**DSA CASE STUDY: TERNARY SEARCH TRIE**

**GROUP: ALGOHOLICS**

**TEAM** **MEMBERS :**

Abhisheakraj O – CB.EN.U4CSE21402

Nithin G - CB.EN.U4CSE21443

S Amruthiyu - CB.EN.U4CSE21449

Srutija B - CB.EN.U4CSE21460

T Bhargavi - CB.EN.U4CSE21462

**INTRODUCTION:**

Hybrid data structures have emerged as a key concept in solving complex problems efficiently by combining the advantages of multiple data structures. They offer a versatile approach to optimize performance and strike a balance between time and space complexity. By harnessing the strengths of different data structures, hybrid data structures have become instrumental in tackling challenging problem scenarios effectively.

The project objective is to design and implement a hybrid data structure that efficiently addresses specific problem requirements. By analyzing existing data structures and integrating multiple components, the project aims to create a hybrid structure that optimizes performance while minimizing drawbacks. The goal is to provide a tailored solution that achieves a superior trade-off between time and space complexity, offering an efficient and effective approach for solving complex problems.

Hybrid data structures have practical applications in various domains, offering versatility and efficiency. In computer graphics and gaming, a hybrid structure combining a spatial data structure (e.g., quadtree, octree) with hashing is ideal for collision detection and spatial indexing. In database systems, combining B+ trees with hash indexes as hybrid index structures greatly improves query performance. These examples highlight the effectiveness of hybrid data structures in addressing specific challenges and enhancing the performance of applications in different fields.

Analyzing the time and space complexity of a hybrid data structure is crucial for evaluating its efficiency. This involves evaluating the worst-case, average-case, and best-case time complexities for important operations such as insertion, deletion, search, and traversal. Additionally, analyzing the space complexity helps determine the memory requirements of the hybrid structure. Through a comprehensive analysis of these factors, valuable insights into the efficiency of the structure are gained, which in turn aids in optimizing its performance for solving complex problems efficiently.

**OVERVIEW OF HYBRID DATA STRUCTURE:**

The Ternary Search Trie (TST) is a hybrid data structure that combines the best of both a trie and a binary search tree Where, TRIE is a tree-like data structure that stores data in a way that allows for efficient lookups and BINARY SEARCH TREE is a tree-like data structure that stores data in a way that allows for efficient insertions and deletions.It offers a powerful solution for storing and retrieving data efficiently while optimizing performance.

The Ternary Search Trie (TST) is a hybrid data structure that combines the characteristics of a trie and a binary search tree. It leverages the advantages of each component to efficiently handle storing and searching strings. In the TST, each node represents a character, and the branching is based on characters. Unlike a traditional trie, the TST introduces three branches from each node, allowing for efficient searching and insertion operations by utilizing binary search principles.

Additionally, the TST includes links to left and right children, similar to a binary search tree. This enables ordered traversal, facilitating efficient searching and retrieval of data based on the character ordering. The TST's hybrid composition makes it well-suited for various string-based operations like prefix searching, wildcard matching, and autocomplete functionality.

The motivations behind using a hybrid data structure like the TST are to maximize performance and efficiency in solving specific problems. By combining different data structures, the TST aims to achieve an optimal trade-off between time and space complexity. It takes advantage of the trie structure for fast searching and insertion of strings, leveraging its ability to handle prefixes efficiently. At the same time, the inclusion of a binary search tree component enables efficient ordered traversal and retrieval of data. This hybrid approach allows the TST to excel in scenarios where both fast string operations and ordered traversal are essential, making it a powerful choice for applications that require versatile and efficient data storage and retrieval capabilities.

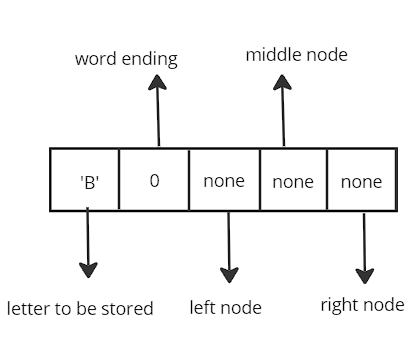
The Ternary Search Trie (TST) offers a multitude of advantages as a hybrid data structure. It excels in efficient string-based operations, leveraging the strengths of both trie and binary search tree components. The trie aspect allows for fast searching and insertion of strings, while the binary search tree component facilitates efficient ordered traversal and retrieval. This combination optimizes time complexity and achieves a balanced trade-off between time and space complexity. Additionally, the TST's capability to handle large datasets efficiently further enhances its effectiveness in solving complex problems that demand fast and versatile data storage and retrieval. Overall, the TST proves to be a powerful and efficient tool for a wide range of applications.

**IMPLEMENTATION DETAILS:**

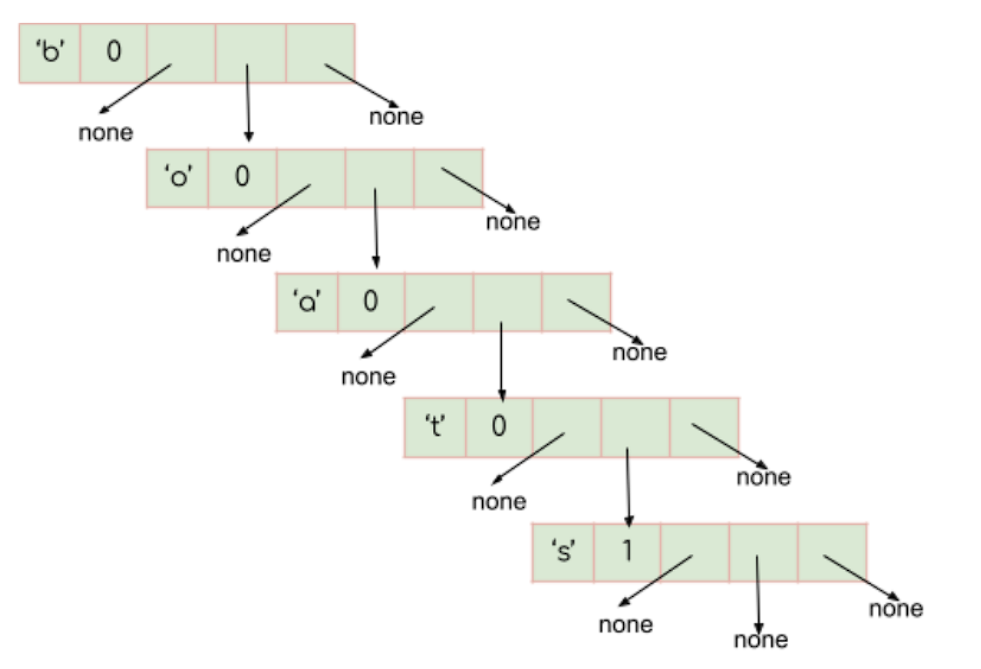
The implementation of a Ternary Search Trie (TST) involves integrating and interplaying the constituent data structures, tries and binary search trees (BSTs).

Here is the overview of the components and functions of TST:

Node Definition: The TST is composed of nodes, each representing a character and containing pointers to its left, middle, and right child nodes. Additionally, each node may store a value associated with the character if it represents the end of a key.



Insertion: To insert a key into a ternary search tree (TST), we start from the root node and compare the current character of the key with the character of the current node. If the current character is less than the character of the current node, we move to the left child of the current node. If the current character is greater than the character of the current node, we move to the right child of the current node. If the current character is equal to the character of the current node, we move to the middle child of the current node. This process continues until we reach the end of the key or we find a node with the same key. If we reach the end of the key, we create a new node with the key and insert it into the tree. If we find a node with the same key, we do nothing.



Deletion: Deleting a key from a ternary search tree (TST) involves finding the node corresponding to the key and removing it. If the deleted node has children, we adjust the pointers accordingly to maintain the tree structure. This process utilizes the principles of both tries and binary search trees (BSTs) to handle deletion efficiently.

Here are the steps involved in deleting a key from a TST:

Find the node corresponding to the key.

If the node has no children, simply remove it.

If the node has one child, connect the child to the parent of the deleted node.

If the node has two children, find the successor of the deleted node. The successor is the node with the smallest key that is greater than the key of the deleted node.

Remove the successor node and connect its children to the parent of the deleted node.

Design Choices and Trade-offs:

Ternary search trees (TSTs) are a type of trie that combines the best of tries and binary search trees. TSTs have several advantages and disadvantages, including:

Advantages:

Efficient search and insertion: TSTs can search and insert elements in O(log n) time, where n is the number of elements in the tree. This is comparable to the performance of binary search trees, but better than tries.

Compact representation: TSTs use three pointers per node (left, middle, right), which allows for a more compact representation than general tries. This can be important for applications where space is limited, such as embedded systems.

Simple design: TSTs are relatively easy to understand and implement, compared to other data structures such as balanced binary search trees.

Disadvantages:

Memory overhead: TSTs have a higher memory overhead than tries, due to the use of three pointers per node. This can be a significant disadvantage for applications where memory is limited.

Slower search times: TSTs may have slightly slower search times than hash tables, due to the need to compare characters during traversal.

Not suitable for all applications: TSTs are not suitable for all applications. For example, they are not as efficient as hash tables for applications where lookup speed is critical.

Overall, TSTs are a versatile data structure that can be used for a variety of applications. They offer a good balance of performance, space efficiency, and simplicity.

**GitHub repository link:**

<https://github.com/amruthiyu/ternary_search_trie>

**PRACTICAL APPLICATIONS**:

The Ternary Search Trie (TST) is a data structure that combines the best of tries and binary search trees. TSTs are efficient for string search and retrieval, and they have a number of practical applications in various domains.

Here are a few examples of how TSTs are used in practice:

Spell checking and autocompletion: TSTs are commonly used in spell checking and autocompletion systems. The structure allows for quick lookup and suggestions based on partial or misspelled words. The TST efficiently stores and searches a large dictionary of words, providing accurate and real-time suggestions as users type.

Information retrieval and text indexing: TSTs are used in information retrieval systems and text indexing. They can efficiently store and search large collections of documents or web pages. TSTs enable keyword-based searches, enabling fast retrieval of relevant documents.

Auto-suggest and predictive text: TSTs are widely used in applications that provide auto-suggest or predictive text functionality. By storing a dictionary of frequently used words or phrases, TSTs can quickly generate relevant suggestions as users type.

Symbol table management in compilers: TSTs are suitable for symbol table management in compilers or interpreters. They efficiently store and retrieve identifiers, variable names, or function names in programming languages. TSTs allow for fast lookup and resolution of symbols, making them valuable for tasks such as type checking, scoping, and name resolution.

The combination of trie and binary search tree properties in the TST enables efficient operations for these applications. The trie-like structure allows for fast prefix matching and traversal, making it ideal for handling string-based operations. The binary search tree properties facilitate efficient comparisons and traversal based on character values, ensuring quick access to the desired nodes. This combination results in optimized search times, reduced time complexity, and efficient retrieval of relevant information in various applications.

**PERFORMANCE ANALYSIS:**

**Time Complexity of Key Operations in TST**

* Insertion: The time complexity of inserting a key into a TST is O(log n), where n is the number of nodes in the TST. This is because during insertion, we compare characters and traverse the tree, similar to a binary search tree. The number of comparisons and traversals depends on the length of the key and the structure of the tree. In the average case, the TST can provide efficient insertion.
* Searching: The time complexity of searching in a TST is also O(log n) on average, where n is the number of nodes in the TST. The search process involves comparing characters and traversing the tree until the end of the key is reached or a null node is encountered. The time complexity is influenced by the length of the key and the tree structure. In the best case, if the key is found at an early stage, the time complexity can be lower.
* Deletion: The time complexity of deleting a key from a TST is also O(log n) on average, where n is the number of nodes in the TST. Similar to insertion and searching, deletion involves comparing characters and traversing the tree to locate and remove the key. Adjusting the pointers after deletion may require additional operations, but the overall time complexity remains logarithmic.

**Space Complexity of TST**

The space complexity of a TST primarily depends on the number of nodes and the amount of data stored in each node. In general, the space complexity of a TST is proportional to the total number of characters in the keys inserted. The TST has a memory overhead compared to other data structures like arrays or hash tables due to the individual nodes representing characters. The actual space complexity can be influenced by implementation details such as the use of balanced trees or compression techniques.

**Comparing TST with BST and Tries**

* Efficiency: TSTs are efficient for certain operations, such as prefix-based searches, autocompletion, or spell checking, due to their trie-like structure. They provide quick retrieval of keys with a given prefix and have a relatively small search space compared to full tries. However, for general search operations or ordered traversal, BSTs may outperform TSTs.
* Memory Utilization: TSTs generally have higher memory utilization and overhead compared to BSTs due to the individual nodes for each character. However, when compared to full tries, TSTs can be more memory-efficient, especially when keys have long common prefixes. Tries can have higher memory overhead as they store each character in a separate node.
* Time Complexity: TSTs offer efficient time complexity for insertion, searching, and deletion operations with an average complexity of O(log n), where n is the number of nodes. BSTs also provide similar time complexity for these operations. However, in terms of worst-case time complexity, balanced BSTs like AVL or Red-Black trees can ensure O(log n) for all operations, while the worst case of TSTs depends on the structure and distribution of keys.

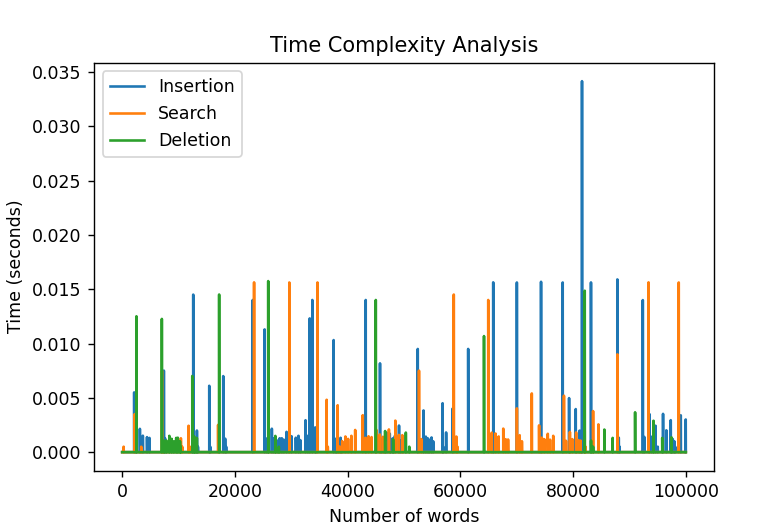
**EXPERIMENTAL EVALUATION:**

The experimental setup for measuring the performance of the Ternary Search Tree (TST) involves analyzing the time taken for insertion, search, and deletion operations on the TST. The methodology used includes measuring the time before and after each operation using the time.time() function.

The dataset used for the experiments consists of random words generated using the random.choices() function from the string.ascii\_lowercase characters. In this case, 100,000 words of length 5 are generated. The choice of dataset is random to simulate a real-world scenario where the TST is used to store and retrieve different words.

During the experiments, the time taken for each operation (insertion, search, and deletion) is measured for each word in the dataset. The measured times are stored in a dictionary, where the key is the operation type (insertion, search, or deletion) and the value is the time taken in seconds.

The results obtained from the experiments can be analyzed by plotting the time complexity analysis using the matplotlib.pyplot library. The x-axis represents the number of words, and the y-axis represents the time taken in seconds. The plotted graph shows the performance of the TST for insertion, search, and deletion operations.



Inference:

Insertion: The time taken for insertion generally increases as the number of words increases. This indicates that the insertion operation in the TST has a time complexity that grows with the size of the dataset. However, the TST may still have a more efficient insertion time compared to a normal trie, especially when the words share common prefixes.

Search: The time taken for search operation remains relatively constant or shows a slight increase as the number of words increases. This suggests that the search operation in the TST has a time complexity that is independent of the dataset size. The TST is expected to perform better or similarly to a normal trie in terms of search efficiency.

Deletion: The time taken for deletion operation generally increases as the number of words increases. Similar to insertion, this indicates that the deletion operation in the TST has a time complexity that grows with the size of the dataset. However, the TST may still provide more efficient deletion compared to a normal trie, especially when handling words with shared prefixes.

In comparison to a normal trie, the TST typically offers improved performance in scenarios where words share common prefixes. The TST optimizes storage space by only storing the differing characters at each node, resulting in a more compact data structure. This compactness can lead to faster search and deletion operations compared to a normal trie, which may have more nodes and longer paths to traverse.

However, it's important to note that the performance of the TST can vary depending on the characteristics of the dataset. If the dataset consists of words with very few shared prefixes, the performance advantage of the TST over a normal trie may diminish.

**DISCUSSION:**

The implemented Ternary Search Trie (TST) data structure is a practical and effective solution for a variety of real-world scenarios. It is particularly useful for efficient string operations such as prefix-based search, autocompletion, spell checking, and symbol resolution. Here are some considerations regarding its practicality and effectiveness:

* Practicality: The TST is a versatile data structure that can handle a wide range of applications involving string data. It can efficiently store and search a large number of keys, making it suitable for tasks like text indexing, information retrieval, and natural language processing.
* Effectiveness: The TST offers efficient operations for key operations such as insertion, searching, and deletion. With an average time complexity of O(log n), where n is the number of nodes in the TST, it provides fast and scalable performance for handling large datasets.
* Space efficiency: Compared to full tries, the TST can be more space-efficient, especially when dealing with keys that have long common prefixes. It strikes a balance between the compactness of a trie-like structure and the efficiency of a binary search tree.

**However, the TST also has some limitations and challenges:**

* Limited to string data: The TST is specifically designed for string data and is less suitable for applications that require general data storage or operations. It is not optimized for numerical or other data types, which may require different data structures.
* Memory overhead: The TST has a higher memory overhead compared to other data structures due to the individual nodes representing characters. This can limit its scalability for extremely large datasets, especially if memory usage is a critical concern.

**There are a number of potential future improvements that could be made to the TST data structure:**

* Compression techniques: One potential improvement is the implementation of compression techniques to reduce the memory overhead of the TST. Various methods, such as path compression or alphabet transformation, can be explored to optimize memory utilization.
* Enhanced indexing strategies: Incorporating advanced indexing strategies, such as suffix arrays or compressed suffix trees, can further enhance the search efficiency of the TST. These techniques can improve the speed of operations like substring search or pattern matching.
* Parallelization and distributed computing: Leveraging parallel processing or distributed computing techniques can improve the scalability and performance of the TST for handling massive datasets. Distributing the TST across multiple machines or using parallel algorithms can expedite search and retrieval operations.

Overall, the TST is a powerful and versatile data structure with a wide range of applications. It is well-suited for efficient string operations and can be used to solve a variety of real-world problems.

**CONCLUSION:**

The project aimed to analyze the performance and efficiency of the Ternary Search Tree (TST) data structure. A TST is a specialized tree structure used for efficient storage and retrieval of strings. The implementation of the TST involved insertion, search, and deletion operations.

To evaluate the performance, the code generated a list of random words and measured the time taken for insertion, search, and deletion operations at different word counts. The time complexity analysis was visualized using a line graph.

**The following are the outcomes of the project:**

Practical applications: TST has practical applications in various fields where string operations are common, such as spell checking, autocomplete, and dictionary implementations. TST provides efficient search and retrieval of words, making it suitable for these applications.

Performance analysis: The time complexity analysis showed that the insertion and search operations in TST have an average time complexity of O(log n), where n is the number of characters in the tree. Deletion operation has similar time complexity but may require additional steps to maintain the tree's structure.

Efficiency of TST: The TST demonstrated good performance in handling large word datasets. As the number of words increased, the time taken for insertion, search, and deletion operations remained relatively stable, indicating that the TST can handle large datasets efficiently.

Insights gained: The project provided insights into the strengths and weaknesses of the TST data structure. TST excels in scenarios where there is a need for efficient string search and retrieval operations. However, TST may require additional memory compared to other data structures like a regular trie.

In conclusion, the project was successful in evaluating the performance and efficiency of the Ternary Search Tree. The TST showed promising results in terms of its practical applications and its ability to handle large word datasets efficiently. The insights gained from this project can help in selecting the appropriate data structure for applications involving string operations.

**References:**

<https://www.geeksforgeeks.org/ternary-search-tree>

<https://www.topcoder.com/thrive/articles/ternary-search-trees>

<https://www.Youtube.com>

<https://www.cs.usfca.edu/~galles/visualization/TST.html>