Tries

Tries (pronounced as "try", not "tree") are a data structure developed for re**trie**val systems. It sometimes works better than hash-based solutions. It is usually used to store strings.

Definition of Tries

A trie is a tree whose nodes contain letters. The leaves contain an end-of-record marker. We will use '#' for this purpose. When following a path from the root down to a leaf or another interior node, we construct a single string. Descendants of any node share a common prefix.

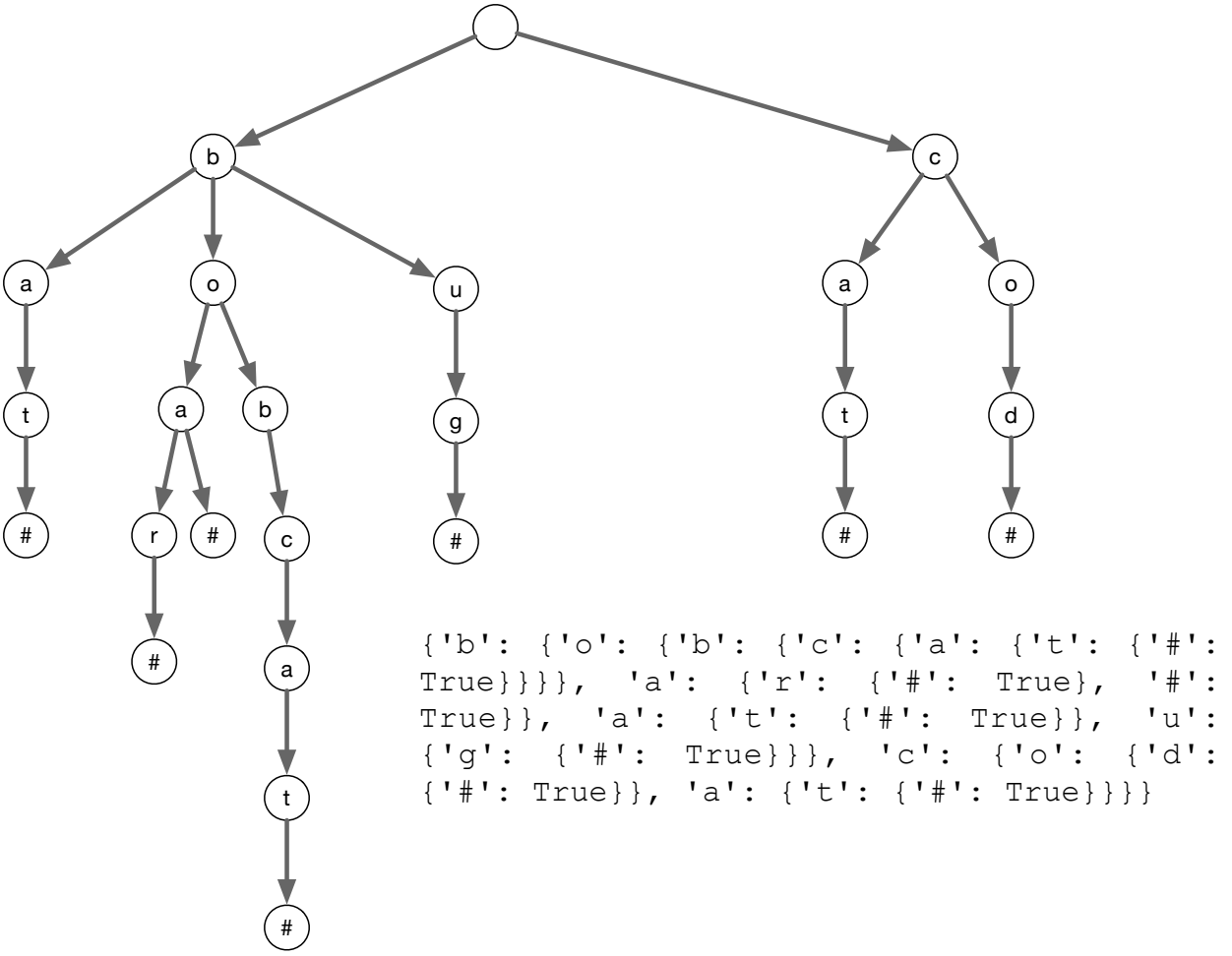

Fig. 1: A trie containing the strings "bat", "boar", "boa", "bobcat", "bug", "cat", "cod". All leaves are adorned with an end of word marker '#'.

An implementation of a trie needs to insert a word and to find a word, checking if a word or a prefix exists in the tree. All of these methods will run in time $O(k)$.

Tries can be implemented in several ways, but in the context of a coding interview, where you might have only time to sketch a solution or you might have an hour to come up with code, using a Python dictionary is a simple way. However, if you ever have to implement a trie with a large number of prefixes, Python dictionaries are not the way to go. (If you are curious and have too much time on your hand, try to understand Witold Litwin's trie implementation using hashing).

The idea of the dictionary implementation is to have the keys be letters and the values be dictionaries representing the tree formed by the descendants of a node with that letter. In Fig. 1, the letter 'c' in the right child of the root corresponds to a sub-dictionary `{'o': {'d': {'#': True}}, 'a': {'t': {'#': True}}}`, from which we can find that the words "cod#" and "cat#" are the only words starting with "c" in the trie.

Implementation

Besides the two main methods, we also implement a third method that returns all strings with a given prefix in the trie.

```
class Trie:
    def __init__(self):
        self.trie = { }

    def insert(self, text):
        trie = self.trie
        for letter in text:
            if letter not in trie:
                trie[letter] = {}
            trie = trie[letter]
        trie['#'] = True

    def find(self, prefix):
        trie = self.trie
        for letter in prefix:
            if letter in trie:
                trie = trie[letter]
            else:
                return None
        return trie
```

```
    def _elements(self, d):
        result = [ ]
        for c,v in d.items():
            if c=='#':
               subresult = ['']
            else:
               subresult = [c+s for s in self._elements(v)]
            result.extend(subresult)
        return result

if __name__=='__main__':
    t = Trie()
     for w in ['bobcat', 'boar', 'boa', 'bat', 'bug', 'cod',
'cat']:
        t.insert(w)
    print(t.find('c'))
    print(t._elements(t.find('c')))
    print(t._elements(t.find('ca')))
    print(t.trie)
```

Your Task

(1) Use this try to implement an auto-complete function for a web-site. You insert words, and when a word is typed in, you write out all possible words in the trie for every letter of the word that you enter.
(2) Write a function that returns the minimum and maximum (using alphabetic ordering) of all words in the trie with a given prefix.

Deliverable:

1. Your complete code, including unit tests. Your unit tests should include number of words in the hundreds.
2. A sample interaction of the auto-complete.