Data Structures for Web Devs

A very abridged intro to Strings, Complexity, Arrays, Maps, Sets and Immutable Data Structures

David Forshner

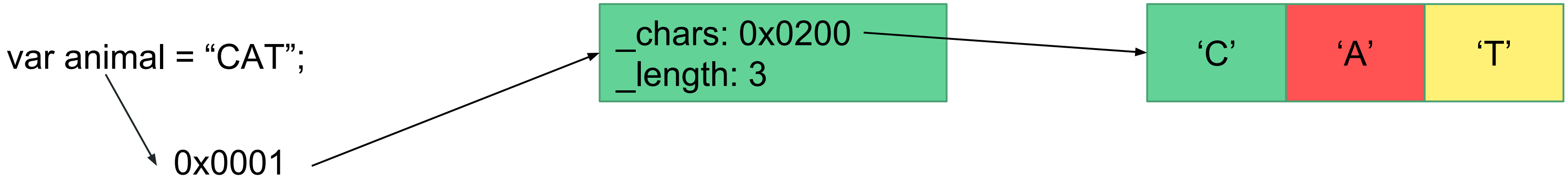
Disclaimer: This is probably all wrong or will be wrong shortly.

Strings

Start off with something simple … a data type.

Strings

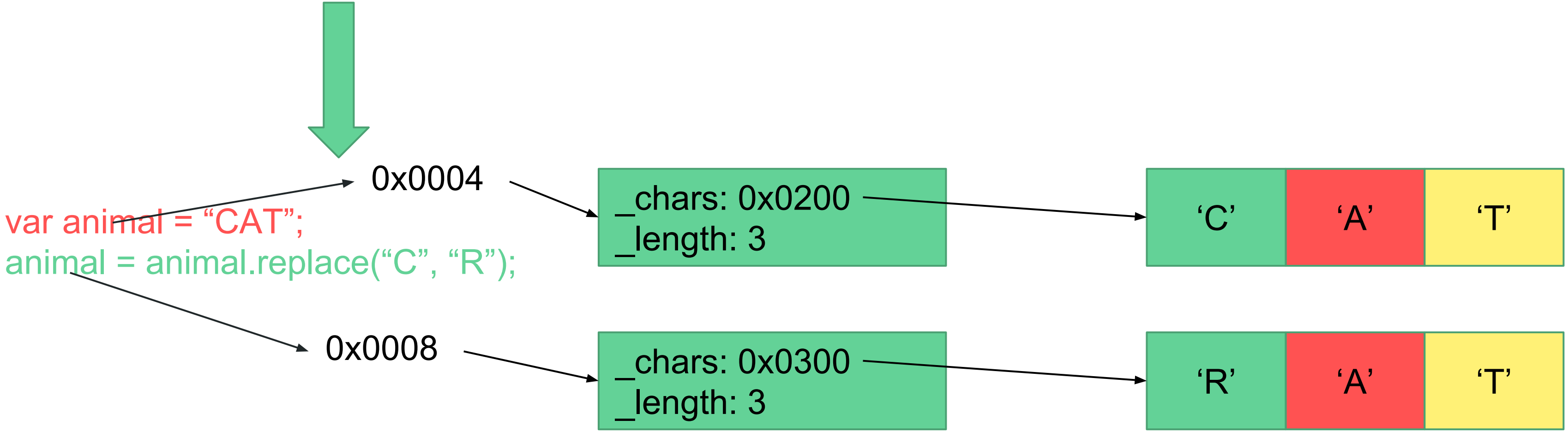
- A wrapper around an internal array of bytes.
- Good at: String … stuff



Strings are immutable

- Once created you cannot change a string’s value.

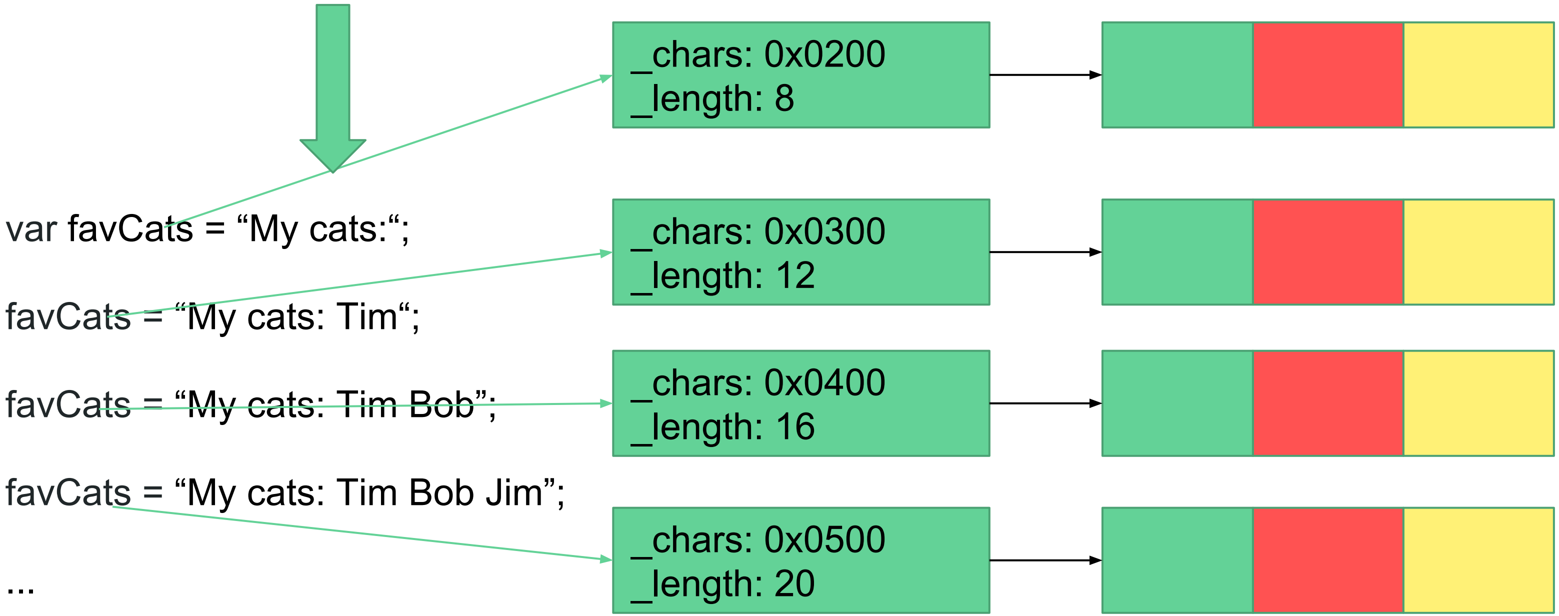
```
var animal = new String(“CAT”).replace(“C”, “R”);
```



```
var animal = “CAT”;
animal = animal.replace(“C”, “R”);
```

String concatenation in loops

```
var cats = [“Tim”, “Bob”, “Jim”, “Kat”, “Kim”, “Sam”];
var favCats = “My cats:“;
cats.forEach(x => favCats + “ “ + x);
```



var favCats = “My cats:“;

favCats = “My cats: Tim“;

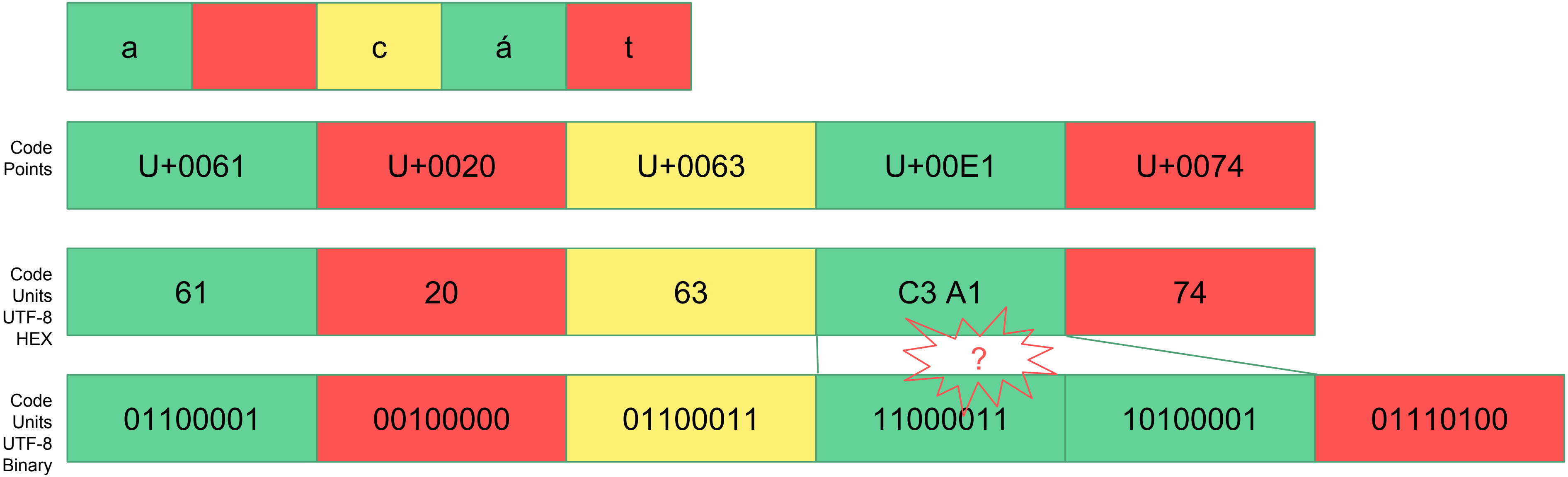
favCats = “My cats: Tim Bob”;

favCats = “My cats: Tim Bob Jim”;

...

Unicode Encoding/Decoding

- Handles encoding/decoding internal byte array to/from Unicode ☺.



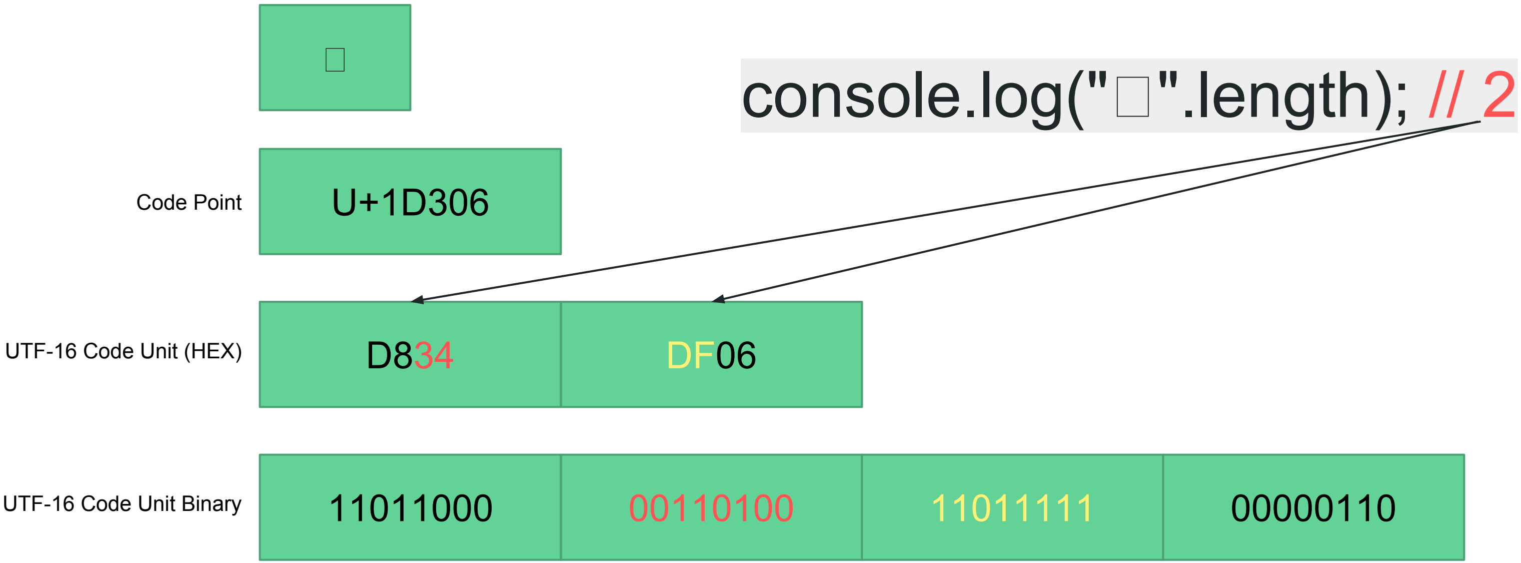
a		c	á	t

Code Points	U+0061	U+0020	U+0063	U+00E1	U+0074
Code Units UTF-8 HEX	61	20	63	C3 A1	74

Code Units UTF-8 Binary	01100001	00100000	01100011	11000011	10100001	01110100

ECMAScript = UTF-16 Encoding

- .length() returns # of UTF-16 code units not the # of characters (code points)



```
console.log("□".length); // 2
```

	□			
Code Point	U+1D306			
UTF-16 Code Unit (HEX)	D834	DF06		
UTF-16 Code Unit Binary	11011000	00110100	11011111	00000110

Jargon

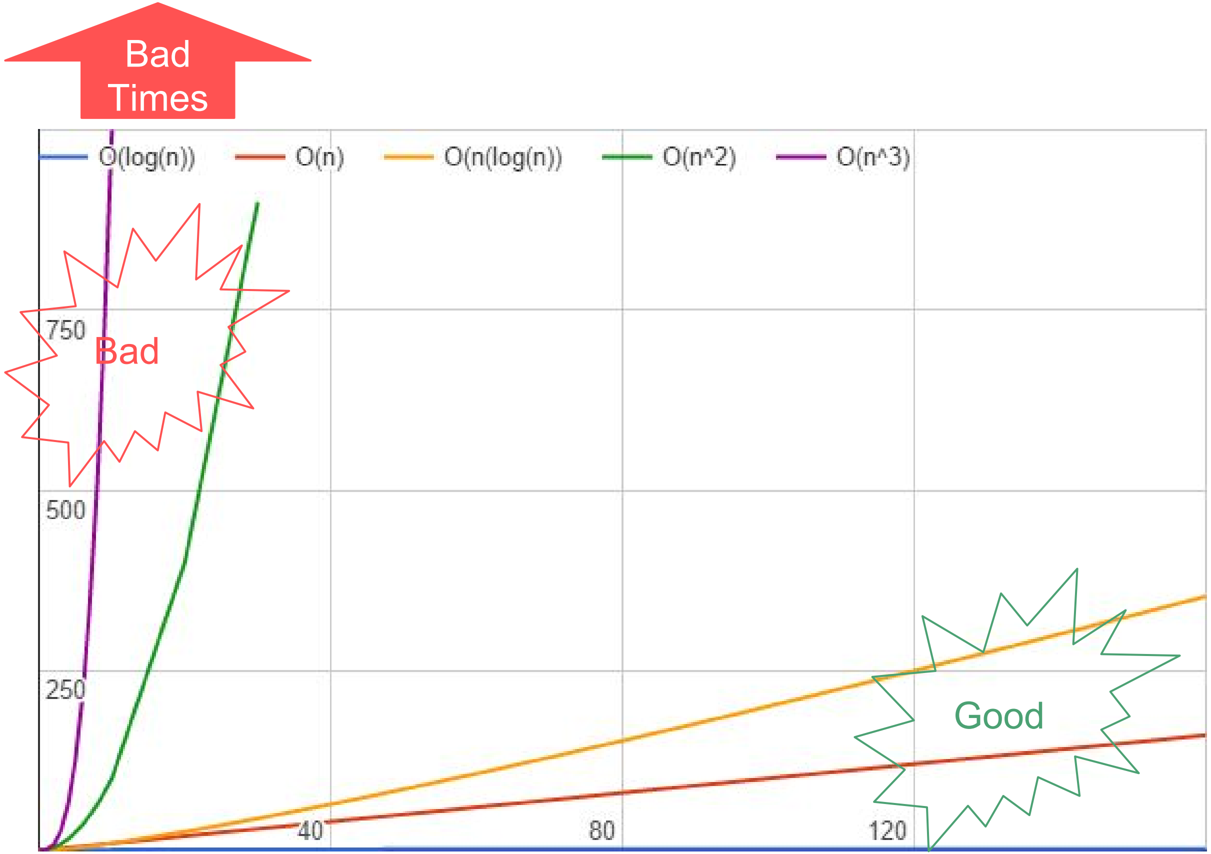
The CS language barrier

Abstract Data Types

- Abstract Data Types
  - Operations that can be performed
  - Operation performance characteristics
- Confusing because people may not use the “correct terminology”.
- Different implementations type may have different performance characteristics.
- Similar sounding names for slightly different things.
  - ECMAScript Map (Standard doesn’t specify but probably a Hash Table)
  - C# Dictionary (hash table)
  - C# SortedDictionary (binary search tree)
  - Java HashMap (hash table)
  - Java TreeMap (binary search tree)

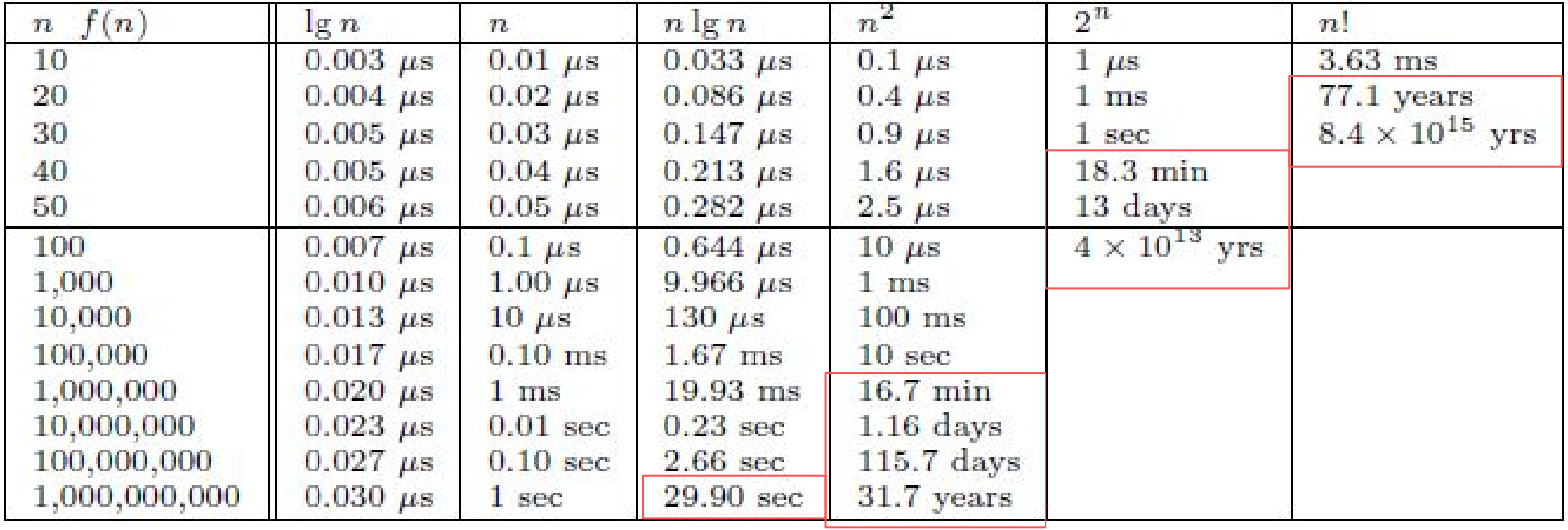
Big O Complexity

- AKA: “the run-time”, “the situation”, worst case runtime
- Way to talk about how something behaves as number of elements grow.
- If I add one extra element how does it affect the performance?
- Worst case number of operations.
- Independent of how long each individual operation takes.



Maximum problem size

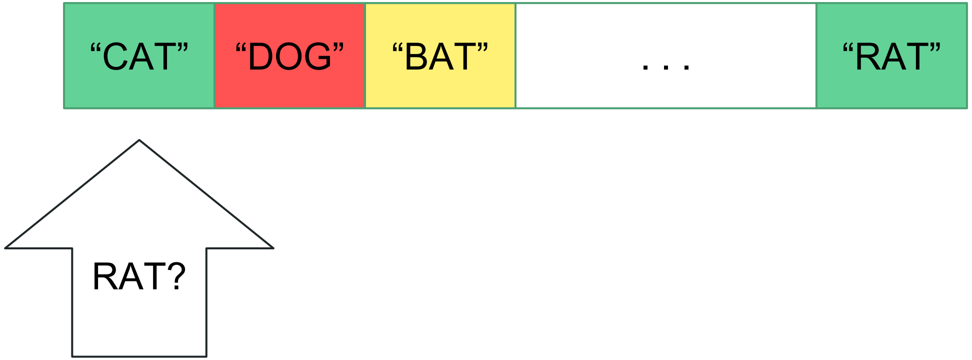
- What is the maximum sized problem that can be solved practically?



n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

Linear Complexity O(n): Find element in collection



“CAT” “DOG” “BAT” . . . “RAT”

RAT?

Hint:

for (var i …

Quadratic Complexity O(n^2): Joining two collections

{ Name: “Tom”, Type: “CAT” }

{ Name: “Bob”, Type: “CAT” }

{ Name: “Fido”, Type: “Dog” }

+

{ Name: “Fido”, Type: “Sean” }

{ Name: “Bob”, Type: “Jeff” }

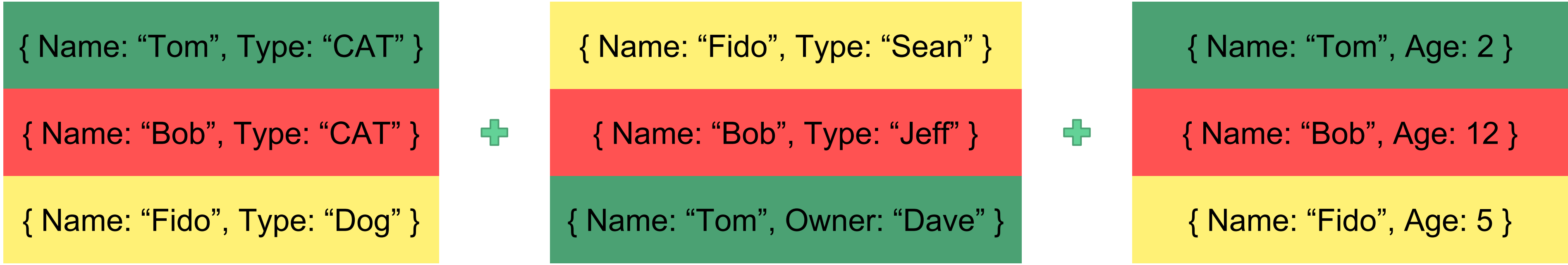
{ Name: “Tom”, Owner: “Dave” }

Hint:

```
for (var i …
   for (var j
```

{ Name: “Tom”, Type: “CAT”, Owner: “Dave” }
{ Name: “Bob”, Type: “CAT”, Owner: “Jeff” }
{ Name: “Bob”, Type: “CAT”, Owner: “Sean” }

Cubic Complexity O(n^3): Joining three collections



{ Name: “Tom”, Type: “CAT” }
{ Name: “Bob”, Type: “CAT” }
{ Name: “Fido”, Type: “Dog” }

+

{ Name: “Fido”, Type: “Sean” }
{ Name: “Bob”, Type: “Jeff” }
{ Name: “Tom”, Owner: “Dave” }

+

{ Name: “Tom”, Age: 2 }
{ Name: “Bob”, Age: 12 }
{ Name: “Fido”, Age: 5 }

Hint:

```
for (var i …
   for (var j ...
      for (var k ...
```

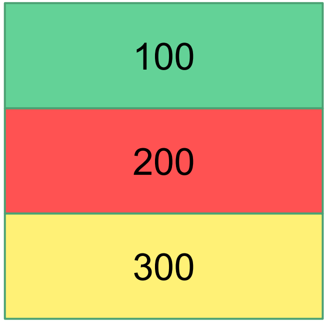
{ Name: “Tom”, Type: “CAT”, Owner: “Dave”, Age: 2 }
{ Name: “Bob”, Type: “CAT”, Owner: “Jeff”, Age: 12 }
{ Name: “Bob”, Type: “CAT”, Owner: “Sean”, Age 5 }

Arrays

The workhorse

Arrays

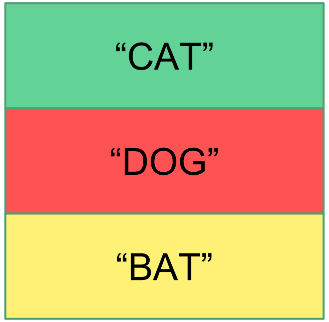
- Collection of elements
- Contiguous chunk of memory
- Should all be of same type*
  - Can insert multiple types in JS crazy-land. Please don't.
- Good at:
  - Iterating
  - Inserting/Removing from end.
  - Finding/Updating elements by index.
- Bad at:
  - Finding an element by some criteria**.
  - Inserting element in front or middle.
- Aliases: List (not LinkedList), Vector



100

200

300



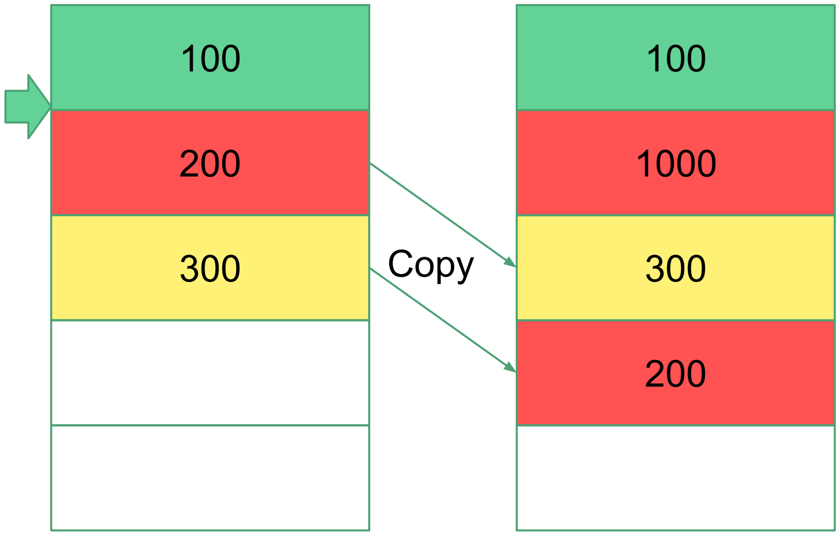
"CAT"

"DOG"

"BAT"

Inserting into arrays

Inserting at front/middle

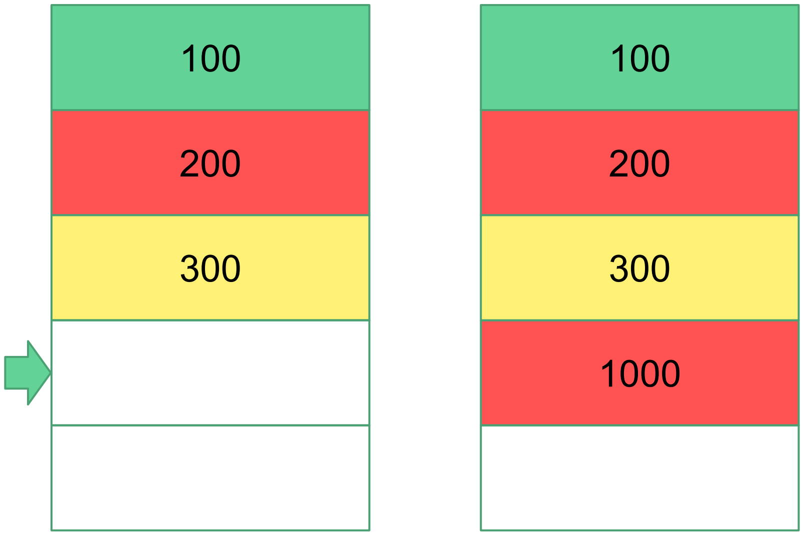


100
200
300

Copy

100
1000
300
200

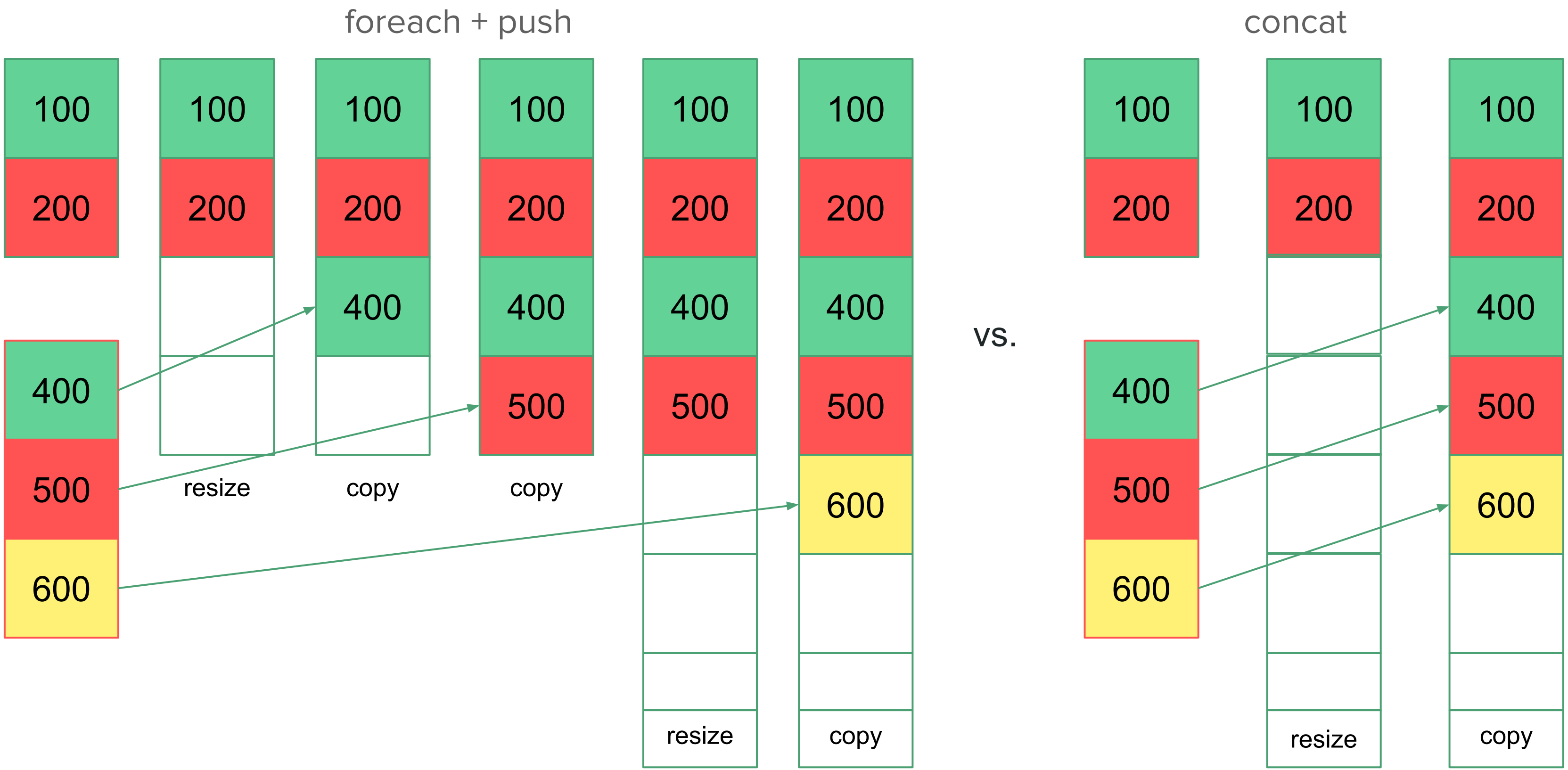
Inserting at end



100
200
300

100
200
300
1000

Resizing



foreach + push

concat

vs.

100 200 400 500 600

resize

copy

copy

resize

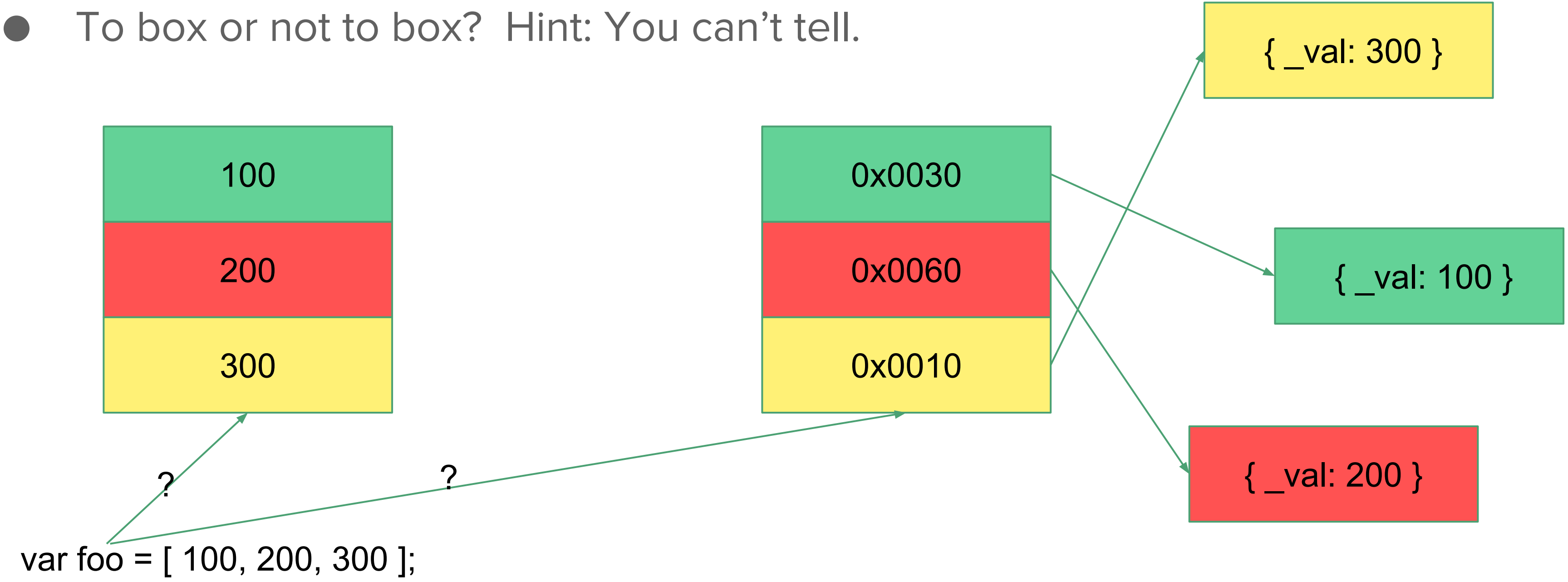
copy

resize

copy

Arrays of primitives or references?

- To box or not to box? Hint: You can’t tell.



Aside: ES6 Typed Arrays

- Allows working with collections of primitives stored as raw binary data.
- Intended for graphics, video and audio so probably not what you want.

```
const buffer = new ArrayBuffer(16); // 16 bytes
const int32View = new Int32Array(buffer); // 16 bytes / 4 bytes per int = 4 ints

for (let i = 0; i < int32View.length; i++) {
  int32View[i] = i;
}

int32View.forEach(x => console.log(x)); // 0, 1, 2, 3
```

Map

The bonus data structure

Maps

- Collection of key:value pairs
- Good at:
  - Looking things up based on key.
  - Random order inserts and deletes based on key.
- Bad at:
  - Iterating in sorted order.
  - Finding next/previous element in sorted order.
- Aliases: HashMap, Dictionary

Hashing

- Soon to be legal in Canada.
- Goal: Create a “hopefully” unique identifying number for an object.

```
function getHashCode(s) {
  if (!s) { return 0; }

  var hash = 7;
  for(var i = 0; i < s.length; i++) {
    hash += 3 * s.charCodeAt(i); // 3 * UTF-16 char code
  }

  return hash;
}
```

```
var a = "This is a string.";
var b = "This is a different string.";
console.log(getHashCode(a)); // 4597
console.log(getHashCode(b)); // 7546

// Aside: Horrible because "abc" gives same value as "cba"
console.log(getHashCode("abc")); // 889
console.log(getHashCode("cba")); // 889
```

Hashing + Array = Hash Map

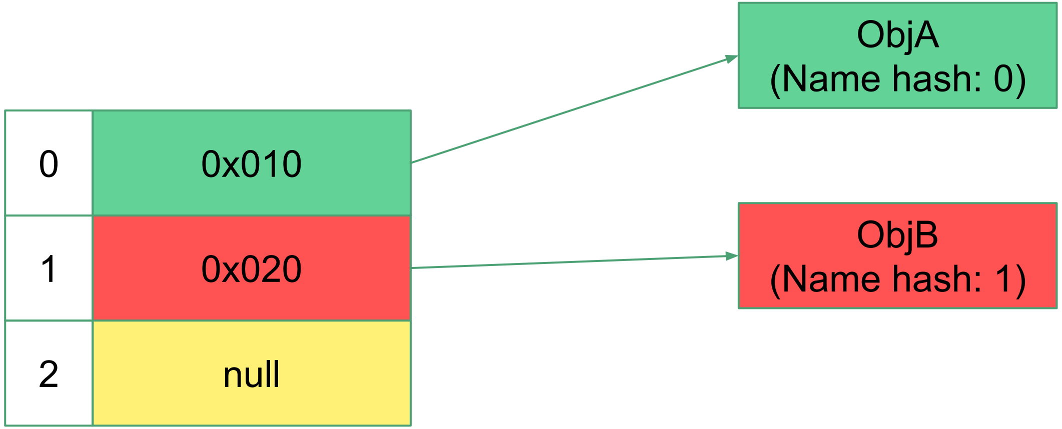
- What if we we used an object’s hash code as an index into an array?

```
var key = objB.Name;
console.log(key.getHashCode()); // 1

// Store
var map[key] = objB;

// Retrieve
var objBcopy = map[key];

// Reference equality
objB === objBcopy; // true
```



0	0x010
1	0x020
2	null

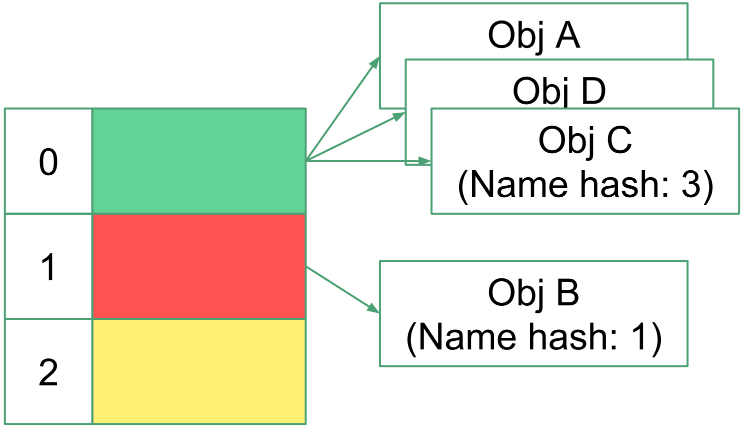
ObjA
(Name hash: 0)

ObjB
(Name hash: 1)

Q: What is the run-time complexity of store? retrieve?

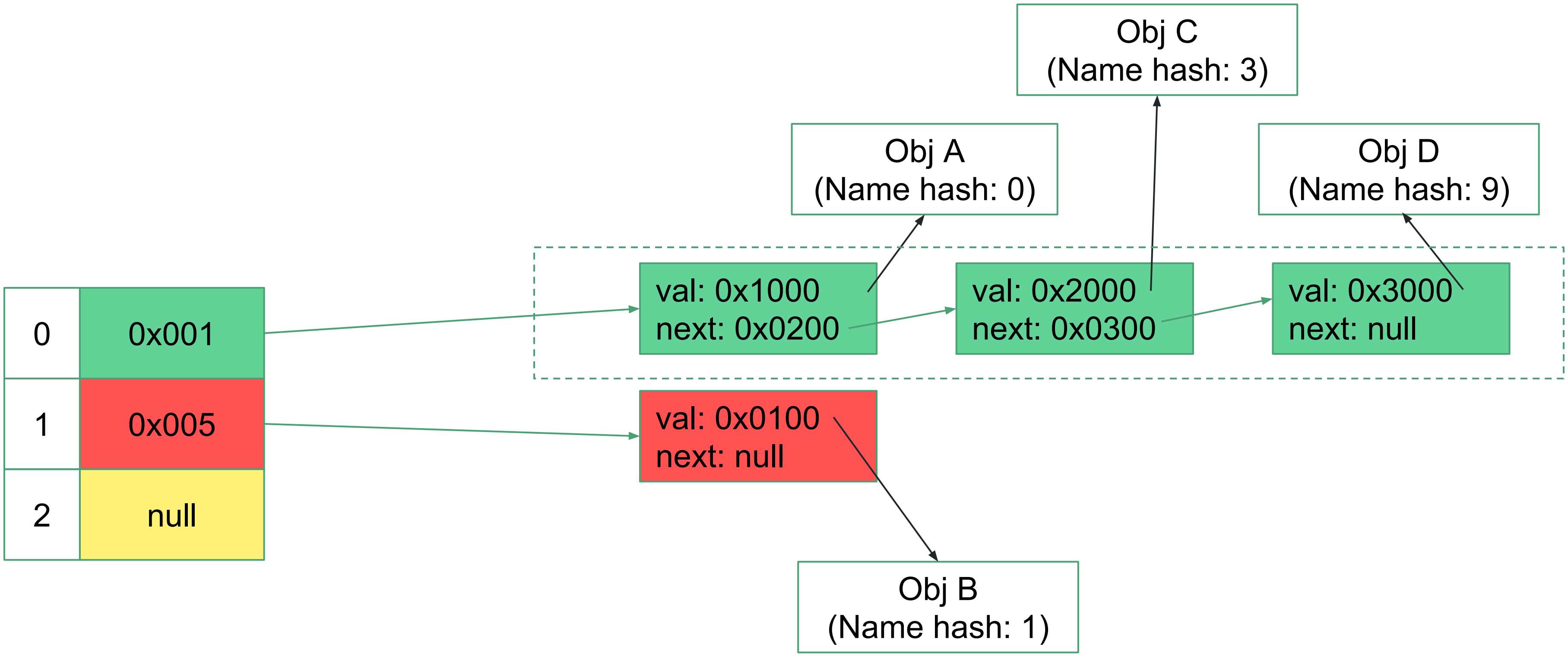
Array Size << Possibile Hash Codes

	% SIZE (% 3)
Obj A (Name hash: 0)	0
Obj B (Name hash: 1)	1
Obj C (Name hash: 3)	0
Obj D (Name hash: 9)	0



Int32: -2,147,483,648 to 2,147,483,647

Separate Chaining



0	0x001
1	0x005
2	null

val: 0x1000
next: 0x0200

val: 0x2000
next: 0x0300

val: 0x3000
next: null

val: 0x0100
next: null

µPattern: Join 2+ collections

```
var primary = [
  { Name: "Tom", Type: "Cat" },
  { Name: "Bob", Type: "Cat" },
  { Name: "Tim", Type: "Dog" }
];

var secondary = new Map([ ["Tom", 5], ["Bob", 12] ]);

var owners = [
  { Name: "Tom", Owner: "Dave" },
  { Name: "Bob", Owner: "Jeff" }
];
var tertiary = new Map();
owners.forEach(x => tertiary.set(x.Name, x.Owner));
```

```
// Join together to create results
var results = [];
primary.forEach(x => results.push({
  Name: x.Name,
  Owner: secondary.get(x.Name) || 'Unknown',
  Age: tertiary.get(x.Name) || null
}));
```

```
Results:
   {"Name":"Tom","Owner":5,"Age":"Dave"},
   {"Name":"Bob","Owner":"Unknown","Age":"Jeff"},
   {"Name":"Tim","Owner":12,"Age":null}
```

Q: What is the run-time complexity?

Q: Why not just objects + properties instead?

Q: Why not just use objects + properties (ES5 style)?

```
var map = new Map([
  ["1", 'String'],
  [1, 'Number']
]);
```

Non-String keys!

```
for (let key of map.keys()) {
  console.log("Key: ", key);
}

for (let value of map.values()) {
  console.log("Value:", value);
}
```

Easily iterate over keys and and values!!!

```
console.log(map.get(1)); // Number
console.log(map.get('1')); // String
console.log("Size: ", map.size); // 2
```

size!!

Also:

- Interpreter may be able to optimize.
- Communicates intent to other programmers.

Q: Why not just use arrays?

Q: Why not just use arrays for everything?

Few elements/items

- Using an array isn't a bad idea
- The number of elements (N) is usually small.
- Arrays are simple. KISS.

⬅80% vs. 20%➡

Lots of elements/items

- What number do we consider "lots"? 10, 100, 1000?
- Apps are getting more complex and pulling more data from the backend.
- Faster to do filtering and sorting on front end than to launch another request.
- Will the amount of data grow over time?

Correctness

- Would another data structure make the intent of this code more obvious?
- Want clarity with a bias towards simplicity.

Set

The bonus data structure

Set

- A Set is basically map with no value.
- Best for:
    - Checking for presence/absence of something of a key.
    - Finding the intersection and disjoint sets of elements between two groups.
- Aliases: HashSet

µPattern: Find unique elements in collection

```
const things = [ "Cat", "Dog", "Rat", "Cat", "Bat", "Bat", "Ant", "Rat" ];
const uniqueThings = new Set(things);
console.log(uniqueThings); // "Cat", "Dog", "Rat", "Bat", "Ant"
```

Q: What is the run-time complexity?

µPattern: Join 2+ collections

```
const primary = [
  { Name: "Tim", Type: "DOG" },
  { Name: "Bob", Type: "CAT" },
  { Name: "Tom", Type: "CAT" }
];

const appointments = [
  { Name: "Tim", Type: "CHECKUP" },
  { Name: "Bob", Type: "CHECKUP" },
  { Name: "Tim", Type: "VACCINATION" },
];

const secondary = new Set();
appointments
  .filter(x => x.Type == "VACCINATION")
  .forEach(x => secondary.add(x.Name));
```

```
const results = [];
primary.forEach(x => results.push({
  Name: x.Name,
  Type: x.Type,
  IsVaccinated: secondary.has(x.Name)
}));
```

```
Results:
 {"Name":"Tim","Type":"DOG","IsVaccinated":true}
 {"Name":"Bob","Type":"CAT","IsVaccinated":true}
 {"Name":"Tom","Type":"CAT","IsVaccinated":false}
```

Q: What is the run-time complexity?

Immutable (Persistent) Data Structures

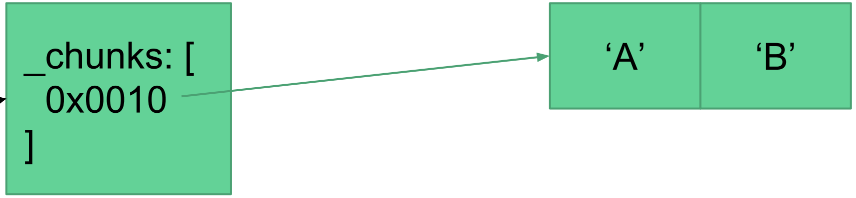
Those things the React/Flux gurus go on about.

Origins of Immutable (Persistent) Data Structures

- Problem: Functional programming likes to create new collections and objects instead of mutating the existing ones.
- Lots of temporary copies = lots of garbage.
- Could we separate collections into changed and unchanged sections replacing only the changed sections?

Hand-wavey explanation [1/3]

```
let setA = new Set(“A”, “B”);
let setB = setA;
let setC = setB;
```



_chunks: [
0x0010
]

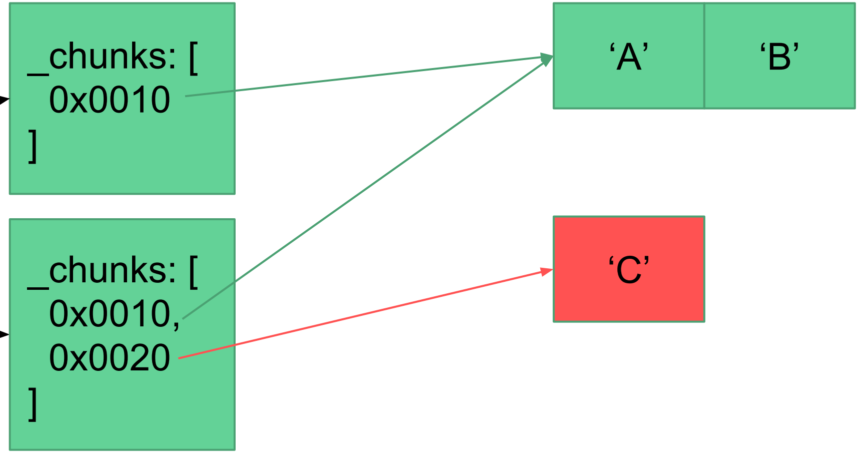
‘A’ ‘B’

```
console.log(setA); // “A”, “B”
console.log(setB); // “A”, “B”
console.log(setC); // “A”, “B”
```

Q: How could we tell that SetA, SetB, and SetC are the same?

Hand-wavey explanation [2/3]

```
let setA = new Set("A", "B");
let setB = setA;
let setC = setB;
setC = setC.add("C");
```



_chunks: [
0x0010
]

_chunks: [
0x0010,
0x0020
]

'A'

'B'

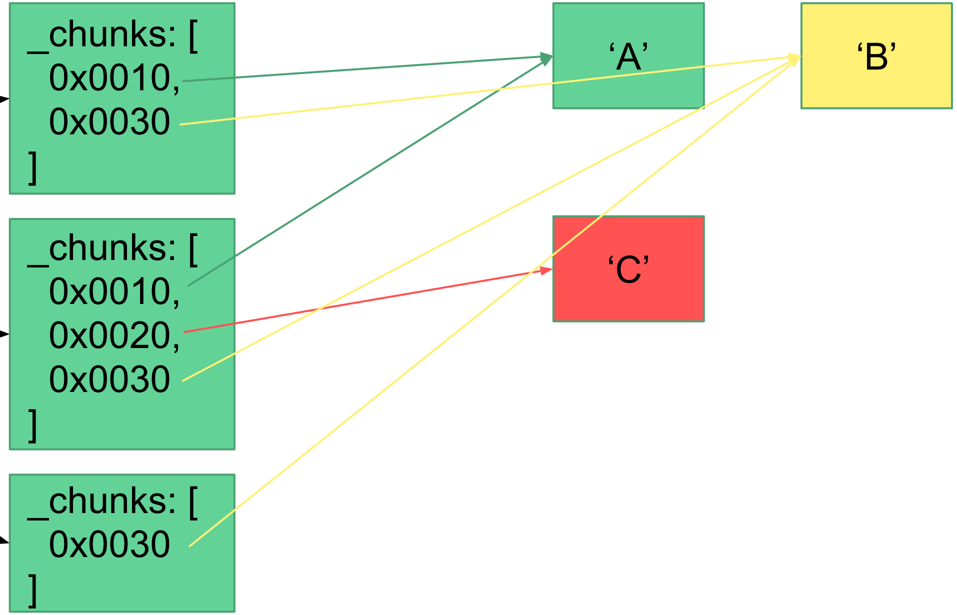
'C'

```
console.log(setA); // "A", "B"
console.log(setB); // "A", "B"
console.log(setC); // "A", "B", "C"
```

Q: How could we tell that SetA and SetC are different?

Hand-wavey explanation [3/3]

```
let setA = new Set(“A”, “B”);
let setB = setA;
let setC = setB;
setC = setC.add(“C”);
setB = setB.remove(“A”);
```



```
_chunks: [
  0x0010,
  0x0030
]
```

```
_chunks: [
  0x0010,
  0x0020,
  0x0030
]
```

```
_chunks: [
  0x0030
]
```

```
console.log(setA); // “A”, “B”
console.log(setB); // “B”
console.log(setC); // “A”, “B”, “C”
```

Immutable.js + React.js

- From a React perspective the interesting bit is that we can check for changes quickly.
- What if we pass in a set as a prop? Do we need to re-render if the set hasn't changed?
- Immutable.js gives you collections that you can check for changes in constant time instead of searching through collection’s elements.

```
shouldComponentUpdate: function(nextProps) {
  return nextProps.setA !== this.props.setA; // Compare references
}
```

Summary

AKA: Quiz Time

Summary / Quiz Time

- [True / False] We should use all the fancy stuff all the time.
- What complex thing do strings “usually” hide from us?
- When is n considered small?
- [True / False] Inserting in the middle of an array always causes a resize?
- How long does it take to search a list looking for x.Id === 2?
- What is a map good at?
- How long does it take to search a map for myMap[objA.Id]?
- What is a hash code?
- What is a collision?
- Why are immutable data structures useful in React?