Big O Notation

Tuesday, December 22, 2020 11:33 AM

- Understanding Big O Notation
  - Example of adding up numbers up to n can be approached with two solutions

1) Using for loop to add up

```
function addUpTo(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += i;
  }
  return total;
}

var t1 = performance.now();
addUpTo(1000000000);
var t2 = performance.now();
console.log(`Time Elapsed: ${(t2 - t1) / 1000} seconds.`)
```

```
undefined                                                    Add1_Timing:1
Time Elapsed: 1.2600000000093132 seconds.                    Add1_Timing:12
undefined                                                    Add1_Timing:1
Time Elapsed: 1.2436999999918044 seconds.                    Add1_Timing:12
undefined                                                    Add1_Timing:1
Time Elapsed: 1.2553000000189058 seconds.                    Add1_Timing:12
undefined                                                    Add1_Timing:1
```

2) Using (n*(n+1)/2) - Efficient and takes lesser time in giving out a solution

```
function addUpTo(n) {
  return n * (n + 1) / 2;
}

var time1 = performance.now();
addUpTo(1000000000);
var time2 = performance.now();
console.log(`Time Elapsed: ${(time2 - time1) / 1000} seconds.`)
```

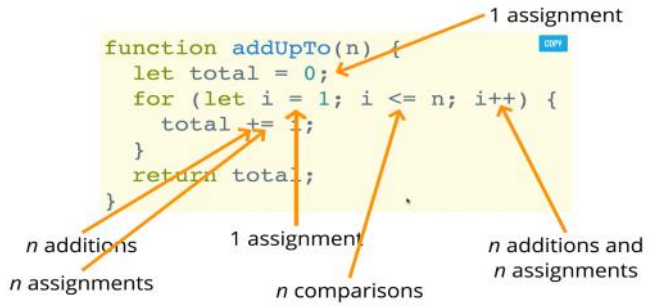
```
undefined                                                    Add2_Timing:1
Time Elapsed: 0.00010000000474974513 seconds.                Add2_Timing:8
undefined                                                    Add2_Timing:1
Time Elapsed: 0.00010000000474974513 seconds.                Add2_Timing:8
undefined                                                    Add2_Timing:1
Time Elapsed: 0.00010000000474974513 seconds.                Add2_Timing:8
undefined                                                    Add2_Timing:1
```

The Problem with Time

- Different machines will record different times
- The *same* machine will record different times!
- For fast algorithms, speed measurements may not be precise enough?

- Approach 1:

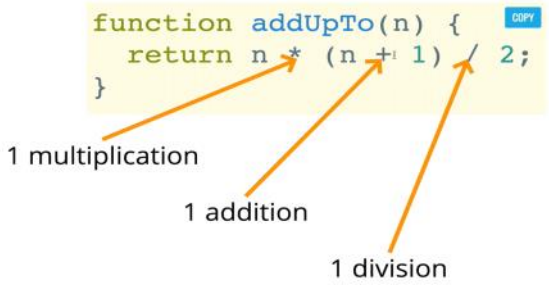
Counting Operations



- But using this approach of for loop, it goes through so many operations and as n grows the number of operations also grows
- The time also grows in proportion as n grows larger

- Approach 2:

Counting Operations



- Regardless of the size of n, only 3 operations will take place as shown above
- Time is constant irrespective of the input n, which makes it significantly better than the other approach.

Introducing....Big O

Big O Notation is a way to formalize fuzzy counting

It allows us to talk formally about how the runtime of an algorithm grows as the inputs grow

Big O Definition

We say that an algorithm is **O(f(n))** if the number of simple operations the computer has to do is eventually less than a constant times **f(n)**, as **n** increases

Big O Definition

We say that an algorithm is **O(f(n))** if the number of simple operations the computer has to do is eventually less than a constant times **f(n)**, as **n** increases

- f(n) could be linear (f(n) = n)
- f(n) could be quadratic (f(n) = n^2)
- f(n) could be constant (f(n) = 1)
- f(n) could be something entirely different!

Example

```
function addUpTo(n) {
  return n * (n + 1) / 2;
}
```

Always 3 operations

O(1)

```
function addUpTo(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += i;
  }
  return total;
}
```

Number of operations is (eventually) bounded by a multiple of *n* (say, 10*n*)

O(*n*)

Another Example

```
function countUpAndDown(n) {
  console.log("Going up!");
  for (let i = 0; i < n; i++) {        // O(n)
    console.log(i);
  }
  console.log("At the top!\nGoing down...");
  for (let j = n - 1; j >= 0; j--) {   // O(n)
    console.log(j);
  }
  console.log("Back down. Bye!");
}
```

Number of operations is (eventually) bounded by a multiple of *n* (say, 10*n*)

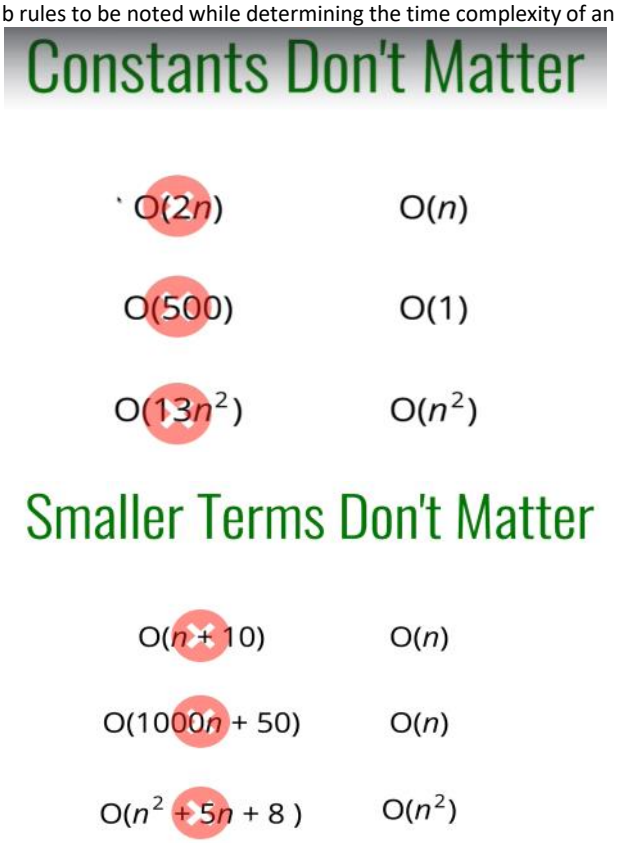
O(*n*)

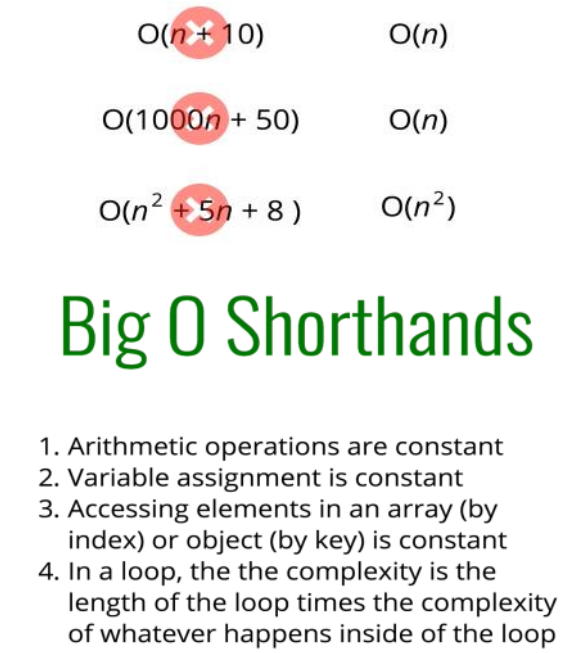
OMG MOAR EXAMPLEZ

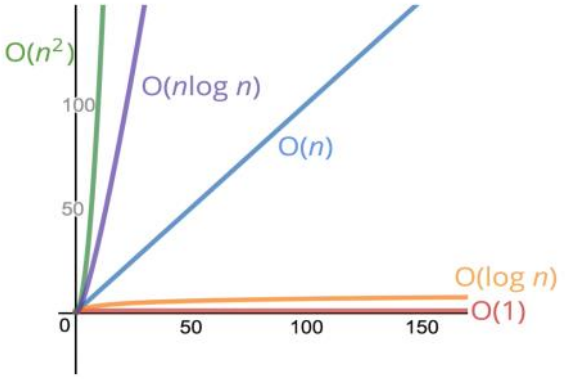
```
function printAllPairs(n) {
  for (var i = 0; i < n; i++) {          // O(n)
    for (var j = 0; j < n; j++) {        // O(n)
      console.log(i, j);
    }
  }
}
```

O(n) operation inside of an O(n) operation.

O(n^2)

- Thumb rules to be noted while determining the time complexity of an algorithm
  - 

  - 

- 

- Examples

A Couple More Examples

```
function logAtLeast5(n) {
  for (var i = 1; i <= Math.max(5, n); i++) {
    console.log(i);
  }
}
```

O(*n*)

```
function logAtMost5(n) {
  for (var i = 1; i <= Math.min(5, n); i++) {
    console.log(i);
  }
}
```

O(1)

- Space Complexity

Space Complexity

So far, we've been focusing on **time complexity**:
how can we analyze the *runtime* of an algorithm as
the size of the inputs increases?

We can also use big O notation to analyze **space complexity**:
how much additional memory do we need to allocate in order
to run the code in our algorithm?

What about the inputs?

Sometimes you'll hear the term **auxiliary space complexity** to refer to space required by the algorithm, not including space taken up by the inputs.

Unless otherwise noted, when we talk about space complexity, technically we'll be talking about auxiliary space complexity.

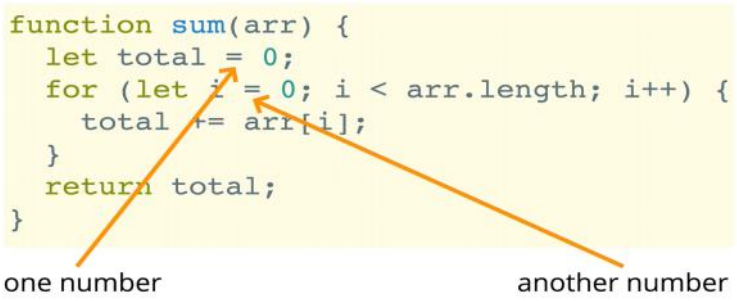
Space Complexity in JS

Rules of Thumb

- Most primitives (booleans, numbers, `undefined`, `null`) are constant space
- Strings require O(*n*) space (where *n* is the string length)
- Reference types are generally O(*n*), where n is the length (for arrays) or the number of keys (for objects)

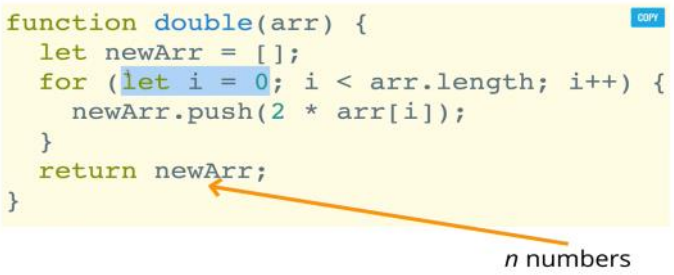
An Example

An Example

- 

O(1) space!

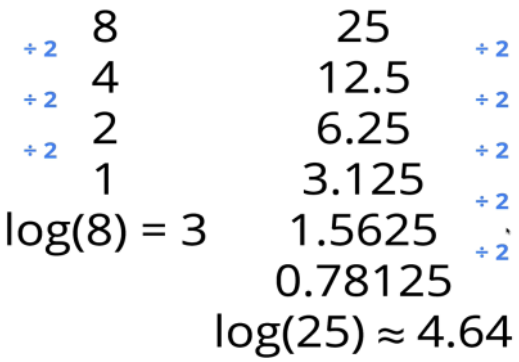
Another Example

- 

O(*n*) space!

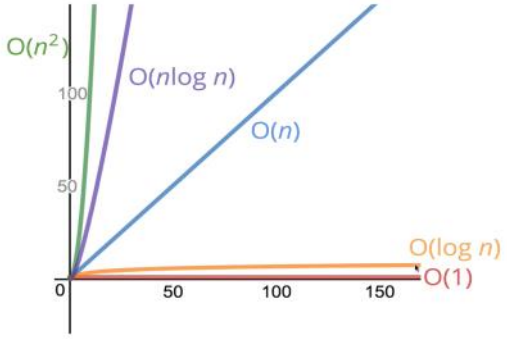
- Logarithms
  - Logarithm is the inverse of exponentiation
  - $\log_2(8) = 3 \longrightarrow 2^3 = 8$
  - $\log_2(value) = exponent \longrightarrow 2^{exponent} = value$

Logarithm Examples

- 

Logarithm Complexity

Logarithmic time complexity is great!



Certain searching algorithms have logarithmic time complexity.

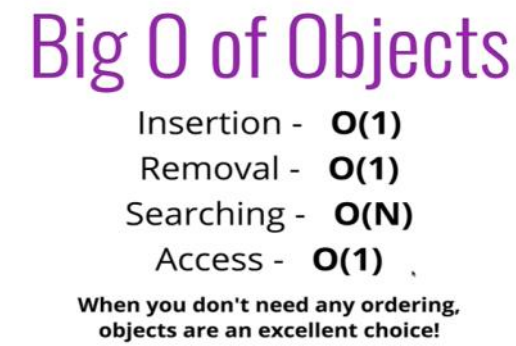
Efficient sorting algorithms involve logarithms.

Recursion sometimes involves logarithmic space complexity.

Recap

- To analyze the performance of an algorithm, we use Big O Notation
- Big O Notation can give us a high level understanding of the time or space complexity of an algorithm
- Big O Notation doesn't care about precision, only about general trends (linear? quadratic? constant?)
- The time or space complexity (as measured by Big O) depends only on the algorithm, not the hardware used to run the algorithm
- Big O Notation is everywhere, so get lots of practice!

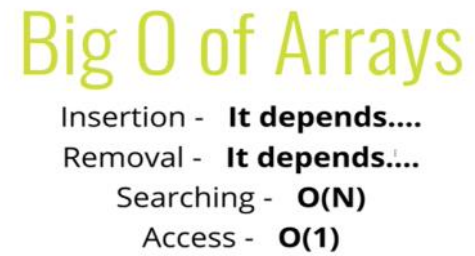
- Big O for Objects

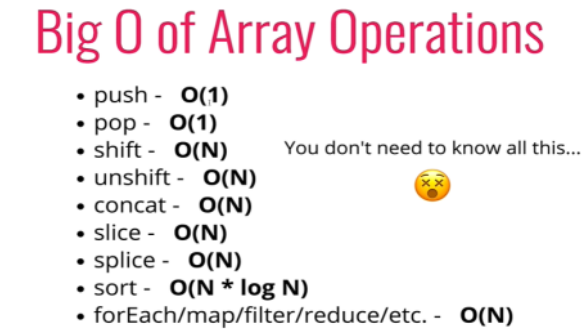
- 

- 

- Big O for Arrays

- 

- 

  - Insertion at beginning / Removing from beginning of array - O(n)
  - Push and pop are always faster than shift and unshift
  - Push and pop are always done at the end of the array which makes it O(1)
- 

  - indexOf() - **O(N) -** as it loops through whole array to find the index

Problem Solving

Tuesday, December 22, 2020 3:02 PM

- **Problem Solving Approach**
  - 

- **Problem: Character count in a string**
  - Solution 1:

```
function charCount(str) {
    var obj = {};
    for (var i = 0; i < str.length; i++) {
        var char = str[i].toLowerCase();
        if (/[a-z0-9]/.test(char)) {
            if (obj[char] > 0) {
                obj[char]++;
            } else {
                obj[char] = 1;
            };
        }
    }
    return obj;
}
```

  - Solution 2:

```
function charCount(str) {
    var obj = {};
    for (var char of str) {
        char = char.toLowerCase();
        if (/[a-z0-9]/.test(char)) {
            obj[char] = ++obj[char] || 1;
        }
    }
    return obj;
}
```

  - Refactor code as regex might take more time than expected to complete the operation

Testing in Chrome 67.0.3396 / Mac OS X 10.13.4

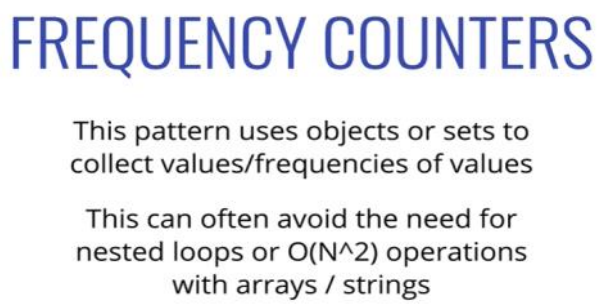
Test		Ops/sec
charCode	`function isAlphaNumeric(str) { var code; for (var i = 0, len = str.length; i < len; i++) { code = str.charCodeAt(i); if (!(code > 47 && code < 58) && // numeric (0-9) !(code > 64 && code < 91) && // upper alpha (A-Z) !(code > 96 && code < 123)) { // lower alpha (a-z) return false; } } return true; } isAlphaNumeric(text);`	60,506,460 ±1.37% fastest
regexp	`function isAlphaNumeric(str) { return /^[a-zA-Z0-9]+$/.test(str); } isAlphaNumeric(text);`	28,026,909 ±4.01% 55% slower

  - Refactored - Solution 3:

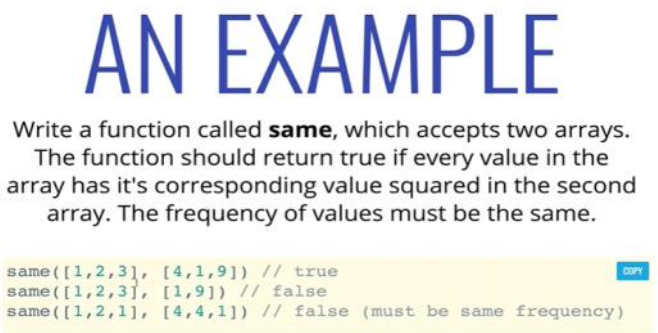
```
function charCount(str) {
    var obj = {};
    for (var char of str) {
        if (isAlphaNumeric(char)) {
            char = char.toLowerCase();
            obj[char] = ++obj[char] || 1;
        }
    }
    return obj;
}

function isAlphaNumeric(char){
    var code = char.charCodeAt(0);
    if (!(code > 47 && code < 58) && // numeric (0-9)
        !(code > 64 && code < 91) && // upper alpha (A-Z)
        !(code > 96 && code < 123)) { // lower alpha (a-z)
      return false;
    }
    return true;
}

charCodeAt(0)
```

- **Problem Solving Patterns**
  - Below are some common patterns which can help us in problem solving

    - **Frequency Counter Pattern**



      - When we have multiple inputs which needs to be compared, we can use this pattern



      - **Naïve-approach - O(n^2)**

```
function checkIfSquaredArray(a, b) {
    if (a.length !== b.length) return false;
    for (let i = 0; i < a.length; i++) {
        let index = b.indexOf((a[i]*a[i]));
        if (index === -1) return false;
        b.splice(index, 1);
    }
    return true;
}

checkArray([1,2,1], [4,1,1]);
```

- **Refactored - O(n)**

```
function checkIfSquaredArray(a, b) {
    if (a.length !== b.length) return false;
    let aCounter = {};
    let bCounter = {};
    for (let val of a) {
        let doubleVal = val*val;
        aCounter[doubleVal] = (aCounter[doubleVal] || 0) + 1;
    }
    for (let val of b) {
        bCounter[val] = (bCounter[val] || 0) + 1;
    }
    for (let key in aCounter) {
        if (aCounter[key] !== bCounter[key]) return false;
    }
    return true;
}

checkIfSquaredArray([1,2,1], [4,1,1]);
```

ANAGRAMS

Given two strings, write a function to determine if the second string is an anagram of the first. An anagram is a word, phrase, or name formed by rearranging the letters of another, such as *cinema*, formed from *iceman*.

```
validAnagram('', '') // true
validAnagram('aaz', 'zza') // false
validAnagram('anagram', 'nagaram') // true
validAnagram("rat","car") // false) // false
validAnagram('awesome', 'awesom') // false
validAnagram('qwerty', 'qeywrt') // true
validAnagram('texttwisttime', 'timetwisttext') // true
```

```
function isValidAnagram(s1, s2) {
    if (s1.length !== s2.length) return false;
    if (s1 === s2) return true;
    let s1Counter = {};
    let s2Counter = {};
    for (let val of s1) {
        s1Counter[val] = (s1Counter[val] || 0) + 1;
    }
    for (let val of s2) {
        s2Counter[val] = (s2Counter[val] || 0) + 1;
    }
    for (let key in s1Counter) {
        if (s1Counter[key] !== s2Counter[key]) return false;
    }
    return true;
}

// isValidAnagram('rat', 'car'); // false
// isValidAnagram('anagram', 'nagaram'); // true
isValidAnagram('iceman', 'cinema'); // true
```

```
function isValidAnagram(s1, s2) {
    if (s1.length !== s2.length) return false;
    if (s1 === s2) return true;
    let lookup = {};
    for (let val of s1) {
        lookup[val] = (lookup[val] || 0) + 1;
    }
    for (let val of s2) {
        if (!lookup[val]) return false;
        lookup[val] -= 1;
    }

    return true;
}

// isValidAnagram('rat', 'car'); // false
// isValidAnagram('anagram', 'nagaram'); // true
isValidAnagram('iceman', 'cinema'); // true
```

- **Multiple Pointers**

- Creating **pointers** or values that correspond to an index or position and move towards the beginning, end or middle based on a certain condition

  Very efficient for solving problems with minimal space complexity as well

- When we have a single input with values and we have to operate on the array/string linearly we can use this pattern.

- Write a function called **sumZero** which accepts a **sorted** array of integers. The function should find the **first** pair where the sum is 0. Return an array that includes both values that sum to zero or undefined if a pair does not exist

```
sumZero([-3,-2,-1,0,1,2,3]) // [-3,3]
sumZero([-2,0,1,3]) // undefined
sumZero([1,2,3]) // undefined
```

- **Naïve approach - O(n^2)**

```
function sumZero(arr){
    for(let i = 0; i < arr.length; i++){
        for(let j = i+1; j < arr.length; j++){
            if(arr[i] + arr[j] === 0){
                return [arr[i], arr[j]];
            }
        }
    }
}

sumZero([-4,-3,-2,-1,0,1,2,5])
```

- **O(n) approach**

```
function sumZeroPair(a) {
    let left = 0;
    let right = a.length - 1;
    while (left < right) {
        let sum = a[left] + a[right];
        if (sum === 0) return [a[left], a[right]];
        if (sum > 0) {
            right--;
            continue;
        }
        left++;
    }
    return [];
}


// sumZeroPair([-3, -2, -1, 0, 1, 2, 3, 10]); // [-3, 3]
// sumZeroPair([-2, 0, 1, 3]); // []
sumZeroPair([1, 2, 3]); // []
```

countUniqueValues

Implement a function called **countUniqueValues**, which accepts a sorted array, and counts the unique values in the array. There can be negative numbers in the array, but it will always be sorted.

```
countUniqueValues([1,1,1,1,1,2]) // 2
countUniqueValues([1,2,3,4,4,4,7,7,12,12,13]) // 7
countUniqueValues([]) // 0
countUniqueValues([-2,-1,-1,0,1]) // 4
```

```
function countUniqueValues(a) {
    if (a.length === 0) return 0;
    let i = 0;
    let j = i+1;
    let count = 0;
    while (i < a.length-1) {
        if (a[i] === a[j]) {
            j++;
            continue;
        }
        i = j;
        count++;
    }
    return count+1;
}

// countUniqueValues([1,1,1,1,1,1,2]); // 2
countUniqueValues([1,2,3,4,4,4,7,7,12,12,13]); // 7
```

- **Sliding Window pattern**

SLIDING WINDOW

This pattern involves creating a **window** which can either be an array or number from one position to another

Depending on a certain condition, the window either increases or closes (and a new window is created)

Very useful for keeping track of a subset of data in an array/string etc.

- Used when we have to deal with subsets of strings/arrays
- Ex. Longest unique character length in a string
- Ex. Maximum sub array sum in an array

An Example

Write a function called maxSubarraySum which accepts an array of integers and a number called **n**. The function should calculate the maximum sum of **n** consecutive elements in the array.

```
maxSubarraySum([1,2,5,2,8,1,5],2) // 10
maxSubarraySum([1,2,5,2,8,1,5],4) // 17
maxSubarraySum([4,2,1,6],1) // 6
maxSubarraySum([4,2,1,6,2],4) // 13
maxSubarraySum([],4) // null
```

- **Naïve approach - O(n^2)**

```
function maxSubarraySum(arr, num) {
  if ( num > arr.length){
    return null;
  }
  var max = -Infinity;
  for (let i = 0; i < arr.length - num + 1; i ++){
    temp = 0;
    for (let j = 0; j < num; j++){
      temp += arr[i + j];
    }
    if (temp > max) {
      max = temp;
    }
  }
  return max;
}
```

- **Optimal Solution - O(n)**

```
function maxSubarraySum(arr, num){
  let maxSum = 0;
  let tempSum = 0;
  if (arr.length < num) return null;
  for (let i = 0; i < num; i++) {
    maxSum += arr[i];
  }
  tempSum = maxSum;
  for (let i = num; i < arr.length; i++) {
    tempSum = tempSum - arr[i - num] + arr[i];
    maxSum = Math.max(maxSum, tempSum);
  }
  return maxSum;
}

maxSubarraySum([2,6,9,2,1,8,5,6,3],3)
```

- **My Solution**

```
function maxSubArraySum(a, num) {
    if (num > a.length) return null;
    let tempSum = 0;
    let maxSum = 0;
    for (let i = 0; i < a.length; i++) {
        if (i < num) {
            tempSum += a[i];
            if (i === num-1) maxSum = tempSum;
            continue;
        }
        tempSum += a[i] - a[i-num];
        maxSum = maxSum < tempSum ? tempSum : maxSum;
    }
    return maxSum;
}


maxSubArraySum([2, 6, 9, 2, 1, 8, 5, 6, 3], 3);
```

- Divide and Conquer

Divide and Conquer

- This pattern involves dividing a data set into smaller chunks and then repeating a process with a subset of data.

  This pattern can tremendously **decrease time complexity**

- Ex. Binary Search

An Example

- Given a **sorted** array of integers, write a function called search, that accepts a value and returns the index where the value passed to the function is located. If the value is not found, return -1

```
search([1,2,3,4,5,6],4) // 3
search([1,2,3,4,5,6],6) // 5
search([1,2,3,4,5,6],11) // -1
```

A naive solution

-

```
function search(arr, val){
    for(let i =!0; i < arr.length; i++){
        if(arr[i] === val){
            return i;
        }
    }
    return -1;
}
```

Linear Search

Time Complexity O(N)

- But if we use Divide and Conquer pattern, we divide the sorted array into two, check if the end number is lesser or greater than the number to be found and then search until the number is found. We repeat this process of dividing until the number is found out.

- Problems

Write a function called **sameFrequency.** Given two positive integers, find out if the two numbers have the same frequency of digits.

Your solution MUST have the following complexities:

Time: O(N)

- Sample Input:

```
sameFrequency(182,281) // true
sameFrequency(34,14) // false
sameFrequency(3589578, 5879385) // true
sameFrequency(22,222) // false
```

```
function sameFrequency(n1, n2) {
    n1 = n1.toString();
    n2 = n2.toString();
    if (n1.length !== n2.length) return false;
    let lookup = {};
    for (let i = 0; i < n1.length; i++) {
        lookup[n1[i]] = (lookup[n1[i]] || 0) + 1;
    }
    for (let j = 0; j < n2.length; j++) {
        if (!lookup[n2[j]]) return false;
        lookup[n2[j]]--;
    }
    return true;
}
sameFrequency(18, 81);
```

Implement a function called, **areThereDuplicates** which accepts a **variable number of arguments,** and checks whether there are any duplicates among the arguments passed in. You can solve this using the frequency counter pattern OR the multiple pointers pattern.

- Examples:

```
areThereDuplicates(1, 2, 3) // false
areThereDuplicates(1, 2, 2) // true
areThereDuplicates('a', 'b', 'c', 'a') // true
```

Restrictions:

Time - O(n)

Space - O(n)

- **Bonus:**

Time - O(n log n)

Space - O(1)

```
function areThereDuplicates() {
    let lookup = {};
    for (let i = 0; i < arguments.length; i++) {
        if (lookup[arguments[i]]) return true;
        lookup[arguments[i]] = 1;
    }
    return false;
}


areThereDuplicates(1, 2, 1);
```

- Using multiple pointers

```
function areThereDuplicates(...args) {
  // Two pointers
  args.sort((a,b) => a > b);
  let start = 0;
  let next = 1;
  while(next < args.length){
    if(args[start] === args[next]){
        return true
    }
    start++
    next++
  }
  return false
}
```

- Single line solution

```
function areThereDuplicates() {
  return new Set(arguments).size !== arguments.length;
}
```

Multiple Pointers - averagePair

Write a function called **averagePair.** Given a sorted array of integers and a target average, determine if there is a pair of values in the array where the average of the pair equals the target average. There may be more than one pair that matches the average target.

Bonus Constraints:

Time: O(N)

Space: O(1)

Sample Input:

```
averagePair([1,2,3],2.5) // true
averagePair([1,3,3,5,6,7,10,12,19],8) // true
averagePair([-1,0,3,4,5,6], 4.1) // false
averagePair([],4) // false
```

```
function averagePairMatch(a, num) {
    let i = 0;
    let j = a.length-1;
    while (i < a.length-1) {
        let avg = (a[i]+a[j])/2;
        console.log(i, j, a[i], a[j], avg);
        if (avg === num) return true;
        j--;
        if (j === i) {
            i++;
            j = a.length-1;
        }
    }
    return false;
}

//averagePairMatch([1,2,3], 2.5); // true
averagePairMatch([1,3,3,5,6,7,10,12,19], 8); // true
//averagePairMatch([-1,0,3,4,5,6], 4.1); // false
//averagePairMatch([], 4); // false
```

Multiple Pointers - isSubsequence

Write a function called **isSubsequence** which takes in two strings and checks whether the characters in the first string form a subsequence of the characters in the second string. In other words, the function should check whether the characters in the first string appear somewhere in the second string, **without their order changing.**

Examples:

```
isSubsequence('hello', 'hello world'); // true
isSubsequence('sing', 'sting'); // true
isSubsequence('abc', 'abracadabra'); // true
isSubsequence('abc', 'acb'); // false (order matters)
```

```
function isSubsequence(s1, s2) {
    s1 = s1.toLowerCase();
    s2 = s2.toLowerCase();
    let j = 0;
    for (let i = 0; i < s2.length; i++) {
        if (s2[i] === s1[j]) {
            j++;
            if (j === s1.length) return true;
            continue;
        }
    }
    return false;
}

//isSubsequence('hello', 'hello world'); // true
//isSubsequence('sing', 'sting'); // true
//isSubsequence('abc', 'abracadabra');
isSubsequence('abc', 'acb');
```

Sliding Window - maxSubarraySum

Given an array of integers and a number, write a function called **maxSubarraySum**, which finds the maximum sum of a subarray with the length of the number passed to the function.

Note that a subarray must consist of *consecutive* elements from the original array. In the first example below, [100, 200, 300] is a subarray of the original array, but [100, 300] is not.

```
maxSubarraySum([100,200,300,400], 2) // 700
maxSubarraySum([1,4,2,10,23,3,1,0,20], 4)  // 39
maxSubarraySum([-3,4,0,-2,6,-1], 2) // 5
maxSubarraySum([3,-2,7,-4,1,-1,4,-2,1],2) // 5
maxSubarraySum([2,3], 3) // null
```

Constraints:

Time Complexity - **O(N)**

Space Complexity - **O(1)**

```
function maxSubArraySum(a, num) {
    if (num > a.length) return null;
    let tempSum = 0;
    let maxSum = 0;
    for (let i = 0; i < a.length; i++) {
        if (i < num) {
            tempSum += a[i];
            if (i === num-1) maxSum = tempSum;
            continue;
        }
        tempSum += a[i] - a[i-num];
        maxSum = maxSum < tempSum ? tempSum : maxSum;
    }
    return maxSum;
}
maxSubArraySum([2, 6, 9, 2, 1, 8, 5, 6, 3], 4); // 22
//maxSubArraySum([100,200,300,400],2); // 700
//maxSubArraySum([1,4,2,10,23,3,1,0,20], 4);  // 39
//maxSubArraySum([-3,4,0,-2,6,-1], 2); // 5
//maxSubArraySum([3,-2,7,-4,1,-1,4,-2,1],2); // 5
//maxSubArraySum([2,3], 3); // null
```

Sliding Window - minSubArrayLen

- Write a function called **minSubArrayLen** which accepts two parameters - an array of positive integers and a positive integer.
- This function should return the minimal length of a **contiguous** subarray of which the sum is greater than or equal to the integer passed to the function. If there isn't one, return 0 instead.
  Examples:

```
minSubArrayLen([2,3,1,2,4,3], 7) // 2 -> because [4,3] is the smallest subarray
minSubArrayLen([2,1,6,5,4], 9) // 2 -> because [5,4] is the smallest subarray
minSubArrayLen([3,1,7,11,2,9,8,21,62,33,19], 52) // 1 -> because [62] is greater than 52
minSubArrayLen([1,4,16,22,5,7,8,9,10],39) // 3
minSubArrayLen([1,4,16,22,5,7,8,9,10],55) // 5
minSubArrayLen([4, 3, 3, 8, 1, 2, 3], 11) // 2
minSubArrayLen([1,4,16,22,5,7,8,9,10],95) // 0
```

- Time Complexity - O(n)
- Space Complexity - O(1)

```
function minSubArrayLen(a, num) {
    if (a.length === 0) return null;
    let tempSum = 0;
    let minSubArrayLen = Infinity;
    let startIndex = 0;
    let endIndex = 0;
    while (startIndex < a.length) {
        if (tempSum < num && endIndex < a.length) {
            tempSum += a[endIndex];
            endIndex++;
            continue;
        }
        if (tempSum >= num) {
            minSubArrayLen = Math.min(endIndex-startIndex, minSubArrayLen);
            tempSum -= a[startIndex];
            startIndex++;
            continue;
        }
        if (endIndex === a.length) break;
    }
    return minSubArrayLen === Infinity ? 0 : minSubArrayLen;
}

//minSubArrayLen([2,3,1,2,4,3], 7); // 2 -> since [4,3] is the smallest array
//minSubArrayLen([2,1,6,5,4], 9); // 2 -> since [5,4] is the smallest array
//minSubArrayLen([3,1,7,11,2,9,8,21,62,33,19], 52); // 1 -> since [62] is greater than 52
//minSubArrayLen([1,4,16,22,5,7,8,9,10],39) // 3
minSubArrayLen([1,4,16,22,5,7,8,9,10],55) // 5
//minSubArrayLen([4, 3, 3, 8, 1, 2, 3], 11) // 2
//minSubArrayLen([1,4,16,22,5,7,8,9,10],95) // 0
```

Sliding Window - findLongestSubstring

- Write a function called **findLongestSubstring**, which accepts a string and returns the length of the longest substring with all distinct characters.

```
findLongestSubstring('') // 0
findLongestSubstring('rithmschool') // 7
findLongestSubstring('thisisawesome') // 6
findLongestSubstring('thecatinthehat') // 7

findLongestSubstring('bbbbbb') // 1
findLongestSubstring('longestsubstring') // 8
findLongestSubstring('thisishowwedoit') // 6
```

- Time Complexity - **O(n)**

```
function findLongestSubstring(str){
  let lookup = {};
  let start = 0;
  let end = 0;
  let longestSubStrLen = 0;
  while (start < str.length) {
      if (!lookup[str[end]] && end < str.length) {
          lookup[str[end]] = 1;
          end++;
          continue;
      }
      longestSubStrLen = (end-start) > longestSubStrLen ? (end-start) : longestSubStrLen;
      start++;
      end=start;
      lookup = {};
  }
  return longestSubStrLen;
}


//findLongestSubstring(''); // 0
//findLongestSubstring('rithmschool'); // 7
//findLongestSubstring('thecatinthehat'); // 7
//findLongestSubstring('thisisawesome'); // 6
//findLongestSubstring('thisishowwedoit'); // 6
//findLongestSubstring('longestsubstring'); // 8
findLongestSubstring('bbbbb'); // 1
```

Recursion

Thursday, December 24, 2020 5:34 PM

What is recursion?

- A **process** (a function in our case) that **calls itself**

- Recursion Uses and example on where it is used:

It's EVERYWHERE!

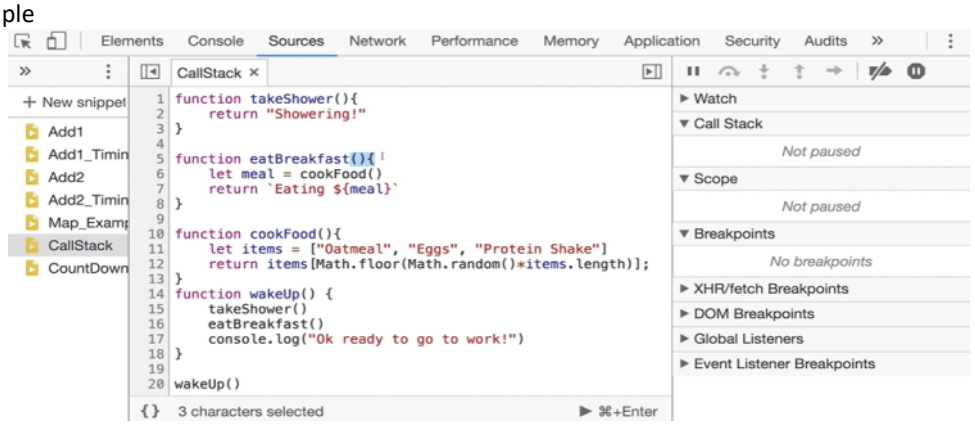
- JSON.parse / JSON.stringify
- document.getElementById and DOM traversal algorithms
- Object traversal
- We will see it with more complex data structures
- It's sometimes a cleaner alternative to iteration

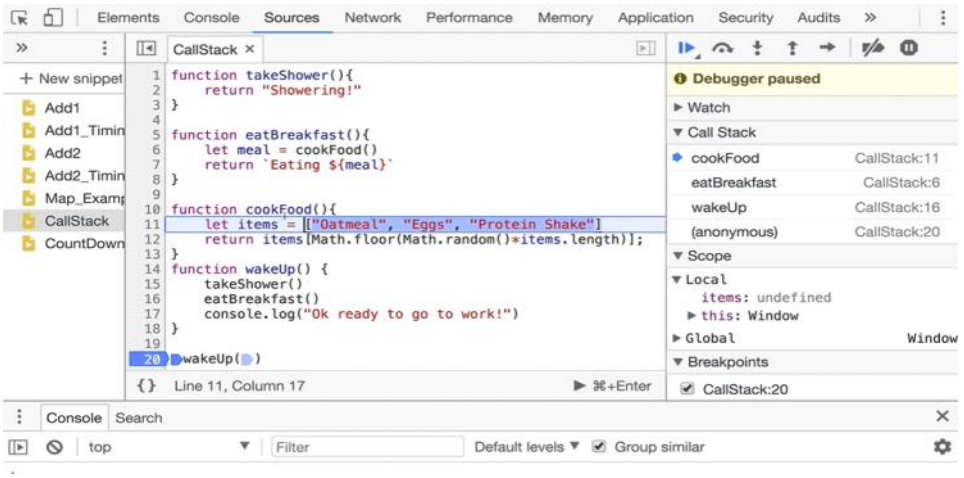
- Call Stack

The call stack

- It's a **stack** data structure - we'll talk about that later!
- Any time a function is invoked it is placed (**pushed**) on the top of the call stack
- When JavaScript sees the **return** keyword or when the function ends, the compiler will remove (**pop**)

- Example





Why do I care?

- You're used to functions being pushed on the call stack and popped off when they are done
- When we write recursive functions, we keep pushing new functions onto the call stack!

- Recursive function example

How recursive functions work

- Invoke the **same** function with a different input until you reach your base case!

Base Case

- The condition when the recursion ends.

This is the most important concept to understand

Two essential parts of a recursive function!

- Base Case
- Different Input

Our first recursive function

```
function countDown(num){
    if(num <= 0) {
        console.log("All done!");
        return;
    }
    console.log(num);
    num--;
    countDown(num);
}
```

Our second recursive function

```
function sumRange(num){
   if(num === 1) return 1;
   return num + sumRange(num-1);
}
```

Can you spot the base case?
Do you notice the different input?
What would happen if we didn't return?

- Example: Factorial using recursion

```
function factorial(num){
    if(num === 1) return 1
    return num * factorial(num-1)
}
```

- Recursion Pitfalls

Where things go wrong

- No base case
- Forgetting to return or returning the wrong thing!
- Stack overflow!

```
function factorial(num){
    if(num === 1) return 1;
    return num * factorial(num);
}
```

```
function factorial(num){
    if(num === 1) console.log(1) ;
    return num * factorial(num-1);
}
```

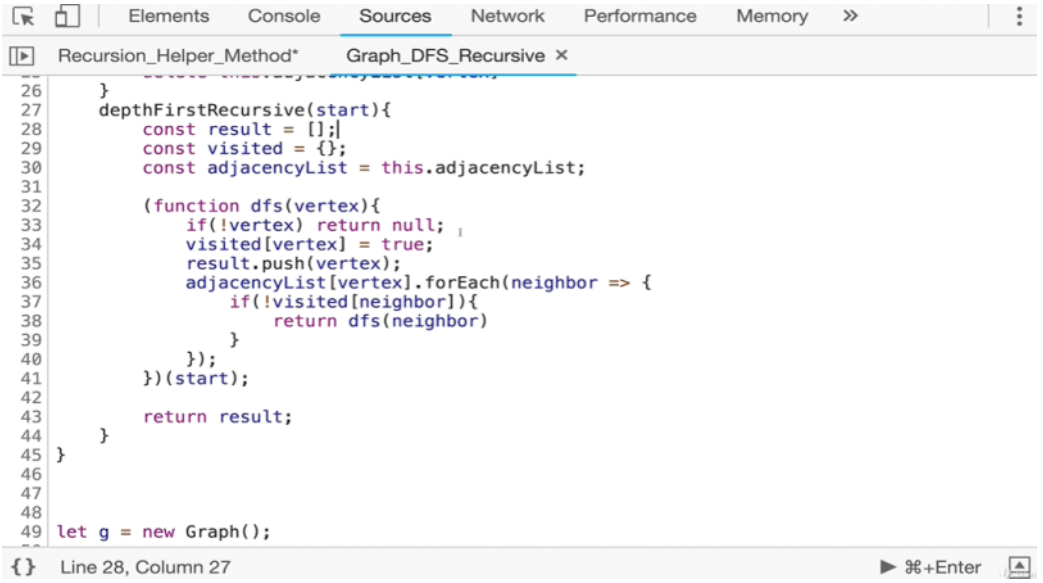
- Helper Method Recursion

HELPER METHOD RECURSION

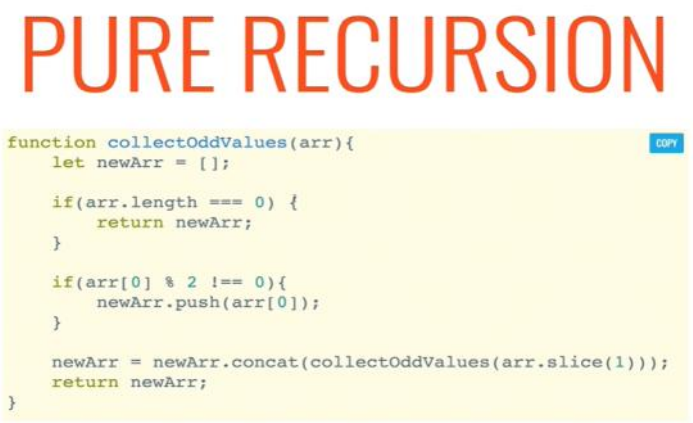
```
function outer(input){

    var outerScopedVariable = []

    function helper(helperInput){
        // modify the outerScopedVariable
        helper(helperInput--)
    }

    helper(input)

    return outerScopedVariable;

}
```

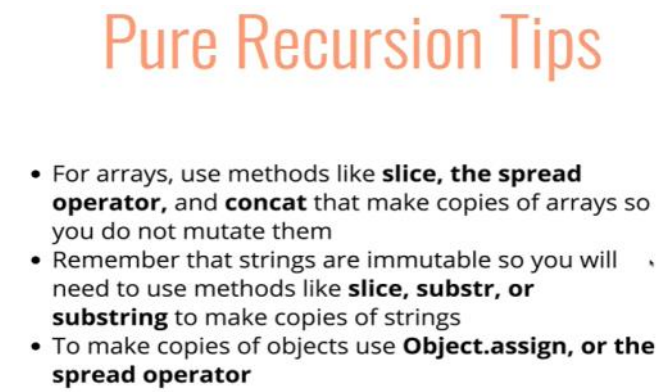
- Example:

```
function collectOddValues(arr){

    let result = []

    function helper(helperInput){
        if(helperInput.length === 0) {
            return;
        }

        if(helperInput[0] % 2 !== 0){
            result.push(helperInput[0])
        }

        helper(helperInput.slice(1))
    }

    helper(arr)

    return result;
}

collectOddValues([1,2,3,4,5])
```



- Pure Recursion





- Problems

power

- Write a function called power which accepts a base and an exponent. The function should return the power of the base to the exponent. This function should mimic the functionality of `Math.pow()` - do not worry about negative bases and exponents.

```
function power(base, exp) {
    if (exp === 0) return 1;
    return base * power(base, exp-1);
}
```

```
power(2, 1); // 2
// power(2, 2); // 4
// power(2, 3); // 8
// power(2, 4); // 16
```

factorial

- Write a function **factorial** which accepts a number and returns the factorial of that number. A factorial is the product of an integer and all the integers below it; e.g., factorial four (*4!*) is equal to 24, because 4 * 3 * 2 * 1 equals 24. **factorial zero (0!) is always 1.**

```
function factorial(num){
   if (num <= 1) return 1;
   return num * factorial(num-1);
}
```

```
// factorial(1) // 1
// factorial(2) // 2
// factorial(4) // 24
// factorial(7) // 5040
```

productOfArray

- Write a function called `productOfArray` which takes in an array of numbers and returns the product of them all.

```
function productOfArray(a) {
    let product;
    function helper(arr) {
        if (arr.length === 0) {
            return;
        }
        product = typeof product !== 'undefined' ? product * arr[0] : arr[0];
        helper(arr.splice(1));
    }
    helper(a);
    return product;
}
//productOfArray([1,2,3]) // 6
productOfArray([1,2,3,10]) // 60
```

```
function productOfArray(a) {
    if (a.length === 0) return 1;
    return a[0] * productOfArray(a.slice(1));
}

//productOfArray([1,2,3]) // 6
productOfArray([1,2,3,10]) // 60
```

recursiveRange

- Write a function called **recursiveRange** which accepts a number and adds up all the numbers from 0 to the number passed to the function

```
function recursiveRange(num){
   let sum;
   function helper(n) {
       if (n === 0) return;
       sum = typeof sum !== 'undefined' ? sum + n : n;
       helper(n-1);
   }
   helper(num)
   return sum;
}


recursiveRange(6) // 21
// recursiveRange(10) // 55
```

```
function recursiveRange(num){
   if (num === 0) return 0;
   return num + recursiveRange(num - 1);
}
```

fib

- Write a recursive function called **fib** which accepts a number and returns the *n*th number in the Fibonacci sequence. Recall that the Fibonacci sequence is the sequence of whole numbers 1, 1, 2, 3, 5, 8, ... which starts with 1 and 1, and where every number thereafter is equal to the sum of the previous two numbers.

```
function fib(num){
    if (num <= 2) return 1;
    return fib(num-1) + fib(num-2);
}
```

reverse

- Write a recursive function called **reverse** which accepts a string and returns a new string in reverse.

```
function reverse(str) {
    if (str.length === 0) return '';
    return str[str.length-1] + reverse(str.substr(0, str.length-1));
}

// reverse('awesome') // 'emosewa'
reverse('rithmschool') // 'loohcsmhtir'
```

```
function reverse(str){
    if(str.length <= 1) return str;
    return reverse(str.slice(1)) + str[0];
}
```

isPalindrome

- Write a recursive function called **isPalindrome** which returns true if the string passed to it is a palindrome (reads the same forward and backward). Otherwise it returns false.

```
function isPalindrome(str){
    const reverseStr = (s) => {
        if (s.length === 0) return '';
        return s[s.length-1] + reverseStr(s.substr(0, s.length-1));
    }
    return str === reverseStr(str);
}

// isPalindrome('awesome') // false
// isPalindrome('foobar') // false
isPalindrome('tacocat') // true
// isPalindrome('amanaplanacanalpanama') // true
// isPalindrome('amanaplanacanalpandemonium') // false
```

```
function isPalindrome(str){
    if(str.length === 1) return true;
    if(str.length === 2) return str[0] === str[1];
    if(str[0] === str.slice(-1)) return isPalindrome(str.slice(1,-1))
    return false;
}
```

someRecursive

- Write a recursive function called **someRecursive** which accepts an array and a callback. The function returns true if a single value in the array returns true when passed to the callback. Otherwise it returns false.

```
function someRecursive(a, condition){
    let isConditionTrue = false;
    function helper(arr) {
        if (arr.length === 0 || isConditionTrue) return;
        isConditionTrue = condition(arr[0]);
        helper(arr.splice(1));
    }
    helper(a);
    return isConditionTrue;
}

// SAMPLE INPUT / OUTPUT
var isOdd = val => val % 2 !== 0;

// someRecursive([1,2,3,4], isOdd) // true
someRecursive([4,6,8,9], isOdd) // true
// someRecursive([4,6,8], isOdd) // false
// someRecursive([4,6,8], val => val > 10); // false
```

```
function someRecursive(array, callback) {
    if (array.length === 0) return false;
    if (callback(array[0])) return true;
    return someRecursive(array.slice(1),callback);
}
```

flatten

- Write a recursive function called **flatten** which accepts an array of arrays and returns a new array with all values flattened.

```
function flatten(oldArr){
  var newArr = []
    for(var i = 0; i < oldArr.length; i++){
        if(Array.isArray(oldArr[i])){
            newArr = newArr.concat(flatten(oldArr[i]))
        } else {
            newArr.push(oldArr[i])
        }
  }
  return newArr;
}

// flatten([1, 2, 3, [4, 5] ]) // [1, 2, 3, 4, 5]
// flatten([1, [2, [3, 4], [[5]]]]) // [1, 2, 3, 4, 5]
// flatten([[1],[2],[3]]) // [1,2,3]
flatten([[[[1], [[[2]]], [[[[[[[3]]]]]]]]]) // [1,2,3]
```

capitalizeFirst

Write a recursive function called **capitalizeFirst**. Given an array of strings, capitalize the first letter of each string in the array.

```
function capitalizeFirst(a, newArr = []) {
    if (a.length === 0) return newArr;
    a[0] = a[0][0].toUpperCase() + a[0].substr(1);
    newArr.push(a[0]);
    return capitalizeFirst(a.splice(1), newArr);
}

capitalizeFirst(['car','taco','banana']); // ['Car','Taco','Banana']
```

nestedEvenSum

Write a recursive function called **nestedEvenSum**. Return the sum of all even numbers in an object which may contain nested objects.

```
function nestedEvenSum(obj, sum = 0) {
    for (var key in obj) {
        if (typeof obj[key] === 'number' && obj[key] % 2 === 0) sum += obj[key];
        if (typeof obj[key] === 'object') sum += nestedEvenSum(obj[key]);
    }
    return sum;
}

var obj1 = {
  outer: 2,
  obj: {
    inner: 2,
    otherObj: {
      superInner: 2,
      notANumber: true,
      alsoNotANumber: "yup"
    }
  }
}
nestedEvenSum(obj1); // 6


var obj2 = {
  a: 2,
  b: {b: 2, bb: {b: 3, bb: {b: 2}}},
  c: {c: {c: 2}, cc: 'ball', ccc: 5},
  d: 1,
  e: {e: {e: 2}, ee: 'car'}
};

// nestedEvenSum(obj2); // 10
```

capitalizeWords

- Write a recursive function called **capitalizeWords**. Given an array of words, return a new array containing each word capitalized.

```
function capitalizeWords(a, newArr = []) {
    if (a.length === 0) return newArr;
    newArr.push(a[0].toUpperCase());
    return capitalizeWords(a.splice(1), newArr);
}

var words = ['i', 'am', 'learning', 'recursion'];
capitalizeWords(words); // ['I', 'AM', 'LEARNING', 'RECURSION']
```

stringifyNumbers

- Write a function called `stringifyNumbers` which takes in an object and finds all of the values which are numbers and converts them to strings. Recursion would be a great way to solve this!

```
function stringifyNumbers(obj, newObj = {}) {
    for (let key in obj) {
        if (typeof obj[key] === 'number') {
            newObj[key] = obj[key].toString();
        } else if (typeof obj[key] === 'object' && obj[key].constructor !== Array) {
            newObj[key] = stringifyNumbers(obj[key]);
        } else {
            newObj[key] = obj[key];
        }
    }
    return newObj;
}


let obj = {
    num: 1,
    test: [],
    data: {
        val: 4,
        info: {
            isRight: true,
            random: 66
        }
    }
}
stringifyNumbers(obj);
// {
//     num: "1",
//     test: [],
//     data: {
//         val: "4",
//         info: {
//             isRight: true,
//             random: "66"
//         }
//     }
// }
```

collectStrings

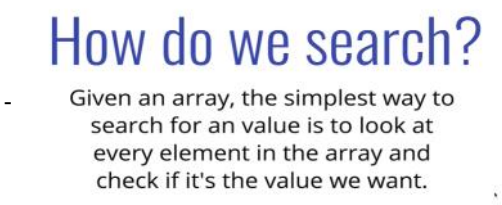
- Write a function called `collectStrings` which accepts an object and returns an array of all the values in the object that have a typeof string

```
function collectStrings(obj, newArr = []) {
    for (let key in obj) {
        if (typeof obj[key] === 'string') {
            newArr.push(obj[key]);
        }
        if (typeof obj[key] === 'object') {
            collectStrings(obj[key], newArr);
        }
    }
    return newArr;
}

var obj1 = {
    stuff: "foo",
    data: {
        val: {
            thing: {
                info: "bar",
                moreInfo: {
                    evenMoreInfo: {
                        weMadeIt: "baz"
                    }
                }
            }
        }
    }
}

collectStrings(obj1) // ["foo", "bar", "baz"])
```

Searching Algorithms

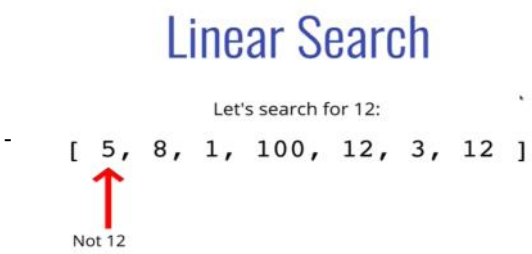
Tuesday, January 5, 2021 4:04 PM

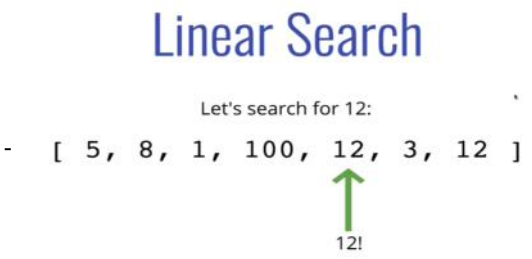
Linear Search

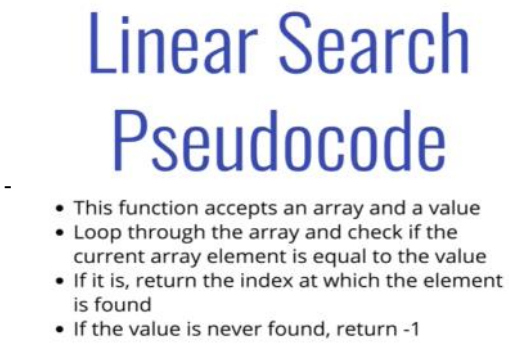
- 

- 

- Linear Search is about checking every item in the given array either from the first index / last index moving forward/backward of the array.

- 

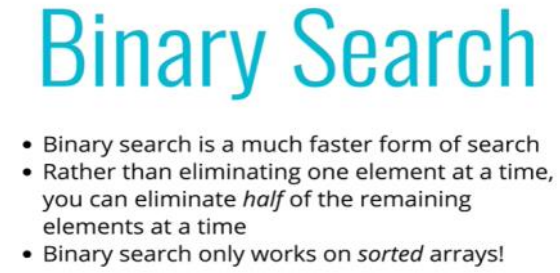
- 

- 

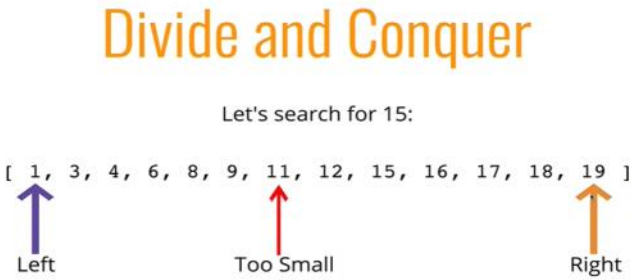
```
function linearSearch(a, val){
    for (let i = 0; i < a.length; i++) {
        if (val === a[i]) return i;
    }
    return -1;
}

linearSearch([10, 15, 20, 25, 30], 15) // 1
// linearSearch([9, 8, 7, 6, 5, 4, 3, 2, 1, 0], 4) // 5
// linearSearch([100], 100) // 0
// linearSearch([1,2,3,4,5], 6) // -1
// linearSearch([9, 8, 7, 6, 5, 4, 3, 2, 1, 0], 10) // -1
// linearSearch([100], 200) // -1
// linearSearch([10, 15, 20, 25, 30], 15) // 1
// linearSearch([9, 8, 7, 6, 5, 4, 3, 2, 1, 0], 4) // 5
// linearSearch([100], 100) // 0
// linearSearch([1,2,3,4,5], 6) // -1
// linearSearch([9, 8, 7, 6, 5, 4, 3, 2, 1, 0], 10) // -1
// linearSearch([100], 200) // -1
```

- Order is O(n)

Binary Search

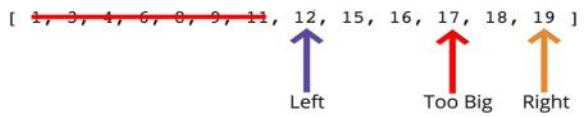


- Example



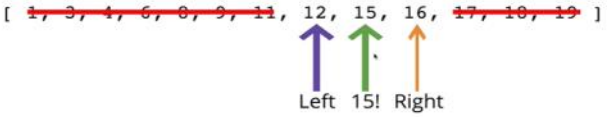
Divide and Conquer

Let's search for 15:



Divide and Conquer

Let's search for 15:



Binary Search Pseudocode

- This function accepts a sorted array and a value
- Create a left pointer at the start of the array, and a right pointer at the end of the array
- While the left pointer comes before the right pointer:
  - Create a pointer in the middle
  - If you find the value you want, return the index
  - If the value is too small, move the left pointer up
  - If the value is too large, move the right pointer down
- If you never find the value, return -1

```
function binarySearch(a, val, index = 0){
    let center = Math.ceil(a.length/2);
    let leftArr = a.slice(0, center);
    let rightArr = a.slice(center);
    if (leftArr.length === 0 || rightArr.length === 0) {
        return -1;
    }
    if (leftArr[center-1] >= val) {
        if (leftArr.length === 1 && leftArr[center-1] === val) {
            return index;
        }
        index = binarySearch(leftArr, val, index);
    }
    if (rightArr[0] <= val) {
        index = index + center;
        if (rightArr.length === 1 && rightArr[0] === val) {
            return index;
        }
        index = binarySearch(rightArr, val, index);
    }
    return index;
}


// binarySearch([1,2,3,4,5],2) // 1
// binarySearch([1,2,3,4,5],3) // 2
binarySearch([1,2,3,4,5],5) // 4
// binarySearch([1,2,3,4,5],6) // -1
// binarySearch([5,6,10, 13,14,18,30,34,35,37,40,44,64,79,84,86,95,96,98,99], 10) // 2
// binarySearch([5,6,10, 13,14,18,30,34,35,37,40,44,64,79,84,86,95,96,98,99], 95) // 16
// binarySearch([5,6,10, 13,14,18,30,34,35,37,40,44,64,79,84,86,95,96,98,99], 100) // -1
// binarySearch([1,2,3,4,5],2) // 1
// binarySearch([1,2,3,4,5],3) // 2
// binarySearch([1,2,3,4,5],5) // 4
// binarySearch([1,2,3,4,5],6) // -1
```

```
function binarySearch(arr, elem) {
    let start = 0;
    let end = arr.length - 1;
    let center = Math.floor((start + end) / 2);
    while(arr[center] !== elem && start <= end) {
        if(elem < arr[center]) end = center - 1;
        else start = center + 1;
        center = Math.floor((start + end) / 2);
    }
    return arr[center] === elem ? center : -1;
}
```

Suppose we're searching for 13

[2,4,5,9,11,14,15,19,21,25,28,30,50,52,60,63]

[2,4,5,9,11,14,15]

- [11,14,15]

[11]

NOPE, NOT HERE!

16 elements = 4 "steps"

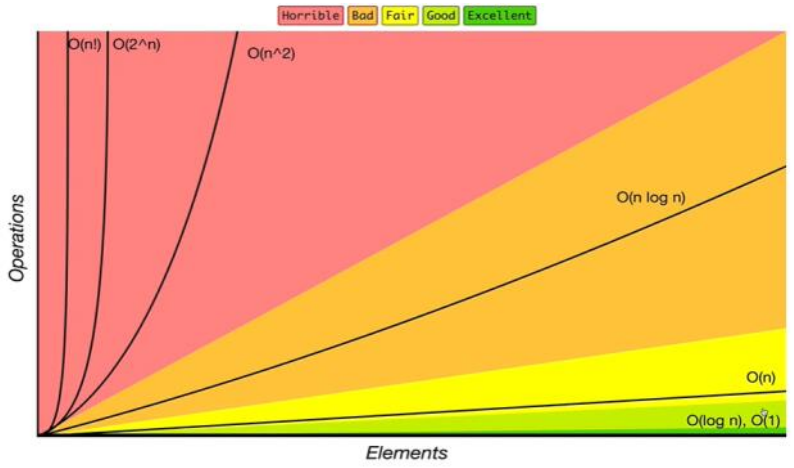
To add another "step", we need to double the number of elements

Let's search for 32

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

- [~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]
- [~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]
- [~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28~~,29,30,32,35]
- [~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30~~,32,35]
- [~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30~~,32,35]

32 elements = 5 "steps" (worst case)



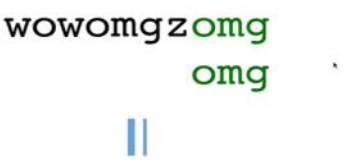
Common Data Structure Operations

Naïve Search

Naive String Search

Long string: wowomgzomg

Short string: omg



Pseudocode

- Loop over the longer string
- Loop over the shorter string
- If the characters don't match, break out of the inner loop
- If the characters do match, keep going
- If you complete the inner loop and find a match, increment the count of matches
- Return the count

```
function naiveSearch(long, short){
    var count = 0;
    for(var i = 0; i < long.length; i++){
        for(var j = 0; j < short.length; j++){
            if(short[j] !== long[i+j]) break;
            if(j === short.length - 1) count++;
        }
    }
    return count;
}

naiveSearch("lorie loled", "lol")
```

```
function naiveStringSearch(str, searchStr) {
    let count = 0;
    for (let i = 0; i < str.length; i++) {
        if (str[i] === searchStr[0]) {
            let subS = str.substr(i, searchStr.length);
            count = subS === searchStr ? count+1 : count;
        }
    }
    return count;
}

naiveStringSearch('wowomgzomg', 'omg'); //2
// naiveStringSearch('lorie loled', 'lol'); //1
// naiveStringSearch('lorie loled', 'pop'); //0
```

Sorting Algorithms

Wednesday, January 6, 2021 4:43 PM

- https://www.toptal.com/developers/sorting-algorithms

Big O of Sorting Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)

- For nearly sorted data Bubble Sort and Insertion Sort has order O(n) but selection sort has always order of O(n^2).

- The sorting algorithms we've learned so far don't scale well
- Try out bubble sort on an array of 100000 elements, it will take quite some time!

Bubble_Sort_Slow ×

```
var data = Array.apply(null, {length: 100000}).map(Function.call, Math.random)
bubbleSort(data);
```

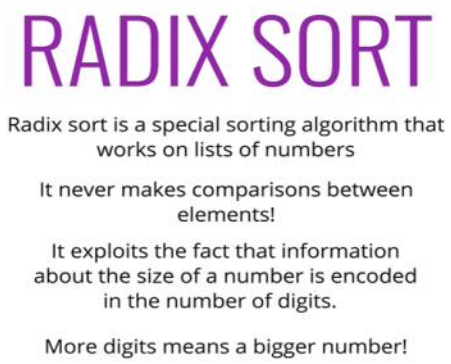
```
var data = Array.apply(null, {length: 100000}).map(Function.call, Math.random)
mergeSort(data);
```

- The same with merge sort is much more faster than bubble sort.

FASTER SORTS

- There is a family of sorting algorithms that can improve time complexity from O(n^2) to O(n log n)
- There's a tradeoff between efficiency and simplicity
- The more efficient algorithms are much less simple, and generally take longer to understand

- All the above are comparison sorts
- There are other type of non-sorting algorithms which do not make direct comparisons and works only on numbers. One of which is **RADIX SORT**



JavaScript Sort

Wednesday, January 6, 2021 4:42 PM

- **Built-In JavaScript sort**

JavaScript has a sort method...

Yes, it does!

...but it doesn't always work the way you expect.

- ☺

```
[ "Steele", "Colt", "Data Structures", "Algorithms" ].sort();
// [ "Algorithms", "Colt", "Data Structures", "Steele" ]
```

☹

```
[ 6, 4, 15, 10 ].sort();
// [ 10, 15, 4, 6 ]
```

- With alphabets, sorting works fine but not for numbers
- The `sort()` method sorts the elements of an array *in place* and returns the array. The sort is not necessarily stable. The default sort order is according to string Unicode code points.

Telling JavaScript how to sort

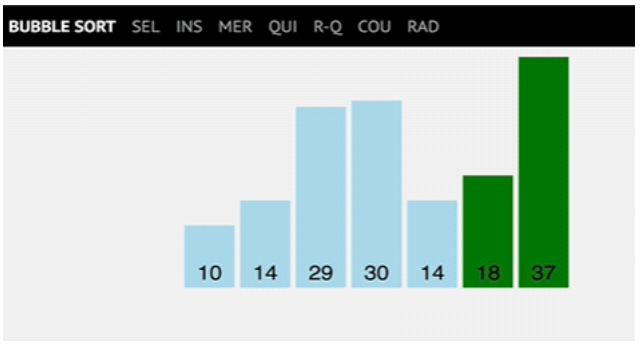
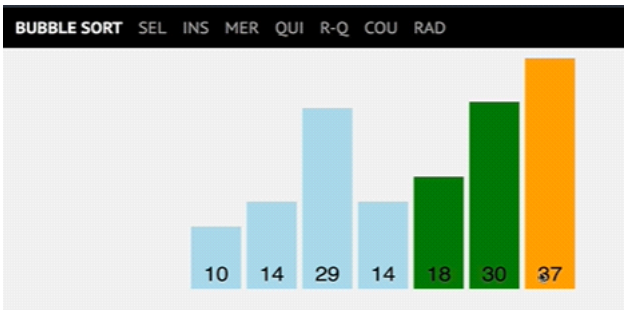
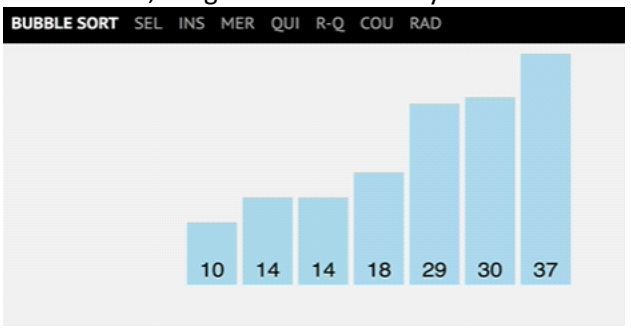
- The built-in sort method accepts an optional *comparator* function
- You can use this comparator function to tell JavaScript how you want it to sort
- The comparator looks at pairs of elements (*a* and *b*), determines their sort order based on the return value
  - If it returns a negative number, *a* should come before *b*
  - If it returns a positive number, *a* should come after *b*,
  - If it returns 0, *a* and *b* are the same as far as the sort is concerned

Telling JavaScript how to sort

Examples

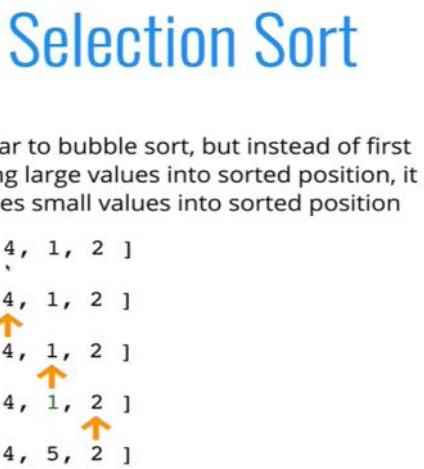
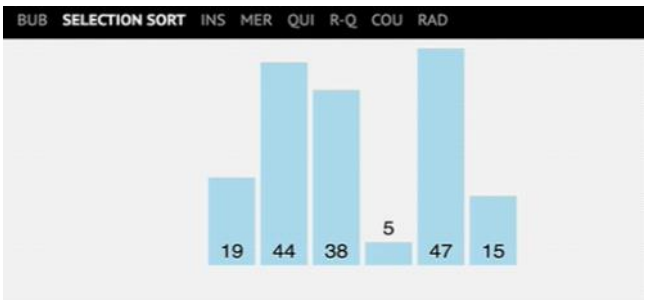
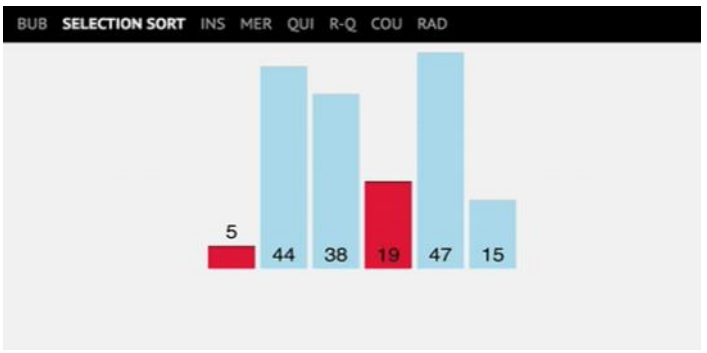
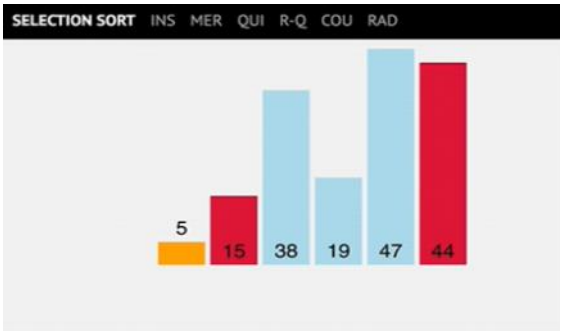
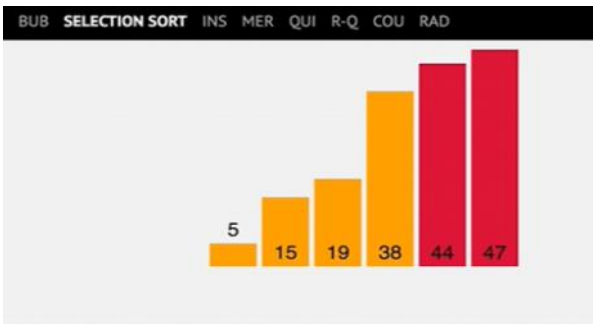
```
function numberCompare(num1, num2) {
  return num1 - num2;
}

[ 6, 4, 15, 10 ].sort(numberCompare);
// [ 4, 6, 10, 15 ]
```

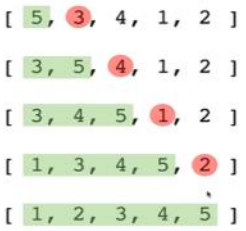
```
// function numberCompare(num1, num2) {
//   return num2 - num1;
// }

// [ 6, 4, 15, 10 ].sort(numberCompare);
```

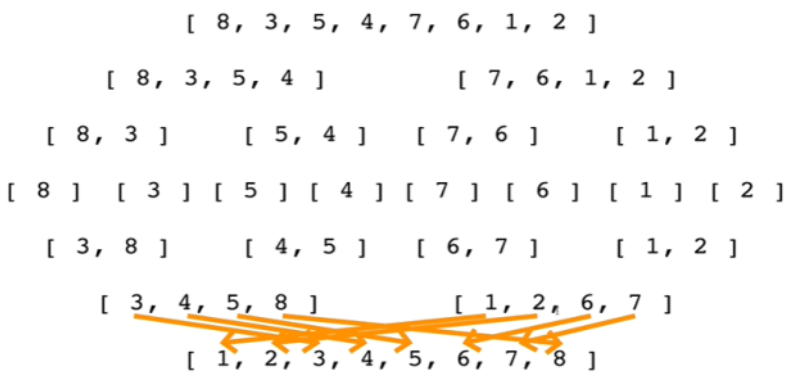
- ```
  function compareByLen(str1, str2) {
    return str1.length - str2.length;
  }

  [ "Steele", "Colt", "Data Structures", "Algorithms" ]
    .sort(compareByLen);
  ```
  ```

Bubble Sort

Wednesday, January 6, 2021 1:01 PM

- Bubble sort works on swapping values by comparing with the next one
- Initial
  - 

- After 1st iteration
  - 

- After 2nd iteration
  - 

- After 3rd iteration, we get the sorted array
  - 

- Another Example

BubbleSort

A sorting algorithm where the largest values bubble up to the top!

- The process of swapping should continue until the array is sorted.

Before we sort, we must swap!

Many sorting algorithms involve some type of swapping functionality (e.g. swapping to numbers to put them in order)

```
// ES5
function swap(arr, idx1, idx2) {
  var temp = arr[idx1];
  arr[idx1] = arr[idx2];
  arr[idx2] = temp;
}

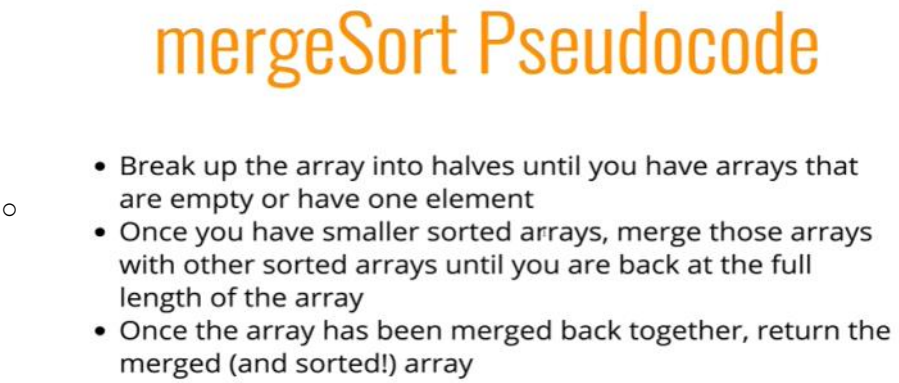
// ES2015
const swap = (arr, idx1, idx2) => {
  [arr[idx1],arr[idx2]] = [arr[idx2],arr[idx1]];
}
```

BubbleSort Pseudocode

- Start looping from with a variable called i the end of the array towards the beginning
- Start an inner loop with a variable called j from the beginning until i - 1
- If arr[j] is greater than arr[j+1], swap those two values!
- Return the sorted array

```
// UNOPTIMIZED VERSION OF BUBBLE SORT
function bubbleSort(arr){
  for(var i = arr.length; i > 0; i--){
    for(var j = 0; j < i - 1; j++){
      console.log(arr, arr[j], arr[j+1]);
      if(arr[j] > arr[j+1]){
        var temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
  return arr;
}
```

-

```
// ES2015 Version
function bubbleSort(arr) {
  const swap = (arr, idx1, idx2) => {
    [arr[idx1], arr[idx2]] = [arr[idx2], arr[idx1]];
  };

  for (let i = arr.length; i > 0; i--) {
    for (let j = 0; j < i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        swap(arr, j, j + 1);
      }
    }
  }
  return arr;
}
```

```
//Optimized Bubble sort
function bubbleSort(arr){
  var noSwaps;
  for(var i = arr.length; i > 0; i--){
    noSwaps = true;
    for(var j = 0; j < i - 1; j++){
      if(arr[j] > arr[j+1]){
        var temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
        noSwaps = false;
      }
    }
    if(noSwaps) break;
  }
  return arr;
}
```

-
```
// My Solution
function bubbleSort(a) {
    let swapping = false;
    for (let i = 0; i < a.length; i++) {
        let j = i+1;
        if (j > a.length-1) break;
        if (a[i] > a[j]) {
            // let temp = a[j];
            // a[j] = a[i];
            // a[i] = temp;
            [a[i], a[j]] = [a[j], a[i]]; // swapping in es5
            swapping = true;
        }
    }
    if (swapping) {
        return bubbleSort(a);
    }
    return a;
}

bubbleSort([5,3,4,1,2]); // [1,2,3,4,5]
```

- Big O Notation for Bubble Sort (Worst Case) - O(n^2) -> Before Optimization
- Big O Notation for Bubble Sort (Best Case, Almost Sorted) - O(n) -> After optimization

Selection Sort

Wednesday, January 6, 2021 4:00 PM



- In the selection sort we go through the whole array, find out the minimum value and swap with the first value in the array
- The next time we iterate we leave the first value and start from the second value to the end of array to find the minimum.

- Example
  - 

  - 

- 

- 

- 

Selection Sort Pseudocode

- Store the first element as the smallest value you've seen so far.
- Compare this item to the next item in the array until you find a smaller number.
- If a smaller number is found, designate that smaller number to be the new "minimum" and continue until the end of the array.
- If the "minimum" is not the value (index) you initially began with, swap the two values.
- Repeat this with the next element until the array is sorted.

-

```
function selectionSort(a) {
    for (let i = 0; i < a.length; i++) {
        let minIndex = i;
        for (let j = i+1; j < a.length; j++) {
            if (a[minIndex] > a[j]) minIndex = j;
        }
        if (i !== minIndex) [a[i], a[minIndex]] = [a[minIndex], a[i]];
    }
    return a;
}

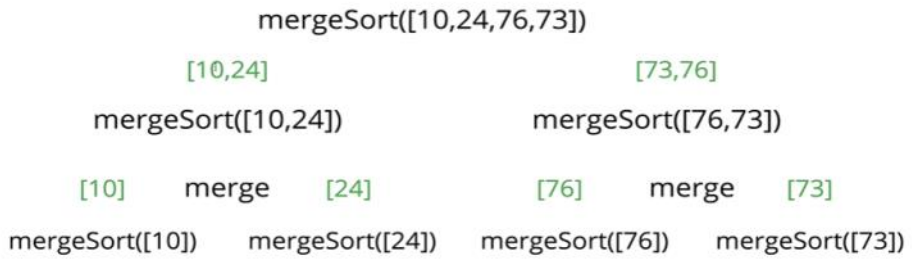
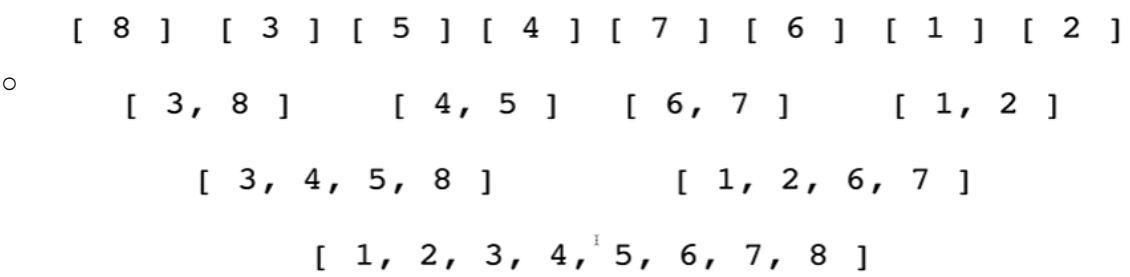
selectionSort([19,44,38,5,47,15]); // [5,15,19,38,44,47]
```

- Big O Notation for Selection Sort -> O(n^2)

Insertion Sort

Wednesday, January 6, 2021 4:44 PM

Insertion Sort

Builds up the sort by gradually creating a larger left half which is always sorted

- 

- Insertion sort deals with sorting by taking an element one at a time and inserting in the right position.

Insertion Sort Pseudocode

-
  - Start by picking the second element in the array
  - Now compare the second element with the one before it and swap if necessary.
  - Continue to the next element and if it is in the incorrect order, iterate through the sorted portion (i.e. the left side) to place the element in the correct place.
  - Repeat until the array is sorted.

```
function insertionSort(a) {
    for (let i = 1; i < a.length; i++) {
        let currentVal = a[i];
        for (var j = i-1; j >= 0; j--) {
            if (a[j] < currentVal) break;
            a[j+1] = a[j];
        }
        a[j+1] = currentVal;
    }
    return a;
}


insertionSort([19,44,38,5,47,15]); // [5,15,19,38,44,47]
```

- Big O Notation (Best Case) -> O(n)
- Big O Notation (Worst Case) -> O(n^2)

Merge Sort

Friday, January 2021 1:53 PM

Merge Sort

- It's a combination of two things - merging and sorting!
- Exploits the fact that arrays of 0 or 1 element are always sorted
- Works by decomposing an array into smaller arrays of 0 or 1 elements, then building up a newly sorted array

How does it work?



- Merge Sort - Part 1 - To merge two sorted arrays into one sorted array

Merging Arrays

- In order to implement merge sort, it's useful to first implement a function responsible for merging two sorted arrays
- Given two arrays which are sorted, this helper function should create a new array which is also sorted, and consists of all of the elements in the two input arrays
- This function should run in **O(n + m)** time and **O(n + m)** space and **should not** modify the parameters passed to it.

```
// Function to merge two sorted arrays into one sorted array
function merge(a, b) {
    let mergedArr = [];
    let i = 0;
    let j = 0;
    while (i < a.length && j < b.length) {
        if(a[i] < b[j]) {
            mergedArr.push(a[i]);
            i++;
            continue;
        }
        mergedArr.push(b[j])
        j++;
    }
    while (i < a.length) {
        mergedArr.push(a[i]);
        i++;
    }
    while (j < b.length) {
        mergedArr.push(b[j]);
        j++;
    }
    return mergedArr;
}

merge([1,10,50], [2,14,99,100]);
// merge([100,200], [1,2,3,5,6])
```

- Merge Sort - Part 2 - Break up a given unsorted array until it is an array of 1 element and merge the sorted arrays




```
function mergeSort(a) {
    if (a.length <= 1) return a;
    let center = Math.floor(a.length/2);
    console.log(a.slice(0,center), a.slice(center));
    let leftArr = mergeSort(a.slice(0,center));
    let rightArr = mergeSort(a.slice(center));
    return mergeArray(leftArr, rightArr);
}


mergeSort([1,50,100,99,14,2,10]);
```

- Merge Sort - Final (Part1 + Part 2)

```
// Function to merge two sorted arrays into one sorted array
function mergeArray(a, b) {
    let mergedArr = [];
    let i = 0;
    let j = 0;
    while (i < a.length && j < b.length) {
        if(a[i] < b[j]) {
            mergedArr.push(a[i]);
            i++;
            continue;
        }
        mergedArr.push(b[j])
        j++;
    }
    while (i < a.length) {
        mergedArr.push(a[i]);
        i++;
    }
    while (j < b.length) {
        mergedArr.push(b[j]);
        j++;
    }
    return mergedArr;
}

function mergeSort(a) {
    if (a.length <= 1) return a;
    let center = Math.floor(a.length/2);
    console.log(a.slice(0,center), a.slice(center));
    let leftArr = mergeSort(a.slice(0,center));
    let rightArr = mergeSort(a.slice(center));
    return mergeArray(leftArr, rightArr);
}

mergeSort([1,50,100,99,14,2,10]);
```



Big O of mergeSort

-

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
O(*n* log *n*)	O(*n* log *n*)	O(*n* log *n*)	O(*n*)

Big O of mergeSort

Why???



- If we have 8 items in the array we had to split it 3 times as shown above (2^3 = 8)
- if we have 32 items in the array we have to split it 5 times (2^5 = 32)
- The above examples derive us to O(log n)
- Why O(n log n) -> As n grows the merge complexity grows to O(n)
- So totally the complexity ends up with O(n log n)

Quick Sort

Monday, January 11, 2021 4:09 PM

Quick Sort

- Like merge sort, exploits the fact that arrays of 0 or 1 element are always sorted
- Works by selecting one element (called the "pivot") and finding the index where the pivot should end up in the sorted array
- Once the pivot is positioned appropriately, quick sort can be applied on either side of the pivot

How does it work?

```
[ 5, 2, 1, 8, 4, 7, 6, 3 ]
```

- We pick the pivot point as "**5**" as in the above example
- Then, we move all the numbers which are "**< 5**" to the **left** of it and the numbers "**> 5**" to the **right** of it.

How does it work?

```
[ 5, 2, 1, 8, 4, 7, 6, 3 ]

                5
3, 2, 1, 4    7, 6, 8
```

- Then we pick another pivot point "**3**" and repeat the same

How does it work?

```
[ 5, 2, 1, 8, 4, 7, 6, 3 ]

                5
        3         7, 6, 8
  1, 2      4
```

- Then we pick "**1**" as the pivot point and repeat the same

How does it work?

```
[ 5, 2, 1, 8, 4, 7, 6, 3 ]

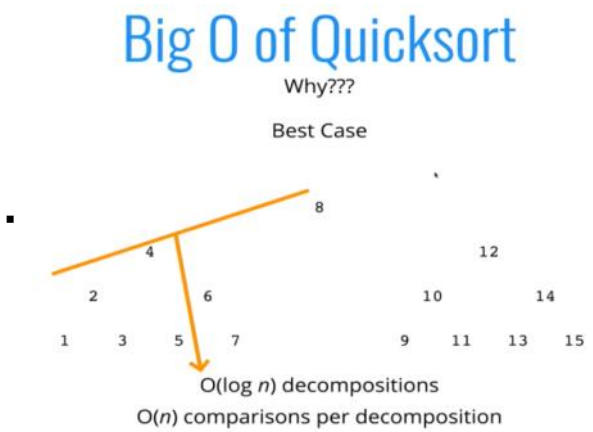
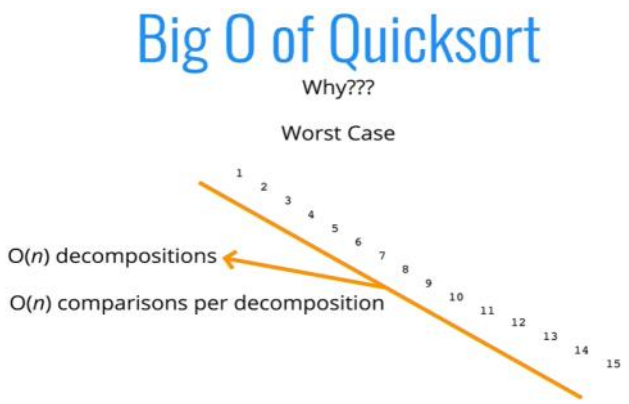
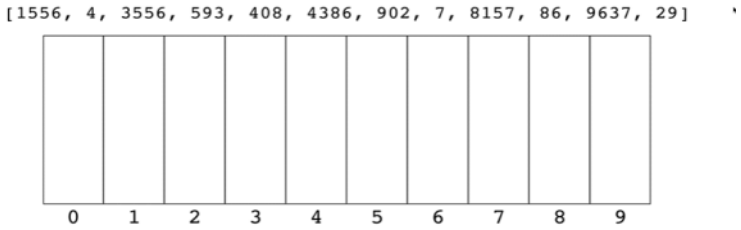
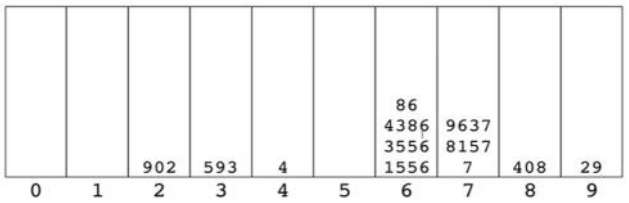
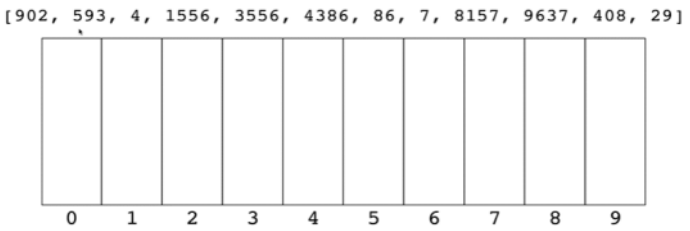
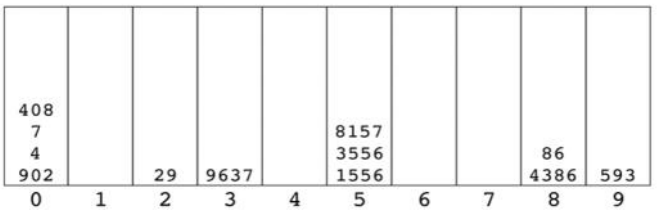
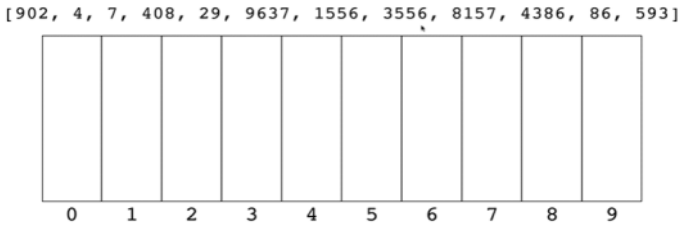
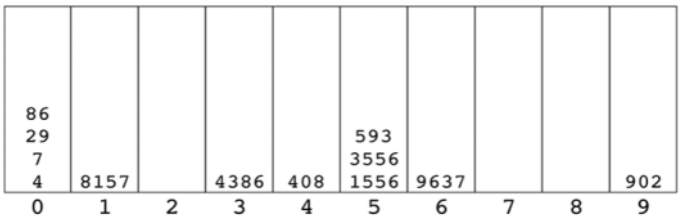
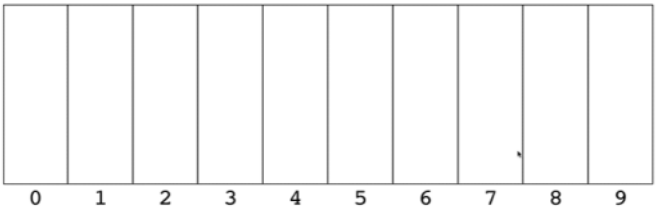
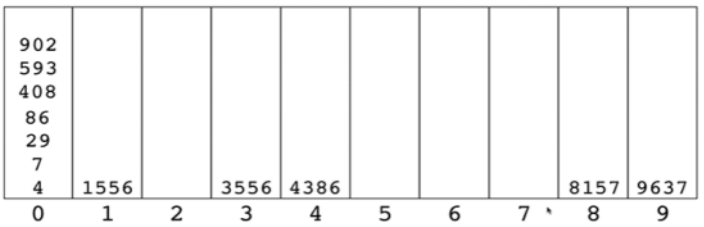
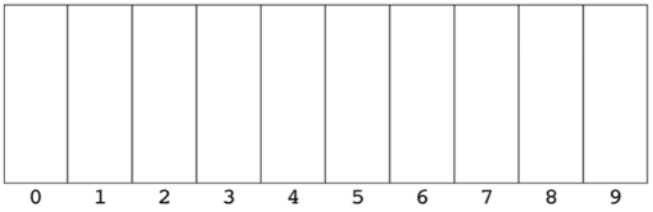
                5
        3               7
    1       4       6       8
      2
```

- Pseudocode

Picking a pivot

- The runtime of quick sort depends in part on how one selects the pivot
- Ideally, the pivot should be chosen so that it's roughly the median value in the data set you're sorting

Pivot Pseudocode

- It will help to accept three arguments: an array, a start index, and an end index (these can default to 0 and the array length minus 1, respectively)
- Grab the pivot from the start of the array
- Store the current pivot index in a variable (this will keep track of where the pivot should end up)
- Loop through the array from the start until the end
  - If the pivot is greater than the current element, increment the pivot index variable and then swap the current element with the element at the pivot index
- Swap the starting element (i.e. the pivot) with the pivot index

- My Solution

```
function pivot(a, start=0, end=a.length-1) {
    while (end > start) {
        if (a[start] > a[end]) {
            start++;
            [a[start],a[end]] = [a[end], a[start]];
            [a[start-1], a[start]] = [a[start], a[start-1]];
            console.log(a, start, end);
            continue;
        }
        end--;
    }
    return start;
}

pivot([4,8,2,1,5,7,6,3]); // 3
pivot([9,4,8,2,1,5,7,6,3]); // 8
```

- Tutorial Solution

```
function pivot(arr, start=0, end=arr.length+1){
  function swap(array, i, j) {
    var temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

  var pivot = arr[start];
  var swapIdx = start;

  for(var i = start + 1; i < arr.length; i++){
    if(pivot > arr[i]){
      swapIdx++;
      swap(arr,swapIdx,i)
    }
  }
  swap(arr,start,swapIdx)
  return swapIdx;
}
```

Quicksort Pseudocode

- Call the pivot helper on the array
- When the helper returns to you the updated pivot index, recursively call the pivot helper on the subarray to the left of that index, and the subarray to the right of that index
- Your base case occurs when you consider a subarray with less than 2 elements

```
function pivot(a, start=0, end=a.length-1) {
    while (end > start) {
        if (a[start] > a[end]) {
            start++;
            [a[start],a[end]] = [a[end], a[start]];
            [a[start-1], a[start]] = [a[start], a[start-1]];
            continue;
        }
        end--;
    }
    return start;
}
// pivot([4,8,2,1,5,7,6,3]); // 3
// pivot([9,4,8,2,1,5,7,6,3]); // 8

function quickSort(a, left=0, right=a.length-1) {
    if (left < right) {
        let start = pivot(a, left, right);
        quickSort(a, left, start-1);
        quickSort(a, start+1, right);
    }
    return a;
}

quickSort([4,8,2,1,5,7,6,3]); // [1,2,3,4,5,6,7,8]
// quickSort([9,4,8,2,1,5,7,6,3]); // [1,2,3,4,5,6,7,8,9]
```

Big O of Quicksort

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

- Best Case -> O(n log n)



- Worst Case - O(n^2)
  - Try picking any random element from the middle of array instead of picking the first element as the pivot point to improve th e worst case so that we don't spend O(n^2) for an already sorted or almost sorted array.



Radix Sort

Saturday, January 16, 2021 4:24 PM

RADIX SORT

Radix sort is a special sorting algorithm that works on lists of numbers

- It never makes comparisons between elements!

It exploits the fact that information about the size of a number is encoded in the number of digits.

More digits means a bigger number!

- As Radix sort works on a list of numbers without comparison, we use ten buckets to sort the list of numbers by checking each digits in a number and which bucket it belongs to, based on the digit in the number from right to left.

How does it work?

- 

  - We place the numbers in each bucket based on the last digit in the number

How does it work?

- 

  - Now, we form the list based on the bucket order

How does it work?

- 

- We repeat the process for the second digit to the right

How does it work?

- 

How does it work?

- 

- Repeat the process for the 3rd digit to the right

How does it work?

- 

How does it work?

[4, 7, 29, 86, 8157, 4386, 408, 1556, 3556, 593, 9637, 902]



- Repeat the process for the 4th digit to the right and we get the sorted list.

How does it work?



How does it work?

[4, 7, 29, 86, 408, 593, 902, 1556, 3556, 4386, 8157, 9637]



- Radix Sort Helper Methods

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

- getDigit(*num*, *place*) - returns the digit in *num* at the given *place* value

```
getDigit(12345, 0); // 5
getDigit(12345, 1); // 4
getDigit(12345, 2); // 3
getDigit(12345, 3); // 2
getDigit(12345, 4); // 1
getDigit(12345, 5); // 0
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

- getDigit(*num*, *place*) - returns the digit in *num* at the given *place* value

```
function getDigit(num, i) {
  return Math.floor(Math.abs(num) / Math.pow(10, i)) % 10;
}
```

```
function getDigitInPlace(num, place) {
    num = num.toString();
    return isNaN(+num[num.length-1-place]) ? 0 : +num[num.length-1-place];
}
```

-
```
getDigitInPlace(12345, 0); //5
// getDigitInPlace(12345, 1); //4
// getDigitInPlace(12345, 2); //3
// getDigitInPlace(12345, 3); //2
// getDigitInPlace(12345, 4); //1
// getDigitInPlace(12345, 5); //0
```

```
function getDigitInPlace(num, place) {
    return Math.floor(Math.abs(num) / Math.pow(10, place)) % 10;
}
```

-
```
getDigitInPlace(12345, 0); //5
// getDigitInPlace(12345, 1); //4
// getDigitInPlace(12345, 2); //3
// getDigitInPlace(12345, 3); //2
// getDigitInPlace(12345, 4); //1
// getDigitInPlace(12345, 5); //0
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

- digitCount(*num*) - returns the number of digits in *num*

```
digitCount(1); // 1
digitCount(25); // 2
digitCount(314); // 3
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

- digitCount(*num*) - returns the number of digits in *num*

```
function digitCount(num) {
  if (num === 0) return 1;
  return Math.floor(Math.log10(Math.abs(num))) + 1;
}
```

```
function getDigitCount(num) {
    return num.toString().length;
}

getDigitCount(12345); //5
// getDigitCount(1234); //4
// getDigitCount(123); //3
// getDigitCount(12); //2
// getDigitCount(1); //1
// getDigitCount(0); //1
```

```
function getDigitCount(num) {
    if (num === 0) return 1;
    return Math.floor(Math.log10(Math.abs(num))) + 1;
}

getDigitCount(12345); //5
// getDigitCount(1234); //4
// getDigitCount(123); //3
// getDigitCount(12); //2
// getDigitCount(1); //1
// getDigitCount(0); //1
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

- mostDigits(*nums*) - Given an array of numbers, returns the number of digits in the largest numbers in the list

```
mostDigits([1234, 56, 7]); // 4
mostDigits([1, 1, 11111, 1]); // 5
mostDigits([12, 34, 56, 78]); // 2
```

```
function mostDigitsInList(a) {
    let tempCount = 0;
    for (let i = 0; i < a.length; i++) {
        let count = getDigitCount(a[i]);
        tempCount = tempCount > count ? tempCount : count;
    }
    return tempCount;
}

mostDigitsInList([1234,56,7]);
// mostDigitsInList([1,11111,2,1]);
// mostDigitsInList([12,34,56,78]);
```

```
function mostDigitsInList(a) {
    let tempCount = 0;
    for (let i = 0; i < a.length; i++) {
        tempCount = Math.max(tempCount,getDigitCount(a[i]));
    }
    return tempCount;
}

mostDigitsInList([1234,56,7]);
// mostDigitsInList([1,11111,2,1]);
// mostDigitsInList([12,34,56,78]);
```

RADIX SORT PSEUDOCODE

- Define a function that accepts list of numbers
- Figure out how many digits the largest number has
- Loop from *k* = 0 up to this largest number of digits
- For each iteration of the loop:
  - Create buckets for each digit (0 to 9)
  - place each number in the corresponding bucket based on its *k*th digit
- Replace our existing array with values in our buckets, starting with 0 and going up to 9
- return list at the end!

```
// function getDigitInPlace(num, place) {
//     num = num.toString();
//     return isNaN(+num[num.length-1-place]) ? 0 : +num[num.length-1-place];
// }

function getDigitInPlace(num, place) {
    return Math.floor(Math.abs(num) / Math.pow(10, place)) % 10;
}

// getDigitInPlace(12345, 0); //5
// getDigitInPlace(12345, 1); //4
// getDigitInPlace(12345, 2); //3
// getDigitInPlace(12345, 3); //2
// getDigitInPlace(12345, 4); //1
// getDigitInPlace(12345, 5); //0
```

```
// function getDigitCount(num) {
//     return num.toString().length;
// }

function getDigitCount(num) {
    if (num === 0) return 1;
    return Math.floor(Math.log10(Math.abs(num))) + 1;
}

// getDigitCount(12345); //5
// getDigitCount(1234); //4
// getDigitCount(123); //3
// getDigitCount(12); //2
// getDigitCount(1); //1
// getDigitCount(0); //1
```

```
// function mostDigitsInList(a) {
//     let tempCount = 0;
//     for (let i = 0; i < a.length; i++) {
//         let count = getDigitCount(a[i]);
//         tempCount = tempCount > count ? tempCount : count;
//     }
//     return tempCount;
// }

function mostDigitsInList(a) {
    let tempCount = 0;
    for (let i = 0; i < a.length; i++) {
        tempCount = Math.max(tempCount,getDigitCount(a[i]));
    }
    return tempCount;
}

// mostDigitsInList([1234,56,7]);
// mostDigitsInList([1,11111,2,1]);
// mostDigitsInList([12,34,56,78]);
```

```
function radixSort(a) {
    const mostDigitsCount = mostDigitsInList(a);
    for (let i = 0; i < mostDigitsCount; i++) {
        // to create 10 sub arrays
        let bucketList = Array.from({length: 10}, () => []);
        for (let j = 0; j < a.length; j++) {
            const digitInPlace = getDigitInPlace(a[j], i);
            bucketList[digitInPlace].push(a[j]);
        }
        // to concatenate all individual arrays in bucket list to a single array
        a = [].concat(...bucketList);
    }
    return a;
}

radixSort([23,345,5467,12,2345,9852]); //[12, 23, 345, 2345, 5467, 9852]
```

RADIX SORT BIG O

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
O(*nk*)	O(*nk*)	O(*nk*)	O(*n* + *k*)

- *n* - length of array
- *k* - number of digits(average)

Data Structures

Monday, January 25, 2021 9:50 AM

- Binary Search Trees
  Queues Singly Linked Lists
  Undirected Unweighted Graphs
  Binary Heaps
  Directed Graphs Hash Tables
  Doubly Linked Lists Stacks

- WHAT DO THEY DO?
  Data structures are collections of values, the relationships among them, and the functions or operations that can be applied to the data

- WHY SO MANY???
  Different data structures excel at different things. Some are highly specialized, while others (like arrays) are more generally used.

WHY CARE?

The more time you spend as a developer, the more likely you'll need to use one of these data structures

You've already worked with many of them unknowingly!

And of course...INTERVIEWS

- Working with map/location data?

Use a graph!

- Need an ordered list with fast inserts/removals at the beginning and end?

Use a linked list!

- Web scraping nested HTML?

Use a tree!

Need to write a scheduler?

- Use a binary heap!

-

ES6 Classes

Monday, January 25, 2021 10:04 AM

What is a class?

- A blueprint for creating objects with pre-defined properties and methods

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax **does not** introduce a new object-oriented inheritance model to JavaScript.

Why do we need to learn this?

- We're going to implement **data structures** as **classes!**

THE SYNTAX

-
```
class Student {
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

The method to create new objects **must** be called constructor

The class keyword creates a constant, so you can not redefine it. Watch out for the syntax as well!

Creating objects from classes

We use the new keyword

```
class Student {
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

let firstStudent = new Student("Colt", "Steele");
let secondStudent = new Student("Blue", "Steele");
```

```
class Student {
    constructor(firstName, lastName, year){
        this.firstName = firstName;
        this.lastName = lastName;
        this.grade = year;
    }
}

let firstStudent = new Student("Colt", "Steele");
let secondStudent = new Student("Blue", "Steele");
```

- this refers to the individual instance of the object instantiated from the class using new keyword

Instance Methods

```
class Student {
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
    fullName(){
        return `Your full name is ${this.firstName} ${this.lastName}`;
    }
}

let firstStudent = new Student("Colt", "Steele");

firstStudent.fullName() // "Colt Steele"
```

```
class Student {
    constructor(firstName, lastName, year){
        this.firstName = firstName;
        this.lastName = lastName;
        this.grade = year;
        this.tardies = 0;
    }
    fullName(){
        return `Your full name is ${this.firstName} ${this.lastName}`;
    }
    markLate(){
        this.tardies += 1;
        return `${this.firstName} ${this.lastName} has been late ${this.tardies} times`;
    }

}

let firstStudent = new Student("Colt", "Steele");
let secondStudent = new Student("Blue", "Steele");
```

- firstStudent.markLate(); // Instance method

Class Methods

```
class Student {
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    fullName(){
        return `Your full name is ${this.firstName} ${this.lastName}`;
    }

    static enrollStudents(...students){
        // maybe send an email here
    }
}

let firstStudent = new Student("Colt", "Steele");
let secondStudent = new Student("Blue", "Steele");

Student.enrollStudents([firstStudent, secondStudent])
```

- In the above example enrollStudents is a class method and not an instance method
- Class method cannot be called using instance

Static methods

The `static` keyword defines a static method for a class. Static methods are called without instantiating their class and **cannot** be called through a class instance. Static methods are often used to create utility functions for an application.

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;

    return Math.hypot(dx, dy);
  }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
```

```
> p2
<· ▶ Point {x: 10, y: 10}
```

- ```
  > Point.distance(p1,p2)
  <· 7.0710678118654755
  >
  ```

How we'll be using classes

```
class DataStructure(){
    constructor(){
        // what default properties should it have?
    }
    someInstanceMethod(){
        // what should each object created from this class be able to do?
    }
}
```

We will be using the **constructor** and **instance methods** quite a bit!

We will almost **never** be using **static** methods

One gotcha with `this`

Inside all of our **instance** methods and **constructor**, the keyword `this` refers to the object created from that class (also known as an **instance**)
```

Singly Linked Lists

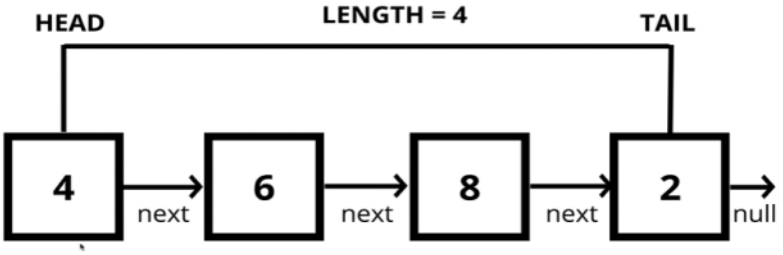
Monday, January 25, 2021 10:35 AM

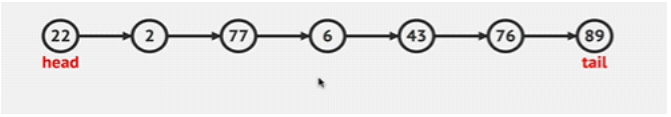
What is a linked list?

- A data structure that contains a **head**, **tail** and **length** property.

Linked Lists consist of nodes, and each **node** has a **value** and a **pointer** to another node or null

Singly Linked Lists



- Uni-directional
- 

Comparisons with Arrays

Lists

- Do not have indexes!
- Connected via nodes with a **next** pointer
- Random access is not allowed

Arrays

- Indexed in order!
- Insertion and deletion can be expensive
- Can quickly be accessed at a specific index

Pushing pseudocode

- This function should accept a value
- Create a new node using the value passed to the function
- If there is no head property on the list, set the head and tail to be the newly created node
- Otherwise set the next property on the tail to be the new node and set the tail property on the list to be the newly created node
- Increment the length by one

```
class Node {
    constructor(value, next = null) {
        this.value = value;
        this.next = null;
    }
}

class SinglyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
    push(val) {
        this.length++;
        let newNode = new Node(val);
        if(this.head === null) {
            this.tail = newNode;
            this.head = newNode;
            return this;
        }
        this.tail.next = newNode;
        this.tail = newNode;
        return this;
    }

}


let singlyLinkedList = new SinglyLinkedList();
singlyLinkedList.push(80);
```

Popping

Removing a **node** from the end of the Linked List!

Popping pseudocode

- If there are no nodes in the list, return undefined
- Loop through the list until you reach the tail
- Set the next property of the 2nd to last node to be null
- Set the tail to be the 2nd to last node
- Decrement the length of the list by 1
- Return the value of the node removed

```
// remove node from end of array
pop() {
    if (!this.head) return undefined;
    let current = this.head;
    let nextCurrent = this.head;
    while (nextCurrent.next) {
        current = nextCurrent;
        nextCurrent = current.next;
    }
    current.next = null;
    this.tail = current;
    this.length--;
    if (this.length === 0) {
        this.head = null;
        this.tail = null;
    }
    return nextCurrent;
}
```

Shifting

Removing a new **node** from the beginning of the Linked List!

Shifting pseudocode

- If there are no nodes, return undefined
- Store the current head property in a variable
- Set the head property to be the current head's next property
- Decrement the length by 1
- Return the value of the node removed

```
// remove node from beginning of array
shift(val) {
    if (!this.head) return undefined;
    let head = this.head;
    this.head = this.head.next;
    this.length--;
    if (!this.head) this.tail = null;
    return head;
}
```

Unshifting

Adding a new **node** to the beginning of the Linked List!

Unshifting pseudocode

- This function should accept a value
- Create a new node using the value passed to the function
- If there is no head property on the list, set the head and tail to be the newly created node
- Otherwise set the newly created node's next property to be the current head property on the list
- Set the head property on the list to be that newly created node
- Increment the length of the list by 1
- Return the linked list

```
// add node to beginning of array
unshift(val) {
    let newNode = new Node(val);
    this.length++;
    if (!this.head) {
        this.head = newNode;
        this.tail = this.head;
        return this;
    }
    let head = this.head;
    this.head = newNode;
    this.head.next = head;
    return this;
}
```

Get pseudocode

- This function should accept an index
- If the index is less than zero or greater than or equal to the length of the list, return null
- Loop through the list until you reach the index and return the node at that specific index

```
// get the value at index in list
get(index) {
    if (index < 0 || index >= this.length) return null;
    let count = 0;
    let current = this.head;
    while (count !== index) {
        current = current.next;
        count++;
    }
    return current;
}
```

Set

Changing the **value** of a node based on it's position in the Linked List

Set pseudocode

- This function should accept a value and an index
- Use your **get** function to find the specific node.
- If the node is not found, return false
- If the node is found, set the value of that node to be the value passed to the function and return true

```
// get the value at index in list
get(index) {
    if (index < 0 || index >= this.length) return null;
    let count = 0;
    let current = this.head;
    while (count !== index) {
        current = current.next;
        count++;
    }
    return current;
}
// set the value at index in list
set(index, value) {
    let current = this.get(index);
    if (!current) return false;
    current.value = value;
    return true;
}
```

Insert

Adding a node to the Linked List at a **specific** position

Insert pseudocode

- If the index is less than zero or greater than the length, return false
- If the index is the same as the length, push a new node to the end of the list
- If the index is 0, unshift a new node to the start of the list
- Otherwise, using the **get** method, access the node at the index - 1
- Set the next property on that node to be the new node
- Set the next property on the new node to be the previous next
- Increment the length
- Return true

```
// insert the value at index in list
insert(index, val) {
    if (index < 0 || index > this.length) return false;
    if (index === 0) return !!this.unshift(val);
    if (index === this.length) return !!this.push(val);
    let current = this.get(index-1);
    let nextCurrent = current.next;
    let newNode = new Node(val);
    current.next = newNode;
    newNode.next = nextCurrent;
    this.length++;
    return true;
}
```

Remove

- Removing a node from the Linked List at a **specific** position

Remove pseudocode

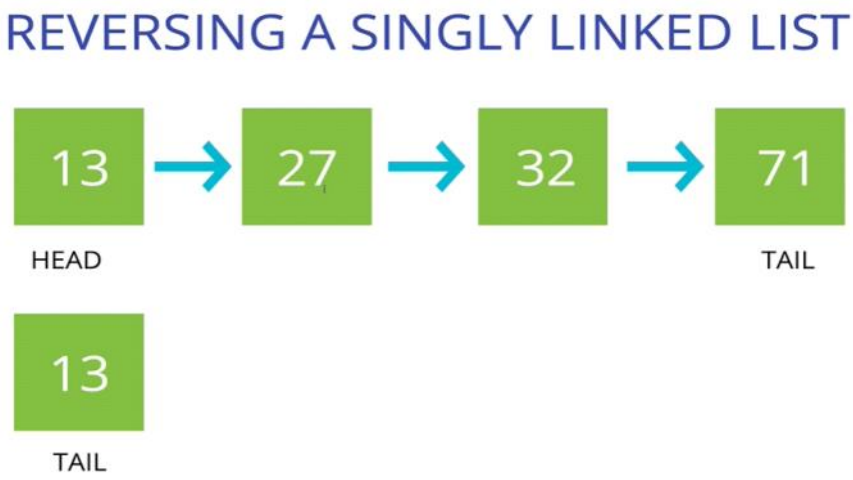
- If the index is less than zero or greater than the length, return undefined
- If the index is the same as the length-1, pop
- If the index is 0, shift
- Otherwise, using the **get** method, access the node at the index - 1
- Set the next property on that node to be the next of the next node
- Decrement the length
- Return the value of the node removed

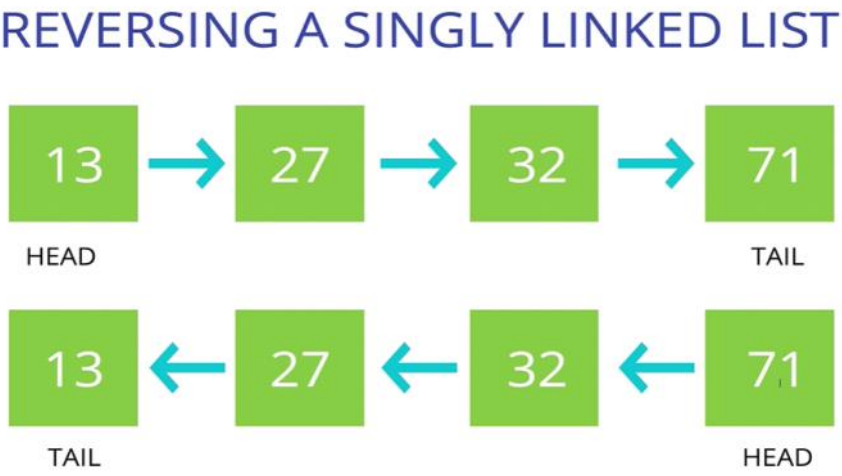
```
// remove value at index in list
remove(index) {
    if (index < 0 || index > this.length) return undefined;
    if (index === 0) return this.shift();
    if (index === this.length) return this.pop();
    let current = this.get(index-1);
    let nextCurrent = current.next;
    current.next = nextCurrent.next;
    this.length--;
    return nextCurrent;
}
```

REVERSE

- Reversing the Linked List **in place!**

- 
REVERSING A SINGLY LINKED LIST
13
27
32
71
HEAD
TAIL

- 
REVERSING A SINGLY LINKED LIST
13
27
32
71
HEAD
TAIL
13
TAIL

- 
REVERSING A SINGLY LINKED LIST
13
27
32
71
HEAD
TAIL
13
27
32
71
TAIL
HEAD

Reverse pseudocode

- Swap the head and tail
- Create a variable called next
- Create a variable called prev
- Create a variable called node and initialize it to the head property
- Loop through the list
- Set next to be the next property on whatever node is
- Set the next property on the node to be whatever prev is
- Set prev to be the value of the node variable
- Set the node variable to be the value of the next variable

- Inversed prev and next for my understanding

```
//reverse list
reverse() {
    let head = this.head;
    this.head = this.tail;
    this.tail = head;
    let prevNode;
    let nextNode = null;
    for (let i = 0; i < this.length; i++) {
        prevNode = head.next;
        head.next = nextNode;
        nextNode = head;
        head = prevNode;
    }
}
```

Big O
of Singly Linked Lists

Insertion - **O(1)**

Removal - **It depends.... O(1) or O(N)**

Searching - **O(N)**

Access - **O(N)**

RECAP

- Singly Linked Lists are an excellent alternative to arrays when insertion and deletion at the beginning are frequently required
- Arrays contain a built in index whereas Linked Lists do not
- The idea of a list data structure that consists of nodes is the foundation for other data structures like Stacks and Queues

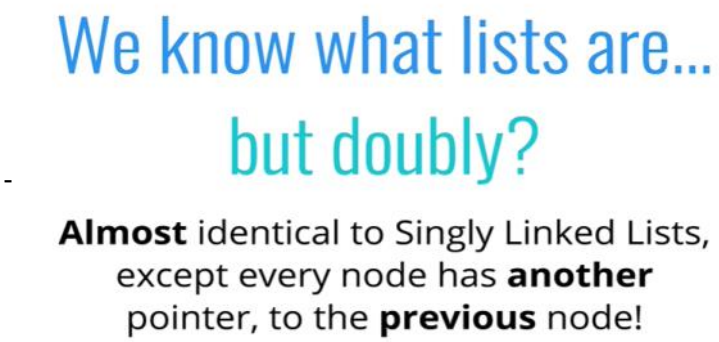
```
class Node {
    constructor(value, next = null) {
        this.value = value;
        this.next = null;
    }
}

class SinglyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
    // traverse through the list
    traverse() {
        let current = this.head;
        while (current) {
            current = current.next;
        }
    }
    // add new node to end of list
    push(val) {
        this.length++;
        let newNode = new Node(val);
        if(!this.head) {
            this.head = newNode;
            this.tail = this.head;
            return this;
        }
        this.tail.next = newNode;
        this.tail = newNode;
        return this;
    }
```

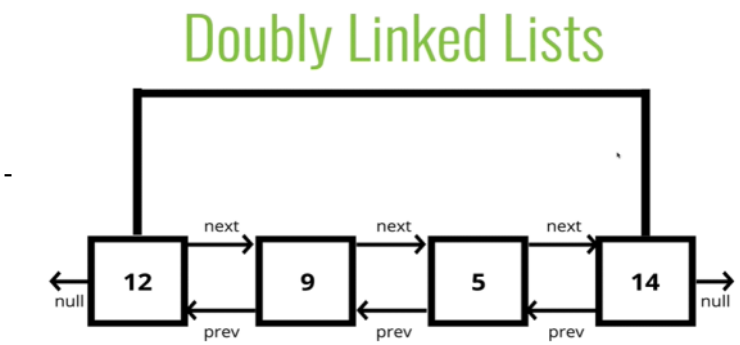
```
    // remove node from end of list
    pop() {
        if (!this.head) return undefined;
        let current = this.head;
        let nextCurrent = this.head;
        while (nextCurrent.next) {
            current = nextCurrent;
            nextCurrent = current.next;
        }
        current.next = null;
        this.tail = current;
        this.length--;
        if (this.length === 0) {
            this.head = null;
            this.tail = null;
        }
        return nextCurrent;
    }
    // add node to beginning of list
    unshift(val) {
        let newNode = new Node(val);
        this.length++;
        if (!this.head) {
            this.head = newNode;
            this.tail = this.head;
            return this;
        }
        let head = this.head;
        this.head = newNode;
        this.head.next = head;
        return this;
    }

    // remove node from beginning of list
    shift(val) {
        if (!this.head) return undefined;
        let head = this.head;
        this.head = this.head.next;
        this.length--;
        if (!this.head) this.tail = null;
        return head;
    }
    // get the value at index in list
    get(index) {
        if (index < 0 || index >= this.length) return null;
        let count = 0;
        let current = this.head;
        while (count !== index) {
            current = current.next;
            count++;
        }
        return current;
    }
    // set the value at index in list
    set(index, value) {
        let current = this.get(index);
        if (!current) return false;
        current.value = value;
        return true;
    }
```

```
    // insert the value at index in list
    insert(index, val) {
        if (index < 0 || index > this.length) return false;
        if (index === 0) return !!this.unshift(val);
        if (index === this.length) return !!this.push(val);
        let current = this.get(index-1);
        let nextCurrent = current.next;
        let newNode = new Node(val);
        current.next = newNode;
        newNode.next = nextCurrent;
        this.length++;
        return true;
    }
    // remove value at index in list
    remove(index) {
        if (index < 0 || index > this.length) return undefined;
        if (index === 0) return this.shift();
        if (index === this.length) return this.pop();
        let current = this.get(index-1);
-       let nextCurrent = current.next;
        current.next = nextCurrent.next;
        this.length--;
        return nextCurrent;
    }
    //reverse list
    reverse() {
        let head = this.head;
        this.head = this.tail;
        this.tail = head;
        let prevNode;
        let nextNode = null;
        for (let i = 0; i < this.length; i++) {
            prevNode = head.next;
            head.next = nextNode;
            nextNode = head;
            head = prevNode;
        }
    }
}

let singlyLinkedList = new SinglyLinkedList();
singlyLinkedList.push(80);
singlyLinkedList.push(81);
singlyLinkedList.push(82);
singlyLinkedList.push(83);
console.log(singlyLinkedList);
singlyLinkedList.pop();
console.log(singlyLinkedList);
singlyLinkedList.shift();
console.log(singlyLinkedList);
singlyLinkedList.unshift(79);
console.log(singlyLinkedList);
singlyLinkedList.get(2);
-singlyLinkedList.set(2, 83);
console.log(singlyLinkedList);
singlyLinkedList.insert(2, 82);
console.log(singlyLinkedList);
singlyLinkedList.insert(1, 80);
console.log(singlyLinkedList);
singlyLinkedList.insert(5, 84);
console.log(singlyLinkedList);
singlyLinkedList.remove(1);
console.log(singlyLinkedList);
singlyLinkedList.remove(5);
console.log(singlyLinkedList);
singlyLinkedList.reverse();
console.log(singlyLinkedList);
```

Doubly Linked Lists

Wednesday, January 27, 2021 12:28 PM

- 

- 

- We need not iterate through the list when we need to remove an item from the end of list.
- Dis-advantage: Takes up more memory

- 

```
class Node {
    constructor(val) {
        this.value = val;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
}
```

PUSHING

Adding a node to the **end** of the Doubly Linked List

Pushing pseudocode

- Create a new node with the value passed to the function
- If the head property is null set the head and tail to be the newly created node
- If not, set the next property on the tail to be that node
- Set the previous property on the newly created node to be the tail
- Set the tail to be the newly created node
- Increment the length
- Return the Doubly Linked List

```
// add value to the end on list
push(val) {
    let newNode = new Node(val);
    this.length++;
    if (!this.head) {
        this.head = newNode;
        this.tail = this.head;
        return this;
    }
    newNode.prev = this.tail;
    this.tail.next = newNode;
    this.tail = newNode;
    return this;
}
```

POPPING

Removing a node from the **end** of the Doubly Linked List

Popping pseudocode

- If there is no head, return undefined
- Store the current tail in a variable to return later
- If the length is 1, set the head and tail to be null
- Update the tail to be the previous Node.
- Set the newTail's next to null
- Decrement the length
- Return the value removed

```
// remove value from the end of the list
pop() {
    if (!this.head) return undefined;
    let currentTail = this.tail;
    if (this.length === 1) {
        this.head = null;
        this.tail = null;
    } else {
        this.tail = this.tail.prev;
        this.tail.next = null;
        currentTail.prev= null;
    }
    this.length--;
    return currentTail;
}
```

SHIFTING

Removing a node from the **beginning** of the Doubly Linked List

Shifting pseudocode

- If length is 0, return undefined
- Store the current head property in a variable (we'll call it old head)
- If the length is one
  - set the head to be null
  - set the tail to be null
- Update the head to be the next of the old head
- Set the head's prev property to null
- Set the old head's next to null
- Decrement the length
- Return old head

```
// remove value from the start of the list
shift() {
    if (!this.head) return undefined;
    let shiftedNode = this.head;
    if (this.length === 1) {
        this.head = null;
        this.tail = null;
    } else {
        this.head = shiftedNode.next;
        shiftedNode.next = null;
        this.head.prev = null;
    }
    this.length--;
    return shiftedNode;
}
```

UNSHIFTING

Adding a node to the **beginning** of the Doubly Linked List

Unshifting pseudocode

- Create a new node with the value passed to the function
- If the length is 0
  - Set the head to be the new node
  - Set the tail to be the new node
- Otherwise
  - Set the prev property on the head of the list to be the new node
  - Set the next property on the new node to be the head property
  - Update the head to be the new node
- Increment the length
- Return the list

```
// add node to the start of the list
unshift(val) {
    let newNode = new Node(val);
    this.length++;
    if (!this.head) {
        this.head = newNode;
        this.tail = this.head;
        return this;
    }
    this.head.prev = newNode;
    newNode.next = this.head;
    this.head = newNode;
    return this;
}
```

GET

Accessing a node in a Doubly Linked List by its position

Get Pseudocode

- If the index is less than 0 or greater or equal to the length, return null
- If the index is less than or equal to half the length of the list
  - Loop through the list starting from the head and loop towards the middle
  - Return the node once it is found
- If the index is greater than half the length of the list
  - Loop through the list starting from the tail and loop towards the middle
  - Return the node once it is found

```
// get node at a specific index in the list
get(index) {
    if (index < 0 || index >= this.length) return null;
    let startIndex = (Math.ceil(this.length/2) >= index) ? (this.length - 1) : 0;
    const isIndexInFirstHalf = startIndex === 0;
    let currentNode = isIndexInFirstHalf ? this.head : this.tail;
    while (index !== startIndex) {
        if (isIndexInFirstHalf) {
            startIndex++;
            currentNode = currentNode.next;
            continue;
        }
        currentNode = currentNode.prev;
        startIndex--;
    }
    return currentNode;
}
```

SET

Replacing the value of a node to the in a Doubly Linked List

Set pseudocode

- Create a variable which is the result of the **get** method at the index passed to the function
  - If the **get** method returns a valid node, set the value of that node to be the value passed to the function
  - Return true
- Otherwise, return false

```
// set value of node at the specified index in the list
set(index, val) {
    let nodeToUpdate = this.get(index);
    if (!nodeToUpdate) return false;
    nodeToUpdate.value = val;
    return true;
}
```

INSERT

Adding a node in a Doubly Linked List by a certain position

Insert pseudocode

- If the index is less than zero or greater than or equal to the length return false
- If the index is 0, **unshift**
- If the index is the same as the length, **push**
- Use the **get** method to access the index -1
- Set the next and prev properties on the correct nodes to link everything together
- Increment the length
- Return true

```
// insert new value to the list at a specified index
insert(index, val) {
    if (index < 0 || index > this.length) return false;
    if (index === 0) return !!this.unshift(val);
    if (index === this.length) return !!this.push(val);
    let newNode = new Node(val);
    let currentNodeAtIndex = this.get(index);
    let currentNodeAtPrevIndex = currentNodeAtIndex.prev;
    currentNodeAtPrevIndex.next = newNode;
    newNode.prev = currentNodeAtPrevIndex;
    newNode.next = currentNodeAtIndex;
    currentNodeAtIndex.prev = newNode;
    this.length++;
    return true;
}
```

REMOVE

Removing a node in a Doubly Linked List by a certain position

Remove pseudocode

- If the index is less than zero or greater than or equal to the length return undefined
- If the index is 0, **shift**
- If the index is the same as the length-1, **pop**
- Use the **get** method to retrieve the item to be removed
- Update the next and prev properties to remove the found node from the list
- Set next and prev to null on the found node
- Decrement the length
- Return the removed node.

```
// remove node from specified index in the list
remove(index) {
    if (index < 0 || index >= this.length) return undefined;
    if (index === 0) this.shift();
    if (index === this.length - 1) this.pop();
    let nodeToRemove = this.get(index);
    nodeToRemove.prev.next = nodeToRemove.next;
    nodeToRemove.next.prev = nodeToRemove.prev;
    nodeToRemove.next = null;
    nodeToRemove.prev = null;
    this.length--;
    return nodeToRemove;
}
```

Big O of Doubly Linked Lists

- Insertion - **O(1)**
  Removal - **O(1)**
  Searching - **O(N)**
  Access - **O(N)**

  Technically searching is **O(N / 2)**, but that's **still O(N)**

RECAP!

- Doubly Linked Lists are almost identical to Singly Linked Lists except there is an additional pointer to previous nodes
- Better than Singly Linked Lists for finding nodes and can be done in half the time!
- However, they do take up more memory considering the extra pointer

- Reversing a Doubly Linked List

```
// reverse the list
reverse() {
    let head = this.head;
    this.head = this.tail;
    this.tail = head;
    this.tail.prev = this.tail.next;
    let prevNode;
    let nextNode = null;
    for (let i = 0; i < this.length; i++) {
        prevNode = head.next;
        head.next = nextNode;
        head.prev = prevNode;
        nextNode = head;
        head = prevNode;
    }
    return this;
}
```

```
class Node {
    constructor(val) {
        this.value = val;
        this.prev = null;
        this.next = null;
    }
}

class DoublyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
    // add value to the end of the list
    push(val) {
        let newNode = new Node(val);
        this.length++;
        if (!this.head) {
            this.head = newNode;
            this.tail = this.head;
            return this;
        }
        newNode.prev = this.tail;
        this.tail.next = newNode;
        this.tail = newNode;
        return this;
    }

    // remove value from the end of the list
    pop() {
        if (!this.head) return undefined;
        let poppedNode = this.tail;
        if (this.length === 1) {
            this.head = null;
            this.tail = null;
        } else {
            this.tail = this.tail.prev;
            this.tail.next = null;
            poppedNode.prev= null;
        }
        this.length--;
        return poppedNode;
    }
    // remove value from the start of the list
    shift() {
        if (!this.head) return undefined;
        let shiftedNode = this.head;
        if (this.length === 1) {
            this.head = null;
            this.tail = null;
        } else {
            this.head = shiftedNode.next;
            shiftedNode.next = null;
            this.head.prev = null;
        }
        this.length--;
        return shiftedNode;
    }
```

```
    // add node to the start of the list
    unshift(val) {
        let newNode = new Node(val);
        this.length++;
        if (!this.head) {
            this.head = newNode;
            this.tail = this.head;
            return this;
        }
        this.head.prev = newNode;
        newNode.next = this.head;
        this.head = newNode;
        return this;
    }
    // get node at a specific index in the list
    get(index) {
        if (index < 0 || index >= this.length) return null;
        let startIndex = (Math.ceil(this.length/2) >= index) ? (this.length - 1) : 0;
        const isIndexInFirstHalf = startIndex === 0;
        let currentNode = isIndexInFirstHalf ? this.head : this.tail;
        while (index !== startIndex) {
            if (isIndexInFirstHalf) {
                startIndex++;
                currentNode = currentNode.next;
                continue;
            }
            currentNode = currentNode.prev;
            startIndex--;
        }
        return currentNode;
    }

    // set value of node at the specified index in the list
    set(index, val) {
        let nodeToUpdate = this.get(index);
        if (!nodeToUpdate) return false;
        nodeToUpdate.value = val;
        return true;
    }
    // insert new value to the list at a specified index
    insert(index, val) {
        if (index < 0 || index > this.length) return false;
        if (index === 0) return !!this.unshift(val);
        if (index === this.length) return !!this.push(val);
        let newNode = new Node(val);
        let currentNodeAtIndex = this.get(index);
        let currentNodeAtPrevIndex = currentNodeAtIndex.prev;
        currentNodeAtPrevIndex.next = newNode;
        newNode.prev = currentNodeAtPrevIndex;
        newNode.next = currentNodeAtIndex;
        currentNodeAtIndex.prev = newNode;
        this.length++;
        return true;
    }
    // remove node from specified index in the list
    remove(index) {
        if (index < 0 || index >= this.length) return undefined;
        if (index === 0) this.shift();
        if (index === this.length - 1) this.pop();
        let nodeToRemove = this.get(index);
        nodeToRemove.prev.next = nodeToRemove.next;
        nodeToRemove.next.prev = nodeToRemove.prev;
        nodeToRemove.next = null;
        nodeToRemove.prev = null;
        this.length--;
        return nodeToRemove;
    }
```

```
    // reverse the list
    reverse() {
        let head = this.head;
        this.head = this.tail;
        this.tail = head;
        this.tail.prev = this.tail.next;
        let prevNode;
        let nextNode = null;
        for (let i = 0; i < this.length; i++) {
            prevNode = head.next;
            head.next = nextNode;
            head.prev = prevNode;
            nextNode = head;
            head = prevNode;
        }
        return this;
    }
}

// let doublyLinkedList = new DoublyLinkedList();
// doublyLinkedList.push(80);
// doublyLinkedList.push(81);
// doublyLinkedList.push(82);
// doublyLinkedList.push(83);
// doublyLinkedList.push(84);
// doublyLinkedList.push(85);
// doublyLinkedList.pop();
// doublyLinkedList.shift();
// doublyLinkedList.unshift(79);
// doublyLinkedList.unshift(78);
// console.log(doublyLinkedList.get(2));
// doublyLinkedList.set(2, 80);
// doublyLinkedList.insert(3, 81);
// doublyLinkedList.remove(3);
// doublyLinkedList.reverse();
// console.log(doublyLinkedList);

let doublyLinkedList = new DoublyLinkedList();
doublyLinkedList.push(5).push(10).push(15).push(20);
console.log('length:',doublyLinkedList.length); // 4
console.log("----before reverse----");
console.log('doublyLinkedList.head.value:',doublyLinkedList.head.value);// 5
console.log('doublyLinkedList.head.next.value:',doublyLinkedList.head.next.value);// 10
console.log('doublyLinkedList.head.next.next.value:',doublyLinkedList.head.next.next.value);// 15
console.log('doublyLinkedList.head.next.next.next.value:',doublyLinkedList.head.next.next.next.value);// 20
doublyLinkedList.reverse(); // singlyLinkedList;
console.log("----after reverse----");
console.log('doublyLinkedList.head.value:',doublyLinkedList.head.value);// 20
console.log('doublyLinkedList.head.next.value:',doublyLinkedList.head.next.value);// 15
console.log('doublyLinkedList.head.next.next.value:',doublyLinkedList.head.next.next.value);// 10
console.log('doublyLinkedList.head.next.next.next.value:',doublyLinkedList.head.next.next.next.value);// 5
```

Stacks

Friday, January 29, 2021 1:59 PM

WHAT IS A STACK?

- A **LIFO** data structure!

  The last element added to the stack will be the first element removed from the stack

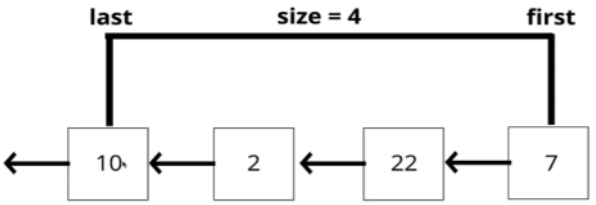
- **L**ast **I**n **F**irst **O**ut (**LIFO**)

HOW IS IT USED?

- Think about a stack of **plates**, or a stack of **markers**, or a stack of....**anything**.
  As you pile it up the last thing (or the topmost thing) is what gets removed first.

How we'll visualize a stack

- A series of nodes!



Let's see this in action!

WE'VE SEEN THIS BEFORE

- **The Call Stack!**

WHERE STACKS ARE USED

- Managing function invocations
- Undo / Redo
- Routing (the history object) is treated like a stack!

THERE IS MORE THAN ONE WAY OF IMPLEMENTING A STACK

ARRAY IMPLEMENTATION

JavaScript Does Not. I forgot to say that part!

- Javascript does not have stack data type while some languages do have.
- Using push and pop / unshift and shift we can get the values LIFO
- push and pop is the efficient way of using STACK compared to unshift and shift as we need to re-index all the list items when we add / remove from the beginning.

LINKED LIST IMPLEMENTATION

A STACK CLASS

```
class Stack {
    constructor(){
        this.first = null;
        this.last = null;
        this.size = 0;
    }
}
```

```
class Node {
    constructor(value){
        this.value = value;
        this.next = null;
    }
}
```

```
class Node {
    constructor(val) {
        this.value = val;
        this.next = null;
    }
}
```

```
class Stack {
    constructor() {
        this.first = null;
        this.last = null;
        this.size = 0;
    }
}
```

PUSHING PSEUDOCODE

- The function should accept a value
- Create a new node with that value
- If there are no nodes in the stack, set the first and last property to be the newly created node
- If there is at least one node, create a variable that stores the current first property on the stack
- Reset the first property to be the newly created node
- Set the next property on the node to be the previously created variable
- Increment the size of the stack by 1

```
// In Singly Linked List if we are adding from the beginning it will take constant time O(1)
// As we need to loop through whole list if we add it to the end which takes order O(n)g
// We use unshift implementation of Singly Linked List for push implementation of Stack
push(val) {
    let newNode = new Node(val);
    this.size++;
    if (!this.first) {
        this.first = newNode;
        this.last = newNode;
        return this;
    }
    let oldFirst = this.first;
    this.first = newNode;
    newNode.next = oldFirst;
    return this.size;
}
```

- In Singly Linked List if we adding from the beginning it will take constant time as we need to loop through whole list if we add it to the end.

POP PSEUDOCODE

- If there are no nodes in the stack, return null
- Create a temporary variable to store the first property on the stack
- If there is only 1 node, set the first and last property to be null
- If there is more than one node, set the first property to be the next property on the current first
- Decrement the size by 1
- Return the value of the node removed

```
// In Singly Linked List if we are removing from the beginning it will take constant time O(1)
// As we need to loop through whole list if we remove it from the end which takes order O(n)
// We use shift implementation of Singly Linked List for pop implementation of Stack
pop() {
    if (!this.first) return undefined;
    let nodeToBeRemoved = this.first;
    if (this.size === 1) {
        this.first = null;
        this.last = null;
    }
    this.first = this.first.next;
    this.size--;
    nodeToBeRemoved.next = null;
    return nodeToBeRemoved.value;
}
```

BIG O of STACKS

Insertion - **O(1)**

Removal - **O(1)**

Searching - **O(N)**

Access - **O(N)**

RECAP

- Stacks are a **LIFO** data structure where the last value in is always the first one out.
- Stacks are used to handle function invocations (the call stack), for operations like undo/redo, and for routing (remember pages you have visited and go back/forward) and much more!
- They are not a built in data structure in JavaScript, but are relatively simple to implement

```
class Node {
    constructor(val) {
        this.value = val;
        this.next = null;
    }
}

class Stack {
    constructor() {
        this.first = null;
        this.last = null;
        this.size = 0;
    }
```

```
    // In Singly Linked List if we are adding from the beginning it will take constant time O(1)
    // As we need to loop through whole list if we add it to the end which takes order O(n)g
    // We use unshift implementation of Singly Linked List for push implementation of Stack
    push(val) {
        let newNode = new Node(val);
        this.size++;
        if (!this.first) {
            this.first = newNode;
            this.last = newNode;
            return this;
        }
        let oldFirst = this.first;
        this.first = newNode;
        newNode.next = oldFirst;
        return this.size;
    }
    // In Singly Linked List if we are removing from the beginning it will take constant time O(1)
    // As we need to loop through whole list if we remove it from the end which takes order O(n)
    // We use shift implementation of Singly Linked List for pop implementation of Stack
    pop() {
        if (!this.first) return undefined;
        let nodeToBeRemoved = this.first;
        if (this.size === 1) {
            this.first = null;
            this.last = null;
        }
        this.first = this.first.next;
        this.size--;
        nodeToBeRemoved.next = null;
        return nodeToBeRemoved.value;
    }
}


let stack = new Stack();
stack.push(10);
stack.push(20);
stack.push(30);
console.log(stack);
stack.pop();
console.log(stack);
```

Queues

Friday, January 29, 2021 3:38 PM

WHAT IS A QUEUE?

- A **FIFO** data structure!

First In First Out

WE'VE SEEN THIS BEFORE

- Queues exist everywhere! Think about the last time you waited in line....

How do we use them in programming?

- Background tasks
- Uploading resources
- Printing / Task processing

- When implementing queue with an array
  - If we use push to add values, we need to use shift to remove values to follow FIFO rule
  - If we use unshift to add values, we need to use pop to remove values to follow FIFO rule
  - Since adding or removing from beginning of array requires re-indexing of all the list items, using array for implementing Queue is not so effective.

- When using Singly Linked List to implement a queue
  - We add to the end of array and remove from the beginning of list

A Queue Class

```
class Queue {
    constructor(){
        this.first = null;
        this.last = null;
        this.size = 0;
    }
}
```

```
class Node {
    constructor(value){
        this.value = value;
        this.next = null;
    }
}
```

```
class Node {
    constructor(val) {
        this.next = null;
        this.value = val;
    }
}
class Queue {
    constructor() {
        this.first = null;
        this.last = null;
        this.size = 0;
    }
}
```

Enqueue Pseudocode

- This function accepts some value
- Create a new node using that value passed to the function
- If there are no nodes in the queue, set this node to be the first and last property of the queue
- Otherwise, set the next property on the current last to be that node, and then set the last property of the queue to be that node
- Increment the size of the queue by 1

```
// enqueue follows push implmentation of Singly Linked List
enqueue(val) {
    let newNode = new Node(val);
    if (!this.first) {
        this.first = newNode;
        this.last = newNode;
    } else {
        this.last.next = newNode;
        this.last = newNode;
    }
    return ++this.size;
}
```

Dequeue pseudocode

- If there is no first property, just return null
- Store the first property in a variable
- See if the first is the same as the last (check if there is only 1 node). If so, set the first and last to be null
- If there is more than 1 node, set the first property to be the next property of first
- Decrement the size by 1
- Return the value of the node dequeued

```
// dequeue follows shift implmentation of Singly Linked List
dequeue() {
    if (this.size === 0) return undefined;
    let nodeToBeRemoved = this.first;
    if (this.size === 1) {
        this.first = null;
        this.last = null;
    }
    this.first = nodeToBeRemoved.next;
    nodeToBeRemoved.next = null;
    this.size--;
    return nodeToBeRemoved.value;
}
```

```
class Node {
    constructor(val) {
        this.next = null;
        this.value = val;
    }
}
class Queue {
    constructor() {
        this.first = null;
        this.last = null;
        this.size = 0;
    }
    // enqueue follows push implmentation of Singly Linked List
    enqueue(val) {
        let newNode = new Node(val);
        if (!this.first) {
            this.first = newNode;
            this.last = newNode;
        } else {
            this.last.next = newNode;
            this.last = newNode;
        }
        return ++this.size;
    }
    // dequeue follows shift implmentation of Singly Linked List
    dequeue() {
        if (this.size === 0) return null;
        let nodeToBeRemoved = this.first;
        if (this.size === 1) {
            this.first = null;
            this.last = null;
        }
        this.first = nodeToBeRemoved.next;
        nodeToBeRemoved.next = null;
        this.size--;
        return nodeToBeRemoved.value;
    }
}

let q = new Queue();
q.enqueue(1);
q.enqueue(11);
q.enqueue(111);
q.enqueue(1111);
q.dequeue();
console.log(q);
```

BIG O of QUEUES

Insertion - **O(1)**

Removal - **O(1)**

Searching - **O(N)**

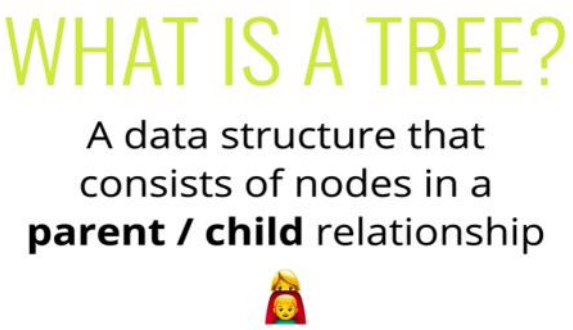
Access - **O(N)**

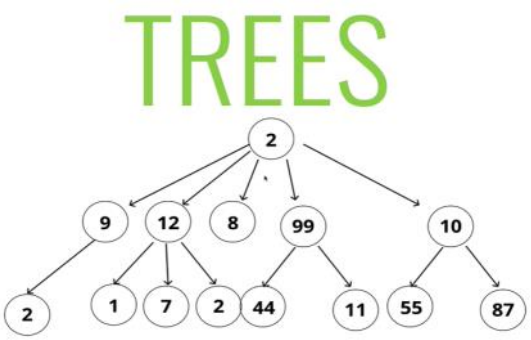
RECAP

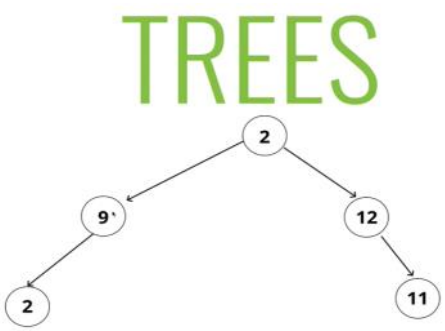
- Queues are a **FIFO** data structure, all elements are first in first out.
- Queues are useful for processing tasks and are foundational for more complex data structures
- Insertion and Removal can be done in **O(1)**

Trees

Friday, January 29, 2021 4:34 PM

- 

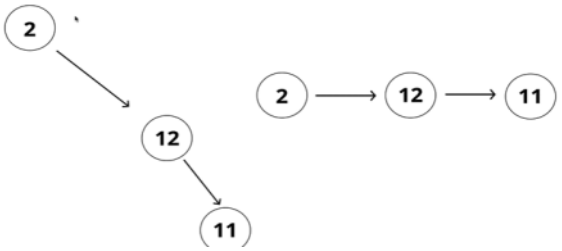
- 

- 

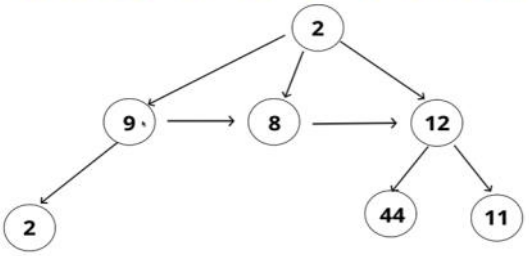
- 

A Singly Linked List

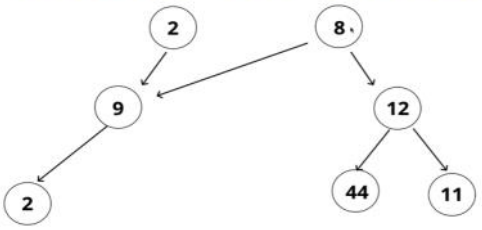
(sort of a special case of a tree)



NOT A TREE



NOT A TREE



TREE TERMINOLOGY

- **Root -** The top node in a tree.
- **Child** -A node directly connected to another node when moving away from the Root.
- **Parent** - The converse notion of a child.
- **Siblings** -A group of nodes with the same parent.
- **Leaf** - A node with no children.
- **Edge** - The connection between one node and another.

TREES

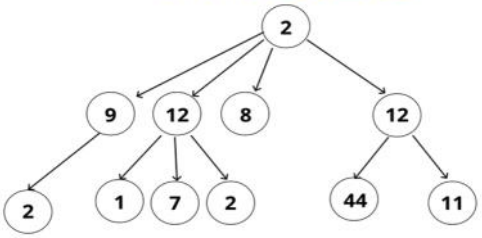
Lots of different applications!

- HTML DOM
- Network Routing
- Abstract Syntax Tree
- Artificial Intelligence
- Folders in Operating Systems
- Computer File Systems

KINDS OF TREES

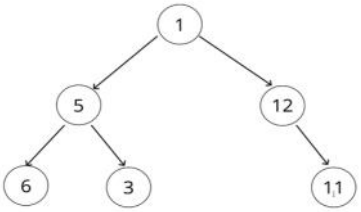
- Trees
- Binary Trees
- Binary Search Trees

TREES



- Binary Trees

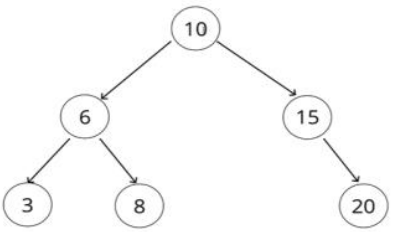
BINARY TREES



  - At most node can have only 2 children

- Binary Search Trees

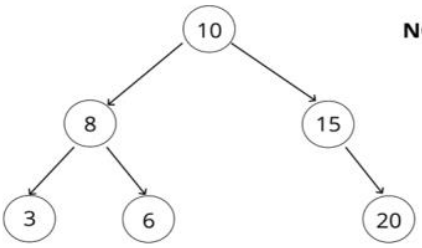
BINARY SEARCH TREES



HOW BSTS WORK

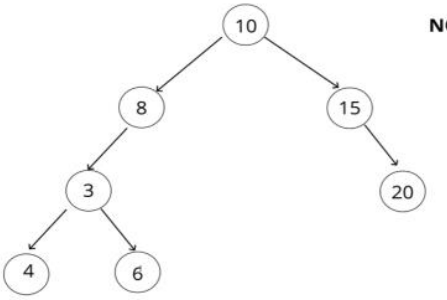
- Every parent node has at most **two** children
- Every node to the left of a parent node is **always less** than the parent
- Every node to the right of a parent node is **always greater** than the parent

Is this a valid BST?



NOPE

Is this a valid BST?



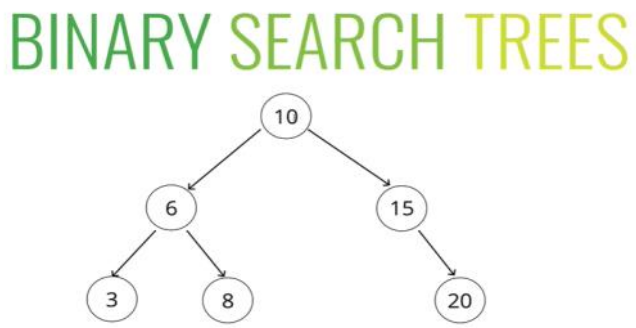
NOPE!

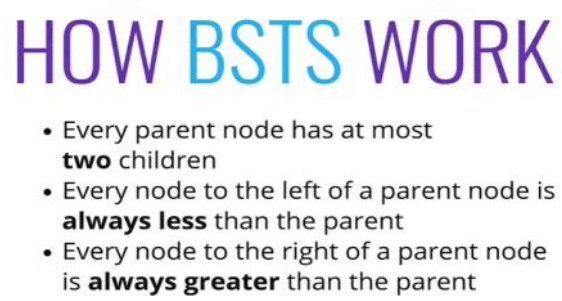
- Search part of BST is quick as data is sorted which helps us in simplifying the traversing through the tree, we need not traverse through the whole list

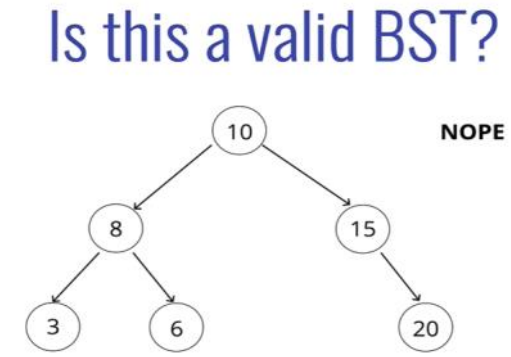
BST

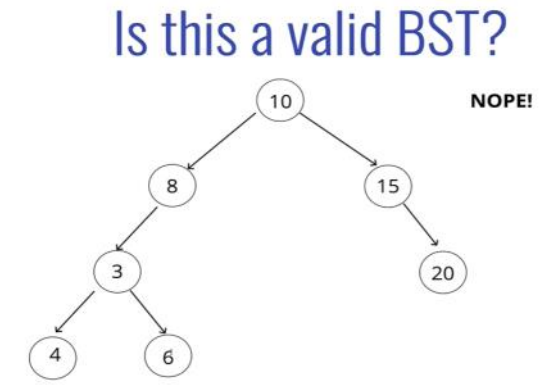
Wednesday, February 3, 2021 12:02 PM

- Binary Search Trees









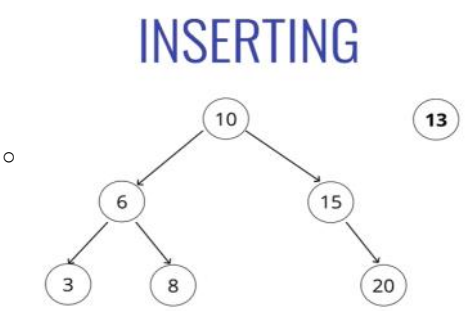
- Search part of BST is quick as data is sorted which helps us in simplifying the traversing through the tree, we need not traverse through the whole list

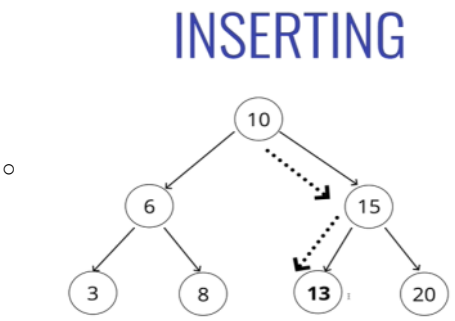
```
class Node {
    constructor(val) {
        this.value = val;
        this.left = null;
        this.right = null;
    }
}

class BinarySearchTree {
    constructor() {
        this.root = null;
    }
}
```

-

```
let bst = new BinarySearchTree();
bst.root = new Node(10);
bst.root.right = new Node(15);
bst.root.left = new Node(7);
bst.root.left.right = new Node(10);
bst.root.left.leftt = new Node(5);
```

- Inserting





INSERTING A NODE

Steps - Iteratively or Recursively

- Create a new node
- Starting at the root
  - Check if there is a root, if not - the root now becomes that new node!
  - If there is a root, check if the value of the new node is greater than or less than the value of the root
  - If it is greater
    - Check to see if there is a node to the right
      - If there is, move to that node and repeat these steps
      - If there is not, add that node as the right property
  - If it is less
    - Check to see if there is a node to the left
      - If there is, move to that node and repeat these steps
      - If there is not, add that node as the left property

```
insert(val) {
    let newNode = new Node(val);
    if (!this.root) {
        this.root = newNode;
        return this;
    }
    let currentNode = this.root;
    while (true) {
        // equal
        if (currentNode.value === val) return undefined;
        // left
        if (val < currentNode.value) {
            if (!currentNode.left) {
                currentNode.left = newNode;
                return this;
            }
            currentNode = currentNode.left;
        } else {
        //right
            if (!currentNode.right) {
                currentNode.right = newNode;
                return this;
            }
            currentNode = currentNode.right;
        }
    }
}
```

Finding a Node in a BST

Steps - Iteratively or Recursively

- Starting at the root
  - Check if there is a root, if not - we're done searching!
  - If there is a root, check if the value of the new node is the value we are looking for. If we found it, we're done!
  - If not, check to see if the value is greater than or less than the value of the root
  - If it is greater
    - Check to see if there is a node to the right
      - If there is, move to that node and repeat these steps
      - If there is not, we're done searching!
  - If it is less
    - Check to see if there is a node to the left
      - If there is, move to that node and repeat these steps
      - If there is not, we're done searching!

Let's visualize this!

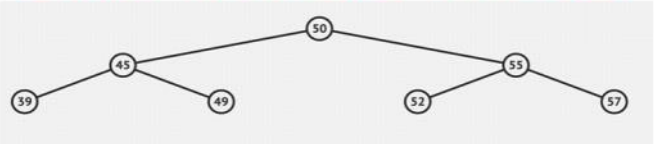
```
contains(val) {
    let currentNode = this.root;
    while (true) {
        if (!currentNode) return false;
        if (currentNode.value === val) return true;
        if (val < currentNode.value) {
            currentNode = currentNode.left;
        } else {
            currentNode = currentNode.right;
        }
    }
    return false;
}
```

-
```
find(val) {
    let currentNode = this.root;
    while (true) {
        if (!currentNode) return undefined;
        if (currentNode.value === val) return currentNode;
        if (val < currentNode.value) {
            currentNode = currentNode.left;
        } else {
            currentNode = currentNode.right;
        }
    }
    return undefined;
}
```

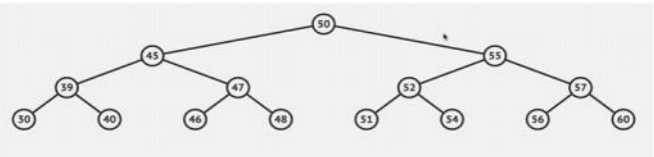
Big O of BST

- Insertion - **O(log n)**
  Searching - **O(log n)**
  NOT guaranteed!

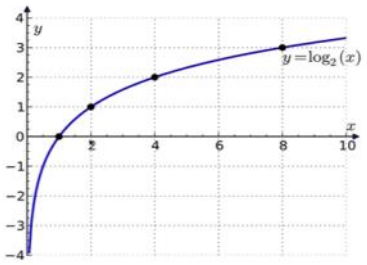
Double the number of nodes...



- You only increase the number of steps to insert/find by 1



  - As the number of nodes doubles the step increases by 1 which sums the order to O(log n)

- 

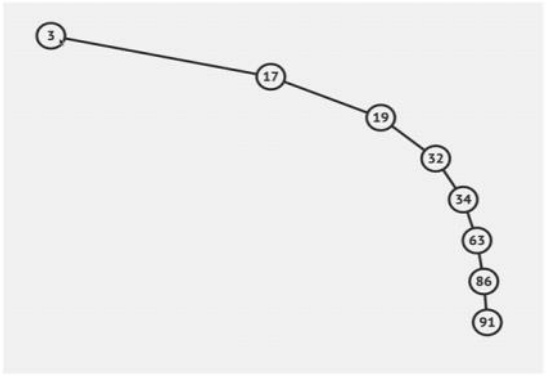
2x number of nodes: 1 extra step

4x number of nodes: 2 extra steps

8x number of nodes: 3 extra steps

- O(log n) is not guaranteed if the BST looks like a linked list as shown below

THIS IS A VALID BINARY SEARCH TREE

- 

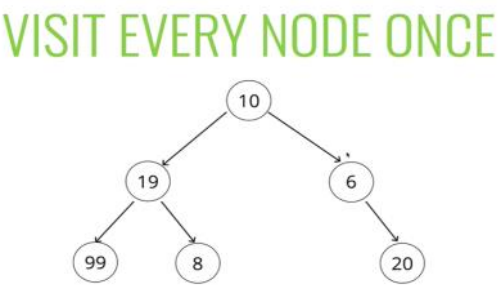

```
class Node {
    constructor(val) {
        this.value = val;
        this.left = null;
        this.right = null;
    }
}
```

```
class BinarySearchTree {
    constructor() {
        this.root = null;
    }

    insert(val) {
        let newNode = new Node(val);
        if (!this.root) {
            this.root = newNode;
            return this;
        }
        let currentNode = this.root;
        while (true) {
            // equal
            if (currentNode.value === val) return undefined;
            // left
            if (val < currentNode.value) {
                if (!currentNode.left) {
                    currentNode.left = newNode;
                    return this;
                }
                currentNode = currentNode.left;
            } else {
            //right
                if (!currentNode.right) {
                    currentNode.right = newNode;
                    return this;
                }
                currentNode = currentNode.right;
            }
        }
    }


    contains(val) {
        let currentNode = this.root;
        while (true) {
            if (!currentNode) return false;
            if (currentNode.value === val) return true;
            if (val < currentNode.value) {
                currentNode = currentNode.left;
            } else {
                currentNode = currentNode.right;
            }
        }
        return false;
    }

    find(val) {
        let currentNode = this.root;
        while (true) {
            if (!currentNode) return undefined;
            if (currentNode.value === val) return currentNode;
            if (val < currentNode.value) {
                currentNode = currentNode.left;
            } else {
                currentNode = currentNode.right;
            }
        }
        return undefined;
    }
}

let bst = new BinarySearchTree();
bst.insert(10);
bst.insert(15);
bst.insert(7);
bst.insert(5);
bst.insert(13);
bst.contains(26); // true
bst.find(13); // returns node if found or undefined
// bst.find(7);
```

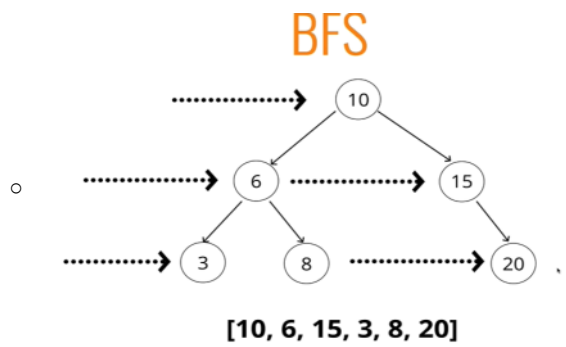
Tree Traversal

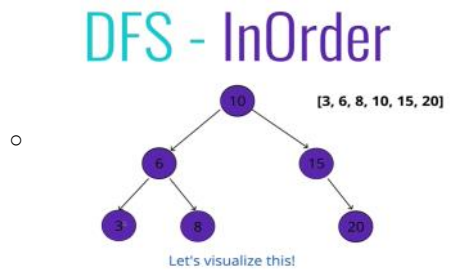
Wednesday, February 3, 2021 2:36 PM

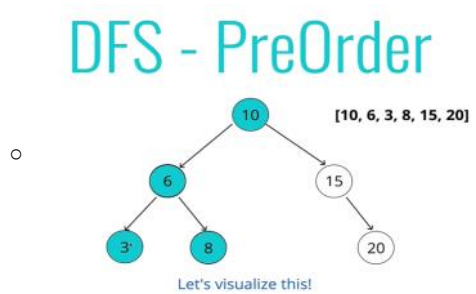
- Traverse the tree in such a way that you visit the node at least once

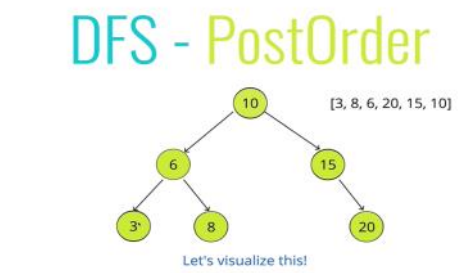
- 

- 

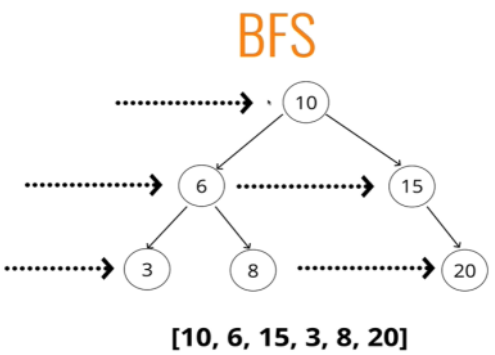
  - 

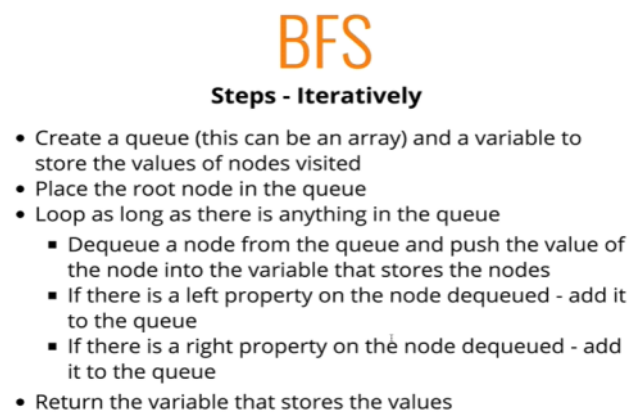
  - 

  - 

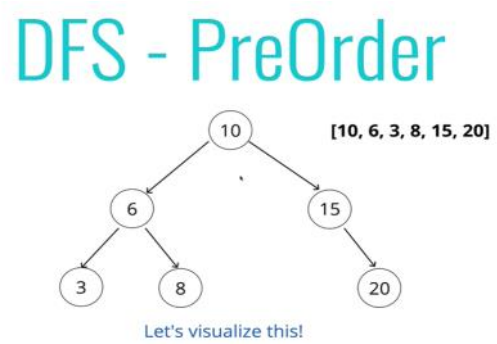
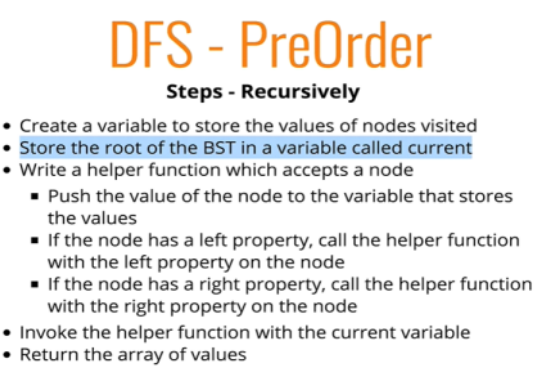
- 

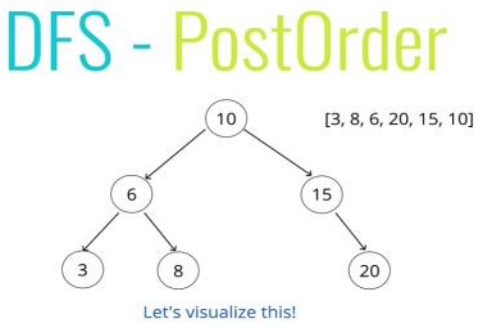
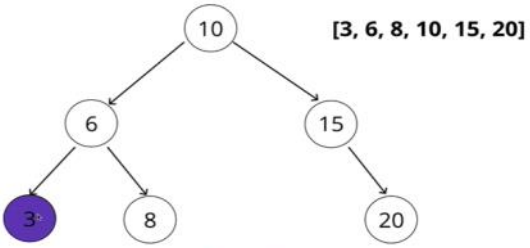
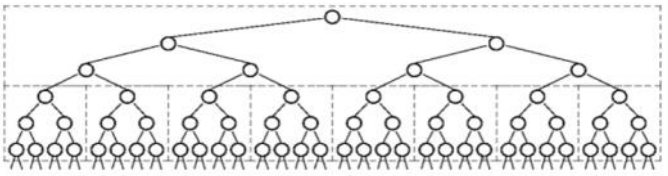
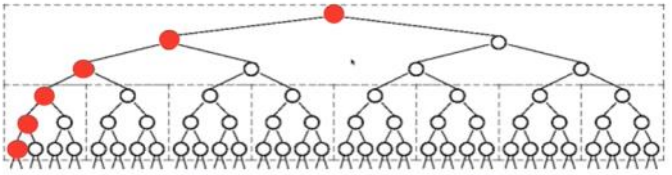
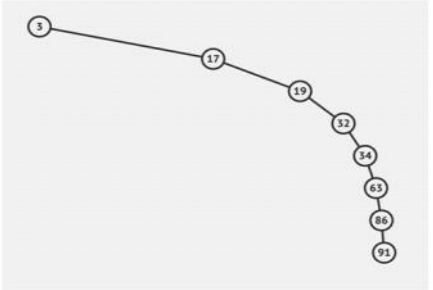
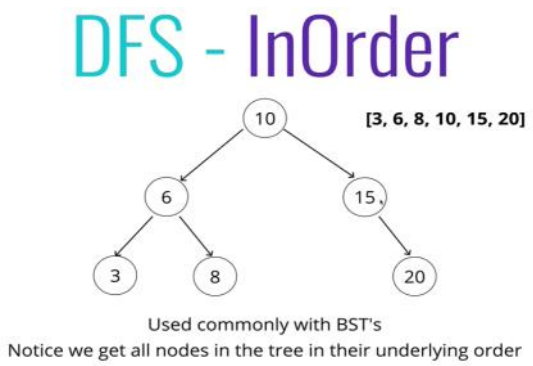
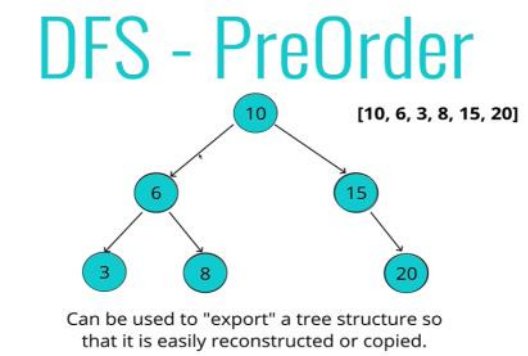
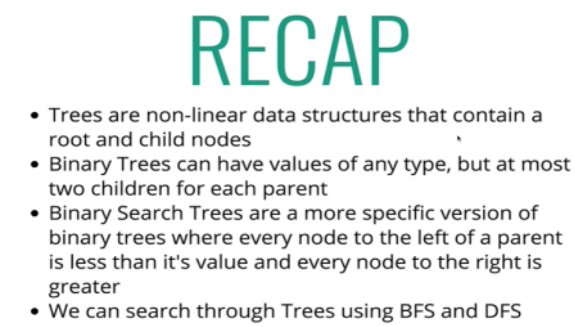
- Breadth First Search

- 

- 


- ```
  breadthFirstSearch() {
      let currentNode = this.root;
      let data = [];
      let queue = [this.root]
      while (queue.length) {
          // removes the node from the start of queue array and returns the node
          currentNode = queue.shift();
          data.push(currentNode);
          if (currentNode.left) queue.push(currentNode.left);
          if (currentNode.right) queue.push(currentNode.right);
      }
      return data;
  }
  ```

- Depth First Search
  ```

- 

- In DFS - Preorder we traverse through root node, left first and right next
- 


```
preOrderDepthFirstSearch() {
    return this.traverse(this.root);
}

traverse(node, visited = [this.root.value]) {
    if (node.left) {
        visited.push(node.left.value);
        this.traverse(node.left, visited);
    }
    if (node.right) {
        visited.push(node.right.value);
        this.traverse(node.right, visited);
    }
    return visited;
}
```

- 

- In DFS - Post-order we traverse through left first, right next and lastly root.

DFS - PostOrder

Steps - Recursively

- Create a variable to store the values of nodes visited
- Store the root of the BST in a variable called current
- Write a helper function which accepts a node
  - If the node has a left property, call the helper function with the left property on the node
  - If the node has a right property, call the helper function with the right property on the node
  - Push the value of the node to the variable that stores the values
  - Invoke the helper function with the current variable
- Return the array of values

```
postOrderDepthFirstSearch() {
    let traverse = (node, visited) => {
        if (node.left) traverse(node.left, visited);
        if (node.right) traverse(node.right, visited);
        visited.push(node.value);
        return visited;
    };
    return traverse(this.root, []);
}
```

DFS - InOrder



[3, 6, 8, 10, 15, 20]

Let's visualize this!

DFS - InOrder

Steps - Recursively

- Create a variable to store the values of nodes visited
- Store the root of the BST in a variable called current
- Write a helper function which accepts a node
  - If the node has a left property, call the helper function with the left property on the node
  - Push the value of the node to the variable that stores the values
  - If the node has a right property, call the helper function with the right property on the node
- Invoke the helper function with the current variable

```
inOrderDepthFirstSearch() {
    let traverse = (node, visited) => {
        if (node.left) traverse(node.left, visited);
        visited.push(node.value);
        if (node.right) traverse(node.right, visited);
        return visited;
    }
    return traverse(this.root, []);
}
```

- BFS?
  DFS?
  Which is better?

BREADTH FIRST



Lots of nodes to keep track of!

- We are visiting every node once so time doesn't matter but space matters
- We have to store tons of nodes in queue is the tree is deeper
- If the tree is wider, we go for BFS

DEPTH FIRST



Fewer nodes to keep track of

- We are visiting every node once so time doesn't matter but space matters
- If the tree is deeper we for DFS

BREADTH FIRST



Fewer nodes to keep track of

- 

    - InOrder will sort out the list from least to highest which will be useful.
- 

    - Can be useful if you cloning or duplicating a tree or you want to flatten it out and store in it in the db and later you can re-construct the tree when retrieved as it follows the node first, left and then right.
- 


```
class Node {
    constructor(val) {
        this.value = val;
        this.left = null;
        this.right = null;
    }
}
```

```
class BinarySearchTree {
    constructor() {
        this.root = null;
    }

    insert(val) {
        let newNode = new Node(val);
        if (!this.root) {
            this.root = newNode;
            return this;
        }
        let currentNode = this.root;
        while (true) {
            // equal
            if (currentNode.value === val) return undefined;
            // left
            if (val < currentNode.value) {
                if (!currentNode.left) {
                    currentNode.left = newNode;
                    return this;
                }
                currentNode = currentNode.left;
            } else {
            //right
                if (!currentNode.right) {
                    currentNode.right = newNode;
                    return this;
                }
                currentNode = currentNode.right;
            }
        }
    }


    contains(val) {
        let currentNode = this.root;
        while (true) {
            if (!currentNode) return false;
            if (currentNode.value === val) return true;
            if (val < currentNode.value) {
                currentNode = currentNode.left;
            } else {
                currentNode = currentNode.right;
            }
        }
        return false;
    }

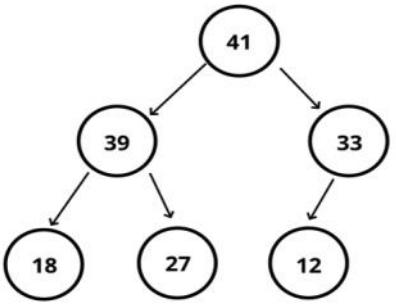
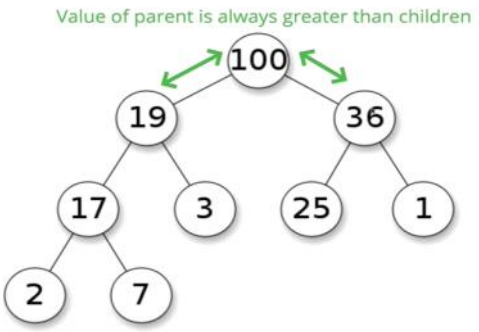
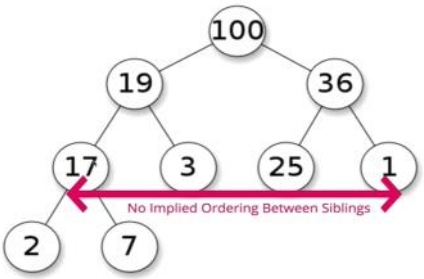
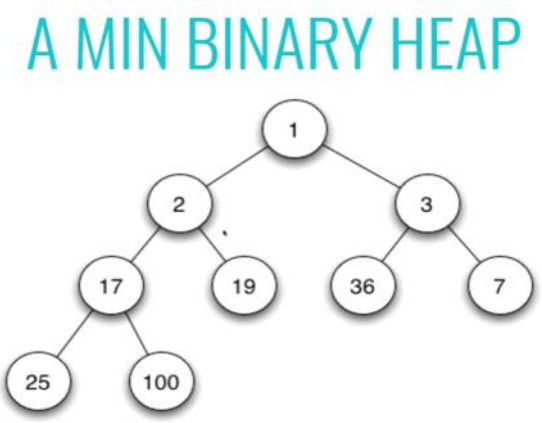
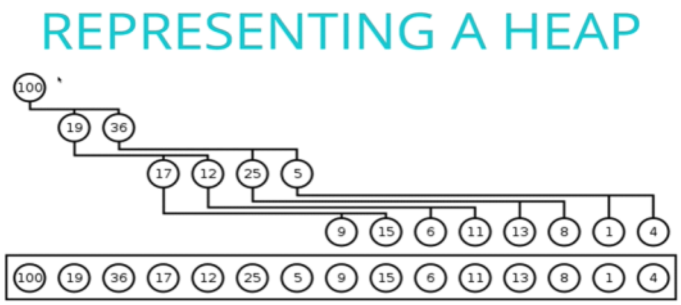
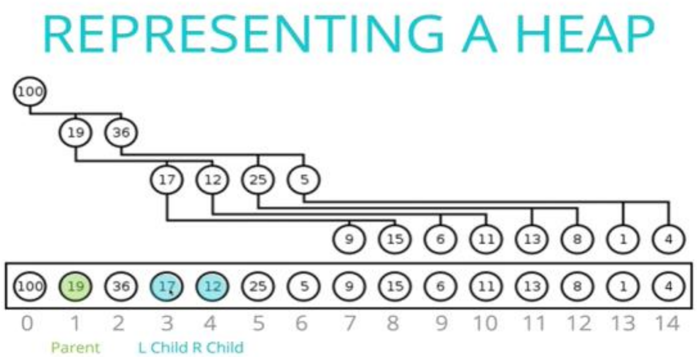
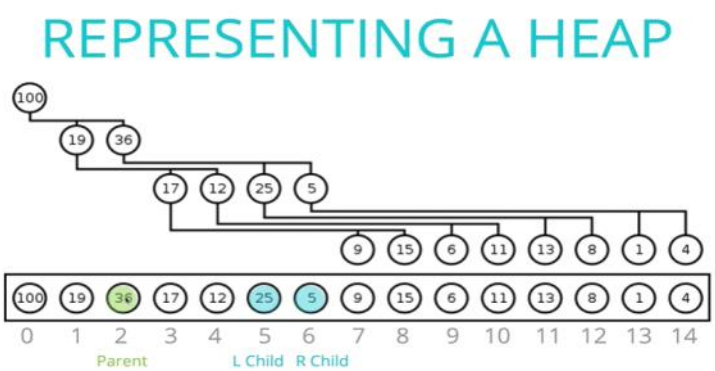
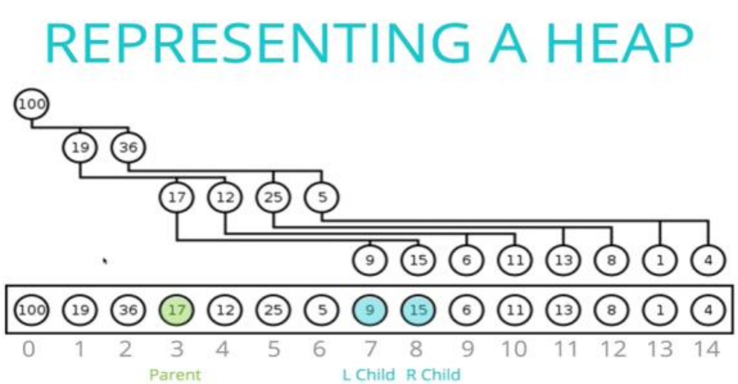
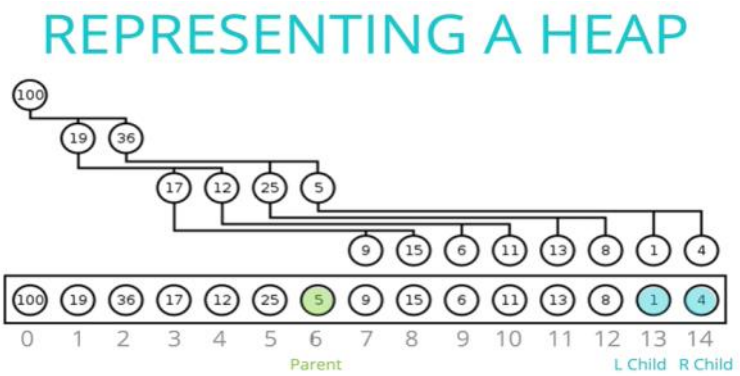
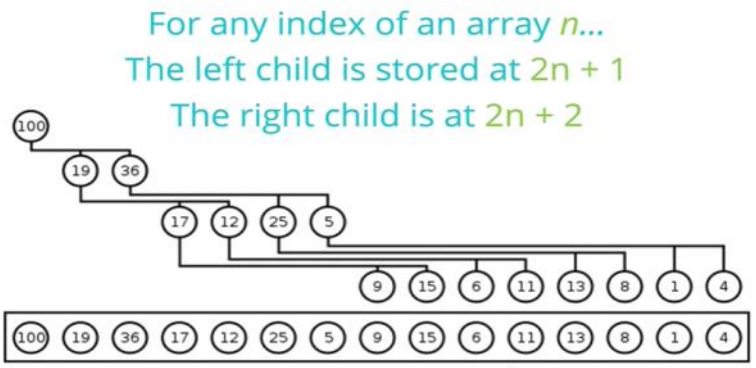
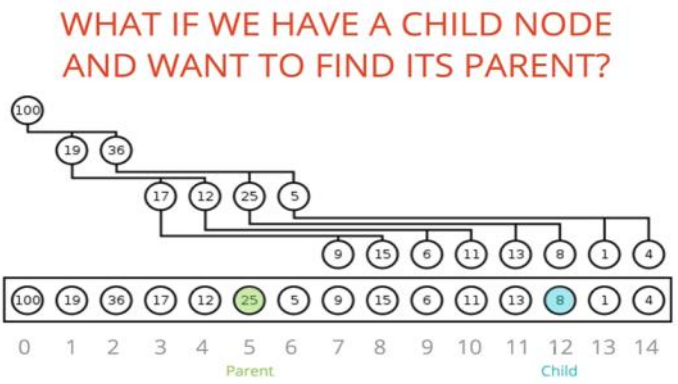
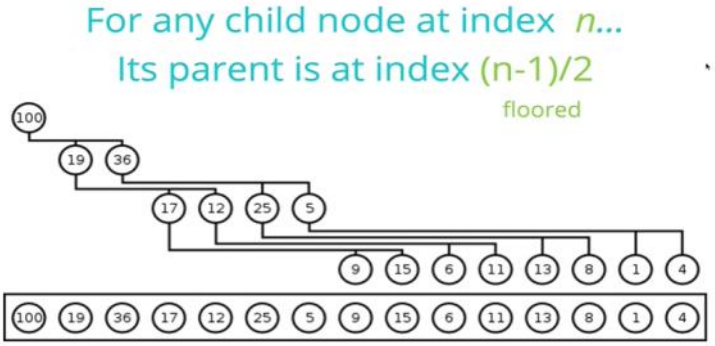
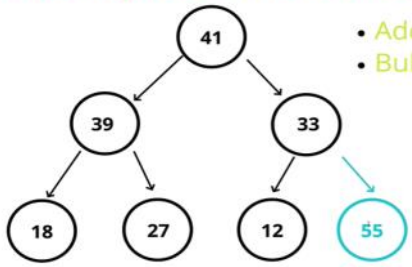
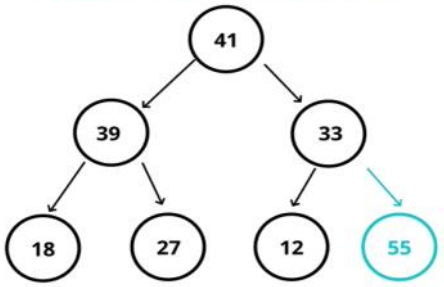
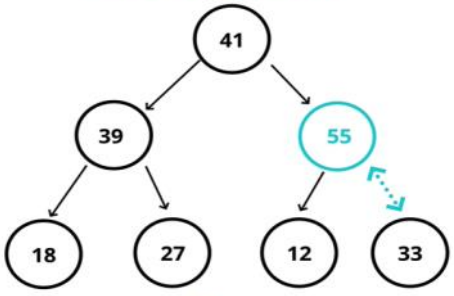
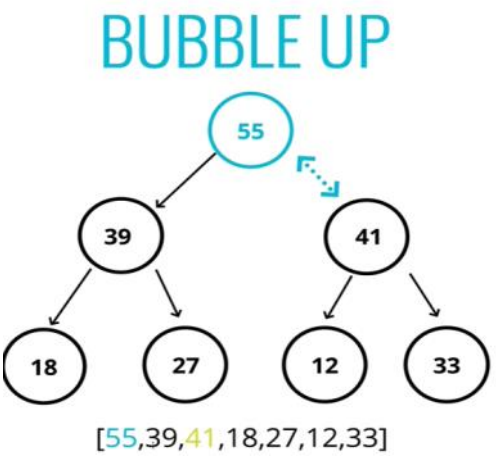
    find(val) {
        let currentNode = this.root;
        while (true) {
            if (!currentNode) return undefined;
            if (currentNode.value === val) return currentNode;
            if (val < currentNode.value) {
                currentNode = currentNode.left;
            } else {
                currentNode = currentNode.right;
            }
        }
        return undefined;
    }


    breadthFirstSearch() {
        let currentNode = this.root;
        let data = [];
        let queue = [this.root]
        while (queue.length) {
            // removes the node from the start of queue array and returns the node
            currentNode = queue.shift();
            data.push(currentNode.value);
            if (currentNode.left) queue.push(currentNode.left);
            if (currentNode.right) queue.push(currentNode.right);
        }
        return data;
    }
```

```
    preOrderDepthFirstSearch() {
        let traverse = (node, visited) => {
            visited.push(node.value);
            if (node.left) traverse(node.left, visited);
            if (node.right) traverse(node.right, visited);
            return visited;
        };
        return traverse(this.root, []);
    }

    postOrderDepthFirstSearch() {
        let traverse = (node, visited) => {
            if (node.left) traverse(node.left, visited);
            if (node.right) traverse(node.right, visited);
            visited.push(node.value);
            return visited;
        };
        return traverse(this.root, []);
    }

    inOrderDepthFirstSearch() {
        let traverse = (node, visited) => {
            if (node.left) traverse(node.left, visited);
            visited.push(node.value);
            if (node.right) traverse(node.right, visited);
            return visited;
        }
        return traverse(this.root, []);
    }

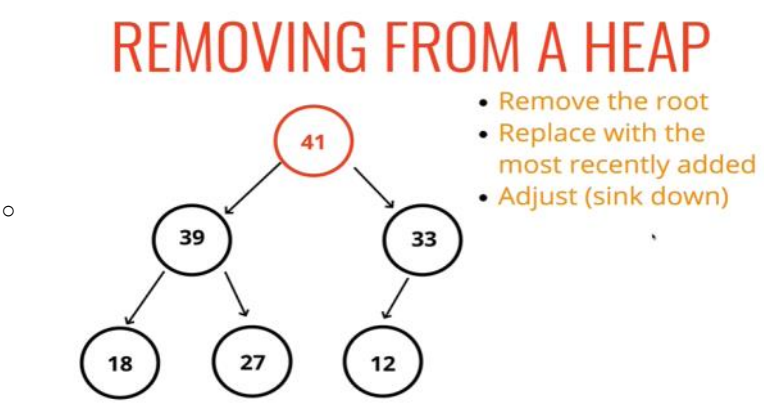
}


let bst = new BinarySearchTree();
bst.insert(10);
bst.insert(6);
bst.insert(15);
bst.insert(3);
bst.insert(8);
bst.insert(20);
bst.contains(26); // true
bst.find(13); // returns node if found or undefined
// bst.find(7);
console.log(bst.breadthFirstSearch()); // [10,6,15,3,8,20]
console.log(bst.preOrderDepthFirstSearch()); // [10,6,3,8,15,20]
console.log(bst.postOrderDepthFirstSearch()); // [3,8,6,20,15,10]
console.log(bst.inOrderDepthFirstSearch()); // [3,6,8,10,15,20]
```

Heaps

Friday, February 5, 2021 1:38 PM

- Everything that applies to trees also applies to heaps with some change in rules.

WHAT IS A BINARY HEAP?

Very similar to a binary search tree, but with some different rules!

-

In a **MaxBinaryHeap**, parent nodes are always larger than child nodes. In a **MinBinaryHeap**, parent nodes are always smaller than child nodes

WHAT DOES IT LOOK LIKE?

-



MAX BINARY HEAP

- Each parent has at most two child nodes
- The value of each parent node is **always** greater than its child nodes
- In a max Binary Heap the parent is greater than the children, but there are no guarantees between sibling nodes.
- A binary heap is as compact as possible. All the children of each node are as full as they can be and left children are filled out first

- Example of max Binary Heap
  - Always the left nodes are filled first
  - Parent node has always a greater value than child node

- 

- 

- Min Binary heaps are the opposite of Max Binary Heap, where the parent is always lesser than the child

- 

- 

-


REPRESENTING A HEAP
100
19 36
17 12 25 5
9 15 6 11 13 8 1 4
100 19 36 17 12 25 5 9 15 6 11 13 8 1 4

-


REPRESENTING A HEAP
100
19 36
17 12 25 5
9 15 6 11 13 8 1 4
100 19 36 17 12 25 5 9 15 6 11 13 8 1 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Parent
L Child R Child

-


REPRESENTING A HEAP
100
19 36
17 12 25 5
9 15 6 11 13 8 1 4
100 19 36 17 12 25 5 9 15 6 11 13 8 1 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Parent
L Child R Child

-


REPRESENTING A HEAP
100
19 36
17 12 25 5
9 15 6 11 13 8 1 4
100 19 36 17 12 25 5 9 15 6 11 13 8 1 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Parent
L Child R Child

- 

- For every parent the child nodes are at Left node: 2n+1 and Right Node: 2n+2

- 

- 

- 

- Max Binary Heap

DEFINING OUR CLASS

- Class Name:
  MaxBinaryHeap
  Properties:
  values = []

Adding to a MaxBinaryHeap

- 

- Add to the end
- Bubble up

- Add to the end of list and bubble up to the place it fits in

ADD TO THE END

- 

[41,39,33,18,27,12,55]

BUBBLE UP

- 

[41,39,31,18,27,12,55]

- 

- # INSERT PSEUDOCODE
  - Push the value into the values property on the heap
  - Bubble the value up to its correct spot!

- # INSERT PSEUDOCODE
  - Push the value into the values property on the heap
  - Bubble Up:
    - Create a variable called index which is the length of the values property - 1
    - Create a variable called parentIndex which is the floor of (index-1)/2
    - Keep looping as long as the values element at the parentIndex is less than the values element at the child index
      - Swap the value of the values element at the parentIndex with the value of the element property at the child index
      - Set the index to be the parentIndex, and start over!

- ```
  class MaxBinaryHeap {
      constructor() {
          this.values = [];
      }

      insert(val) {
          this.values.push(val);
          this.bubbleUp();
      }

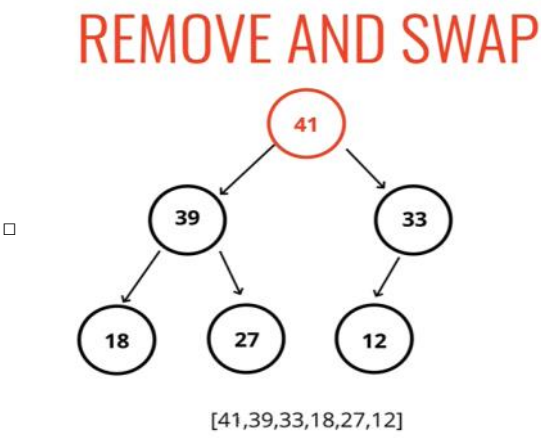
      bubbleUp() {
          let index = this.values.length - 1;
          // get parent by using formula (n-1)/2
          while (index > 0) {
              let parentIndex = Math.floor((index-1)/2);
              if (this.values[index] <= this.values[parentIndex]) break;
              // swap the values
              [this.values[parentIndex], this.values[index]] = [this.values[index], this.values[parentIndex]];
              index = parentIndex;
          }
          console.log(this.values);
          return;
      }
  }

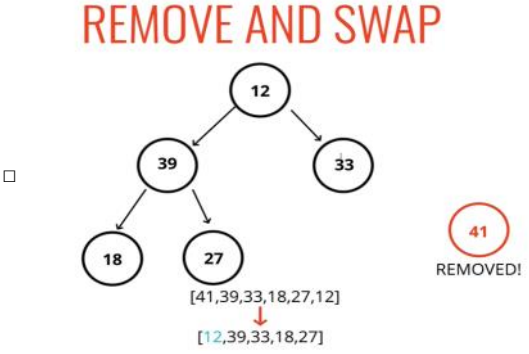
  let maxBinaryHeap = new MaxBinaryHeap();
  maxBinaryHeap.insert(41);
  maxBinaryHeap.insert(39);
  maxBinaryHeap.insert(33);
  maxBinaryHeap.insert(18);
  maxBinaryHeap.insert(27);
  maxBinaryHeap.insert(12);
  maxBinaryHeap.insert(55);
  console.log(this.values); // [55,39,41,18,27,12,33]
  ```
  ```

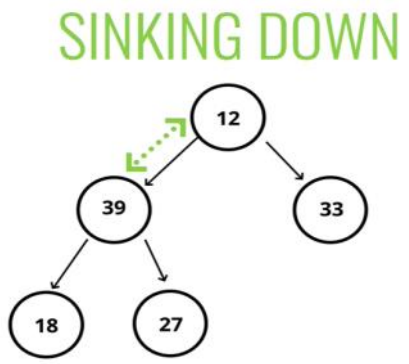
- 

SINK DOWN?

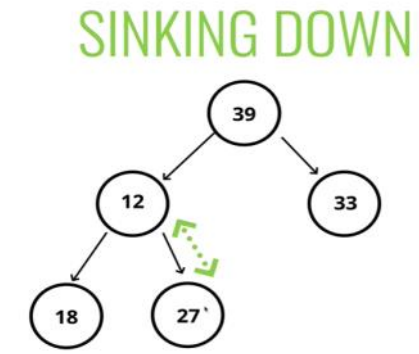
- The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down*, and *extract-min/max*).

- 

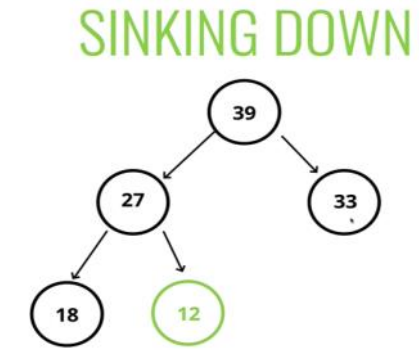
- 

- 

[12,30,33,18,27]

- 

[39,12,33,18,27]

- 

[39,27,33,18,12]

REMOVING

(also called extractMax)

- Swap the first value in the values property with the last one
- Pop from the values property, so you can return the value at the end.
- Have the new root "sink down" to the correct spot...
  - Your parent index starts at 0 (the root)
  - Find the index of the left child: 2 * index + 1 (make sure its not out of bounds)
  - Find the index of the right child: 2*index + 2 (make sure its not out of bounds)
  - If the left or right child is greater than the element...swap. If both left and right children are larger, swap with the largest child.
  - The child index you swapped to now becomes the new parent index.
  - Keep looping and swapping until neither child is larger than the element.
  - Return the old root!

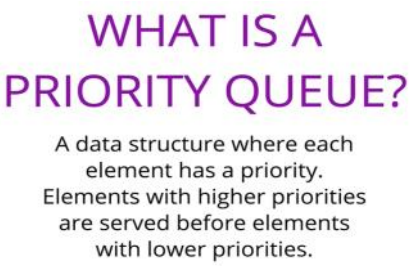
```
remove() {
    // swap first and last values
    [this.values[0], this.values[this.values.length-1]] = [this.values[this.values.length - 1], this.values[0]];
    const max = this.values.pop();
    this.rearrangeHeap();
    return max;
}

rearrangeHeap() {
    let index = 0;
    while (true) {
        let leftChildIndex = (2*index) + 1;
        let rightChildIndex = (2*index) + 2;
        let swapIndex = null;
        if (leftChildIndex < this.values.length && this.values[leftChildIndex] > this.values[index])
            swapIndex = leftChildIndex;
        if (
            (swapIndex === null &&
            (rightChildIndex < this.values.length) &&
            (this.values[rightChildIndex] > this.values[index])) ||
            (swapIndex !== null &&
            (rightChildIndex < this.values.length) && (
            this.values[rightChildIndex] > this.values[leftChildIndex]))
        ) swapIndex = rightChildIndex;
        if (swapIndex === null) break;
        [this.values[index], this.values[swapIndex]] = [this.values[swapIndex], this.values[index]];
        index = swapIndex;
    }
}
```

- Final Code for inserting and removing from a max binary heap

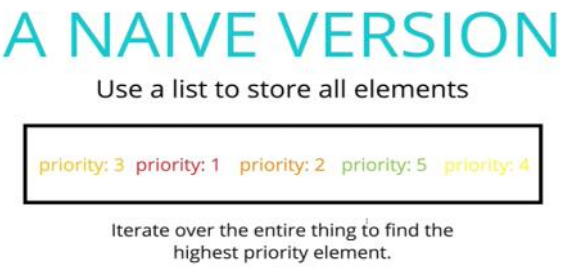
```
class MaxBinaryHeap {
    constructor() {
        this.values = [];
    }

    insert(val) {
        this.values.push(val);
        this.bubbleUp();
    }

    bubbleUp() {
        let index = this.values.length - 1;
        // get parent by using formula (n-1)/2
        while (index > 0) {
            let parentIndex = Math.floor((index-1)/2);
            if (this.values[index] <= this.values[parentIndex]) break;
            // swap the values
            [this.values[parentIndex], this.values[index]] = [this.values[index], this.values[parentIndex]];
            index = parentIndex;
        }
        console.log(this.values);
        return;
    }

    remove() {
        // swap first and last values
        [this.values[0], this.values[this.values.length-1]] = [this.values[this.values.length - 1], this.values[0]];
        const max = this.values.pop();
        this.rearrangeHeap();
        return max;
    }


    rearrangeHeap() {
        let index = 0;
        while (true) {
            let leftChildIndex = (2*index) + 1;
            let rightChildIndex = (2*index) + 2;
            let swapIndex = null;
            if (leftChildIndex < this.values.length && this.values[leftChildIndex] > this.values[index])
                swapIndex = leftChildIndex;
            if (
                (swapIndex === null &&
                (rightChildIndex < this.values.length) &&
                (this.values[rightChildIndex] > this.values[index])) ||
                (swapIndex !== null &&
                (rightChildIndex < this.values.length) && (
                this.values[rightChildIndex] > this.values[leftChildIndex]))
            ) swapIndex = rightChildIndex;
            if (swapIndex === null) break;
            [this.values[index], this.values[swapIndex]] = [this.values[swapIndex], this.values[index]];
            index = swapIndex;
        }
    }
}

let maxBinaryHeap = new MaxBinaryHeap();
maxBinaryHeap.insert(41);
maxBinaryHeap.insert(39);
maxBinaryHeap.insert(33);
maxBinaryHeap.insert(18);
maxBinaryHeap.insert(27);
maxBinaryHeap.insert(12);
maxBinaryHeap.insert(55);
console.log(this.values); // [55,39,41,18,27,12,33]
```

Priority Queue

Monday, February 22, 2021 10:31 AM

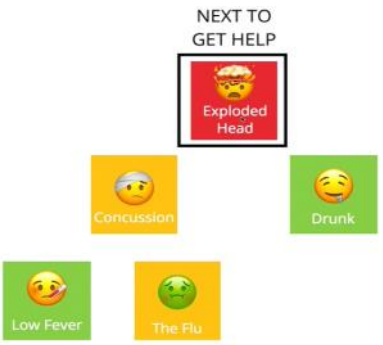


- A data structure where each element has a priority. Elements with higher priorities are served before elements with lower priorities.

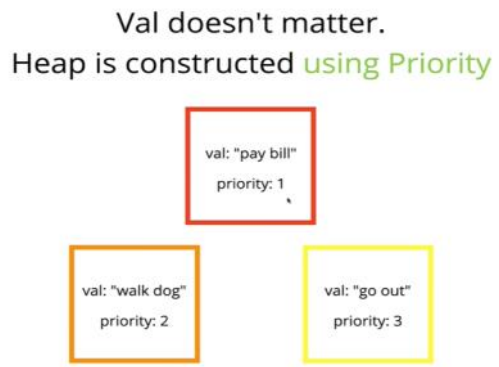


-

  - This will have an order of O(n)
- We can use heaps to implement this priority queue.
  - Example:
    -



  - Heaps has an order O(log n) for insertion and removal.
- Pseudocode
  - We can use MinBinaryHeap to implement PriorityQueue
    -



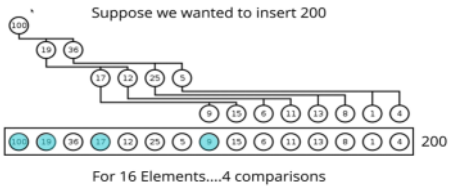
OUR PRIORITY QUEUE

- Write a Min Binary Heap - lower number means higher priority.
- Each Node has a val and a priority. Use the priority to build the heap.
- **Enqueue** method accepts a value and priority, makes a new node, and puts it in the right spot based off of its priority.
- **Dequeue** method removes root element, returns it, and rearranges heap using priority.

Big O of Binary Heaps

- Insertion - **O(log N)**
  Removal - **O(log N)**
  Search - **O(N)**

WHY LOG(N)?



- For 32 nodes, it will be 5 comparisons.
- As there is no guaranteed order, for searching it would take O(n) for searching

- Final Code

```
class Node {
    constructor(val, priority) {
        this.value = val;
        this.priority = priority;
    }
}

class PriorityQueue {
    constructor() {
        this.values = [];
    }

    enqueue(val, priority) {
        this.values.push(new Node(val, priority));
        this.bubbleUp();
    }

    bubbleUp() {
        let index = this.values.length - 1;
        while (index > 0) {
            let parentIndex = Math.floor((index-1)/2);
            if (this.values[index].priority >= this.values[parentIndex].priority) break;
            [this.values[index], this.values[parentIndex]] = [this.values[parentIndex], this.values[index]];
            index = parentIndex;
        }
    }


    dequeue() {
        [this.values[0], this.values[this.values.length-1]] = [this.values[this.values.length-1], this.values[0]];
        const min = this.values.pop();
        this.rearrangeQueue();
        return min;
    }


    rearrangeQueue() {
        let index = 0;
        while (true) {
            let leftChildIndex = (2*index) + 1;
            let rightChildIndex = (2*index) + 2;
            let swapIndex = null;
            if (this.values[index]?.priority > this.values[leftChildIndex]?.priority) swapIndex = leftChildIndex;
            if (
                (swapIndex === null &&
                this.values[index]?.priority >= this.values[rightChildIndex]?.priority) ||
                (swapIndex !== null &&
                this.values[index]?.priority >= this.values[rightChildIndex]?.priority &&
                this.values[rightChildIndex]?.priority < this.values[leftChildIndex]?.priority)
               )
                swapIndex = rightChildIndex;
            if (swapIndex === null) break;
            [this.values[index], this.values[swapIndex]] = [this.values[swapIndex], this.values[index]];
            index = swapIndex;
        }
    }

}

let pq = new PriorityQueue();
pq.enqueue("Common Cold", 4);
pq.enqueue("Gunshot wound", 1);
pq.enqueue("Broken Leg", 2);
pq.enqueue("High Fever", 3);
console.log(pq.values);
// [{value: "Gunshot wound", priority: 1},{value: "High Fever", priority: 3},{value: "Broken Leg", priority: 2},{value: "Common Cold", priority: 4}]
pq.dequeue();
console.log(pq.values);
// [{value: "Broken Leg", priority: 2},{value: "High Fever", priority: 3},{value: "Common Cold", priority: 4}]
// o/p -> {value: "Gunshot wound", priority: 1}
```

Hash Table

Monday, February 22, 2021 12:50 PM

WHAT IS A HASH TABLE?

Hash tables are used to store *key-value* pairs.

- They are like arrays, but the keys are not ordered.

Unlike arrays, hash tables are *fast* for all of the following operations: finding values, adding new values, and removing values!

WHY SHOULD I CARE?

- Nearly every programming language has some sort of hash table data structure

Because of their speed, hash tables are very commonly used!

HASH TABLES IN THE WILD

- Python has Dictionaries
- JS has Objects and Maps*
- Java, Go, & Scala have Maps
- Ruby has...Hashes

    - Objects can be used as hash table but with only strings

HASH TABLES

Introductory Example

Imagine we want to store some colors

- We could just use an array/list:

```
[ "#ff69b4","#ff4500","#00ffff" ]
```

Not super readable! What do these colors correspond to?

- The above can be used in cases where we need to assign random colors

HASH TABLES

Introductory Example

- It would be nice if instead of using indices to access the colors, we could use more human-readable keys.

colors["cyan"]

- is way better than

colors[2]

HASH TABLES

Introductory Example

How can we get human-readability *and* computer readability?

Computers don't know how to find an element at index *pink*!

Hash tables to the rescue!

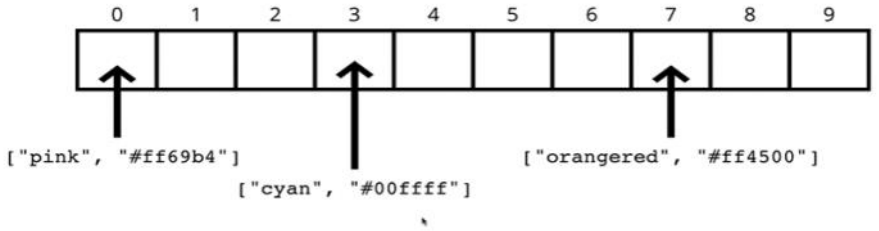
THE HASH PART

To implement a hash table, we'll be using an array.

In order to look up values by key, we need a way to convert keys into valid array indices.

A function that performs this task is called a *hash function*.

HASHING CONCEPTUALLY



Hash function

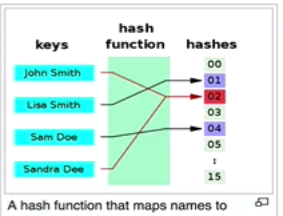
From Wikipedia, the free encyclopedia

This article is about a programming concept. For other meanings of "hash" and "hashing", see Hash (disambiguation).

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(July 2010) (Learn how and when to remove this template message)*

A **hash function** is any function that can be used to map data of arbitrary size to data of a fixed size. The values returned by a hash function are called **hash values**, **hash codes**, **digests**, or simply **hashes**. Hash functions are often used in combination with a hash table, a common data structure used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file. One such application is finding similar stretches in DNA sequences. They are also useful in cryptography. A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or any equivalent alternatives) by knowing the stored hash value. This is used for assuring integrity of transmitted data, and is the building block for HMACs, which provide message authentication.

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, lossy compression, randomization functions, error-correcting codes, and ciphers. Although the concepts overlap to some extent, each one has its own uses and requirements and is designed and optimized differently. The HashKeeper database maintained by the American National Drug Intelligence Center, for instance, is more aptly described as a catalogue of file fingerprints than of hash values.



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

WHAT MAKES A GOOD HASH?

(not a cryptographically secure one)

1. Fast (i.e. constant time)
2. Doesn't cluster outputs at specific indices, but distributes uniformly
3. Deterministic (same input yields same output)

- Bad example of hash function

What Makes for a Good Hash?

Fast

Non-Example

```
function slowHash(key) {
  for (var i = 0; i < 10000; i++) {
    console.log("everyday i'm hashing");
  }
  return key[0].charCodeAt(0);
}
```

What Makes for a Good Hash?

Uniformly Distributes Values

Non-Example

```
function sameHashedValue(key) {
  return 0;
}
```

What Makes for a Good Hash?

Deterministic

Non-Example

```
function randomHash(key) {
  return Math.floor(Math.random() * 1000)
}
```

What Makes for a Good Hash?

Simple Hash Example

Here's a hash that works on *strings only*:

```
function hash(key, arrayLen) {
  let total = 0;
  for (let char of key) {
    // map "a" to 1, "b" to 2, "c" to 3, etc.
    let value = char.charCodeAt(0) - 96
    total = (total + value) % arrayLen;
  }
  return total;
}
```

```
hash("pink", 10); // 0
hash("orangered", 10); // 7
hash("cyan", 10); // 3
```

COPY

REFINING OUR HASH

Problems with our current hash

1. Only hashes strings (we won't worry about this)
2. Not constant time - linear in key length
3. Could be a little more random

Prime numbers? wut.

The prime number in the hash is helpful in spreading out the keys more uniformly.

It's also helpful if the array that you're putting values into has a prime length.

You don't need to know why. (Math is complicated!)
But here are some links if you're curious.

Why do hash functions use prime numbers?

Does making array size a prime number help in hash table implementation?

```
function hash(key, arrayLen) {
  let total = 0;
  let WEIRD_PRIME = 31;
  for (let i = 0; i < Math.min(key.length, 100); i++) {
    let char = key[i];
    let value = char.charCodeAt(0) - 96
    total = (total * WEIRD_PRIME + value) % arrayLen;
  }
  return total;
}
```

- Handling Collisions

Dealing with Collisions

Even with a large array and a great hash function, collisions are inevitable.

There are many strategies for dealing with collisions, but we'll focus on two:

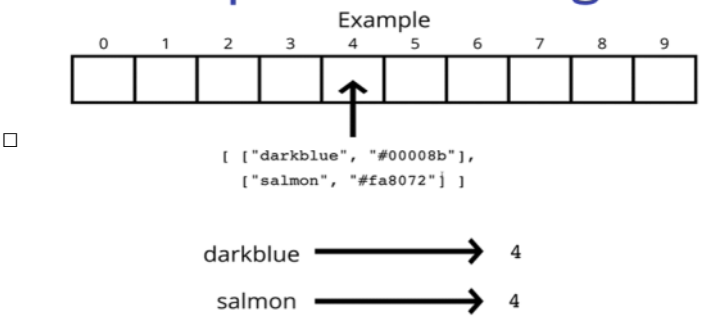
1. Separate Chaining
2. Linear Probing

Separate Chaining

With *separate chaining*, at each index in our array we store values using a more sophisticated data structure (e.g. an array or a linked list).

This allows us to store multiple key-value pairs at the same index.

Separate Chaining

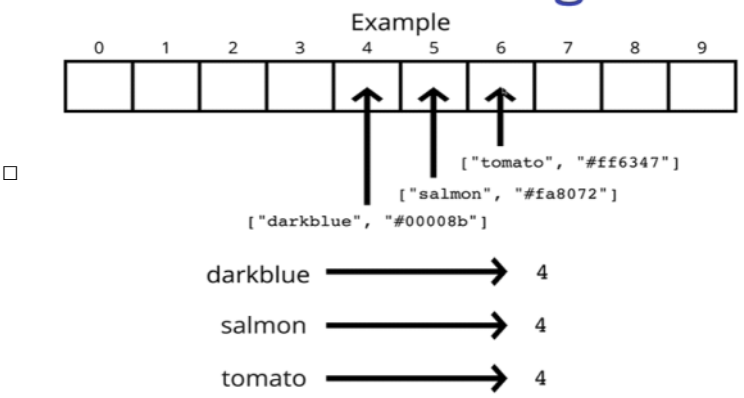


Linear Probing

With *linear probing*, when we find a collision, we search through the array to find the next empty slot.

Unlike with separate chaining, this allows us to store a single key-value at each index.

Linear Probing



A HashTable Class

```
class HashTable {
  constructor(size=53){
    this.keyMap = new Array(size);
  }

  _hash(key) {
    let total = 0;
    let WEIRD_PRIME = 31;
    for (let i = 0; i < Math.min(key.length, 100); i++) {
      let char = key[i];
      let value = char.charCodeAt(0) - 96
      total = (total * WEIRD_PRIME + value) % this.keyMap.length;
    }
    return total;
  }
}
```

Set / Get

set

1. Accepts a key and a value
2. Hashes the key
3. Stores the key-value pair in the hash table array via separate chaining

get

1. Accepts a key
2. Hashes the key
3. Retrieves the key-value pair in the hash table
4. If the key isn't found, returns `undefined`

```
class HashTable {
    constructor(size = 53) {
        this.keyMap = new Array(size);
    }

    _hash(key) {
        let keyIndex = 0;
        let WEIRD_PRIME = 31;
        for (let i = 0; i < Math.min(key.length, 100); i++) {
            let char = key[i];
            let value = char.charCodeAt(0) - 96;
            keyIndex = (keyIndex * WEIRD_PRIME + value) % this.keyMap.length;
        }
        return keyIndex;
    }

    set(key, value) {
        const index = this._hash(key);
        console.log(index);
        if (!this.keyMap[index]) this.keyMap[index] = [];
        this.keyMap[index].push([key, value]);
        console.log(this.keyMap);
    }

    get(key) {
        const index = this._hash(key);
        if (!this.keyMap[index]) return undefined;
        for (let i = 0; i < this.keyMap[index].length; i++) {
            if (this.keyMap[index][i][0] === key) return this.keyMap[index][i][1];
        }
        return undefined;
    }
}


let hashTable = new HashTable(7);
hashTable.set("colour", "red");
hashTable.set("model", "hatchback");
hashTable.set("brand", "hyundai");
hashTable.set("fuelsource", "petrol");
console.log(hashTable.get("fuelsource")); // petrol
console.log(hashTable.get("model")); // hatchback
console.log(hashTable.get("colour")); // red
console.log(hashTable.get("engine")); // undefined
```

Keys / Values

keys

1. Loops through the hash table array and returns an array of keys in the table

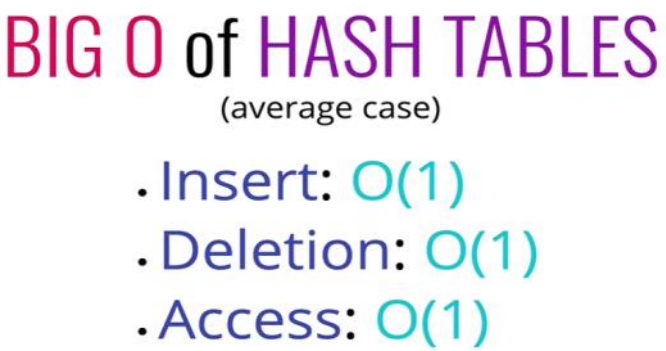
values

1. Loops through the hash table array and returns an array of values in the table

```
class HashTable {
    constructor(size = 53) {
        this.keyMap = new Array(size);
        // this.keys = [];
        // this.values = [];
    }

    _hash(key) {
        let keyIndex = 0;
        let WEIRD_PRIME = 31;
        for (let i = 0; i < Math.min(key.length, 100); i++) {
            let char = key[i];
            let value = char.charCodeAt(0) - 96;
            keyIndex = (keyIndex * WEIRD_PRIME + value) % this.keyMap.length;
        }
        return keyIndex;
    }

    set(key, value) {
        const index = this._hash(key);
        console.log(index);
        if (!this.keyMap[index]) this.keyMap[index] = [];
        this.keyMap[index].push([key, value]);
        // this.keys.push(key);
        // if (this.keys.indexOf(key) === -1) this.keys.push(key);
        // this.values.push(value);
        // if (this.values.indexOf(value) === -1) this.values.push(value);
        console.log(this.keyMap);
    }

    get(key) {
        const index = this._hash(key);
        if (!this.keyMap[index]) return undefined;
        for (let i = 0; i < this.keyMap[index].length; i++) {
            if (this.keyMap[index][i][0] === key) return this.keyMap[index][i][1];
        }
        return undefined;
    }
```

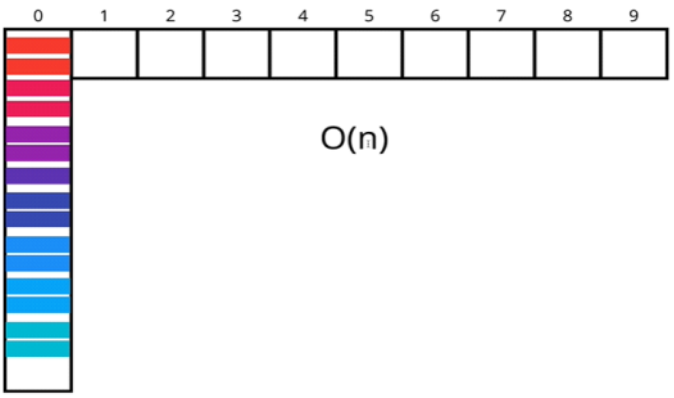
```
        keys() {
            // return this.keys;
            return this._helper(true);
        }

        values() {
            // return this.values;
            return this._helper(false);
        }

        _helper(isKey = false) {
            let list = [];
            const listIndex = isKey ? 0 : 1;
            for (let i = 0; i < this.keyMap.length; i++) {
                if (!this.keyMap[i]) continue;
                for (let j = 0; j < this.keyMap[i].length; j++) {
                    if (list.includes(this.keyMap[i][j][listIndex])) continue;
                    list.push(this.keyMap[i][j][listIndex]);
                }
            }
            return list;
        }
    }

    let hashTable = new HashTable(7);
    hashTable.set("colour", "red");
    hashTable.set("model", "hatchback");
    hashTable.set("brand", "hyundai");
    hashTable.set("brand", "honda");
    hashTable.set("brand", "hyundai");
    hashTable.set("fuelsource", "petrol");
    console.log(hashTable.get("fuelsource")); // petrol
    console.log(hashTable.get("model")); // hatchback
    console.log(hashTable.get("colour")); // red
    console.log(hashTable.get("engine")); // undefined
    console.log(hashTable.keys()); // ["model", "fuelsource", "brand", "colour"]
    console.log(hashTable.values()); // ["hatchback", "petrol", "hyundai", "honda", "red"]
```

-



  - Access here means getting the value assigned to a particular key.

With the world's worst hash function...



Recap

- Hash tables are collections of key-value pairs
- Hash tables can find values quickly given a key
- Hash tables can add new key-values quickly
- Hash tables store data in a large array, and work by *hashing* the keys
- A good hash should be fast, distribute keys uniformly, and be deterministic
- Separate chaining and linear probing are two strategies used to deal with two keys that hash to the same index

Graphs

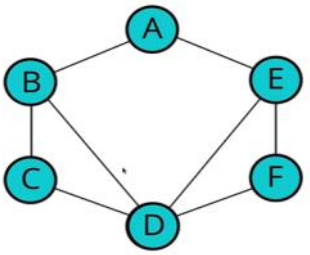
Wednesday, March 3, 2021 11:25 AM

WHAT ARE GRAPHS

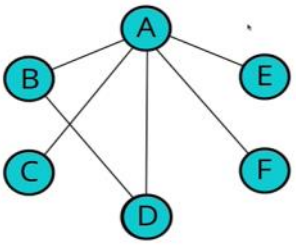
- A **graph data structure** consists of a finite (and possibly mutable) set of vertices or nodes or points, together with a set of unordered pairs of these vertices for an undirected **graph** or a set of ordered pairs for a directed **graph**.

- Examples

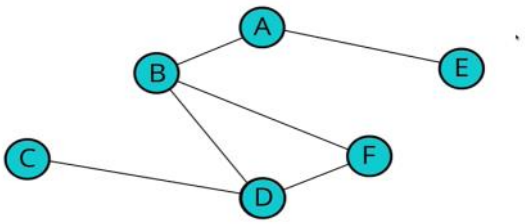
NODES + CONNECTIONS



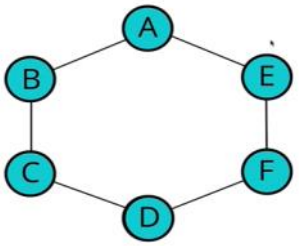
NODES + CONNECTIONS



NODES + CONNECTIONS



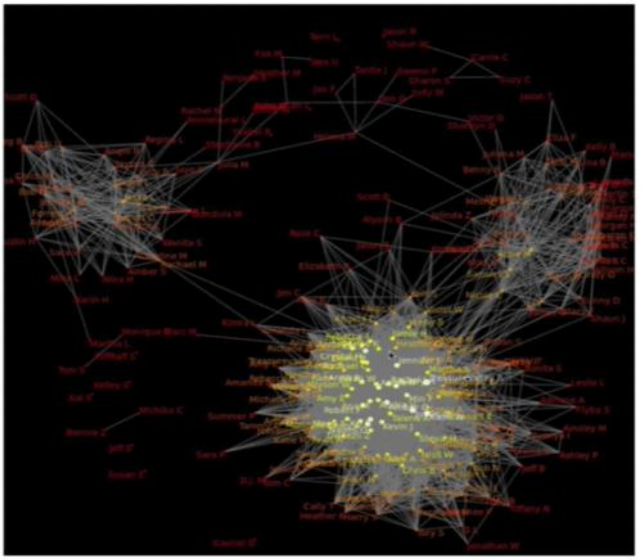
NODES + CONNECTIONS

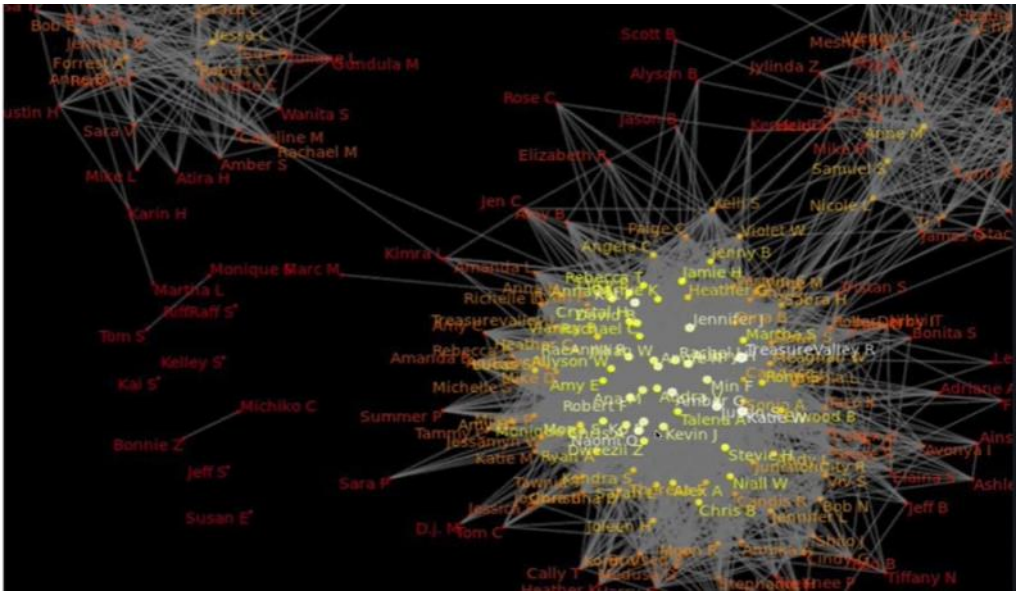


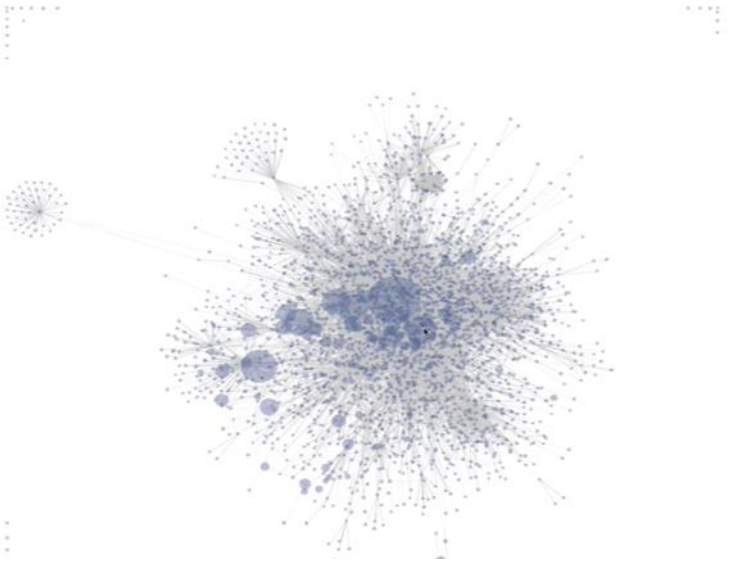
USES FOR GRAPHS

- Social Networks
- Location / Mapping
- Routing Algorithms
- Visual Hierarchy
- File System Optimizations
- **EVERYWHERE!**

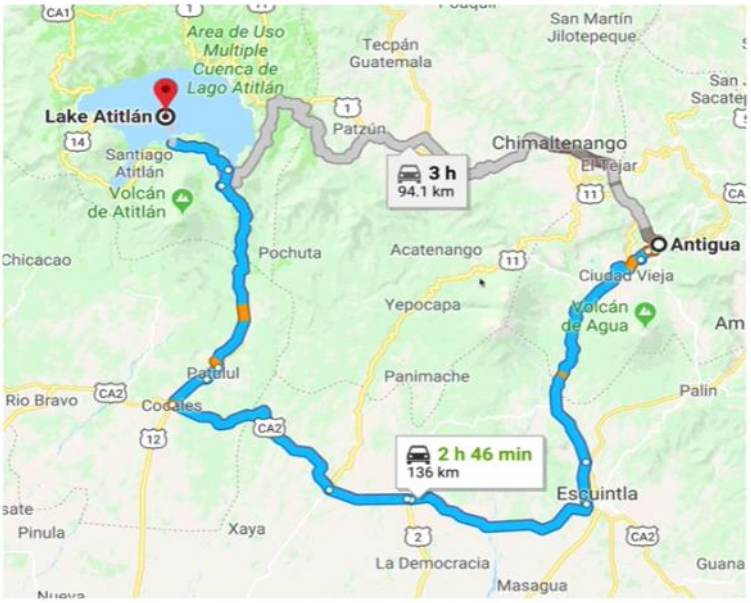
- Example of Friend network in a social networking site



- 

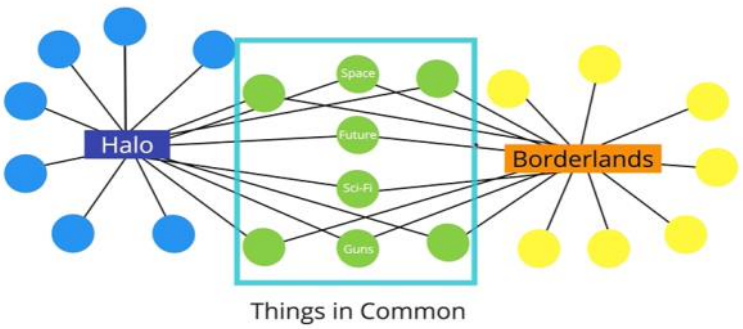
- Wikipedia and it's web pages
- 

- Google maps (Fastest path based on mode of transportation)

- 

Recommendations

- "People also watched"
- "You might also like..."
- "People you might know"
- "Frequently bought with"

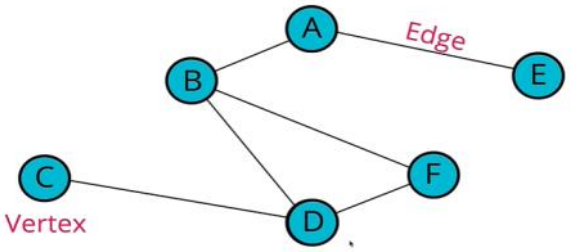
- People who played/searched/bought Halo will be recommended with Border Lands based on the common things it has.
- 

- Types of Graphs

ESSENTIAL GRAPH TERMS

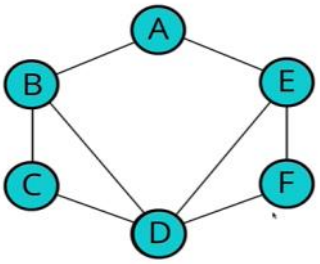
- **Vertex -** a node
- **Edge -** connection between nodes
- **Weighted/Unweighted -** values assigned to distances between vertices
- **Directed/Undirected -** directions assigned to distanced between vertices

TERMINOLOGY

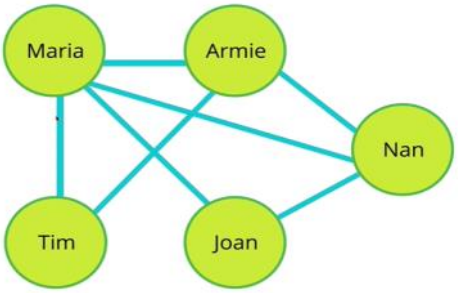


- Undirected Graph is one which has no direction restriction, we can traverse through vertices in any direction through edges.

UNDIRECTED GRAPH

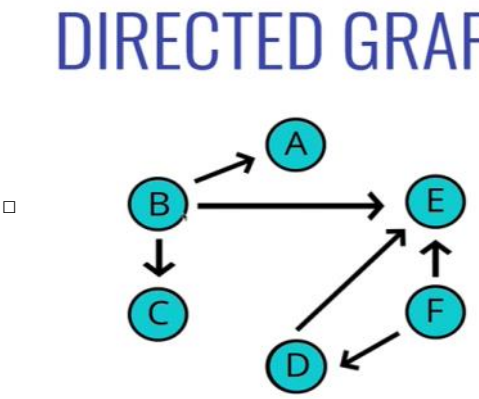


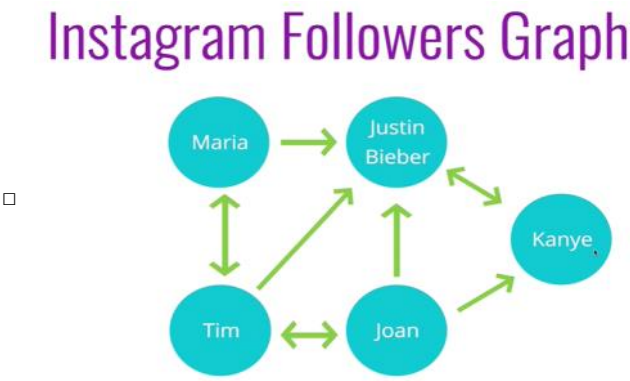
Facebook Friends Graph



- In the above Undirected Graphs Maria can see Tim's content and Tim can see Maria's content.

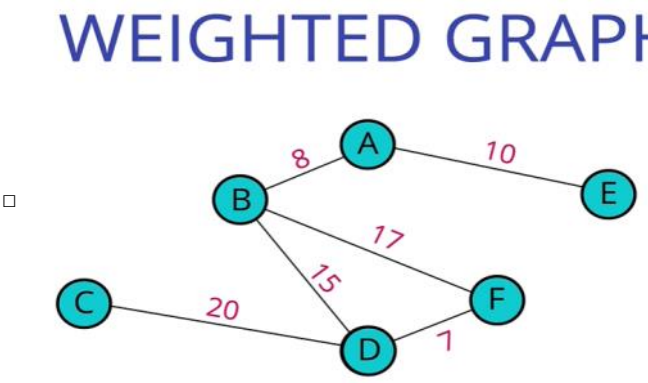
- In Directed Graph, the traversing is directed

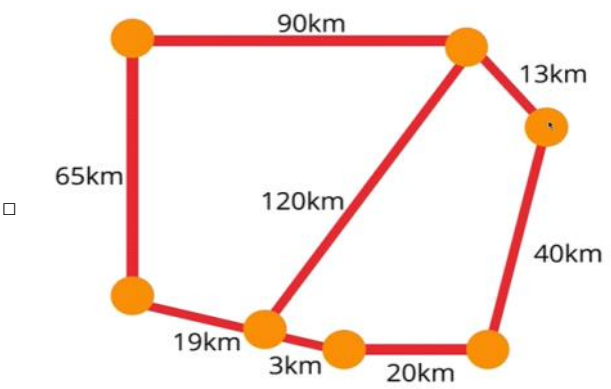




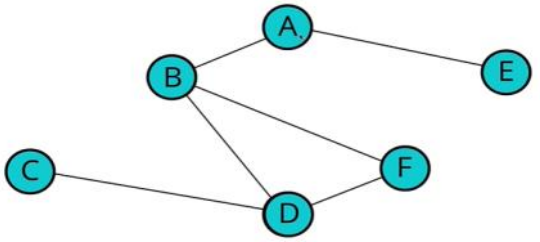
- In above example Tim follows Justin Bieber but Justin Beiber does not follow Tim

- Weighed Graph has weights along its edges



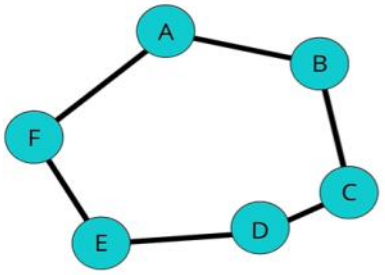


- Unweighted Graph



- Representation of Graphs

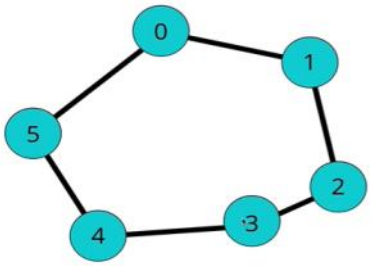
ADJACENCY MATRIX



-	A	B	C	D	E	F
A	0	**1**	0	0	0	**1**
B	**1**	0	**1**	0	0	0
C	0	**1**	0	**1**	0	0
D	0	0	**1**	0	**1**	0
E	0	0	0	**1**	0	**1**
F	**1**	0	0	0	**1**	0

- 1 in the matrix represents an edge between the vertices

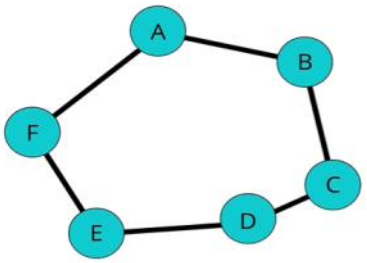
ADJACENCY LIST




```
[
0  [1,5],
1  [0,2],
2  [1,3],
3  [2,4],
4  [3,5],
5  [4,0]
]
```

- Array of each node represents the edges to the vertices in the array.

ADJACENCY LIST




```
{
    A: ["B", "F"],
    B: ["A", "C"],
    C: ["B", "D"],
    D: ["C", "E"],
    E: ["D", "F"],
    F: ["E", "A"]
}
```

- Above is an example with Adjacency List with Hash Map/Hash Table

DIFFERENCES & BIG O

|V| - number of vertices
|E| - number of edges

OPERATION	ADJACENCY LIST	ADJACENCY MATRIX
Add Vertex	O(1)	O(|V^2|)
Add Edge	O(1)	O(1)
Remove Vertex	O(|V| + |E|)	O(|V^2|)
Remove Edge	O(|E|)	O(1)
Query	O(|V| + |E|)	O(1)
Storage	O(|V| + |E|)	O(|V^2|)

Adjacency List vs. Adjacency Matrix

- Can take up less space (in sparse graphs)
- Faster to iterate over all edges
- Can be slower to lookup specific edge

- Takes up more space (in sparse graphs)
- Slower to iterate over all edges
- Faster to lookup specific edge

What will we use?

- An Adjacency List
- An Adjacency List
- An Adjacency List

Why?

Most data in the real-world tends to lend itself to sparser and/or larger graphs

- sparser - thinly dispersed/scattered

ADDING A VERTEX

- Write a method called addVertex, which accepts a name of a vertex
- It should add a key to the adjacency list with the name of the vertex and set its value to be an empty array

```
g.addVertex("Tokyo")
```

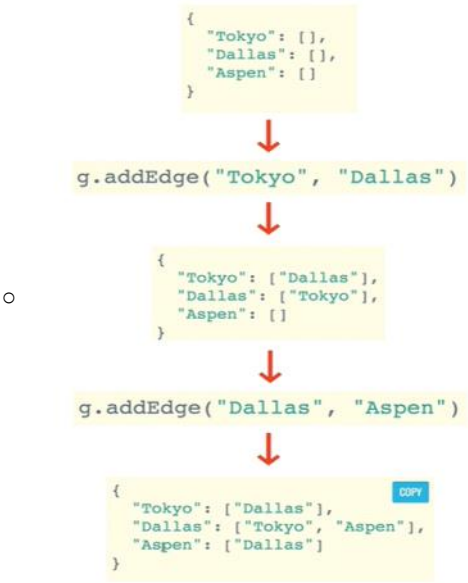
↓

```
{
    "Tokyo": []
}
```

```
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(vertex) {
        if (this.adjacencyList[vertex]) return;
        this.adjacencyList[vertex] = [];
    }
}
```

ADDING AN EDGE

- This function should accept two vertices, we can call them vertex1 and vertex2
- The function should find in the adjacency list the key of vertex1 and push vertex2 to the array
- The function should find in the adjacency list the key of vertex2 and push vertex1 to the array
- Don't worry about handling errors/invalid vertices

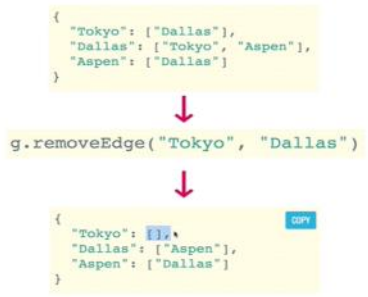



```
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(vertex) {
        if (this.adjacencyList[vertex]) return;
        this.adjacencyList[vertex] = [];
    }

    addEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1].push(vertex2);
        this.adjacencyList[vertex2].push(vertex1);
    }
}
```

REMOVING AN EDGE

- This function should accept two vertices, we'll call them vertex1 and vertex2
- The function should reassign the key of vertex1 to be an array that does not contain vertex2
- The function should reassign the key of vertex2 to be an array that does not contain vertex1
- Don't worry about handling errors/invalid vertices

REMOVING AN EDGE




```
// Undirected and Unweighted Graph
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(vertex) {
        if (this.adjacencyList[vertex]) return;
        this.adjacencyList[vertex] = [];
    }

    addEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1].push(vertex2);
        this.adjacencyList[vertex2].push(vertex1);
    }

    removeEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            !this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1] = this.adjacencyList[vertex1].filter(v => v !== vertex2);
        this.adjacencyList[vertex2] = this.adjacencyList[vertex2].filter(v => v !== vertex1);
    }
}


let graph = new Graph();
graph.addVertex("Tokyo");
graph.addVertex("Dallas");
graph.addVertex("Aspen");
graph.addEdge("Tokyo", "Dallas");
graph.addEdge("Dallas", "Aspen");
console.log(graph.adjacencyList);
graph.removeEdge("Tokyo", "Dallas");
console.log(graph.adjacencyList);
```

REMOVING A VERTEX

- The function should accept a vertex to remove
- The function should loop as long as there are any other vertices in the adjacency list for that vertex
- Inside of the loop, call our **removeEdge** function with the vertex we are removing and any values in the adjacency list for that vertex
- delete the key in the adjacency list for that vertex

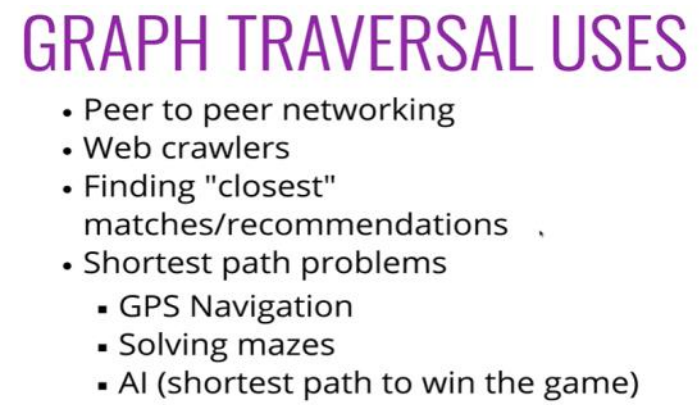
REMOVING A VERTEX


```
// Undirected and Unweighted Graph
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(vertex) {
        if (this.adjacencyList[vertex]) return;
        this.adjacencyList[vertex] = [];
    }

    addEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1].push(vertex2);
        this.adjacencyList[vertex2].push(vertex1);
    }

    removeEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            !this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1] = this.adjacencyList[vertex1].filter(v => v !== vertex2);
        this.adjacencyList[vertex2] = this.adjacencyList[vertex2].filter(v => v !== vertex1);
    }

    removeVertex(vertex1) {
        if (!this.adjacencyList[vertex1] || this.adjacencyList[vertex1]?.length === 0) return;
        while (this.adjacencyList[vertex1].length) {
            const lastVertexIndex = this.adjacencyList[vertex1].length - 1;
            const vertex2 = this.adjacencyList[vertex1][lastVertexIndex];
            this.removeEdge(vertex1, vertex2);
        }
        delete this.adjacencyList[vertex1];
    }
}

let graph = new Graph();
graph.addVertex("Tokyo");
graph.addVertex("Dallas");
graph.addVertex("Aspen");
graph.addVertex("Hong Kong");
graph.addVertex("Los Angeles");
graph.addEdge("Tokyo", "Dallas");
graph.addEdge("Dallas", "Aspen");
graph.addEdge("Dallas", "Los Angeles");
graph.addEdge("Tokyo", "Hong Kong");
graph.addEdge("Dallas", "Hong Kong");
graph.addEdge("Hong Kong", "Los Angeles");
graph.removeEdge("Tokyo", "Dallas");
graph.removeVertex("Hong Kong");
```

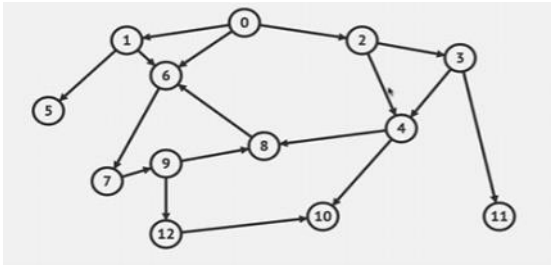
Graph Traversal

Wednesday, March 3, 2021 1:56 PM

- 

- 

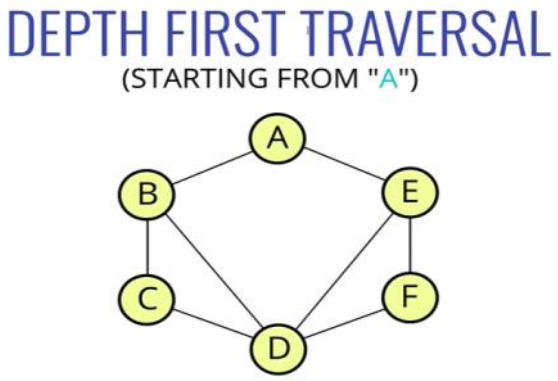
- We need graph traversal to visit each vertex in a graph atleast once to calculate what the shortest path can be.
- A tree is also kind of graph.
- Just like tree traversal we have two methods for traversing through a graph
  - DFS
  - BFS
- 

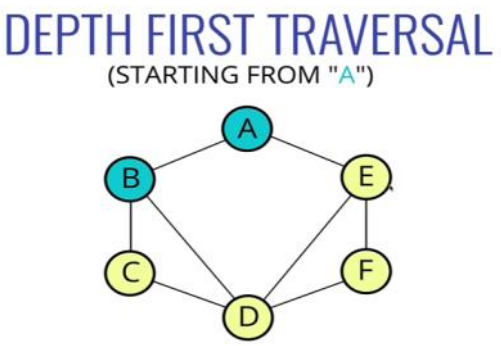
  - DFS in Tree traversal is actually visiting the children first before visiting the siblings.
  - 

  - But in Graph Traversal DFS refers to moving away from the node.

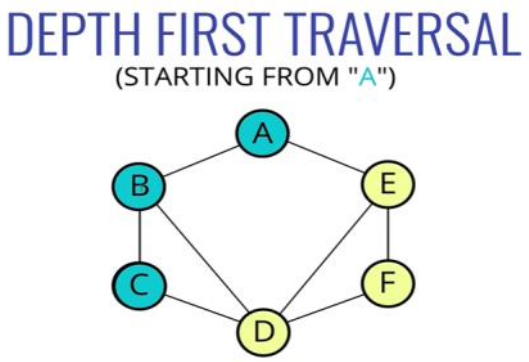
- In the above example if we take DFS
  - Follow the neighbors of a node before visiting it's siblings
  - There is no kind of root node where we start from in a graph.
  - In DFS, if we start at 0, we traverse 0 -> 1 -> 5, 1 -> 6 -> 7 -> 9 -> 12 -> 10

- Example of Traversal

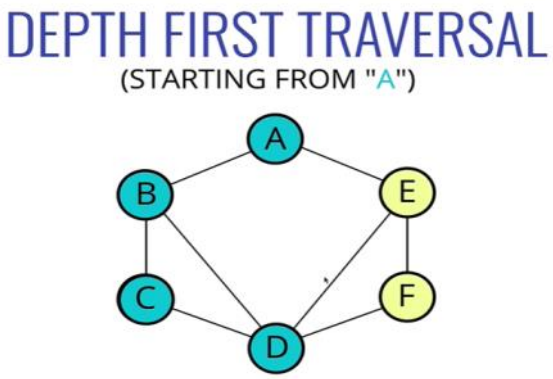




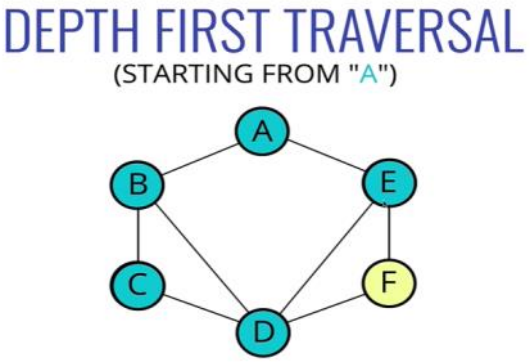
  - From B, we have the option to go to C or D



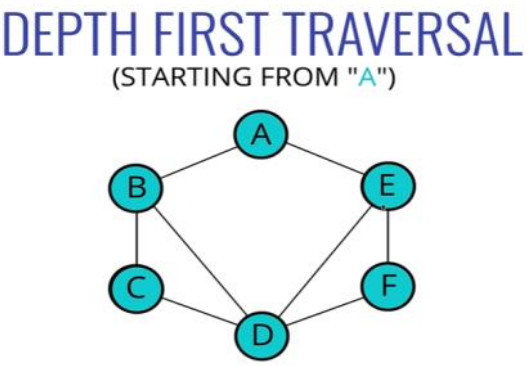
  - From C, we have the option to go to B or D
  - Since we came from B we go to D



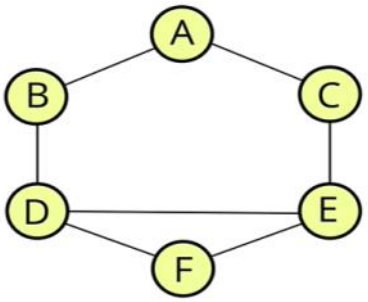
- From D, we have the option to go to B or E
- Since we already visited B we go to E



- From E, we have the option to go to A or F
- Since we already visited A we go to F

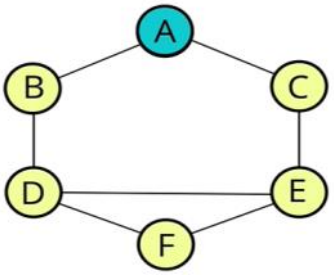


- Another Example of Graph Traversal

- 


```
{
"A":["B", "C"],
"B":["A", "D"],
"C":["A", "E"],
"D":["B", "E", "F"],
"E":["C", "D", "F"],
"F":["D", "E"]
}
```

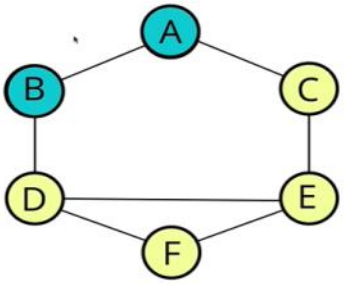
  - We start at A

- 

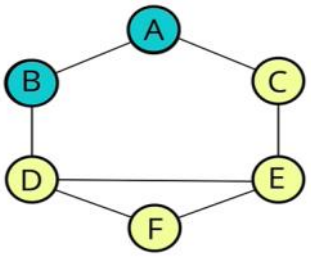

```
{
"A":["B", "C"],
"B":["A", "D"],
"C":["A", "E"],
"D":["B", "E", "F"],
"E":["C", "D", "F"],
"F":["D", "E"]
}
```

DEPTH FIRST TRAVERSAL

(STARTING FROM "A")

- 

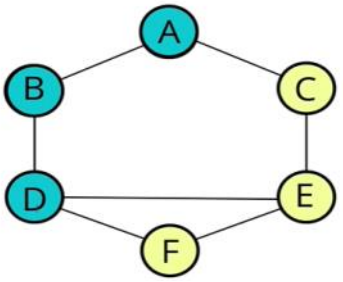
  - From A, we have the option to go to B or C
  - We go to B
  - Pictorial representation in Array by crossing out vertices which have been visited already

- 

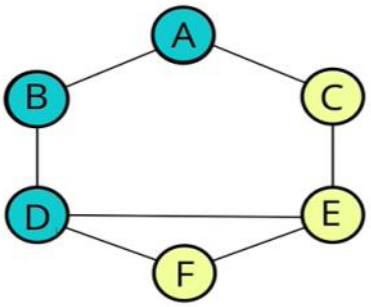

```
{
"A":["X", "C"],
"B":["X", "D"],
"C":["X", "E"],
"D":["X", "E", "F"],
"E":["C", "D", "F"],
"F":["D", "E"]
}
```

DEPTH FIRST TRAVERSAL

(STARTING FROM "A")

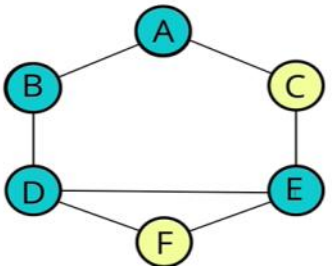
- 

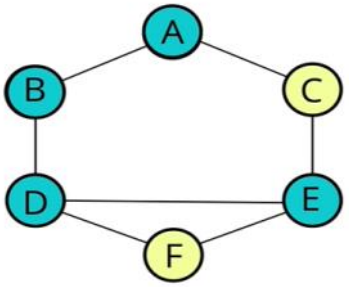
  - We have only one option to go from B to D

  - 


```
{
"A":["X", "C"],
"B":["X", "X"],
"C":["X", "E"],
"D":["X", "E", "F"],
"E":["C", "X", "F"],
"F":["X", "E"]
}
```

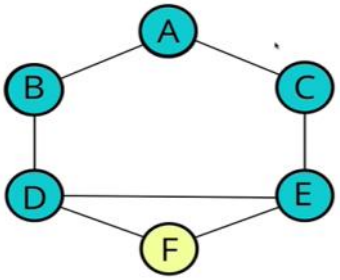
DEPTH FIRST TRAVERSAL
(STARTING FROM "A")

- 

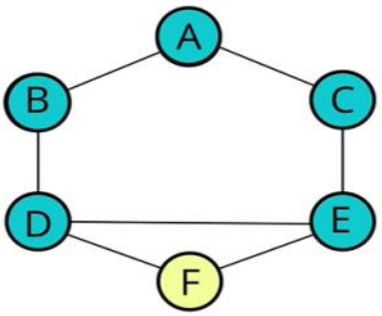
  - From D, we have the option to go to E or F
  - We go to E
  - 


```
{
"A":["X", "C"],
"B":["X", "X"],
"C":["X", "X"],
"D":["X", "X", "F"],
"E":["C", "X", "F"],
"F":["X", "X"]
}
```

DEPTH FIRST TRAVERSAL
(STARTING FROM "A")

- 

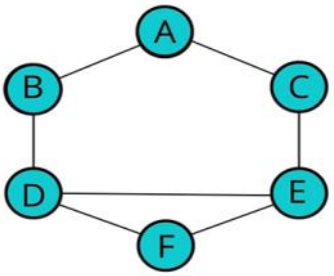
  - From E, we have the option to go to C or F
  - We go to C
  - But now from C, it is a dead-end, we have visited all the nodes
  - So, we go back to E and then to F

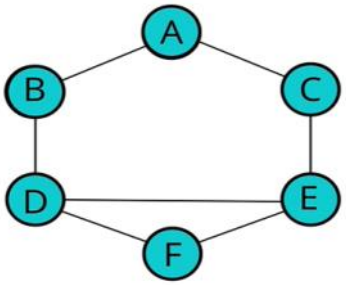
- 


```
{
"A":["X", "X"],
"B":["X", "X"],
"C":["X", "X"],
"D":["X", "X", "F"],
"E":["X", "X", "F"],
"F":["X", "X"]
}
```

DEPTH FIRST TRAVERSAL

(STARTING FROM "A")

- 

- 


```
{
"A":["X", "X"],
"B":["X", "X"],
"C":["X", "X"],
"D":["X", "X", "X"],
"E":["X", "X", "X"],
"F":["X", "X"]
}
```

DFS PSEUDOCODE

Recursive

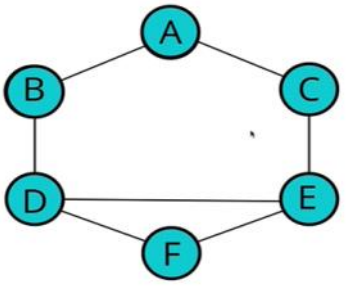
```
DFS(vertex):
   if vertex is empty
       return (this is base case)
   add vertex to results list
   mark vertex as visited
   for each neighbor in vertex's neighbors:
      if neighbor is not visited:
         recursively call DFS on neighbor
```

VISITING THINGS

```
{
  "A": true,
  "B": true,
  "D": true
}
```

```
g.addVertex("A")
g.addVertex("B")
g.addVertex("C")
g.addVertex("D")
g.addVertex("E")
g.addVertex("F")
```

```
g.addEdge("A","B")
g.addEdge("A","C")
g.addEdge("B","D")
g.addEdge("C","E")
g.addEdge("D","E")
g.addEdge("D","F")
g.addEdge("E","F")
```




```
{
"A":["B", "C"],
"B":["A", "D"],
"C":["A", "E"],
"D":["B", "E", "F"],
"E":["C", "D", "F"],
"F":["D", "E"]
}
```

```
{
  "A": true,
  "B": true,
  "D": true,
  "E": true,
  "C": true,
  "F": true
}
```

DEPTH FIRST TRAVERSAL

Recursive

- The function should accept a starting node
- Create a list to store the end result, to be returned at the very end
- Create an object to store visited vertices
- Create a helper function which accepts a vertex
  - The helper function should return early if the vertex is empty
  - The helper function should place the vertex it accepts into the visited object and push that vertex into the result array.
  - Loop over all of the values in the adjacencyList for that vertex
  - If any of those values have not been visited, recursively invoke the helper function with that vertex
- Invoke the helper function with the starting vertex
- Return the result array

```
depthFirstRecursive(start) {
    let result = [];
    let visited = {};
    const self = this;
    (function dfs(vertex) {
        if (!self.adjacencyList[vertex]) return null;
        visited[vertex] = true;
        result.push(vertex);
        self.adjacencyList[vertex].forEach((node) => {
            if (!visited[node]) return dfs(node);
        });
    })(start);
    console.log(result);
    return result;
}
```

- *O/p -> ["A", "B", "D", "E", "C", "F"]*

DFS PSEUDOCODE

Iterative

```
DFS-iterative(start):
    let S be a stack
    S.push(start)
    while S is not empty
        vertex = S.pop()
        if vertex is not labeled as discovered:
            visit vertex (add to result list)
            label vertex as discovered
            for each of vertex's neighbors, N do
                S.push(N)
```

DEPTH FIRST TRAVERSAL

Iterative

- The function should accept a starting node
- Create a stack to help use keep track of vertices (use a list/array)
- Create a list to store the end result, to be returned at the very end
- Create an object to store visited vertices
- Add the starting vertex to the stack, and mark it visited
- While the stack has something in it:
  - Pop the next vertex from the stack
  - If that vertex hasn't been visited yet:
    - Mark it as visited
    - Add it to the result list
    - Push all of its neighbors into the stack
- Return the result array

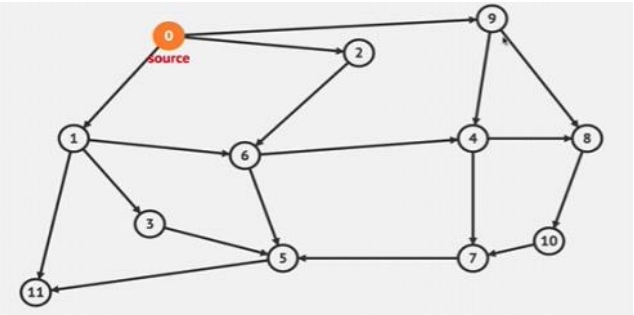
```
depthFirstIterative(start) {
    let result = [];
    let stack = [start];
    let visited = {};
    visited[start] = true;
    while (stack.length) {
        let vertex = stack.pop();
        result.push(vertex);
        this.adjacencyList[vertex].forEach((node) => {
            if (!visited[node]) {
                stack.push(node);
                visited[node] = true;
            }
        });
    }
    return result;
}
```

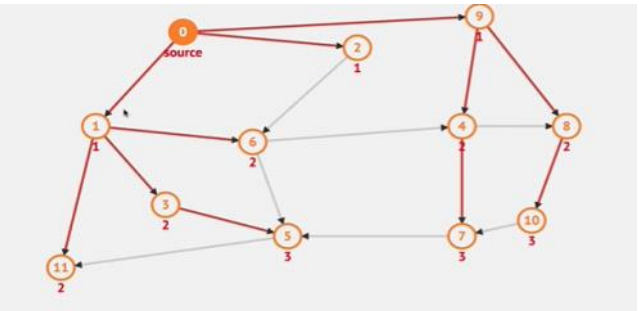
- *O/p -> ["A", "C", "E", "F", "D", "B"]*
- *It is DFS but order is different from recursive since it uses stack to pop out the last value.*

BREADTH FIRST

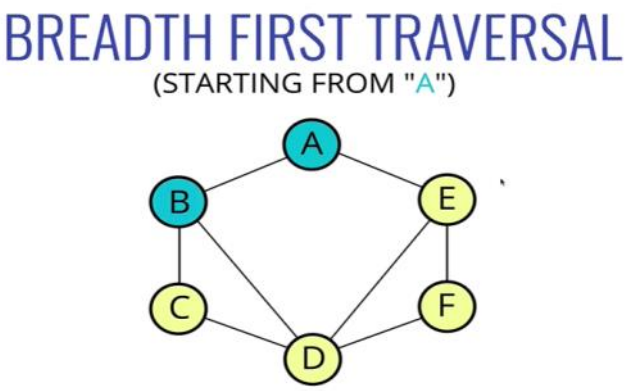
Visit neighbors at current depth first!

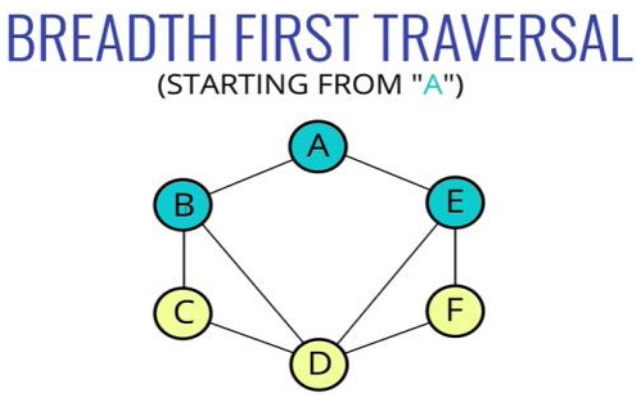
- Example

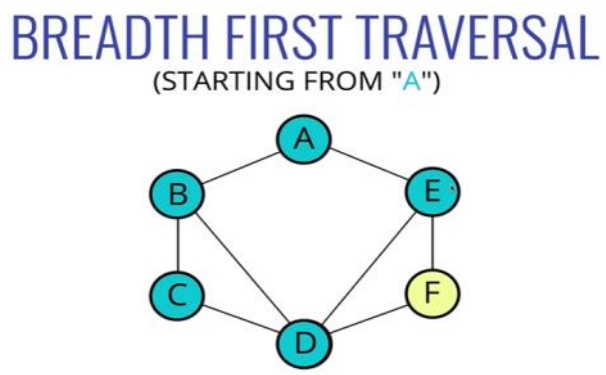




- Another Example

- 

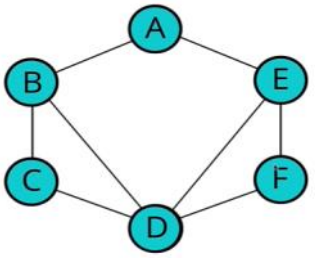
  - A -> B
- 

  - A -> E
- 

  - We go back to B and traverse through all its non-visited neighbors
  - B -> C, B -> D

BREADTH FIRST TRAVERSAL

(STARTING FROM "A")



- We go back to E, and traverse through all its non-visited neighbors
- E -> F

BREADTH FIRST

- This function should accept a starting vertex
- Create a queue (you can use an array) and place the starting vertex in it
- Create an array to store the nodes visited
- Create an object to store nodes visited
- Mark the starting vertex as visited
- Loop as long as there is anything in the queue
- Remove the first vertex from the queue and push it into the array that stores nodes visited
- Loop over each vertex in the adjacency list for the vertex you are visiting.
- If it is not inside the object that stores nodes visited, mark it as visited and enqueue that vertex
- Once you have finished looping, return the array of visited nodes

- For BFS, we are going to use the QUEUE approach (FIFO) with push to insert into queue and shift to remove from queue

```
breadthFirstTraversal(start) {
    let queue = [start];
    let result = [];
    let visited = {};
    visited[start] = true;
    while (queue.length) {
        let vertex = queue.shift();
        result.push(vertex);
        this.adjacencyList[vertex].forEach((node) => {
            if (!visited[node]) {
                visited[node] = true;
                queue.push(node);
            }
        });
    }
    return result;
}
```

.

```
// Undirected and Unweighted Graph
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(vertex) {
        if (this.adjacencyList[vertex]) return;
        this.adjacencyList[vertex] = [];
    }

    addEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1].push(vertex2);
        this.adjacencyList[vertex2].push(vertex1);
    }

    removeEdge(vertex1, vertex2) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            !this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1] = this.adjacencyList[vertex1].filter(v => v !== vertex2);
        this.adjacencyList[vertex2] = this.adjacencyList[vertex2].filter(v => v !== vertex1);
    }

    removeVertex(vertex1) {
        if (!this.adjacencyList[vertex1] || this.adjacencyList[vertex1]?.length === 0) return;
        while (this.adjacencyList[vertex1].length) {
            const lastVertexIndex = this.adjacencyList[vertex1].length - 1;
            const vertex2 = this.adjacencyList[vertex1][lastVertexIndex];
            this.removeEdge(vertex1, vertex2);
        }
        delete this.adjacencyList[vertex1];
    }


    depthFirstRecursive(start) {
        let result = [];
        let visited = {};
        const self = this;
        (function dfs(vertex) {
            if (!self.adjacencyList[vertex]) return null;
            visited[vertex] = true;
            result.push(vertex);
            self.adjacencyList[vertex].forEach((node) => {
                if (!visited[node]) return dfs(node);
            });
        })(start);
        console.log(result);
        return result;
    }

    depthFirstIterative(start) {
        let result = [];
        let stack = [start];
        let visited = {};
        visited[start] = true;
        while (stack.length) {
            let vertex = stack.pop();
            result.push(vertex);
            this.adjacencyList[vertex].forEach((node) => {
                if (!visited[node]) {
                    stack.push(node);
                    visited[node] = true;
                }
            });
        }
        return result;
    }
```

```
    breadthFirstTraversal(start) {
        let queue = [start];
        let result = [];
        let visited = {};
        visited[start] = true;
        while (queue.length) {
            let vertex = queue.shift();
            result.push(vertex);
            this.adjacencyList[vertex].forEach((node) => {
                if (!visited[node]) {
                    visited[node] = true;
                    queue.push(node);
                }
            });
        }
        return result;
    }
}

let graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addVertex("E");
graph.addVertex("F");

//           A
//         /     \
//       B        C
//       |        |
//       D ----- E
//         \     /
//           F


graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");
graph.addEdge("C", "E");
graph.addEdge("D", "E");
graph.addEdge("D", "F");
graph.addEdge("E", "F");
// A: (2) ["B", "C"]
// B: (2) ["A", "D"]
// C: (2) ["A", "E"]
// D: (3) ["B", "E", "F"]
// E: (3) ["C", "D", "F"]
// F: (2) ["D", "E"]
graph.depthFirstRecursive("A"); // ["A", "B", "D", "E", "C", "F"]
graph.depthFirstIterative("A"); // ["A", "C", "E", "F", "D", "B"]
graph.breadthFirstTraversal("A"); // ["A", "B", "C", "D", "E", "F"]
```

Dijkstra's Algorithm

Monday, March 8, 2021 4:16 PM

WHAT IS IT

One of the most famous and widely used algorithms around!

Finds the shortest path between two vertices on a graph

"What's the fastest way to get from point A to point B?"

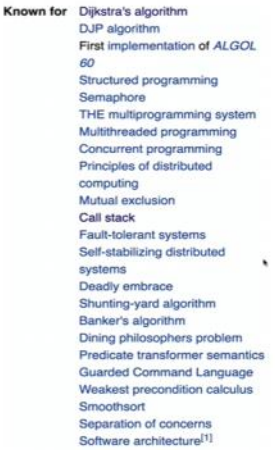
WHO WAS HE?

Edsger Dijkstra was a Dutch programmer, physicist, essayist, and all around smarty-pants

He helped to advance the field of computer science from an "art" to an academic discipline

Many of his discoveries and algorithms are still commonly used to this day.

HE DID A LOT...



Known for

- Dijkstra's algorithm
- DJP algorithm
- First implementation of *ALGOL 60*
- Structured programming
- Semaphore
- THE multiprogramming system
- Multithreaded programming
- Concurrent programming
- Principles of distributed computing
- Mutual exclusion
- Call stack
- Fault-tolerant systems
- Self-stabilizing distributed systems
- Deadly embrace
- Shunting-yard algorithm
- Banker's algorithm
- Dining philosophers problem
- Predicate transformer semantics
- Guarded Command Language
- Weakest precondition calculus
- Smoothsort
- Separation of concerns
- Software architecture[1]

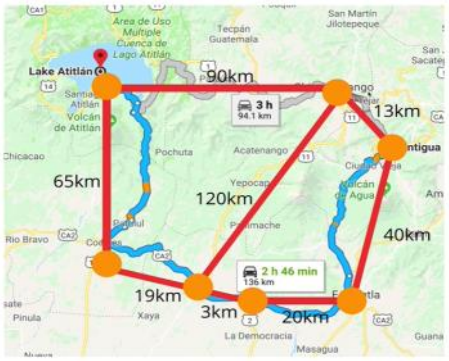
A QUOTE

What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame.

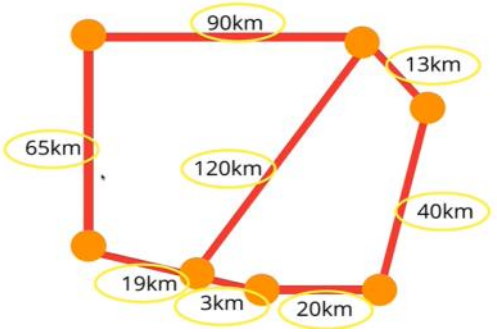
— Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001[2]

WHY IS IT USEFUL?

- GPS - finding fastest route
- Network Routing - finds open shortest path for data
- Biology - used to model the spread of viruses among humans
- Airline tickets - finding cheapest route to your destination
- Biology - used to model the spread of viruses among humans
- Many other uses!

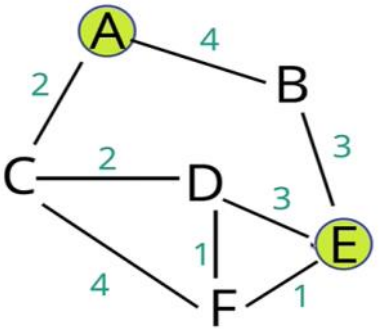


OUR GRAPH CAN'T DO THIS YET!



- To do this, we need to implement an weighted graph

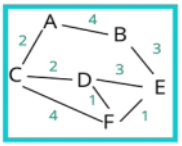
Find the shortest path from A to E



-

THE APPROACH

1. Every time we look to visit a new node, we pick the node with the smallest known distance to visit first.
2. Once we've moved to the node we're going to visit, we look at each of its neighbors
3. For each neighboring node, we calculate the distance by summing the total edges that lead to the node we're checking *from the starting node.*
4. If the new total distance to a node is less than the previous total, we store the new shorter distance for that node.



FIND THE SHORTEST PATH FROM A TO E

Vertex	Shortest Dist From A
A	
B	
C	
D	
E	
F	

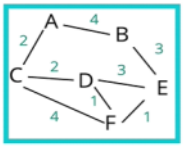
Visited:

```
[]
```

Previous:

```
{
    A: null,
    B: null,
    C: null,
    D: null,
    E: null,
    F: null
}
```

- Initially we initialize all the shortest distances from A to each vertex



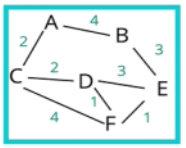
FIND THE SHORTEST PATH FROM A TO E

Visited:

```
[]
```

Vertex	Shortest Dist From A
A	0
B	Infinity

Previous:



FIND THE SHORTEST PATH FROM A TO E

- Visited:

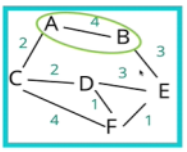
```
[]
```

Vertex	Shortest Dist From A
A	0
B	Infinity
C	Infinity
D	Infinity
E	Infinity
F	Infinity

Previous:

```
{
    A: null,
    B: null,
    C: null,
    D: null,
    E: null,
    F: null
}
```

- We assign infinity to all the vertices we did not visit or when we do not know what the shortest distance is.



FIND THE SHORTEST PATH FROM A TO E

- Pick The Smallest...A

Vertex	Shortest Dist From A
A	0
B	~~Infinity~~,4
C	Infinity
D	Infinity
E	Infinity
F	Infinity

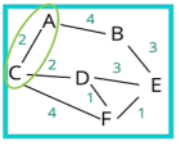
Visited:

```
[A]
```

Previous:

```
{
    A: null,
    B: A,
    C: null,
    D: null,
    E: null,
    F: null
}
```

  - Previous is to maintain where we came from



FIND THE SHORTEST PATH FROM A TO E

- Pick The Smallest...A

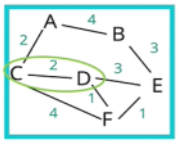
Vertex	Shortest Dist From A
A	0
B	~~Infinity~~,4
C	~~Infinity~~,2
D	Infinity
E	Infinity
F	Infinity

Visited:

```
[A]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: null,
    E: null,
    F: null
}
```



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...C

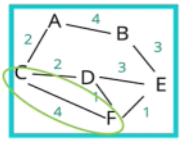
Vertex	Shortest Dist From A
~~A~~	0
B	~~Infinity~~,4
C	~~Infinity~~,2
D	~~Infinity~~,4
E	Infinity
F	Infinity

Visited:

```
[A,C]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: null,
    F: null
}
```



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...C

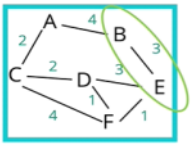
Vertex	Shortest Dist From A
~~A~~	0
B	~~Infinity~~,4
C	~~Infinity~~,2
D	~~Infinity~~,4
E	Infinity
F	~~Infinity~~,6

Visited:

```
[A,C]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: null,
    F: C
}
```



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...B

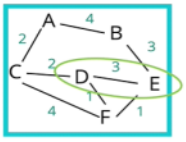
Vertex	Shortest Dist From A
~~A~~	0
B	~~Infinity~~,4
~~C~~	~~Infinity~~,2
D	~~Infinity~~,4
E	~~Infinity~~,7
F	~~Infinity~~,6

Visited:

```
[A,C]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: B,
    F: C
}
```



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...D

Vertex	Shortest Dist From A
~~A~~	0
~~B~~	~~Infinity~~,4
~~C~~	~~Infinity~~,2
D	~~Infinity~~,4
E	~~Infinity~~,7
F	~~Infinity~~,6

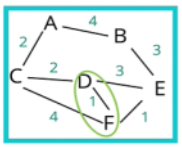
Visited:

```
[A,C,B]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: B,
    F: C
}
```

- We check from D -> E, since the total is 7 and for A -> E is already 7, we do not update where we came from



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...D

Vertex	Shortest Dist From A
~~A~~	0
~~B~~	~~Infinity~~,4
~~C~~	~~Infinity~~,2
D	~~Infinity~~,4
E	~~Infinity~~,7
F	~~Infinity~~,5

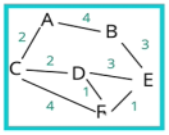
Visited:

```
[A,C,B]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: B,
    F: D
}
```

- We update the shortest distance from A->F from 6 to 5, since D->F takes the shortest distance of 5.
- So we update the previous node also that we reached F through D using the shortest distance of 5.



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...F

Vertex	Shortest Dist From A
~~A~~	0
~~B~~	~~Infinity~~,4
~~C~~	~~Infinity~~,2
~~D~~	~~Infinity~~,4
E	~~Infinity~~,7
F	~~Infinity~~,5

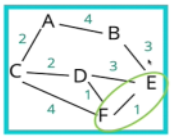
Visited:

```
[A,C,B,D]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: B,
    F: D
}
```

- Now, since we have visited A,C,B,D and as we are the finding the shortest distance between A and E, we move to F and not to E



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...F

Vertex	Shortest Dist From A
~~A~~	0
~~B~~	~~Infinity~~,4
~~C~~	~~Infinity~~,2
~~D~~	~~Infinity~~,4
E	~~Infinity~~,7
F	~~Infinity~~,5

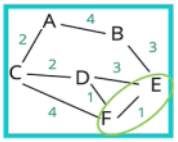
Visited:

```
[A,C,B,D]
```

Previous:

```
{
    A: null,
    B: A,
    C: A,
    D: C,
    E: B,
    F: D
}
```

- When we check by summing up the total for A->E through F, we get the total as 6.



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...F

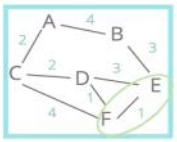
Vertex	Shortest Dist From A
~~A~~	0
~~B~~	~~Infinity~~,4
~~C~~	~~Infinity~~,2
~~D~~	~~Infinity~~,4
E	~~Infinity~~,~~7~~,6
F	~~Infinity~~,5

Visited:

```
[A,C,B,D]
```

Previous:

```
{
  A: null,
  B: A,
  C: A,
  D: C,
  E: F,
  F: D
}
```



FIND THE SHORTEST PATH FROM A TO E

Pick The Smallest...F

Vertex	Shortest Dist From A
~~A~~	0
~~B~~	~~Infinity~~,4
~~C~~	~~Infinity~~,2
~~D~~	~~Infinity~~,4
E	~~Infinity~~,~~7~~,6
F	~~Infinity~~,5

Visited:

```
[A,C,B,D,F]
```

Previous:

```
{
  A: null,
  B: A,
  C: A,
  D: C,
  E: F,
  F: D
}
```

- Since 6 is less than 7, we update the shortest path total and the previous where we came from to E.
- Shortest path: A->C->D->F->E (Total: 6)

- Naïve version of Priority queue

A simple PQ

```
class PriorityQueue {
  constructor(){
    this.values = [];
  }
  enqueue(val, priority) {
    this.values.push({val, priority});
    this.sort();
  };
  dequeue() {
    return this.values.shift();
  };
  sort() {
    this.values.sort((a, b) => a.priority - b.priority);
  };
}
```

Notice we are sorting which is O(N * log(N))

```
class PriorityQueue {
    constructor(){
        this.values = [];
    }
    enqueue(val, priority) {
        this.values.push({val, priority});
        this.sort();
    }
    dequeue() {
        return this.values.shift();
    }
    sort() {
        this.values.sort((a, b) => a.priority - b.priority);
    }
}
```

Dijkstra's Pseudocode

- This function should accept a starting and ending vertex
- Create an object (we'll call it distances) and set each key to be every vertex in the adjacency list with a value of infinity, except for the starting vertex which should have a value of 0.
- After setting a value in the distances object, add each vertex with a priority of Infinity to the priority queue, except the starting vertex, which should have a priority of 0 because that's where we begin.
- Create another object called previous and set each key to be every vertex in the adjacency list with a value of null
- Start looping as long as there is anything in the priority queue
  - dequeue a vertex from the priority queue
  - If that vertex is the same as the ending vertex - we are done!
  - Otherwise loop through each value in the adjacency list at that vertex
    - Calculate the distance to that vertex from the starting vertex
    - if the distance is less than what is currently stored in our distances object
      - update the distances object with new lower distance
      - update the previous object to contain that vertex
      - enqueue the vertex with the total distance from the start node

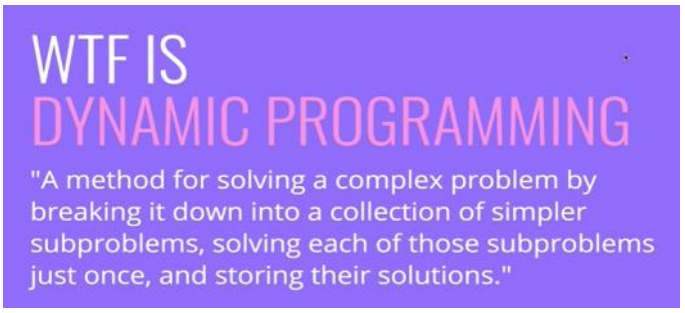
```
class WeightedGraph {
    constructor() {
        this.adjacencyList = {};
    }
    addVertex(vertex) {
        if (this.adjacencyList[vertex]) return;
        this.adjacencyList[vertex] = [];
    }
    addEdge(vertex1, vertex2, weight) {
        if (!this.adjacencyList[vertex1] ||
            !this.adjacencyList[vertex2] ||
            this.adjacencyList[vertex1].includes(vertex2)) return;
        this.adjacencyList[vertex1].push({node: vertex2, weight});
        this.adjacencyList[vertex2].push({node: vertex1, weight});
    }
```

```
    Dijkstra(start, finish) {
        let nodes = new PriorityQueue();
        let distances = {};
        let previous = {};
        let smallest;
        let path = [];
        // Setting up initial state
        for (let vertex in this.adjacencyList) {
            distances[vertex] = vertex === start ? 0 : Infinity;
            nodes.enqueue(vertex, distances[vertex]);
            previous[vertex] = null;
        }
        // Until nodes are left to be visited
        while (nodes.values.length) {
            smallest = nodes.dequeue().value;
            if (smallest === finish) {
                console.log('finish smallest:'+smallest);
                console.log('finish distances:'+JSON.stringify(distances));
                console.log('finish previous:'+JSON.stringify(previous));
                while (previous[smallest]) {
                    path.push(smallest);
                    smallest = previous[smallest];
                }
                break;
            }

            if (smallest || smallest !== Infinity) {
                for (let neighbor in this.adjacencyList[smallest]) {
                    // find neighboring node
                    let nextNode = this.adjacencyList[smallest][neighbor];
                    console.log('smallest:'+smallest);
                    console.log('this.adjacencyList[smallest]:'+JSON.stringify(this.adjacencyList[smallest]));
                    console.log('nextNode:'+JSON.stringify(nextNode));
                    // calculate new distance to neighboring nodes
                    let newDistance = distances[smallest] + nextNode.weight;
                    if (newDistance < distances[nextNode.node]) {
                        // updating new smallest distance to neighbor
                        distances[nextNode.node] = newDistance;
                        // updating previous with node from where we came from
                        previous[nextNode.node] = smallest;
                        console.log('distances:'+JSON.stringify(distances));
                        console.log('previous:'+JSON.stringify(previous));
                        nodes.enqueue(nextNode.node, newDistance);
                        console.log(nodes);
                    }
                }
            }
        }
        return path.concat(smallest).reverse();
    }
}

class PriorityQueue {
    constructor() {
        this.values = [];
    }
    enqueue(value, priority) {
        this.values.push({value, priority});
        this.bubbleUp();
        return this;
    }
    bubbleUp() {
        let index = this.values.length - 1;
        while (index > 0) {
            let parentIndex = Math.floor((index-1)/2);
            if (this.values[index].priority >= this.values[parentIndex].priority) break;
            [this.values[index], this.values[parentIndex]] = [this.values[parentIndex], this.values[index]];
            index = parentIndex;
        }
    }
    dequeue() {
        [this.values[0], this.values[this.values.length-1]] = [this.values[this.values.length-1], this.values[0]];
        const removedNode = this.values.pop();
        this.rearrangeQueue();
        return removedNode;
    }
```

```
    rearrangeQueue() {
        let index = 0;
        while (true) {
            let leftChildIndex = (2*index) + 1;
            let rightChildIndex = (2*index) + 2;
            let swapIndex = null;
            if (this.values[index]?.priority > this.values[leftChildIndex]?.priority) swapIndex = leftChildIndex;
            if (
                (swapIndex === null &&
                this.values[index]?.priority >= this.values[rightChildIndex]?.priority) ||
                (swapIndex !== null &&
                this.values[index]?.priority >= this.values[rightChildIndex]?.priority &&
                this.values[rightChildIndex]?.priority < this.values[leftChildIndex]?.priority)
               )
                swapIndex = rightChildIndex;
            if (swapIndex === null) break;
            [this.values[index], this.values[swapIndex]] = [this.values[swapIndex], this.values[index]];
            index = swapIndex;
        }
    }
}


let weightedGraph = new WeightedGraph();
weightedGraph.addVertex("A");
weightedGraph.addVertex("B");

weightedGraph.addVertex("C");
weightedGraph.addVertex("D");
weightedGraph.addVertex("E");
weightedGraph.addVertex("F");

weightedGraph.addEdge("A","B",4);
weightedGraph.addEdge("A","C",2);
weightedGraph.addEdge("B","E",3);
weightedGraph.addEdge("C","D",2);
weightedGraph.addEdge("C","F",4);
weightedGraph.addEdge("D","E",3);
weightedGraph.addEdge("D","F",1);
weightedGraph.addEdge("E","F",1);

weightedGraph.Dijkstra("A","E"); // ["A", "C", "D", "F", "E"]
```

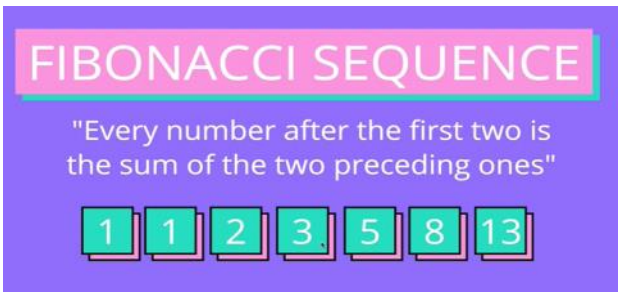
Dynamic Programming

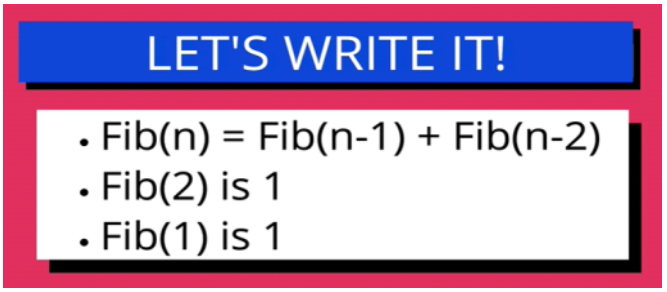
Thursday, March 11, 2021 5:16 PM

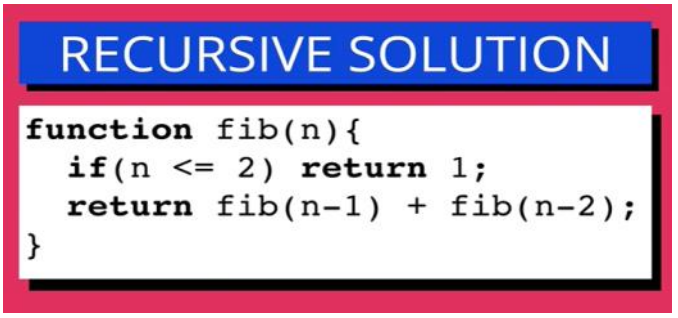
- 

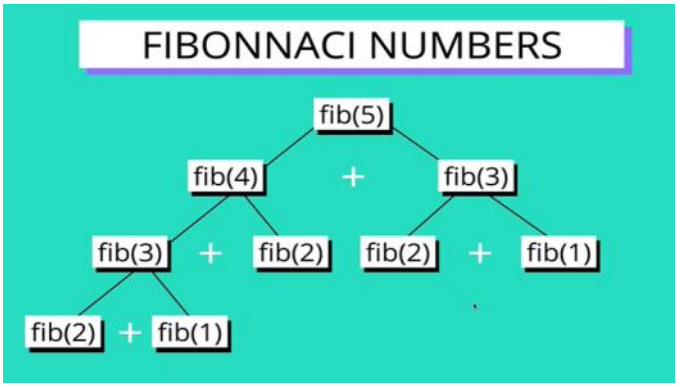
- 

    - Example

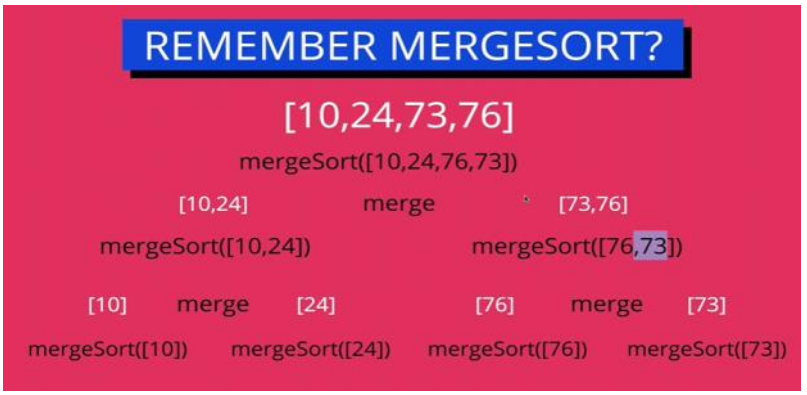
        - 

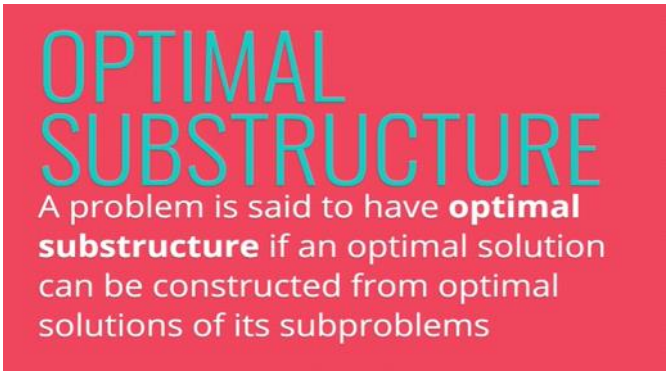
        - 

- 

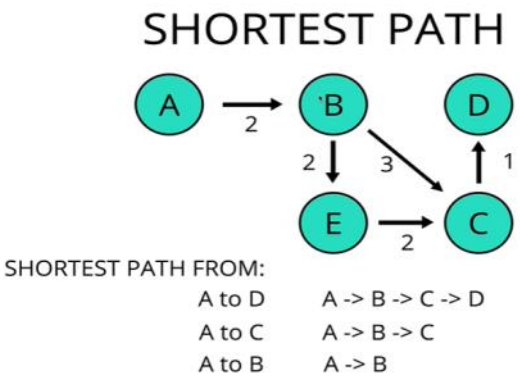
- 

  - Example of non-overlapping sub problem

- 

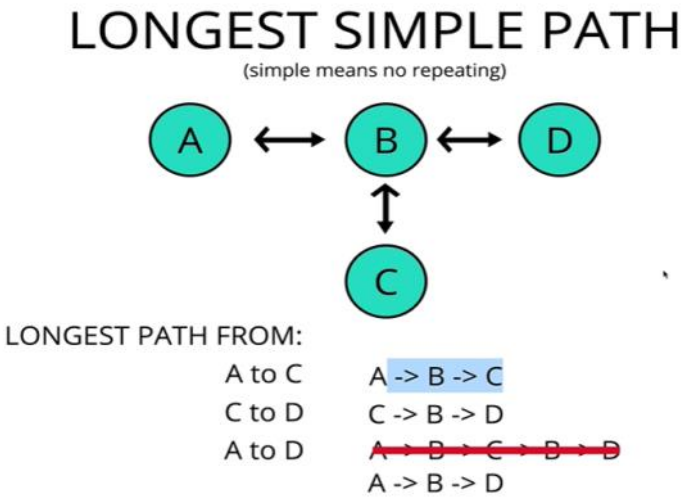
- 

  - Example of optimal sub structure



- If we find the shortest path from A -> D, it also solves the shortest path for the points which come along this path like in the example shown above (A->B->C->D also find shortest path for A->B->C and A->B)

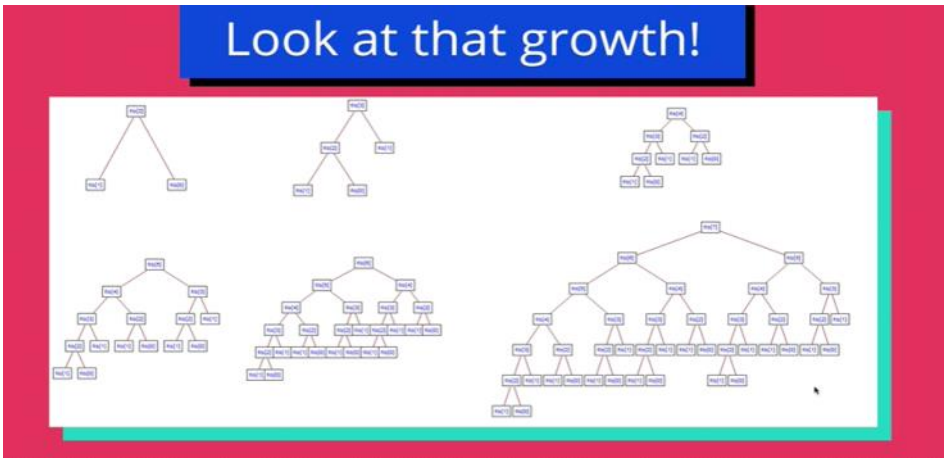
- Example of non-optimal sub structure

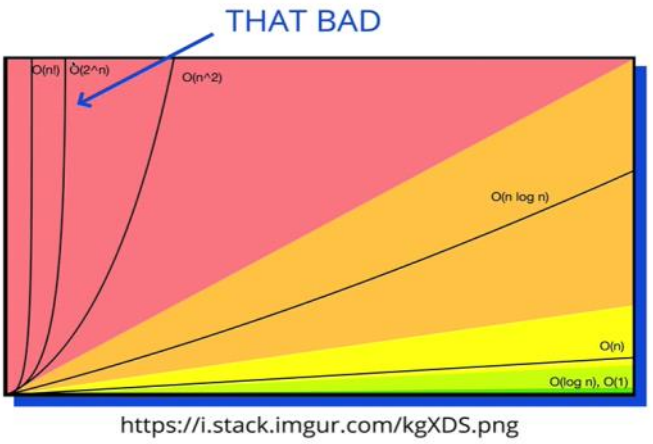


- He we cannot use optimal sub structure since we have to find the longest path between points in which we cannot use the solutions of the subproblems

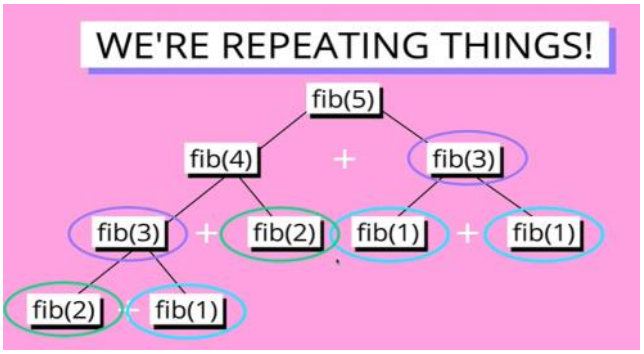
- Big O

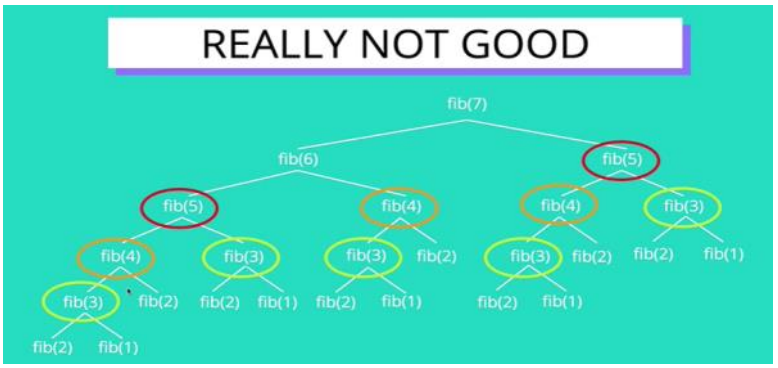
- For recursive implementation of fibonacci series, it grows worse as n increases



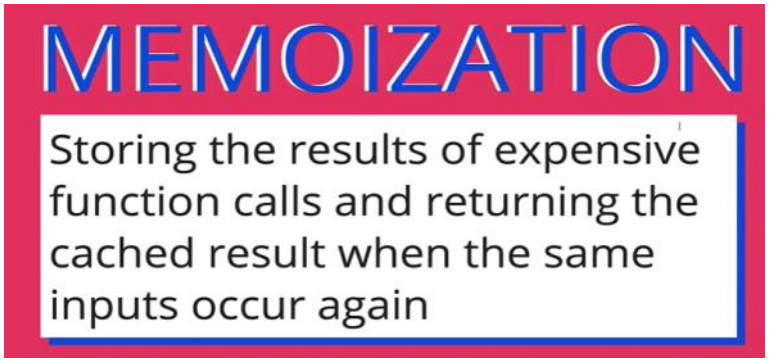
- 

https://i.stack.imgur.com/kgXDS.png
    - Approximately O(1.6^n)

- The reason on why it is so bad is that we repeating/recalculating everything over and over again
    - 

    - 

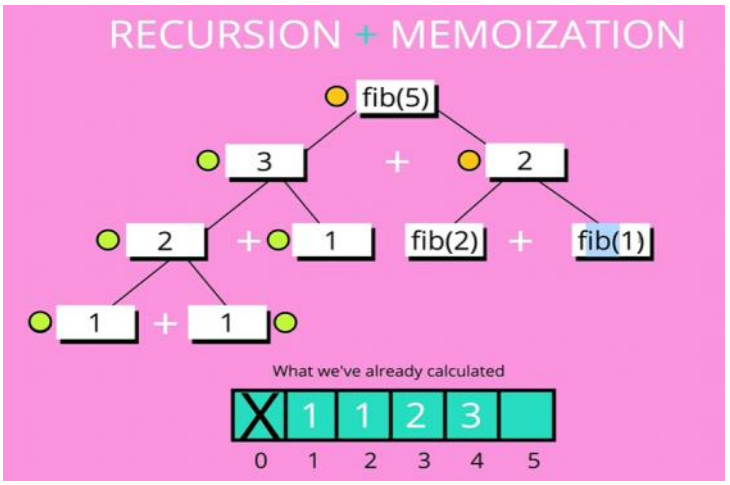
    - To solve this we can store the solutions which are already calculated
        - 

- 

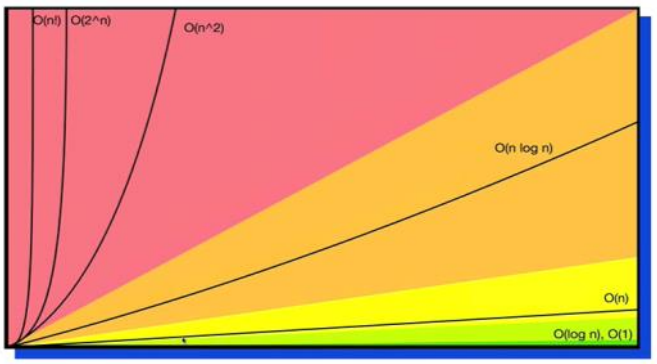

```
function fib(num, memo = {1: 1, 2: 1}){
    if (memo[num]) return memo[num];
    memo[num] = fib(num-1, memo) + fib(num-2, memo);
    return memo[num];
}
```

-

```
// fib(4) // 3
// fib(10) // 55
// fib(28) // 317811
// fib(35) // 9227465
fib(100) // 354224848179262000000
```

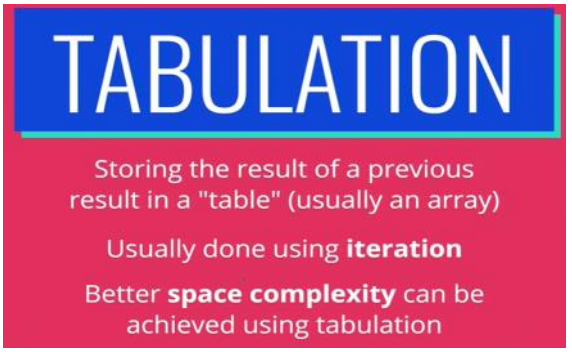
- 

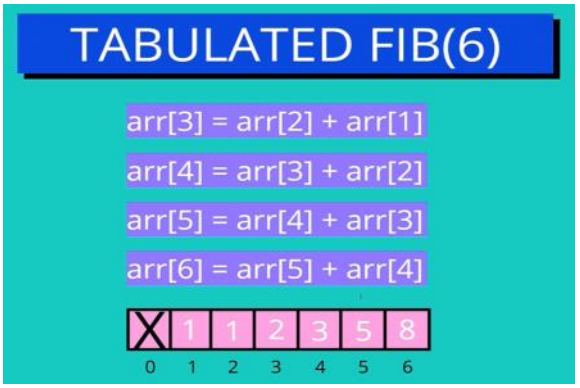
- O(N) is Big O with memoization
  - 

- 

    - When we want a fibonacci result for a bigger number(Ex.10000) using memoization, the only drawback is that we get the error "Maximum call stack exceeded!"

- Bottom up approach (Tabulation)
    - 

    - 

    - 

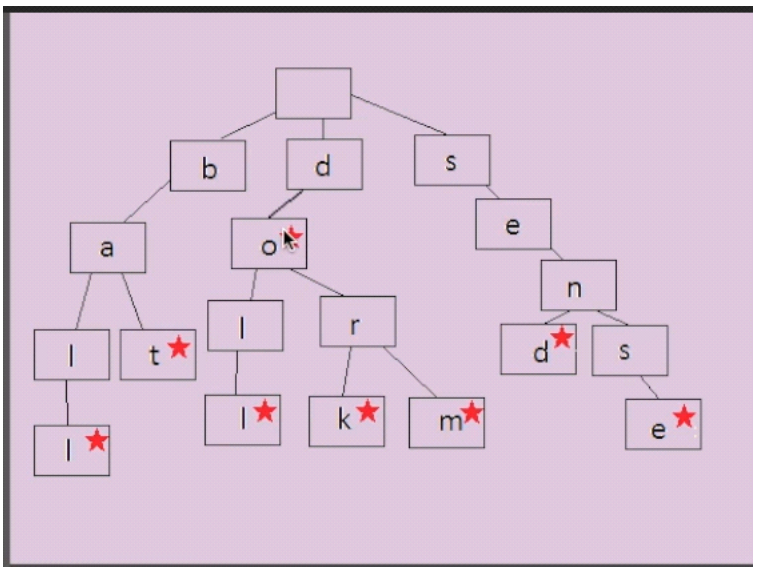
```
function fib(num){
    if (num <= 2) return 1;
    let fibSeries = [0, 1, 1];
    for (let i = 3; i <= num; i++) {
        fibSeries[i] = fibSeries[i-1] + fibSeries[i-2];
    }
    return fibSeries[num];
}

// fib(4) // 3
// fib(10) // 55
// fib(28) // 317811
// fib(35) // 9227465
fib(100) // 354224848179262000000
```

- When we want a fibonacci result for a bigger number(Ex.10000) using memoization, the only drawback is that we get the error "Maximum call stack exceeded!" but when we use tabulation for a fibonacci result for a bigger number it works as expected.

- Big O is O(n) for tabulation as well.

Trie Data Structure

Monday, June 28, 2021 10:37 AM

- 

    - Start for a character represents end of the word

Hash Tables

Wednesday, August 18, 2021 3:17 PM

- A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

- Hash table is used to implement associative arrays, mapping of key-value pairs.

-

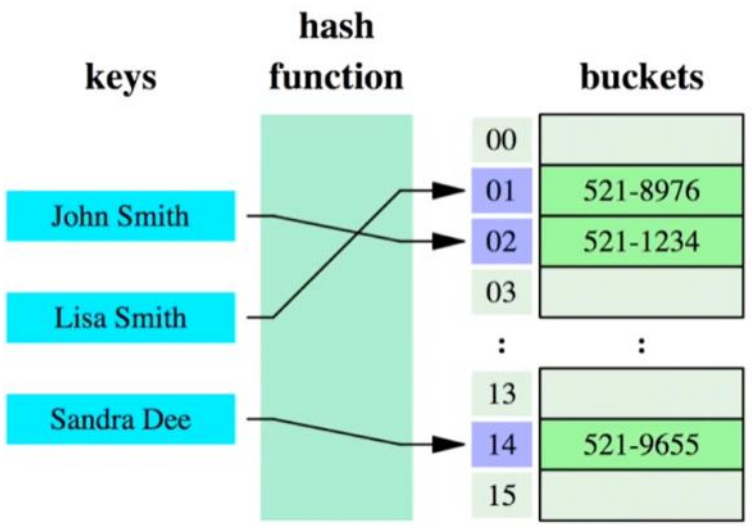
Hash table

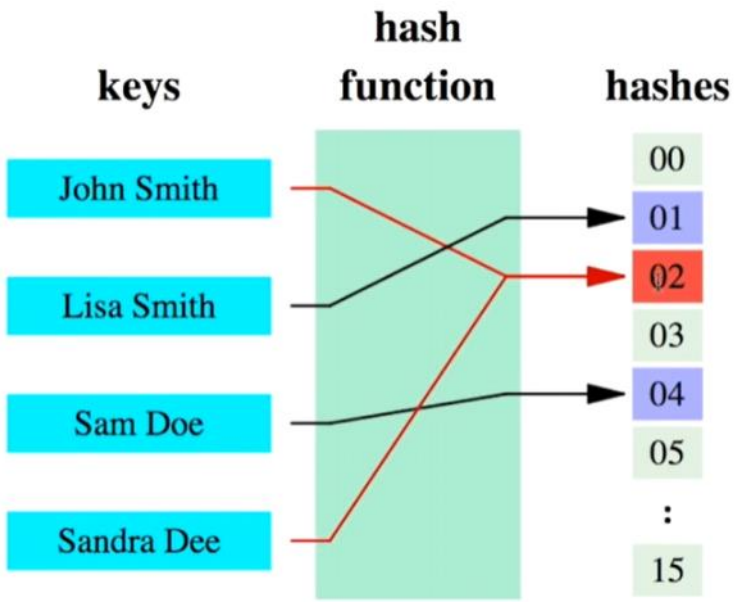
Type	Unordered associative array
Invented	1953

Time complexity in big O notation

Algorithm	Average	Worst Case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

-





- The above marked in red color is called a Collision where two keys have been hashed to the same number.