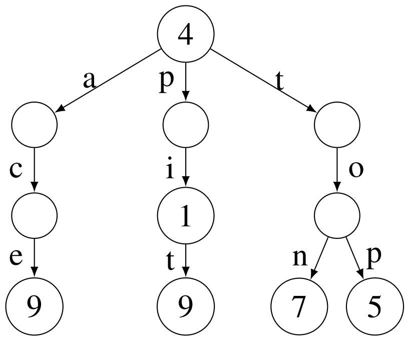
CSC324 Fall 2020 — Assignment 2
Due: Saturday, December 5, 9PM

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

Trie

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
(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 value or no value, and a collection of character-labelled edges/pointers to child nodes. A trie works as a key-value dictionary by: It maps a string *k* to a value *v* iff starting from the root node, traversing the path that matches the character sequence *k* is successful and reaches a node that has the value *v*.

We use the standard library type *Maybe a* to represent the possible absence and presence of a value. We use a list of (*char*, *node*) tuples to represent the collection of character-labelled [pointers to] children. So a trie is coded as:

```
data Trie a = TrieNode (Maybe a) [(Char, Trie a)]
```

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

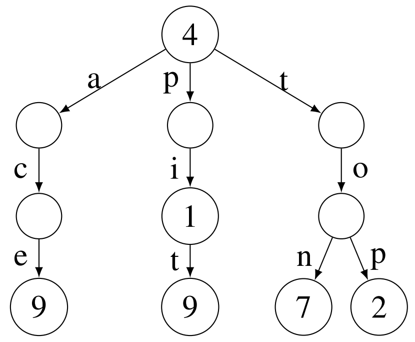
In this assignment, you will implement trie insertion and deletion in the file TrieInsertDelete.hs; TrieDef.hs has the definitions and helpers, testTrieInsertDelete.hs has sample test cases.

Since the trie is immutable in functional programming, our insertion and deletion mean producing a new trie rather than changing the original trie. Our insertion has replacement semantics: If the original trie already maps *k* to some value, insertion produces a new trie that maps *k* to the new value.

Insertion Examples

Insertion example 1: *trieInsert* "top" 2 *albertTrie*. The old trie mapped "top" to 5; the new trie maps "top" to 2.

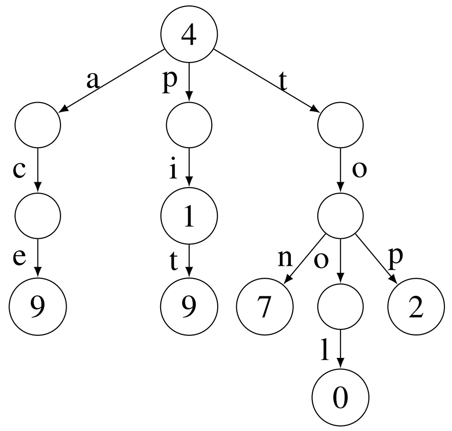
key	value
(empty string)	4
ace	9
pi	1
pit	9
top	2
ton	7



```
"" : 4
  a
    c
      e : 9
  p
    i : 1
      t : 9
  t
    o
      n : 7
      p : 2
```

Insertion example 2: *trieInsert* "tool" 0 *albertTrie*. Inserting a key like this can cause new child nodes, even new paths, to appear.

key	value
(empty string)	4
ace	9
pi	1
pit	9
tool	0
top	2
ton	7

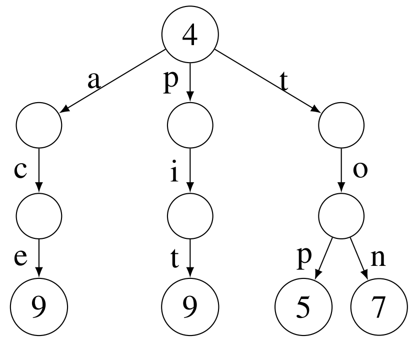


```
"" : 4
  a
    c
      e : 9
  p
    i : 1
      t : 9
  t
    o
      n : 7
      o
        l : 0
      p : 5
```

Deletion Examples

Deletion example 1: *trieDelete* "pi" *albertTrie*. In this case, you lose a value, but you lose no nodes, because the affected node still has descendents that hold values (in this example the child via the label "t" still has a value).

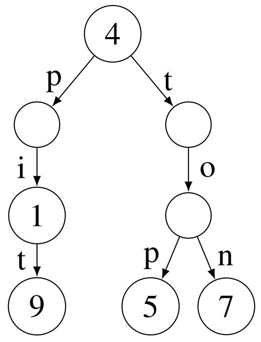
key	value
(empty string)	4
ace	9
pit	9
top	5
ton	7



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

Deletion example 2: *trieDelete* "ace" *albertTrie*. You are required to prune nodes and paths wherever a node has no value and has no descendents that hold values. Algorithmically, you can think and implement this in the bottom-up direction: If a node has no value and no child, lose it; repeat at the parent.

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



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

Deletion 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 bit of cloning is OK if it makes your code simpler.)