← 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

← Overview

Prefix Matching →

Implement Trie Methods

DESCRIPTION

Implement the search and delete methods of a Trie.

- `search(word)` returns true if the word is in the Trie, and false otherwise.
- `delete(word)` removes the word from the Trie, and does not return a value.

The creation of the Trie and the `insert` method are already implemented for you.

The test cases include two parameters:

- `initialWords`: a list of words to add to the Trie.
- `commands`: a list of commands to run. Each command is a tuple, where the first element is either "search" or "delete", and the second element is the word to search or delete.

The test cases will create the Trie with the initial words, and then run the commands in order, and compare the output to the expected output. Note we only compare the output of the search commands, not the delete commands.

EXAMPLES

Input:

```
initialWords = ["apple", "app", "apartment"]
commands = [
["search","apple"],
["search","apartment"],
["search","app"],
["delete","app"],
["search","app"],
]
```

Output: [True, True, False, False]

Explanation:

```
trie.search("apple") -> True
trie.search("apartment") -> True
trie.search("app") -> False
trie.delete("app") -> Note # Return value not checked
trie.search("app") -> False
```

⚠ Do not modify any parts of the code other than the functions labled with "=== YOUR CODE HERE ===".

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.isEndOfWord = False

class Solution:
    def create_trie(self, words):
        # === DO NOT MODIFY ===
        self.root = TrieNode()
        for word in words:
            self.insert(word)

    def insert(self, word):
        # === DO NOT MODIFY ===
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.isEndOfWord = True

    def search(self, word):
        """
        Search the trie for the given word.

        Returns True if the word exists in the trie, False otherwise
        """
        # === YOUR CODE HERE ===
        return False

    def delete(self, word):
        """
        Deletes the given the word from the Trie.

        Returns None.
        """
        # === YOUR CODE HERE ===
        pass

    def trie(self, initialWords, commands):
        # === DO NOT MODIFY ===
        self.create_trie(initialWords)

        output = []
        for command, word in commands:
            if command == "search":
                output.append(self.search(word))
            elif command == "delete":
                self.delete(word)
        return output
```

Results | AI Feedback | View Answer | Run

Run your code to see results here

Explanation

It will help to refresh your memory on how to visualize each operation before diving into the implementation.

Search
Insert

Search

The intuition for searching is to search for each character in the word by traversing down nodes in the trie. When we reach the end of the word, we check if that node is marked as the end of a word. This is necessary

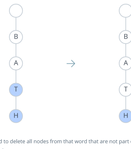
```python
def search(self, word):
    """
    Search the trie for the given word.

    Returns True if the word exists in the trie, False otherwise
    """
    # start from the root node
    node = self.root

    # traverse down the trie
    for char in word:
        if char not in node.children:
            # the word does not exist in the trie
            return False
        node = node.children[char]

    return node.is_end_of_word
```

Delete

Intuition

To delete a word from a trie, we need to first unmark the node corresponding to the last character of the word as the end of a word.

The Trie below contains BAT and BATH. To delete BAT, we need to unmark T as the end of a word.



Then, we need to delete all nodes from that word that are not part of any other words in the trie.

For example, the Trie below contains two words: "BALLET" and "BALLOON". If we were to delete "BALLET", we can safely delete "T" and "T", but not "BALL" because it is part of "BALLOON".

If we look closer at the nodes we can delete, they have two properties in common:

- They are not the end of any word
- They don't have any children

So we want to first traverse down to the node corresponding to the last character of the word we are trying to delete and unmark it as the end of a word. From there, we can traverse back "up" by deleting nodes that are not part of any other words based on the two conditions above.

Implementation

The above logic is best implemented recursively with a helper function `_delete`. Each call to `_delete` returns a boolean indicating whether the current node can be deleted from its' parent's dictionary of children.

It returns True if:

- The current node is not the end of any word (not node.isEndOfWord) AND
- The current node has no children (len(node.children) == 0)

Base Case

The base case for the recursive function is when we reach the end of the word. We need to set `node.isEndOfWord = False` to ensure that we removed the given word from the trie.

At this point, we can start deleting nodes that are not part of any other words. Each node returns `True` if it should be deleted from its parent's dictionary of children based on the two conditions above.

The parent receives that boolean and:

- If the boolean is `True`, it deletes the child node from its dictionary of children. The parent node then returns if it should be deleted as well.
- If the boolean is `False`, it does not delete the child node and returns `False`, which prevents any further deletions.

```python
def delete(self, word):
    """
    Deletes the given the word from the Trie.

    Returns None.
    """
    def _delete(node, index):
        # base case: we have reached the end of the word
        if index == len(word):
            if not node.isEndOfWord:
                return False
            node.isEndOfWord = False
            return len(node.children) == 0

        char = word[index]
        if char not in node.children:
            return False

        delete_child = _delete(node.children[char], index + 1)
        if delete_child:
            del node.children[char]
            return not node.isEndOfWord and len(node.children) == 0
        return False

    if _delete(self.root, 0):
        if word and word[0] in self.root.children:
            # safe just to delete if true the root's children
            del self.root.children[word[0]]
```

Next: Prefix Matching →