CompSci – Data Structures

The Coding Bootcamp | July 8, 2017

Outline

- Project Check-In
- Computer Science Context
- Big O Notation
- Data Structures
  - Arrays
  - Stacks / Queues
  - Linked Lists
  - Dictionaries
  - Hash Tables
  - Sets
  - Binary Trees and Binary Search Trees
  - Graphs

Project Check-In

Project Status?

Smooooooth Sailing?

Project Check-In

Remember!

Deliverable #2 is due today by the end of class.

Please send the following to your Instructor + TAs:

- Prototype of application key features

Submit by the end of the day (9:00 PM)!

Computer Science Context

Welcome To…

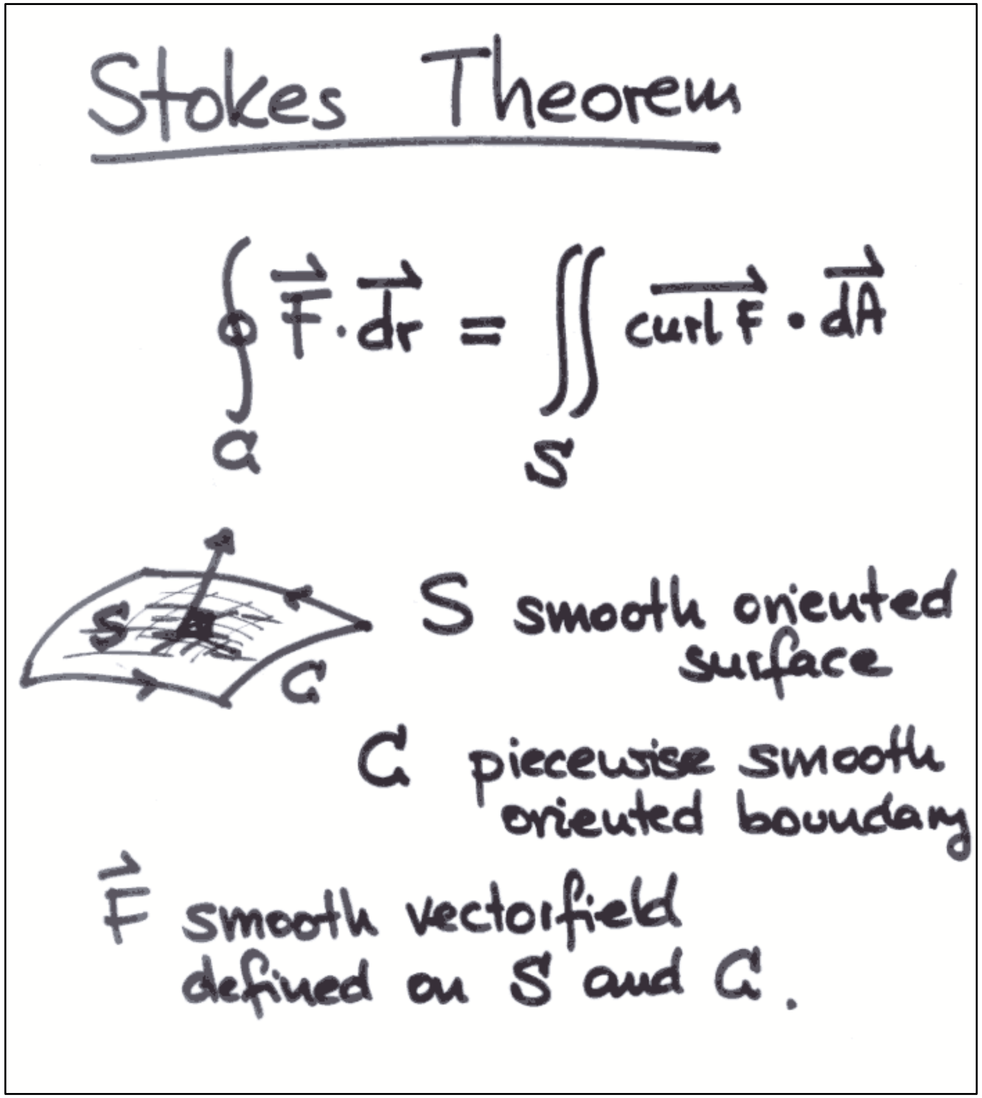
“Computer Science Fundamentals”

Remember…

Computer Science "Fundamentals"

- Isn't about "easy" computer science stuff.
- Rather, it's about the "fundamental" concepts that underlie all of the work we've been doing to date.
- The biggest takeaway is to understand that there are different tools to increase computational efficiency.

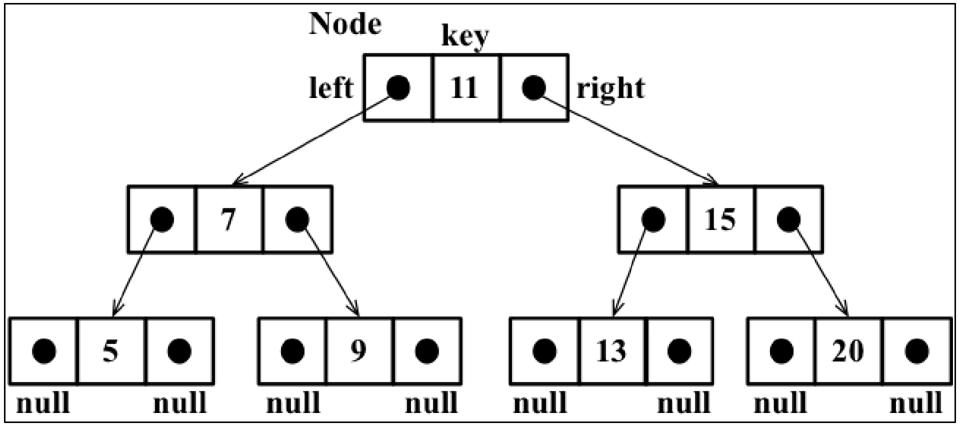
“Fundamentals”



Remember this stuff?

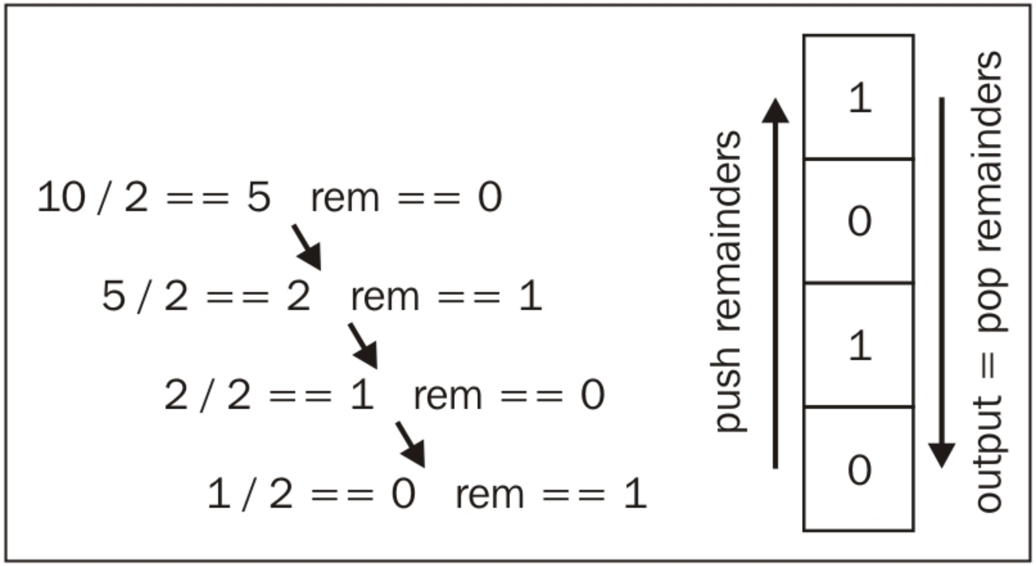
Yeah. Me neither.

It gets hairy… and scary.

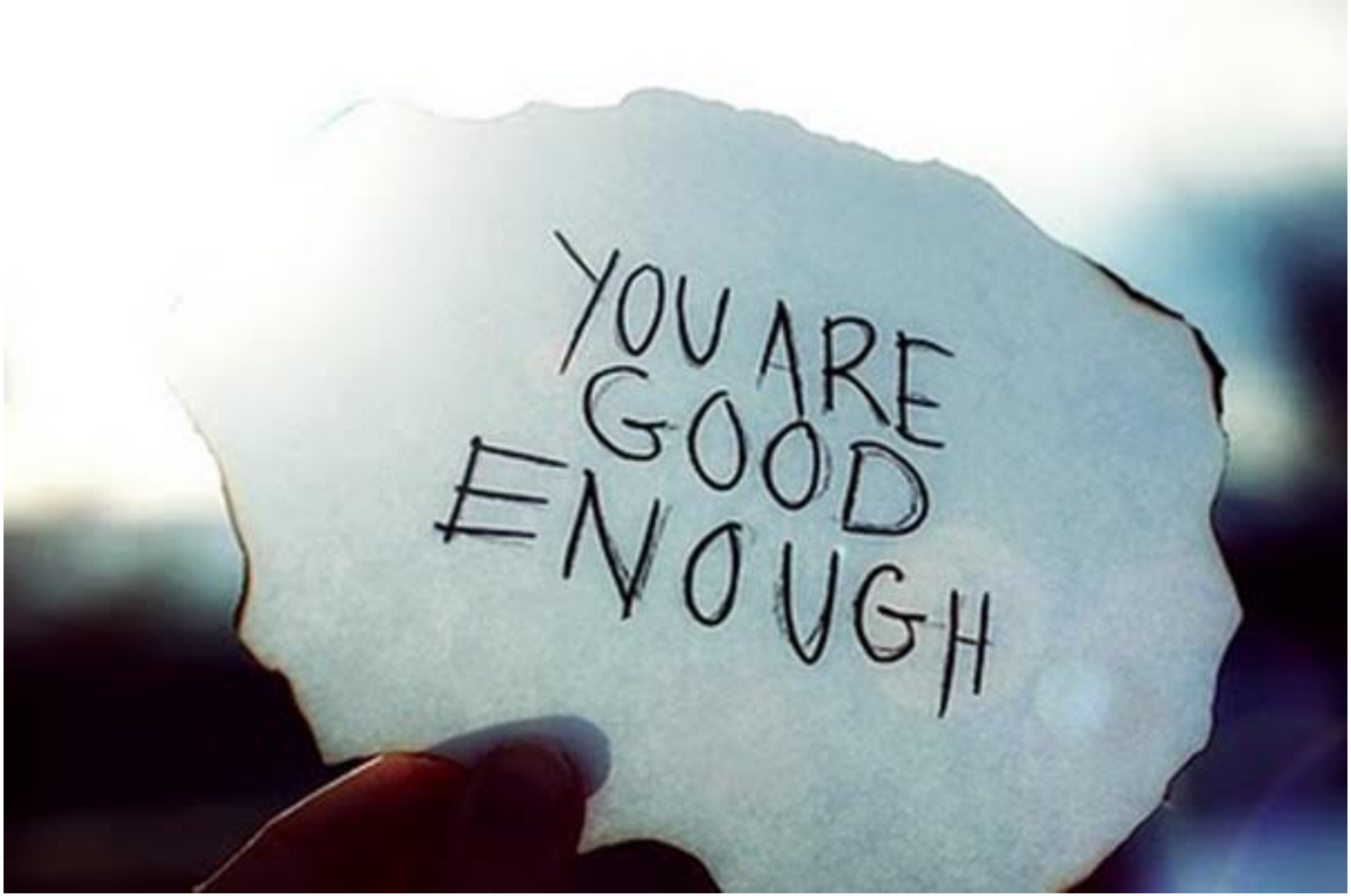



```
var fromVertex = myVertices[0]; //{9}
for (var i=1; i<myVertices.length; i++){ //{10}
  var toVertex = myVertices[i], //{11}
  path = new Stack();        //{12}
  for (var v=toVertex; v!== fromVertex;
  v=shortestPathA.predecessors[v]) { //{13}
    path.push(v);                          //{14}
  }
  path.push(fromVertex);       //{15}
  var s = path.pop();          //{16}
  while (!path.isEmpty()){     //{17}
    s += ' - ' + path.pop(); //{18}
  }
  console.log(s); //{19}
}
```

```
function divideBy2(decNumber){

  var remStack = new Stack(),
  rem,
  binaryString = '';

  while (decNumber > 0){ //{1}
    rem = Math.floor(decNumber % 2); //{2}
    remStack.push(rem); //{3}
    decNumber = Math.floor(decNumber / 2); //{4}
  }

  while (!remStack.isEmpty()){ //{5}
    binaryString += remStack.pop().toString();
  }

  return binaryString;
}
```



Be Wary of Imposter Syndrome!



Don’t let the hard stuff scare you…

Why Cover This?

1. These concepts sometimes appear in **coding interviews**

2. When inheriting large code-bases you may be tasked to “**optimize” code efficiency.**

3. The computational challenges here forces you to **deepen your understanding**.

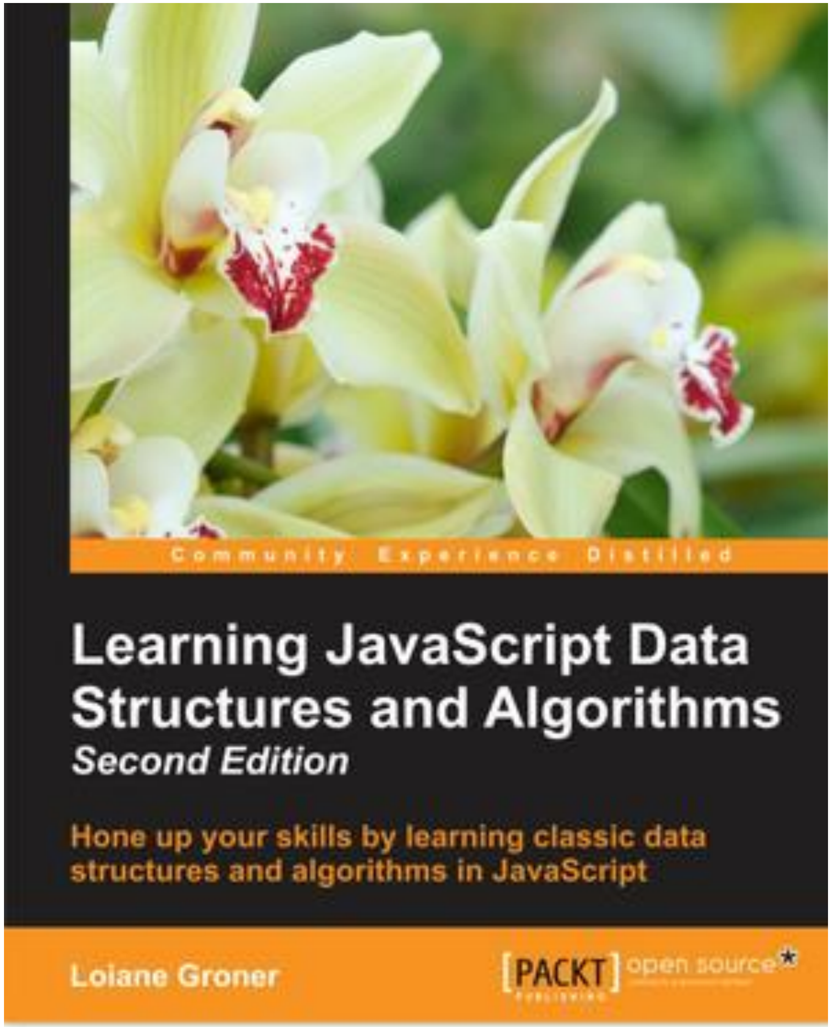
Bottom Line

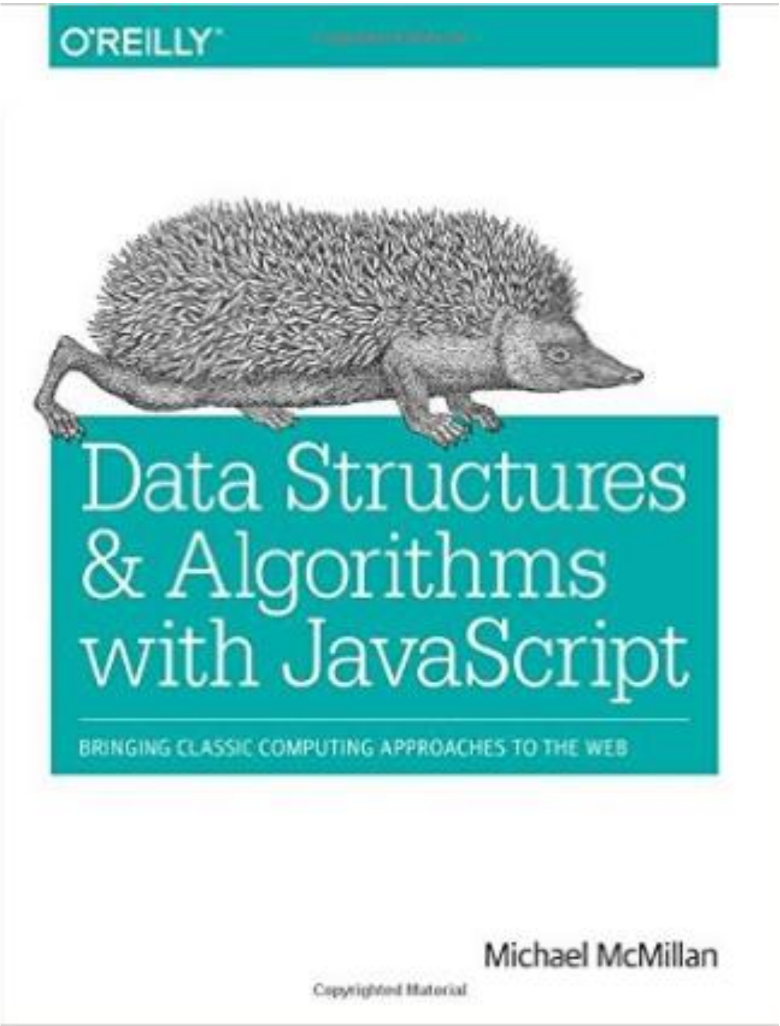
My goal is to give you the <u>terminology</u> and the <u>concepts.</u>

Enough insight that you can understand the context of interview questions that come your way.

And… to encourage those of you into math to take a second look.

Going Deep





For those that dare dive deeper.

Efficiency

What does “efficient” mean?

We talk a lot about “efficiency”.

What does “efficient” mean?

But…

What does “efficient” mean?

What, *exactly*, does “**efficient**” mean?

Fewer Steps = Faster Code

Number of steps ~ Efficiency

Fewer Steps = Faster Code

More steps = Less Efficient

Fewer Steps = More Efficent

What’s a “step”?

- A step is an **instruction** to the computer.
- All computations boil down to a handful of “basic steps”.
  - Arithmetic (+, `*`, etc.)
  - Assignment (`var x = 42;`)
  - Boolean tests (`x === 42`)
  - Reading from memory
  - Writing from memory

What’s a “step”?

Each of these counts as a step.

What’s a “step”?

Fewer Steps = Faster Code

Pop Quiz (!)

Which function is more efficient?

```
function list_items (list) {
  for (var i = 0; i < list.length; i += 1) {
    // Log each item in the array
    console.log(list[i]);
  }
}

function head (list) {
  // Return first item of a list
  return list[0];
}
```

(Which has fewer instructions?)

Count Instructions

Count the instructions!

Count Instructions

`head =` 1 instruction

Count Instructions

`list_items = n` instructions

...

`(n = list.length)`

The Verdict

`head` is more efficient.

The Verdict

But `list_names` isn't bad…

Time Complexity

Quantifying Efficiency

`head` ***always*** executes one instruction…

Quantifying Efficiency

…No matter how long our array is

Quantifying Efficiency

```
// Three elements...
var names = ['Gogol', 'Pushkin', 'Dostoevsky'];

// One thousand elements...
var huge_array = generate_array(1000);

// ...But these statements take
//    the same amount of time.
console.log( head(names) );
console.log( head(huge_array) );
```

`head` takes same amount of time on **any** input

Quantifying Efficiency

`list_items` needs `n` instructions

Quantifying Efficiency

```
function list_items (list) {
  for (var i = 0; i < list.length; i += 1) {
    // Log each item in the array
    console.log(list[i]);
  }
}
```

One `console.log` **per item**

Quantifying Efficiency

`console.log` is fast…

Quantifying Efficiency

…but **not** free.

Quantifying Efficiency

Longer arrays = more time

Quantifying Efficiency

Double array length = Double time

Triple array length = Triple time

In other words…

The **running time** of `head` and `list_items` **scale differently.**

The **running time** of `head` and `list_items` **scale differently.**

Time complexity = Rate at which algorithm **slows** as input **grows**

Quantifying Efficiency

`head` is **always** one instruction

Quantifying Efficiency

Running time **does not** slow for larger inputs

In other words…

Quantifying Efficiency

The running time of `head` is **constant.**

Quantifying Efficiency

`list_items` takes `n` instructions

Running time **depends on array**

Quantifying Efficiency

Double array length, double time

Etc…

Quantifying Efficiency

Running time **increase linearly** with array length.

Big O Notation

Big O

- **Big O notation** lets us describe how running time scales when we increase the input size (*n*)

- Denoted with a big O, and the “growth factor” in parentheses

- Examples:
  - `head` ~ O(1)
    - Grows like “1”—i.e., running time never grows
  - `list_items` ~ O(n)
    - Grows like “n”—i.e., gets bigger as *n* gets bigger

There are other Big O “classes”

Big O

```
function find_duplicates (list) {
  var duplicates = [];

  for (var i = 0; i < list.length; i += 1) {
    var current = list[i];

    for (var j = 0; j < list.length; j += 1) {
      if (j === i)
        continue;
      else if (current === list[j] && !duplicates.includes(list[j]))
        duplicates.push(current);
    }

  }

  return duplicates;
}
```

`n` steps for each of the `n` items in `list` (!)

Big O

2x length = 4x time

3x length = 9x time

nx length = n^2 time

Running time grows as *square* of input

`find_duplicates` ~ $O(n^2)$

"***Quadratic*** time complexity"

<u>MAJOR INSIGHT</u>

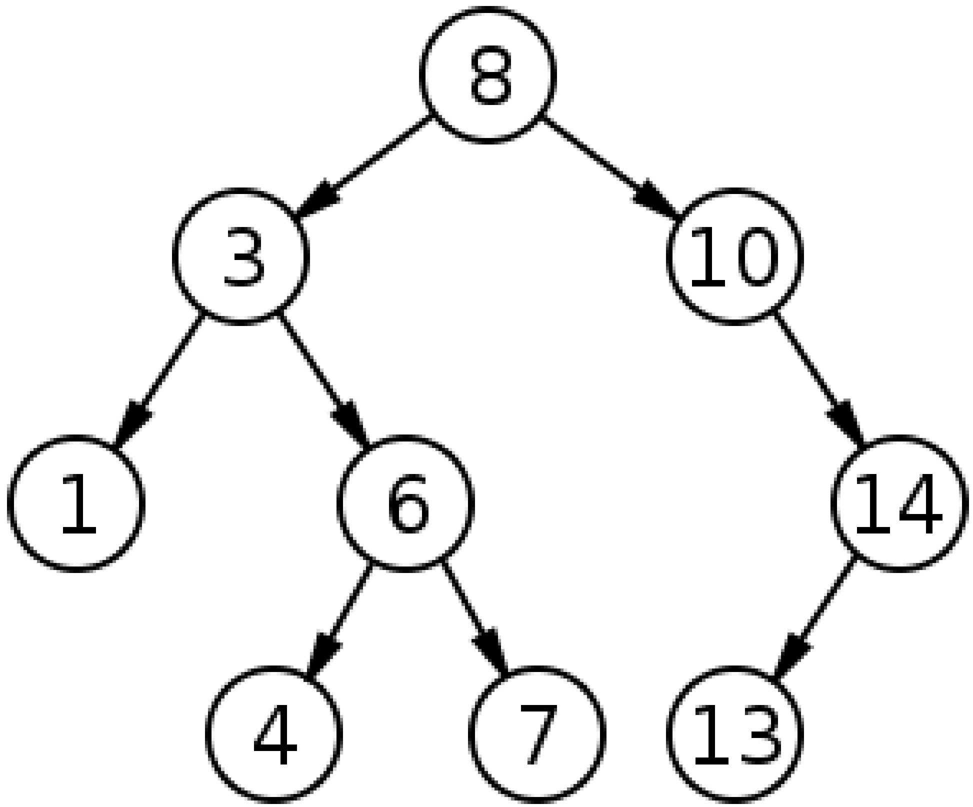
2 nested `for` loops ~ $O(n^2)$

NOT COINCIDENCE!

3 nested `for` loops ~ O(n^3)

Etc.

One more…



How fast is binary search?

Is it…

- $O(1)$
- $O(n)$
- $O(n^2)$
- Something else?...

Something else.

Why?

Exercise

```
// Ready for binary search!
var sorted = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Binary search this array by hand, for 3, then 9.

Count the steps.

Big O

3 steps.

Add the digits 11-20. Repeat.

4 steps (!)

<u>Much</u> faster than linear.

(input size)^2 ~ 2x running time

(input size)^3 ~ 3x running time

Etc.

This is called O(lg n).

lg n = how many times do I divide n by two to get to 1?

Logarithm Example

What is lg 8?

Logarithm Example

8 / 2 = 4 (1)
4 / 2 = 2 (2)
2 / 2 = 1 (**3**)

Logarithm Example

$$\lg 8 = 3$$

But if this is confusing…

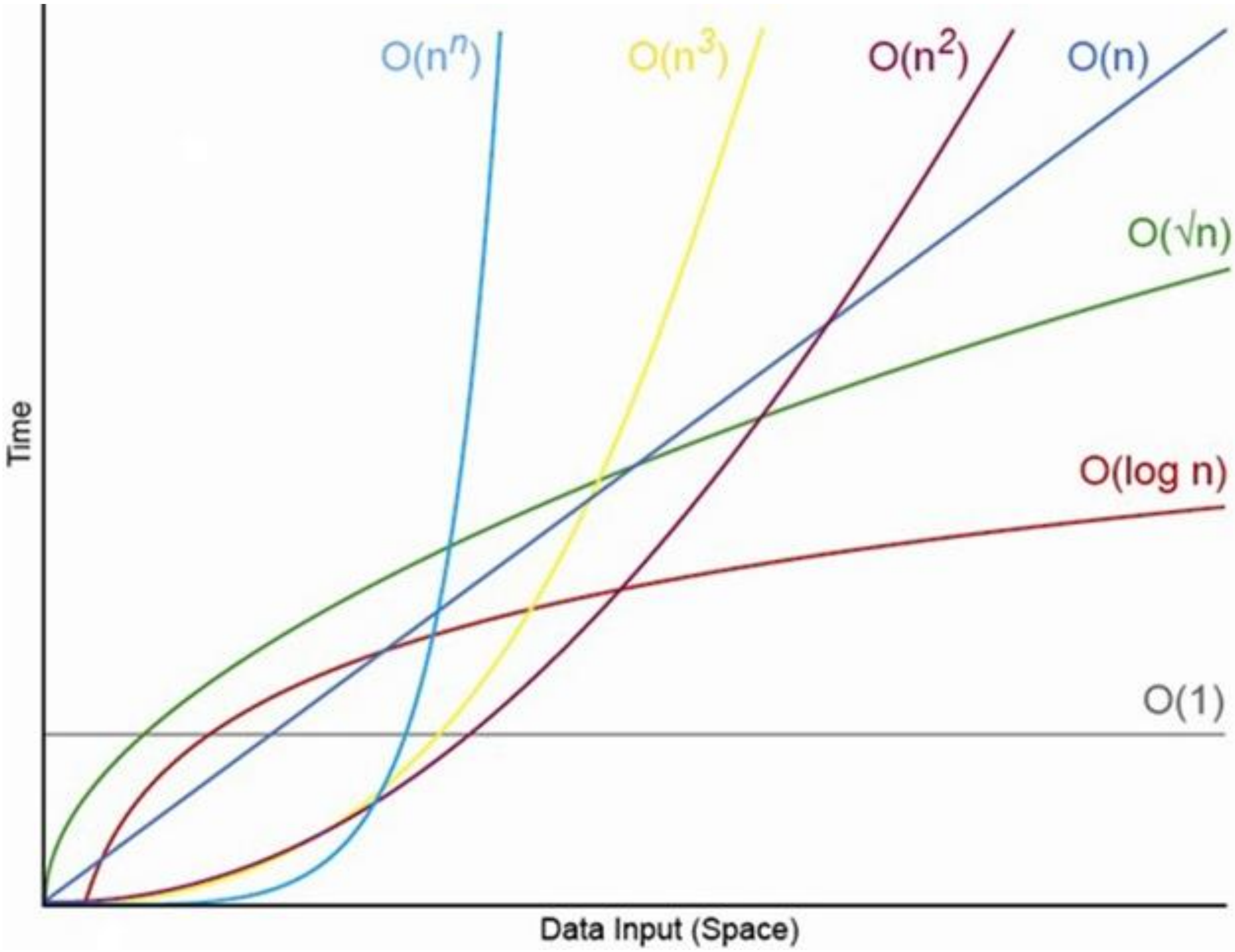
Logarithm Example

Don't worry about it.

Big O Review

- `head` ~ O(1)
  - Grows like "1"—i.e., 2x input size -> 1x running time
- `list_items` ~ O(n)
  - Grows like "n"—i.e., 2x input size -> 2x running time
- `find_duplicates` ~ O(n^2)
  - Grows like "n^2"—i.e., 2x input size -> 4x running time
- `binary_search` ~ O(lg n)
  - Grows like "lg n"—i.e, $(\text{input size})^2$ -> 2x running time

Big O Comparisons



Data Structures

Data Structures? (Tricky Question)

What is a data structure?

(And what is an example?)

Data Structures? (Tricky Question)

Before we answer that…

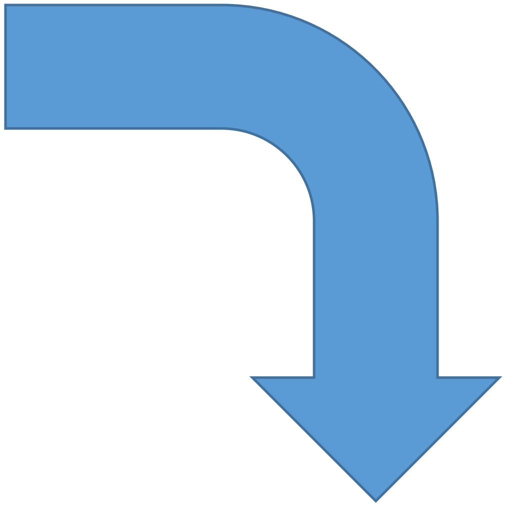
Code = Data. Data is Saved.

Code we write…

```
var name = Ahmed
```

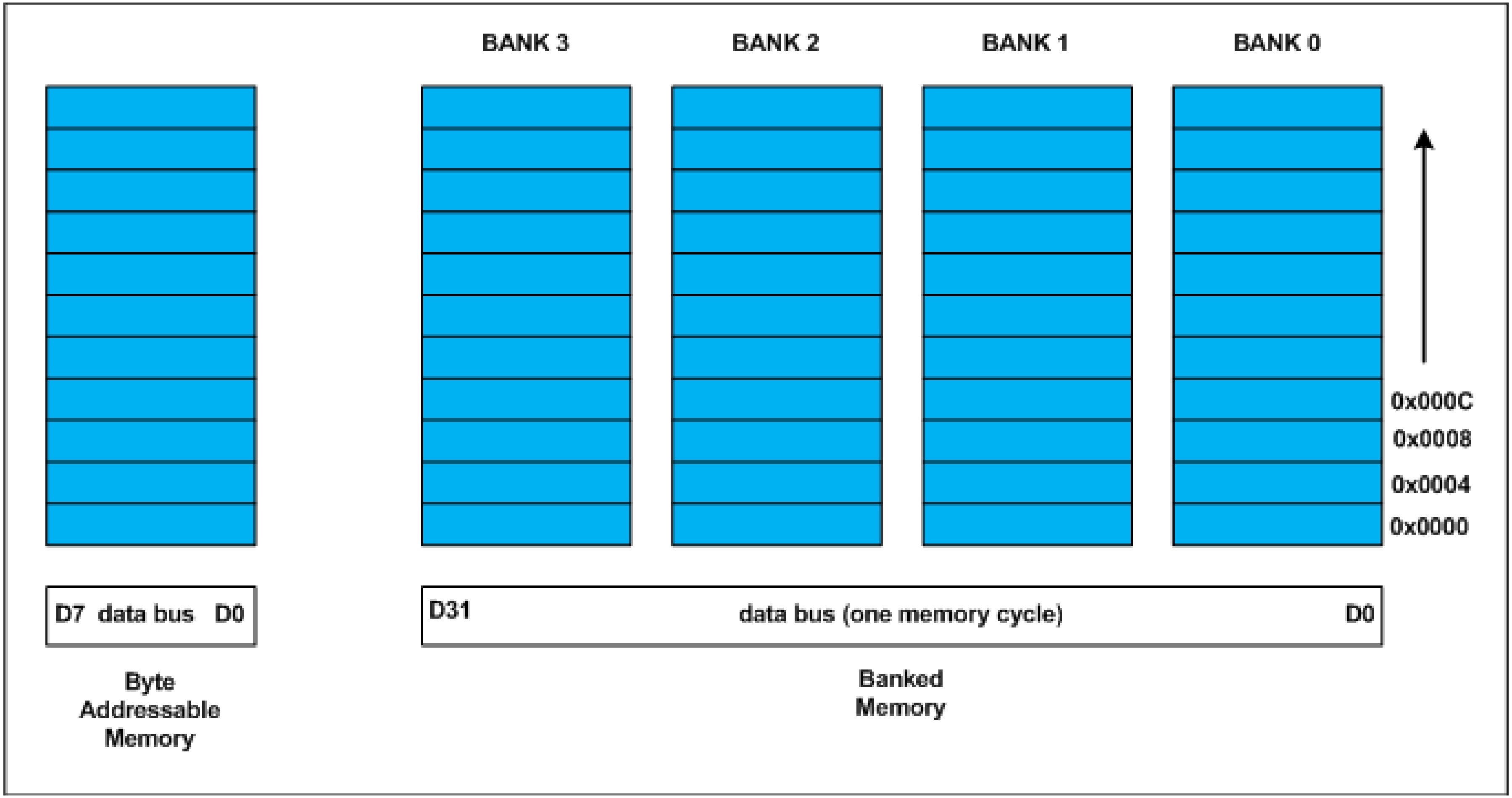
```
var age = 82
```

```
var isCool = true
```



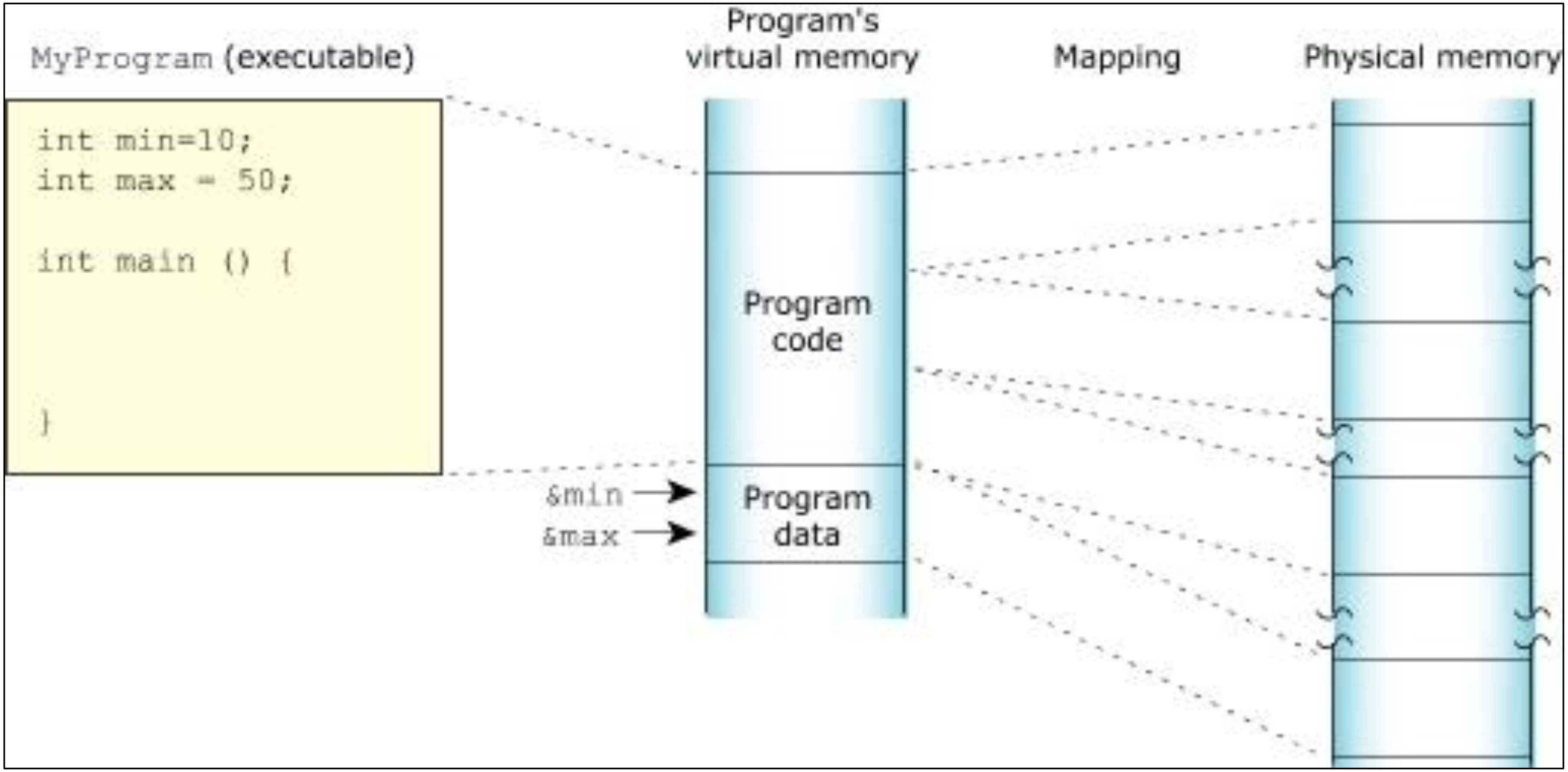
Gets saved in memory…

Different Ways to Save…



Memory can be visualized as slots. Data is then allotted into these slots.

Memory on My Mind



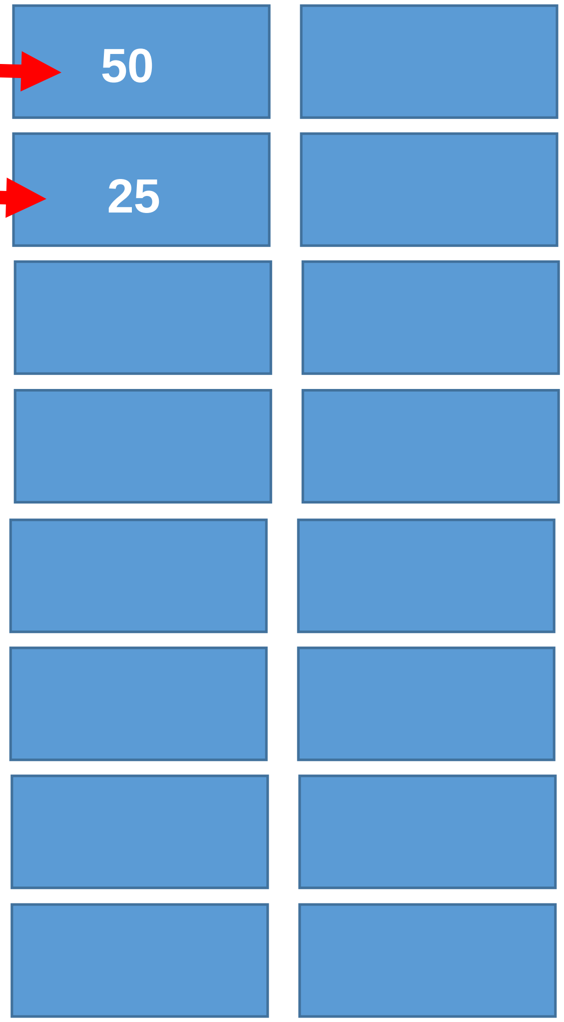
- Our code as a whole takes some of these slots of memory.
- Our variable data itself also takes slots of memory.

Saving to Memory…

Code

```
var num1 = 50;
var num2 = 25;
```

Memory



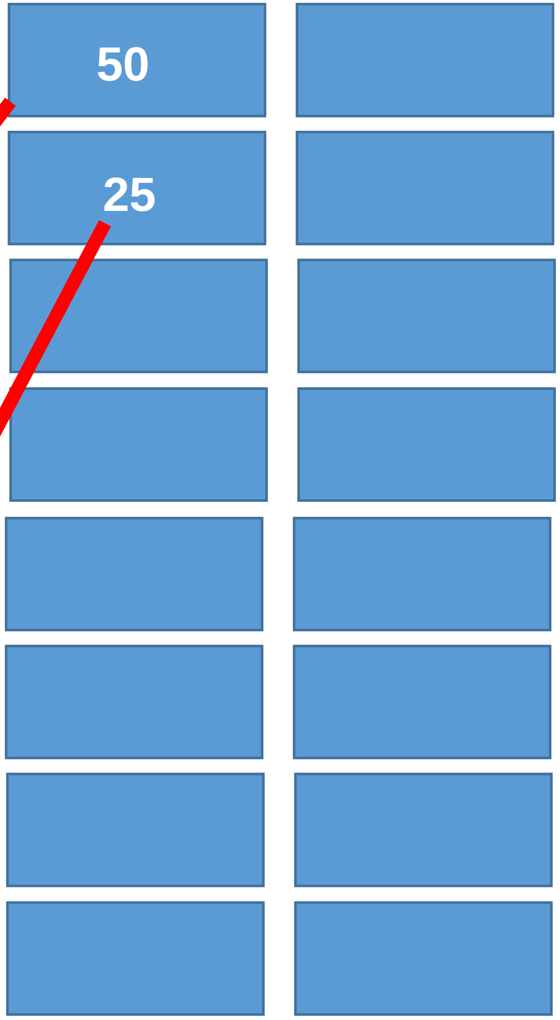
Each time we declare or instantiate a variable, we are **saving** that data to memory.

Retrieving from Memory…

Code

```
var num1 = 50;
var num2 = 25;
```

Memory



When we reference these variables in our code, we are **retrieving** the data from memory.

```
console.log(num1 + num2);
```

Growing Data = Growing Problem

- As applications grow and we begin to incorporate larger quantities of information with inter-relationships…

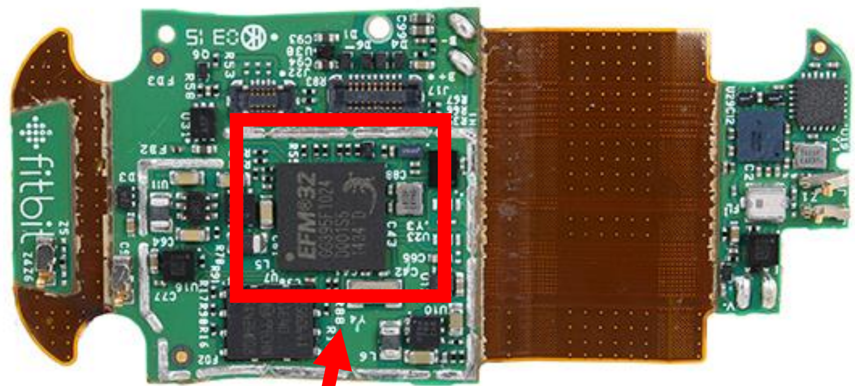
- These simple operations of saving, retrieving, etc.

- Become a lot more intensive (both time-wise and CPU processing wise).

- ***Don’t let the simplicity fool you!***

Building Devices

<u>Fitbit Surge</u>



You have 1 MB. Use it wisely

<u>Devices inherently have limited memory because of space requirements – making efficiency decisions critical</u>

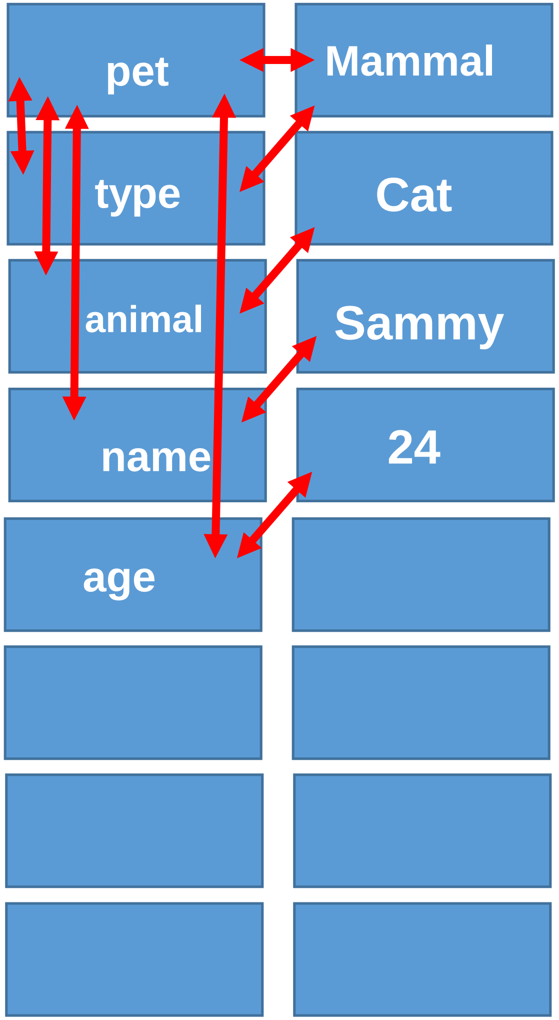
Retrieving from Memory…

Code

```
var pet = {

    type: "Mammal",
    animal: "Cat"
    name: "Sammy",
    age: 24
}
```

Even simple objects, require memory to keep track of numerous relationships in memory.

Memory



What is a data structure?

A way of storing data so that it can be used efficiently by the computer or browser.

<u>What is a data structure?</u>

They are built upon simpler primitive data types (like variables)

What is a data structure?

They are non-opinionated, in the sense, that they are just responsible for holding the data.

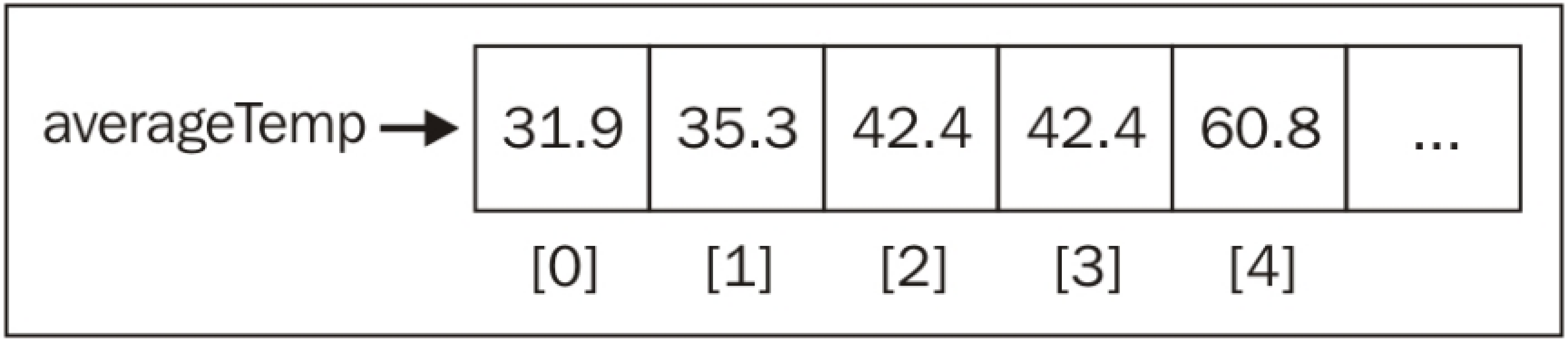
Example Data Structure:

Arrays

```
var favFoods ["Pickles", "Onions", "Carrots"]
```

Arrays

Arrays!



- ***Arrays* are the simplest data structure.**
- Javascript includes it natively.
- In most languages, arrays do not allow mixing of types.
- In most languages, arrays are not extendable. (They are fixed sizes)

```
var averageTemp = [];
averageTemp[0] = 31.9;
averageTemp[1] = 35.3;
averageTemp[2] = 42.4;
averageTemp[3] = 52;
averageTemp[4] = 60.8;
```

Arrays in Javascript

- In most languages (non-Javascript), arrays are **immutable** – meaning that upon declaration, the length of the array is fixed.

- With Javascript, we can easily add elements using the **.push method().**

<u>Question for You</u>

.push adds elements to which side of the array?

31.9	35.3	42.4	42.4	60.8
[0]	[1]	[2]	[3]	[4]

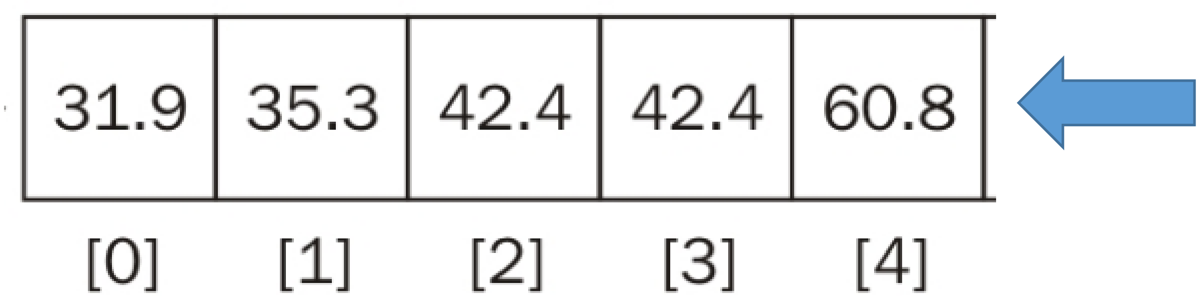
Arrays in Javascript

- In most languages (non-Javascript), arrays are **immutable** – meaning that upon declaration, the length of the array is fixed.

- With Javascript, we can easily add elements using the **.push method().**

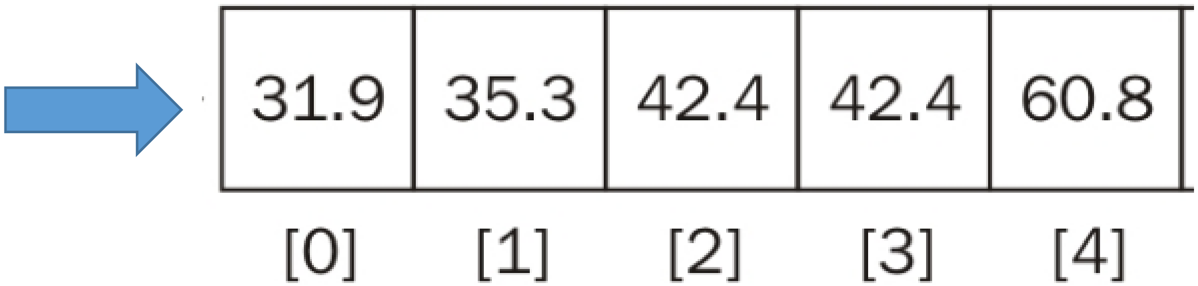
<u>Question for You</u>

.push adds elements to which side of the array?



2nd Question for You

How can we add an element to the beginning of the array?



If you finish early, implement it yourself.

(i.e. Don’t use the in-built method).

Arrays in Javascript

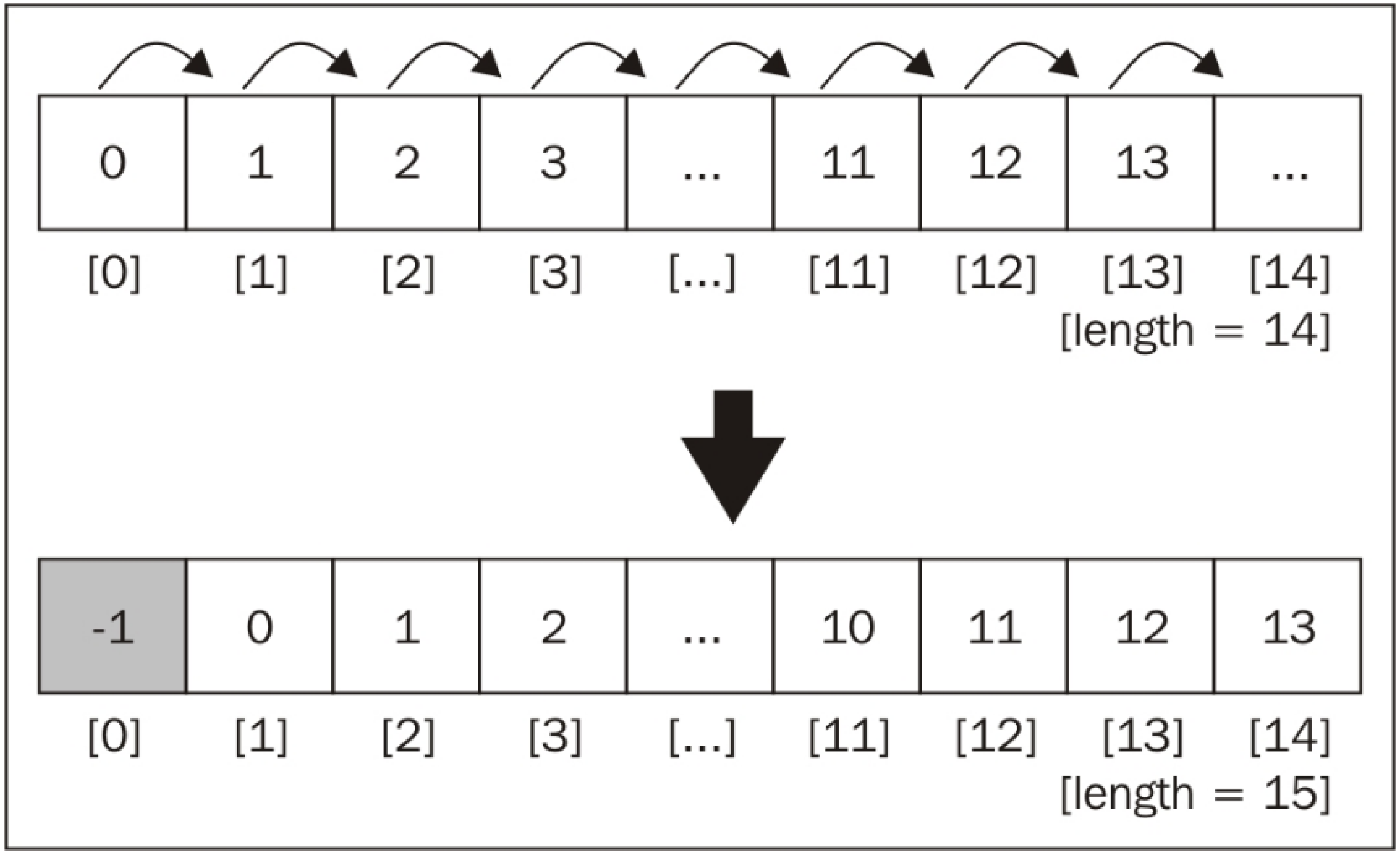
Unshift Method

```
myArray.unshift(1);
```

What's really happening…

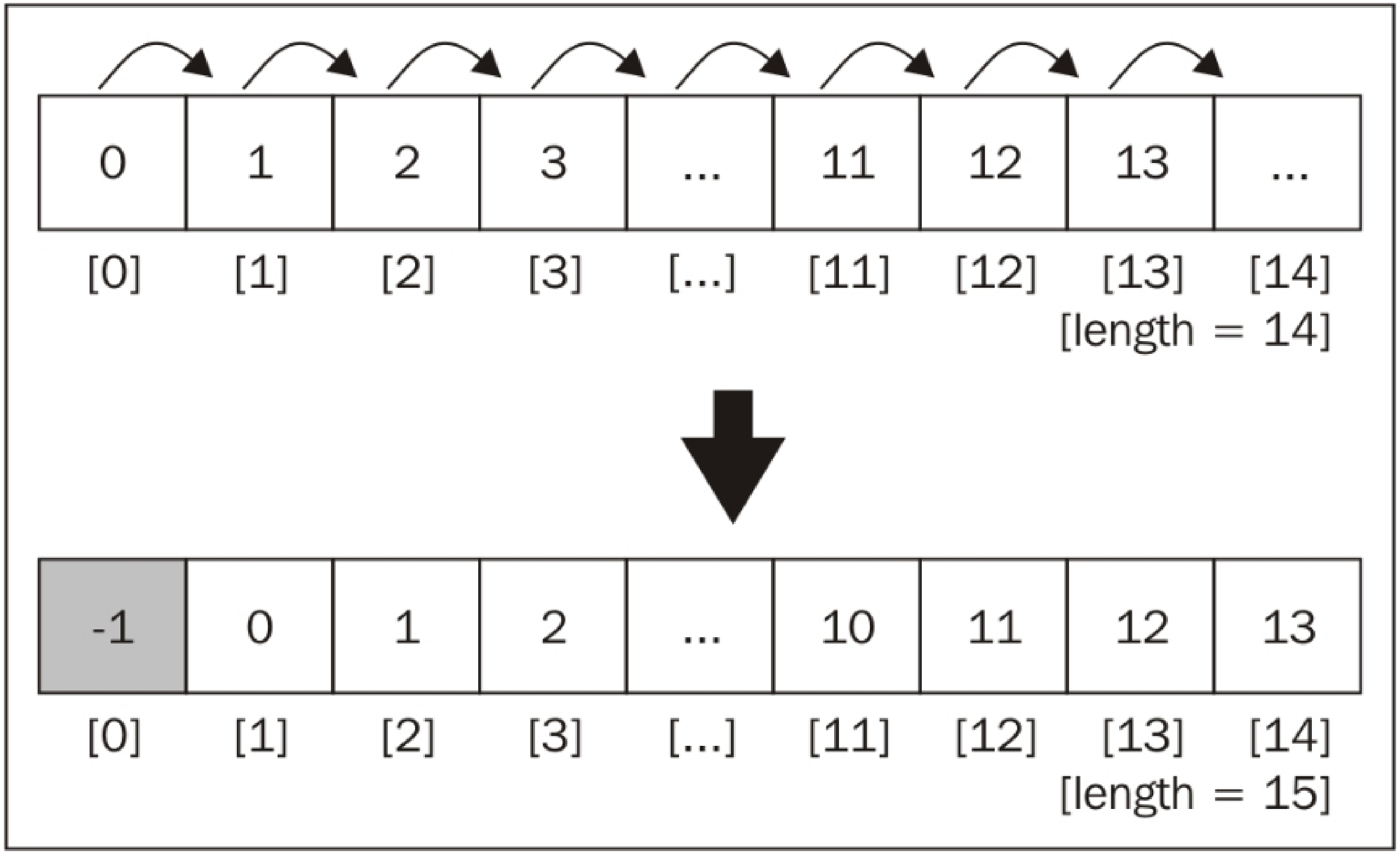
```
for (var i=myArray.length; i>=0; i--){
  myArray[i] = myArray[i-1];
}
myArray[0] = -1;
```

An inefficiency emerges!



An inefficiency emerges!

We'll come back to this.



Stacks / Queues

Going forward, treat each of the following data structures as <u>concepts.</u>

These are paradigmatic ways of organizing data that are commonly seen in code.

Stacks

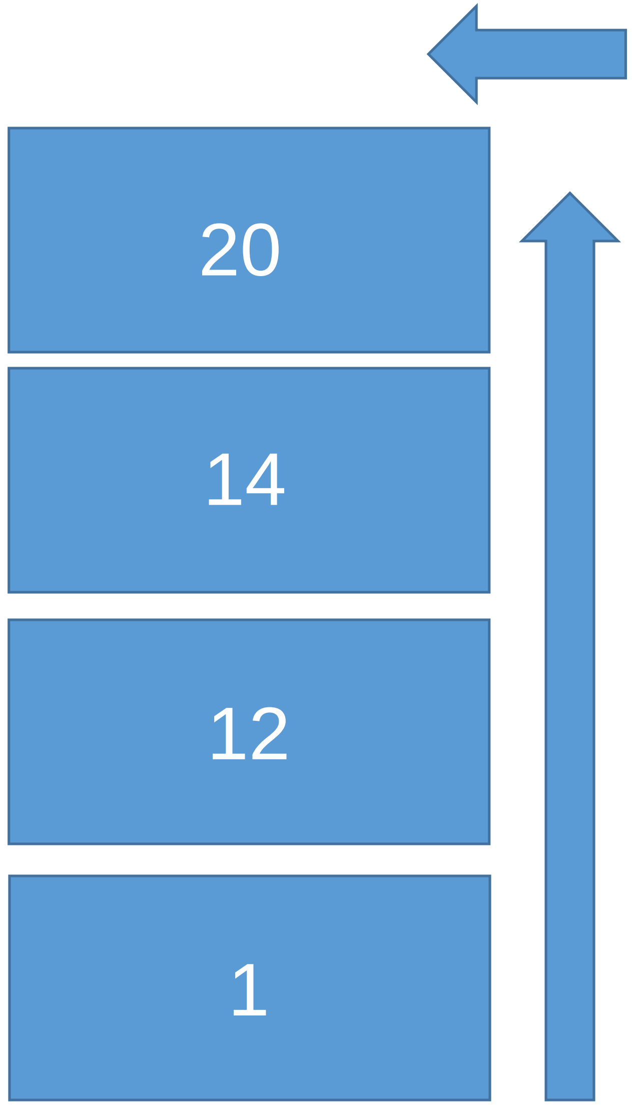
Stacks are another common data structure.

- They are similar to arrays in that they are a sequenced order of numbers.

- The difference is they **only allow access to the top element.**

- These data structures obey **"LIFO" (Last-in-first-out).** This means that new elements are placed at the top and removed from the top.

- *Stacks are an **abstraction** for how data can be arranged.*

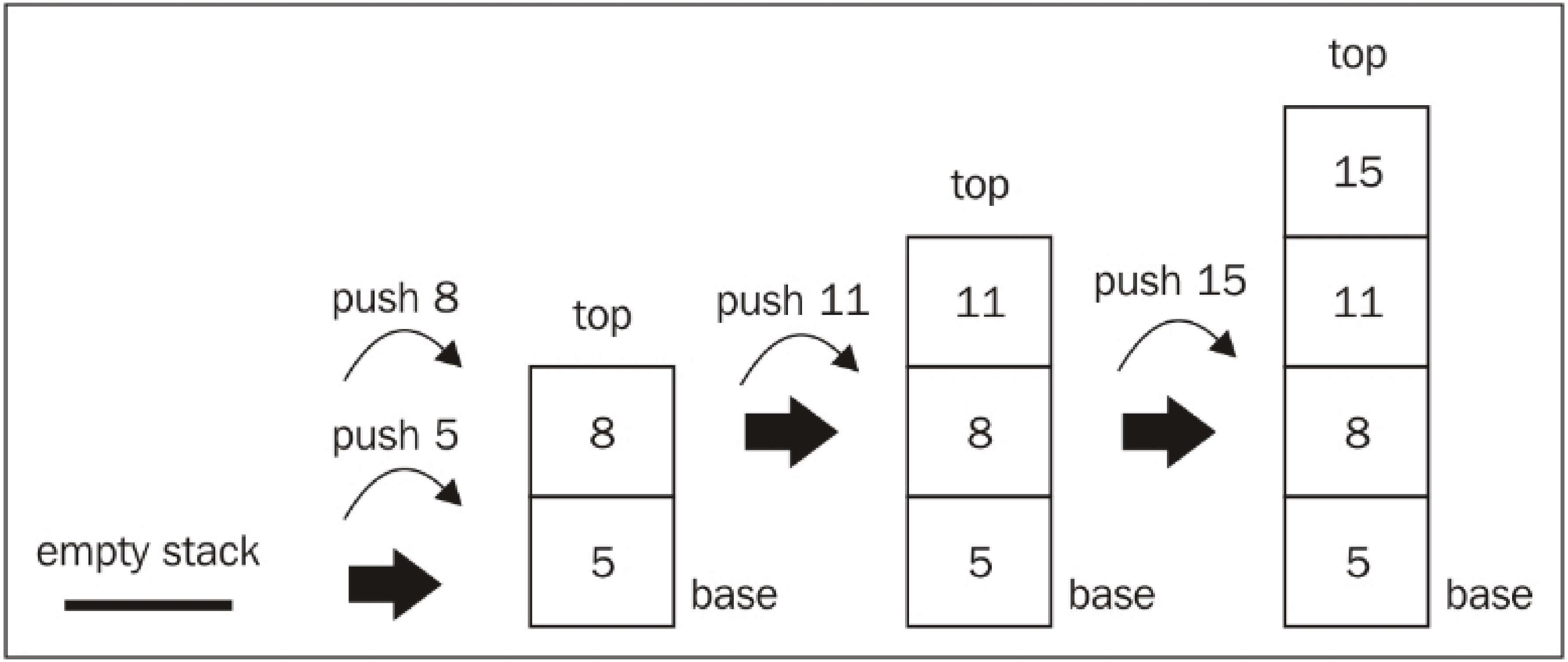


Stacks

Stacks are another common data structure.

- They are similar to arrays in that they are a sequenced order of numbers.
- The difference is they **<u>only allow access to the top element.</u>**
- These data structures obey **<u>"LIFO" (Last-in-first-out)</u>.** This means that new elements are placed at the top and removed from the top.
- <u>*Stacks are an **abstraction** for how data can be arranged.*</u>

Stacks



Last in First Out:

Items added to the top. Removed from the top

Stacks – In Code

```
class Stack {

  constructor () {
    this.items = [];
  }

  // Push, Pop, Peek
  push(element){
    this.items.push(element);
  }

  pop(element){
    this.items.pop();
  }

  peek(){
    return this.items[this.items.length-1];
  }

  isEmpty(){
    return this.items.length;
  }

  clear(){
    this.items = [];
  }

}
```

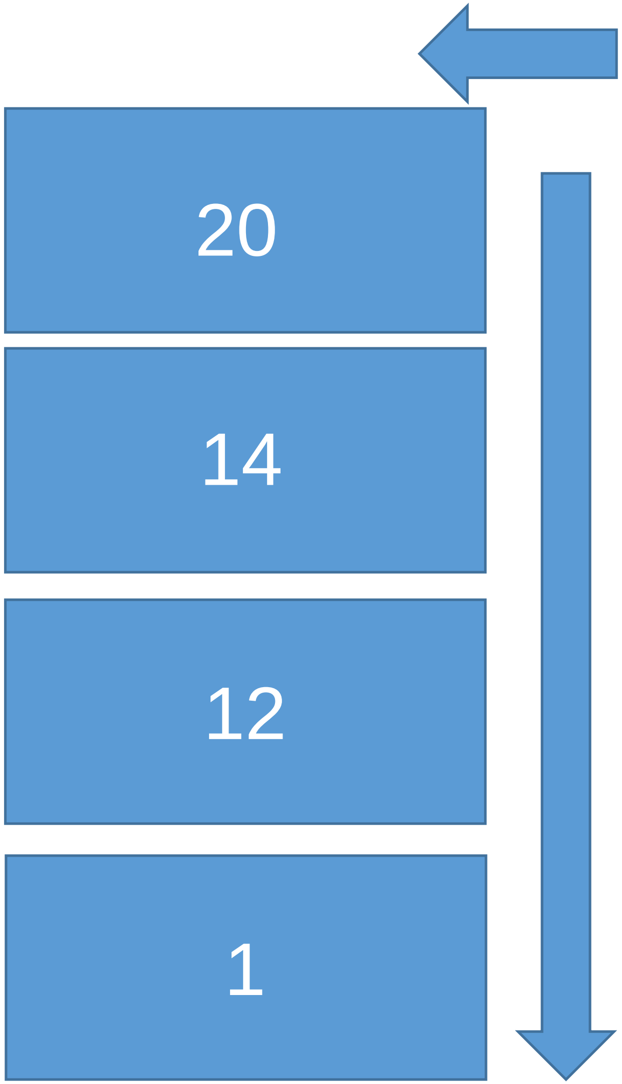
- “Stacks” aren’t supported natively in Javascript.
- To utilize this structure, one needs to create the class themselves.
- Once you’ve created a class you can create and utilize these structures in your code.

```
// Creates an instance of the Stack
var newStack = new Stack()

// Starts running methods
newStack.push(1);
newStack.push(2);
newStack.push(4);

console.log(newStack.peek());
```

Queue

Queues are another common data structure.

- They are similar to arrays in that they are a sequenced order of numbers.
- The difference is they **only allow access to the first element.**
- These data structures obey **“FIFO” (First-in-first-out).** This means that new elements are placed at the “back” but that the “first” element is removed from the front.
- *Queue are an **abstraction** for how data can be arranged.*



Queue

Queues are best remembered as similar to a movie queue. The first one in line is the first one to enter (or exit).

Queue – In Code

```
class Stack {

  constructor () {
    this.items = [];
  }

  // Push, Pop, Peek
  push(element){
    this.items.push(element);
  }

  pop(element){
    this.items.pop();
  }

  peek(){
    return this.items[this.items.length-1];
  }

  isEmpty(){
    return this.items.length;
  }

  clear(){
    this.items = [];
  }

}
```

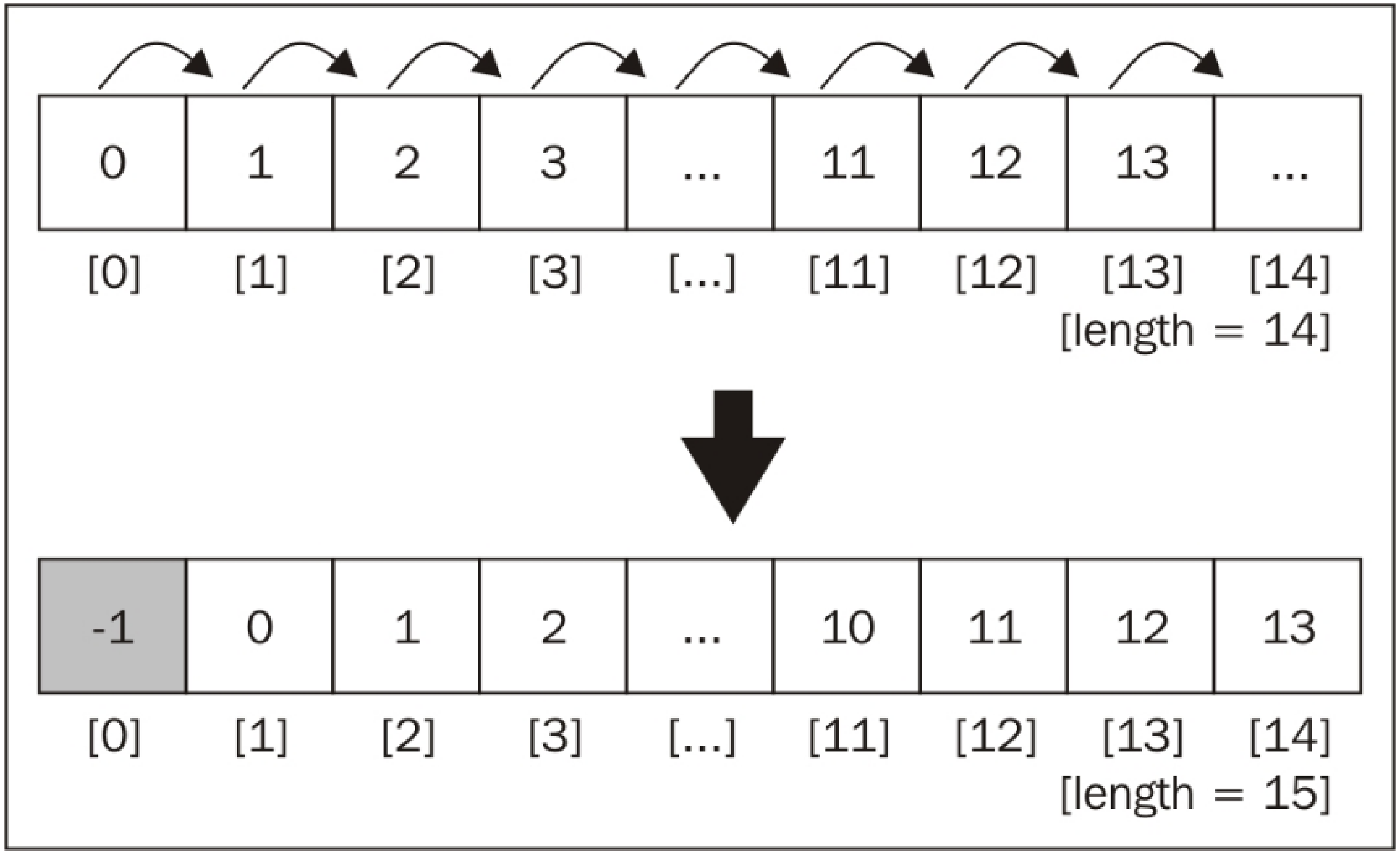
- "Queues" aren't supported natively in Javascript.
- Again, this means we need to create our own for use.
- Queues provide two common methods: **enqueue** and **dequeue**.

```
// Creates an instance of the Stack
var newStack = new Stack()

// Starts running methods
newStack.push(1);
newStack.push(2);
newStack.push(4);

console.log(newStack.peek());
```

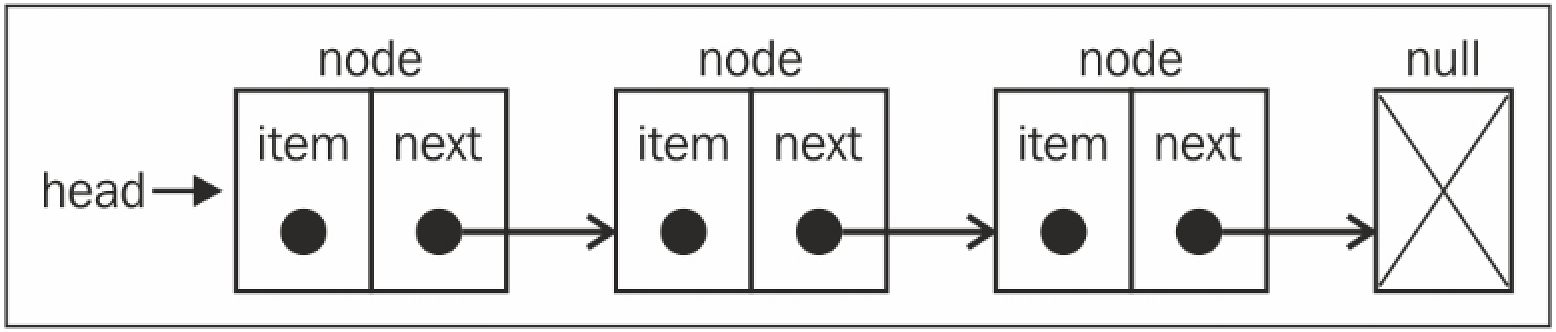
Linked Lists

An inefficiency emerges!

We’ll come back to this.



Linked List



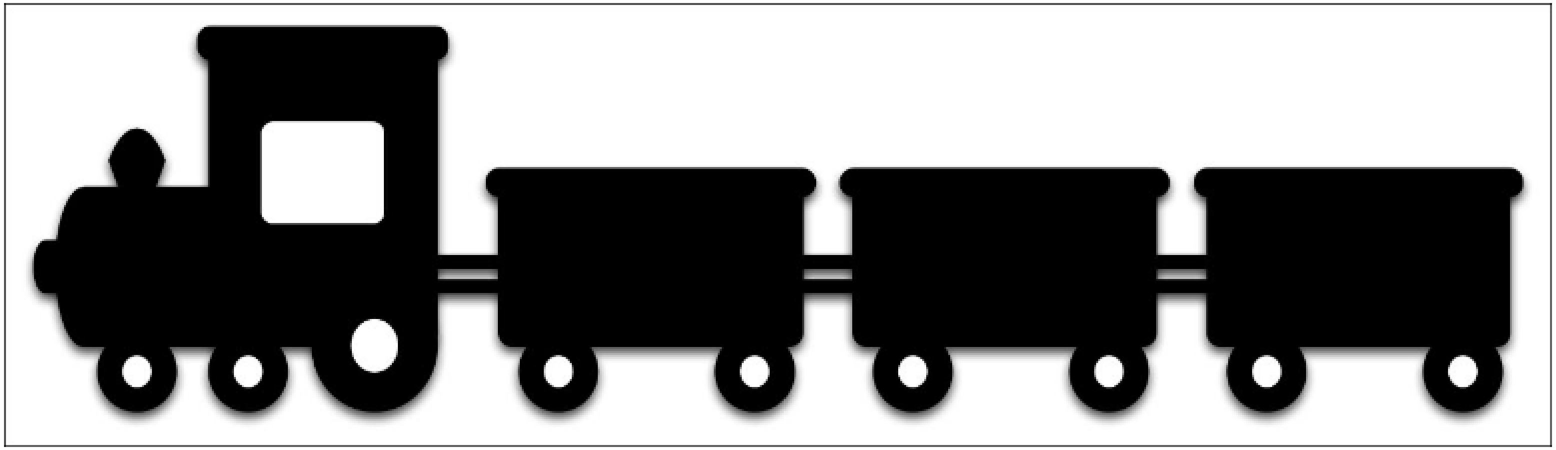
- **Linked Lists** are data structures in which each element of the list is sequentially joined to the next element.

- The major difference is that the list elements are not stored **contiguously** in memory (i.e. they fall in different memory slots).

- These linked lists keep track of the position of elements using **pointers** which explicitly point to the "connected item".

- Each element (**called nodes)** track both the item and the "next item's" position.

Linked List



- **Linked Lists** are like trains.

- Each car of the train not only knows its own position – but it also knows the position of the train in front of it.

Linked List – In Code

```
class Node {
  constructor(data, next) {
    this.data = data;
    this.next = next;
  }

  getData() {
    return this.data;
  }

  setData(data) {
    this.data = data;
  }

  getNext() {
    return this.next;
  }

  setNext(next) {
    this.next = next;
  }
}

class LinkedList {
  constructor(dataArray) {
    this.first = new Node();

    var counter = 0;
    if (dataArray) {
      var actual = this.first;
      for (var data of dataArray) {
        var newNode = new Node(data);
        actual.setNext(newNode);
```

- JS does not include Linked Lists natively
- But when you need one…
- Plenty of implementations are available online.
- http://codepen.io/gben/pen/ZGLava

For the Lazy… (Myself included)

linkedlist public

Array like linked list with iterator

LinkedList is a data structure which implements an array friendly interface

Class Methods

```
LinkedList.prototype.push(data)
LinkedList.prototype.pop()
LinkedList.prototype.unshift(data)
LinkedList.prototype.shift()
LinkedList.prototype.next()
LinkedList.prototype.unshiftCurrent()
LinkedList.prototype.removeCurrent()
LinkedList.prototype.resetCursor()
```

What happens when npr
together to share with or

`npm install li`

how? learn more

kilianc published 4

Pulse Check…

You Be the Teacher

To the person, next to you, explain each of the following concepts:

1. What is a data structure?
2. What does FIFO and LIFO stand for and mean?
3. What is a Stack?
4. What is a Queue?
5. What is a Linked List?
6. How are they each different from arrays?
7. What is one disadvantage of an array?
8. Most important question: Why are we doing all this again?

Dictionaries (Maps)

Dictionaries (Maps) **** (Actually Useful) ****

Dictionaries are an incredibly important data structure..

- In fact, they address a common situation you've faced in this class.

```
var myPets = {
    cat: "Mr. Hyena",
    lizard: "Mr. Big Big",
    goat: "Wolf Who Ate Wall Street",
    pigeon: "Joan"
}
```

How would you print all the pet names?

Dictionaries (Maps) **** (Actually Useful) ****

Dictionaries are an incredibly important data structure..

- In fact, they address a common situation you've faced in this class.

```
var myPets = {

    cat: "Mr. Hyena",

    lizard: "Mr. Big Big",

    goat: "Wolf Who Ate Wall Street",

    pigeon: "Joan"

}
```

How would you print all the pet names?

Arrays don't solve the problem either….

```
var myPetAnimals = ["cat", "lizard", "goat", "pigeon"]
var myPetNames = ["Mr. Hyena", "Mr. Big Big", "Wolf Who Ate Wall Street", "Joan"]
```

Dictionaries (Maps) **** (Actually Useful) ****

The solution is to use a **dictionary (map).**

- In a way, dictionaries serve as a hybrid between objects and arrays.
- They can be iterated over like arrays.
- They have key, value pairs like objects.
- Aaaand, it’s included in the latest version of Javascript (ES6).

```
var map = new Map();

map.set("cat", "Mr. Hyena");
map.set("lizard", "Mr. Big Big");
map.set("goat", "Wolf Who Ate Wall Street");
map.set("pigeon", "Joan");

console.log(map.keys());
console.log(map.values());
console.log(map.get("pigeon"));
```

BIG DEAL!

Dictionaries (Maps) **** (Actually Useful) ****

Learn more about Dictionaries (Maps) in JS:

Map

SEE ALSO

Standard built-in objects

`Map`

- Properties
  - `Map.prototype`
  - `Map.prototype.size`
  - `Map.prototype[@@toStringTag]`
  - `get Map[@@species]`
- Methods
  - `Map.prototype.clear()`
  - `Map.prototype.delete()`

The **`Map`** object is a simple key/value map. Any value (both objects and primitive values) may be used as either a key or a value.

Syntax

```
new Map([iterable])
```

Parameters

`iterable`

Iterable is an Array or other iterable object whose elements are key-value pairs (2-element Arrays). Each key-value pair is added to the new Map. `null` is treated as `undefined`.

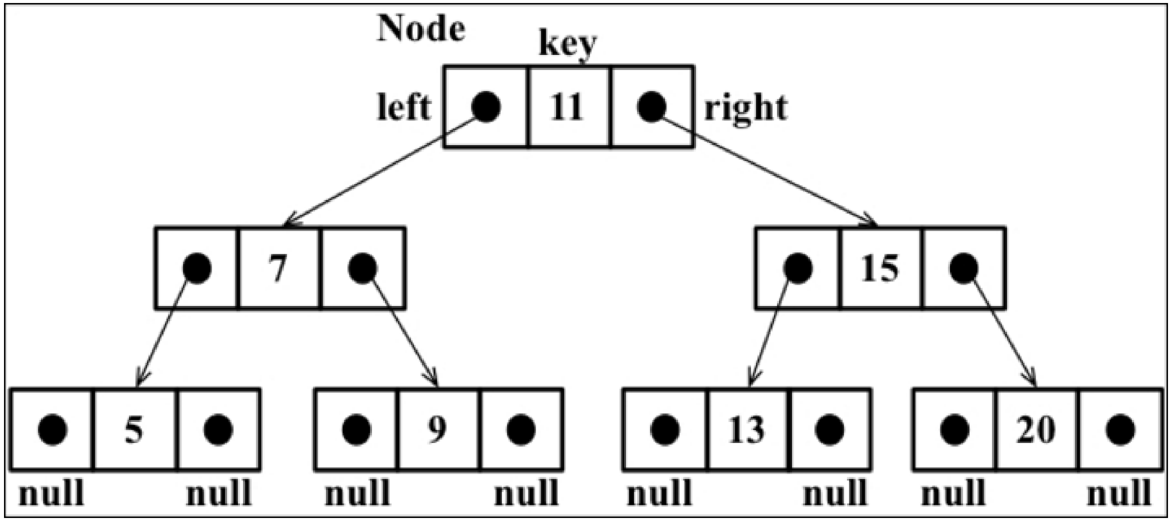
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Trees

Trees

Trees are a favorite data structure for computer scientists

- Trees are a non-sequential data structure made of **parent-child** relationships.
- The top node of a tree is the **root.**
- Trees have **internal nodes and external nodes**
- Each node has **ancestors and descendants**



Kind of like a linkedlist

Binary Trees

Binary Trees / Binary Search Trees (BST) are particularly useful

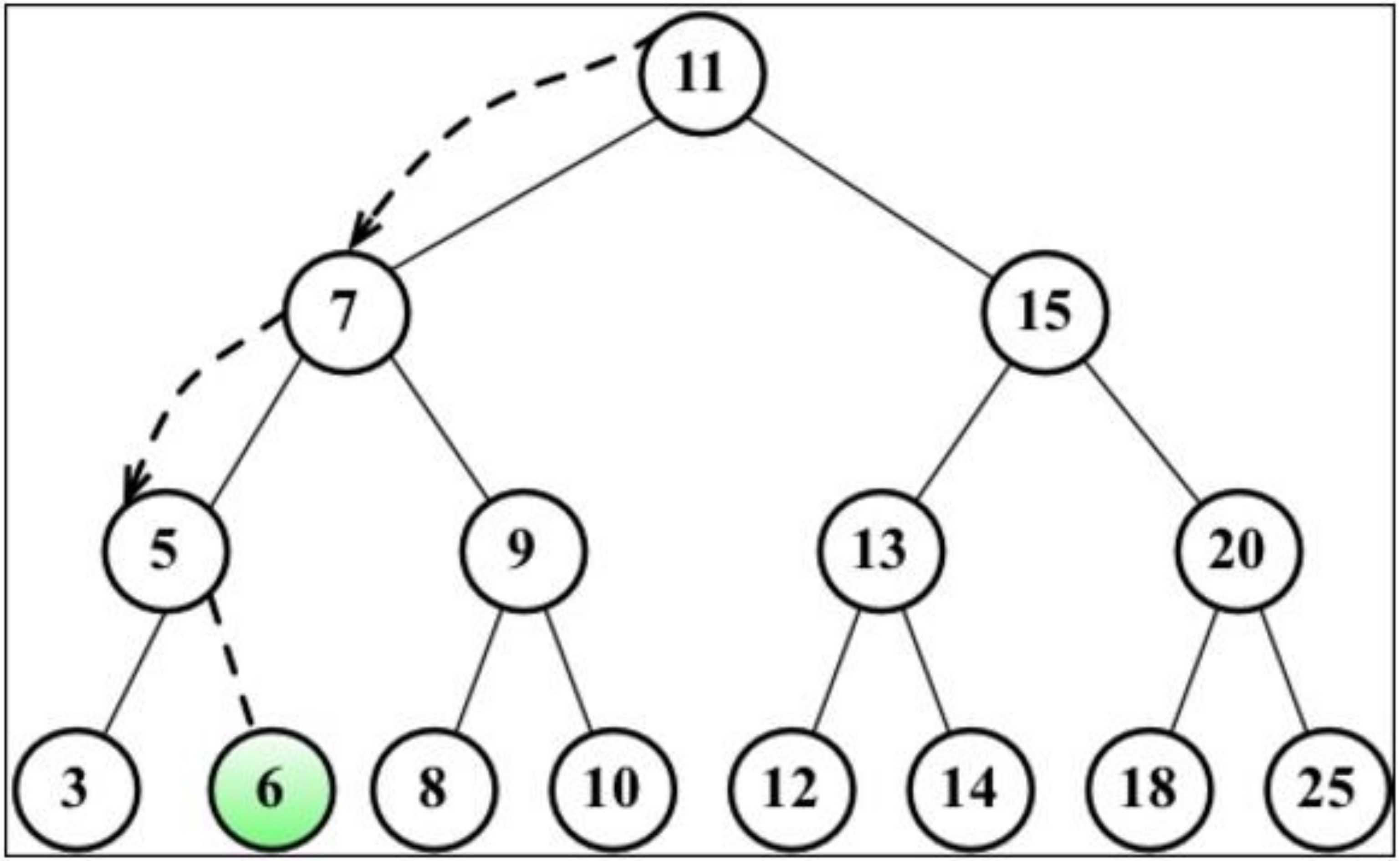
- In a **Binary Tree**, nodes have **two children** at most. One on left and on right.

- In a **Binary Search Tree**:

  - Left-hand side is lesser number; right-hand side is the larger

  - Paradigm makes it easy to insert, search, and delete from tree

Binary Trees



- Binary search trees are extremely efficient for searching.

Binary Search Trees

binary-search-tree public

Different binary search tree implementations, including a self-balancing one (AVL)

Binary search trees for Node.js

Two implementations of binary search tree: basic and AVL (a kind of self-balancing binmary search tree). I wrote this module primarily to store indexes for NeDB (a javascript dependency-less database).

Installation and tests

Package name is `binary-search-tree`.

```
npm install binary-search-tree --save

make test
```

Usage

The API mainly provides 3 functions: `insert`, `search` and `delete`. If you do not create a unique-type binary search tree, you can store multiple pieces of data for the same key. Doing so with a unique-type BST will result in an error being thrown. Data is always returned as an array, and you can delete all data relating to a given key, or just one piece of data.

`npm i binary-search-tree`

how? learn more

louischatriot published 4 months ago

0.2.6 is the latest of 15 releases

github.com/louischatriot/node-binary-search-tree

MIT

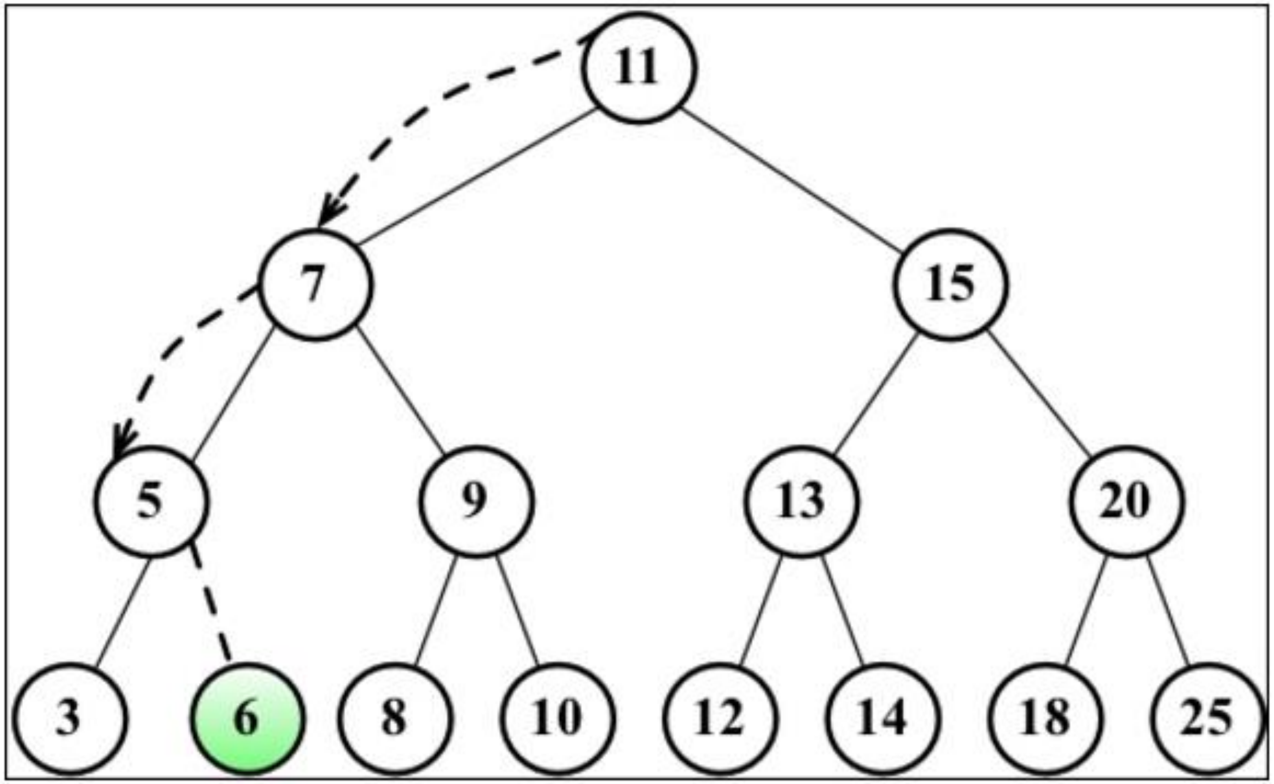
Collaborators

https://www.npmjs.com/package/binary-search-tree

Let's Build this!

- Take a few moments to build a binary search tree with those around you. As a suggestion, implement the following tree.

- Then run a search for any number in the tree.



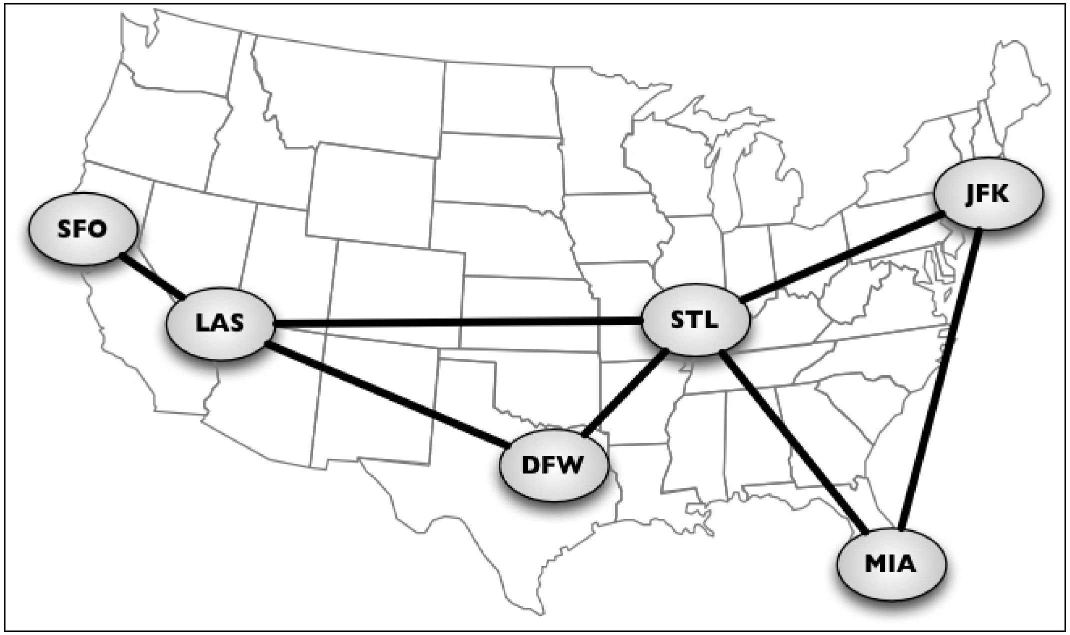
Graphs

Graphs

<u>Graphs are extremely powerful and increasingly common structures.</u>

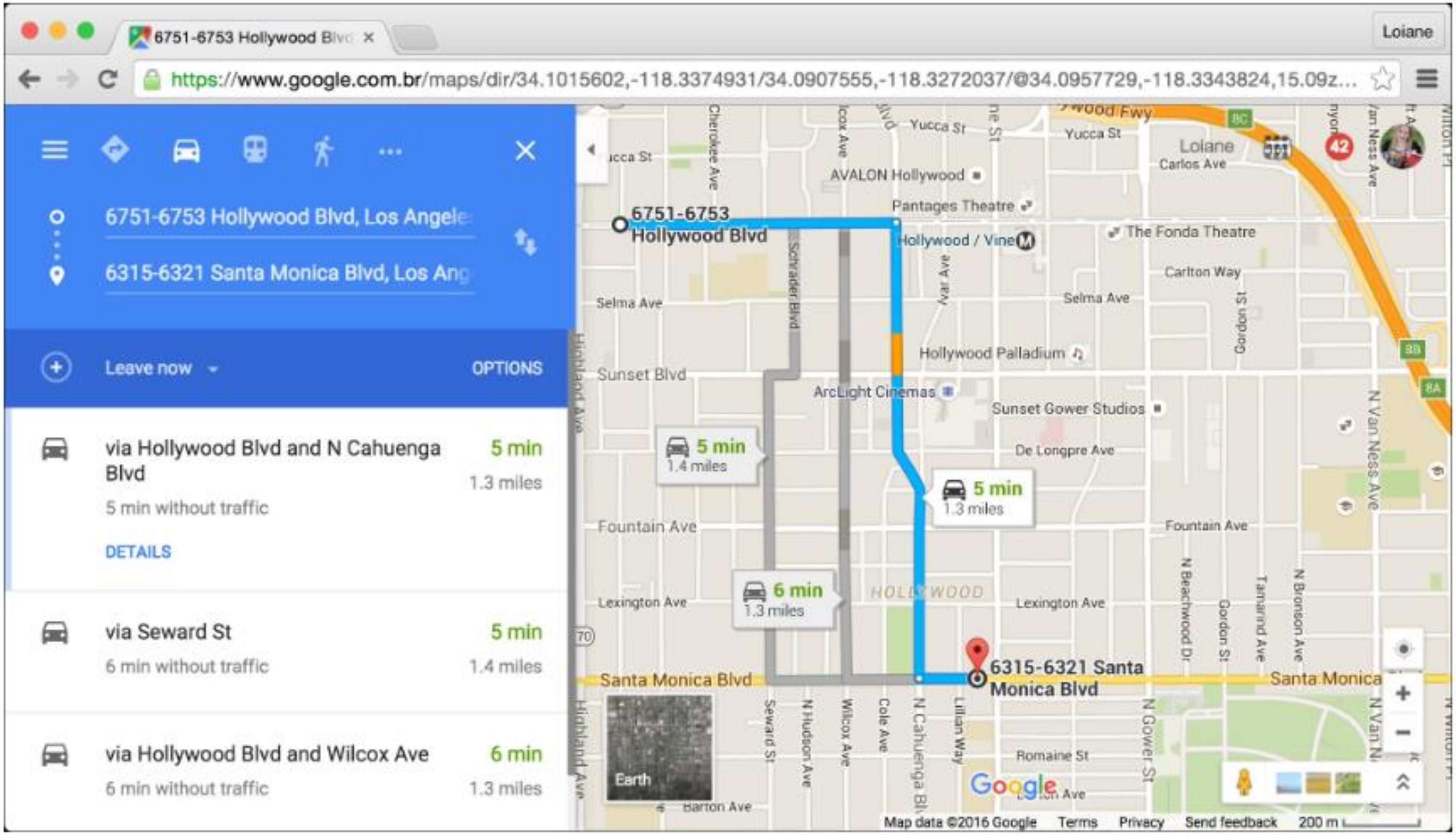
- Graphs are abstract models of a network structure. They are a set of **nodes (or vertices)** connected by **edges.**

- They are the essence of social networks and geographic maps.



The math gets ridiculously scary with this stuff…

Graphs



But through graphs and "shortest-path" algorithms we can build map applications like the ones found on Google Maps

Back to Projects!

Questions