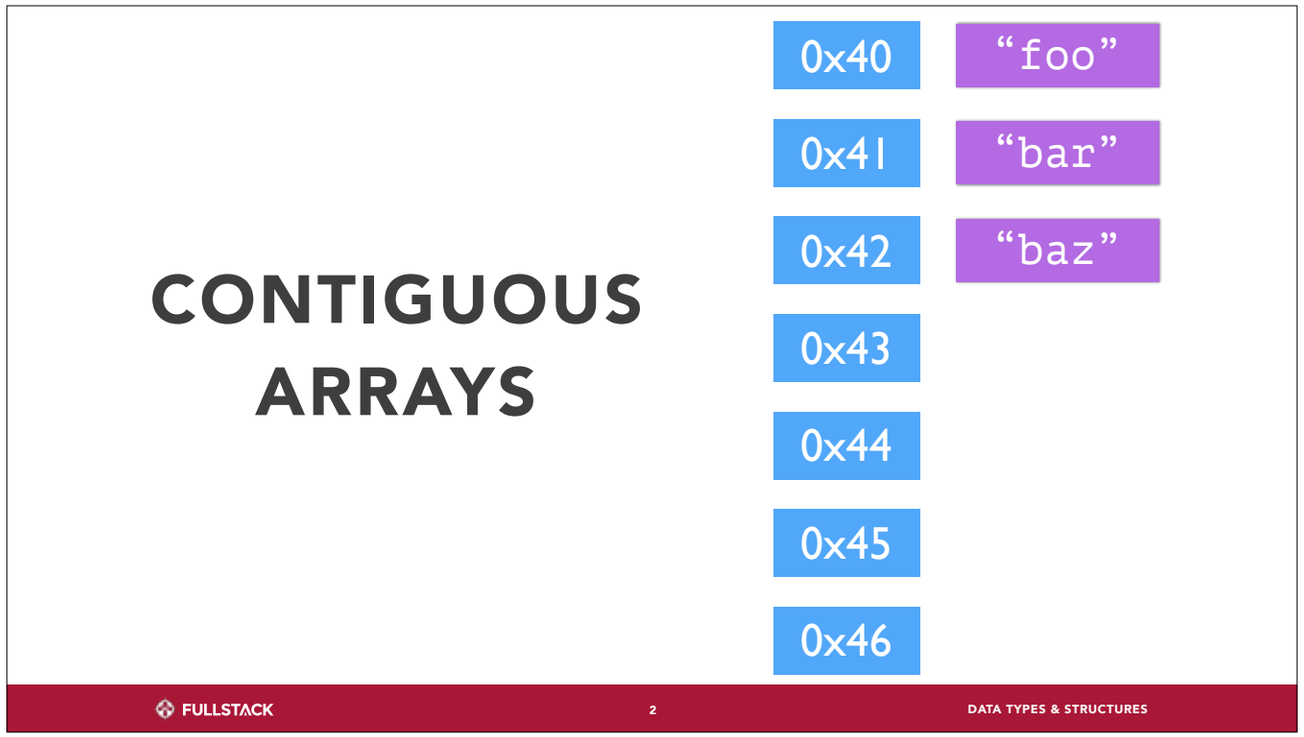
HASH TABLES

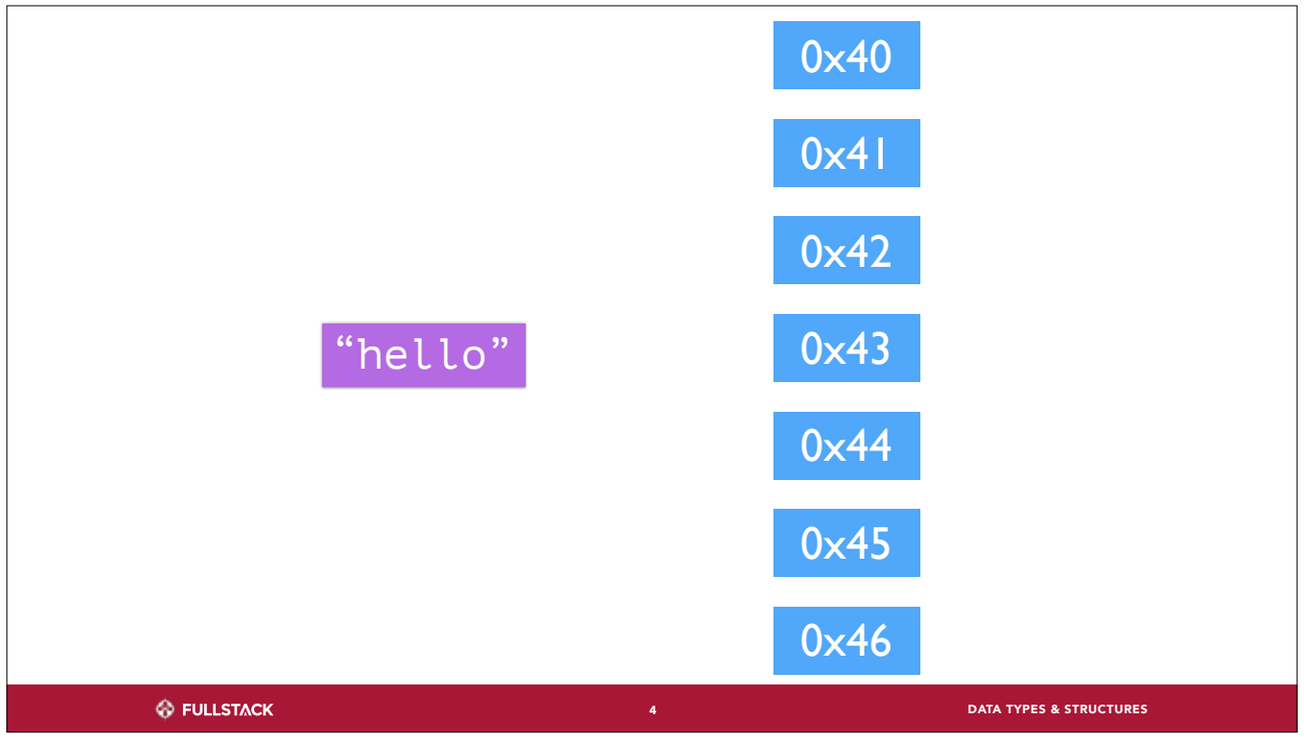
Let’s talk about the Hash Table data structure.



At this point, we’re familiar with the concept of a “contiguous array” - that is, storing data in adjacent cells in memory.

PROBLEMS WITH CONTIGUOUS ARRAYS

However, let’s consider some of the problems that come from storing data this way.

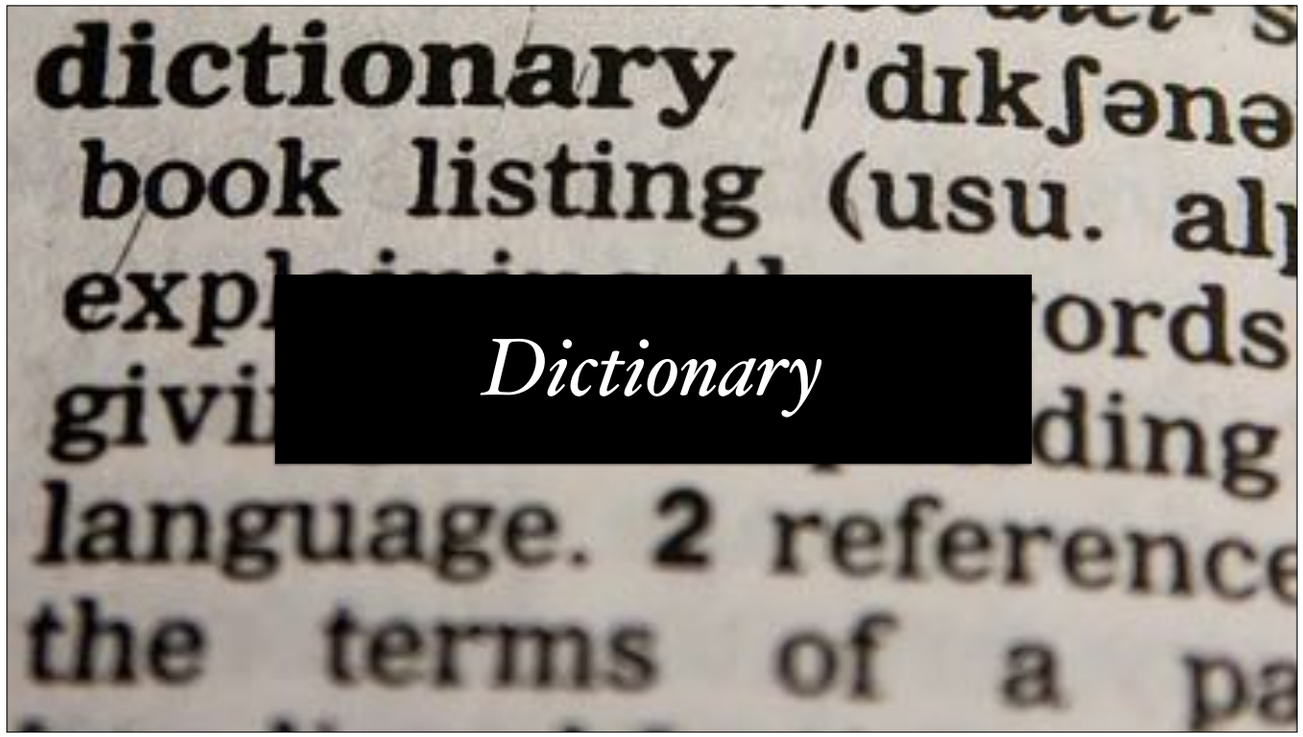


For example, say we set aside an array with 7 memory cells - (0 to 6). And we want to store some information there, like this string that says hello. We can go ahead and put it there, but if we want to find it later, we have to check one-by-one through the array to find the one we want.

Now, if you found that annoying, that's sort of what your computer thinks of going through all of the cells in an array to find a value. It does it much much faster of course, but it's still this relatively laborious operation.

```
“doe” - “a deer”
“ray” - “drop of golden sun”
“me” - “a name”
```


Another problem is: what if our data looks like this? We could store it in a contiguous array, and just remember that index 0 is the value of “doe”, and index 1 is “ray”, and index 2 is “me”…but how are we as humans supposed to remember that? Code written that way would be extremely prone to bugs and human error. And it’s important to remember that an extremely profound concern of ours in writing code isn’t just writing code that a machine can execute efficiently - it’s even more-so writing code that can be easily understood and maintained over time by humans.



With that in mind, we might come up with some kind of abstract data type that behaves similar to the way that a dictionary works in real life.

The Dictionary ADT

- AKA *Associative Array*, *Map*, or *Symbol Table*
- Stores *key-value* pairs (e.g. "location" : "NY")
- **Unique keys**
- **Modify & lookup**
  - Set value for key
  - Get value for key
- **Dynamic alteration**
  - Add new pairs
  - Delete existing pairs

(click) A dictionary is an abstract data type - it also goes by the name associative array, map or symbol table - but I like dictionary because that helps me visualize what it actually does - (click) it stores a list of key-value pairs, just like a dictionary holds word-definition pairs.
(click) Dictionaries have a couple of properties that make them interesting - each key in the dictionary is going to be unique. You could store multiple definitions for the same word, but you wouldn't store the same word twice. (click) You can also modify and lookup the values stored at those keys. And if we were to stop there, we would have something called a "struct" - a collection of key value pairs that never add *new* key-value pairs. You can imagine this in JavaScript as being an object that you're not allowed to add new key-value pairs to. (click) However, the *dictionary* ADT also supports dynamic alteration - adding and removing keys. This makes dictionaries trickier than structs, but it also makes them more useful for certain tasks.

How to implement this ADT?

So now that we know we want this behavior, let’s think about how we might implement this with a data structure.

The classic data structure: a Hash Table

High-concept: an array to hold values, and a *hash function* that transforms a string key into a numerical index

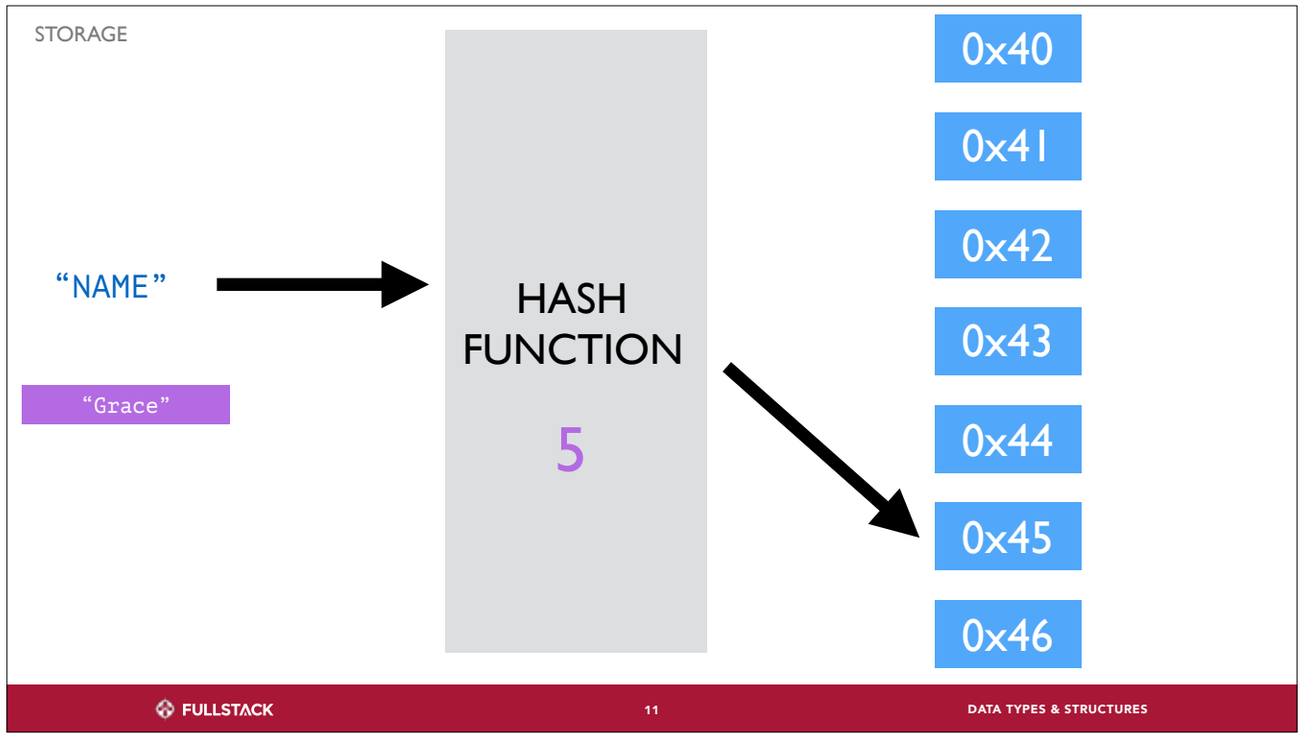
For a hash table, we need two ingredients. First we need a contiguous array to hold values. The second ingredient is a hash function, which transforms some string key into a numerical index in that array.

A simple hash function

```
function hash (keyString) {
  let hashed = 0;
  for (let i = 0; i < keyString.length; i++) {
    hashed += keyString.charCodeAt(i);
  }
  return hashed % 7; // number of spaces in array
}
```


Let's focus on that first ingredient. Here's a very simple hash function - it takes some string as input, adds up the sum of the character codes of each character in that string, and returns the REMAINDER of dividing that number by 7. Though this is very simple, it does the two things that hashing functions must do - first, it takes a string of some arbitrary length, and transforms it into a smaller value within a specified range. No matter how long a string we put into this hash function, we will always get a number between 0 and 6, because we are always returning the remainder of division by the number 7. Second, it always returns the same output for the same input - it's totally deterministic. So no matter how many times we invoke this function for a given input, we will always get the same output.

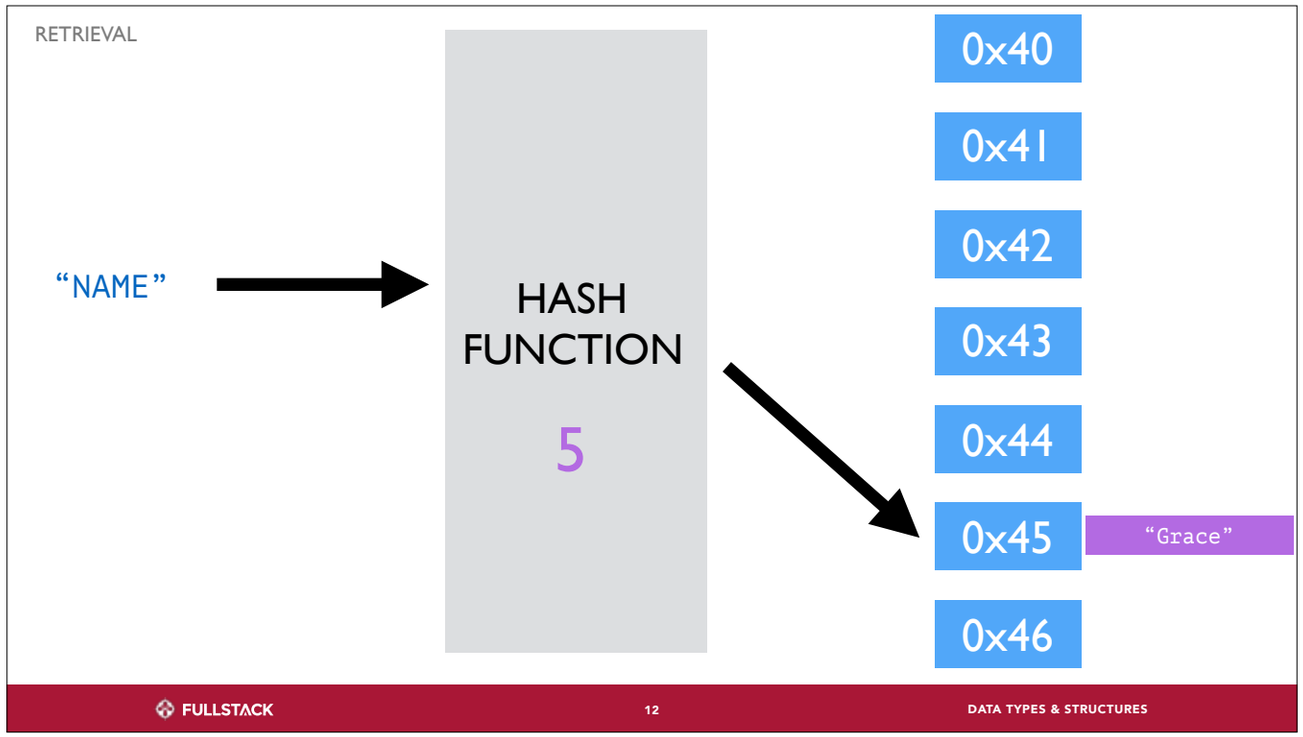
Note that good hashing functions are typically more complex than this. They need to have excellent distribution and other slightly more esoteric mathematical properties. We could spend an entire semester-long class talking about hashing functions, but let's not. What's important is that it does those two things we care about.



The second ingredient for a hash table is a contiguous array. So let’s say our process asks the operating system for some memory, and it gives us 7 contiguous memory cells, starting at address 0x40.

Now, the data we want to store is information about a person’s contact information. For example, a person might have a name and a phone number. Remember, we could say, okay, I’ve got this array, and I’m going to store the name in the 0th index and the phone number in the 1st index…but this is difficult to remember. It’s not very ergonomic from the developer’s perspective.

Instead, we’ll associated each value with a key - which will be a string representing the name of the color. Because this hash function always returns a number between 0 - 6 (which correspond to the indices of the array), we can put our key into the hash function, and use its output (the number between 0 - 6) to determine which memory cell, or index, should hold the value. ‘Name’, for example, might give us ‘5’. So we’ll store the color code at memory address 0x40 + 5.



Then, if we want to retrieve that value later, we don’t have to search cell-by-cell for it. We just need to put the key into the hashing function again, and it will tell us exactly where to look.

```
contact.name = 'Grace'
contact.phone = 8675309
```


Let’s look at this contact info example in more depth. Here’s our goal: we want to store the value “Grace” with the key “name”, and the number value 8675309 with the key “phone”

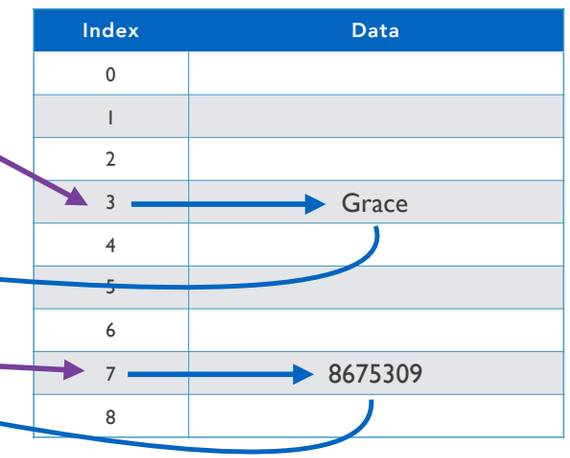
Example with a 9-bucket array

1. We want to store the value 'Grace' for the key 'name'.
2. Hashing the string 'name' yields the numerical index 3.
3. Store 'Grace' at index 3.
4. Now store the value 8675309 for key 'phone'
5. 'phone' hashes to 7
6. Store 8675309 at index 7

Index	Data
0	
1	
2	
3	Grace
4	
5	
6	
7	8675309
8	

Fetching/changing values

1. Q: what is the value for the key 'name'?
2. Hashing the string 'name' yields the numerical index 3.
3. Find the value at index 3.
4. Result: 'Grace'
5. Similar steps for checking the key 'phone', gets the value 8675309.



questions?

Anyone see a problem?

Pause the video here and take a moment to think - is there a problem with this system?

Collisions

1. Now we want to store the value grace@gmail.com for the key email.
2. Hashing email yields the numerical index 7.
3. But we already have a value there (for phone)!
4. Because there are many more possible keys than buckets, collisions are inevitable.

Index	Data
0	
1	
2	
3	Grace
4	
5	
6	
7	8675309
8	

The problem is, what if two keys hash to the same index? This is called a “collision”. (go through slide)

Now, to be fair, we are using an exaggeratedly small number of “buckets" in this example. Good hash tables have a lot more buckets (reducing odds of collision), and for rather esoteric reasons related to the hash function chosen, a prime number of buckets typically results in better distribution of values (again, fewer collisions). But even with a decent number of buckets, it is still going to be very limited compared to the gargantuan number of possible key strings. Note that if all permitted keys are known beforehand and enough buckets are provided, it is possible to have a "perfect" hash function which maps each key to a unique bucket. But that is not the typical case.

How to resolve collisions?

Take a moment and pause the video again - how could we resolve a collision? There’s more than one answer.

Two main strategies

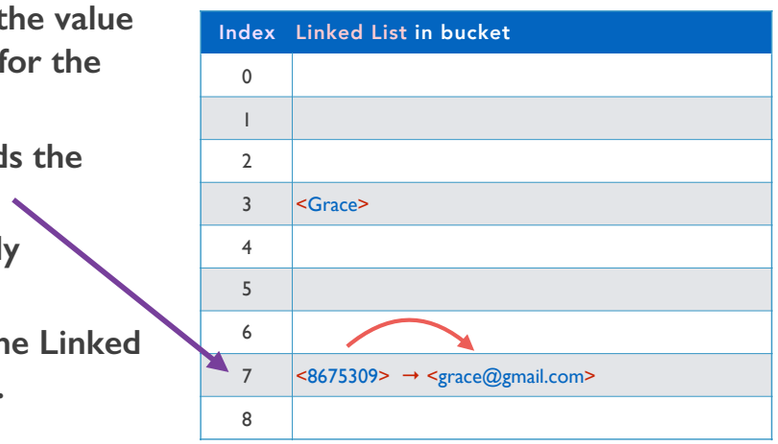
- Open addressing: if a bucket is full, find the next empty bucket. Place the value in that spot instead of the original.

- Separate chaining: every bucket stores a secondary data structure, like a linked list. Collisions create new entries in that data structure.

There are two main strategies that we could take.
The workshop will take the second approach, known as "separate chaining."

Separate chaining: adding a value

1. We want to store the value grace@gmail.com for the key email.
2. Hashing email yields the numerical index 7.
3. That bucket already contains 8675309.
4. Add the value to the Linked List as a new node.



Here's how "separate chaining" works. Instead of just storing a value in each bucket, we actually store a Linked List, which will hold the values.

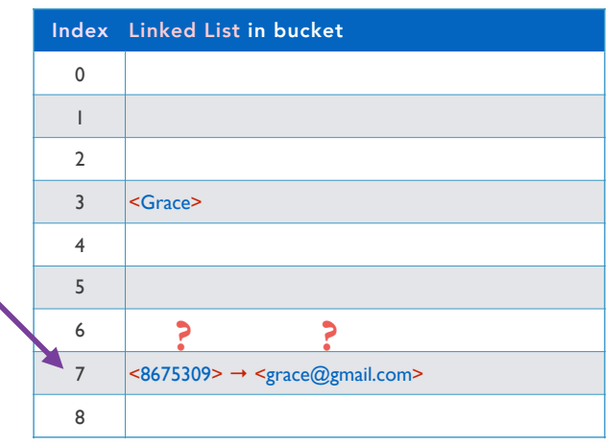
(go through slide)

We still have a problem...

We still have a problem though - pause again and see if you can spot it.

Separate chaining: retrieving a value

1. What is the value for the key email?
2. Hashing email yields the numerical index 7.
3. In bucket 7 there is a linked list.
4. There are *two* nodes in that list; how do we know which value is for the key email?

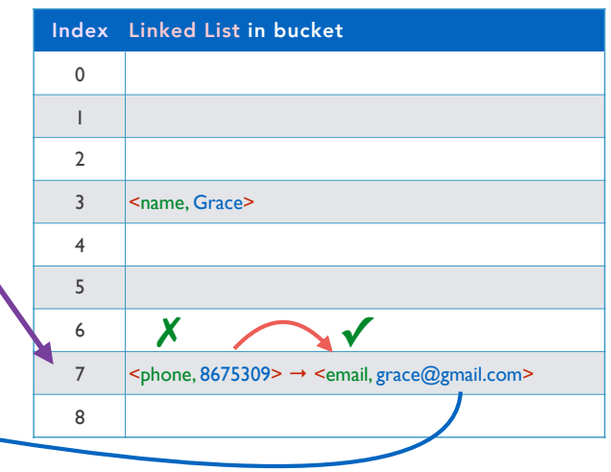


Index	Linked List in bucket
0	
1	
2	
3	<Grace>
4	
5	
6	? ?
7	<8675309> → <grace@gmail.com>
8	

The problem appears when we try to retrieve a value - how do we know which value we want for a given key? For example…(run through slide)

Separate chaining: store value *and* key

1. Hashing email yields the numerical index 7.
2. In bucket 7 there is a linked list.
3. The head node has a key of phone, so we need to keep looking.
4. The next node has a key email, so we return the value: grace@gmail.com



One possible solution is that we could store both the key and the value. Let’s see what this looks like.

So... how is a hash table better than a linked list?

So, if we’re still potentially traversing a linked list to get our data…how is a hash table any better?

Performance

- Assume many buckets and a good hashing function
- *Usually*: assign or check pair takes just 1 step (hash invocation)
- *Sometimes*: collisions occur
  - Traverse a few nodes of a linked list; but just a few (still pretty fast)
  - Way better than having to traverse all the data in the entire structure!

The answer is that it is still in almost every case going to be way more performant.

Our example used a small number of buckets and had a very basic hash function, but hash tables in real life typically have many buckets, and use a good hash function that will be unlikely to produce collisions

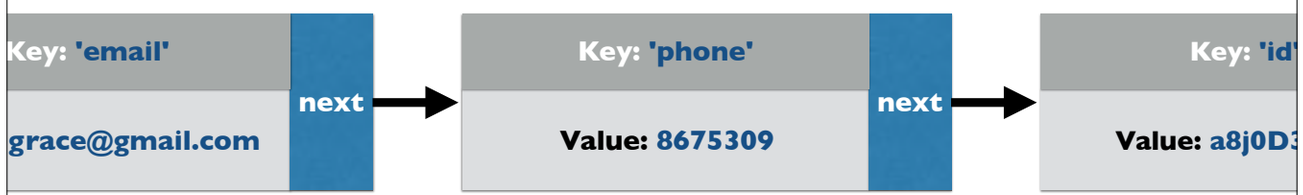
Usually, assigning or checking a key-value pair just takes one step - invoking the hash function.

While sometimes collisions will occur, you'll be traversing a few linked list nodes, rather than all of the data you've stored.

List nodes with key-value pairs — how?

Let’s return our attention to this concept of having a list nodes that store key-value pairs. You might have taken this for granted, but remember that the list nodes you implemented in class could only contain “value” and “next”. Let’s take a moment and pretend that we’re working at a much lower level, where we might have to take this decision pretty seriously. Let’s consider a few solutions.

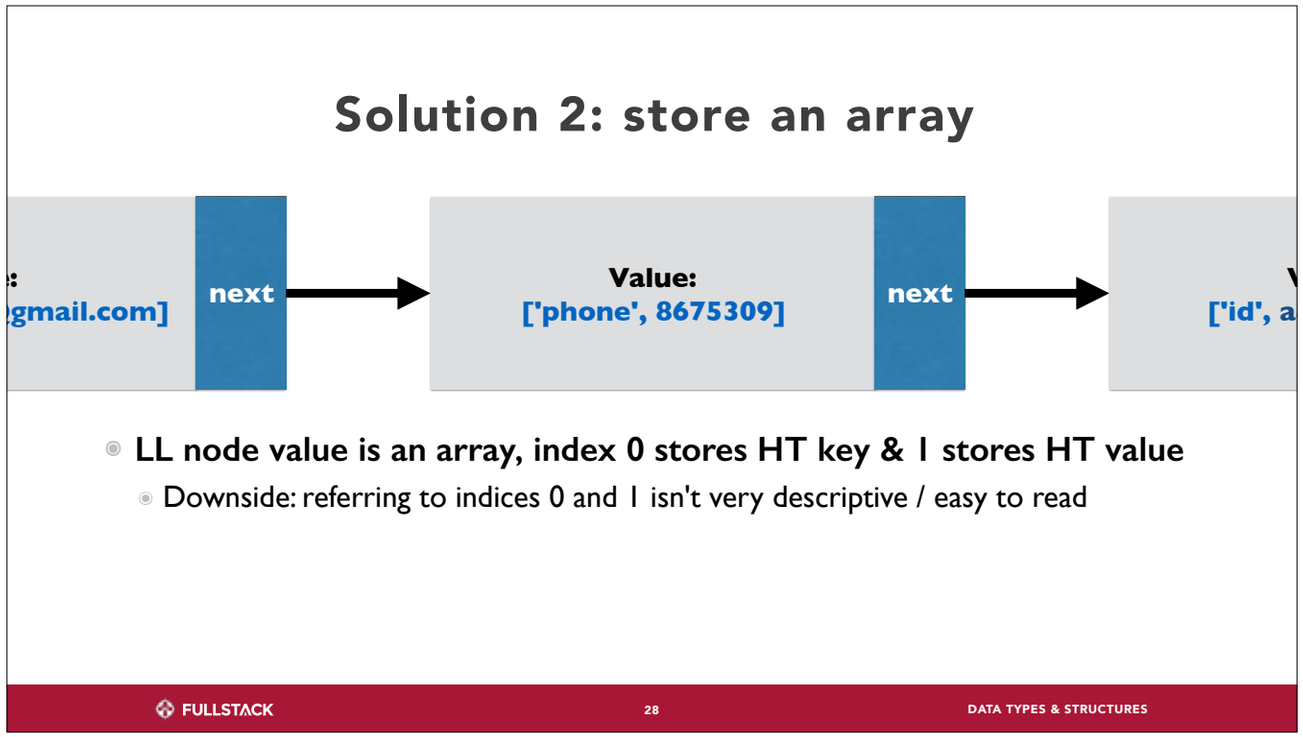
Solution 1: *Association List*



- Nodes have key as well as value & next
  - Advantage: best for purpose, most straightforward.
  - Downside: need a custom LL implementation

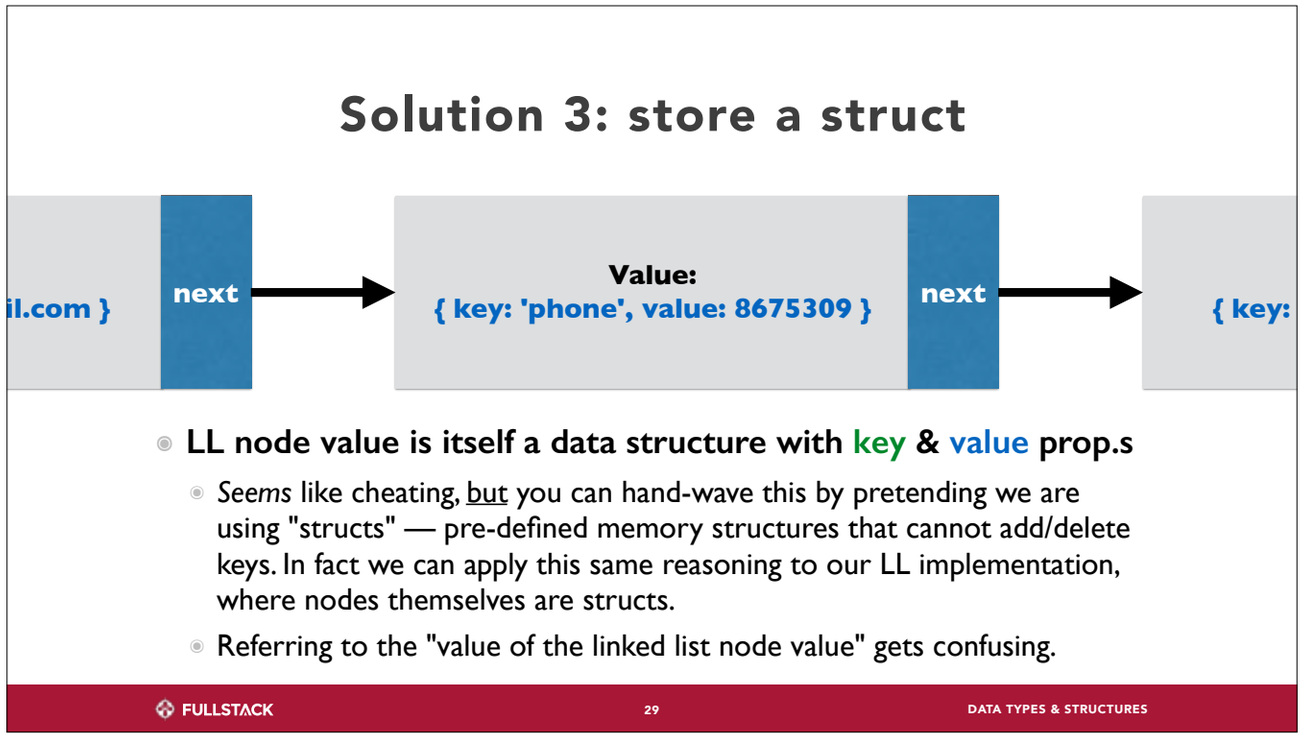
One solution is we could use a variant of a linked list, called an “association list”.

Association lists are linked lists which don't store just a value, but rather key-value pairs. That makes them ideal for use in hash table buckets. You’ll actually end up building one of these during the exercise, but consider that the downside here is that you need to build a brand new data structure - you can’t just re-use the linked list you already built.



What if we wanted to use our existing linked list? We would need to figure out how to store multiple elements in the `value` field of the list node.

Well, one approach we could take would be to wrap them in a container, like a contiguous array. Downside, storing the hash table key in node.value[0] and hash table value in node.value[1] is quite opaque - it has that same ergonomics problem.



Slightly better would be if we could use a *struct*, which is another way to store properties with string keys.

This might seem like cheating, but remember that with a struct, you have to pre-define all the keys that it can ever have - it can never have new keys added or removed. JS doesn't have structs, but if we were implementing this in javascript, we could pretend a simple object is a struct so long as you never add new keys (properties) to it.

So now a lot of what we’re imagining when we’re modeling this data structure is that we’re dealing with a lower level language like C. How does this pertain to JavaScript, a much higher level language?

Sound familiar?

- **JavaScript Objects**
- **JS *Engines* (like V8) implement most Objects as structs**
  - V8 defines a new struct every time you add a property
  - If this would be madness (ex: you are storing a phone book), it switches to using a hash table
- **In the end, the Object specified in EcmaScript is a data *type;* we know what behavior it should exhibit, but not necessarily how it is implemented at runtime. That's up to the engine.**

Hash tables of course sound just like JavaScript objects.

However, you might be very surprised to learn that most JavaScript engines (like v8) actually don’t use hash tables for most objects - they use something like an immutable list of structs. This might sound a little bit crazy, but because the v8 engine knows how to re-use these structs, it actually results in less memory usage most of the time. We’ll revisit this concept later in the course when we discuss functional programming and immutable data structures.

However, if the engine sees that your object is so big that using linked structs would actually be wasteful, then it will optimize by switching to use a hash table.

https://developers.google.com/v8/design#fast-property-access
http://jayconrod.com/posts/52/a-tour-of-v8-object-representation