1.0 Data Generation.

The provided text file “sampleData200k.txt” contains 200k records, each consisting of a word and its corresponding frequency. We imported this file into Microsoft Excel and copied to multiple sheets with different set of records using the formula, for example “A1:A500”, which specifies a range from the first cell (A1) to the 500th cell (A500) in column A of the sheet. We used random range of dataset, and this process allowed us to create various datasets with different numbers of records, ranging from 500 to 175,000. These datasets were utilized for conducting empirical analysis experiments.

1.1 Dataset Segregation.

The datasets consist of a range of sizes, and we've categorized them accordingly. Each dataset has been generated with a random range of number of records. Following the naming convention "sampleData<Number of records>.txt".

1.1.1 *Small Dataset.*

We randomly selected 500 and 1k records using "sampleData200k.txt" and saved these records in separate files. Subsequently, we conducted tests on all the functions to analyse their time complexity through empirical analysis.

1.1.2 *Medium Dataset.*

The reason for the second phase of dataset segregation was to gain insights into the behaviour of functions as we extracted the dataset size exponentially from 1k records to over 10k records, and up to 25k records in random. This approach allowed us to observe and analyse the variations in time differences and complexities as the dataset size grew.

1.1.3 *Large Dataset.*

We continued our analysis by generating a third dataset consisting of 50k records. Eventually, we expanded our dataset range by including additional datasets with 100k, and 175k, records in random. This set of datasets allowed us to further investigate the performance of functions across a random and wide spectrum of records

1.1.4 *Time Function.*

After gathering the datasets, we utilized time function from python module we tested out the “time.time()” and “time.time_ns()” from the python module functions. The function “time.time()” returns the time in seconds since the start of the time as a floating-point number whereas “time.time_ns()” returns time as an integer number of nanoseconds since the start of the time (Python Software Foundation 2023).

In our experiment, we switched to using "time.time_ns()" instead of "time.time()" for plotting. This change offers finer-grained units in our graphs, enhancing clarity and precision in analysing time complexity trends. When dealing with quick function execution times, measuring in nanoseconds helps capture subtle variations for a better empirical analysis of scaling with input sizes.

2.0 Experimental Set-up.

We created various datasets and implemented code for our data structures. To assess their performance, we're testing how long it takes to execute these implementations across different datasets. This will help us understand how well our data structures and algorithms (Array, Linked list, and Trie) perform under various conditions against theoretical interpretations. In our testing process, we execute the operation by inserting a record both at the beginning and at random positions within the file. Importantly, we deliberately exclude the record required for the operation. We repeat this process ten times for each scenario and calculate the average outcome. By operation we mean adding a word "abandon", searching word "abandon", deleting word "abroad", and auto-complete "ab". This approach ensures that our estimations are robust and reliable. We further enhance our analysis by plotting these results on a line graph and conducting empirical analysis for deeper insights.

Command: python3 dictionary_file_based.py <approach> <sampleData(Number of records).txt> <testfile.in> <testfile.out>

3.0 Empirical Analysis.

The practice of algorithm analysis involves conducting practical experiments and measurements, without solely depending on theoretical analysis. This process includes executing algorithms with real-world data, measuring their runtime behaviour, and assessing resource usage (Knuth n.d.).

3.1 Search Operation.

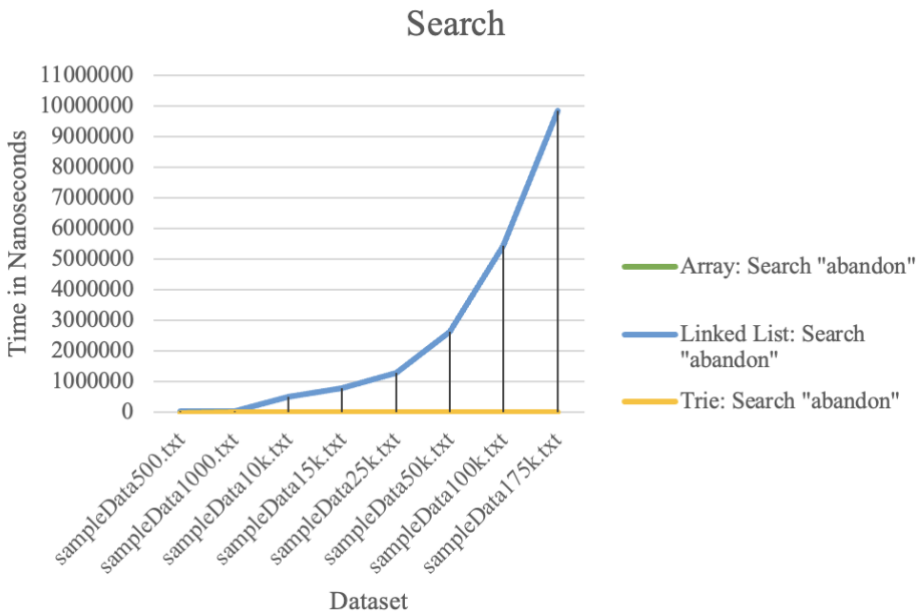

Figure 3.1.0 Average time taken by Search operation.

3.1.1 *Array based search.*

The search operation involves searching the input parameter 'word' in the array. On average, searching for "abandon" with its frequency across all the datasets takes nearly '5872.2222' nanoseconds this shows the liner relationship with dataset size and the time taken which is slightly increasing. According to the graph (Figure 3.1.0) the arrays-based search is constant

and appears to be **O(1)** with empirical analysis but considering the growth of the dataset its slightly liner.
The complexity of the array in theory is **O(n)** where n is the size of the array, and it traverses through the whole array for each input parameter that needs to be searched making it less efficient for larger datasets (freecodecamp 2019).

3.1.2 *Linked-List based search.*

The linked list search operation involves searching the input parameter 'word' if its available within the created nodes. Analysis indicates that the node "abandon" with its frequency considering all nodes in the linked list, exhibits a pronounced **quadratic** relationship with the dataset size, time taken for this operation is nearly '2577675' nanoseconds which has significantly increased. According to the graph (Figure 3.1.0) and empirical analysis the operation is **quadratic** as it checks each node resulting with the complexity of **O(n^2)** whereas theoretically it exhibits the complexity of **O(n)** (Forcha 2018).

3.1.3 *Trie based search.*

Tire-based search consists of checking if there is an edge in the Trie node for each character in the input 'word' parameter. If a matching edge is found, proceed to its child node, continue till the node marked as the end of the word "abandon" (i.e., 'is_last' is true) (Geeksforgeeks 2011). The average time taken is '2104.1666' nanoseconds over all the datasets, according to the empirical analysis and graph (Figure 3.1.0) the performance remains fairly constant resulting in a complexity of **O(1)**
But theoretically it exhibits **O(m)** where m is the length of the word being search which make it the most efficient (Geeksforgeeks 2019; Geeksforgeeks 2011).

3.2 Add Operation.

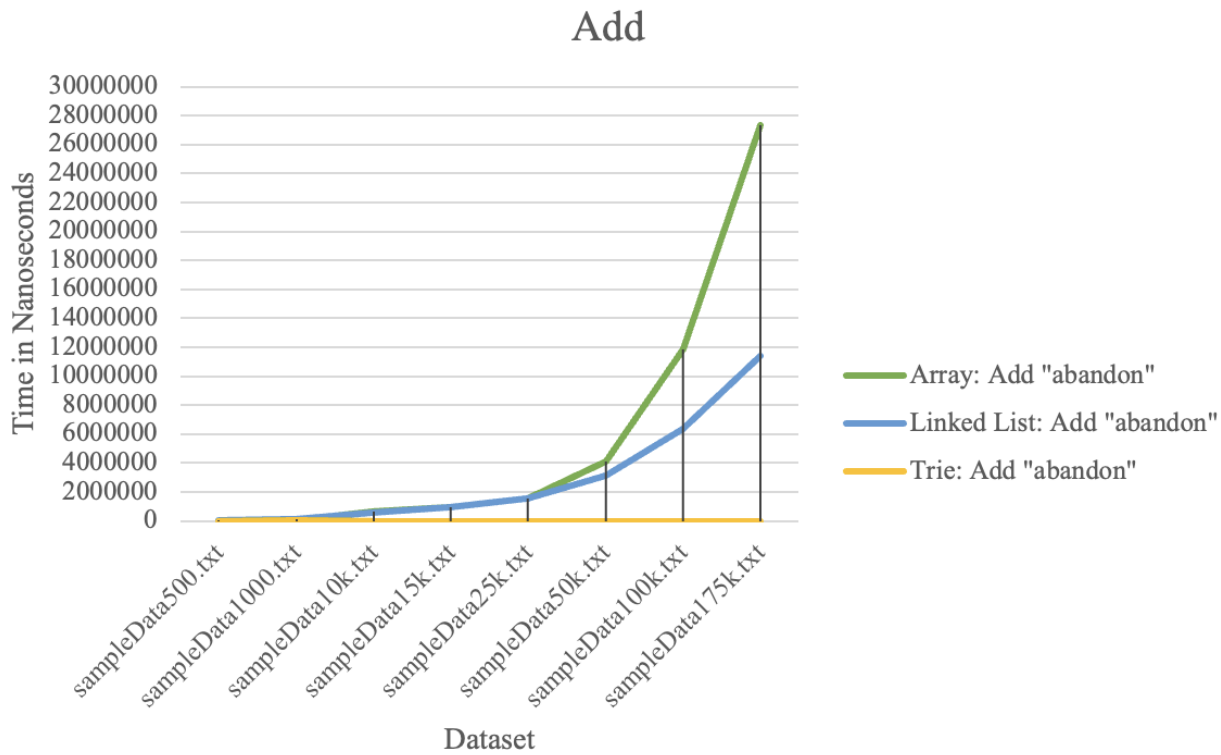

Figure 3.2.0 Average time taken by Add operation.

3.2.1 *Array based add.*

The average time taken by the array-based adding for all the datasets is around '5818012.5' nanoseconds which is greater than Trie and the linked list. The add operation is expensive in terms of time because it must first iterate through each item to check if the 'word' input parameter "abandon" exists and if it's not part of the array then the word and its frequency must be added followed by sorting the array. According to the graph (Figure 3.2.0), it exhibits a **quadratic** relationship with the increasing dataset size i.e., **O(n^2)**. Therefore, the time complexity of adding the word in theory would be **O(n)** and sorting the array would take **O(n log(n))** but empirically it shows **O(n^2)**.

3.2.2 *Linked-List based add.*

In linked list each node is traversed to check if the input parameter 'word' already exists, if there is no node that contains "abandon" then a new node is created with the word and its frequency. The average time taken by this operation for all the datasets is up to '3015287.5' nanoseconds making it relatively costly due its complete traversal strategy. Its execution time will increase proportionally as the dataset size grows because the time complexity climbs linearly with **O(n)** both in theory and empirically**.**

3.2.3 *Trie based add.*

The average time taken by Trie for all the dataset in consideration is approximately '2500' nanoseconds making it most efficient. The operation is mainly carried on with the length of the word "abandon", which is being added. The data structure iterates through each letter creates children node which make it the most efficient among the data structures we are testing. According to the Figure 3.2.0 and empirical analysis the operation tends to exhibit a constant time nearly **O(1)**.

However theoretically, the time complexity is **O(m)** where m is the length of the word (Geeksforgeeks 2019; Geeksforgeeks 2011).

3.3 **Delete Operation.**

Figure 3.3.0 Average time taken by Delete operation.

3.3.1 *Array based Delete.*

The average time taken by the array-based delete for overall datasets is around '5959587.5' nanoseconds which is more significant than Trie and the linked list. The delete operation is time consuming because it must first iterate through each item to check if the 'word' input parameter exists. If it's part of the array, then the word "abroad" and its frequency must be removed followed by sorting the array. According to the graph (Figure 3.3.0), it exhibits a **quadratic** relationship with the increasing dataset size i.e., **O(n^2)**. Therefore, the time complexity of adding the word in theory would be **O(n)** and sorting the array would take **O(nlog(n))** but through empirical analysis it shows that it exhibits **O(n^2)**.

3.3.2 *Linked-List based Delete.*

The average time taken for the overall dataset by linked list delete operation is near '4532637.5' nanoseconds. The operation includes iterating through the node and finding the input parameter 'word' if "abroad" exists in the nodes and pointing to the next of next node to erase the node from the linked list. As per the empirical analysis and the Figure 3.3.0 this operation seems to be taking **O(nlog(n))** as the line in the graph leans below the '*Array based Delete*' with a curve. However theoretically, this operation is as expensive as **O(n)** where n is the number of nodes (Forcha 2018).

3.3.3 *Trie based Delete.*

The operation iterates through each letter in the input 'word' parameter traversing to the bottom of the tree structure edge of the letter. If the word "abroad" is found, then the necessary changes are made to the flags which preserve the structure without deleting the relevant children. In theory the worst case of deletion is **O(m)**, however according to the empirical analysis and Figure 3.3.0 the line in the graph hovering on the dataset-axis which is **O(1)** indicating the average time of '2700' nanoseconds proving to the most efficient data structure.

3.4 Auto-Complete Operation.

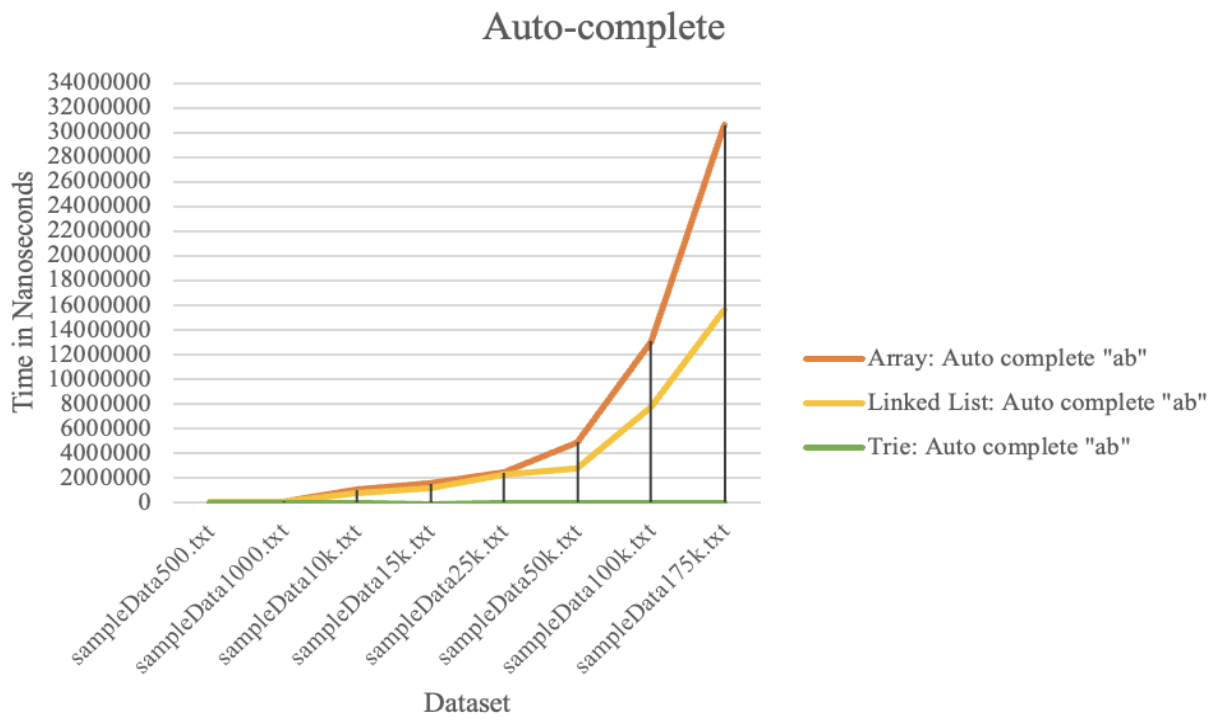

Figure 3.4.0 Average time taken by Auto-complete operation.

3.4.1 *Array based Auto-complete.*

The average time taken by auto-complete operation for overall datasets is around '6718644.445' nanoseconds which is more significant than Trie and the linked and is time consuming as it must first iterate through each item to check if the 'prefix_word' input parameter to find the closest words to "ab". For this operation its essential to sort the array with the highest frequency that added up more time. According to the graph (Figure 3.4.0), it exhibits a **quadratic** relationship with the increasing dataset size i.e., **O(n^2)**. However, the time complexity of auto-completion operation in theory is **O(n*m)** where n is the number of items and m is length of the prefix and sorting the array would take **O(*l**log*(l)*)** where *l* is number of words (Geeksforgeeks 2019)**.**

3.4.2 *Linked-List based Auto-complete.*

The average time taken for the overall dataset by linked list auto-complete "ab" is near '3815416.6665' nanoseconds. The operation includes iterating through the node and finding the closest nodes to parameter 'ab', if it exists then returning nodes by sorting them with respect to higher frequencies. As per the empirical analysis and the Figure 3.4.0 this operation seems to be taking **O(nlog(n))** as the line in the graph leans below the '*Array based Auto-complete'* with a curve. However theoretically, this operation is as expensive as **O(n*m)** where n is the number of nodes and m is the length of the word.
Sorting linked list, time complexity is **O(*l*log*(l)*)** where *l* is number of words (Geeksforgeeks 2019).

3.4.3 *Trie based Auto-complete.*

The operation iterates through each letter in the input 'word' parameter 'ab' traversing recursively to the bottom of the tree structure edge of the letter to find the closest words with top 3 highest frequencies by performing a sort function. According to the empirical analysis and Figure 3.4.0 the line in the graph leans on the dataset-axis which is **O(1)** indicating the average time of '138050' nanoseconds proving to the most efficient data structure.
In theory it typically takes O(m) where m is length of the word and sorting takes **O(*l*log*(l)*)** where *l* is number of words (Geeksforgeeks 2019; Geeksforgeeks 2011)**.**

4.0 Recommendations & Conclusion.

The theoretical analysis is an estimation of required recourses for an algorithm to solve a specific problem with this concept in mind the above data structures performance is computed (Geeksforgeeks 2021). However, according to empirical analysis, the graphs, and the time measurement of the execution of the data structures evidently show that they take less time than expected. However, Array-based performance took longer than the estimated theory, proving it to be an expensive data manipulation operation. Whereas there is no significant difference in the empirical and theoretical analysis of Linked list operations, it still involves node traversal which could be optimized by combining it with other data structures.

Among array, linked list, and Trie we found that Trie is more efficient than the rest. Through empirical analysis, we found this data structure performs better than expected. Practically the graphs indicate that all the tree-based operations take nearly **O(1)** especially on word-based operations which was expected to be **O(n log(n))**. Considering the performance tries could be memory intensive when there is exponential increase in the size of the data, it is good to test it with real-time data as an overhead for Trie.

5.0 References.

1. Python Software Foundation (2023) *time — Time access and conversions — Python 3.10.12 documentation*, *docs.python.org*, https://docs.python.org/3.10/library/time.html#epoch, accessed 30 August 2023.

2. Knuth DE (n.d.) *The Art of Computer Programming--Volume 1: - ProQuest*, *www.proquest.com*, https://www.proquest.com/docview/926211588?accountid=13552&parentSessionId=MauDMZKcUqo6vdOaNY5IjZTbUkwX7VclM%2BIR4MH08uE%3D&pq-origsite=primo, accessed 7 September 2023.

3. Geeksforgeeks (2011) *Trie | (Insert and Search)*, *GeeksforGeeks*, https://www.geeksforgeeks.org/trie-insert-and-search/, accessed 2 September 2023.

4. Geeksforgeeks (2021) *What is algorithm and why analysis of it is important?*, *GeeksforGeeks*, https://www.geeksforgeeks.org/what-is-algorithm-and-why-analysis-of-it-is-important/, accessed 10 September 2023.

5. freecodecamp (2019) *The complexity of simple algorithms and data structures in JS*, *freeCodeCamp.org*, https://www.freecodecamp.org/news/the-complexity-of-simple-algorithms-and-data-structures-in-javascript-11e25b29de1e/#:~:text=Arrays, accessed 2 September 2023.

6. Forcha C (2018) *Linked List Time Complexity*, *Linux Hint LLC*, https://linuxhint.com/linked-list-operations-time-complexity/, accessed 2 September 2023.

7. Geeksforgeeks (2019) *Python | Sort tuple list on basis of difference of elements*, *GeeksforGeeks*, https://www.geeksforgeeks.org/python-sort-tuple-list-on-basis-of-difference-of-elements/, accessed 7 September 2023.