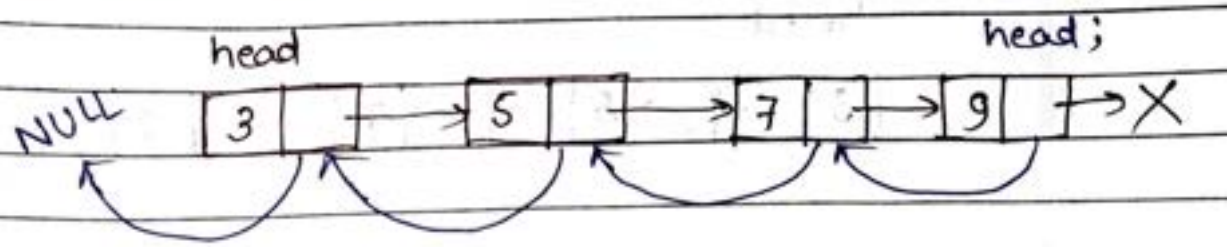
Linked List

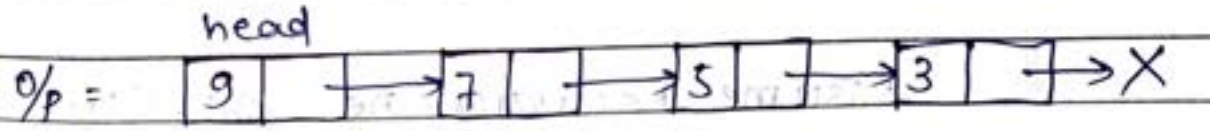
-By Nikhil Kumar

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

LINKED LIST

Q: Reverse linked list

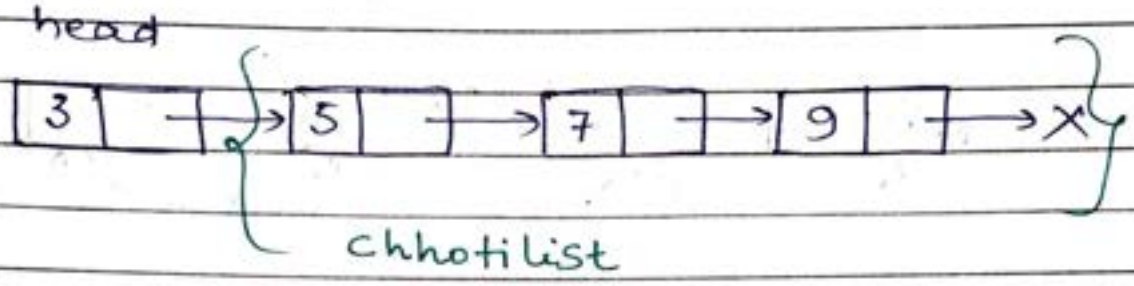




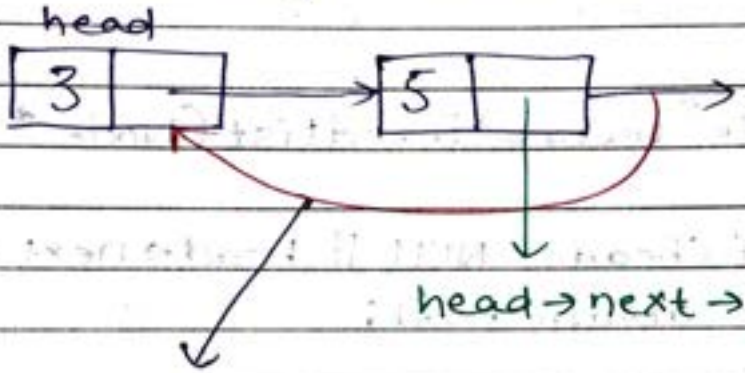
Program:-

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

  Node* Prev = NULL;
  Node * curr = head;

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

→ Recursive approach



Assume Recursion ne chhotilist ko reverse kr diya, uske bad



head → next → next = head;

(9 → 7 → 5 → 3)

↓ chhotaHead

uske baad, head → next = NULL;

(9 → 7 → 5 → 3 → X)

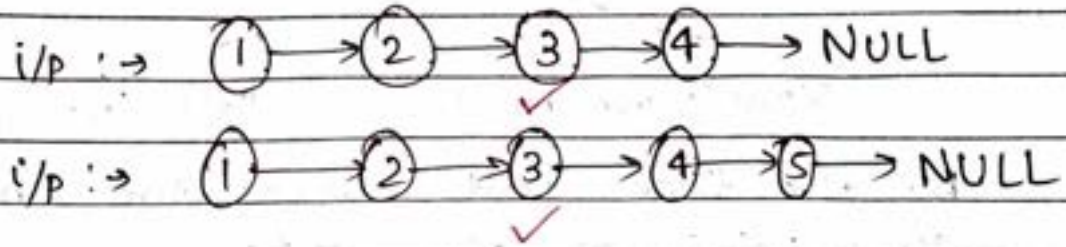
Program :-

```
Node* reverse (Node* head) {

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

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

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



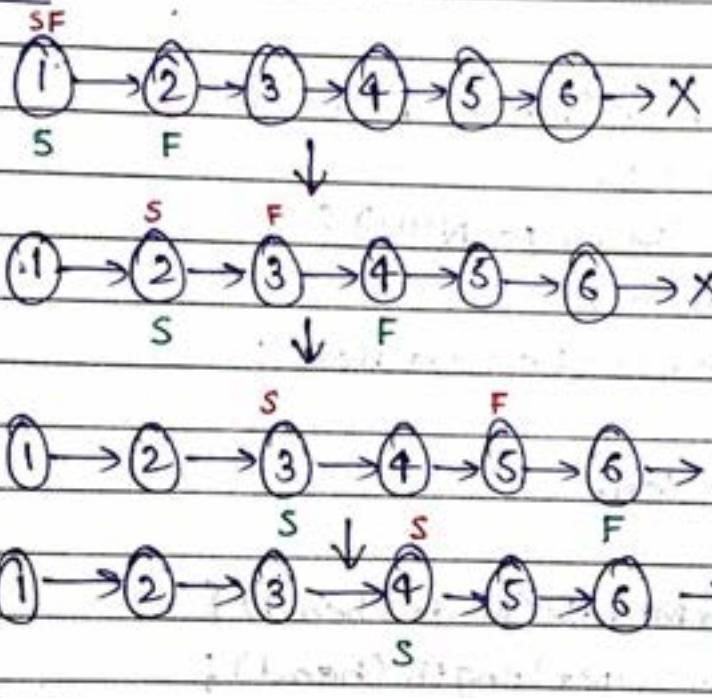
Approach 1 :→

```
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

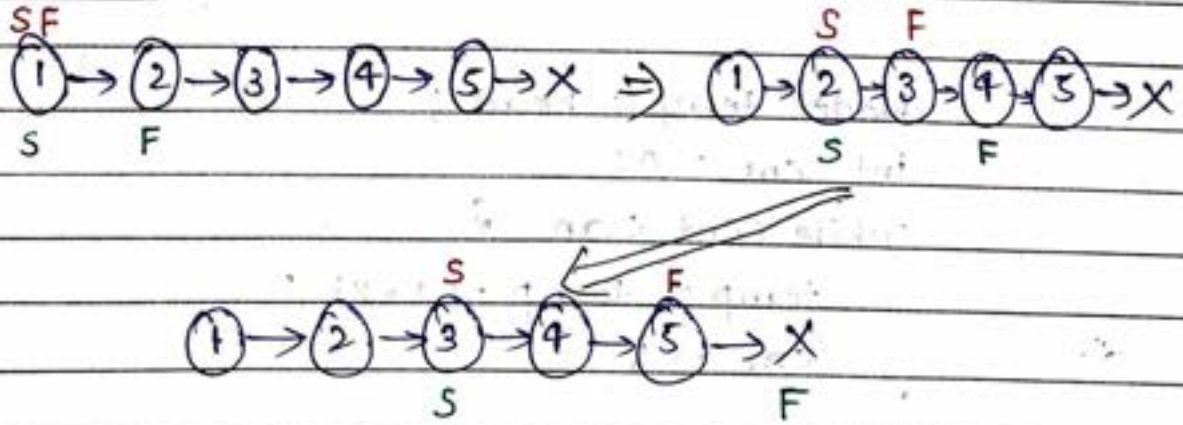
By the time, fast reaches the end, slow points to the mid.

Even :→



→ More optimized program (↑)

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

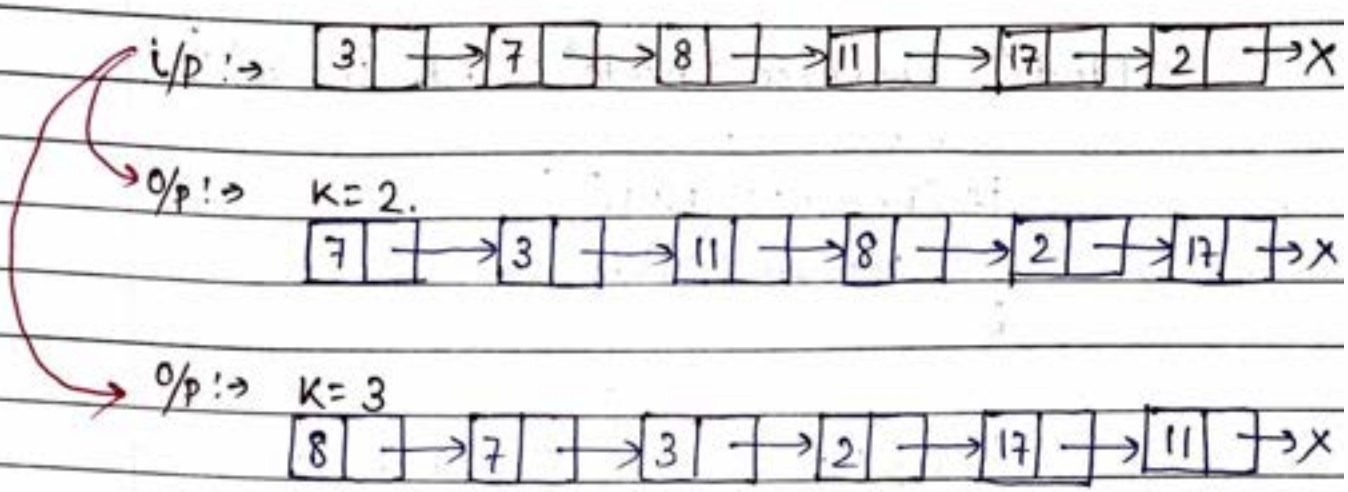
More Optimized Method :-

```
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 :→ [1, 2, 3, 4] ⇒ o/p = [3, 4]

i/p :→ [1, 2, 3, 4] ⇒ o/p = [2, 3, 4]

Q Reverse list in K-Groups



Approach 1 :->

Algorithm :->

- 1 case solve krnge
  - Iterative algo (count < K)
  - ⇓
  - First K Node reverse
- head -> next = recursion call
- return head of reversed Linked list
  - ⇓
  - return Prev;

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 jaau toh circular linked list hai.

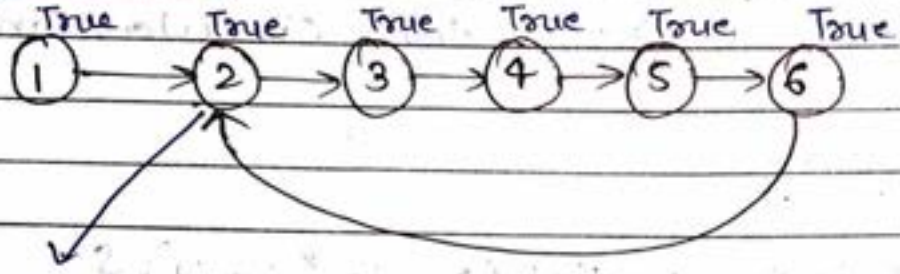
Program :->

```
bool isCircularlist (Node * head) {
    //empty list
    if (head == NULL) {
        return true;

    Node* temp = head -> next;
    while (temp != NULL && temp != head) {
        temp = temp -> next;
    }

    if (temp == head) {
        return true;
    }
    return false;
}
```

Q Detect and Remove Loop

→ Detect Loop

Algorithm 1 :-



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

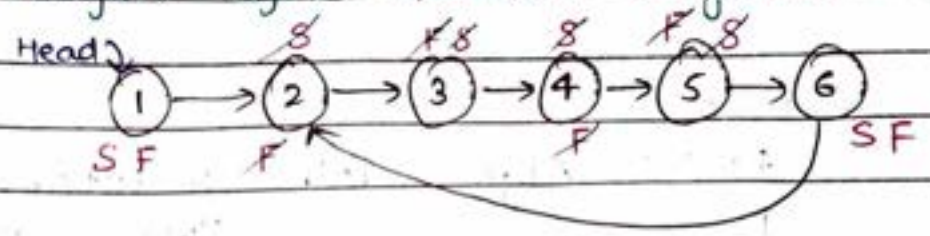
Program :-

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

    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;
}
```

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

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) ⇒ loop is present

if (fast = NULL) ⇒ No loop

Program :- (T.C → O(n), S.C → O(1))

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

    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;
    }
    return false;
}                NULL;
```

while (fast → next != NULL && fast → next → next != NULL) {
    fast = fast → next → next;
    slow = slow → next;

→ Starting Node of Loop

Approach :→

→ FCD Algo ⇒ Point of intersection (slow == fast)

→ slow = head

⇓

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

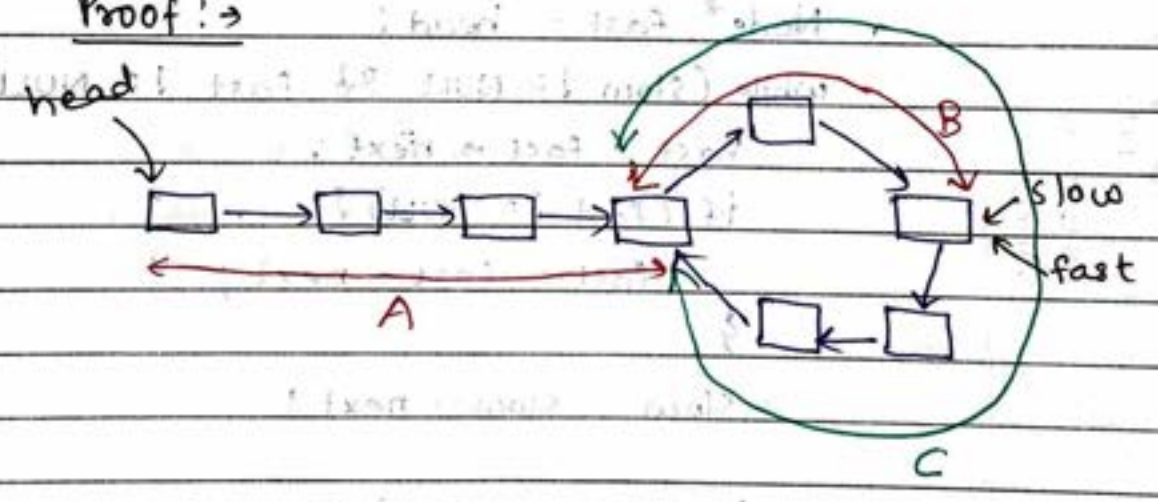
⇓

When (slow == fast)

⇓

Starting Point of loop.

Proof :→



Distance by fast pointer = 2 * Distance by slow Pointer

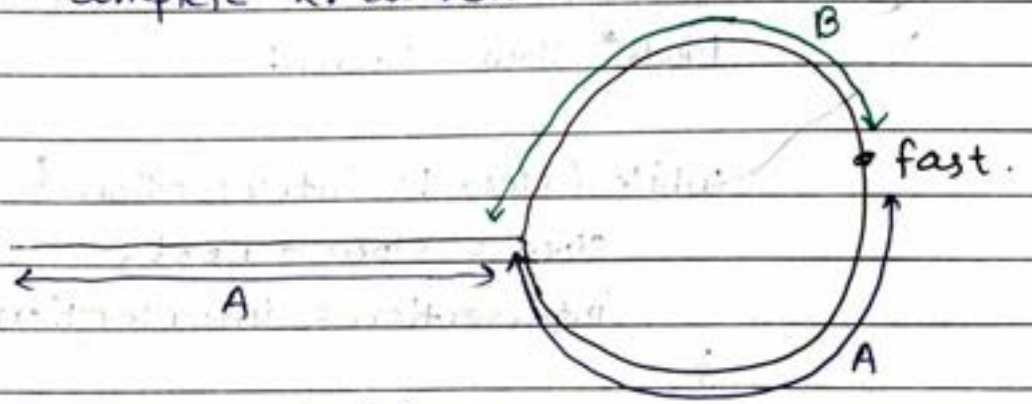
$(A + (x * C) + B) = 2 * (A + (y * C) + B)$

where, x & y are no. of cycles.

$\Rightarrow C(\underbrace{x-2y}_{\to K}) = A + B$

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

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 bdhne pe Jb (slow = fast) ho jaaye toh whi starting ~~pe~~ node hogi loop ki.

if (intersection == NULL)
    return NULL;

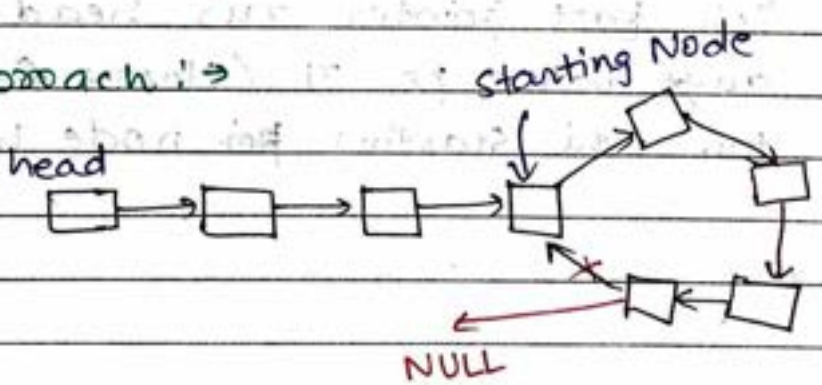
→ Jb loop nhi ho toh starting node kahan se hoga.

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 L.L

Approach :→



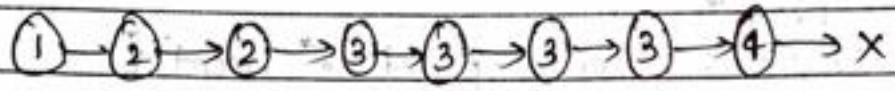
Agar starting Node ke ek phle wala node ke pointer ko NULL pointer bna du toh loop remove ho jayega.

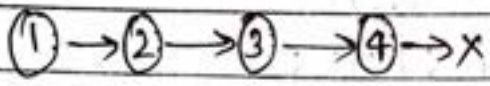
```
if (startOfLoop == NULL)
    return NULL;
```

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;
}
```

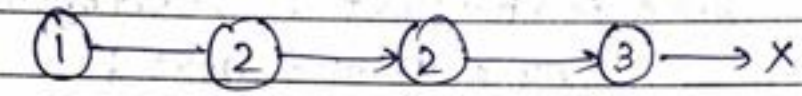
8 Remove Duplicates from Sorted Linked list

i/p :→ 

o/p :→ 

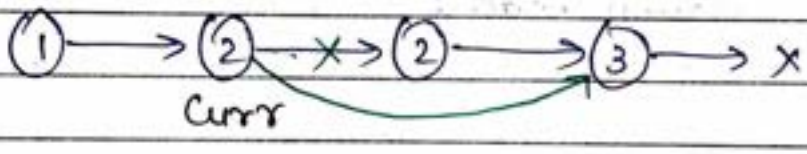
Program :-

Approach :→



Curr

Now, 1 != 2 ⇒ Curr = Curr → next



Now, 2 == 2

⇒ ~~Curr → next~~ next_next = Curr → next → next

And delete Curr → next

Curr → next = next_next

Kyuki agar delete ke phle Curr → next ko change krnge toh jo Node delete krna hai usko track nhi kr payenge.

Program :-

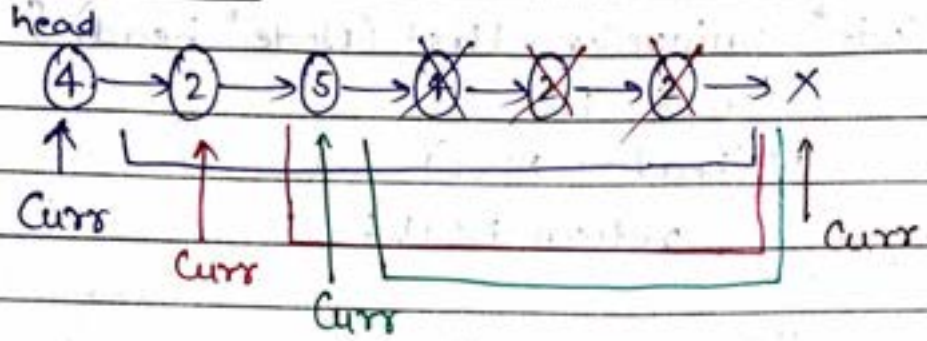
```
Node* uniqueSortedList (Node* head) {
    //empty list
    if (head == NULL)
        return NULL;

    // 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)

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 rha wo element delete krte ja rhe jbtk (curr != NULL)

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

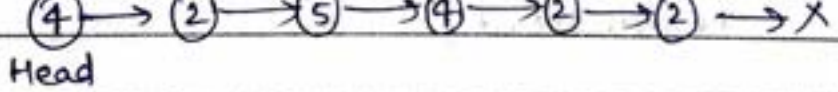
Approach 2 :→

Step 1 :- Sort Linked list → $O(n \log n)$

Step 2 :→ Previous qstn → $O(n)$

T.C → $O(n \log n)$, S.C → $O(1)$

Approach 3 :→



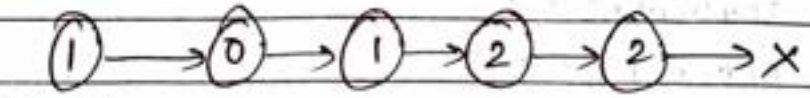
map < Node*, bool > visited

T.C → $O(n)$, S.C → $O(n)$

Linked list of

Q Sort ↑ 0s, 1s, 2s.

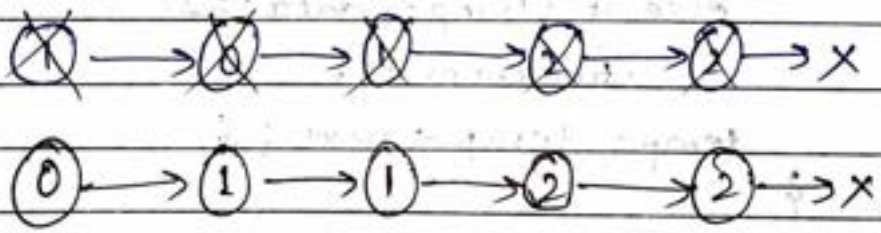
Approach1:-



head

Algo :→ Count 0, 1, 2

No.	Count
0 →	1
1 →	2
2 →	2



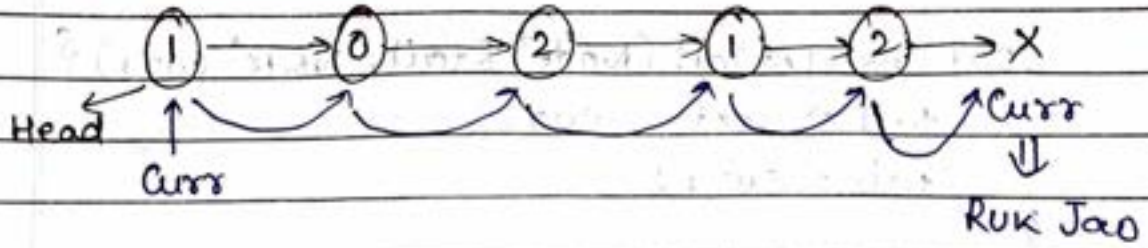
T.C → O(n), S.C → O(1)

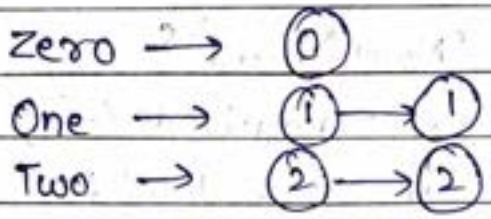
Phle saare no. ko count kr liye fir data ko replace kr de rhe.

Program:-

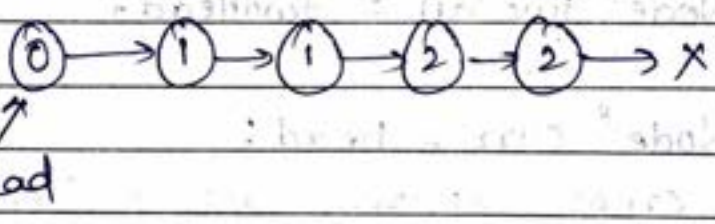
```
Node* SortList (Node *head) {
    int zeroCount = 0;
    int oneCount = 0;
    int twoCount = 0;
    Node* temp = head;
    while (temp != NULL) {
        if (temp -> data == 0)
            zeroCount ++;
        else if (temp -> data == 1)
            oneCount ++;
        else if (temp -> data ==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

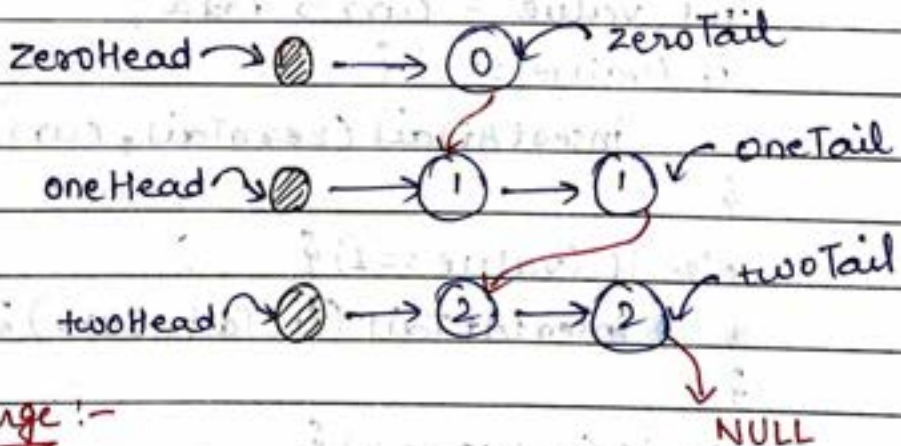




3 alag-alag linked list bn gya, ab Teeno ko merge kr dena hai.



→ Create dummy nodes.



Merge :-

T.C → O(n), S.C → O(1)

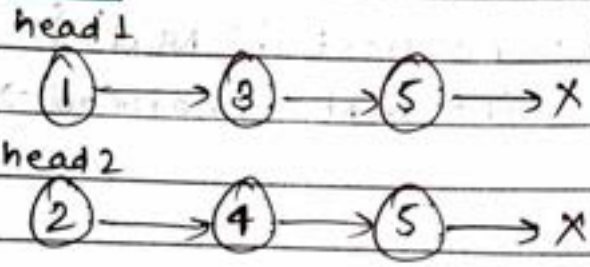
Program :-

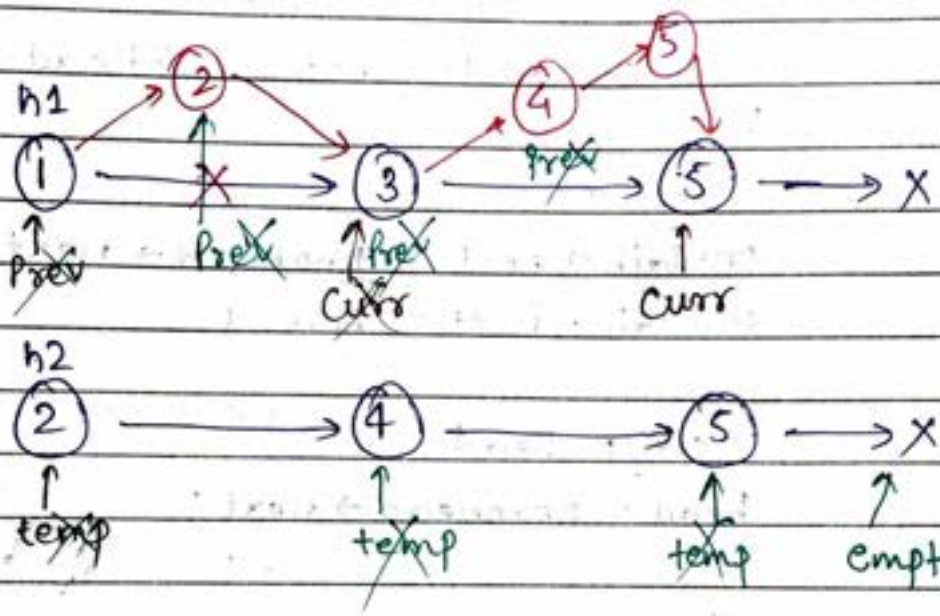
```
void insertAtTail (Node* &tail, Node* curr) {
    tail -> next = curr;
    tail = curr;
}

Node* sortList (Node* head) {
    Node* zeroHead = newNode (-1);
    Node* zeroTail = zeroHead;
    Node* oneHead = new Node (-1);
    Node* oneTail = oneHead;
    Node* twoHead = new Node (-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

//1s list not empty
if (oneHead -> next != NULL) {
    zeroTail -> next = oneHead -> next;
}
else {
    //1s 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 :⇒





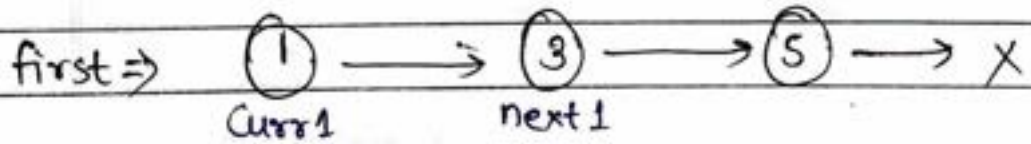
Prev → data ≤ temp → data ≤ Curr → data

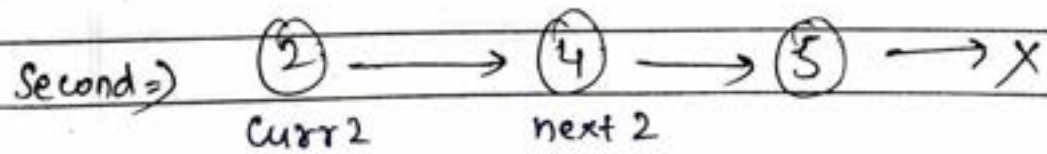
TRUE

FALSE

daal do Node bich mein aur pointers update Kr do.

Curr pointer aage bdha do.





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;

            //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 1:-

step 1 :- create an array.
step 2 :- copy L.L content into array.
step 3 :- write logic to check palindrome in array.

T.C → O(N), S.C → 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;
    }
    return checkPalindrome (arr);
}
```

→ step 3 ka function jo hm likh 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 → O(N), S.C → 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]

o/p = [7,0,8].

Explanation ⇒ 342 + 465 = 807

GFG

Q. Add two numbers represented by linked list.

Algorithm:→ Step1:→ Reverse both L.L

Step2:→ Add them from left

Step3:→ reverse ans.

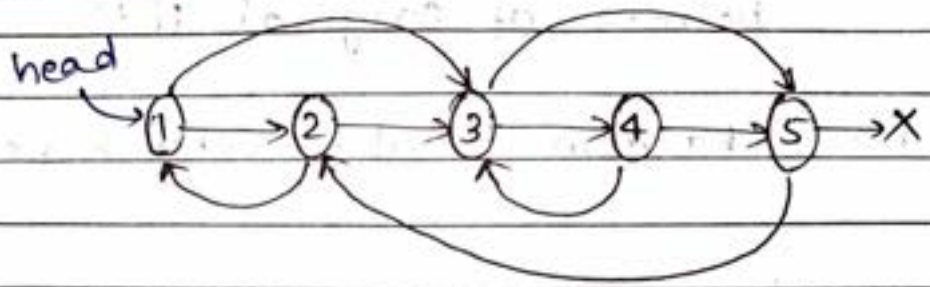
Program:→

```
Node* reverse (Node* head){
    - - - - - - - - - ;
}

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

struct Node* add (struct Node* first, struct Node* second) {
    int carry = 0;
    Node* ansHead = NULL;
    Node* ansTail = NULL;

    while (first != NULL || second != NULL || carry != 0) {
        int val1 = 0;
        if (first != NULL)
            val1 = first → data;

        int val2 = 0;
        if (second != NULL)
            val2 = second → data;

        int sum = carry + val1 + val2;
        int digit = sum % 10;
```

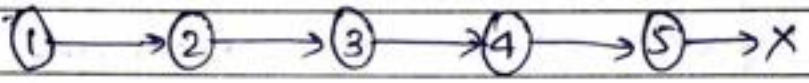
```
        // create node and add in ans L.L
        insertAtTail (ansHead, ansTail, digit);

        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 L.L
    first = reverse (first);
    second = reverse (second);

    // step 2 -> add 2 L.L
    Node* ans = add (first, second);

    // step 3 -> reverse ans
    ans = reverse (ans);

    return ans;
}
```

8 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 dur hai aur usko point krwa dunga, fir next node pe jaunga.

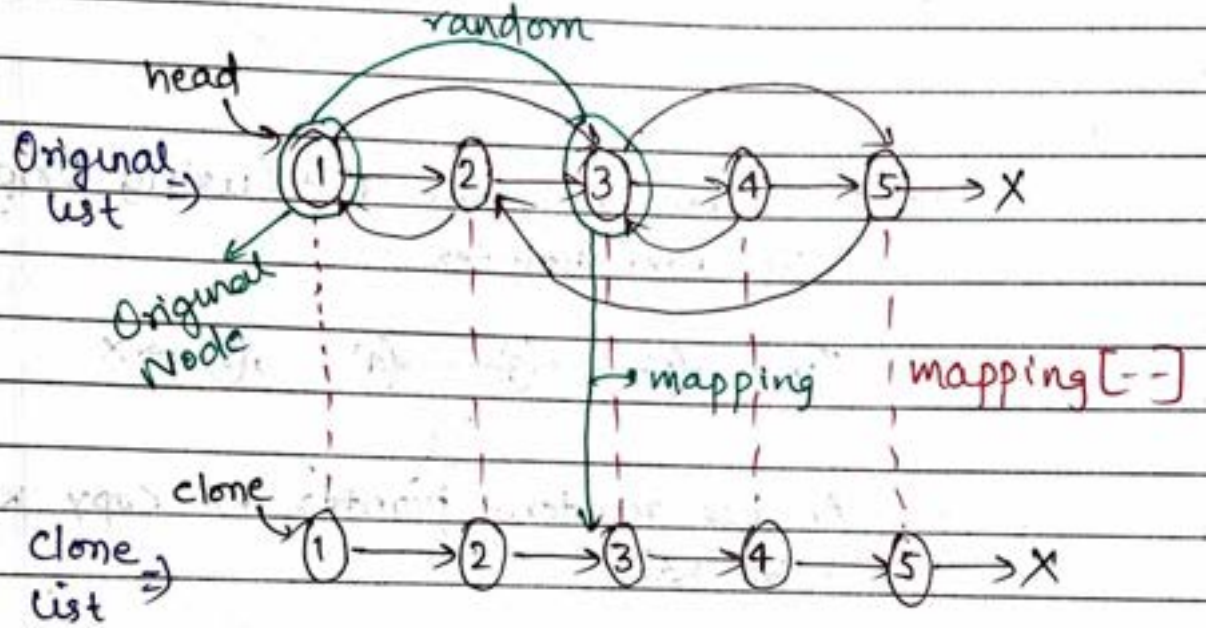
For (Original list mein dhund ke ki uska random ptr. kitna distance pe h.
{.
$O(n^2)$ While (Jbtk main clone wale list ko uske random ptr. ko sii jgh nhi lga deta).
.}}
};

T.C → $O(n^2)$

Approach 2:-

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

Step 2:- Random pointer copy krne hai



Clone Node → random = mapping [original Node ↓ random]

T.C → O(N)

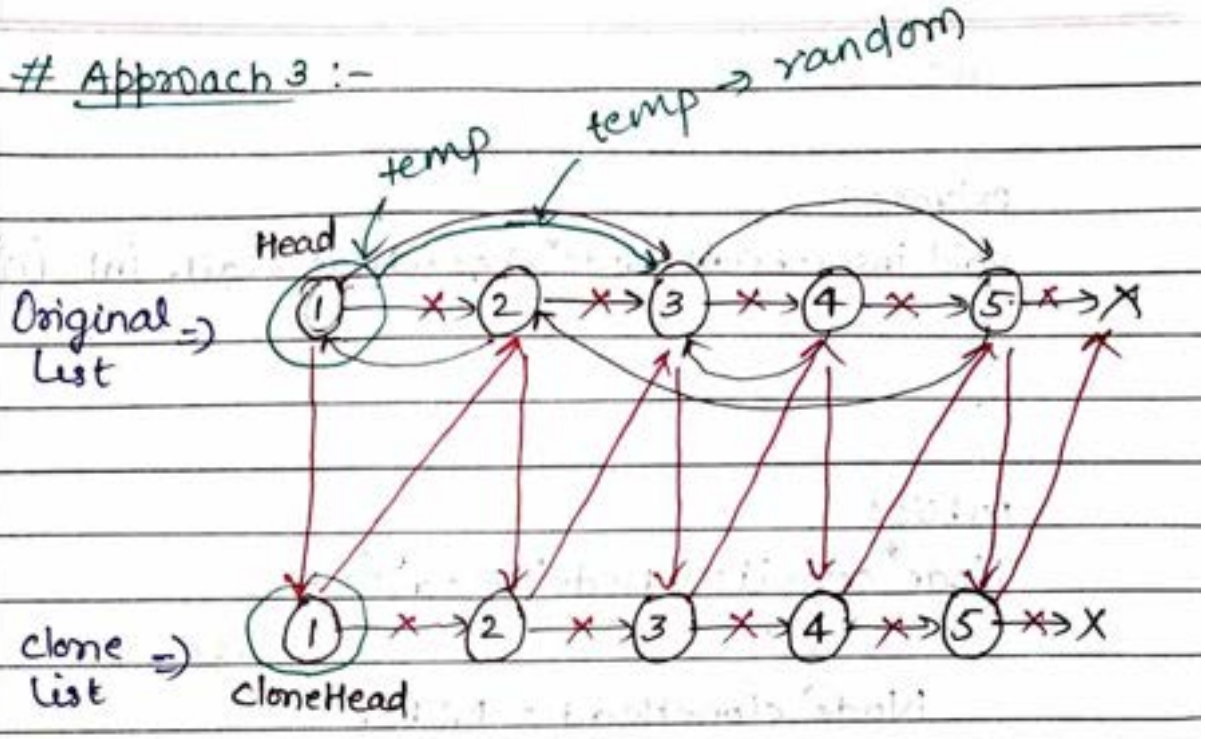
S.C → O(N), because of map.

Program :-

```
Private :
void insertAtTail (Node* &head , Node* &tail, int d) {
    - - - - - - - .
}

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) {
                                   optional
        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 Originallist.

step 3 :- Random pointer (clone list)

temp → next → random = temp → random → next

step 4 :- revert changes done in step 2.

step 5 :- return ans (cloneHead)

Program :-

```
private :
void insertAtTail (Node* &head, Node* &tail, int d){
    . . . . . .
}

public:
Node* copyList (Node* head){
    //step1 :- create a clone List
    Node* cloneHead = NULL;
    Node* cloneTail = NULL;

    Node* temp = head;
    while (temp != NULL) {
        insertAtTail (cloneHead, cloneTail, temp->data);
        temp = temp -> next;
    }

    //step2 :- 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
        ? 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;
}
```

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;
        }
    }
```

```
    while (left != NULL) {
        temp -> next = right;
        temp = right;
        right = right -> next;
    }
    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;
}
```

Q 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;
```