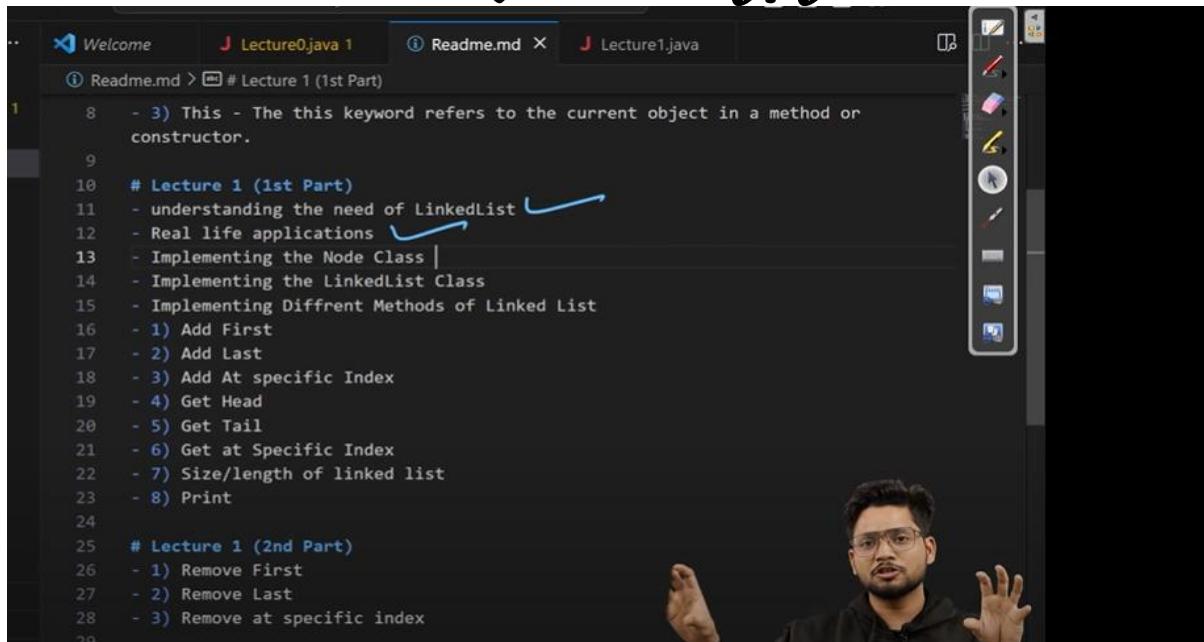
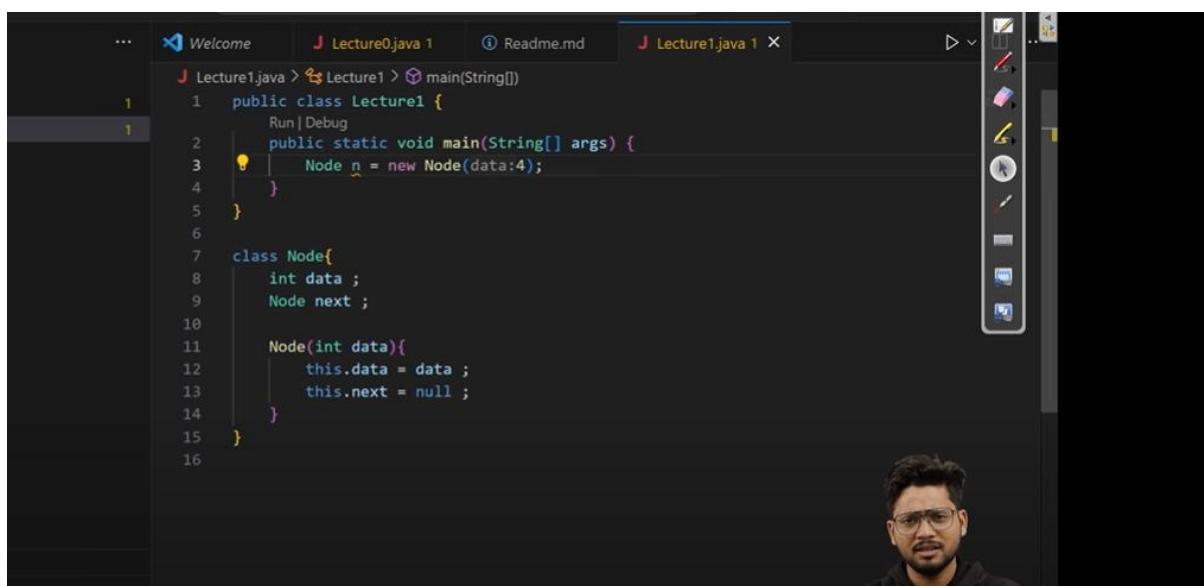


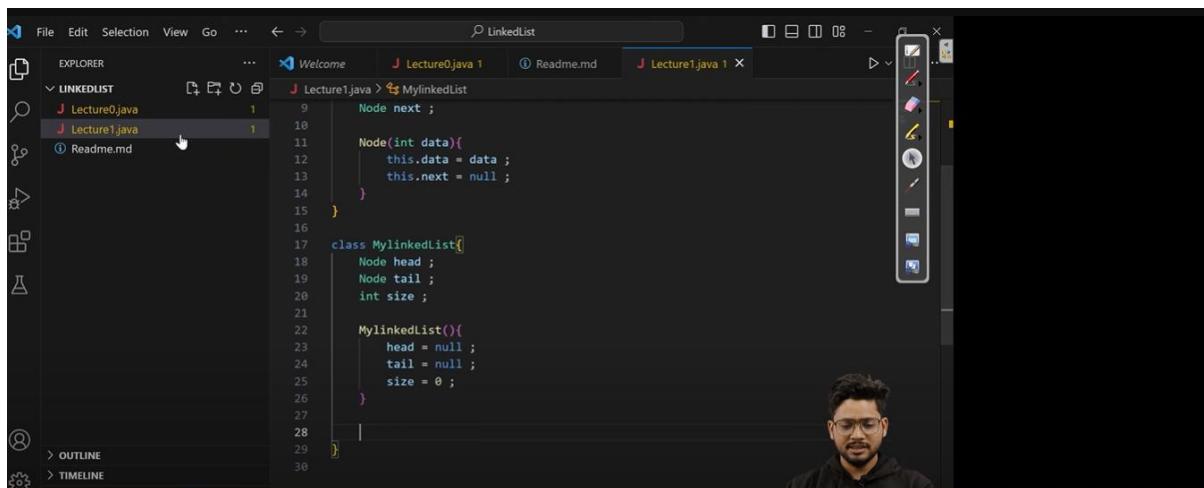
Linked List(gfg)



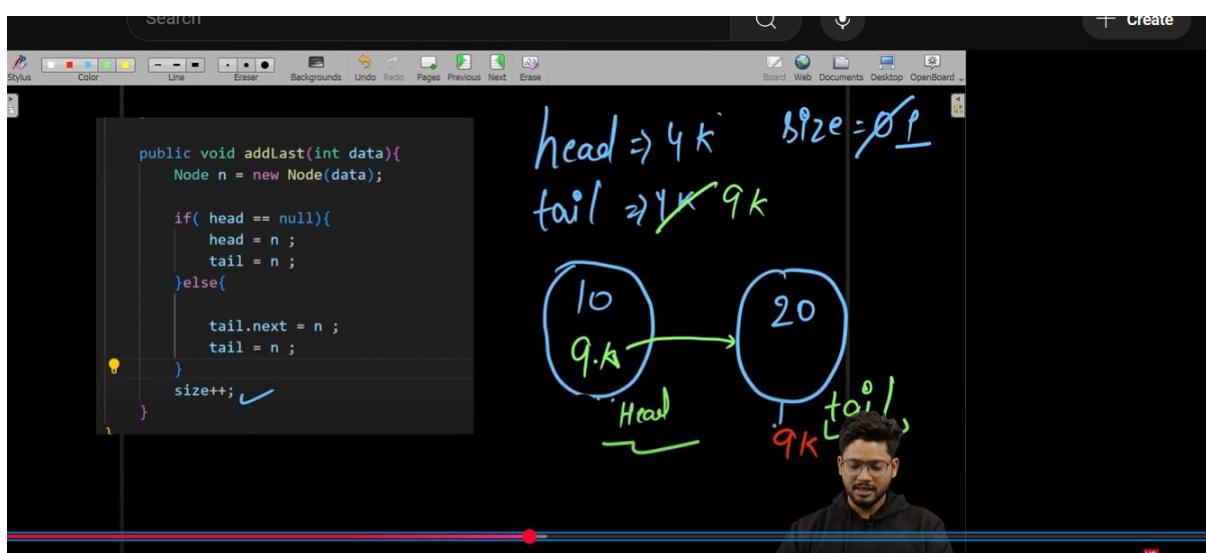
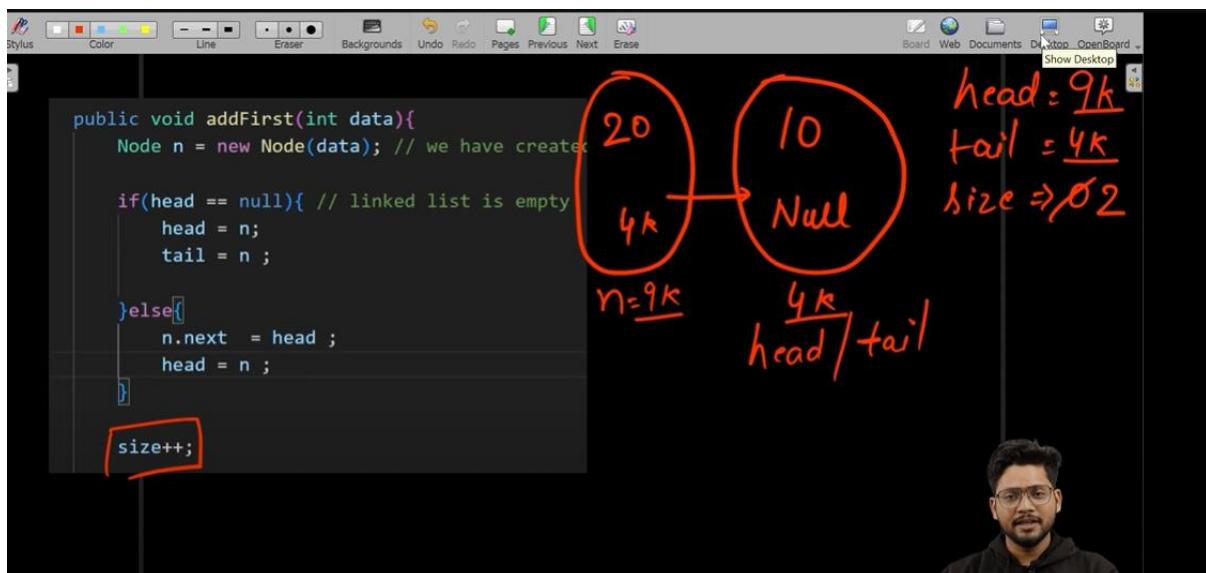
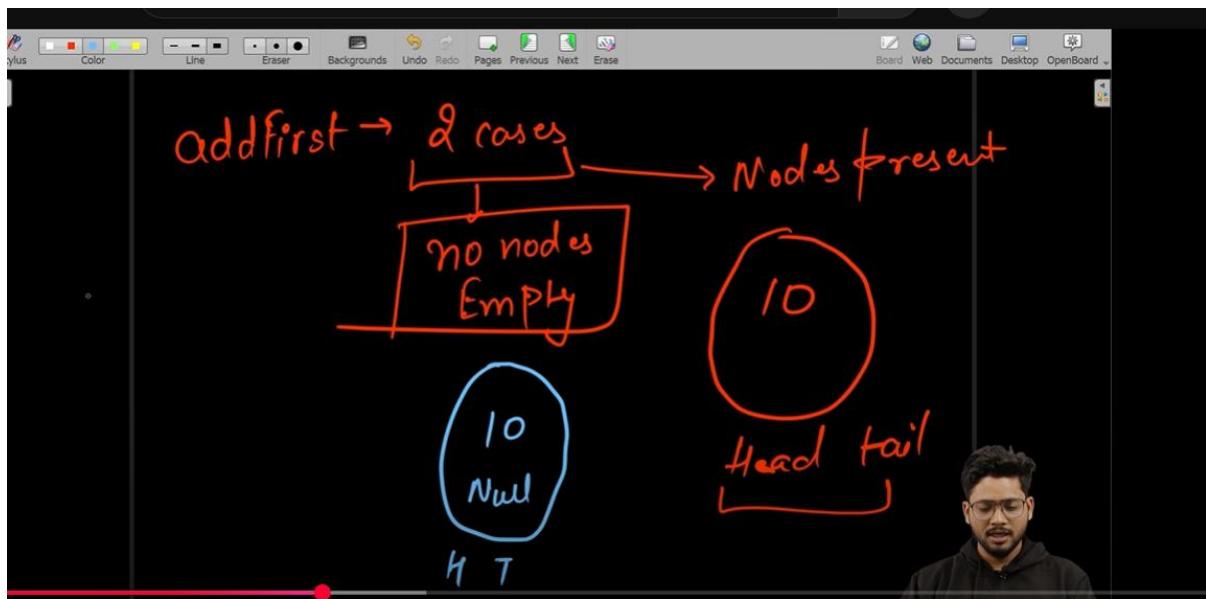
```
1     - 3) This - The this keyword refers to the current object in a method or
2       constructor.
3
4   # Lecture 1 (1st Part)
5   - understanding the need of LinkedList ✓
6   - Real life applications ✓
7   - Implementing the Node Class |
8   - Implementing the LinkedList Class
9   - Implementing Different Methods of Linked List
10  - 1) Add First
11  - 2) Add Last
12  - 3) Add At specific Index
13  - 4) Get Head
14  - 5) Get Tail
15  - 6) Get at Specific Index
16  - 7) Size/length of linked list
17  - 8) Print
18
19 # Lecture 1 (2nd Part)
20 - 1) Remove First
21 - 2) Remove Last
22 - 3) Remove at specific index
23
24
```



```
1  public class Lecture1 {
2      Run|Debug
3      public static void main(String[] args) {
4          Node n = new Node(data:4);
5      }
6
7      class Node{
8          int data ;
9          Node next ;
10         Node(int data){
11             this.data = data ;
12             this.next = null ;
13         }
14     }
15 }
16
```



```
1  File Edit Selection View Go ...
2  EXPLORER ... Welcome J Lecture0.java 1 Readme.md J Lecture1.java 1
3  LINKEDLIST J Lecture0.java 1 J Lecture1.java 1
4  Readme.md
5
6
7  9     Node next ;
8
9  10    Node(int data){
10      this.data = data ;
11      this.next = null ;
12  }
13
14  15 }
15
16  17 class MyLinkedList[|
17      Node head ;
18      Node tail ;
19      int size ;
20
21
22  22     MyLinkedList(){
23         head = null ;
24         tail = null ;
25         size = 0 ;
26     }
27
28  28 |]
29
30
```



```

Code 1 ⊖  Code 1 ⊖  +
1  public class Lecture1 {
2    public static void main(String[] args) {
3      MylinkedList list = new MylinkedList();
4
5      list.addFirst(10);
6      list.addFirst(20);
7      list.addLast(100);
8      list.addLast(190);
9
10     list.addAtSpecificIndex(110, 2);
11
12     list.addAtSpecificIndex(99, 1);
13
14     System.out.println(" before deletion " + list);
15
16     System.out.println(list.removeAtSpecificIndex(2));
17     System.out.println(list.removeAtSpecificIndex(3));
18
19     System.out.println("After deletion " + list);
20   }
21 }
22
23 class Node {
24   int data;
25   Node next;
26
27   public void addAtSpecificIndex(int data, int idx) {
28
29     if (idx < 0 || idx > size) {
30       System.out.println(" Index is not valid ");
31       return;
32     } else if (idx == 0) {
33       addFirst(data);
34     } else if (idx == size) {
35       addLast(data);
36     } else {
37       Node n = new Node(data);
38
39       Node pre = head;
40
41       while (idx - 1 > 0) {
42         pre = pre.next;
43         idx--;
44       }
45
46       Node nbr = pre.next;
47
48       pre.next = n;
49       n.next = nbr;
50
51       size++;
52     }
53   }
54
55   public int getFirst() {
56
57     if (head == null) {
58       System.out.println("LinkedList is empty");
59       return -1;
60     } else {
61       return head.data;
62     }
63   }
64
65   public int getLast() {
66
67     if (tail == null) {
68       System.out.println("LinkedList is empty");
69       return -1;
70     } else {
71       return tail.data;
72     }
73   }
74
75   public int getAtSpecificIndex(int idx) {
76
77     if (idx < 0 || idx >= size) {
78       System.out.println("Invalid index ");
79       return -1;
80     } else if (idx == 0) {
81       return getFirst();
82     } else if (idx == size - 1) {
83       return getLast();
84     } else {
85       Node curr = head;
86
87       while (idx > 0) {
88         curr = curr.next;
89         idx--;
90       }
91
92       return curr.data;
93     }
94   }
}

```

```

142 public int removeFirst() {
143     if (head == null) {
144         System.out.println("LinkedList is empty");
145         return -1;
146     } else if (head.next == null) {
147
148         int data = head.data;
149         head = null;
150         tail = null;
151
152         size--;
153         return data;
154     } else {
155
156         int data = head.data;
157
158         head = head.next;
159
160         size--;
161
162         return data;
163     }
164 }
165
166 public int removeLast() {
167     if (head == null) { // empty LL
168         System.out.println(" Linked list is empty ");
169         return -1;
170     } else if (head.next == null) { // one Node in LL
171         int data = head.data;
172
173         head = null;
174         tail = null;
175         size--;
176         return data;
177     } else {
178
179         Node curr = head;
180
181         int data = tail.data;
182
183         while (curr.next != tail) {
184             curr = curr.next;
185
186         curr.next = null;
187
188         size--;
189
190         tail = curr;
191
192         return data;
193     }
194 }
195

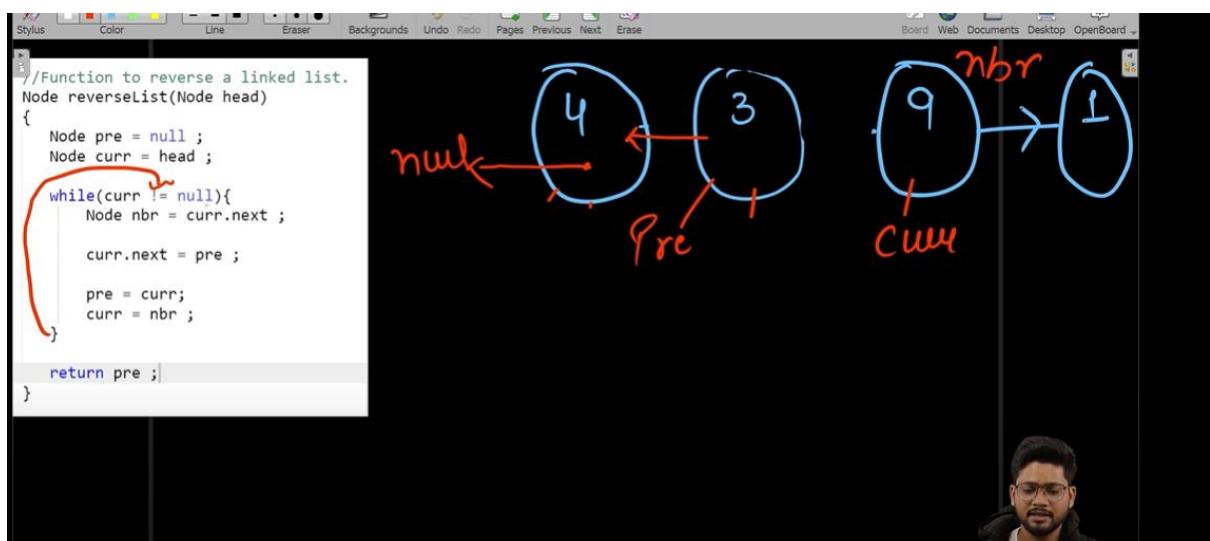
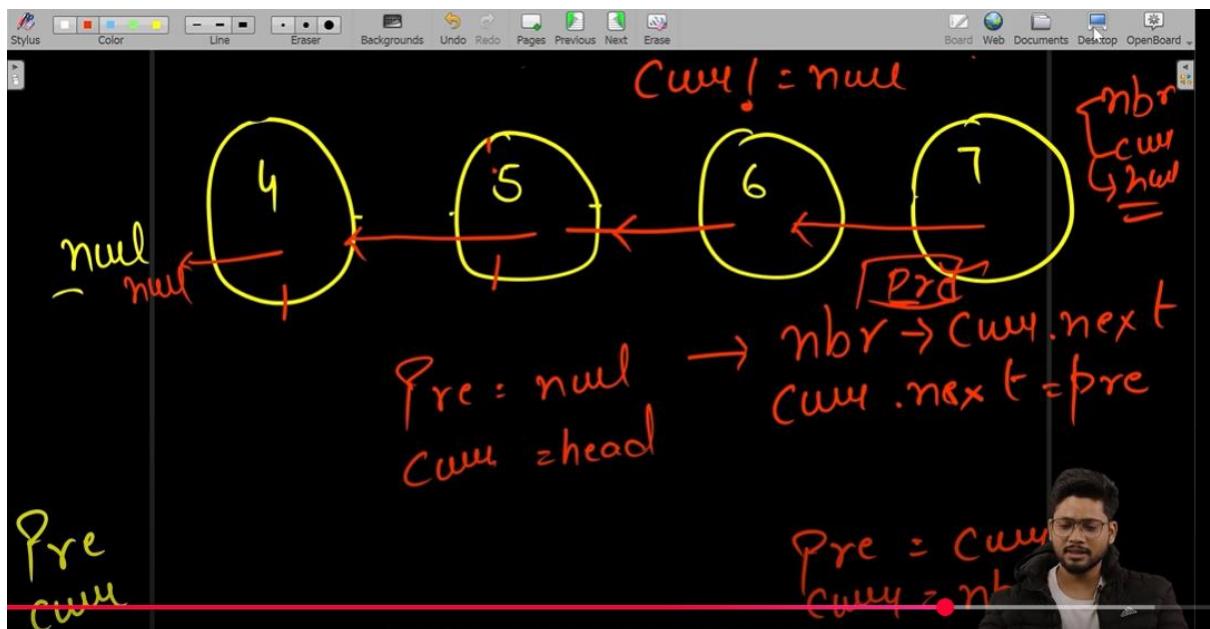
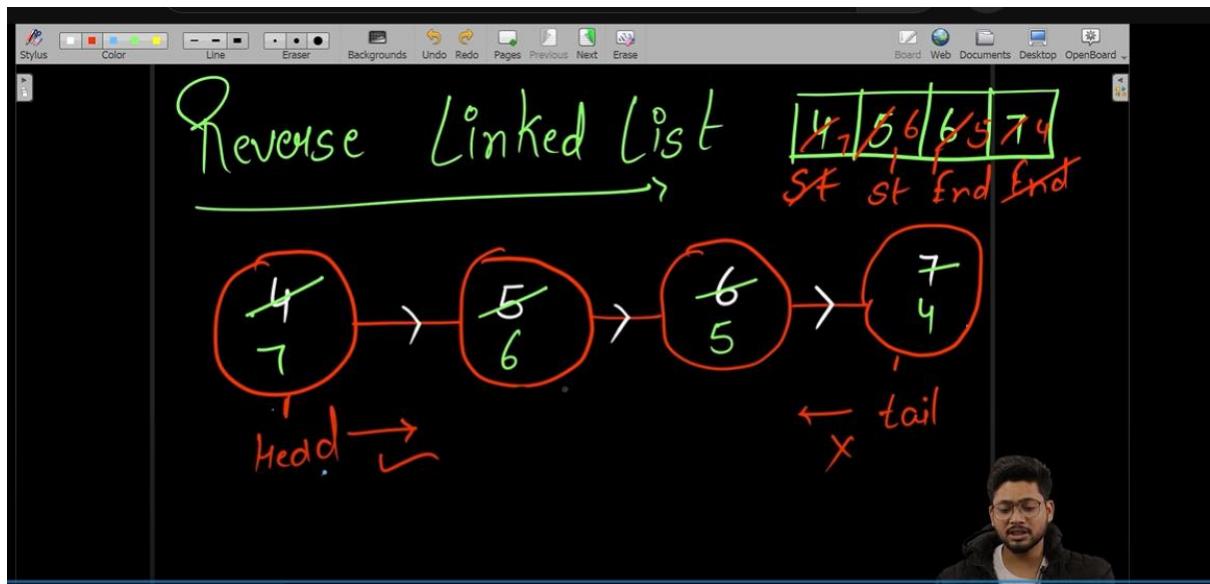
```

```

196
197 public int removeAtSpecificIndex(int idx){
198
199     if ( idx<0 || idx>= size ){
200         System.out.println("Invalid idx");
201         return -1 ;
202     }else if ( idx == 0 ){
203         return removeFirst();
204     }else if ( idx == size-1 ){
205         return removeLast();
206     }else {
207
208         Node curr = head ;
209
210         while( idx-1 > 0){
211             curr = curr.next ;
212
213             idx-- ;
214
215             int data = curr.next.data ;
216             curr.next = curr.next.next ;
217
218             size-- ;
219
220             return data ;
221     }
222
223
224     public String toString() {
225
226         String str = "";
227
228         Node curr = head;
229
230         while (curr != null) {
231
232             str = str + curr.data + " ";
233
234             curr = curr.next;
235
236         return str;
237     }
238
239     public int length() {
240         return size;

```

1. Reverse Linked List



2. Middle Element in LinkedList

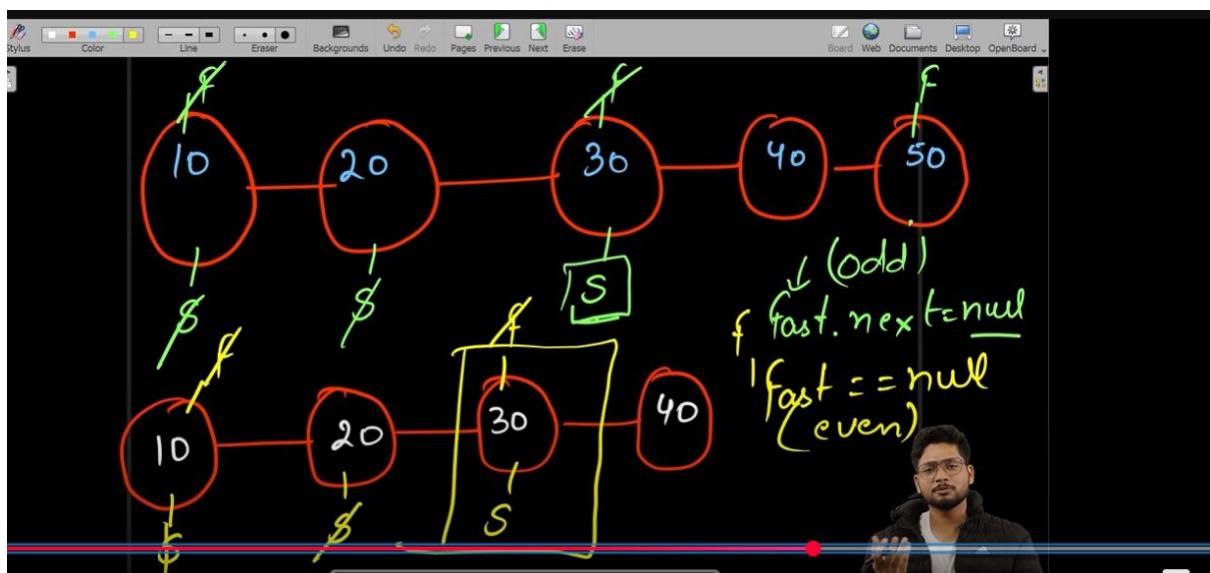
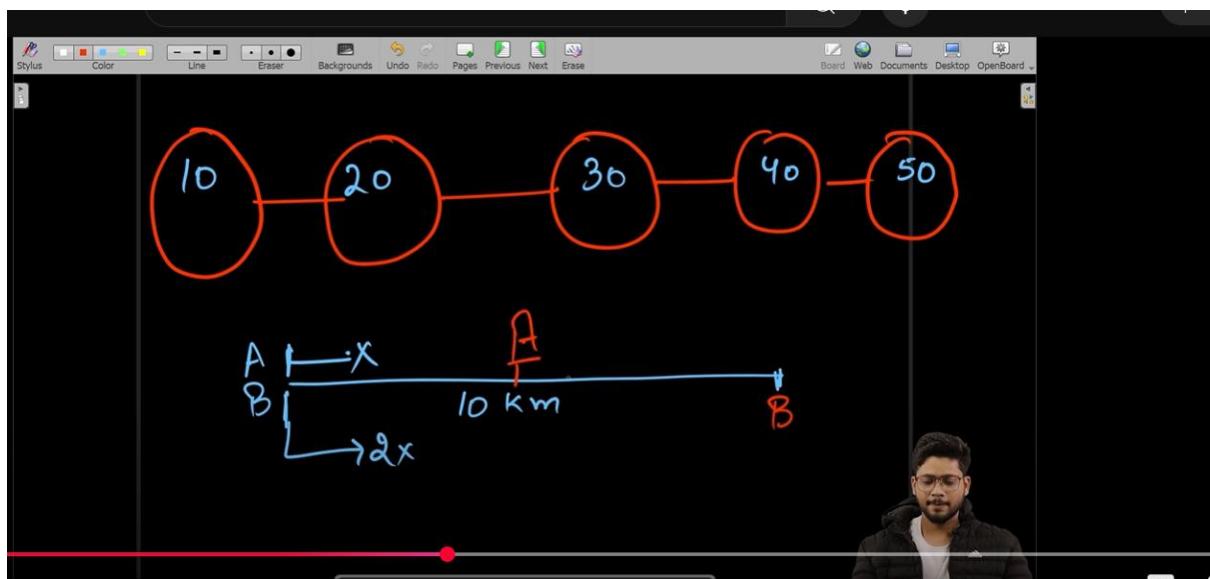
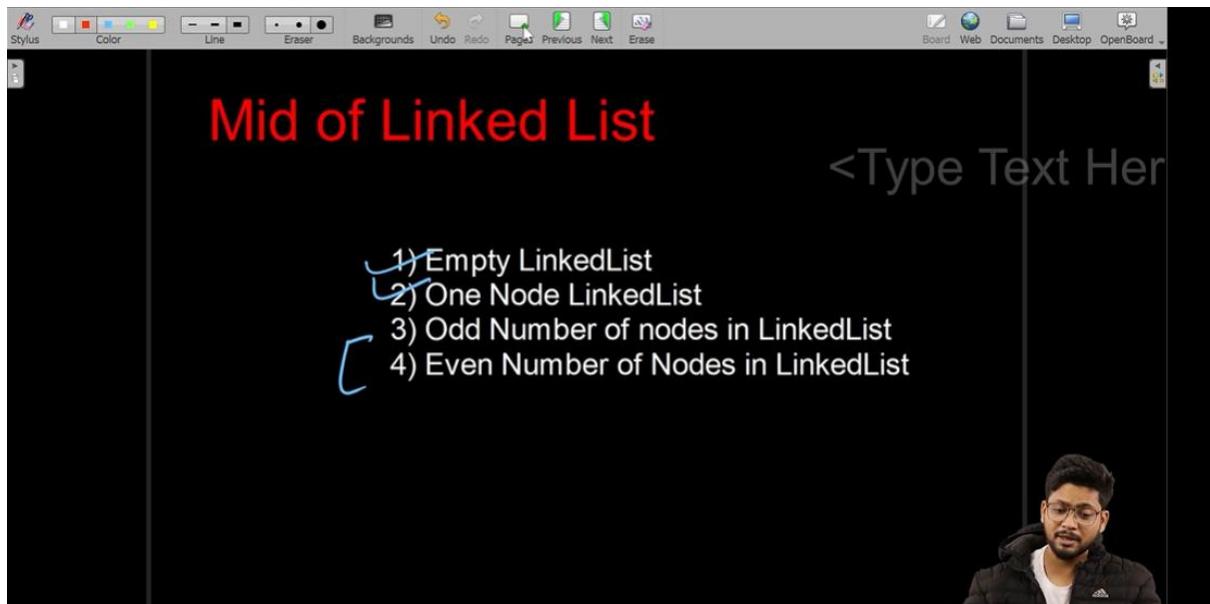


Diagram illustrating the slow and fast pointer approach to find the middle element of a linked list.

```

int getMiddle(Node head)
{
    if( head == null){
        return -1 ;
    }else if ( head.next == null){
        return head.data ;
    }

    Node fast = head ;
    Node slow = head ;

    while(fast!=null && fast.next!=null){
        fast = fast.next.next ;
        slow = slow.next ;
    }

    return slow.data ; ✓
}

```

3.Nth Node from End of LinkedList

Diagram illustrating the two-pointer approach to find the n^{th} node from the end of a linked list.

Diagram illustrating the two-pointer approach to find the n^{th} node from the end of a linked list.

```

if( head == null){
    return -1 ;
}

Node curr = head ;

while(curr!=null && n>0){
    curr = curr.next ;
    n--;
}

if( curr == null && n>0){
    return -1 ;
}

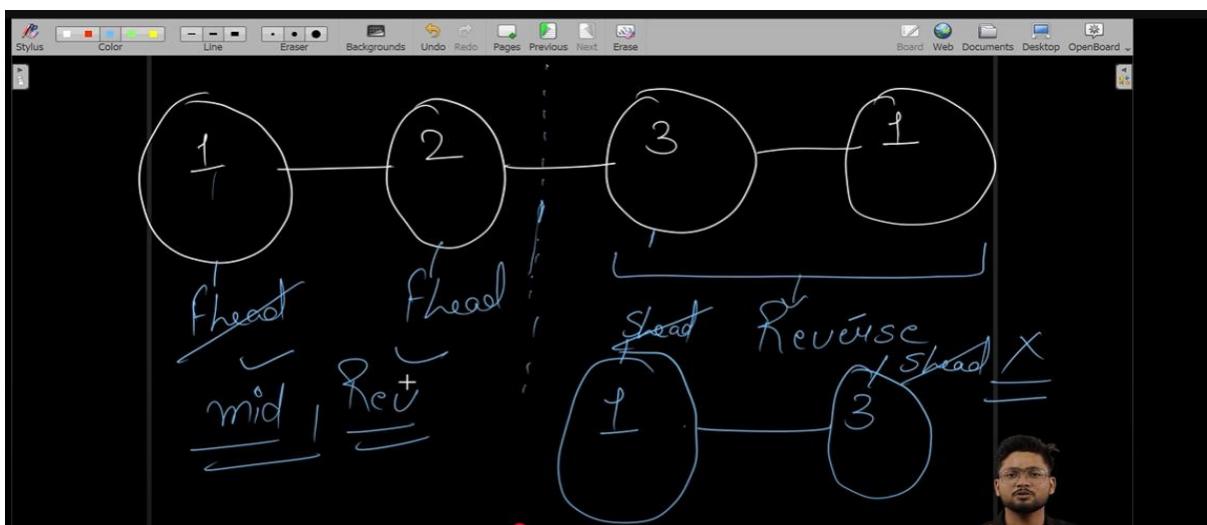
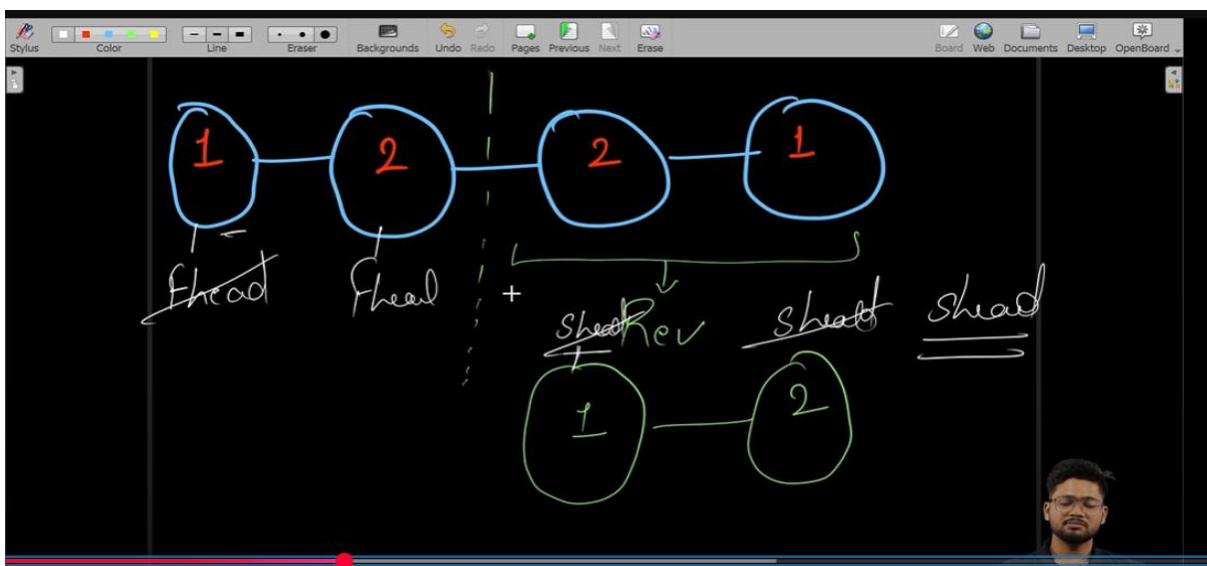
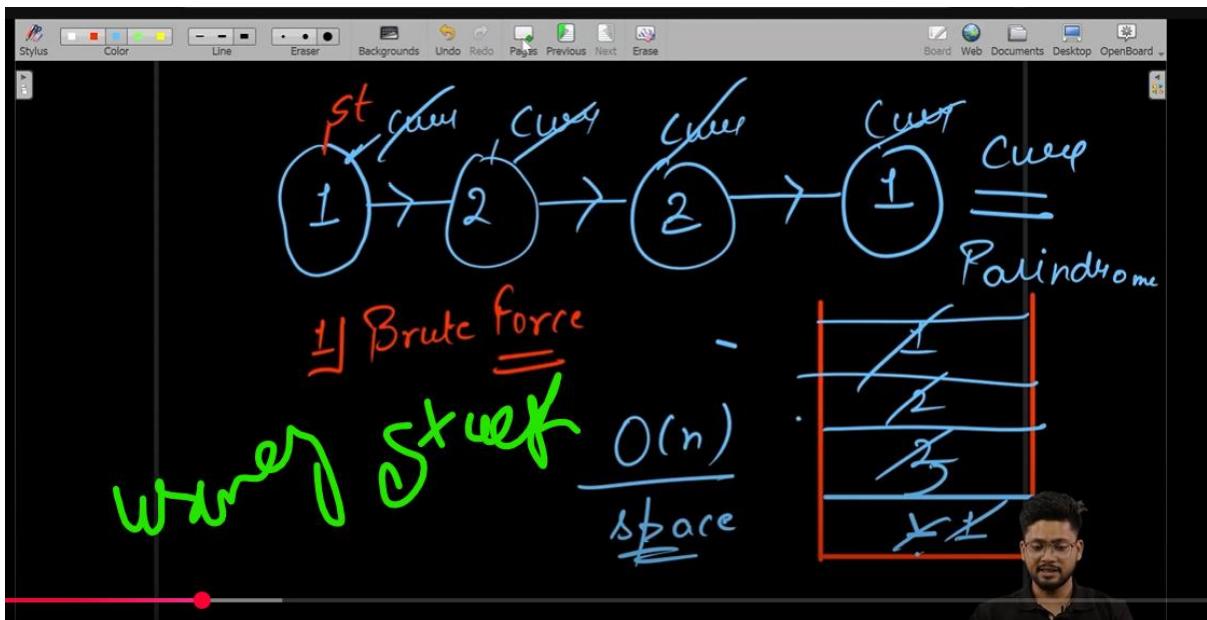
Node ans = head ;

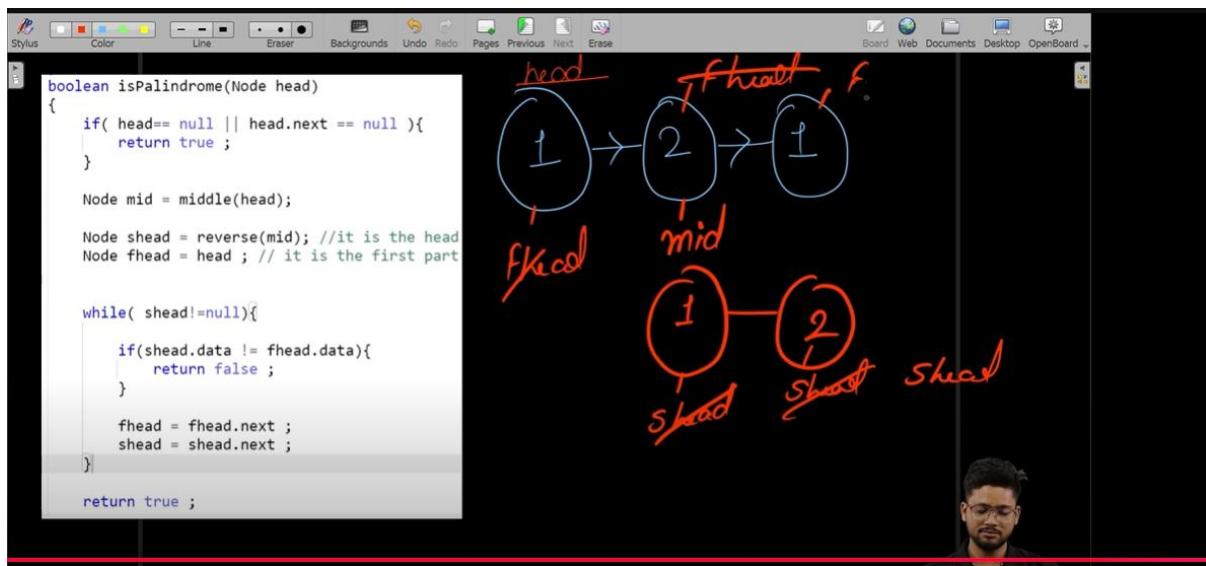
while( curr!= null){
    curr = curr.next ;
    ans = ans.next ;
}

return ans.data ;

```

4. Check if linked list is palindrome





if the given linked list is palindrome or not.

Example 1:

Input:
N = 3
value[] = {1,2,1}
Output: 1
Explanation: The given linked list is 1 2 1 , which is a palindrome and Hence, the output is 1.

Example 2:

Input:
N = 4
value[] = {1,2,3,4}

```

86 class Solution
87 {
88     Node middle(Node head ){
89         if( head == null){
90             return head ;
91         }
92
93         Node fast = head ;
94         Node slow = head ;
95
96         while( fast!=null && fast.next!= null){
97             fast = fast.next.next ;
98             slow = slow.next;
99         }
100
101         return slow ;
102     }
103
104
105     boolean isPalindrome(Node head)
106     {
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127

```

if the given linked list is palindrome or not.

Example 1:

Input:
N = 3
value[] = {1,2,1}
Output: 1
Explanation: The given linked list is 1 2 1 , which is a palindrome and Hence, the output is 1.

Example 2:

Input:
N = 4
value[] = {1,2,3,4}

Output: 0
Explanation: The given linked list

```

104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127

```

5. Implement stack using Linked list

A screenshot of a video call interface. On the left, a code editor shows a C++ function for pushing an integer into a stack. The function creates a new node and sets it as the top if the stack was empty. Otherwise, it links the new node's next pointer to the previous top and updates the top pointer. Handwritten annotations show the initial state where `top = null`, followed by three pushes: `push → x`, `push → y`, and `push → z`. A diagram illustrates the linked list structure with nodes labeled `x`, `y`, and `z`, each with a self-loop arrow pointing to its `next` pointer. The `top` pointer is shown pointing to the `node` containing `z`. A small video window of a teacher is visible on the right.

```
//Function to push an integer into the stack.  
void push(int a)  
{  
    StackNode node = new StackNode(a);  
  
    if(top == null){ // this is the first node  
        top = node;  
    }else{  
        node.next = top;  
        top = node;  
    }  
}
```

A screenshot of a video call interface. On the left, a code editor shows a C++ function for pushing an integer into a stack and another for popping an item from the top. The push function is identical to the one in the previous screenshot. The pop function checks if the stack is empty, returns -1 if so, otherwise retrieves the top node's data, sets the new top to its next, and returns the data. Handwritten annotations show four pushes: `push → 20`, `push → 40`, `push → 75`, and `pop → 75`. A final diagram shows the stack with nodes `99`, `40`, and `20`, with the `top` pointer pointing to the `node` containing `20`. The text `Ans = 75` is written above the stack. A small video window of a teacher is visible on the right.

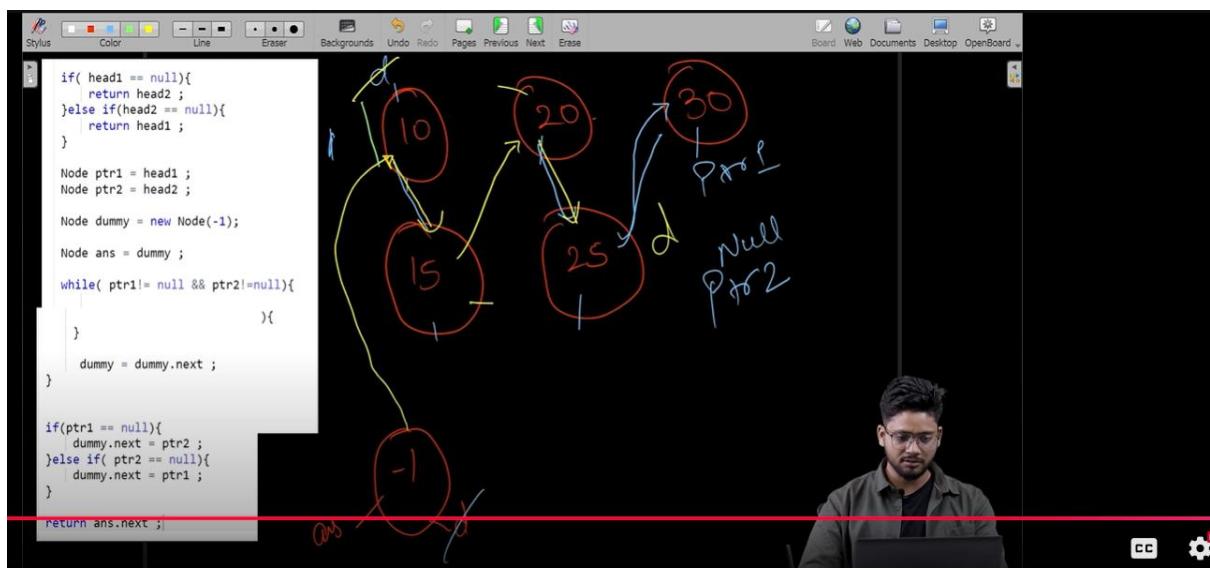
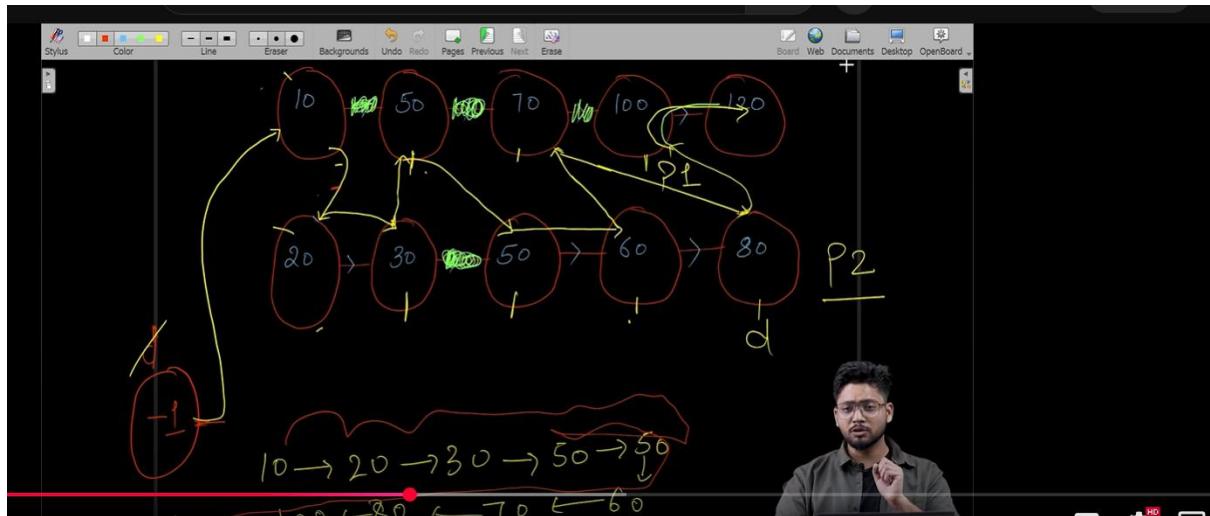
```
void push(int a)  
{  
    StackNode node = new StackNode(a);  
  
    if(top == null){ // this is the first node  
        top = node;  
    }else{  
        node.next = top;  
        top = node;  
    }  
}  
  
//Function to remove an item from top of the stack  
int pop()  
{  
    if(top == null){ // the stack is empty  
        return -1;  
    }else{  
        int ans = top.data;  
  
        top = top.next;  
  
        return ans;  
    }  
}
```

6. Implement Queue using Linkedlist

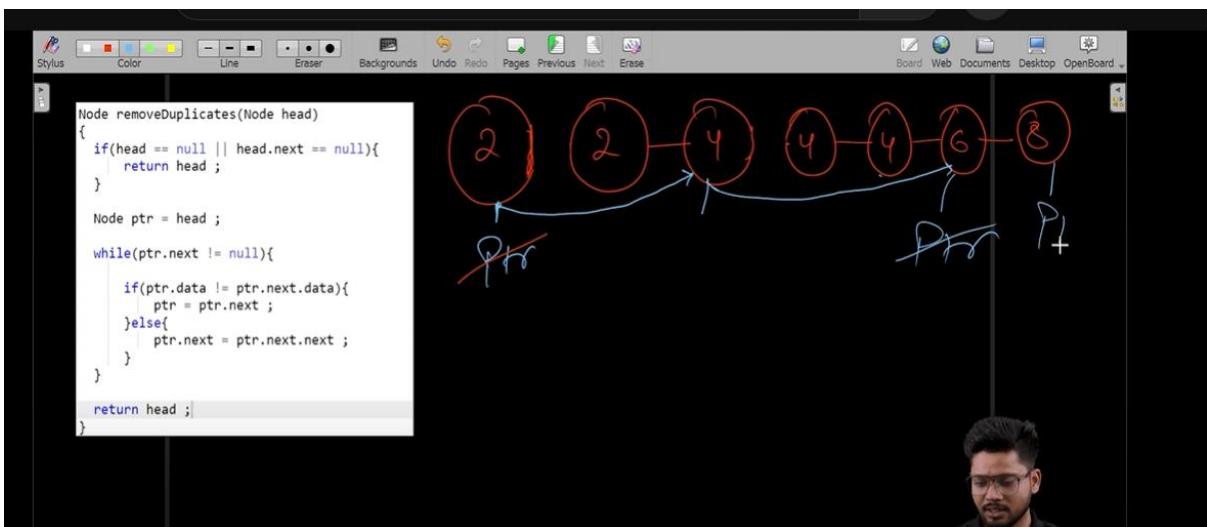
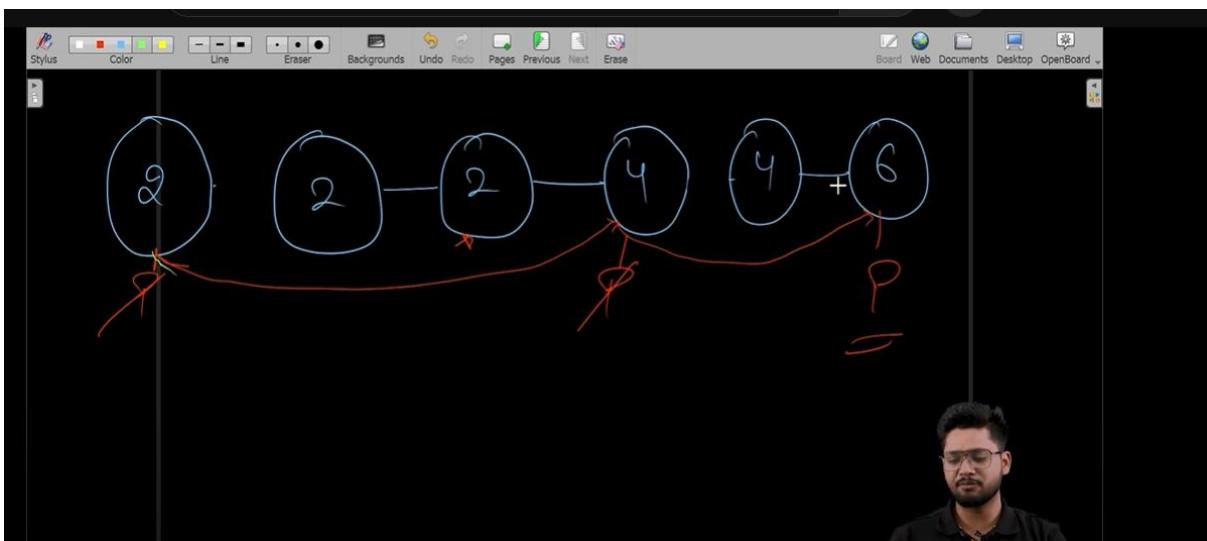
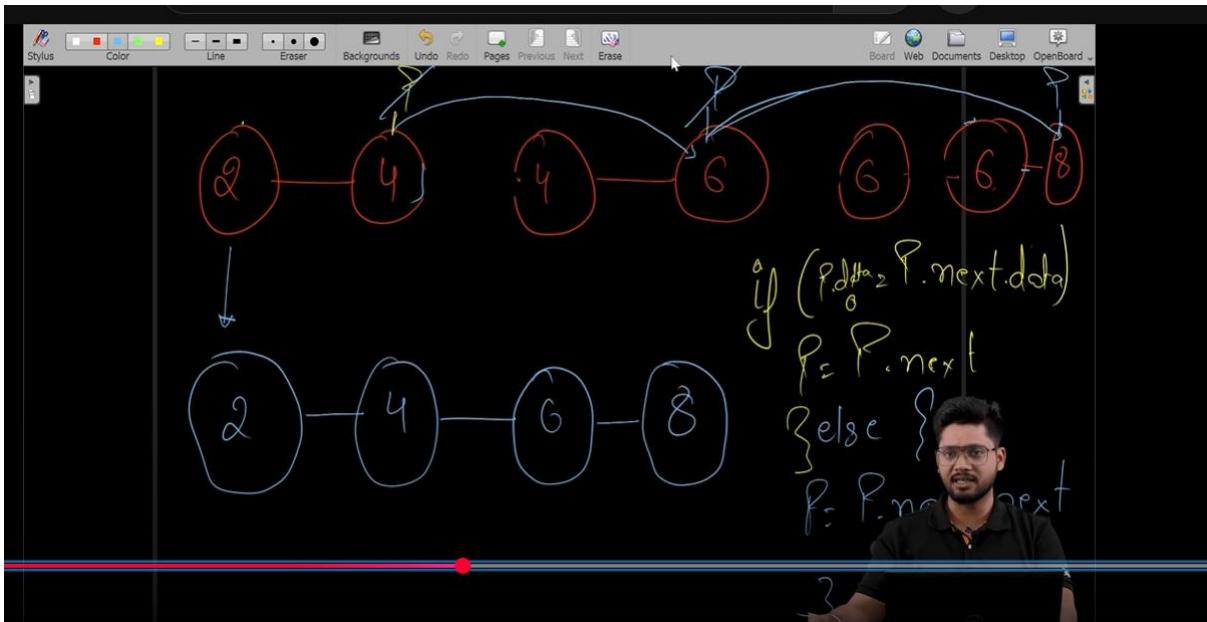
The diagram shows a linked list representation of a queue. It consists of three nodes containing the values 10, 20, and 30. The nodes are connected by arrows pointing to the right, indicating the direction of the queue's flow. The first node (10) has its 'front' pointer pointing to the second node (20), and the second node (20) has its 'front' pointer pointing to the third node (30). The third node (30) has its 'next' pointer pointing to 'N' (null). The 'rear' pointer of the second node (20) also points to 'N'. Handwritten annotations include 'Pop →' above the first node, 'Front' and 'Rear' labels with arrows pointing to the first and second nodes respectively, and 'Rear.next = n' and 'n = Rear' on the right side. On the left, there is a stack of three cards labeled 'push → 10', 'push → 20', and 'push → 30'. Below the stack, 'Front=null' and 'Rear=null' are written with a line through them, and '20' is written below the stack.

The diagram shows a linked list representation of a queue with three nodes containing the values 10, 20, and 30. The nodes are connected by arrows pointing to the right. The first node (10) has its 'front' pointer pointing to the second node (20), and the second node (20) has its 'front' pointer pointing to the third node (30). The third node (30) has its 'next' pointer pointing to 'N' (null). Handwritten annotations include 'push → 10', 'push → 20', 'push → 30', 'pop →', and 'pop →' with arrows pointing to the nodes. Above the nodes, 'Front=null' and 'Rear=null' are written with a line through them. To the right, 'FR' is written above the second node (20) and 'dat or' is written below the third node (30). On the left, there is a stack of three cards labeled 'push → 10', 'push → 20', and 'push → 30'. Below the stack, 'Front=null' and 'Rear=null' are written with a line through them, and '20' is written below the stack.

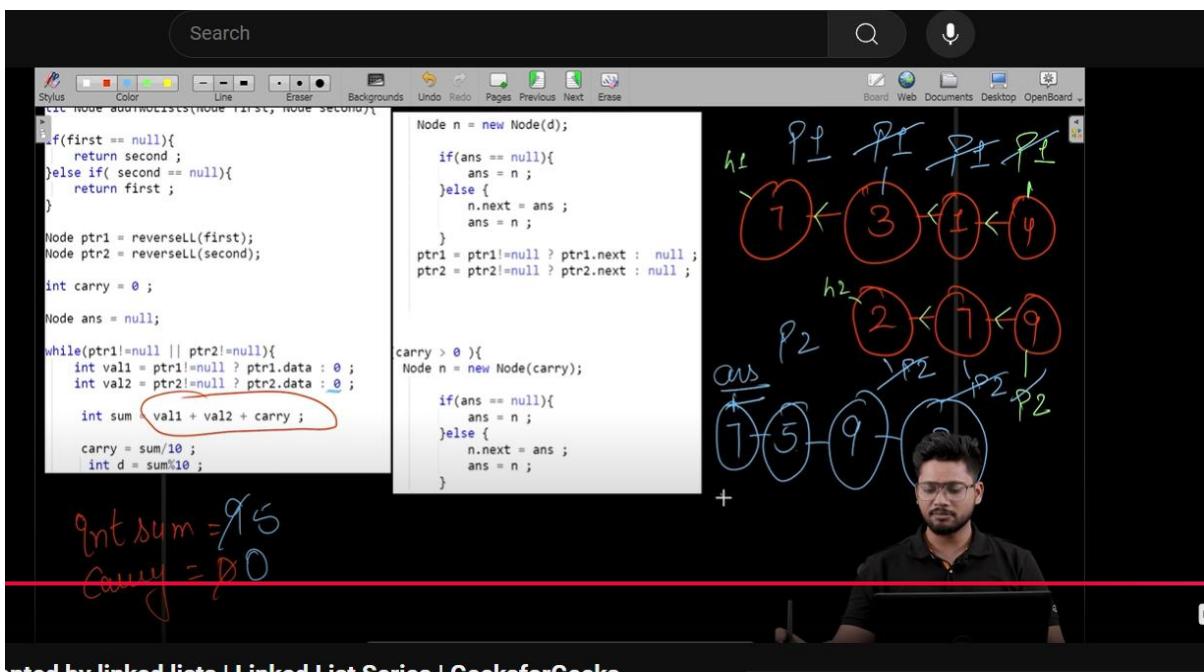
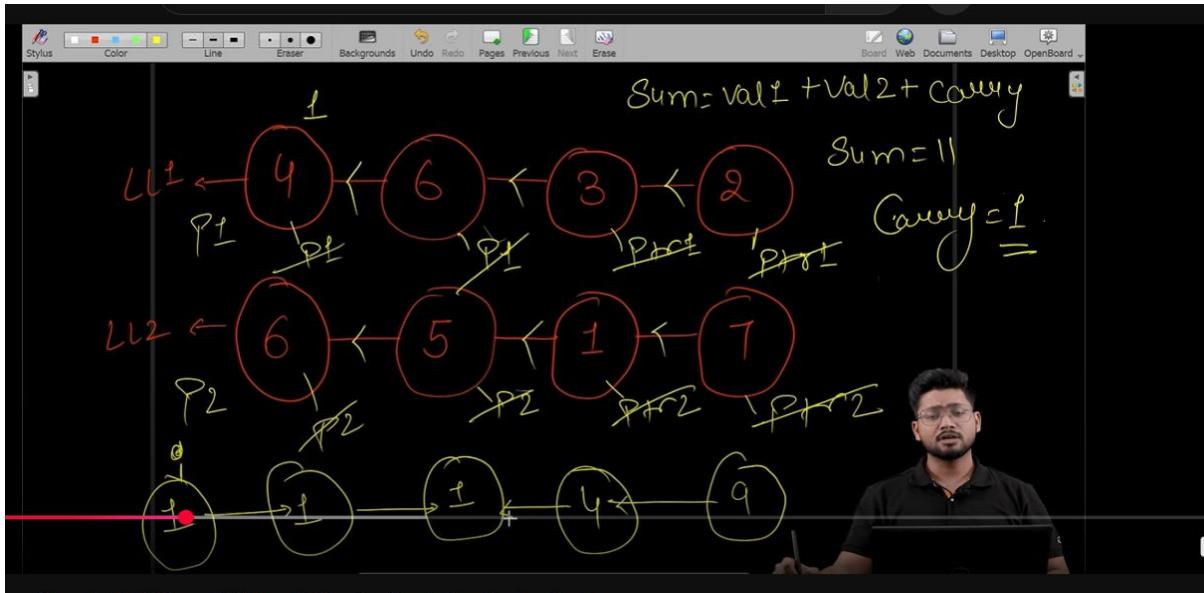
7. merge Two Sorted Linkedlist



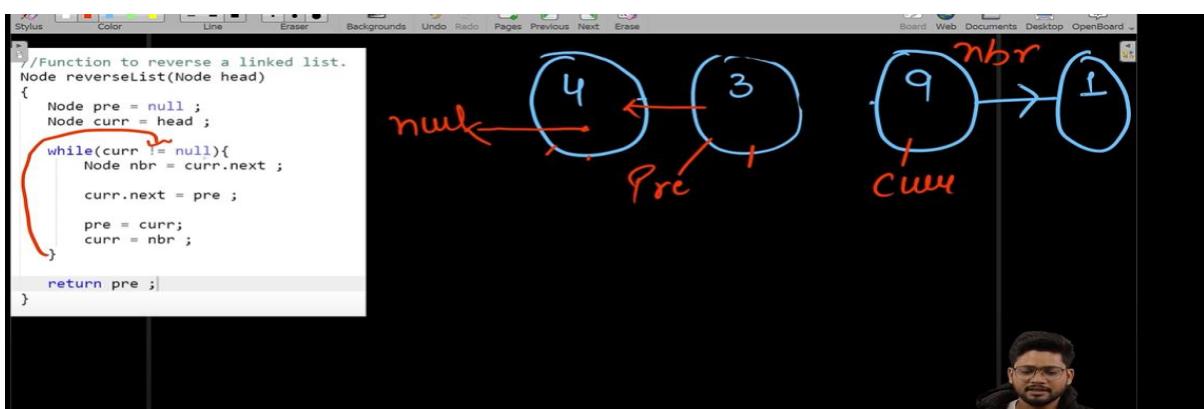
8. Remove duplicate element from sorted linked list



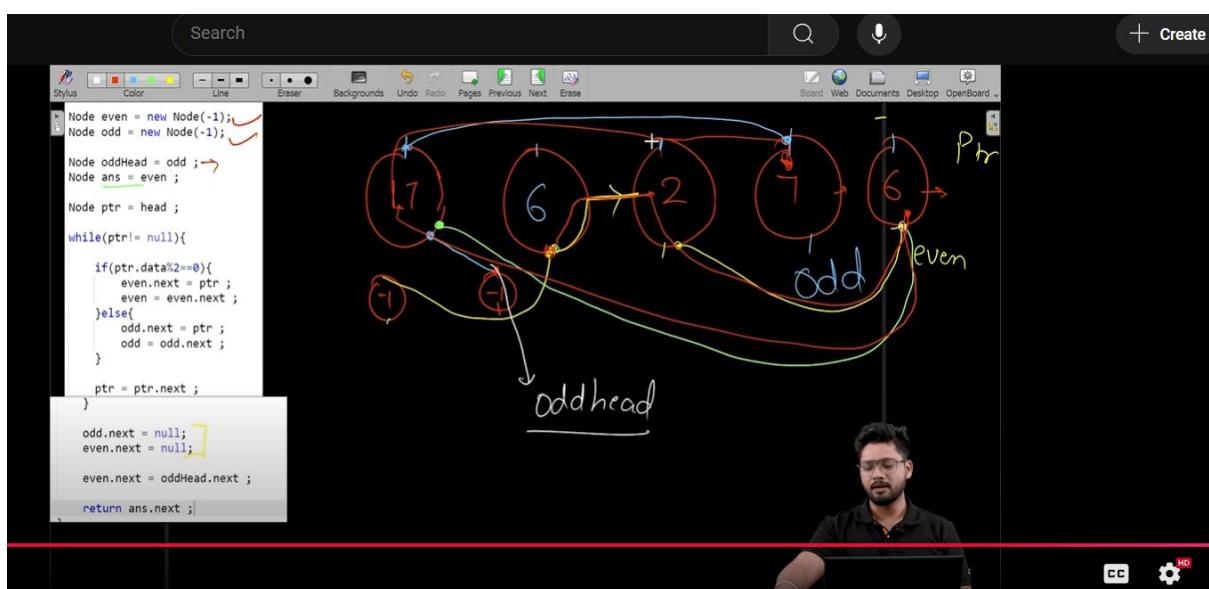
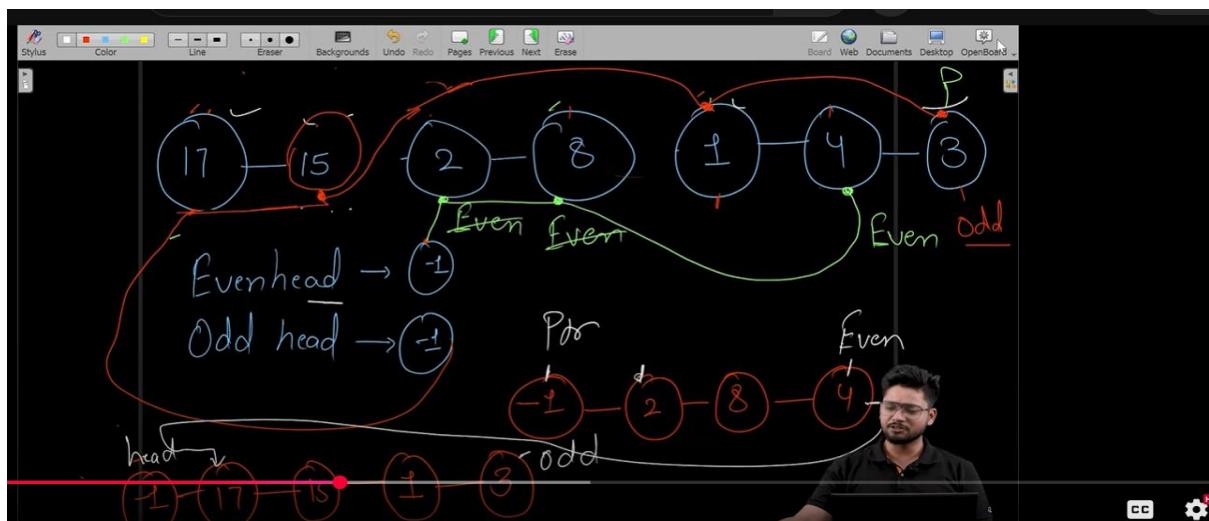
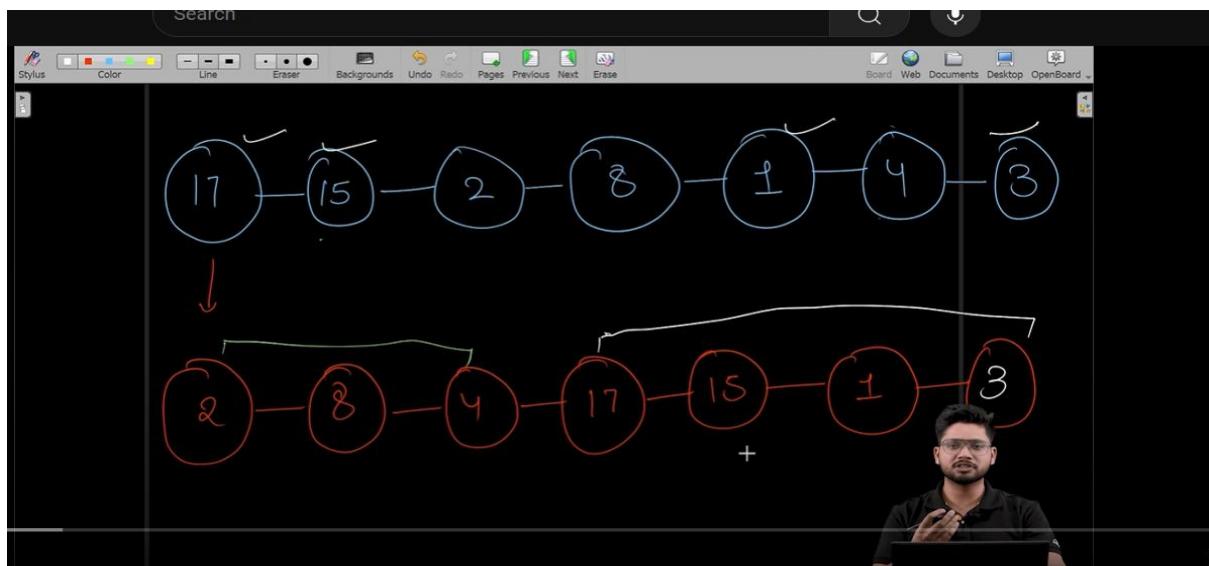
9. Add Two Numbers represented by linked list



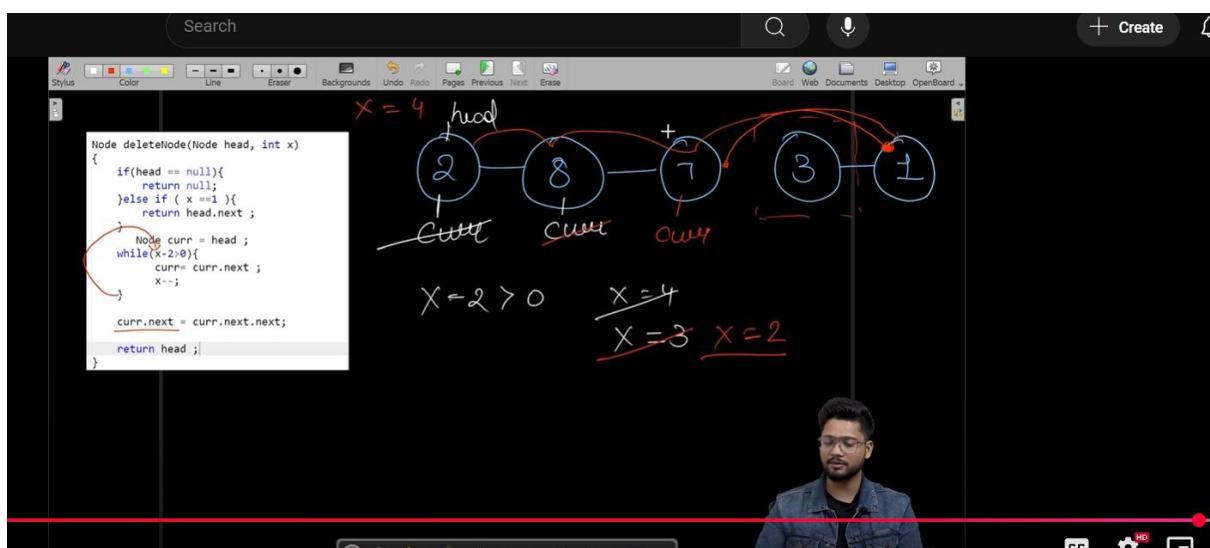
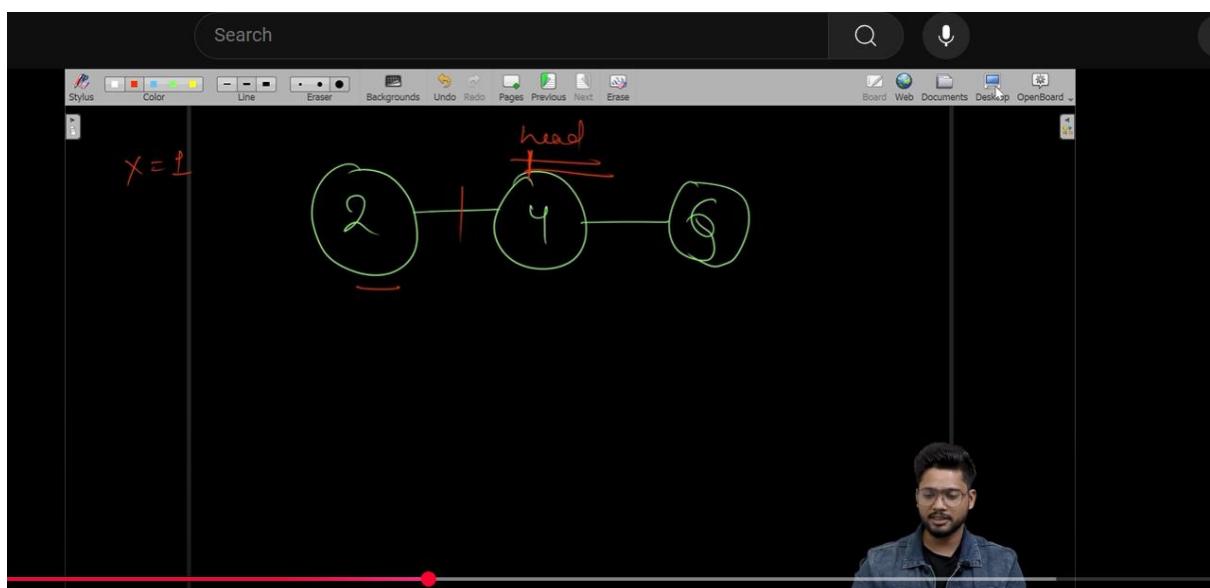
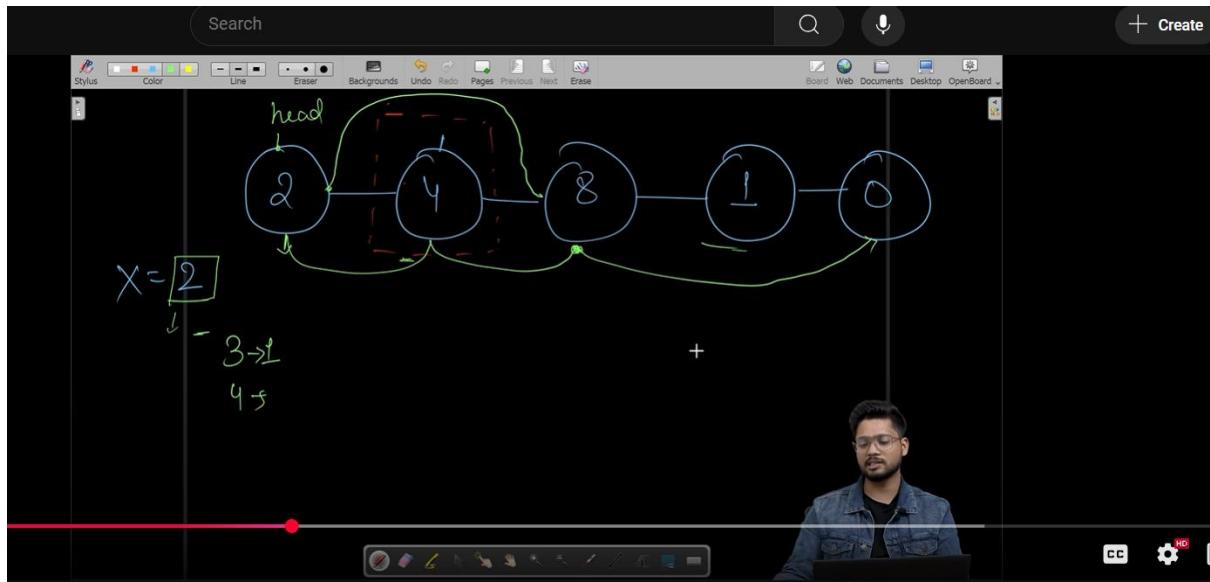
Addition of two numbers represented by linked lists | Linked List Series | GeeksforGeeks



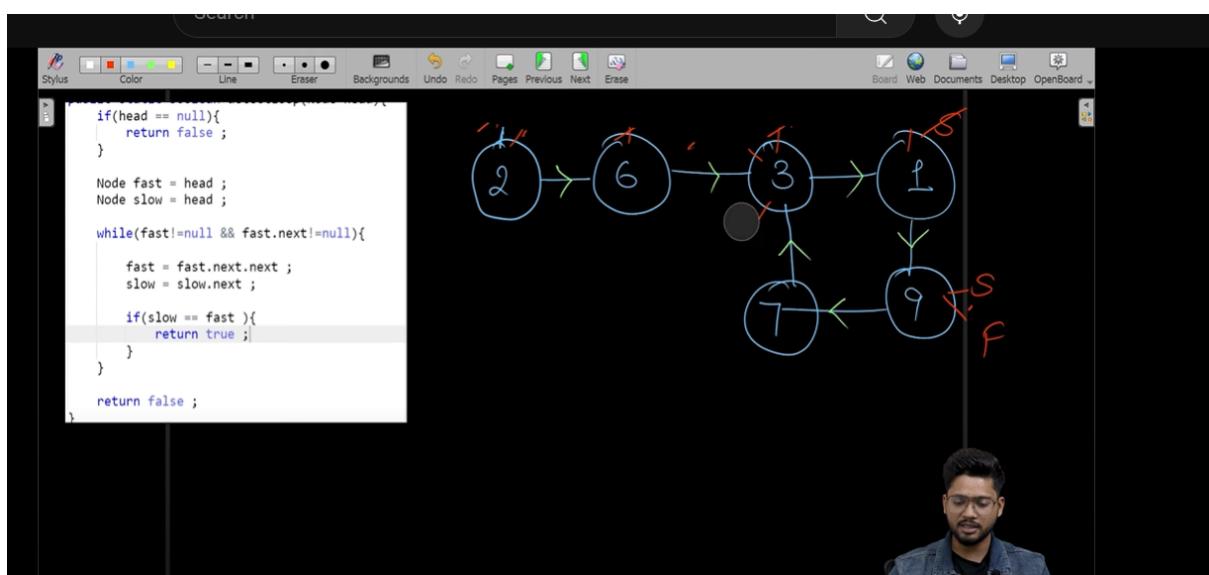
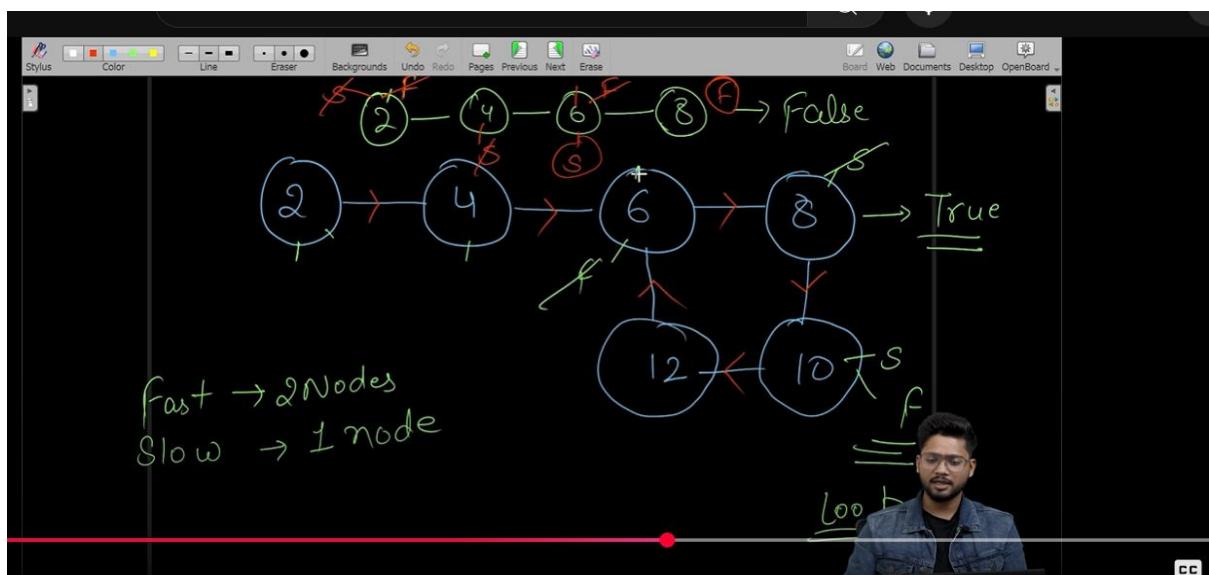
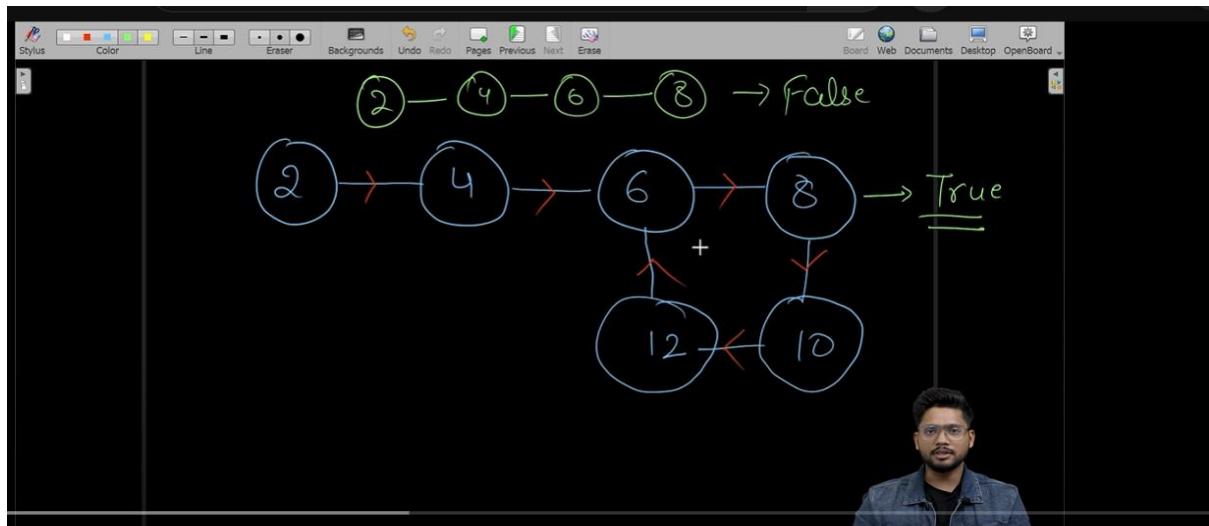
10. Segregate even ans odd in linked list



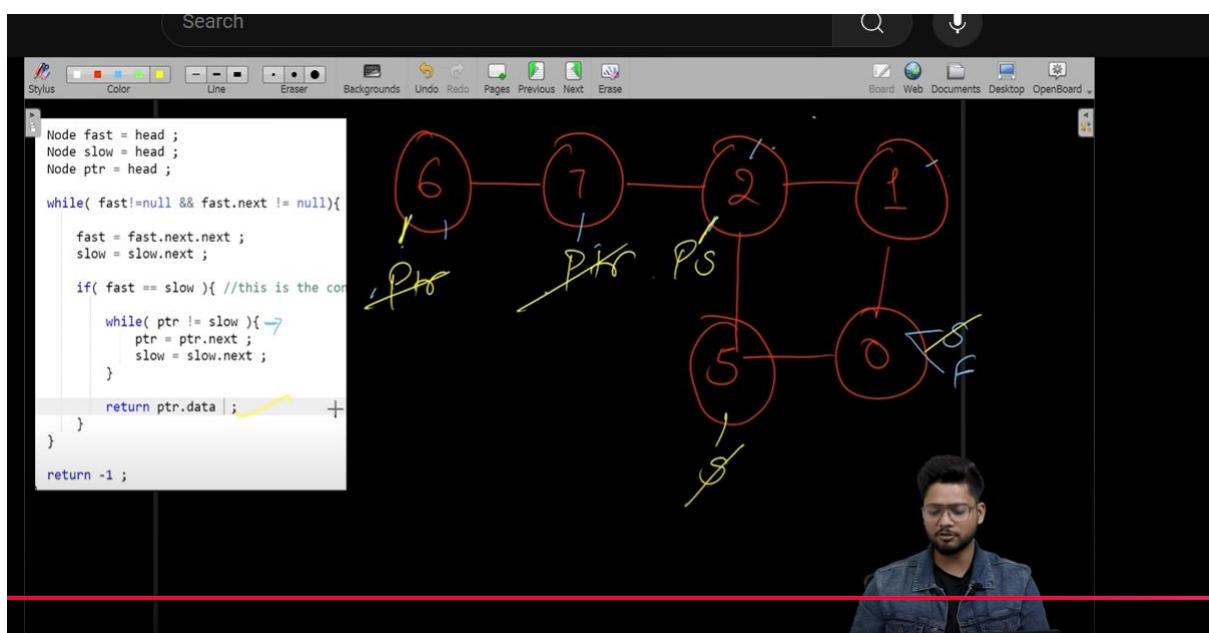
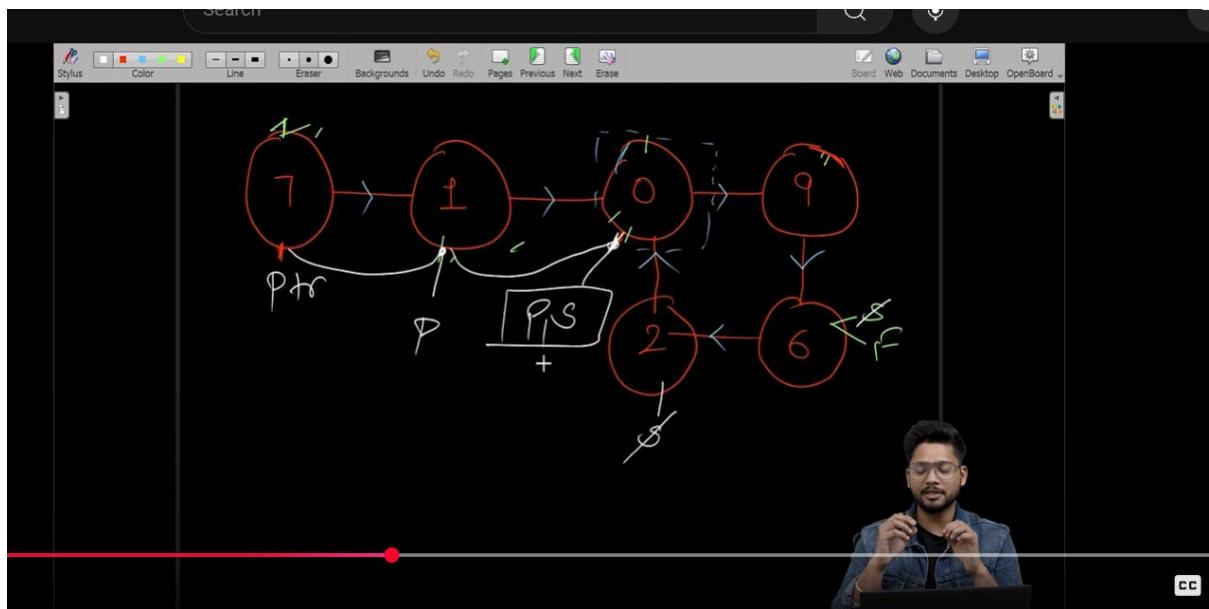
11. Delete a node in single linked list



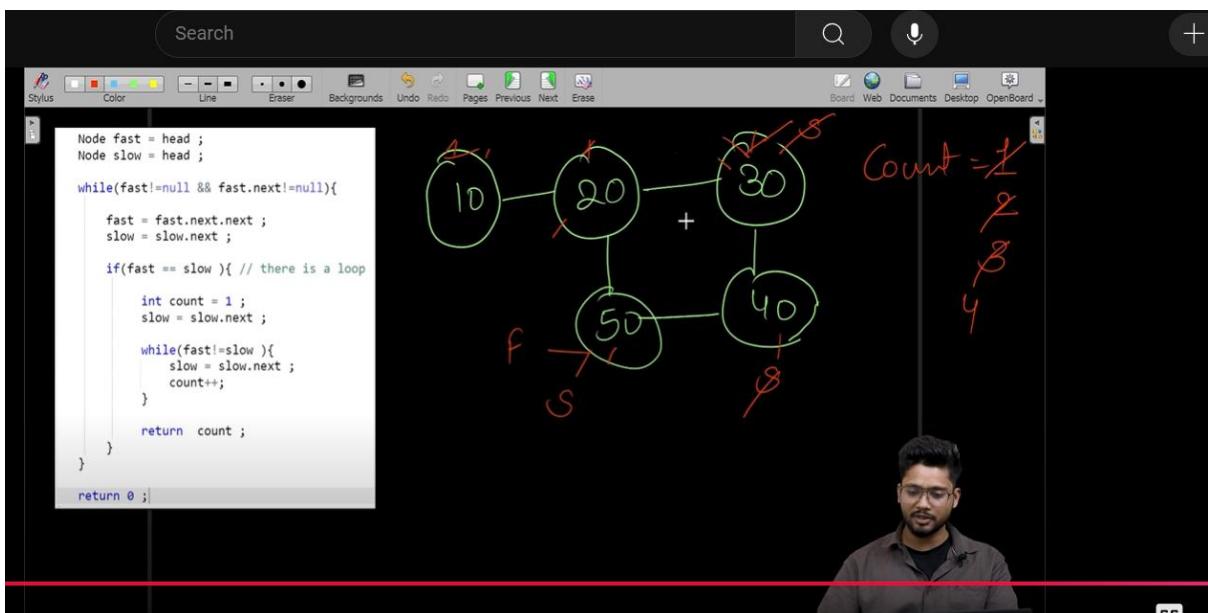
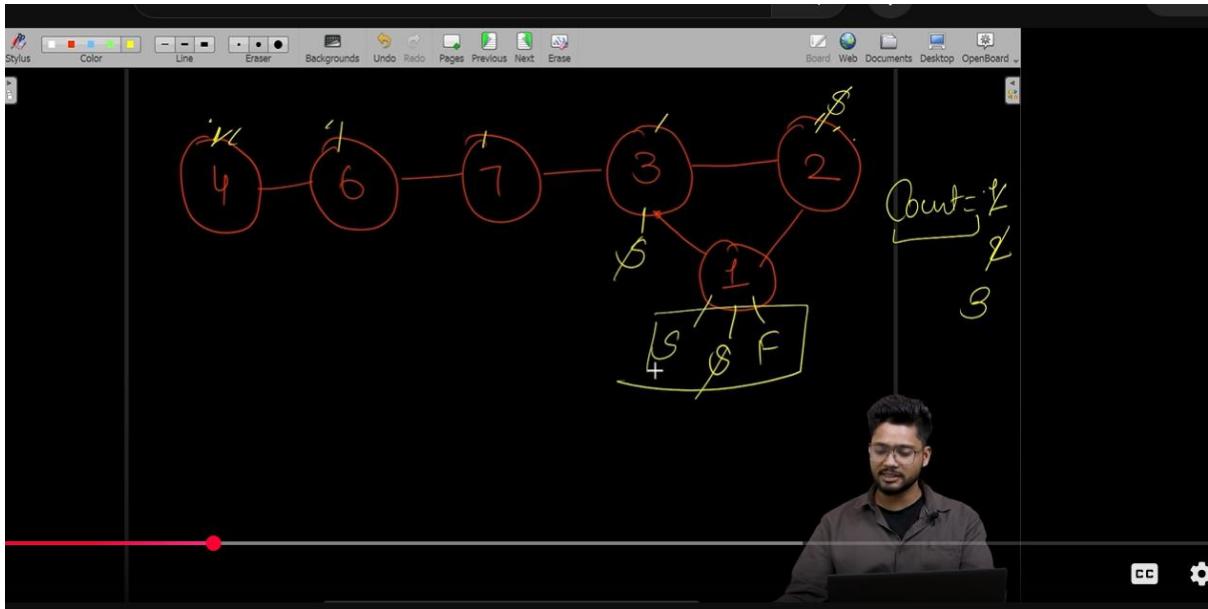
12. Detect loop in linked list



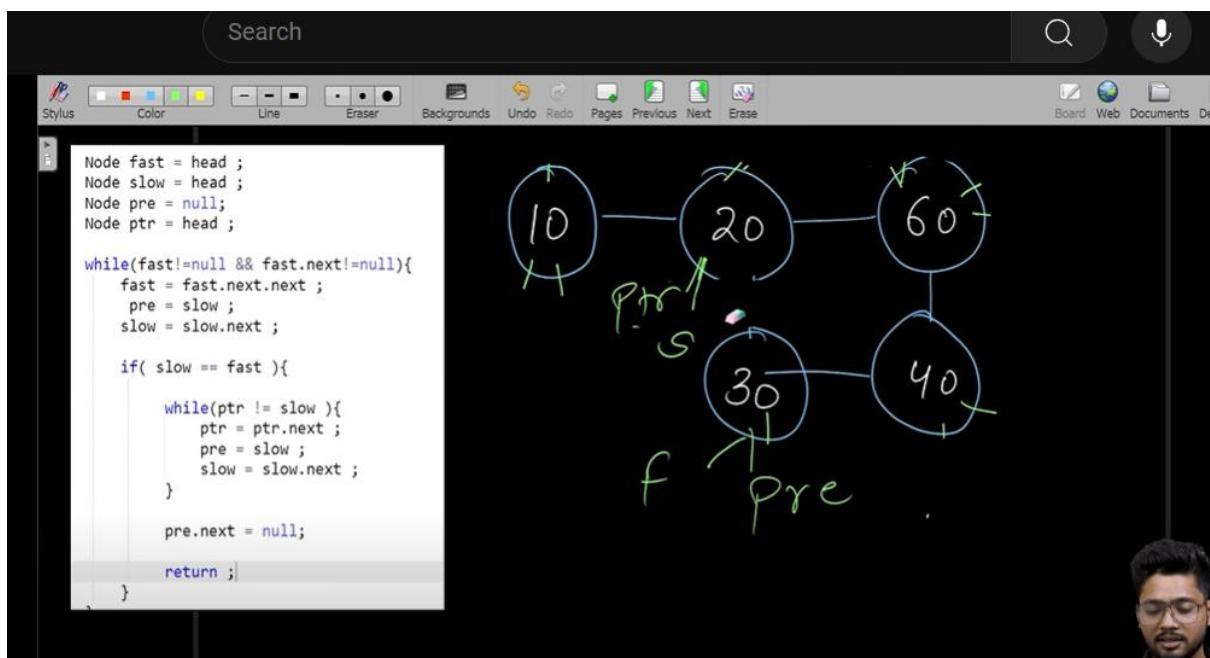
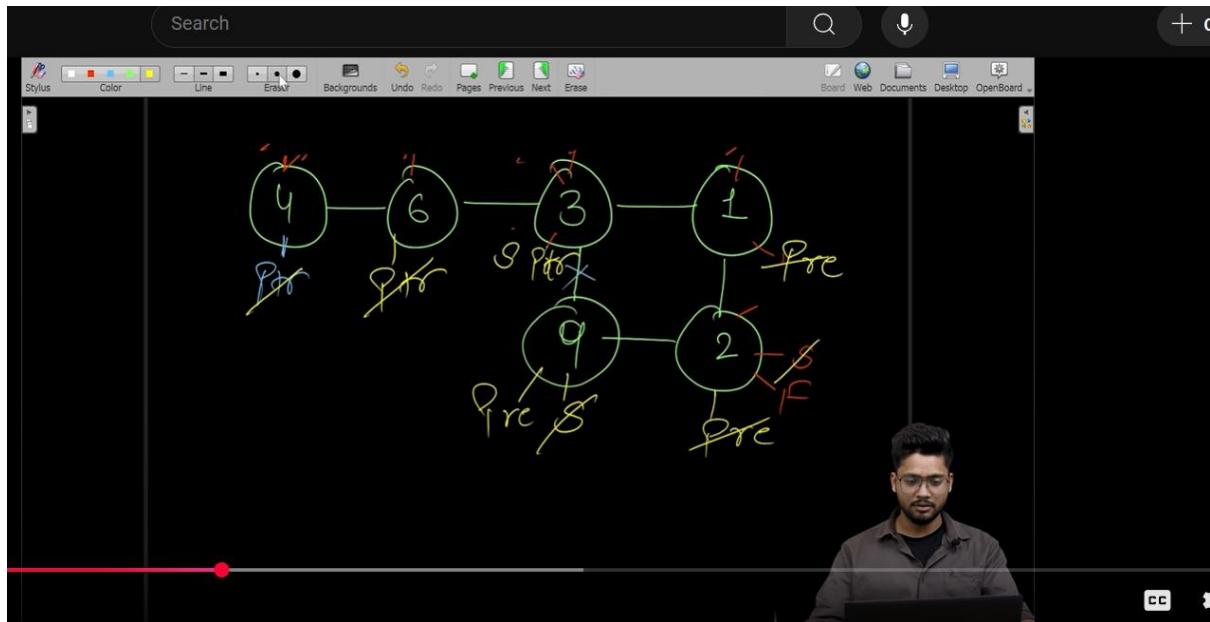
13. find the first node of loop in linked list



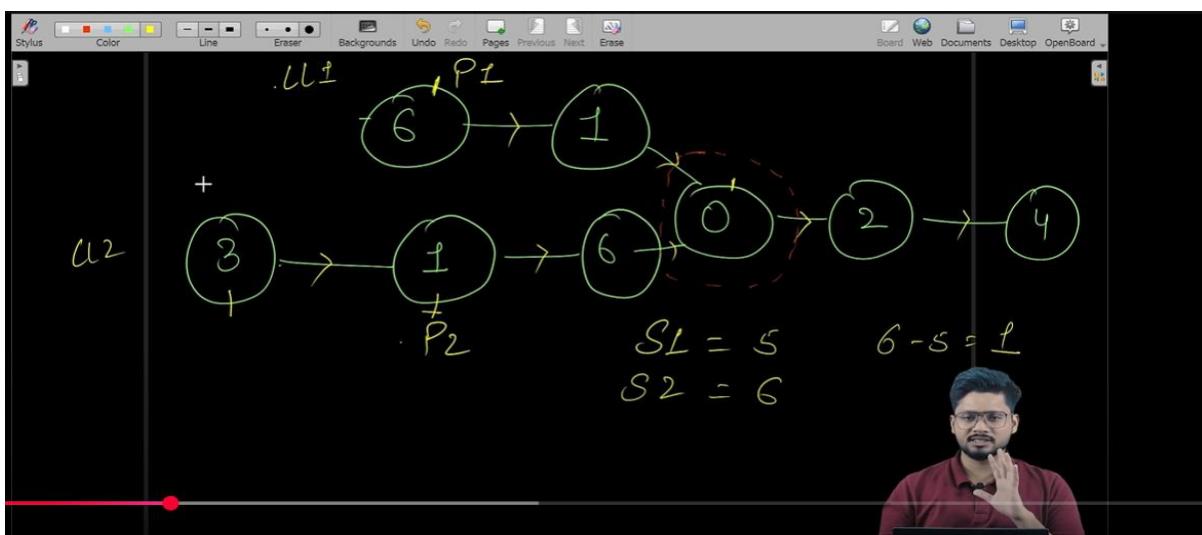
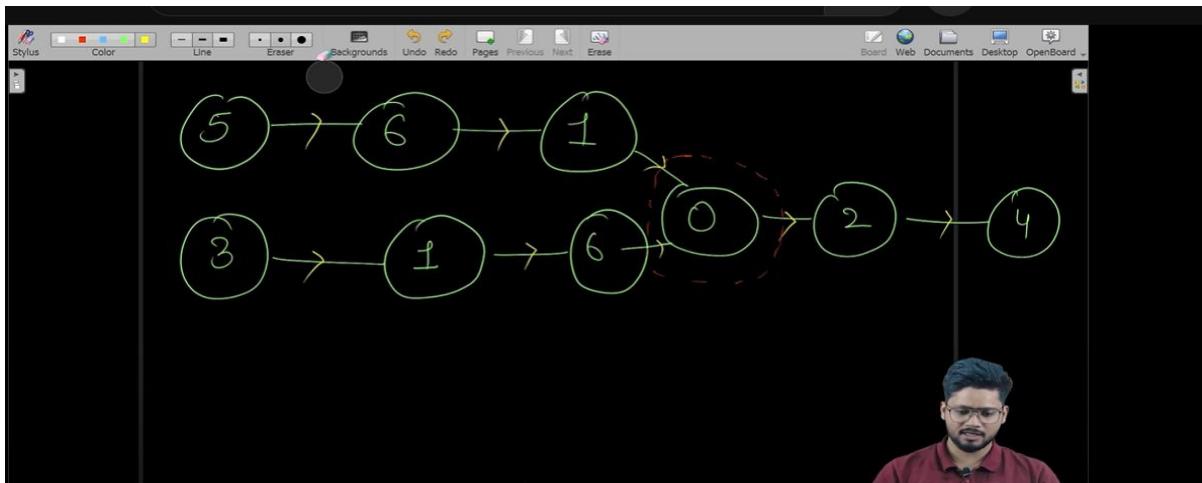
14 .Find the length of the loop in linked list



15. Remove loop in linked list



16. Intersection point in Y shaped linked list



```

int intersectPoint(Node head1, Node head2)
{
    if(head1 == null || head2 == null){
        return -1;
    }

    int s1 = sizeLL(head1);
    int s2 = sizeLL(head2);

    Node ptr1 = head1;
    Node ptr2 = head2;

    int diff = s1-s2;

    if(diff>0){ // head1 LL is larger
        while(diff>0){
            ptr1 = ptr1.next;
            diff--;
        }
    }else{ //head2 LL is larger
        while(diff<0){
            ptr2 = ptr2.next;
            diff++;
        }
    }

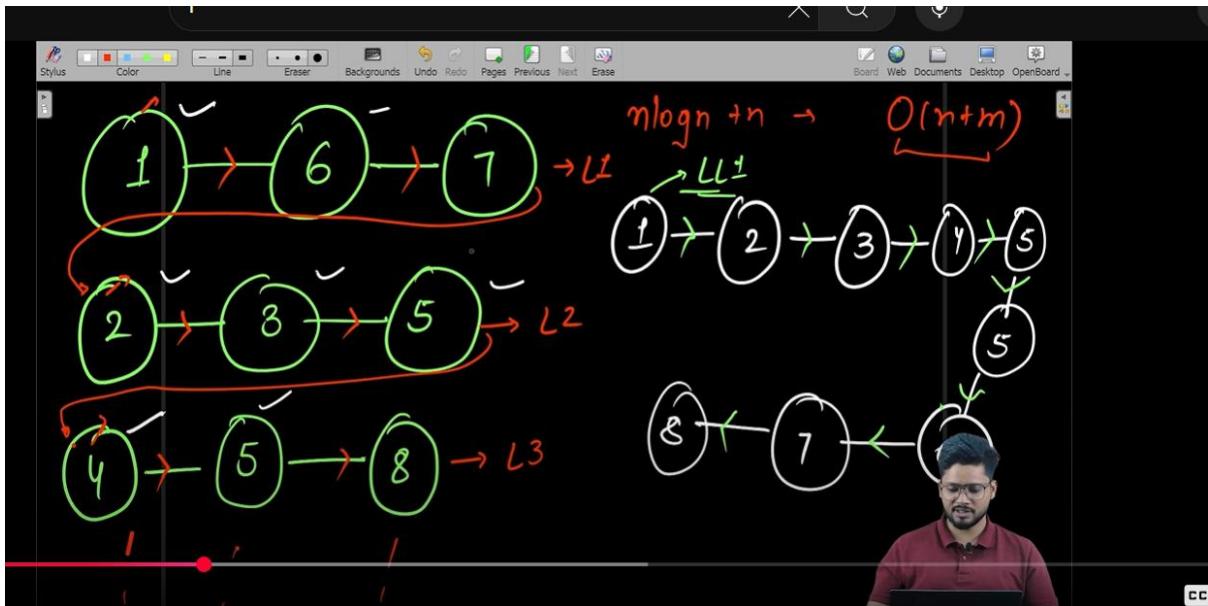
    while(ptr1!=null && ptr2!=null){

        if(ptr1 == ptr2){ //intersection point condition
            return ptr1.data;
        }

        ptr1 = ptr1.next;
        ptr2 = ptr2.next;
    }
}
return -1 ;

```

17. Merge k Sorted Linked List



Approach : In this, first we merge the first two arr into linked list and then for upcoming arr we use for loop from $i=2$, and most important is to use the exact code for (merge two sorted Linked list) ans we call that function(sortedMerge) when converted arr to linked node.

```
Node* mergeKList(Node** arr, int K)
{
    if(arr.length == 0) {
        return null;
    } else if (arr.length == 1) {
        return arr[0];
    } else if (arr.length == 2) {
        return sortedMerge(arr[0], arr[1]);
    }

    Node ans = sortedMerge(arr[0], arr[1]);

    for(int i=2;i<arr.length;i++){
        Node temp = arr[i];
        ans = sortedMerge(ans , temp);
    }

    return ans;
}
```

The diagram illustrates the merging process. It shows three initial lists: L1 (1, 3, 5), L2 (2, 4, 8), and L3 (5, 6, 7). The first two lists are merged into a new list (1, 2, 3, 5, 8). This result is then merged with the third list (5, 6, 7) to produce the final merged list (1, 2, 3, 4, 5, 6, 7, 8).

18. Rotate a Linked List

Input:

```
N = 5
value[] = {2, 4, 7, 8, 9} ✓
k = 3 ✓
Output: 8 9 2 4 7
```

Explanation:

```
Rotate 1: 4 -> 7 -> 8 -> 9 -> 2
Rotate 2: 7 -> 8 -> 9 -> 2 -> 4
Rotate 3: 8 -> 9 -> 2 -> 4 -> 7
```

Inhead

$R_1 \rightarrow$ (4) - (7) - (8) - (9) - (2)

$R_2 \rightarrow$ (7) - (8) - (9) - (2) - (4)

$R_3 \rightarrow$ (8) - (9) - (2) - (4) - (7)

A man in a maroon shirt is speaking.

Input:

```
N = 5
value[] = {2, 4, 7, 8, 9} ✓
k = 3 ✓
Output: 8 9 2 4 7
```

Explanation:

```
Rotate 1: 4 -> 7 -> 8 -> 9 -> 2
Rotate 2: 7 -> 8 -> 9 -> 2 -> 4
Rotate 3: 8 -> 9 -> 2 -> 4 -> 7
```

$R_1 \rightarrow [2, 4, 7, 8, 9]$

$R_{10} \rightarrow [2, 4, 7, 8, 9] \text{ ex=2}$

$R_{11} \rightarrow [4, 7, 8, 9, 2]$

$R_{12} \rightarrow [7, 8, 9, 2, 4]$

$R_1 \rightarrow [7, 8, 9, 2, 4]$

$R_2 \rightarrow [8, 9, 2, 4, 7]$

$R_3 \rightarrow [9, 2, 4, 7, 8]$

$k = 12 \rightarrow \frac{12}{5} \Rightarrow 2$

$\text{size} \rightarrow [4, 7, 8, 9]$

A man in a maroon shirt is speaking.

$ex = 2$

$ptr1$

$Inhead$

$tail$

(3) - (4) - (1) - (2)

A man in a maroon shirt is speaking.

GeeksforGeeks Practice Problem

Rotate a Linked List

[PUBLIC] [PUBLISHED]

Medium Accuracy: 39.95% Submissions: 209K+ Points: 4

Internship Alert!
Become an SDE Intern by topping this monthly leaderboard!

Given a singly linked list of size N. The task is to left-shift the linked list by k nodes, where k is a given positive integer smaller than or equal to length of the linked list.

Example 1:

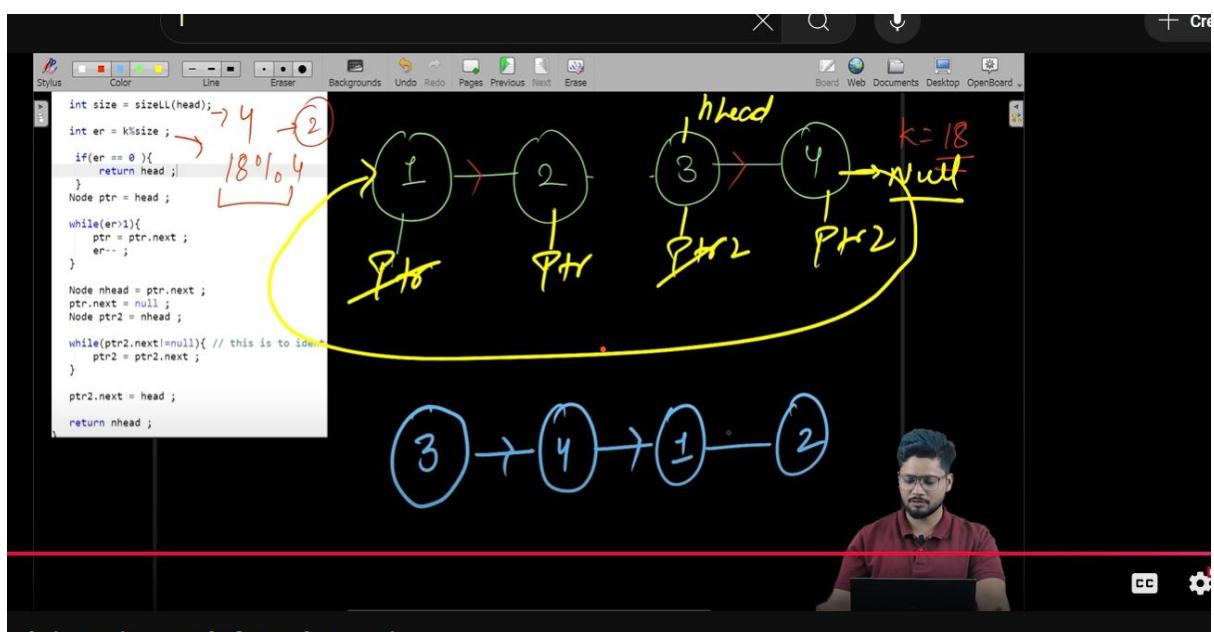
Input:
N = 5
value[] = {2, 4, 7, 8, 9}
k = 3

```

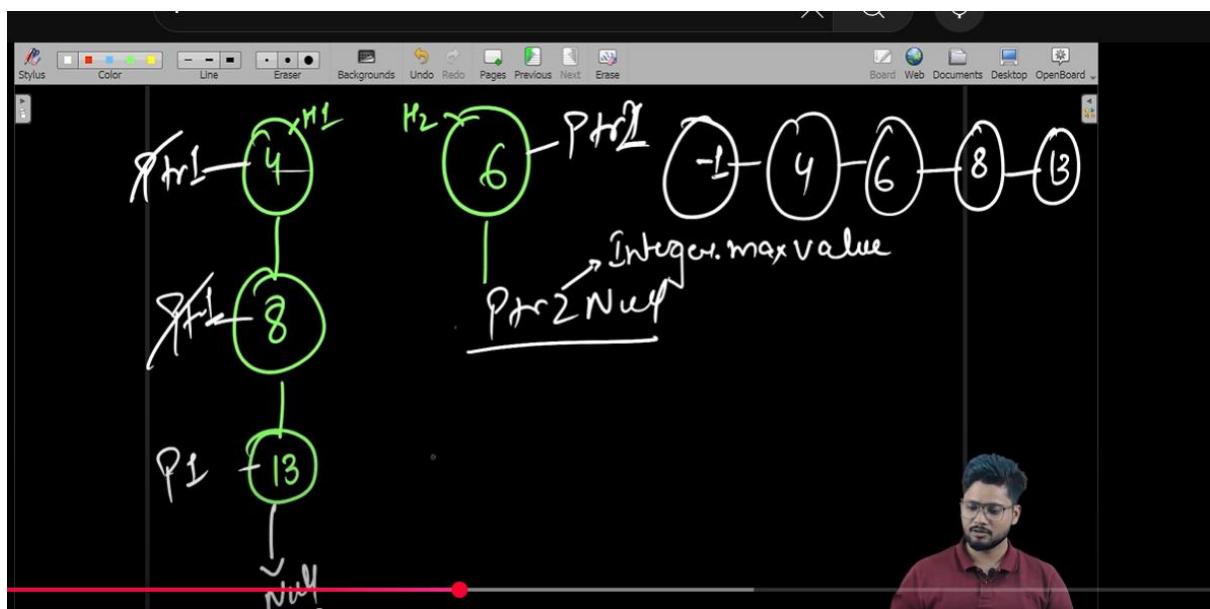
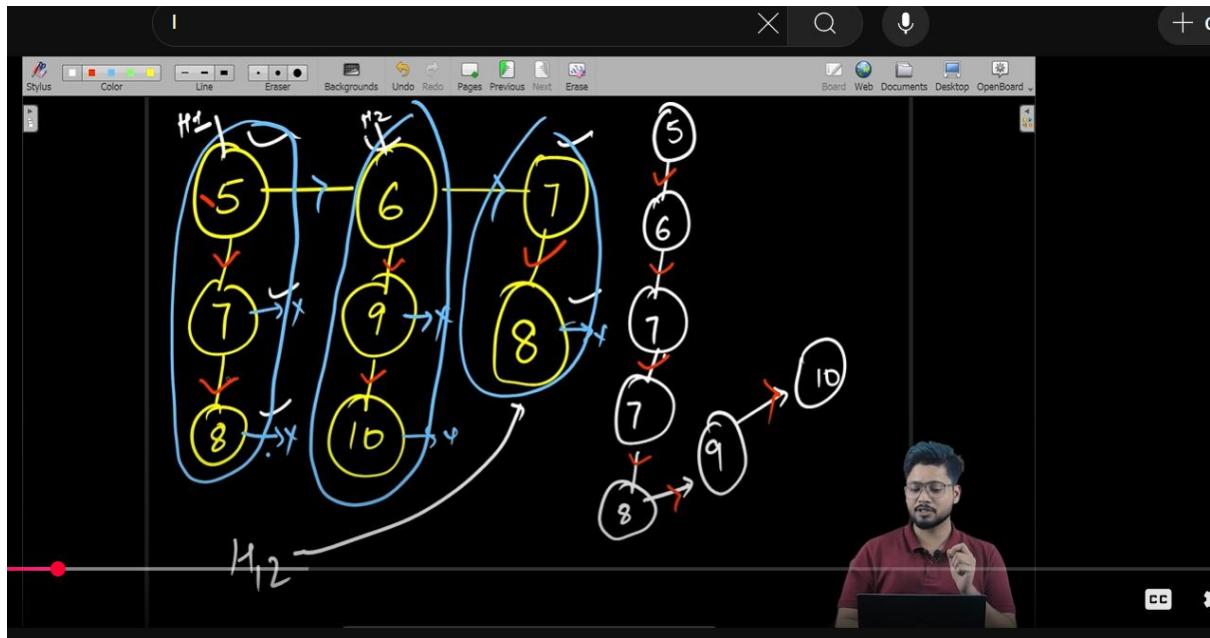
46+ class Solution{
47+     public int sizeLL(Node head){
48+         if(head == null){
49+             return 0 ;
50+         }
51+         Node curr = head ;
52+         int count = 0 ;
53+
54+         while(curr !=null){
55+             curr = curr.next ;
56+             count++;
57+         }
58+         return count ;
59+     }
60+     public Node rotate(Node head, int k) {
61+
62+         if(head == null){
63+             return null ;
64+         }else if ( k == 0 ){
65+             return head ;
66+         }
67+         int size = sizeLL(head);
68+
69+         int er = k%size ;
70+         if(er == 0 ){
71+             return head ;
72+         }
73+         Node ptr = head ;
74+
75+         while(er!=1){
76+             ptr = ptr.next ;
77+             er-- ;
78+         }
79+
80+         Node nhead = ptr.next ;
81+         ptr.next = null ;
82+         Node ptr2 = nhead ;
83+
84+         while(ptr2.next!=null){ // this is to ignore
85+             ptr2 = ptr2.next ;
86+         }
87+
88+         ptr2.next = head ;
89+
90+         return nhead ;
91+     }
92+ }

```

Count



19. Flattening a Linked List



```

Node mergeBottomSorted(Node head1 , Node head2){

    if(head1 == null || head2 == null){
        return head1==null?head2:head1;
    }

    Node ptr1 = head1 ;
    Node ptr2 = head2 ;
    Node dummy = new Node(-1);
    Node ans = dummy ;

    while(ptr1!=null || ptr2!=null){

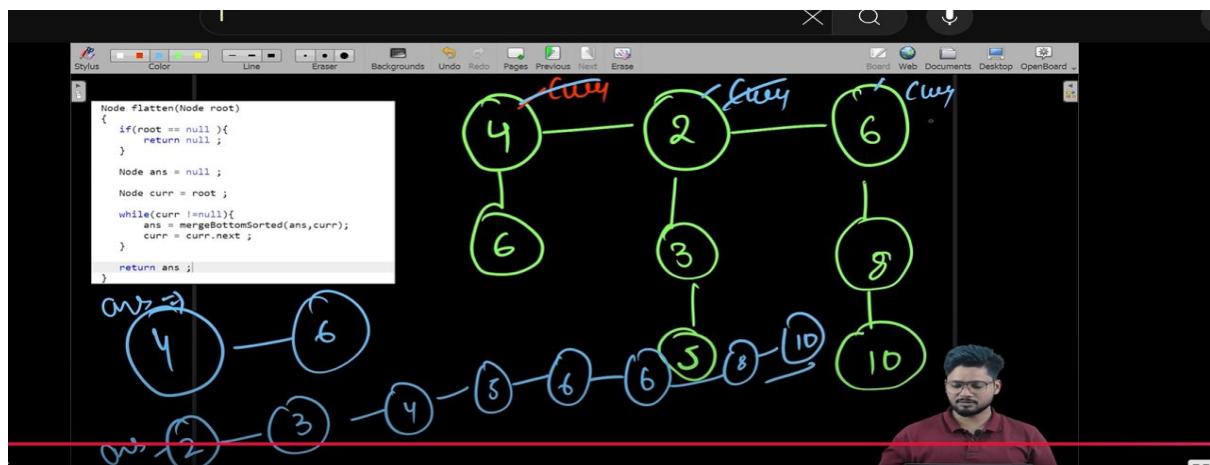
        int val1 = ptr1!=null ? ptr1.data : Integer.MAX_VALUE ;
        int val2 = ptr2!=null ? ptr2.data : Integer.MAX_VALUE ;

        if(val1<val2 ){
            dummy.bottom = ptr1 ;
            ptr1 = ptr1.bottom ;
        }else{
            dummy.bottom = ptr2 ;
            ptr2 = ptr2.bottom ;
        }

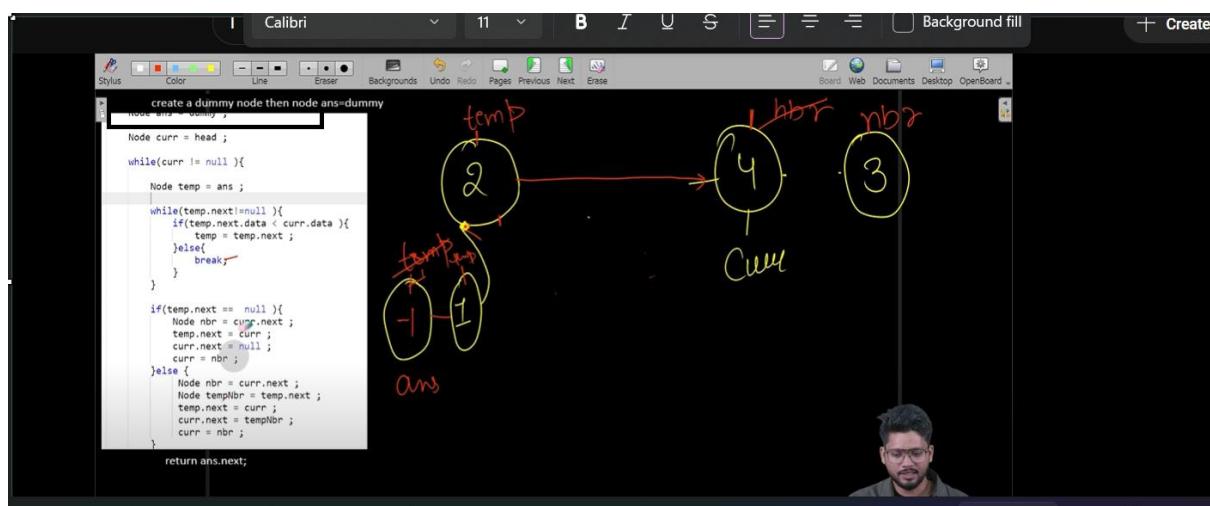
        dummy = dummy.bottom ;
    }

    return ans.bottom ;
}

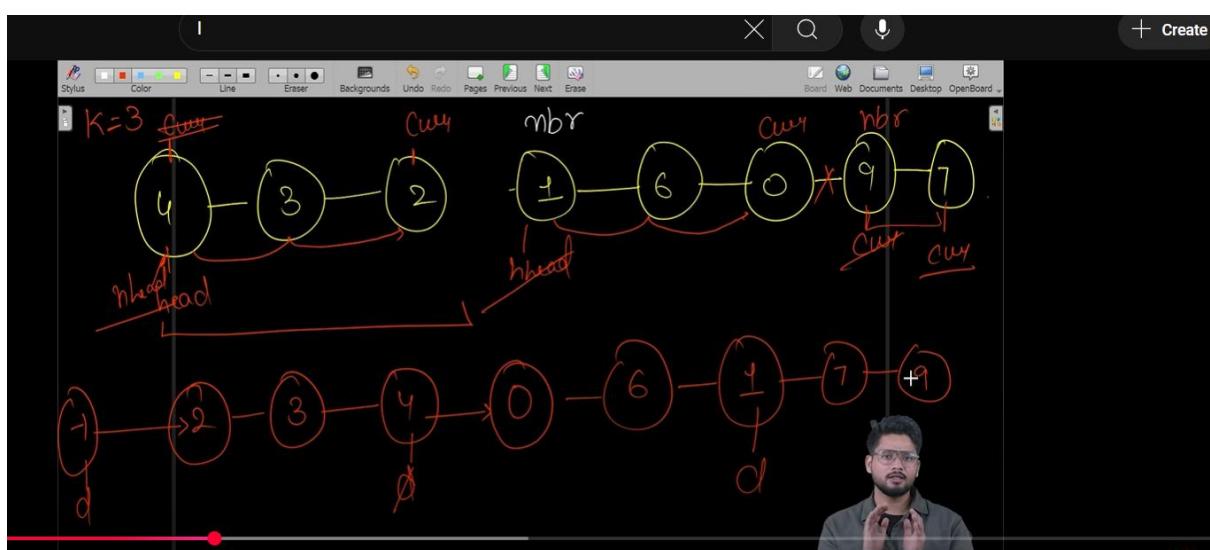
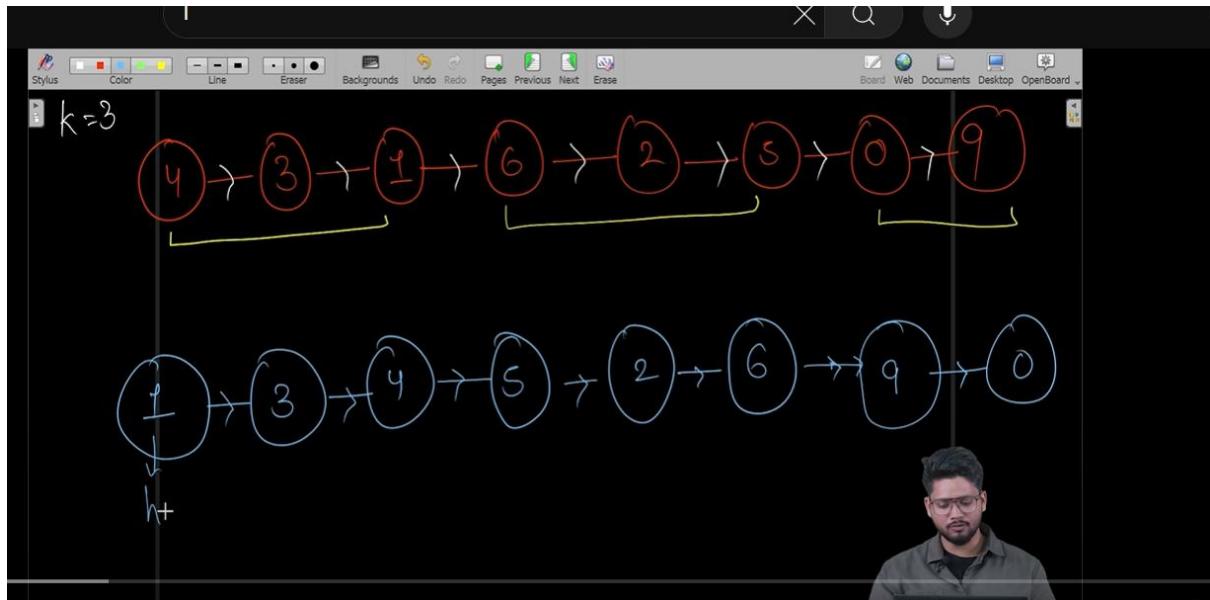
```



20. Insertion sort for Single linked list



21.Reverse a linked list in group size



A screenshot of a web-based programming environment showing a Java solution for reversing a linked list in groups of size k .

Code Snippet:

```
1 * // } Driver_Code_Ends
24 class Solution
25 {
26     public static Node rev(Node head){
27         if(head == null || head.next == null){
28             return head ;
29         }
30
31         Node curr = head ;
32         Node pre = null ;
33
34         while(curr!=null){
35             Node nbr = curr.next ;
36
37             curr.next = pre ;
38
39             pre = curr ;
40             curr = nbr ;
41
42         }
43         return pre ;
44     }
45
46     public static Node reverse(Node node, int k)
47     {
48
49
50
51
52
53
54
55
56
57
58
59
59 }
```

Problem Description:

Given a linked list of size N. The task is to reverse every k nodes (where k is an input to the function) in the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should be considered as a group and must be reversed (See Example 2 for clarification).

Example 1:

Input:

