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
  - Case 1: Word with No Suffix or Prefix
  - Case 2: Word is a Prefix
  - Case 3: Word Has a Common Prefix
- Implementation
  - Time Complexity

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.

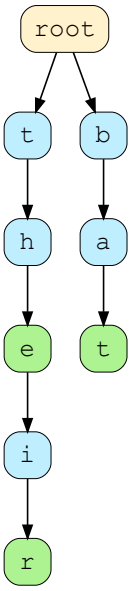
However, if the node contains child branches, this opens up a few scenarios which we will cover below.

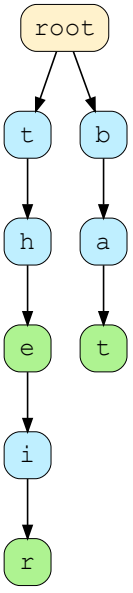
Case 1: Word with No Suffix or Prefix

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.

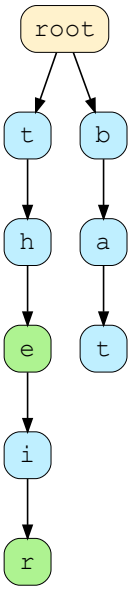
However, if any of these nodes have other children (are part of another branch), then they will not be deleted. This will be explained further in **Case 2**.

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

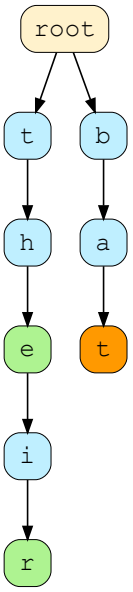




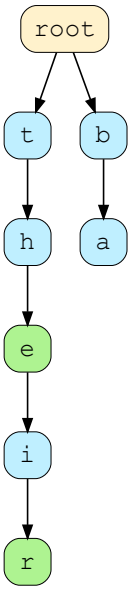
Deleting 'bat'

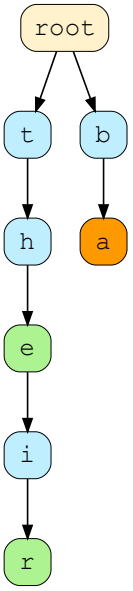


Unmark 't'

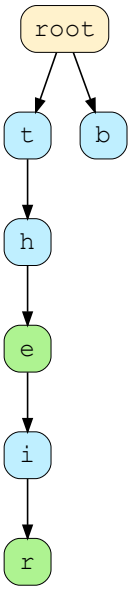


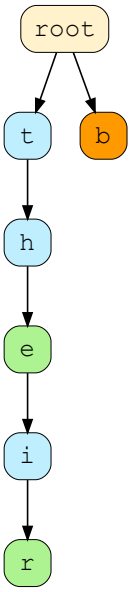
Delete 't'



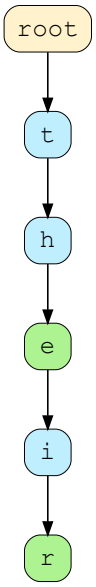


Delete 'a'





Delete 'b'

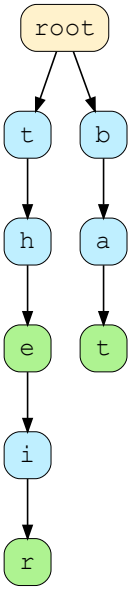


9 of 9

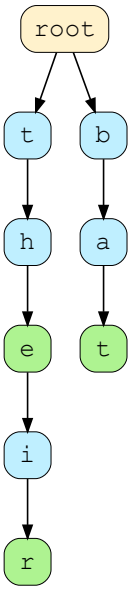
Case 2: Word is a Prefix

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.

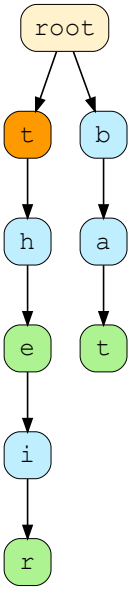
For example, to delete the word `the`, we will simply unmark `e` to show that the word doesn't exist anymore.

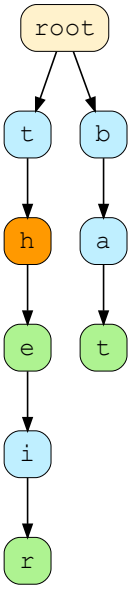


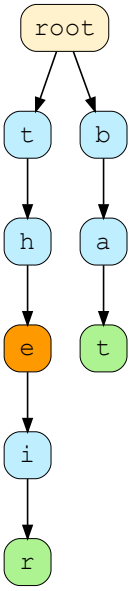
1 of 6

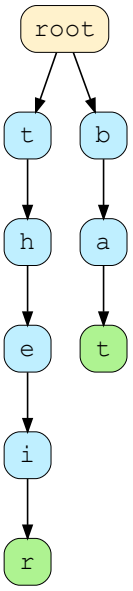


Deleting 'the'




root
t
b
h
a
e
t
i
r


root
t
b
h
a
e
t
i
r



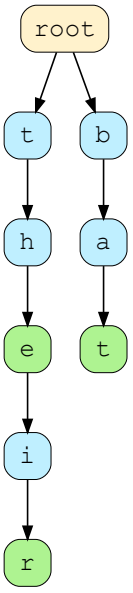
Unmark 'e'

6 of 6

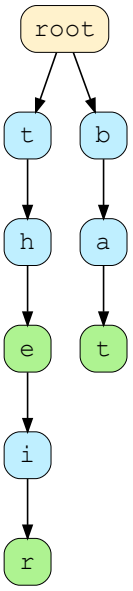
Case 3: Word Has a Common Prefix

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.

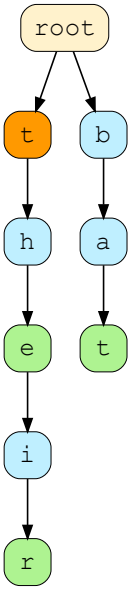
Take a look at the figure below. In order to delete `their`, we'll traverse the common path up to `the` and delete the characters `i` and `r`.

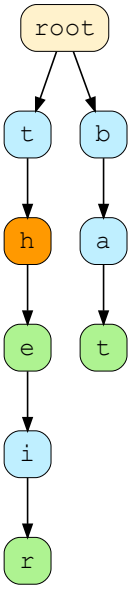


1 of 15

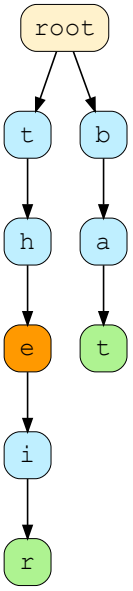


Deleting 'their'

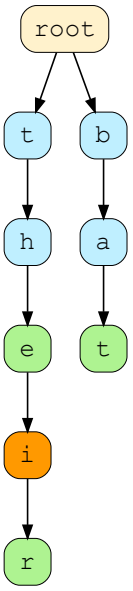


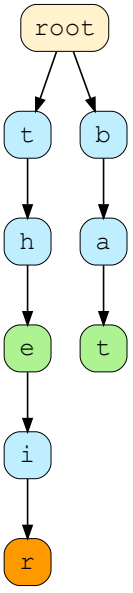


4 of 15

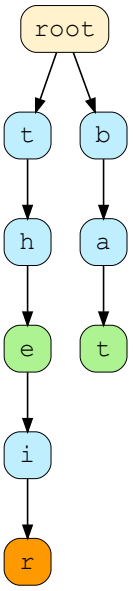


5 of 15

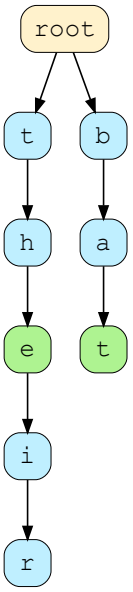


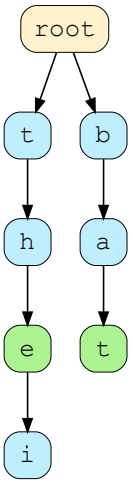


Word found!

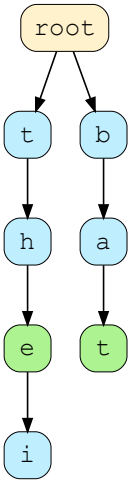


Unmark 'r'

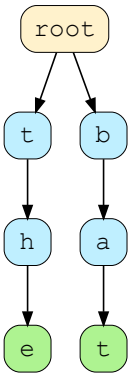




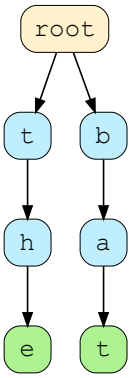
Delete 'r'



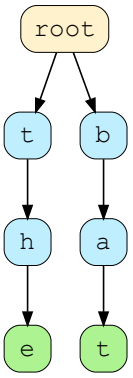
Delete 'r'



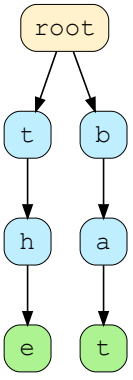
Delete 'i'



Delete 'i'



Do not delete 'e' as its been marked as the end of word



Do not delete 'e' as its been marked as the end of word

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

```
1  om TrieNode import TrieNode
2
3
4  lass Trie:
5      def __init__(self):
6          self.root = TrieNode()  # Root node
7
8      # Function to get the index of character 't'
```

```
    # Function to get the index of character 't'
    def get_index(self, t):
        return ord(t) - ord('a')

    # Function to insert a key into the trie
    def insert(self, key):
        # None keys are not allowed
        if key is None:
            return

        key = key.lower()  # Keys are stored in lowercase
        current_node = self.root
        index = 0  # To store the character index

        # Iterate the trie with the given character index,
        # If the index points to None
        # simply create a TrieNode and go down a level
        for level in range(len(key)):
            index = self.get_index(key[level])

            if current_node.children[index] is None:
                current_node.children[index] = TrieNode(key[level])
                print(key[level] + " inserted")

            current_node = current_node.children[index]

        # Mark the end character as leaf node
        current_node.mark_as_leaf()
        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_node = self.root
        index = 0

        # Iterate the Trie with given character index,
        # If it is None at any point then we stop and return false
        # We will return true only if we reach leafNode and have trave
        # Trie based on the length of the key

        for level in range(len(key)):
            index = self.get_index(key[level])
            if current_node.children[index] is None:
                return False
            current_node = current_node.children[index]

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

        return False

    # Helper Function to return true if current_node does not have any

    def has_no_children(self, current_node):
        for i in range(len(current_node.children)):
            if current_node.children[i] is not None:
                return False
        return True

    # Recursive function to delete given key
    def delete_helper(self, key, current_node, length, level):
        deleted_self = False

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

```

```
        # Base Case:
        # If 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
            if self.has_no_children(current_node):
                current_node = None
                deleted_self = True

            # If there are nodes ahead of current_node in this path
            # Then we cannot delete current_node. We simply unmark thi
            else:
                current_node.unMarkAsLeaf()
                deleted_self = False

        else:
            child_node = current_node.children[self.get_index(key[leve
            child_deleted = self.delete_helper(
                key, child_node, length, level + 1)
            if child_deleted:
                # Making children pointer also None: since child is de
                current_node.children[self.get_index(key[level])] = No
                # If current_node is leaf node then
                # current_node is part of another key
                # So, we cannot delete this node and it's parent path
                if current_node.is_end_word:
                    deleted_self = False

                # If child_node is deleted and current_node has more c
                # then current_node must be part of another key
                # So, we cannot delete currenNode
                elif self.has_no_children(current_node) is False:
                    deleted_self = False

                # Else we can delete current_node
                else:
                    current_node = 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

        self.delete_helper(key, self.root, len(key), 0)


 Input keys (use only 'a' through 'z' and lower case)
eys = ["the", "a", "there", "answer", "any", "by", "bye", "their", "a
utput = ["Not present in trie", "Present in trie"]

 = Trie()
rint("Keys to insert: ")
rint(keys)

 Construct Trie
or key in keys:
   t.insert(key)

 Search for different keys
f t.search("the") is True:
   print("the --- " + output[1])
lse:
   print("the --- " + output[1])
```

```
150
151 f t.search("these") is True:
152     print("these --- " + output[1])
153 lse:
154     print("these --- " + output[0])
155
156 f t.search("abc") is True:
157     print("abc --- " + output[1])
158 lse:
159     print("abc --- " + output[1])
160
161 .delete("abc")
162 int("Deleted key \"abc\"")
163
164 f t.search("abc") is True:
165     print("abc --- " + output[1])
166 lse:
167     print("abc --- " + output[0])
168
```

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 it to `None` and move back up in the trie to check for the remaining nodes.

Time Complexity

Since we iterate over the whole key of size **n**, deletion works in *O(n)*.

And now that we have covered all the nitty-gritty details of the trie data structure, let's try to solve some practice questions related to tries.

Back

Search in a Trie

Next

Challenge 1: Total Number of Words i...

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/deletion-in-trie__trie__data-structures-for-coding-interviews-in-python)