Student 1: Ngoc Bao Vy Le
Student 2: Oliver Southon

We certify that this is all our group's original work. If we took any parts from elsewhere, then they were nonessential parts of the assignment, and they are clearly attributed in our submission. We will show that we agree to this honour code by typing "Yes": YES.

Comparison of List-Based, Hashtable-Based and Ternary-Search-Tree-Based Dictionary Implementations

1.Introduction

The objective of this report is to study how a real-world problem of word completion can be solved by different data structures and algorithms. Another objective is to evaluate the performance of these data structures by various usage scenarios and input data. The goal of word completion is to provide users with predictive suggestions after typing the initial letters of a word. The predictive suggestions are the top 3 words with the highest frequencies in the dataset. In this paper, the three approaches that were used are Lists, Hashtables and Ternary Search Trees.

The efficiencies of these data structures are compared by the running time through the data size and the number of operations(add, delete, search and auto-complete). The execution time was measured by using the time() function on Python 3. To maintain consistency and accurate results, all tests were conducted on one local machine.

2. Experiment Setup

❖ Test Data

From the sampleData200k.txt file, we extracted three datasets in groups of 60,000. We used standard python iterating to read each row in the file, and convert them to WordFrequency objects. We then stored those objects in an array. By the end of it, we had three WordFrequency arrays all of unique values and the same size of 60000.

❖ Generation of scenarios

Before testing each scenario, we made a framework (represented as a Python class) to keep our controlled variables consistent. Each framework contained:

- The subset of the data (an array of WordFrequencies). We made 3 frameworks consisting of 0-60k rows, 60k-120k, and 120k-180k respectively.

- The values representing each starting size of the dictionary. We wanted to keep this constant for both growing and shrinking scenarios. The increments for the growing/shrinking scenarios were [2000, 5000, 10000, 20000, 30000, 40000, 50000].

- The percentage of the current dictionary (defined using one of the values above) that is to be added or shrunken. We set ours to 10%. Hence 200 words were added for the starting size of 2000, 500 for 5000, etc. The same approach applied for shrinking; 200 words would be removed for a starting size of 2000.

Scenario 1 - Growing dictionary

The dictionary is growing in size. The process is as follows

1. The dictionary begins with current sizes of values provided earlier.
2. For each initial size, *n* add operations are performed, where *n* is equal to 10% of the dictionary's initial size.
3. Each running time is recorded for all *n* add operations.
4. The mean is derived from all *n* running times recorded.
5. Finally, the means are compared by initial dictionary size and implementation type.

This way, we can compare not only the differences in add operation times as the dictionary's initial size increases, but also receive a standardised comparison of the efficiency in all three implementations; list, hashtable, and a ternary search tree.

Scenario 2 - Shrinking dictionary

The dictionary is shrinking in size. The process is similar to what was done previously with the growth scenario

1. The dictionary begins with current sizes of values provided earlier. This time, we work in descending order.
2. For each initial size, *n* delete operations are performed, where *n* is once again equal to 10% of the dictionary's initial size. It is important to note that this variable remained the same for consistency in our results. We treated it as a controlled variable.
3. Each running time is recorded for all *n* delete operations.
4. The mean is derived from all *n* running times recorded.
5. Finally, the means are compared by initial dictionary size and implementation type.

The main difference of this scenario, compared to the growing dictionary scenario, is that the results are recorded (and visualised) in descending order and that delete operations are used instead of add operations.

Scenario 3 - Static dictionary

For this scenario, the dictionary will not only be fixed in size initially, but it will also be fixed in size throughout the whole process. Instead of using the initial starting size values that we defined earlier, we will instead break up the dataset into:

- Small (25% of the used dataset)
- Medium'(50% of the used dataset)
- Large (100% of the used dataset)

This leaves us with sizes of 15,000, 30,000, and 60,000 respectively. This was done to provide a more general overview of our results, as the static scenario is less focused on dictionary sizes with static operations and rather more emphasis is placed on the running times of the static operations themselves.

3. Empirical Analysis Results

3.1. Scenario 1 - Growing dictionary

The running times of three data structures were evaluated on the growing data sizes by add operations. For each starting point (the x axis ticks), 10% of that number equates to the number of times add operations were performed, with each average being taken across all 3 subsets.

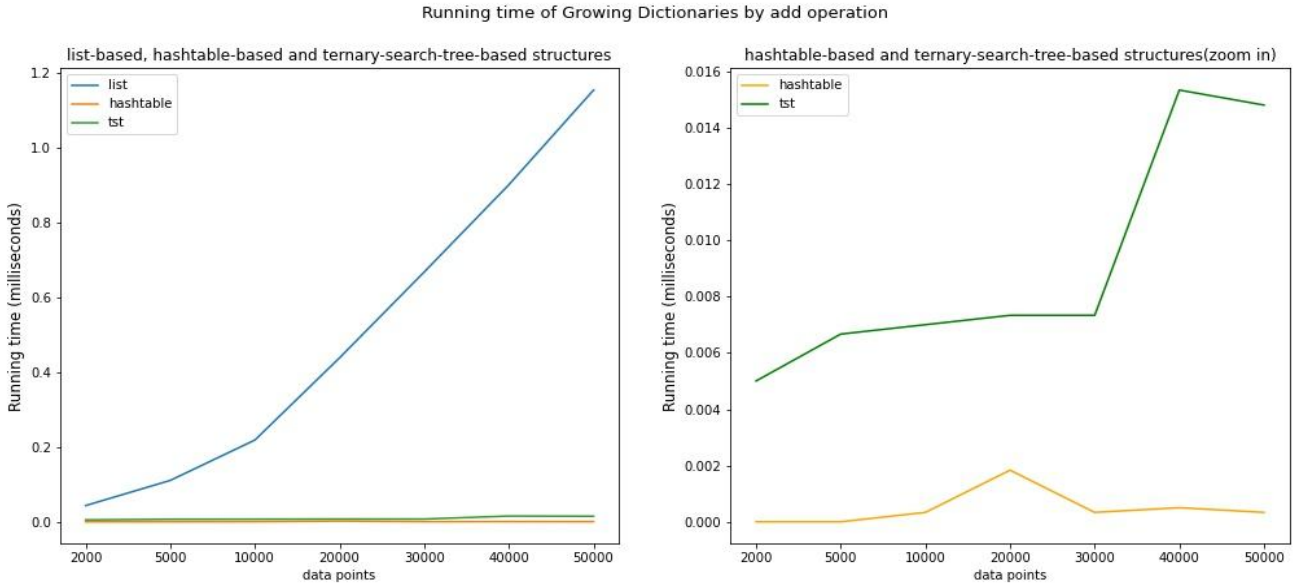

Figure 1: Running Times of growing dictionaries across three data structures by add operation

Observation & conclusion:

Increasing data size has a significant effect on the running time of the list-based dictionary. For TST, the execution time slightly increases when data size is in between 2000 to 4000 where it has a little drop in running time with data size above 40000. The running time of the hashtable-based dictionary has little to non effect on data size, since there is an inconsiderable change detected between 5000 to 40000 data points. Therefore, it can be concluded that the hashtable-based dictionary is the most effective algorithm for a growing dictionary followed by TST and list-based dictionaries.

3.2. Scenario 2 - Shrinking dictionary

The running times of three data structures were evaluated on shrinking data size, ranging from 50000 to 2000 by delete operation. For each data point in the range, we perform a number of delete operations which equals to 10% of each data point.

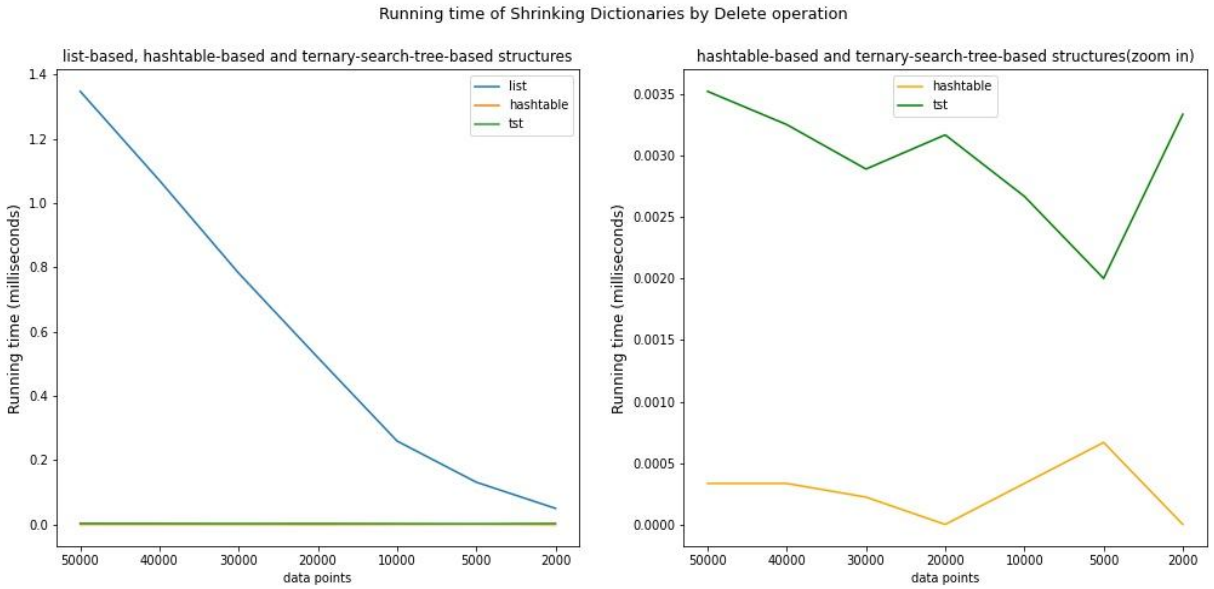

Figure 2: Running Times of shrinking dictionaries across three data structures by delete operation

Observation & conclusion

Decreasing the volume of data size has a significant effect on the list-based dictionary. For TST, the running time drops slightly between 50000 to 5000 data points then lightly increases from 5000 to 2000 data points. There is a marginal shift in running time of the hashtable-based dictionary overall with an imperceptible unsteady change between 20000 to 2000 data points. Thus, it can be concluded that the hashtable-based dictionary is the most effective algorithm for a shrinking dictionary followed by TST and list-based dictionaries.

3.3 Scenario 3 - Static dictionary operations

50 running times of the three data structures were evaluated on static data sizes by search/auto-complete operations, with each average being recorded. This was done on small, medium, and large datasets (15,000, 30,000, and 60,000 rows respectively).

3.3.1 Search operation

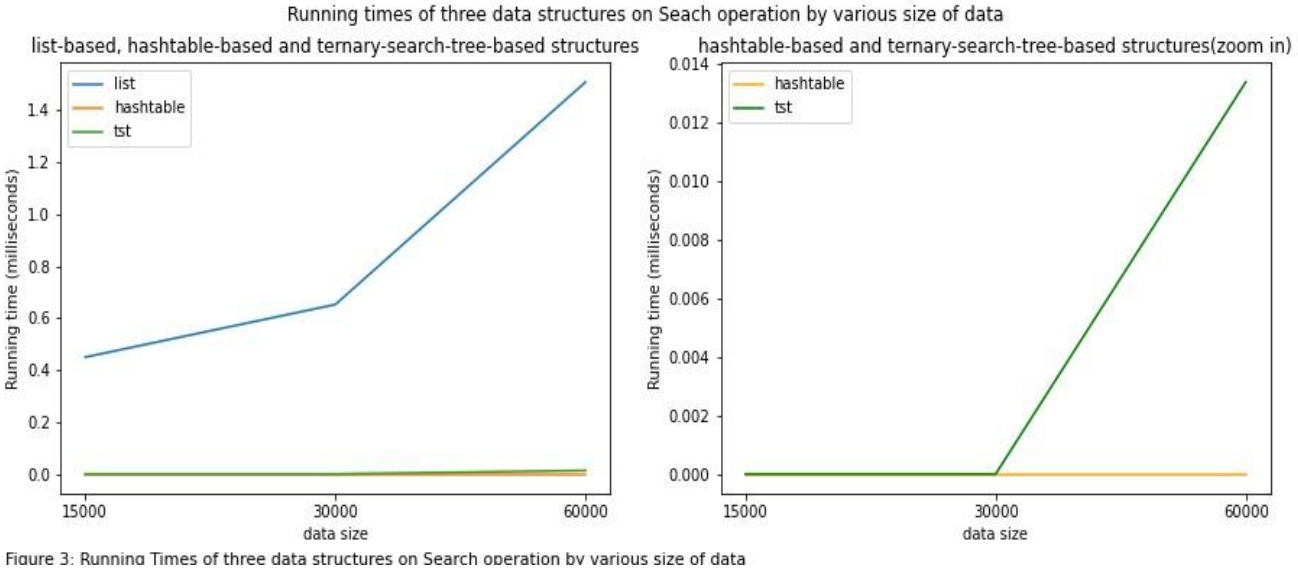

Figure 3: Running Times of three data structures on Search operation by various size of data

Observation & conclusion

The list-based implementation struggled the most with static operations. The extremely poor running times (in comparison with the hashtable and TST) is most likely a consequence of Python's *'unboxed'* array design, which lowers memory space at the cost of performance [1]. Producing much better results, the ternary search tree proved to be a much better implementation. Its tree-like structure is specifically designed for searching, and hence this result is expected. Our fastest implementation, however, was the hashtable. Its significantly low running times can be attributed to the fact that by design, python dictionaries use key-based indexing, making search operations almost instant and the ideal choice for this operation. It performed extremely well, even at higher dataset sizes.

3.3.2 Auto-complete operation

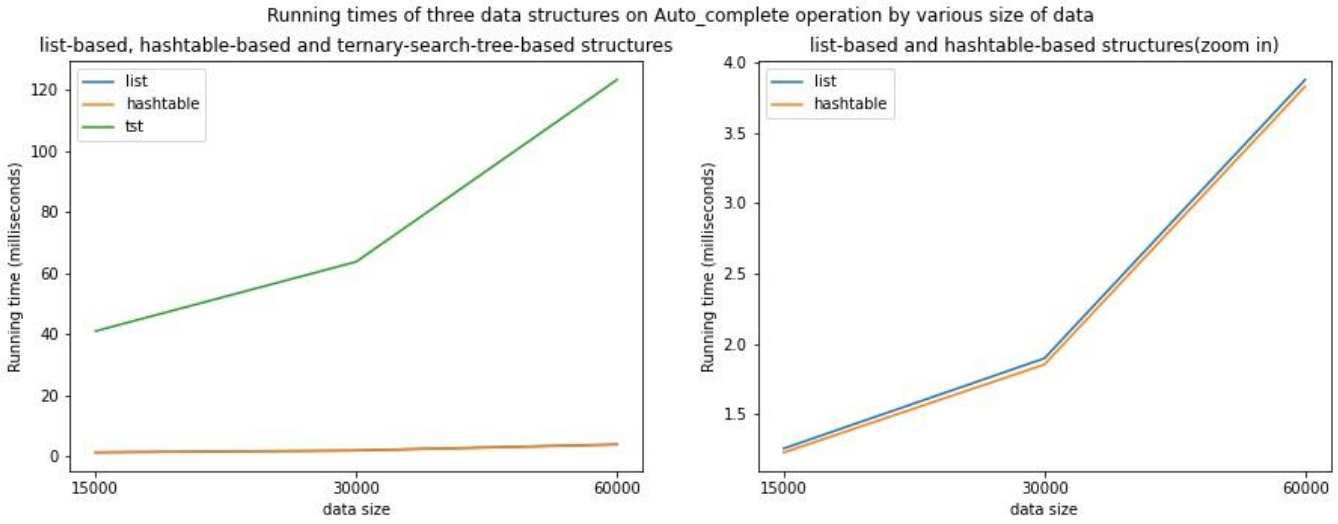

Figure 4: Running Times of three data structures on Auto-Complete operation by various size of data

Observation & conclusion

Increasing data size impacts on the running time auto-complete operation across three data structures, since the execution times increase with the increasing data size. The impact on the execution time is significant for the TST dictionary and small for list-based and hashtable-based dictionaries. The hashtable-based dictionary's running time is slightly faster than the list-based dictionary one. Thus, it can be concluded that the hashtable-based dictionary is the most effective on auto-complete operation, followed by list-based and TST dictionaries.

Before concluding here, it is important to note that our autocomplete method for the TST is objectively quite inefficient. This will be further discussed in *4. Recommendations*. Due to its complex design, it struggled the most with the autocomplete static operation with about 40, 60 and 120 milliseconds for each respective fixed data size. This pales in comparison to the list and hashtable, which both produced similar results under 4 milliseconds. The similarity in running times between these two implementations is most likely due to the fact that the autocomplete functions themselves are very similar. They both involve a full search, followed by a sort. Iterating through a Python 'list' is almost identical to iterating through a Python

dictionary's values. With the way we have designed our autocomplete functions, it can be concluded that both lists or hash tables are an ideal choice for autocomplete operations.

4. Possible Improvements

An ideal autocomplete function for the TST would be to perform a normal word 'search' by taking in the prefix provided. The function would then use that final node (obtained at the end of the search) as a root, and then perform a traversal from there to obtain all words that match the prefix provided. This is done by accessing a node's '*end_word*', which is a boolean attribute.

However, our autocomplete works by *fully* searching the tree (from the tree's root), and placing all of the obtained words into a list. From there, the standard autocomplete method (similar to the list implementation) takes place. Obviously, this adds a lot of unnecessary time complexity, which was reflected in our analysis.

5. Sources

1. Bradfieldcs.com. 2022. *Performance of Python Types*. [online] Available at: <https://bradfieldcs.com/algos/analysis/performance-of-python-types/> [Accessed 22 April 2022].