Student Name: Jessica Bath / Kelvin Young
Student ID: s3899733, s3718060

This report is an empirical study on the executional efficiency of different data types and data structures, within the context of an autocorrect system. The three data types and structures featured in this report are: Python list, Python dictionary (hashtable), and Ternary search tree (Python abstraction).

Experimental setup

Tests will be completed on each function for each of the data structures.
To record execution time for differing dictionary sizes, the following code is used:

```
if i % (len(words_frequencies) / x) == 0:
     Run timed command
Else:
     Run untimed command
```

This code makes it so that for any given size dictionary, we can run x amount of tests, evenly spaced from each other.
For each of the given scenarios, three successive tests were run for each data type; timing was recorded for each using the `time.time_ns()` method from the python time module, to allow for the accurate measurement of the runtime of the relevant algorithm in nanoseconds.

Design of Evaluation

Scenario 1. – Growing Dictionary

- Measurements are taken by sequentially building the dictionary using the add function, and timing the add function intervallically (or at equidistance); in this case for every 10000 words.
- The dataset used contains no duplicate values, however the algorithms themselves must still check for duplicate values where needed for the sake of completion.
    - Therefore, the algorithm is oblivious to the fact that the dataset contains no duplicates.
- Measurements are written to a .csv file three times; the averages are then used for evaluation below.

Scenario 2. – Shrinking Dictionary

- Measurements are taken by sequentially destroying the dictionary using the delete function, and timing the delete function intervallically (or at equidistance); in this case for every 10000 words.

- The word chosen for deletion at each interval is not randomised, instead the word at the end of the list is popped off.
    - This is done in order to avoid resizing the list while we are iterating through it.
- Measurements are written to a .csv file three times; the averages are then used for evaluation below.

Scenario 3. – Static Dictionary

- Measurements of the search and autocorrect functions are taken on dictionaries of various sizes.
- The search parameter is randomly chosen from the target dictionary itself to prevent discernible bias across different tests (i.e. a search on test dictionary size 10000 randomly selects a search parameter from the 10000 words within that specific dictionary).
    - This allows us to achieve a better estimate of the average runtime, in comparison to searching the same word in each iteration.
- For autocorrect testing, the prefix word is determined by selecting a random letter from the alphabet. The prefix is only one character long *(see line 75 in ternarysearchtree_dictionary.py for implementation).*
- This assumes a similarity to real world autocorrect implementations in which predictions immediately begin generating once even a single letter is entered.
- We used a range of different sized dictionaries starting from an empty dictionary to a dictionary of size 190000. As with the previous scenarios the measurements are written to a .csv file three times and averaged out for analysis.

Analysis of Results

Scenario 1. – Growing Dictionary

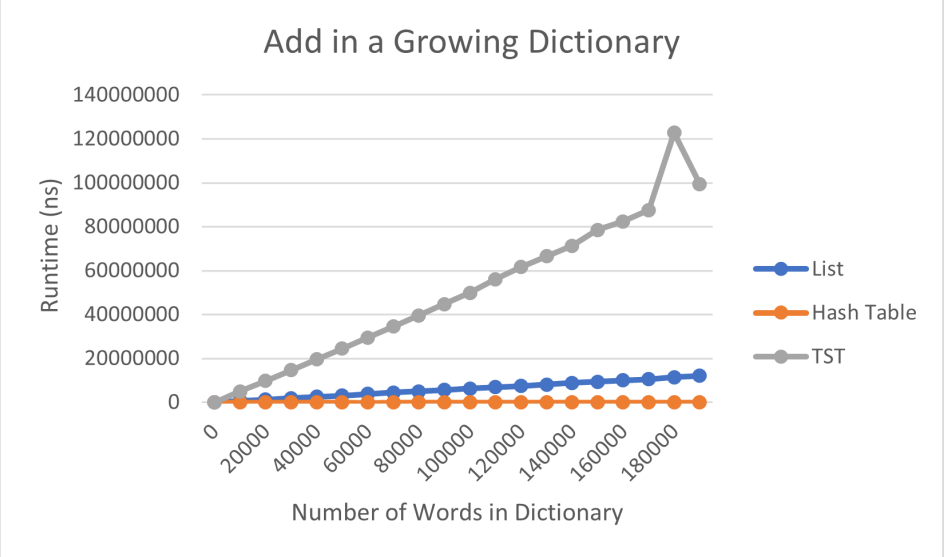

Figure 1. Runtime of the add function for List, Hash Table and TST implementations with a growing dictionary.

Above we can see that the ternary search tree and list are evidently both O(n); this is because they are both required to check if the word to be added already exists in the dictionary. Naturally, the hash table implementation had the best runtime in the growing dictionary scenario; unlike for the other two there is no need to check if the value to be added is a duplicate. Therefore a hashtable has a runtime complexity of O(1). Though there is some variation in measured values the overall trend of the data acquired supports the theoretical time complexities of all operations. The distinct runtime measured in the TST data when the dictionary is of size 180000 appears to be an outlier; manual inspection of the dataset reveals that the 180000'th word is considerably larger than average:

```
179999    pgi  267504
180000    paradoxically  361996
180001    rambill  42561
```

Understandably, this abnormally large word does not affect the operation time of list and hash table as their complexities are not inherently proportional to the literal size of the word itself.

Scenario 2. – Shrinking Dictionary

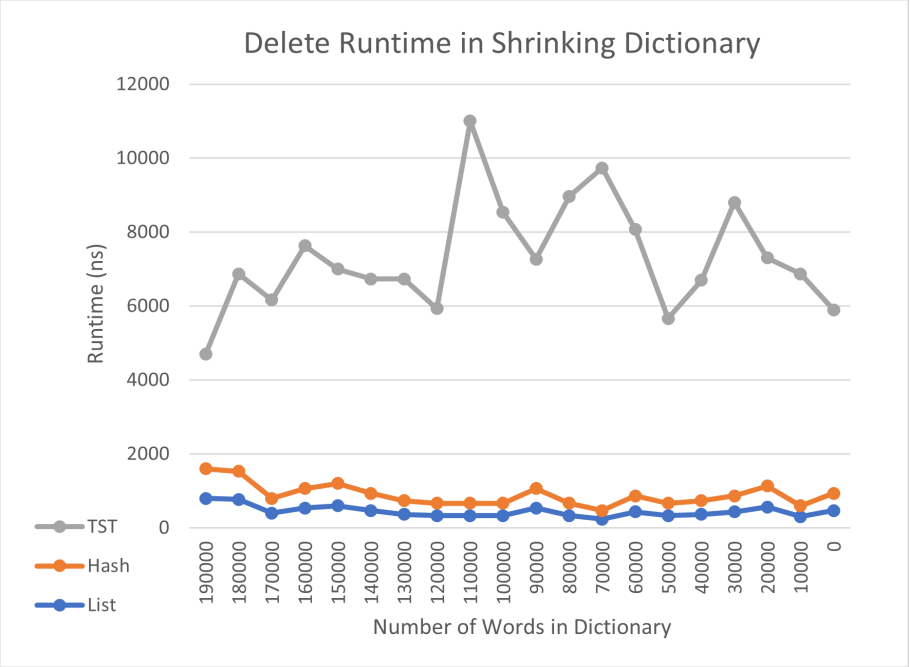

Figure 2. Runtime of the delete function for TST, List and Hash Table implementation.

The hash table and list implementations have a theoretical constant time complexity of O(1), as the deleted word is popped from the end of the list/hash table, rather than found and deleted. While this may negatively impact the results it is also done to avoid resizing the list and hash table while they are being iterated through.
The data collected from our evaluation shows an overall constant trend, despite significant variation, supporting the theoretical time complexity. If we were to select a random word to delete, the time complexity of both the hash table and list delete

would be O(n) instead, in order to facilitate manually searching for the random word in order to be deleted.

Scenario 3. – Static Dictionary

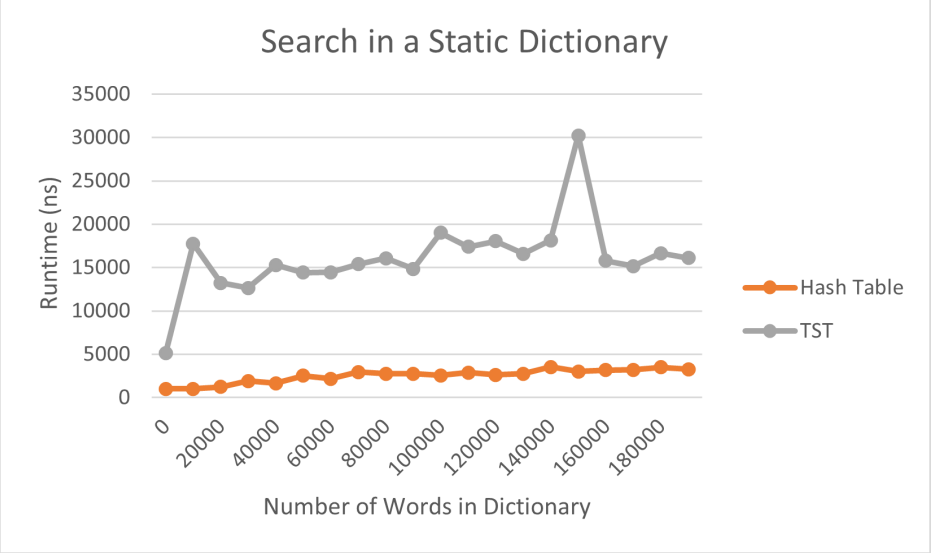

Figure 3. Runtime of the Search Function Hash Table and TST implementations with a static dictionary.

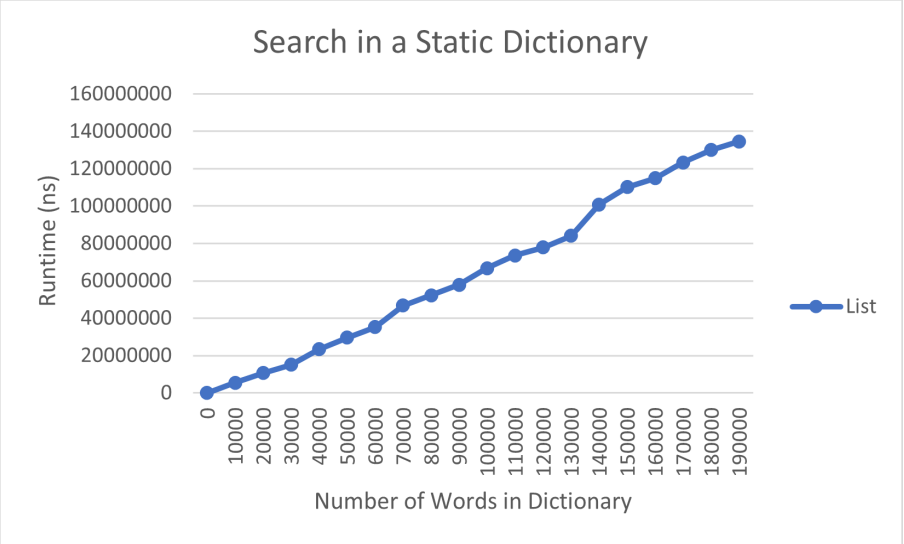

Figure 4. Runtime of the Search Function List implementation with a static dictionary.

Predictably, the graph shows a linear trend in search time for list, and a constant search time for hash table. There is a very slight overall increase in search time for the hash table as the size of the dictionary increases; this is expected behaviour as under the hood the hash table is also index based but within the scope of the C language backend. Therefore the indexing for the hash table is much faster than the python indexing used for lists.

There exists irregular spikes in the ternary search times for search; this is because the ternary search operation process iterates through each letter of the word. This behaviour is exclusive to the ternary search tree, which explains the lack of irregularity for lists and hash tables.

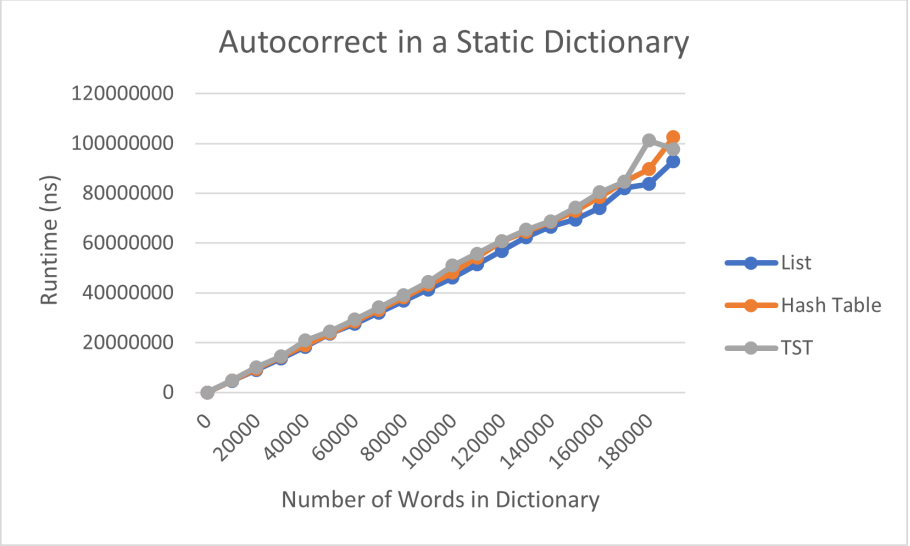

Figure 5. Worst case Runtime of the Autocorrect Function List, Hash Table and TST implementations with a static dictionary.

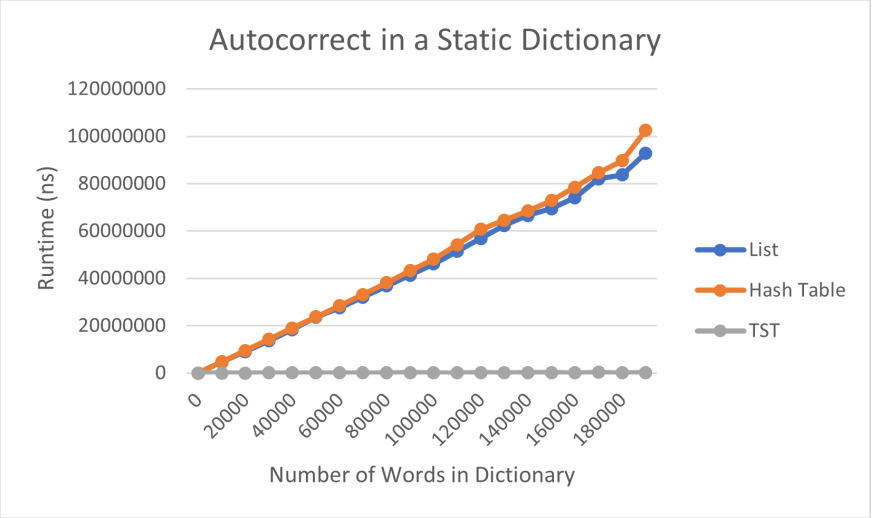

Figure 6. Simulated average case runtime of autocorrect function, static dictionary.

It is important to note that figure 5 targets the worst case; because the autocorrect prefix word parameter is defined as a random letter from the alphabet, as described above, the ternary search tree has to traverse more of the tree to find suitable autocorrections. Therefore, despite the theoretical speed of ternary search tree autocorrect being O(nlogn), figure 5 shows that the worst-case trends towards, or may even well be, O(n).

The larger the prefix parameter given to ternary search, the faster it theoretically runs, as it traverses a smaller percentage of the tree to find corrections. Therefore in order to procure a closer-to-average case for ternary search, we also run an extra test where the prefix is a randomly selected four letter word *(see line 75 in ternarysearchtree_dictionary.py for implementation)* Evidently, the ternary search tree is much faster; list and hash do not change as they still have to iterate through the entire dictionary regardless of word input size. This result is much closer to the predicted average case for ternary search tree autocorrect.

Conclusion

The hash table implementation was the best performing implementation for the growing dictionary scenario. As there is no need for a for loop to check if the word already exists in the dictionary it has a O(1) runtime complexity. Therefore hash tables are recommended for growing dictionary scenarios.

The list implementation was the best performing implementation in the shrinking dictionary scenario. While both the hash table and list implementation have a runtime complexity of O(1), the list implementation is faster as it only requires the deletion of a single element rather than needing to delete both a key and index value. Therefore lists are recommended for use in shrinking dictionary scenarios.

The ternary search tree implementation excelled in the static dictionary scenario. The ternary search and autocorrect implementations have an average runtime complexity of O(nlogn). Therefore ternary search trees are recommended for use in static dictionary scenarios.

Improvements

1. Despite the assignment specification mandating the use of the `time.time_ns()` function, there exists a more accurate timer. The `time.time_ns()` function rounds to the nearest 10 nanoseconds, whereas `default_timer` does no rounding whatsoever. Both natively exist in the import time module. Therefore to achieve more accurate results if the experiment were to be conducted again, `default_timer` should be used instead of `time.time_ns()`.
2. Lists and hash tables, being integrated python data types, are primarily written in the C language under the hood, and thus are compiled using a C compiler. Ternary search trees, uniquely being an abstract data type, are fully written in Python (within the scope of this report only), and are therefore compiled/interpreted in Python. We can presume that part of the list and dictionary implementations will be inherently faster simply due to the fact that they are written in a different, lower level language.