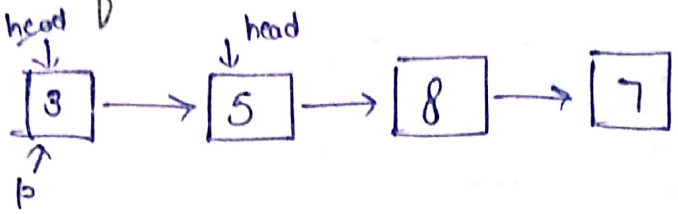
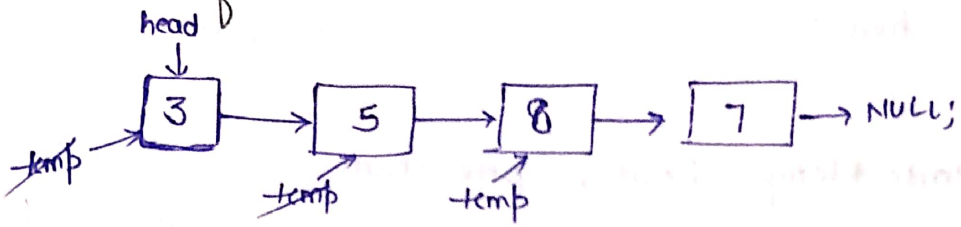
⇒ Deletion of head in LL :-



Code :-

```
Node *p = head;
head = head → next;
free(p);
return head;
```

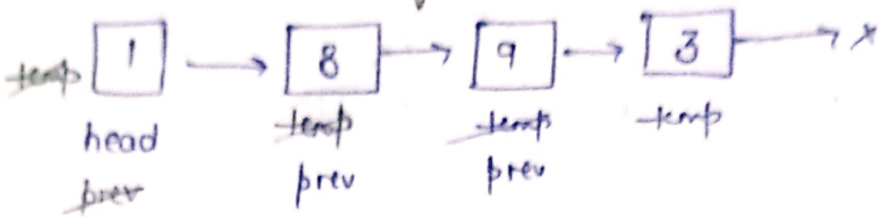
⇒ Deletion of ~~head~~ Tail in LL :-



Code :-

```
if(head == NULL)  return NULL;
if (head → next == NULL) {
        free(head);
        return NULL;
    }
else {
        Node *temp = head;
        while (temp → next → next != NULL) {
                temp = temp → next;
            }
        free (temp → next);
        temp → next = NULL;
        return head;
    }
```

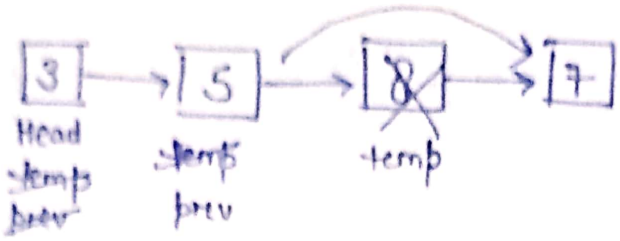
⇒ Delete Kth element of LL :-



⇒ Code :-

```
if (head == NULL)  return NULL;
if (K==1) {
    Node * temp = head;
    head = head -> next;
    free (temp);
    return head;
}
cnt=0  Node *temp = head ,  prev = null
While (temp != NULL) {
    cnt ++;
    if (cnt == K) {
        prev -> next = prev -> next -> next;
        free (temp);
        break;
    }
    prev = temp;
    temp = temp -> next;
}
return head;
```

Delete the node contains value x :-

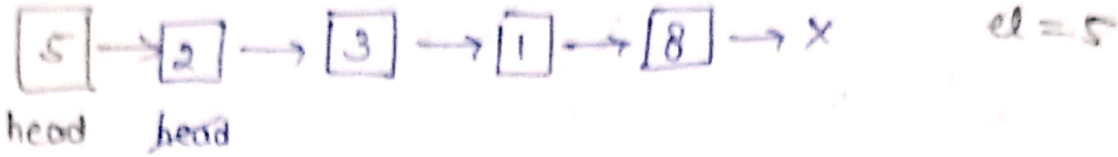


Code :-

```
if (head == NULL)  return NULL;
Node * temp = head;
if (head -> data == val) {
        head = head -> next;
        free (temp);
        return head;
}
else {
     Node * prev = NULL;
     while (temp != NULL) {
          if (temp -> data == val) {
               prev -> next = prev -> next -> next;
               free (temp);
               return head;
          }
          prev = temp;
          temp = temp -> next;
}
```

tum head;

→ Insertion at head :-



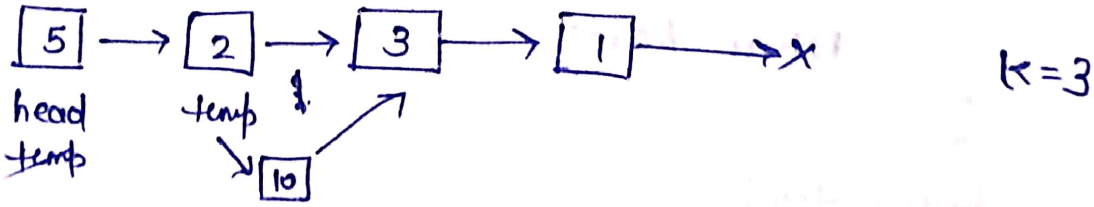
Code :-

```
Node *newNode = new Node (val, head);
return newNode;
```

→ Insertion at last :-

```
Node *newNode = new Node (val);
if (head == NULL)  return newNode;
Node * temp = head;
While(temp->next != NULL) {
    temp = temp->next;
}
temp->next = newNode;
return head;
```

→ Insert at Kth position :-

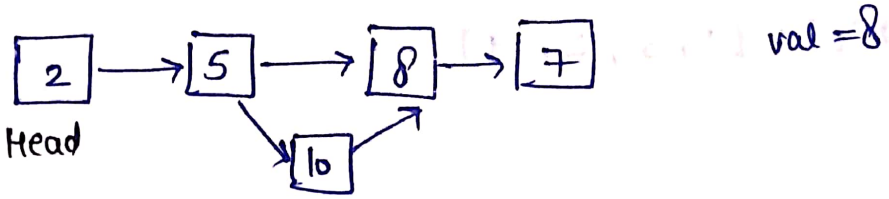


Code :-

```
Node *newNode = new Node (val);
If(head == NULL) {
    if(k==1)  return newNode;
```

```
if (k == 1) {
    newNode -> next = head;
    return newNode;
}
Node * temp = head;
int cnt = 0;
While (temp != NULL) {
    cnt ++;
    if(cnt == K-1) {
        newNode -> next = temp -> next;
        temp -> next = newNode;
    }
    temp = temp -> next;
}
tum .head;
```

Insert element before value x :-

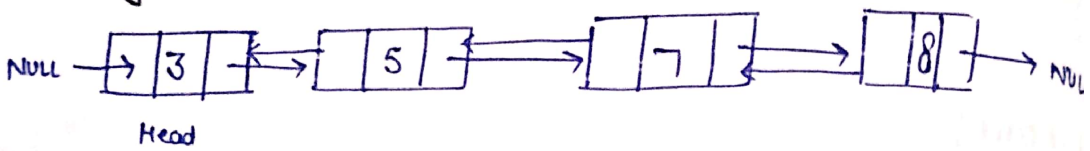


Code:-

```
Node * newNode = new Node (val);
if (head == NULL)  return NULL;
if (head -> data == k) {
        newNode -> next = head;
        return newNode;
}
```

```
Node * temp = head;
while (temp->next != NULL) {
    if (temp->next->data == k) {
        newNode->next = temp->next;
        temp->next = newNode;
        break;
    }
    temp = temp->next;
}
return head;
```

Doubly Linked List :-



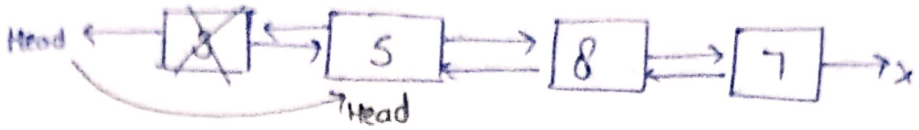
⇒ Convert array into DLL :-

arr[] = [1, 3, 2, 4]

Code :-

```
Node * head = new Node (arr[0])
Node * prev = head;
for (i=1 → n-1) {
    Node * temp = new Node (arr[i], nullptr, prev);
    prev->next = temp;
    prev = temp;        → temp->back = prev;
}
return head;
```

Delete the head of DLL :-



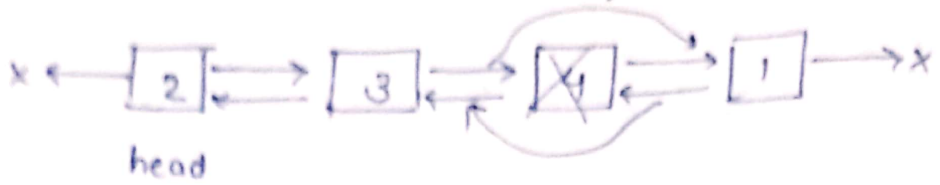
Code :-

```
if (head == NULL)  return head;
Node * temp = head;
head = head → next;
head → back = NULL;
temp → next = NULL;
free (temp);
return head;
```

Delete the tail of DLL :-

```
if (head == NULL)  return NULL;
Node * temp = head;
if (temp → next == NULL  &&  temp → back == NULL) {
    free (temp);
    return NULL;
}
while (temp → next != NULL) {
        temp = temp → next;
    }
Node *prev = temp → back;
prev → next = NULL;
temp → next = NULL;
free (temp);
return head;
```

Delete the k^{th} element of DLL :-



Code :-

```
cnt = 0;
While (temp != null) {
    cnt ++;
    if (cnt == k) break;
    temp = temp -> next;
}
prev = temp -> back;
front = temp -> next;
if (prev == NULL && front == NULL) {
    delete temp;
    return NULL;
}
else if (prev == NULL) {
    deleteHead (head);
    return head;
}
else if (front == NULL) {
    deleteTail (head);
}
else {
    prev -> next = front;
    front -> back = prev;
    temp -> next = NULL;
    temp -> back = NULL;
    free (temp);
}
return head;
```

Delete the Node of the DLL :- (Node != head)

Code :-

```
Node * prev = node -> back;
Node * front = node -> next;
if (front != NULL) {
        front -> back = prev;
    }
prev -> next = front;
```

Insert node before Head in DLL :-

code :-

```
Node * newHead = new Node (val, head, nullptr);
head -> back = newHead;
return newHead;
```

Insert before tail in DLL :-

```
Node * tail = head;
if (head -> next == NULL) {
        return insertBeforeHead (head, val);
 }
While (tail -> next != NULL) {
     tail = tail -> next;
}
Node * prev = tail -> back;
Node * temp = new Node (val, tail, prev);
prev -> next = temp;
tail -> back = temp;
return head;
```

Insert before kth node in DLL :-

Code :-

```
if(k==1)  return insertBeforeHead(head, val);
int cnt=0;
Node *temp=head;
while(temp!=NULL){
    cnt++;
    if(cnt==k){
        break;
    }
    temp=temp->next;
}
Node *prev=temp->back;
Node *newNode = new Node(val, temp, prev);
prev->next=newNode;
temp->back=newNode;
return head;
```

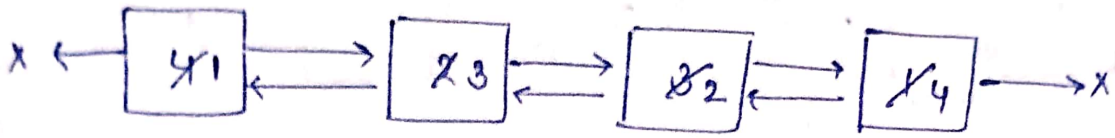
Insert before value in DLL :-

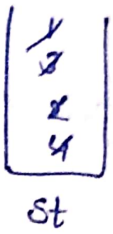
Code :-

```
Node *prev= node->back;
Node *newNode= new Node(val, node, prev);
prev->next = newNode;
node->back = newNode;
return head;
```

Reverse a DLL :-

→ Brute :-





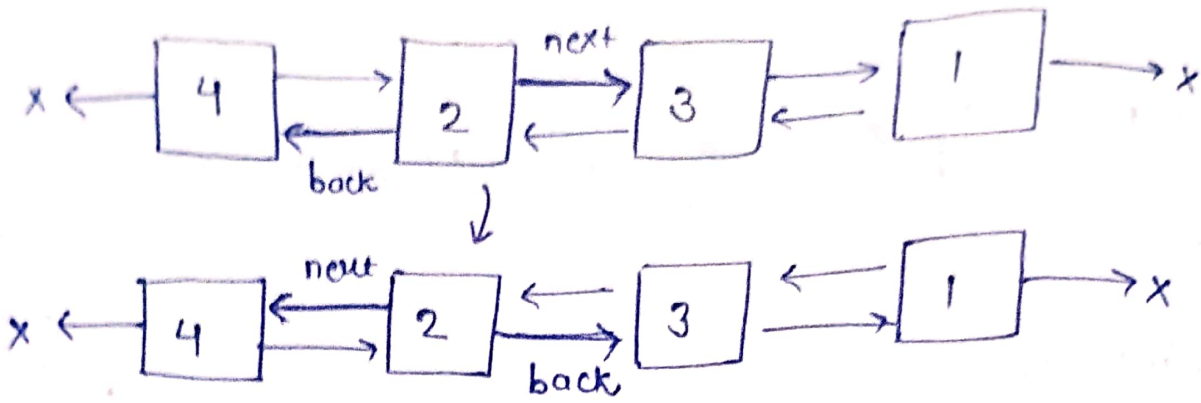
Code :-

```
stack<int> st;
Node * temp = head;
while (temp != NULL) {
    st.push(temp->data);      ⟶ O(n)
    temp = temp->next;
}
temp = head;
while (temp != NULL) {
    temp->data = st.top();
    st.pop();                 ⟶ O(n)
    temp = temp->next;
}
```

TC :- O(2n)

SC :- O(n)

→ Optimal :- (In-place by links)



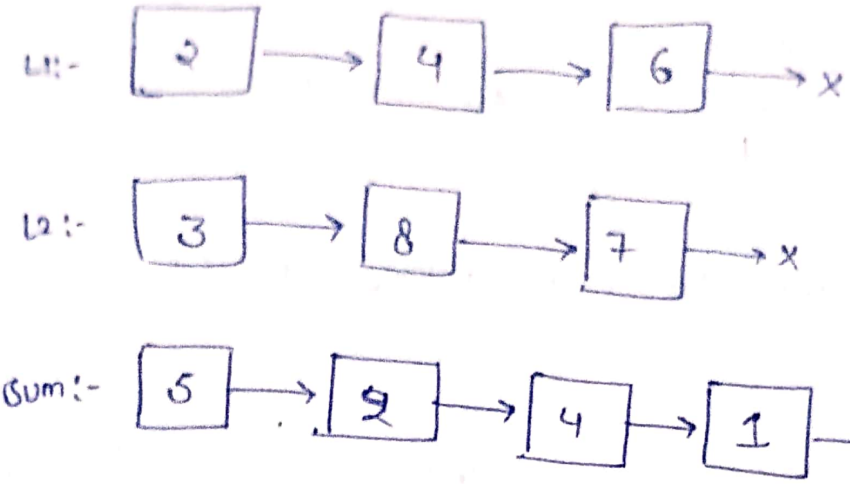
⇒ code :-

```
last = NULL,  current = head
while (current != NULL) {                    → O(n)
        last = current → back;
        current → back = current → next;
        current → next = last;
        current = current → back;
}
if (last → back != NULL) {
        return  head = last → back;
}
return head;
```

⇒ TC :- O(n)

SC :- O(1)

Add 2 numbers in LL :- (dummy node)




```
  1
  642
+ 783
 1425
```

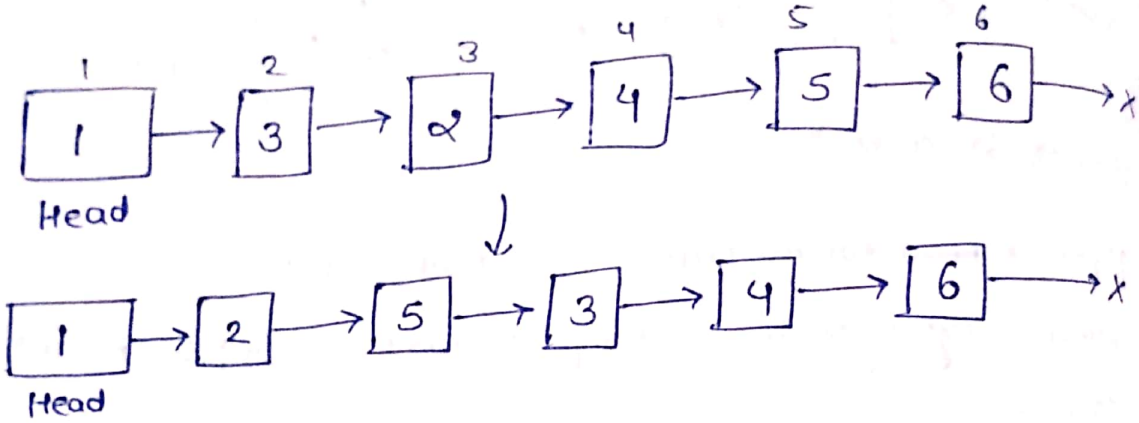
~~Brute~~ & code :-

```
Node * head dummyNode = new Node (-1);
temp = dummyNode , temp1 = head1 , temp2 = head2
carry = 0;
while (temp1 != NULL || temp2 != NULL) {
    sum = carry
    if (temp1)  sum = sum + temp1->data;
    if (temp2)  sum = sum + temp2->data;
    carry = sum/10;
    Node * newNode = new Node (sum%10);
    temp->next = newNode;
    temp = temp->next;
    if (temp1)  temp1 = temp1->next;
    if (temp2)  temp2 = temp2->next;
}
if (carry) {
    Node * newNode = new Node (carry);
    temp = temp->next;
}
```

return dummyNode->next;

* TC :- $O(\max(N_1, N_2))$;
SC :- $O(\max(N_1, N_2))$ → for storing the Ans.

⇒ Segregate odd & even nodes in LL :-



⇒ Brute :- (using arrays)

```
if (head == NULL || head->next == NULL) {
        return head;
}
list<int> arr;
Node *temp = head;
// for the odd index :-
while (temp != NULL && temp->next != NULL) {
    arr.push_back(temp->data);
    temp = temp->next->next;          → O(n/2)
}
if (temp) arr.push_back(temp->data);
```

for the Even Index :-

```
temp = head->next;
while (temp != NULL && temp->next != NULL) {
    arr.push_back(temp->data);
    temp = temp->next->next;
}
if (temp) arr.push_back(temp->data);
```

→ O(n/2)

push the elements back into the linked list :-

```
temp = head;
for (auto it : arr) {
    temp->data = it;
    temp = temp->next;
}
return head;
```

→ O(n)

⇒ TC :- O(2n)
SC :- O(n)

Better :- (using dummy node)

```
if(head == NULL || head->next == NULL) return head;
int cnt = 1;
Node *oddHead = new Node(-1);
Node *evenHead = new Node(-1);
temp1 = oddHead;
temp2 = evenHead;
temp = head;
```

```
while (temp) {
    if (cnt % 2 != 0) {
        temp1 -> next = temp;          ---> O(n)
        temp1 = temp1 -> next;
    }
    else {
        temp2 -> next = temp;
        temp2 = temp2 -> next;
    }
temp2 -> next = NULL;
temp1 -> next = evenHead -> next;
Node * newHead = oddHead -> next;
free(oddHead);
free(evenHead);
return newHead;
```

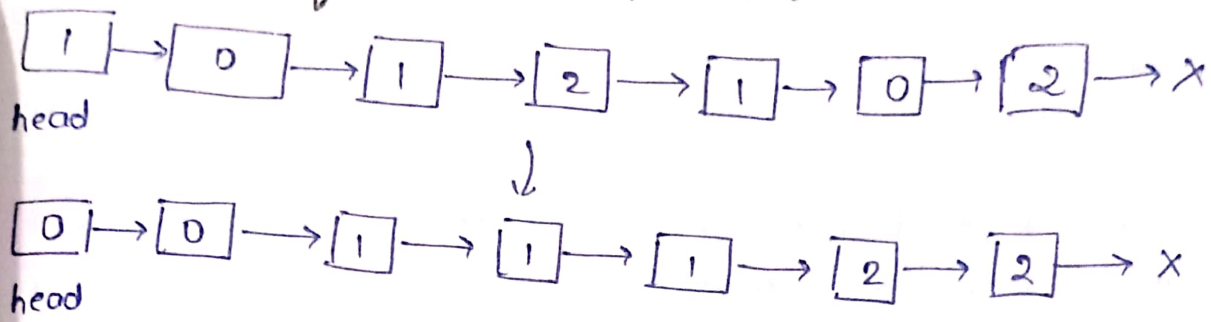
TC :- O(n)
SC :- O(1)

⇒ Optimal :- (In-Place)

```
if (head == NULL || head -> next == NULL) return h
evenNode = head -> next , odd = head , even = head -> n
while (even != NULL && even -> next != NULL) {
    odd -> next = odd -> next -> next;      ---> O(n/2) x 2 = O
    even -> next = even -> next -> next;
    odd = odd -> next;
    even = even -> next;
}
odd -> next = evenNode;
even -> next = NULL;
return head;
```

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

⇒ Sort a LL of 0's, 1's & 2's :-



⇒ Brute :-

```
cnt0=0 , cnt1=0 , cnt2=0 , temp = head;
while (temp) {
    if (temp→data ==0)   cnt0 +=1 ;
  else if (temp→data ==1)   cnt1+=1;
    else  cnt2+=1;                    → O(n)
   temp = temp→next;
}
temp =head;
while (temp) {
    if (cnt0 != 0) {
        temp→ data =0;
        cnt0 --1;                          TC :- O(2n)
    }
                          → O(n)           SC :- O(1)
  else if (cnt1 != 0) {
        temp→data = 1;
        cnt1 --1;
    }
  else {
      temp→ data =2;
```

→ Optimal :-

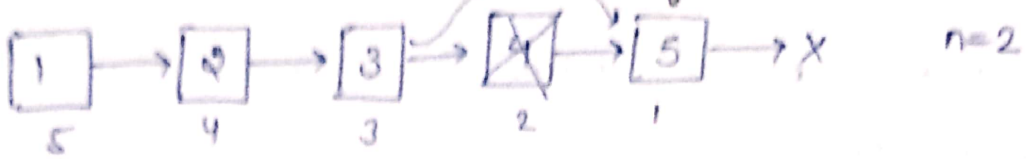
Dummy Node —

```
Node * zeroHead = new Node (-1);
Node * oneHead = new Node (-1);
Node * TwoHead = new Node (-1);
temp = head , zero = zeroHead , one = oneHead ,
Two = TwoHead
While (temp) {
    if (temp → data == 0) {
        zero → next = temp;
        zero = temp;
    }
    else if (temp → data == 1) {
        one → next = temp;
        one = temp;
    }
    else {
        two → next = temp;
        two = temp;
    }
temp = temp → next;
}
// Imp cond^n —
Zero → next = (oneHead → next) ? oneHead → next : twoHe
one → next = twoHead → next;
two → next = NULL;
free(all dummy Nodes)
```

TC :- O(n)

SC :- O(1)

Remove nth node from back of LL :-



→ Brute :-

code :-

```
temp = head;
cnt = 0;
while (temp) {
    cnt++;                    → O(n)
    temp = temp->next;
}
temp = head;
if (cnt == n) {
    head = temp->next;
    free (temp);
    return head;
}
int res = cnt - n;
while (temp) {
    res --;
    if (res == 0) {
                              → O(N)
        break;
    }
    temp = temp->next;
}
temp->next = temp->next->next;
// free that delete node
return head;
```

TC :- O(n) + O(N)

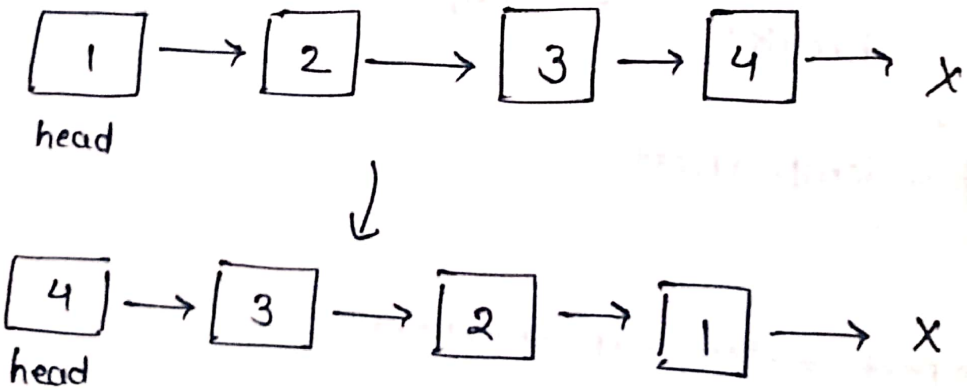
SC :- O(1)

⇒ Optimal :- (fast & slow pointer approach)

```
if (head == NULL)   return head;
fast = head = slow
while (n>0){
    fast = fast → next;
    n--;
}
if (fast == NULL){
    head = head → next;
    return head;
}}
while (fast → next != NULL){
    fast = fast → next;
    slow = slow → next;
}
slow → next = slow → next → next;
return head;
```

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

⇒ Reverse a LL :-



Brute :-

```
Node * temp = head;
Stack<int> st;
while (temp) {
    st.push(temp->data);
    temp = temp->next;
}
temp = head;
while (temp) {
    temp->data = st.top();
    st.pop();
    temp = temp->data;
}
return head;
```

] O(n)

] O(n)

TC :- O(2n)

SC :- O(n)

optimal :-

Iterative :-

```
prev = null , temp = head .
While (temp) {
    ~~first~~ front = temp->next;
    temp->next = prev;
    prev = temp
    temp = front
}
return prev
```

] O(n)

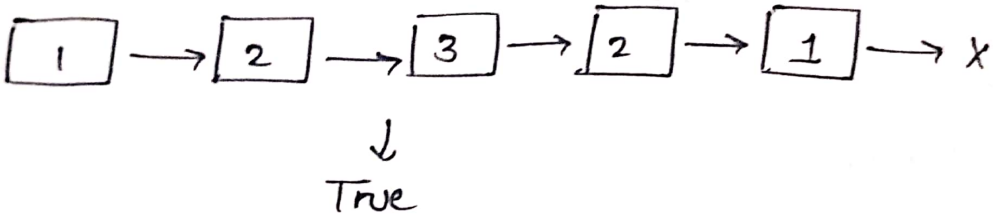
TC :- O(n)

SC :- O(1)

→ Recursive :-

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

→ Palindrome of LL :-



→ Brute :-

```
stack<int> st;
Node * temp = head;
while (temp) {
st.push(temp->data);
temp = temp->next;
}
temp = head;
while (temp) {
    if(temp->data != st.top()) {
        return false;
    }
```

] O(n)

] O(n)

TC :- O(2n)

SC :- O(n)

```
st.pop();
temp = next(temp)
}
return true;
```

→ Optimal :-

```
slow = head = fast
While ( fast->next != NULL && fast->next->next != NULL )
{
    slow = slow->next;
    fast = fast->next->next;
}
```

] O(n/2)

```
newHead = reverse_LL( slow->next ); → O(n/2)
first = head , second = newHead
while (second) {
    if (first->data != second->data) {
        newHead->next = reverse_LL( newHead );
        return false;
    }
    first = first->next;
    second = second->next;
}
reverse_LL(newHead); → O(n/2)
return true;
```

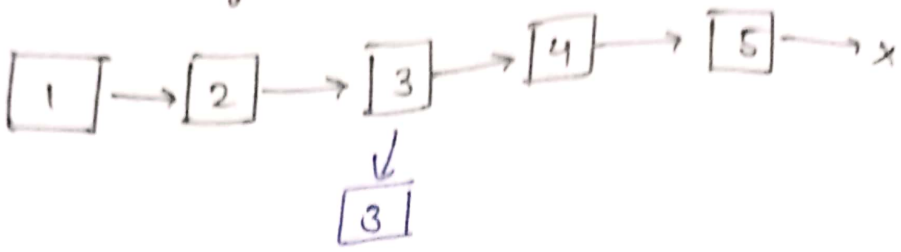
(define in prev Que)

] O(n/2)

TC :- O(2n)

SC :- O(n)

→ middle node of LL :- (Tortoise - Hare)

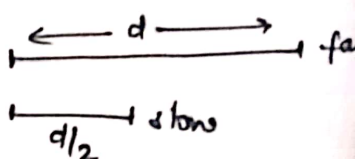


→ Brute :-

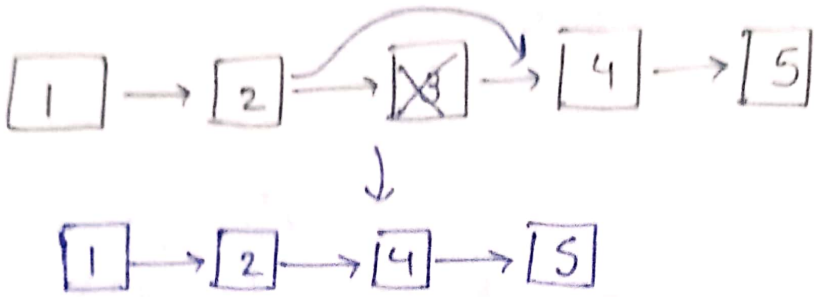
```
temp = head;  cnt = 0
While (temp) {
    cnt ++
    temp = next (temp)
}
mid = (cnt/2) + 1
temp = head
While (temp) {
    mid -= 1
    if (mid == 0)
        break
    temp = next (temp)
}
return temp
```

→ optimal :-

```
fast = slow = temp = head
While (fast != NULL && fast → next != NULL) {
    slow = next (slow)
    fast = ~~next (fast~~ fast → next → next;
}
return slow
```



Delete the middle node of LL :-



Brute :-

→ cnt length & half it

code :-

```
temp = head;   cnt = 0
if (head == null || head→next == null) return null
// find no. of nodes —
while (temp) {
        cnt ++
        temp = temp.next ] O(n)
}
mid = cnt/2
temp = head
while (temp) {
        mid -= 1;
        if (mid == 0) break ] O(n/2)
        temp = temp.next
}
temp.next = temp.next.next;
return head;
```

TC :- $O(n + n/2)$

SC :- $O(1)$

⇒ Optimal :- (Tortoise - Hare approach)

slow = head , fast = head → next → next

if (head == null || head → next == null)

return null

while (fast != null && fast → next != null) {

slow = slow → next;

fast = fast → next → next;

}

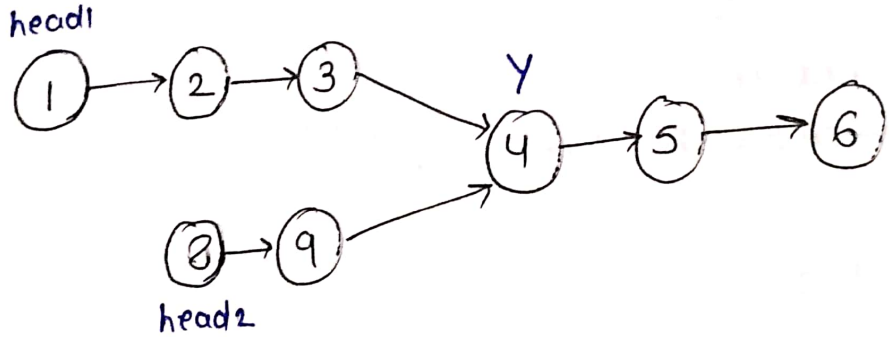
slow → next = slow → next → next;

return head;

TC :- O(n/2)

SC :- O(1)

⇒ find the intersection point of Y LL :-



→ Brute :-

map<Node*, int> mpp , temp = head1

while (temp) {

mpp[temp] = 1;

temp = temp · next

}

temp = head2

```
while (temp) {
    if(mpp.find(temp) != mpp.end())
        return temp;
    temp = temp.next
}
return NULL
```

TC :- O(N1 * map time) + O(N2 * find map time)

SC :- O(N1)

→ Better :-

→ Length difference can be the collision point

Code :-

```
temp1 = head1 , temp2 = head2
N1=0 , N2=0
while (temp1) {
    N1++;
    temp1 = temp1.next;
}
while (temp2) {
    N2++;
    temp2 = temp2.next;
}
if(N1>N2)
    ans = collision_point ( head1, head2 , N1-N2);
else
    ans = collision_point ( head2 , head1, N2-N1);
return ans
```

→ collision_point ( ) {

```
while (n>0) {
    n--;
    temp = temp.next;
}
while ( temp1 != temp2) {
    temp1 = temp1.next;
    temp2 = temp2.next;
}
```

}

TC :- $O(N1) + O(N2) + O(N2 - N1) + O(1)$

$\simeq O(N1 + 2N2)$

SC :- $O(1)$

* optimal :-

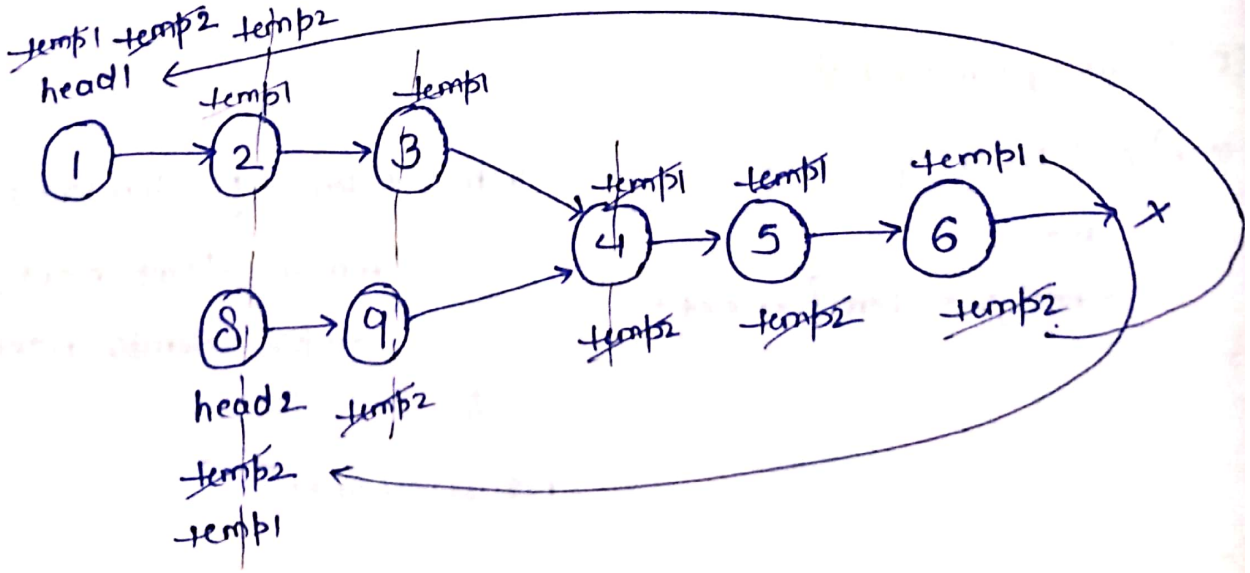
Whenever a pointer reaches null point it to the opposite head.

code :-

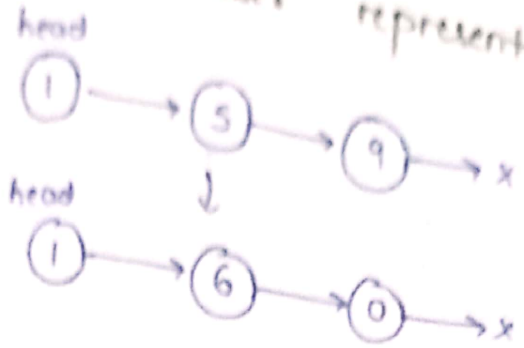
```
temp1 = head1 , temp2 = head2
While (temp1 != temp2) {
    temp1 = temp1.next;
    temp2 = temp2.next;
    if (temp1 == temp2)
    if (temp1 == NULL)
    if (temp2 == NULL)
}
return temp1;
```

```
return temp1;
temp1 = head2;
temp2 = head1;
```

⇒ TC :- O(N1) + some
SC :- O(1)



Add 1 to a number represented by LL :-



159
+1
160

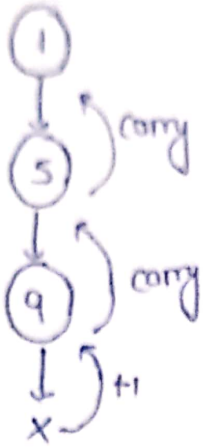
→ Brute :-

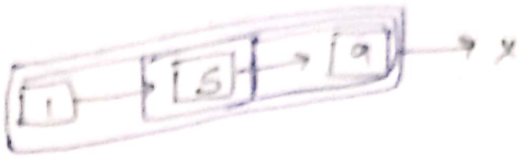
```
carry = 1;
head = reverse_LL (head);  → O(n)
temp = head , last = null
while (temp) {
    temp.data = temp.data + carry
    if (temp < 10) {                    ] O(n)
        carry = 0
        break;
    }
    else {
        temp.data = 0;
        carry = 1;
    }
    temp = temp → next
}
if (carry != 0) {
    newNode (carry)
    last.next = newNode
}
head = reverse_LL (head);  → O(n)
return head;
```

TC :- ~~O(3n)~~ 3O(n)

SC :- O(1)

⇒ Optimal :- (Recursive)





⇒ Code :-

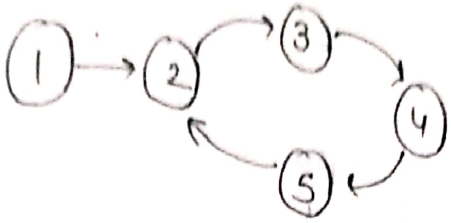
```
add() {
    carry = carryAdder(head)
    if (carry == 1) {
        newNode(carry)
        newNode.next = head
        head = newNode
    }
    return head

→ carryAdder() {
    if (head == null)  return 1
    carry = carryAdder(head → next)
    head.data = head.data + carry
    if (head.data < 10)  return 0
    else {
        head.data = 0;
        return = 1;
    }
}
```

TC :- O(n)

SC :- O(n)

Detect a loop in LL :-



Brute :-

```
map<Node*, int> mpp
temp = head
while (temp) {
    if (mpp.find(temp) != mpp.end()){
        return true;
    }
    mpp[temp] = 1;
    temp = temp.next;
}
return false;
```

TC :- O(n) * map-time)

SC :- O(n)

Optimal :-

```
slow = fast = head
while (fast != null && fast.next != null) {
    slow = slow -> next
    fast = fast -> next -> next
    if (fast == slow) return true;
}
return false;
```

movement

slow → far goes

fast → towards slow

TC :- O(n)

SC :- O(1)

intuition

d+1==0

fast —2 step→ slow

slow ←1 step— fast

fast —1 step→ slow (towards)

d+1 = 1

⇒ Starting point of the loop :-

⇒ Brute :-

Same as detect of the loop

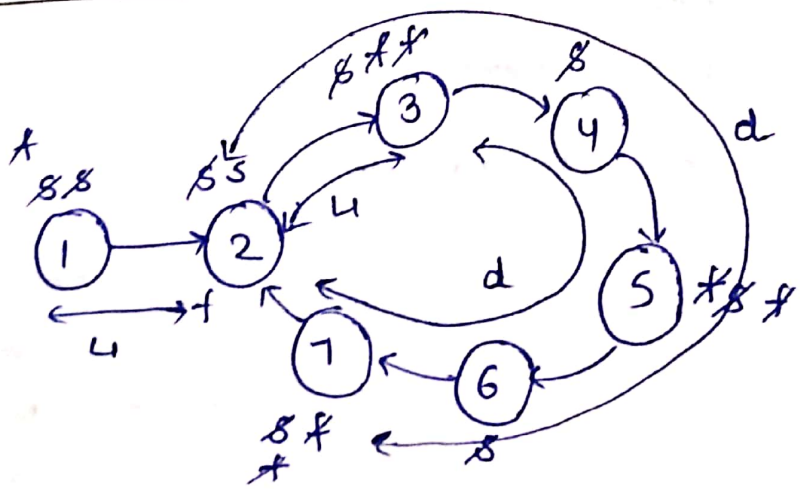
⇒ Optimal :-

```
slow = head = fast
while (fast != null && fast.next != null) {
    slow = slow.next
    fast = fast.next.next
    if (slow == fast) {
        slow = head;
        while (slow != fast) {
            slow = slow.next
            fast = fast.next
        }
        return slow
    }
}
return null
```

TC :- O(n)

SC :-

⇒ Intuition :-



total len = L1 +

fast slow

2d d

remaining = L1

move slow to hea

Length of loop in LL :-

Brute :-

```
cnt = 1 , map , temp = head
while (temp) {
    if (mp.find(temp) != mp.end()) {
        int val = mp[temp]
        return cnt - val
    }
    mp[temp] = cnt
    cnt ++
    temp = temp.next
}
return -1;
```

TC :- $\leq O(n)$
SC :- $O(n)$

optimal :-

```
slow = fast = head , len = 0
While (fast != null && fast.next != null) {
    slow = slow.next
    fast = fast.next.next
    if (slow == fast) {
        fast = fast.next;
        While (slow != fast) {
            len ++;
            fast = fast.next;
        }
        return len;
    }
}
return -1;
```

TC :- $\leq O(2n)$
SC :- $\approx O(1)$