The Big O objects and Arrays.

OBjects are unorderd datastructure, Key value pairs.
datastored in Key value Pairs

```
let manager ={
    firstName: "Bulla",
    isManager: true,
    favoritNumber:[69,46,28,40]
    }
```

when to use objects

- when you dont need order
- when you need fast access/insertion and removal

Object are Very fast

Big O of objects.

insertion - O(1)
Removal - O(1)
Searching - O(N)
Access - O(1)

when you dont need any ordering, objects are an excellent Choice

We will learn more in Hashmap

The Big O object methods.

object.keys(manager) object.keys - O(N)
object.values(manager) object.values - O(N)
object.entries(manager) object.entries - O(N)
hasownProperty - O(1)
manager.hasOwnProperty("firstName") → True ← it have firstname

ARRAYS ~~ordered list~~

ordered list.

```
let names = ["billu", "choka", "Pungi"]
              0         1        2     ← Index

let values = [true, {}, [], 2, "bkws"];
               ↓    ↓   ↓   ↓    ↓
          boolean object array number string.
                               or
                               int
```

When to use arrays.

- when you need order.
- when you need fast access / insertion and removal (Short of....)

Big O of Arrays.

Insertion - It depends... ← where we are inserting.
Removal - It depends...
Searching - O(N) ← discuss in searching section.
Access - O(1) ← super fast.

when we want to acces a data it's fast

like names [2] ⇐ its fast when you have 10000 element and ask for 9997 javascript dont go counting all the way to 9997 and access every element and get ou what we need, i have shortcut when we have index number we can directly jump to that data and that fast. No matter how long data is.

Insertion ← if we are inserting element at the last then it have no problem (then big O will be)
O(1) ← hypothetical,

Problems comes in when we have to insert it starting then it going to messup (because of indexing)
it will re-index every single one
O(n)

Sames goes for removing from beginning we have to re-indexing every single one . O(n)

[Try to avoid insertion or removal from the beginning if it is possible]
← Be avoid it dont need to.

Push and pop are fastest
shift and unshift slowest.

Search grows O(n) because it will search all the data in array and it depends on array length.
learn more in searching session.

Big O of Array operations

NO reindexing involved ↙
- push - O(1)
- pop - O(1)

reindexing involves and grows
- Shift - O(N)
- unshift - O(N)

general O(N)
- concat - O(N)
- slice - O(N)
- splice - O(N)

sort session → • Sort - O(N * log N)

- forEach/map/filter/reduce/etc - O(N)

~~thr~~ let arr = [1,2,3,4]

remove
arr.Pop() ← it will pop one item from last
arr.Push(5) ← it will add this 5 to the array for at l
arr.Shift() ← it will remove item from start
Add ← arr.unshift(0) ← it will add item to start

let arr2 = [20,39 40]

let mergearr = arr.Concate (~~arr~~, arr 2)
merge

·if want to add more array you can add like (, arr3) like this

To get specific slice of arr we use slice () method
→ it dosnt mutate the original slice

let slice = arr.slice(1, 3) //(2,3) op
start ↙ ↑ end
start number will get sliced } or index
end number will be there }

or we can only give ~~to~~ start number.

let sliceStart = arr.slice (2) //[3,4]

(all codes are on git ~~hoop~~ hub / object and Arrays folder)