**Object Oriented Programming in JavaScript**

As JavaScript is widely used in Web Development, in this article we would explore some of the **Object Oriented**mechanism supported by **JavaScript** to get most out of it.

There are certain features or mechanisms which makes a Language Object Oriented like:

* **Object**
* **Classes**
* **Encapsulation**
* **Inheritance**

**Object**– An Object is a **unique** entity which contains **property** and **methods**.

An Object is an **instance** of a class. Objects are everywhere in JavaScript almost every element is an Object whether it is a function,arrays and string.

For example “car” is a real life Object, which have some characteristics like color, type, model, horsepower and performs certain action like drive. The characteristics of an Object are called as Property, in Object Oriented Programming and the actions are called methods.

**Note:** A Method in javascript is a property of an object whose value is a function.  
Object can be created in two ways in JavaScript:

Using an **Object Literal**

//Defining object

let person = {

    first\_name:'Raman',

    last\_name: 'Sinha',

    //method

    getFunction : function(){

        return (`The name of the person is

          ${person.first\_name} ${person.last\_name}`)

    },

    //object within object

    phone\_number : {

        mobile:'996834555',

        landline:'1234'

    }

}

console.log(person.getFunction());

console.log(person.phone\_number.landline);

Using an **Object Constructor:**

//using a constructor

function person(first\_name,last\_name){

   this.first\_name = first\_name;

   this.last\_name = last\_name;

}

//creating new instances of person object

let person1 = new person('Raman','Sinha');

let person2 = new person('Monika','Sharma');

console.log(person1.first\_name);

console.log(`${person2.first\_name} ${person2.last\_name}`);

Using **Object.create() method:**

The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.

// Object.create() example a

// simple object with some properties

const coder = {

    isStudying : false,

    printIntroduction : function(){

        console.log(`My name is ${this.name}. Am I

          studying?: ${this.isStudying}.`)

    }

}

// Object.create() method

const me = Object.create(coder);

// "name" is a property set on "me", but not on "coder"

me.name = 'Raman';

// Inherited properties can be overwritten

me.isStudying = 'False';

me.printIntroduction();

**Classes**–

Classes are **blueprint** of an Object. A class can have many Object, because class is a **template** while Object are **instances** of the class or the concrete implementation.  
Before we move further into implementation, we should know unlike other Object Oriented Language there is **no classes in JavaScript** we have only Object.

To be more precise, JavaScript is a prototype based object oriented language, which means it doesn’t have classes rather it define behaviors using constructor function and then reuse it using the prototype.

**Example:**  
Lets use ES6 classes then we will look into traditional way of defining Object and simulate them as classes.

// Defining class using es6

class Vehicle {

  constructor(name, maker, engine) {

    this.name = name;

    this.maker =  maker;

    this.engine = engine;

  }

  getDetails(){

      return (`The name of the bike is ${this.name}.`)

  }

}

// Making object with the help of the constructor

let bike1 = new Vehicle('Maruti', 'Suzuki', '1600cc');

let bike2 = new Vehicle('Honda', 'City', '1200cc');

console.log(bike1.name);    // Suzuki

console.log(bike2.maker);   // City

console.log(bike1.getDetails());

**Traditional Way.**

// Defining class in a Traditional Way.

function Vehicle(name,maker,engine){

    this.name = name,

    this.maker = maker,

    this.engine = engine

};

Vehicle.prototype.getDetails = function(){

    console.log('The name of the bike is '+ this.name);

}

let bike1 = new Vehicle('Maruti', 'Suzuki', '1600cc');

let bike2 = new Vehicle('Honda', 'City', '1200cc');

console.log(bike1.name);

console.log(bike2.maker);

console.log(bike1.getDetails());

**Encapsulation** –

The process of **wrapping property and function**within a **single unit** is known as encapsulation.  
Let’s understand encapsulation with an example.

//encapsulation example

class person{

    constructor(name,id){

        this.name = name;

        this.id = id;

    }

    add\_Address(add){

        this.add = add;

    }

    getDetails(){

        console.log(`Name is ${this.name},Address is: ${this.add}`);

    }

}

let person1 = new person('Raman',31);

person1.add\_Address('Delhi');

person1.getDetails();

Sometimes encapsulation refers to **hiding of data** or **data Abstraction** which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but their are no such access modifiers in JavaScript but their are certain way by which we can restrict the scope of variable within the

**Class/Object.**

**Example:**

// Abstraction example

function person(fname,lname){

    let firstname = fname;

    let lastname = lname;

    let getDetails\_noaccess = function(){

        return (`First name is: ${firstname} Last

            name is: ${lastname}`);

    }

    this.getDetails\_access = function(){

        return (`First name is: ${firstname}, Last

            name is: ${lastname}`);

    }

}

let person1 = new person('Raman','Sinha');

console.log(person1.firstname);

console.log(person1.getDetails\_noaccess);

console.log(person1.getDetails\_access());

**Inheritance**–

It is a concept in which some property and methods of an Object is being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Object inherits Object i.e. certain features (property and methods)of one object can be reused by other Objects.  
Lets’s understand inheritance with example:

//Inhertiance example

class person{

    constructor(name){

        this.name = name;

    }

    //method to return the string

    toString(){

        return (`Name of person: ${this.name}`);

    }

}

class student extends person{

    constructor(name,id){

        //super keyword to for calling above class constructor

        super(name);

        this.id = id;

    }

    toString(){

        return (`${super.toString()},Student ID: ${this.id}`);

    }

}

let student1 = new student('Raman',22);

console.log(student1.toString());

**Note:** The Person and Student object both have same method i.e toString(), this is called as **Method Overriding**. Method Overriding allows method in a child class to have the same name and method signature as that of a parent class.  
In the above code, super keyword is used to refer immediate parent class instance variable.

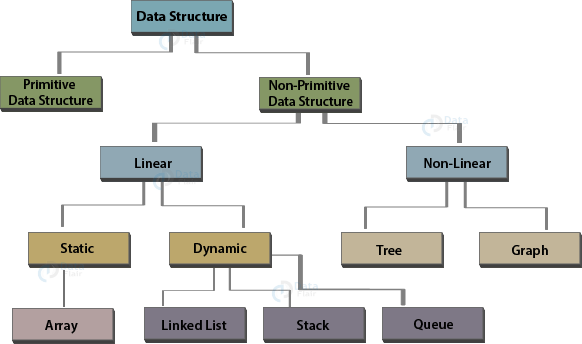
## What are JavaScript Data Structures?

A JavaScript Data Structure is a specific technique to organize and store data in a computer so that we can access and modify it efficiently. More accurately, it is a collection of data values, the relationships among them, and the functions or operations that we can apply to the data.

Some of the advantages of using data structures are as follows:

* They provide an easy way to manage large datasets.
* They simplify the processing of data on a computer.
* These are essential for designing efficient program algorithms.

But, only advanced users can make changes to data structures. So if you run into any problem, you’ll need an expert’s advice. The diagram below illustrates the types of data structures available in every programming language, including JavaScript. You will see some other data structures including **trie**, but these are advanced topics and it’s not relevant to mention them here.



### Types of Data Structures in JavaScript

We discussed various primitive data structures that JavaScript provides in our prior tutorial on ***Data Types in JavaScript***. This tutorial will focus on non-primitive data structures. They are as follows:

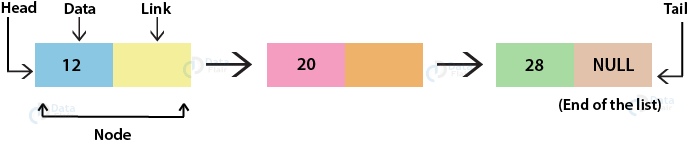
* A**linear data structure** traverses its elements sequentially. We can reach only one data item directly. To access other elements, you need the help of that base element.
* Unlike linear data structures, **non-linear data structures** don’t traverse in a sequence. Every data item connects with numerous other items, reflecting specific relationships.
* **Static data structures** have a fixed memory size, that is, you need to state the maximum size of the structure well in advance. We cannot allocate the memory later. The example of static data structure is arrays. We learned about them in the DataFlair’s previous tutorial on JavaScript Arrays.
* **Dynamic data structures** differ from static data structures in the way that we can modify the memory size allocated to it. These include queue, stack and linked lists.

The following table states the difference between static and dynamic data structures very clearly.

|  |  |
| --- | --- |
| **Static Data Structures** | **Dynamic Data Structures** |
| Memory is either wasted (underflow) or is insufficient (overflow). | The amount of memory allocated varies,  preventing memory wastage. |
| Fast access to the data elements since the memory location is decided while writing the program. | Slower access to each element since memory  allocation happens at run time. |
| Structures have fixed size, making them predictable and easier to work with. | Structures vary in size, thus complex  algorithms are necessary to deal with  them. |
| The relationship between different elements remains constant. | The relationship between the elements  changes as the program executes. |

### Linked List

A linked list maintains a list in the memory. The script has the base address of the first element, that contains the link to the next element in the list. You can only access an element (except the base element) using the previous element.



Linked lists store these elements in what we call a **node**. A node comprises of a**data value** and the link (address) to the next element. The first node is the **head** while the last node is the **tail** of the list. The tail, as shown in the diagram, contains the value **NULL** in place of a link. The code below creates a node:

**Code:**

1. <html>
2. <body>
3. <script>
4. class Node { //defining a JavaScript class
5. **constructor**(data) { //constructor method
6. this.data = data; //data value of the node
7. this.next = null; //link of the next node
8. }
9. }
10. const head = new **Node**(12); //creating a node
11. </script>
12. </body>
13. </html>

Let’s add another node in our code with the following statement at the end of the script:

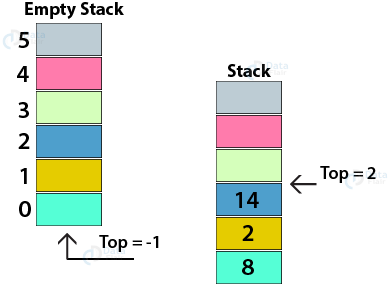
1. head.next = new **Node**(45);

The **output** in your console window will look something like this:

Note that now the**next** of the first element contains the link of the subsequent element.

There is a more efficient way to add several nodes to a linked list effectively, but they are not part of basic JavaScript. As you grow as a programmer, you will gain a better understanding of how it works.

### Stack



A stack is an ordered list which follows **LIFO** (last in first out) algorithm. You can access the elements of a stack from only a single end, called **top**. A very common example of a stack, we see in our daily lives, is the stack of chairs. You can only access the top chair at a time. The only way to reach a chair in the middle is by going through all the chairs above it. In our very first tutorial on ***Introduction to JavaScript***, we learned about the call stack which works with the instructions we want to execute. Stacks are crucial for programming, so you must know them well because you are going to work with them a lot while developing.

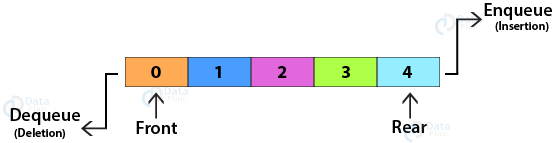
We can implement a stack using a static data structure (array) or a linked list. Both approaches have their own advantages but as a beginner, we will implement stack using an array. The code below creates a new stack and pushes two elements in the stack using the predefined **push()** function:

**Code:**

1. <html>
2. <body>
3. <script>
4. let stack = []; //create empty stack using array
5. //push
6. stack.**push**(65);
7. stack.**push**(10);
8. </script>
9. </body>
10. </html>

I removed the top element from my stack directly from the console using the **pop()** function. You can add the statement in your source code as well. The **output** will look something like this:

### 3. Queue



A queue is another type of ordered list which follows **FIFO** (first in first out) algorithm. It has two ends, **front** (elements added) and **rear** (elements removed). A common example you may see is a queue of cars in a one-way lane. The first car that enters the lane is the first one to exit. Also, the car in the middle cannot exit the queue until all the cars before it.

The process of inserting an element in the queue is **enqueueing** while removing an element is **dequeuing.** Similar to stacks, we can implement queues with the help of both an array and a linked list. Both approaches have their pros and cons, as explained in the difference between static and dynamic data structures. The array implementation of a queue is clear in the example below. Note that I performed dequeue directly on the Browser Console Window.

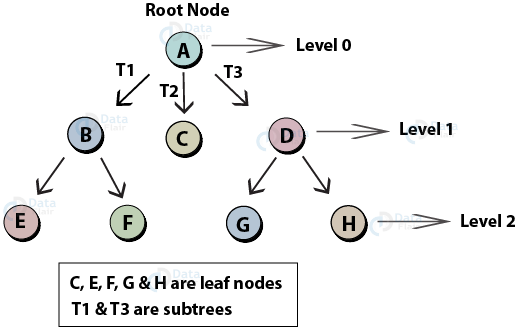
**Code:**

1. <html>
2. <body>
3. <script>
4. let queue = []; //create an empty queue
5. //enqueue
6. queue.**push**(65);
7. queue.**push**(10);
8. </script>
9. </body>
10. </html>

**Output:**

### [queue op](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2019/07/queue-op.jpg)

### 4. Tree



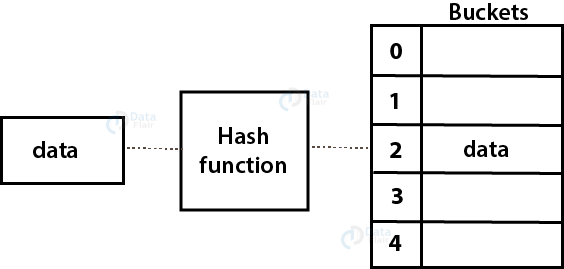
A JavaScript tree is a special data structure that implements the hierarchical tree structure with a root node, child and parent nodes and leaf nodes represented as a set of linked nodes. A tree data structure is a collection of nodes, starting with a root node, with data values in each of the nodes along with the reference to the child nodes. The root node has no parent node and the leaf nodes have no child nodes. If the child node of a parent node has one or more child nodes, it is a subtree. In the diagram above, the nodes **BEF** (T1) and **DGH** (T2) form two subtrees.

### 5. Graph

A graph is a group of a finite number of vertices and edges that connect these vertices. The edges can be **‘directed’** (directed graph) and **‘undirected’** (undirected graph). Unlike trees, who maintain a parent-child relationship between their nodes (vertices), the nodes of the graph maintain a complex relationship among them.

### directed undirected graph - JavaScript Data Structures

### 6. Hash table

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2019/08/hash-table.png)

A hash table, or hash map, is a data structure that implements associative arrays since it maps keys to values. It uses a hash function to determine the index of the data value in the array (bucket). These buckets help to identify the storage location of the data we want. The earlier version of JavaScript didn’t support associative arrays, so there was no built-in hash table available. But as the developers introduced new features, JavaScript gained additional functionality and addition of associative arrays was one of them.

## Sorting Algorithm

1. Bubble Sort - explanation with tested code
2. Selection Sort - explanation with tested code
3. Insertion Sort - explanation with tested code
4. Merger Sort - explanation with tested code
5. Quick Sort - explanation with tested code
6. Heap Sort - explanation with tested code

## Bubble Sort

**How it works:**

**step-1:** you compare the first item with the second. If the first item is bigger than the second item. you swap them so that the bigger one stays in the second position.

**step-2:**And then compare second with third item. if second item is bigger than the third, we swap them. otherwise, they stayed in their position. Hence, the biggest among first three is in the third position.

**step-3:**we keep doing it. until we hit the last element of the array. In that way we bubble up the biggest item of the array to the right most position of the array.

**step-4:** Look at the inner loop in the code below.

**step-5:** We repeat this process, starting from the last item of the array. look at the outer loop in the code below. We do this way, so that after finishing the first inner loop, the biggest one will be in the last item of the array.

**step-6:** and then we move backward inside the outer loop.

same thing is going on....

function bubbleSort(arr){

var len = arr.length;

for (var i = len-1; i>=0; i--){

for(var j = 1; j<=i; j++){

if(arr[j-1]>arr[j]){

var temp = arr[j-1];

arr[j-1] = arr[j];

arr[j] = temp;

}

}

}

return arr;

}

bubbleSort([7,5,2,4,3,9]); //[2, 3, 4, 5, 7, 9]

bubbleSort([9,7,5,4,3,1]); //[1, 3, 4, 5, 7, 9]

bubbleSort([1,2,3,4,5,6]); //[1, 2, 3, 4, 5, 6]

**complexity:**

## Selection Sort

**how does it works:**

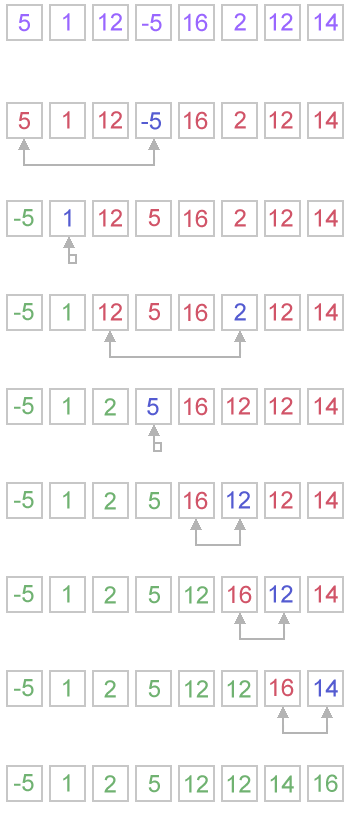
This is very simple.

Go through the array, find the index of the lowest element swap the lowest element with the first element.

Hence first element is the lowest element in the array.

Now go through the rest of the array (excluding the first element) and find the index of the lowest and swap it with the second element.

thats how it continues to select (find out) the lowest element of the array and putting it on the left until it hits the last element.



function selectionSort(arr){

var minIdx, temp,

len = arr.length;

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

minIdx = i;

for(var j = i+1; j<len; j++){

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

minIdx = j;

}

}

temp = arr[i];

arr[i] = arr[minIdx];

arr[minIdx] = temp;

}

return arr;

}

**complexity:**

## Insertion sort

**How it works:**Imagine you are playing cards.

Somebody is giving you cards one by one. When you are receiving card, you are planning to put them in a way so that the smaller one is on the left. This means you want to insert them in a sorted way

**step-1:** If the first card you are getting is 5. Just hold the card in your hand. you dont have to do anything.

**step-2:** If the second card is 2, you want to put it before 5 so that the two cards you have are sorted. When you are putting the card with number 2 at the left, you are changing the position of the card 5 from first position to second position. And then first position becomes available and you put 2 there.

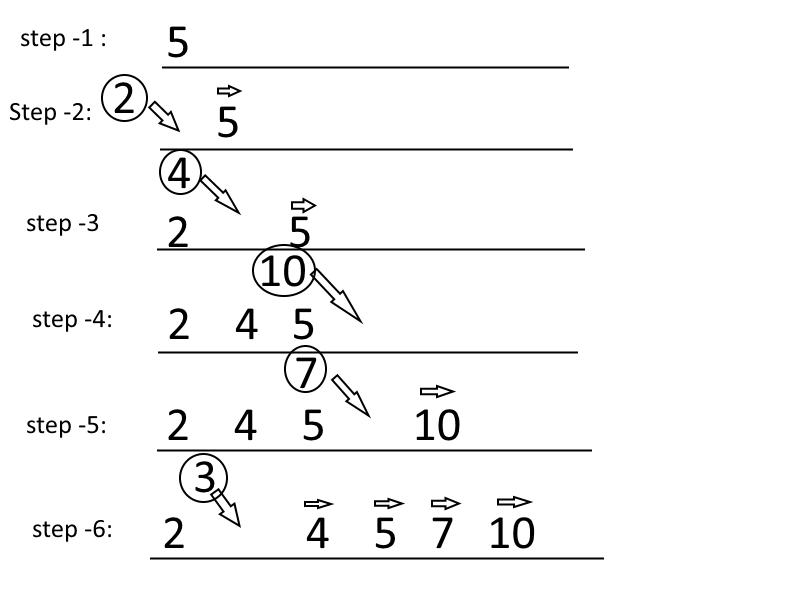
**step-3:** If the third card is 4. you will start from second position. In the second position, you have card 5 which is bigger than 4. Hence you will move 5 to the third position. The next card to the left is 2 which is smaller than 4. Hence, you wont move 2. And you will insert card 4 in the second position.

**step-4:** Then you got 10. It is bigger than the previous card which is 5. Hence, you just add it at the last position.

**step-5:** The next card is 7. You just move the position of the card 10 to the right and insert card 7.

**step-6:** If the last card is 3. You will have to move 10 to the right as it is bigger than 3. and then you check with the next card to the left it is 7 which is bigger than 3. you move it to the right. similarly, you move 5, 4 to the right. And put the number 3 before 2 as 2 is smaller than 3.

congrats. you are done.



**Code Insertion sort**: Code is similar to the card and image above. It starts with the second element. Pick the second element to be inserted and then compare to the previous element. If the first one is bigger, move the first one to second position and second one at first.

Now first and second item is sorted.

Then, pick the third element and check whether the second element is bigger than the third. keep going similar way until you hit the first element or a element smaller than the element you are comparing with. When you get an item smaller than the picked item, you insert it.

super easy.

function insertionSort(arr){

var i, len = arr.length, el, j;

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

el = arr[i];

j = i;

while(j>0 && arr[j-1]>toInsert){

arr[j] = arr[j-1];

j--;

}

arr[j] = el;

}

return arr;

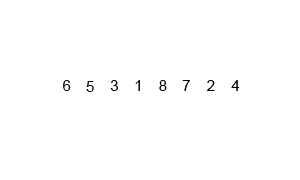
}

**complexity:** the worst case O(n2) and best case O(n)

## Merge Sort

its a divide and conquer type algorithm.

just break down your array into small and small pieces and until you have one items in each pieces. then merge together by comparing them. If you still have hard time to figure out what i am talking about, look at merge sort gif taken from wikipedia



**Code Merge Sort:** Merge sort has two parts. Main part does divide or breaks down and second part is merging/combining parts. At the time of combining, parts are combined together.

**Divide:** the first function named as mergeSort is actually a divide function. where an array is divided into two.

**merge:** this is just merging two sorted array. Just be careful this two array could be in different size

function mergeSort(arr){

var len = arr.length;

if(len <2){

return arr;

}

var mid = Math.floor(len/2),

left = arr.slice(0,mid),

right =arr.slice(mid);

//send left and right to the mergeSort to broke it down into pieces

//then merge those

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

}

function merge(left, right){

var result = [],

lLen = left.length,

rLen = right.length,

l = 0,

r = 0;

while(l < lLen && r < rLen){

if(left[l] < right[r]){

result.push(left[l++]);

}

else{

result.push(right[r++]);

}

}

//remaining part needs to be addred to the result

return result.concat(left.slice(l)).concat(right.slice(r));

}

ref: merge sort

## Quick sort

**how does it works:**

**Step-1:** You have to pick a pivot. This could be randomly selected or the middle one. Here we select the last element of the array.

**Step-2:** Put all the items smaller than the pivot value to the left and larger than the pivot value to the right.

**Step-3:**Repeat the step-2 for both left and right side of the pivot (pick a pivot, put all item smaller than the pivot to the left and larger on the right)

#### Explain the code

**Call Quick sort:** Pass the array and pass left and right to the quickSort function. For the first call, left would be the index of the first element which is 0 and right would be the index of the last element which would be length -1.

**Select Pivot:** We select pivot as the last index of the array.

**Call Partition function:** After calculating the pivot, we send the pivot to the partition function. In the partition function we pass array, pivot index, left and right.

**partitionIndex:** In the partition function, we keep move all the items smaller than the pivot value to the left and larger than pivot value to the right. We have to keep track of the position of the partition. so that we can split the array into two parts in the next step. This tracking of the index where we partition the array is done by using partitionIndex variable. the initial value is left.

**Swap function:** This is just a helper function to swap values of the array.

**move elements:** we start a for loop from the left, and if the values is smaller than the pivot values we swap it with the position of the partitionIndex and increase the value of the partitionIndex. If the value is bigger, we don't do anything. We keep going until the element before the last element (remember last element is the pivot)

**place pivot** After moving all the smallest element to the left, we swap the last element (pivot value) with the partitionIndex. By doing this, the pivot sits where it suppose to sit when the full array is sorted. As all elements left to it smaller and all element right to it is bigger. End of the function partition, return the partitionIndex

**Repeat the process:**Now come back to quickSort function. when you get the partitionIndex, apply quickSort for the left side of the array and right side of the array. keep doing it until left is smaller than right.

function quickSort(arr, left, right){

var len = arr.length,

pivot,

partitionIndex;

if(left < right){

pivot = right;

partitionIndex = partition(arr, pivot, left, right);

//sort left and right

quickSort(arr, left, partitionIndex - 1);

quickSort(arr, partitionIndex + 1, right);

}

return arr;

}

function partition(arr, pivot, left, right){

var pivotValue = arr[pivot],

partitionIndex = left;

for(var i = left; i < right; i++){

if(arr[i] < pivotValue){

swap(arr, i, partitionIndex);

partitionIndex++;

}

}

swap(arr, right, partitionIndex);

return partitionIndex;

}

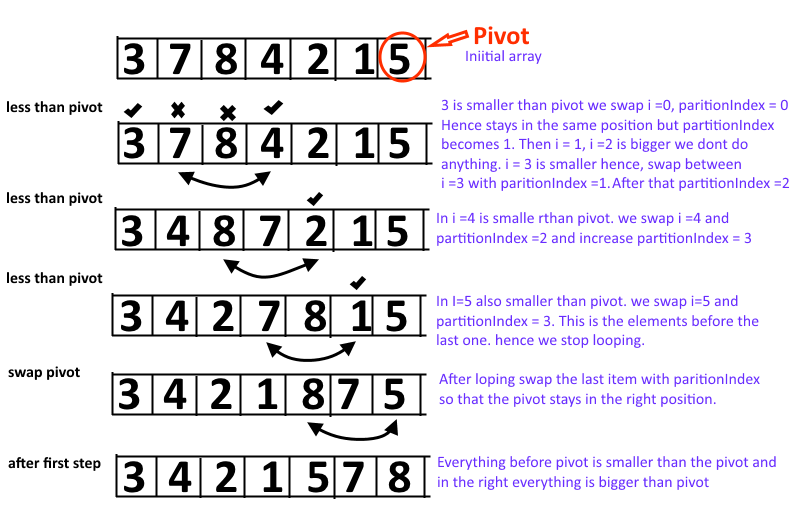
function swap(arr, i, j){

var temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}



quickSort([11,8,14,3,6,2,7],0,6);

//[2, 3, 6, 7, 8, 11, 14]

quickSort([11,8,14,3,6,2,1, 7],0,7);

//[1, 2, 3, 6, 7, 8, 11, 14]

quickSort([16,11,9,7,6,5,3, 2],0,7);

//[2, 3, 5, 6, 7, 9, 11, 16]

ref: quick sort

**time complexity:** talk about time complexity of quick sort

## Heap sort

**how does it works:**first step is to build an heap. That is done in the heapify function. Hepaify put the largest element at the root.

## Add explanation for shiftDown

## Add graphical description

Try to understand from here. It doesn't have any code.

function heapSort(arr){

var len = arr.length,

end = len-1;

heapify(arr, len);

while(end > 0){

swap(arr, end--, 0);

siftDown(arr, 0, end);

}

return arr;

}

function heapify(arr, len){

// break the array into root + two sides, to create tree (heap)

var mid = Math.floor((len-2)/2);

while(mid >= 0){

siftDown(arr, mid--, len-1);

}

}

function siftDown(arr, start, end){

var root = start,

child = root\*2 + 1,

toSwap = root;

while(child <= end){

if(arr[toSwap] < arr[child]){

swap(arr, toSwap, child);

}

if(child+1 <= end && arr[toSwap] < arr[child+1]){

swap(arr, toSwap, child+1)

}

if(toSwap != root){

swap(arr, root, toSwap);

root = toSwap;

}

else{

return;

}

toSwap = root;

child = root\*2+1

}

}

function swap(arr, i, j){

var temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

**Question:** Between merge and heap sort, which one you will choose?

**compare with other:**compare heap sort with others

## Bucket Sort

bucket sort or bin sort. you partition the whole array into small buckets. put numbers in the related bucket and then sort each bucket by another algorithm or using bucket sort.

for example, you have an array where numbers between 1 to 100 is for a million times.

If you want to sort it by any of the method we talked before, will be time consuming.

Bucket sort would be convenient here

## Shell Sort

**how does it works:**it takes a gap. for example gap of five. in that case it will take all the elements at 1, 6, 11, 16... and will sort them among themselves. Hence this gapped array will be sorted. secondly it will take 2, 7, 12, 17th positioned element and will sort among themselves

second time. it will take all in three positioned. like 1, 4, 7, 10, 13, 16, 19 and will sort among themselves. after this all these will be sorted. and then goes to 2, 5, 8, 11, 14, 17, 20

Then it will do 1 gap sorting...

shell sort is rarely used in application as it performs more operations and have higher cache miss ratio than quick sort.

ref: wiki: shell sort

## Pigeonhole Sort

also known as count sort (not counting sort). you have an array. and each element has a key value pair. you iterate over the original array and put each element based on the key to its small array (pigeon hole). and then iterate over the pigeonhole array in order.

**Counting sort:**counting sort works same way, but it returns number of elements for each key other than the original elements. very thoughtful name, indeed.

ref: wiki pigeonhole sort

## Binary Tree sort

## Radix Sort

its a non comparative integer sorting algorithm. it compares the integer based on the position of the number (significance).

there are two radix sort. LSD (least significant digit) radix sort and MSD (most significant digit) radix sort

## cocktail sort

also known as bidirectional bubble sort, cocktail shaker sort, shaker sort, ripple sort, shuffle sort, shuttle sort.

difference with bubble sort is that this sort in both direction each pass. Its difficult to implement.

## Other sorting algorithms

**patience sorting:** sorting based on solitaire card game. you draw card either put in a new pile or existing pile. when you are putting card on a pile, a higher value card will go on top

**comb sort:**it is an improvement on bubble sort. the inner loop of bubble sort is modified so that gap between swapped elements goes down for each iteration of the outer loop by using a shrink factor. wiki comb sort the usual shrink factor is 1.3 and the gap (>=1) is calculated by Math.floor(lenght/1.3)

**Spaghetti sort:** it takes linear time but need parallel processor to sort. read it to understand

**Intro sort:**combines both quick sort and heap sort. so that worst and average becomes O(nlongn). it starts with quick sort and switch to heap sort if recursion depth increases based on the number of elements to be sorted.

**Timsort:**this is a hybrid of merge and insertion sort. try to find subset of data that is already sorted. and use that data to sort rest efficiently. and then merged by merge sort. this algorithm is primarily used in python. This actually invented by Tim peters to use in python.

**Block sort:**runs at least two merge operation with one insertion sort. wiki: block sort

**library sort:** it is an insertion sort with gaps in the array to improve insertion. wiki: library sort

**Tournament sort:**Its a selection sort with a priority queue to find the next element to sort. wiki: tournament sort

**others:**Gnome sort, unshuffle sort, Strand sort, smooth sort