Tries

Bilal Ahmed - 028
Usman Abubakr - 031
Dawood Shahzad - 034
Hassam Azam - 035
Ali Hassan - 037
Syed Wabil Shah - 029

Table of Contents

Exploring Tries

What is a Trie? [2]

'Trie' is a Tree-based data structure that is used commonly in search-based applications.

The name 'Trie' is actually influx of the word "Retrieval", which means to obtain or get something back. Trie also has a few other names, such as the Prefix or Radix Tree.

Trie data structure is a very powerful tool for efficient string processing and retrieval. It is a multi-way tree structure useful for storing strings over an alphabet, when we are storing them. It has been used to store large dictionaries of English, say, words in spell-checking programs. However, the penalty on tries is the storage requirements.

This report highlights Trie's efficiency and discusses its usecases, implementation details, algorithms for search, insertion, and deletion, as well as brief comparisons with other popular data structures and shows how it is more efficient in its domain.

Figure 1: Image of a tree in a dense forest, depicting a 'Trie' with countless nodes

Why Trie?

The Trie data structure, also referred to as a prefix tree, is a versatile and efficient solution for managing strings and facilitating rapid search, insertion, and deletion operations. It finds widespread use in applications involving string processing and retrieval, such as autocomplete systems, spell checking, and information retrieval.

Essentially, a Trie is a tree-like arrangement that organizes strings by representing them as a series of interconnected nodes. Each node corresponds to a character, and the path from the root to a node forms a string. The main advantage of the Trie lies in its ability to store and retrieve strings with efficiency, especially when working with large datasets.

A primary advantage of the Trie lies in its efficiency in search operations. By traversing through the Trie from the root, matching characters at each level, it enables swift retrieval of strings matching a given prefix or complete strings. This property makes it particularly suitable for autocomplete applications where suggestions are based on partial input.

Key Features

Trie data structure has some key features which makes it stand out from other data structures if we see specifically in the domain of string processing. Following are some properties:

1. Prefix based storage

   Tries store strings as a collection of interconnected nodes, with each node representing a character. The path from the root to a node forms a string, allowing for efficient storage and retrieval based on prefixes. User will just have to type 1 letter in order to search for whole name, word etc.

2. Auto-Complete

   Tries are good option for auto-complete systems. It quickly generates suggestions by traversing the Trie based on single word input and retrieves all the strings that match the provided prefix which provides fast and efficient service in almost run-time.

3. Diverse String Operations

   Tries can handle various string-related operations, such as exact match search, prefix search, and even fuzzy search. This versatility allows for flexible querying and manipulation of strings according to requirements by the end user.

4. Capability

   Trie data structure can handle large data sets and give output in linear time. As the number of strings increase, it remains effective and ensures fast retrieval and search operations.

Internal Working of a Trie

Structure of a Trie [3]

First we have to define the structure for a Node in our Trie, this can be done using a struct or a class

Inside the class, we have:

children[26]: This is an array of size 26, which represents the children of the current node. Since the Trie is designed to store words consisting of lowercase English alphabets, each node can have up to 26 children, one for each letter of the alphabet. The array is initialized to NULL initially.

end_of_word: This is a boolean flag that indicates whether the current node marks the end of a word. It is set to false by default.

letter: This variable stores the character associated with the current node. For example, if the node represents the letter 'a', the letter variable will be assigned the value 'a'. The default value is '\0' (null character).

TrieNode(): This is the constructor of the TrieNode class. It initializes the end_of_word flag to false, sets all the elements of the children array to NULL, and assigns the letter variable the value '\0'.

```
class TrieNode {

    public:

        // An array of pointers, one pointer for each alphabet
        TrieNode *children[26];

        // A flag that marks if the word ends on this particular node.
        bool end_of_word;

        // Character stored in this node
        char letter;

        // Constructor
        TrieNode() {

            // Initially, no word ends anywhere
            end_of_word = false;

            // Setting all child nodes as NULL because we only have the root node
            for (int i = 0; i < 26; i++) {
                children[i] = NULL;
            }

            // Storing the NULL character on the root node
            letter = '\0';
        }
};
```

Internal Working of a Trie

```
class Trie {

    // Root node of the Trie
    TrieNode root;

    // Insert a string in the Trie
    // Returns: void
    void insert (TrieNode* root, string str);

    // Remove a string from the Trie
    // Returns: void
    void delete (TrieNode* root, string str);

    // Search the word in Trie
    // Returns: bool (whether the word was found or not)
    bool search(TrieNode* root, string str);

    // Print the Trie
    void print(TrieNode* root);

    //              Other Functions
    // Print the words with the specified prefix (Auto-complete)
    void PrintPrefix (TrieNode *root, string prefix);

};
```

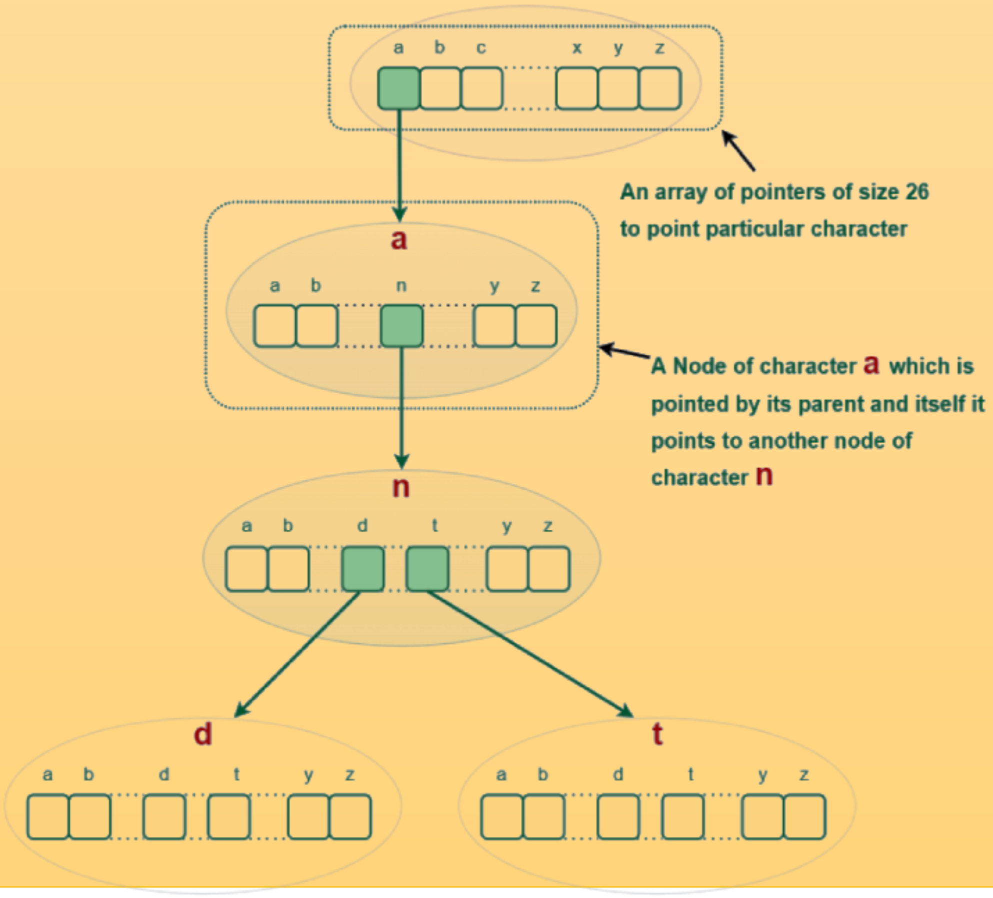

a b c x y z
An array of pointers of size 26 to point particular character
a
a b n y z
A Node of character a which is pointed by its parent and itself it points to another node of character n
n
a b d t y z
d
a b d t y z
t
a b d t y z

Figure 2: Illustration of the structure of a Trie

Internal Working of a Trie

Insertion in a Trie

1. For each character in the target string, check if there is an outgoing edge from the current node that corresponds to that character.

2. If an edge is found, move to the next node by following that edge. If an edge for the character does not exist, proceed to the next step.

3. If an edge for the character is not found, it means that the path for the character needs to be created. Create a new node and connect it to the current node using an edge labeled with the character. Move to the newly created node.

4. Continue this process for each subsequent character in the target string. For each character, check if there is an edge from the current node that matches the character, and if so, move to the corresponding child node.

5. If an edge is not found, create a new node and connect it to the current node with an edge labeled with the character.

6. At the node representing the last character of the target string, mark it as the end of the string to indicate that a complete string terminates at this node

7. After successfully inserting the string into the Trie, the insertion process is complete.

Internal Working of a Trie

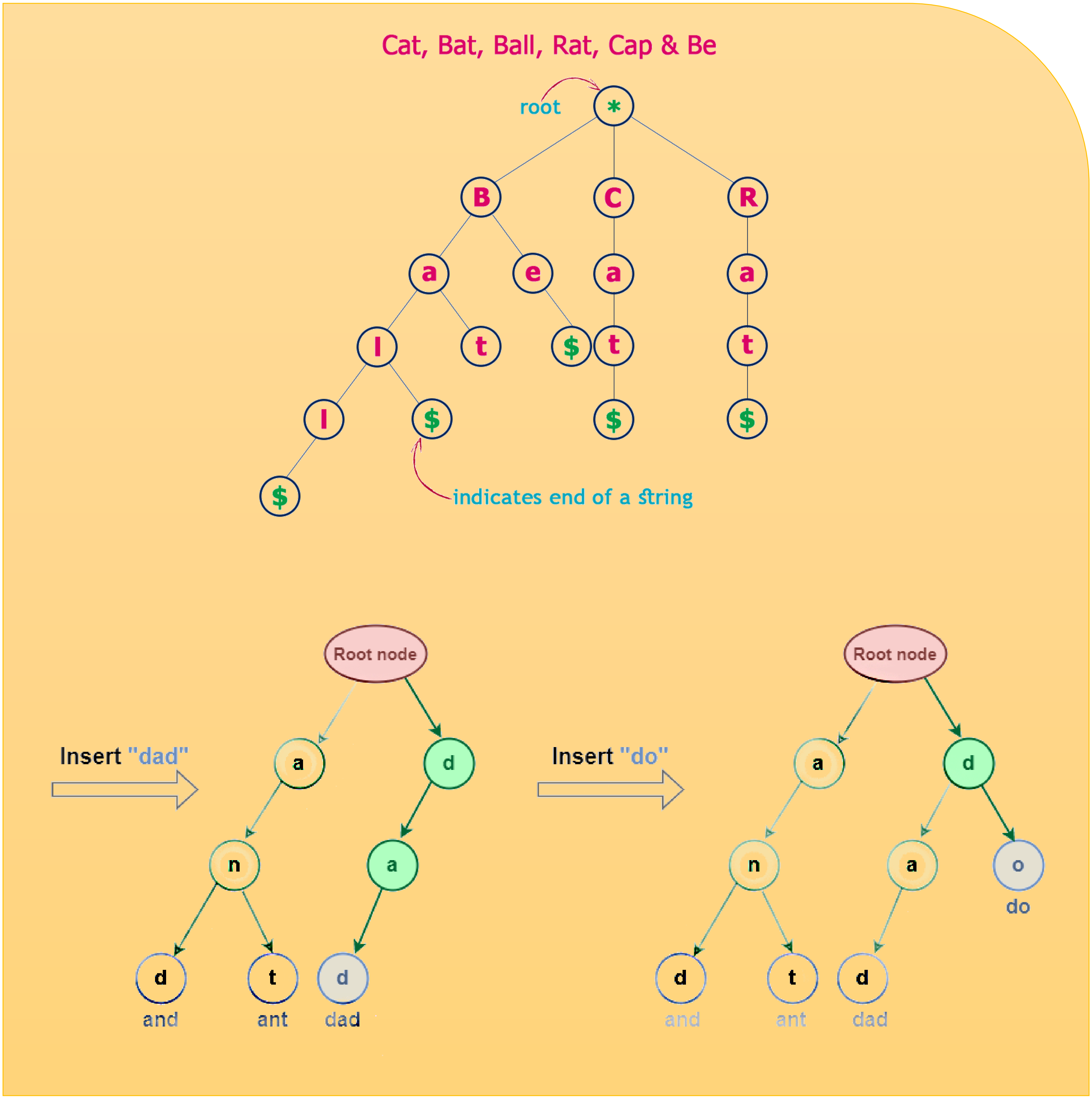


Figure 3: Demonstration of the Insertion procedure

Internal Working of a Trie

Searching in a Trie

1. Searching in a Trie involves traversing through the nodes to locate a specific string or a prefix.
2. The search begins at the root node which usually has an empty string or the common prefix for all the strings stored from that alphabet.
3. Now take the first character of the target string and check if there is an outgoing string from the current node that matches the required string.
4. Repeat the following steps for each character of the required string, check if there is an edge from the current node that matches the character, and if so, move to the corresponding child node.
5. There are 2 possible termination cases for this process.
6. Firstly if the whole string is completely matched and no other alphabet is left to search, it will return a positive result and end searching process.
7. If at any point during the search, a character is not found as an outgoing edge from a node, or there are no more nodes to traverse, it means the string does not exist in the Trie, returns a negative result indicating an unsuccessful search.

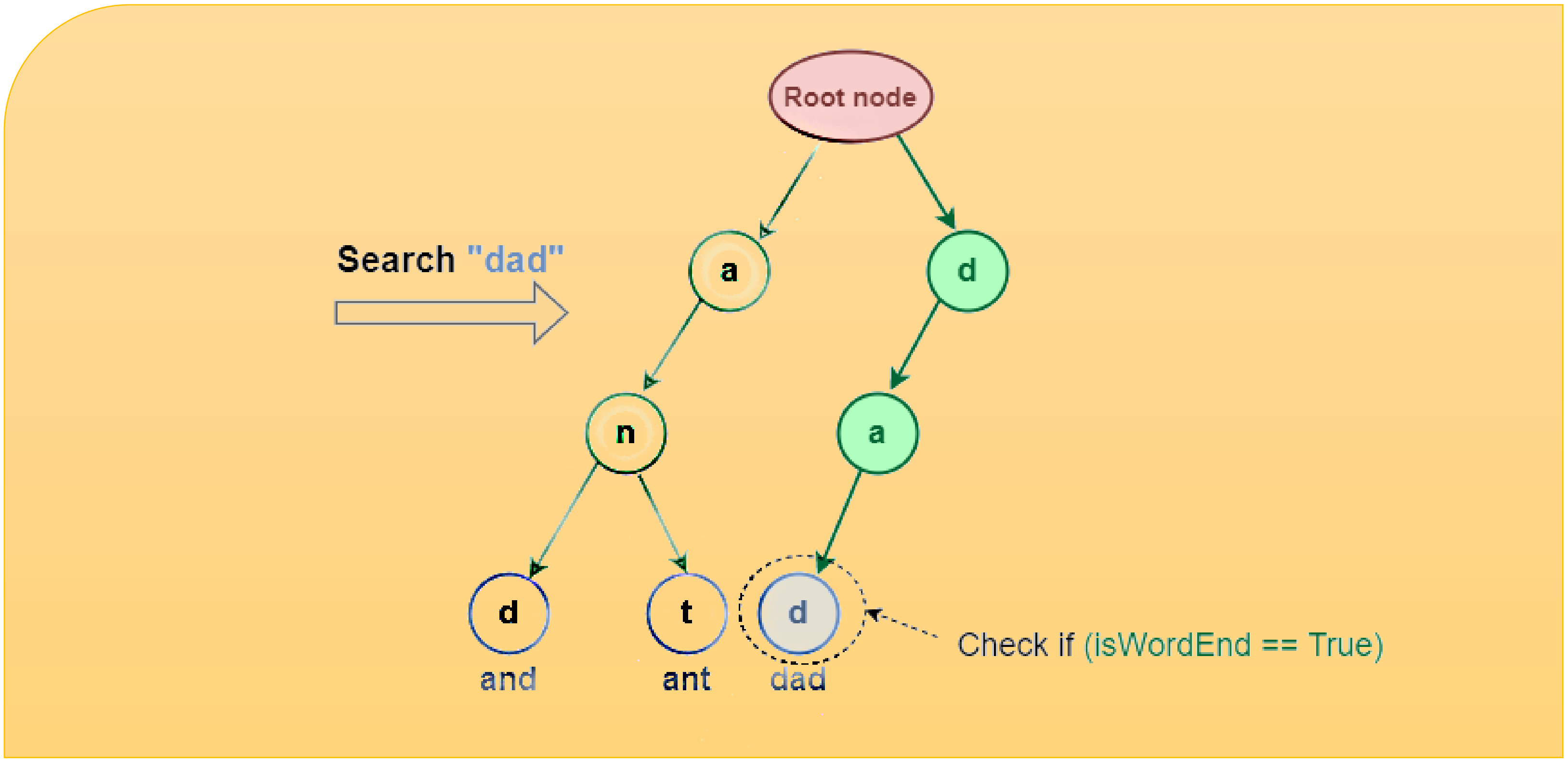


Figure 4: Demonstration of the searching procedure in a Trie

Internal Working of a Trie

Deletion in a Trie [4]

The delete operation in a Trie data structure involves removing a specific string from the Trie. The process requires careful consideration to ensure the integrity of the Trie's structure.

Deletion process has cases where two strings are sharing a common prefix or a middle letter so this task needs to be catered carefully.

1. The process starts at the root node of the structure.

2. Move through the Trie, following the nodes that correspond to each character of the target string. This traversal brings us to the node representing the last character of the string.

3. At the node representing the last character of the string, mark the end of the string to indicate that the string terminates at this point.

4. Check if the current node has any other children. If it does not have any other children and is not marked as the end of any other string, it means that there are no other strings with a common prefix. In this case, remove the node and continue to the step 6.

5. If the current node has other children or is marked as the end of another string, do not remove the node. Instead, unmark the end of the string to maintain the integrity of the Trie for other strings.

6. Starting from the node representing the last character of the string, move upwards through the Trie. At each node, check if it can be safely removed. A node can be safely removed if it has no children and is not marked as the end of any other string. Remove all such unnecessary nodes until reaching a node that either has other children or is marked as the end of another string.

7. If there are other strings with shared prefixes that were unaffected by the deletion, repeat the last 2 steps to handle those cases as well.

Internal Working of a Trie

Deletion Cases

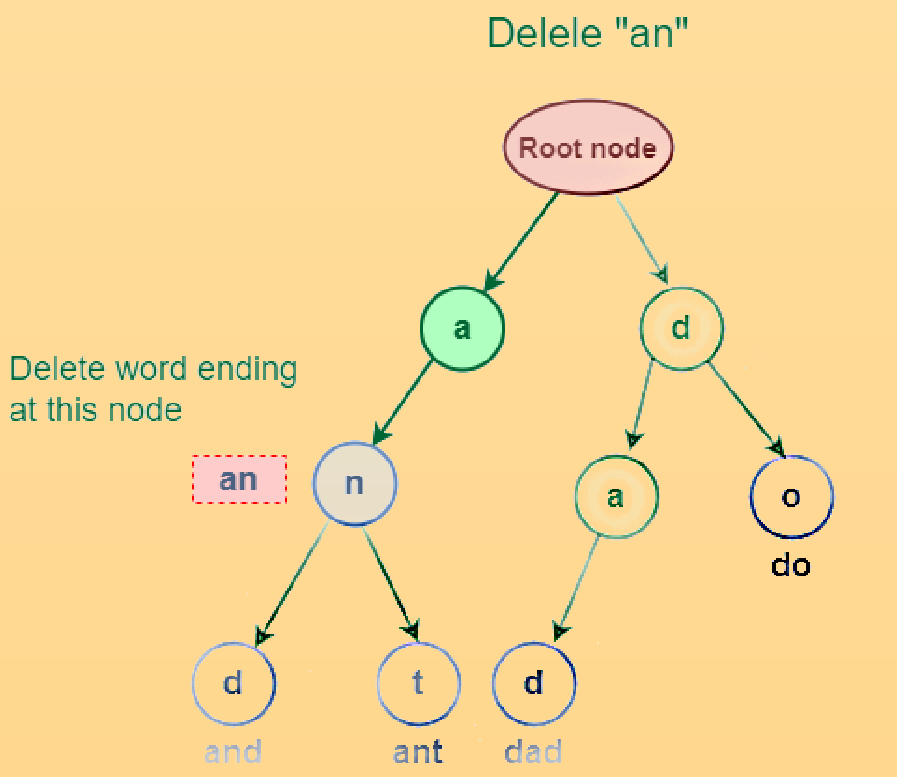
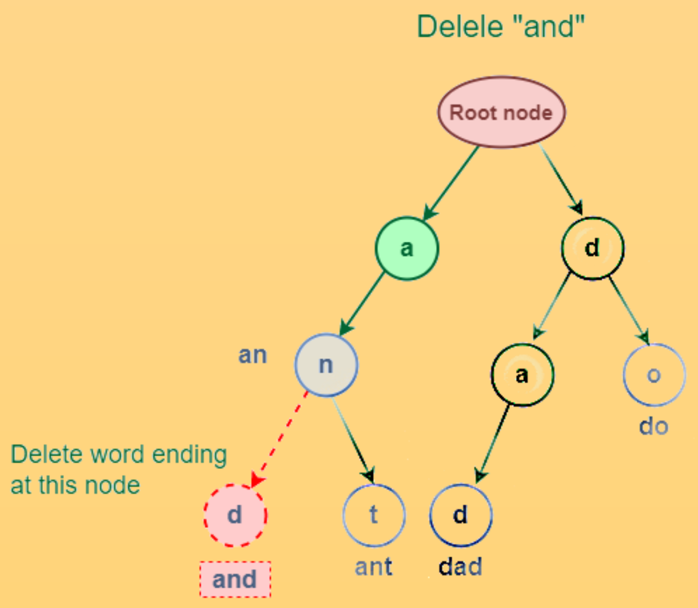


Figure 5: Case where the word is a prefix for other words



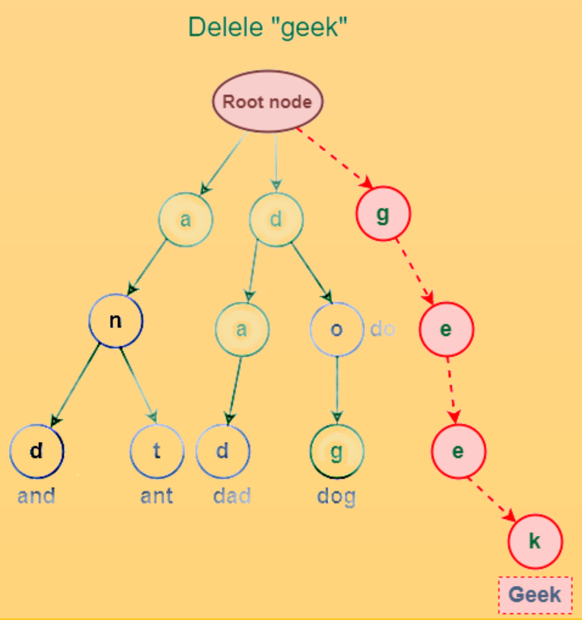


Figure 6: Case where the word shares a common node

Figure 7: Case where the word is safe to delete

Gauging Performance

Time Complexity

The time complexity of the operations like insertion, deletion, and searching, is O(L) where L is the length of the string that needs to be searched, or to be deleted, or to be inserted.

The time increases as the size of string increases. But it has a linear time complexity and it can handle long and large data sets easily and efficiently.

Trie allows us to input and finds strings in O(L) time, It is faster as compared to both hash tables and binary search trees.

Tries don't have any overhead of Hash function in the implementation hence they're faster than other data structures that require hashing of a key. But if the string being searched is very long, it may take more time than what it would take for a hashing algorithm.

Time Complexity Table

Operation	Best Case	Average Case	Worst Case
Insertion	O(L)	O(L)	O(L)
Searching	O(1)	O(L)	O(L)
Deletion	O(L)	O(L)	O(L)

**'L' represents the length of the string being passed in*

'Search' Best Case *(Special Case):*

The best case time complexity of searching in a Trie is constant [O(1)] because it means that the string to-be-searched contains only 1 character, hence it's length being 1 -> O(length)

For all other cases, it depends on the length of the key being passed in

Gauging Performance

Space Complexity

If you're don't have good amount of space to spare, Tries aren't a good choice, because they take up lots of memory, this is due to the fact that a Trie contains an array of pointers at each node

Let's consider a case where we have to make up a Trie containing only lowercase alphabets

This makes up 26 different alphabets, which makes every single node in the Trie contain an array of pointers of size 26, ands this grows exponentially as the Trie gains more depth.

Operation	Space Complexity
Insertion	O(L*N)
Searching	O(1)
Deletion	O(1)

*'L' represents the length of the string and

*'N' represents the total number of strings that are stored in the trie.

Making Tries more efficient: [6]

However, there are a few ways to tacke the space inefficiency of tries

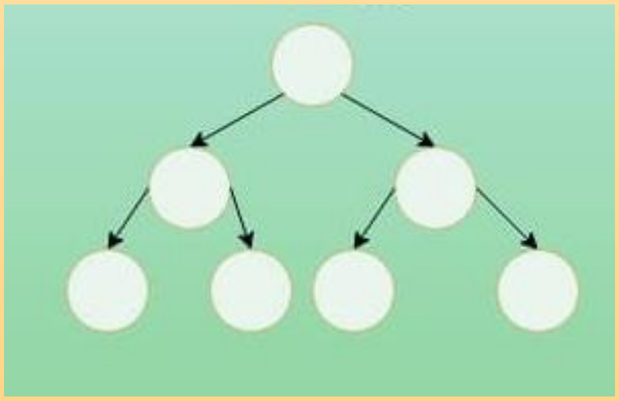
For example:

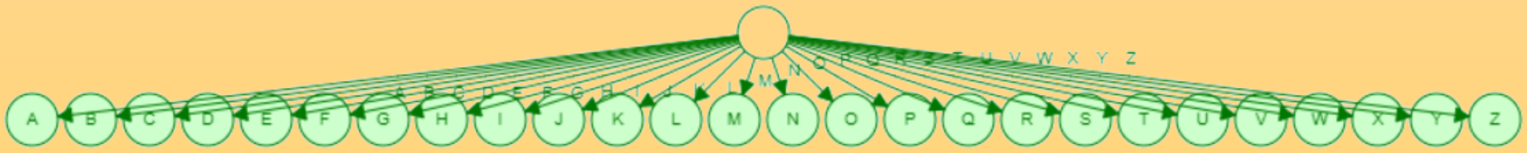
- Changing the data structure for holding the pointers.
- Eliminating unnecessary trie nodes.
- Eliminating the use of a character variable to store the letter in (using array to determine it instead)

Comparisons with other data structures

Trie VS Balanced Tree [7]

Structure





Search Time Complexity

A balanced search tree ensures efficient search operations with a time complexity of O(log n), where n represents the number of elements in the tree. This logarithmic time complexity allows for quick retrieval of ordered data, making it well-suited for such scenarios.

In a trie, the search time complexity is O(L), where L denotes the length of the key or string being searched. Tries offer constant time complexity for searching and retrieving strings.

Insertion/Deletion

Keeping a balanced condition during insertion and deletion operations in a balanced search tree involves more intricate steps, such as node rotations and reordering. Although these operations have a time complexity of O(log n), they are more involved compared to trie operations.

Memory Usage

Tries are not memory efficient in the worst case, lets put the worst case aside for this comparison.

Comparatively, balanced search trees consume more memory than trie. Each node in a balanced search tree requires extra memory to store pointers for left and right children, along with additional balance factor information in the case of AVL trees.

Trie, on the other hand, can be more memory-efficient, particularly for storing extensive string collections. Tries optimize memory usage by sharing common prefixes, resulting in reduced memory requirements. However, in scenarios with a large alphabet size or sparse data, tries may use more memory.

Comparisons with other data structures

Trie vs Hash Table [5]

Advantages

- Prefix Search:

  Trie efficiently performs prefix search or auto-complete by following the links corresponding to the prefix characters. In contrast, hashing would require checking every string in the hash table for a prefix match.

- Sorted Order:

  Trie can easily print words in alphabetical order by traversing the trie in a depth-first manner and printing complete nodes encountered. Hashing would require sorting the strings before printing, making it less convenient.

- No Collisions:

  Trie avoids hash collisions and the need for a hash function. Hash collisions occur when different keys map to the same index, affecting hashing performance and requiring handling techniques like chaining or open addressing. Designing a good hash function to distribute keys uniformly and prevent collisions is non-trivial. In a trie, each string has a unique path, eliminating collisions.

Dis-advantages

- Space Consumption:

  Tries may consume more space compared to hashing. Tries can have numerous nodes, each storing only a single character, which can be inefficient in terms of memory usage. Hashing can store multiple characters in a single entry of the hash table, making it potentially more space-efficient.

- Time Complexity:

  Tries may have higher time complexity than hashing for certain operations. Inserting or searching a string in a trie can take longer, particularly when the string is lengthy or shares many common prefixes with other strings. In contrast, hashing can perform these operations in constant time on average, provided a good hash function is used and collisions are minimized.

Applications of Trie

Autocomplete Systems [9]

When you start thinking about a problem that can be solved easily with the help of tries, your brain may stumble upon the idea of using Tries in a search engine, a grammatical error checker, or some other complicated application, such as the LIKE query in SQL.

Well, let's take a look at how Trie can be used in a Search Engine's autocomplete system.

1. Build the Trie: The first step is to construct a trie data structure by inserting all the words or phrases that will be used for auto-completion. Each character of a word is represented as a node in the trie, with edges connecting the nodes representing consecutive characters.

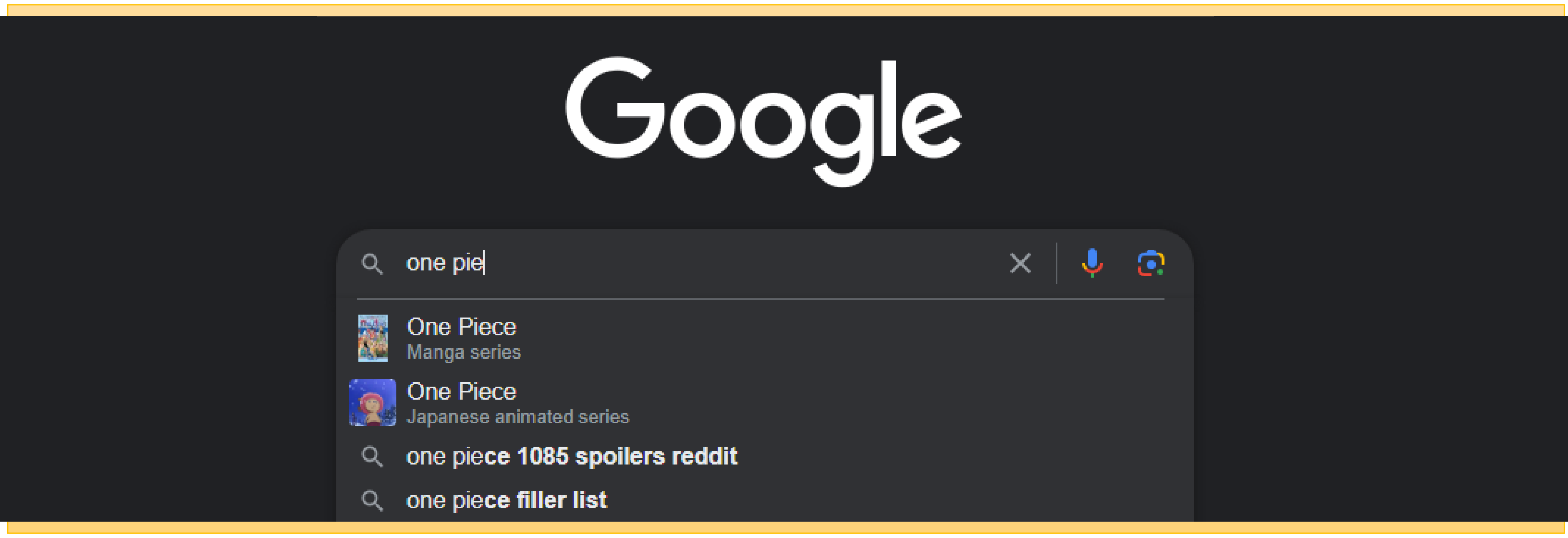
2. Input Processing: As the user types a prefix or partial word, the auto-completion system receives the input and starts searching for completions based on the entered prefix.

3. Traverse the Trie: Starting from the root node of the trie, traverse down the tree based on the characters in the user's input.

4. Find Completions: Once the traversal reaches the node corresponding to the last character of the input, you can find all possible completions by exploring the subtree rooted at that node. This can be done by performing a depth-first search (DFS) or any other traversal technique.

5. Retrieve Suggestions: While traversing the subtree, collect all the words or phrases associated with the nodes encountered. These are the suggested completions for the user's input. You can store the completions in a list or another data structure to present them to the user.

6. Display Suggestions: Present the suggestions to the user interface, such as a dropdown menu or a list of options. The user can then select the desired completion from the presented suggestions.


one pie
One Piece
Manga series
One Piece
Japanese animated series
one piece 1085 spoilers reddit
one piece filler list

Applications of Trie

Tries in DNA [8]

Tries can be used in DNA sequence analysis and bioinformatics applications. They offer efficient storage, retrieval, and manipulation of DNA sequences. Here are a few ways in which tries can be used in DNA-related tasks:

1. DNA Sequence Storage:
   Tries can be employed to store and organize DNA sequences in a hierarchical structure. Each node in the trie represents a nucleotide, and the edges represent the possible nucleotide transitions. This allows for quick and efficient retrieval of specific DNA sequences. [10]

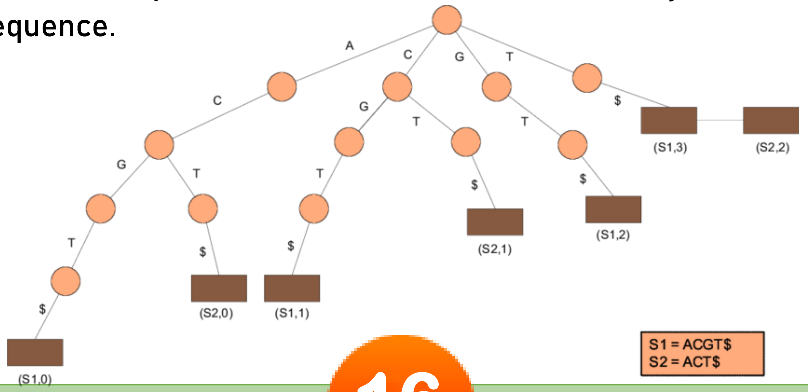
2. Sequence Alignment:
   Tries can aid in DNA sequence alignment algorithms, such as the Smith-Waterman algorithm or the Needleman-Wunsch algorithm. By constructing a trie from a set of reference sequences, it becomes easier to search for similarities or matches between the reference sequences and an input sequence.

3. Motif Finding:
   Tries can assist in the identification of DNA sequence motifs. A motif is a recurring pattern or sequence that has functional or structural significance. By constructing a trie from a set of known motifs, it becomes easier to search for these motifs in larger DNA sequences.

4. DNA Database Indexing:
   Tries can be used to index and search DNA databases efficiently. By constructing a trie from a collection of DNA sequences, it becomes faster to search for specific sequences or patterns within the database.

5. DNA Sequence Analysis Tools:
   Tries can be integrated into DNA sequence analysis tools and software. These tools can use trie-based algorithms for tasks such as sequence assembly, variant calling, gene prediction, or identifying common patterns or features within DNA sequences. [10]

6. DNA Spelling Correction:
   Tries can assist in DNA spelling correction or error correction algorithms. By constructing a trie from a set of known DNA sequences, it becomes easier to identify and correct errors or mutations in an input DNA sequence.



References

[1] "Trie Data Structure Implementation," Freecodecamp.Org, November 2020. [Online]. Available: https://www.freecodecamp.org/news/trie-data-structure-implementation/.

[2] GeeksforGeeks, "Introduction to Trie - Data Structure and Algorithm Tutorials," May 2022. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-trie-data-structure-and-algorithm-tutorials.

[3] AlgoTree, "Trie in C++," [Online]. Available: https://algotree.org/algorithms/trie/. [Accessed June 2023].

[4] GeeksforGeeks, "Trie | (Delete)," February 2023. [Online]. Available: https://www.geeksforgeeks.org/trie-delete/.

[5] A. Jha, "Hash Table vs Trie," December 2022. [Online]. Available: https://www.geeksforgeeks.org/hash-table-vs-trie/.

[6] Stanford, "Tries and String Matching," [Online]. Available: https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/09/Small09.pdf.

[7] Joe, "Difference between Tries and Balanced Trees," [Online].

Available: https://stackoverflow.com/questions/4737904/difference-between-tries-and-trees.

[8] S. Wan and Q. Zou, "HAlign-II: efficient ultra-large multiple sequence alignment and phylogenetic tree reconstruction with distributed and parallel computing," 2017. [Online].

Available: https://almob.biomedcentral.com/articles/10.1186/s13015-017-0116-x.

[9] GeeksforGeeks, "Auto-complete feature using Trie," [Online].

Available: https://www.geeksforgeeks.org/auto-complete-feature-using-trie/.

[10] S. Park, J. Yoon, S.-W. Kim and J.-I. Won, "An efficient approach for sequence matching in large DNA databases," February 2006. [Online]. Available: https://www.researchgate.net/publication/220195697_An_efficient_approach_for_sequence_matching_in_large_DNA_databases

Credits

Ali Hassan	--	02-136221-037
Hassam Azam	--	02-136221-035
Syed Wabil Shah	---	02-136221-029
M. Usman Abubakar	---	02-136221-031
Dawood Shahzad	---	02-136221-034
Bilal Ahmed	--	02-136221-028