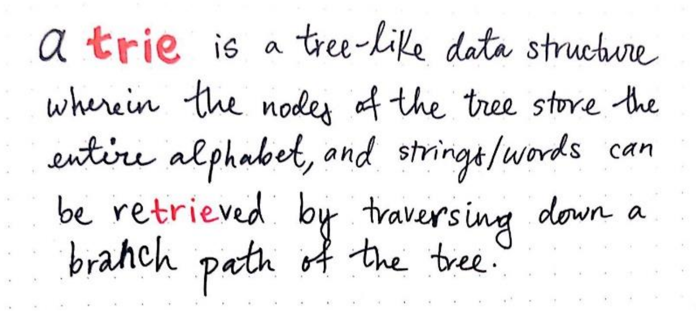
National University of Computer & Emerging Sciences (NUCES) Islamabad,
Department of Computer Science

**CS201: Data Structures (Fall 2018)**

## TRIE

There are a handful of different ways to represent something as seemingly simple as a set of words. For example, a hash or dictionary is one that we're probably familiar with, as is as hash table. But there's another structure that was created to solve the very problem of representing a set of words: a ***trie***. The term "trie" comes from the word *re**trie**val*, and is usually pronounced "try", to distinguish it from other "tree" structures.

However, a trie is basically a tree data structure, but it just has a few rules to follow in terms of how it is created and used.



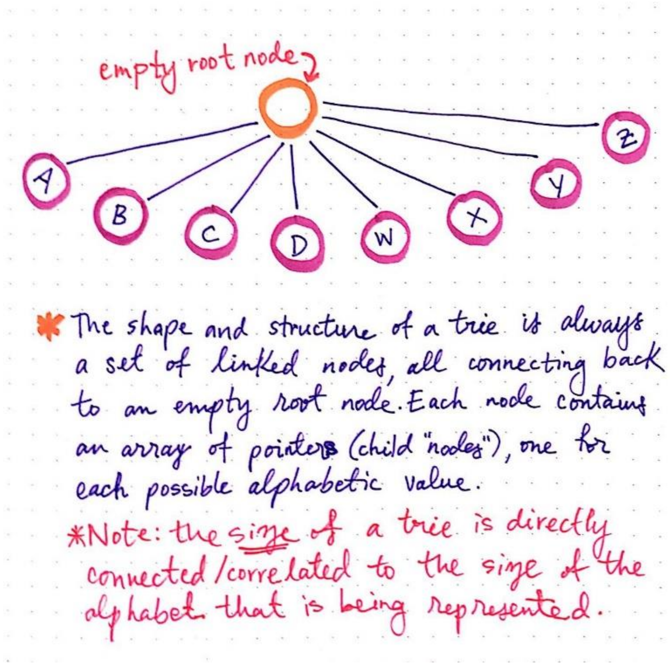
a trie is a tree-like data structure wherein the nodes of the tree store the entire alphabet, and strings/words can be retrieved by traversing down a branch path of the tree.

A trie is a tree-like data structure whose nodes store the letters of an alphabet. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree.

Tries in the context of computer science are a relatively new thing. The first time that they were considered in computing was back in 1959, when a Frenchman named René de la Briandais suggested using them. According to Donald Knuth's research in The Art of Computer Programming:

*Trie memory for computer searching was first recommended by René de la Briandais. He pointed out that we can save memory space at the expense of running time if we use a linked list for each node vector, since most of the entries in the vectors tend to be empty.*

The original idea behind using tries as a computing structure was that they could be a nice compromise between running time and memory. But we'll come back to that in a bit. First, let's take a step back and try and understand what exactly this structure looks like to start.


empty root node
A
B
C
D
W
X
Y
Z
✱ The shape and structure of a trie is always a set of linked nodes, all connecting back to an empty root node. Each node contains an array of pointers (child "nodes"), one for each possible alphabetic value.
✱Note: the size of a trie is directly connected/correlated to the size of the alphabet that is being represented.

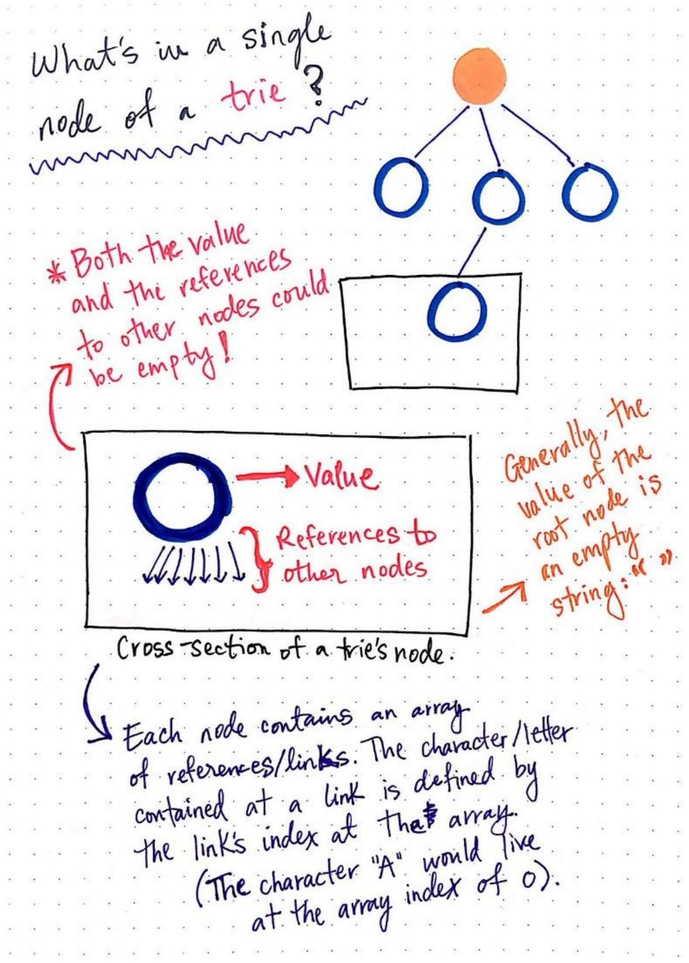
We know that tries are often used to represent words in an alphabet. In the illustration shown here, we can start to get a sense of how exactly that representation works.

Each trie has an empty root node, with links (or references) to other nodes—one for each possible alphabetic value.

The shape and the structure of a trie is always a set of linked nodes, connecting back to an empty root node. An important thing to note is that the number of child nodes in a trie depends completely upon the total number of values possible. For example, if we are representing the English alphabet, then the total number of child nodes is directly connected to the total number of letters possible. In the English alphabet, there are 26 letters, so the total number of child nodes will be 26.

Imagine, however, that we were creating a trie to hold words from the Khmer (Cambodian) alphabet, which is the longest known alphabet with 74 characters. In that case, the root node would contain 74 links to 74 other child nodes.

*The size of a trie is directly correlated to the size of all the possible values that the trie could represent.*


What's in a single node of a trie?
* Both the value and the references to other nodes could be empty!
Value
References to other nodes
Generally, the value of the root node is an empty string: " "
Cross-section of a trie's node.
Each node contains an array of references/links. The character/letter contained at a link is defined by the link's index at that array. (The character "A" would live at the array index of 0).

Okay, so a trie could be pretty small or big, depending on what it contains. But, so far, all we've talked about is the root node, which is empty. So where do the letters of different words live if the root node doesn't house them all?

The answer to that lies in the root node's references to its children. Let's take a closer look at what a single node in a trie looks like, and hopefully this will start to become clearer.

In the example shown here, we have a trie that has an empty root node, which has references to children nodes. If we look at the cross-section of one of these child nodes, we'll notice that a single node in a trie contains just two things:

1. A value, which might be null
2. An array of references to child nodes, all of which also might be null

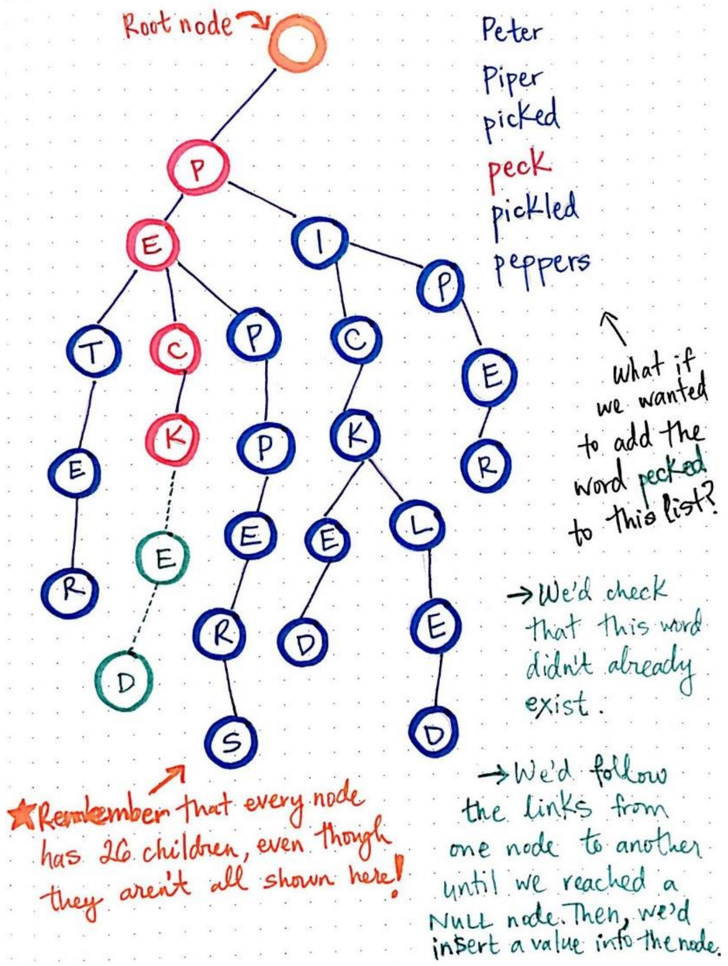
Each node in a trie, including the root node itself, has only these two aspects to it. When a trie representing the English language is created, it consists of a single root node, whose value is usually set to an empty string: "".

That root node will also have an array that contains 26 references, all of which will point to null at first. As the trie grows, those pointers start to get filled up with *references* to other nodes nodes, which we'll see an example of pretty soon.

The way that those pointers or references are represented is particularly interesting. We know that each node contains an array of references/links to other nodes. What's cool about this is that we can use the array's indexes to find specific references to nodes. For example, our root node will hold an array of indexes 0 through 25, since there are 26 possible slots for the 26 letters of the alphabet. Since the alphabet is in order, we know that the reference to the node that will contain the letter A will live at index 0.

So, once we have a root node, where do we go from there? It's time to try growing our trie!

In the trie shown below, we're representing the nursery rhyme that starts off with something like "*Peter Piper picked a peck of pickled peppers*".


Root node
Peter
Piper
picked
peck
pickled
peppers
what if we wanted to add the word pecked to this list?
→ We'd check that this word didn't already exist.
→ We'd follow the links from one node to another until we reached a NULL node. Then, we'd insert a value into the node.
★ Remember that every node has 26 children, even though they aren't all shown here!
P
E
I
P
T
C
P
C
E
K
E
P
K
R
R
E
E
E
L
D
R
D
E
S
D

Looking at our trie, we can see that we have an empty root node, as is typical for a trie structure. We also have six different words that we're representing in this trie: Peter, piper, picked, peck, pickled, and peppers.

To make this trie easier to look at, I've only drawn the references that actually have nodes in them; it's important to remember that, even though they're not illustrated here, every single node has 26 references to possible child nodes.

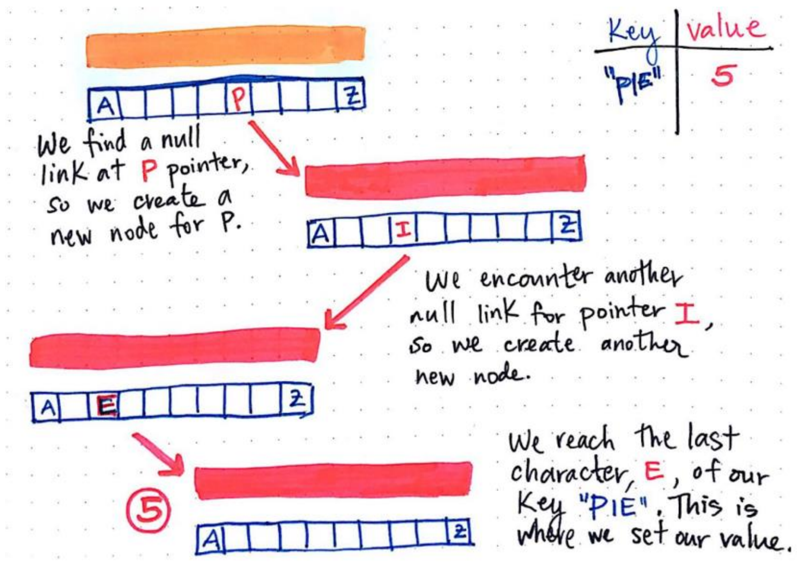
Notice how there are six different "branches" to this trie, one for each word that's being represented. We can also see that some words are sharing parent nodes. For example, all of the branches for the words Peter, peck, and peppers share the nodes for p and for e. Similarly, the path to the word picked and pickled share the nodes p, i, c, and k.

So, what if we wanted to add the word pecked to this list of words represented by this trie? We'd need to do two things in order to make this happen:

1. First, we'd need to check that the word pecked doesn't already exist in this trie.
2. Next, if we've traversed down the branch where this word *ought* to live and the words doesn't exist yet, we'd insert a value into the node's reference where the word should go. In this case, we'd insert e and d at the correct references.

But how do we actually go about checking if the word exists? And *how* do we insert the letters into their correct places? This is easier to understand with a small trie as an example, so let's look at a trie that is empty, and try inserting something into it.

We know that we'll have an empty root node, which will have a value of "", and an array with 26 references in it, all of which will be empty (pointing to null) to start. Let's say that we want to insert the word "pie", and give it a value of 5. Another way to think about it is that we have a hash that looks like this: {"pie": 5}.

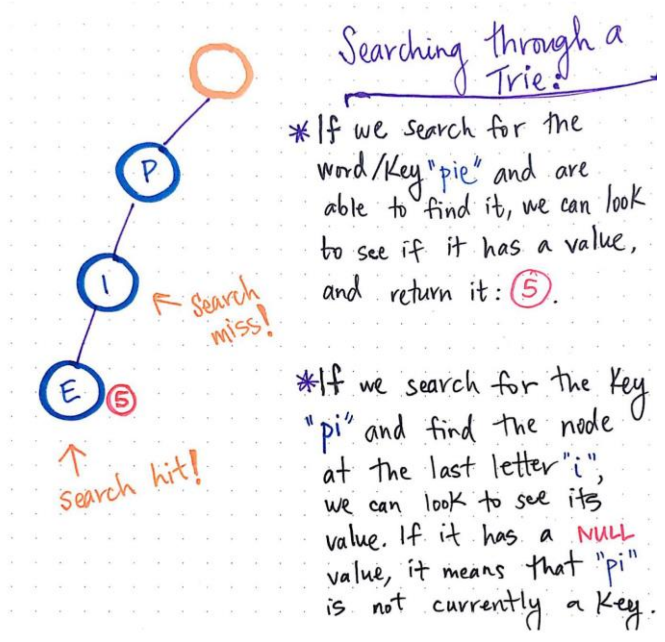

Key
value
"PIE"
5
A
P
Z
We find a null link at P pointer, so we create a new node for P.
A
I
Z
We encounter another null link for pointer I, so we create another new node.
A
E
Z
We reach the last character, E, of our Key "PIE". This is where we set our value.
5
A
Z

We'll work our way through the key, using each letter to build up our trie and add nodes as necessary.

We'll first look for the pointer for p, since the first letter in our key "pie" is p. Since this trie doesn't have anything in just yet, the reference at p in our root node will be null. So, we'll create a new node for p, and the root node now has an array with 25 empty slots, and 1 slot (at index 15) that contains a reference to a node.

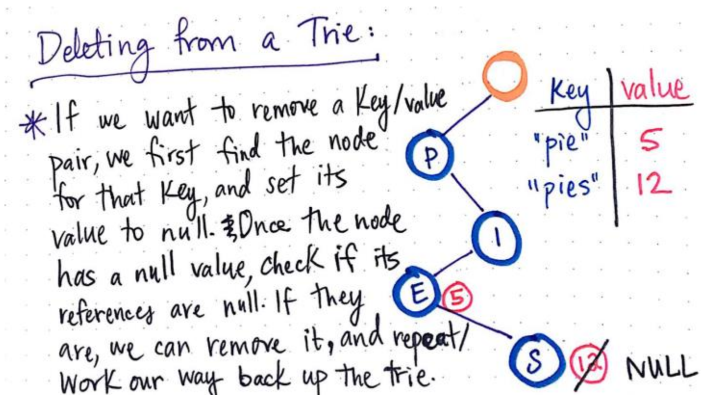
Now we have a node at index 15, holding the value for p. But, our string is "pie", so we're not done yet. We'll do the same thing for this node: check if there is a null pointer at the next letter of the key: i. Since we encounter another null link for the reference at i, we'll create another new node. Finally, we're at the last character of our key: the e in "pie". We create a new node for the array reference to e, and inside of this *third* node that we've created, we'll set our value: 5.

In the future, if we want to retrieve the value for the key "pie", we'll traverse down from one array to another, using the indices to go from the nodes p, to i, to e; when we get to the node at the index for e, we'll stop traversing, and retrieve the value from that node, which will be 5.


Searching through a Trie:
P
I
E
5
Search miss!
search hit!
*If we search for the word/Key "pie" and are able to find it, we can look to see if it has a value, and return it: 5.
*If we search for the Key "pi" and find the node at the last letter "i", we can look to see its value. If it has a NULL value, it means that "pi" is not currently a Key.

Let's actually take a look at what searching through our newly-built trie would look like! In the illustration shown here, if we search for the key "pie", we traverse down each node's array, and look to see if there is a value for the branch path: p-i-e. If it does have a value, we can simply return it. This is sometimes referred to as a search hit, since we were able to find a value for the key. But what if we search for something that doesn't exist in our trie? What if we search for the word "pi", which we haven't added as a key with a value? Well, we'll go from the root node to the node at index p, and then we'll go from the node at p to the node at index i . When we get to this point, we'll see if the node at the branch path p-i has a value. In this case, it doesn't have a value; it's pointing at null. So, we can be sure that the key "pi" doesn't exist in our trie as a string with a value. This is often referred to as a search miss, since we could not find a value for the key.

Finally, there's one other action that we might want to do to our trie: delete things! How can we remove a key and its value from our trie structure? To illustrate this, I've added another word to our trie. We now have both the keys "pie" and "pies", each with their own values. Let's say we want to remove the key "pies" from our trie.



Deleting from a Trie:

* If we want to remove a Key/value pair, we first find the node for that Key, and set its value to null. & Once the node has a null value, check if its references are null. If they are, we can remove it, and repeat/ Work our way back up the trie.

| Key | value |
|---|---|
| "pie" | 5 |
| "pies" | 12 |

In order to do this, we’d need to take two steps:

1. First, we need to find the node that contains the value for that key, and set its value to null. This means traversing down and finding the last letter of the word "pies”, and then resetting the value of the last node from 12 to null.
2. Second, we need to check the node’s references and see if all of its pointers to other nodes are also null. If all of them are empty, that means that there are no other words/branches below this one, and they can all be removed. However, if there are pointers for other nodes that do have values, we don’t want to delete the node that we’ve just set to null.

This last check is particularly important in order to not remove longer strings when we remove substrings of a word. But other than that single check, there’s nothing more to it!