Search in a Trie

This lesson defines the algorithm for a word search in a trie. It also highlights the different scenarios which are taken care of in the algorithm.

We'll cover the following

Search Algorithm

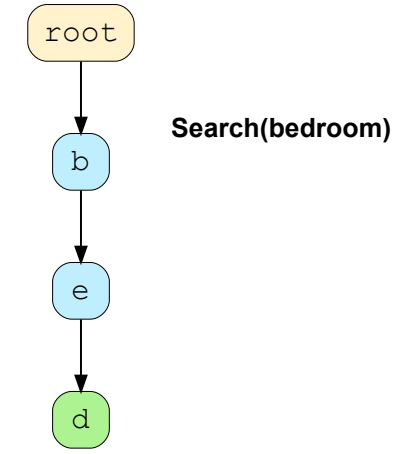
If we want to check whether a word is present in the trie or not, we just need to keep tracing the path in the trie corresponding to the characters/letters in the word.

The logic isn't too complex, but there are a few cases we need to take care of.

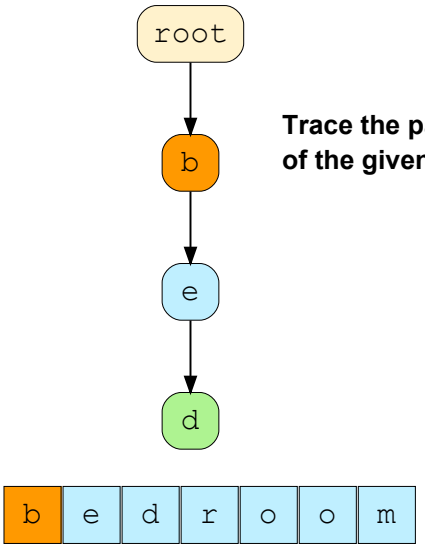
Case 1: Non-Existent Word

If we are searching for a word that doesn't exist in the trie and is not a subset of any other word, by principle, we will find `None` before the last character of the word can be found.

For a better understanding, check out the illustration below:

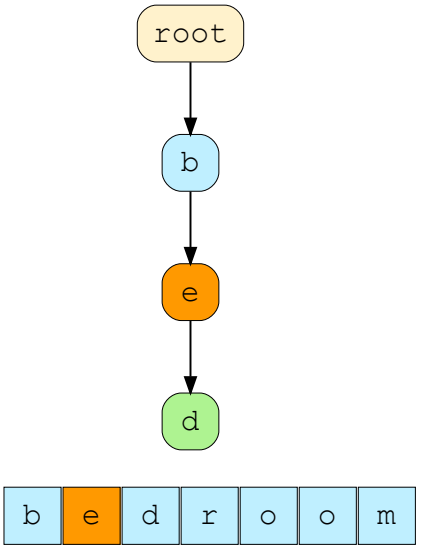


1 of 5

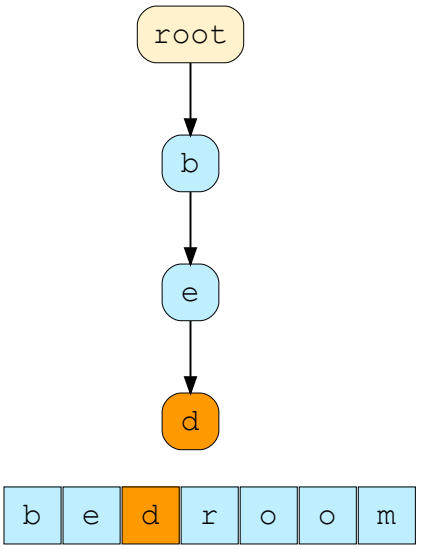


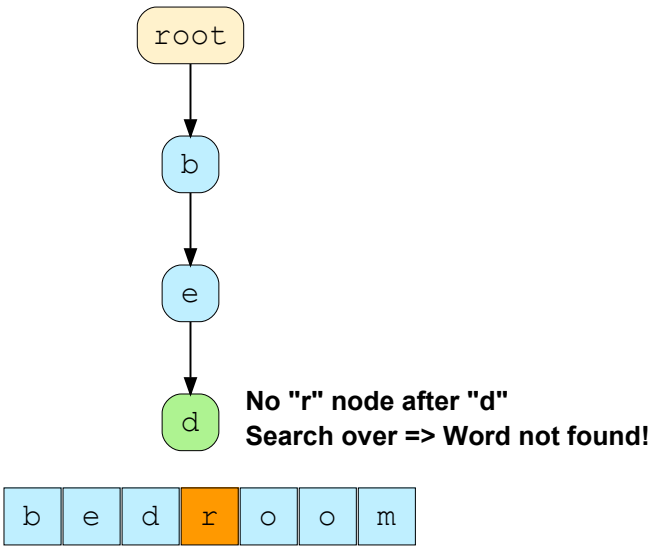
Trace the path till the length of the given word "bedroom"

2 of 5



3 of 5

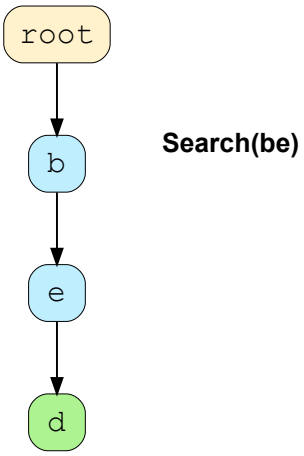


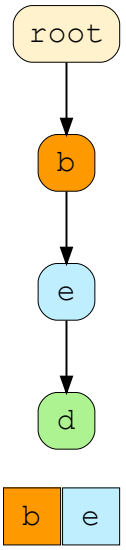


Case 2: Word Exists as a Substring

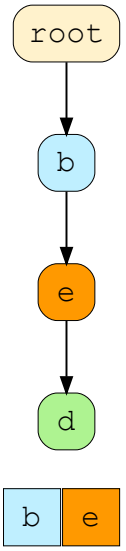
This is the case where our word can be found as a substring of another word, but the `isEndWord` property for it has been set to `False`.

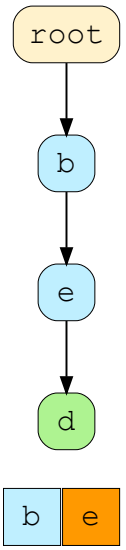
In the example below, we are searching for the word `be`. It is a subset of the already existing word `bed`, but the `e` node has not been flagged as the end of a word. Hence, `be` will not be detected.





Trace the path till the length of the given word "be"

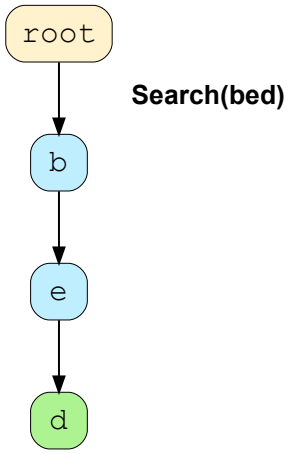




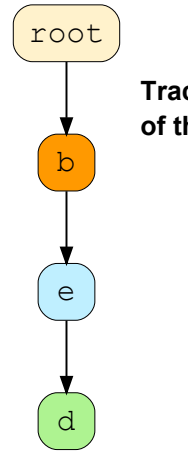
isEndWord is not set for 'e', indicating that 'be' does not exist in this Trie

Case 3: Word Exists

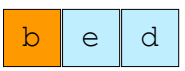
The success case is when there exists a path from the root to the node of the last character and the node is also marked as `isEndWord`:

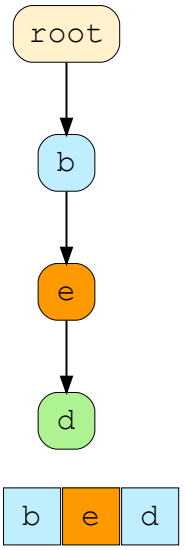


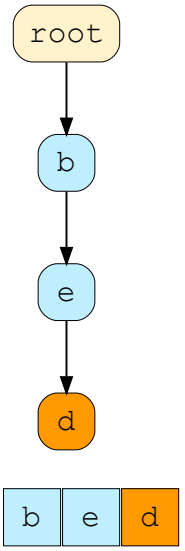
Search(bed)

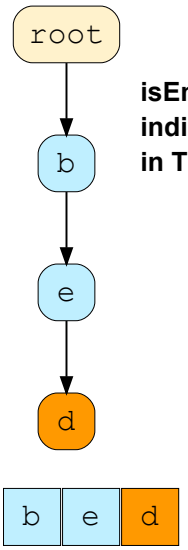


Trace the path till the length of the given word "bed"









**isEndWord is set for node "d"
indicating that "bed" exists
in Trie => Word found!**

Implementation

You can find the implementation for the `search` function below. We'll discuss it in detail afterwards.

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 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 tra
        # 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
```

```
61          return False
62
63      # Function to delete given key from Trie
64      def delete(self, key):
65          pass
66
67
68  # Input keys (use only 'a' through 'z' and lower case)
69  keys = ["the", "a", "there", "answer", "any",
70          "by", "bye", "their", "abc"]
71  output = ["Not present in the trie", "Present in the trie"]
72
73  t = Trie()
74  print("Keys to insert: ")
75  print(keys)
76
77  # Construct Trie
78  for i in range(len(keys)):
79      t.insert(keys[i])
80
81  # Search for different keys
82  if t.search("the") is True:
83      print("the --- " + output[1])
84  else:
85      print("the --- " + output[0])
86
87  if t.search("these") is True:
88      print("these --- " + output[1])
89  else:
90      print("these --- " + output[0])
91
92  if t.search("abc") is True:
93      print("abc --- " + output[1])
94  else:
95      print("abc --- " + output[0])
96
```

The function takes in a string `key` as an argument and returns `True` if the `key` is found. Otherwise, it returns `False`.

As we know from insertion, `None` keys aren't allowed and all characters are stored in lowercase.

Beginning from the root, we will traverse the trie and check if the sequence of characters is present. Another thing we need to make sure is that the last character node has the `isEndWord` flag set to `True`. Otherwise, we will fall into **Case 2**.

Time Complexity

Just like insertion, search works in *O(n)* where **n** is the number of letters in the word.

Now that we've covered insertion and search, we'll move on to word deletion in a trie.

Back

Insertion in a Trie

Next

Deletion in Trie

Mark as Completed