ALGORITHM IMPLEMENTATION AND EMPIRICAL ANALYSIS REPORT

DATA GENERATION AND EXPERIMENTAL SETUP

The 200k data sample text file (sampleData200k.txt), we have generated our data file based on the given sample. We firstly created a python file named “data_Generation.py” within which we ran loops to break the big text file into smaller data samples. Each method within the python file created a new data file, we had decided to split the data into 25k, 50k, 100k, 150k, 175k and finally 200k.

Along with the different sizes of data files, we also created different input files for all the functions for measuring time of each function. The different files created were search.in, delete.in, add.in and autocomplete.in. Each input file had 12 commands in it which will help us in recording running time multiple times. For all the input files except add.in, we selected words from the bottom of each sample data file so that we can record running time in worst case.

```
def sample25K():
    main_datafile = open("sampleData200k.txt", "r")
    smallSample = []
    sampleFile25k = open("Data25k.txt", "w")
    for line in main_datafile:
        smallSample.append(line)
        sampleFile25k.write(line)
        if len(smallSample) >= 25000:
            break


def sample50K():
    main_datafile = open("sampleData200k.txt", "r")
    mediumOneSample = []
    sampleFile50K = open("Data50k.txt", "w")
    for line in main_datafile:
        mediumOneSample.append(line)
        sampleFile50K.write(line)
        if len(mediumOneSample) >= 50000:
            break
```

Regarding capturing total time of execution for each function we altered dictionary_file_based.py which had the commands to call each function. We inserted two variables start_time and end_time before and after each command call respectively inside the python file (e.g., “agent.Search(…)”). The time measurement used was nanoseconds and for that we used the time.time_ns() method which is pre-built into python.

Following this, we ran the terminal command which was provided for each data structure to record the total time of each function for a particular data structure. We had written additional code which allowed us to capture timings of each function by printing the running time for each input within an input file and at the

end, average of all. We successfully recorded the timings and noted in a separate file CSV file(can be accessed by clicking here) after running commands in the terminal for three data structures and functions which had been implemented in the previous task.

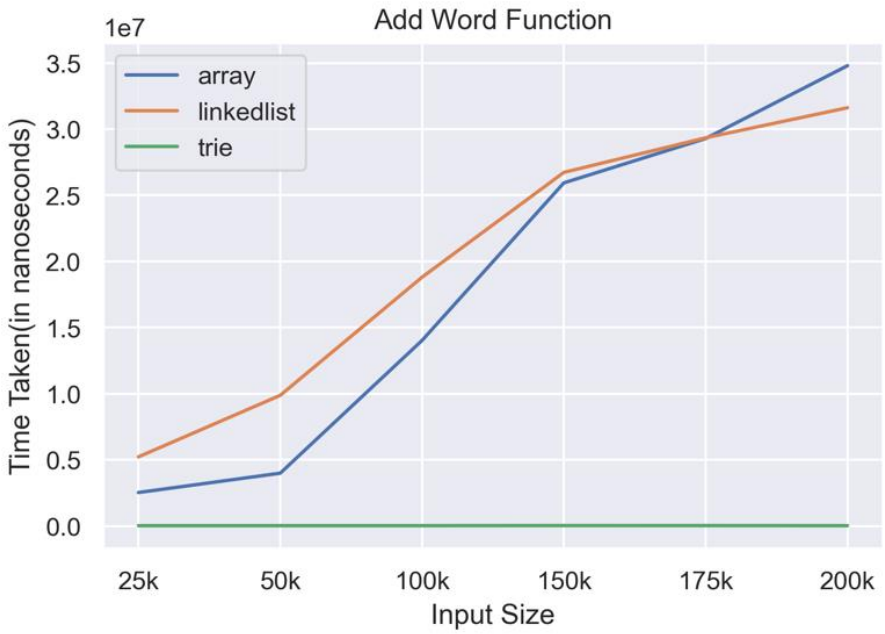
After capturing the timings of each function for different data structure, we recorded the timings on a table inside a word document. The timings we have used for the graph are an average of several runs of each function for all data structures. For example, we ran LinkedList three-four times and then got an average to make sure the timings were accurate and reflected consistency.

After we finished recording times of each function. We plotted the graph using a file called graphs.ipynb which can be accessed by clicking here. I added all the timing of each function of each data structure in an array. Then I used seaborn library to plot the graphs of each function for all three data structures over a range of different size of input sizes.

As the timings of array and LinkedList were very large as compared to timings of trie, that in all graphs, it seems that trie has taken a constant time for all functions to run which is not the case. So, we decided to make graphs for trie individually for all functions. All the graphs can be accessed from google drive by clicking here.

FUNCTIONAL EVALUATION, ANALYSIS AND GRAPHS:

ADD FUNCTION: -



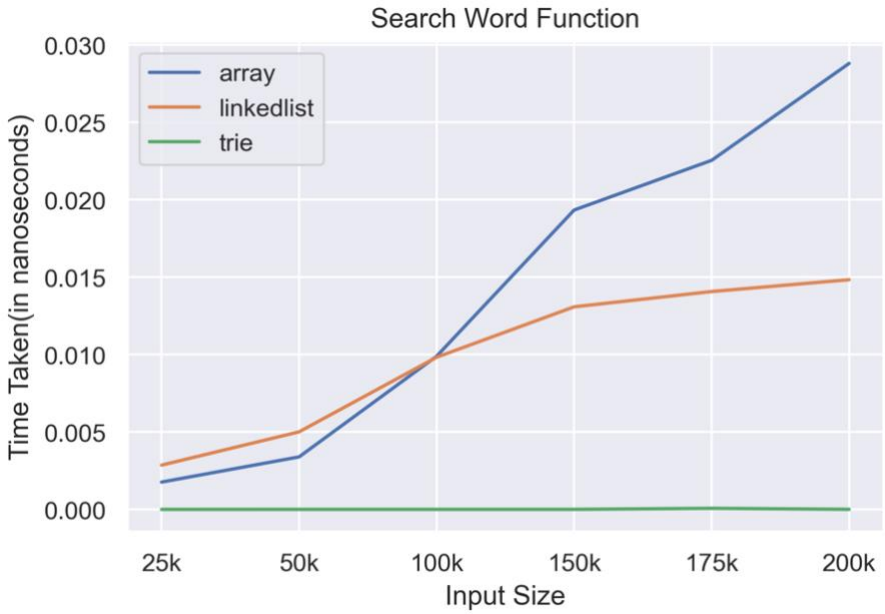
Theoretical Time Complexity = O(n) -> Array, LinkedList and Trie

Empirical:

- The above graph displays the timings for add function for the three different data structures implemented. We tested the function on various data samples ranging from 25,000 to 200,000 words.
- The array and LinkedList data structure shown in the blue and orange respectively, have a linear growth from 0 to 200k.

- For array data structure we can see that from 25k-50k the time has grown at gradually in a linear manner, then from 50k-150k the growth is steeper while maintaining linear growth pattern, and the same linear growth can be seen between 150k-200k at a gradual slope. In terms of LinkedList, the growth in time from 25-150k is linear and steep, which means that time increased at a faster rate between these ranges. However, after 150k, the growth is more gradual while maintaining a linear growth pattern.
- The Trie implementation for add function was much faster compared to the previous two data structures.

SEARCH FUNCTION: -

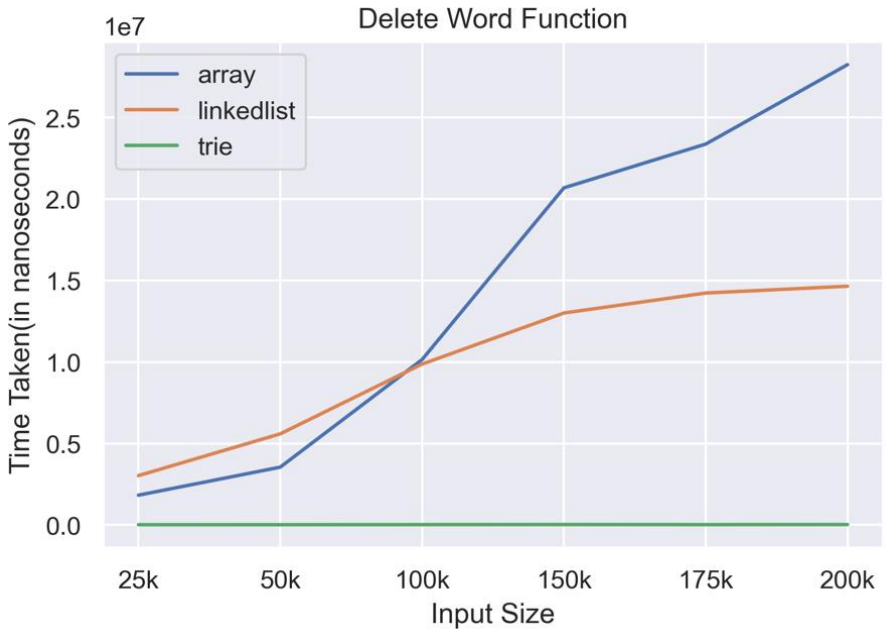


Theoretical: Time Complexity -> O(n) – Array, LinkedList and Trie

Empirical:

- The graph above is for the search functionality which we implemented for each data structure, and then tested using the different input sizes.
- As displayed in the graph the growth of array and LinkedList is linear.
- The graph shows us that between 25k and 50k input size the growth of array and LinkedList is very gradual, however after 50k, the array seems to have a steeper growth compared to LinkedList. After 100k mark, the growth of LinkedList is more gradual moving towards 200k input size while still being linear. The growth of array also becomes gradual after 150k but the difference in times between the two data structures is quite clear.
- The graph of tire has lower time compared to LinkedList and array throughout all the different input sizes.

DELETE FUNCTION: -

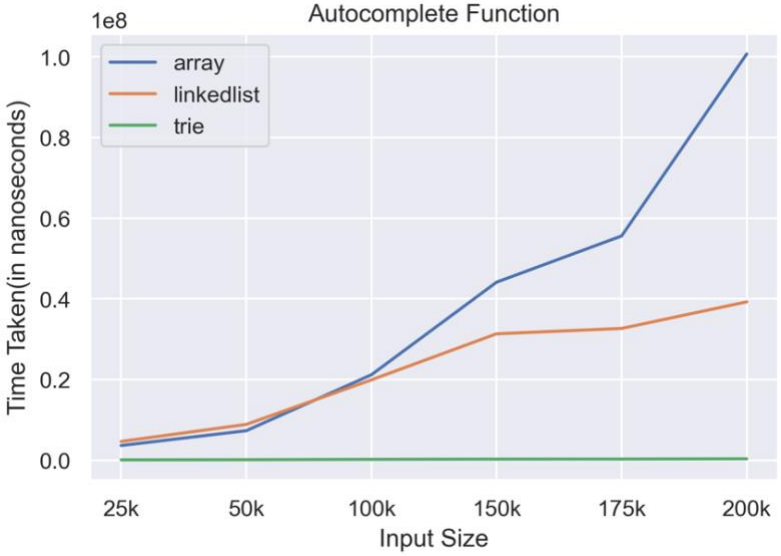


Theoretical: Time Complexity = O(n) – Array, LinkedList and Trie

Empirical:

- In the graph above we can see that both and array and LinkedList had a similar growth rate between 25k and 50k. However, between 50k and 100k we can see a shift in the growth between the two data structures, as the growth of LinkedList becomes gradual moving towards large number of inputs, while the graph of array has a steeper growth between 50k and 150k.
- After the 150k mark, while LinkedList becomes even more gradual moving towards the largest input size, we can see that the growth of array also becomes gradual as compared growth of array between 50k and 150k.
- Just like the previous functions the trie data structure remains faster compared to the other two data structures and thus we can observe a horizontal line along the x-axis at lowest possible y-value
- shown in the graph.

AUTO-COMPLETE FUNCTION: -



Theoretical: Time Complexity = O(n) – Array, LinkedList & Trie(O(M) in case of trie for traversing, where M is length of longest word)

Empirical:

- The graph of LinkedList and Array both have a gradual linear growth between the 25k input size and 100k input size. We can analyse that time taken by both the data structure between the first three input sizes is very similar and moving from 50k to 100k we can see that both graphs intersect.
- The difference is much clear between the time taken after 100k input size. While the growth of LinkedList remains gradual in a linear pattern, the growth of array is much steeper maintaining the linear growth.
- In the final input size, the growth for both LinkedList and array are steeper. In comparison, the growth of LinkedList seems to be more gradual in the last input size (as opposed to array) but when we compare previous growth rates of LinkedList in large input size, it can be noticed that there is a small spike between 175k and 200k. Similarly, the growth spike in array is much steeper compared to previous large input size.
- In this case, also the time taken by trie to complete the auto-complete functionality is much lower as compared to array and LinkedList.

OVERALL ANALYSIS:

- The generated datasets and the timings collected for the 4 different functions search, add, delete, and auto complete, while the timings of both LinkedList and Array are very similar in small input sizes, however as the input sizes increase so does the time taken, therefore we come to conclude that Array takes the most time overall as compared to LinkedList and Trie. We can also conclude that out of the three data structures, the Trie data structure is the fastest.
- When comparing LinkedList and Array we can see that overall LinkedList is faster and the reason behind LinkedList that is because, during the search, delete we selected words which were at the bottom of the dataset file. The LinkedList data structure has a last in first out structure, which means that when search is executed using LinkedList, the word is closer to the head thus taking less time to search for the word. In the array the word is at the bottom which means that we must traverse all the to the bottom of the array to find the word.
- However in the Add function we can see that LinkedList takes more time overall as compared to array, and the reason behind this is that there are several ways to insert a new value In LinkedList, the way we have chosen is appending the new value at the end of the LinkedList, which means it has to traverse through the whole LinkedList before adding the word, and the traversing through LinkedList is more expensive in terms of time as compared to array.
- The trie data structure is the quickest and most effective overall because it allows for each character of the user input to be compared to the characters in the trie.
- While we can analyse that growth of graphs for the three functions are quite linear between input sizes. We cannot conclude based on the timings collected that the empirical time complexity gained from the functions we implemented is the same as the theoretical time complexity of each function for each data structure (which in this case comes to be O(n)). The data collected was subject to conditions, aspects and features which was used, and therefore cannot completely justify the time complexity.