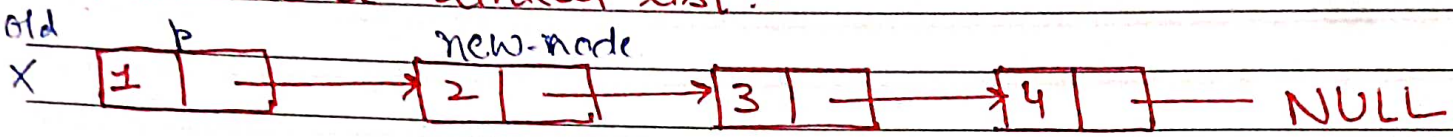
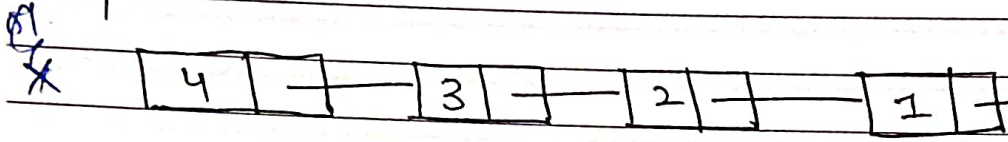
LINKED-LIST.

Reverse a linked-list :-



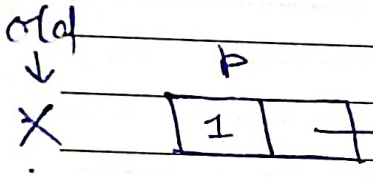
Output :



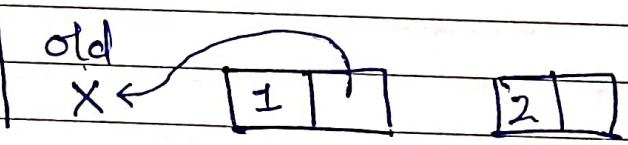
→ Use three pointers:

Set, old = Null, p = head;

new-node = p→next;

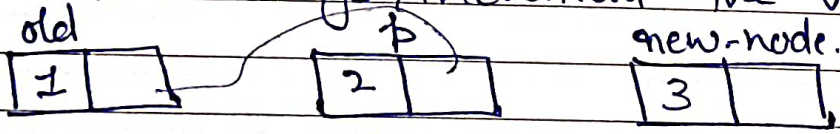


Now, p→next = old i.e



Now, Repeat till p1 = NULL.

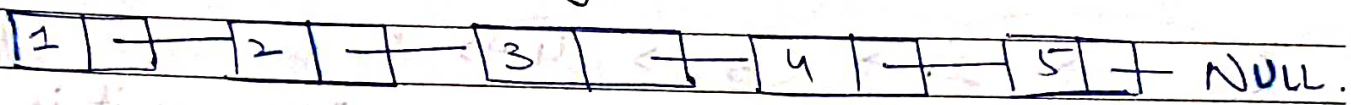
But change/increment the value of old, p, newnode.



old = p, p = new-node;

// Checkout code for better understanding.

Find Middle-note of a linked-list :-



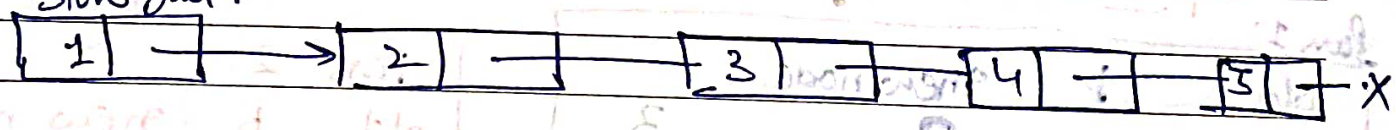
Approach 1: (Bruteforce) :→ get length of list

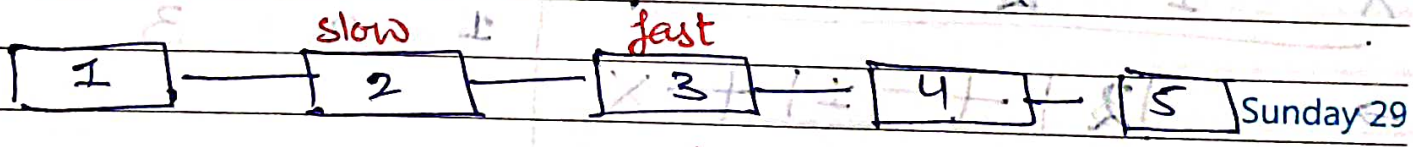
i.e 5, now (length/2) + 1 th element is the middle one

↳ for odd cases, and for even it depends on the question.

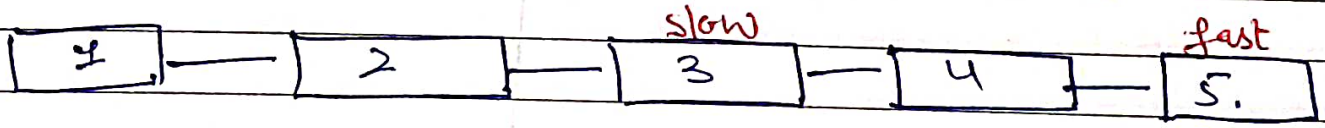
Approach two (hare and tortoise, fast & slow pointers).

In this Approach, we declare 2 pointers slow and fast, such that with each pass fast moves two times and slow moves 1 time.





Sunday 29



if ((fast || fast → next) == NULL) : return slow.

That's the Basic Approach here.

30 | MONDAY
MAY | 2-0-2-2 | Week-23 (150-215)

May 2022
S M T W T F S S M T W T F S
1 2 3 4 5 6 7
8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31

June
S M
12 13
26 27

Reverse Nodes in K-groups :-

1 → 2 → 3 → 4 → 5.

K = 2

2 → 1 → 4 → 3 → 5

Return as it is if they're not in pair.

Recursive Approach :-

1.) Solve for K - nodes.
2.) Call recursion for Solving next nodes.

Base-Case :- if (length (head) < K) return head;
// Cause we don't have to handle this kinda cases!

Solve 1 Case :- Reverse K nodes! (using basic Algo).

Pass 1:

old	p	new-node.
X	1	2 3

→ [2] → [1] → X

Pass 2 :-

old	p	new node.
1	2	3

if (new-node != NULL) → basically there is/are nodes left to Solve.
{
head → next = reverse (new-node, K);
}

Man becomes man only by his intelligence, but he is man only by his heart. – Henri Frederic Amiel

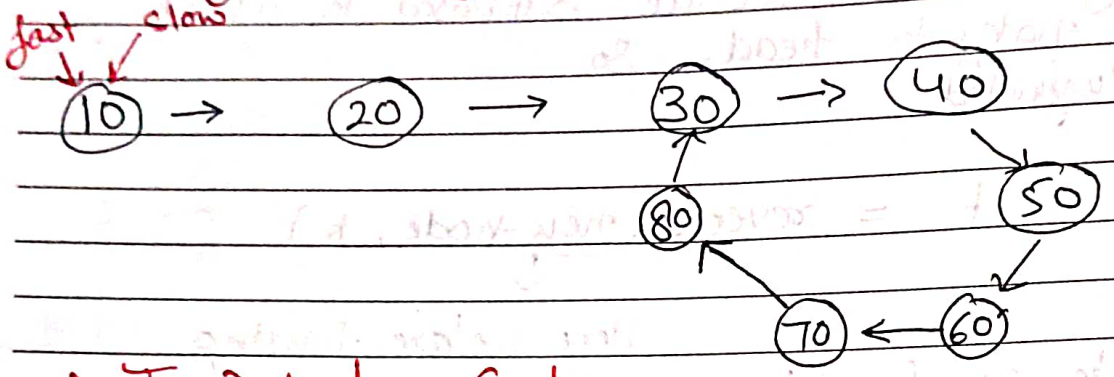
As head is going to be the last element of reversed group. so, we are supposed to attach the next nodes to head. So,
↳ unfinished

head → next = reverse (new-node, k)
↓
Here we are passing new-node as head!

Reverse a Doubly-linked list:-


```
while (p != NULL) {   new = p->next;
    p->next = old;
    p->prev = new;
    old = p;
    p = new;
}
head = old;
```

Floyd's Algorithm to Detect loop in a linked-list.



★ To Detect a Cycle:-

→ Initialize two pointers slow & fast :-

→ Move fast pointer by 2 positions and slow pointer by 1 position.

→ if both pointers meet at some point then a loop exists and if fast pointer meets the end position then no loop exists.

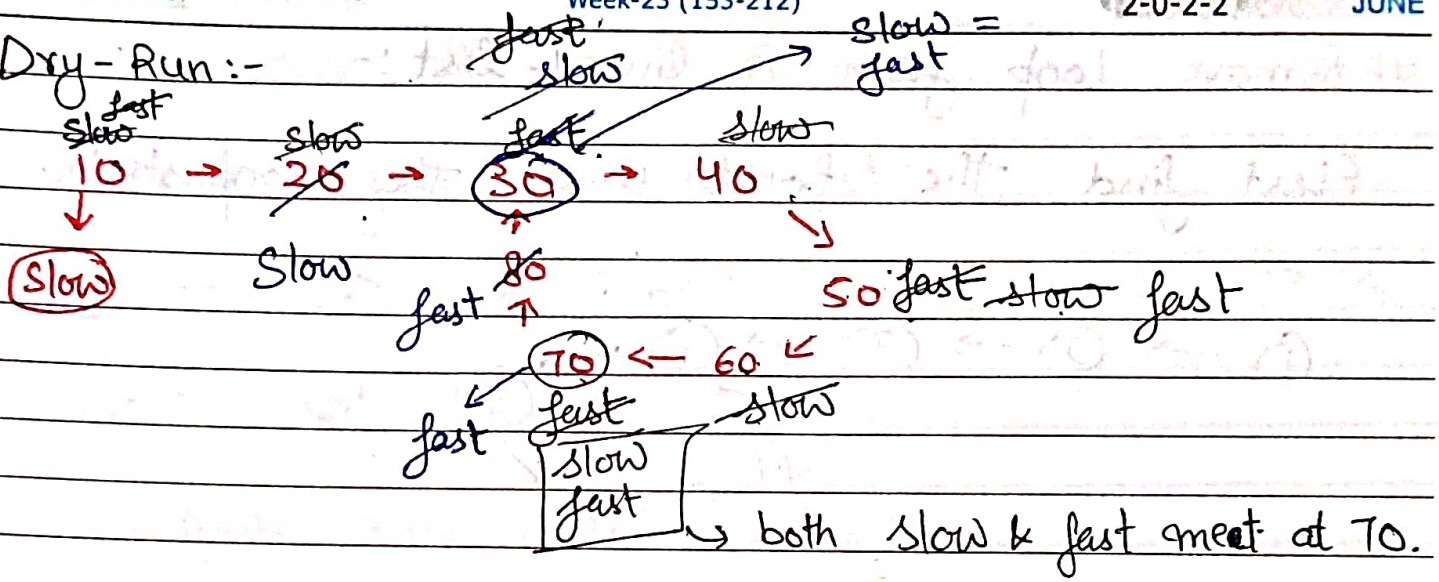
→ To check the Starting point of Cycle:

→ first do repeat the above steps and when (slow == fast).

→ Put Slow = head, and move both slow and fast pointer by 1 position.

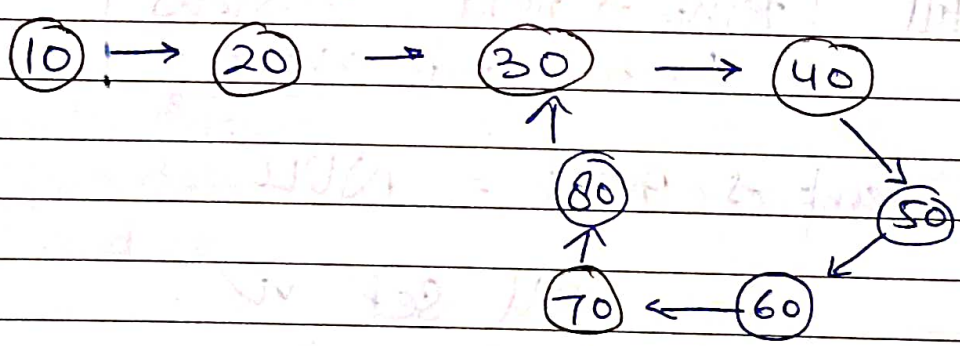
→ Now where slow and fast pointers will meet, it'll be the starting point of loop.

Dry-Run :-



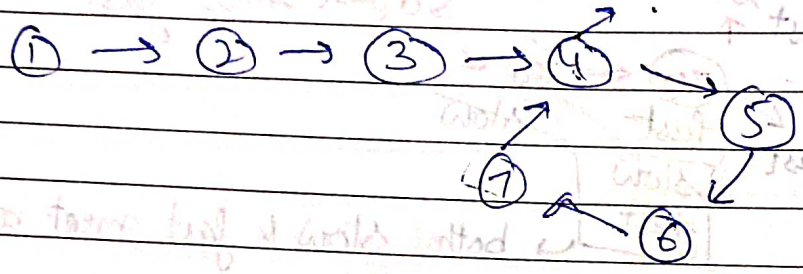
Now, Slow = head

We can see that slow equals fast at 30rd Node. Hence, we can say that the loop begins at 30.



Remove Loop from a linked-list :-

First find the point where the loop starts.



* Using floyd's algo. find the starting point of loop.

Now take a variable, let's say temp. and initialise it with 'slow → next;'

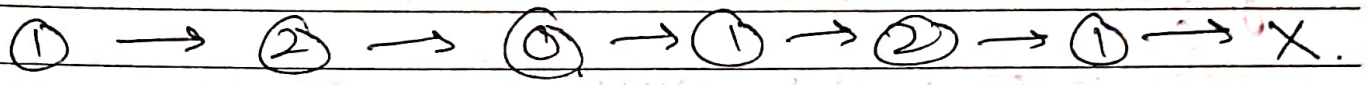
Now till `temp → next != slow`
`temp = temp → next`

Now, temp → next = NULL

All set ✓.

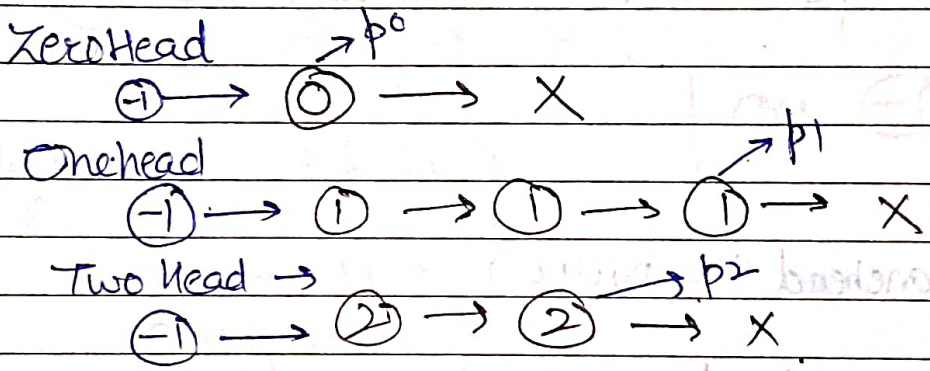
Sort 0's 1's and 2's.

Suppose, given a linked-list :-



Now sort them

(i) Make three dummy Nodes :-



Iterate through the linked-list and
if (p→data == 0) put it next to Zerohead
and so on.

Sunday 05

Now join these three ~~Nodes~~ Lists. but first remove that dummy nodes using.

```
Zhead = Zhead → next;
Onehead = Onehead → next;
Twohead = Twohead → next;
```

Now Join these three linked-lists using:-

```
if (zerohead != NULL)
{
    if (onehead != NULL)
    { p0 -> next = oneHead;
      p1 -> next = twoHead;
    }
    else p0 -> next = twohead;
}

else
{
    if (onehead != NULL)
    {
      zerohead = onehead;
      p1 -> next = twohead;
    }
    else zerohead = twohead;
}

return zerohead;
}.
```

Add two numbers represented by linked-lists.

List 1. (6) → (3) → X

List 2: (7) → X.

```
  1
  6  3
+ 7
-------
  7  0
```

Reverse both the linked lists.

Carry = 0. (3) → (6) → X

(7) → X.

ans (-1) →

```
while ( l1 && l2 )
{
    sum = l1->data + l2->data;
    carry = sum / 10;
    sum = sum % 10;
    ans->next = newnode (sum);
}
```

Main Thing Baki khud karo.

Now if l1 is not equal to NULL

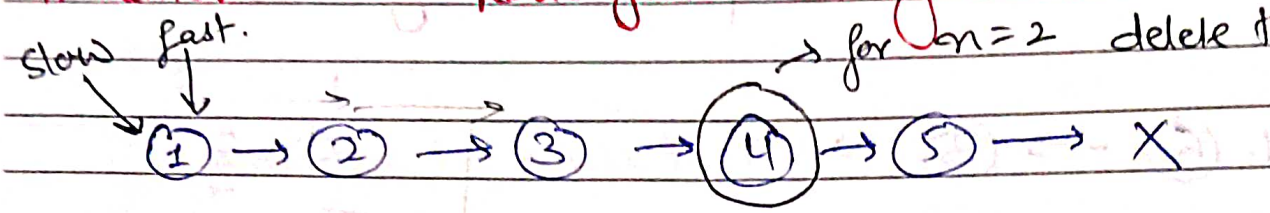
```
sum = l2->data + carry;
carry = sum / 10;
sum = sum % 10;
```

Same goes for l2.

& if still carry remains: ans->next = carry;

reverse the ans; after removing dummy node.

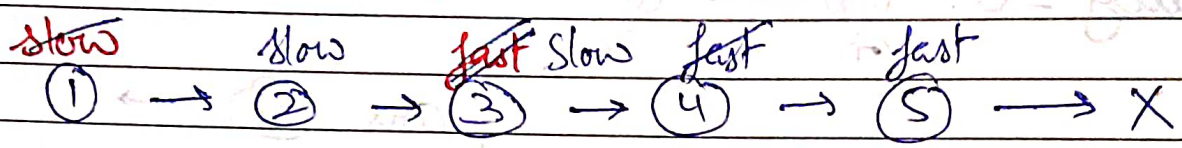
Remove Nth Node from end of linked-list :-



Put two pointers slow & fast on head

```
for (i<n)
    fast = fast -> next;
```

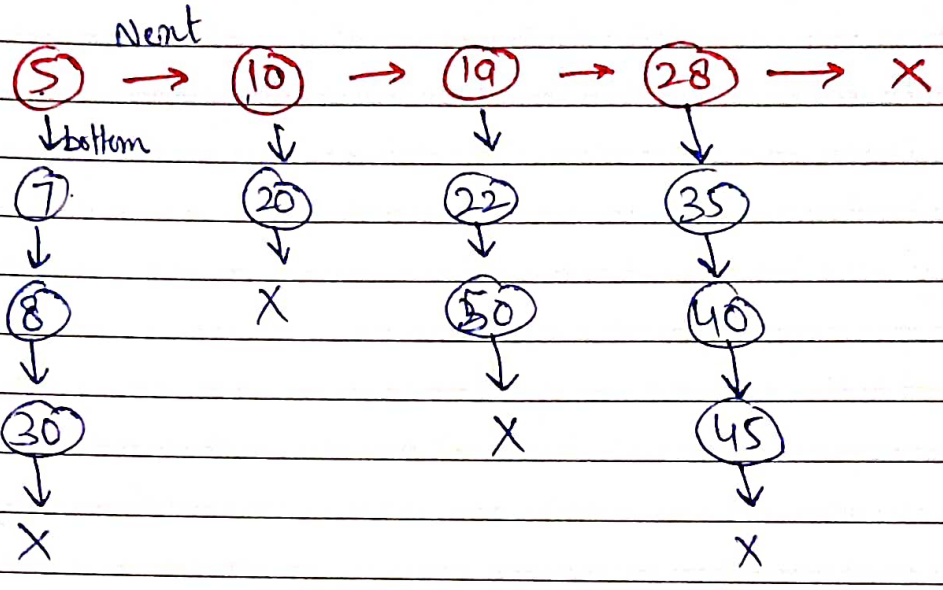
for n=2




```
while ( fast -> next )
{
    fast = fast -> next;
    slow = slow -> next;
}.

slow -> next = slow -> next -> next;
```

Flatening a linked-list:-



first use recursion to merge two nodes into one.

Take ~~two~~ one dummy node

```
ans = NULL;
```

basically we're doing this.

5 → 7 → 8 → 10 → 20 → 30 → X

```
if (a->data < b->data)
{   ans = a;
    ans->bottom = (a->bottom, b);
}
else {
    result = b;
    ans->bottom = (a, b->bottom);
}
return result;
```

Now merge from last node & merge all blah, blah.