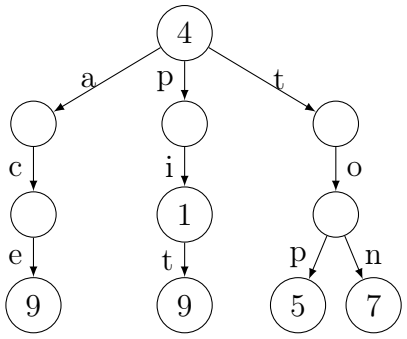
CSCC24 & CSC324 Winter 2018 – Assignment 1
Due: Sunday, February 3, midnight
This assignment is worth 10% of the course grade.

In this assignment, you will work in Haskell with a data structure given by a recursive data type and an abstract data type from the standard library, and you will write an interesting recursive function over it. Note that you should also aim for reasonably efficient algorithms and reasonably good and simple coding style, as usual.

A "trie" is a data structure that implements a dictionary (finite map) that uses strings for keys. The name "trie" came from "retrieval". Most confusingly (when you pronounce it), it is a tree structure indeed. Here is an example (called *albertTrie* in the starter code): On the left is the dictionary represented, in the middle is the trie picture, and on the right is how the provided *printTrie* function prints it.

key	value
(the empty string)	4
ace	9
pi	1
pit	9
top	5
ton	7




```
"" : 4
  a
    c
      e : 9
  p
    i : 1
      t : 9
  t
    o
      n : 7
      p : 5
```

In general, a trie node has a possible value and a collection of character-labelled edges/pointers to child nodes. Its meaning is: A trie maps a string k to a value v iff starting from the root node, traversing the path that matches the character sequence k, if it exists, reaches a node that has the value v.

We use Data.Map from the standard library to model the collection of character-labelled pointers to children. (And in this high-level setting, pointers are implicit, we just talk about the children directly.) So a trie is coded as:

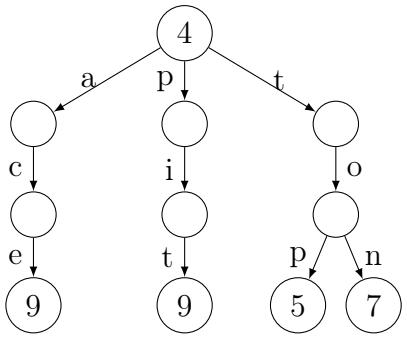
```
data Trie a = TrieNode (Maybe a) (Map Char (Trie a))
```

with a type parameter so the user can choose the value type.

In this assignment, you will implement trie deletion. Since the trie is immutable in functional programming, deletion in this context means producing a new trie rather than changing the original trie. Deleting a key k from a trie means that the new trie no longer maps k to any value, but all other keys and values are still present. It may be better to rephrase "delete k from the trie t" as "compute t minus k".

Example 1: *trieDelete* "pi" *albertTrie*:

key	value
(the empty string)	4
ace	9
pit	9
top	5
ton	7

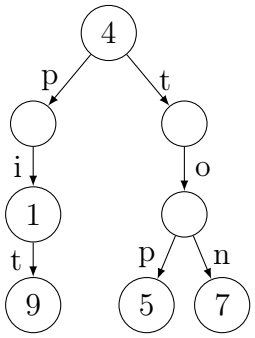



```
"" : 4
  a
    c
      e : 9
  p
    i
      t : 9
  t
    o
      n : 7
      p : 5
```

So the new trie no longer maps "pi" to a value. However, in this example, you cannot lose the node under i, since you still need it to hold the t edge for a child that contains actual data. You can only clear out the value field in the node.

Example 2: *trieDelete* "ace" *albertTrie*:

key	value
(the empty string)	4
pi	1
pit	9
top	5
ton	7



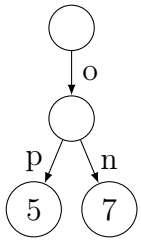

```
"" : 4
  p
    i : 1
      t : 9
  t
    o
      n : 7
      p : 5
```

So the new trie no longer maps "ace" to a value. Note that in this example, you are supposed to also lose the whole path for "ace", since that path would no longer hold actual data. To this end, the function *trieIsEmpty* is provided to help you detect nodes that hold no value and no children.

Example 3: *trieDelete* "foo" *albertTrie*

In this case, since "foo" does not map to any value in the original tree, the new tree is the same as the original tree. (A little bit of cloning is OK if it makes your code simpler.)

Strategy suggestion: Direct recursion works great for this task. For example,
deleteTrie "tone" *albertTrie*
makes a recursive call to
deleteTrie "one" *child*
where *child* is the following subtree:


o
p
n
5
7