LINKED LISTS

Linked List :-

Linked List is a linear data structure, which consists of a group of nodes in a sequence (OR) Linked List in which we store data in linear form.

Arrays also store data in linear form.

But the structure of array and linked list are different.

We have to define size of Array before whereas LinkedList is dynamic, i.e., we don't have to define it's size.

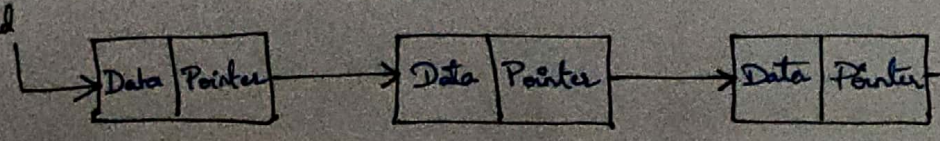
Applications :-

1. Implementing HashMaps, File System and Adjacency Lists.
2. Dynamic Memory allocation: We use linked lists of free blocks.
3. Performing arithmetic operations on long integers.
4. Maintaining a directory of names.
5. Previous - n - next page in browser.
6. Image Viewer.
7. Music Player.

Structure of Linked List :-

A linked list is formed by nodes that are linked together like a chain. Each node holds data, along with a pointer to the next node in the list.

The Singly Linked List (SLL) is the type of linked list where each node has only one pointer that stores the reference to the next value.



Advantages

1. Dynamic Nature
2. Optimal insertion & deletion.
3. Stacks & Queues can be easily implemented.
4. No memory wastage

Disadvantages

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

Node Structure :-

① Value: The data element stored in the node.
② Next: A reference to the next node in the list.

Why Linked Lists ?

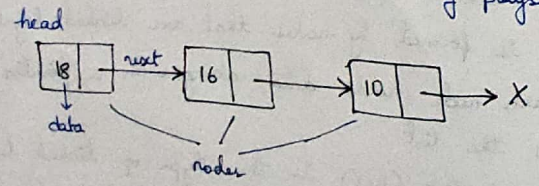
① Dynamic Size: Easily grows and shrinks in size.
② Efficient Insertions/Deletions: Quick at the beginning and middle of the list.

Types of Linked Lists :-

① Singly Linked List :-

Each node points to the next node.

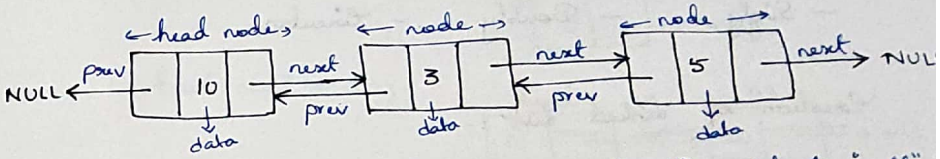
Eg: A music playlist where each song plays after the previous one.



② Doubly Linked List :-

Each node has pointers to both the next and the previous nodes.

Eg: A web browser's history, enabling forward and backward navigation.



First node's "prev" & last node's "next" point to "null".

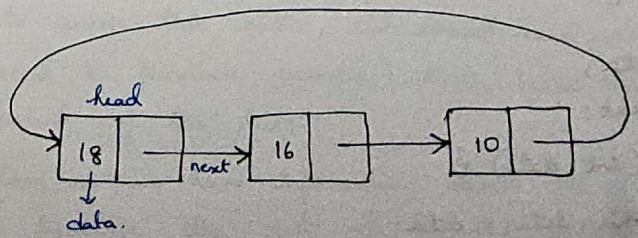
each node has 3 parameters :

→ prev
→ data
→ next

③ Circular Linked List :-

The last node in the list points back to the first node.

Eg: A multiplayer board game where play returns to the first player after the last.



④ Circular Doubly Linked List :-

Each node has pointers to both previous and next nodes, last node next points to first node & first node previous points to last node.

In practice, we only use 3 types of Linked List.
- Singly - Doubly - Circular.

Creation of Linked List :-

① Class Node.
② Class Linked List.

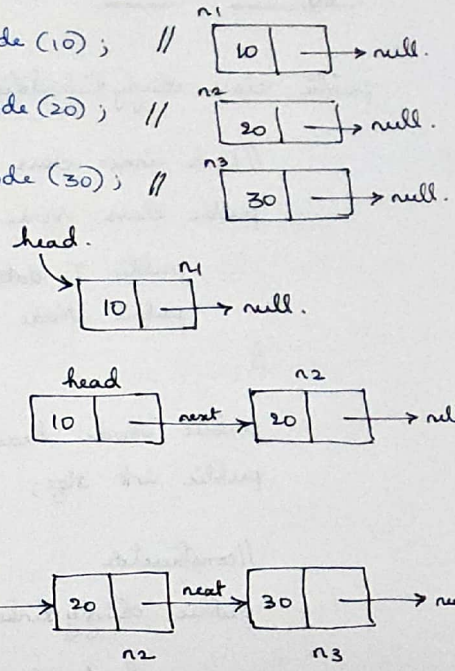
Class Node :-

```
public class Node<T> {
    public T data;          // data to store
    public Node nextNode;   // pointer to next node
}
```

Example :-

```
class Node {
    int data;
    int next;
    Node (int data) {
        this.data = data;
    }
}
```

```
public static void main(String[] args) {
    Node n1 = new Node(10);   //
    Node n2 = new Node(20);   //
    Node n3 = new Node(30);   //
    Node head = n1;
    head.next = n2;
    n2.next = n3;
    // print
}
```



Class Linked List :-

Singly Linked List: is made up of nodes that are linked together like a chain.

Now to access this chain, we would need a pointer ("head" as discussed in previous example) that keeps track of the first element of the list.

As long as we have information about the first element, we can traverse the rest of the list without worrying about memorizing their storage locations.

Singly Linked List class :-

```
public class SinglyLinkedList<T> {

    //Node inner class for SLL
    public class Node {

        public T data;
        public Node nextNode;

    }

    public Node headNode; //head node of linked list
    public int size; //size of list.

    //constructor
    public SinglyLinkedList() {

        headNode = null;
        size = 0;
    }
}
```

Linked List in java :-

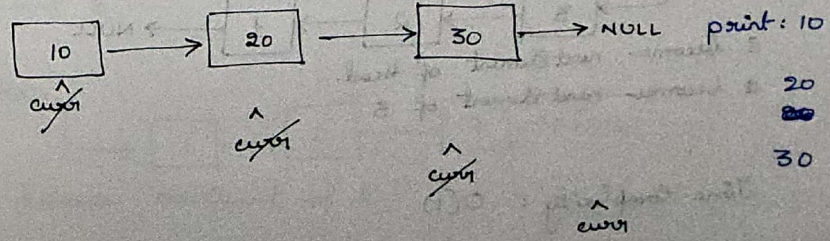
java offers a Linked List class in its standard library.

```
import java.util.LinkedList;
class Main {
public static void main(String[] args) {

    LinkedList<String> list = new LinkedList<>();
    list.add("Node1");
    list.add("Node2");
    list.add("Node3");
    System.out.println(list);
}
```

Traverse in a Linked List :-

```
class Node<T> {

    T data;
    Node next;
    Node(T data) {
        this.data = data;
    }
}
void main() {

    traverse(head);
}
void traverse(Node head) {

    Node curr = head;
    while (curr != null) {

        print(curr.data);
        curr = curr.next;
    }
}
```

Linked List :-

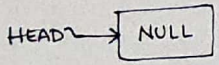


Operations in Singly Linked List :-

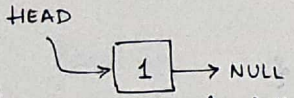
1. Insert

* Insertion at Head :-

Inserting at Head means inserting the first element in the list.

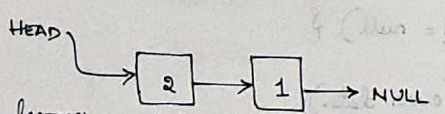


Insert 1



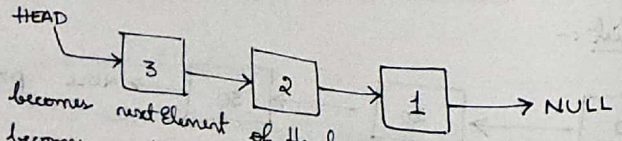
1 becomes nextElement of Head.

Insert 2



2 becomes nextElement of Head.
1 becomes nextElement of 2

Insert 3



3 becomes nextElement of Head.
2 becomes nextElement of 3

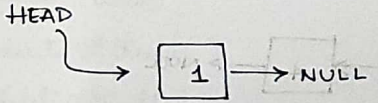
Time Complexity : O(1)

* Insert at End :-

Insert a new element at the last element of the list. The original 'tail' element of the list has a 'nextElement' pointer that points to NULL.

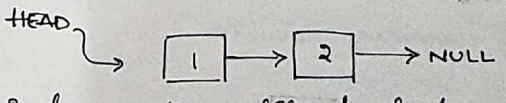
To insert a new 'tail' node, we have to point the 'nextElement' pointer of the previous 'tail' node to the new 'tail' node, allowing the 'nextElement' of the next 'tail' to now point to 'NULL'.

Insert 1



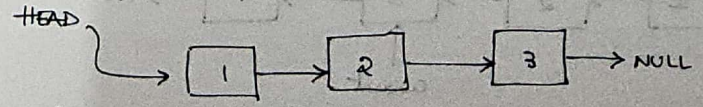
1 becomes nextElement of Head.

Insert 2



2 becomes the nextElement of 1.

Insert 3



3 becomes nextElement of 2.

Traverse the list to the last node and add the new node after it.

Time Complexity: O(n).

* Insert After :-

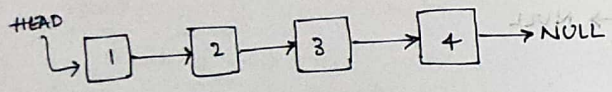
We specify the node after which we want to insert the new node.

To insert this node, we follow these steps:

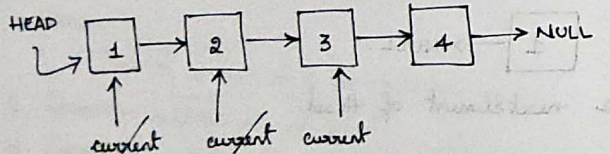
① We traverse the linked list to look for 'n'.

② As soon as we find it, we assign n's 'nextElement' to the new node's 'nextElement'.

③ Then we point n's 'nextElement' to the new node.

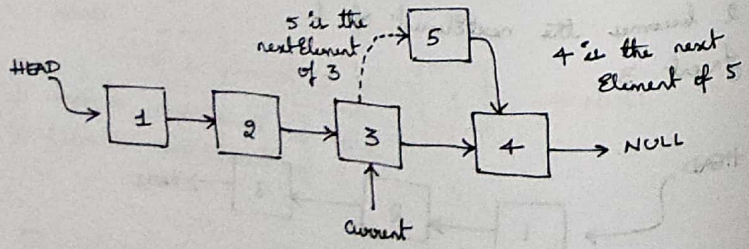


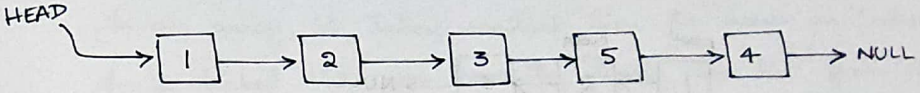
Insert 5 after 3



3 not found 3 not found 3 found.

Create node 5





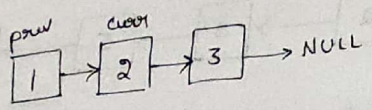
Traverse the list to the desired node and insert the new node after it.

Time Complexity: O(n).

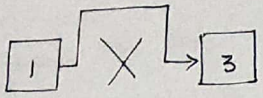
Basic Linked List Operations :-

①	insertAtEnd (data)	O(n)
②	insertAtHead (data)	O(1)
③	insertAfter (prev, data)	O(n)
④	delete (data)	O(n)
⑤	deleteAtHead()	O(1)
⑥	deleteAtEnd()	O(n)
⑦	search (data)	O(n)
⑧	isEmpty() — returns 'true' if the linked list is empty, otherwise returns 'false'.	O(1)

Delete a node in Linked



delete val = 2



prev.next = curr.next

Linked List vs Arrays :-

- Memory Allocation.
- Insertion and Deletion.
- Searching.

* Memory Allocation :-

Arrays instantiate a whole block of memory, e.g., array[1000] gets space to store 1000 elements at the start even if it doesn't contain any element yet.

On the other hand, a linked list only instantiates the portion of memory it uses.

* Insertion and Deletion :-

In linked list, insertion and deletion at head happens in O(1) time.

In arrays, insertion and deletion at head happens in O(n) time because you have to shift the array elements left or right after that operation.

* Searching :-

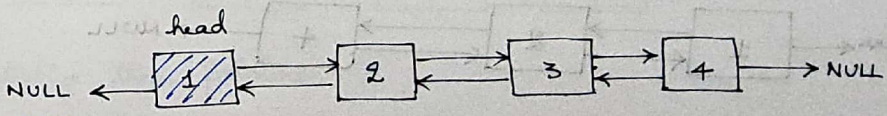
In an array, it takes constant time to access an index.

In a linked list, you have to iterate the list from the start until you find the node with the correct value

Operation	Linked List	Array
Access	O(n)	O(1)
Insert (at head)	O(1)	O(n)
Delete (at head)	O(1)	O(n)
Insert (at tail)	O(n)	O(n)
Delete (at tail)	O(n)	O(n).

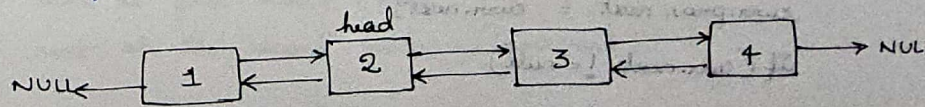
* Deletion on Doubly Linked List :-

In doubly linked list, we do not need to keep track of the previous element while searching.

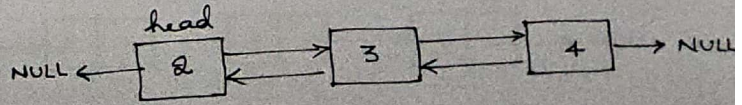


Delete 1

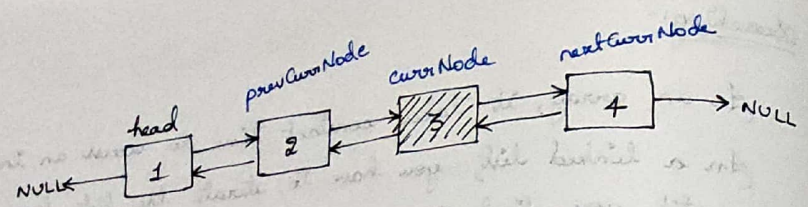
head points to the second node.



head.prev points to NULL



Eg, case when head node needs to be deleted.

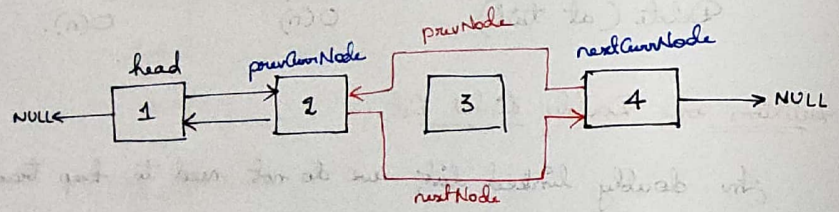


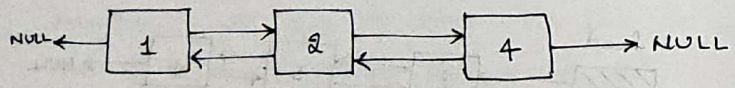
Delete 3

* currNode: Node to be deleted. (curr)
* prevCurrNode: prevNode of currNode. (curr.prev)
* nextCurrNode: nextNode of currNode. (curr.next)

To delete the currNode successfully,

① Set the nextNode of prevCurrNode to be nextCurrNode.
② Set the prevNode of nextCurrNode to be prevCurrNode.



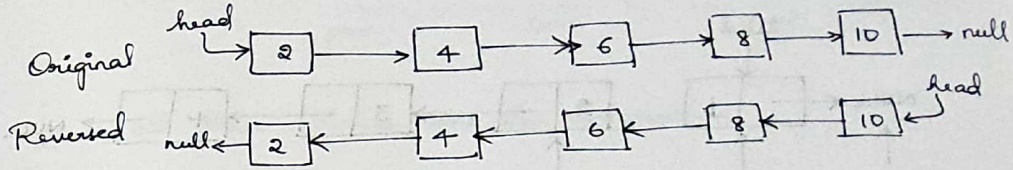


Code:-

```
curr.prev.next = curr.next;
if (curr.next != null)
    curr.next.prev = curr.prev;
```

PATTERN: IN-PLACE REVERSAL OF A LINKED LIST

Given Singly Linked List. Return reversed Linked List.

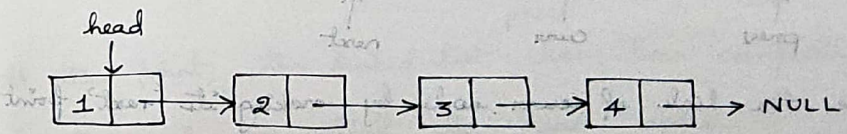


We iterate over the linked list while keeping track of three nodes:

the current node.
the next node.
the previous node.

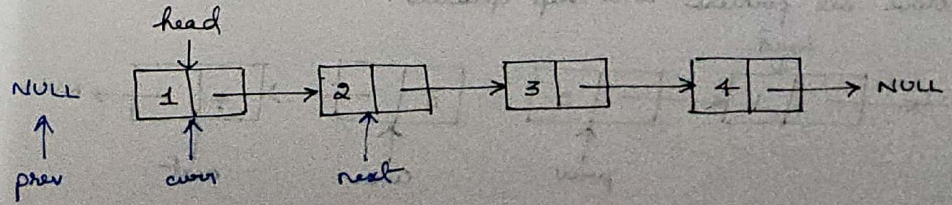
TC: O(n) SC: O(1).

Example:-

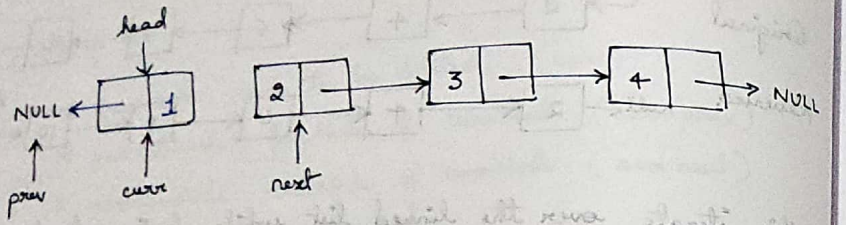


Initialize three pointers:

→ prev at NULL
→ curr at first node. (head)
→ next at the second node. (head.next).



Reverse the link of curr node by making its 'next' point to 'prev'.

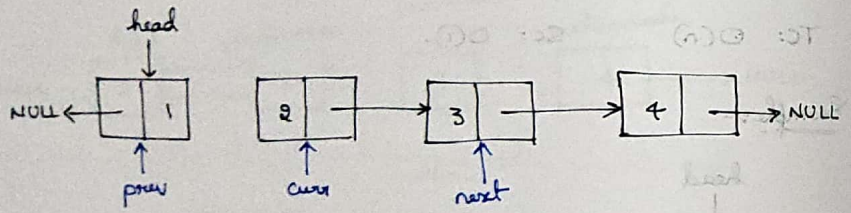


Move all pointers one step forward.

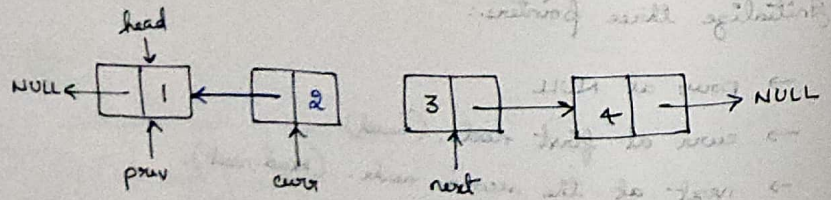
prev = curr

curr = next

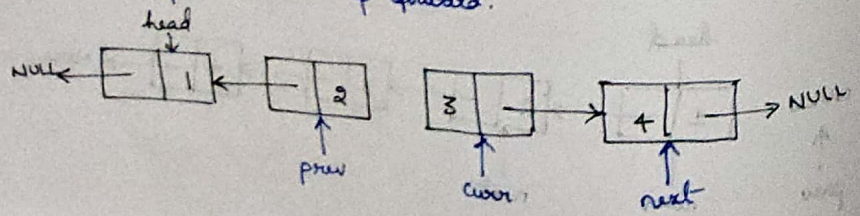
next = next.next



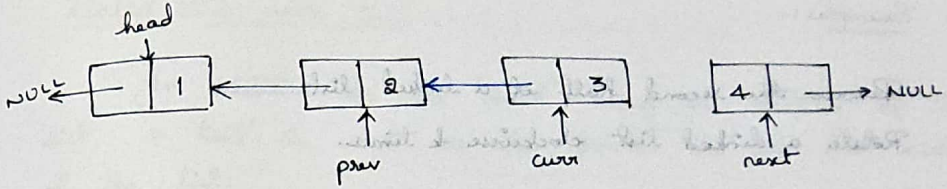
Reverse the link of curr node by making its 'next' point to 'prev'.



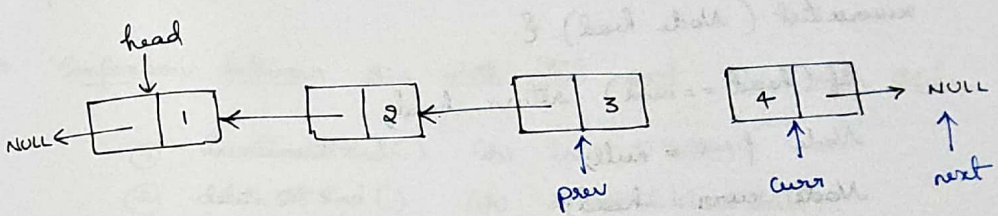
Move all pointers one step forward.



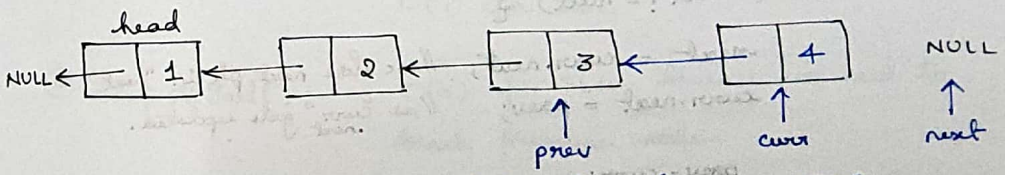
Reverse the link of "curr" node.



Move all three pointers one step forward.



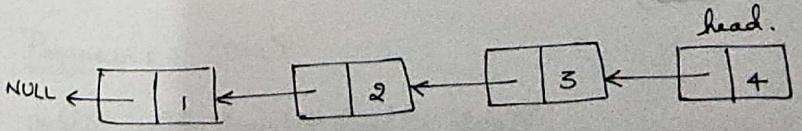
Reverse the link of "curr" node.



At this point, the linked list has been completely reversed.

In the end, we adjust the 'head' node.

head = curr.



Examples:-

Reverse the second half of a linked list.

Rotate a linked list clockwise k times.

Code:-

```
reverseList (Node head) {
    if (head == null) return head;
    Node prev = null;
    Node curr = head;
    Node next = null;
    while (curr != null) {
        next = curr.next; // we store next ptr in "next"
        curr.next = prev; // as "curr.next" gets updated.
        prev = curr;
        curr = next;
    }
    head = prev;
    return head;
}
```

Linked List with Tail :-

In addition to the 'head' being the starting of the list, a 'tail' is used as the pointer to the last node of the list.

SLL and DLL can be implemented using a 'tail pointer'

* Comparison between SLL with Tail and DLL with Tail :-

① insertAtEnd() (or) insertAtTail()

② deleteAtEnd() (or) deleteAtTail().

① insertAtTail() :- O(1) for SLL

O(1) for DLL

② deleteAtTail() :- O(n) for SLL because we need to track previous node.

O(1) for DLL because DLL already has a pointer to previous node.

So the advantage of having a "tailNode" is for the DLL to delete node at the end.

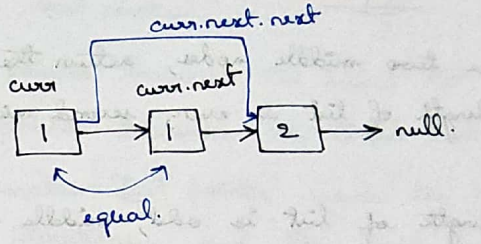
* Program :- (DLL)

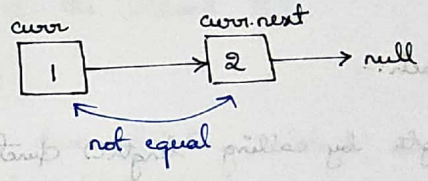
```
public void insertAtHead(T data){
    Node newNode = new Node();
    newNode.data = data;
    newNode.next = this.head;
    newNode.prev = null;
```

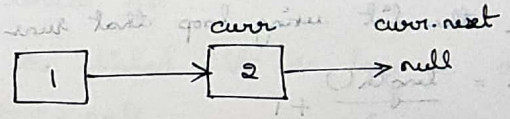
```
        if(!isEmpty()) {
            head.prev = newNode;
        } else {
            tail = newNode;
        }
        this.head = newNode;
        size++;
    }

public void insertAtEnd(T data) {
    if(isEmpty()) {
        insertAtHead(data);
        return;
    }
    Node newNode = new Node();
    newNode.data = data;
    newNode.next = null;
    newNode.prev = tail;
    tail.next = newNode;
    tail = newNode;
    size++;
}
```

Problem: Remove Duplicates from Sorted List LC 83.

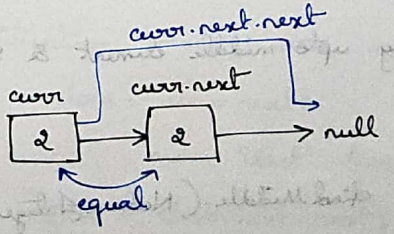
*

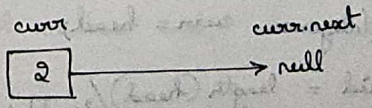






*





Middle Of Linked List (LC 876)

Two pointers

If there are two middle nodes, return the second middle node. When length of list is even, second middle node occurs at length/2.

When length of list is odd, middle node occurs at length/2 +1.

* Brute force approach:-

↳ calculate length by calling length() function.

↳ Traverse the list using loop that runs mid-1 times where mid = $\frac{\text{length}()}{2} + 1$

⇒ Traverse only upto middle element & return it.

Code:-

```
Node<Integer> findMiddle (Node<Integer> head){
    Node<Integer> curr = head;
    int mid = length(head)/2 +1;
    for(int i=0; i<mid-1; i++){
        curr = curr.next;
    }
    return curr;
}
```

* Two pointers approach :- (Fast & Slow pointers).

* slow, fast pointers.
* 'slow' pointer traverses the linked list one step at a time.
* 'fast' pointer traverses the linked list two steps at a time.
* This makes 'fast' pointer reach the end of the linked list in $\frac{n}{2}$ iterations, and the 'mid' pointer reaches the middle of the linked list.

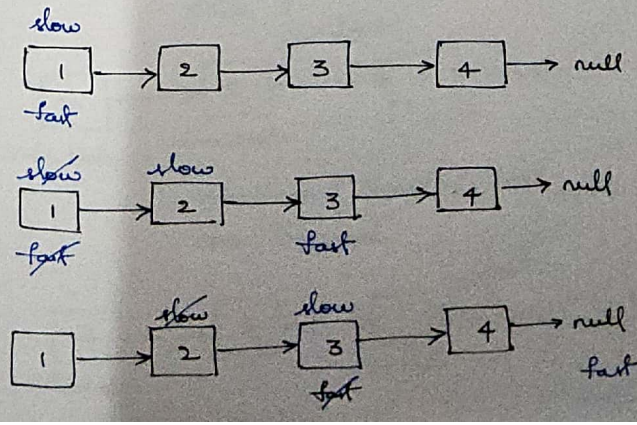
Algorithm :-

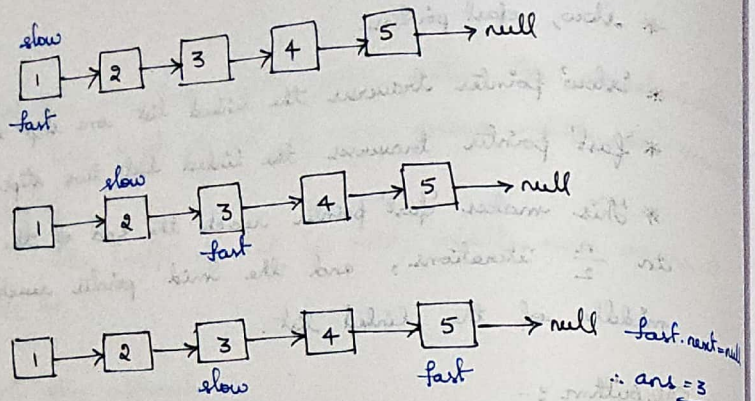
1. Initialize two pointers, 'slow' and 'fast', to the head of the linked list.
2. Traverse the linked list as long as fast != null & fast.next != null.
3. At each iteration, move 'slow' one step ahead.

   slow = slow.next;

   move 'fast' two steps ahead.

   fast = fast.next.next;
4. When fast reaches end, slow reaches mid. Return slow.





Remove duplicates from Unsorted Linked List

1) i/p: 12 → 11 → 12 → 21 → 41 → 43 → 21

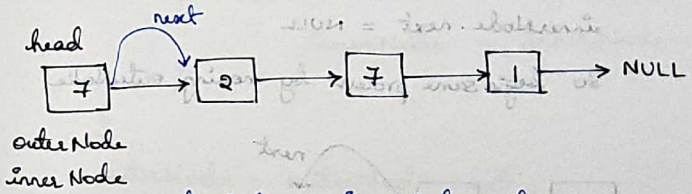
   o/p: 12 → 11 → 21 → 41 → 43

2) i/p: 1 → 2 → 3 → 2 → 4

   o/p: 1 → 2 → 3 → 4

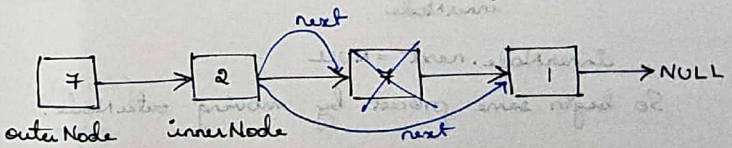
Approach :- Using nested loops – $O(n^2)$

innerNode & outerNode.



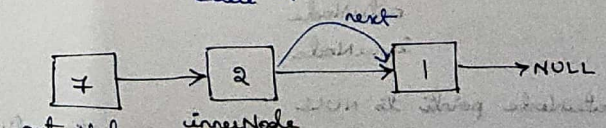
outerNode ≠ innerNode.next

move innerNode



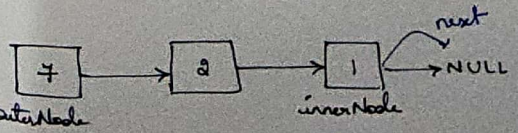
outerNode = innerNode.next

delete 7.

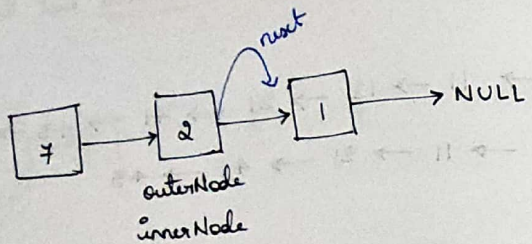


outerNode ≠ innerNode.next

move innerNode

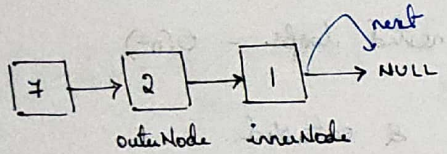


innerNode.next = NULL, so begin same proan by moving outerNode one step.



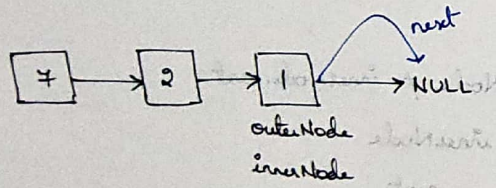
outerNode ≠ innerNode.

So move innerNode.



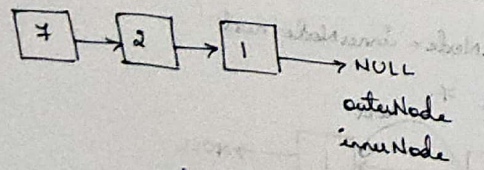
innerNode.next = NULL

So begin same process by moving outerNode



innerNode.next = NULL

So begin same process by moving outerNode.



outerNode points to NULL indicating that we have iterated over the entire list.

Program :-

```
Node outerNode = head;
while (outerNode != null) {
    Node innerNode = outerNode;
    while (innerNode.next != null) {
        if (outerNode.data == innerNode.next.data) {
            innerNode.next = innerNode.next.next;
        } else {
            innerNode = innerNode.next;
        }
    }
    outerNode = outerNode.next;
}
return head;
```

TC: $O(n^2)$

SC: $O(1)$

Approach using HashSet :-

We can use hash set to keep track of the values that have already been seen.

As we traverse the linked list, for each node, we check if its value is already in the hash set.

If the value is found, it means it's a duplicate, so we remove that node by adjusting the pointers of the previous node to skip the current one.

If the value is not found, we add it to the hash set and move to the next node.

Program :-

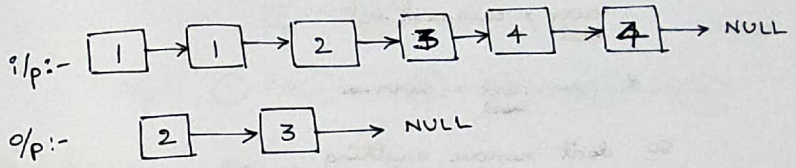
```
HashSet<Integer> set = new HashSet<>();
Node curr = head;
Node prev = null;
while (curr != null) {
    if (set.contains(curr.data)) {
        prev.next = curr.next;
        curr = curr.next;
    } else {
        hashSet.add(curr.data);
        prev = curr;
        curr = curr.next;
    }
}
return head;
```

T.C : O(n)

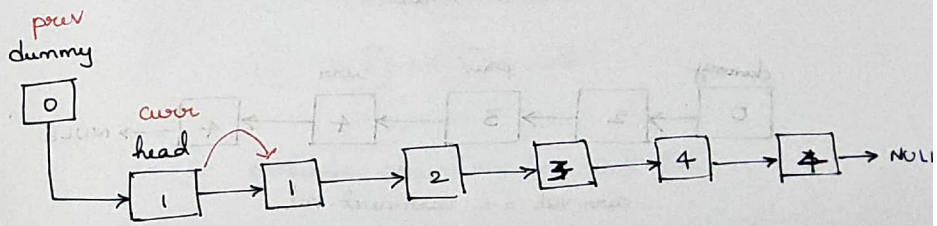
S.C : O(n)

Remove Duplicates from sorted Linked List 2

(LC 82)

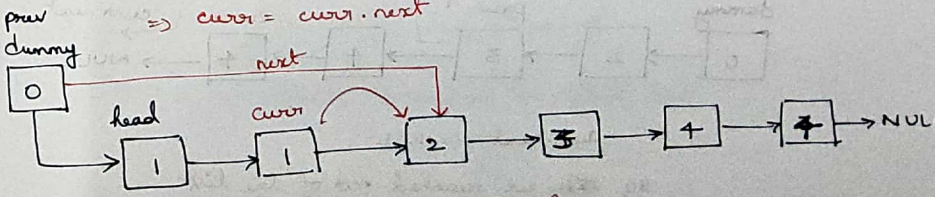


Approach :-



curr.val == curr.next.val

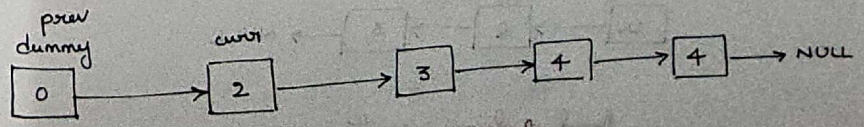
=> curr = curr.next



curr.val ≠ curr.next.val

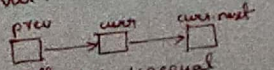
=> prev.next = curr.next

curr = curr.next



curr.val ≠ curr.next.val

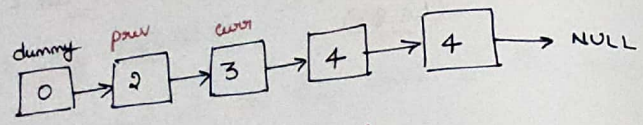
& prev.next = curr =>



all are unequal

so don't remove anything

prev = prev.next, curr = curr.next



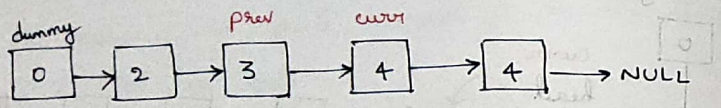
curr ≠ curr.next.val

& prev.next == curr

so, don't remove anything.

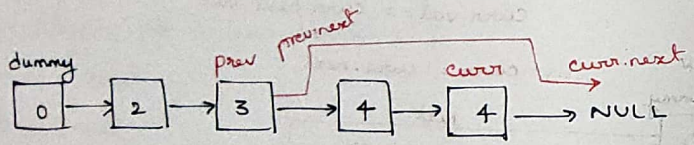
prev = prev.next

curr = curr.next



curr.val == curr.next.val

so curr = curr.next

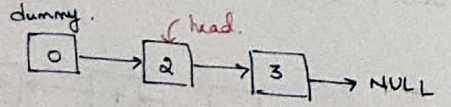


curr.next == NULL

so, we reached end of the list.

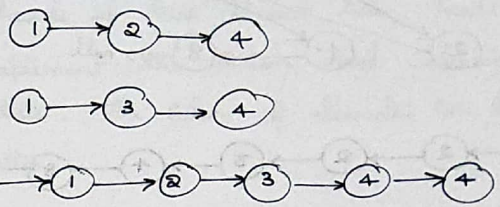
set prev.next = curr.next

& terminate.

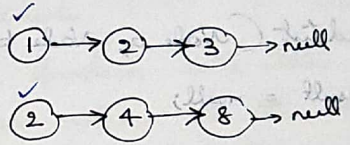


∴ head = dummy.next = 2

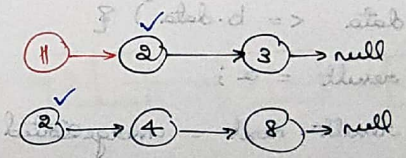
Merge two sorted list LC21



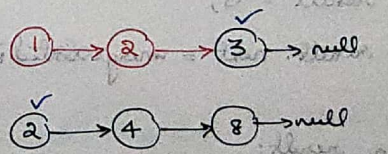
Approach 1 :- RECURSIVE MERGE :-

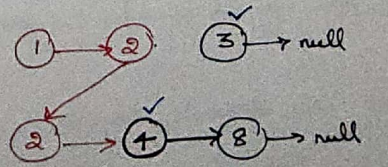


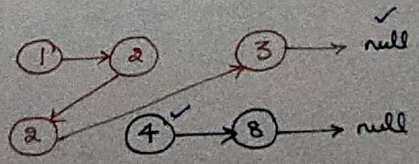
① < ② continue recursion in first list.

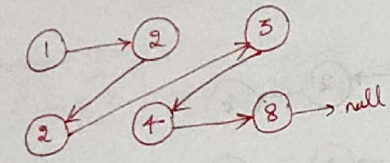


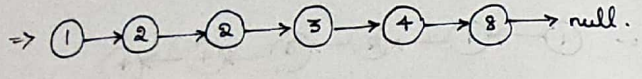
② ≤ 2 continue recursion in first list.











Program :-

```
Node mergeSortedList (Node a, Node b) {
    Node * result = null;
    if (a == null) return b;
  else if (b == null) return a;
    if (a.data <= b.data) {
        result = a;
        result.next = mergeSortedList (a.next, b);
    } else {
        result = b;
        result.next = mergeSortedList (a, b.next);
    }
    return result;
}
```

Union and Intersection of linked lists

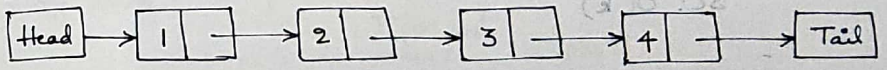
Given the heads of two linked lists, head1 and head2 as input. Implement the union and intersection functions for the linked lists. The order of elements in the output lists doesn't matter.

Union :- This function will take two linked lists as input and return a new linked list containing all the unique elements.

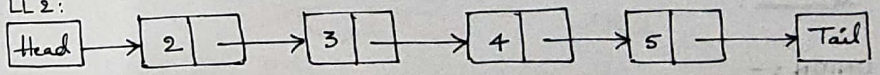
Intersection :- This function will take two linked lists as input and return all the common elements between them as a new linked list.

Example :-

LL 1 :

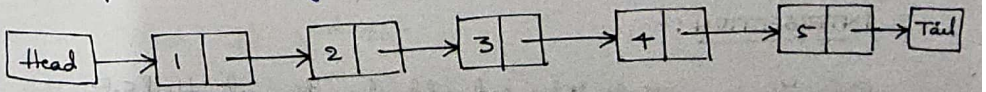


LL 2 :

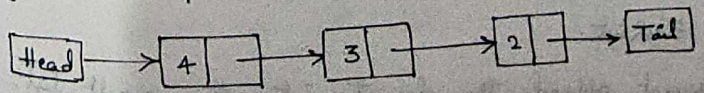


outputs :

Union of lists merged (uniquely) to form a SLL



Intersection of lists common elements are taken out from lists.



Approach 1:-

UNION:-

① Keep head of second list as next node to tail of first list. TC: O(n+m) ~~SC: O(n+m)~~

② Remove duplicates TC: O(n+m) SC: O(n+m).
using HashSet

INTERSECTION:-

① Iterate on both lists, when a common element is found, if it already exists in result, do not add to result. If not, add to result.

TC: O(n × m × k)

k - no. of common elements.

SC: O(k)

~~Approach~~ 2:-

UNION:-

① Use a HashSet to collect unique elements from both list

② Append elements from HashSet into result list.

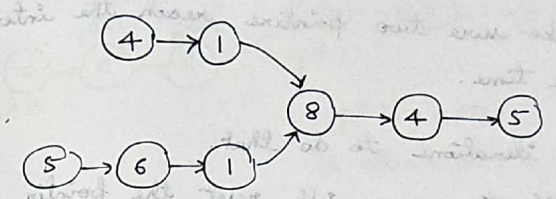
INTERSECTION:-

① Traverse first list and add elements into HashSet

② Traverse second list and check if elements are present in the HashSet

③ If present, append the elements into the result list.

Intersection of Two Linked List LC160

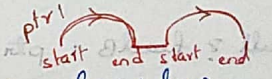


(Without using extra space, return node (8) & use O(1) space.)

Visualization:

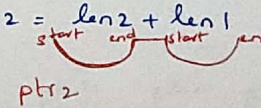
If two lists are of different lengths, we can make them equal.

If len1 & len2 are lengths, then make them equal by doing:



len1 = len1 + len2 ptr 1

len2 = len2 + len1 ptr 2.



Above figure illustrates that

when "ptr1" reaches "end" of 1st list, then it can start from "start" of 2nd list.

when "ptr2" reaches "end" of 2nd list, then it can start from "start" of 1st list.

In this way two lists appear to be equal length and can traverse parallely.

If at a parallel traversal when pointers meet returns that node.

Approach :-

We want to make sure two pointers reach the intersection node at the same time.

We can use two iterations to do that.

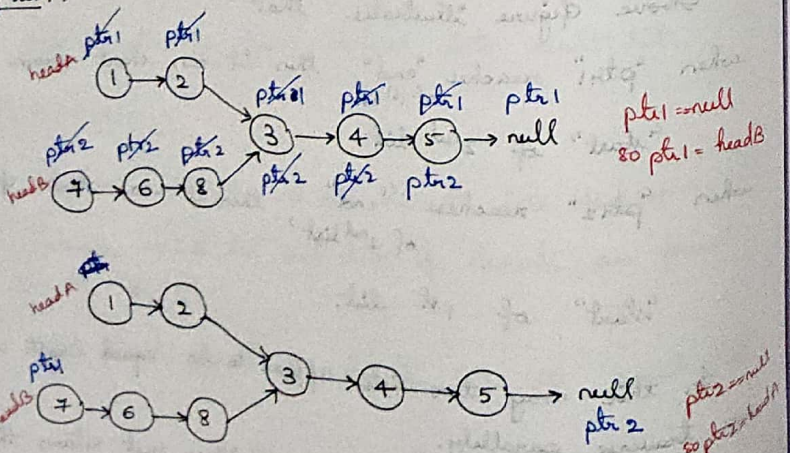
In the first iteration, we will reset the pointer of one linked list to the head of another linked list after it reaches tail node.

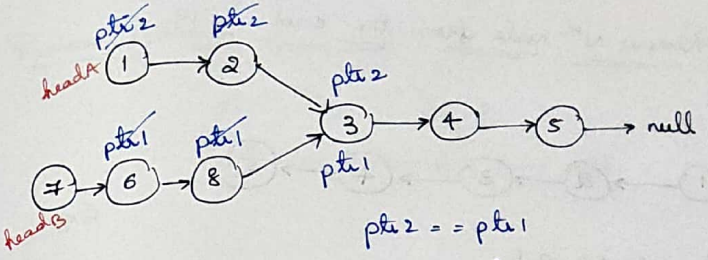
In the second iteration, we will move two pointers until they point to the same node.

```
ptr1 = headA, ptr2 = headB;
while(ptr1 != ptr2) {

    ptr1 = ptr1 == null ? headB : ptr1.next;

    ptr2 = ptr2 == null ? headA : ptr2.next;

}
return ptr1;
```

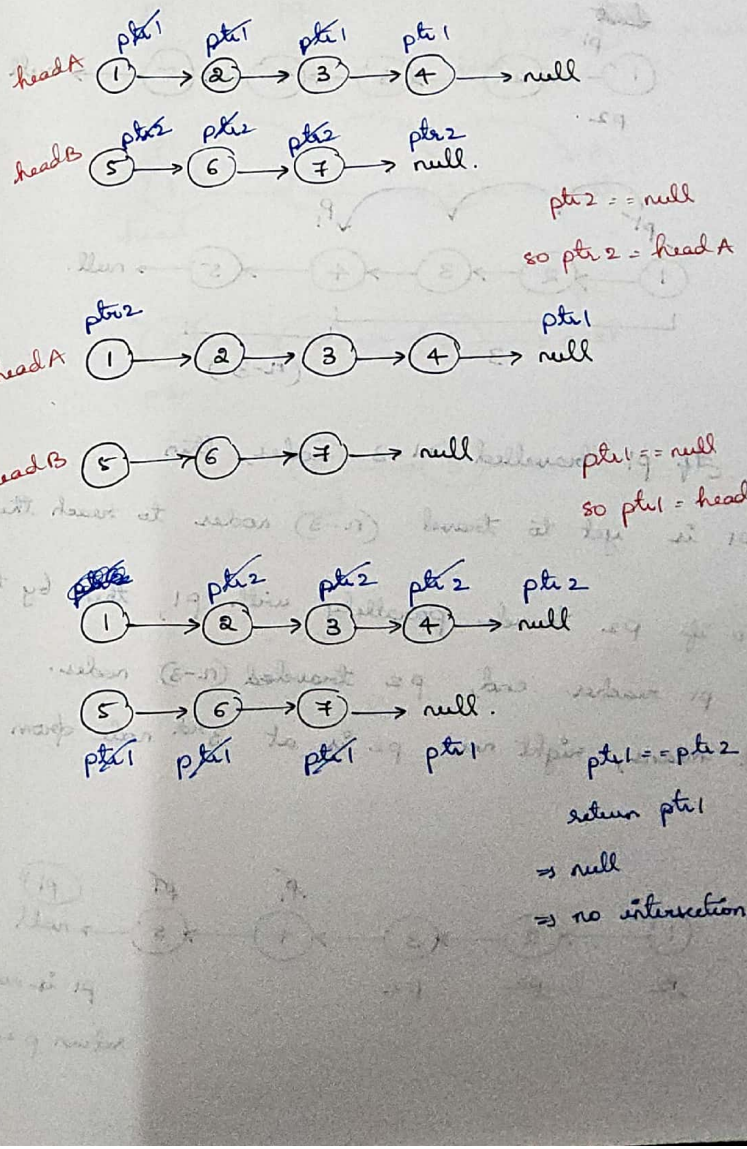
Dry run :-

(1)

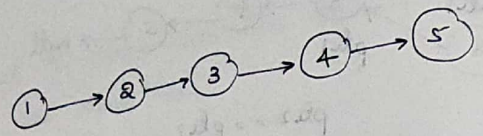




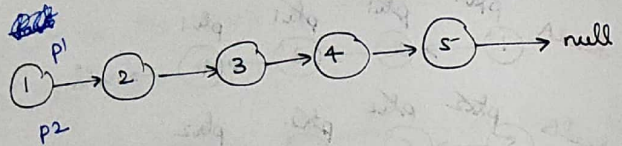
(2)



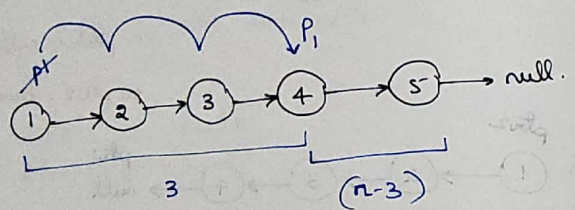
Remove Nth node from the end LC19



Find Nth node from End.



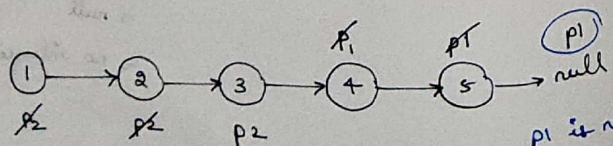
N=3.



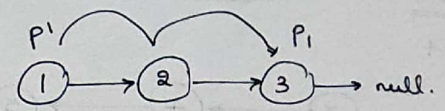
If p1 travelled N=3 nodes, then p1 is yet to travel (n-3) nodes to reach the end.

Now, So if p2 travels parallely with p1, then by the time p1 reaches end, p2 traveled (n-3) nodes.

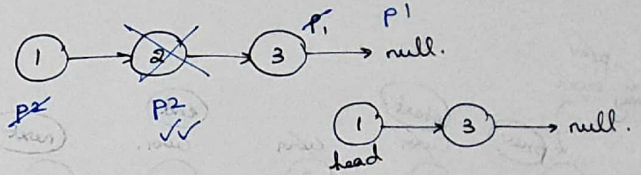
That means, right now p2 is at 3rd node from the end.



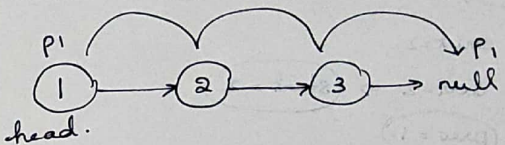
Remove Nth node from end:-

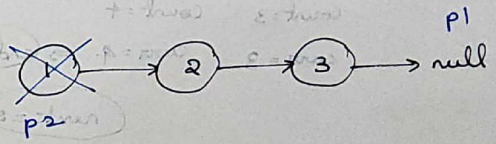


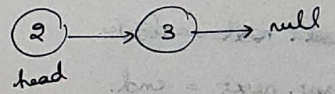
n=2



n=3

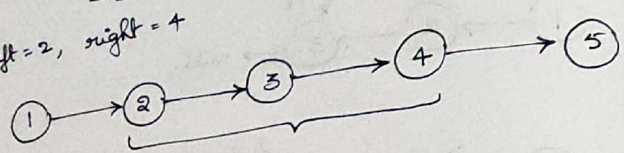


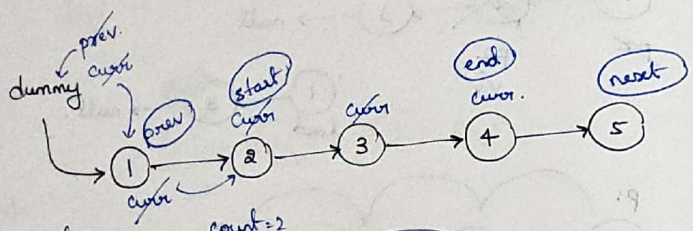




Reverse Sub List LC92.

left = 2, right = 4





count = 1
curr = 1
prev = dummy

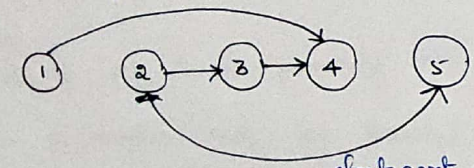
count = 2
curr = 2 ⇒ start = 2
prev = 1.

count = 3
curr = 3

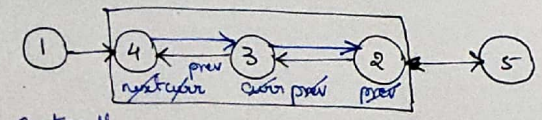
count = 4
curr = 4 ⇒ end = 4

next = 5

prev. next = end.

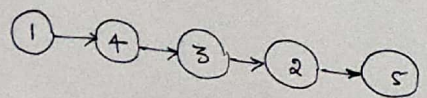


start.next = next (should be done after reverse)



next = null
curr = null

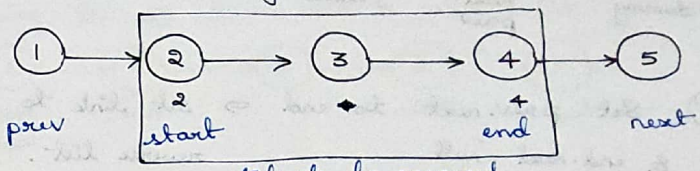
reverse (start, end).



Approach 1 :- Brute force :-

prev
next
start
end.

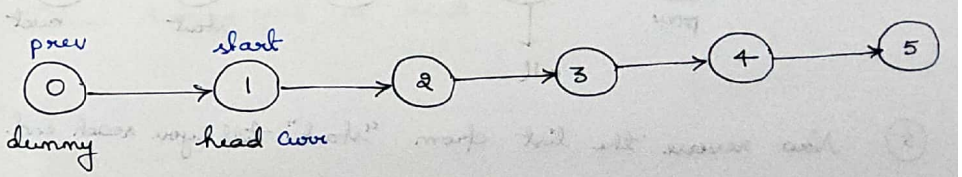
if left = 2, right = 4



list to be reversed.

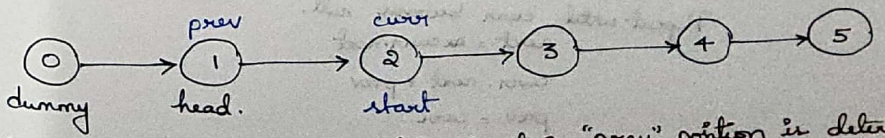
Procedure :-

(1) initialize prev = dummy.
start = head and curr = head
next = null
end = null.



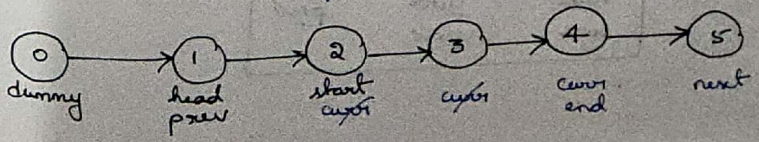
next
end

(2) move prev, curr (left - 1) no. of times ⇒ 1 time, and set start to curr.

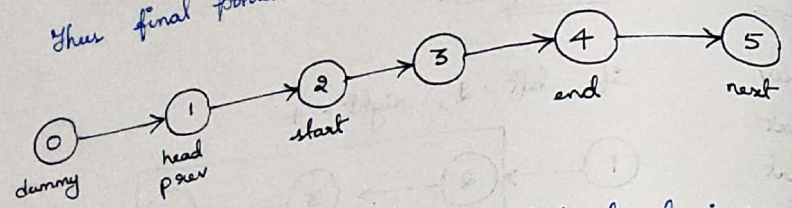


Thus "start" position is determined & "prev" position is determined.

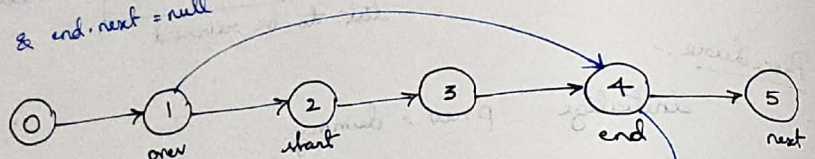
(3) continue to move 'curr' (right - left) no. of times ⇒ 2 times, and set end & next positions. to curr & curr.next positions

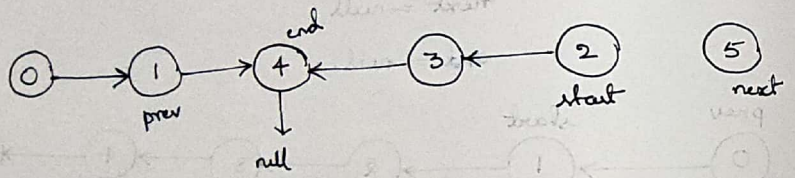


Thus final positions look like:



(4) Set prev.next to end ⇒ set link to beginning of reverse list.
& end.next = null





(5) Now reverse the list from "start" till you reach end.

Reversing:- Algorithm:-

prev = ~~null~~ start

curr = start.next

next = null.

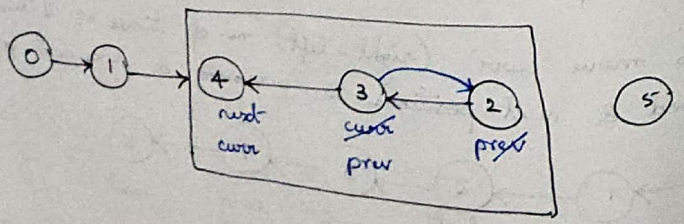
Repeat until curr becomes null.

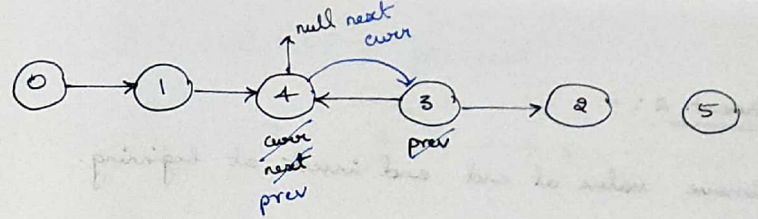
next = curr.next

curr.next = prev

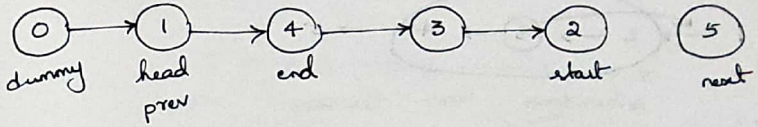
prev = curr

curr = next

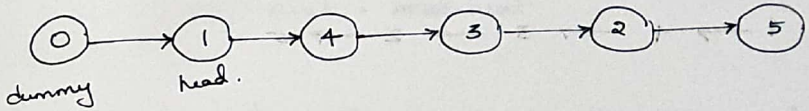




(6) After reversing, the call returns back to caller & list looks like:



(7) Now set start.next = next

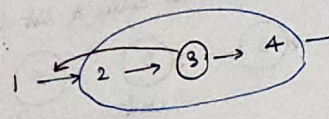


(8) Return dummy.next as head.

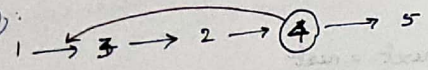
Approach 2:-

Remove value at end and insert at beginning.

Step ①:



Step ②:



1 → 4 → 3 → 2 → 5

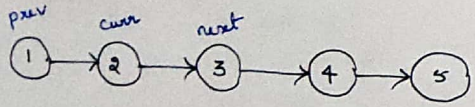
Explanation:-

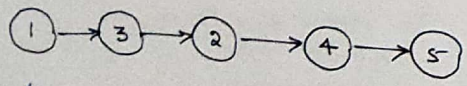
Let's have 3 pointers

prev = previous node of the reversable linked list

curr = first node of the reversable LL

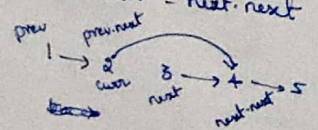
next = next node of node to be inserted.



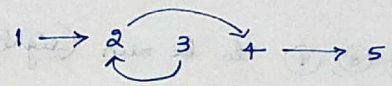


If we observe in the figure, to insert next between prev & prev.next, we need to do following.

① curr.next = next.next



② next.next = prev.next



③ prev.next = next

1 → 3 → 2 → 4 → 5

prev prev.next curr curr.next

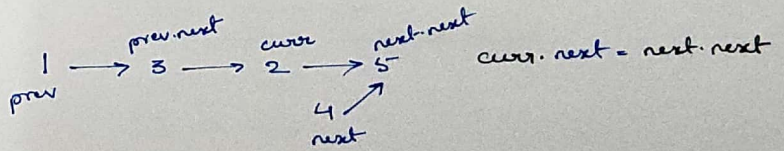
④ Now node to be inserted is 'next' node is ④.

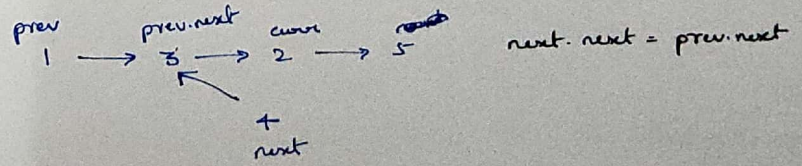
next = curr.next

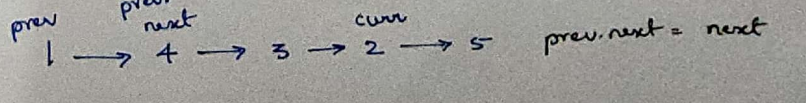
1 → 3 → 2 → 4 → 5

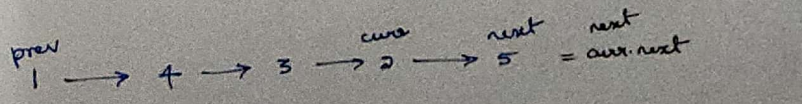
prev prev.next curr next next.next.

Repeat ①, ②, ③, ④ steps again. (Insert next btw prev & prev.next then update next = next.next.)







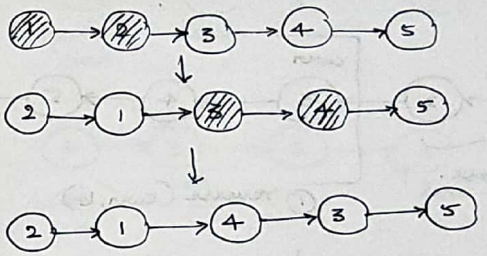


Thus,
Steps ①, ②, ③, ④ to be run (right - left) = 2 times.

Reverse Nodes in k-group (LC-25).

reverse nodes of list 'k' at a time and return the modified list.

Eg:- k = 2



Approach 1 :-

curr = head.

k = 2 times curr.next.

next group head = call recursive function reverse( ) for $(k+1)^{th}$ node

for current k-group, reverse the links

tmp = head.next
head.next = next groupHead.
curr = head.
head = tmp.

Detailed Explanation :-

First find (k+1) node.

If that exists, then reverse list with (k+1) node as head. and also reverse current k-group.

Eg:- 1 → 2 → [3 → 4] → 5

head

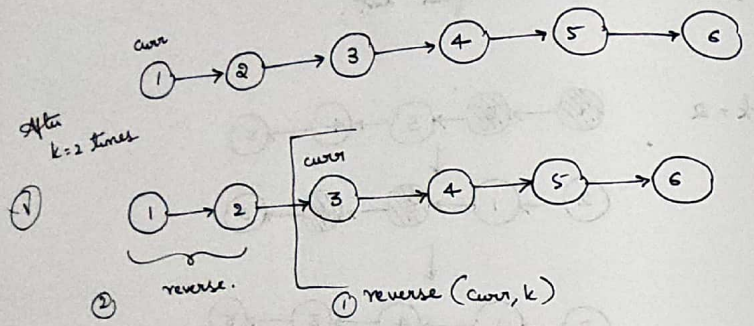
[(k+1) node exists]

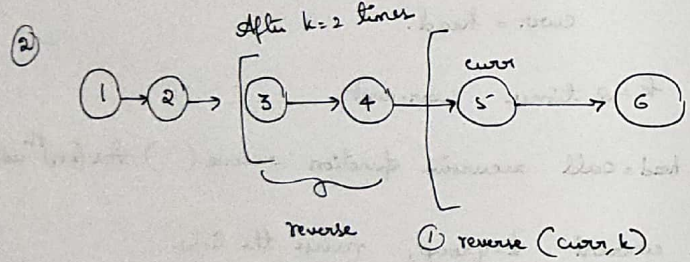
reverse ($k+1^{th}$ node)

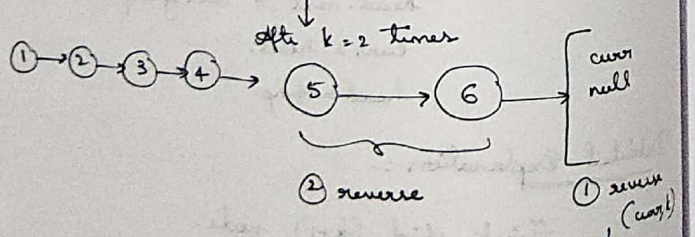
reverse current k group :- 2 → 1 → 4 → 3 → 5

($(k+1)^{th}$ reversed head)

Step-by-Step :-

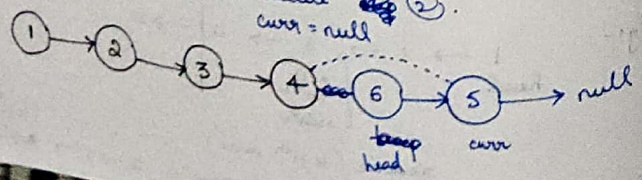






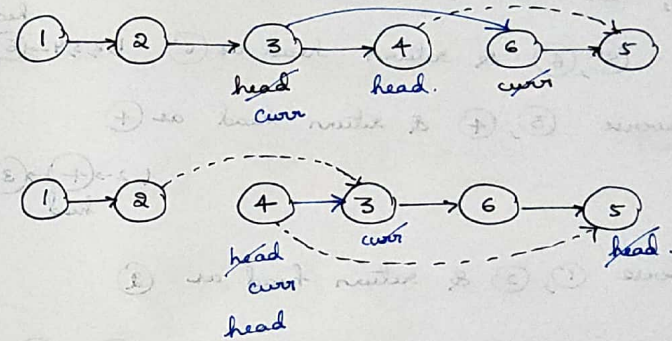
returns null.

④ ① returns null, so execute ②.

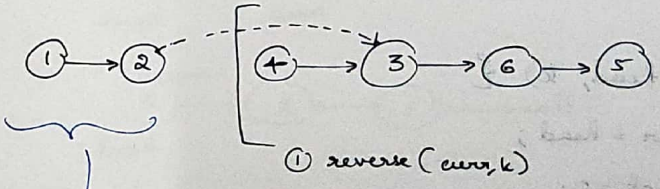


⑤ execute step ②.

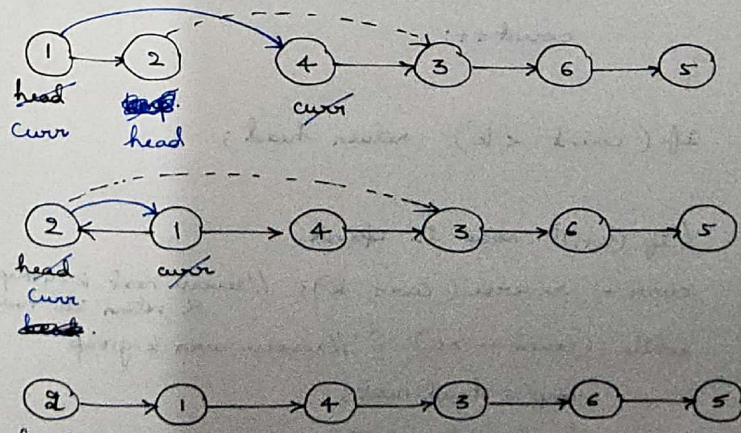
subtask ① returns node 6 ⇒ curr = 6.



⑥ execute step ①.



reverse current k-group.



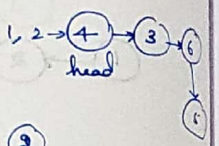
Approach / conclusion :-

We do reversal from last to first.

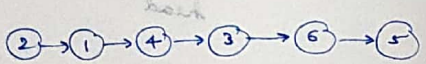
For suppose, we have 6 elements and k = 2.

We reverse (5), (6) & return head as (6)

Then we reverse (3), (4) & return head as (4)



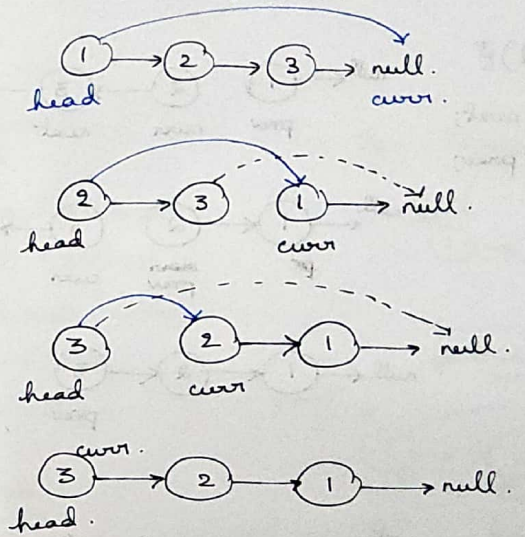
Then we reverse (1), (2) & return head as (2)



Program :-

```
reverse (head, k) {
    curr = head;
    count = 0.
    while( curr != null && count != k) {
        curr = curr.next;
        count++;
    }
    if (count < k) return head;

    //if (k+1)th node is found.
    curr = reverse (curr, k); //reverse next k-group & return its head.
    while (count-- > 0) { //reverse curr k-group
        tmp = head.next;
        head.next = curr;
        curr = head;
        head = tmp;
    }
    return head;
}
```

Reverse current k-group explanation :-



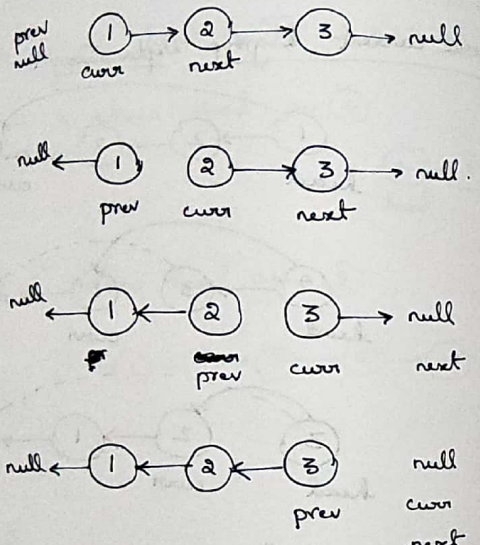
Reversing is done in such a way that we are ~~moving~~ linking 'head' ~~towards~~ the ~~end~~ "curr" of current group for k times.

THREE TEMPLATES OF RECURSION IN-PLACE :-

* TEMPLATE 1 :-

Eg:- Reverse complete linked list.

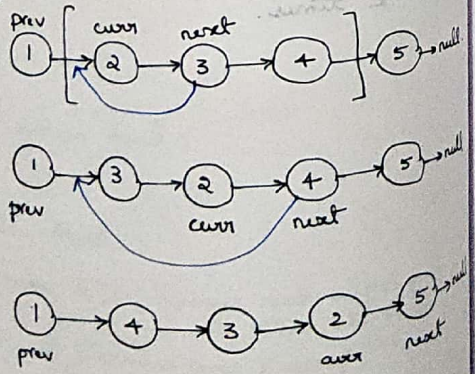
```
prev = null
curr = head
next = null

while (curr != null) {
    next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
}
head = prev;
return head;
```



* TEMPLATE 2 :- Insert "next" between "prev" & "prev.next"

```
reverse (left, right, prevNode):
    curr = prevNode.next
    next = curr.next
    for (int i=1; i<= right-left; i++){
        curr.next = next.next;
        next.next = prev.next;
        prev.next = next;
        next = curr.next;
    }
```

Eg:- Reverse sublist from 2 to 4



Note that "prev" & "curr" nodes are static.
We only move "next" pointer.

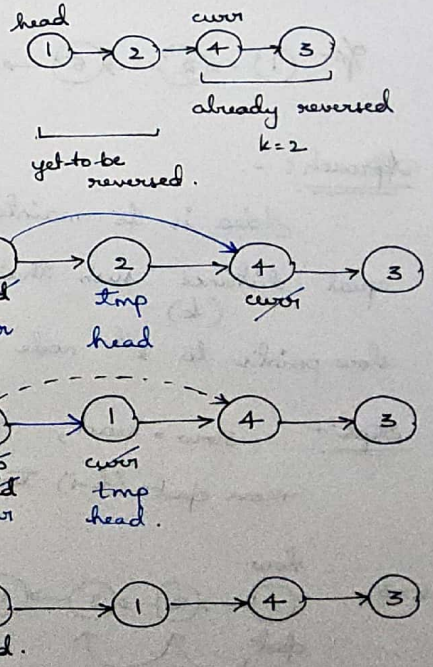
* In TEMPLATE 1, we keep moving prev, curr, next pointers to update links.

* In TEMPLATE 2, we do more like an insertion method. we only move next pointer.
we prefer this when we have "prev" node.

* TEMPLATE 3 :-

```
reverse (head, k) :
    count = k
    while (count-- > 0) {
        tmp = head.next;
        head.next = curr;
        curr = head;
        head = tmp;
    }
    head = curr;
```

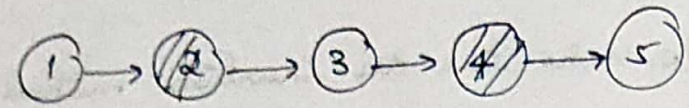
Eg:- Reverse 'k' sub-list
size



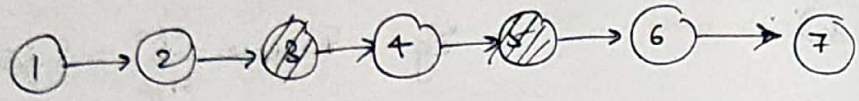
In TEMPLATE 3, we have head of next sublist as "curr".
We link current sublist "head" to the "curr" and update "curr" as head, k times.
At the end of 'k' times, our "head" is "curr"

* We use TEMPLATE 3 to reverse the list by adding head to the end of already reversed list (1→2→4→3 ⇒ 2→1→4→3 ⇒ 2→1→4→3)

Swapping Nodes in a Linked List LC(1721)



o/p:- 1 → 4 → 3 → 2 → 5



o/p:- 1 → 2 → 5 → 4 → 3 → 6 → 7

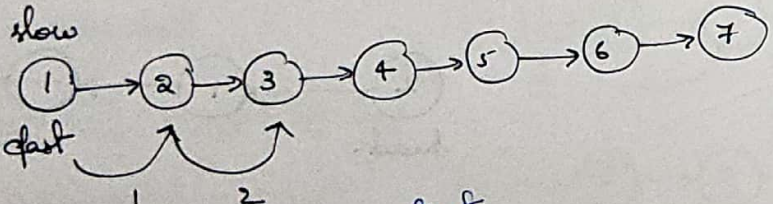
Approach:-

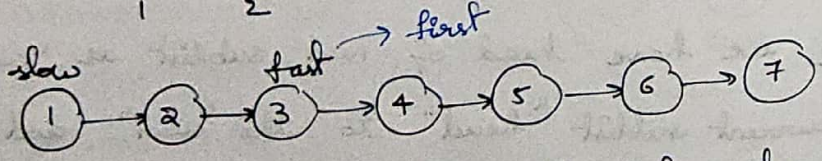
Idea is to maintain fast & slow pointers at equal distance such that when fast is at last node, slow points to k^{th} (k) node from the end.

Steps:- slow = head, fast = head.

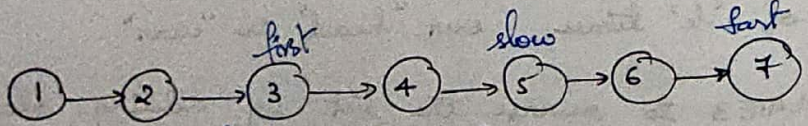
move fast (k-1) times.

k=3





move fast & slow one step until fast reaches end (fast.next != null)



swap first & slow values