Back to Home

Two Pointers
Sliding Window
Intervals
Stack
Linked List
Binary Search
Heap
Depth-First Search
Breadth-First Search
Backtracking
Graphs
Dynamic Programming
Greedy Algorithms
Trie
Overview
Implement Trie Methods
Prefix Matching
Prefix Sum
Matrices

Greedy Algorithms... Implement Trie Met...

Tries

A trie (also known as a Prefix Tree) stores a set of strings in a tree-like data structure. The trie below stores the strings APPLE, APP, BAT, BALL, BATS, and BALL:

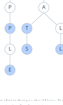
Strings with a common prefix share the same nodes in the trie. For example, the strings APPLE and APP share the nodes A, P, and P.

A trie allows us to efficiently search if a given word exists in the trie. For example, we can search for the word APPLE by starting at the root of the trie and following the nodes along the path annotated below:

APPLE

A trie is commonly used to implement features like spell-checkers and auto-complete. For example, if we type in the string "BA", then our trie can suggest "BALL" and "BAT" as possible completions.

BA



Tries provide a powerful data structure for prefix-based lookups, making them ideal for string operations. In particular, we will learn how to visualize the most common operations on a trie: search, insert, and delete.

Basics

- A trie consists of a series of nodes arranged in a tree. Each node in the trie represents a character in one of the strings in the trie, and its children represent the next character in the string.
- Each node also has a boolean value that indicates whether the node represents the end of a word. Nodes where this boolean value is true are colored blue in this guide.
- A trie supports 3 main operations: search(word), insert(word), and delete(word).

Trie Class

A trie is typically implemented as a class with a reference to `root` of the type `TrieNode`. The class has methods to search, insert, and delete words from the trie.

```
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def search(self, word):
        # Return True if word is in trie, False otherwise
        pass

    def insert(self, word):
        # Insert word into trie
        pass

    def delete(self, word):
        # Delete word from trie
        pass
```

TrieNodes

The `TrieNode` class can be defined as:

```
class TrieNode:
    def __init__(self, children = {}, is_end = False):
        self.children = children
        self.is_end_of_word = is_end
```

The `children` dictionary is a mapping from characters to `TrieNode` objects. For example, the code snippet below shows what the `children` dictionary looks like for a node with two children, "A" and "B":

In practice, we won't be manually creating `TrieNode` objects. Instead, we'll use the `Trie` class to interact with the trie. This snippet is only included to help you understand the structure of a trie.

```
root = TrieNode()

root.children = {
    'A': TrieNode(),
    'B': TrieNode()
}
```

Next, we'll learn how to visualize the 3 main operations on a trie: search, insert, and delete.

Trie Operations

This section focuses on how to visualize each operation to get a feel for how they work. We'll cover the implementation in-depth in the first practice problem.

Search

The `search` operation takes a search term as input and returns whether the term exists in the trie.

We start from the root node and the first character of the search term. We then traverse down the trie by checking if any of the children of the current node match the next character in the search term. If they do, we move to that node and continue the search with the next character in the search term.

The animation below visualizes the search operation for an input (case-sensitive) of your choice, with a trie storing APPLE, APP, BAT, BALL, BATS, and BALL.

Try different search terms to get a feel for how the search operation works.

BALL returns true

BA returns false, as BA is not marked as the end of a word

Time Complexity

$O(L)$, where L is the length of the word being searched in the worst case we need to traverse L nodes in the Trie to find the word. Each node traversal takes constant time $O(1)$. For a total of $O(L)$ operations.

Insertion

The `insert` operation takes a word as input and adds it to the trie.

We traverse the trie until we reach the last character of the search term. From there, we add the nodes that don't exist already in the trie, and mark the last node as the end of a word.

The animation visualizes the insert operation for a word of your choice (case-sensitive) into a trie containing APPLE, APP, BAT, BATS, and BALLET.

The animation resets back to the original trie after each insertion - the trie does not accumulate words as you insert them.

Some example words to insert:

APPLE (already exists)

BALL (no new nodes created, but "L" is marked as the end of a word)

CON, (creates a new branch in the trie from the root)

Time Complexity

$O(L)$, where L is the length of the word being inserted. In the worst case, such as when the trie is empty or the word being inserted has no common prefixes with existing words, we need to insert L nodes, each of which takes constant time $O(1)$.

Deletion

The `delete` operation deletes a word from the trie.

We traverse down to the last character of the word we want to delete, set the "end of word" flag to false, and then remove any nodes that are not part of any other words in the trie.

The animation visualizes a delete operation from a trie containing APPLE, APP, BAT, BATS, BALL, and BALLET.

The animation resets back to the original trie after each deletion - meaning the trie does not continuously shrink as you delete words.

Some example words to delete:

BALL (removes the EOW marker from the "L" node)

CON, (does nothing, as the word does not exist in the trie)

BATS (removes the "S" node)

Time Complexity

$O(L)$. In the worst case, such as when the word to delete is the only word in the trie, we need to first traverse L nodes in the trie to find the word to delete, and then delete L nodes. Each node traversal takes constant time $O(1)$. For a total $2L$ operations, which is $O(L)$.

Summary

Operation	Description	Time Complexity
Search	Search for a word in the trie	$O(L)$
Insert	Insert a word in the trie	$O(L)$
Delete	Delete a word from the trie	$O(L)$

Space Complexity

The space complexity of a trie is $O(C)$, where C is the total number of characters between all the words stored in the trie. This is due to the worst case, which happens when there are no common prefixes between the words stored in the trie.

A trie in which there are no shared nodes.

Next: Implement Trie Methods

Add a comment...

Anonymous

Post as Tanny Williams

No comments yet.

© 2024 Optick Labs Inc. All rights reserved.