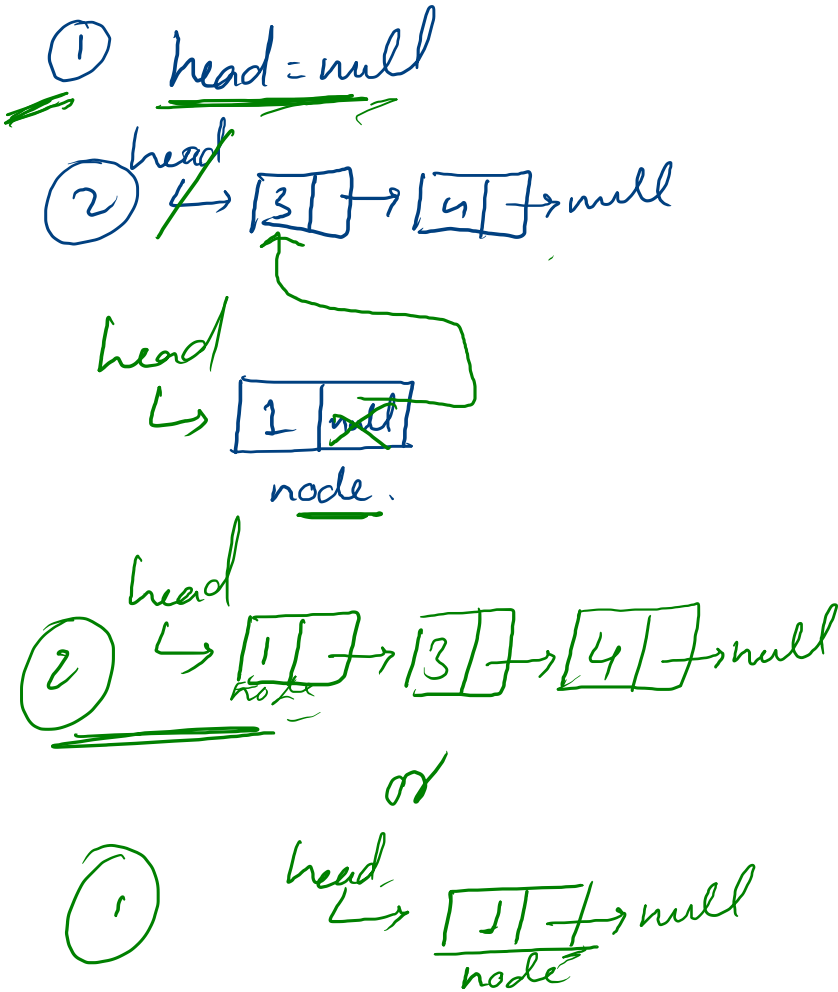
Insert a node at head:

Step 1:- Create a node

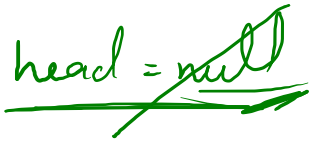
```
node = new Node(1)
```

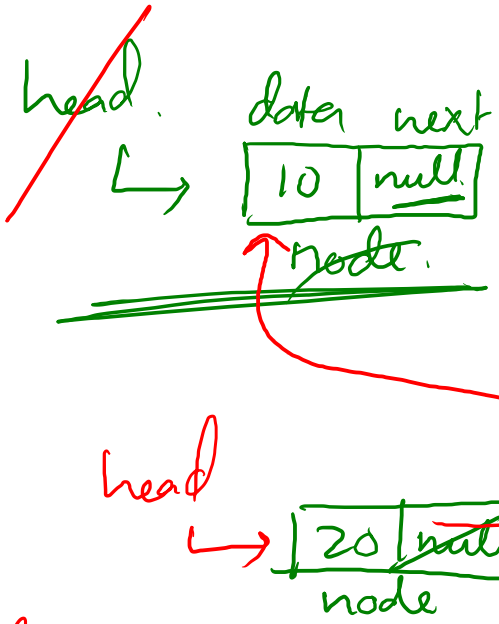
Step 2:

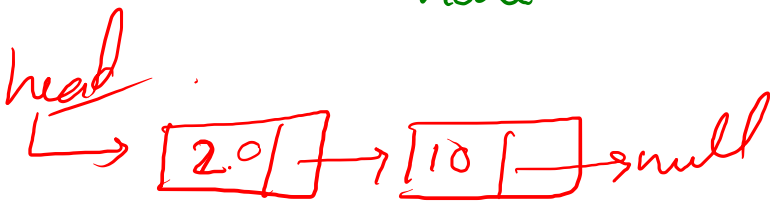
```
node.next = head
head = node
return head
```



```
const LinkedList = class {
        constructor() {
                this.head = null;
        }
};

const LinkedListNode = class {
        data
        next
        constructor(nodeData) {
                this.data = nodeData;
                this.next = null;
        }
};

function PrintLinkedList(head){
        temp = head;
        while(temp != null){
                console.log(temp.data);
                temp = temp.next
        }
}
// Complete the function below

function insertNodeAtHead(head, data) {
        var node = new LinkedListNode(data);
        node.next = head;
        head = node;
        return head;
}

function main() {
        let list = new LinkedList();
        list.head = insertNodeAtHead(list.head, 10);
        list.head = insertNodeAtHead(list.head, 20);
        PrintLinkedList(list.head);
}
```

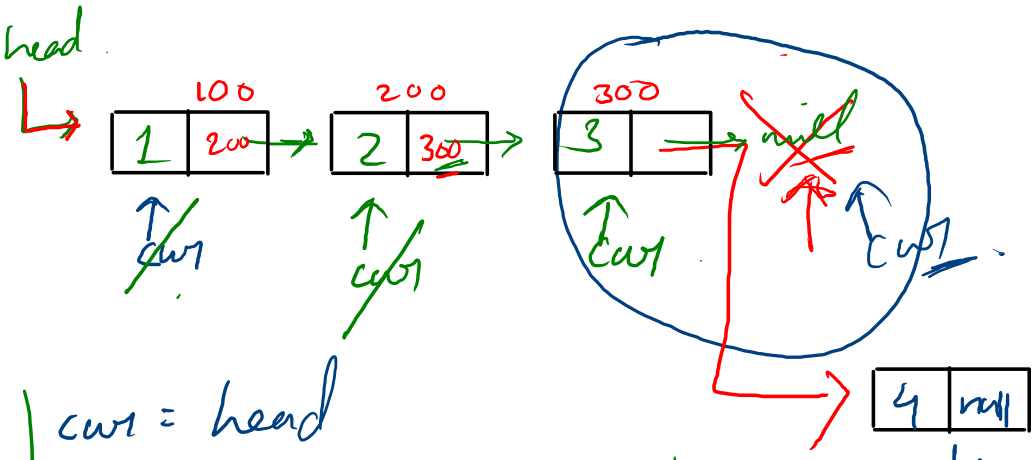






(1) Insert a node at the end/tail

on a SLL, last node's next field → null.



insert 4.

(1) create a new node having
data = 4
next = null.
node = new Node(4)

(2) take a pointer & reach at the last node.

```
curr = head
while(curr.next != null)
    curr = curr.next
                200 800
}
//when loop terminates my curr pointer is pointing at the last node.
```

null != null ✗
false
∴ term.

(3) curr.next = node
return head.

Code :- (in JavaScript)

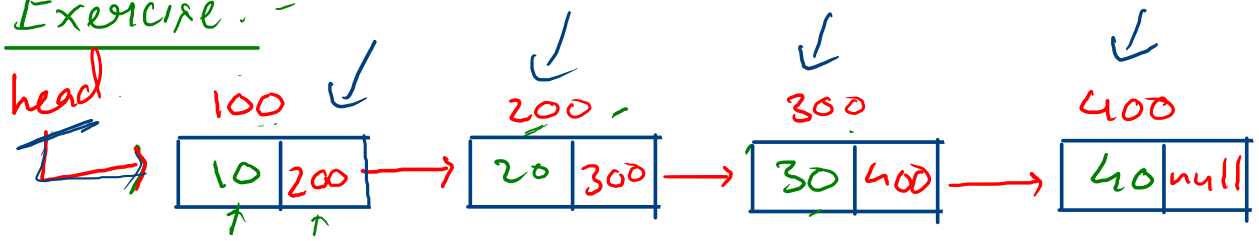
head = null

```
function insertNodeAtTail(head, data) {

    let node =new LinkedListNode(data)

    if(head===null){
        head =node;
    }else{
        let cur =head;
        while (cur.next!==null){
            cur =cur.next;
        }
        cur.next =node;
    }
    return head;
}
```

Code :- (in Java)

```
public static Node insertAtTail(Node head, int val){

   Node node = new Node(val);

   if(head == null){
       head = node;
   }else{
     Node cur = head;
      while (cur.next != null){
          cur = cur.next;
      }
      cur.next = node;
   }
   return head;
 }
```

Exercise :-



O/P

① print (head) → 10 100 30

② print (head.data) → 10

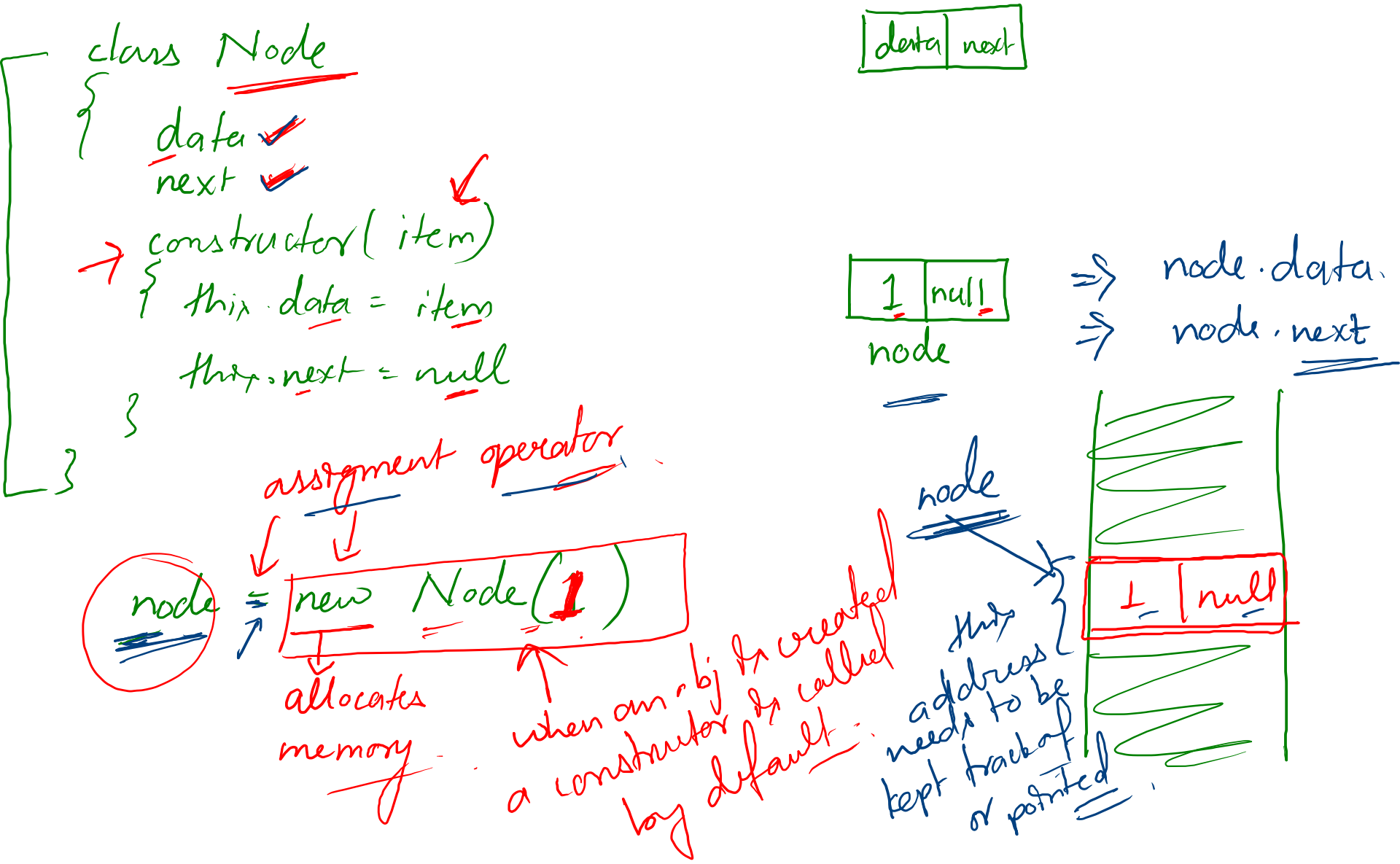
100.data → 10

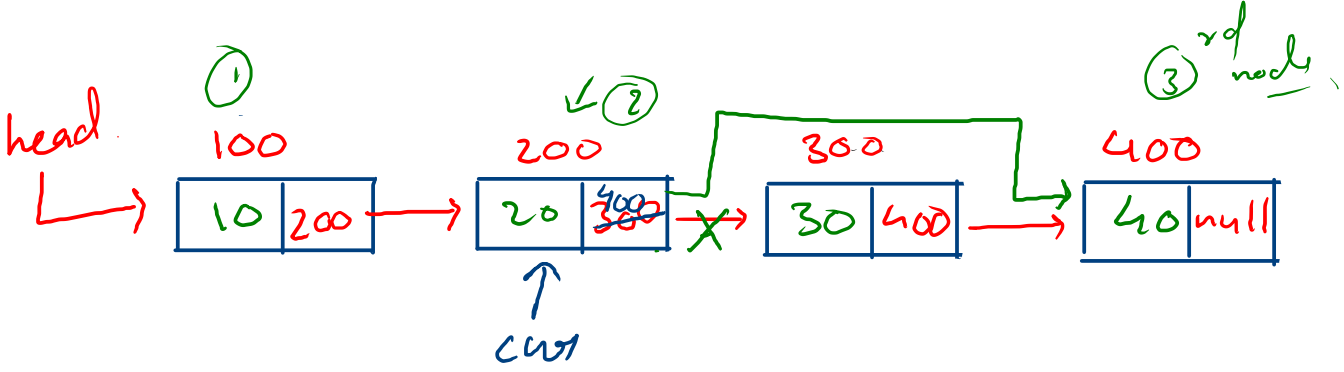
③ print (head.next.next.data) →

100.next → 200.next → 300.data → 30

① ~~20~~ ② 30 ③ ~~40~~

④ ~~others~~


class Node
{
data
next
constructor(item)
{ this.data = item
this.next = null
}
}
data next
1 null
node
=> node.data.
=> node.next
assignment operator
node = new Node(1)
allocates memory
when an obj is created a constructor is called by default.
node
1 null
this address needs to be kept track of or pointed



assignment operator

LHS RHS

(1) curr = head.next ⇒ curr = 200

100.next

↳ 200

(2) curr.next = head.next.next.next

200.next

↳ 300

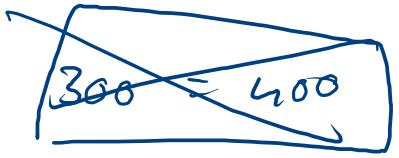
100.next

200.next

300.next

↳ 400

assigning operation



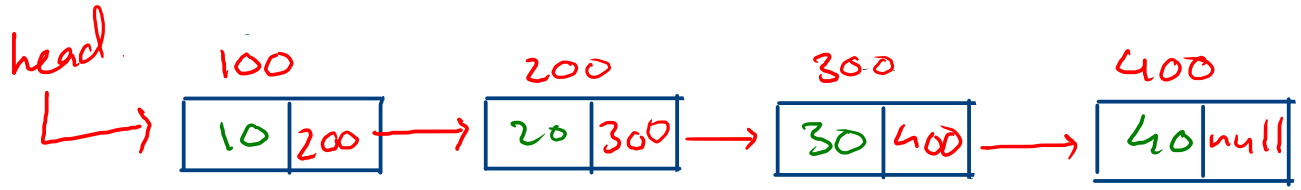
curr.next = 400

(3) print(curr.next.data) ⇒ 40

200.next

400.data

↳ 40



(1) cur = head.next ⇒

(2) cur.next = head.next.next.next

(3) print (cur.next.data) ⇒

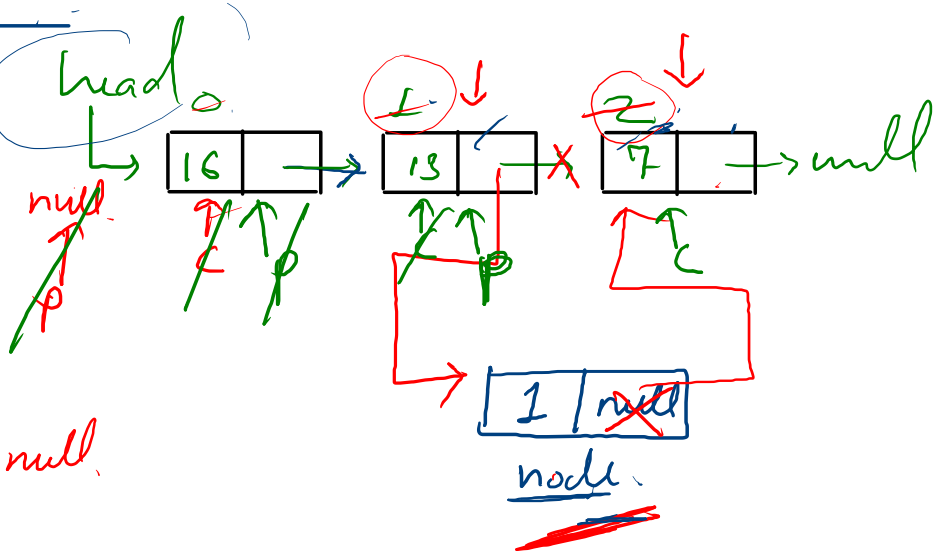
Break

back at 10:40 am

(2) Insert at a Specific Position :-

data = 1, position = 2

Step 1 → create a new node.



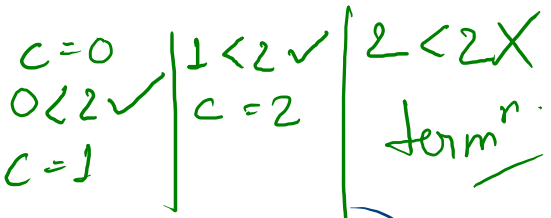
Step 2 → traverse & reach the given position.

count = 0, cur = head, prev = null

while (count < position && cur != null)

{
- prev = cur
- cur = cur.next
- count ++

}

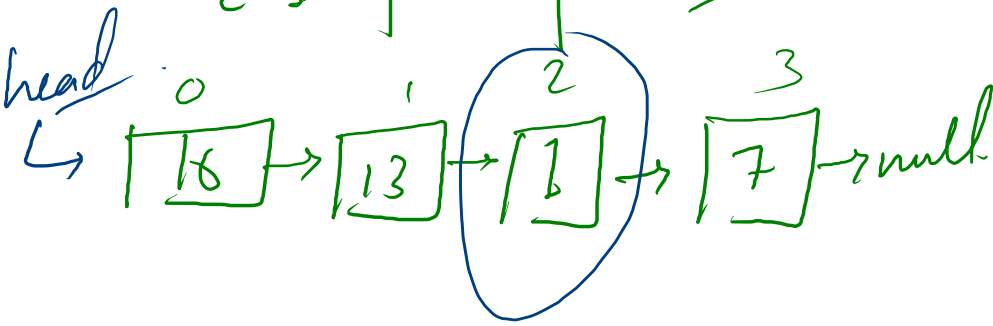


Step 3 → connect / disconnect

prev.next = node

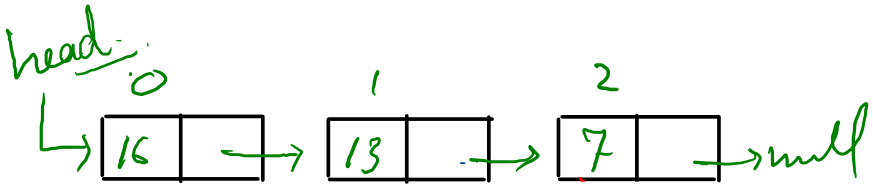
node.next = cur

return head.



Pseudo Code :-

```
function insertNodeAtPosition(head, data, position) {
        node = new LinkedListNode(data);
        if(position === 0 || head === null){
                node.next=head
                head = node;
                return head;
        }
    var count = 0;
    var prev = null;
    var cur = head;
        while ( count < position ){
                prev = cur;
                cur = cur.next;
                count++;
        }

    node.next = cur;
    prev.next = node;
    return head;
}
```



head = null

① head == null

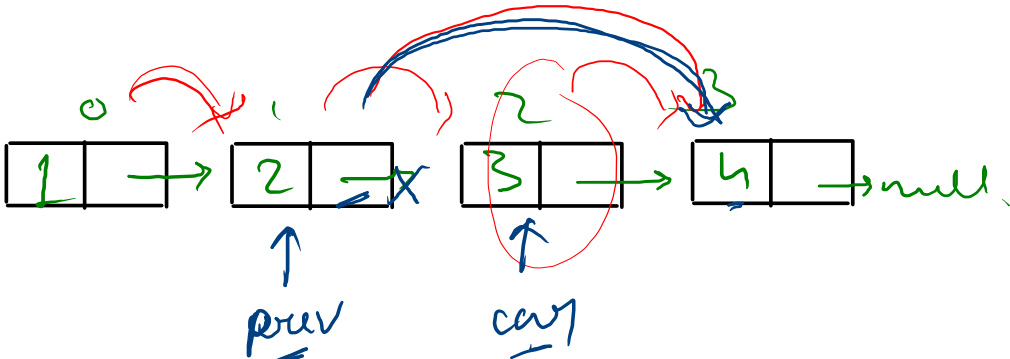
② head.next == null.

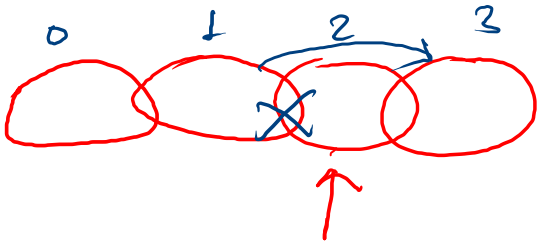
null.next

→ you are trying to access null, it's illegal.

(3) Deleting a node at a given position.

pos => 2





Step 1 → reach to the given position.

1 2 3 4

1 2 4

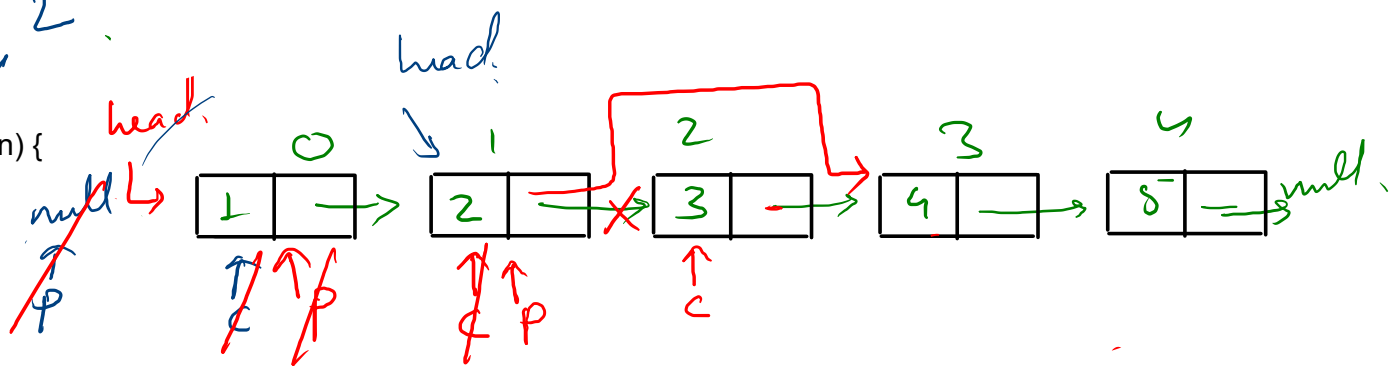
Step 2 →

prev.next = curr.next

return head

Pseudo Code :-

```
function deleteNode(head, position) {
    let i = 0;
    let cur = head;
    let prev = null;
    if(position == 0){
    head = head.next;
    return head;
    }

    while(i < position && cur != null){
        prev = cur;
        cur = cur.next;
        i++;
    }
    if(cur != null) {
        prev.next=cur.next;
    }

    return head;
}
```



i = 0 | 1 < 2 ✓ | 2 < 2 X
0 < 2 ✓ | i = 2 | term^n
i = 1

T.C → O(n)

S.C → O(1)

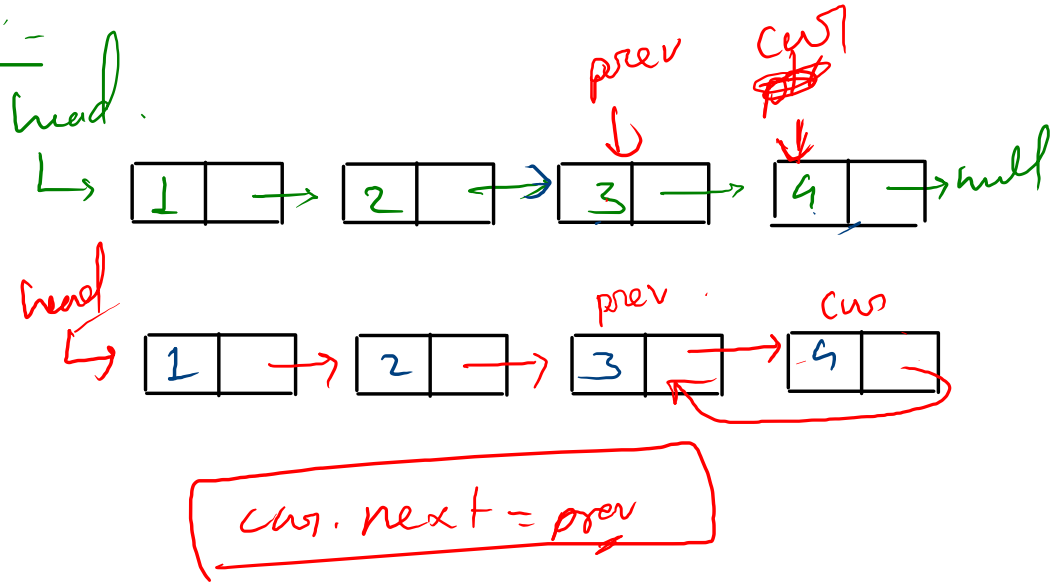
4 Reverse the Linked List :-

1 2 3 4

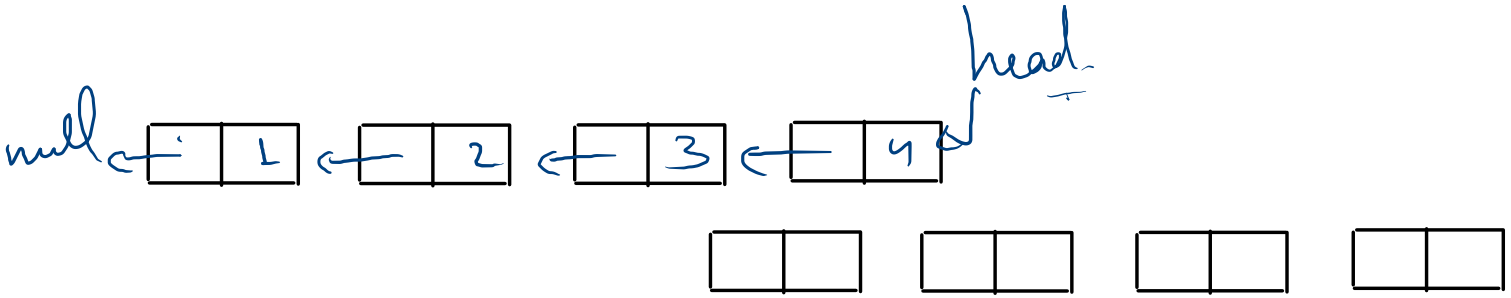
after reversal.

4 3 2 1

so, we need to reverse the links.



head.prev



4 Reverse the Linked List :-

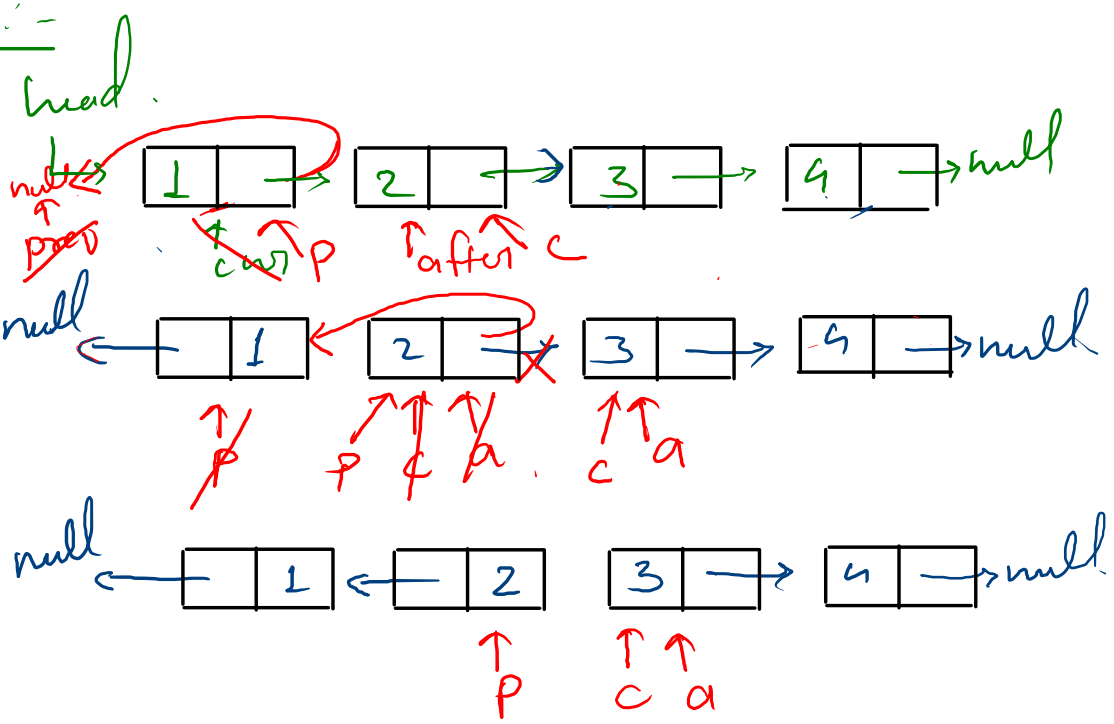
let's reverse the links one by one from the beginning → to reverse the links

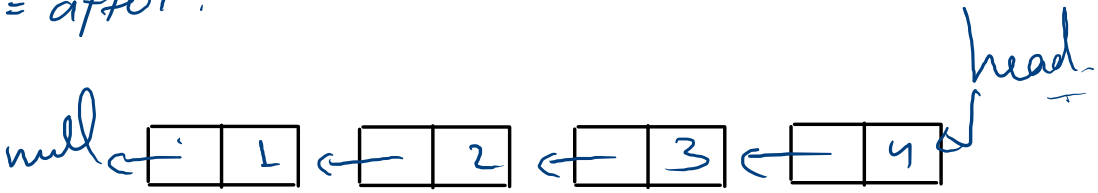
curr = head

to keep track of unreversed LL

prev = null ← to keep track of reversed list

```
while (curr != null)
{
    after = curr.next
    curr.next = prev
    prev = curr
    curr = after
}
```





Pseudo Code :-

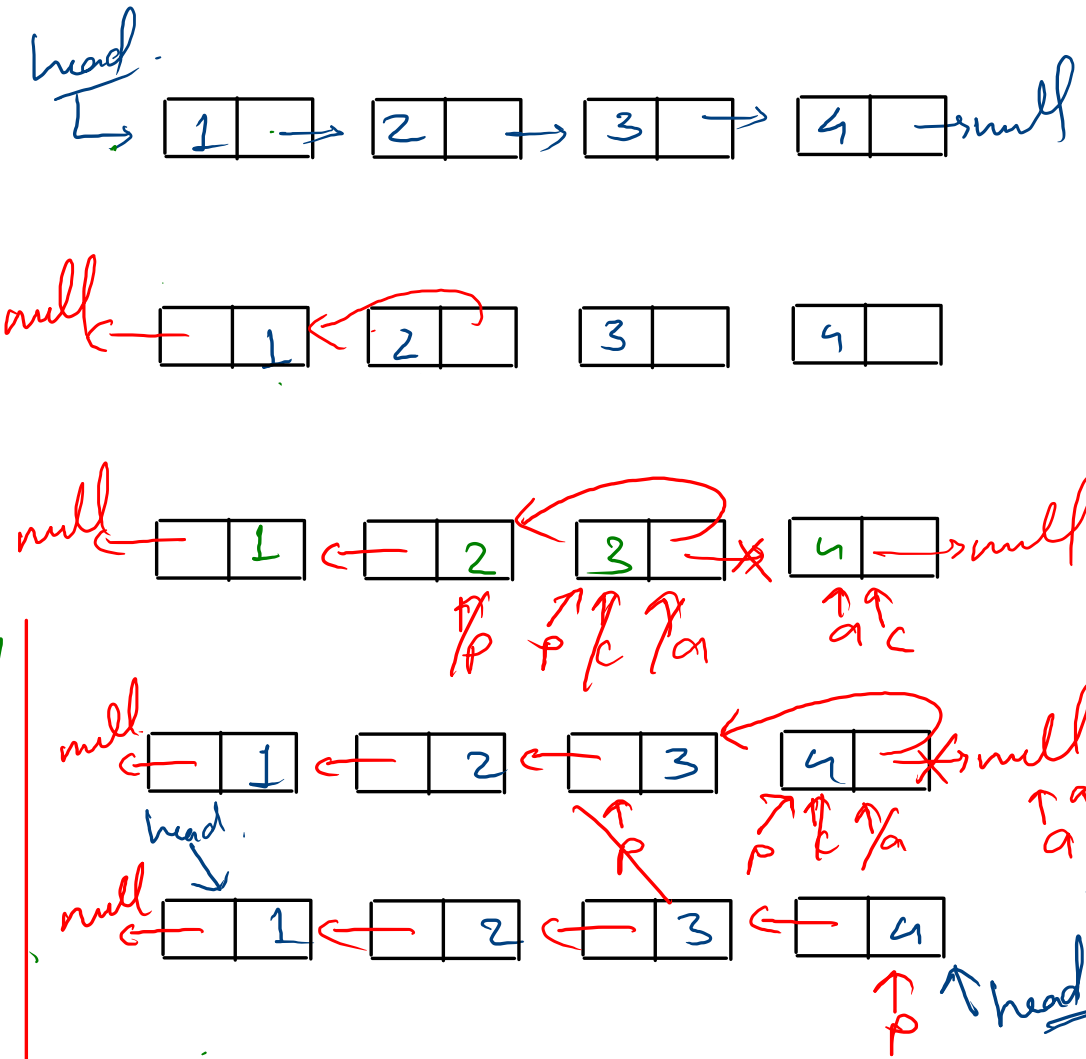
```
function reverse(head) {
  let cur = head;
  let prev = null;
  let after = null;

  while(cur !== null) {
    after = cur.next;
    cur.next = prev;
    prev = cur;
    cur = after;
  }

  head = prev;
  return head;
}
```

$n \not\leq 10^9$

$3 \not\leq 10^9$

T.C → O(n)

S.C → O(1)

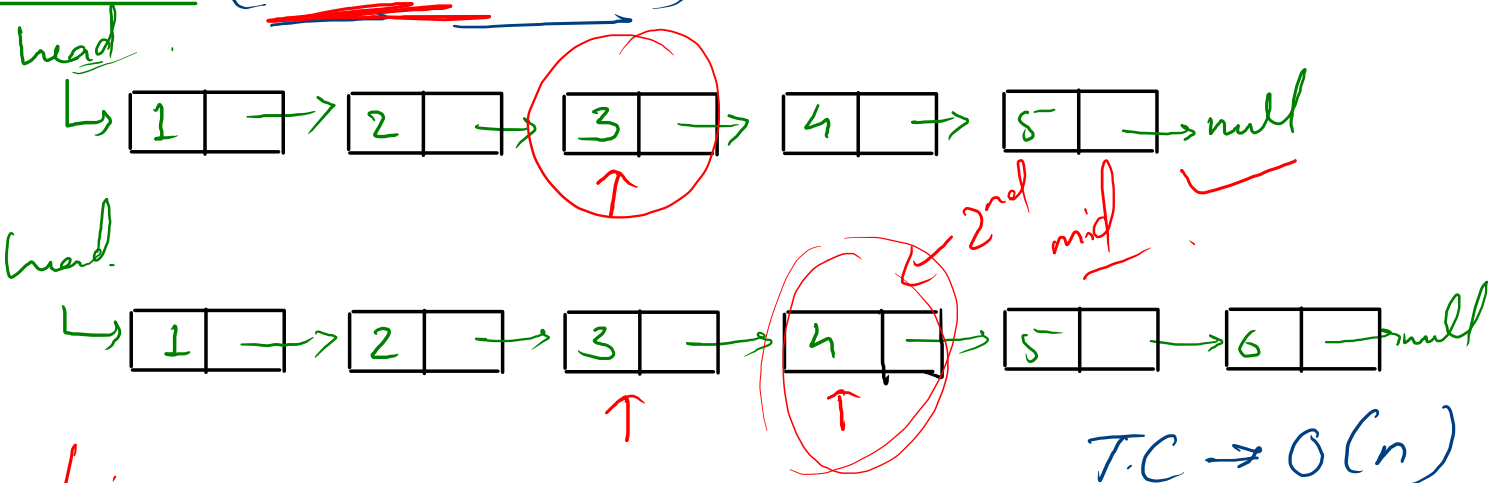


after → using to keep track of rest of the LL

cur → to operate or reverse the link of node

prev → pointing to the reversed list

(5) Middle Node :- (2nd Mid Position)



T.C → O(n)
S.C → O(1)

1st Approach :-

(1) find the length of the LL → O(n)

(2) mid = floor(length/2) + 1 → O(1)

(3) traverse to the mid position → O(n)

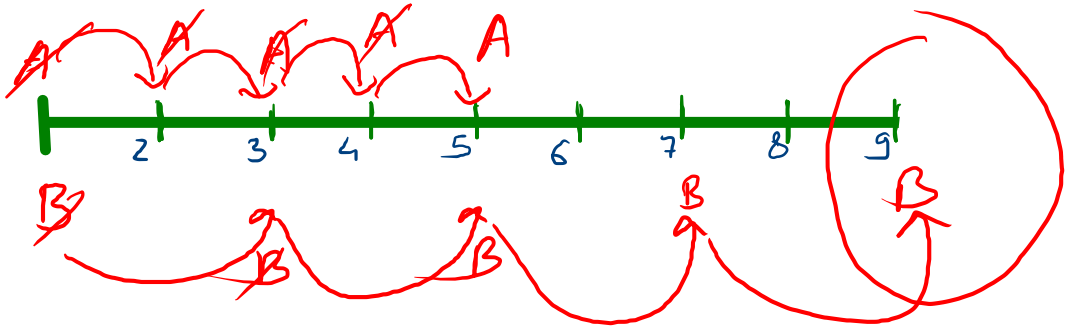
length = 5
mid = floor(5/2) + 1
= 2 + 1 = 3

length = 6
mid = floor(6/2) + 1
= 3 + 1 = 4

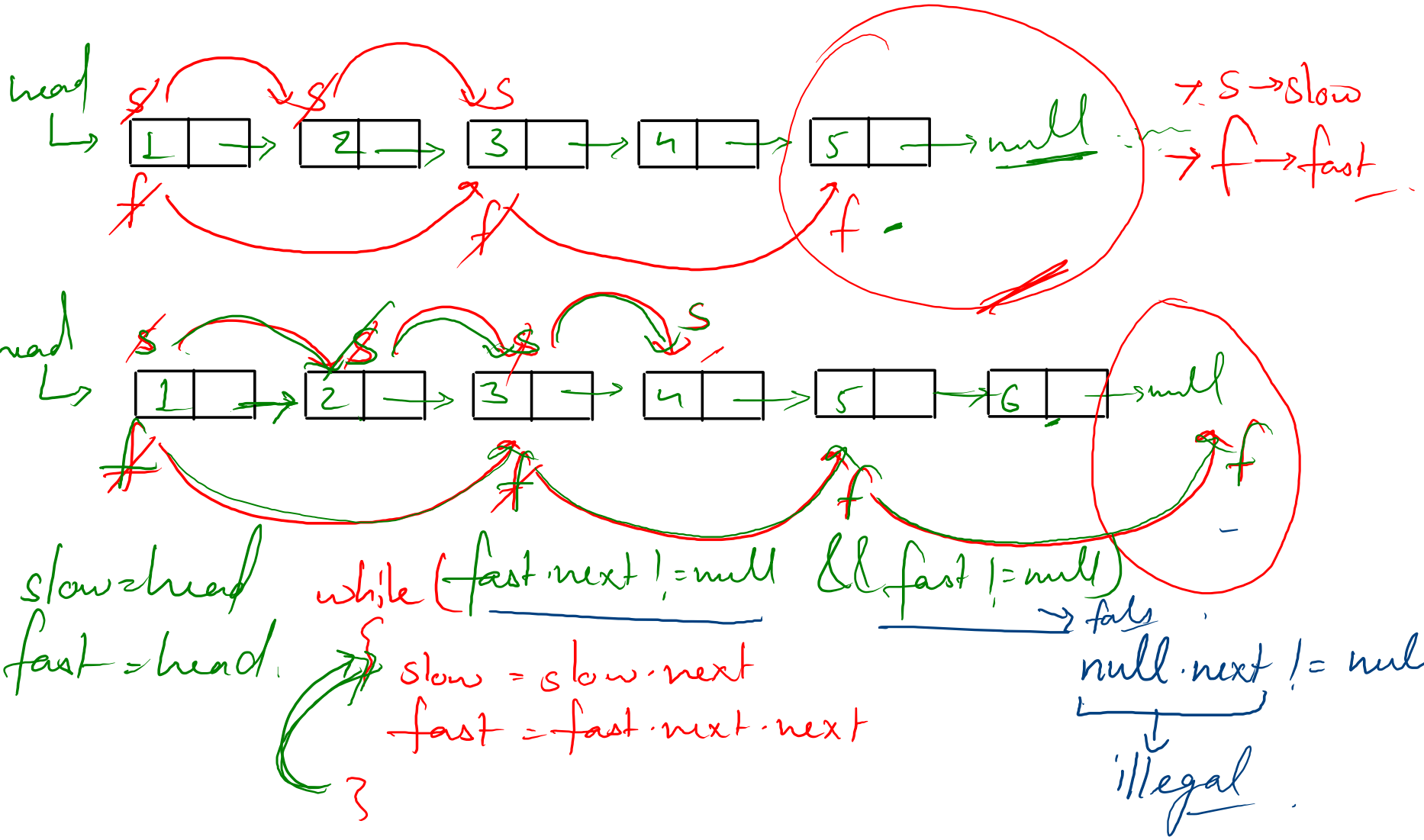
A → 1 step/sec.

B → 2 steps/sec.

B is twice as fast as A



when fast person reaches the end, the slow person is at the mid.


head
1
2
3
4
5
null
s
f
s → slow
f → fast
head
1
2
3
4
5
6
null
slow = head
fast = head
while (fast.next != null && fast != null)
{
slow = slow.next
fast = fast.next.next
}
false
null.next != null
illegal

```
while(fast.next != null && fast != null)
{
    slow = slow.next
    fast = fast.next.
}
```

A	B	A&&B
1	1	1
1	0	0
0	-1	0
0	0	0

```
while(fast != null && fast.next != null)
{
    slow = slow.next
    fast = fast.next.next
}
```

Short Circuiting

A	B	A && B
0	1	0
1	0	0
0	0	0
1	1	1

if ever A == 0 or false
↳ compiler won't check B

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

if ever A == 1 or true
compiler won't check for B

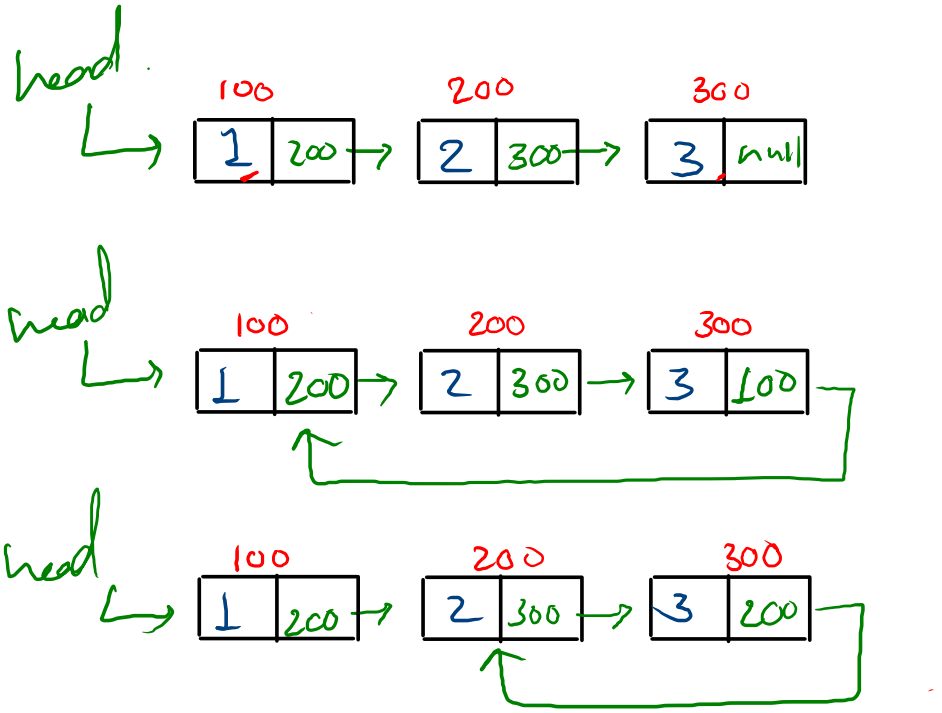
Pseudo Code (Using Slow & Fast Pointer Approach).

```
middleNode(head) {
    let slow = head;
    let fast = head;

    while(fast !== null && fast.next !== null) {
        slow = slow.next;
        fast = fast.next.next;

    }

    return slow.data;
}
```

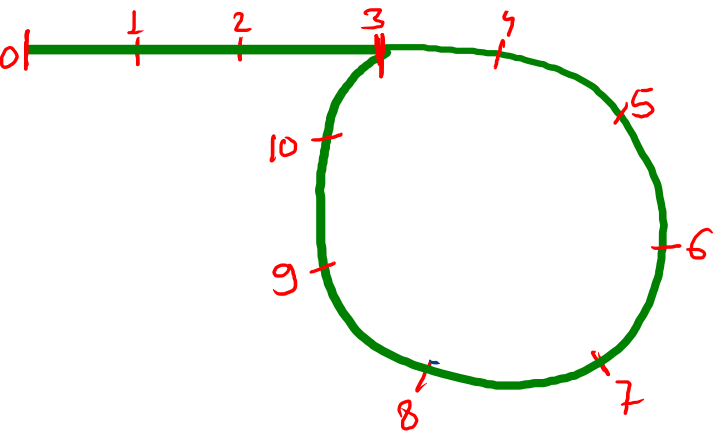
6 Linked List Cycle :-

while traversing if you reach some node which you have already visited

↳ ∴ Cycle is there



Slow & fast pointer appr.

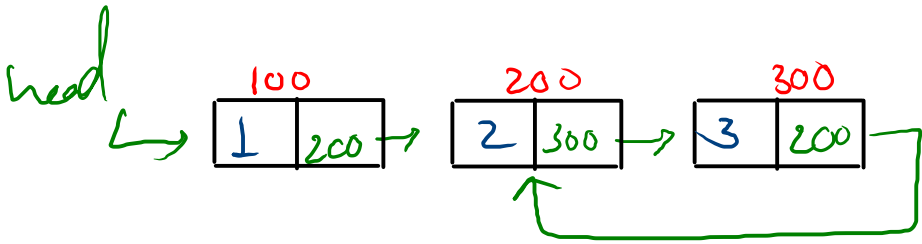


A → 1 step / sec

B → 2 steps / sec.

Pseudo Code (Using Slow & Fast Pointer Approach).

```
detectCycle(head) {
    let slow = head;
    let fast = head;

    while(fast !== null && fast.next !== null) {
        slow = slow.next;
        fast = fast.next.next;

        if(slow === fast) {
            return true
        }
    }

    return false;
}
```



(7) Delete a node without head :-

↳ only pointer to the node to be deleted is given.