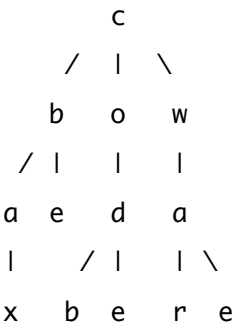
Daily Coding Problem #348

Problem

This problem was asked by Zillow.

A ternary search tree is a trie-like data structure where each node may have up to three children. Here is an example which represents the words code, cob, be, ax, war, and we.



The tree is structured according to the following rules:

- left child nodes link to words lexicographically earlier than the parent prefix
- right child nodes link to words lexicographically later than the parent prefix
- middle child nodes continue the current word

For instance, since code is the first word inserted in the tree, and cob lexicographically precedes cod, cob is represented as a left child extending from cod.

Implement insertion and search functions for a ternary search tree.

Solution

Let's begin by creating a class to represent a tree node, just as if we were creating a binary tree. In this case, we will declare three child nodes instead of two, and add an additional variable that tells us if we have reached the end of a word.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.mid = None
        self.right = None
        self.end = False
```

With this in place, we can think about how to define our insertion function. If there are currently no nodes in our tree, the process is straightforward: we keep adding `mid` child nodes with each consecutive letter until we spell out the whole word, and set the last child's `end` value to be `True`.

To handle more complicated cases, it helps to think recursively. At each step, we will consider two objects: the current node, and the remainder of the word that must be inserted. If the current node is `None`, we find ourselves in the case above. Otherwise, we can consider three possibilities:

- If the first letter of the word remainder precedes the current node's letter, we branch left.
- If the first letter of the word remainder succeeds the current node's letter, we branch right.
- If the two are equal, we continue to the middle child node with the tail of our word.

Since this recursive process applies to an arbitrary node and suffix, we will call it as a helper from our main `insert` function, which updates the root of our tree.

```
class TernaryTree:
    def __init__(self):
        self.root = None

    def _insert(self, node, word):
        if not word:
            return node

        head, tail = word[0], word[1:]
        if not node:
            node = Node(head)

        if head < node.data:
            node.left = self._insert(node.left, word)
        elif head > node.data:
            node.right = self._insert(node.right, word)
        else:
            if not tail:
                node.end = True
            else:
                node.mid = self._insert(node.mid, tail)

        return node

    def insert(self, word):
        self.root = self._insert(self.root, word)
```

For each word of length M that we insert, we must traverse down M middle child nodes of our tree, and create these nodes if they do not already exist. Therefore, the space and time complexity of `insert` will be O(M).

It is likely that we will traverse more than M nodes, especially if the tree has many words already inserted. For example, suppose we have already inserted a word beginning with every letter of the alphabet to a tree whose root letter is *a*. If we now want to insert the word *zebra*, we will pass to the right child 25 times before dropping to a middle child. However, assuming we are only inserting lowercase words in English, the number of such passes at each level has a constant upper bound of 25, and therefore this will not affect our big-O complexity.

Traversing the tree to search for a given word follows a very similar process.

We will define a helper function that recursively searches for each suffix of the word, until we run out of letters. If we reach a leaf node of our tree without being able to finish the word, the word could not have been inserted. Otherwise, if we exhaust all the letters and find that the final node has its `end` value set to `True`, we can return `True`.

As above, we will call this helper function by applying it to the root of our tree.

```
class TernaryTree:
    def __init__(self):
        self.root = None

    ...

    def _search(self, node, word):
        if not node or not word:
            return False

        head, tail = word[0], word[1:]
        if head < node.data:
            return self._search(node.left, word)
        elif head > node.data:
            return self._search(node.right, word)
        else:
            if not tail and node.end:
                return True
            return self._search(node.mid, tail)

    def search(self, word):
        return self._search(self.root, word)
```

For a word of length `M`, we must again traverse at most `M` middle child nodes, so the time complexity of `search` will be `O(M)`. The space required for this operation is `O(M)` as well, since we create new variables at each level of the stack.