Deletion in Trie

After insertion and search, let's figure out the logic behind deletion in tries.

We'll cover the following

Deleting a Word in a Trie

While deleting a node, we need to make sure that the node that we are trying to delete does not have any further child branches. If there are no branches, then we can easily remove the node.

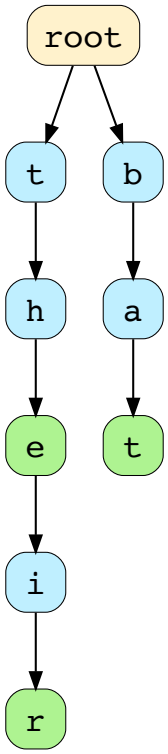
However, if the node contains child branches, this opens up a few scenarios which we will cover below.

Case 1: Word with No Suffix or Prefix

If the word to be deleted has no suffix or prefix and all the character nodes of this word do not have any other children, then we will delete all these nodes up to the root.

However, if any of these nodes have other children (are part of other branch), then they will not be deleted. This will be explained further in **Case 2**.

In the figure below, the deletion of the word `bat` would mean that we have to delete all characters of `bat`.

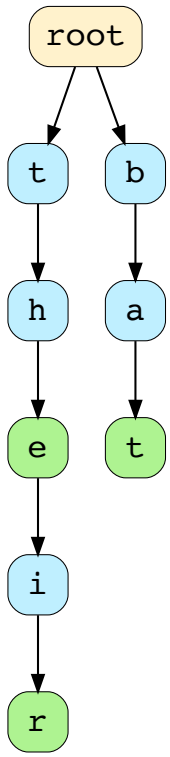


1 of 9

Case 2: Word is a Prefix

If the word to be deleted is a prefix of some other word, then the value of `is_end_word` of the last node of that word is set to `False` and no node is deleted.

For example, to delete the word `the`, we will simply unmark s[illegible]h the word doesn't exist anymore.

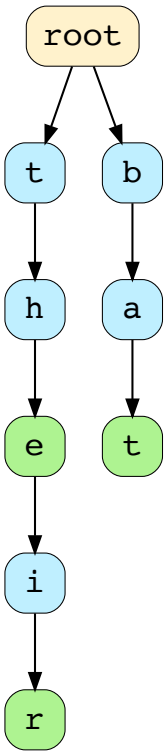


1 of 6

Case 3: Word Has a Common Prefix

If the word to be deleted has a common prefix and the last node of that word does not have any children, then this node is deleted along with all the parent nodes in the branch which do not have any other children and are not end characters.

Take a look at the figure below. In order to delete `their`, we'll traverse the common path up to `the` and delete the characters `i` and `r`.



1 of 15

Implementation

Here's the implementation for the `delete` function in the trie class. We'll explain the code step-by-step as well.

Trie.py

TrieNode.py

```
from TrieNode import TrieNode


class Trie:
    def __init__(self):
        self.root = TrieNode()  # Root node

    # Function to get the index of character 't'
```

```
    def get_index(self, t):
        return ord(t) - ord('a')

    # Function to insert a key in the Trie
    def insert(self, key):
        if key is None:
            return False  # None key

        key = key.lower()  # Keys are stored in lowercase
        current = self.root

        # Iterate over each letter in the key
        # If the letter exists, go down a level
        # Else simply create a TrieNode and go down a level
        for letter in key:
            index = self.get_index(letter)

            if current.children[index] is None:
                current.children[index] = TrieNode(letter)
                print(letter, "inserted")

            current = current.children[index]

        current.is_end_word = True
        print("'" + key + "' inserted")

    # Function to search a given key in Trie
    def search(self, key):
        if key is None:
            return False  # None key

        key = key.lower()
        current = self.root

        # Iterate over each letter in the key
        # If the letter doesn't exist, return False
        # If the letter exists, go down a level
        # We will return true only if we reach the leafNode and
        # have traversed the Trie based on the length of the key

        for letter in key:
            index = self.get_index(letter)
            if current.children[index] is None:
                return False
            current = current.children[index]

        if current is not None and current.is_end_word:
            return True

        return False

    # Recursive function to delete given key
```

```
    def delete_helper(self, key, current, length, level):
        deleted_self = False

        if current is None:
            print("Key does not exist")
            return deleted_self

        # Base Case:
        # We have reached at the node which points
        # to the alphabet at the end of the key
        if level is length:
            # If there are no nodes ahead of this node in
            # this path, then we can delete this node
            print("Level is length, we are at the end")
            if current.children.count(None) == len(current.children):
                print("- Node", current.char, ": has no children, delete it")
                current = None
                deleted_self = True

            # If there are nodes ahead of current in this path
            # Then we cannot delete current. We simply unmark this as leaf
            else:
                print("- Node", current.char, ": has children, don't delete \
                it")
                current.is_end_word = False
                deleted_self = False

        else:
            index = self.get_index(key[level])
            print("Traverse to", key[level])
            child_node = current.children[index]
            child_deleted = self.delete_helper(
                key, child_node, length, level + 1)
            # print( "Returned from", key[level] , "as",  child_deleted)
            if child_deleted:
                # Setting children pointer to None as child is deleted
                print("- Delete link from", current.char, "to", key[level])
                current.children[index] = None
                # If current is a leaf node then
                # current is part of another key
                # So, we cannot delete this node and it's parent path nodes
                if current.is_end_word:
                    print("- - Don't delete node", current.char, ": word end")
                    deleted_self = False

                # If child_node is deleted and current has more children
                # then current must be part of another key
                # So, we cannot delete current Node
                elif current.children.count(None) != len(current.children):
                    print("- - Don't delete node", current.char, ": has \
                    children")
                    deleted_self = False
```

```
                # Else we can delete current
                else:
                    print("- - Delete node", current.char, ": has no children")
                    current = None
                    deleted_self = True

            else:
                deleted_self = False

        return deleted_self

    # Function to delete given key from Trie
    def delete(self, key):
        if self.root is None or key is None:
            print("None key or empty trie error")
            return
        print("\nDeleting:", key)
        self.delete_helper(key, self.root, len(key), 0)


# Input keys (use only 'a' through 'z')
keys = ["the", "a", "there", "answer", "any",
        "by", "bye", "their", "abc"]
res = ["Not present in trie", "Present in trie"]

t = Trie()
print("Keys to insert: \n", keys)

# Construct Trie
for words in keys:
    t.insert(words)

# Search for different keys
print("the --- " + res[1] if t.search("the") else "the --- " + res[0])
print("these --- " + res[1] if t.search("these") else "these --- " + res[0])
print("abc --- " + res[1] if t.search("abc") else "abc --- " + res[0])

# Delete abc
t.delete("abc")
print("Deleted key \"abc\" \n")

print("abc --- " + res[1] if t.search("abc") else "abc --- " + res[0])
```

The `delete` function takes in a key of type string and then checks if either the trie is empty or the key is `None`. For each case, it simply returns from the

function.

`delete_helper()` is a recursive function to delete the given key. Its arguments are a key, the key's length, a trie node (`root` at the beginning), and the `level` (index) of the key.

It goes through all the cases explained above. The base case for this recursive function is when the algorithm reaches the last node of the key:

```
    if level is length:
```

At this point, we check if the last node has any further children or not. If it does, then we simply unmark it as an end word. On the other hand, if the last node doesn't contain any children, all we have to do is to set the parameter `deleted_self` to `True` to mark this node for deletion.

Time Complexity#

If the length of the word is h, the worst-case time complexity is $O(h)$. In the worst case, we have to look at h consecutive levels of a trie for a character in the key being searched for. The presence or absence of each character from the key in the trie can be determined in $O(1)$ because the size of the alphabet is fixed. Subsequently, in the worst case, we may have to delete h nodes from the trie. Thus, the running time of delete in a trie is $O(h)$.

Search in a Trie

Challenge 1: Total Number of Words I...

Completed

Report an Issue