We've seen binary trees, where each node has, at most, two children:

```
struct node {
   int data;
   struct node *left, *right;
};
```

Today's topic is the trie, which is pronounced "tree" (as in retrieval).

However, we will break from this tradition and pronounce it "try," so as to distinguish it from the tree data structure.

There are two super amazing characteristics of tries that I want to introduce you to straight away:

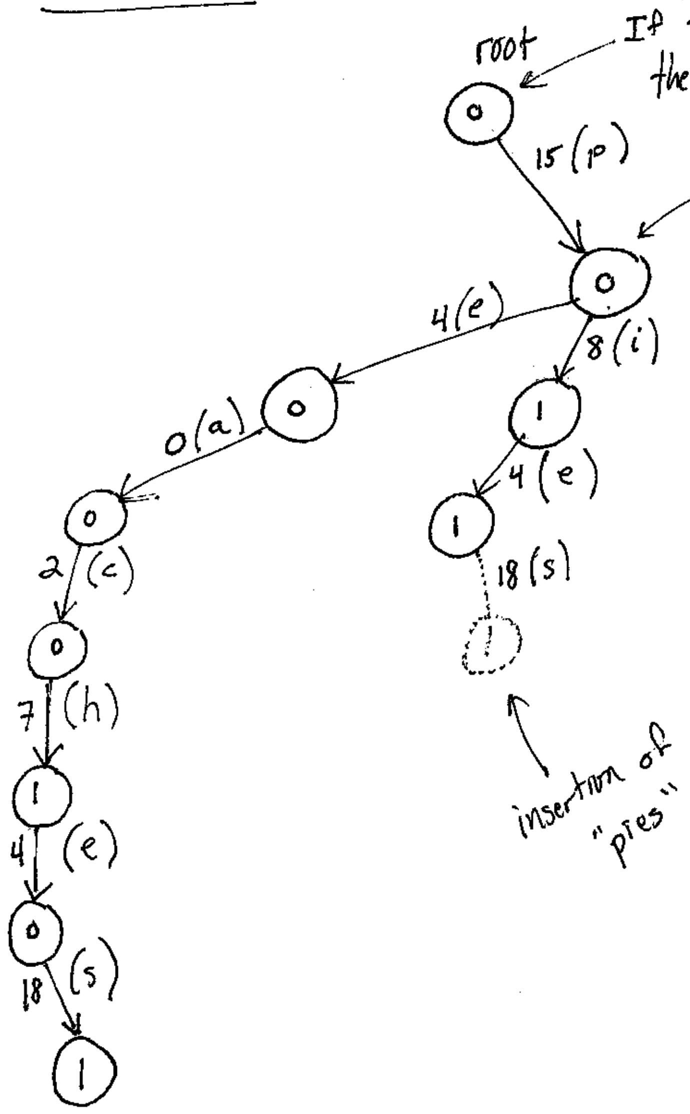
Firstly, a trie is a tree in which every node has 26 children (or, rather, 26 child pointers —— some, or all, of which may be NULL):

```
struct trieNode {
     int count;
     struct trieNode * children[26];  //an array of 26 trieNode pointers
};
```

Secondly, although we will use tries to represent strings, the strings we insert into a trie are not stored as data inside our nodes. Instead, the string that a node represents is based on the path you take to reach that node.

Note: The integer value in each node is the value of the struct's "count" field.

For example:



If the trie contained the empty string (""), the value here would be non-zero.

This is the node at root → children[15]. This node represents "p" because 'p' is the 16th letter of the alphabet. If we start counting from zero, 'p' is integer # 15. (for letter 'a')

Note: Each node has many children that I am not drawing, for the sake of clarity and simplicity. We assume those children are NULL.

	Worst	Best
The big-oh runtime for insertion into a trie is:	O(k)	O(k)
The big-oh runtime for look-up in a trie is:	O(k)	O(1)
(We will see the big-oh runtime for deletion is also:	O(k)	O(1))

Where k is the length of the string we're dealing with.

Some Applications: dictionary (spell check)
document word count / word frequency
find all words beginning with some prefix
word prediction for typing / texting / voice recognition (speech-to-text processing)

So, <u>if</u> the word "apple" has been inserted into your trie, how can you access its count field?

The letter 'a' corresponds to index 0 in the children arrays. 'p' corresponds to index 15. 'l' corresponds to index 11, and 'e' corresponds to index 4.

So, we could access that field like so:

```
root→children[0]→children[15]→children[15]→children[11]→children[4]→count
```

(Again, this assumes "apple" is in the trie, so we don't have to worry about segfaults.)

What if we don't know the index of 'p' off the top of our heads, though? How can we get to "apple"?

```
root→children['a'-'a']→children['p'-'a']→children['p'-'a']→children['l'-'a']→children['e'-'a']→count
```

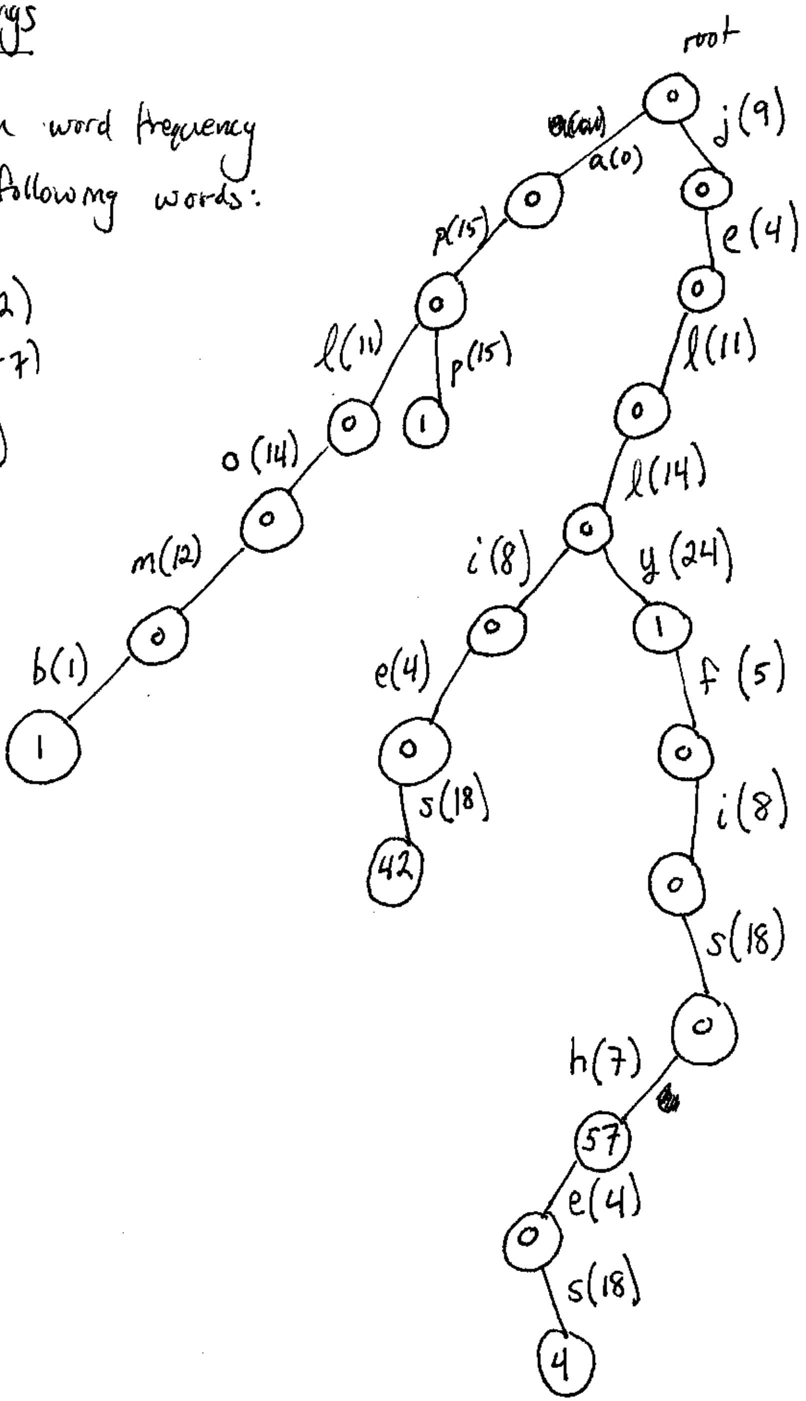
Or, more generally speaking: Given some char variable, c, we can traverse the edge corresponding to that character by accessing:

```
→children[c - 'a']
```

Tries: Deleting Strings

Suppose we have a word frequency trie with the following words:

jelly	(x1)
jellies	(x42)
jellyfish	(x57)
jellyfishes	(x4)
aplomb	(x1)
app	(x1)



1. To delete "aplomb", we can decrement the count at its terminal node to zero, then delete all the nodes up to (and including) the 'l', since they have no children.

2. To delete one occurrence of "jelly", we simply decrement the count value at its terminal node. (We cannot delete its terminal node even though its count is now zero. Why?)

3. To delete one occurrence of "jellyfishes", we decrement the count field at its terminal node, but no nodes get deleted. (Why not?)

Tries

Notes by: Sean Szumlanski

COP 3502, Fall 2015

Monday, November 2, 2015

Attachment: trie-notes.pdf

Today, we introduced the trie data structure. Some notes and diagrams are attached in trie-notes.pdf.

We started class with a discussion of dictionaries and spell checking using BSTs or binary search through a sorted array:

- We discussed how a BST could be used as a dictionary (by supporting insertion and look-up functions). Potential weakness: If we insert a dictionary of words into a BST in alphabetical order, it devolves into a linked list, ruining our runtimes for insertion and look-up.
- We could insert words into an array and then sort it with an O(n log n) sorting algorithm. After that, we can determine whether a word is in the array using binary search. The big limitation here is that if we want to add words while the program is running, we need to expand our array, which could be a costly operation (in terms of runtime). (And this isn't a far-fetched scenario; people add words to their web browser and cell phone spell checkers all the time.)
- Note that if we want to know how many times a word occurs in a corpus, we could modify our BST node struct to have a word count field (instead of inserting words into a BST multiple times, which would take up extra space and also require us to write a slower function to determine how many times a word occurs in the BST).
- Recall that a *corpus* is a body of text.
- All the approaches mentioned above suffer tremendously if we have a dictionary full of fake words that share a long prefix. For example, suppose I have a dictionary with 200 million words that all begin with "spaghettiHasNothingToDoWithAnything", such as, "spaghettiHasNothingToDoWithAnythingA", "spaghettiHasNothingToDoWithAnythingB", "spaghettiHasNothingToDoWithAnythingAndThisCouldGoOnForever", and so on.) Suddenly, looking up those strings is a bit more expensive, because we have to go through 35 characters *every single time* we perform a comparison. That wasn't the case when we were dealing with arrays of integers, and we often sweep that issue under the rug when talking about runtime, because we assume string length is bounded by some sort of reasonable constant, but it's a good thing to keep in mind, because it does have a real impact on runtime, especially if you're processing a ton of data.

On the topic of tries, we saw:

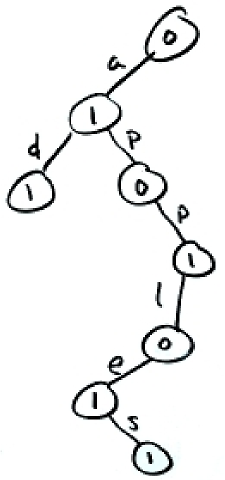
- Strings are not stored as data inside trie nodes. Instead, the node that a string represents is based on the path you take to reach that node.
- To get from one node to another via an edge that represents a specific letter (for example, the letter 'p'), we can access:
  ```
  n->children['p' - 'a']
  ```
- Insertion, look-up, and deletion are now O(k) operations (worst-case), where *k* is the length of the string being inserted, retrieved, or deleted. Notice that this is entirely independent of the number of strings we have already inserted into the trie.
- The best-case runtime for insertion is O(k), because we have to process all *k* letters in the string.
- The best-case runtime for look-up and deletion is O(1). If the first letter of the string leads us to a NULL pointer in the trie, we terminate immediately.
- There are some serious space trade-offs with tries. On the one hand, words with shared prefixes use the same nodes to represent those prefixes, so we're saving some space. (For example, the "app" in "app," "application," "applications," "apple," "apples," and "applesauce," is represented by the same three nodes in the trie.) However, each node in the trie has 26 children, so the number of pointers can be exponentially explosive.
- The word "trie" comes from "retrieval," and should technically be pronounced like "tree." However, I will pronounce it like "try" in an effort to be clear about which data structure I'm talking about.

We also discussed the process for deleting a word from a trie. (This assumes that our nodes hold a *count* -- the number of times a string has been inserted into the trie. It also assumes that each delete operation wants to remove only a *single* occurrence of that string from the trie -- not all of them.)

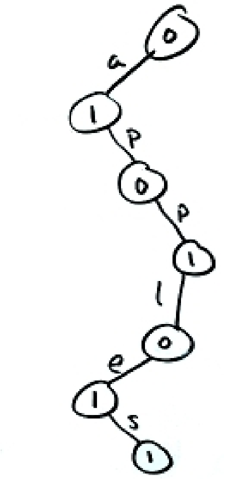
First, we traverse down to the terminal node for that word. At that node, we do the following:

```
if this node's count field is 0
   do nothing (it's not in the trie, so we can't delete it!)
otherwise, if this node's count field is > 1
   just decrement the count variable by one
otherwise, if this node has children
   just set the count field to 0
otherwise
   prune this node
   walk back up the chain toward the root node, deleting each node you encounter along the way, until either:
      a. you encounter a node with count > 0
       -- or --
      b. you encounter a node that has at least one non-null child
```

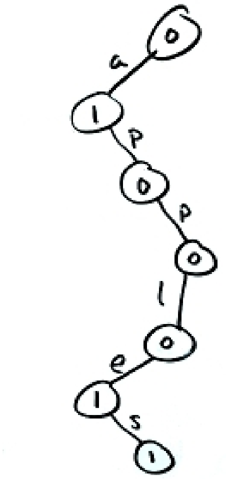
Let's trace through an example. Suppose I start with this trie:



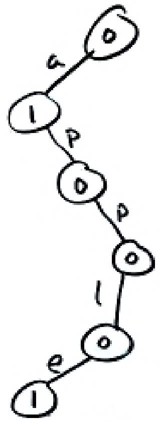
Deleting "ad" yields the following trie. We removed the 'd' node since it was a leaf with no children and its *count* field was decremented from 1 to 0:



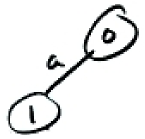
Deleting "app" yields the following trie. We set the second 'p' node's a *count* field to 0, but we could not delete that node because it has children:



Deleting "apples" yields the following trie. We removed the 's' node since it was a leaf with no children and its *count* field was decremented from 1 to 0:



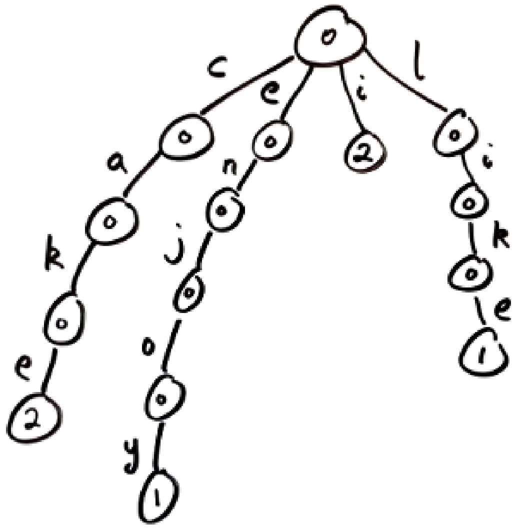
Finally, deleting "apple" yields the following trie. We removed the 'e' node since it was a leaf with no children and its *count* field was decremented from 1 to 0. We then followed the chain up toward the root, deleting nodes until we encountered one that either had non-null children or a *count* field that was greater than 0:



I also mentioned in class that we could embed more information in each trie node. For example, we could have each trie node point to a whole, separate trie that contains words that frequently co-occur with the word that got us to that node.

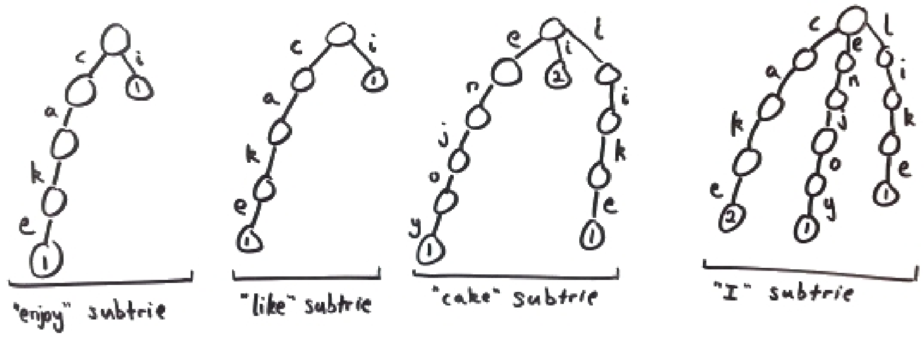
For example, what if I want to maintain a separate trie that tells me all the words I have typed into my cell phone recently in all sentences that contain the word "cake?" Or, what if I want to know how many times every word has occurred in a sentence with every other word I've typed, and the only sentences I've typed recently are "I like cake" and "I enjoy cake"? Then I might build the following trie, with the four subtries listed below.

(*Continued on next page...*)



<u>Derived from the following corpus:</u>

I like cake.
I enjoy cake.



* Assume all the empty nodes have *count* values of 0.

What's next?

On Wednesday, we'll delve into some code for tries, and then we'll quickly hop along to another data structure: minheaps.

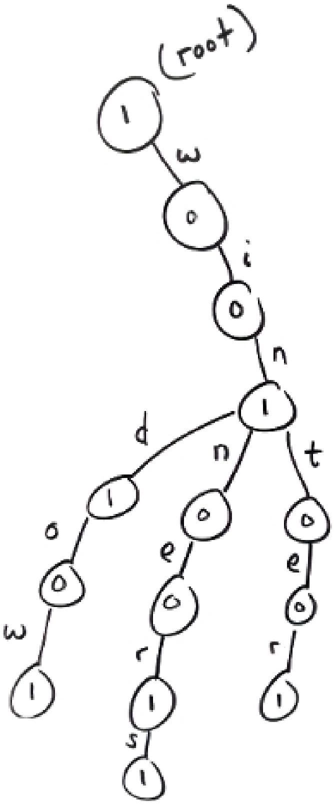
Practice Problems

1. Write a function to insert a word into a trie. (But don't share your solution with anyone, because you have to write a similar function for your next programming assignment.) Make sure your function is case insensitive, so insert("SoMeSTriNG") should insert "somestring". The node struct is:

```
typedef struct trieNode
{
   // for this exercise, you can treat this as a simple 0 or 1 flag,
   // or you can get fancy with it and print the count associated
   // with each word in the trie
   int count;

   // 26 child nodes, one for each letter of the alphabet
   struct trieNode *children[26];
} trieNode;
```

2. Refer to the following trie for the exercises below:



a. What strings are represented in the trie?

b. Show what the trie looks like after inserting the strings "winnowing" and "virtue."

c. Show what the trie looks like after deleting each of the following strings: "window", "winners", and "win"