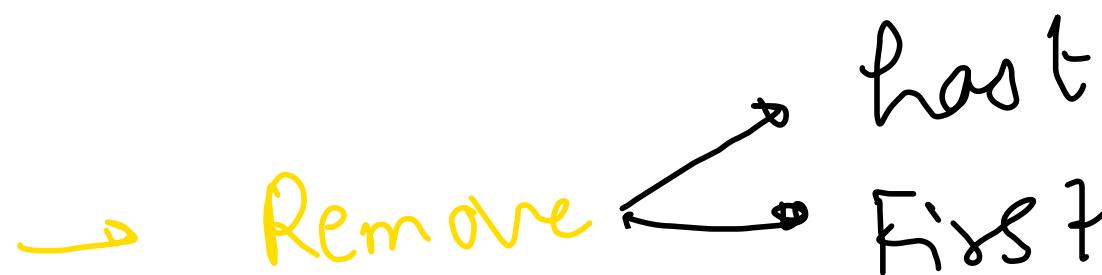
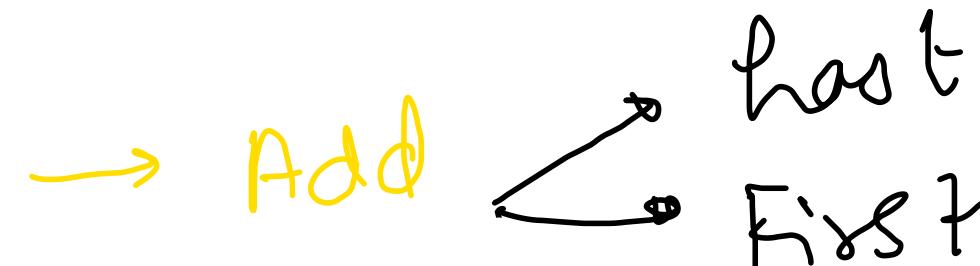


linked list (level 1 + level 2)

Lecture ① {Monday}

→ L2 Basics

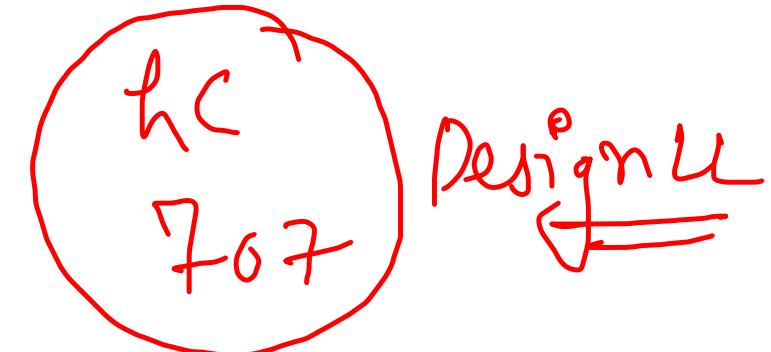
→ Display, Get At



Lecture ② {Tuesday}

→ Add At

→ Remove At



→ Reverse linked list

(C 206)

→ Data Iterative $O(n^2)$

→ Pointer Iterative $O(n)$

→ Data Recursive $O(N)$

✓ ↴ ↴ ↴
Local Return static

Lecture ③ { Wednesday }

→ Display Reverse

→ Reverse Lh

Pointer Recursive

→ K Reverse

LC HARD 25

→ Middle Node

LC 876

→ Kth node from end

GFG

→ Palindrome Lh

LC 234

Lecture ④

{ Thursday }

→ Merge Lh

→ merge 2 sorted

→ merge K sorted

→ merge Sort

LC 21

LC HARD 23

LC 148

→ Big integers

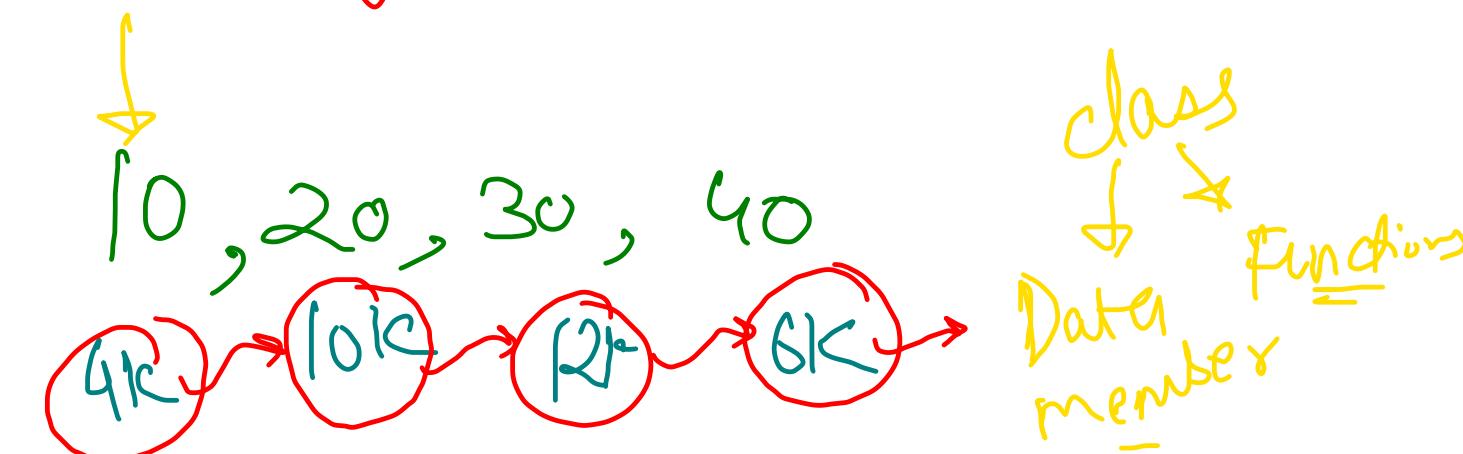
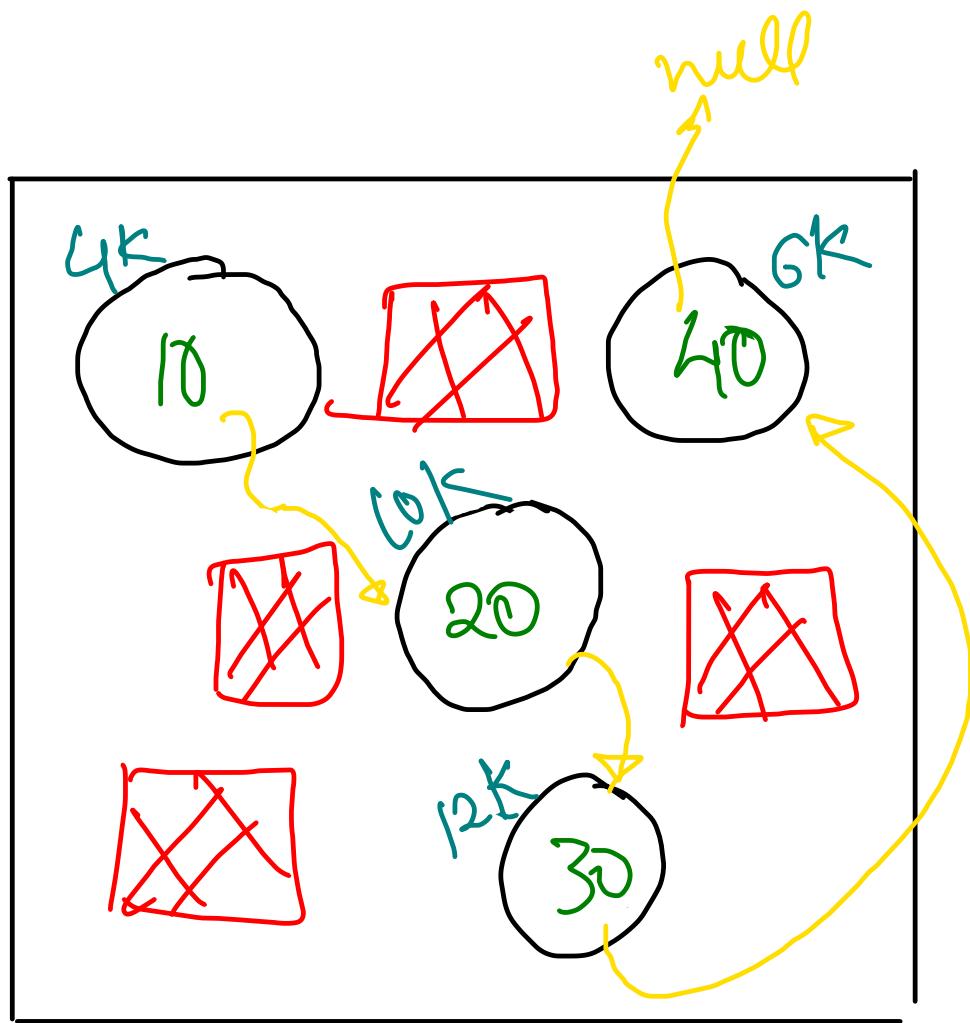
→ Addition of Lh

→ Subtract of Lh

Monday lecture 7 { 8:30 PM to 12 Am }

- Big Integers ↗ subtraction
multiplication
- Remove Duplicates (LC) 83 & 82 (18/1)
- Fold & unfold th (LC) 143 & Pepcoding
- Remaining Problems

Singly linked list :- Non contiguous memory allocation



public static Node {

int data;

Node next;

extra memory

public static
linked list

Node head;

Node tail;

int size;

class
↓
Data member
= functions

```
public static class Node{
    int data;
    Node next;
}
```

```
public static class LinkedList{
    Node head; Node tail; int size;

    public void display(){
    }

    public void addFirst(int val){
    }

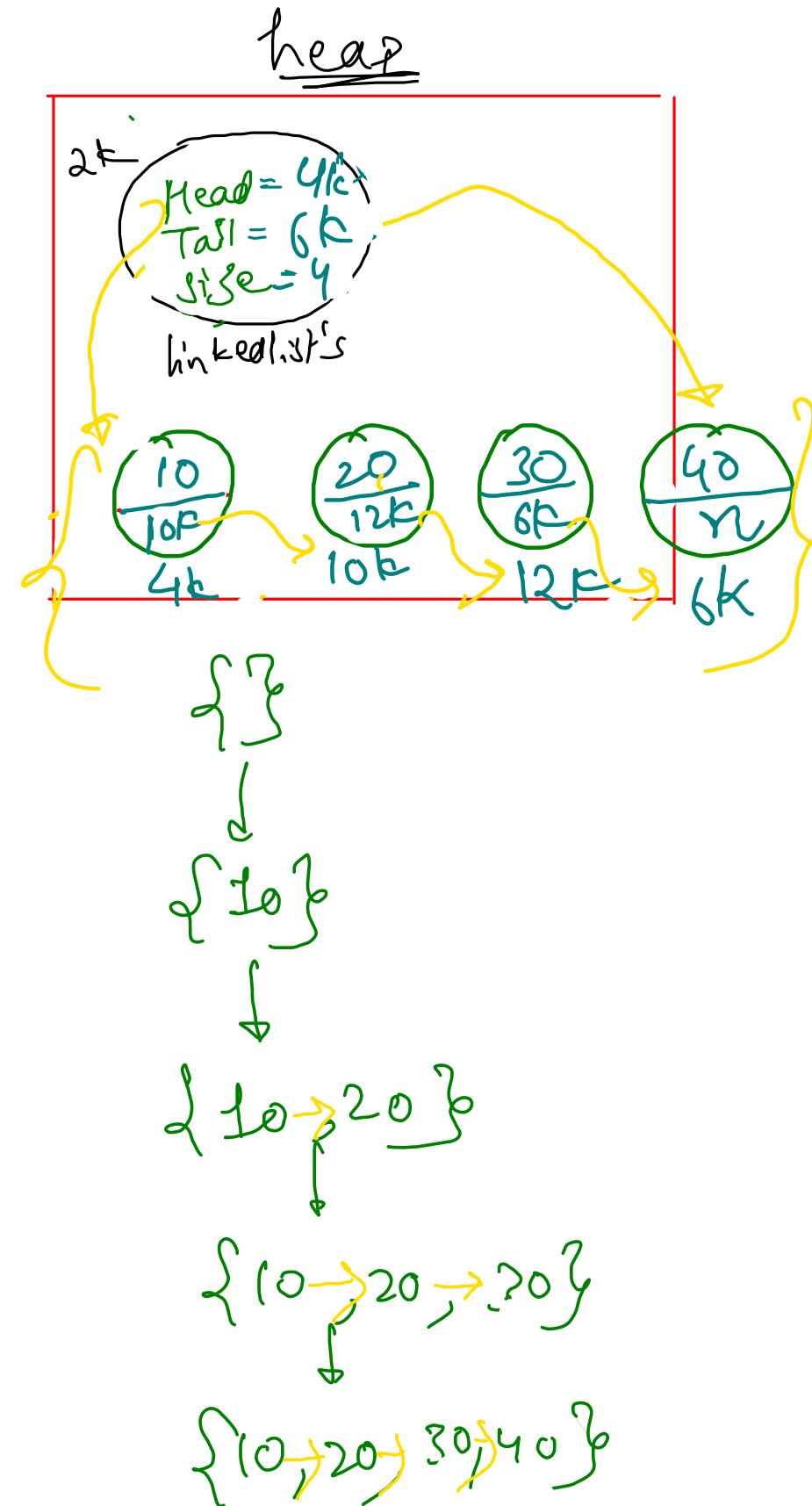
    public void addLast(int val){
    }

    public int get(int idx){
    }
}
```

Main() {
 → LinkedList list
 = new LinkedList();
 → list.addLast(10);
 → list.addLast(20);
 → list.addLast(30);
 → list.addLast(40);

Stack

list = 2F



{ }
 { 10 }
 { 10 → 20 }
 { 10 → 20 → 30 }
 { 10 → 20 → 30 → 40 }
 LL
 LL obj → (f + 8 + 4)
 = 20 bytes

ArrayList
 616 bytes
 vs
 2 nodes → (bytes + stack)
 > 4

```

public static class Node {
    int data;
    Node next;
}

public static class LinkedList {
    Node head;
    Node tail;
    int size;

    void addLast(int val) {
        // 1. Create a new Node
        Node temp = new Node();
        temp.data = val;

        if(size == 0){
            // new Node is the first as well as the last node
            head = temp;
            tail = temp;
        } else {
            tail.next = temp;
            tail = temp;
        }
        size++;
    }
}

```

You are sc

O(1)

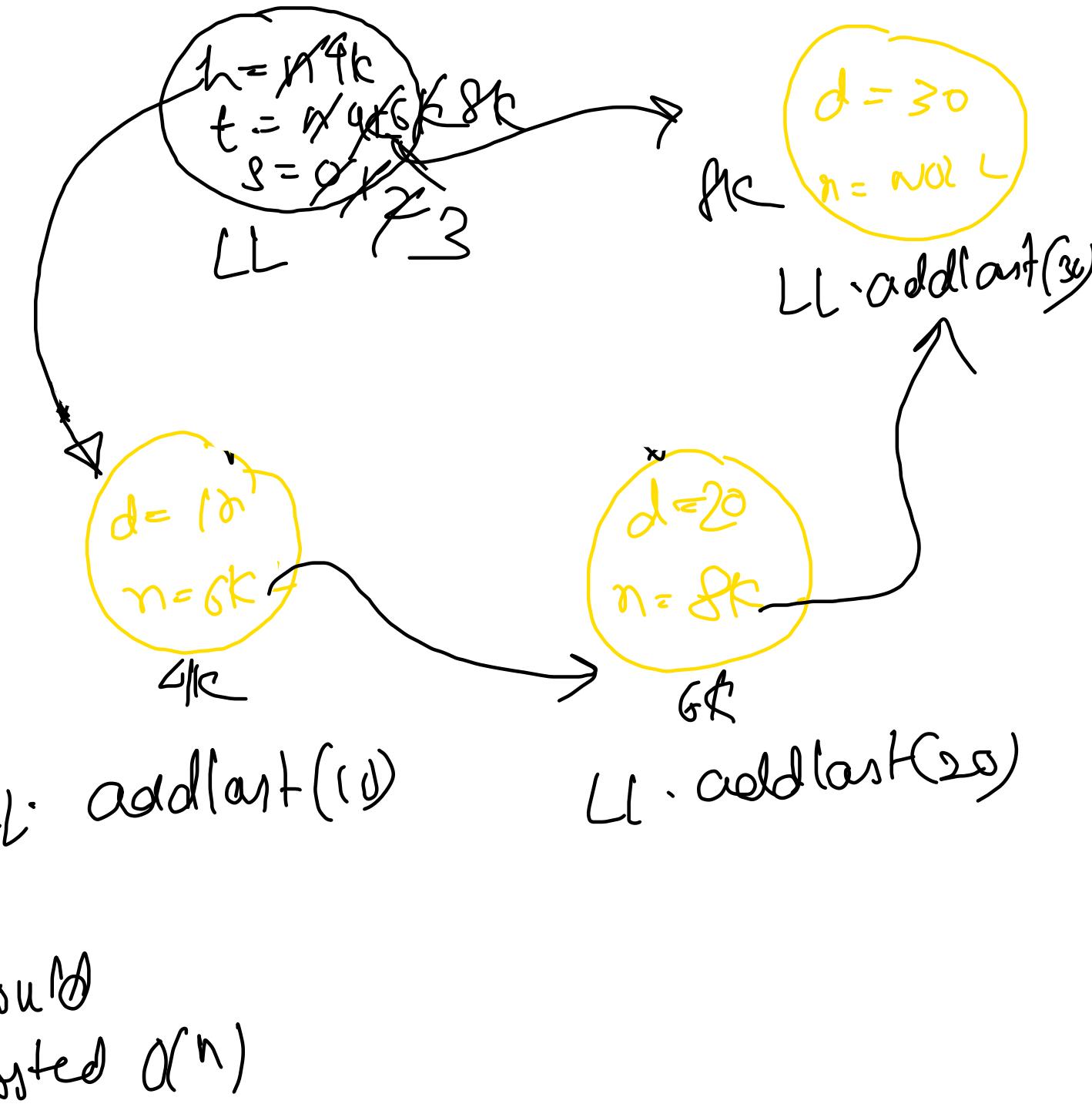
tail

O(1)

*if tail
way
not given
→ addLast*

*would
have costed O(n)*

```
LinkedList list = new LinkedList();
```



Disadvantages of linked list over Array

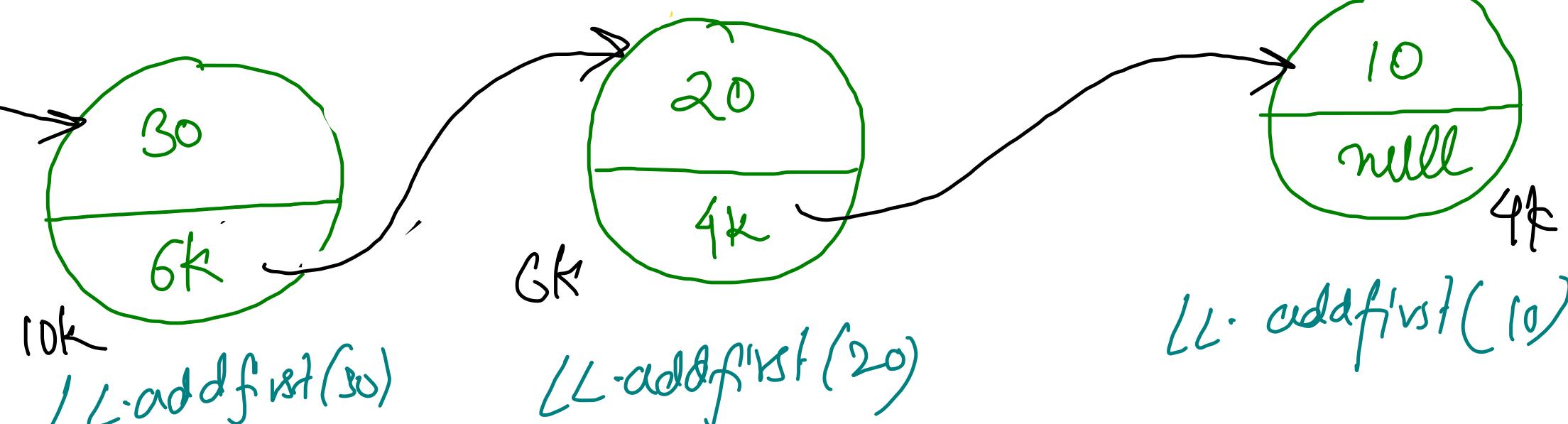
- ① → Extra space for storing 1 Node
- ② → Random Access not available.

Advantages of linked list over Array

- ① → Non-contiguous memory allocation
- ② → NO problem of fragmentation.

```
LinkedList list = new LinkedList();
```

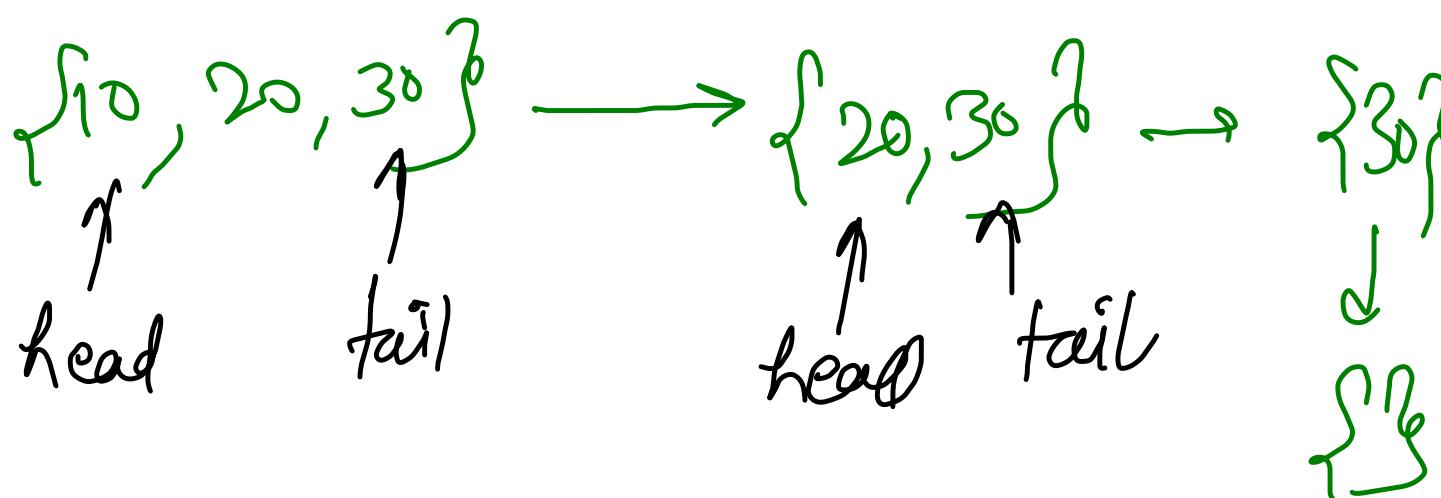
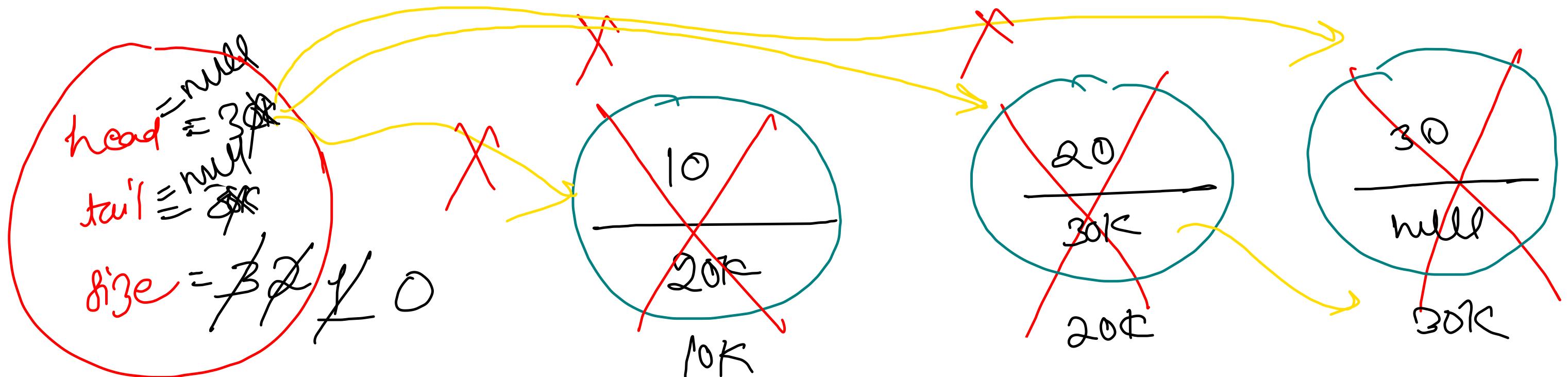
$$\begin{aligned} h &= 10k \\ t &= 4k \\ \text{size} &= 3 \end{aligned}$$



```
public void addFirst(int val) {
    // Creation of new Node
    Node temp = new Node();
    temp.data = val;

    if(size == 0){
        head = temp;
        tail = temp;
    } else {
        temp.next = head;
        head = temp;
    }

    size++;
}
```



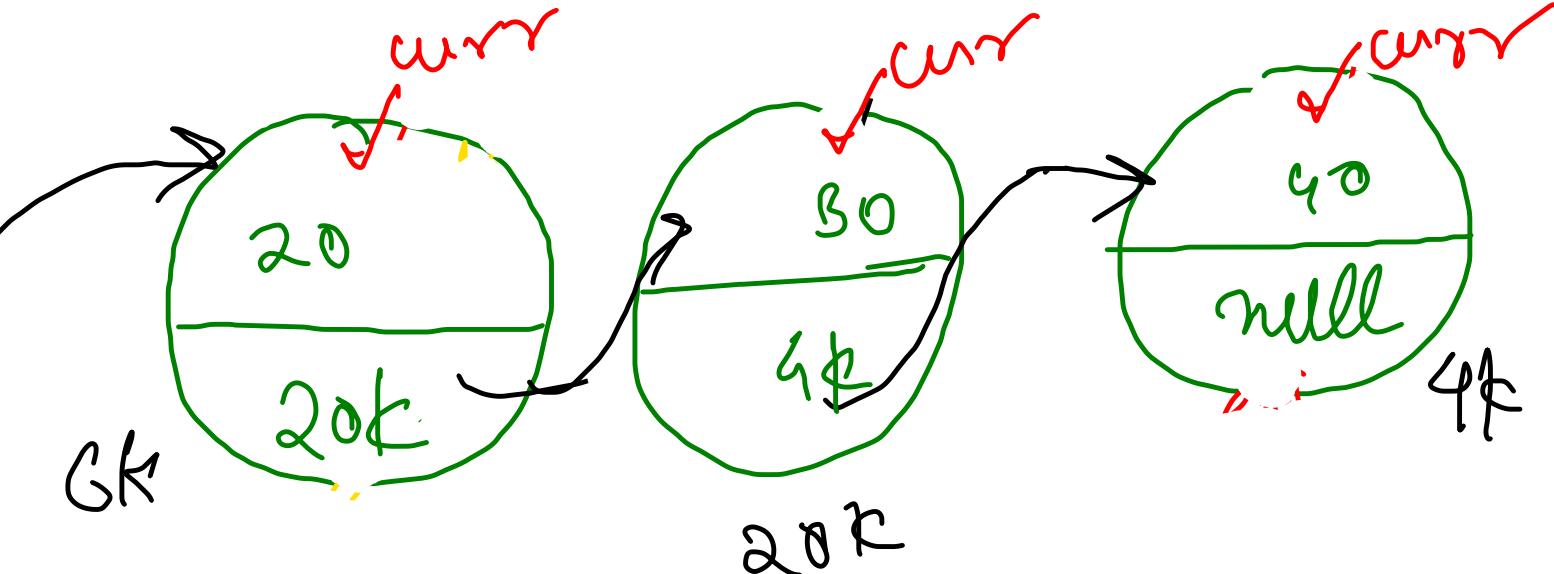
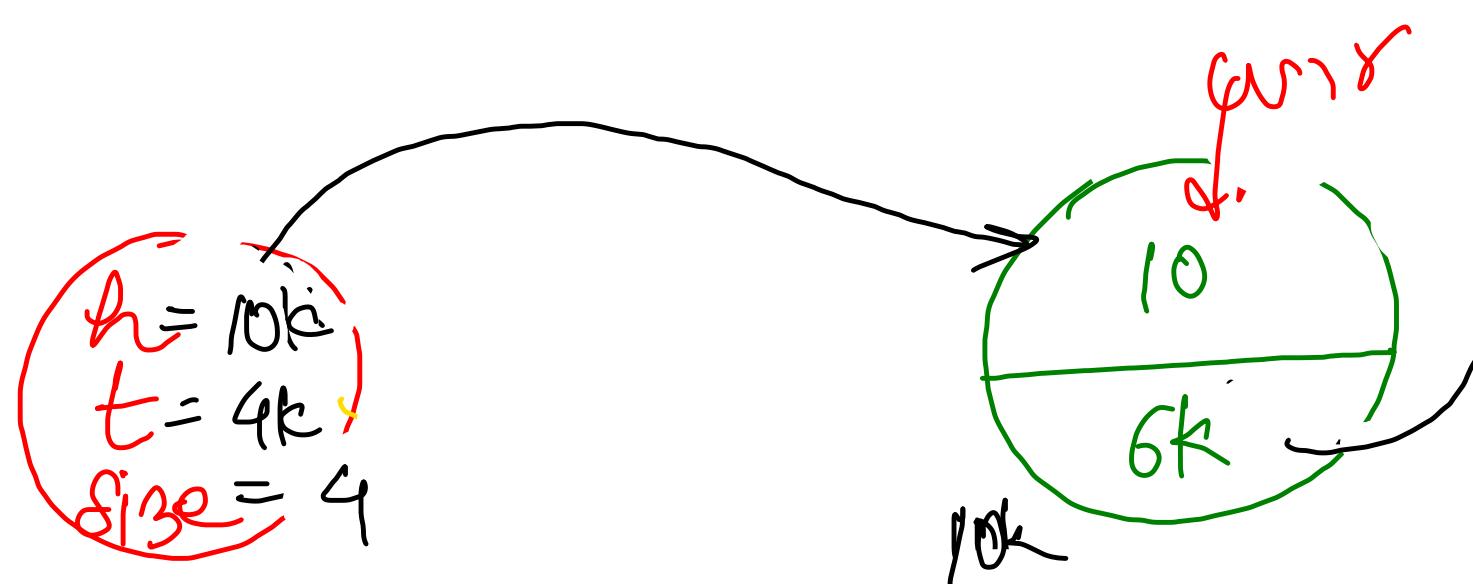
```

public void removeFirst(){
    if(size == 0) {
        System.out.println("List is empty");
        return;
    }

    if(size > 1){
        head = head.next;
    } else {
        head = tail = null;
    }
    size--;
}

```

O(1)



"Traversing/Displaying"

a
l
i
n

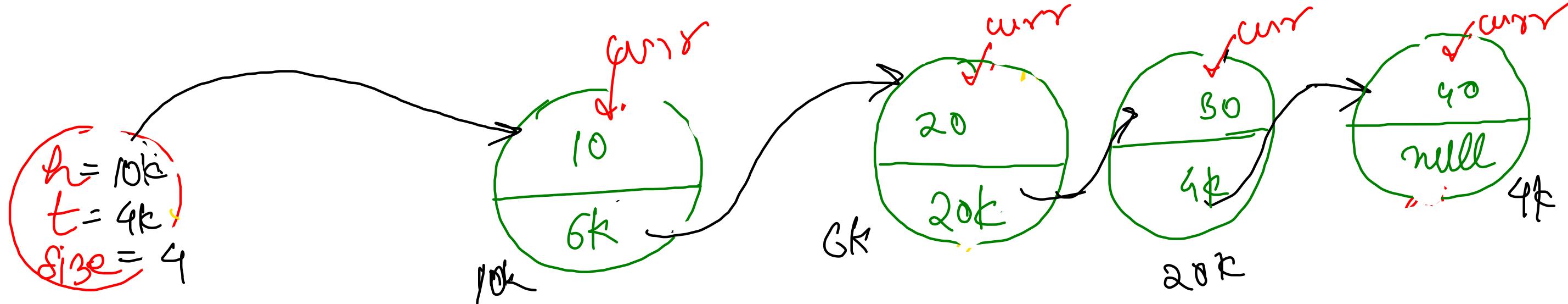
1st, 2nd, 3rd, 4th
 10, 20, 30, 40



```
for(Node curr = head; curr != null; curr = curr.next){
    System.out.print(curr.data + " ");
}
System.out.println();
```

```
public void display(){
    if(size == 0){
        return;
    }

    Node curr = head;
    while(curr != null){
        System.out.print(curr.data + " ");
        curr = curr.next;
    }
    System.out.println();
}
```



3.1. `getFirst` - Should return the data of first element. If empty should return -1 and print "List is empty".

3.2. `getLast` - Should return the data of last element. If empty should return -1 and print "List is empty".

3.3. `getAt` - Should return the data of element available at the index passed. If empty should return -1 and print "List is empty". If invalid index is passed, should return -1 and print "Invalid arguments".



```
public int getFirst(){
    if(size == 0){
        System.out.println("List is empty");
        return -1;
    }

    return head.data;
}

public int getLast(){
    if(size == 0){
        System.out.println("List is empty");
        return -1;
    }

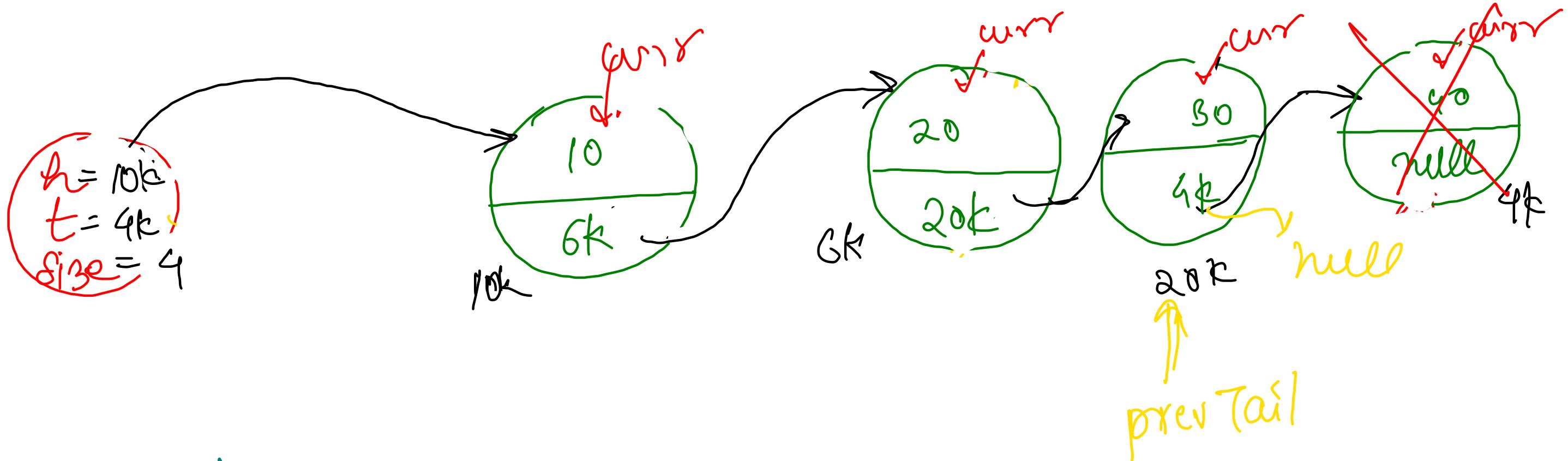
    return tail.data;
}

public int getAt(int idx){
    if(size == 0){
        System.out.println("List is empty");
        return -1;
    }
    if(idx < 0 || idx >= size){
        System.out.println("Invalid arguments");
        return -1;
    }

    if(idx == 0) return getFirst();
    if(idx == size - 1) return getLast();

    Node curr = head;
    for(int i=0; i<idx; i++){
        curr = curr.next;
    }
    return curr.data;
}
```

$O(N)$



Remove last

~~if size > 1~~

Node prevTail =

prevTail.next = null;

tail = prevTail;

getAt(size - 2);

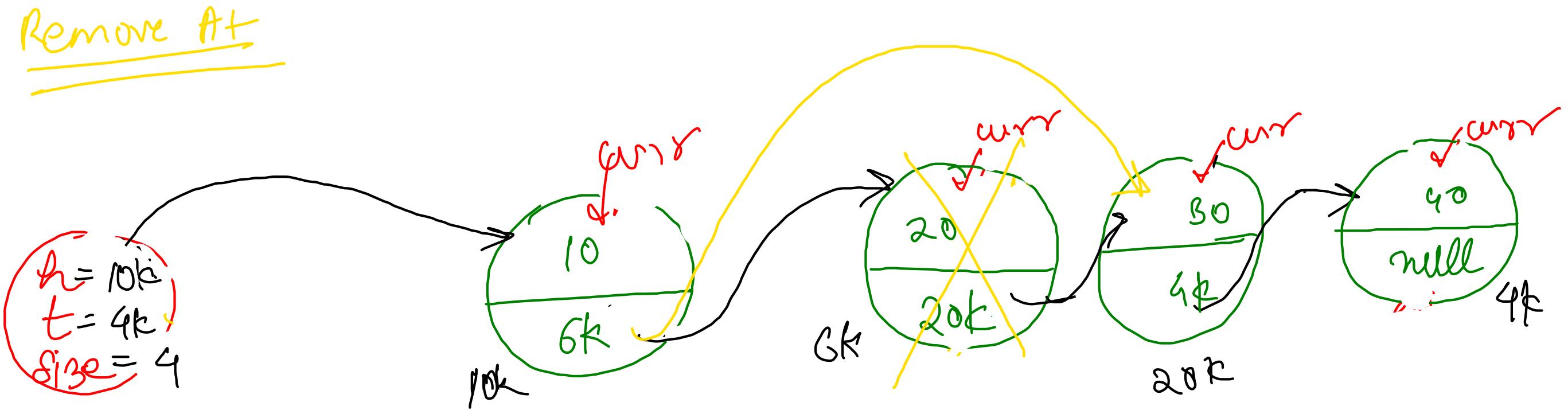
```
public void removeLast(){
    if(size == 0){
        System.out.println("List is empty");
        return;
    }

    if(size == 1){
        head = tail = null;
    } else {
        Node prevTail = head;
        for(int i=0; i<size-2; i++){
            prevTail = prevTail.next;
        }

        prevTail.next = null;
        tail = prevTail;
    }

    size--;
}
```

$\rightarrow O(N)$



① remove At(0) → removeFirst()

② remove At(size-1) → removeLast()

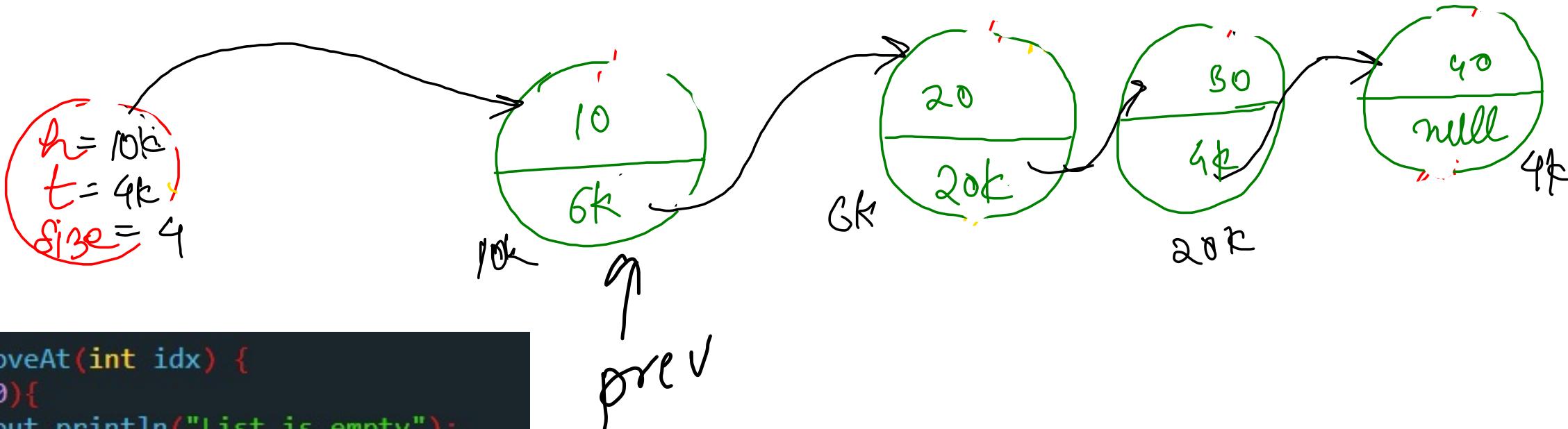
③ remove At(idx)

$\{ 0 \leq idx < size - 1 \}$

-
- ① Get $idx-1$ th node $O(N)$
- ② $prev.next = prev.next.next$
- ③ $size--$

$idx \begin{cases} \leftarrow i-1 & \text{if } i \leq size \\ \geq size & \text{if } i > size \end{cases}$

$size == 0 \} \text{ List is empty}$



```

public void removeAt(int idx) {
    if(size == 0){
        System.out.println("List is empty");
        return;
    }

    if(idx < 0 || idx >= size){
        System.out.println("Invalid arguments");
        return;
    }

    if(idx == 0) removeFirst();
    else if(idx == size - 1) removeLast();
    else {
        Node prev = head;
        for(int i=0; i<idx-1; i++){
            prev = prev.next;
        }

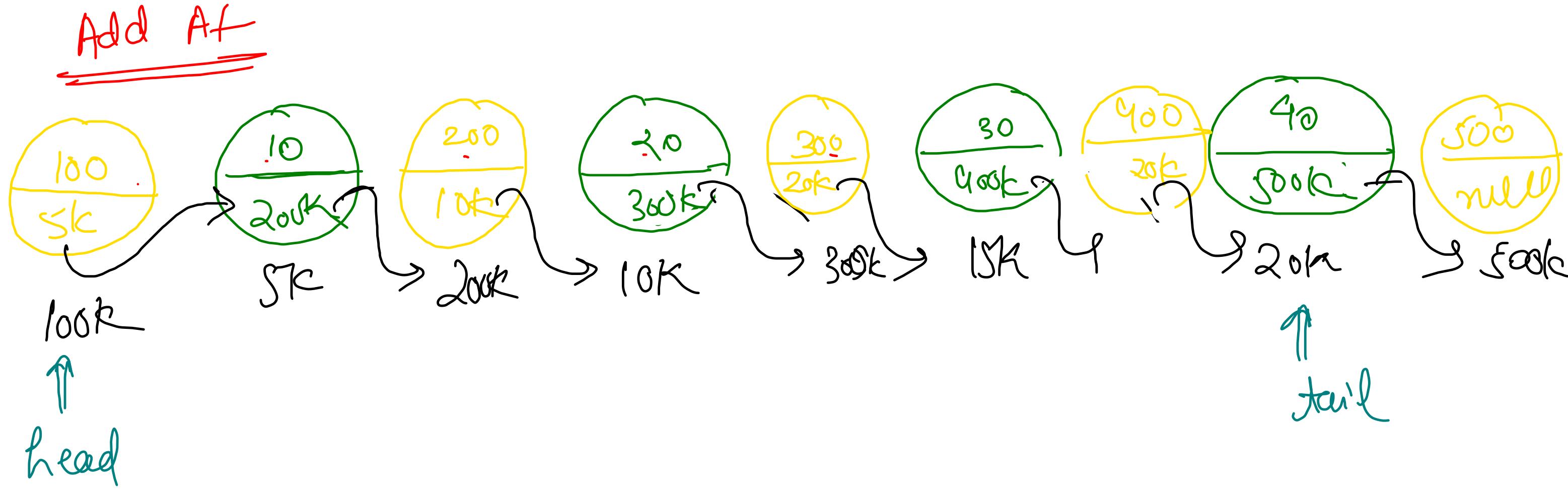
        prev.next = prev.next.next;
        size--;
    }
}

```

LL.remove At(2)

worst
 $O(N)$

$O(N)$
In worst case



Add Af(0, 100) \Rightarrow add First(100);

Add Af(2, 200)

Add Af(4, 300)

AddAf(i , x) \Rightarrow Add at Second last
 \uparrow
size-1

AddAf(i , x) \nrightarrow Add at last
 \downarrow
size

```
public void addAt(int idx, int val){  
    if(idx < 0 || idx > size){  
        System.out.println("Invalid arguments");  
        return;  
    }  
  
    if(idx == 0) addFirst(val);  
    else if(idx == size) addLast(val);  
    else {  
        Node temp = new Node();  
        temp.data = val;  
  
        Node prev = head;  
        for(int i=0; i<idx-1; i++){  
            prev = prev.next;  
        }  
  
        temp.next = prev.next;  
        prev.next = temp;  
        size++;  
    }  
}
```

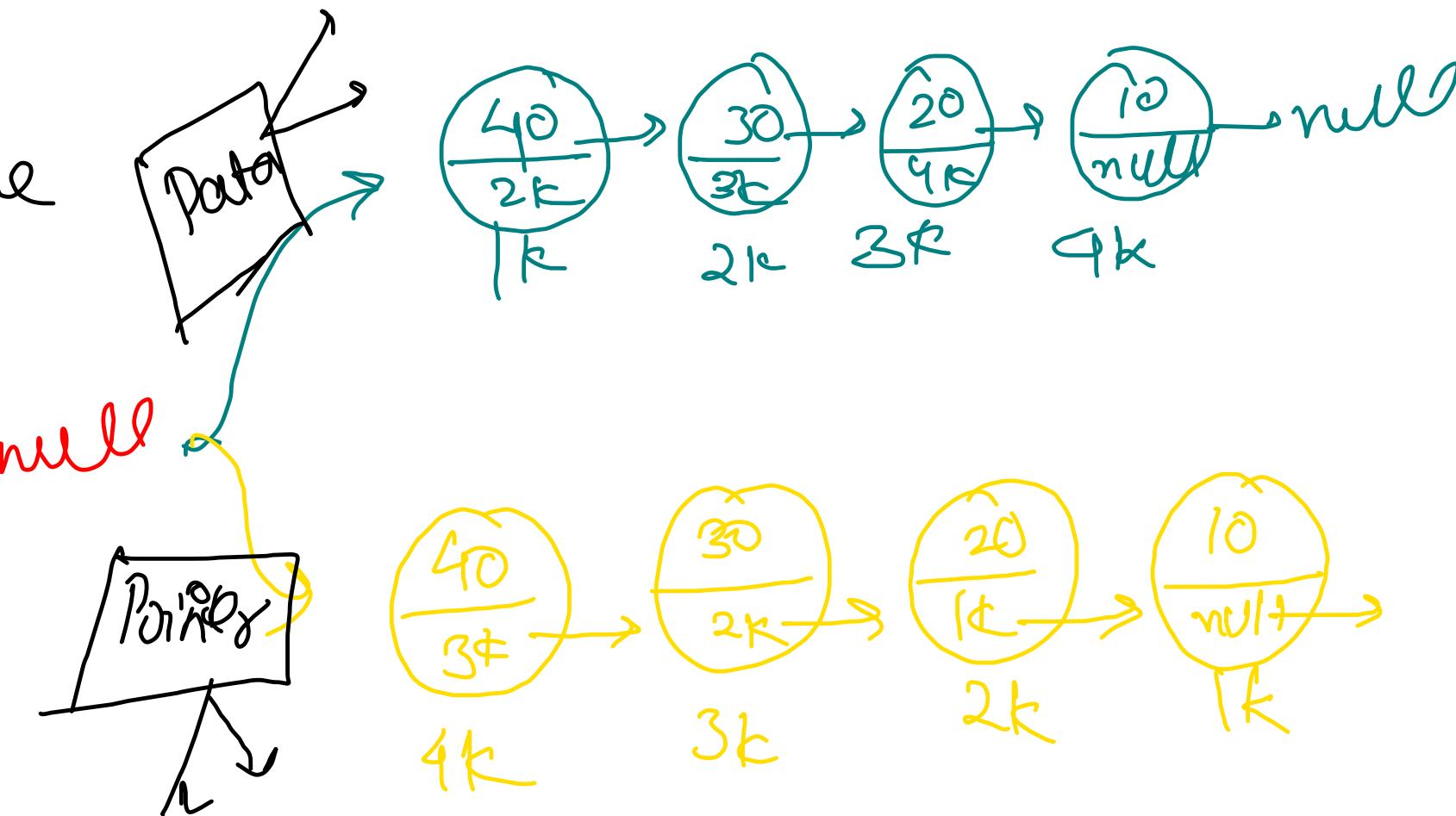
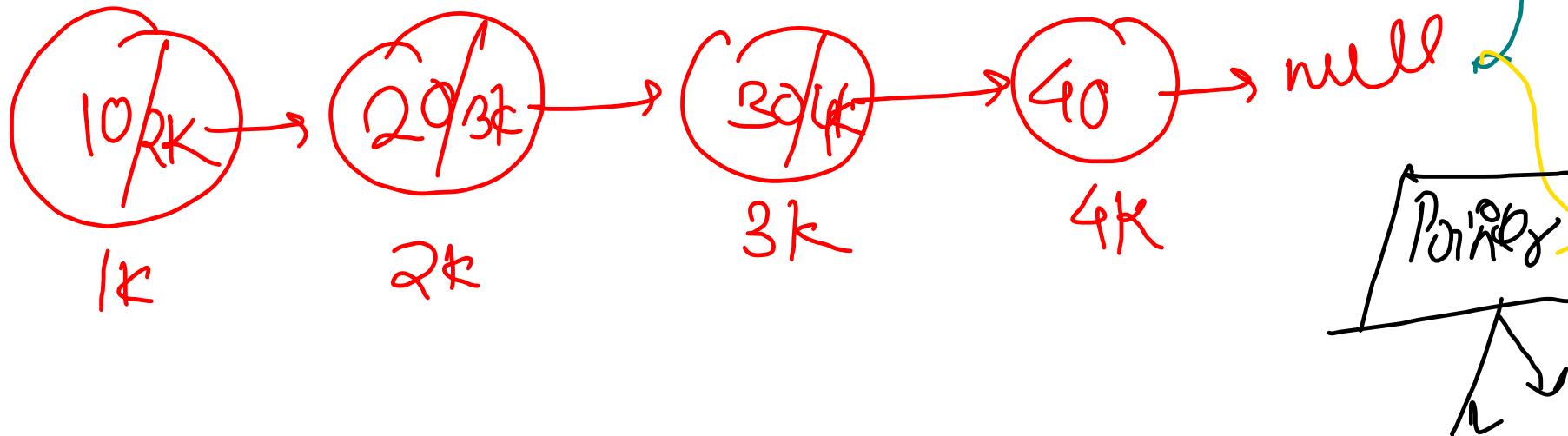
Reverse a linked list

→ ① Data - Iterative

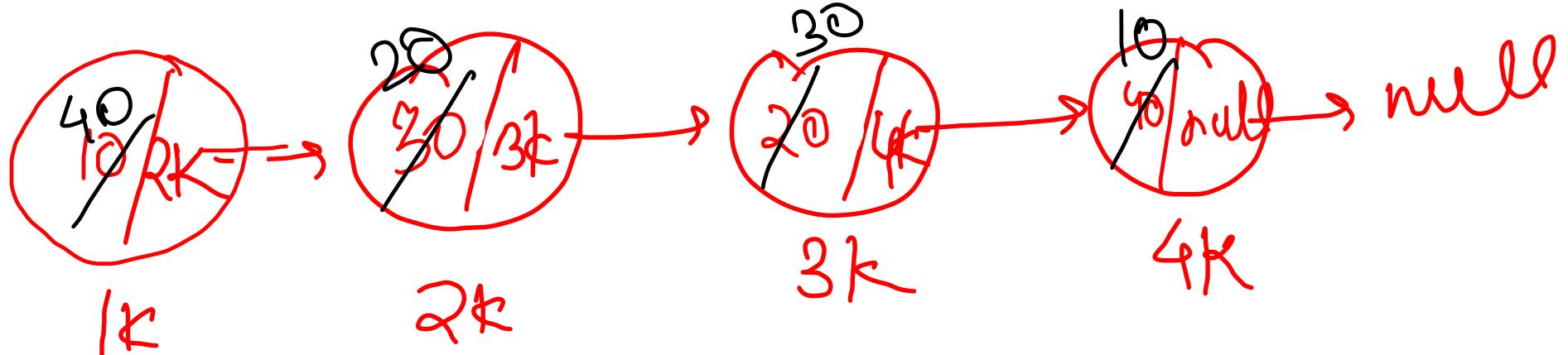
② Pointer - Iterative

③ Data - Recursive

④ Pointer - Recursive



Data Iterative



↑
right
↓
left

```
while(left < right){
    leftNode = getAT(left)
    rightNode = getAT(right)
    ... O(N)
    left++ + right--}
```

② left = getAT(1)
right = getAT(n-2)
⇒ O(N)

}

$$\text{Time Comp} \Rightarrow \frac{N}{2} * O(N) = O(N^2)$$

```

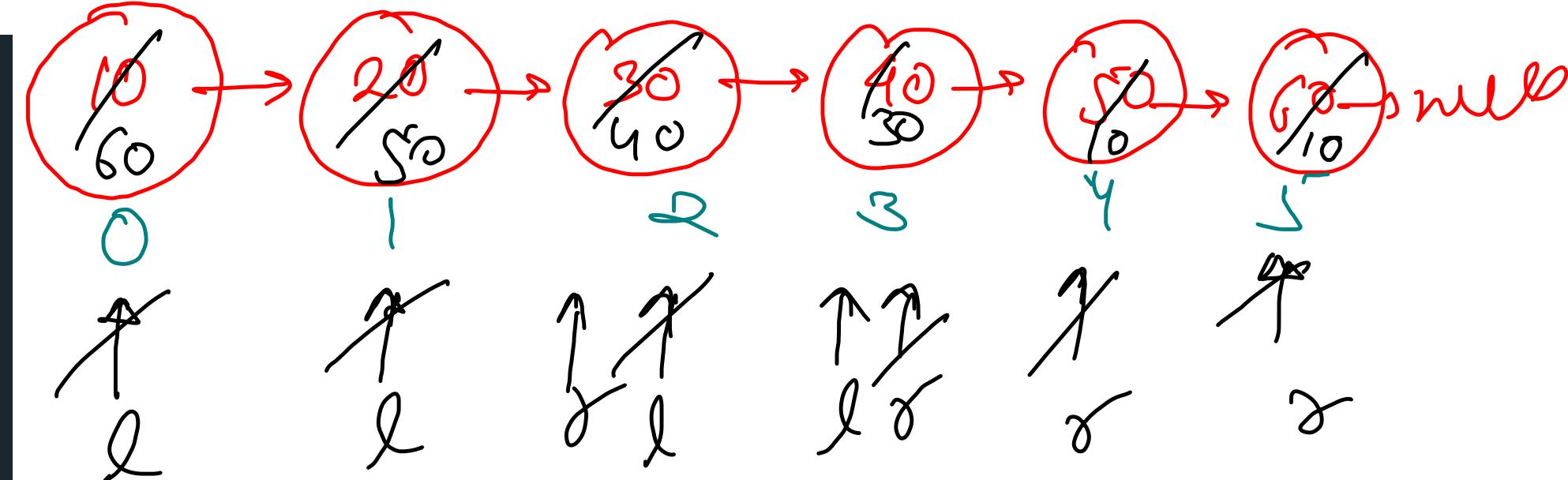
public Node getNodeAt(int idx){
    Node curr = head;
    for(int i=0; i<idx; i++){
        curr = curr.next;
    }
    return curr;
}

public void swap(Node left, Node right){
    int leftData = left.data;
    left.data = right.data;
    right.data = leftData;
}

public void reverseDI() {
    int left = 0, right = size - 1;
    while(left < right){
        Node leftNode = getNodeAt(left);
        Node rightNode = getNodeAt(right);

        swap(leftNode, rightNode);
        left++; right--;
    }
}

```

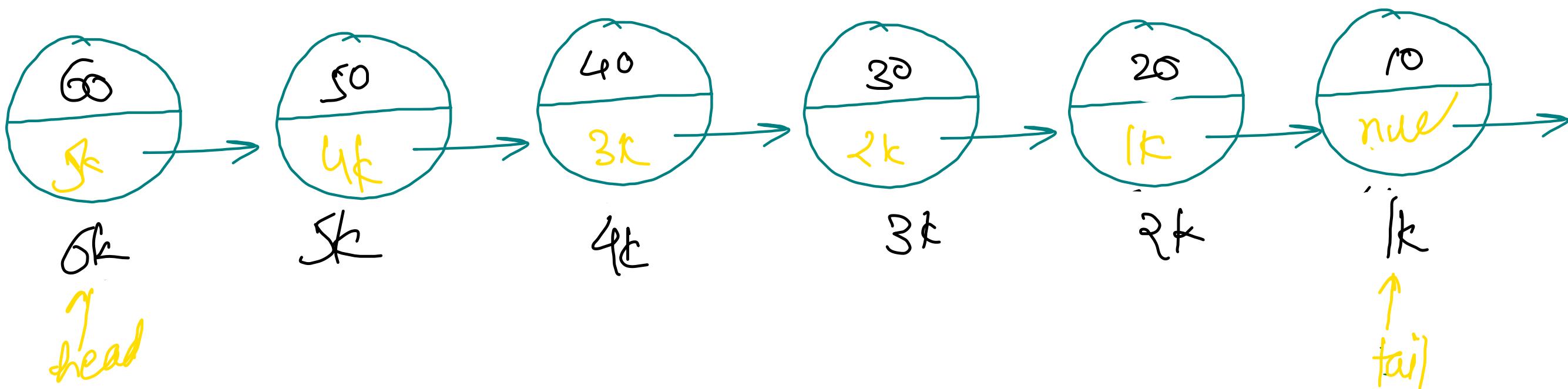
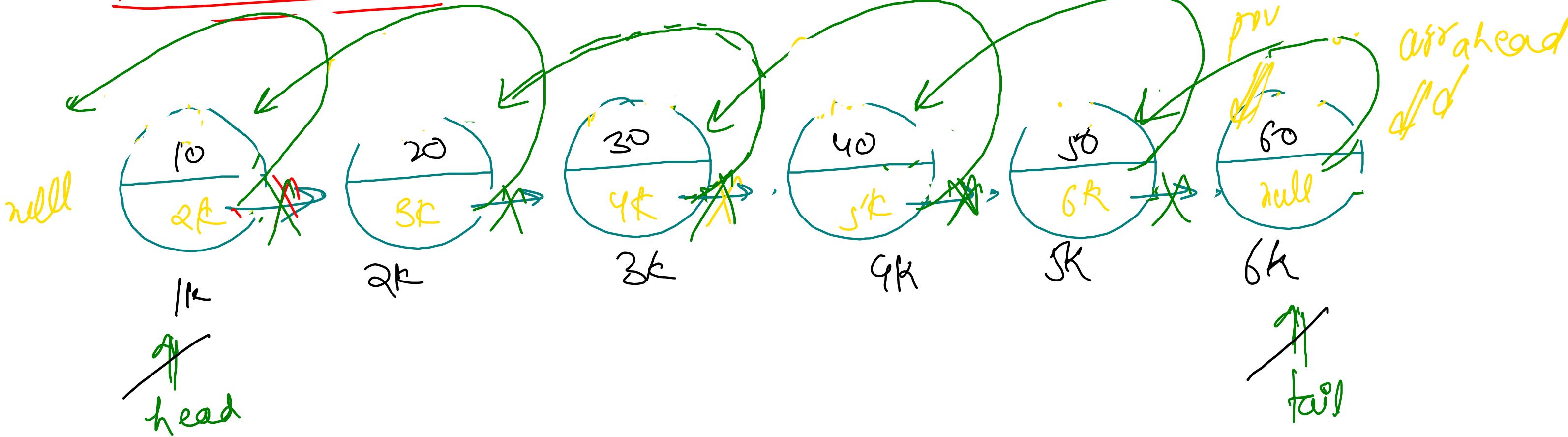


$O(N) \rightarrow (0, 5)$
 $O(N) \rightarrow (1, 4)$
 $O(N) \rightarrow (2, 3)$

3 swaps

$$\frac{N}{2} \times O(2N) \rightarrow O(N^2)$$

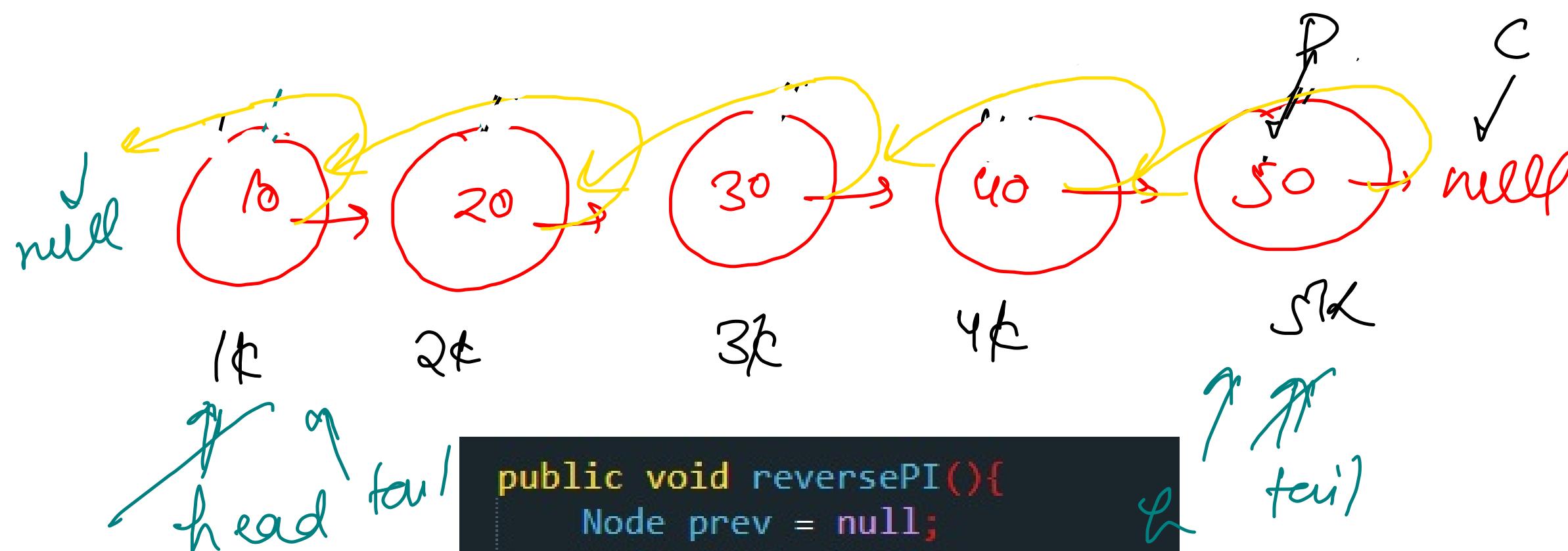
Pointer - Iterative



```
Node prev = null, curr = head;  
while (curr != null) {  
    Node ahead = curr.next;  
    curr.next = prev;  
    prev = curr; curr = ahead;  
}
```

swap (head, tail)

- ① Even
- ② Odd
- ③ 1 Node
- ④ 0 Node



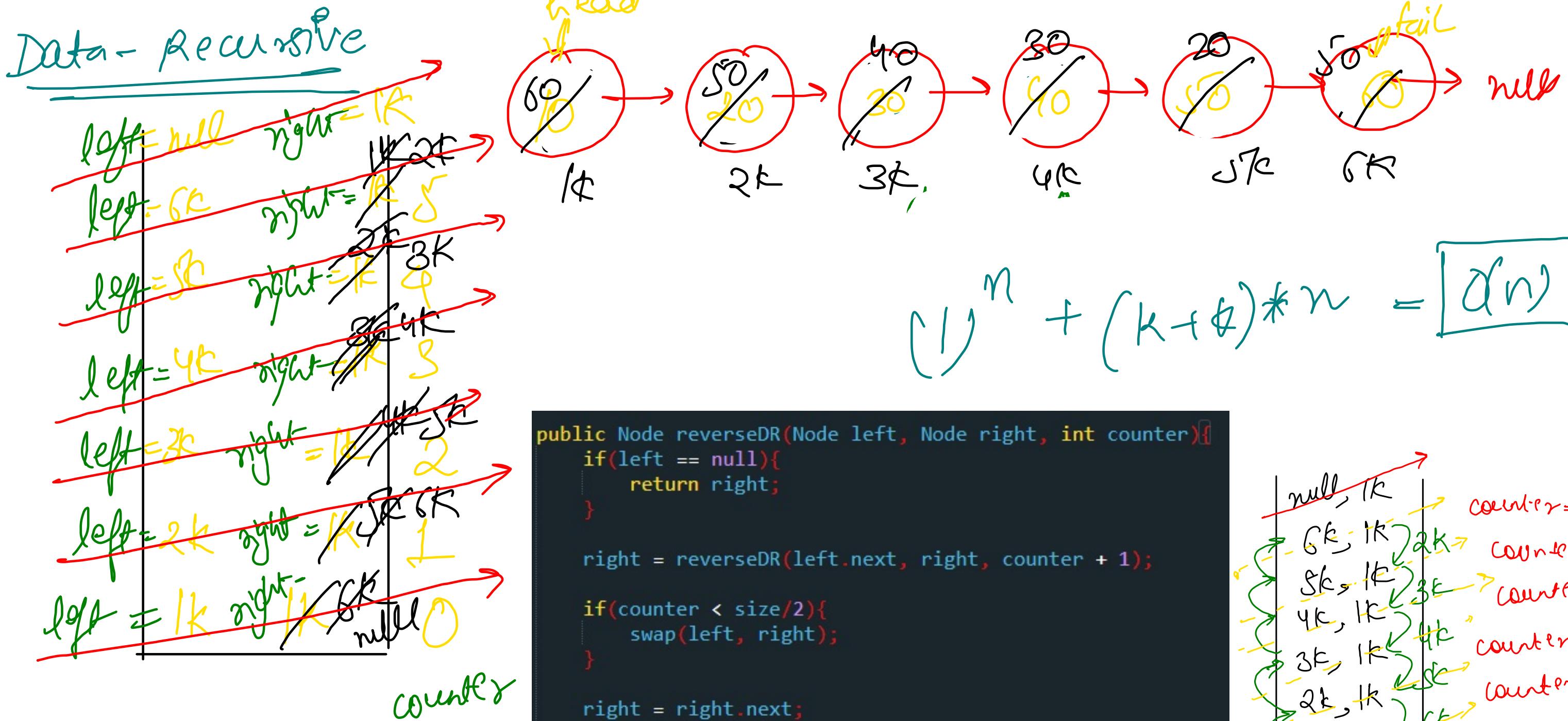
```

public void reversePI(){
    Node prev = null;
    Node curr = head;

    while(curr != null){
        Node ahead = curr.next;
        curr.next = prev;
        prev = curr;
        curr = ahead;
    }

    Node temp = head;
    head = tail;
    tail = temp;
}

```



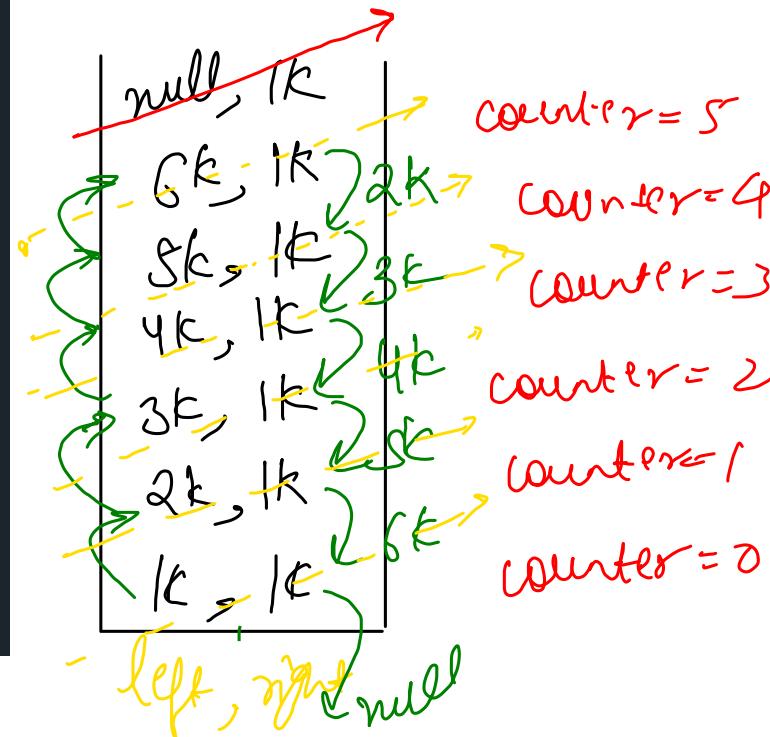
$$n^m + (k+1)*n = \boxed{O(n)}$$

```
public Node reverseDR(Node left, Node right, int counter){
    if(left == null){
        return right;
    }

    right = reverseDR(left.next, right, counter + 1);

    if(counter < size/2){
        swap(left, right);
    }

    right = right.next;
    return right;
}
```



```

static Node right;
public void reverseDR(Node left, int counter){
    if(left == null){
        return;
    }

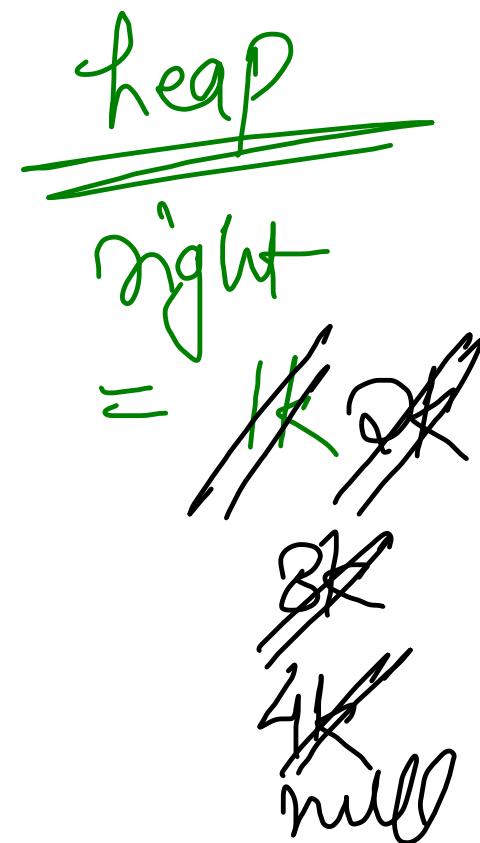
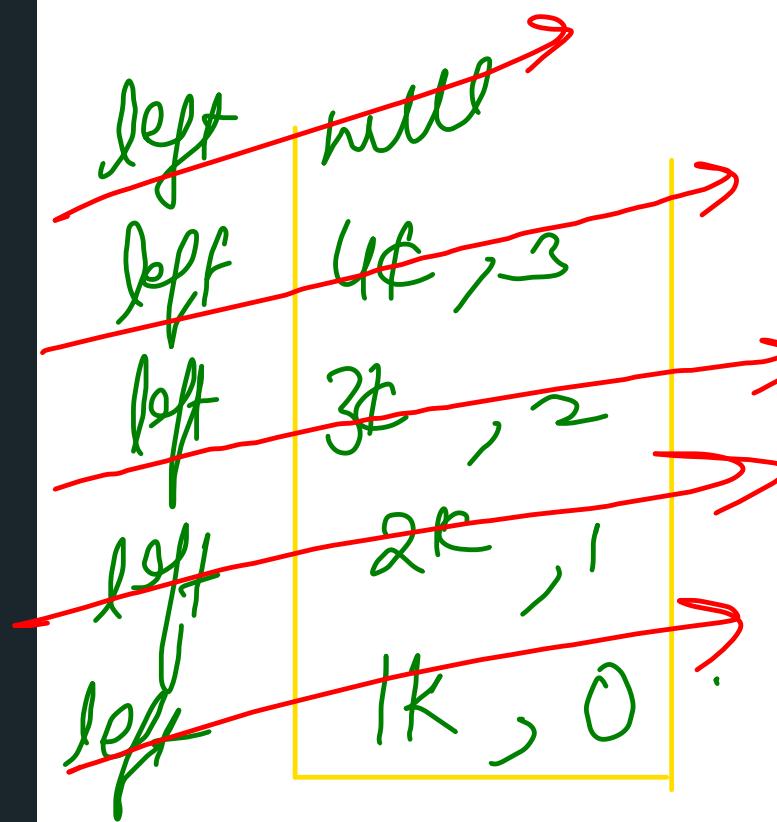
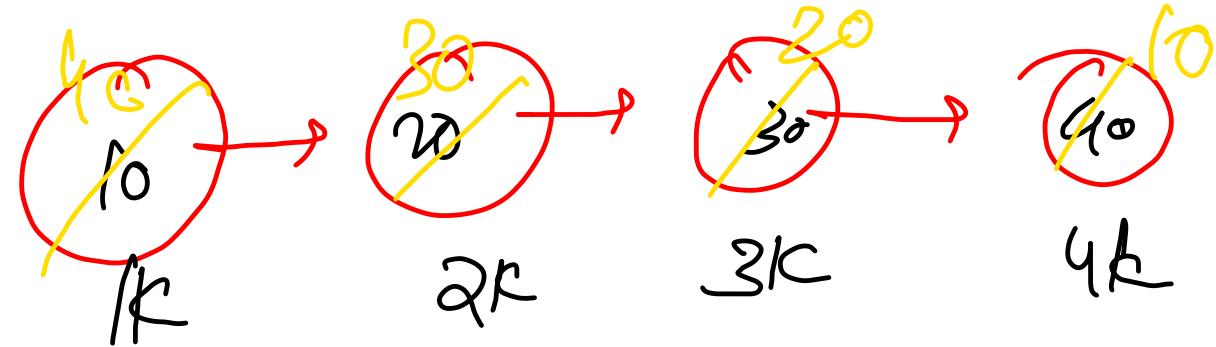
    reverseDR(left.next, counter + 1);

    if(counter < size/2){
        swap(left, right);
    }

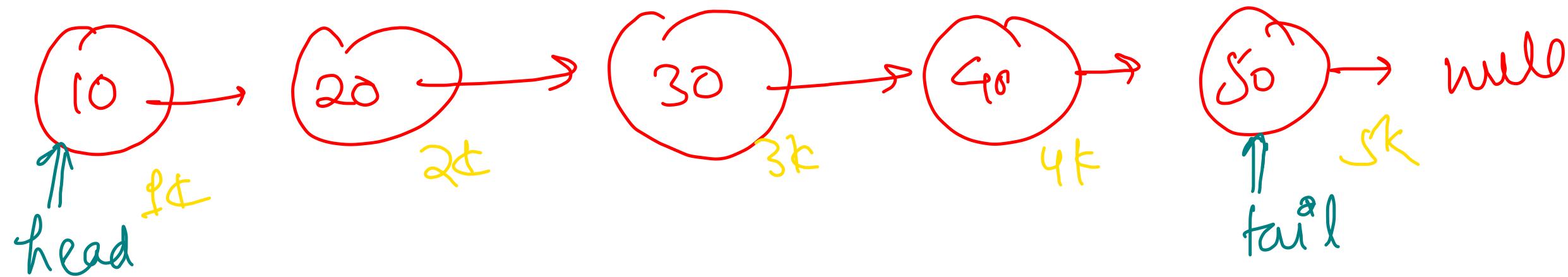
    right = right.next;
}

public void reverseDR() {
    Node left = head;
    right = head;
    reverseDR(left, 0);
}

```



Display Reverse {Recursion}

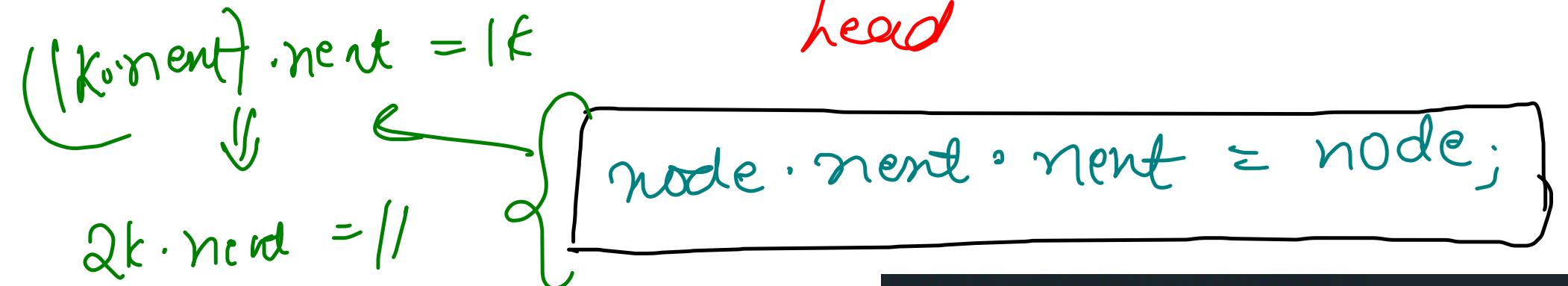
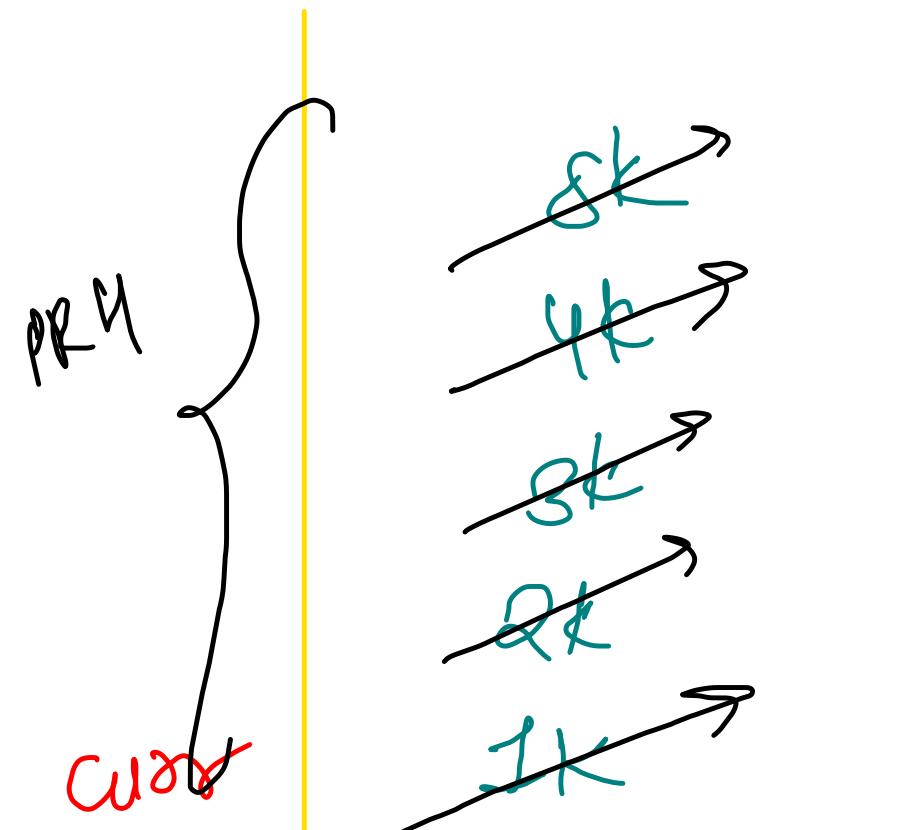


50 , 40 , 30 , 20 , 10

```
private void displayReverseHelper(Node node){  
    if(node == null) return;  
  
    displayReverseHelper(node.next);  
    System.out.print(node.data + " ");  
}
```

$O(N)$

Pointers Recursive → Reverse Lh

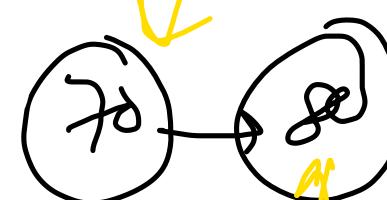
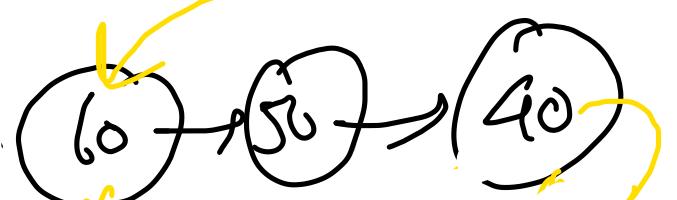
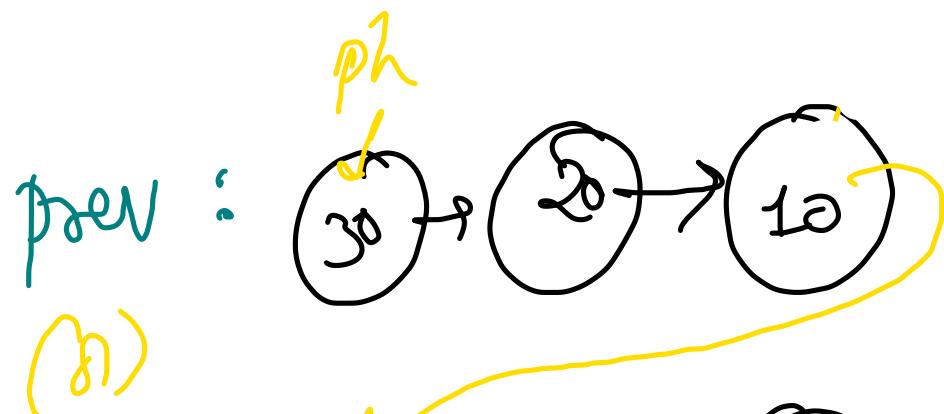
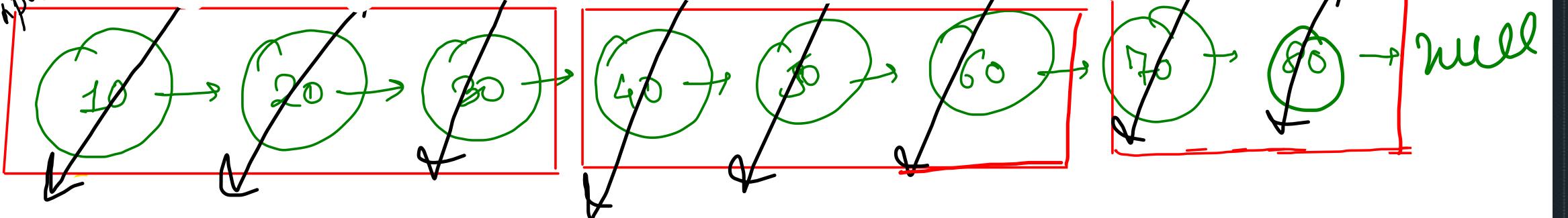


```
private void reversePRHelper(Node node){  
    if(node == null || node.next == null){  
        return;  
    }  
  
    reversePRHelper(node.next);  
  
    // update link of next node  
    node.next.next = node;  
}
```

```
public void reversePR(){  
    Node curr = head;  
    reversePRHelper(curr);  
  
    // swap head and tail  
    Node temp = head;  
    head = tail;  
    tail = temp;  
  
    // make tail's node next as null  
    tail.next = null;  
}
```

this? At reverse LL

input:



pt

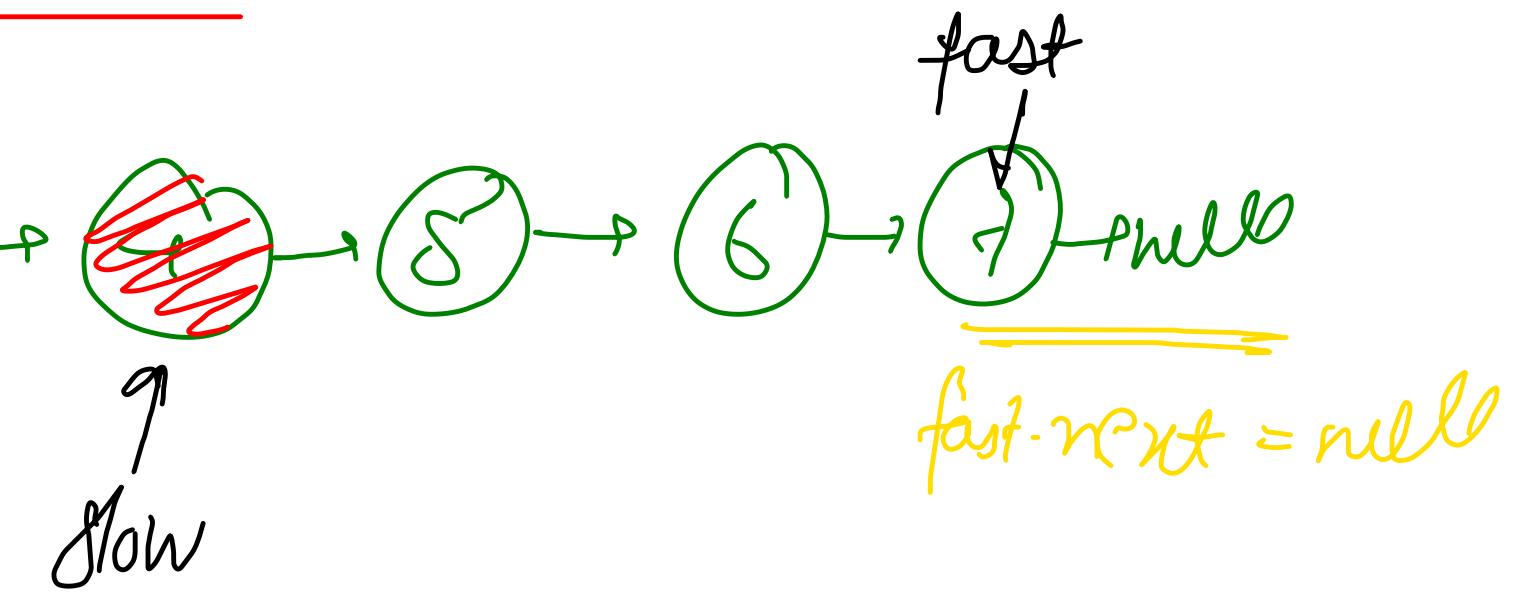
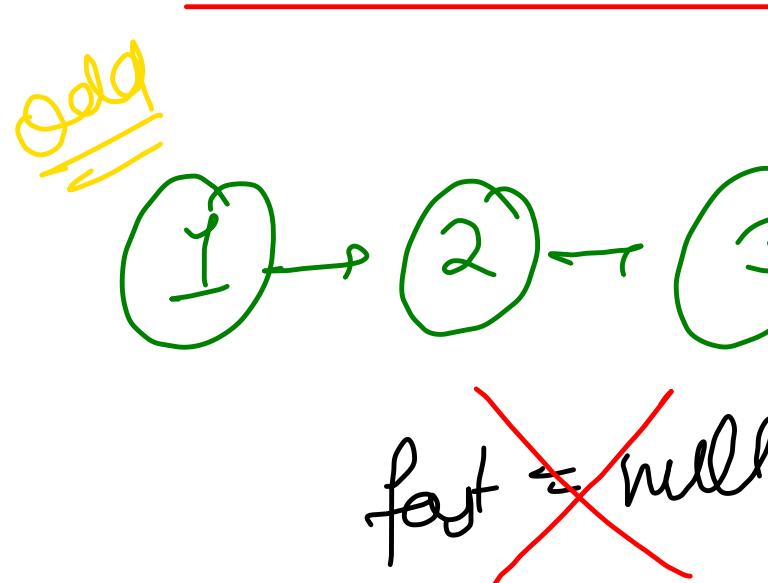
```
if(prev.head == null){  
    // when we have extracted our first group (30, 20, 10)  
    prev = curr;  
} else {  
    // adding one group of size k in the prev linked list  
    prev.tail.next = curr.head;  
    prev.tail = curr.tail;  
    prev.size += curr.size;  
}  
  
prev.tail.next = null;  
this.head = prev.head;  
this.tail = prev.tail;  
this.size = prev.size;
```

```
LinkedList prev = new LinkedList();  
  
while(size > 0){  
    LinkedList curr = new LinkedList();  
  
    if(size < k){  
        // if group is partially filled  
        while(size > 0){  
            int val = head.data;  
            this.removeFirst();  
            curr.addLast(val);  
        }  
    }  
  
    else {  
        // extracting a group of size k  
        for(int i=0; i<k; i++){  
            int val = head.data;  
            this.removeFirst();  
            curr.addFirst(val);  
        }  
    }  
}  
  
if(prev.head == null){  
    // when we have extracted our first group (30, 20, 10)  
    prev = curr;  
} else {  
    // adding one group of size k in the prev linked list  
    prev.tail.next = curr.head;  
    prev.tail = curr.tail;  
    prev.size += curr.size;  
}  
  
prev.tail.next = null;  
this.head = prev.head;  
this.tail = prev.tail;  
this.size = prev.size;
```

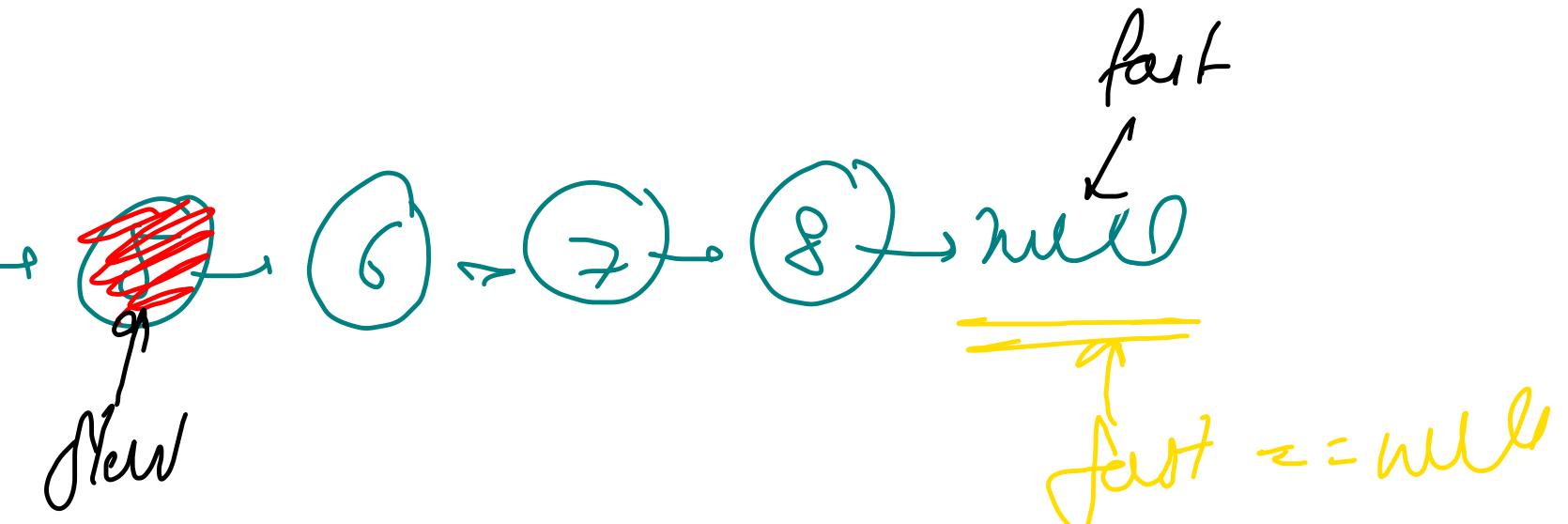
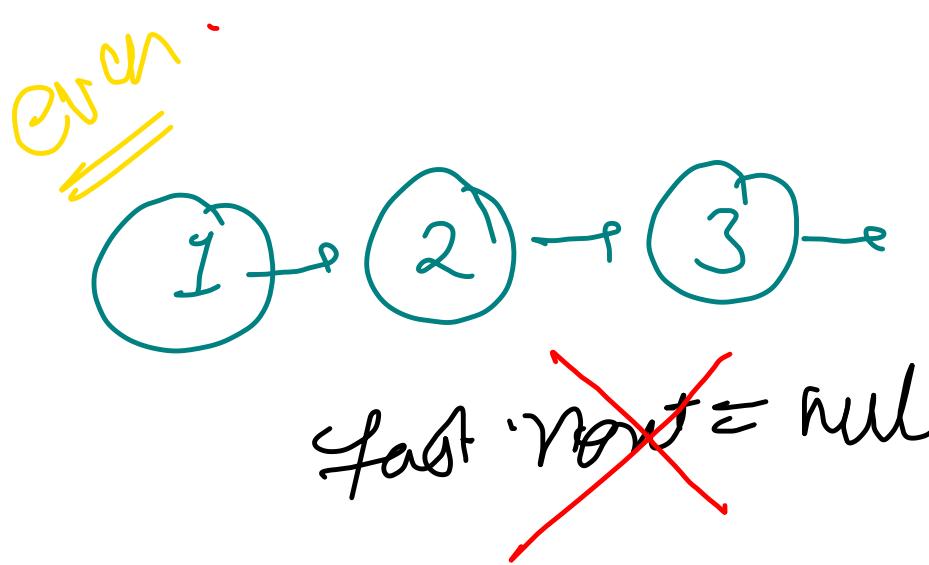


Middle Linked List

Hare & Tortoise } { Two pointers }



① Two traversal
get $\left(\frac{\text{size}}{2}\right) + 1\}$



```
public ListNode reverse(ListNode head){  
    ListNode prev = null, curr = head;  
    while(curr != null){  
        ListNode ahead = curr.next;  
        curr.next = prev;  
        prev = curr;  
        curr = ahead;  
    }  
    return prev;  
}
```

$O(N)$

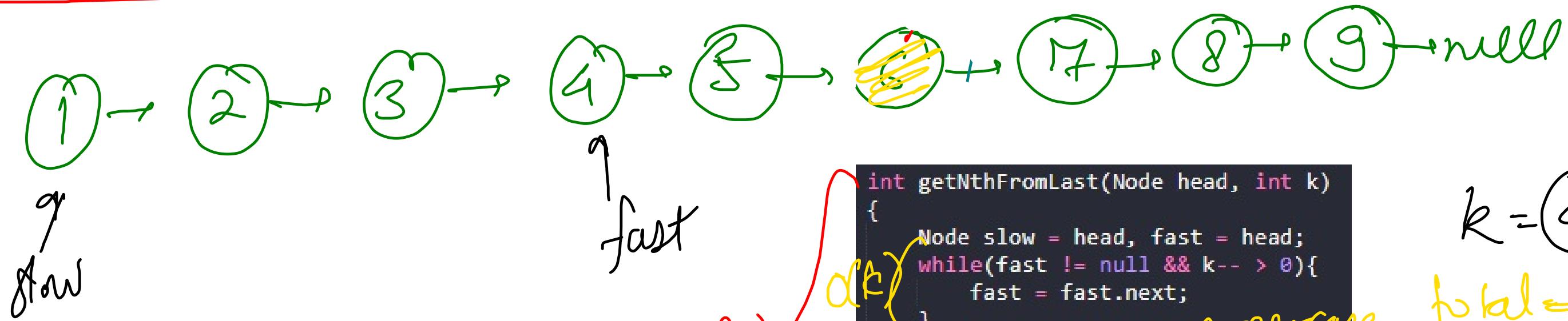
```
public ListNode middle(ListNode head){  
    ListNode slow = head, fast = head;  
    ListNode prev = null;  
  
    while(fast != null && fast.next != null){  
        prev = slow;  
        slow = slow.next;  
        fast = fast.next.next;  
    }  
  
    if(fast == null) // even  
        return prev;  
    return slow;  
}
```

$O(N)$

```
public boolean isPalindrome(ListNode head) {  
    if(head == null || head.next == null){  
        return true;  
    }  
  
    ListNode mid = middle(head);  
    ListNode second = reverse(mid.next);  
  
    while(head != null && second != null){  
        if(head.val != second.val) return false;  
        head = head.next;  
        second = second.next;  
    }  
    return true;  
}
```

$O(N)$

k th Node from End



1st App : 2 Traversal

→ count No of nodes in LL $\Rightarrow N$

→ get(
 $N - k$
 $g - 4$)

```
int getNthFromLast(Node head, int k)
{
    Node slow = head, fast = head;
    while(fast != null && k-- > 0){
        fast = fast.next;
    }
    if(k > 0) return -1; corner case  $k > N$ 
    while(fast != null){
        slow = slow.next;
        fast = fast.next;
    }
    return slow.data;
}
```

$k = 4$

$total = 9 = N$

2nd App : Single traversal
of two pointers

```

int getNthFromLast(Node head, int k)
{
    Node slow = head, fast = head;
    while(fast != null && k-- > 0){
        fast = fast.next;
    }

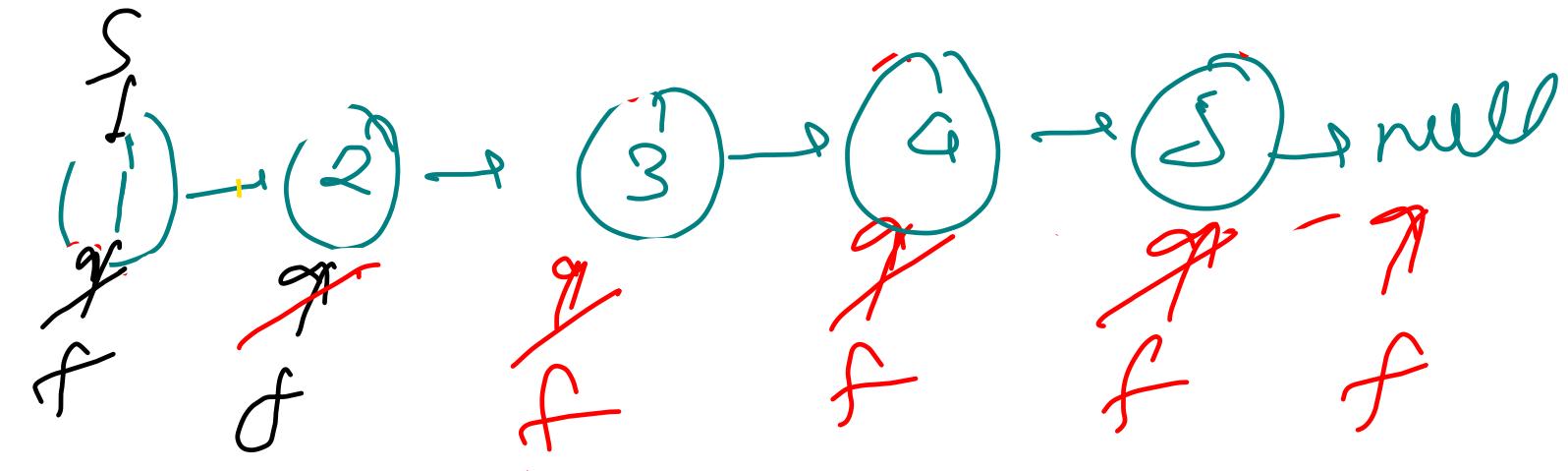
    if(k > 0) return -1;

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

    return slow.data;
}

```

Maintain
dist



~~k = 5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ $\Rightarrow \textcircled{1}$

~~k = 10~~ $\Rightarrow \textcircled{5} \rightarrow k \text{ became } \textcircled{0}$

Both fast became
null & $k = 0$
together

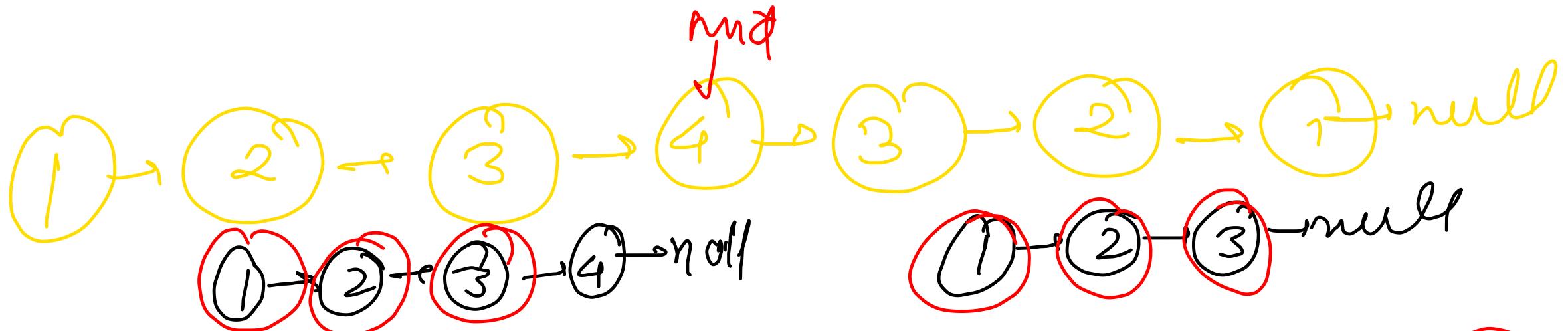
~~k = 6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ $\textcircled{1} \Rightarrow \textcircled{-1}$

\Rightarrow only
fast
became
null

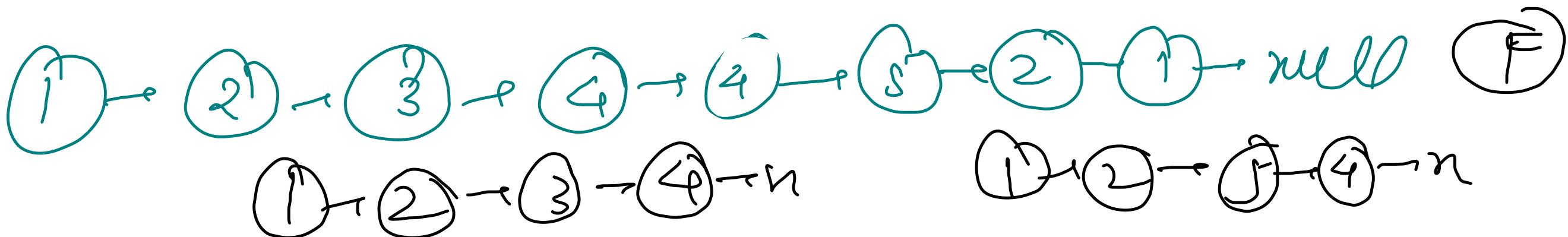
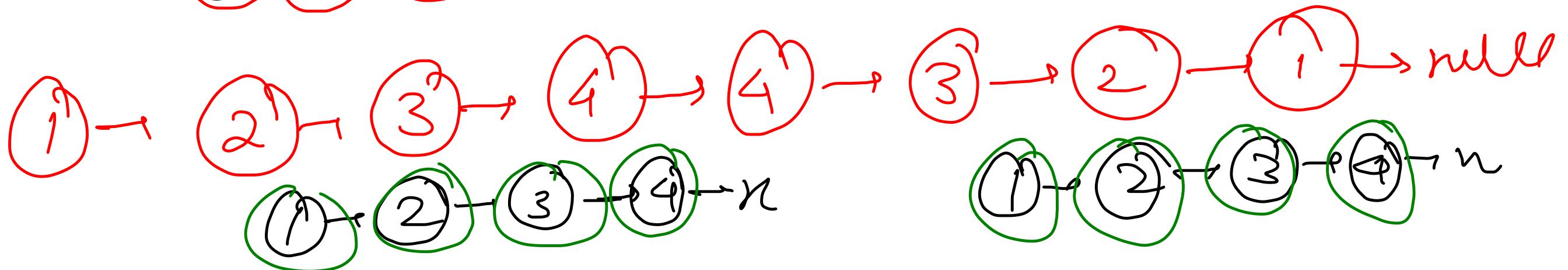
Palindrome LL

HINT → Reverse LL
→ Middle of LL

odd:



Even



Approach

- ① Find Middle node
- ② Reverse the 2nd part
- ③ Compare both the parts
 - ↳ equal \rightarrow Palindrome(T)
 - ↳ not equal \rightarrow Palindrome(F)

```

public ListNode reverse(ListNode head){
    ListNode prev = null, curr = head;
    while(curr != null){
        ListNode ahead = curr.next;
        curr.next = prev;
        prev = curr;
        curr = ahead;
    }
    return prev;
}

```

Pointer Iterative

$O(N)$

```

public ListNode middle(ListNode head){
    ListNode slow = head, fast = head;
    ListNode prev = null;

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

    if(fast == null) // even
        return prev;
    return slow;
}

```

$O(N)$

even \Rightarrow first mid
odd \Rightarrow single mid

```

public boolean isPalindrome(ListNode head) {
    if(head == null || head.next == null){
        return true;
    }

    ListNode mid = middle(head);
    ListNode second = reverse(mid.next);

    while(head != null && second != null){
        if(head.val != second.val) return false;
        head = head.next;
        second = second.next;
    }
    return true;
}


```

corner case

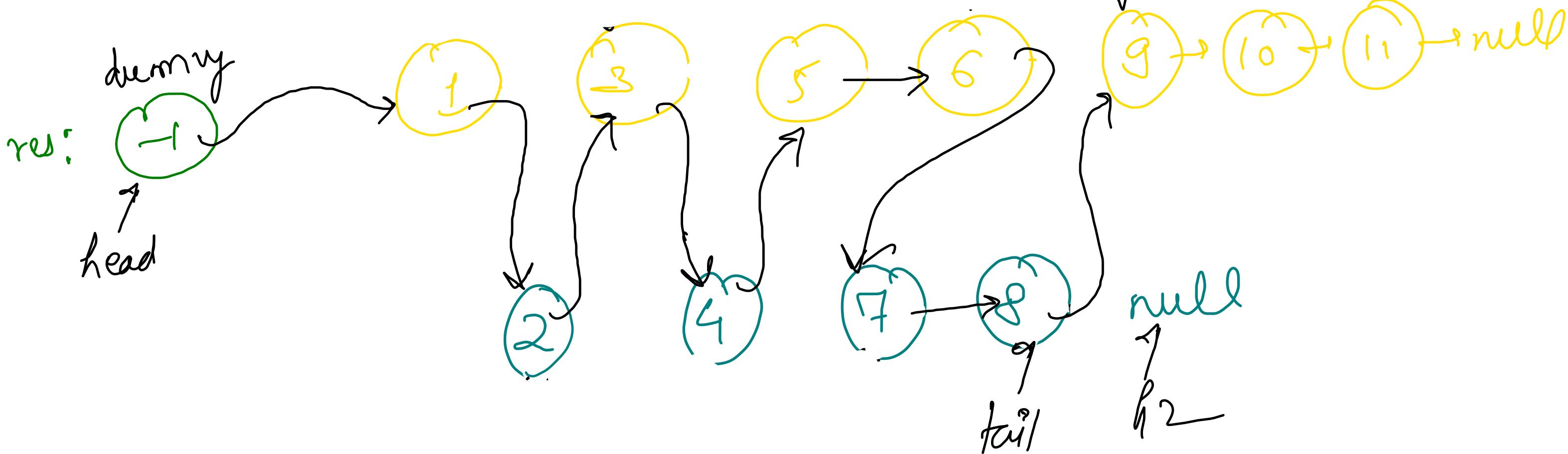
$O(N)$

Merge 2 sorted Lh

Inplace

$O(N+M)$

tail.next = $\min(h_1, h_2)$
tail = tail.next



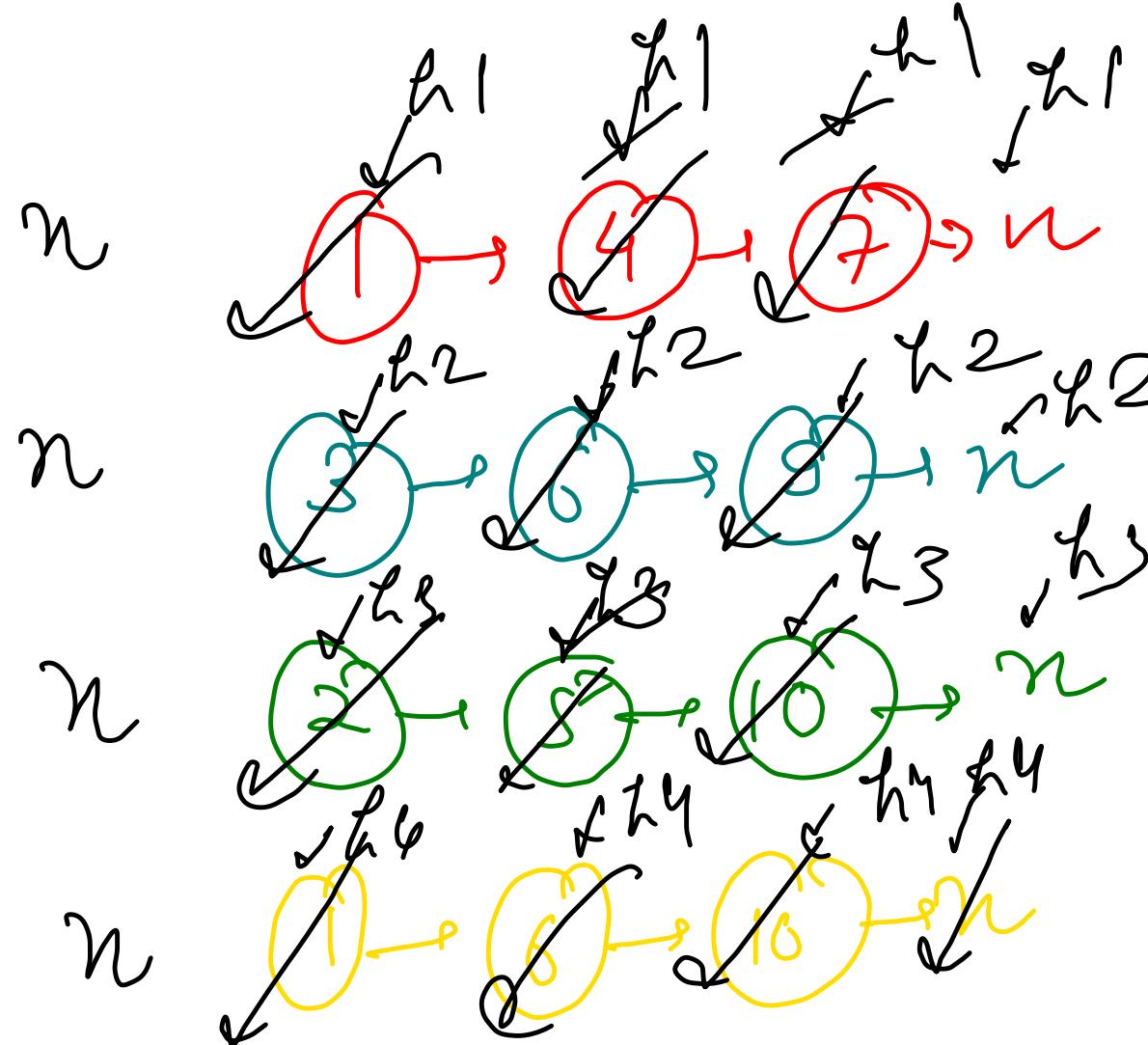
```
Node sortedMerge(Node head1, Node head2) {  
    Node dummy = new Node(-1);  
    Node head = dummy, tail = dummy;  
  
    while(head1 != null && head2 != null){  
        if(head1.data < head2.data){  
            tail.next = head1;  
            head1 = head1.next;  
        }  
        else{  
            tail.next = head2;  
            head2 = head2.next;  
        }  
        tail = tail.next;  
    }  
  
    if(head1 != null) tail.next = head1;  
    else tail.next = head2;  
  
    return dummy.next;  
}
```

Lecture 4

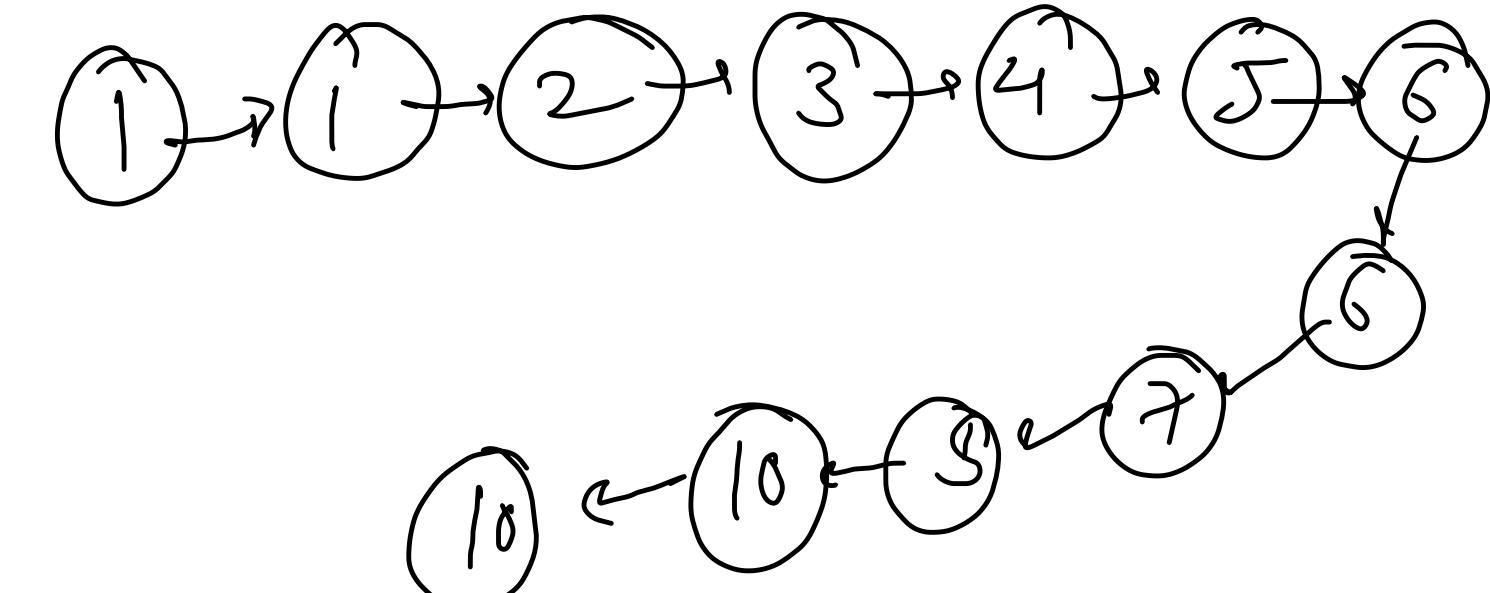
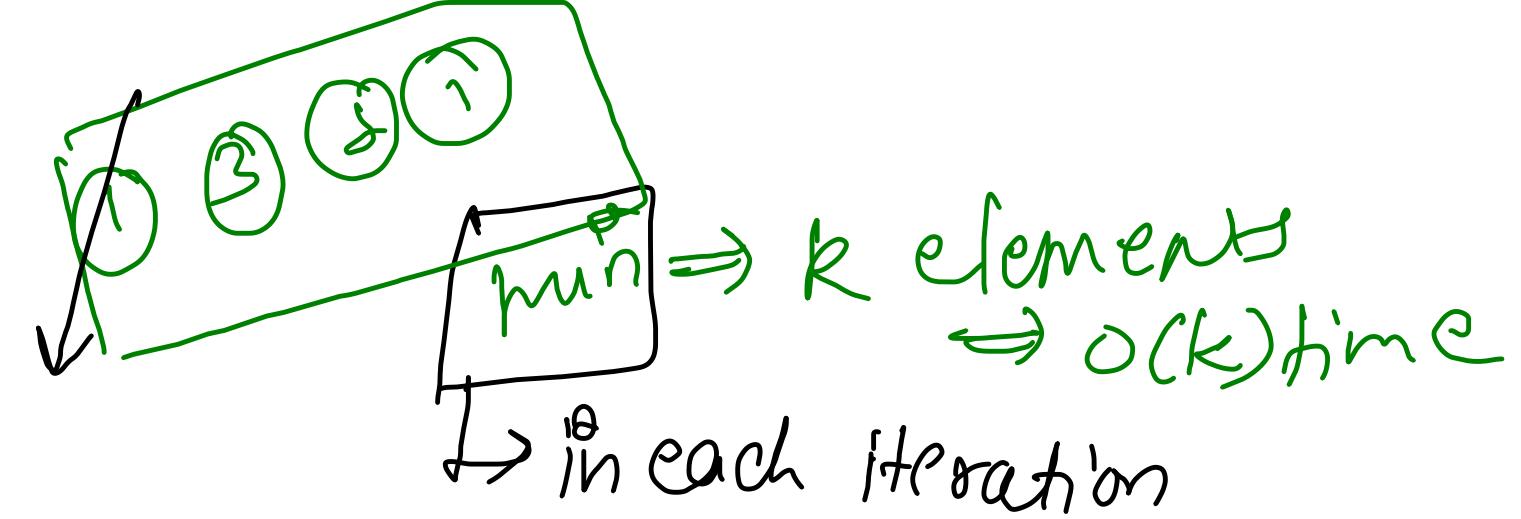
{Thursday}

- merge th
 - merge 2 sorted
 - merge K sorted
 - merge sort
 - Remove Duplicates
(18/1)
 - Fold & unfold th
Pepcoding
- LC 21
- LC HARD 23**
- LC 148
- LC 83 & 82

Merge K sorted LL {HARD}



resultant
node
 $N^{(12)}$
 $N = n * k \approx$ number of ll
node in 1 LL



Brute force
 $O(n * k^2)$
or $O(N * k)$

~~Brute force~~ - $O(n^k)$ = $O(N^k)$

```
public ListNode mergeKLists(ListNode[] lists) {
    if(lists.length == 0) {
        // 0 linked lists
        return null;
    }

    ListNode dummy = new ListNode(-1);
    ListNode head = dummy, tail = dummy;

    while(true){
        int minIdx = minNode(lists);
        if(minIdx == -1) break;

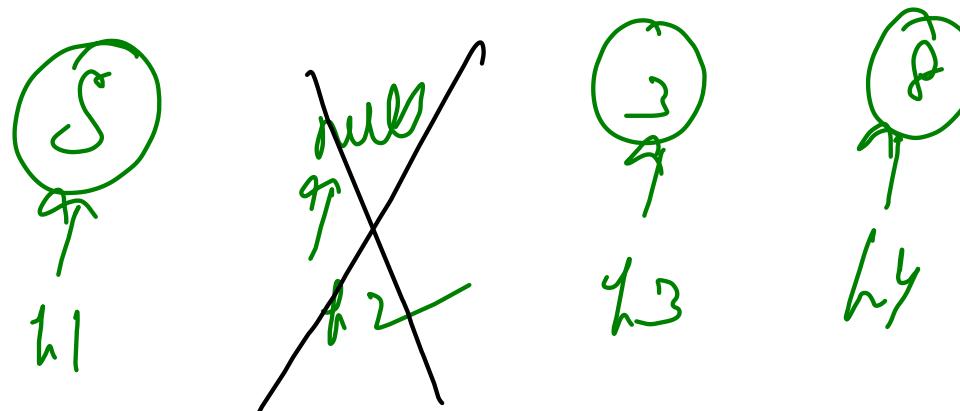
        tail.next = lists[minIdx];
        lists[minIdx] = lists[minIdx].next;
        tail = tail.next;
    }

    return dummy.next;
}
```

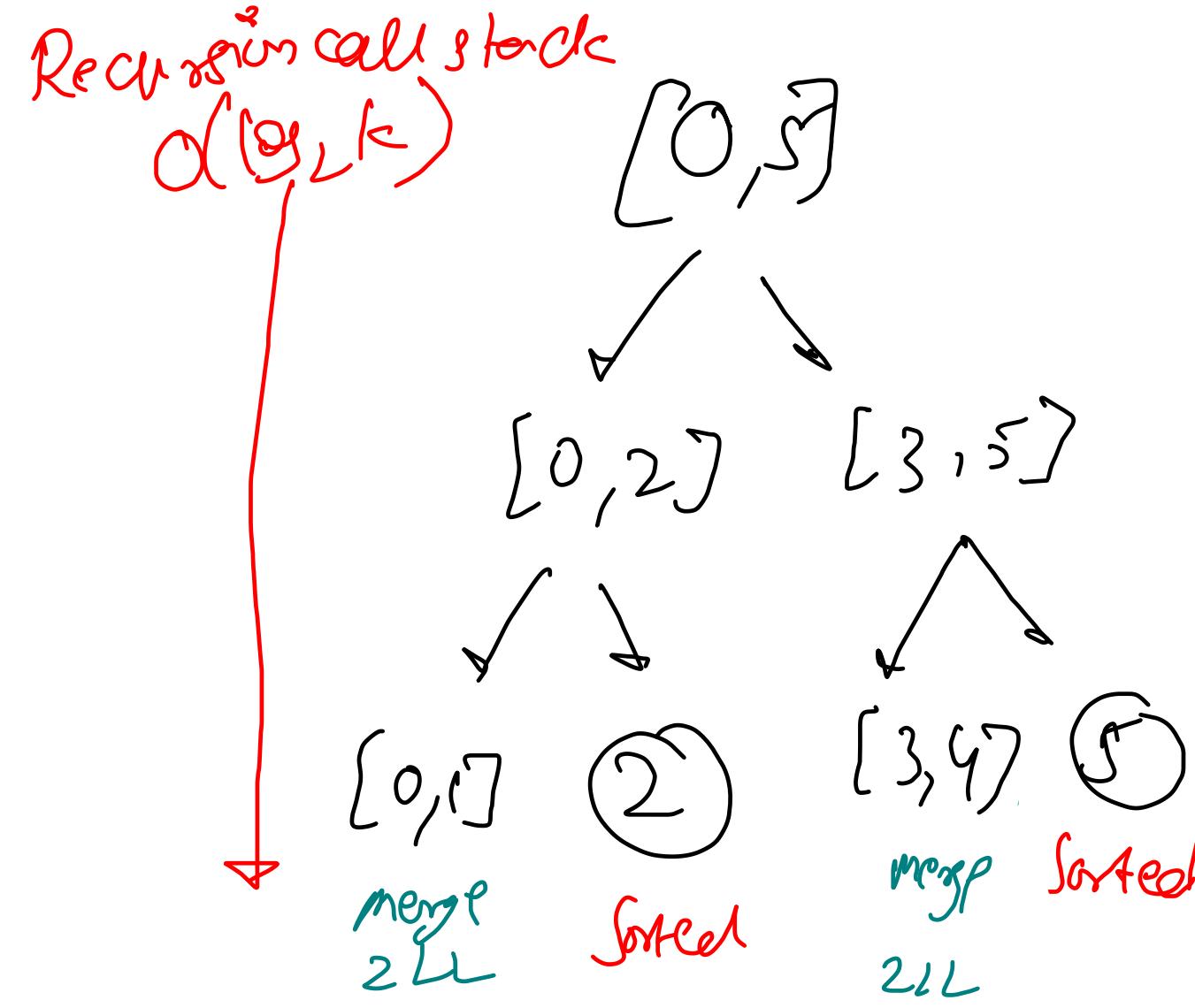
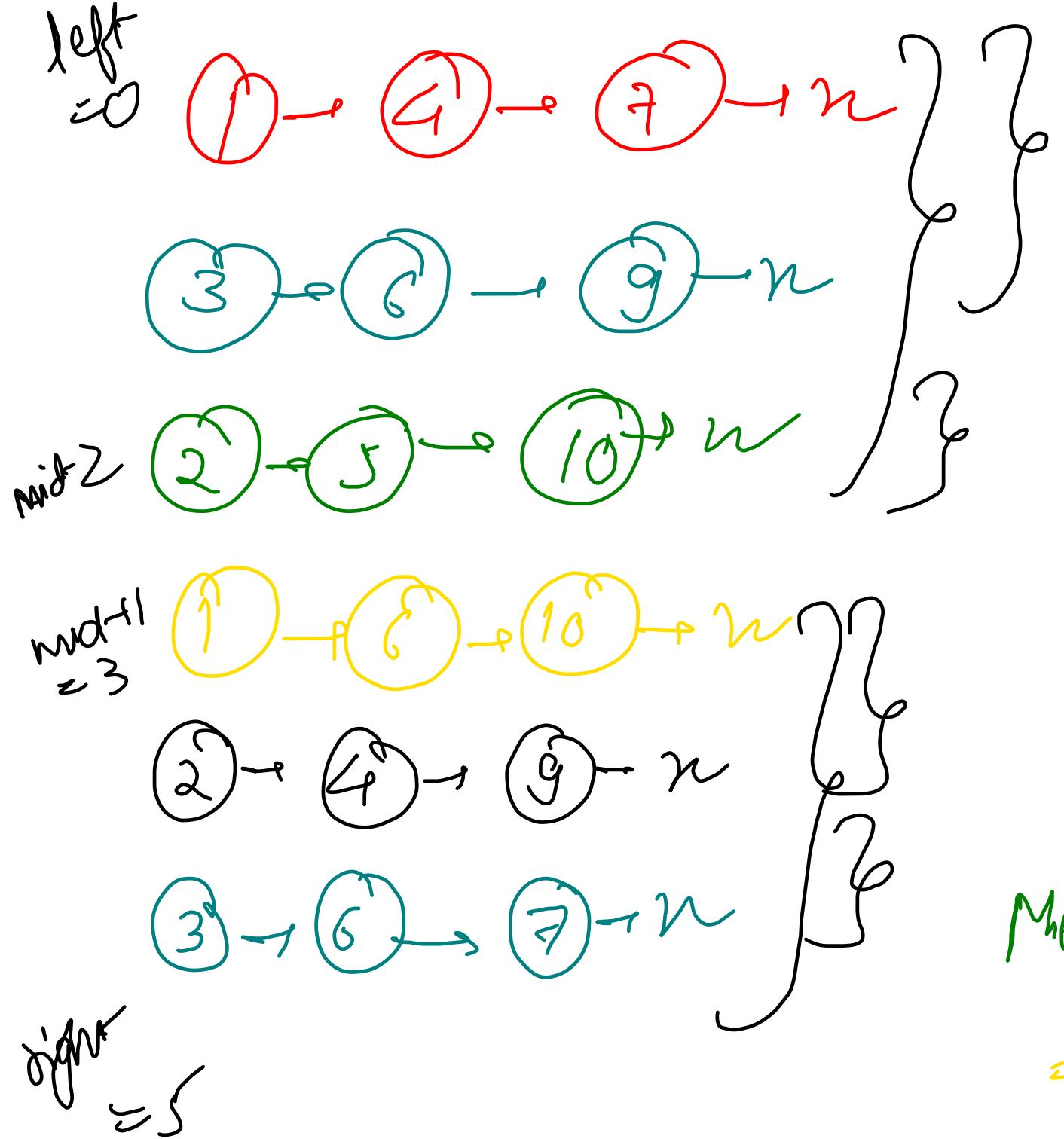
```
public int minNode(ListNode[] lists){
    int min = Integer.MAX_VALUE;
    int idx = -1;

    for(int i=0; i<lists.length; i++){
        if(lists[i] != null && lists[i].val < min){
            idx = i;
            min = lists[i].val;
        }
    }

    return idx;
}
```



$\minIndex = 1$, $\min = 3$



Merge K Sorted (l, r)

$$\begin{aligned}
 &= \text{Merge K Sorted } (l, m) + \text{Merge K Sorted } (m+1, r) \\
 &\quad + \text{Merge 2 Lh}
 \end{aligned}$$

```

ListNode merge2List(ListNode head1, ListNode head2) { }

public ListNode helper(ListNode[] lists, int left, int right){
    if(left > right) return null;
    if(left == right) return lists[left];

    int mid = (left + right) / 2;
    ListNode l1 = helper(lists, left, mid);
    ListNode l2 = helper(lists, mid + 1, right);
    return merge2List(l1, l2);
}

public ListNode mergeKLists(ListNode[] lists) {
    if(lists.length == 0) {
        // 0 linked lists
        return null;
    }

    return helper(lists, 0, lists.length - 1);
}

```

} } Divide & Conquer

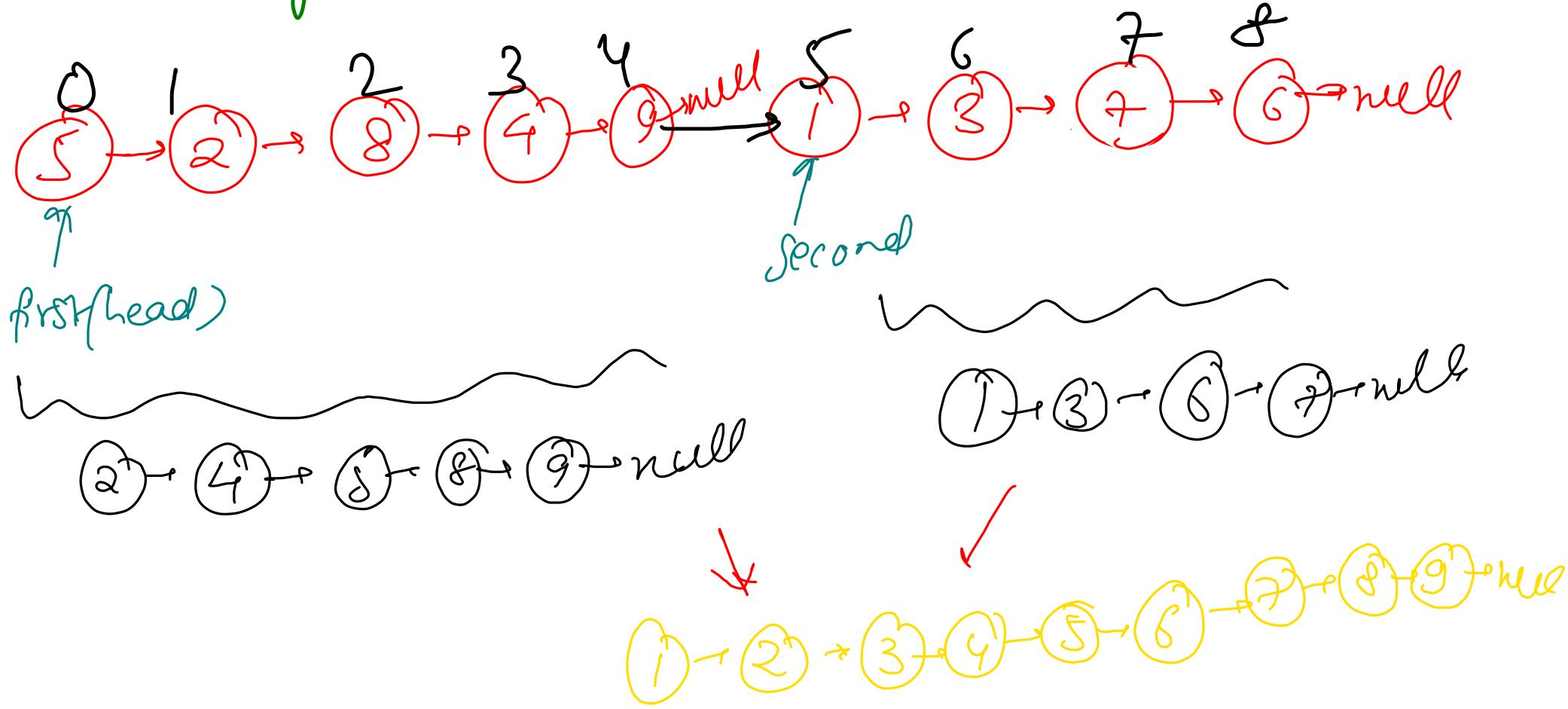
$$T(k) = k \cancel{T(0)} + k \cdot n \cancel{\log k}$$

no of terms

$$\begin{aligned}
T(k) &= 2T(k/2) + \frac{k \cdot n}{2^0} \\
T(k/2) &= 2T(k/4) + \frac{k \cdot n}{2^1} \times 2 \\
T(k/4) &= 2T(k/8) + \frac{k \cdot n}{2^2} \times 2^2 \\
&\vdots \\
T(1) &= 2T(0) + \frac{k \cdot n}{2^{\log_2 k}} \times 2^{\log_2 k}
\end{aligned}$$

$$\mathcal{O}(k \cdot n \cdot \log_2 k)$$

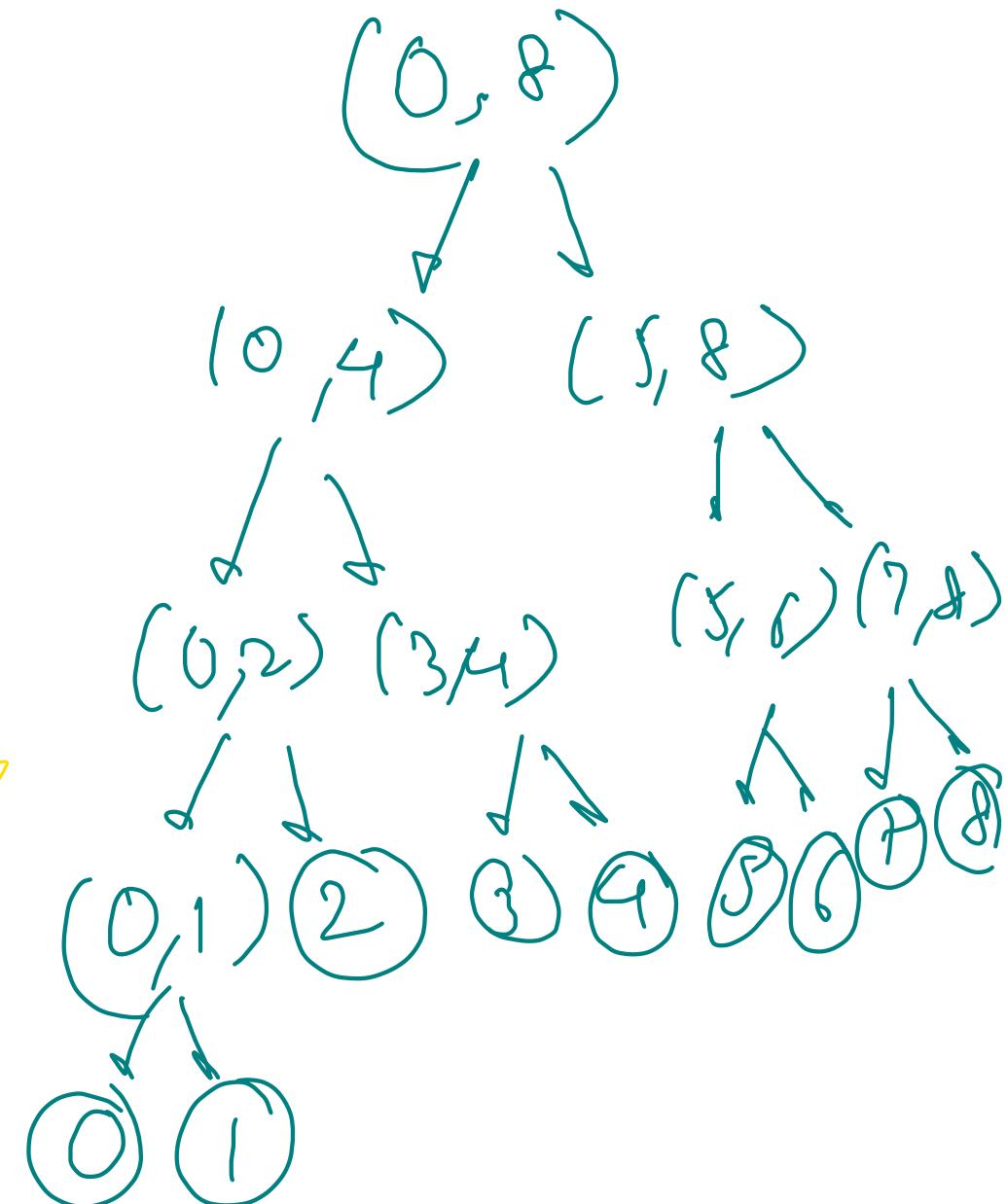
Merge Sort LL {Tc & LC}



Approach

MergeSort(LL) →

- ① Find middle & break LL into 2 parts
- ② mergeSort(left)
mergeSort(right)
- ③ Merge & return



```
public ListNode middle(ListNode head){  
    ListNode slow = head, fast = head;  
    ListNode prev = null;  
  
    while(fast != null && fast.next != null){  
        prev = slow;  
        slow = slow.next;  
        fast = fast.next.next;  
    }  
  
    if(fast == null) // even  
        return prev;  
    return slow; // odd  
}
```

```
ListNode merge2list(ListNode head1, ListNode head2) {  
  
    public ListNode sortList(ListNode head) {  
        if(head == null || head.next == null){  
            return head;  
        }  
  
        ListNode mid = middle(head);  
        ListNode midNext = mid.next;  
        mid.next = null;  
  
        ListNode left = sortList(head);  
        ListNode right = sortList(midNext);  
        return merge2list(left, right);  
    }  
}
```

$$T(n) = \Theta(n) + 2T(n/2) + O(n)$$

Why merge sort is preferred for linked list

and quicksort is preferred for arrays.

TC: $\Theta(n \lg n)$ SC: $O(1)$

merge sort on LL

$T(N) \rightarrow \Theta(n \lg n)$

Space \rightarrow in place \star

Recursive call
slack $\Rightarrow O(\lg n)$

merge 2 list
 \downarrow
is in place

→ Big-Integers

→ Add 2 Lh

→ Subtract 2 Lh

→ Multiply 2 Lh

Int

$$2^{31} - 1$$

$$\approx 10^9$$

Long

$$2^{63} - 1$$

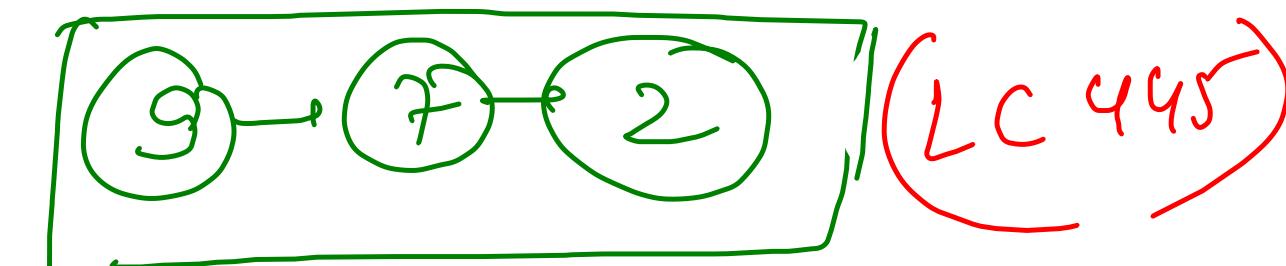
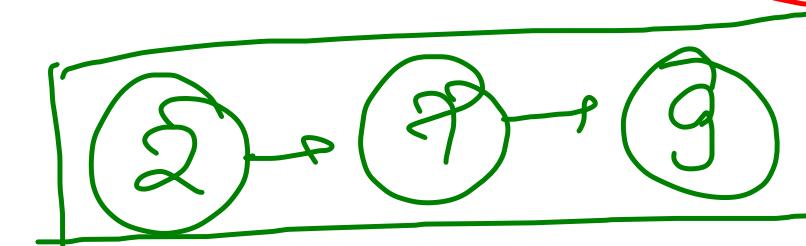
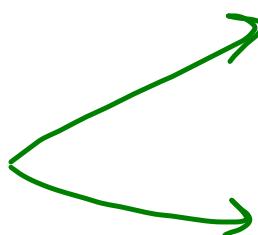
$$\approx 10^{19}$$

Addition of 2 LL

LC 2

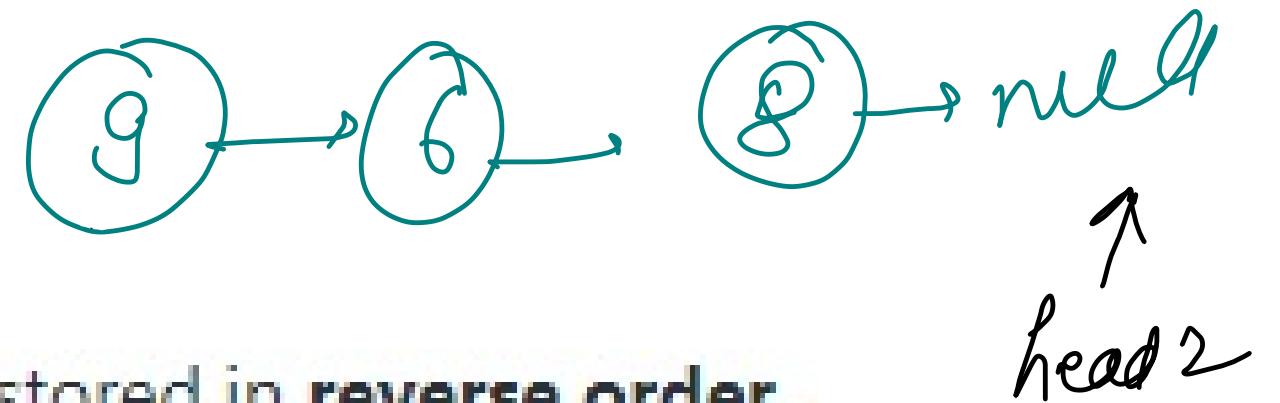
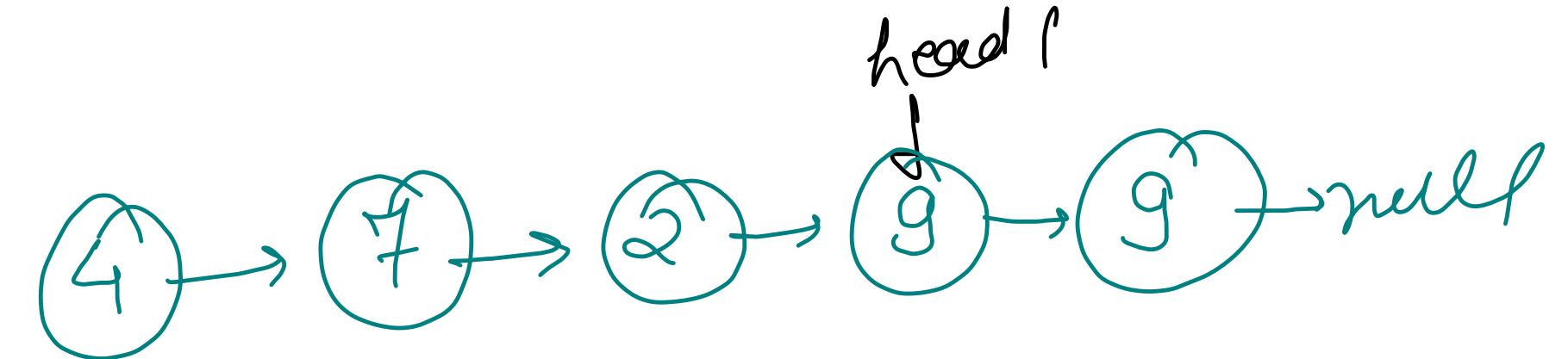
LC 445

972



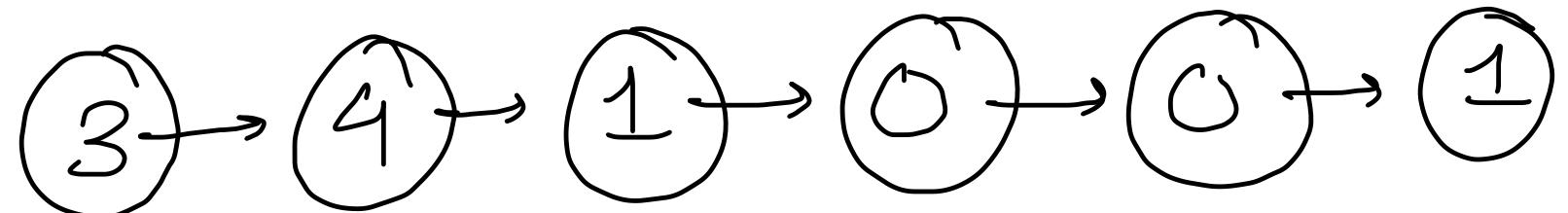
~~LCQ~~

$c = \emptyset / 0$



The digits are stored in **reverse order**.

so:



$$\begin{aligned} & (d_1 + d_2 + c) / 10^0 \\ & (d_1 + d_2 + c) / 10^1 \text{ carry} \end{aligned}$$

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  
    ListNode dummy = new ListNode(-1);  
    ListNode head = dummy, tail = dummy;  
  
    int carry = 0;  
    while(l1 != null || l2 != null || carry > 0){  
        int d1 = (l1 == null) ? 0 : l1.val;  
        int d2 = (l2 == null) ? 0 : l2.val;  
  
        ListNode temp = new ListNode((d1 + d2 + carry) % 10);  
        carry = (d1 + d2 + carry) / 10;  
  
        tail.next = temp;  
        tail = temp;  
        if(l1 != null) l1 = l1.next;  
        if(l2 != null) l2 = l2.next;  
    }  
  
    return dummy.next;  
}
```



Getting Started

LC 445

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    l1 = reverse(l1);
    l2 = reverse(l2);

    ListNode dummy = new ListNode(-1);
    ListNode head = dummy, tail = dummy;

    int carry = 0;
    while(l1 != null || l2 != null || carry > 0){
        int d1 = (l1 == null) ? 0 : l1.val;
        int d2 = (l2 == null) ? 0 : l2.val;

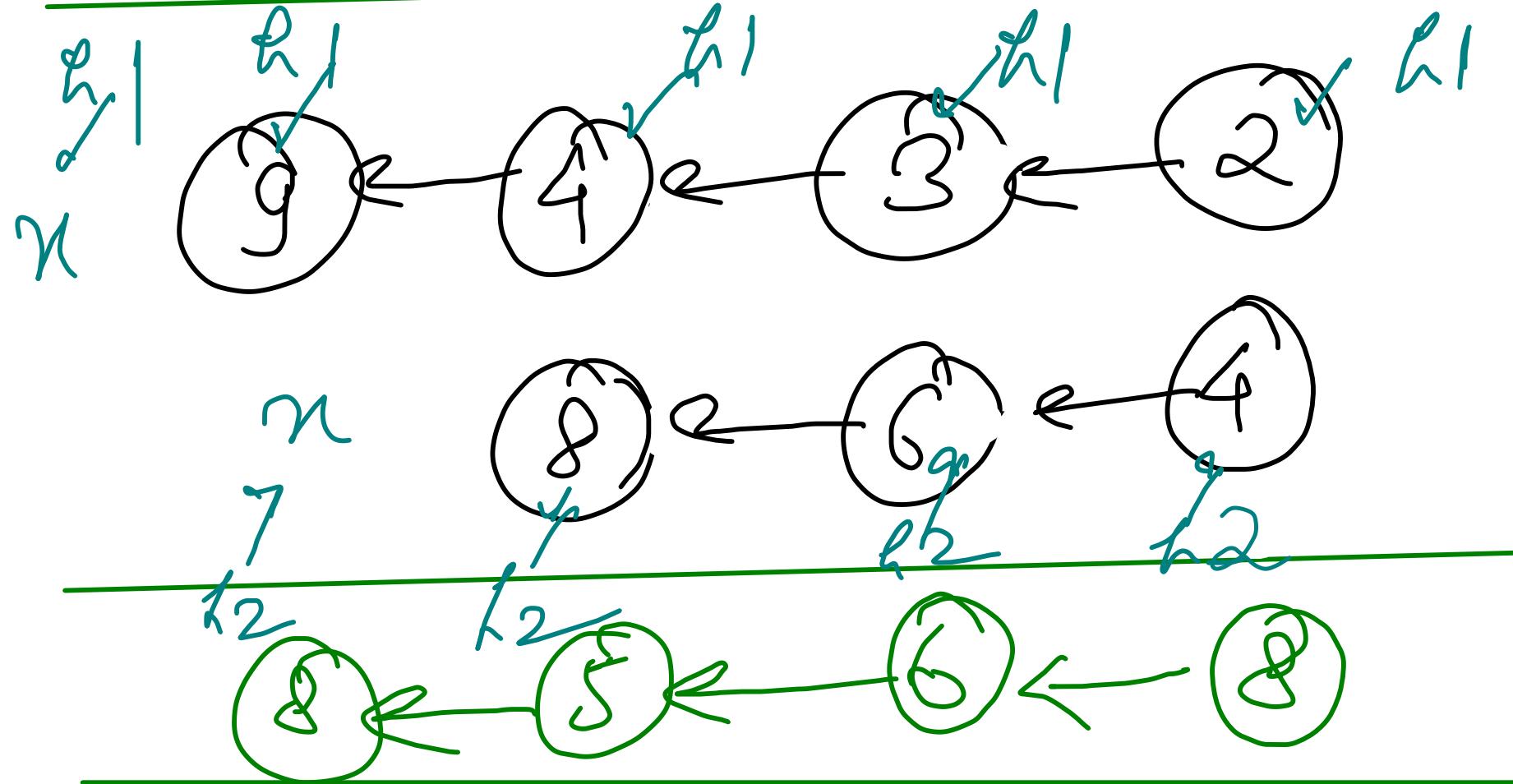
        ListNode temp = new ListNode((d1 + d2 + carry) % 10);
        carry = (d1 + d2 + carry) / 10;

        tail.next = temp;
        tail = temp;
        if(l1 != null) l1 = l1.next;
        if(l2 != null) l2 = l2.next;
    }

    return reverse(dummy.next);
}
```

```
public ListNode reverse(ListNode head){
    ListNode prev = null, curr = head;
    while(curr != null){
        ListNode ahead = curr.next;
        curr.next = prev;
        prev = curr;
        curr = ahead;
    }
    return prev;
}
```

Subtraction in linked list



linked list

lecture 5

→ Floyd's cycle

→ Cycle Detection

→ Starting pt of cycle

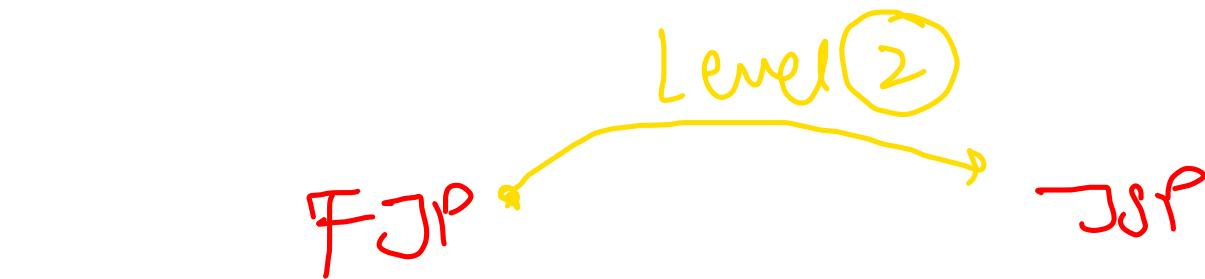
→ Mathematical Proof

→ Insertion Node

→ Clone th

→ With extra space

→ w/o extra space



M, T, W, F

DSA level ①

Thurs, weekend ^{Dev}

DSA

LL → 1/2

Wed → Resume Review
SQA → Level 1

Thurs
Sat (2 classes)
Sun (2 classes)

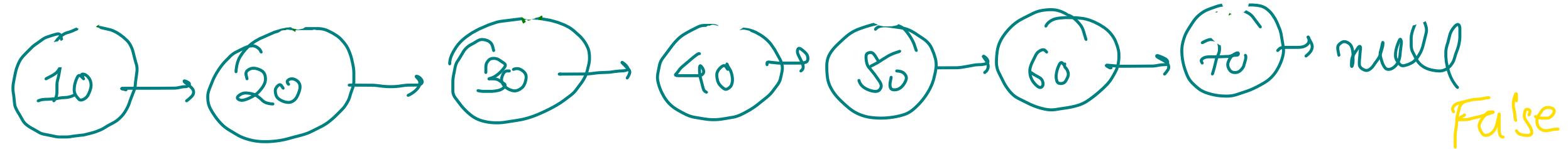
SDX

SFS [3-6, 9-12]

Floyd's cycle Detection

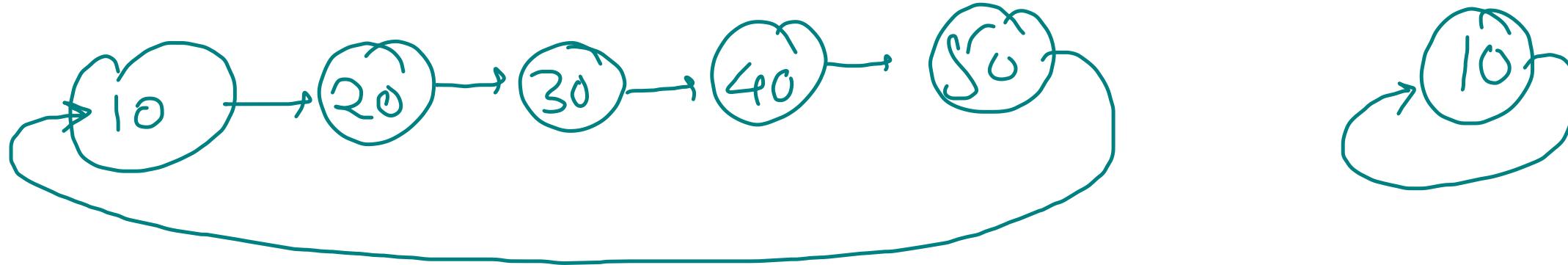
YC 191

case 1)
SLL



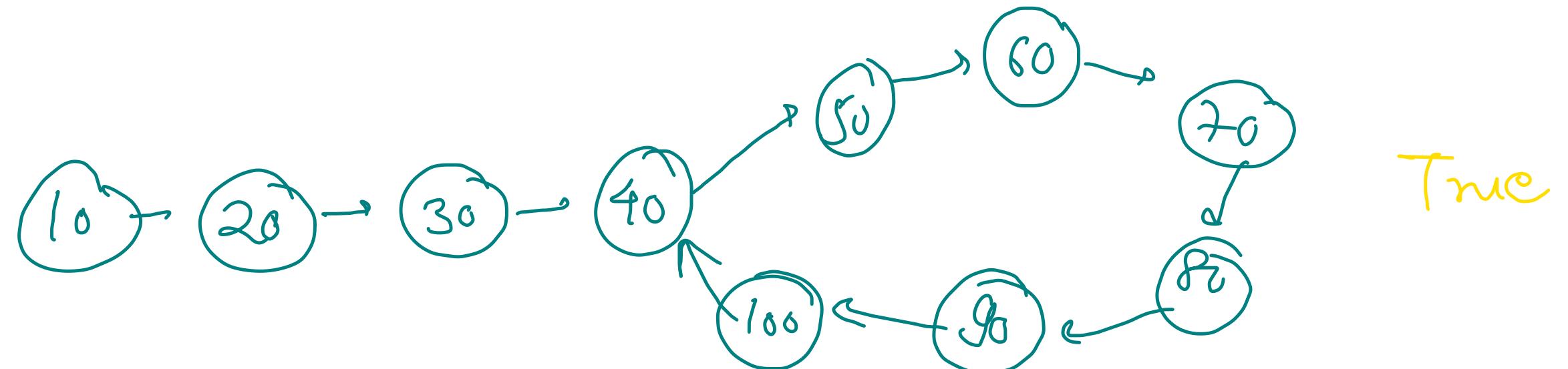
False

case 2)
CSLL



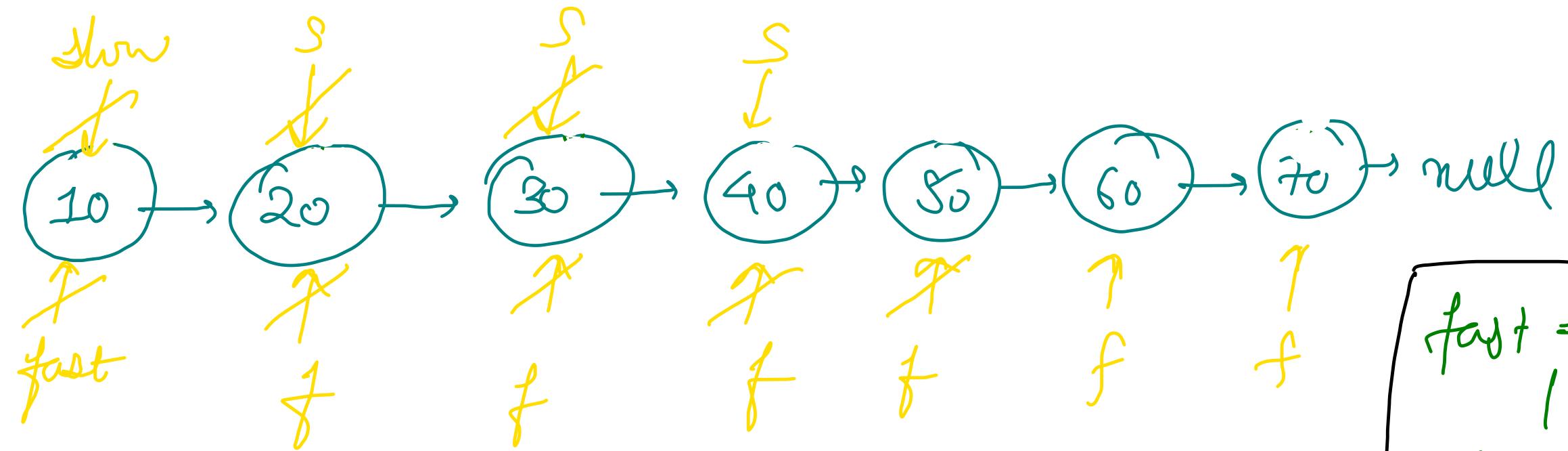
True

Case 3)
SLL Tail
+
Loop



True

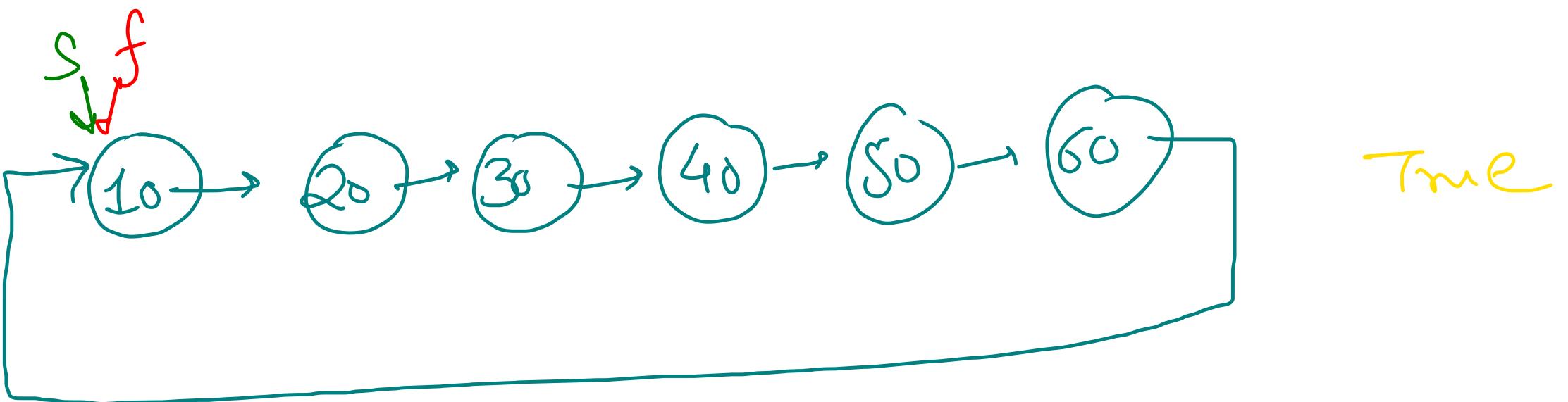
case 1)
SLL



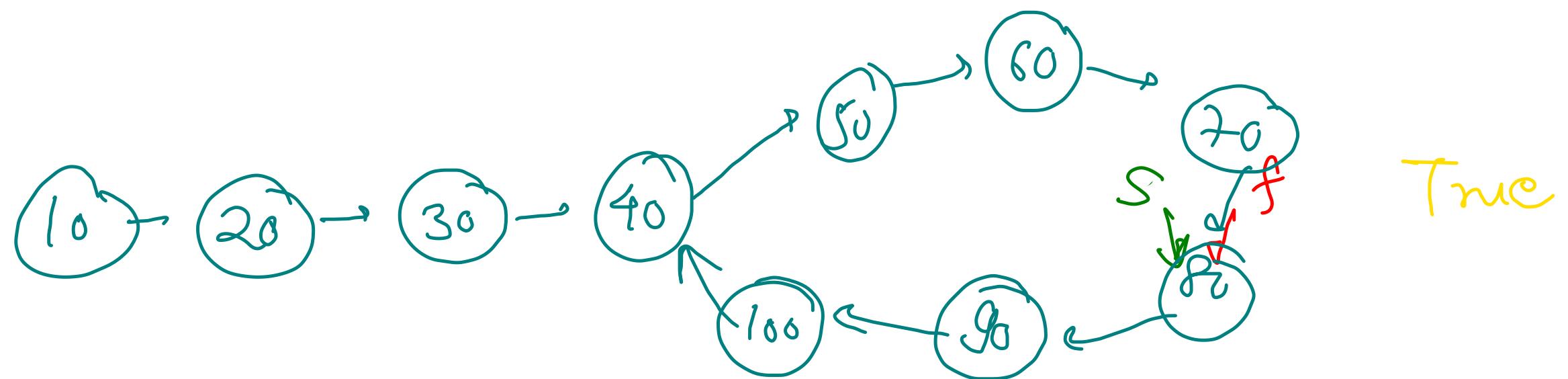
fast == null
||
fast - next
== null

false
(no cycle)

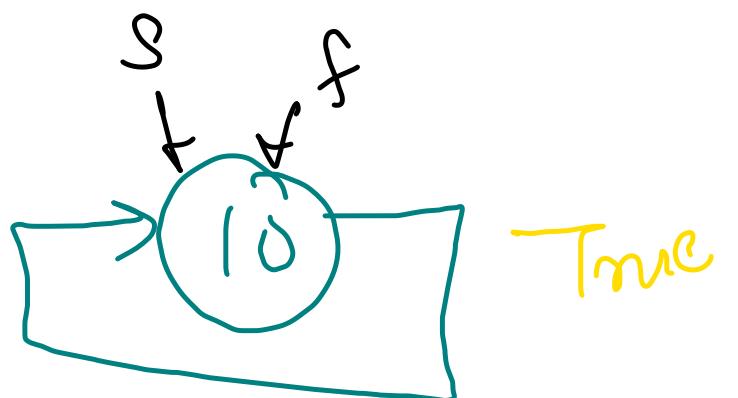
case 2)
CSLL



case 3)
Tail + loop



Corner case



Corner case
0 nodes

head = tail = null
false

```

public boolean hasCycle(ListNode head) {
    if(head == null || head.next == null) // 0 Nodes or 1 Node
        return false;
    if(head.next == head)
        return true; // Self Referential Node

    ListNode slow = head, fast = head;

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

        if(slow == fast) return true; // Cycle Detected
    }

    return false; // Cycle Not Detected
}

```

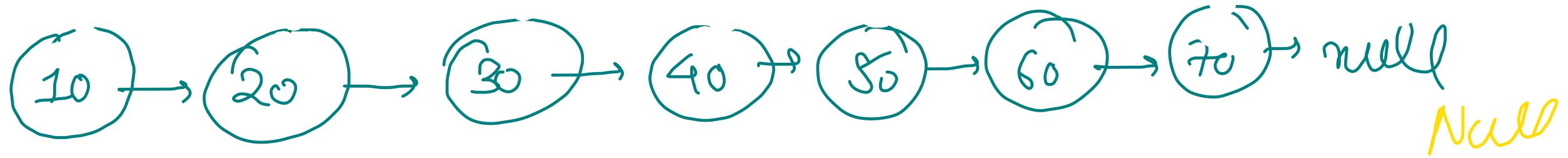
$\{$
 $\quad \text{if}(\text{slow} == \text{head})$
 $\quad \quad \text{CSLL}$
 $\quad \text{else tail+loop}$
 $\}$



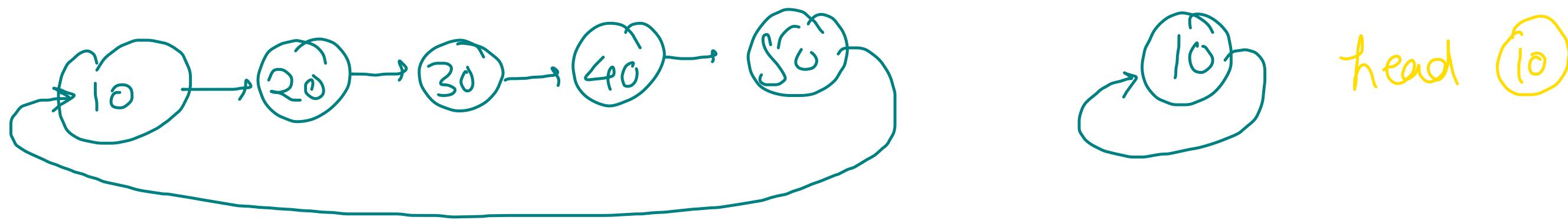
$O(1)$ auxiliary space
 $O(N)$ input space (N nodes)

142 Starting Node of Loop

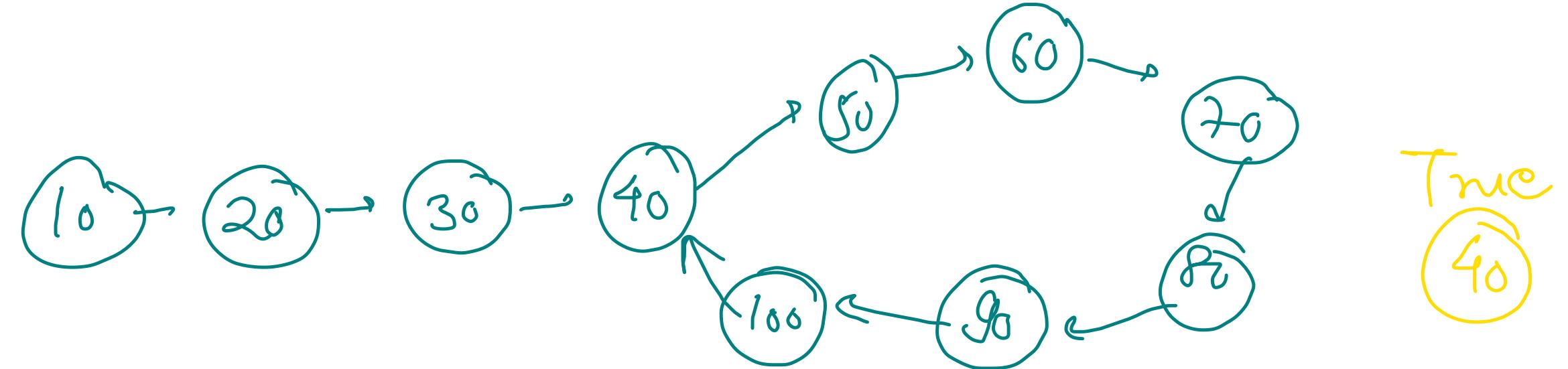
case 1)
SLL



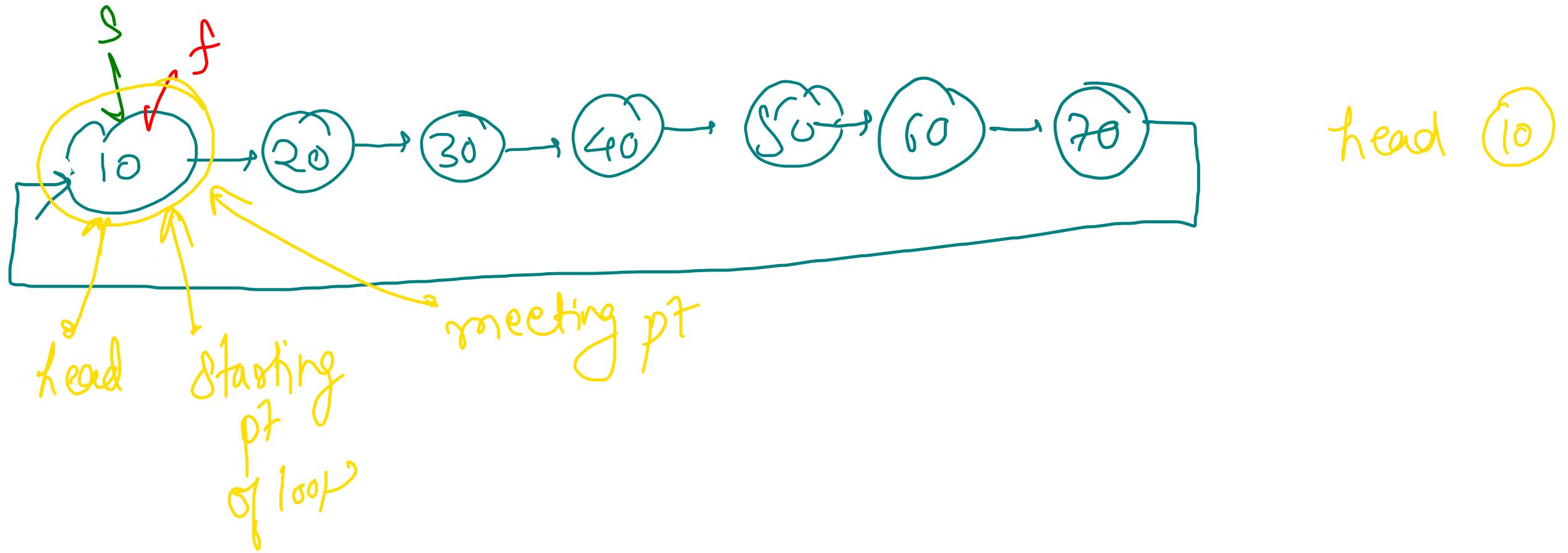
case 2)
CSLL

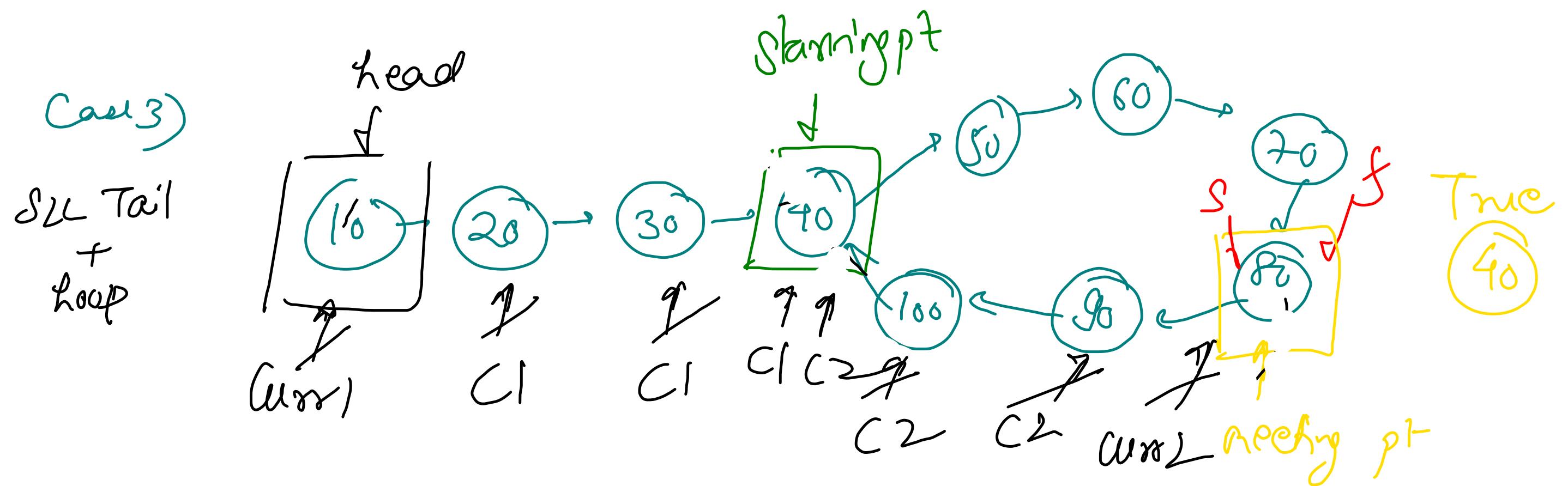


Case 3)
SLL Tail
+
Loop



case 2)
CSLL





Distance b/w starting pt &
head is equal to
distance b/w starting pt
& meeting point

```

public ListNode detectCycle(ListNode head) {
    ListNode slow = head, fast = head;

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

        if(slow == fast) break;
    }

    NO LOOP { if(fast == null || fast.next == null) SLL }

    loop detected {
        ListNode curr1 = head, curr2 = slow;
        while(curr1 != curr2){
            curr1 = curr1.next;
            curr2 = curr2.next;
        }
        return curr1;
    }
}

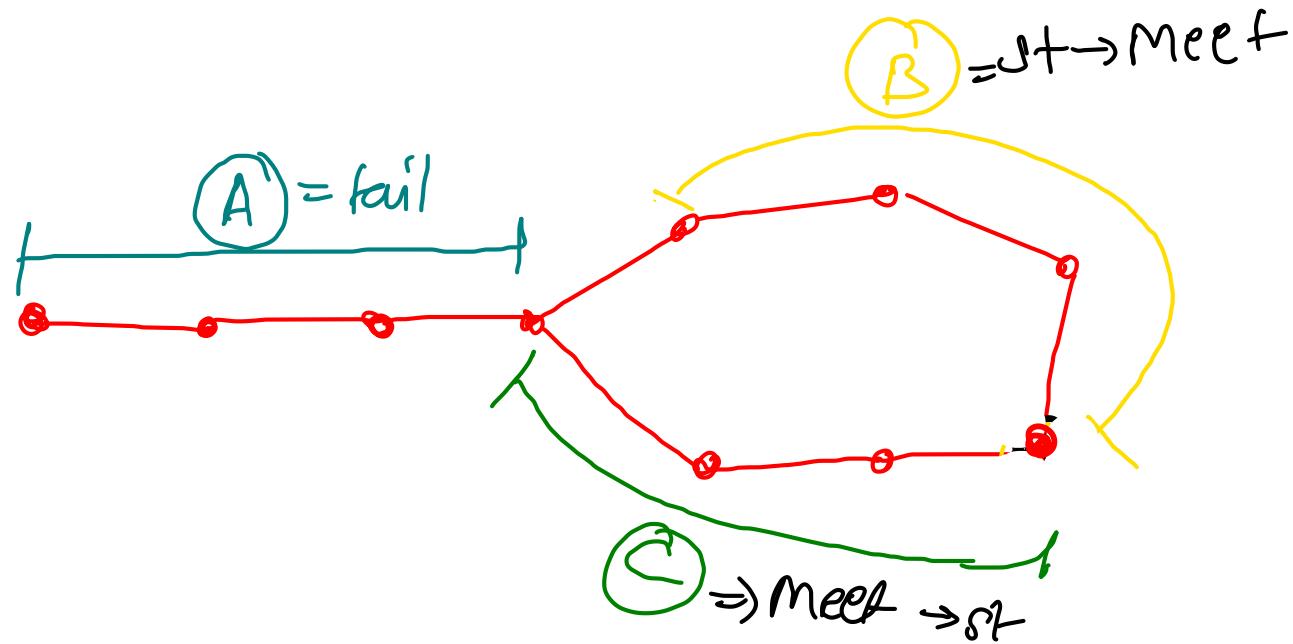
```

$O(N)$

Dist bw

H & S = M & S

Mathematical Proof



Slow = 1 node/it^r

fast = 2 nodes/it^r

Total time
taken by
Slow

Total time taken
by
fast

$$\frac{d_{\text{slow}}}{\text{Speed}_{\text{slow}}} = \frac{d_{\text{fast}}}{\text{Speed}_{\text{fast}}}$$

$$\frac{A + (B+C)i + B}{ss} = \frac{A + (B+C)j + B}{sf}$$

Length of loop = $(B+C)$

Length of tail = A

No of rotation taken by slow = i

No of start taken by fast = j

ss = Speed of slow

sf = speed of fast



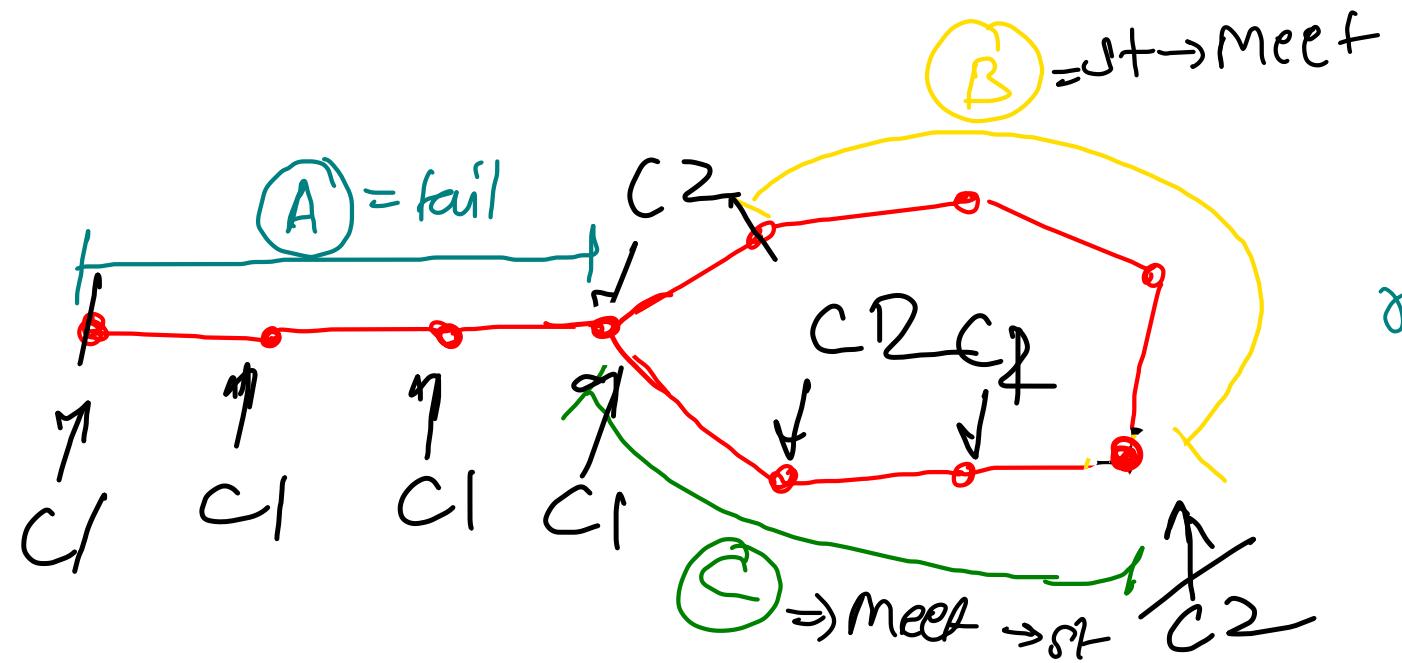
$$\frac{A + (B+C)i + \beta}{ss} = \frac{A + (B+C)j + \beta}{sf}$$

relative speed of fast w.r.t slow

$$\left\{ A + (B+C)i + \beta \right\} = A + (B+C)j + \beta$$

$$\frac{sf}{ss}$$

= γ



$$\gamma A + \gamma(B+C)^i + \gamma B = A + (B+C)^j + B$$

$$\gamma A + \gamma B - A - B = (B+C)(j - \gamma^i)$$

$$(\gamma - 1)A + (\gamma - 1)B = (B+C)(j - \gamma^i)$$

$$(\gamma - 1)(A + B) = B + C(j - \gamma^i)$$

$c / n/s$

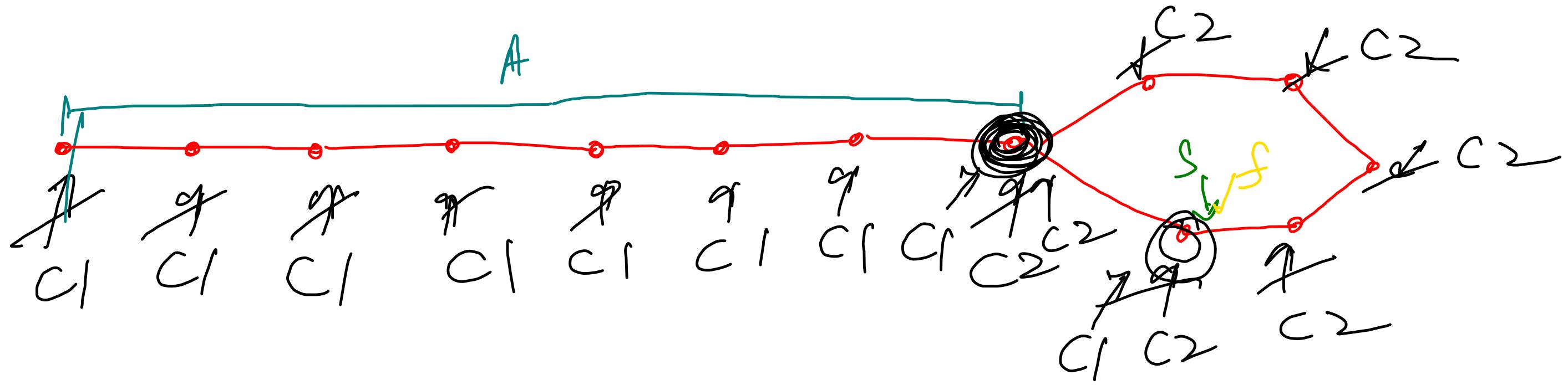
$A \rightarrow A \text{ sec}$

$C_2 \text{ ins}$
Cycles - fall short
 Ω_j

$$A + B = B + C \left\{ \frac{j - \gamma^i}{\gamma - 1} \right\}$$

$$\boxed{A = (B+C)k - B}$$

j tail k Some cycles - B

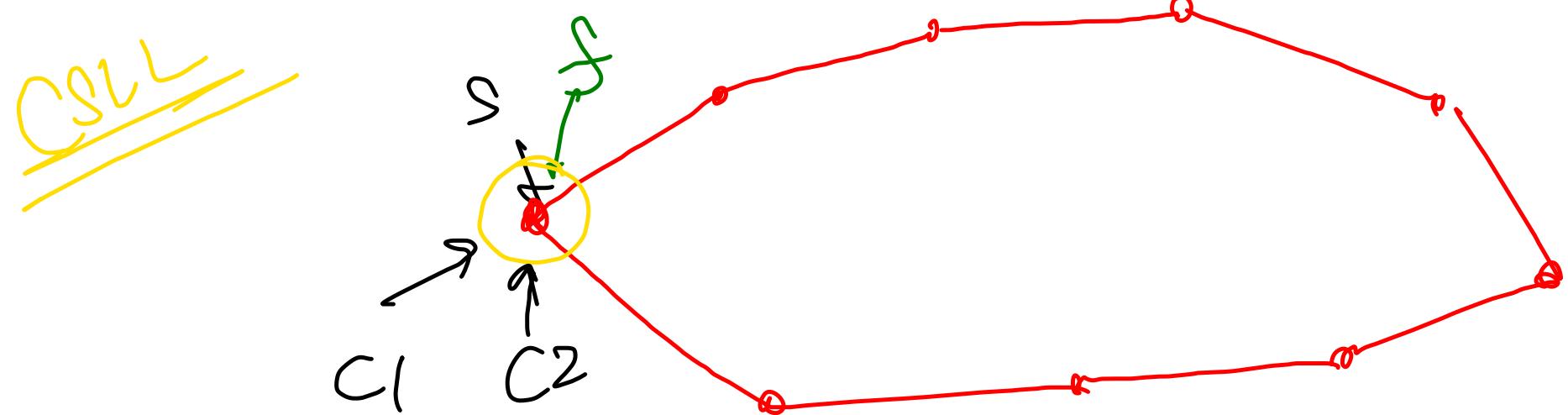


$$A = (B+C)k - B$$

$$A = (B+C) \boxed{k'} + C$$

$$A = (B+C)(k'+l) - B$$

$$\in (B+C)k' + (B+C)l$$



$$A = (B + C)k - B$$

$$(a) \quad A = \underbrace{(B + C)k'}_0 + \underbrace{C}_0$$

$$d_{\text{slow}} = A + \text{cycle}(i) + B$$

$\left\{ \begin{array}{l} S_{\text{fast}} = 2d_{\text{slow}} \\ t_{\text{slow}} = t_{\text{fast}} \end{array} \right.$

$$d_{\text{fast}} = A + \text{cycle}(j) + B$$

$\left\{ \begin{array}{l} +_{\text{slow}} = t_{\text{fast}} \end{array} \right.$

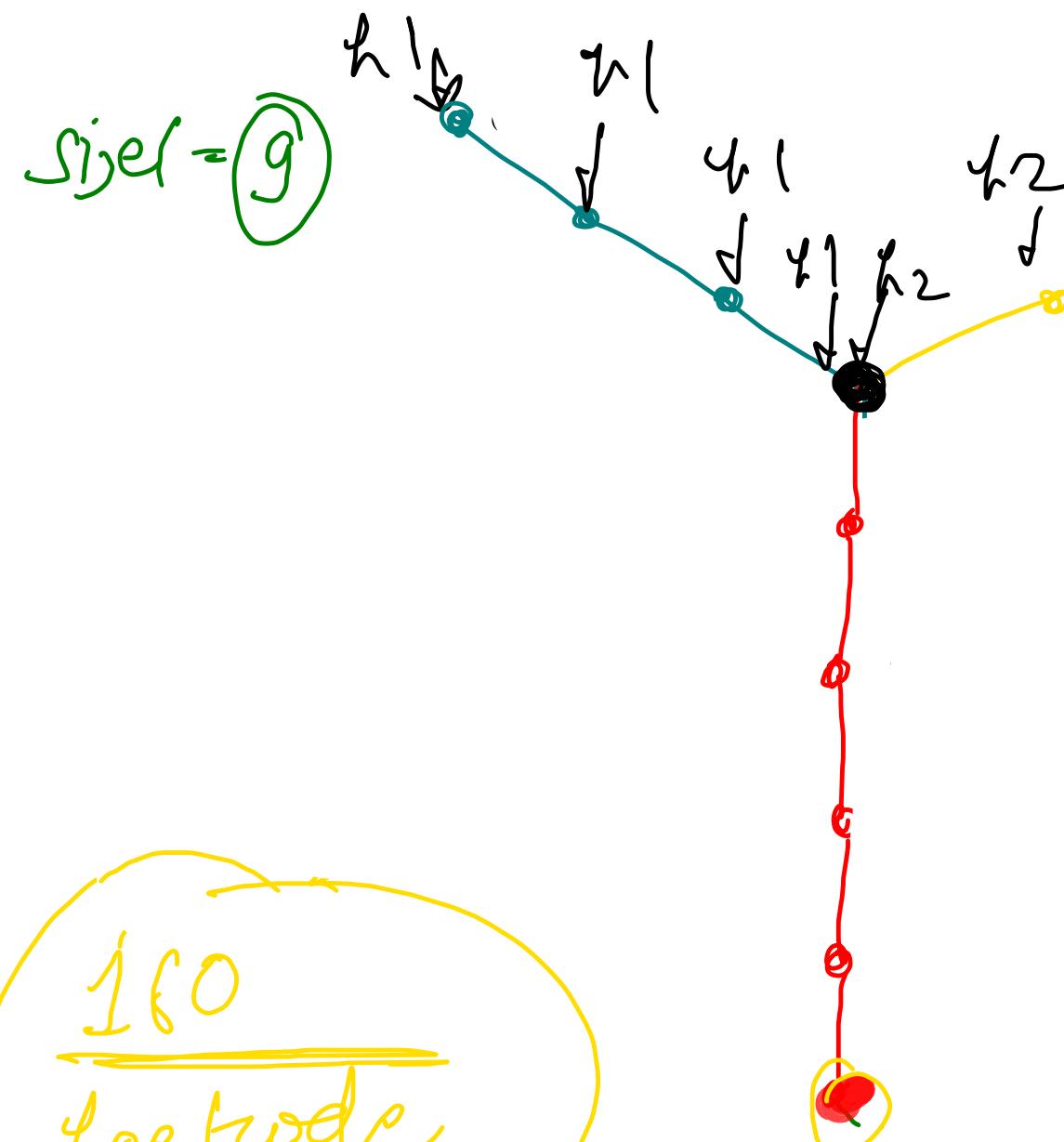
$$d_{\text{fast}} = 2 d_{\text{slow}}$$

$$A + \text{cycle}(j) + B = 2(A + \text{cycle}(i) + B)$$

$$A + \text{cycle}(j) + B = 2A + (2^i) + 2B$$

$$A = \text{cycle}(j - 2^i) - B$$

Intersection Point

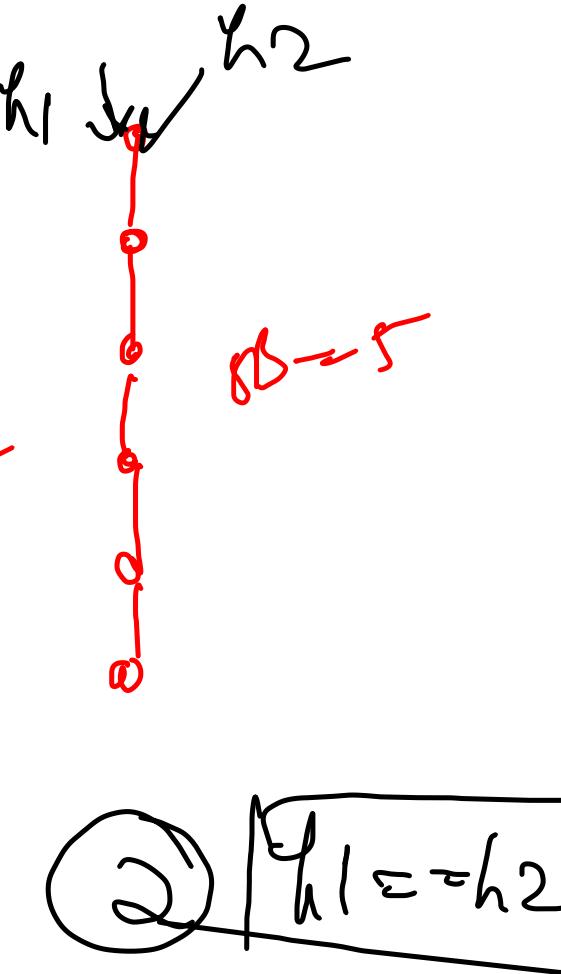
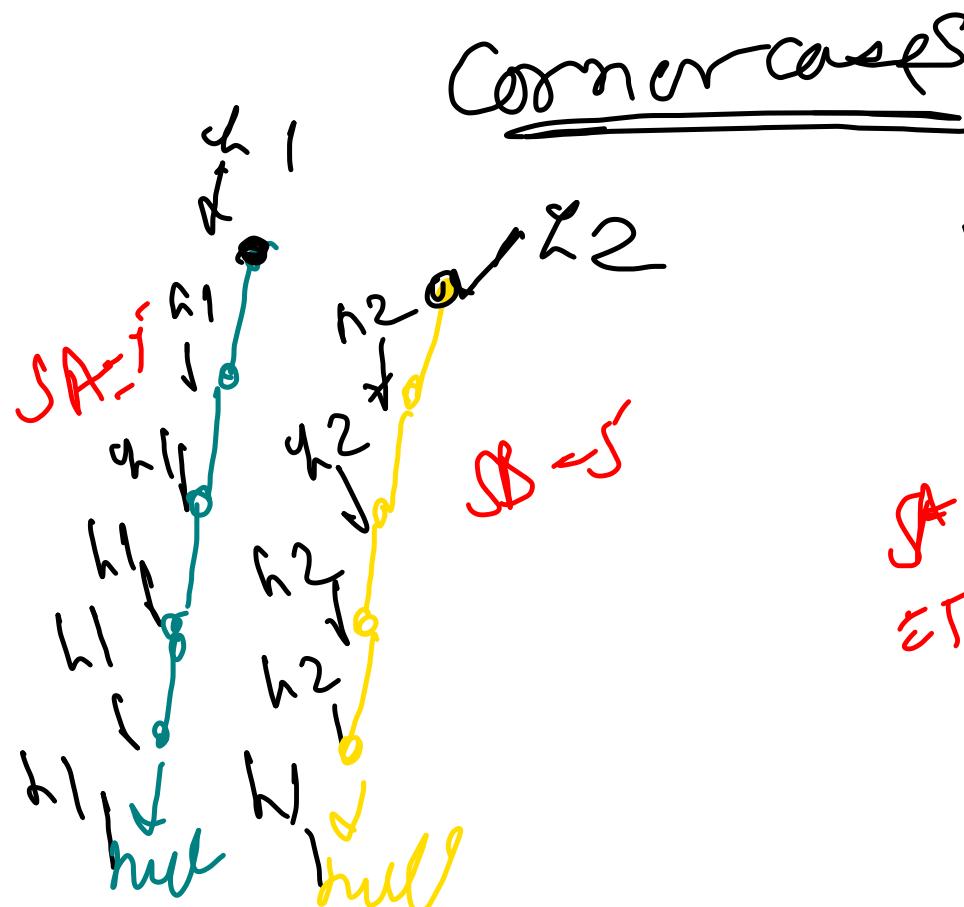


~~# using floyd's cycle~~

```
public ListNode detectCycle(ListNode head) { } 142

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    ListNode tailB = headB;
    while(tailB.next != null){
        tailB = tailB.next;
    }

    tailB.next = headB;
    ListNode ans = detectCycle(headA);
    tailB.next = null;
    return ans;
}
```



```

public int getSize(ListNode head){
    int size = 0;
    while(head != null){
        head = head.next;
        size++;
    }
    return size;
}

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    int sizeA = getSize(headA);
    int sizeB = getSize(headB);

    if(sizeA > sizeB){
        for(int i=0; i<sizeA-sizeB; i++)
            headA = headA.next;
    } else {
        for(int i=0; i<sizeB-sizeA; i++)
            headB = headB.next;
    }

    while(headA != headB){
        headA = headA.next;
        headB = headB.next;
    }
    return headA;
}

```

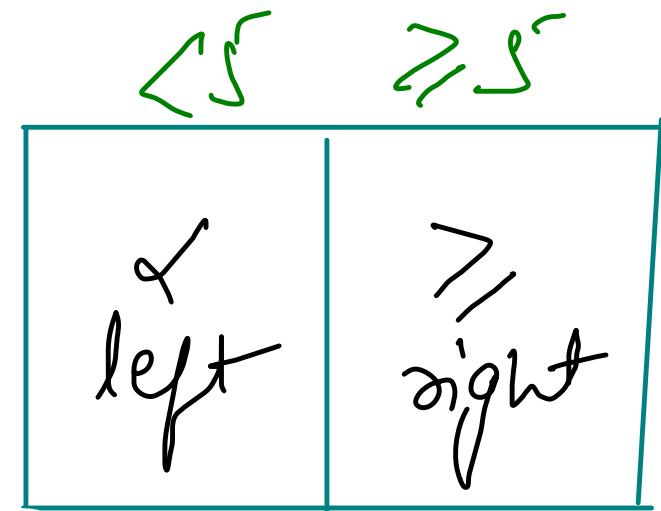
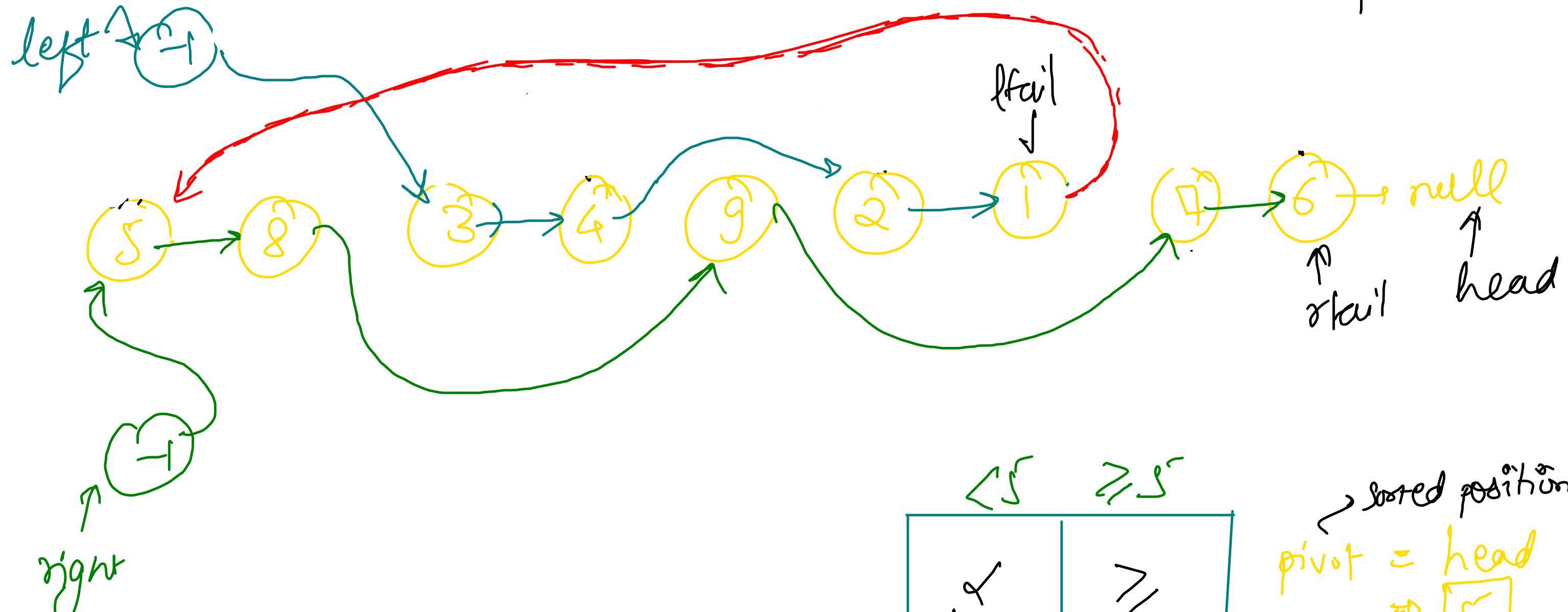
Lecture ⑥ { 3 PM to 6 PM - Sunday }

- Partition list 86 LC
- Sort 01^(IB), sort 012^(GFG)
- Quick sort on the 148 LC

Leetcode 86

Partition list → Stable

3, 4, 2, 1, 5, 8, 9, 7, 6



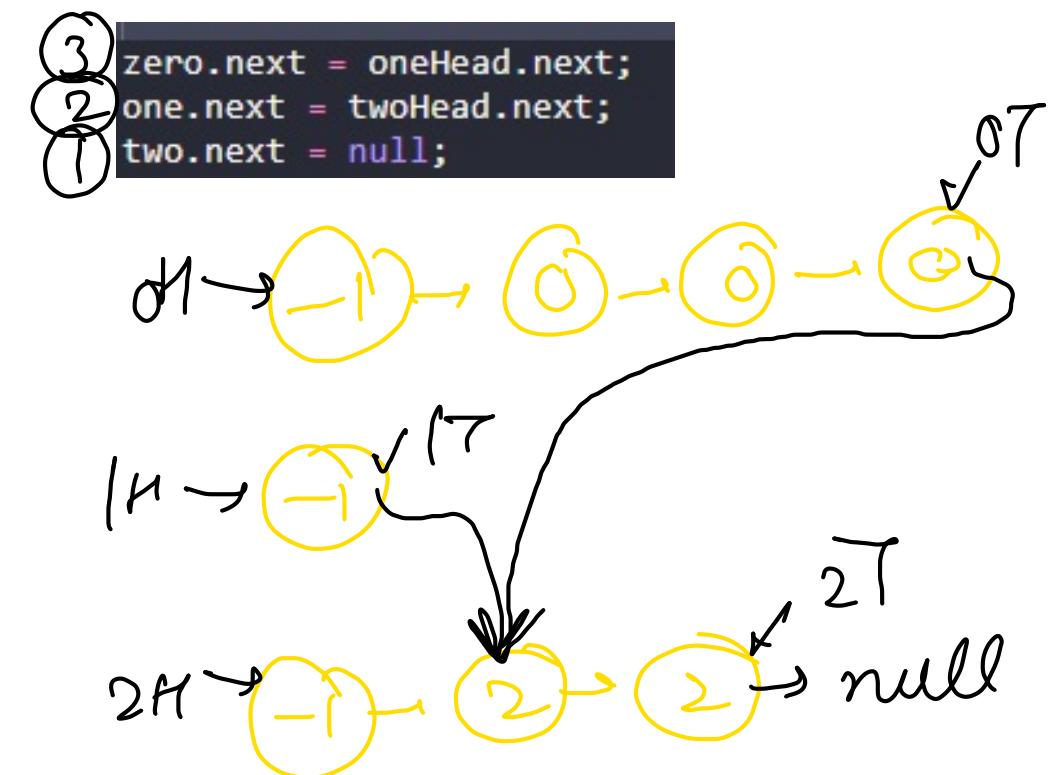
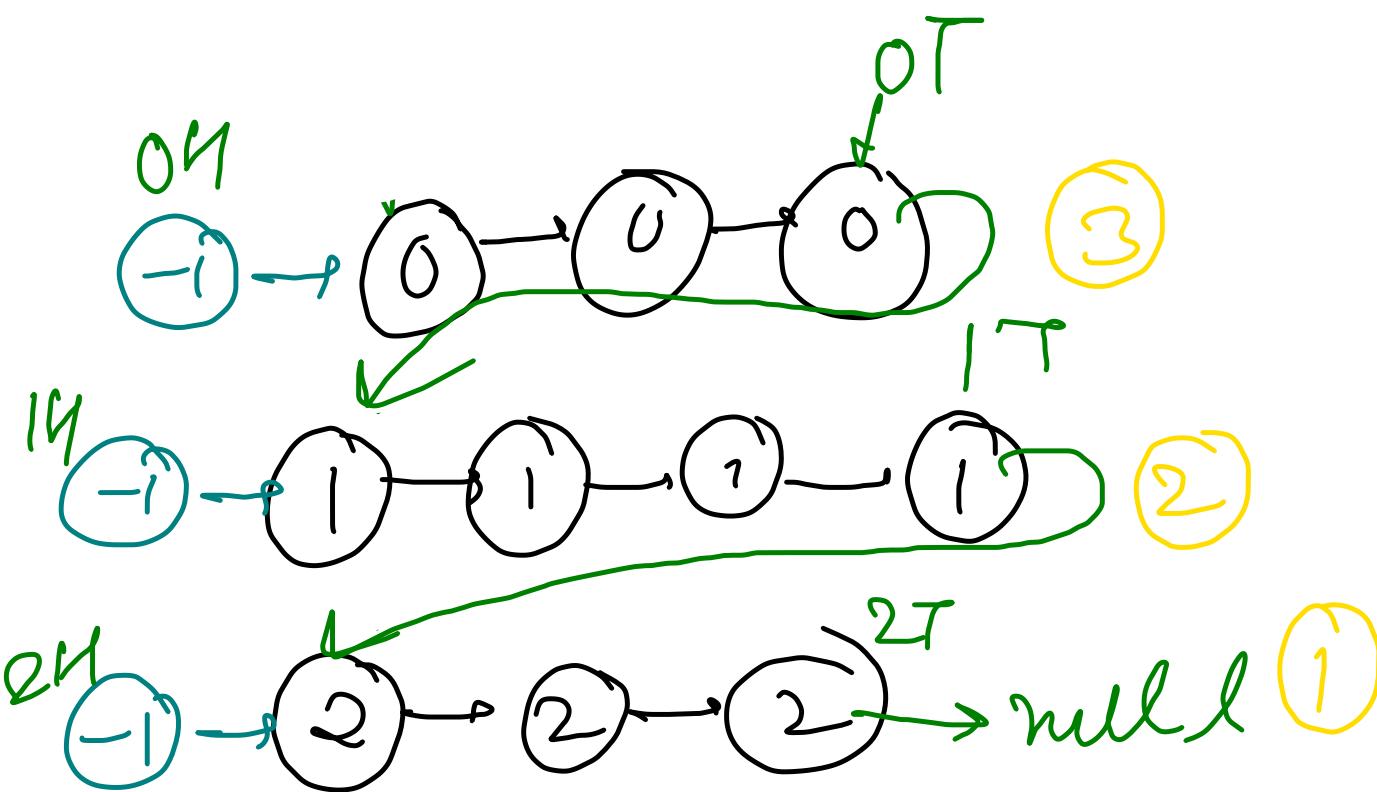
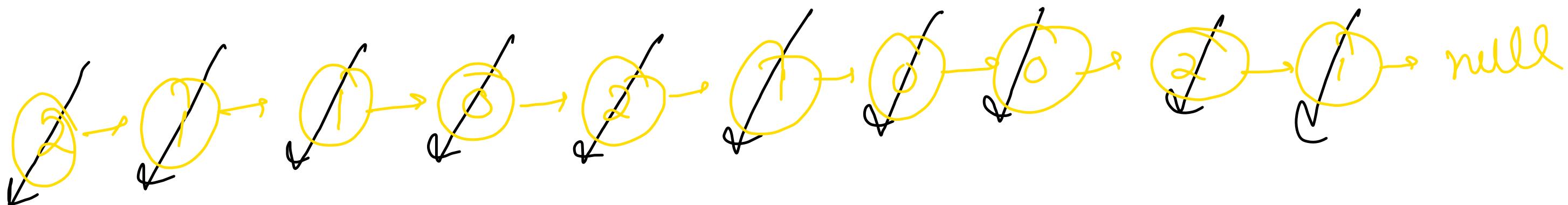
sorted position
pivot = head
⇒ 5

```
    *
public ListNode partition(ListNode head, int x) {
    ListNode lHead = new ListNode(-1);
    ListNode lTail = lHead;
    ListNode rHead = new ListNode(-1);
    ListNode rTail = rHead;

    while(head != null){
        if(head.val < x){
            lTail.next = head;
            lTail = head;
        } else {
            rTail.next = head;
            rTail = head;
        }
        head = head.next;
    }

    lTail.next = rHead.next;
    rTail.next = null;
    return lHead.next;
}
```

Sort 012



```
Node zeroHead = new Node(-1);
Node oneHead = new Node(-1);
Node twoHead = new Node(-1);
Node zero = zeroHead, one = oneHead, two = twoHead;

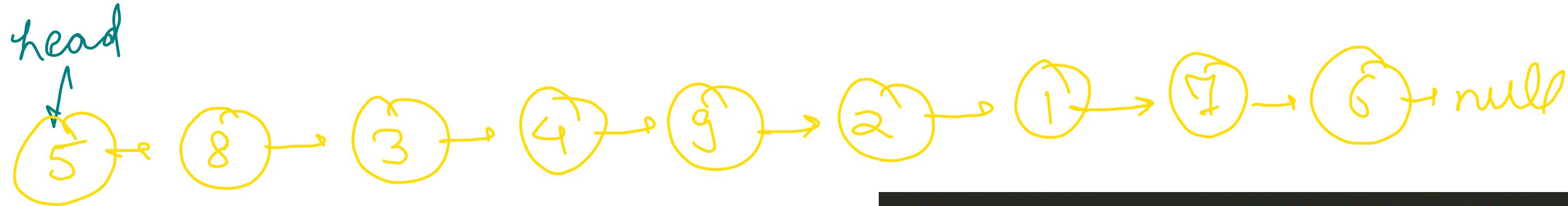
while(head != null){
    if(head.data == 0){
        zero.next = head;
        zero = head;
    } else if(head.data == 1){
        one.next = head;
        one = head;
    } else {
        two.next = head;
        two = head;
    }
    head = head.next;
}

two.next = null;
one.next = twoHead.next;
zero.next = oneHead.next;
return zeroHead.next;
```

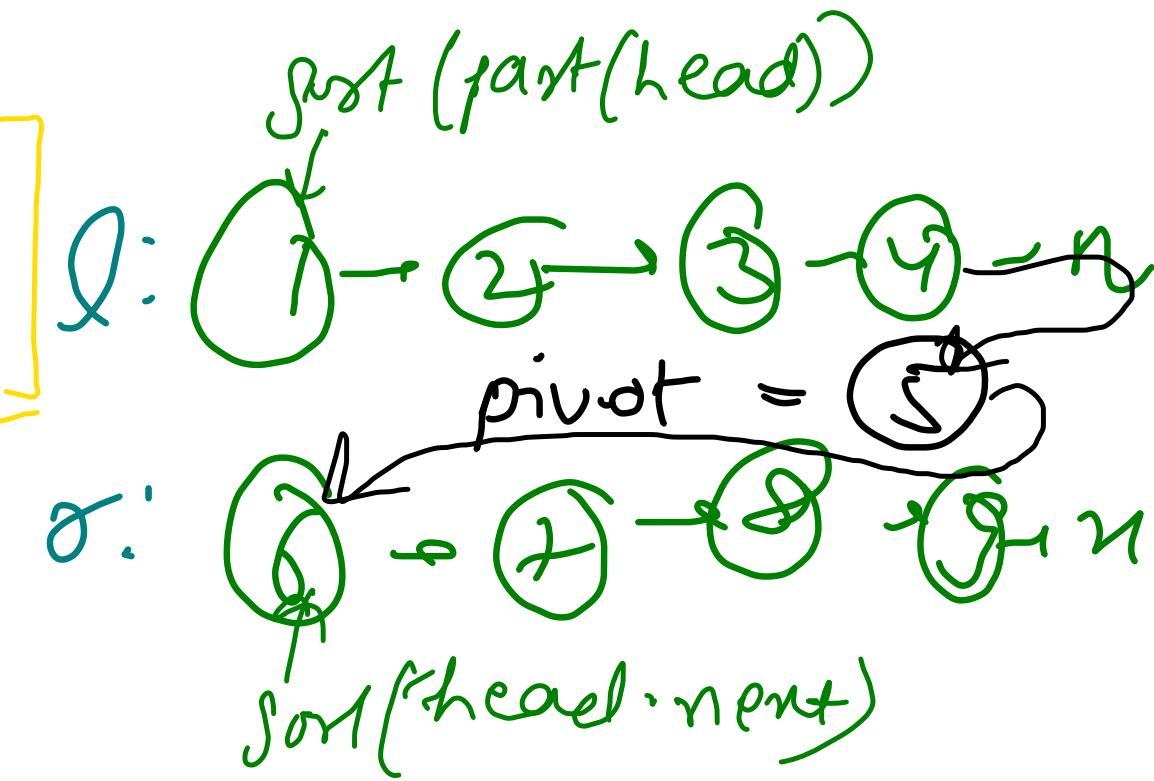
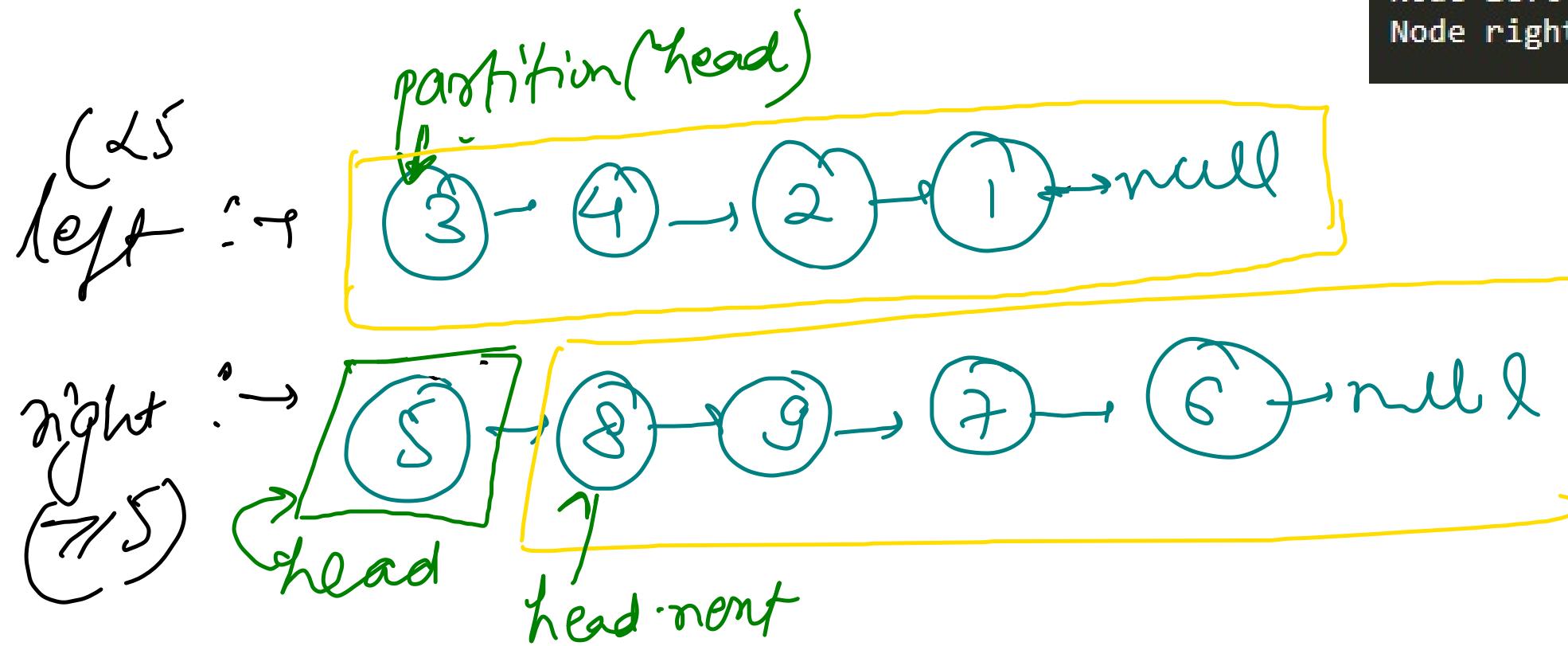
Sort LL using Quick Sort

148

(342) 5 (8976)



```
Node left = sortList(partition(head, head.val));
Node right = sortList(head.next);
```



```

public ListNode partition(ListNode head, int x) {
    ListNode lHead = new ListNode(-1);
    ListNode lTail = lHead;
    ListNode rHead = new ListNode(-1);
    ListNode rTail = rHead;

    while(head != null){
        if(head.val < x){
            lTail.next = head;
            lTail = head;
        } else {
            rTail.next = head;
            rTail = head;
        }
        head = head.next;
    }

    lTail.next = null;
    rTail.next = null;
    return lHead.next;
}

```

```

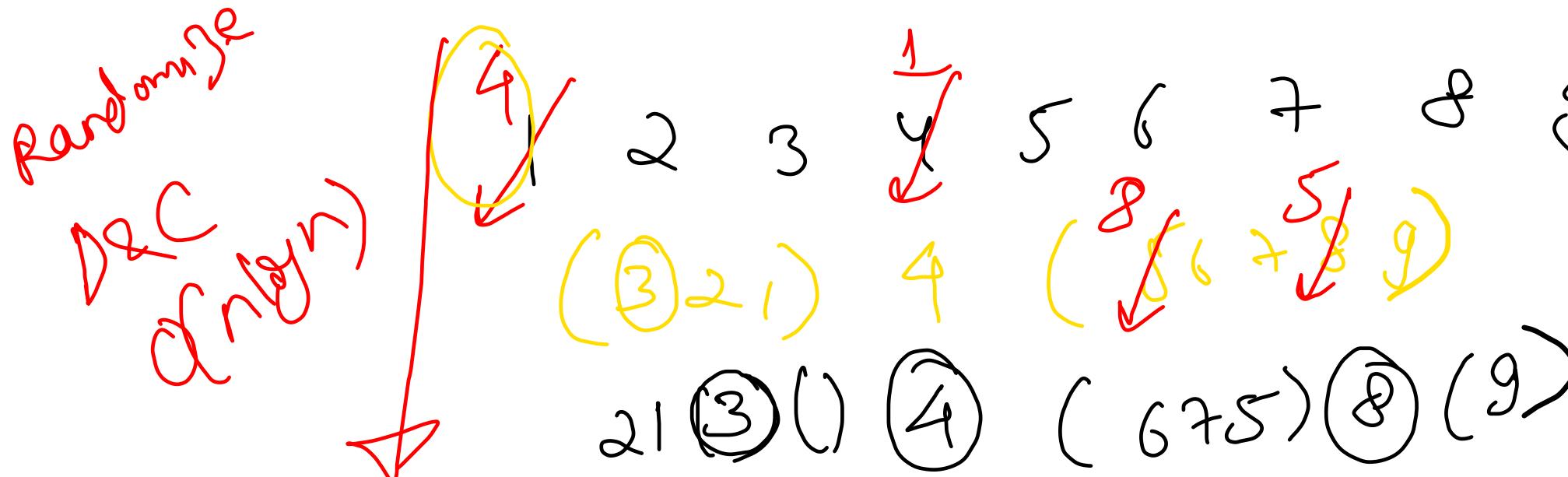
public ListNode getTail(ListNode curr){
    if(curr == null) return null;
    while(curr.next != null) curr = curr.next;
    return curr;
}

public ListNode sortList(ListNode head) {
    // base case
    if(head == null || head.next == null){
        return head;
    }
    randomize()
    ListNode left = sortList(partition(head, head.val));
    ListNode right = sortList(head.next);
    head.next = right;

    // left -> pivot (head) -> right
    if(left == null) return head;

    ListNode leftTail = getTail(left);
    leftTail.next = head;
    return left;
}

```



```
public void randomize(ListNode head){  
    int size = 0;  
    ListNode curr = head;  
    while(curr != null){  
        size++;  
        curr = curr.next;  
    }  
  
    int randomIdx = (new Random()).nextInt(size);  
  
    curr = head;  
    while(randomIdx-- > 0){  
        curr = curr.next;  
    }  
  
    int temp = head.val;  
    head.val = curr.val;  
    curr.val = temp;  
}
```

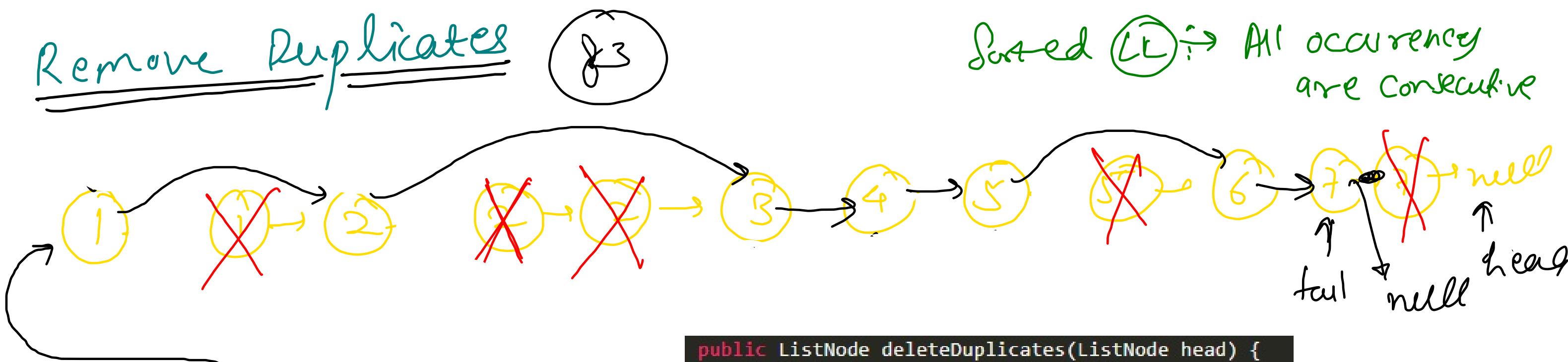
[0, size]

Linked list → Lecture ⑦ {Monday, 8:30 PM to 10:00 AM}

- ① Remove Duplicates → I & II
 - ② Fold hh, Unfold LL
 - ③ Odd Even hh $\begin{cases} \text{Index} \\ \text{Value} \end{cases}$
 - ④ Rotate list
- M, T, W, F \Rightarrow DSA
Thurs \Rightarrow DSA cont'd-
Weekends \Rightarrow Dev
Weekends + Thurs
 \Rightarrow optional ⑫

Remove Duplicates

83



yes
-1
↑
head

O(1) extra space

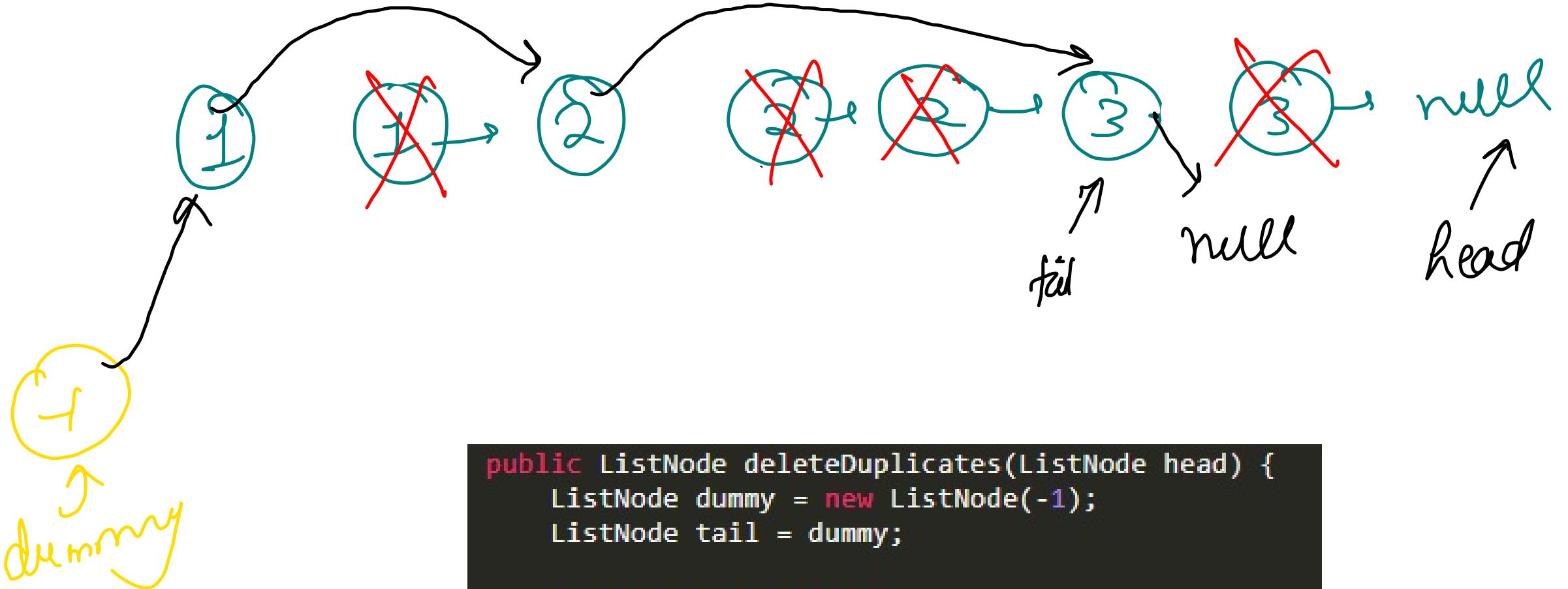
Sorted LL \Rightarrow All occurrence are consecutive

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(head != null){
        if(tail == dummy || head.val != tail.val){
            tail.next = head;
            tail = head;
        }
        head = head.next;
    }

    tail.next = null;
    return dummy.next;
}
```

} $O(N)$ Time Complexity



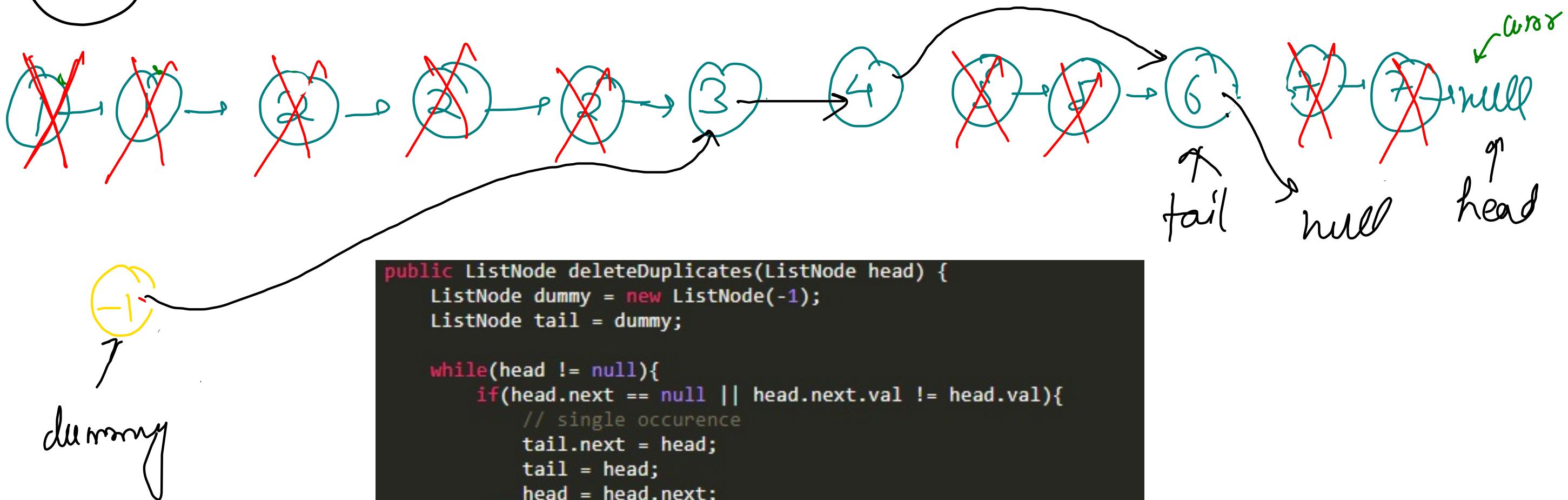
```
public ListNode deleteDuplicates(ListNode head) {
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(head != null){
        if(tail == dummy || head.val != tail.val){
            tail.next = head;
            tail = head;
        }
        head = head.next;
    }

    tail.next = null;
    return dummy.next;
}
```

82

Remove All Duplicates

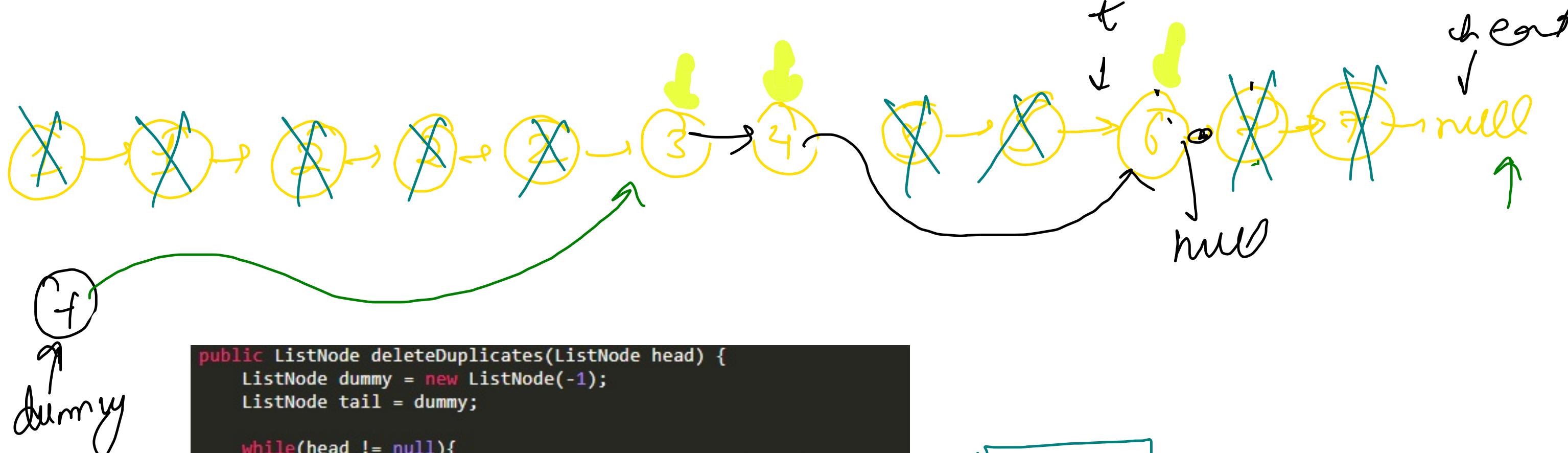


dummy

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(head != null){
        if(head.next == null || head.next.val != head.val){
            // single occurrence
            tail.next = head;
            tail = head;
            head = head.next;
        } else {
            // delete all occurrences of head.val including head
            ListNode curr = head;
            while(curr != null && curr.val == head.val){
                curr = curr.next;
            }
            head = curr;
        }
    }

    tail.next = null;
    return dummy.next;
}
```



```

public ListNode deleteDuplicates(ListNode head) {
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(head != null){
        if(head.next == null || head.next.val != head.val){
            // single occurrence
            tail.next = head;
            tail = head;
            head = head.next;
        } else {
            // delete all occurrences of head.val including head
            ListNode curr = head;
            while(curr != null && curr.val == head.val){
                curr = curr.next;
            }
            head = curr;
        }
    }

    tail.next = null;
    return dummy.next;
}

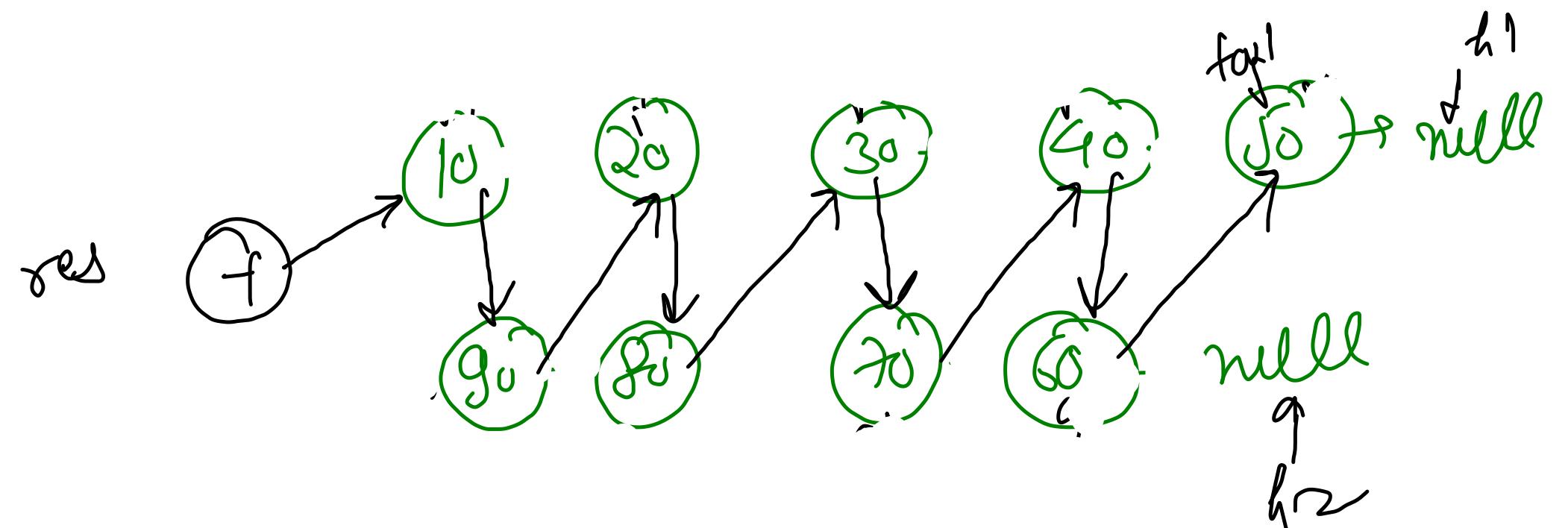
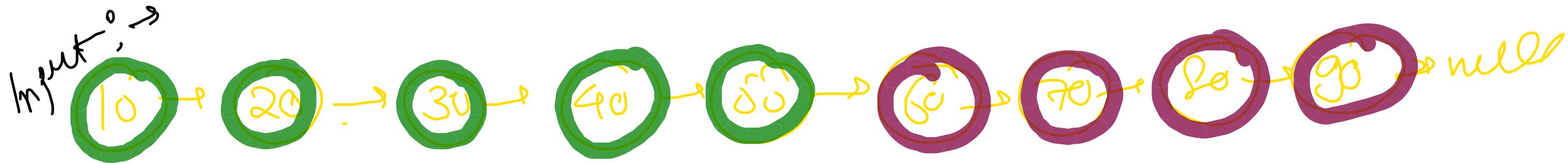
```

$O(N)$

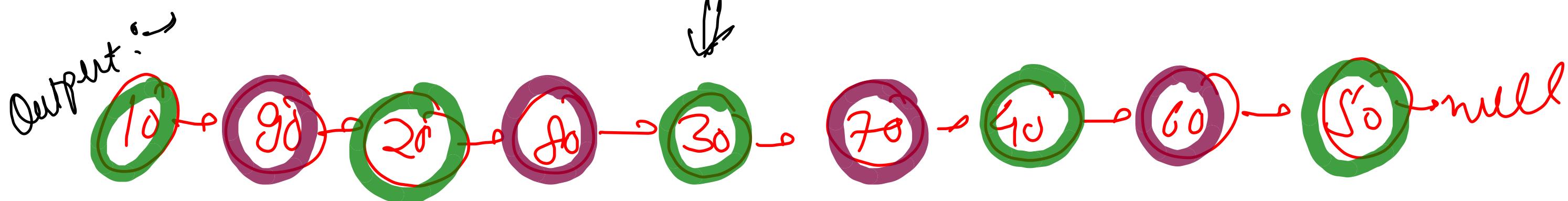
Fold of lh

{ Reorder list (43) }

WHY?



- (1) Find middle $\rightarrow \Theta(n)$
- (2) Second part - reverse $\rightarrow \Theta(n)$
- (3) Add Alternative Nodes $\rightarrow \Theta(N)$



```

public ListNode getMiddle(ListNode head){→}
public ListNode reverse(ListNode head){→}

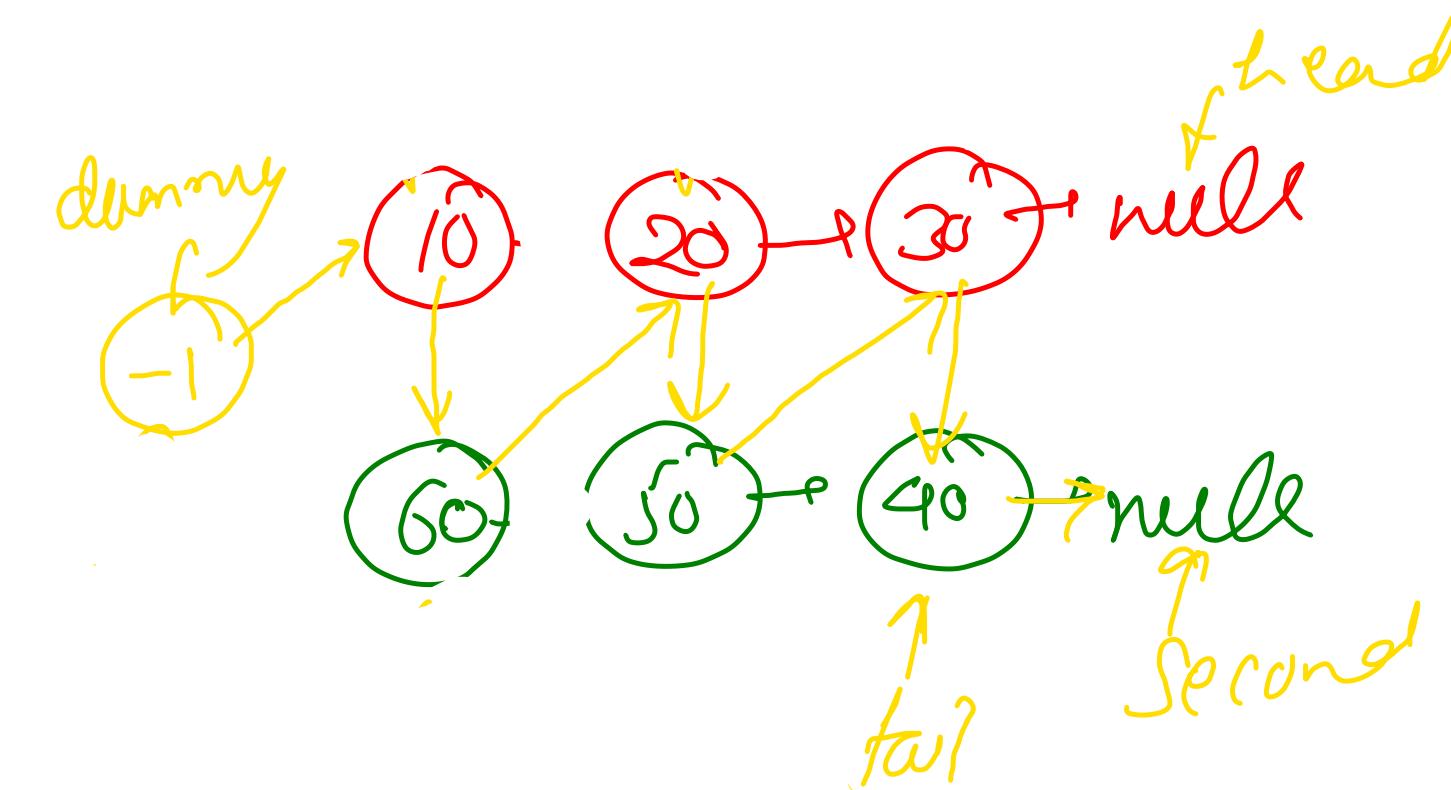
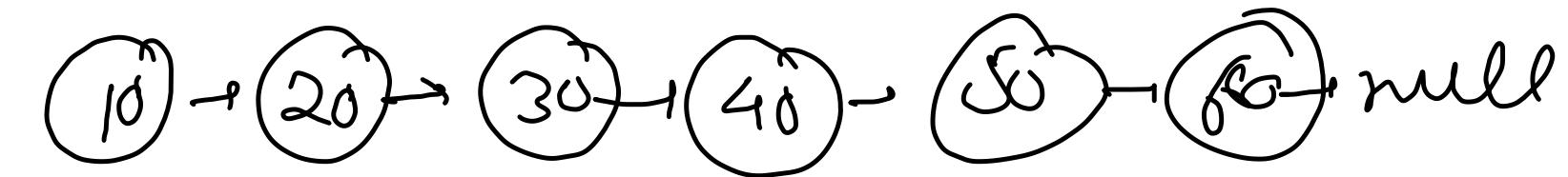
public void reorderList(ListNode head) {
    ListNode mid = getMiddle(head); → O(N)
    ListNode second = reverse(mid.next); → O(N)
    mid.next = null;

    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

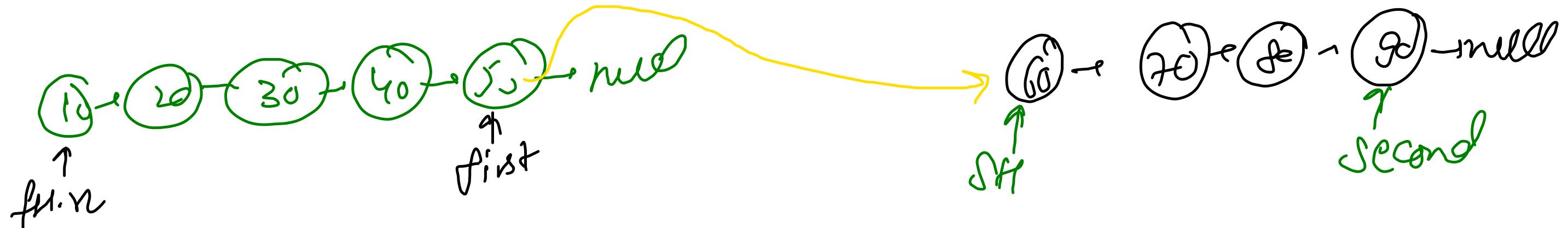
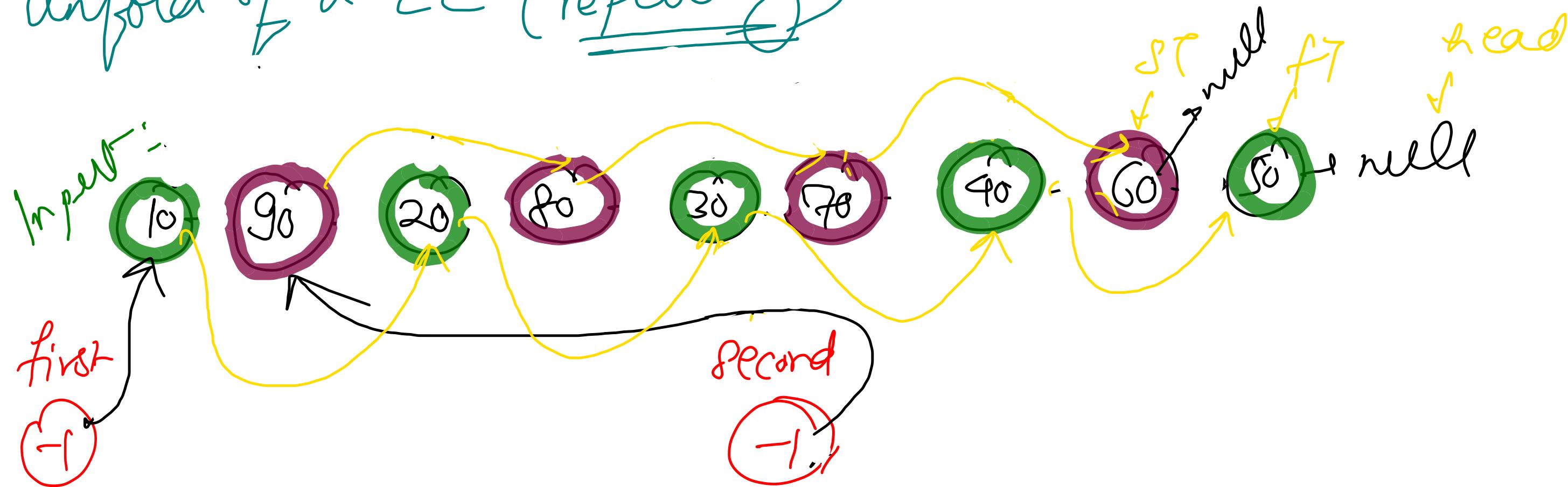
    while(head != null || second != null){
        if(head != null){
            tail.next = head;
            tail = head;
            head = head.next;
        }
        if(second != null){
            tail.next = second;
            tail = second;
            second = second.next;
        }
    }

    tail.next = null;
    head = dummy.next;
}

```



Unfold of a LC (repcoiling)



```

public static ListNode reverse(ListNode head){ }

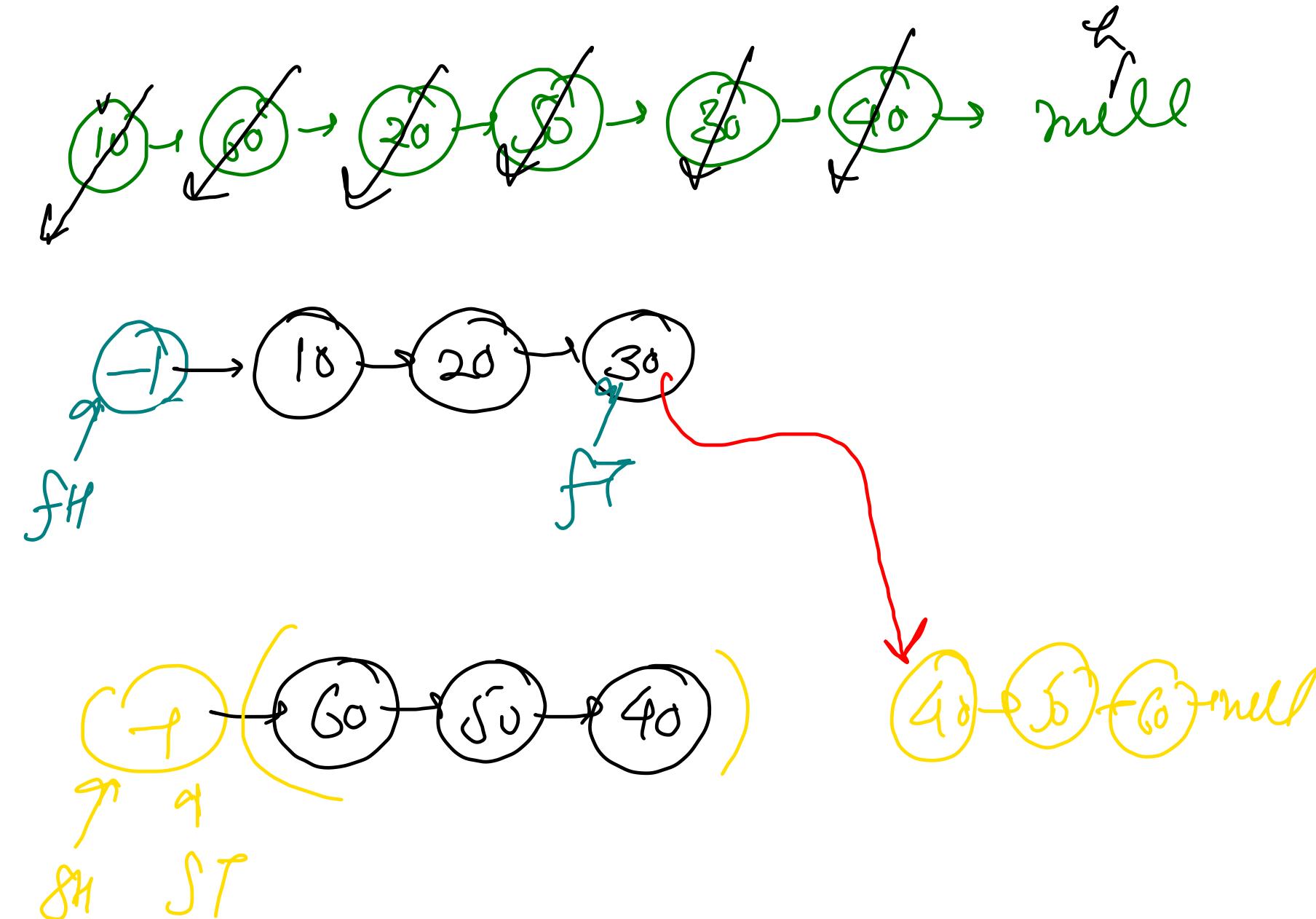
public static void unfold(ListNode head) {
    ListNode firstHead = new ListNode(-1);
    ListNode secondHead = new ListNode(-1);
    ListNode first = firstHead, second = secondHead;

    while(head != null){
        first.next = head;
        first = head;
        head = head.next;

        if(head != null){
            second.next = head;
            second = head;
            head = head.next;
        }
    }

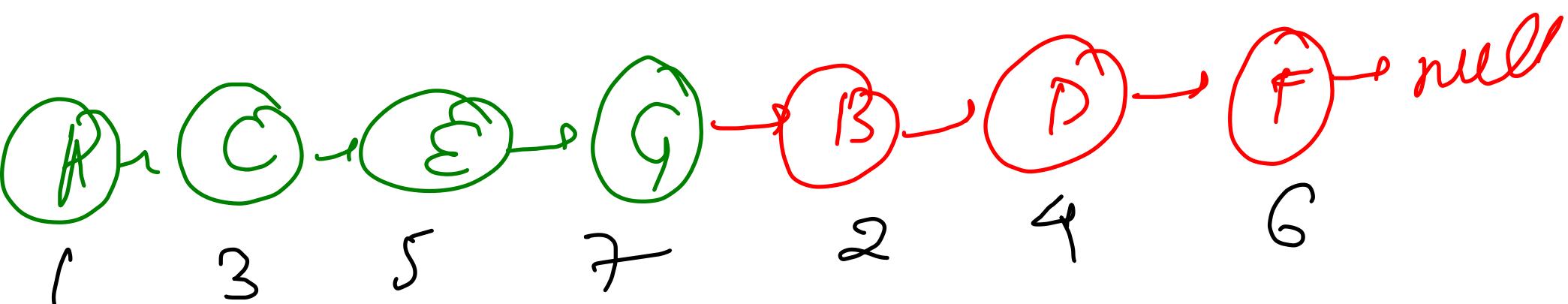
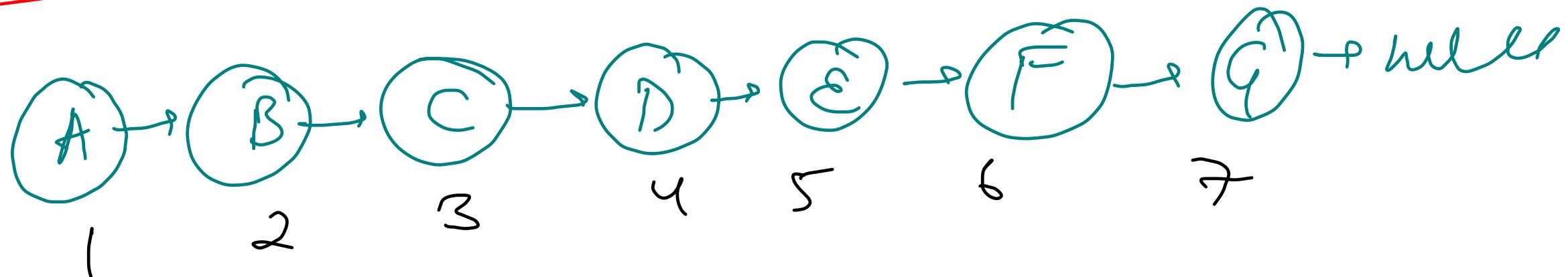
    first.next = null;
    second.next = null;
    head.val = 1
    first.next = reverse(secondHead.next);
    head = first.next;
}

```



Odd Even Ll (By Index)

{ Anfold }



```
public ListNode oddEvenList(ListNode head) {
    ListNode firstHead = new ListNode(-1);
    ListNode secondHead = new ListNode(-1);
    ListNode first = firstHead, second = secondHead;

    while(head != null){
        first.next = head;
        first = head;
        head = head.next;

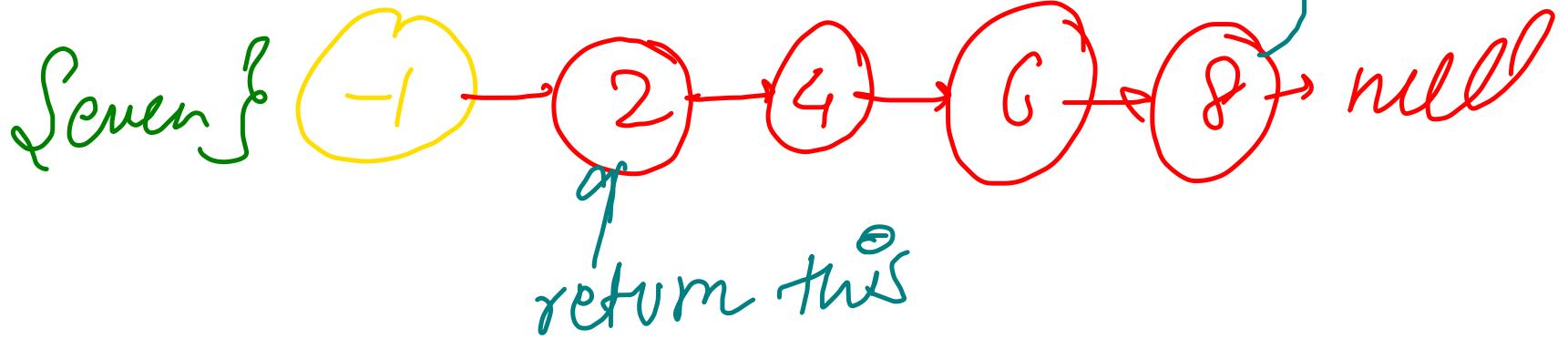
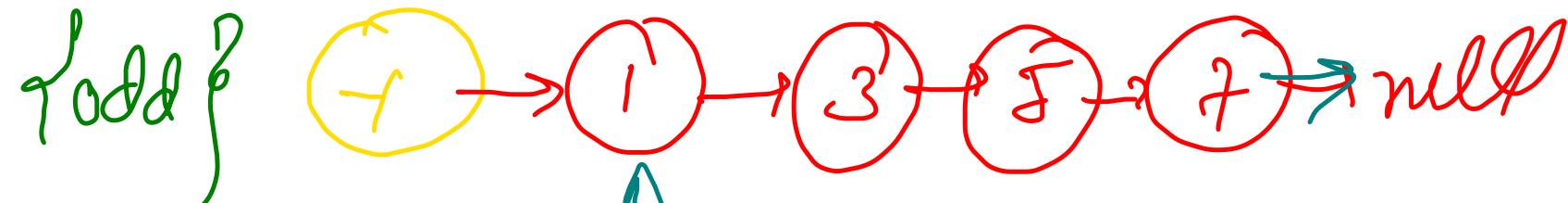
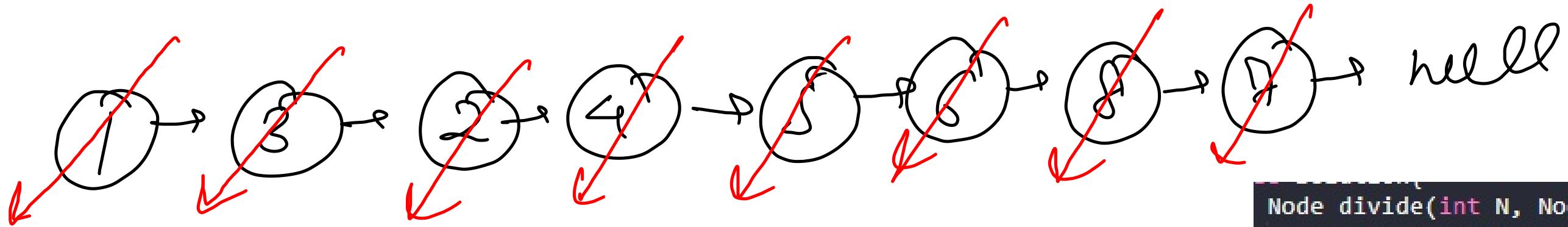
        if(head != null){
            second.next = head;
            second = head;
            head = head.next;
        }
    }

    second.next = null;
    first.next = secondHead.next;
    return firstHead.next;
}
```

Odd Even Lh {By value}

Sort O1

false



```
Node divide(int N, Node head){  
    Node oddHead = new Node(-1);  
    Node evenHead = new Node(-1);  
    Node odd = oddHead, even = evenHead;  
  
    while(head != null){  
        if(head.data % 2 == 1){  
            odd.next = head;  
            odd = head;  
        } else {  
            even.next = head;  
            even = head;  
        }  
        head = head.next;  
    }  
  
    odd.next = null;  
    even.next = oddHead.next;  
    return evenHead.next;  
}
```