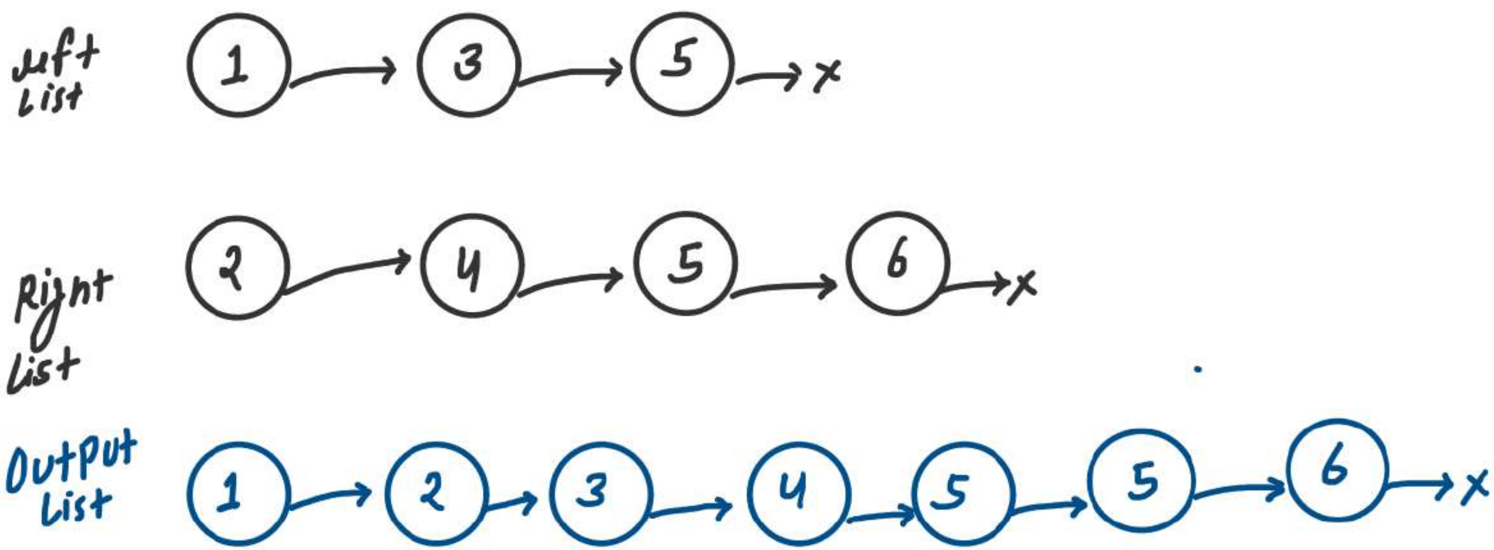
HW 01: Merge Two Sorted Lists (Leetcode-21)

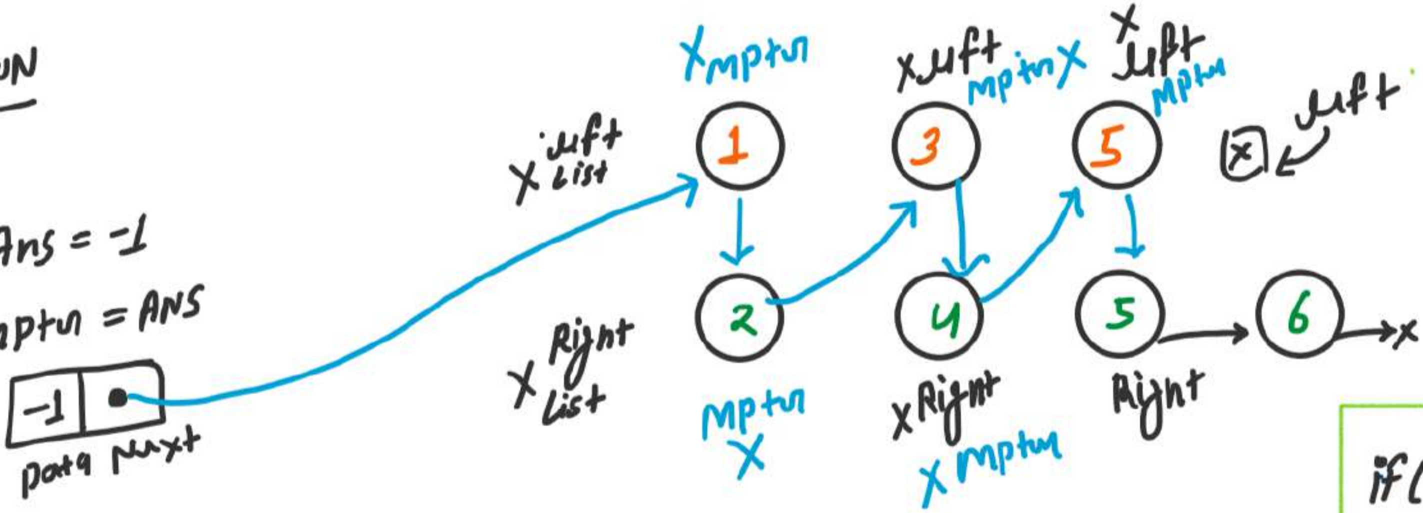
Ex

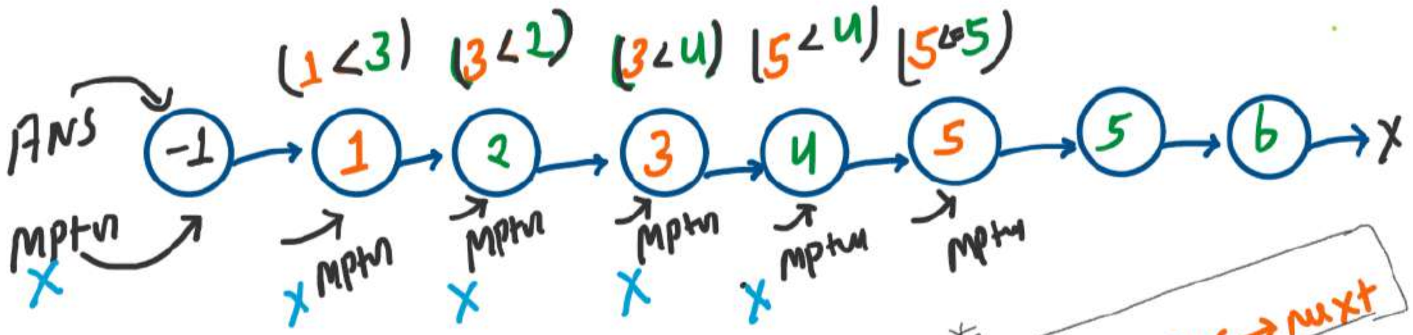


DRY RUN

let Ans = -1

mptr = ANS





** return Ans->next

```
if(left){
    mptr->next = left;
}
if(Right){
    mptr->next = Right;
}
```

```
if(left->data <= right->data)
{
    mptr->next = left;
    mptr = left;
    left = left->next;
}
Else{
    mptr->next = right;
    mptr = Right;
    Right = Right->next;
}
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* left, ListNode* right) {
        if(left == NULL) return right;
        if(right == NULL) return left;

        ListNode* ans = new ListNode(-1);
        ListNode* mptr = ans;

        while(left != NULL && right != NULL){
            if(left->val <= right->val){
                mptr->next = left;
                mptr = left;
                left = left->next;
            }
            else{
                mptr->next = right;
                mptr = right;
                right = right->next;
            }
        }

        if(left != NULL){
            mptr->next = left;
            // mptr = left;
            // left = left->next;
        }

        if(right != NULL){
            mptr->next = right;
            // mptr = right;
            // right = right->next;
        }

        return ans->next;
    }
};
```

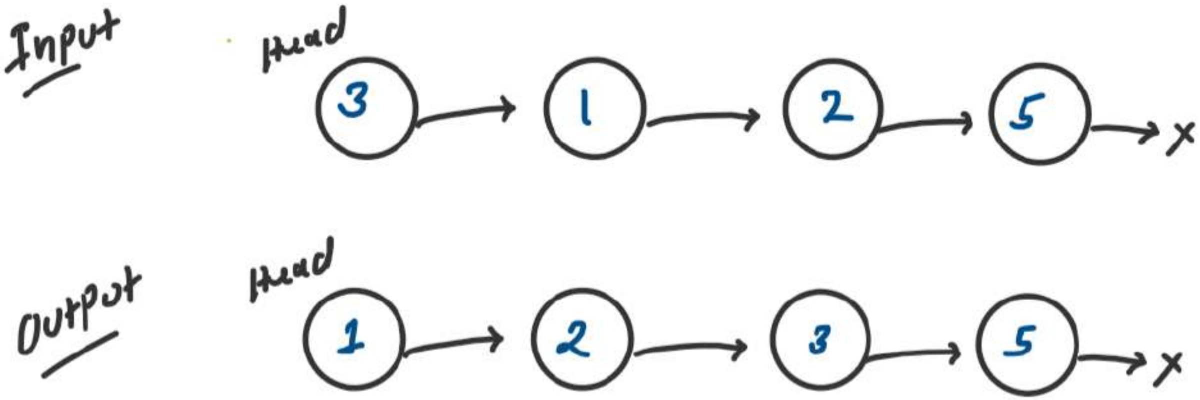
Time complexity = O(N)

where, N is total Numbers of Nodes of both list.

Space complexity = O(1)

No extra space used by ANS and mptr.

HW 02: Sort Lists using Merge Sort (Leetcode-148)



MERGE SORT ALGORITH

Step 1: *Find mid position of the list*

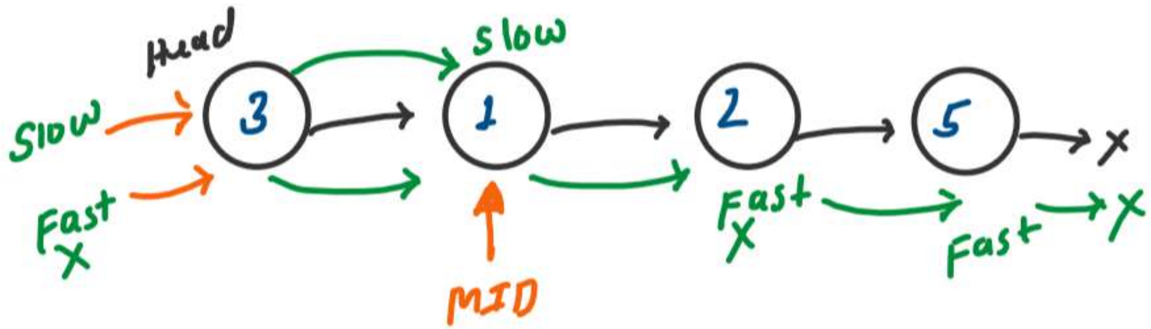
Step 2: *Divide list into two half using mid*

Step 3: *Sort RE*

Step 4: *Merge both sorted list left and right*

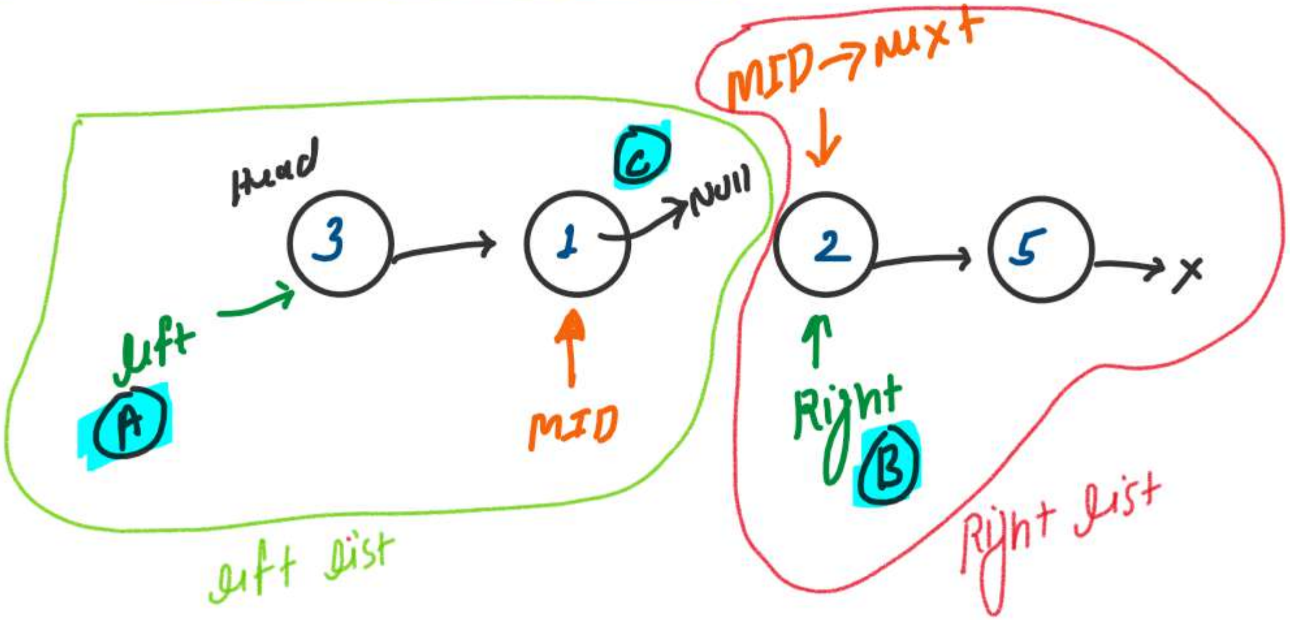
Step 1: *Find mid position of the list*

using slow & Fast Algorithm




```cpp
ListNode* getMid(ListNode* head){
    ListNode* slow = head;
    ListNode* fast = head;

    while(fast->next != NULL){
        fast = fast->next;
        if(fast->next != NULL){
            fast = fast->next;
            slow = slow->next;
        }
    }
    return slow;
}
```

Step 2: *Divide list into two half using mid*

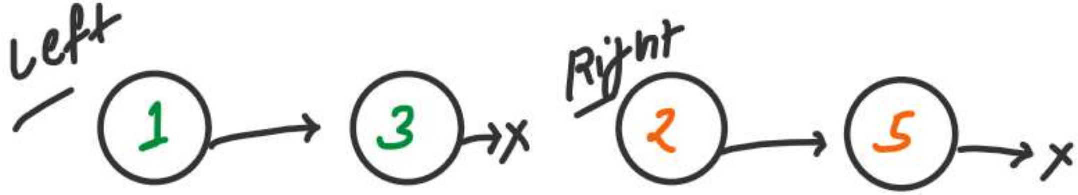


Ⓐ ListNode* left = head;

Ⓑ ListNode* Right = mid->next;

Ⓒ mid->next = NULL;

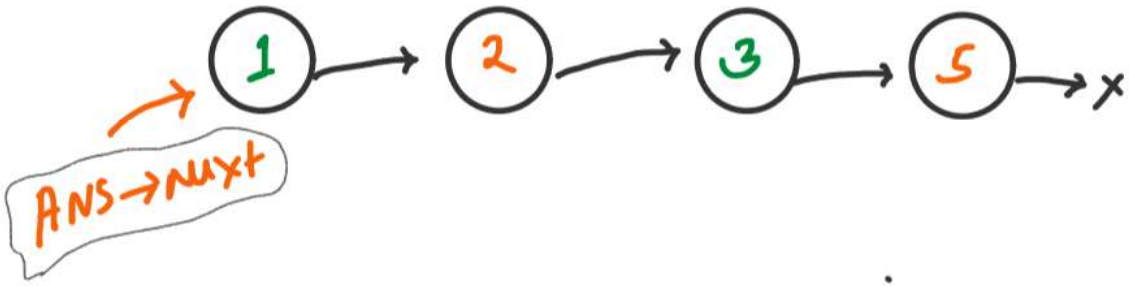
Step 3: *Sort both list left and right RE*



left = sortList(left)

Right = sortList(Right)

Step 4: *Merge both sorted list left and right*




```
ListNode* merge(ListNode* left, ListNode* right) {
    if(left == NULL) return right;
    if(right == NULL) return left;

    ListNode* ans = new ListNode(-1);
    ListNode* mptr = ans;

    while(left != NULL && right != NULL){
        if(left->val <= right->val){
            mptr->next = left;
            mptr = left;
            left = left->next;
        }
        else{
            mptr->next = right;
            mptr = right;
            right = right->next;
        }
    }

    if(left != NULL){
        mptr->next = left;
    }

    if(right != NULL){
        mptr->next = right;
    }

    return ans->next;
}
```

COMPLETE CODE

```cpp
// HW 02: Sort Lists using Merge Sort (Leetcode-148)
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* getMid(ListNode* head){...}

    ListNode* merge(ListNode* left, ListNode* right){...}

    ListNode* sortList(ListNode* head) {
        // Base case
        if(head == NULL || head->next == NULL){
            return head;
        }

        // Step 1: Find mid position of the list
        ListNode* mid = getMid(head);

        // Step 2: Divede list into two half using mid
        ListNode* left = head;
        ListNode* right = mid->next;
        mid->next = NULL;

        // Step 3: Sort RE
        left = sortList(left);
        right = sortList(right);

        // Step 4: Merge both sorted list left and right
        ListNode* mergeLR = merge(left, right);
        return mergeLR;
    }
};
```

T.C.

getmid ⇒ T.C. $= O(N)$

merge ⇒ T.C. $= O(N)$

sortlist ⇒ T.C. ⇒ $O(\log N)$

overall T.C. $= O\big((O(\text{getmid}) + O(\text{merge})) * (O(\text{sortlist}))\big)$

$= O\big((O(N) + O(N)) * (O(\log N))\big)$

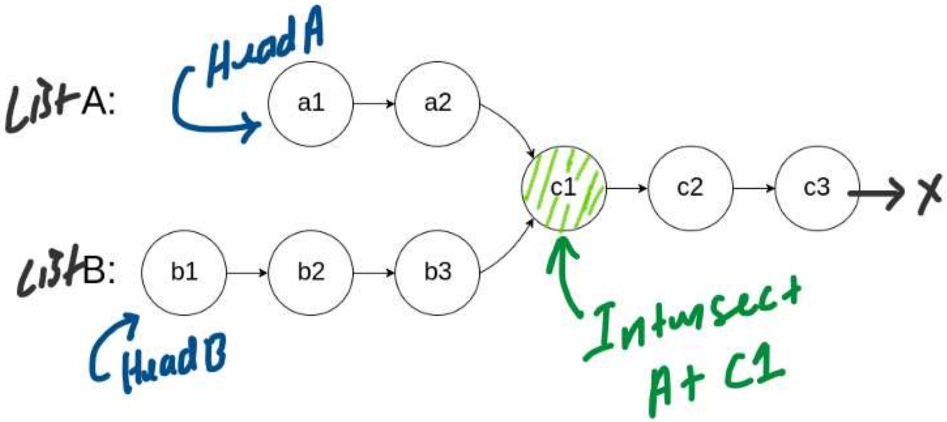
$= O\big(O(N) * O(\log N)\big)$

$= O(N \log N)$

HW 03: Intersection of Two Linked Lists (Leetcode-160)

PROBLEM STATEMENT:
*Given the heads of two singly linked-lists **headA** and **headB**, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return **null**.*

For example,
the following two linked lists begin to intersect at node c1:

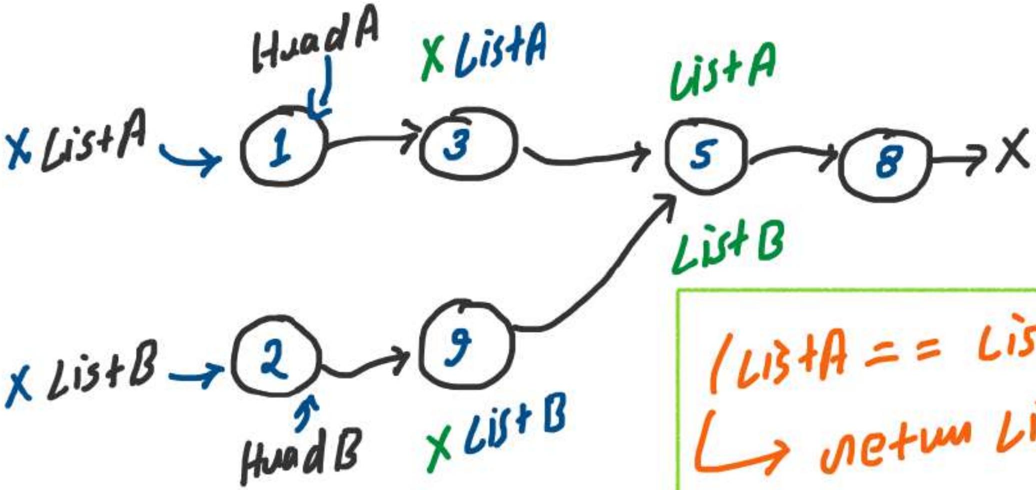


__Note__ that the linked lists must retain their original structure after the function returns.

Ex:1 DRY RUN

Equal Length of listA & listB

listALength = listBLength = 4



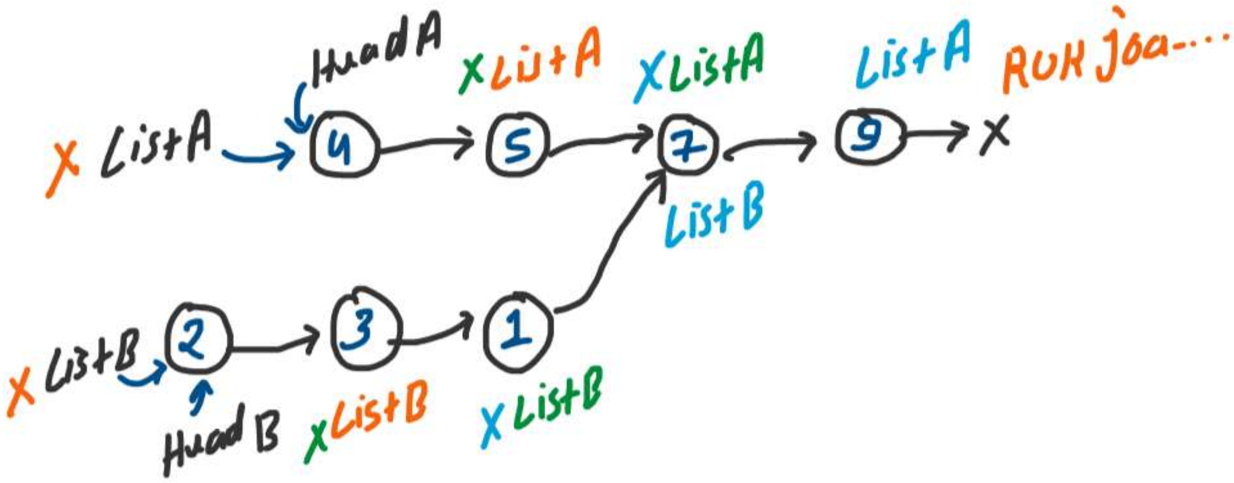
(listA == listB)
→ return listA

Output = 5

```
ListNode* listA = headA;
ListNode* listB = headB;

while(listA->next != NULL && listB->next != NULL){
    if(listA == listB){
        // Agar listA and listB equal length ki hai
        // iska mtlb wo yahin se intersect Node return kar degi
        return listA;
    }
    listA = listA->next;
    listB = listB->next;
}
```

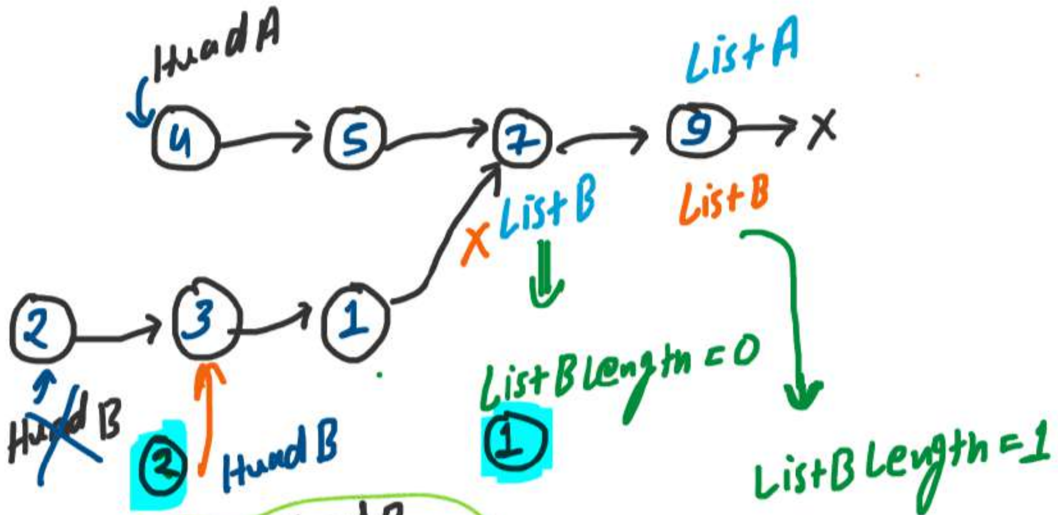
Ex:2 DRY RUN

NO Equal Length of ListA & ListB



Step-1

```
ListNode* listA = headA;
ListNode* listB = headB;

while(listA->next != NULL && listB->next != NULL){
    if(listA == listB){
        // Agar listA and listB equal length ki hai
        // iska mtlb wo yahin se intersect Node return kar degi
        return listA;
    }
    listA = listA->next;
    listB = listB->next;
}
```

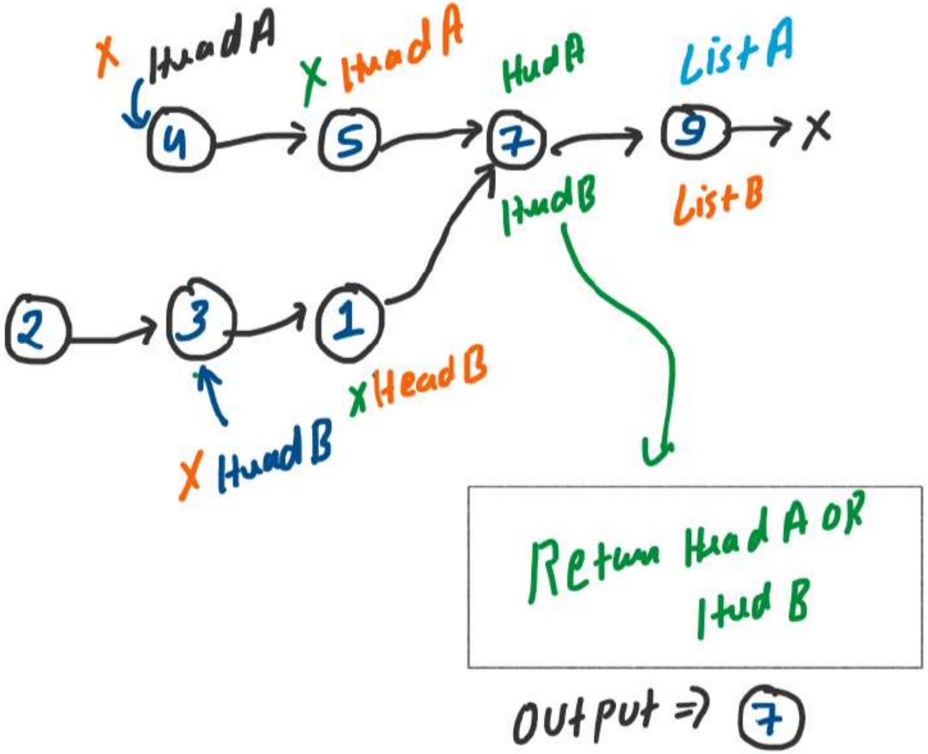


Agar HeadB ko iss 3 node par set kardi To Intersection Node mil jayega

But How

1. Find Diff b/w both list
2. Set at head at right position.

```
// Me yanha tak tabhi pahuncha hu
// jab listA and listB ki length equal nhi hai
if(listA->next == NULL){
    // First..................
    // ListB is bigger
    // We need to find the length of ListB
    int listBLength = 0;
    while(listB->next != 0){
        listBLength++;
        listB = listB->next;
    }
    // Second..................
    // In starting, Set headB at right node to get the intersection Node
    while(listBLength--){
        headB = headB->next;
    }
}
```

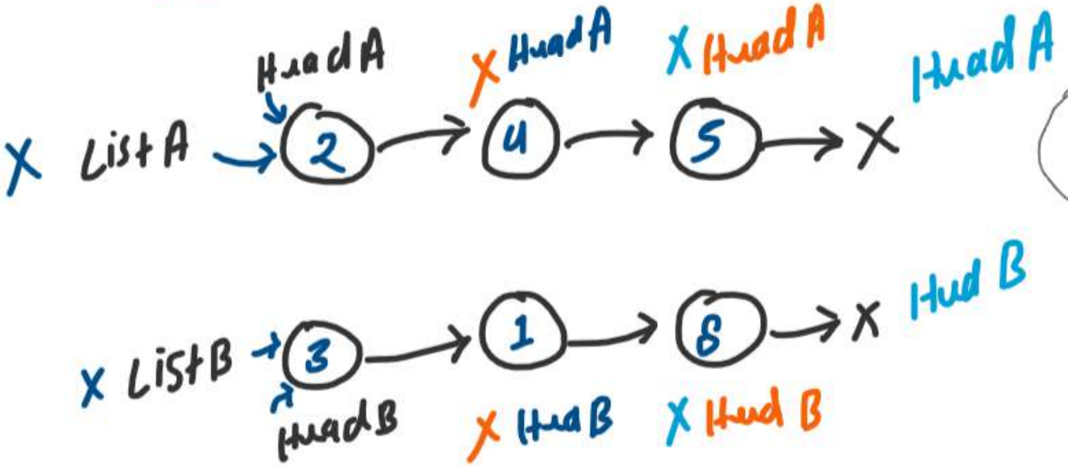



```
// Me yaha tak tabhi pahucha hu jab headA and headB starting me right
// Node par set ho chuke honge
// We need to traverse again to get the intersection Node
while(headA != headB){
    headA = headA->next;
    headB = headB->next;
}
return headA;
```

Ex: 3 DRY RUN

No Intersection b/w both lists

```
// Me yaha tak tabhi pahucha hu jab headA and headB starting me right
// Node par set ho chuke honge
// We need to traverse again to get the intersection Node
while(headA != headB){
    headA = headA->next;
    headB = headB->next;
}
return headA;
```



Return HeadA OR HeadB

Output = Null

Complete code

```cpp
// HW 03: Intersection of Two Linked Lists (Leetcode-160)
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode* listA = headA;
        ListNode* listB = headB;

        while(listA->next != NULL && listB->next != NULL){
            if(listA == listB){
                // Agar listA and listB equal length ki hai
                // iska mtlb wo yahin se intersect Node return kar degi
                return listA;
            }
            listA = listA->next;
            listB = listB->next;
        }

        // Me yanha tak tabhi pahuncha hu
        // jab listA and listB ki length equal nhi hai
        if(listA->next == NULL){...}

        if(listB->next == NULL){...}

        // Me yaha tak tabhi pahucha hu jab headA and headB starting me right
        // Node par set ho chuke honge
        // We need to traverse again to get the intersection Node
        while(headA != headB){...}
        return headA;
    }
};
```

```cpp
        // Me yanha tak tabhi pahuncha hu
        // jab listA and listB ki length equal nhi hai
        if(listA->next == NULL){
            // ListB is bigger
            // We need to find the length of ListB

            int listBLength = 0;
            while(listB->next != 0){
                listBLength++;
                listB = listB->next;
            }

            // In starting, Set headB at right node
            // to get the intersection Node
            while(listBLength--){
                headB = headB->next;
            }
        }

        if(listB->next == NULL){
            // ListA is bigger
            // We need to find the length of ListA

            int listALength = 0;
            while(listA->next != 0){
                listALength++;
                listA = listA->next;
            }

            // In starting, Set headA at right node
            // to get the intersection Node
            while(listALength--){
                headA = headA->next;
            }
        }
```

```cpp
        while(headA != headB){
            headA = headA->next;
            headB = headB->next;
        }
```

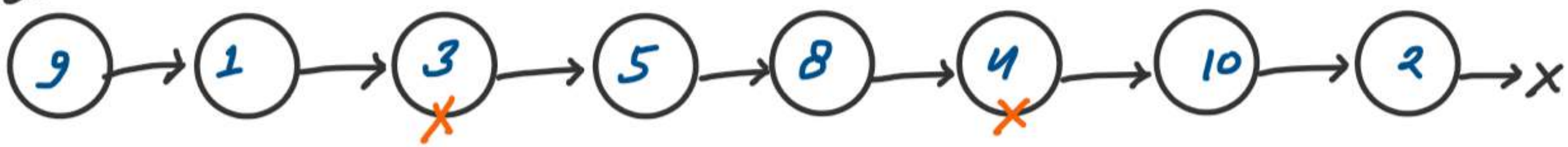
T.C. ⇒ O(N)

S.C. ⇒ O(1)

HW 04: Delete N Nodes after M Nodes (GFG)

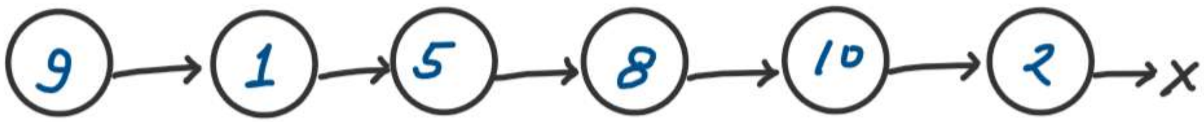
Ex 1

Head



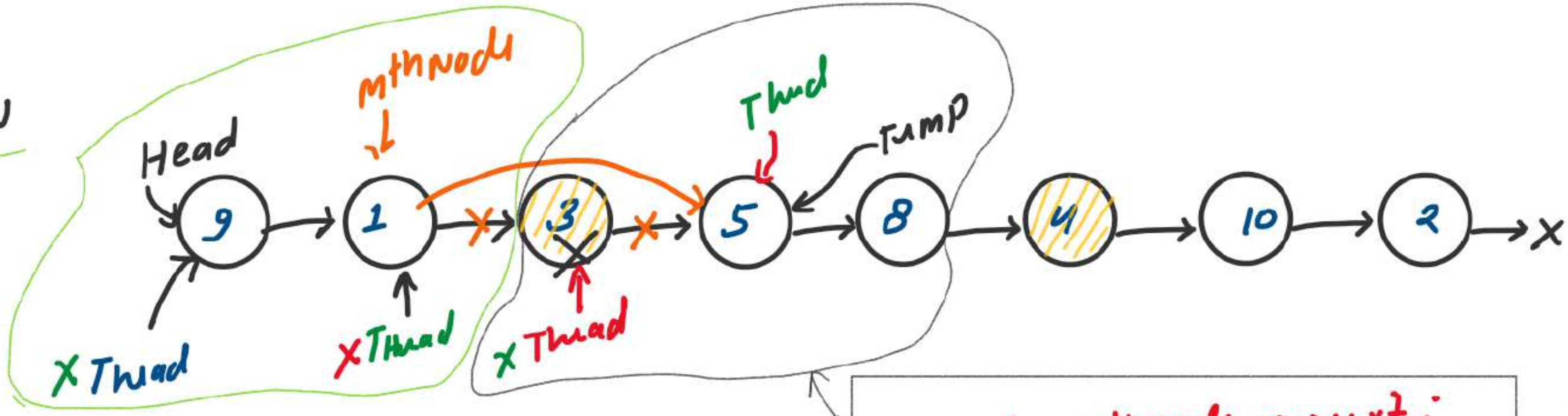
M=2 N=1

Output



DRY RUN

M=2
N=1



STEP:1

```
Node* Thead = Head;
for(int i=0; i<M-1; i++)
{
    Thead = Thead->next;
}
Node* MthNode = Thead;
```

STEP 2

```
Thead = mthNode->next;
for(int i=0; i<N; i++)
{
    if(!Thead) Break;  **
    Node* Temp = Thead->next;
    delete Thead;
    Thead = Temp;
}
mthNode->next = Thead;
```

RUN TIME ERROR

Nth Node Available nahi Hai

STEP3

fun(Thead, M, N) recursion call

BASE case

if(!head)
return

Ex2

Head

9 → 1 → 3 → 5 → 8 → 4 → 10 → 2 → X

$M=3$ $N=1$

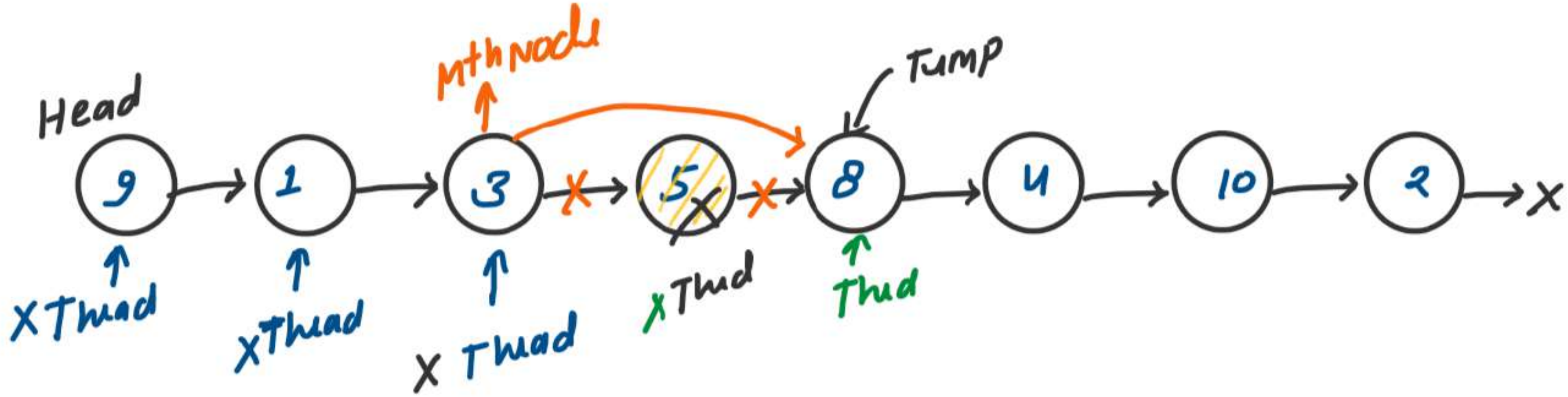
Output

9 → 1 → 3 → 8 → 4 → 10 → X

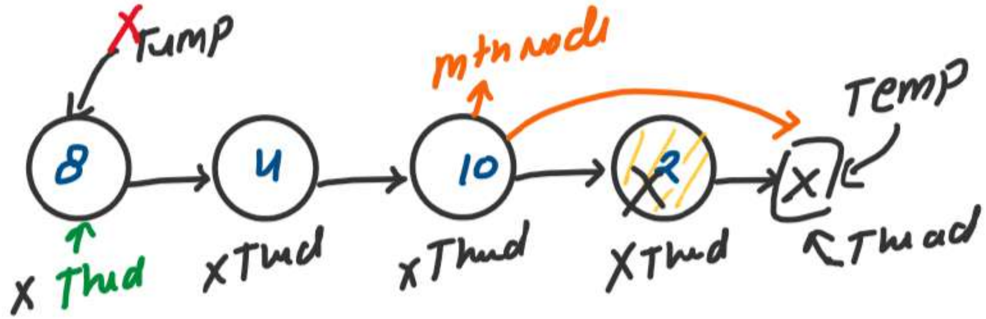
DRY RUN

M=3
N=1

Iteration 1

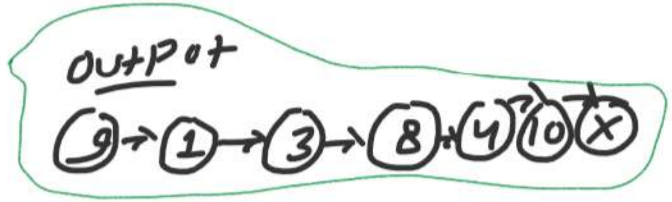


Iteration 2

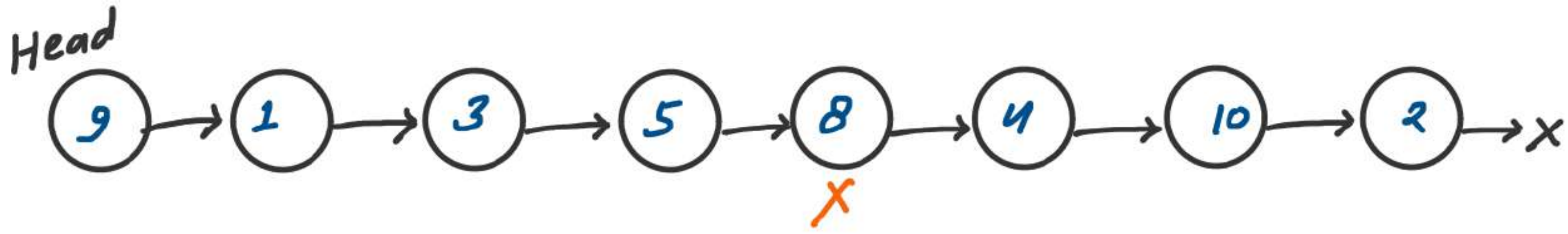


Iteration 3

if(!head) return



Ex3



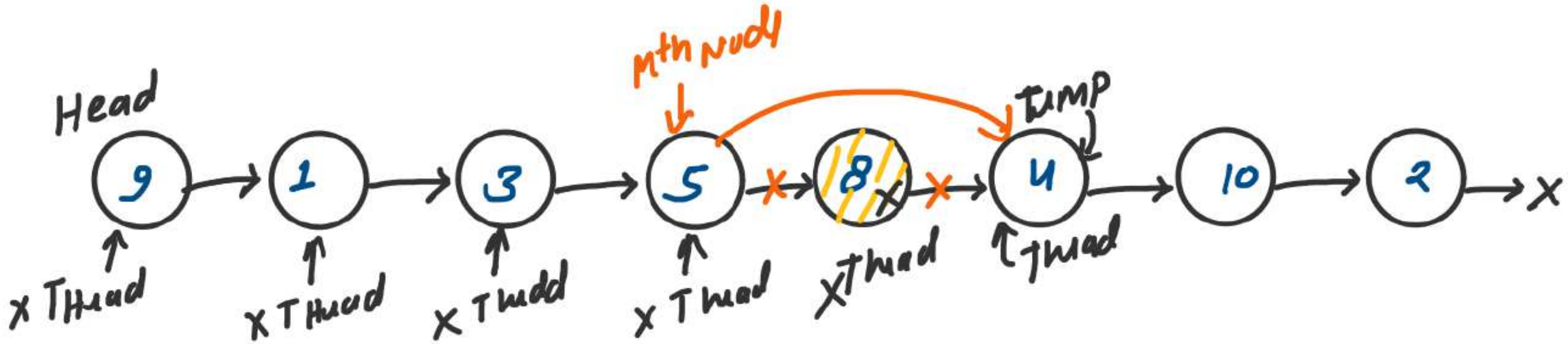
M=4 N=1

Output

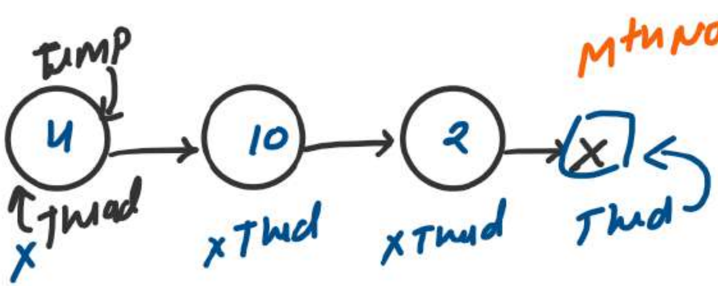
DRY RUN

M=4
N=1

Iteration 1



Iteration 2



Mth Node yadi Null par Hai

if (!mthNode)
return

→ output

Ex 4

Head

$9 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 4 \rightarrow 10 \rightarrow 2 \rightarrow X$

$M = 5 \qquad N = 1$

Output

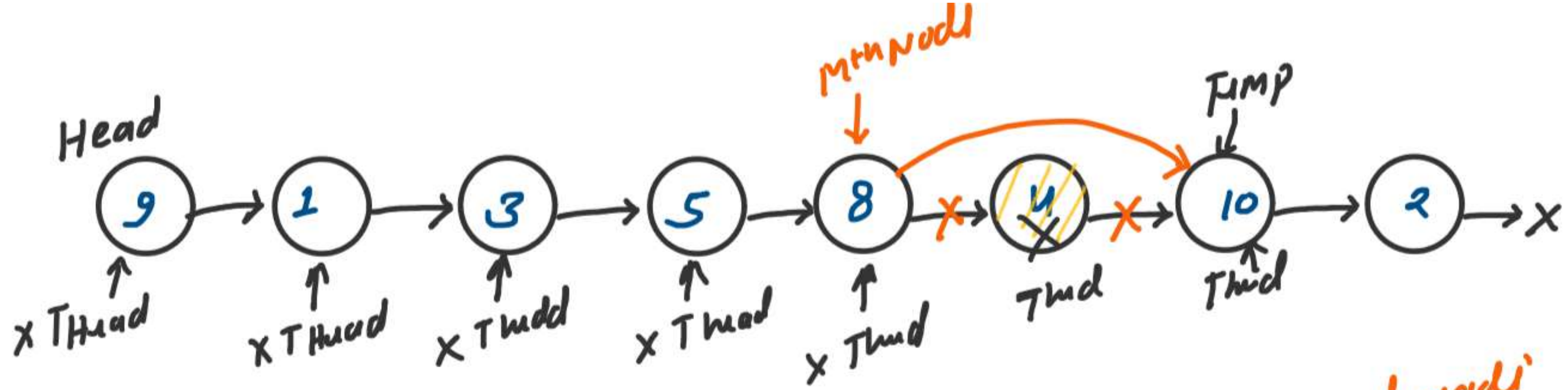
$9 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10 \rightarrow 2 \rightarrow X$

DRY RUN

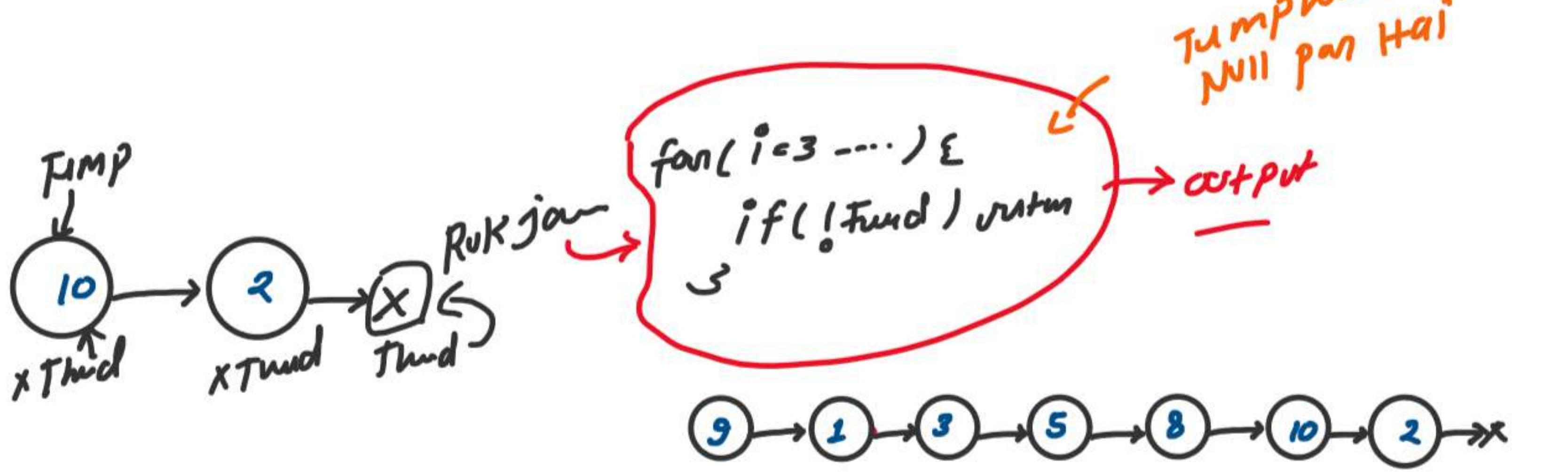
M=5

N=1

Iteration 1



Iteration 2



```cpp
// HW 04: Delete N Nodes after M Nodes (GFG)
class Solution
{
    public:
    void linkdelete(struct Node  *head, int M, int N)
    {
        // Base case
        if(!head) return;

        // Step 1: Traverse list to M position from 0th to (M-1)
        Node* tempHead = head;
        for(int i=0; i<M-1; i++){
            // Temp Head yadi Null par hai
            if(!tempHead) return;
            tempHead = tempHead->next;
        }
        Node* MthNode = tempHead;

        // Mth Node yadi null par hai
        if(!MthNode) return;

        // Step 2: Delete N node
        tempHead = MthNode->next;
        for(int i=0; i<N; i++){

            // Nth node available nhi hai
            // mtlb tempHead null hai
            if(!tempHead) break;

            Node* temp = tempHead->next;
            delete tempHead;
            tempHead = temp;
        }
        MthNode->next = tempHead;

        // Recursive call
        linkdelete(tempHead, M, N);
    }
};
```

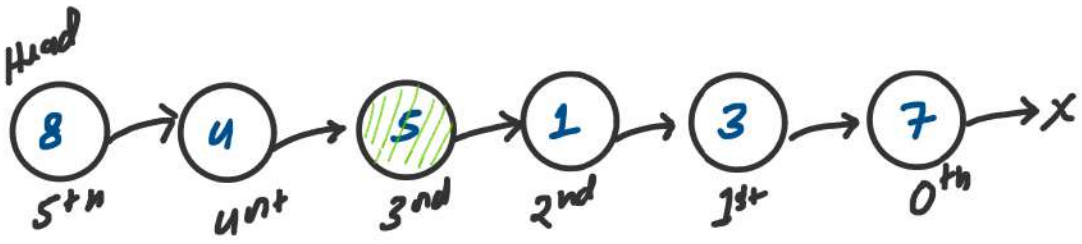
T.C. ⇒ O(N)

Where N is Numbers of nodes in the list.

S.C. ⇒ O(1)

HW 05: Print kth Node from the End (Hacker Rank)

Ex 1



pos = kth = 3

output = 5
3rd

RECURSIVE APPROACH

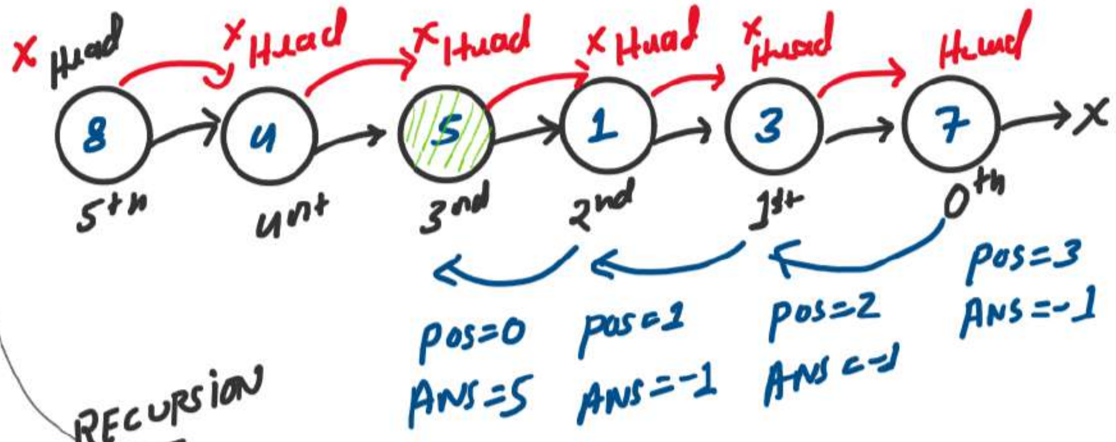
STEP:1 Traverse the list from Head to Tail

STEP:2 Again Traverse the list from Tail to [Jab tak pos==0]

STEP:3 Return Ans = 5

DRY RUN

pos = 3



RECURSION

Step 1

Base case
if (!Head)
return

fun(Head→next, pos, ANS)

Step: 3

return Ans

BACKTRACKING

Step 2

```
if( pos==0){
    ANS = head→data;
}
pos--;
```

catch1 (Galti ki chances Hai)

```cpp
// HW 05: Print kth Node from the End (Hackerrank)

/*
 * For your reference:
 *
 * SinglyLinkedListNode {
 *     int data;
 *     SinglyLinkedListNode* next;
 * };
 *
 */

void solve(SinglyLinkedListNode* head, int &pos, int &ans){
    // Base case
    if(head == 0) return;

    // Step 1: traverse list from head to tail
    solve(head->next, pos, ans);

    // Step 2: traverse list from tail to (Jab tak pos == 0)
    if(pos == 0){
        ans = head->data;
    }
    --pos;
}

int getNode(SinglyLinkedListNode* llist, int positionFromTail) {
        // Step 3: return ans
        int ans = -1;
        solve(llist, positionFromTail, ans);
        return ans;
}
```

REC

BackT

Final ANS

T.C. $\Rightarrow O(N)$

Where N is numbers of nodes in the list.

S.C. $\Rightarrow O(1)$

Ex1

Head

$8 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 3 \rightarrow 7 \rightarrow X$

5th 4th 3rd 2nd 1st 0th

pos = kth = 3

output = 5

3rd

ITERATIVE APPROACH

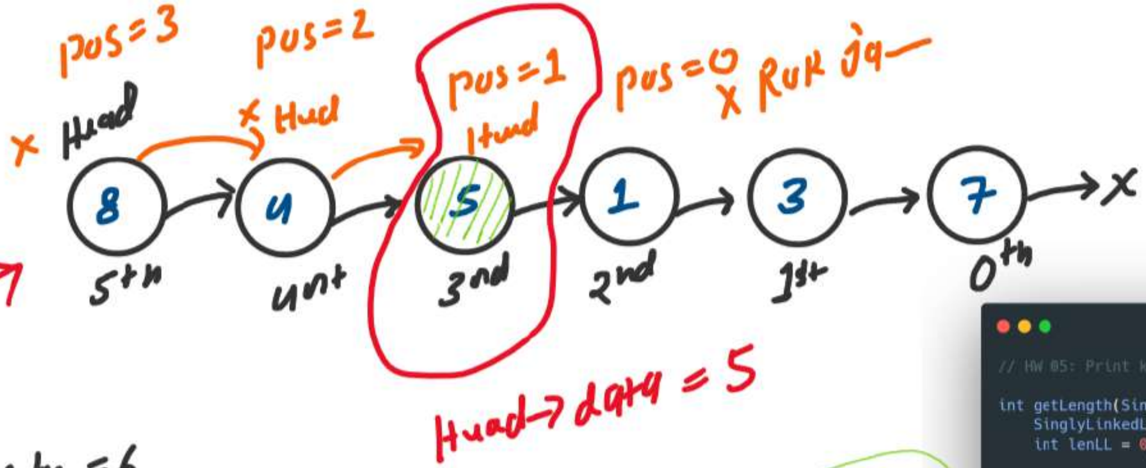
STEP:1 Get length of list

Step:2 subtract length - pos.

Step:3 Traverse list from Head to [Jab tak pos == 0].

DRY RUN

$K = 3$



Step:1 length = 6

Step:2 pos = length − K = 6 − 3
= 3

Step:3 return Head→data

T.C. ⇒ O(N) + O(N)
= O(N)

S.C. ⇒ O(1)

```
// HW 05: Print kth Node from the End (Hackerrank)

int getLength(SinglyLinkedListNode* head){
    SinglyLinkedListNode* temp = head;
    int lenLL = 0;

    // Base case
    if(head == 0) return lenLL;

    while (temp->next != 0) {
        lenLL++;
        temp = temp->next;
    }

    return lenLL;
}

int getNode(SinglyLinkedListNode* llist, int positionFromTail) {
        // Step 1: get length of list
        int length = getLength(llist);

        // Step 2: subtract pos from length
        int posFromHead = length - positionFromTail;

        // Step 3: traverse list from head to (Jab tak pos==0)
        while(posFromHead != 0){
            llist = llist->next;
            posFromHead--;
        }

        return llist->data;
}
```

→ O(N)

→ O(N)

HW 06: Flatten Linked List (GFG)

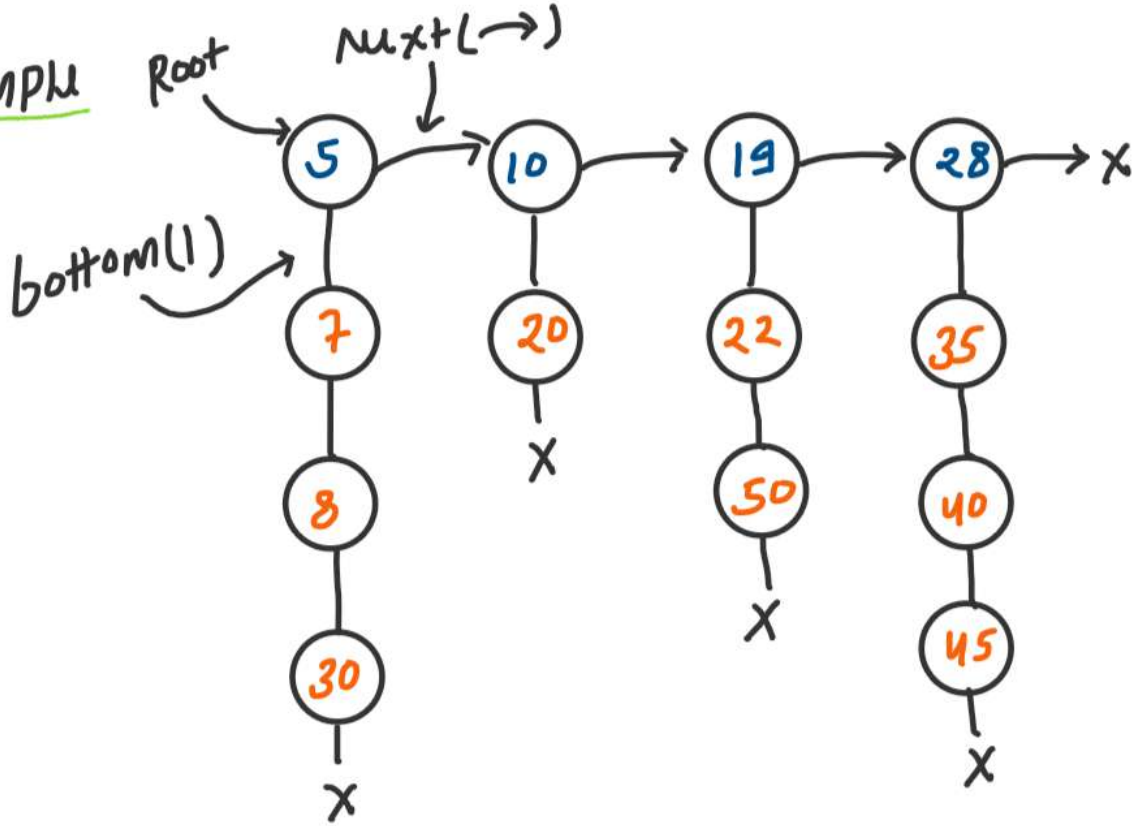
PROBLEM STATEMENT:
Given a Linked List of size N, where every node represents a ***sub-linked-list*** *and contains two pointers:*
(I) a ***next*** *pointer to the next node,*
(II) a ***bottom*** *pointer to a linked list where this node is head.*

Each of the sub-linked-list is in sorted order.

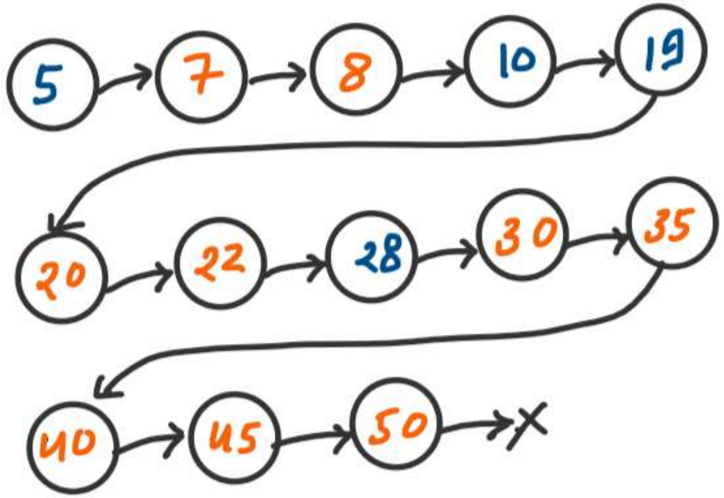
Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.

Note: *The flattened list will be printed using the bottom pointer instead of the next pointer.*

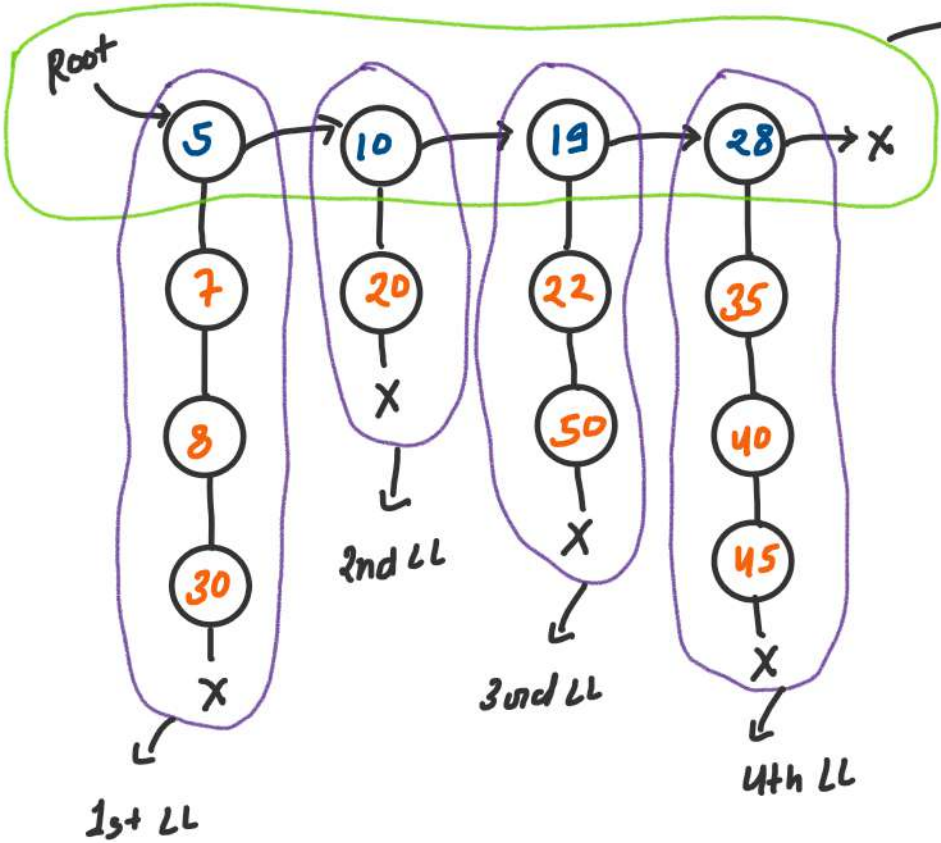
Example



Output



Build Logic



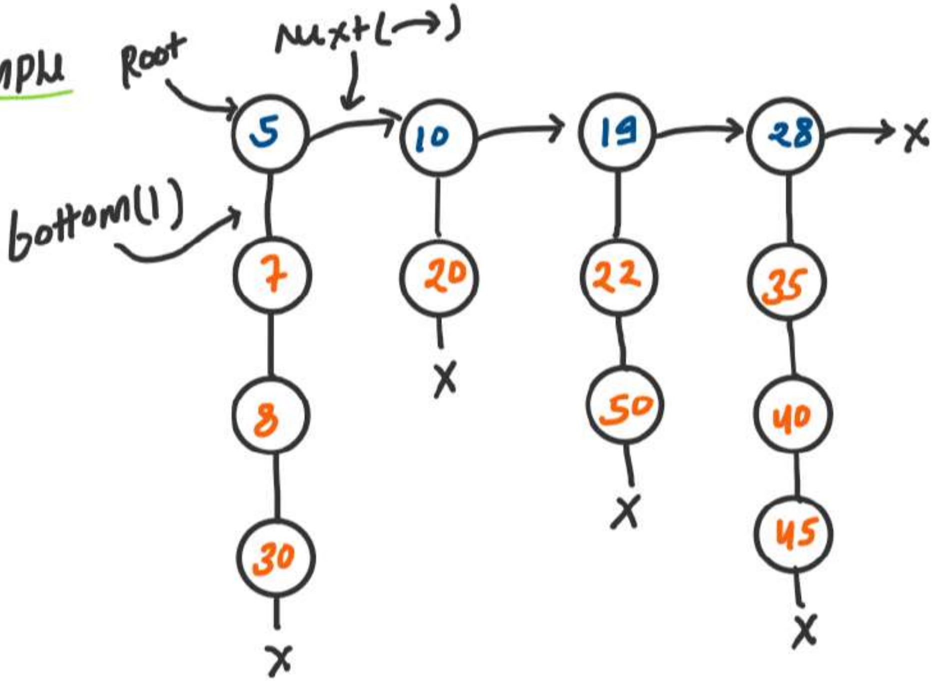
Approach: Recursive Approach

Step1 merge Two sorted LL

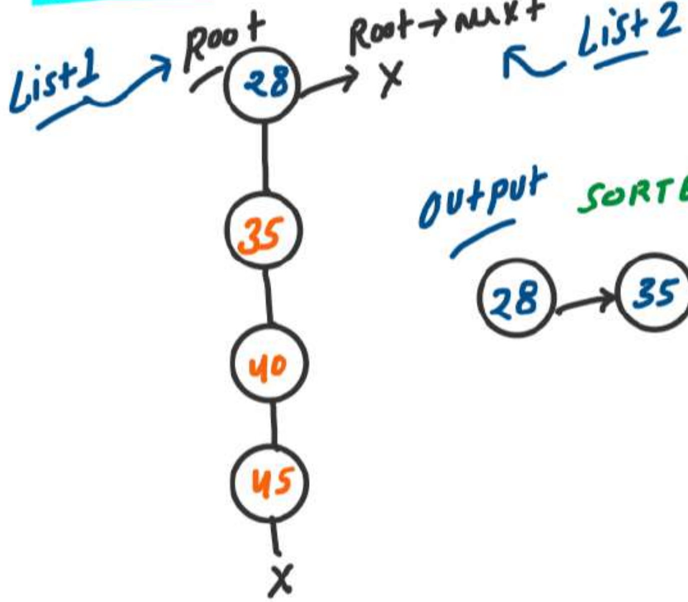
Step2 merge a next sorted LL with already two merged LL

DRY RUN

Example

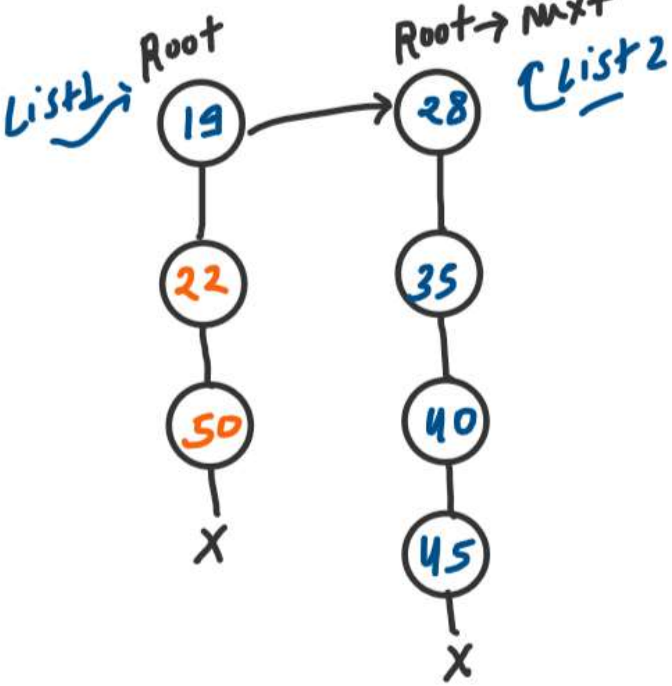


R. Call 1 f(28, NULL)



Output SORTED MERGE

R. Call2 f(19, 28)

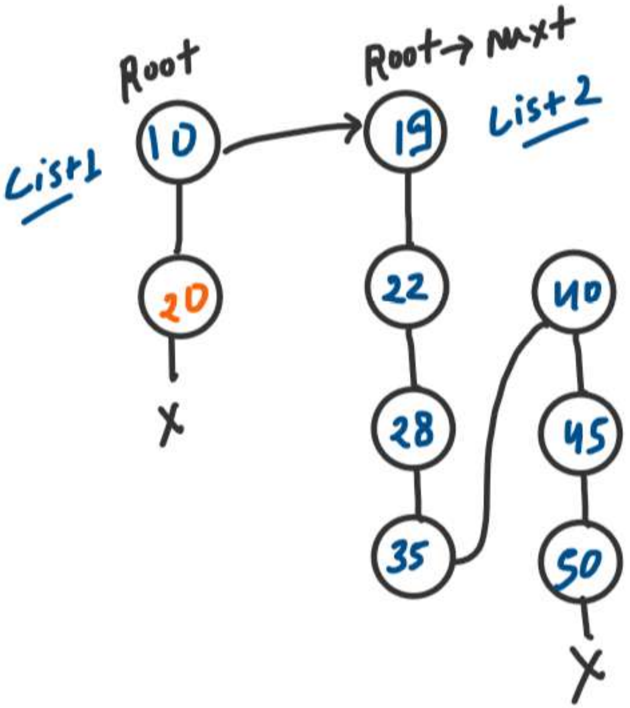


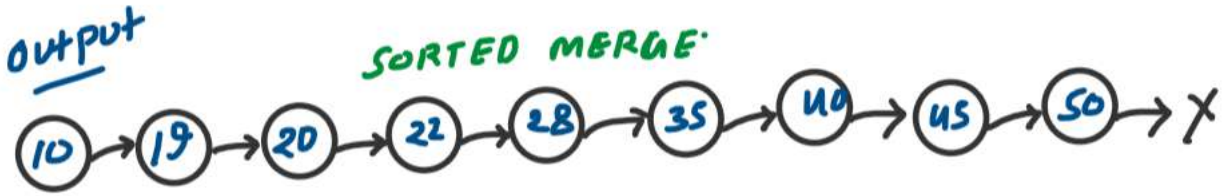
output

SORTED MERGE

R. Call 3

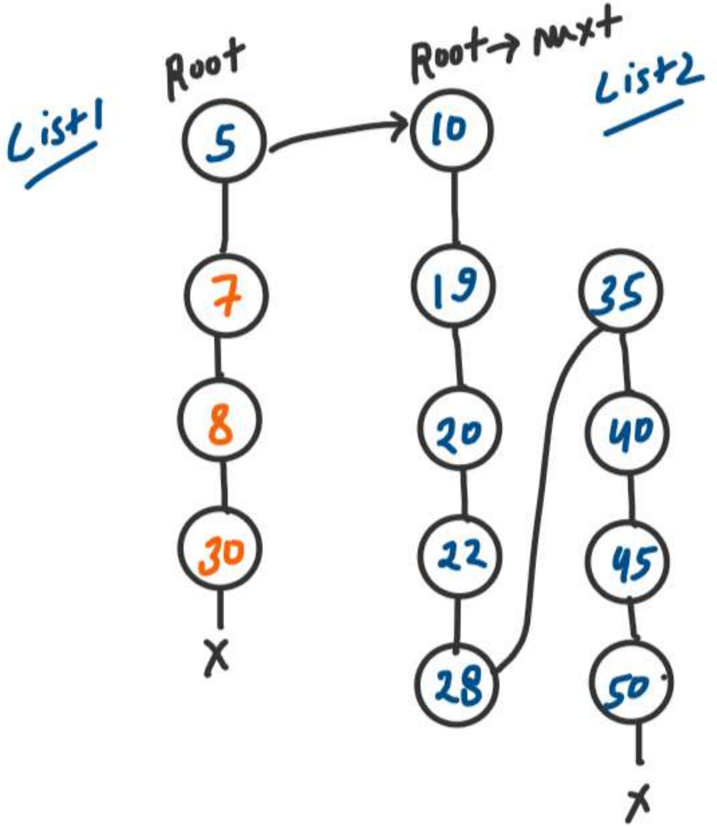
f(10, 19)





R. Call 4

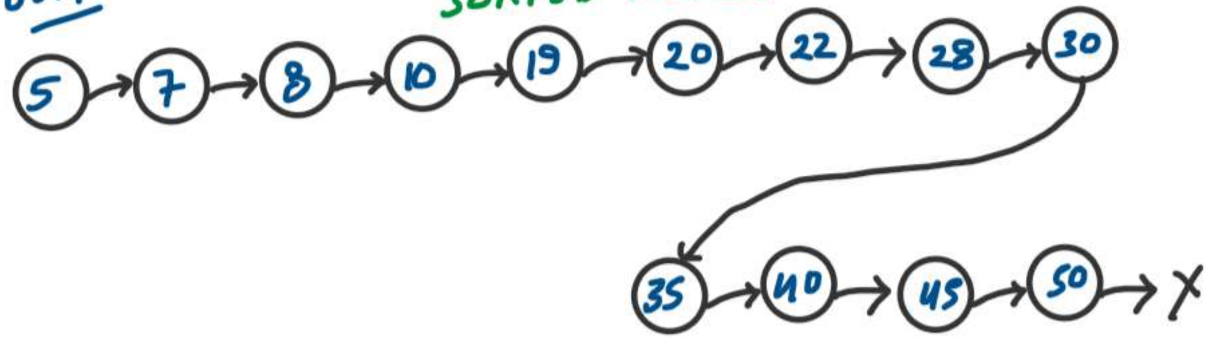
f(5, 10)



Final

output

SORTED MERGE.



ALGORITHM

Main LL → First Two Node pick

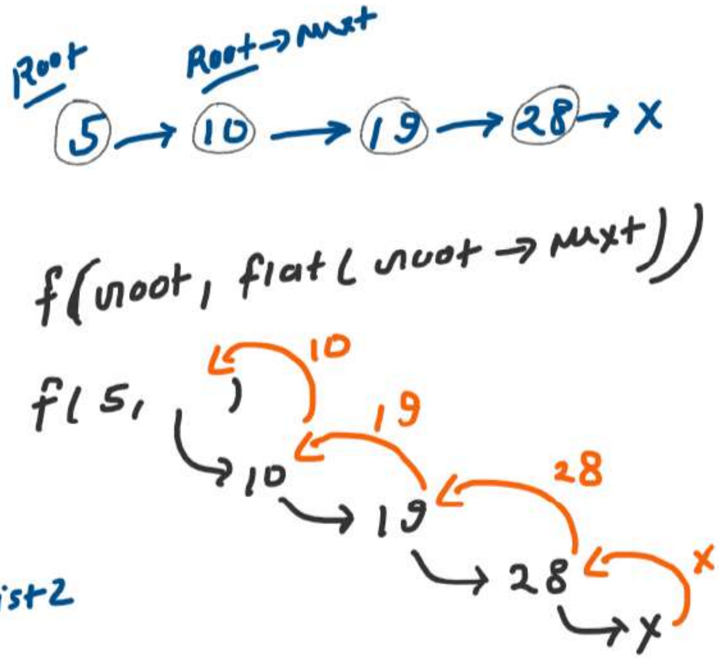
Root & Root → next

List1   List2

MERGE
with Bottom
in single sorted list

```
/* Node structure  used in the program
struct Node{
    int data;
    struct Node * next;
    struct Node * bottom;

    Node(int x){
        data = x;
        next = NULL;
        bottom = NULL;
    }
};
*/

Node* mergeTwoSortedLL(Node* list1, Node* list2){...}

Node *flatten(Node *root)
{
   // Base case
   if(root == NULL) return NULL;

   Node* mergedLL = mergeTwoSortedLL(root, flatten(root->next));
   return mergedLL;
}
```

```
Node* mergeTwoSortedLL(Node* list1, Node* list2){
    // Base case
    if(list1 == NULL) return list2;
    if(list2 == NULL) return list1;

    // Processing
    Node* ans = NULL;
    if(list1->data < list2->data){
        ans = list1;
        list1->bottom = mergeTwoSortedLL(list1->bottom, list2);
    }
    else{
        ans = list2;
        list2->bottom = mergeTwoSortedLL(list1, list2->bottom);
    }

    return ans;
}
```



list1 list2

1st call f(28, X)

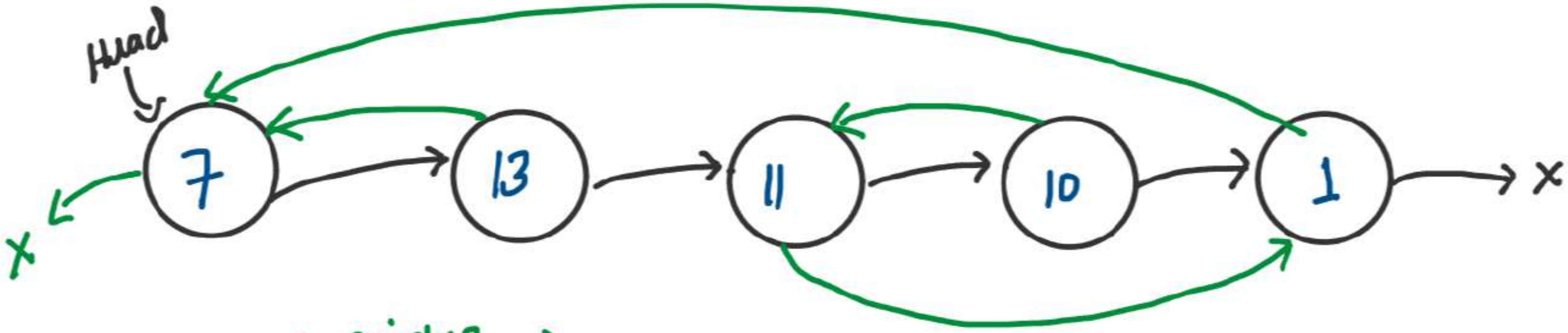
2nd call f(19, 28)

3rd call f(10, 19)

4th call f(5, 10)

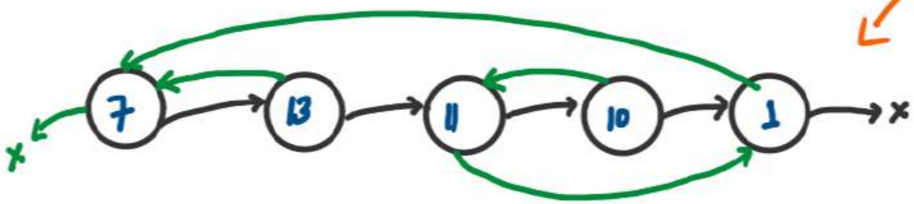
HW 07: Copy List with Random Pointer (Leetcode-138)

Input



- Random pointer →
- next pointer →

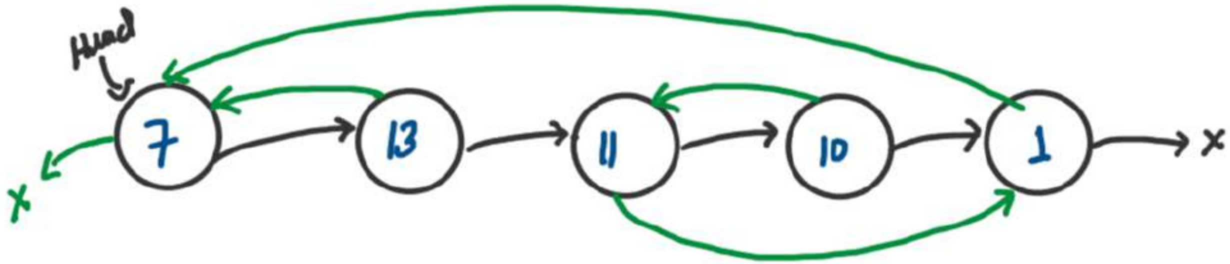
Output



Deep copy of list

Approach 1 Using map

step1 Copy List in map
using Recursion

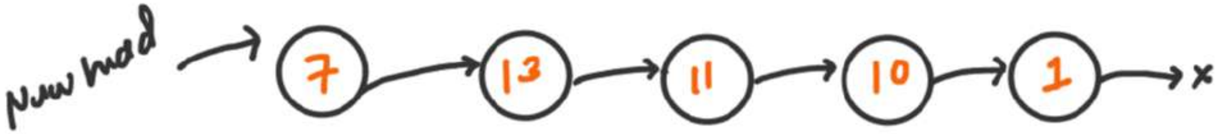


mp

Old ptr	New ptr
7	7
13	13
11	11
10	10
1	1

Base case
if(!head) return NULL;

Node* newNode = new Node(head->val);
mp[head] = newHead;
newNode->next = f(head->next, mp);



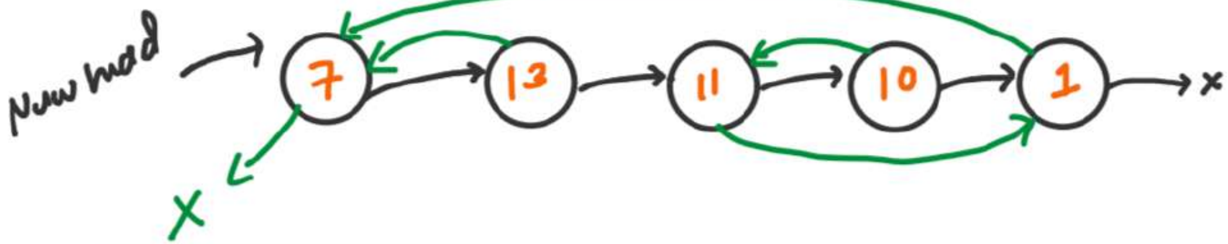
Step2 Allocate the Random pointer

```
if( head→Random){
    Newhead→Ramom = mp[head→Random];
}
```

Map[old ptr] = New pointer

key of map

value of map



Return Newhead

Output

```cpp
// HW 07: Copy List with Random Pointer (Leetcode-138)

/*
Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};
*/

class Solution {
public:
    Node* solve(Node* head, unordered_map<Node*, Node*> &mp){

        // Base case
        if(!head) return NULL;

        // Step 1: Copy list in map
        Node* newHead = new Node(head->val);
        mp[head] = newHead;
        newHead->next = solve(head->next, mp);

        // Step 2: Allocate the random pointer
        if(head->random){
            newHead->random = mp[head->random];
        }

        return newHead;
    }

    Node* copyRandomList(Node* head) {
        unordered_map<Node*, Node*> mp;
        return solve(head, mp);
    }
};
```

Time Complexity: *O(N),*
Where N is number of nodes in list

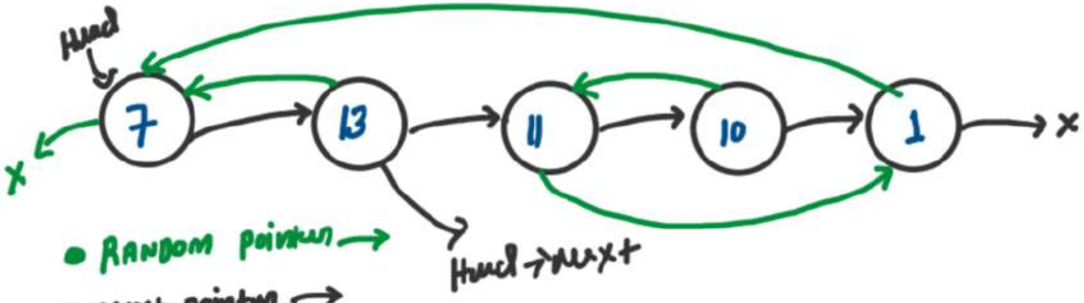
Space Complexity: *O(N),*
where N is number of elements (Nodes) stored in map

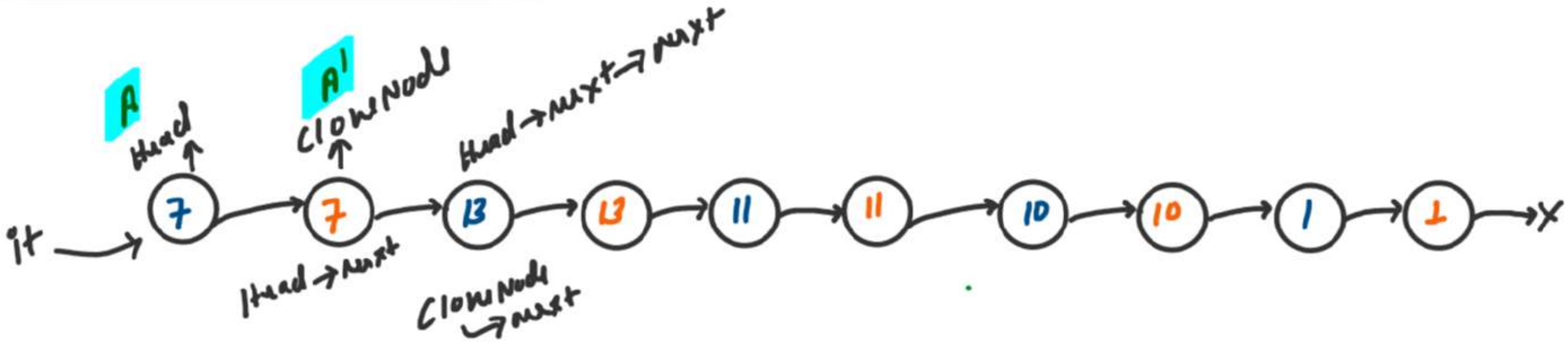
Approach 2 without using map

Step:1

```
Node* solve(Node* head){
    if(!head) return NULL;

    // Step 1: Clone A->A'
    Node* it = head; // Iterating Over Old Head
    while(it){
        Node* cloneNode = new Node(it->val);
        cloneNode->next = it->next;
        it->next = cloneNode;
        it = cloneNode->next;
    }
}
```

Input

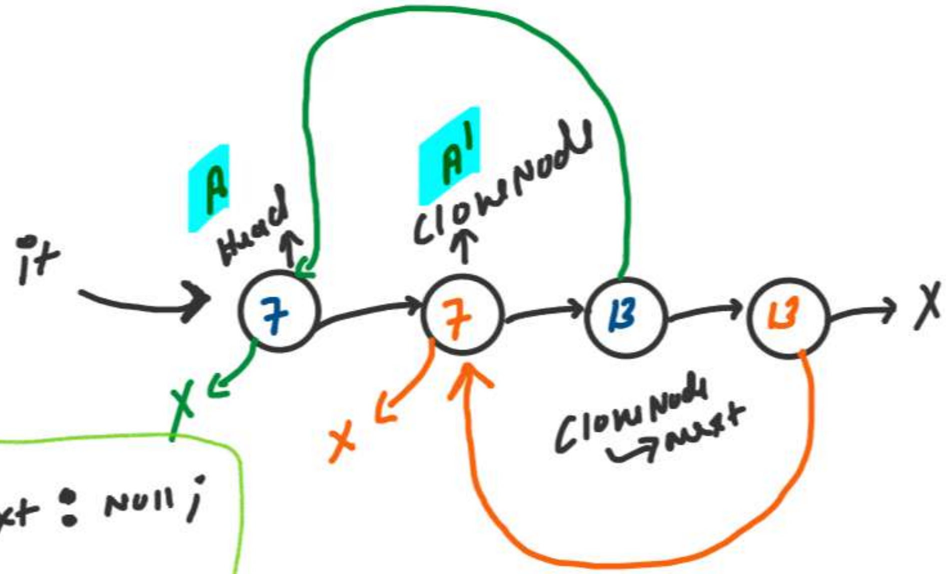




step2

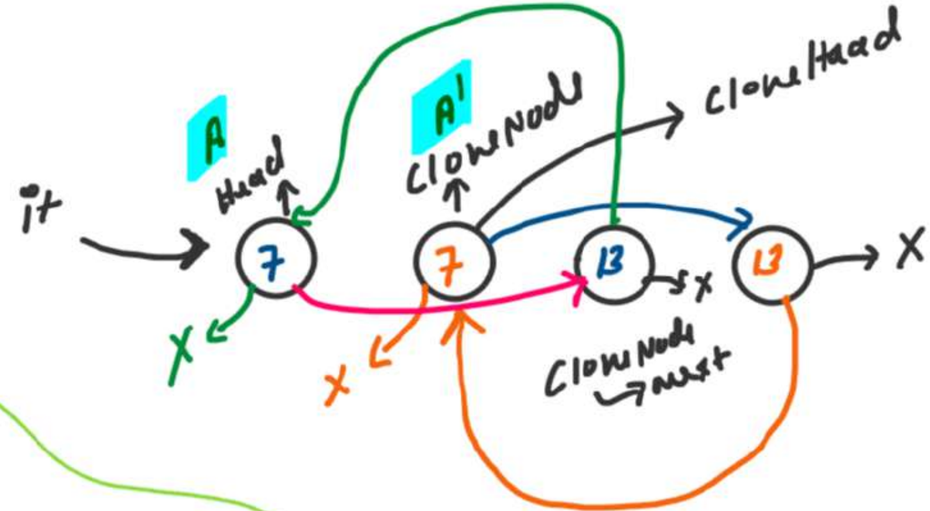
Assign Random pointer A'
with the help of A

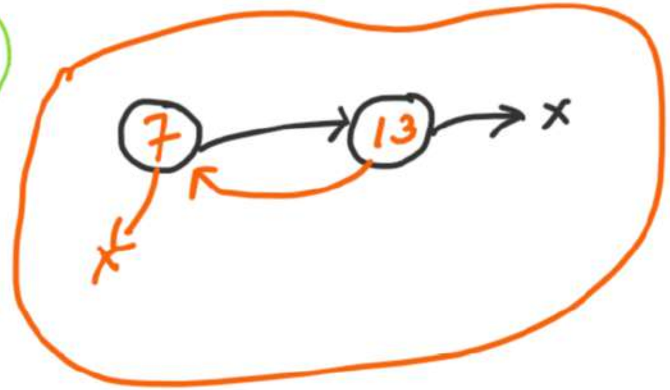
```
it = head;
while(it) {
    Node* cloneNode = it->next;
    cloneNode->Random =
        it->Random ? it->Random->next : NULL;
    it = it->next->next;
}
```



Step3 Detach A' from A

```
it = head;
Node* cloneHead = it->next;
while(it) {
    Node* cloneNode = it->next;
    it->next = cloneNode->next;
    if(cloneNode->next) {
        cloneNode->next = cloneNode->next->next;
    }
    it = it->next;
}
return cloneHead;
```





```cpp
class Solution {
public:
    Node* solve(Node* head){
        if(!head) return NULL;

        // Step 1: Clone A->A'
        Node* it = head; // Iterating Over Old Head
        while(it){
            Node* cloneNode = new Node(it->val);
            cloneNode->next = it->next;
            it->next = cloneNode;
            it = cloneNode->next;
        }

        // Step 2: Assign random pointer of A' with the help of A
        it = head;
        while(it){
            Node* cloneNode = it->next;
            cloneNode->random = it->random ? it->random->next : NULL;
            it = cloneNode->next;
        }

        // Step 3: Detach A' from A

    }

    Node* copyRandomList(Node* head) {
        return solve(head);
    }
};
```

step 3

```cpp
it = head;

// cloneHead is not changed after its initial assignment
Node* cloneHead = it->next;

while(it){
    Node* cloneNode = it->next;
    it->next = cloneNode->next;
    if(cloneNode->next){
        cloneNode->next = cloneNode->next->next;
    }
    it = it->next;
}
return cloneHead;
```

T.C. ⇒ O(N)

S.C. ⇒ O(1)

HW 08: Rotate List (Leetcode-61)

Ex

Head

1 → 2 → 3 → 4 → 5 → x

K = 1 5 → 1 → 2 → 3 → 4 → x

K = 2 4 → 5 → 1 → 2 → 3 → x

K = 3 3 → 4 → 5 → 1 → 2 → x

K = 4 2 → 3 → 4 → 5 → 1 → x

K = 5 1 → 2 → 3 → 4 → 5 → x

K = 6

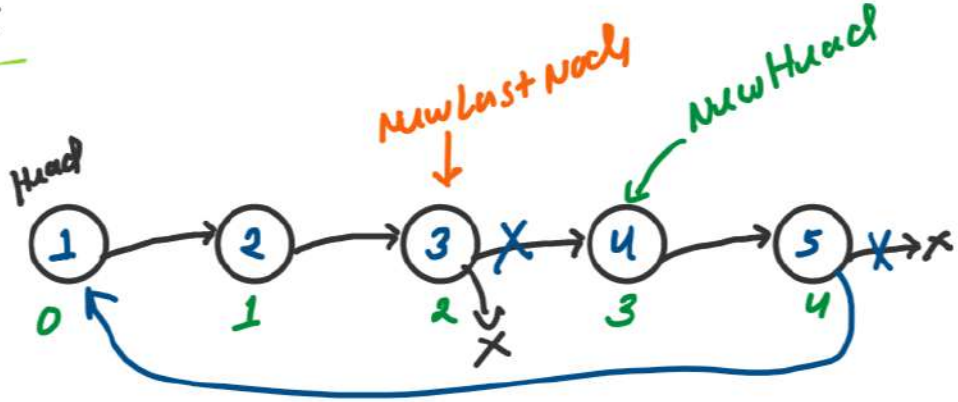
5 → 1 → 2 → 3 → 4 → x

K = 7

4 → 5 → 1 → 2 → 3 → x

Logic

Input =



$K = 2$

Step1 Find length of List

Step2 Find Actual Rotation of K

Step3 Find newLastNode Position

Step1

List length = 5

Step2

Actual Rotation = K % length

= 2 % 5

= 2

Step3

newLastNode posi = len − ActualRota − 1

= 5 − 2 − 1

= 2

```cpp
// HW 08: Rotate List (Leetcode-61)

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:

    int getLength(ListNode* head){...}

    ListNode* rotateRight(ListNode* head, int k) {
        // Corner Case
        if(!head) return NULL;

        // Step 1: Find length of list
        int len = getLength(head);

        // Step 2: Find actual rotation of k
        int actualRotateK = k % len;

        // Corner case
        if(actualRotateK == 0) return head;

        // Step 3: Find position of lastNewNode

    }
};
```

```cpp
int getLength(ListNode* head){
    ListNode* temp = head;
    int len = 0;

    while(temp){
        len++;
        temp = temp->next;
    }

    return len;
}
```

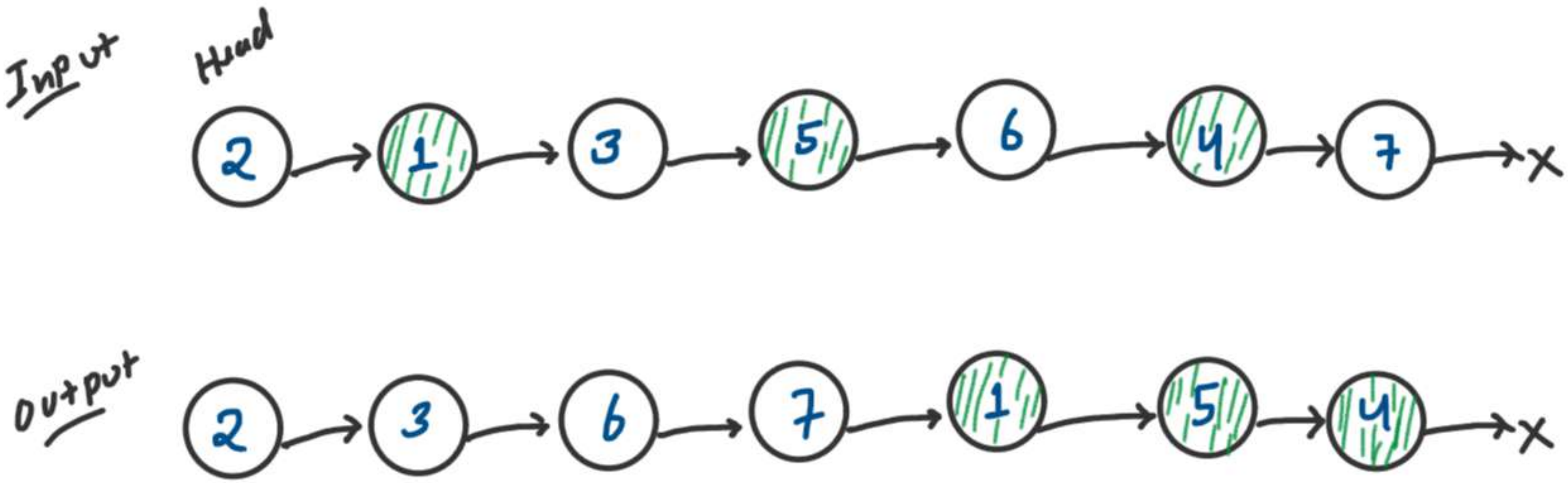
$T.C. \Rightarrow O(N)$

$S.C. \Rightarrow O(1)$

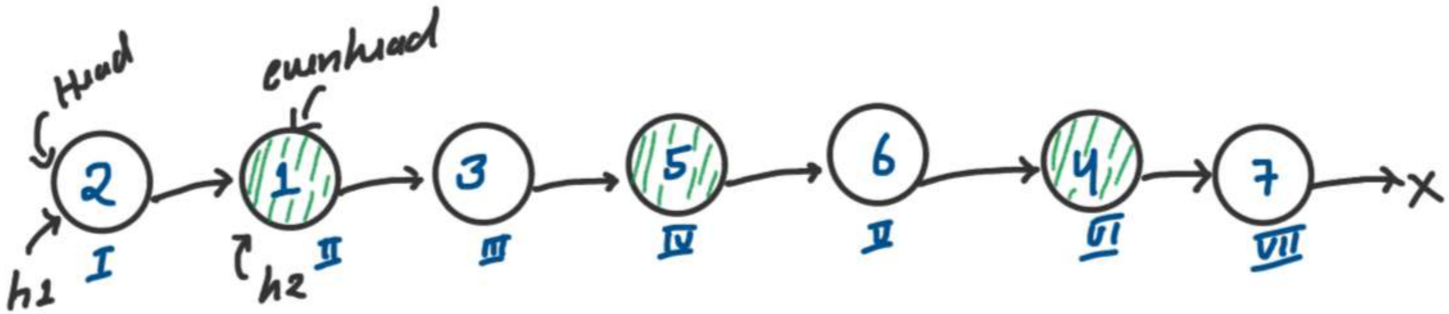
```cpp
    int newLastNodePos = len - actualRotateK - 1;

    ListNode* newLastNode = head;
    for(int i=0; i<newLastNodePos; i++){
        newLastNode = newLastNode->next;
    }

    // Save newLastNode->next in newHead to track
    ListNode* newHead = newLastNode->next;
    newLastNode->next = NULL;

    // newHead ka next node yadi null ho jata hai
    // to use old Head se meet kara do
    ListNode* it = newHead;
    while(it->next != NULL){
        it = it->next;
    }
    it->next = head;

    return newHead;
```

HW 09: Odd Even Linked List (Leetcode-328)

Ex



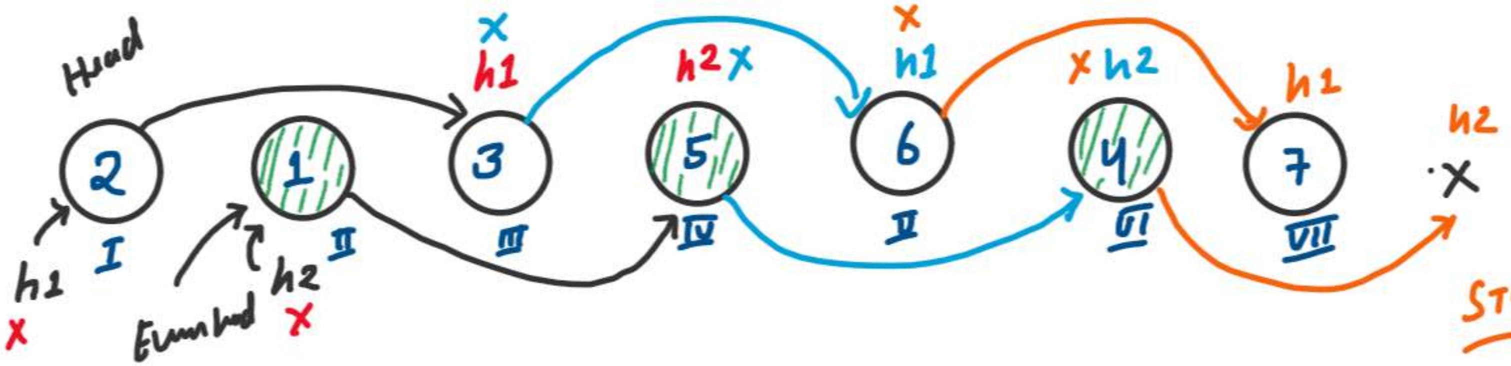
Logic build



Node* h1 = head; ← odd Indexed list

Node* h2 = head → next; ← Even Indexed list

Node* evenHead = h2; ← save the evenhead to link the odd list



STOP

h2 == NULL &&

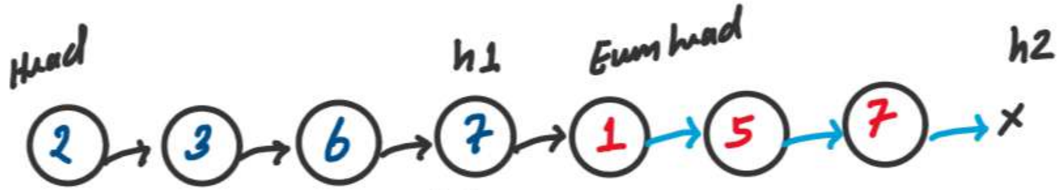
h2->next == NULL

h1->next = h2->next;

h2->next = h2->next->next;

h1 = h1->next;

h2 = h2->next;



Re-group the both list

{ h1->next = evenhead;

return head; }

```cpp
// HW 09: Odd Even Linked List (Leetcode-328)

class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(head == NULL || head->next == NULL) return head;
        // Odd indexed list
        ListNode* h1 = head;
        // Even indexed list
        ListNode* h2 = head->next;
        // Save h2 for attaching the odd index list
        ListNode* evenHead = h2;

        while(h2 && h2->next){
            h1->next = h2->next;
            h2->next = h2->next->next;

            h1 = h1->next;
            h2 = h2->next;
        }

        // Odd and even indexed list ko regroup krdo
        h1->next = evenHead;
        return head;
    }
};
```

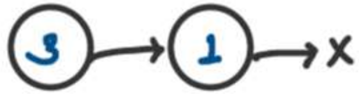
Time complexity: *O(N),*
Where N is number of nodes of the linked list

Space complexity: *O(1),*
Where no extra space used

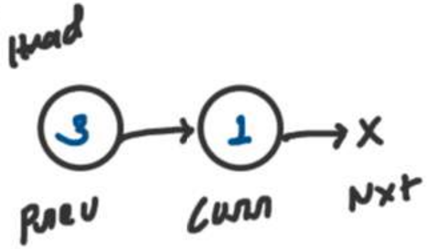
HW 10: Find Minimum and Maximum Number of Nodes Between Critical Points (Leetcode-2048)

Ex 1

Input



Dry Run



No critical point exist

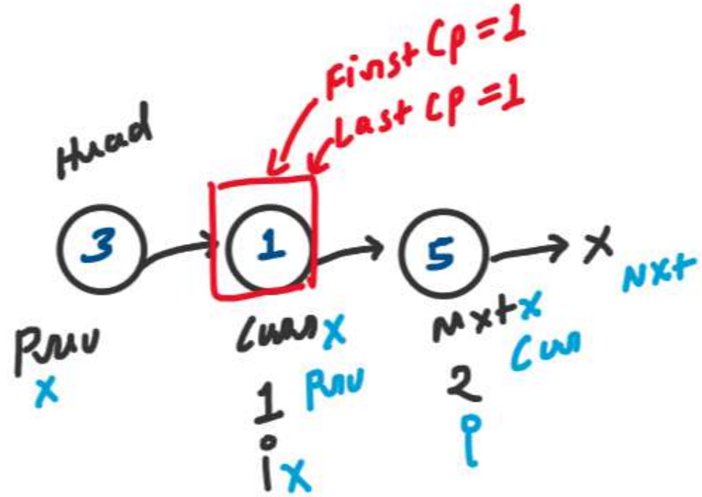
Output [-1,-1]

```
vector<int> ans = {-1, -1};
Node* prev = head;
if(!prev) return Ans;
Node* curr = head->next;
if(!(curr)) return Ans;
Node* nxt = curr->next;
if(!nxt) return Ans;
```

Ex2

$3 \rightarrow 1 \rightarrow 5 \rightarrow X$

Dry RUN

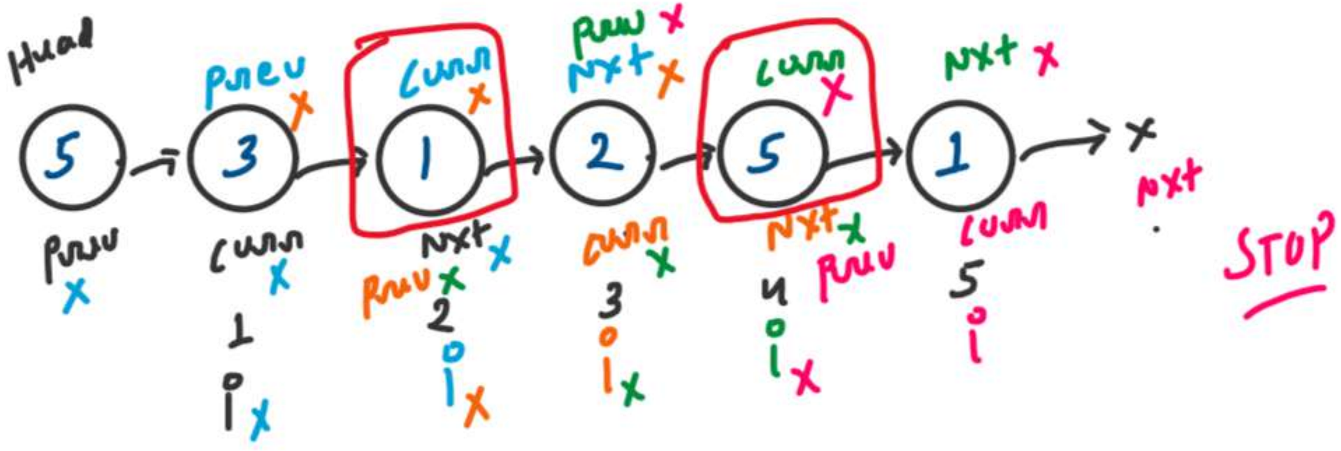


One critical point exist

$1 < 3$ && $1 < 5$

Output ⇒ [-1 , -1]

Ex:3



minDis = i - LastCP => 4 - 2 => 2

FirstCP = -1 2 2

LastCP = -1 2 4

MaxDis = LastCP - FirstCP

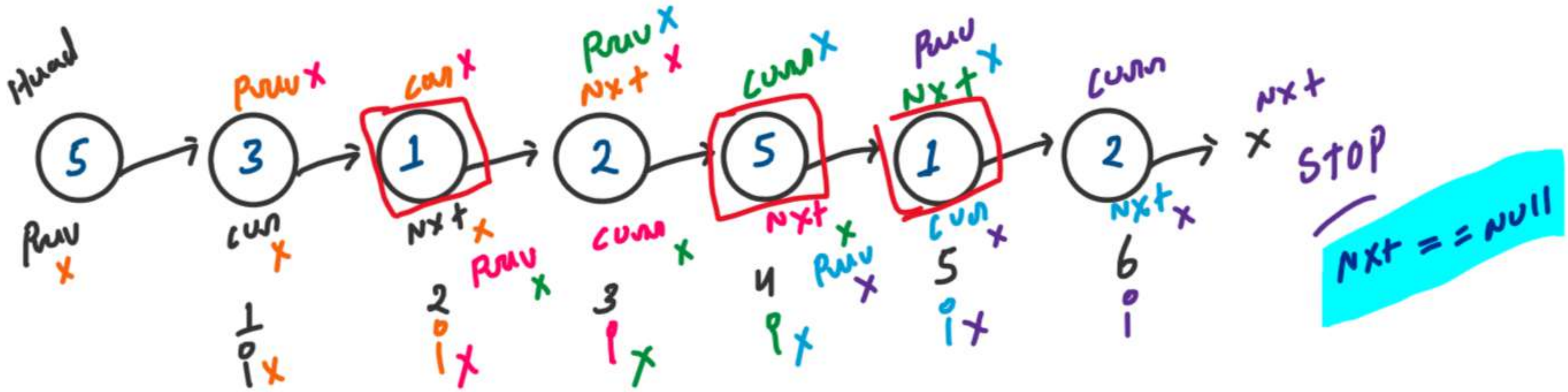
= 4 - 2

= 2

Two Critical points Exist

O/P => [2,2]

Ex:4



First CP = ~~-1~~ 2

Last CP = ~~-1~~ ~~2~~ ~~4~~ 5

minDis = i - Last CP ⇒ 4 − 2 = ~~2~~ ⇒ 5 − 4 = 1

maxDis ⇒ LastCP − firstCP

⇒ 5 − 2 = 3

Three critical points exist

output ⇒ [1, 3]

```cpp
// HW 10: Find Minimum and Maximum Number of Nodes Between
Critical Points (Leetcode-2048)


class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
        // vector initialized with minDis and maxDis
        vector<int> ans = {-1, -1};
        ListNode* prev = head;
        if(!prev) return ans;
        ListNode* curr = prev->next;
        if(!curr) return ans;
        ListNode* nxt = curr->next;
        if(!nxt) return ans;

        int firstCP = -1;
        int lastCP = -1;
        int i = 1;
        int minDis = INT_MAX;

        while(nxt){...}

        // Only one critical poit found condition
        if(lastCP == firstCP){
            return ans;
        }
        else {
            ans [0] = minDis;
            ans [1] = lastCP - firstCP;
        }
        return ans;
    }
};
```

```cpp
    while(nxt){
        bool isCP = ((curr->val > prev->val && curr->val > nxt->val)
                    || (curr->val < prev->val && curr->val < nxt->val))
                     ? true : false;

        // first critical point condition
        if(isCP && firstCP == -1){
            firstCP = i;
            lastCP = i;
        }
        // at least one critical point ke liye condition
        else if(isCP){
            minDis = min(minDis, i - lastCP);
            lastCP = i;
        }
        i++;
        prev = prev->next;
        curr = curr->next;
        nxt = nxt->next;
    }
```

Time complexity: *O(N),*
Where N is number of nodes of the linked list

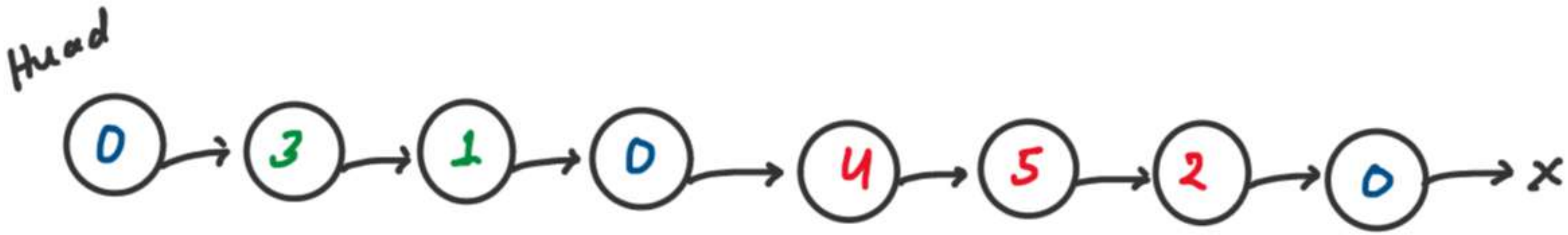
Space complexity: *O(1),*
Where no extra space used

HW 11: Merge Nodes in between Zeros (Leetcode-2181)

Example 1:
Input: head = [0,3,1,0,4,5,2,0]
Output: [4,11]

Example 2:
Input: head = [0,1,0,3,0,2,2,0]
Output: [1,3,4]

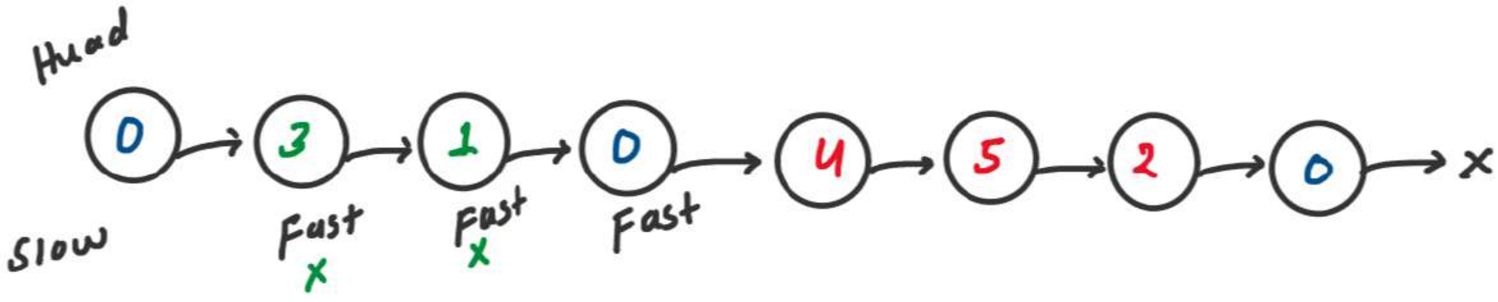
Ex1



Explain

Output [3+1 , 4+5+2]
[4 , 11]

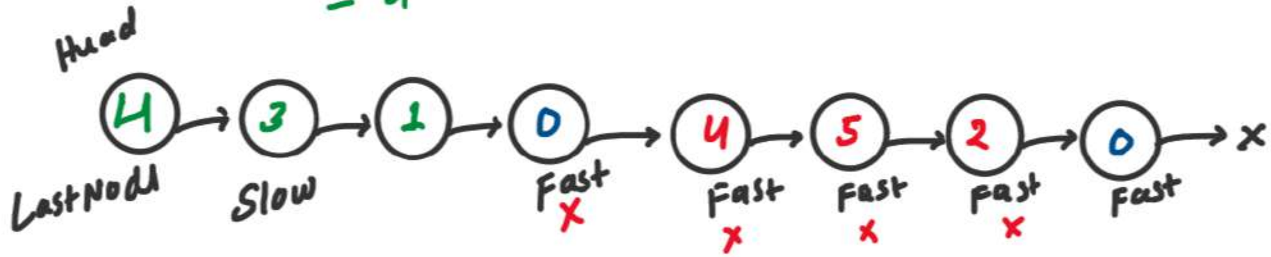
DRY RUN



Node* slow = head;
Node* Fast = head->next;
Node* newLastNode = Null;
int sum = 0

sum kab tak karna hai - jab tak Fast->val == 0 na ho

sum = 0 + 3 + 1
= 4

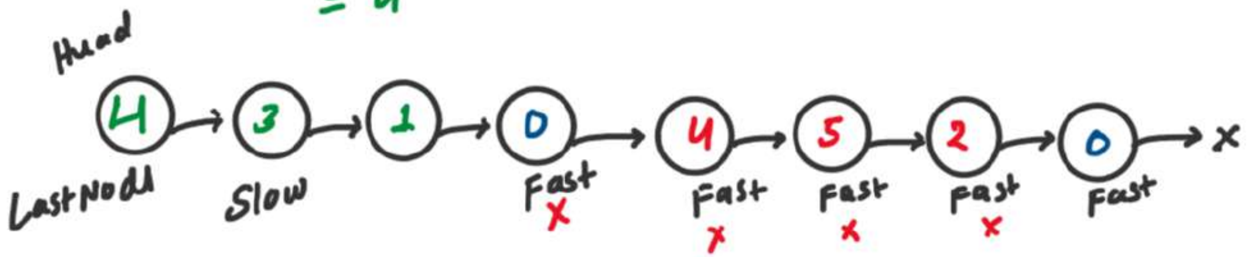


sum = 0 + 4 + 5 + 2
= 11

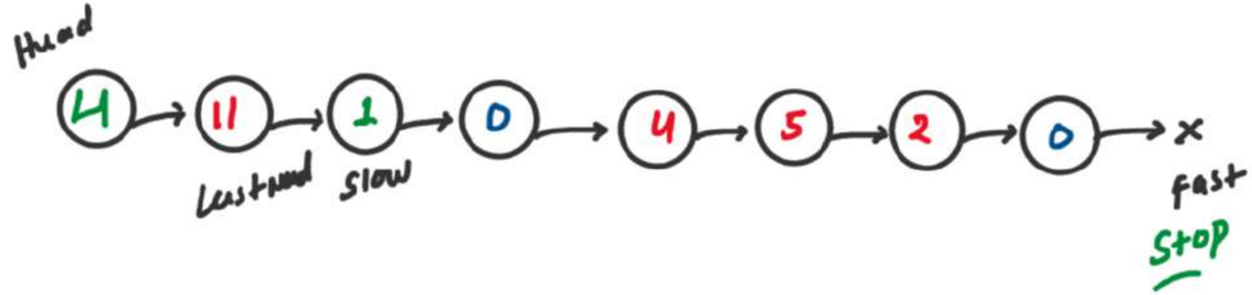
```
Node* slow = head;
Node* fast = head->next;
Node* newLastNode = NULL;
int sum = 0

while(fast){
    if(fast->val != 0){
        sum += fast->val;
    }
    else if(fast->val == 0){
        slow->val = sum;
        lastNode = slow;
        slow = slow->next;
        sum = 0;
    }
    fast = fast->next
}
```

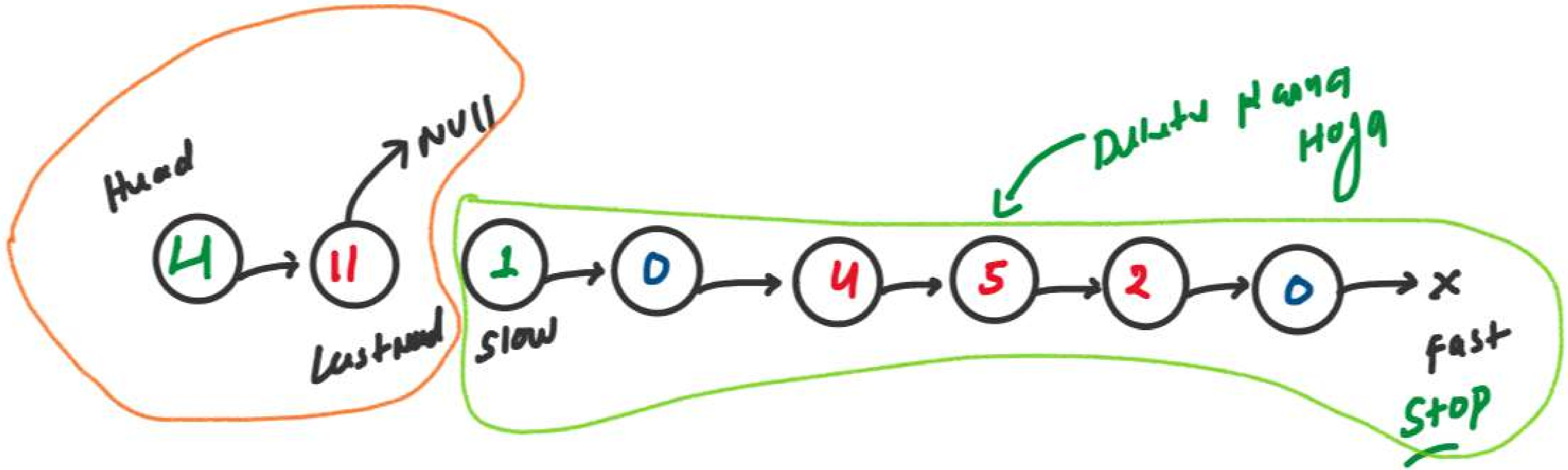
Sum kab tak karna hai - jab tak fast → val == 0 na ho

Sum = 0 + 3 + 1
= 4



Sum = 0 + 4 + 5 + 2
= 11






```
node* Temp = slow;
Lastnode->next = NULL;
delete Temp;

return head;
```

```cpp
// HW 11: Merge Nodes in between Zeros (Leetcode-2181)

class Solution {
public:
    ListNode* mergeNodes(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head->next;
        ListNode* newLastNode = NULL;
        int sum = 0;

        while(fast){
            if(fast->val != 0){
                sum += fast->val;
            }
            else{
                // fast->val == 0
                slow->val = sum;
                newLastNode = slow;
                slow = slow->next;
                sum = 0;
            }
            fast = fast->next;
        }

        ListNode* temp = slow;

        // Just formed new list
        newLastNode->next = NULL;

        // Deleting old list
        delete temp;

        return head;
    }
};
```

Time complexity: O(N),
Where N is number of nodes of the linked list

Space complexity: O(1),
Where no extra space used