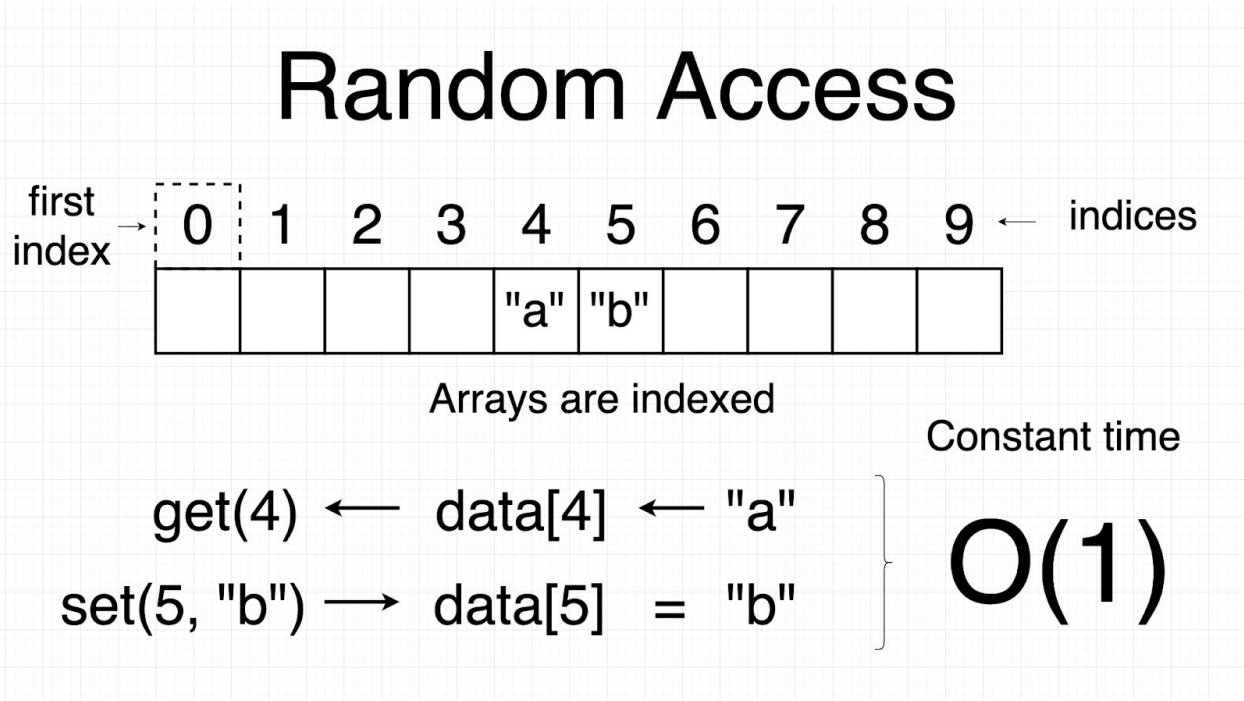
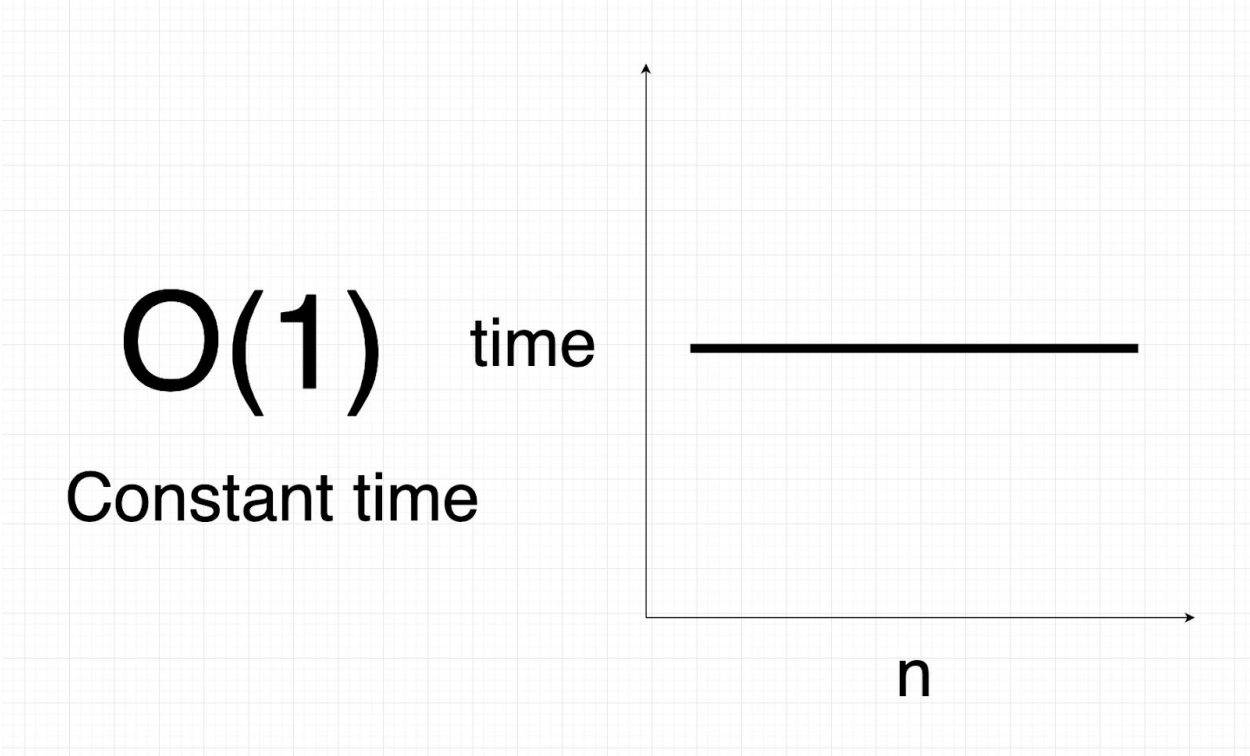
Data Structures and Algorithms Bootcamp Cheat Sheet

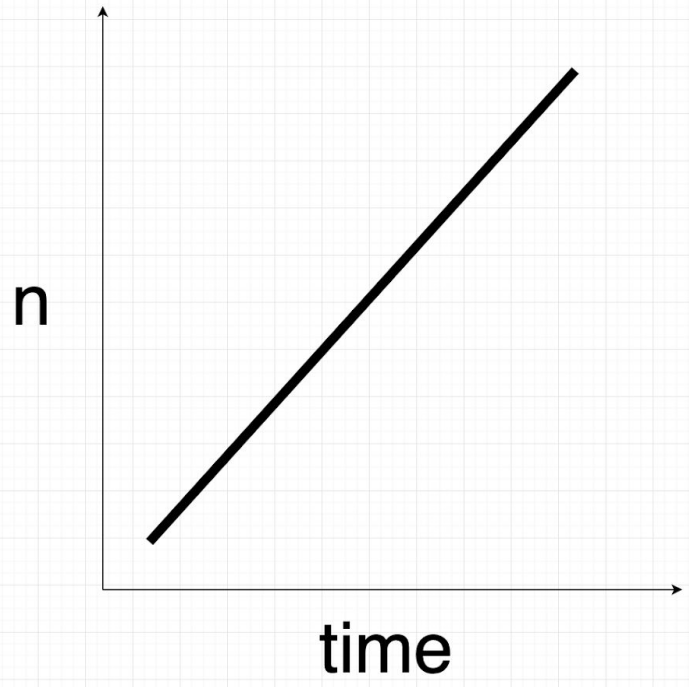
Arrays


Random Access
first index
0 1 2 3 4 5 6 7 8 9
indices
"a"
"b"
Arrays are indexed
Constant time
get(4) ← data[4] ← "a"
set(5, "b") → data[5] = "b"
O(1)

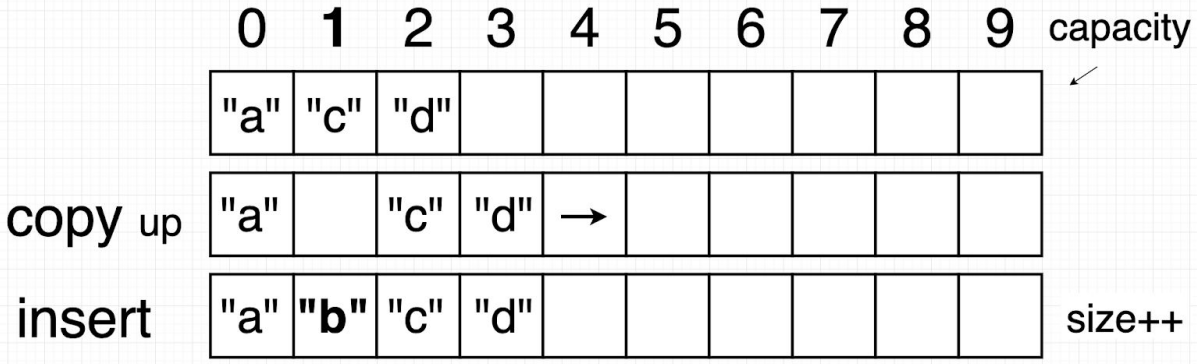

O(1)
time
Constant time
n

O(n)

Linear time



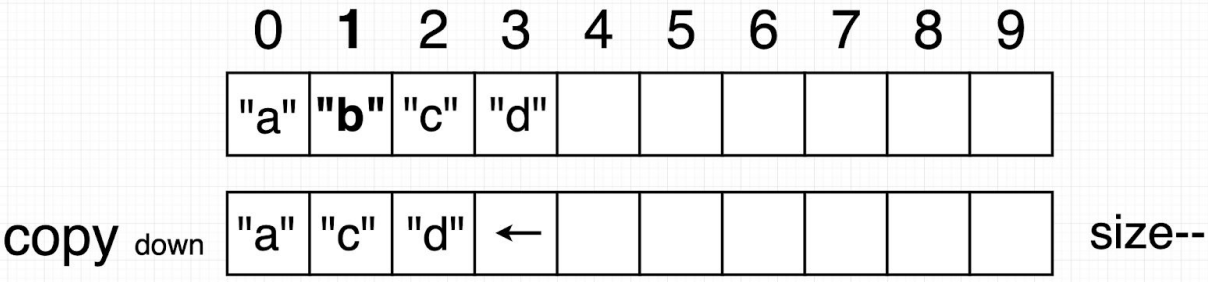
Insert



Linear time

insert(1, "b") O(n)

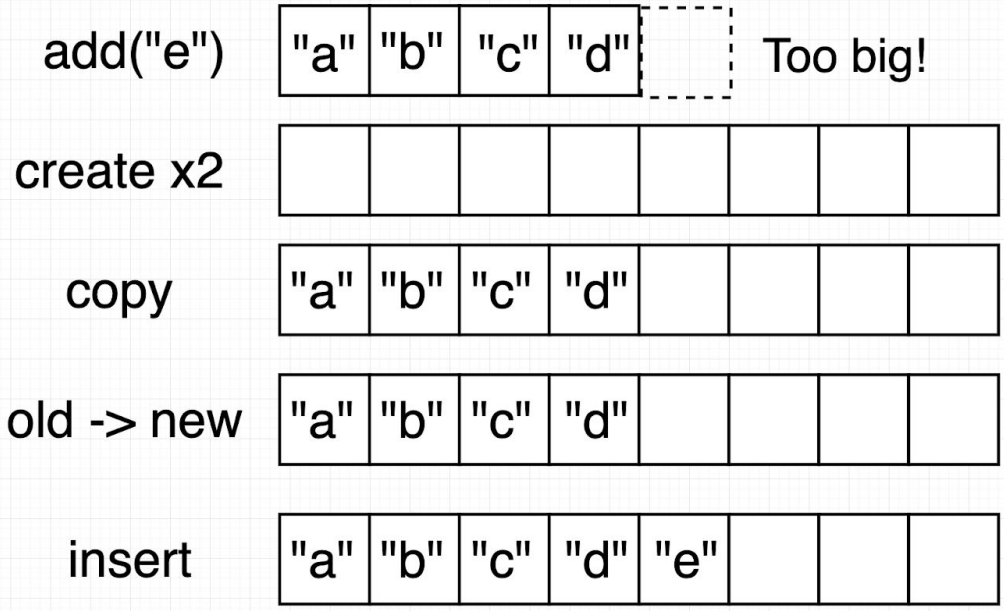
Delete



Linear time

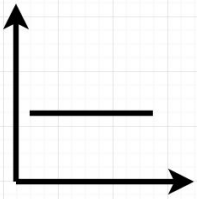
delete(1) } O(n)

Dynamic Arrays O(n)



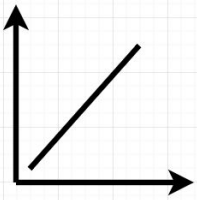
Runtime characteristics

Get/Set



O(1) Constant time

Insert
Delete
Contains
Resize



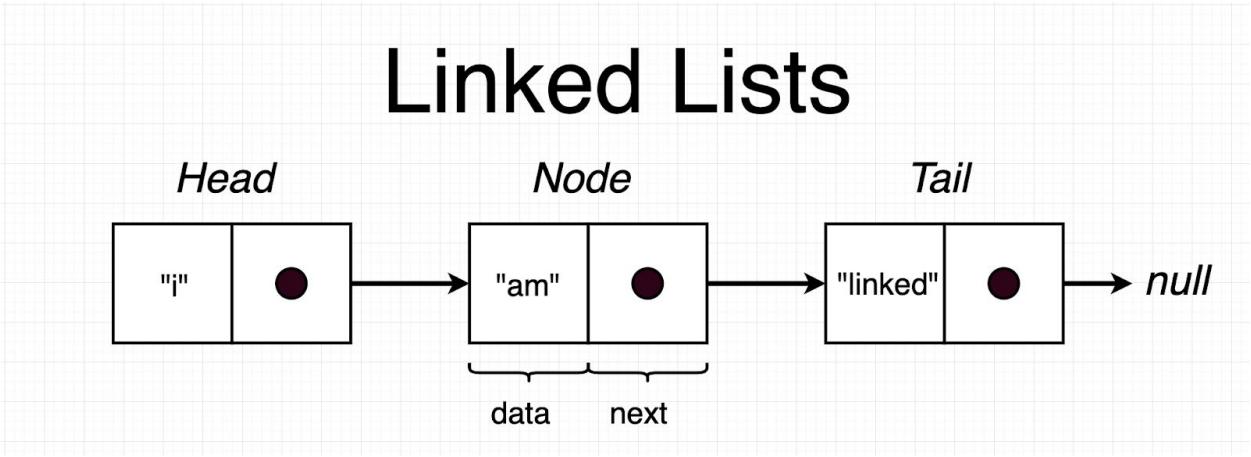
O(n) Linear time

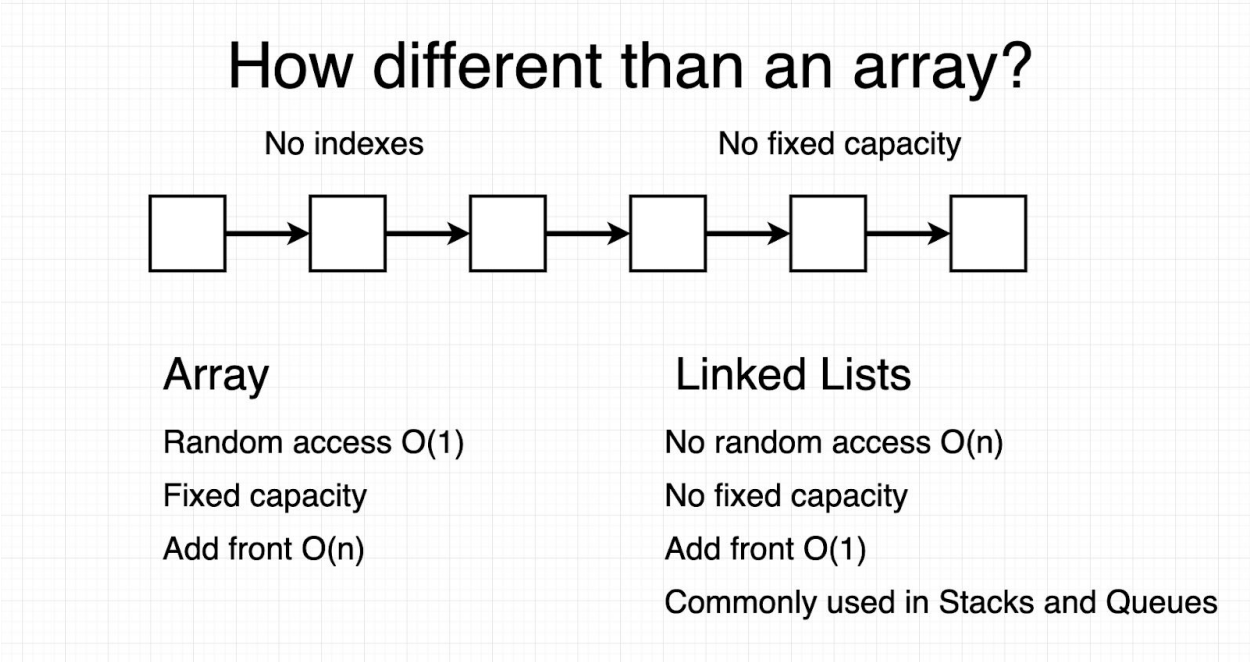
Need to know

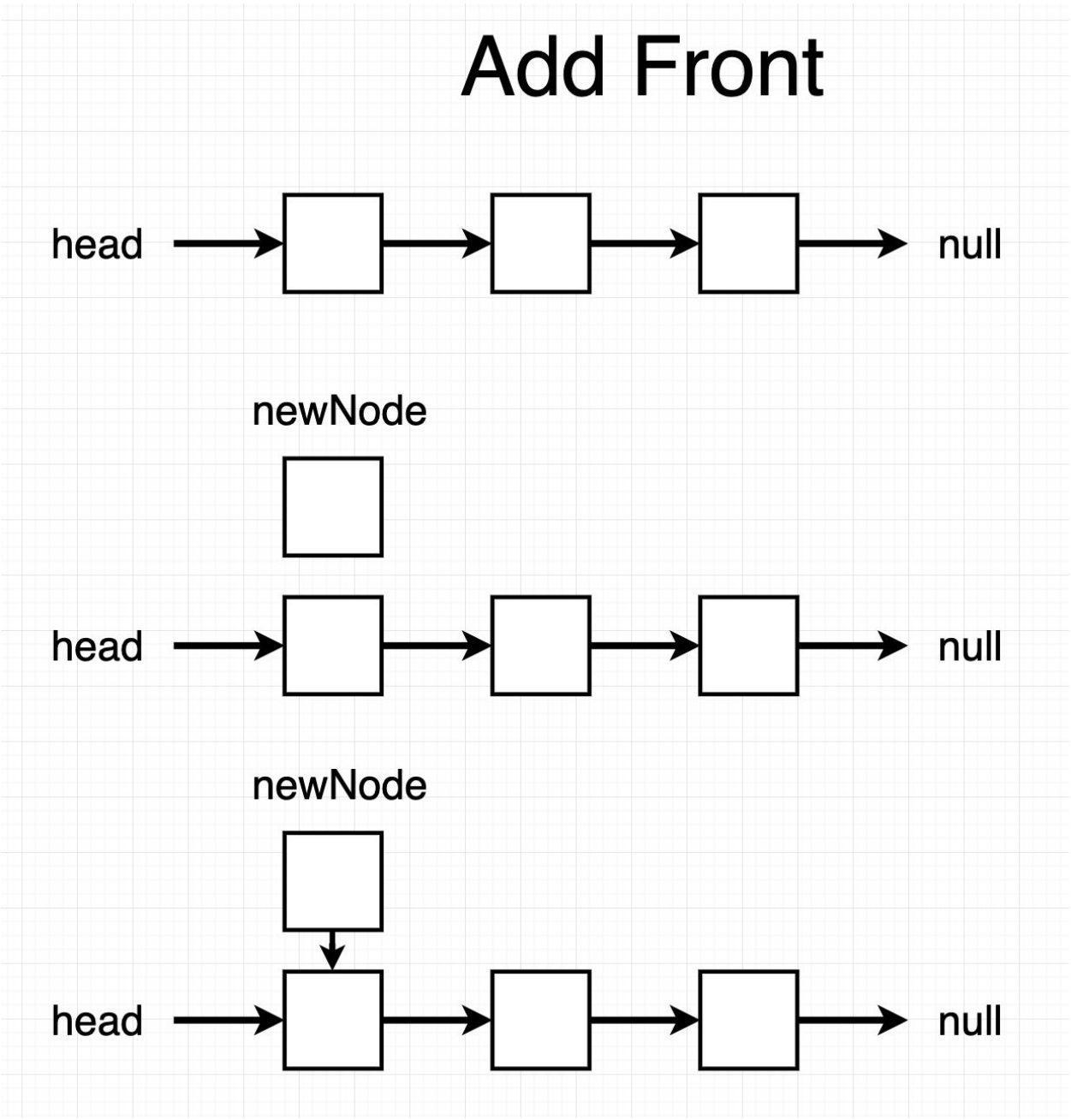
Killer feature

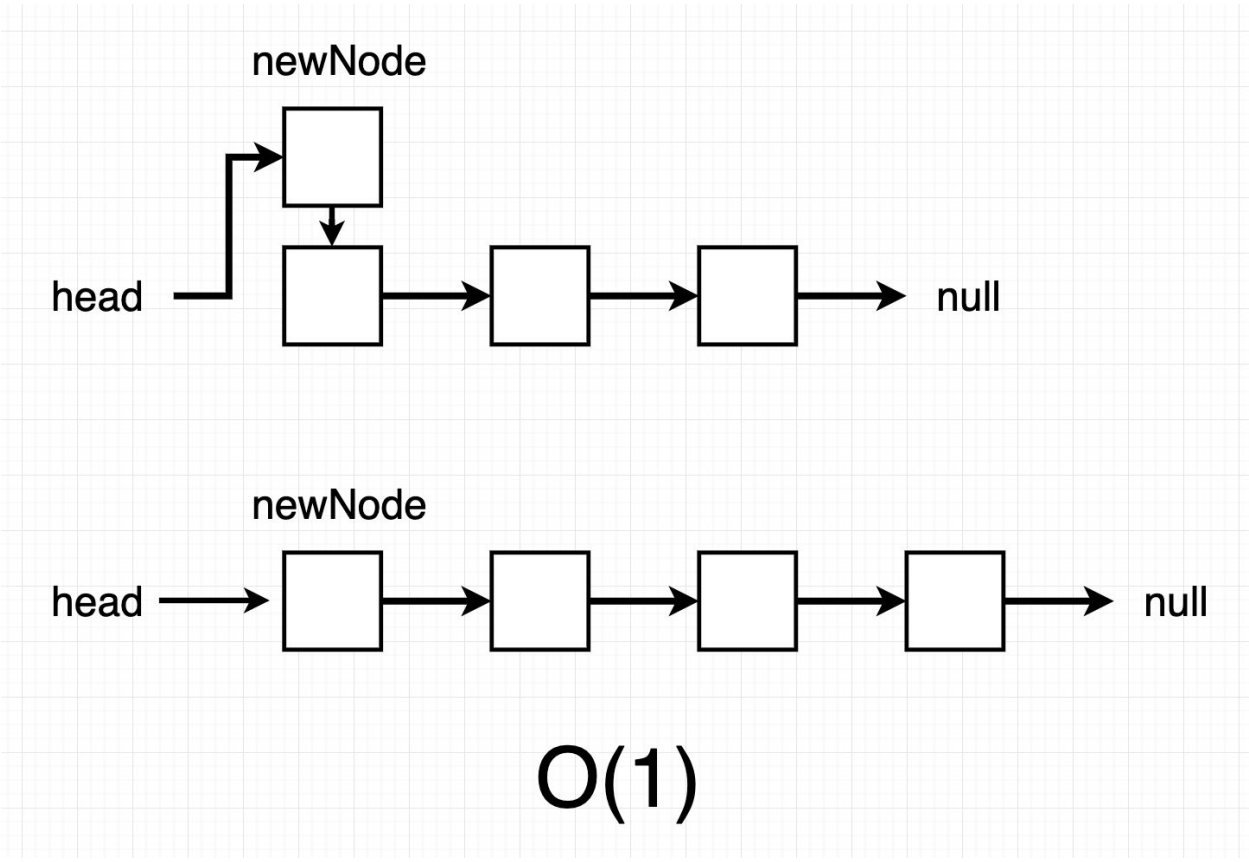
Get/Set	O(1)	Random Access
Insert	O(n)	Fixed capacity
Delete	O(n)	Double when resize

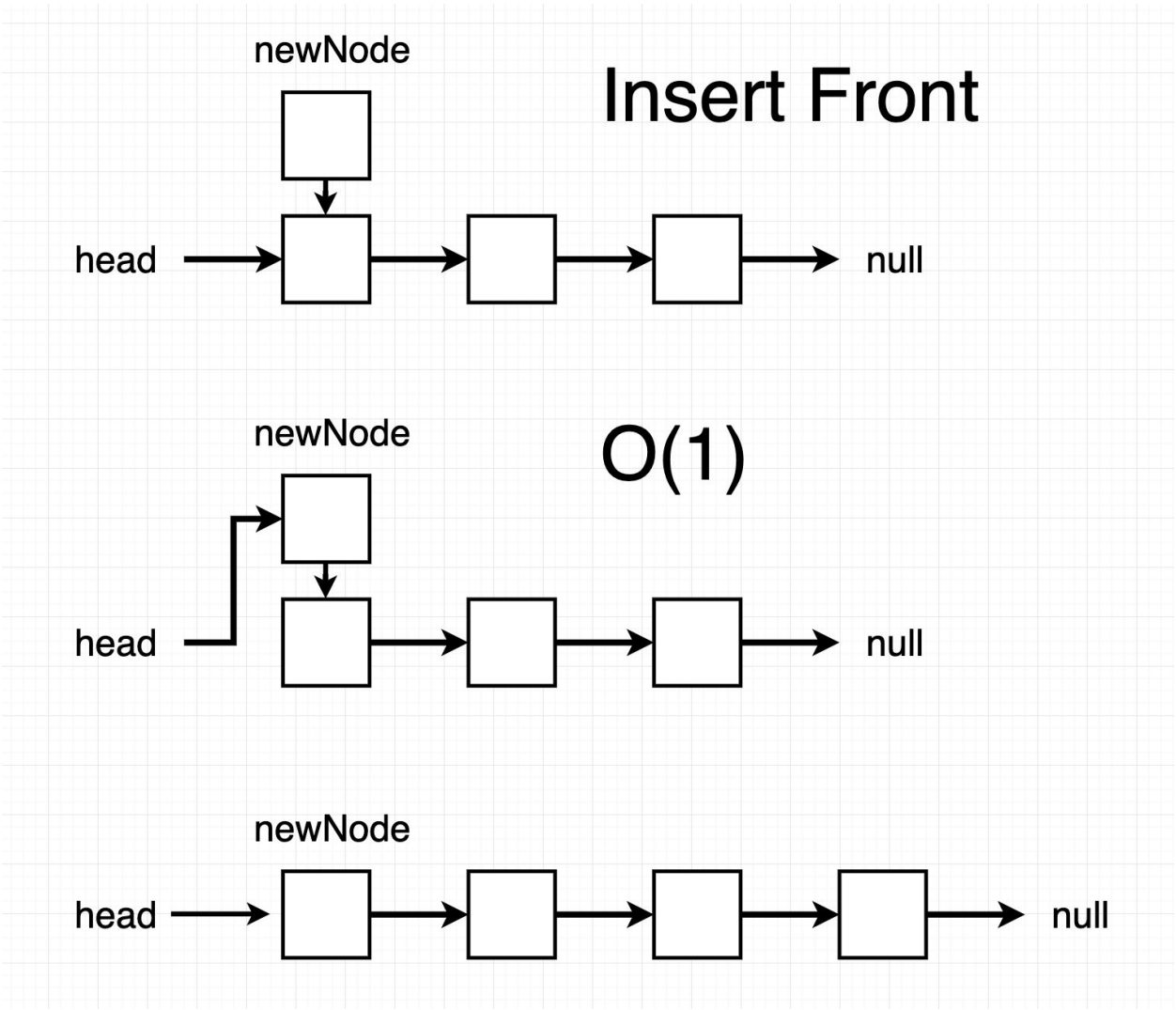
Linked Lists

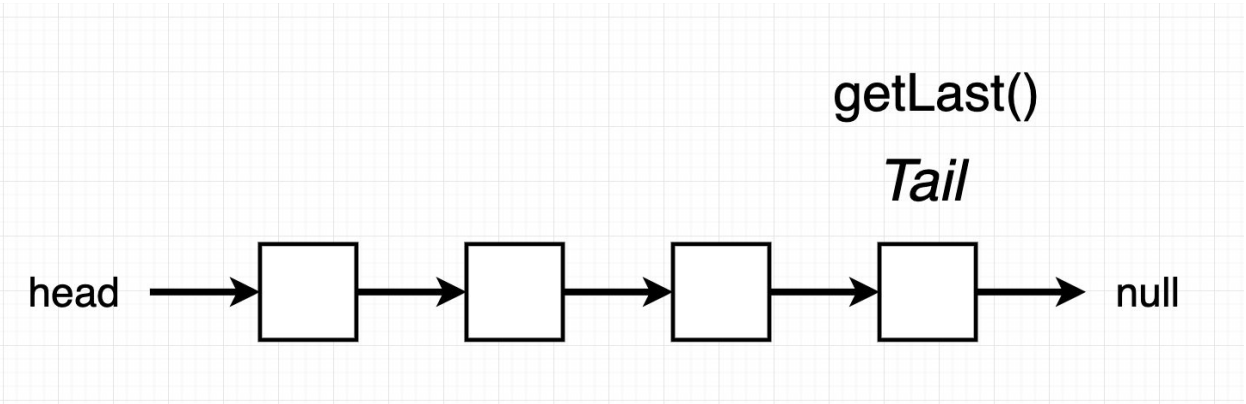


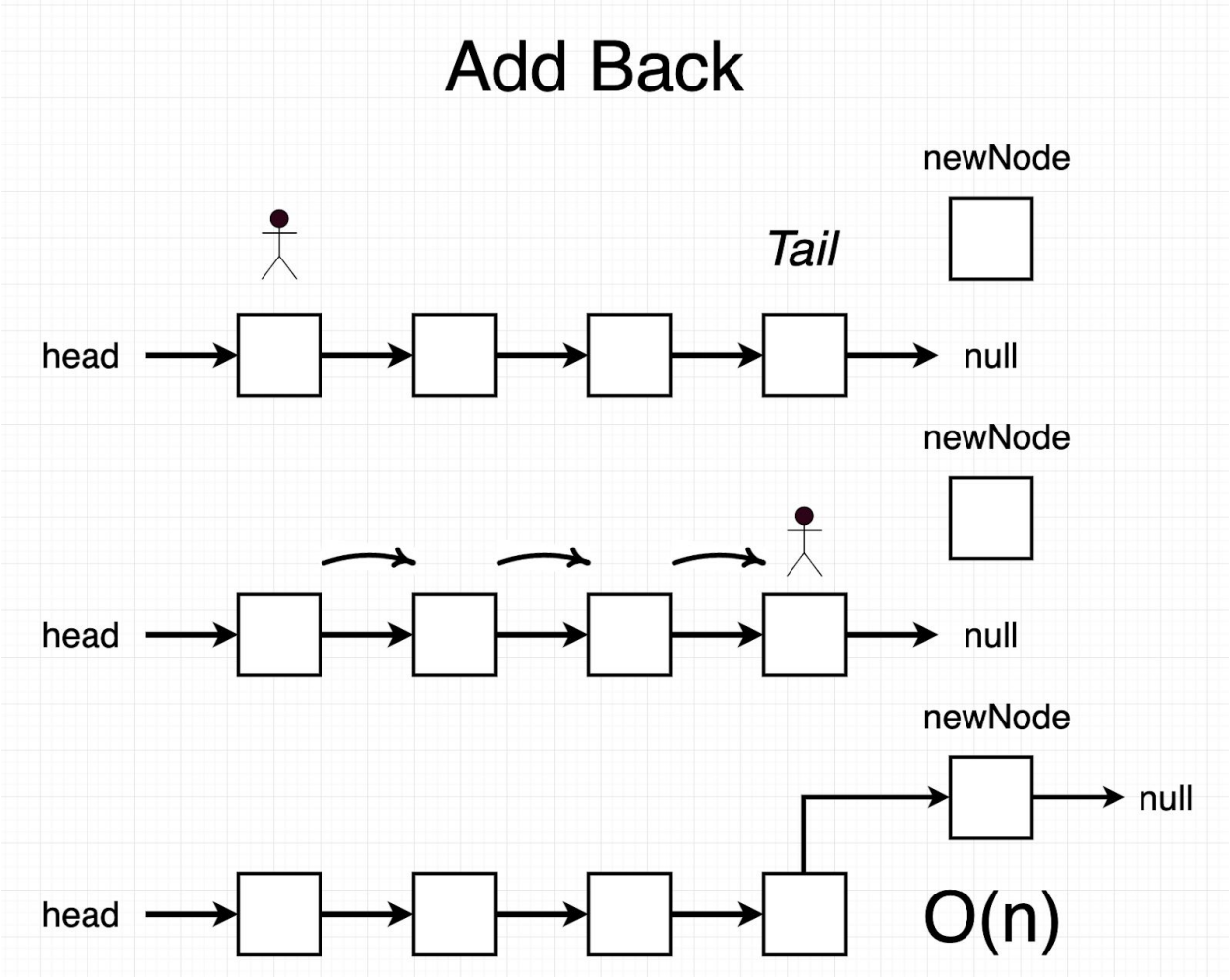


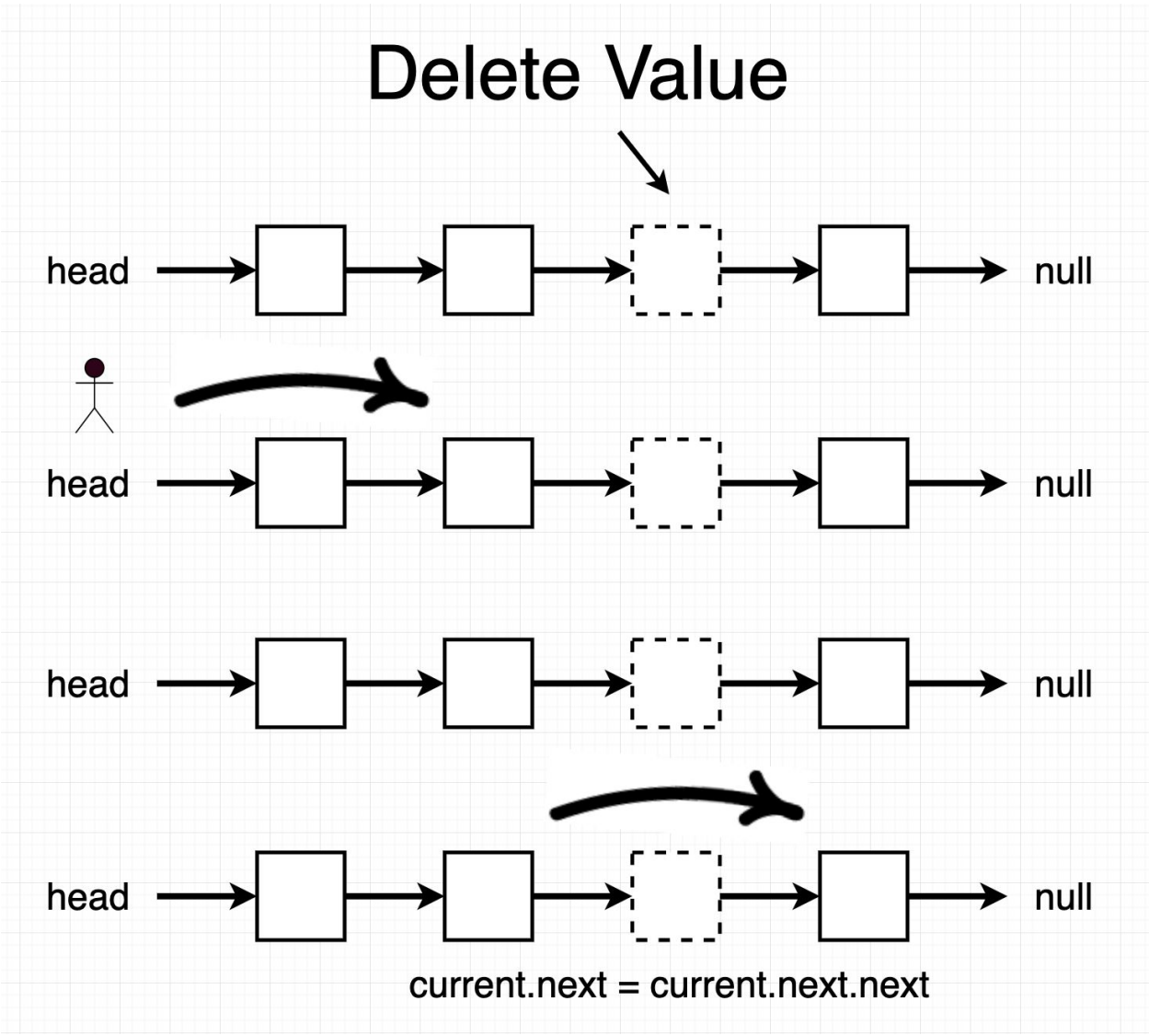

Add Front
head
null
newNode
head
null
newNode
head
null


newNode
head
null
newNode
head
null
O(1)


newNode
Insert Front
head
null
newNode
O(1)
head
null
newNode
head
null


getLast()
Tail
head
null


Add Back
newNode
Tail
head
null
newNode
head
null
newNode
null
head
O(n)


Delete Value
head
null
head
null
head
null
head
null
current.next = current.next.next

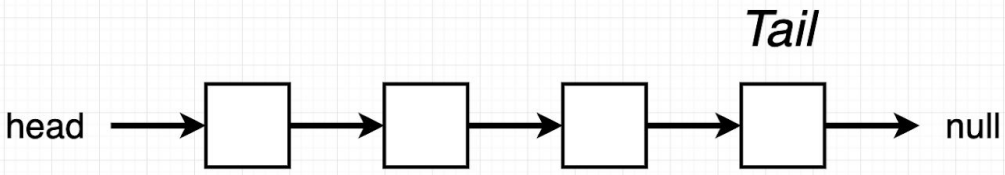
Runtime Characteristics

Add front	O(1)
Get first	O(1)
Add back	O(n)
Size	O(1) or O(n)
Clear	O(1)
Delete	O(n)

What you need to know

- Add Front O(1)
- Add Back O(n)
- Delete O(n)

- No random access
- No fixed capacity
- Always the right size



Big O Notation

What is Big O Notation?

Aka Time complexity

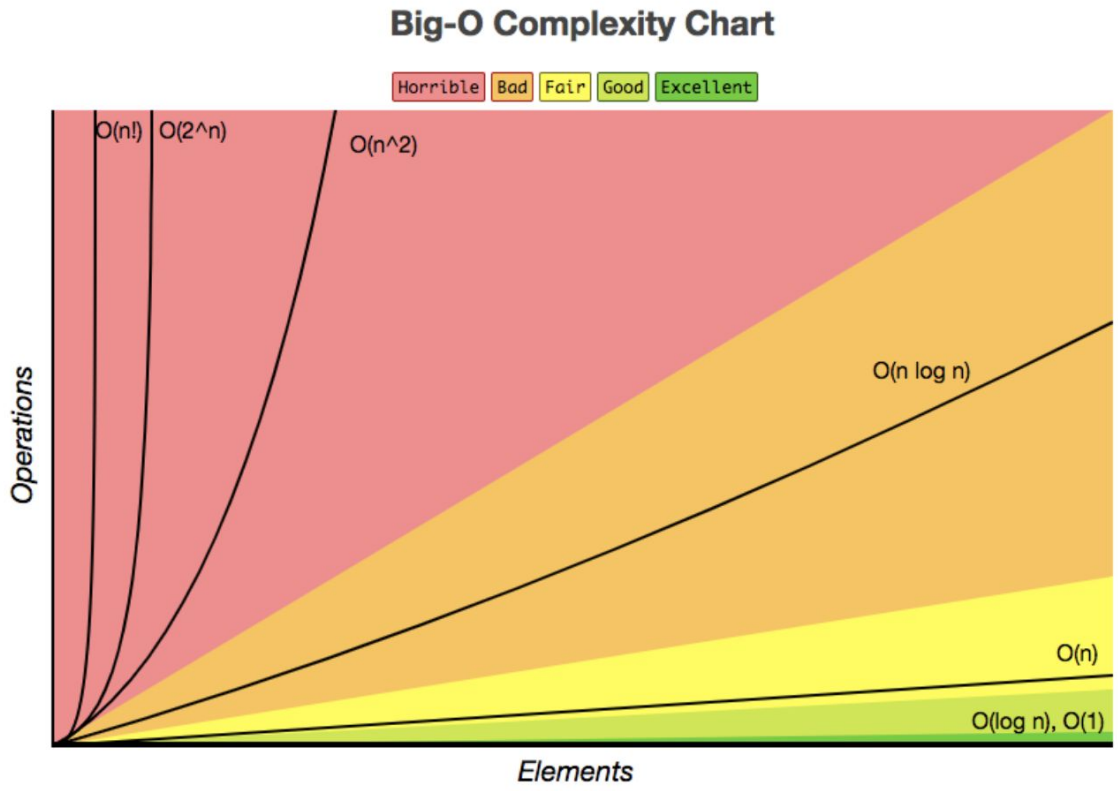
- The efficiency of your algorithm
- Used to describe runtime characteristics of our data structures and algorithms

O(1)	O(n)
Constant time	Linear time
Random access in array	Delete Value linked list

Why important?

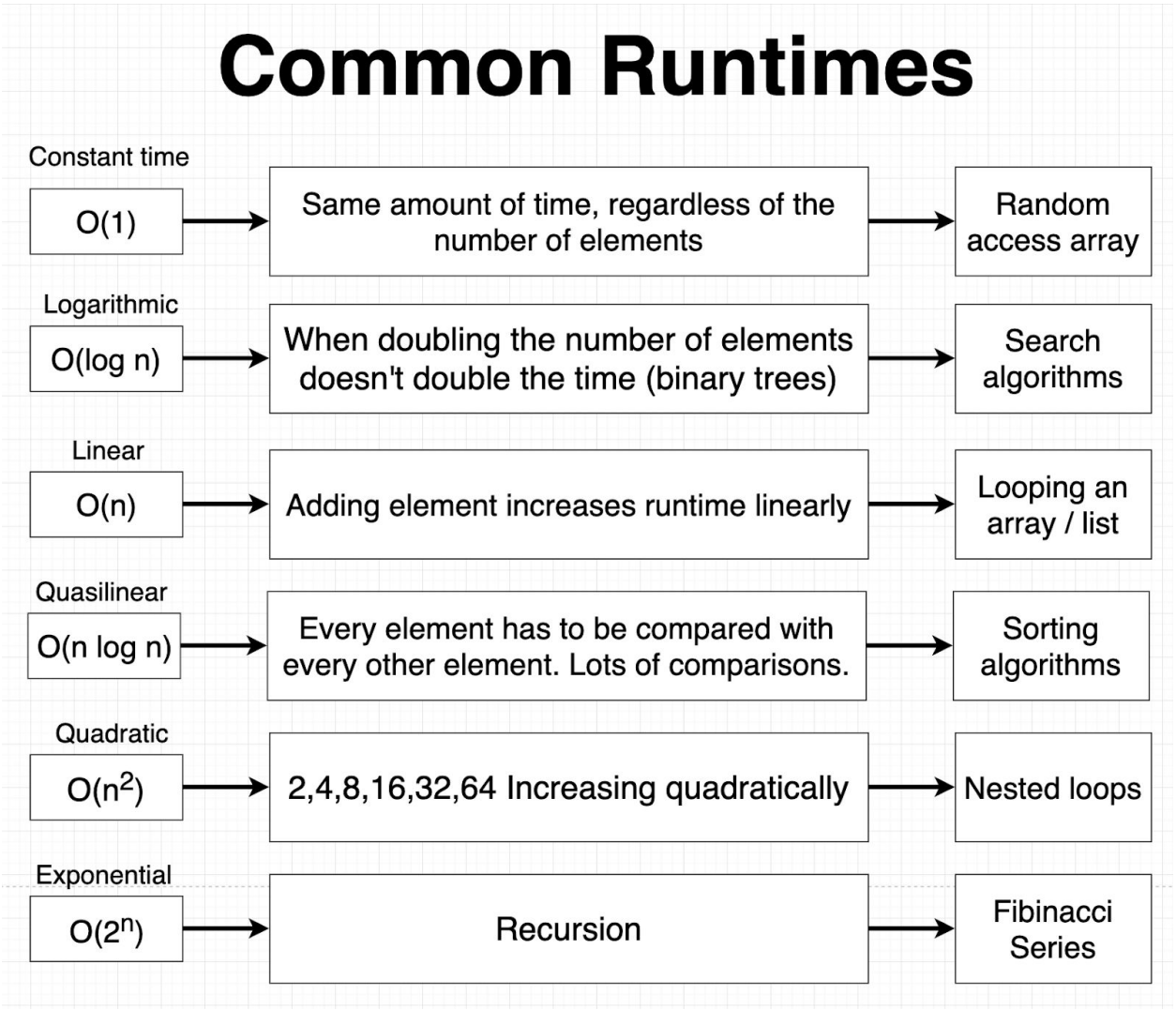
- Gives us a way to compare runtimes...
- You may be asked to describe the runtime characteristics of algorithms in your interview

Common Runtimes

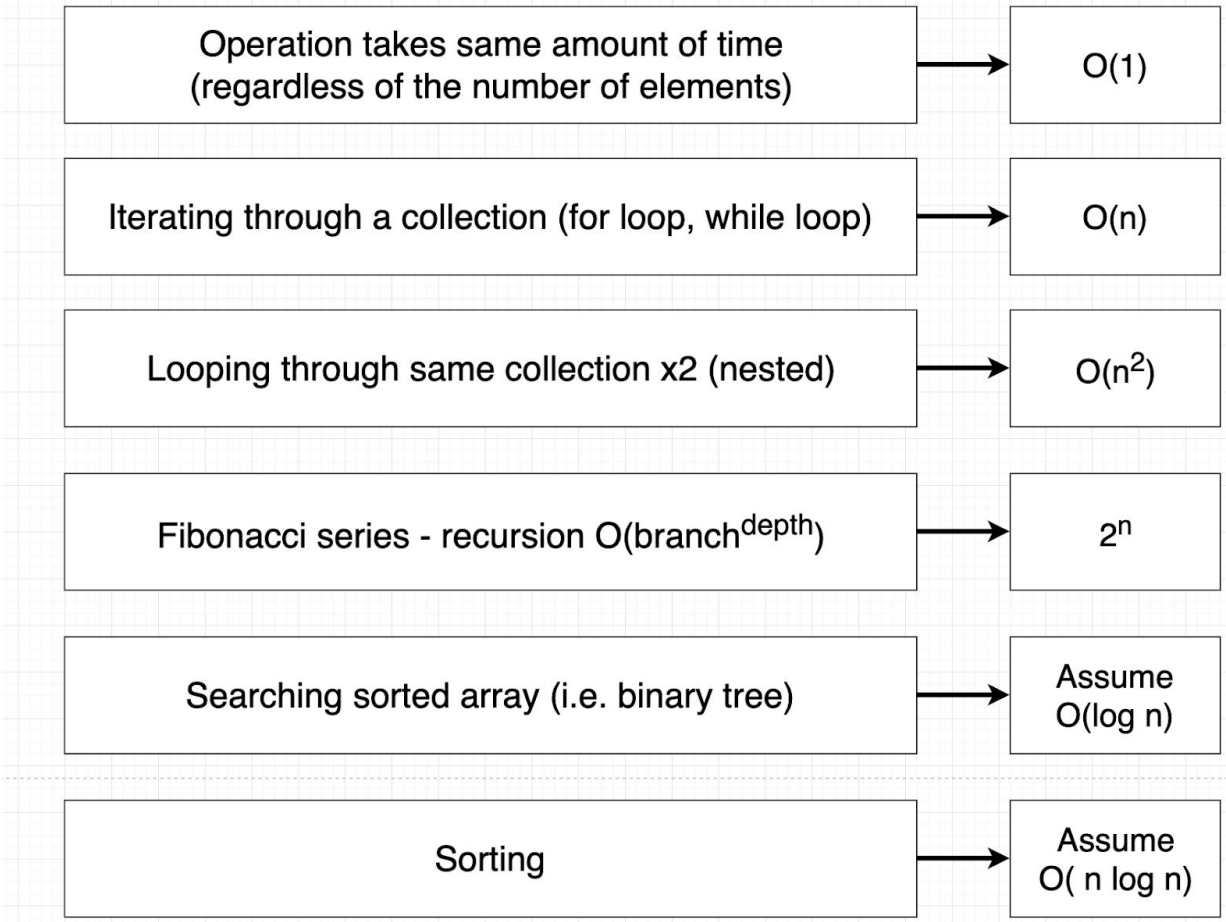


bigocheatsheet.com

Common Runtimes



How to identify patterns



Rules of thumb

Drop the non-dominant terms

$O(n^2 + n) \longrightarrow O(n^2)$

Drop the constants

$O(3n) \longrightarrow O(n)$

Add Runtimes: O(n + m)

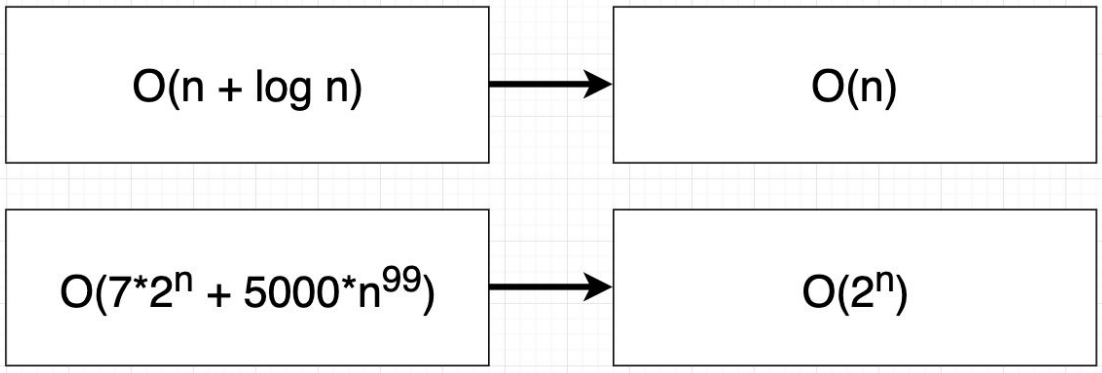
```
for (int n: arrayN) {
   print(n);
}

for (int m: arrayM) {
   print(m);
}
```

Multiply Runtimes: O(n * m)

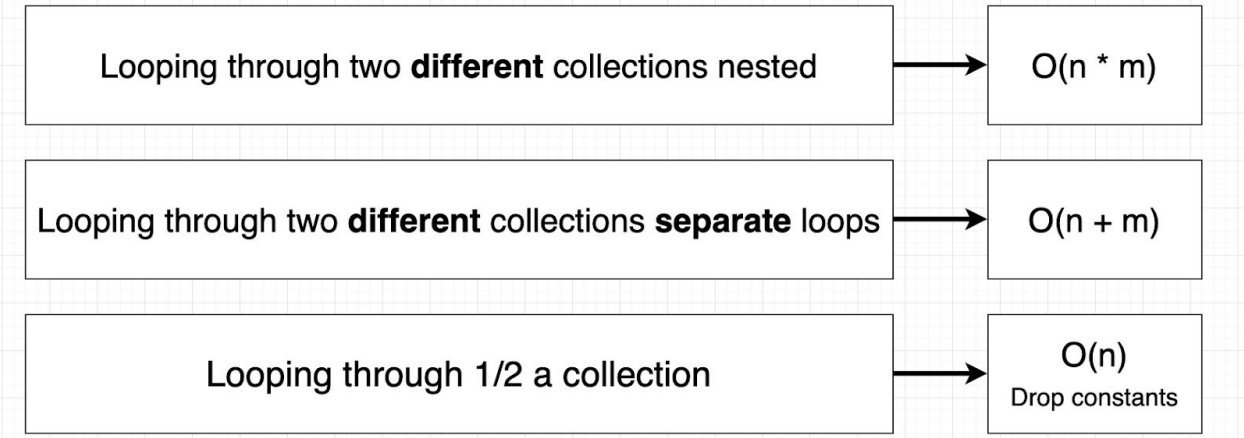
```
for (int n: arrayN) {
   for (int m: arrayM) {
      print(n + "," + n);
}
```

Note: Not $O(n^2)$

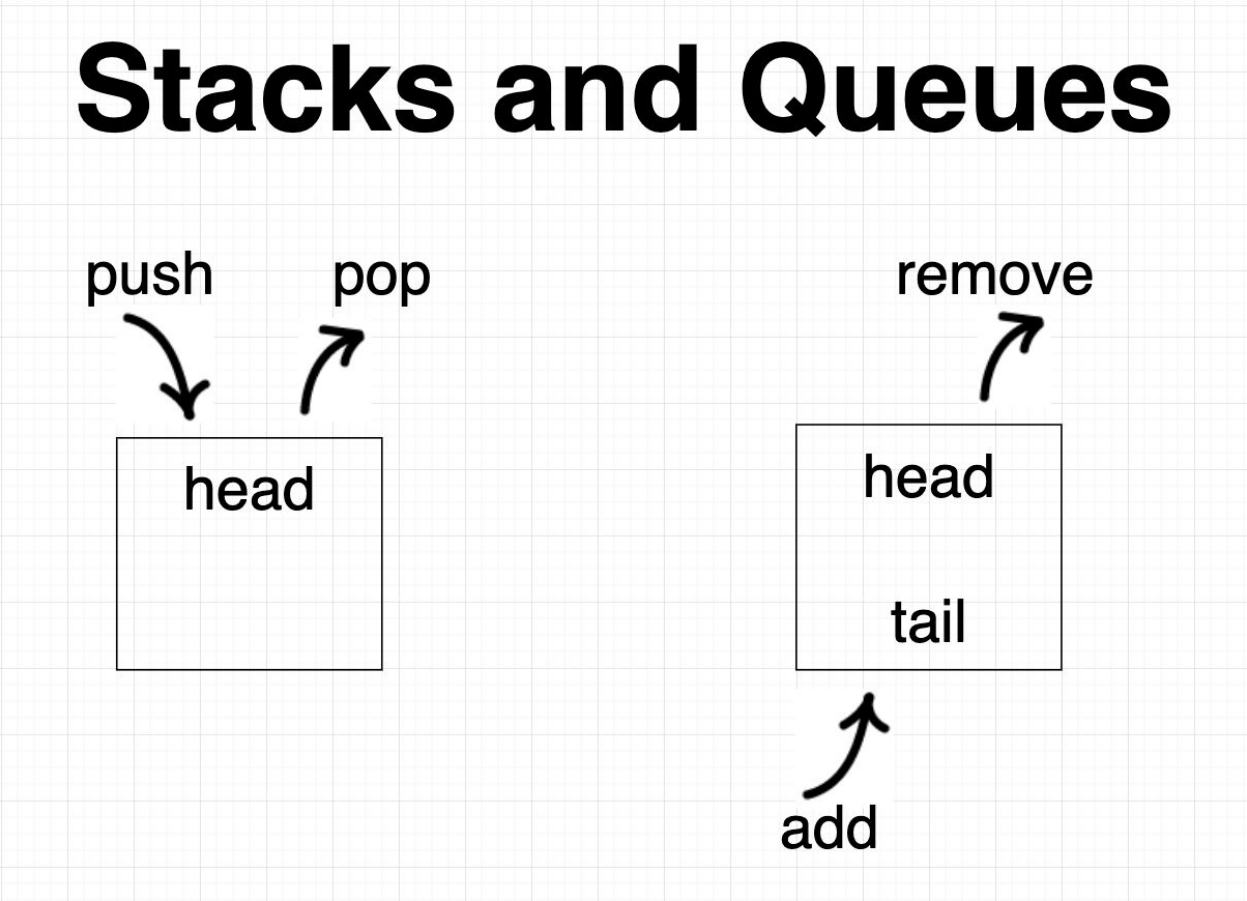
Reduce these



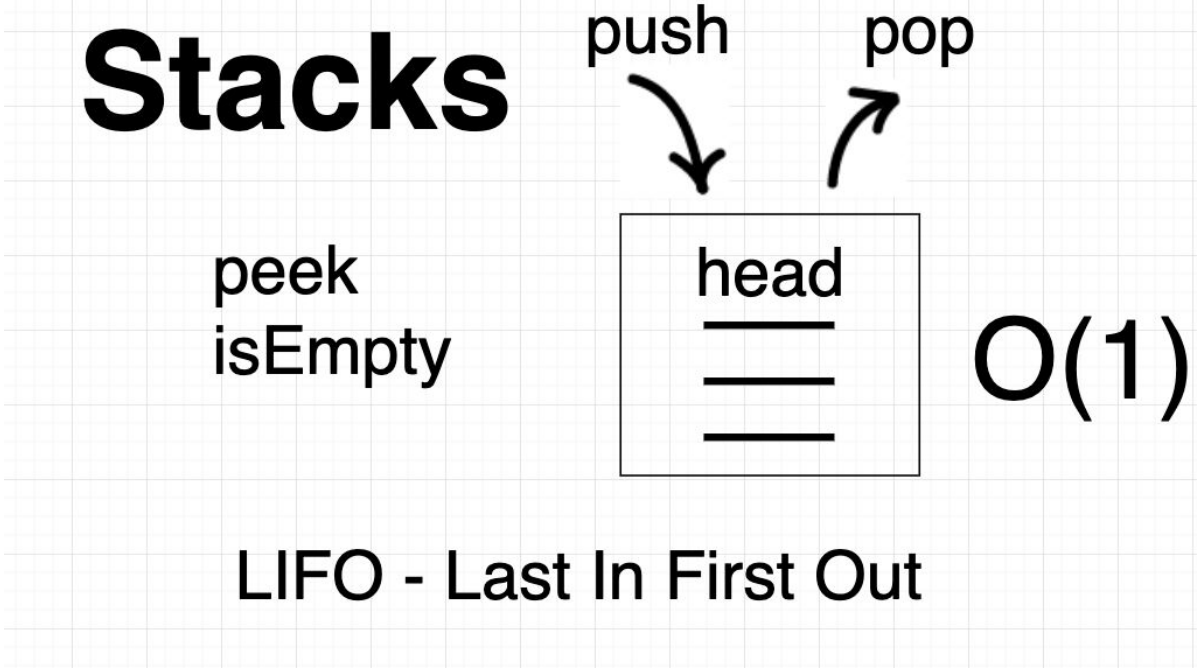
Watch for these gotchas



Stacks & Queues

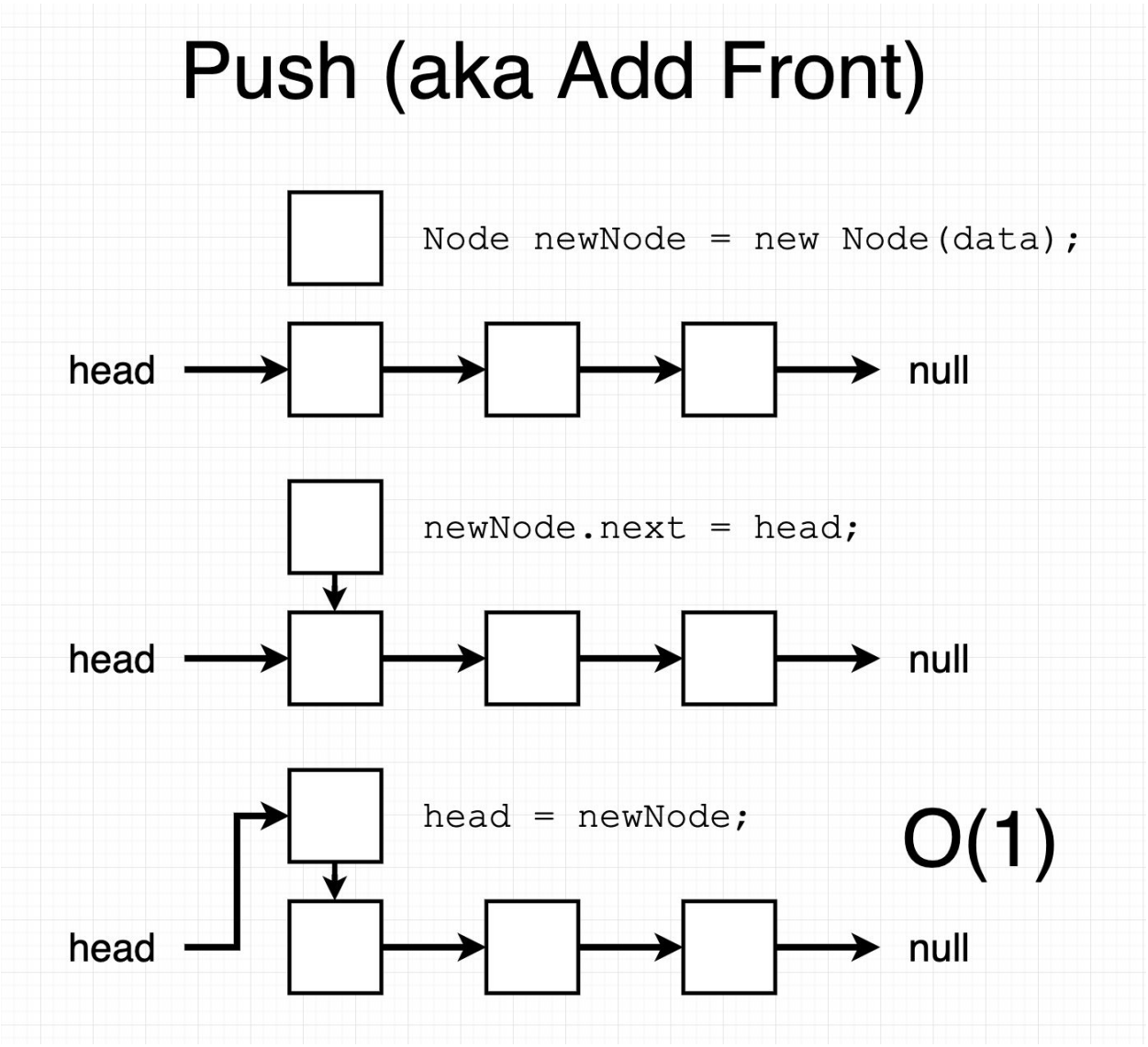


Stacks



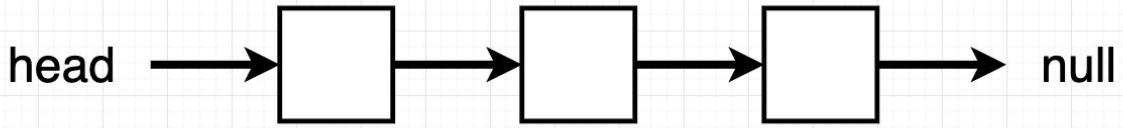
LIFO - Last In First Out

Push (aka Add Front)

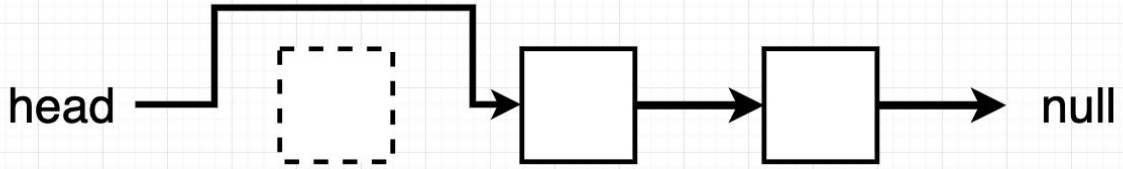


Pop

```
int data = head.data;
```

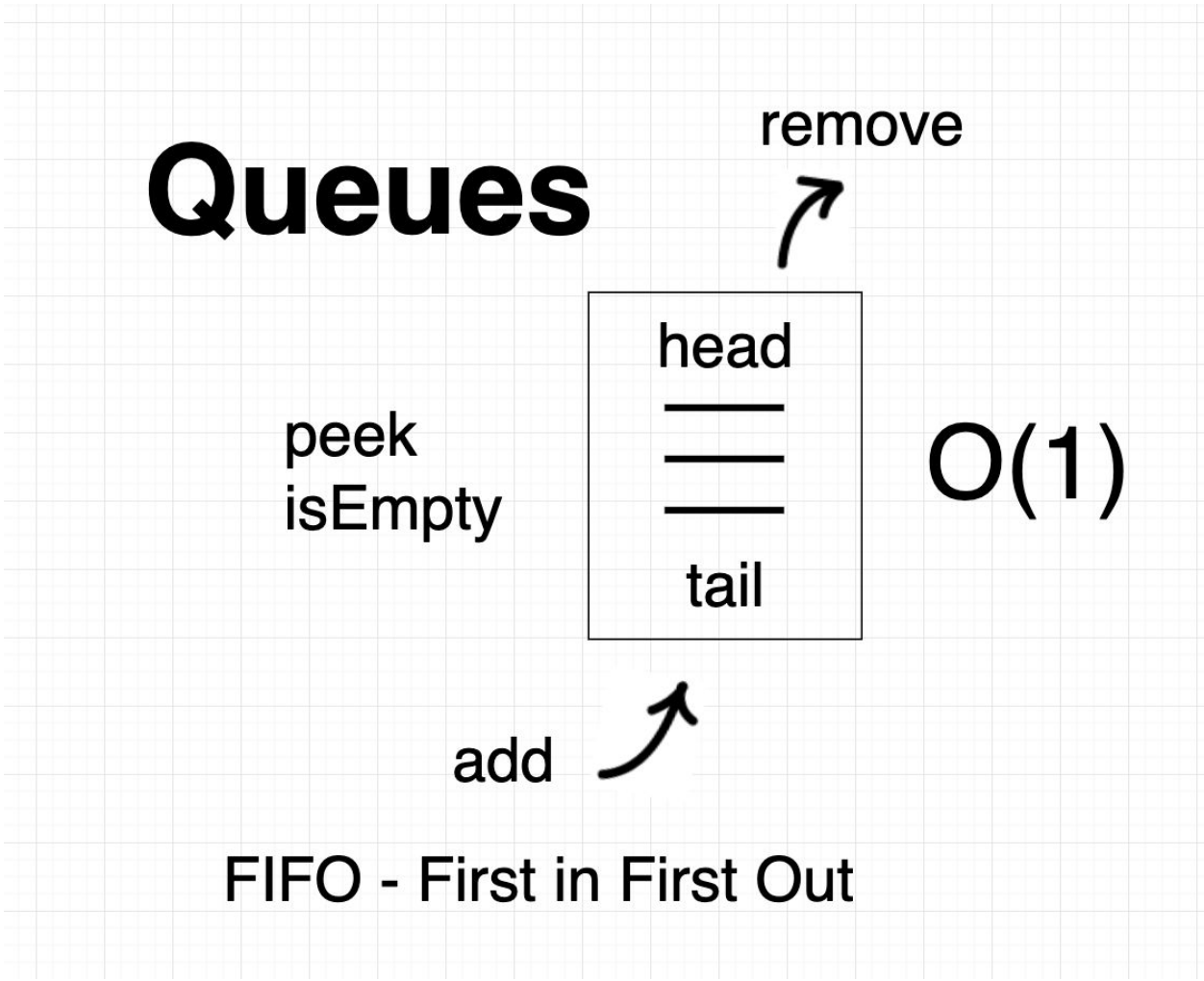



```
head = head.next;
```



O(1)

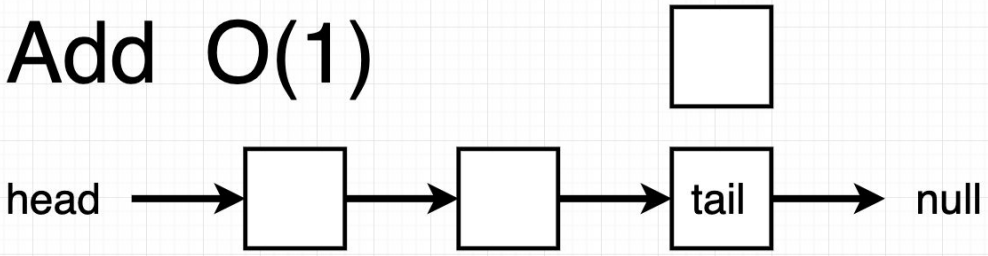
Queues



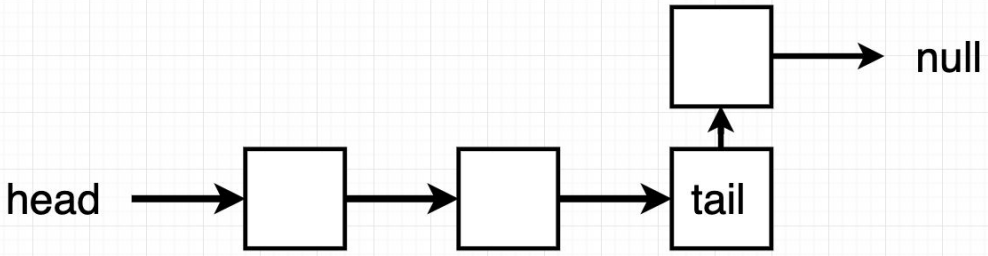
FIFO - First in First Out

```
Node newNode = new Node(data);
```

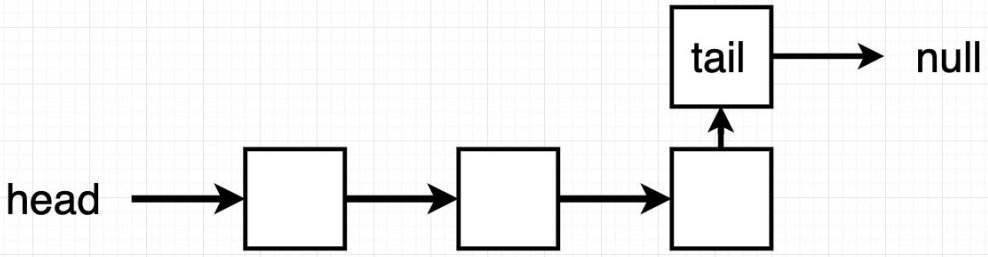
Add O(1)




```
tail.next = newNode;
```

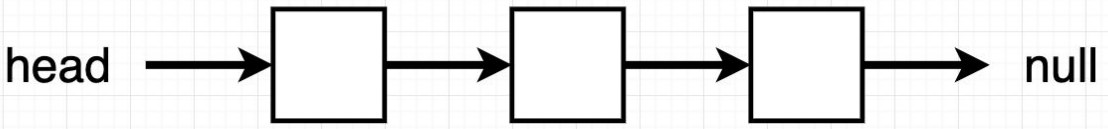



```
tail = newNode;
```

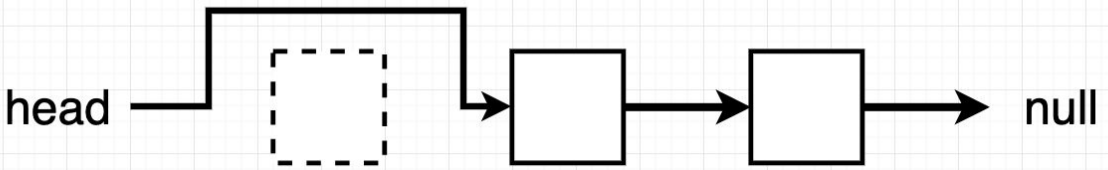


Remove O(1)

```
int data = head.data;
```

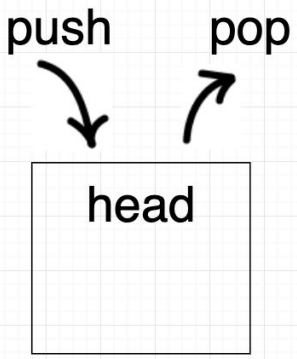



```
head = head.next;
```



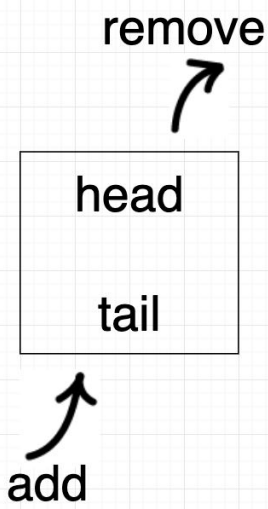
What you need to know

Stack



LIFO - Last In First Out

Queue



FIFO - First In First Out

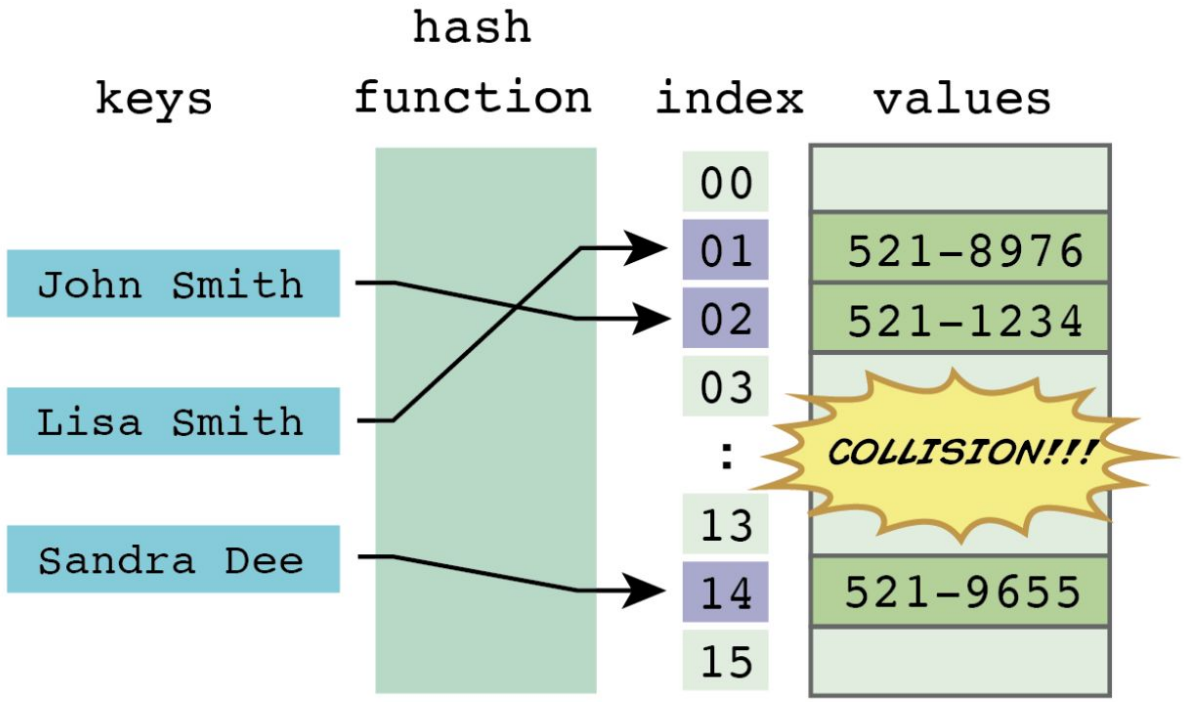
O(1)

Runtime Characteristics

Access	Search	Insertion	Deletion
O(n)	O(n)	O(1)	O(1)

http://bigocheatsheet.com

Hash Tables

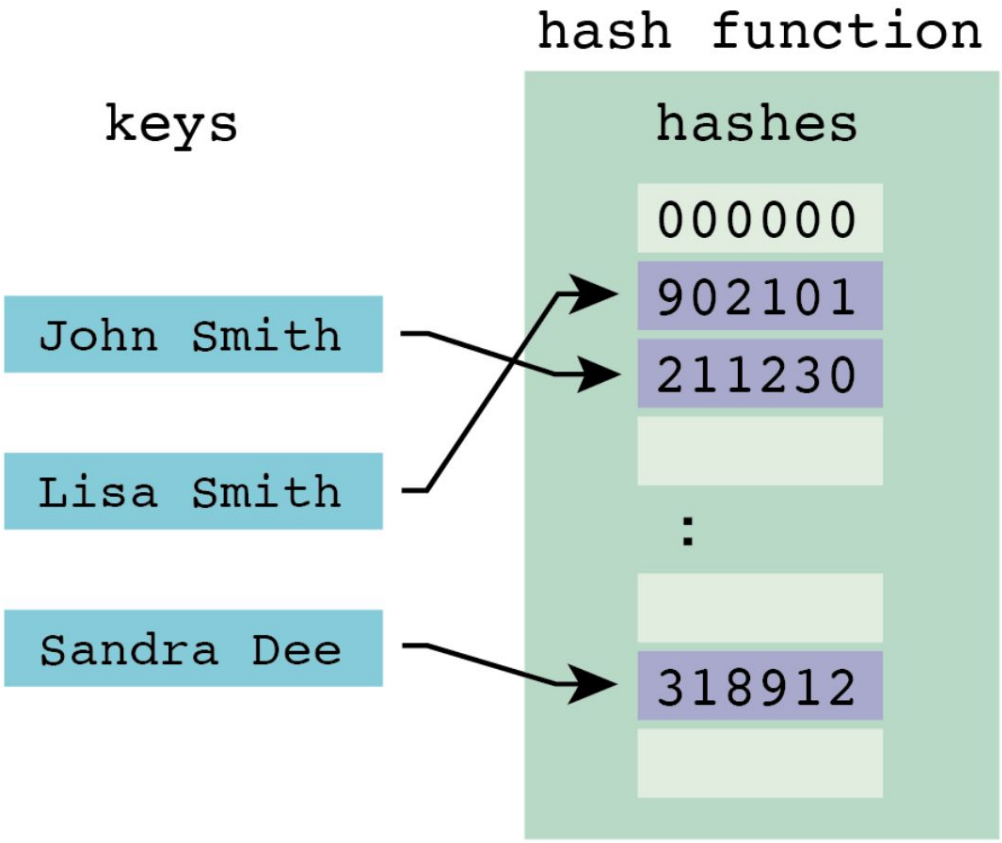


A small phone book as a hash table

https://en.wikipedia.org/wiki/Hash_table

There are three parts that go into building a hash table

1. The hash function
2. Converting the hash into an index
3. Handling collisions



- The hash function is a method we call to generate the key of the object we are trying to save
- A good hash function is
  - Fast
  - Gives a good even distribution of numbers
  - And minimizes collisions (more on that later)
- Each object we store in a hash function is responsible for generating it's own hash code

Runtime Characteristics

search, insert, delete { O(1) for a 'good' hashtable
O(n) for a terrible hashtable

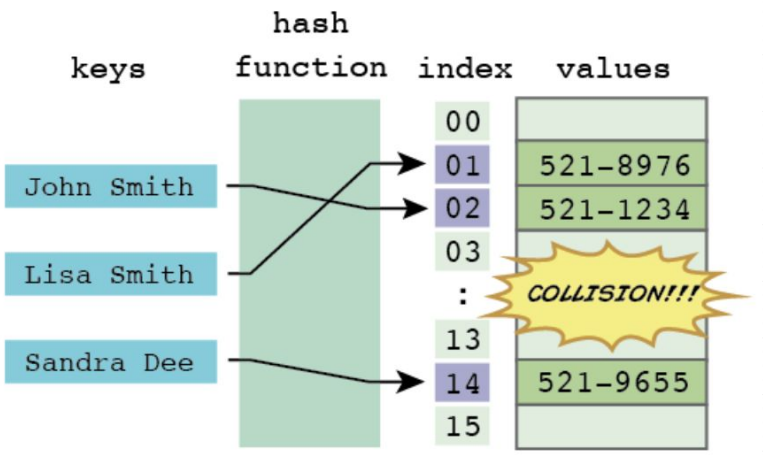
What you need to know

Super fast lookup

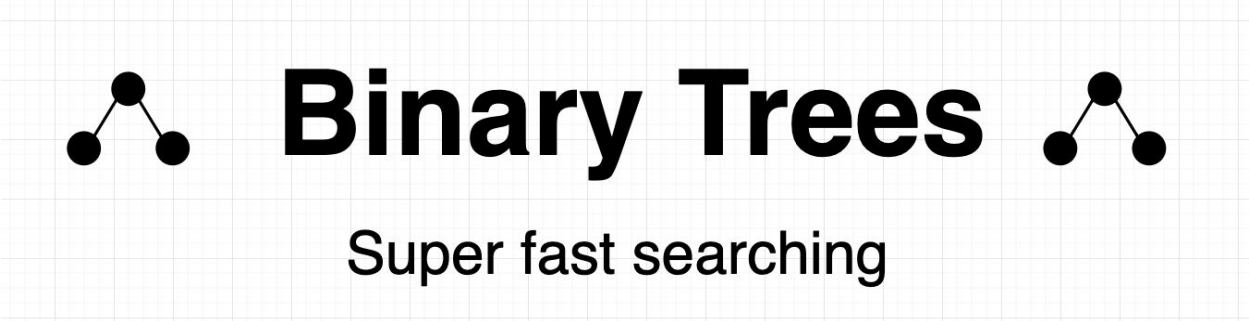
Know what hashing is

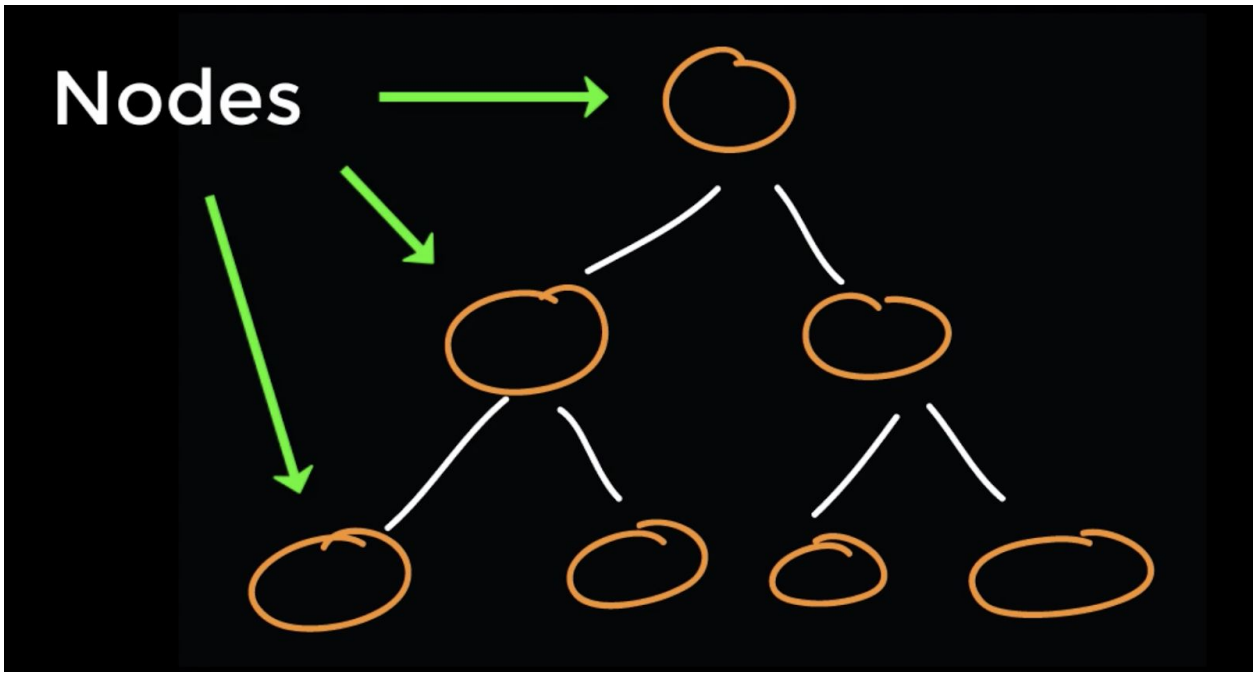
Collisions are handled via chaining

Index (not hash) used for lookup

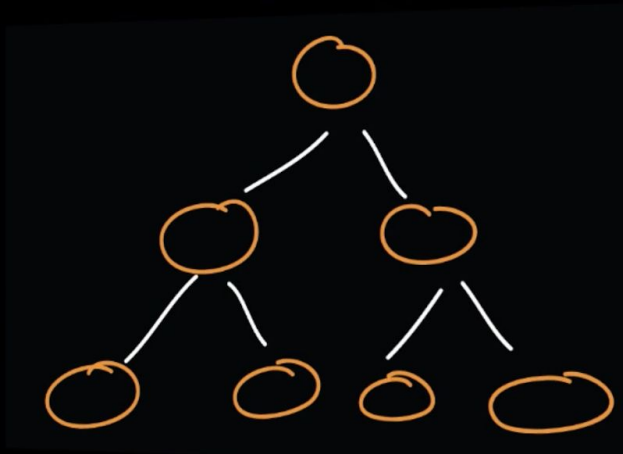


Binary Trees

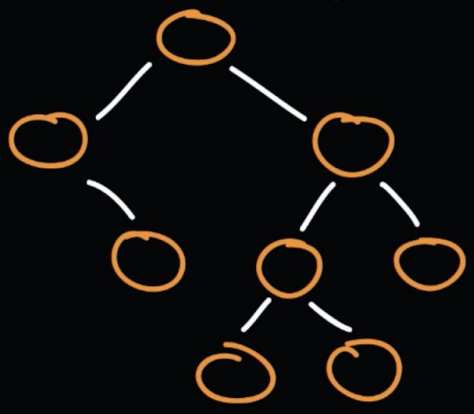




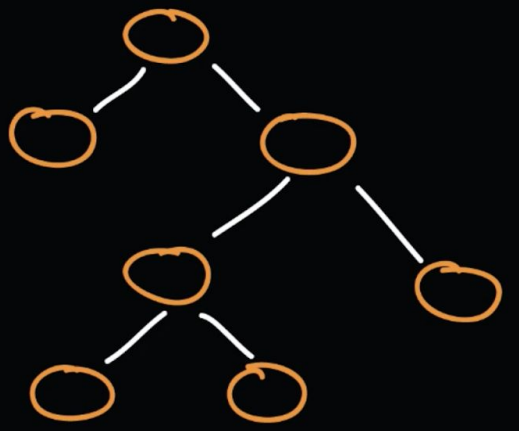
Full
Perfect
Balanced



Not Full Binary Tree

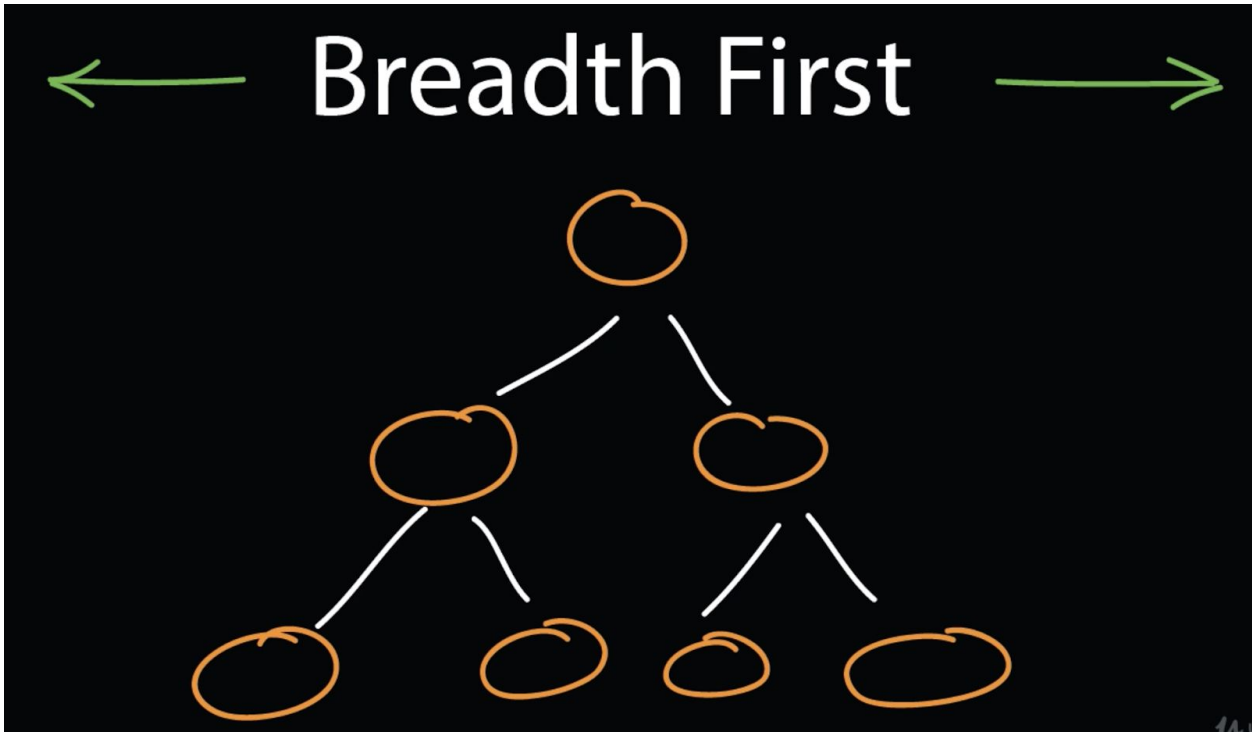


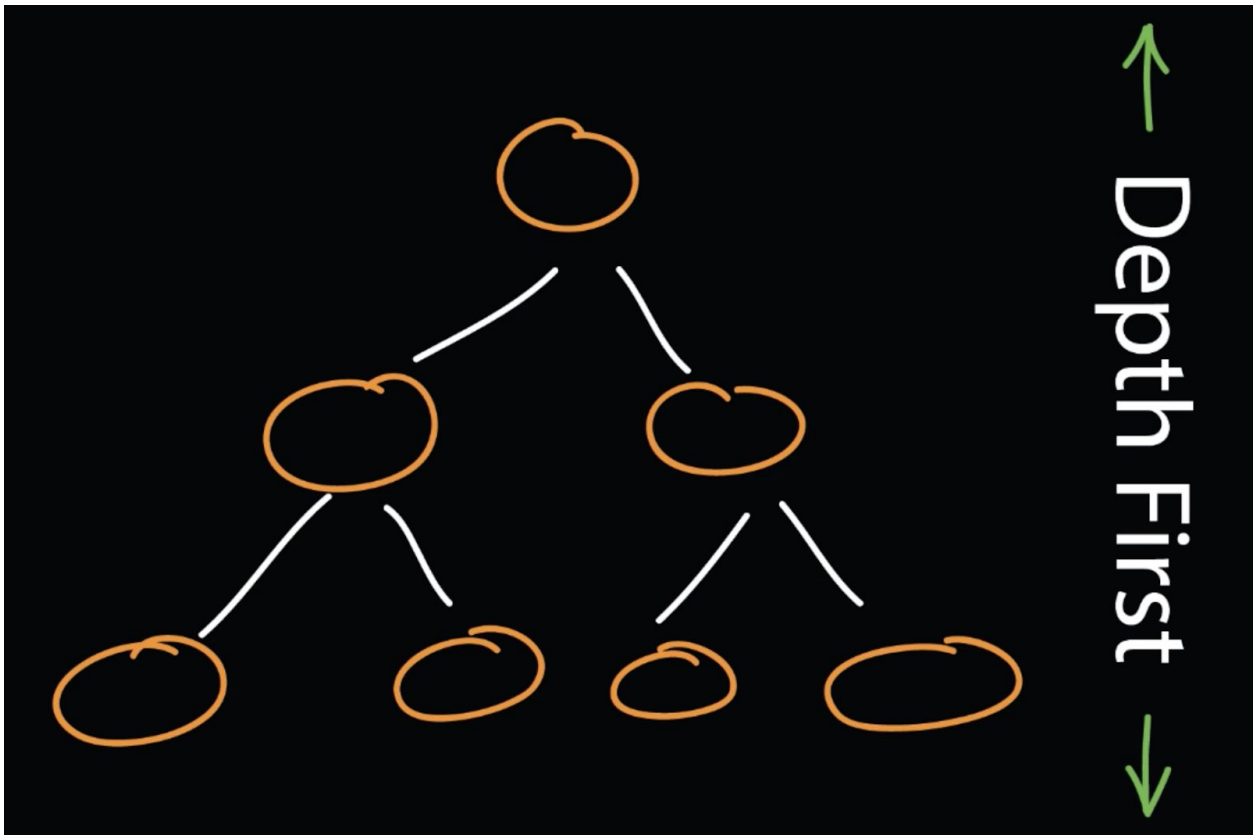
Full Binary Tree



Full Binary Tree : Tree with zero or two children

(no nodes with only one child)


Breadth First

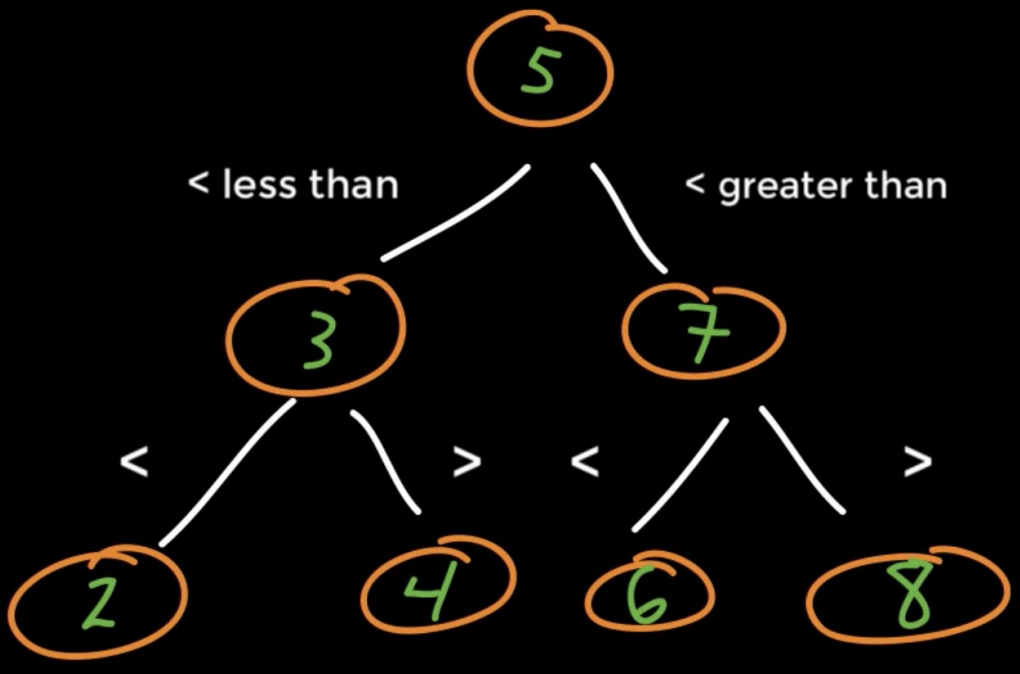

Depth First

Good terms to know

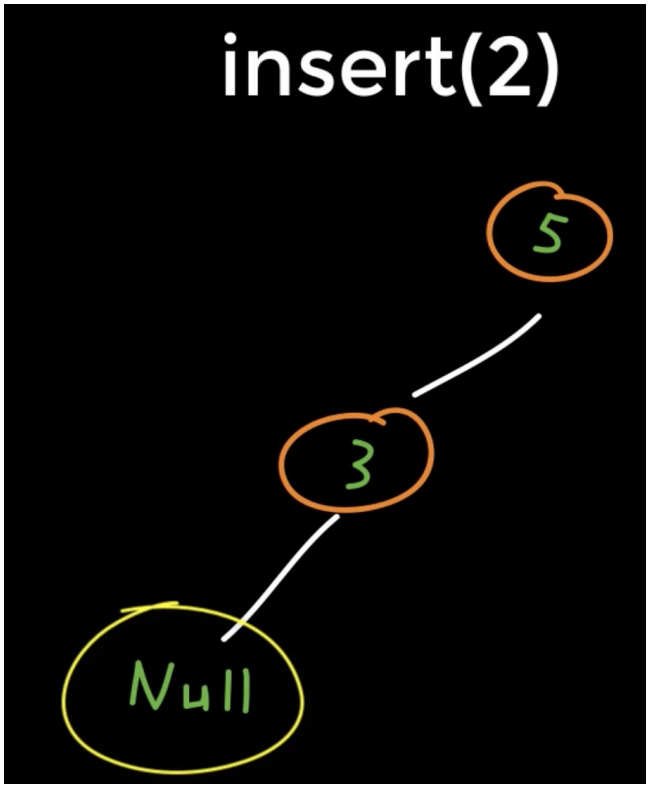
Full
Perfect
Balanced

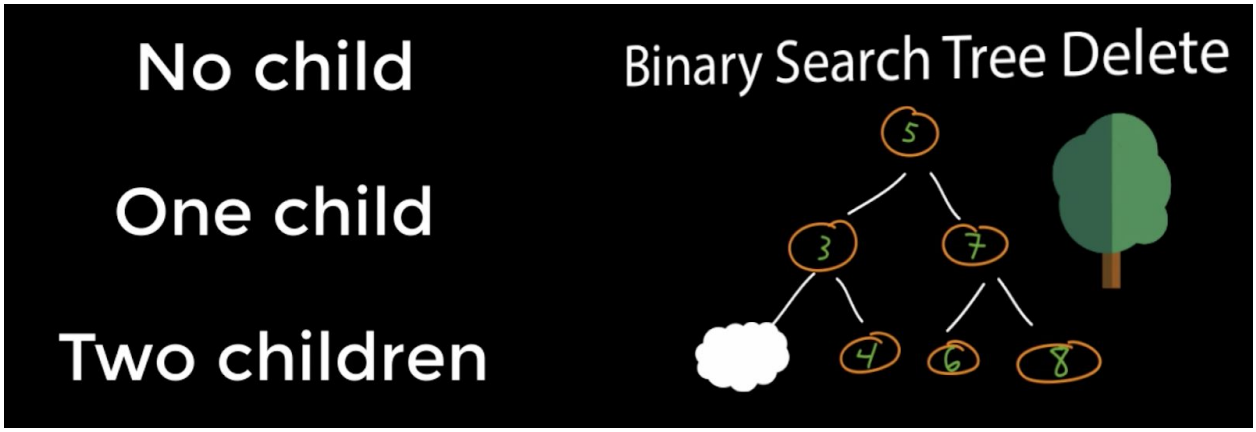
Breadth
vs
Depth

Binary Search Tree (BST)



ARE SORTED


insert(2)
5
3
Null

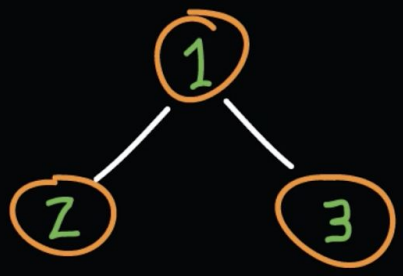

No child
One child
Two children
Binary Search Tree Delete
5
3
7
4
6
8

Depth First Traversal

Inorder

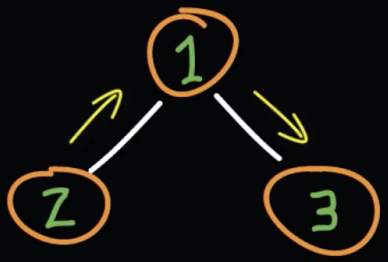
Preorder

Postorder



Depth First Traversal

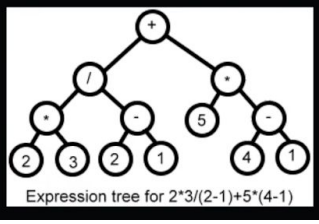
Inorder (L, Root, R) : 2 1 3



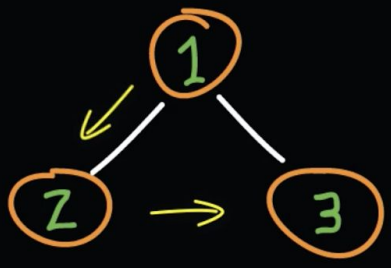
Depth First Traversal

Preorder (Root, L, R) : 1 2 3

GOOD FOR COPYING & EXPRESSION TREES



Expression tree for 2*3/(2-1)+5*(4-1)

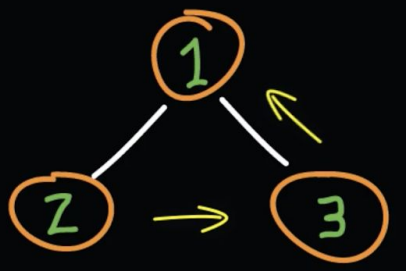


Depth First Traversal

Postorder (L, R, Root) : 2 3 1

USED IN DELETES!

```
public Node delete(Node node, int key) {
    if (node == null) {
        return null;
    } else if (key < node.key) {
        node.left = delete(node.left, key);
    } else if (key > node.key) {
        node.right = delete(node.right, key);
    } else { // Woohoo! Found you. This is the
```



Binary Search Tree Runtime

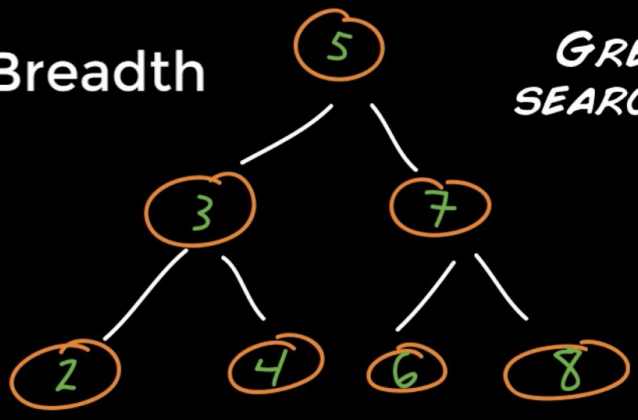
O(log n)

Find / Insert / Delete

Binary Search Trees (BST)

Depth vs Breadth

Inorder
Preorder
Postorder



GREAT FOR SEARCHING!!!

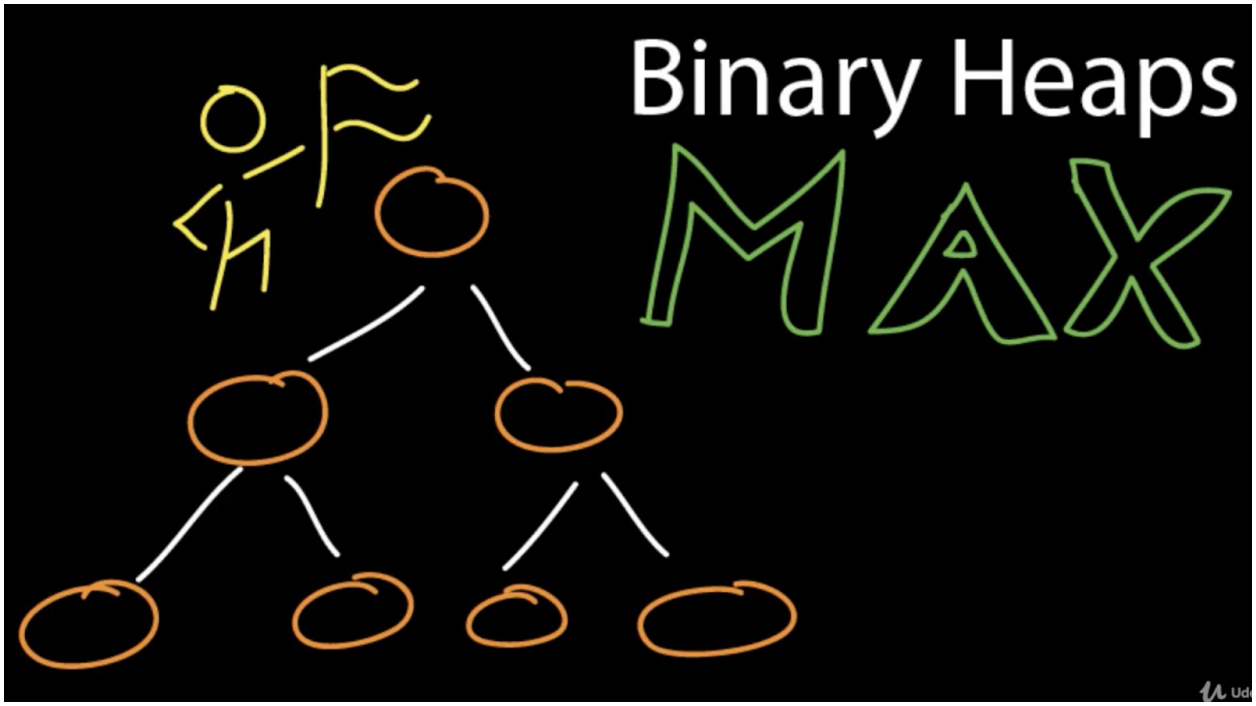
Full
Perfect
Balanced

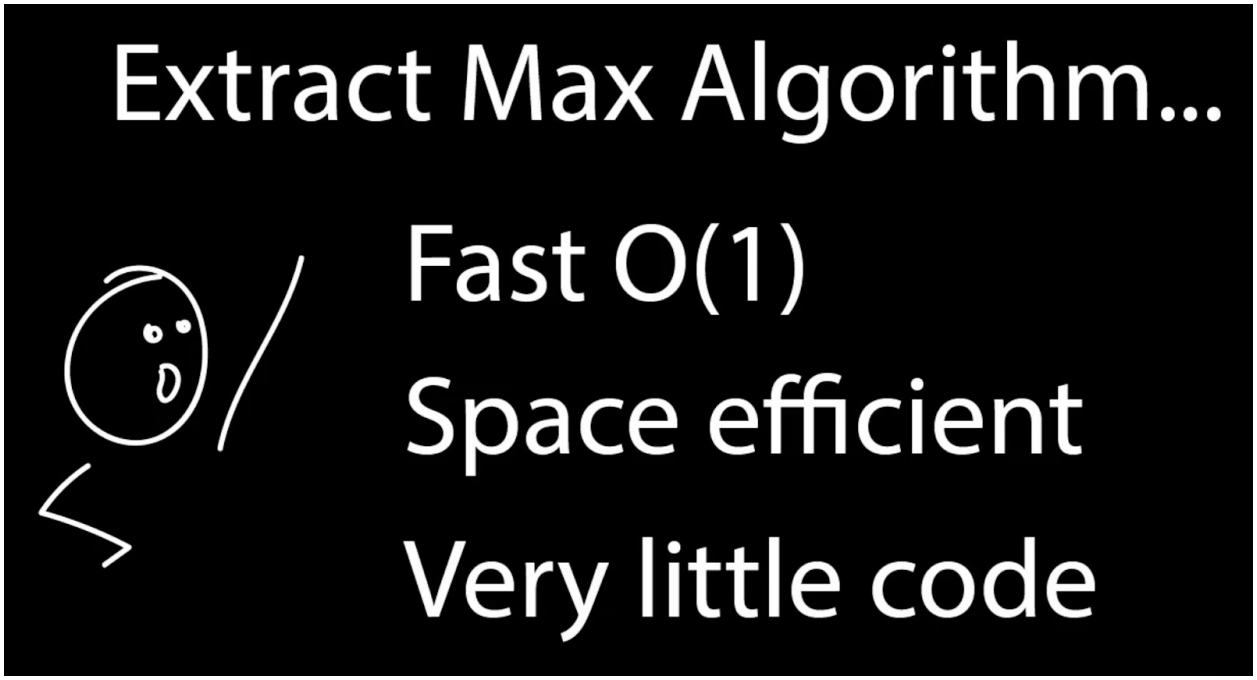
Ordered

O(log n)

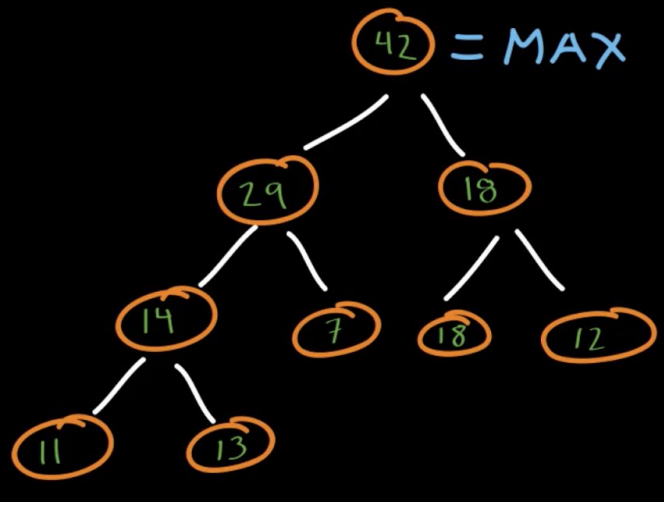
Recurisve

Binary Heaps



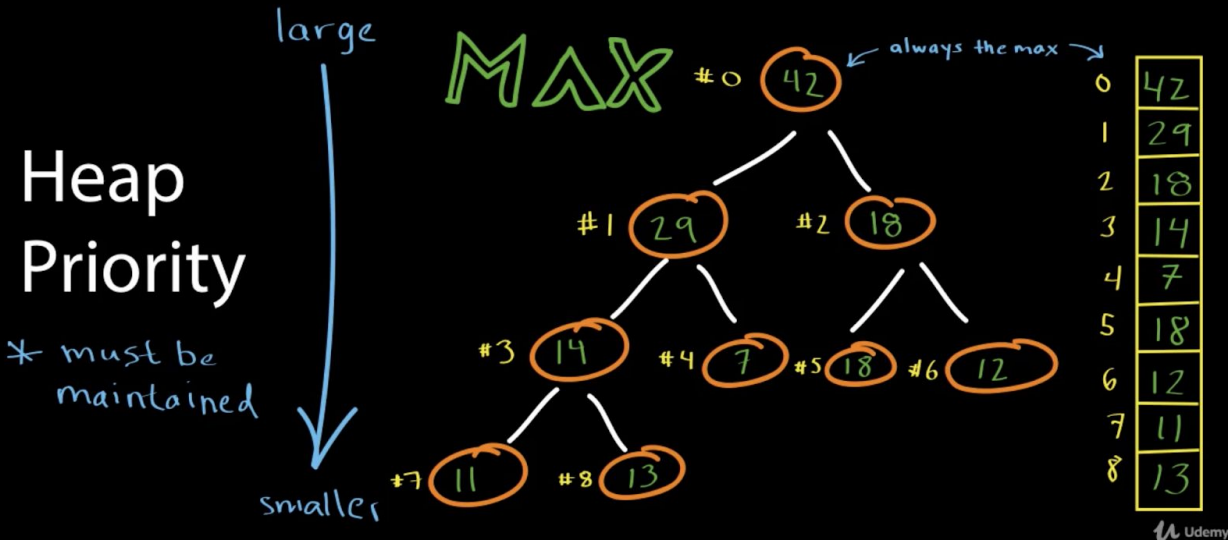


Binary Heap MAX



Routers
Shortest path
Priority queues
Kernel process scheduling

Binary Heap as an Array



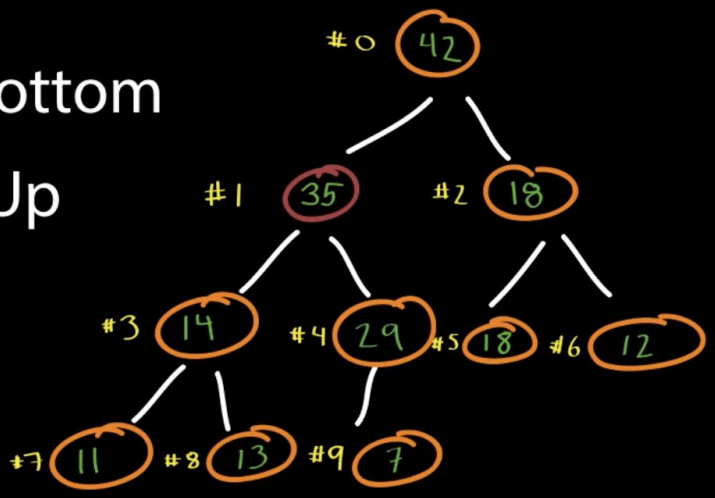
Heap Priority

* must be maintained

Handling Insert (35)

Add to bottom

Heapify Up



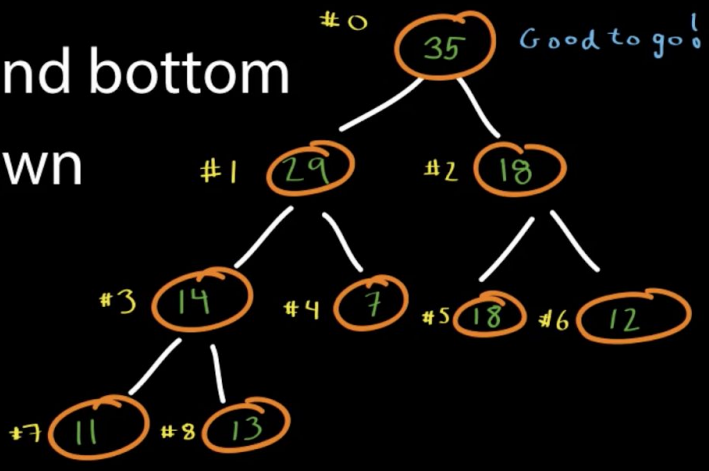
0	42
1	35
2	18
3	14
4	29
5	18
6	12
7	11
8	13
9	7

Handling Extract

Swap top and bottom

Heapify Down

Taking the larger



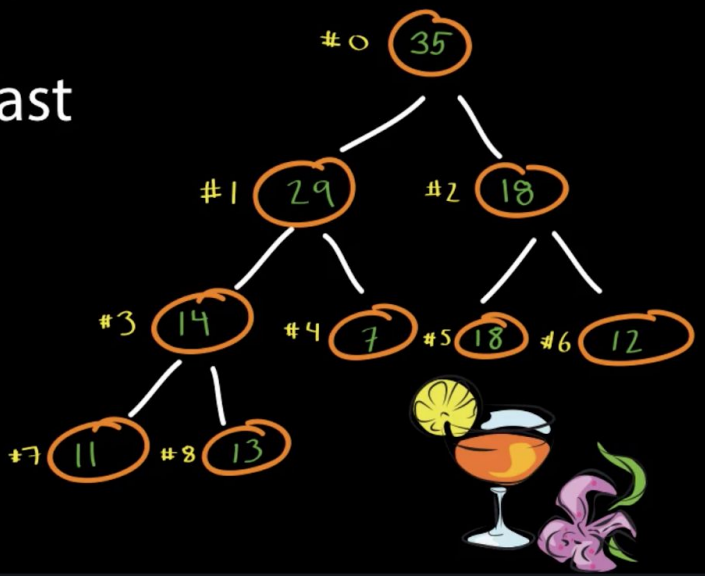
0	35
1	29
2	18
3	14
4	7
5	18
6	12
7	11
8	13
9	

Why so amazing?

Extremely Fast
ExtractMax: O(1)

Compact

Few lines

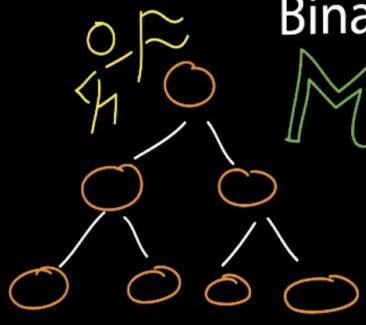


Need to know

ExtractMax: O(1)

Heap Order
Insert - Up
Extract - Down

Priority Queues
Scheduling



Binary Heaps
MAX

Fibonacci Series

What is a Fibonacci series?

1, 1, 2, 3, 5, 8, 13, 21...

Series of numbers where every number, after the first two, is the sum of the two preceding ones

$$F_n = F_{n-1} + F_{n-2}$$

Fibonacci in Code

```
public int fib(int n) {
  if (n <= 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

Memoization Defn

An optimization technique that stores expensive calculated results and returns them when asked for again.

IT'S LIKE CACHING EXPENSIVE RESULTS

Memoization

```
private int[] memo = new int[1001];

public int fib(int n) {
  if (n <= 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else if (memo[n] == 0){
    memo[n] = fib(n - 1) + fib(n - 2);
  }
  return memo[n];
}
```

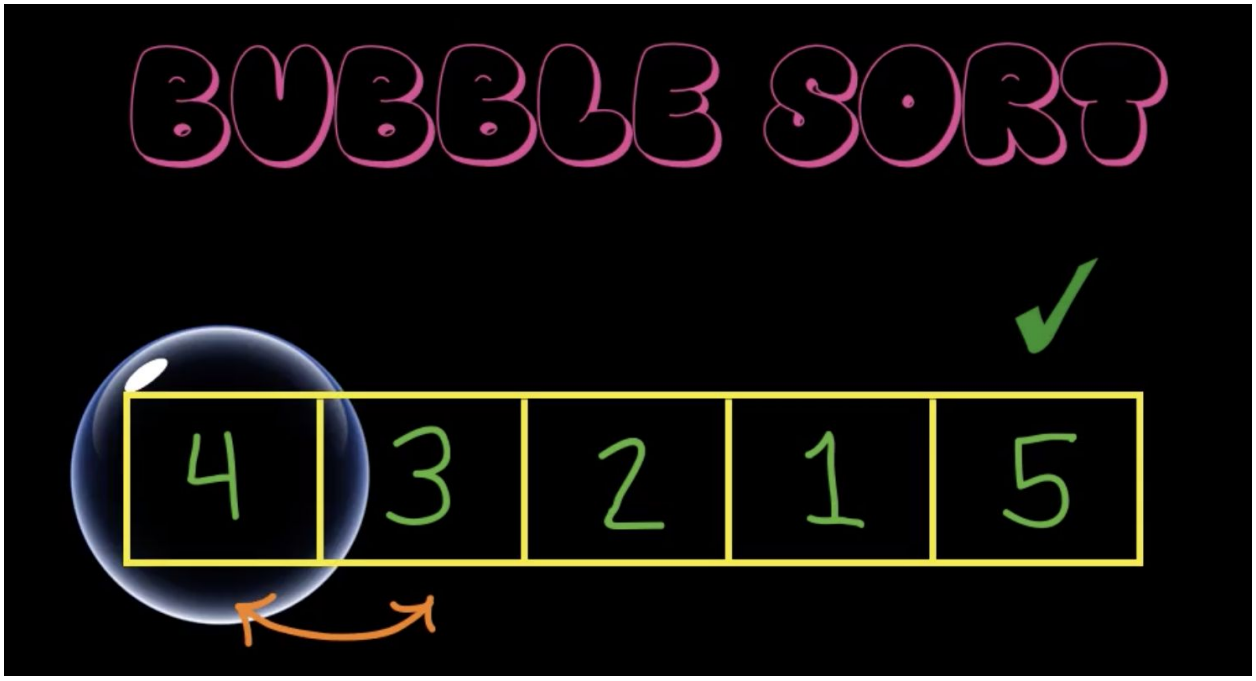
fib(1)	fib(2)	fib(3)	fib(4)	fib(5)
1	1	1	1	1

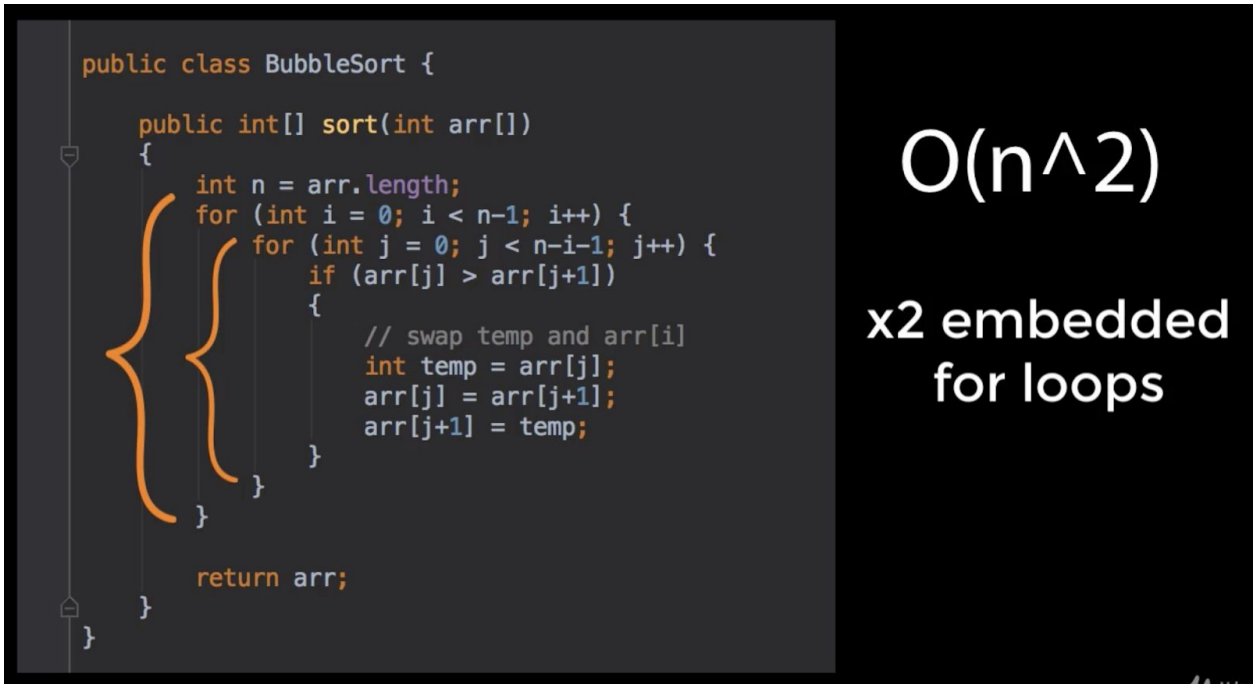
Need to know

Memorize Fib memoized

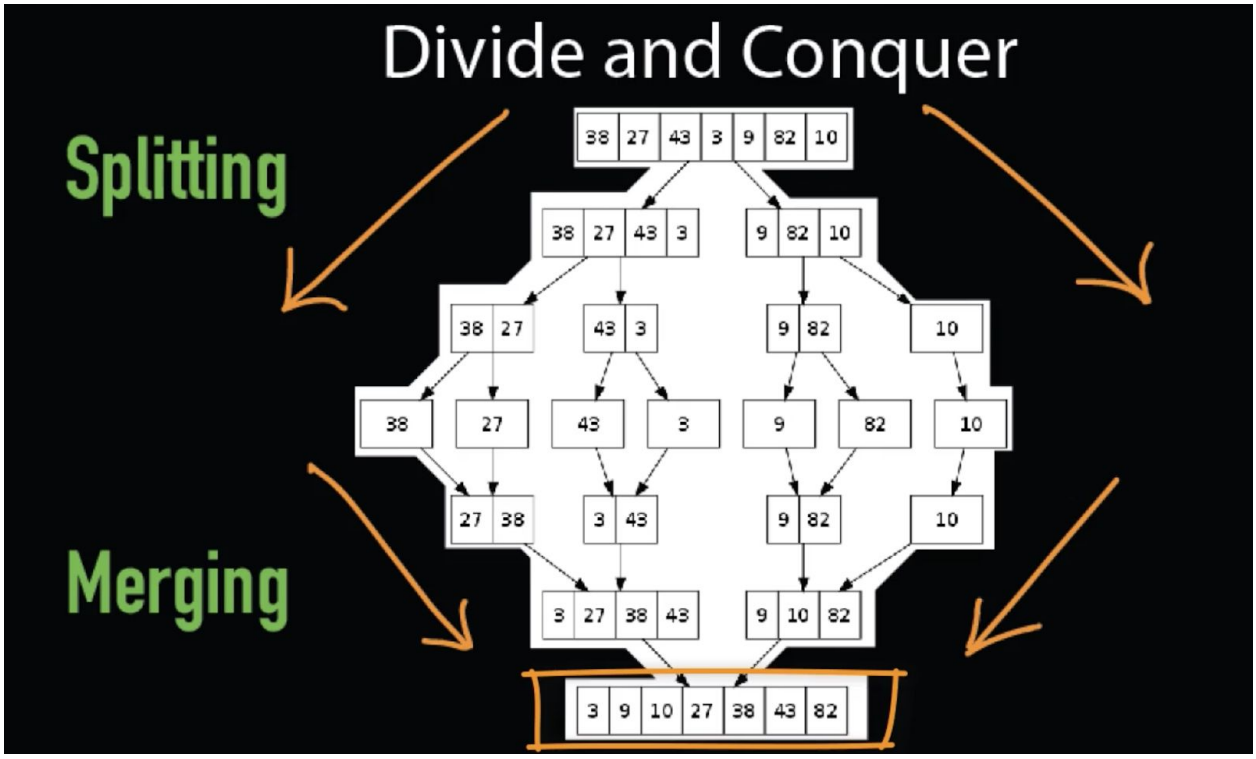
```
private int[] memo = new int[1001];

public int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if (memo[n] == 0){
        memo[n] = fib( n: n - 1) + fib( n: n - 2);
    }
    return memo[n];
}
```

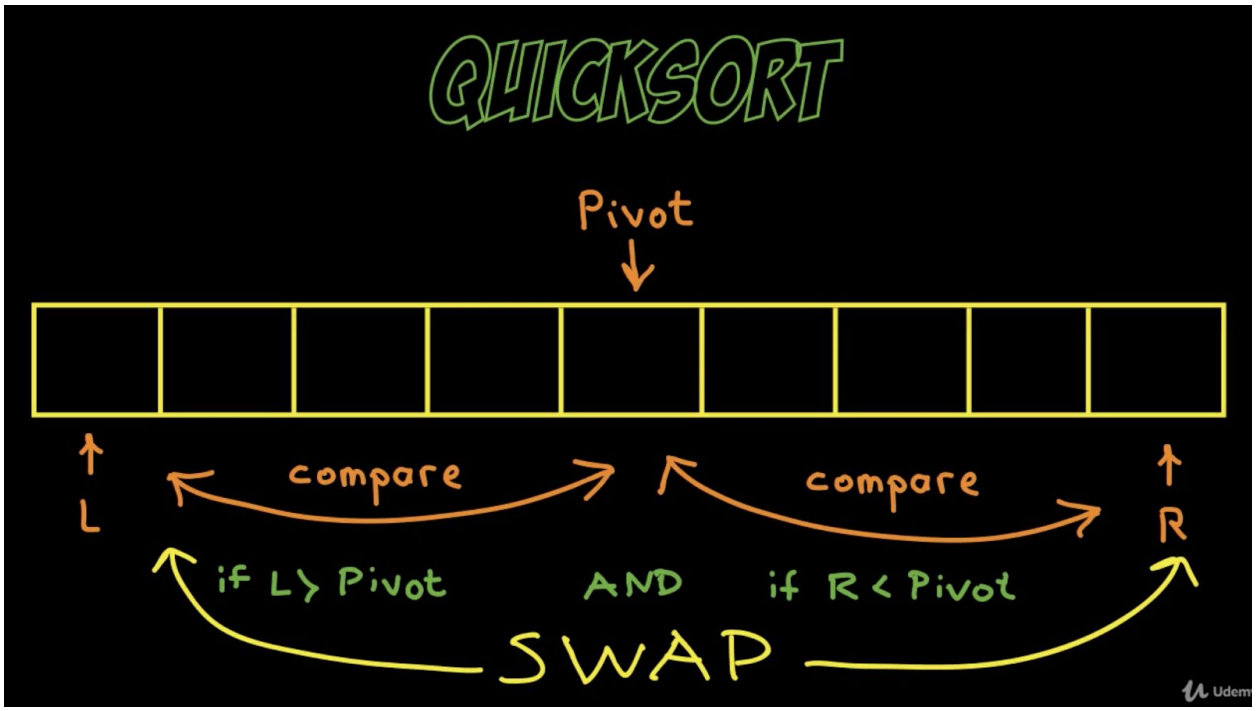
Bubble Sort


BUBBLE SORT
4
3
2
1
5


public class BubbleSort {
public int[] sort(int arr[])
{
int n = arr.length;
for (int i = 0; i < n-1; i++) {
for (int j = 0; j < n-i-1; j++) {
if (arr[j] > arr[j+1])
{
// swap temp and arr[i]
int temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
}
}
}
return arr;
}
}
O(n^2)
x2 embedded for loops

Merge Sort


Divide and Conquer
Splitting
Merging
38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82


QUICKSORT
Pivot
L
compare
compare
R
if L > Pivot
AND
if R < Pivot
SWAP

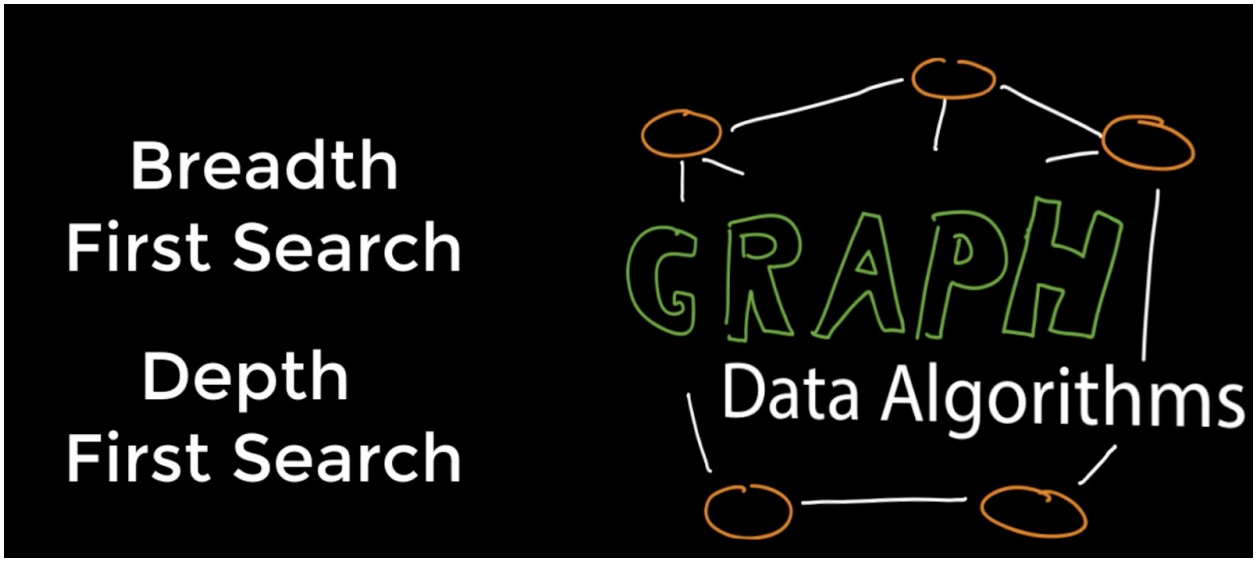
The Headlines

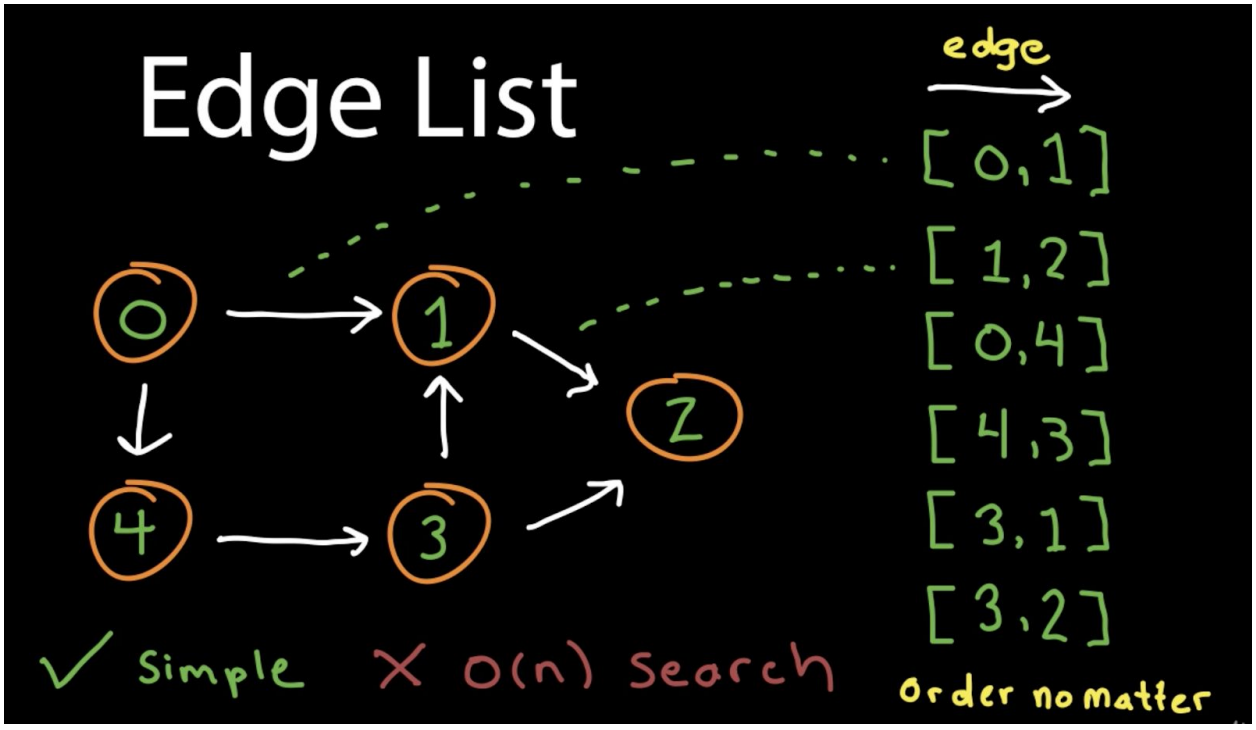
Bubble sort O(n^2)

Merge sort O(n log n)

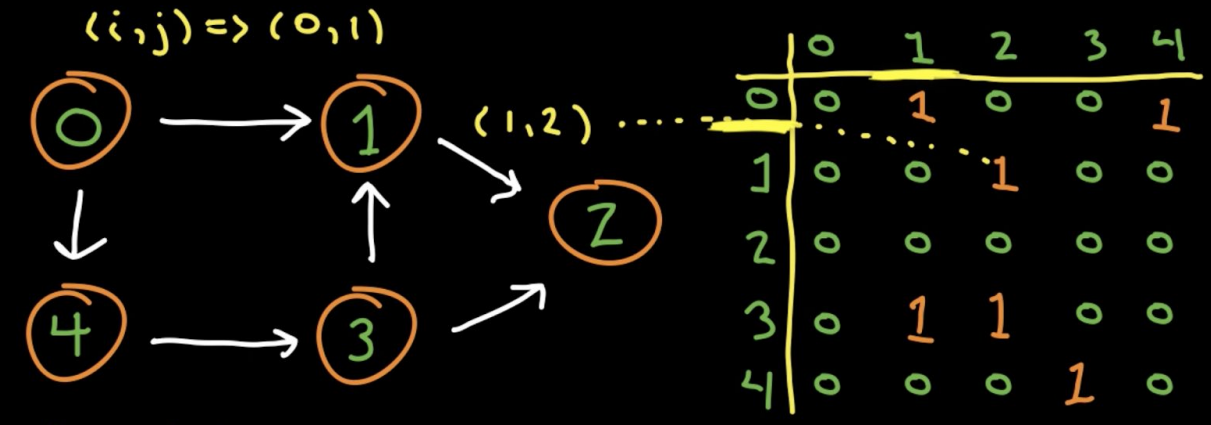
Quicksort O(n log n)

Graphs


Breadth
First Search
Depth
First Search
GRAPH
Data Algorithms

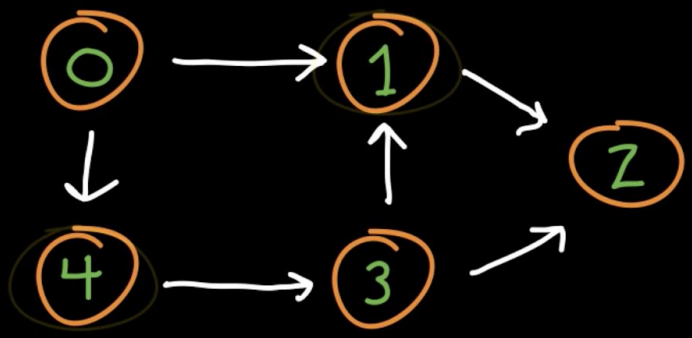

Edge List
edge
[0,1]
[1,2]
[0,4]
[4,3]
[3,1]
[3,2]
0
1
2
4
3
✓ Simple
✗ O(n) Search
order no matter

Adjacency Matrix



✓ O(1) lookup ✗ Not space efficient

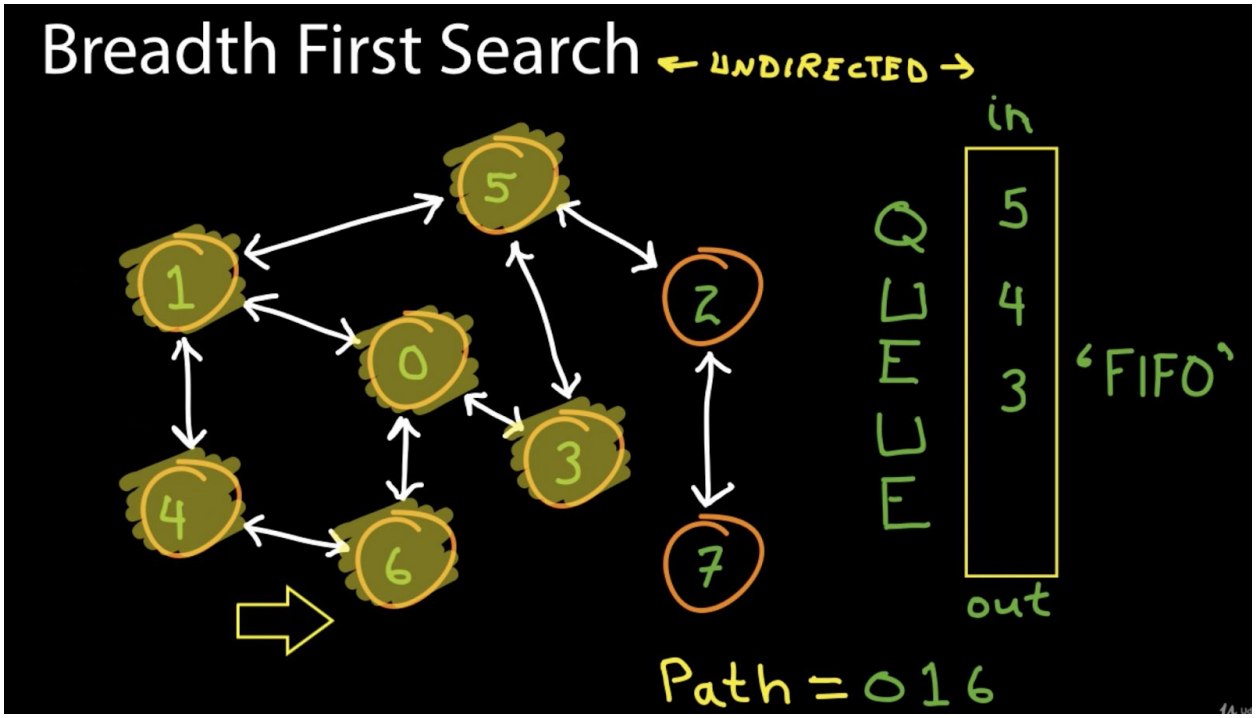
Adjacency List

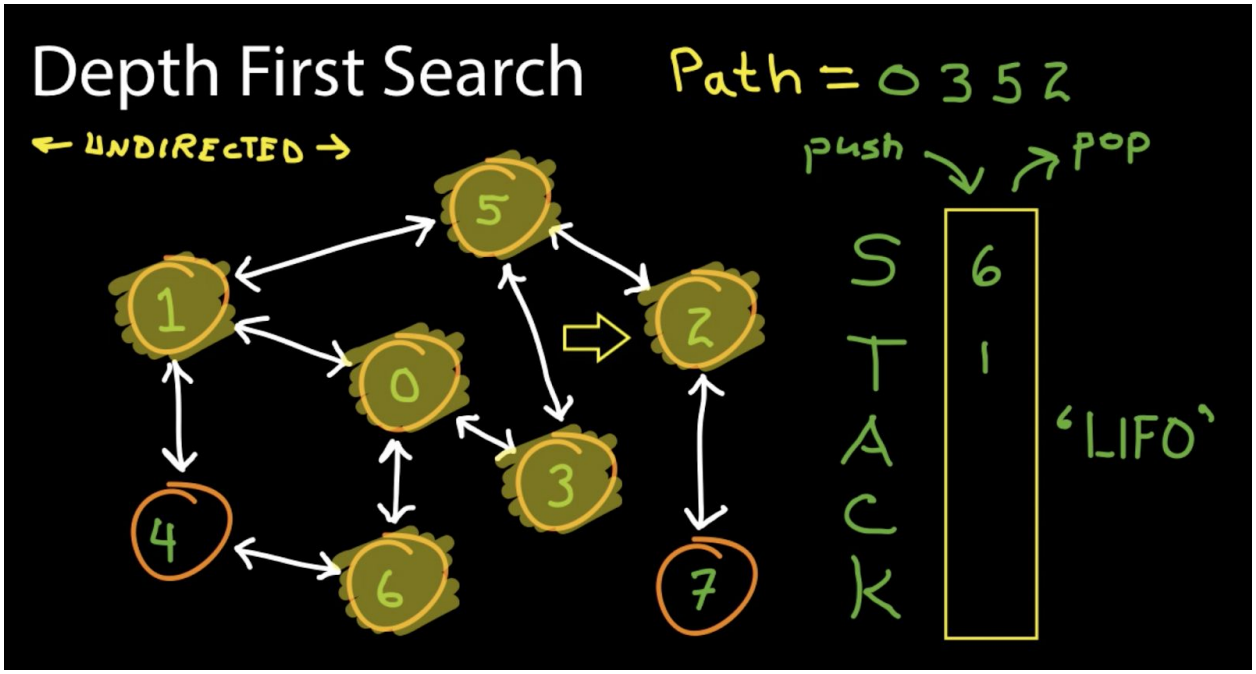


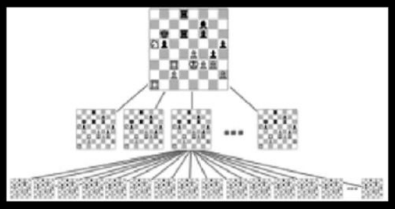
neighbours

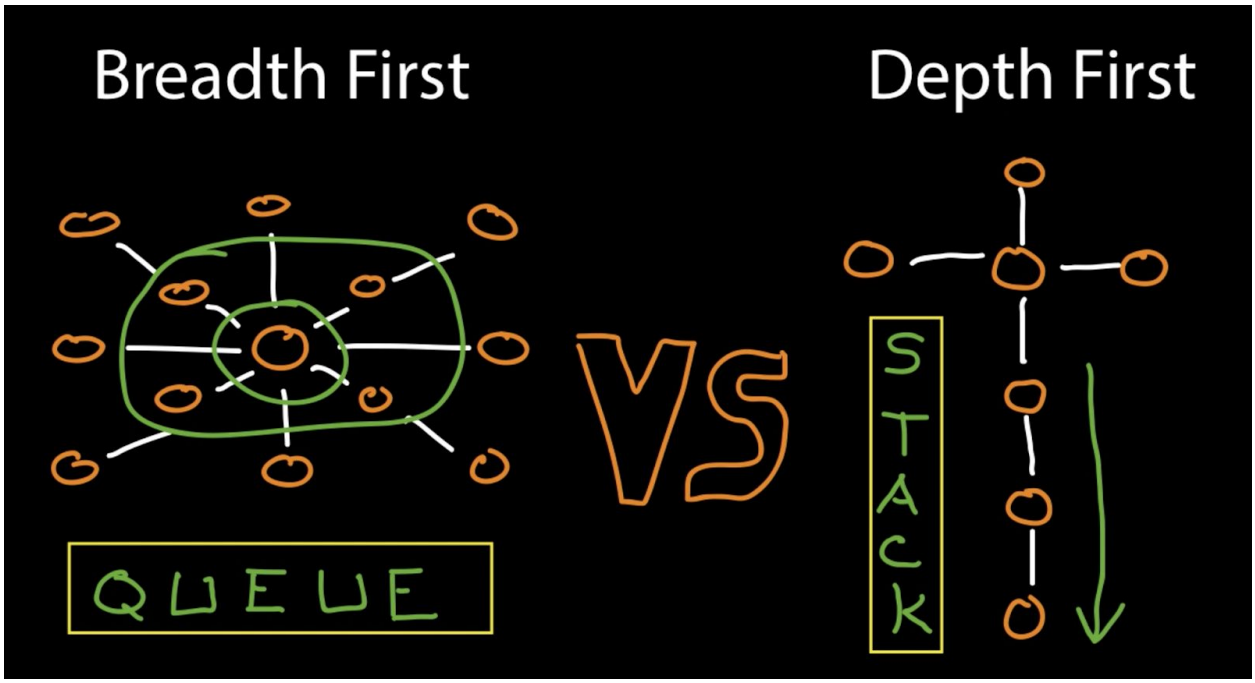
0 → [1,4]
1 → [2]
2 → [ ]
3 → [1,2]
4 → [3]

✓ Fast lookup ✓ Easy to find neighbours


Breadth First Search
← UNDIRECTED →
5
1
2
0
3
4
6
7
in
QUEUE
5
4
3
out
'FIFO'
Path = 0 1 6


Depth First Search
Path = 0 3 5 2
← UNDIRECTED →
push
pop
5
1
2
0
3
4
6
7
STACK
6
1
'LIFO'

Breadth First	Depth First
Better near the top	Better faraway
Social networks (FB, LinkedIn)	Game simulations
Nearby peers in games	
Nearest neighbour	Faraway



Good luck on all your interviews!

Cheers - Jonathan