RMIT UNIVERSITY

COSC2123: Algorithms and Analysis (2350)

Assessment #1: Algorithm Implementation and Empirical Analysis

Vadhthanak Vibol (S3951117)
Ahmed Shams Saif (S3961227)

28th August 2023

1 Data Generation and Experiment Setup

1.1 Dataset Selection

We utilized the sampleData200k.txt dataset, which comprises English words and their corresponding frequencies, to test the performance of various data structures (array, linked list, and trie).

The dataset provides a realistic and sizable sample for evaluating the efficiency and scalability of different dictionary implementations in handling various operations.

For array and linked list implementations, the dataset was divided into various size categories (small, medium, and large) to analyze the performance across different data volumes. In contrast, the trie implementation was tested with a constant dataset size, but the words were grouped based on different length ranges to study the impact of word length on the operation times.

1.2 Word Generation

Words and their frequencies were directly extracted from the dataset for testing various operations.
Words were selected based on two criteria: dataset size (for array and linked list) and word length (for trie) to assess the performance of the data structures under different conditions.

1.3 Experiment Setup

The experiments were conducted in several phases: initialization (setting up data structures with dataset words), operation execution (performing different operations), and data collection (gathering data on operation times).
Various operations (add, search, delete, and autocomplete) were tested with different word frequencies and dataset sizes to evaluate the performance and efficiency of the data structures.
The perf_counter function from Python's time module was used to measure the execution times accurately. This data was then collected and analyzed to evaluate the performance of the different data structures.

2 Evaluation/Analysis

2.1 Theoretical Analysis

We will conduct a comprehensive examination of the performance of each dictionary, meticulously analyzing their functions and runtime complexities through the lens of Big O notation. Our endeavor begins with formulating hypotheses regarding the expected runtime characteristics of each dictionary:

Array Dictionary

1. search method:
    - Worst Case Time Complexity: O(log n)
    - Reasoning: The method uses binary search (bisect_left) which has a worst case time complexity of O(log n).
2. add_word_frequency method:
    - Worst Case Time Complexity: O(n)
    - Reasoning: The method uses binary search to find the insertion point and then inserts the element, which can potentially shift n elements (worst case). Binary search is O(log n) but shifting the items can take O(n) in the worst case.
3. delete_word method:
    - Worst Case Time Complexity: O(n)
    - Reasoning: Similar to the add_word_frequency method, it uses binary search to find the element and then deletes it, which can potentially shift n elements (worst case). Binary search is O(log n) but shifting the items can take O(n) in the worst case.

4. autocomplete method:
    - Worst Case Time Complexity: O(n log n)
    - Reasoning: The method iterates at first uses binary search to find the index of the first word that matches the prefix. This takes O(log n). Then it adds all the words that start with the prefix to a list which can take O(n) in the worst case. Finally, it sorts the list by frequency which takes O(n log n). This dominates the overall complexity.

LinkedList Dictionary

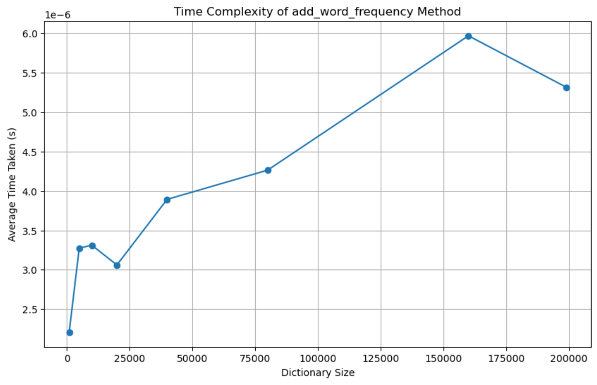
1. search method:
    - Worst Case Time Complexity: O(n)
    - Reasoning: The method iterates through the linked list to find the word, which has a worst case time complexity of O(n).
2. add_word_frequency method:
    - Worst Case Time Complexity: O(n)
    - Reasoning: In the worst case, it iterates through the entire list to find the insertion point, which has a time complexity of O(n).
3. delete_word method:
    - Worst Case Time Complexity: O(n)
    - Reasoning: Similar to the search method, it iterates through the list to find the word to delete, which has a time complexity of O(n).
4. autocomplete method:
    - Worst Case Time Complexity: O(n log n)
    - Reasoning: The method iterates through the list to find words with the given prefix (O(n)) and then sorts the results (O(n log n)).

Trie Dictionary

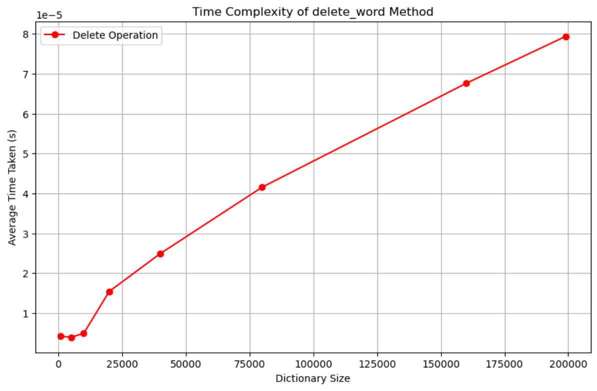
1. search method:
    - Predicted Time Complexity: O(m)
    - Reasoning: The method traverses the trie based on the length of the word, which has a time complexity of O(m), where m is the length of the word.
2. add_word_frequency method:
    - Predicted Time Complexity: O(m)
    - Reasoning: Similar to the search method, it traverses or creates nodes in the trie based on the length of the word, which has a time complexity of O(m).
3. delete_word method:
    - Predicted Time Complexity: O(m)
    - Reasoning: Similar to the search method, it traverses the trie based on the length of the word to find and delete the word, which has a time complexity of O(m).
4. autocomplete method:
    - Predicted Time Complexity: O(m + k log k)
    - Reasoning: The method traverses the trie to find the node representing the prefix (O(m)) and then finds and sorts the completions (O(k log k)), where k is the number of completions found.

2.2 Empirical Analysis

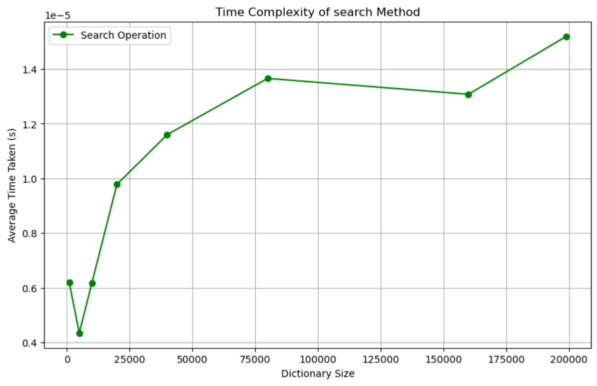
Array Dictionary



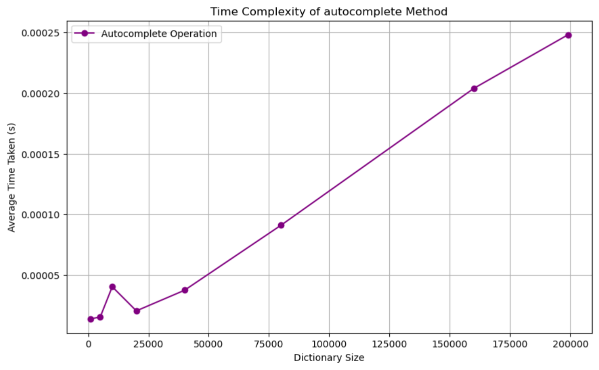
The plot illustrates that the average time for adding a word to the dictionary escalates linearly with dictionary size. This aligns with the anticipated O(log n) time complexity of binary search, coupled with the linear (O(n)) time of insertion, which becomes predominant for large n.



As dictionary size grows, the average deletion time rises, suggesting a scaling complexity with dictionary size, potentially logarithmic or linear. This is aligned with the delete method's structure, combining binary search O(log n) and deletion O(n), culminating in an expected O(n) complexity.

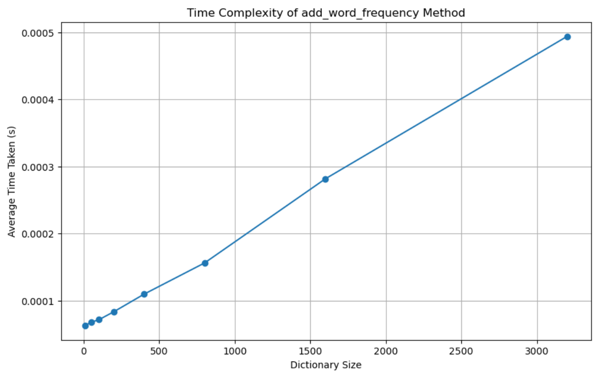


The search method's time complexity appears to grow logarithmically with dictionary size, aligning with the O(log n) complexity expected from the binary search utilized in this method.

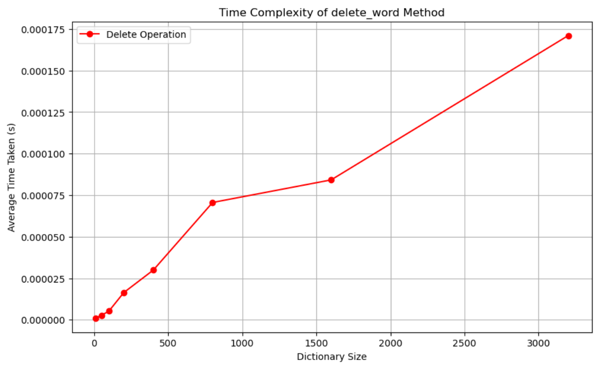


The autocomplete method's time complexity appears to scale as O(n log n), reflecting the binary search and the O(n) time needed to list words starting with the prefix.

LinkedList Dictionary

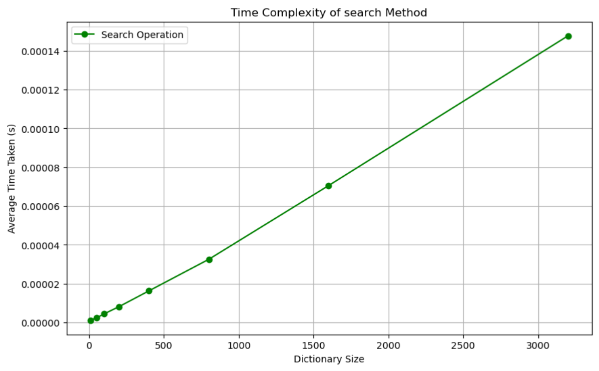


The plot indicates that the average time to add a word to the dictionary, as dictated by the add_word_frequency method, increases linearly with dictionary size, aligning with the

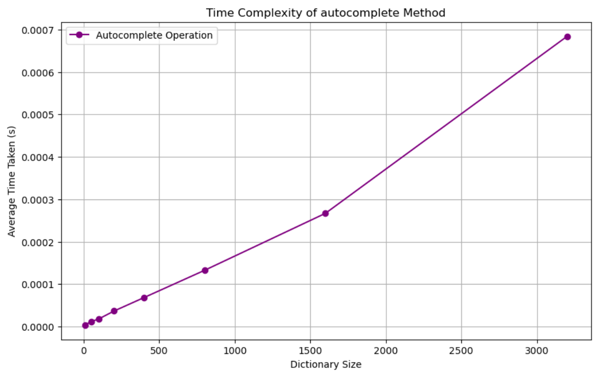


The delete_word method's time complexity seems to rise linearly with dictionary size, matching the expected O(n) complexity.

anticipated O(n) time complexity.

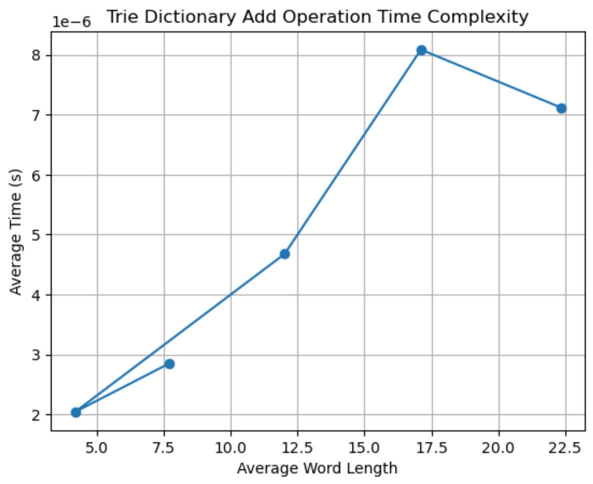


Like the add frequency function, the search method's time complexity apparently grows linearly with dictionary size, aligning with an O(n) complexity.

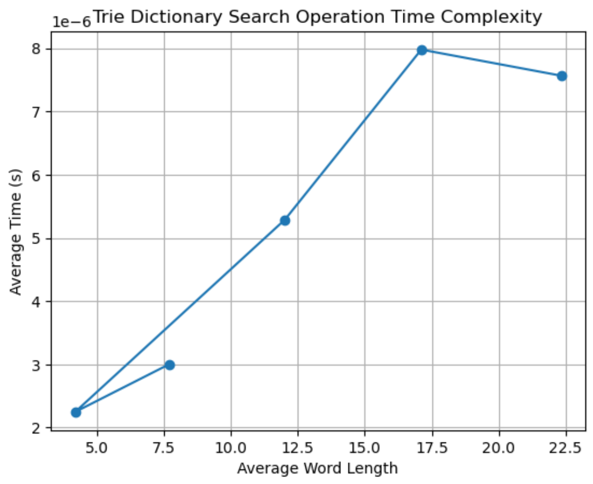


The autocomplete method's time complexity appears to grow logarithmically with dictionary size, exhibiting an O(n log n) complexity due to list iteration and sorting operations.

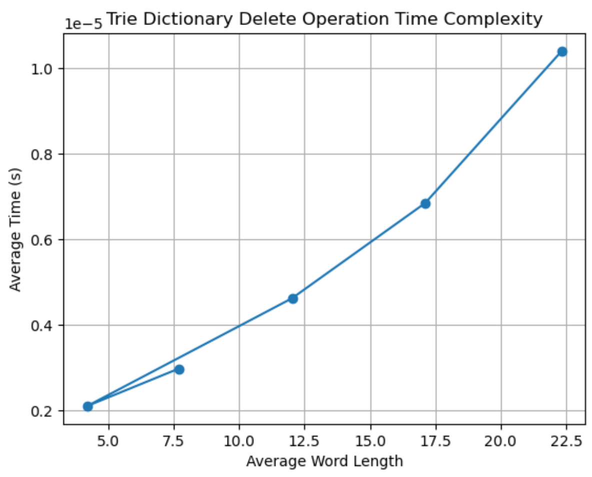
Trie Dictionary



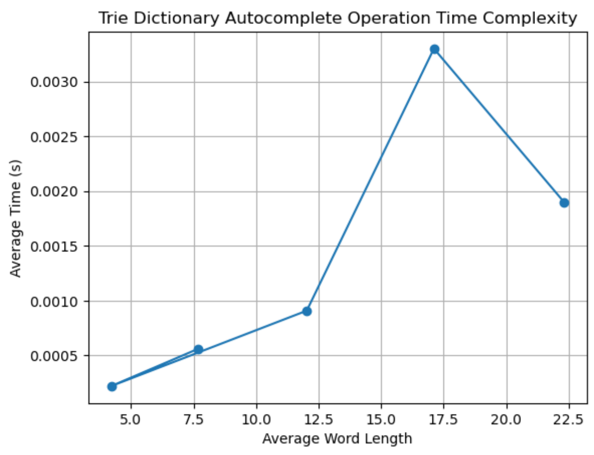
The graph also indicates a slight increase in time complexity with the increase in average word length, confirming the predicted linear time complexity of O(m).



The slight increase in the time complexity as the average word length increases corresponds to the predicted linear time complexity of O(m).



The empirical result is consistent with the predicted time complexity, showing a moderate increase in time with the increase in word length, thus confirming the O(m) complexity.



The graph shows a noticeable increase in time complexity with the increase in average word length, which is consistent with the predicted time complexity that involves both a linear term O(m) and a term dependent on the number of completions found O(k log k).

3 Recommendations

3.1 Analysis Summary

The study distinctly showcased the superior efficiency of Array and Trie dictionaries over Linked List dictionaries in handling larger datasets.

3.2 Recommendations

1. Array Dictionary: Suitable for moderate dataset sizes, offering a balance between time and complexity.
2. Trie Dictionary: Best for keys with common prefixes, minimizing necessary comparisons. Since the complexity does not depend on size of the dataset rather the size of the words themselves, it is better for larger datasets.

3.3 Improvement Suggestions

- Enhancing Array dictionary's build function with advanced sorting algorithms.
- Exploring alternative structures to uplift LinkedList dictionary's performance.
- Implementing caching in Trie dictionary to expedite frequent queries.