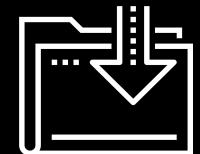




# Big O and Data Structures

Web Development Boot Camp  
Lesson 22.3



# Outline

---



Project Check-In



Computer Science Context



Big O Notation



Data Structures



Arrays



Hash Tables



Stacks/Queues



Sets



Linked Lists



Binary Trees & Binary Search Trees



Dictionaries



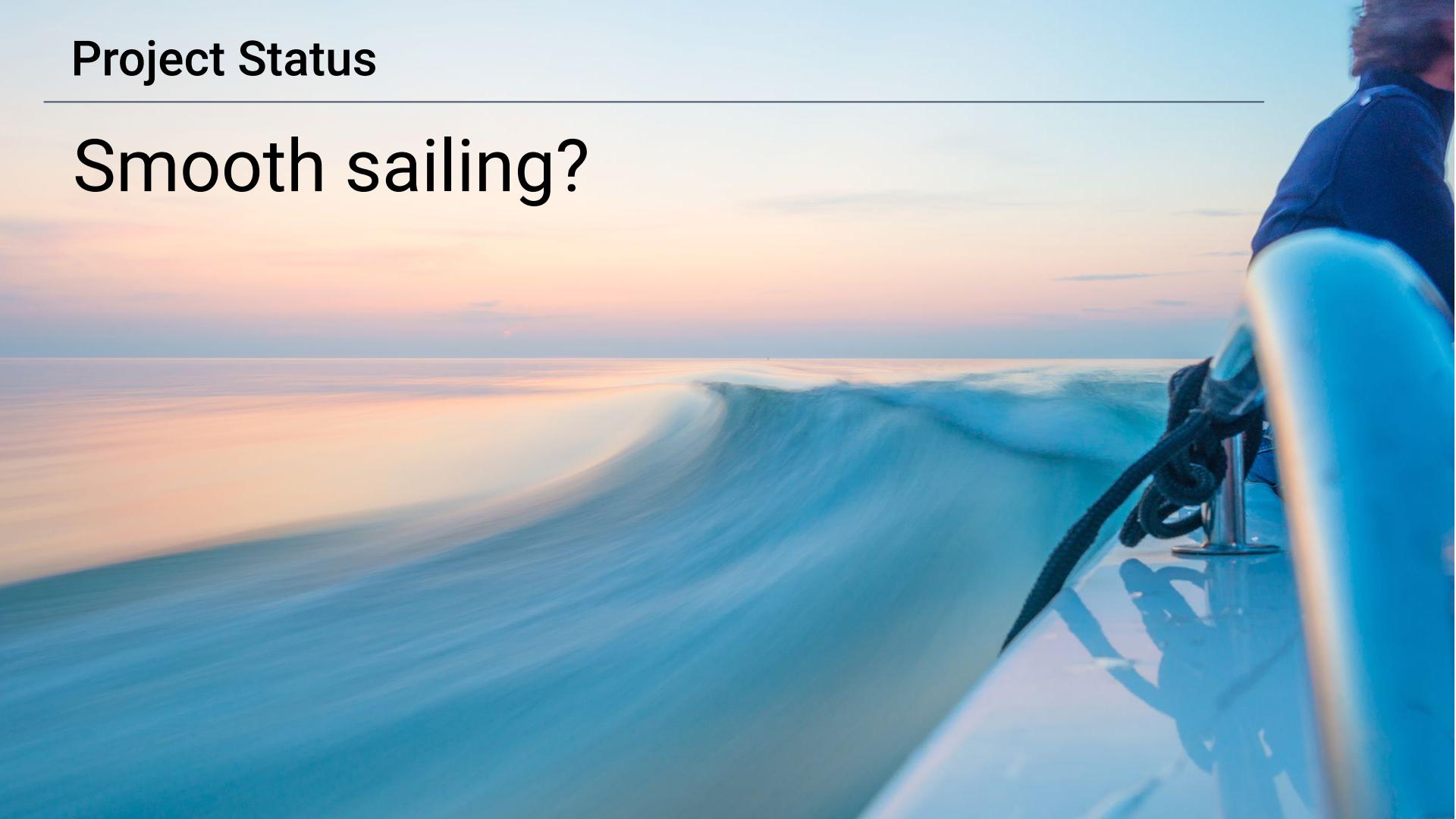
Graphs

# Project Check-In

# Project Status

---

## Smooth sailing?



# Project Check-In: Deliverable #1 is Due by the End of Today's Class

---

Please send the following to your Instructor and TAs:



Overview of intended application



Detailed screen-by-screen UI layouts with annotations



Breakdown of group member roles



Screenshot of project management tool

Submit by the end of the day!



# Computer Science Context



**Welcome to...**  
Computer Science Fundamentals!

# Remember...

---

## Computer science fundamentals



Fundamentals aren't the "easy" computer science stuff.



Rather, they are 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

---

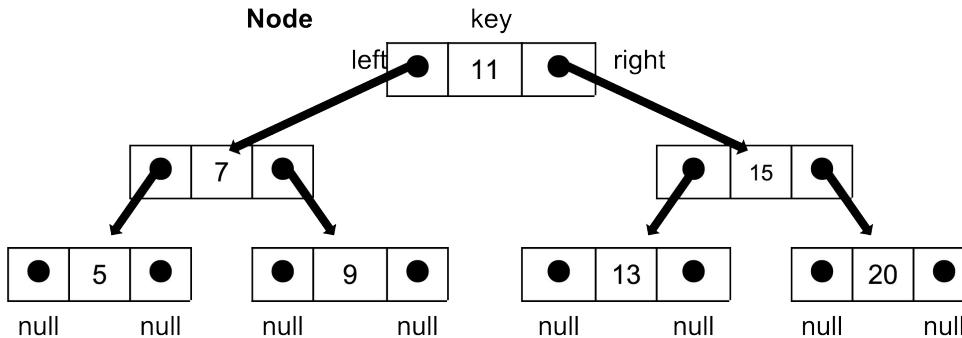
Remember this stuff? Yeah, me neither.

Stokes Theorem

$$\oint_C \vec{F} \cdot d\vec{r} = \iint_S \text{curl } \vec{F} \cdot d\vec{A}$$


S smooth oriented surface  
C piecewise smooth oriented boundary  
 $\vec{F}$  smooth vectorfield defined on S and C.

# It Gets Hairy and Scary



```

function divideBy2(decNumber) {
  var remStack = new Stack(),
      rem,
      binaryString = '';

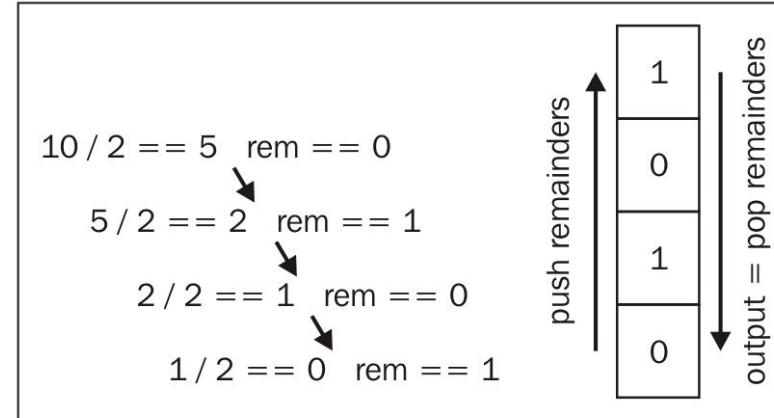
  while (decNumber > 0){ //1
    rem = Math.floor(decNumber % 2); //2
    remStack.push(rem); //3
    decNumber = Math.floor(decNumber / 2); //4
  }

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

  return binaryString;
}
  
```

```

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
}
  
```





## **Be Wary of Imposter Syndrome**

Don't let the hard stuff scare you.

# Why Cover This?

---

01

These concepts sometimes appear in coding interviews.

02

When inheriting large codebases, you might be tasked to optimize code efficiency.

03

The computational challenges in this lesson force you to deepen your understanding.

# Bottom Line

---

My goal is to give you the terminology and the concepts.



I want to give you enough insight so you can understand the context of interview questions that come your way.

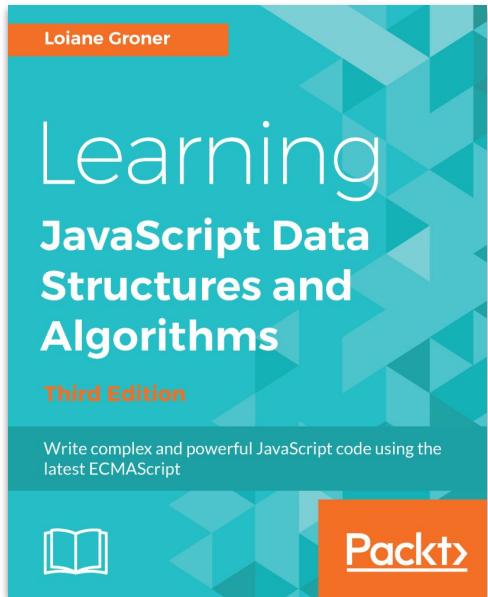


And to encourage those of you who are into math to take a second look!

# Going Deep

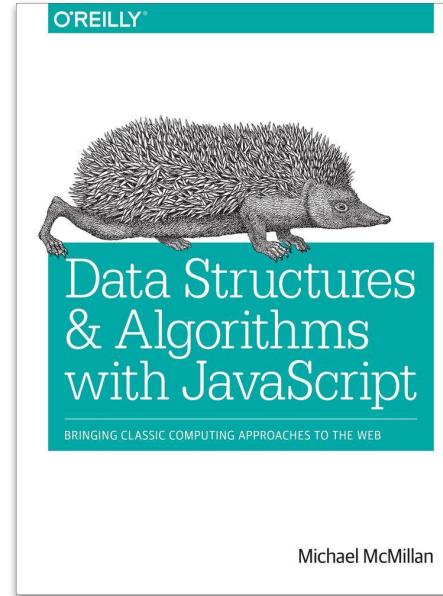
---

For those who dare dive deeper



**Learning JavaScript Data Structures and Algorithms—Third Edition**  
by **Loiane Groner**

Publisher: Packt Publishing  
Release Date: April 2018



**Data Structures and Algorithms with JavaScript**  
by **Michael McMillan**  
Publisher: O'Reilly Media, Inc.  
Release Date: March 2014

# Efficiency

# What does “Efficient” Mean?

---

We talk a lot about efficiency. But what exactly does “efficient” mean?



Fewer Steps = Faster Code  
**Number of Steps ~ Efficiency**

More Steps = Less Efficient  
**Fewer Steps = More Efficient**

# 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?



Which has fewer instructions?

```
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];
}
```



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_items` isn't bad.

# Time Complexity

# Quantifying Efficiency

---

**head always executes one instruction...**

# Quantifying Efficiency

---

...no matter how long the array is.

# Quantifying Efficiency

---

head takes the same amount of time on any input

```
// 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) );
```

# Quantifying Efficiency

---

`list_items` needs  $n$  instructions

# Quantifying Efficiency

---

One `console.log` per item:

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

# 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

# Quantifying Efficiency

---

In other words...

# Quantifying Efficiency

---

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



**Time complexity** =  
the rate at which algorithm  
**slows** as input **grows**.

# Quantifying Efficiency

---

head is **always** one instruction.



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

# Quantifying Efficiency

---

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 **increases 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 ( $n$ ).



It is 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)

# Big O

---

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  
 $n \times \text{length} = n^2 \text{ time}$

# Big O

---

Running time grows  
as **square** of input.

# Big O

---

find\_duplicates ~  $O(n^2)$

“**Quadratic** time complexity”



Big O:  
**MAJOR INSIGHT!**

# Big O

---

2 nested

for

loops  $\sim O(n^2)$



# NOT COINCIDENCE!

# Big O

---

**3** nested for loops  $\sim O(n^3)$

*Etc.*

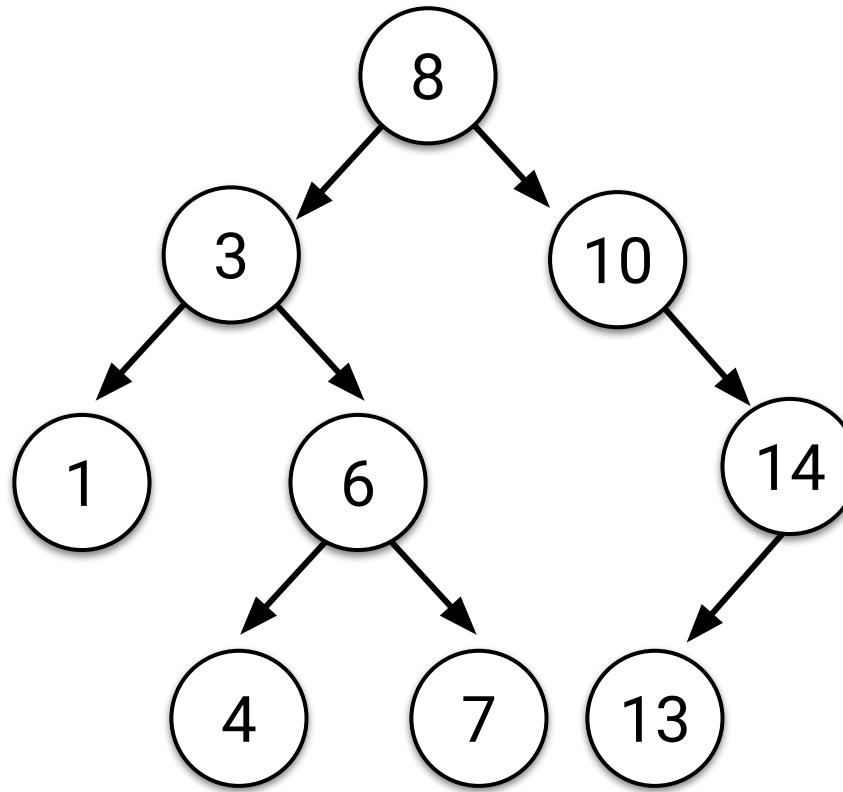


Big O:  
**One More...**

# Big O

---

How fast is binary search?



# Big O

---

Is it...



$O(1)$



$O(n)$



$O(n^2)$



Something else?

# Big O

---

Something else? Why?!



# Activity:

---

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

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



# Activity:

---

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

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

Answer: 3 Steps



# Big O

---

Add the digits 11–20. Repeat.

# Big O

---

Add the digits 11–20. Repeat.

Answer: 4 Steps (!)



**Much** faster than linear.

# Big O

---

$(\text{input size})^2 \sim 2x$  running time

$(\text{input size})^3 \sim 3x$  running time

*Etc.*

# Big O

---

This is called  $O(\lg n)$ .

# Big O

---

$\lg n$  = how many times do I divide  $n$   
by 2 to get to 1?

# Logarithm Example

---

What is  $\lg 8$ ?

# Logarithm Example

---

$$8 / 2 = 4 \text{ (1)}$$

$$4 / 2 = 2 \text{ (2)}$$

$$2 / 2 = 1 \text{ (3)}$$

# Logarithm Example

---

$$\lg 8 = 3$$

*But if this is confusing...*

## Logarithm Example

---

**Don't worry about it.**

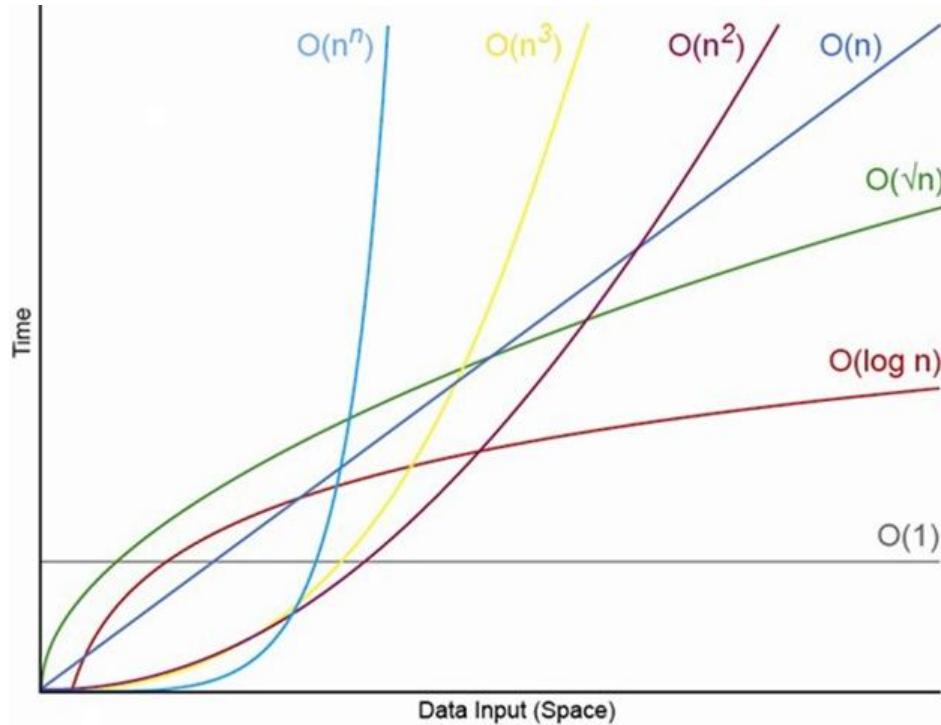
# Big O Review

---

<code>head</code> ~ $O(1)$	Grows like "1" (i.e., 2x input size -> 1x running time)
<code>list_items</code> ~ $O(n)$	Grows like " $n$ " (i.e., 2x input size -> 2x running time)
<code>find_duplicates</code> ~ $O(n^2)$	Grows like " $n^2$ " (i.e., 2x input size -> 4x running time)
<code>binary_search</code> ~ $O(\lg n)$	Grows like " $\lg n$ " (i.e., (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.

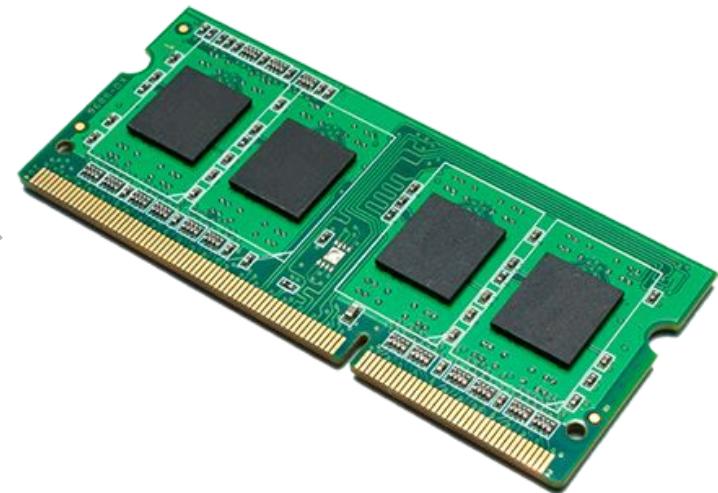
---

Code that we write gets saved in memory.

```
var name = Ahmed
```

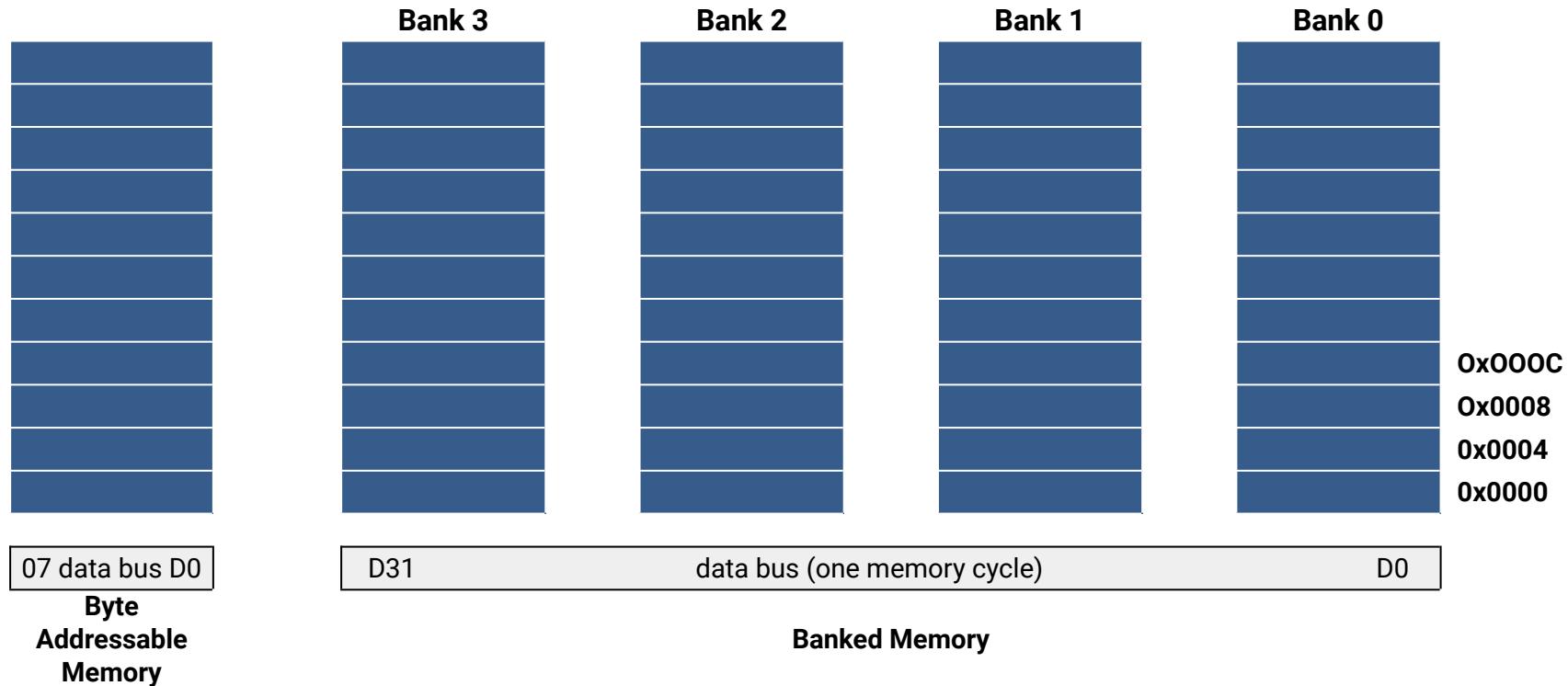
```
var age = 82
```

```
var isCool = true
```



# Different Ways to Save

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



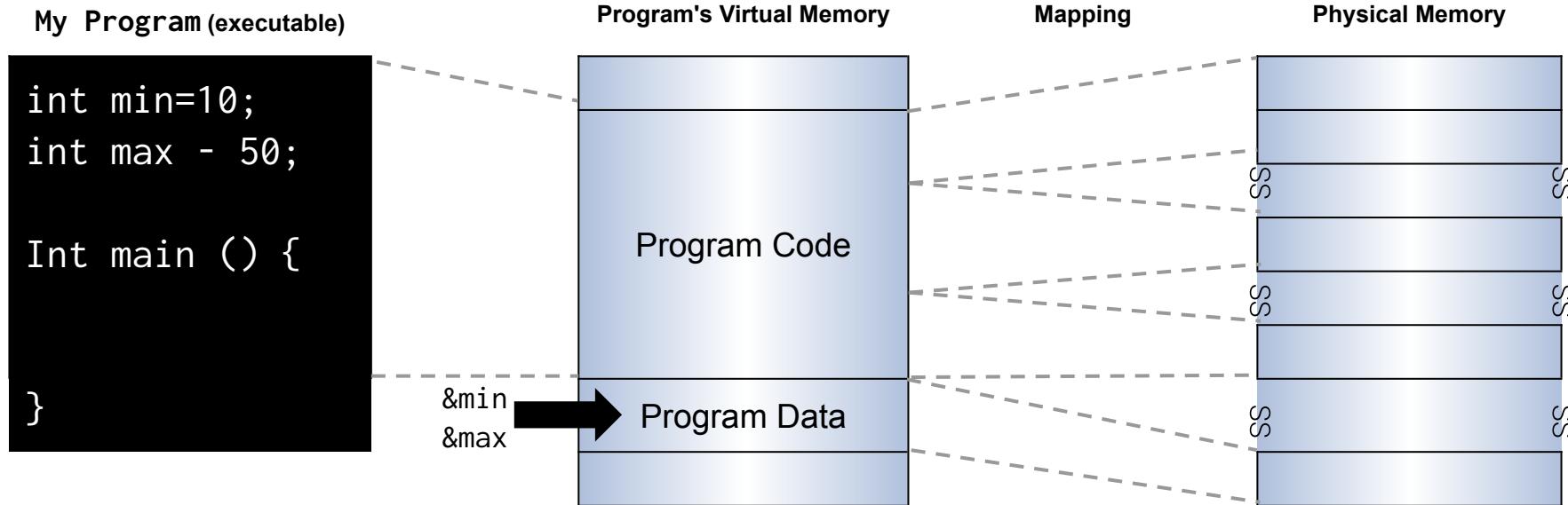
# Memory on My Mind



Our code as a whole takes some of these slots of memory.



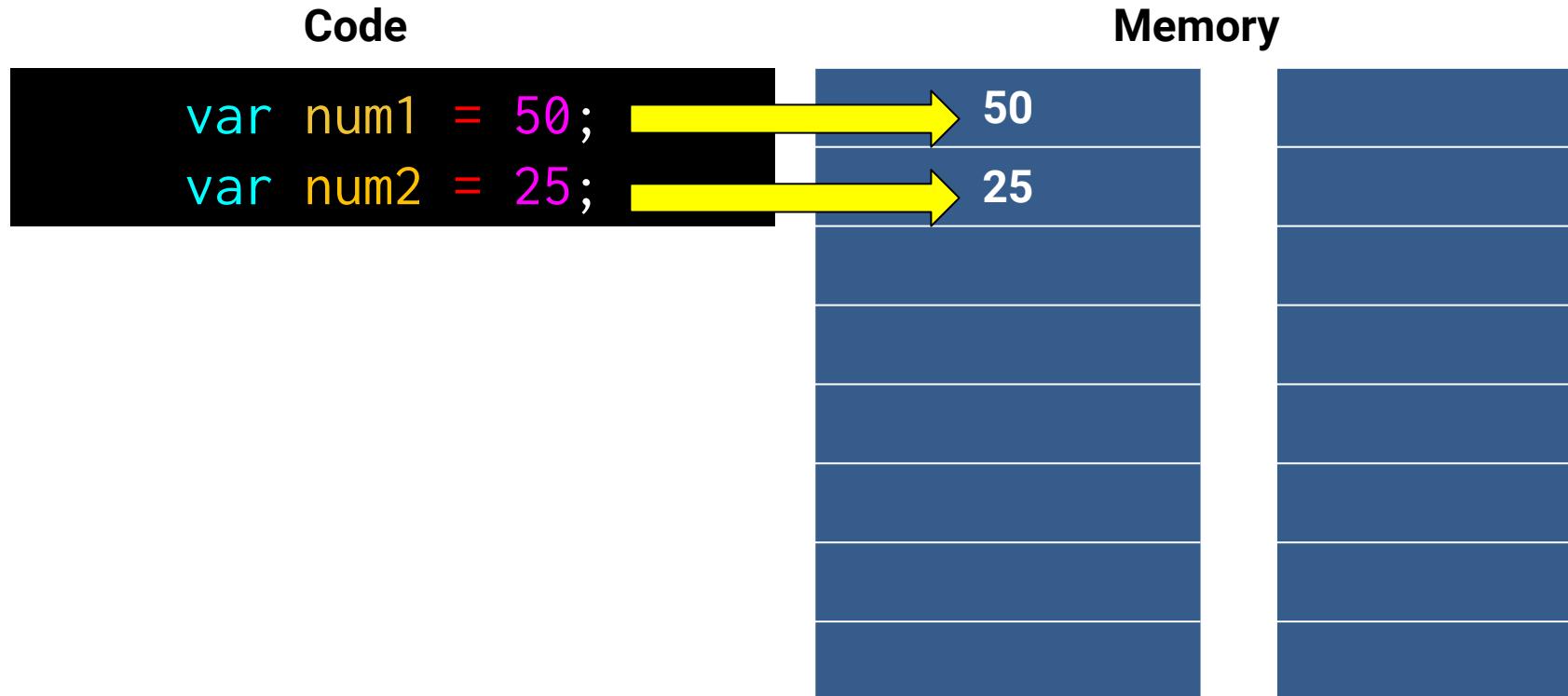
Our variable data itself also takes slots of memory.



# Saving to Memory

---

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



# Retrieving from Memory

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

Code

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

Memory

50  
25

```
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).

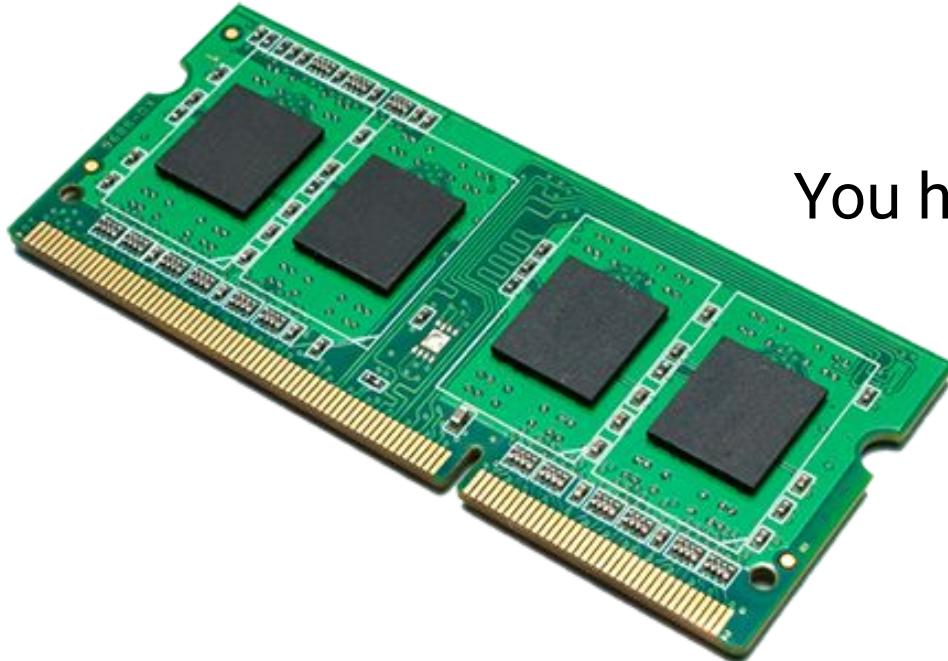


**Don't let the simplicity fool you!**

# Building Devices

---

Devices inherently have limited memory because of space requirements—making efficiency decisions critical.



You have 1 MB. Use it wisely!

# Retrieving from Memory

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

Code

```
var pet = {  
    type: "Mammal",  
    animal: "Cat"  
    name: "Sammy",  
    age: 24  
}
```

Memory





What is a  
**data structure?**



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



What is a  
**data structure?**



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



What is a  
**data structure?**



It is non-opinionated, in  
the sense that it is only  
responsible for holding  
the data.

# Data Structures?

---

## 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.)

averageTemp

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

```
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.



`.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.

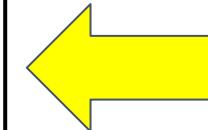


With JavaScript, we can easily add elements using the `.push()` method.



`.push` adds elements to which side of the array?

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

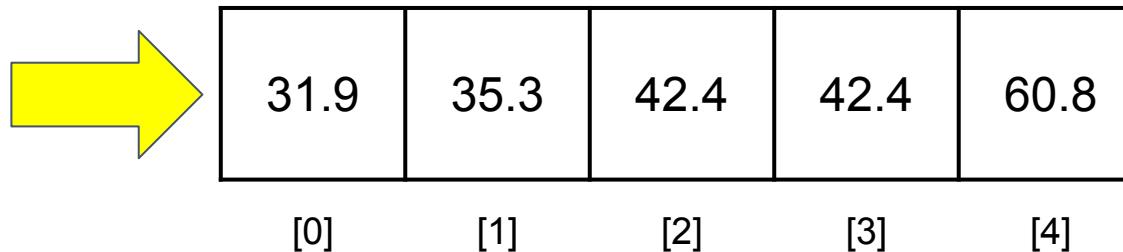


# Arrays in JavaScript

---



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



How would you solve this without a built-in Array method?

# Arrays in JavaScript

---

Unshift method:

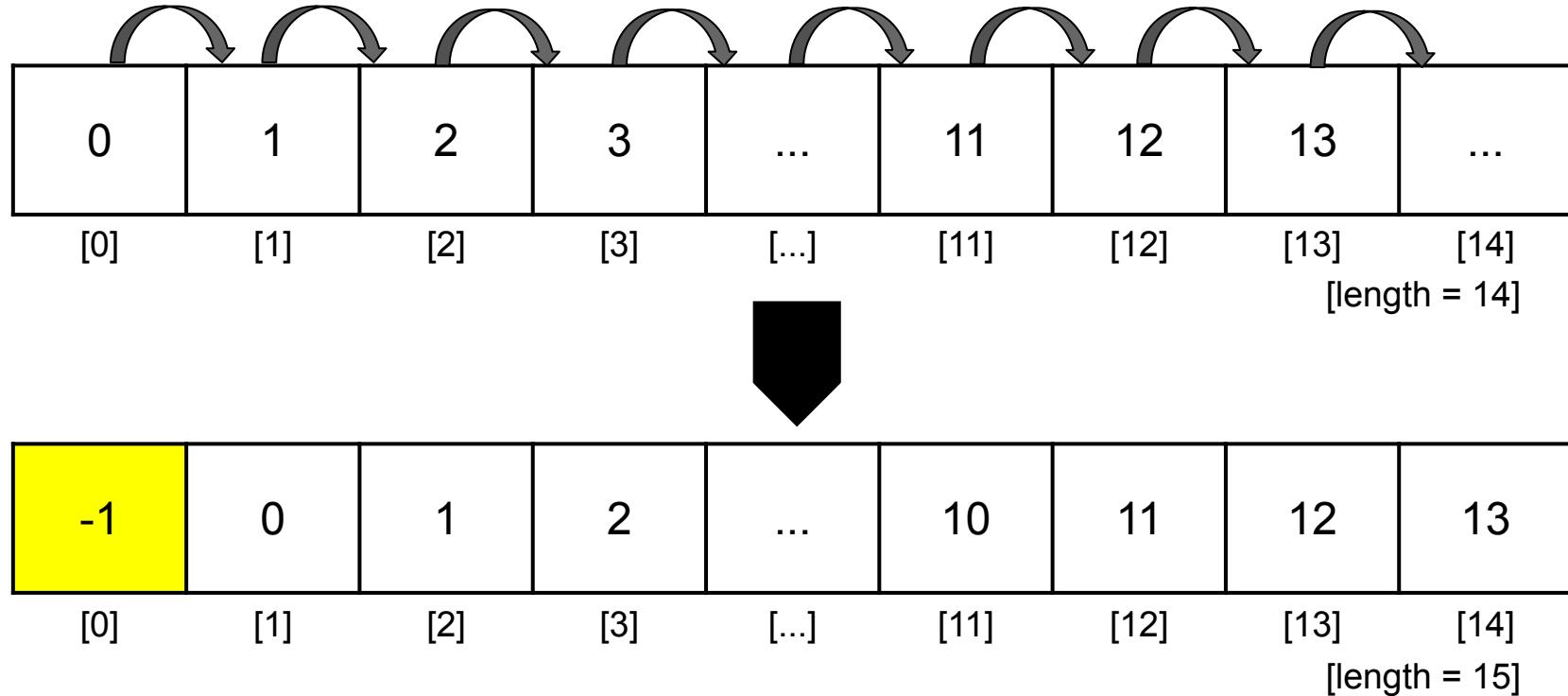
```
myArray.unshift(-1);
```

What's really happening:

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

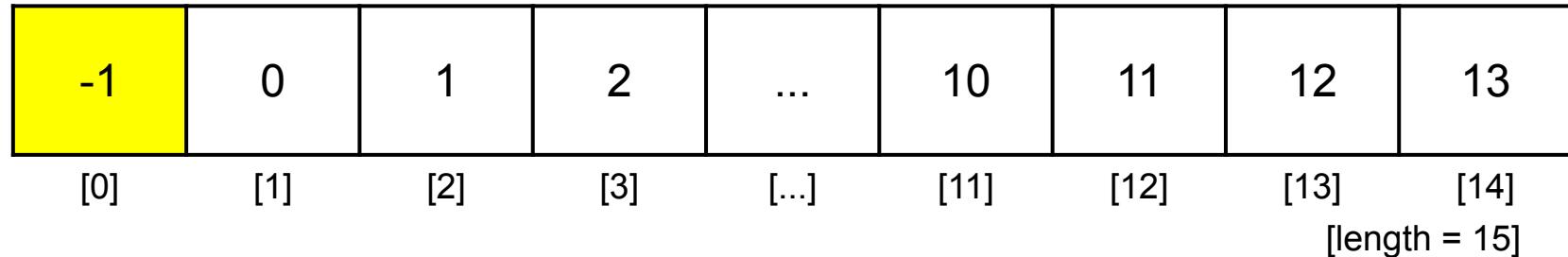
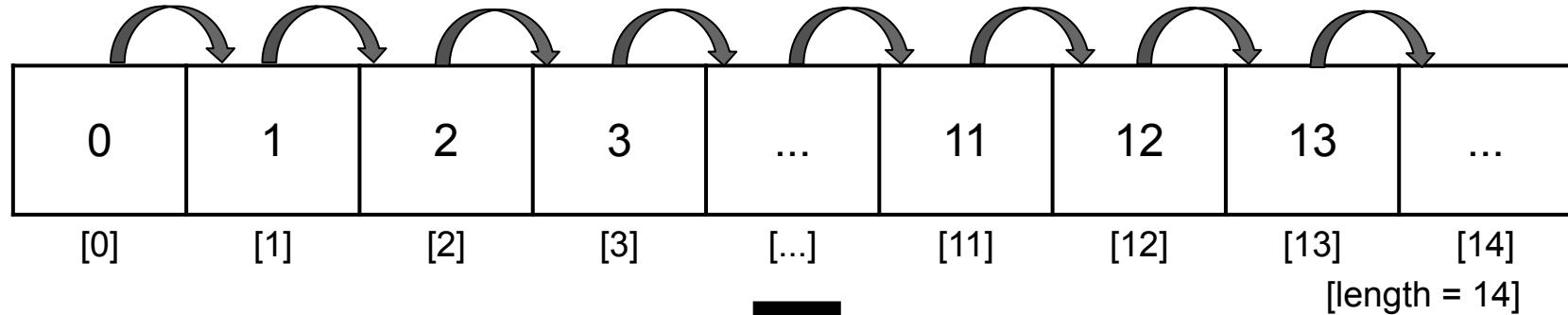
# Arrays in JavaScript

An inefficiency emerges!



# Arrays in JavaScript

An inefficiency emerges! We'll come back to this.



# Stacks/Queues



## **Data Structures = Abstractions**

Going forward, treat each of the following data structures as concepts. 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.



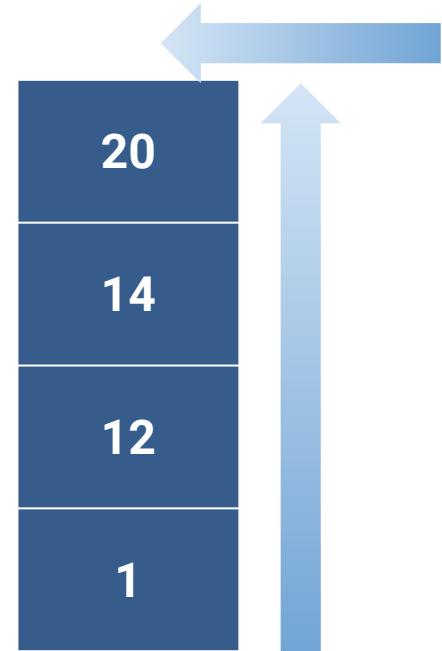
The difference is they **only allow access to the top element**.



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** of 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.



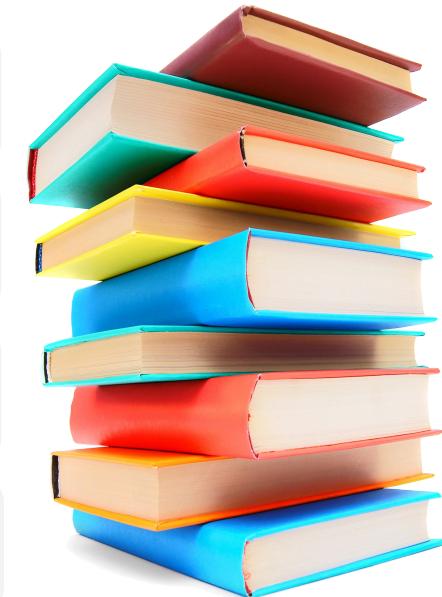
The difference is that they **only allow access to the top element.**



These data structures obey “**LIFO**” (**last in, first out**). This means that new elements are placed at the top and removed from the top.

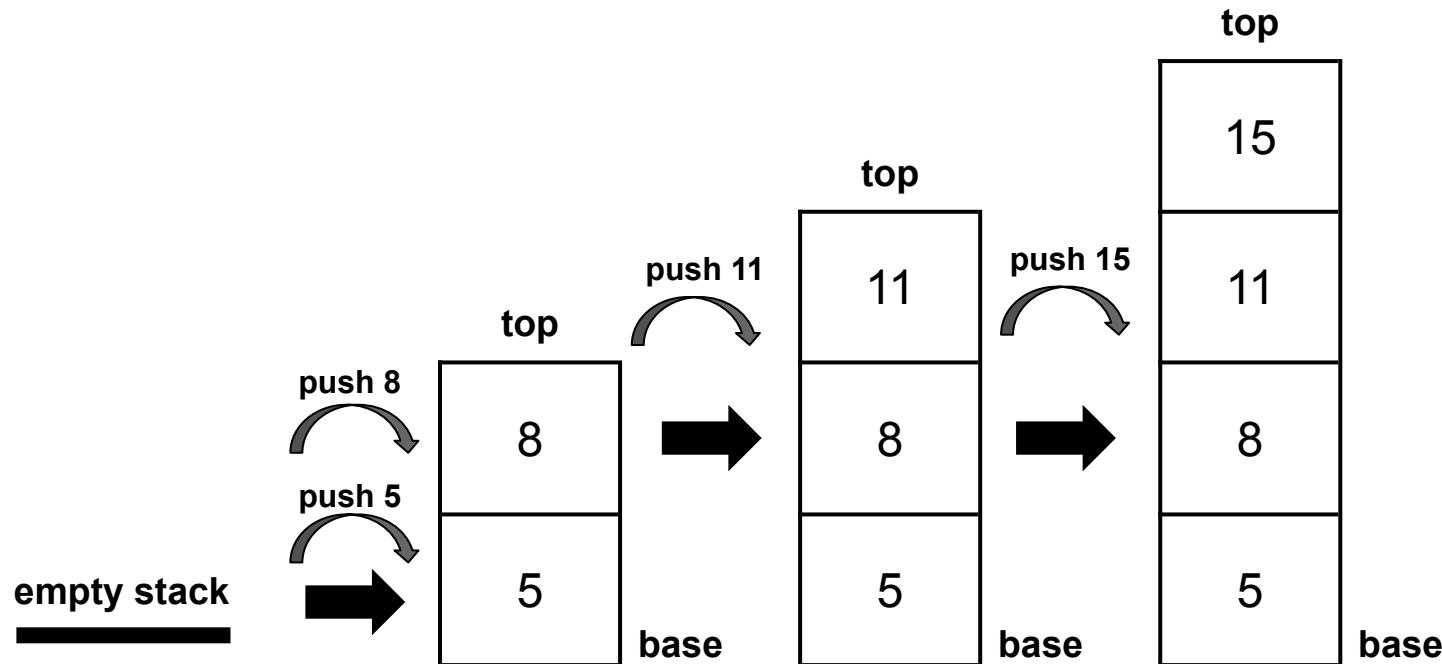


Stacks are an **abstraction** of how data can be arranged.



# Stacks

**Last in First Out:** Items added to the top. Removed from the top



# Stacks: In Code

“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.

```
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 = [];  
  }  
}
```

```
// 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.



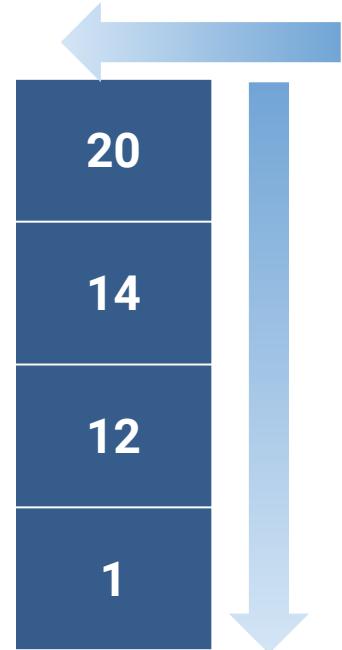
The difference is they **only allow access to the first element**.



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.



Queues are an **abstraction** of 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

“Queues” aren’t supported natively in JavaScript.

Again, this means we need to create our own.

Queues provide two common methods: **enqueue** and **dequeue**.

```
// Creates the Queue Class for use later
class Queue {

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

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

  dequeue() {
    this.items.shift();
  }

  get first() {
    return this.items[0];
  }

  isEmpty() {
    return this.items.length === 0;
  }

  size() {
    return this.items.length;
  }
}
```

```
// Creates an instance of the Queue
var newQueue = new Queue();

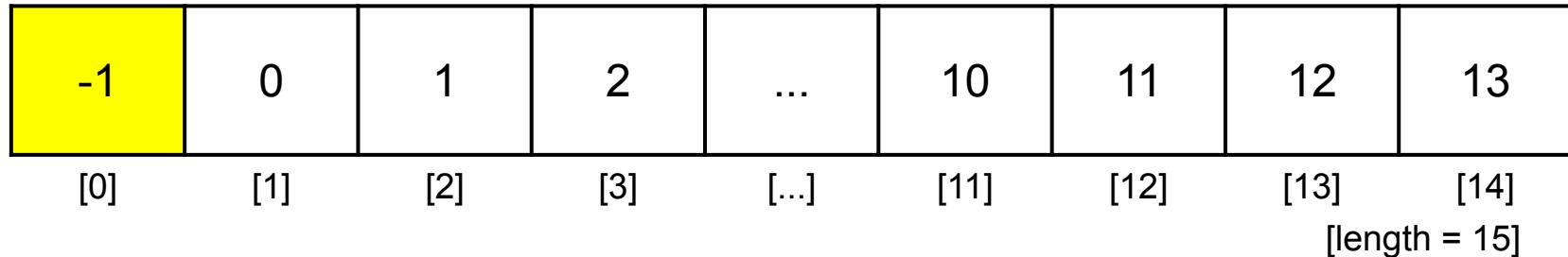
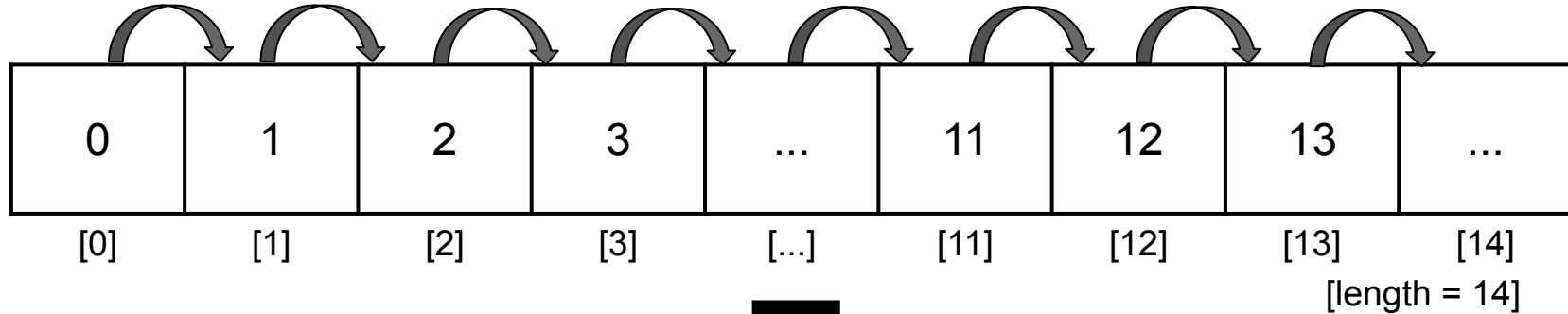
// Starts running methods
newQueue.enqueue("Ahmed");
newQueue.enqueue("Roger");
newQueue.enqueue("John");

console.log(newQueue.first);
```

# Linked Lists

# Arrays in JavaScript

An inefficiency emerges! We'll come back to this.



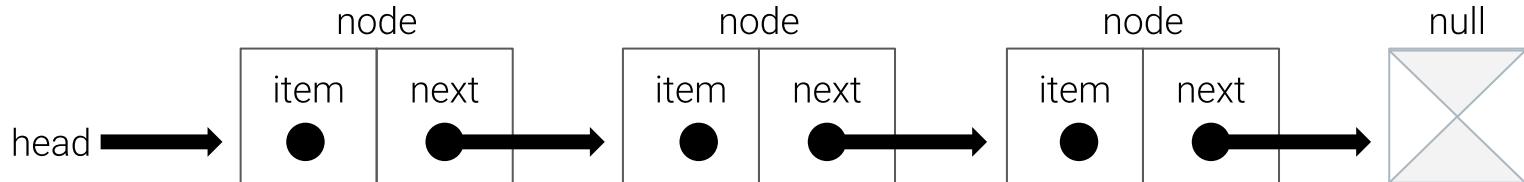
# Linked Lists

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 a **node**) tracks both the item’s and the next item’s position.



# Linked Lists

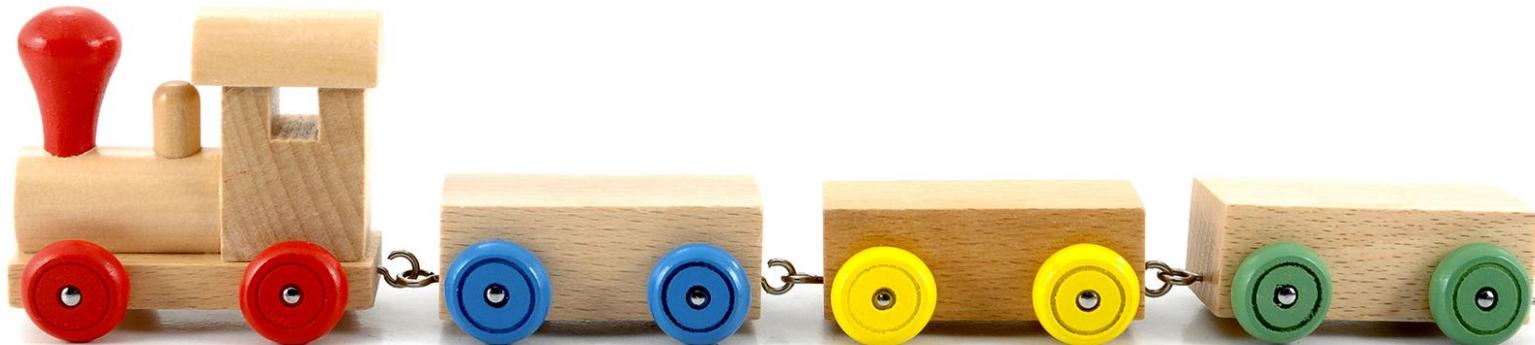
---



Linked lists are like trains.



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



# Linked List: In Code

---

JS does not include linked lists natively, but when you need one, plenty of implementations are available online.

```
1  class Node {
2    constructor(data, next) {
3      this.data = data;
4      this.next = next;
5    }
6
7    getData() {
8      return this.data;
9    }
10
11   setData(data) {
12     this.data = data;
13   }
14
15   getNext() {
16     return this.next;
17   }
18
19   setNext(next) {
20     this.next = next;
21   }
22 }
23
24 class LinkedList {
25   constructor(dataArray) {
26     this.first = new Node();
27
28     var counter = 0;
29     if (dataArray) {
30       var actual = this.first;
31       for (var data of dataArray) {
32         var newNode = new Node(data);
33         actual.setNext(newNode);
```

# Pulse Check

# You Be the Teacher

---

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



What is a data structure?



What does FIFO and LIFO stand for and mean?



What is a stack?



What is a queue?



What is a linked list?



How are they each different from arrays?



What is one disadvantage of an array?



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.



How would you print all the pet names?

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

# Dictionaries (Maps): Actually Useful

Dictionaries are an incredibly important data structure. In fact, they address a common situation you've faced in this class.



How would you print all the pet names?

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

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 are like a hybrid of objects and arrays.



They can be iterated over like arrays.



They have key, value pairs like objects.



They're included in 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!**

## Learn More About Dictionaries (Maps) in JS:

[developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Map](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.



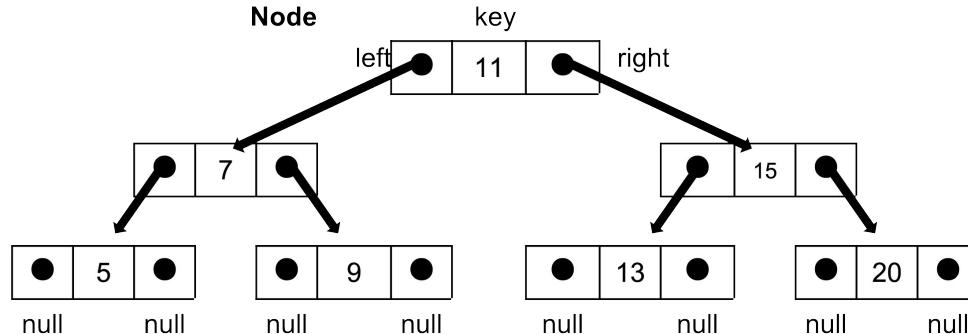
The top node of a tree is the root.



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 the left and the right.

In a Binary Search Tree:



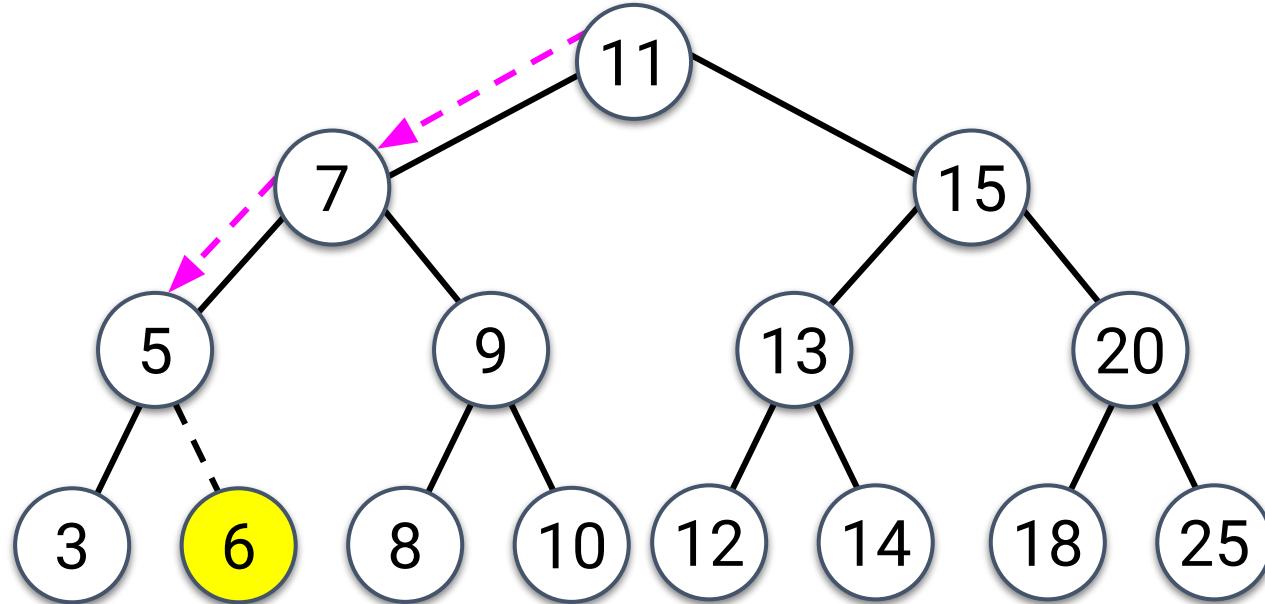
Left-hand side is lesser number; right-hand side is the larger



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

# Binary Trees

Binary search trees are extremely efficient for searching.

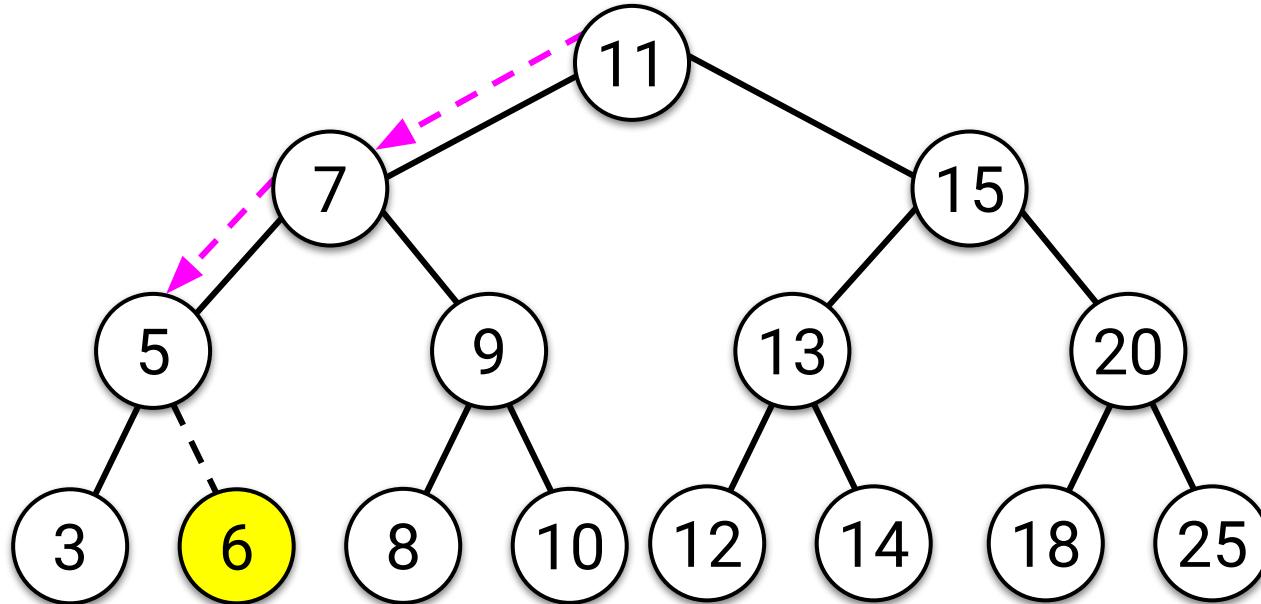


# Binary Search Trees

[npmjs.com/package/binary-search-tree](https://npmjs.com/package/binary-search-tree)

# Activity: 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

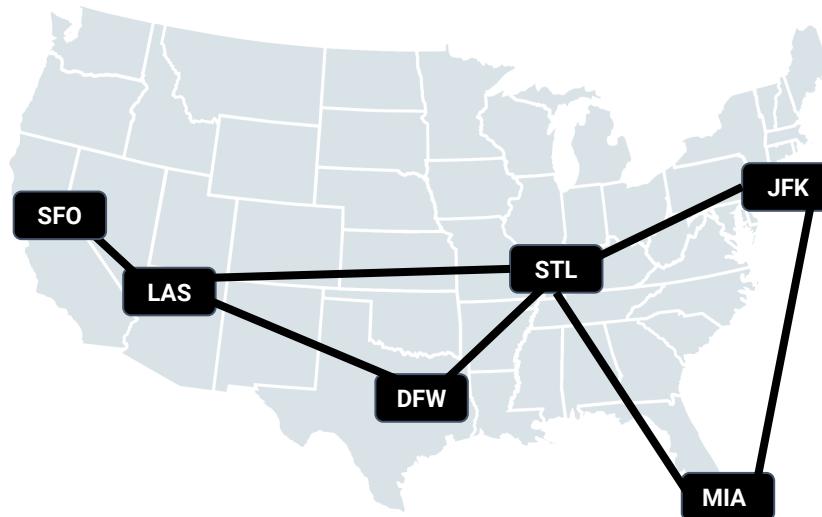
Graphs are extremely powerful and increasingly common structures.



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.

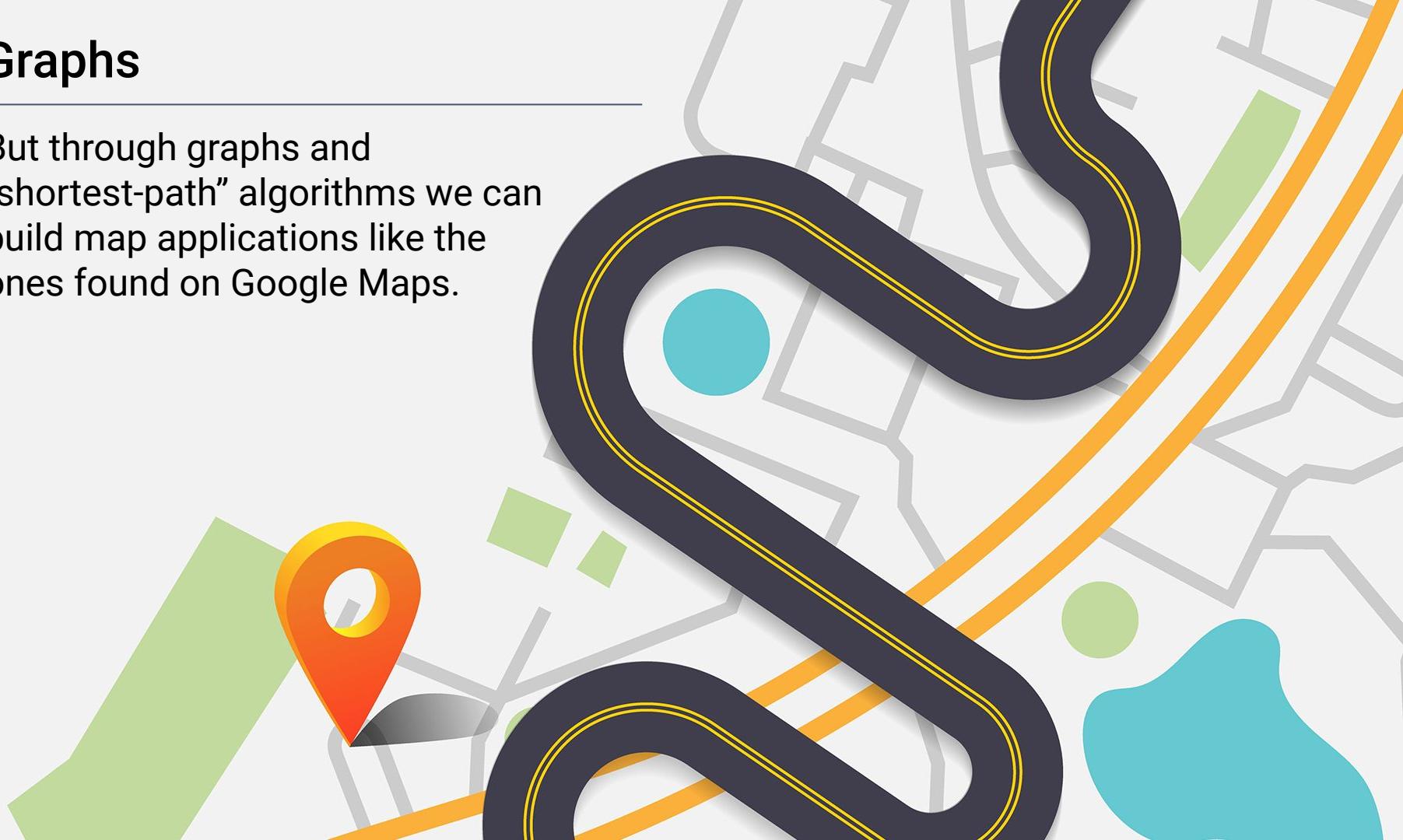


The math gets ridiculously scary with this stuff.

# Graphs

---

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



# Back to Projects!

# Questions?