Insertion in a Trie

This lesson defines all the cases needed for inserting a word into a trie, along with the Pythonic implementation.

We'll cover the following

Word Insertion

The insertion process is fairly simple. For each character in the key, we check if it exists at the position we desire. If the character is not present, then we insert the corresponding trie node at the correct index in `children`. While inserting the last node, we also set the value of `isEndWord` to `True`.

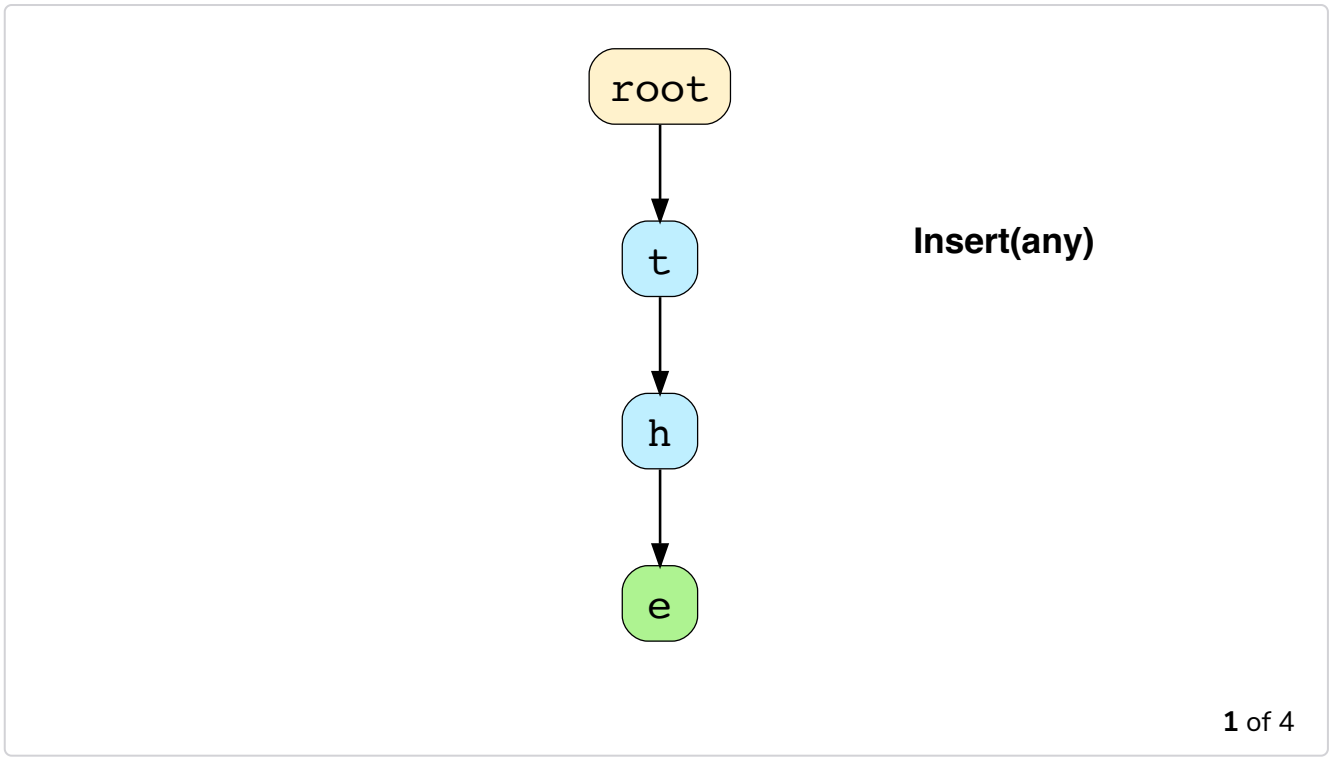
There are three primary cases we need to consider during insertion. Let's discuss them now.

Case 1: No Common Prefix

In this situation, we want to insert a word whose characters are not comm with any other node path.

The illustration below shows the insertion of `any` in a trie w[illegible] only `the`.

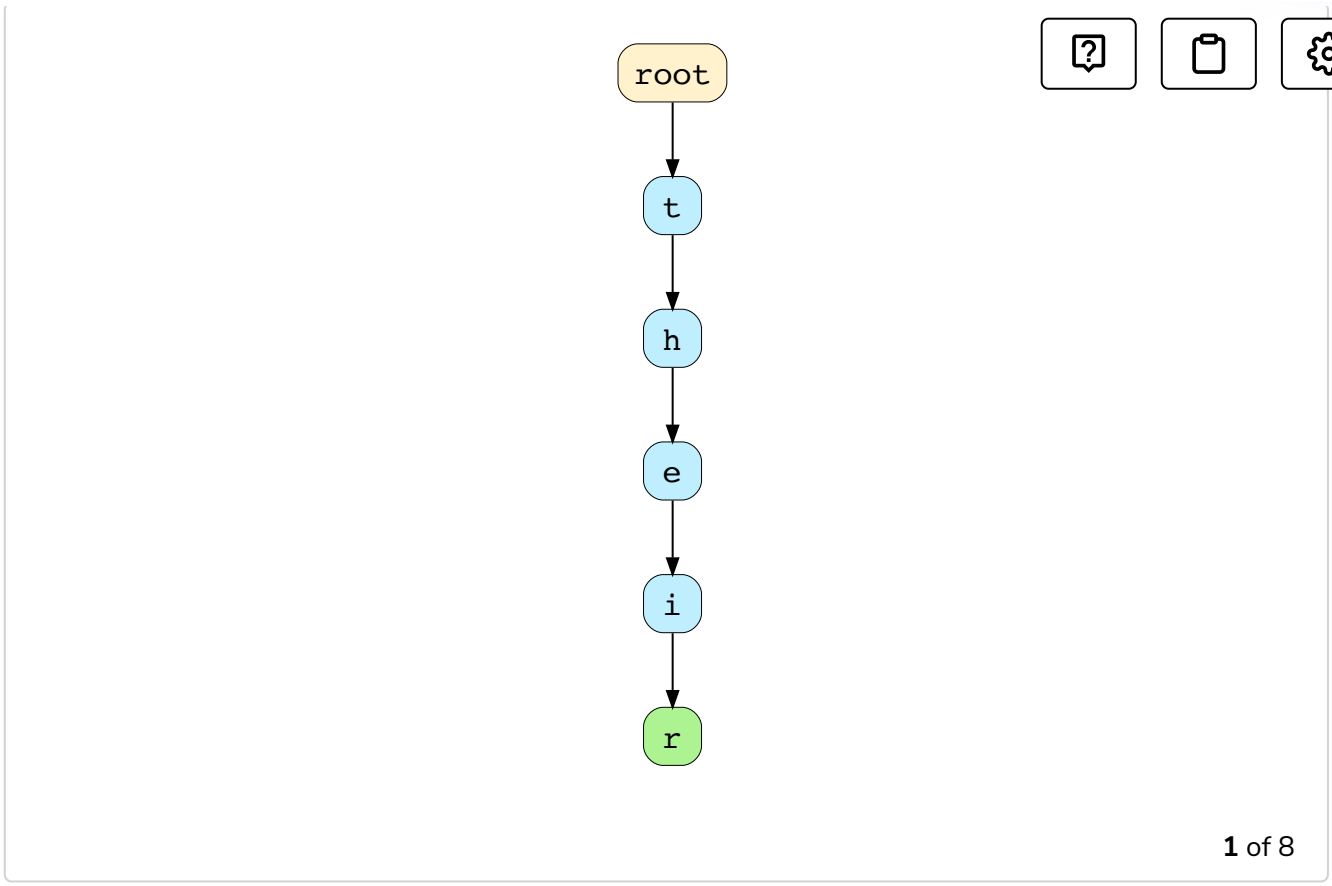
We need to create nodes for all the characters of the word `any` as there is no common subsequence between `any` and `the`.



Case 2: Common Prefix

This occurs when a portion of the starting characters of your word already in the trie starting from the `root` node.

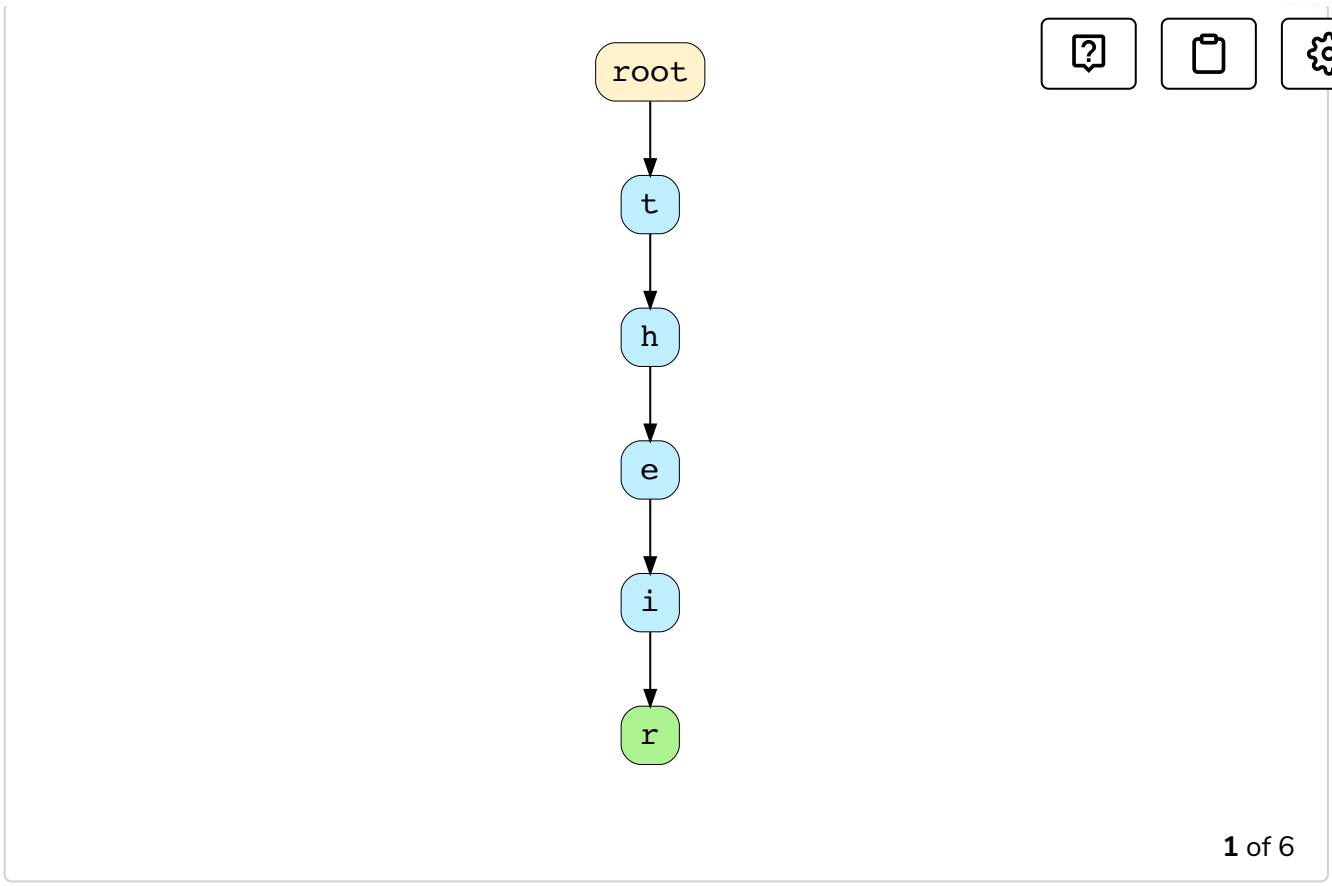
For example, if we want to insert a new word `there` in the trie which consists of a word `their`, the path till `the` already exists. After that, we need to insert two nodes for `r` and `e` as shown below.



Case 3: Word Exists

This occurs if your word is a substring of another word that already exists in the trie.

For example, if we want to insert a word `the` in the trie which already contains `their`, the path for `the` already exists. Therefore, we simply need to set the value of `isEndWord` to true at `e` in order to represent the end of the word for `the` as shown below.



Implementation

Here is the implementation of the `insert` function based on the three cases we've seen.

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):
        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"]

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

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

The function takes in a string `key` indicating a word. `None` k re allowed, and all keys are stored in *lowercase*.

To make things easier, we use the `get_index()` method to return the index of a character. The `get_index` method subtracts the ordinal value of *'a'* from the character to return a numerical value in the range of 0 to 25.

To insert a `key`, we iterate over the characters in the `key`. For each character, we check if a `TrieNode` exists for it. If it does not exist, we insert a new `TrieNode` in the `children` array at the index returned by the `get_index` function. However, if a `TrieNode` already exists, we simply move on to the next character by setting our `current` node to the `TrieNode` at the character's index.

Once we have iterated over all the letters, we mark the last node as leaf since the word has ended.

Time Complexity#

For a key with **n** characters, the worst case time complexity turns out to be *O(n)* since we need to make **n** iterations.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. See how

Back

Structure of a Trie

Next

Search in a Trie

Comple