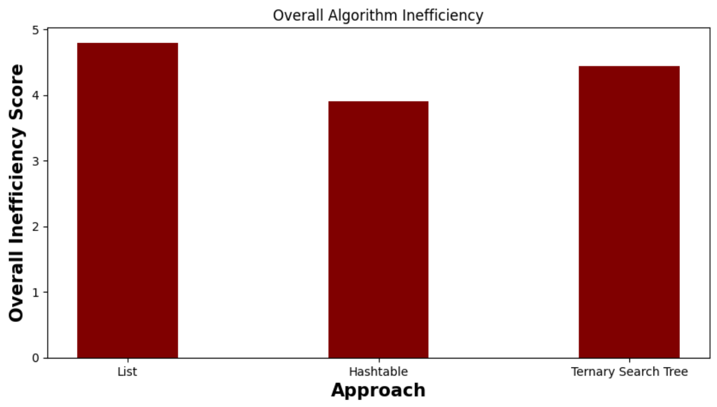
Contrary to the popular trend of hashtable being the superior implementation, the TST appears to be the most efficient implementation in this graph. This is also backed up by the time complexities' worst case and best case (average case not calculated). The TST's worst case for the autocomplete algorithm is better than the other two implementation's best case (O(n) vs O(n * k)).



The graph on the left is a side-by-side comparison of each approach and its algorithm's runtime. As we can see, the hashtable is the fastest for three out of the four algorithms.

Summary

To summarise, we have devised what we call the "Overall Algorithm Inefficiency" graph, which arbitrarily assigns each data structure a value on how "inefficient" it is based on previous graphical recordings. The higher the score, the more inefficient the data structure is in comparison to its peers. The approach with the lowest score is the best choice for a general purpose use-case, where the input sizes are unknown or trivial prior. The hashtable is the most efficient algorithm in terms of a general purpose use-case, and thus, this would be our recommendation, as in our use-case, we do not know the input size beforehand. In a similar vein, if you are interested in an approach that specialises in one particular algorithm, then the hashtable would be an ideal choice for the add, search and delete algorithms. Alternatively, if autocompletion was of particular importance, then the TST would be the ideal candidate.



That said, we are aware that our implementation certainly isn't flawless. As noticeable in the empirical analysis, the ternary search tree was the best data structure for autocompletion, which is to be expected. Though, had we had more time to re-write its algorithms using an iterative approach over a recursive one, then it is quite probable that the ternary search tree would've been on par with the hashtable. When possible, the list data structure should be avoided, as even with considerable optimisations it was still outperformed by both the hashtable and the ternary search tree.

Theoretical Time Complexities

Approach	Algorithm	Worst Case	Average Case	Best Case
List	Search	Θ(log(n))	O(log(n))	Θ(1)
	Add	Θ(n)	O(n)	Θ(log(n))
	Delete	Θ(n)	O(n)	Θ(n)
	Autocomplete [Note 4, 5]	Θ(n * k)	N/A	Θ(n * k)
Hashtable	Search [Note 6]	O(n)	O(1)	Θ(1)
	Add	O(n)	O(1)	Θ(1)
	Delete	O(n)	O(1)	Θ(1)
	Autocomplete [Note 4]	Θ(n * k)	N/A	Θ(n * k)
TST	Search	Θ(n)	O(log(n))	Θ(k)
	Add [Note 4]	O(n + k), Ω(n)	O(log(n) + k)	Θ(k)
	Delete	Θ(n)	O(log(n))	Θ(k)
	Autocomplete [Note 5]	O(n)	N/A	O(k)

NOTE #1: All time complexities that incorporate a *log(n)* are of base 2 (i.e., $log_2(n)$).

NOTE #2: All time complexities below were calculated by taking into account the underlying data structure's time complexity in the **CPython** Python implementation. Hence, the underlying time complexities of some of the in-built methods may vary between implementations.

NOTE #3: The **input size (n)** of each time complexity is the **number of elements** in that dictionary. The cases account for the overall input to achieve said case, while the upper, lower and tight bounds take into consideration the position of the element being processed within the dictionary (as this is not something that can be manipulated by the input alone, but nonetheless, can occur). The input size (n) is not to be confused with the length (i.e., number of characters or digits) of the *word* string or the *frequency* integer; this, however, does affect the overall runtime of the algorithm, but it is not relevant when discussing the theoretical time complexity, as it is considered an operation among the T(n) operations performed and is NOT the input size.

NOTE #4: The letter, k, denotes the length of the word argument (it is part of the basic operation's functionality).

NOTE #5: An N/A means we have not calculated the average case time complexity due to running out of time. It would be better to leave it blank rather than to make an educated guess with the potentiality of error (prevents our analysis from relying on faulty results, and thus, becoming incorrect as well).

NOTE #6: Regardless if open addressing for collisions or closed addressing is utilised, the order of growth is still O(n). However, this relies on the assumption that the Python hash table implementation does not utilise a tree data structure, making searching become O(log(n)). Unfortunately, due to limited time, this cannot be verified, hence, the safer choice of O(n) was used (while it is not as accurate, it is still correct).