Vaidehi Joshi Follow
Writing words, writing code. Sometimes doing both at once.
Apr 17, 2017 · 10 min read

Taking Hash Tables Off The Shelf

Truth time: learning about theoretical things every week can get a bit monotonous. As much as it's important to learn the method behind the madness, it's equally crucial to understand why we're embarking upon the path of madness to begin with it. In other words—what's the *point* of the theory that we're learning here together?

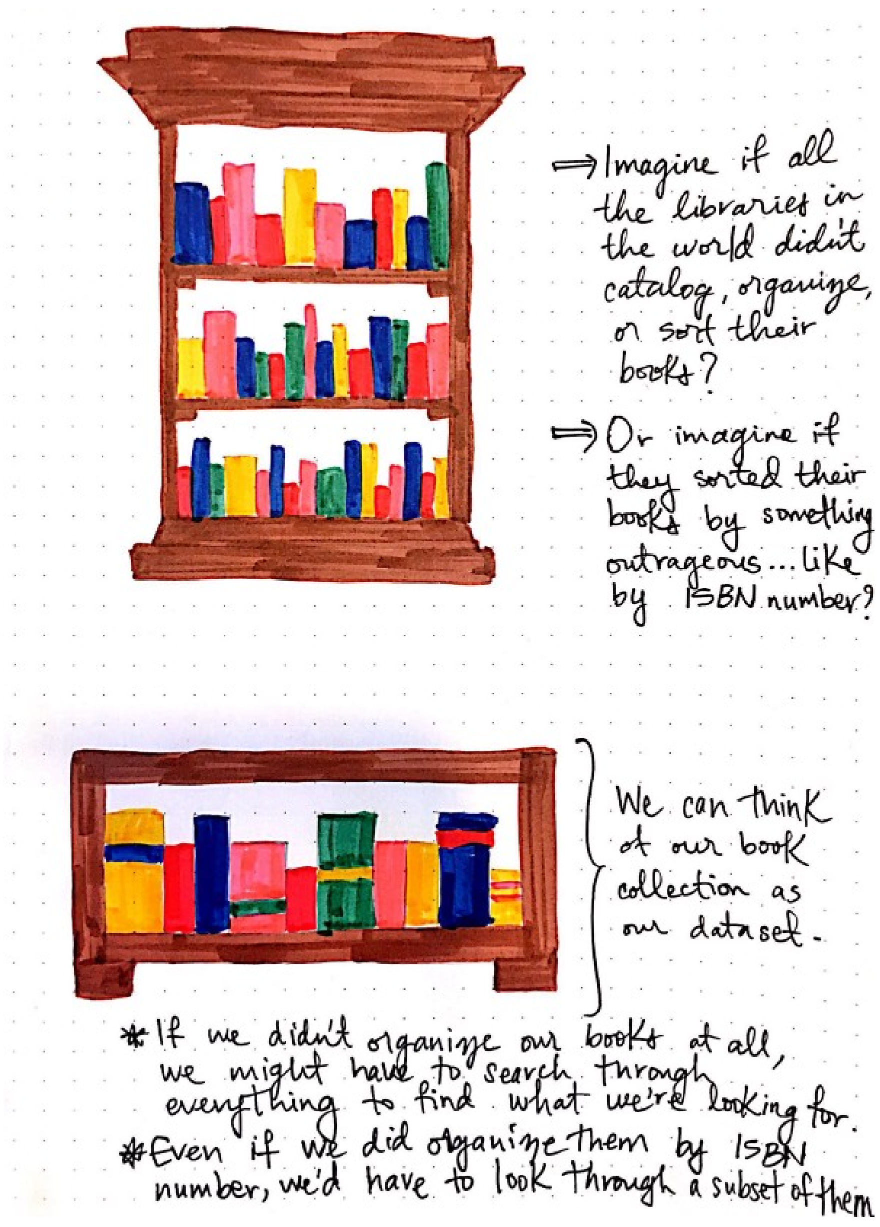
Sometimes, this can be a little tricky to illustrate. It's hard to understand breadth first search if you don't already know Big O notation. And Big O notation doesn't mean anything to you unless you've seen it in action, in the context of an algorithm.

But every so often, we get lucky in that we stumble across the most perfect metaphor. Metaphors can be a great way of learning through analogy, and they're useful in illustrating otherwise complicated concepts. I spent a few days trying to understand a data structure I was only recently introduced to, and it was only in the context of this metaphor that I really began to grasp it.

So, let's see if we can unpack together.

Books on books on books

If you've moved across the country recently like I have, you probably have vivid memories of the horrors of trying to unpack everything. Or, maybe you've blocked it all out, in which case: well done! I was unpacking all of my books last week, and found myself sticking them all on the shelves rather haphazardly, without any order or logic behind which book went where.


⟹ Imagine if all the libraries in the world didn't catalog, organize, or sort their books?
⟹ Or imagine if they sorted their books by something outrageous… like by ISBN number?
We can think of our book collection as our dataset.
* If we didn't organize our books at all, we might have to search through everything to find what we're looking for.
* Even if we did organize them by ISBN number, we'd have to look through a subset of them

We can think of our book collection as a dataset.

I own a lot of books, sure, but I don't have *that* many that it becomes difficult for me to find one. This got me thinking about libraries and their giant book collections, and how organized they actually have to be.

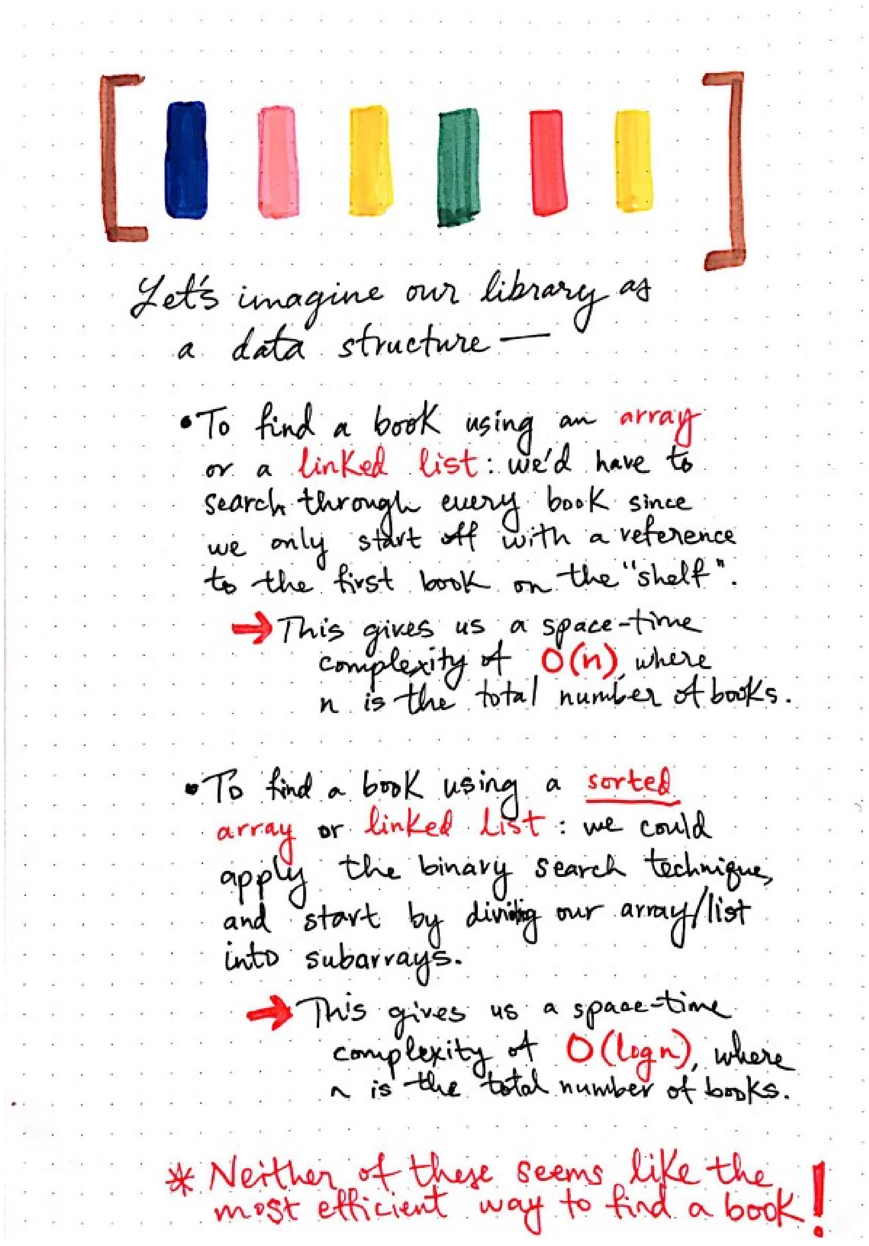
Imagine, for a moment, if all the libraries in the world didn't organize or sort their books? Or, imagine if they were sorted by something super complicated…like by ISBN number?

We're all better off because libraries are so methodically organized. But how did they come up with that? In fact, how does *anyone* come up with a good way of sorting through and finding an item within a large number of things?

Hopefully, a lightbulb just lit up in your head, and you're thinking the same thing that I am: *this is a computer science problem!*

If we abstract this problem into programmatic terms, we can think of our book collection as our dataset. You can imagine that if we didn't organize our books at all, there's a good chance that we'd have to search through every single book on the shelf before we find the book that we're looking for. And if we organized our books by ISBN number, or by the year that they were published, we'd still have to look through a subset of them, since it would still be a little difficult to find one item from amongst all the items on the shelf.

We're already familiar with so many different data structures at this point—perhaps we can use one of them to handle our dataset? Let's try them out for size.



Let's imagine our library as a data structure —

- To find a book using an array or a linked list: we'd have to search through every book since we only start off with a reference to the first book on the "shelf".
  → This gives us a space-time complexity of O(n) where n is the total number of books.
- To find a book using a sorted array or linked List: we could apply the binary search technique, and start by dividing our array/list into subarrays.
  → This gives us a space-time complexity of O(logn), where n is the total number of books.

* Neither of these seems like the most efficient way to find a book!

Imagine a library as a data structure

How about using an array or a linked list?

Well, that would probably work fine in terms of *holding* all of our data. However, does this really solve our problem of searching through our dataset? Let's think about it. In order to find a book using a linked list or an array, we'd still have to search through *every single node* in the worst case that the data that we were looking for was in the last node. This basically means that the time it takes for us to search through all of our books is directly related to how many books we have, which is *linear time* or O(n).

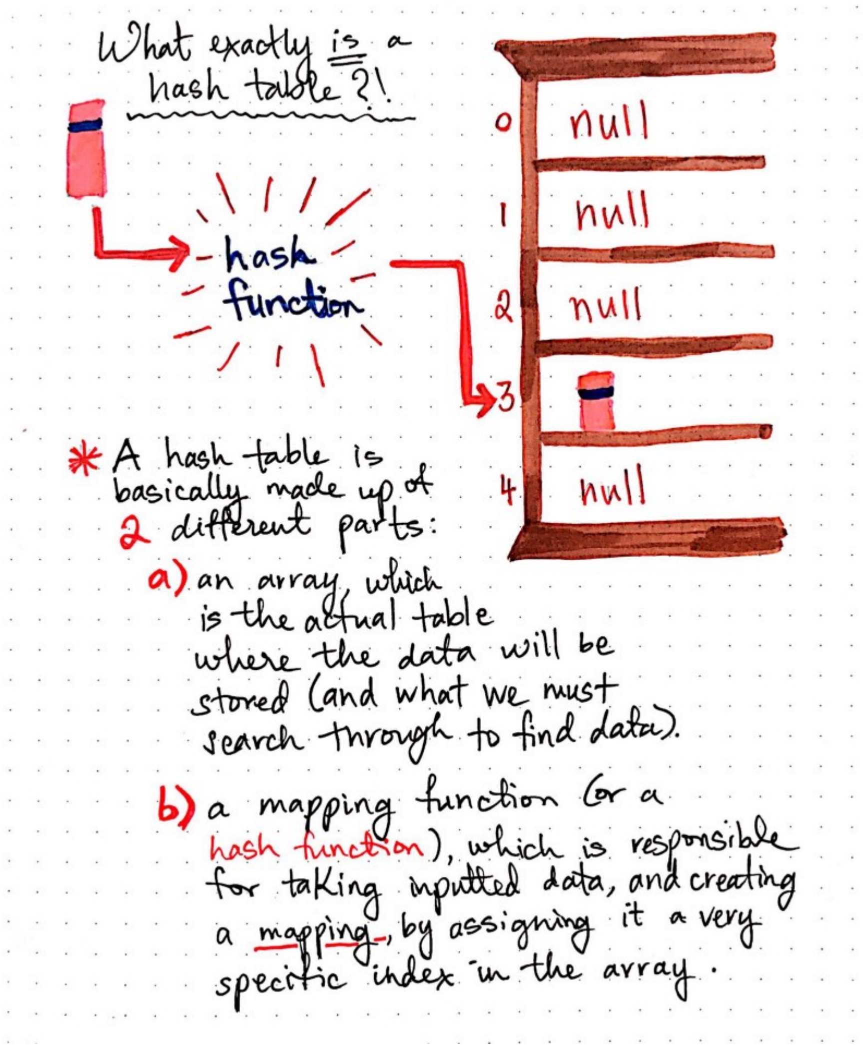
And what if we sorted our linked list, instead of adding in data randomly? Well, this would be a bit more helpful, since we could implement *binary search* on our linked list. Remember that we can apply binary search to any collection of sorted data, which is what we'd have in this case. We could split our large, sorted linked list or array into smaller sub-arrays, and narrow down for the book that we're looking for. Even in the best case scenario, however, binary search would only ever let us reach logarithmic time, or O(log n). This would be nicer than linear time, of course, but still…none of these really seem like the most efficient way to search for a book.

It would be awesome if there was a way to just look up a book *instantly* —without having to search through everything. Surely, there must be a better way of doing this.

Mappings to make our lives easier

Okay, okay, we're just learning about a cool data structure today—not inventing a new one! I guess I should finally let you in on the secret: there *is* a better way of searching through our dataset. And, as you might have guessed, it's called a **hash table**.

Hash tables are made up of two distinct parts: **an array**, which you're already familiar with, and **a hash function**, which is probably a new concept! So, why these two parts? Well, the array is what actually holds our data, and the hash function is more or less the way that we decide *where* our data will live—and how we'll get it out of the array, when the time comes!


What exactly is a hash table?!
hash function
0 null
1 null
2 null
3
4 null
A hash table is basically made up of 2 different parts:
a) an array, which is the actual table where the data will be stored (and what we must search through to find data).
b) a mapping function (or a hash function), which is responsible for taking inputted data, and creating a mapping, by assigning it a very specific index in the array.

What exactly is a hash table?

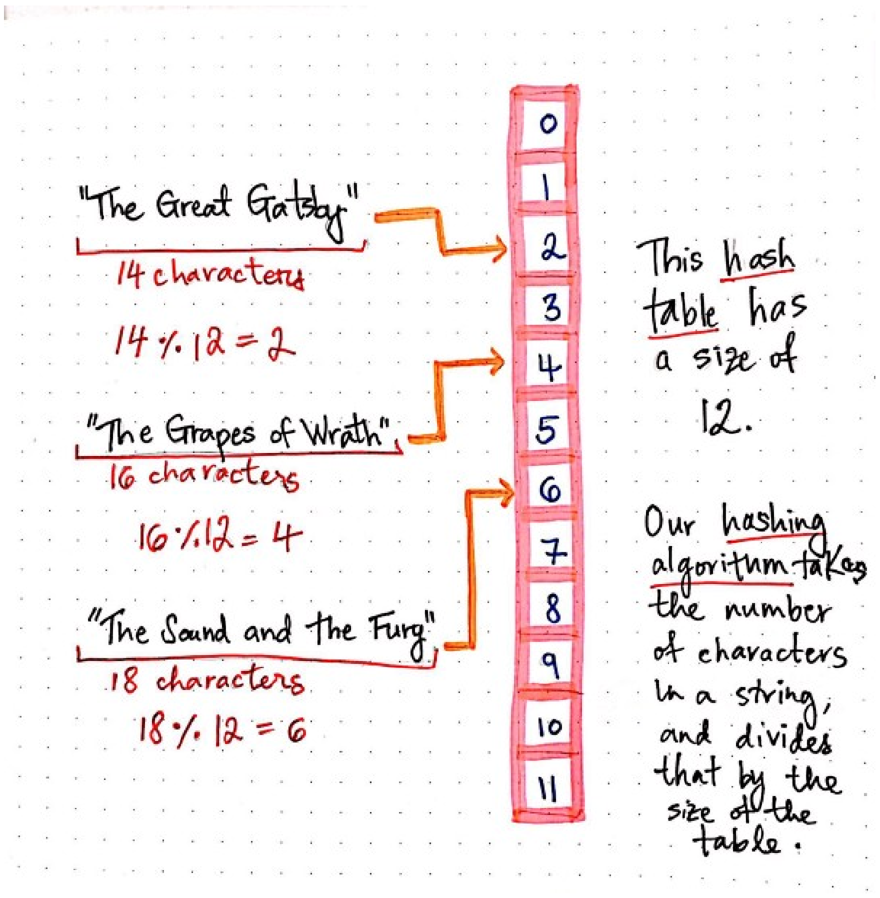
In the example above, we're restructuring our bookshelf to be a hash map! Our shelves start off empty, and in order to add books (our data) to them, we pass the book into a hash function, which tells us which shelf to put it on.

Hash tables are interesting for a lot of reasons, but what makes them so efficient for our purposes is that they create a **mapping**, which is a relationship between two sets of data. Hash tables are made up of sets of pairs: a key and a value, often referred to as *(k, v)*. The idea here is that if you have the key in one set of data, you can find the value that corresponds to it in another set of data.

> *In the case of a hash table, it is the hash function that creates the mapping. Since we're working with arrays,*

the key is the index of the array, and the value is the data that lives at that index. The hash function's job is to take an item in as input, and calculate and return the key for where that item should live in the hash table.

Alright: if we're really going to sink our teeth into this topic, we'll need an example. Let's abstract out our bookshelf from earlier and simplify it further. In the example below, we're looking at a hash table for book titles. The hash tables has its two predictable parts: an array, and a hash function.



A hash table with a size of 12

Our hash table has a size of 12, which ultimately means that every single book needs to be stored *somewhere* within these twelve slots, which are also referred to as **hash buckets**.

Wait a second: how is that hash function deciding where to put those books!? If you take a closer look, you'll notice that we're using the modulo operator `%` , which you might have already seen before. The modulo operator returns the remainder when one number is divided by another (or it returns 0, if the number can be divided evenly). So what is our hashing algorithm doing? Maybe you've already figured out the pattern here!

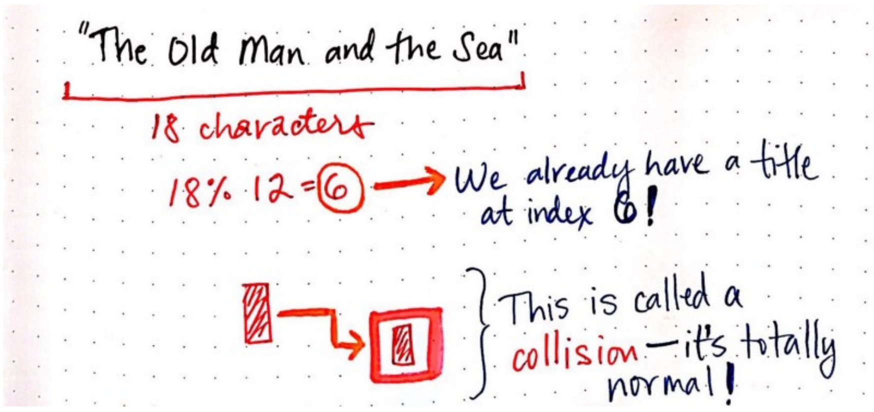
Our hash function takes the number of characters in the title, adds them up, and divides that summed total by the size of the table. For example, `"The Great Gatsby"` has 14 characters in it (ignoring spaces), which, when divided by the 12, the size of the table, gives us a remainder of `2` . So, our hash function will determine that `"The Great Gatsby"` should live in the hash bucket with an index of `2` .

We can write out what our hash function would actually look like in code:

```
function bookHashing(bookTitle, hashTableSize) {
  // Remove any spaces from book title.
  var strippedBookTitle = bookTitle.replace(/\s/g, ''
  // Divide the length of the title by the hash table
  // Return the remainder.
  return strippedBookTitle.length % hashTableSize;
}

bookHashing("The Grapes of Wrath", 12);
```

Alright, that's simple enough! In fact, it's almost…*too* simple. I feel like something's going to go wrong now. Let's keep going and see how far we can get with our hash function. We'll try another book title: `"The Old Man and the Sea"` :


"The Old Man and the Sea"
18 characters
18% 12 = 6
We already have a title at index 6!
This is called a collision — it's totally normal!

A collision occurs when two elements are supposed to be inserted at the same place in an array.

Oh no. I had a feeling this was going to happen!

When we input `"The Old Man and the Sea"` into our hash function, it returns `6`. This is bad because we already have a `"The Sound and the Fury”` at hash bucket `6`! That *is* bad, right?

Well, as it turns out, this isn't bad at all. It's actually quite normal. In fact, it's so normal that this phenomenon even has a name: a **collision**, which occurs when two elements are supposed to be inserted at the exact same place in an array—in other words, at the same hash bucket.

What does this tell us about our hash function?

> *For starters, we can be sure that no hash function will always return a unique hash bucket value for every item that give it. In fact, a hash function will always input multiple elements to the same output (hash bucket), because the size of our dataset will usually be larger than the size of our hash table.*

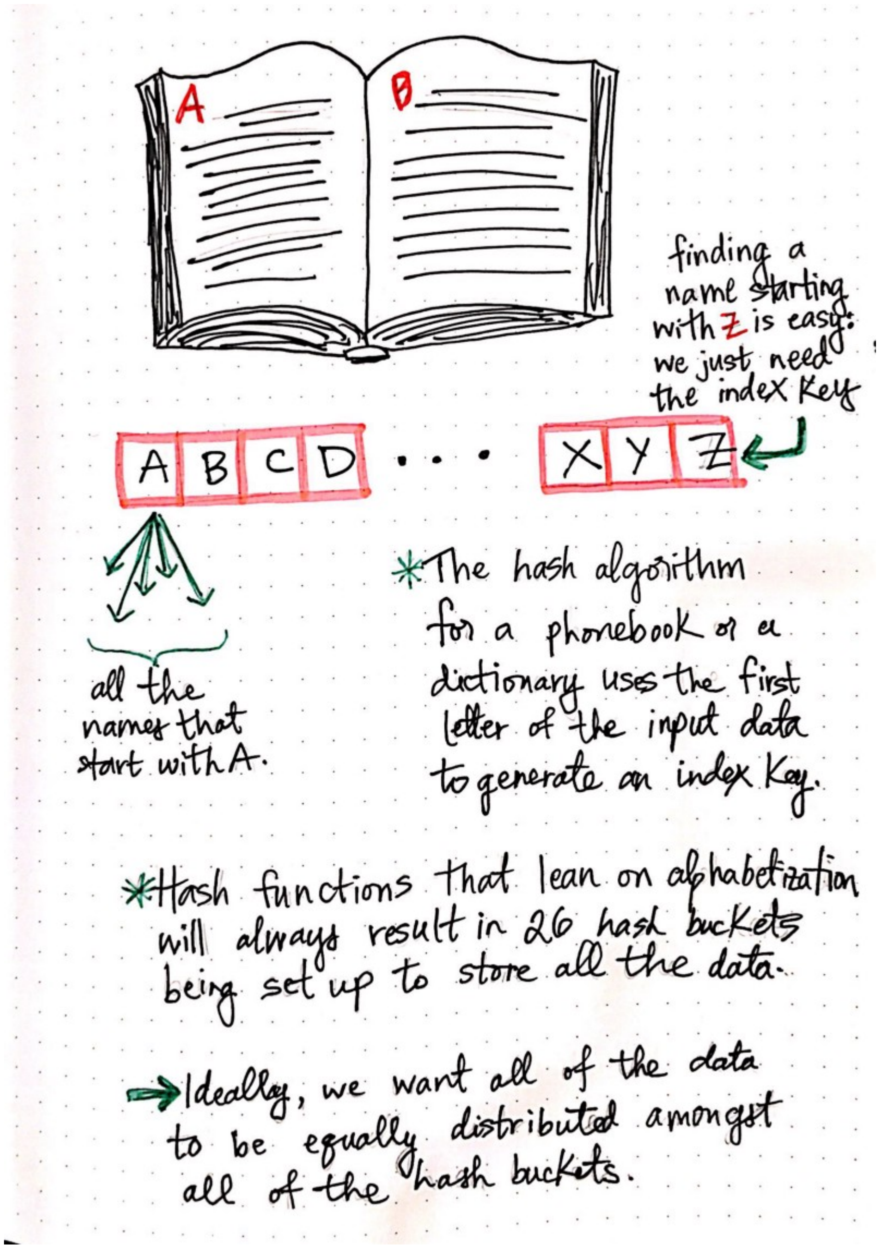
But why is this the case?

Super cool constant time

If we think more about hash tables in a practical context, it becomes pretty clear that we will pretty much *always* have multiple elements per hash bucket.

A good example of a hash table being applied to a real-world problem is a phonebook. A phonebook is massive, and contains all the data and contact information of every person that lives in a certain area.

How is it organized? By last name, of course. Each letter of the alphabet will have many people with that last name, which means that we'll have many pieces of data at each letter's hash bucket.

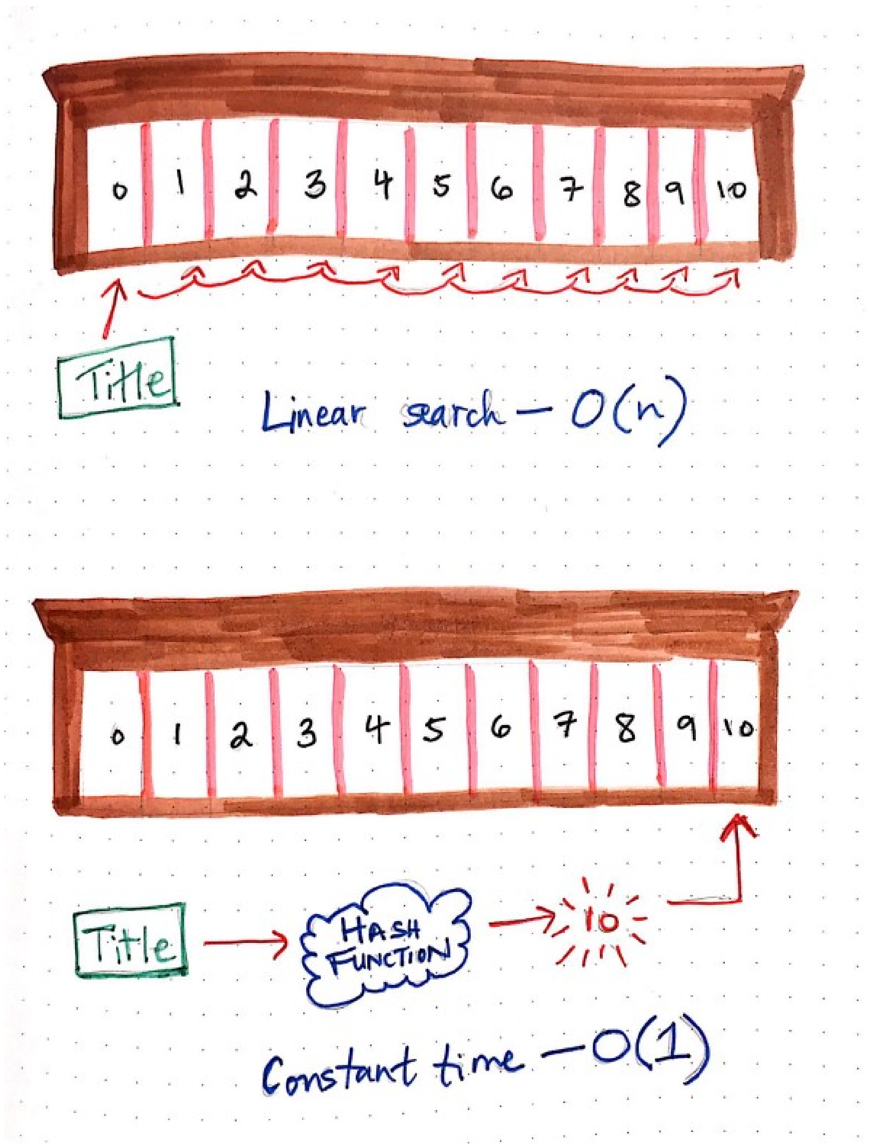

A
B
finding a name starting with Z is easy: we just need the index key
A B C D . . . X Y Z
all the names that start with A.
*The hash algorithm for a phonebook or a dictionary uses the first letter of the input data to generate an index key.
*Hash functions that lean on alphabetization will always result in 26 hash buckets being set up to store all the data.
Ideally, we want all of the data to be equally distributed amongst all of the hash buckets.

Hash functions that lean on alphabetization always result in 26 hash buckets.

The hash function for a phonebook or a dictionary will always group any inputted data into one of 26 buckets—one bucket for each letter of the alphabet. This ends up being rather interesting, because what this effectively means is that we'll never have a hash bucket (or in this case, a letter of the alphabet) that is empty. In fact, we'll have *close to* an even distribution of last names that start with each letter of the alphabet.

In fact, that's an important part of what makes a good hash algorithm: *an equal distribution of data amongst all the possible hash buckets.* But hold that thought for now—we'll get more into that next week!

For now, let's look at the *other* cool part about hash tables: their efficiency! Whether we're dealing with a phonebook, a dictionary, or a library, the approach remains the same: our hash function will be what tells us *where* to put our data, and also *where* to get it out from!



Hash tables leverage constant time search, which is what makes them so powerful.

Without using hash tables, we have to search through an entire dataset, probably starting with the first element in an array or linked list. If our items is at the very end of the list—well, good luck! We could be searching for a *long* time. In fact, as much time as there are elements! Herein lies the major problem with using linked lists, stacks, queues, arrays: searching through them for something always ends up being linear in time, or O(n).

Comparing this to hash tables, the choice between the two seems pretty easy, now that we understand how hash functions work! If we're looking for an element, we need only to pass that element into the hash function, which will tell us exactly what index in the array—that is to say, which