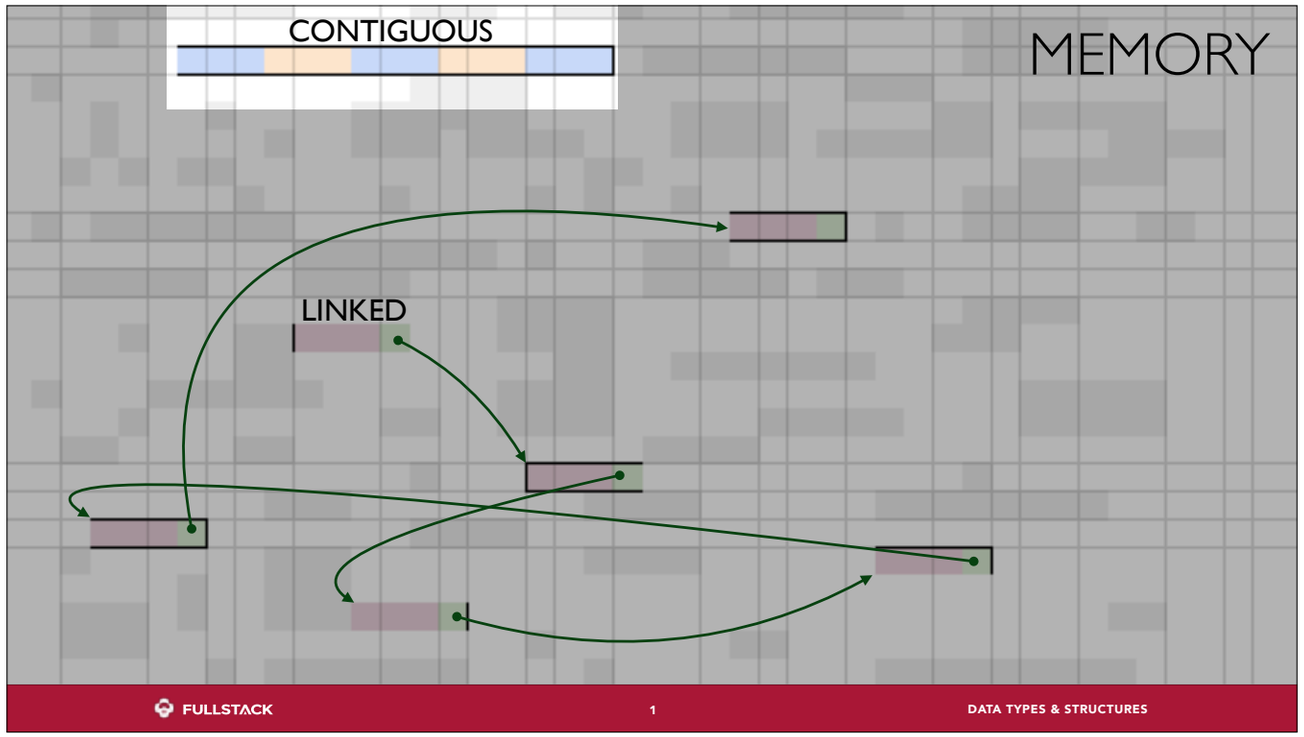


Data structures can typically be stored either in one solid block (contiguous) or in many small blocks, each with a memory address for the next block (linked). What are some pros and cons of each approach? Contiguous pros: good physical locality of memory means faster in practice. No wasted space for memory addresses. Linked pros: infinite adding so long as there are open spaces big enough for element + address.

DS is concerned with physical locality in the machine. The memory itself.
ADT is a specification for how something should behave

TRAJECTORY

- Memory allocation and Contiguous Arrays
- Stacks
- Data Structures vs Abstract Data Types
- Queues

When your code is running, all of the variables and data you use are stored in memory, or RAM. I like to think of RAM as a restaurant with billions of tables - each table can seat 8 bits, has a certain id or address that the house uses.

Metaphor: making reservations at a restaurant
Your app is like a diner/customer
The OS is like the front of house
Memory cells are like tables

Your application

Operating System

Memory

1	2	3	4
5	6	7	8
9	10	11	12

Your application

Operating System

Memory

1	2	3	4
5	6	7	8
9	10	11	12

int[4]; …I mean, I'd like to reserve 4 blocks of memory, please!

Your application

Operating System

Memory

1	2	3	4
5	6	7	8
9	10	11	12

I'll see what we have available!

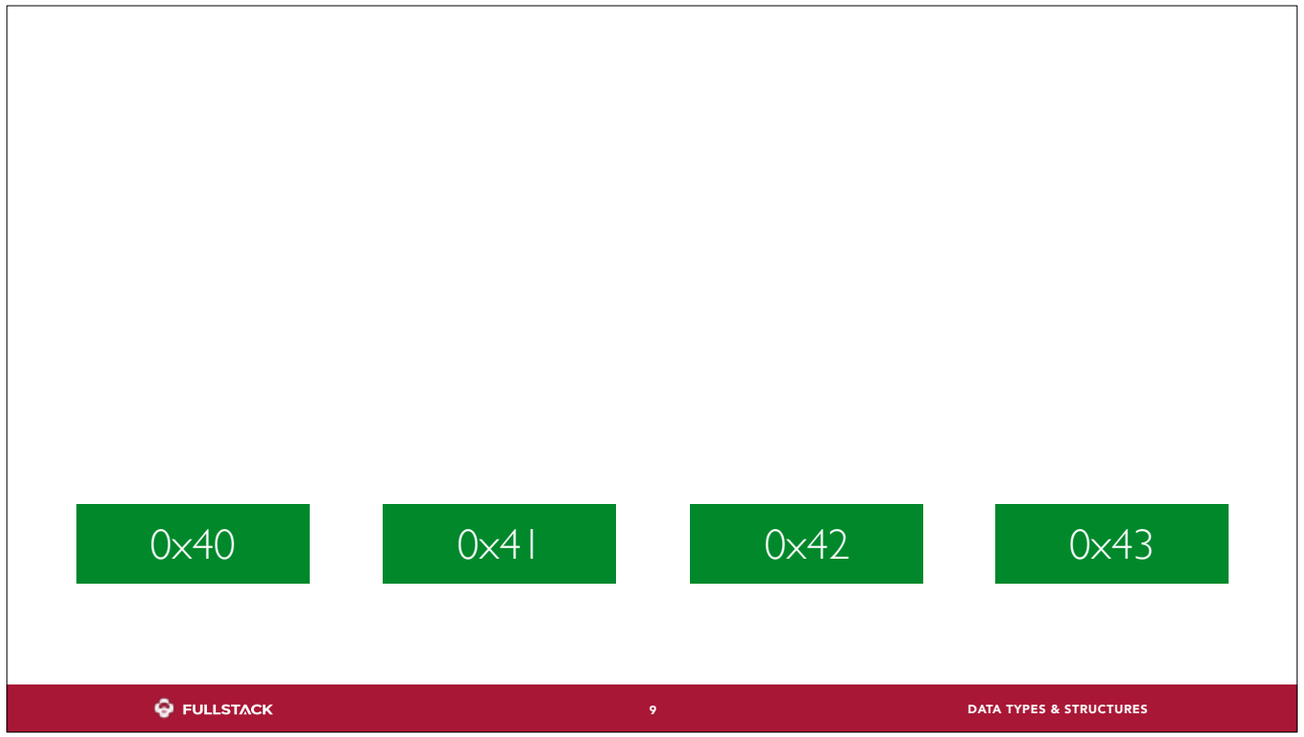
Your application

Operating System

Memory

1	2	3	4
5	6	7	8
9	10	11	12


CONTIGUOUS ARRAY



When referring to memory, we often use hexadecimal, or base 16.

This is because referring to memory using 0s and 1s would be a mess! But hex is very easy to convert back and forth to binary. Much easier than binary to decimal and back.

```c
int myArr[4]; //myArr is 4 bytes starting at 0x40
```

```c
int myArr[4]; //myArr is 4 bytes starting at 0x40
```

```
int myArr[4]; //myArr is 4 bytes starting at 0x40
int myArr[0] = 9; // put `9` at 0x40 + 0
```

```
int myArr[4]; //myArr is 4 bytes starting at 0x40
int myArr[0] = 9; // put `9` at 0x40 + 0
int myArr[2] = 3; // put `3` at 0x40 + 2 (Ox42)
```

```
int myArr[4]; //myArr is 4 bytes starting at 0x40
int myArr[0] = 9; // put `9` at 0x40 + 0
int myArr[2] = 3; // put `3` at 0x40 + 2 (Ox42)

int myArr[2] // what is at 0x40 + 2 (Ox42)?
```


STACKS

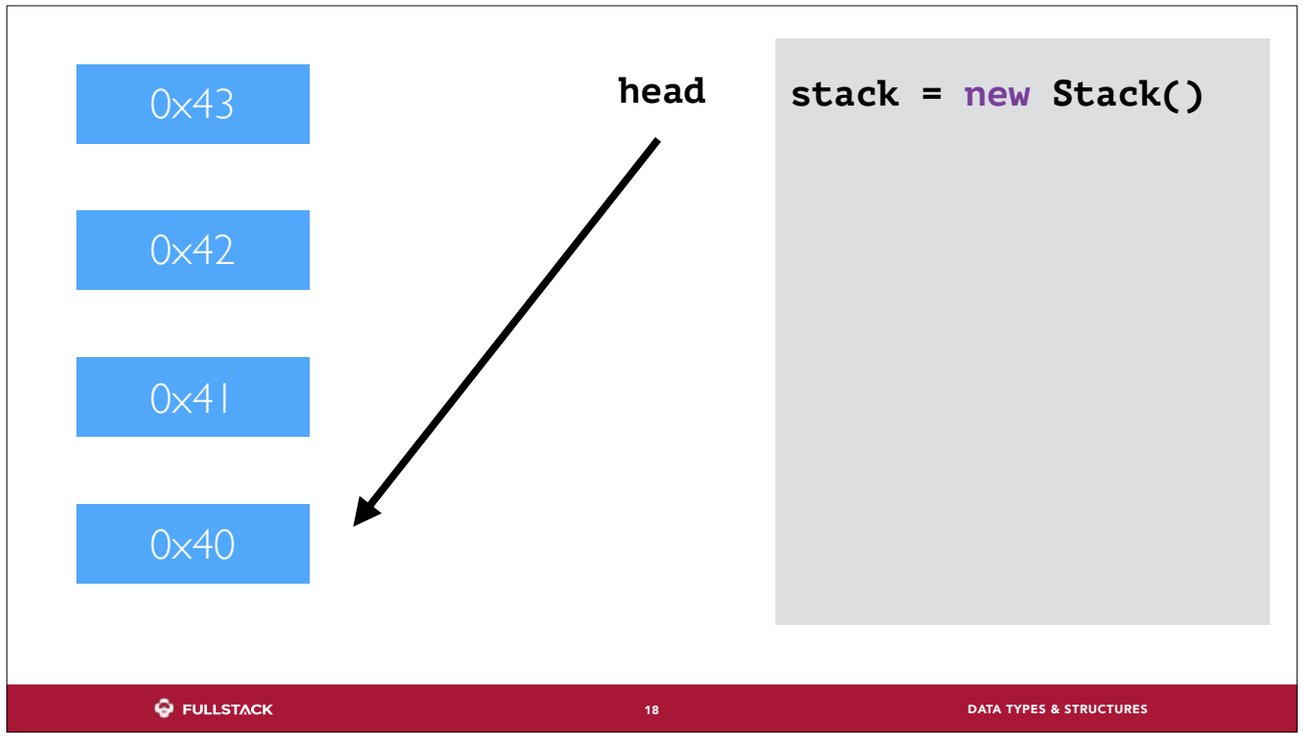
Stacks

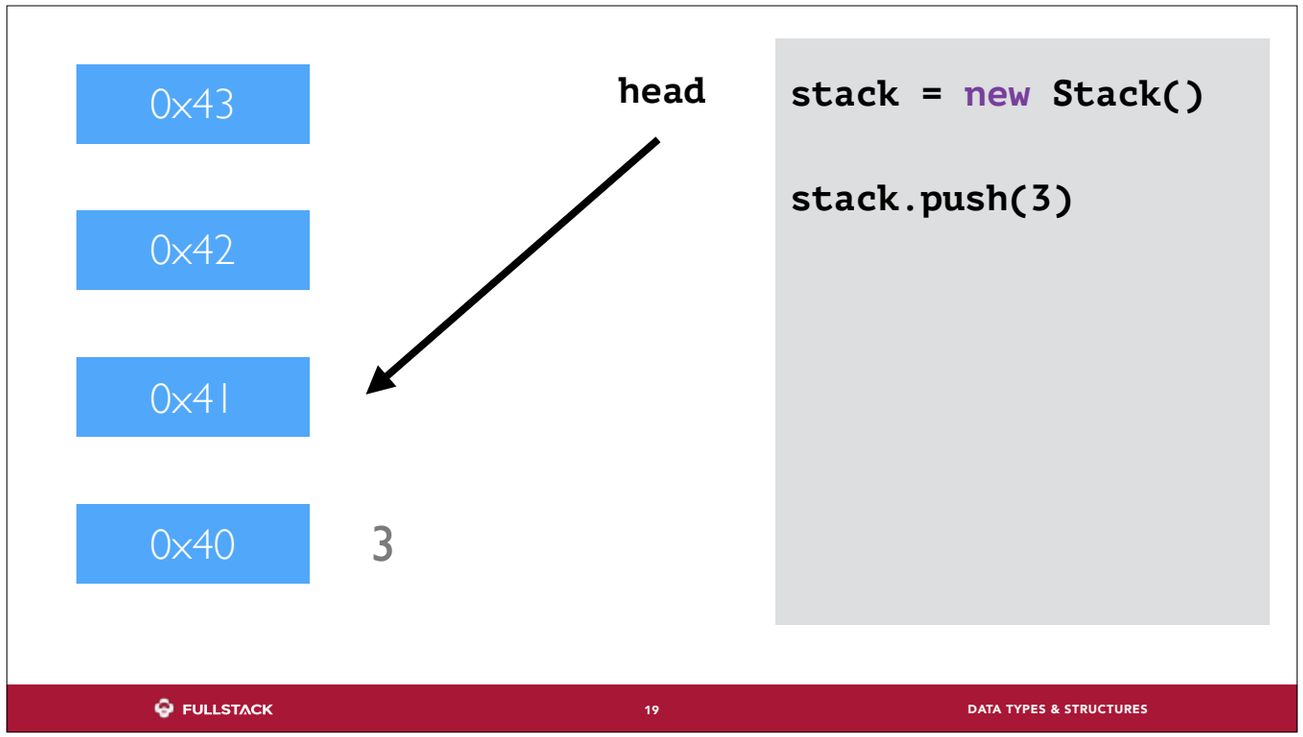
- Collection of elements
- Ordered
- Elements can repeat (not a set)
- Some example operations:
  - Make a new stack
  - Add an element to the stack ("push")
  - Retrieve an element from the list ("pop") - *must be LIFO (Last In, First Out)*
  - Check if stack is empty
  - Look at the top element without removing it ("peek")
  - Clear the stack

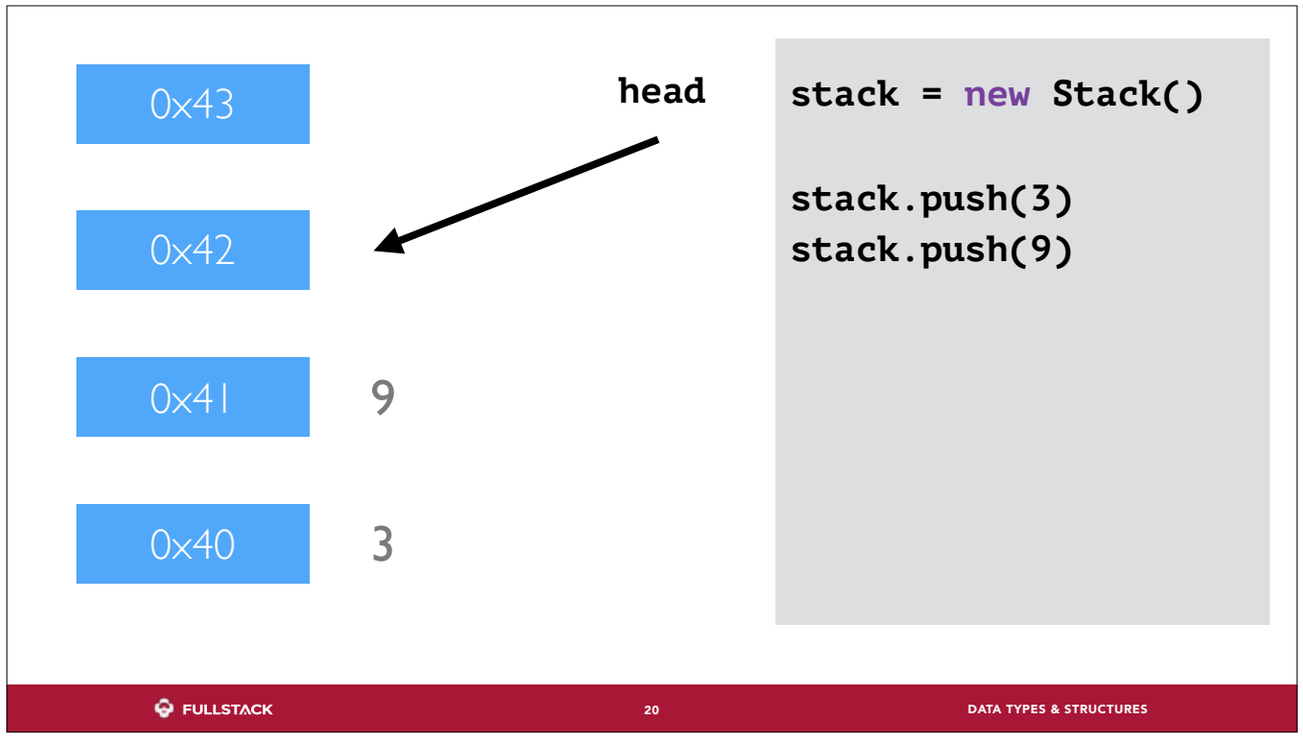
Stacks are useful for a couple key applications in programs. Remember the RPN calculator? Also very useful for backtracking (command-z, anybody?), holding old values waiting to be processed (*ahem*, call stack or recursion), etc.

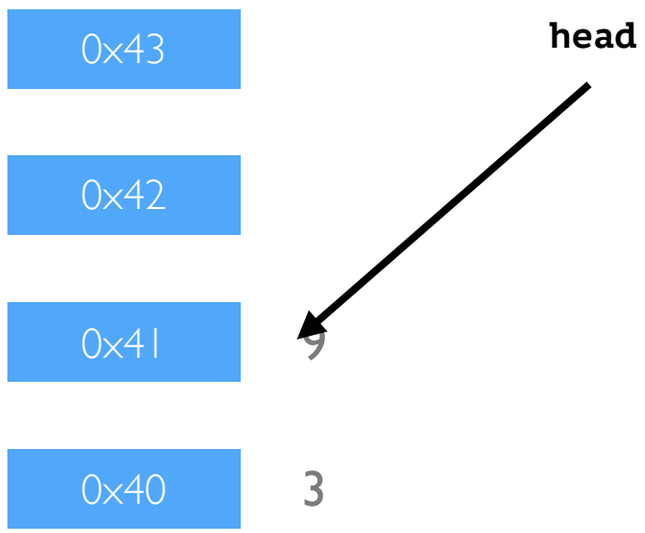
Stack of cards - not code, but still a stack. Because a stack is a ADT as opposed to a DS.


0x43
0x42
0x41
0x40


0x43
0x42
0x41
0x40
head
stack = new Stack()


0x43
0x42
0x41
0x40
head
3
stack = new Stack()
stack.push(3)


0x43
0x42
0x41
0x40
head
9
3
stack = new Stack()
stack.push(3)
stack.push(9)




```
stack = new Stack()

stack.push(3)
stack.push(9)
stack.pop() // 9
```

Data Structures vs Abstract Data Types

So now we've looked at these two concepts - stacks and a contiguous array. A stack is more abstract - it describes data operations, like how to push and pop, and we saw that the way we implemented it was with a contiguous array and a "head" variable. This is the difference between ADTs and Data Structures

Abstract Data Types

- **Describes *how* information is *related***
  - Ex. is the information ordered in some way? Are elements connected together?
- **Describes operations we can perform on that information**
  - Ex. add, remove find

Data Structures

- Concrete, programmatic *implementations* of an ADT
- Describe how information is actually stored in memory
- Determines how performant operations are

The Stack ADT & Array DS

Stack Feature / Operation	Array Implementation with `top`
Collection of elements	Store values at memory addresses
Ordered	Sequential addresses maintain ordering
Create new stack	Initialize a new array
Push onto stack	Insert value at `top` var (next index to use)
Pop off of stack (LIFO)	Use `top` index to return latest value
Check if stack is empty	Return whether `top` variable is 0

By doing some minimal accounting using a variable (like `top`), we can implement the Stack ADT using a true array fairly easily. In fact, the `push` and `pop` methods of a JavaScript array basically use its `.length` property in a similar way. Thought: how performant is pushing and popping using an array to implement our stack? Well, we know that accessing an array element by index (either reading or writing) is a constant-time operation (i.e., it doesn't matter which index we use, it will always take the same time give or take). That's excellent!

ADTs vs DSs

Common Abstract Data Types	(Some) Data Structures
Set	List, Linked Tree, Trie, Hash Table, etc.
List	Linked List, Array
Stack	Array, Linked List
Queue	Linked List, Deque
Map (Associative Array / Dictionary / etc.)	Hash Table, Association List, Red-Black Tree
Graph	Adjacency List, Adjacency Matrix
Tree	Linked Tree, Heap

An ADT is a description of:
1. how information is related (e.g. ordered elements, connected elements)
2. operations we can perform on that information (e.g. add, remove, find)

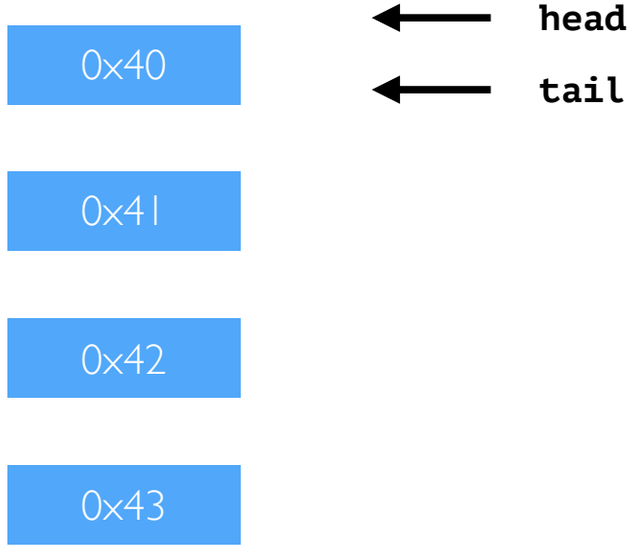
A DS is a concrete, programmatic implementation of an ADT that determines:
1. how performant the operations actually are


QUEUES

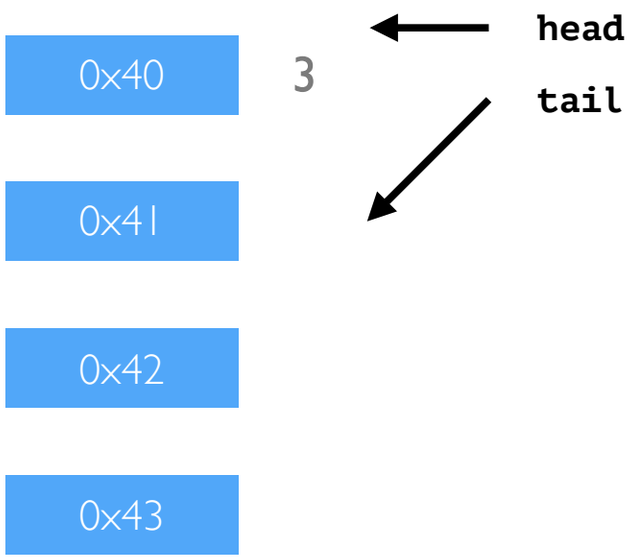
The Queue ADT

- Like Stack, except FIFO (First In, First Out)
- Collection of elements
- Ordered / sequential
- Operations:
  - Enqueue (add)
  - Dequeue (remove)
  - Peek
  - Clear
  - IsEmpty… etc.

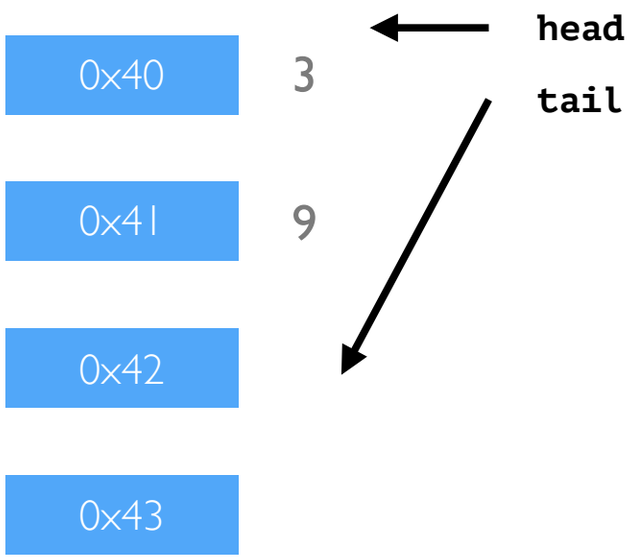
Queues are a natural fit for handling incoming values in a fair way (buffering). In the workshop today you will be implementing the Queue ADT, initially using JavaScript's "array" fundamental type. However, for the academic practice, treat those arrays like "true" arrays and do not use any of the built-in JS helper functions on `Array.prototype`. That means no push, pop, shift, unshift; also, do not use `.length`. The only thing you really have access to is indices. Note that since Queues are an ADT, not a DS, that implies there is more than one way to skin a cat.



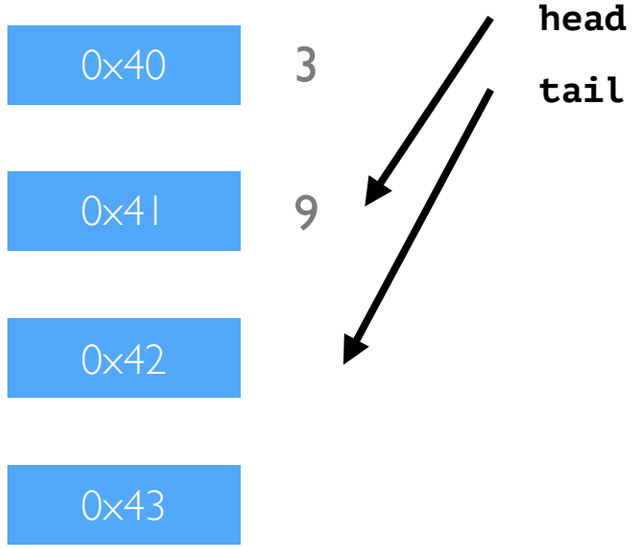

```
queue = new Queue()
```



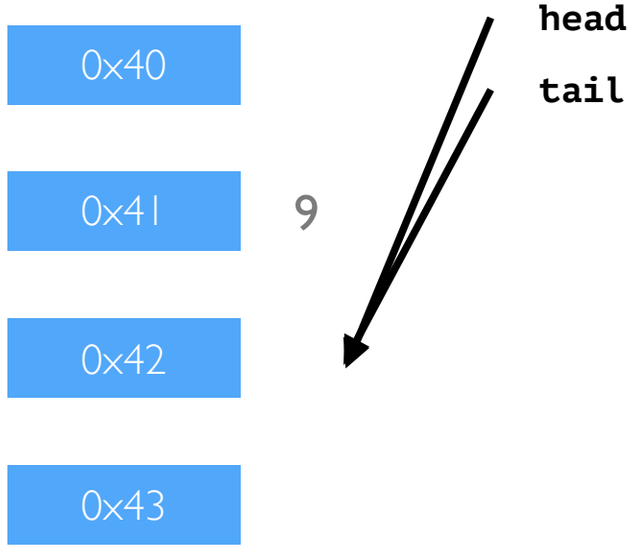

```
queue = new Queue()

queue.enqueue(3)
```




```
queue = new Queue()

queue.enqueue(3)
queue.enqueue(9)
```




```
queue = new Queue()

queue.enqueue(3)
queue.enqueue(9)

queue.dequeue() // 3
```




```
queue = new Queue()

queue.enqueue(3)
queue.enqueue(9)

queue.dequeue() // 3
queue.dequeue() // 9
```

WHAT ABOUT JS ?

In JavaScript

- **Arrays are NOT contiguous arrays**
  - Their implementation is left up to the JavaScript engine
  - ES6 - you can create typed arrays (ex. Uint8Array)
- **Simulate a stack**
  - push
  - pop
- **Simulate a queue**
  - `push` to enqueue
  - `shift` to dequeue
- **Data Structures take-home: implement a stack/queue *without* using these methods!**