**WORD SEARCH AND RETRIEVAL ALGORITHM**



**BTech/ II Year CSE/ IV Semester**

**19CSE212/ Data Structures and Algorithms**

**Case Study Report**

**Group - 17**

**TEAM MEMBERS:**

|  |  |
| --- | --- |
| **NAME:** | **ROLL NUMBER:** |
| M Jaswanth Reddy | CB.EN.U4CSE21237 |
| Mathamgi Nair | CB.EN.U4CSE21238 |

**Department of Computer Science and Engineering**

**Amrita School of Computing, Coimbatore**

**Amrita Vishwa Vidyapeetham**

**2022 – 2023 Even Semester**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| SL.NO | TITLE | PG.  NO |
| Chapter 1.  1.1  1.2 | Introduction  Overview  Significance | 3  3  4 |
| Chapter 2.  2.1  2.2  2.3  2.4 | Implementation  Trie Implementation  AVL Tree Implementation  Integration an Interplay  Design Choices and Trade-Offs | 5  5  5  5  6 |
| Chapter 3. | Practical Applications | 8 |
| Chapter 4.  4.1  4.2  4.3 | Performance Analysis  Time Complexity  Space Complexity  Performance Comparison | 9  9  9  10 |
| Chapter 5.  5.1  5.2  5.3  5.4  5.5 | Discussion  Practicability and Effectiveness  Limitations and Challenges  Future Improvements  Experimental Evaluation  Presentation and Interpretation of Results | 12  12  12  13  13  14 |
| Chapter 6. | Conclusion | 15 |
| Chapter 7. | References | 17 |

**CHAPTER 1: INTRODUCTION:**

Hybrid data structures, also known as combination data structures, are data structures that combine the characteristics and functionality of multiple traditional data structures. They aim to leverage the strengths of different data structures to achieve improved performance or provide specialized operations for specific use cases.

Hybrid data structures are designed to address specific requirements or optimize certain operations that may be inefficient with a single data structure. They often combine the strengths of two or more data structures, such as arrays, linked lists, trees, or hash tables, while minimizing their weaknesses.

Hybrid data structures are data structures that combine the properties of two or more different data structures. This can be done to improve the performance or functionality of the data structure.

**1.1: Overview:**

**Objective and Practical Applications:**

In this case study, we have chosen to implement a hybrid data structure that combines the functionalities and properties of Trie and AVL Tree: and implemented that into constructing a Dictionary wherein large datasets of words are efficiently stored and searched up to produce accurate results. The case study involves creation of an English dictionary wherein we store words into the dictionary and provides a fast look-up for the words in the same. The user can also give a prefix search and retrieve all the words that is in sorted order.

The project aims to demonstrate the practical applications of hybrid data structures in creation and search of words in an English Dictionary and retrieval of words from the same. How the hybrid data structure of our choice will enhance the functionalities, efficiency and performance of the chosen scenario is also demonstrated in this project.

**Analysis of Space Complexity and Time Complexity:**

The algorithms used in the logic and implementation are scrutinized and examined for their space and time complexity. This in turn is believed to enhance our understanding of the hybrid data structure's working, efficiency, benefits and challenges, thus optimizing the algorithms and their usage.

**1.2: Significance of Hybrid Data Structures:**

Hybrid data structures are very significant when it comes to implementing real life scenarios effectively, since they enhance the functionalities and performance of the chosen scenario. With regard to Word Search Algorithm, the hybrid data structures used ensures that the efficiency in search operations, space optimization and flexibility in the searching criteria are met with. It also enables us to appraoch the bottlenecks of our scenario wherein we must accommodate large datasets like a dictionary with its thousands of words and carry out storing and searching among them in an optimal way, thereby improving the scalability and performance by large, and also enable fast, memory-optimized and accurate output.

**Significance of Trie:**

The Trie data structure was chosen for its ability to efficiently store and search words in the English dictionary.

**Prefix search:** Trie works well in situations where we need to look for words based on a prefix that is frequently used.This structure makes it simple to quickly search for all words that contain a particular prefix.

**Space efficiency:** Tries are memory-efficient when it comes to storing large datasets of words. Since words with common prefixes share common nodes in the Trie, memory is effectively utilized by reusing existing nodes.

**Fast retrieval:** Trie offers quick word retrieval from the lexicon.It is very effective because the time complexity of searching a word in a Trie is inversely proportional to the word's length.

**Significance of AVL Tree’s:**

In the English dictionary scenario, the AVL Tree data structure was employed to maintain a balanced tree and improve the effectiveness of operations.

AVL, a balanced tree Self-balancing binary search trees are what trees are. They make sure that any node's left and right sub trees are separated by no more than one height. The balanced tree structure that this attribute helps to maintain leads to effective search, insertion, and deletion operations.

**Sorted order:** AVL Trees inherently maintain the order of elements, allowing for easy retrieval of words in sorted order. This is beneficial for scenarios where users need to retrieve words alphabetically or in a specific order from the dictionary.

**CHAPTER 2: IMPLEMENTATION**

On the implementation front of our topic Word Search Algorithm, the hybrid data structure that we used, which is an AVL Tree and Trie combined, consists of 2 main components, the Trie data structure and the AVL Tree. The functionality of the same two are described as follows:

**2.1: Trie Implementation:**

Achieved with the help of the Dictionary class, the Trie data structure is used to store the words and enable operations such as word insertion, word search, and prefix search. The Dictionary class implements the trie using a nested TrieNode class, which contains a dictionary of child nodes representing the characters in the words. The trie allows for efficient retrieval of words with a given prefix also.

**2.2: AVL Tree Implementation:**

This is achieved with the help of the AVLTree class. The AVL tree is used to maintain the words in sorted order. The AVLTree class implements the AVL tree using a nested AVLNode class, which contains the word as the key. The AVL tree ensures that the words are stored in a balanced manner, allowing for efficient traversal and retrieval of words in sorted order. The DictionaryWithTrieAndAVL class combines the functionality of the trie and AVL tree by maintaining instances of both data structures. When a word is inserted, it is added to both the trie and the AVL tree.

**2.3: Integration and Interplay:**

The integration and interplay between the Trie and AVL Tree in the hybrid data structure involves leveraging the strengths of each data structure to enhance the overall functionality and performance of the dictionary.

The integration of the Trie and AVL Tree allows for efficient word insertion, search, and retrieval operations. The Trie handles prefix-based searches and supports quick word lookups. It is also responsible for effectively storing the words.

The AVL Tree provides the functionality for maintaining the frequency information, for enabling retrieval of the words in sorted order. it makes sure that the words are stored in the form of a balanced binary search tree, which is what makes retrieval easy. By combining the strengths of both data structures, the hybrid data structure achieves efficient word search capabilities while maintaining sorted word retrieval.

The **DictionaryWithTrieAndAVL** class integrates the Trie and AVL Tree by combining them into a single hybrid data structure.

Words are inserted into both the Trie and AVL Tree simultaneously to maintain consistency between the two data structures. The interplay between the two combined data structures ensures efficient word search and retrieval in the use case, all achieved in accurate and optimal ways.

**2.4: Design Choices and Trade-Offs:**

During the implementation phase, various design choices were considered which led to certain trade-offs but functionalities and performance efficiency were ensured.

**Choice of Data Structures:**

1. **Trie Design Choices:**

* It was chosen as the primary data structure for storing and searching words based on prefixes. Trie was chosen for this purpose because it would provide fast prefix-based search operations, making it suitable for word search functionalities.
* Each TrieNode has a dictionary-like data structure (or, children) to store the child nodes associated with characters.
* The is\_word attribute in TrieNode is used to mark the end of a word.

1. **AVL Tree Design Choices:**

* AVL Tree was also chosen to implement this scenario so as to maintain frequency information and to enable sorted word retrieval.
* In our use case, Each AVLNode contains a key (or, word) and maintains references to left and right child nodes.
* The height of each AVLNode is maintained to ensure the tree remains balanced, which unturn would ensure that the words are all in a sorted order.

**Trade-Offs:**

* **Insertion and Search Trade-offs:**
* Insertion in the hybrid data structure involves inserting a word into both the Trie and AVL Tree. This ensures that words are efficiently stored for fast retrieval and sorted word retrieval.
* The trade-off is the additional memory required to store the words in both data structures. However, this trade-off enables efficient search and retrieval operations.
* **Prefix Search Trade-offs:**
* Prefix-based search (starts\_with) is implemented using the Trie. It efficiently traverses the Trie based on the given prefix and collects all words with that prefix.
* The trade-off is that the AVL Tree is not involved in prefix search operations. However, since the Trie is optimized for prefix searches, this trade-off is reasonable.
* **Sorted Word Retrieval Trade-offs:**
* The AVL Tree is utilized for retrieving all words in sorted order.
* The trade-off is that maintaining the AVL Tree and performing an inorder traversal may require additional time and memory compared to a regular Trie. However, it enables efficient retrieval of words in sorted order.

**CHAPTER 3: PRACTICAL APPLICATIONS**

The hybrid data structure that we used, the one which combines an AVL Tree and Trie, has several practical applications in real life scenarios:

1. **Auto Complete and Spell-Checking Systems:**

* Hybrid data structures can be used in auto-complete systems, where the Trie efficiently stores and retrieves words based on prefixes.
* The AVL Tree can be used to track word frequencies and provide suggestions based on the most frequently used words.

1. **Text Editors and Word Processors:**

* Hybrid data structures can be employed in text editors and word processors to implement features like spell-checking, word suggestions, and word frequency analysis.
* The Trie can be used for efficient word search, auto-correction, and suggesting alternative words.
* The AVL Tree can maintain a sorted list of words, allowing for the presentation of word suggestions in alphabetical or frequency-based order.

1. **Word Games and Puzzles:**

* Hybrid data structures can be applied in word games and puzzles to facilitate word lookups, verification, and scoring.
* The Trie component can validate words and check if they exist in a given dictionary.
* The AVL Tree can track word frequencies to assign scores or rankings based on word usage.

**CHAPTER 4: PERFORMANCE ANALYSIS**

In this case study, the major operations that were dealt with were insertion, searching the words in the AVL Tree, Prefix-Checker and sorting of the words in the dictionary. In terms of performance analysis, we shall look into the time complexity and space complexity of the use case, with special regard to the operations insertion and searching.

**4.1: Time Complexity:**

* **Insertion:**
* Inserting a word into the dictionary would mean insertion of the certain word via two operations: inserting into the Trie and inserting into the AVL Tree.
* Inserting into the Trie takes O(L) time, where L would be the length of the word being inserted.
* Inserting into the AVL Tree takes O(log N) time, where N would be the number of nodes in the tree.
* Therefore, the overall time complexity for insertion is O(L + log N).
* **Searching:**
* Searching for a word in the dictionary deals with traversing through the Trie and checking the AVL Tree.
* Searching in the Trie takes O(L) time, where L would be the length of the word being searched.
* Checking the AVL Tree takes O(log N) time, where N would be the number of nodes in the tree.
* Therefore, the overall time complexity for search is O(L + log N)
* **Prefix-Check:**
* Finding all words with a given prefix is yet another operation that the word search algorithm carries out, and it involves traversing the Trie to find the prefix and then collecting the relevant words using depth-first search.
* Finding the prefix in the Trie takes O(P) time, where P is the length of the prefix.
* Collecting words using depth-first search takes O(W) time, where W is the number of words with the given prefix.
* Therefore, the overall time complexity for starts\_with is O(P + W)
* **Sorting of the Words:**
* Getting all words in sorted order involves performing an inorder traversal of the AVL Tree.
* Inorder traversal of an AVL Tree with N nodes takes O(N) time.
* Therefore, the time complexity for obtaining all words in sorted order is O(N).

**4.2: Space Complexity:**

* **Insertion and Search:**
* The space complexity for inserting and searching in the hybrid data structure is determined by the space required by the Trie and AVL Tree.
* The Trie requires additional space to store the characters of the words and the is\_word flags.
* The AVL Tree requires space for storing the AVL nodes.
* The space complexity for both insertion and search operations is proportional to the total number of characters in all words, i.e., O(S), where S is the total size of all words.
* **Prefix-Checker and Sorted Words:**
* The space complexity for starts\_with and sorted\_words operations depends on the space required by the words collected or returned.
* For starts\_with, the space complexity is proportional to the number of words with the given prefix, i.e., O(W)
* For sorted\_words, the space complexity is proportional to the total number of words, i.e., O(W)

**4.3: Performance Comparison:**

The hybrid data structure used for the Word Search Algorithm in a Dictionary involves a combination of a Trie and an AVL Tree and the combined functionalities of the same. The combination of these data structures or the usage of hybrid data structure in implementing this use case would enhance the performance, accuracy in output and efficiency, among other improved parameters.

As opposed to the benefits that usage of either a single Trie or AVL Tree would have in a word search algorithm, the following are the advantages of using a hybrid data structure that combines a Trie and AVL Tree instead:

* **Efficient Prefix Search**: Tries excel at prefix search operations. By using a Trie, you can efficiently find all words that start with a given prefix. This operation has a time complexity of O(k), where k is the length of the prefix, making it very fast.
* **Sorted Order:** AVL Trees are self-balancing binary search trees that naturally maintain elements in sorted order. This makes it easy to obtain words in sorted order without the need for additional sorting operations. AVL Trees provide efficient in-order traversal, which allows you to retrieve the words in sorted order with a time complexity of O(N), where N is the total number of words.
* **Space Efficiency:** Tries are efficient in terms of memory usage for storing words. They take advantage of shared prefixes among words to save space. Compared to storing words directly in an AVL Tree, a Trie can reduce memory consumption, especially when dealing with large datasets.
* **Flexibility:** The hybrid data structure combining Trie and AVL Tree combines the strengths of both data structures. It allows for efficient prefix search using the Trie while maintaining the ability to retrieve words in sorted order using the AVL Tree. This flexibility is useful in various scenarios, such as autocomplete functionality or dictionary applications.
* **Balanced Tree Properties**: AVL Trees ensure that the tree remains balanced, preventing the worst-case scenarios of highly unbalanced trees. This guarantees efficient search and insertion operations with a logarithmic time complexity on average.

By combining the strengths of Tries and AVL Trees in a hybrid data structure, we’re able to achieve efficient prefix search, sorted order retrieval, space efficiency, and balanced tree properties, making it a favorable choice for word search algorithms compared to using only a Trie or only an AVL Tree.

**CHAPTER 5: DISCUSSION**

**5.1: Practicability and Effectiveness:**

The hybrid data structure that we have implemented, word search algorithm in a dictionary, can be both practical and effective for dictionary applications. With respect to the practicability, the trie component of this hybrid data structure enables efficient and excellent prefix search, a feature that is considered very crucial in dictionary applications or for autocomplete functionality, something that we all use on a regular basis. Also, the AVL Tree components which sorts the words all in sorted order makes it easy to retrieve words in alphabetical order, again something very vital for dictionary applications or in search engine applications for words. Corresponding to criteria like memory efficiency, tries are efficient even when dealing with large datasets as in a dictionary’s case.

In terms of effectiveness, both trie and AVL Tree components in this use case ensure fast search operations, especially with AVL Trees offering time complexity in logarithmic terms. Also, efficient insertions are a sure feature. The flexibility offered by this hybrid data structure for the chosen use case is high, considering the combined strengths of Tries and AVL Trees, thereby making it a sought-after choice for handling various needs of dictionary applications and other such practical applications.

**5.2: Limitations, Challenges:**

Although the implemented solution for the chosen scenario offers many advantages and benefits with respect to criteria like enhanced searching and insertion and other operations, it also has certain limitations and bottlenecks. Those are as follows:

1. It may take up more memory when compared to usage of single data structures like say, an AVL Tree or a Trie, since both these data structures are supposed to be concurrently maintained and used.
2. Complexity in code and understanding, since it accommodates 2 different data structures. This also has a chance to increase the number of errors that could appear in the code, thereby making processes like debugging quite complex.
3. This use case implementation using the chosen hybrid data structure demands high maintenance to ensure correctness and accuracy in output.

**5.3: Future Improvements:**

Due to time constraints and other factors, the implementation of words search algorithm using the combination of trie and AVL Tree data structures couldn’t meet all the edge case necessities but however the following could be considered in terms of scope for future improvements for the same:

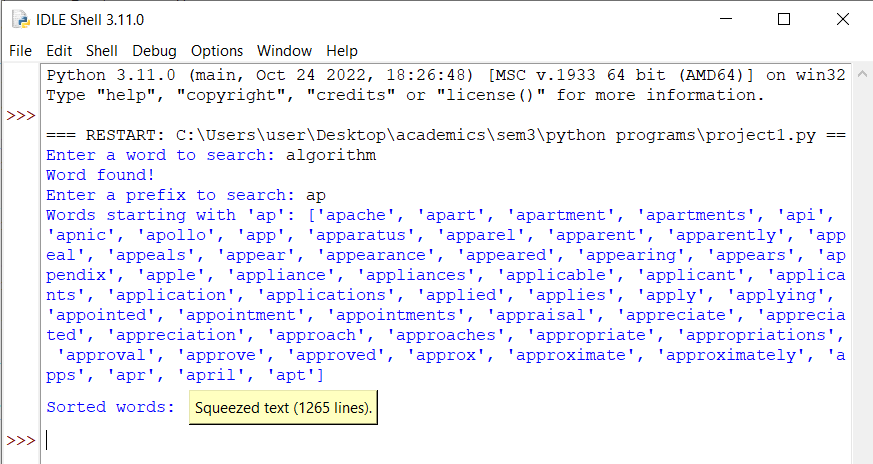
1. **Space optimization**: although this cannot be generalized much as it by large depends on the requirements and characteristics of the dictionary, it could be seen to it that space consumption could be constantly monitored and required optimization techniques can be taken up.
2. **Error handling and robustness:** implementing exception handling mechanisms would ensure that the code implemented provides accurate results, probably even in all the boundary cases
3. **Developing the project for a combination of words:** the current version that was developed works for inserting and searching a single word. In terms of future improvements, this could be improved to function for a combination of words

**5.4: Experimental Evaluation:**

In terms of the datasets used, we imported a text file which has near to 10,000 words: thus simulating as a dictionary. The file has been invoked in the following way: a variable file\_path is assigned a string value which would allot it to the dataset that we used, word1.txt file. After figuring out the location of the word1 file, the function open( ) is used to open the dataset file in read mode. file.read( ) method reads the content in string format and split( ) is to split the string involved into a list of separate words.

**5.5: Presentation and Interpretation of Results Obtained:**

Upon compilation and execution of the code, so the is the results obtained:



The squeezed text icon if selected would display all the sorted words in the dictionary alphabetical order.

**CHAPTER 6: CONCLUSION**

The objective of this project was to design and implement a hybrid data structure for a Word Search Algorithm in a Dictionary. A combination of Trie and AVL Tree is used for achieving the same. The project deals with the user inserting a word and the implemented code would show if the inputted word is found in the dictionary or not. For the dictionary simulation, a text file consisting of thousands of English words is imported into the code. Also, the user could input a word prefix and the code would search through the dictionary and display all the words in the dictionary which has that certain prefix in sorted order.

In this case study report, we had included topic overview, major findings, implementation details, practical applications, performance criteria, limitations, bottlenecks and future improvements possible for the chosen use case. With respect to Word Search Algorithm performed in a Dictionary, the chosen hybrid data structure proves to be highly efficient in terms of performance and accuracy in output display. After the integration of data structures Trie and AVL Tree for our use case, we found how arriving at the optimized and accurate solution got efficient when our hybrid data structure is used, as opposed to otherwise.

The time and space complexity of the chosen scenario was also analyzed for the operations that were involved, namely insertion of word, word search, prefix-check and sorted words. The time complexity of the word insertion and word search operations were obtained to be O(L + log N), L and N being the length of the input word and number of nodes in the tree respectively. Time complexity of prefix check is O(P + W), P being length of prefix and W being the number of words in the prefix. That of sorted words is O(N).

In terms of space complexity, that of insertion and word-search operations are O(S), where S is the total size of all words. That of prefix-check and sorted words though is O(W), where W is the number of words concerned.

We also gained certain insights from the performance analysis and comparison on the importance of choosing the suitable data structures for implementing the chosen use case in an optimized way, which would thereby produce accurate output. The project focuses on optimizing insertion and searching of words in the dictionary which by itself is a large dataset, and how to effectively sort the words alphabetically. We also delved into the possibilities of future improvements for implementing word search algorithm scenario that was chosen and also analyzed the limitations and bottlenecks of the same.

In conclusion, the project successfully implemented a hybrid data structure for performing a word search and retrieval algorithm for an english dictionary. The practicability, effectiveness, performance, trade-offs and challenges were all approached. The insights gained from the completion of this project contributes to developing a greater understanding on the concept of hybrid data structures and their application in solving problems in real life scenarios. This project also bears scope for further research and improvements in the use case of word search and retrieval algorithm and similar applications.

**Github Repository URL:** <https://github.com/Jaswanth0304/Word-Search-and-Retrieval-using-hybdrid-data-structures>

**REFERENCES:**

[1] “Data structures and algorithms in Python”. John Wiley & Sons Ltd;

[2] “An introduction to data structures with applications. “Second Edition, McGraw-Hill;

[3] <https://www.geeksforgeeks.org/trie-insert-and-search/> -- for ref. on Trie operations

[4] <https://www.programiz.com/dsa/avl-tree> -- for ref. on AVL Tree operations