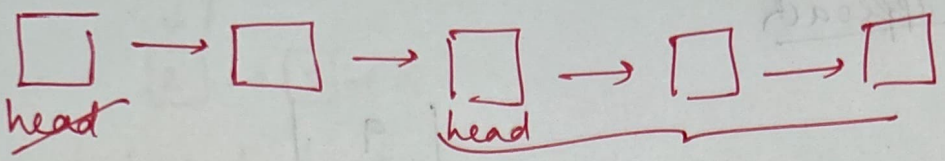
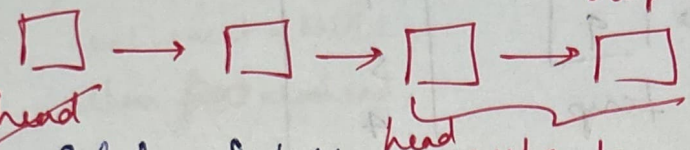
Middle of a LL

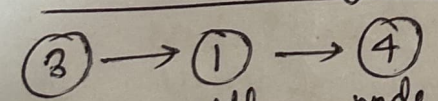




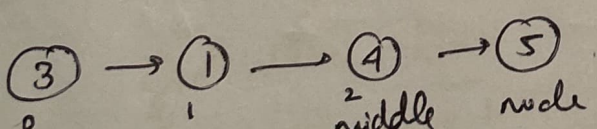

```
Class Solution { ListNode* output
  vector<ListNode*> nodes; → create a vector to store pointers to nodes
  ListNode* temp = head
  while (temp != NULL) {
      nodes.push_back(temp);
      temp = temp->next;
  }
  return nodes[nodes.size()/2];
}
};
```

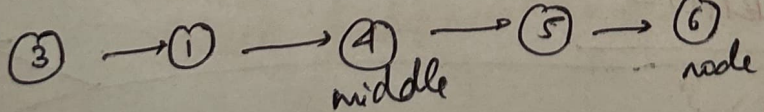
TC: O(N)
SC: O(N)

Method 2 : Using pointers (TORTOISE - HARE METHOD)



TC: O(N)
SC: O(1) → only use 2 pointers & no DS created.





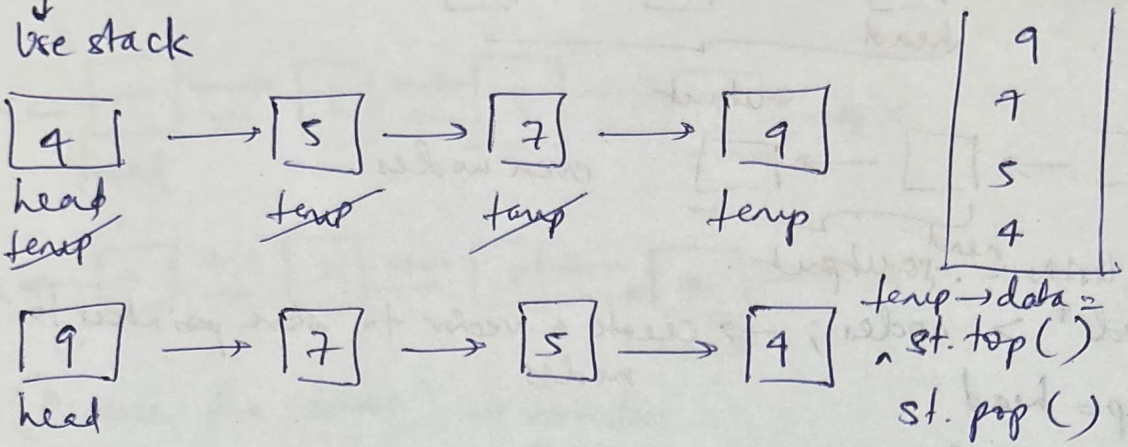
middle node
middle node

Middle increments by 1 when node moves ahead by 2 nodes. When node reaches the end of LL, middle node will be in the right place.

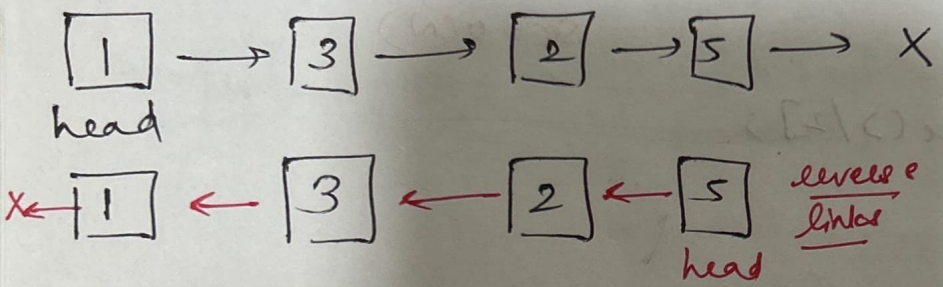
Reverse a LL → 3 Methods

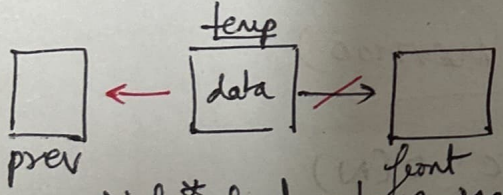
METHOD 1 : Brute force approach

↓ Use stack



METHOD 2 : Iterative approach

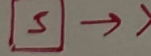





```
Node* front = temp->next
temp->next = prev;
prev = temp;
temp = front;
```

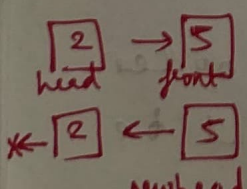
pop.

METHOD 3 : RECURSIVE METHOD

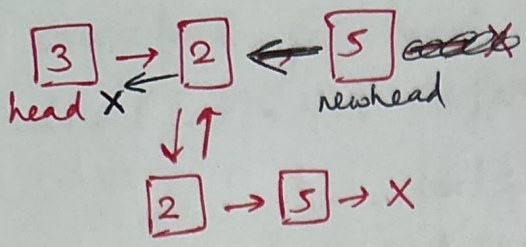


Base case

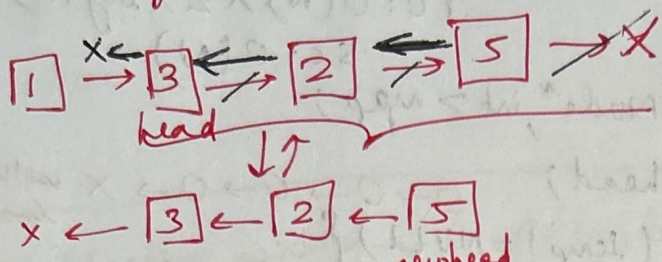
```
reverse(head) {
    if (head==NULL || head->next==NULL) {
        return head // for 0 node or 1 node
    }
```




```
    front->next = head;
    head->next = NULL;
    return front; // new head now
}
```



front → next = head ;
head → next = NULL;
return newhead;

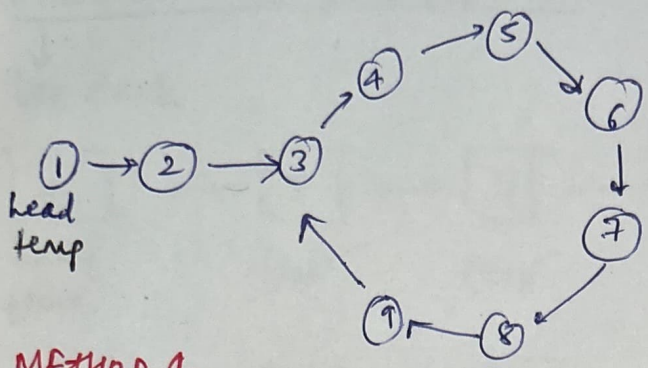


return newhead

Reverse (4)
Reverse (3)
Reverse (2)
Reverse (1)

Node * newHead = Reverse (head → next);
Node * front = head → next
front → next = head
head → next = NULL;
return newHead;

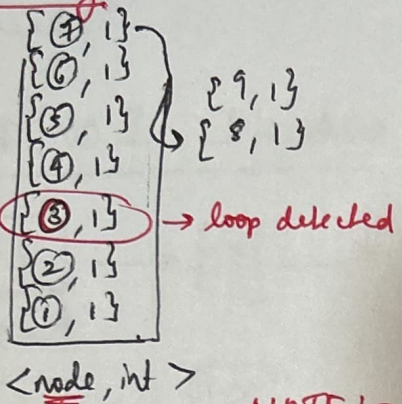
Detecting a loop in a LL



TC: O(N) × 2 × log N) ← maybe!
SC: O(N)

METHOD 1

Hashing



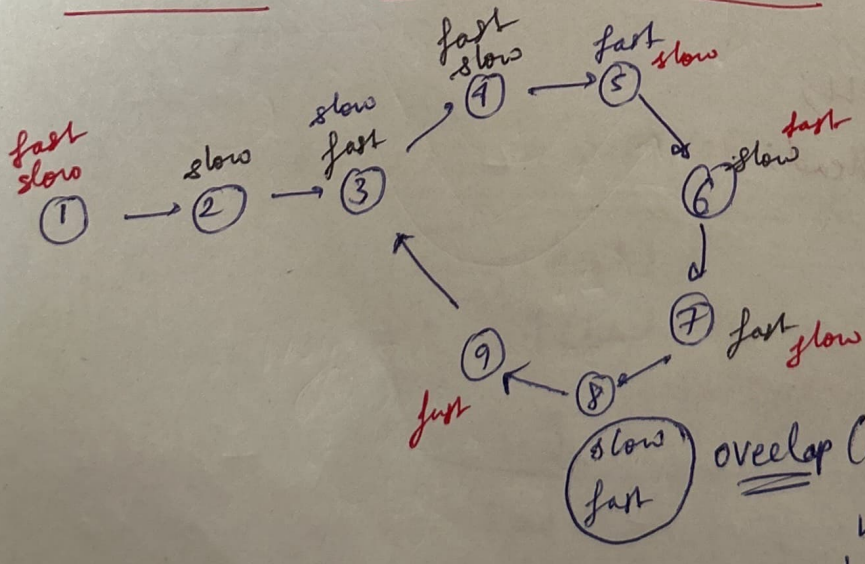

```
map<Node*, int> mpp;
temp = head;
while (temp != NULL) {
    if (mpp.find(temp) == mpp.end()) {
        either true;
    }
    else {
        mpp[temp] = 1
        mpp.put(temp, 1);
        temp = temp -> next;
    }
}
```

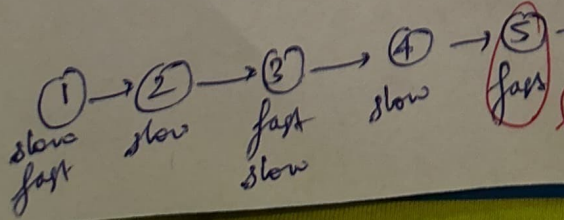
0 → 0 → 0 → 0 → X without loop.

NOTE:- store the entire node & not the values as the values might repeat in the cycle but may represent different nodes.

METHOD 2 - Tortoise & Hare algorithm

slow → 1 steps
fast → 2 steps



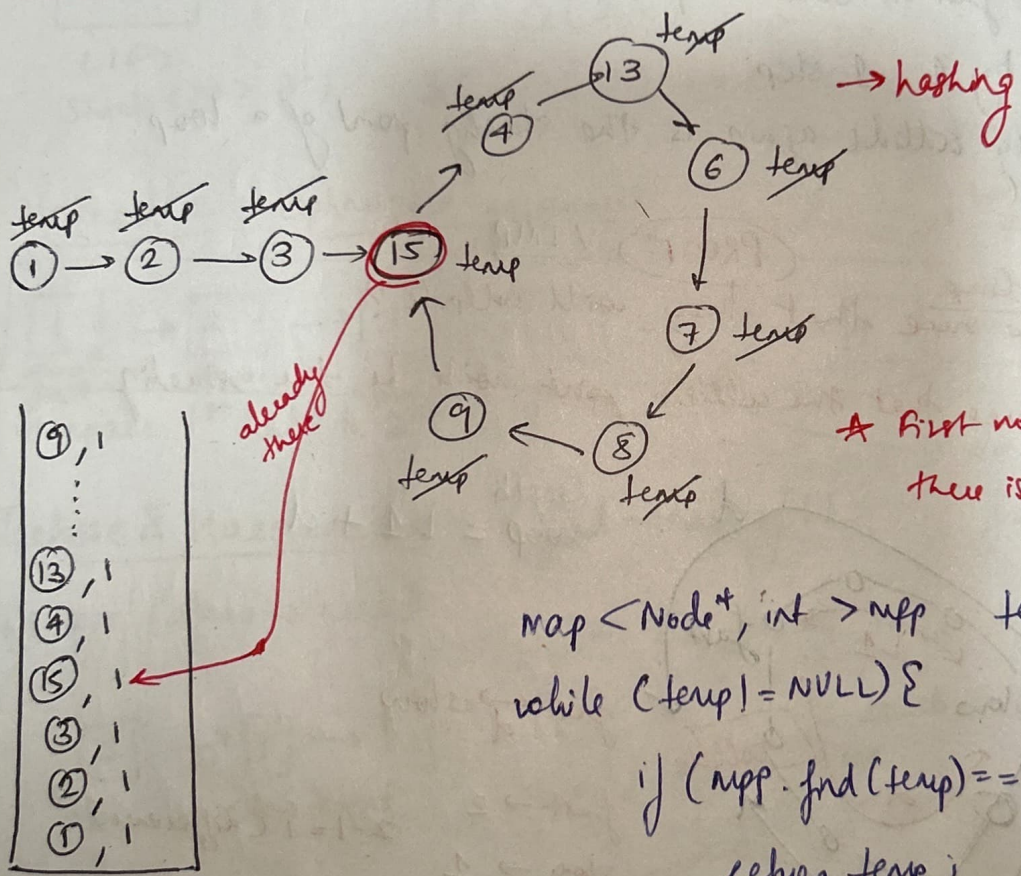


```
slow = head , fast = head
while (fast != NULL && fast -> next != NULL) {
    slow = slow -> next;
    fast = fast -> next -> next;
    if (slow == fast) return true;
}
return false;
}
```

TC ≈ O(N) depending on where you find both overlap

SC = O(1) → no extra DS used

Find starting point of a loop/cycle in LL



Node*, int

we dont store data because there may be some other node with same value.

```
map<Node*, int> mpp   temp = head
while (temp != NULL) {
    if (mpp.find(temp) == ✓) {
        return temp;
    }
    mpp[temp] = 1;
    temp = temp -> next;
}
return NULL; // linear LL → temp == NULL
```

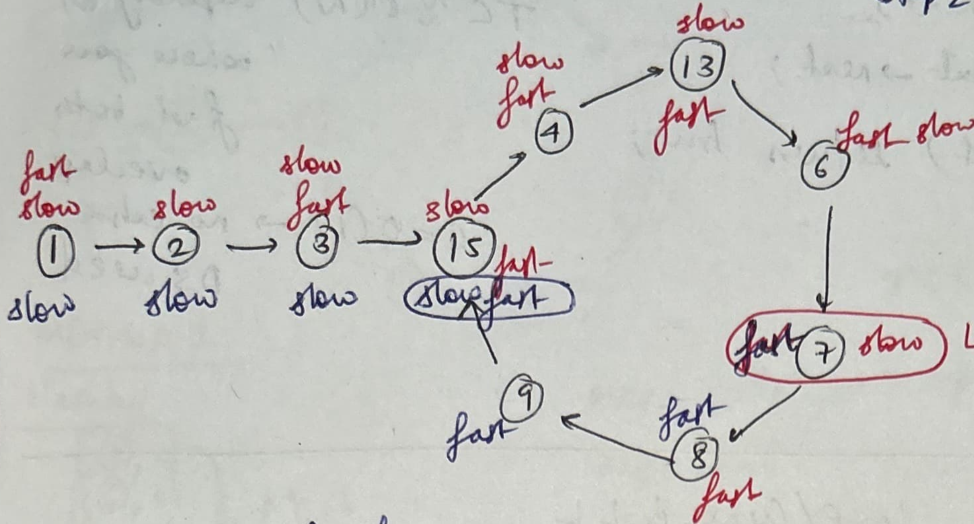
TC : ~~O(N)~~ O(N x 2x log) , SC : find func° ~~O(N x 2 x log)~~ O(N) insert func°

Method 2 - Tortoise & Hare Algorithm

slow → 1, fast → 2

Step 1: Detect a loop

Step 2: find starting point



Step 2 - explanation

slow at head, & fast at collision point.

move slow & fast by 1, 1 step

The point where they collide again is the starting point of a loop.

(Refer Code)

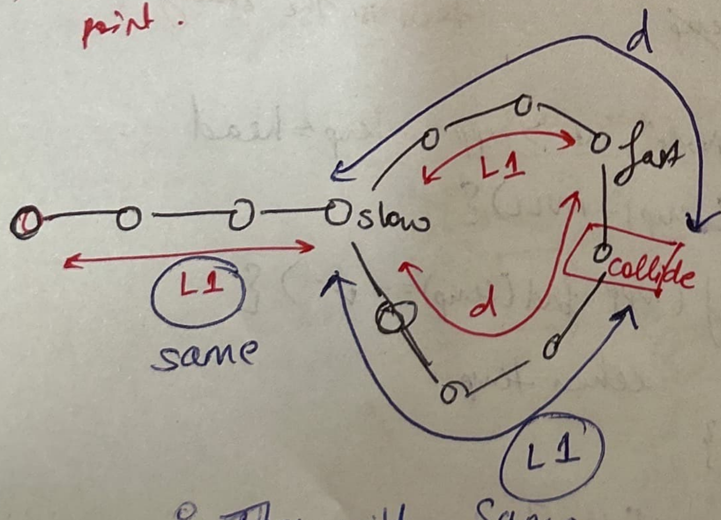
PROOF VIMP

After detection of loop

Q1) How are we so sure that they will collide?

Q2) How are we sure that the collision point will be the starting point.

length loop = L1 + d



fast ↔ slow (d)

fast → 2

slow → 1

2-1=1 (difference)

fast ⟷ slow

d-1

↓

d-2

↓

⋮

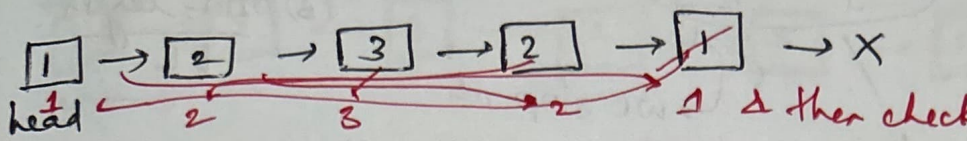
0 (collide ☐)

∴ They will collide at starting point of loop.

∴ fast travelled 2d steps

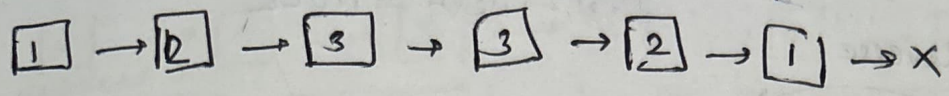
& slow travelled d steps

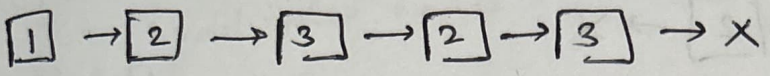
Check Palindrome in LL

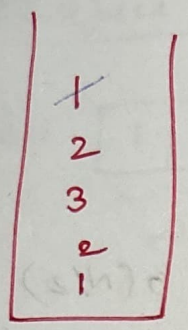


values
1 2 3 2 1 then check with original LL

if all match
then true;
else
then false;







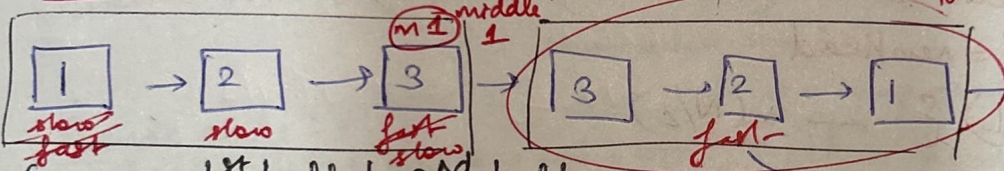
LIFO
Stack<int> st;

```
if(st.top() == temp->val){
    st.pop();
    temp = temp->next;
}
```

If all satisfy → then true;
else then false;

TC: O(2N)
SC: O(N) → all elements stored in stack

Method 2: Optimized approach



we try to reverse the 2nd half

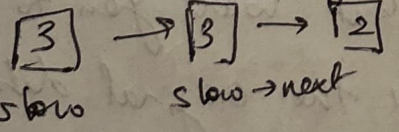
(A) Even-length LL

Compare 1st half to 2nd half

we stop at fast → next → next = NULL

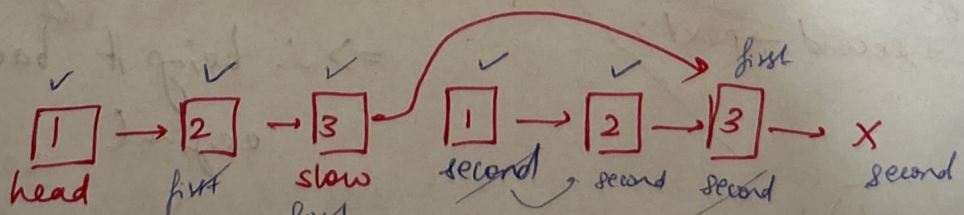
Tortoise & Hare method – to find middle of LL

reverse (slow → next)



even LL → M_1, M_2 ↑ we reach here.

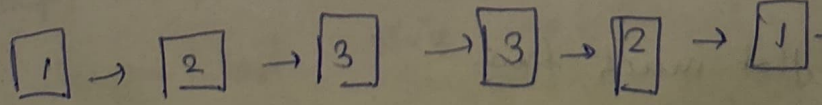
odd LL → M_1

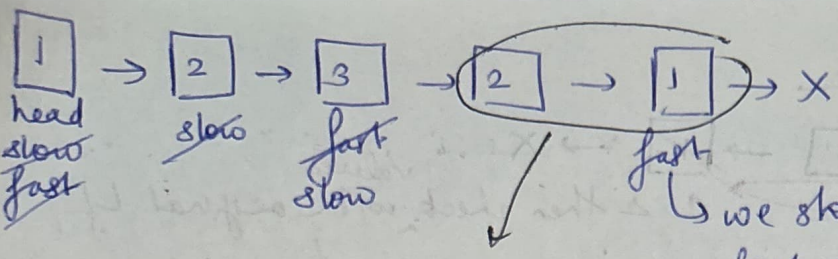


newHead returned

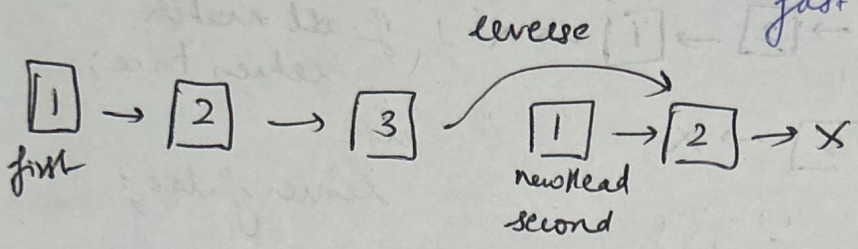
after matching & checking → pls reverse back the LL

reverse (newHead);





(B) ODD-length LL



Pseudocode :-

```
slow = head  fast = head
while (fast → next != NULL && fast → next → next != NULL) {
    slow = slow → next;
    fast = fast → next → next;
}
```
} O(N/2)

```
Node* newHead = reverse (slow → next);
```
} O(N/2)

```
first = head    second = newHead
while (second != NULL) {
    if (first → data != second → data) {
        reverse (newHead);
        return false;
    }
    else {
        first = first → next;
        second = second → next;
    }
}
reverse (newHead);
return true;
```

→ O(N/2)

→ O(N/2)

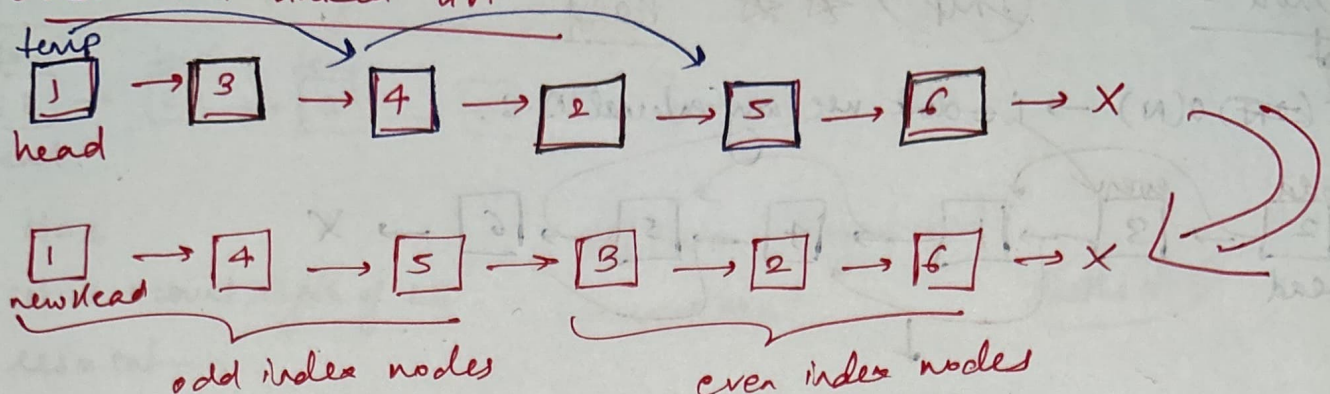
very imp in interviews as they dont want you to alter the original data.
=> ∴ bring it back to the original.

TC : O(N/2 + N/2 + N/2 + N/2)
= O(2N)

SC : O(1)

This method better than stack as it wouldnt take that much time.

Odd Even Linked list



Method 1:-
Brute force method – data replacement.
temp = head
temp jump by 2 places & store value in list till

list | 1 | 4 | 5 | 3 | 2 | 6 | temp == NULL. TC: O(N/2)

odd even

Now for even indexed nodes

temp = head → next ;

& continue putting data in list till temp → next || temp == NULL

arr[] ⇒ | 1 | 4 | 5 | 3 | 2 | 6 |

Create over LL & insert the data from arr

temp → data = arr[i]

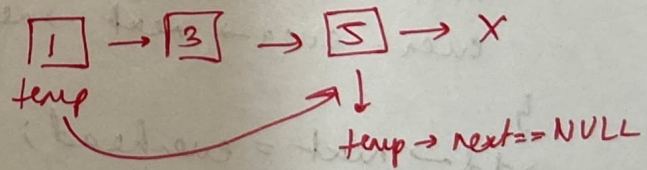
temp = temp → next

Pseudo code:-

```
if (head == NULL || head -> next == NULL) return head; // single element or empty LL.
arr[] temp = head;
while (temp != NULL && temp -> next != NULL)
  { arr.push_back (temp -> data);
    temp = temp -> next -> next;
  }
```

TC: O(N/2)

→ but we miss one case here




```
  if (temp == true) { //last node
      arr.push_back (temp -> data);
  }
```

No array

N/2 + N/2 + N

```
temp = head -> next ;
while (temp != NULL && temp -> next != NULL) {
    arr.push_back (data);
    temp = temp -> next -> next;
}
if (temp) arr.push_back (temp -> data);
```

O(N/2)

TC: O(2N)

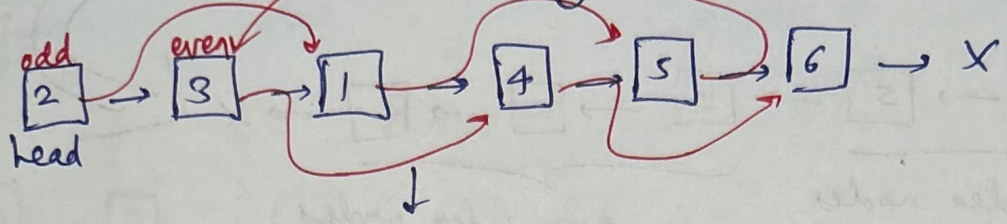
SC: O(N) array we created

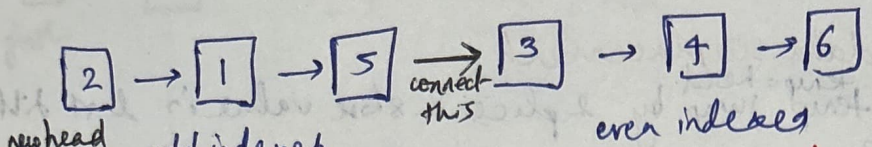
```
i = 0   temp = head
while (temp != NULL) {
    temp -> data = arr[i];
    i++; temp = temp -> next; }
return head;
```

O(N)

Method 2 (Imp) ⭐⭐ Hard

SC ~~(O(1))~~ O(N) → i.e dont use any external D.S.





evenHead = head → next // save so that it can be used later
odd → next = odd → next → next;
even → next = even → next → next;
move even & odd ahead by one node
odd → next = evenhead;
return newhead;

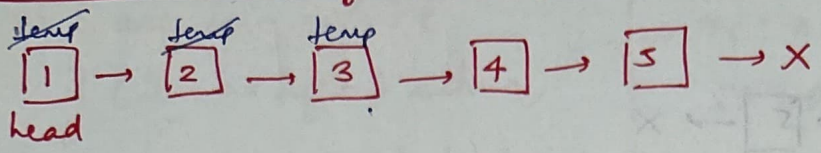
⭐ even will always be ahead of odd.

Pseudo code :- if (head == NULL || head → next == NULL) return head;
odd = head, even = head → next; evenhead = head → next;
while (even != NULL && even → next != NULL) O(N/2)
{
odd → next = odd → next → next;
even → next = even → next → next;
odd = odd → next
even = even → next → next;
} 2 steps
odd → next = evenhead;
return newhead;

TC: O(N/2) × 2 (steps)
≈ O(N)

SC: O(1)

Remove N^{th} Node from end of LL



N=2

cnt=0 → count length of LL

res = cnt − n

dllNode = temp → next

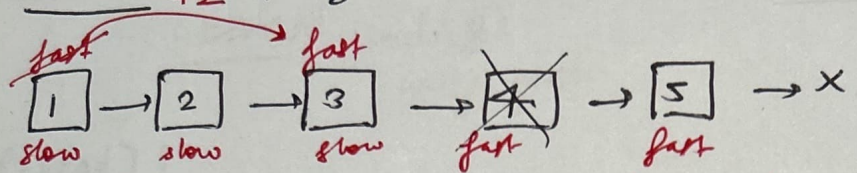
temp → next

Method 1

Brute force code - see code

Method 2 – Optimized Code

n=2



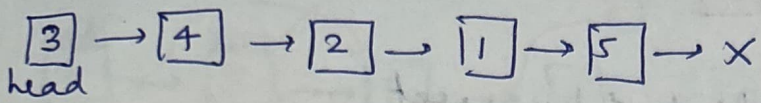
& then increment both one one step till fast → next != NULL

then slow would point to the element that is to be deleted ke peeche.

```
Node* deletedNode = slow->next;
∴ slow -> next = slow -> next -> next;
  delete deletedNode;
  return head;
```

Sort a LL

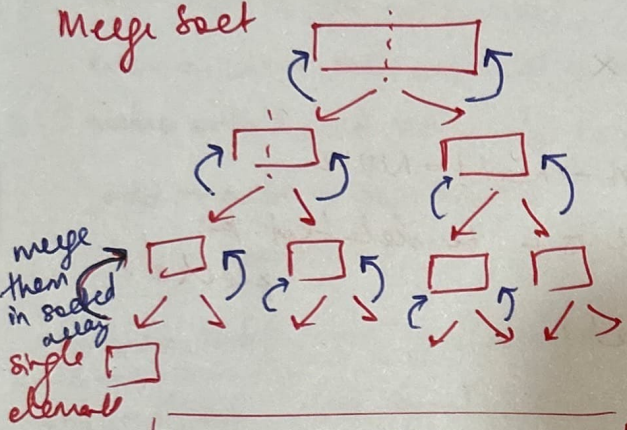
Method 1 - Brute force method

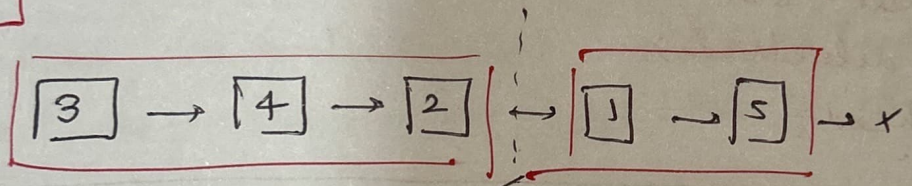


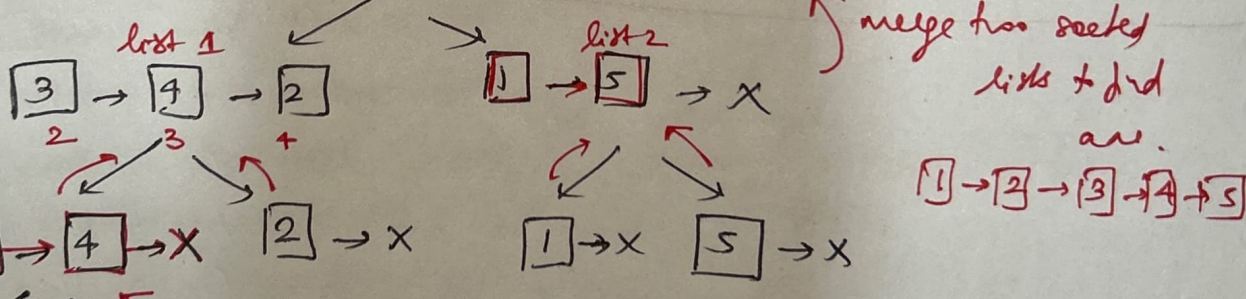
- create an array & store all the values of node.
- sort(arr.begin(), arr.end());
- traverse the LL again & store each value of arr[i] into temp→ data till temp!=NULL

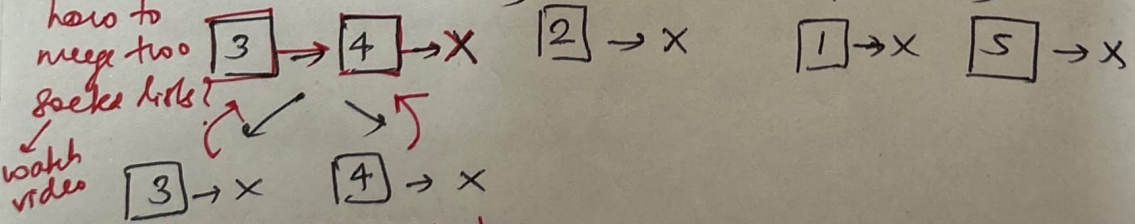
Method 2 - Optimized Code

Merge Sort









single element → ∴ stop & merge them back

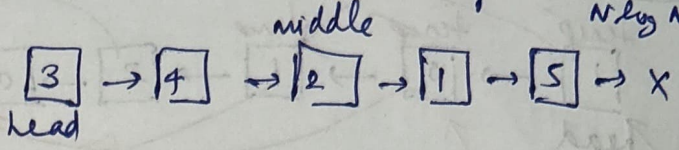
Pseudo Code :-

Pseudo Code :-

```
ms (arr, low, high) {
    if (low == high) return;
    mid = (low + high) / 2;
    ms (arr, low, mid);
    ms (arr, mid+1, high);
    merge (arr, low, mid, high);
}
```

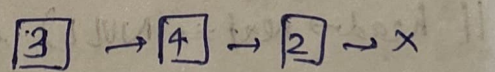
→ TC: $n \log N$ → because height of recursive tree was $\log N$ & operation was done on all N operations :. $N \log N$

SC: $O(N)$



LEARN code for merge sort

```
f(head) {
    if (head == NULL || head -> next == NVLL) return head;

    // now find middle - we Tekbise Kare
    middle = findmiddle (head);
    lefthead = head;
    righthead = middle -> next;

    // seperate both LL
    middle -> next = NVLL;

lefthead = ms (lefthead);
righthead = ms (righthead);

return merge2 (lefthead, righthead);
}
```



TC: $\log N \times (N + N/2)$

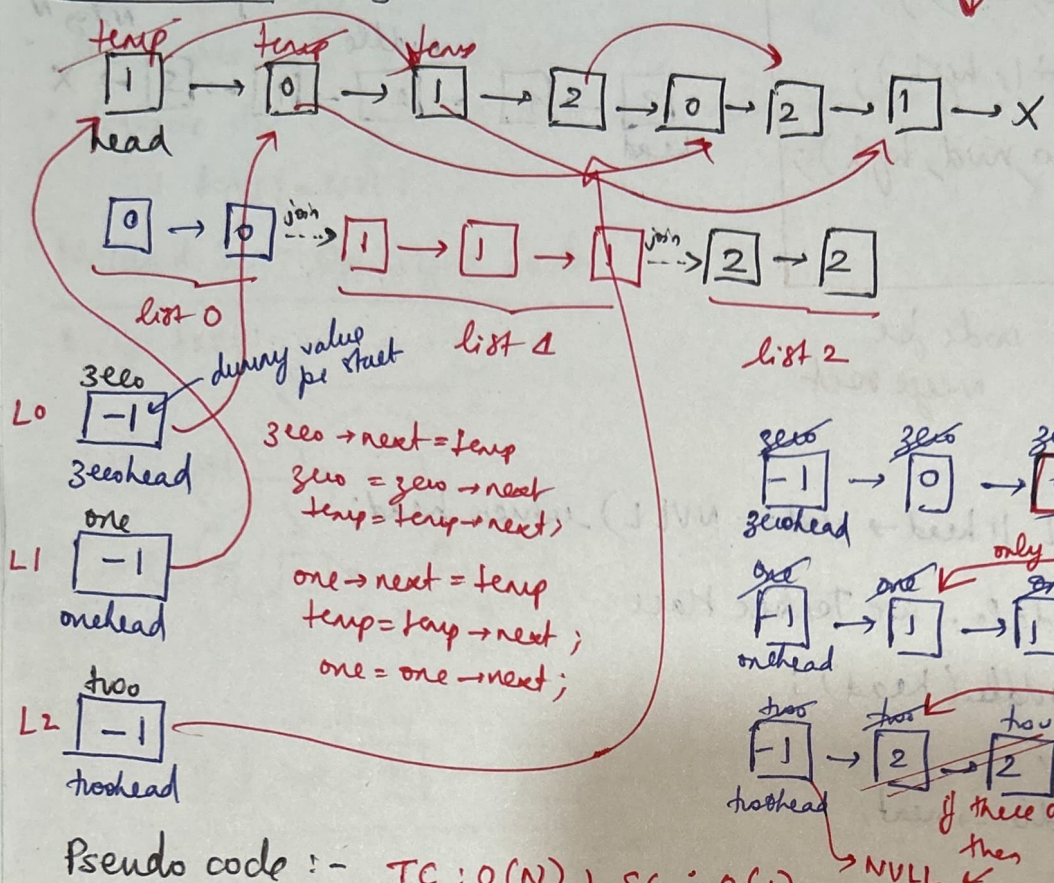
SC: $O(1)$

Sort a LL of 0's, 1's & 2's

Method 1
Method 2
} Brute force - see code

Important method

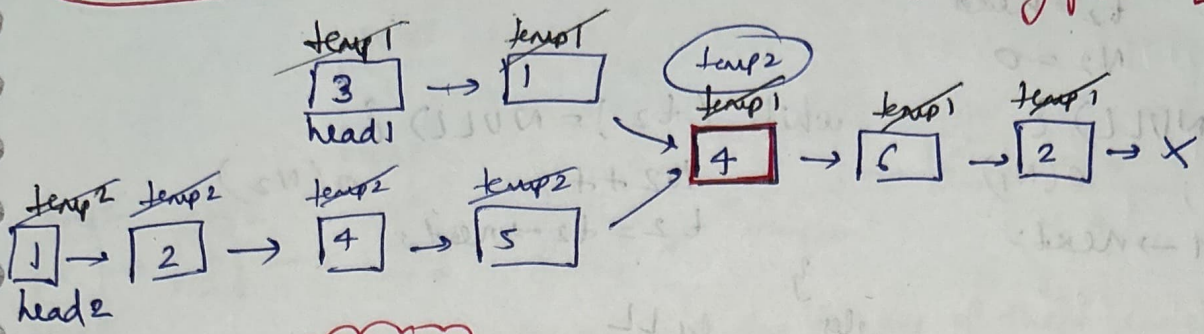
Method 3 - Optimized Code - one traversal



Pseudo code :- TC : O(N)) SC : O(1)

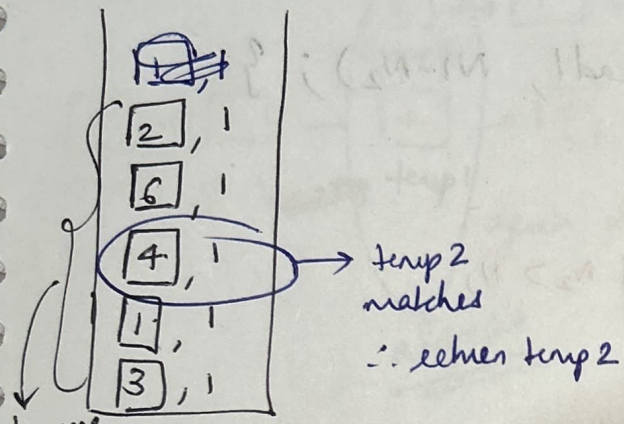
```
if (head == NULL || head->next == NULL) return head;
zeroHead = newNode(-1); , zero = zeroHead
oneHead = newNode(-1); one = oneHead
twoHead = newNode(-1); two = twoHead
temp = head
while (temp != NULL) { //O(N)
    if (temp->data == 0) {
        zero->next = temp;
        zero = zero->next; // zero = temp;
        temp = temp->next;
    }
    else if (      == 1) { }
    else if (      == 2) { }
    temp = temp->next;
}
// point 0 node end to 1 or 2 depending.
zero->next = (oneHead->next) ?
             oneHead->next : twoHead->next

one->next = twoHead->next;
two->next = NULL;
return zeroHead->next;

// delete all dummy heads
delete zeroHead
       oneHead
       twoHead
       one
       two
       zero
```

Find the intersection point of Y LL → return first intersecting point i.e 4 node.



Method 1 → Use hashing

store the node as key & int to count



Pseudo code TC: O(N1 X _) + O(N2 X _)
SC: O(N1) or O(N2) depends on LL saved in hash table

```
map<Node, int> mpp;
temp1 = head
while (temp1 != NULL) {
    mpp[temp1] = 1;
    temp = temp->next;
}
temp = head 2
while (temp != NULL) {
    if (mpp[temp].find(temp) == ✓) {
        return temp;
    }
    else {temp = temp->next;}
}
return NULL
```

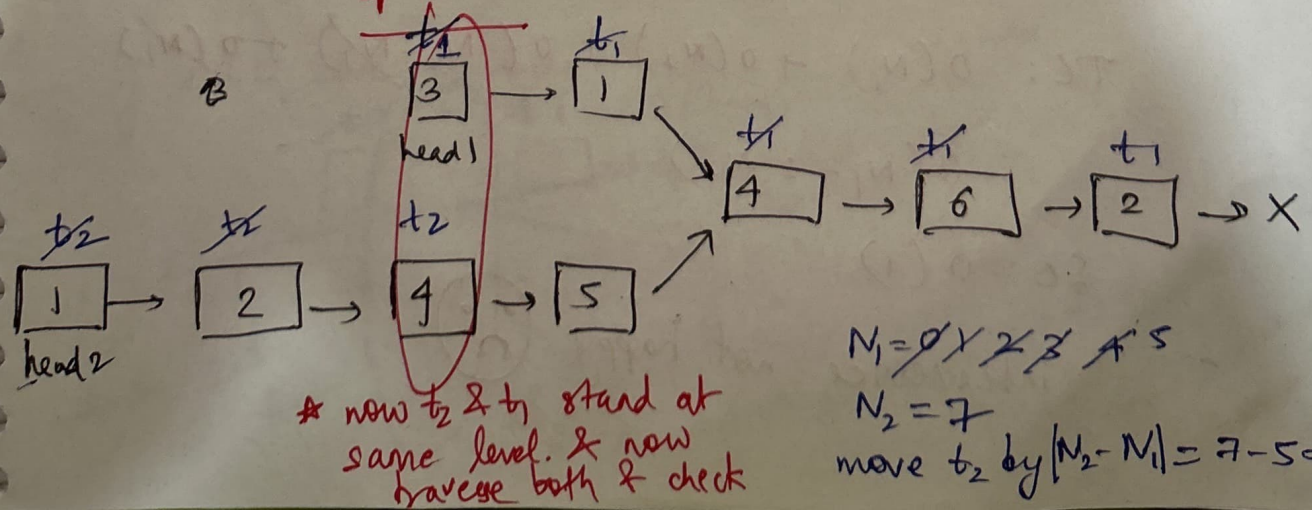
// TC: O(N1 X _)

the time map takes log or O(_) depends

depends on log & kind of map used

// TC: O(N2 X _)

Method 2 → without using external space



* now t_2 & t_1 stand at same level. & now traverse both & check

$N_1 = 5$

$N_2 = 7$

move t_2 by $|N_2 - N_1| = 7 - 5 = 2$ steps

Pseudocode

```
t1 = head        t2 = head
N1 = 0           N2 = 0
while (t1 != NULL) {
    N1++;                          O(N1)
    t1 = t1 -> next;
}
while (t2 != NULL) {
    N2++;                          O(N2)
    t2 = t2 -> next;
}
if (N1 < N2) {
    return collisionpoint (head1, head2, N2-N1);   // head1: smaller LL, head2: greater LL
}
else { return collisionpoint (head2, head1, N1-N2); }

collisionPoint (temp1, temp2, d) {
    while (d) {                    O(N2-N1) if N2 > N1
        d--;                       O(N1-N2)
        t2 = t2 -> next;
    }
    while (t1 != t2) {             O(N1) or O(N2)   shorter length
        t1 = t1 -> next;
        t2 = t2 -> next;
    }
    return t1;  // or return t2;  (same thing)
}
```

can return at colliding node or else at null if there is no collision

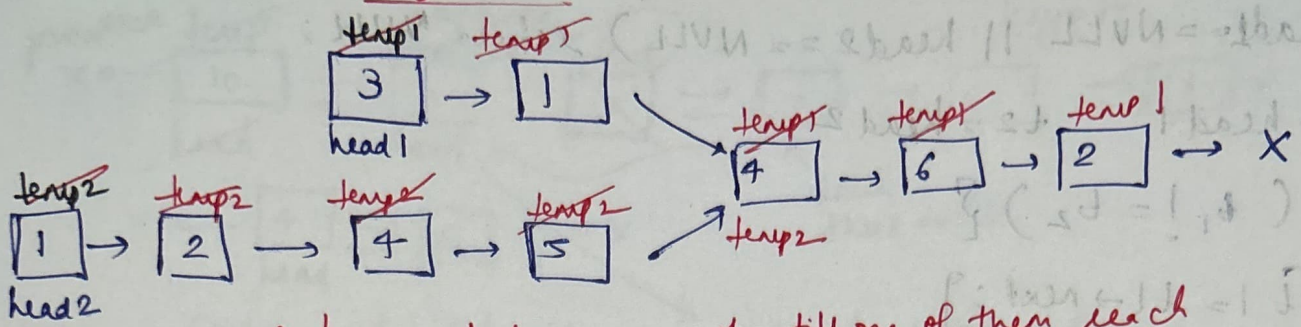
TC: $O(N_1) + O(N_2) + O(N_2 - N_1) + O(N_1)$

: $O(N_1 + 2N_2)$

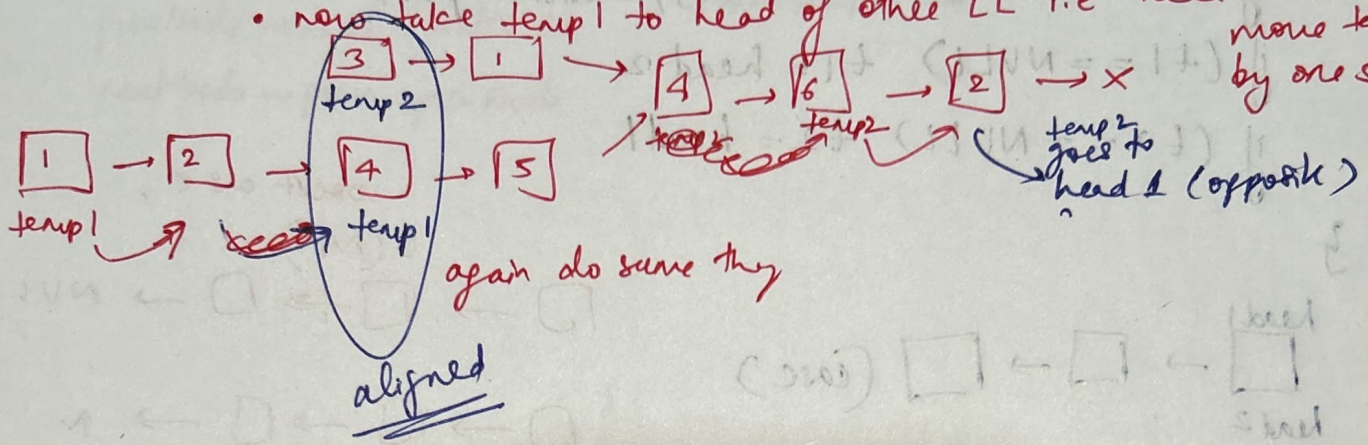
SC: $O(1)$

interviewer not happy ☹

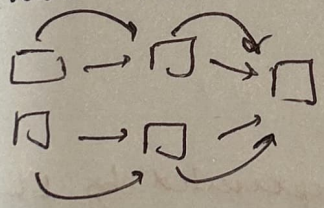
Method 3 Optimized code



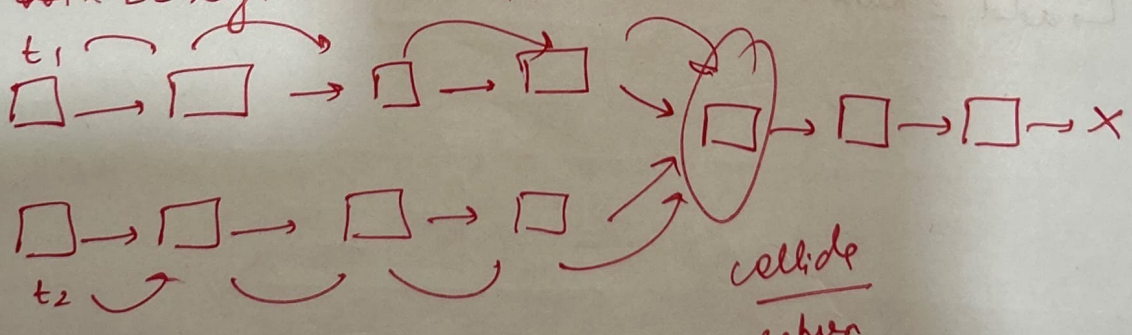
- traverse both one one step till one of them reach last node. → don't take it to null.
- now take temp1 to head of other LL i.e head 2 & move temp2 by one step



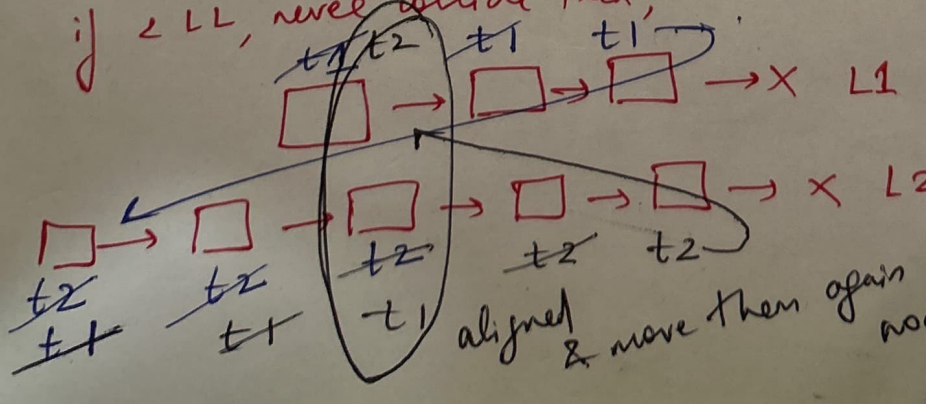
- move one-one step till both nodes are same. i.e first node where both have same node.



if both LL length are same, then



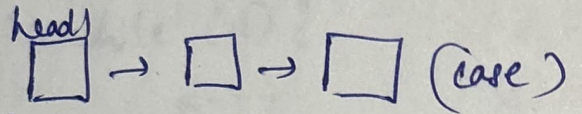
if 2 LL, never collide then,



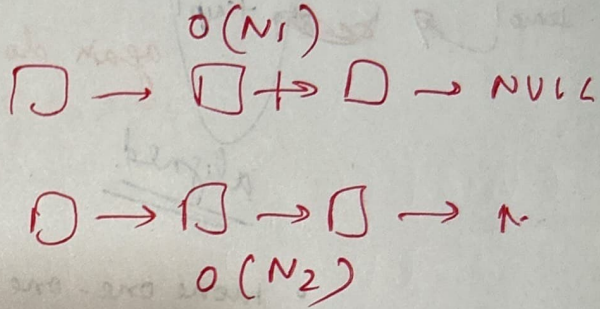
If both move one-one step then they would be NULL together

Pseudo code :- V Imp ★★

```
if (head1 == NULL || head2 == NULL) return NULL;
t1 = head1      t2 = head2
while ( t1 != t2 ) {
  { [ t1 = t1 -> next;
      t2 = t2 -> next; ]
    if (t1 == t2) return t1; // collision point
    if (t1 == NULL) t1 = head2
    if (t2 == NULL) t2 = head1
}
```



worst case.



TC: $O(N_1 + N_2)$

SC: $O(1)$

Left 0's → add 1 to a no. represented by LL

↳ add 2 numbers in LL