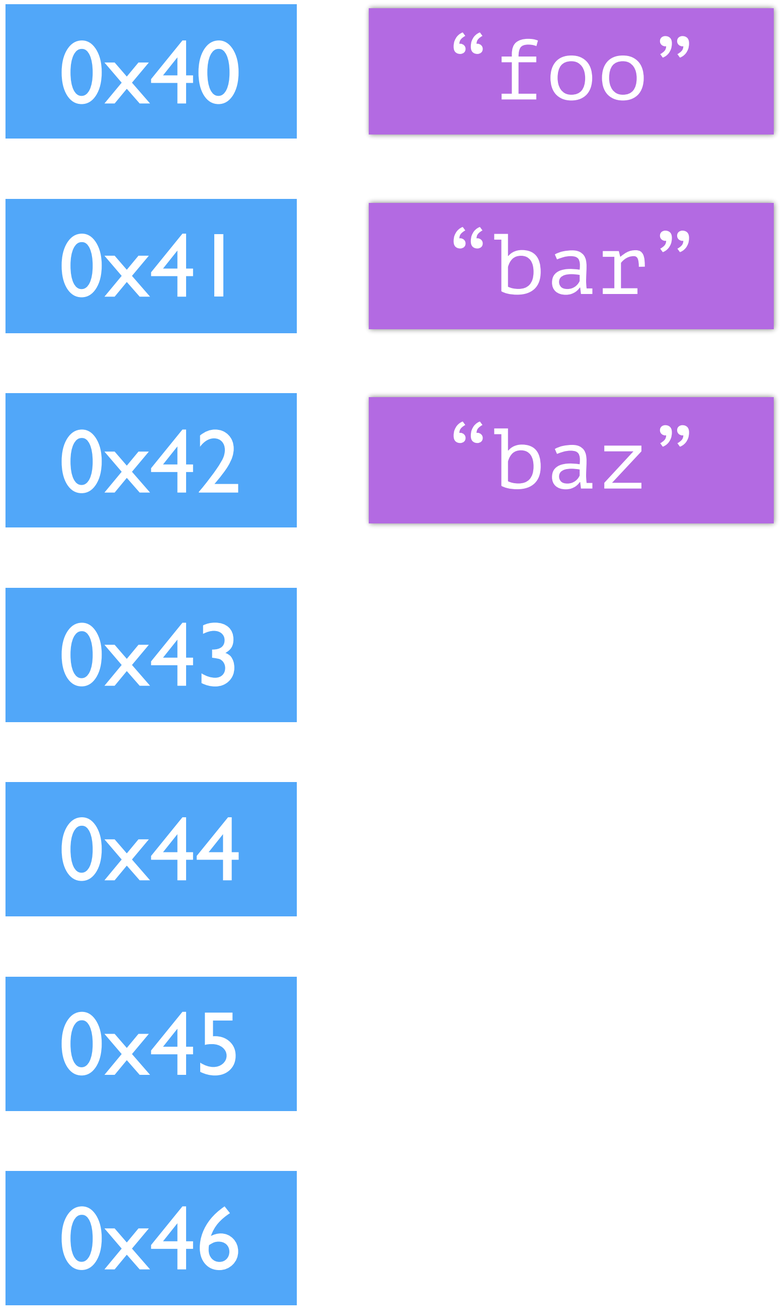
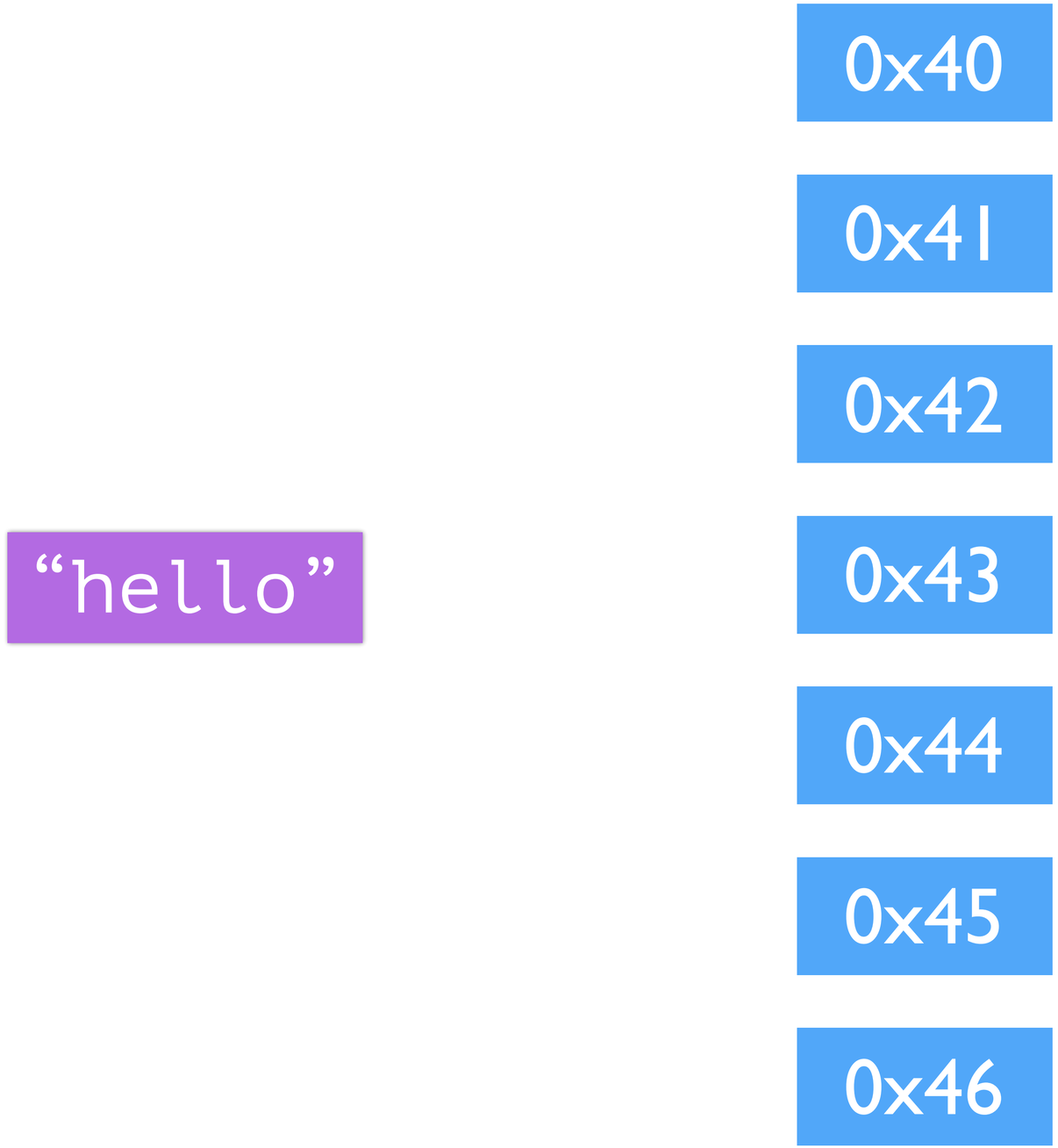
HASH TABLES

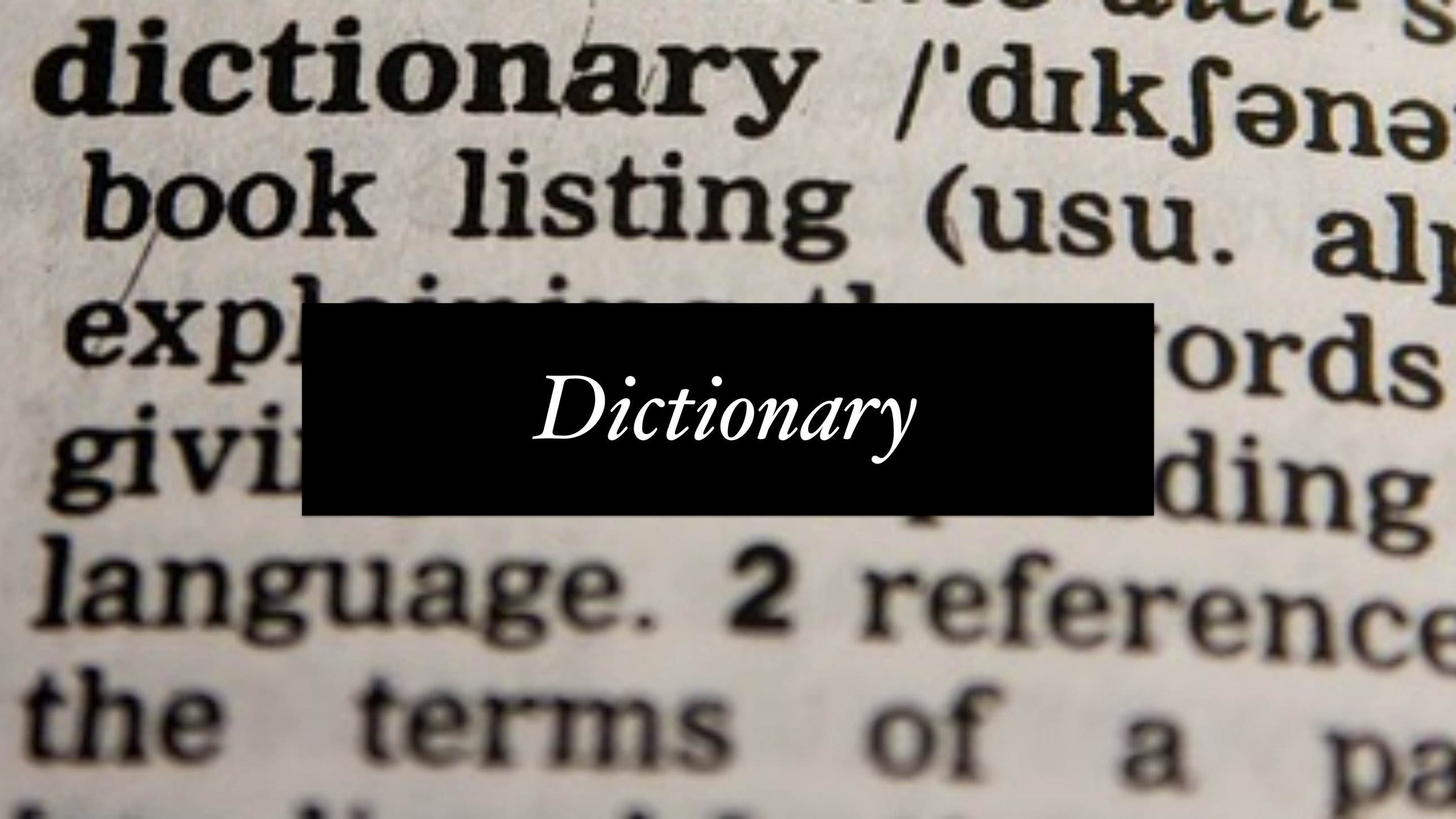
CONTIGUOUS ARRAYS



PROBLEMS WITH CONTIGUOUS ARRAYS


"hello"
0x40
0x41
0x42
0x43
0x44
0x45
0x46

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


dictionary /'dɪkʃənə
book listing (usu. alp
ords
ding
language. 2 reference
the terms of a pa
Dictionary

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

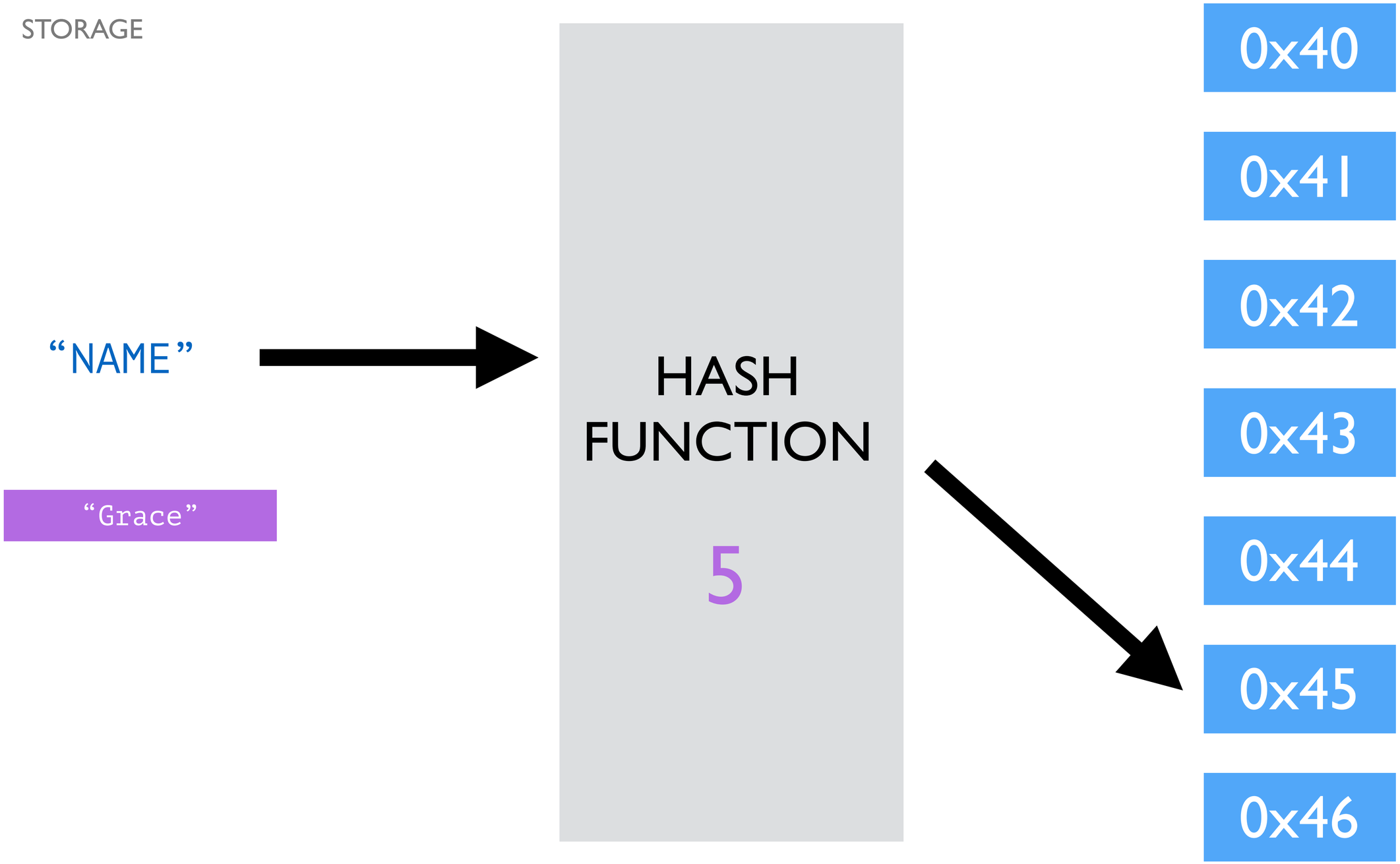
How to implement this ADT?

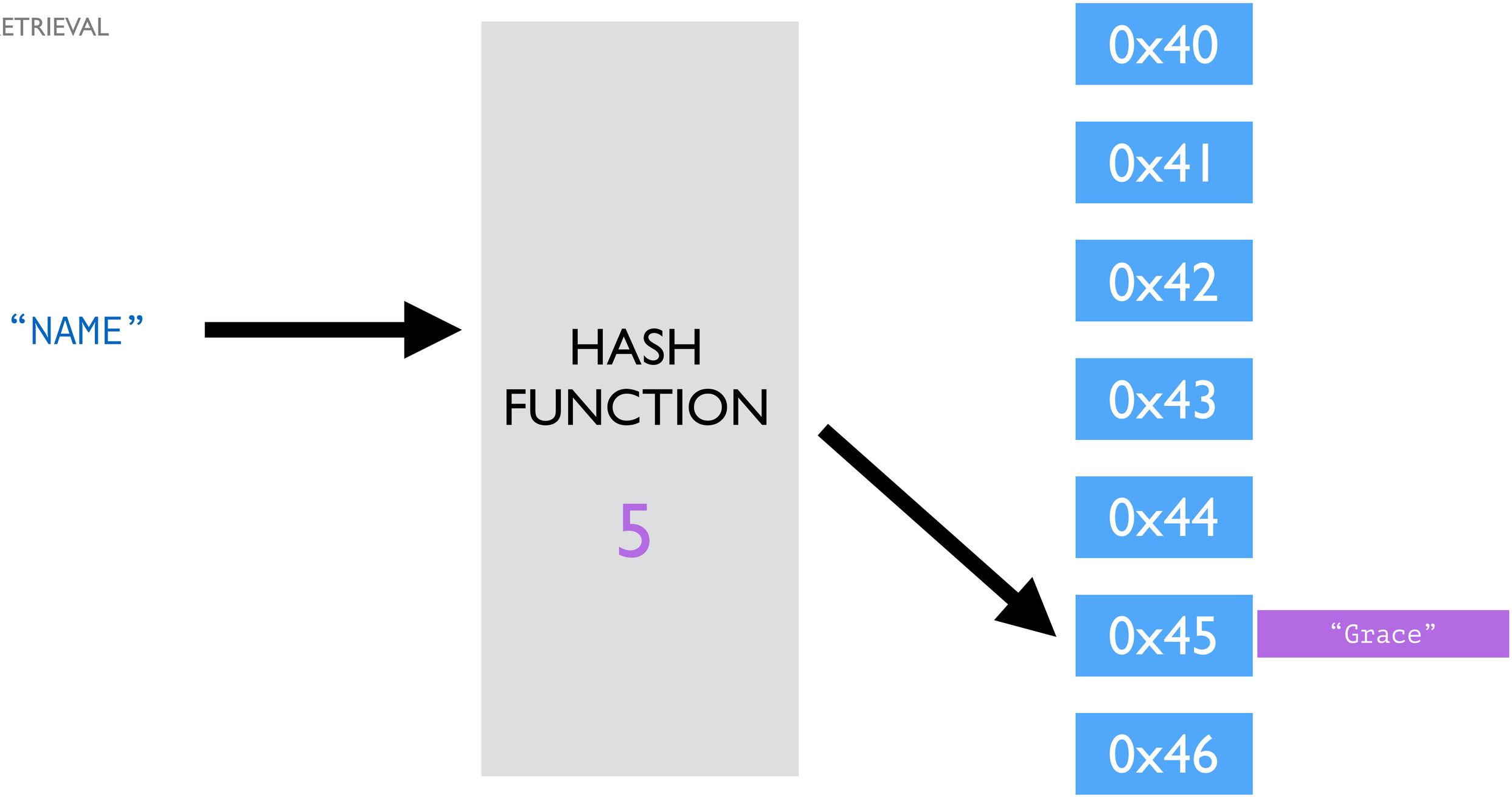
The classic data structure: a Hash Table

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

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
}
```

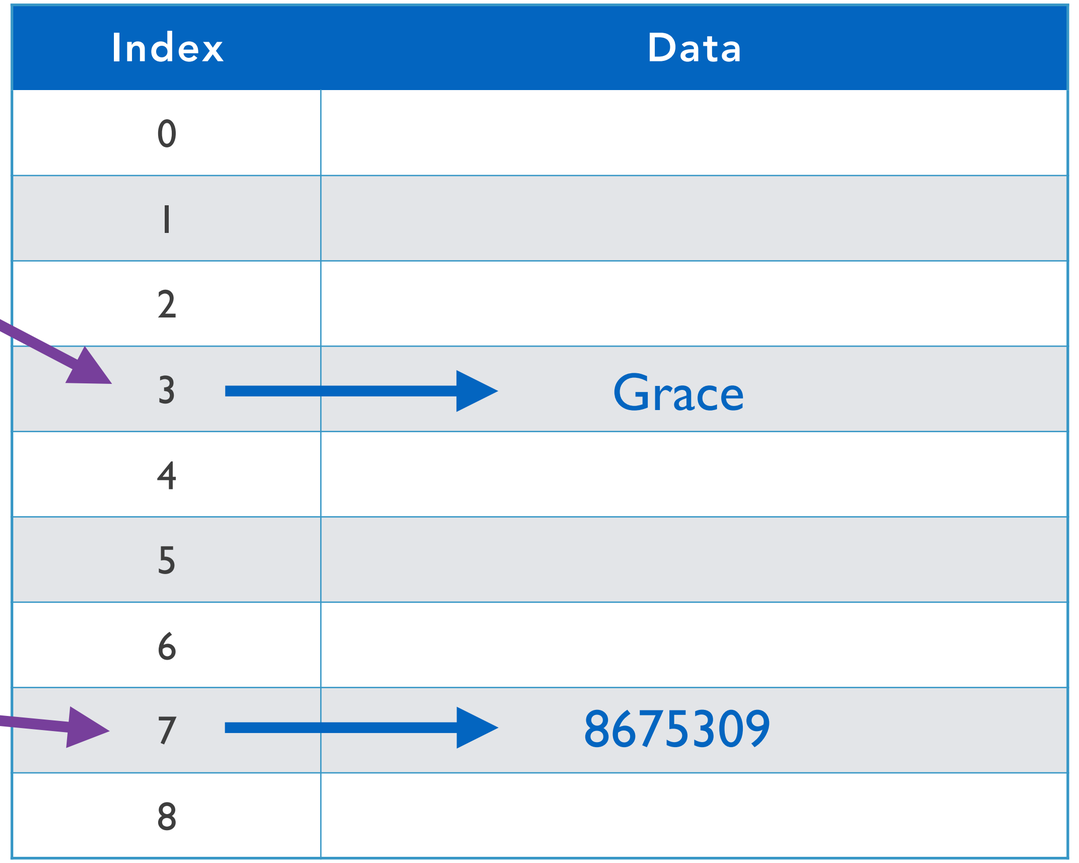

"NAME"
"Grace"
HASH
FUNCTION
5
0x40
0x41
0x42
0x43
0x44
0x45
0x46


"NAME"
HASH FUNCTION
5
0x40
0x41
0x42
0x43
0x44
0x45
"Grace"
0x46

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

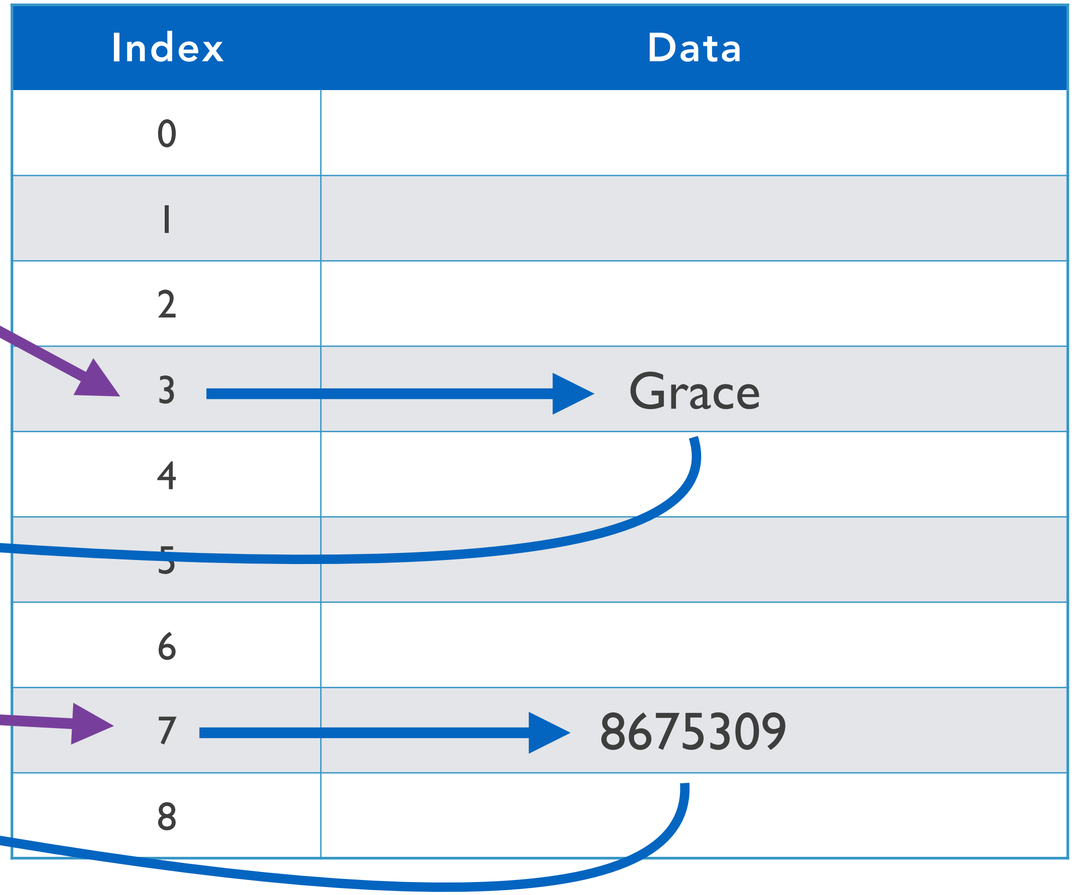
Example with a 9-bucket array

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



Fetching/changing values

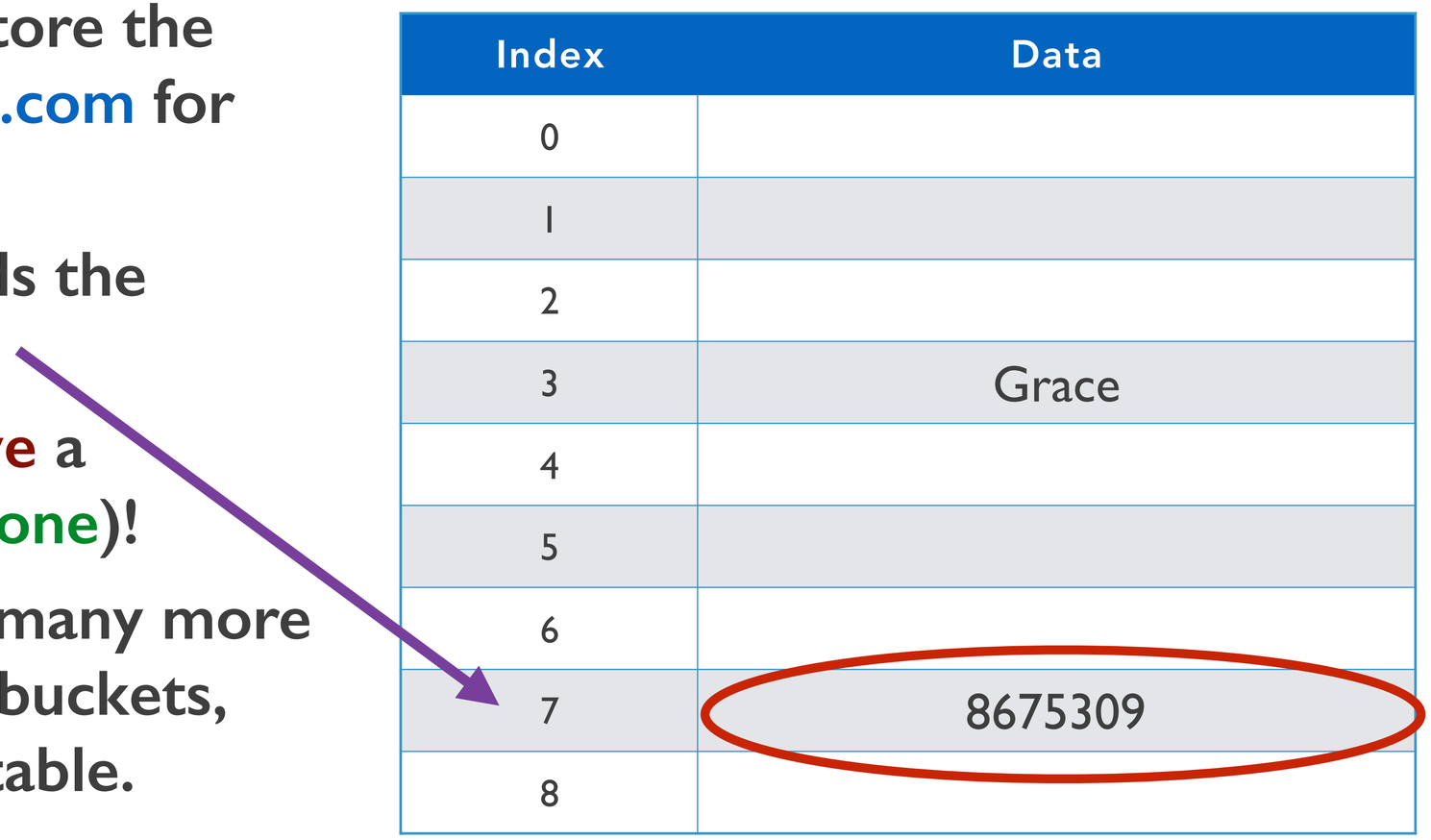
1. Q: what is the value for the key 'name'?
2. Hashing the string 'name' yields the numerical index 3.
3. Find the value at index 3.
4. Result: 'Grace'
5. Similar steps for checking the key 'phone', gets the value 8675309.



Anyone see a problem?

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	

How to resolve collisions?

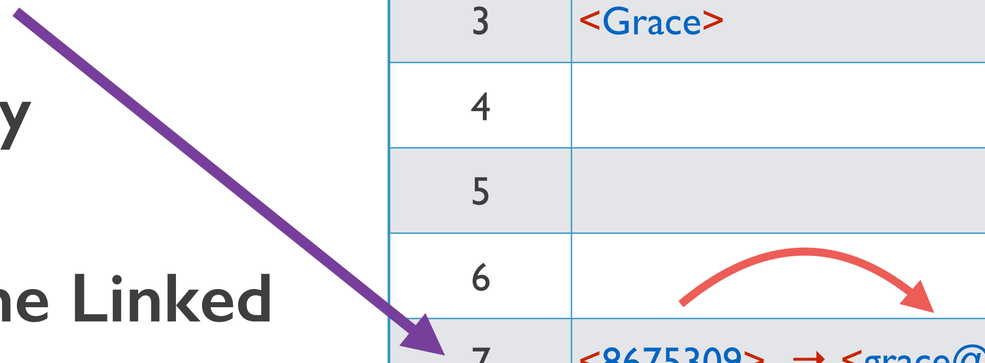
Two main strategies

- **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.

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.



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

We still have a problem...

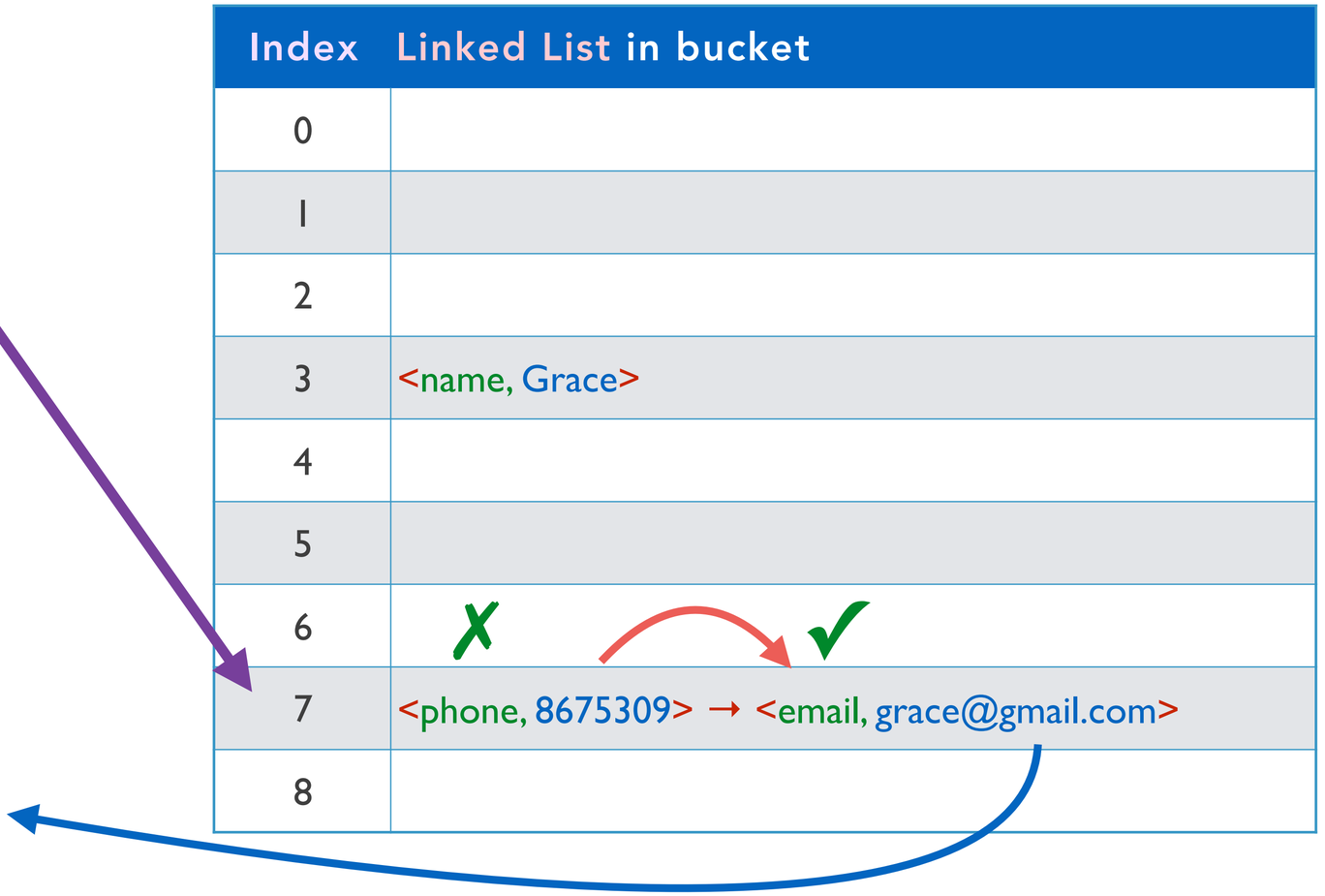
Separate chaining: retrieving a value

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	

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



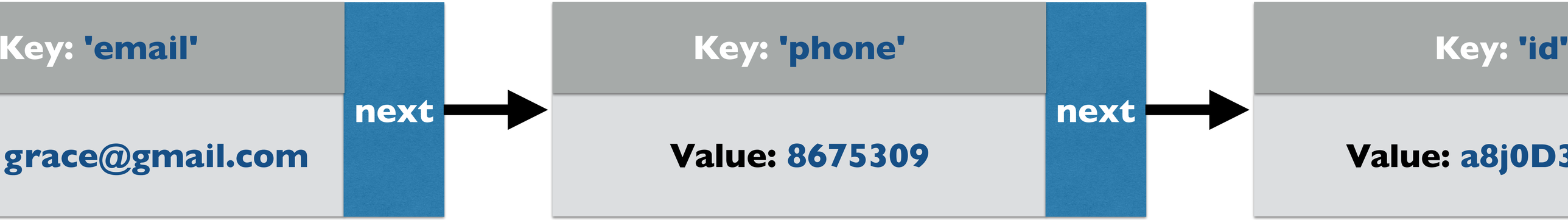
So... how is a hash table better than a linked list?

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!

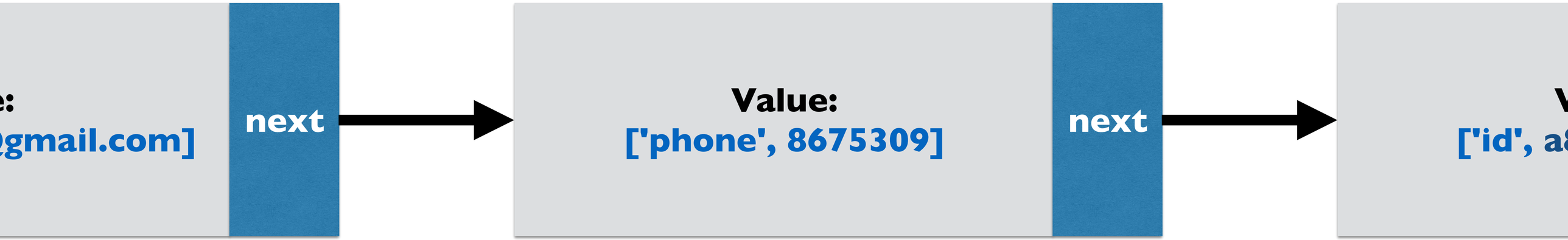
List nodes with key-value pairs — how?

Solution 1: *Association List*



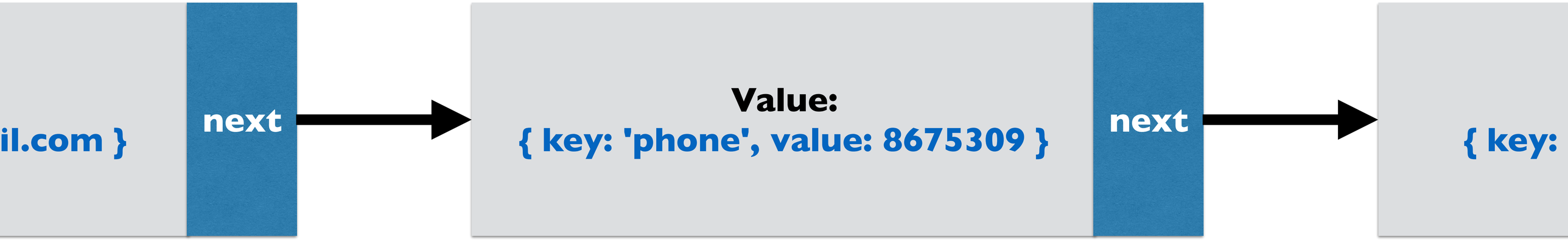
- Nodes have key as well as value & next
  - Advantage: best for purpose, most straightforward.
  - Downside: need a custom LL implementation

Solution 2: store an array



- LL node value is an array, index 0 stores HT key & 1 stores HT value
  - Downside: referring to indices 0 and 1 isn't very descriptive / easy to read

Solution 3: store a struct



- **LL node value is itself a data structure with key & value prop.s**
  - *Seems* like cheating, but you can hand-wave this by pretending we are using "structs" — pre-defined memory structures that cannot add/delete keys. In fact we can apply this same reasoning to our LL implementation, where nodes themselves are structs.
  - Referring to the "value of the linked list node value" gets confusing.

WHAT ABOUT JS?

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.**