Empirical Analysis Report

We will set up the experiment to test our dictionary implementations in respect to the operations it can do. We will compare them across 4 different scenarios:

- Adding a word to the dictionary
- Deleting a word from the dictionary
- Searching for words in the dictionary
- Creating a list of auto-completed words

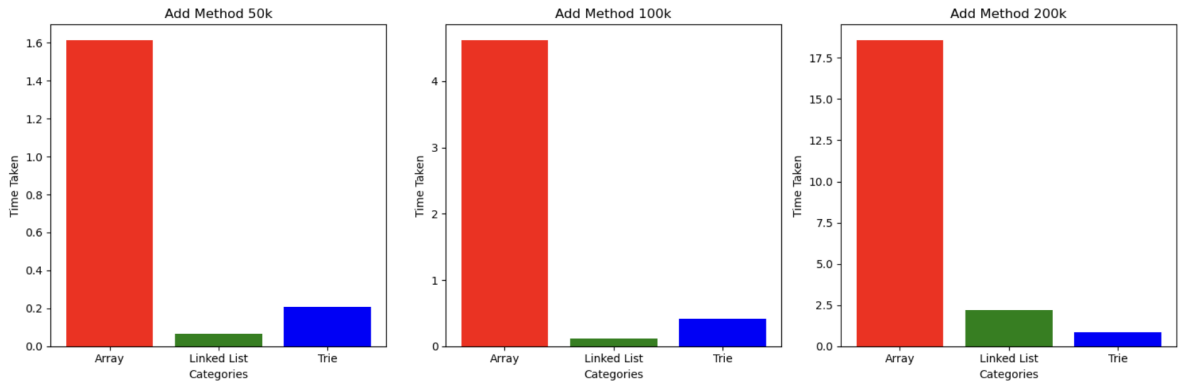
The goal of the experiment is to empirically test the efficiency of the operations as the datasets grow. To do so, we will perform our tests against dictionaries of differing sizes. We use 3 datasets of 50,000 words, 100,000 words, and 200,000 words to stress test the functions.

For generating our testing files, we will use a custom python script to generate test.in files 500 lines long using the method we will be testing. We can use this file in another custom python script which re-runs the code from **dictionary_file_based.py** 50 times and calculates the average time taken for the file to finish running. It is important to use a large average to help remove any outliers from the results to provide an accurate representation of the data.

For our evaluation environment, we will use an installation of python 3.11 on our local machine and use **time.time()** to measure the operations' efficiency.

Adding Word Frequencies

Adding a new word frequency works similarly across different dictionary implementations. First the dictionary must be checked if the word already exists then if it doesn't the word must be added. The major action being executed is the loop that is required to search the dictionaries so we would expect a time complexity of O(n) where n represents the dictionary's size.

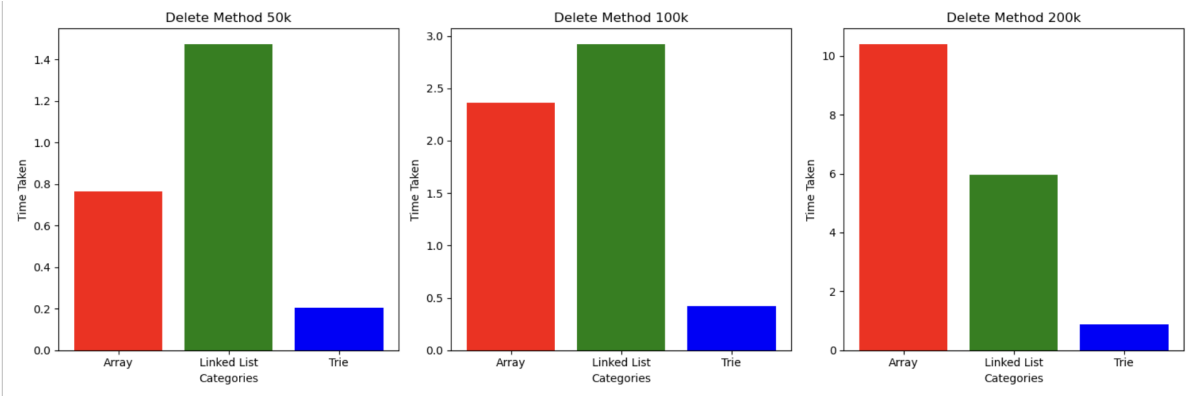


For the array based implementation we can observe that it is very inefficient due to its staggering time difference between it and the other two implementations. As the array dictionary increases, we can see that the time taken for the operations to finish is linear. This

supports the suspected time complexity of O(n). The linked list implementation shares a different analysis where the time taken to finish grows at a rapid pace. This differs from the expected O(n), and would suggest it has a time complexity of O(2^x) due to the exponential growth it shows. The final implementation is similar to the array-based dictionary by matching the expected time complexity of O(n). Overall, the trie implementation would be ideal for adding new word frequencies to a dictionary due to its faster speed at higher volumes of data. However, at lower data sizes the linked list implementation would be a faster solution to use but would lose speed exponentially as the data size grows.

Deleting Word Frequencies

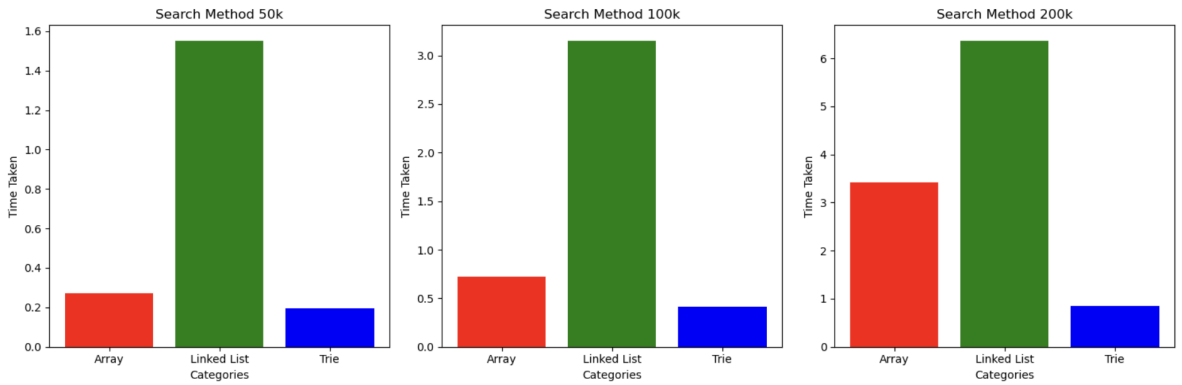
For deleting a word, the method is similar to the above adding method where all implementations first search for the word frequency and then delete it. Due to this the expected time complexity for the three implementations are O(n) where n is the size of the data in the dictionary.



For the array based implementation the time taken for the operations to complete increases quickly from a second or two all the way to ten seconds on average. This suggests a time complexity of O(n^2) which differs from the expected complexity of O(n). The linked list implementation however follows the expected O(n) by linearly increasing in size as the data set increases in size. Lastly, the trie-based dictionary also supports the expected time complexity with O(n) due to the time taken doubling with the sample size as it doubles. With all three implementations, the trie-based dictionary is faster than the rest and with a time complexity of O(n) it will continue to stay the fastest option for deleting words no matter the sample size of data.

Searching Word Frequencies

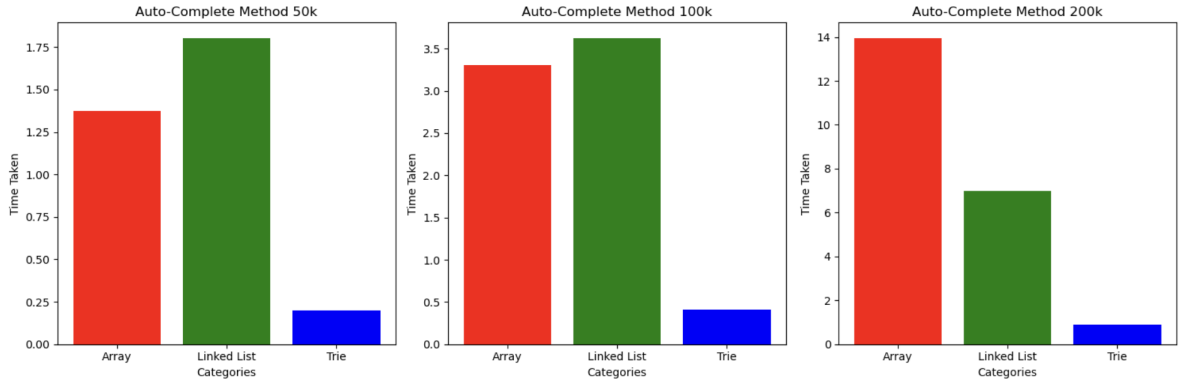
Searching for a word is a simple function that all three dictionaries implement very similarly. Using a loop they iterate through the values until they either find the word they are looking for or they reach the end. This would indicate a time complexity of O(n) demonstrating linear growth when compared to the dictionary's size.



For the array based dictionary the time taken for the operations to finish searching don't appear to be linearly increasing like predicted. Instead, the data suggests a time complexity of O(n) which would imply the time it takes is exponentially increasing. Differently however, the linked list implementation appears to follow the predicted time complexity of O(n), linearly increasing as the data size increases with it. The trie based dictionary follows similarly to the linked list and linearly increases the time taken in conjunction with the data size. Overall, while both the trie and array dictionaries start off the fastest, the trie-based implementation stays consistently the fastest with its O(n) time complexity while the array will continue to take longer the larger the sample size gets.

Auto-Completing Words

For the array and linked list implementations, both use very similar methods to find words starting with the prefix. They both first initialise an empty array. They then loop through the list and check if the word starts with the prefix. If it does, it gets added to the array. It is then sorted by frequency and all but the first 3 are deleted. The main action being taken is the loop through the list so we would expect a time complexity of O(n). The trie based dictionary is slightly different as the function traverses the trie to find the node corresponding to the last letter of the prefix, and then performs a depth first search on the subtree to find all the words that start with the prefix. They are then stored in an array, like the array and linked list implementations. this means that the trie implementation has a time complexity of O(n log n) …



We can see from the data that while the linked list may take longer when the dataset is small, the time taken for the array based implementation increases exponentially with O(n^2) time, which differs from the expected O(n) time, while the linked list implementation increases only linearly, which is the same as the expected O(n) time. The trie based

implementation is clearly the most efficient as it increases linearly, which although the same rate as the linked list implementation, the trie based implementation is quicker even on smaller values, and grows at a slower rate than the linked list.

Recommendations

Overall, the trie-based dictionary is the clear winner in terms of speed across all functions tested. It also benefits from the consistent O(n) time complexity allowing it to consistently be the fastest implementation across all data sizes. Linked-lists appear to be the slowest consistently when tested across every operation apart from the adding word frequency operation where it performs quite well. However, it is very consistent in the latter three functions allowing it to have consistent performance across dictionary sizes. The array implementation is the most inconsistent out of the three having two functions with a higher time complexity than the others. This is poor for the dictionary as the performance and efficiency get worse as the input size increases reducing expandability.