# Linked List

- Linear data structure
  like arr / list

arr [ 3 ]

this might not
be free

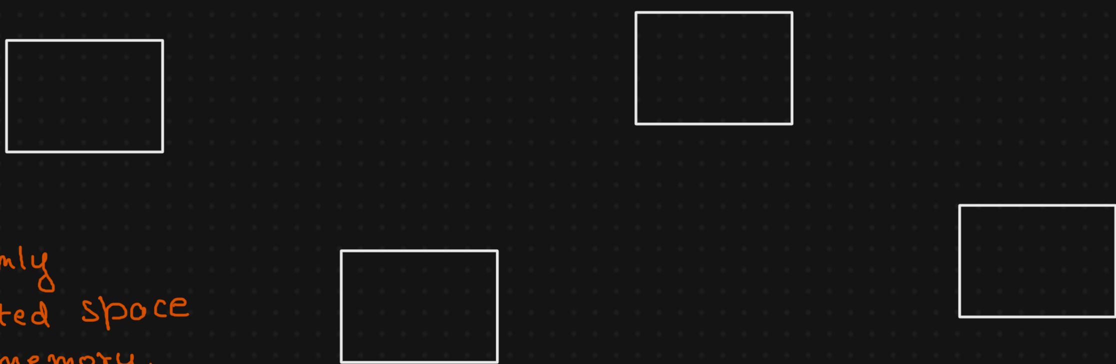
arr [ 10000 ]

getting wasted

- insert
- memory wastage

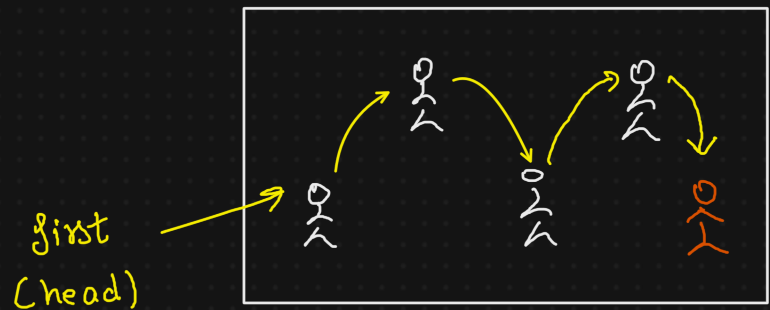
arr: elements are continously stored.

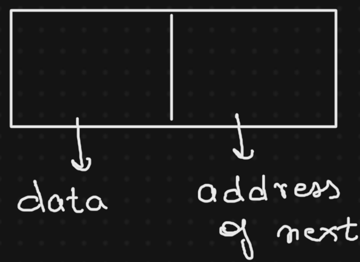
Add elements as per our requirements → Basic idea of LL

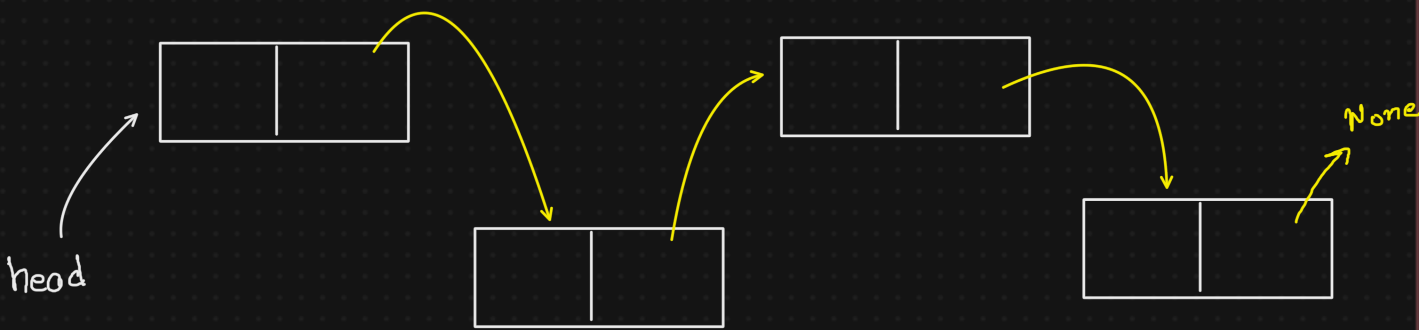


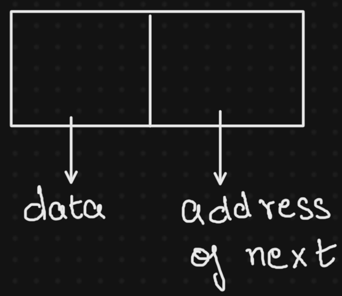
randomly
allocated space
in memory.

1. array / list → to store addresses
2. ask my elements to store the address of element after
   it


first
(head)


data
address
of next


head
None



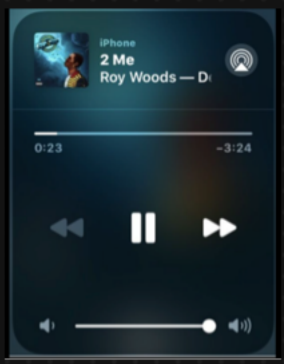

Node

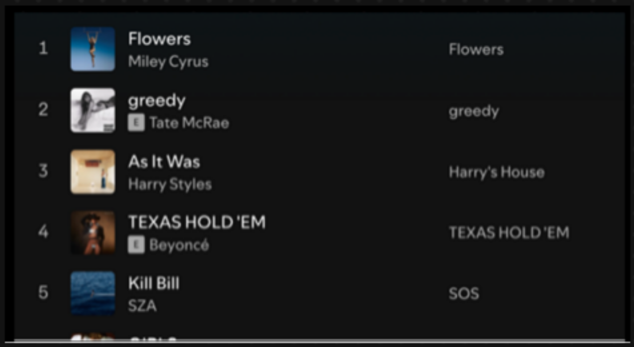
OOPS / Classes
↓
user defined data - type

```
Class Node :
    __init__( value ) :
        self.data = value
        self.next = None
```

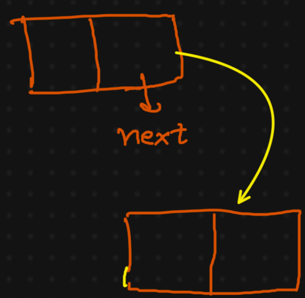
0x10


1. Playlist



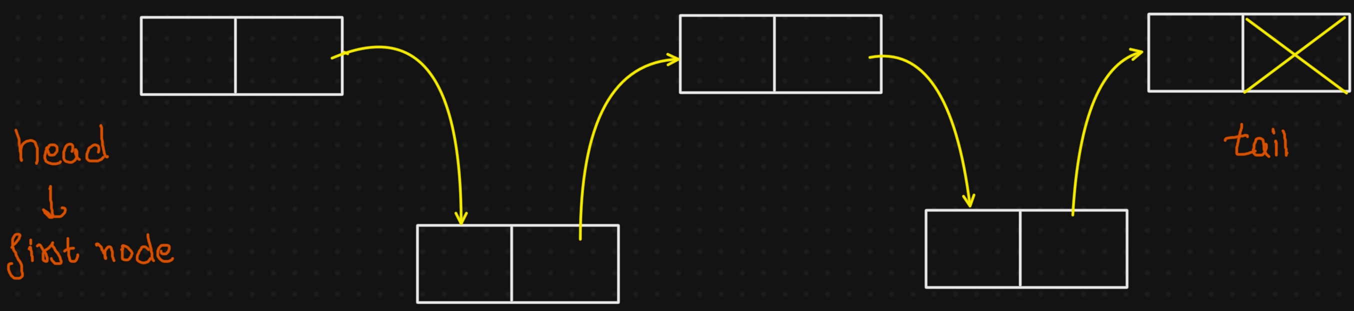




2. Treasure Hunt


3. Browsing History

# Linked List



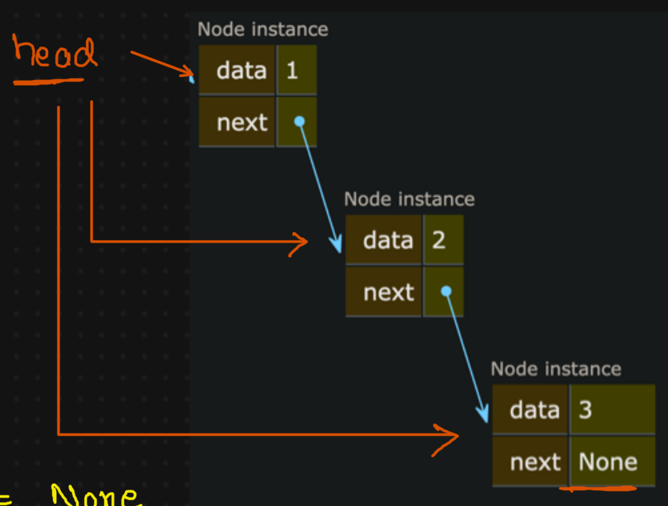

## Print ll

1
2
3

{ head . data → 1
head = head.next }

{ head.data → 2
head. = head.next }

{ head.data → 3
head = head.next }

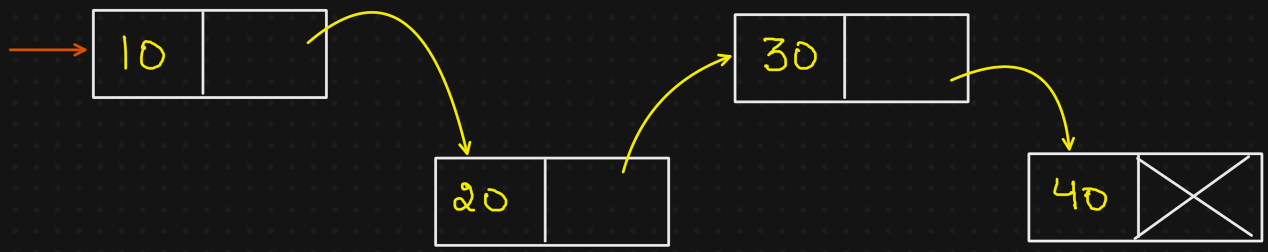



head = None

we cannot use. .data or .next on my $\boxed{\text{None .}}$ *

Take Input of Linked List

1 → 2 → 3 → None

head


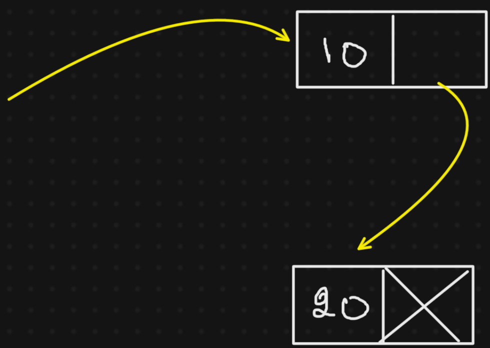

10
20
30
40
-1

empty linked list   -1

head


return None as head

first node created

head = ~~None~~



first

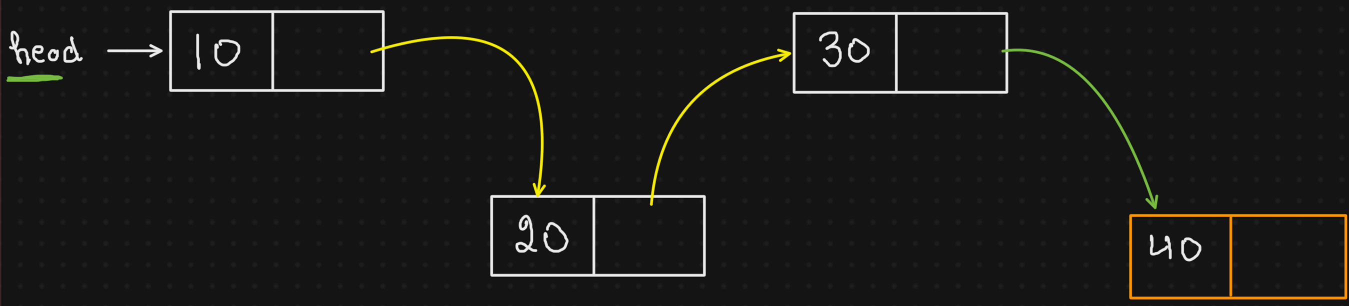
first.next = Second

second node created

second

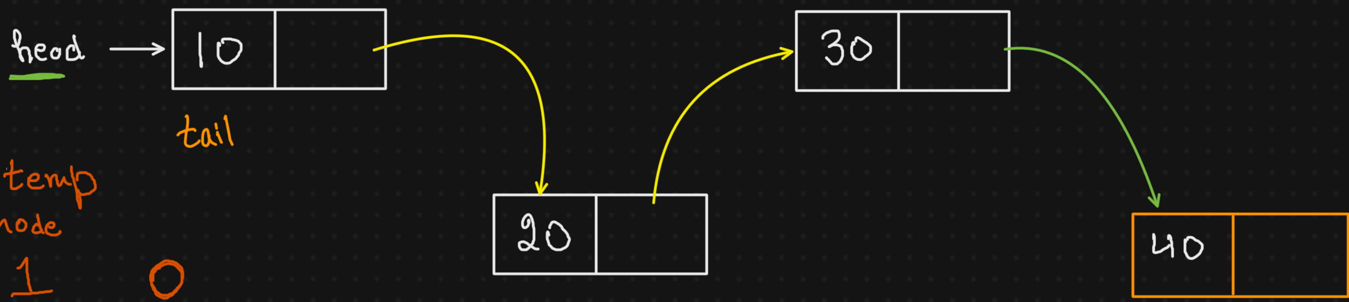
# Take Input (Linked List) - Corrected

We saw how take input was failing as we failed to connect last node to the newly created node.




Last node is where next is None

What is the time complexity of take input ( ) function?



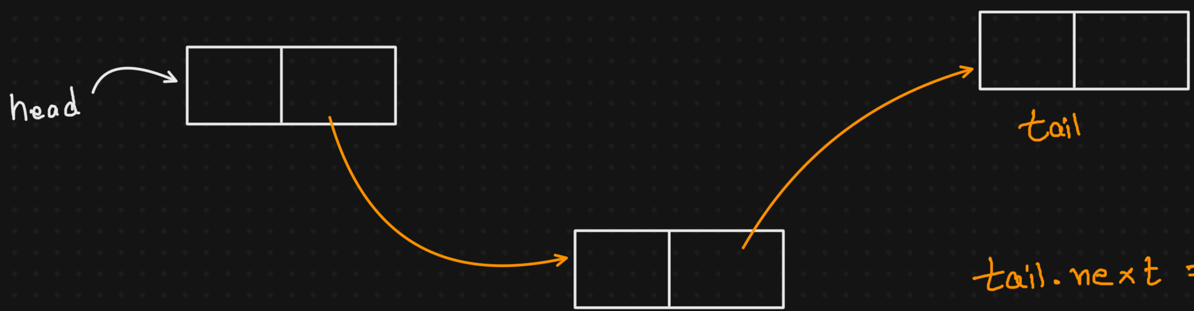

| temp node | |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |
| ⋮ | |
| n | n-1 |

Outer loop → $n$

inner loop → $n$

time complexity $O(n^2)$ ← pretty bad just for taking input

Is there some way we can reduce our time complexity?

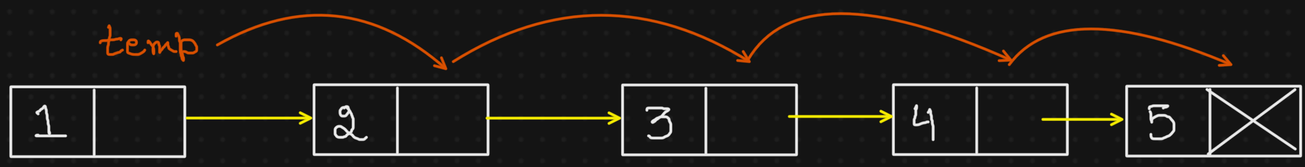



tail.next = newNode

tail = newNode

Using a tail variable/ pointer, we have successfully reduced Time complexity from $O(n^2) \longrightarrow O(n)$

Length of LL

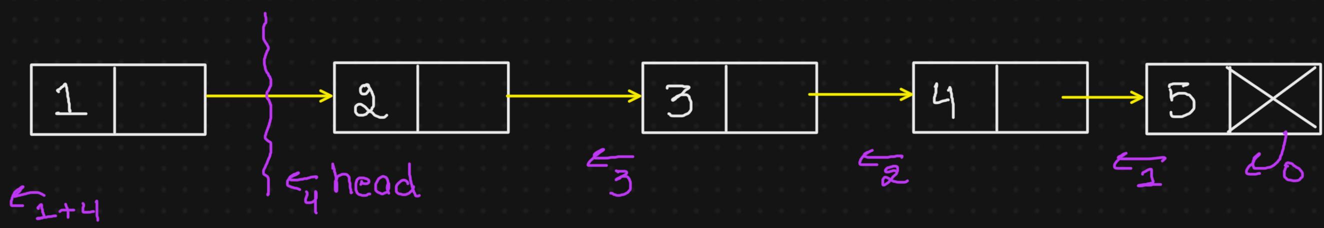



head

def length (head):

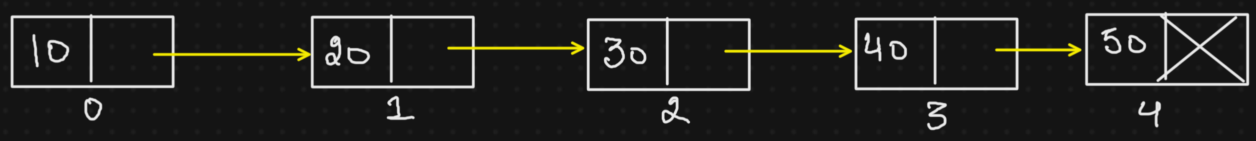
return length

Linked List length using recursion




1. Base Case ⟶ head == None   return 0
2. Recursive Call ⟶ to get length of head.next
3. Our Work ⟶ 1 + recursion answer

# Linked List Operations



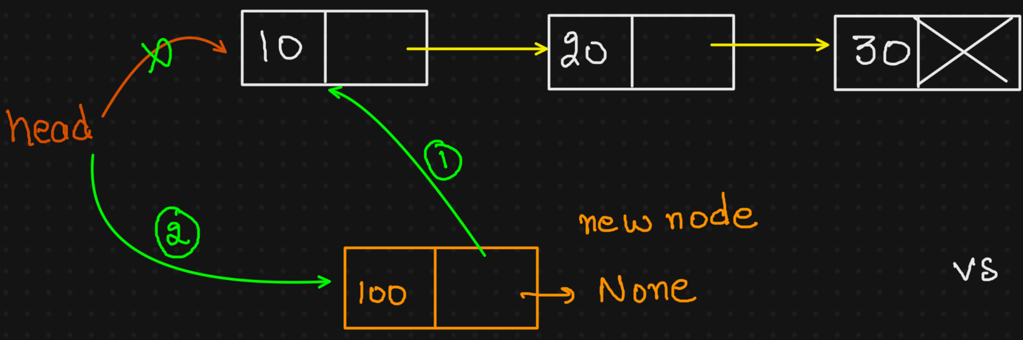

1. Insert
   - Insert at head
   - Insert at tail
   - Insert in between

2. Delete
   - delete head
   - delete tail
   - delete by index
   - delete by value

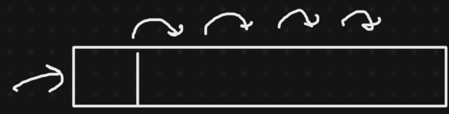
3. Search
   - Search by index
   - Search by value

4. Traverse
   - Print

# Insert at head of LL




vs array / list



$O(n) \Leftarrow$
1. capacity
2. shifting

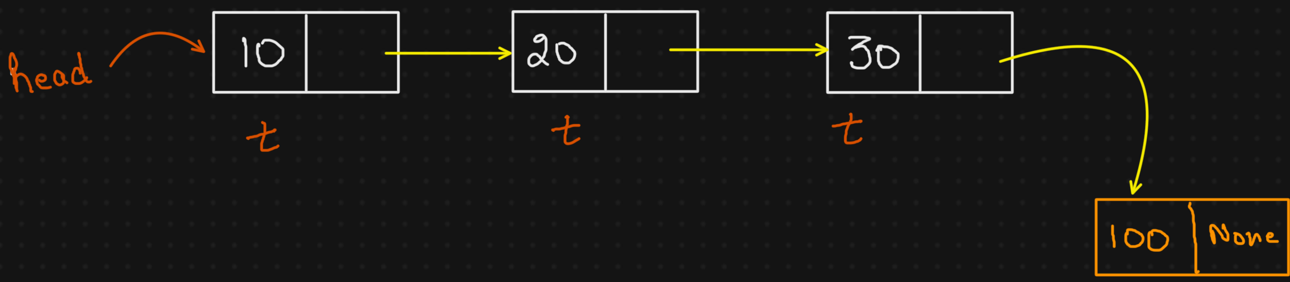
function can either just make the change or return the updated head.

Time Complexity : Insert at head

$\Rightarrow$ as only constant operations $\Rightarrow O(n^0)$

$O(1)$

Insert at tail of LL




when LL is empty
we have to just
return the newnode

compare with array/list

O(n) time complexity

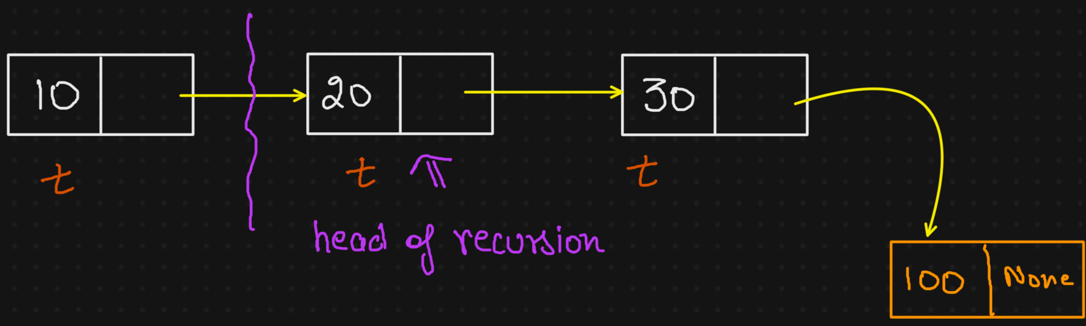
Time complexity : O(n) as we traverse to tail

Insert at Tail - Recursive

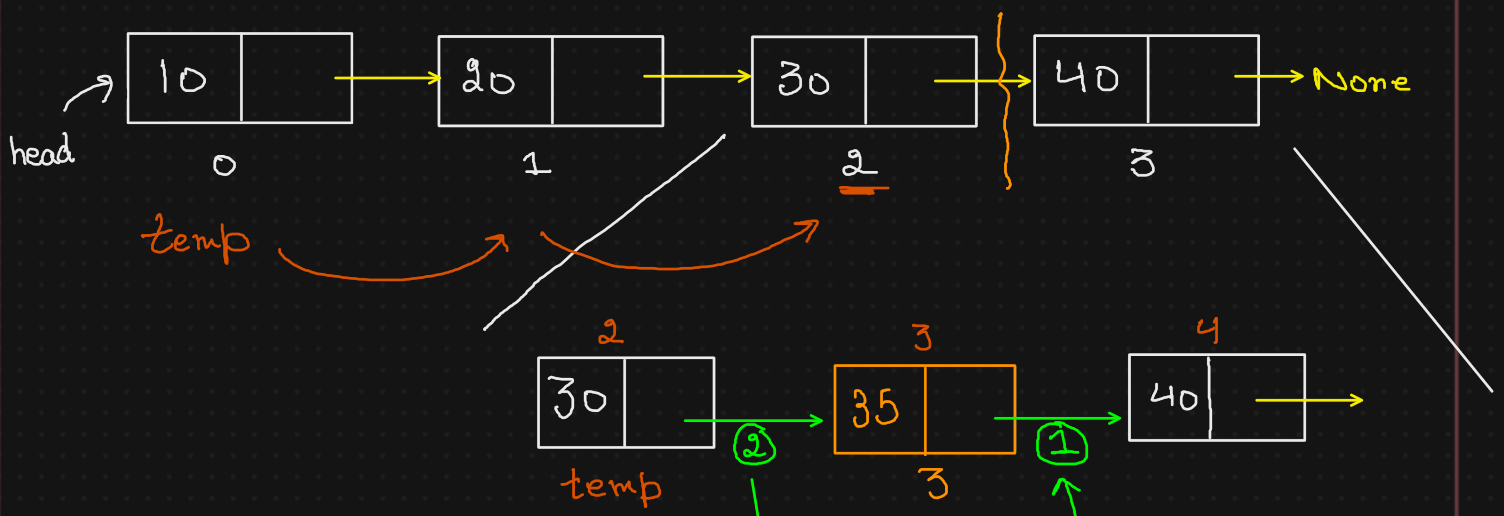
1. Base Case → If (head == None) return newNode

2. Recursion Call →

3. Our work → If (head.next == None) ⇒ we are tail



# Insert a node at an Index




temp.next = newnode

should be done first using node (30)

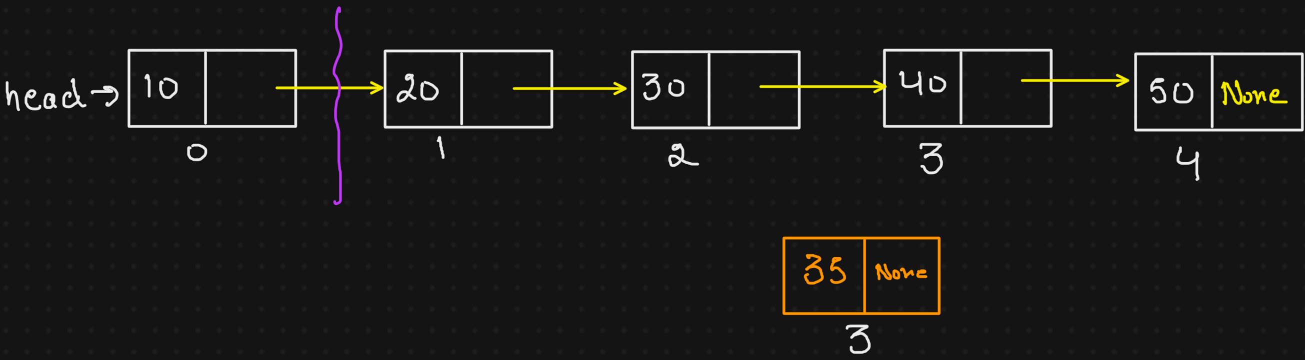
new node.next = temp.next

1. make a temp
2. move temp till index-1, using count var
3. Update the connection to insert new node
4. Order is very important

## Edge cases

1. if index is 0 ⇒ we have to add at head
2. if index > len ⇒ out of bounds error
3. if index == len i.e. add at end.

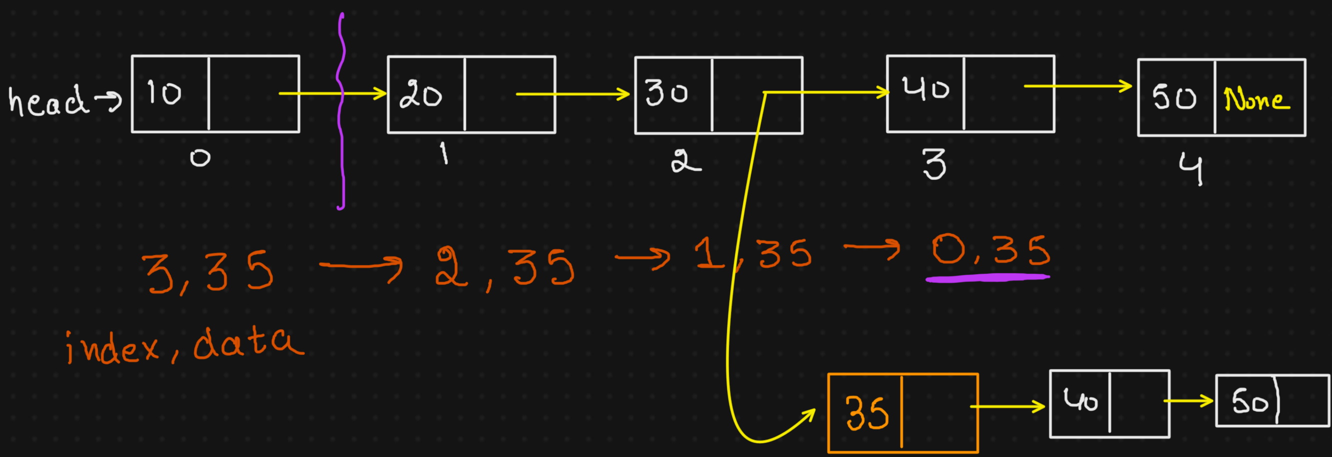
# Insert node at an Index (Recursively)



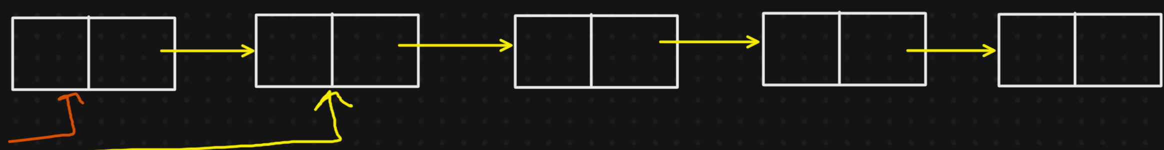

1. Base case     index == 0 insert at head
   head = None

2. recursive call.    head.next = insert ( head.next , data, index - 1 )

3. Our work    return head



# Delete Node in a Linked list = Head node



Head

updated head

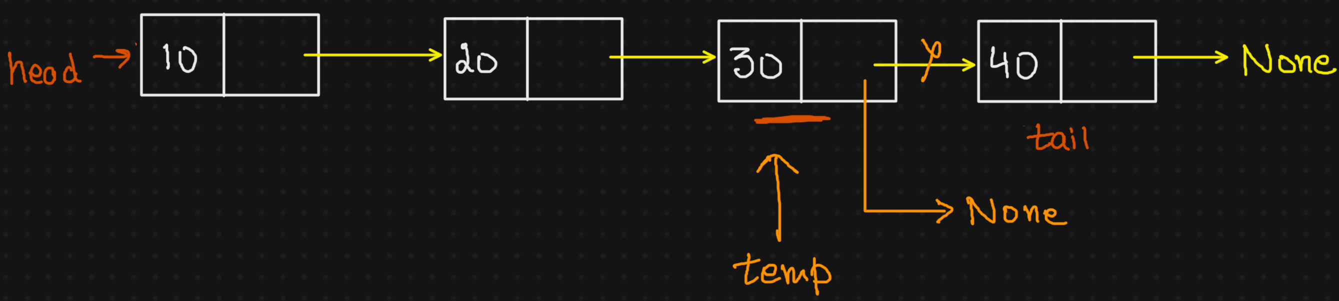
head = head.next

Time complexity = $O(1)$

$O(n)$

* Make sure to handle edge case of empty LL

## Delete a node in LL → Tail

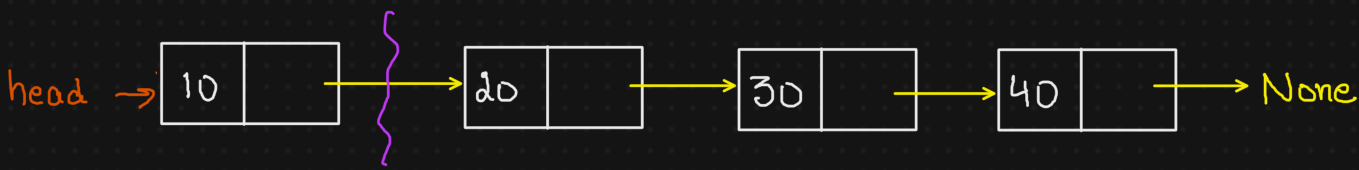



When we want to delete a node, we stop at a node before.

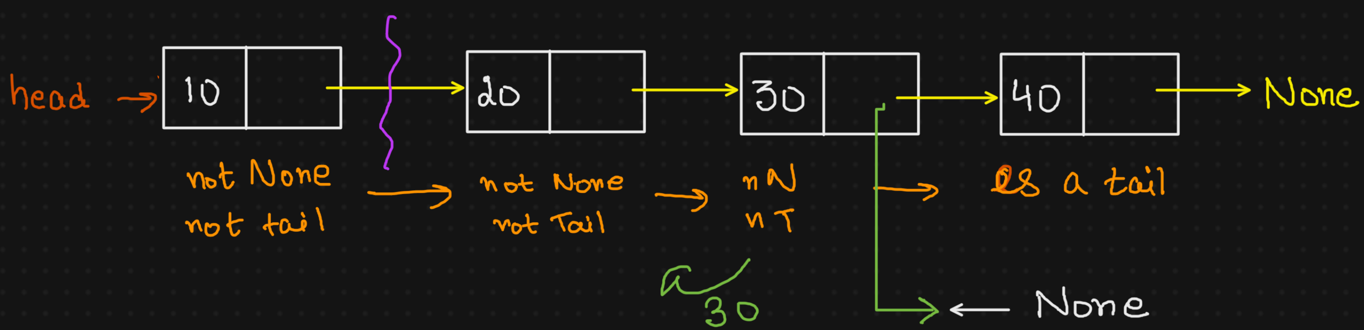
When we have to insert a node, we stop on the node where we will insert

while ( temp.next.next is not None)

## Delete a tail node - recursively




1. Base case → if head == None
2. Recursive call → del (head.next)
3. Our work → if (head.next) is None:
   return None



# Delete a node at a given index

$10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow$ None

head ↗ 10 (index 0), 20 (1), 30 (2), 40 (3), 50 (4)

delete node at Index 3

Complexity $O(n)$

# Delete a node at Index : Recursively

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow$ None

1. Base case : if (head is None)
2. Recursive call : head.next = del (head.next, index-1)
3. Our work : if index == 0 , return head.next

Delete a node by value

$$1 \to 2 \to 3 \xrightarrow{2} 4 \xrightarrow{1} 5 \to \text{None}$$

head

we will have to stop 1 node before so we can change the connection

delete 4

1. head value
2. value not present
3. value is present

# Search in a Linked List : Value

$$\text{head} \nearrow \underset{0}{1} \rightarrow \underset{1}{2} \rightarrow \underset{2}{3} \rightarrow \underset{3}{4} \rightarrow \underset{4}{5} \rightarrow \text{None}$$

Similar to Linear Search

value = 3

O(n)
as worst case

Search in Linked List : Index

Iterative / Recursive

head

$$\underset{0}{\underline{1}} \rightarrow \underset{1}{2} \rightarrow \underset{2}{3} \rightarrow \underset{3}{4} \rightarrow \underset{4}{5} \rightarrow \text{None}$$

| | Array / List | Linked List |
|---|---|---|
| Access | index<br>O(1) | O(n) |
| Memory management | Fixed size<br>- Some vacant<br>- Some filled<br>memory fragmentation not possible | → No unused memory<br>→ All nodes have data<br>→ extra memory for my pointers<br>memory fragmentation possible |
| Insert | O(n) start<br>O(1) end, not full<br>O(N) end, full<br>O(N) middle | O(1) head<br>O(n) at tail<br><br>O(N) |
| | good for indexing<br>read fast | insert/delete from my head<br>write fast |

Linked List Class

1. Create
2. Insert
3. Delete
4. Search
5. Traverse / Display