Algorithm and Analysis Assignment 1

Group 78

- ***Ashmit Sachan s3873827***
- ***Om Khokhar s3886577***

INTRODUCTION

In Algorithm and Analysis assignment one, we implemented different data structures/ algorithms such as list, hashtable and ternary search tree (**TST**) for word auto-completion. We evaluated the performance of these data structures for different scenarios such as growing, shrinking and static dictionaries with multiple input sizes.

Let us go through these data structures one by one quickly.

*The **List** or **Singly-linked-list*** is a mutable data structure (elements in the list can be changed). It is ordered and also allows duplicate values.

A Hashtable is a data structure that utilizes a hash function to generate the index value of the data element. Data is stored in key-value pair.

Ternary Search Tree or ***TST*** is very similar to a Binary search tree in terms of structure. **TST**s are more efficient in storing string values and are comparatively efficient for operations like word completion and spell checking.

DATA GENERATION

PART A:

After completing the code for word auto-completion for lists, hashtable and ternary search trees, we began testing the code on the provided data sets. We ran tests on our code using ***sampleData.txt*** and ***sampleDataToy.txt***. Upon getting the correct output, we performed the tests using ***sampleData200k.txt.*** After doing the tests successfully on our local computers, we ran our code on the RMIT servers to test if it ran without errors.

PART B:

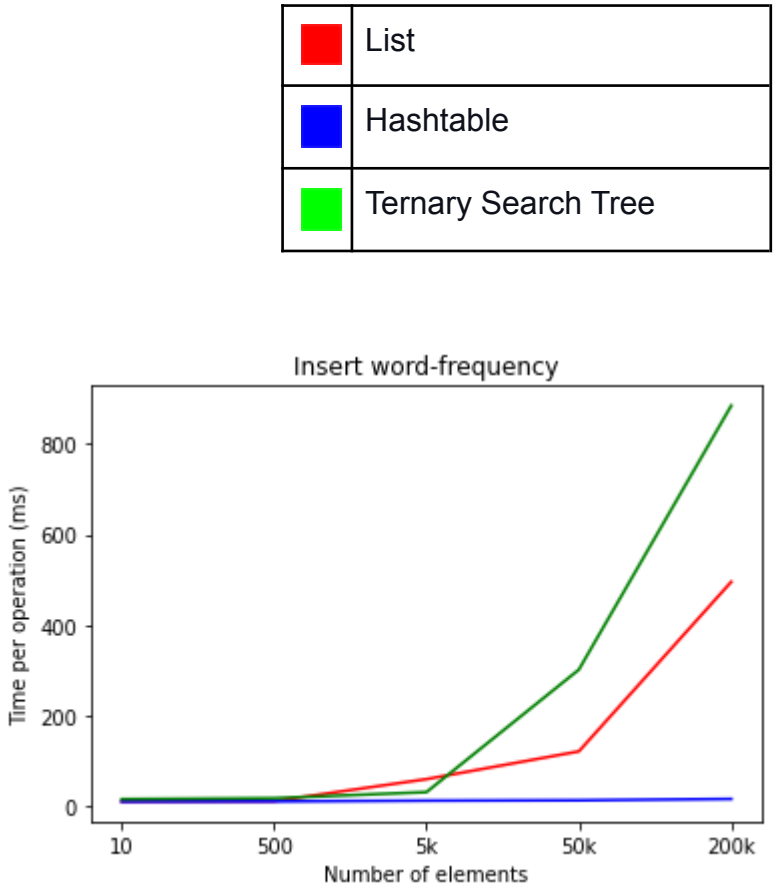
For empirical analysis, we had to do some **DATA PREPARATION** and **RESULT ANALYSIS**. For this part:

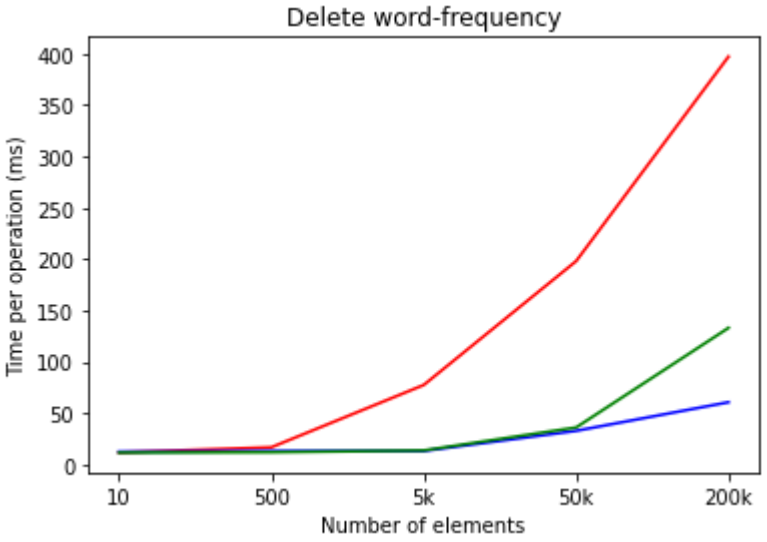
DATA PREPARATION:

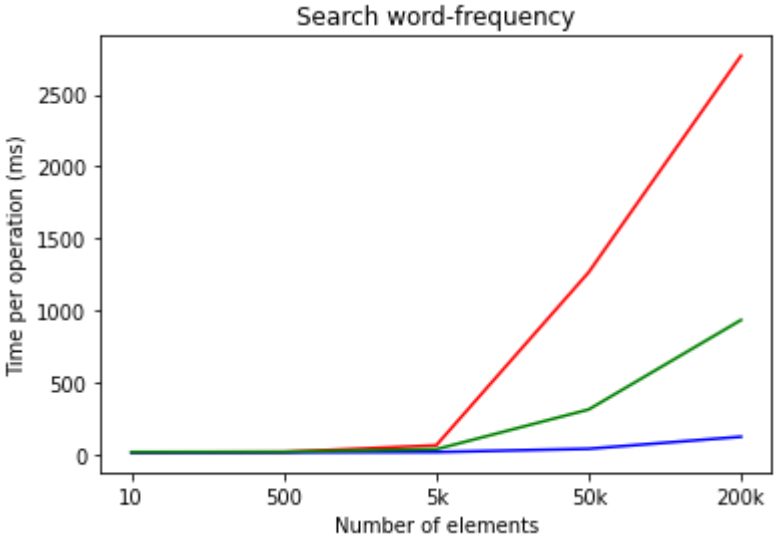
1. **Dividing the data:** We divided our data into several subparts. The data were divided into sets of 10, 500, 5000, 50000 and 200000 word-frequency pairs

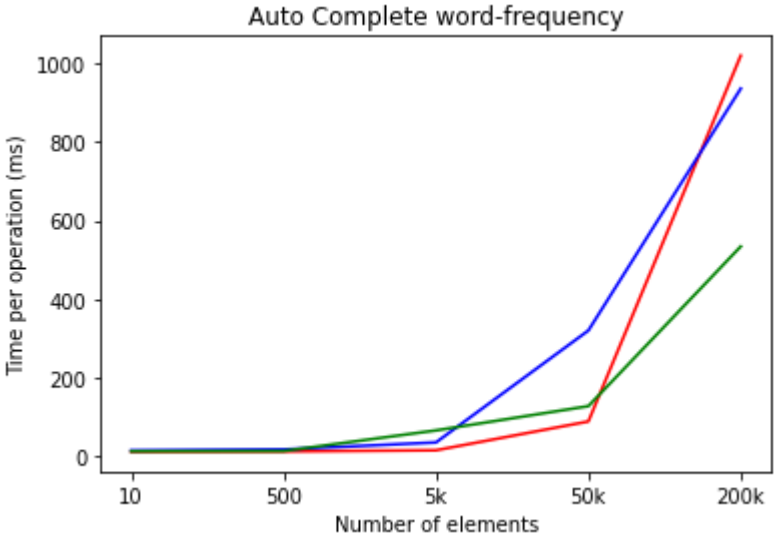
using a python program. The program is in the generation directory (generation/main.py)

2. **Measuring time:** To measure the time taken by any operation, we took a unique approach. Before running the python file from the terminal, if you fetch system time, then run the file and then get system time again, you will get the time in microseconds the program took to run. In this manner, we managed to get the run times of 5 add, delete, search and auto-complete operations respectively on each data structure. We did this multiple times and took the average time and plotted graphs using the average time which we got.
3. **Plotting graphs:** using a popular graph plotting library, matplotlib, we plotted graphs with the data obtained from the above activity. Proceeding to the Jupyter Notebook. Graphs are listed below:




Delete word-frequency
Time per operation (ms)
0
50
100
150
200
250
300
350
400
10
500
5k
50k
200k
Number of elements


Search word-frequency
Time per operation (ms)
0
500
1000
1500
2000
2500
10
500
5k
50k
200k
Number of elements


Auto Complete word-frequency
Time per operation (ms)
0
200
400
600
800
1000
10
500
5k
50k
200k
Number of elements

ANALYSIS OF RESULTS:

From the graphs we can be observe the following data.
During use case **Scenario One**, that is, addition of the elements of the dataset in which the list is growing in size. The hashtable has performed the best. It's performance was much better compared to List and Ternary Search Tree:

1. Time complexity for the list increases linearly with increase in number of elements. It makes sense because each time an element is added, a new list is made. Previous elements are then transferred to the new list and the new element is added to the last.
2. Hashtable's time complexity was more or less the same for any dataset size. This was because of the way it was implemented. Any new value is directly added to the dictionary as a key-value pair, that is, word-frequency pair.
3. The performance of TST was also more or less static over the whole dataset and it had O(log n) time complexity. In comparison to hashtable, TST was quite slower. The list performed similarly to TST and hashtable when the datasets were smaller, but it kept on increasing as the dataset grew bigger. The list had a time complexity of O(n log n).

During use case **Scenario Two**, that is, deleting elements of the dataset in a shrinking dataset. Time taken by list increases linearly with the size of the dataset. Hashtable managed to perform the best. The performance of the hashtable was constant regardless of the size of the dataset. TST performed well in this case as well. There was a subtle increase in time seen as the size of the dataset increased. Time taken by list increased linearly. More the number of elements, the more are the number of iterations to find and delete it.

In use case **Scenario Three,** the performance of data structures while searching for a word and word auto-completion.

In **search** and **autocomplete operations**, time taken by list and hashtable increased exponentially with the increase in size of dataset. Ternary Search Tree outperformed list and hashtable in autocomplete. Although the time taken by TST increases with the size of dataset, the increase in time is not explosively high.

Theory vs Practical

Theoretical vs practical time complexity of TST for each of the use cases.

Theoretically, the time complexity of TST should be O(log n) for all the four use case scenarios i.e. add/insert, delete, search and autocomplete. Our observed graphs showed that the complexity of the TST was found to be O(log n).

Now let’s discuss the hashtable data structure.

Theoretically, the time complexity of the hashtable is O(1) in insertion, deletion & search and O(n) in word completion. It was observed from our graphs that it showed O(1) complexity for insertion, deletion and search operations which matches our graphs. However, our graphs showed O(log n) complexity which doesn’t match the theory.

Finally, let’s discuss the lists.

Theoretically, the time complexity of the list is O(n) for insertion, deletion & search and O(n log n) for word completion.In addition, our graph supports the theory as we found that the time complexity was O(n) for insertion and deletion and O(n log n) for the search and word completion.

Conclusion

From all the data generated and explained above, it's formidable to say that hashtable is the good option for dataset sizes below 5000 word-frequency pairs. Any dataset size over 5000 word-frequency pairs should be built with TST.