Linked List

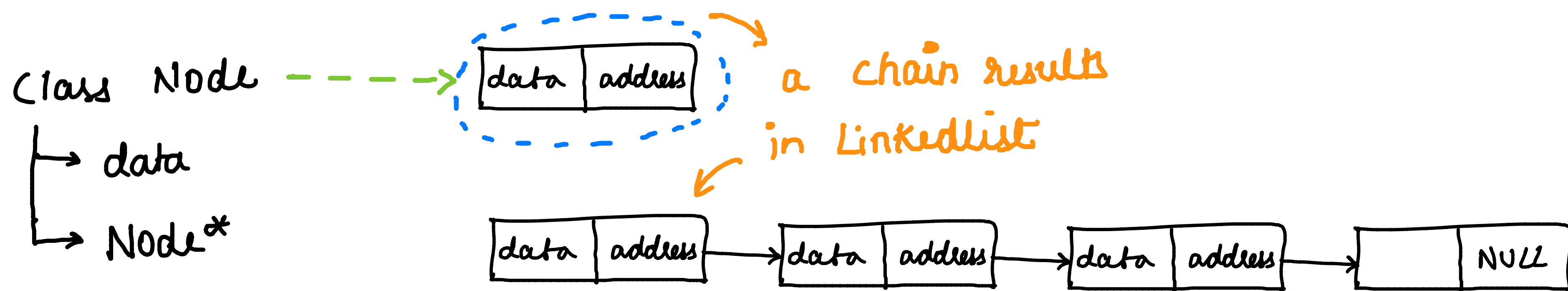
- Karun Karthik

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.



Advantages

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

Disadvantages

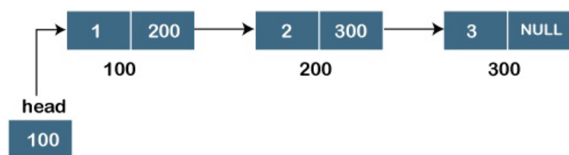
1. More memory usage due to address pointer.
2. Slow traversal compared to arrays.
3. No reverse traversal in singly linked list
4. No random access.

Real life Applications

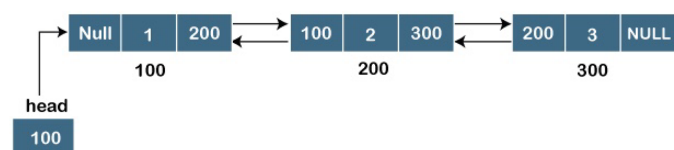
1. Previous & next page in browser
2. Image Viewer
3. Music player

Types

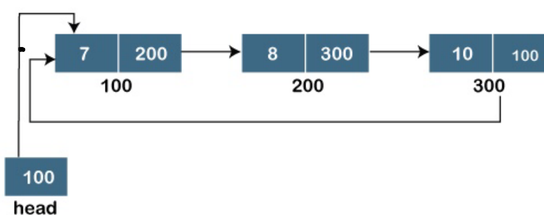
1. Singly linkedlist



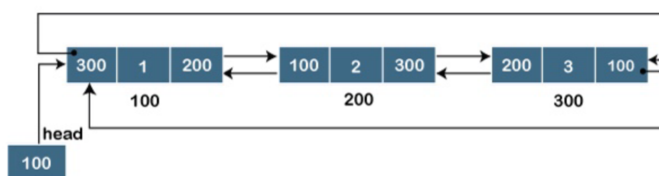
2. Doubly linkedlist



3. Circular Linkedlist



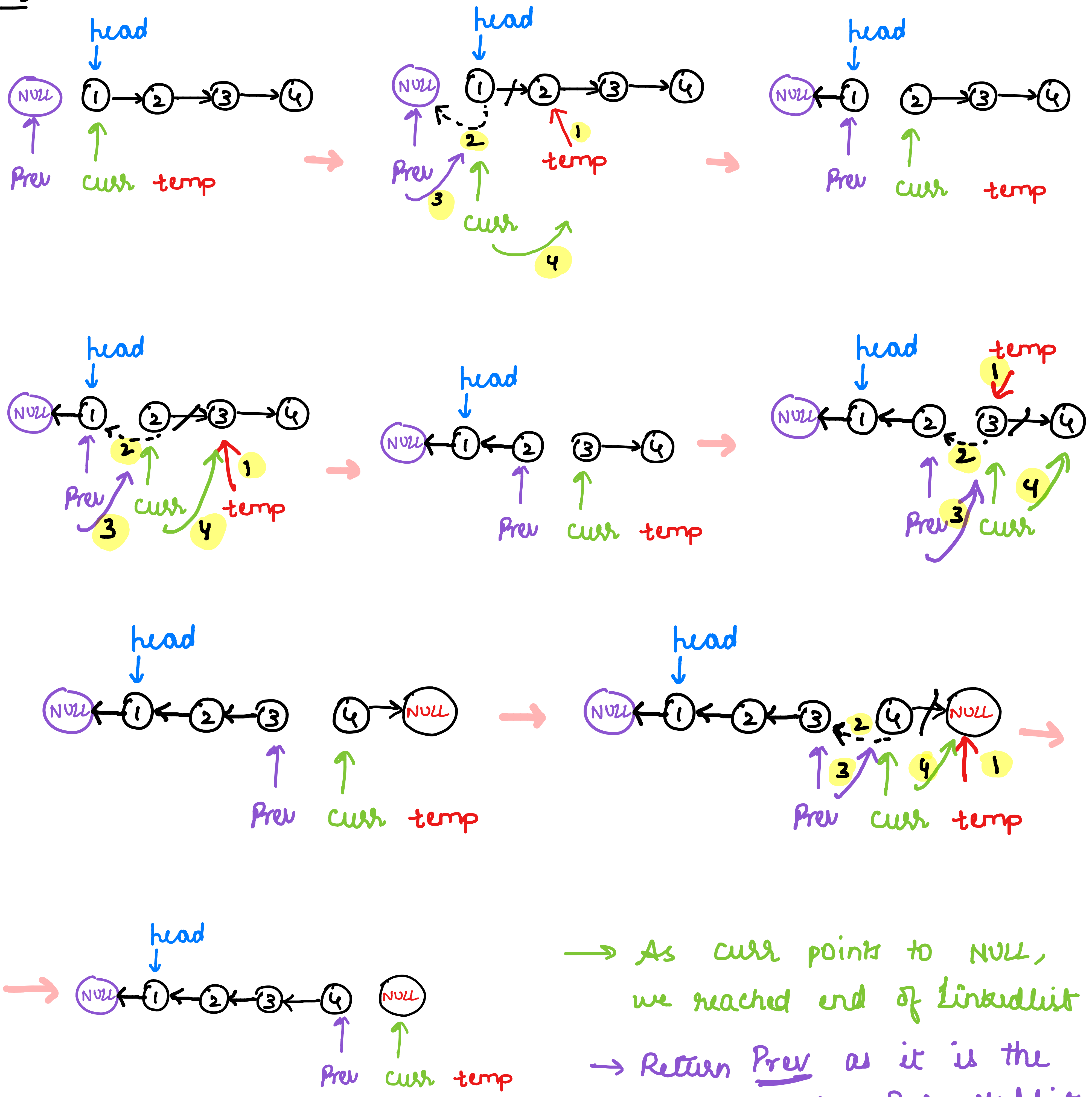
4. Doubly circular linkedlist



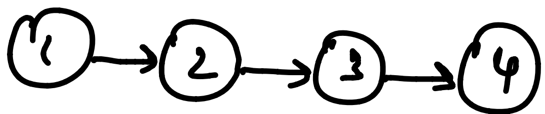
① Reverse a linkedlist → given a linkedlist, return reversed list.

Eg 1 → 2 → 3 → 4 ⇒ 4 → 3 → 2 → 1

Sol)



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

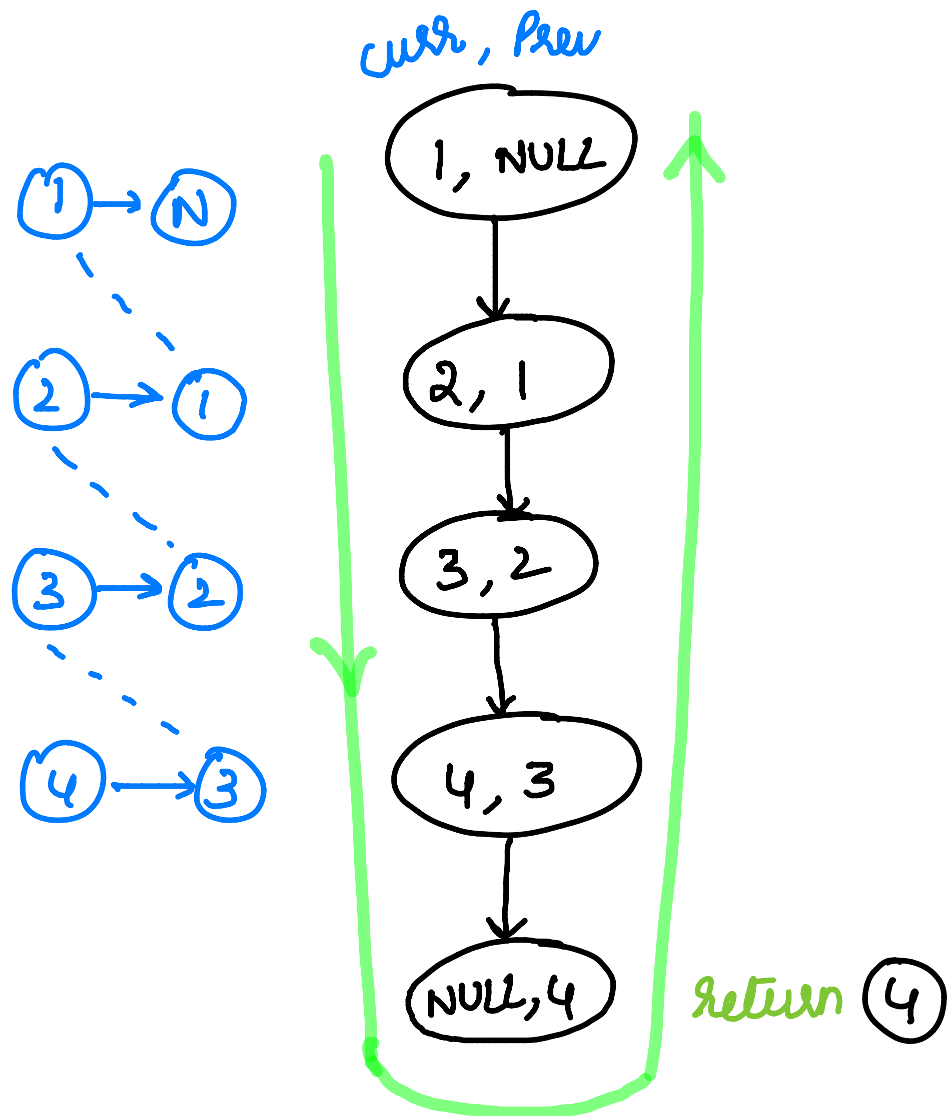
4 → next = 3

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 as curr & curr as prev

https://www.linkedin.com/in/karun-karthik

Code →

TC → O(n)

SC → O(1)

```
// Iterative ->
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *curr=head, *temp;
        while(curr){
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }
};


// Recursive ->
class Solution {
public:
    ListNode* reverseLinker(ListNode* curr, ListNode* prev) {
        if(curr==NULL)
            return prev;
        ListNode* newNode = curr->next;
        curr->next = prev;
        return helper(newNode, curr);
    }

    ListNode* reverseLinker(ListNode* head) {
        return helper (head, NULL);
    }
};
```

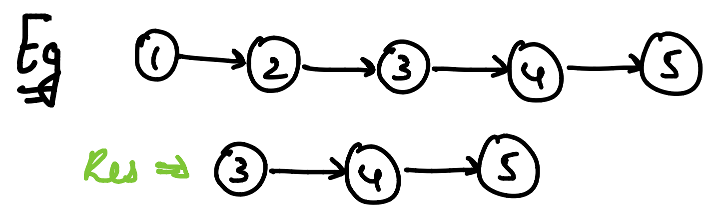
https://www.linkedin.com/in/karun-karthik

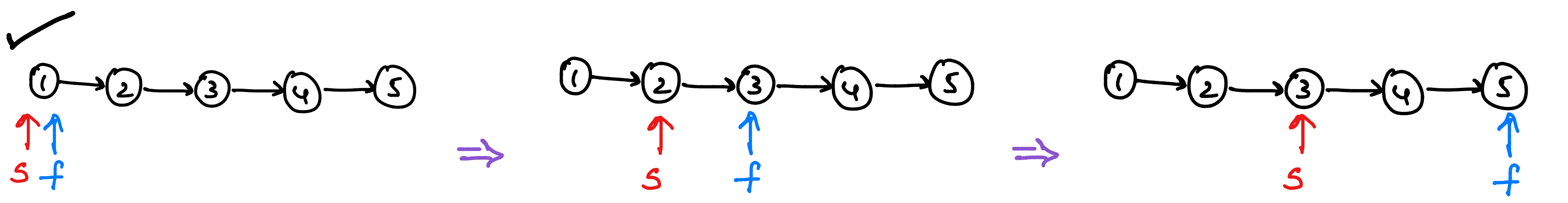
② Middle of Linkedlist → given the head, return middle node.

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

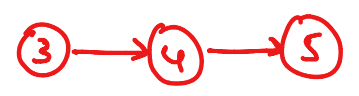
Approach 2 → use 2 pointers → slow (moves by 1), fast (moves by 2) } By time fast reaches end, slow points to the mid.

Eg





as fast reached end return slow.



Code

```
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 };
```

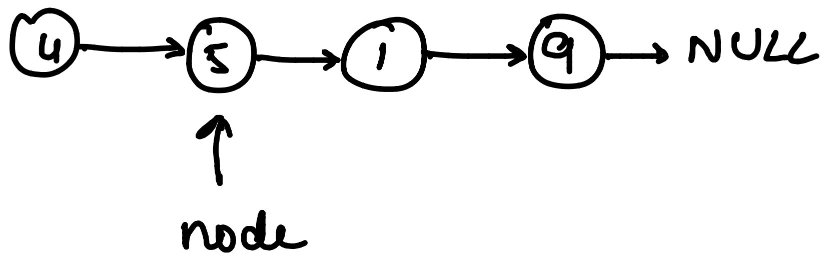
TC → O(n)
SC → O(1)

③ Delete node in a linkedlist →

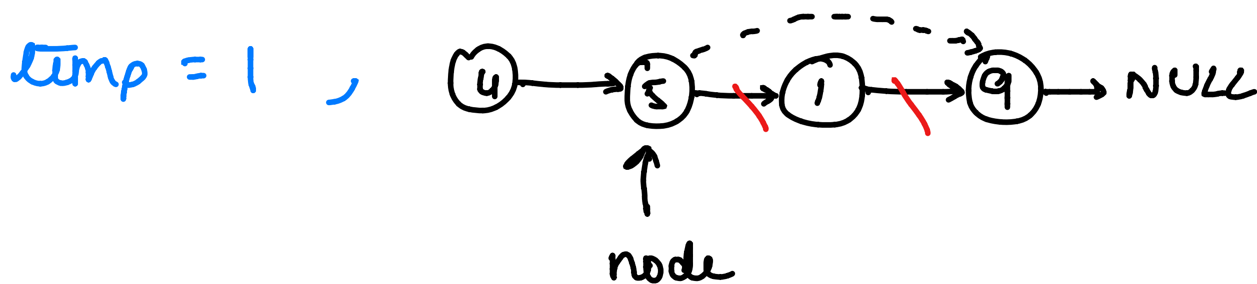
given a linkedlist's node, delete it.

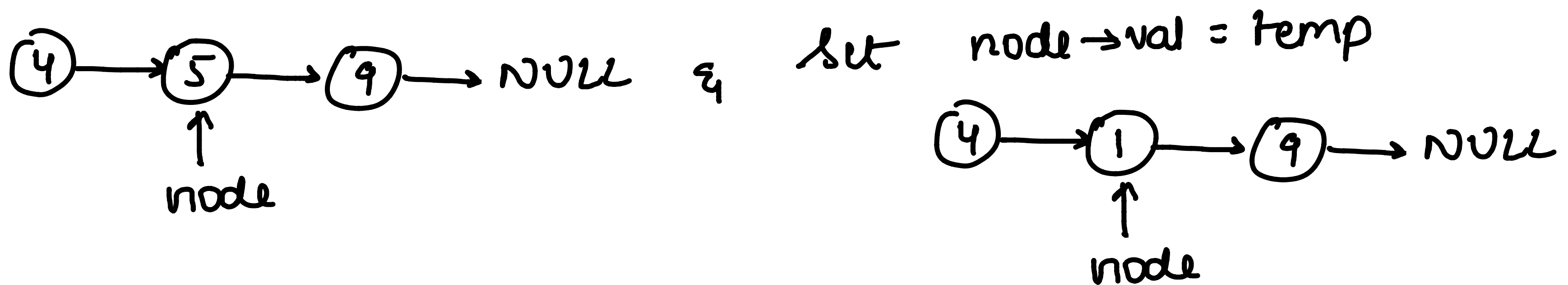
→ copy node's next node's val into a temp variable
→ skip the node→next node
→ copy the temp variable's value into the node.

Eg



temp = 1 ,



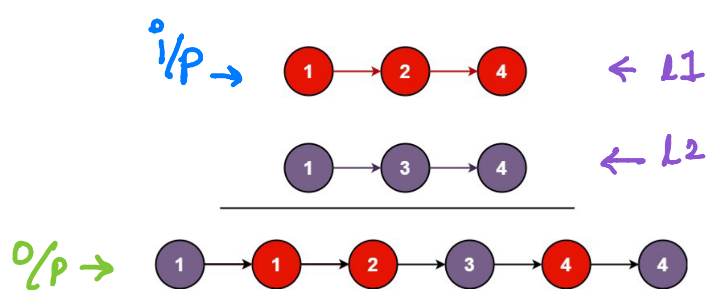


code →

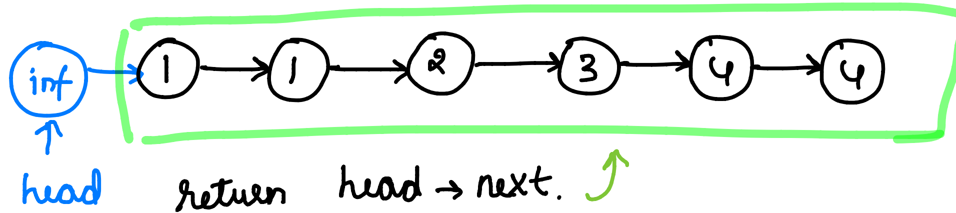
```
1  class Solution {
2  public:
3      void deleteNode(ListNode* node) {
4          int val = node->next->val;
5          node->next = node->next->next;
6          node->val = val;
7      }
8  };
```

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, the 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 };
```

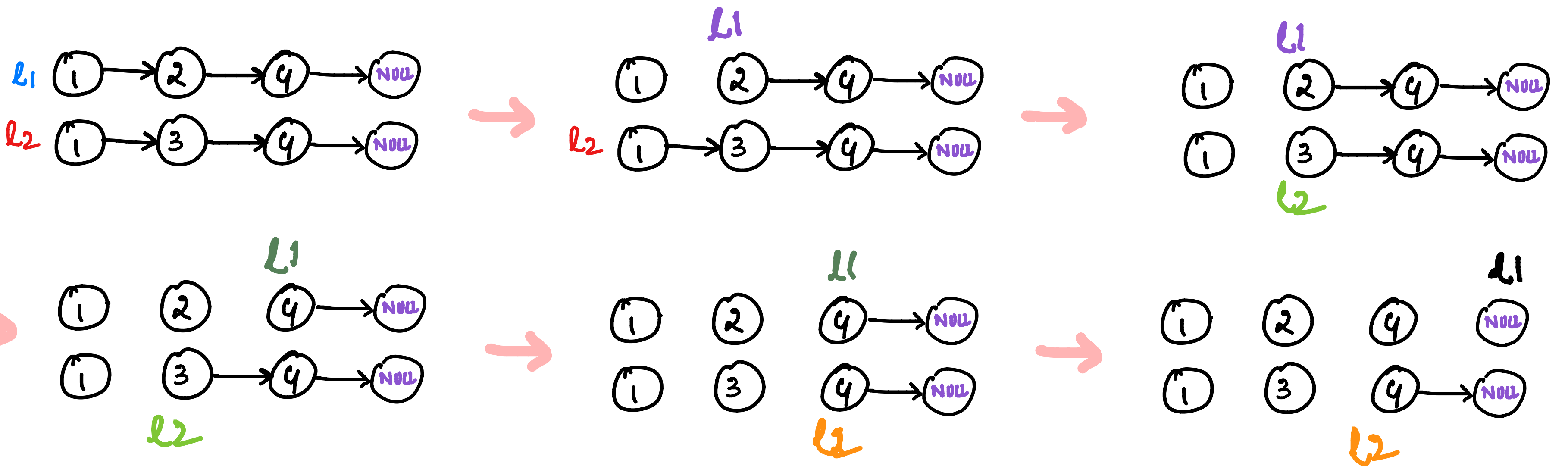
TC → O(m+n)

SC → O(m+n)

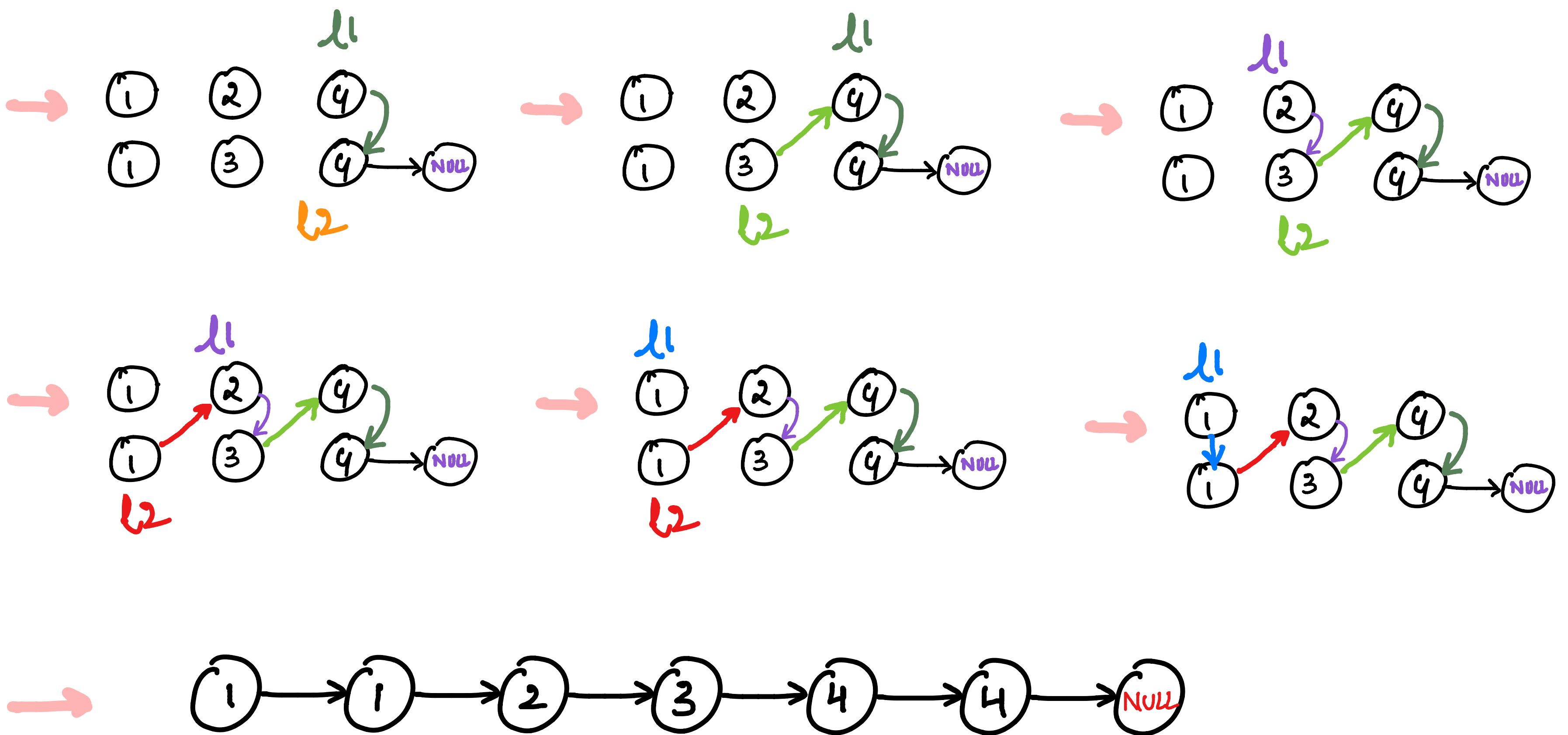
Recursive Code →

```cpp
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;
        }
    }
};
```

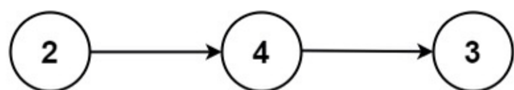
Dry Run

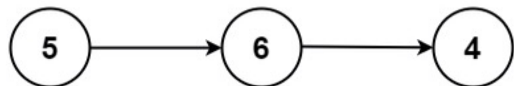


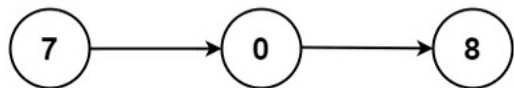
As l1 is null, return l2



⑤ Add two Numbers





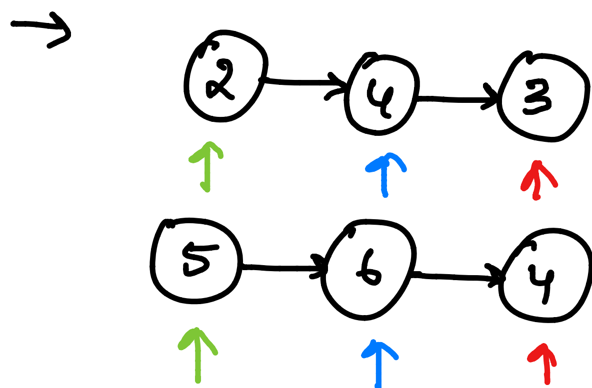


342
+465
807

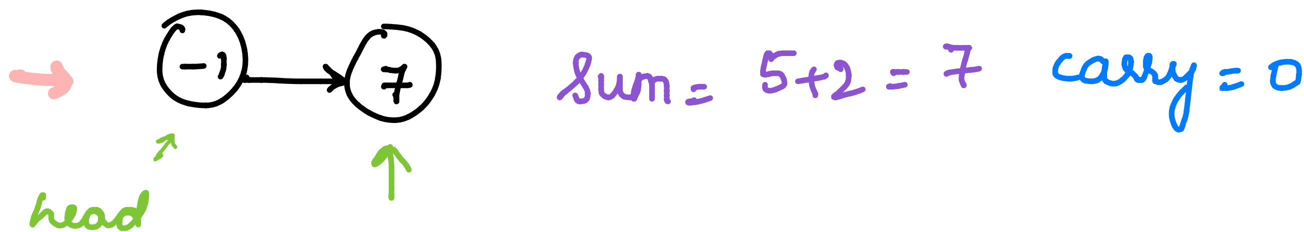
Given 2 lists in reverse order, add them and return the sum.

TC → O(m+n)
SC → O(max(m,n))

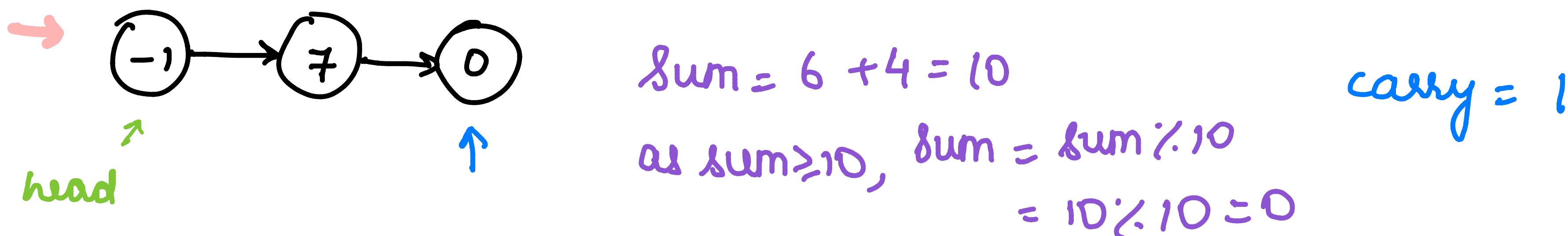
Initially,



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

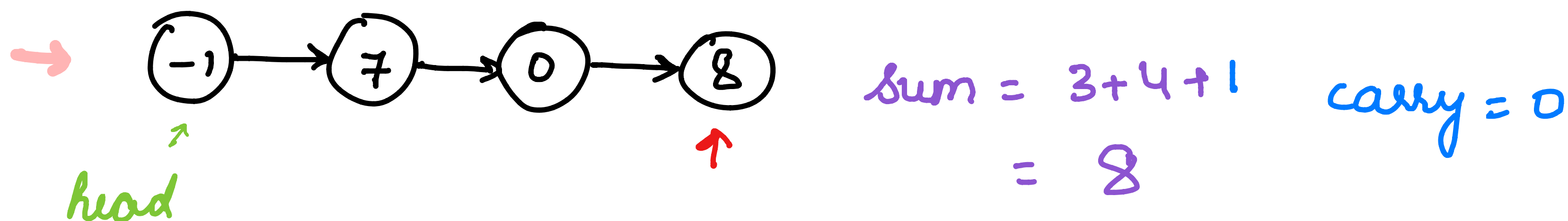


sum = 5+2 = 7 carry = 0



sum = 6 +4 = 10
as sum≥10, sum = sum % 10
= 10 % 10 = 0

carry = 1



sum = 3+4+1
= 8

carry = 0

code

```
class Solution {
public:

    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {

        ListNode* dummyNode;
        ListNode* head;
        dummyNode = head = new ListNode(-1);
        if(!l1)
            return l2;
        if(!l2)
            return l1;

        int carry = 0;

        while(l1 || l2){
            int firstVal = l1 ? l1->val : 0;
            int secondVal = l2 ? l2->val : 0;

            int total = firstVal + secondVal + carry;
            carry = total / 10;
            total = total % 10;

            ListNode* newNode = new ListNode(total);
            dummyNode->next = newNode;

            dummyNode = dummyNode->next;

            l1 = l1 ? l1->next : l1;
            l2 = l2 ? l2->next : l2;
        }

        if(carry)
            dummyNode->next = new ListNode(1);

        return head->next;
    }
};
```

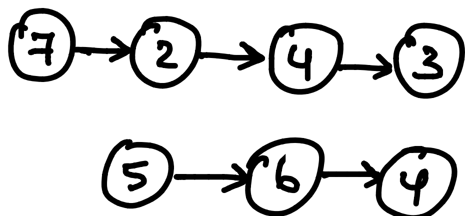
⑥ Add two numbers II

→ Problem solving approach is same as previous problem

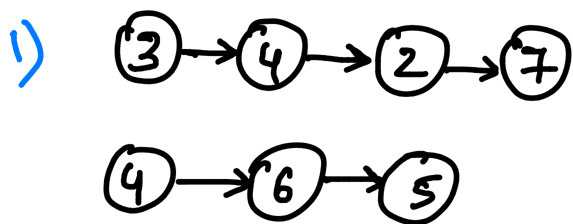
→ Points to note :

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

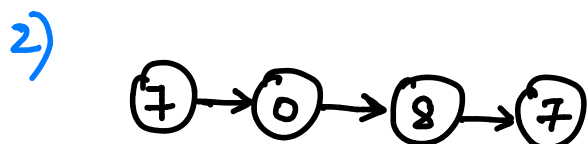
Eg →



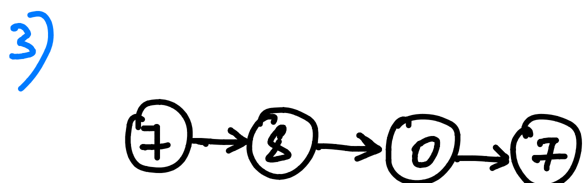
1)



2)



3)



Result ↑

Code →

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* curr = head;
        ListNode* temp = NULL;
        while(curr!=NULL)
        {
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }

    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        l1=reverseList(l1); // O(n)
        l2=reverseList(l2); //O(n)
        ListNode* dummyNode;
        ListNode* head;
        dummyNode = head = new ListNode(-1);
        if(!l1)
            return l2;
        if(!l2)
            return l1;

        int carry = 0;

        while(l1 || l2){
            int firstVal = l1 ? l1->val : 0;
            int secondVal = l2 ? l2->val : 0;

            int total = firstVal + secondVal + carry;
            carry = total / 10;
            total = total % 10;

            ListNode* newNode = new ListNode(total);
            dummyNode->next = newNode;

            dummyNode = dummyNode->next;

            l1 = l1 ? l1->next : l1;
            l2 = l2 ? l2->next : l2;
        }

        if(carry)
            dummyNode->next = new ListNode(1);

        return reverseList(head->next); //o(max(m,n))
    }
};
```

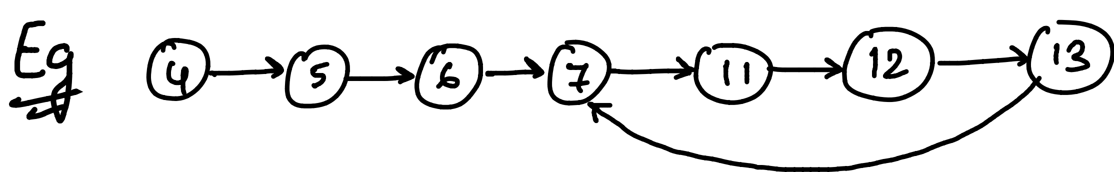
(7) Linked list cycle →

TC → O(n)
SC → O(n)

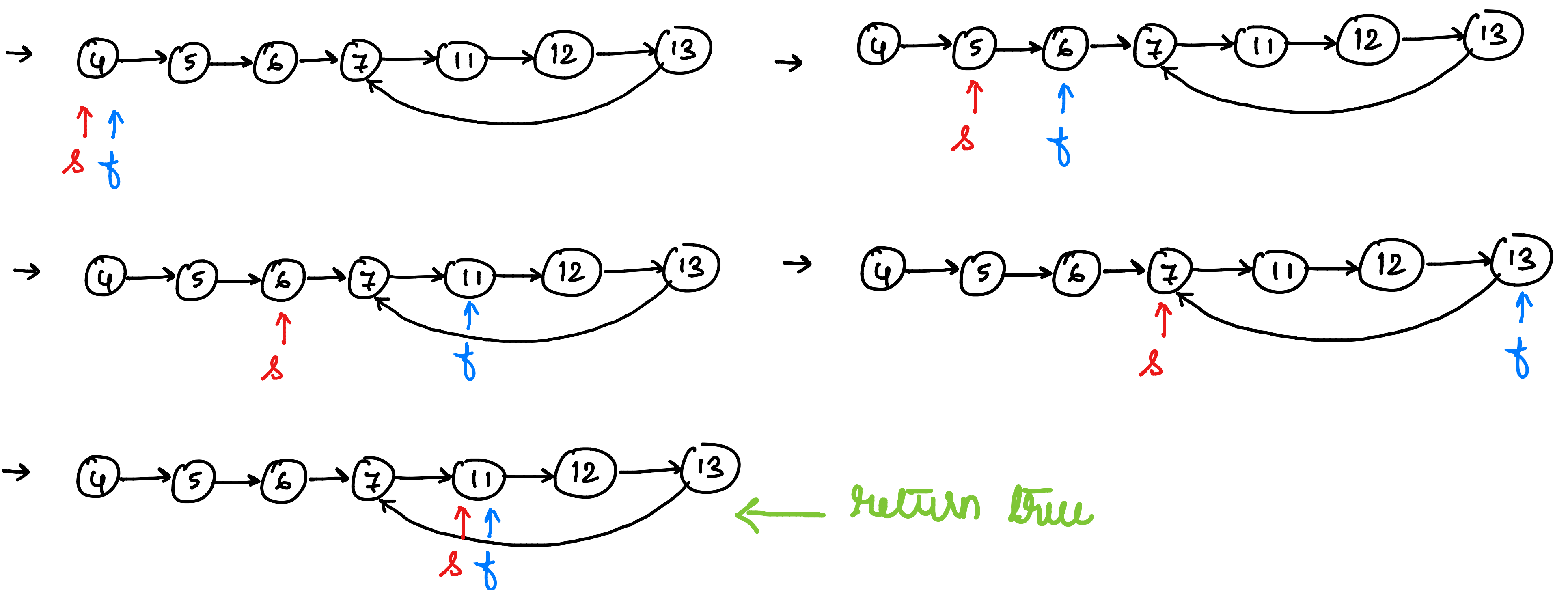
Approach - 1 → Create a set of nodes & insert every Node into it, if already exist then return True else false.

Approach - 2 → using fast & slow pointer TC → O(n) SC → O(1)

Eg



keep iterating till fast→next && fast→next→next exist.

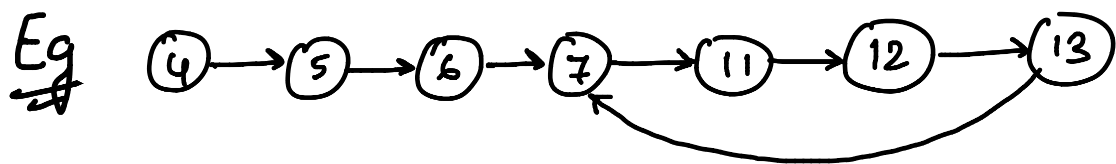


Code →

```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head==NULL) return false;
        ListNode *fast = head, *slow = head;
        while(fast->next!=NULL && fast->next->next!=NULL)
        {
            fast=fast->next->next;
            slow=slow->next;
            if(fast==slow)  return true;
        }
        return false;
    }
};
```

(8) Linked list cycle II → return the node where cycle begins.

Eg



result = (7)

keep iterating while fast→next && fast→next→next exist.

→ 4 → 5 → 6 → 7 → 11 → 12 → 13 (s, f at 4)

→ s at 5, f at 6

→ s at 6, f at 11

→ s at 7, f at 13

→ s, f at 11

← once slow == fast, then set slow = head, & move pointers by 1 unit

→ s at 4, f at 11

→ s at 5, f at 12

→ s at 6, f at 13

→ s, f at 7

→ when slow == fast, it denotes the node where cycle begins.

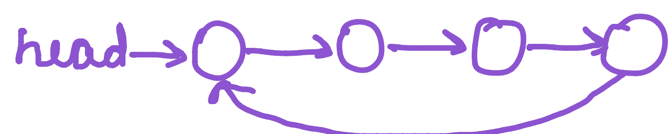
return slow; => (7)

code →

```cpp
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if(head==NULL) return NULL;
        ListNode *fast = head, *slow = head;
        while(fast->next!=NULL && fast->next->next!=NULL)
        {
            fast = fast->next->next;
            slow = slow->next;
            if(fast == slow)
            {
                slow = head;
                while (slow != fast)
                {
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
        return NULL;
    }
};
```

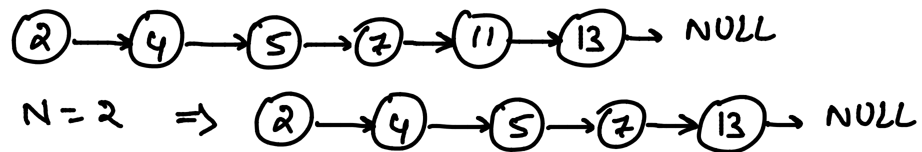
TC → $O(n) + O(n) = O(2n) = O(n)$

* worst case when its a loop



SC → $O(1)$

9 Remove Nth node from End of list



Approach - 1 → find length of list l, delete $l-n+1$ th node

Approach - 2 → a) Reverse b) Delete Nth node c) Reverse

code →

TC → O(n)

SC → O(1)

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *curr = head, *temp;
        while(curr){
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }

    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *dummy = new ListNode(-1);
        dummy->next = reverseList(head);
        head = dummy;
        ListNode *curr = head;
        ListNode *prev = NULL;
        // Iteration
        for(int i=0; i<n; i++)
        {
            prev = curr;
            curr = curr->next;
        }
        // Deletion
        prev->next = curr->next;
        return reverseList(head->next);
    }
};
```

(10) Pallindrome LinkedList

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 it value by value.

code →

```
class Solution {
public:
    ListNode* midNode(ListNode*head)
    {
        ListNode *fast = head, *slow = head;
        while(fast->next!=NULL and fast->next->next!=NULL){
            fast=fast->next->next;
            slow = slow->next;
        }
        return slow;
    }
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *curr = head, *temp;
        while(curr!=NULL)
        {
            temp=curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }
    bool compare(ListNode*l1,ListNode* l2)
    {
        while(l1!=NULL && l2!=NULL)
        {
            if(l1->val!=l2->val)    return false;
            l1 = l1->next;
            l2 = l2->next;
        }
        return true;
    }
    bool isPalindrome(ListNode* head) {
        if(head==NULL) return false;
        if(head->next == NULL) return true;
        ListNode *mid = midNode(head);
        ListNode *l2 = mid->next;
        mid->next = NULL;
        l2 = reverseList(l2);
        return compare(head,l2);
    }
};
```

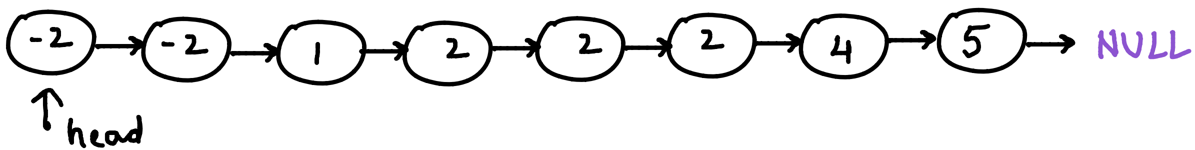
TC → O(n)

SC → O(1)

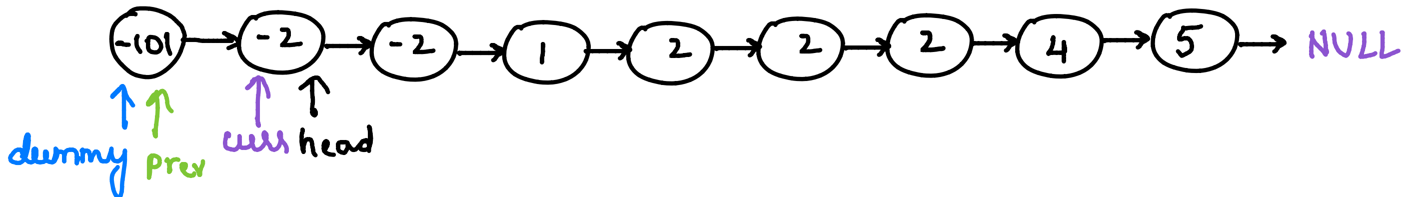
(1) Remove duplicates from sorted list →

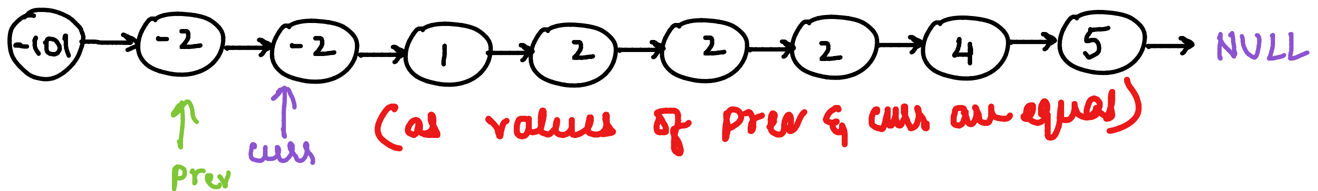
Given a linkedlist, return linkedlist without duplicates.

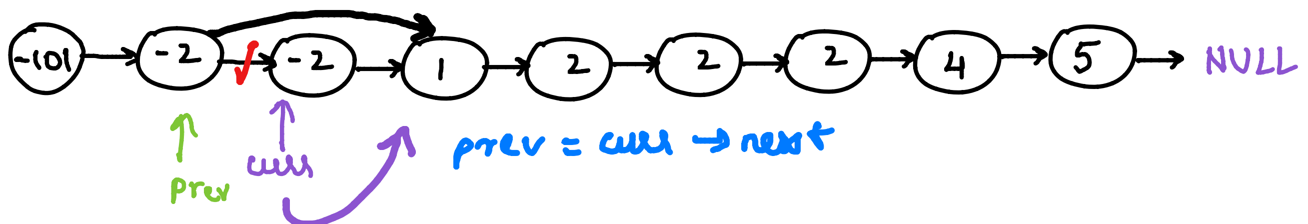
Eg

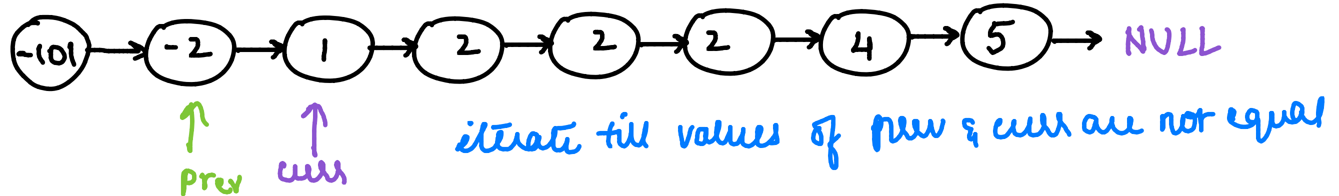


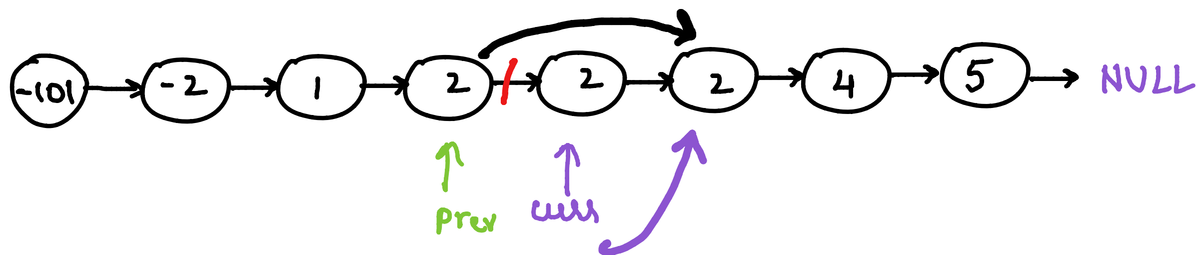
⇒

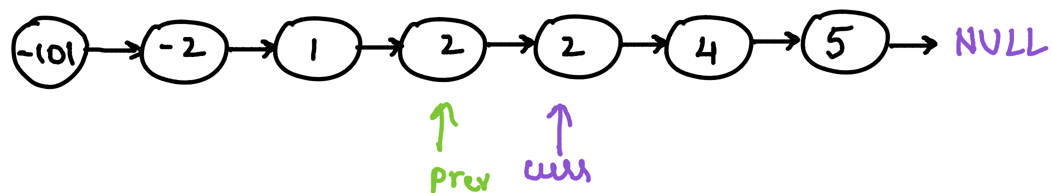


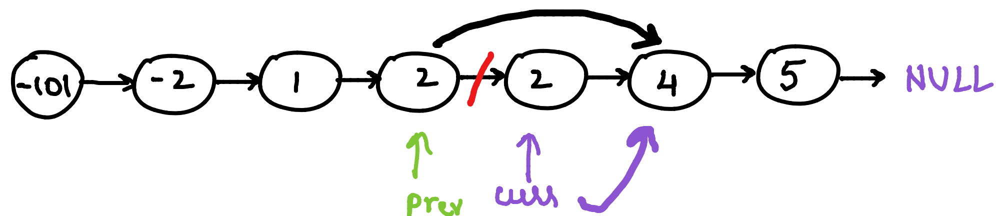


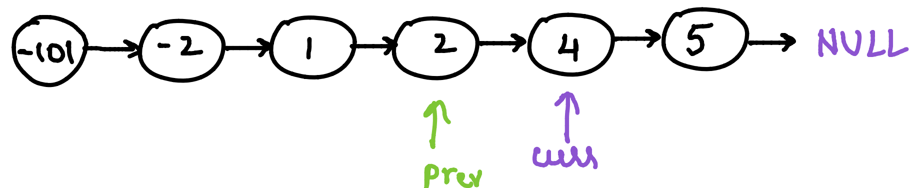












code →

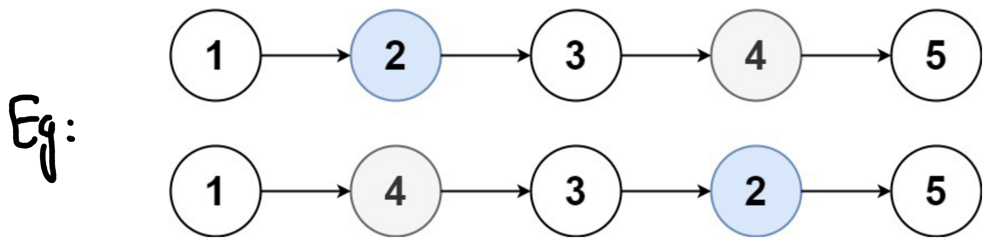
```cpp
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* dummy = new ListNode(101);
        dummy->next = head;
        ListNode* curr = head;
        ListNode* prev = dummy;
        while(curr!=NULL)
        {
            if(curr->val==prev->val){
                prev->next = curr->next;
                curr = curr->next;
            } else {
                prev = curr;
                curr = curr->next;
            }
        }
        return dummy->next;
    }
};


// Another approach

ListNode* deleteDuplicates(ListNode* head) {
    if(head==NULL || head->next==NULL) return head;
    ListNode *curr = head;
    while(curr->next!=NULL){
        if(curr->val == curr->next->val){
            curr->next = curr->next->next;
        } else {
            curr = curr->next;
        }
    }
    return head;
}
```

Tc → O(n)

Sc → O(1)

(12) Swapping Nodes in Linked List →

Eg:



k = 2

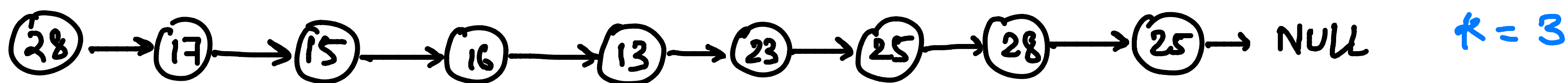
given a linkedlist swap k^{th} node from both ends.

Eg

* for K-1 times iterate f & mark n1, then iterate s & f till f is not NULL, once null mark S as n2.

(as it is 1 indexed)

swap (n1, n2)



k = 3

28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (n1 s f at 28) → 28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (s at 28, n1 f at 17)

28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (s at 28, n1 f at 15) → 28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (s at 17, n1 at 15, f at 16)

28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (n1 s at 15, f at 13) → 28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (n1 at 15, s at 16, f at 23)

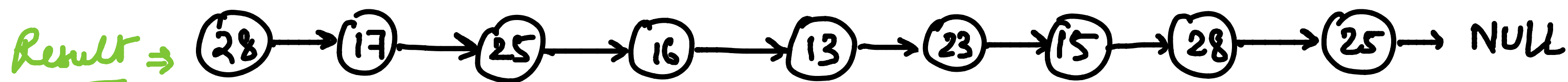
28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (n1 at 15, s at 13, f at 25) → 28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (n1 at 15, s at 23, f at 28)

28 → 17 → 15 → 16 → 13 → 23 → 25 → 28 → 25 → NULL (n1 at 15, s n2 at 25, f at 25)

(f → next == NULL, so s is n2)

swap (15, 25)

Result ⇒



code →

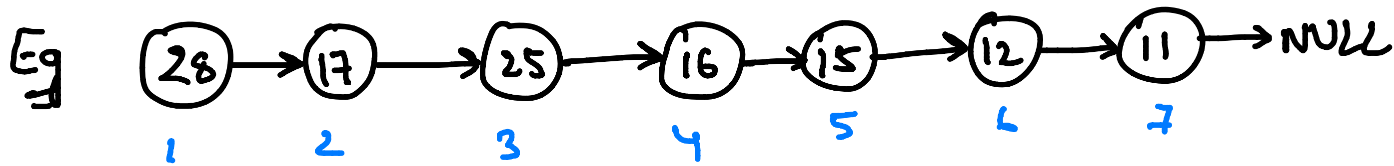
```cpp
class Solution {
public:
    ListNode* swapNodes(ListNode* head, int k) {
        ListNode *slow = head, *fast = head, *n1 = head;
        // finding n1
        for(int i=0; i<k-1; i++){
            fast = fast->next;
            n1 = fast;
        }
        // finding n2 (i.e slow)
        while(fast->next!=NULL){
            fast = fast->next;
            slow = slow->next;
        }
        // swapping
        int n1_val = n1->val;
        n1->val = slow->val;
        slow->val = n1_val;
        return head;
    }
};
```

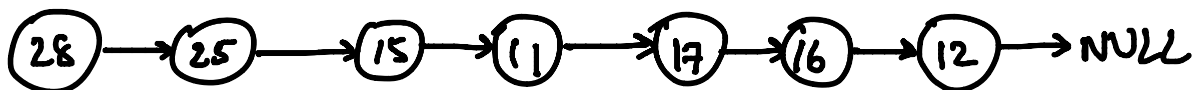
TC → O(n)
SC → O(1)

(13) Odd Even Linked List →

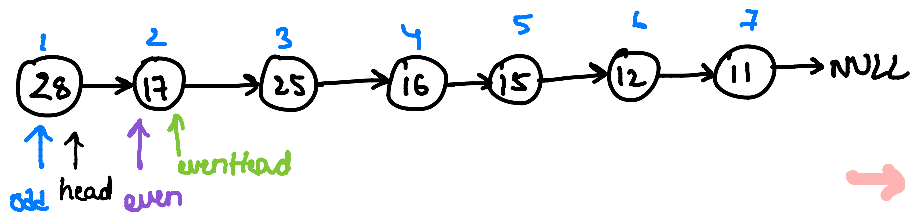
given a linkedlist group all odd indices nodes followed by even nodes

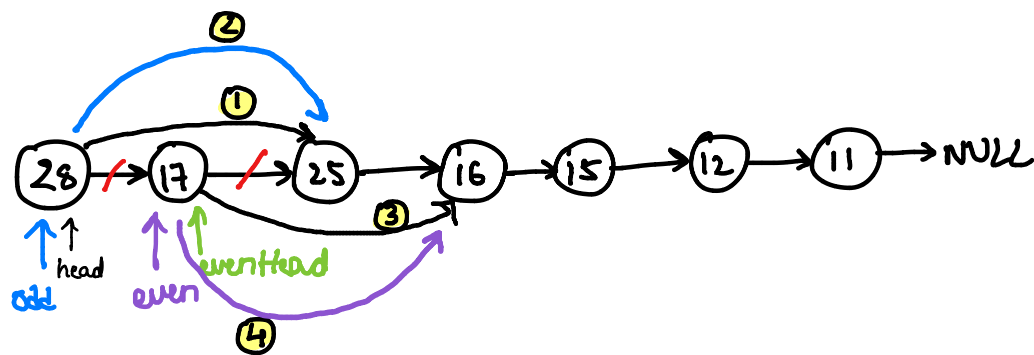
Eg

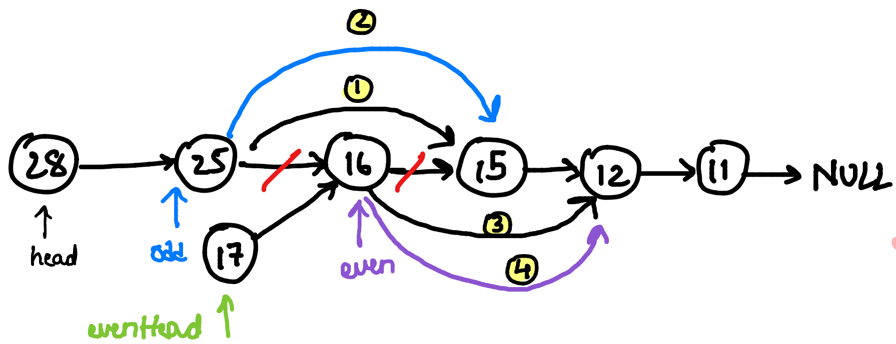


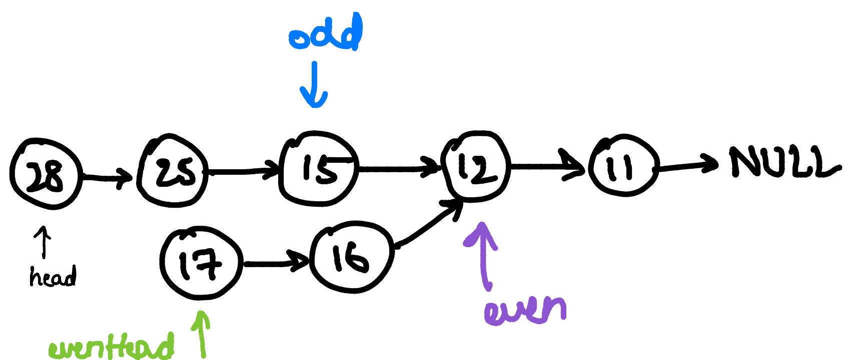


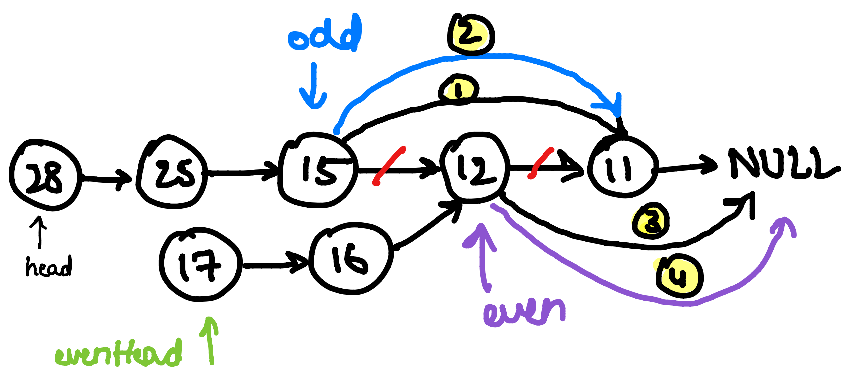
⇒

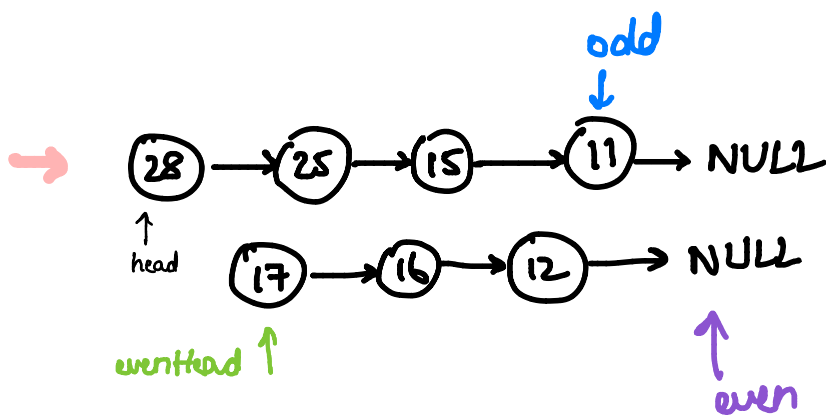




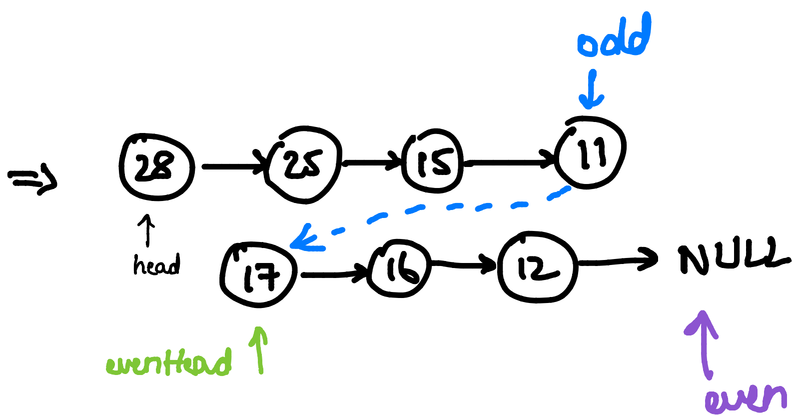


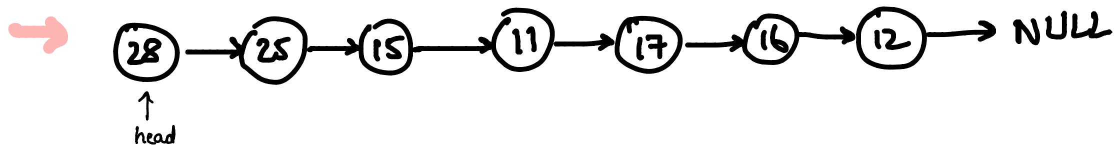






⇒





odd → next = evenHead

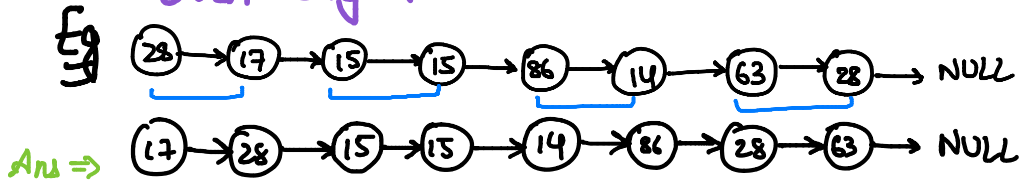
code →

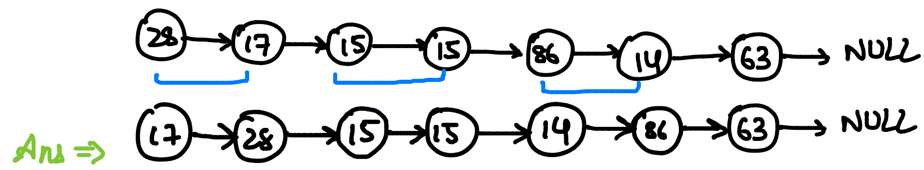
```cpp
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(!head) return NULL;

        ListNode *even =  head->next;
        ListNode *odd = head;
        ListNode *evenHead = even;

        while(even && even->next){
            odd->next=even->next;
            odd=odd->next;
            even->next=odd->next;
            even=even->next;
        }

        // like odd and even lists
        odd->next = evenHead;
        return head;
    }
};
```

TC → O(n)
SC → O(1)

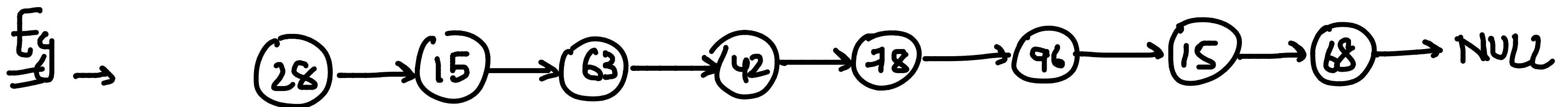
(14) Swap Nodes in Pairs → given a linkedlist swap adjacent nodes.

Eg

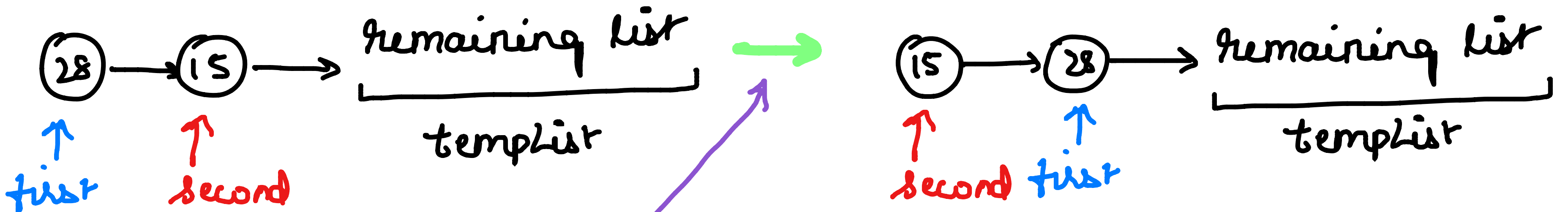


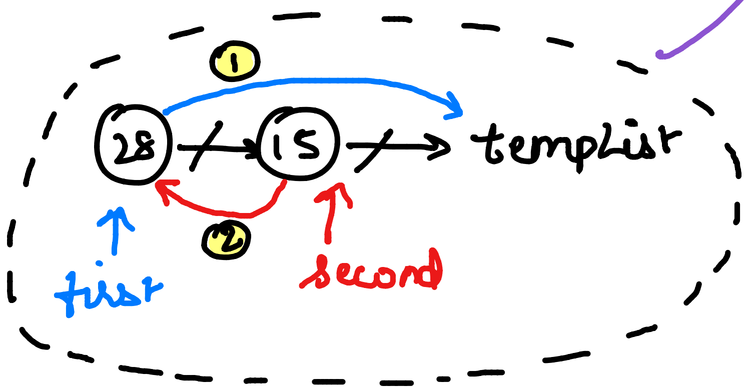


Eg →



Consider for 1st pair,





tempList = second → next
first → next = tempList
second → next = first

} Logic

Solve recursively for all pairs.

Code →

```
class Solution {
public:
    ListNode* SwapAdjacentNodes(ListNode* head)
    {
        if(head==NULL || head->next==NULL)  return head;
        ListNode *first = head;
        ListNode *second = head->next;
        // start logic
        ListNode *tempList = SwapAdjacentNodes(second->next);
        first->next = temp;
        second->next = first;
        return second;
    }
    ListNode* swapPairs(ListNode* head) {
        return SwapAdjacentNodes(head);
    }
};
```

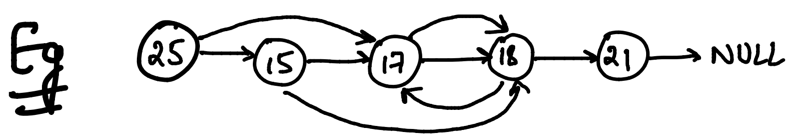
TC → O(N)

SC → O(1)

Recursive stack → O(N/2) ≃ O(N)

(15) Copy list with random pointer → *given a list, clone & return.*

Eg



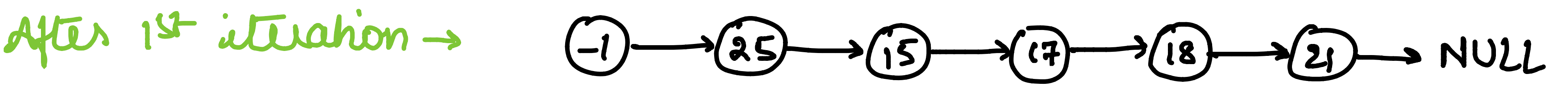
mp

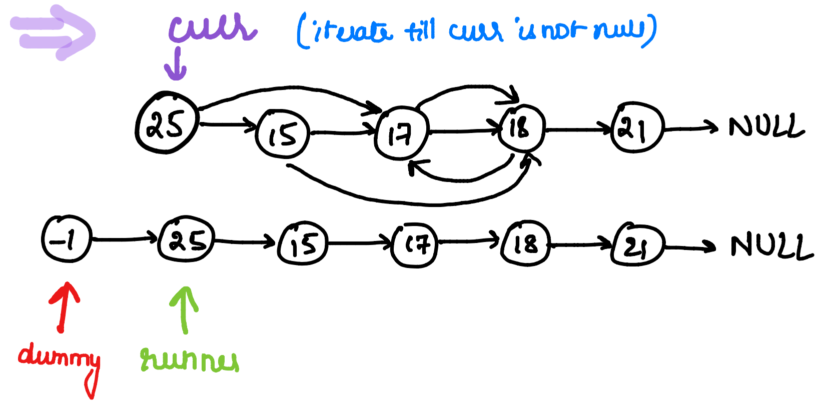
Old	New
25	25
15	15
17	17
18	18
21	21
NULL	NULL

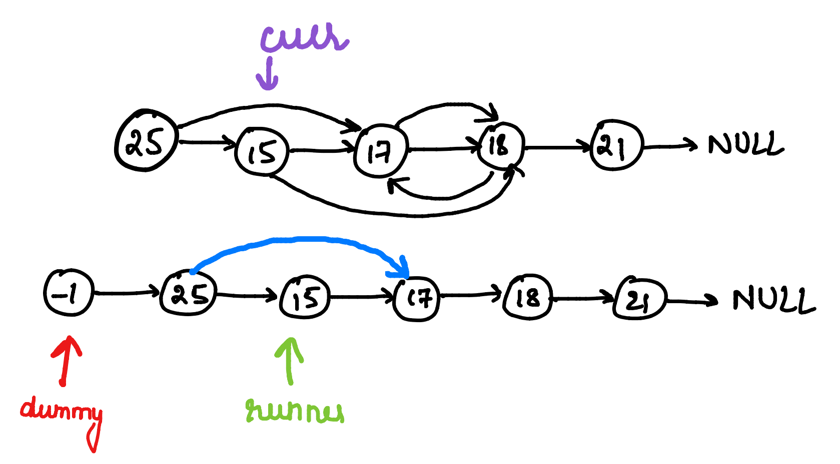
-> In 1st iteration create the list without random pointers & also maintain hashmap for mapping nodes pointed by random pointers

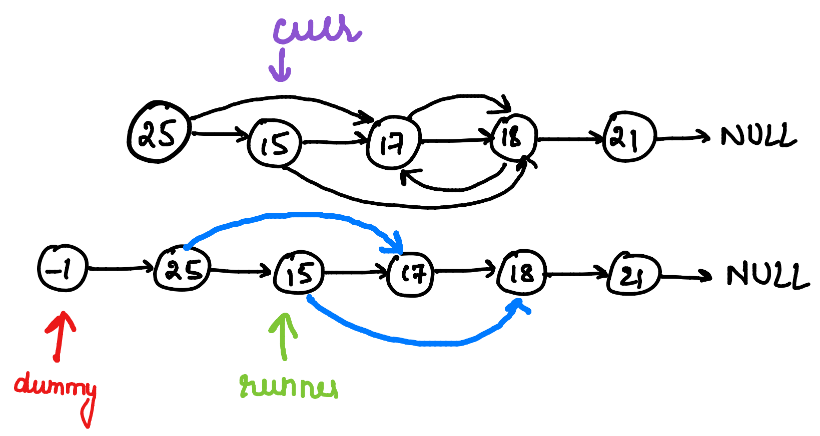
-> In 2nd iteration use map to link nodes pointed by random pointers

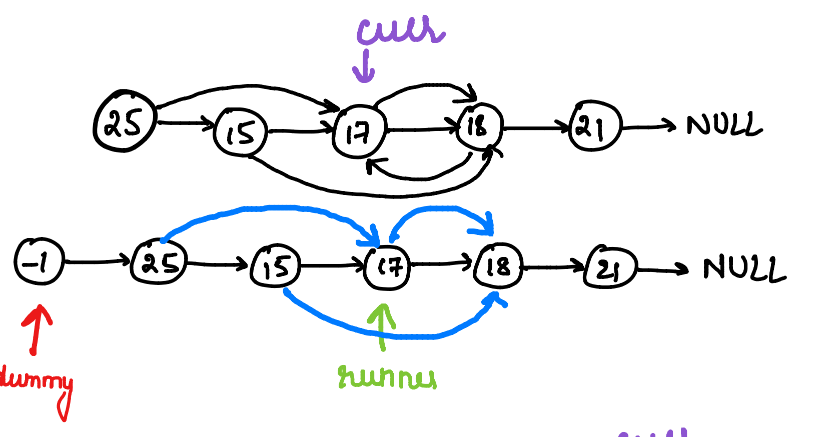
After 1st iteration →

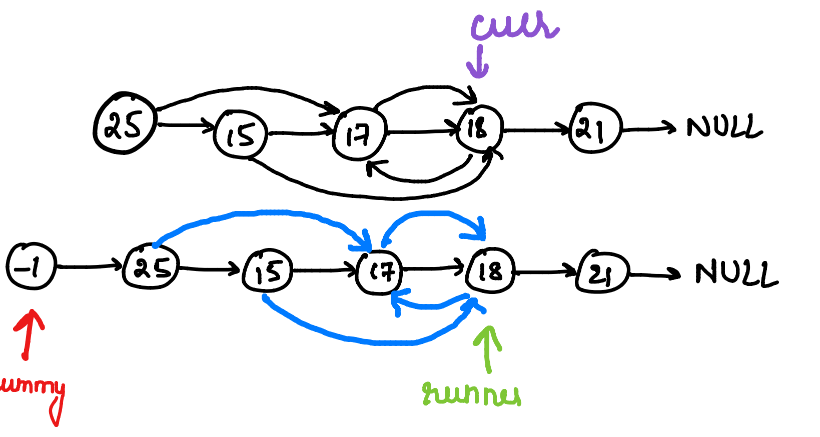


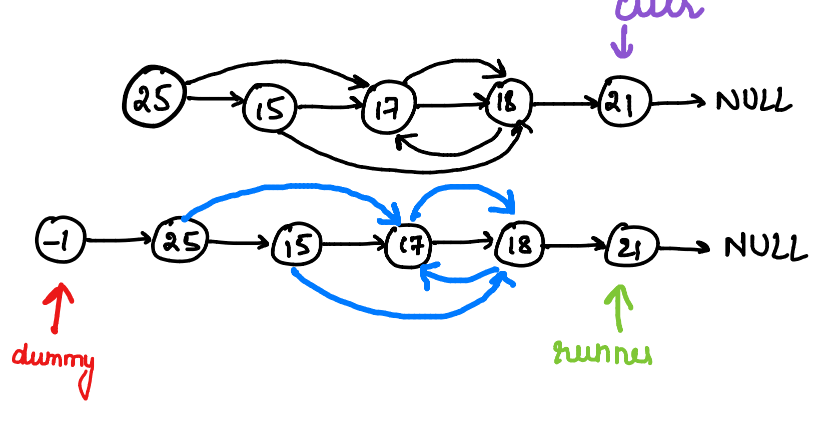


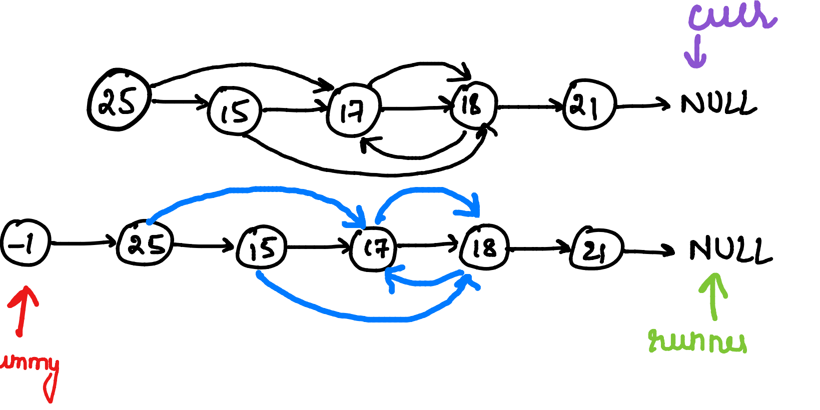


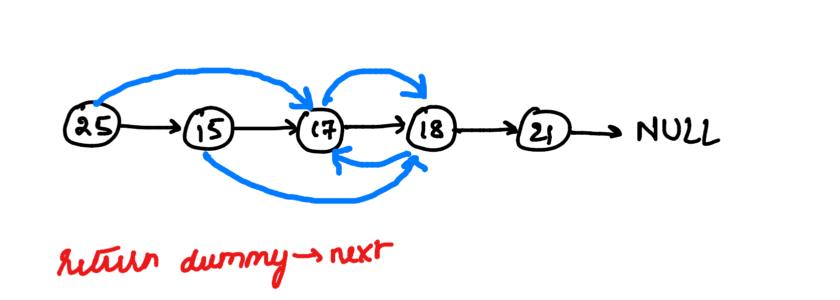












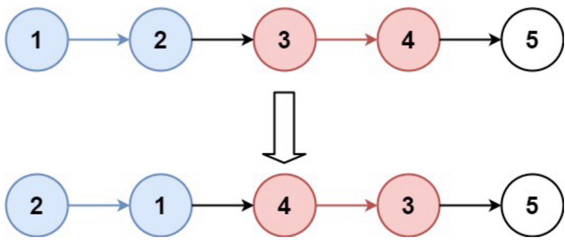
https://www.linkedin.com/in/karun-karthik

Code →

TC → O(n)
SC → O(n)

```
class Solution {
public:
    Node* copyRandomList(Node* head) {

        unordered_map<Node*, Node*> mp;
        Node *dummy = new Node(100001);
        Node *runner = dummy, *curr = head;

        // initial iteration
        while(curr != NULL){
            Node *newNode = new Node(curr->val);
            runner->next = newNode;
            mp[curr] = newNode;
            curr = curr->next;
            runner = runner->next;
        }

        // setting starting points in both lists
        curr = head;
        runner = dummy->next;

        // setting the random pointers
        while(curr != NULL){
            if(curr->random != NULL)
                runner->random = mp[curr->random];
            runner = runner->next;
            curr = curr->next;
        }

        return dummy->next;
    }
};
```

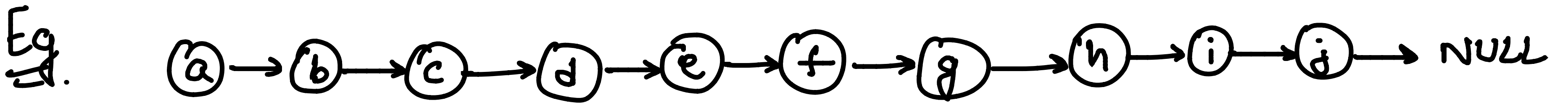
(16) Reverse Nodes in K-Group



ie K=2

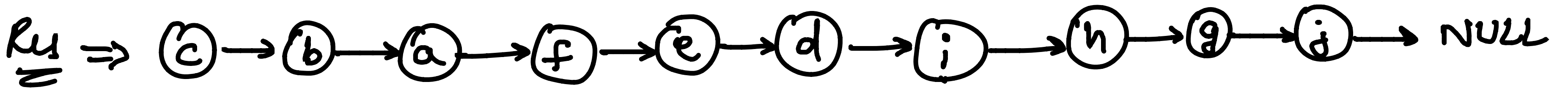
given a linkedlist & K, return a list with reversed nodes by K-groups.

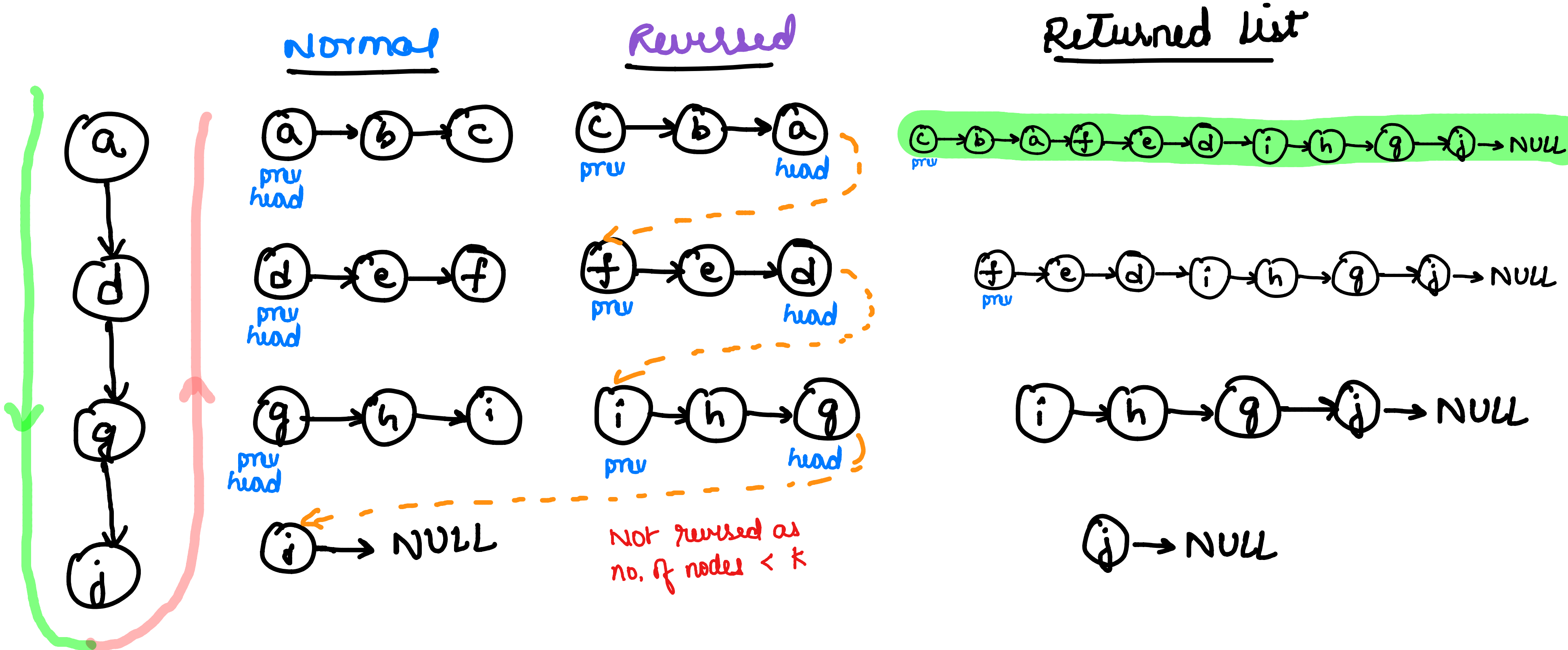
Eg.



k = 3

Res ⇒





* consider the case of (f)→(e)→(d) (prev, head), then tempList = (i)→(h)→(g)

linking would happen as head→next = tempList & this list would become tempList to (c)→(b)→(a)

code →

TC → O(n)
SC → O(1)

```
class Solution {
public:
    ListNode* reverseList(ListNode* head)
    {
        ListNode *prev = NULL, *curr = head, *temp;
        while (curr!=NULL)
        {
            temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }

    ListNode* reverseInGroups(ListNode* head, int k)
    {
        ListNode *curr = head;
        int currlen = 1;
        if(head == NULL) return head;
        while(curr->next!=NULL && currlen<k ){
            curr=curr->next;
            currlen+=1;
        }
        if(currlen<k)   return head;
        ListNode *tempNode = curr->next;
        curr->next = NULL;

        // start linking
        ListNode *tempList = reverseInGroups(tempNode,k);
        ListNode *prev = reverseList(head);
        head->next = tempList;
        return prev;
    }

    ListNode* reverseKGroup(ListNode* head, int k) {
        return reverseInGroups(head,k);
    }
};
```

(17) Design Linked List → Implementation of Doubly Linked List

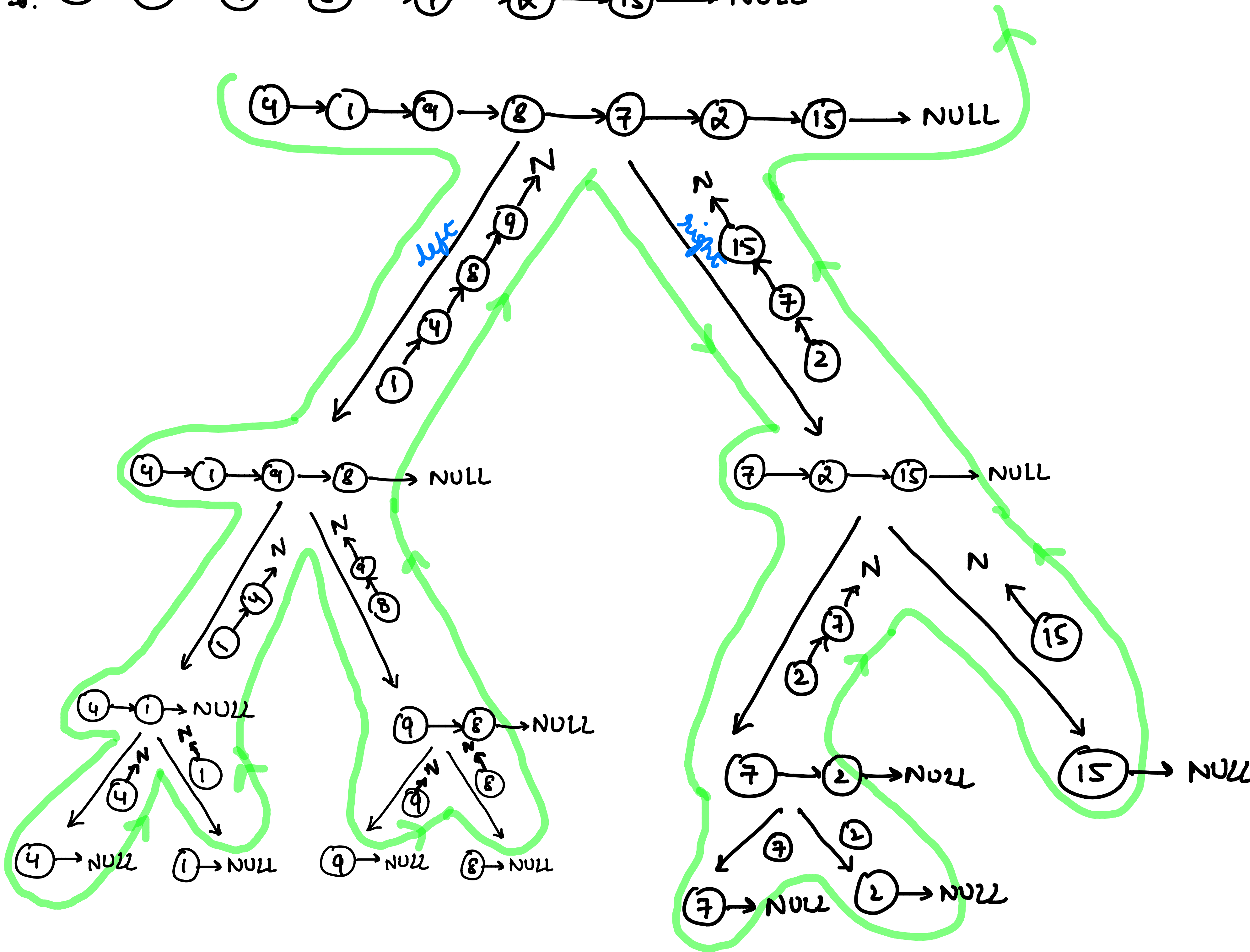
Code →

```
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 → 1 → 9 → 8 → 7 → 2 → 15 → NULL



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

left = 1 → 4 → 8 → 9 → NULL & right = 2 → 7 → 15 → NULL

so create dummy node & merge, i.e -1 → 1 → 2 → 4 → 7 → 8 → 9 → 15 → NULL

return dummy → next, 1 → 2 → 4 → 7 → 8 → 9 → 15 → NULL

* The same happens at every intermediate merge

code →

TC → O(m+n)

SC → O(n)

```cpp
class Solution {
public:
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1);
        ListNode *curr = dummy;
        while(l1 && l2){
            if(l1->val < l2->val){
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }
        if(l1) curr->next = l1;
        if(l2) curr->next = l2;

        return dummy->next;
    }

    ListNode* sortList(ListNode* head) {
        if(!head || !head->next) return head;

        ListNode *slow = head;
        ListNode *fast = head->next;
        while(fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        // dividing the lists into 2 parts
        fast = slow->next;
        slow->next = NULL;

        // sort & merge
        head = sortList(head);
        fast = sortList(fast)
        return merge(head, fast);
    }
};
```

Find the rest on
https://linktr.ee/KarunKarthik

Follow Karun Karthik For More Amazing Content !