

# Linked List

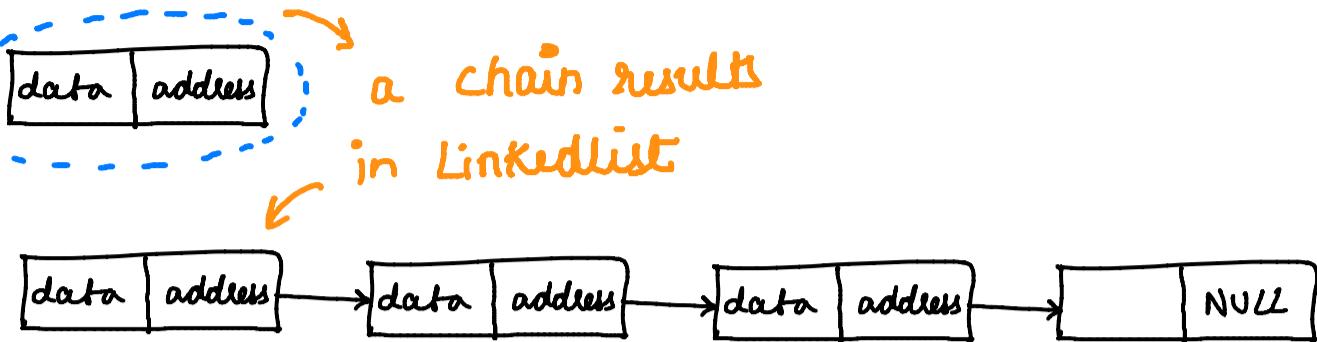
Contents →

0. Introduction
1. Reverse a Linked List
2. Middle of Linked List
3. Delete node in a Linked List
4. Merge two sorted Lists
5. Add two numbers
6. Add two numbers II
7. Linked List Cycle
8. Linked List Cycle II
9. Remove Nth node from End of List
10. Palindrome Linked List
11. Remove duplicates from sorted List
12. Swapping nodes in Linked List
13. Odd Even Linked List
14. Swap Nodes in Pairs
15. Copy list with Random Pointer
16. Reverse Nodes in K-group
17. Design Linked List
18. Sort List

# Linked List

LinkedList is linear data structure, which consists of a group of nodes in a sequence.

Class Node  
→ data  
→ Node\*



## Advantages

1. Dynamic nature
2. Optimal insertion & deletion
3. Stacks and queues can be easily implemented
4. No memory wastage

## Disadvantages

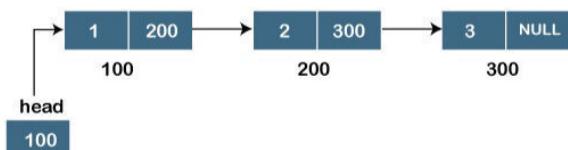
1. More memory usage due to address pointers.
2. Slow traversal compared to arrays.
3. No reverse traversal in singly linked list
4. No random access.

## Real life Applications

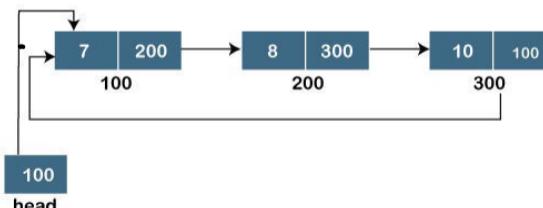
1. Previous & next page in browser
2. Image Viewer
3. Music player

## Type

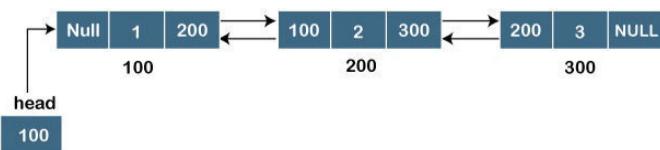
### 1. Singly linkedlist



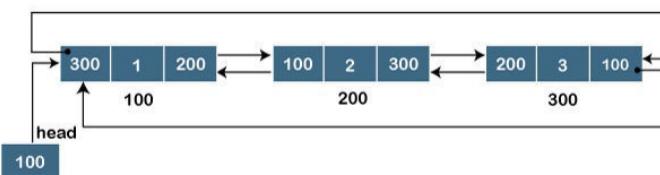
### 3. Circular Linkedlist



### 2. Doubly linkedlist



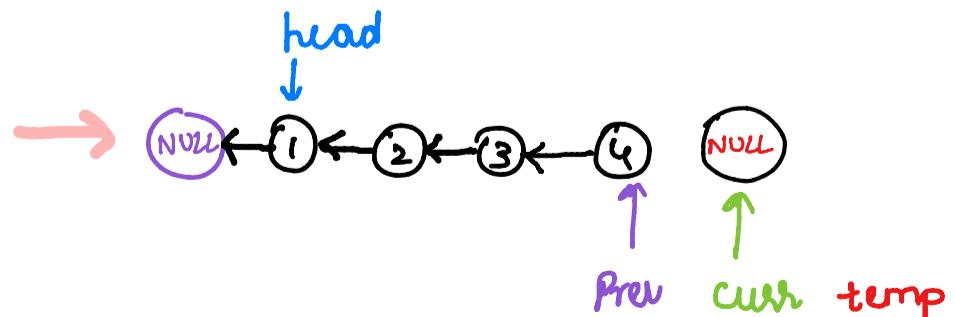
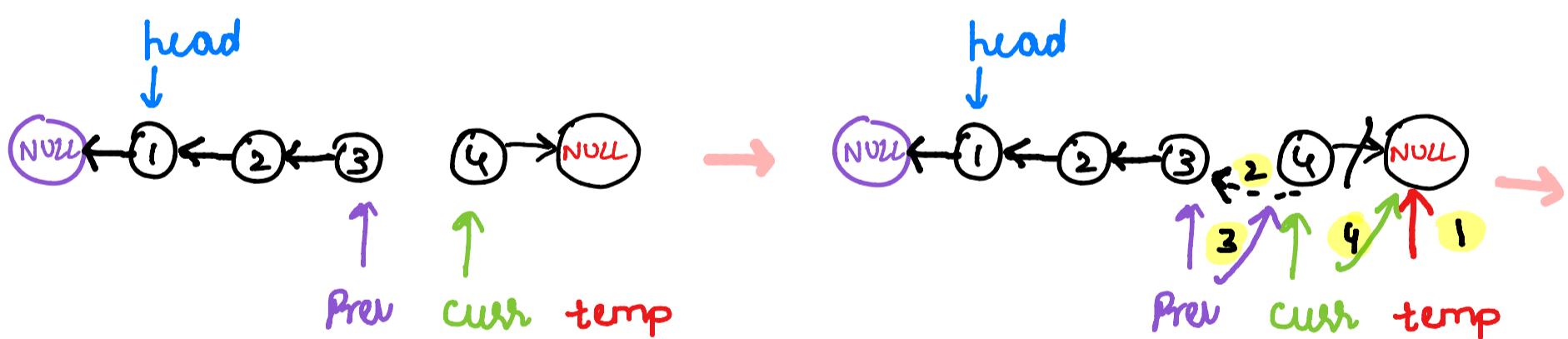
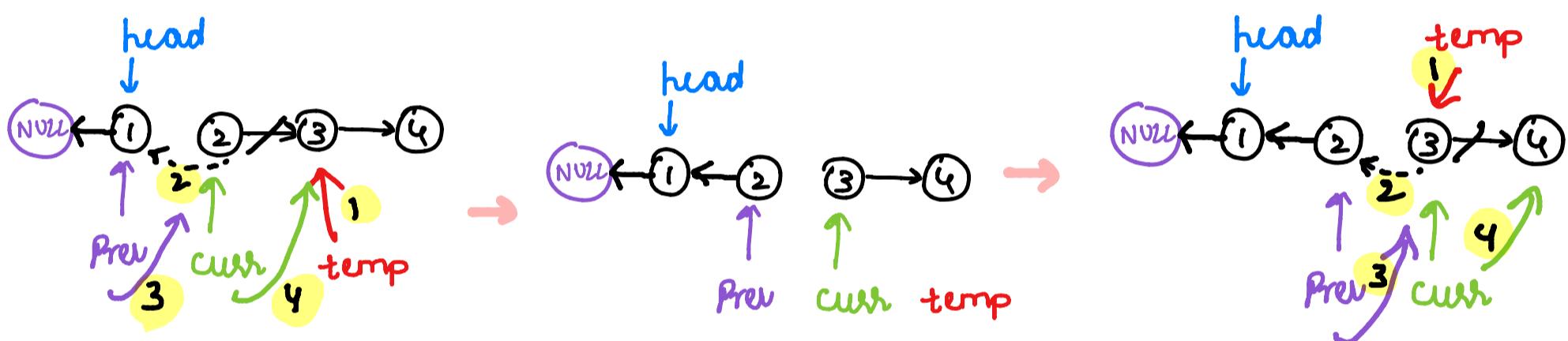
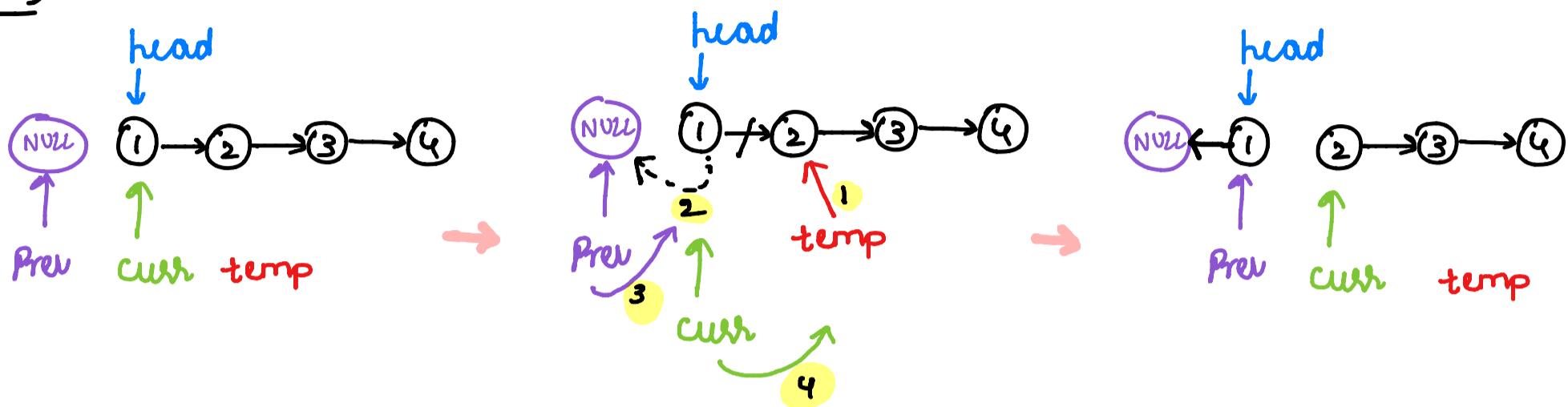
### 4. Doubly circular linkedlist



① Reverse a linkedlist → gives a linkedlist, return reversed list.

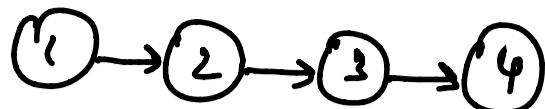
Eg  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \Rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

Sol)



→ As curr points to NULL, we reached end of linkedlist  
→ Return Prev as it is the starting pointer of reversed list.

## Recursive →



curr, prev

(1, NULL):

newNode = 1 → next = 2

1 → next = null

call (2, 1)

curr, prev

(2, 1):

newNode = 2 → next = 3

2 → next = 1

call (3, 2)

curr, prev

(3, 2):

newNode = 3 → next = 4

3 → next = 2

call (4, 3)

curr, prev

(4, 3):

newNode = 4 → next = null

4 → next = 3

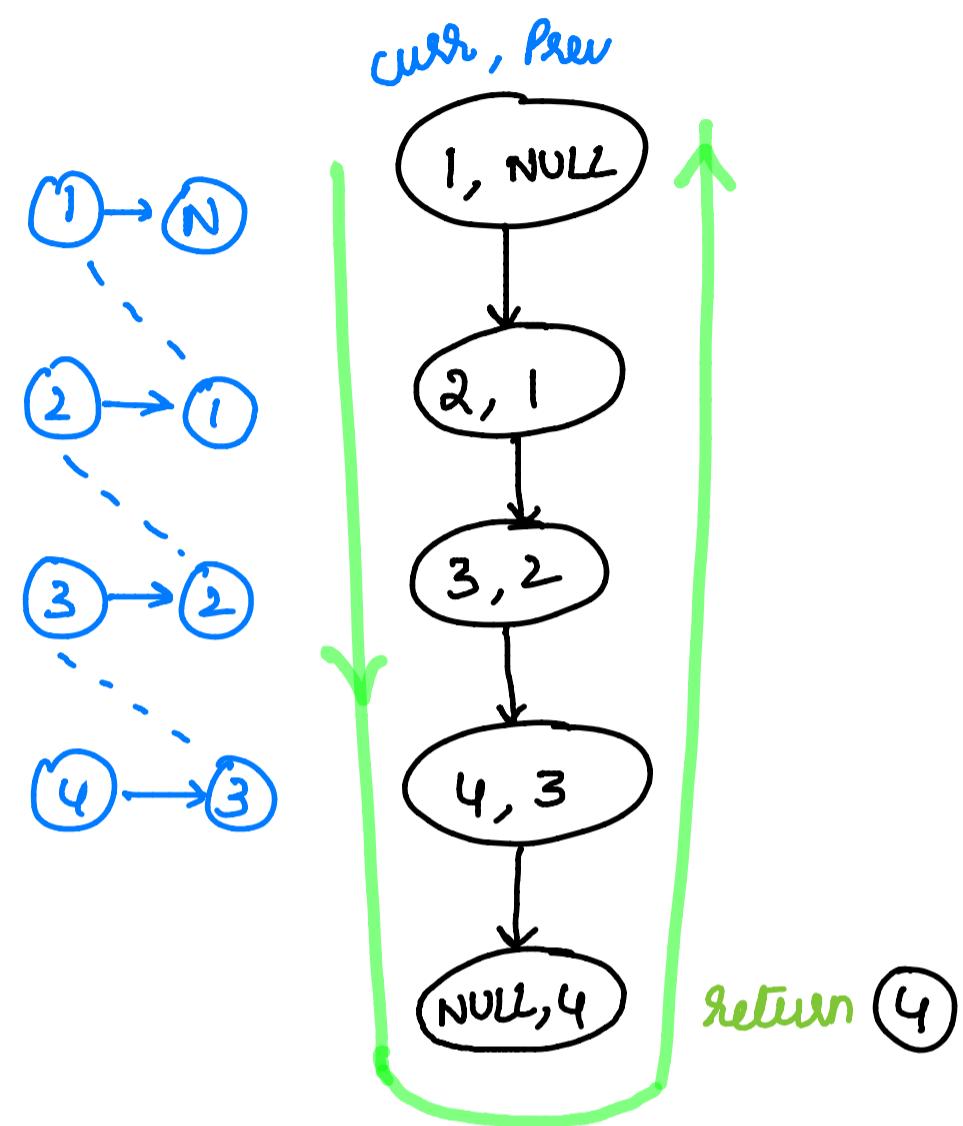
call (NULL, 4)

curr, prev

(NULL, 4):

as curr == NULL

return prev



func (curr, prev):

if curr == NULL

return prev

newNode = curr → next

curr → next = prev

recursively call newNode & curr as  
as curr → prev

Code →

$Tc \rightarrow O(n)$

$Sc \rightarrow O(1)$

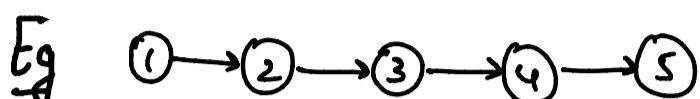


```
1
2 // Iterative ->
3 class Solution {
4 public:
5     ListNode* reverseList(ListNode* head) {
6         ListNode *prev = NULL, *curr=head, *temp;
7         while(curr){
8             temp = curr->next;
9             curr->next = prev;
10            prev = curr;
11            curr = temp;
12        }
13        return prev;
14    }
15 };
16
17
18 // Recursive ->
19 class Solution {
20 public:
21     ListNode* reverseLinker(ListNode* curr, ListNode* prev) {
22         if(curr==NULL)
23             return prev;
24         ListNode* newNode = curr->next;
25         curr->next = prev;
26         return helper(newNode, curr);
27     }
28
29     ListNode* reverseLinker(ListNode* head) {
30         return helper (head, NULL);
31     }
32 }
```

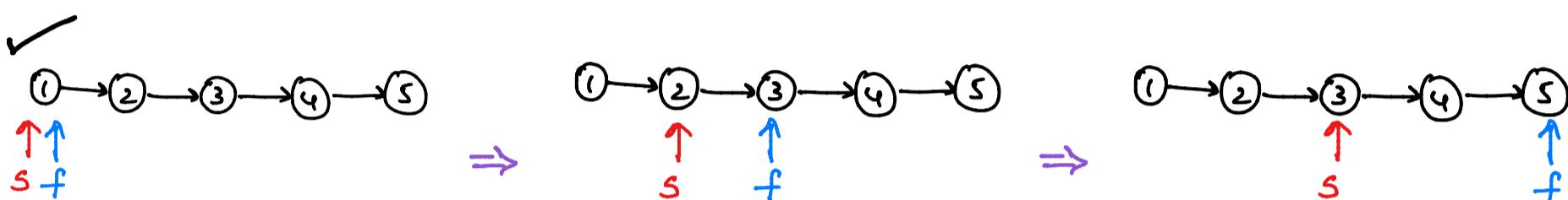
② Middle of Linkedlist → Given the head, return middle node.

Approach 1 → traverse the list & find no of nodes & return mid

Approach 2 → use 2 pointers → slow (moves by 1) } By time  
fast (moves by 2) } fast reaches end, slow points to the mid.



Res ⇒ 



code

as fast reached end  
return slow.



```
1 class Solution {
2 public:
3     ListNode* middleNode(ListNode* head) {
4         if(head == NULL)
5             return head;
6         ListNode* slow = head, *fast = head;
7
8         // Traverse the LinkedList
9         while(fast != NULL && fast -> next != NULL)
10        {
11            slow = slow -> next;
12            fast = fast -> next -> next;
13        }
14
15        return slow;
16    }
17};
```

TC → O(n)

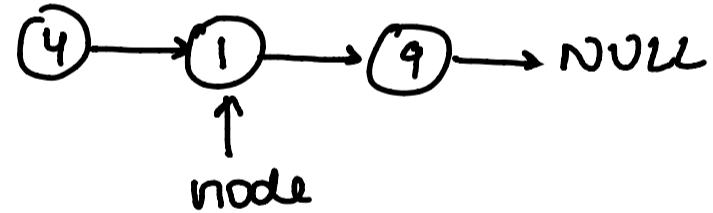
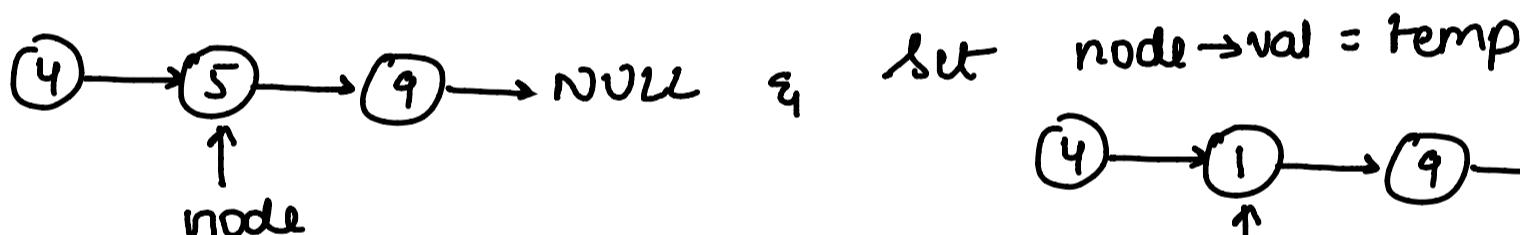
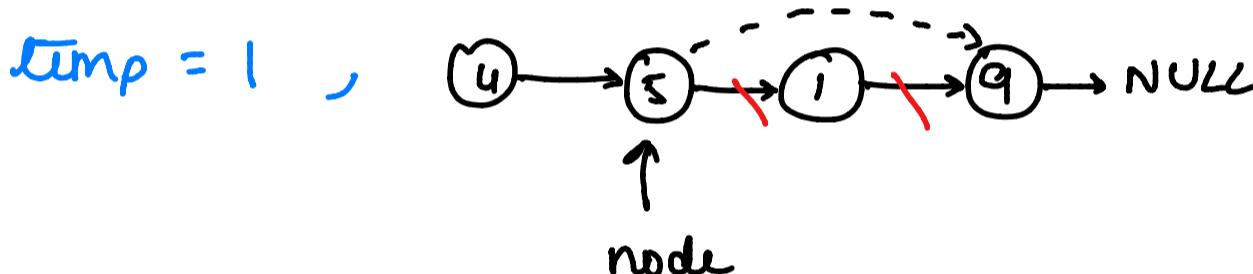
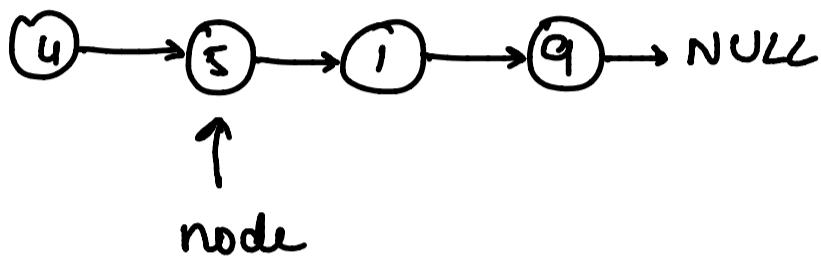
SC → O(1)

### ③ Delete node in a linkedlist →

given a linkedlist's node, delete it.

- copy node's next node's val into a temp variable
- skip the node→next node
- copy the temp variable's value into the node.

Ex

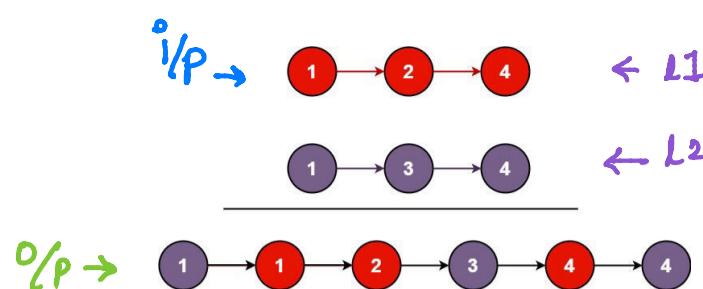


code →

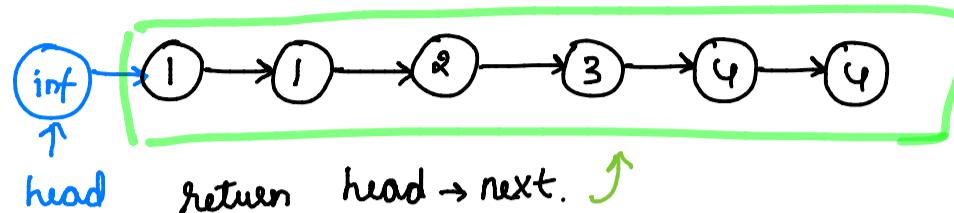
```
● ● ●  
1 class Solution {  
2 public:  
3     void deleteNode(ListNode* node) {  
4         int val = node->next->val;  
5         node->next = node->next->next;  
6         node->val = val;  
7     }  
8 }
```

Tc → O(1)  
Sc → O(1)

#### ④ Merge two sorted lists



→ Take a dummyNode & chain the next node which contains smaller value of  $L_1$  &  $L_2$



Code →

```

1  class Solution {
2  public:
3      ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
4
5          if( l1 == NULL ) return l2;
6          if( l2 == NULL ) return l1;
7
8          ListNode* dummy = new ListNode(-101);
9          ListNode* head = dummy;
10
11         // Traverse the lists
12         while( l1 != NULL && l2 != NULL )
13     {
14             if( l1->val < l2->val )
15             {
16                 ListNode* newnode = new ListNode(l1->val);
17                 dummy->next = newnode;
18                 l1 = l1->next;
19             }
20             else
21             {
22                 ListNode* newnode = new ListNode(l2->val);
23                 dummy->next = newnode;
24                 l2 = l2->next;
25             }
26             dummy = dummy->next;
27         }
28
29         /* If a particular list is NULL, then directly chain
30         the other */
31         if(l1!=NULL) dummy->next = l1;
32         if(l2!=NULL) dummy->next = l2;
33
34         return head->next;
35     }
36 };

```

$Tc \rightarrow O(m+n)$

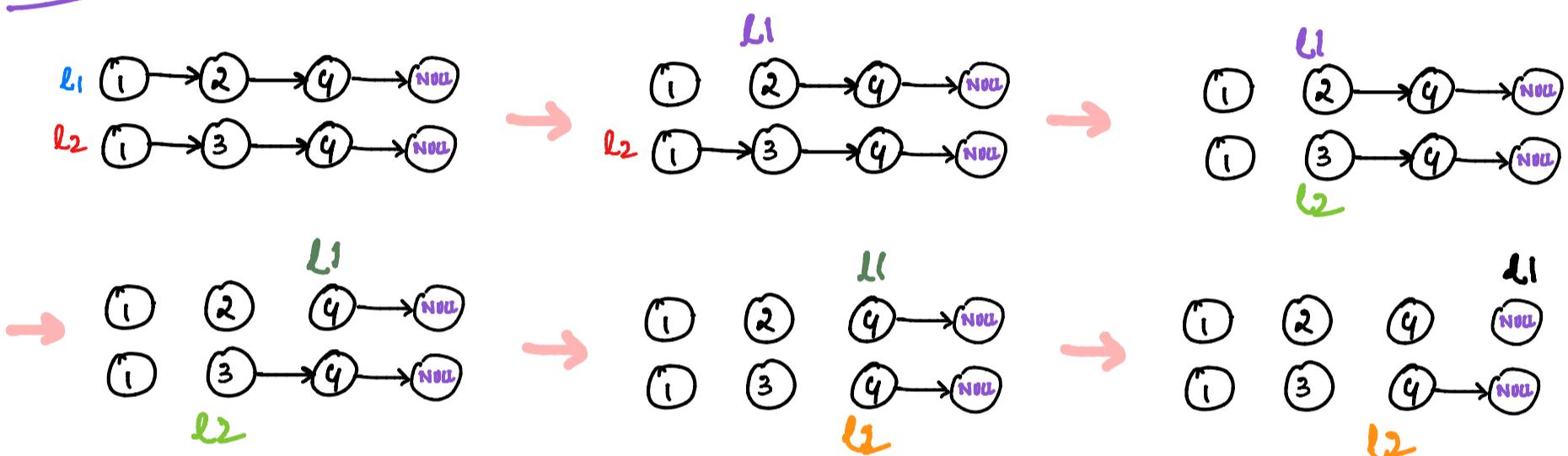
$Sc \rightarrow O(m+n)$

## Recursive Code →

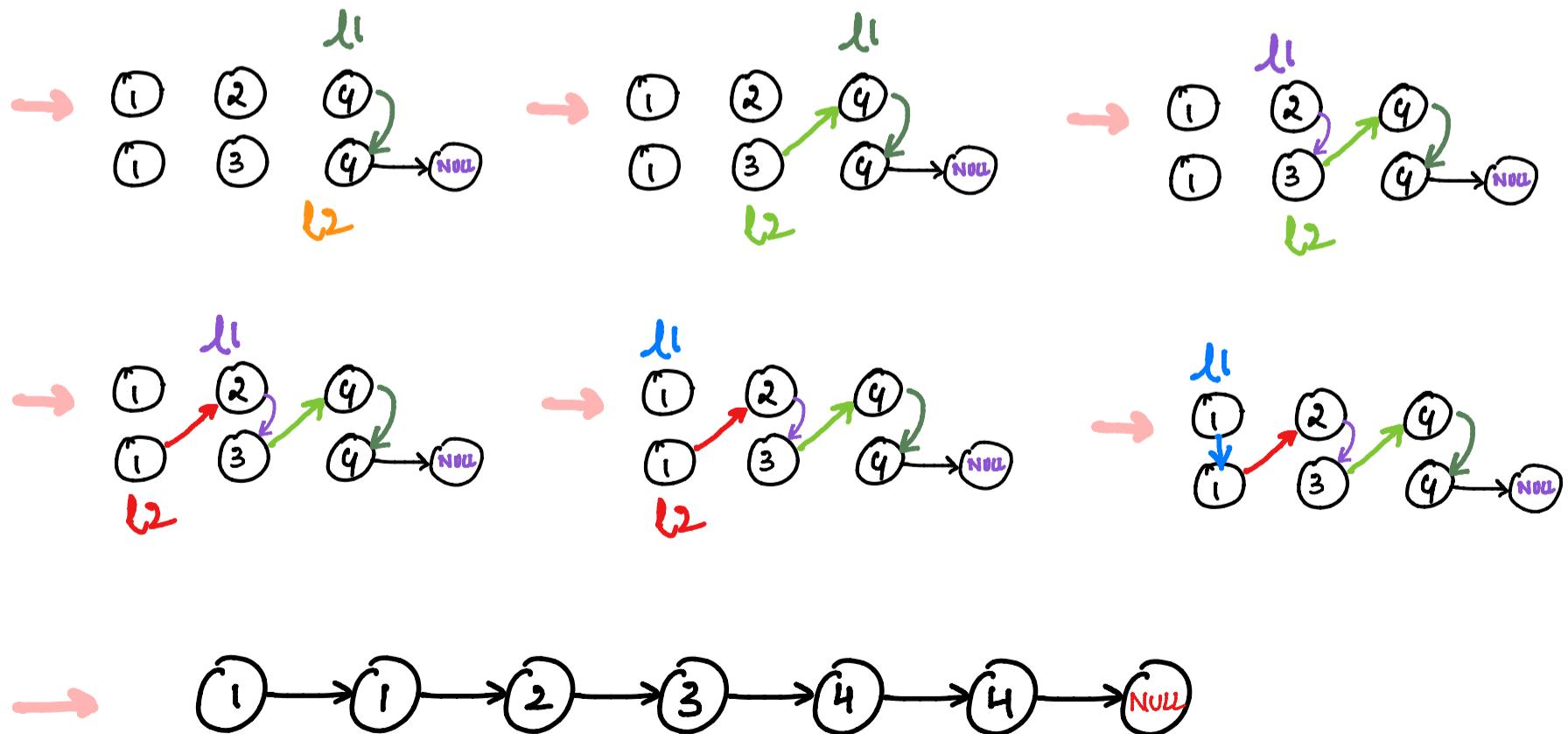
```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (l1 == NULL) return l2;
        if (l2 == NULL) return l1;

        // compare the starting values and link the nodes
        if (l1->val <= l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

## Dry Run

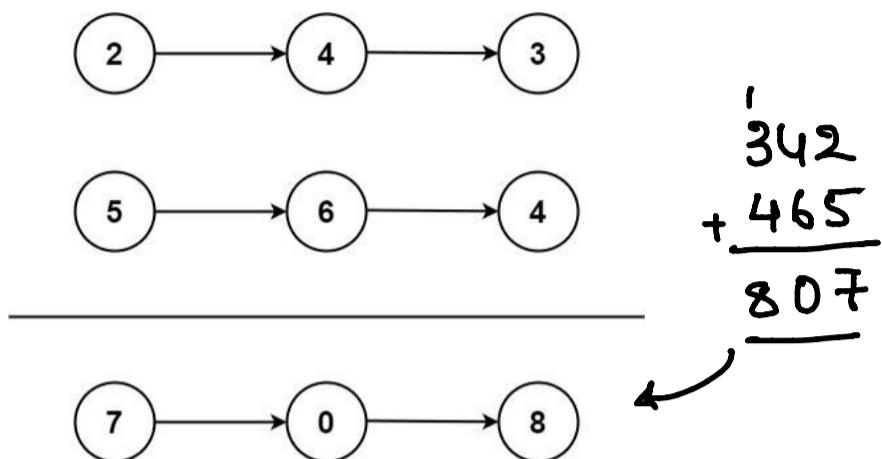


As  $l_1$  is null, return  $l_2$



## 5 Add two Numbers

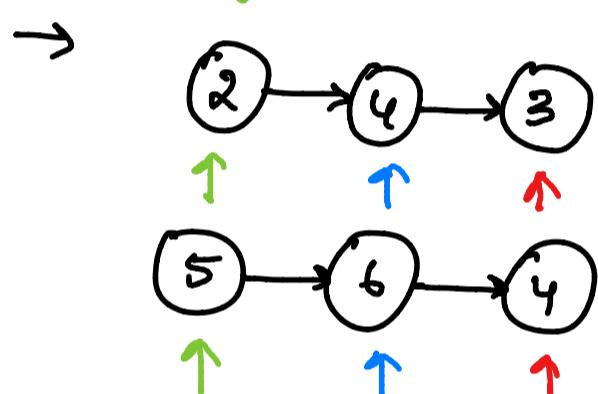
Given 2 lists in  
reverse order, add them  
and return the sum.



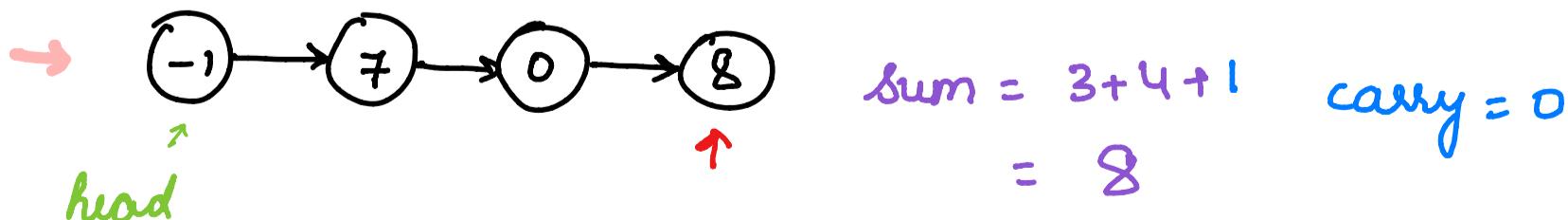
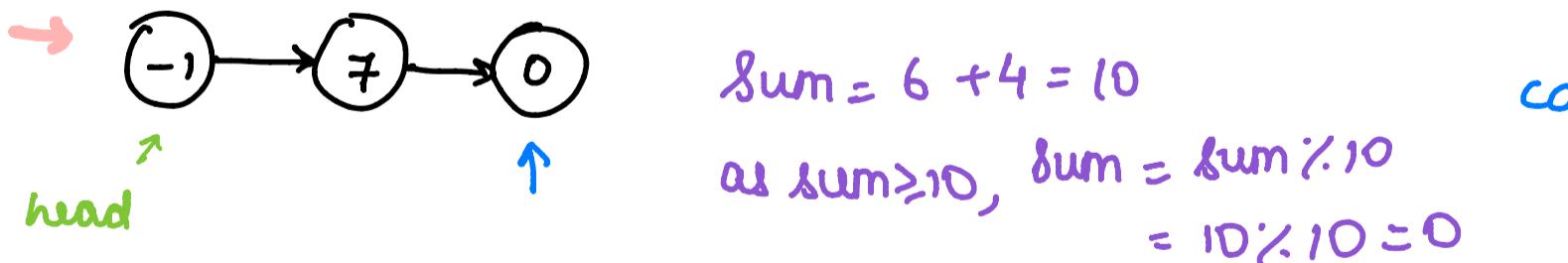
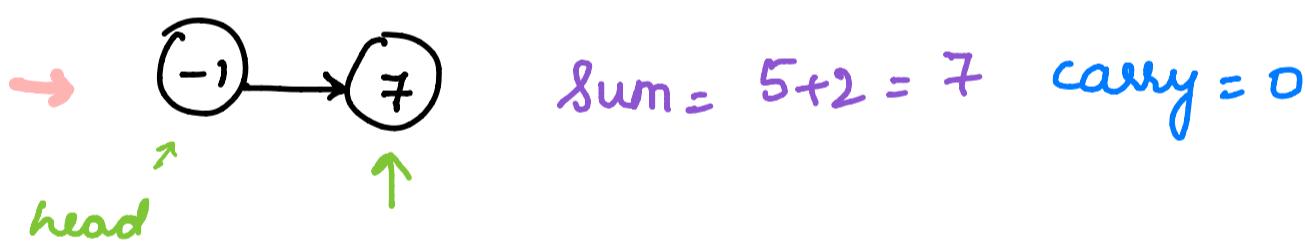
$$Tc \rightarrow 0(m+n)$$

$$Sc \rightarrow O(\max(m,n))$$

Initially,



- 1) traverse both list simultaneously  
if sum  $\geq 10$ , then set carry=1
  - 2) add both values + carry
  - 3) Create newNode with this value



## code



```
1 class Solution {
2 public:
3
4     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
5
6         ListNode* dummyNode;
7         ListNode* head;
8         dummyNode = head = new ListNode(-1);
9         if(!l1)
10             return l2;
11         if(!l2)
12             return l1;
13
14         int carry = 0;
15
16         while(l1 || l2){
17             int firstVal = l1 ? l1->val : 0;
18             int secondVal = l2 ? l2->val : 0;
19
20             int total = firstVal + secondVal + carry;
21             carry = total / 10;
22             total = total % 10;
23
24             ListNode* newNode = new ListNode(total);
25             dummyNode->next = newNode;
26
27             dummyNode = dummyNode->next;
28
29             l1 = l1 ? l1->next : l1;
30             l2 = l2 ? l2->next : l2;
31         }
32
33         if(carry)
34             dummyNode->next = new ListNode(1);
35
36         return head->next;
37     }
38 }
```

## ⑥ Add two numbers !!

→ Problem solving approach is same as previous problem

→ Points to note :

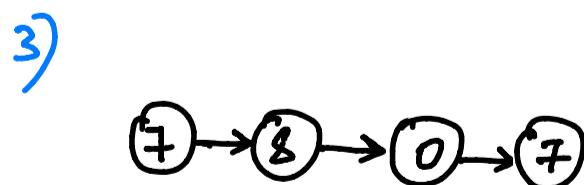
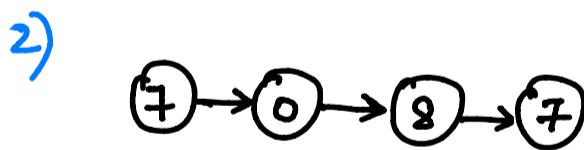
1. Reverse both the lists
2. Add them
3. Reverse the result

Eg →



Code →

```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         ListNode* prev = NULL;
5         ListNode* curr = head;
6         ListNode* temp = NULL;
7         while(curr!=NULL)
8         {
9             temp = curr->next;
10            curr->next = prev;
11            prev = curr;
12            curr = temp;
13        }
14        return prev;
15    }
16
17    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
18        l1=reverseList(l1); // O(n)
19        l2=reverseList(l2); // O(n)
20        ListNode* dummyNode;
21        ListNode* head;
22        dummyNode = head = new ListNode(-1);
23        if(!l1)
24            return l2;
25        if(!l2)
26            return l1;
27
28        int carry = 0;
29
30        while(l1 || l2){
31            int firstVal = l1 ? l1->val : 0;
32            int secondVal = l2 ? l2->val : 0;
33
34            int total = firstVal + secondVal + carry;
35            carry = total / 10;
36            total = total % 10;
37
38            ListNode* newNode = new ListNode(total);
39            dummyNode->next = newNode;
40
41            dummyNode = dummyNode->next;
42
43            l1 = l1 ? l1->next : l1;
44            l2 = l2 ? l2->next : l2;
45        }
46
47        if(carry)
48            dummyNode->next = new ListNode(1);
49
50        return reverseList(head->next); // O(max(m,n))
51    }
52};
```



Result ↑

## ⑦ Linked List Cycle

$Tc \rightarrow O(n)$   
 $Sc \rightarrow O(1)$

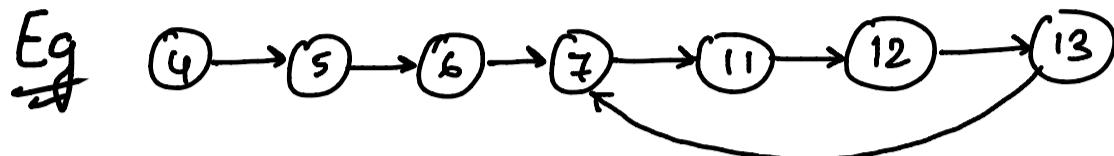
### Approach - 1

Create a set of nodes & insert every node into it,  
if already exist then return true else false.

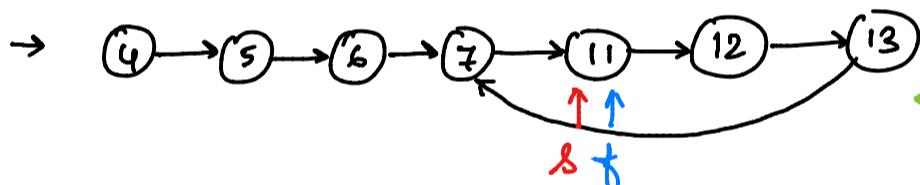
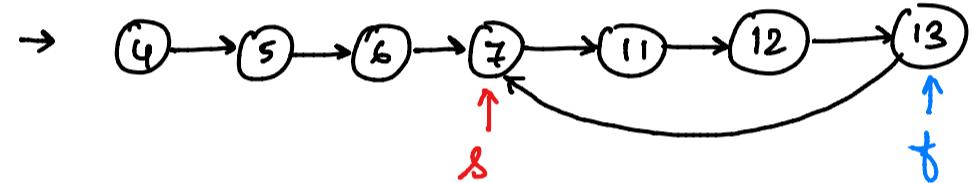
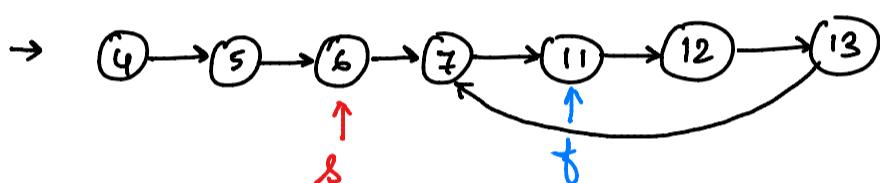
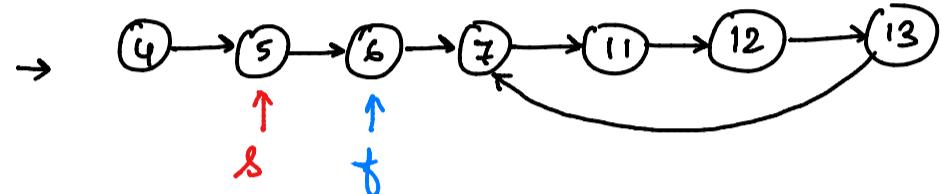
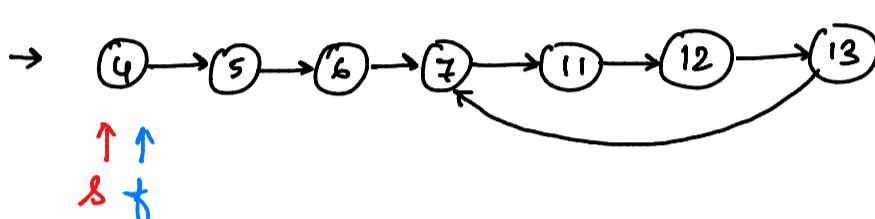
### Approach - 2

using fast & slow pointer  $Tc \rightarrow O(n)$   $Sc \rightarrow O(1)$

Eg



keep iterating till  
fast->next && fast->next->next exist.

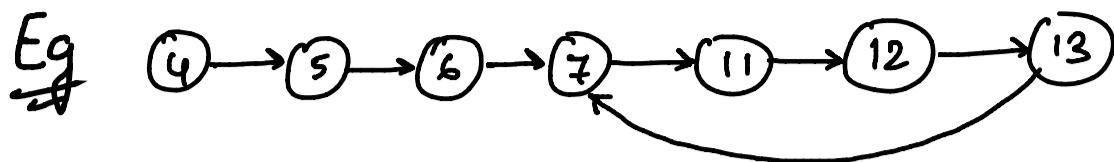


return true

Code →

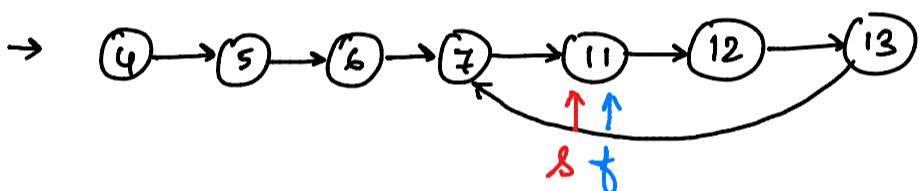
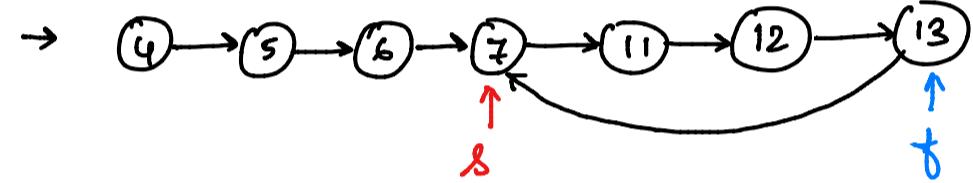
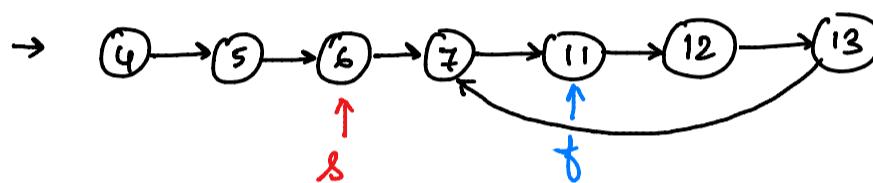
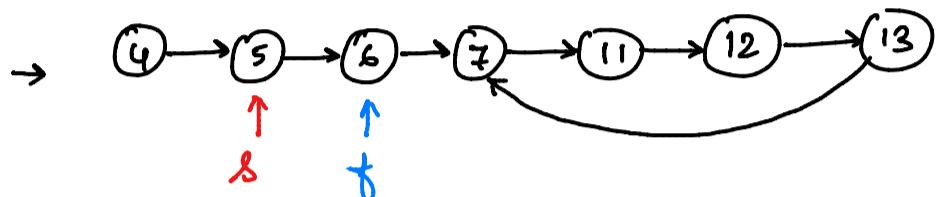
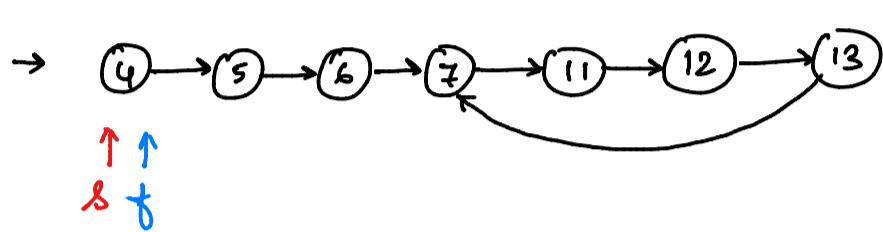
```
1 class Solution {
2 public:
3     bool hasCycle(ListNode *head) {
4         if(head==NULL) return false;
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL && fast->next->next!=NULL)
7         {
8             fast=fast->next->next;
9             slow=slow->next;
10            if(fast==slow) return true;
11        }
12        return false;
13    }
14};
```

(8) Linked list cycle  $\Rightarrow$  returns the node where cycle begins.

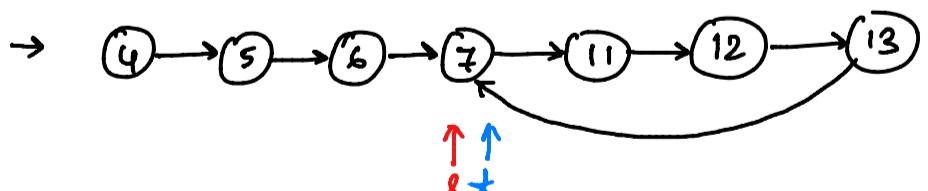
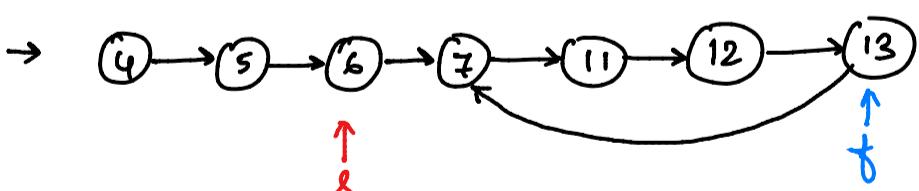
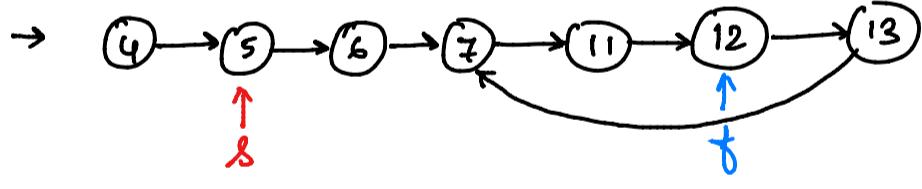
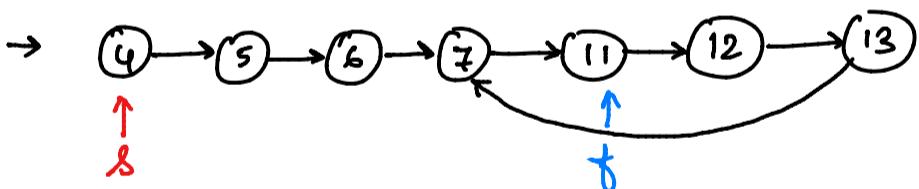


result = 7

keep iterating while  $\text{fast} \rightarrow \text{next}$  &  $\text{fast} \rightarrow \text{next} \rightarrow \text{next}$  exist.



← once  $\text{slow} == \text{fast}$ , then set  
 $\text{slow} = \text{head}$ , & move pointers  
by 1 unit



$\rightarrow$  when  $\text{slow} == \text{fast}$ , it denotes the node where cycle begins.

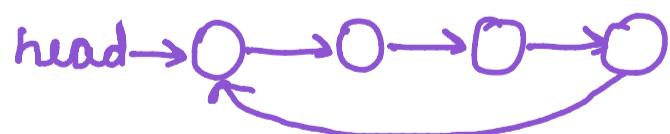
return slow;  $\Rightarrow$  7

code →

```
● ● ●  
1 class Solution {  
2 public:  
3     ListNode *detectCycle(ListNode *head) {  
4         if(head==NULL) return NULL;  
5         ListNode *fast = head, *slow = head;  
6         while(fast->next!=NULL && fast->next->next!=NULL)  
7         {  
8             fast = fast->next->next;  
9             slow = slow->next;  
10            if(fast == slow)  
11            {  
12                slow = head;  
13                while (slow != fast)  
14                {  
15                    slow = slow->next;  
16                    fast = fast->next;  
17                }  
18                return slow;  
19            }  
20        }  
21        return NULL;  
22    }  
23};
```

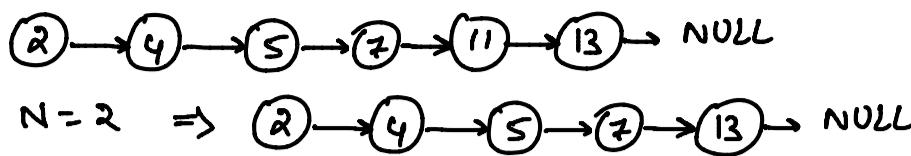
$$TC \rightarrow O(n) + O(n) = O(2n) = \underline{\underline{O(n)}}.$$

\* worst case when its a loop



$$SC \rightarrow O(1)$$

## ⑨ Remove $N^{\text{th}}$ node from End of list



Approach - 1 → find length of list  $L$ , delete  $l-n+1^{\text{th}}$  node

Approach - 2 → a) Reverse b) Delete  $N^{\text{th}}$  node c) Reverse

Code →

$T_C \rightarrow O(n)$

$S_C \rightarrow O(1)$

```
1 class Solution {
2 public:
3     ListNode* reverseList(ListNode* head) {
4         ListNode *prev = NULL, *curr = head, *temp;
5         while(curr){
6             temp = curr->next;
7             curr->next = prev;
8             prev = curr;
9             curr = temp;
10        }
11        return prev;
12    }
13
14    ListNode* removeNthFromEnd(ListNode* head, int n) {
15        ListNode *dummy = new ListNode(-1);
16        dummy->next = reverseList(head);
17        head = dummy;
18        ListNode *curr = head;
19        ListNode *prev = NULL;
20        // Iteration
21        for(int i=0; i<n; i++)
22        {
23            prev = curr;
24            curr = curr->next;
25        }
26        // Deletion
27        prev->next = curr->next;
28        return reverseList(head->next);
29    }
30}
```

## ⑩ Pallindrome Linked List

Approach - 1 → Create a copy of list & reverse it. Compare value by value.  
If all are equal true else false.

Approach - 2 → Reach middle node & return the remaining list as new list. Reverse the newList & compare its value by value.

Code →

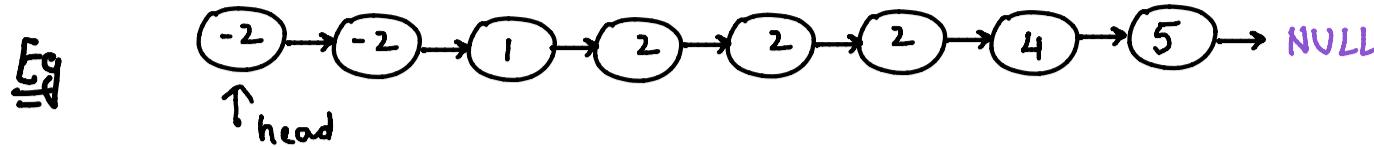
TC → O(n)

SC → O(1)

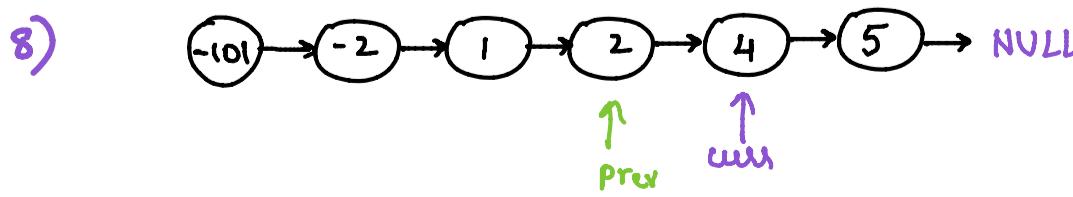
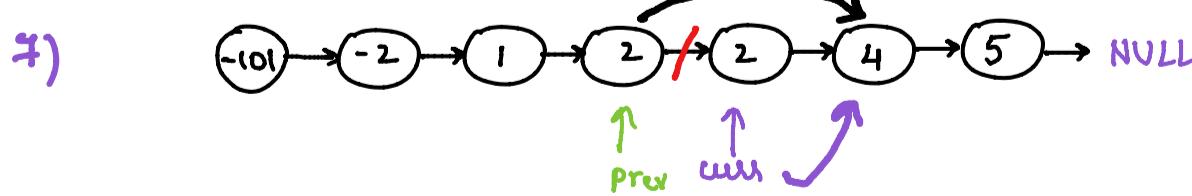
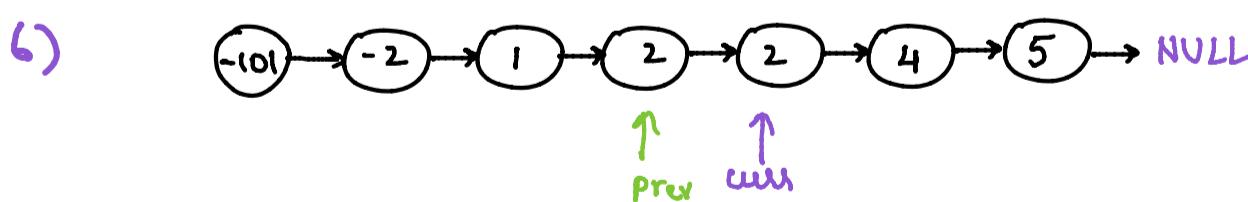
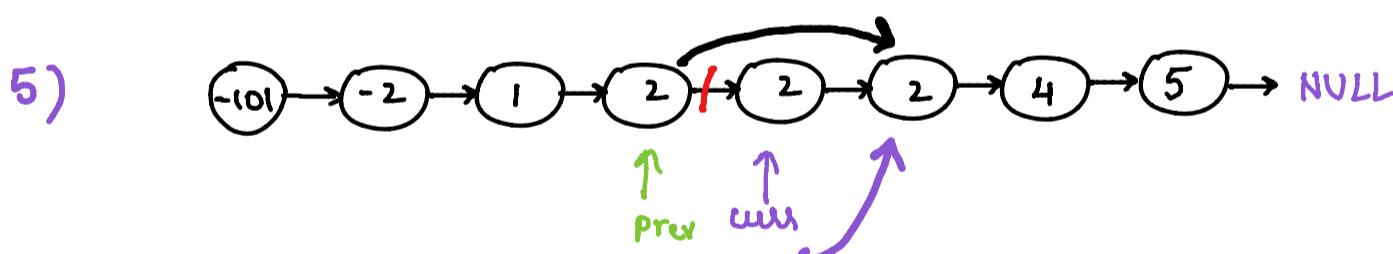
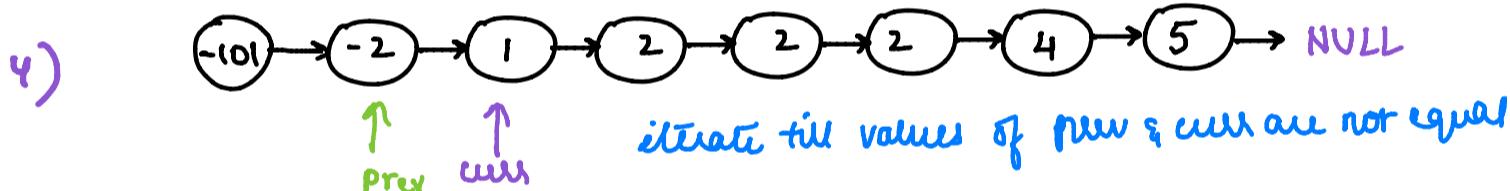
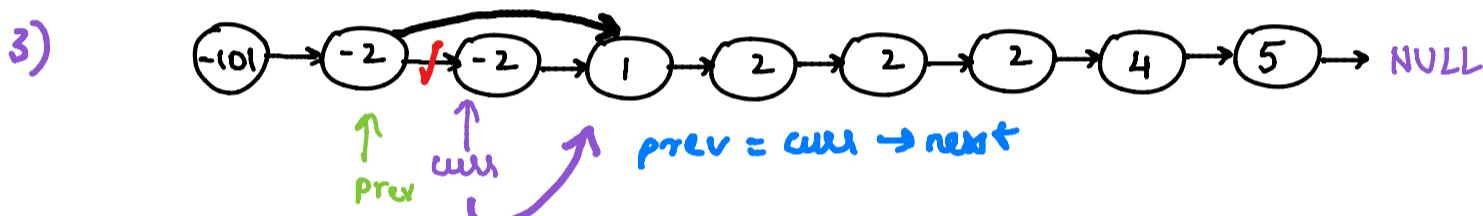
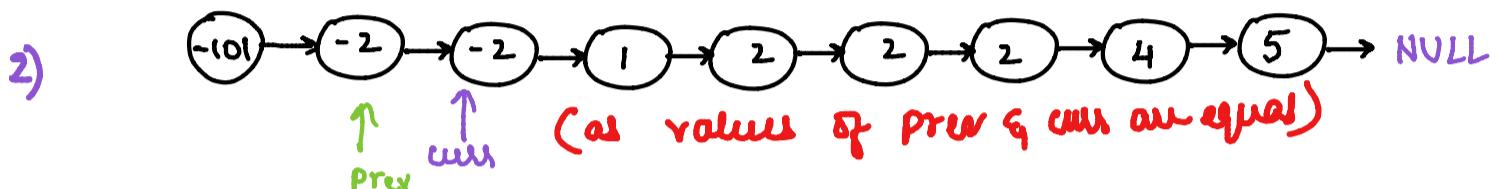
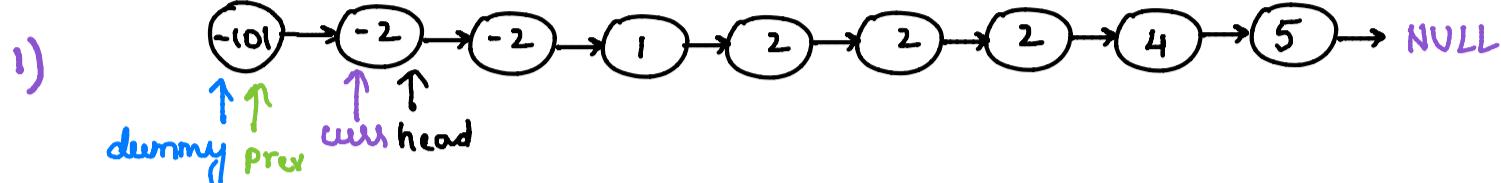
```
● ● ●
1 class Solution {
2 public:
3     ListNode* midNode(ListNode*head)
4     {
5         ListNode *fast = head, *slow = head;
6         while(fast->next!=NULL and fast->next->next!=NULL){
7             fast=fast->next->next;
8             slow = slow->next;
9         }
10        return slow;
11    }
12    ListNode* reverseList(ListNode* head) {
13        ListNode *prev = NULL, *curr = head, *temp;
14        while(curr!=NULL)
15        {
16            temp=curr->next;
17            curr->next = prev;
18            prev = curr;
19            curr = temp;
20        }
21        return prev;
22    }
23    bool compare(ListNode*l1,ListNode* l2)
24    {
25        while(l1!=NULL && l2!=NULL)
26        {
27            if(l1->val!=l2->val)    return false;
28            l1 = l1->next;
29            l2 = l2->next;
30        }
31        return true;
32    }
33    bool isPalindrome(ListNode* head) {
34        if(head==NULL) return false;
35        if(head->next == NULL) return true;
36        ListNode *mid = midNode(head);
37        ListNode *l2 = mid->next;
38        mid->next = NULL;
39        l2 = reverseList(l2);
40        return compare(head,l2);
41    }
42};
```

# (1) Remove duplicates from sorted list →

Given a linkedlist, return linkedlist without duplicates.



⇒ any out of range value



code →

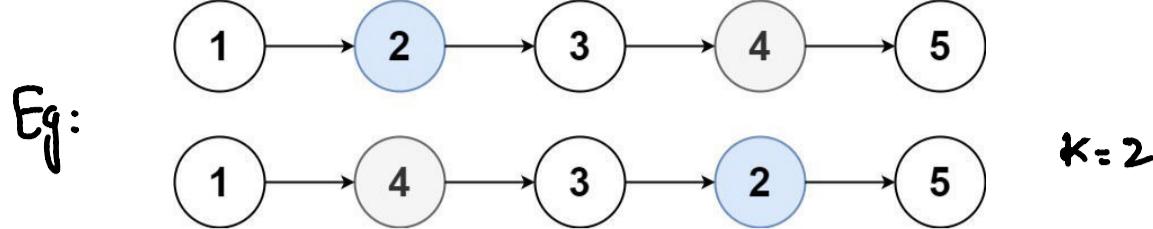
```
● ● ●

1 class Solution {
2 public:
3     ListNode* deleteDuplicates(ListNode* head) {
4         ListNode* dummy = new ListNode(101);
5         dummy->next = head;
6         ListNode* curr = head;
7         ListNode* prev = dummy;
8         while(curr!=NULL)
9         {
10             if(curr->val==prev->val){
11                 prev->next = curr->next;
12                 curr = curr->next;
13             } else {
14                 prev = curr;
15                 curr = curr->next;
16             }
17         }
18         return dummy->next;
19     }
20 };
21
22
23 // Another approach
24
25 ListNode* deleteDuplicates(ListNode* head) {
26     if(head==NULL || head->next==NULL) return head;
27     ListNode *curr = head;
28     while(curr->next!=NULL){
29         if(curr->val == curr->next->val){
30             curr->next = curr->next->next;
31         } else {
32             curr = curr->next;
33         }
34     }
35     return head;
36 }
```

$$Tc \rightarrow O(n)$$

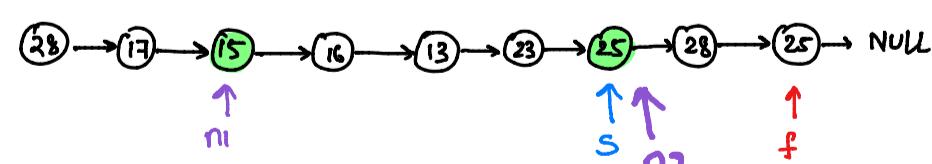
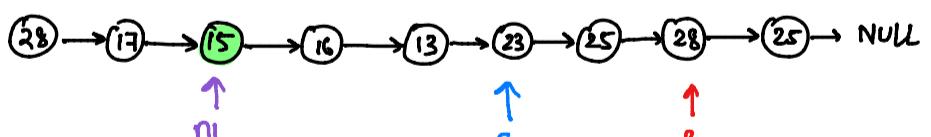
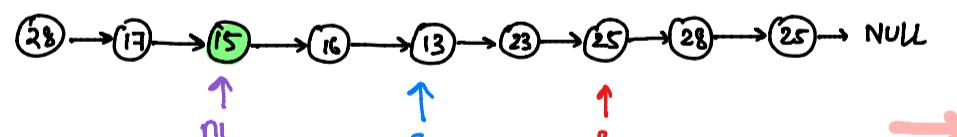
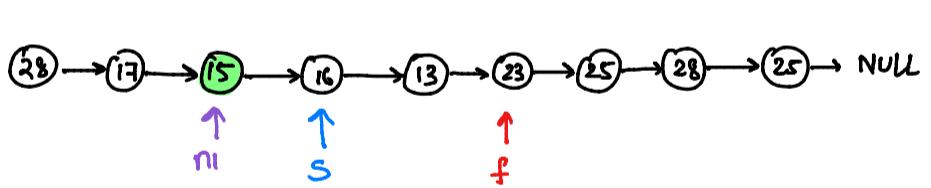
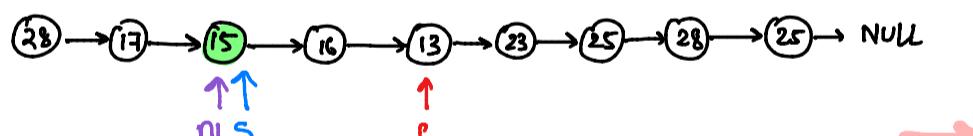
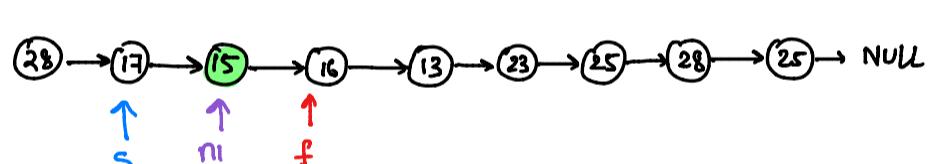
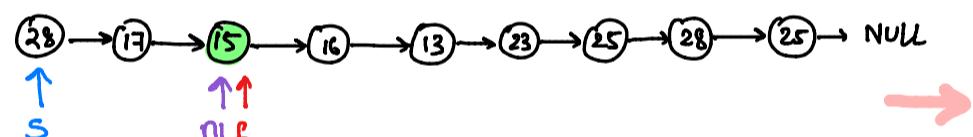
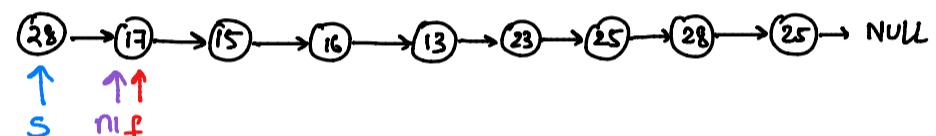
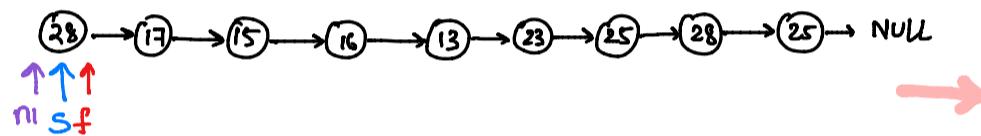
$$Sc \rightarrow O(1)$$

## (12) Swapping Nodes in Linked list →



given a linkedlist swap  
k<sup>th</sup> node from both ends.

Eg \* for  $k-1$  times iterate f & mark n1, then iterate s & f  
(as it is 1 indexed) till f is not NULL, once null mark s as n2.  
swap(n1, n2)



( $f \rightarrow next == NULL$ , so s is n2)

swap(15, 25)

Result ⇒

code →

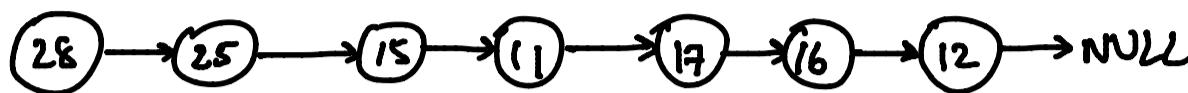
```
1 class Solution {
2 public:
3     ListNode* swapNodes(ListNode* head, int k) {
4         ListNode *slow = head, *fast = head, *n1 = head;
5         // finding n1
6         for(int i=0; i<k-1; i++){
7             fast = fast->next;
8             n1 = fast;
9         }
10        // finding n2 (i.e slow)
11        while(fast->next!=NULL){
12            fast = fast->next;
13            slow = slow->next;
14        }
15        // swapping
16        int n1_val = n1->val;
17        n1->val = slow->val;
18        slow->val = n1_val;
19        return head;
20    }
21};
```

$Tc \rightarrow O(n)$

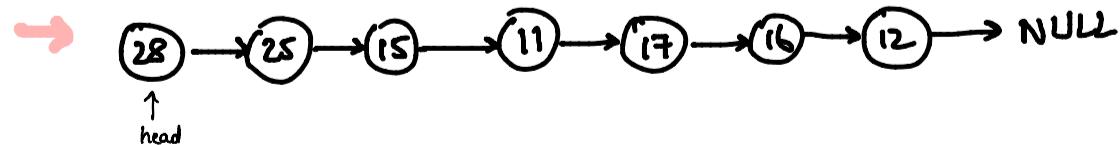
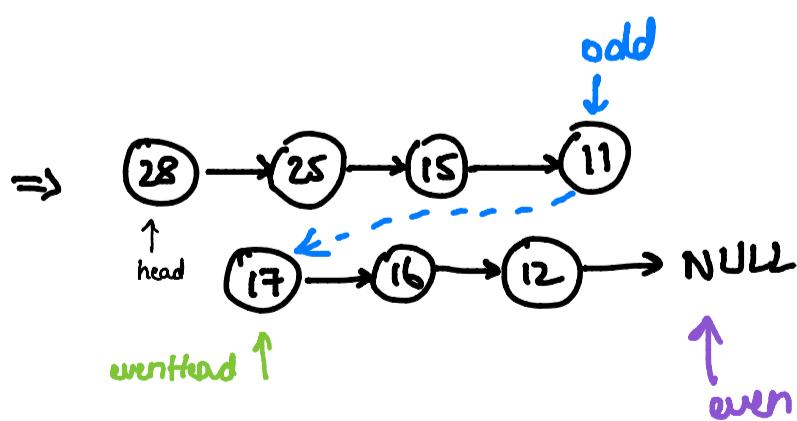
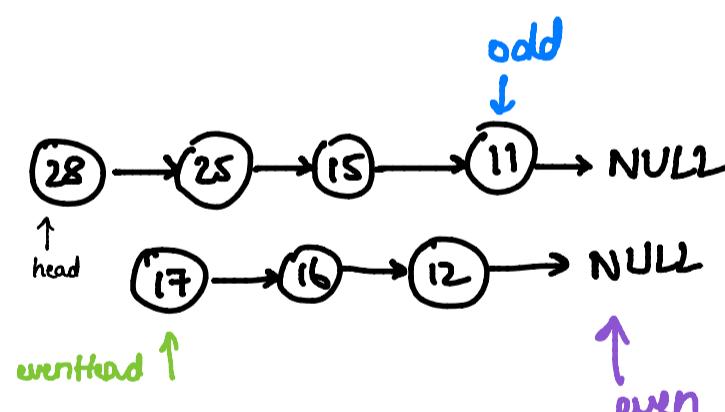
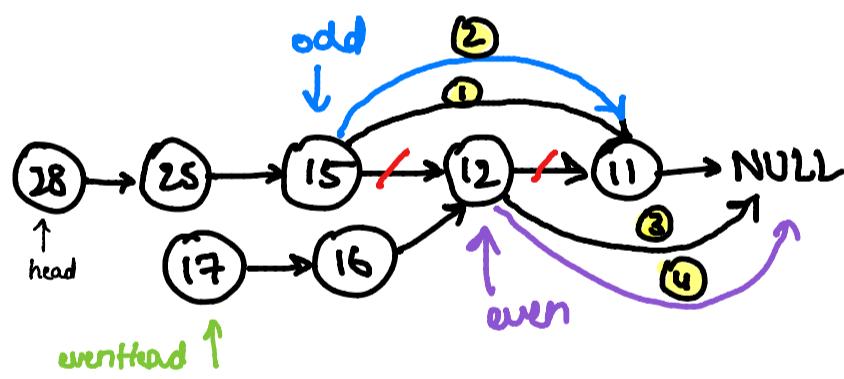
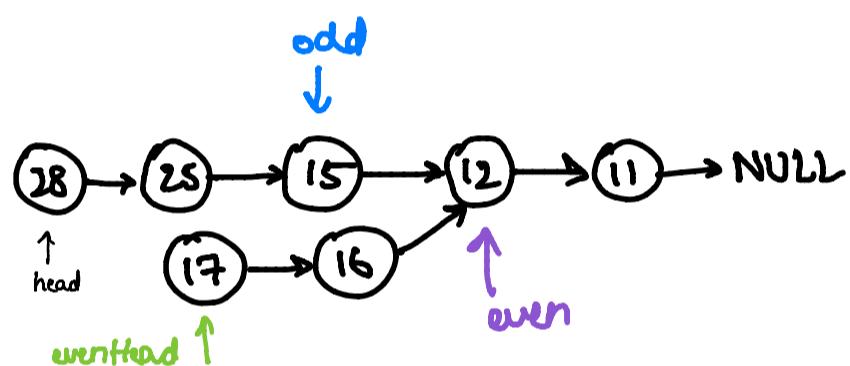
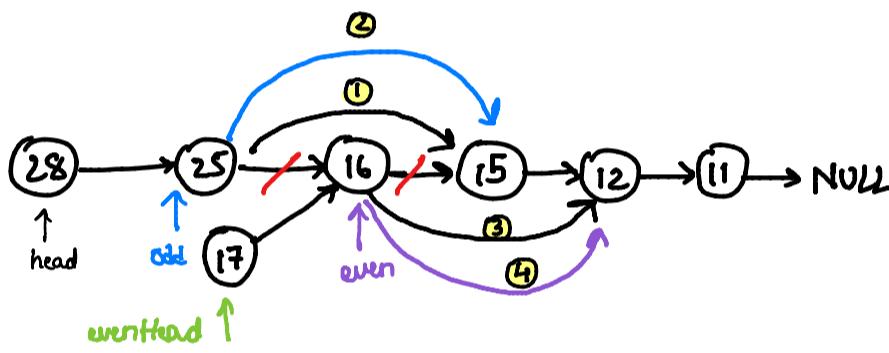
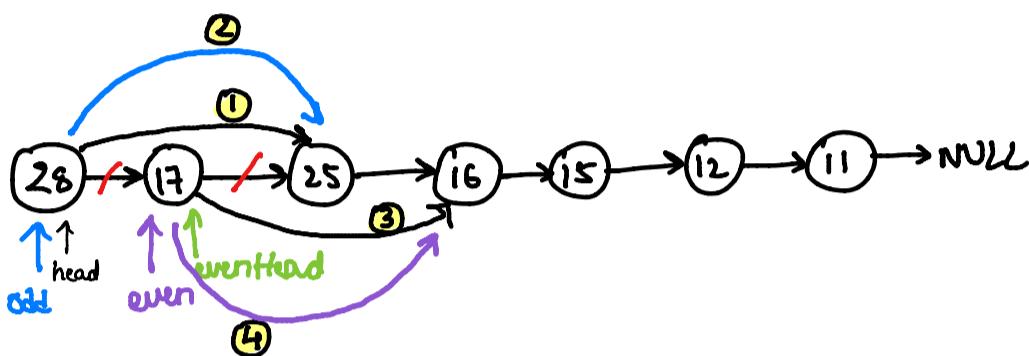
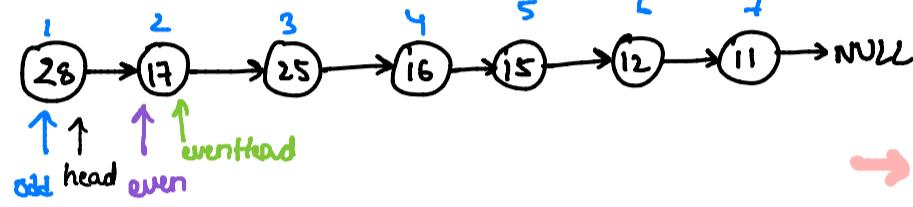
$Sc \rightarrow O(1)$

### 13 Odd Even Linked List →

gives a linkedlist group all odd indices nodes followed by even nodes



⇒



$\text{odd} \rightarrow \text{next} = \text{evenhead}$

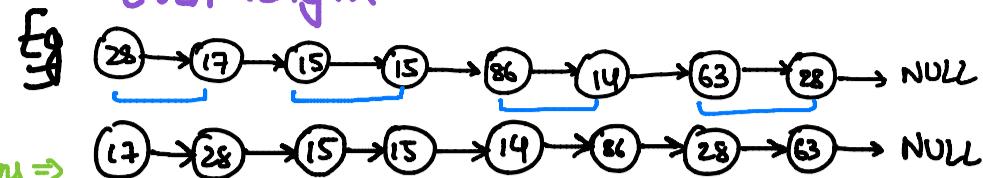
code →

```
1 class Solution {
2 public:
3     ListNode* oddEvenList(ListNode* head) {
4         if(!head) return NULL;
5
6         ListNode *even = head->next;
7         ListNode *odd = head;
8         ListNode *evenHead = even;
9
10        while(even && even->next){
11            odd->next=even->next;
12            odd=odd->next;
13            even->next=odd->next;
14            even=even->next;
15        }
16
17        // like odd and even lists
18        odd->next = evenHead;
19        return head;
20    }
21};
```

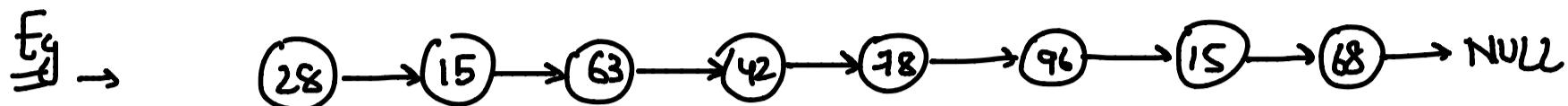
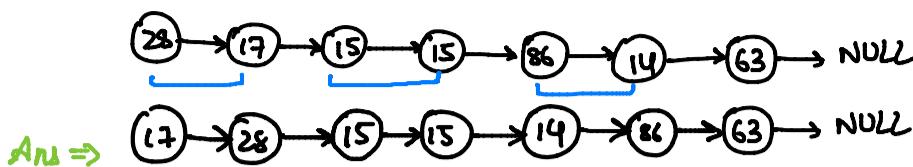
$T_C \rightarrow O(n)$   
 $S_C \rightarrow O(1)$

(14) Swap Nodes in Pairs → Given a linkedlist swap adjacent nodes.

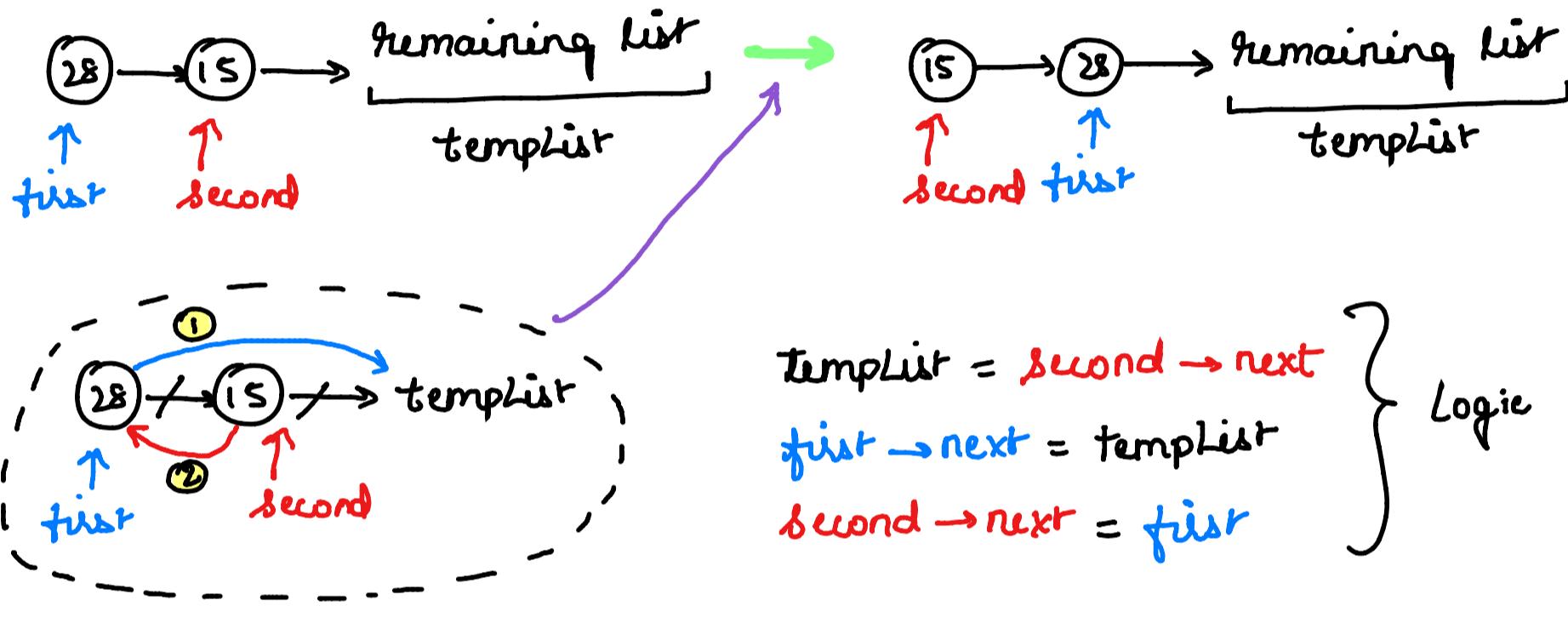
Even length



odd length



Consider for 1st pair,



Solve recursively for all pairs.

Code →

```
● ● ●  
1 class Solution {  
2 public:  
3     ListNode* SwapAdjacentNodes(ListNode* head)  
4     {  
5         if(head==NULL || head->next==NULL)  return head;  
6         ListNode *first = head;  
7         ListNode *second = head->next;  
8         // start logic  
9         ListNode *tempList = SwapAdjacentNodes(second->next);  
10        first->next = temp;  
11        second->next = first;  
12        return second;  
13    }  
14    ListNode* swapPairs(ListNode* head) {  
15        return SwapAdjacentNodes(head);  
16    }  
17};
```

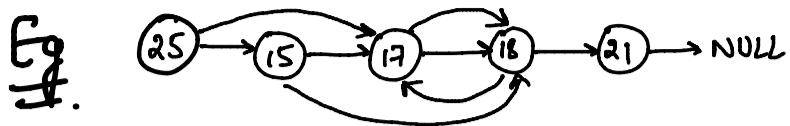
TC → O(N)

SC → O(1)

Recursive stack → O(N/2)  
 $\approx O(N)$

# 15 Copy list with random pointer

Given a list, clone & return.



mp	
Old	New
25	25
15	15
17	17
18	18
21	21
NULL	NULL

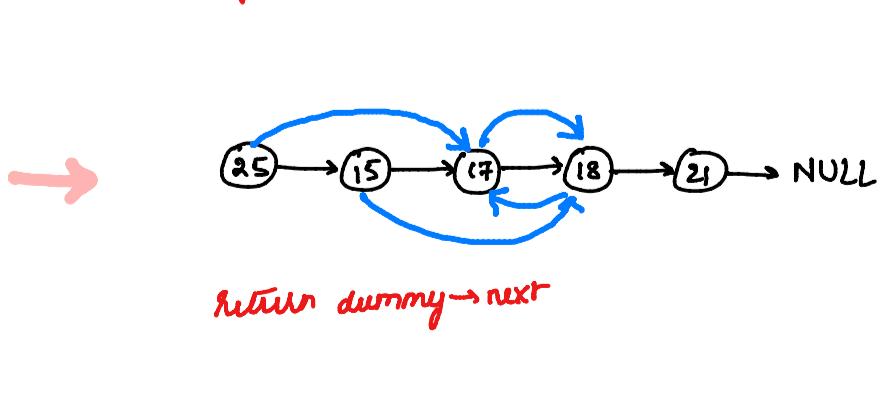
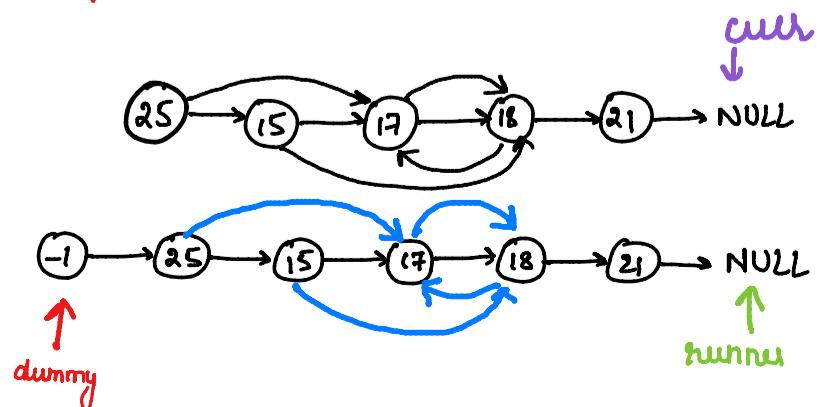
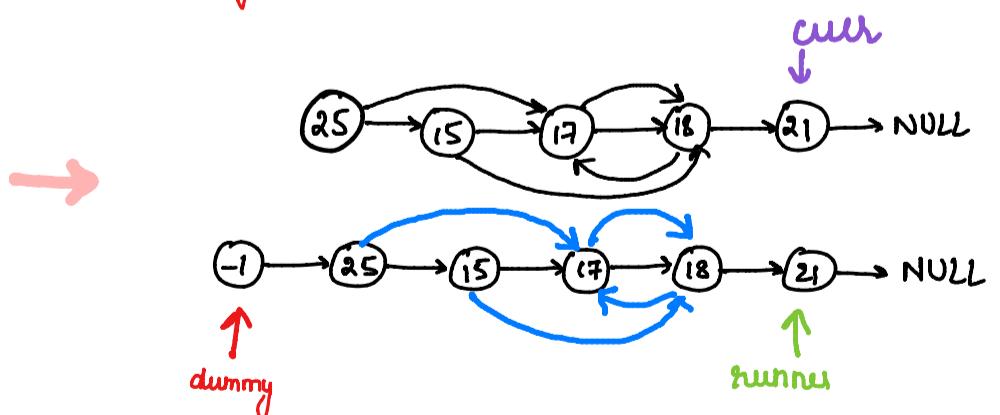
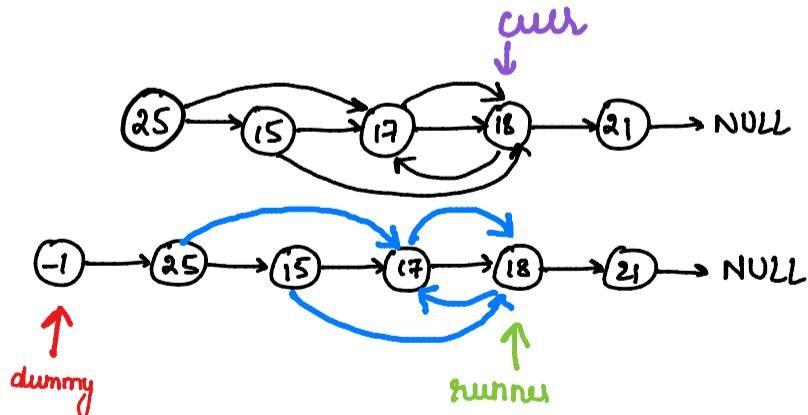
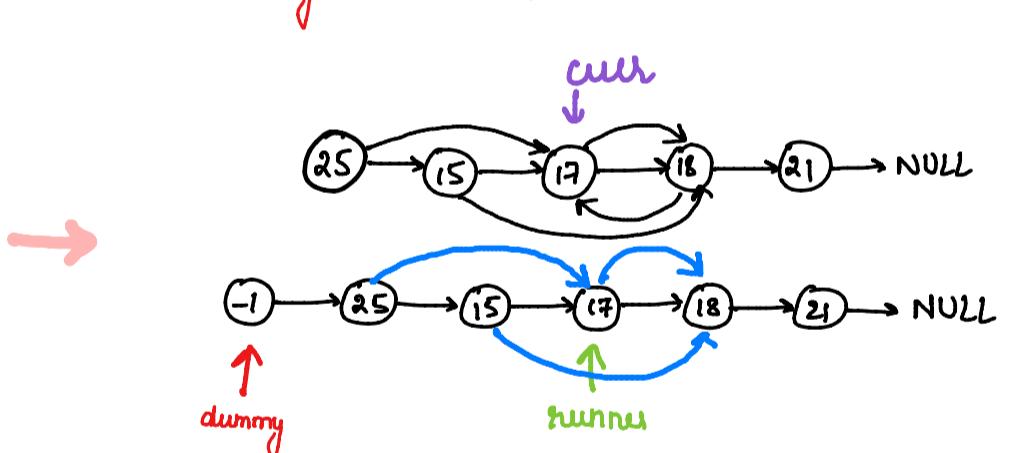
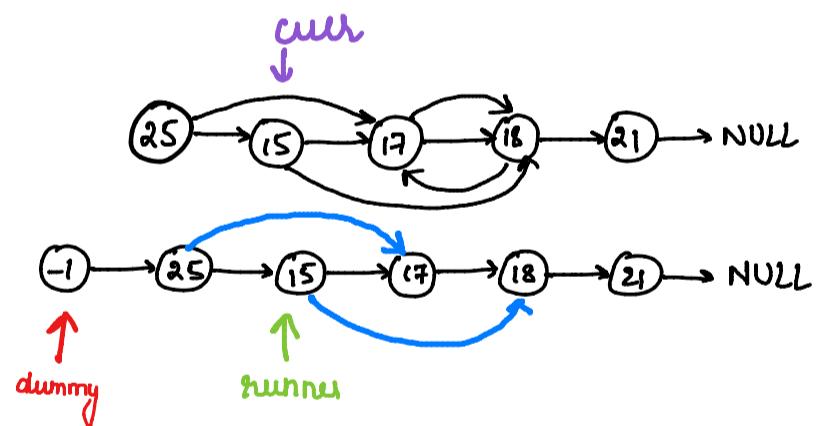
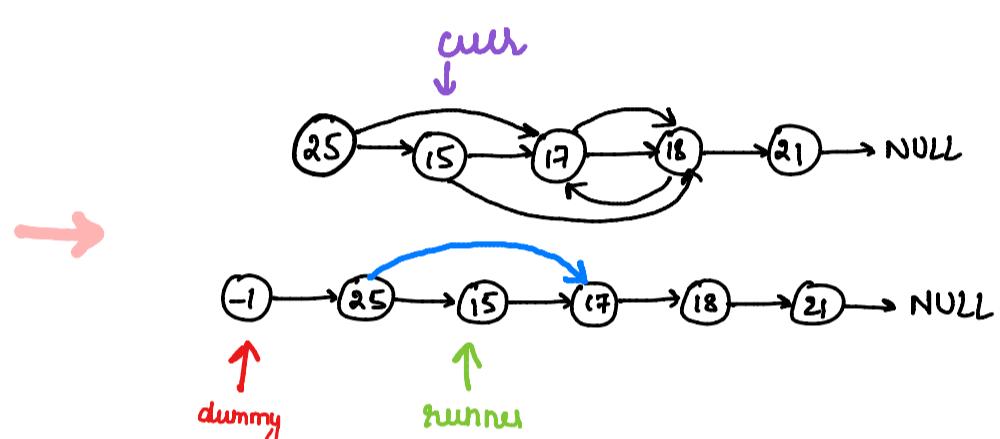
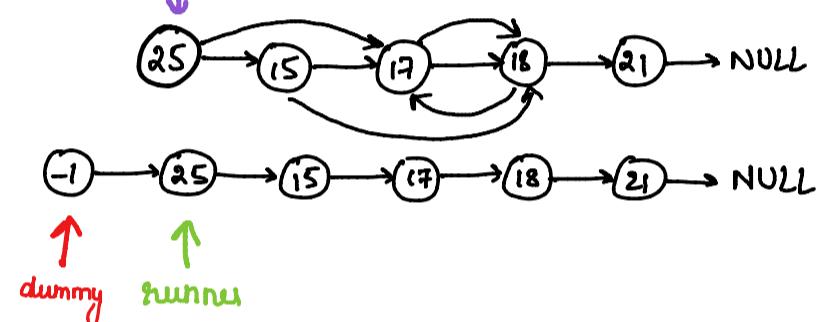
→ In 1st iteration create the list without random pointer & also maintain hashmap for mapping node pointed by random pointer

→ In 2nd iteration use map to link node pointed by random pointer

After 1st iteration →



→ curr (iterate till curr is not null)



Code →

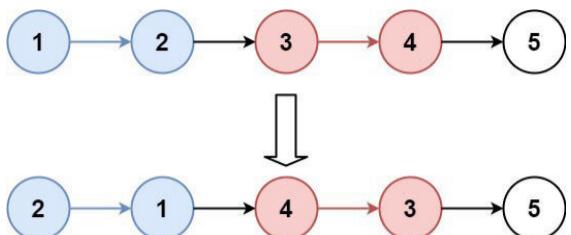
$T_C \rightarrow O(n)$

$SC \rightarrow O(n)$



```
1 class Solution {
2 public:
3     Node* copyRandomList(Node* head) {
4
5         unordered_map<Node*, Node*> mp;
6         Node *dummy = new Node(100001);
7         Node *runner = dummy, *curr = head;
8
9         // initial iteration
10        while(curr != NULL){
11            Node *newNode = new Node(curr->val);
12            runner->next = newNode;
13            mp[curr] = newNode;
14            curr = curr->next;
15            runner = runner->next;
16        }
17
18        // setting starting points in both lists
19        curr = head;
20        runner = dummy->next;
21
22        // setting the random pointers
23        while(curr != NULL){
24            if(curr->random != NULL)
25                runner->random = mp[curr->random];
26            runner = runner->next;
27            curr = curr->next;
28        }
29
30        return dummy->next;
31    }
32};
```

## 16 Reverse Nodes in K-Group



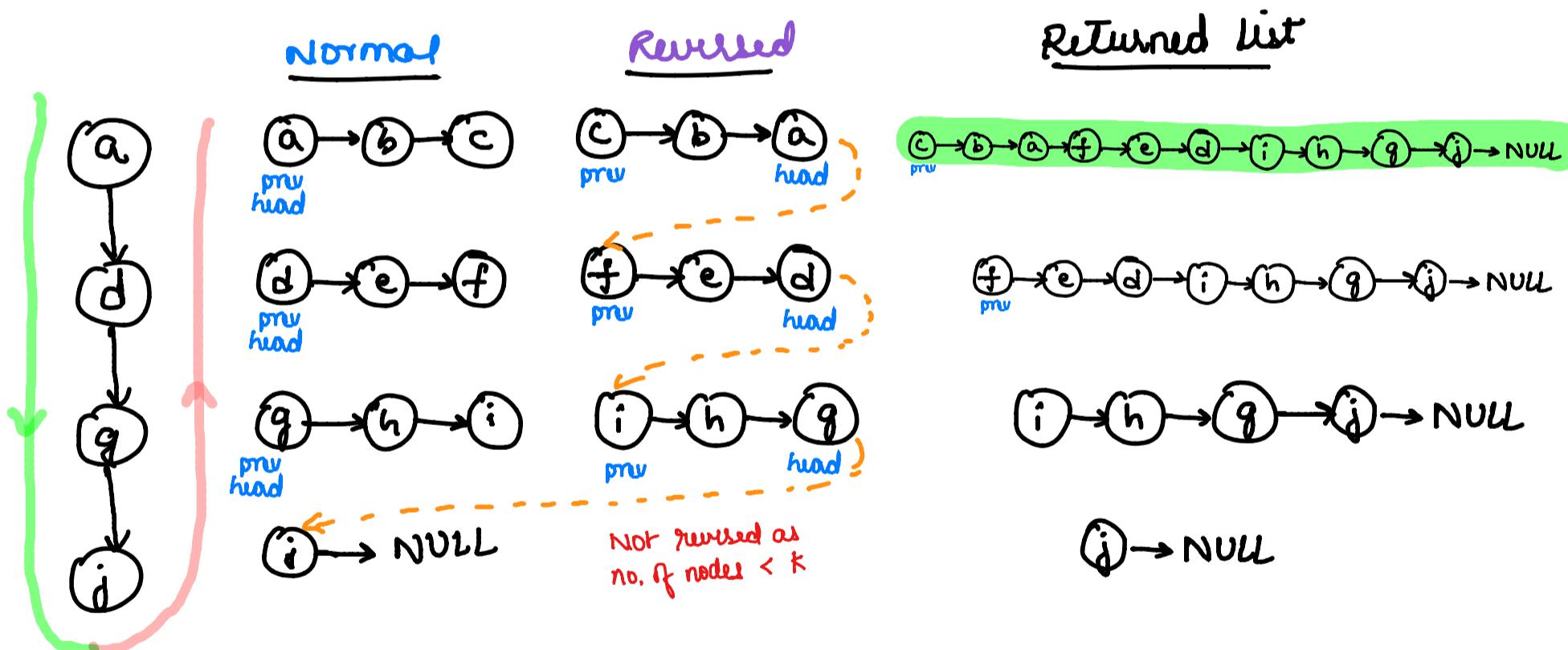
$\hookrightarrow k=2$

Given a linkedlist  $\& K$ , return a list with reversed nodes by  $K$ -groups.

Eg.  $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow \text{NULL}$

$k = 3$

$\hookrightarrow \Rightarrow c \rightarrow b \rightarrow a \rightarrow f \rightarrow e \rightarrow d \rightarrow i \rightarrow h \rightarrow g \rightarrow j \rightarrow \text{NULL}$



\* consider the case if  $f \rightarrow e \rightarrow d$ , then  $\text{tempList} = i \rightarrow h \rightarrow g$   
 linking would happen as  $\text{head} \rightarrow \text{next} = \text{tempList}$  & this list  
 would become  $\text{tempList}$  to  $c \rightarrow b \rightarrow a$

code →

$T_C \rightarrow O(n)$

$S_C \rightarrow O(1)$

```
● ● ●  
1 class Solution {  
2 public:  
3     ListNode* reverseList(ListNode* head)  
4     {  
5         ListNode *prev = NULL, *curr = head, *temp;  
6         while (curr!=NULL)  
7         {  
8             temp = curr->next;  
9             curr->next = prev;  
10            prev = curr;  
11            curr = temp;  
12        }  
13        return prev;  
14    }  
15  
16    ListNode* reverseInGroups(ListNode* head, int k)  
17    {  
18        ListNode *curr = head;  
19        int currlen = 1;  
20        if(head == NULL) return head;  
21        while(curr->next!=NULL && currlen<k ){  
22            curr=curr->next;  
23            currlen+=1;  
24        }  
25        if(currlen<k)  return head;  
26        ListNode *tempNode = curr->next;  
27        curr->next = NULL;  
28  
29        // start linking  
30        ListNode *tempList = reverseInGroups(tempNode,k);  
31        ListNode *prev = reverseList(head);  
32        head->next = tempList;  
33        return prev;  
34    }  
35  
36    ListNode* reverseKGroup(ListNode* head, int k) {  
37        return reverseInGroups(head,k);  
38    }  
39};
```

# 17 Design linked list → Implementation of Doubly Linked list

Code →

```

● ● ●

class Node{
public:
    int val;
    Node* prev;
    Node* next;
    Node(int val){
        this->val=val;
        prev = nullptr;
        next = nullptr;
    }
};

class MyLinkedList {
public:
    Node *head;
    Node *tail;
    MyLinkedList(){
        head = nullptr;
        tail = nullptr;
    }

    int get(int index){
        if(head == NULL)    return -1;
        Node *temp = head;
        int count = 0;
        while(temp!=NULL){
            temp=temp->next;
            count++;
        }
        if(index>=count)    return -1;
        temp = head;
        while(temp != NULL && index>0){
            temp=temp->next;
            index--;
        }
        return temp->val;
    }

    void addAtHead(int val){
        Node *newNode = new Node(val);
        if(head == NULL){
            head = newNode;
            tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }

    void addAtTail(int val){
        Node *temp = head;
        if(head == NULL){
            Node *newNode = new Node(val);
            head = newNode;
            tail = newNode;
            return;
        }
        while(temp->next!=NULL){
            temp = temp->next;
        }
        Node *newNode = new Node(val);
        temp->next = newNode;
        newNode->prev = temp;
        tail = newNode;
    }
}

```

```

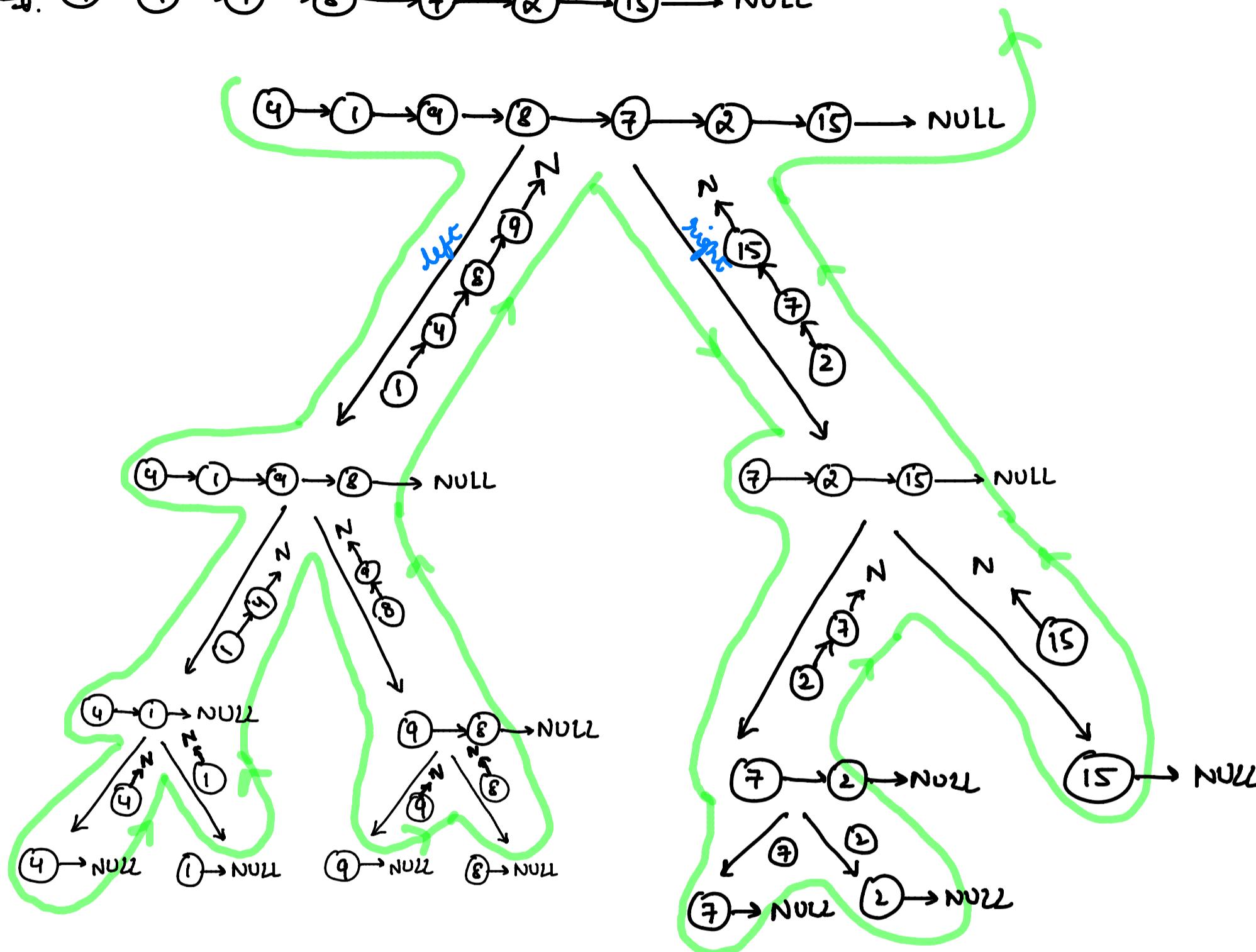
void addAtIndex(int index, int val){
    Node *temp = head;
    int count = 0;
    while(temp != NULL){
        temp = temp->next;
        count++;
    }
    if(index>count) return ;
    if(index==0){
        addAtHead(val);
        return;
    } else if(count == index){
        addAtTail(val);
        return;
    } else {
        temp = head;
        while(temp != NULL && index>0){
            temp = temp->next;
            index--;
        }
        Node* newNode = new Node(val);
        Node* temp2 = temp->prev;
        temp->prev->next = newNode;
        temp->prev = newNode;
        newNode->prev = temp2;
        newNode->next = temp;
    }
}

void deleteAtIndex(int index) {
    Node* temp = head;
    int count = 0;
    while(temp != NULL){
        temp=temp->next;
        count++;
    }
    if(index>=count) return;
    if(count==1 && index==0){
        head = NULL;
        return;
    } else if(count-1 == index){
        tail = tail->prev;
        tail->next = NULL;
        return;
    } else {
        if(index==0){
            head->next->prev = NULL;
            head = head->next;
            return;
        }
        temp=head;
        while(temp!=NULL && index>0){
            temp = temp->next;
            index--;
        }
        Node* temp2 = temp->next;
        temp->prev->next = temp2;
        temp->next->prev = temp->prev;
    }
}

```

18 Sort List → By following Merge Sort.

Eg.  $4 \rightarrow 1 \rightarrow 9 \rightarrow 8 \rightarrow 7 \rightarrow 2 \rightarrow 15 \rightarrow \text{NULL}$



In the last step while returning from both branches we have,

$\text{left} = 1 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow \text{NULL}$  &  $\text{right} = 2 \rightarrow 7 \rightarrow 15 \rightarrow \text{NULL}$

so create dummy node & merge, i.e.  $(-1 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 15 \rightarrow \text{NULL})$

return  $\text{dummy} \rightarrow \text{next}$ ,  $(1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 15 \rightarrow \text{NULL})$

\* The same happens at every intermediate merge.

code →

$Tc \rightarrow O(m+n)$

$Sc \rightarrow O(n)$



```
1 class Solution {
2 public:
3     ListNode* merge(ListNode* l1, ListNode* l2) {
4         ListNode *dummy = new ListNode(-1);
5         ListNode *curr = dummy;
6         while(l1 && l2){
7             if(l1->val < l2->val){
8                 curr->next = l1;
9                 l1 = l1->next;
10            } else {
11                curr->next = l2;
12                l2 = l2->next;
13            }
14            curr = curr->next;
15        }
16        if(l1) curr->next = l1;
17        if(l2) curr->next = l2;
18
19        return dummy->next;
20    }
21
22    ListNode* sortList(ListNode* head) {
23        if(!head || !head->next) return head;
24
25        ListNode *slow = head;
26        ListNode *fast = head->next;
27        while(fast && fast->next) {
28            slow = slow->next;
29            fast = fast->next->next;
30        }
31        // dividing the lists into 2 parts
32        fast = slow->next;
33        slow->next = NULL;
34
35        // sort & merge
36        head = sortList(head);
37        fast = sortList(fast)
38        return merge(head, fast);
39    }
40};
```