$T(N) = 2*T(N/2) + O(N)$

H.W ⇒ Do substitution, expand and find TC

⇒ TC ⇒ $O(N \log N)$

i) logic

ii) write recursive code

iii) write TC relationship

iv) expand & TC

$T(N) = 2*T(N/2) + O(N)$ ✓

```
→ func(N) {
      func(N/2)
      fun(N/2)
      for(i=0→N-1) {
      }
  }
```

$$\underbrace{N \to N/2 \to N/4 \to N/8 \to N/16 \to 1}_{\log N \text{ times func call}}$$

So, each func call has O(N) for loop.

total TC ⇒ $O(N \log N)$

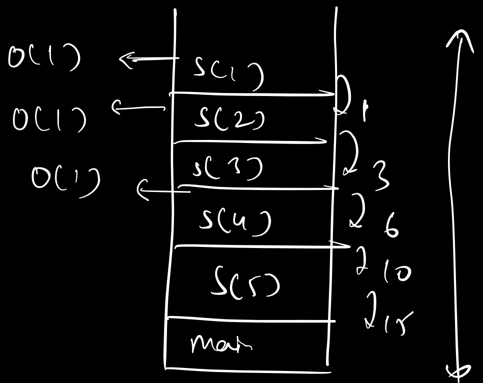
⇒ Space complexity for recursion

sum(N)

sum(5)

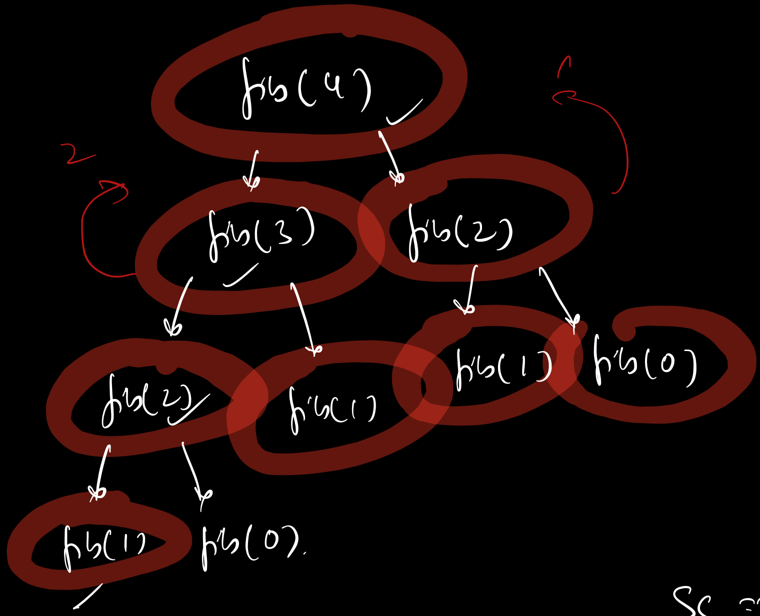
maxm size of call stack

SC ⇒ O(N)



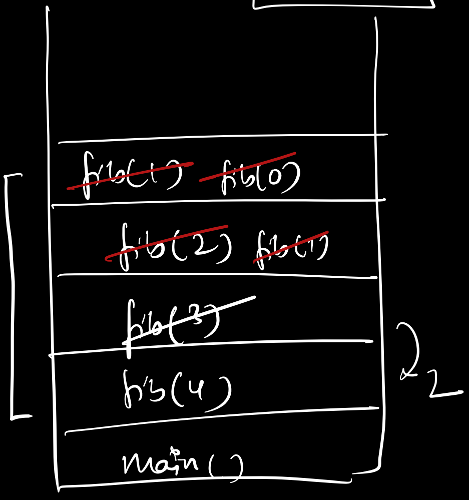
fibonacci

fib(N) = fib(N-1) + fib(N-2)



// fib(3) = fib(2) + fib(1)

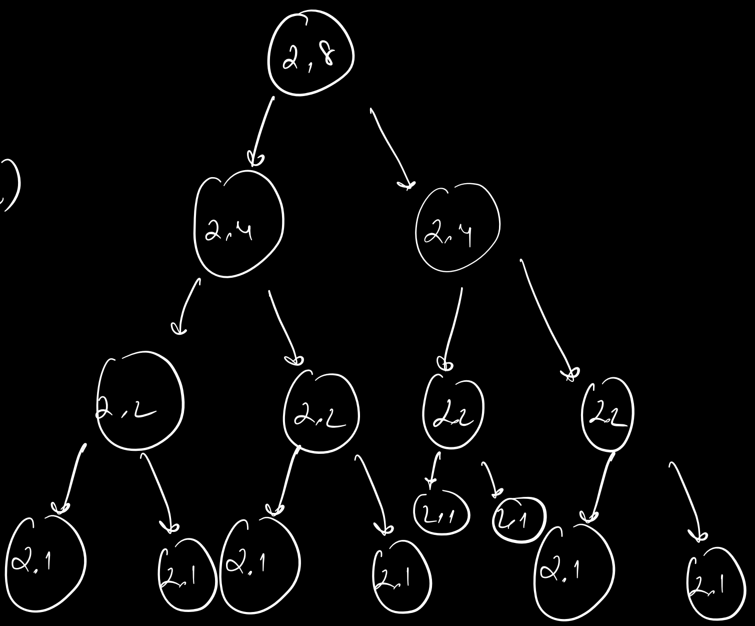
1 1

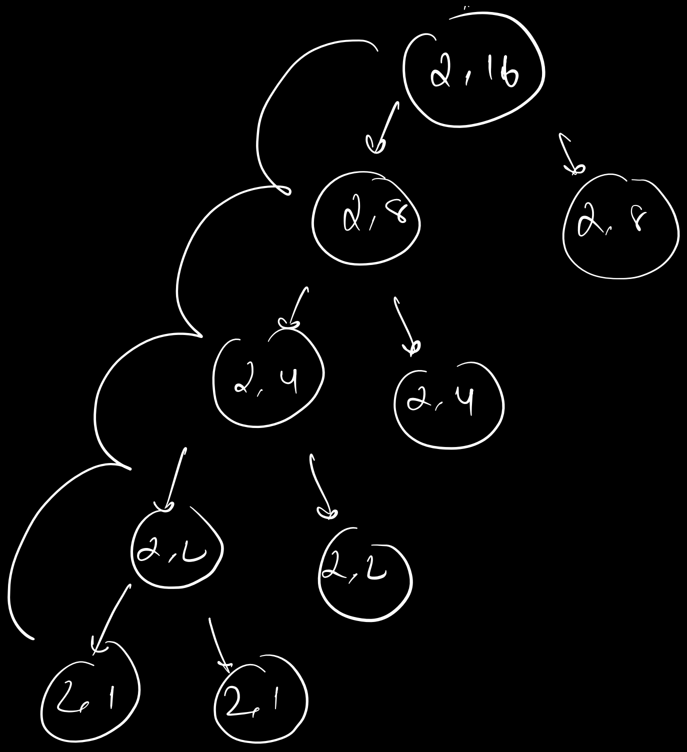


SC => O(N)

```
pow(N) {

    pow(N/2) * pow(N/2)

}
```

SC => O(log N)





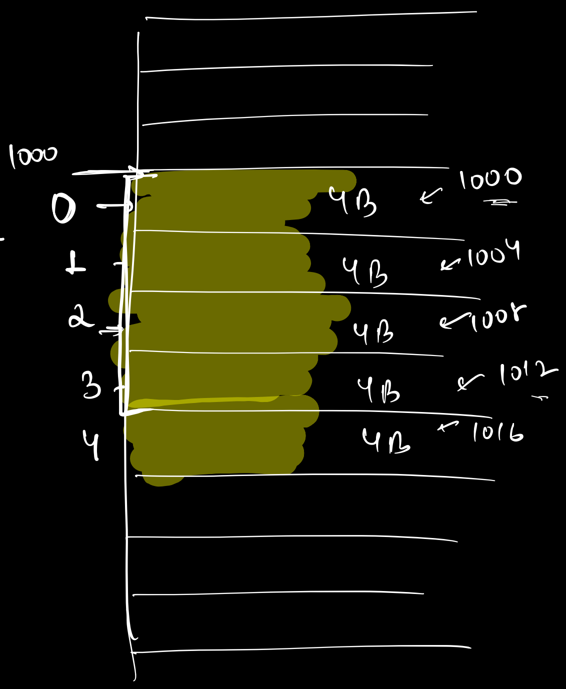
$\log_2 16 \Rightarrow \boxed{4} + 1$

Linkedlist

arrays

O(1) accessing any element

int arr[5] => size is fixed
↓
4B



int arr[5]
↓
stores memory add for 0th idx

* array is defined as contiguous memory

⇒ working of arraylist

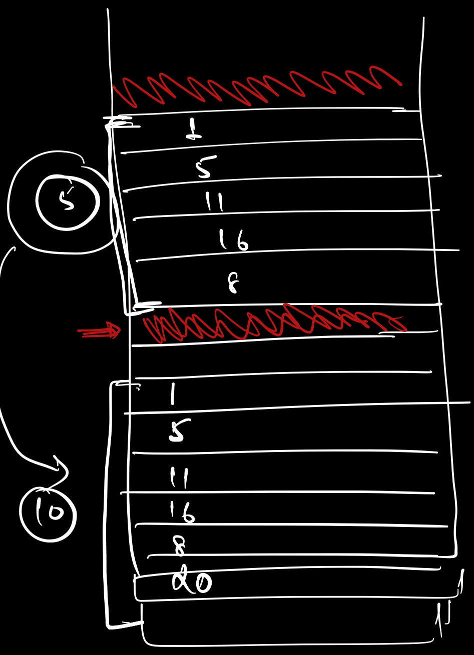
uses an array internally

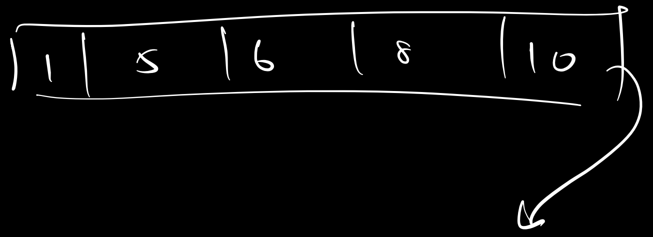
→ initial size → 5

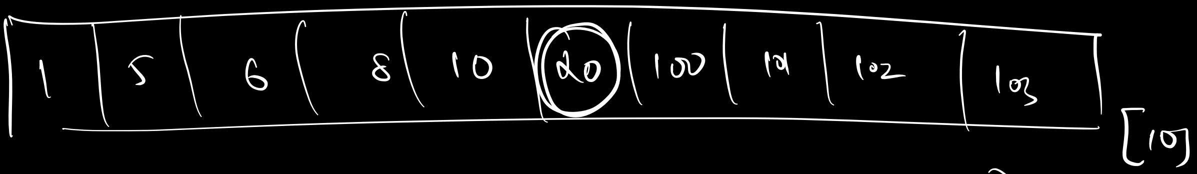
→ loadfactor → 1 ←

doubles

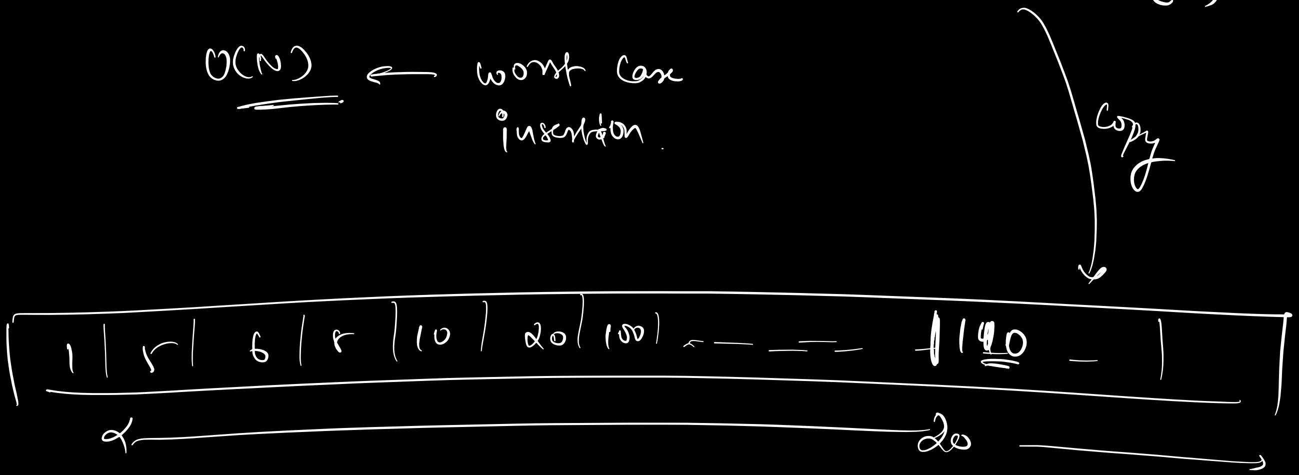
Arraylist<Integer> = new ArrayList<>()







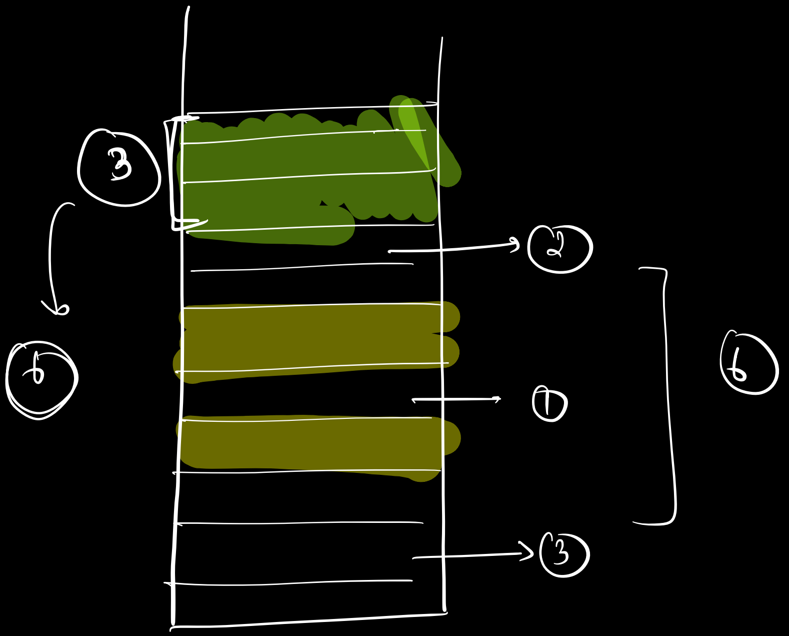
O(N) ← worst case insertion



Arraylist → insert
read
update } O(1) [amortised]

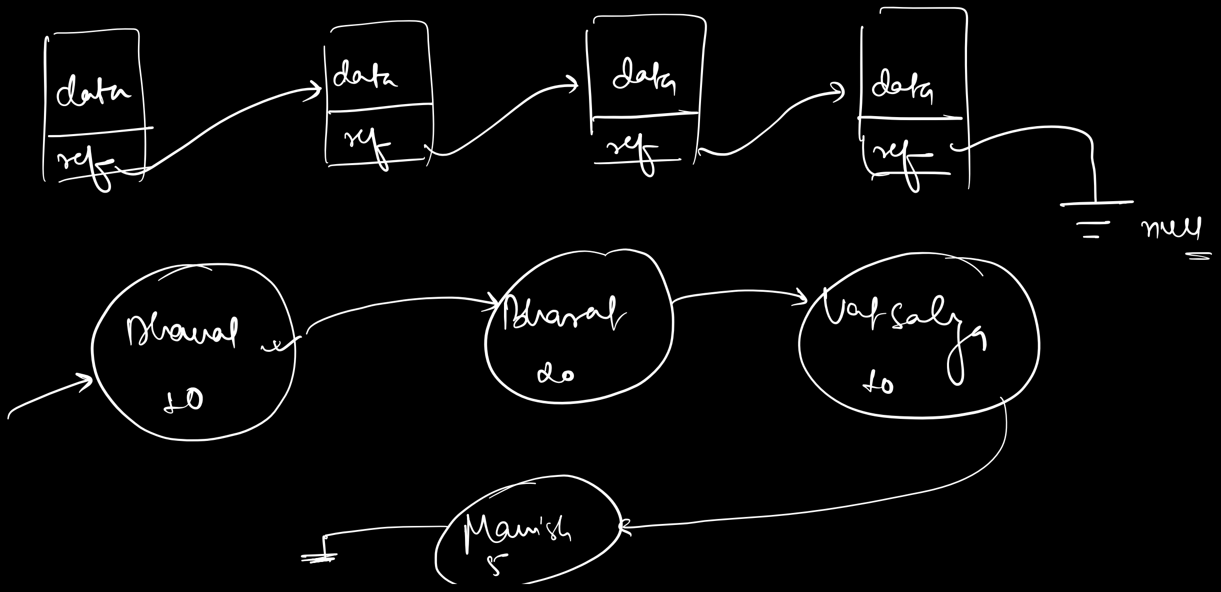
plastic cans → 20 l → 100l -

machine → 10L



* even though we have memory, we are not able to use it.

⇒ How LL works?

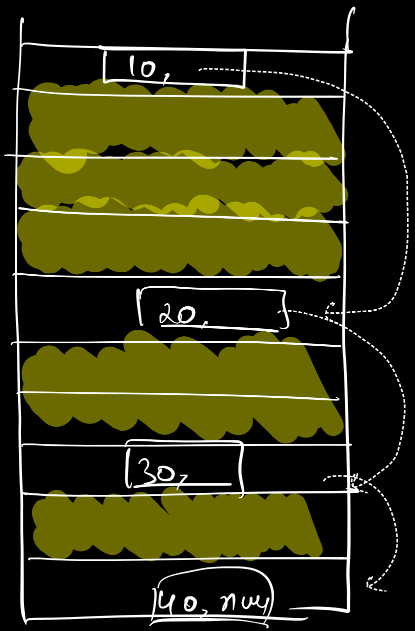


4 integers [10,20,30,40]

array ?

arraylist ?

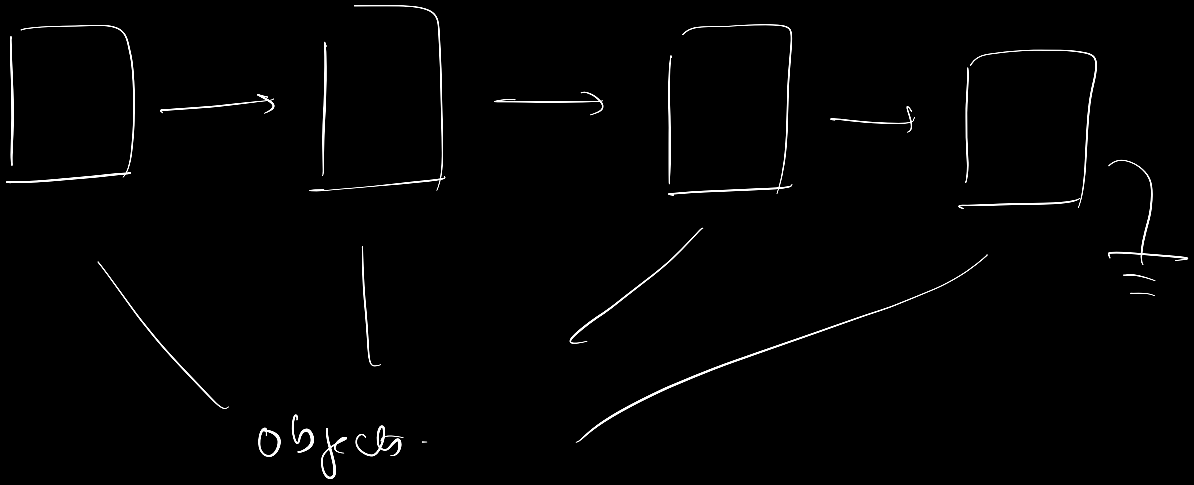
Linkedlist



Node

[data] → int, String, char, Hashmap, Array

ref



```
Class Node {
        int data;
        Node next;
}
```

→ data

→ ref. variable to next node.

Java

```
Node head = new Node();
```

head.data = 0/null/ -   —

head.next = - — —

Constructor

```
public Node(int a){
    data = a
    next = null
}
```

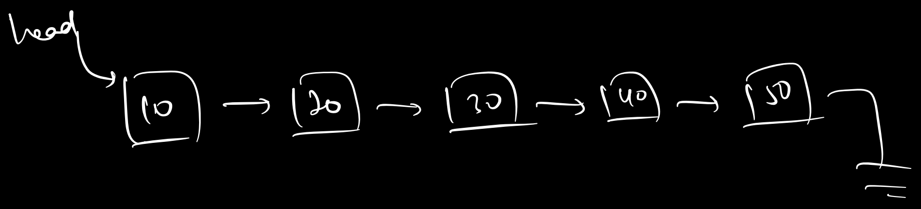
```
Node head = new Node(10);
```

head.data = 10.

head.next = null

Q1. Find length of a LL, given LL

i/p ⇒ head



O|P => 5

```
int getlength (Node head){
        int len = 0;  Node temp = head;
        while( temp != null ){
                len++;
            temp = temp.next;
        }
      return len;
}
```

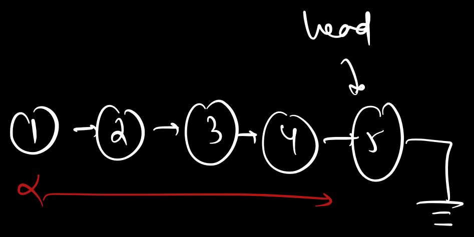
l = ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5

TC => O(N)

SC => O(1)

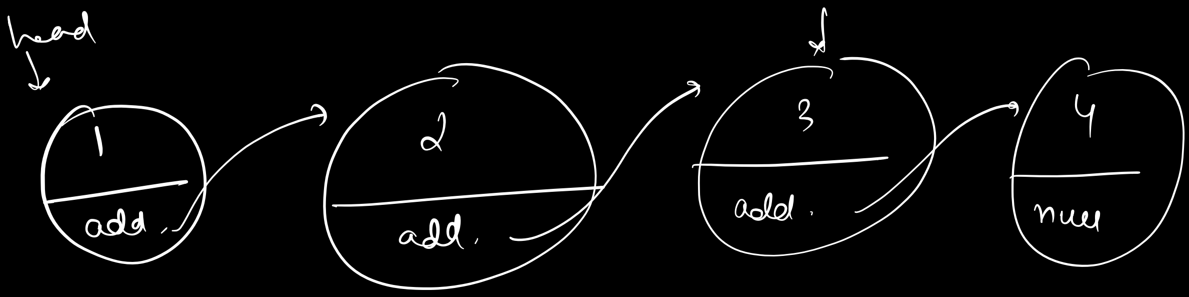
O|P = 5

```
int getlength (Node head){
      int len = 0;
      while( head.next != null){
            len++;
          head = head.next;
      }
   return len;
}
```



l = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

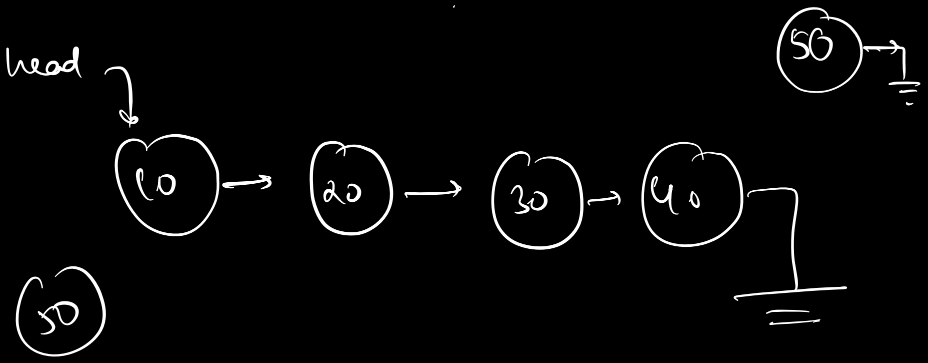
O|P = 4



* access randomly → LL is not a good option

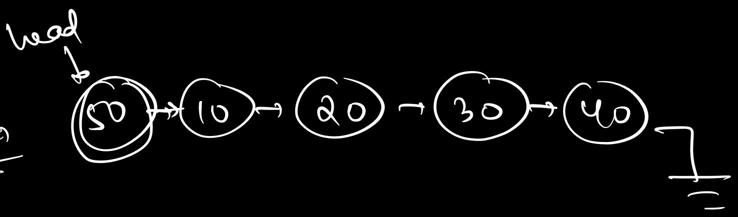
Q2. Given a LL, add data in front of LL

I/p ⇒



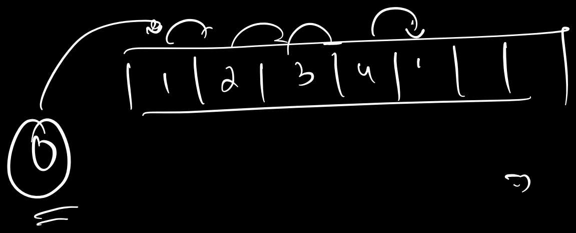
insert

O/p ⇒



```
Node newNode = new Node(50); ←
        data = 50
        next = null.

newNode.next = head;

head = newNode
```

adding element at front => LL => O(1)



=> array => O(N)

=> use case

Elastic Search
Solr
Lucene

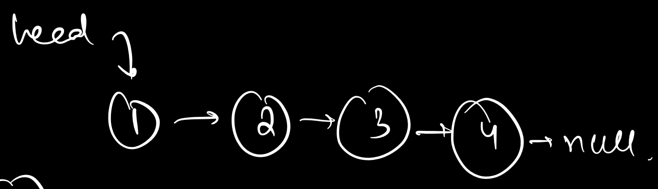
→ special types of LL [Skip List]

iphone	doc1	doc2	doc3	doc10	
Samsung	doc 4	dos	doc 6		
phone	doc1 doc4	doc2 dos	doc3 doc6		

Inverted Index

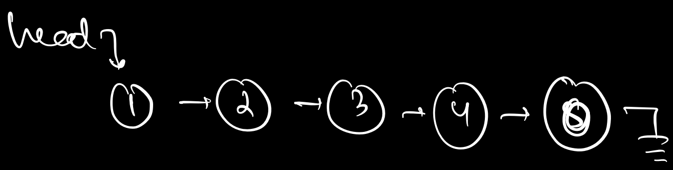
Q 3. Given a LL add data at end of LL.

I/p => head



insert → 0 at end

O/p =>

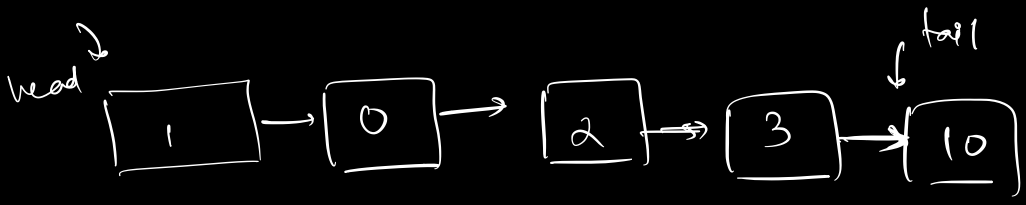


```
Node newNode = new Node (0)
temp = head
 while( temp.next != null){
            temp = temp.next.
 }
 temp.next = newNode;
```

TC => O(N)
SC => O(1)

=> Optimise

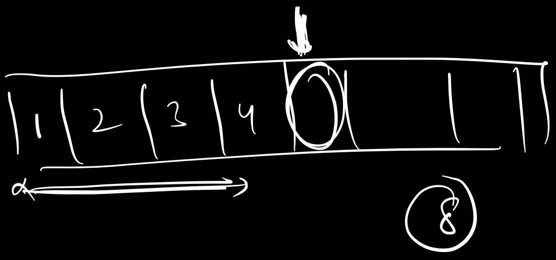
[TO-DO] -> google



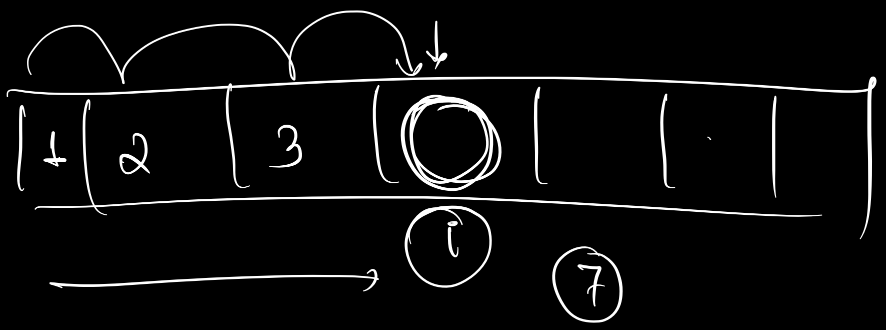
```
Node newNode = new Node(x);
    tail.next = newNode
    tail = newNode
```

=> TC => O(1)
SC => O(1)

array => O(N)
LL => O(1)



arr[N-1] → $\underline{O(1)}$



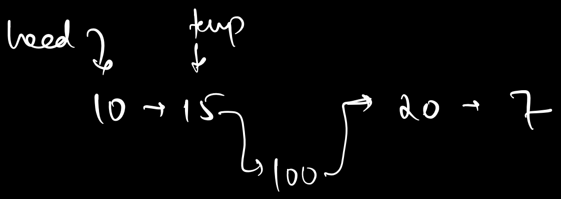
* insertion at edges [start or end]

LL → O(1)

Array → O(N)

Q 4. Given a LL, insert a node at kth position

I/p ⇒



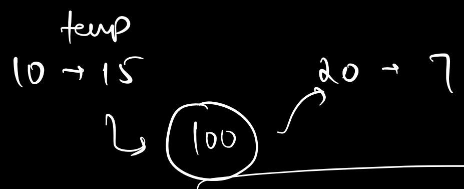
k = 2

$\underline{100}$

O/p ⇒ head

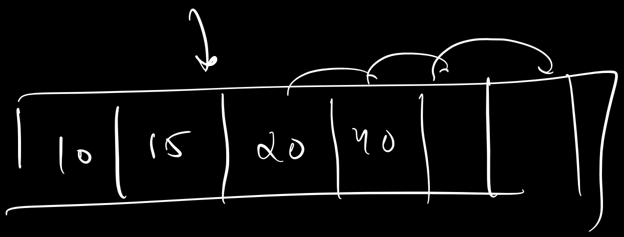
10 → 15 → 100 → 20 → 7.

```
c = 0
temp = head;
 while ( c < k-1 ) {
        c++
        temp = temp.next;

}
    newNode.next = temp.next
    temp.next = newNode

}
```



at kth position

LL } O(N)
array



H.W

Q 1. Delete a node from begining

Q 2. Delete a node from end.