1: Introduction to Tries

A Trie is a tree-like data structure that is particularly efficient for storing and searching strings.

Tries are also known as prefix trees because they allow for efficient prefix-based searches.

2: Structure of a Trie

Each Trie node contains an array of child nodes, one for each character in the alphabet.

A boolean flag in each node indicates whether it represents the end of a word.

The root of the Trie represents an empty string, and each path from the root to a node represents a string formed by concatenating characters along the path.

3: Key Trie Operations

Insertion: Adding a new word to the Trie.

Searching: Checking if a word is present in the Trie.

Deletion: Removing a word from the Trie.

Prefix-based search: Finding all words with a given prefix.

4: Implementation Details

We have implemented a Trie class, which internally uses TrieNode, CustomString, and StringArray classes.

The TrieNode class represents the nodes of the Trie and manages child nodes and the end-of-word flag.

The CustomString class is a custom string manipulation class that allows for dynamic resizing and efficient concatenation.

The StringArray class is a custom dynamic array for storing and managing strings.

5: Space Usage

The Trie data structure is space-efficient, as it allows for the sharing of common prefixes.

However, the space complexity increases as the number of unique characters in the alphabet increases.

The next logical step would be to add Unicode support which would greatly increase the alphabet size and would surely introduce different challenges.

6: Time Complexity

Insertion, searching, and deletion have time complexity O(L), where L is the length of the string.

Prefix-based search has a time complexity of O(L + N), where L is the length of the prefix, and N is the number of words with the given prefix.

7: Trie in Action (testing)

Our Trie implementation supports insertion, searching, deletion, and prefix-based search.

We've demonstrated its usage with a test program that inserts several words, searches for them, and deletes one. Then it gathers words that start with a prefix, known as autocomplete.

Our custom classes, TrieNode, CustomString, and StringArray, ensure that no external dependencies are needed. CustomString and StringArray are, surprise, things we have implemented already.

8: Conclusion

Tries offer an efficient solution for string storage, search, and prefix-based operations.

Our custom implementation showcases a robust and dependency-free Trie data structure.

Tries can be applied to various applications such as autocomplete (implemented here in a basic form), spell checking, and IP routing.