

Arrays :-

↳ faster access.

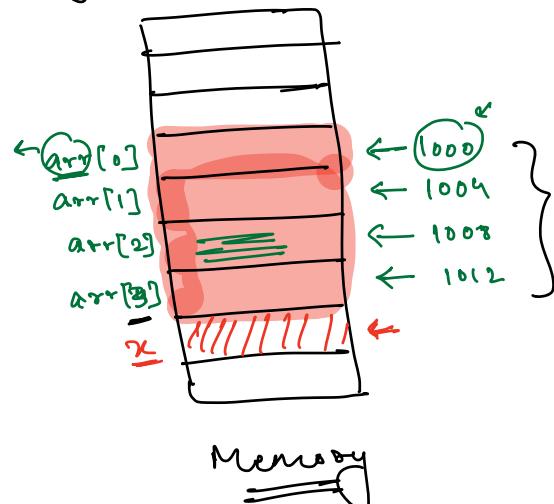
↳ $O(1)$ random access.

$$\text{arr}[i] \rightarrow \underline{\underline{O(1)}}$$

⇒ How array provides $O(1)$ access?

↳ Contiguous memory allocation.

int arr[4];
↓
4B.



⇒ Can we append array?

↳ NO

⇒ ArrayList / Vectors / List :-

↳ Dynamic Array.

→ Access time. :- $O(1) \leftarrow$
 \downarrow
Continuous
memory.

List<Int> list; \leftarrow

list.add(1) } O(1)

list.add(5) ↵

list.add(6)

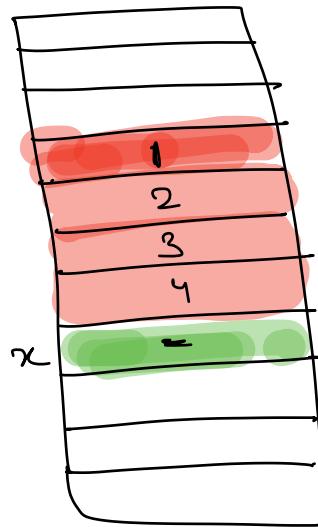


Diagram illustrating the insertion of element 5 into an array. The first array (dist) contains elements 1, 2, 3, 4. The second array (dist ==) contains elements 1, 2, 3, 4, 5, 6. A red arrow points from index 4 of the first array to index 5 of the second array, indicating the shift operation. A green arrow points from index 5 of the second array to the value 5, indicating the new element. A red circle highlights the value 5 in the second array. A green bracket above the arrays indicates the range of elements being shifted. A red annotation 'O(1)' is placed next to the second array's index 5.

1) Worst Case TC of insertion in Dynamic Array $\Rightarrow \underline{\underline{O(N)}}$

2) Insertion in Dynamic Array : $O(1)$ [Amortized] TC

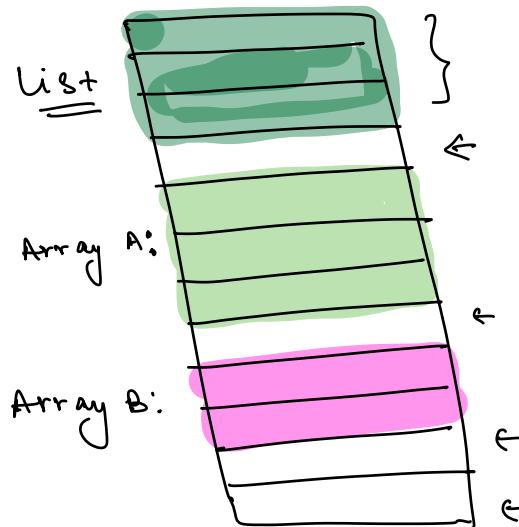
⇒    190 \$/s1/week.

Amortized
TC
↑
Avg. TC
when we
are performing
Many operations

A hand-drawn diagram consisting of three main parts. On the left is a circle containing the red handwritten text "10k". In the center is a rectangle containing the text "R0" above a double horizontal line. On the right is a larger rectangle with two small black dots near the top edge. A thick, double-lined arrow points from the "10k" circle to the "R0" rectangle.

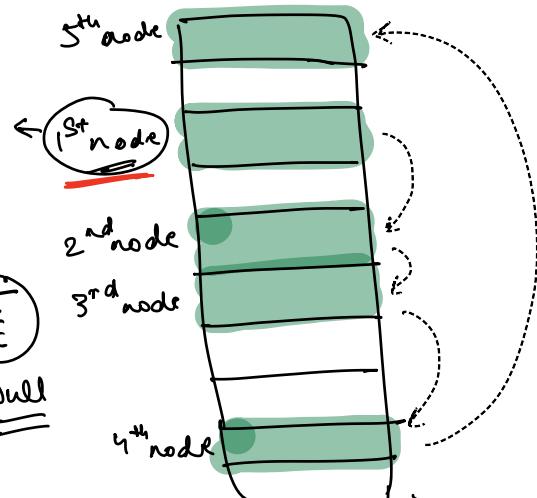
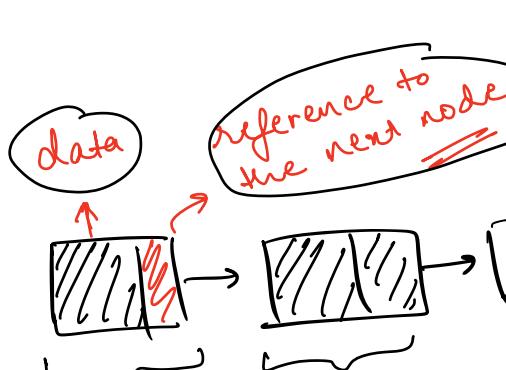
⇒ Dynamic

Array
↓
[Contiguous
Memory]



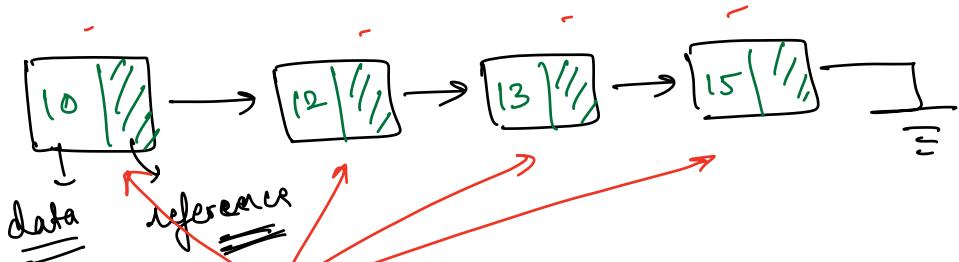
⇒ If we don't want ~~O(1)~~ access time?

↳ Linked List



↳ Stores the reference
of 1st node of L.L.

Implementation of L.L :-



Class Node {

 int data;

 Node next;

}

⇒ Any oops long

⇒ Class Node {
 int data;
 Node next;
 Node (int x) {
 this.data = x;
 this.next = null;

}

3

✓ Node newNode = new Node();
 newNode.data ⇒ 0/garbage.
 newNode.next → null/
 garbage
 null

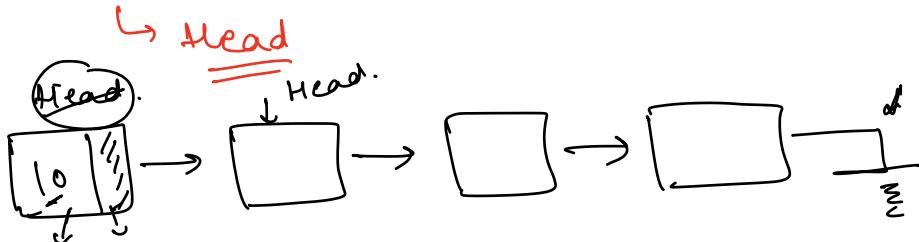
Constructor

Node n = new Node(10);

n.data ⇒ 10

n.next ⇒ null.

Q Given a L.L, find its length?



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

\Rightarrow Never Update Head of the L.L

Insert :-

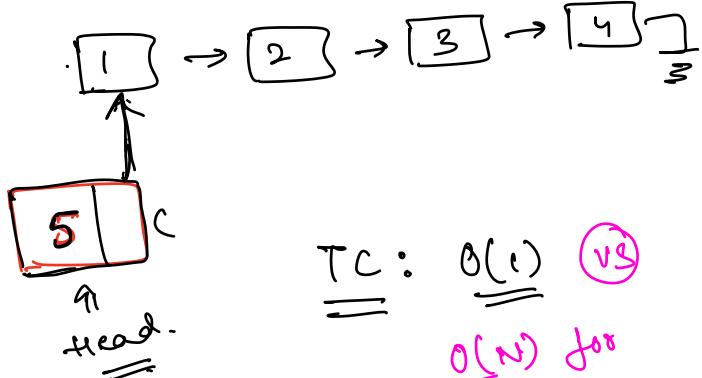
1) At front :-

- Create a new node.

Node newNode = new Node(5);

• newNode.next = head;

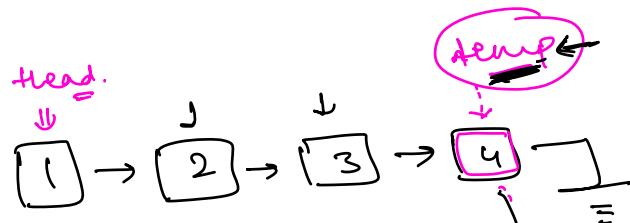
• head = newNode;



TC: $O(1)$ vs

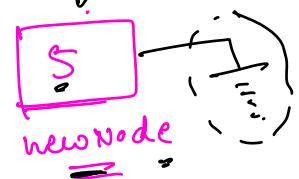
$O(N)$ for
Array-

1) At Back :-



- Create the new Node :

```
Node newNode = new Node(5);
```

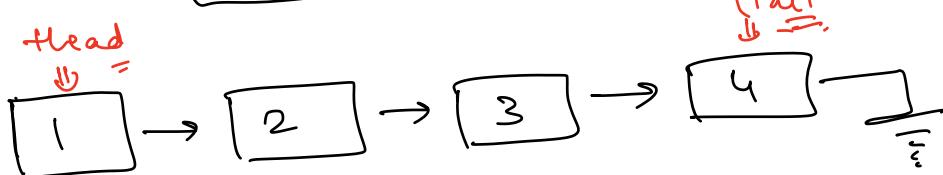


- $\text{temp} = \text{head};$

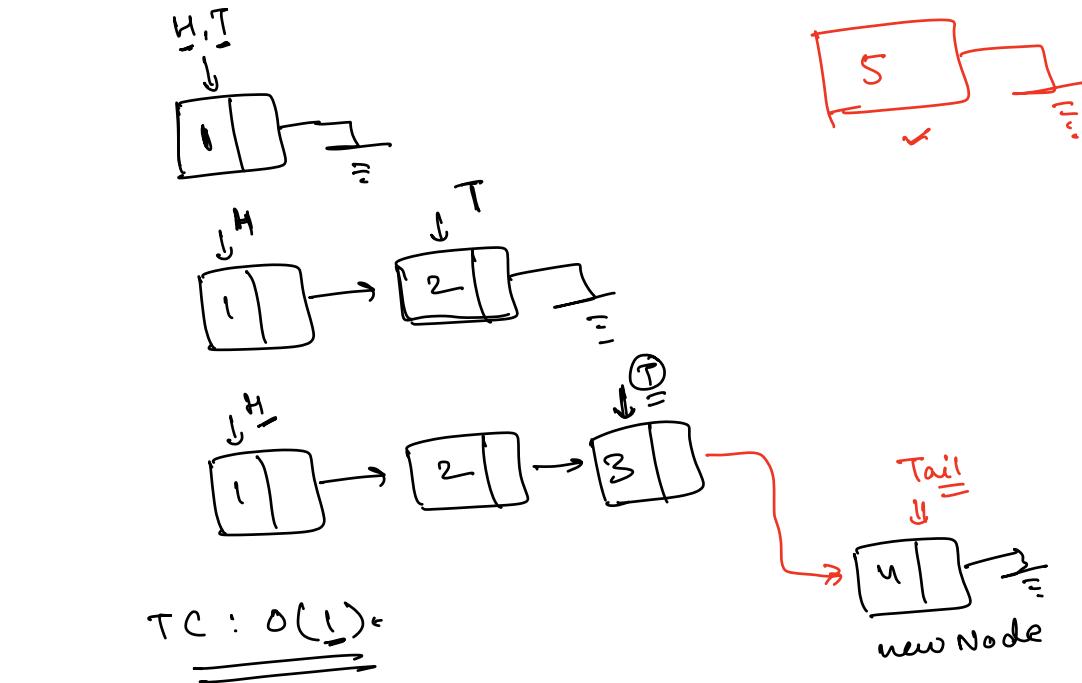
```
while (temp.next != null)  
    temp = temp.next;
```

- $\text{temp.next} = \text{newNode};$

TC : $O(N)$



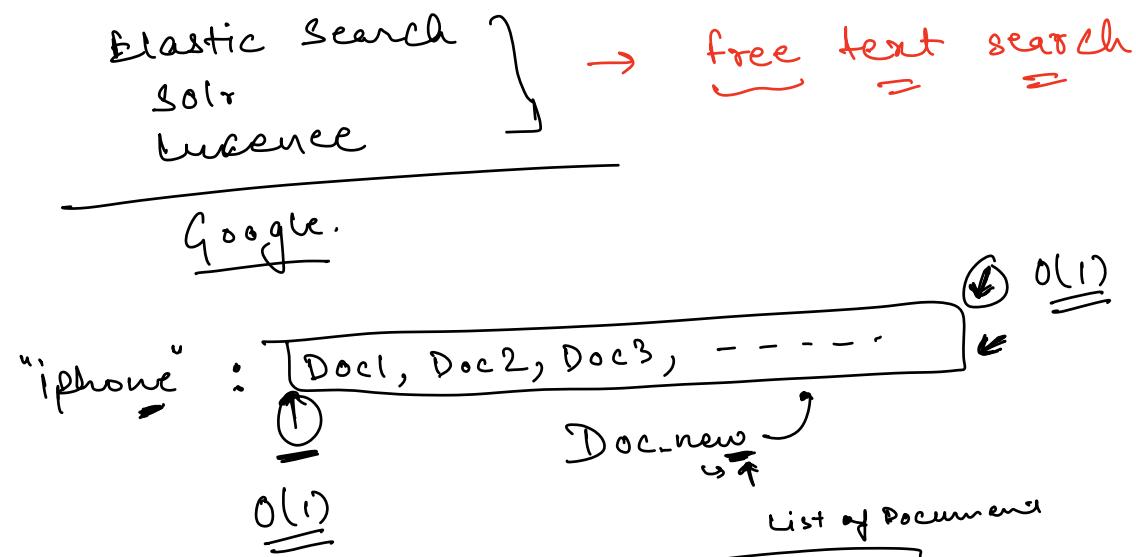
\Rightarrow



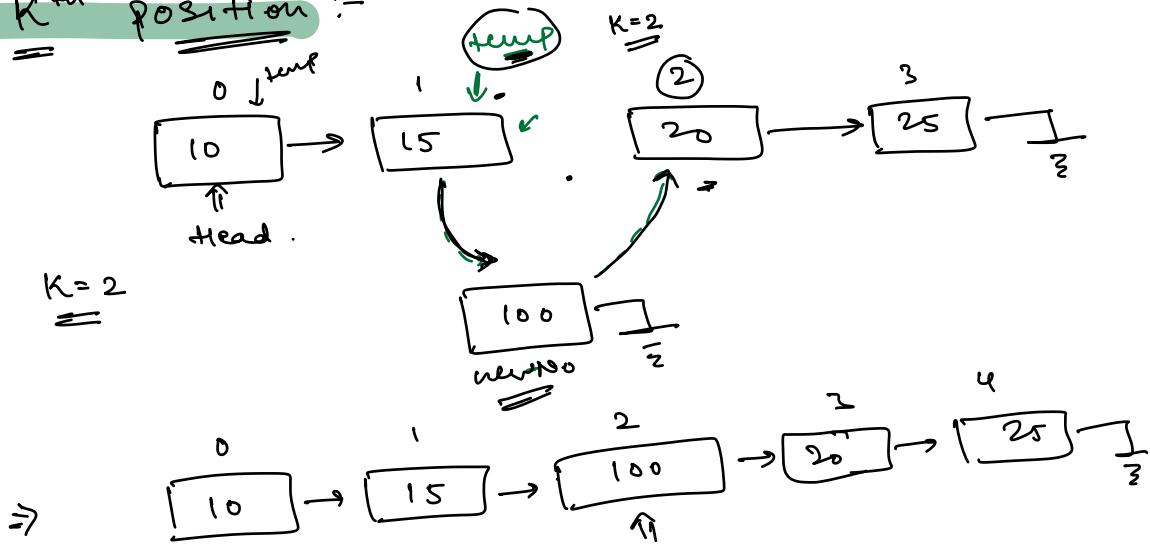
- Node newNode = new Node(4);
- Tail.next = newNode;
- Tail = newNode;

$\Rightarrow \text{TC} : \underline{\underline{O(1)}}$

Applications of L.L :-



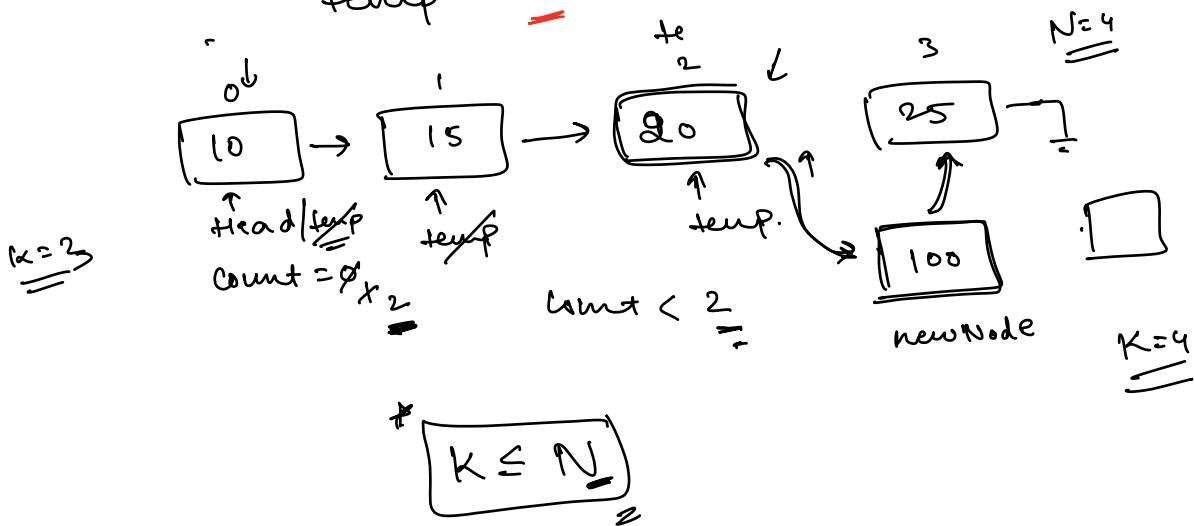
(11) K^{th} position :-



Count = 0
 $temp = head;$ // $K=2.$

while ($count < k-1$) {
 $temp = temp.next;$
 $count++;$ }

3
 $node newNode = new Node(100);$
 $newNode.next = temp.next;$
 $temp.next = newNode;$



⇒ Reference manipulation is very important.

⇒ Steps :-

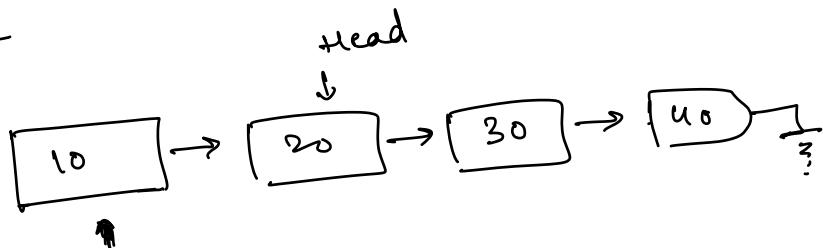
- 1) Write code on pen paper / google doc.
- 2) Dry run.

③ Edge cases :-

- Null L.L / Head = null.
- L.L of size = 1 | 2 | 3.
- Problem specific T.C's.

Deletion

1) At front :-

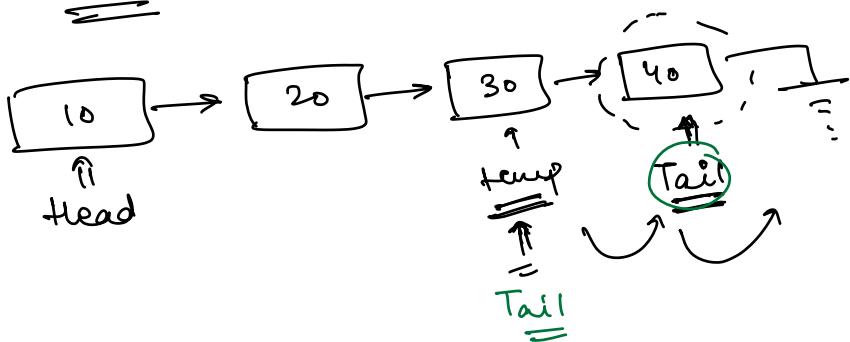


$\boxed{\text{Head} = \text{Head} \cdot \text{next}}$

TC: $O(1)$ vs $O(N)$
in
Array

C++ : Explicitly delete the reference.

2) from back :-



~~temp. next = null~~

while (~~temp. next~~. next != null) {
 temp = ~~temp. next~~;

}

temp. next = null;
tail = temp;

Tc: ~~O(N)~~