Compressing Radix Trees Without (Too Many) Tears

Vaidehi Joshi · Follow

Published in basecs · 13 min read · Aug 7, 2017

As it turns out, there comes a turning point in researching computer science that you think to yourself, *"Oh wow, no one really knows how to explain this."* (And then of course, if you're me, you think to yourself: *"Oh no — how am I going to explain this?!"*.)

I have some good news and some bad news.

Bad news first: I finally hit that turning point. This week's topic is one that I felt *should* have been easy enough to research and learn about; however, it was so much tricker than I thought! In fact, if there is one topic that has brought me to the brink of tears in this series, it is, without a doubt, the one that I'm going to share with you today. Here's the good news though: I powered through, and after an uphill battle (which mostly consisted of me furiously Googling for answers), I finally started to understand not just how this concept works, but also just how cool it is!

Today's topic is all about *compressing* things. For example, I'm going to attempt to compress a deeply complicated topic, which people have spent their entire careers developing and understanding, down into one single post. So, what else are we compressing? Time to find out.

Hardcore (trie) compression

Last week, we learned about a powerful structure called a *trie*, which powers features that we interact with everyday (such as autocomplete), and is used for implementing matching algorithms, and sometimes even radix sort. We know that a trie is a tree-like data structure whose nodes store the letters of an alphabet, and that words and strings can be retrieved from the structure by traversing down a branch path of the tree.

We also learned that tries are great examples of tradeoffs. They make it easy to retrieve words, but they also take up a lot of memory and space with empty (`null`) pointers. Last week, we pretty much accepted this as a fact: tries are always going to take up a lot memory and space as they grow in size. But what if there was a way for us to cut down on some of the space and memory that a standard trie uses?

Well...surprise! It turns out there *is* a way. And we're finally ready to learn about it.

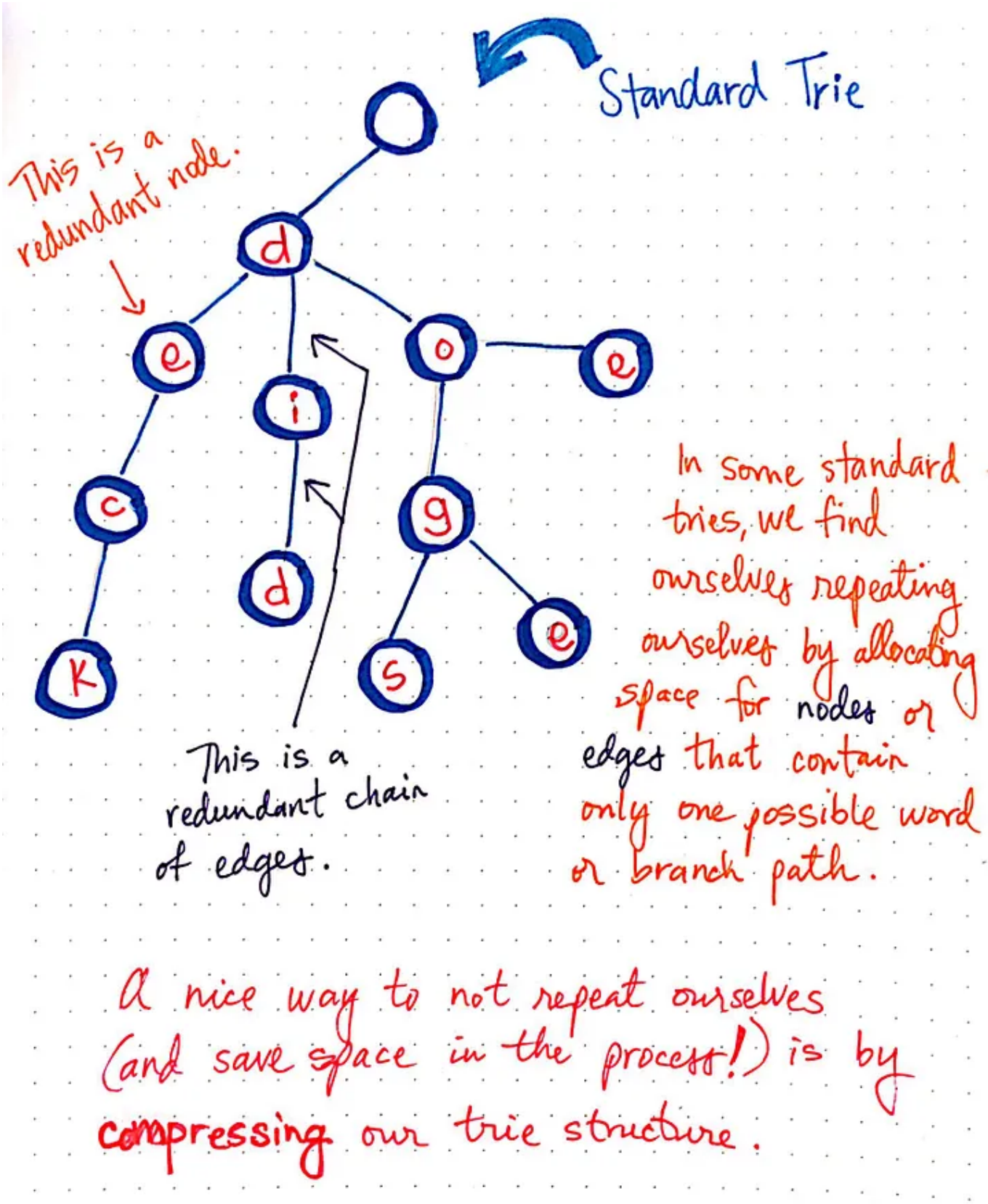
In order for us to undertand *how* to cut down on the amount of space that our trie uses, we need to be able to easily see what exactly is responsible for taking up sapce.

Imagine that we have a standard trie that represents a group of keys: `["deck", "did", "doe", "dog","doge", "dogs"]` . Remember that it's not really

an array of strings that we're dealing with here — each of these strings have values, so it's more like a hash/dictionary, where each key has a value:

```
{
  "deck": someValue,
  "did": someValue,
  "doe": someValue,
  "dog": someValue,
  "doge": someValue,
  "dogs": someValue
}
```

In the illustration shown below, we've transformed those keys into a standard trie.



In some standard tries, we repeat ourselves.

The first thing that we'll notice when we look at this trie is that there are two keys for which we have ***redundant nodes*** as well as a ***redundant chain of edges***.

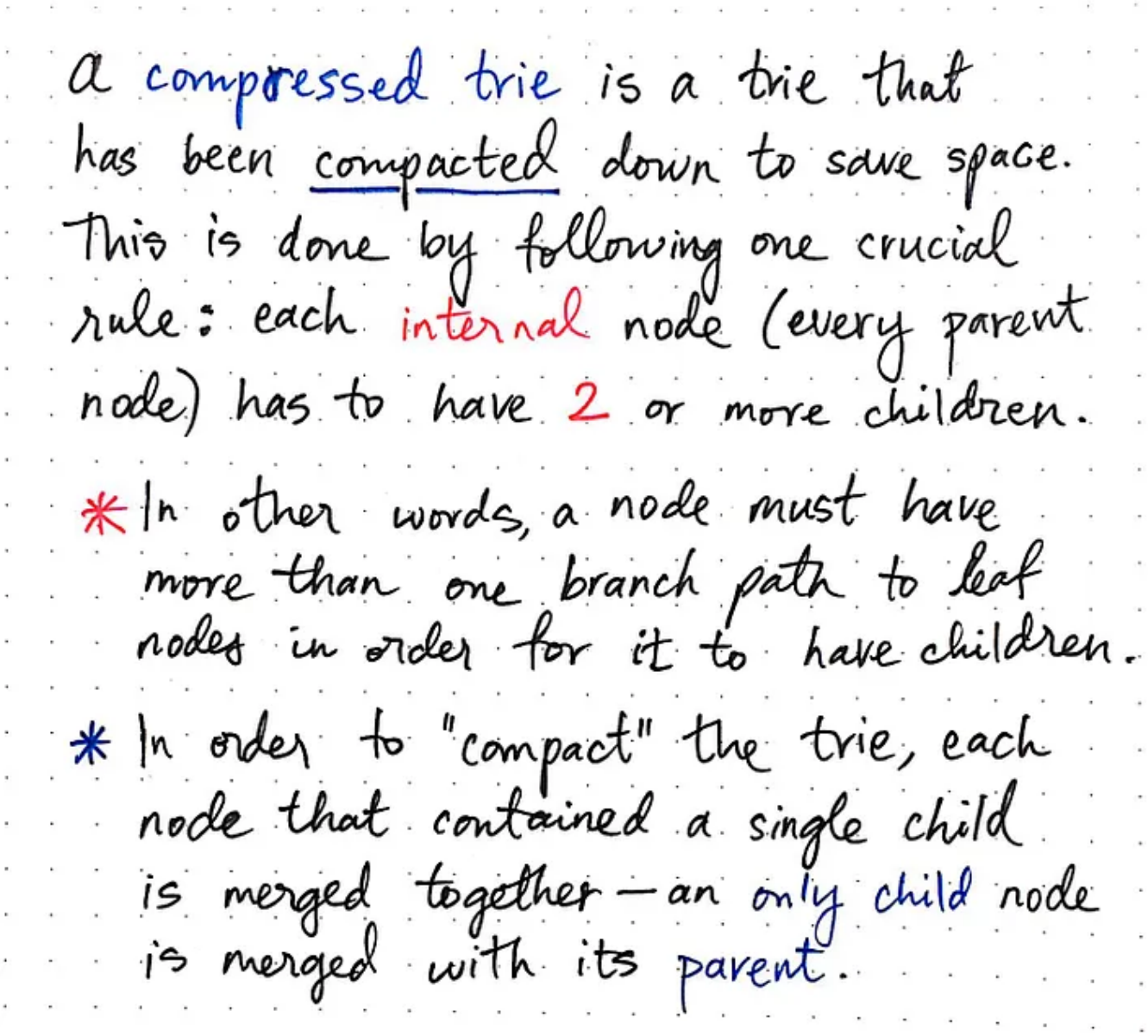
A redundant node is one that takes up an undue amount of space because it only has one child node of its own. We'll see that for the key `"deck"`, the node for the character `"e"` is redudant, because it only has a single child node, but we still have to initialize an entire node, with all of its pointers, as a result.

Similarly, the edges that connect the key `"did"` are redundant, as they connect redundant nodes that don't *really* all need to be initialized, since they each have only one child node of their own.

In some standard tries — like the one we're dealing with above — we often find that we're repeating ourselves quite a bit.

> The redundancy of a standard trie comes from the fact that we are repeating ourselves by allocating space for nodes or edges that contain only one possible string or word. Another way to to think about is that we repeat ourselves by allocating a lot of space for something that only has one possible branch path.

This is where compression comes in. We can compress our trie so that we neither repeat ourselves, nor use up more space than is necessary. A ***compressed trie*** is one that has been compacted down to save space.



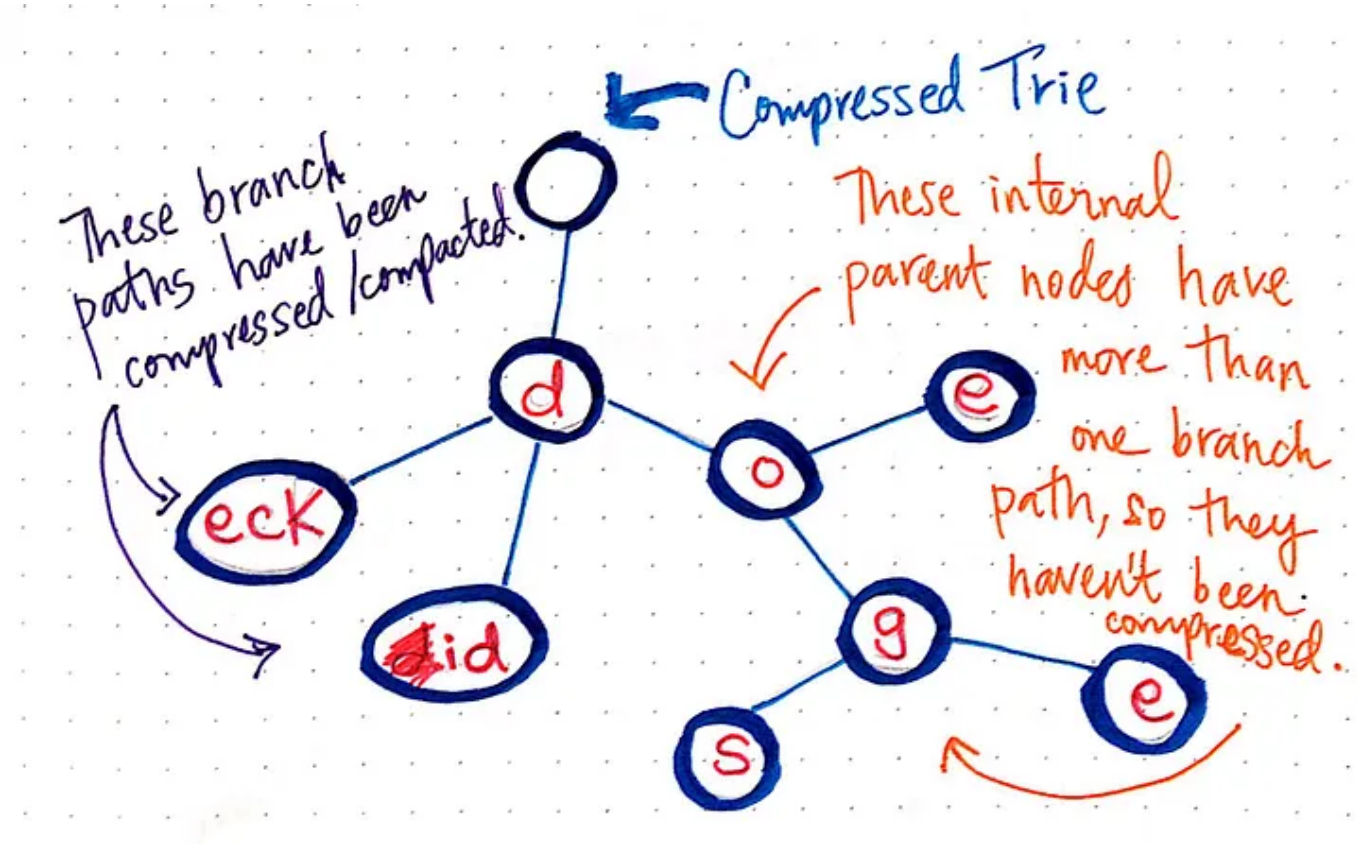
Compressed tries: a definition

The single most important rule when it comes to knowing *what* we can compress comes down to this: *each internal node (every parent node) must have two or more child nodes*. If a parent has two child nodes, which is at two branch paths to potential leaf nodes, then it doesn't need to be compressed, since we actually need to allocate space and memory for both of these branch paths.

However, if a parent node only has one child node — that is to say, if it only has one possible branch path to a leaf node — then it can be compressed. In

order to do the work of "compacting" the trie, each node that is the only child of its parent node is merged into its parent. The parent node and the single-child node are fused together, as are the values that they contain.

We can start compacting our standard trie example from earlier and compress the "single branch paths" like this:

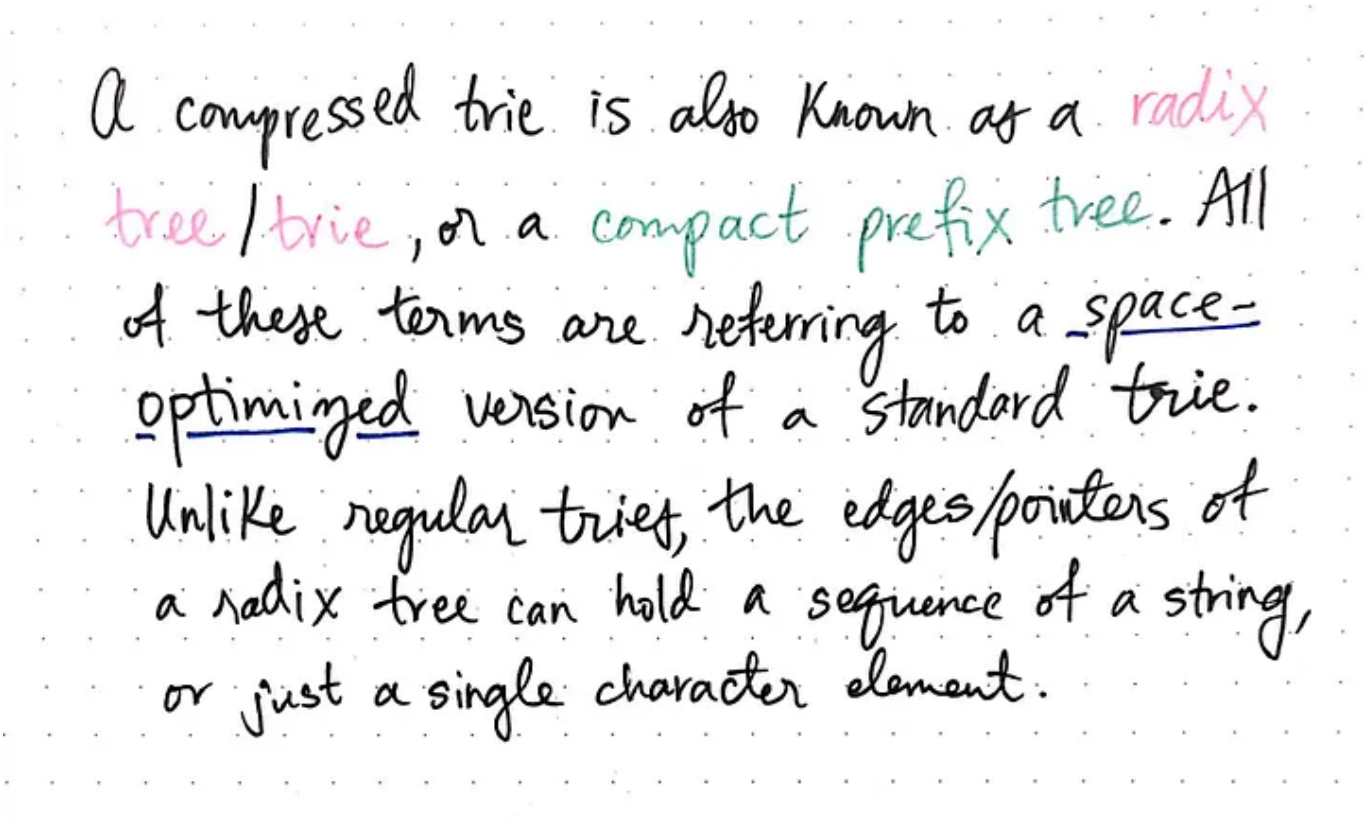


A compressed trie

We'll notice that we've compacted the nodes that formerly contained `"e"`, `"c"`, `"k"` so that the node `"d"` has a reference to `"eck"` as a single string contained in one node, rather than three. It's a similar story with the key `"did"`, which has been compressed. The internal parent nodes that haven't been compressed have more than one branch path; however, those internal nodes with single child nodes have been compacted down, and we've reduced the number of nodes we previously needed.

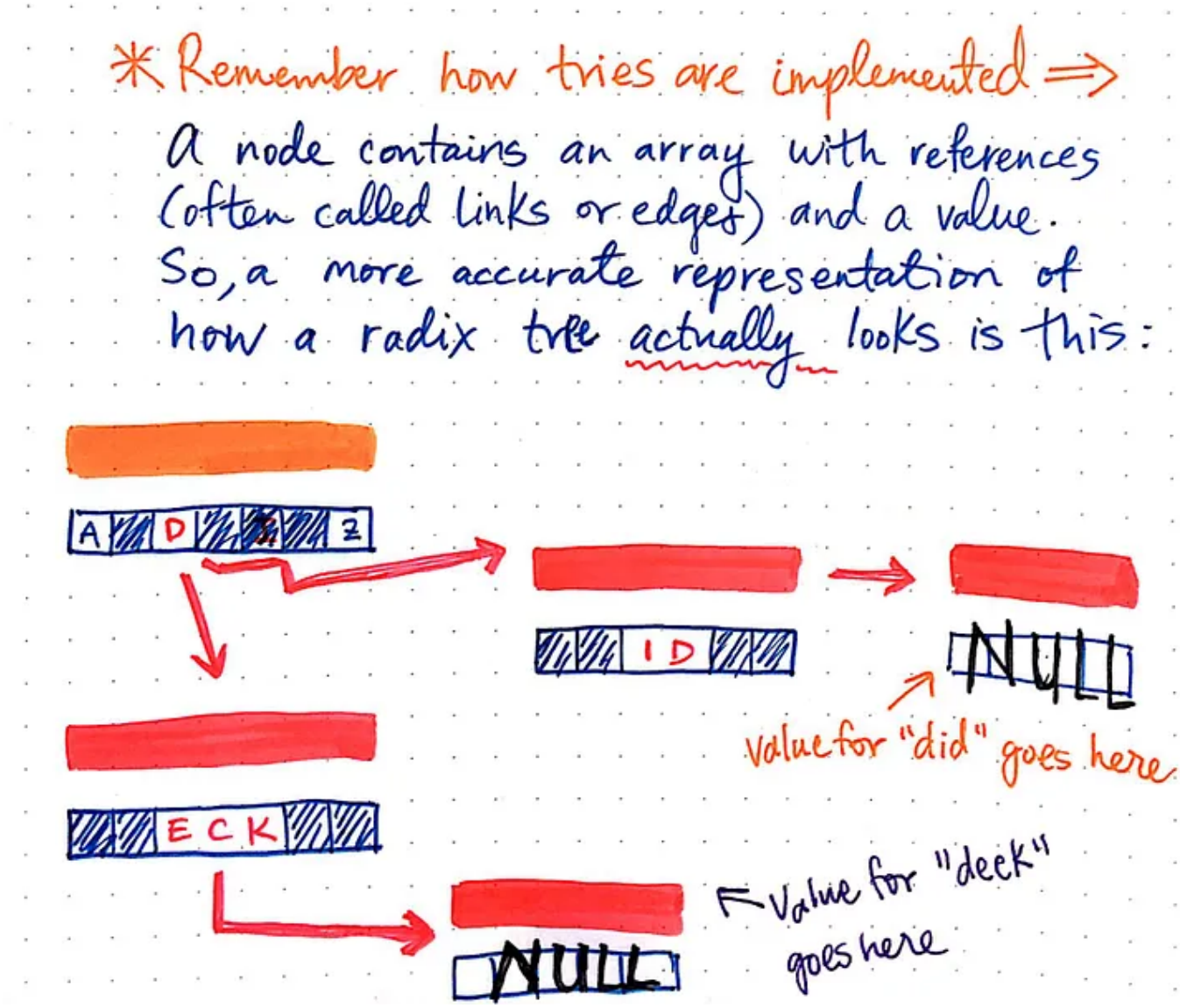
Radix trees

The idea of "trie compression" is a concept that is well-known enough to warrant its own name. Compressed tries are also known as ***radix trees***, ***radix tries***, or ***compact prefix trees***. (I know, I know, first it was "tries", now we're back to "trees", I don't know why no one can agree on what to call anything!)



A compressed trie is also known as a radix tree

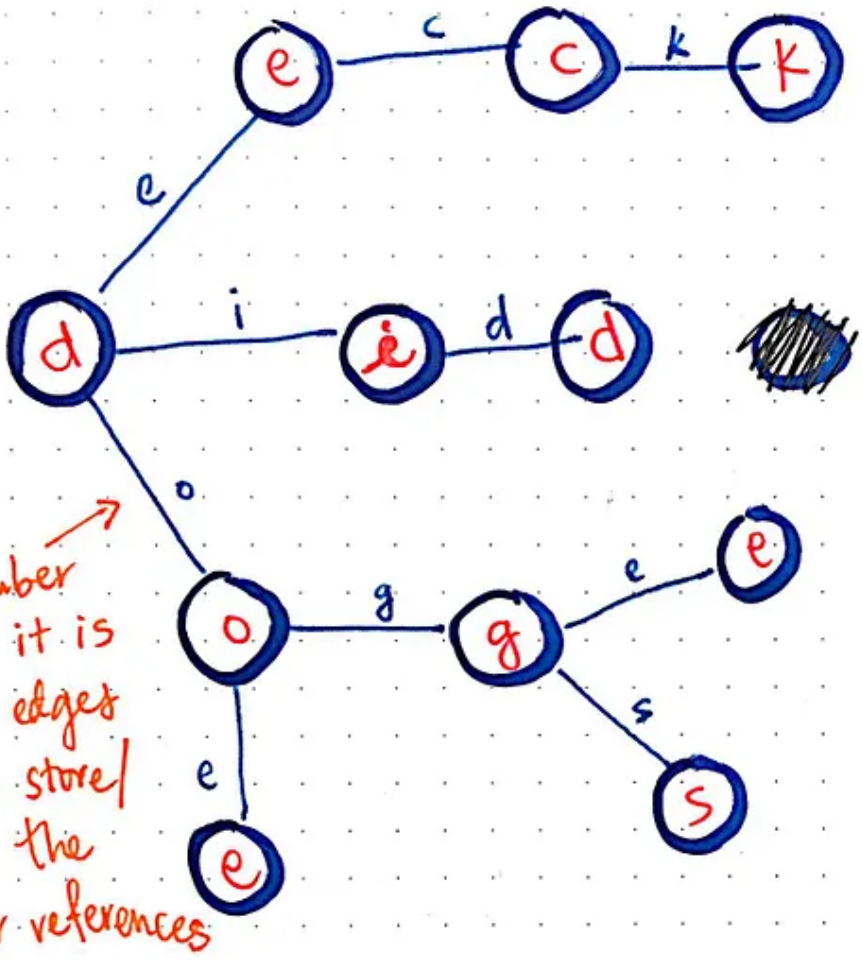
All of these terms really just refer to one thing: *a space-optimized version of a standard trie*. Unlike regular tries, the references/edges/pointers of a radix tree can hold a sequence of a string, and not just a single character element.



How tries are implemented under the hood

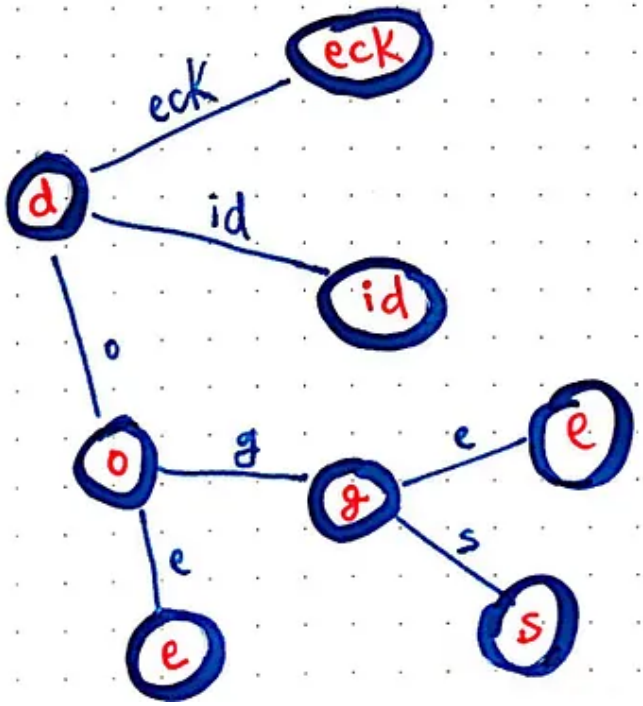
It's important to point out that, even though I have been illustrating tries as though the string characters of a word live in the word, we should remind ourselves quickly how tries actually work. Remember that a single node in a trie contains an array with references, and a value.

The illustration above is a far more accurate representation of what's going on under the hood of a radix tree. Notice how a both a single-character value and a string value are all contained in the references to other nodes. When the string terminates, the last reference points to an empty value/ `null` .



remember that it is the edges that store/label the letter references

a standard trie needs (11) nodes to represent ["doe", "dogs", "dog", "doge", "did", "deck"].



a radix tree needs (8) nodes to represent ["doe", "dogs", "dog", "doge", "did", "deck"].

A standard trie versus a radix tree representation in node count.

Since we already know how to compress a standard trie, we should probably figure out just how useful the work of compressing a trie actually is. In order to understand how a radix tree is more useful in terms of saving memory, we can compare our two tries — the standard trie we started off with, and the radix tree that we converted it into.

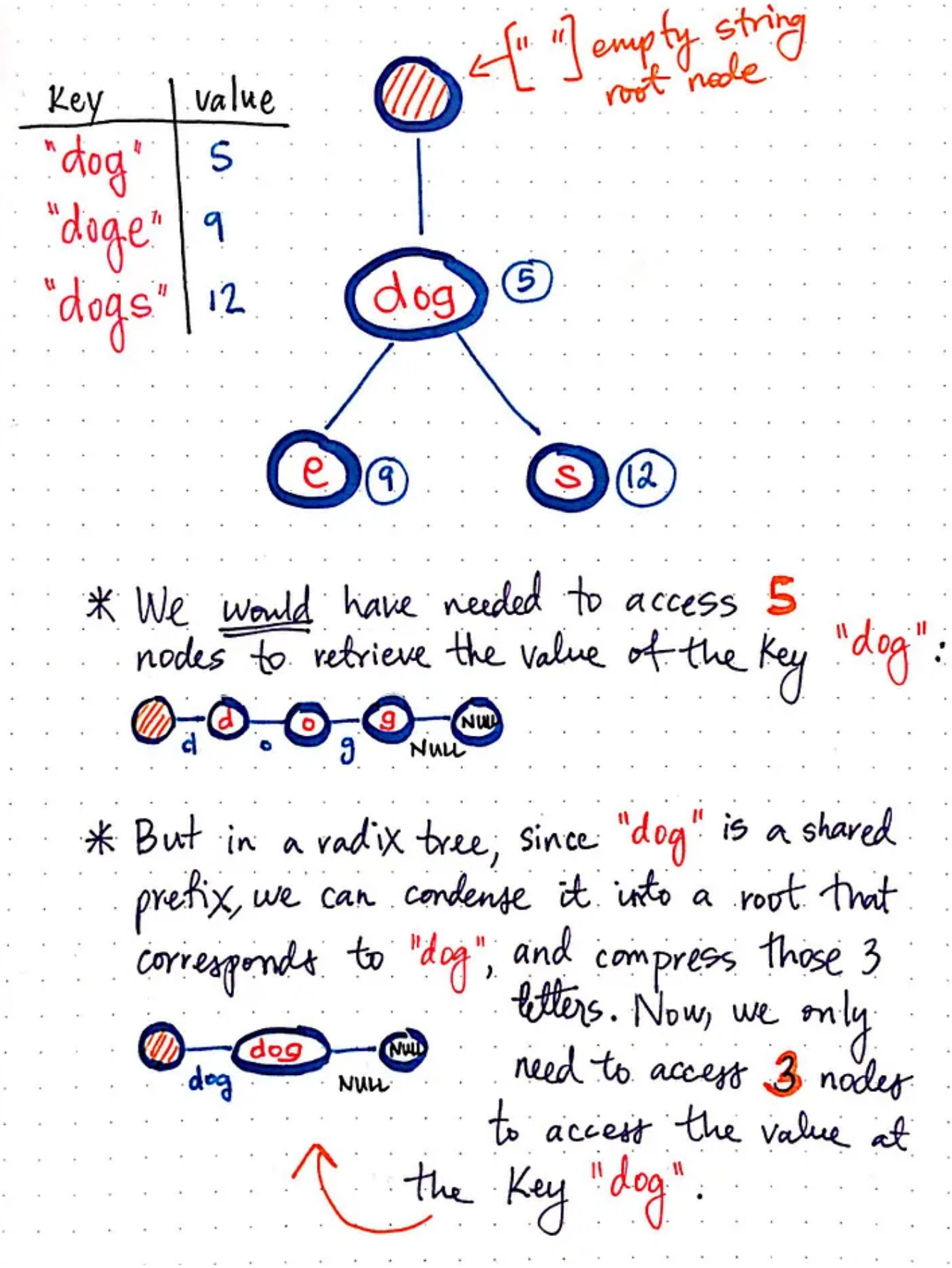
We can see that in our standard trie, we needed 11 nodes in order to represent the group of keys: `["deck", "did", "doe", "dog","doge", "dogs"]`.

However, when we compressed our standard trie down into a radix tree such every single child node was merged with its parent, we only needed 8 nodes to represent that same group of keys. Recall that every time we intialize a node, we also need to initialize an array with (usually) 26 keys, one for each of the 26 letters of the English alphabet. By eliminating 3 nodes in the process of compression, we saved that much space and memory that we would have needed to allocate otherwise.

We can also see the power of compression in radix trees when it comes to accessing and retrieving keys. In the example shown here, we've slimmed down our radix tree so that we're just working with three keys now: `"dog"`, `"doge"`, and `"dogs"`. Each of these three keys has a value, so the structure that we're *really* trying to represent here looks like this:

```
{
  "dog": 5,
  "doge": 9,
  "dogs": 12
}
```

Since we now know how to write and represent these key value pairs in radix tree format, we can draw out what that radix tree would look like.



Node access and key retrieval in a standard trie vs radix tree

We have an empty string as the value of our root node, with a reference to the string `"dog"`, which points to a single parent node, with a value of `5`.

When we had written this as standard trie earlier, we needed to access 5 different nodes (including the root node) in order to retrieve the value for the node at the value ending in `"g"`, which is the last character of `"dog"`.

But, in a radix tree, since `"dog"` is a shared prefix, we can condense it down into a single root node, which eliminates some of the redundant nodes that we had before. Now, we only need to access 3 nodes in order to access the value for the key `"dog"`. Remember that even though we have a single node at the reference point for `"dog"`, we still need an additional terminating node with a `null` reference in order to store the actual value for this key.

The power of radix trees comes from the fact that we compress down substrings based on their prefixes. Even from just this small example, we can see how the substring `"dog"` here can save two nodes and their corresponding space by compressing the trie into its radix tree format.

Back to bit and byte basics

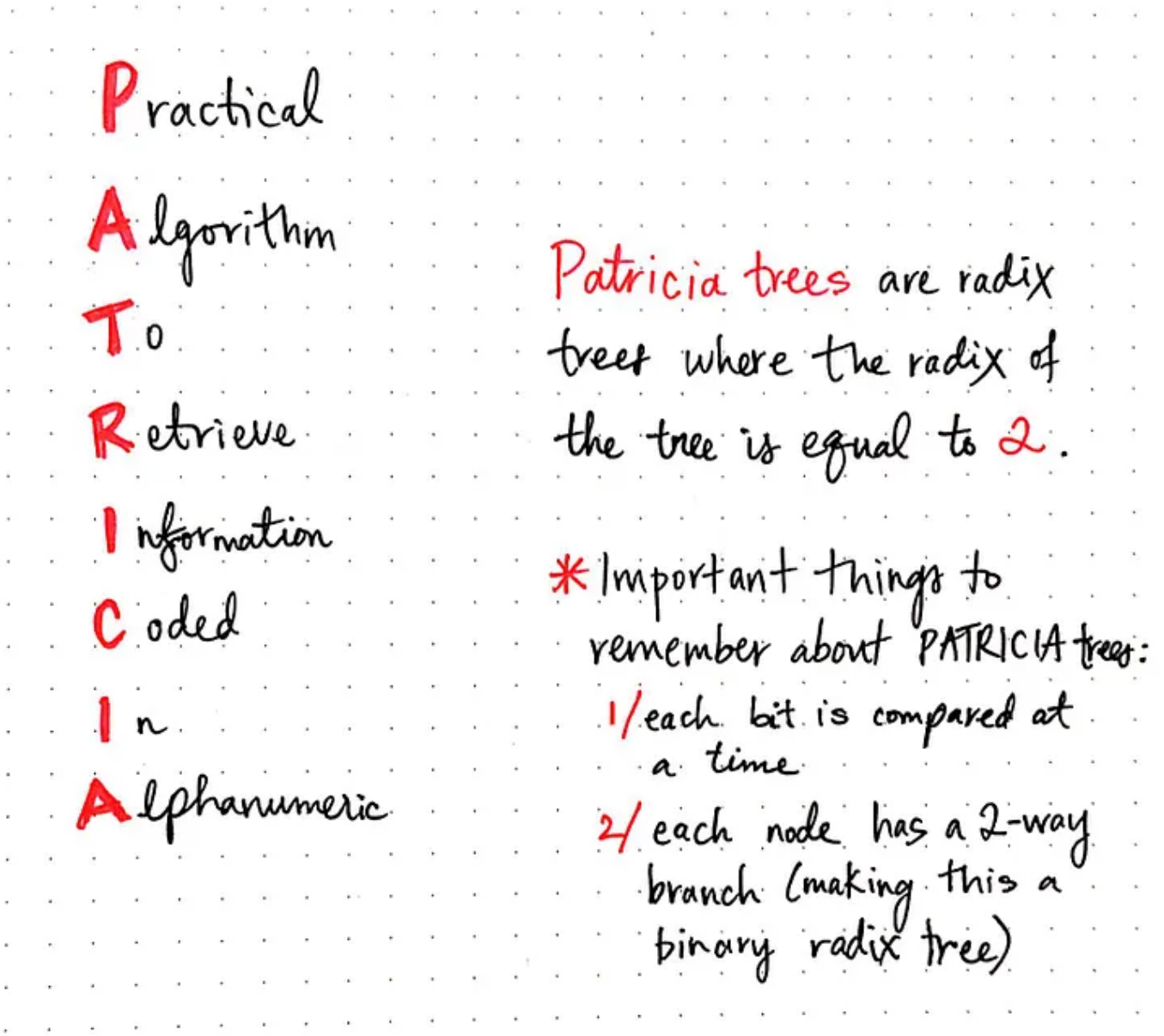
We've answered many questions about radix trees, but there's one thing that still remains a mystery: what does the *radix* have to do with the tree? In order to answer this question, we need to get back to the basics a little bit.

Some of the very first core concepts that we learned about were bits, bytes, and building with binary. We've run into binary a few times over the course of this series, and guess what? It's back again!

> In order to understand the radix of a tree, we must understand how tries are read by our machines. In radix trees, keys are read in bits, or binary digits. They are compared r bits at a time, where 2 to the power of r is the radix of the tree.

A trie's keys could be read and processed a byte at a time, half a byte at a time, or two bits at a time. However, there is one particular type of radix tree that processes keys in a really interesting way, called a **PATRICIA tree.**

The PATRICIA tree was created in 1968 by Donald R. Morrison, who coined the acronym based on an algorithm he created for retrieving information efficiently from tries; PATRICIA stands for "*Practical Algorithm To Retrieve Information Coded In Alphanumeric*".

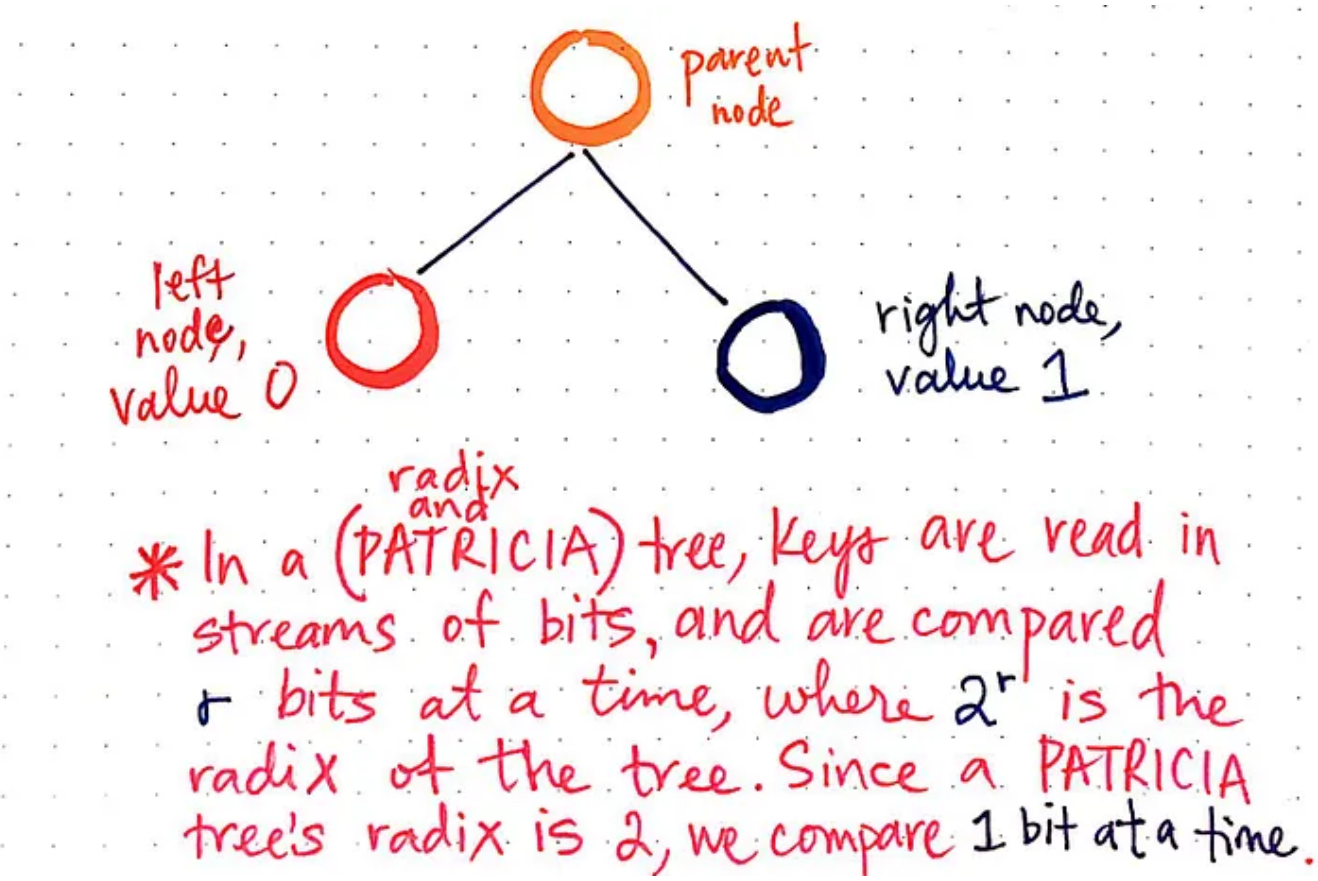


PATRICIA trees

The reason that PATRICIA trees are so interesting are because of the way that they process keys. And, as we've learned, the way that key are processed are tied directly to the radix of a radix tree.

The most important thing to remember about a PATRICIA tree is that its radix is 2. Since we know that the way that keys are compared happens *r* bits at a time, where 2 to the power of *r* is the radix of the tree, we can use this math to figure out how a PATRICIA tree reads a key.

Since the radix of a PATRICIA tree is 2, we know that *r* must be equal to 1, since $2^1 = 2$. Thus, a PATRICIA tree processes its keys ***one bit at a time***.



PATRICIA trees read keys 1 bit at a time

Because a PATRICIA tree reads its keys in streams of bits, comparing 1 bit at a time, the entire tree is built up to represent binary digits. If you remember learning about binary search trees, you remember that a binary tree can only have two children, with the value of the left node being less than the value of the right node. So, in a binary radix tree, the right node is always used to represent binary digits (or bits) of 1, and the left node is used to represent bits of 0.

Because of this, each node in a PATRICIA tree has a 2-way branch, making this particular type of radix tree a ***binary radix tree***. This is more obvious with an example, so let's look at one now.

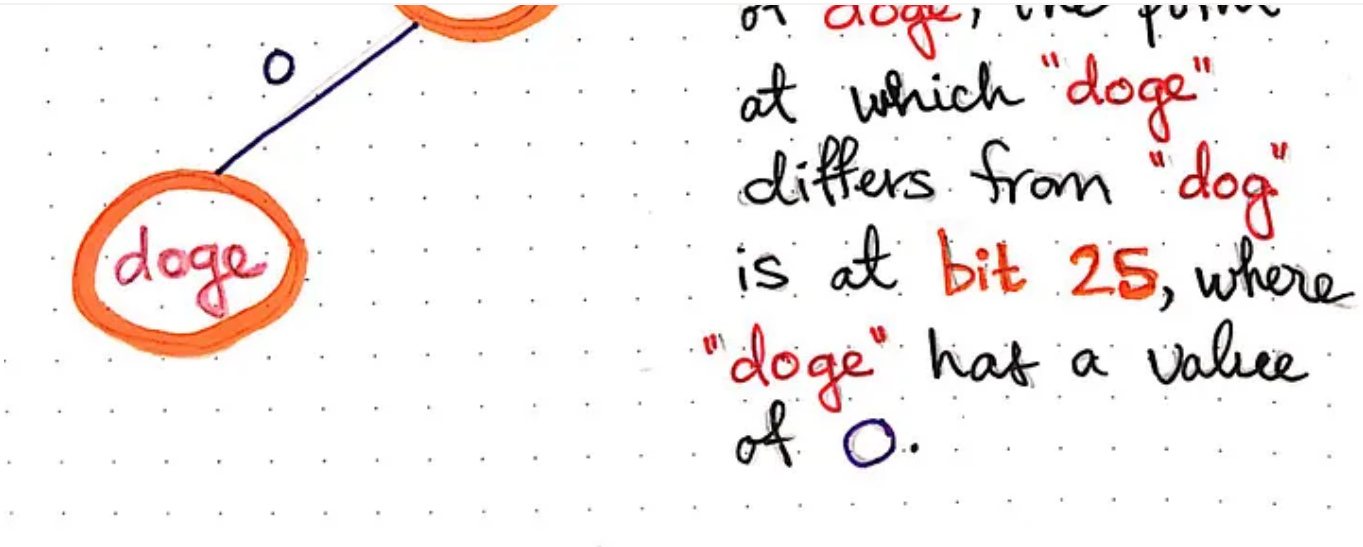
Let's say that we want to turn our original set of keys, `["dog", "doge", "dogs"]` into a PATRICIA tree representation. Since a PATRICIA tree reads keys one bit at a time, we'll need to convert these strings down into binary so that we can look at them bit by bit.

```
dog:  01100100 01101111 01100111
doge: 01100100 01101111 01100111 01100101
dogs: 01100100 01101111 01100111 01110011
```

Notice how the keys `"doge"` and `"dogs"` are both substrings of `"dog"`. The binary representation of these words is the exact same up until the 25th digit. Interestingly, even `"doge"` is a substring of `"dogs"`; the binary representation of both of these two words is the same up until the 28th digit!

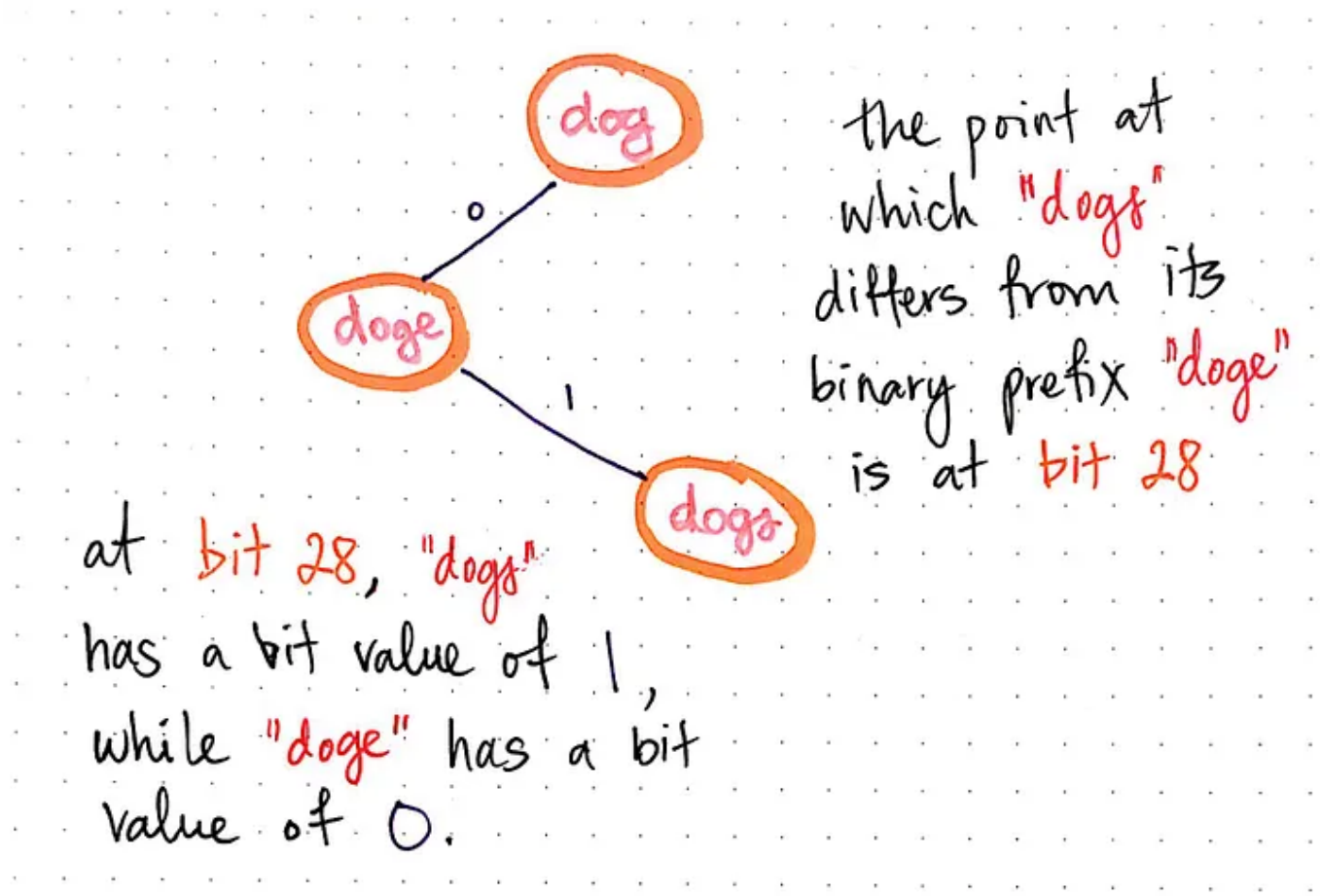
Pretty cool, right!? Who knew that `"doge"` was a binary substring of `"dogs"`?

Okay, so since we know that `"dog"` is a prefix of `"doge"`, we will compare them bit by bit. The point at which they diverge is at bit 25, where `"doge"` has a value of `0`. Since we know that our binary radix tree can only have 0's and 1's, we just need to put `"doge"` in the correct place. Since it diverges with a value of `0`, we'll add it as the left child node of our root node `"dog"`.



Adding “doge” to our PATRICIA tree

Now we’ll do the same thing with `"dogs"`. Since `"dogs"` differs from its binary prefix `"doge"` at bit 28, we’ll compare bit by bit up until that point.



Adding “dogs” to our PATRICIA tree

At bit 28, `"dogs"` has a bit value of `1`, while `"doge"` has a bit value of `0`. So, we’ll add `"dogs"` as the right child of `"doge"`.

This is, of course, a simplified version of Donald R. Morrison’s algorithm, which is far more complex. But ultimately, that’s what’s happening at a very low level! His *Abstract on Practical Algorithm To Retrieve Information Coded in Alphanumeric* succinctly explains the basic idea behind PATRICIA and where it has been used:

> *PATRICIA is an algorithm which provides a flexible means of storing, indexing, and retrieving information in a large file, which is economical of index space and of reindexing time. It retrieves information in response to keys furnished by the user with a quantity of computation which has a bound*

> *which depends linearly on the length of keys and the number of their proper occurrences and is otherwise independent of the size of the library. It has been implemented in several variations as FORTRAN programs for the CDC-3600, utilizing disk file storage of text. It has been applied to several large information-retrieval problems and will be applied to others.*

Radix and PATRICIA trees can be a difficult structures to understand because they involve abstract concepts, but leverage very low-level (quite *literally* binary) things.

To be totally honest, this was the first topic in this series where I truly struggled to understand a concept. But when I finally *did* comprehend how things were working, it was pretty cool to see how small building blocks are the very things that make up bigger constructs that power parts of the our world.

But if this topic makes you want to cry, don't worry — it nearly brought me to tears, too.

I almost cried while writing about radix trees so that you don't have to!

Resources

It's pretty tricky to find approachable resources on radix trees, but if you dig deep enough around the internet, they *do* exist. Luckily, you don't need to dig for them, because I already did! Here are some good places to start reading if you want to learn more about binary radix and PATRICIA trees.

1. Compressed tries (Patricia tries), Professor Shun Yan Cheung
2. What is the difference between radix trees and Patricia tries?, Computer Science StackExchange
3. What is the difference between trie and radix trie data structures?, StackOverflow

4. Data Structures for Strings, Professor Pat Morin
5. PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric, Professor Donald R. Morrison
6. Dictionary of Algorithms and Data Structures — Patricia tree, Stefan Edelkamp
7. Tries, Patricia Tries, and Suffix Trees, Professor Yufei Tao

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

Written by Vaidehi Joshi

29K Followers · Editor for basecs

Writing words, writing code. Sometimes doing both at once.

Follow

More from Vaidehi Joshi and basecs