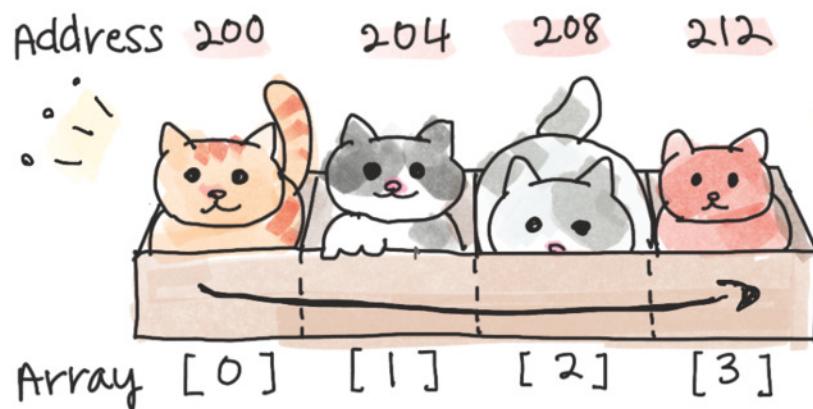


# Data Structures

## Array & Linked List

**Array** a linear data structure, stored in contiguous memory locations.



(♥ Assume each 🐱 is an integer)

= requires 4 bytes space

♥ The array of 🐱 must be allocated contiguously!

→ address 200 — 216



♥ can randomly access w/ index

$a[2] \rightarrow$

♥ contiguous = no extra memory allocated = no memory overflow



☺ fixed size. Large space may not be avail for big array

:= 🐱 took the space! :=

☺ Insert + delete elements are costly.

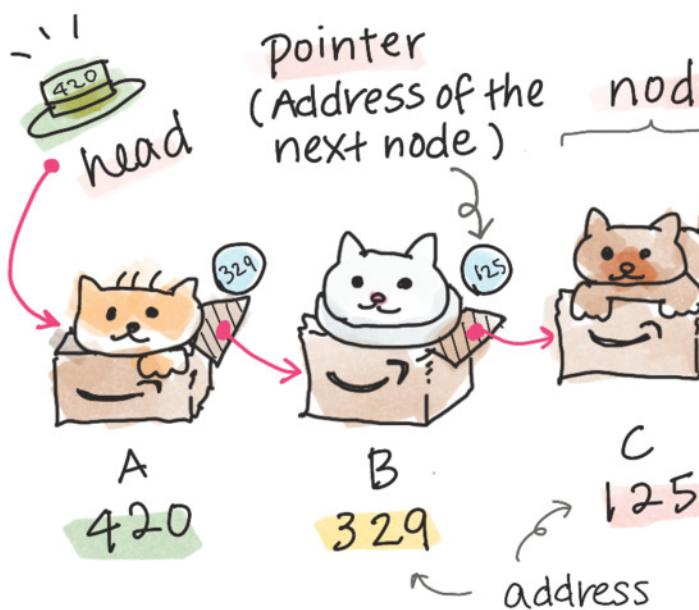
→ may need to create a new copy of the array + allocate at a new address.

# Data Structures

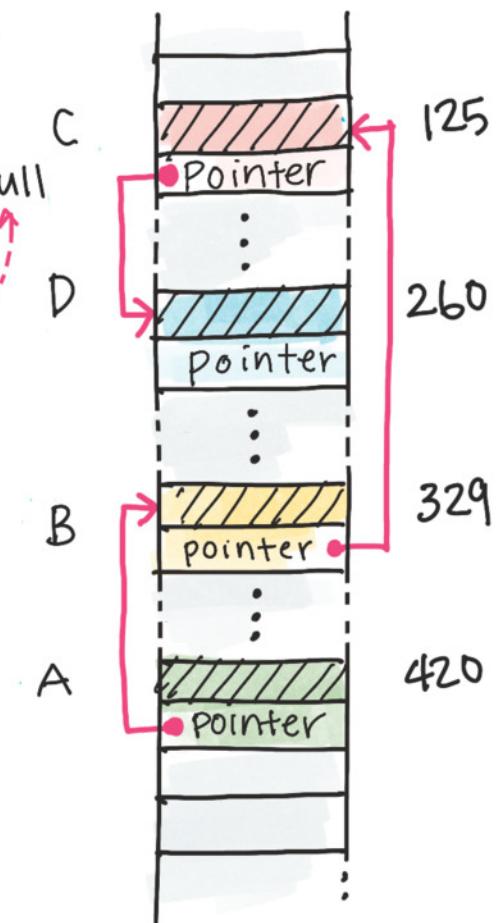
## Array & Linked List

### Linked list

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



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



Yay!

meh!

- 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

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

# Big Notation

## Time complexity of an algorithm

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

Const

array1 = [ , , , ,  ] array  
number of elements  
 $n=5$

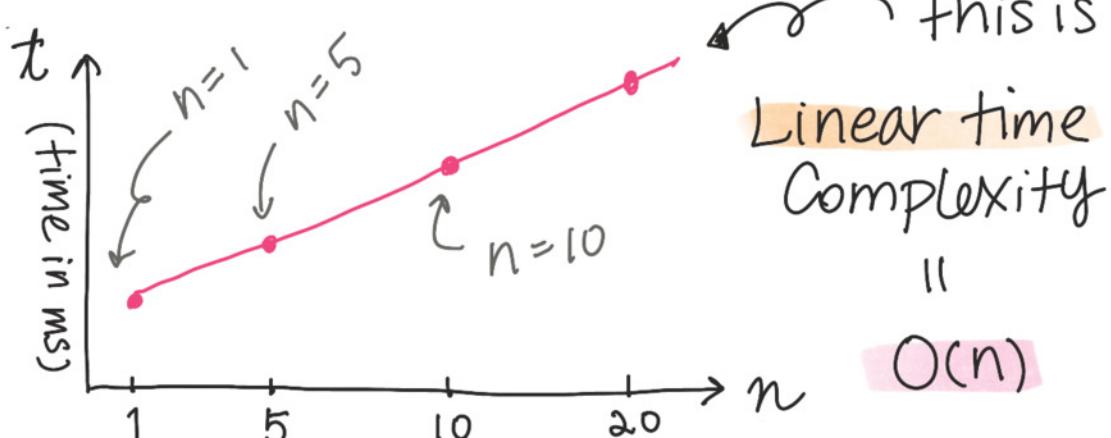
Let's see if there is a needle in the haystack!

JS  
Const numNeedles=(haystack, needle)  $\Rightarrow \{$   
let count=0  
for(let i=0; haystack.length ; i++) {  
, if(haystack[i]  $\equiv$  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!



@girlie\_mac

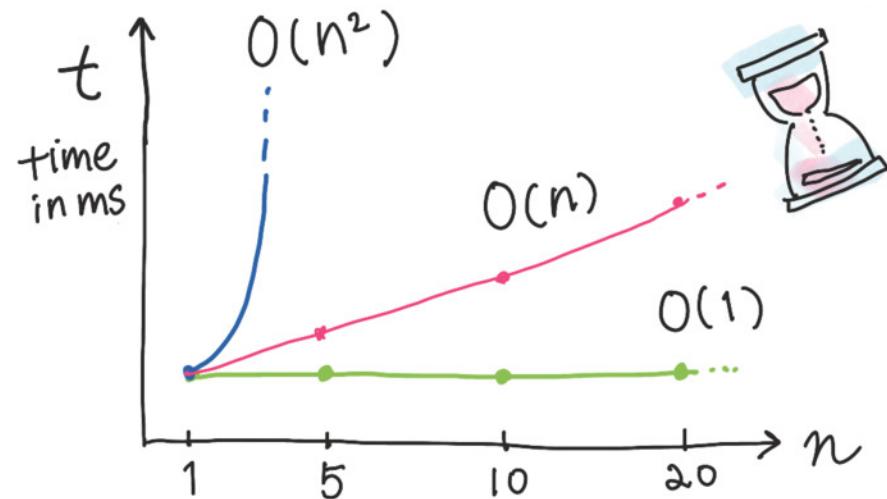
# Big O Notation

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

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

$n=5$  ~ Array size  
 $n=10$  has no effect  
 $n=20$  on the runtime  
⋮

Constant time  
" "  
 $O(1)$



Quadratic time =  $O(n^2)$

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

Const

```
array2 = [ , , , ,  ];
```

proportional to  $n^2$

JS 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;

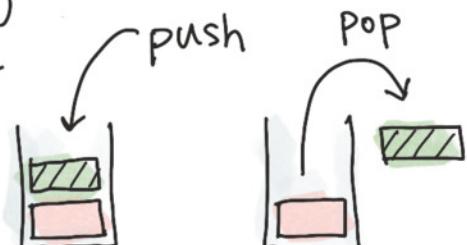
① Loop thru the array

② Another array lookup w/ indexOf method

# Data Structures

## Stack & Queue

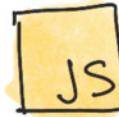
A stack is a LIFO (Last-in-First-out) data structure, where an element added last (=push) gets removed first (=pop)



just like a stack of ice cream scoops!



★ Stack as an array in



stack is:

```
let stack = [ ];  
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']
```

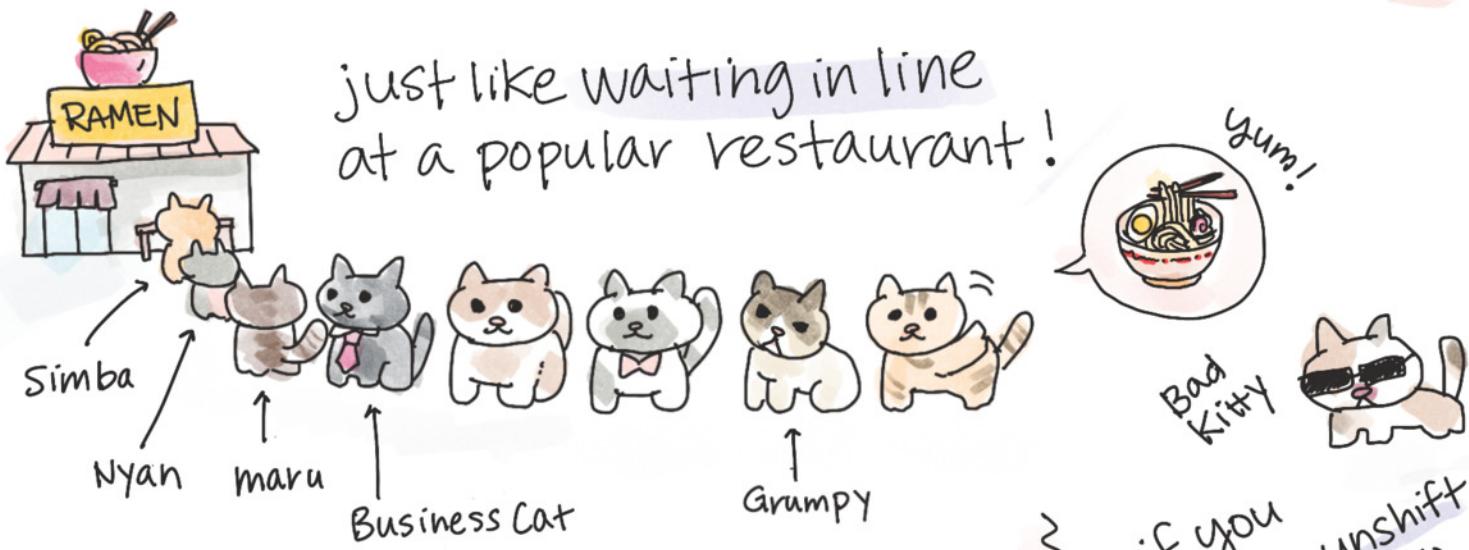
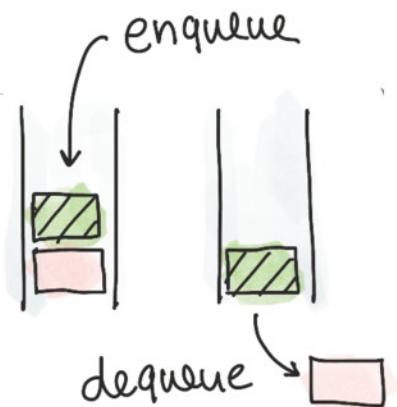
Time complexity is O(1)  
for both pop + push.

omg, the bottom one is always melting !!

# Data Structures

## Stack & Queue

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



### ★ Stack as an array in JS

```
let queue = [ ];
queue.push('Simba'); // ['Simba']
queue.push('Nyan'); // ['Simba', 'Nyan']
queue.push('Maru'); // ['Simba', 'Nyan', 'Maru']
let eater = queue.shift(); // eater is 'Simba'
```

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

if you  
queue.unshift()  
(`badKitty`),  
then the cat cuts  
in to the front  
of line!

queue is: instead of push(),  
queue is: ['nyan', 'Maru']

# Data Structures Hash Table

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

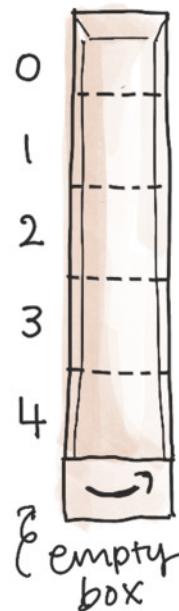
① Let's create an array of size 5.

We're going to add  data.

Key = "Tabby"

Value = "pizza"

Some data  
Let's say, favorite food!

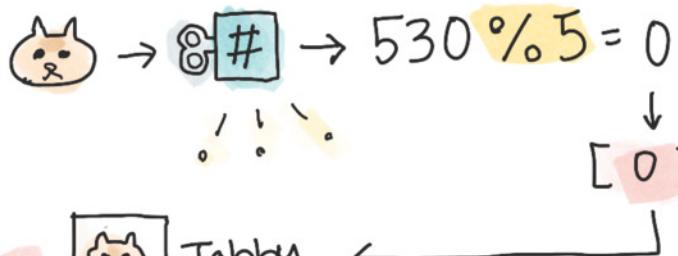


② Calculate the hash value by using the key, "Tabby".

e.g. ASCII code, MD5, SHA1



③ Use modulo to pick a position in the array!



Tabby

★ The hash is divided by the size of the array.

The remainder is the position!

④ Let's add more data.

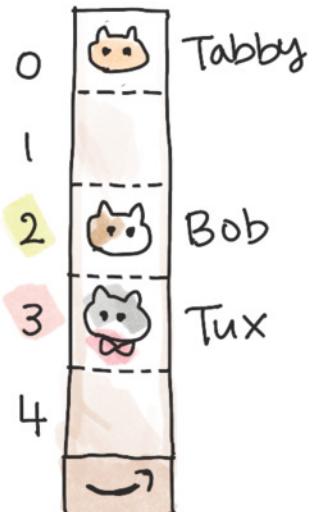
 8# →  $353 \% 5 = 3$

Tux

 8# →  $307 \% 5 = 2$

Bob

Use the same method to add more 



# # Hash Table

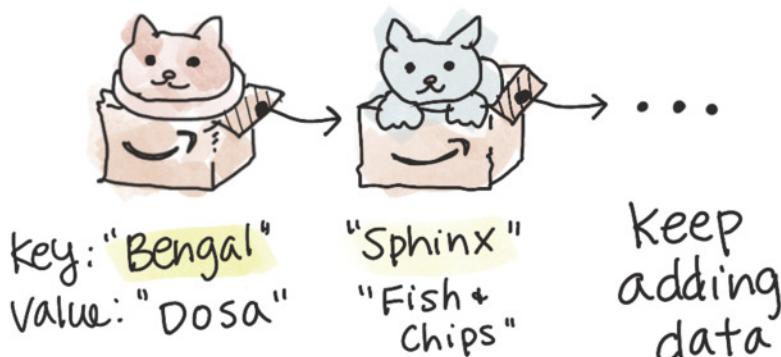
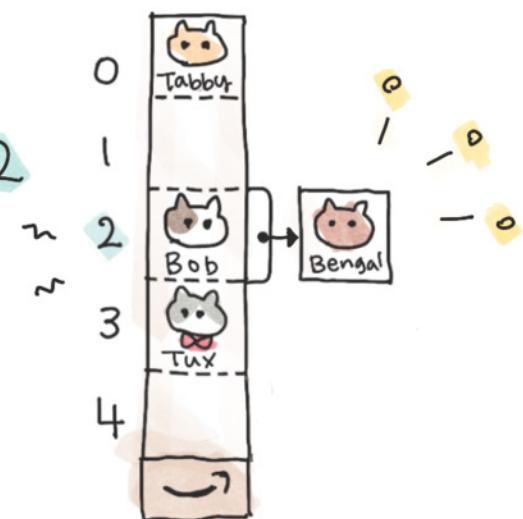
@girlie-mac

## Collision!

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

"Bengal" → 8# → 617%5 = 2

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



## Searching for data

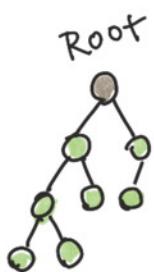
Let's look up the value for "Bob"

- ① Get the hash → 307
  - ② Get the index → 307 % 5 = 2
  - ③ Look up Array [2] → found!
- ← O(1)

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)
-

# Binary Search Tree



**Binary tree**

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

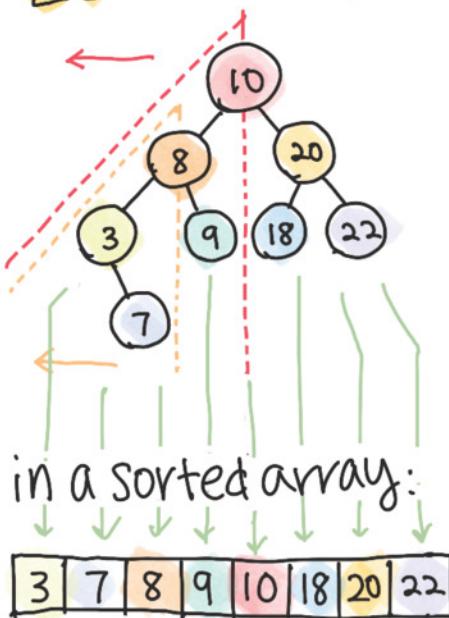
Binary heap

Binary Search Tree

- a.k.a. Ordered or Sorted binary tree

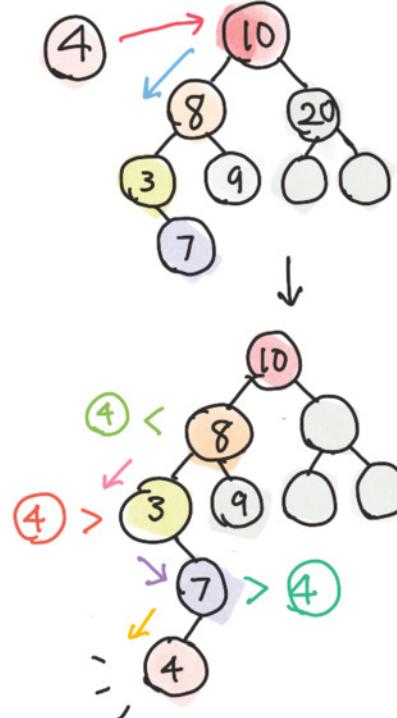
- fast look up  
e.g. phone number lookup table by name

Rule of thumb



- each value of all nodes in the left subtrees is lesser
- $\triangle$  10's left subtrees: 8, 3, 9, 7
- $\triangle$  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.  $O(n)$

## Data Structures

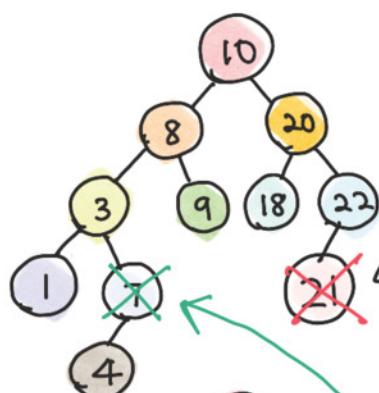
## BST

Binary Search Tree!



## ★ Deletion

- Case 1: the to-be-deleted node has no child
- Case 2: the node has 1 child
- Case 3: the node has 2 children



Let's remove 21 ← Case 1.

It has no child. So just remove it from the node. Done! Easy peachy!

Now, let's delete 7 ← Case 2

1. just remove it
2. then move the child, 4 to the spot!

Now, let's delete 8 ← Case 3!

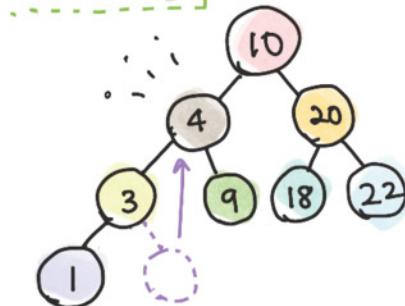
1. Remove it from the spot
2. Then look for the largest node from the left subtree

3. The largest is 4!

Done!

move the node to the removed spot!

(Alternatively, look for the smallest from the right subtree.)



④ originally had no child.  
but if it has children?  
→ Repeat the process!

↳ Find the largest from left Subtree. Move it

↳ Find the largest from left subtree...

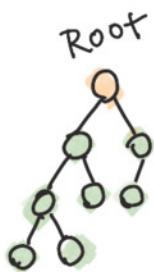
Recursive

Complexity:

Ave.  $O(\log n)$ Worst.  $O(n)$ 

# Data Structures

# Binary Heap



**Binary tree**

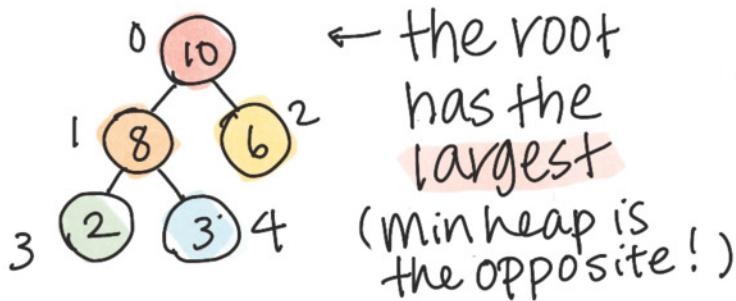
tree data structure  
each node has at most 2 children

→ Binary search tree

→ Binary heap :

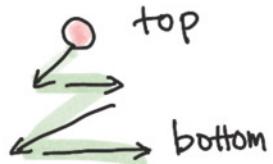
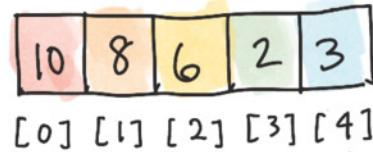
- Complete tree
- Min heap or max heap
- used for priority queue, heap sort etc.

## Max heap



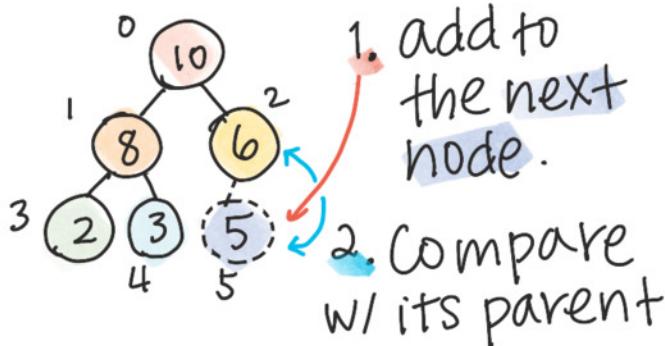
- each node has 0 - 2 children
- always fill top → bottom, left → right

in array



## Insertion

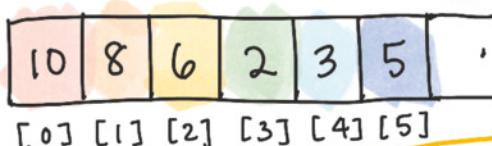
Let's add 5 to the heap!



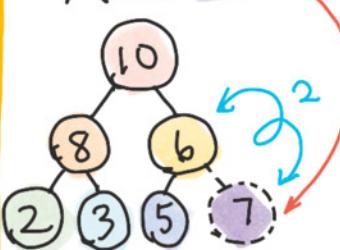
3. the parent is greater.

Cool, it's done!

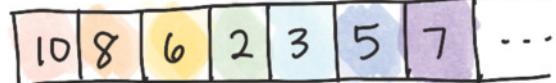
Let's add more!



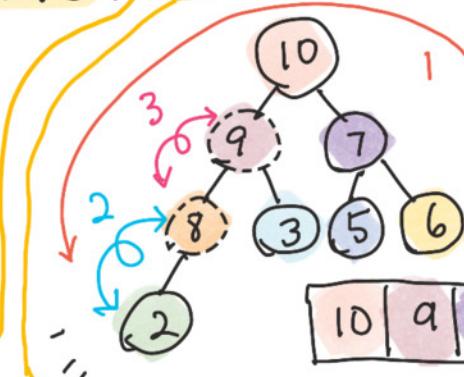
Add 7



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



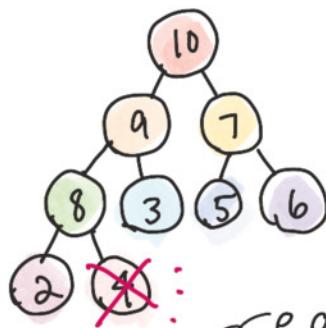
Add to the next node & repeat the process!



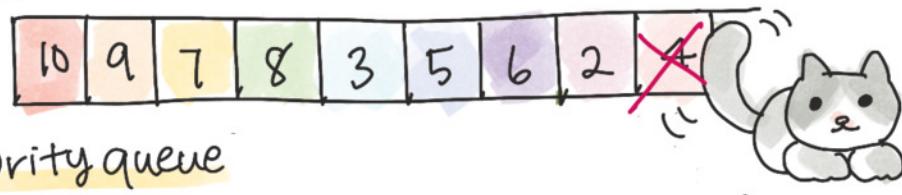
## Data Structures

## Binary Heap

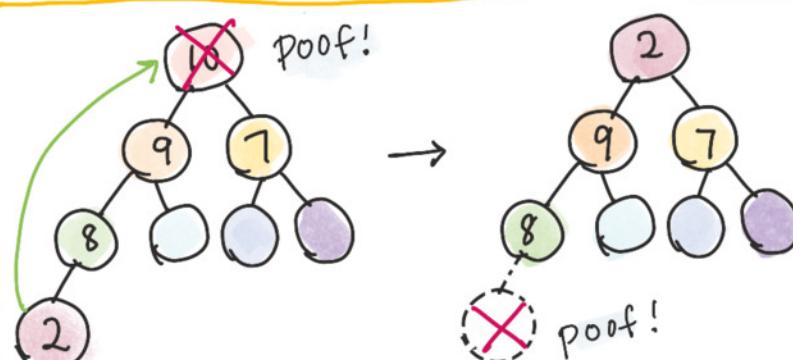
## ★ Heap Deletion



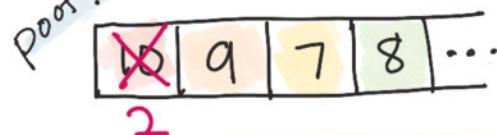
If you want to delete the last leaf,  
just delete it!



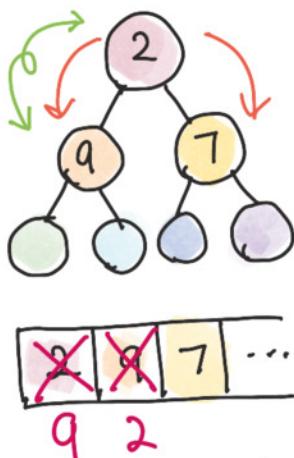
But typically, you would delete root + heapify!



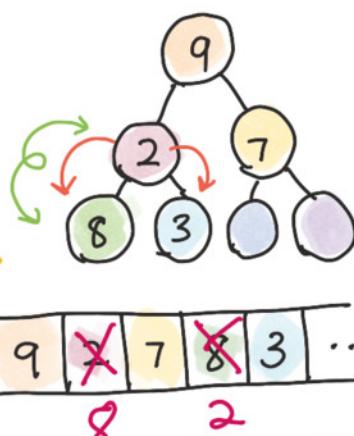
1. Remove the root
2. Move the last leaf to the root



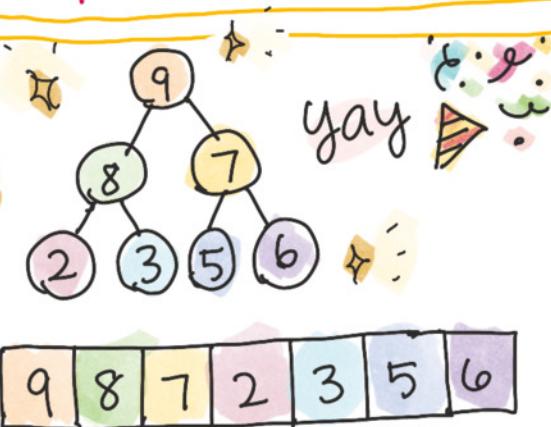
Now, let's place them in the correct order!



If either child is larger than the root, swap it with the larger child.



Repeat.  
Compare w/ the children & swap w/ the larger child



## Time Complexity

- Insert
- Delete

	Average	Worst
Insert	$O(1)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

