Big O Notation

Time complexity of an algorithm

"How much time it takes to run a function as the size of the input grows."

Runtime

array

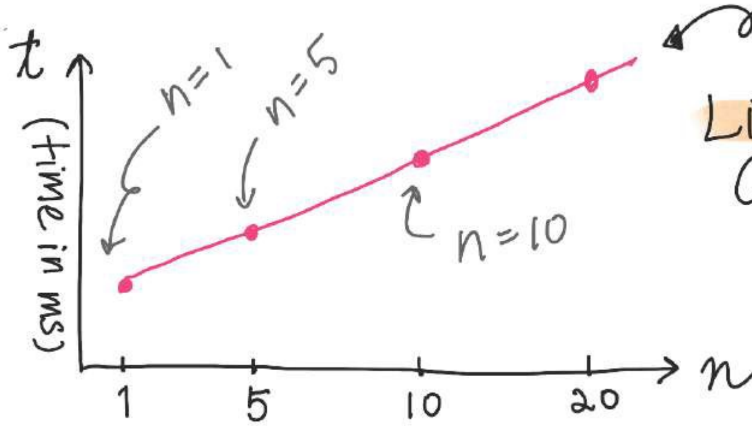
Const array1 = [, , , ,] number of elements n=5

Let's see if there is a needle in the haystack!

```js
Const numNeedles = (haystack, needle) => {
  let count = 0
  for (let i = 0; haystack.length ; i++) {
    if (haystack[i] === needle) count += 1;
  }
  return count;
}
```

How long does it take to execute when the number of elements (n) is:

♥ execution time grows linearly as array size increases!



this is Linear time Complexity = O(n)

@girlie_mac

@girlie_mac

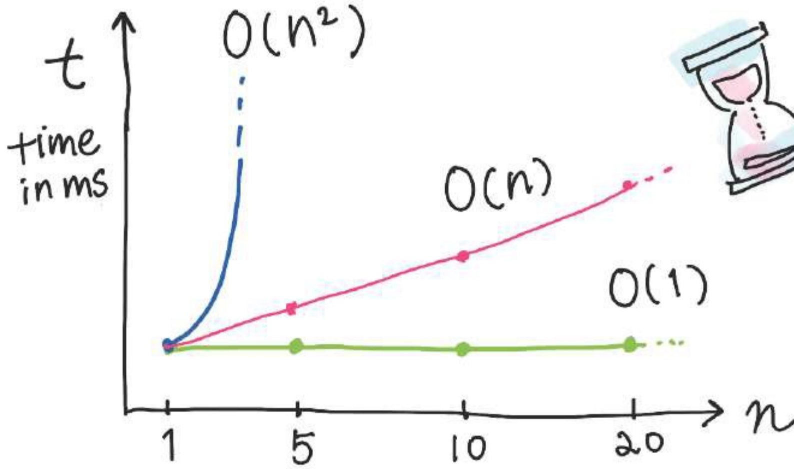
Big O Notation

Let's see if we have some function that doesn't actually loop the array:

```
const alwaysTrueNoMatterWhat = (haystack) => {
  return true;
```

n=5
n=10
n=20
⋮

Array size has no effect on the runtime



☆ Constant time = O(1)

☆ Quadratic time = O(n²)

```
Const
array2 = [
```

```
];
```

n=5, however the runtime is proportional to n^2

```
const hasDuplicates = (arr) => {
  for (let i = 0; i < arr.length; i++)
    let item = arr[i];
    if(arr.slice(i+1).indexOf(item) !== -1) {
      return true;
    }
  }
  return false;
}
```

① Loop thru the array

② Another array lookup w/ indexOf method

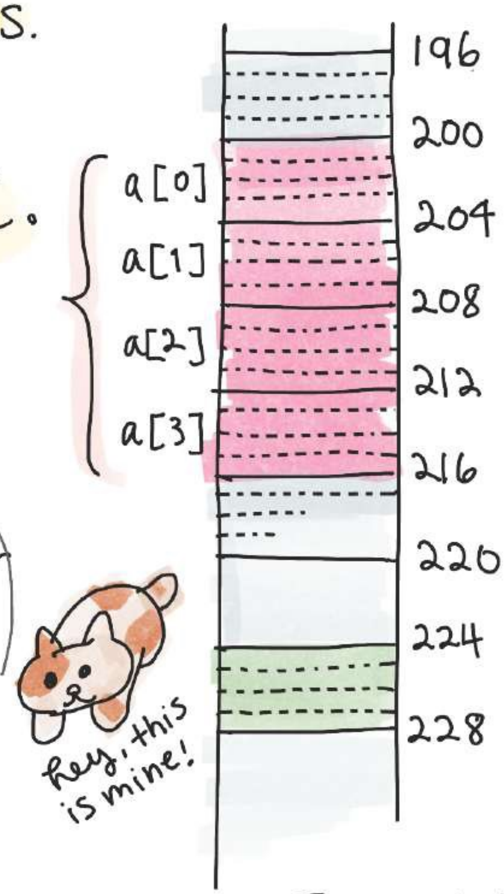
Data Structures

Array & Linked List

Array

a linear data structure, stored in contiguous memory locations.

memory



- Assume each cat is an integer = requires 4 bytes space
- The array of cats must be allocated contiguously!
  → address 200 — 216

yay!

- Can randomly access w/ index
  a[2] → cat
- contiguous = no extra memory allocated = no memory overflow

meh!

- Fixed size. Large space may not be avail for big array
  cat took the space!
- Insert + delete elements are costly.
  → may need to create a new copy of the array & allocate at a new adress.

@girlie_mac

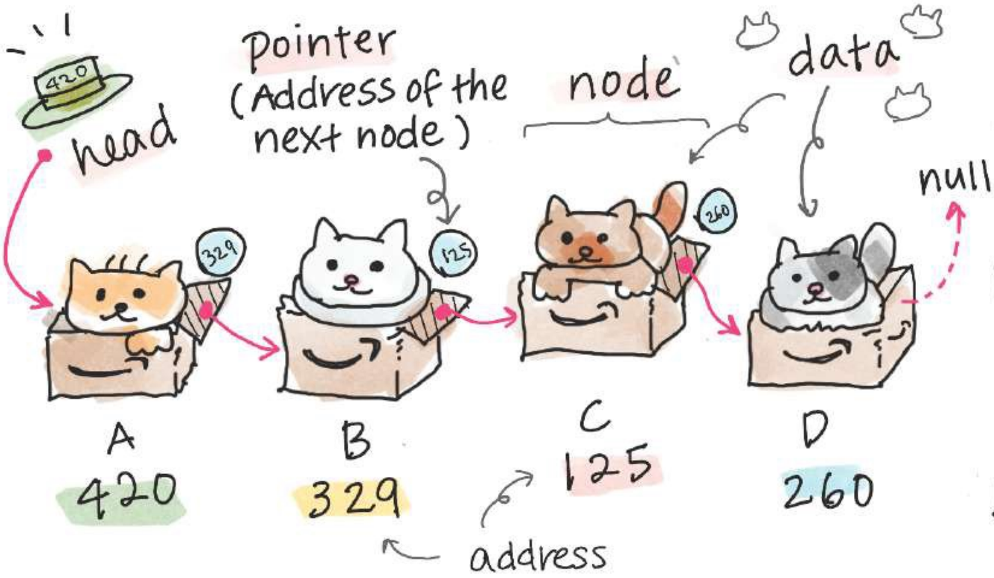
Data Structures

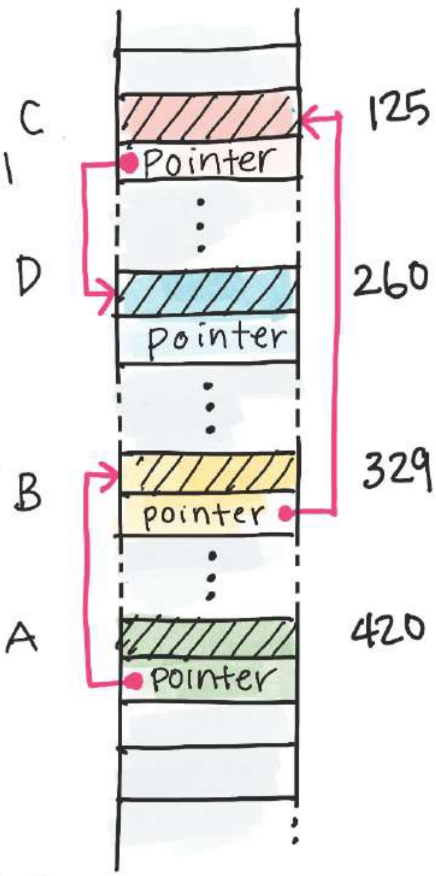
Array & Linked List

Linked list

- ★ a linear data structure
- ★ each element is a separated object & elements are linked w/ pointers

memory





★ Unlike an array, LinkedList elements are not stored in contiguous locations.

yay!

- ♡ Dynamic data = size can grow or shrink
- ♡ Insert & delete element are flexible.
  → no need to shift nodes like array insertion
- ♡ memory is allocated at runtime

meh!

- No random access memory.
  → Need to traverse n times
  → time complexity is O(n). array is O(1)
- Reverse traverse is hard

Data Structures

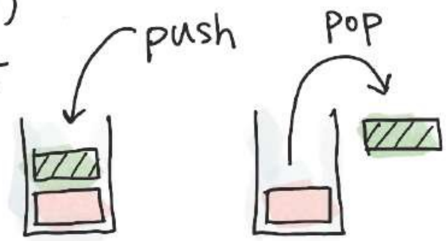
Stack & Queue

LIFO

FIFO

@girlie_mac

A stack is a LIFO (Last-in-First-out) data structure, where an element added last (=push) gets removed first (=pop)



just like a stack of ice cream scoope!

omg, the bottom one is always melting !!

Stack as an array in JS

arrays in JavaScript are dynamic!

```
let stack = [ ];                      stack is:
stack.push('mint choc');              // ['mint choc']
stack.push('vanilla');                // ['mint choc', 'vanilla']
stack.push('strawberry');             // ['mint choc', 'vanilla',
                                          'strawberry']
let eaten = stack.pop( );             // eaten is
                                          'strawberry'
                                          ['mint choc', 'vanilla']
```

Time complexity is O(1) for both pop + push.

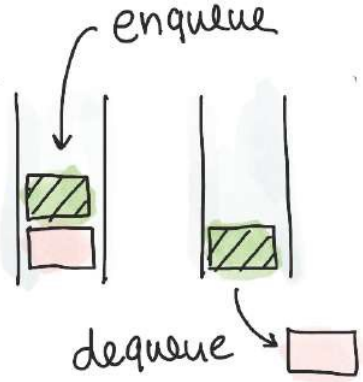
Data Structures

Stack & Queue

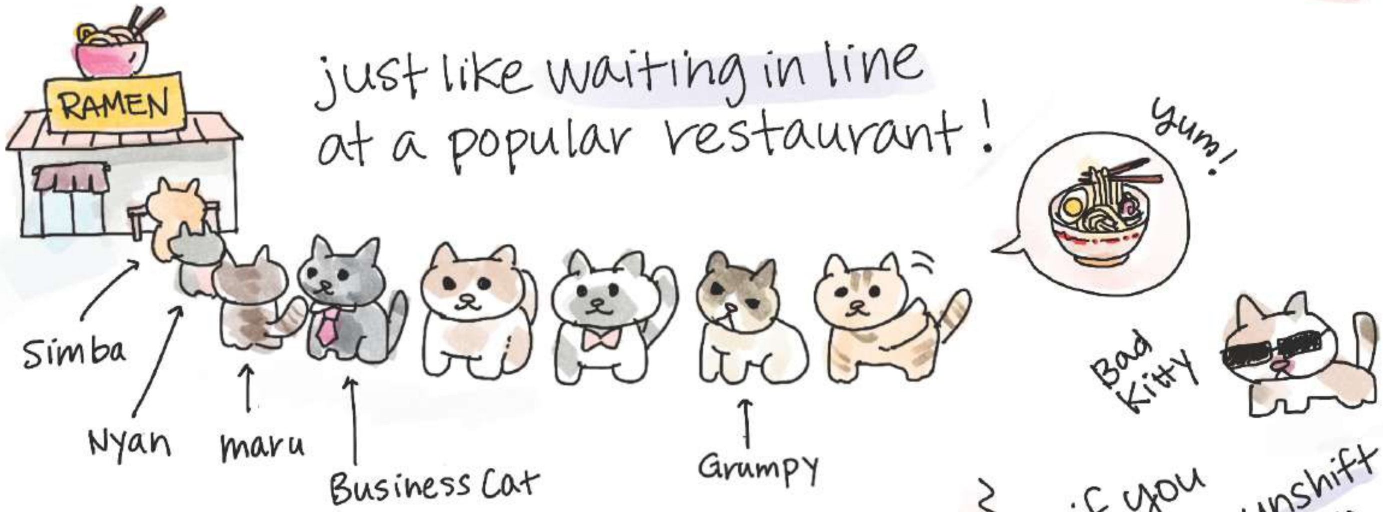
LIFO FIFO

@girlie_mac

A queue is a FIFO (First-in-First-out) data structure, where an element added first (= enqueue) gets removed first (= dequeue)



just like waiting in line at a popular restaurant!



Stack as an array in JS

```
let queue = [ ];                                  queue is:
queue.push('Simba');   // ['simba']
queue.push('Nyan');    // ['simba', 'nyan']
queue.push('maru');    // ['simba', 'nyan', 'maru']
let eater = queue.shift(); // eater is 'Simba'
                              queue is ['nyan', 'maru']
```

wrong! if you queue.unshift('bad kitty'), instead of push(), then the cat cuts in to the front of line!

Time Complexity should be O(1) for both enqueue + dequeue but JS shift() is slower!

Data Structures # Hash Table

- A hash table is used to index large amount of data
- Quick key-value look up. O(1) on average
  - ↳ Faster than brute-force linear search

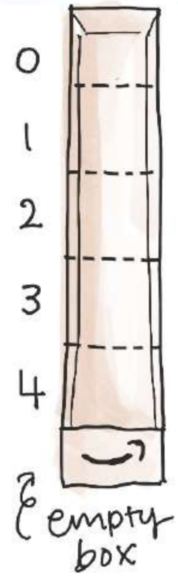
① Let's create an array of size 5.

We're going to add data.

key = "Tabby"

value = "pizza"

some data Let's say, favorite food!



② Calculate the hash value by using the key. "Tabby".

e.g. ASCII code, MD5, SHA1

→ # → 530

Hash

③ Use modulo to pick a position in the array!

→ # → 530 % 5 = 0

[0]

The hash is divided by the size of the array.

The remainder is the position!

④ Let's add more data.

Tux # → 353 % 5 = 3

Bob # → 307 % 5 = 2

Use the same method to add more

@girlie_mac

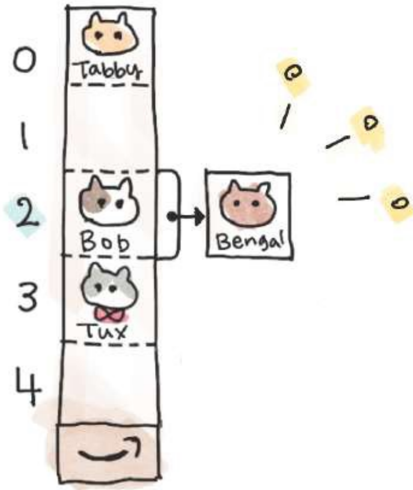
Hash Table

@girlie_mac

Collision!

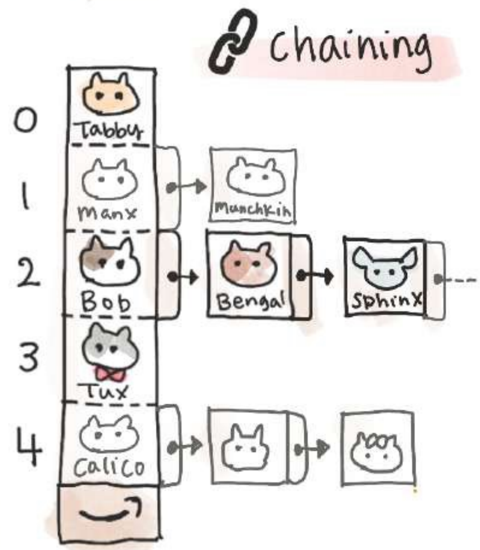
Now we want to add more data.
Let's add "Bengal".

"Bengal" → # → 617 % 5 = 2



But [2] slot has been taken by "Bob" already! = collision!
so let's chain Bengal next to Bob! = chaining

key: "Bengal"	"Sphinx"	keep adding data
Value: "Dosa"	"Fish + Chips"	



Searching for data

☆ Let's look up the value for "Bob"

① Get the hash → 307 ← O(1)

② Get the index → 307 % 5 = 2

③ Look up Array [2] → found!

☆ Let's look up "munchkin"

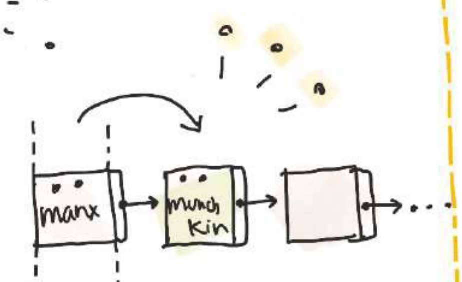
① Hash → 861

② Index → 861 % 5 = 1

③ Array [1] → "manx"

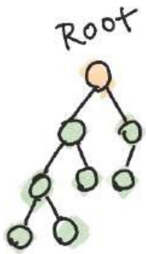
④ Operate a linear-search to find "munchkin"

↳ Average O(n)



@girlie_mac

Data Structures Binary Heap



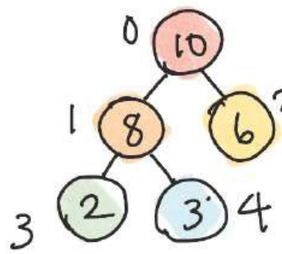
Binary tree → Binary search tree
→ Binary heap

- tree data structure
- each node has at most 2 children

Binary heap

- Complete tree
- min heap or max heap
- used for priority queue heap sort etc.

Max heap



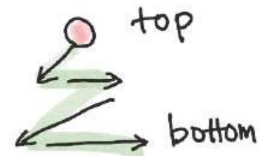
← the root has the largest

(min heap is the opposite!)

in array →

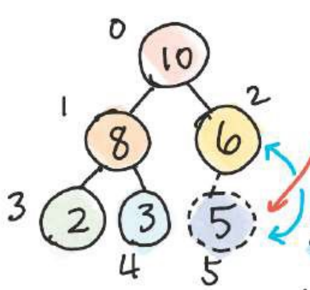
10	8	6	2	3
[0]	[1]	[2]	[3]	[4]

- each node has 0 – 2 children
- always fill top → bottom, left → right



Insertion

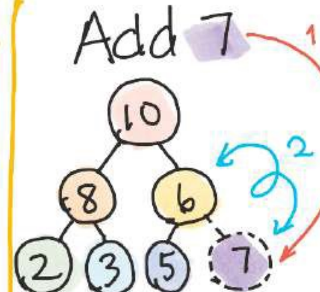
Let's add 5 to the heap!



1. add to the next node.
2. compare w/ its parent
3. the parent is greater. Cool, it's done! Let's add more!

10	8	6	2	3	5	...
[0]	[1]	[2]	[3]	[4]	[5]	

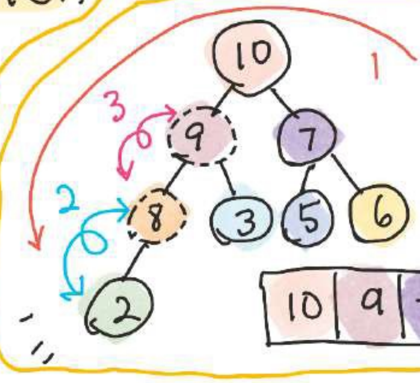
Add 7



1. Add to the next node
2. Compare w/ parent.

Oh, no! the parent is smaller than its child! Swap them!!!

10	8	6	2	3	5	7	...



Add to the next node & repeat the process!

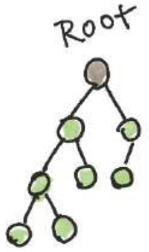
10	9	7	8	3	5	6	2	...

@girlie_mac

Data Structures

Binary Search Tree

BST



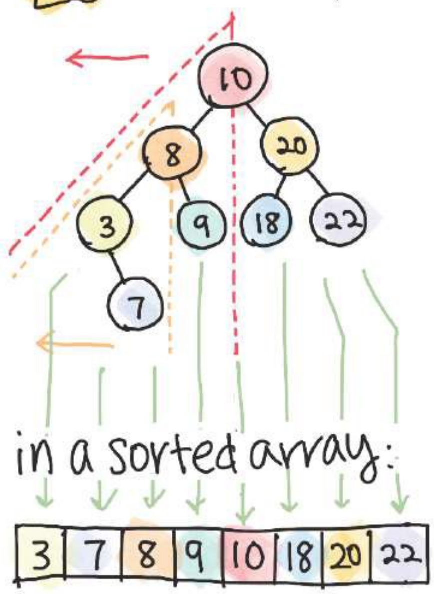
Binary tree → Binary heap
Binary tree → Binary Search Tree

- tree data structure
- each node has at most 2 children

Binary Search Tree

- a.k.a. Ordered or Sorted binary tree
- fast lookup
  e.g. phone number lookup table by name

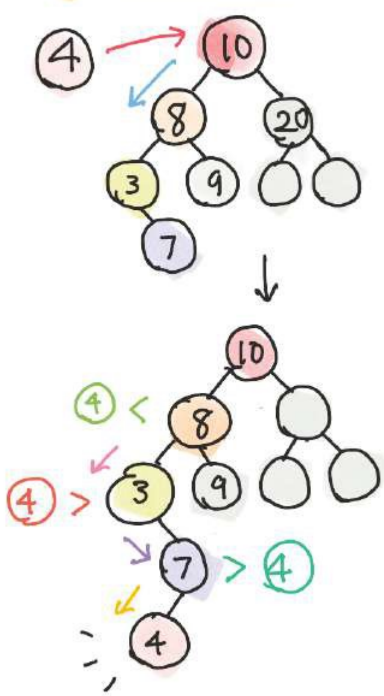
Rule of thumb



- ★ each value of all nodes in the left subtrees is lesser
  - 10's left subtrees: $\underbrace{8, 3, 9, 7}$
  - 8: 3, 7 ← smaller than parent
- ★ each value of all nodes in the right subtrees is larger
- ★ no duplicate values

Insertion

→ Always add to the lowest spot to be a leaf. No rearrange!



Let's add 4

1. Compare w/ the root first.
2. 4 < 10 so go left.
3. then compare w/ the next, 8
4. 4 < 8 so go left
5. Compare w/ the 3
6. 4 > 3 so go right.
7. Compare w/ the 7
8. 4 < 7, so add to the left! Done.

Complexity:
Ave. O(log n)
Worst. O(n)