25/02/22

DSML Intermediate – DSA

Linked List Basics

=> Arrays

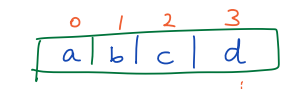
=> — continuously stored in memory (contiguous)

✓ — linear (1D)

✓ — sequential

Start idx 0

end idx (N-1)



List of 4 characters.

Random Access = O(1)

TC : Access an element is O(1)

Insertion . O(N)

=> Linked list

DYNAMIC

=> buy a new apartment

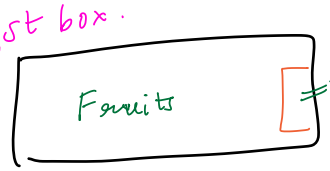
STATIC

Contiguous.

Kitchen

Drawers.

=> 1st box.



Box 1

⇓ Room

store location of next box => Gallery.

Box 2

⇓ Bathroom

Box 4

⇓ Study Room

=> study room

Box 3

⇓ Gallery.

Why Linked Lists?

⇒ A lot of other data structures used LL to implement them.

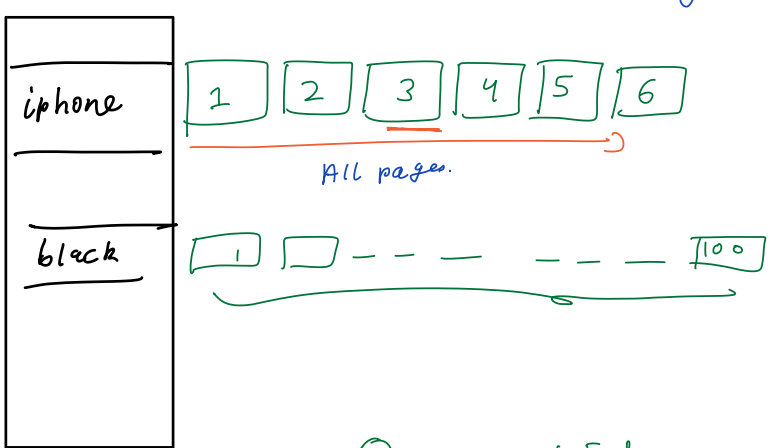
Real use case

Elastic search ES ⇒ inverted index

solar

Lucene

iphone ⇒ List of pages) documents that contain keyword iphone.



Hash Map is internally implemented in a similar way.

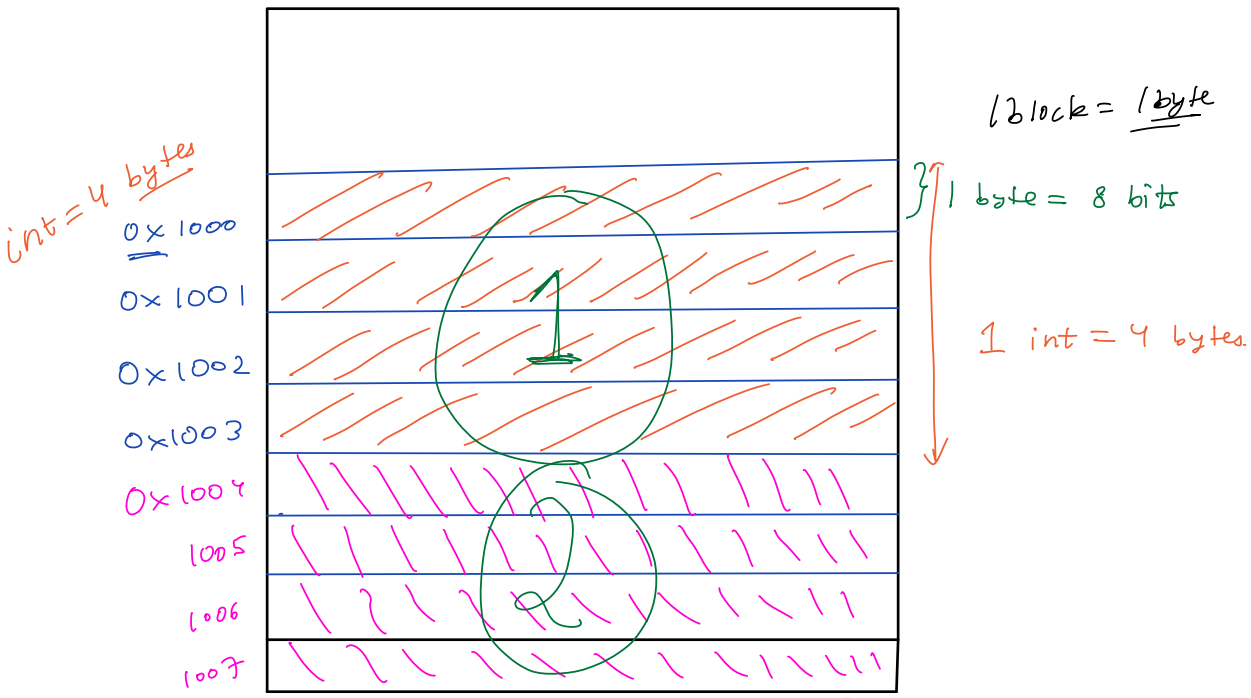
Dynamic nature ⇒

1. Insertion at End / Start
2. Deletions.

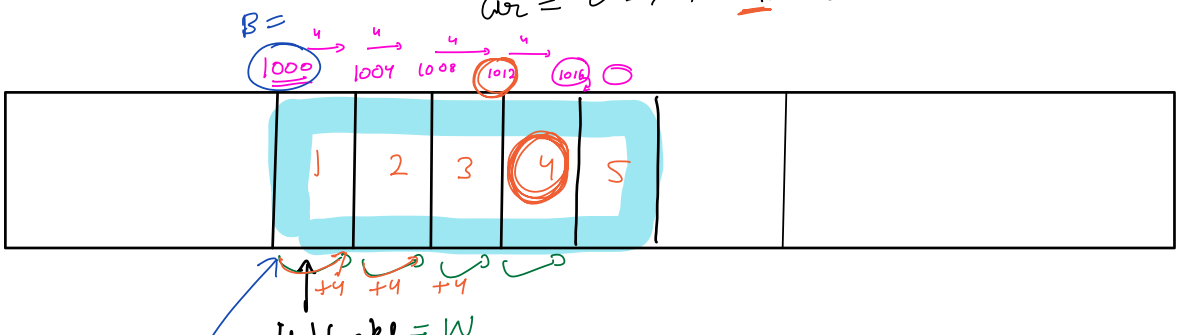
Random Access is not needed

Memory

ar = [1, 2]
0 1



ar = [1, 2, 3, 4, 5]
0 1 2 3 4



idx W
1000 + 4 * 4

$B + i * W$

O(1) time.

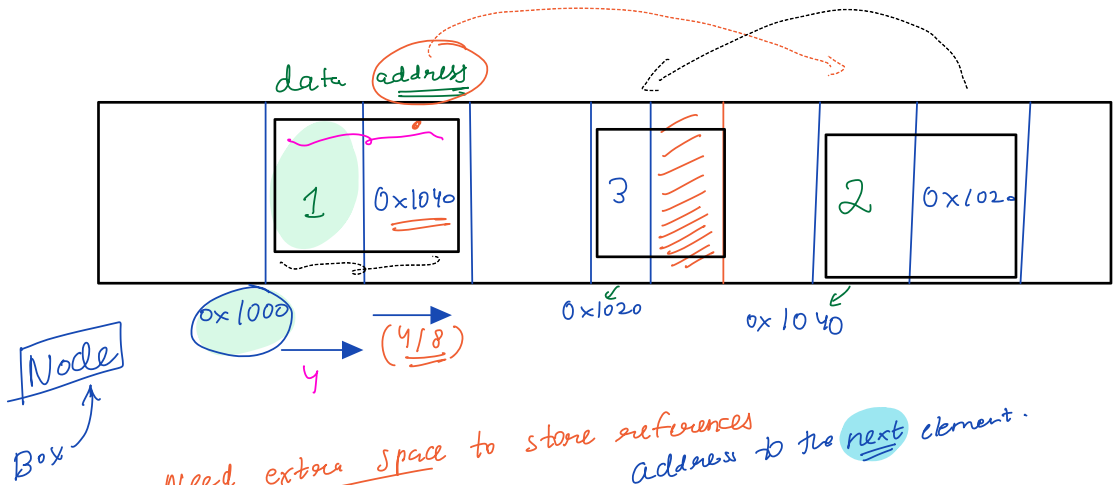
Random Access Memory.

B + 3 * 4

1000 + 12 = 1012

Linked List

1, 2, 3,



Need extra space to store references address to the next element.

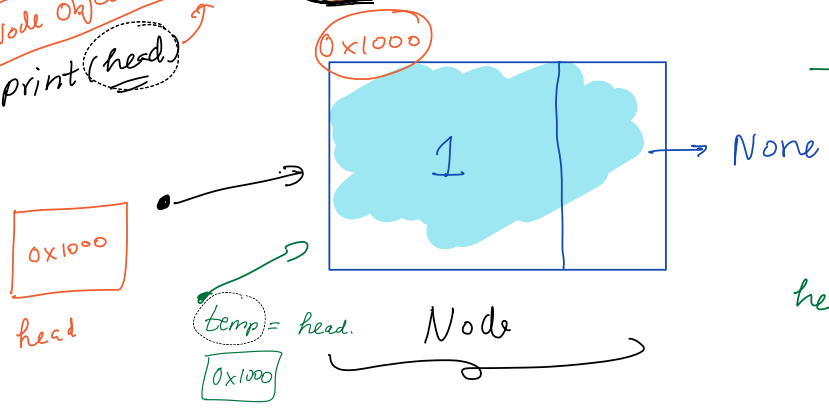
Head node = start node.

```
Class Node :

    def __init__(self, val):
        self.data = val
        self.next = None
```

Node object at 0x1000

print(head)

head = Node(1) → memory allocation is done automatically by Python.



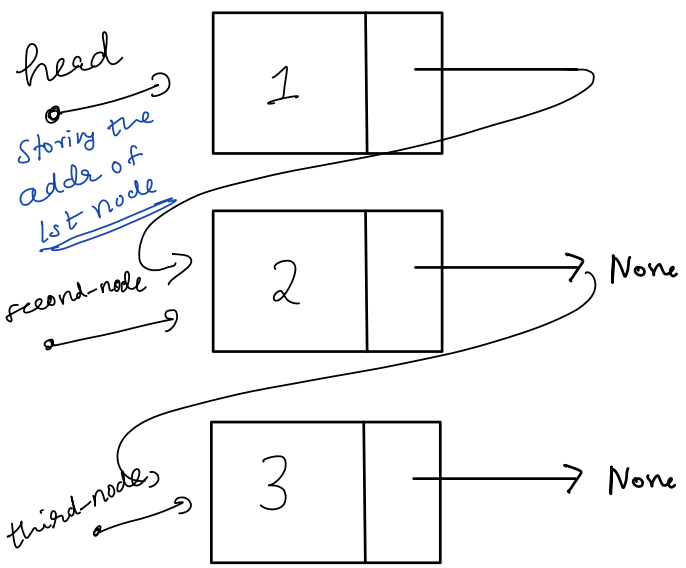
head & temp are different.

```python
class Node:
  def __init__(self, val):
    # attribute for storing the data
    self.data = val
    # we want to store the reference for the next node in the memory
    self.next = None

head = Node(1)
print(head)

second_node = Node(2)
print(second_node)

third_node = Node(3)
print(third_node)

temp = head # temp is now pointing to the same node as the head node (reference)
print(temp)
```

head.next = second-node

second_node.next = third_node

None.



head.next = second_node

```python
class Node:
  def __init__(self, val):
    # attribute for storing the data
    self.data = val
    # we want to store the reference for the next node in the memory
    self.next = None

# head it is the reference to actual node in the memory
head = Node(1)
print(head)

temp = head
temp = temp.next # what will be the value of temp
print(temp)

# now temp is referring to None but head still refers to 1st node
print(head)
```

head → 1 (0x1000)

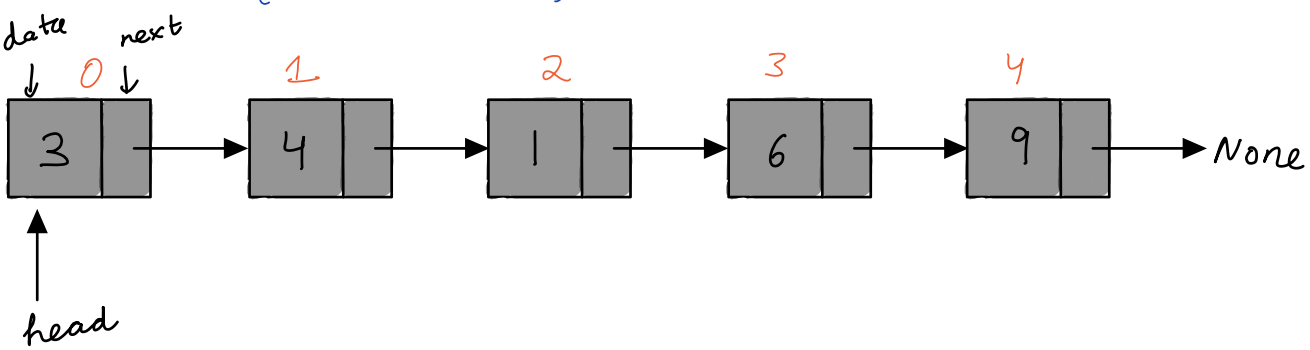
temp. 0x1000 → None. (empty address.)

None

```
<__main__.Node object at 0x7f0eb7a826d0>
None
<__main__.Node object at 0x7f0eb7a826d0>
```

All the problems ⇒ given with a head reference.

Q1) Given a LL (given the head), find the K^{th} element (return its value)



K=2 → K=1 → K==0

```
def findKthNode(head, K):
    while(K > 0):
        head = head.next
        K -= 1
    return head.data
```

Should not change head.

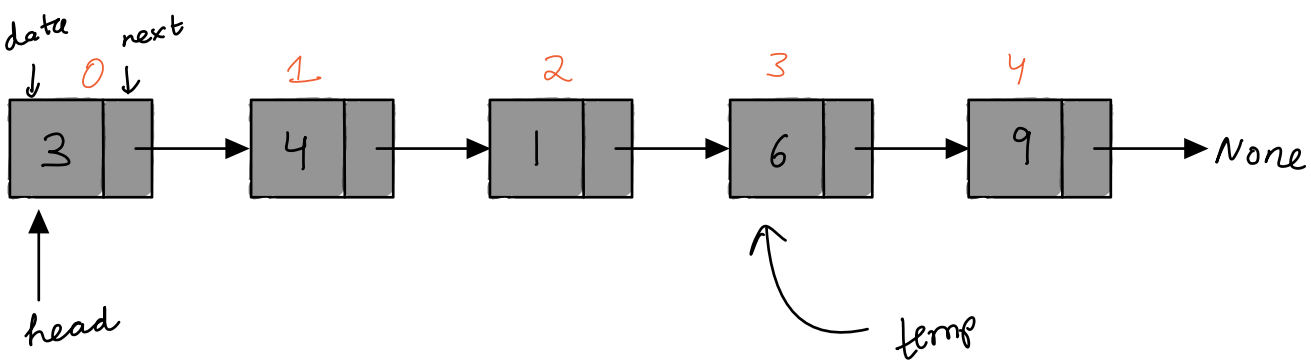
```
def findKthNode(head, K):
    temp = head
    while(K > 0):
        temp = temp.next
        K -= 1
    return temp.data
```

K = 3 ⇒ 6

2

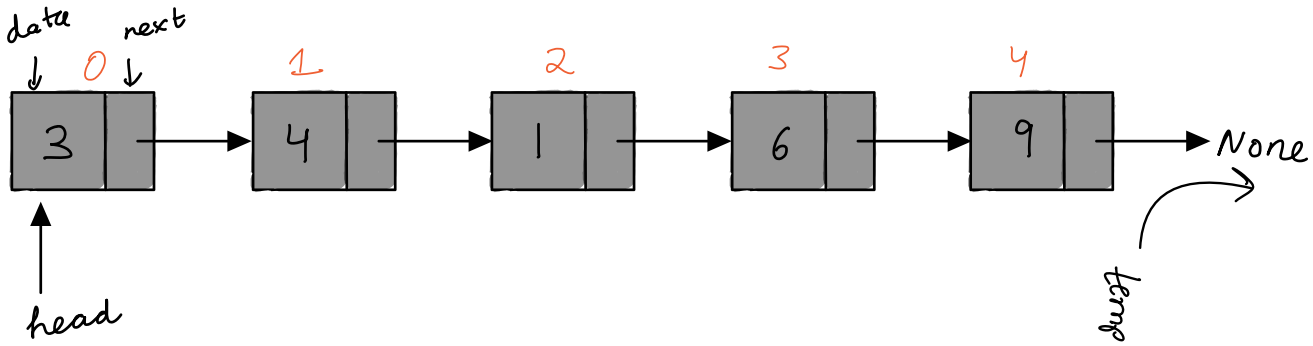
1 → 0

make sure data is present.



Edge Cases

K = 5 → 4 → 3 → 2 → 1 → 0



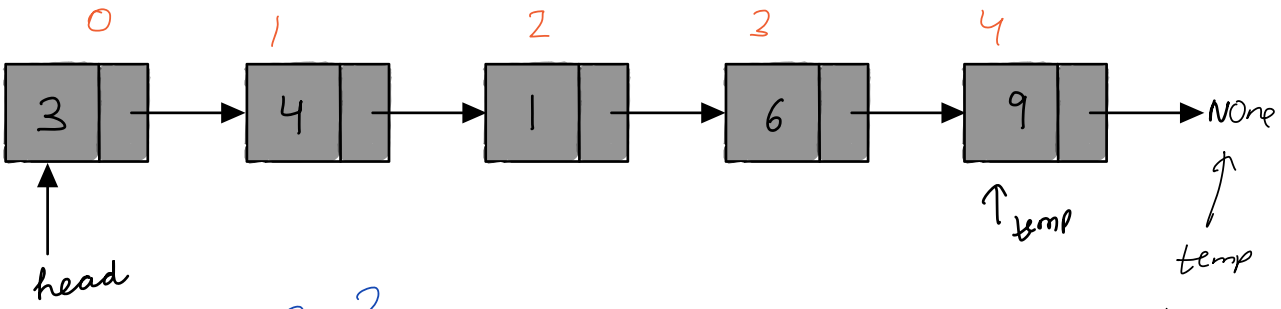
Seg. Fault
Null pointer exception

Error — Trying to access invalid data

K >= N

```
def findKthNode(head, K):
    temp = head
    while (temp != None and K > 0):
        temp = temp.next
        K -= 1
    if (temp == None):
        return None # no data present (K >= N)
    return temp.data
```

before accessing temp. make sure that it is not None.



K=0 ⇒ 3

K=5 ⇒ None

K=4

head = None

N=0

K=0 ⇒ None.

K=1

K=2

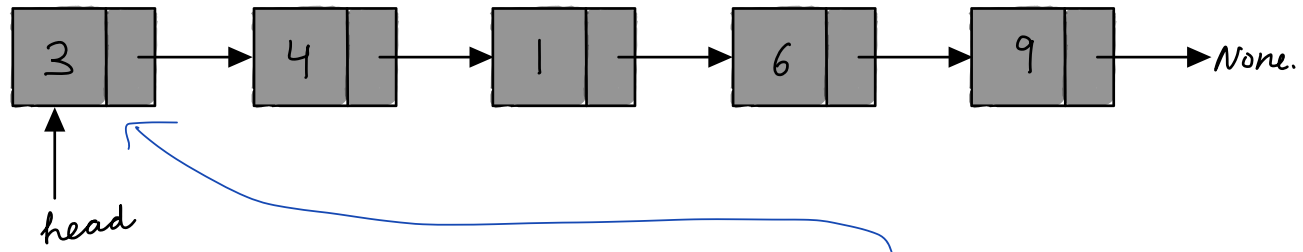
K=N

TC: O(N)

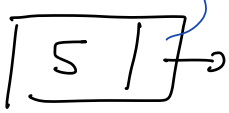
SC: O(1)

Insertions in a Linked List

A. At Head



(1) Insertion at Head. (5)



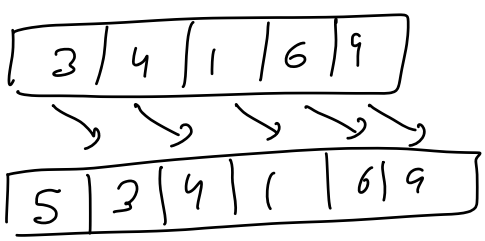
for one node.

```
def insert_at_head(head,val):
    Node  X = Node(val)
    X.next = head
    head = X
```

TC: O(1)

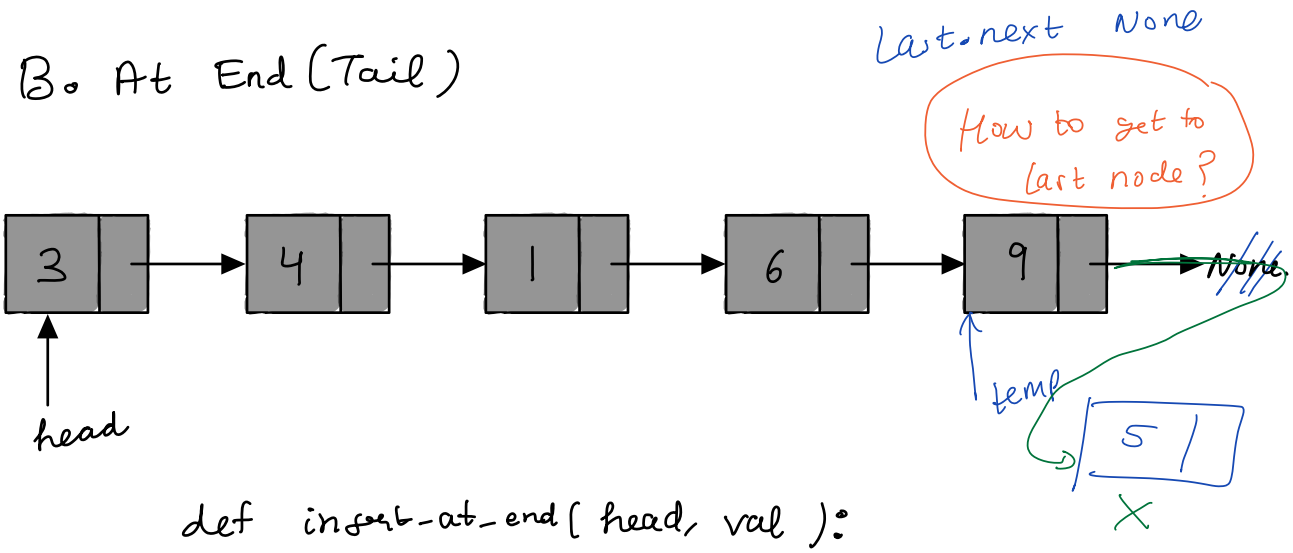
SC: O(1)

Arrays



TC: O(N)

B. At End (Tail)




```
def insert_at_end( head, val ):
    x = Node(val)
    temp = head
    while( temp.next != None ):
        temp = temp.next
    temp.next = x
```

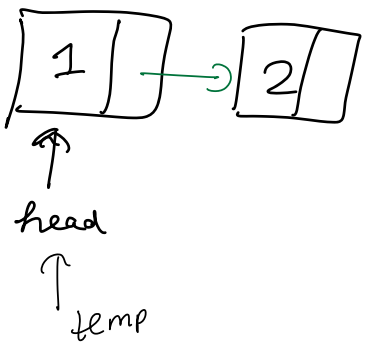
Can it be None?

=> Fail for empty list

X

return type = None

```
def insert_at_end(head, val):
    x = Node(val)
    temp = head
    if head == None:
        head = x
        return
    while(temp.next != None):
        temp = temp.next
    temp.next = x
```

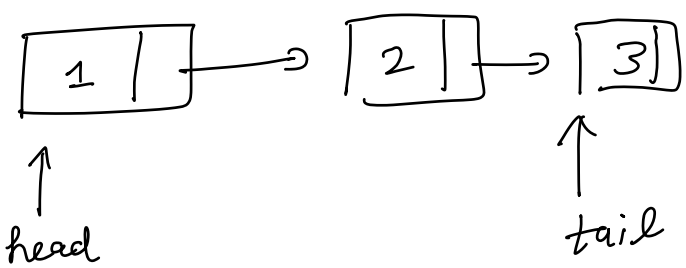


O(N) =>

TC: O(N)

SC: O(1)

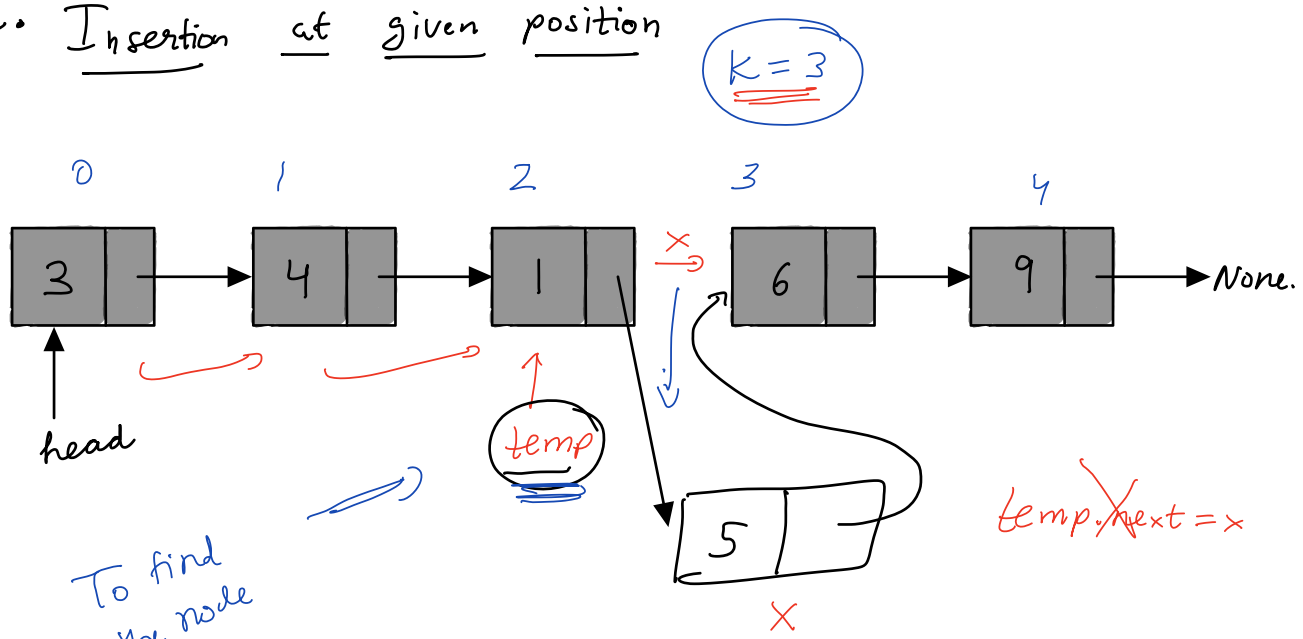
Doubly Linked list



```
def insert_at_end(head, tail, val):
    x = Node(val)
    if tail == None:        # empty list case.
        head = x, tail = x
    else:
        tail.next = x
        tail = tail.next
```

TC: O(1), SC: O(1)

C. Insertion at given position




```
x = Node(val)
x.next = temp.next
temp.next = x
```

Worst case TC = O(N)

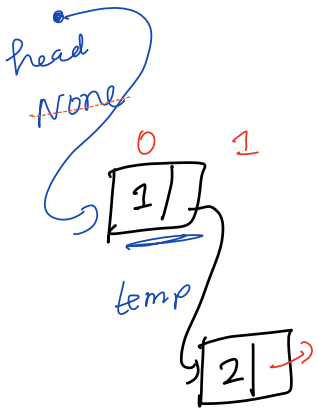
K=1 $0 \le K < N$

```
def insert_at_pos(head, K, val):
```

O(K) ⇒ `temp = findKthNode(head, K-1)` # return a ref

```
    if temp == None:
        insert_at_head(head, x)
        return
    x = Node(val)
    x.next = temp.next
    temp.next = x
```

actual insertion



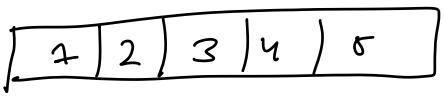
TC: O(N), SC: O(1)

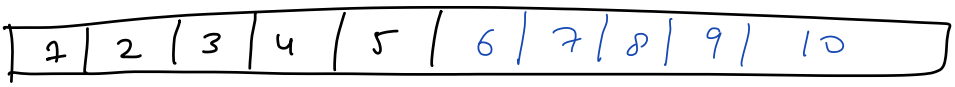
=> Dynamic Arrays.

Worst case : O(N)

cur
max size.

Doubling size concept





Amortized insertion O(1)