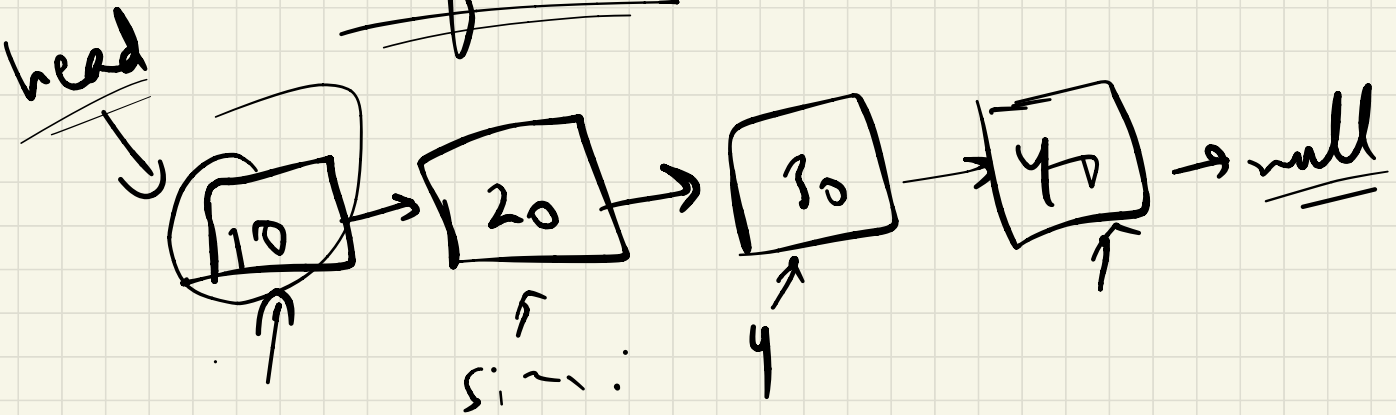
Size of a LL

count = 0 + 1 + 1 + 1 = 4



Insertion of a Node in a LL

3 possible cases :-

1) Inserting a new node at the head (at the beginning)

2) " " " after the tail (at the end of the list)

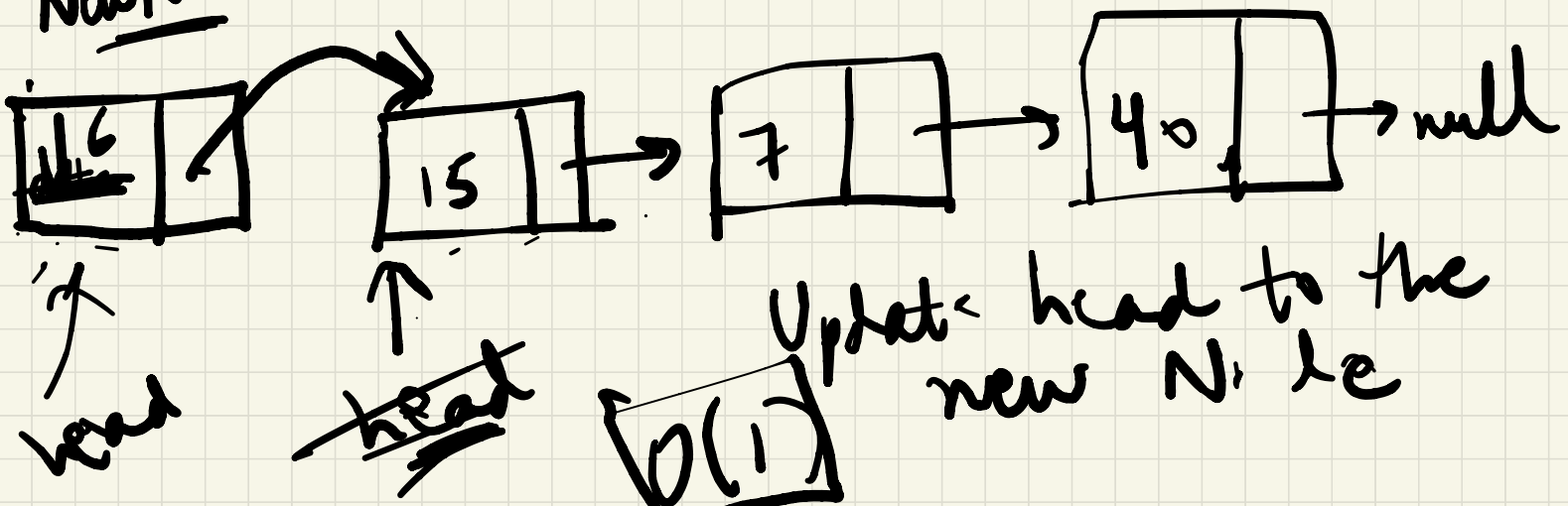
3) " " " in the middle (random location)

Case1 :- Insert node at the beginning :-

↳ new node is inserted before the current head node.

↳ the only case where head of LL changes

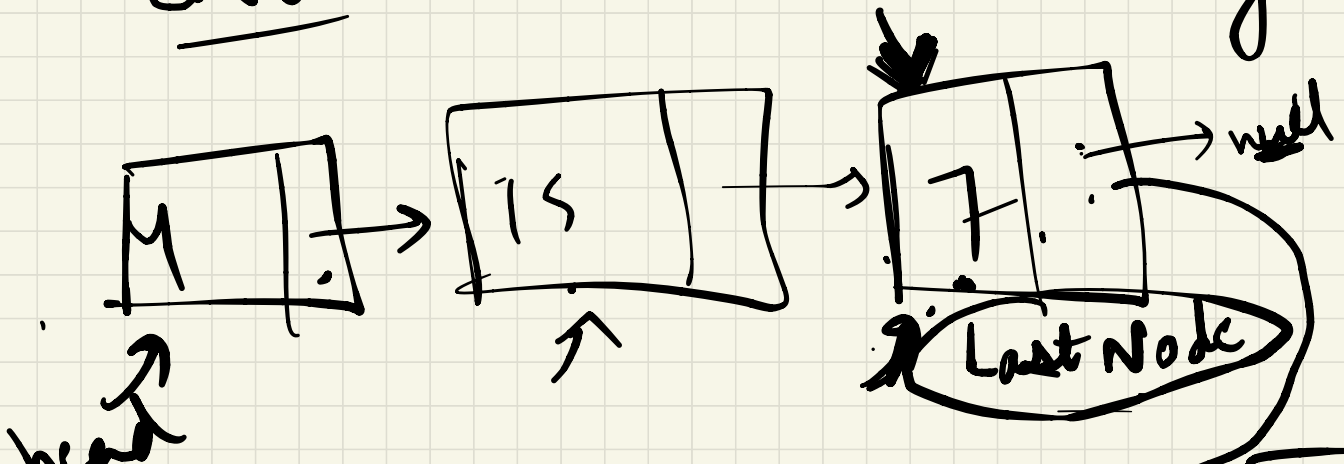
head = newNode



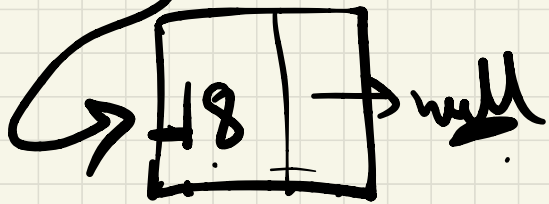
```
public static Node insertAtStart(Node head, int data){

// Step1 : create a new node
    Node newNode = new Node(data);

// Step 2: Set next of newNode to current head
    newNode.next = head;

// Step 3: Update the head pointer to the new node
    //head = newNode;
    //return head;
    return newNode;
}
```

Time complexity : O(1) to insert at the head

Case 2:- insert at the ending

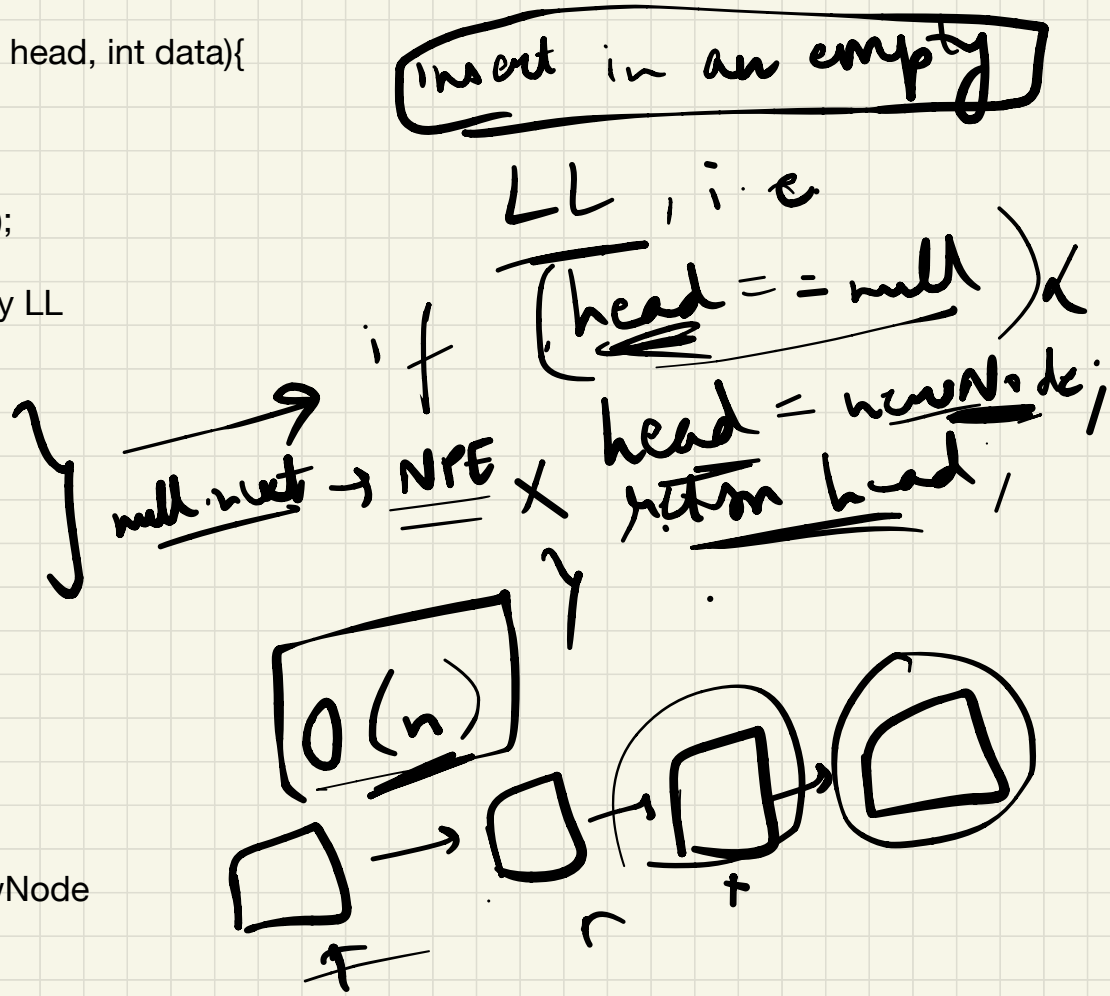


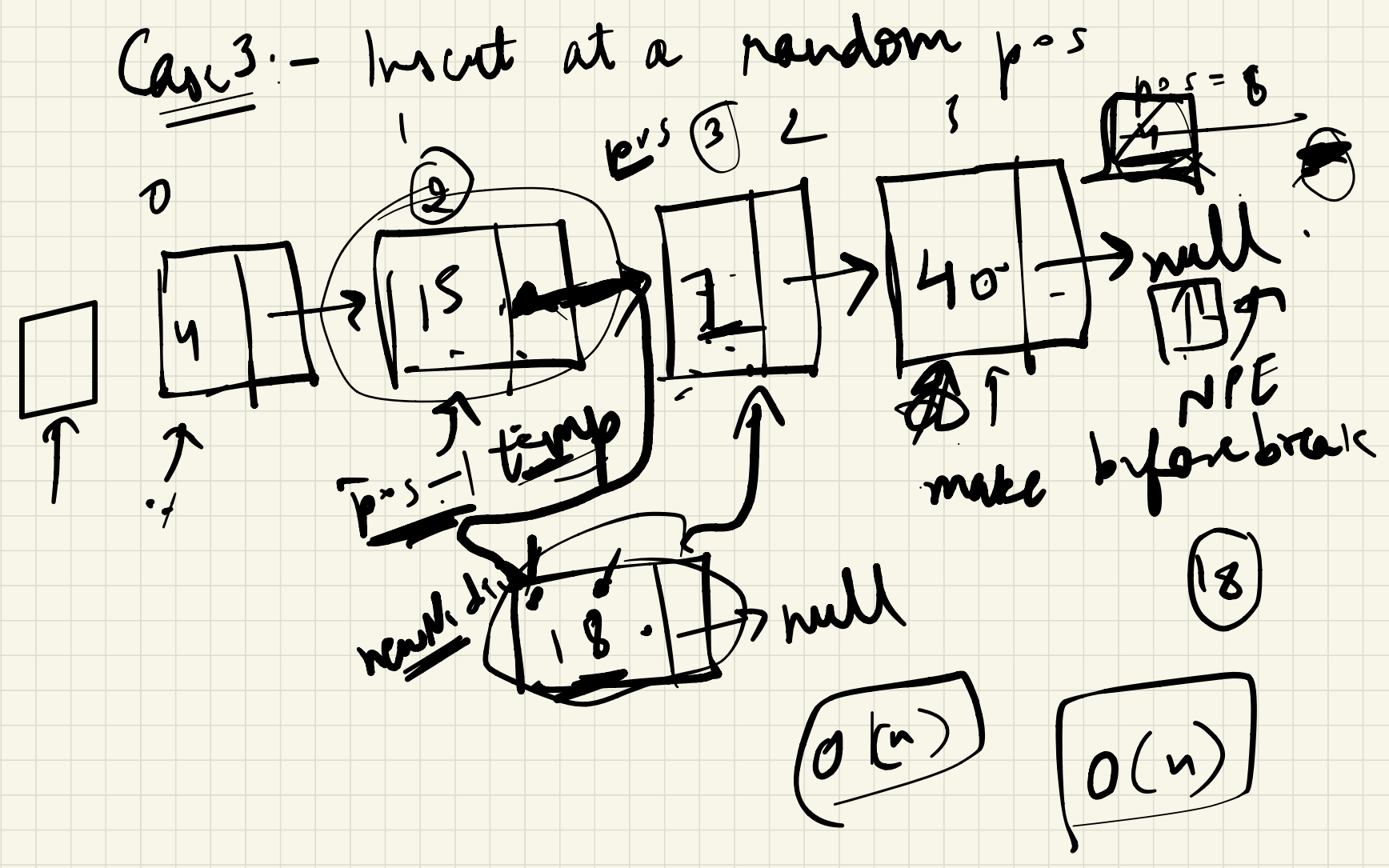
lastNode.next = newNode



condition for lastNode
till
↳ temp.next != null newNode

```
public static Node insertAtEnd(Node head, int data){

    // Step 1: create a new node

    Node newNode = new Node(data);

    if(head == null){ // in case of empty LL
        head = newNode;
        return head;
    }

            . .       .
            :  .

// Step 2: Traverse till the last node.
    Node temp = head;
    while(temp.next != null){
        temp = temp.next;
    }


// Step 3: Set lastNode's next to newNode
    lastNode.next = newNode;
      temp

temp.next = newNode;
```

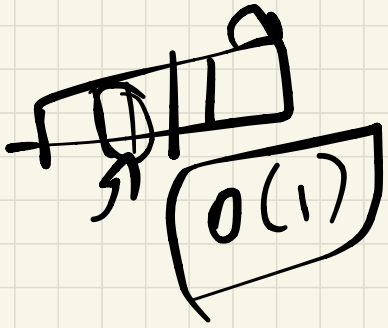



Case 3 :- Insert at a random pos
0
1
2
pos 3
2
3
pos = 6
4
15
7
40
null
temp
pos - 1
NPE
make before break
newNode
18
null
18
O(n)
O(n)

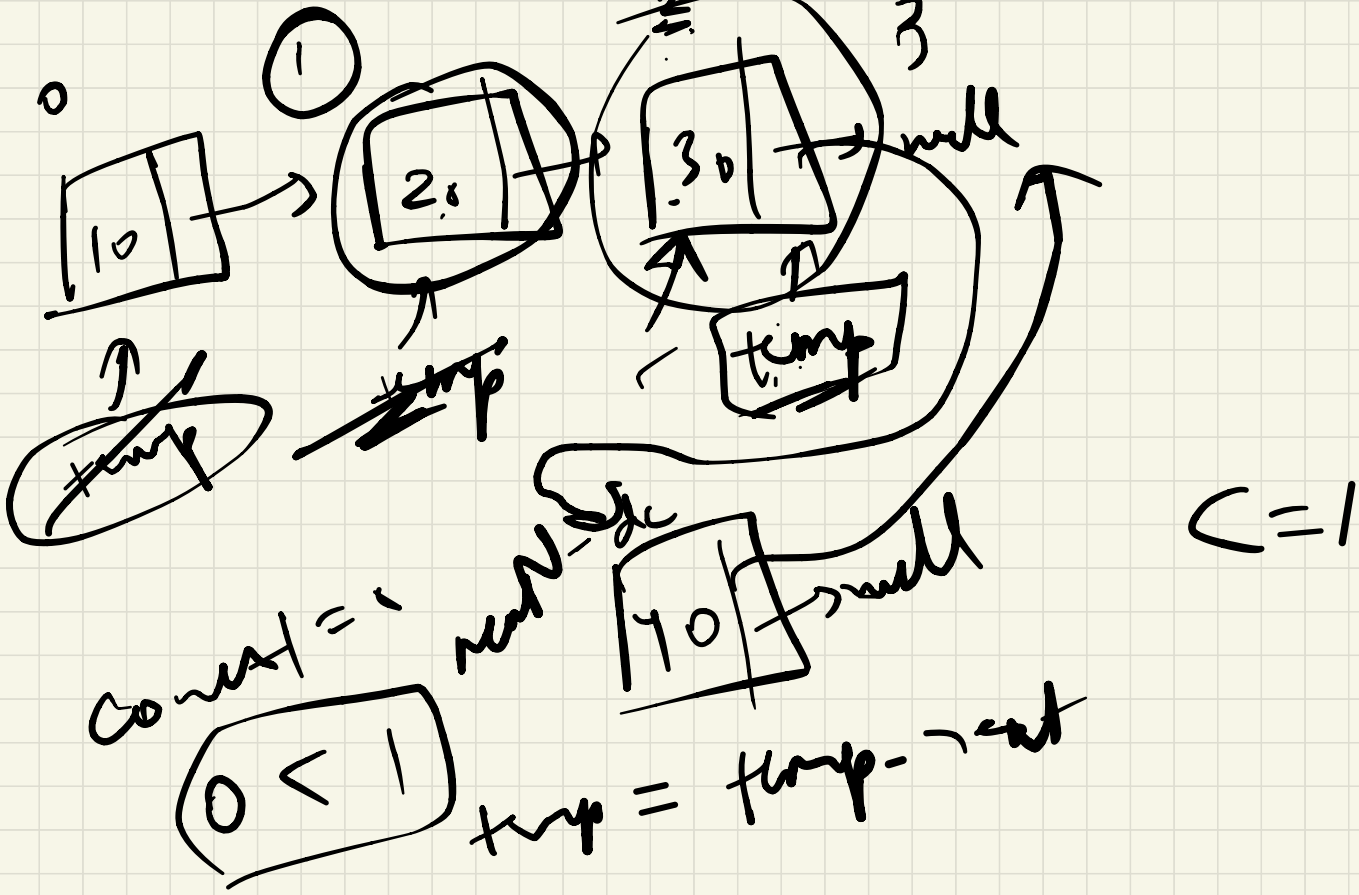
Insertion in LL

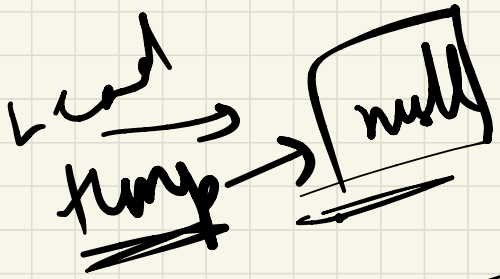
↳ O(n) ✓

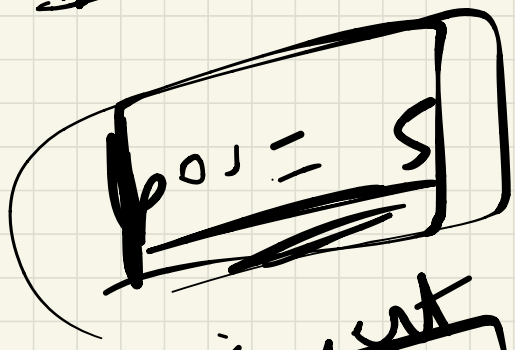
O(1)

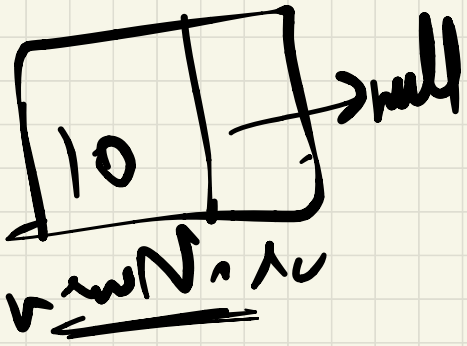


pos = 2 → insert at Head


0
1
2
3
10
20
30
null
temp
temp
temp
10
null
count = 1
0 < 1
temp = temp->next
<=1







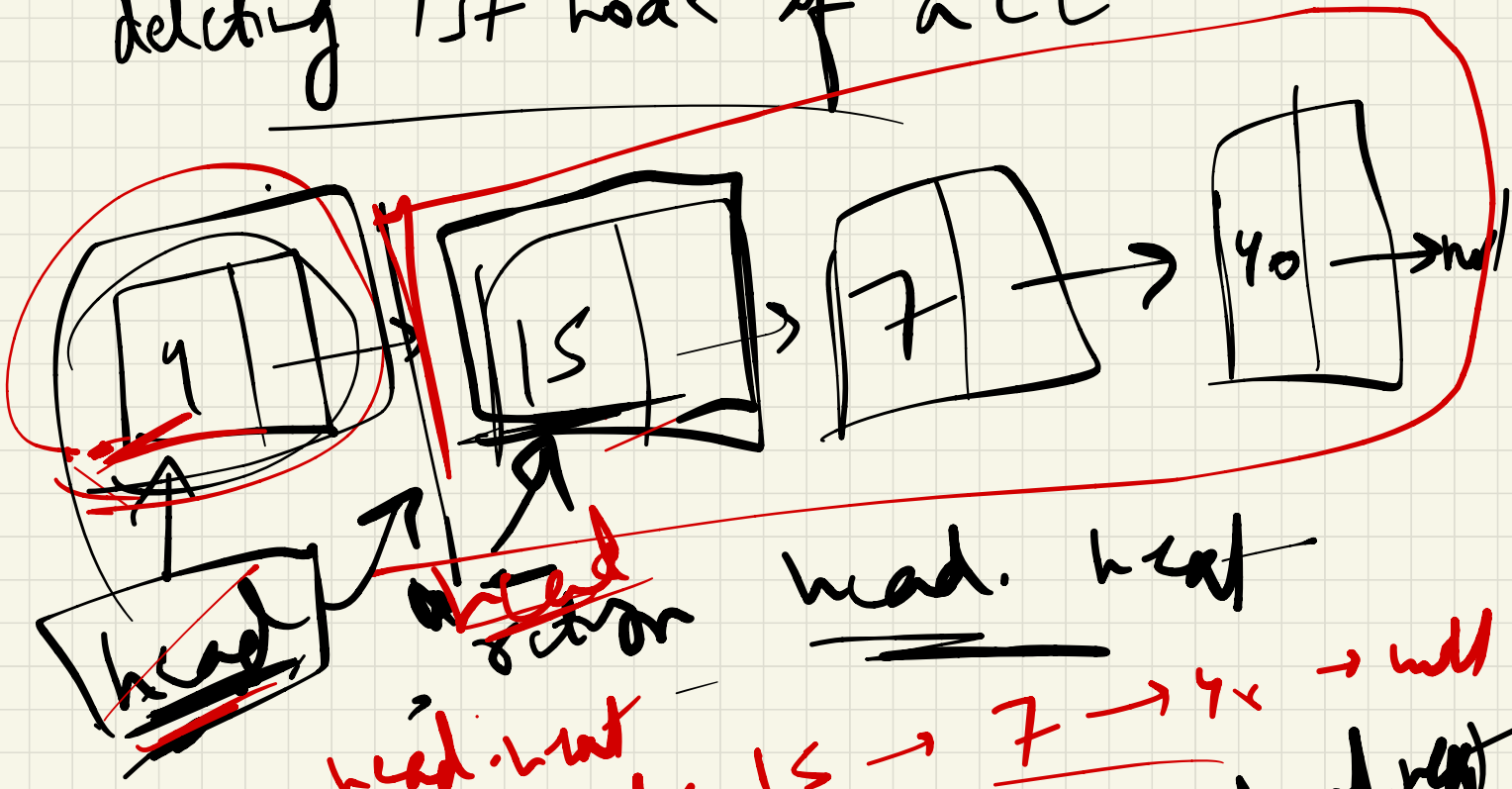
insert(10);

null.next ✗ NPE

Delete a Node

- 1st Node
- last Node
- random pos

deleting 1st node of a LL



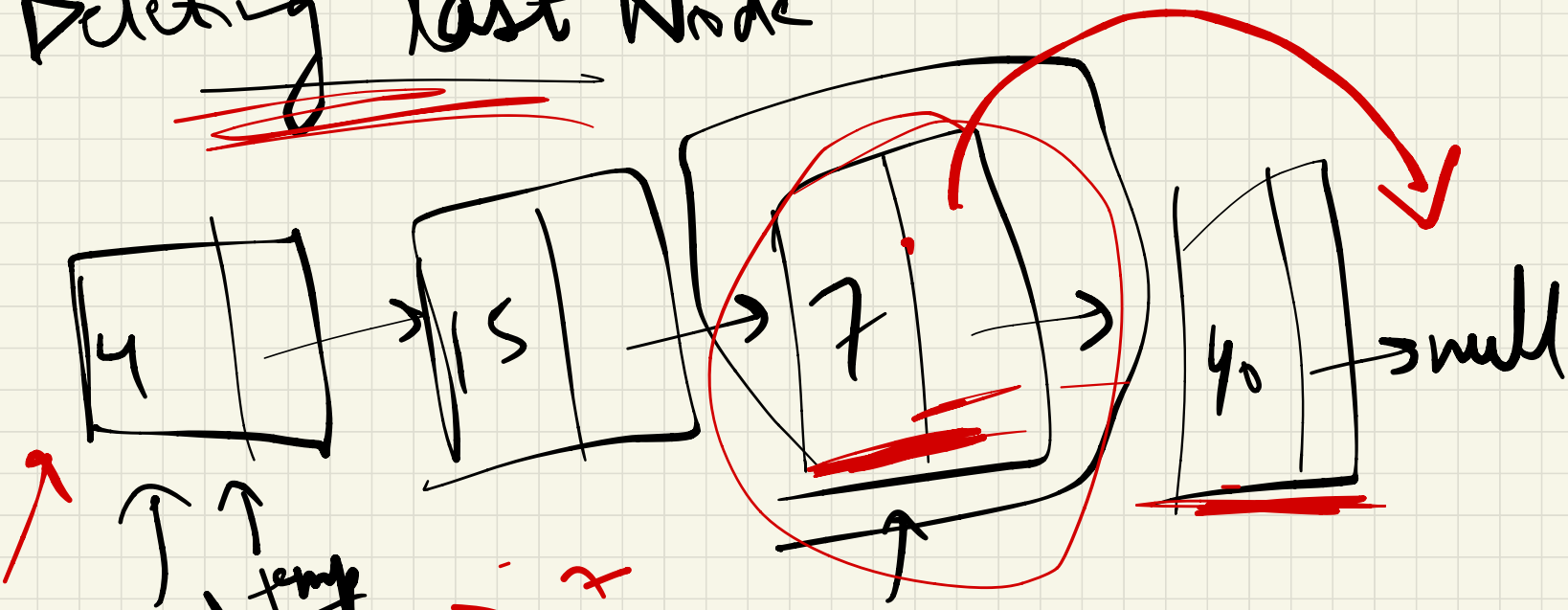
head = head.next
return head;

head. next

15 → 7 → 40 → null

head = head.next

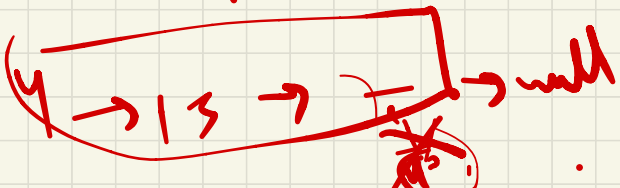
Deleting last Node



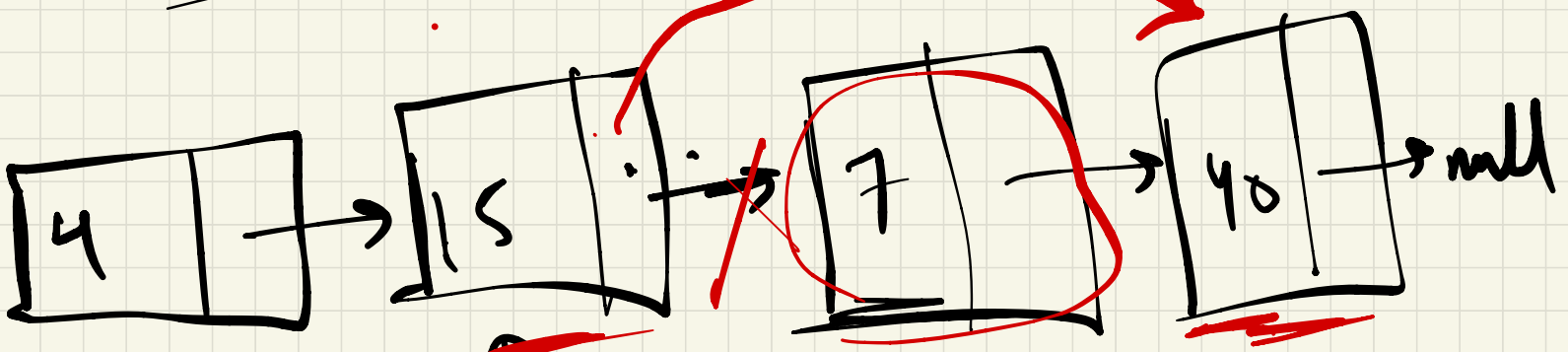
2nd last node

temp.next.next != null

temp.next = null



Delete in random fn



temp.next = temp.next.next

prev

temp

node to Be deleted

prev.next = ntb.next

ntb

4 → 15 → 40 → null

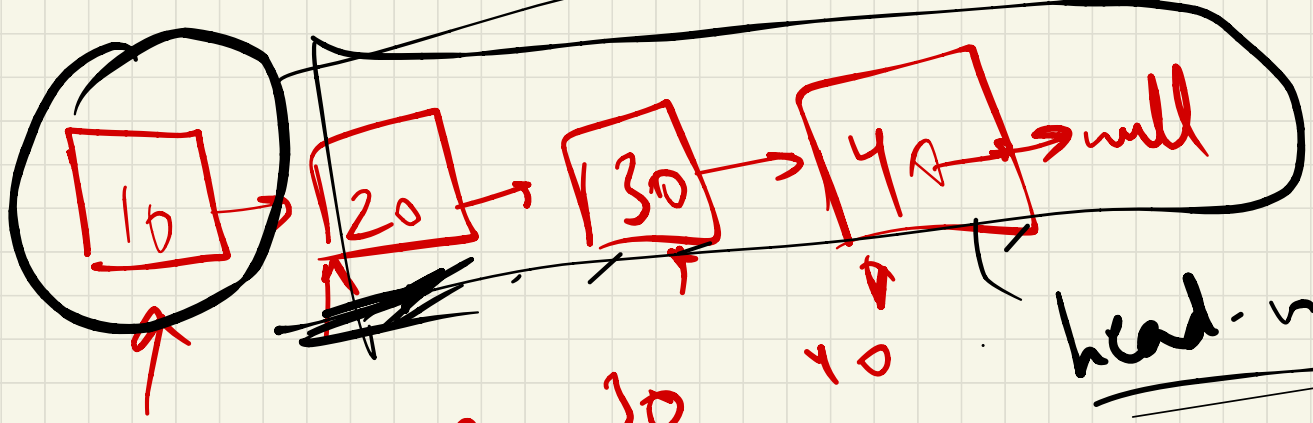
4 → 15 → 40

7

head → null

just stop

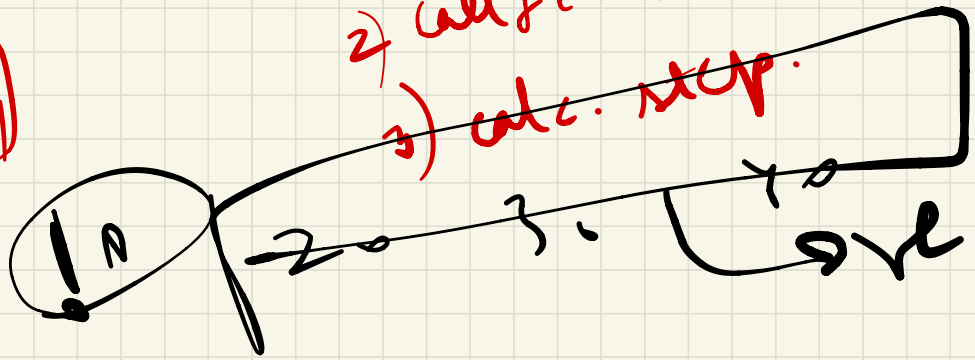
delete

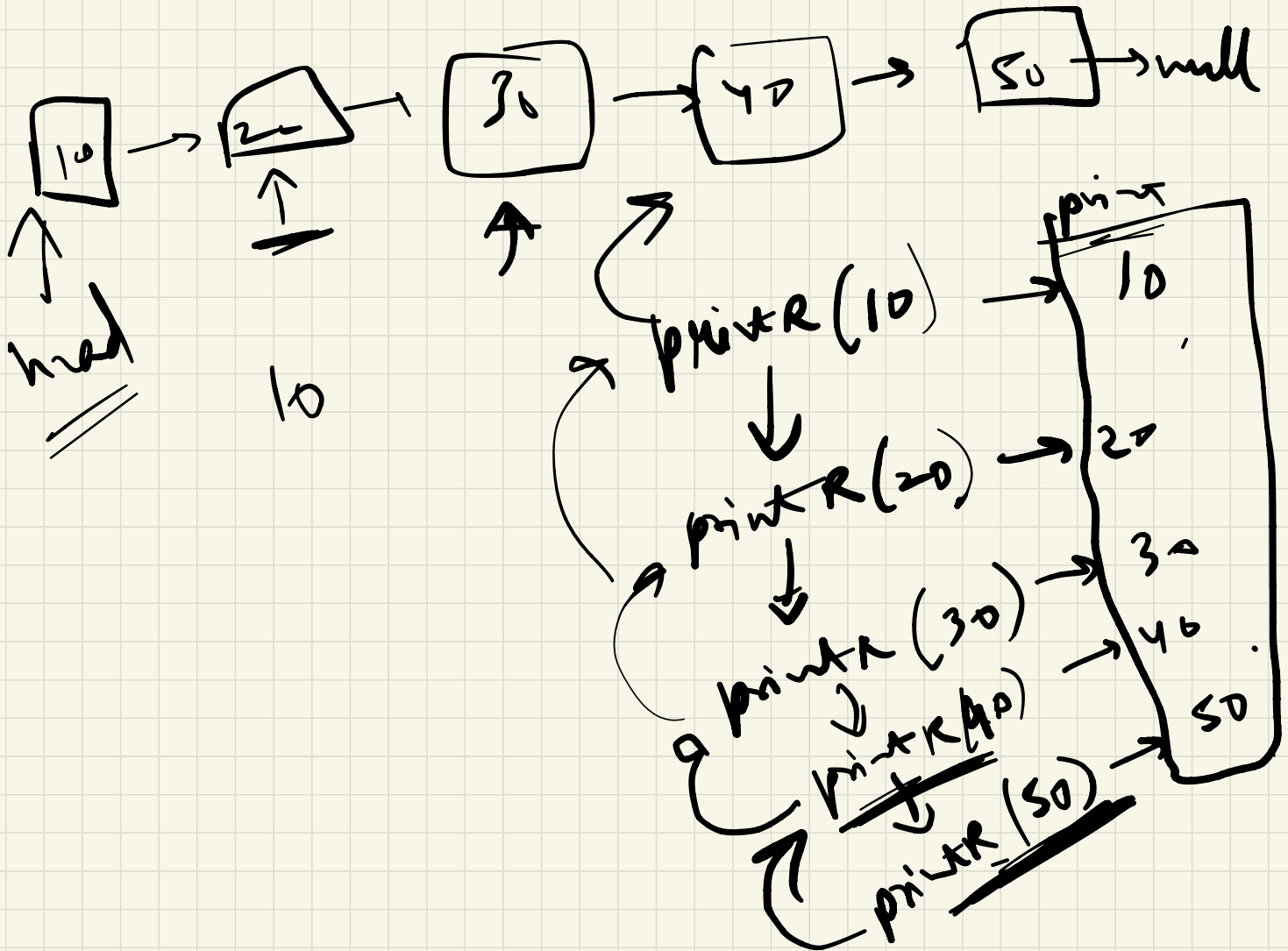


head == null

return;

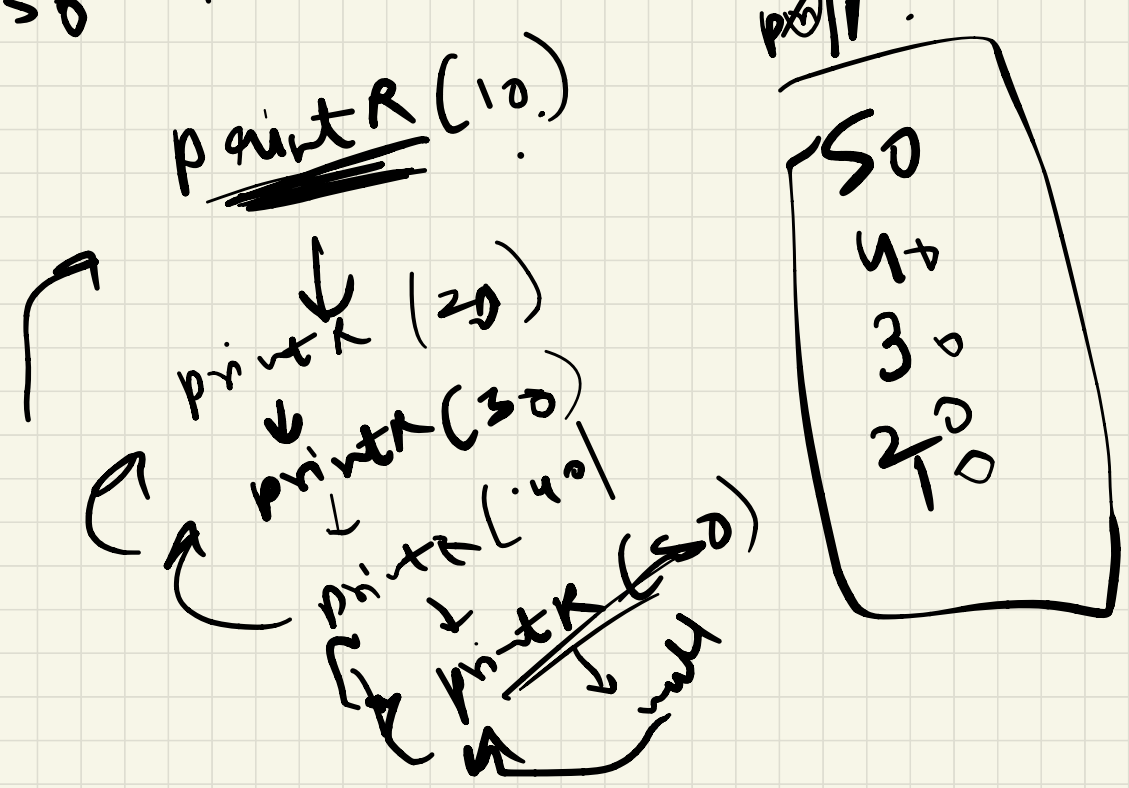
1) base case
2) call rec on smaller i/p
3) calc. step.




10
20
30
40
50
null
head
10
printR(10)
printR(20)
printR(30)
printR(40)
printR(50)
print
10
20
30
40
50

i/l : 10 → 20 → 30 → 40 → 50 → null

o/p : 50 40 30 20 10



Insert Recursively

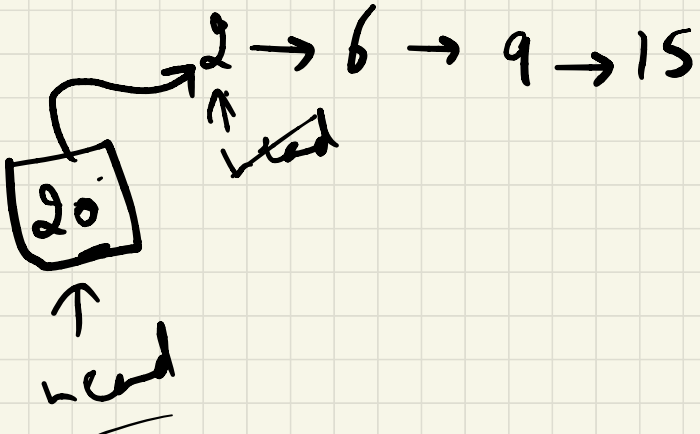
Steps:-

1) If head is null & pos is not zero.
   Exit.

2) If head is null & pos = 0.
   insert a new Node to the Head
   & exit.

3) If Head is not null & pos = 0.
   ↳ newNode.next = head
   head = newNode
   ↳ update head reference to the new node.

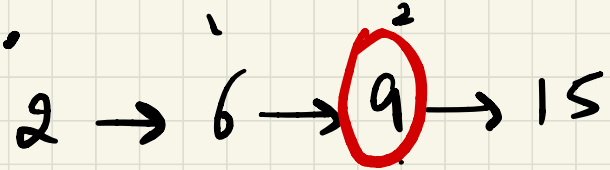
4 Nth pop or end



(val, pos)

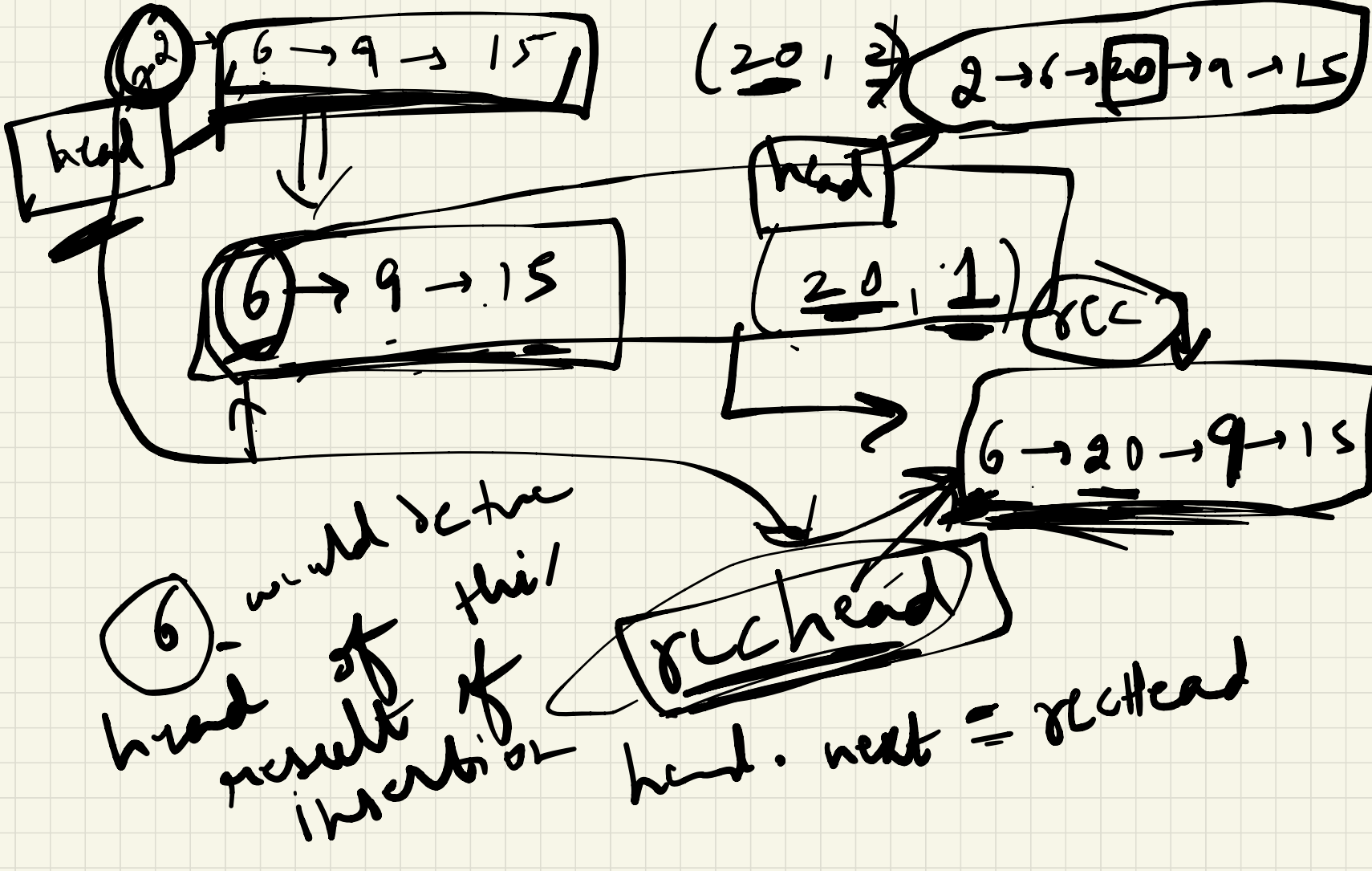
(20, 0)

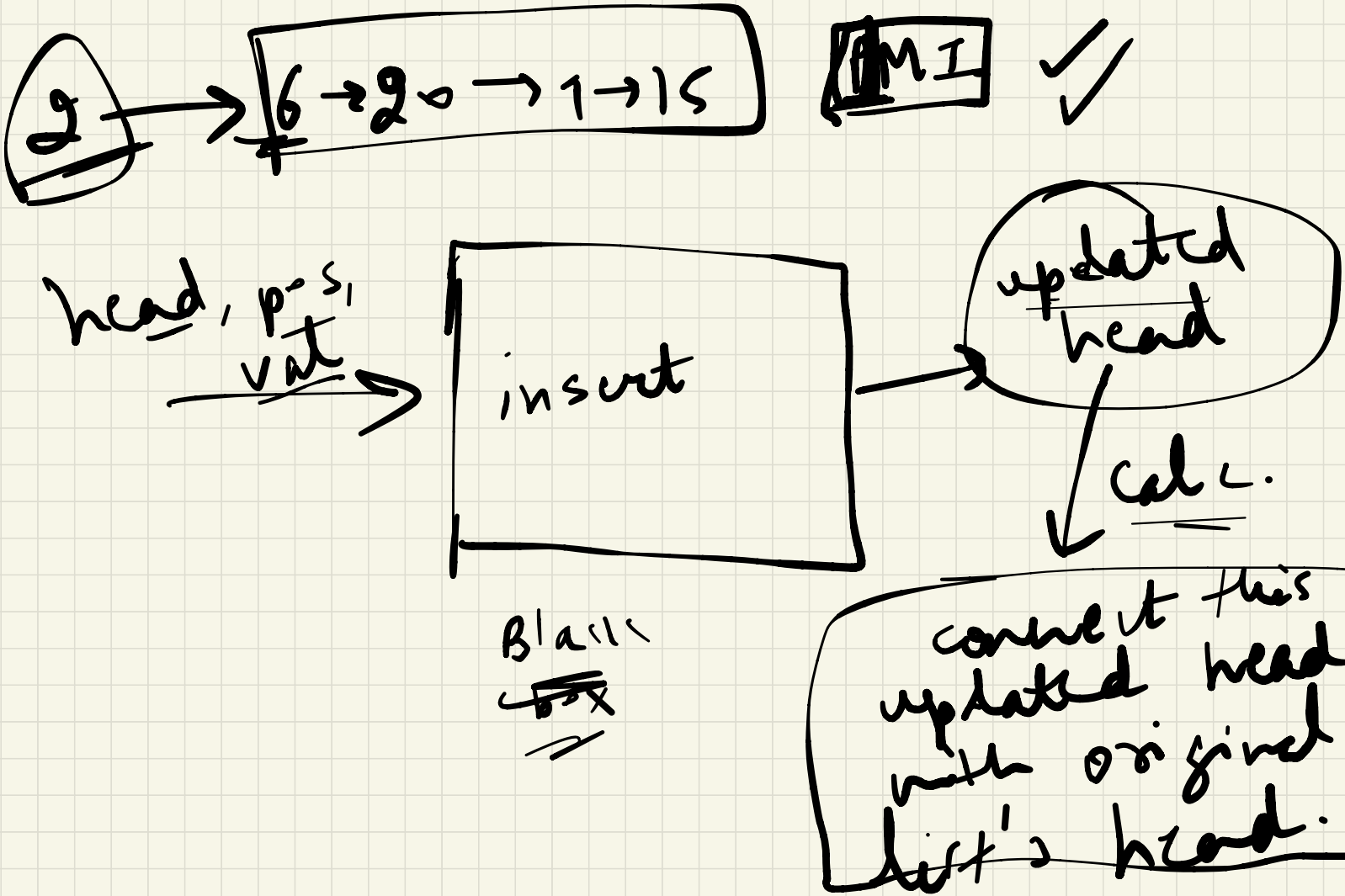
The only case where head changes



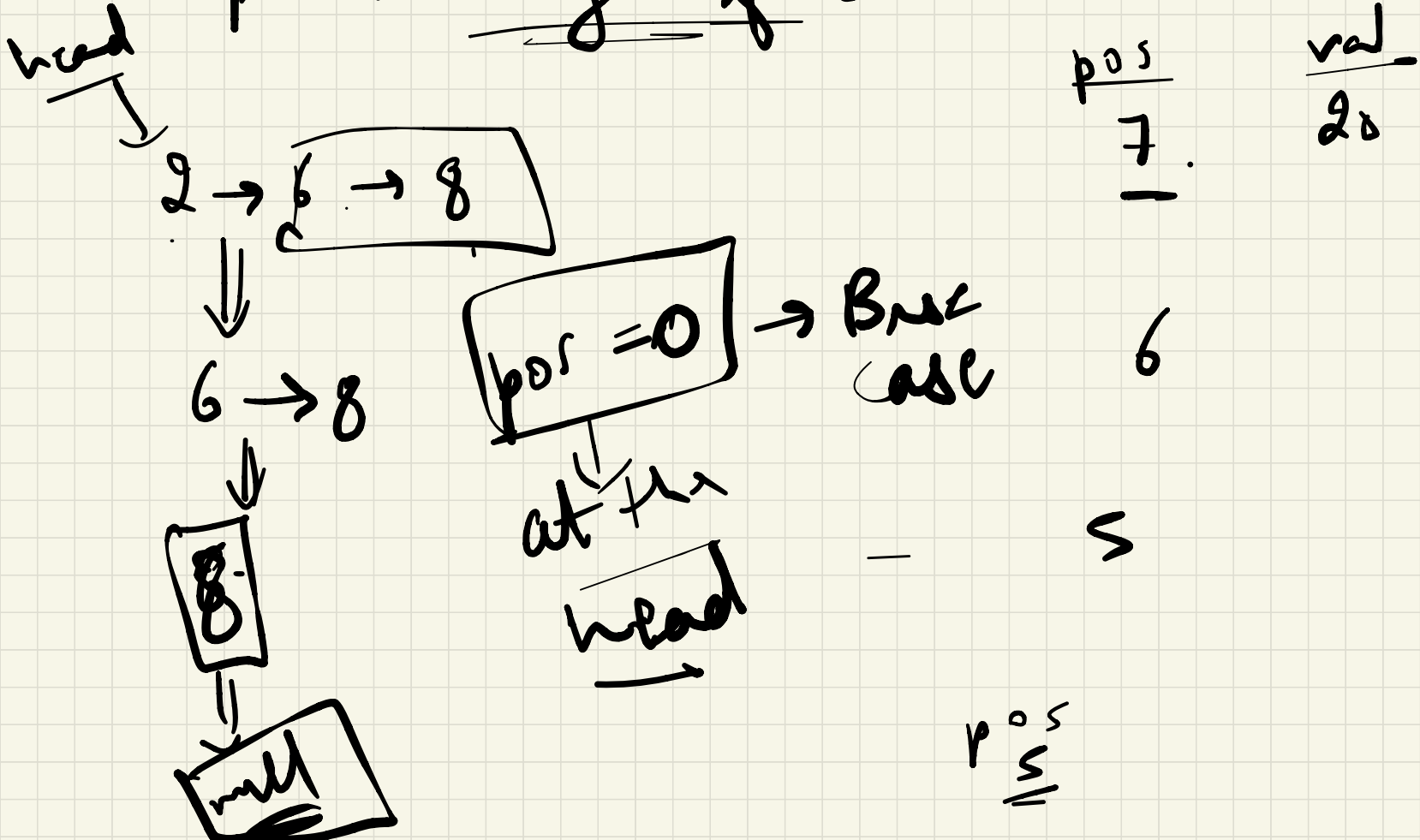
(val, pos)

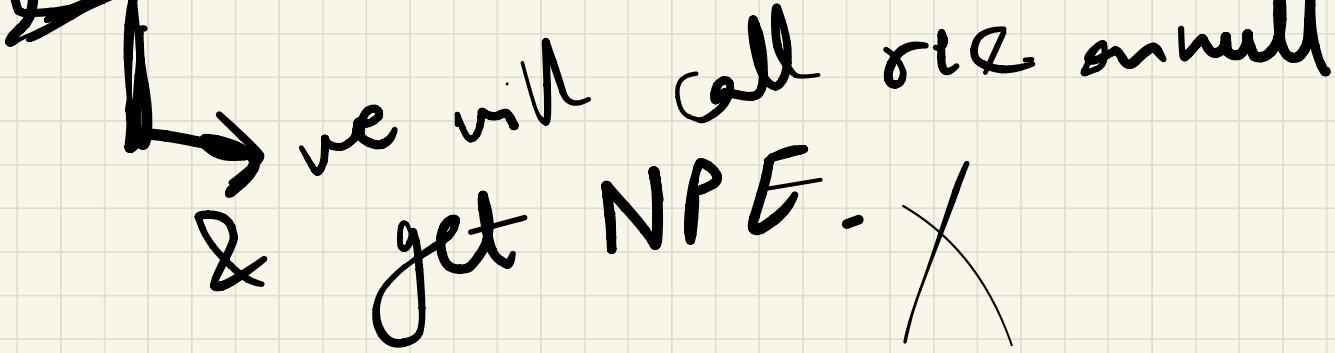
(20, 2)


head
6 → 9 → 15
6 → 9 → 15
head
(20, 2)
2 → 6 → 20 → 9 → 15
(20, 1)
rcc
6 → 20 → 9 → 15
rcchead
6 would be the head of this/ result of insertion
head.next = rcchead

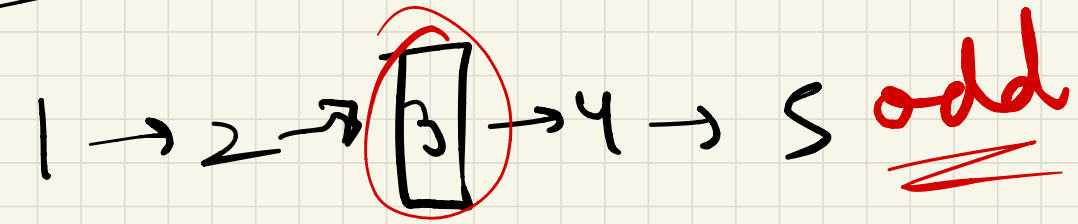

2
6 → 20 → 1 → 15
MI
head, pos, val
insert
updated head
call.
Black box
connect this updated head with original list's head.

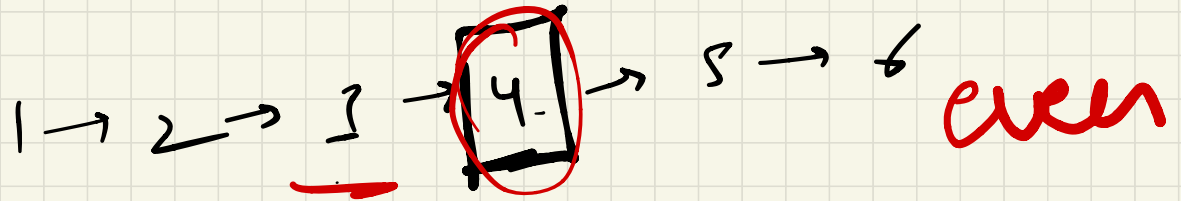
pos > length of LL

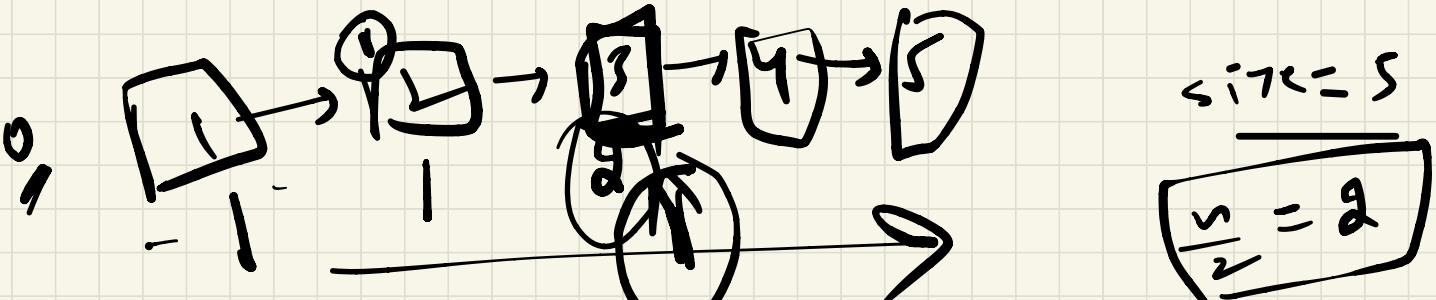



we will call str on null
& get NPE. X

Middle of LL







1st approach

O(N) → count total no. of nodes i.e. size.

O(N/2) → iterate till (n/2)th V's. & return it

2 pass

2 pointer approach

slow, fast approach

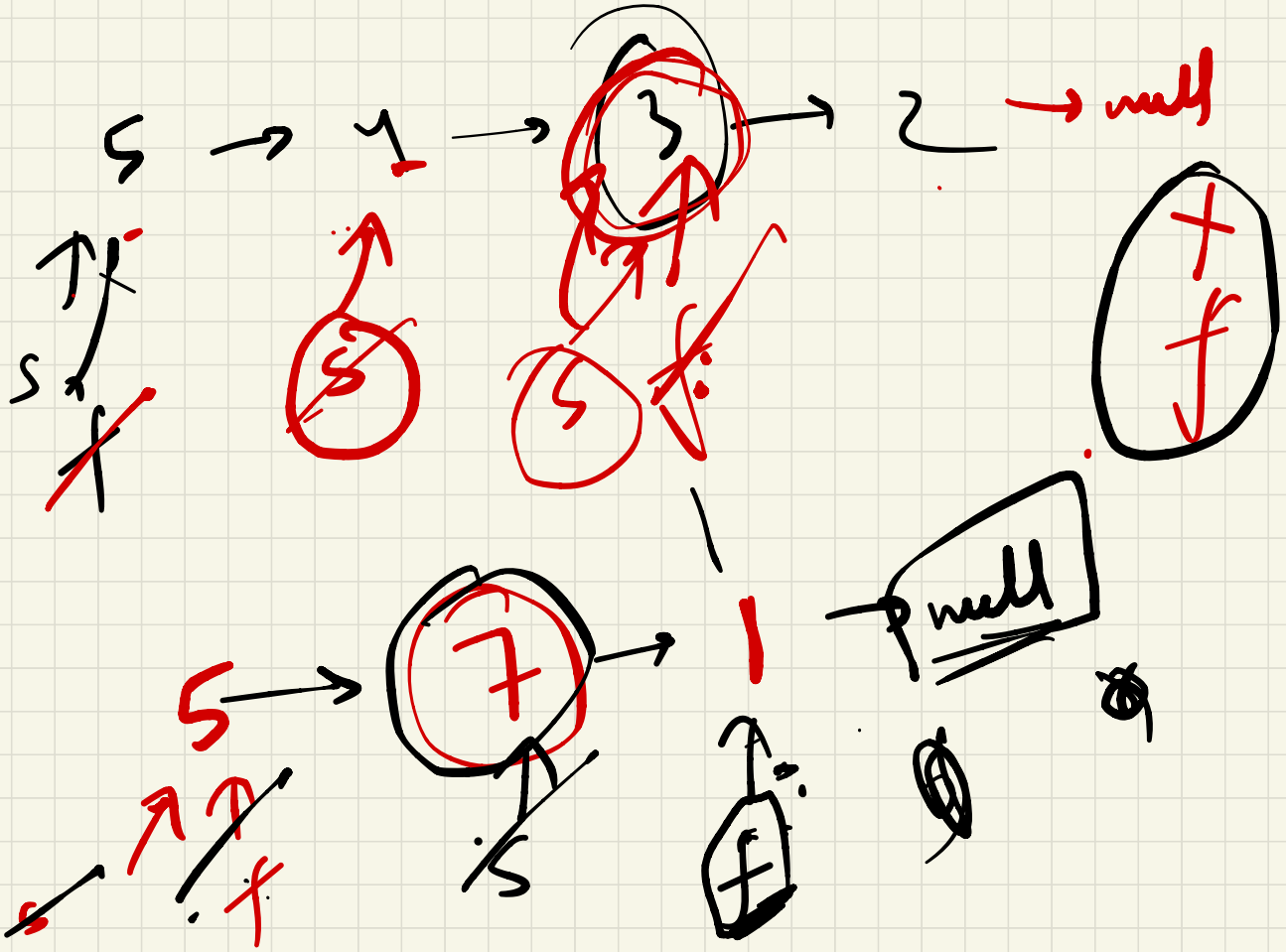
→ 2 ptrs → slow & fast.

slow → jumps 1 time
fast → jumps 2 time
} in 1 iteration

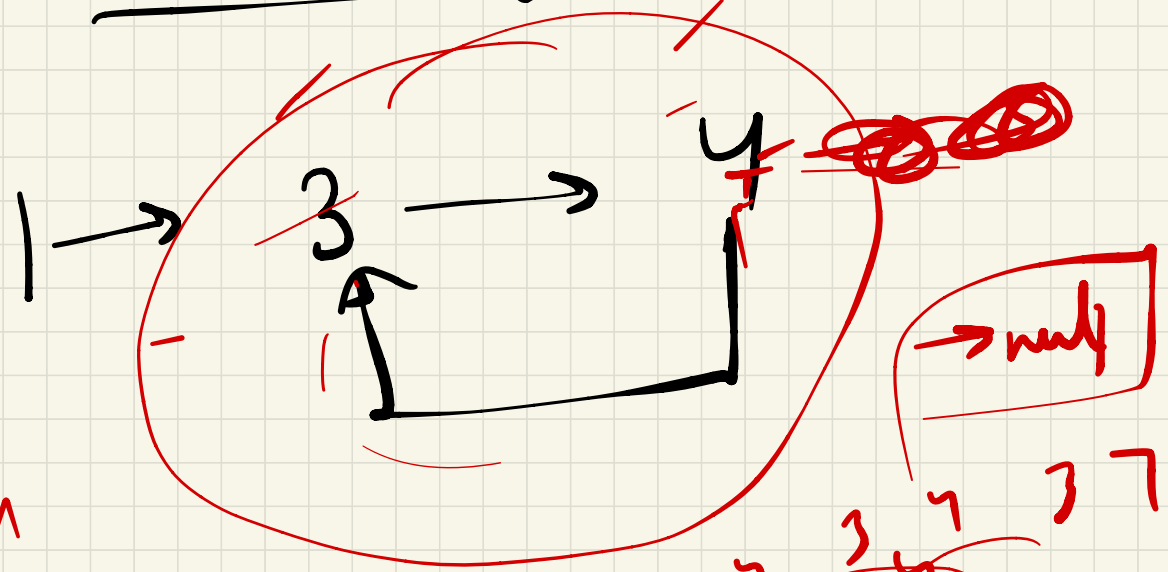
O(n)

→ when fast reaches end, slow is at the middle of LL.

pass


null
null

Detect loop / cycle in LL



Slow, fast / 2 pointer approach

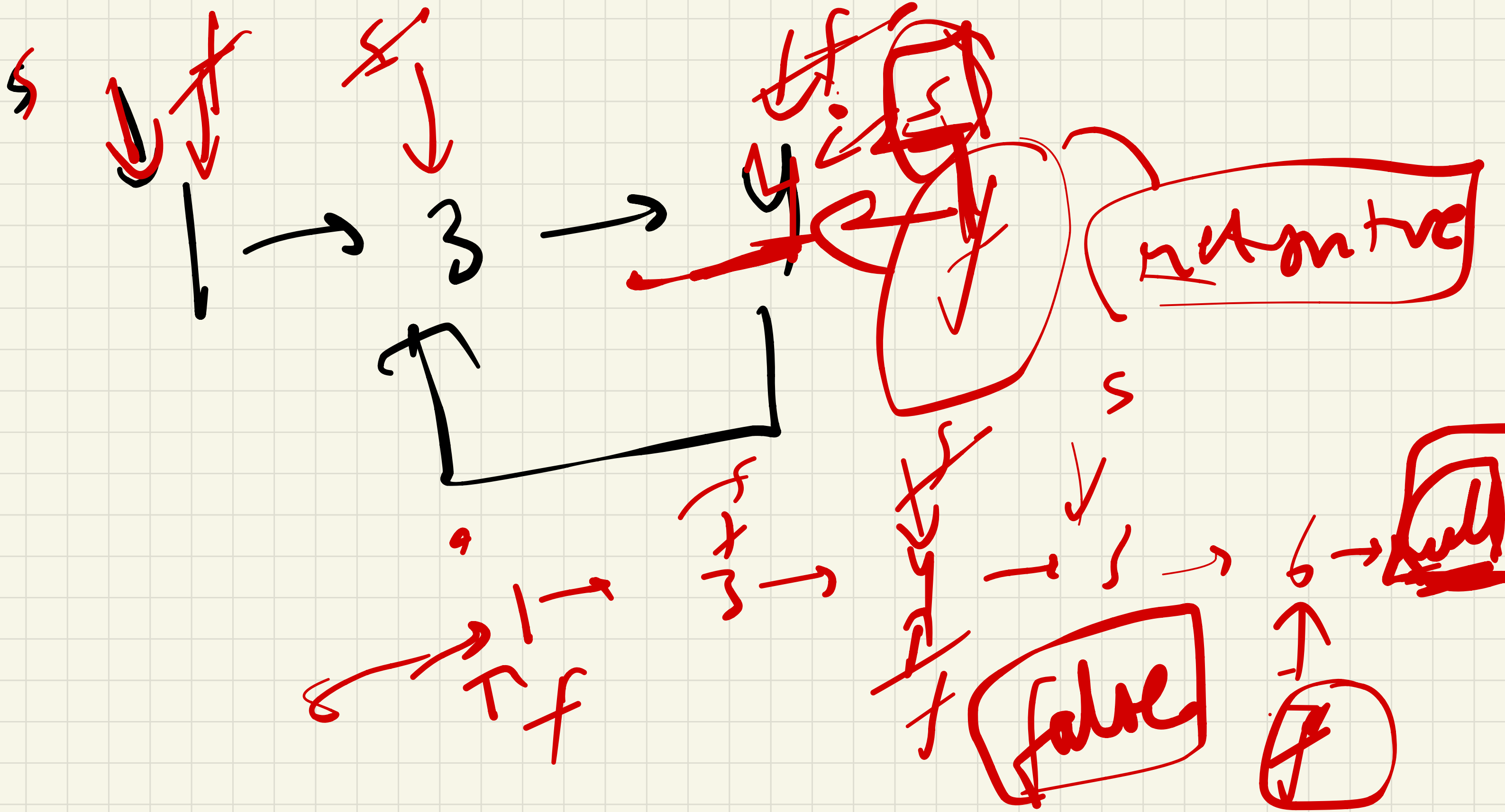
Floyd's cycle detection algo

steps

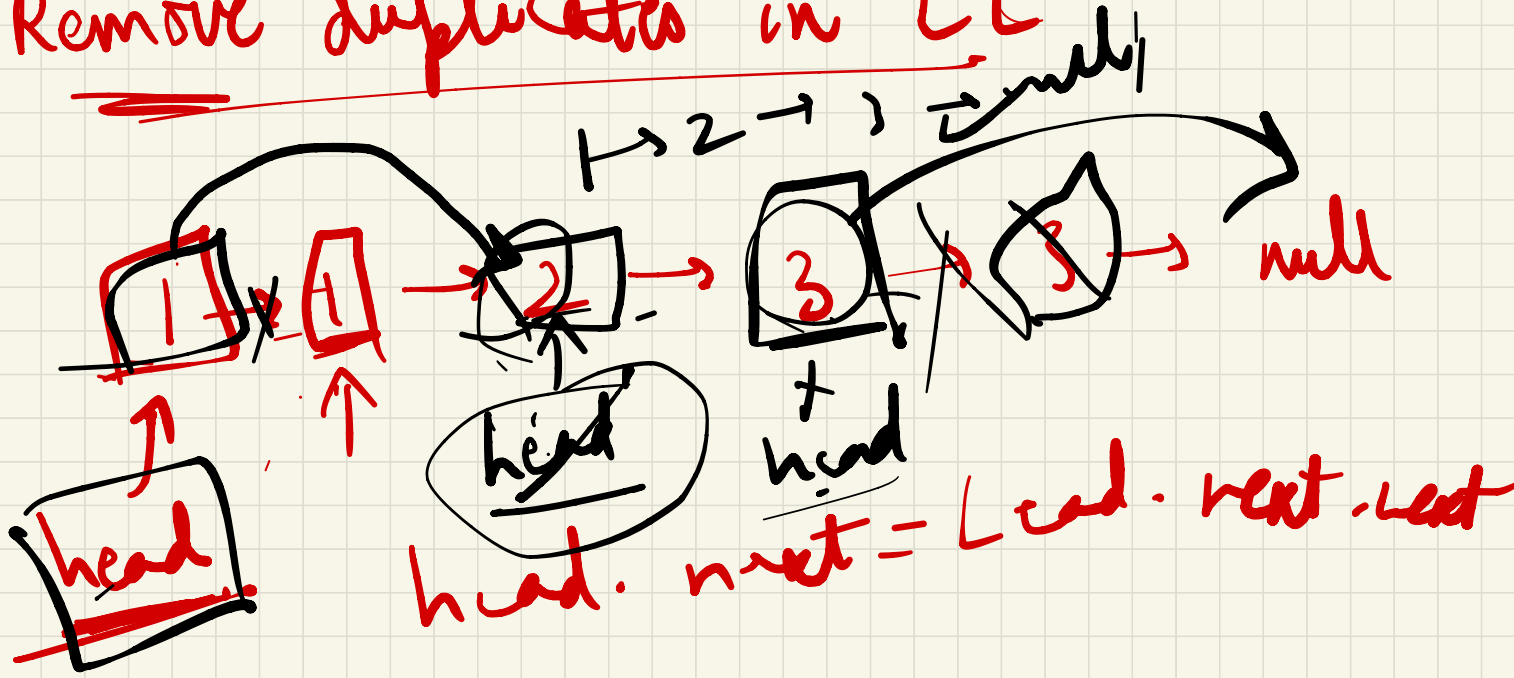
1) Traverse the list using 2 pointers.

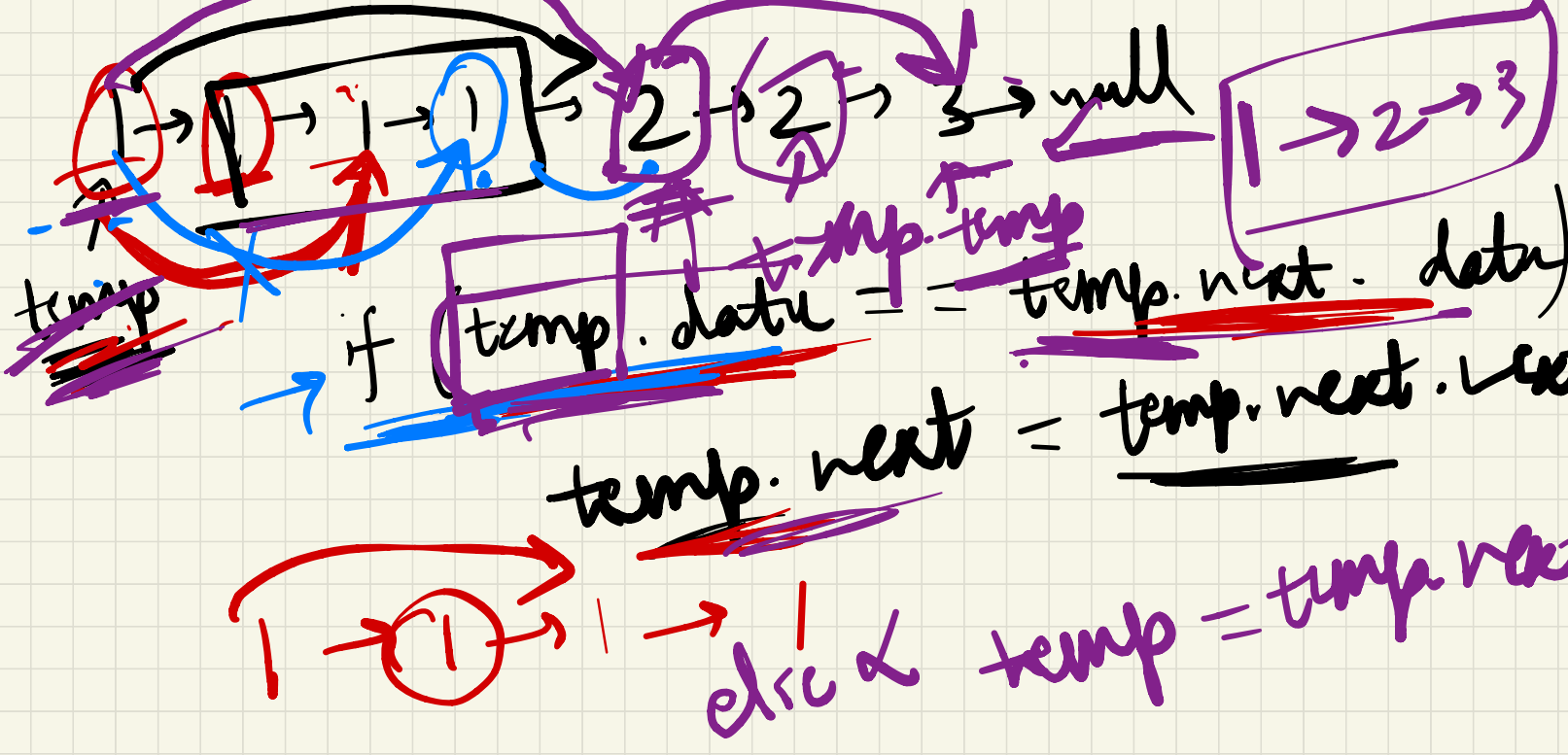
2) Move 1 pointer (slow) by one & fast by 2.

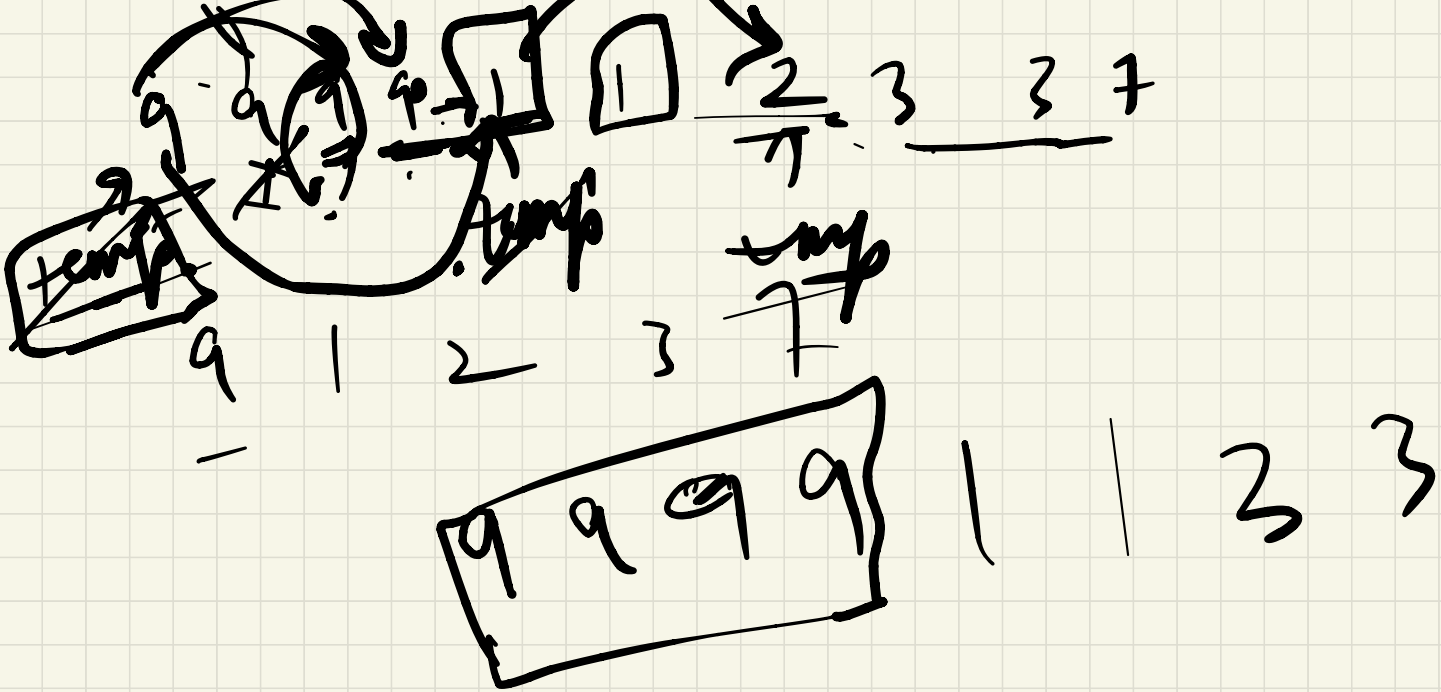
3) If these ptrs meet at the same node then there is a LOOP.
Else No loop


return true
false

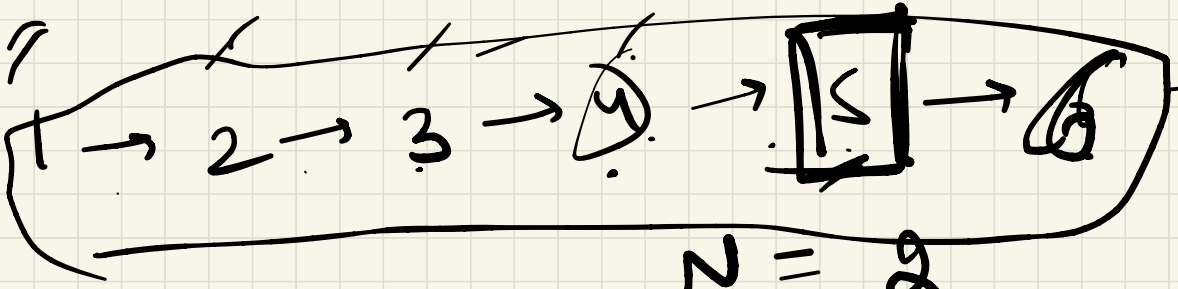
Remove duplicates in LL




1 → 1 → 1 → 2 → 2 → 3 → null
1 → 2 → 3
temp
if (temp.data == temp.next.data)
temp = temp.next
temp.next = temp.next.next;
else temp = temp.next

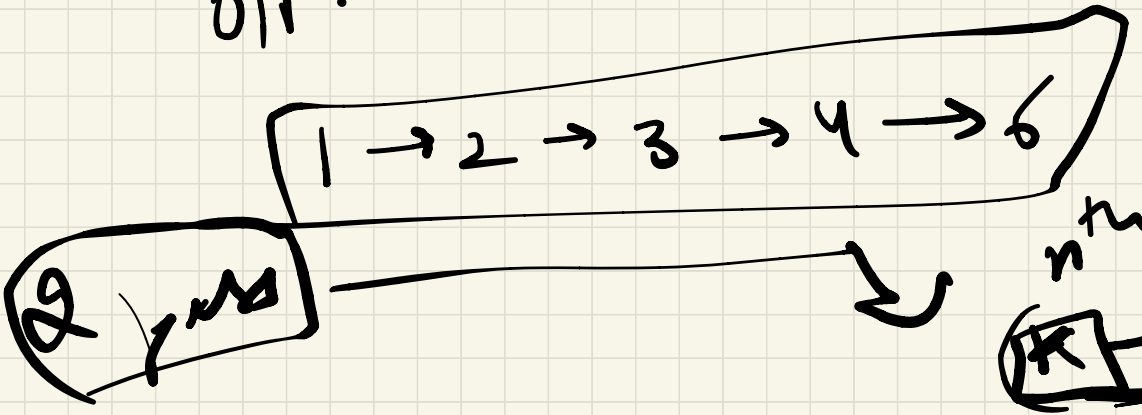

temp
temp
temp
9 1 2 3 7
9 9 9 9 1 1 3 3

Remove N^{th} node from end



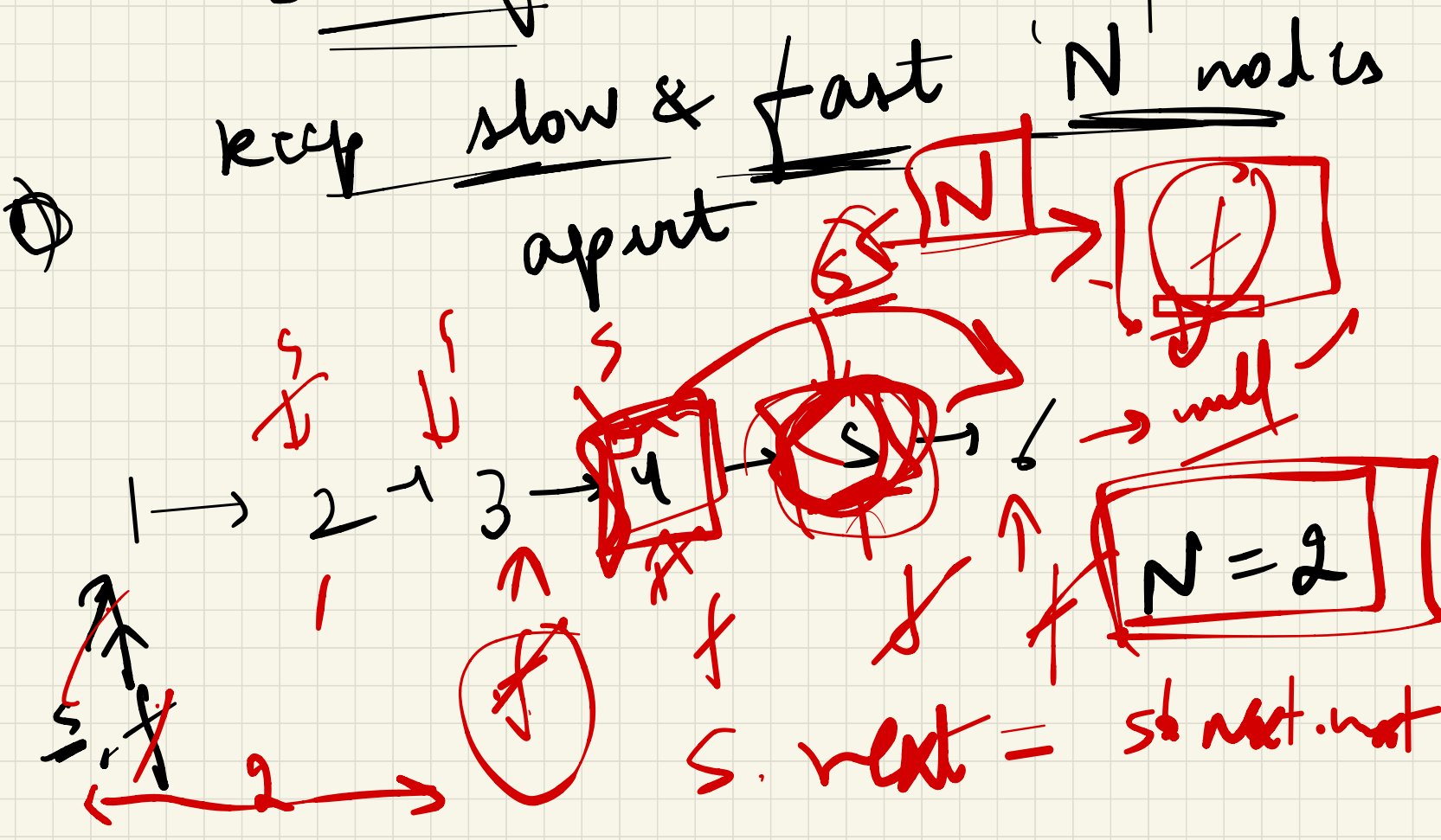
O/P: $6 - 2 + 1 = 4$

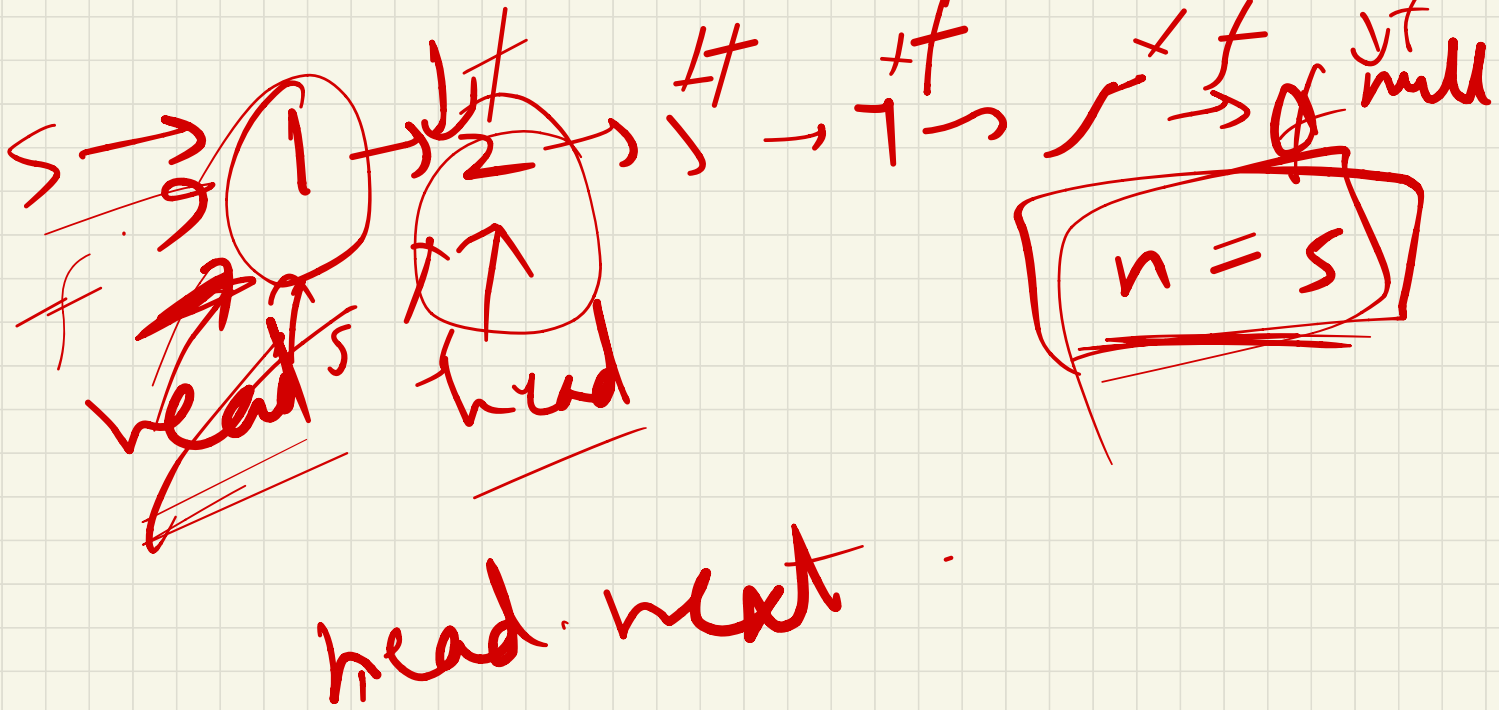
$6 - 2 + 1 = 4$



Slow, fast

keep slow & fast 'N' nodes apart





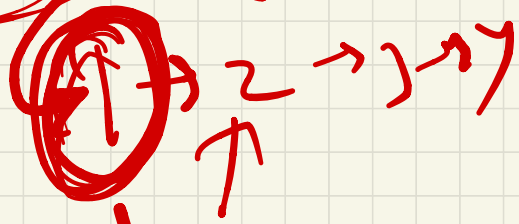
head.next

fast, slow =

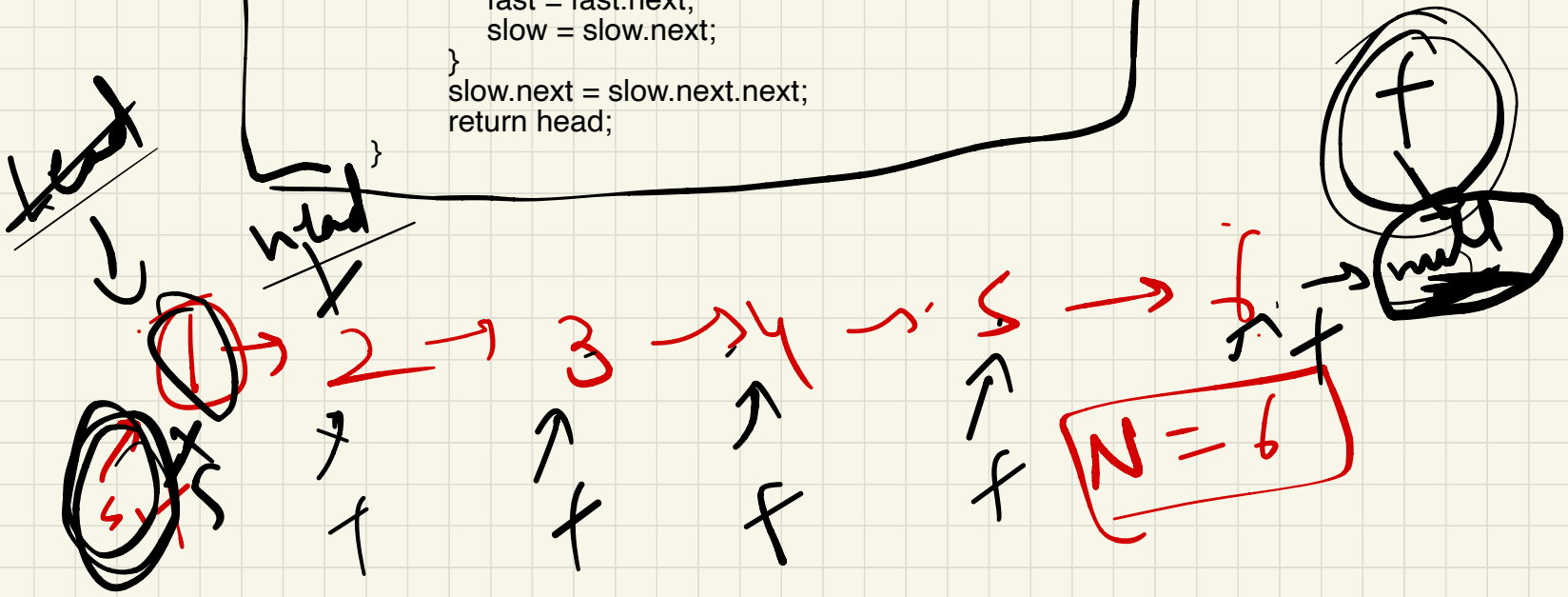
fast = dummy

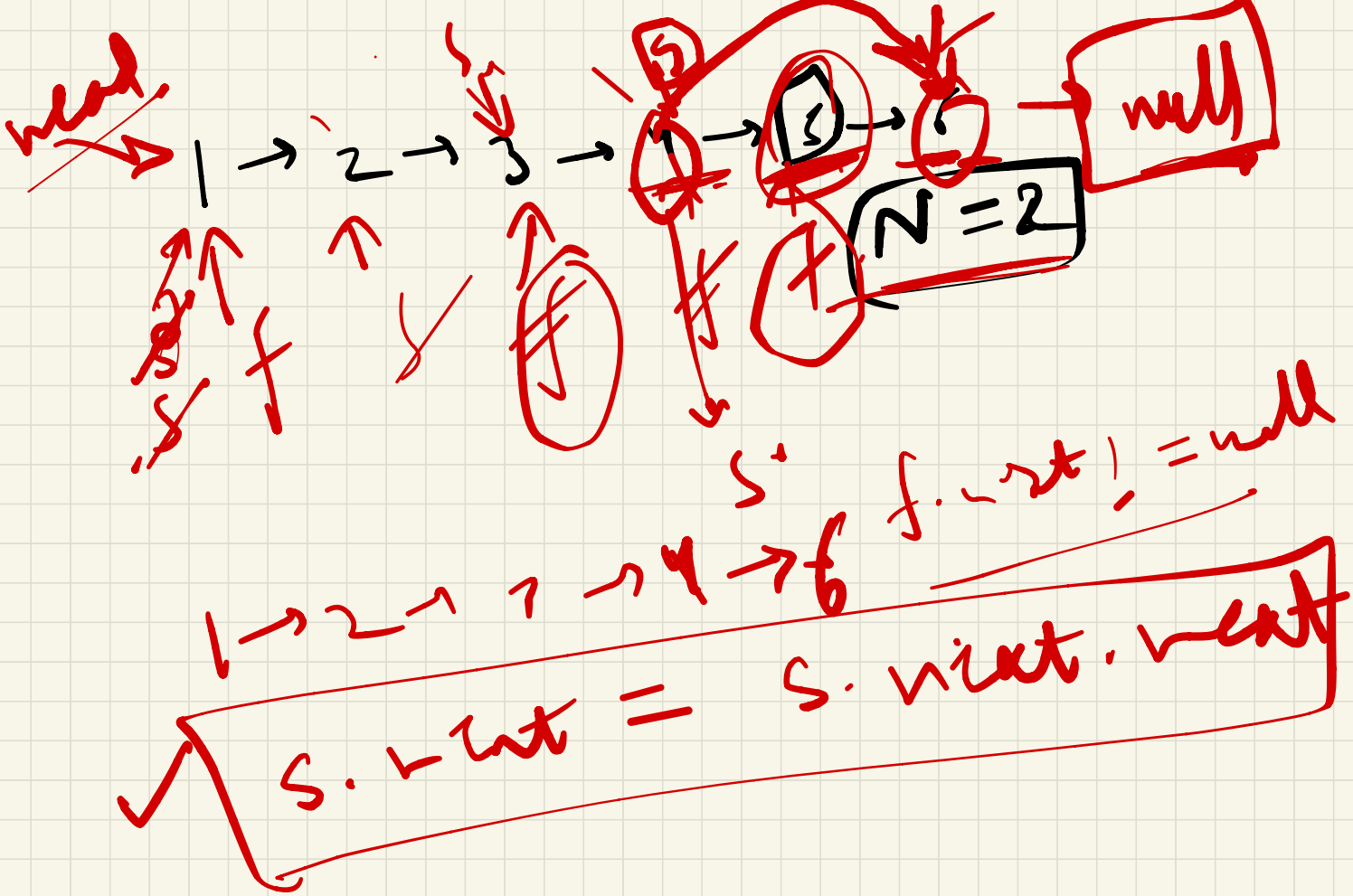
slow = dummy

dummy.next = Lnd



```
public static Node removeNthFromEnd(Node head, int n) {
    //Write your code here
        Node slow = head;
        Node fast = head;
        for(int i=0; i<n; i++){
            fast = fast.next;
        }
        if(fast == null){
            return head.next;
        }
        while(fast.next != null){
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        return head;
}
```




head
1
2
3
4
5
null
N = 2
1 → 2 → 3 → 4 → 5
s
f
s.next
f.next != null
s.next = s.next.next