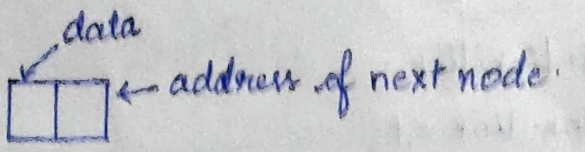
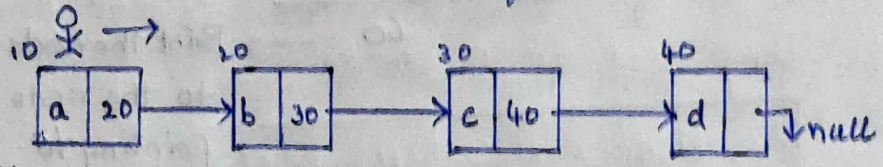
Linked List:

Collection of nodes/structures/classes which are all self referential.




```
Node node {
    int data;
    Node next;
}
```

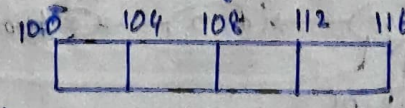
Self Referential:- Every node type has structures inside it which will point to the structure of same type.



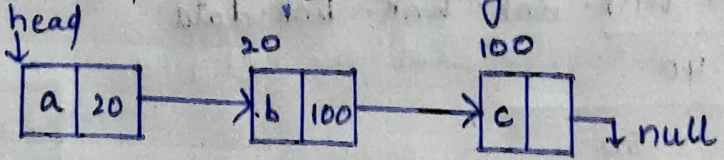
Properties:-

1. only one direction i.e only left to right not right to left.
2. every node has one connection.

→ In array the memory allocation is contiguous

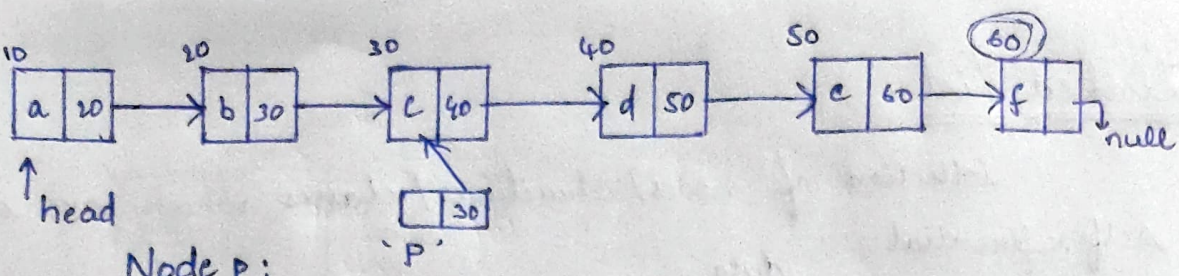


→ In case of Linked List the memory allocation is dynamic.



→ Accessing elements in linked list T.C is O(n) because it is having reference only to the first Node. We need to traverse to that node.

ex:-




```
Node p;
P= head.link.link;
P.link.link.link =P;
Head.link= P.link;
S.O.pln(head.link.link.link.link.data)
```

1) → P= head.link.link

10

20

30

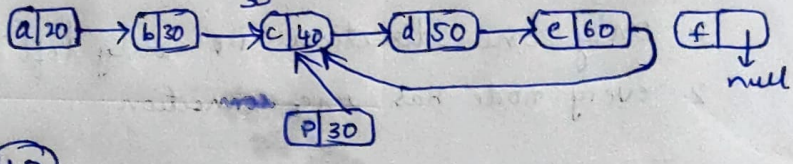
2) P.link.link.link = P

30

40

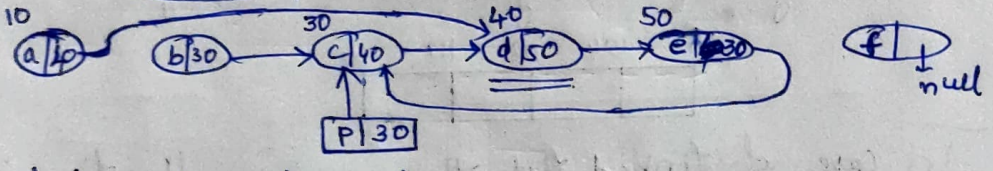
50

60 → Point the node having 60 to the node which 'P' is pointing to.



3) head.link = P.link

10 = 30 → 40



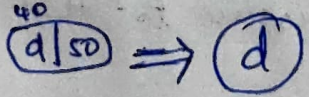
4) head.link.link.link.link.data.

40

50

30

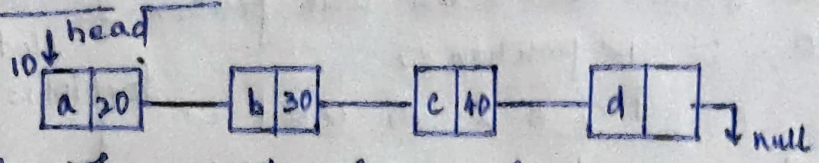
40.



Types of Linked Lists:

1. Singly Linked List
2. Circular Linked List
3. Doubly Linked List

Singly Linked List:



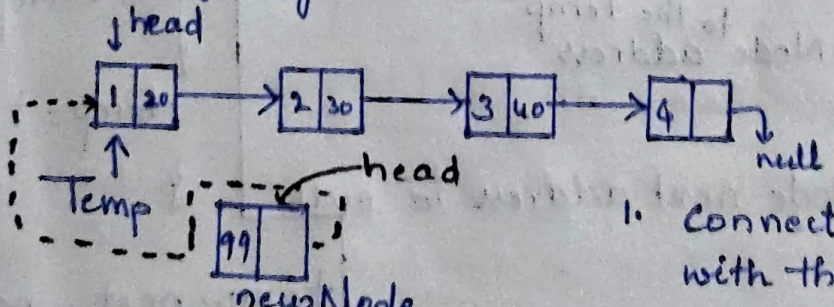
Traversing Through The Linked List:

① create a Temp Node and point it to the first / head node.
② Traverse through the List until Temp is not null.
③ Print the current node data
④ Point the temp to the next node.

```
traverse(Node head)
{
1. Node temp = head;
2. while(temp != null){
3.     s.o.p(temp.data);
4.     temp = temp.next;
   }
```

Insertion:-

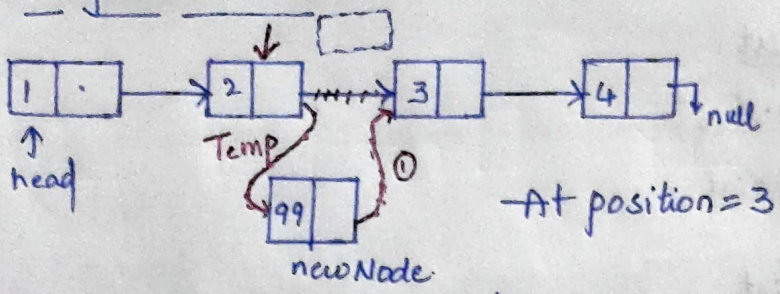
a) Insertion at begining:-



1. Connect the new node with the first node.
2. point the new node to head.

```
newNode.next = temp.next;
head = newNode
```

b) Insert at given Position:



→ iterate through the linked list until position - 2.

i = 0 ; i < position - 2

i < 3 - 2 ⇒ 0 < 1 ✓

i = 1 ; 1 ≮ 1 ⇒ stop

```
Node temp = head;
for(int i=0; i<pos-2; i++)
{
    temp = temp.next;
}
```

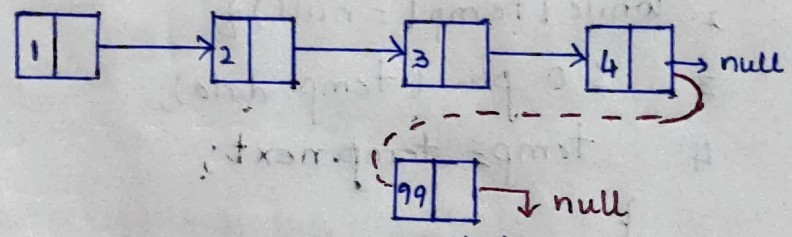
① Point the new node to the temp.next.

② Because first we need to get the next node address of temp and then Remove the connection b/w Temp and Next node.

③ Point the Temp.next to new node.

```
newNode.next = temp.next;
temp.next = newNode;
```

c) Insert at Last:



Step 1:
Iterate through the list until the last node.

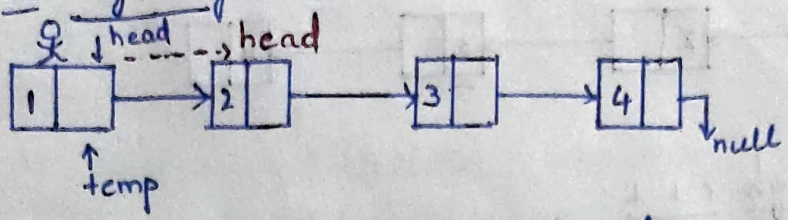
Step 2:
Point the newNode to the temp. address

Step 3:
The new Node next address to null.

```
while (temp.next != null)
{
    temp = temp.next;
}
temp.next = newNode
newNode.next = null;
```

Deletion:

a) Delete at beginning:-



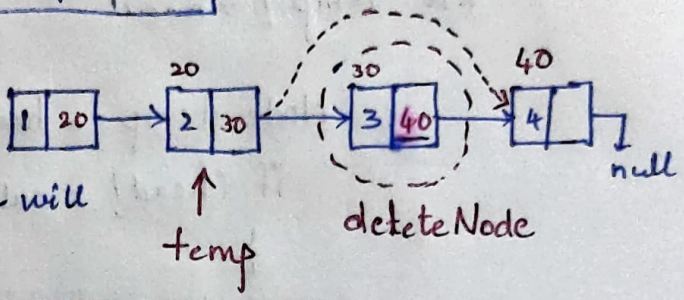
1) Move the head node to the next node.

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

b) Delete at Position:

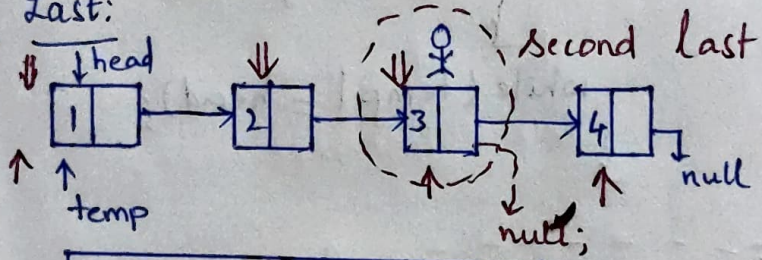
Position = 3

→ go upto position - 1 which will be the temp node.



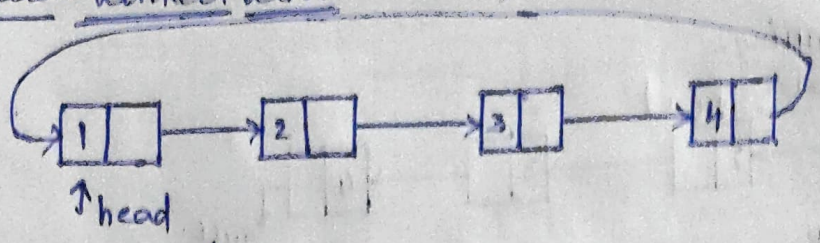

```
Node deleteNode = temp.next;
temp.next = deleteNode.next;
```

c) Delete at Last:




```
while(temp.next != null) {
    Second Last = temp
    temp = temp.next;
}
second Last.next = null
```

Circular Linked List: (C.L.L)

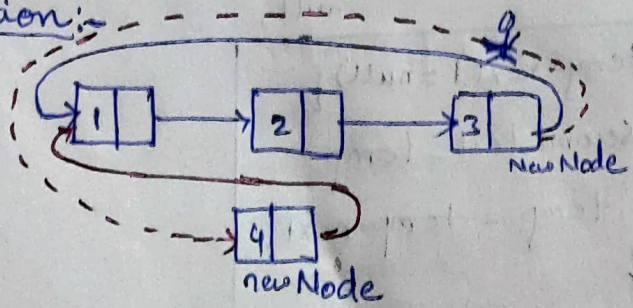


→ Traverse through C.L.L:

→ Iterate through the list and print the element till temp != head.

```
Node temp = head;
if (head != null)
{
do {
    s.o.pln (temp.data + " ");
    temp = temp.next;
}
while (temp != head);
}
```

Insertion:-

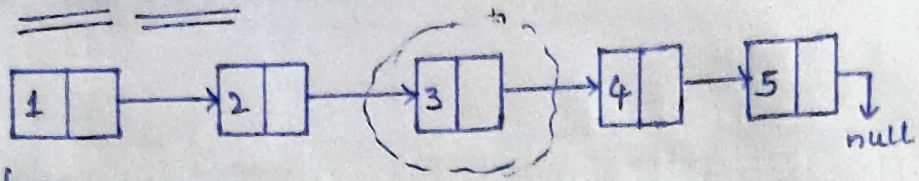


→ Go until last node & then

newNode.next = temp.next (nothing but the circular element)

temp.next = newNode

Middle of Linked List;

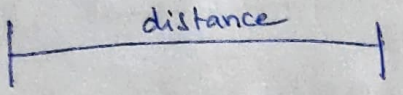


Brute force;

→ Traverse through each node. and increment the length variable

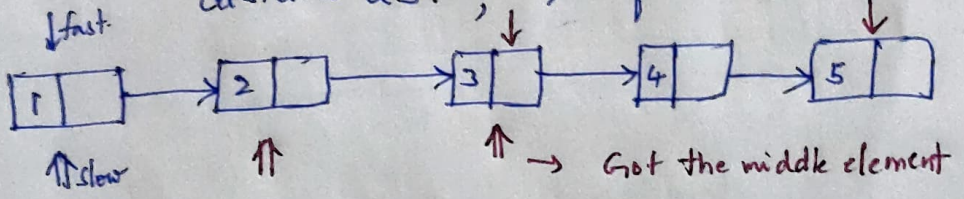
→ length/2 ⇒ 5/2 ⇒ (2) go to this element.

→ Fast and Slow Pointer Approach;



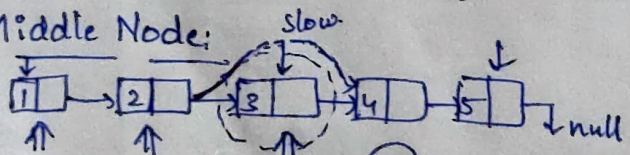
distance = s x t

distance ∝ s. , so fast car = 2 x Slow car



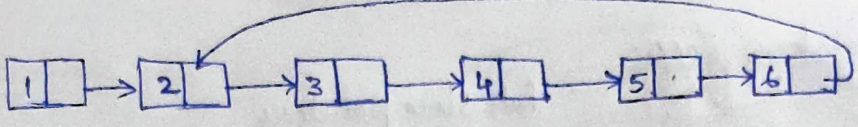

```
while (fast != null && fast.next!=null)
{
  slow = slow.next;
  fast = fast.next.next;
}
return slow.data;
```

* Delete the Middle Node;



→ In this case, we need to main Prev. as well.

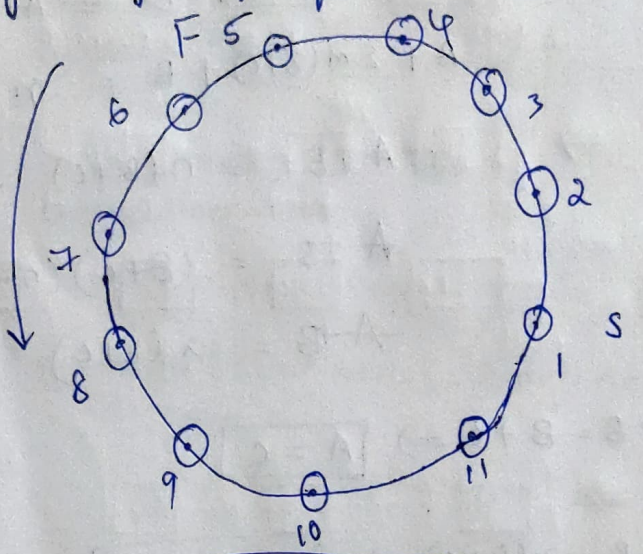
```
while (fast!=null & fast.next!=null){
{
    Prev = slow;
    slow = slow.next;
    fast = fast.next.next;
}
Prev.next = slow.next
```



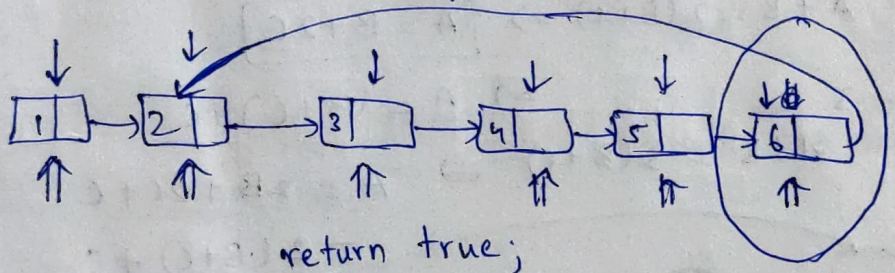
1. Whether there exists a cycle (or) not.

Floyd Warshal Algorithm:

* We will take 2 pointers fast and slow, fast moves by 2 steps & slow by 1 step. if fast == slow there exists a cycle.



F	S	Diff
5	1	7
7	2	6
9	3	5
11	4	4
2	5	3
4	6	2
6	7	1
7	7	0 ✓

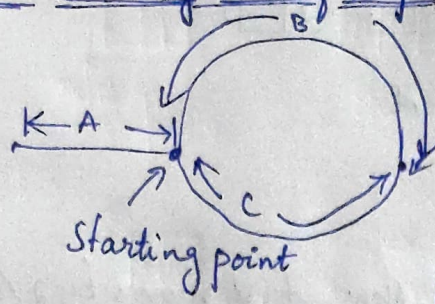


return true;

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

```
if (slow == fast)
return true;
```

* First Starting Point of Loop:



fast & slow pointers meeting point

(Distance covered by slow) * 2 = Distance covered by fast.

$$[A + m(B+C) + B] * 2 = A + n(B+C) + B$$

$$2A + 2m(B+C) + 2B = A + n(B+C) + B$$

$$A + 2m(B+C) + B = n(B+C)$$

$$A + B = n(B+C) - 2m(B+C)$$

$$A + B = (B+C)(n-2m)$$

$$A + B = \lambda(B+C)$$

$\lambda = 1$

$A + B = B + C \Rightarrow \boxed{A = C}$

$\lambda = 2$

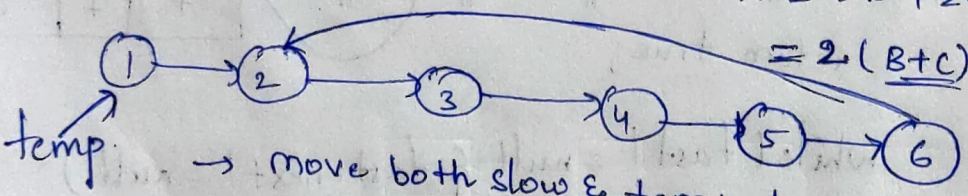
$A + B = 2(B+C) \Rightarrow \boxed{A = B + 2C}$

$\lambda = 3$

$A + B = 3(B+C) \Rightarrow A = (B+C) + C$

$A = 2B + 2C + C$

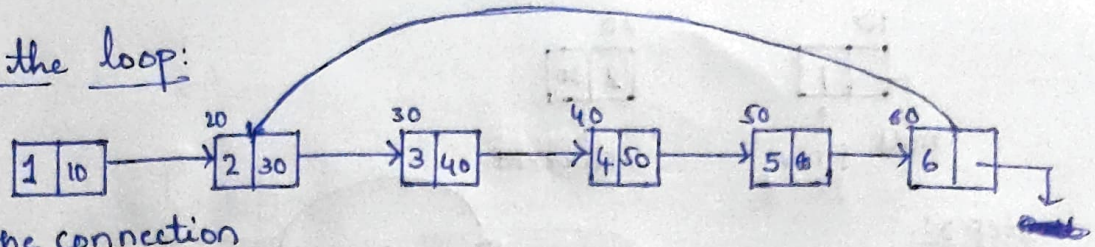
$= 2(B+C) + C$



→ move both slow & temp by one location till they meet.

```
while(slow != temp)
{
    slow = slow.next;
    temp = temp.next;
}
return slow.data;
```

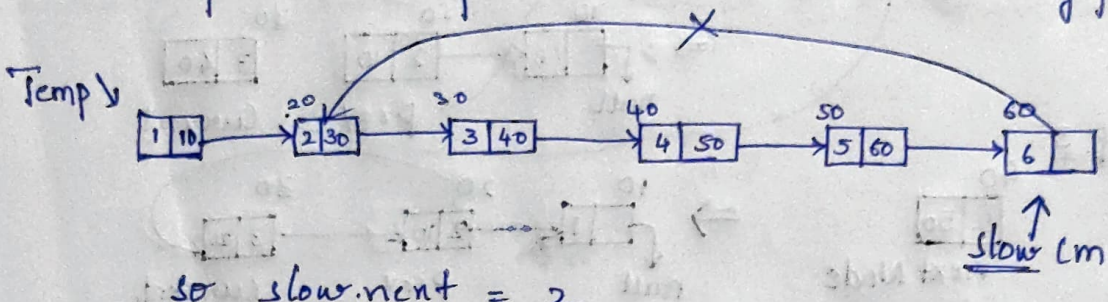
→ Remove the loop:



→ Break the connection

→ Make the last node next address to null

→ Take 2 pointers Temp and slow (which is the meeting point)

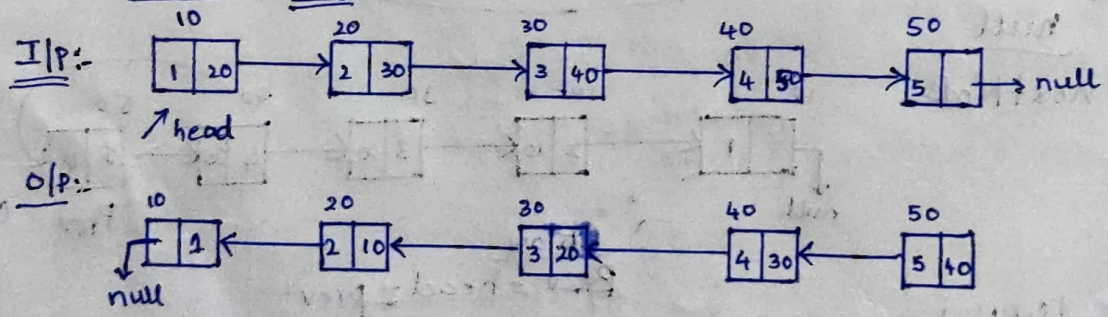


so slow.next = 2

temp.next = 2 , So go till slow.next != temp.next

```
while (slow.next != temp.next)
{
    slow = (slow.next);
    temp = temp.next;
}
slow.next = null → next address to null.
```

Reverse Linked List:



Step 1:

Prev = null

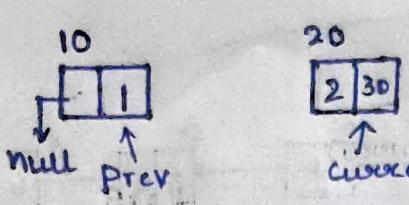
= current

= Nextnode = current.next

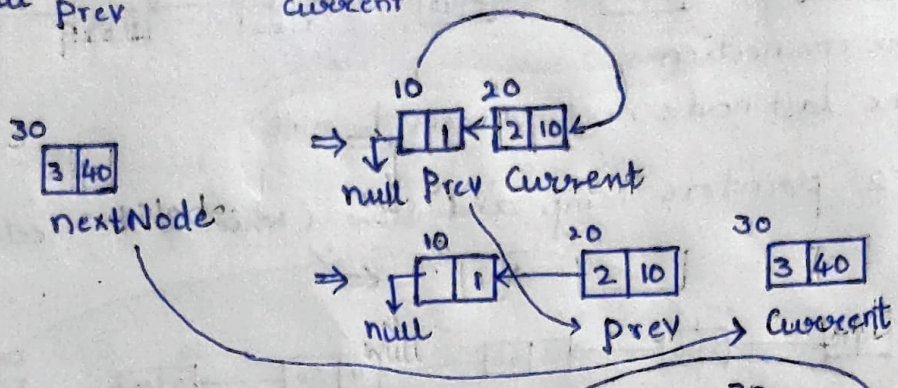
→ As seen in the output, we need to make the current next address point to prev.

→ Move the / Make the Current to prev

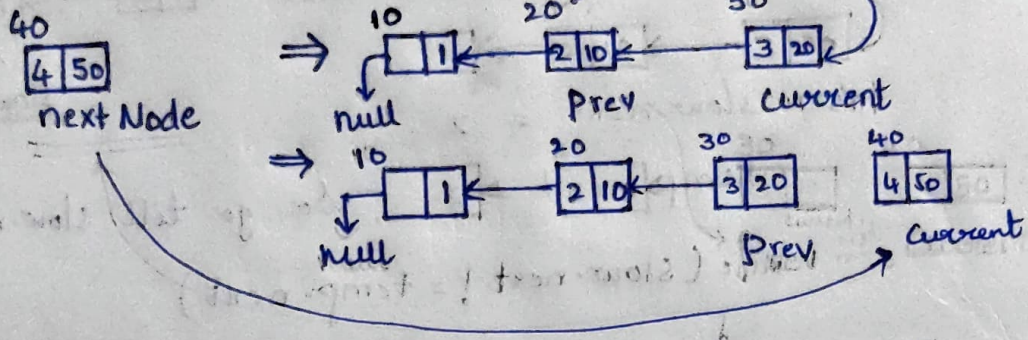
→ Nextnode to Current.



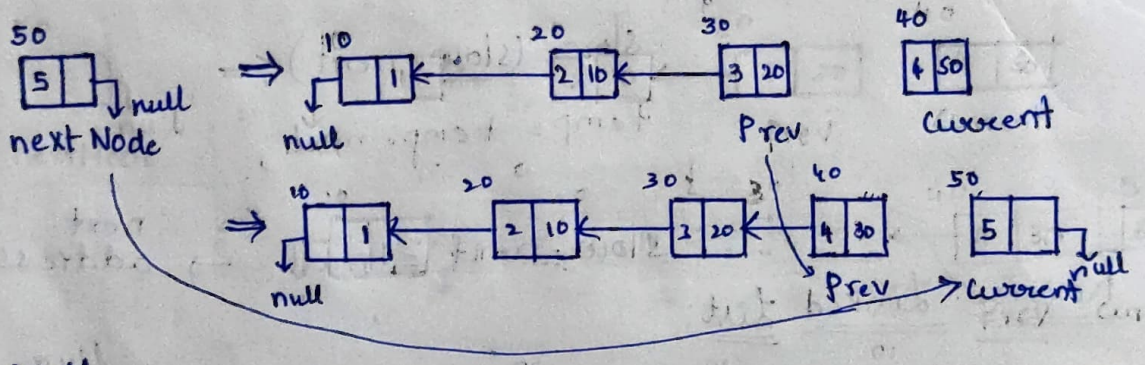
Step 2:



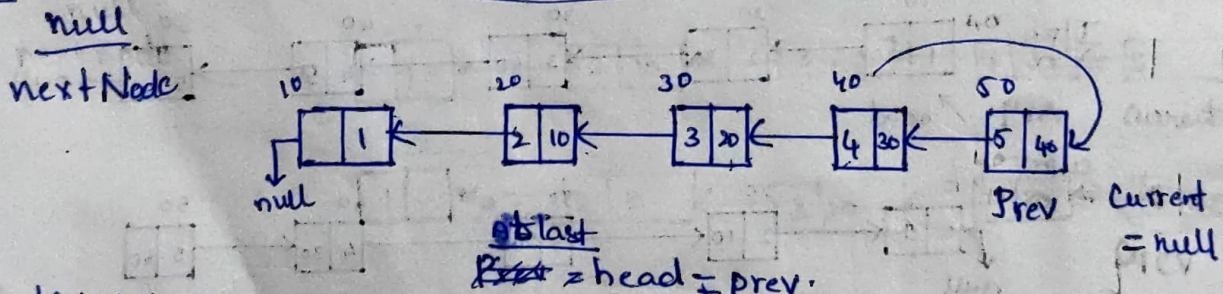
Step 3:



Step 4:



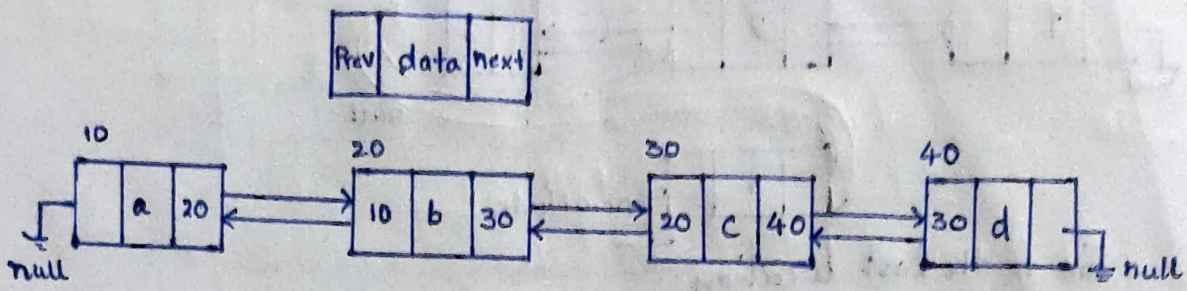
Step 5:



~~Node~~ last
~~Node~~ = head = prev;

```
ListNode prev = Null;
ListNode current = head
  while (current != null)
   {
     ListNode nextNode = current.next;
      current.next = prev ;
        prev = current;
       current = nextNode;
   }
      head = prev;
   return head;
```

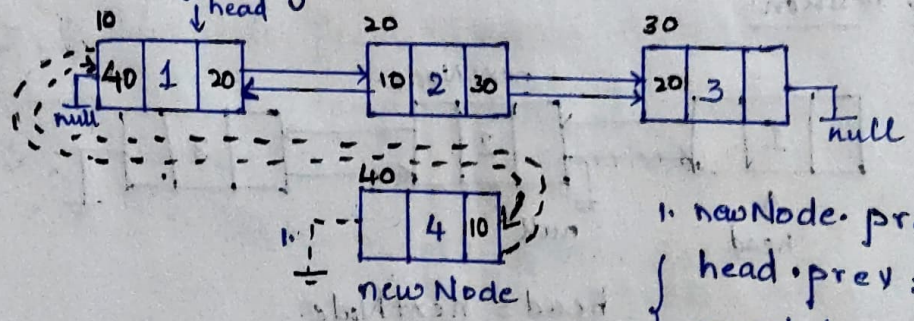
Doubly Linked List:




```
Node node {
int data;
Node next;
Node prev;
}
```

Insertion:-

a) Insertion at Beginning:



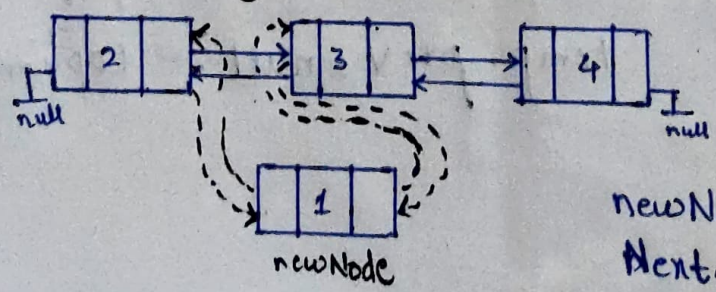
1. newNode.prev = null.
2. head.prev = newNode
3. newNode.next = head
4. head = newNode

Tightly coupled →

b) Insert at Position:

```
for(int i=0; i<Pos-2; i++){
    temp = temp.next;
}
```

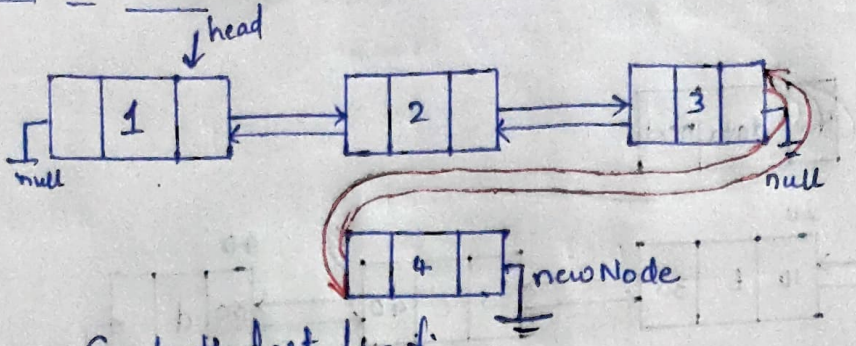
②



* Get the Address of Next node ~~to~~ newNode address

```
newNode.next = Next;
Next.prev = newNode
temp.next = newNode
newNode.prev = temp;
```

c) Insert at Last:



→ Go to the last location.

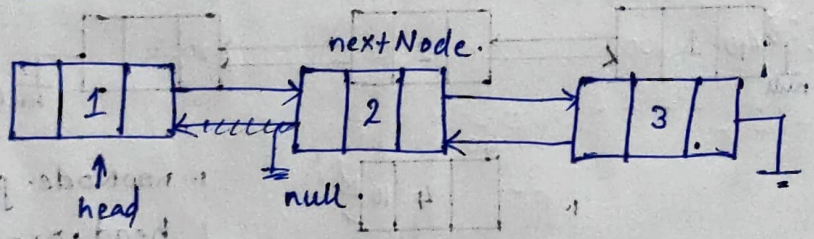
```
while (temp.next != null) {
    temp = temp.next;
}

temp.next = newNode
newNode.prev = temp
newNode.next = null
```

Deletion:

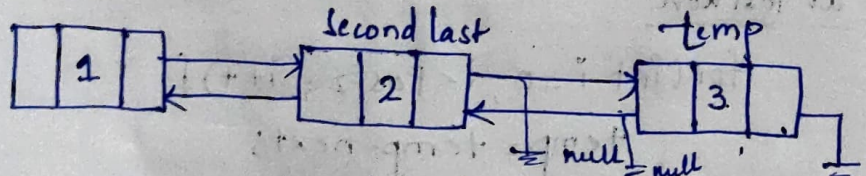
a) Delete at Position:

at Start:-



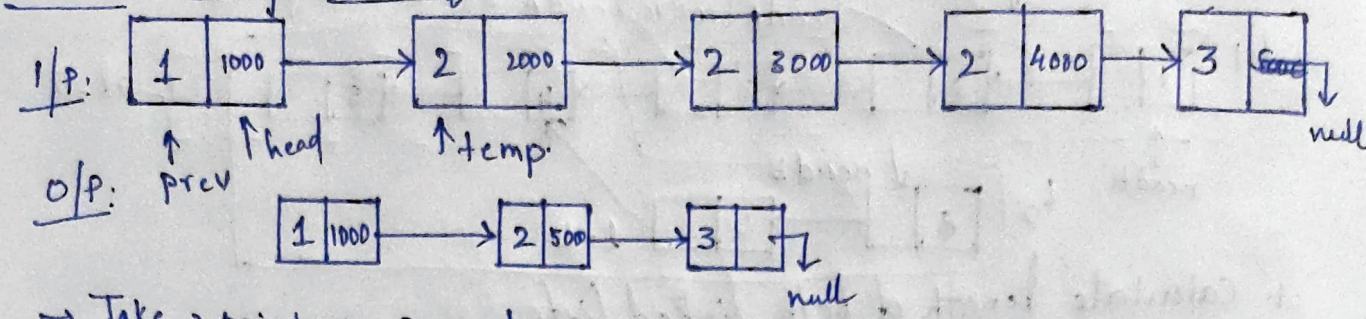

```
head = nextNode.
nextNode.prev = null;
```

At last:




```
Second last.next = null;
temp.prev = null; (optional)
```

Delete Duplicates from a Sorted List:



→ Take 2 pointers, prev, temp.

→ Iterate through the list till temp is not null

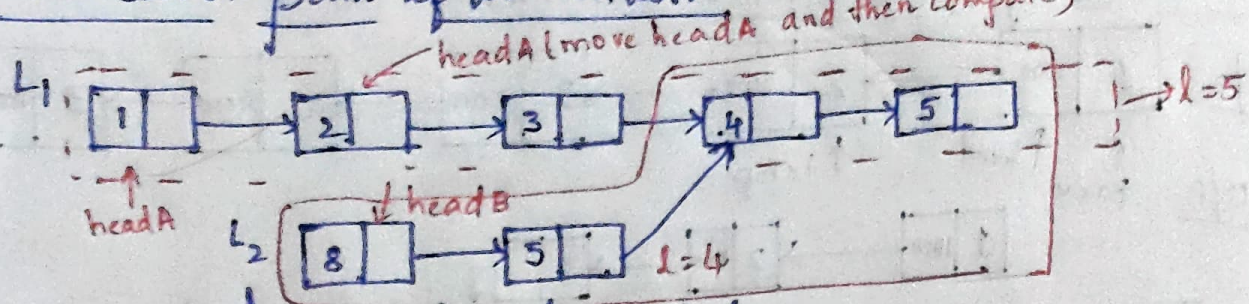
→ compare the data of temp & prev.

→ If equal make prev next to temp's next and move temp to temp.next

→ else move temp and prev.

```
if (head == null) {
    retur null;
}
ListNode prev = head
         temp = prev.next;
while (temp != null) {
    if (temp.data == prev.data) {
        Prev.next = temp.next
        temp = temp.next;
        continue;
    }
    temp = temp.next;
    prev = prev.next;
}
return head;
```

Intersection point of Linked List.



1. Calculate length of both linked lists.

   $L_1 = 5, \quad L_2 = 4$

2. Now, make the length of both linked lists equal by moving the highest length linked list to next node.

3. Then move both the linked lists and check if they are pointing to the same node else move to the next node.

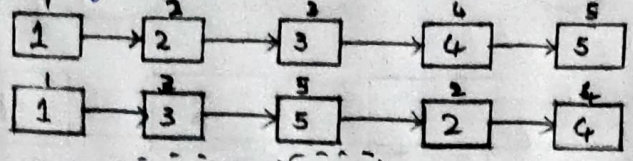
```
int lenA = getlength(headA)
int lenB = getlength(headB)
while(lenA > lenB)
  headA = headA.next;
  lenA--;
}
while(lenB > lenA){
  headB = headB.next;
  lenB--;
}
while(headA != headB)
{
  headA = headA.next;
  headB = headB.next;
}
return headA;
```

So:- O/P:- 4

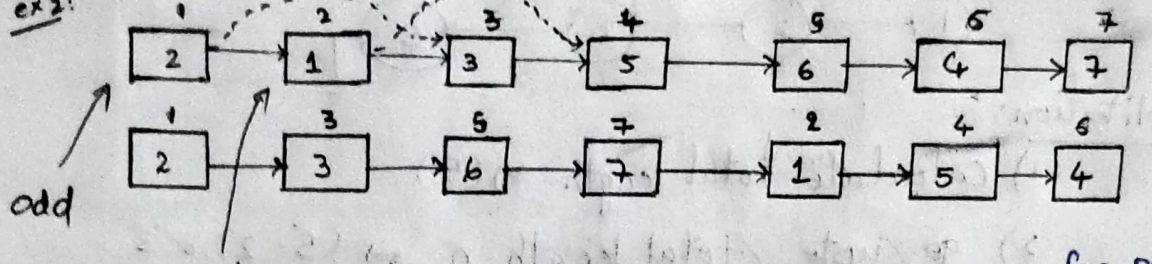
Odd Even Linked List:

→ Given a LL, group all the nodes with odd and even indices together. The relative order of elements should remain same.

ex:1



ex:2



i.e odd.next = odd.next.next → for odd

→ Then move the odd pointer to next odd indices.

even.next = even.next.next → for even.

→ Then move the even pointer to next even index.

```
ListNode odd = head;
ListNode even = head.next;
ListNode evenHead = even;
while (even != null && even.next != null){
    odd.next = odd.next.next;
    odd = odd.next;
    even.next = even.next.next;
    even = even.next;
}
odd.next = evenHead; → Assigning even list at the end of odd list
return head;
```

odd = (2), even = (1) evenhead = (1)

1) (2)→(3) (1)→(5)

2) (2)→(3)→(6) (1)→(5)→(4)

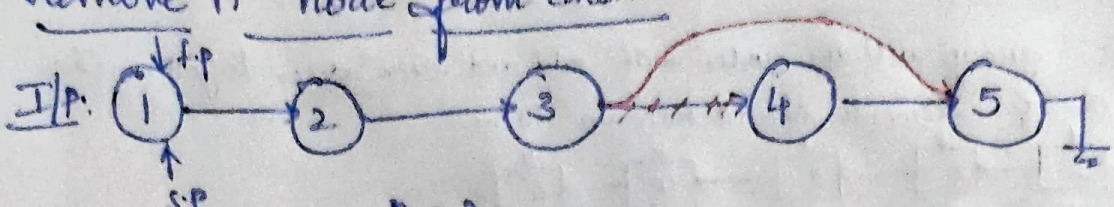
3) (2)→(3)→(6)→(7) (1)→(5)→(4)

After the loop (evenhead = (1))

(2)→(3)→(6)→(7)→(1)→(5)→(4)

Remove nth node from End:

I/P:

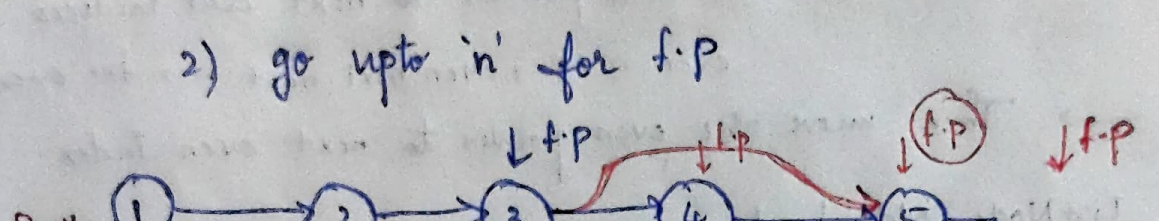


n = 2

O/P:

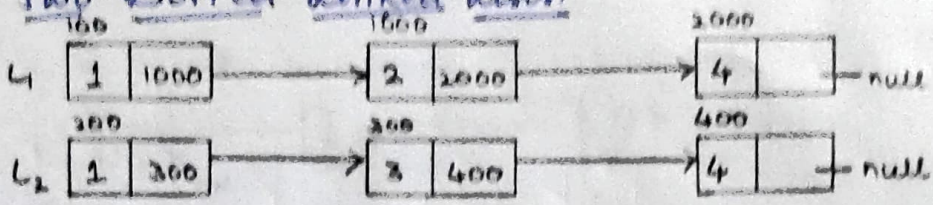
Intitution:

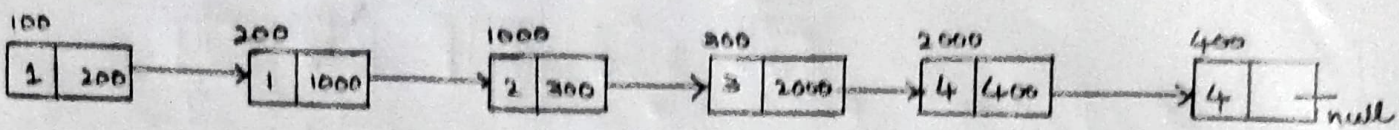
1) Calculate total length. ⇒ (5)
2) go upto total length - n ⇒ (5 - 2) = 3
3) Then break the connection.
2) go upto 'n' for f.p



3) Move f.p and s.p to next until f.p != null
4) So s.p move to (3).
5) Prev = temp

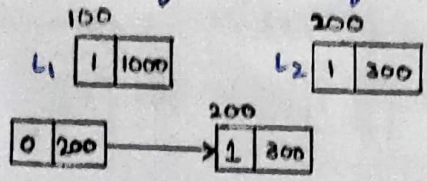
Merge Two Sorted Linked Lists:



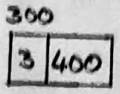


Step 1: Create a new node → ansNode

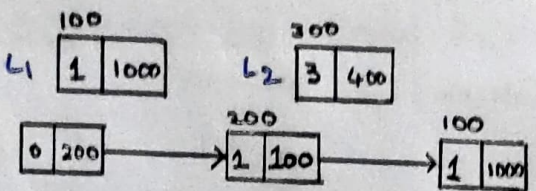
Step 2: Compare the value of Nodes of L_1 and L_2.



L_1 L_2
$1 \neq 1$

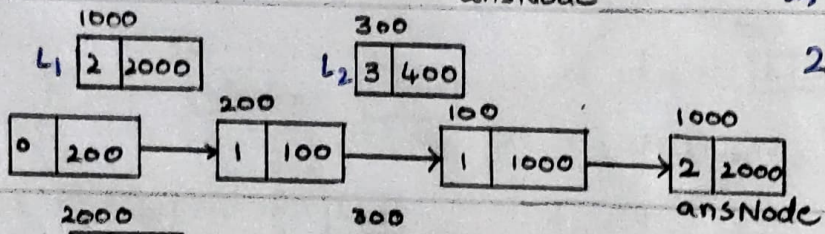


→ ans.next = L_2
→ $L_2 = L_2$.next
→ ans = ans.next



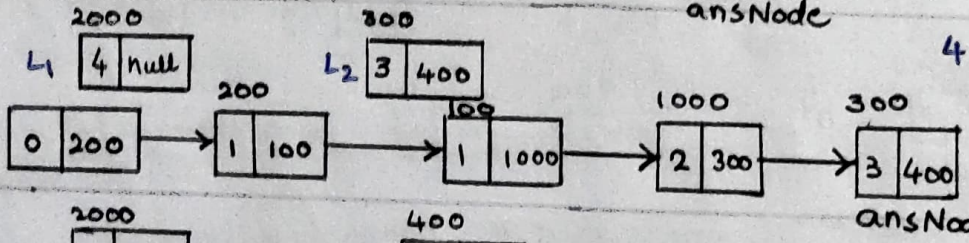
L_1 L_2
$1 < 3$

→ ans.next = L_1
→ L_1 = L_1.next
→ ans = ans.next



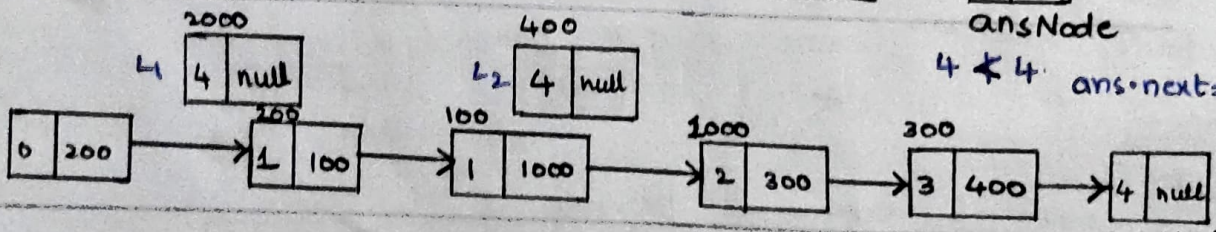
$2 < 3$

ans.next = L_1



$4 \neq 3$

ans.next = L_2

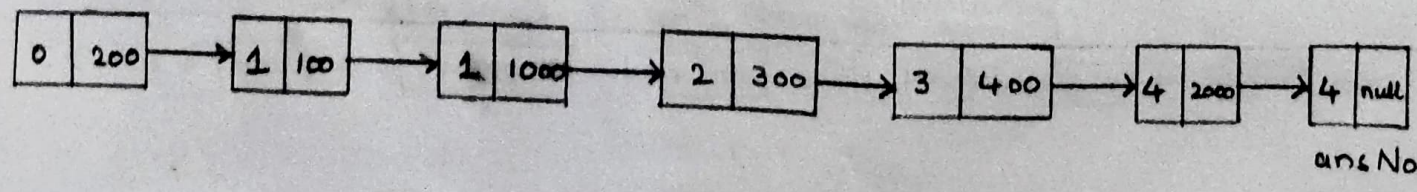


$4 \neq 4$

ans.next = L_1

$4 = null$ $L_2 = 4$ null (2000)

ans.next = L_2



```
while (fn != null && sn != null)
{
    if (fn.data < sn.data) {
        ansNode.next = fn
        ansNode = ansNode.next;
        fn = fn.next;
    }
    else {
        ansNode.next = sn;
        sn = sn.next;
        ansNode = ansNode.next;
    }
    if (fn != null) {
        ansNode.next = fn
    }
    if (sn != null) {
        ansNode.next = sn
    }
```

ReOrder List:

I/P: 1 → 2 → 3 → 4 → 5

O/P: 1 → 5 → 2 → 4 → 3

Approach:

L1: 1 → 2 → 3

L2: 5 → 4 → ?

1 → 5 → 2 → 4 → 3

1 → 2 → 3 | → 4 → 5

→ Finding the middle of Linked list.

L_1 1 → 2 → 3.

L_2 4 → 5

→ Reversing the second linked list.

L_2 5 → 4

→ Merge Two Linked Lists. L_1 and L_2.

```
{ ListNode slow = head;
  ListNode fast = head;
  while (fast != null && fast.next != null){
        slow = slow.next;
        fast = fast.next.next;
  }
  ListNode secondHalf = reverse(slow.next);
        slow.next = null;
  merge(head, secondHalf);
}

reverse(ListNode node){
ListNode prev = null, curr = node, next = null;
        while(curr != null){
         next = curr.next;
        curr.next = prev;
         prev = curr;
        curr = next;
        }
        return prev;
}
```

```
merge(ListNode L1, ListNode L2)
{
 while(L1 != null && L2 != null){
 ListNode n1 = L1.next, n2 = L2.next;
    L1.next = L2;
    if(n1 == null){
      break; }
   L2.next = n1, L1 = n1, L2 = n2;
 }
}
```