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

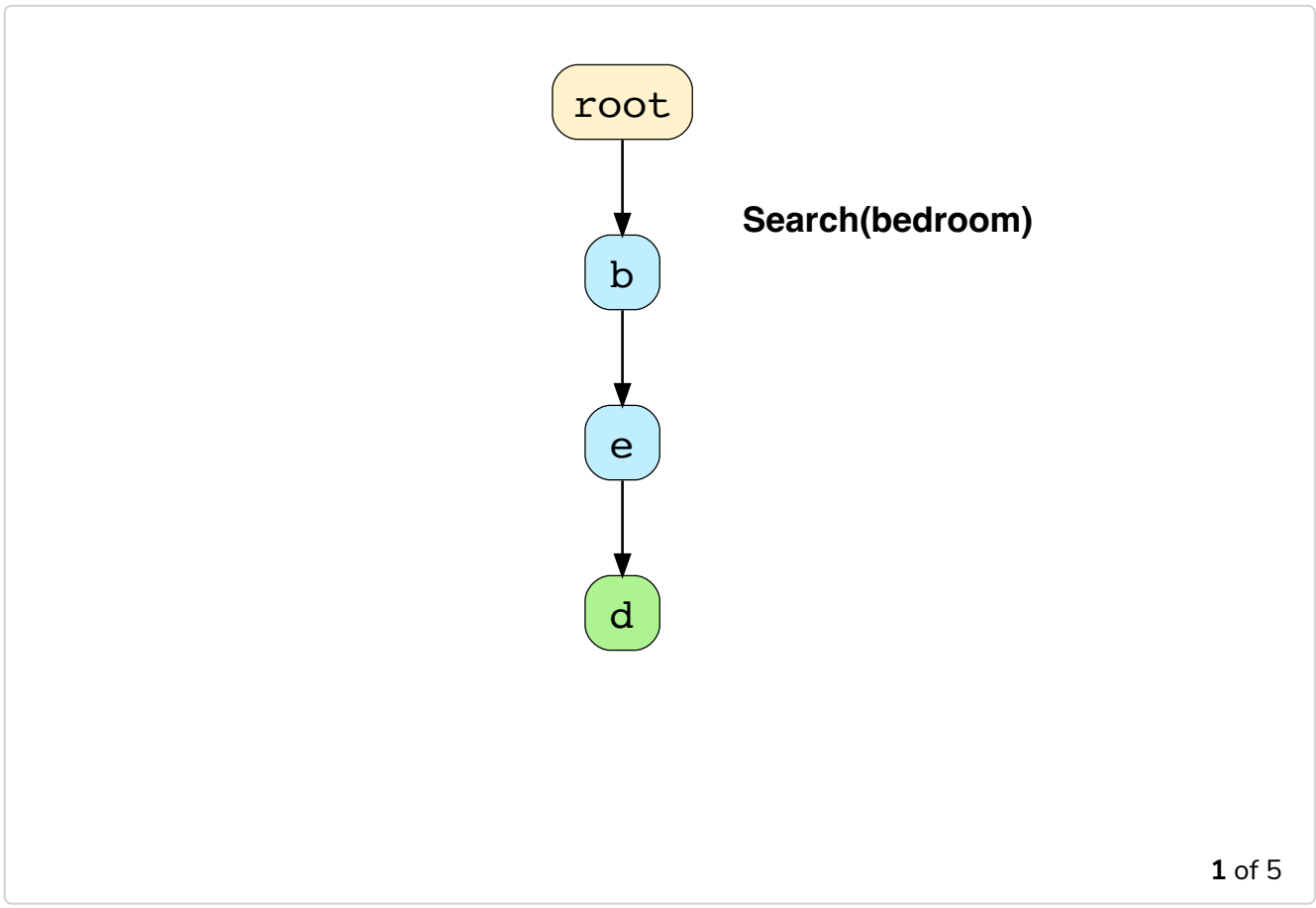
If we want to check whether a word is present in the trie or not, we just need to keep tracing the path in the trie corresponding to the characters/letters in the word.

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.

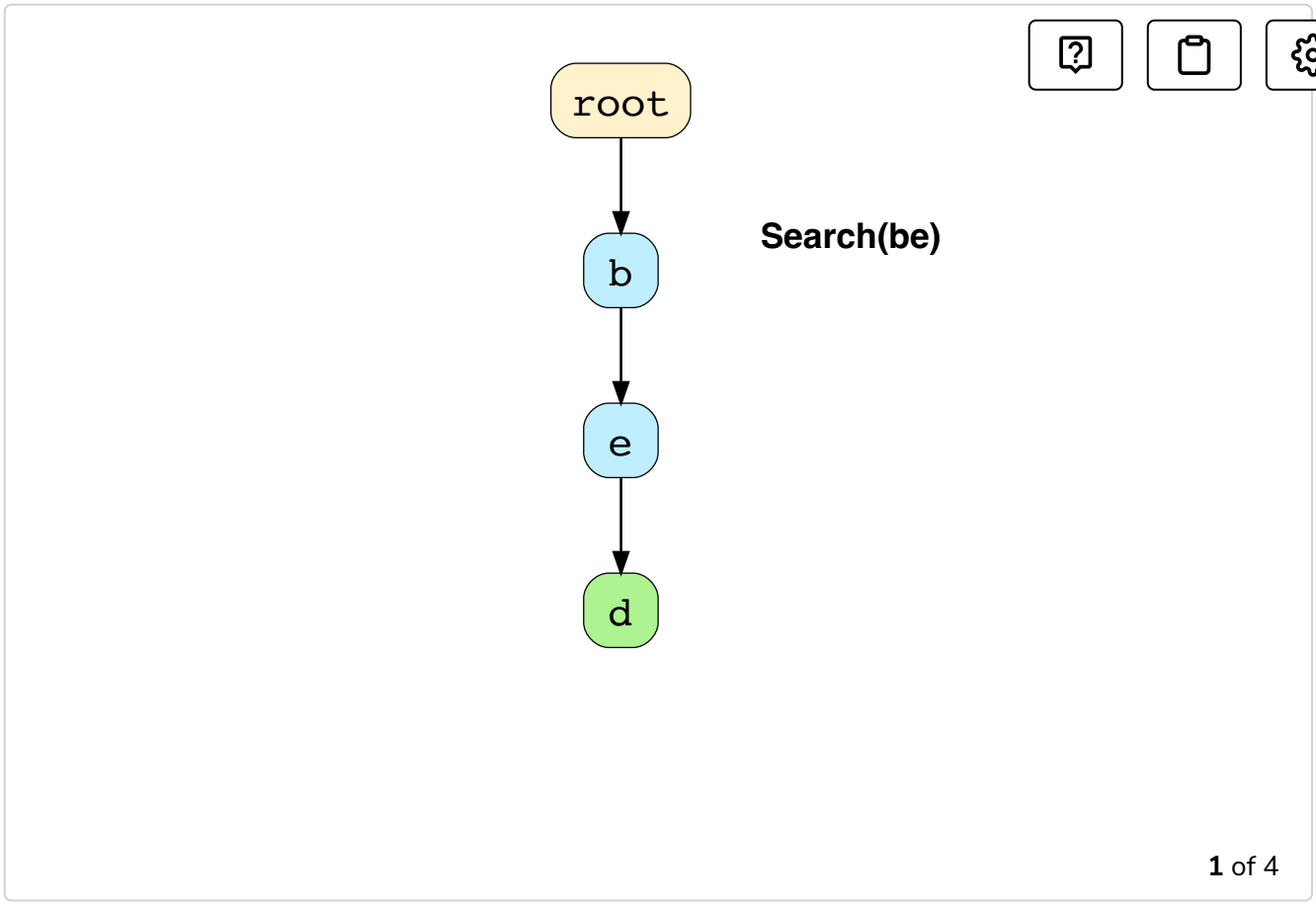
For a better understanding, check out the illustration below:



Case 2: Word Exists as a Substring

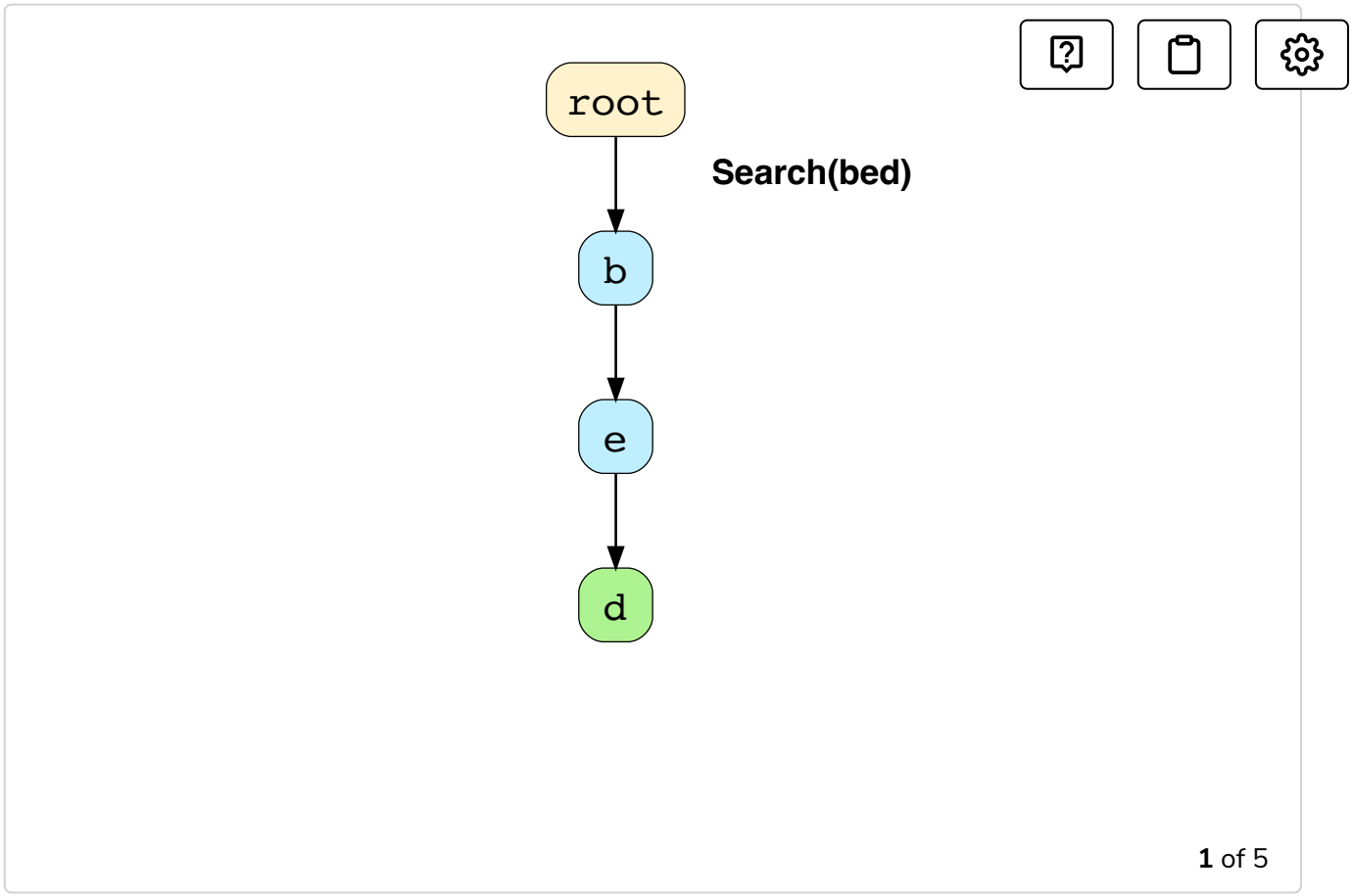
This is the case where our word can be found as a substring of another word, but the `isEndWord` property for it has been set to `False`.

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.



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`:



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 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

    # Function to delete given key from Trie
```

```
    def delete(self, key):
        pass


# 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])
```

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

Much like the insertion process, `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#

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 fror

the key in the trie can be determined in $O(1)$ because the siz h alphabet is fixed. Thus, the running time of search in a trie is $O(h)$.