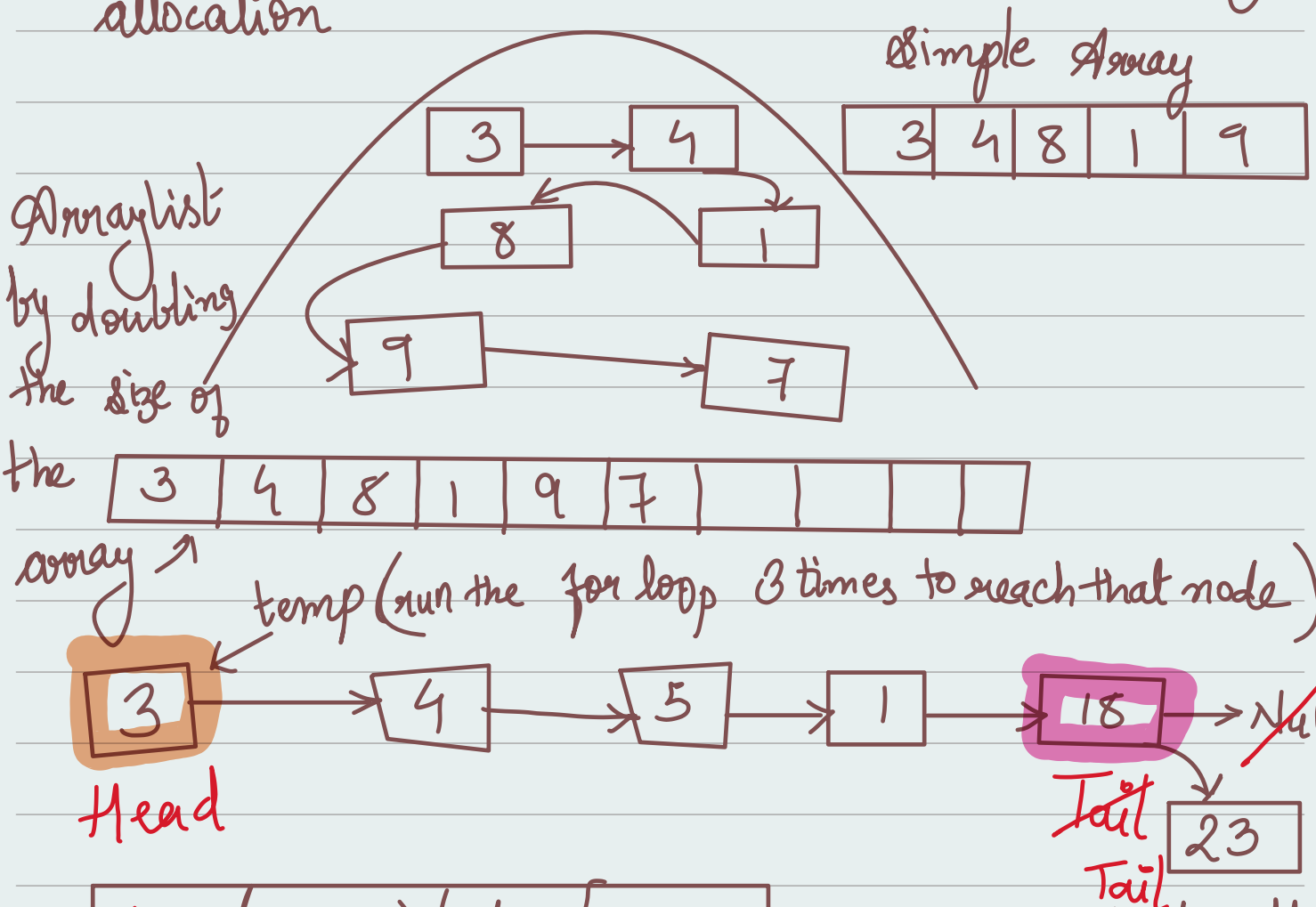
Linked List :—

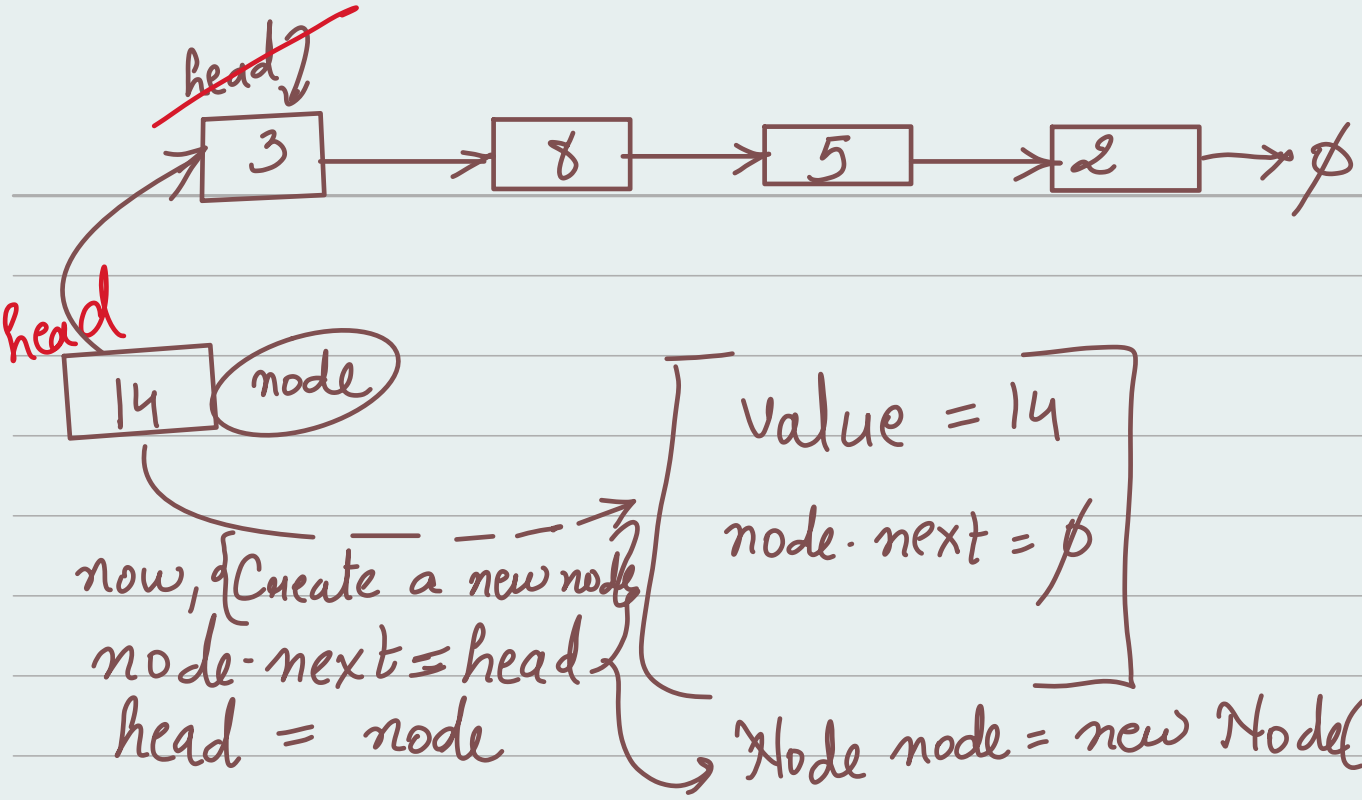
Linked list - It is not continuous memory allocation



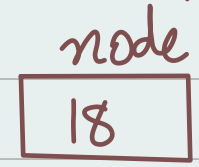

```
private class Node {
    int value;
    Node next;
}
```

→ We cannot directly access the index

→ Every nodes know who they are connecting to next node & what's their integer value.



if there is one item or Node;
Head and Tail is equal




```
if (tail == null)
    tail = head
size ++
```

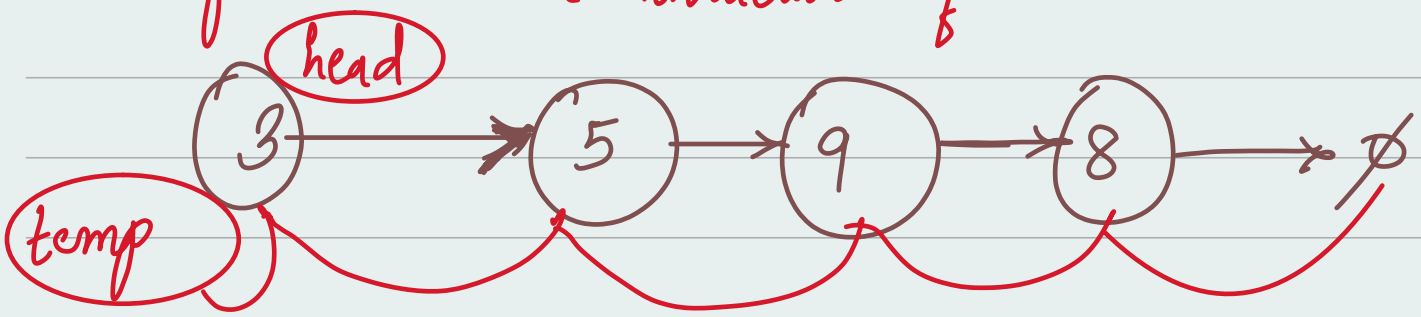
How display the Linked List.

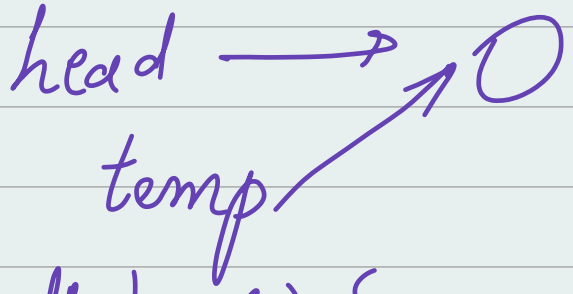
wrong

```
while (head != Null)
    print(head.val)
    head = head.next
```

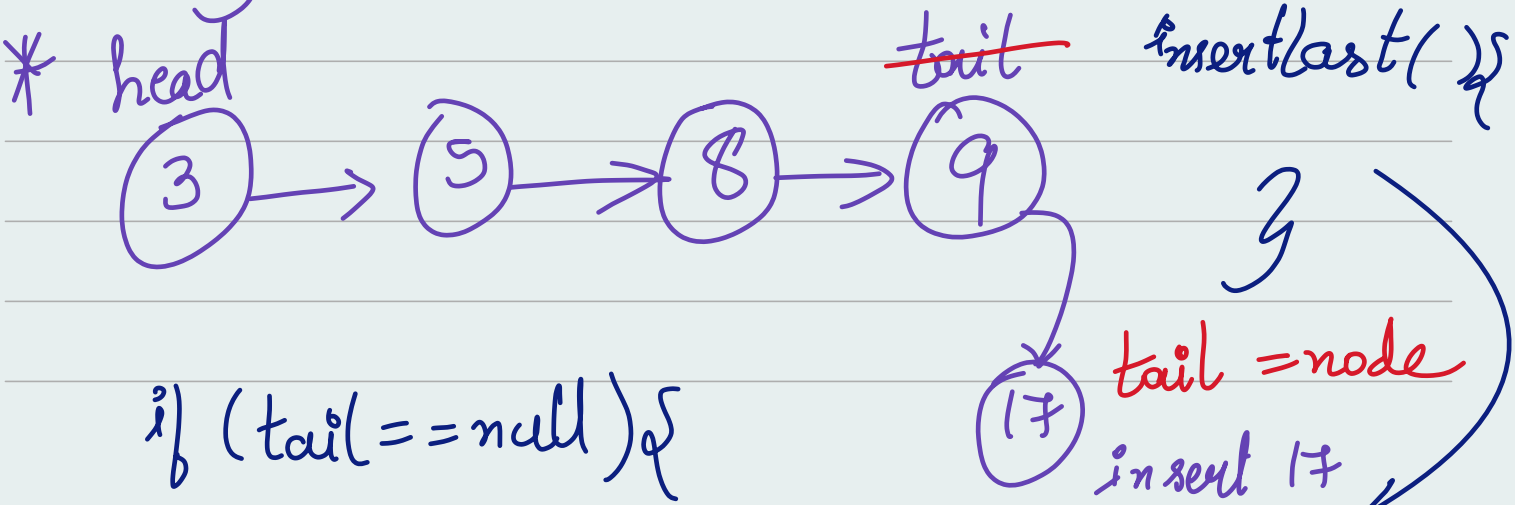
The above code is wrong cause it will print all the values uptill the last node

So, we will create a temp node.
& Temp is node a structure of the Linked List





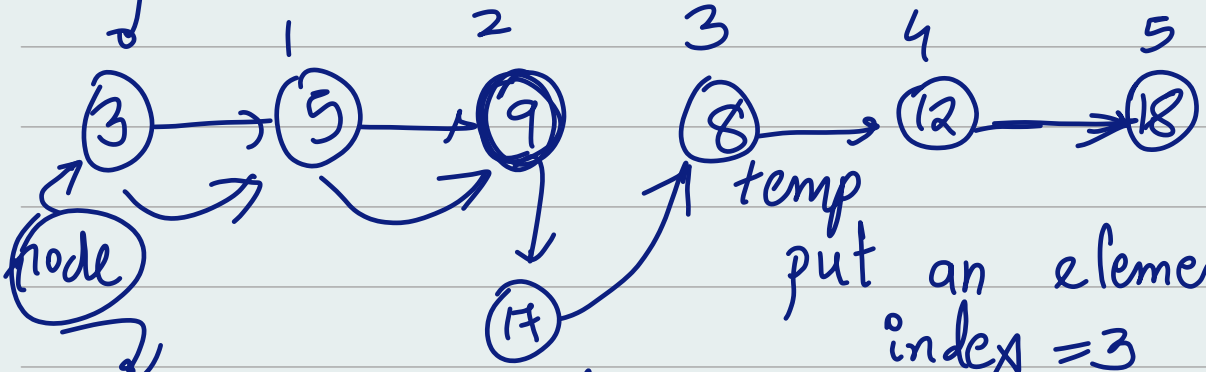

```
public void display (){
    Node temp = head; //temp itself is changing
    while (temp != null){
        System.out.println(temp.value+" -> ");
        temp = temp.next;
    }
    System.out("END");
}
```



insertFirst(val);
return;
}

tail.next = node
tail = node
size ++

* what if we want to insert at a particular index



put an element at index = 3

Node temp = head; // head taking 0th posn

for(i=1 ; i < index; i++){

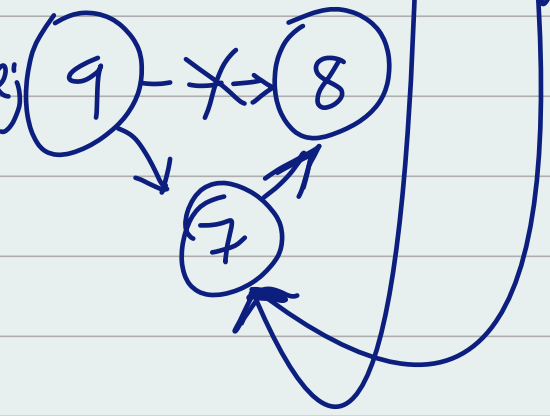
temp = temp.next; (9)

}

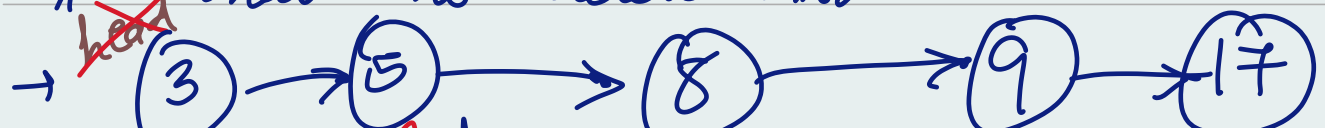
Node node = new Node(value, temp.next);

temp.next = node;

size ++;



* what is delete first ?



head

tail

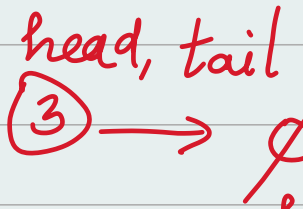
(head should move over 5)

This means delete first.

int value = head.value ;

head = head.next,

For one item




```
public int deleteFirst(){
    int value= head.value;
    head = head.next
    if (head == null){
        tail = null;
    }
    size--;
    return value;
}
```

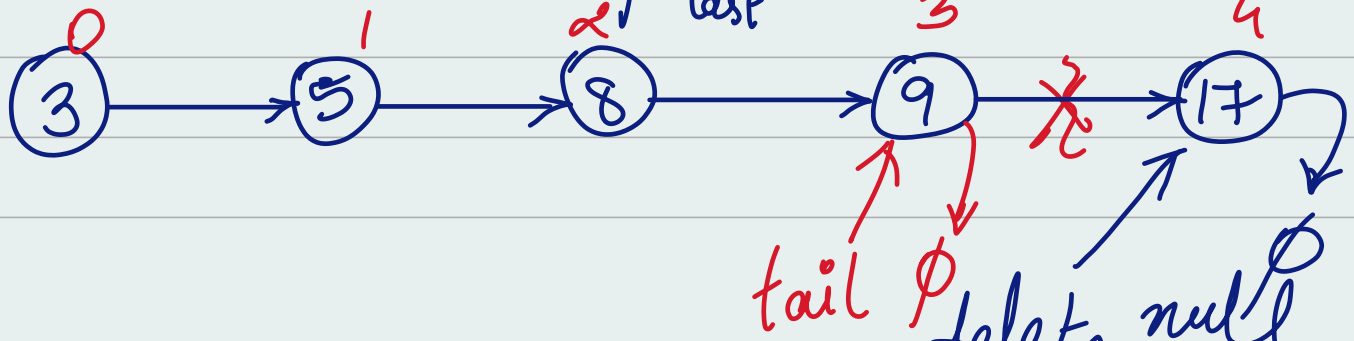
head = head.next(null)

means its empty

But tail is pointing to null make tail = null

tail = tail.next(null)

* Delete last and keep 2nd last element as last.



deleteLast

To get the value of 3rd item just loop the element 3 times

// create a get function to get the element and the reference of that node.

```
public Node get (int index){
    Node node = head;
    for( int i=0; i< index ; i++){
        node= node.next;
    }
    return node;
}
```

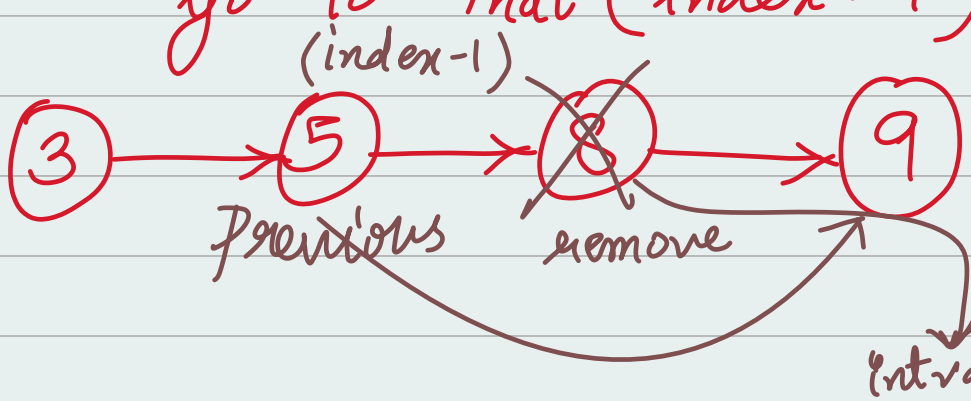
Now, delete the last element;

```
public int deleteLast( ){
    if (size<=1){ //this means only 1 node;
        return deleteFirst();
    }
    Node secondLast=get(size-2);
    int value= tail.value;
```

```
tail = secondLast;
tail.next = null; // make tail points to null
return value; // then return the last value.
}
```

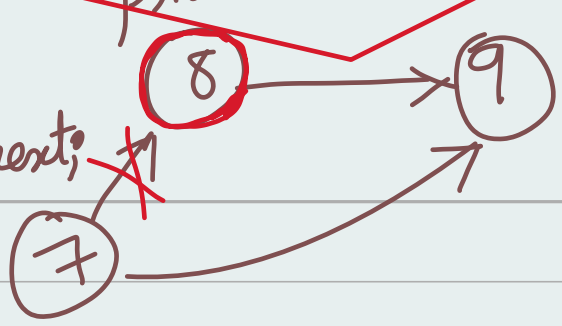
* Remove from a particular index,

Go to that (index - 1)




```
public int delete(int index) {
    if (index == 0) {
        return deleteFirst();
    }
    if (index == size - 1) {
        return deleteLast();
    }
    Node previous = get(index - 1);
    int value = previous.next.value;
```

previous.



previous.next =
previous.next.next;

return value;

}

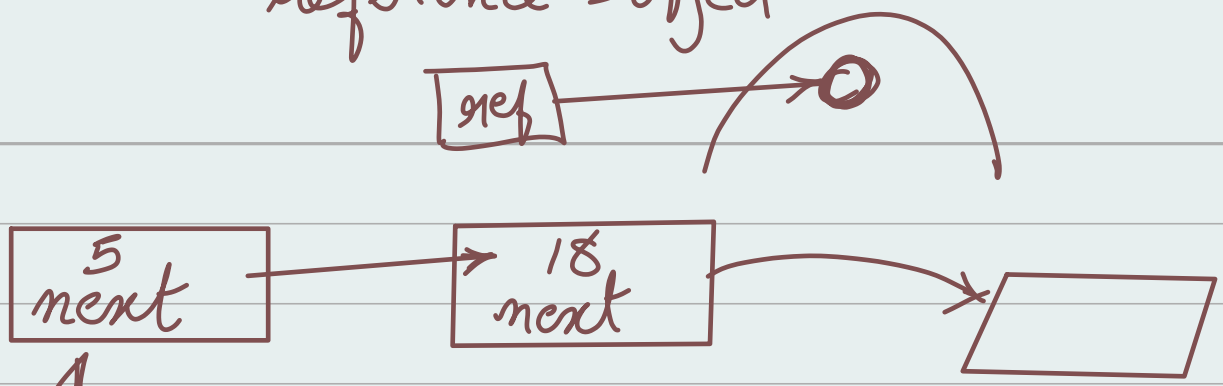
complexity O(N)

* finding a particular value & returning the node for that by using similar to get function

```
public Node find(int value){
    Node node = head;
    while (node != null){
        if (node.value = value){
            return node;
        }
        node = node.next;
    }
    return null;
}
```

basic understanding :-

reference = object



type node pointing to a type node

node = head

↓

scope will be in this function only and it will not change structure of linked list

node.something = something

↳ it's making actual change

node = node.next

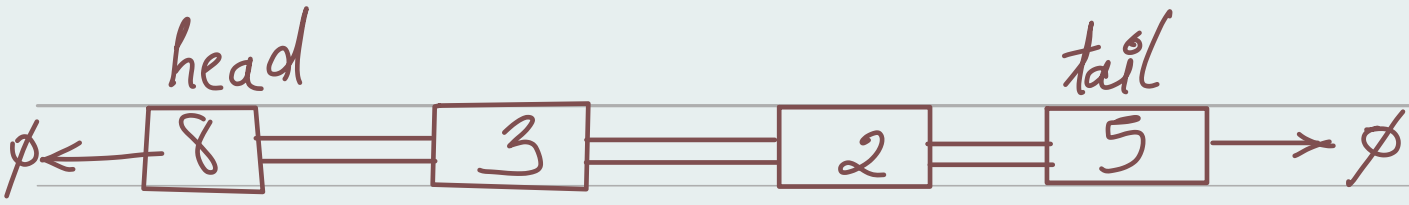
↓

this is not changing head pointer.

↓

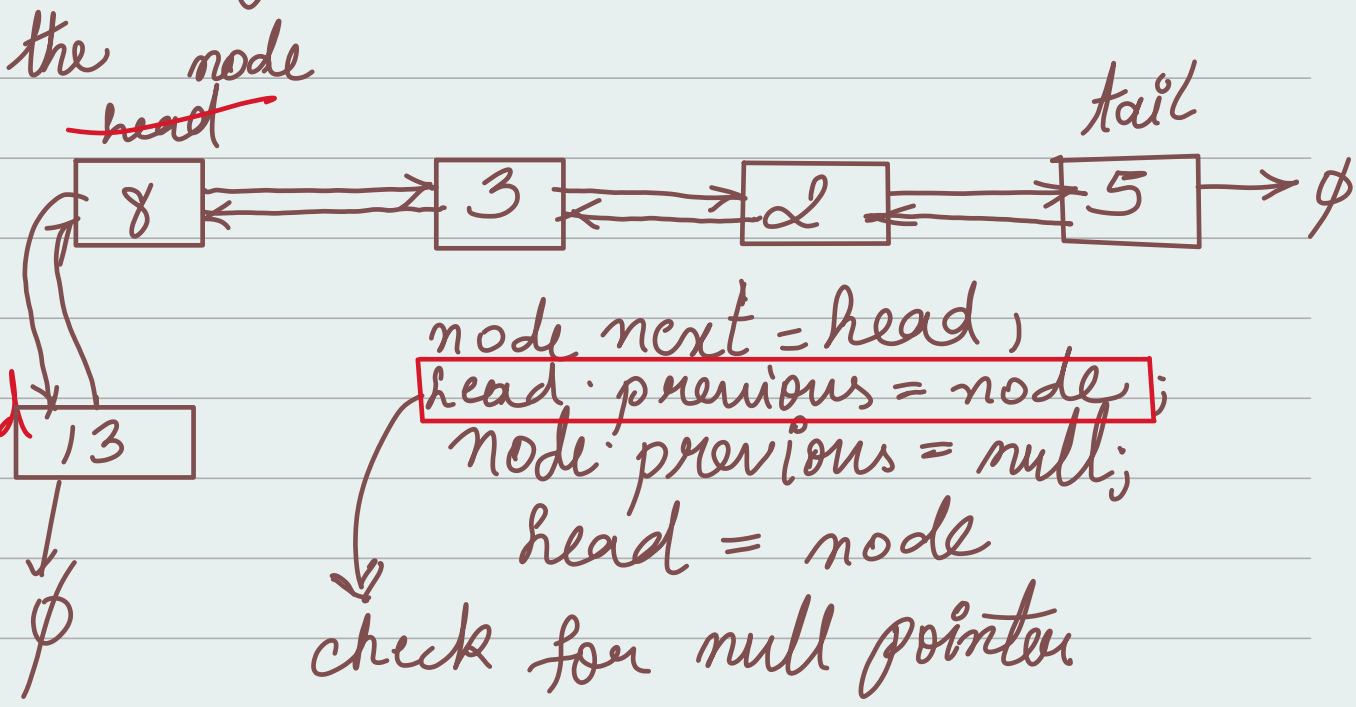
reassigning

Doubly Linked List

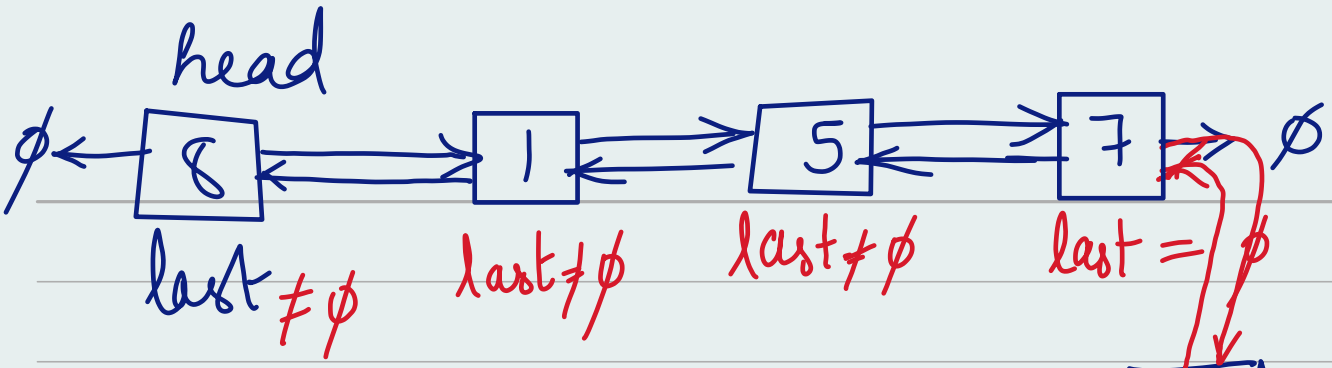


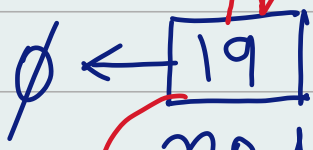

```
class NodeC
    Int val;
    Node next;
    Node previous;
}
```

* Inserting an element in the head of the node



* Add a new element in the linked list.





created a node

If tail is not provided then make a check if last is equal to null or not. If last is equal to null then it's tail.

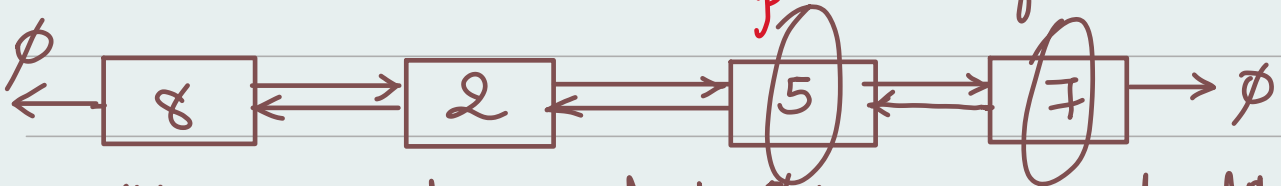
node is first node

```
node.next = null;
// find last.
last.next = node
node.previous = last;
```

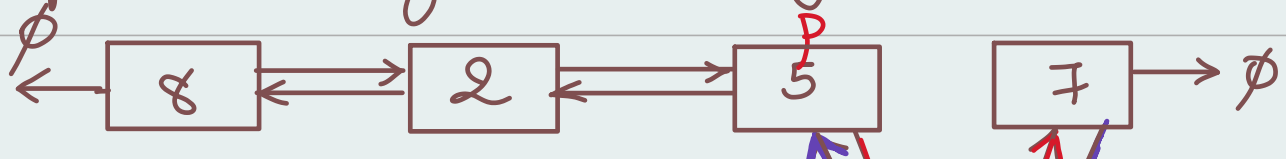
edges cases : When the list is empty

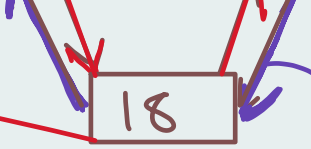
```
If(head == null)
node.previous = null;
head = node;
```

* Now insert an element 2 after 5



After inserting what it's gonna look like;

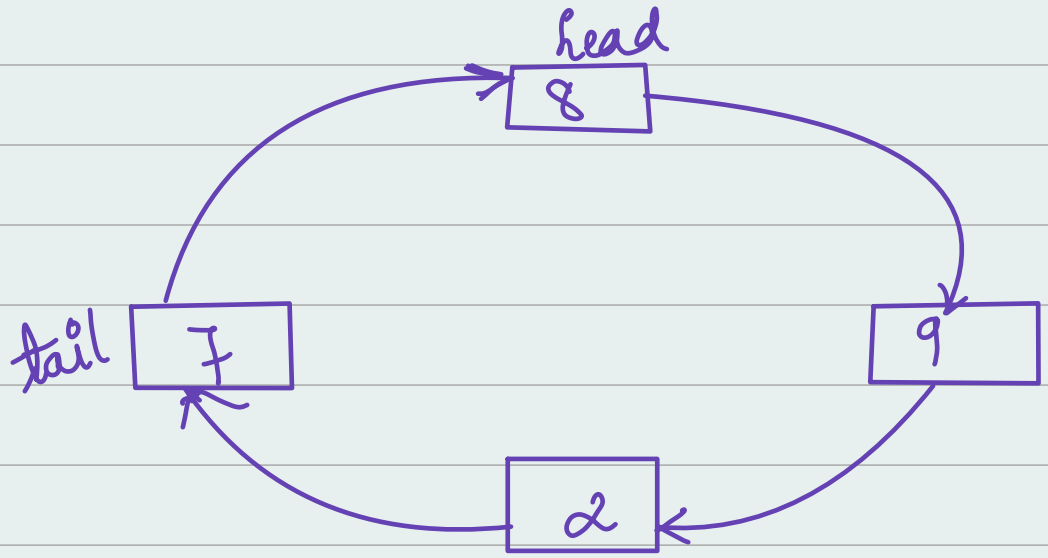





```
node.next = p.next;
p.next = node;
node.prev = p;
node.next.prev = node;
```

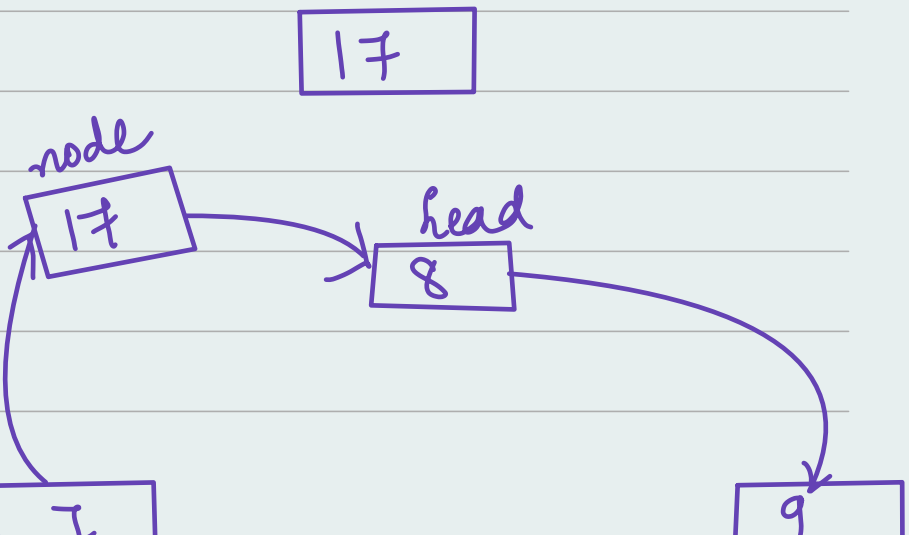
this may be null if it was inserted at last ie. after 7.

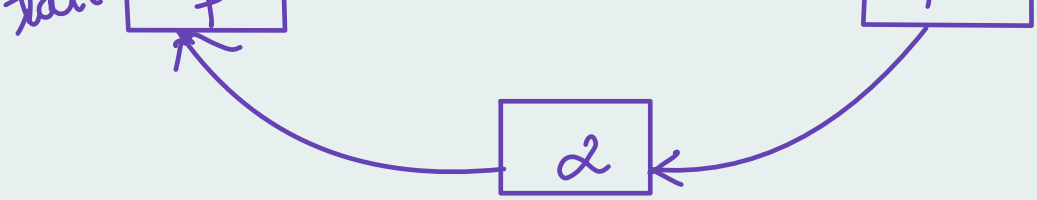
* Circular Linked List




```
class Node {
    int value;
    Node next;
}
```

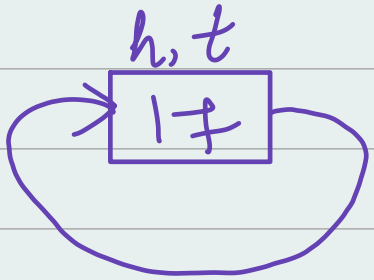
// create a new node



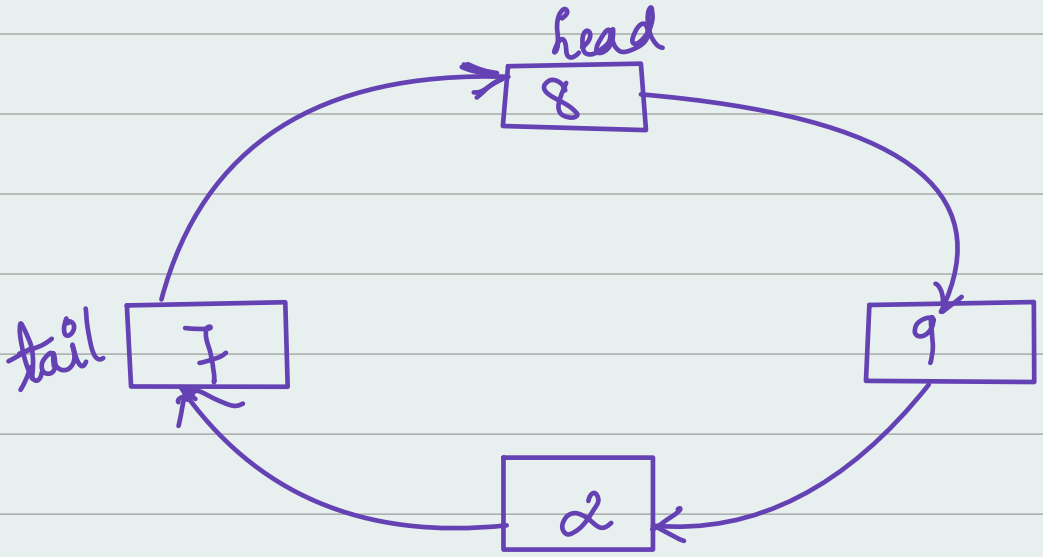



```
tail.next = node
node.next = head
tail = node
```

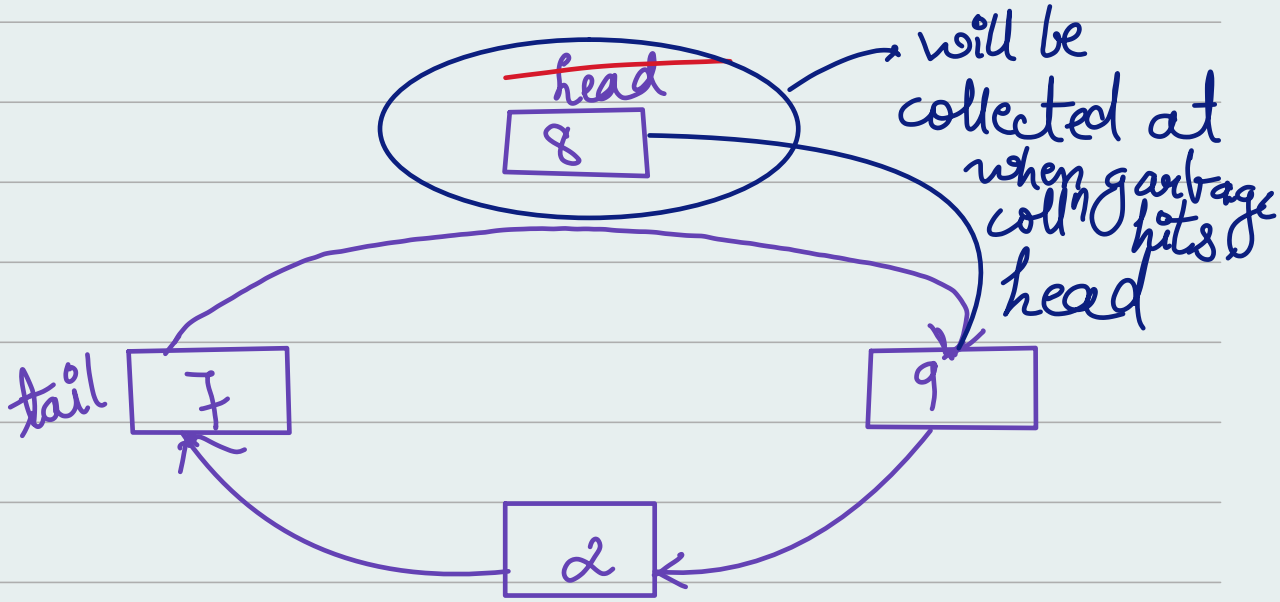
```
if  head == null
    head = node
    tail = node
```



* Deletion in a circular linked list

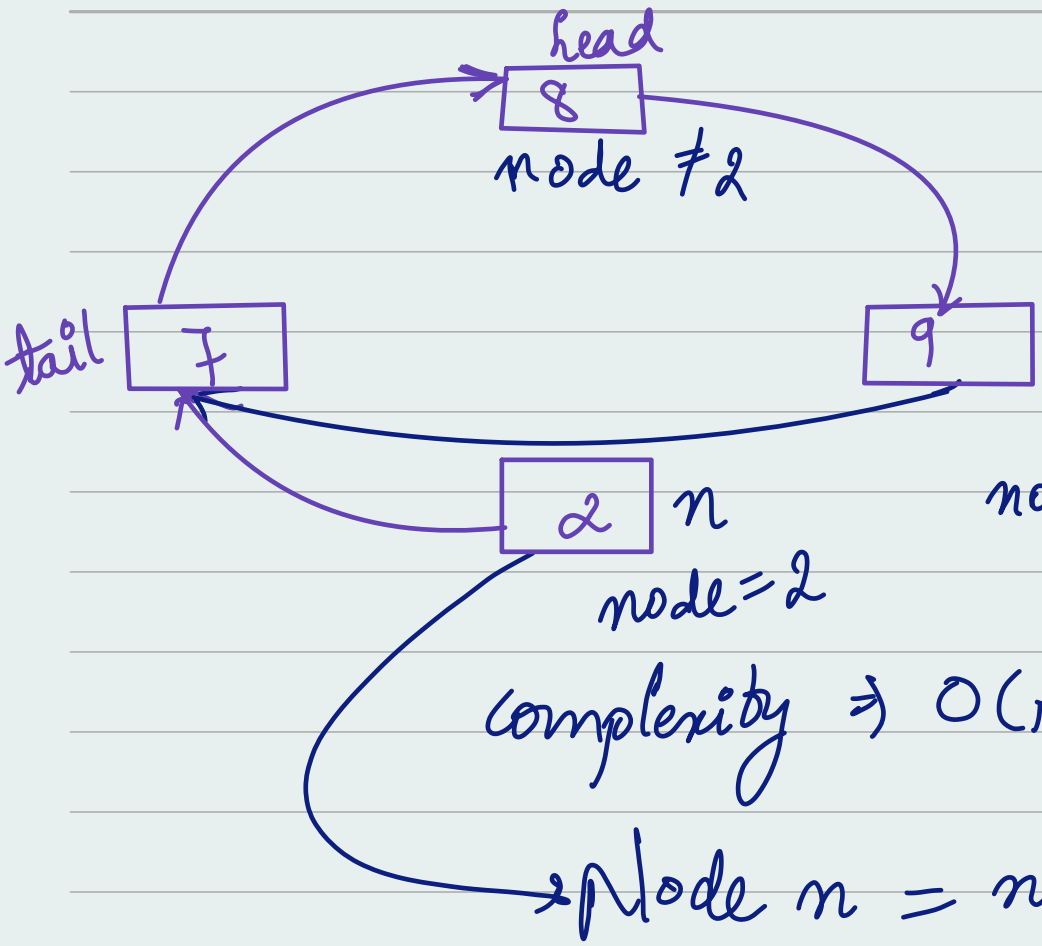


value=8
delete




```
head = head.next;
```

tail.next = head;



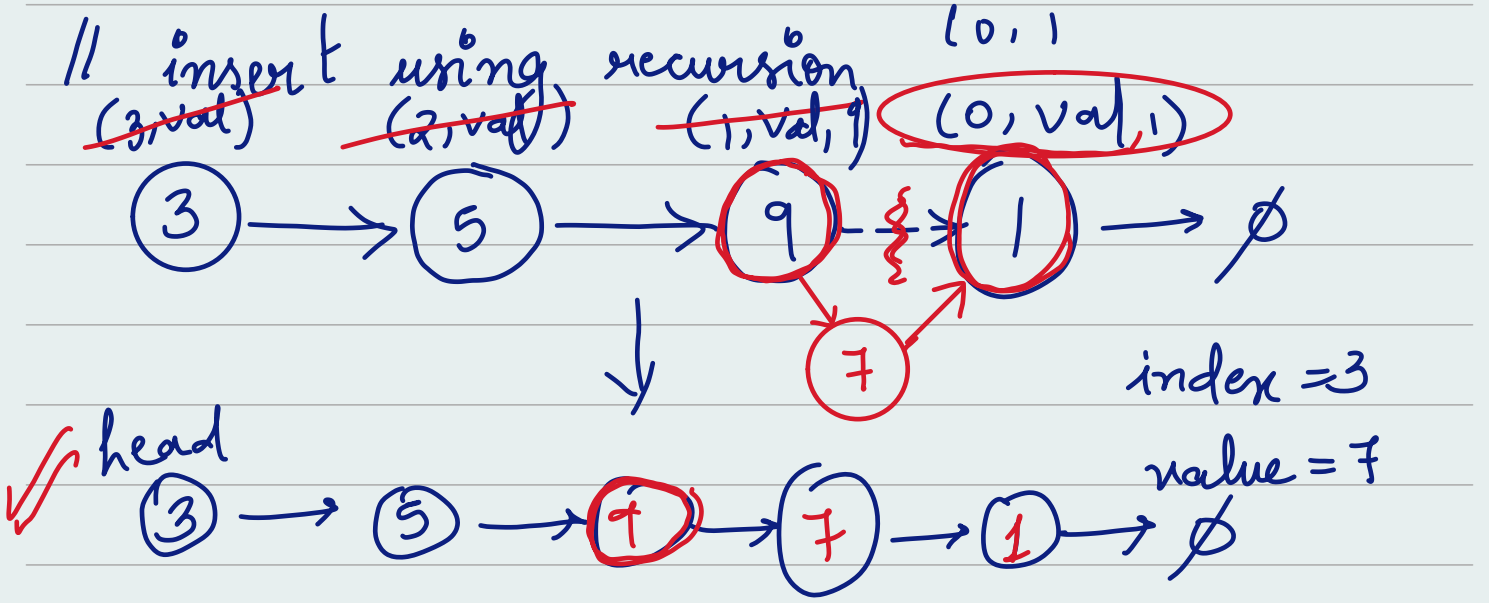
node next = n.next

complexity ⇒ O(N)

→ Node n = node.next;

Linked List Problems :-

// insert using recursion



① void return type and make changes in LL

② node return type that returns the list node to change the structure

(3, value)

Code:

```
public void InsertRec (int value, int index){
    head = InsertRec (value, index, head);
}
private Node InsertRec (int value, int index,
                        Node node){
    if (index == 0){
        Node temp = new Node(value, node);
        size ++;
```

return temp;

}

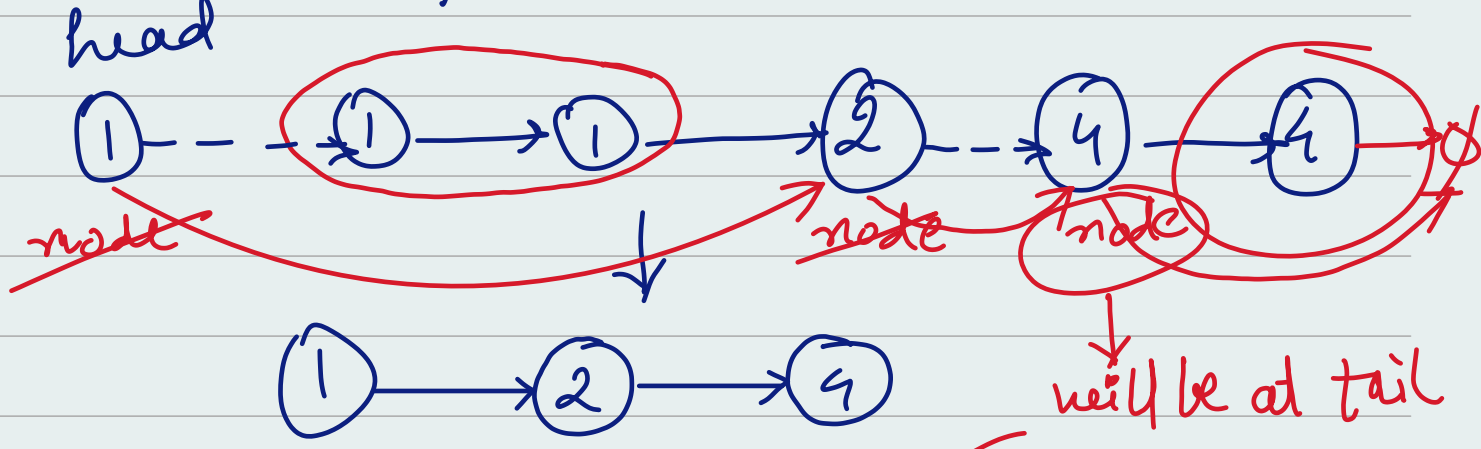
node.next = insertRec(value, index--,
current node next → 5.next = returned(9) node.next),

return node;

}

Time & Space complexity O(N).

8) Remove Duplicates from sorted LinkedList.



tail = node

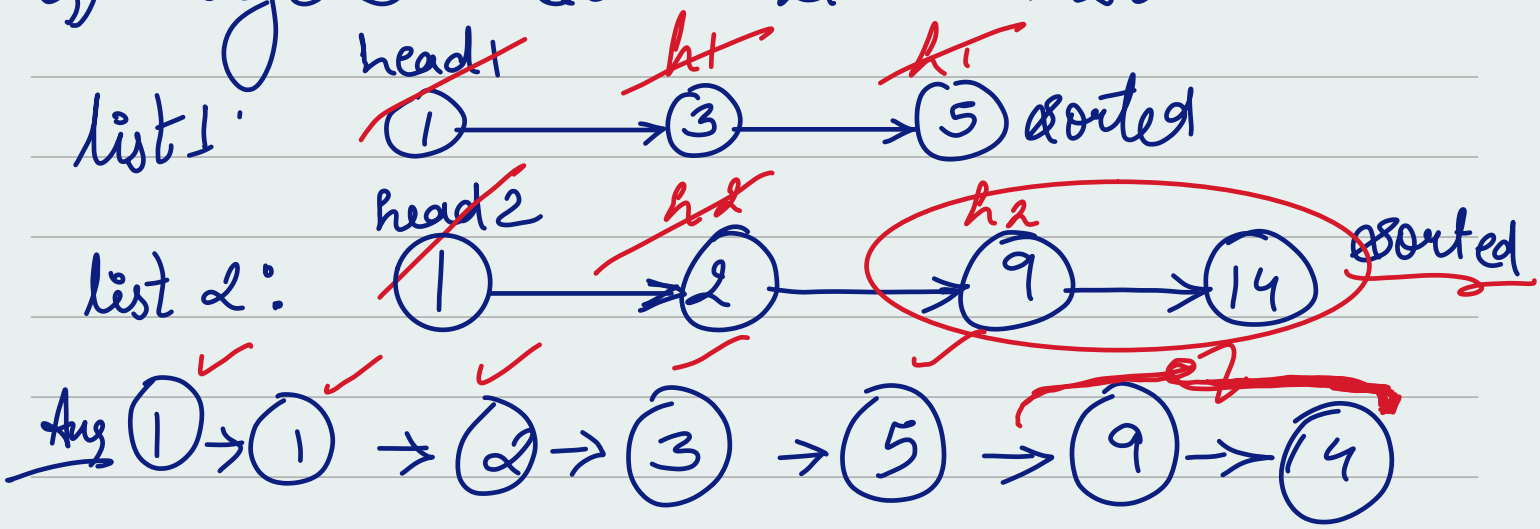
tail.next = ∅

Code:

```
public void duplicates(){
    Node node = head;
    while (node.next != null){
        if (nod value == node next.value){
```

```
        node.next = node.next.next;
        size--;
    }else{
        node = node.next;
    }
    tail = node;
    tail.next = null;
}
```

Q) Merge Two sorted Linked List



Code:

```
private static LL merge(LL first, LL second){
    Node f = first.head;
    Node s = second.head;
```

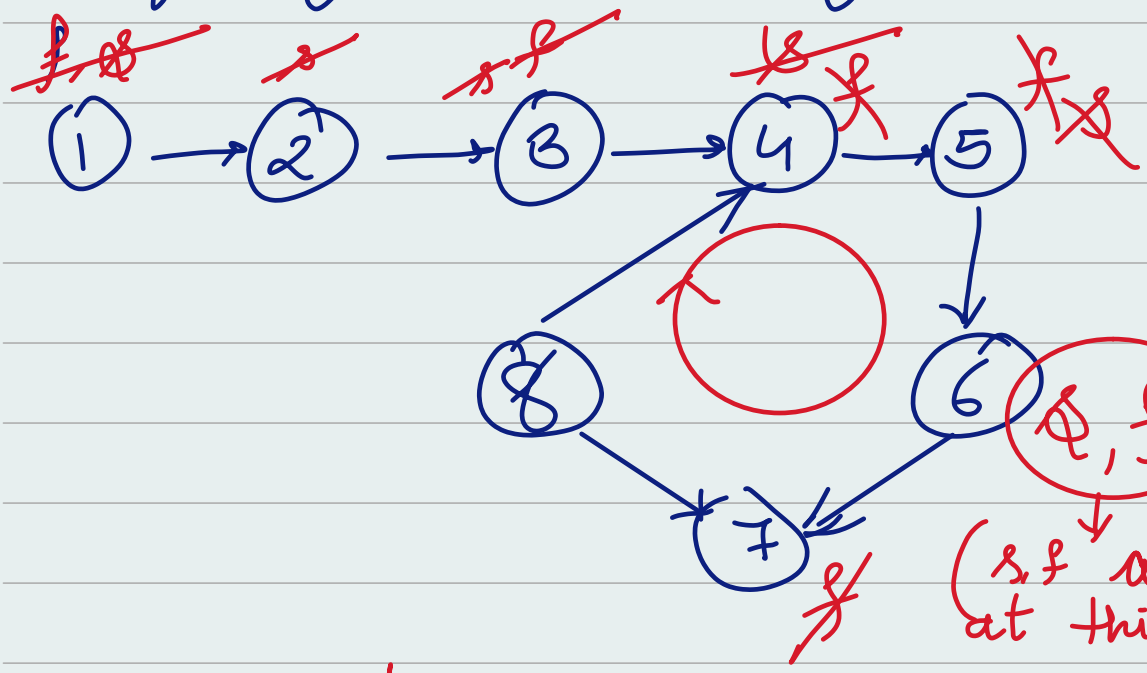
```
LL ans = new LL();
while (f != null && s != null){
    if (f.value < s.value){
        ans.insertLast(f.value);
        f = f.next;
    }
    else {
        ans.insertLast(s.value);
        s = s.next;
    }
}
while (f != null){ //When the first list is not null till then it will run
    ans.insertLast(f.value)
    f = f.next;
}
while(s != null){
    ans.insertLast(s.value),
    s = s.next;
}
return ans;
}
```

8) Linked List Cycle :-

* fast and slow pointer :-

→ cycle detection

→ finding a node in cycle, etc.

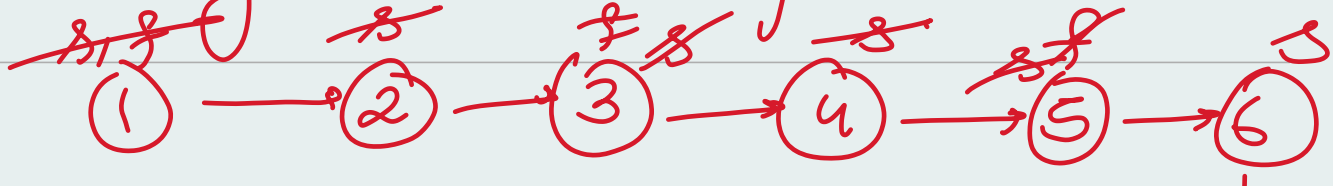


→ Slow pointer will move by one node.

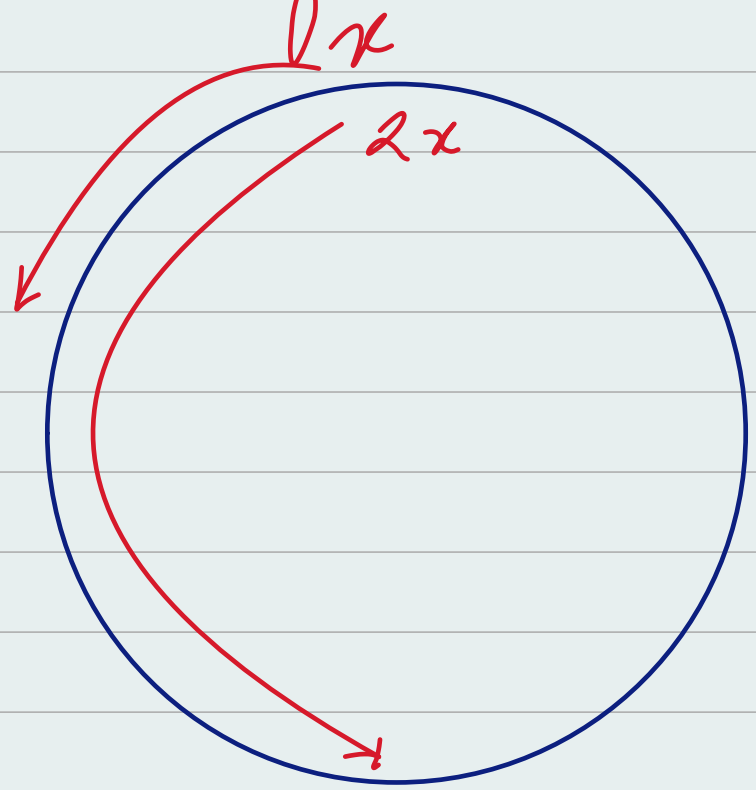
→ Fast pointer will move by two node.

→ If s, f are meeting at some point that means a cycle is present.
But if any these become null that means cycle is not present.

Ex :-

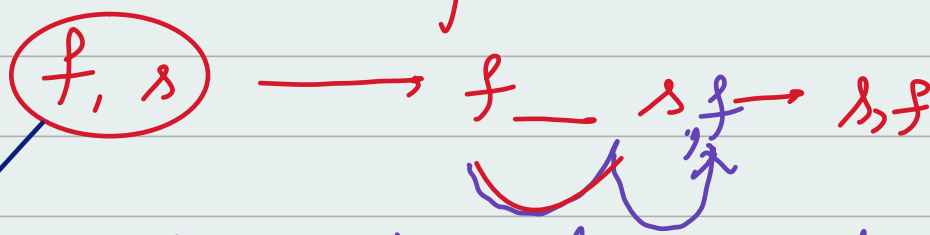


fast pointer is null that means cycle is not present.



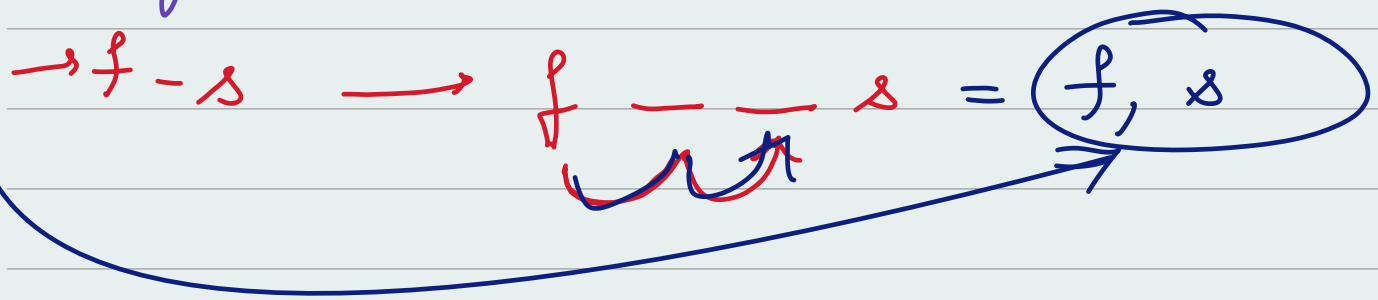
2 possibilities.

① fast pointer is one step behind slow pointer



In one step the fast pointer overtake the slow pointer and it meets

② (f - s) when fast pointer is two step behind slow pointer it will take two steps to meet each other

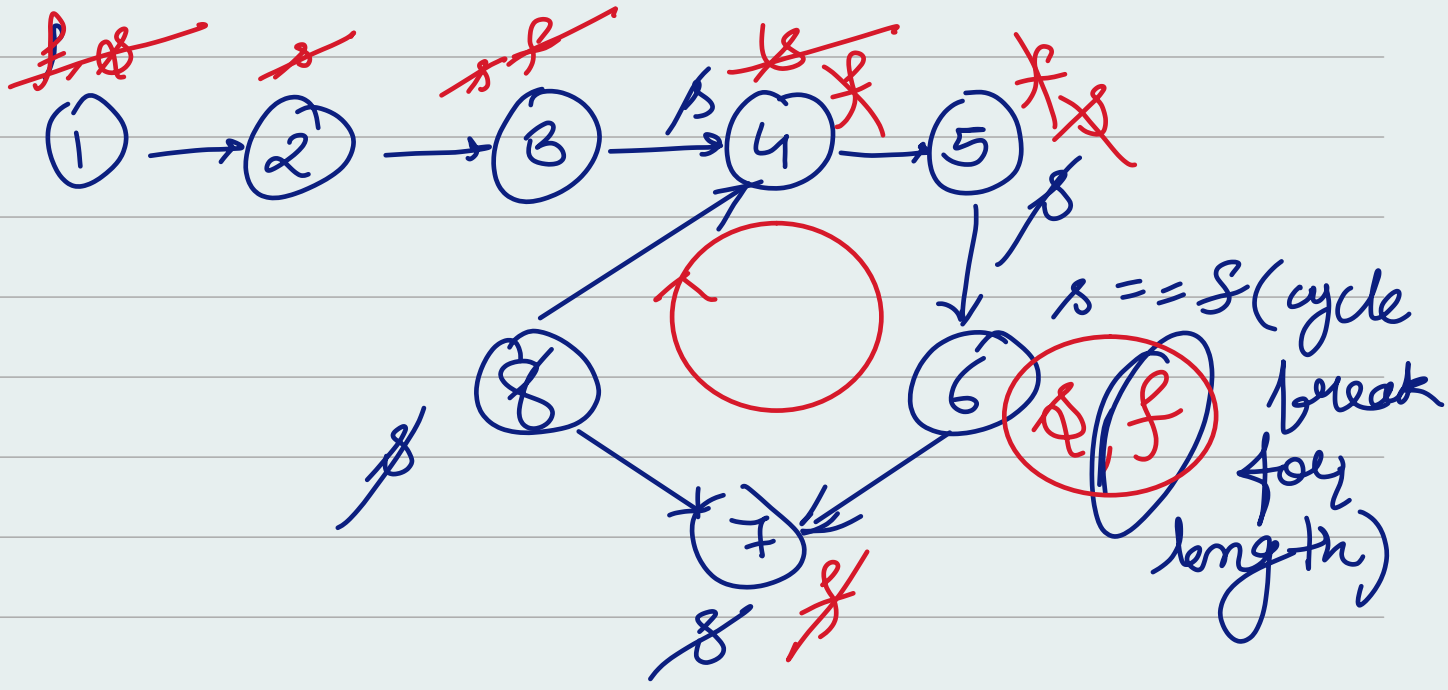


O(N) → Time complexity for circular and normal linked list (For fast & slow pointer).

Code :-

```
public class solution {
    public boolean hasCycle(ListNode
                            head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast != null && fast.next
                != null) {
            fast = fast.next.next;
            slow = slow.next;
```

```
if( fast == slow){
    return true;
}
}
return false;
}
}
```



Length of the cycle for that

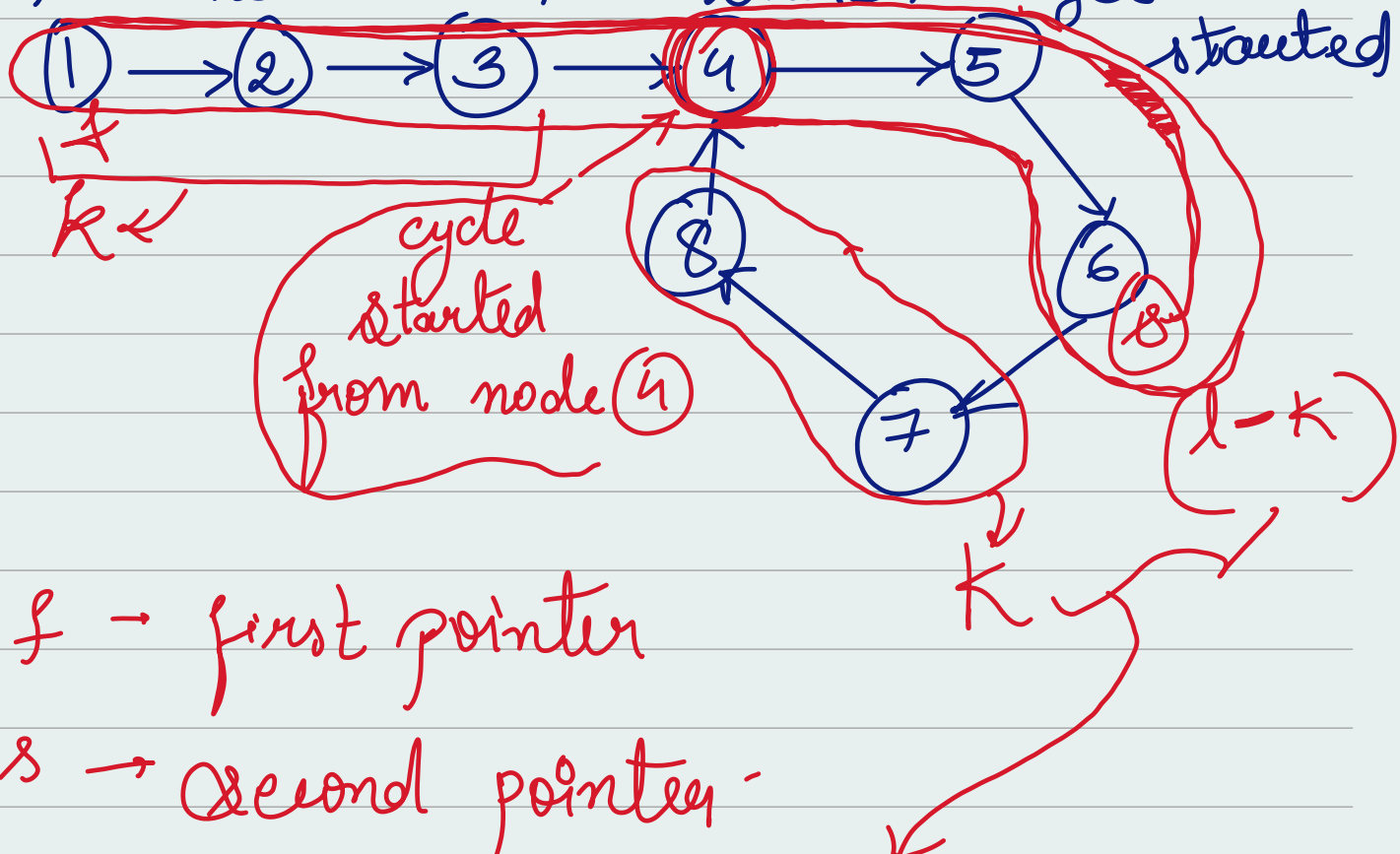
take a temporary variable and assign it to slow and while the slow is not equal to fast just increase the length of the cycle by 1. Use do while loop to start at least once

iteration.

```
ListNode temp = &low;
int length = 0;
```

```
length++;
```

Q) Find the node from where the cycle has started



f → first pointer

s → second pointer.

{Move s ahead by l of cyclic list} $l - k + k = l$ = length of the cll

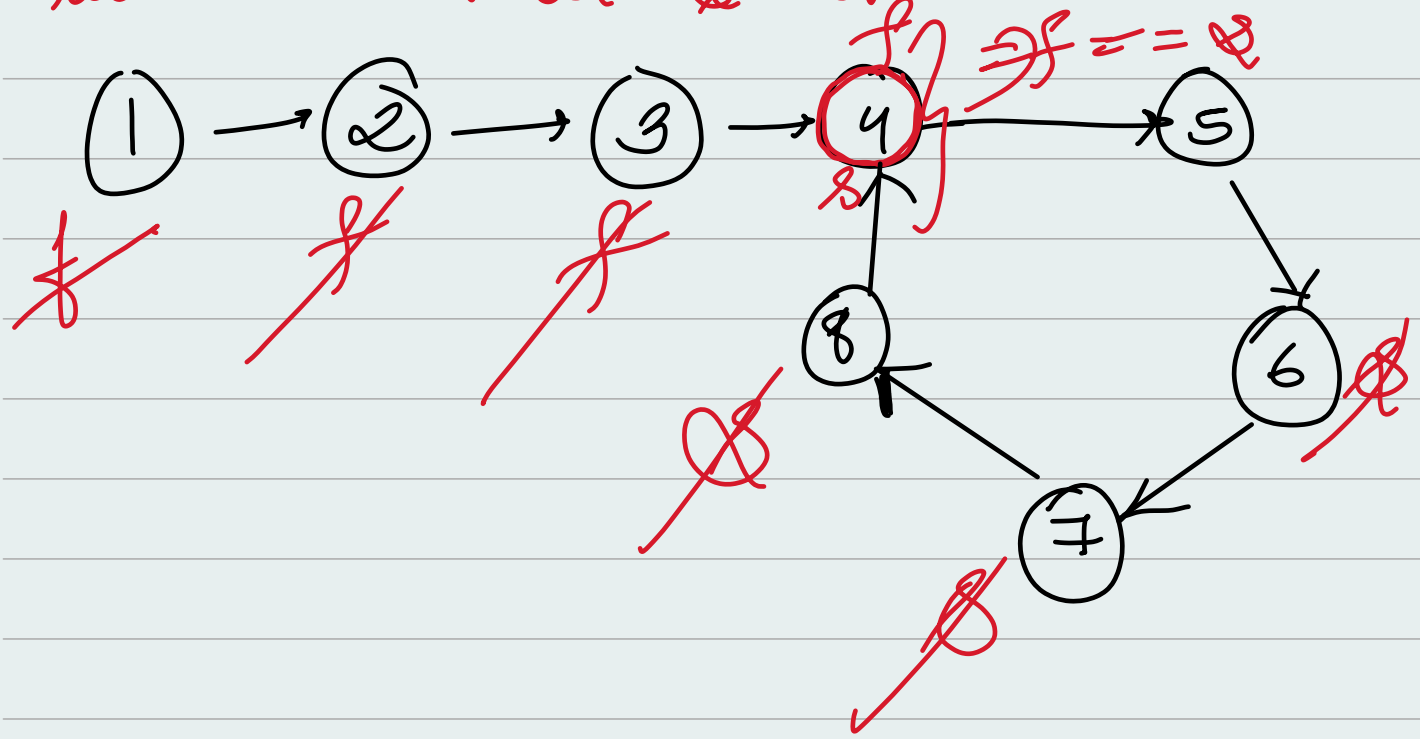
eg, $l - (l - k) = k$,

① find length of cycle.

② Move s ahead by length of cycle

times.

(3) Move s and f one by one it will meet at start.



Code:

1st step find length of the cycle.

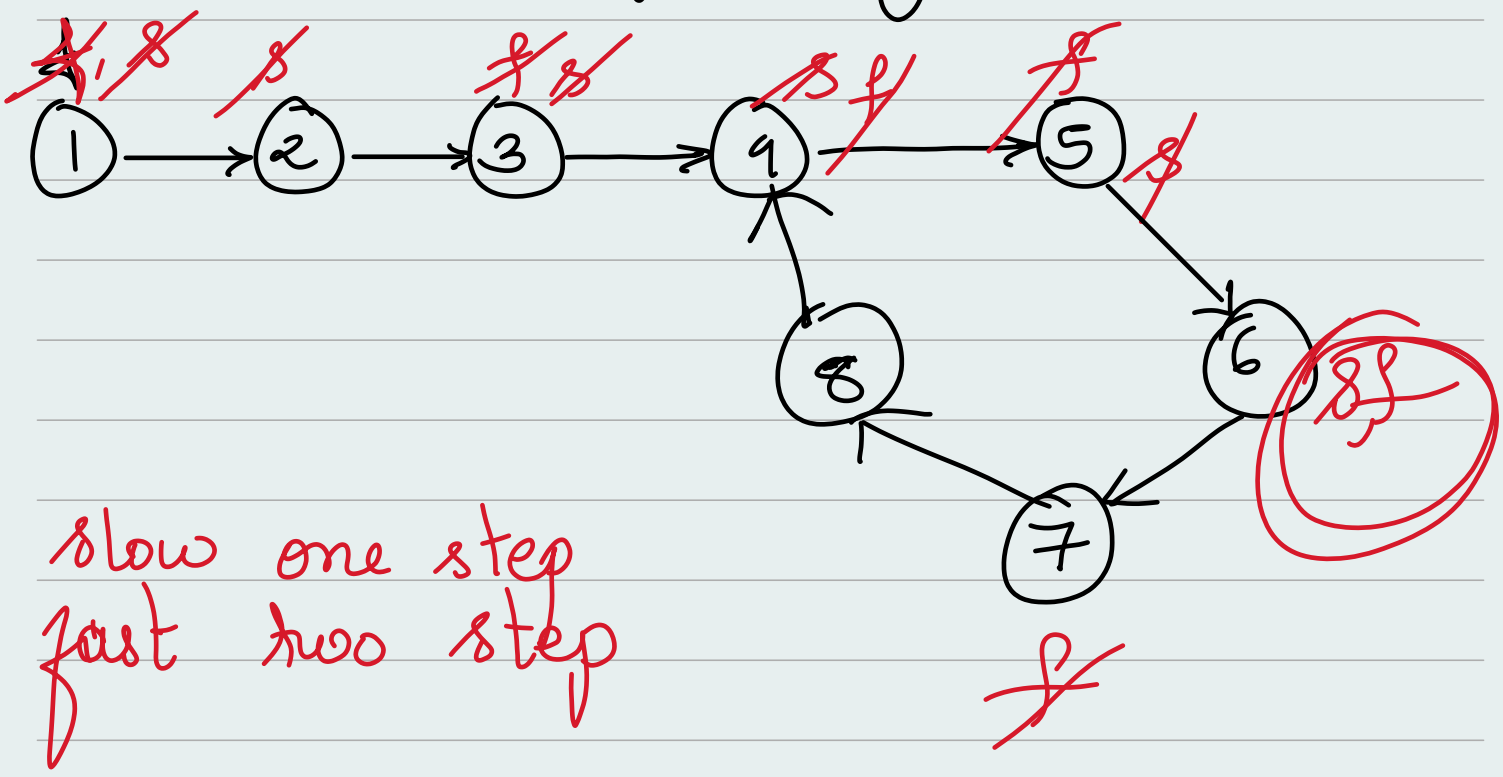
```
public ListNode detectCycle(
            ListNode head){

    int length =0;

    ListNode first = head;

    ListNode second = head;
// Detect a cycle first
  while (first != null && second != null){

        first = first.next.next;
    second = second.
```

```
        if (first == second) {
            length = lengthCycle(second);
            break;
        }
    }
}
if (length == 0) {
    return null;
}

// find the start node
ListNode f = head;
ListNode s = head;
while (length > 0) {
    s = s.next;
    length--;
}
// keep moving both forward and they will meet at cycle start.
while (f != s) {
    s = s.next;
    f = f.next;
```

```
    }
    return ss;
}
```

* By using Floyd's detection method for solving the above program with accuracy

→ (1) first detect the cycle and where they meet In the below example they meet at 6 node



slow one step
fast two step

(2) Now, after detecting a cycle we reset back the fast pointer at the beginning and slow pointer name the fast pointer as start pointer

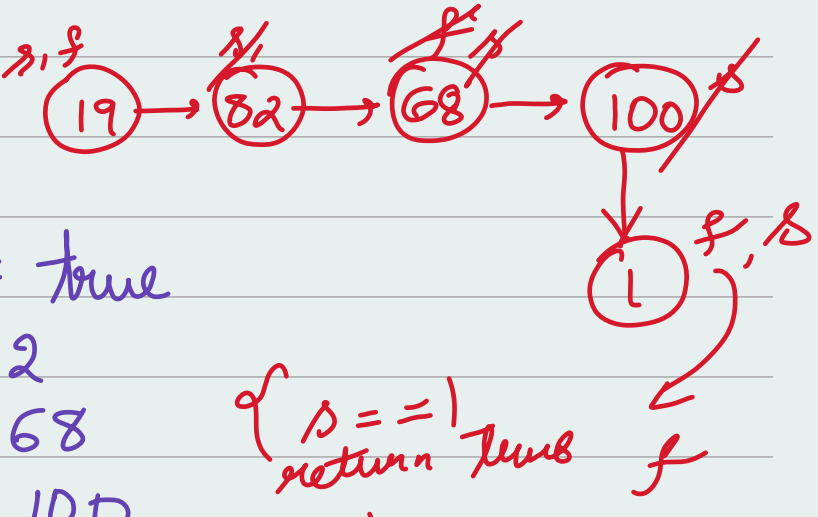
Now, the slowpointer would be at the same node (6)
we then keep moving these both pointers at a single step and finally they meet at 4

```
public class Solution {
public int lengthCycle(ListNode head){

ListNode slow = head;

ListNode fast = head;
// Detect cycle and find the length

int length = 0;
do {
slow = slow.next;
fast = fast.next.next;
length++;
} while (slow != fast);
return length;
}
public ListNode detectCycle(ListNode head)
{
// Detect cycle
ListNode slow = head;
ListNode fast = head;
while (fast != null && fast.next != null) {
 slow = slow.next;
fast = fast.next.next;
if (slow == fast) {
// Cycle detected, find the start node
ListNode start = head;
 while (start != slow) {
start = start.next;
slow = slow.next;
}
 return start; // This line is where they meet at
the cycle start node. } }
 // No cycle found
 return null;
 }
 }
```

optimized code

Q) Google Question

→ Determine whether the number is happy or not.



Input: n = 19

output = true

$1^2 + 9^2 = 82$

$8^2 + 2^2 = 68$

$6^2 + 8^2 = 100$

$1^2 + 0^2 + 0^2 = 1$ → loop will end when number is 1.

when the number is not a happy number

$12 = 1^2 + 2^2 = 5$

$5^2 = 25$

$2^2 + 5^2 = 29$

$2^2 + 9^2 = 85$

$8^2 + 5^2 = 64 + 25 = 89$

$9^2 + 8^2 = 81 + 64 = 145$

$1^2 + 4^2 + 5^2 = 1 + 16 + 25 = 42$

$2^2 + 4^2 = 20$

$2^2 + 0^2 = 4$

$4^2 = 16$

$1^2 + 6^2 = 37$

$7^2 + 3^2 = 49 + 9 = 58$

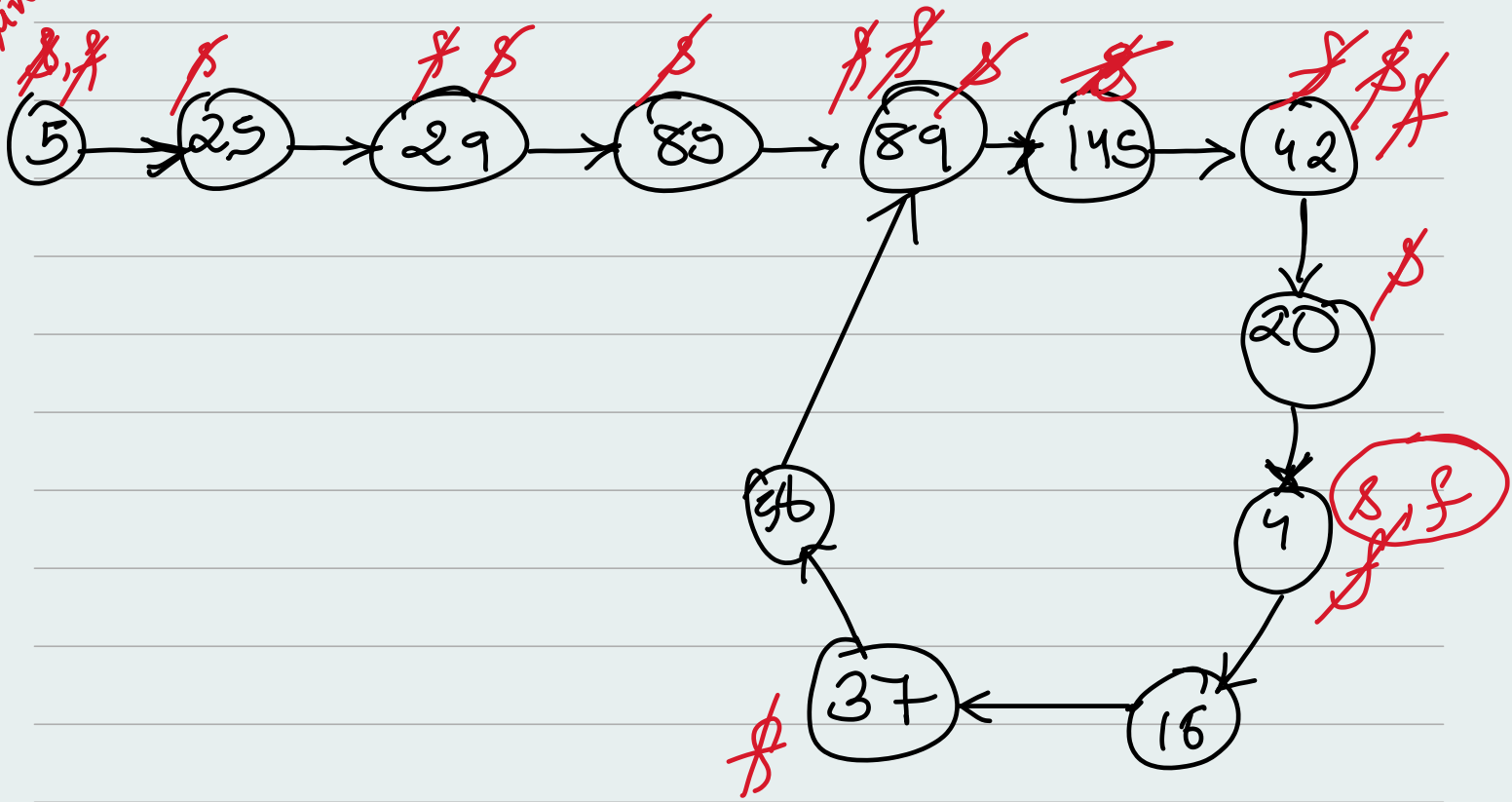
$8^2 + 5^2 = 64 + 25 = 89$

$8^2 + 9^2 = 81 + 64 = 145$

145

In loop

So looking at the results try to make a linked list and see how to solve it



From the above process the result is that it's a circular linked list now with the usual approach using two pointer's method.

Here, S. F are meeting at node 4.

Code :-

```
public boolean isHappy (int n){
        int slow = n;
        int fast = n;
        do {
            slow = findSquare(slow);
            fast = findSquare(findSquare(fast));
        } while( fast != slow);
        if (slow == 1) {
            return true
        }
        return false
    }
    private int findSquare( int number){
            int ans = 0;
            while(number > 0){
                int rem = number % 10;
                ans = ans + rem * rem
                number = number / 10
```

145 ⇒

145 % 10

= 5

$= 5^2 = 25$ //

Now,

145/10

14

140%10

= 4

$\Rightarrow 4^2 = 16$ //

Now

14/10

1%10

$\Rightarrow 1^2 = 1$ //

1/10

$\Rightarrow$ 0

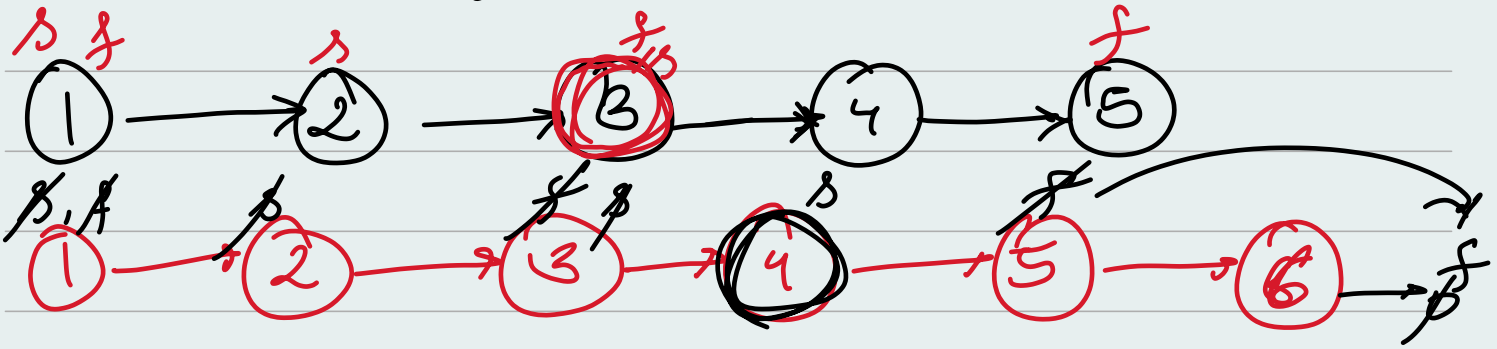
ans = ans + rem^2

```
    }
  return ans;
}
```

Q) find the middle of a linked list

→ Solution.

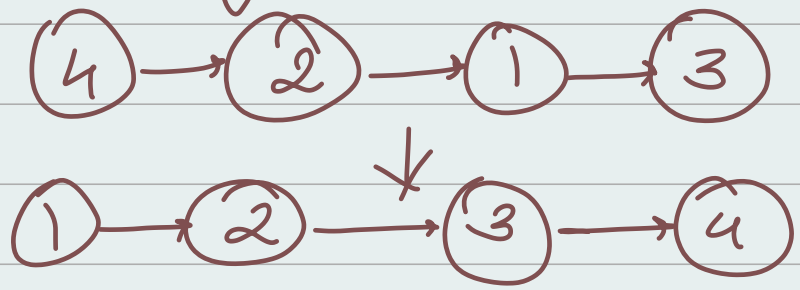
if there are two nodes return the second middle node



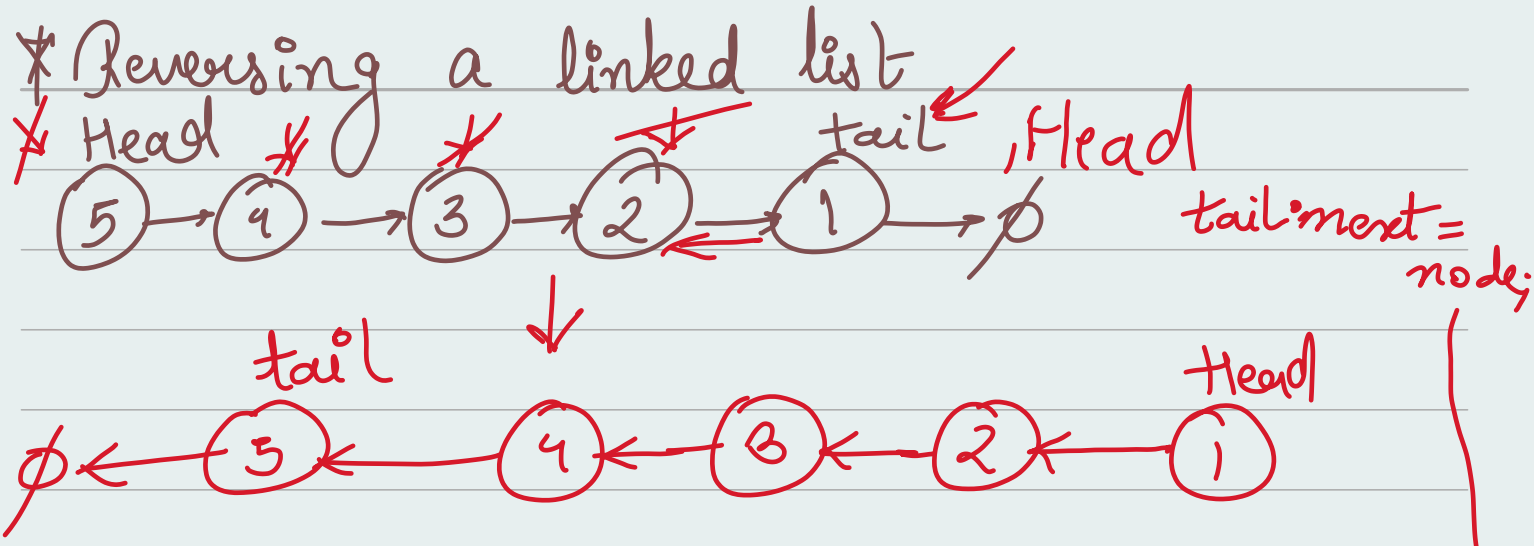
Code.

```
public ListNode middleNode(ListNode head){
    ListNode fast = head;
    ListNode slow = head;
    while(fast != null && fast.next != null){
        slow = slow.next;
        fast = fast.next;
    }
    return slow;
}
```

8) Sorting the Linked List

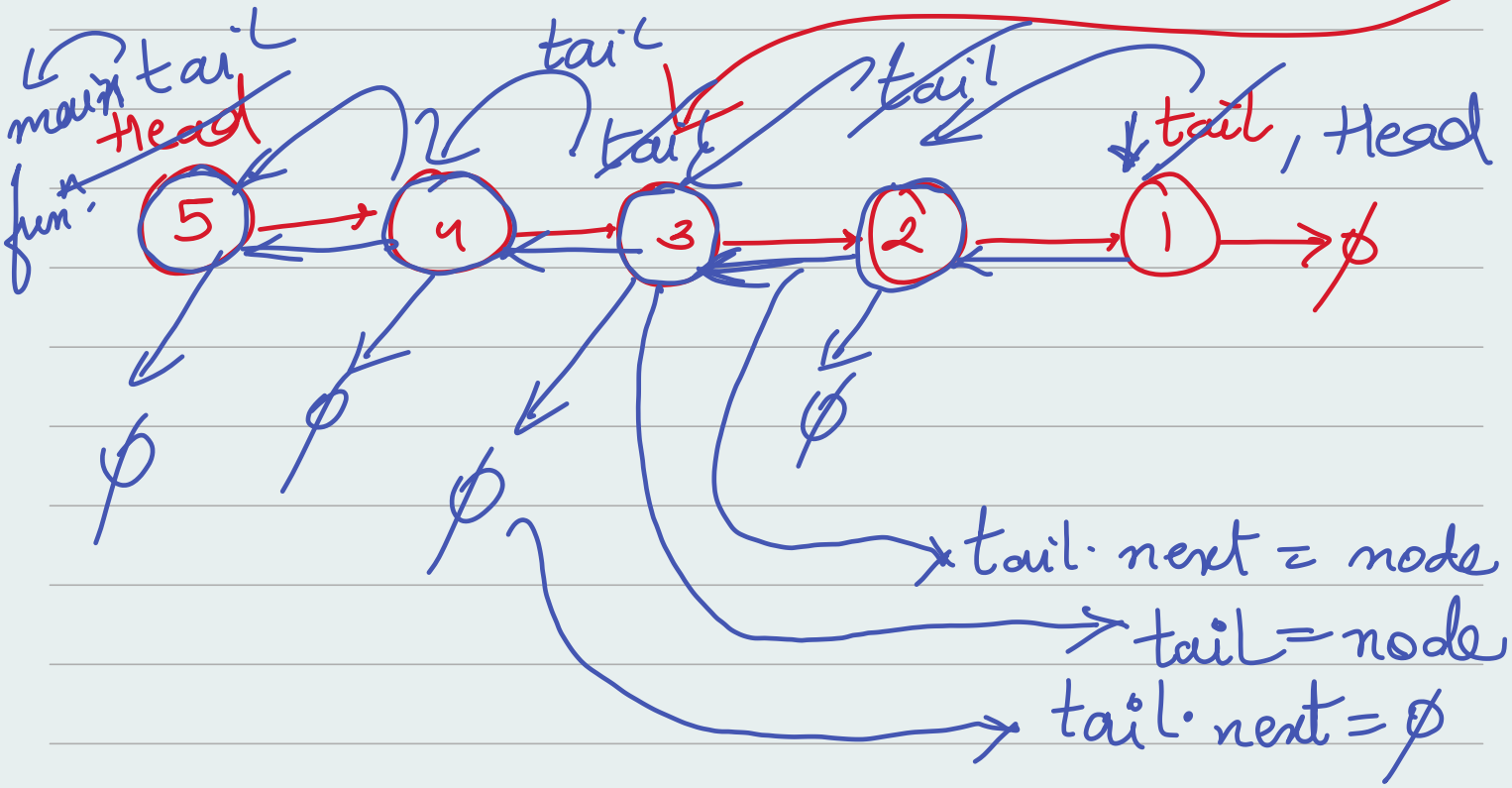


Reversing a linked list



Breaking down problems into smaller problems.

→ 1) Move a head pointer till it points to tail // Base Condition.



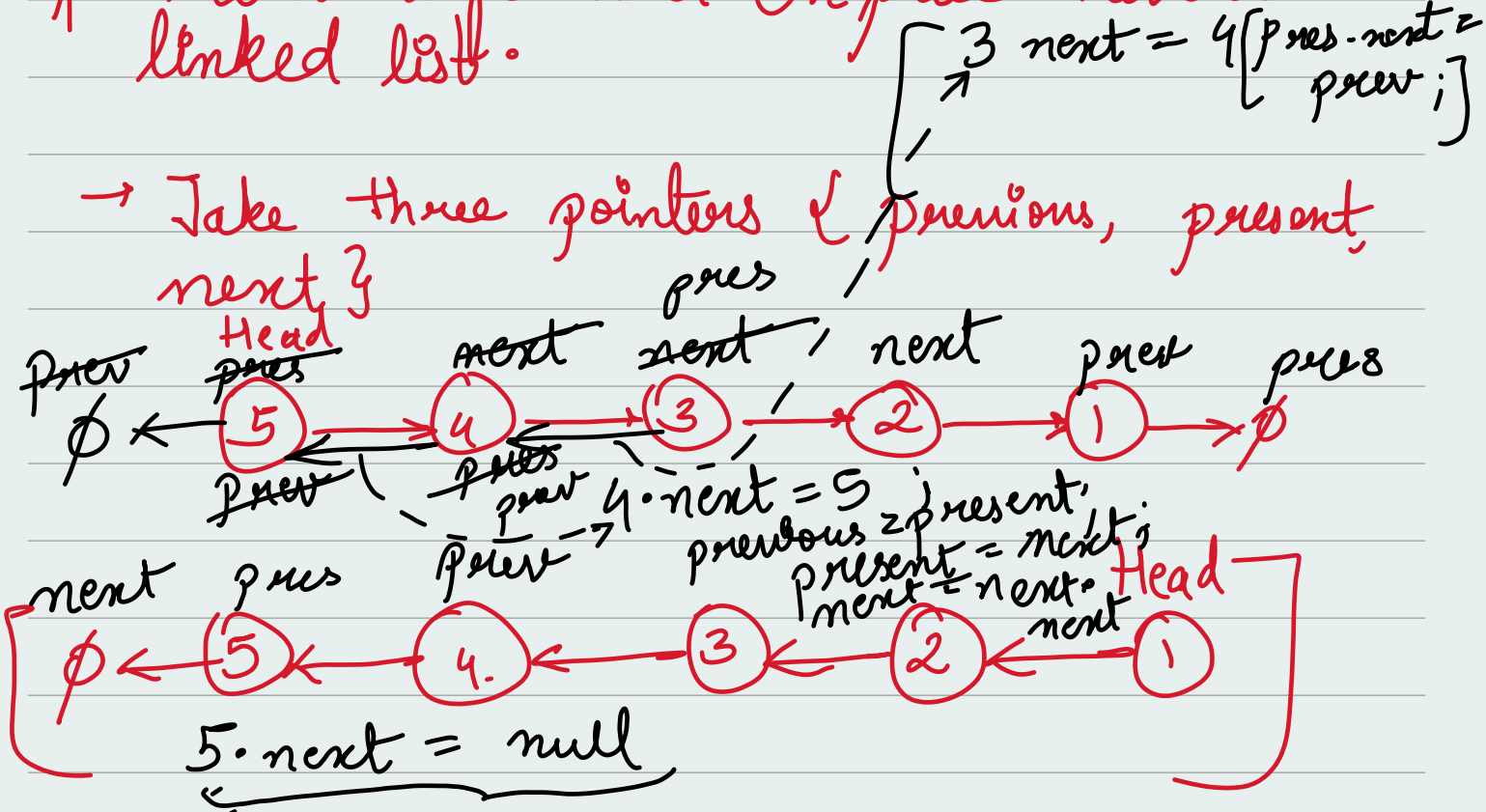
Code:

```
private void reverse(Node node){
    if(node == tail){
```

```
        head = tail;
        return;
    }

    reverse ( node.next) ;
    tail.next = node
     tail = node;
     tail.next = null;
}
```

* Most important In place reversal linked list.

→ Take three pointers (previous, present, next)




```
while present != null

    pres.next = prev;
    prev = pres;
    pres = next;
    next = next.next // null pointer
```

check.

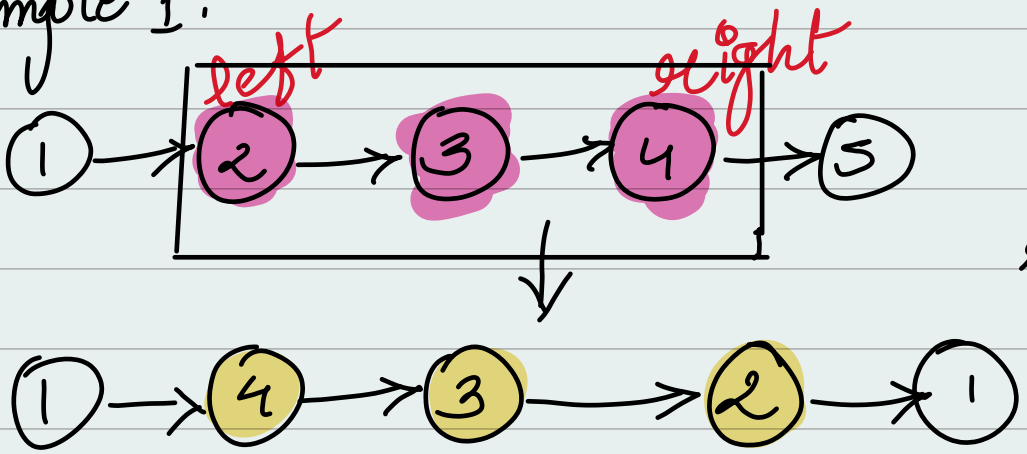
In the end present = null that means previous = head.

Code:

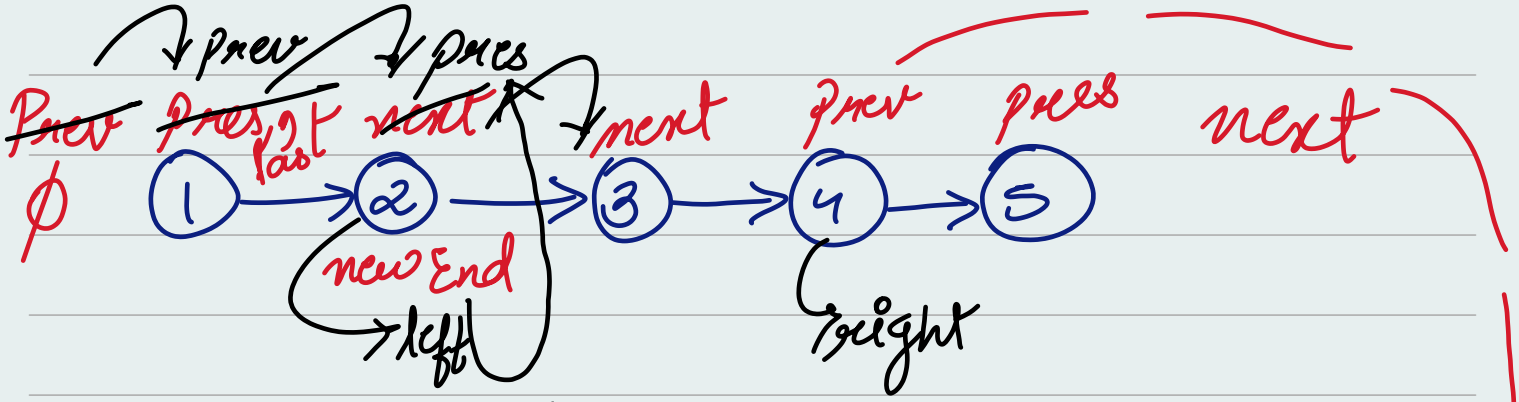
```
public void reverse (){
    if (size >2){
        return;
    }

    Node prev = null;
    Node present = head;
    Node next = present.next

    while (present != null){
        present.next = prev;
        prev = present;
        present = next
        if(next != null){
            next = next.next;
        }
    }
    head = prev;
}
```

8) Reverse a linked list but only a part of it

Example 1:



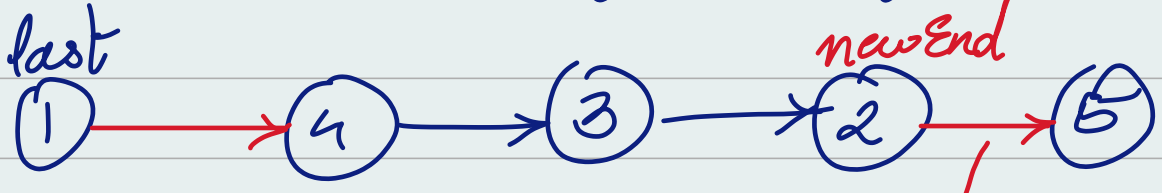
left = 2
right = 4



move present & previous ahead by one node

Now → last = previous;
newEnd = pres;

Now Reverse the list:



1.next = 4

2.next ≠ 5

↳ last.next = prev
newEnd.next = pres;

There might be a possibility that there is no node (1) on the list

start from (4) then last = null & head = prev;

Base condition :-

if left == right then return head;

Code :-

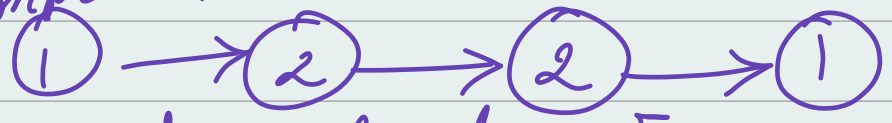
```
public ListNode reverseBetween(ListNode
        head, int left, int right){

    if (left == right){
        return head;
    }

    // skip the first left-1 nodes.
    ListNode present = head;
```

```
ListNode prev = null;

for (int i = 0; present != null &&
        i < left - 1; i++) {

    previous = present;
    present = present.next
}

ListNode last = prev
ListNode newEnd = present;

// Reverse between left and right

ListNode next = present.next;

for (int i = 0; present != null &&
        i < right - left + 1; i++) {
                    4 - 2 + 1 = 3 node

    present.next = prev;
    prev = present;
    present = next;

    if (next != null) {
        next = next.next;
```

```
        }
    }
    if (last != null) {
        last.next = prev;
    } else {
        head = prev;
    }
    newEnd.next = present;
    return head;
}
```

8) Palindrome Linked List

Example 1:

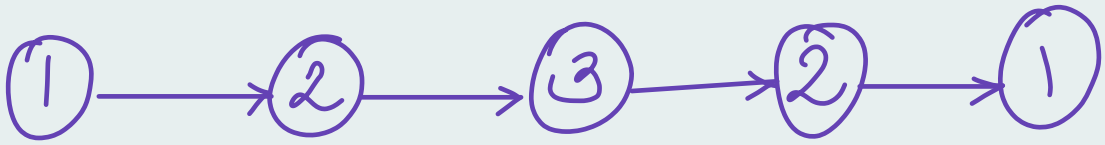


Input : head = [1, 2, 2, 1]
output : true

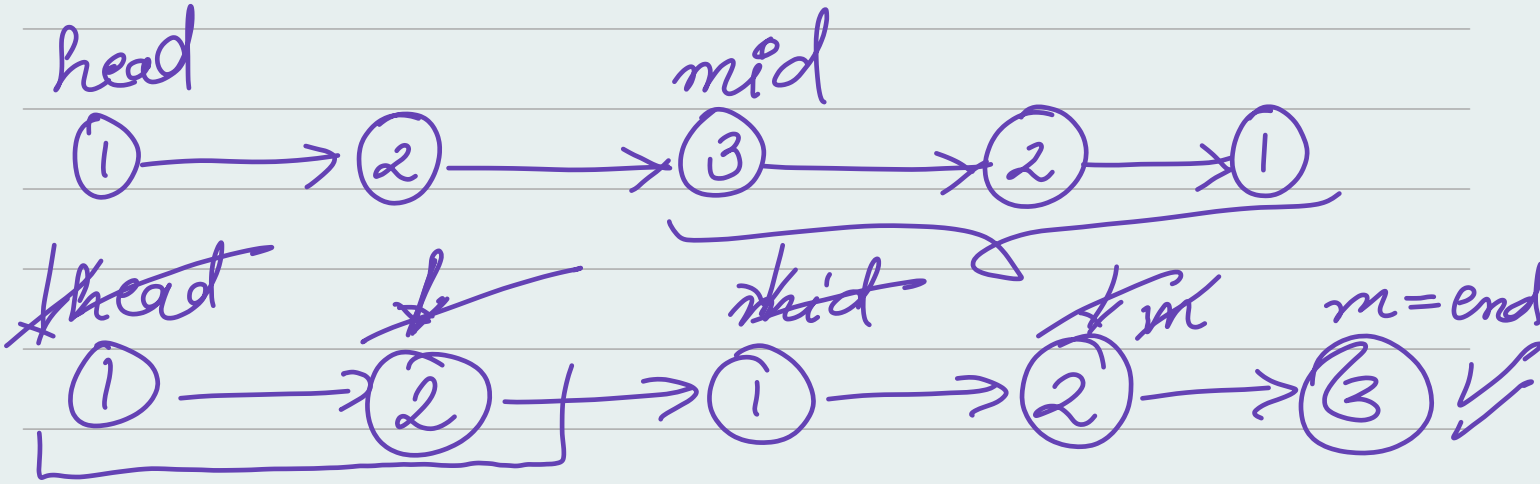
Example 2: Input : head = 1 → 2

output = false

head



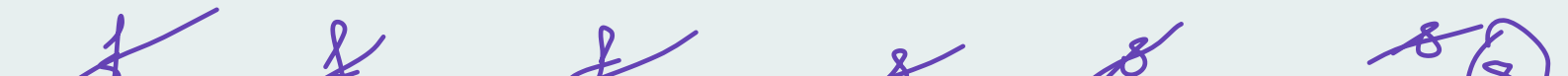
i) Find the middle of the linked list.
ii) Reverse the second half of the linked list
iii) Compare the first half with second half using two pointers
iv) Re-Reverse the second half of the linked list

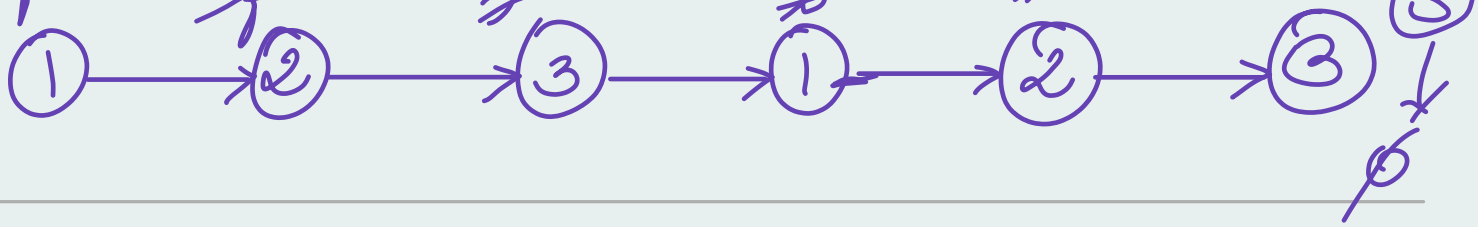


is head & mid equal yes move ahead or break.

For even number of elements :-

Reverse half of the linked list from index = 3.





// return true

// If any one of the nodes were not equal return false.

// After done traversing re-reverse the second half

Code :-

```
public boolean isPalindrome(ListNode
                            head) {

    ListNode mid = middleNode(head);

    ListNode headSecond = reverseList(mid);

    ListNode reReverseHead = headSecond;

    // Compare both the halves.

    while (head != null && headSecond!
                          = null) {

        if (head.val != headSecond.val) {
```

```
            break;
        }
        head = head.next;
        headSecond = headSecond.next;
    }
    reverseList(reReverseHead);
    if (head == null || headSecond == null){
        return true;
    }
    return false;
}
```

ReverseList :- Inplace

```
public ListNode reverseList(ListNode head){
    if(head == null){
        return head;
    }
    ListNode prev = null;
    ListNode present = head;
    ListNode next = present.next;
```

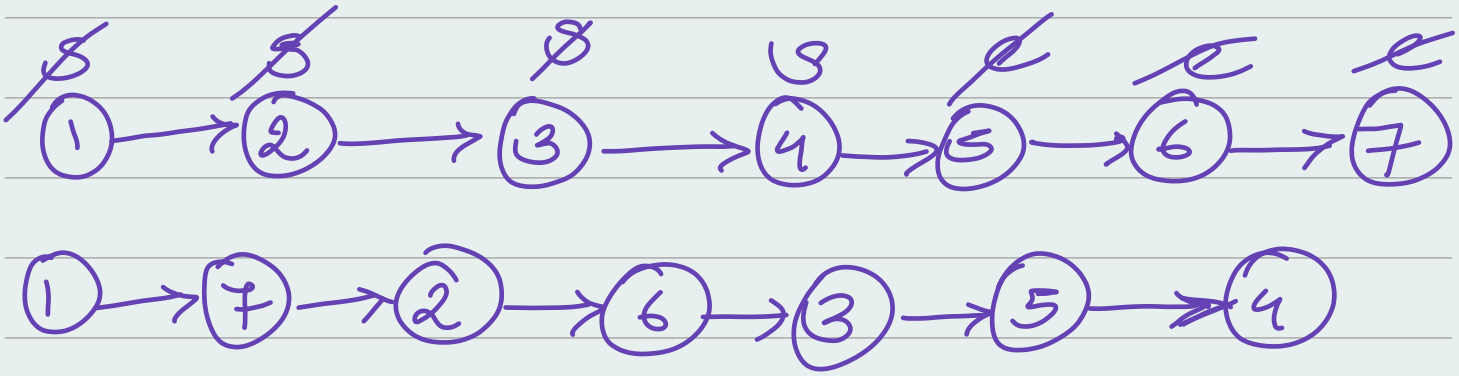
```
while (present != null){
        present.next = prev;
        prev = present;
            present = next;
        if (next != null){
            next = next.next;
    }
}
    return head;
}
```

Middle Node :-

```
public ListNode middleNode (
                    ListNode head){

    ListNode fast = head;
    ListNode slow = head;

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

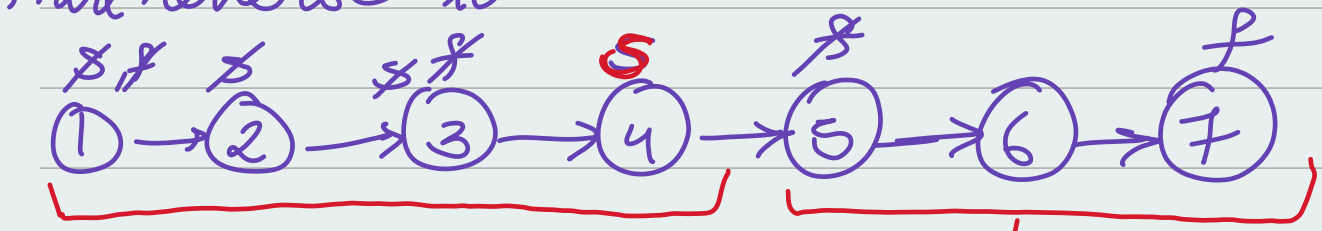
y

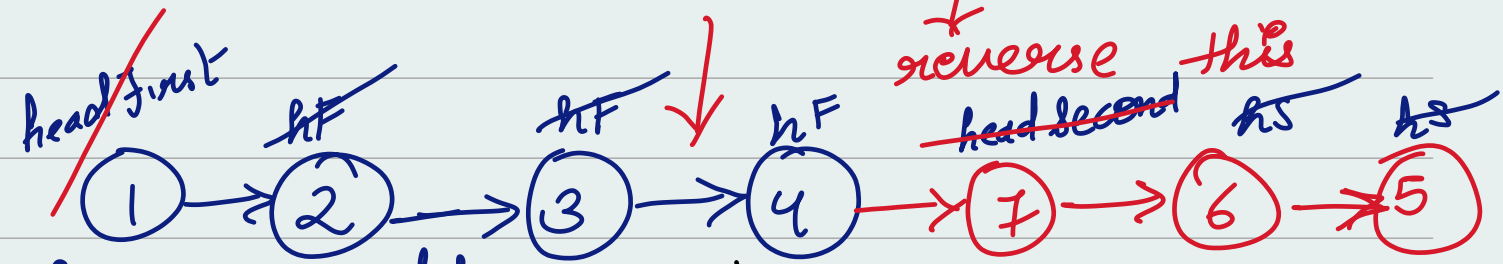
8) Reorder List : Goog, FB..

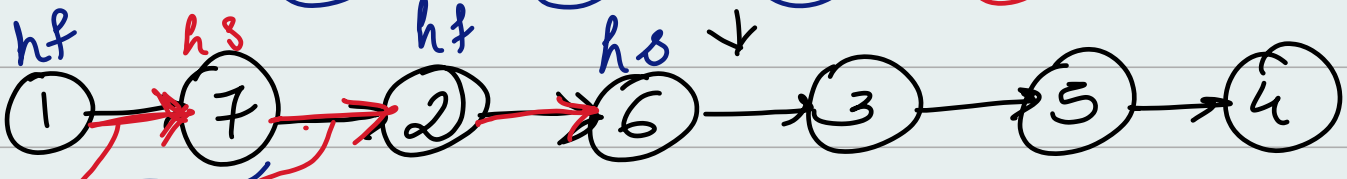


How to simplify,

We will take middle element and from there reverse it







1. next = 7

so,

head First. next = head Second;

Now we want to Merge (7) → (2) node for that we have to use temp variable.

temp = hf.next

hf.next = hs;

hf = temp

temp = hs.next

hs.next = hf;

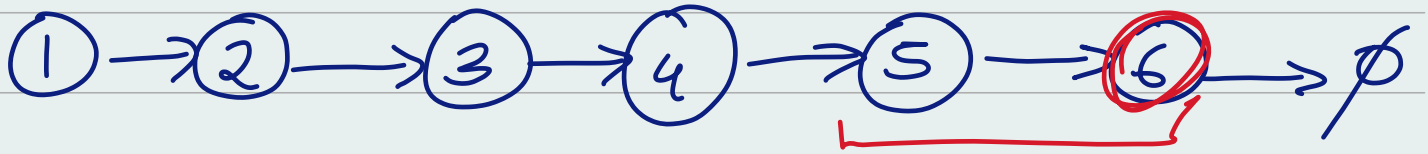
hs = temp;

Code:-

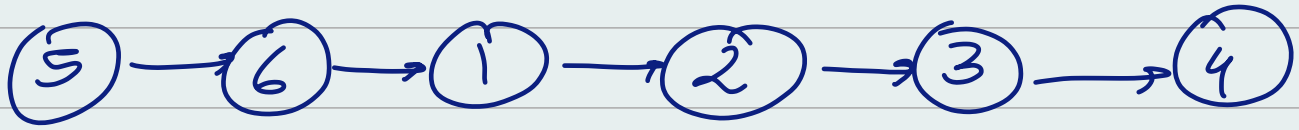
```
public void reorderList (ListNode head){
    if (head == null || head.next == null){
        return;
    }
    ListNode mid = middleNode(head);
    ListNode hs = reverseList(mid);
```

```
ListNode hf = head;

// rearrange
while (hf != null && hs != null){

    ListNode temp = hf.next;

    hf.next = hs;

    hf = temp ;

    temp = hs.next;

    hs.next = hf

    hs = temp;
// next of tail / to null.
    if (hf != null) {

        hf.next = null;

    }
```

8) Rotated Linked List



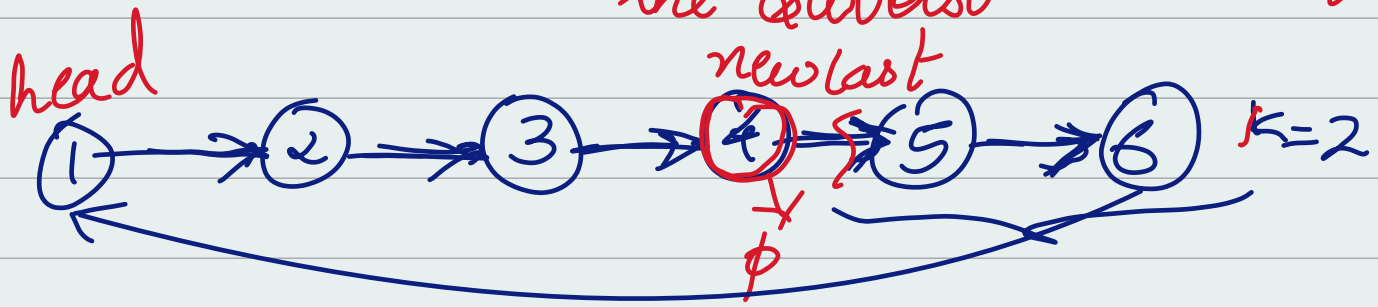
$k = 2$



① original last node · next will be equal to original head.

② New head = start of the sublist.

③ new tail = node before start of the sublist.



i) find the last node & connect it with head

skip = l - k

skip = 6 - 2 = 4

= 4 nodes

skipped

head = newlast.next.

newlast.next = ∅ [④ → ∅]

→ If it is rotated 12 times the list would be same cause it is multiple of 2. If it is rotated 8 times it means 6+2 the list would be rotated 2 times.

Formula : **rotation = k % l**

Time complexity : O(N)

Space complexity : O(1)

```
public ListNode rotateRight (
        ListNode head, int k){
if (k <=0 || head == null || head.
    next == null){
      return head;
}

ListNode last = head;
int length = 1;
```

```
while ( last.next != null ){
    last = last.next;
    length++;
}
    last.next = head;
    int rotations = k % length;

    int skip = l - rotations
ListNode newlast = head;
for(int i=0 ; i< skip -1 ; i++)
    newlast = newlast.next;

}
    head = newlast.next;
    newlast.next = null;
    return head;

}
```