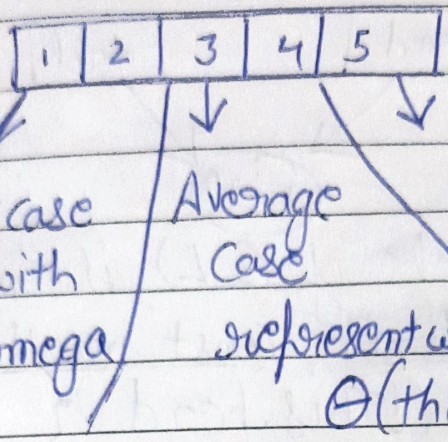
DSA with JavaScript

✱ Big O



⇒ well technically there is no Best case or average case,

⇒ so when we measure Big O, we always measuring worst case.

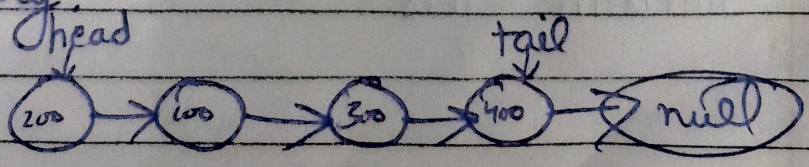
✱ Big O(n)

funct

✱ Linked Lists

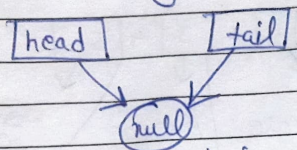
Are not placed contigius in memory as compare to Array

⇒ whereas array are placed contigues in memory.



☆ Push node

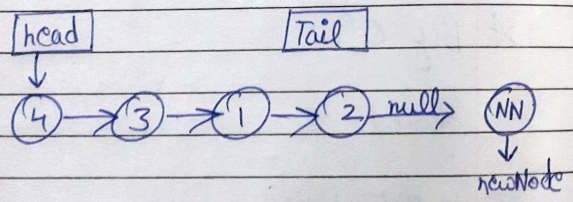
(i) if there is empty linked List :-



(i) First check if (LL) is empty or not.

```
const newNode = newNode(value)
if (!this.head){
    this.head = newNode
    this.tail = newNode
}
```

(ii) if (Linked lists is not empty) than




```
else {
    this.tail.next = newNode
    this.tail = newNode
}
this.length ++;
return this
```

☆ Create a class Node with a constructor the value & next properties of the node

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

☆ Create a class LinkedList with a constructor that initializes the head, tail and length properties of the Linked lists

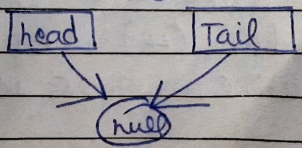
```
class LinkedList {
    constructor(value){
    const newNode = new Node(value);
    this.head = newNode
    this.tail = newNode
    this.length ++;
    }
}
```

☆ Pop node

→ There are 3 case :-

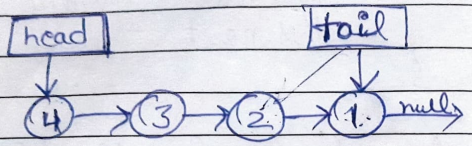
(i) if there is no node in (LL)
(ii) if there is number of node in (LL)
(iii) if there is only one node in (LL)

(i) →



```
if (!this.head){
    return undefined;
}
```

(ii)




```
let temp = this.head
let prev = this.head
while(temp.next){
    prev = temp
    temp = temp.next
}
this.tail = prev
this.tail.next = null
this.length--;
if(length===0){
    this.head = null
    this.tail = null
}
return this
```

(iii)

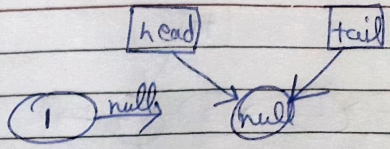
✱ Insert at beginning

⇒ There 2 case:-

(i) Insert when (LL) is empty

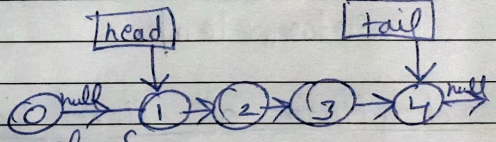
(ii) Insert when there is node

(i)




```
if (!this.head){
    this.head = newNode
    this.tail = newNode
}
```

(ii)




```
else{
    newNode.next = this.head
    this.head = newNode
}
```

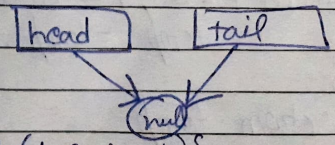
✱ Deletion at Beginning

⇒ There is 2 case:-

(i) when there (LL) is empty.

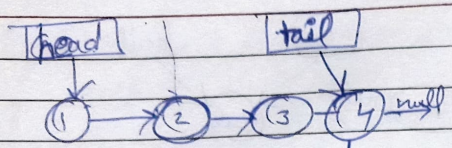
(ii) when there @ node in (LL).

(i)



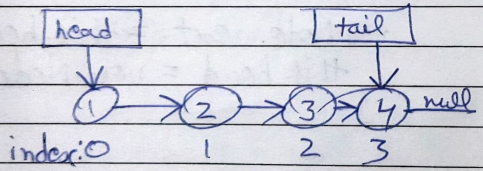

```
if(!this.head){
    return undefined
}
```

(4)



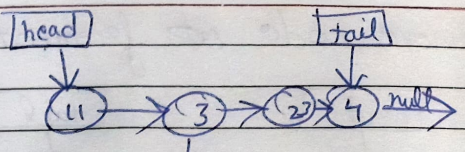

```
let temp = this.head
this.head = this.head.next
temp.next = null
if (this.length === 0){
    this.tail = null
}
return temp
```

☆ SearchAnNode at Linked List



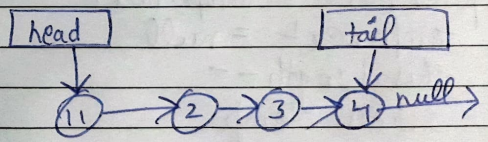

```
if(index<0 || index >= this.length){
    return undefined
}
let temp = this.head
for(let i=0; i<index; i++){
    temp = temp.next
}
return temp
```

☆ Change the Value at index in (LL):-



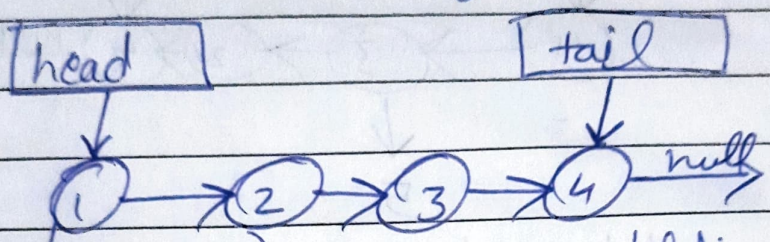

```
let temp = this.SearchAnNode(index)
if (temp)
    temp.Value = Value
    return true
}
return false
```

☆ Insert Node at Any index in (LL):-




```
if (index === 0 ){
    return this.insertAtBeggining(value)
}else if (index === this.length){
    return this.push(Value)
}else if (index<0 || index.>this.length){
    return false
}
const newNode = newNode(Value)
const temp = this.SearchANNode(index-1)
newNode.next = temp.next
temp.next = newNode
```

☆ Remove node at from at any index




```
if(index===0) return this.deletionAtBeggining()
if(index===this.length-1)
    return this.pop();
if(index<0 || index>=this.length){
    return 'Error: Index out of Bound'.

const prev = this.serchAnNode(index-1)
const temp = prev.next
 prev.next = temp.next
  temp.next = null
  this.length--
  return temp
```