HASH
TABLES

OBJECTIVES

- Explain what a hash table is
- Define what a hashing algorithm
- Discuss what makes a good hashing algorithm
- Understand how collisions occur in a hash table
- Handle collisions using separate chaining or linear probing

WHAT IS A HASH TABLE?

Hash tables are used to store *key-value* pairs.

They are like arrays, but the keys are not ordered.

Unlike arrays, hash tables are *fast* for all of the following operations: finding values, adding new values, and removing values!

WHY SHOULD I CARE?

Nearly every programming language has some sort of hash table data structure

Because of their speed, hash tables are very commonly used!

HASH TABLES IN THE WILD

Python has Dictionaries

JS has Objects and Maps*

Java, Go, & Scala have Maps

Ruby has...Hashes

* Objects have some restrictions, but are basically hash tables

LET'S PRETEND...

~~Python has Dictionaries~~

~~JS has Objects and Maps*~~

~~Java, Go, & Scala have Maps~~

~~Ruby has...Hashes~~

Existing implementations mysteriously disappear

How would we implement our own version???

HASH TABLES

Introductory Example

Imagine we want to store some colors

We could just use an array/list:

```
[ "#ff69b4","#ff4500","#00ffff" ]
```

Not super readable! What do these colors correspond to?

HASH TABLES

Introductory Example

It would be nice if instead of using indices to access the colors, we could use more human-readable keys.

pink → `#ff69b4`

orangered → `#ff4500`

cyan → `#00ffff`

`colors["cyan"]`

is way better than

`colors[2]`

HASH TABLES

Introductory Example

How can we get human-readability *and* computer readability?

Computers don't know how to find an element at index *pink*!

Hash tables to the rescue!

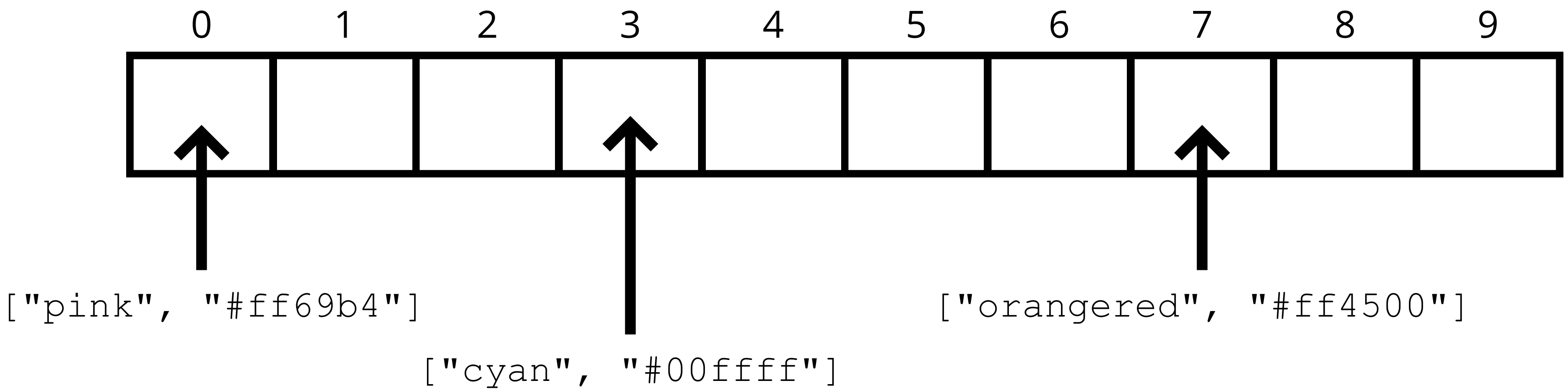
THE HASH PART

To implement a hash table,
we'll be using an array.

In order to look up values by key,
we need a way to convert keys
into valid array indices.

A function that performs this
task is called a *hash function*.

HASHING CONCEPTUALLY



["pink", "#ff69b4"]

["cyan", "#00ffff"]

["orangered", "#ff4500"]

WHAT MAKES A GOOD HASH?

(not a cryptographically secure one)

1. Fast (i.e. constant time)
2. Doesn't cluster outputs at specific indices, but distributes uniformly
3. Deterministic (same input yields same output)

What Makes for a Good Hash?

Fast

Non-Example

```
function slowHash(key) {
  for (var i = 0; i < 10000; i++) {
    console.log("everyday i'm hashing");
  }
  return key[0].charCodeAt(0);
}
```

What Makes for a Good Hash?

Uniformly Distributes Values

Non-Example

```
function sameHashedValue(key) {
  return 0;
}
```

What Makes for a Good Hash?

Deterministic

Non-Example

```
function randomHash(key) {
  return Math.floor(Math.random() * 1000)
}
```

What Makes for a Good Hash?

Simple Hash Example

Here's a hash that works on *strings only*:

```
function hash(key, arrayLen) {
  let total = 0;
  for (let char of key) {
    // map "a" to 1, "b" to 2, "c" to 3, etc.
    let value = char.charCodeAt(0) - 96
    total = (total + value) % arrayLen;
  }
  return total;
}
```

```
hash("pink", 10); // 0
hash("orangered", 10); // 7
hash("cyan", 10); // 3
```

REFINING OUR HASH

Problems with our current hash

1. Only hashes strings (we won't worry about this)
2. Not constant time - linear in key length
3. Could be a little more random

Hashing Revisited

```
function hash(key, arrayLen) {
  let total = 0;
  for (let i = 0; i < key.length; i++) {
    let char = key[i];
    let value = char.charCodeAt(0) - 96
    total = (total + value) % arrayLen;
  }
  return total;
}
```

```
function hash(key, arrayLen) {
  let total = 0;
  let WEIRD_PRIME = 31;
  for (let i = 0; i < Math.min(key.length, 100); i++) {
    let char = key[i];
    let value = char.charCodeAt(0) - 96
    total = (total * WEIRD_PRIME + value) % arrayLen;
  }
  return total;
}
```

Prime numbers? wut.

The prime number in the hash is helpful in spreading out the keys more uniformly.

It's also helpful if the array that you're putting values into has a prime length.

You don't need to know why. (Math is complicated!)
But here are some links if you're curious.

Why do hash functions use prime numbers?

Does making array size a prime number help in hash table implementation?

Dealing with Collisions

Even with a large array and a great hash function, collisions are inevitable.

There are many strategies for dealing with collisions, but we'll focus on two:

1. Separate Chaining
2. Linear Probing

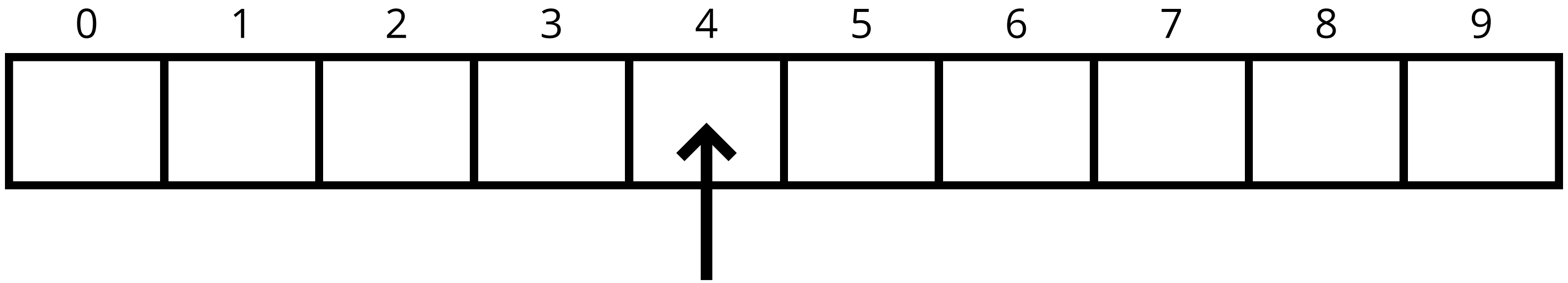
Separate Chaining

With *separate chaining*, at each index in our array we store values using a more sophisticated data structure (e.g. an array or a linked list).

This allows us to store multiple key-value pairs at the same index.

Separate Chaining

Example



```
[ ["darkblue", "#00008b"],
  ["salmon", "#fa8072"] ]
```

darkblue → 4

salmon → 4

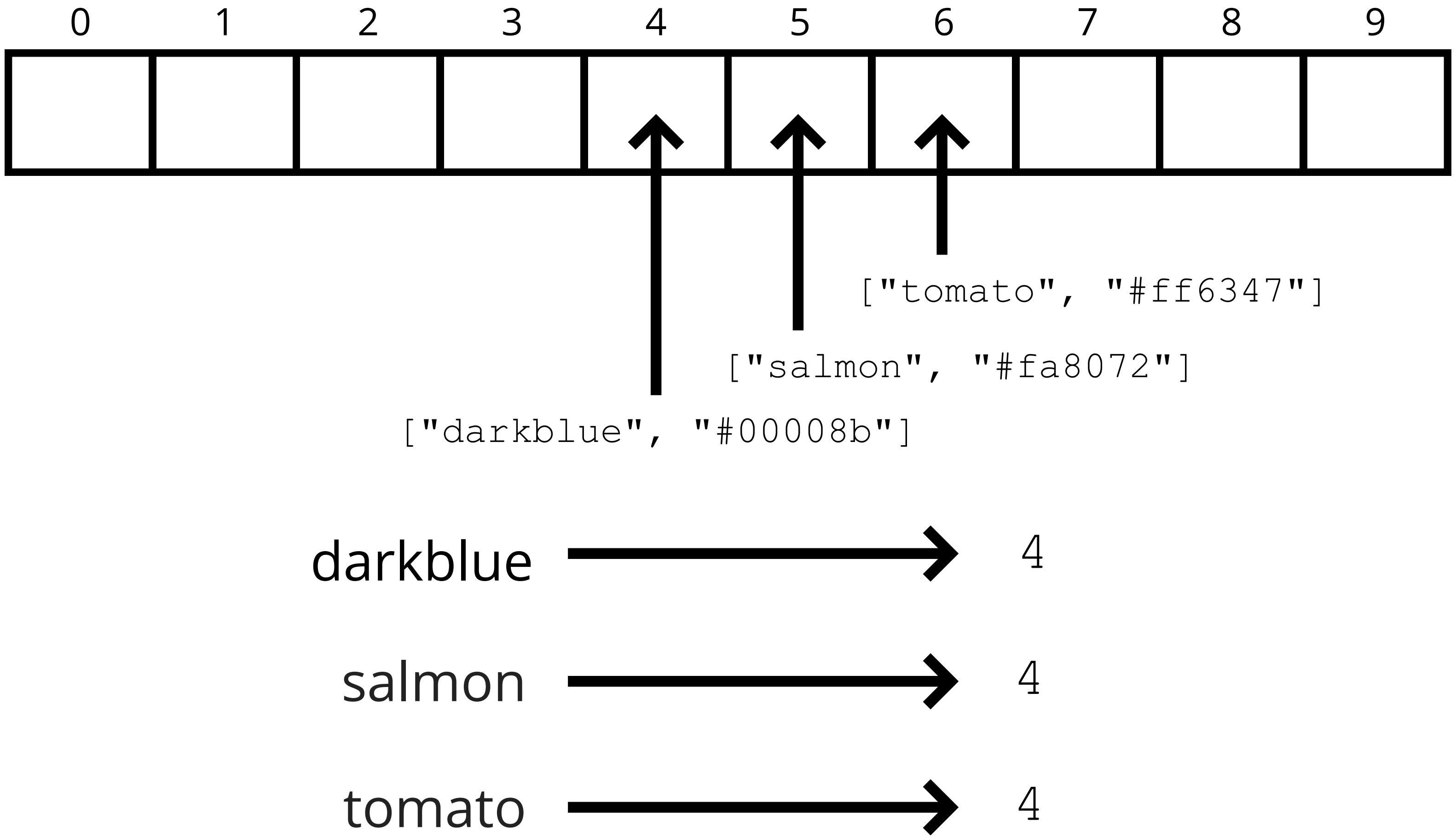
Linear Probing

With *linear probing*, when we find a collision, we search through the array to find the next empty slot.

Unlike with separate chaining, this allows us to store a single key-value at each index.

Linear Probing

Example



0 1 2 3 4 5 6 7 8 9

```
["tomato", "#ff6347"]
["salmon", "#fa8072"]
["darkblue", "#00008b"]
```

darkblue → 4

salmon → 4

tomato → 4

A HashTable Class

```
class HashTable {
  constructor(size=53){
    this.keyMap = new Array(size);
  }

  _hash(key) {
    let total = 0;
    let WEIRD_PRIME = 31;
    for (let i = 0; i < Math.min(key.length, 100); i++) {
      let char = key[i];
      let value = char.charCodeAt(0) - 96
      total = (total * WEIRD_PRIME + value) % this.keyMap.length;
    }
    return total;
  }
}
```

Set / Get

`set`

1. Accepts a key and a value
2. Hashes the key
3. Stores the key-value pair in the hash table array via separate chaining

`get`

1. Accepts a key
2. Hashes the key
3. Retrieves the key-value pair in the hash table
4. If the key isn't found, returns `undefined`

YOUR TURN

Keys / Values

`keys`

1. Loops through the hash table array and returns an array of keys in the table

`values`

1. Loops through the hash table array and returns an array of values in the table

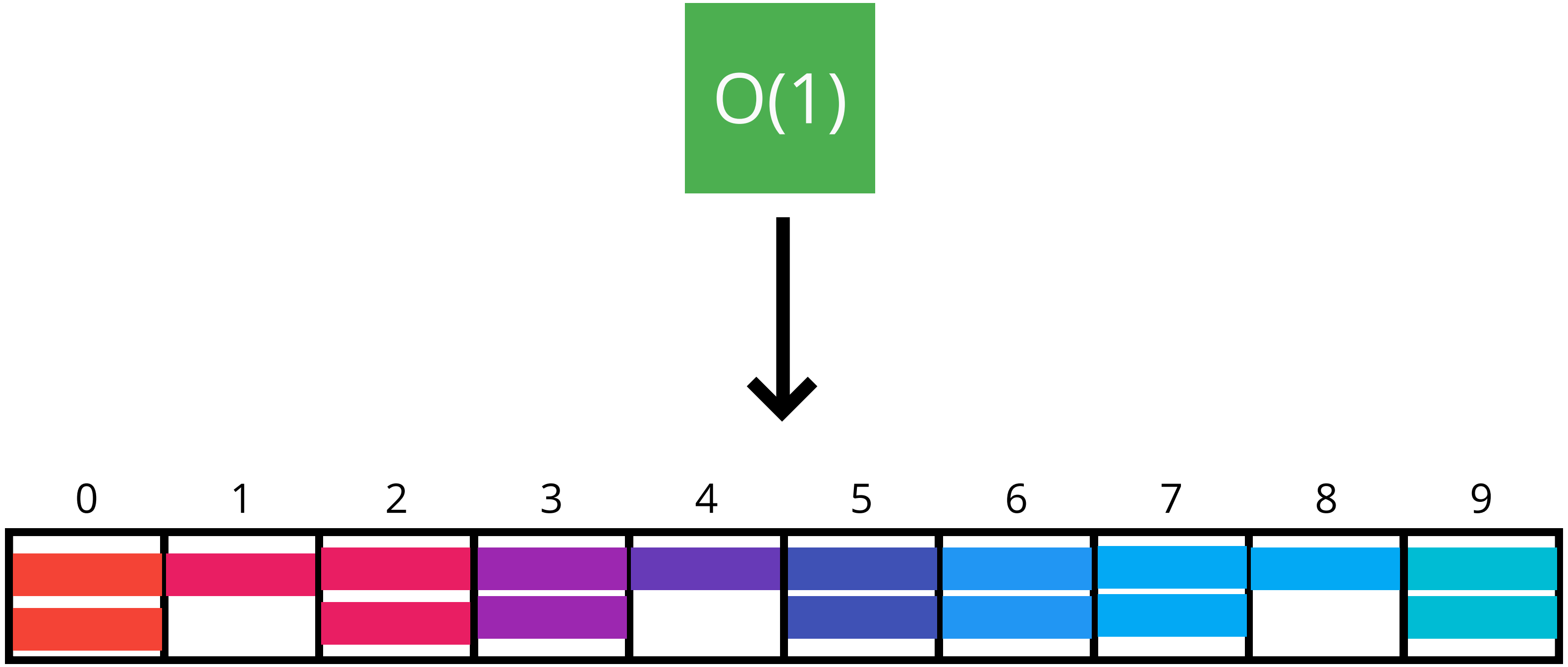
YOUR
TURN

BIG O of HASH TABLES

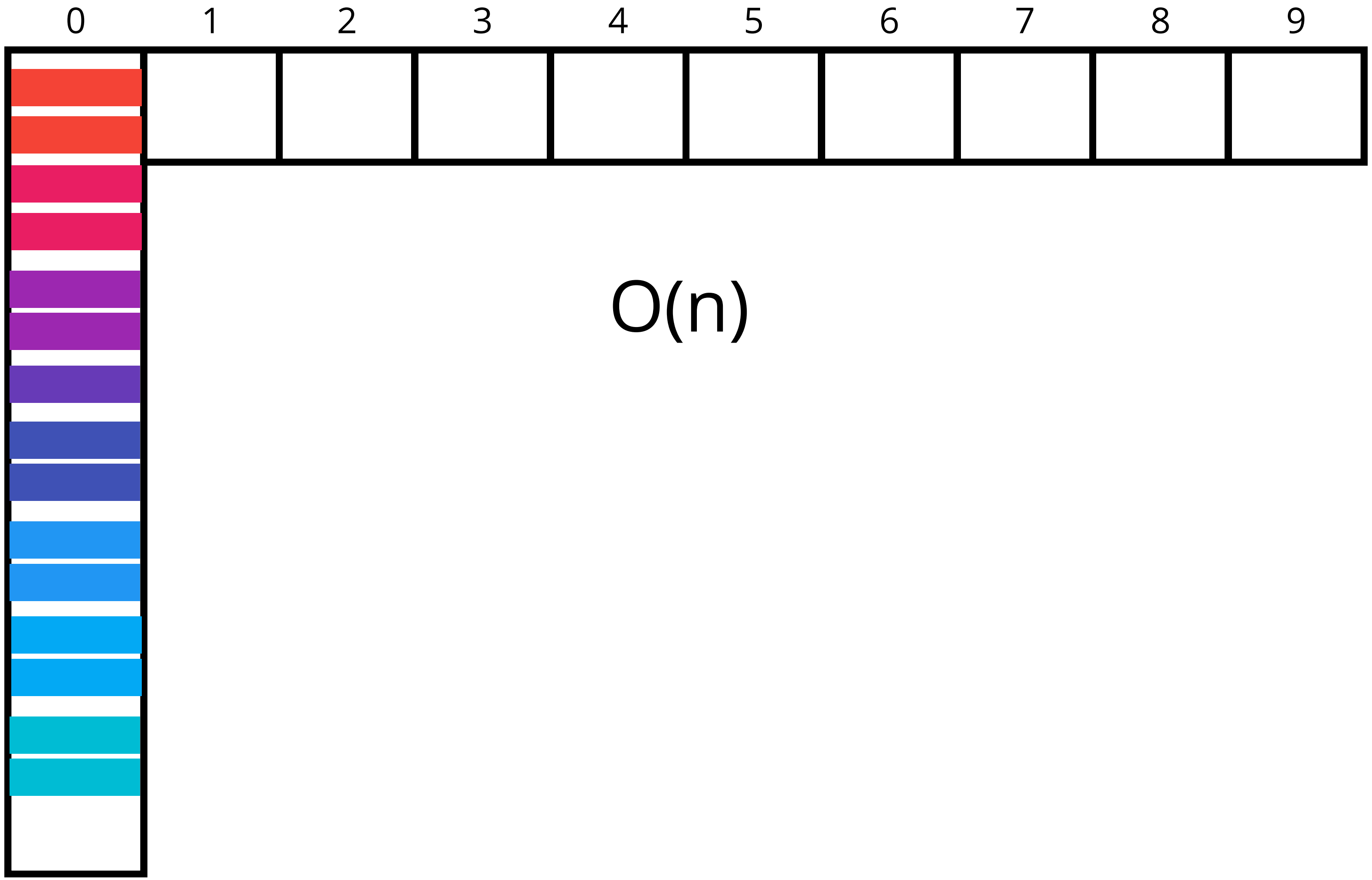
(average case)

- Insert: O(1)
- Deletion: O(1)
- Access: O(1)

A good hash function



With the world's worst hash function...



O(n)

Recap

- Hash tables are collections of key-value pairs
- Hash tables can find values quickly given a key
- Hash tables can add new key-values quickly
- Hash tables store data in a large array, and work by *hashing* the keys
- A good hash should be fast, distribute keys uniformly, and be deterministic
- Separate chaining and linear probing are two strategies used to deal with two keys that hash to the same index
- When in doubt, use a hash table!