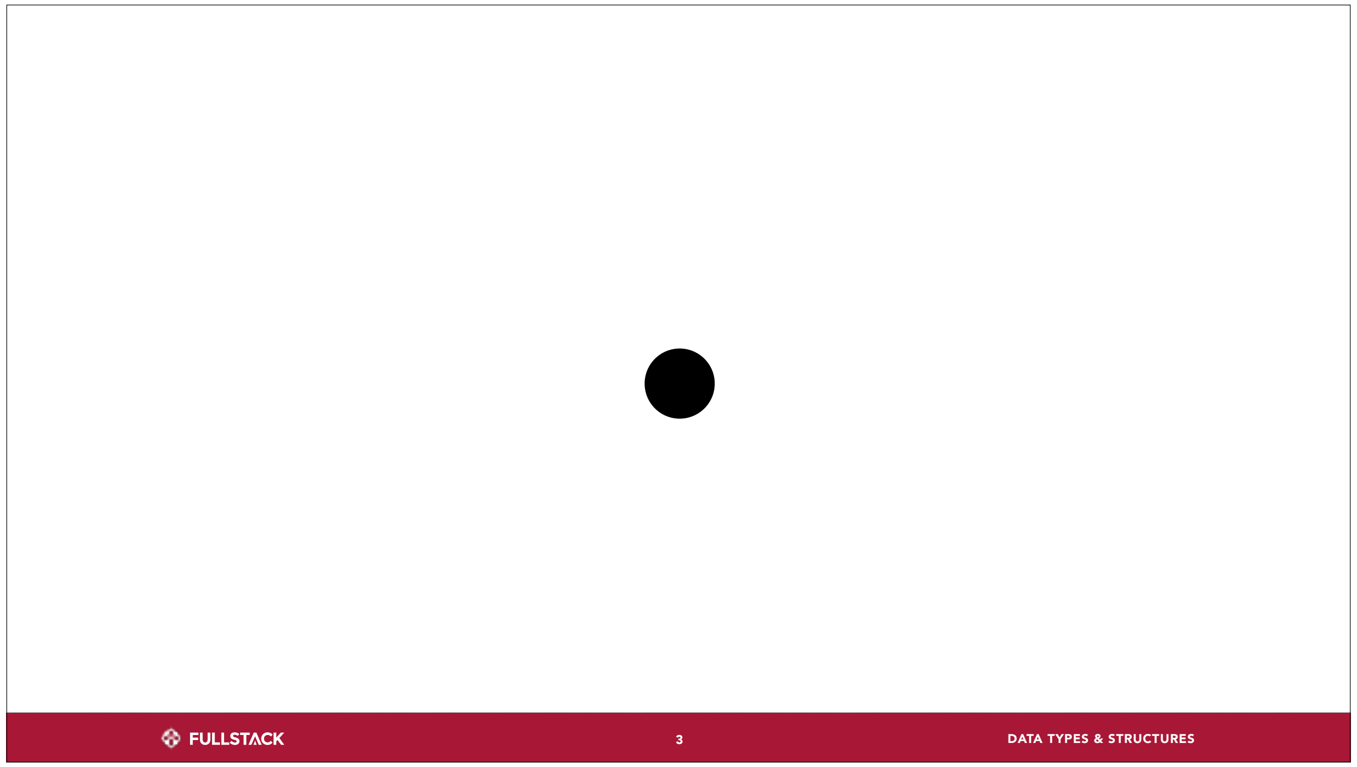
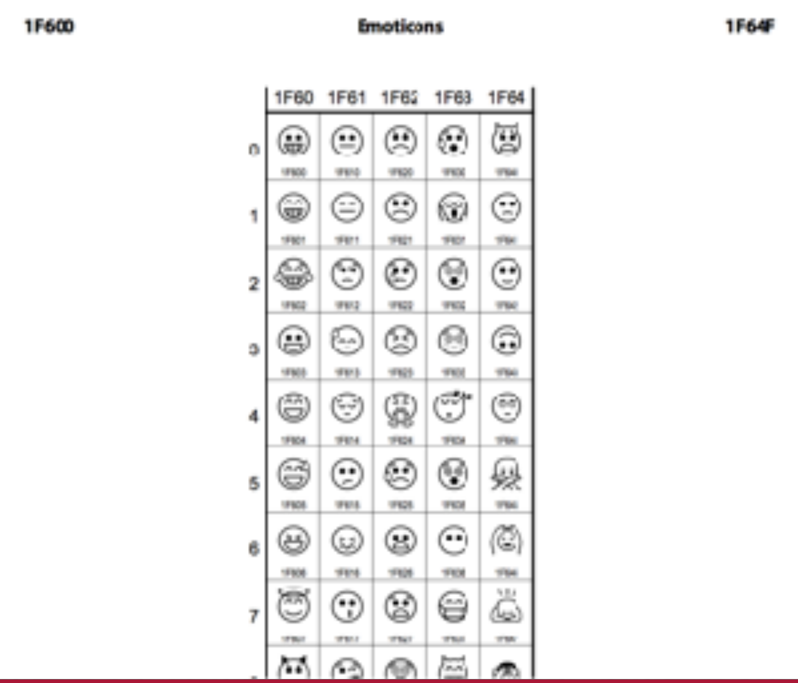
Data Structures

Structure ALL the data



Light/darkness, on/off, zero/one

The circuits in a computer's processor are made up of billions of transistors. A transistor is a tiny switch that is activated by the electronic signals it receives. The digits 1 and 0 used in binary reflect the on and off states of a transistor



1F600 Emoticons 1F64F

	1F60	1F61	1F62	1F63	1F64
0	😀 1F600	😐 1F610	😠 1F620	😰 1F630	🙀 1F640
1	😁 1F601	😑 1F611	😡 1F621	😱 1F631	🙁 1F641
2	😂 1F602	😒 1F612	😢 1F622	😲 1F632	🙂 1F642
3	😃 1F603	😓 1F613	😣 1F623	😳 1F633	🙃 1F643
4	😄 1F604	😔 1F614	😤 1F624	😴 1F634	🙄 1F644
5	😅 1F605	😕 1F615	😥 1F625	😵 1F635	🙅 1F645
6	😆 1F606	😖 1F616	😦 1F626	😶 1F636	🙆 1F646
7	😇 1F607	😗 1F617	😧 1F627	😷 1F637	🙇 1F647

How do we store this data?

What if we were storing a bunch of rocks, representing bits (with one side representing 1 and the other representing 0).

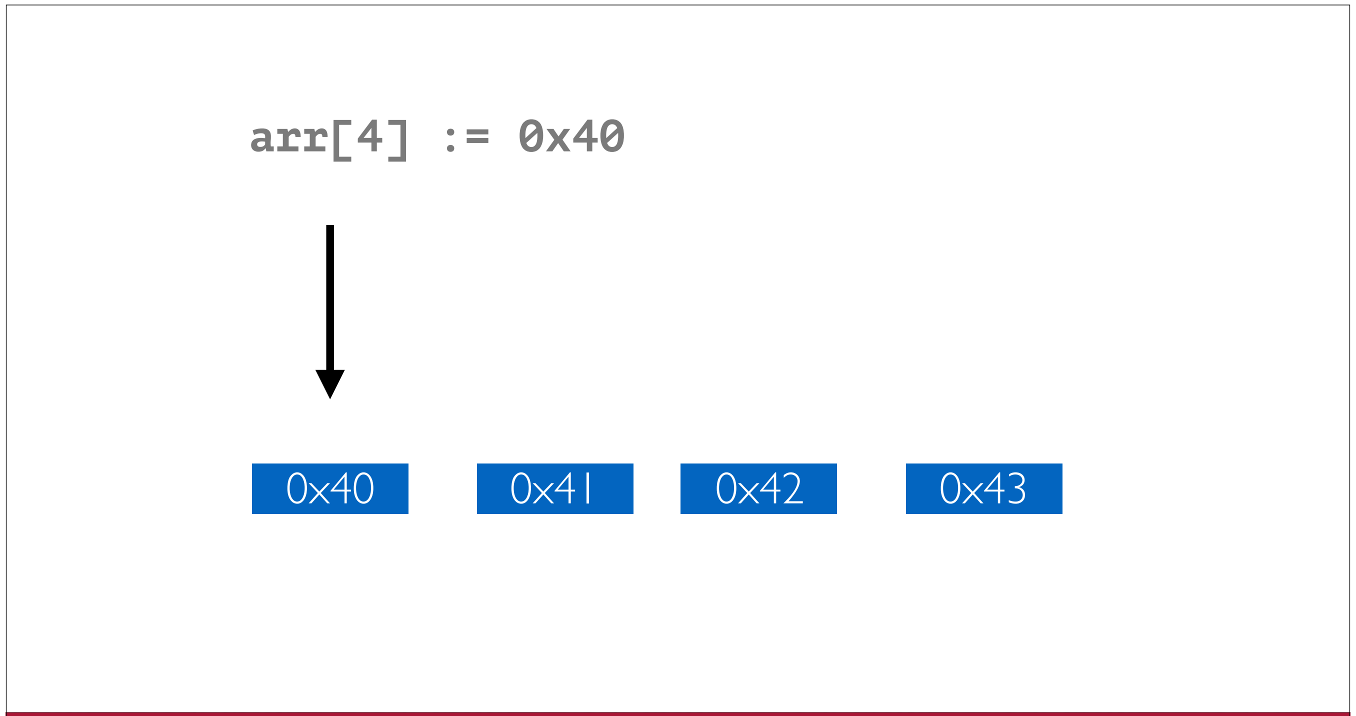
CONTIGUOUS ARRAY

Contiguous Array

- Represents adjacent addresses in memory
- Fixed size
- Each element is the same size
- Analogy: book

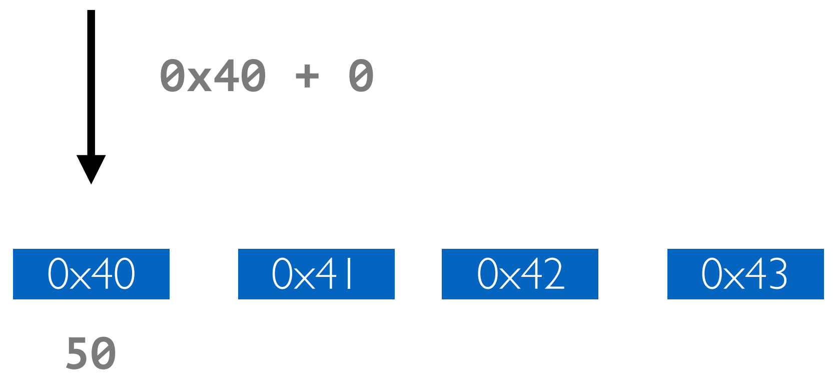
```
int arr[4];
// reserves 4 buckets in memory that
// are right next to each other

// “arr” is actually a reference to
// the first memory address in our
// contiguous series

// store “50” at that address + 0
arr[0] = 50;

// store “30” at the address + 2
arr[2] = 30;
```

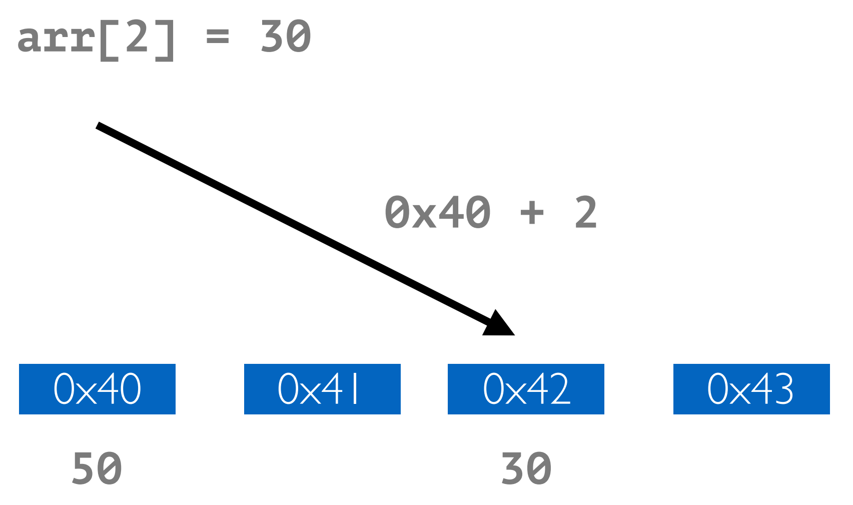
0x40 0x41 0x42 0x43



```
arr[4] := 0x40
```

```
arr[0] = 50
```

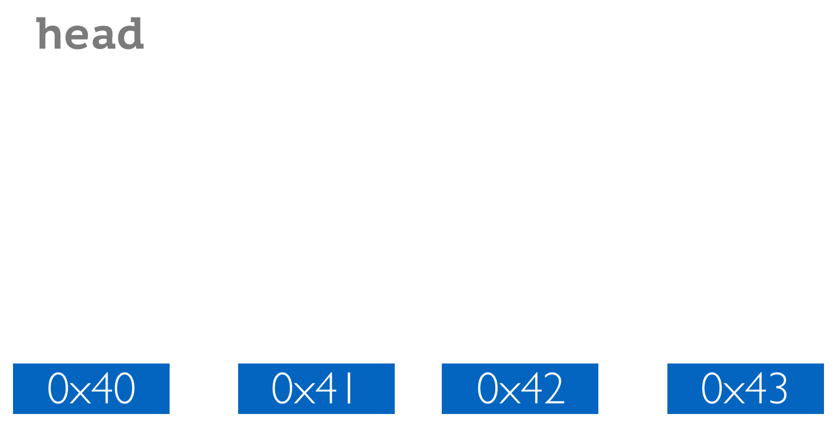


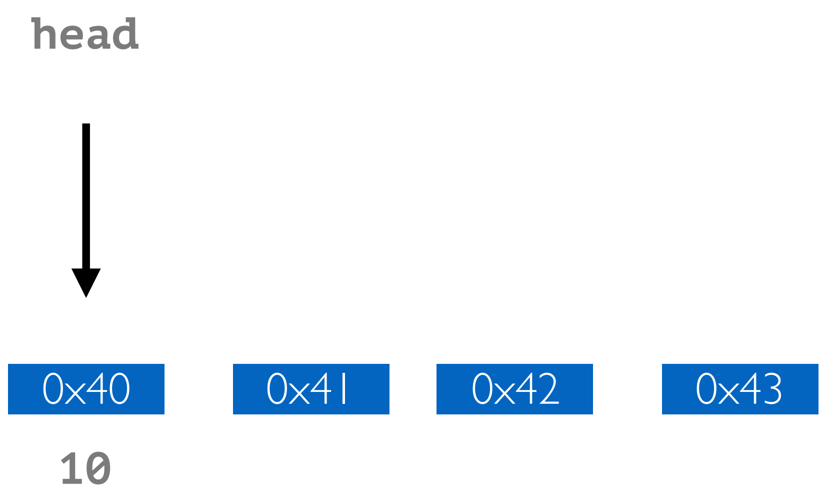

arr[2] = 30
0x40 + 2
0x40
0x41
0x42
0x43
50
30

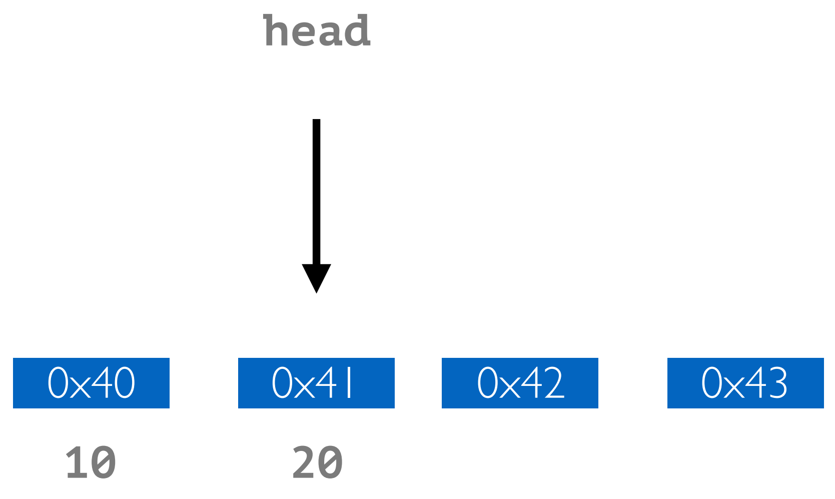
THAT'S WHY WE START COUNTING AT 0

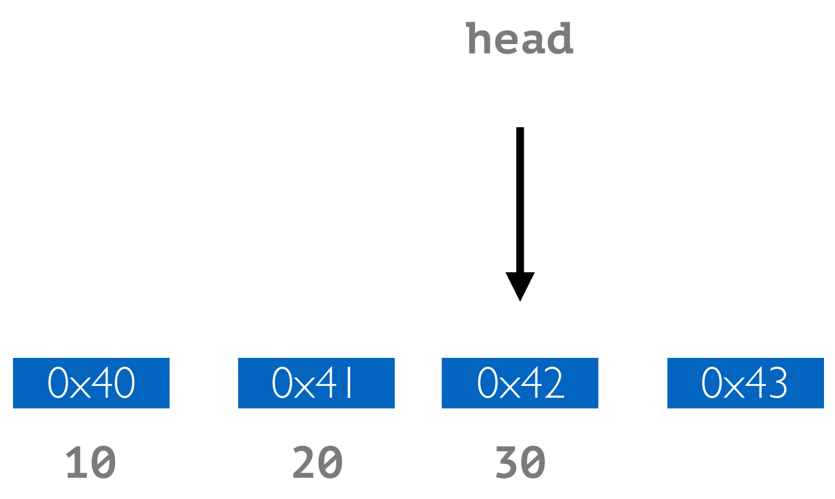
Segue: We can leverage pointer arithmetic to help with retrieval

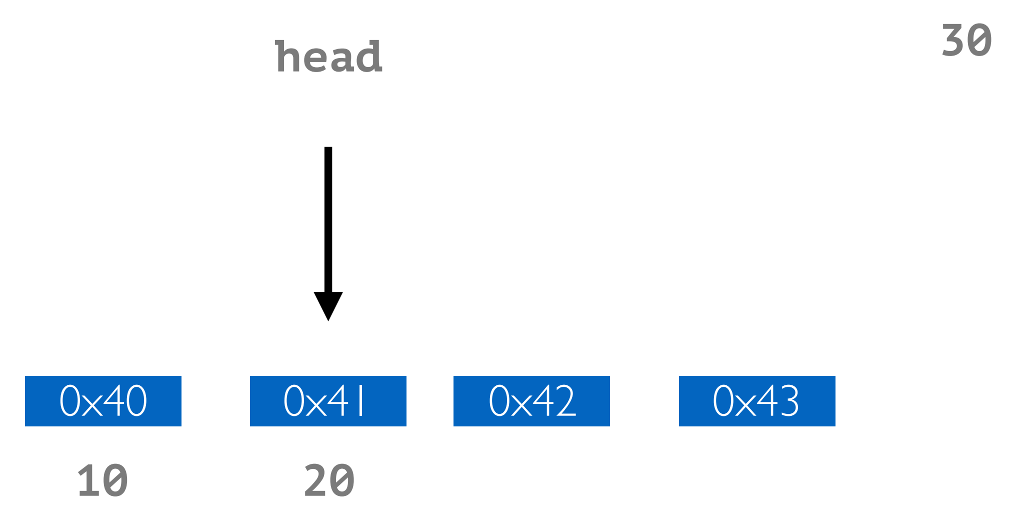
STACKS & QUEUES

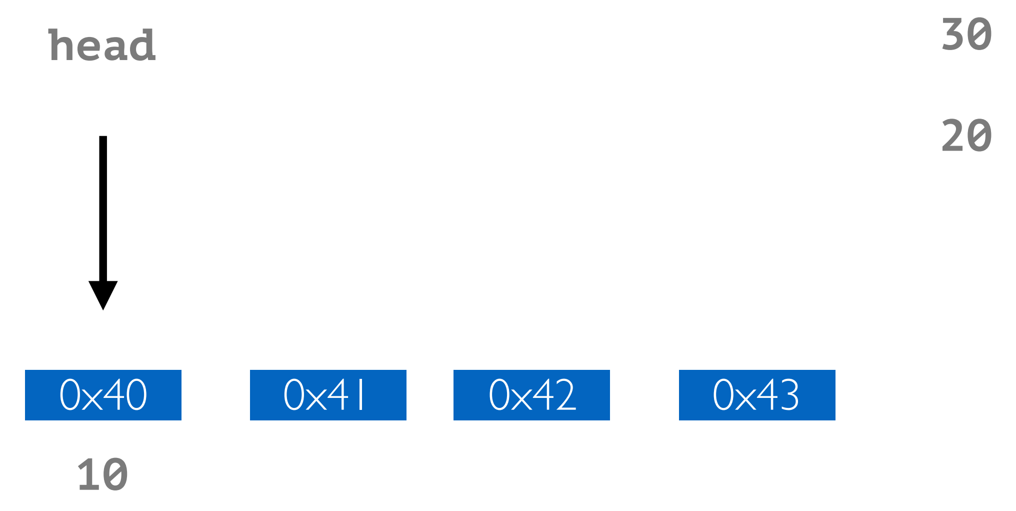

head
0x40
0x41
0x42
0x43

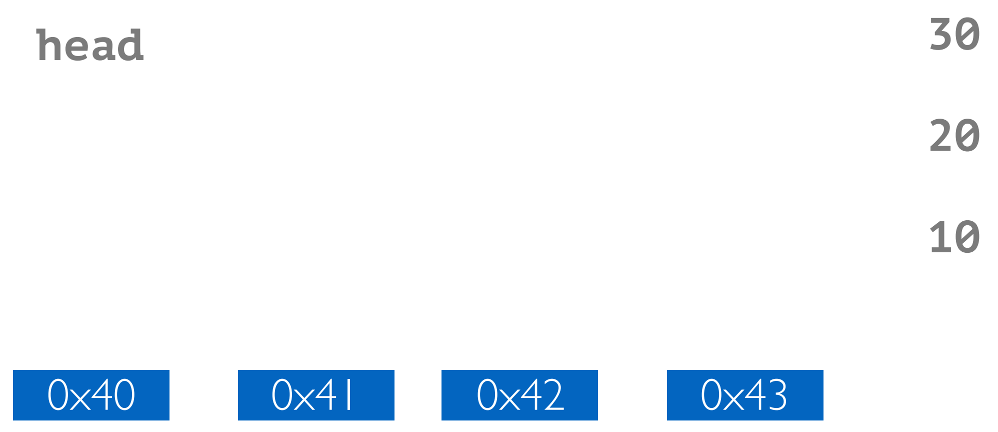

head
0x40
0x41
0x42
0x43
10


head
0x40
0x41
0x42
0x43
10
20


head
0x40
0x41
0x42
0x43
10
20
30


head
30
0x40
0x41
0x42
0x43
10
20


head
30
20
0x40
0x41
0x42
0x43
10

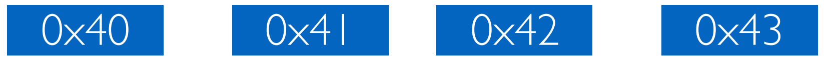

head
30
20
10
0x40
0x41
0x42
0x43

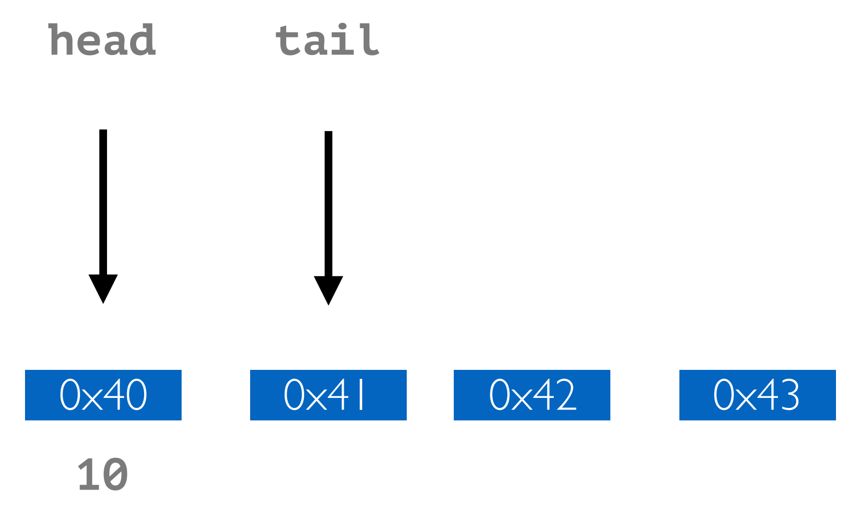
Stacks

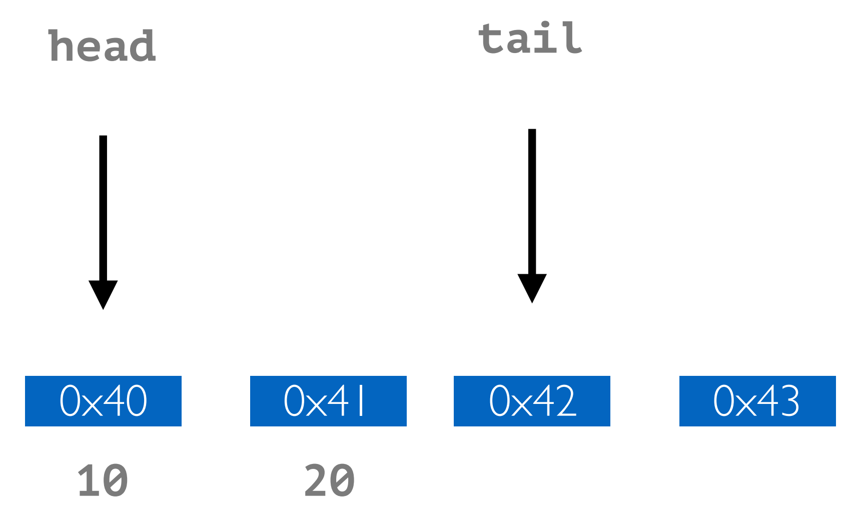
- Use array for storage
- LIFO (FILO)
- Push, pop
- Analogy: pancakes

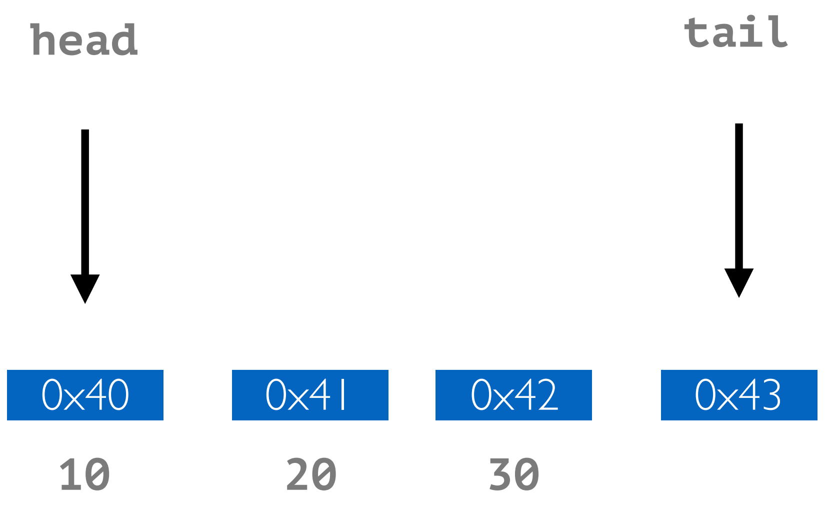
```
const stack = []

stack.push(10)
stack.push(20)
stack.push(30)

stack.pop() // 30
stack.pop() // 20
stack.pop() // 10
```

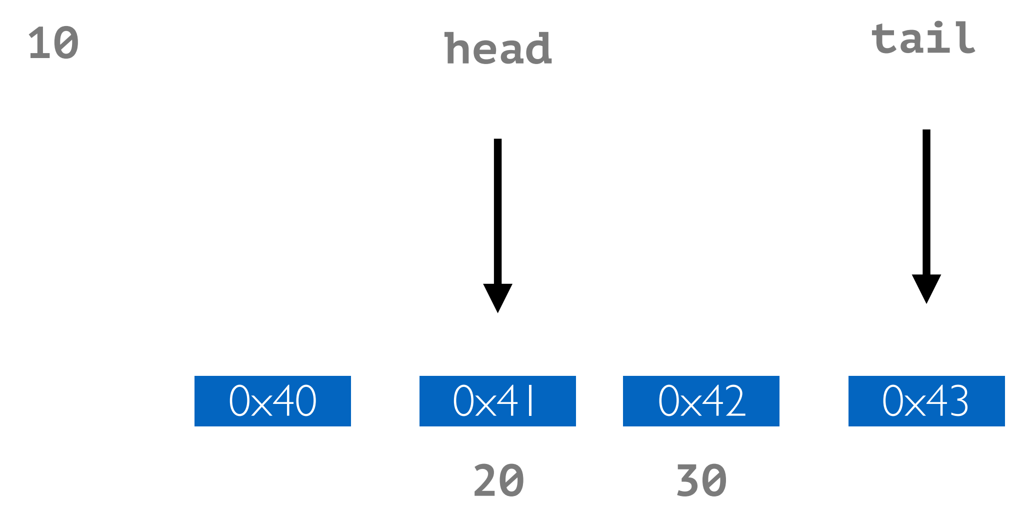
```
head
tail
```

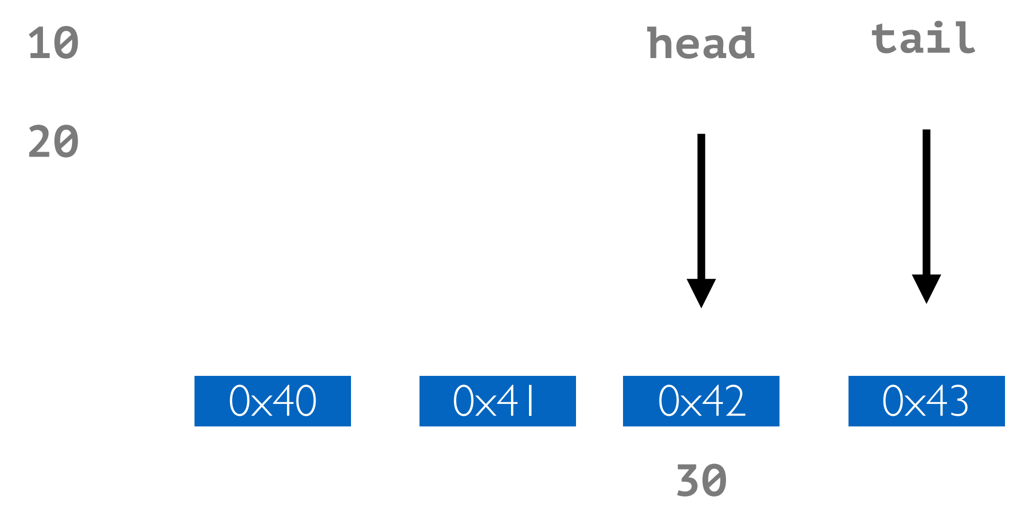


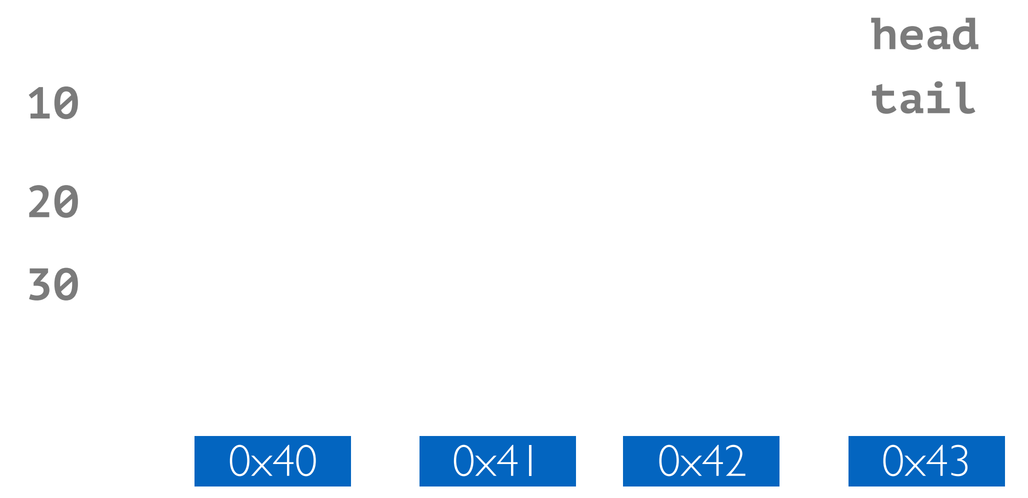

head
tail
0x40
0x41
0x42
0x43
10


head
tail
0x40
0x41
0x42
0x43
10
20


head
tail
0x40
0x41
0x42
0x43
10
20
30


10
head
tail
0x40
0x41
0x42
0x43
20
30


10
20
head
tail
0x40
0x41
0x42
0x43
30

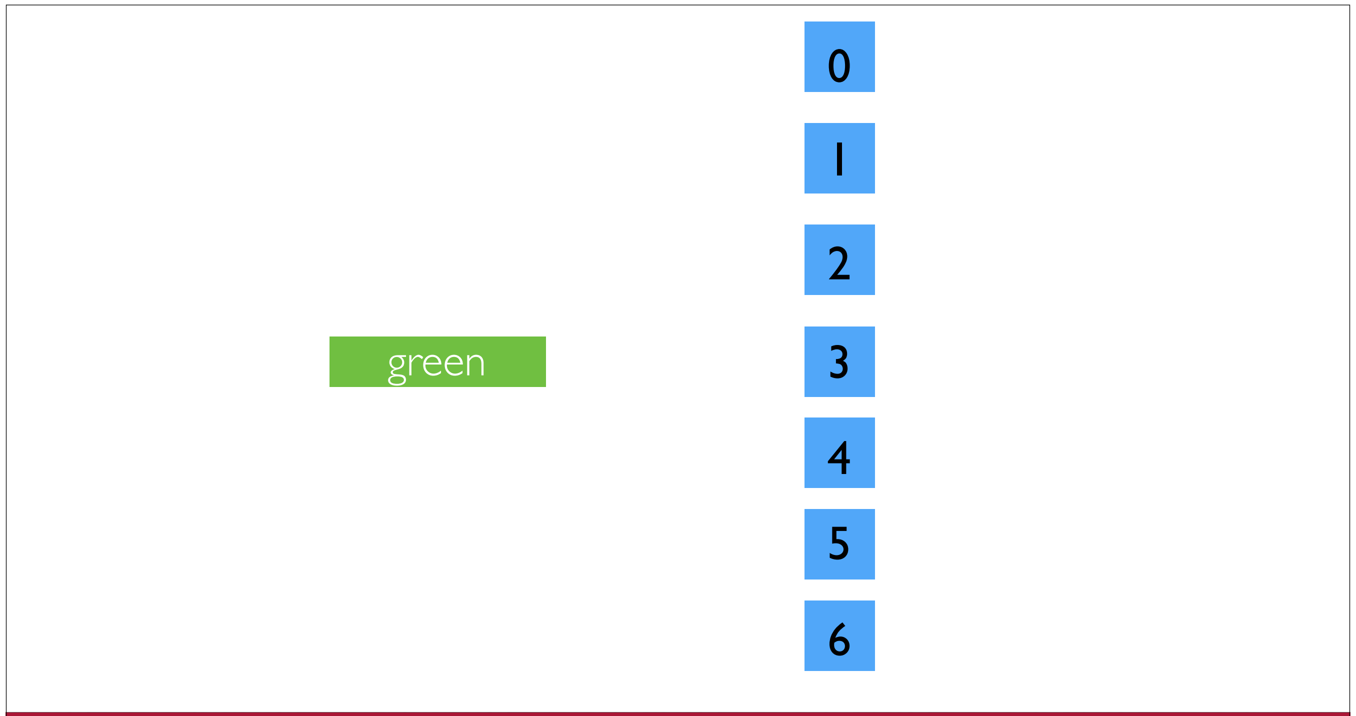

head
tail
10
20
30
0x40
0x41
0x42
0x43

Queues

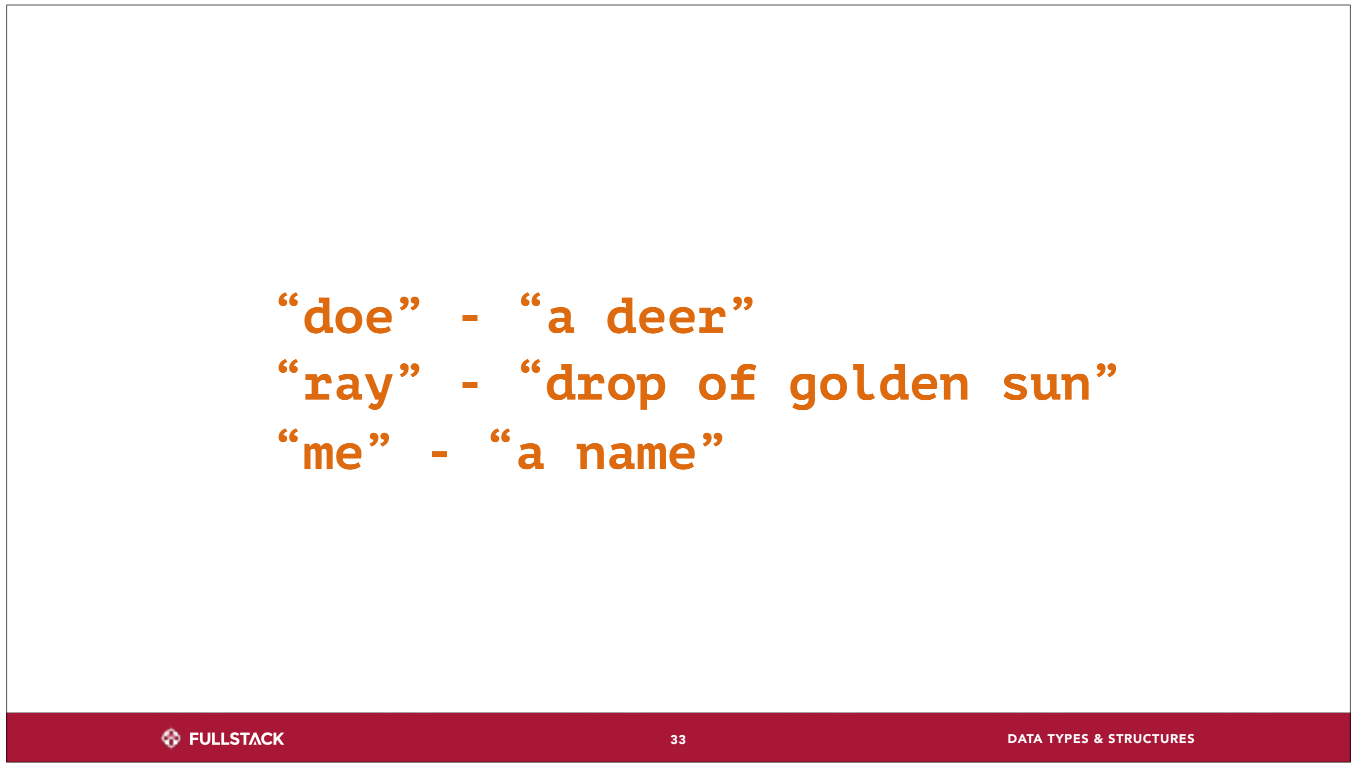
- Use array for storage
- FIFO (LILO)
- Push, shift
- Analogy: standing in line

```
const queue = []

queue.push(10)
queue.push(20)
queue.push(30)

queue.shift() // 10
queue.shift() // 20
queue.shift() // 30
```

PROBLEMS



For example, say we set aside an array with 7 memory cells - (0 to 6). And we want to store some information there, like color codes, for example. We could just store these color codes in the array one after another, but then if we wanted to find one, we would have to check one-by-one through the array of memory to find the one we want.



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

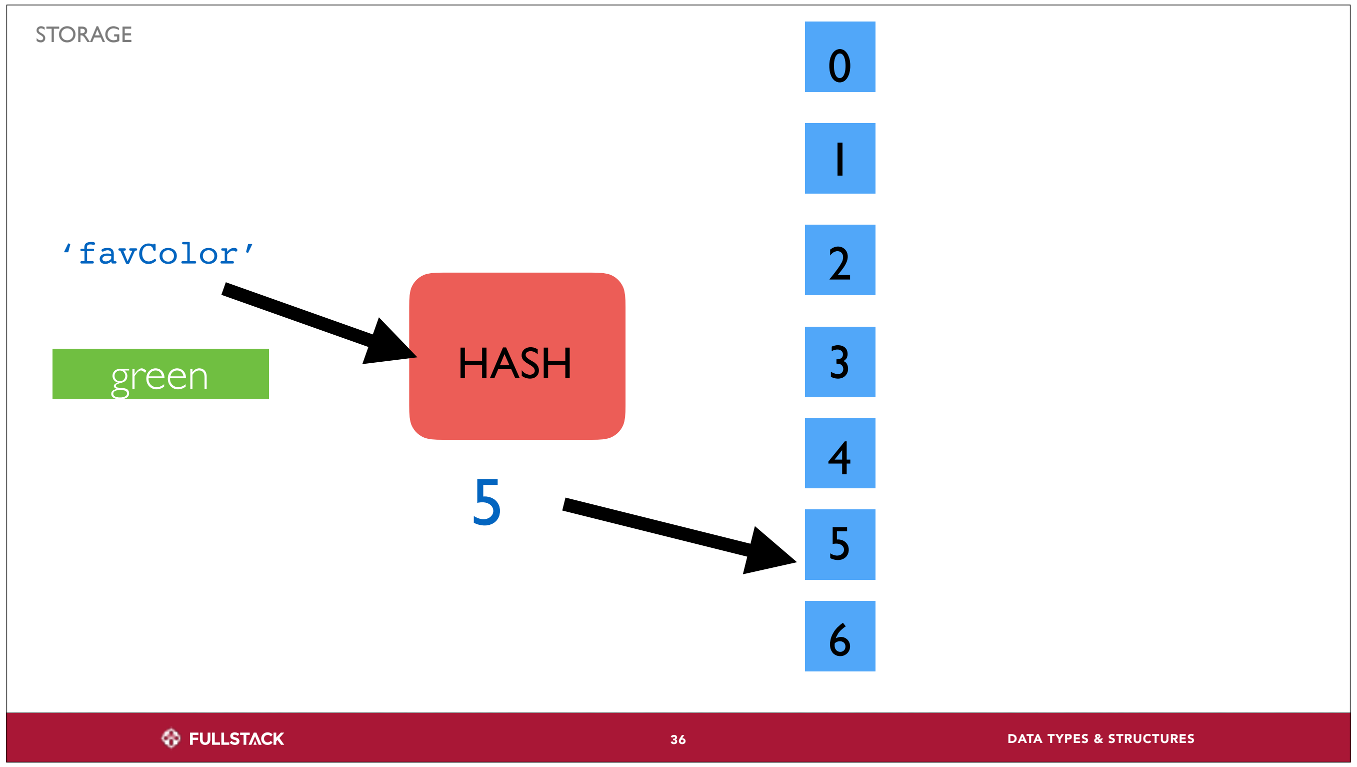
what if our data looks like this?

HASH TABLES

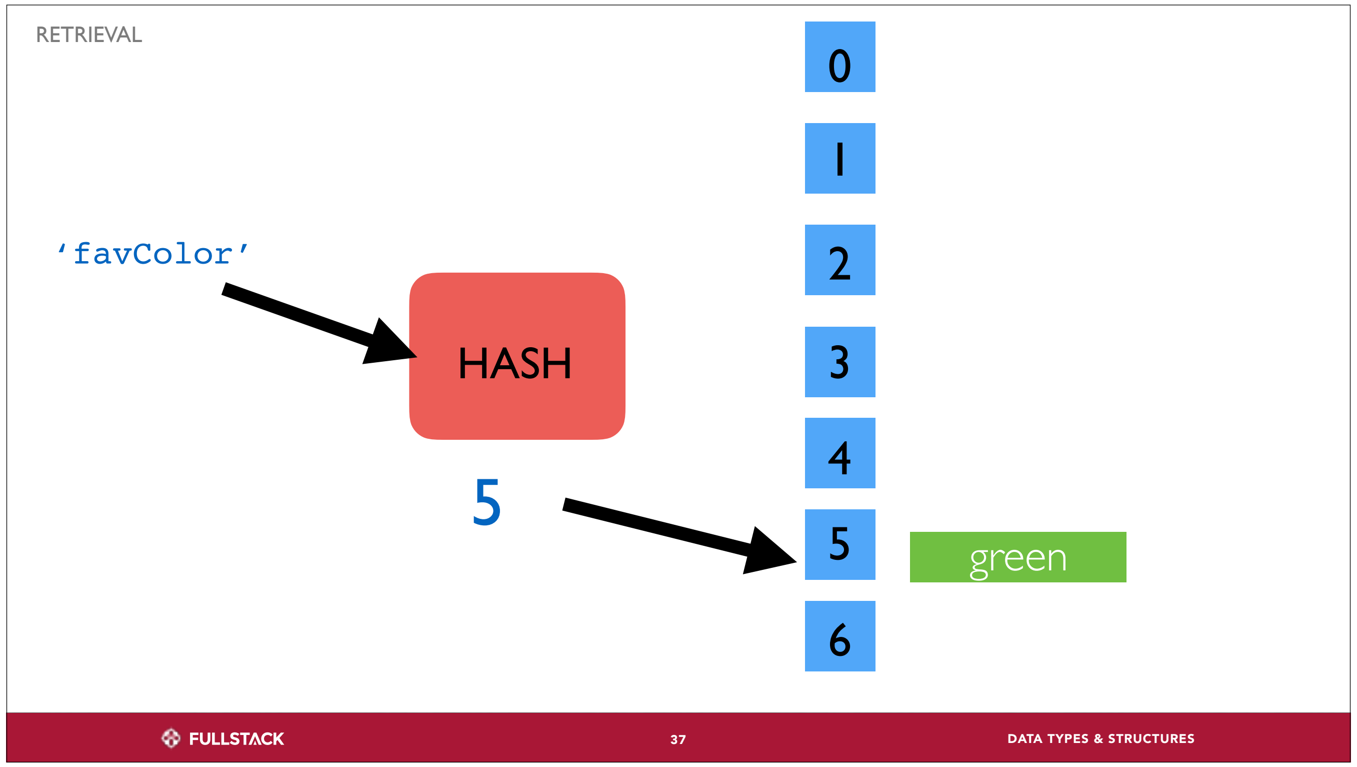
Hash Table

- Uses a hash function on each key to convert to an index in a contiguous array
- Analogy: Dewey Decimal system, filing cabinet

```
const HT = {}

HT[‘blue’] = 0x0000ff
HT[‘red’] = 0xff0000



console.log(HT[‘blue’]) // ‘0x0000ff’
```



Instead, we'll associated each color code value with a key - which will be a string representing the name of the color. Because this hash function always returns a number between 0 - 8 (which correspond to the indices of the array), we can put our key into the hash function, and use its output (the number between 0 - 8) to determine which memory cell, or index, should hold the value. 'Blue', for example, might give us '1'. So we'll store the color code at 1. Then, if we want to retrieve that value later, we don't have to search cell-by-cell for it. We just need to put the key into the hashing function again, and it will tell us exactly where to look.



RETRIEVAL

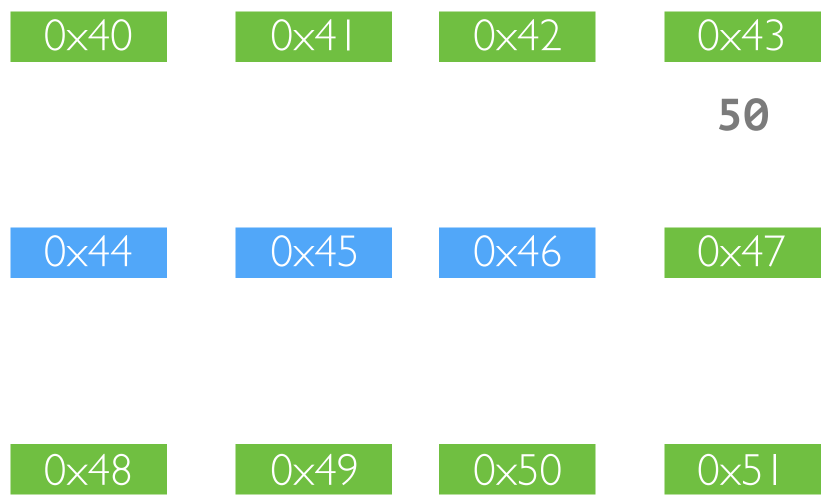
Instead, we’ll associated each color code value with a key - which will be a string representing the name of the color. Because this hash function always returns a number between 0 - 6 (which correspond to the indices of the array), we can put our key into the hash function, and use its output (the number between 0 - 6) to determine which memory cell, or index, should hold the value. ‘Blue’, for example, might give us ‘1’. So we’ll store the color code at 1. Then, if we want to retrieve that value later, we don’t have to search cell-by-cell for it. We just need to put the key into the hashing function again, and it will tell us exactly where to look.

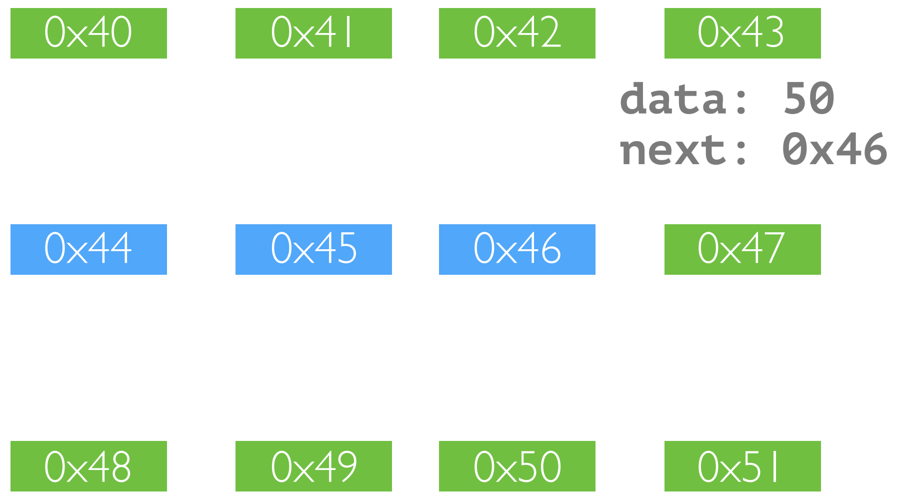
PROBLEMS

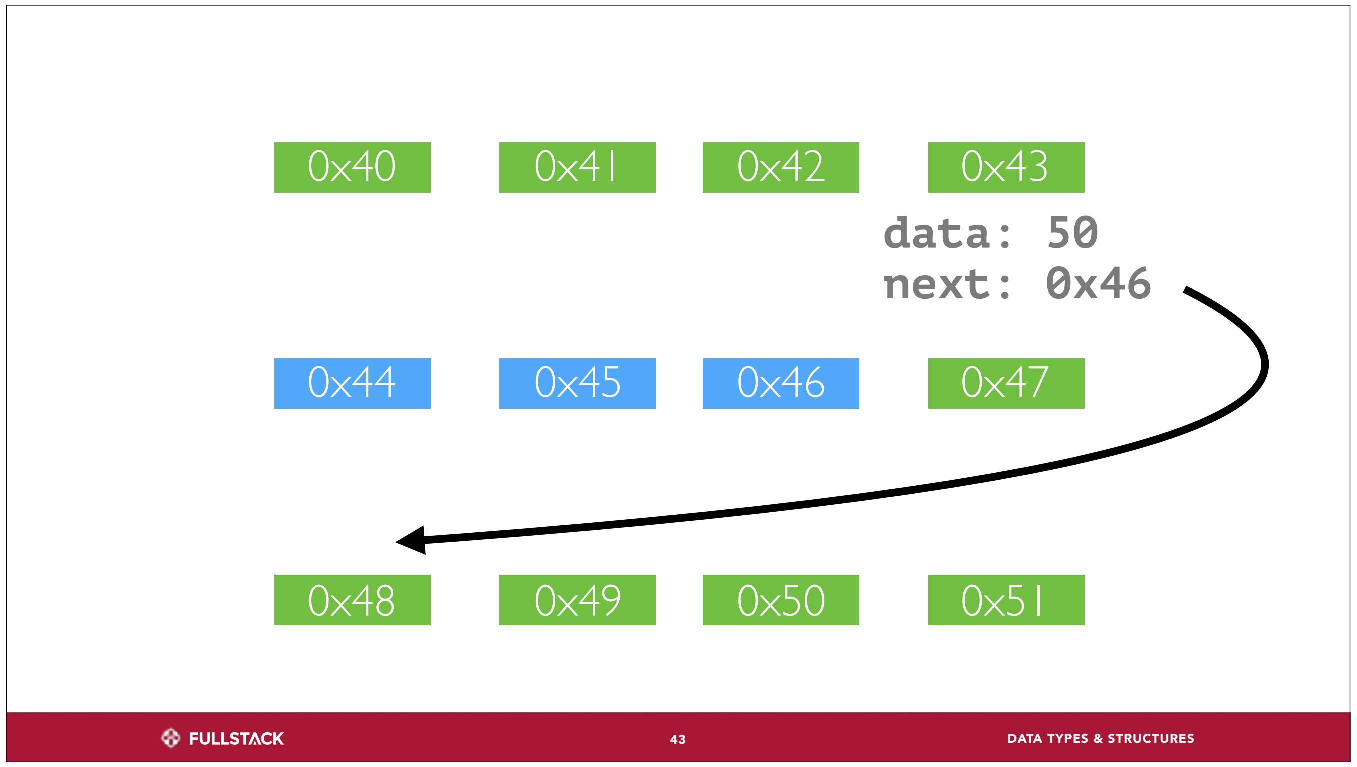
Hash Table Collisions

- 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. Collisions create new entries in that data structure.

Linked Data Structures?


0x40
0x41
0x42
0x43
50
0x44
0x45
0x46
0x47
0x48
0x49
0x50
0x51

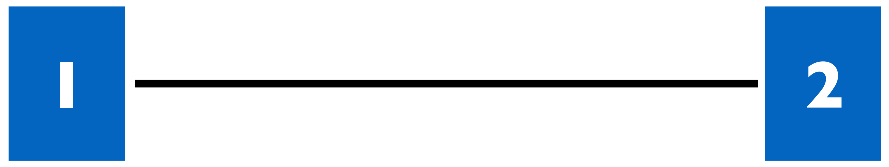

0x40
0x41
0x42
0x43
data: 50
next: 0x46
0x44
0x45
0x46
0x47
0x48
0x49
0x50
0x51

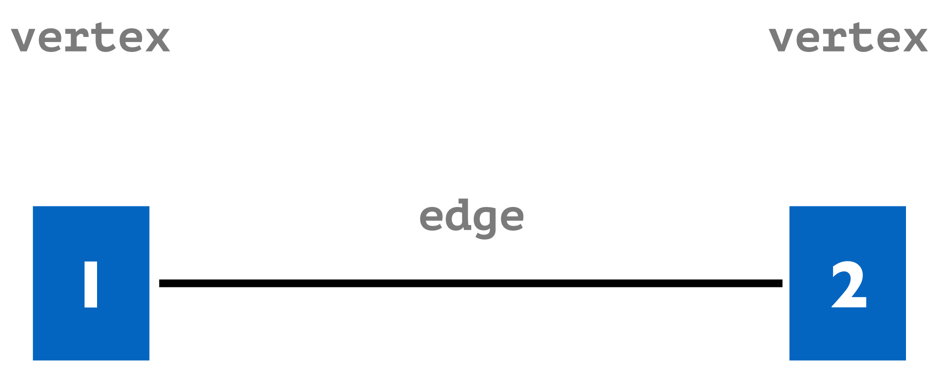


Being able to do this opens up a whole new can of worms.
Segue: but first, let's dive into some elementary set theory

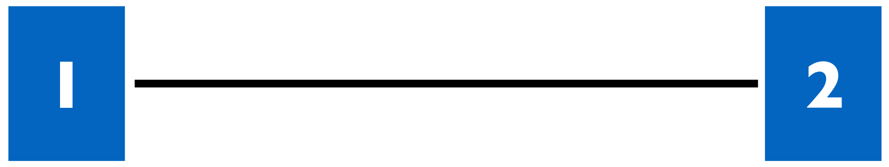
A Crash Course in Graph Theory

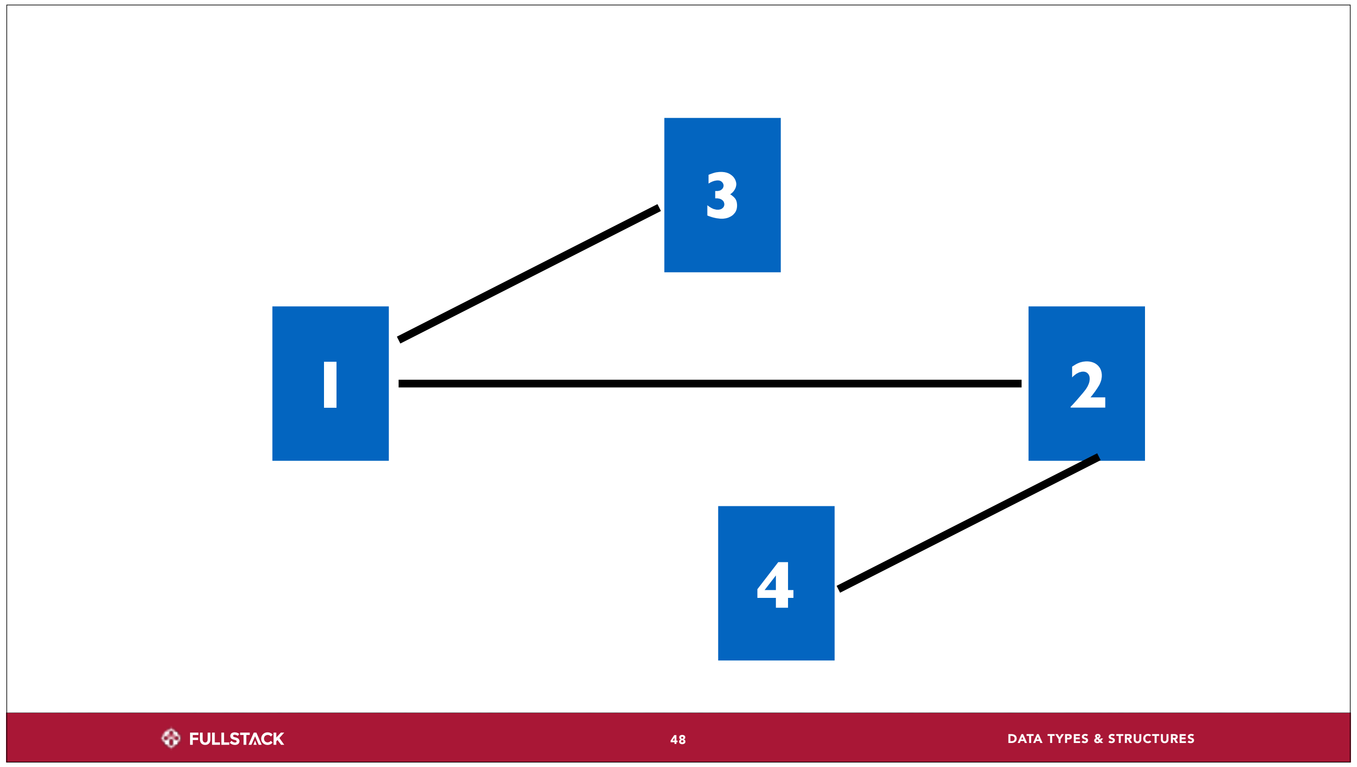
Graphs! Graphs everywhere!

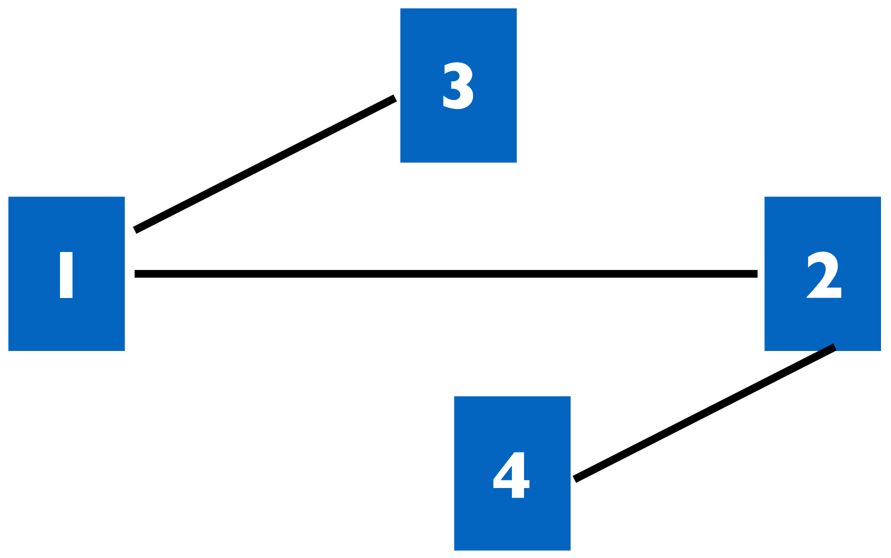

1
2


vertex
vertex
edge
1
2

```
vertices: nodes, elements, items
edges: links, connections, relationships
```





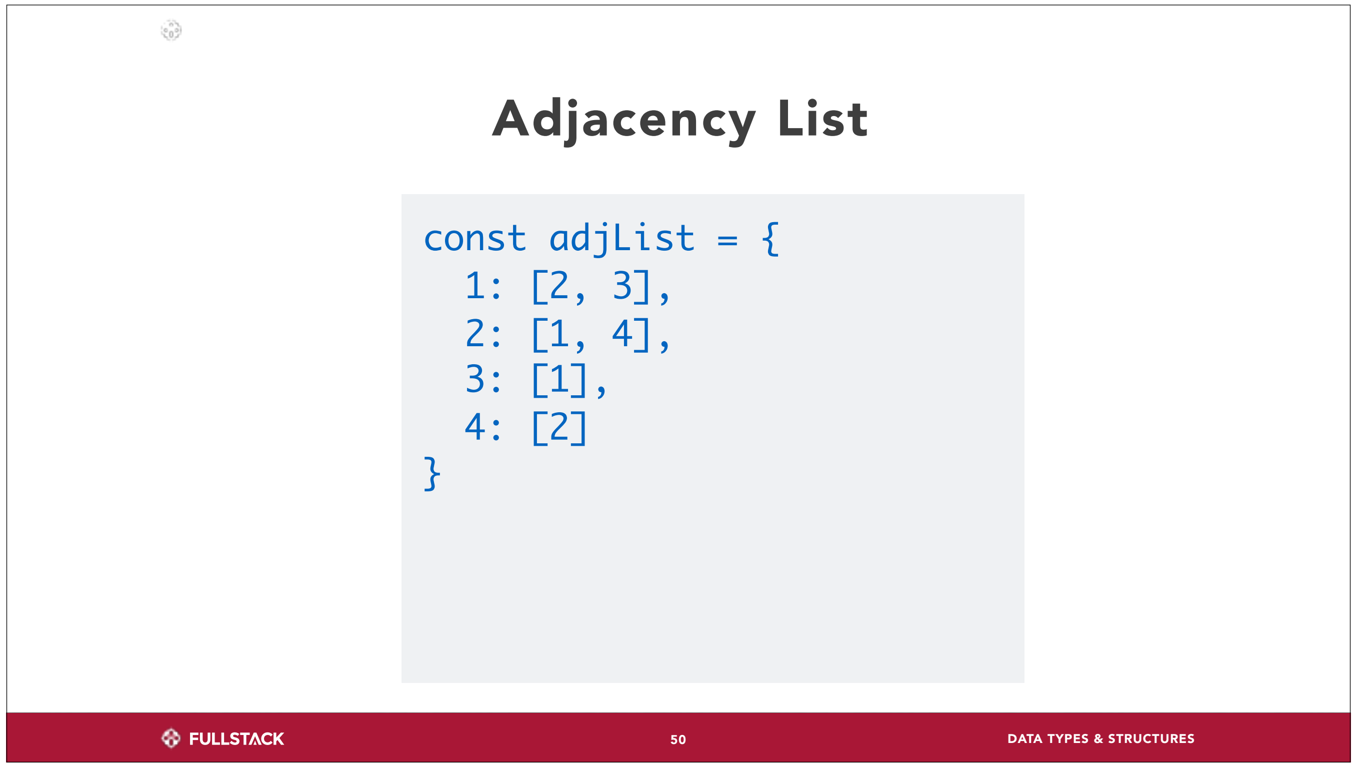


Here's a slightly more verbose graph. How would we represent this? There are two main ways

Adjacency Matrix

```
const adjMatrix = [
       /* 1  2  3  4   */
/* 1 */  [0, 1, 1, 0],
/* 2 */  [1, 0, 0, 1],
/* 3 */  [1, 0, 0, 0],
/* 4 */  [0, 0, 1, 0],
]
```

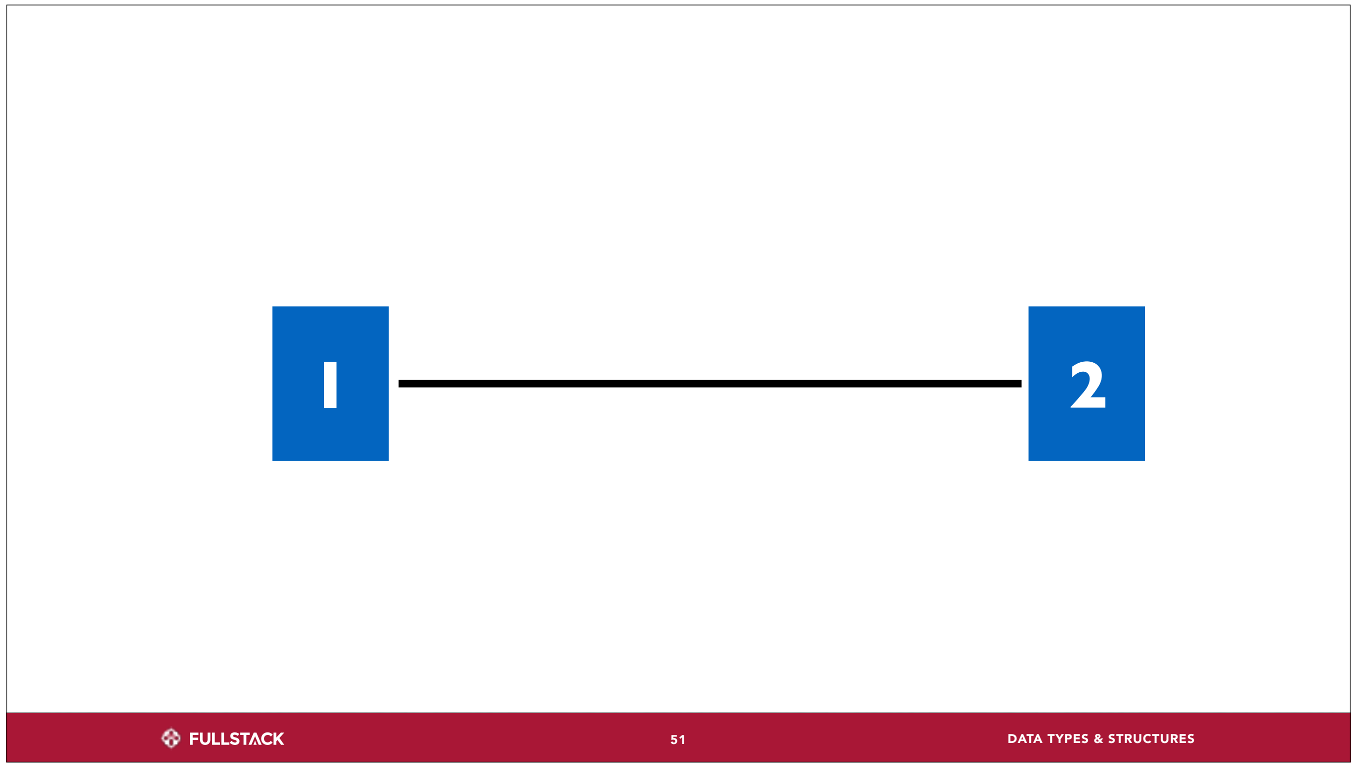
adv: very easy to query edges
disadvantages: space



Adjacency List

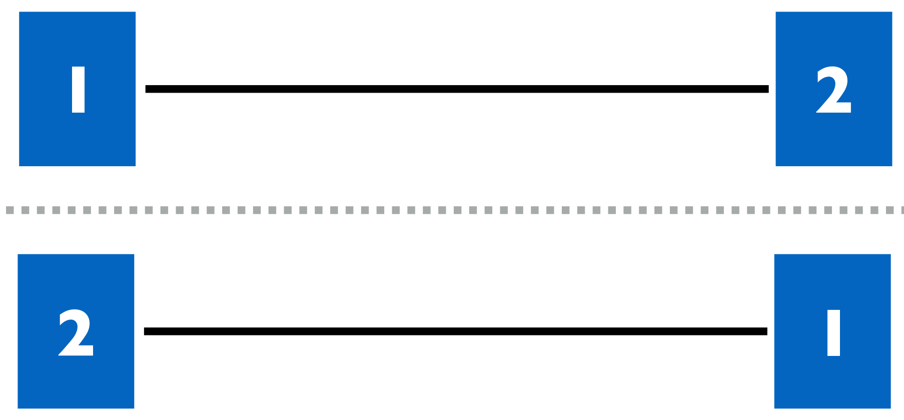
```
const adjList = {
  1: [2, 3],
  2: [1, 4],
  3: [1],
  4: [2]
}
```

Less space, more time to look up an edge

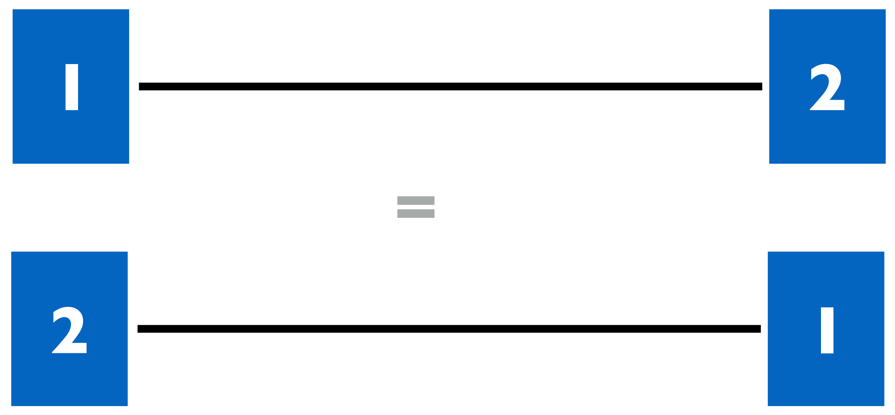


Okay, let's get back to talking about what graphs actually are…

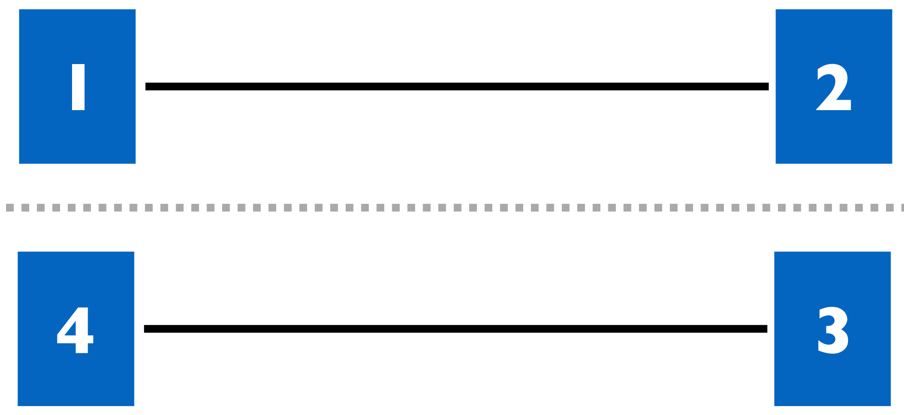
Are these graphs the same?

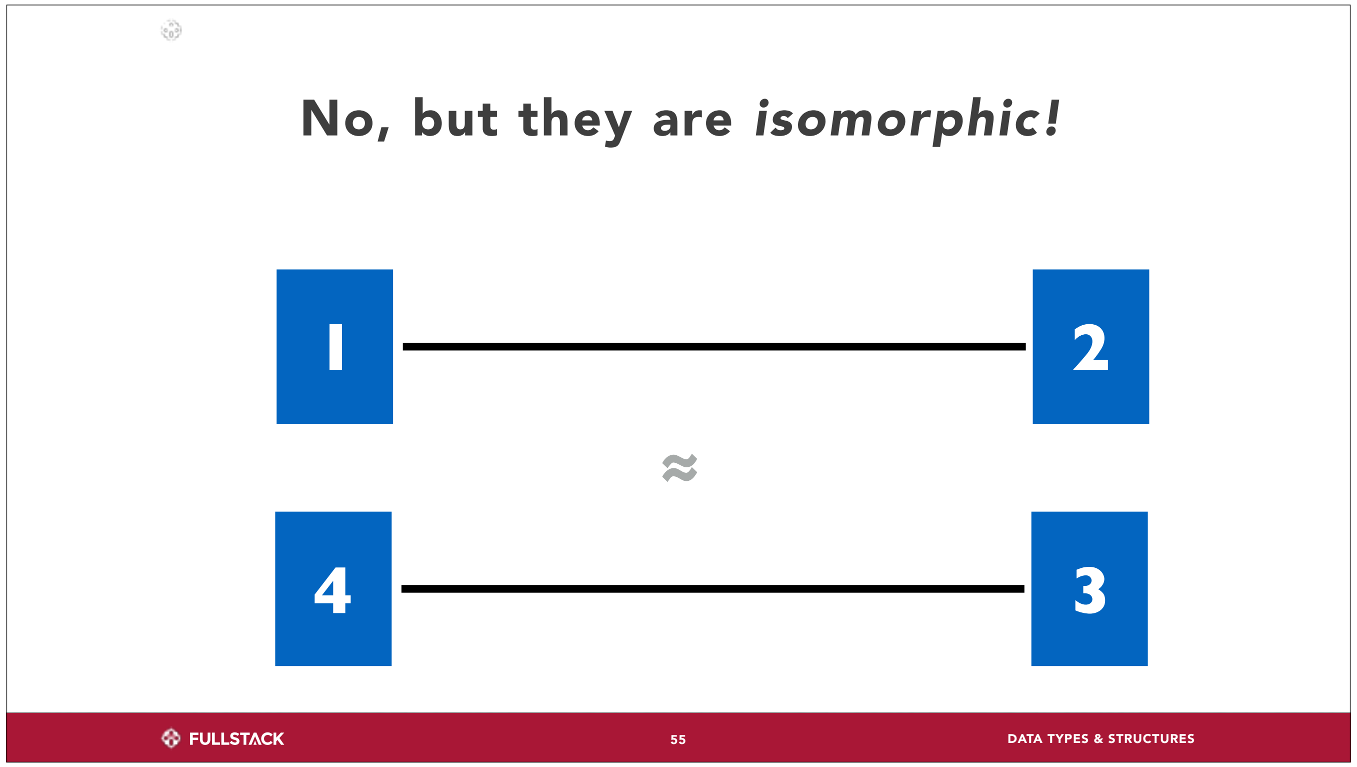


Yes!

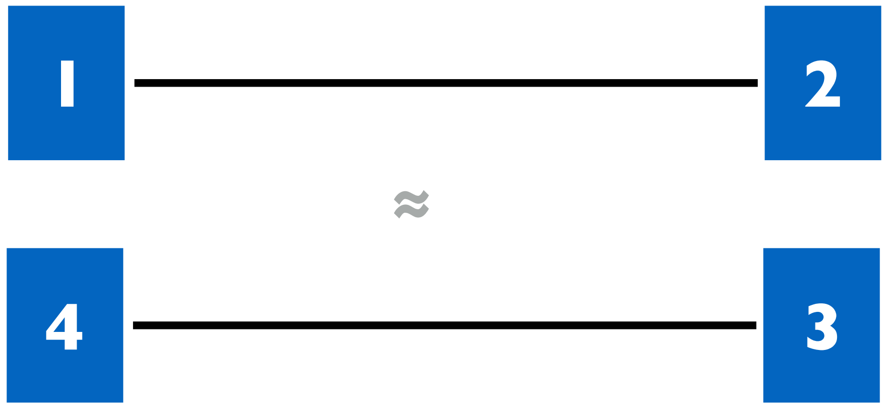


Are these graphs the same?





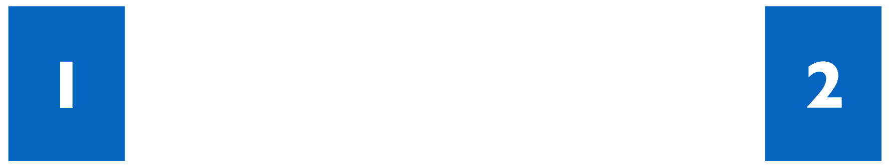
No, but they are *isomorphic!*



Corresponding edges and vertices

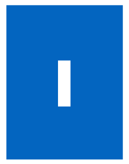
Isomorphism is an interesting focus in graph theory, but we won’t deal with it much

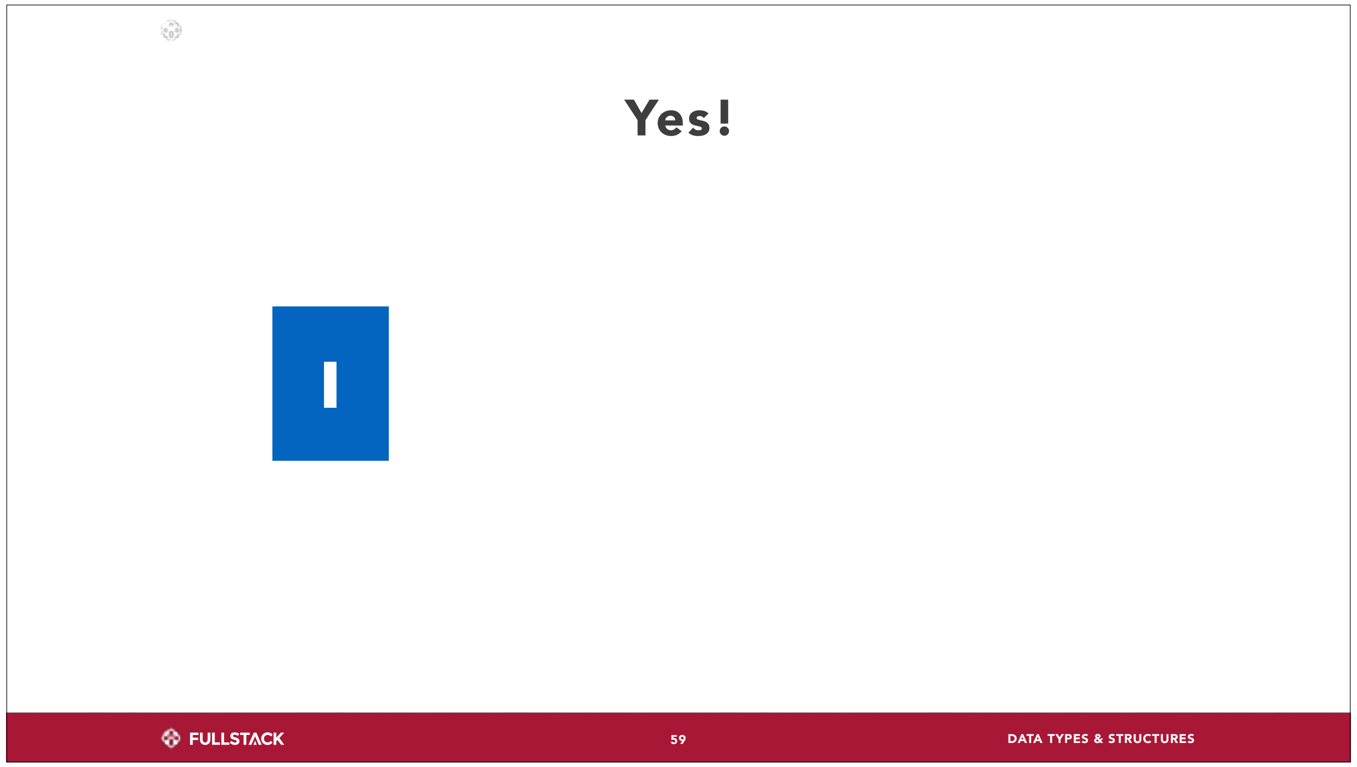
Is this still a graph?



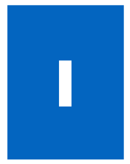
Yes!
Nodes may be “connected” or “disconnected"

Is this still a graph?





Yes!



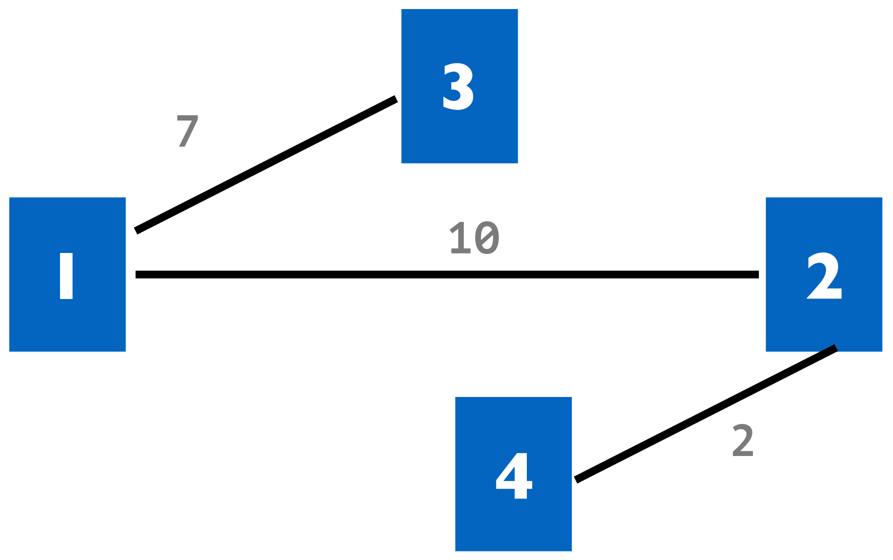
…and here's why…

Graph definition

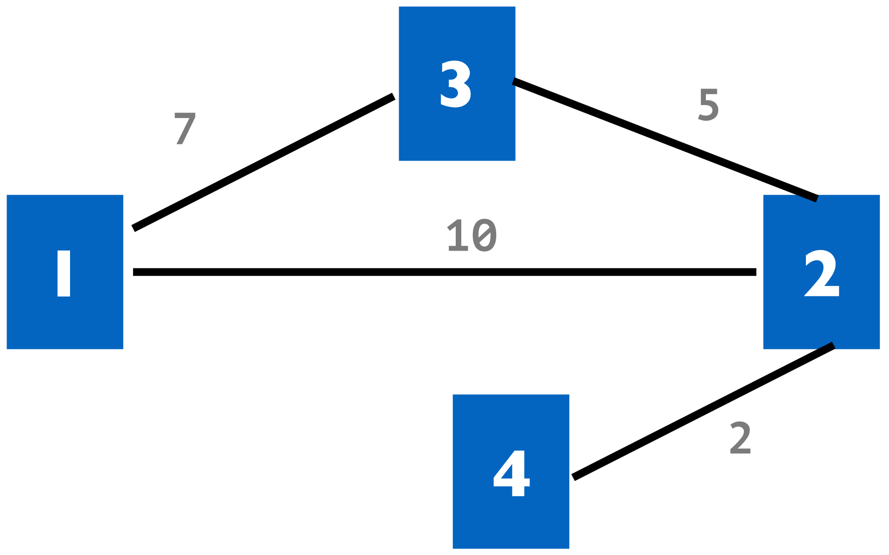
```
A graph is an object consisting of two sets called
its vertex set and its edge set. The vertex set is
a finite nonempty set. The edge set may be empty,
but otherwise its elements are two-element subsets
of the vertex set.
```

Trudeau, Richard J.. Introduction to Graph Theory (Dover Books on Mathematics) (Kindle Locations 425-427). Dover Publications. Kindle Edition.

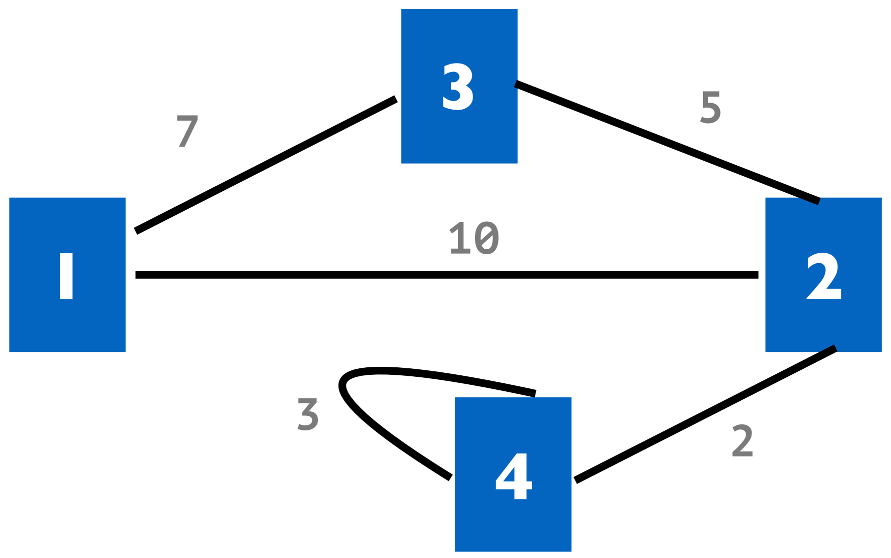
"Weighted" Graphs vs "Unweighted"



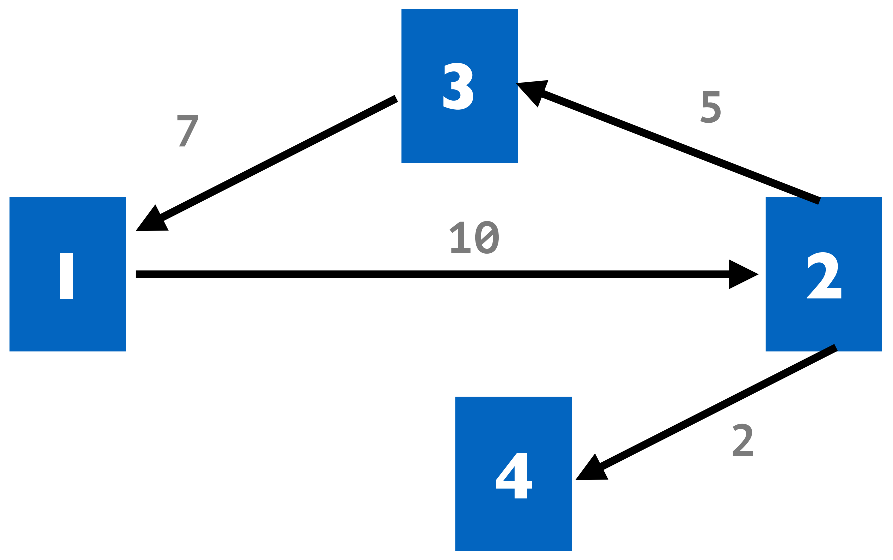
"Cyclic" Graphs



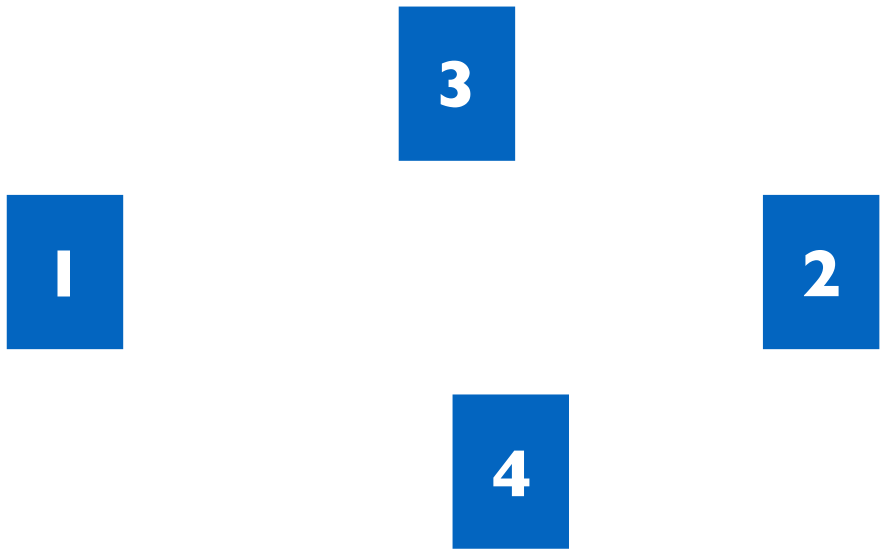
(a node can even cycle on itself)



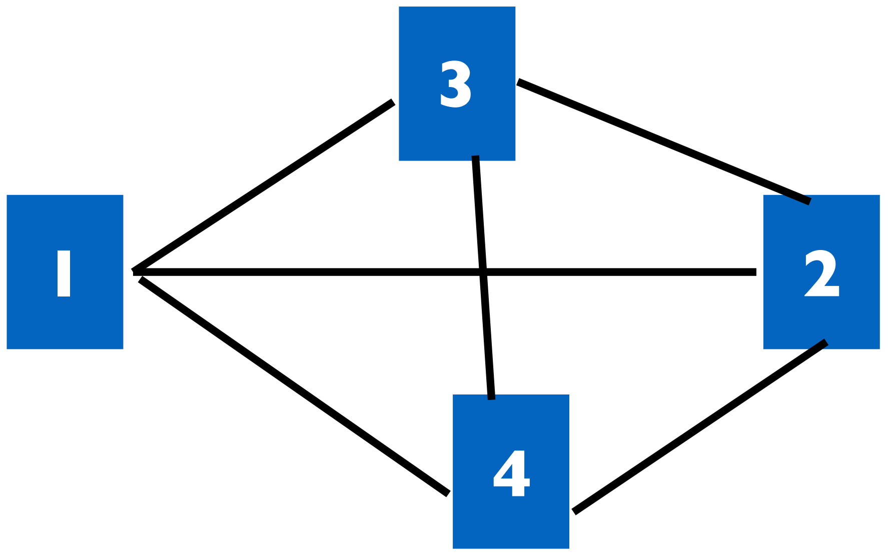
"Directed" Graphs

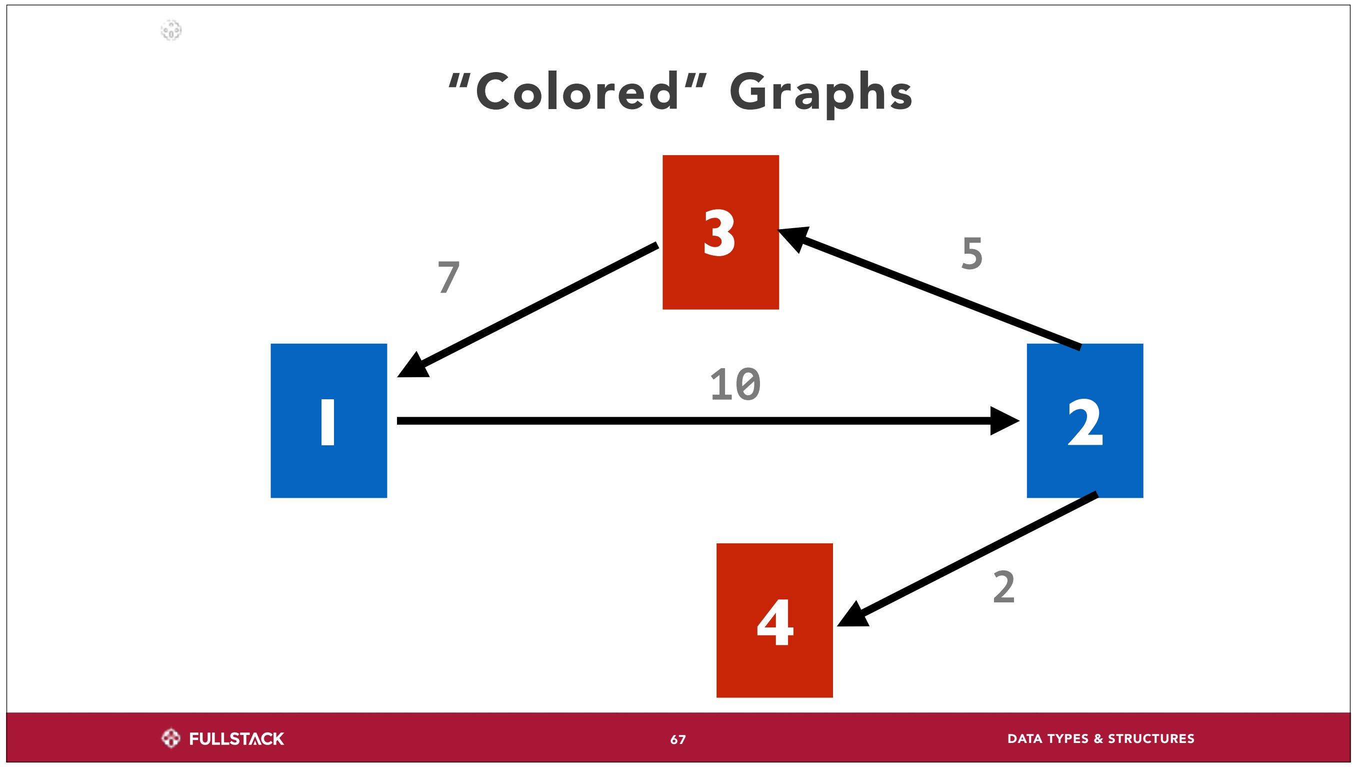


“Sparse” Graphs

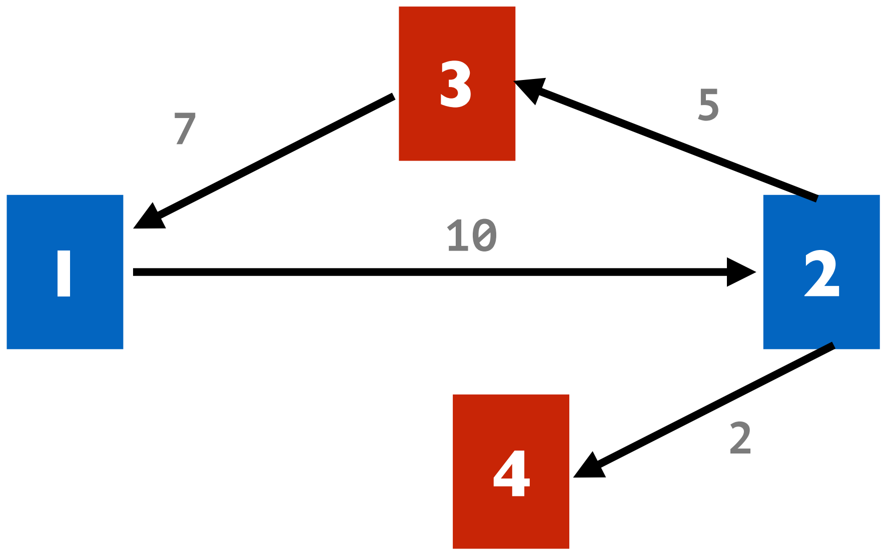


"Complete" Graphs



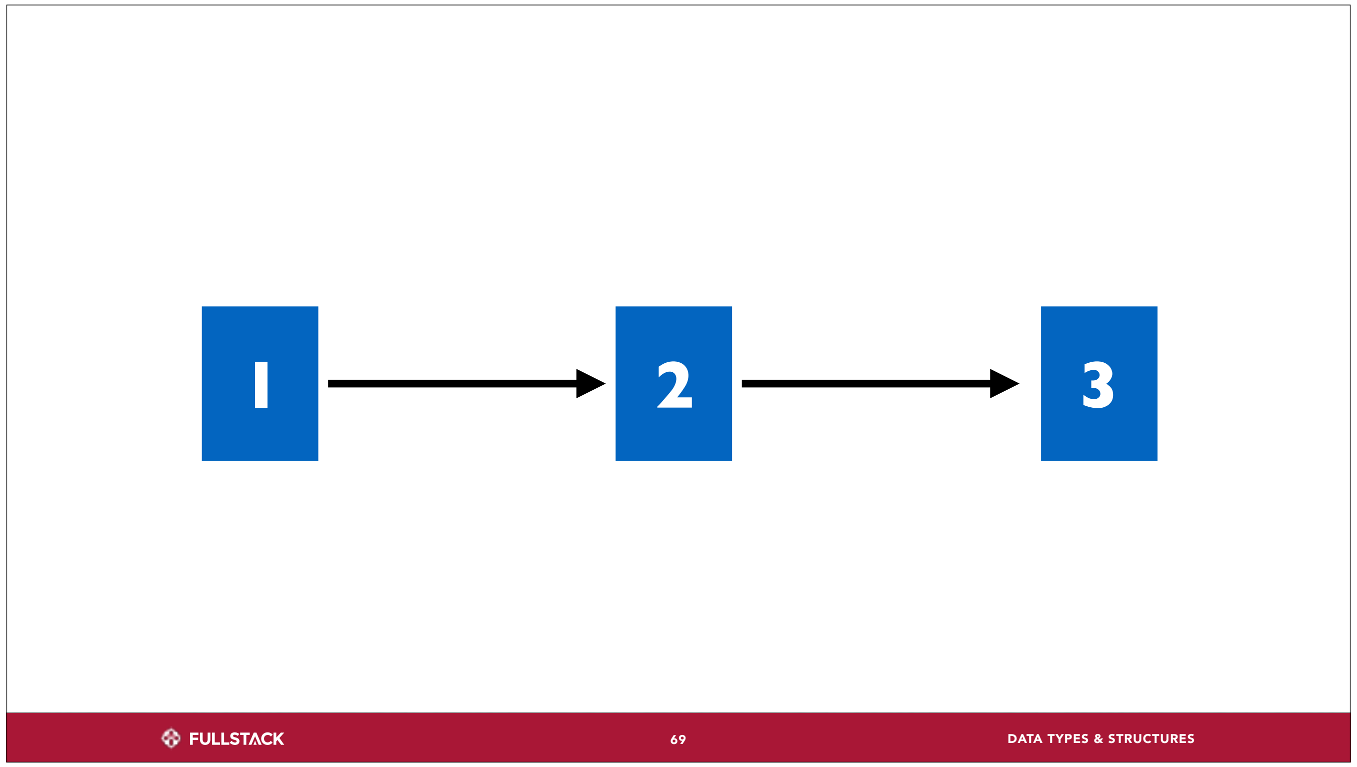


"Colored" Graphs

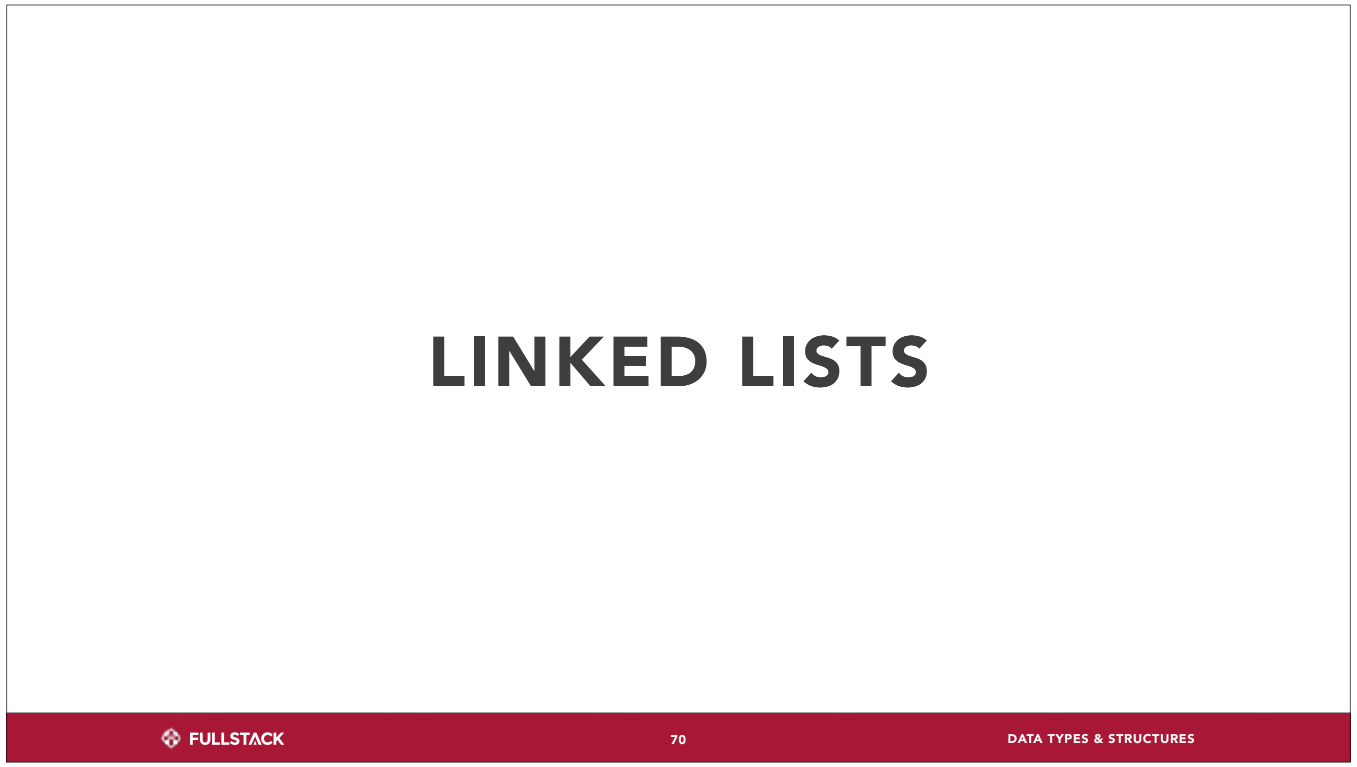


"can a graph be colored so that no node is connected to one of the same color?"

Graphs + Some Rules = Basically Everything Else



For example, what does this look like?



LINKED LISTS

Special case of a directed graph

Linked Lists

- Pointer to head node
- Each node has a value and pointer to the “next” node
- Can be “singly” linked or “double” linked
- Analogy: Scavenger hunt

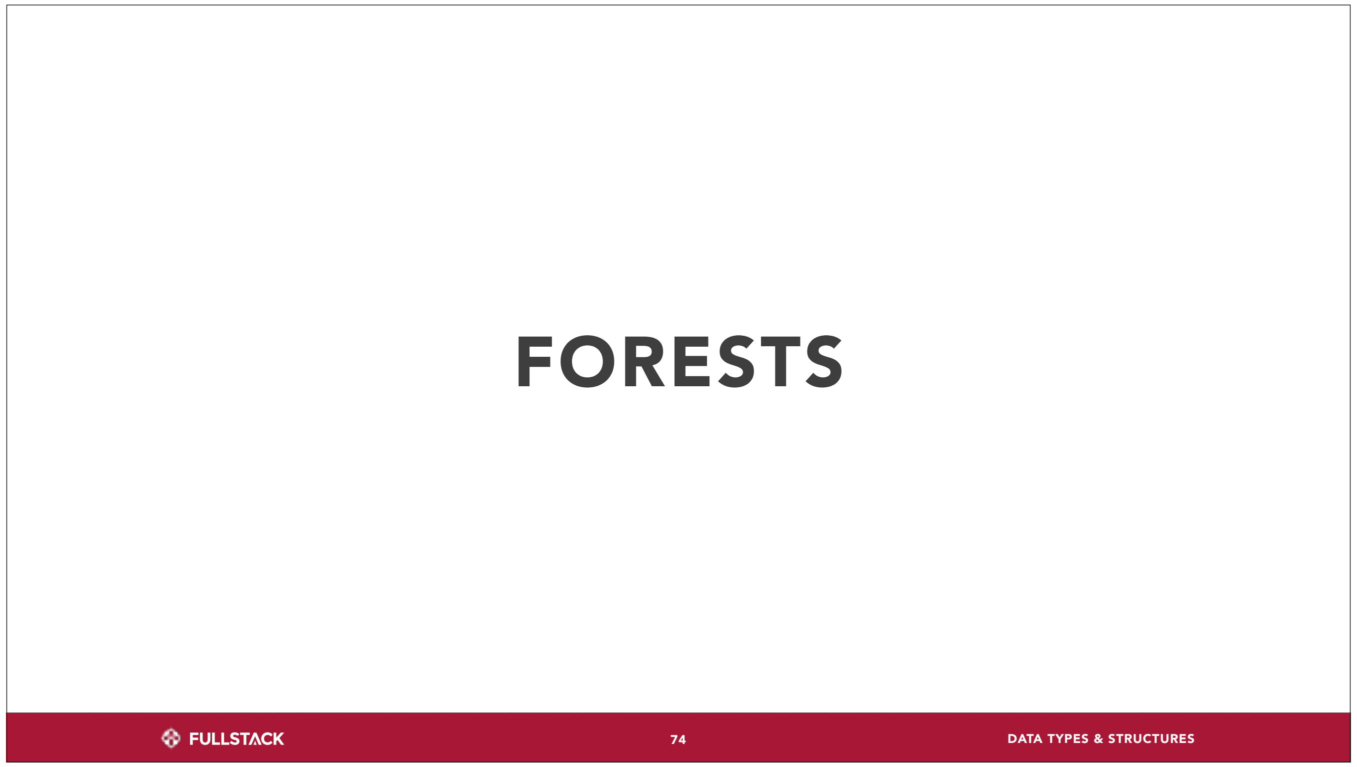
```
class LN {
  constructor (value, next = null) {
    this.value = value
    this.next = next
  }

  add (value) {
    this.next === null
      ? this.next = new LN(value)
      : this.next.add(value)
  }
}
```

TREES

Trees

- Directed, a-cyclic graphs
- Root with children (branches)
- Branches may have other branches, or end in terminal nodes
- Operation strategies: breadth-first and depth-first
- Analogy: ….um, a tree?

```
class Tree {
  constructor (value, branches = []) {
    this.value = value
    this.branches = branches
  }

  depthFirst (callback) {
    callback(this.value)
    this.branches.forEach(branch => {
      branch.depthFirst(callback)
    })
  }
}
```



FORESTS

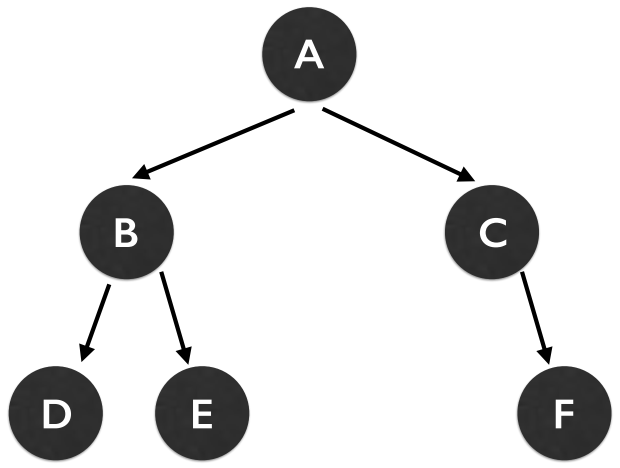
Trees are directed any

…just kidding, let's talk about different types of trees

Trees are directed any

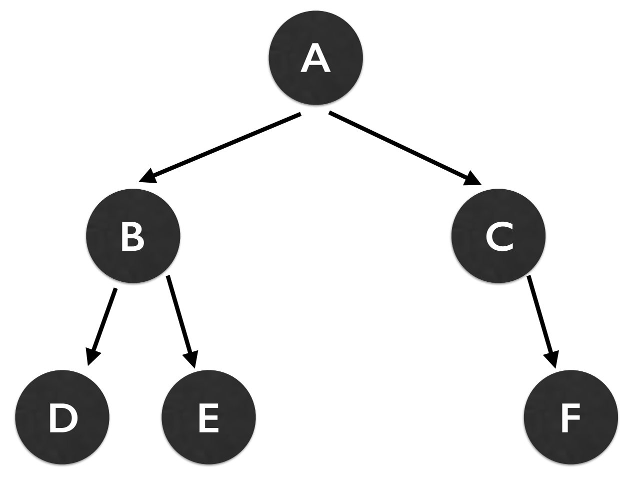
Binary Tree

- Every node has at most 2 branches



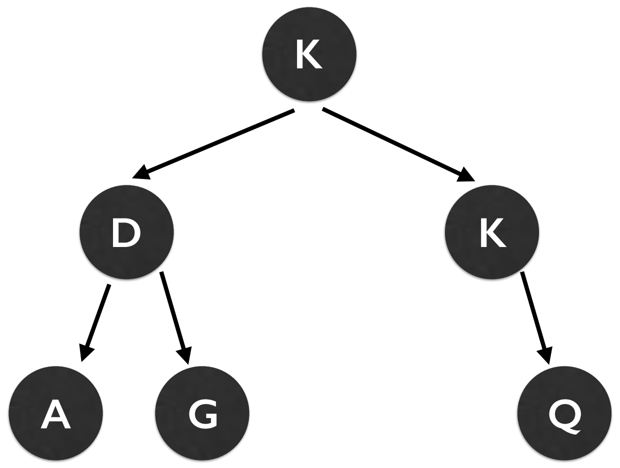
N-ary Tree

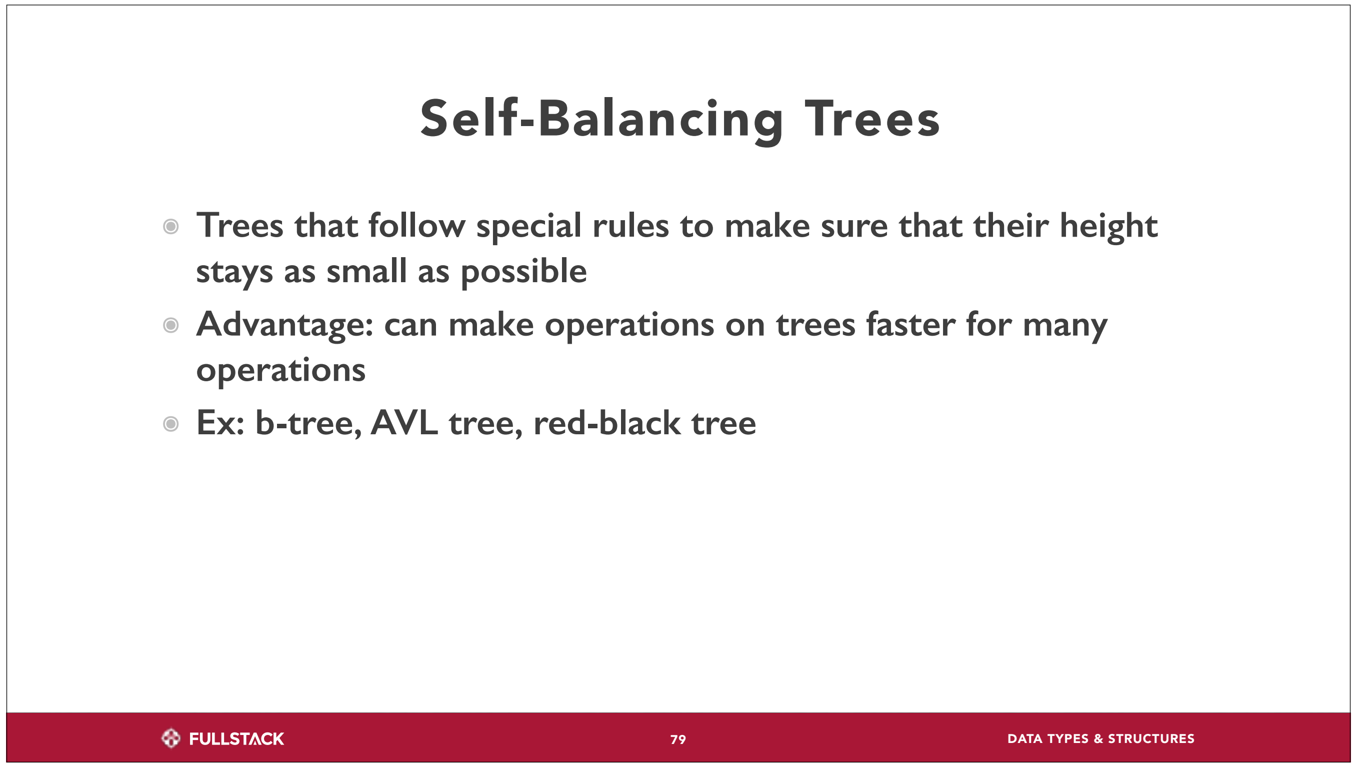
- Every node has at most n branches
- A binary tree is an n-ary tree where n = 2



Binary Search Tree

- A binary tree where each node satisfies an ordering
- left < node <= right


$$\begin{tikzcd}
 & & K \arrow[dll] \arrow[dr] & \\
D \arrow[d] \arrow[dr] & & & K \arrow[d] \\
A & G & & Q
\end{tikzcd}$$


Self-Balancing Trees

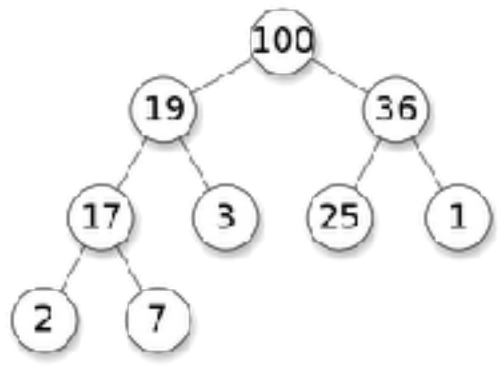
- Trees that follow special rules to make sure that their height stays as small as possible
- Advantage: can make operations on trees faster for many operations
- Ex: b-tree, AVL tree, red-black tree

Many operations on BSTs take time proportional to the height of the tree

HEAPS

HEAPS

- Special case of a tree
- Parents are either *greater than* (*“max heap”*) or *lesser than* (*“min heap”*) the children
- No ordering of siblings (only parent-child relationships are ordered)
- Warning: the term “heap” is used both to describe this DS, and to describe the memory area allocated for runtime variables - they’re totally unrelated!

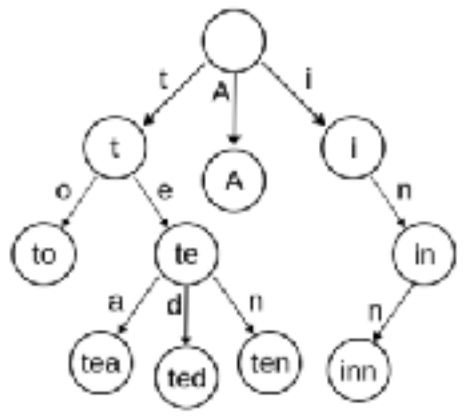


Use case: priority queues, sorting (‘heapsort’), any time you need the min or max of something frequently

TRIES

Tries

- Essential a tree for storing strings
- Split by character
- Each character is a node that points to the next
- Use case: predictive autocomplete
- Employs structural sharing



Tries

```
const trie = {
  ‘h’: {
    ‘e’: {
      ‘y’: ‘’,
      ‘l’: {
         ‘l’: {
           ‘o’: ‘’
         }
      }
    }
  }
}
```

What We've Covered

- Arrays
- Stacks
- Queues
- Hash Tables
- Graphs
- Linked Lists
- Trees
- Heaps
- Tries

IT'S GONNA BE OKAY

Some Nice Review Material

- **InterviewCake**
  - https://www.interviewcake.com/
- **Graph Representation at Khan Academy**
  - https://www.khanacademy.org/computing/computer-science/algorithms#graph-representation
- **Visualgo**
  - https://visualgo.net/en
- **Various DS's implemented in JS**
  - https://github.com/eyas-ranjous/datastructures-js