**JS Interview Bootcamp**

[https://github.com/StephenGrider/algocasts](file:///C:\Program%20Files\MedleyText\resources\app.asar\src\dist\%5bobject%20Object%5d)

***Content***

0. Debugger (Lecture 12)

1 String Reverse

2. Palindrome

3 Integer Reversal

4 MaxChar (Perfect for String Queries)

5 Fizzbuzz

6 Chunk Array (Create small chunks fo array from the array)

7 Anagrams (Mapping table / sorting ABC)

8 Sentence Capitalization (Uppercase,Slice / string loop)

9 Steps (string concat)

10 Pyramid Steps

11. Count Vowels

12 Spiral Matrix (Keep looping with descreasing conditionals)

13 RunTime Complexity

14 Queue

15 Weave (Combine two queue)

15 Stacks

16 Queue from stacks

17 Linked List

17.1 Generators

1. Create a generator

2. Looping over the \*yielded values

3. Nest generator loopings

4. Nesting generators, inside class + looping

18. Midpoint of LinkedList

19.Circular Linked List

20. N element from last node

**Running Tests:**

jest libraryName/filename --watch

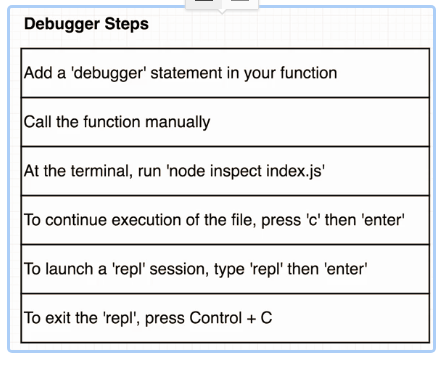
## **0. Debugger (Lecture 12)**

function reverse(str) {

debugger;

return str.split('').reduce((rev, char) => char + rev, '');

}



**1 String Reverse**

**//\*\*1. Sollution \*\*\*/// Array.prototype.reverse()**

function reverse(str) {

return str.split("").reverse().join("");

}

**//\*\*2. Sollution \*\*\*/// Reversed For loop**

function reverse(str) {

let len = str.length;

let reversed = "";

for(i=len;i>=0;i--){

reversed += str.charCodeAt(i);

}

}

**//\*\*3. Sollution \*\*\*/// For each loop**

function reverse(str) {

let reversed = '';

for (let character of str) {

reversed = character + reversed;

}

return reversed;

}

**//\*\*4. Sollution \*\*\*///Reduce**

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce>

(Basically, reduce is an accumulator, it will add everything together in a reversed order)

function reverse(str) {

return str.split('').reduce((rev, char) => char + rev, '');

//This would return the same string

return str.split('').reduce((rev, char) => rev + char, '');

}

## **2. Palindrome**

**//\*\*\*\*\* 1 Solution \*\*\* /// reverse()**

Will loop through the whole string, will seperated, will reverese each of the charachters in the array, add them together.

**3\*O(n)**

function palindrome(str) {

let string = str.split("").reverse().join("");

if(str === string){

return true;

}else{

return false;

}

}

**//\*\*\*\*\* 1 Solution \*\*\* /// (For loop)**

(More efficient, since it will just loop as long as it is true)

**O(n)**

function palindrome(str) {

let len = str.length;

for(i=0;i<len;i++){

if(str.charCodeAt(i) !== str.charCodeAt(len-1-i)){

return false;

}

}

return true;

}

It will remove all of the whiteSpaces + non alphanumerical characters

let string = str.toLowerCase()

.replace(/[^a-zA-Z 0-9]+/g,'').replace(/\s/g,'');

## **3 Integer Reversal**

**My Sollution**

I modified anythin in the string,

function reverseInt(n) {

if(n ==0) {return n; } //Check he input if it 0 return immediately

let str = n.toString(); //Convert integer to string

let len = str.length; //Define it outside the loop, for faster performance

let limit = 0; //Will do the looping i the string until the limit

let reversed =""; //Basic value

if(str.charAt(0)=="-"){

limit = 1; //Will skip the looping of the "-"

reversed ="-"; //Will add to the reversed string "-"

}

for(i=len;i>=limit;i--){ //Reversing

reversed += str.charAt(i);

}

while(reversed.charAt(limit) =="0"){ //removing zeros

reversed = reversed.replace("0","");

}

return parseInt(reversed); //returnin converted integer

}

**Course Solution**

It is reverseing the whole number as string,

parseInt() --> will remove the zeros from the begining

Math.sign() --> will return 1 or -1, so we just have to multiply it with the correct sign

function reverseInt(n) {

const reversed = n

.toString()

.split('')

.reverse()

.join('');

return parseInt(reversed) \* Math.sign(n);

}

## **4 MaxChar (Perfect for String Queries)**

Returnin the most used charachter from the string

**function** maxChar(str) {

**let** **max**=0;

//This is to store the max value

**let** maxCharacter = "";

//This is to store the max characther string, which will be returned back

**let** charsObj = {};

//This is our mapped table

**let** len = str.**length**;

//Declared outside of the for loop, so it is faster

**let** **char** = "";

//Same declared outside

**for**(i=0;i<len;i++){

**char** = str[i];

charsObj[**char**] = charsObj[**char**] +1 || 1;

//Check if it can find the charachter as a property of the object

**if** (charsObj[**char**] > **max**) {

//Instead of looping again over, we keep it O(n) if we keep track of the max value

**max** = charsObj[**char**];

maxCharacter = **char**;

}

}

**return** maxCharacter;

}

**\*\*Note\*\***

**This line**

charsObj[char] = charsObj[char] +1 || 1;

//Check if it can find the charachter as a property of the object

**Is the same as this one.**

if(!charsObj[char]){

//If it can't find the property, it will create one and assign a value of 1 to it

charsObj[char] = 1;

}else{

charsObj[char]++;

//Else it will just add one to it.

}

## **5 Fizzbuzz**

**MySolution**

function fizzBuzz(n) {

for(i=1;i<=n;i++){

if( i%3===0 & i%5===0 ){

console.log("fizzbuzz");

}else if(i%3===0){

console.log("fizz");

}else if(i%5===0){

console.log("buzz");

}else{

console.log(i);

}

}

}

## **6 Chunk Array (Create small chunks fo array from the array)**

**My Solution**

Using a temporary array + for loop

// chunk([1, 2, 3, 4], 2) --> [[ 1, 2], [3, 4]]

// chunk([1, 2, 3, 4, 5], 10) --> [[ 1, 2, 3, 4, 5]]

function chunk(array, size) {

if (size <= 0 || size>= array.length){ return array};

//Return immedeitaly, if it is 0, or bigger, or the same size

let finalArr = [];

//Final array, to be returned

let tempArr = [];

//Temporary array for the chunks

let limit = size;

//Limit for the temporary array size

let len = array.length;

//TO make the for loop faster

for(i=0;i<len;i++){

if(tempArr.length <limit){

//If the chunk is not full push the element here tempArr.push(array[i]);

} else{

//Else push the chunk to the array, empty it, and push the new element into the chunk

finalArr.push(tempArr);

tempArr = [];

tempArr.push(array[i]);

}

}

if(tempArr.length >0){ //Push the reamining chunk if it is not empty

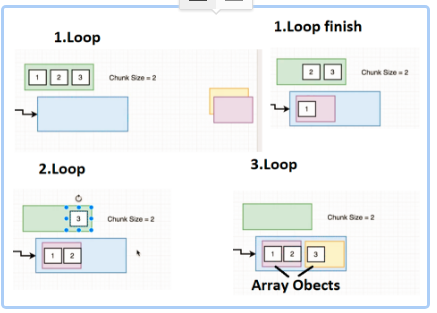
finalArr.push(tempArr);

}

return finalArr;

}

**Course Solution 1**



**Explanation**

**0.**

- We create a chunked array, which will be the created array of storing the chunk arrays.

1. Loop:

- last: will be undefined since chunked is empty therefore !last ,(!undefined) will be true

- Thus, it will push a new array object, to the chunked array

1. Loop finish:

- So are a result, we are storing a reference to the array object, which is pushed to the chunked array

2. Loop:

- Since ourobject, is not undefined and full, we push the current element to the last object, which is in the chunked array!!

3. Loop:

- Since the last array object is full, we push a new array object, with the current element in it to the chunked array.

**\*\*Trick:**

- last array object is passed by reference, and pushed inside of an other array,(chunked)

- So if you push anything to the last array, it will be a part of chunked as well.

\*/

**Course Solution 1**

function chunk(array, size) {

const chunked = [];

//Tha main array, which holds the chunked arrays.

//If the input array.lengt is 0, it won't loop.

for (let element of array) { //Looping through each element

const last = chunked[chunked.length - 1];

//Keeping track of the last element, we are referencing array objects, not storing it by value!

if (!last || last.length === size) {

//Will push a new array object, into chunked, if, last is not defined, or the size has been reached

chunked.push([element]);

} else {

//Will push the element to the last array ojbect, and since we are storing the last value by reference, it will push to the array, stored in the chnked array.

last.push(element);

}

}

return chunked;

}

console.log(chunk([1,2,3,4,5,6],2));

module.exports = chunk;

**Course Solution 2**

**//We are looping through the array, and we slice, chunked arrays from the original array and that's what we push into the cunked array.**

function chunk(array, size) {

const chunked = [];

let index = 0;

while (index < array.length) {

chunked.push(array.slice(index, index + size));

index += size;

}

return chunked;

}

## **7 Anagrams (Mapping table / sorting ABC)**

**// anagrams('rail safety', 'fairy tales') --> True**

**// anagrams('Hi there', 'Bye there') --> False**

**Solution 1 (Creating Mapping)**

function anagrams(stringA, stringB) {

const aCharMap = buildCharMap(stringA);

const bCharMap = buildCharMap(stringB);

if (Object.keys(aCharMap).length !== Object.keys(bCharMap).length) { //if the charachter lenghts do not match, it is false

return false;

}

for (let char in aCharMap) { //check the index values.

if (aCharMap[char] !== bCharMap[char]) {

return false;

}

}

return true;

}

//so we get a string as an argument, we remove the non latin characthers, we make everythin lowercase and we make return the mapping

function buildCharMap(str) {

const charMap = {};

for (let char of str.replace(/[^\w]/g, '').toLowerCase()) {

charMap[char] = charMap[char] + 1 || 1;

}

return charMap;

}

**Solution 2**

function anagrams(stringA, stringB) {

return cleanString(stringA) === cleanString(stringB); //If they perfectly match, they are an anagram!!

}

function cleanString(str) {

return str

.replace(/[^\w]/g, '') //remove the non latin characthers

= .toLowerCase() //not case sensitive

.split('') //Create an array

.sort() //Sort it according to the ABC

.join(''); //Create a string from them.

}

## **8 Sentence Capitalization (Uppercase,Slice / string loop)**

**My solution**

**//Really important, you can't overwrite only just a letter in a string!**

**//** arr[i][0] = "x" is just not working

function capitalize(str) {

let arr = str.split(" ");

for(i=0;i<arr.length;i++){

arr[i] = arr[i][0].toUpperCase()+ arr[i].slice(1);

}

return arr.join(" ");

}

console.log(capitalize('i am a cool guy'));

**Course solution2**

**//o we are looping over the whole string and recreating it letter by letter,**

function capitalize(str) {

let result = str[0].toUpperCase();

for (let i = 1; i < str.length; i++) {

if (str[i - 1] === ' ') { // if the previoues charachter is a space, then ..

result += str[i].toUpperCase();

} else { //Else just copy it to the new string.

result += str[i];

}

}

return result;

}

## **9 Steps (string concat)**

// --- Examples

// steps(2)

// '# '

// '##'

// steps(3)

// '# '

// '## '

// '###'

//console.logging all of the levels seperately

function steps(n) {

let str = ''; //Initial value for our returning text

let hash = ""; //Initial value of our hash concatenated

let space = ""; //Initial value of our hash concatenated

for(i=1;i<=n;i++){ //looping for as many rows as the recieved integer

if(n!=1){ //If we provide 1, we just return a prewritten string

space = "";

//In every row, space amount is chaning, has to be set to ""

hash +="#";

//In every row, from 1 we increment this string with 1 hash

for(j=1;j<=n-i;j++){

space += " ";

//We calculate how many spaces do we have in a row, in row 2 we have (total-2) spaces

}

} else{

return console.log('#' + space );

//If we provide 1, we just return a prewritten string

}

console.log( hash + space ) ;

//Will be modified in each loop (console.log each row)

str += '\''+ hash + space + '\'\n';

//Will be modified in each loop (concatenate the string)

}

}

steps(2);

**Course Solution 1**

//Rewritten with for loop.

function steps(n) {

let str = ''; //Initial value for our returning text

for(i=0;i<n;i++){ //looping for as many rows as the recieved integer

str ="";

for(j=0;j<n;j++){

if(j<=i){

str +="#";

}else{

str +=" ";

}

}

console.log(str) ; //Will be modified in each loop (console.log each row)

}

}

steps(3);

**Solution 1 Rewritten**

//Rewritten with for loop.

function steps(n) {

let str = '';//Initial value for our returning text

for(i=0;i<n;i++){ //looping for as many rows as the recieved integer

str ="";

for(j=0;j<n;j++){

if(j<=i){

str +="#";

}else{

str +=" ";

}

}

console.log(str) ;

//Will be modified in each loop (console.log each row)

}

}

steps(3);

**Recursive Solution Mine (Recursing rows)**

//n is the number, how many times, it should be console.log in total

//hash --> is the current string of "#"

function steps(n, hash="#") {

let str = hash;

if(n===1){

//This is our base, if we reach 1, we return the concatenated "#" string.

return console.log(str);

}

for(i=0;i<n-1;i++){

//If not, we add to the current hash the spaces which in total is(n-1)

str +=" ";

}

console.log(str) ;

//Concatenated hashes with spaces

steps(n-1,hash+"#");

//We decrement the rows to be shown, + add basi hash string.

}

**Recurson Course solution 2**

**Recursons are done for every charachter in the row.**

function steps(n, row = 0, stair = '') {

if (n === row) {//The and of the cube, last cell in the table.

return;

}

if (n === stair.length) {

//If we run through each column, we console.log the current row + set string to default

console.log(stair);

return steps(n, row + 1); //and go to the next one

}

const add = stair.length <= row ? '#' : ' ';

// for every row, if there are still column to be looped through

// We add a # or "space" if the column lenght is smaller, or less then the current row

steps(n, row, stair + add); //Since there are other columns to be looped through, we just call the funciton again

}

## **10 Pyramid Steps**

**// \*\*\*\*\*Recursion One \*\*\*\***

function pyramid(n, hash="#") {

if(n===1){

//This is our base, if we reach 1, we return the concatenated "#" string.

return console.log(hash);

}

let space = '';

for(i=0;i<n-1;i++){

//If not, we add to the current hash the spaces which in total is(n-1)

space +=" ";

}

console.log(space + hash + space) ; //Concatenated hashes with spaces

pyramid(n-1,hash+"##");

//We decrement the rows to be shown, + add basi hash string.

}

**//\*\*\*\*\*\*Normal loop with creating the rows\*\*\*\***

// function pyramid(n) {

// let hash ="#";

// for(i=0;i<n;i++){ //looping for as many rows as the recieved integer

// let str =''; //Initial value for our returning text

// for(j=0;j<n-i-1;j++){

// str +=" ";

// }

// console.log(str + hash + str) ;

//Will be modified in each loop (console.log each row)

// hash +="##";

// }

// }

## **11. Count Vowels**

**solution 1(Object)**

function vowels(str) {

let obj = {'a':1,'e':1,'i':1,'o':1,'u':1};

let counter = 0;

for(i=0;i<str.length;i++){

if(obj[str[i].toLowerCase()]){

//trying to reference the object, if it is falsy, it is not a property

counter++;

}

}

return counter;

}

**Solution2 (array.include)**

function vowels(str) {

let count = 0;

const checker = ['a', 'e', 'i', 'o', 'u'];

for (let char of str.toLowerCase()) {

if (checker.includes(char)) {

count++;

}

}

return count;

}

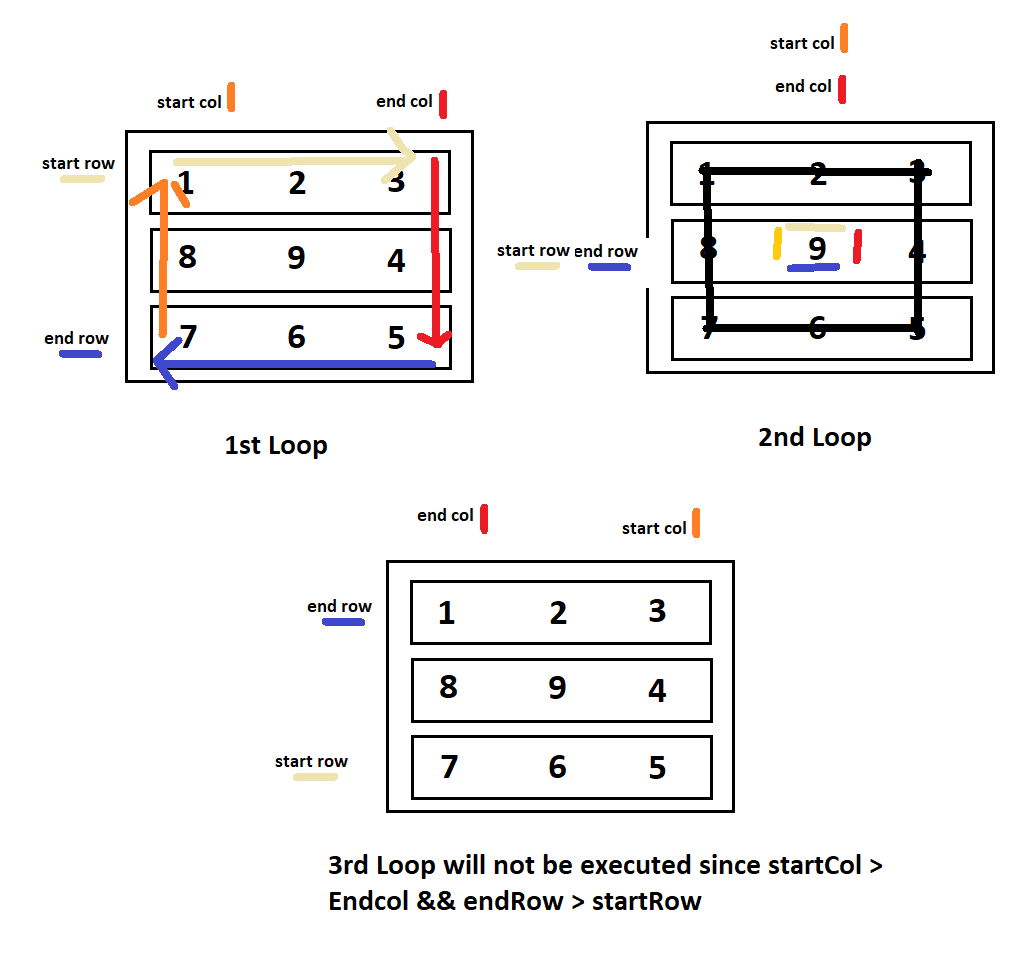
## **12 Spiral Matrix (Keep looping with descreasing conditionals)**

The key to this challange, is to combine 4 loops, keep track of the outer edge of the matrix with varibles.

1. loop:

- We loop through the first row, populate the data (+ incerement the startRow number, since the first row is full)

- We loop through the last column, populate the data (decrease the lastCOl number, since the last colis full)



// --- Directions // Write a function that accepts an integer N

// and returns a NxN spiral matrix.

// matrix(2)

// [[1, 2],

// [4, 3]]

// matrix(3)

// [[1, 2, 3],

// [8, 9, 4],

// [7, 6, 5]]

function matrix(n) {

let results = [];

let startRow = 0;

let endRow = n-1;

let startCol = 0;

let endCol = n-1;

let counter = 1;

for(i=0;i<n;i++){//To fill with n number of arrays

results.push([]);

}

while(startRow <= endRow || startCol <= endRow){

//Loop to fill the Top Row

for(i=startCol;i<=endCol;i++){

results[startRow][i] = counter;

counter++;

//Loop to fill the RightMost Column

}

startRow++;

for(i=startRow;i<=endRow;i++){

results[i][endCol] = counter;

counter++;

}

//Loop to fill the Bottom Row

--endCol;

for(i=endCol;i>=startCol;i--){

results[endRow][i]=counter;

counter++;

}

//Loop to fill the LeftMost Col

--endRow;

for(i=endRow;i>=startRow;i--){

results[i][startCol]=counter;

counter++;

}

startCol++;

}

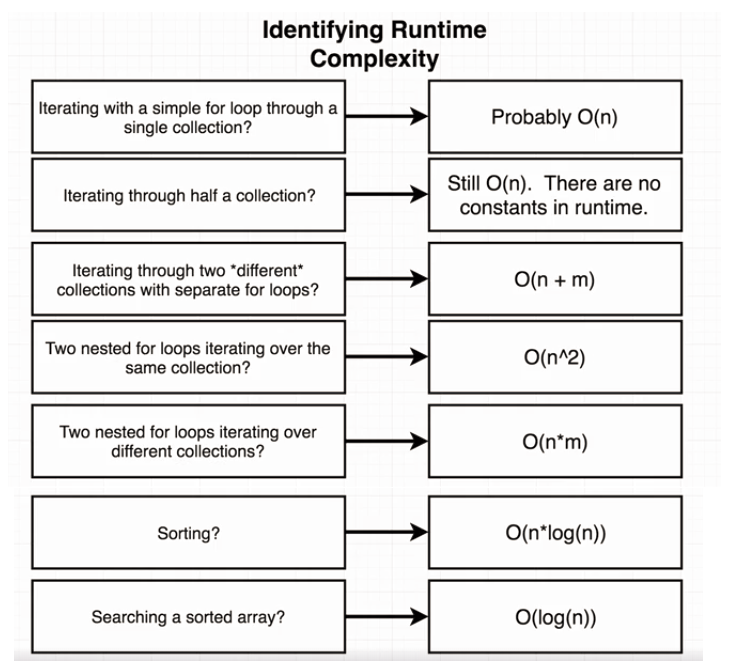
console.log(results);

return results;

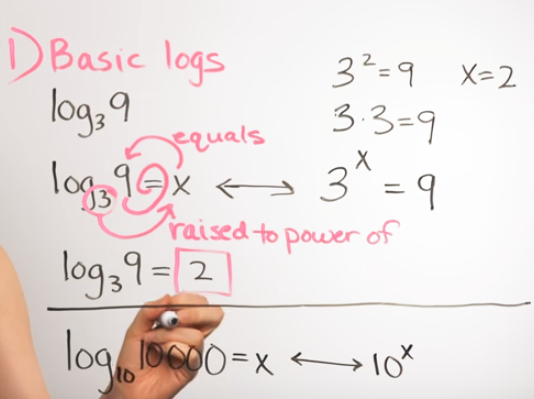
}

matrix(6);

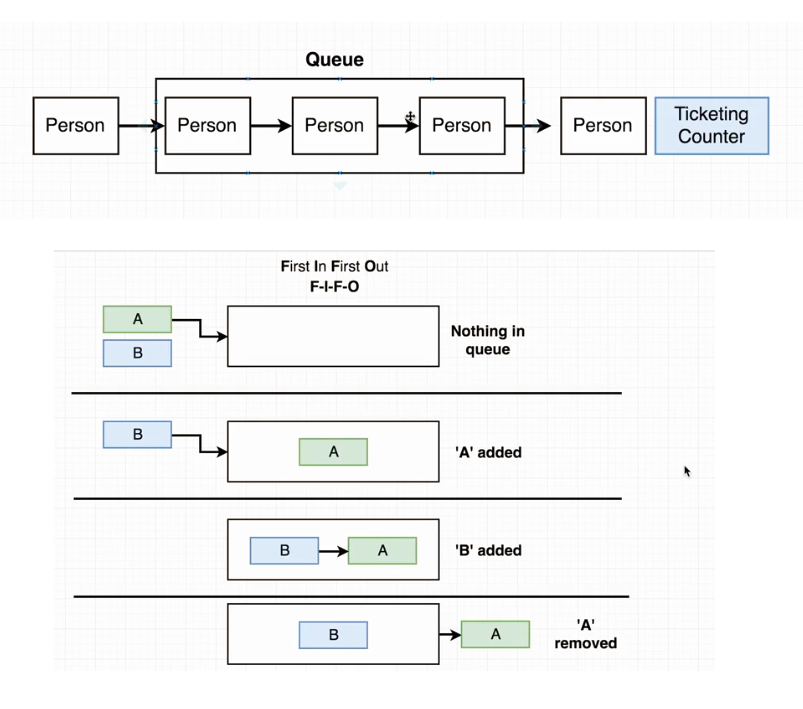
**13 RunTime Complexity**

****

**Log Explained**

****

**14 Queue**

****

class **Queue** {

constructor() {

this.data = [];

}

add(record) {

this.data.unshift(record);

}

remove() {

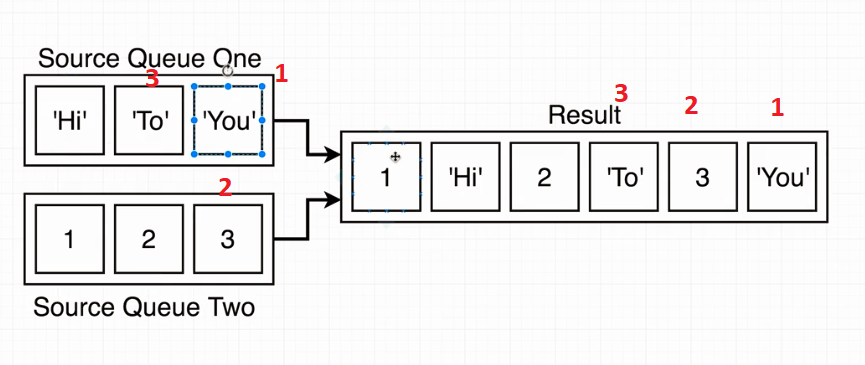
return this.data.pop();

}

}

module.exports = Queue;

**15 Weave (Combine two queue)**

****

const Queue = require('./queue');

const queueOne = new Queue();

queueOne.add(1);

queueOne.add(2);

const queueTwo = new Queue();

queueTwo.add('Hi');

queueTwo.add('There');

function weave(sourceOne, sourceTwo) {

let q = new Queue();

for(i=0;i<sourceOne.data.length;i++){

q.add(sourceOne.data[i]);

q.add(sourceTwo.data[i]);

}

return q;

}

**15 Stacks**

class Stack {

constructor() {

this.data = [];

}

push(record) {

this.data.unshift(record);

}

pop() {

return this.data.shift();

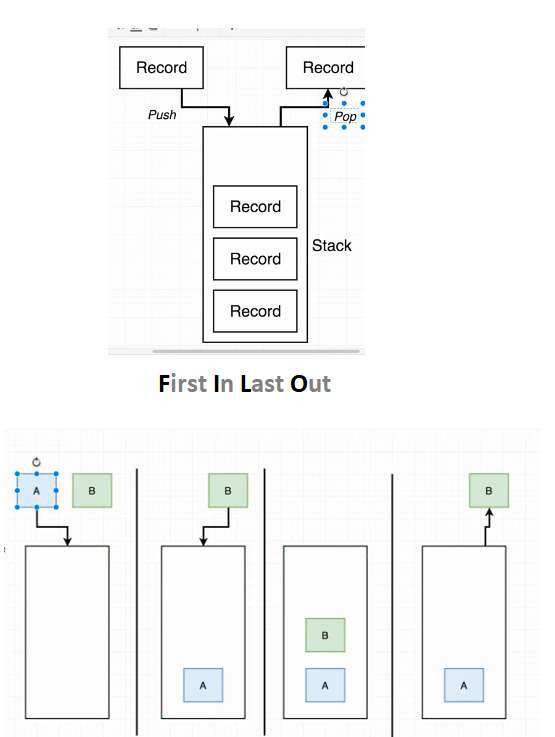
}

peek() {

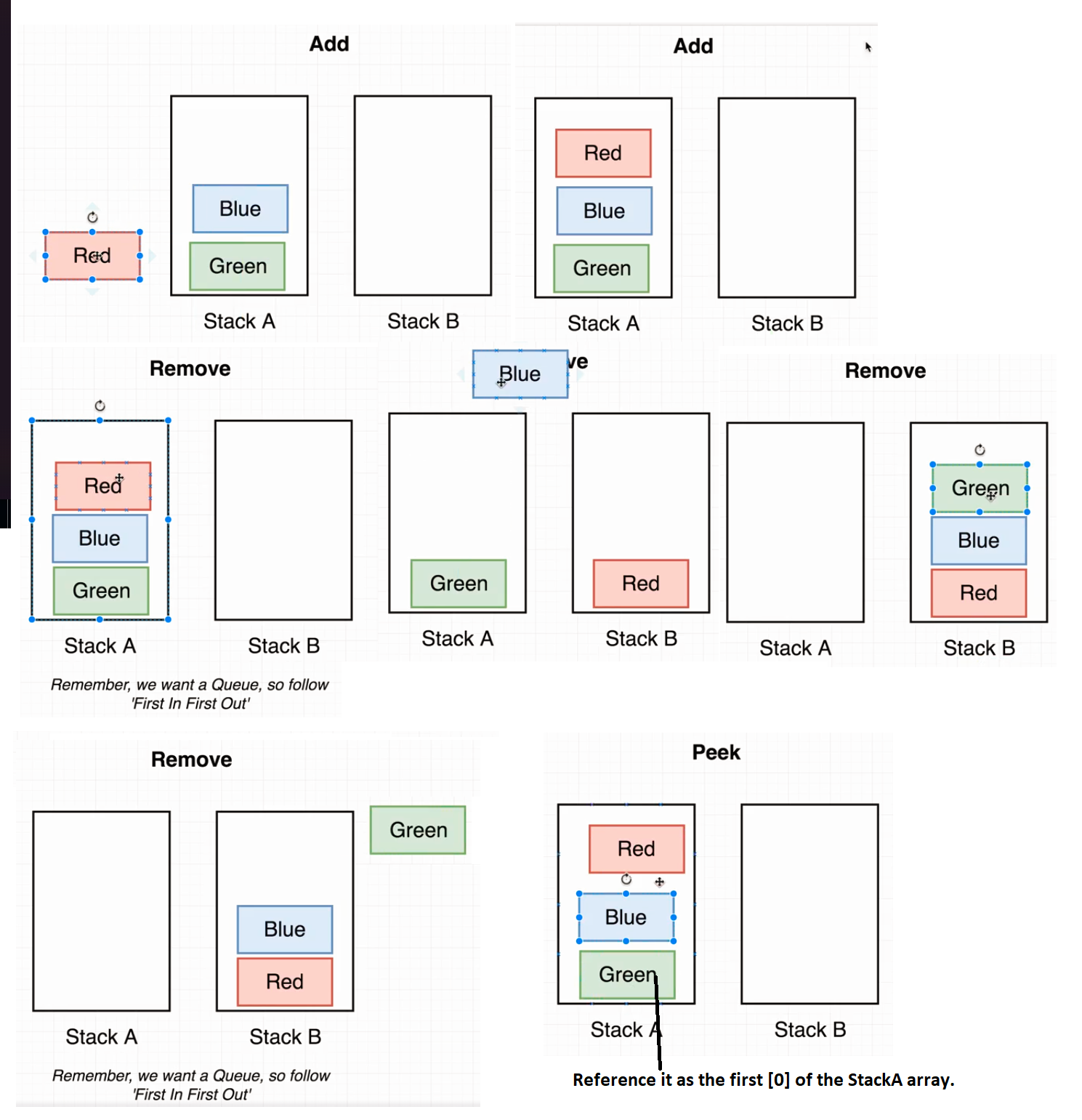
return this.data[0];

}

}



**16 Queue from stacks**

We havetwo stacks, which will be flipped, so we can check what was the first element.****

const Stack = require('./stack');

class Queue {

constructor() {

this.stackA = new Stack();

this.stackB = new Stack();

}

add(record) {

this.stackA.push(record) ;

}

remove() {

//Move all of the items from stackA to stackB, so the last item from stackA will be the first item for stackB,

let len = this.stackA.data.length;

while(len>0){

this.stackB.push(this.stackA.data[len-1]);

this.stackA.pop();

len = this.stackA.data.length;

}

//Remove the first item from stackB

return this.stackB.pop();

//Switch the two stacks, to always have the data in the stackA

let temp = this.stackA;

this.stackA = this.stackB;

this.stackB = temp;

}

peek(){

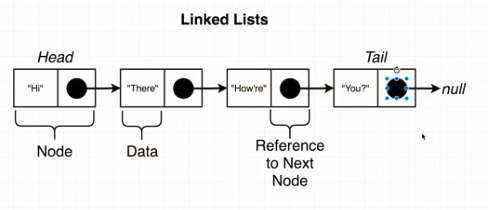
return this.stackA.peek();

}

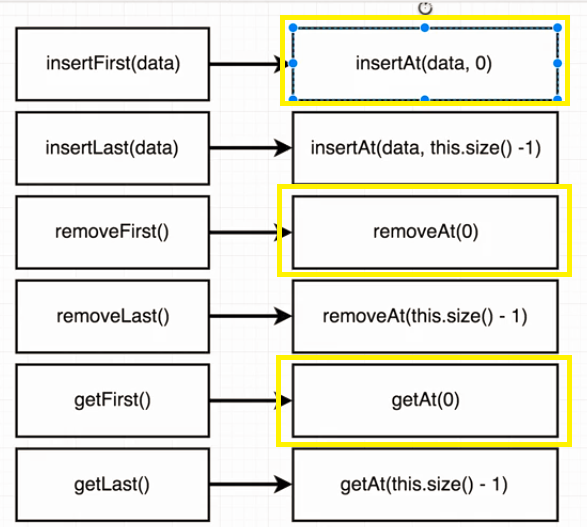
}

**17 Linked List**

**(with generator for..of loop)**



The follo wing functions are crutial in the linkedList : **insertAt**(), **removeAt**(), **getAt(**)



The created whole linked list, can be found in the mentione repository

//Creating instantly the iterator, when the class is created

\*[Symbol.iterator]() {

let node = this.head;

while (node) {

yield node;

node = node.next;

}

}

//Now this object, can be iterated through with the created iteration protocoll

const l = new List();

for (let node of l) {

node.data += 10;

}

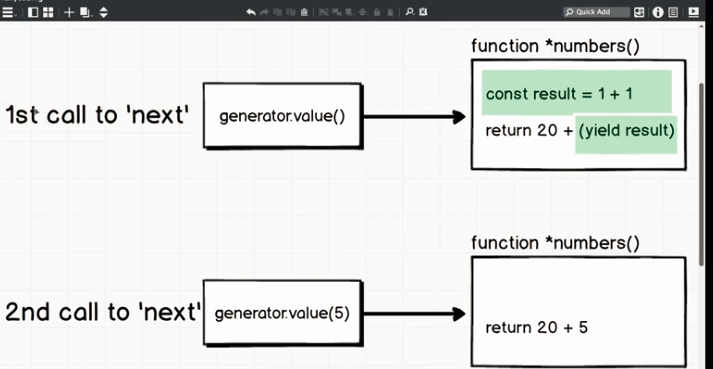
**17.1 Generators**

1**. Create a generator** with

function \* fname(){

}

* 1. The generator will pause, it will provide back the value of the returned generator object.



function \* numbers(){

let result = 1 + 1;

return 20 + (yield result);

}

const generator = numbers();

//The execution at the \*yield\* keyword is paused, and the yielded value has been returned

generator.next(); //value --> 2 -- {'value':2, "done":false}

generator.next(); // value --> 32 -- {'value':32, "done":true}

//if we pass a value to the .next() we will overwrite the yielded value

generator.next(10);

**2. Looping over the \*yielded values**

function \* list(){

yield 1;

yield 2;

yield 3;

yield 4;

}

const generator = list();

//The execution at the \*yield\* keyword is paused, and the yielded value has been returned

generator.next(); //value --> 1 -- {'value':1, "done":false}

generator.next(); // value -->2 -- {'value':2, "done":true}

//or we can loop over the yield steps

let numbers = [];

for (let value of generator){

numbers.push(value);

}

console.log(numbers);;

**3. Nest generator loopings**

function \* list(){

yield 1;

yield 2;

yield\* moreNumbers() ;

yield 5;

}

//The generator will call the inserted other generator function and will yield through of its content

function \* moreNumbers(){

yield 3;

yield 4;

}

const generator = list();

let numbers = [];

for (let value of generator){

numbers.push(value);

}

console.log(numbers); // [1,2,3,4,5]

**4. Nesting generators, inside class + looping**

//Creating a tree object, wich has a value and children array

class Tree {

constructor(value = null, children = []){

this.value = value;

this.children = children;

}

// THis is a property of the Tree class, which is a generator

//Returning the current object value first

//Then it will yield(return) and call the child's generator property

\*printValues(){

yield this.value;

for(let child of this.children){

yield\* child.printValues();

}

}

}

//Creating a tree

// 1

// 2 3

// 4

let tree = new Tree(1,[

new Tree(2,[new Tree(4)]),

new Tree(3),

]);

let arrValues = [];

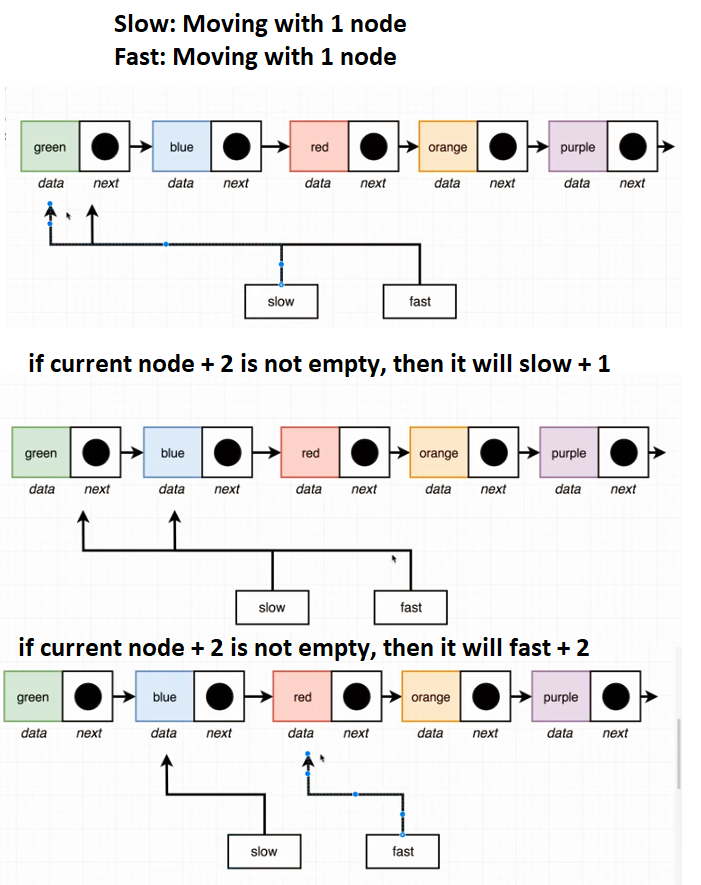
for(let child of tree.printValues()){

arrValues.push(child);

}

console.log(arrValues); // [1,2,4,3]

**18. Midpoint of LinkedList**

****

function midpoint(list) {

//Return null if the list is empty.

if(!list.head){return};

let slow = list.head; //It is for the midpoint

let fast = list.head; //For advance looping

while(fast.next && fast.next.next){ //Loop until there is item

slow = slow.next; //Storing the object references

fast = fast.next.next;

}

return slow; //returning the midNode

}

return slow; //returning the midNode

}

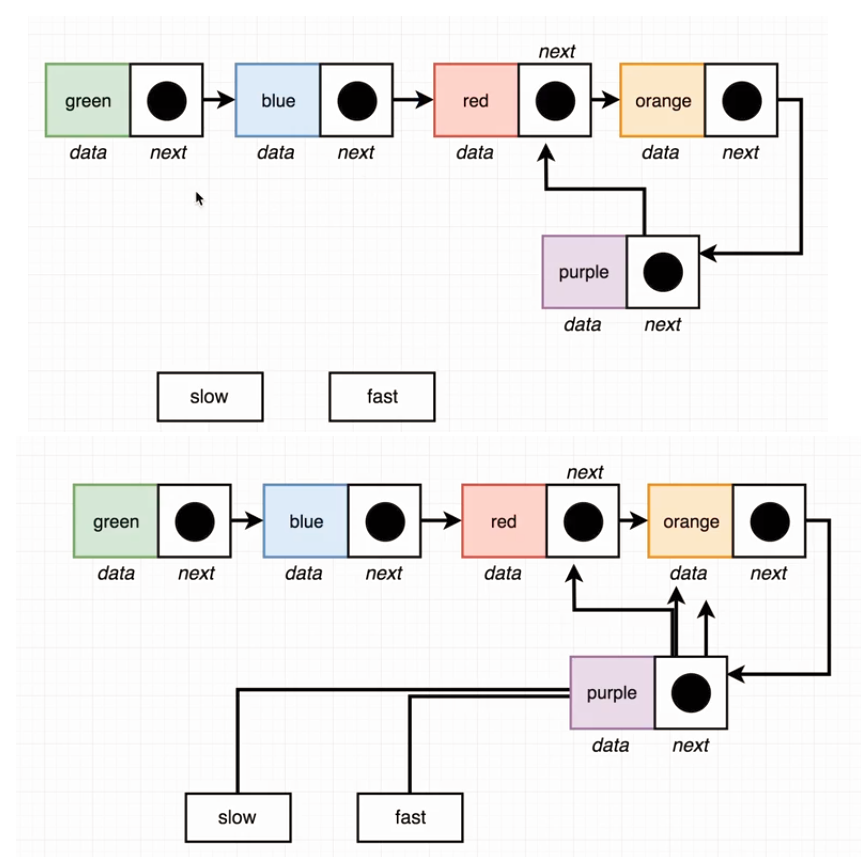
**19.Circular Linked List**

There could be some cases, when you accidentaly have a circular reference, thus it will kill all of our for loops.

Solution:

- Since, fast node is moving 1 node ahead all the time compared to the slow one.

- If it is circular, then they will meet at one node.

****

function circular(list) {

if(!list.head){return false}; //Return false if empty

let slow = list.getFirst();

let fast = list.getFirst();

while(fast.next && fast.next.next){

//loop until it finds a last el

slow = slow.next;

fast = fast.next.next;

if(slow === fast){

//or until they equal.

return true;

}

}

return false;

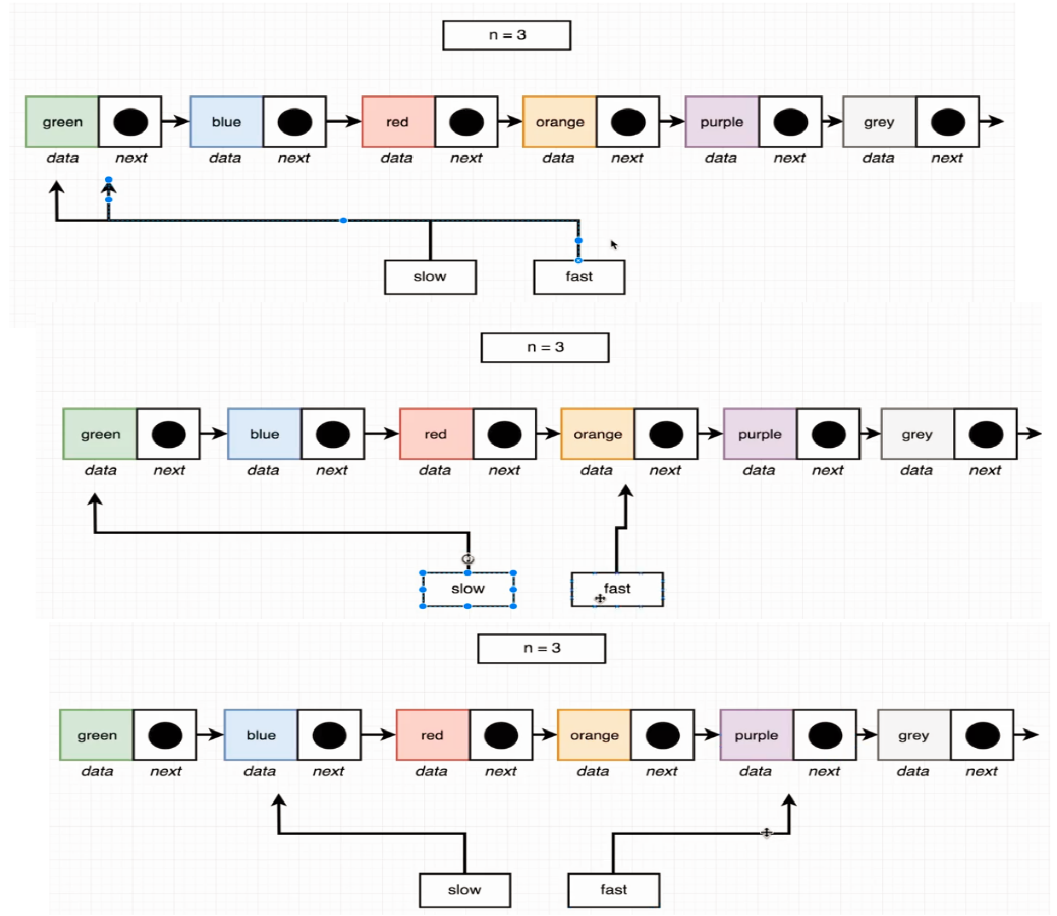
}

**20. N element from last node**

1. Phase we equal the slow, fast

2. Phase we move the fast n distance away

3. We are looping until the last element

****

function fromLast(list, n) {

if(!list.head){return}

let slow = list.head;

let fast = list.head;

for(let i = 0; i<n;i++){ //moving away n step

fast = fast.next;

}

while(fast.next){ //Since we are moving 1 step

slow = slow.next;

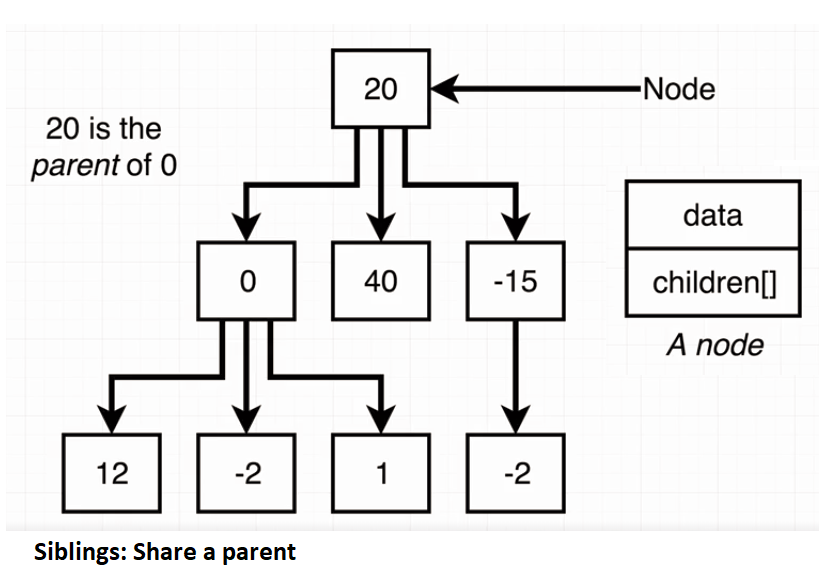
fast = fast.next;

}

return slow;

}

**21 Building Tree**

****

**21.1 Tree Nodes**

Basically, they are objects, which has data, and instead of next property they have an array of childrens.

class Node {

constructor(data,children=[]){

this.data = data;

this.children = children;

}

add(value){

this.children.push(new Node(value));

}

remove(data) {

this.children = this.children.filter(node => { //Filter will return filtered data as an array

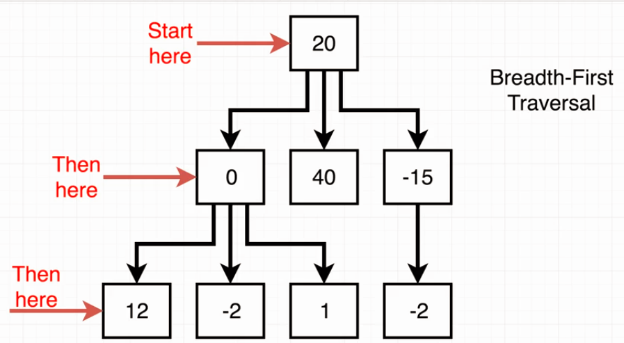
return node.data !== data;

});

}

}

**21.2 Breadth First Traversal**

****We are using a queue based execution, we put every node in an array with **FIFO** storing and by removing always the first one, we run the looping until no item is left in the array.

traverseBF(fn){

if(!this.root){return null;} //If empty tree

let nodeArr = [this.root];

while(nodeArr[0]){

fn(nodeArr[0]);

for(let i = 0; i<nodeArr[0].children.length;i++){

nodeArr.push(nodeArr[0].children[i]); //push to be last

}

nodeArr.shift(); //remove first

}

}

**//instead of for loop, ... separator has been used**

traverseBF(fn) {

const arr = [this.root];

while (arr.length) {

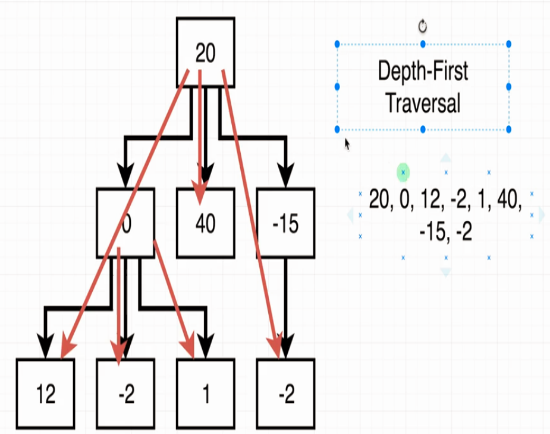
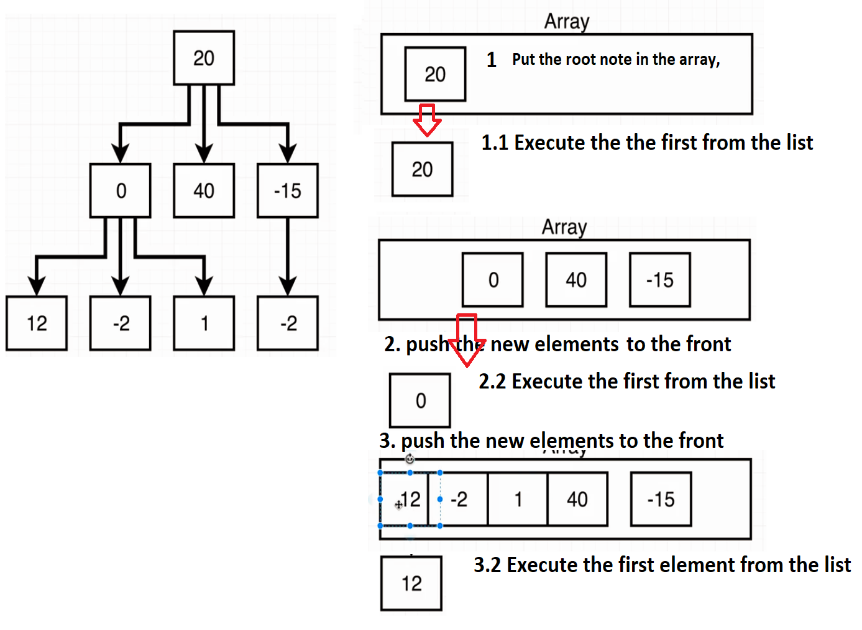
const node = arr.shift();

arr.push(**...**node.children);

fn(node);

}

}

**21.2 Depth First Traversal**

traverseDF(fn) {

const arr = [this.root];

while (arr.length) {

const node = arr.shift();

arr.unshift(...node.children);

fn(node);

}

}

**Recursion**

//We are executing the passed function on the root, or on the passed node first

//Then base = if it has children, do the looping, else, return

//If a lower call stack is returned, it will continue to the next loop

traverseDF(fn,node = this.root){

if(!this.root){return null;}

fn(node);

if(!node.children[0]){

return;

}

for(let i = 0; i<node.children.length;i++){

this.traverseDF(fn,node.children[i]);

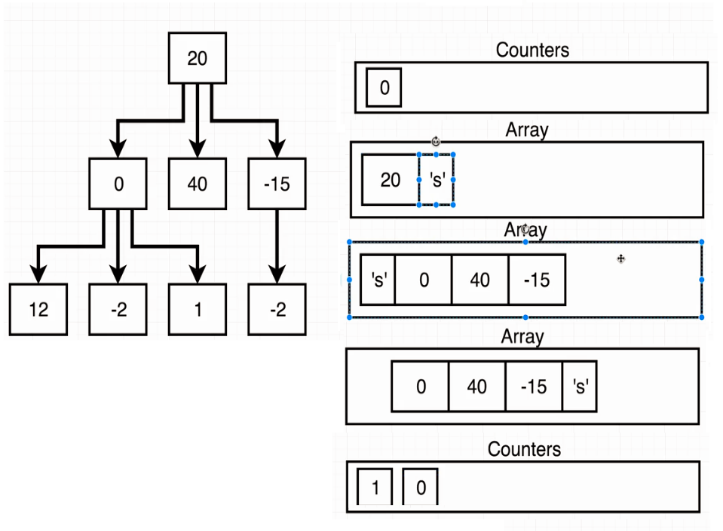
}

}

**21.3 Levelwidth**

So, this is almost the same as breadth First Traversal, but the way we keep track of the levels, is a pointer variable in the array, which is not a node, just a specific indicator.

Then if the array is not empty, we put it to the end of the array and continue looping.

****

// 0

// / | \

// 1 2 3

// | |

// 4 5

// Answer: [1, 3, 2]

function levelWidth(root) {

const arr = [root, 's'];

const counters = [0];

//Run until every item is remove expect 's'

while (arr.length > 1) {

const node = arr.shift();

//if indicator reached, check for other element and create a new array with 0

if (node === 's') {

counters.push(0);

arr.push('s');

//if not indicator, spread the elements in the array and increment the counter

} else {

arr.push(...node.children);

counters[counters.length - 1]++;

}

}

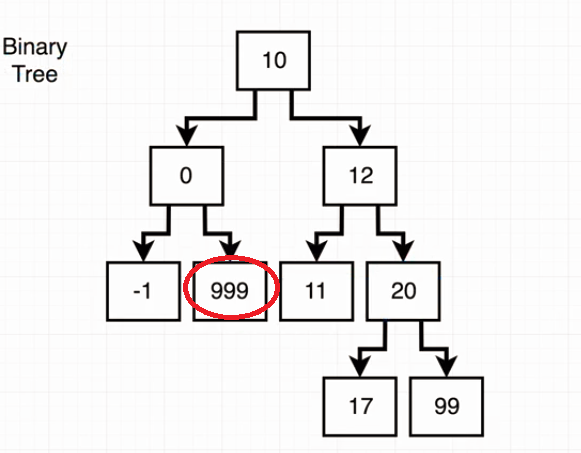
return counters;

}

**22. Binary Search Tree (bst)**

At Binary Search Trees, you can make sure that the left arguments are smaller than the rights, so binary search method can be used.

**Binary Tree**

****

While binary trees are same in the format, but you are not sure if the left side if the tree contains lesser values than the right side of the tree.

**22.1 Inserting(bst)**

//Recursion looping, check the right side and return null if the node is there

// or find the last node and insert to one of it's properties

insert(data){

if(data === this.data){return};

if(data> this.data){

if(!this.right){

this.right = new Node(data);

}else{

this.right.insert(data);

}

}else{

if(!this.left){

this.left = new Node(data);

}else{

this.left.insert(data);

}

}

}

**Course solution (Combined elseif)**

//1.So it is doing the comparison + check the existence of this.left property

//1.if yes, it will call itself

//2.if data is simply bigger and there are no left properties, it will create one

//3. check the same for the right one

insert(data) {

if (data < this.data && this.left) { //1

this.left.insert(data);

} else if (data < this.data) { //2

this.left = new Node(data);

} else if (data > this.data && this.right) { //3

this.right.insert(data);

} else if (data > this.data) {

this.right = new Node(data);

}

}

**22.2 Contains (Recursion return)(bst)**

**Main difference is that I am storing the returned value in recursion in an variable**

//Recursion loopin, where the returned answer has to be stored

//to obtain the node

contains(data){

let **answer** = null;

if(data === this.data) {

return this;

}else if(data > this.data && this.right){

**answer** = this.right.contains(data);

}else if(data> this.data){

return null;

}else if(data< this.data && this.left){

**answer** = this.left.contains(data);

}else if(data< this.data){

return null;

}

return **answer**;

}

**While here, instead of storing it, it is just returned.**

contains(data) {

if (this.data === data) {

return this;

}

if (this.data < data && this.right) {

**return this.right.contains(data);**

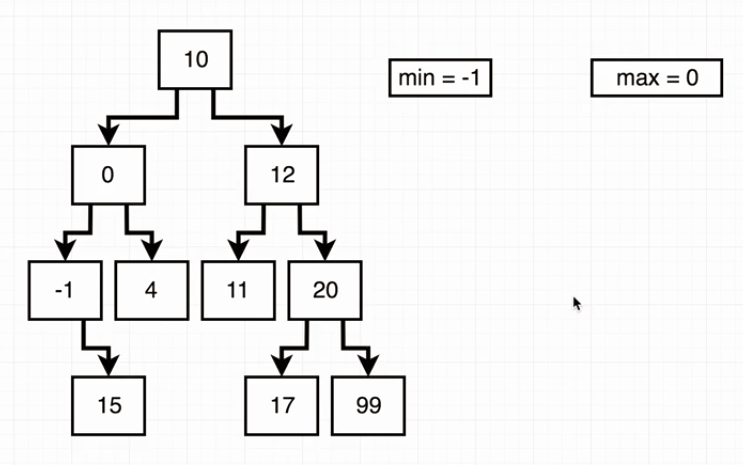
} else if (this.data > data && this.left) {

**return this.left.contains(data);**

}

return null;

}

**22.3 Validating Binary tree(validate)**

The key here, is to keep track of the possible minimum or maximum value during the recursion.

function validate(node, min = null, max = null) {

if(!node){return false};

//Will start looping on the left

//Check if the below node data is smaller, than the current node

// and checks if it is bigger then min, or min is null --> recursion with update min,max

//else false

if(node.left){

max = node.data;

if(node.left.data<max && min == null || node.left.data>min ){

return validate(node.left,min,max);

}else{

return false;

}

}

//Doing the same thing on the right

if(node.right){

min = node.data;

if(node.right.data>min && node.right.data<max || max == null){

return validate(node.right,min,max);

}else{

return false;

}

}

//Return true nor the left or right side is false or they are empty

return true;

}

**Notes:**

* Try to avoid nested, if statements. Instead of nesting, try to focus on the negative criteria, if the statement can survive all of the statements then it will return true.
* It is like a funnel, keep the possibilities less and less as we process the code.

//1.Test the first cases, if it is bigger or smaller then the criterias

//2.If there is a node function + it is returning false !false = true

function validate(node, min = null, max = null) {

/\*\*1\*\*/

if (max !== null && node.data > max) {

return false;

}

if (min !== null && node.data < min) {

return false;

}

// !validate(node.left, min, node.data)-> will return true or false

// !false = true

// So if there is a node left, and the recursed function results in false

// there is a mistake in the binary tree

/\*\*2\*\*/

if (node.left && !validate(node.left, min, node.data)) {

return false;

}

if (node.right && !validate(node.right, node.data, max)) {

return false;

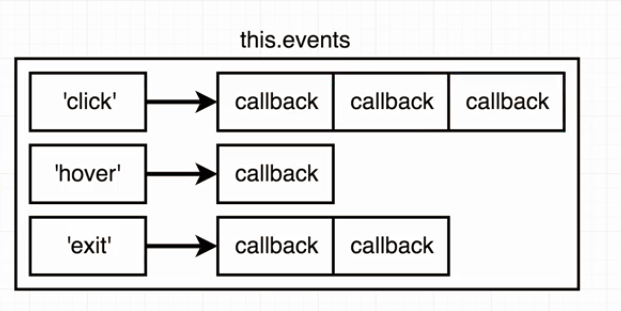
}

//if no error has been found, than it will return true

return true;

}

**23. Events**

**On:** Register a custom event, in the events object

(Note that multiple call back functions can be registered under the same name)

**Trigger:**

Will directly call the registered event name

**Off:** Will clear the registered event name from the Events Object

class Events {

constructor(){

this.events = {};

}

// Register an event handler

on(eventName, callback) {

if(!this.events[eventName]){

this.events[eventName] = [callback];

}else{

this.events[eventName].push(callback);

}

}

// Trigger all callbacks associated

// with a given eventName

trigger(eventName) {

let event = this.events[eventName];

if (event == null){return};

for(let i=0;i<event.length;i++){

event[i]();

}

}

// Remove all event handlers associated

// with the given eventName  
 off(eventName) {

//Delete is 100 times slover, the assigning null.

// delete this.events[eventName];

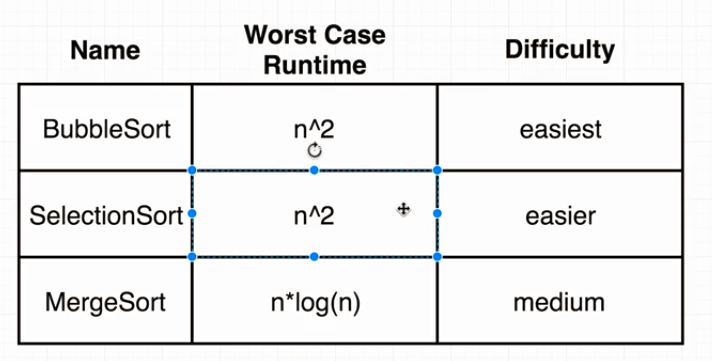
this.events[eventName] = null;

}

}

**24. Sorting**

<http://dissertation.dayanpetrow.eu/>



**24.1 BubbleSort**

Comparing each value to each other into a nested loop

function bubbleSort(arr) {

let temp = null;

for(let i = 0; i<arr.length;i++){

for(let y = 0; y<=arr.length-i;y++){

if(arr[y]>arr[y+1]){

temp = arr[y+1];

arr[y+1] = arr[y];

arr[y] = temp;

}

}

}

return arr;

}

**24.2 Selection Sort**

Selecting the first element in the array, setting it as a minimum.

During the loop we are trying to find a smaller one and in the end we switch to the lowest one to the first position.

function selectionSort(arr) {

for (let i = 0; i < arr.length; i++) {

let indexOfMin = i;

for (let j = i+1; j <arr.length; j++) {

if (arr[j] < arr[indexOfMin]) {

indexOfMin = j;

}

}

if (indexOfMin !== i) {

let lesser = arr[indexOfMin];

arr[indexOfMin] = arr[i];

arr[i] = lesser;

}

}

return arr;

}

**24.3 MergeSort**

<https://www.youtube.com/watch?v=JSceec-wEyw>

From the video, you can visually see recursive sorting algorithm.

//Slicing up the recieved aarray in half and return if there

//is only 1 element in the arr

function mergeSort(arr) {

if (arr.length === 1) {

return arr;

}

const center = Math.floor(arr.length / 2);

const left = arr.slice(0, center);

const right = arr.slice(center);

return merge(mergeSort(left), mergeSort(right));

}

//From two array, create a sorted one

function merge(left, right) {

const results = [];

while (left.length && right.length) {

if (left[0] < right[0]) {

results.push(left.shift());

} else {

results.push(right.shift());

}

}

return [...results, ...left, ...right];

}