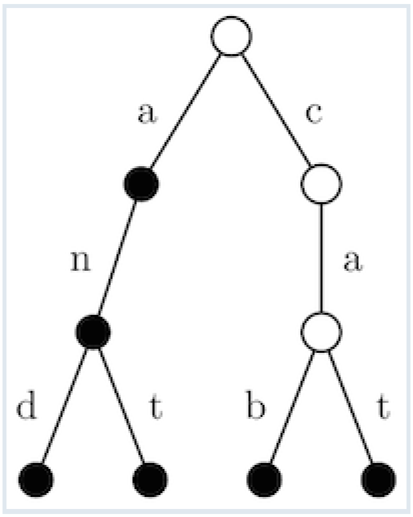
The context for this activity is a spell checker program. Spell checkers need to store a spelling dictionary in a way that makes it as fast as possible to see whether a word is in the dictionary. The data structure we will use to store the dictionary is called a trie, which is derived from the middle letters of the word retrieval. Tries were invented many years ago to speed information retrieval tasks.

A trie is a tree whose edges are labeled with letters. Paths from the root of the trie spell out words. The image below shows a trie.



The black nodes indicate that the path to this point in the trie spells a word. The words represented by this trie are a, an, and, ant, cab and cat. It is very fast to check whether a word is in a trie: one need only start from the root and trace out the word along the edges; if one ends up at a word-terminating node (a black node), then the word is in the trie, otherwise it is not. A trie is very good for spell checking: simply construct a trie representing all the words in a dictionary, and then check each word in a document to see if it is in the trie. The words not in the trie are (presumably) misspelled.

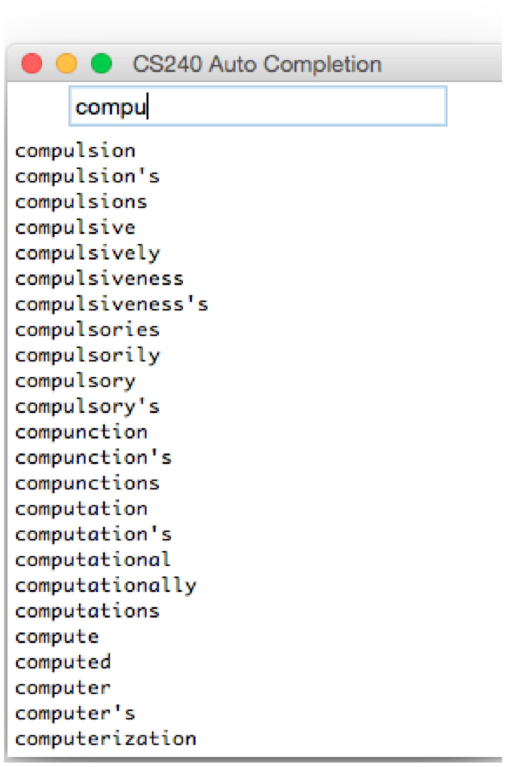
Tries can also efficiently support auto-completion. Given the prefix-based structure of the trie, it is straightforward to enumerate all of the words that begin with a given letter sequence: these will be represented by nodes that are descendants of the node corresponding to that prefix.

There are many ways to implement tries. We will use a simple implementation that uses a lot of space, but is fast. Each node of the trie has a Boolean field indicating whether a string ending there is a word (corresponding to whether the node is black or white in the diagram). Each node also contains a Python dictionary representing outgoing edges. The dictionary keys are individual letters, and the values are references to other trie nodes.

Your task is to complete the Trie class in *trie.py*

```
class Trie:
   """Trie class that supports spell checking and auto-completion"""

   def __init__(self):
       """Create a new empty Trie"""
       ### Your code here
       pass

   def __len__(self):
       """Return the number of words stored in this Trie"""
       ### Your code here
       pass

   def __contains__(self, word):
       """Returns True if word is in the Trie; False otherwise"""
       ### Your code here
       pass

   def __iter__(self):
       """Returns an iterator that will allow iteration over all words in the
Trie in lexicographical (alphabetical)
       order
       """
       ### Your code here
       pass

   def insert(self, word):
       """Insert word into the Trie. If the word is already in the Trie; do
nothing. This method has no return value"""
       pass

   def contains_prefix(self, prefix):
       """Return True if the string is a word in the trie or is a prefix of any
word in the True. Return False otherwise
       """
       ### Your code here
       pass
   def prefix_iter(self, prefix):
       """Returns an iterator that will allow iteration over all the words in
the Trie that have the indicated prefix,
       in lexicographic (alphabetical) order
       """
       ### Your code here
       pass
```

You are also provided with an example spell-checking program that will use your implementation of the Trie class. To run the program, run *auto_complete.py.* Here is an example screenshot of the program



```
compu
compulsion
compulsion's
compulsions
compulsive
compulsively
compulsiveness
compulsiveness's
compulsories
compulsorily
compulsory
compulsory's
compunction
compunction's
compunctions
computation
computation's
computational
computationally
computations
compute
computed
computer
computer's
computerization
```

To get full credit for this task, *auto_complete.py* should run without errors and show a similar output to the screenshot given above.