

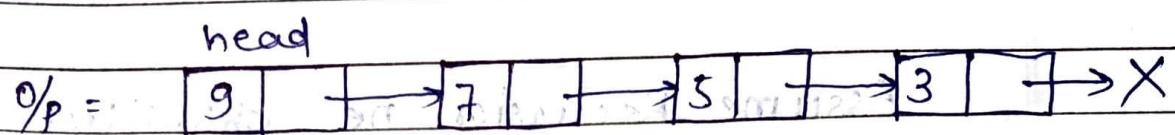
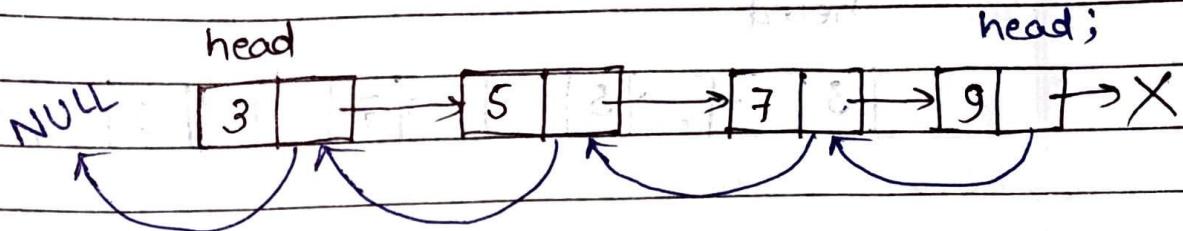
Linked List

-By Nikhil Kumar

<https://www.linkedin.com/in/nikhilkumar0609/>

LINKED LIST

Q Reverse linked list



Program :-

```
Node* reverseLinkedList(Node* head)
```

```
{
```

```
    if (head == NULL || head->next == NULL) {
```

```
        return head;
```

```
}
```

```
; base case reached
```

```
Node* prev = NULL;
```

```
Node* curr = head;
```

```
; move to first node
```

```
while (curr != NULL) {
```

```
    Node* forward = curr->next;
```

```
    curr->next = prev;
```

```
    prev = curr;
```

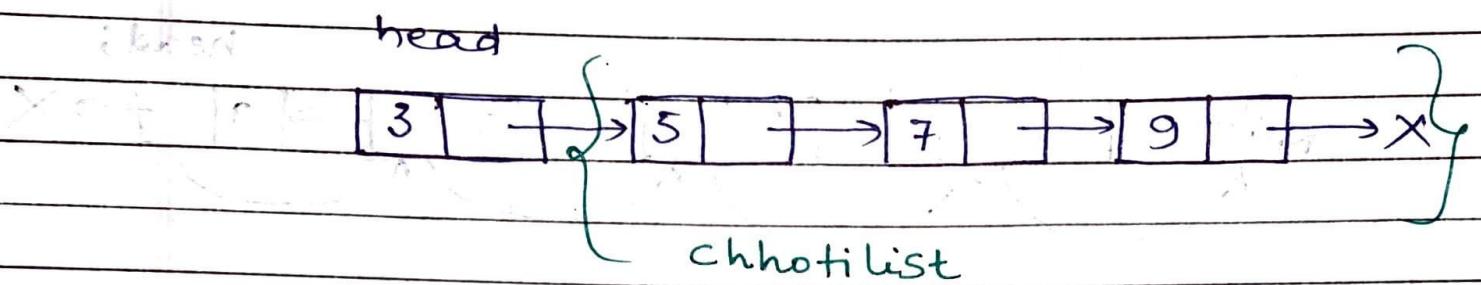
```
    curr = forward;
```

```
}
```

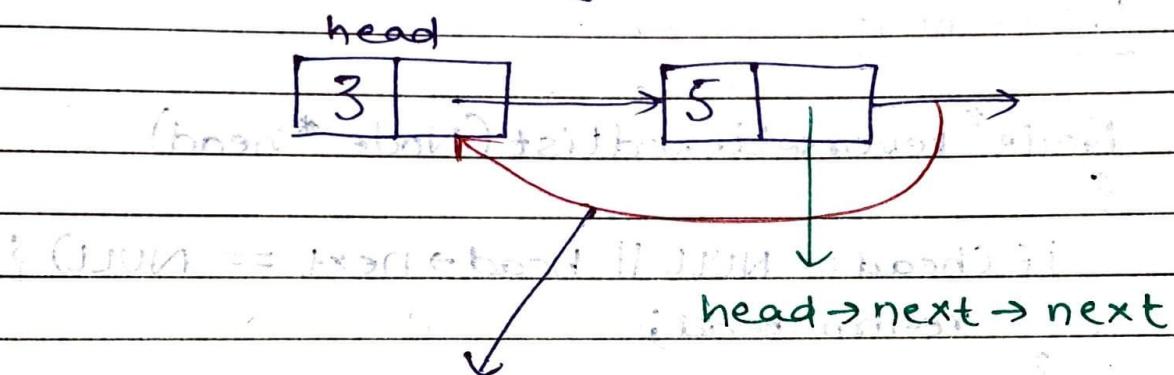
```
return prev;
```

```
}
```

→ Recursive approach



Assume Recursion ne chhotolist ko
reverse kr diya, uske baad



$\text{head} \rightarrow \text{next} \rightarrow \text{next} = \text{head};$

$(9 \rightarrow 7 \rightarrow 5 \rightarrow 3)$ shall
be called chhotahHead

uske baad, $\text{head} \rightarrow \text{next} = \text{NULL};$

$(9 \rightarrow 7 \rightarrow 5 \rightarrow 3 \rightarrow x)$

Program:-

```
Node* reverse(Node* head) {  
    if (head == NULL || head->next == NULL) {  
        return head;  
    }
```

```
    Node* chhotatHead = reverse(head->next);  
    head->next->next = head;  
    head->next = NULL;  
    return chhotatHead;
```

```
}
```

```
Node* reverseLinkedList(Node* head) {  
    if (head == NULL) return head;  
    Node* prev = NULL;  
    Node* curr = head;  
    while (curr != NULL) {  
        Node* next = curr->next;  
        curr->next = prev;  
        prev = curr;  
        curr = next;  
    }  
    return prev;
```

Another Recursive approach :-

```
void reverse(Node* &head, Node* curr, Node* prev);
```

// base case

```
? if (curr == NULL) {
```

```
    head = prev;
```

```
    return;
```

```
}
```

```
? Node* forward = curr->next;
```

```
reverse(head, forward, curr);
```

```
Curr->next = prev;
```

```
?
```

```
? Node* reverselinkedlist(Node* head) {
```

```
    Node* curr = head;
```

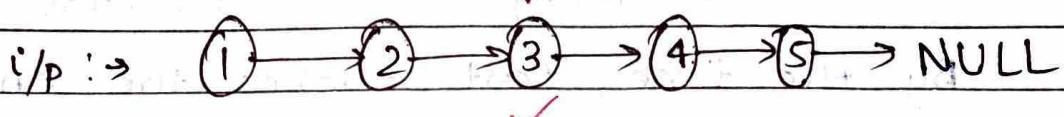
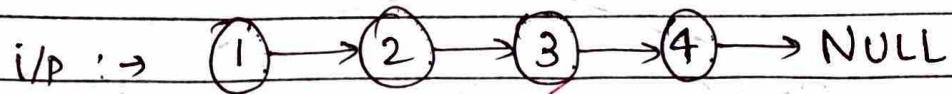
```
    Node* prev = NULL;
```

```
    reverse(head, curr, prev);
```

```
    return head;
```

```
}
```

Q Middle of a linked list



Approach 1:

```
int getLength(Node* head){  
    int len = 0;  
    while (head != NULL) {  
        len++;  
        head = head->next;  
    }  
    return len;  
}
```

```
Node* findMiddle(Node* head){  
    int len = getLength(head);  
    int ans = (len/2);  
  
    Node* temp = head;  
    int cnt = 0;  
    while (cnt < ans) {  
        temp = temp->next;  
        cnt++;  
    }  
    return temp;  
}
```

Approach 2 :- Use 2 pointers

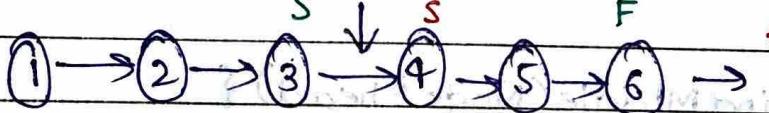
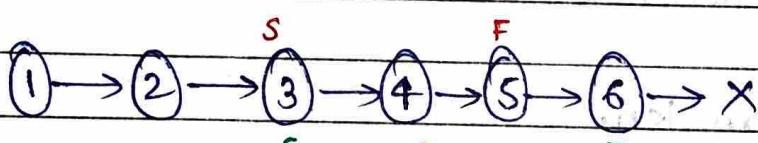
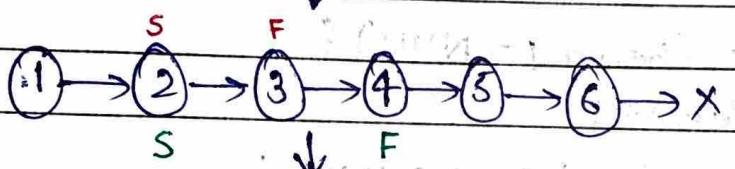
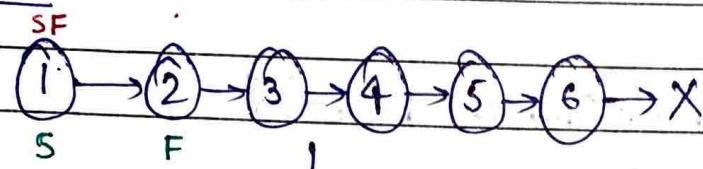
Slow (moves by 1)

fast (moves by 2)

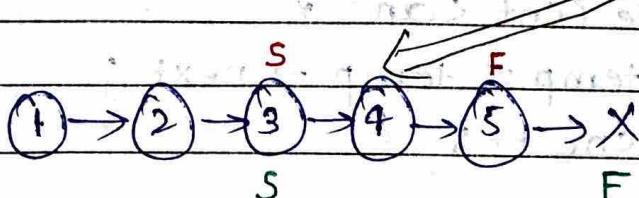
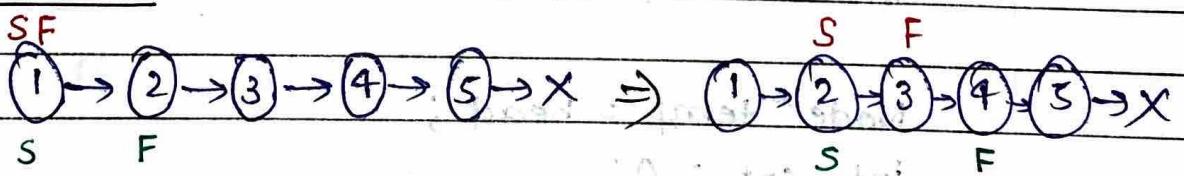
By the time, fast reaches the end,
slow points to the mid.

Even :-

More optimized (2)
Program (2)



Odd :-



Program :- (2)

```
Node* getMiddle (Node* head) {  
    if (head == NULL || head->next == NULL)  
        return head;
```

```
    if (head->next->next == NULL) {  
        return head->next;  
    }
```

```
    Node* slow = head;
```

```
    Node* fast = head->next;
```

```
    while (fast != NULL) {
```

```
        fast = fast->next;
```

```
        if (fast != NULL) {
```

```
            fast = fast->next;
```

```
}
```

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

```
}
```

```
    return slow;
```

```
}
```

```
Node* findMiddle (Node* head) {
```

```
    return getMiddle (head);
```

```
}
```

Important

More Optimized Method :-

```
Node* findMiddle(Node* head) {  
    if (head == NULL)  
        return head;
```

```
    Node* slow = head;
```

```
    Node* fast = head; → next;
```

```
    while (fast != NULL && fast->next != NULL) {
```

```
        fast = fast->next->next;
```

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

```
}
```

```
return slow;
```

```
},
```

i/p : $\Rightarrow [1, 2, 3, 4] \Rightarrow$ o/p = [3, 4]

i/p : $\Rightarrow [1, 2, 3, 4] \Rightarrow$ o/p = [2, 3, 4]

Forward + odd with initial + slow

: (forward) alternating slow, odd, even,

<https://www.linkedin.com/in/nikhilkumar0609/>

Q Reverse list in K-Groups

i/p : $\rightarrow [3] \rightarrow [7] \rightarrow [8] \rightarrow [11] \rightarrow [17] \rightarrow [2] \rightarrow x$

o/p : $K=2$,

$[7] \rightarrow [3] \rightarrow [11] \rightarrow [8] \rightarrow [2] \rightarrow [17] \rightarrow x$

o/p : $K=3$

$[8] \rightarrow [7] \rightarrow [3] \rightarrow [2] \rightarrow [17] \rightarrow [11] \rightarrow x$

Approach 1 :

Algorithm :

→ 1 case solve k range

• Iterative algo (count < k)

↓
extract current node

First K Node reverse

head → next = recursion call

return head of reversed linked list

↓
return (Prev; next);

(head; tail) = front & head

Program:-

```
Node* kReverse (Node* head, int k) {
    // base case
    if (head == NULL) {
        return NULL;
    }

    // Step 1: reverse first k nodes
    Node* next = NULL;
    Node* curr = head;
    Node* prev = NULL;
    int count = 0;

    while (curr != NULL && count < k) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
        count++;
    }

    // step 2: recursion dekh lega aage ka.
    if (next != NULL) {
        head->next = kReverse(next, k);
    }

    // step 3: return head of reversed list
    return prev;
}
```

Q Circularly linked list

(Assume empty list as Circular)

Approach 1 :-

Agar M jis node se start kiya, uss node pe
dubara aa jaati toh circular linked list hai.

Program :-

```
bool isCircularList (Node * head) {  
    //empty list  
    if (head == NULL) {  
        return true;  
  
    Node * temp = head->next; // take head  
    while (temp != NULL && temp != head) {  
        temp = temp->next;  
    }  
  
    if (temp == head) {  
        return true; // means linked  
    }  
    return false; // not linked  
}
```

last of front & last

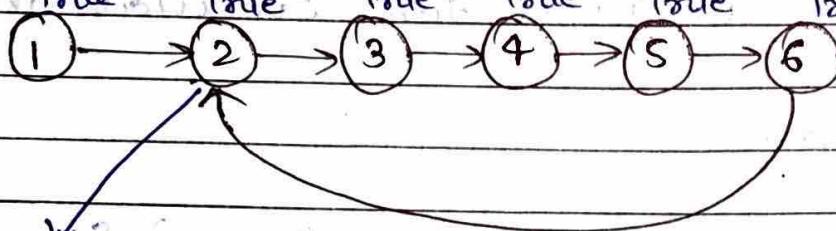
front & front = front

last & last

Q Detect and Remove Loop

→ Detect Loop

Algorithm :-



Yahan Phle se hi true mark kiya hua hai,
mtlb loop exist krta hai.

Program :-

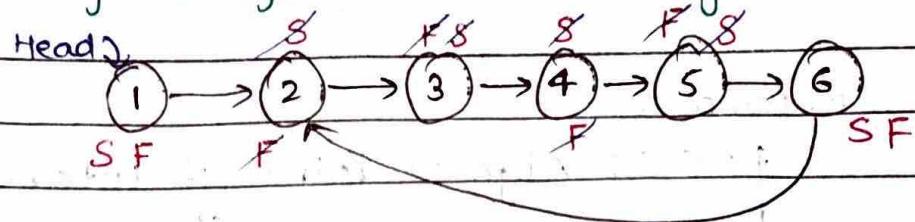
```
bool detectLoop(Node* head) {  
    if (head == NULL)  
        return false;
```

S.C = O(n)
T.C = O(n)

```
map<Node*, bool> visited;  
Node* temp = head;  
while (temp != NULL) {  
    // Cycle is present  
    if (visited[temp] == true)  
        return true;  
    visited[temp] = true;  
    temp = temp->next;  
}  
return false;
```

Note :- In this page, blue pen is used to return true/false and red pen is used for the code to return the node from where the loop is starting

Floyd's Cycle detection Algorithm :-



if (slow == fast) \Rightarrow loop is present

if (fast = NULL) \Rightarrow No Loop

Program :- $(T.C \rightarrow O(n), S.C \rightarrow O(1))$

```
bool * floydDetectLoop(Node* head) {
    Node slow = head;
    Node fast = head;
    while (slow != NULL && fast != NULL) {
        fast = fast->next;
        if (fast == NULL) {
            fast = fast->next;
        }
        slow = slow->next;
    }
}
```

if (slow == fast) {

return true;

slow;

$$(1 + (2 * p^2 + 1) * L = (1 + (2 * 26) + 1) * L)$$

return false;

NULL;

→ Starting Node of Loop

Approach:

→ FCD Algo \Rightarrow Point of intersection
(slow = fast)

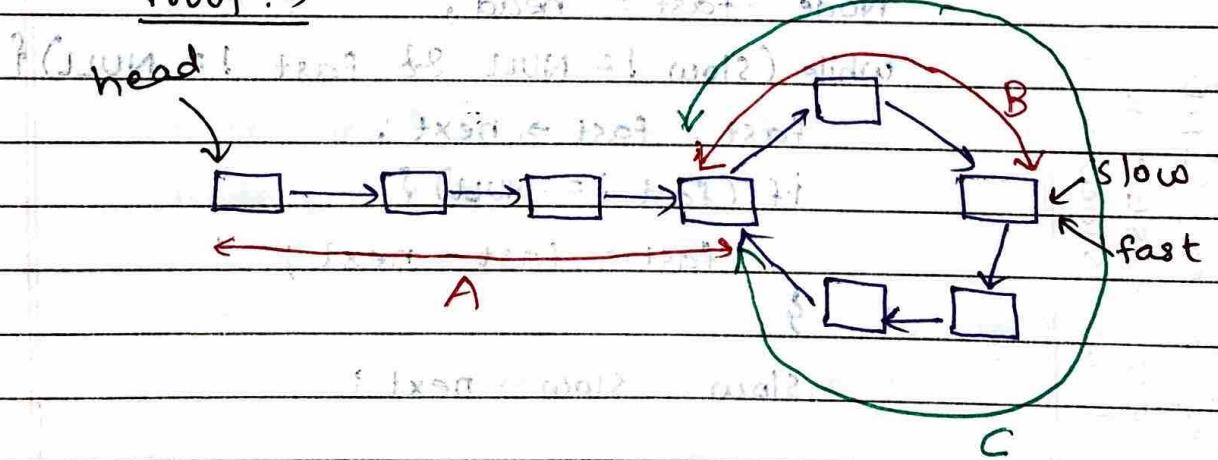
→ slow = head

Move slow & fast at same pace
of '1'.

When (slow == fast)

Starting Point of Loop.

Proof:



Distance by fast pointer = 2 * Distance by slow pointer

$$(A + (x \cdot c) + B) = 2 * (A + (y \cdot c) + B)$$

where, x & y are no. of cycles.

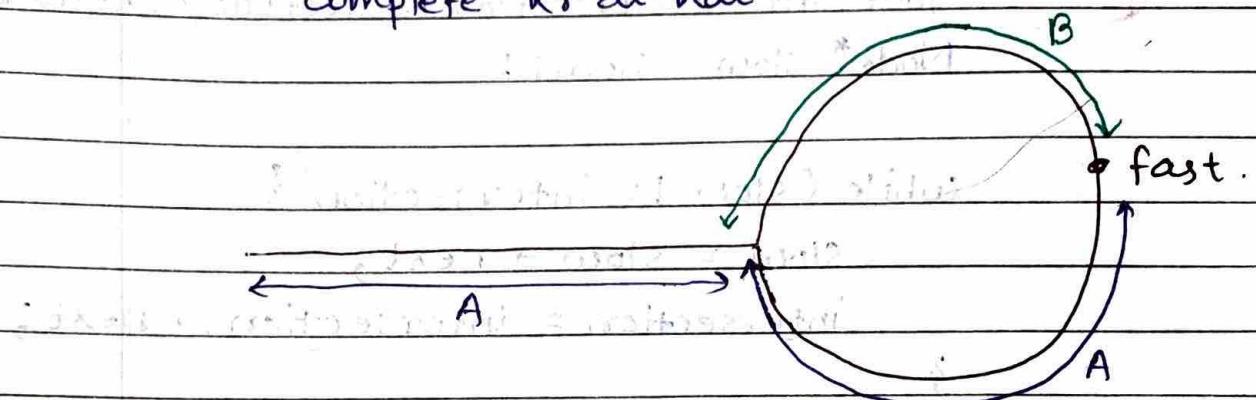
$$\Rightarrow C(x - 2y) = A + B$$

$\underbrace{\qquad\qquad}_{K}$

$$\Rightarrow A + B = K \text{ times } C \cdot i$$

Mtlb, A + B ka mtlb hai ki maine Cycle

Complete kr di hai.



Toh agar main 'B' distance pe hu toh
iss cycle ko complete krne ke liye
'A' distance lgega.

Toh fast pointer aur head se 'i' steps
aage bhnے pe jb (slow = fast) ho jayे
toh whi starting node hogi loop ki.

Fast singh se i short pointers enga
aur astha jaha aur rata hoga aur
correct and accurate question don't ask

if (intersection == NULL) } \rightarrow Jb Loop nhi ho toh
return NULL; starting node kahan
se hoga.

Program :-

```
Node* getStartingNode (Node* head) {  
    if (head == NULL)  
        return NULL;
```

```
    Node* intersection = floydDetectLoop (head);  
    Node* slow = head;
```

```
    while (slow != intersection) {
```

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

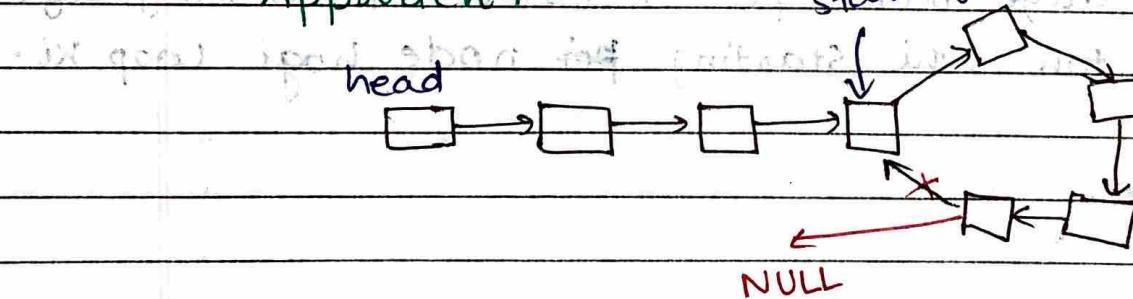
```
        intersection = intersection->next;
```

```
    }
```

```
    return slow;
```

→ Remove Cycle from LL

Simpler Approach :- If : starting Node



Agar Starting Node ke ek phle wala
node ke pointer KO NULL pointer bna
du toh loop remove ho jayega.

```
if (startOfLoop == NULL)
    return NULL;
```

Program :-

```
Node*
Void removeLoop (Node* head) {
    if (head == NULL)
        return NULL;
```

```
    Node* startOfLoop = getStartingNode (head);
    Node* temp = startOfLoop;
```

```
    while (temp->next != startOfLoop) {
        temp = temp->next;
    }
```

```
    temp->next = NULL;
```

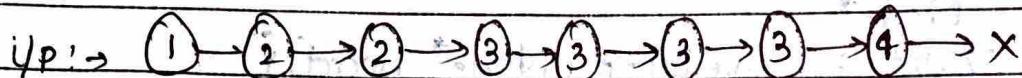
```
    return head;
```

```
}
```

getStartingNode
function to find starting node
of loop

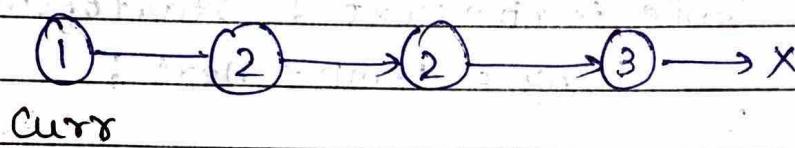
removeLoop
function to remove loop
from linked list

Q Remove Duplicates from Sorted linked list

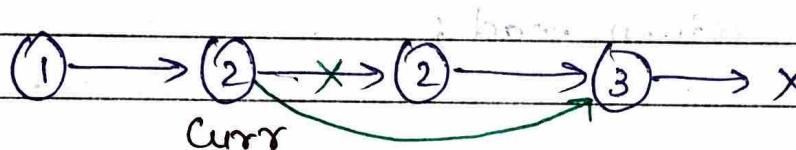


Program :-

Approach :-



Now, $1 \neq 2 \Rightarrow \text{Curr} = \text{Curr} \rightarrow \text{next}$



Now, $2 \neq 2$

$\Rightarrow \text{curr} \rightarrow \text{next}^{\text{next_next}} = \text{curr} \rightarrow \text{next} \rightarrow \text{next}$

And delete $\text{curr} \rightarrow \text{next}$

$\text{curr} \rightarrow \text{next} = \text{next_next}$

Kyuki agar delete ke phle $\text{curr} \rightarrow \text{next}$ ko change krnege toh jo Node delete krna hai usko track nhi kr payenge.

Program :-

```
Node* uniqueSortedList (Node* head) {
    //empty list
    if (head == NULL)
        return NULL;

    //non-empty list
    Node* curr = head;

    while (curr->next != NULL) {
        if (curr->data == curr->next->data) {
            Node* next_next = curr->next->next;
            Node* nodeToDelete = curr->next;
            delete (nodeToDelete);
            curr->next = next_next;
        } else { //Not equal
            curr = curr->next;
        }
    }

    return head;
}
```

$$T.C = O(n)$$

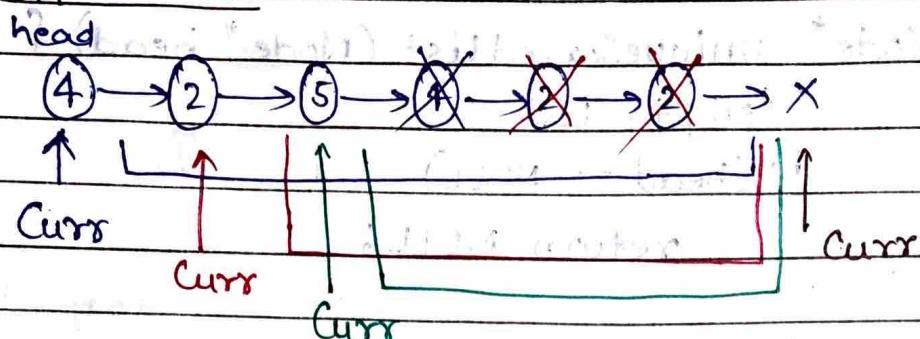
$$S.C = O(1)$$

GFG

<https://www.linkedin.com/in/nikhilkumar0609/>

Q Remove duplicates from Unsorted linked list

Approach 1 :-



• Har ek element ke liye aage ki linked list traverse kr rhe hai, aur jahan jahan mil tha wo element delete krte ja rhe
if curr != NULL

$$T.C = O(n^2), S.C = O(1)$$

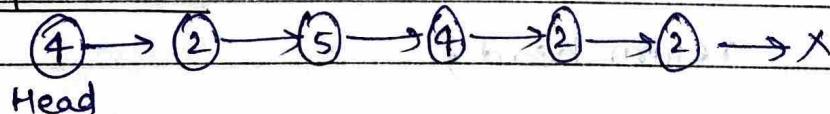
Approach 2 :-

Step 1 :- Sort linked list $\rightarrow O(n \log n)$

Step 2 :- Previous qstn $\rightarrow O(n)$

$$T.C \rightarrow O(n \log n), S.C \rightarrow O(1)$$

Approach 3 :-

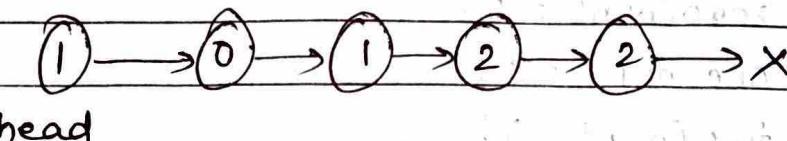


map <Node*, bool> visited

$$T.C \rightarrow O(n), S.C \rightarrow O(n)$$

linked list of
⑧ Sort ↑ 0s, 1s, 2s

Approach:-



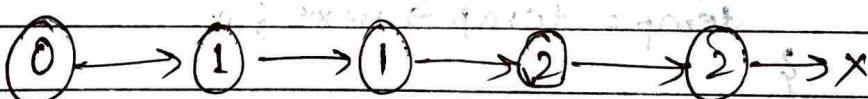
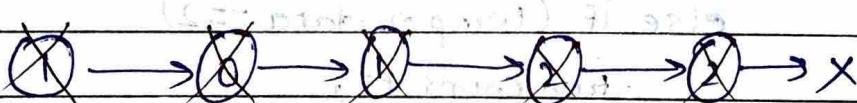
Algo :- Count 0, 1, 2

No. Count

0 → 1

1 → 2

2 → 2



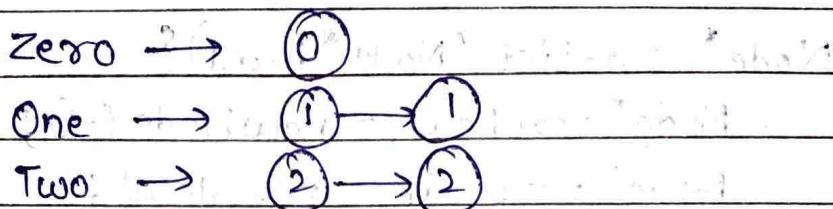
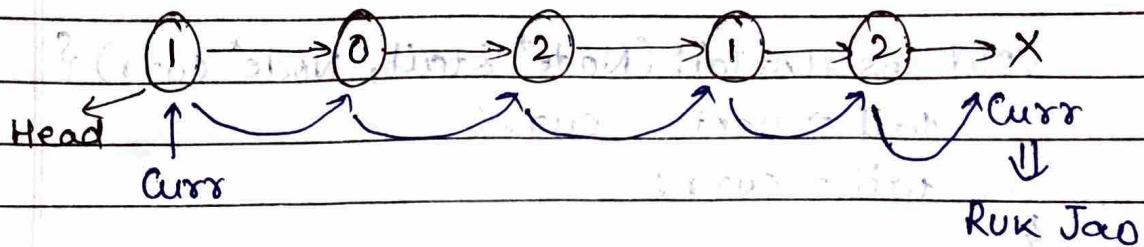
T.C → O(n), S.C → O(1)

Phle saare no. ko count kr liye fir data ko replace kr de rhe.

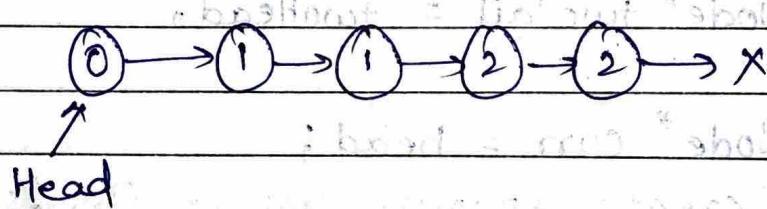
Program:-

```
Node* SortList (Node* head) {
    int zeroCount = 0;
    int oneCount = 0;
    int twoCount = 0;
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == 0)
            zeroCount++;
        else if (temp->data == 1)
            oneCount++;
        else if (temp->data == 2)
            twoCount++;
        temp = temp->next;
    }
    temp = head;
    while (temp != NULL) {
        if (zeroCount != 0) {
            if (temp->data == 0)
                zeroCount--;
        }
        else if (oneCount != 0) {
            temp->data = 1;
            oneCount--;
        }
        else if (twoCount != 0) {
            temp->data = 2;
            twoCount--;
        }
        temp = temp->next;
    }
    return head;
}
```

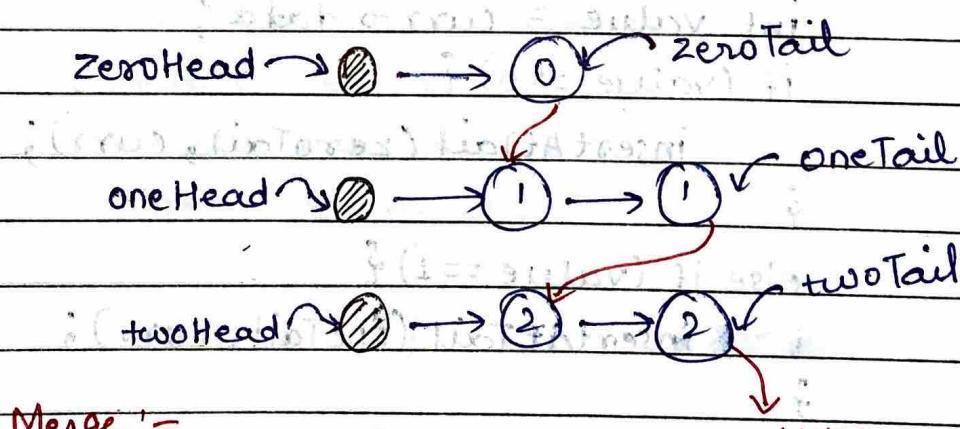
Approach 2 :- When Data Replacement isn't allowed.



3 alag-alag linked list bn gye, ab Teeno ko merge kar dena haisa hota hai



→ Create dummy nodes and links



T.C → O(n), S.C → O(1)

Program :-

```
void insertAtTail (Node* &tail, Node* curr) {
```

```
    tail->next = curr;
```

```
    tail = curr;
```

```
} }
```

```
Node* sortlist (Node* head) {
```

```
    Node* zeroHead = newNode (-1);
```

```
    Node* zeroTail = zeroHead;
```

```
    Node* oneHead = newNode (-1);
```

```
    Node* oneTail = oneHead;
```

```
    Node* twoHead = newNode (-1);
```

```
    Node* twoTail = twoHead;
```

```
    Node* curr = head;
```

```
// create separate list of 0s, 1s, 2s
```

```
while (curr != NULL) {
```

```
    int value = curr->data;
```

```
    if (value == 0) {
```

```
        insertAtTail (zeroTail, curr);
```

```
}
```

```
    else if (value == 1) {
```

```
        insertAtTail (oneTail, curr);
```

```
}
```

```
    else if (value == 2) {
```

```
        insertAtTail (twoTail, curr);
```

```
}
```

```
    curr = curr->next;
```

```
}
```

// merge 3 sublist

// 1st list not empty

```
if (oneHead->next != NULL) {  
    zeroTail->next = oneHead->next;  
}
```

else {

// 1st list is empty

```
    zeroTail->next = twoHead->next;  
}
```

oneTail->next = twoHead->next;

twoTail->next = NULL;

// setup head

head = zeroHead->next;

// delete dummy nodes

delete zeroHead;

delete oneHead;

delete twoHead;

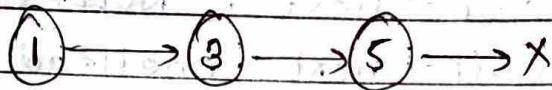
return head;

8

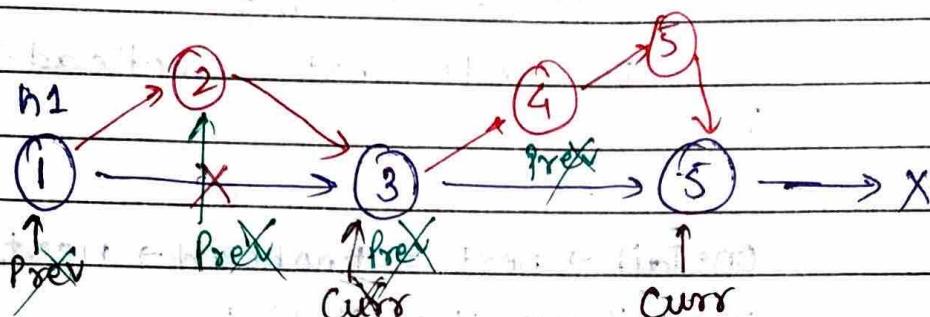
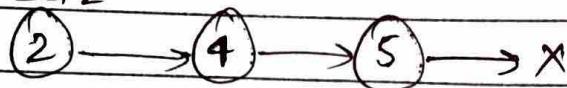
Merge 2 Sorted linked list

Approach :-

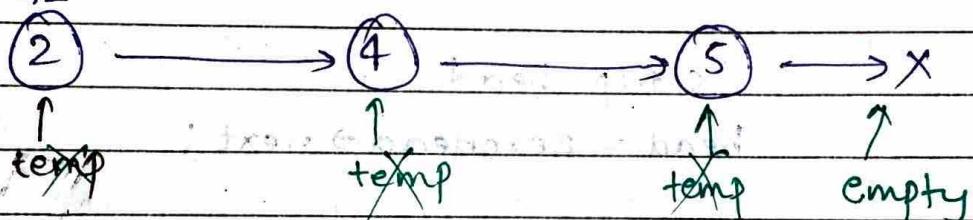
head1



head2



h2



Prev → data ≤ temp → data ≤ curr → data

TRUE

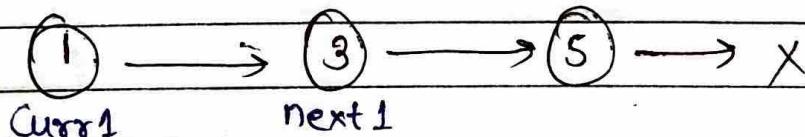
FALSE

daal do Node bich mein

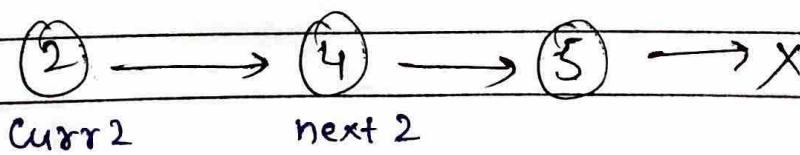
aur pointers update
kr do.

Curr pointer
aage bda do

first =



Second =



Program :-

```
Node<int>* solve(Node<int>* first, Node<int>* second) {
    //if only one element is present in first list
    if (first->next == NULL) {
        first->next = second;
        return first;
    }
```

```
Node<int>* curr1 = first;
```

```
Node<int>* next1 = curr1->next;
```

```
Node<int>* curr2 = second;
```

```
Node<int>* next2 = curr2->next;
```

```
while (next1 != NULL && curr2 != NULL) {
```

```
    if ((curr2->data >= curr1->data) &&
        (curr2->data <= next1->data)) {
```

```
        //add node in b/w the node of first list
```

```
        curr1->next = curr2;
```

```
        next2 = curr2->next;
```

```
        curr2->next = next1;
```

```
i (curr1, curr2) sv102
```

```
//updating pointer
```

```
        curr1 = curr2;
```

```
        curr2 = next2;
```

```
}
```

else { // go one step ahead in first list

curr1 = next1;

next1 = next1->next;

if (next1 == NULL) {

curr1->next = curr2;

return first;

}

}

return first;

}

Node<int>* sortTwoLists (Node<int>* first, Node<int>* second) {

if (first == NULL) {

return second;

if (second == NULL)

return first;

if (first->data <= second->data) {

solve (first, second);

else {

solve (second, first);

}

Q Check palindrome in a Linked List

Approach :-

Step 1 :- Create an array

Step 2 :- Copy L.L content into array.

Step 3 :- Write logic to check palindrome in array.

T.C \rightarrow $O(N)$, S.C \rightarrow $O(N)$

Program :-

→ for copying L.L content into array

```
bool isPalindrome (Node* head)
```

```
{
```

```
vector<int> arr;
```

```
Node* temp = head;
```

```
while (temp != NULL) {
```

```
arr.push_back (temp->data);
```

```
temp = temp->next;
```

```
}
```

```
if (arr == arr) return 1;
```

```
return 0;
```

```
}
```

is 3rd step

→ Step '3' ka function

fixed & change Jaise hum kih skte hai.

Approach 2:-

	T.C
Step 1:- find middle	$O(N)$
Step 2:- reverse L.L after it	$O(N)$
Step 3:- compare both halves	$O(N)$
Step 4:- repeat step 2.	$O(N)$

$$T.C \rightarrow O(N), S.C \rightarrow O(1)$$

Program :-

```
bool isPalindrome (Node* head) {
    if (head == NULL || head->next == NULL)
        return true;
    // Step 1 :- find middle
    Node* middle = getMid(head);
    // Step 2 :- reverse L.L after middle
    Node* temp = middle->next;
    middle->next = reverse(temp);
    // Step 3 :- Compare both halves
    Node* head1 = head;
    Node* head2 = middle->next;
    while (head2 != NULL) {
        if (head1->data != head2->data)
            return false;
        head1 = head1->next;
        head2 = head2->next;
    }
    // Step 4 :- repeat step 2. (Optional)
    temp = middle->next;
    middle->next = reverse(temp);
    // Yahan tk aa gya toh true hoga
    return true;
}
```

~~first = [2, 4, 3], second = [5, 6, 4]~~

~~0/p = [7, 0, 8]~~

~~Explanation $\Rightarrow 342 + 465 = 807$~~

Add two numbers represented by linked list.

Algorithm: \rightarrow Step 1: \rightarrow Reverse both LL

Step 2: \rightarrow Add them from left

Step 3: \rightarrow Reverse ans.

Program: \rightarrow

Node* reverse (Node* head) {

}

function to reverse linked list

Void insertAtTail (struct Node*& head, struct Node*& tail, int val) {

}

function to insert at tail

Struct Node* add (struct Node* first, struct Node* second) {

int carry = 0;

Node* ansHead = NULL; // ans = answer

Node* ansTail = NULL; // ans = answer

while (first != NULL || second != NULL || carry != 0) {

int val1 = 0; // ans = answer

if (first != NULL)

val1 = first \rightarrow data;

i (ans) answer = 200

int val2 = 0;

if (second != NULL)

val2 = second \rightarrow data;

int sum = carry + val1 + val2;

int digit = sum % 10;

// create node and add in ans LL
insertAtTail (ansHead, ansTail, digit);

Carry = sum/10;

if (first != NULL)

 first = first -> next;

if (second != NULL)

 second = second -> next;

}

return ansHead;

public:

struct Node* addTwoLists (struct Node* first, struct Node* second) {

// Step 1 → reverse input LL

 first = reverse (first);

 second = reverse (second);

// step2 → add 2 LL

 Node* ans = add (first, second);

(ans = 1. Node of ans)

// step3 → reverse ans

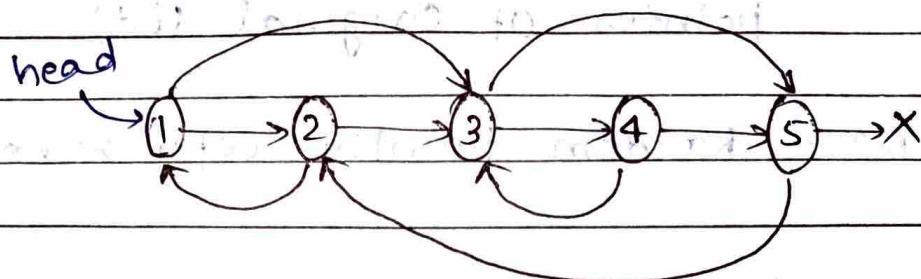
 ans = reverse (ans);

return ans;

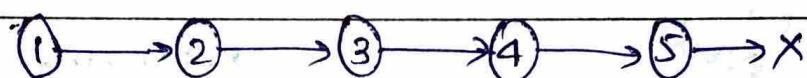
}

Q) Clone a linked list with next and random pointers.

Approach 1 :-



Step 1:- Create a clone list (using Original list next pointer) $O(n)$



Step 2:- Ab bss random pointer ko copy krna bcha hai.

Toh M hr node ke liye dekhunga ki uska random pointer kitna dus hai aur usko point krwa dunga, fir next node pe jaunga.

for (Original list mein chlund she ki)
uska random ptr. kitna distance ph.

q.

$\Theta(n^2)$ while (Jabtک main clone wale list ko iske random ptr. ko slie jgh nahi lgा deta).

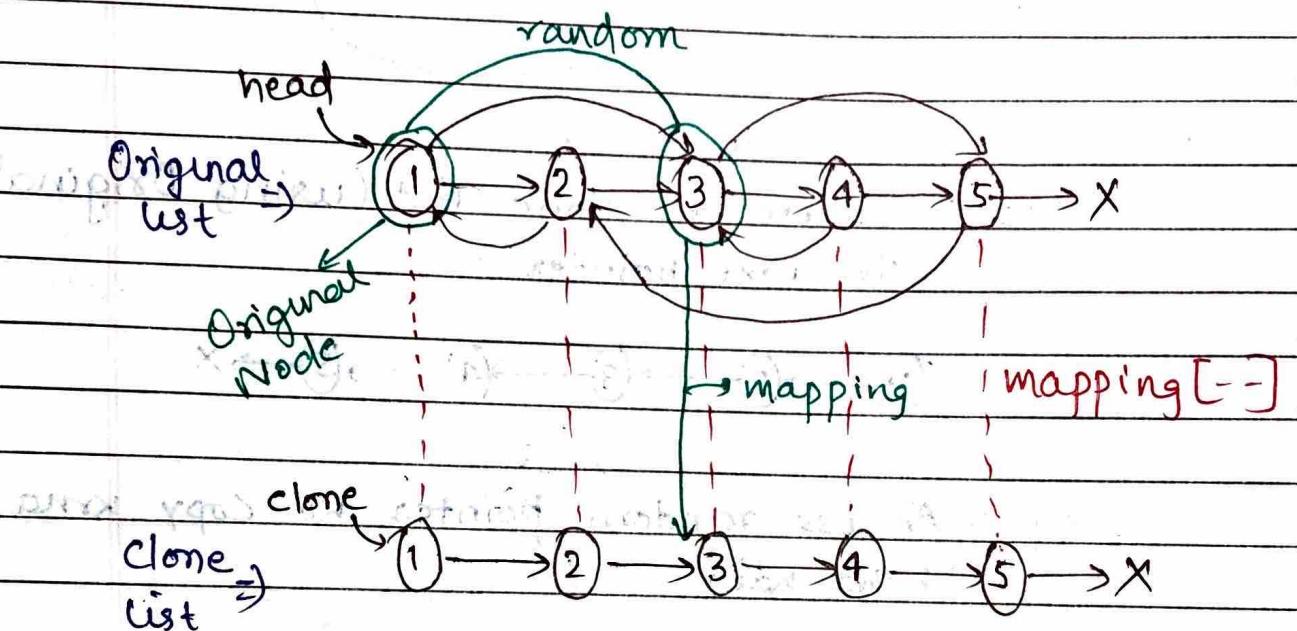
• § 3.
q;

T.C $\rightarrow \Theta(n^2)$

Approach 2:-

Step 1:- Create a clone list (using next pointer of Original list)

Step 2:- Random pointer copy kرنے hai



T.C $\rightarrow O(N)$

S.C $\rightarrow O(N)$, because of map.

Program :-

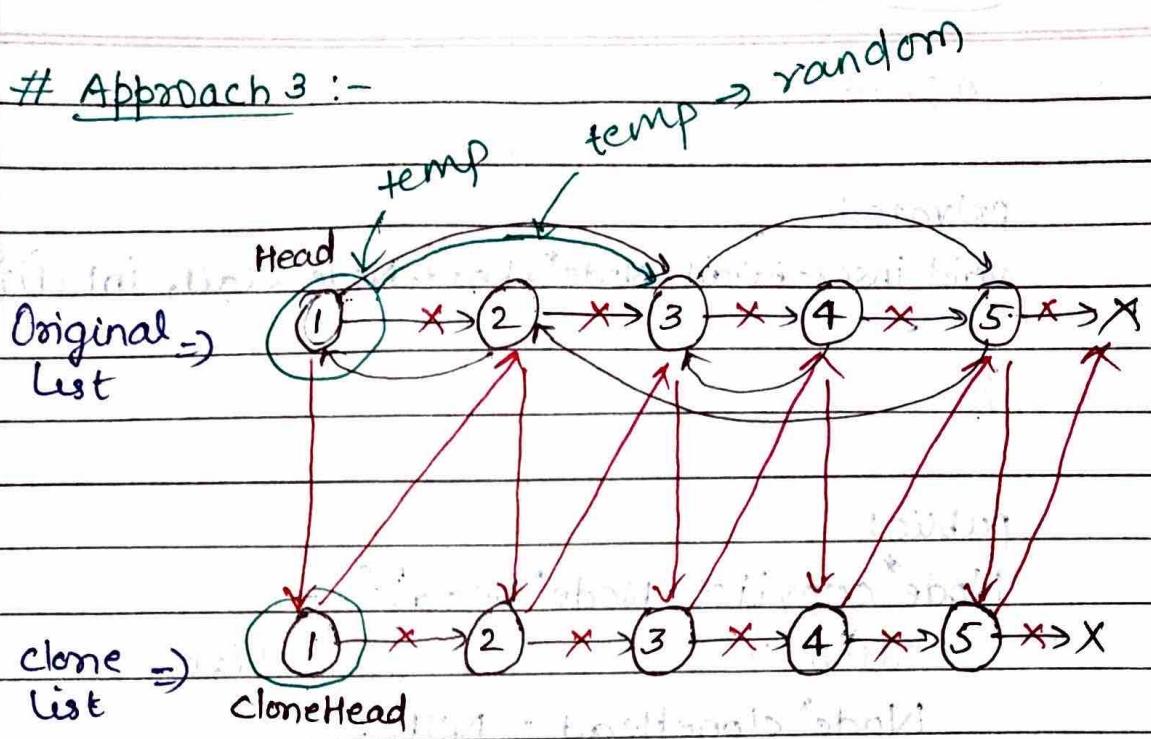
```
private:  
void insertAtTail(Node*&head, Node*&tail, int d){  
    Node* temp = new Node(d);  
    if(tail == NULL){  
        head = temp;  
        tail = temp;  
    } else {  
        tail->next = temp;  
        tail = temp;  
    }  
}  
  
public:  
Node* copyList(Node* head){  
    //create a clone list (step 1)  
    Node* cloneHead = NULL;  
    Node* cloneTail = NULL;  
  
    Node* temp = head;  
    while(temp != NULL){  
        insertAtTail(cloneHead, cloneTail, temp->data);  
        temp = temp->next;  
    }  
  
    //step 2 :- create a map  
    unordered_map<Node*, Node*> oldToNewNode;  
  
    Node* originalNode = head;  
    Node* cloneNode = cloneHead;  
    while(originalNode != NULL && cloneNode != NULL){  
        oldToNewNode[originalNode] = cloneNode;  
        originalNode = originalNode->next;  
        cloneNode = cloneNode->next;  
    }  
}
```

```
// set random pointer
originalNode = head;
cloneNode = cloneHead;

while (originalNode != NULL) {
    cloneNode->random = oldToNewNode[originalNode
        → random];
    originalNode = originalNode->next;
    cloneNode = cloneNode->next;
}

return cloneHead;
```

Approach 3 :-



Step 1:- create a clone list

Step 2:- Add cloneNodes in b/w Original list.

Step 3:- Random pointer (clone list)

temp \rightarrow next \rightarrow random = temp \rightarrow random \rightarrow next

Step 4:- revert changes done in Step 2.

Step 5:- return ans! (cloneHead)

Program :-

private :

```
void insertAtTail(Node*& head, Node*& tail, int d){  
    Node* temp = new Node(d);  
    if(tail == NULL){  
        head = temp;  
        tail = temp;  
    } else {  
        tail->next = temp;  
        tail = temp;  
    }
```

public :

```
Node* copyList(Node* head){
```

//Step 1:- Create a clone list

```
Node* cloneHead = NULL;
```

```
Node* cloneTail = NULL;
```

for (Node* temp = head; temp != NULL; temp = temp->next){

Node* cloneTemp = new Node(temp->data);

```
    if(cloneHead == NULL){  
        cloneHead = cloneTemp;  
        cloneTail = cloneTemp;  
    } else {  
        cloneTail->next = cloneTemp;  
        cloneTail = cloneTemp;  
    }
```

temp = temp->next;

```
}
```

//Step 2:- Add cloneNodes in b/w Original list

```
Node* originalNode = head;
```

```
Node* cloneNode = cloneHead;
```

```
while (originalNode != NULL && cloneNode != NULL){
```

```
    Node* next = originalNode->next;
```

```
    originalNode->next = cloneNode;
```

```
    originalNode = next;
```

```
    next = cloneNode->next;
```

```
    cloneNode->next = originalNode;
```

```
    cloneNode = next;
```

```
}
```

//Step 3:- random pointer copy

temp = head;

```
while (temp != NULL) {  
    if (temp->next != NULL) {  
        temp->next->random = temp->random;  
        if (temp->random->next != temp->random)  
            temp = temp->next->next;  
    }  
}
```

//step 4 :- revert changes done in step 2

originalNode = head;

cloneNode = cloneHead;

```
while (originalNode != NULL && cloneNode != NULL) {
```

originalNode->next = cloneNode->next;

originalNode = originalNode->next;

```
    if (originalNode != NULL) {
```

cloneNode->next = originalNode->next;

}

cloneNode = cloneNode->next;

}

//step 5 :- return ans

return cloneHead;

}

<https://www.linkedin.com/in/nikhilkumar0609/>

8 Merge Sort in Linked list

```
Node* findMid (Node* head) {
```

```
    Node* slow = head;
```

```
    Node* fast = head->next;
```

```
    while (fast != NULL && fast->next != NULL) {
```

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

```
        fast = fast->next->next;
```

```
}
```

```
    return slow;
```

```
}
```

```
Node* merge (Node* left, Node* right) {
```

```
    if (left == NULL)
```

```
        return right;
```

```
    if (right == NULL)
```

```
        return left;
```

```
    Node* ans = new Node(-1);
```

```
    Node* temp = ans;
```

```
//merge 2 sorted L-L
```

```
    while (left != NULL && right != NULL) {
```

```
        if (left->data < right->data) {
```

```
            temp->next = left;
```

```
            temp = left;
```

```
            left = left->next;
```

```
}
```

```
        else { temp->next = right; }
```

```
            temp = right;
```

```
            right = right->next;
```

```
}
```

```
}
```

```

        right
while (left != NULL) {
    temp → next = right;
    temp = right;
    right = right → next;
}
        left
while (right != NULL) {
    temp → next = left;
    temp = left;
    left = left → next;
}
ans = ans → next;
return ans;
}

```

```

Node* mergeSort (Node* head) {
    // base case
    if (head == NULL || head → next == NULL)
        return head;
    // break LL into two halves, after finding Mid
    Node* mid = findMid(head);
    Node* left = head;
    Node* right = mid → next;
    mid → next = NULL;
    // recursive calls to sort both halves
    left = mergeSort(left);
    right = mergeSort(right);
    // merge both left and right halves
    Node* result = merge(left, right);
    return result;
}

```

8 Delete Node in a linked list (Leetcode 237)

NOTE: → you will not be given access to the head of list, instead only the node to be deleted directly.

Program:-

```
void deleteNode(Node* node) {  
    *node = *node -> next;  
}
```

(or)

```
node -> val = node -> next -> val;
```

```
node -> next = node -> next -> next;
```