linked list (level 1 + level 2)

Lecture ① {Monday}

→ LL Basics

→ Display , Get At

→ Add
- last
- First

→ Remove
- last
- First

Lecture ② {Tuesday}

→ Add At

→ Remove At

LC 707 Design LL

→ Reverse linked list LC 206

- → Data Iterative $O(N^2)$
- → Pointer Iterative $O(N)$
- → Data Recursive $O(N)$
  - Local
  - Return
  - Static

Lecture (3) {Wednesday}

→ Display Reverse

→ Reverse ll
Pointer Recursion

→ k Reverse — LC HARD 25

→ Middle Node — LC 876

→ Kth node from End — GFG

→ Palindrome ll — LC 234

Lecture (4) {Thursday}

→ merge ll

→ merge 2 Sorted — LC 21

→ merge K Sorted — LC HARD 23

→ merge Sort — LC 148

→ Remove Duplicates (1&1) — LC 83 & 82

→ Fold & Unfold ll — LC 143 & Pepcoding

{ Friday 8:30 to 12 }

lecture (5)

→ Partition Lth

→ Partition around last Idx
→ Partition around pivot
→ Odd Even Lth
→ Quick Sort Lth

→ Flatten Lth (1&11)

→ Wave Sort Lth

→ Rotate List

{ Sunday 9 AM to 12 PM }

lecture (6)

→ Floyd's cycle
→ Cycle Detection
→ Starting pt of cycle
→ Mathematical Proof

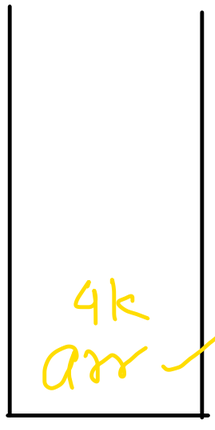
→ Clone Lth
→ With extra space
→ w/o extra space

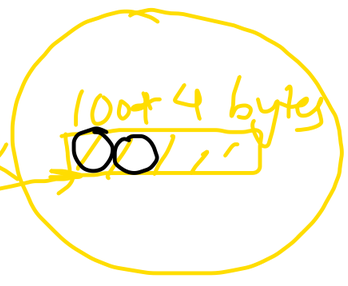
→ Insertion Node

ArrayList is an array of dynamic array → contiguous

Arrays : Contiguous memory Allocation

int[] arr = new int[100];



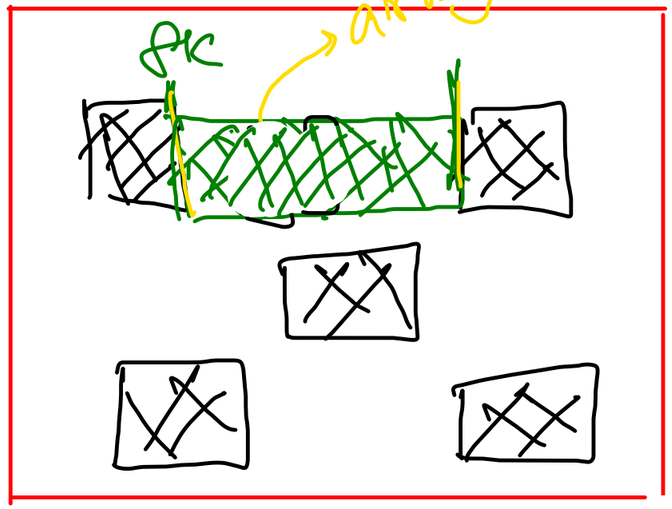


O(1) get

Random Access

16 integer

array max size



Memory is available

⇓

fragmentation

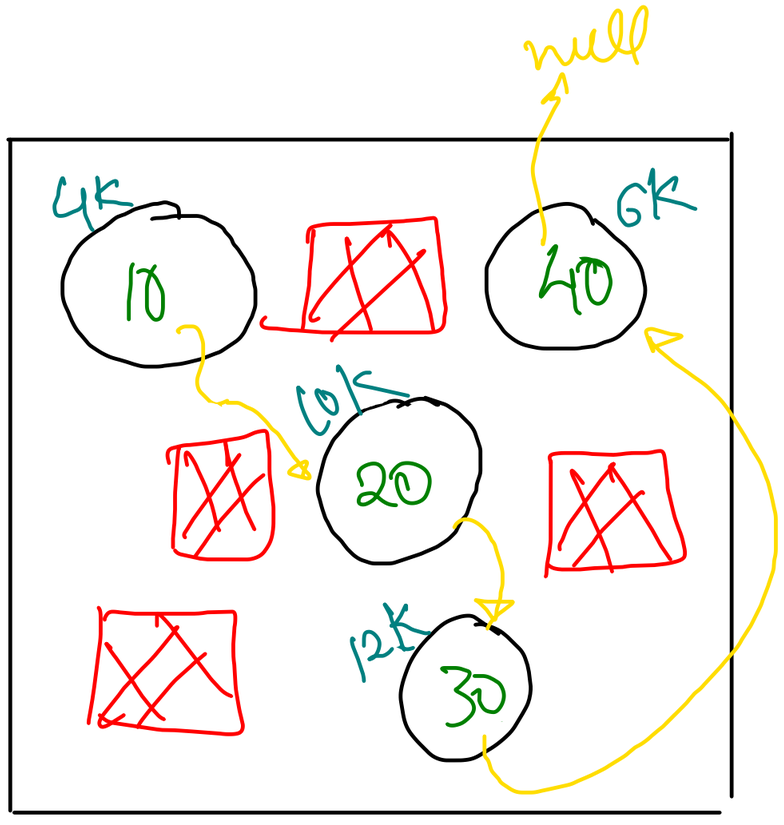
16 byte

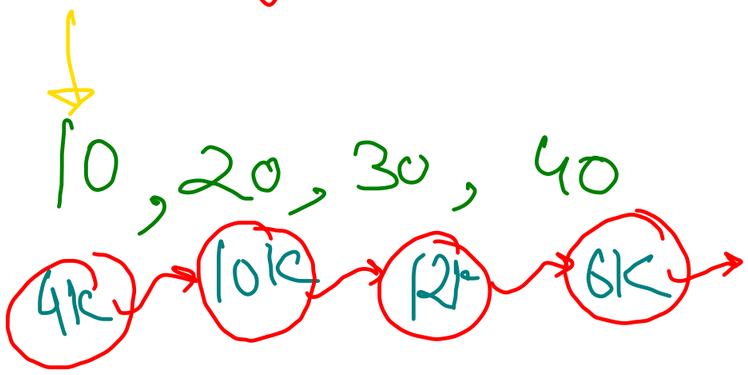
4k {10, 20, 30, 40}

Base index, index, size

Address of ith ele = Base Add + (i*size)

Singly linked list :→ Non contiguous memory allocation





class
Functions
Data member

public static Node{

int data;

Node next;

extra memory

}

public static
linkedList{

Node head;

Node tail;

int size;

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

```java
public static class LinkedList{
    Node head;

    public void display(){

    }

    public void addFirst(int val){

    }

    public void addLast(int val){

    }

    public int get(int idx){

    }
}
```

Node fail; int size

Main( ) {

→ LinkedList list = new LinkedList();

→ list.addLast(10);

→ list.addLast(20);

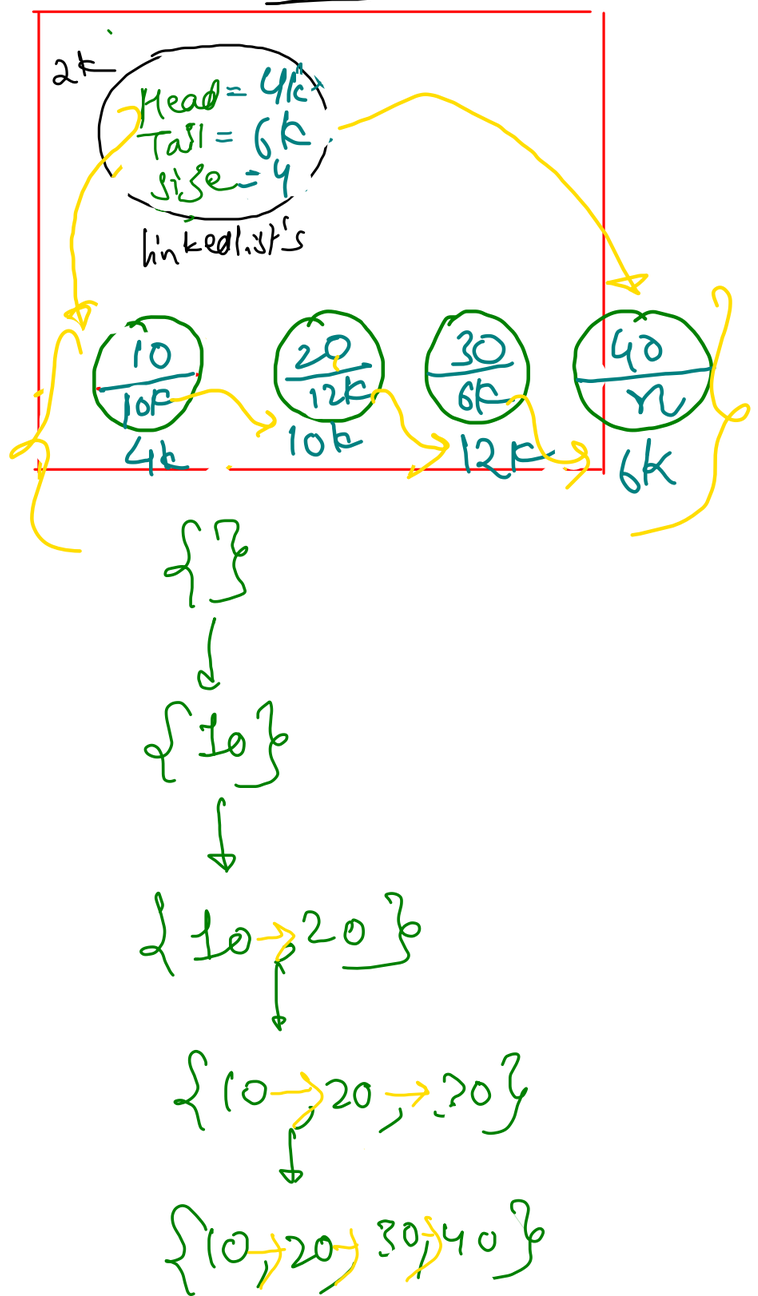
→ list.addLast(30);

→ list.addLast(40);

Stack

list = 2k

heap



Array

16 bytes

vs

LL

4 nodes → (4 bytes + 4 bytes) * 4

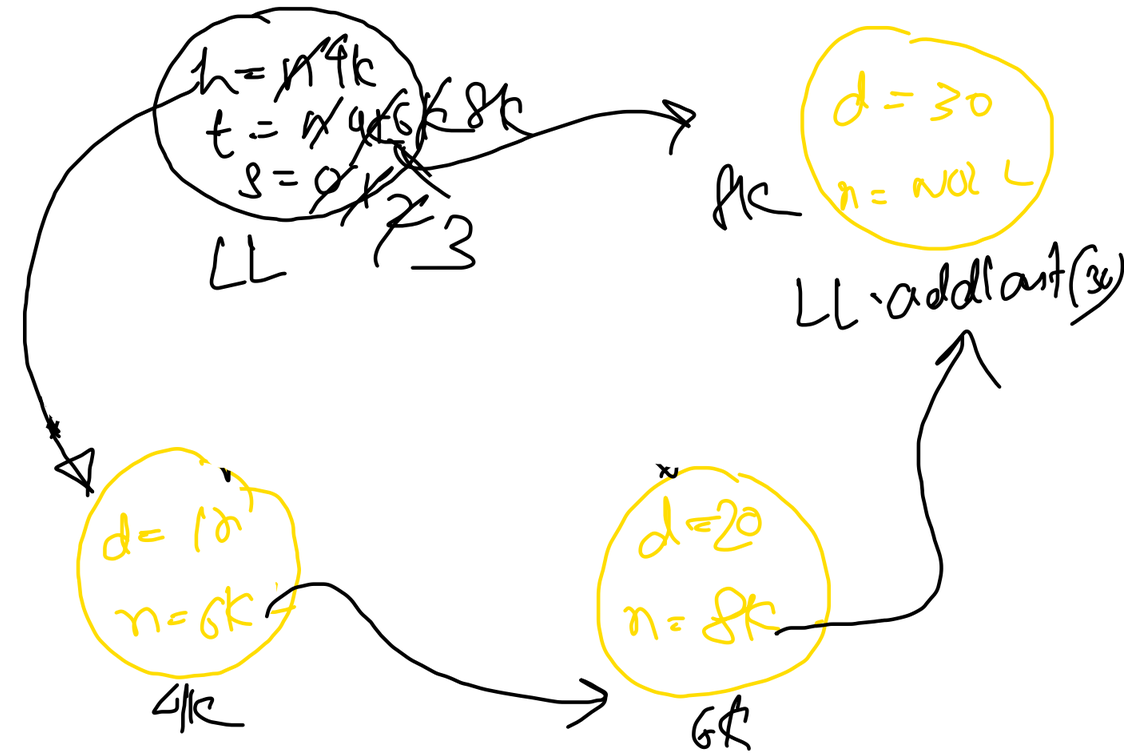
LL obj → (8+8+4) = 20 bytes

```java
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++;
  }
}
```

tail
O(1)

If tail
was
not given
⇒ addLast would have costed O(n)

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



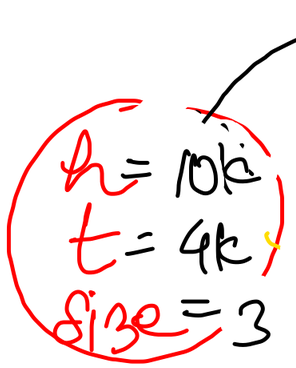
Disadvantages of Linked list over Array

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

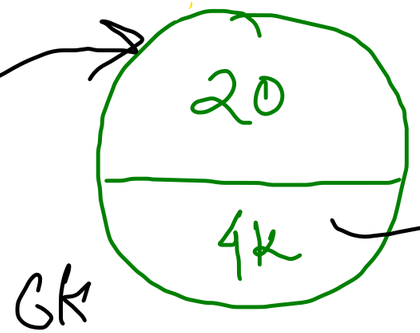
Advantages of Linked list over Array

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

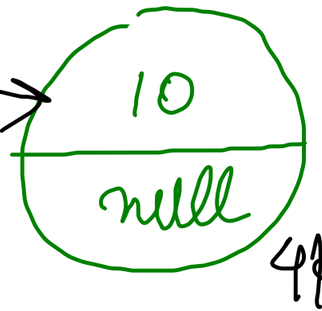
```
LinkedList list = new LinkedList();
```



LL.addfirst(30)



LL.addfirst(20)

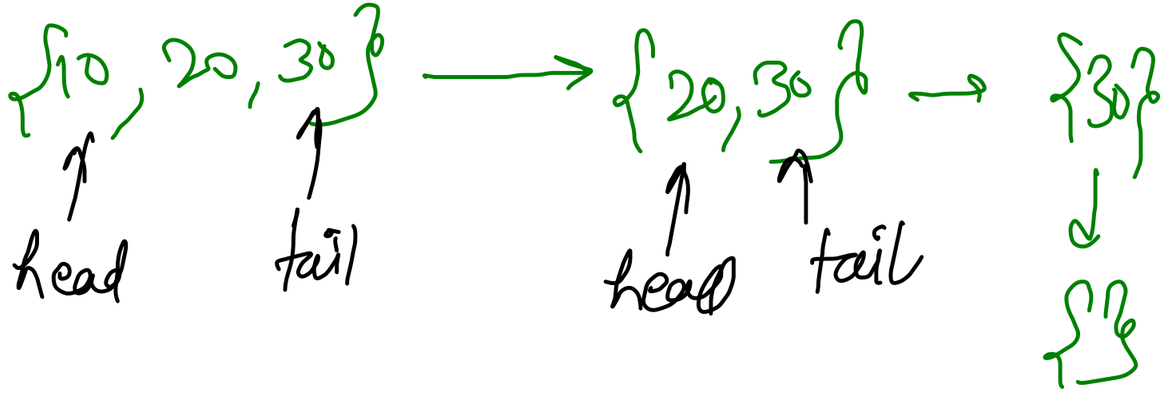


LL.addfirst(10)

{} + {10} → {20,10}

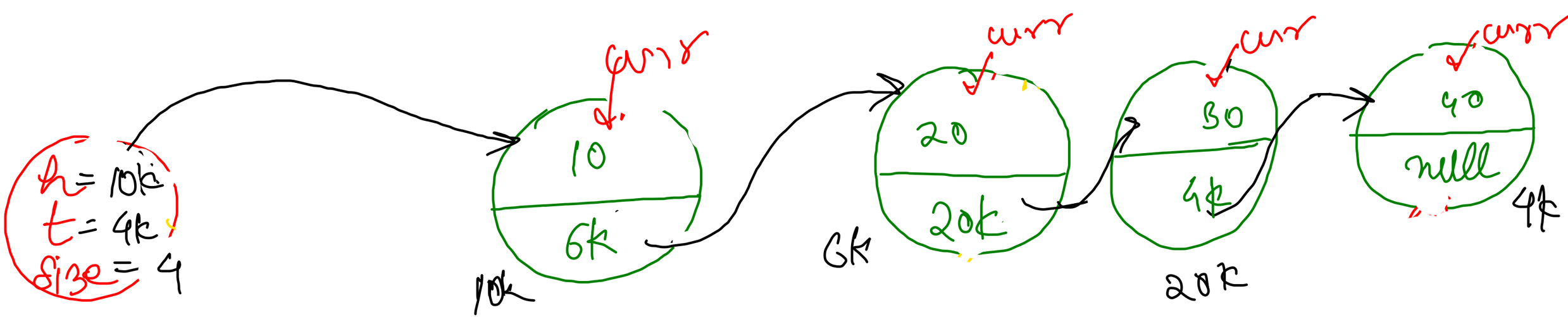
{ 30,20,10 }

```
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++;
}
```




```java
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 LL

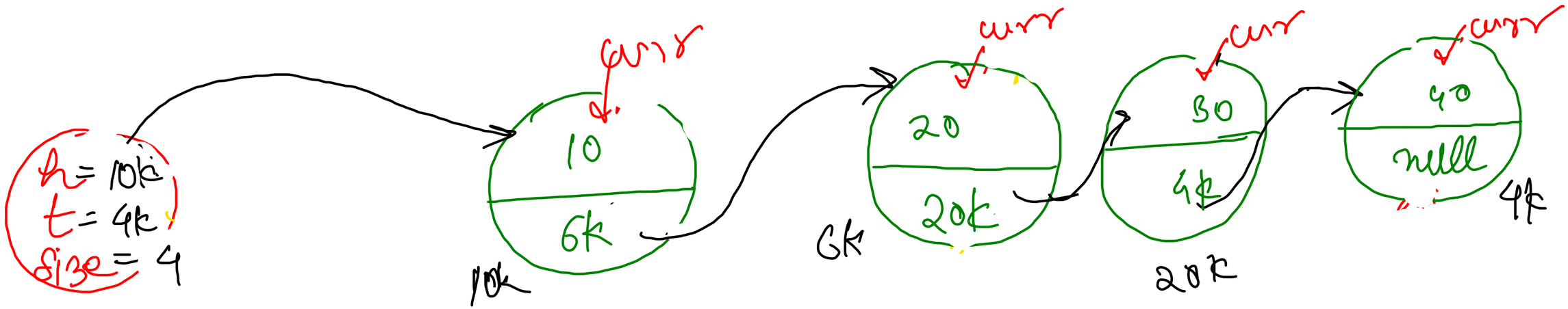
10, 20, 30, 40

1st 2nd 3rd 4th

OR

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

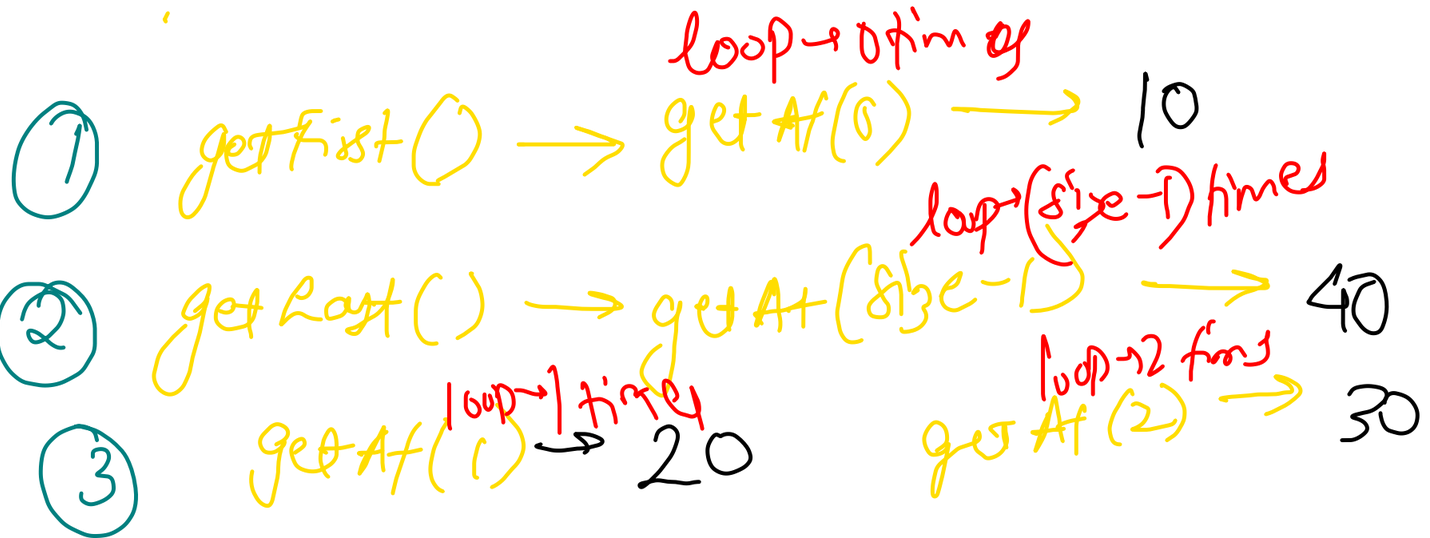
```
public void display(){
    if(size == 0){
        return;
    }

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

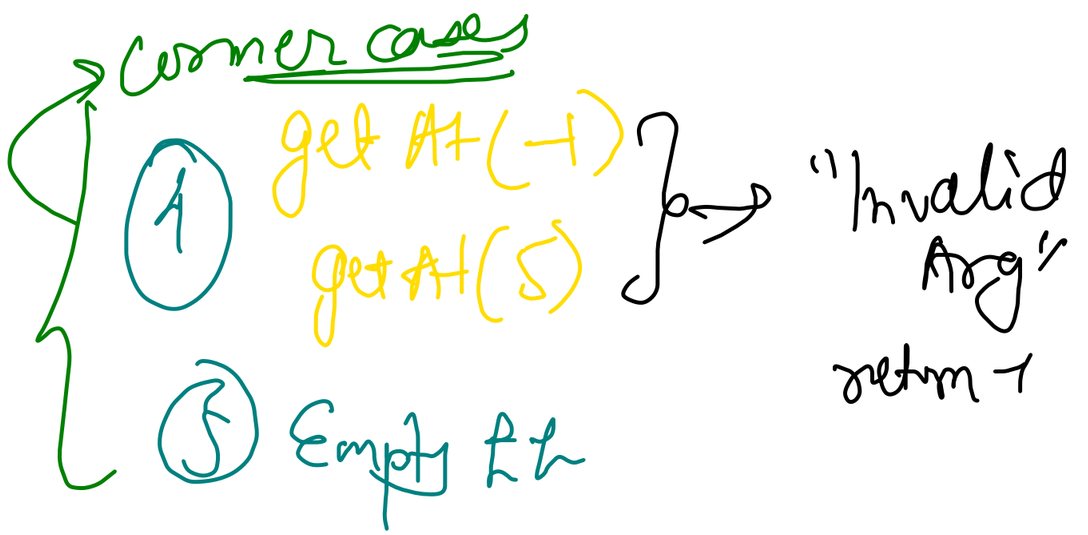


3.1. getFirst - Should return the data of first element. If empty should return -1 and print "List is empty".

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

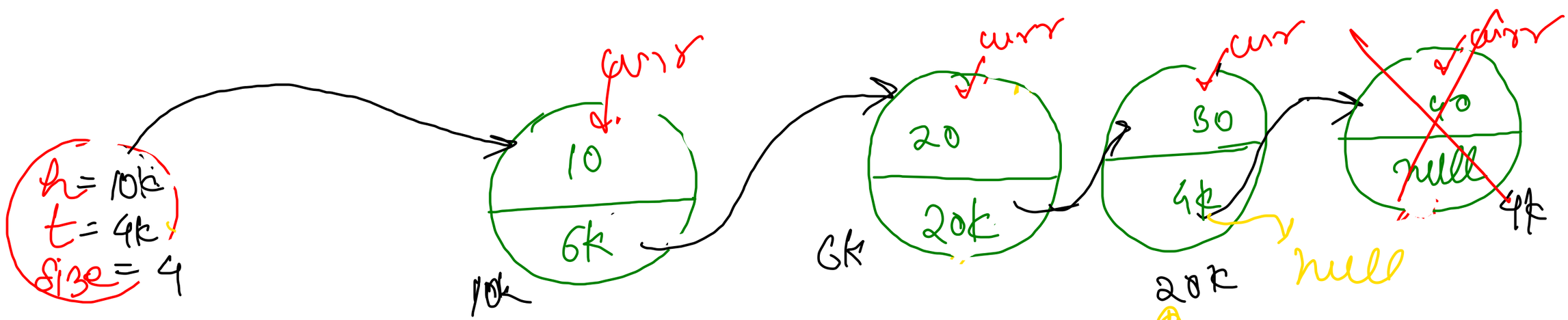
3.3. getAt - Should return the data of element available at the index passed. If empty should return -1 and print "List is empty". If invalid index is passed, should return -1 and print "Invalid arguments".





```java
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)



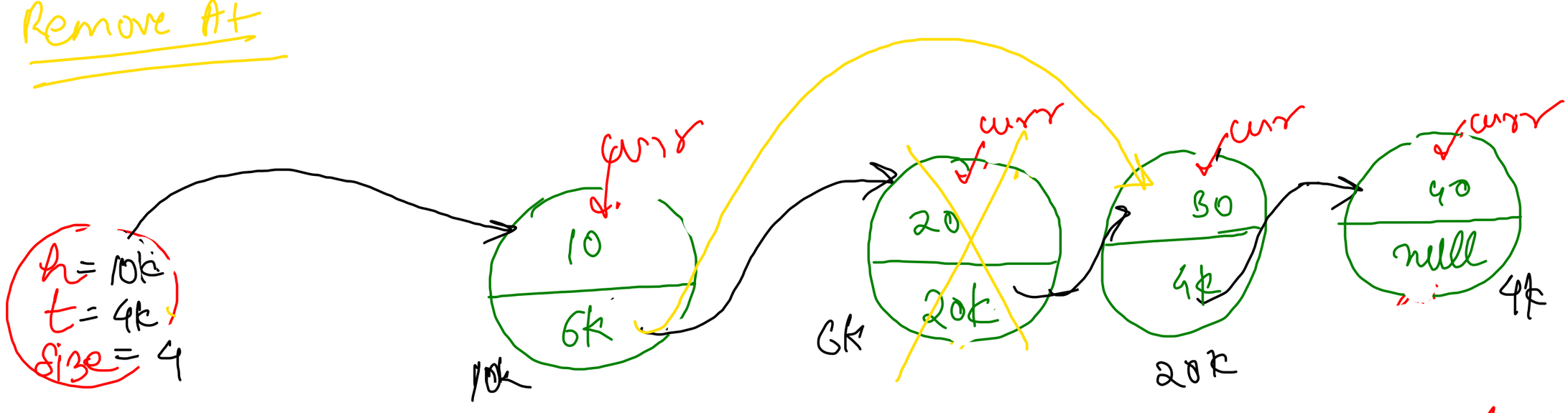
Remove last

```
if(size>1){
    Node prevTail = getAt(size-2);
    prevTail.next = null;
    tail = prevTail;
}
```

```java
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--;
}
```

→ O(n)

Remove At



① removeAt(0) ⟶ removeFirst()

② removeAt(size-1) ⟶ removeLast()

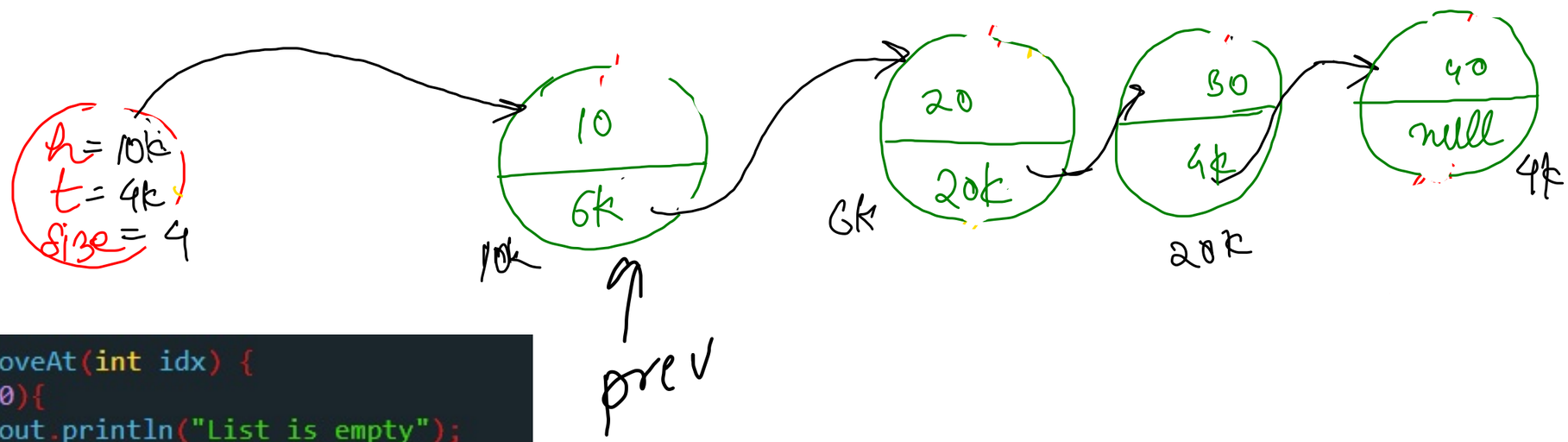
③ removeAt(idx) ⟶

{ 0 < idx < size-1 }

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

idx < -1 } invalid arg
idx >= size

size == 0 } List is empty



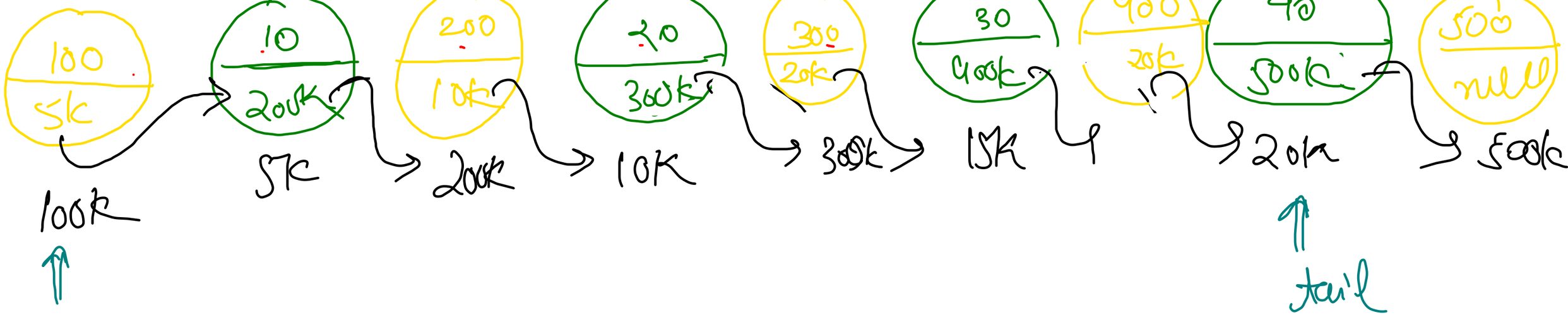

```java
public void removeAt(int idx) {
    if(size == 0){
        System.out.println("List is empty");
        return;
    }

    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--;
    }
}
```

O(N) → In worst case

LL.removeAt(2)

Worst O(N)

Add At



head

AddAt(0, 100) ⇒ addFirst(100);

AddAt(2, 200)

AddAt(4, 300)

AddAt(6, 40) ⇒ Add at Second last

size-1

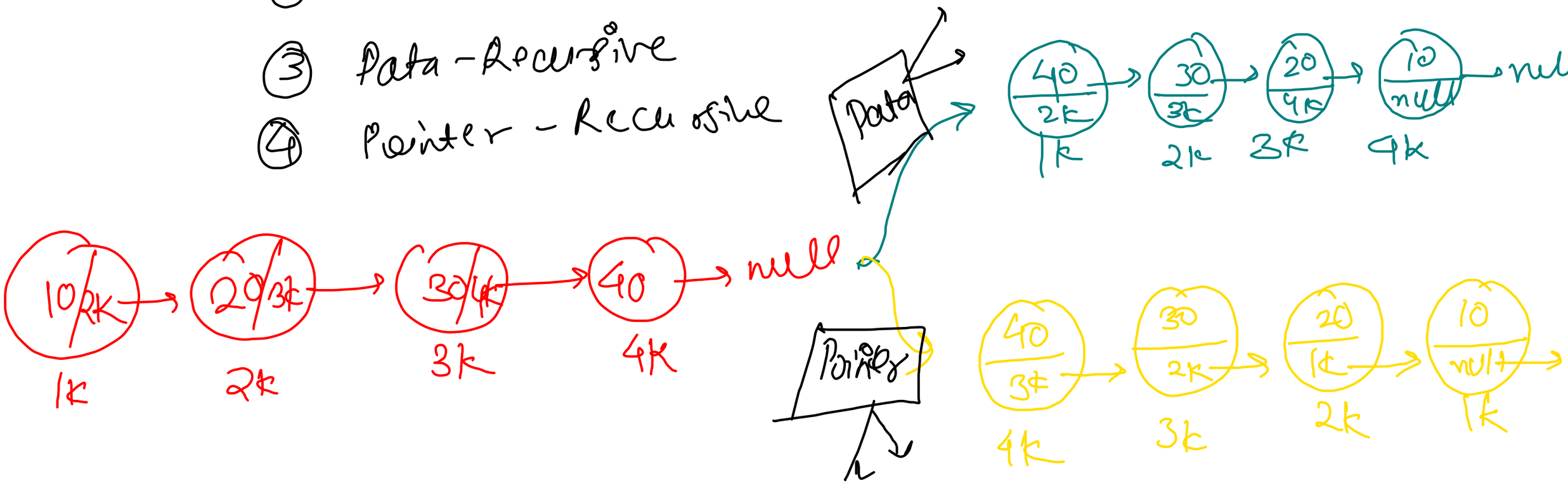
AddAt(8, 500) ⇒ AddLast

size

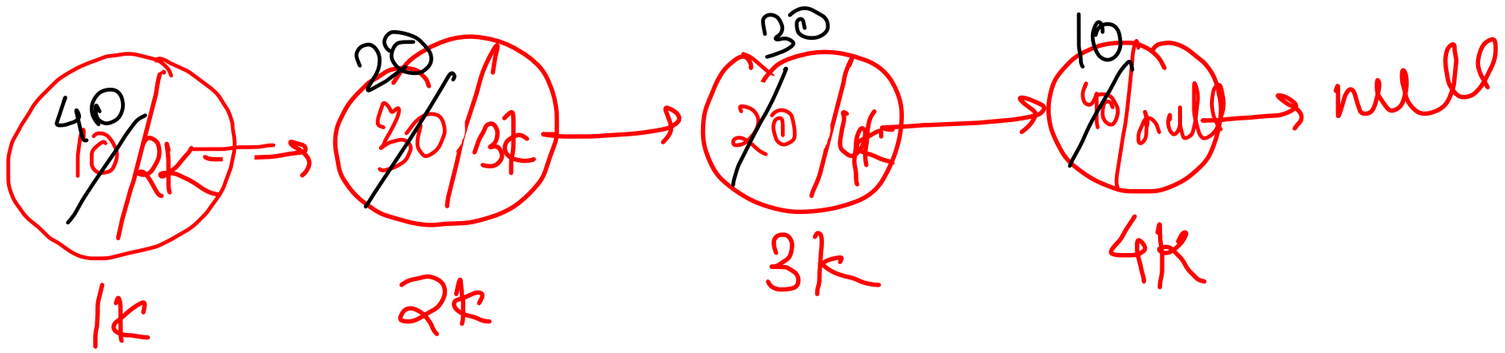
```java
public void addAt(int idx, int val){
    if(idx < 0 || idx > size){
        System.out.println("Invalid arguments");
        return;
    }

    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

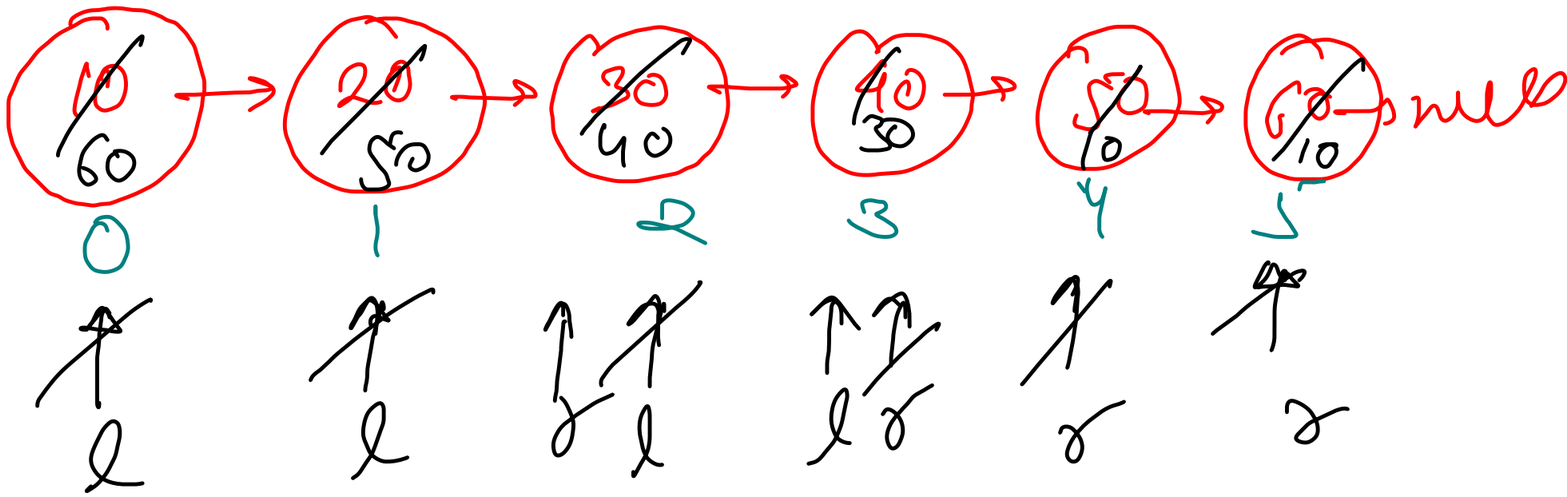


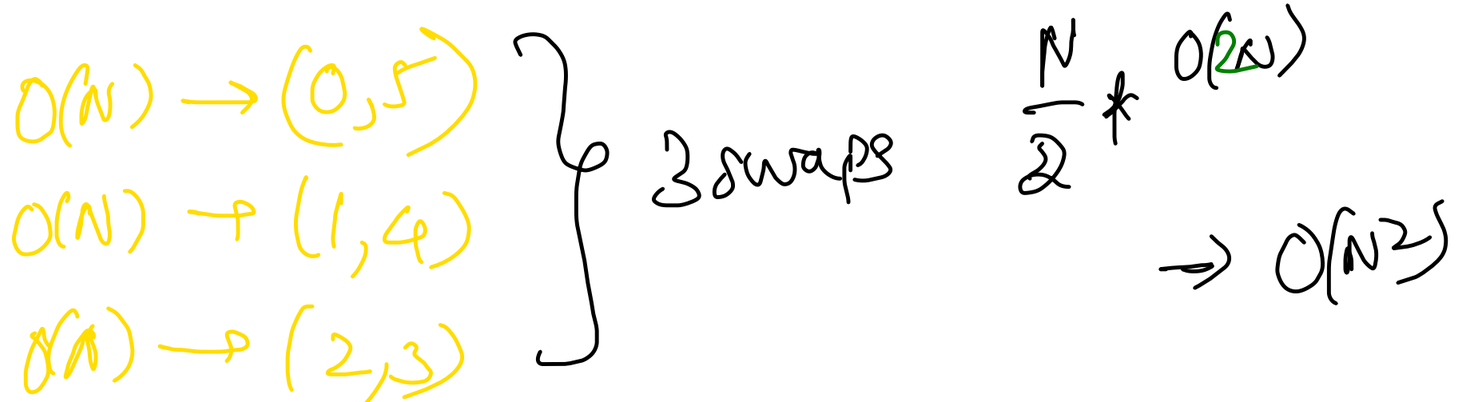
right left

```
while(left < right){
(1) leftNode = getNAt(0)
    rightNode = getNAt(n-1)
    ⇒ O(N)
    left++ right--

(2) left-N = getAt(1)
    rightN = getAt(n-2)
    ⇒ O(N)
}
```

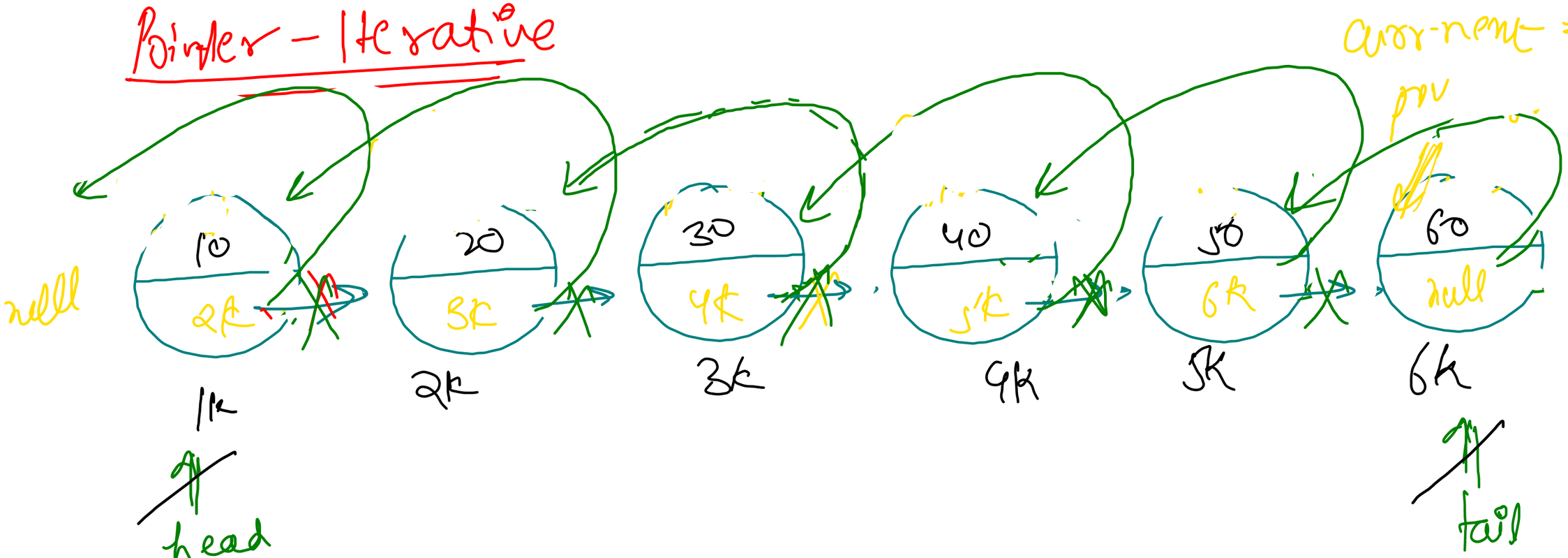
Time Comp ⇒ $\frac{N}{2} * O(N) = O(N^2)$

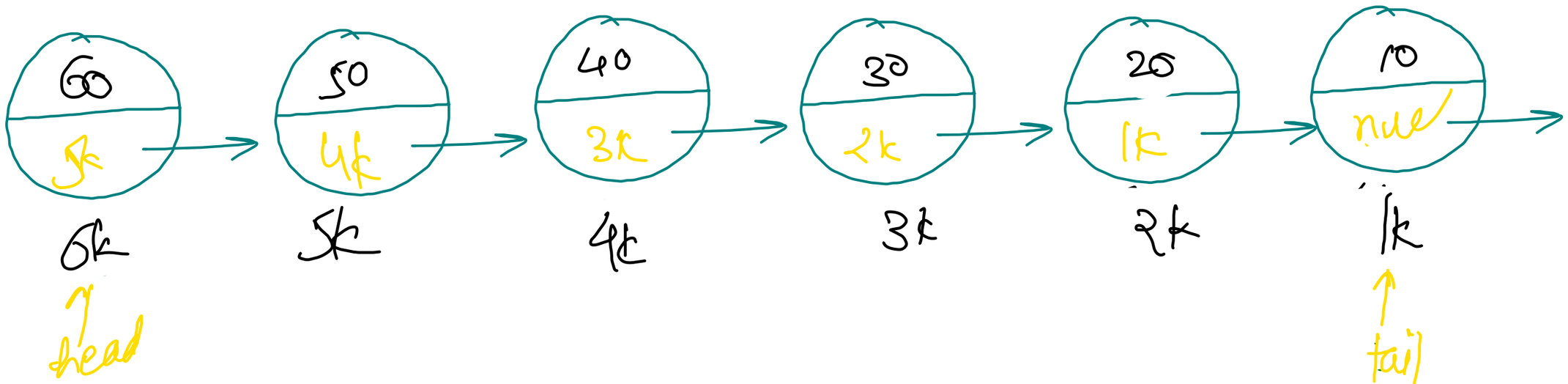
```java
public Node getNodeAt(int idx){
    Node curr = head;
    for(int i=0; i<idx; i++){
        curr = curr.next;
    }
    return curr;
}

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--;
    }
}
```





Pointer - Iterative

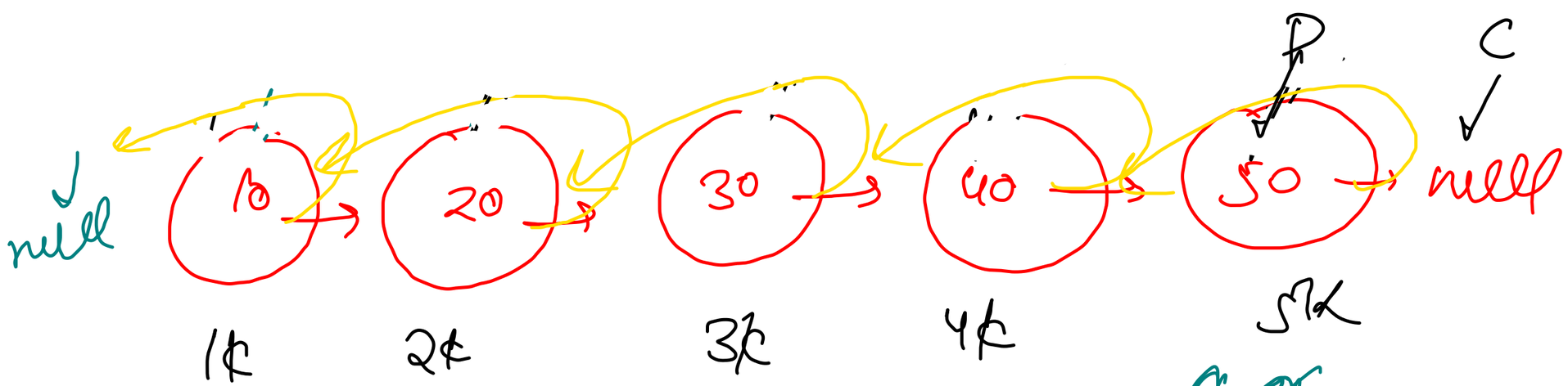




```
Node prev = null, curr = head;
while (curr != null) {
    Node ahead = curr.next;
    curr.next = prev;
    prev = curr; curr = ahead;
}
```

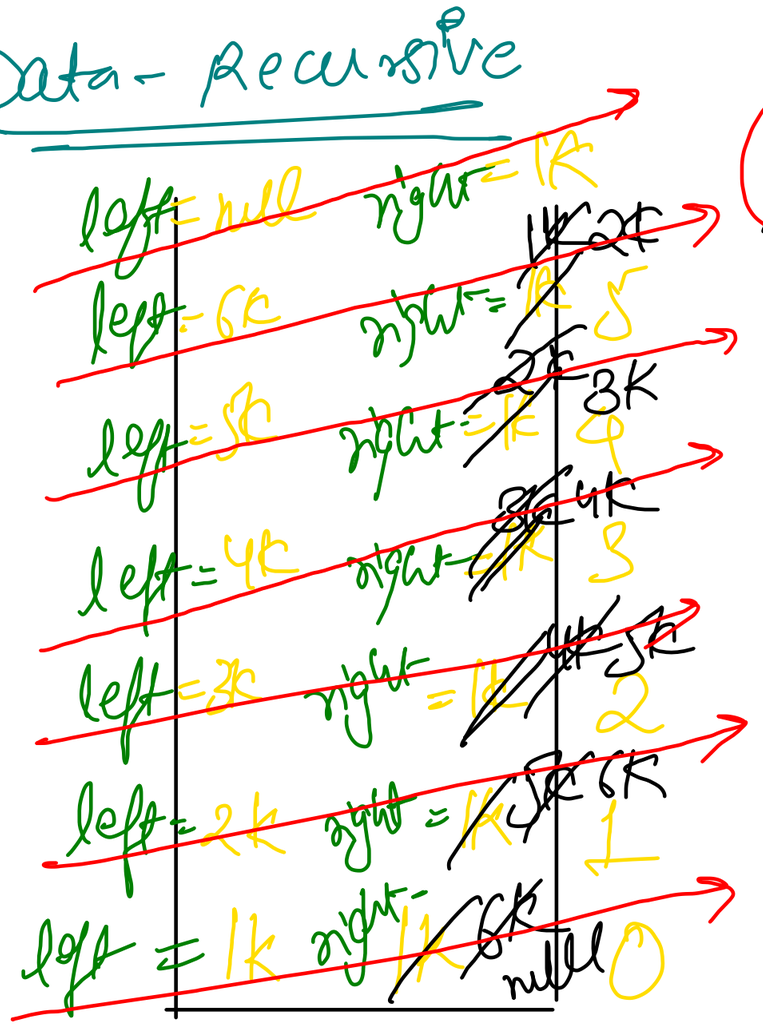
swap (head, tail)

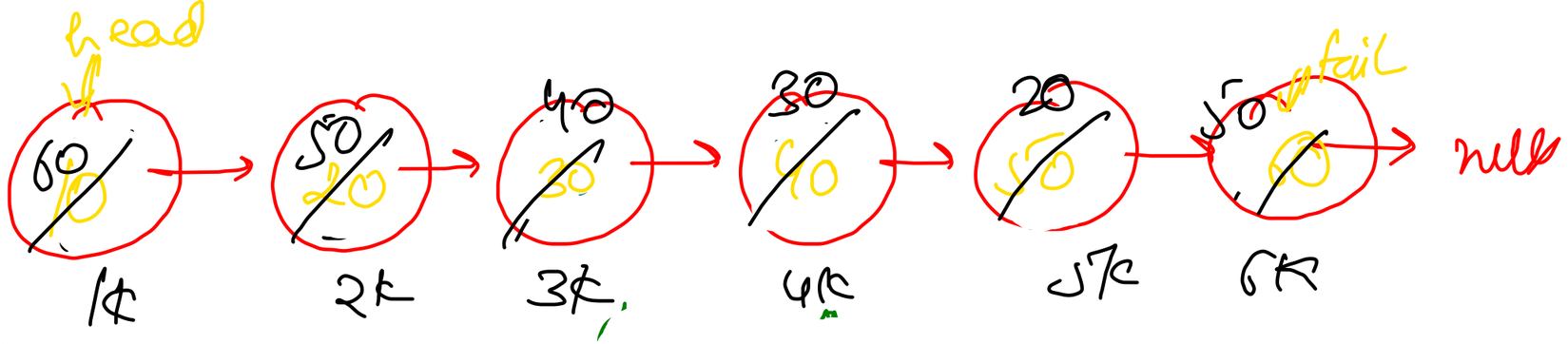
1. Even
2. Odd
3. 1 Node
4. 0 Node




```java
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;
}
```

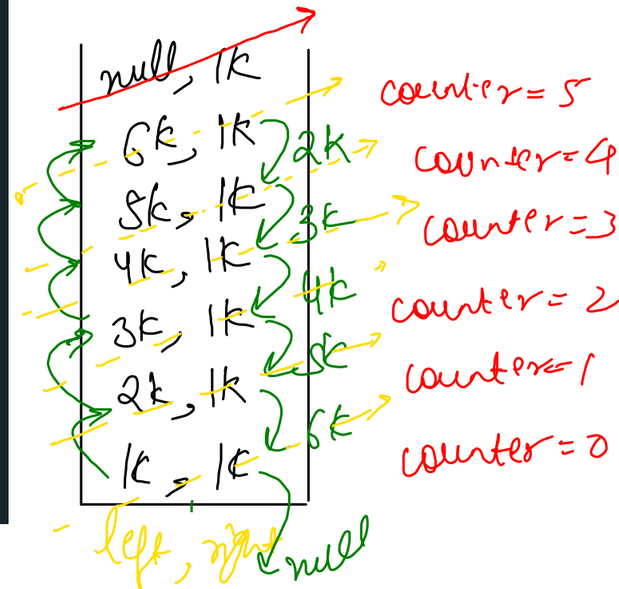
Data - Recursive



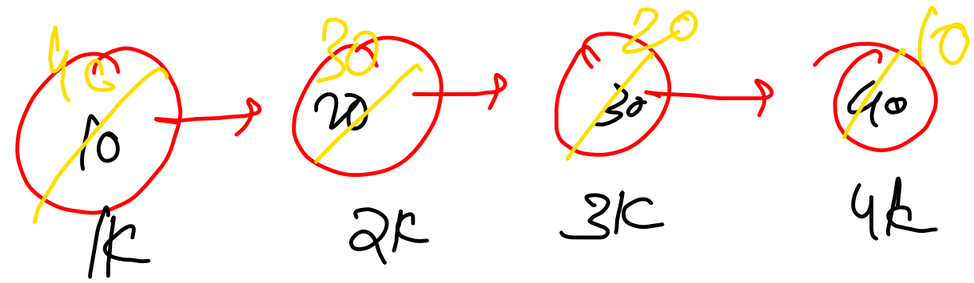


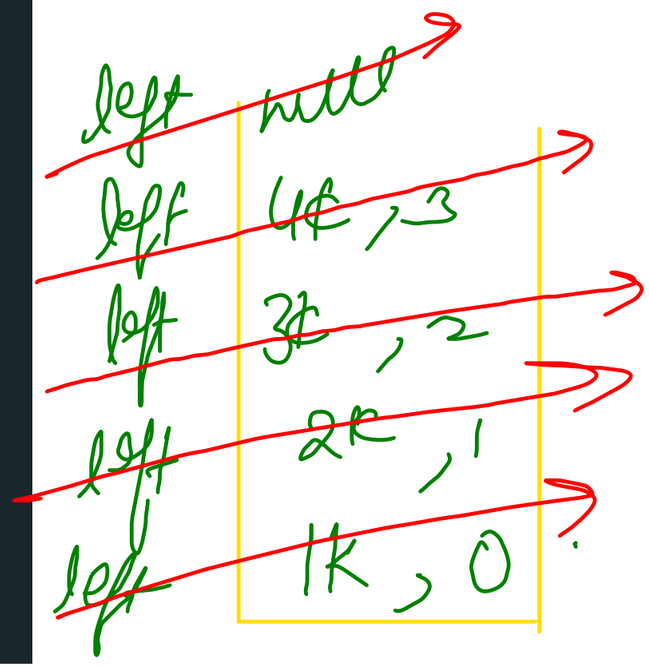
$$(1)^n + (k+\phi) * n = \boxed{O(n)}$$

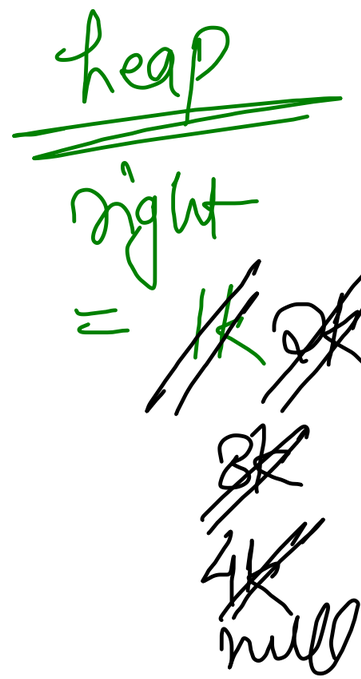
```java
public Node reverseDR(Node left, Node right, int counter){
    if(left == null){
        return right;
    }

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

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

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



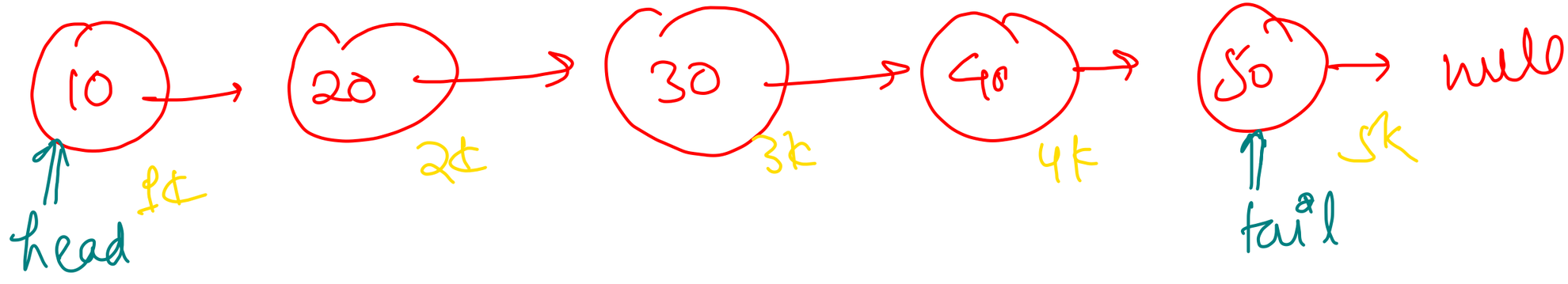
```java
static Node right;
public void reverseDR(Node left, int counter){
    if(left == null){
        return;
    }

    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);
}
```

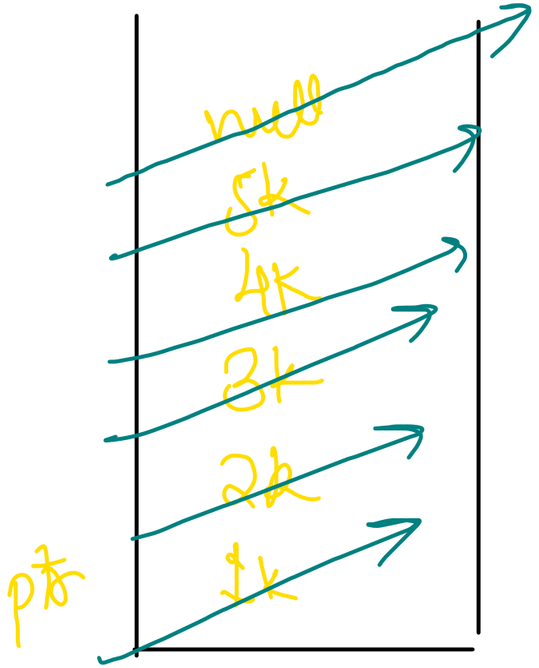






Display Reverse { Recursion }



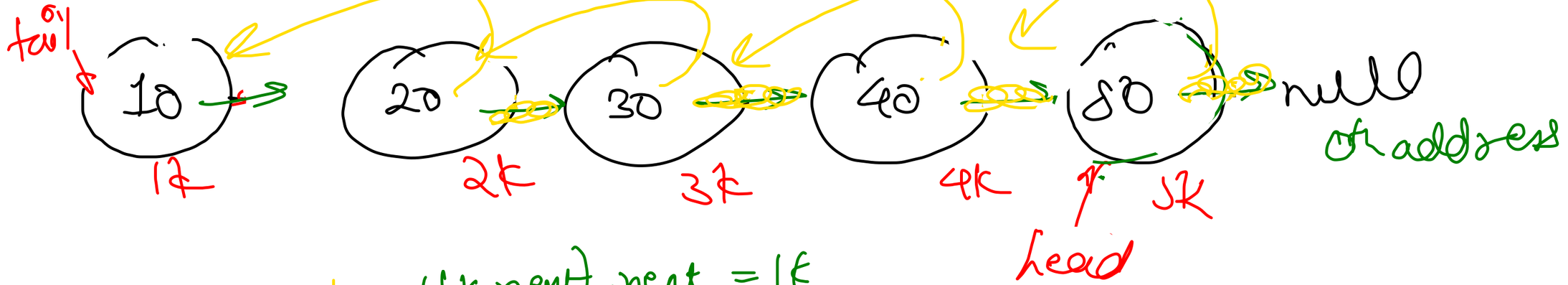


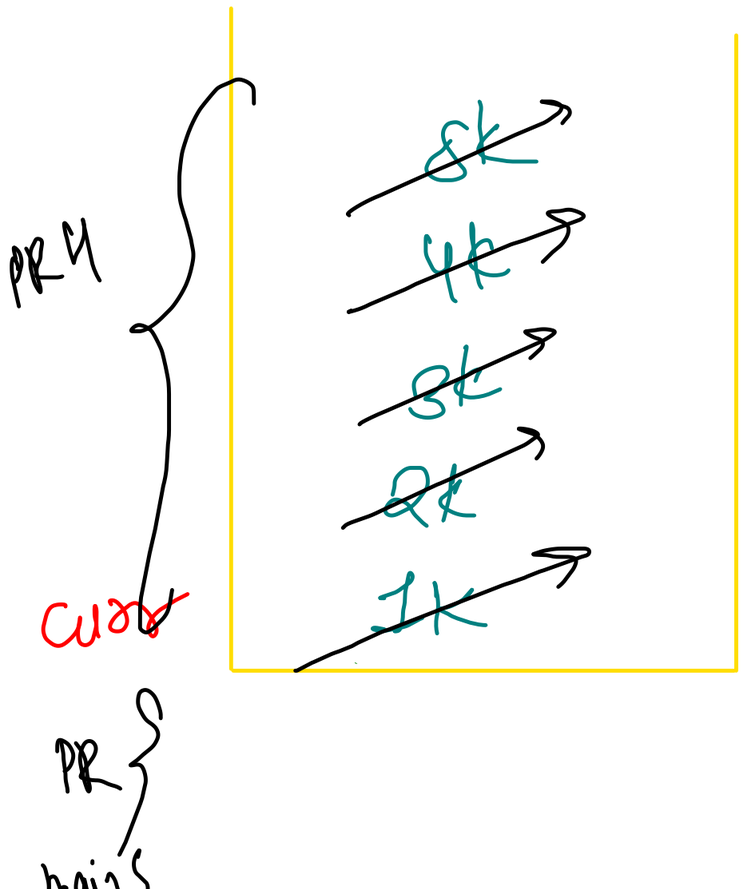
50 , 40 , 30 , 20 , 10

```java
private void displayReverseHelper(Node node){
    if(node == null) return;

    displayReverseHelper(node.next);
    System.out.print(node.data + " ");
}
```

O(N)

Pointer Recursive → Reverse th





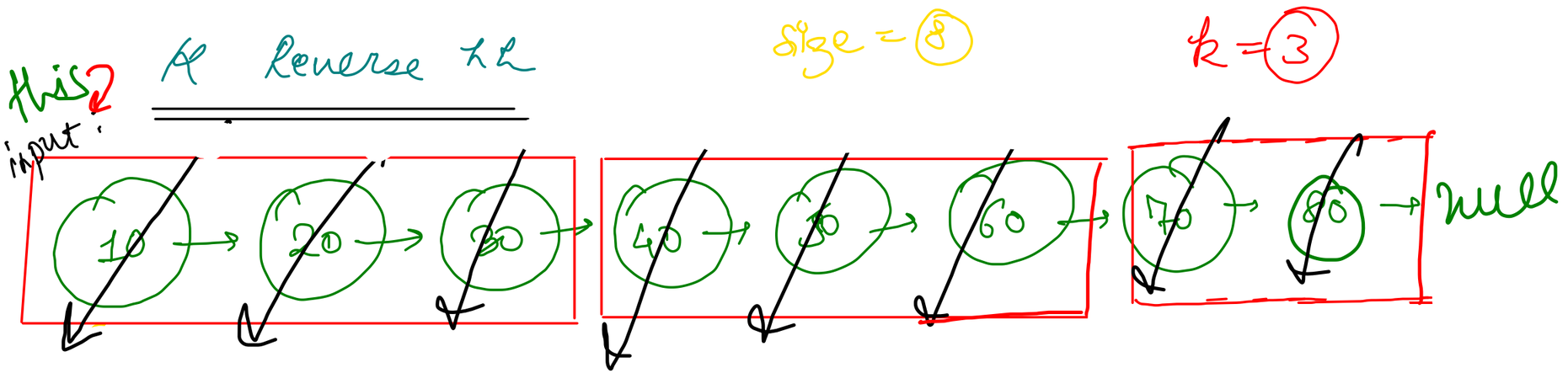
(1K).next = 1K

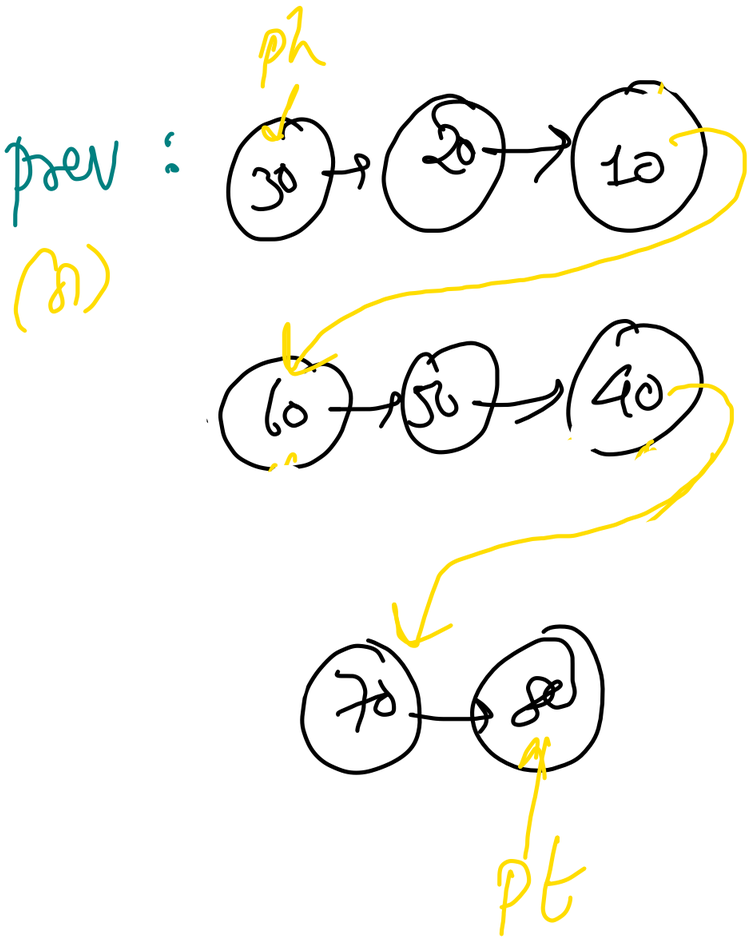
2K.next = //

```
node.next.next = node;
```

```
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;
}
```



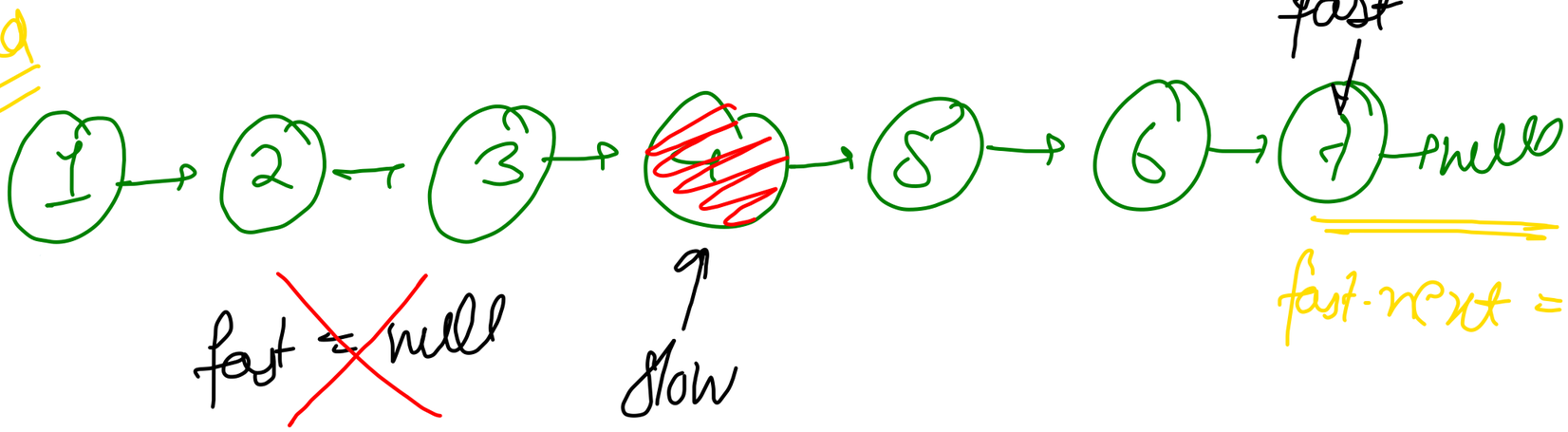



```
LinkedList prev = new LinkedList();

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;
```

876 LC

Middle Linked List { Hare & Tortoise } { Two pointer }

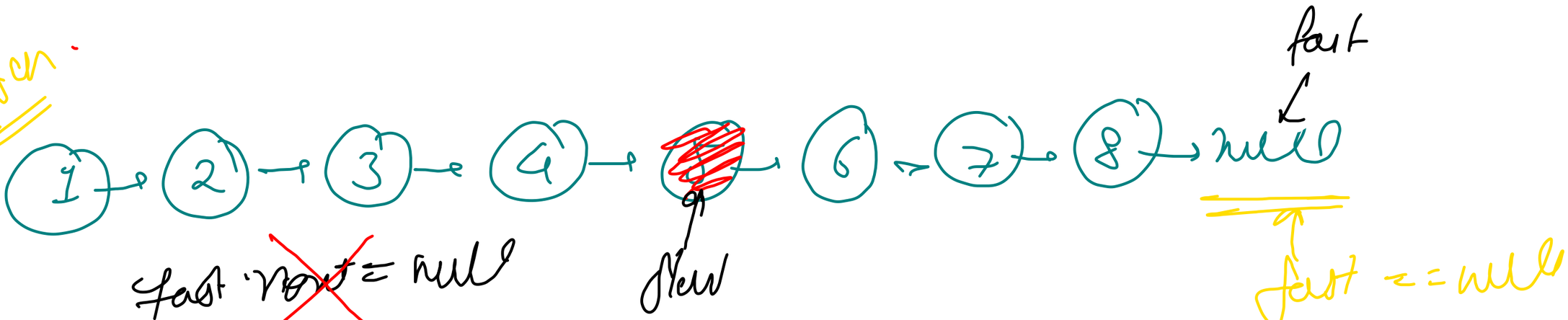
odd



① Two traversal

get (size/2) + 1

even



O(N)

```java
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;
}
```

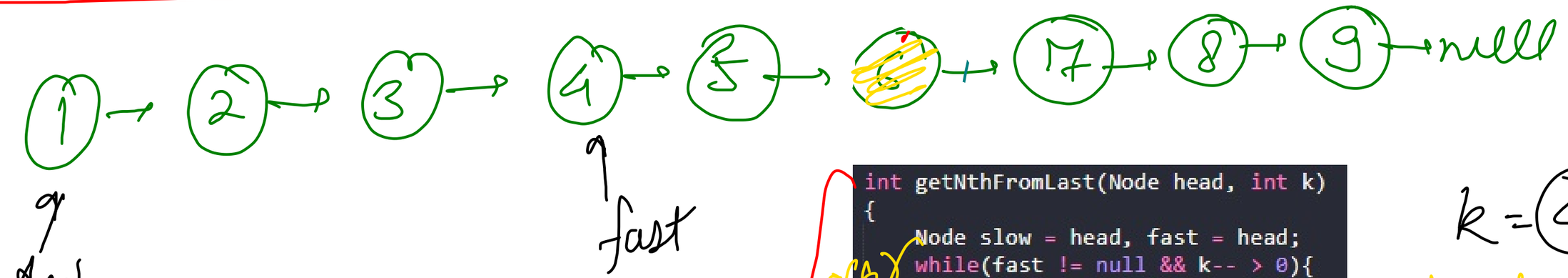
O(N)

```java
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)

```java
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;
}
```

Kth Node from End




```
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;
}
```

O(N)

O(K)

O(N-K)

corner case
k > N

k = 4

total = 9 = N

1st Appr : 2 Traversal

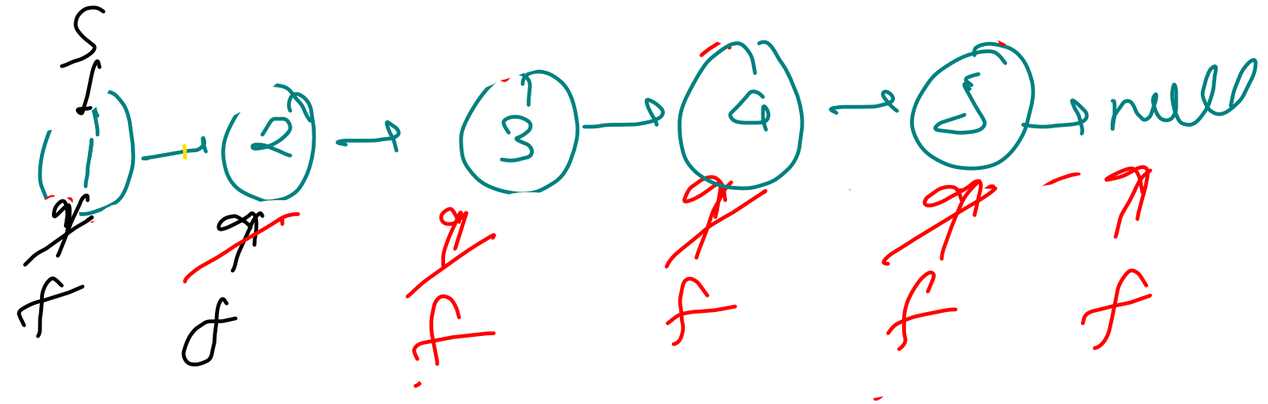
→ count No of nodes in LL ⇒ N

→ get(N - k)
9 - 4

2nd Appr : Single traversal
{ Two pointer }

```java
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 diff



k = 5 4 3 2 1 0 ⇒ 1

Both fast became null & k = 0 together

k = 1 0 ⇒ 5 → k became 0

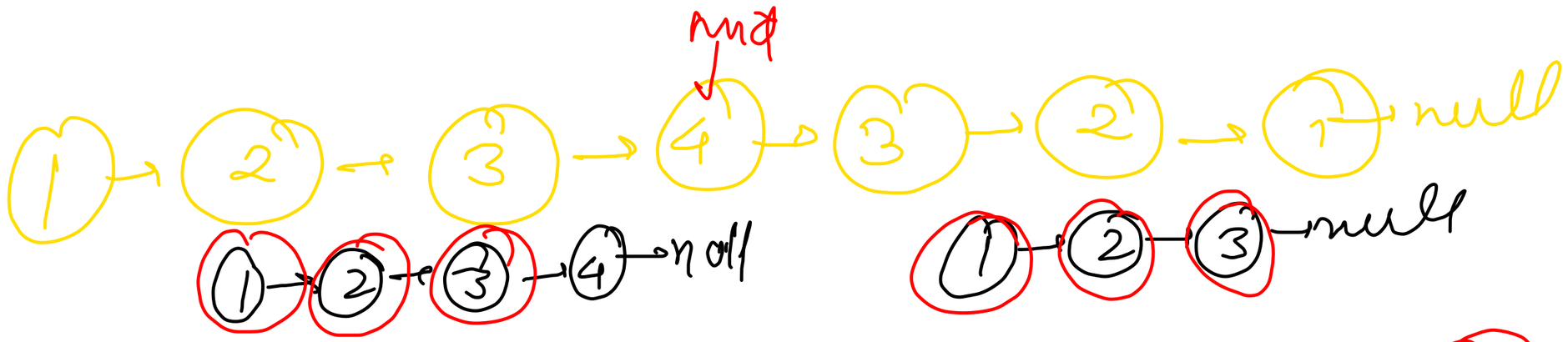
k = 6 5 4 3 2 1 ⇒ -1

⇒ only fast became null

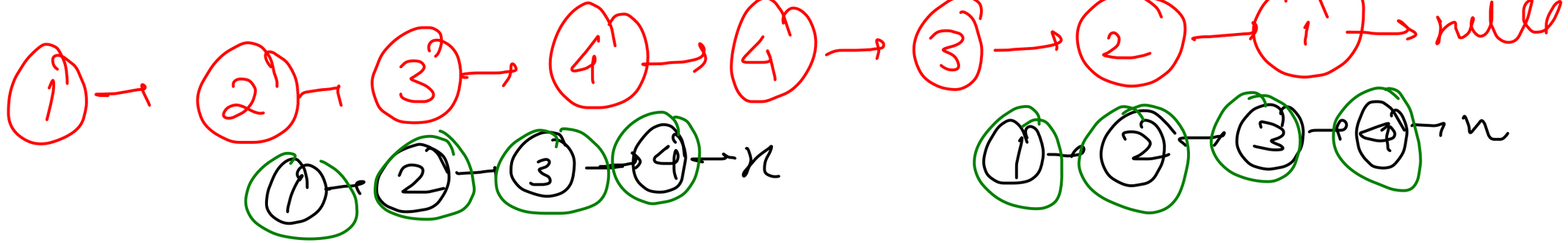
Palindrome LL

HINT → Reverse LL
→ Middle of LL

Odd:



Even



Approach

(1) Find Middle node

(2) Reverse the 2nd part

(3) Compare both the parts
- ↳ equal → Palindrome (T)
- ↳ Palindrome (F)

O(N)

```java
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)

```java
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;
}
```

even → first mid
Odd → single mid

O(N)

```java
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;
}
```

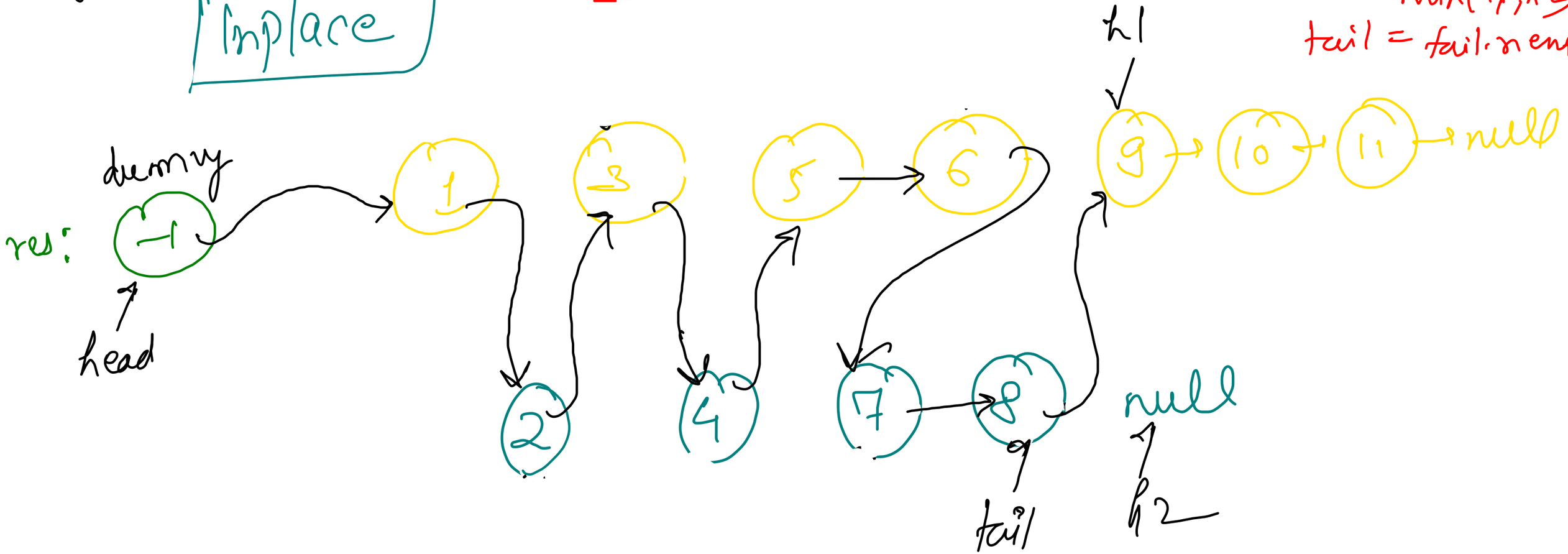
Corner Case

Merge 2 sorted lls

Inplace

O(N+M)

tail.next = min(h1, h2)
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 kth

→ merge 2 Sorted (LC) 21

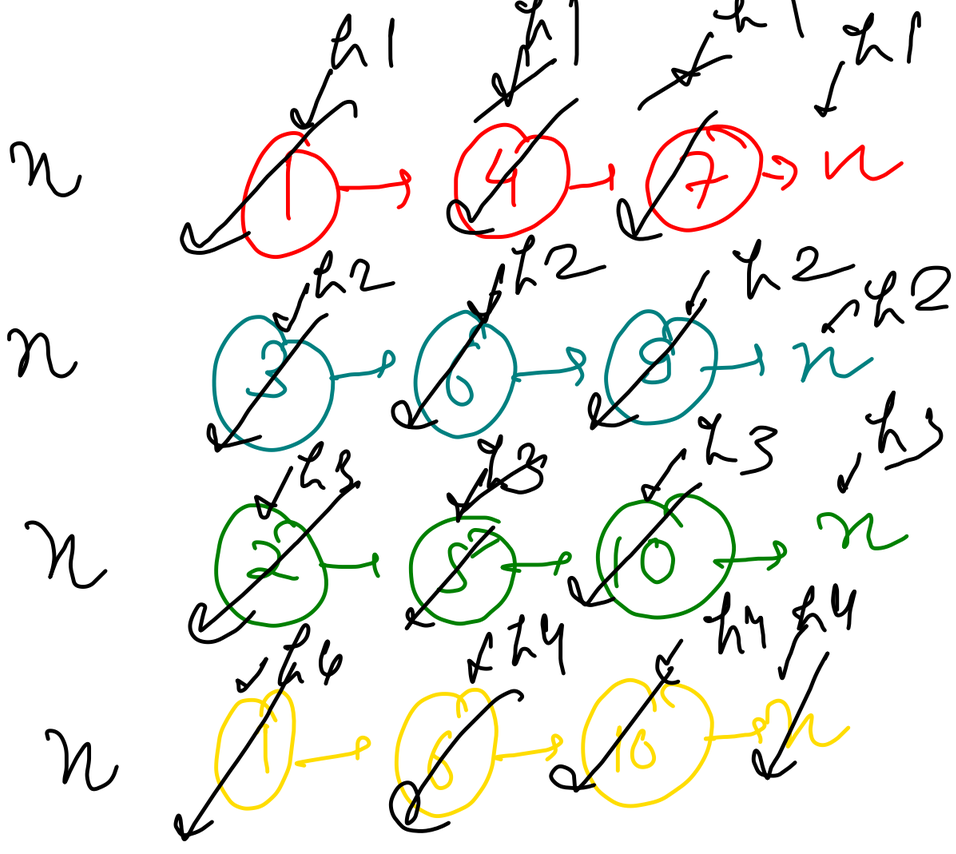
→ merge K Sorted LC HARD 23

→ merge Sort (LC) 148

→ Remove Duplicates (LC) 83 & 82
(1&11)

→ Fold & Unfold kth (LC) 143 & Pepcoding

Merge K sorted LL {HARD}

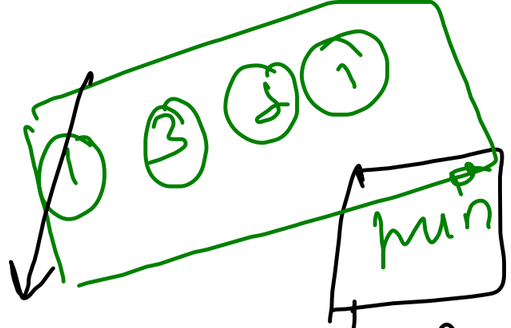


resultant node of ll

$N = n * k$ ← number of LL

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

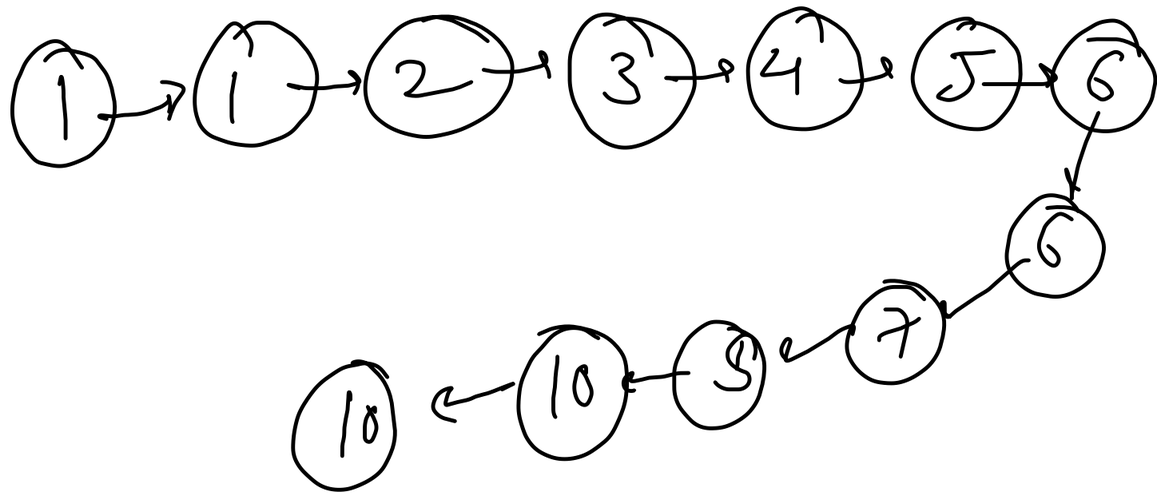
node in 1 LL



⇒ k elements

⇔ O(k) time

in each iteration



Brute force

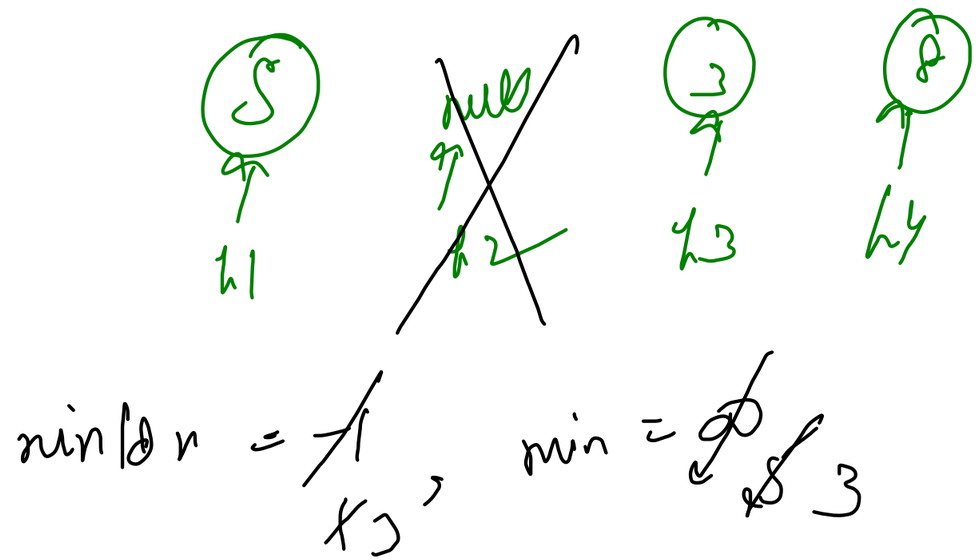
$O(n * k^2)$

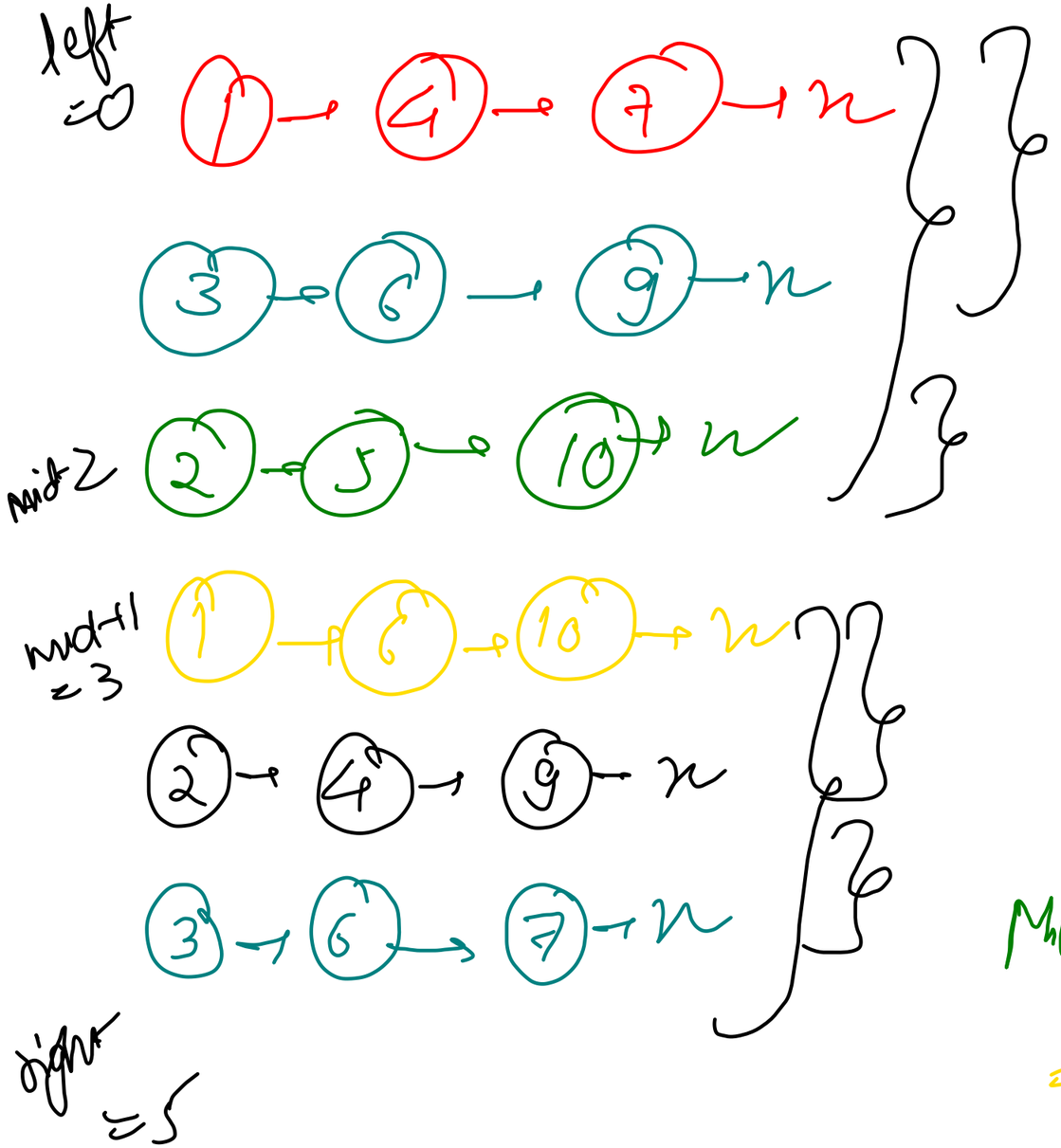
or $O(N * k)$

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

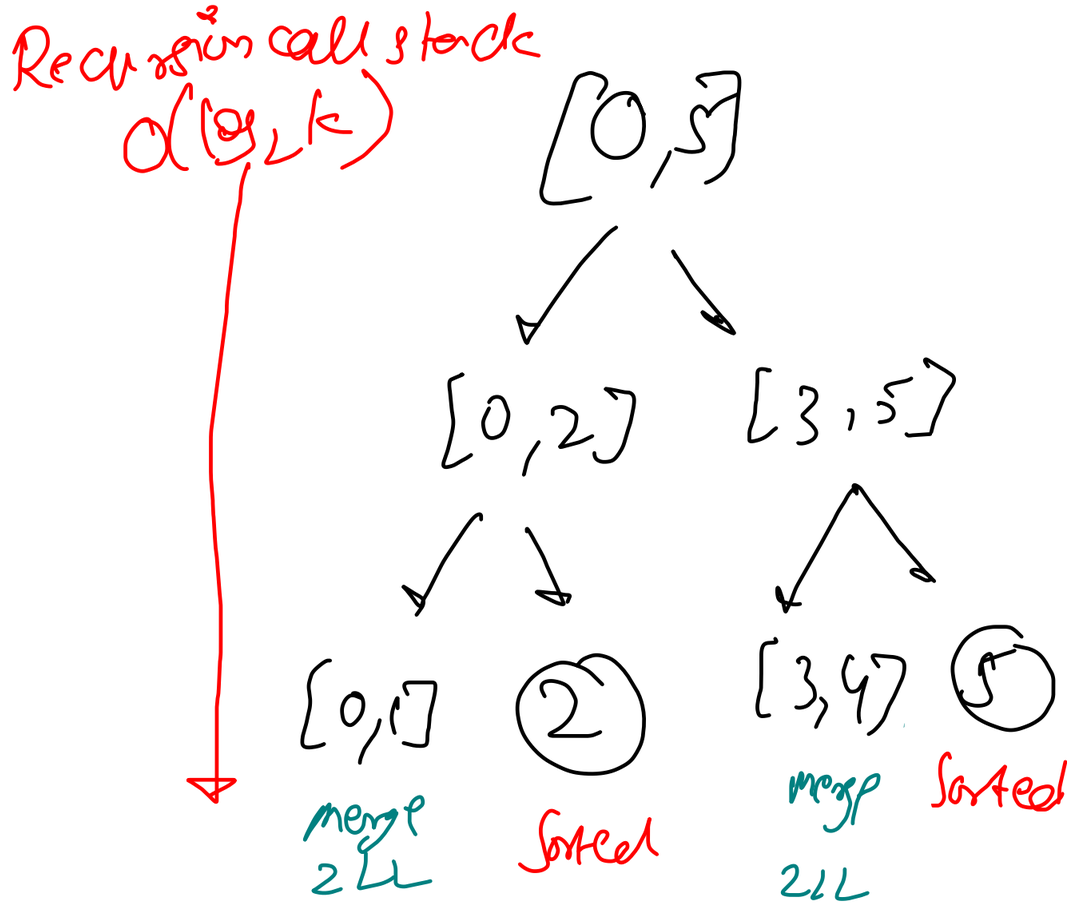
```java
public ListNode mergeKLists(ListNode[] lists) {
    if(lists.length == 0) {
        // 0 linked lists
        return null;
    }

    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;
}
```

```java
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;
}
```





10:10



Merge K Sorted (l, r)

= Merge K Sorted (l, m) + Merge K sorted (m+1, r)

+ Merge 2 LL

```java
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

$T(k) = k*T(0) + k*n*\log_2 k$

no of terms

$O(k*n*\log_2 k)$

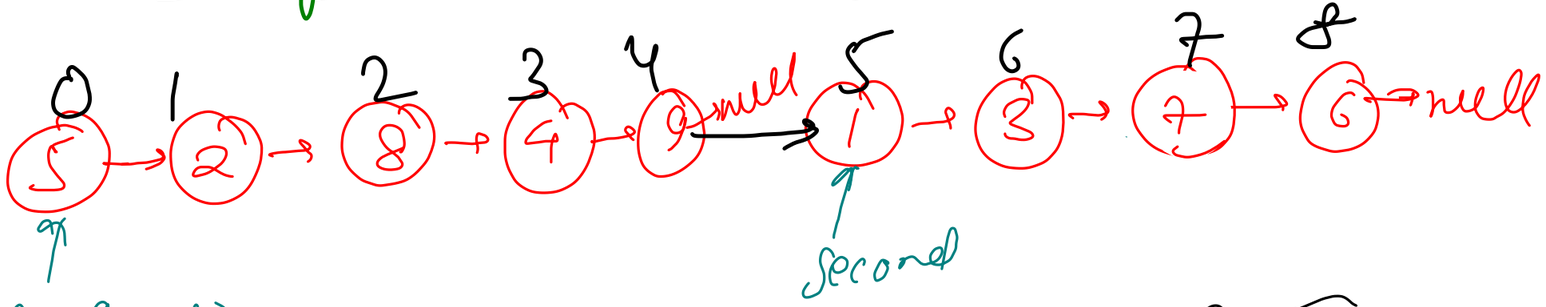
$T(k) = 2T(k/2) + \frac{k*n}{2^0}$

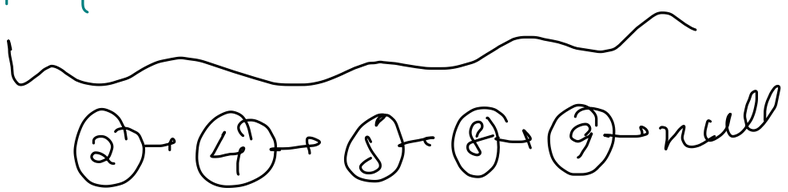
$T(k/2) = 2T(k/4) + \frac{k}{2^1}*n) * 2^1$

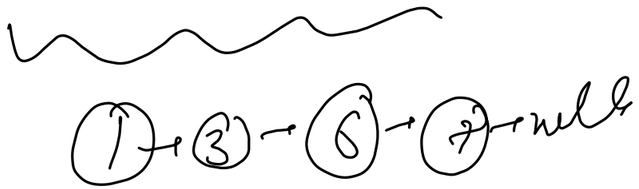
$T(k/4) = 2T(k/8) + \frac{k}{4}*n) * 2^2$

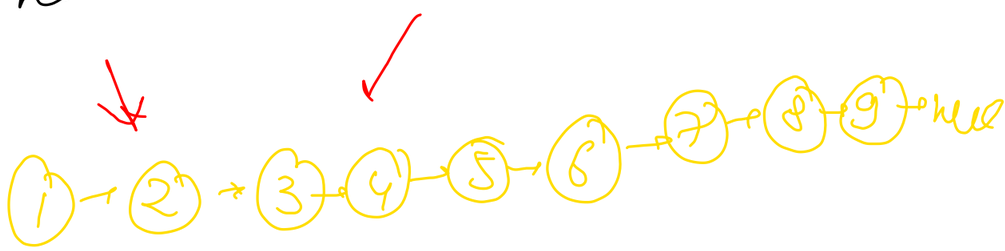
$T(1) = 2T(0) + \frac{k}{k}*n) * 2^{\log_2 k}$

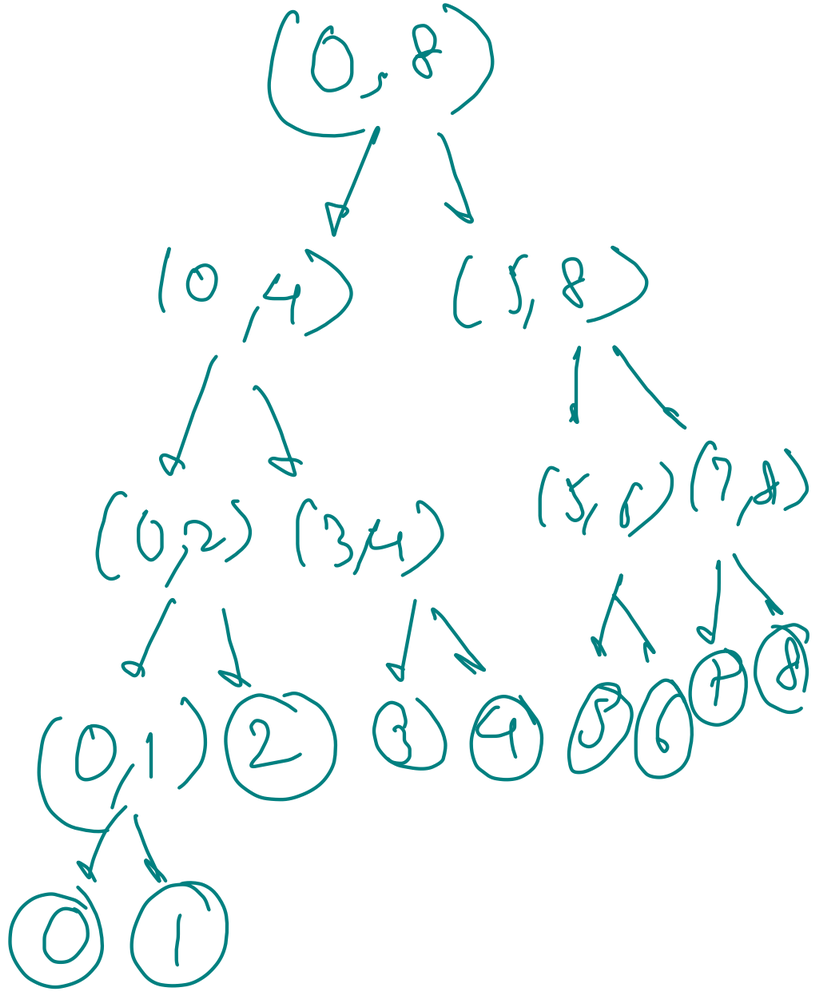
Merge Sort LL {14 LC}











Approach

MergeSort(LL) →

(1) Find middle & break LL into 2 parts

(2) mergeSort(left)
mergeSort(right)

(3) Merge & Return

```java
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
}
```

```java
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);
}
```

mergeSort → $T(n) = O(n) + 2T(n/2) + O(n)$

middle → $O(n)$

Recursive calls → $2T(n/2)$

merge → $O(n)$

$T(n) = 2T(n/2) + 2*O(n)$ ⋮ → $O(n \log_2 n)$

MS on LL = MS on Array (in Time Comp.)

Why merge sort is preferred for linked list
and quick sort is preferred for arrays.

TC: O(nlogn) SC: O(1)

merge Sort on LL

T(N) → O(nlogn)

Space → inplace

Recursion call
Stack ⇒ O($\log_2 n$)

merge 2 List
is inplace

→ Big-Integers

→ Add 2 th

→ Subtract 2 th

→ Multiply 2 th

int

$2^{31}-1$

$\approx 10^9$

Long

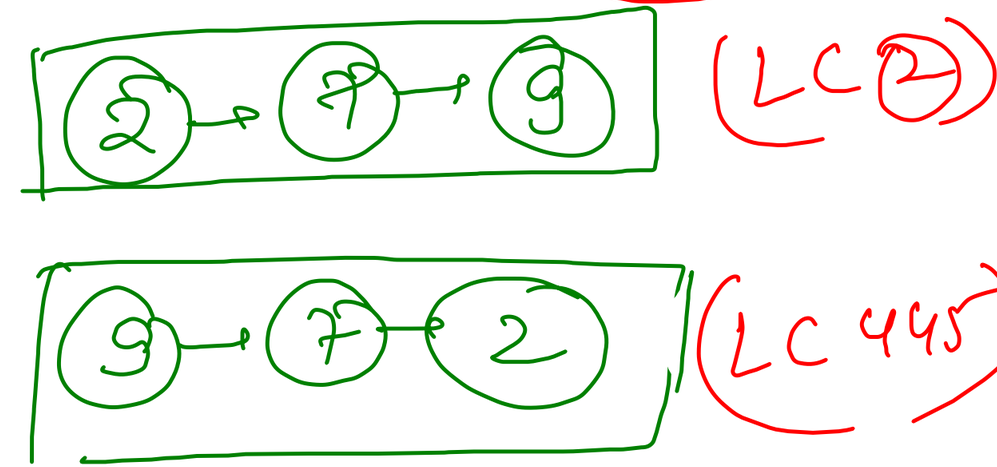
$2^{63}-1$

$\approx 10^{18}$

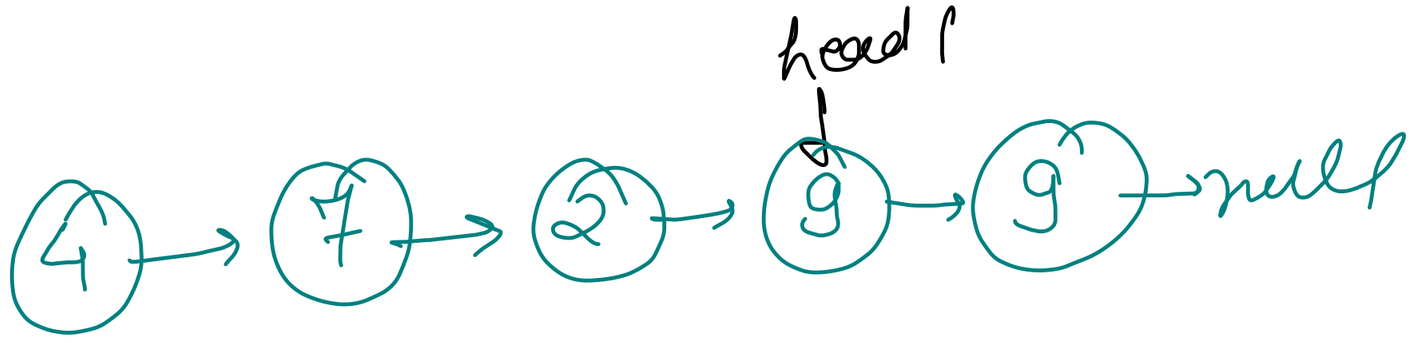
Addition of 2 LL

LC (2) LC (445)

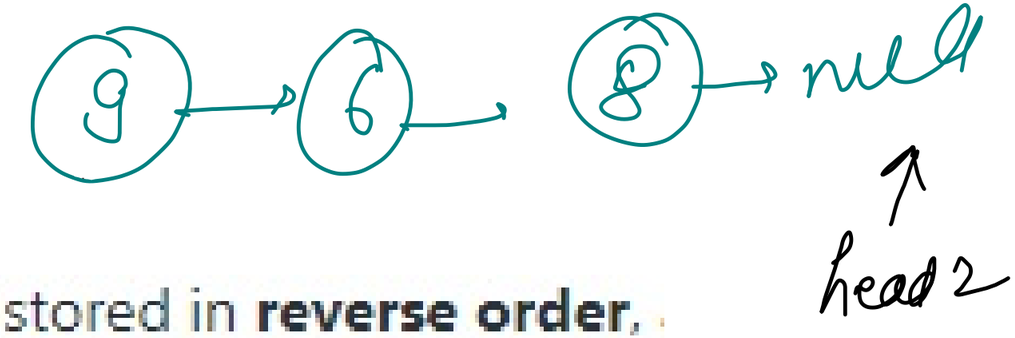
972



LC 2



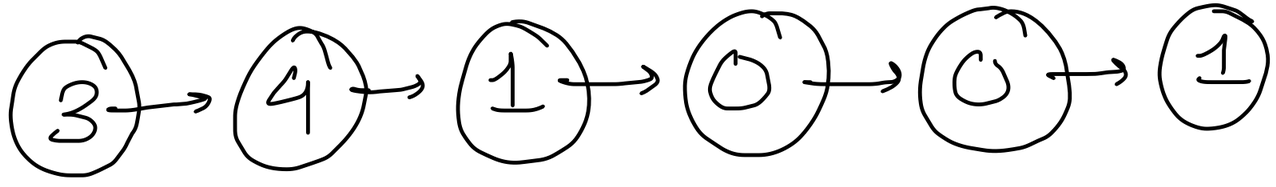
c = 0



99274
+
869
100143

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

ex:



(d1+d2+c)%10 ye
(d1+d2+c)/10 carry

```java
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

```java
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);
}
```

```java
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