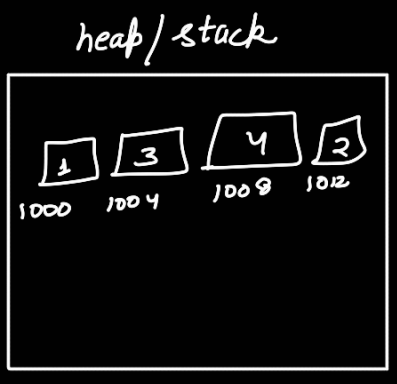
[Linked Lists]:

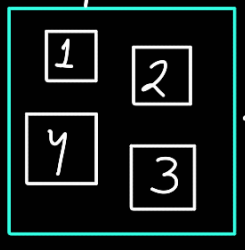
— Striver

↳ not in contiguous locations

eg → array: | 1 | 3 | 4 | 2 |

Linked list: 1 3 4 2



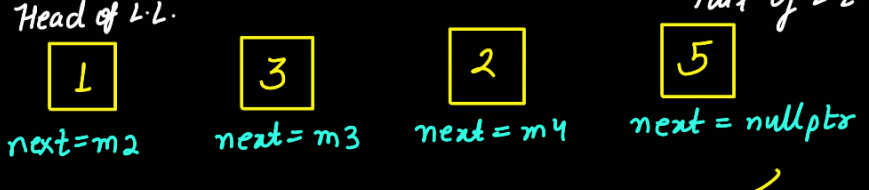


↳ can be anywhere, not contiguous.

* Linked-list is similar to array except, they are not stored in contiguous memory locations.

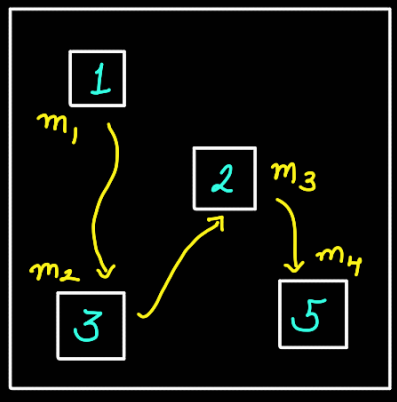
↳ you can also increase / decrease the size unlike arrays.

eg →



head = m_1

creating links to next elements



* to increase the size of L.L.: Create another link to m_5 and shift the tail to next element.

* Where is L.L. used?

⟹ stacks / queues

Real life: Browser History

⇒ above is a 1-D linked-list as it stores address of forward element, it doesn't know what was its previous element.

linked list ki node mein what are we storing?

⇒ data and pointer to the next node.

for this we need to create a user-defined data-type.

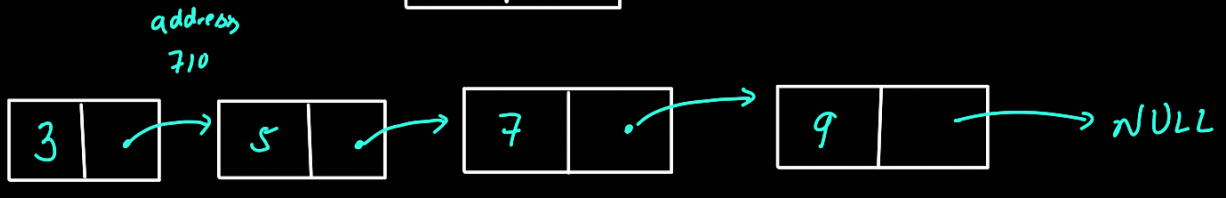
```
struct Node {
    int data;
    Node* next;
Node(data1, next1)
{ data = data1;
  next = next1;
}
};
```

Linked lists: Linear data structure

↓

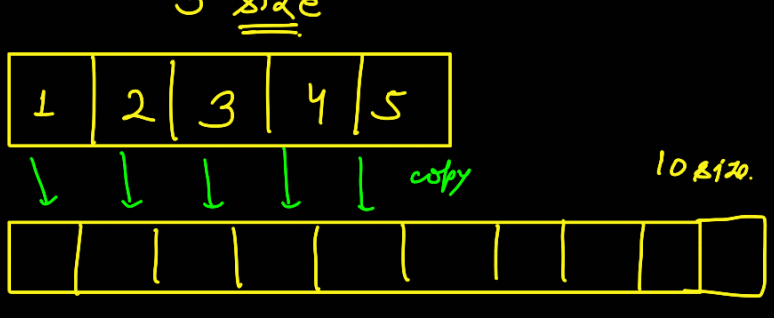
collection of nodes

↓



Why not use vectors →

vector<int> v(5) →



* new storage
* values copy

} not efficient

(when array is full, it doubles up its size & copy elements to that doubled up storage, hence it is not optimal / efficient.

Create a node :

```
class node {
    int data;
    node* next;
    node(int data, node *next) {
        this->data = data;
        this->next = next;
    }
};
```

// constructor

[Diff. b/w Node and Node*]:

Create → `node* y = new node(2, nullptr);`

`node y = node(2, nullptr);`

↳ only object created.

y.data, y.next.

the object created as well as the address of this object (node) is stored in y too as new keyword returns the pointer to the memory location.

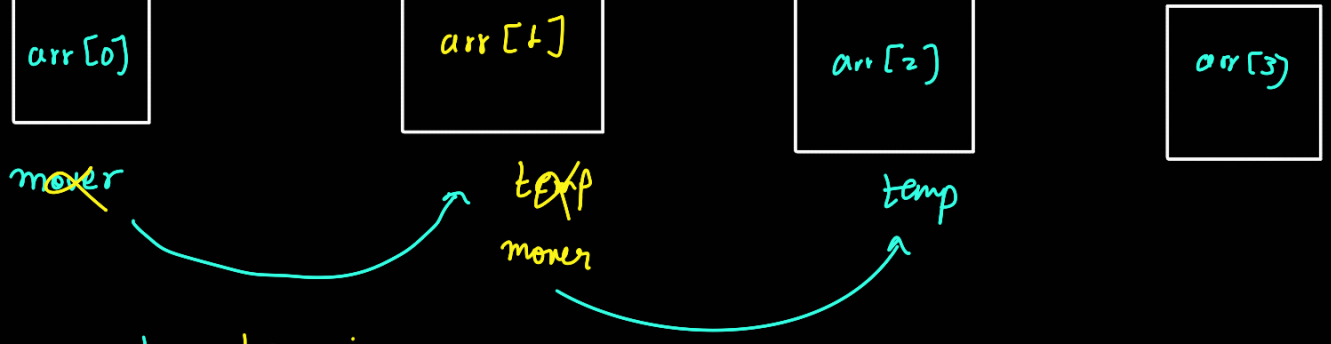
y → data , y → next.

32-bit
int → 4 bytes
* → 4 bytes
8 bytes

64-bit
int → 4 bytes
* → 8 bytes
12 bytes.

Array to Linked list (code in Lappy)

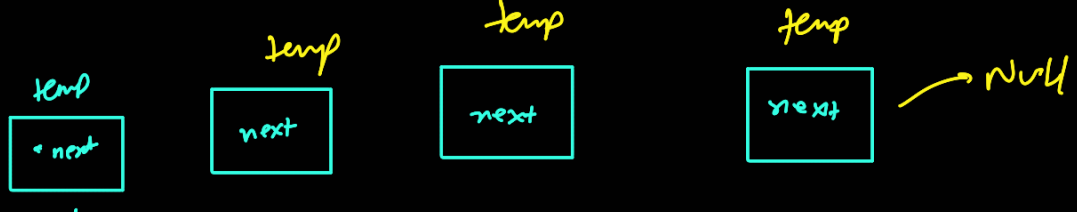
`int arr[] = {1,2,3,4};`




```
mover -> next = temp;
mover = temp;
```

```
node* convertArrayToLL (int arr[]) {
    node* head = new node(arr[0]);
    node* mover = head;
    for (int i = 1; i < n; i++) {
        node* temp = new node(arr[i]);
        mover -> next = temp;
        mover = temp;
    }
    return head;
};
```

Traversal in a LL →



temp = next until temp == NULL

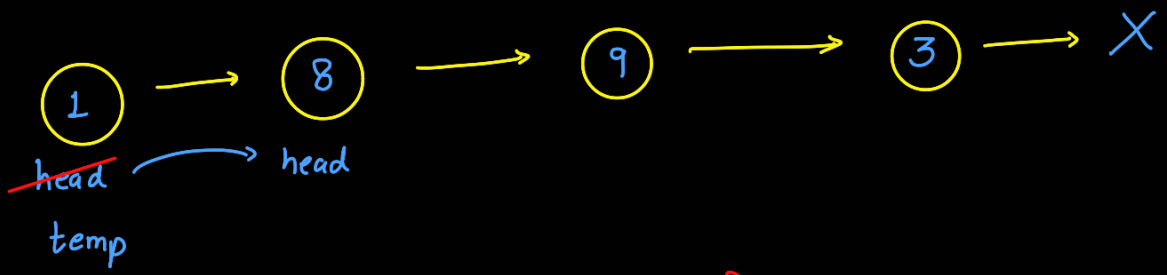
Time Complexity: O(N)

Length of LL, Printing etc in Vs code

Insertion and Deletion

- Delete at the head :

① Deletion of the ...

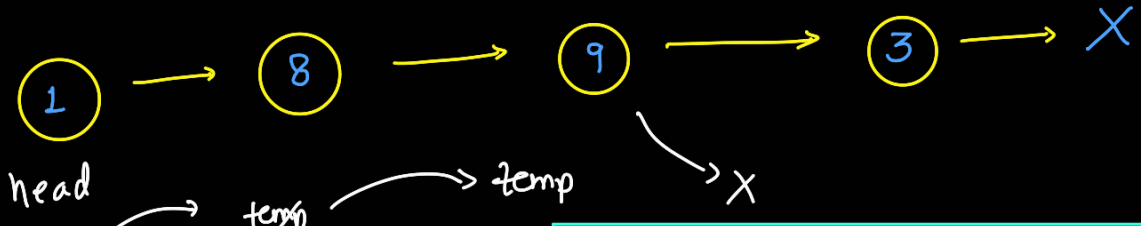


⇒
```
node* deleteHead (node* head) {
    if (head == NULL or head->next == NULL) return NULL;
    node* temp = head;
    head = head->next;
    free (temp);    // or   delete (temp)

    return head;
}
```

* Delete tail of L.L. → see in Vs code.

element before tail → null &
delete tail;

If 1 element in L.L ⇒ delete 1 element (as that 1 element will be the tail as well as the head).



tip → start thinking of the edge cases, then try to build your logic around it.

Delete K^{th} node :

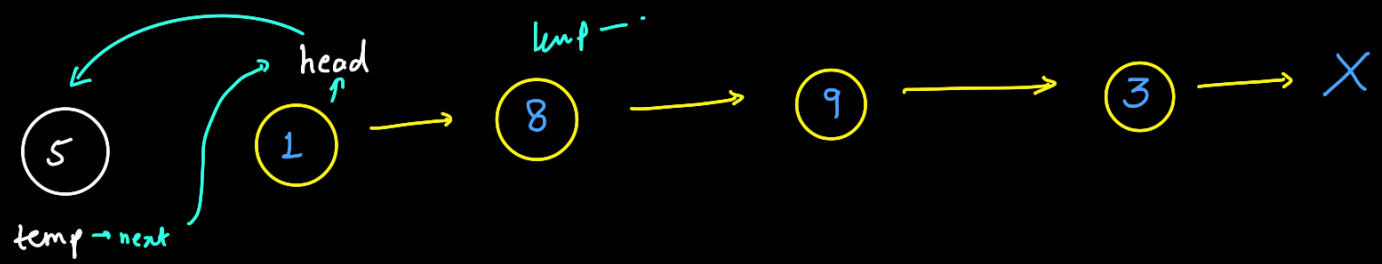
⇐ Pseudocode

```
node* prev = nullptr;
node* temp = head.
while (temp) {
    temp = temp->next;
    cnt++;
    if (cnt == k) {
        temp->next = temp->next->next;
```

free (temp);
break }

Insertion

① Insert at head :



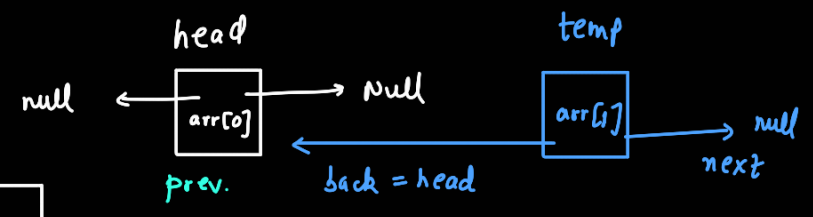
temp → next = head;
head = temp

DOUBLY LINKED LIST :

↳ Singly L.L. + pointer to previous element.

```
class node {
    int data;
    node* next;
    node* back;
    node (int data){
        this → data = data;
        this → next = nullptr;
        this → back = nullptr;
    }
    node (int data, node* next, node* back){
        this → data = data;
        this → next = next ;
        this → back = back ;
```

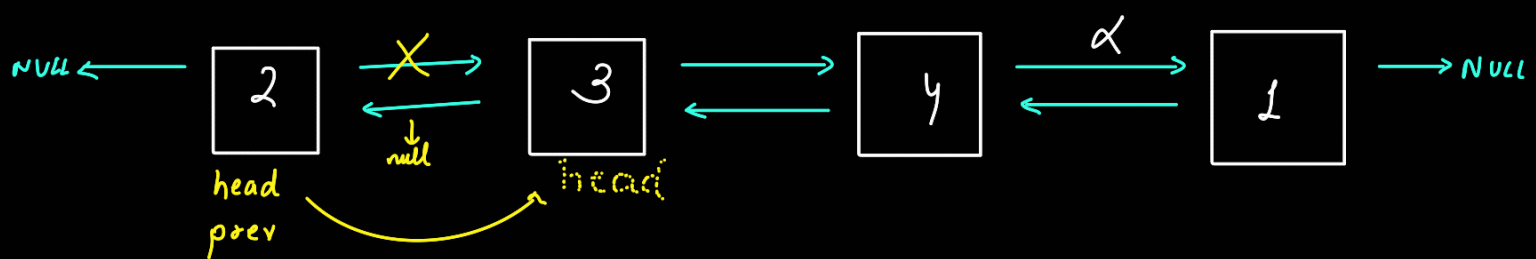
* Array to DLL :




```
node* convertArrayTODLL (vector<int>&arr){
    node* head = new node (arr[0])
    node* prev = head   // copy of head
    for (int i=1; i< arr.size(); i++){
    node* temp = new node (arr[i], nullptr, prev);
        prev → next = temp;
        prev = prev → next;    // prev = temp
    }
    return head;
}
```

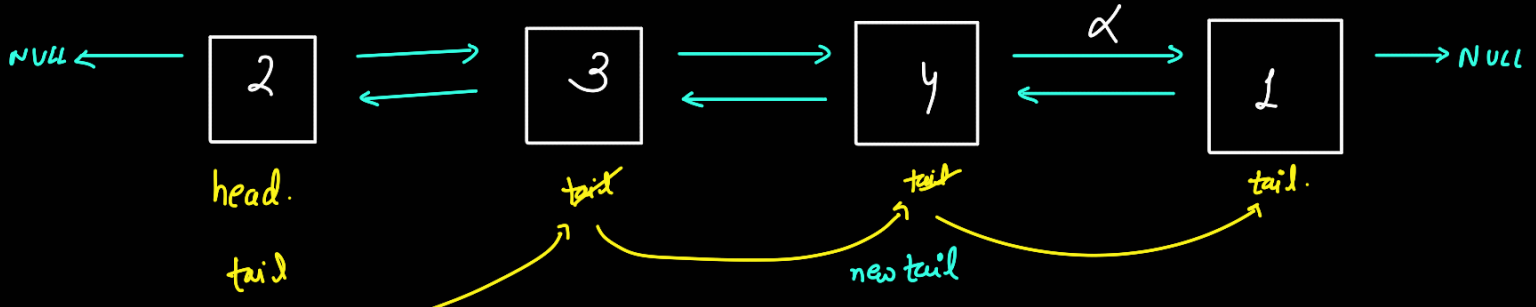
```
    }
};
```

[Delete Head of the L.L.]:




```
prev = head;
head = head → next;
prev → next = null;
head → back = null;
free (prev);
```

Delete Tail of the DL.L :




```
node* deleteTail (node* head) {
    node* tail = head;
    while (tail → next != null) {
        tail = tail → next;
    }
    node* newtail = tail → back;
    tail → back = null;
```

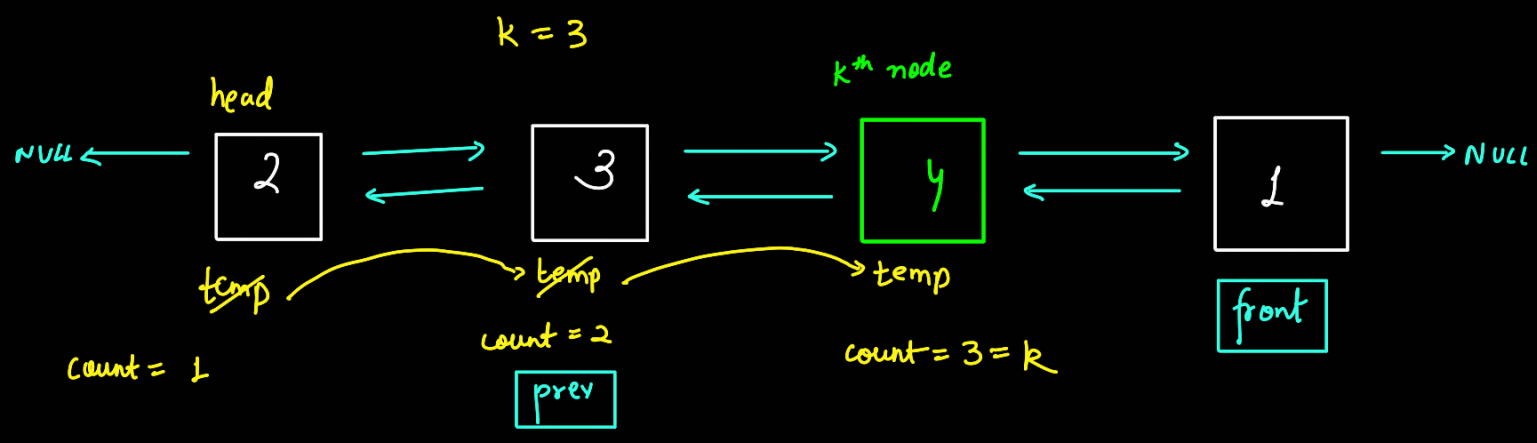
newtail → next = NULL;

delete tail;

return head;

}

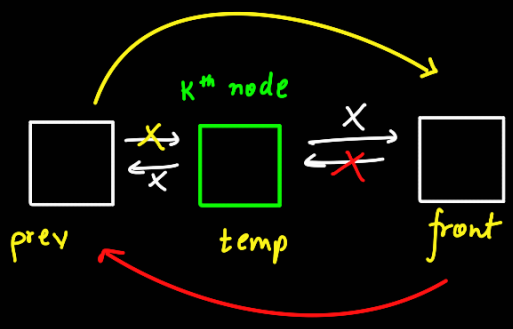
Delete Kth node:



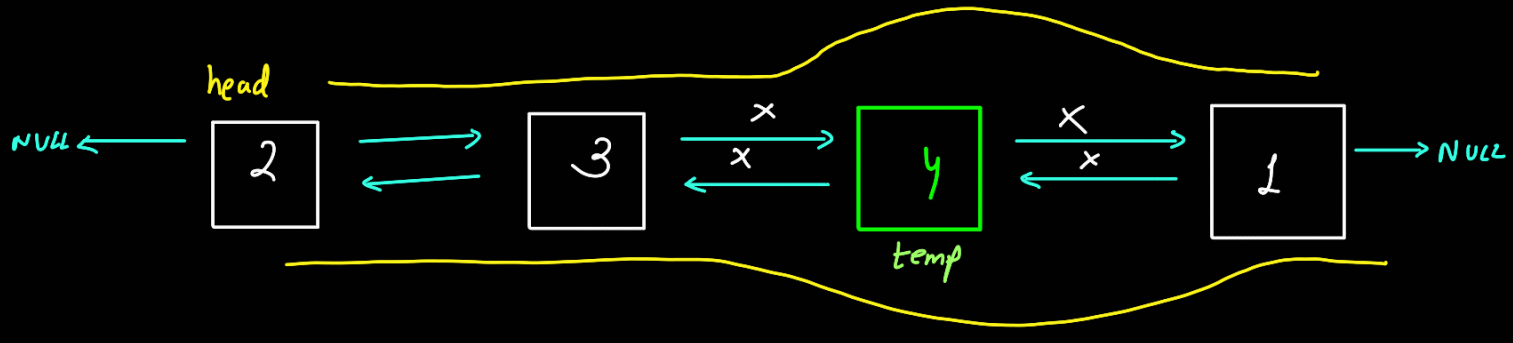

```
node* deleteKthNode (node* head, int k) {
    if (head == null) return nullptr;

    node* temp = head;
    int count = 0;

    while (temp) {
        count++;
        if (count == k)
            break;

        temp = temp -> next;
    }

    if (prev == NULL && front == NULL) { // single noded DLL
        delete temp;
        return null;
    }
    else if (prev == null) {
        return deleteHead (head); // func" we made above.
```

```
else if ( front == null) {
    return deleteTail (head);  // func" we made above.
```

```
else {
    prev → next = front
    front → back = prev
    temp → next = null;
    temp → back = null;
    delete temp;
}
return head;
}
```



[Delete node from a L.L.]: (node != head)

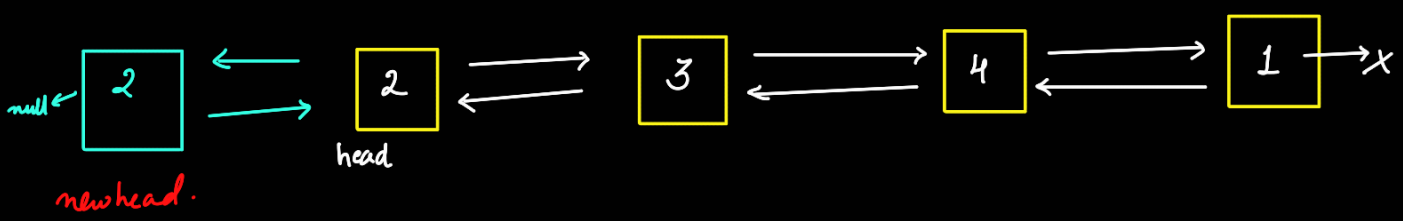



```
void delete (node* temp) {
    node* front = temp → next;
    node* prev = temp → back;
    if (front == null) {
        prev → next = null;
        temp → back = null;
        free (temp);
        return;
    }
    prev → next = front;
    front → back = prev;
```

```
temp->next = temp->back = null;
free temp;
}
```

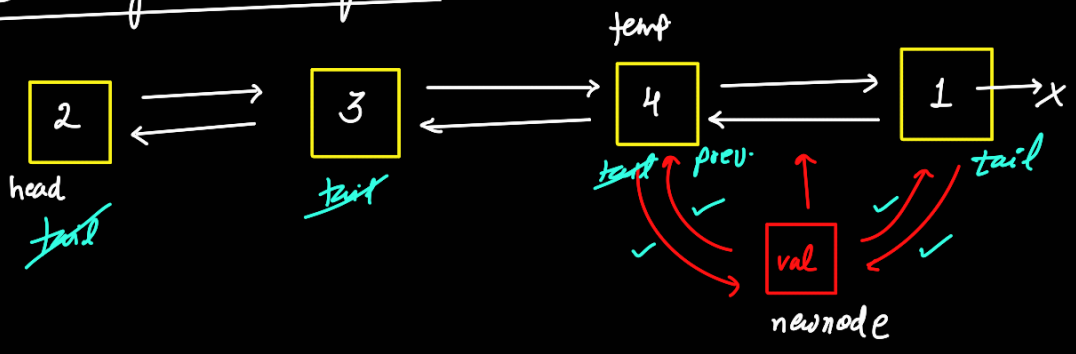
INSERTION BEFORE THE NODE:

insert before head of L.L. =>




```
node* newhead = new node ( val, head, nullptr);
head -> back = newhead;
return newhead;
```

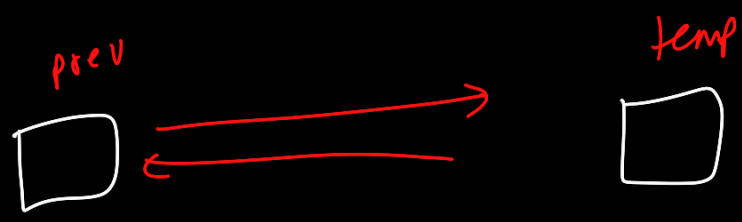
insert before tail of L.L → similar to inserting before Kth node.

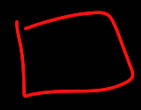


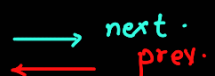

```
tail = head;
while (tail) {
tail = tail -> next;
}
```

```
if (tail -> next == NULL) {
return insertBeforeHead (head, val);
```

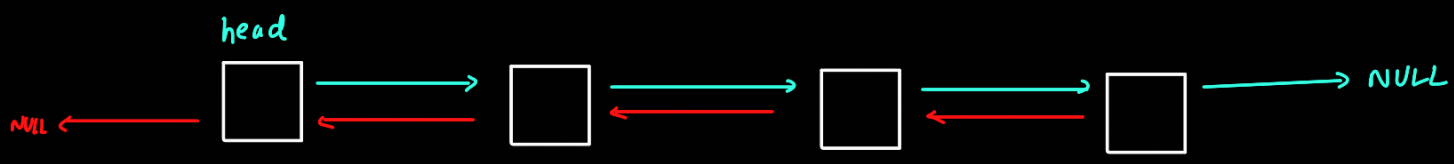
```
prev = tail -> back;
newnode = newnode (val, tail, prev);
prev -> next = newnode;
tail -> back = newnode;
```

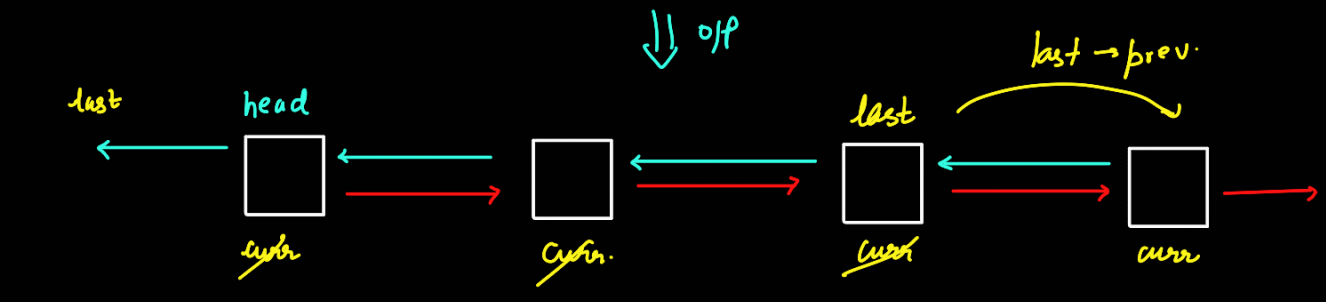






Ques → Reverse a DLL



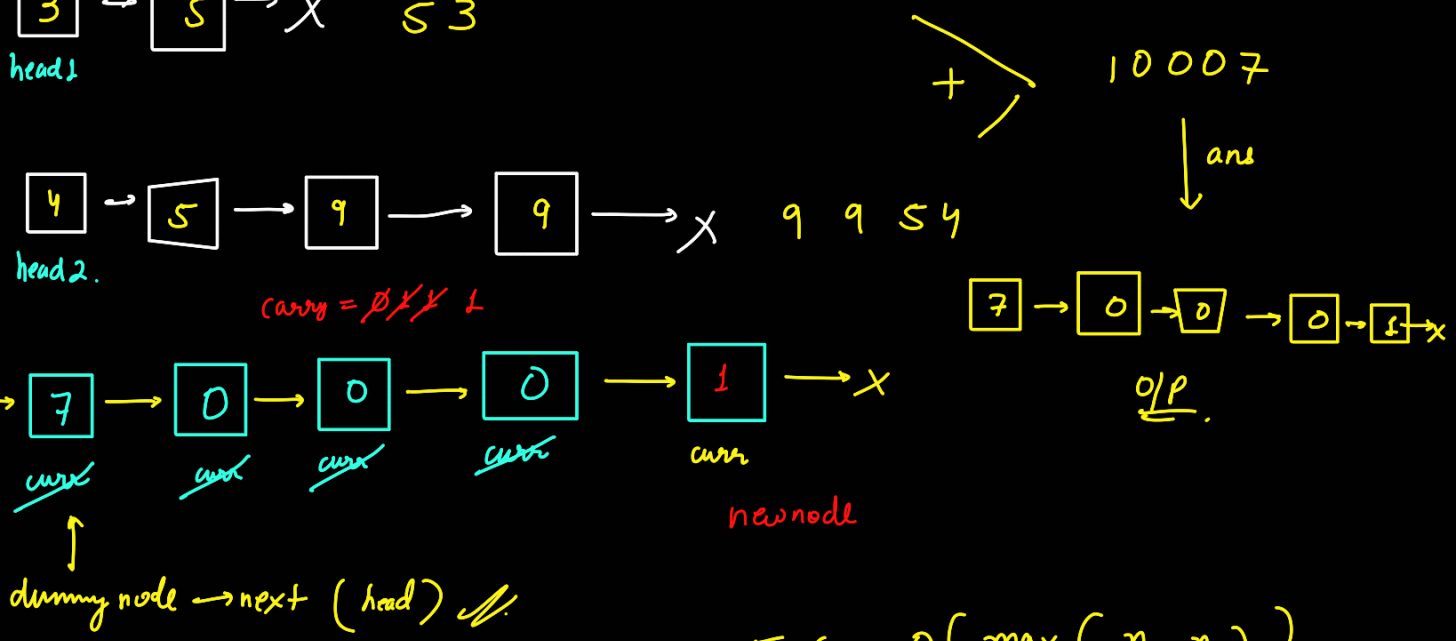



```
node* curr = head;
node * last = nullptr;
while (curr) {
    last = curr → prev;
    curr → prev = curr → next;
    curr → next = last
    curr = curr → prev;
}
return. last → prev;
```

```
temp = a
a = b
b = temp.
```

[Linked-List Questions]:

Ques → Add-2 numbers ⇒



$T.C = O(\max(n_1, n_2))$

$S.C. = O(\max(n_1, n_2))$

```
node* addTwoNum (node* head1, node* head2) {
    node* dummynode = new node (-1); // makes implementation easier.
    node* curr = dummynode;
    node* temp1 = head1;  // to traverse L1
    node* temp2 = head2;  // to traverse L2
    int carry = 0;
    while (temp1 or temp2) {
    int sum = carry;
        if (temp1) sum += temp1 → data;
        if (temp2) sum += temp2 → data;
        node* newnode = new node (sum % 10);
              carry = sum/10;
              curr → next = newnode;
              curr = curr → next;
        if (temp1) temp1 = temp1 → next;
        if (temp2) temp2 = temp2 → next;
    }
    if (carry) {
              node* newnode = new node (carry);
              curr → next = newnode;
    }
```

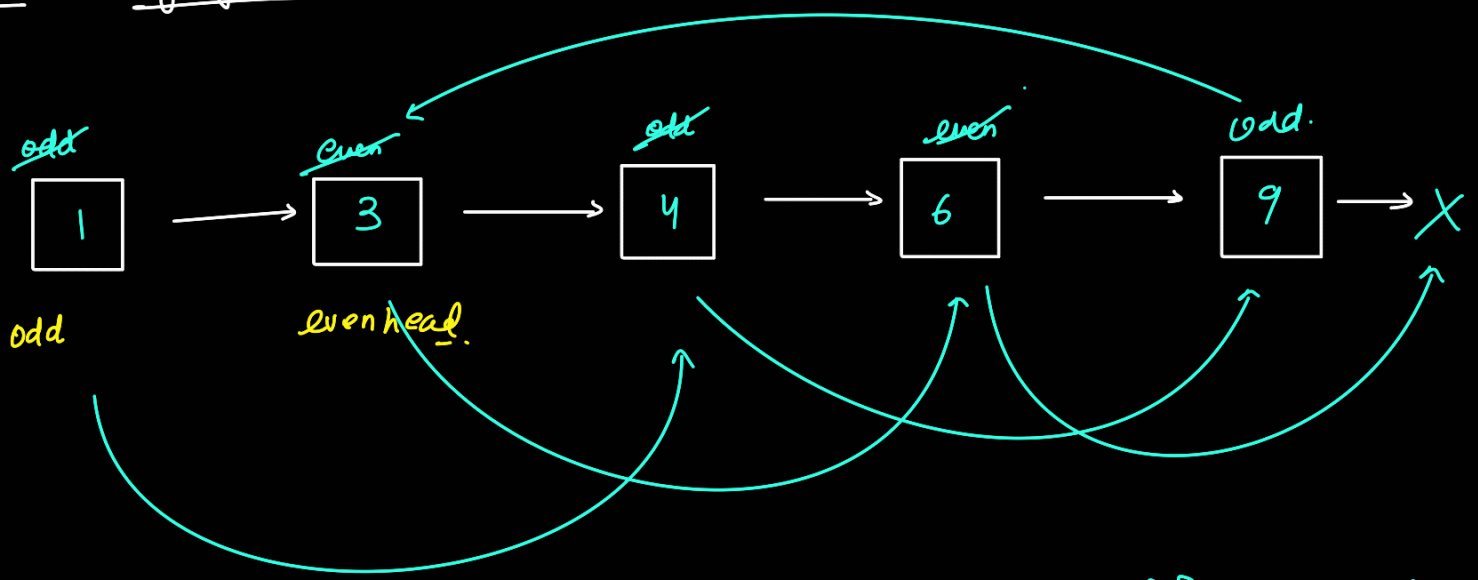
```
}
return dummynode -> next;   // head of the ans linked list.
}
```

then

```
temp -> next -> next;
```

temp -> next != NULL or it will throw an error.

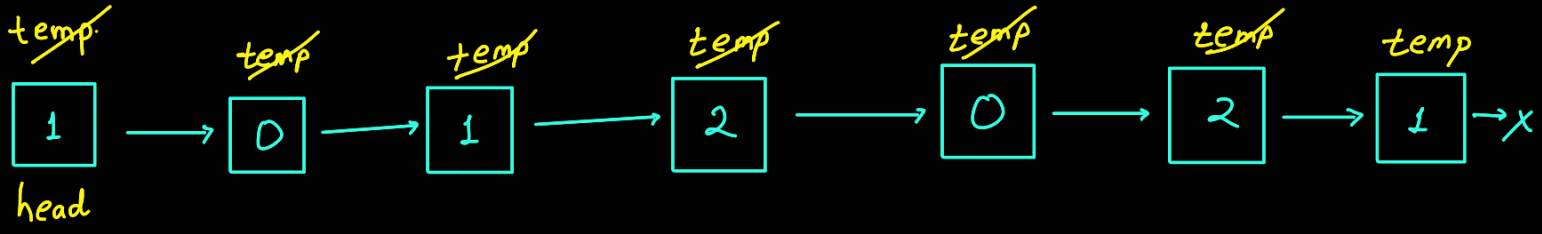
Ques -> Segregate even and odd indices ->




```
while (even != NULL && even->next != NULL)
{
```

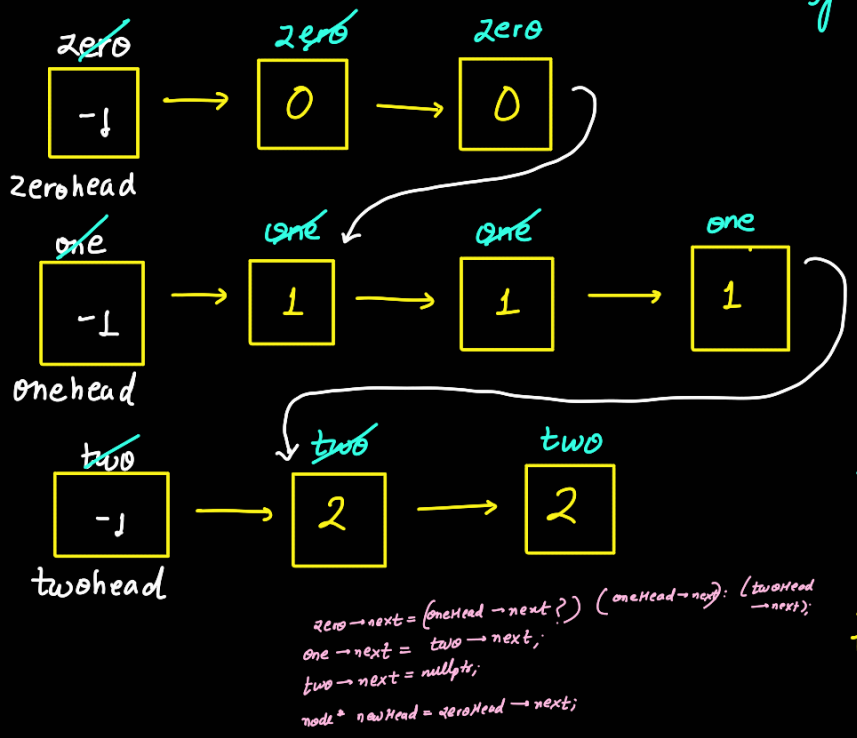
Dummy Node concept :

Ques: Seggregate a L.L. of 0's, 1's and 2's :



three dummy nodes

```
while (temp != NULL) {
  if (temp->data == 0) {
```




```
zero->next = (oneHead->next?) (oneHead->next): (twoHead->next);
one->next = two->next;
two->next = nullptr;
node* newHead = zeroHead->next;
```

```
if (temp->data==0) {
    zero->next = temp
    zero = zero->next }

if (temp->data==1) {
    one->next = temp
    one = one->next }

if (temp->data==2) {
    two->next = temp
    two = two->next }

temp = temp->next

}
```

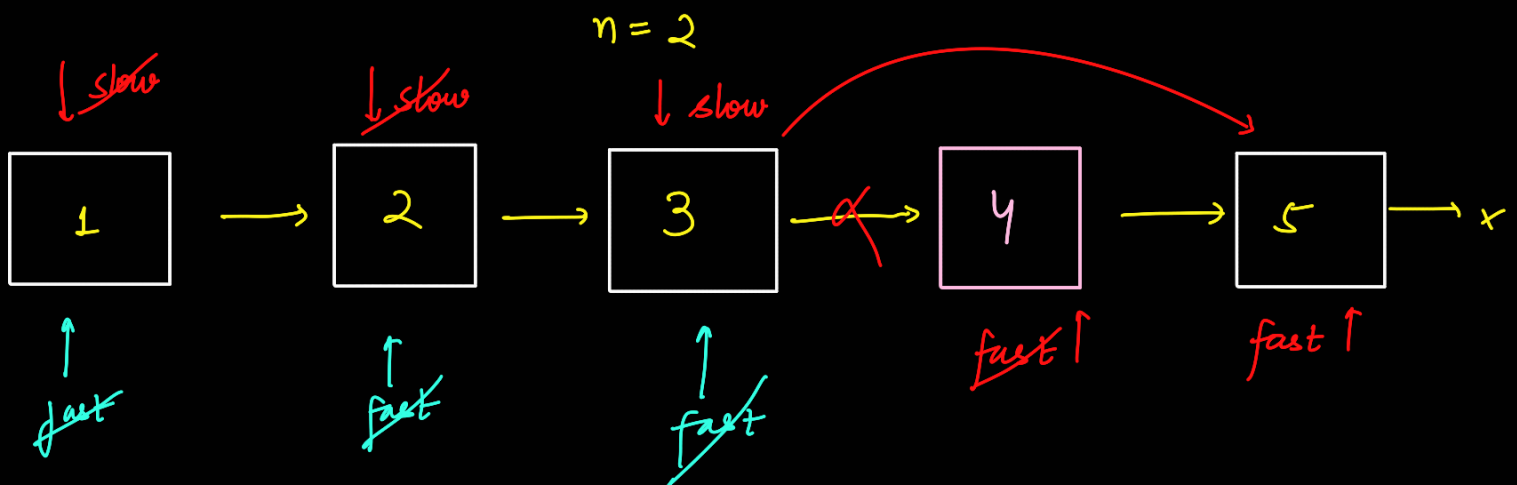
Code :

```
node* sortlist (node* head) {

if (!head or !head->next) return head;

node* zeroHead = new node (-1), node* zero = zeroHead;
node* oneHead = new node (-1), node* one = oneHead;
node* twoHead = new node (-1), node* two = twoHead;
node* temp = head;
while (temp) {
    if (temp->data==0) {
        zero->next = temp;
        zero = zero->next; }
    else if (temp->data==1) {
        one->next = temp;
        one = one->next; }
    else {
        two->next = temp;
        two = two->next;
```

```
    }
    temp = temp -> next;
  }
  zero -> next = (oneHead -> next ?) (oneHead -> next) : (twoHead -> next);
  one -> next = two -> next;
  two -> next = nullptr;
  node* newHead = zeroHead -> next;
  delete zeroHead;
  delete oneHead;
  delete twoHead;
  return newHead;
}
```

→ O(N) and O(1)

Delete n^{th} node from last in 1 traversal :




```
for (i = 0 to i < n) {
    fast = fast -> next;
}
if (!fast) return head -> next; free(head);
while (fast -> next) {
    slow = slow -> next;
    fast = fast -> next;
```

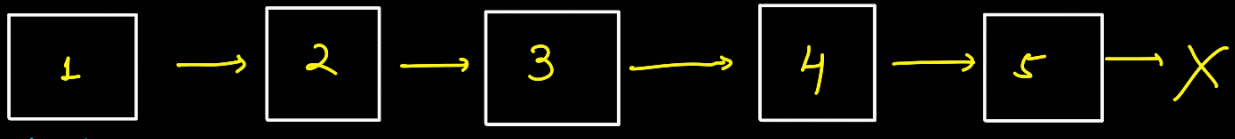
```
    }
    node* delNode = slow -> next;
    slow -> next = slow -> next -> next;
    free (delNode);
    return head;
}
```

`while (fast -> next)` → to stop at last element

`while (fast)` to traverse whole L.L.

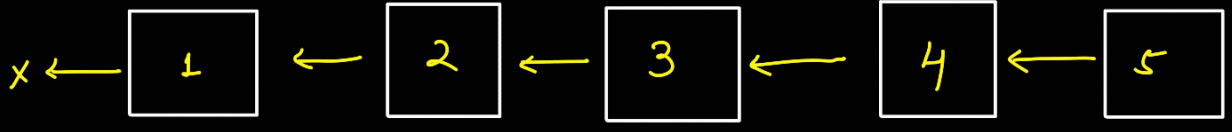
Ques: Reverse a Linked-list :

(a) Iterative →



head

Reverse the links




```
temp = head;   prev = front = nullptr;
while (temp) {
    front = temp -> next;
    temp -> next = prev;
    prev = temp;
    temp = front;
}
```

```
    }
    ret. prev;
}
```

T.C = O(N) ; S.C = O(N)

(b) Recursive:

```
reverse (head){
  if ( !head or !head→next) return head;
  Node* newhead = reverse ( head→next);
  Node* front = head→next;
  front→next = head;
  head→next = nullptr;
  return newhead;
}
```

Palindrome Linked list :

naive → using stack — O(2N) and O(N)

Two pointers → O(2N) and O(1).

Steps:

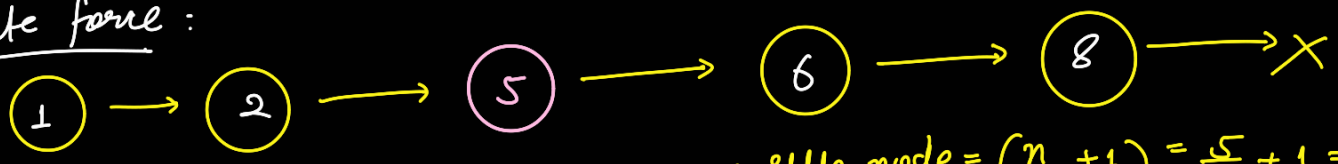
find middle → reverse middle's next → compare first half & second half.

See code on website

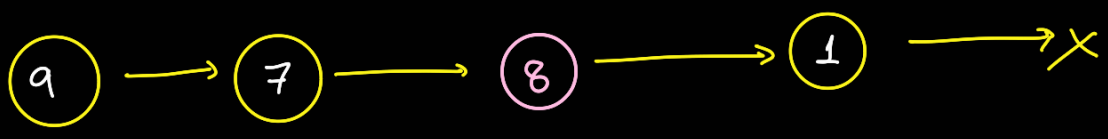
Middle of the linked-list :

$n = 5$

Brute force :



middle node = $\left(\frac{n}{2}+1\right) = \frac{5}{2}+1 = 3^{rd}$



$n = 4$

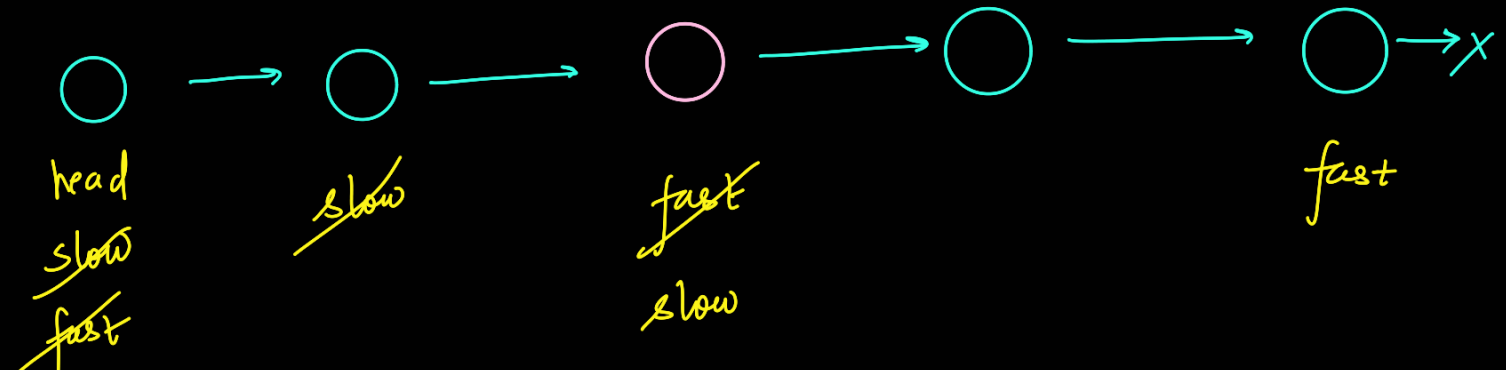
middle = $\frac{n}{2}+1 = \frac{4}{2}+1 = 3^{rd}$ node

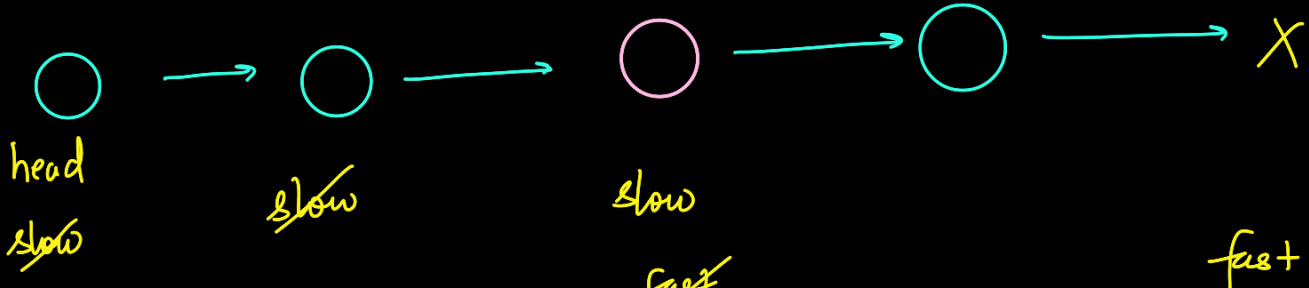
$O(N + \frac{N}{2})$

Optimal : Tortoise and Hare Algorithm :

fast pointer. ⇒ move by 2 steps.

slow pointer ↳ move by 1 step.



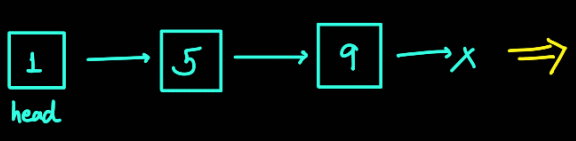


fast fast

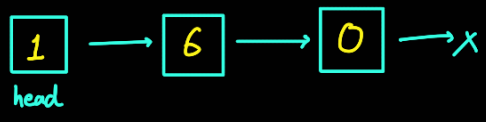
(odd) (even)

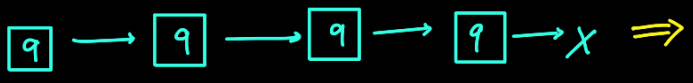
```
while ( fast & fast -> next ){
    slow = slow -> next;
    fast = fast -> next -> next;
}
return slow;
```

Ques → Add 1 to Linked-List :



⇒





⇒


```
head = Reverse (head);
    node* temp = head;
    int carry = 1;
    while (temp) {
        temp -> data += carry;
        if (temp -> data < 10) {
            carry = 0;
            break; }
        else { carry = 1
               temp -> data = 0
        }
        temp = temp -> next;
```

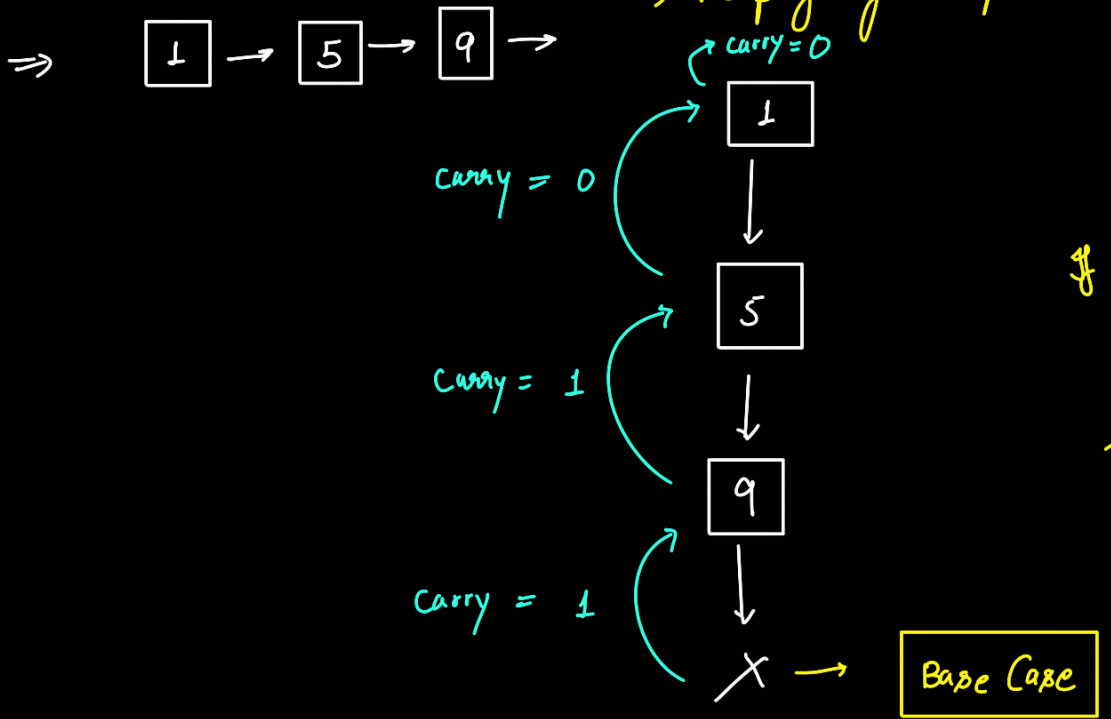
```
temp = temp->next;
}
if (carry) {
head = reverse(head);
node* newnode = new node(1, head)
return newnode;
}
head = reverse(head);
return head;
}
```

O(3N) & O(1)

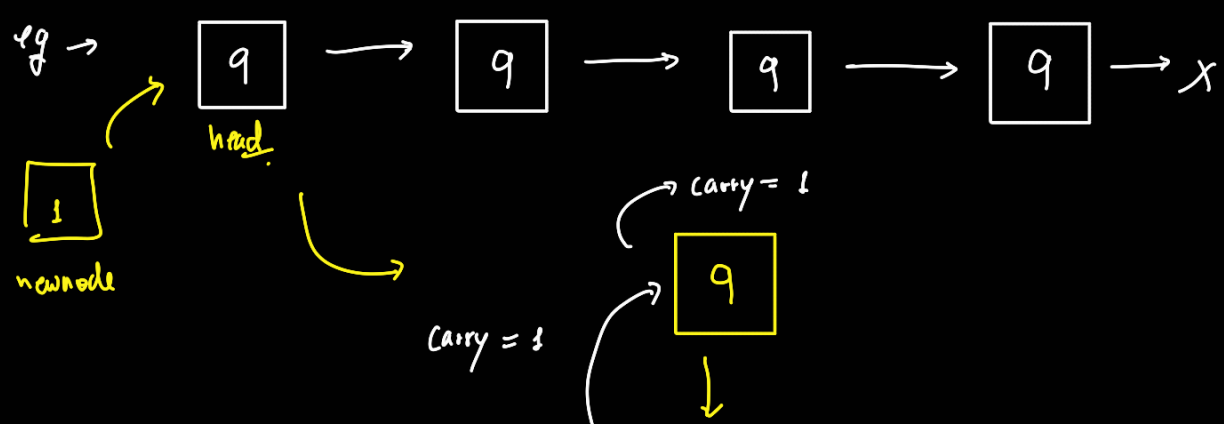
Recursive → O(N):

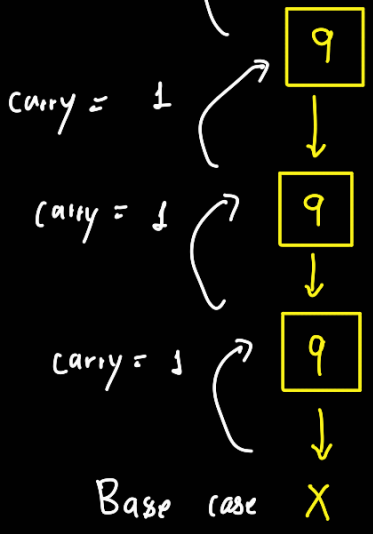
(Back-tracking):

↳ keep going deep into recursion:




```
if (sum == 10)
  { temp -> data = 0;
return 1; }
else {temp->data = sum;
  return 0; }
```



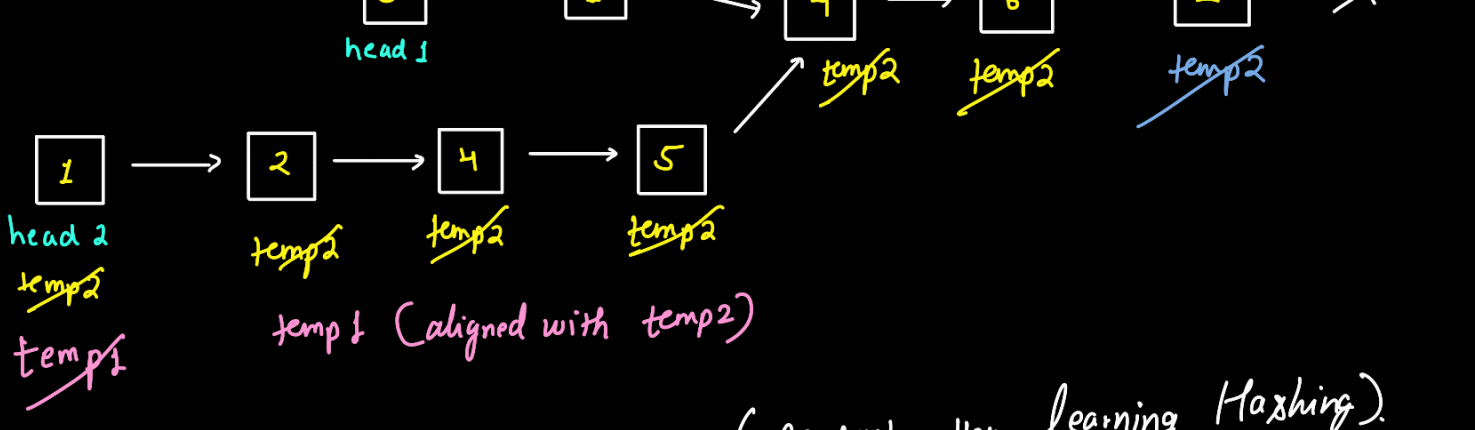



```
helper (temp)
    if (temp == NULL) return 1;
carry = helper (temp->next);
 temp -> data += carry;
    if (temp -> data < 10) return 0;
else {temp -> data = 0;
    return 1;
 }

plusOne (head)
     carry = helper (head);
    if (carry == 1) {
          newnode = newNode (1);
          newnode -> next = head;
           return newnode;
       }
     return head;
    }
```

T.C = O(N)

S.C. = O(N)

{Find Intersection point of Y-LL}

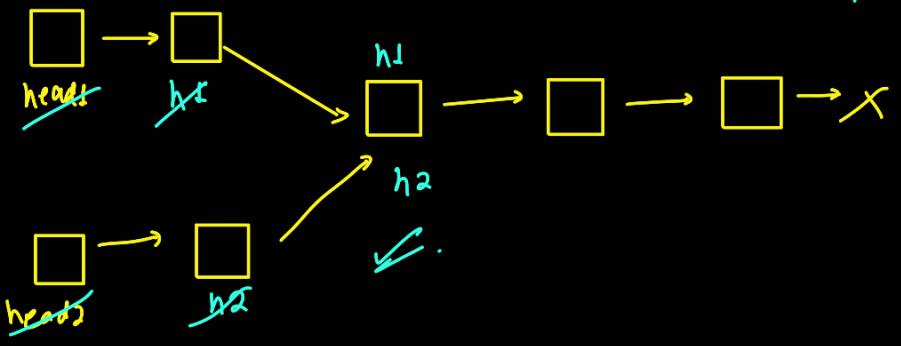


naive soln → using Hashing (Revisit after learning Hashing).

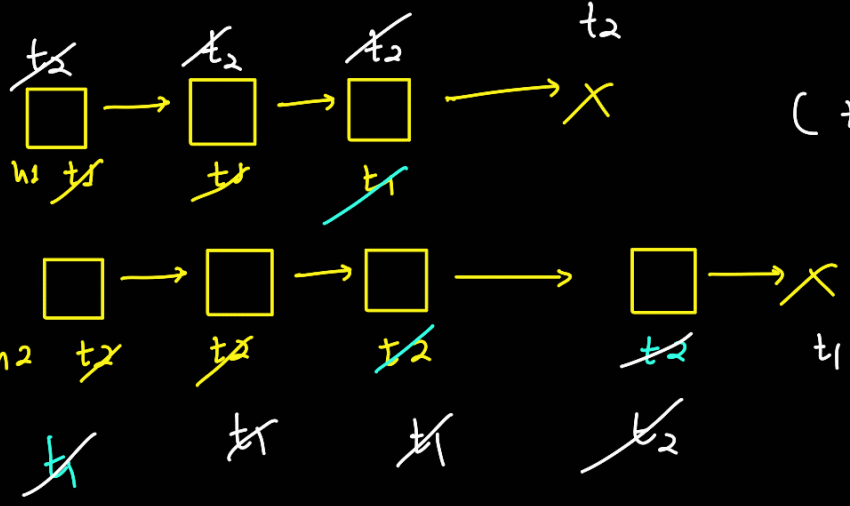
Optimal → Beautiful soln →

* Keep traversing temp 1 and temp 2.
If temp 1 reaches last → replace it with opposite head (i.e. head 2)
do same incase of temp 2.

edge cases → (1) If $L_1 = L_2$ (length). (just traverse & you will eventually find the intersecting node).



(2)



($t_1 = t_2 = null$) simultaneously, then no intersection (prev. algo is working !!).

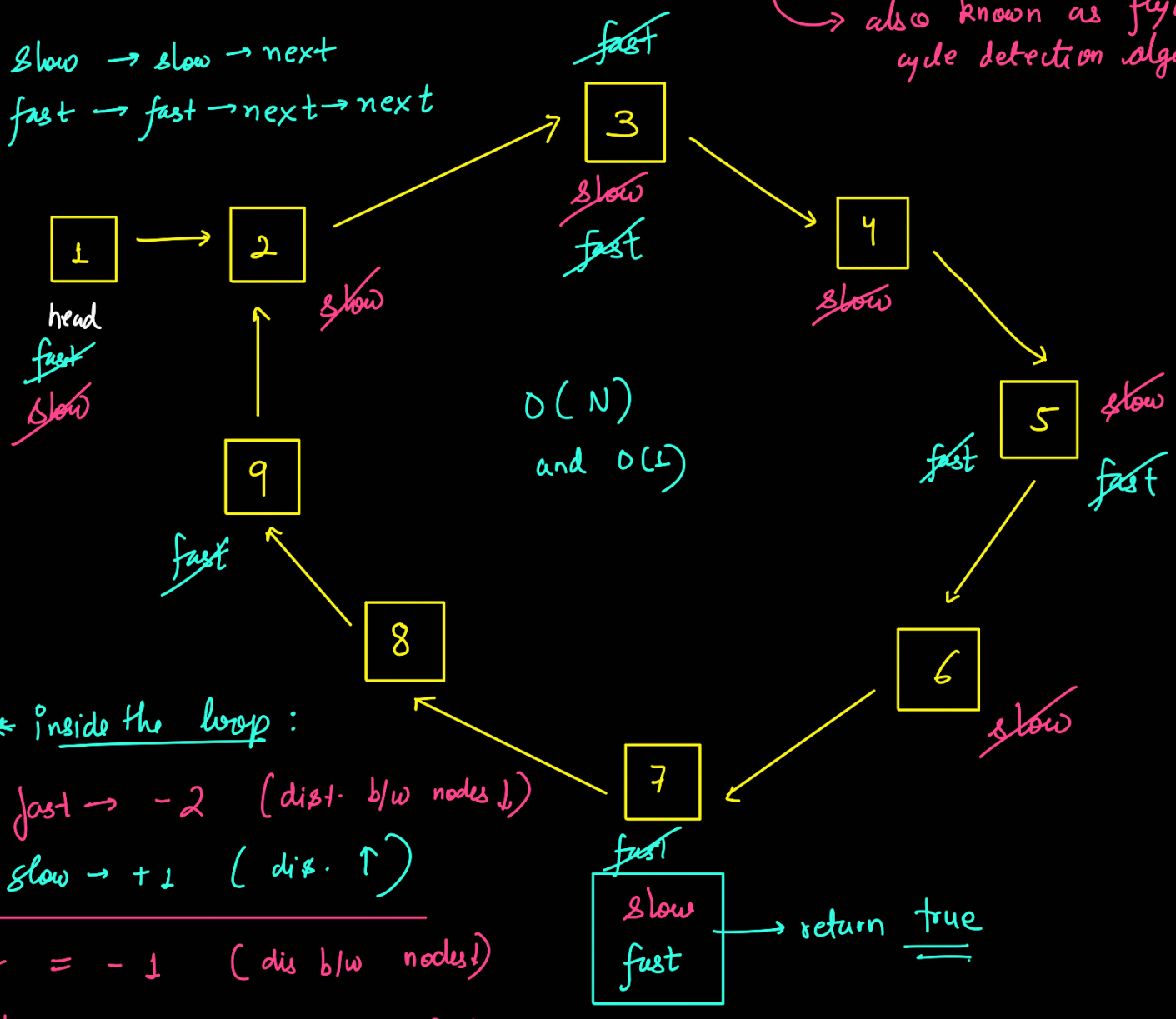
* T.C. : $O(N_1 + N_2)$
* S.C : $O(1)$

Ques → detect loop → (tortoise and hare algorithm's beautiful implementation)

↳ also known as flyod's cycle detection algo.

slow → slow → next

fast → fast → next → next



* inside the loop :

fast → -2 (dist. b/w nodes ↓)

slow → +1 (dis. ↑)

net = -1 (dis b/w nodes!)

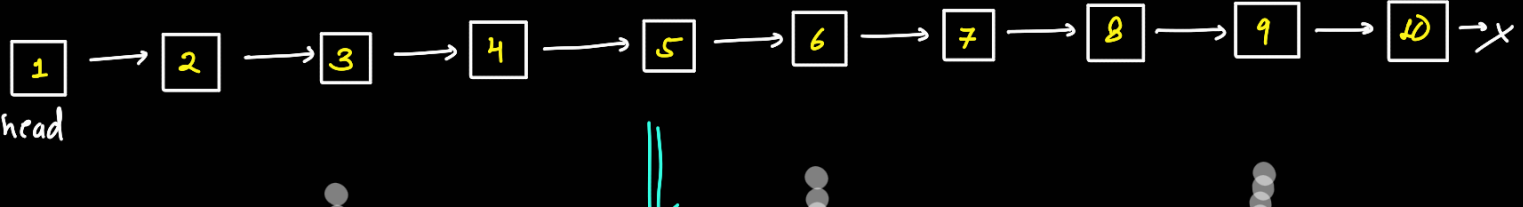
↳ eventually slow and fast are bound to intersect if there exists a loop.

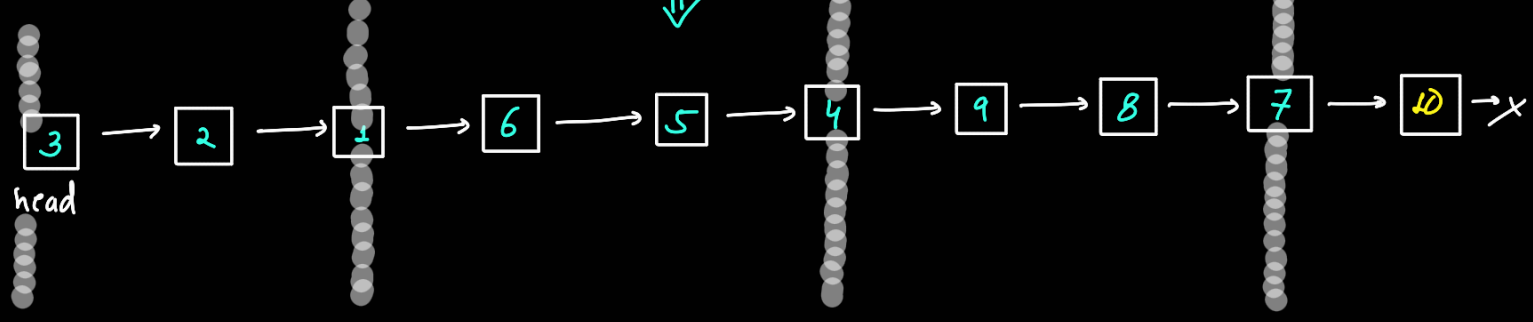
Ques → Design a Browser History :

↳ just design the constructors and simple implementation of traversal in DLL and deletion.

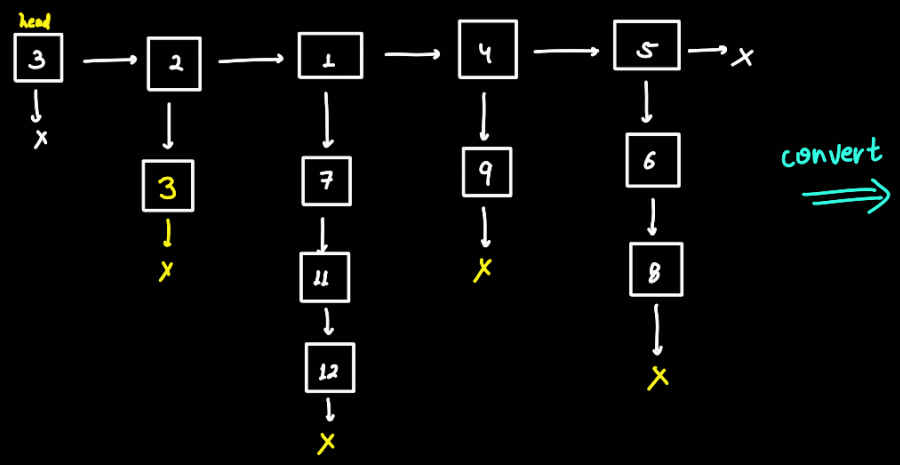
Reverse nodes in K-Groups :

K = 3

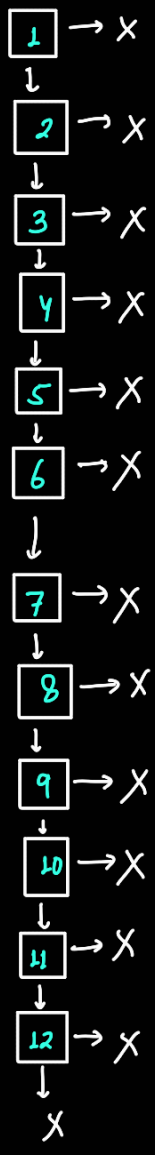




Flatten a L-L :



convert ⟹



temp , temp → next

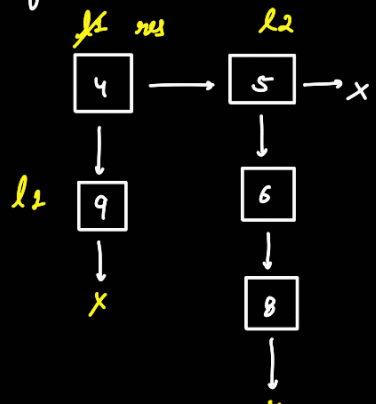
temp
|
child

brute force → store all el. in array
↓
sort it
↓
convert to LL.

(space 2 × O(N × m)

(time 2×O(N×m) + N×m log (N×m)

If 2 lists →

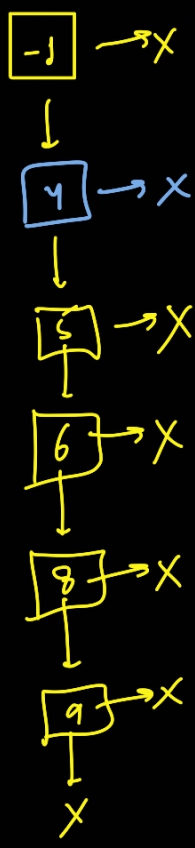


↗ returns updated head after merging two L.L.

```
merge2lists ( list1, list2) {
    dummynode = new node (-1);
    res = dummynode;
while ( list1 and list1) {
    if ( list1 → val < list2 → val)
    { res → child = list1;
```

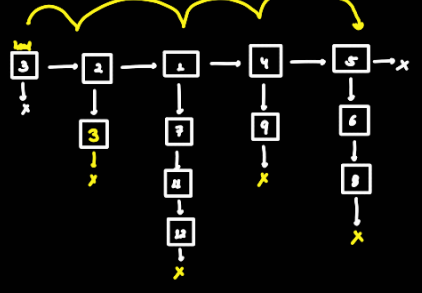
```
        res = list1
        list1 = list1 → child; }
    else { res → child = list2;
        res = list2;
        list2 = list2 → child; }
    res → next = NULL;
    }
if (list1)  res → child = list1;
else        res → child = list2;
return dummynode → child;
}
```

$T.C. = O(N_1 + N_2)$




```
node* flatten (head) {
    if (!head or !head→next) return head;
    merged head. = flatten (head→next);
    return merge 2 lists (head, mergedhead);
}
```

dry!



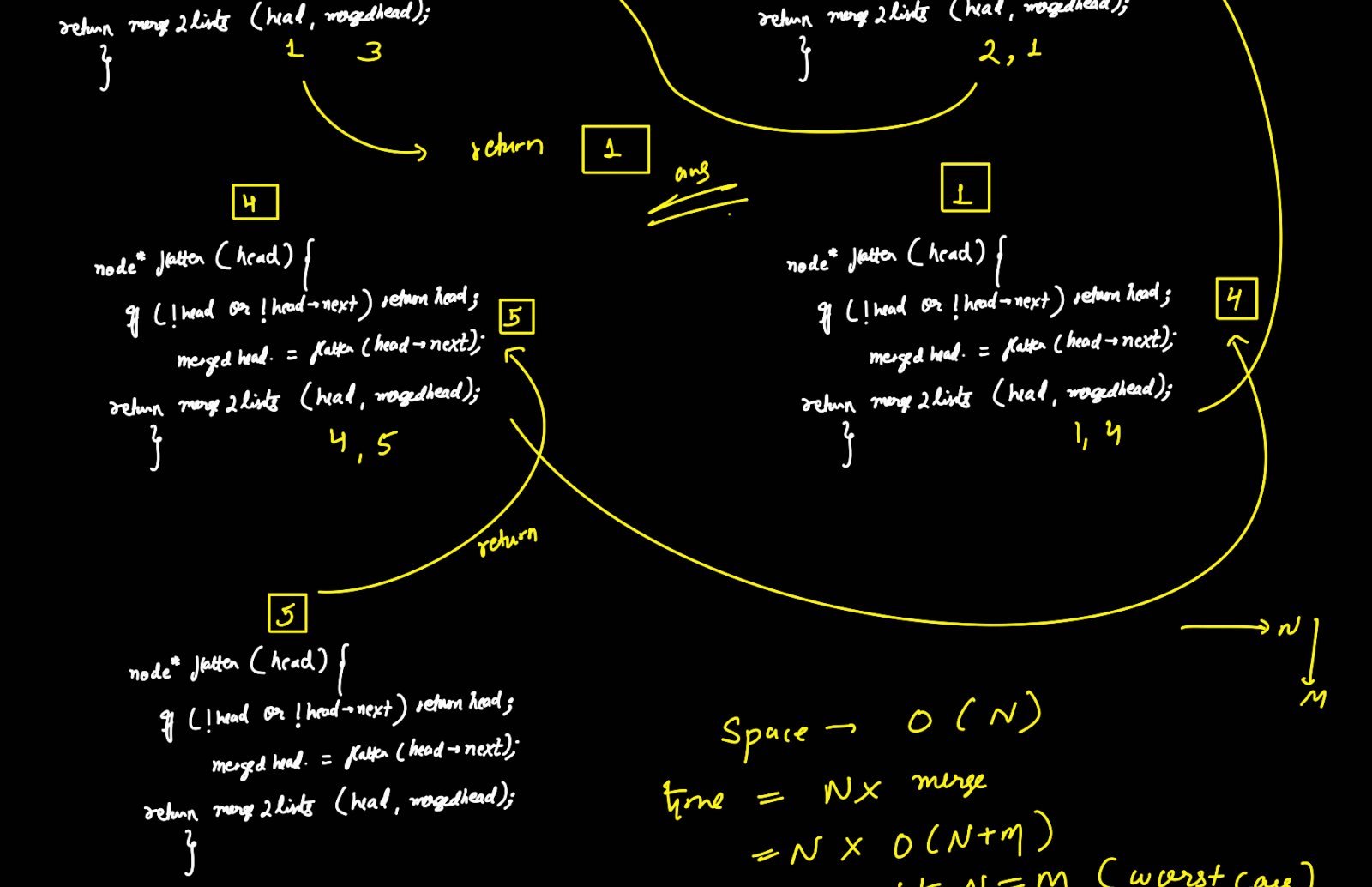
3

```
node* flatten (head) {
    if (!head or !head→next) return head;
    merged head. = flatten (head→next);   1
```

2

```
node* flatten (head) {
    if (!head or !head→next) return head;
    merged head. = flatten (head→next);
```

1



Space → O(N)

Time = N × merge

= N × O(N+M)

let N = M (worst case)

TC = N × O(2M)

= O(2NM)