Get unlimited access to the best of Medium for less than $1/week. Become a member

Trying to Understand Tries

Vaidehi Joshi · Follow
Published in basecs · 14 min read · Jul 31, 2017

8.4K 33

In every installment of this series, we've tried to understand and dig deep into the tradeoffs of the things that we're learning about.

When we were learning about data structures, we looked at the pros and cons of each structure, in an effort to make it easier and more obvious for us to see what types of problems that structure was created to solve. Similarly, when we were learning about sorting algorithms, we focused a lot on the tradeoffs between space and time efficiency to help us understand when one algorithm might be the better choice over another.

As it turns out, this is going to become more and more frequent as we start looking at even more complex structures and algorithms, some of which were invented as solutions to *super* specific problems. Today's data structure is, in fact, based on another structure that we're already familiar with; however, it was created to solve a particular problem. More specifically, it

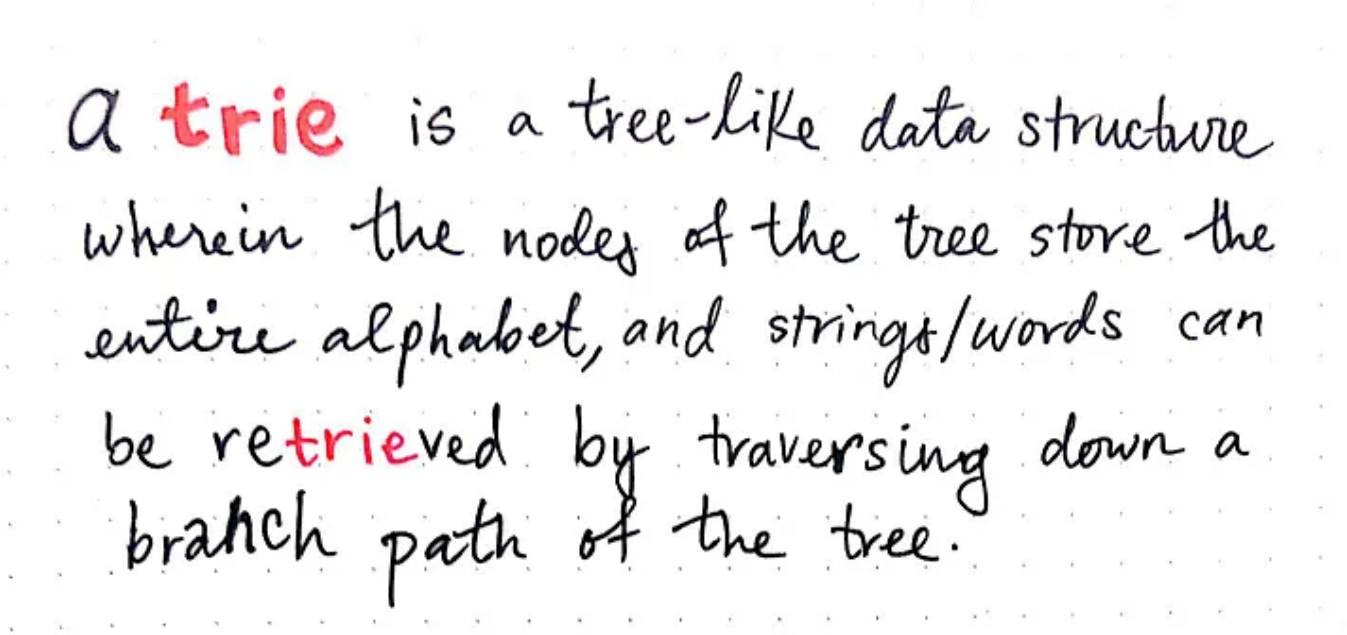
was created as a compromise between running time and space — two things that we're pretty familiar with in the context of Big O notation.

So, what is this mysterious structure that I keep talking about so vaguely but not actually naming? Time to dig in and find out!

Trying on tries

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.



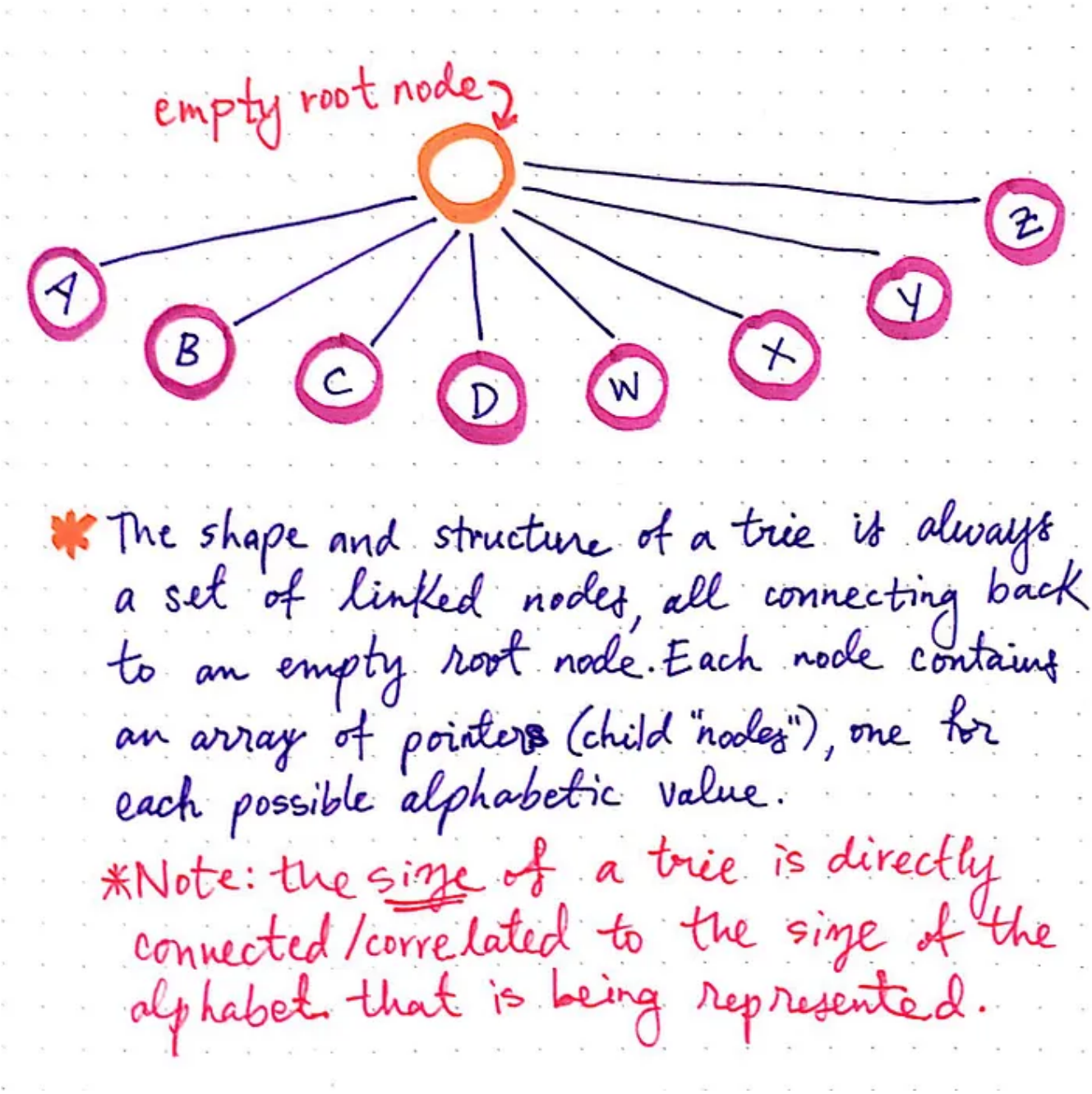
Trie: a definition

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.



The size of a trie correlates to the size of the alphabet it represents.

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.

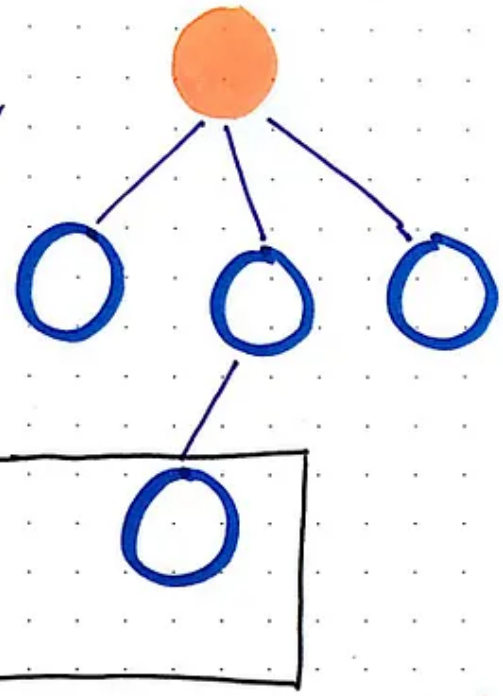
Imagine, however, that we were creating a trie to hold words from the Khmer (Cambodian) alphabet, which is the longest known alphabet with 74 characters. In that case, the root node would contain 74 links to 74 other child nodes.

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

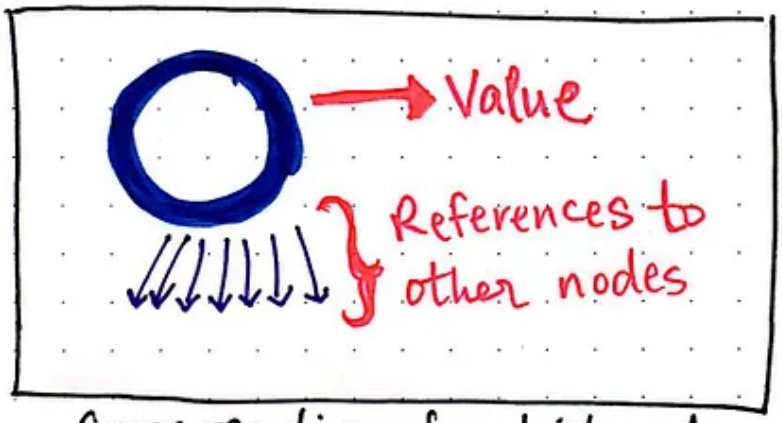
Okay, so a trie could be pretty small or big, depending on what it contains. But, so far, all we've talked about is the root node, which is empty. So where do the letters of different words live if the root node doesn't house them all?

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 more clear.

What's in a single node of a trie?



* Both the value and the references to other nodes could be empty!



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).

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.

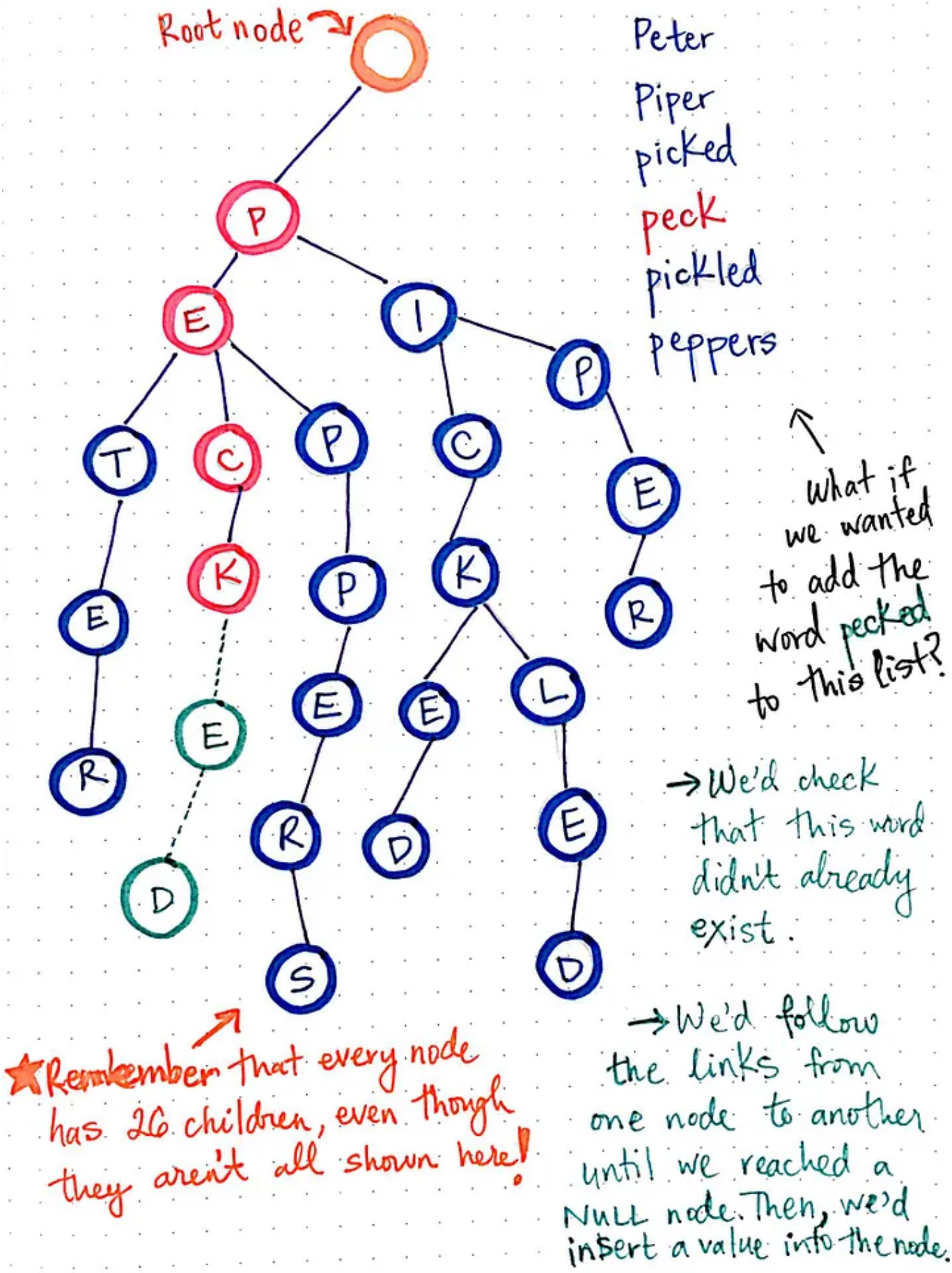
The way that those pointers or references are represented is particularly interesting. We know that each node contains an array of references/links to other nodes. What's cool about this is that we can use the array's indexes to find specific references to nodes. For example, our root node will hold 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!

Giving trie traversal a try

A trie with nothing more than a root node is simply no fun at all! So, let's complicate things a bit further by playing with a trie that has some words in it, shall we?

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*". I won't try to make you remember the rest of it, mostly because it is confusing and makes my head hurt.



What if we wanted to add a word to our trie list?

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.

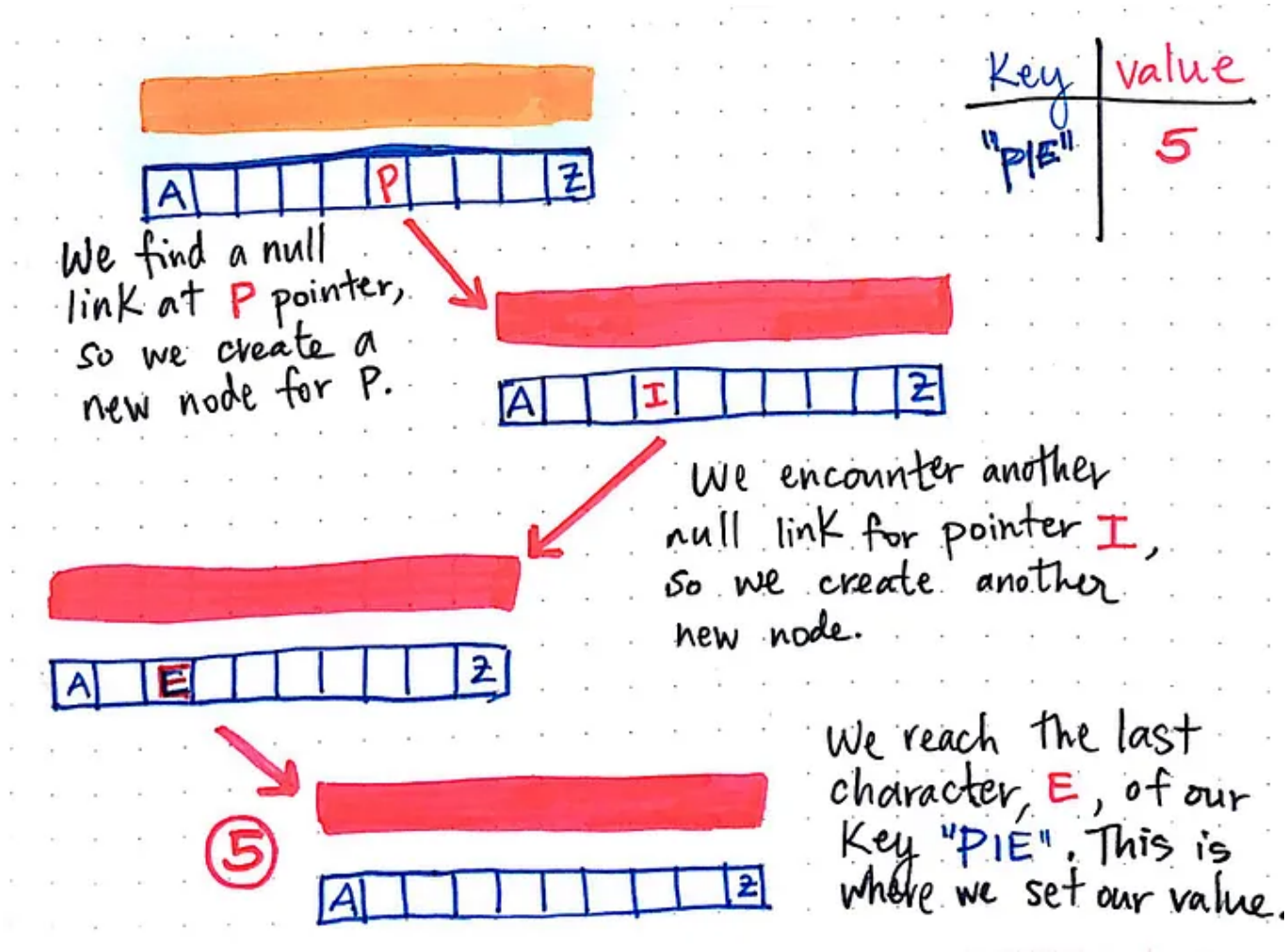
Notice how there are six different "branches" to this trie, one for each word that's being represented. We can also see that some words are sharing parent nodes. For example, all of the branches for the words `Peter`, `peck`, and `peppers` share the nodes for `p` and for `e`. Similarly, the path to the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



Understanding array pointers in a trie structure

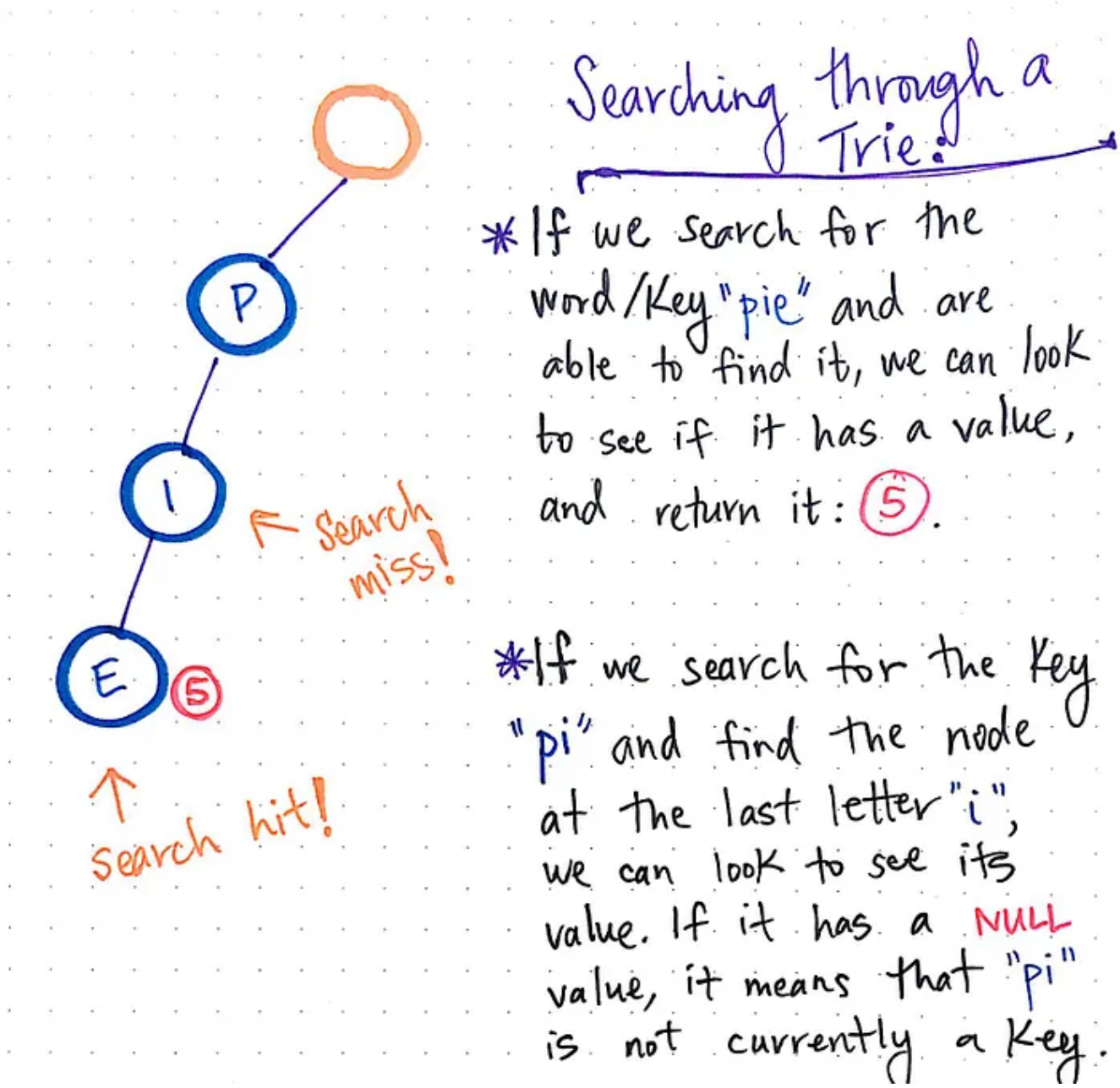
We'll work our way through the key, using each letter to build up our trie and add nodes as necessary.

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`.



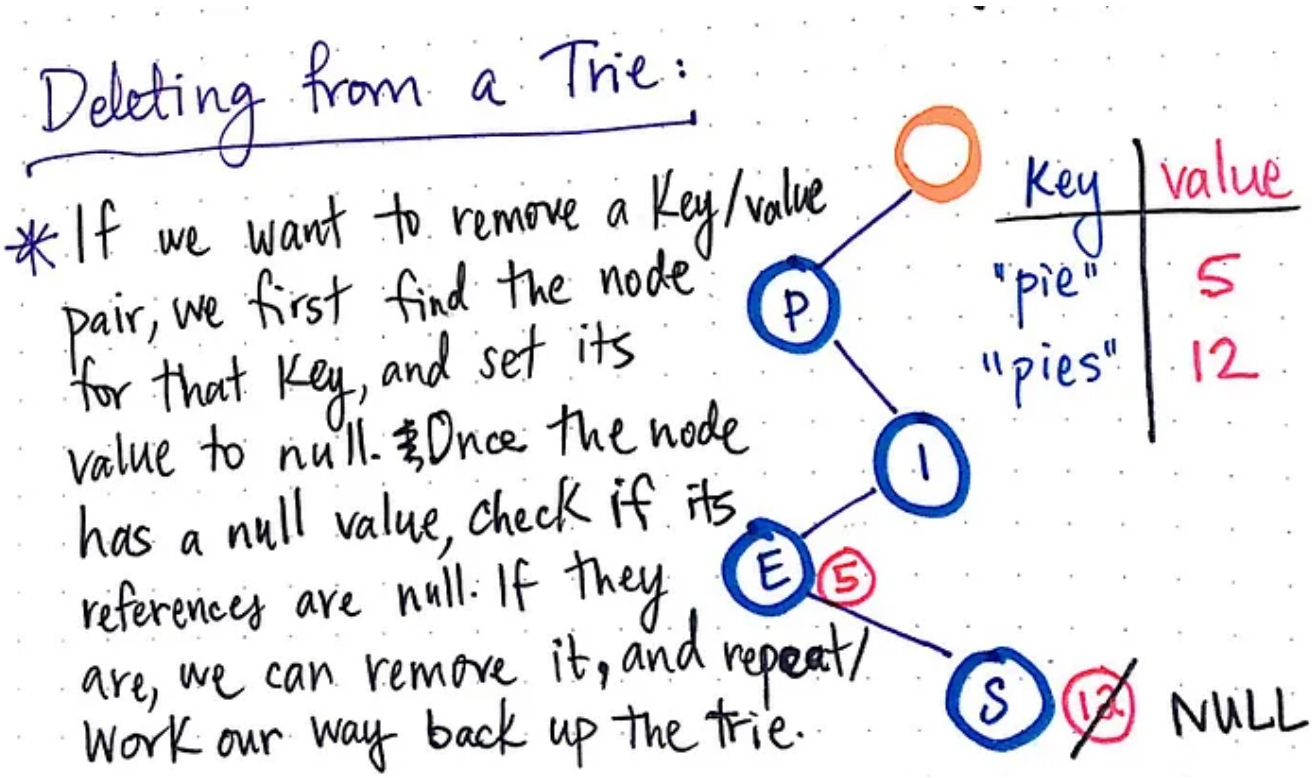
Searching through a trie

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

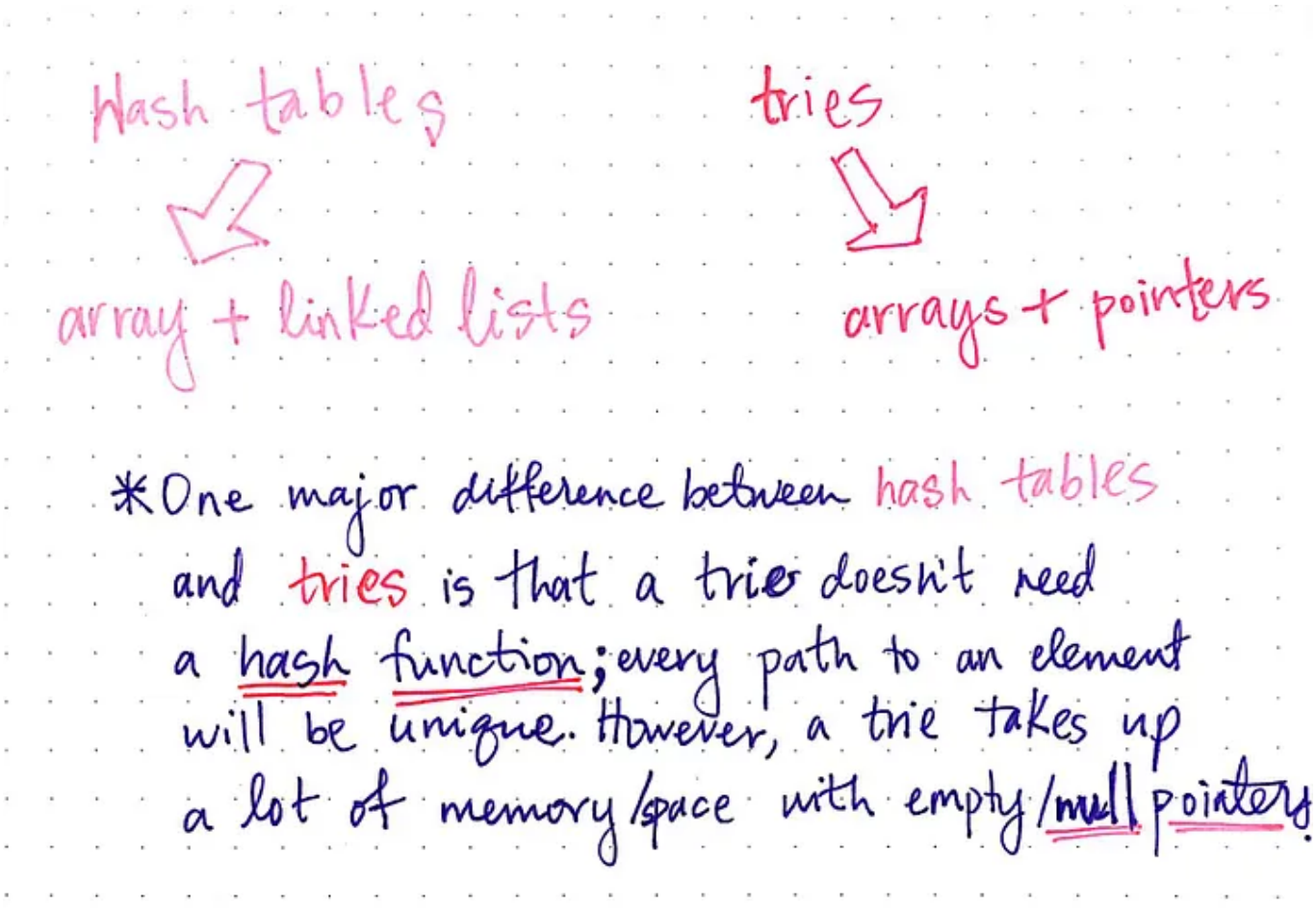
In order to do this, we'd need to take two steps:

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!

Trying our hand at tries

When I was first learning about tries, they reminded me a lot of hash tables, which we learned about earlier in this series. In fact, the more that I read about tries and how to build and search through them, the more I wondered what the tradeoffs between the two structures actually were.

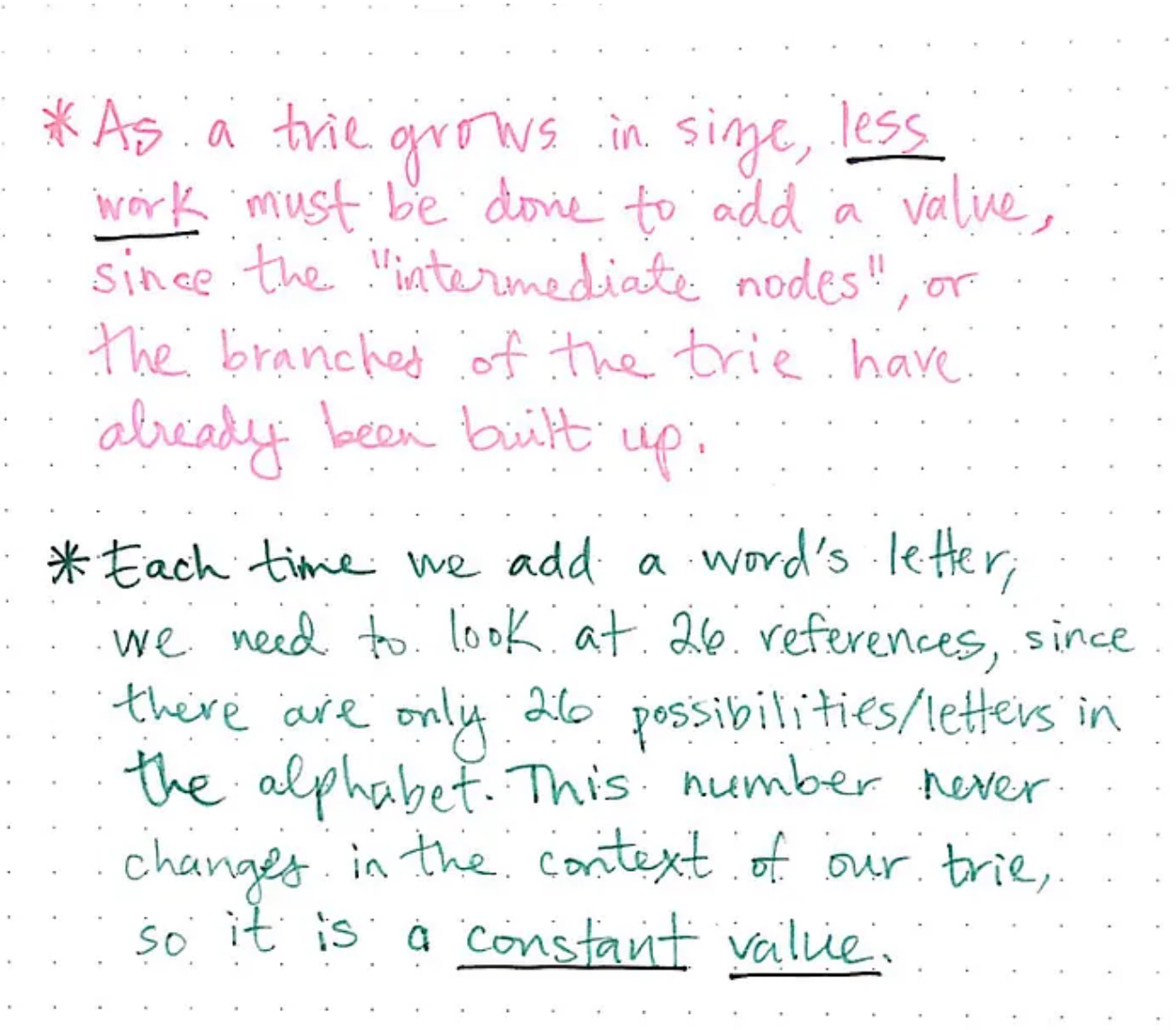


Hash tables vs. tries

As it turns out, both tries and hash tables are reminscient of one another because they both use arrays under the hood. However, hash tables use arrays combined with linked lists, whereas tries use arrays combined with pointers/references.

There are quite a few minor differences between both of these two structures, but the most obvious difference between hash tables and tries is that a trie has no need for a hash function, because every key can be represented in order (alphabetically), and is uniquely retrievable since every branch path to a string's value will be unique to *that* key. The side effect of this is that there are no collisions to deal with, and thus a relying on the index of an array is enough, and a hashing function is unnecessary.

However, unlike hash tables, the downside of a trie is that is takes up a lot of memory and space with empty (`null`) pointers. We can imagine how a large trie would start grow in size, and with each node that was added, an entire array containing 26 `null` pointers would have to be initialized as well. For longer words, those empty references would probably never get filled up; for example, imagine we had a key "*Honorificabilitudinitatibus*", with some value. That's super long word, and we're probably not going to be adding any other sub-branches to that word in the trie; that's a bunch of empty pointers for each letter of that word that are taking up space, but not really ever being used!



How tries changes as they grow

Hopefully though, we're not going to use the word "*Honorificabilitudinitatibus*" as a string.

There are some *great* benefits to using tries, however. For starters, the bulk of the work in creating a trie happens early on. This makes sense if we think about it, because when we're first adding nodes, we have to do some heavy lifting of allocating memory for an array each time. But, as the trie grows in size, we have to do less work each time to add a value, since it's likely that we've already initialized nodes and their values and references. Adding "intermediate nodes" becomes a lot easier since the branches of the trie have already been built up.

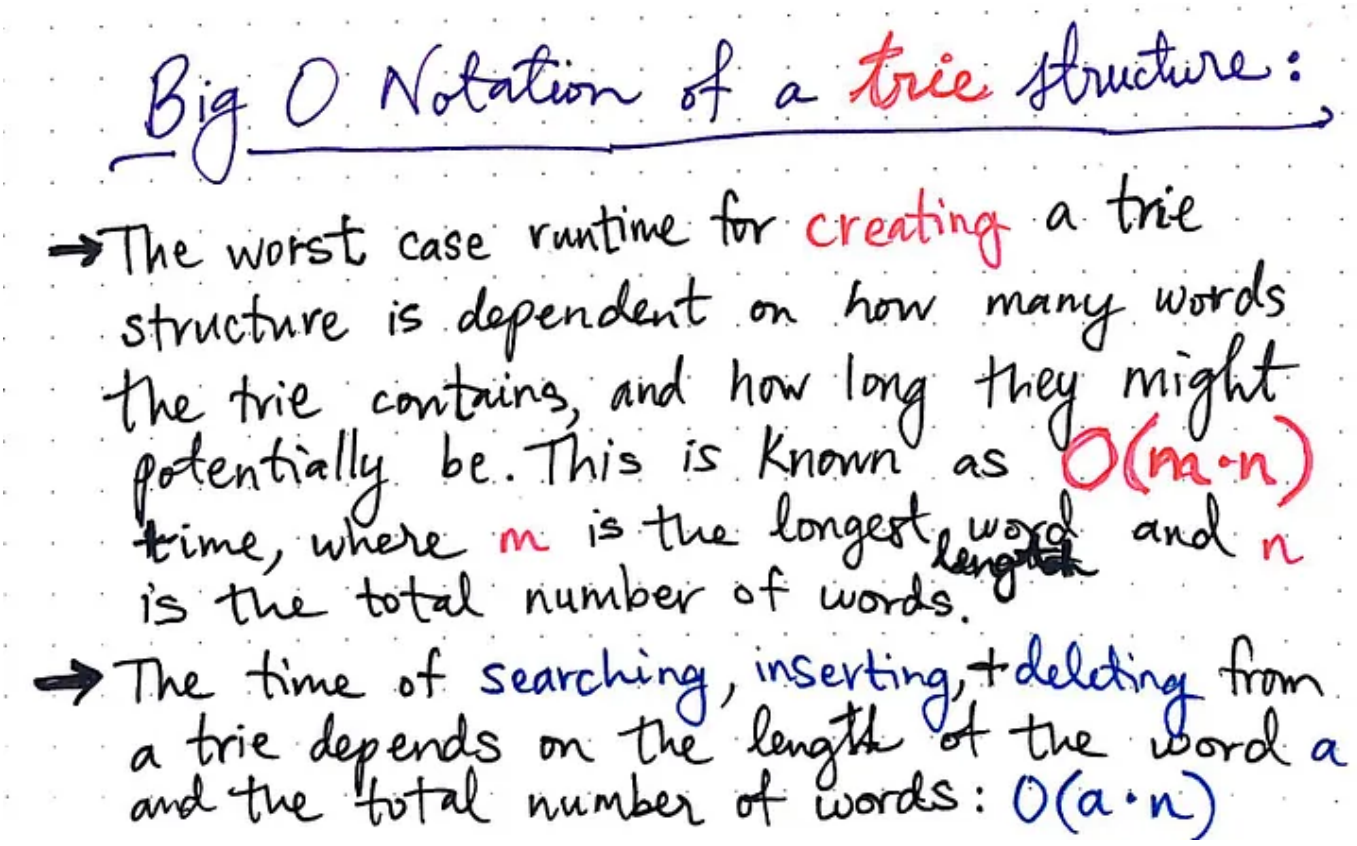
Another fact in the "pro column" for tries is that each time we add a word's letter, we know that we'll only ever have to look at 26 possible indexes in a node's array, since there are only 26 possible letters in the English alphabet. Even though 26 seems like a lot, for our computers, it's really not *that* much space. However, the fact that we are *sure* that each array will only ever contain 26 references is a huge benefit, because this number will never change in the context of our trie! It is a ***constant value***.

On that note, let's look quickly at the Big O time complexity of a trie data

Open in app ↗

Search Write 1

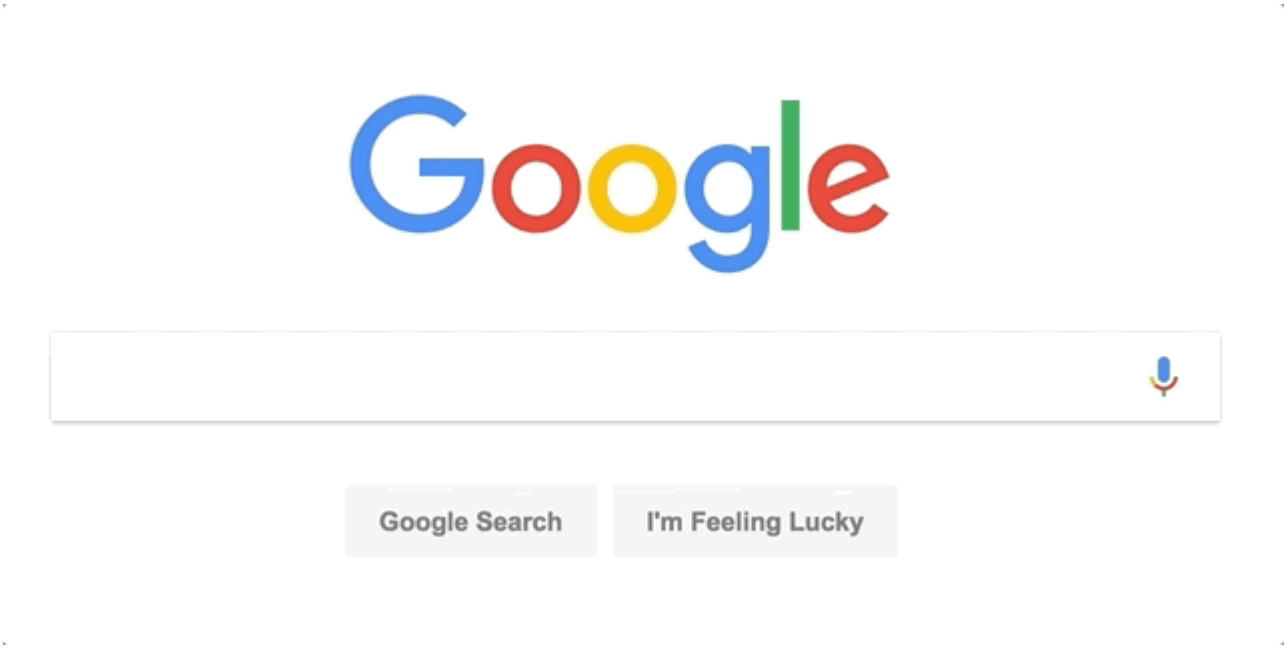
m, the length of the longest key in the trie, and *n*, the total number of keys in the trie. Thus, the worst case runtime of creating a trie is ***O(mn)***.



Big O Notation of a trie structure

The time complexity of searching, inserting, and deleting from a trie depends on the length of the word *a* that's being searched for, inserted, or deleted, and the number of total words, *n*, making the runtime of these operations ***O(an)***. Of course, for the longest word in the trie, inserting, searching, and deleting will take more time and memory than for the shortest word in the trie.

So, now that we know all the inner working of tries, there's one question that's still left to answer: where are tries used? Well, the truth is that they're rarely used exclusively; usually, they're used in combination with another structure, or in the context of an algorithm. But perhaps the coolest example of how tries can be leveraged for their form and function is for autocomplete features, like the one used in search engines like Google.



Autocomplete as a subset of a trie structure

Now that we know how tries function, we can imagine how typing two letters into a search box would retrieve a subset of a much larger trie structure. Another powerful aspect of this is that tries make it easy to search for a subset of elements, since, similar to binary search trees, each time we traverse down a branch of a tree, we are cutting out the number of *other* nodes we need to look at! It's worth mentioning that search engines probably have more complexity to their tries, since they will return certain terms based on how popular they are, and likely have some additional logic to determine the weight associated with certain terms in their trie structures. But, under the hood, they probably are using tries to make this magic happen!

Tries are also used for matching algorithms and implementing things like spellcheckers, and can also be used for imlementing versions of radix sort, too.

I suppose that if we trie hard enough, we'll see that tries are all around us! (Sorry, I just couldn't resist the pun)

Resources

Tries often show up in white boarding or technical interview questions, often in some variation of a question like “search for a string or substring from this sentence”. Given their unique ability to retrieve elements in constant time, they are often a great tool to use, and luckily, many people have written about them. If you want some helpful resources, here are a few good places to start.

1. Digit-based sorting and data structures, Professor Avrim Blum
2. Lecture Notes on Tries, Professor Frank Pfenning
3. Algorithms: Tries, Robert Sedgewick and Kevin Wayne
4. Tries, Brilliant Learning
5. Tries, Daniel Ellard
6. Tries, Harvard CS50

Data Structures | Programming | Computer Science | Tech | Software Development