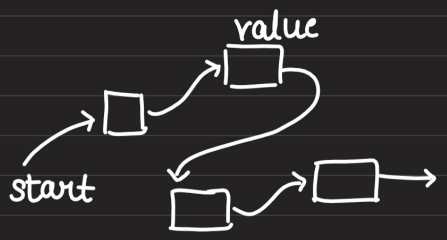
Linked list :-



Less workload
↓
more output

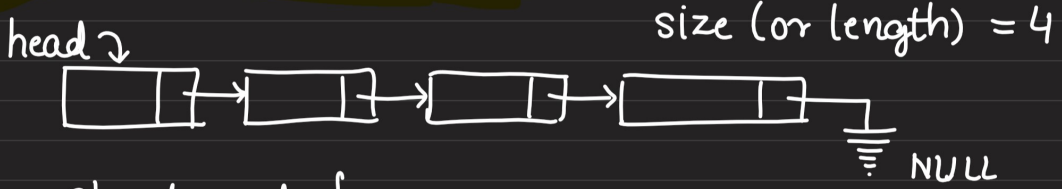
Different Types of Linked List

1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list
4. XOR Linked list [Misc]

&a → adress of a

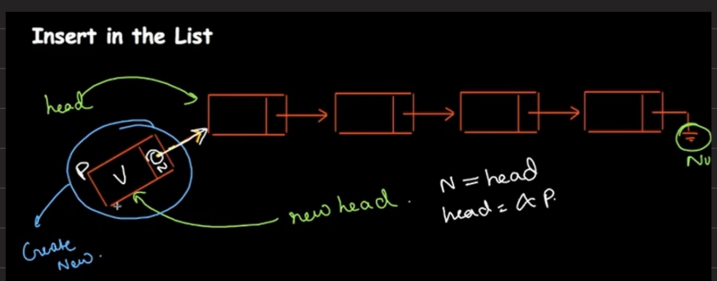
*a → value at adress a

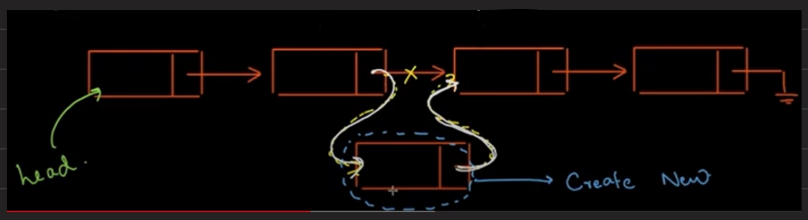
Singly linked list :



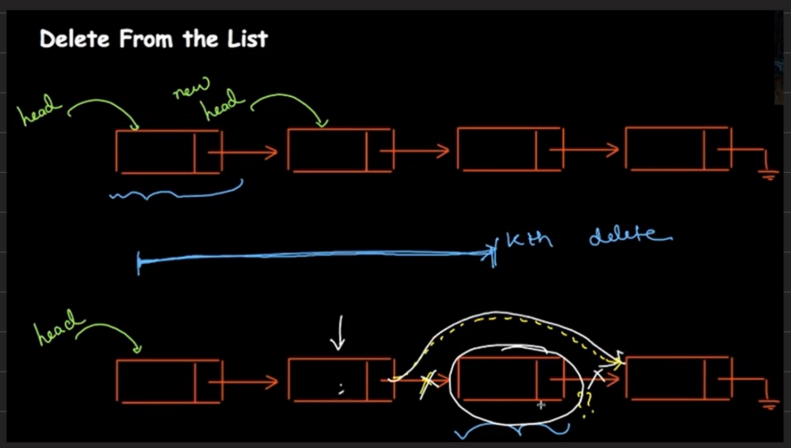

```
struct node {
    int data;
    node *next;
};

node *head;
```

Insert in the list :

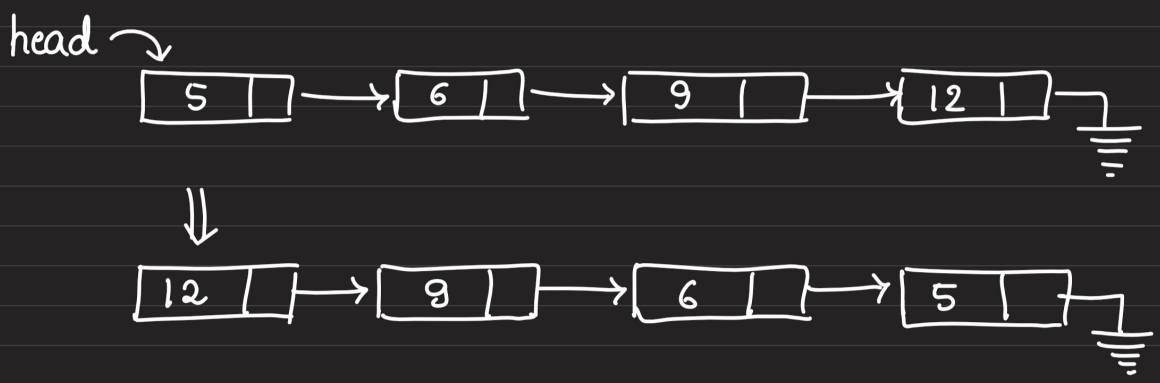




Deletion :



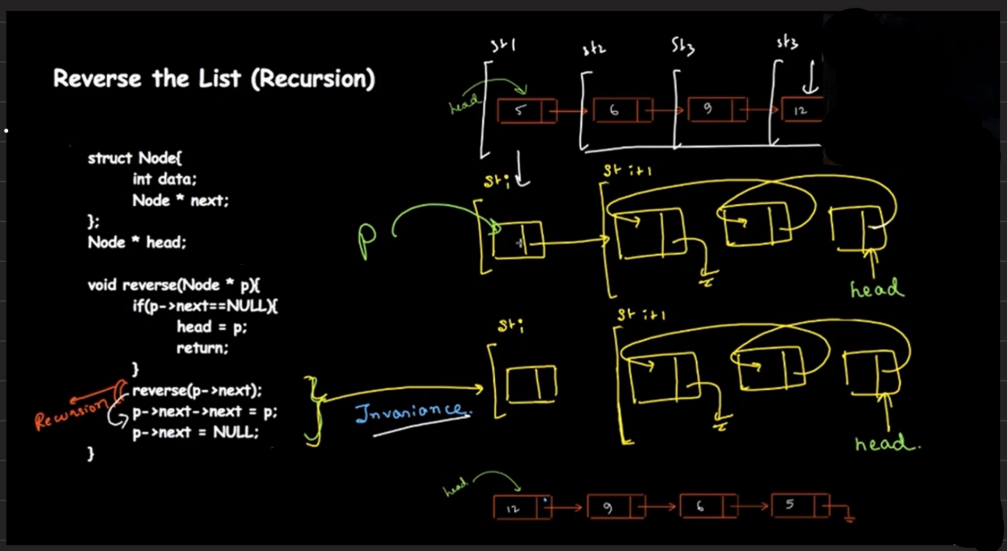
Reverse the list



Code :

```
void reverse (node *p) {
    if (p->next == NULL)        // Base case
    {
        head = p;
        return;
    }
```

```
reverse(p->next);
p->next->next = p;
p->next = NULL;
}
```



Using it like a stack

Do problems

Circular linked list II

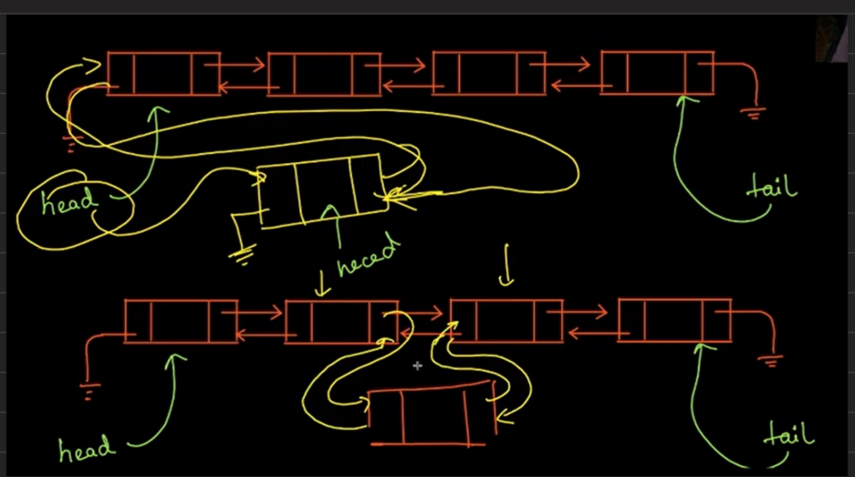
Steps to Solve :

1. Always visualize the action first
2. Make sure to check Nulls ! Also empty linked list in input !
3. To make coding easy, u can use the dummy node idea !!

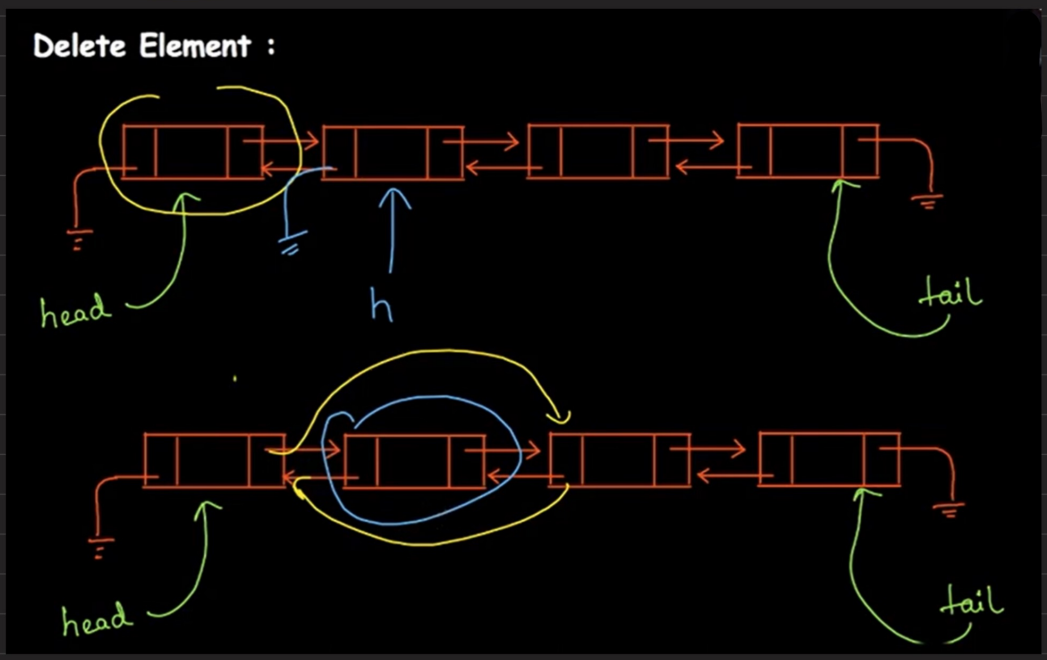
Double linked list :


```
struct node {
    int data;
    node* next;
    node *prev;
};
node* head;
```

Adding element :



Delete Element :



INSERTION IN SINGLE LINKED LIST

Create a link list of size N according to the given input literals. Each integer input is accompanied by an indicator which can either be 0 or 1. If it is 0, insert the integer in the beginning of the link list. If it is 1, insert the integer at the end of the link list.

Hint: When inserting at the end, make sure that you handle NULL explicitly.

```
class Solution{
  public:
    //Function to insert a node at the beginning of the linked list.
    Node *insertAtBegining(Node *head, int x) {
       Node *temp = new Node(x);
       temp->next = head;
       return temp;
    }


    //Function to insert a node at the end of the linked list.
    Node *insertAtEnd(Node *head, int x)  {
       Node *cur = head;
       Node *temp = new Node(x);
       if(head==NULL) return temp; //edge case
       while(cur->next!=NULL){
           cur = cur->next;
       }
       cur->next = new Node(x);
       return head;
    }
};
```

DELETING NODE JUST BY NODE ACCESS

There is a singly-linked list `head` and we want to delete a node `node` in it.

You are given the node to be deleted `node`. You will **not be given access** to the first node of `head`.

All the values of the linked list are **unique**, and it is guaranteed that the given node `node` is not the last node in the linked list.

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
        return;
    }
};
```

MIDDLE OF THE LL

Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

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

REVERSE - LL

Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

RECURSIVE

```
ListNode* reverseList(ListNode* head) {
    if(head==NULL) return head; //edge case 1
    if(head->next==NULL) return head; //edge case 2

    //Storing the value of the first node
    ListNode* cur = new ListNode(head->val);

     //using recursion to reverse the rest of the list
    ListNode* ans = reverseList(head->next);

    //Inserting cur node at the end
    ListNode* temp = ans;
    while(temp->next!=NULL) temp = temp->next;
    temp->next = cur;
    return ans;
}
```

ITERATIVE

```
ListNode* reverseList(ListNode* head) {
    if(head==NULL) return head; //edge case 1
    if(head->next==NULL) return head; //edge case 2

    ListNode* prev = NULL;
    ListNode* cur = head;
    while(head){
        head = head->next;
        cur->next = prev;
        prev = cur;
        cur = head;
    }
    //here head/cur will become null and prev will give the reversed LL
    return prev;
}
```

DELETING NODE IN DOUBLY LINKED LIST

Given a doubly **Linked list** and a position. The task is to **delete** a node from a given position (position starts from 1) in a doubly linked list and return the head of the doubly Linked list.

Input: LinkedList = 1 <--> 3 <--> 4, x = 3

Output: 1 3

Explanation: After deleting the node at position 3 (position starts from 1), the linked list will be now as 1->3.

Input: LinkedList = 1 <--> 5 <--> 2 <--> 9, x = 1

Output: 5 2 9

```
class Solution {
  public:
    Node* deleteNode(Node* head, int x)
    {
        x--;
        int pos = 1;
        Node *cur = head;
        //Base Case
        if(x==0) {
            head = cur->next;
            if(head) head->prev = NULL;
            return head;
        }

        //Iterating to the element just before the element to be deleted
        while(pos<x){
            cur = cur->next;
            pos++;
        }
        //First change the previous then only update the cur->next
        if(cur->next->next) cur->next->next->prev = cur;
        cur->next = cur->next->next;

        return head;
    }
};
```

REVERSING DOUBLY LL

Given a **doubly linked list** of **n** elements. Your task is to **reverse** the doubly linked list **in-place**.

RECURSIVE

```
Node* reverseDLL(Node * head) //O(n2), T(n) = T(n-1) + O(n)
{
    if(head==NULL) return head;
    if(head->next==NULL){
        head->prev = NULL; //for last node
        return head;
    }

    Node *init = new Node(head->data);
    Node *ans = reverseDLL(head->next);
    Node *temp = ans;
    while(temp->next!=NULL) temp = temp->next;
    init->prev = temp;
    temp->next = init;
    return ans;
}
```

ITERARTIVE O(N)

```
Node* reverseDLL(Node * head) //O(n)
{
    if(head==NULL || head->next==NULL) return head;
    Node *l = NULL;
    Node *r = head;
    while(head){
        head = head->next;
        r->next = l;
        r->prev = head;
        if(l) l->prev = r;
        l = r;
        r = head;
    }
    return l;
}
```

Given a sorted doubly linked list of positive distinct elements, the task is to find pairs in a doubly-linked list whose sum is equal to given value **target**.

2-SUM

```
vector<pair<int, int>> findPairsWithGivenSum(Node *head, int target)
{
    vector<pair<int,int>> ans;
    if(head->next==NULL) return ans;
    Node *l = head, *r = head;
    while(r->next){
        r = r->next;
    }
    while(l && r && l!=r)
    {
        int lval = l->data, rval = r->data;
        if(lval+rval==target)
        {
            ans.push_back({lval,rval});
            l = l->next;
            if(l==r) break;
            r = r->prev;
        }
        else if(lval+rval<target){
            l = l->next;
        }
        else{
            r = r->prev;
        }
    }
    return ans;
}
```

Given a doubly linked list of **n** nodes sorted by values, the task is to remove duplicate nodes present in the linked list.

```
Node * removeDuplicates(struct Node *head)
{
    if(head->next==NULL) return head;
    Node *h = head;
    int prev = head->data;
    head = head->next;
    while(head)
    {
        int cur = head->data;
        if(cur==prev){ //remove this node
            if(head->next) head->next->prev = head->prev;
            head->prev->next = head->next;
        }
        prev = cur;
        head = head->next;
    }
    return h;
}
```

You are given the **head_ref** of a doubly Linked List and a **Key**. Your task is to **delete all occurrences** of the given key if it is present and return the new DLL.

2<->2<->10<->8<->4<->2<->5<->2

2

Output:

10<->8<->4<->5

Explanation:

All Occurences of 2 have been deleted.

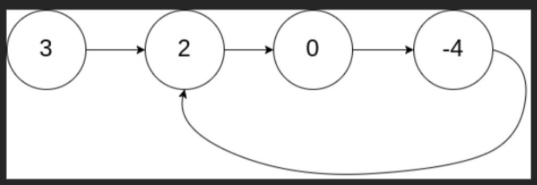
```
void deleteAllOccurOfX(struct Node** head_ref, int x) {
    Node *head = (*head_ref);
    Node *h = head;
    while(h)
    {
        if(h->data==x){
            h = h->next;
            h->prev = NULL;
        }else{
            break;
        }
    }
    Node *temp = h;
    *head_ref = h; //Note this is the correct way to update
    if(temp==NULL) return;

    temp = temp->next;
    while(temp){
        if(temp->data==x){
            if(temp->next) temp->next->prev = temp->prev;
            temp->prev->next = temp->next;
        }
        temp = temp->next;
    }
}
```

SINGLY LINKED LIST STANDARD PROBLEMS

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter**.

Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.



Input: head = [3,2,0,-4], pos = 1
Output: true

USING MAP

```cpp
bool hasCycle(ListNode *head) {
    map<ListNode*,int> mp;
    while(head!=NULL){
        mp[head]++;
        if(mp[head]>1) return 1;
        head = head->next;
    }
    return 0;
}
```

USING FAST SLOW POINTER

```cpp
bool hasCycle(ListNode *head) {
    ListNode *s = head;
    ListNode *f = head;
    while(f && f->next){
        s = s->next;
        f = f->next->next;
        if(s==f) return 1;
    }
    return 0;
}
```

Given the `head` of a linked list, return *the node where the cycle begins. If there is no cycle, return* `null`.

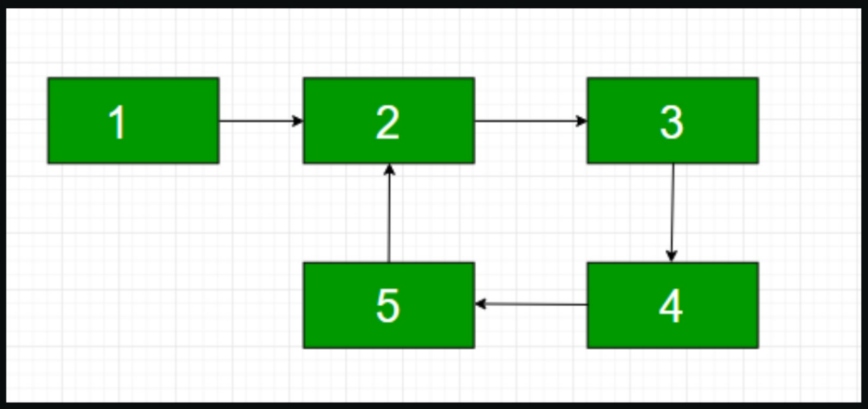
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to (**0-indexed**). It is `-1` if there is no cycle. **Note that** `pos` **is not passed as a parameter**.

Do not modify the linked list.

```
ListNode *detectCycle(ListNode *head) {
    map<ListNode*,int> mp;
    while(head){
        mp[head]++;
        if(mp[head]==2) return head;
        head = head->next;
    }
    return NULL;
}
```

```
ListNode *detectCycle(ListNode *head) {
    ListNode *s = head, *f = head;
    ListNode *col = NULL;
    while(f && f->next){
        s = s->next;
        f = f->next->next;
        if(s==f){
            col = f; break;
        }
    }
    if(col){
        while(head!=s){
            head = head->next;
            s = s->next;
        }
        return head;
    }
    return col;
}
```

Given a linked list of size **N**. The task is to complete the function **countNodesinLoop()** that checks whether a given Linked List contains a **loop or not** and if the **loop** is present then **return the count of nodes** in a loop or else **return 0. C** is the position of the node to which the last node is connected. If it is 0 then no loop.




```
int countNodesinLoop(struct Node *head)
{
    Node *s = head, *f = head;
    Node *col = NULL;
    int len = 0;
    while(f && f->next){
        s = s->next;
        f = f->next->next;
        if(s==f){
            col = f; break;
        }
    }
    if(col){
        while(head!=s){
            head = head->next;
            s = s->next;
        }
        col = s; //first finding the starting point of the loop
        int ans = 1;
        while(s->next!=col){ //iterating till we reach initial point again
            ans++;
            s = s->next;
        }
        return ans;
    }
    else{
        return 0;
    }
}
```

Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise.*

```cpp
bool isPalindrome(ListNode* head) {
    vector<int> v;
    while(head){
        v.push_back(head->val);
        head = head->next;
    }
    for(int i=0;i<v.size()/2;i++){
        if(v[i]!=v[v.size()-i-1]) return false;
    }
    return true;
}
```

INTERVIEW APPROCH

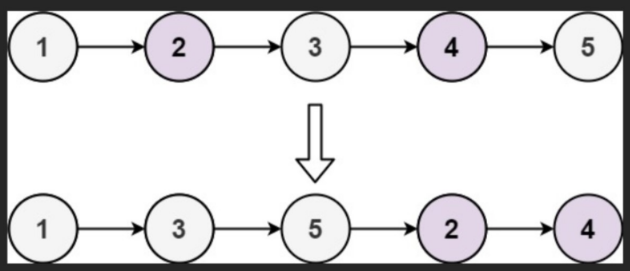
1. CREATE A NEW COPY (DONT USE THE SAME LINKED LIST HEAD, CREATE A NEW ONE)
2. REVERSE THE GIVEN LINKED LIST
3. COMPARE THE COPY OF LL WITH THE REVERSE

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.



Input: `head = [1,2,3,4,5]`
Output: `[1,3,5,2,4]`

You are given the `head` of a linked list. **Delete** the **middle node**, and return *the* `head` *of the modified linked list.*

The **middle node** of a linked list of size `n` is the `⌊n / 2⌋th` node from the **start** using **0-based indexing**, where `⌊x⌋` denotes the largest integer less than or equal to `x`.

- For `n` = `1`, `2`, `3`, `4`, and `5`, the middle nodes are `0`, `1`, `1`, `2`, and `2`, respectively.

```
ListNode* deleteMiddle(ListNode* head)
{
    //Handling for one or 2 nodes
    if(head->next==NULL) return NULL;
    if(head->next->next==NULL){
        head->next = NULL;
        return head;
    }
    //Finding middle element
    ListNode *s = head, *f = head;
    while(f && f->next){
        s = s->next;
        f = f->next->next;
    }
    //Deleting when only node ptr is available
    s->val = s->next->val;
    s->next = s->next->next;
    return head;
}
```

Given the `head` of a linked list, return *the list after sorting it in **ascending order***.

Merge Sort on Linked List

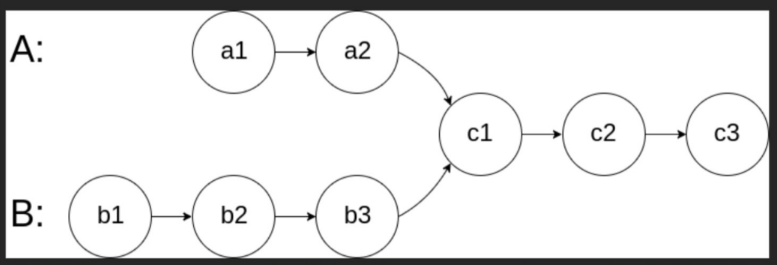
```
class Solution {
public:
    ListNode* merge(ListNode *l1, ListNode *l2)
    {
        if(l1==NULL) return l2;
        if(l2==NULL) return l1;
        ListNode *ans, *temp;
        ans = new ListNode(min(l1->val,l2->val));
        if(l1->val<=l2->val) l1 = l1->next;
        else l2 = l2->next;

        temp = ans; //to store head of merged linked list

        while(l1 && l2){
            if(l1->val <= l2->val){
                ans->next = new ListNode(l1->val);
                l1 = l1->next;
            }else{
                ans->next = new ListNode(l2->val);
                l2 = l2->next;
            }
            ans = ans->next;
        }
        while(l1){
            ans->next = new ListNode(l1->val);
            l1 = l1->next;
            ans = ans->next;
        }
        while(l2){
            ans->next = new ListNode(l2->val);
            l2 = l2->next;
            ans = ans->next;
        }
        return temp;
    }
    ListNode* sortList(ListNode* head)
    {
        if(head==NULL || head->next==NULL) return head;
        ListNode *prev = NULL, *s = head, *f = head;
        while(f && f->next){
            prev = s;
            s = s->next;
            f = f->next->next;
        }
        prev->next = NULL;
        //head is now first half of the linked list
        //s is the second half
        ListNode *l = sortList(head);
        ListNode *r = sortList(s);
        ListNode *ans = merge(l, r);
        return ans;
    }
};
```

Given a linked list of **N** nodes where nodes can contain values **0s**, **1s,** and **2s** only. The task is to segregate **0s**, **1s,** and **2s** linked list such that all zeros segregate to head side, 2s at the end of the linked list, and 1s in the mid of 0s and 2s.

```
Node* segregate(Node *head) {
    int arr[3] = {0};
    while(head){
        arr[head->data]++;
        head = head->next;
    }
    Node *ans = NULL;
    for(int i=0;i<3;i++)
    {
        while(arr[i]--)
        {
            Node *temp = new Node(i);
            if(ans==NULL){
                ans = temp;
                head = ans;
            }else{
                ans->next = temp;
                ans = ans->next;
            }
        }
    }
    return head;
}
```

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

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



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

```cpp
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    map<ListNode*,int> mp;
    while(headA){
        mp[headA]++;
        headA = headA->next;
    }
    while(headB){
        mp[headB]++;
        if(mp[headB]==2) return headB;
        headB = headB->next;
    }
    return NULL;
}
```

```cpp
int len(ListNode *headA){
    ListNode *temp = headA;
    int ans = 0;
    while(temp){
        ans++;
        temp = temp->next;
    }
    return ans;
}
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    int l1 = len(headA), l2 = len(headB);
    if(l1>l2) swap(headA,headB);
    int diff = abs(l2-l1);
    while(diff--){
        headB = headB->next;
    }
    while(headA){
        if(headA==headB) return headA;
        headA = headA->next;
        headB = headB->next;
    }
    return NULL;
}
```

TC - O(n)
SC - O(1)

ADD TWO NUMBERS USING LINKED LIST

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

```
int len(ListNode *l){
    ListNode *temp = l; int ans = 0;
    while(temp){
        temp = temp->next;
        ans++;
    }return ans;
}
```

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    int n = len(l1), m = len(l2);
    if(n<m) {
        swap(l1,l2);
        swap(n,m);
    }
    ListNode *head = l1, *prev = NULL;
    int carry = 0;
    int sum;
    while(l2){
        sum = (l1->val + l2->val + carry)%10;
        carry = (l1->val + l2->val + carry)/10;
        l1->val = sum;
        prev = l1;
        l1 = l1->next;
        l2 = l2->next;
    }
    while(l1){
        sum = (l1->val + carry)%10;
        carry = (l1->val + carry)/10;
        prev = l1;
        l1->val = sum;
        l1 = l1->next;
    }
    if(carry){
        ListNode *temp = new ListNode(carry);
        prev->next = temp;
    }
    return head;
}
```

Given the `head` of a linked list, rotate the list to the right by `k` places.

Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]

- The number of nodes in the list is in the range `[0, 500]`.
- `-100 <= Node.val <= 100`
- `0 <= k <= 2 * 10^9`

```
int len(ListNode *l){
    ListNode *temp = l; int ans = 0;
    while(temp){
        ans++; temp = temp->next;
    }return ans;
}
```

```
ListNode* rotateRight(ListNode* head, int k) {
    if(head==NULL || head->next==NULL) return head;
    int l = len(head); k%=l;
    if(k==0) return head;
    ListNode *t1 = head, *t2 = head;
    while(t2->next){
        t2 = t2->next;
    }
    t2->next = t1; //loop is created
    int pos = l - k - 1;
    while(pos--){
        t1 = t1->next;
    }
    head = t1->next;
    t1->next = NULL;
    return head;
}
```

Given the `head` of a linked list, reverse the nodes of the list `k` at a time, and return *the modified list*.

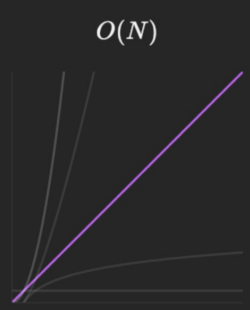
`k` is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of `k` then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

Input: `head = [1,2,3,4,5], k = 2`
Output: `[2,1,4,3,5]`

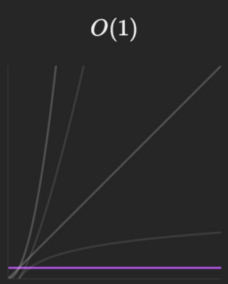
```
int len(ListNode *head){
    ListNode *temp = head; int ans = 0;
    while(temp){
        ans++; temp = temp->next;
    }return ans;
}
ListNode* reverseKGroup(ListNode* head, int k) {
    if(k==1) return head;
    int n = len(head);
    if(k>n) return head;

    ListNode *l = NULL, *r = head, *init = head;
    for(int i=0;i<k;i++){
        head = head->next;
        r->next = l;
        l = r;
        r = head;
    }
    if(k==n) return l;
    ListNode *temp = reverseKGroup(head,k);
    init->next = temp;
    return l;
}
```

Time Complexity



$$T(n) = T(n-k) + O(k)$$

Space Complexity



Note that we are passing the pointers, so, no extra space is used in the stack

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

```
ListNode* oddEvenList(ListNode* head)
{
    if(head==NULL || head->next==NULL) return head; //Size 0 or 1
    ListNode *odd, *even,*h_odd, *h_even;
    odd = new ListNode(head->val);
    even = new ListNode(head->next->val);
    h_odd = odd;
    h_even = even;

    head = head->next->next;
    int cnt = 1;
    while(head){
        if(cnt%2){
            odd->next = new ListNode(head->val);
            odd = odd->next;
        }else{
            even->next = new ListNode(head->val);
            even = even->next;
        }
        cnt++;
        head = head->next;
    }
    odd->next = h_even;
    return h_odd;

}
```

TC - O(N)
SC - O(N)

```
ListNode* oddEvenList(ListNode* head)
{
    if(head==NULL || head->next==NULL) return head; //Size 0 or 1
    ListNode *odd = head, *even = head->next,*h_odd, *h_even;
    h_odd = odd;
    h_even = even;

    head = head->next->next;
    int cnt = 1;
    while(head){
        if(cnt%2){
            odd->next = head;
            odd = head;
        }else{
            even->next = head;
            even = head;
        }
        cnt++;
        head = head->next;
    }
    even->next = NULL;
    odd->next = h_even;
    return h_odd;
}
```

TC - O(N)
SC - O(1)

FLATENING A LINKED LIST

Given a Linked List of size n, where every node represents a sub-linked-list and contains two pointers:

(i) a **next** pointer to the next node,

(ii) a **bottom** pointer to a linked list where this node is head.

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

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

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

Input:

```
5 -> 10 -> 19 -> 28
|     |     |     |
7     20    22    35
|           |     |
8           50    40
|                 |
30                45
```

Output: 5-> 7-> 8- > 10 -> 19-> 20-> 22-> 28-> 30-> 35-> 40-> 45-> 50.

Explanation: The resultant linked lists has every node in a single level.(**Note:** | represents the bottom pointer.)

```
Node *flatten(Node *root)
{
   if(root==NULL || root->next==NULL) return root;
   vector<Node*> v;
   Node *h = root;
   while(root){
       v.push_back(root);
       root = root->next;
   }
   int f = 1;
   while(f)
   {
       Node *t;
       int val = 10000, ind = -1;
       for(int j=1;j<v.size();j++)
       {
           if(v[j] && v[j]->data < val)
           {
               t = v[j];
               val = v[j]->data;
               ind = j;
           }
       }
       if(v[0]->bottom)
       {
           if(v[0]->bottom->data > val)
           {
               v[ind] = v[ind]->bottom;
               t->bottom = v[0]->bottom;
               v[0]->bottom = t;
           }
           v[0] = v[0]->bottom;
       }
       else
       {
           v[ind] = v[ind]->bottom;
           v[0]->bottom = t;
           t->bottom = NULL;
           v[0] = v[0]->bottom;
       }
       f = 0;
       for(int i=1;i<v.size();i++){
           if(v[i]!=NULL) f = 1;
       }
   }
   return h;
}
```

pushing all the roots to a vector

finding the node with minimum value

→ if bottom element is present

→ if bottom element value is larger then insert t after v[0] and then update v[ind] = v[ind] → bottom

} → move v[0] to next element

if v[0] is last element then insert it after it, also update v[ind] = v[ind] → bottom

if all the other list except v[0] is traversed, break the loop.

COPY LL WITH NEXT AND RANDOM POINTER

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a **deep copy** of the list. The deep copy should consist of exactly `n` **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

```
Node* copyRandomList(Node* head) {
    Node *h = NULL, *temp, *orig = head; //h is the cloned head
    if(head==NULL) return h;

    h = new Node(head->val);
    temp = h;
    map<Node*,Node*> mp; //stores the corresponding node of the head in h
    mp[head] = temp;

    head = head->next;
    while(head){ //creating next links in the list along with updating the map
        temp->next = new Node(head->val);
        temp = temp->next;
        mp[head] = temp;
        head = head->next;
    }

    Node *cpy = h; //updating the randoms in the copy of head
    while(orig){
        if(orig->random==NULL) cpy->random = NULL;
        else{
            cpy->random = mp[orig->random];
        }
        orig = orig->next;
        cpy = cpy->next;
    }
    return h;
}
```

TC - O(NlogN)
SC - O(N)

How to do it without using the hash-map?