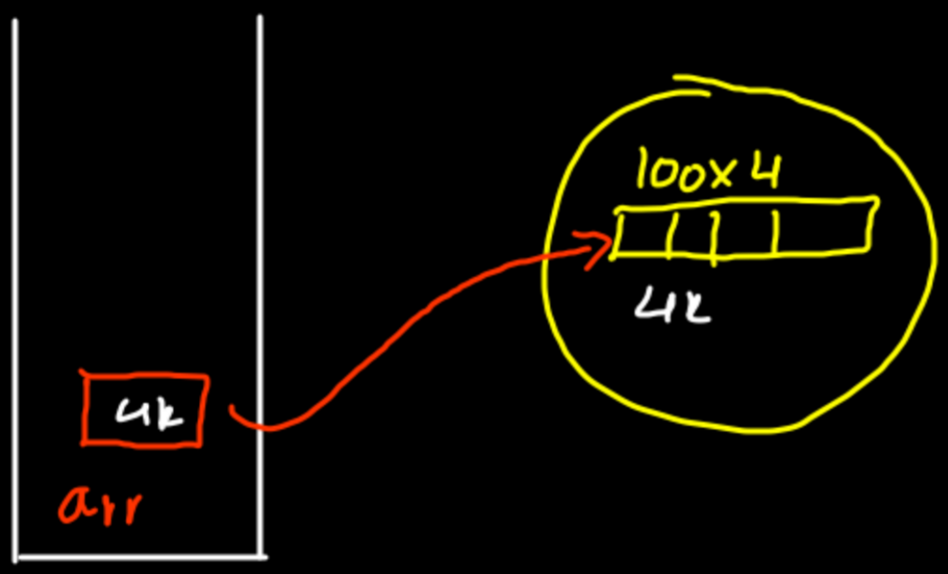
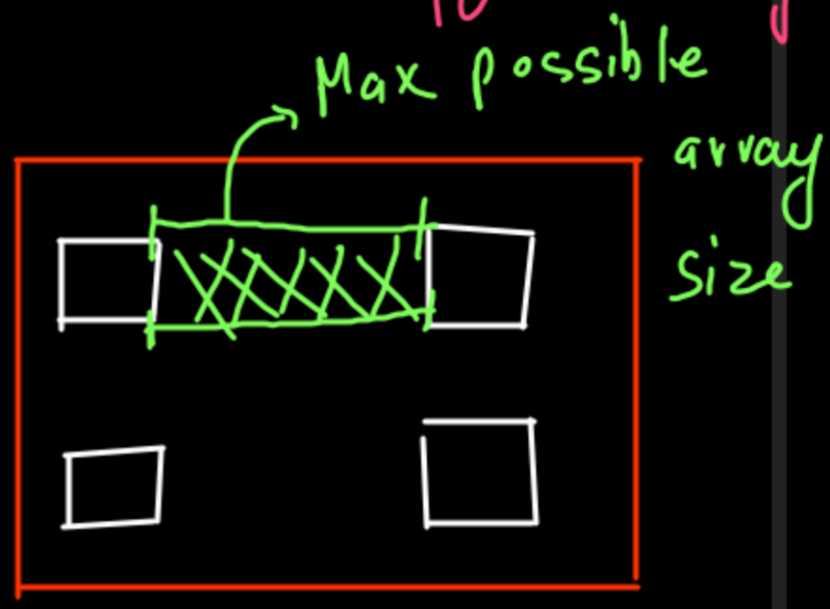
Linked List (Level 1+2)

Arrays

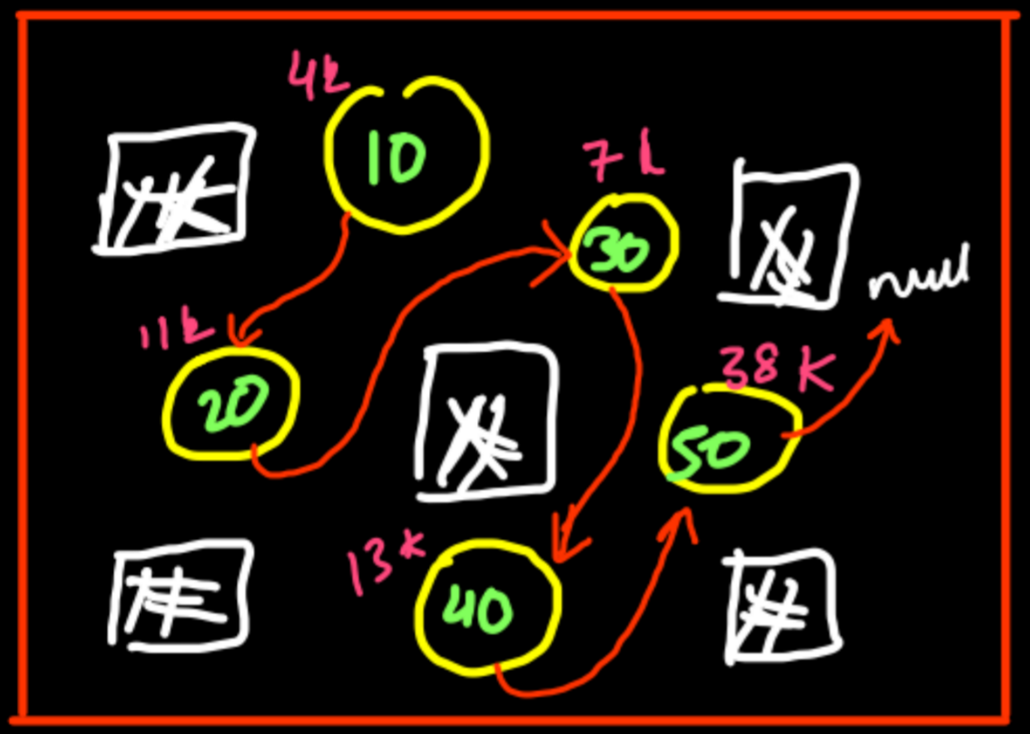
```
int [] arr = new int [100];
```





fragmentation issue (Memory got divided into fragments & enough Mem not avail)

Linked List: Non contiguous memory allocation.



We need links to connect data.

10, 20, 30, 40, 50

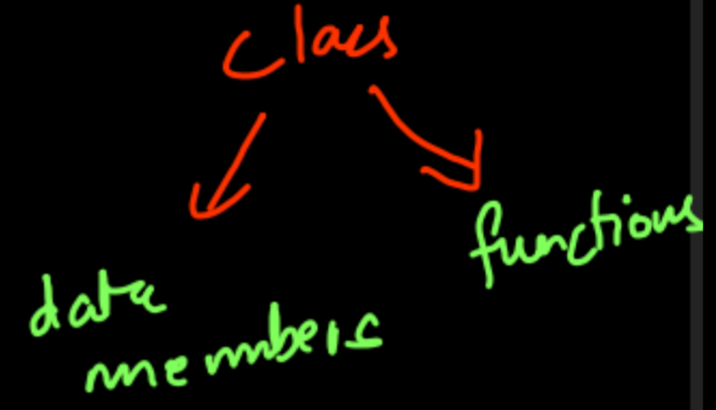
4k 11k 7k 13K 38k

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

```
Node{object}{
  int data;
  Node next;   → Address of Next Node
}
```

Inner class ko static banana hai.

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```



We only need one node (first Node) to get all the elements of the linked list because second ele will get from first's next & so on.

```java
public static class LinkedList {

    Node head;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```

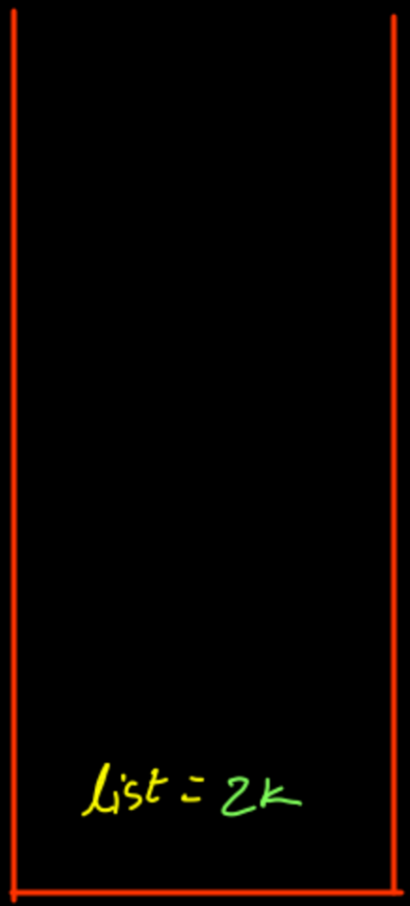
```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```

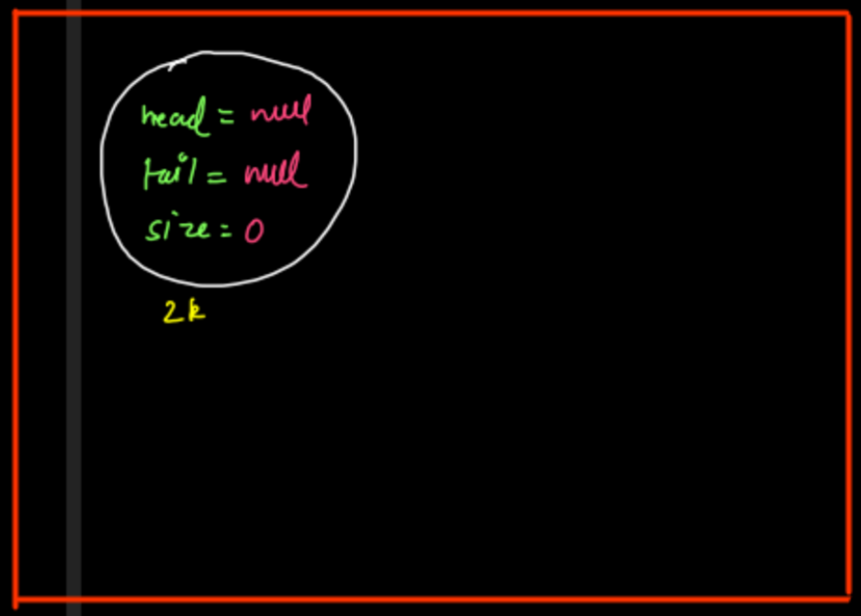
psv main() {

LinkedList list = new LinkedList();

}



Stack



Heap

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

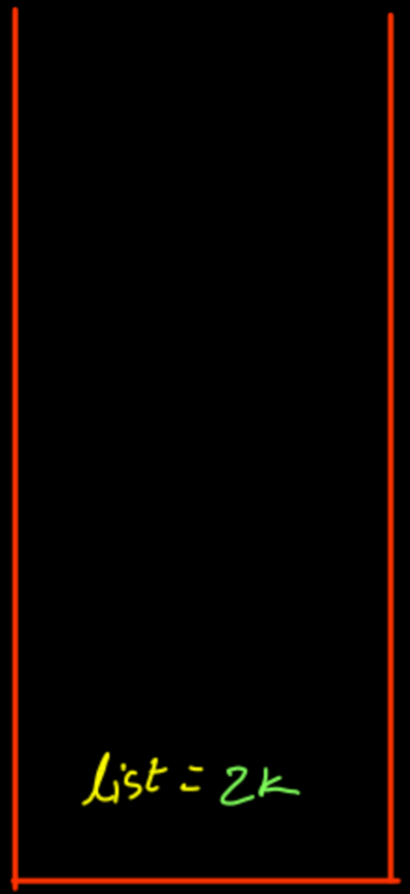
```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }
}
```

psv main() {

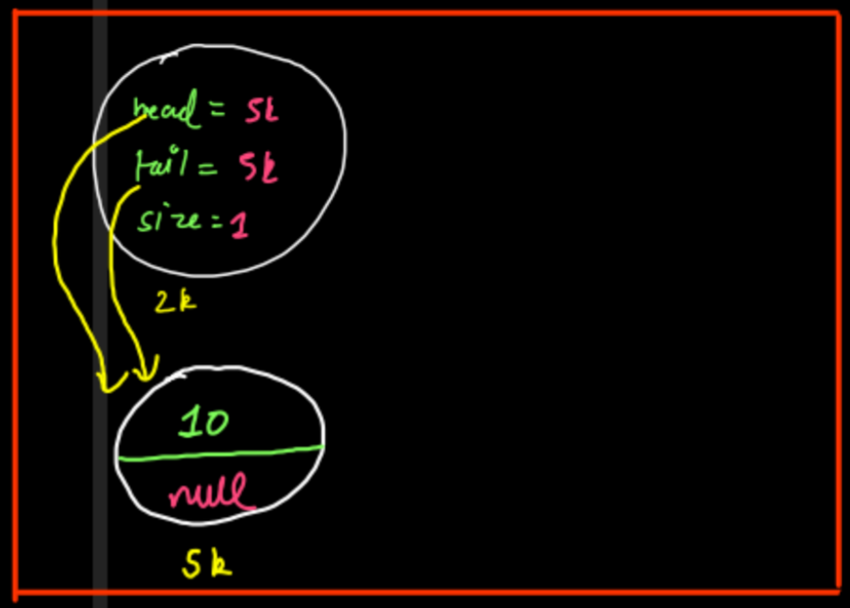
LinkedList list = new LinkedList();

list.addLast(10);

}



Stack



Heap

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```

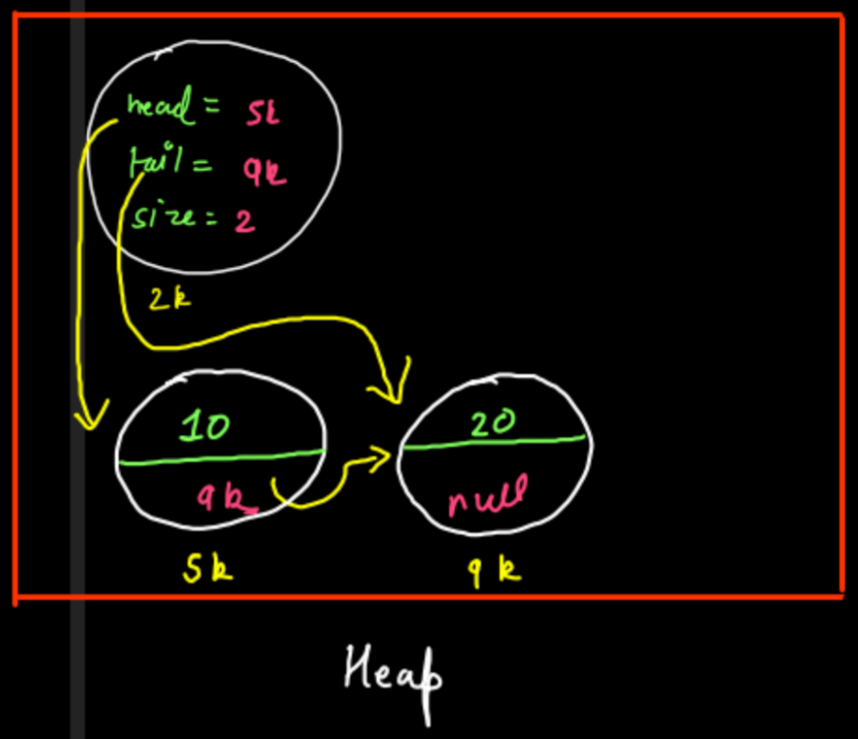
psv main() {

LinkedList list = new LinkedList();

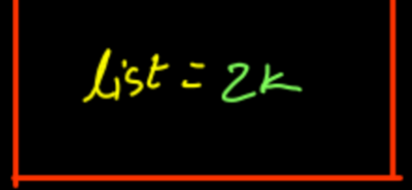
list.addLast(10);

list.addLast(20);

}



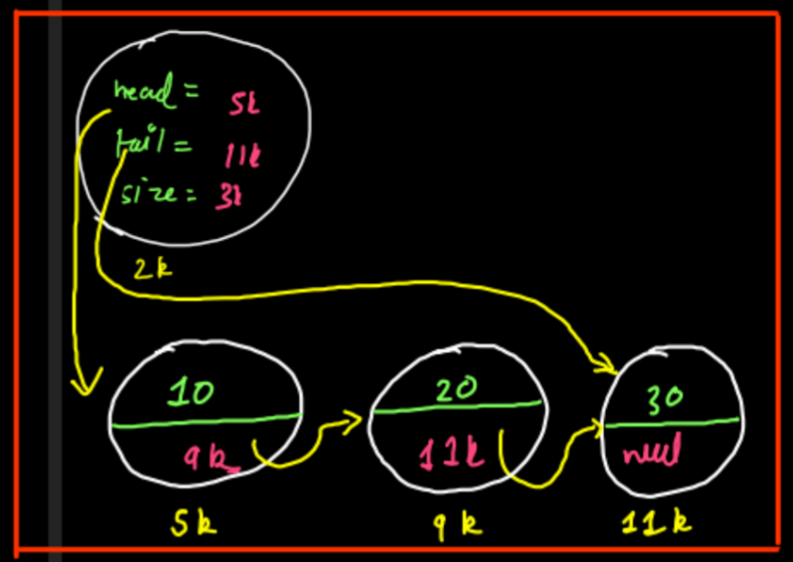
Heap



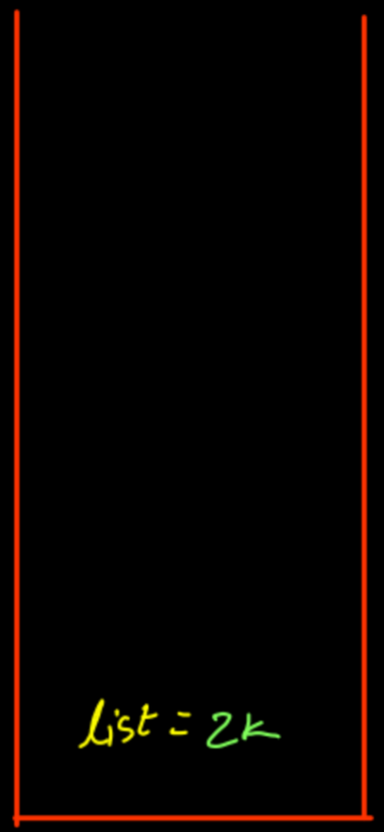
Stack

```java
public static class Node {
    int data;
    Node next; //self referential data member
}
```

```java
public static class LinkedList {

    Node head;
    Node tail;
    int size;

    public void display() {

    }

    public void addFirst(int data) {

    }

    public void addLast(int data) {

    }

    public int get(int index) {

    }

}
```



Heap



Stack

psv main() {

LinkedList list = new LinkedList();

list.addLast(10);

list.addLast(20);

list.addLast(30);

}

** Size of reference is unknown in LL.

LL is taking lot of extra space

Random Access is not allowed in LL

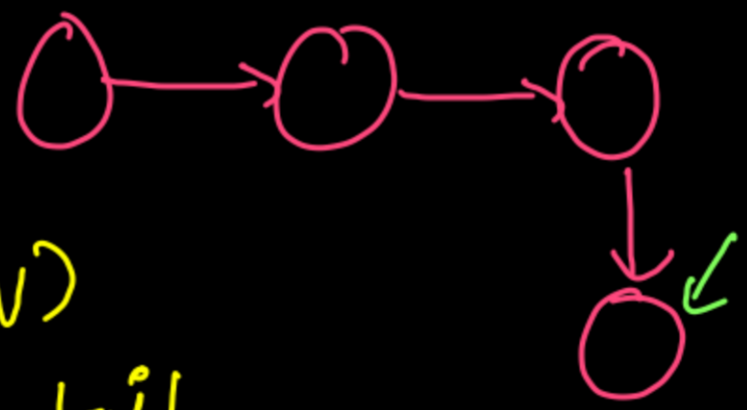
Add Last Code

```
void addLast(int val) {
  // Write your code here
  Node temp = new Node();
  temp.data = val;

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

  size++;
}
```

TC = O(1) → using tail

addLast is a constant operation

#If tail is not given

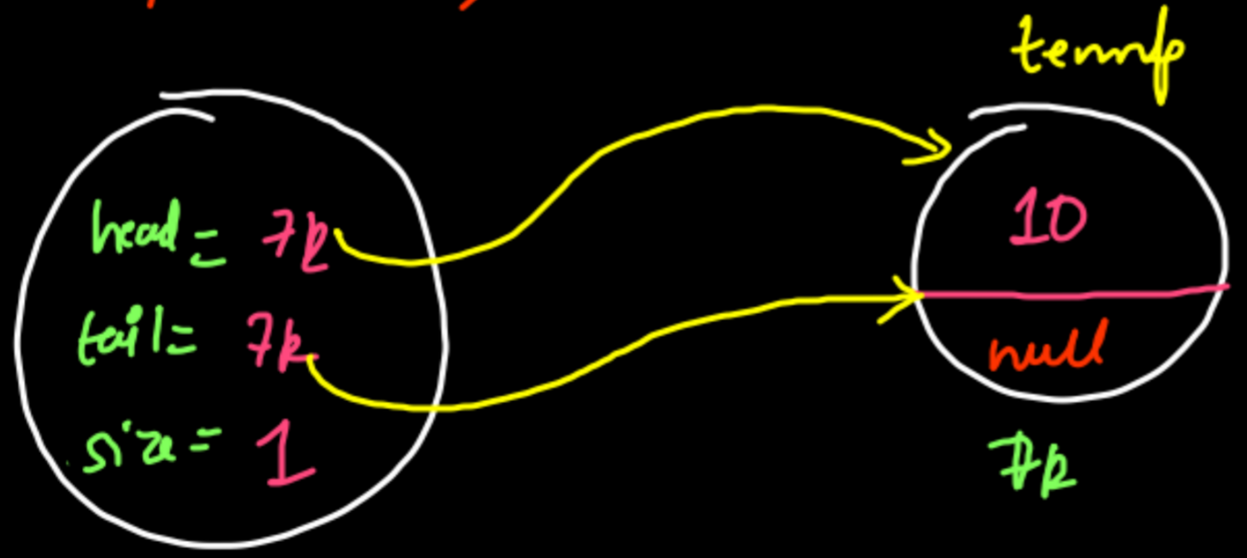


TC = O(N)
without tail

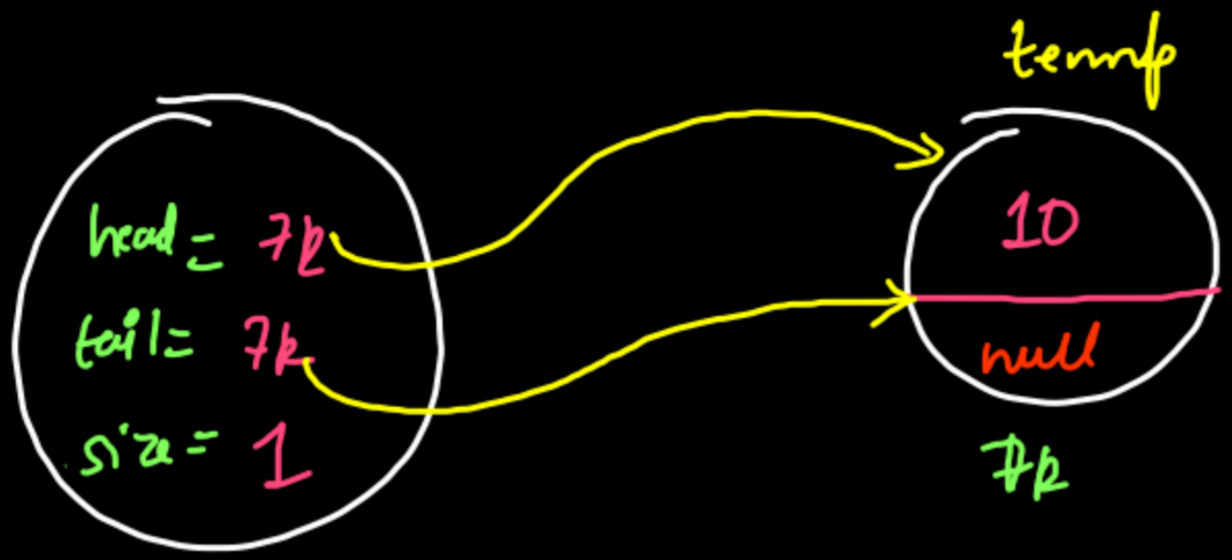
Add first

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

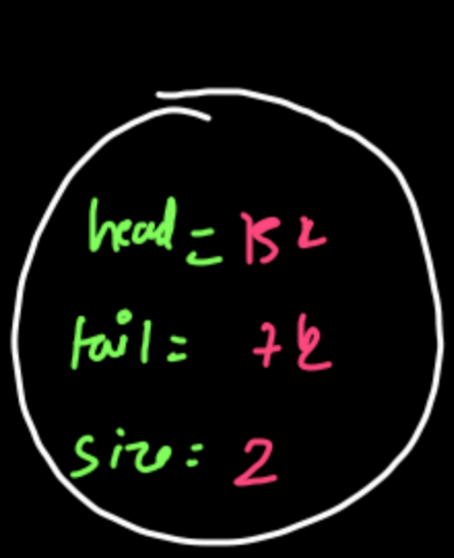
add first(10);

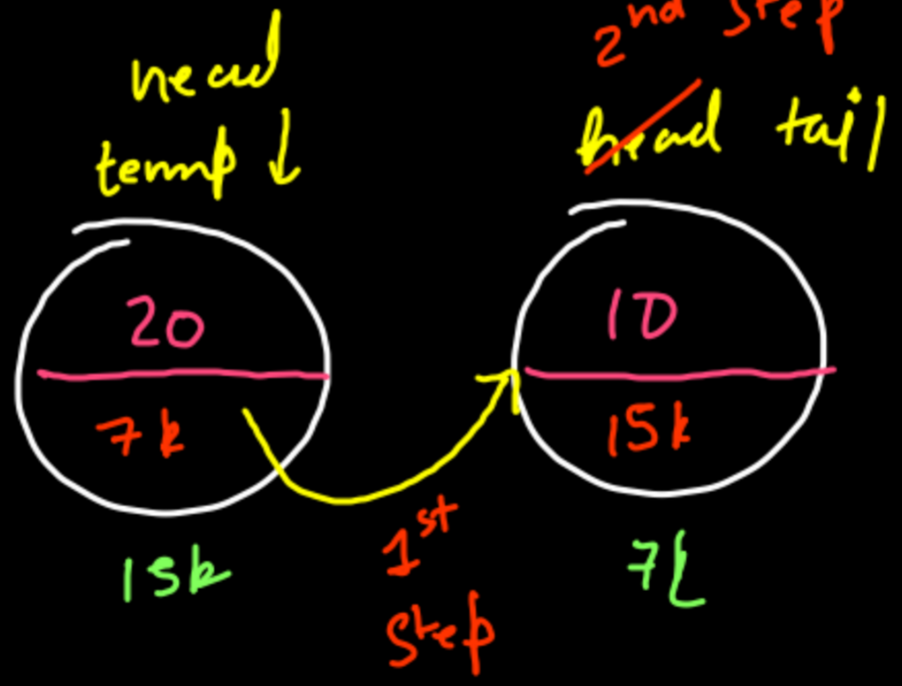



```
if (size == 0)
{ head = temp;
  tail = temp;
}
```



addfirst(20);





if size != 0

1 temp.next = head;

2 head = temp;

Add first Code

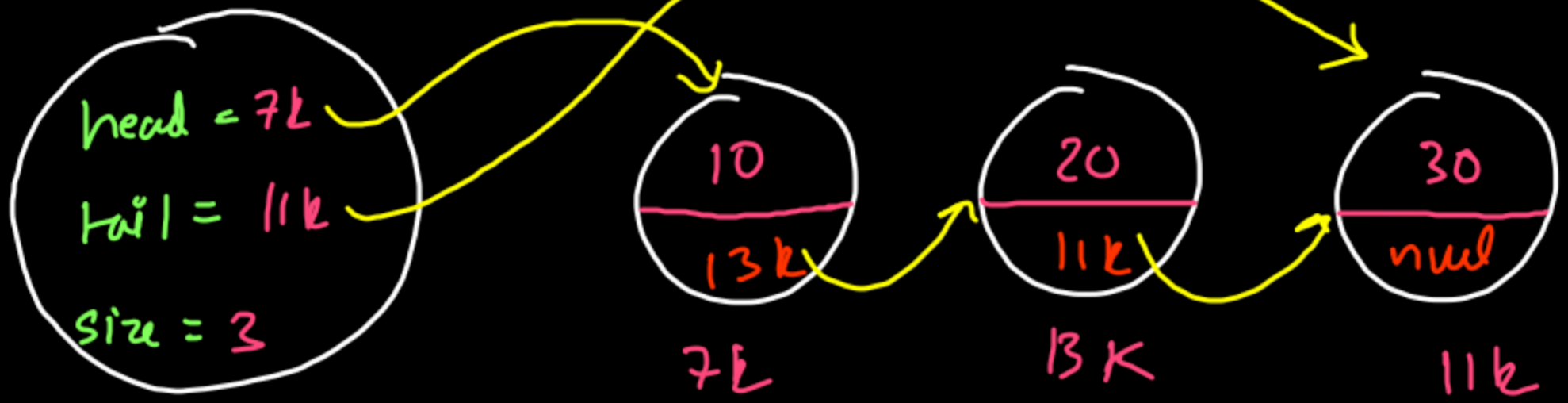
```java
public void addFirst(int val) {
  // write your code here

  Node temp = new Node();
  temp.data = val;

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

  size++;
 }
}
```

TC = O(1)

⇓

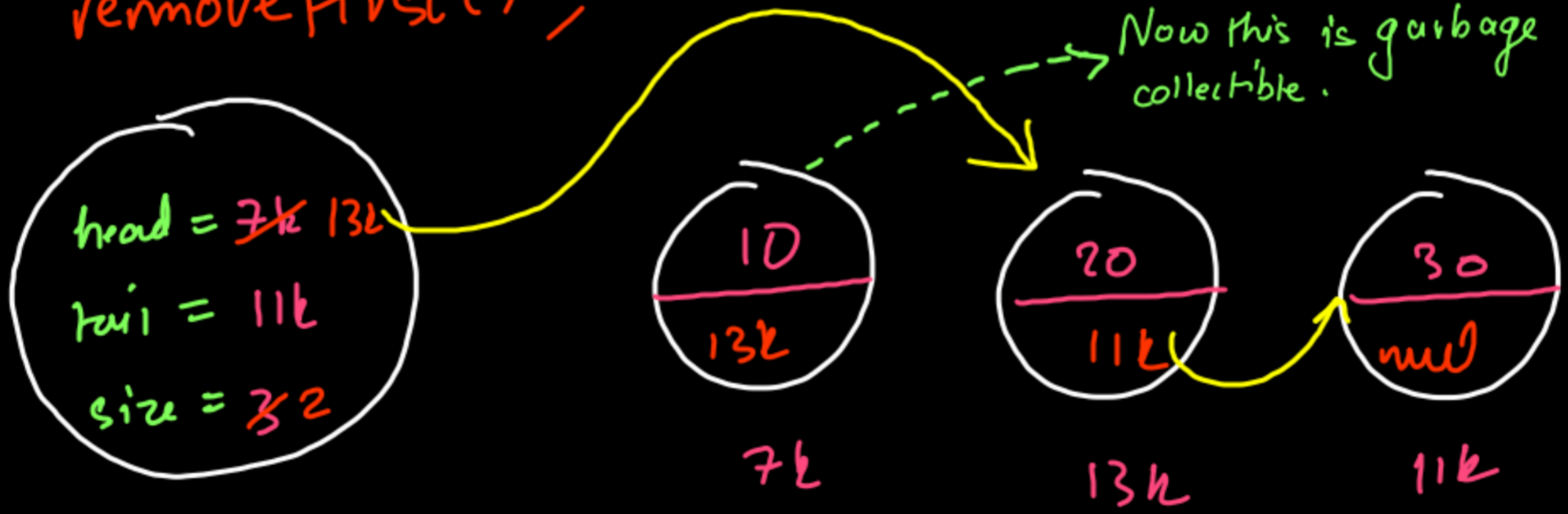
always constant because head will always be there.

Remove first



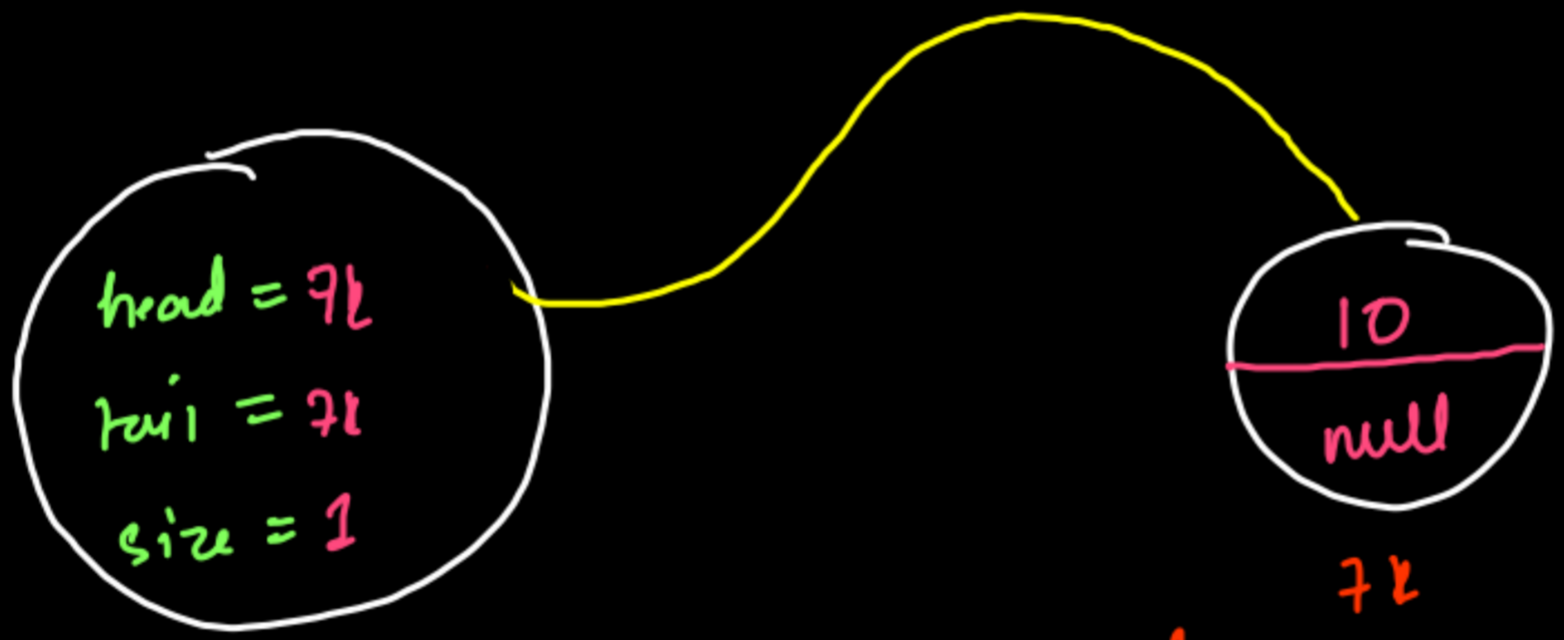
removeFirst();

if (size>1) head = head.next;



if size is 1 then deleting node will empty the LL.

When LL is empty, head = tail = null



head = tail = null

Removefirst Code

```java
public void removeFirst(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return;
  }

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

  size--;
}
```

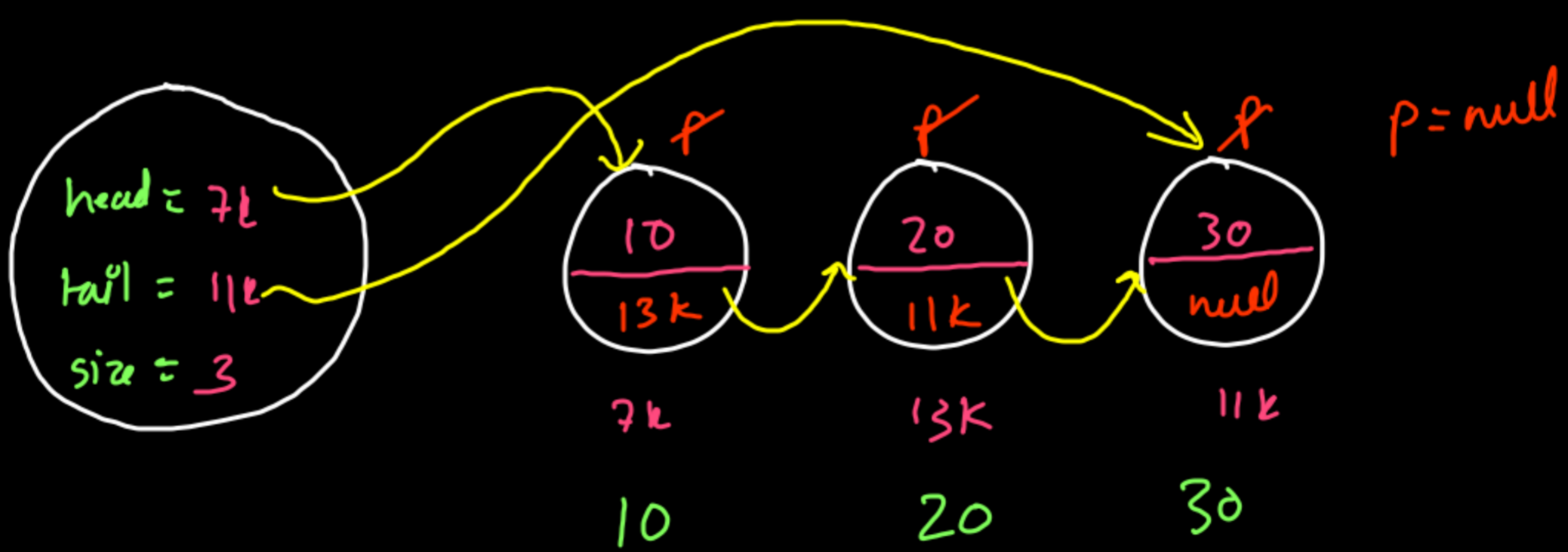
Display List

↗ Using Size

```java
public void display(){
  // write code here

  if(size == 0) return;

  Node p = head;

  for(int i=1;i<=size;i++) {
      System.out.print(p.data + " ");
      p = p.next;
  }
  System.out.println();
}
```

↗ Without using size.

```java
public void display(){
  // write code here

  if(size == 0) return;

  Node curr = head;

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



Get Value in LL

GetFirst()

return head.data;
else return -1;
} In case of invalid index print "Invalid arguments"

If list is empty print "List is empty"

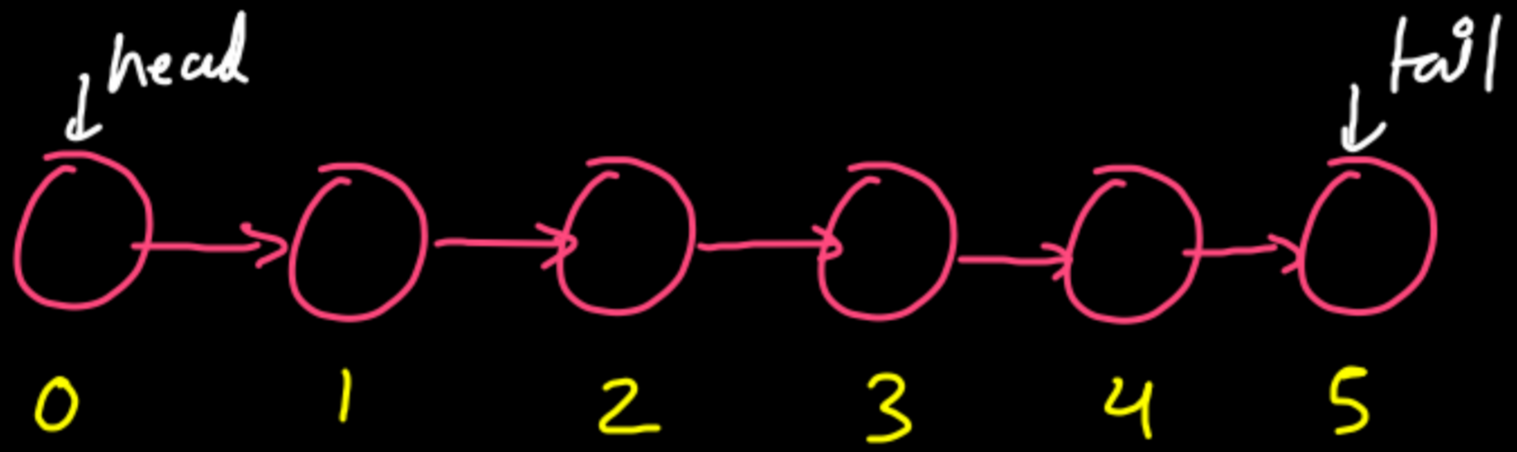
GetLast()

return tail.data;
else return -1;
} In case of invalid index print "Invalid arguments"

If list is empty print "List is empty"

GetAt(Index)

Indexing is 0 based



Apply a Loop now starting from 0 to idx-1
& return curr.data.

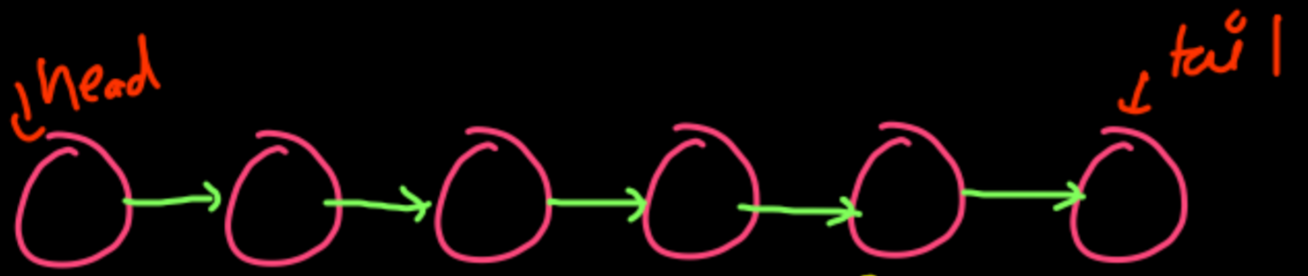
All Get Codes

```java
public int getFirst(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return -1;
  } else {
      return head.data;
  }
}
```

```java
public int getLast(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      return -1;
  } else {
      return tail.data;
  }
}
```

```java
public int getAt(int idx){
  // write your code here
  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;
 }
}
```

Removelast



we need previous of tail to delete tail.

Hence, even if we have tail, we have to traverse the LL to delete the lastNode.

Hence, the Time Complexity will always be O(N).

① Go till prev of tail.

```
Node prevTail = get(size-2);
prevTail.next = null;
tail = prevTail;
```

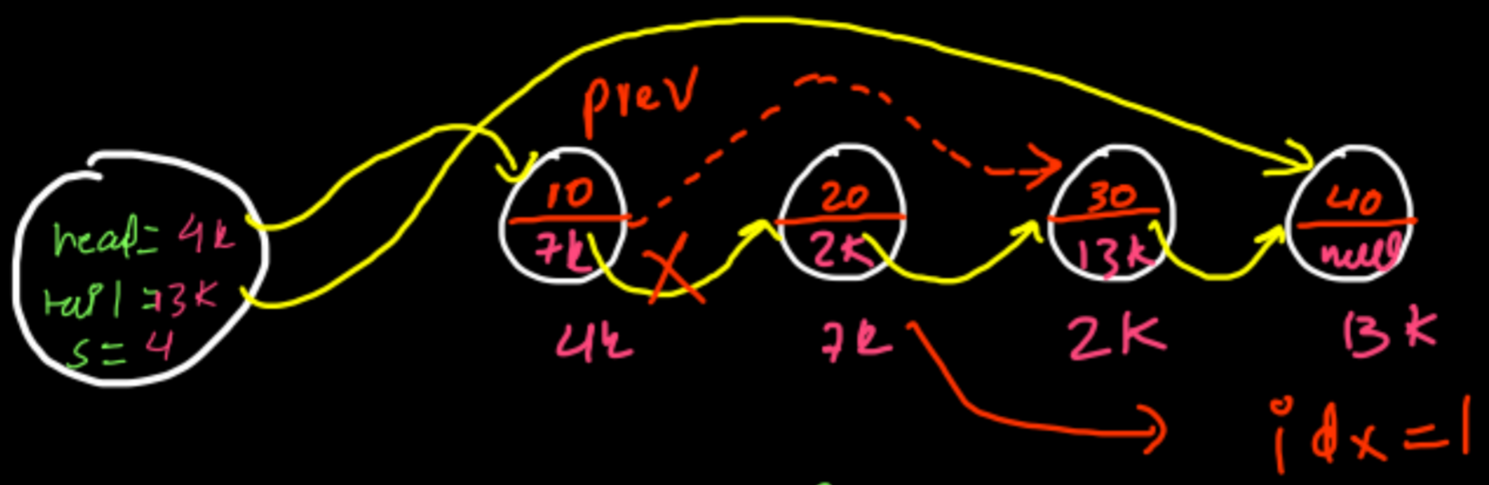
} size > 0

```
if (size == 1)
    head = tail = null;
```

Remove Last Code

```java
public void removeLast(){
  // write your code here
  if(size == 0) {
      System.out.println("List is empty");
      size--;
      return;
  }

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

  Node curr = head;
  for(int i=0;i<size-2;i++) {
      curr = curr.next;
  }

  curr.next = null;
  tail = curr;

  size--;
 }
}
```

Remove At



1) idx == 0 → removeFirst();

2) idx == size - 1 → removeLast();

3) removeAt idx → ① Get $(idx-1)^{th}$ node

$0 < idx < size - 1$ ② prev.next = prev.next.next

③ size--;

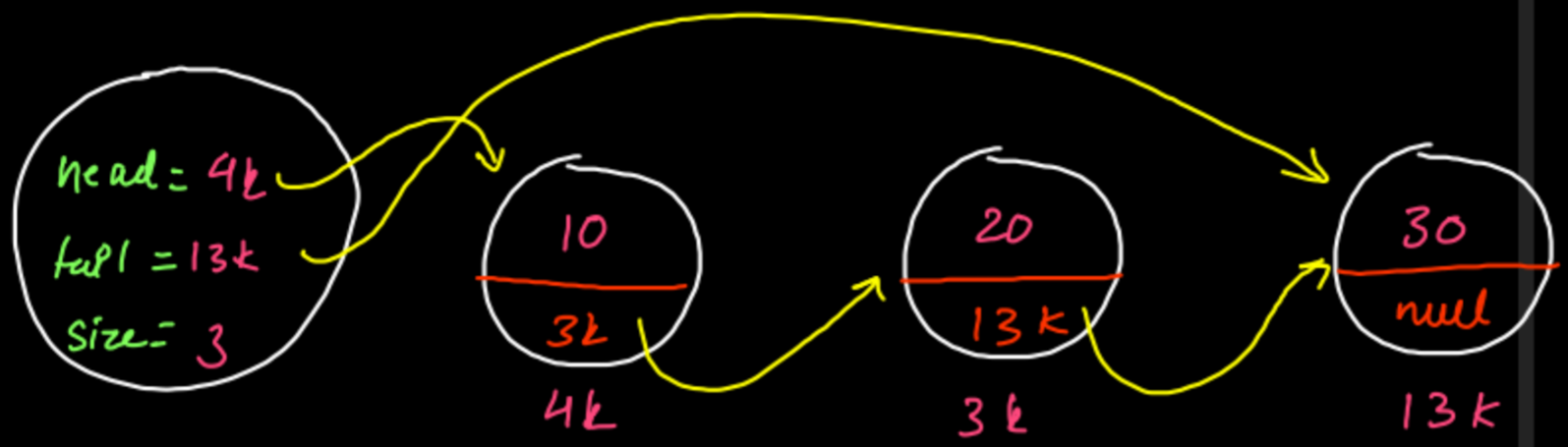
idx < 0 || idx >= size } invalid args

size == 0 } LL is empty

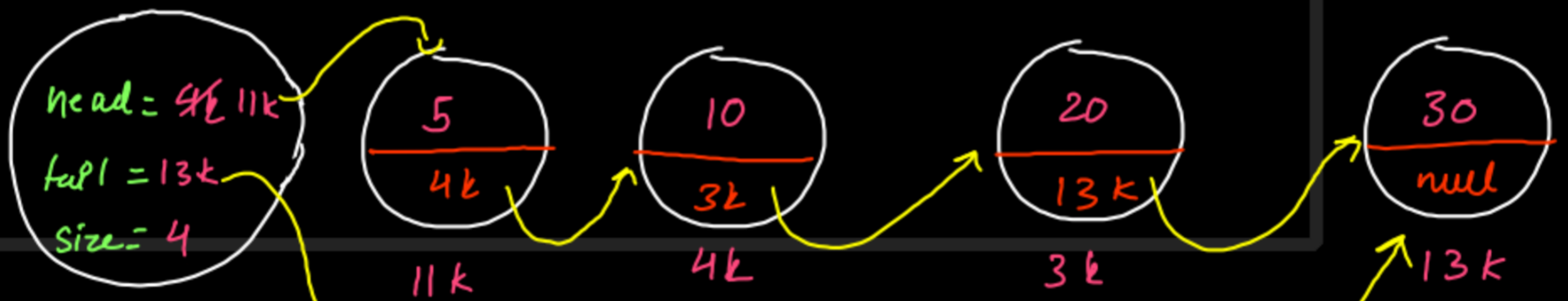
Remove At Code

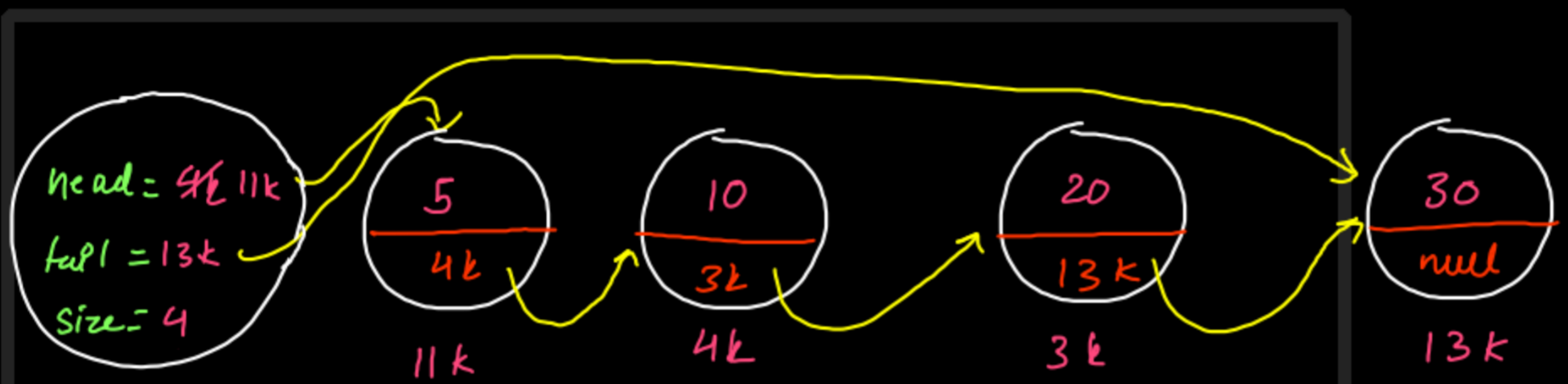
```java
public void removeAt(int index) {
  // write your code here
   if(index < 0 || index >= size) {
        System.out.println("Invalid arguments");
        return;
    }

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

    if(index == 0) {
        removeFirst();
        return;
    }

    if(index == size - 1) {
        removeLast();
        return;
    }

    Node curr = head;
    for(int i=1;i<index;i++) {
        curr = curr.next;
    }

    curr.next = curr.next.next;
    size--;
}
```

Add At

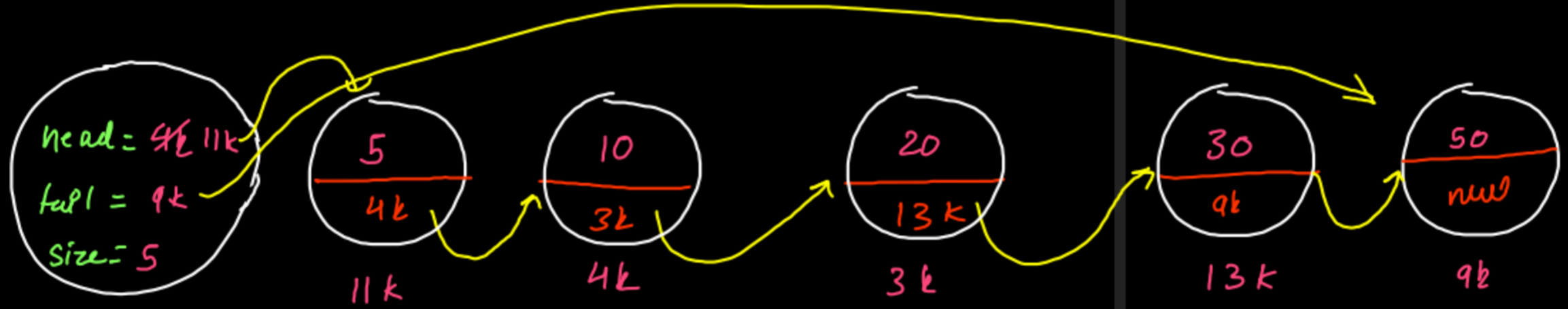


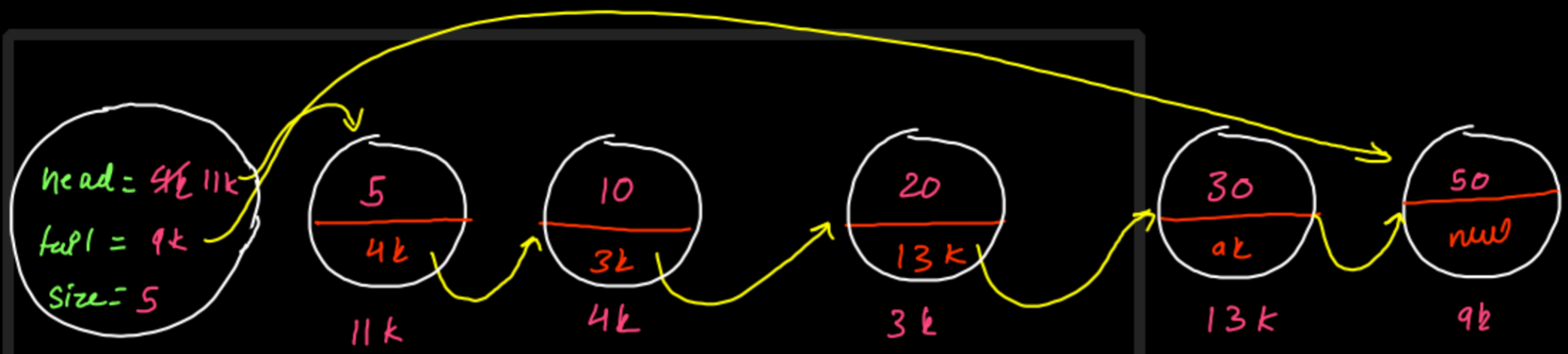
addAt(0,5) -> addFirst();



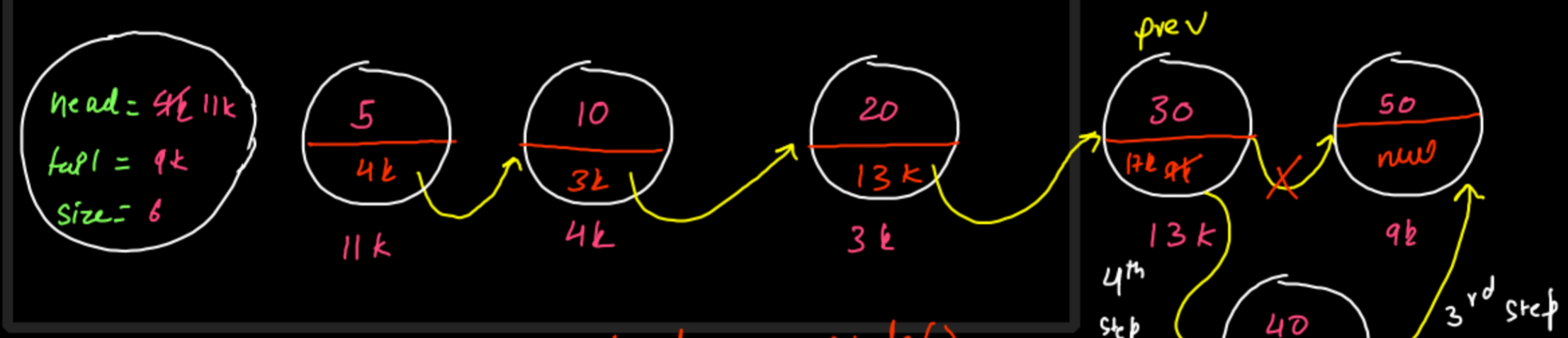


addAt (4, 50) ⇒ addLast ()





addAt(4,40); ⇒ addAt(size-1) is not addAtTail




```
1 temp = new Node();
2 temp.data = val;
3 temp.next = prev.next;
4 prev.next = temp;
```

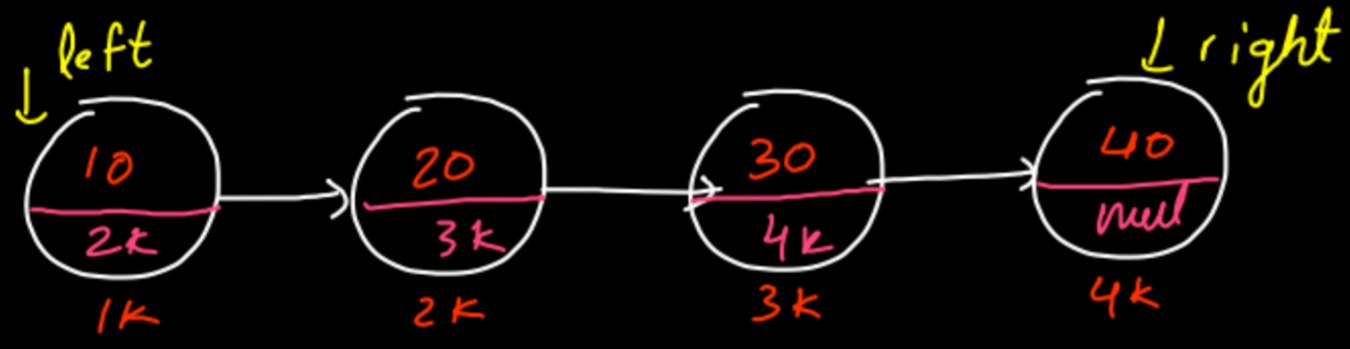
Add At Code

```java
public void addAt(int idx, int val){
  // write your code here

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

  if(idx == 0) {
      addFirst(val);
      return;
  }

  if(idx == size) {
      addLast(val);
      return;
  }

  Node prev = head;

  for(int i=1;i<idx;i++) {
      prev = prev.next;
  }

  Node temp = new Node();
  temp.data = val;
  temp.next = prev.next;
  prev.next = temp;
  size++;
 }
}
```

Reverse a Linked List

1. Data Iterative
2. Pointer Iterative
3. Data Recursive
4. Pointer Recursive

Reverse a Linked List Data Iterative



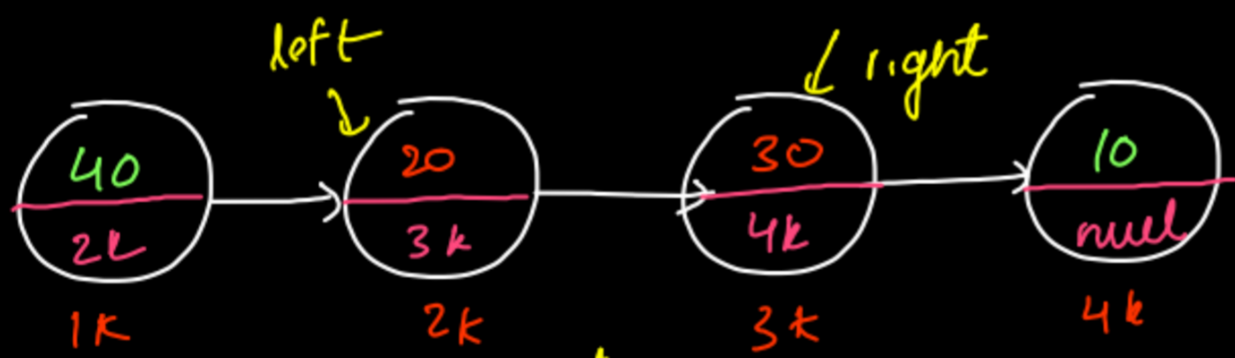
getAt (left) } O(N)
getAt (right) }

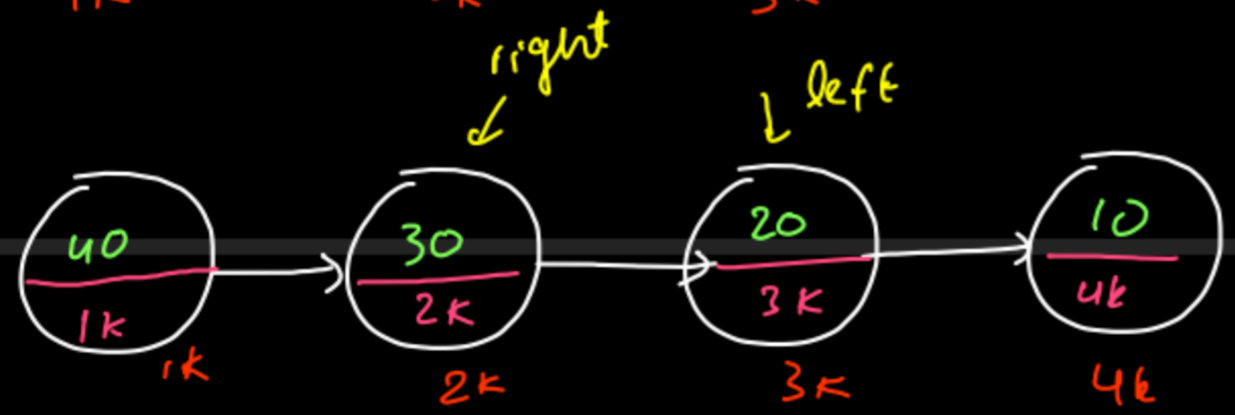
do this for N times

$\frac{N}{2}$

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

$\Rightarrow TC = O(N^2)$



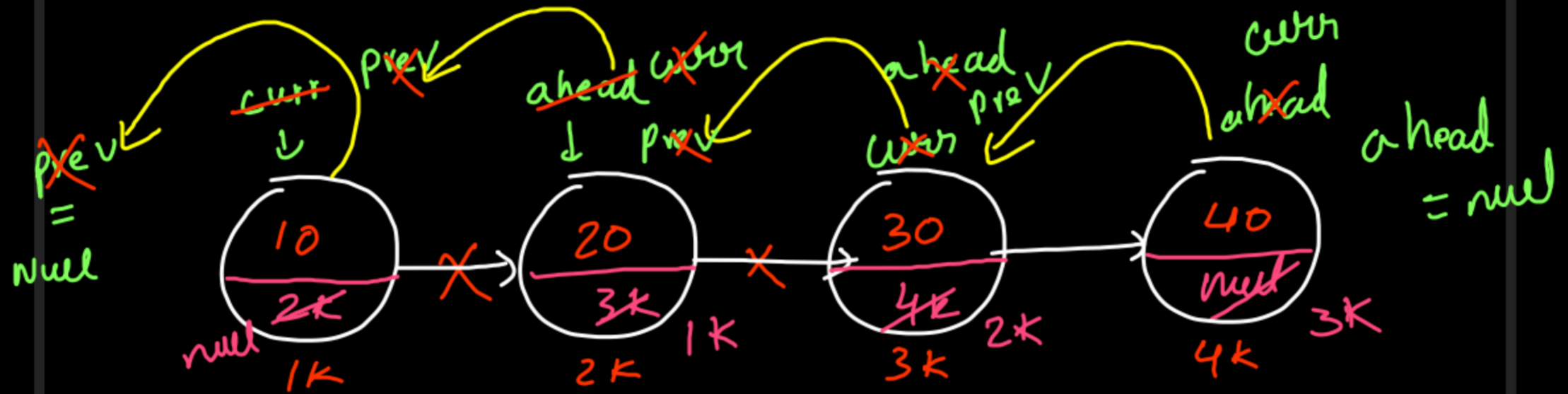


Reverse DI Code

```
public void reverseDI() {
  // write your code here
  int left = 0;
  int right = size - 1;

  while(left <= right) {
      Node leftNode = getNodeAt(left);
      Node rightNode = getNodeAt(right);
      int temp = leftNode.data;
      leftNode.data = rightNode.data;
      rightNode.data = temp;
      left++;
      right--;
    }
  }
}
```

However it is an $O(n^2)$ approach, it is still the only & best possible approach for reversing LL DI way.

Reverse a Linked List Pointer Iterative



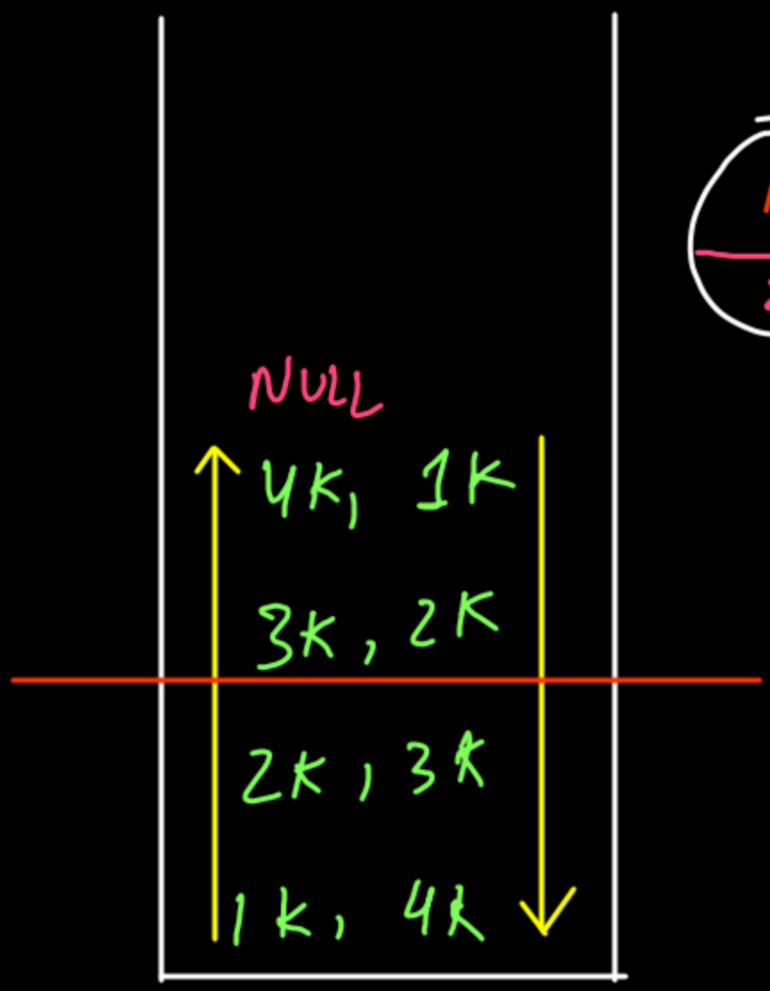
We will need 3 pointers

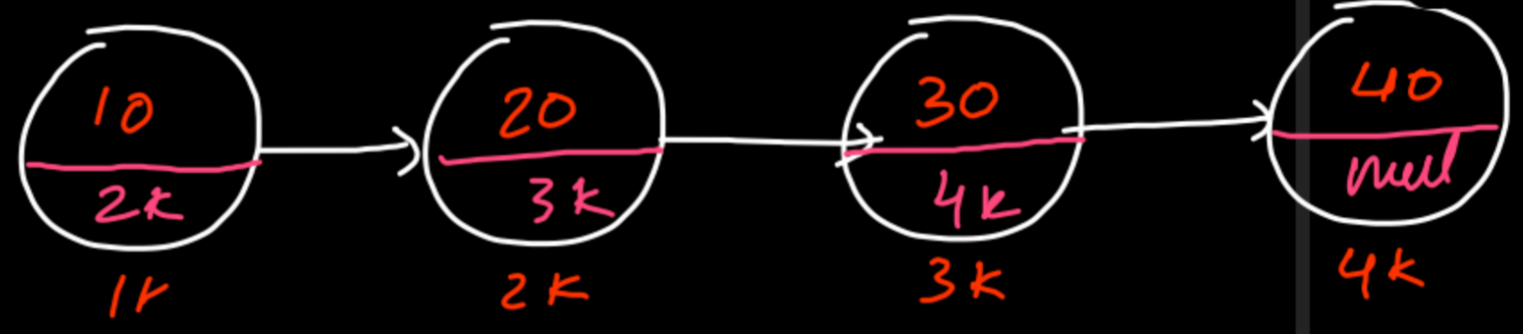
```
curr.next = prev;
prev = curr;
curr = ahead;
```
} while (curr != null)

Reverse a LL Pointer Iterative Code

```
public void reversePI(){
  // write your code here
  if(size == 0 || size == 1) {
      return;
  }
  Node prev = null;
  Node curr = head;

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

//   swap head and tail
  Node temp = head;
  head = tail;
  tail = temp;
}
```

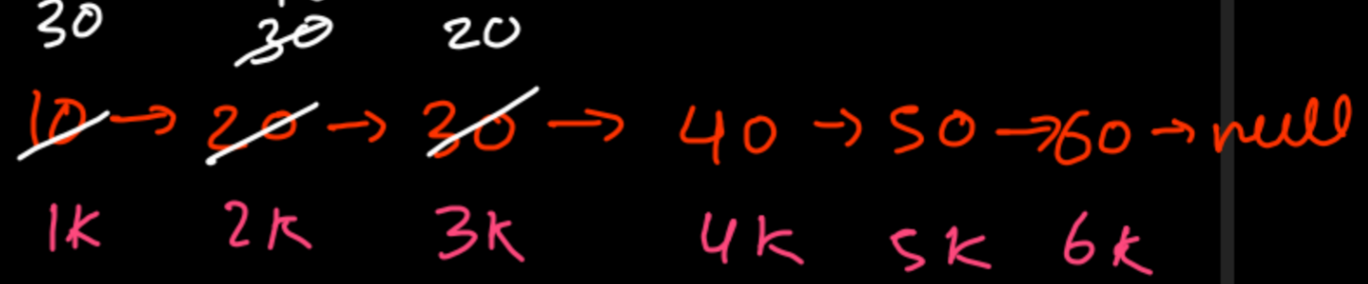
Reverse a LL Data Recursive





Do only 1/2 times because else the LL will remain as it is.

30 10 20



l = null

l = 6k r = 1k c = 5

l = 5k r = 2k c = 4

l = 4k r = 1k c = 3

l = 3k r = 2k c = 2

l = 2k r = 1k c = 1

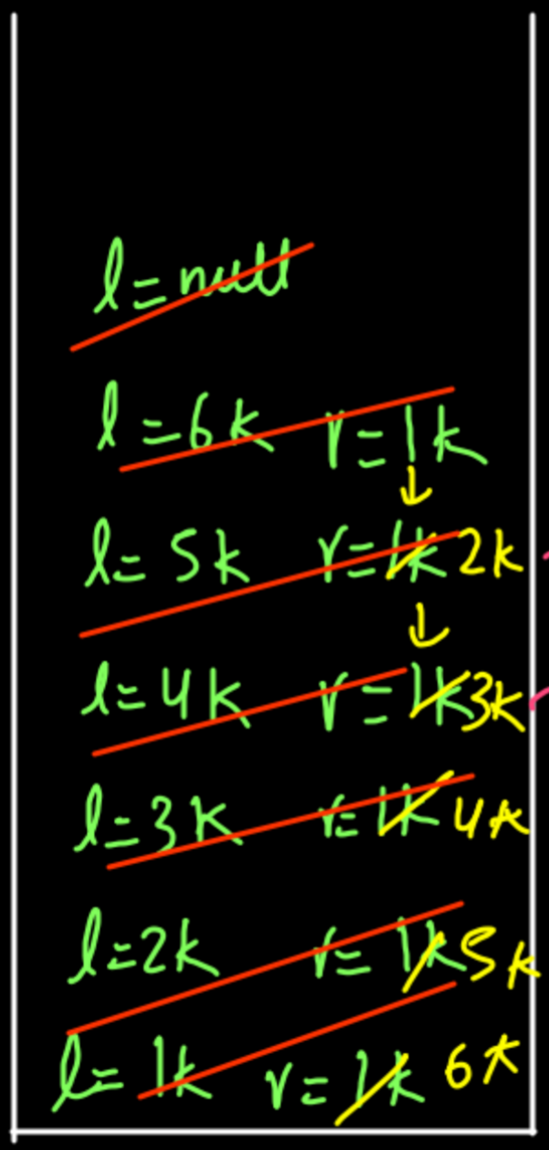
l = 1k r = 1k c = 0

This is wrong as right = right.next. will persist only locally. To persist it in the next call, make right global or return it.

```
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);
}
```

Dry Run for Data Recursive using Return

if (counter > size(2)



r 60 l 10
10 → 20 → 30 → 40 → 50 → 60 → null
1k 2k 3k 4k 5k 6k

r 50 l 20
60 → 20 → 30 → 40 → 50 → 10 → null
1k 2k 3k 4k 5k 6k

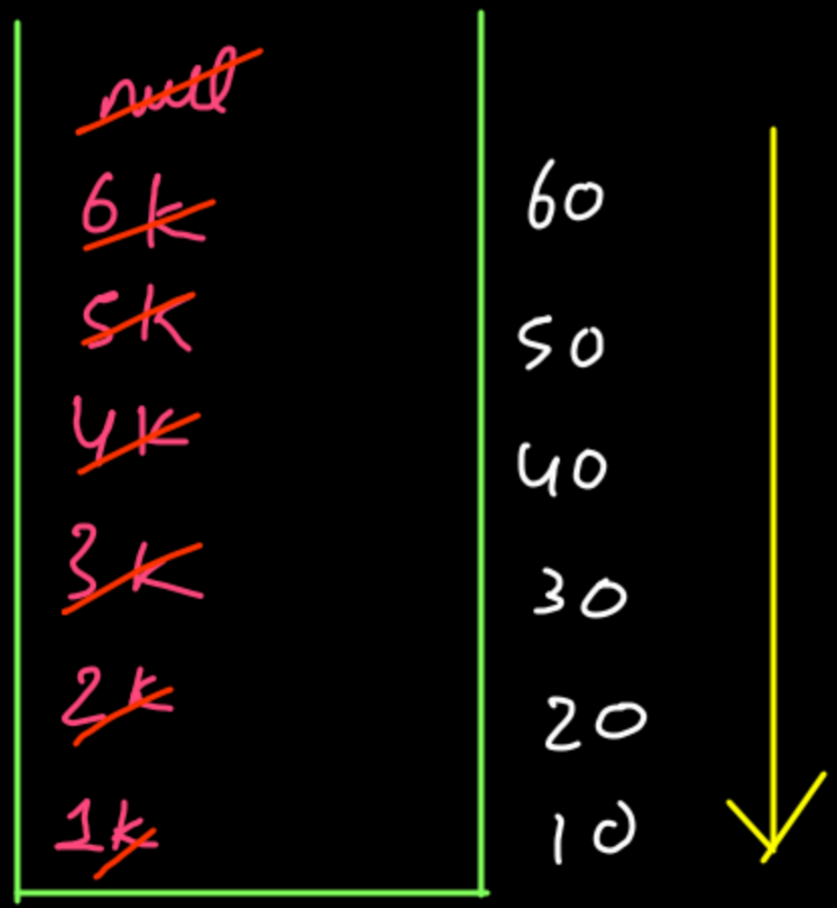
r 40 l 30
60 → 50 → 30 → 40 → 20 → 10 → null
1k 2k 3k 4k 5k 6k

60 → 50 → 40 → 30 → 20 → 10 → null
1k 2k 3k 4k 5k 6k

Display Reverse

10 → 20 → 30 → 40 → 50 → 60

1k 2k 3k 4k 5k 6k



Print in postorder

```
private void displayReverseHelper(Node node){
  // write your code here

  if(node == null) {
      return;
  }

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

public void displayReverse(){
  displayReverseHelper(head);
  System.out.println();
}
```

TC = O(N)

Reverse Linked List Pointer Recursive

10 → 20 → 30 → 40 → 50 → 60 → null

1k 2k 3k 4k 5k 6k

↳ After complete recursion 1k's next still points at 2k.

curr → null
curr → 6k
curr → 5k
curr → 4k
curr → 3k
curr → 2k
curr → 1k

→ meri next node hai 6k. uska next mujhe bana do.

node.next.next = node

So we have to swap head & tail first & then set tail's next to null too.

Reevrse LL Pointer Recursive Code

```
private void reversePRHelper(Node node){
  // write your code here

  if(node == null || node.next == null) {
      return;
  }

  reversePRHelper(node.next);

  //khud pe khade hoke apne next ka kaam karao
  node.next.next = node;
}
```

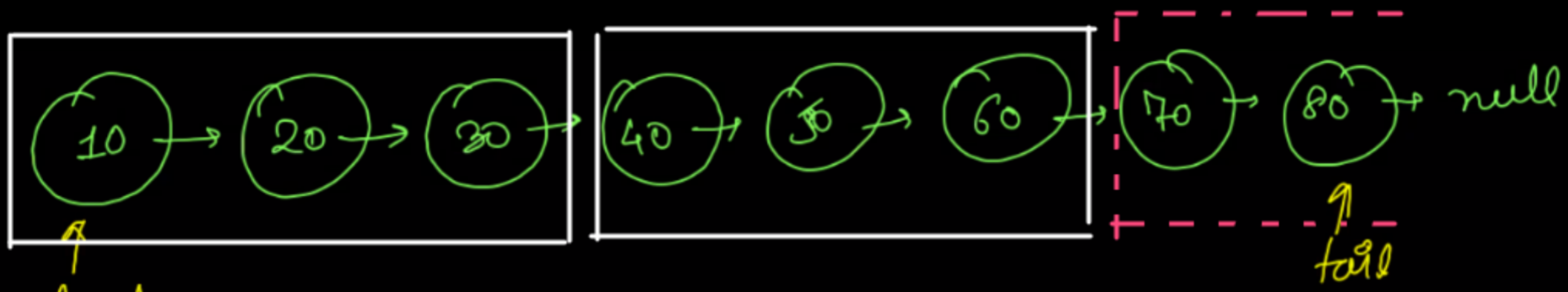
Ye zaroori hai taaki agar head null ho to null pointer exc na aae.

TC = O(N)

```
public void reversePR(){
  // write your code here
  reversePRHelper(head);

  //swap head and tail
  Node temp = head;
  head = tail;
  tail = temp;

  //set tail's next to null
  tail.next = null;
 }
}
```

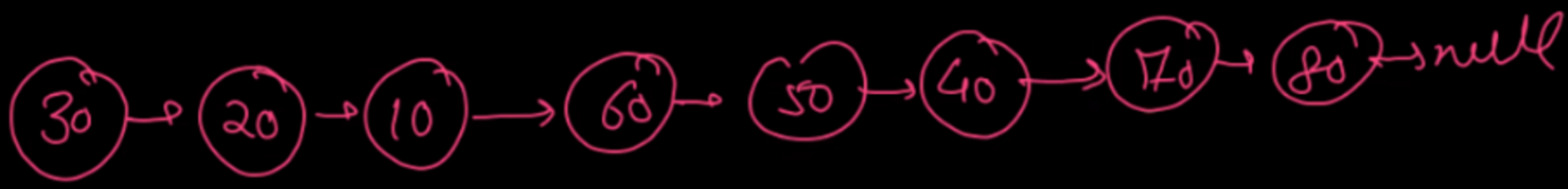
Check for head == null here also to avoid null pointer exception.

K - Reverse in Linked List



k = 3

After k Reverse

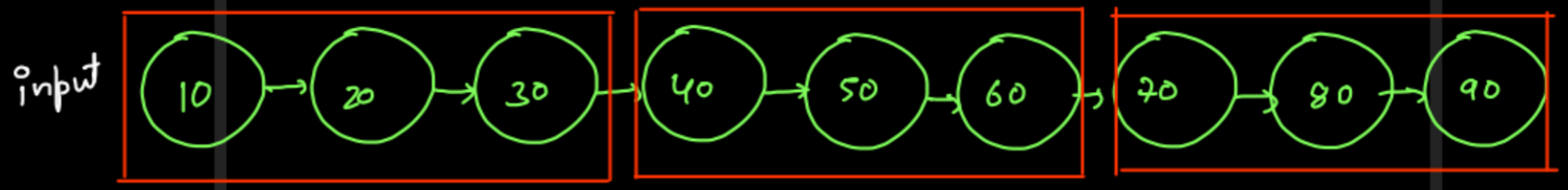


for each group

→ Remove first from Group

→ Add first to curr LL

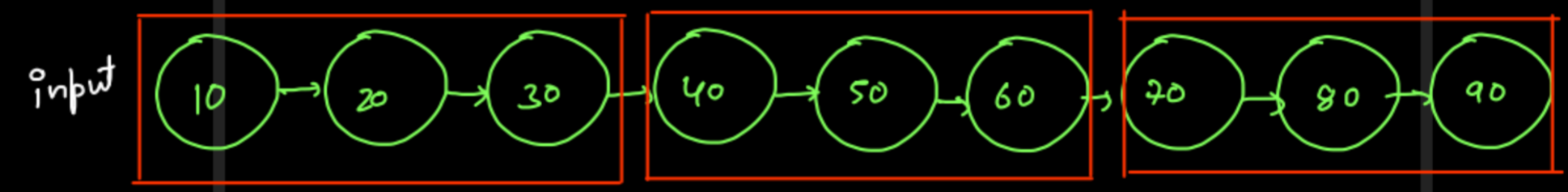
Initial state



prev : empty list

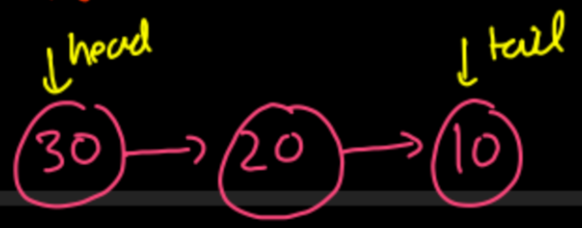
curr : empty list

Reversing 1st Group

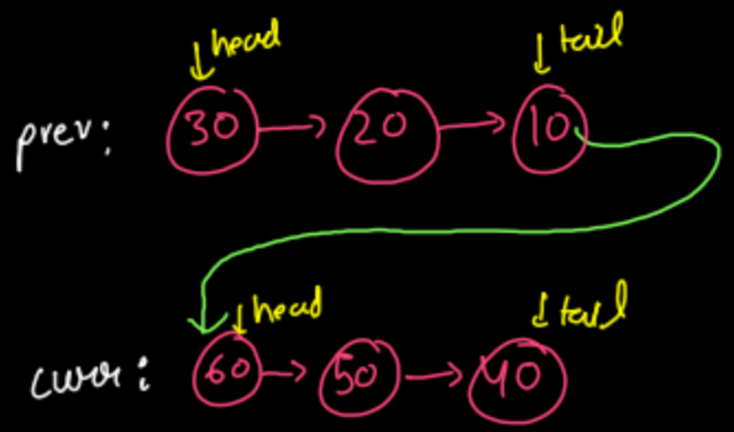


prev : empty list

curr :

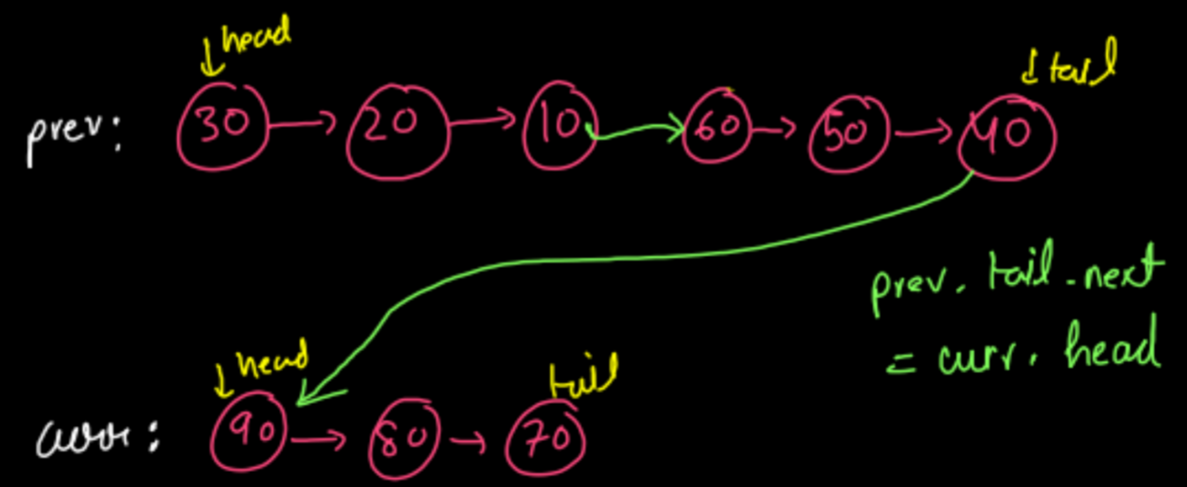


Reversing 2nd Group



prev.tail.next
= curr.head

Reversing 3rd Group

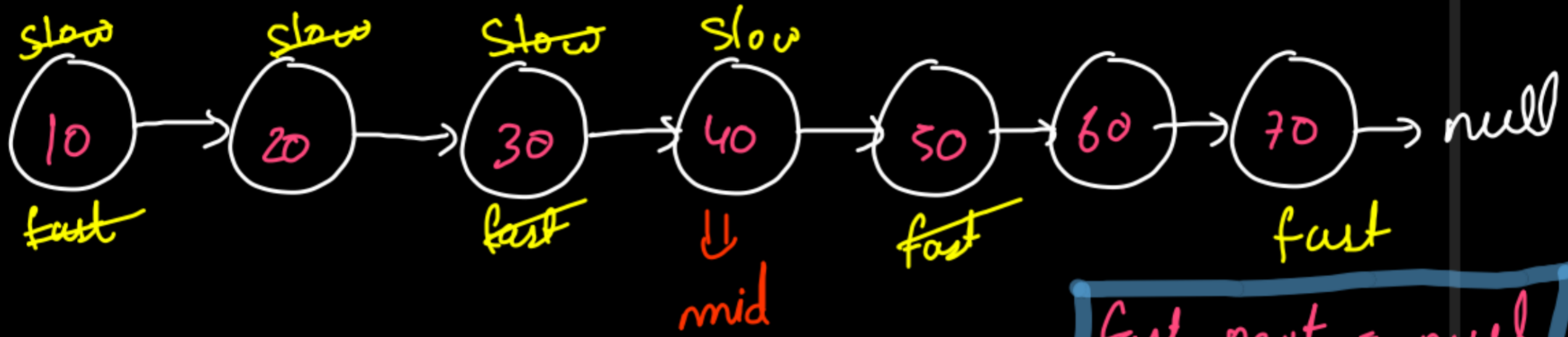


prev.tail.next
= curr.head

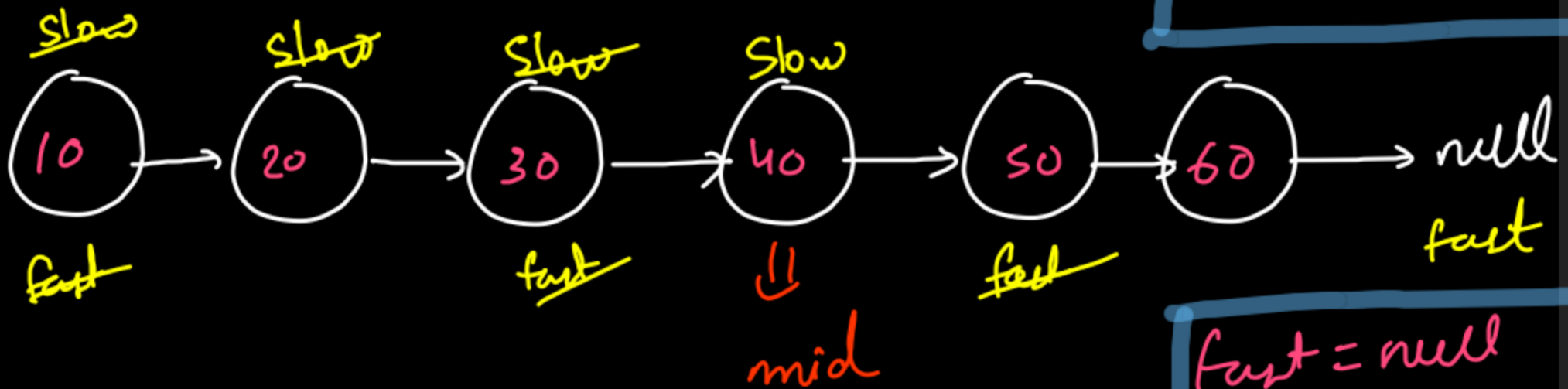
Code for K Reverse

```java
public void kReverse(int k) {
  // write your code here
  LinkedList prev = new LinkedList();

  while(size > 0) {
      LinkedList curr = new LinkedList();

      if(size < k) {
          while(size > 0) {
              int val = head.data;
              this.removeFirst();
              curr.addLast(val);
          }
      }
      else {
          for(int i=0;i<k;i++) {
              int val = head.data;
              this.removeFirst();
              curr.addFirst(val);
          }
      }

      if(prev.head == null) {
          prev = curr;
      } else {
          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;

 }
}
```

Mid of Linked List {Hare and Tortoise}{Two Pointer}



fast.next = null
in odd size



fast = null
in even size

Mid of LL Code

```java
/
class Solution {
    public ListNode middleNode(ListNode head) {

        ListNode slow = head;
        ListNode fast = head;

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

        return slow;
    }
}
```

TC = O(N)

Mathematical Proof

slow

Speed = 1 Node / sec

Time = t

Dist = ? say x

fast

Speed = 2 Nodes / sec

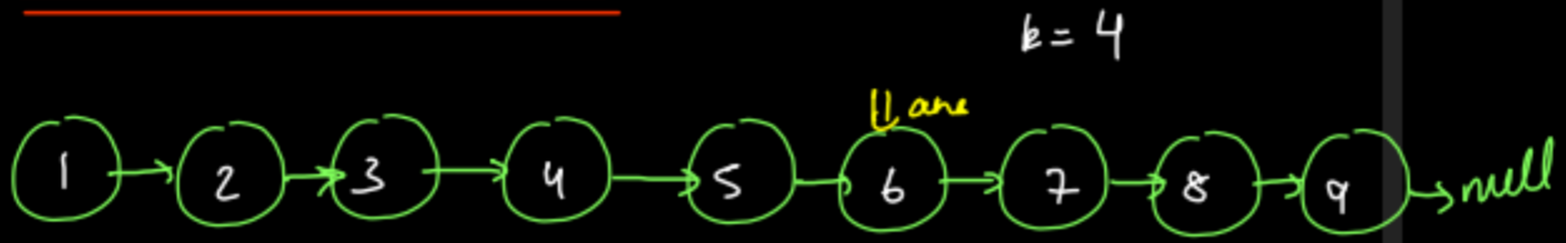
Time = t

Dist = N

$$\Rightarrow \text{time} = \frac{\text{Dist}}{\text{speed}}$$

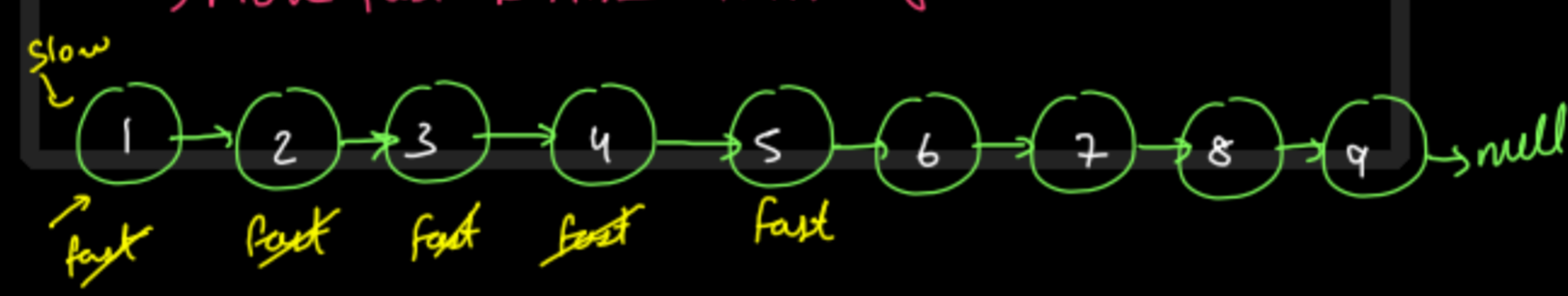
$$\frac{x}{1} = \frac{N}{2} \Rightarrow \boxed{x = \frac{N}{2}}$$

K^{th} Node from End

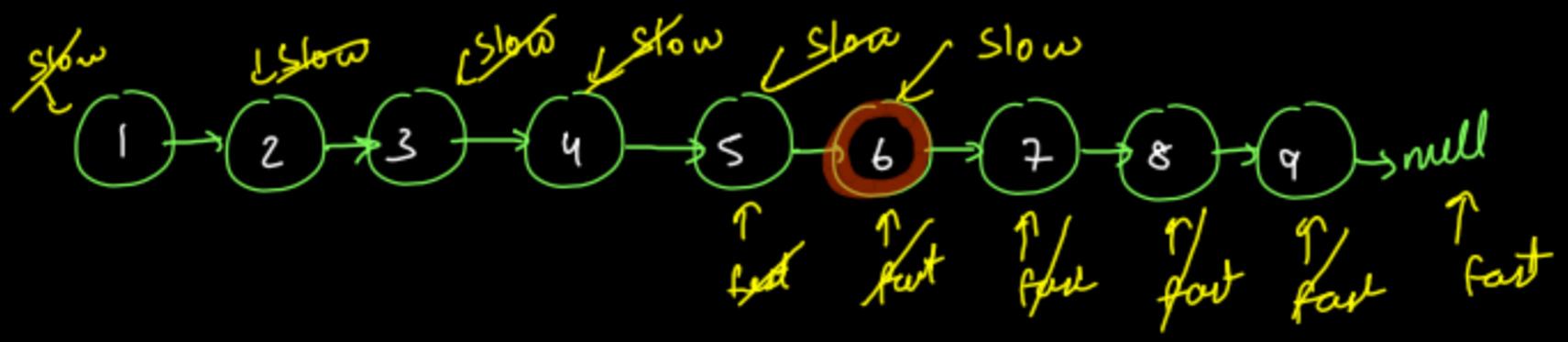


Idea is to have slow at k^{th} node when fast becomes null. To do that, have 2 pointers with some speed but keep some initial distance b/w them.

→ Move fast k times initially



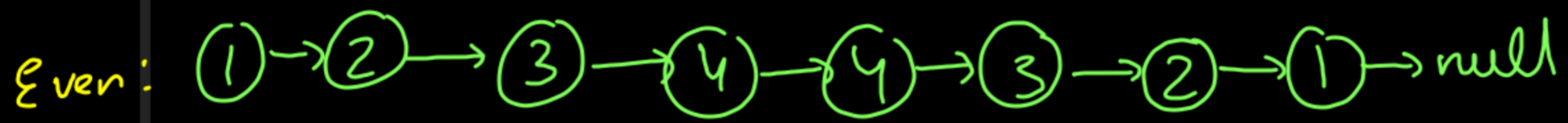
→ Now move slow & fast one step till fast becomes null



K^{th} Node from End Code

```
int getNthFromLast(Node head, int k)
{
      Node slow = head;
     Node 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;
}
```

Palindrome List



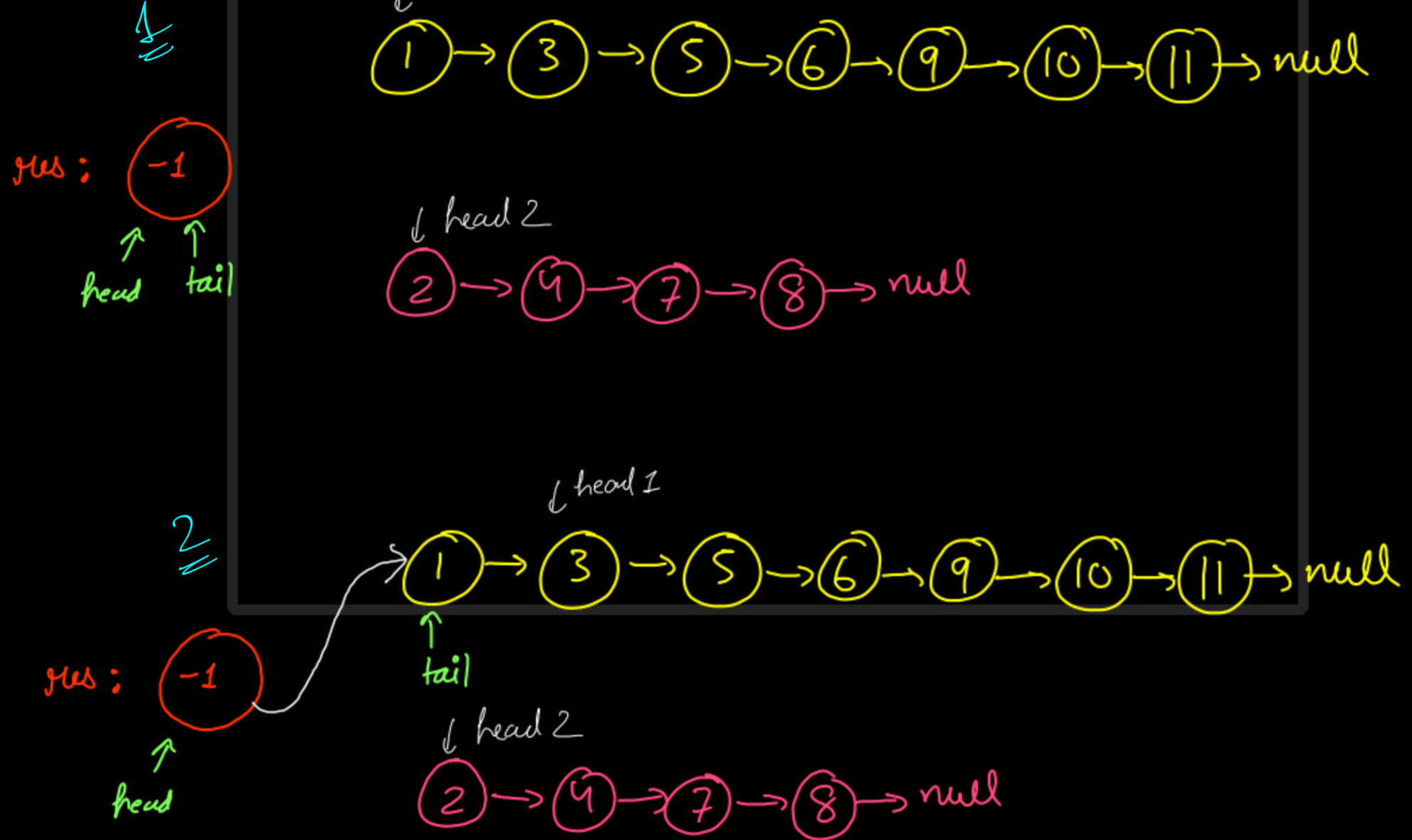
Approach:

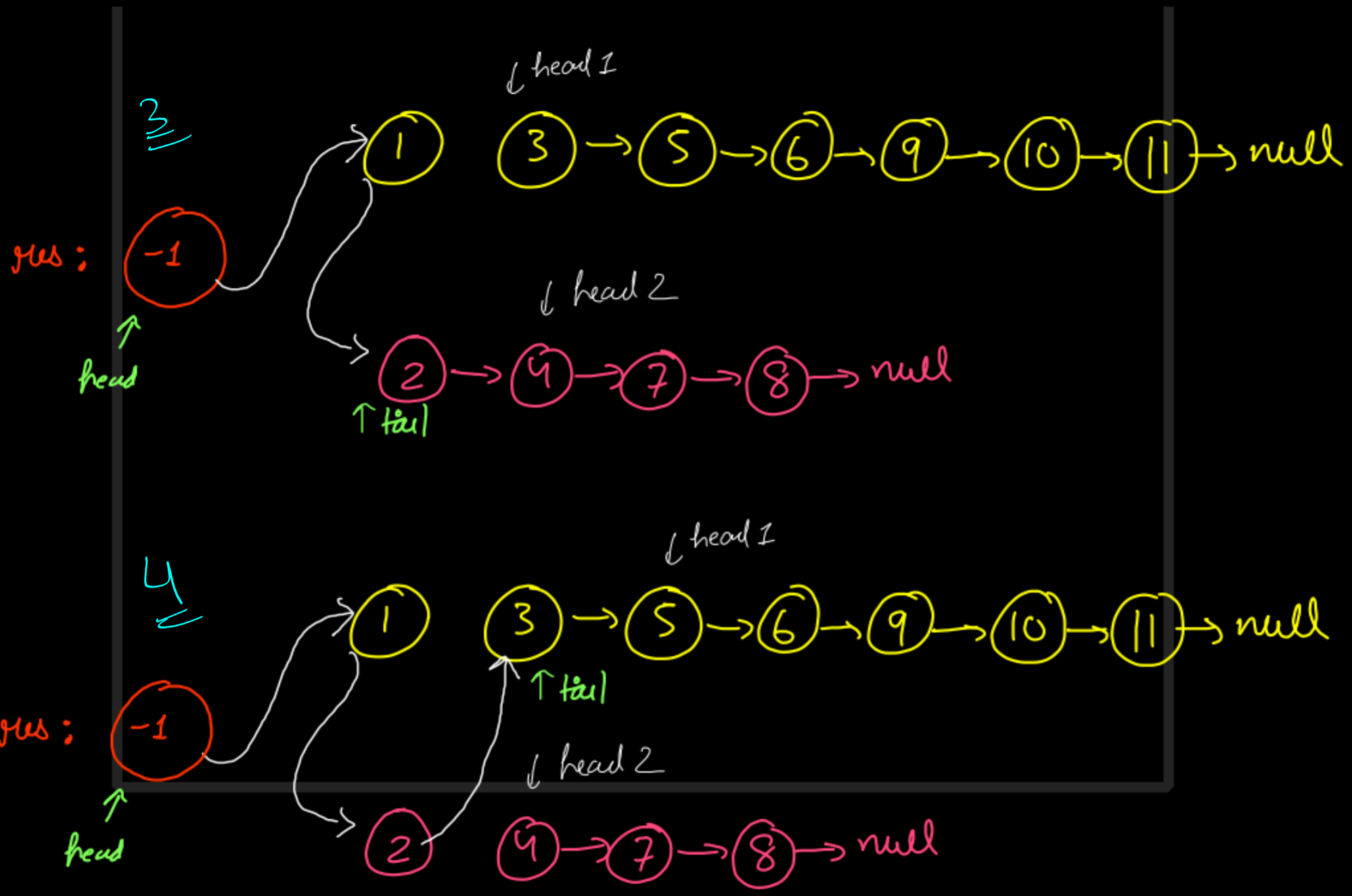
1. Find middle node
2. Reverse the second part
3. Compare both the parts
   - equal → Yes (Palindrome)
   - not equal → No

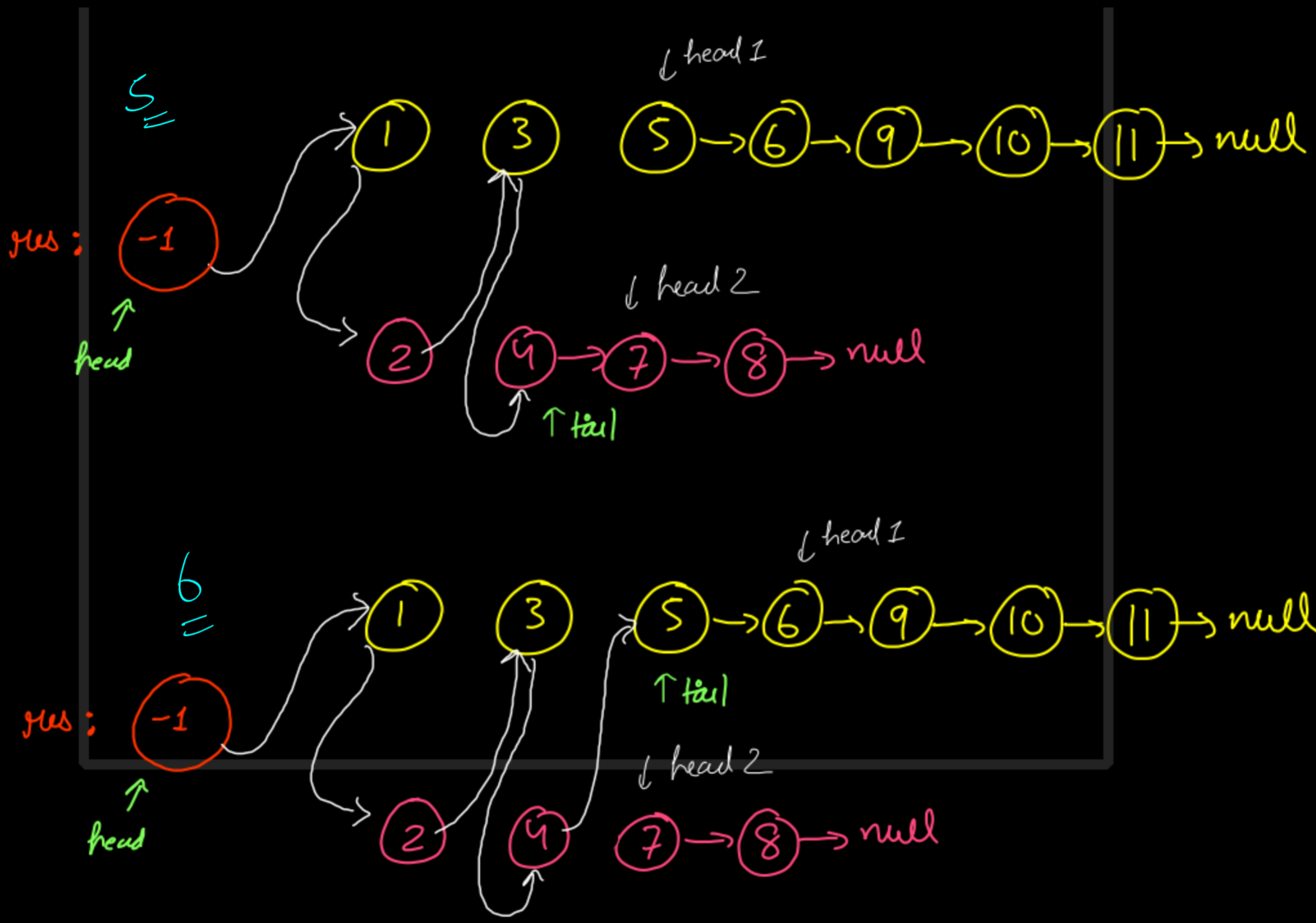
```java
public ListNode reverse(ListNode head){
    ListNode prev = null, curr = head;
    while(curr != null){
        ListNode ahead = curr.next;
        curr.next = prev;
        prev = curr;
        curr = ahead;
    }
    return prev;
}

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;
}
```

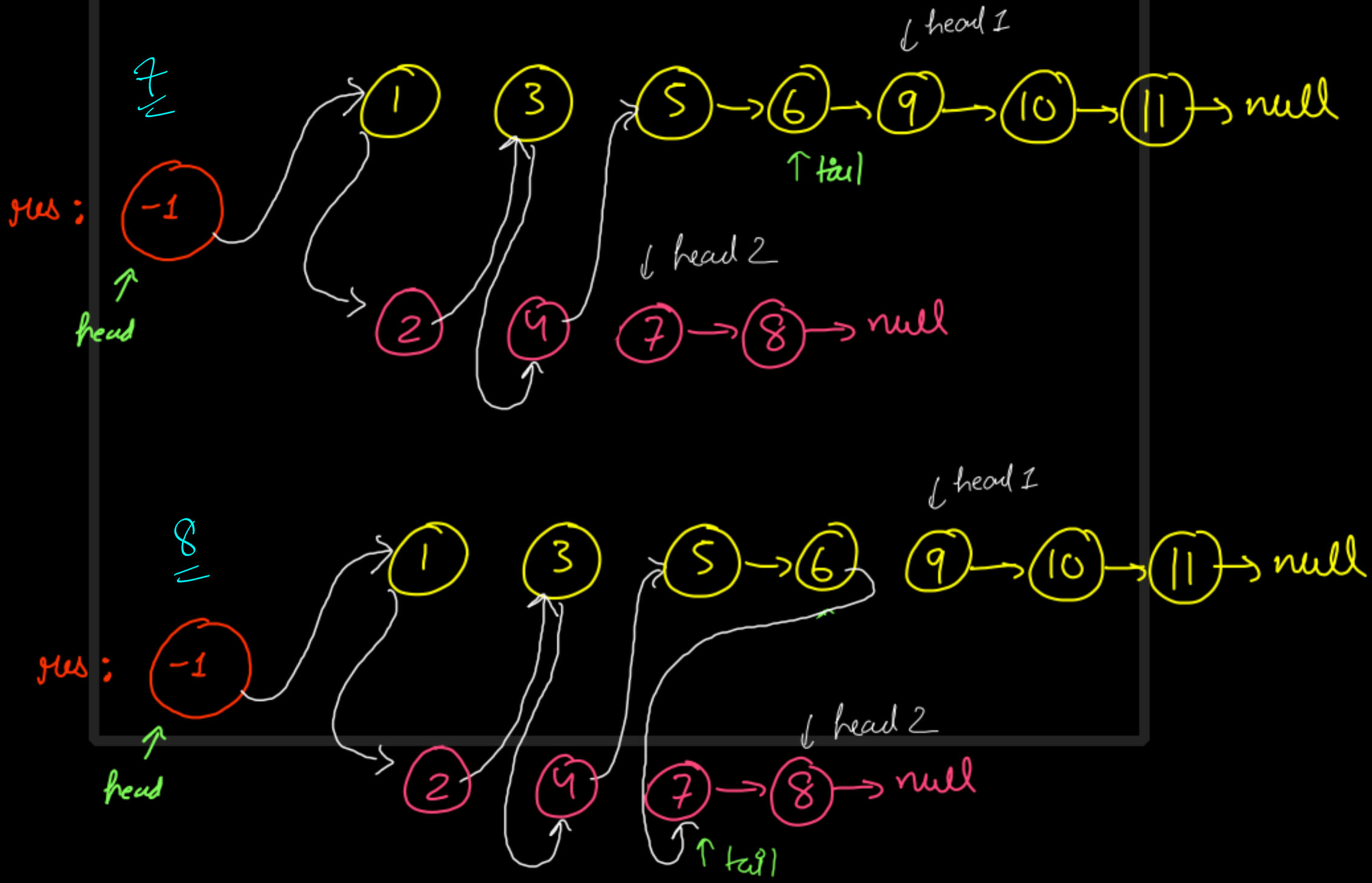
```java
public boolean isPalindrome(ListNode head) {
    if(head == null || head.next == null){
        return true;
    }

    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;
}
```

Merge 2 sorted LL

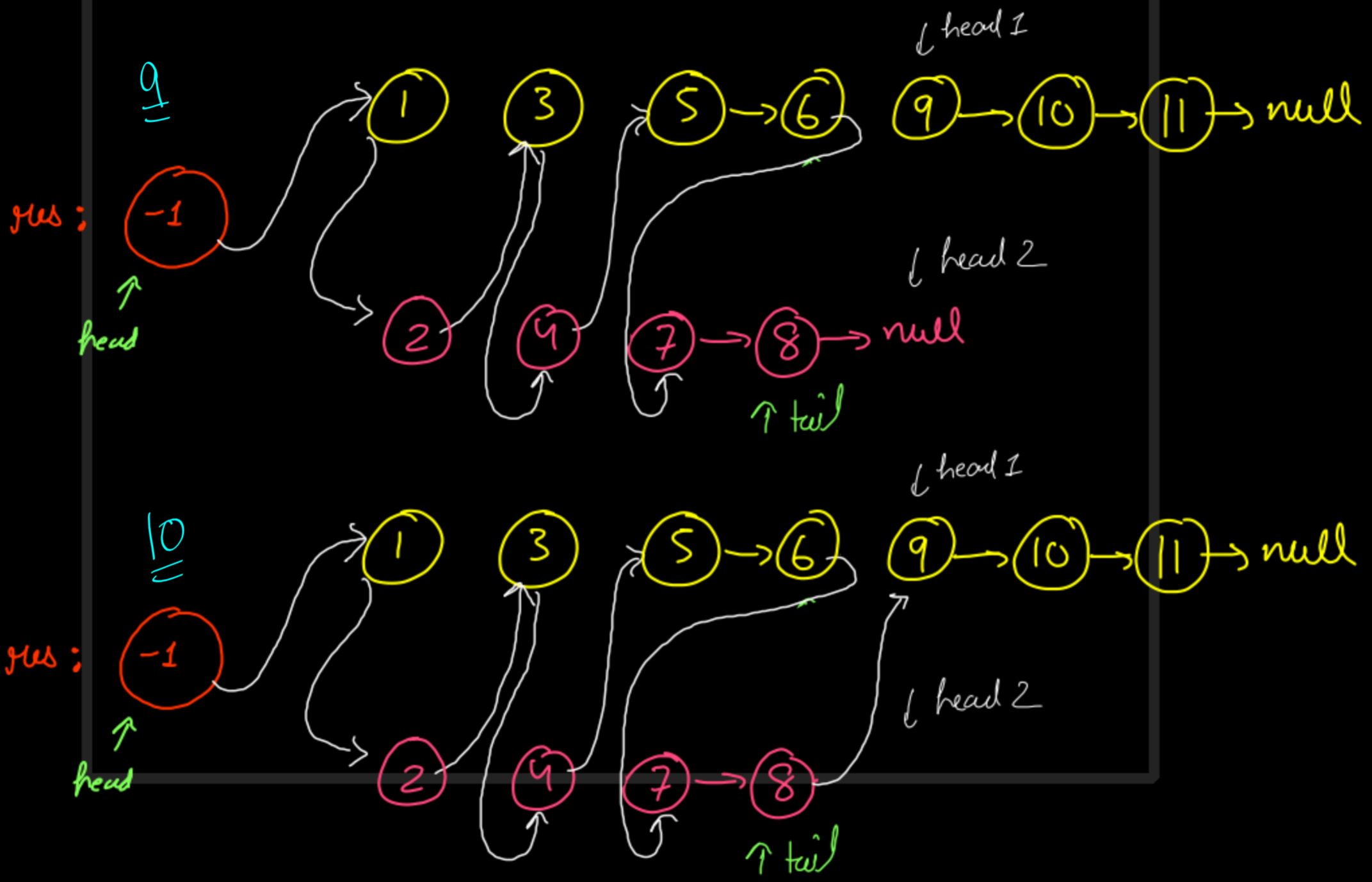



head 1
3
1
3 → 5 → 6 → 9 → 10 → 11 → null
res :
-1
head
head 2
2 → 4 → 7 → 8 → null
tail
4
1
3 → 5 → 6 → 9 → 10 → 11 → null
head 1
tail
res :
-1
head 2
head
2 → 4 → 7 → 8 → null


5
head 1
1
3
5
6
9
10
11
null
res :
-1
head
2
4
7
8
null
head 2
tail
6
head 1
1
3
5
6
9
10
11
null
tail
res :
-1
head
2
4
7
8
null
head 2


7
res :
-1
head
1
3
5
6
9
10
11
null
head 1
tail
2
4
head 2
7
8
null
8
res :
-1
head
1
3
5
6
9
10
11
null
head 1
2
4
head 2
7
8
null
tail

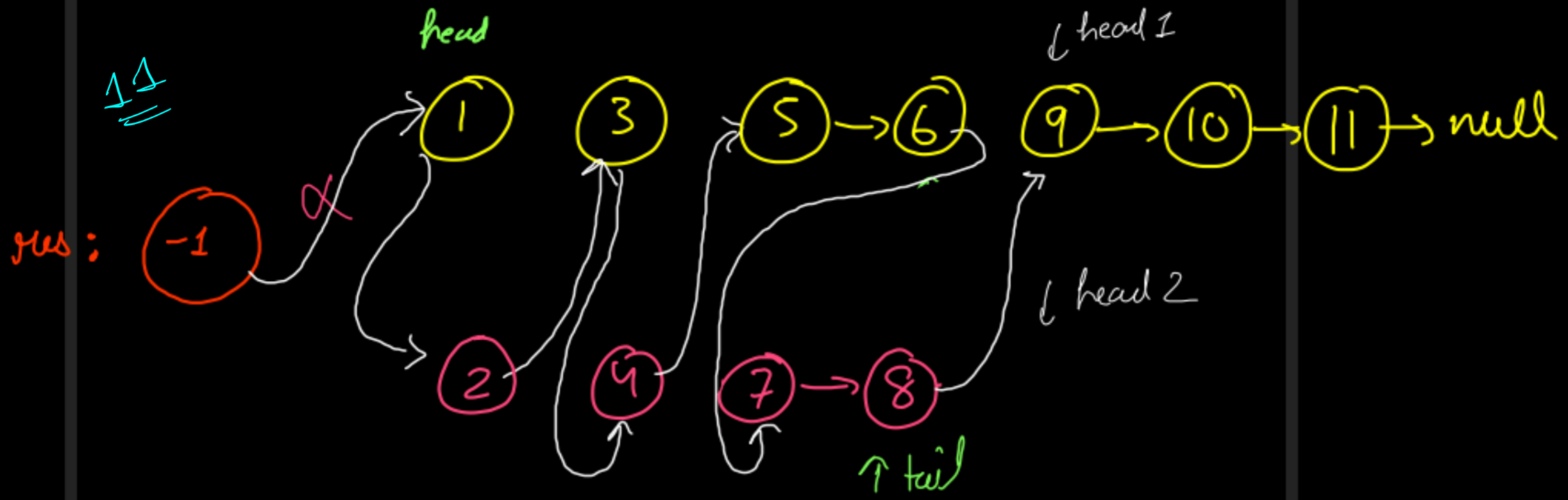
9


head 1
res :
-1
1
3
5
6
9
10
11
null
head
2
4
7
8
null
head 2
tail

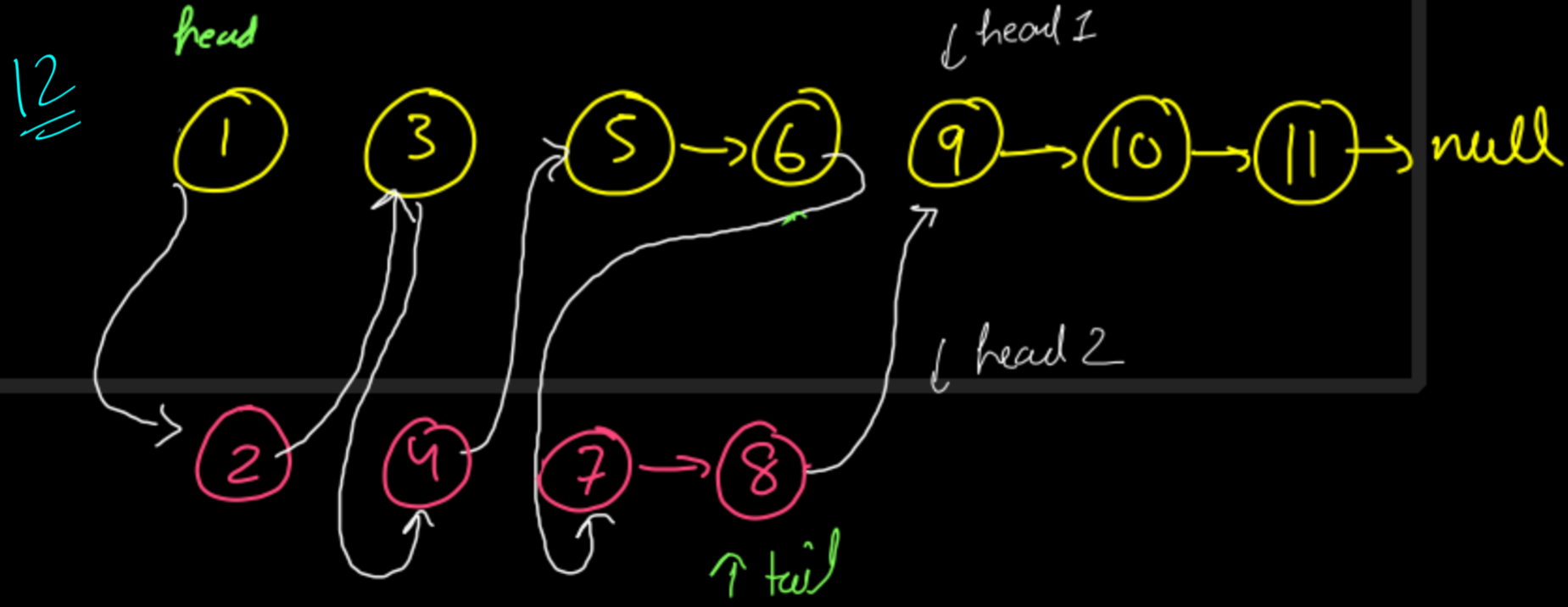
10

head 1
res :
-1
1
3
5
6
9
10
11
null
head
2
4
7
8
head 2
tail

11


head
head 1
head 2
res ;
-1
1
3
5
6
9
10
11
null
2
4
7
8
tail

12


head
head 1
head 2
1
3
5
6
9
10
11
null
2
4
7
8
tail

Merge 2 Sorted Lists Code (GfG)

```
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;
}
```

→ to remove dummy node.

Merge 2 Sorted Lists (Leetcode)

```java
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    ListNode dummy = new ListNode(-1);
    ListNode head = dummy, tail = dummy;

    while(head1 != null && head2 != null) {
       if(head1.val < head2.val) {
           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;
}
```

Merge K Sorted Linked Lists (Leetcode Hard)

n → 1 → 4 → 7

n → 3 → 6 → 9

n → 2 → 5 → 10

↑ → 1 → 6 → 10

$N = n * k$

to find min ⇒ k elements

O(k) time

$TC = O(\sum_{1}^{N} * k)$

$TC = O(nk^2)$

Same approach as prev

1 → 1 → 2 → 3 → 4 → 5 → 6 → 6 → 7 → 9 → 10 → 10

Code for Merge K Sorted Lists

```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;
}
```

```java
public ListNode mergeKLists(ListNode[] lists) {

    if(lists.length == 0) {
        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;
}
```

Constraints:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` is sorted in ~~ascending order~~.
- The sum of `lists[i].length` won't exceed `10^4`.

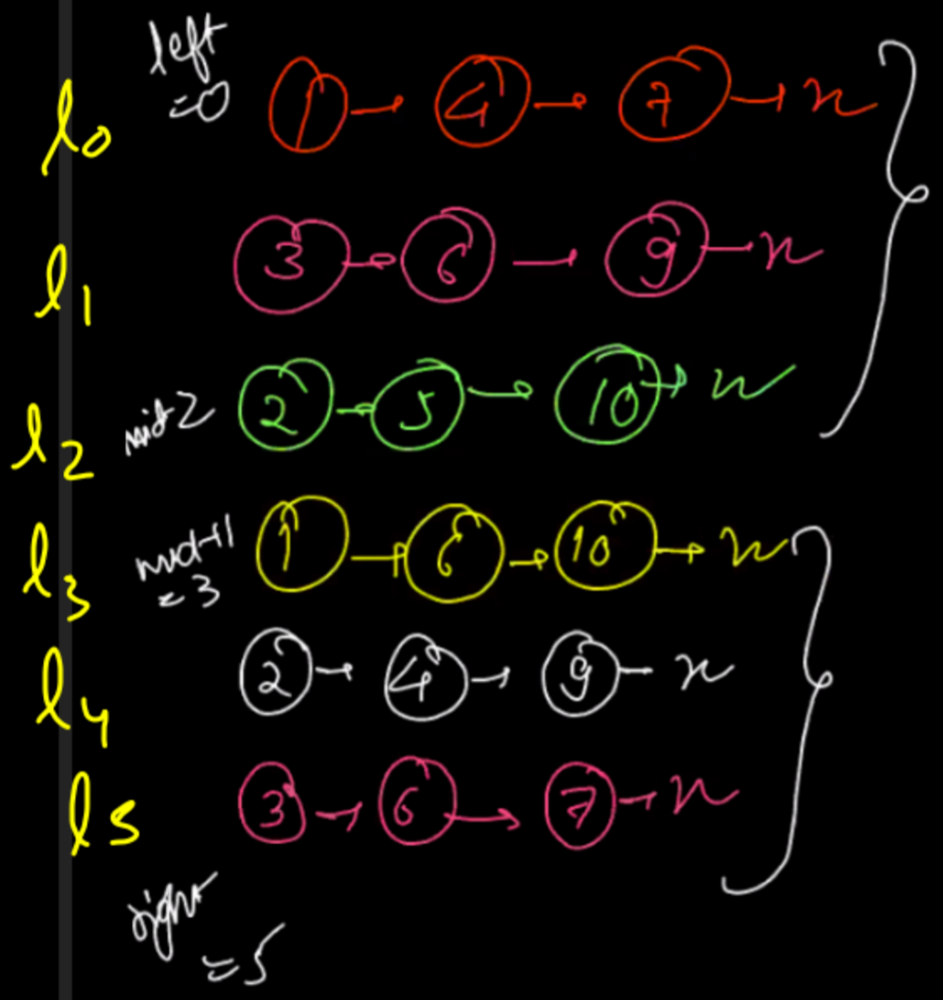
This constraint means that capital N will not exceed 10^4 & TC = N * k

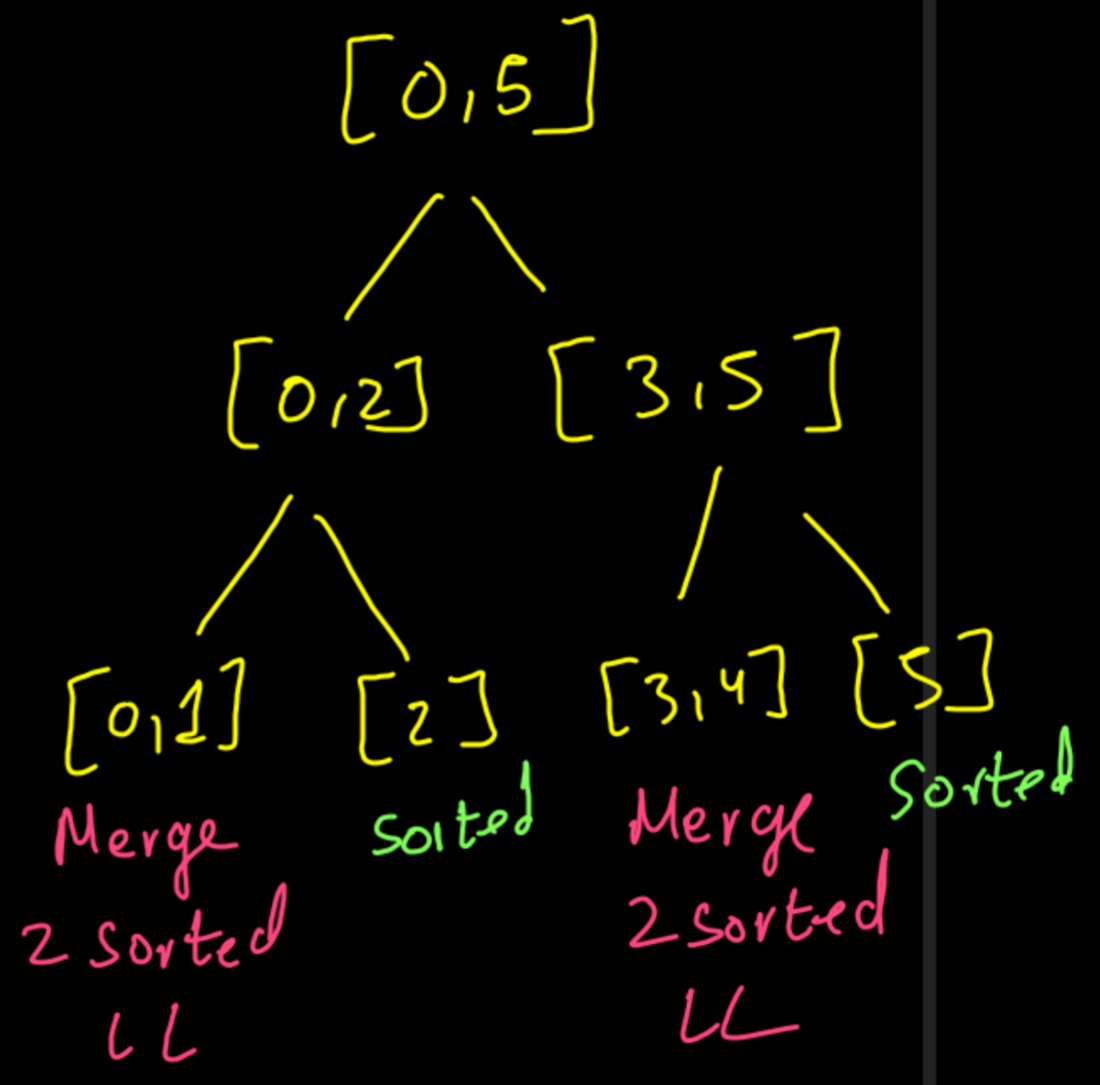
$\Rightarrow$ TC = $10^4 * 10^4 = 10^8$ (Hence our brute force solution worked)

** Use priority queue for comparing nodes instead of loop.

I implement Comparator as we will make PQueue of List Nodes

Best optimized Approach {Divide & Conquer}





Code Using Divide & Conquer

```java
public ListNode mergeKLists(ListNode[] lists) {
    return mergeKLists(lists,0,lists.length-1);
}

public ListNode mergeKLists(ListNode[] lists,int lo, int hi) {

    if(lo > hi) {
        return null;
    }

    if(lo == hi) {
        return lists[lo];
    }

    int mid = lo + (hi-lo)/2;
    ListNode leftHalf = mergeKLists(lists,lo,mid);
    ListNode rightHalf = mergeKLists(lists,mid + 1,hi);

    return mergeTwoLists(leftHalf,rightHalf);
}
```

```java
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    ListNode dummy = new ListNode(-1);
    ListNode head = dummy, tail = dummy;

    while(head1 != null && head2 != null) {
       if(head1.val < head2.val) {
           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;
}
```

Time Complexity (Divide & Conq)

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

$$2\,T(k/2) = \left(2T(k/4) + \frac{k}{4} * n\right) 2$$

$$\vdots$$

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

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

$$TC = O(nk \log_2 k)$$

Codes for Merge Sort of LL

```java
public ListNode sortList(ListNode head) {
    return helper(head);
}

public ListNode helper(ListNode head) {

    if(head == null || head.next == null) {
        return head;
    }

    ListNode mid = getMiddle(head);
    ListNode list2 = mid.next;
    mid.next = null;

    ListNode list1 = head;

    ListNode leftSortedList = helper(list1);
    ListNode rightSortedList = helper(list2);

  return  mergeTwoLists(leftSortedList, rightSortedList);
}
```

```java
public ListNode getMiddle(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    ListNode prev = null;

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

    if(fast == null) return prev; //this is even case
    return slow;
}
```

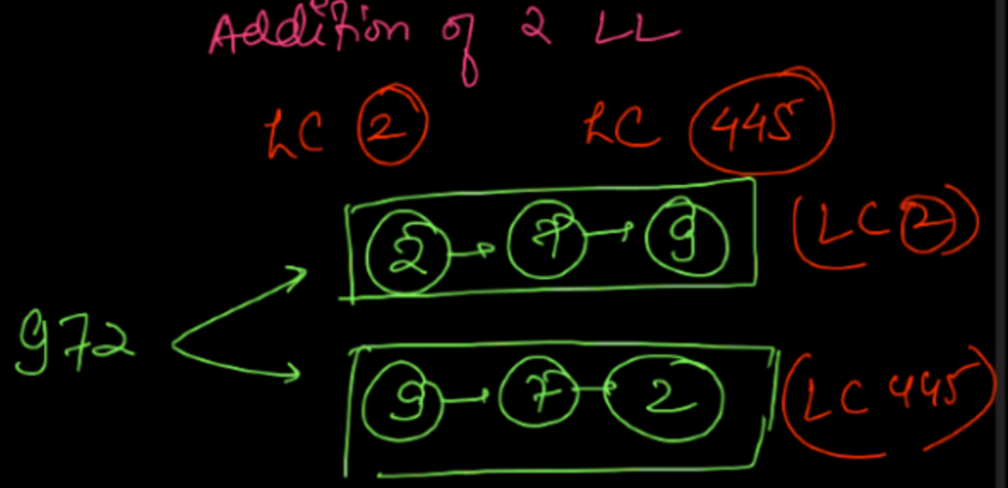
```java
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    ListNode dummy = new ListNode(-1);
    ListNode head = dummy, tail = dummy;

    while(head1 != null && head2 != null) {
       if(head1.val < head2.val) {
           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;
}
```

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

merge Sort on LL ↘ T(N) → O(nlogn)
↘ Space → inplace ⭐

However Recursion call stack space is $O(\log_2 N)$

Add 2 Linked Lists



boost library C++

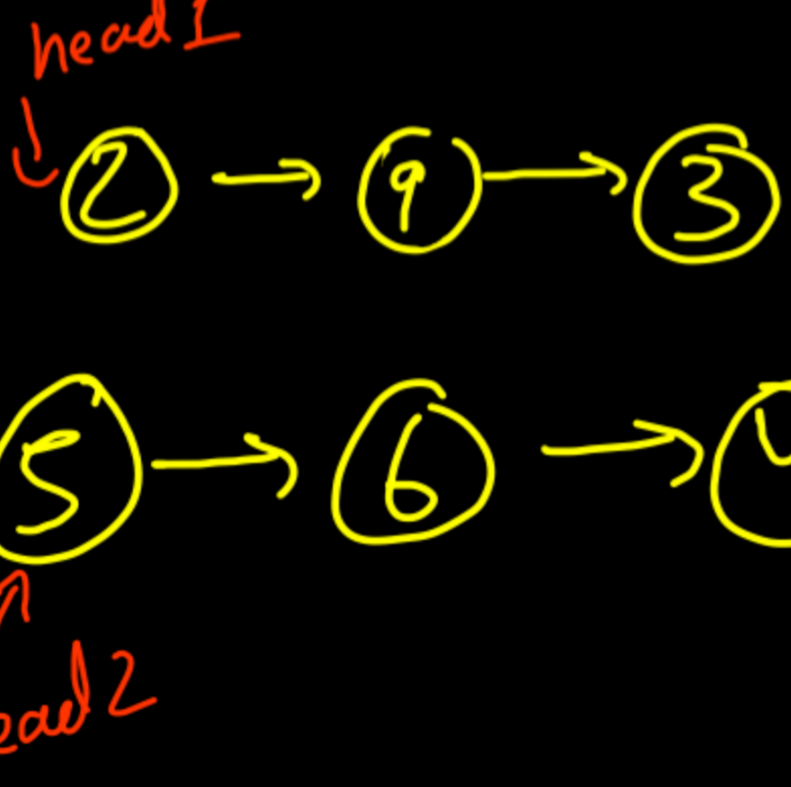
java has big Integer library

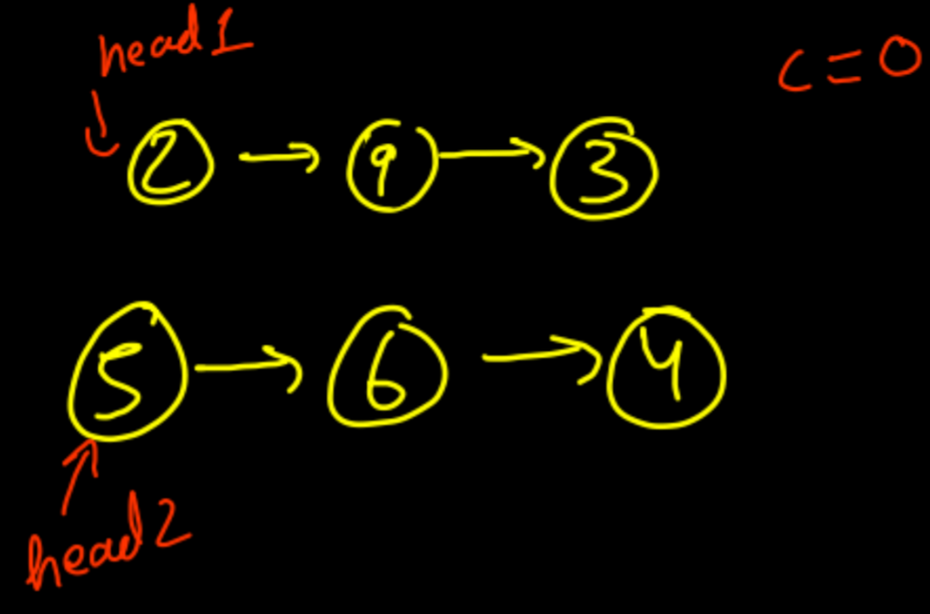
Python handles big integers easily by internally implementing LL.

Leetcode Ques 2

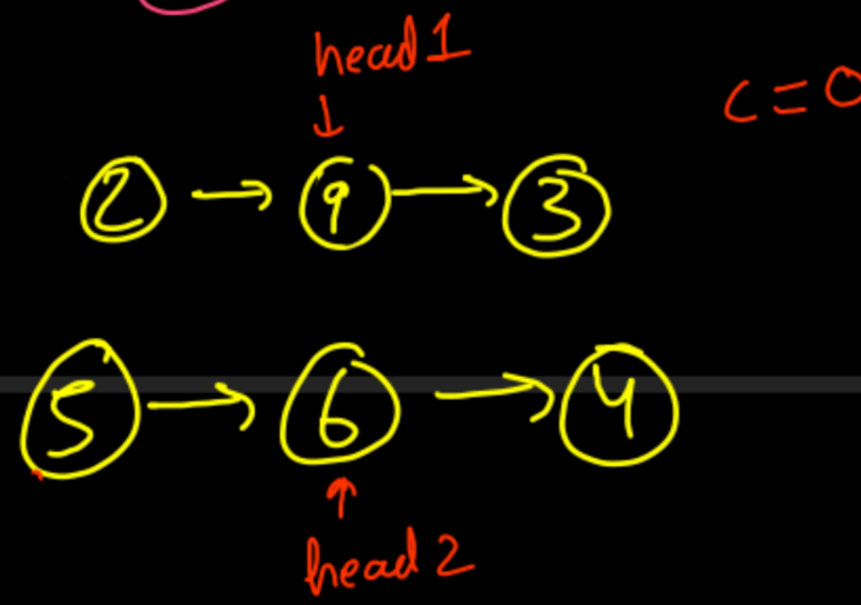
No = 342 LL: 2 → 9 → 3

No = 465 LL: 5 → 6 → 4



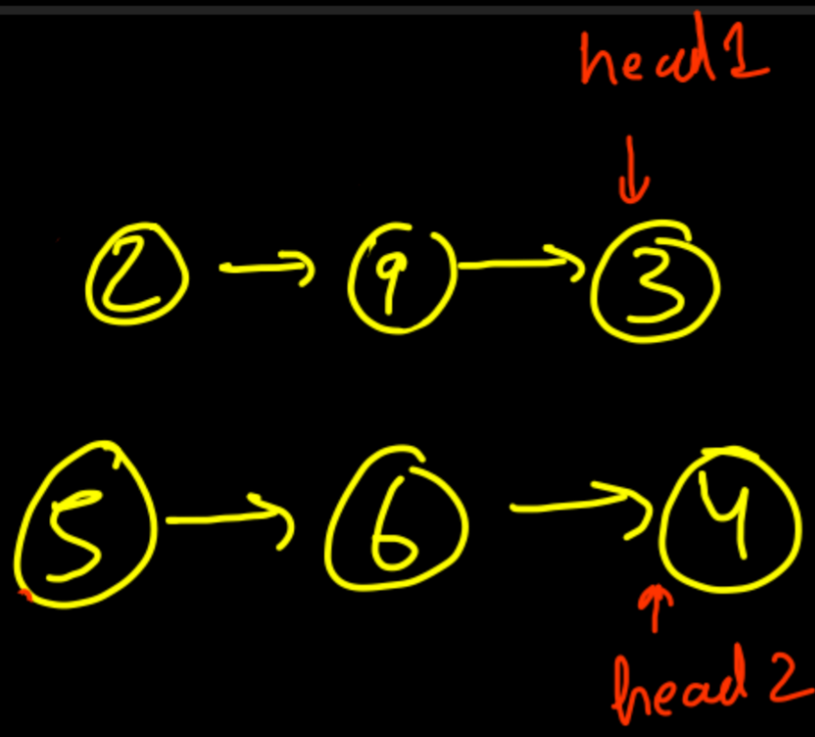


res: 7 & c = 0



9 + 6 + 0 = 15 o/p 10 = 5

res: 7 → 5 & c = 15/10 = 1



3 + 4 + 1 = 8

res: 7 → 5 → 8 & ca = 0

if one LL becomes empty, work will continue. & same as in add^n of arrays

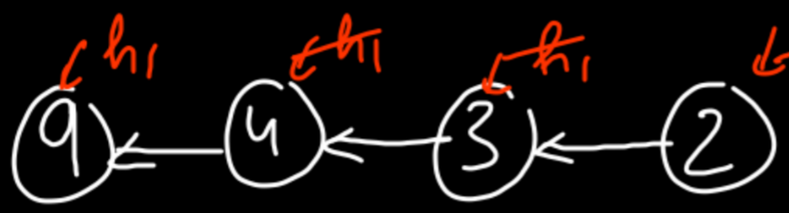
Leetcode : 2 (Solution code)

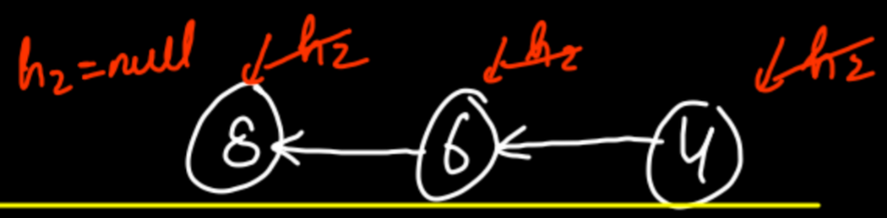
```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

    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;
}
```

Leetcode : 445 Solution

```java
class Solution {
    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);
    }
```

Subtraction in Linked List



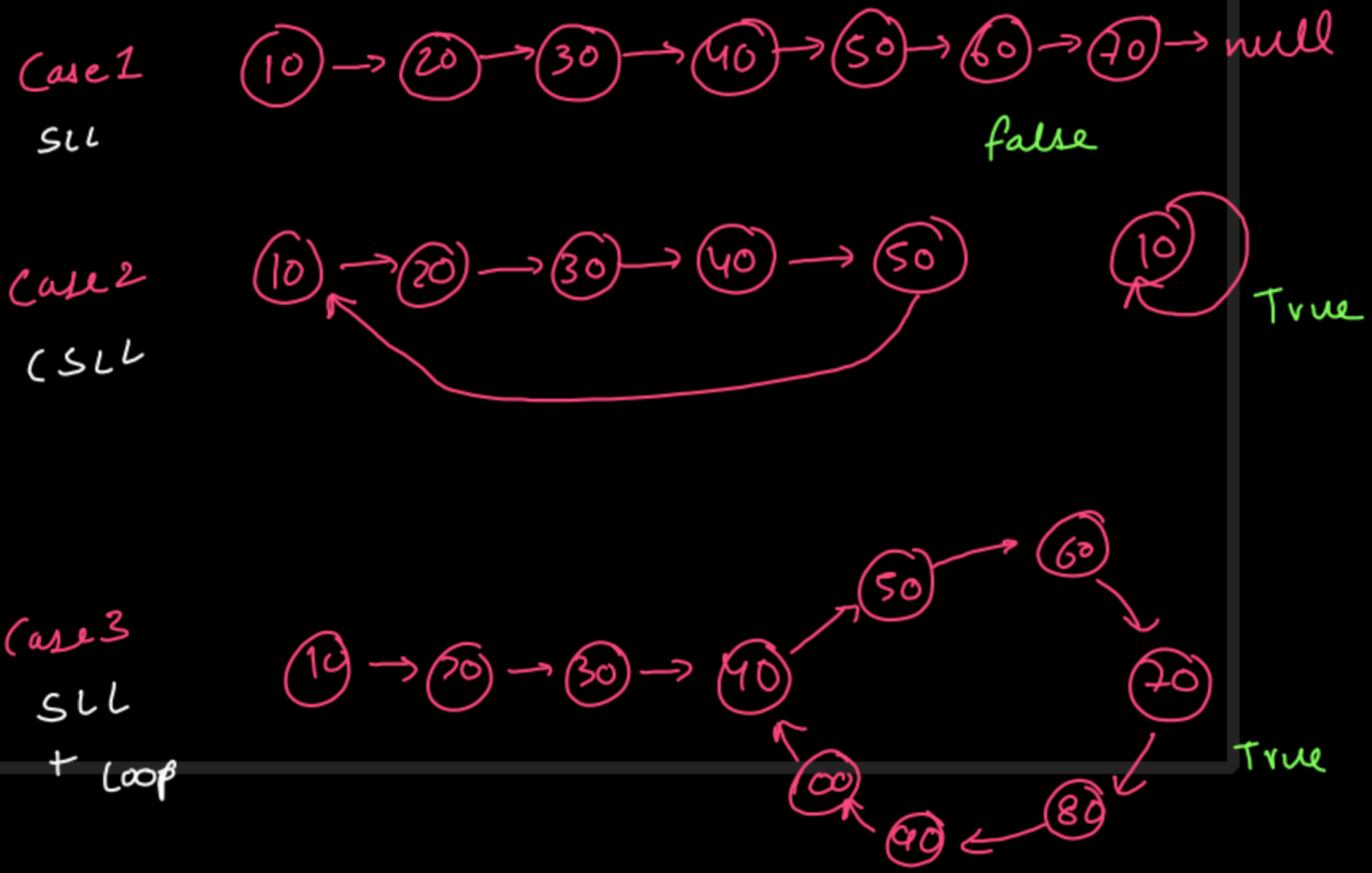


borrow = ~~0~~ ~~-1~~
-1

while(h1 != null)

Code in HW

Floyd's Cycle Detection (LC 141)

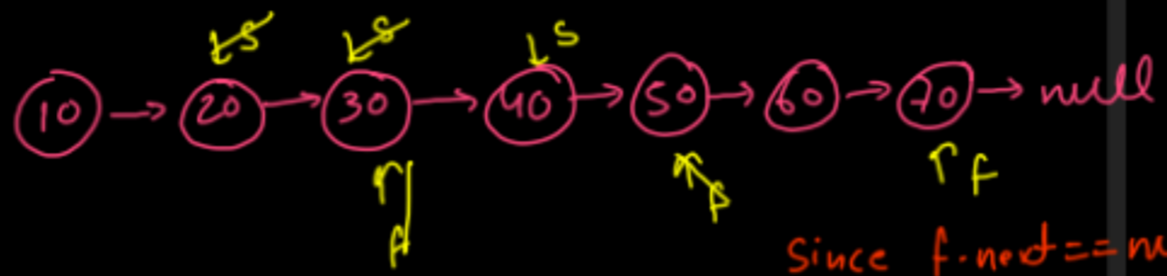


First Approach

Store the traversed nodes in a HashMap or HashSet & if the node is visited again, cycle is detected

Floyd's Cycle Detection (Two Pointer)

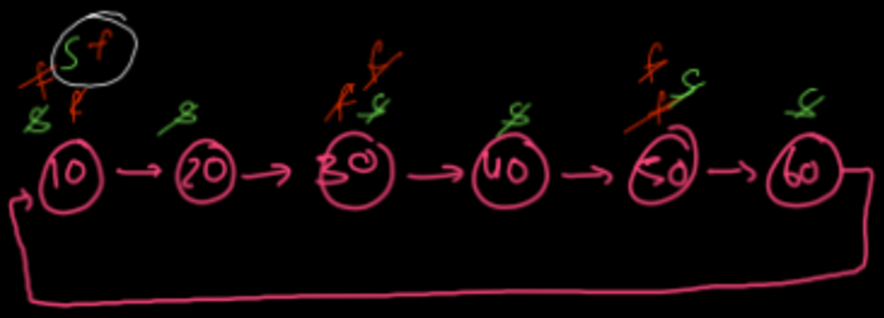
Case 1



Since f.next == null
No Loop detected

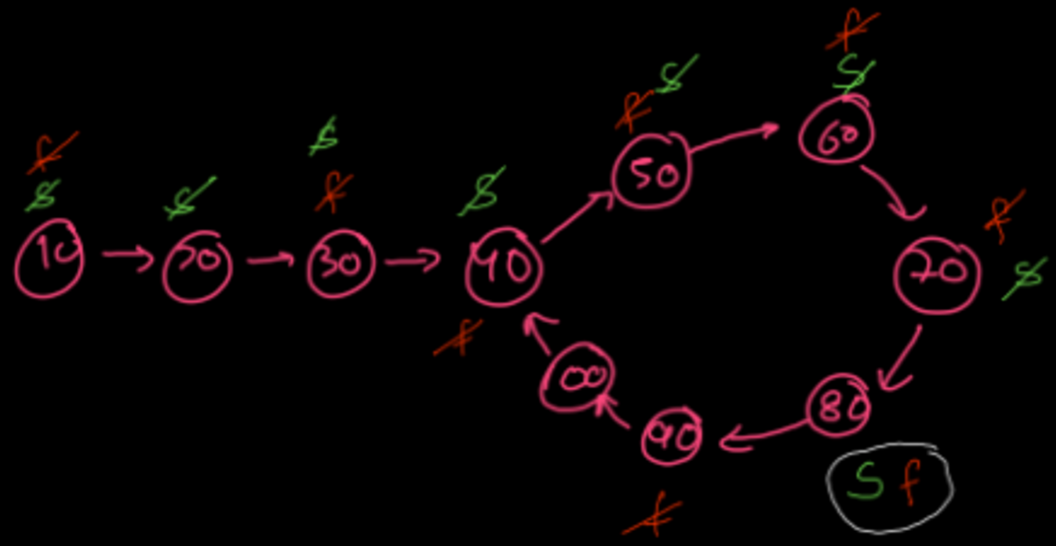
Case: 2

CLL



Since fast & slow meet
a cycle is detected.

Case3
SLL
+ Loop



Since slow & fast meet again
a loop is detected

Corner Case :

Again, slow & fast meet again

Corner case ask to interviewer

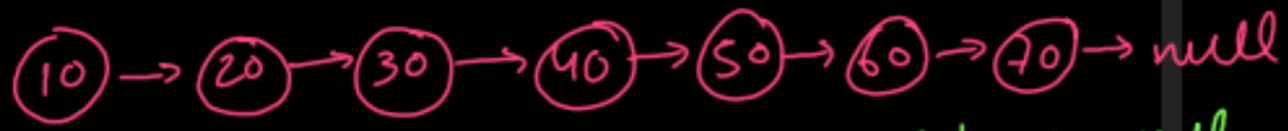
if (head == = null) -> return true or false

Code for floyds' cycle Detection

```java
public boolean hasCycle(ListNode head) {
    if(head == null || head.next == null) return false;

    ListNode slow = head;
    ListNode fast = head;

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

        if(slow == fast) return true;
    }

    return false;
}
```

Starting Point of Loop (LC 142)

Case 1
SLL



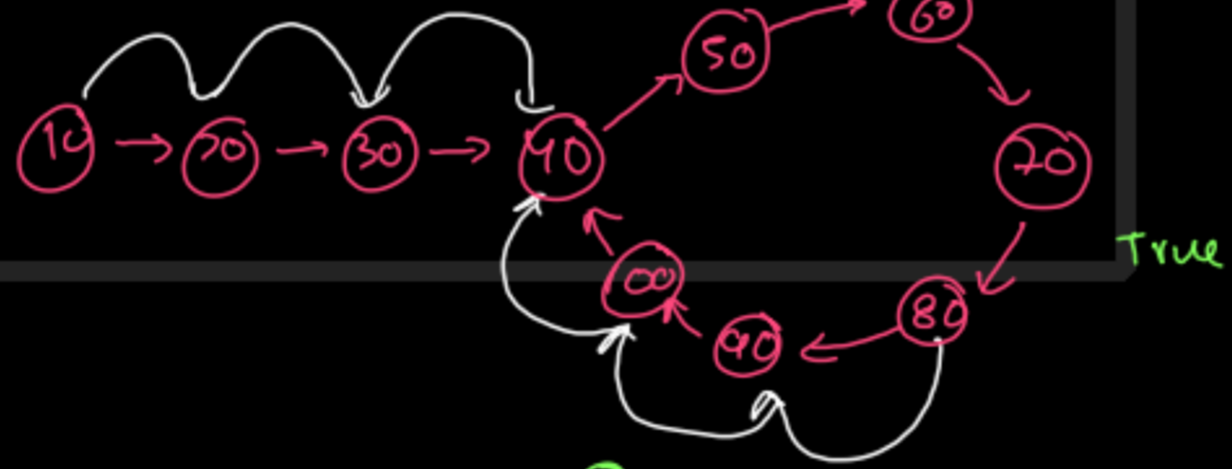
return null

Case 2
CSLL

return (10) head

return (10) head

Case 3
SLL
+ Loop

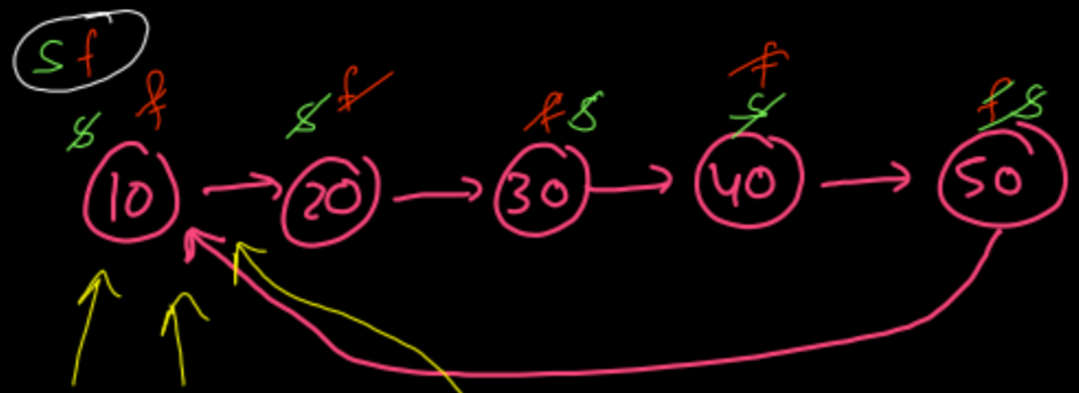


return (80)

+ Starting point of the loop is at the same distance from meeting point & head & we will use this obs

Case 2

CSLL



head Starting point of loop

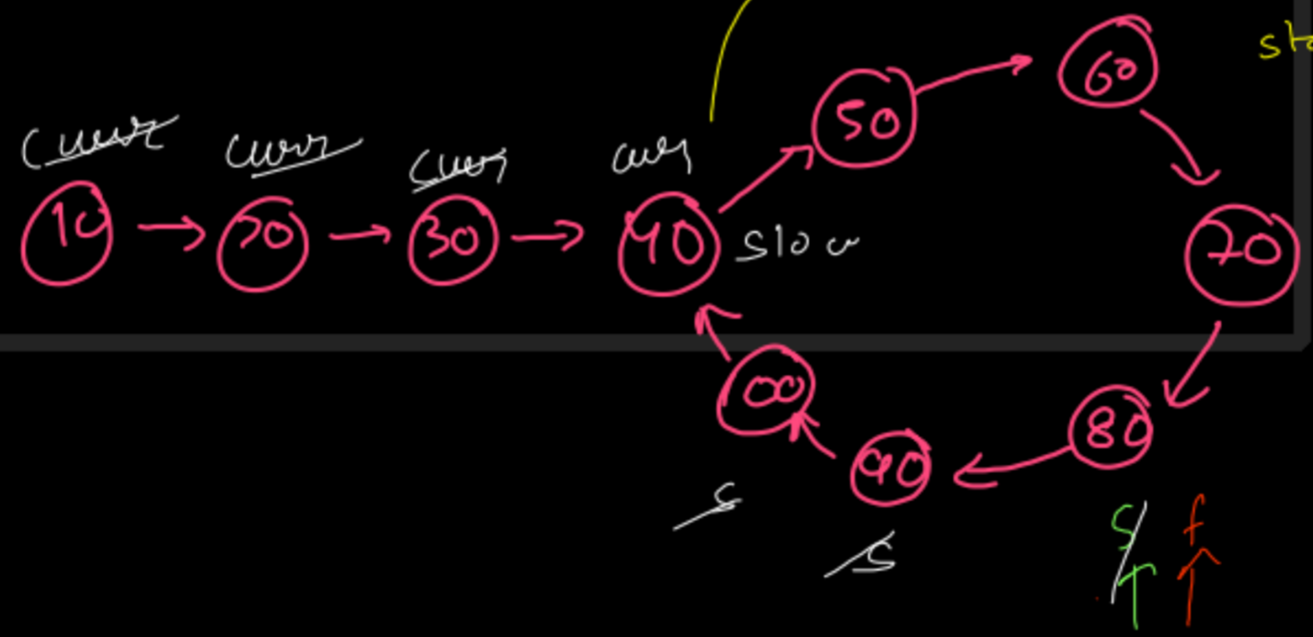
Meeting Point of Loop

Case 3

SLL

+ Loop

curr = slow (found the starting point)



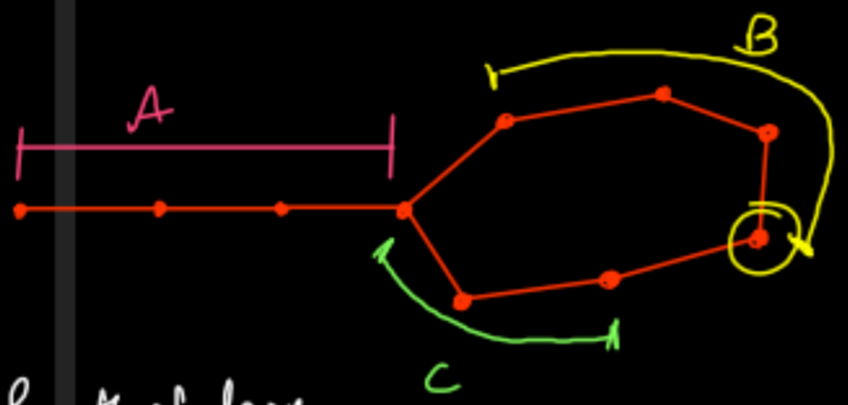
True

Code for Starting Pt of a Loop (LC 142)

```java
public ListNode detectCycle(ListNode head) {

    ListNode slow = head;
    ListNode fast = head;

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

        if(slow == fast) break;
    }

    if(fast == null || fast.next == null) return null;

    ListNode curr1 = head;
    ListNode curr2 = slow;

    while(curr1 != curr2) {
        curr1 = curr1.next;
        curr2 = curr2.next;
    }

    return curr1;
}
```

O(N)

Mathematical Proof



Length of loop = $B+C$

Length of tail = A

Slow = 1 node / itr

fast = 2 nodes / itr

Total time taken by slow = Total time taken by fast

$d_{slow} = d_{fast}$

$s_s \qquad s_f$

$d_{fast} = A + (B+C)j + B$

↳ j loops travelled times

$d_{slow} = A + (B+C)i + B$

↳ i times loops travelled

{ $i=0$ because slow does not travel any loop in curr example }

$$\frac{d_{slow}}{s_s} = \frac{d_{fast}}{s_f}$$

$$\frac{A+(B+C)\hat{i}+B}{S_s} = \frac{A+(B+C)\hat{j}+B}{S_f}$$

relative speed of fast wrt slow (S_f)

$$\frac{S_f}{S_s}\{A+(B+C)\hat{i}+B\} = A+(B+C)\hat{j}+B$$

$$rA + r(B+C)\hat{i} + rB = A+(B+C)\hat{j}+B$$

$$rA + rB - A - B = (B+C)\{\hat{j} - r\hat{i}\}$$

$$(r-1)(A+B) = (B+C)(\hat{j}-r\hat{i})$$

$$A+B = (B+C)\left\{\frac{\hat{j}-r\hat{i}}{r-1}\right\}$$

$$\frac{\hat{j}-r\hat{i}}{r-1} = k$$

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

Tail — Some Cycles

A = some cycles − fail shorting of B

Hence Proved

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

Tail — some cycles

This can also be understood as

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

k' = some no of cycles

↝ uske baad C chalega

Simple Proof (Explanation for Interviews)

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

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

{ $S_{fast} = 2 * S_{slow}$ }

$d_{fast} = * 2 dist_{slow}$

$A + cycle(j) + B = 2(A + cycle(i) + B)$

$A + cj + B = 2A + c2i + 2B$ → cycle ko again c likh de

$$A = c(j - 2i) - B$$

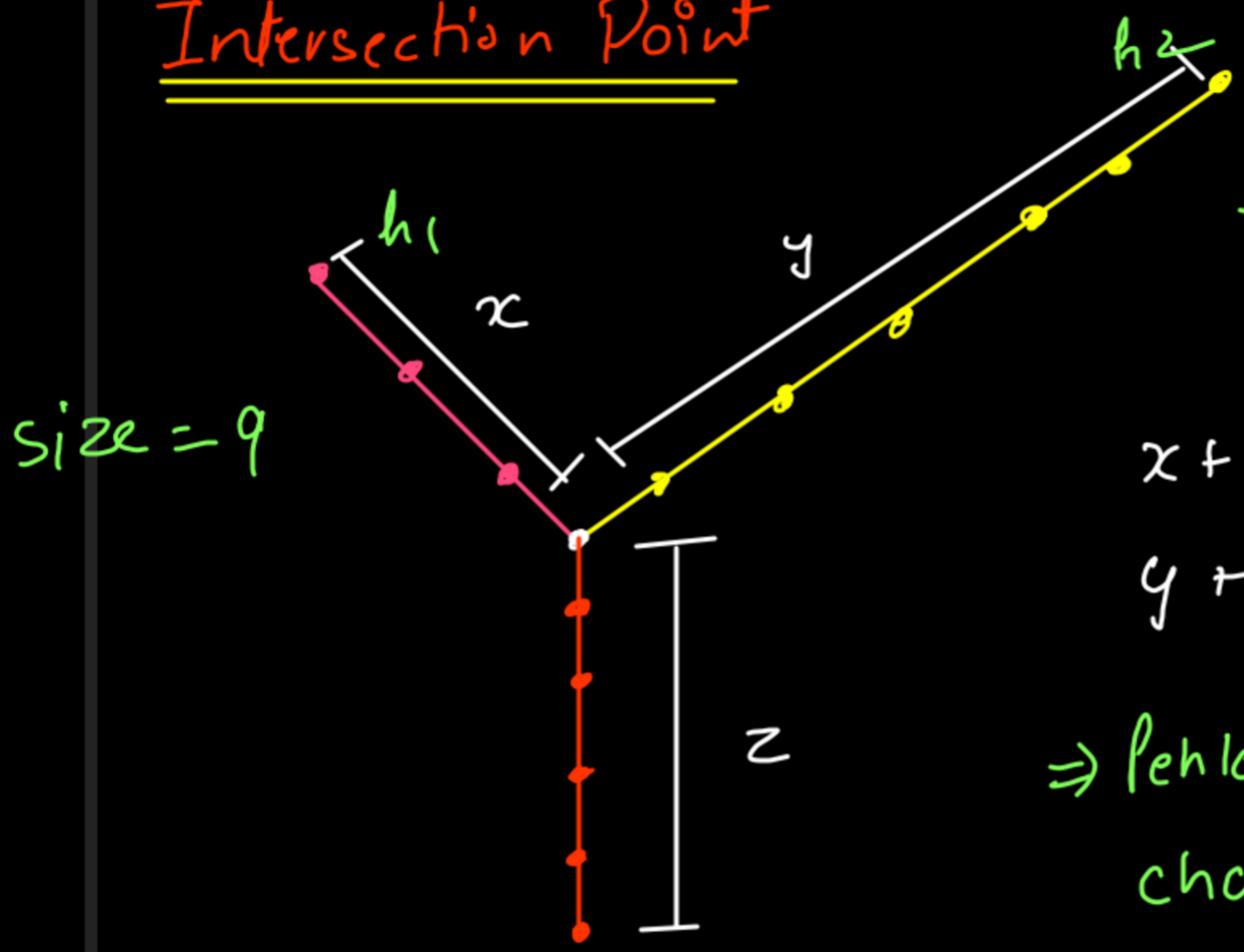
Remove Loop ⇒ HW

Length of the Loop (GfG)

```
static int countNodesinLoop(Node head)
{

    Node slow = head;
    Node fast = head;

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

        if(slow == fast) break;
    }

    if(fast == null || fast.next == null) return 0;

    Node curr1 = head;
    Node curr2 = slow;

    while(curr1 != curr2) {
        curr1 = curr1.next;
        curr2 = curr2.next;
    }

    Node p = curr1;
    int dist = 1;
    p = p.next;
    while(p != curr1) {
        p = p.next;
        dist +=1;
    }

    return dist;
}
```

find the starting point of the loop & then traverse the loop.

Intersection Point



size = 9

size = 12

diff = 12 - 9
= 3

$x + y = 9$

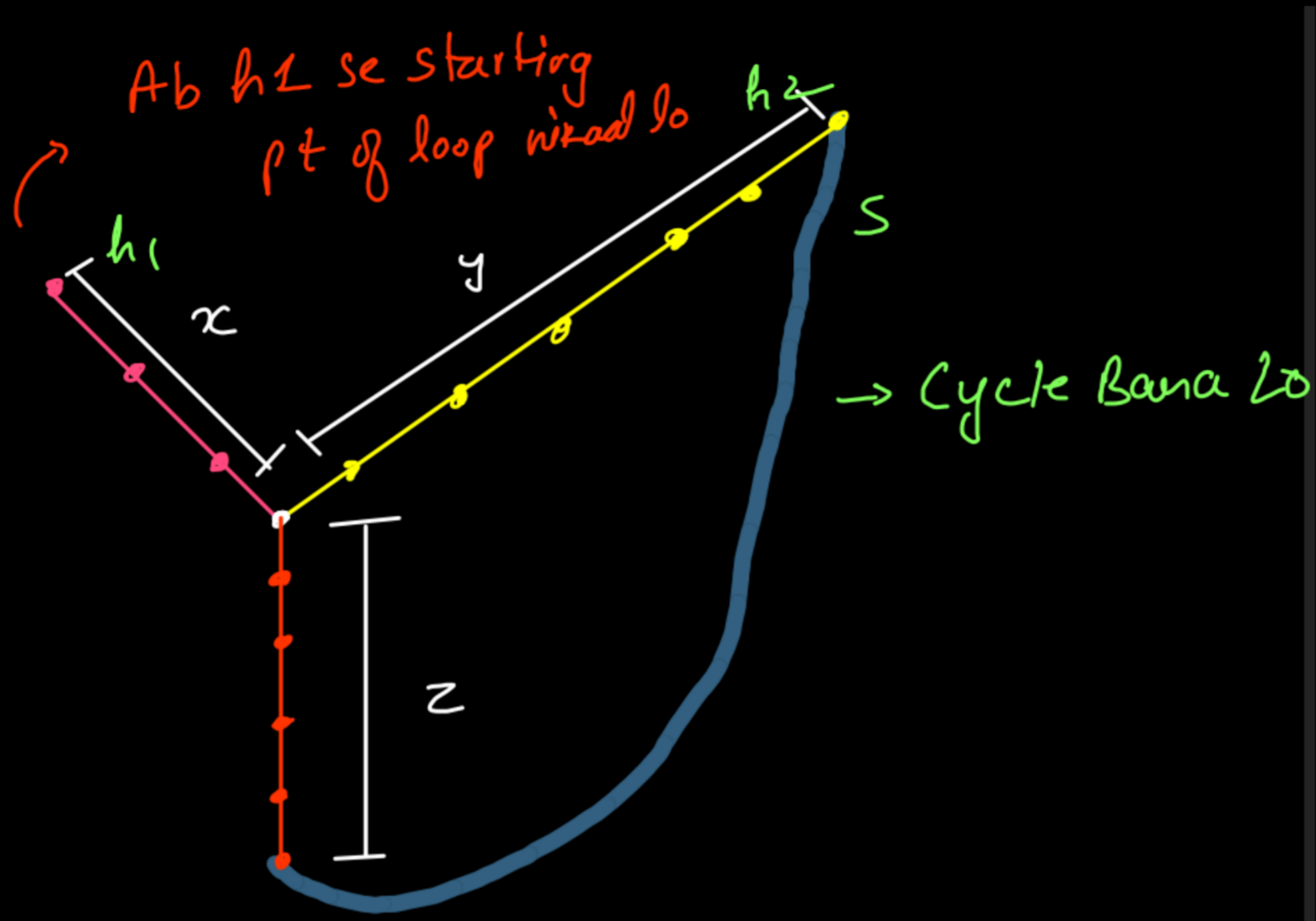
$y + z = 12$

⇒ Pehle h_2 ko 3 baar chala do

⇒ Jab tak dono barabar nahi hote dono ko chalao

Intersection Point

```java
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;
}
```


Ab h1 se starting
pt of loop nikaal lo
h2
h1
x
y
z
s
→ Cycle Bana lo

Using floyd's Cycle

```
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;
}
```

Partition List (Leetcode 86)

① Partition on head is correct

② Partition in LL will be Stable

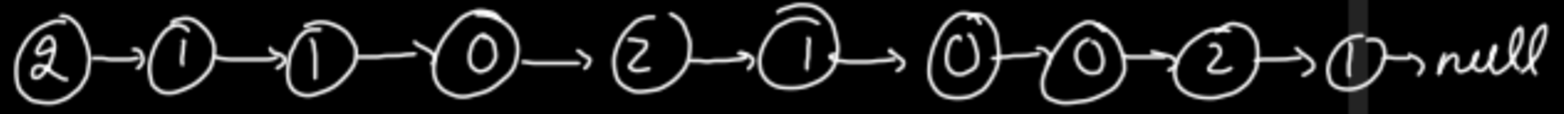
③ Quick Sort (obviously) gets the element at its sorted position.

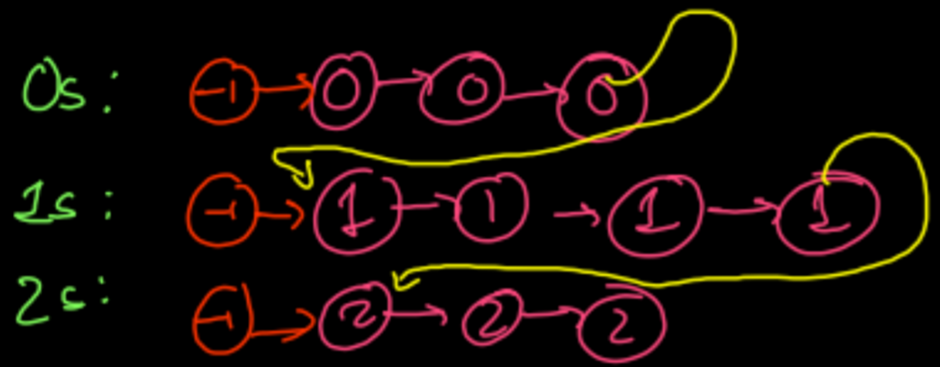
for quick Sort, partition wrt head.

Code for Partition List

```
class Solution {
    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; //join less than and greater than region
        rTail.next = null; //overall tail
        return lHead.next; //return head excluding head

    }
```

Sort 012



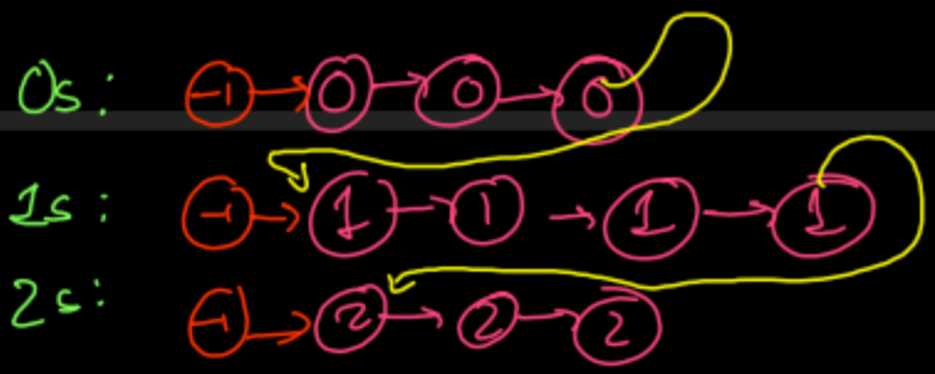


0Tn = 1Hn

1Tn = 2Hn

2Tn = null

To avoid corner case:



0Tn = 1Hn 3rd

1Tn = 2Hn 2nd

2Tn = null 1st establish this link

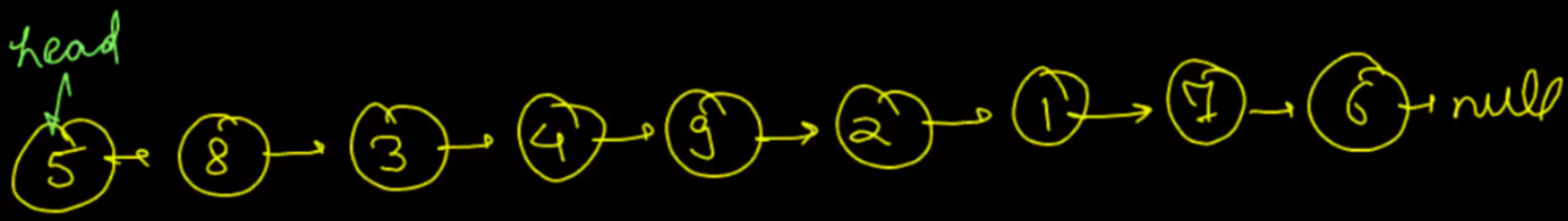
follow this order

Code for Sort 012

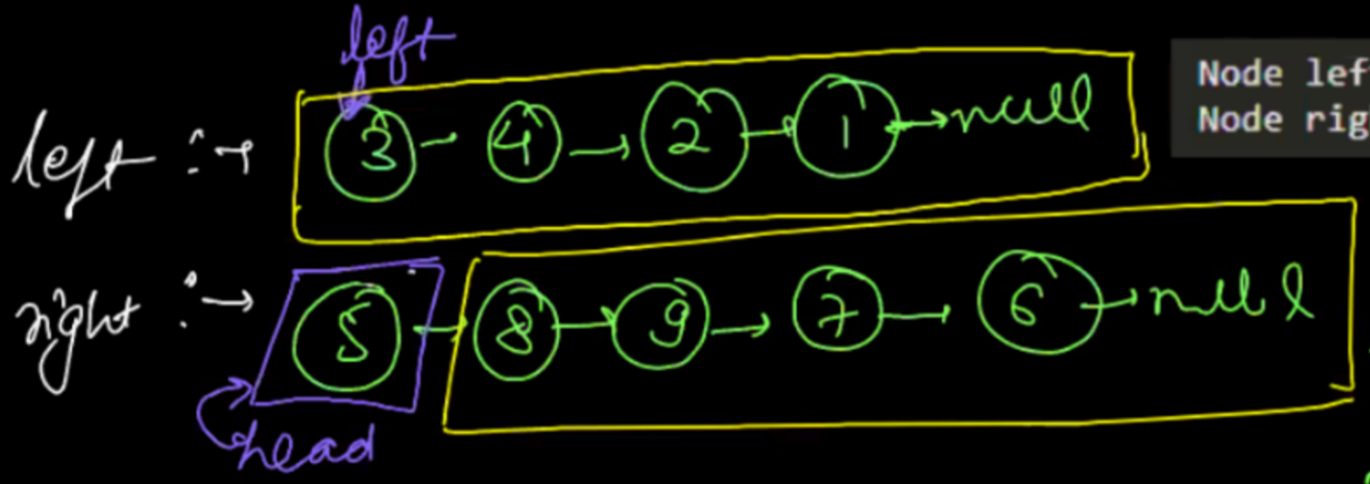
```
//Function to sort a linked list of 0s, 1s and 2s.
static Node segregate(Node head)
{
    Node zeroHead = new Node(-1);
    Node zeroTail = zeroHead;
    Node oneHead = new Node(-1);
    Node oneTail = oneHead;
    Node twoHead = new Node(-1);
    Node twoTail = twoHead;

    while(head != null) {
        if(head.data  == 0) {
            zeroTail.next = head;
            zeroTail = head;
        } else if(head.data == 1) {
            oneTail.next = head;
            oneTail = head;
        } else {
            twoTail.next = head;
            twoTail = head;
        }
        head = head.next;
    }

    twoTail.next = null;
    oneTail.next = twoHead.next;
    zeroTail.next = oneHead.next;

    return zeroHead.next;
}
```

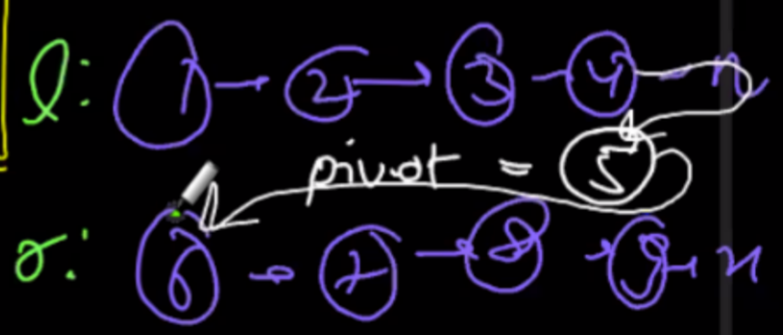
Quick Sort Linked List

Sort it using Quick sort (148) (3421) (5) (8976)




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





```java
public ListNode sortList(ListNode head) {

    if(head == null || head.next  == null) return head;

    ListNode left = sortList(partition(head,head.val));
    ListNode right = sortList(head.next);
    head.next = right;
   if(left == null) return head;

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

}
```

```java
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;
}
```

Passes all the test cases but gives

TLE {Worst case $O(N^2)$ QuickSort}

{Randomise: Swap head with any random node}

Randomized QuickSort Code

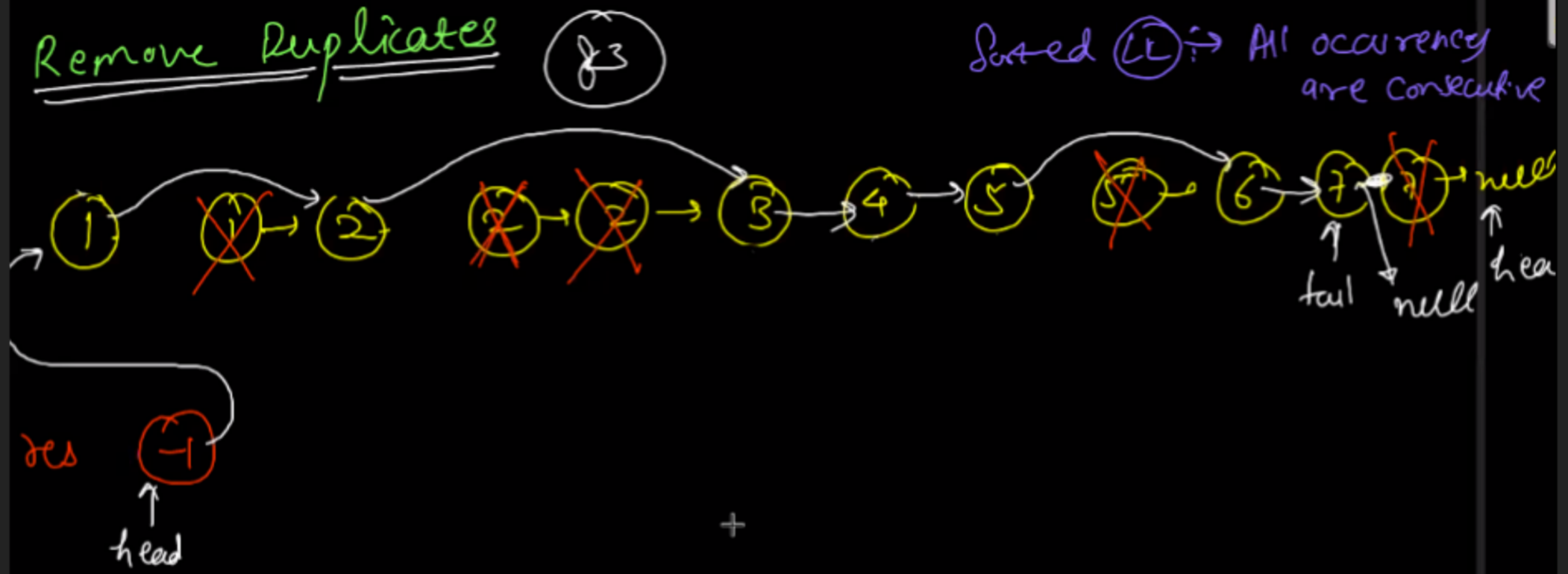
```java
public void randomize(ListNode head) {
    int size = 0;
    ListNode curr = head;

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

    int randomIdx = (new Random()).nextInt(size); //0 to size-1

    curr = head;
    while(randomIdx-- > 0) {
        curr = curr.next;
    }

    int temp = head.val;
    head.val = curr.val;
    curr.val = temp;
}
```

```java
public ListNode sortList(ListNode head) {

    if(head == null || head.next  == null) return head;

    randomize(head);
    ListNode left = sortList(partition(head,head.val));
    ListNode right = sortList(head.next);
    head.next = right;
   if(left == null) return head;

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

}
```

→ this is change

Remove Duplicates {from Sorted List}

Remove Duplicates



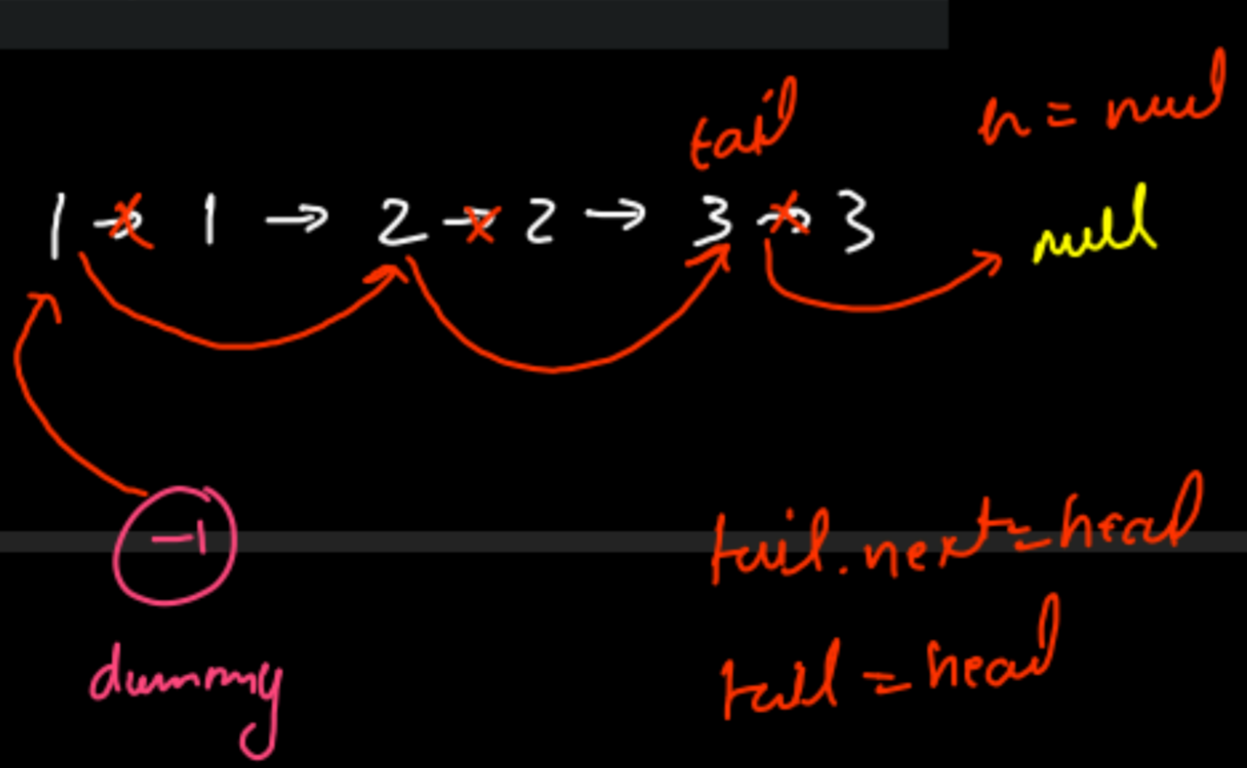
O(1) extra space

My Code {LC 83} [This is without dummy Node]

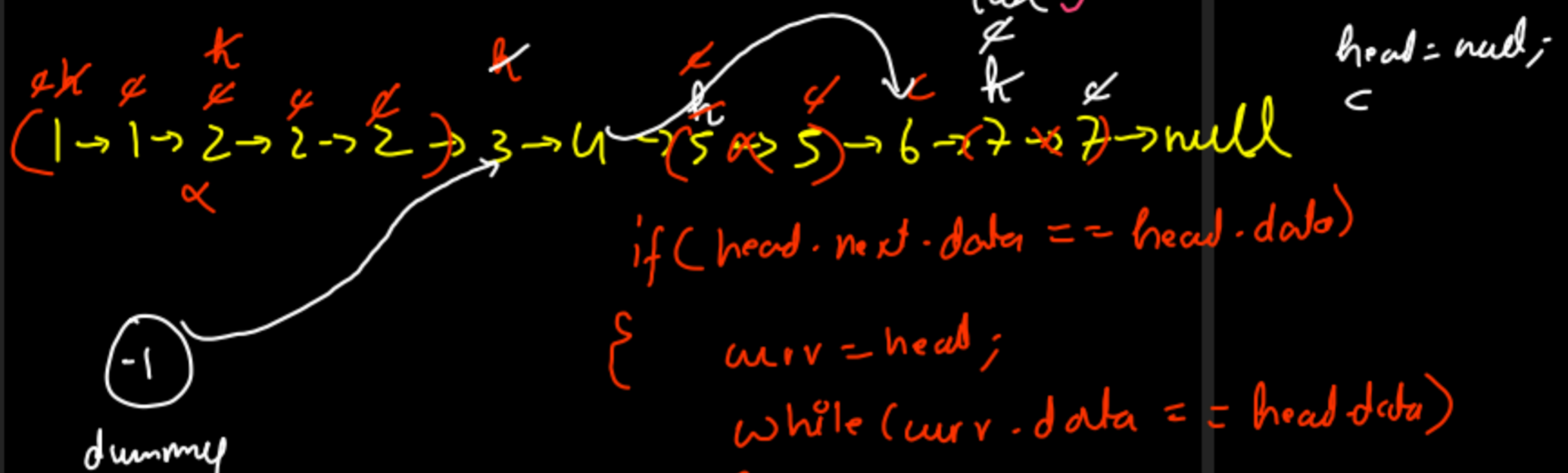
```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null) return head;

        while(head != null && head.next != null && head.val == head.next.val) {
            head = head.next;
        }

        ListNode curr = head.next;

        while(curr != null && curr.next != null) {
            if(curr.val == curr.next.val) {
                curr.next = curr.next.next;
            } else {
                curr = curr.next;
            }
        }

        return head;
    }
}
```

Using dummy Node

```java
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;
}
```



Remove All Duplicates (LL 82)




```
if(head.next.data == head.data)
{   curr = head;
    while(curr.data == head.data)
    {  curr = curr.next;
    }
    head = curr;
}
else {
    tail.next = head;
    tail = head;
}
```

My code without Using Dummy Node

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {

        if(head == null || head.next == null) return head;

        int curr = -200; //invalid val acc to constraints

        while(head != null && head.next != null) {
            if(head.val == head.next.val) {
                curr = head.val;
                head = head.next;
            } else if(head.val == curr) {
                head = head.next;
            } else {
                break;
            }
        }

        if(head.next == null && head.val == curr) return null;

        if(head == null || head.next == null) return head;

        curr = -200;
        ListNode p = head.next;
        ListNode prev = head;

        while(p.next != null) {
            if(p.val == p.next.val) {
                curr = p.val;
                p.next = p.next.next;
            } else if(p.val == curr) {
                prev.next = p.next;
                p = prev.next;
                curr = -200;
            } else {
                prev = p;
                p = p.next;
            }
        }

        if(p.val == curr) prev.next = null;

        return head;

    }
}
```

O(N)

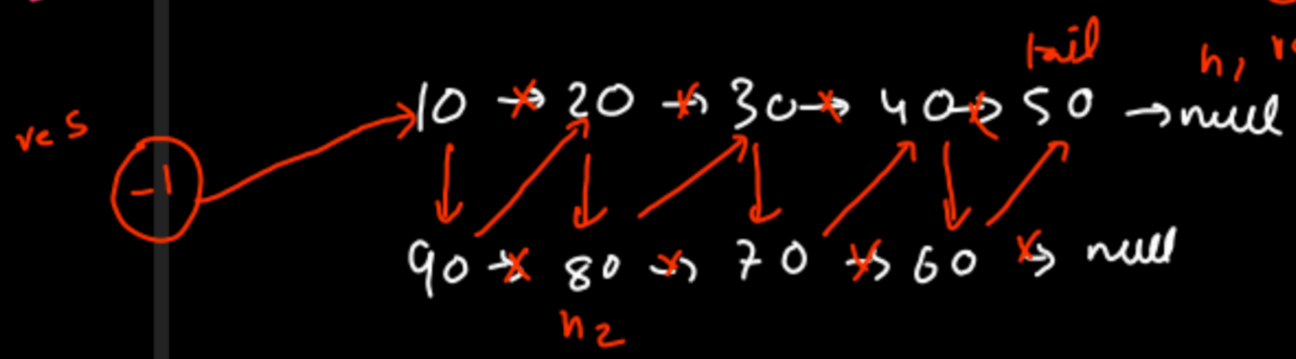
Code Using Dummmy Node -

```java
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) {
            tail.next = head;
            tail = head;
            head = head.next;
        } else {
            ListNode curr = head;
            while(curr != null && curr.val == head.val) {
                curr = curr.next;
            }
            head = curr;
        }
    }

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

O(N)

fold a LL {Reorder List LC 143}

Input: $(10 \to 20 \to 30 \to 40 \to 50) \to (60 \to 70 \to 80 \to 90 \to null)$



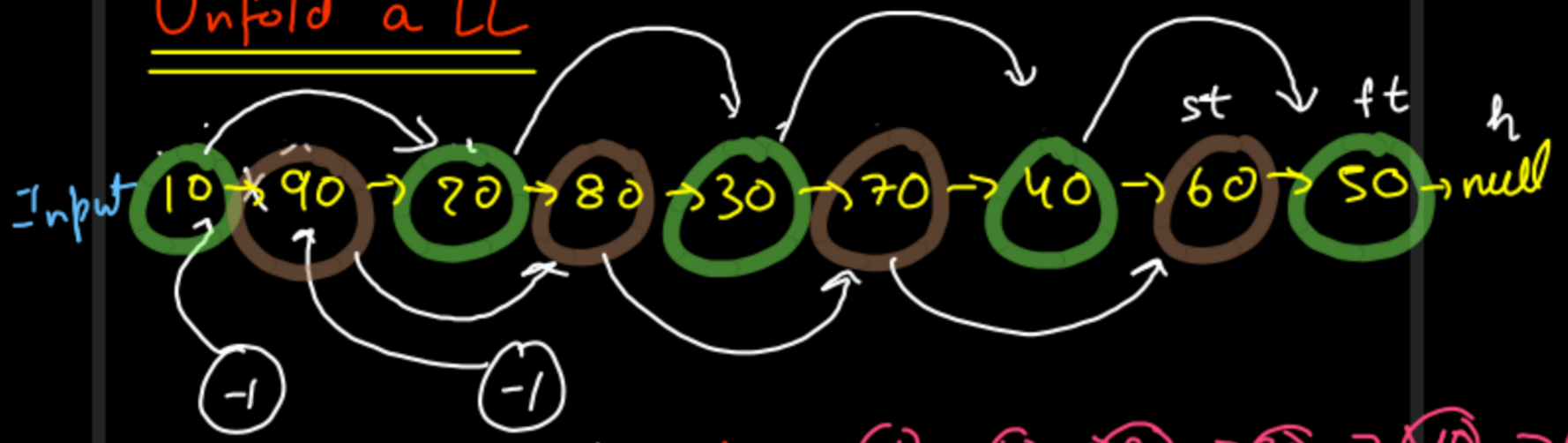
① Find Middle
② Reverse sec Part
③ Add Alternate nodes

Output: $10 \to 90 \to 20 \to 80 \to 30 \to 70 \to 40 \to 60 \to 50 \to null$

Code for fold all

```java
public void reorderList(ListNode head) {
    ListNode mid = middleNode(head);
    ListNode second = reverseList(mid.next);
    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 a LL



first (-1) → 10 → 20 → 30 → 40 → 50 → null

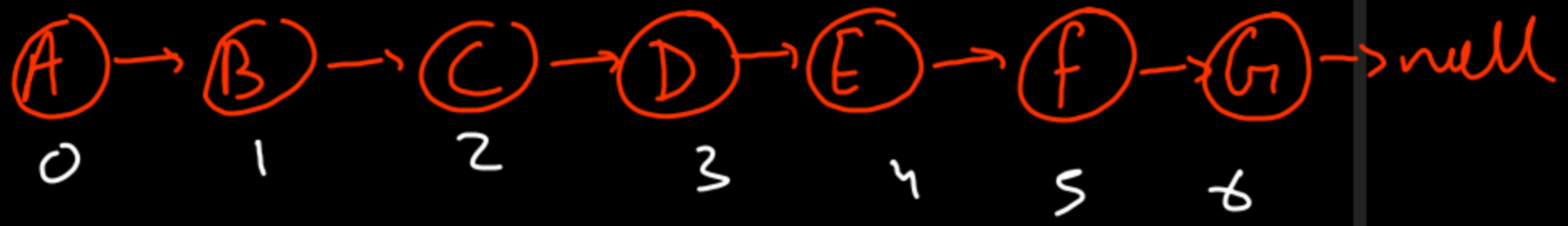
second (-1 → 90 → 80 → 70 → 60 → null) ⇒ Reverse & then Join

output 10 → 20 → 30 → 40 → 50 → 60 → 70 → 80 → 90 → null

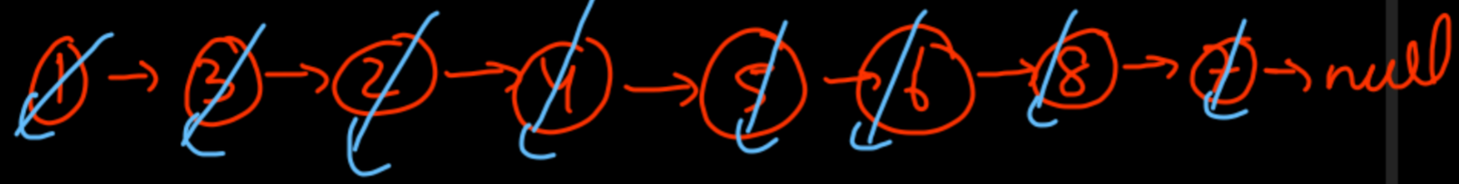
Unfold LL code

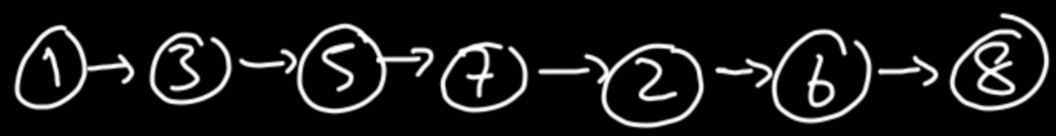
```
public static void unfold(ListNode head) {
    ListNode firstHead = new ListNode(-1);
    ListNode secondHead = new ListNode(-1);

    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;

    first.next = reverse(secondHead.next);
    head = firstHead.next;
}
```

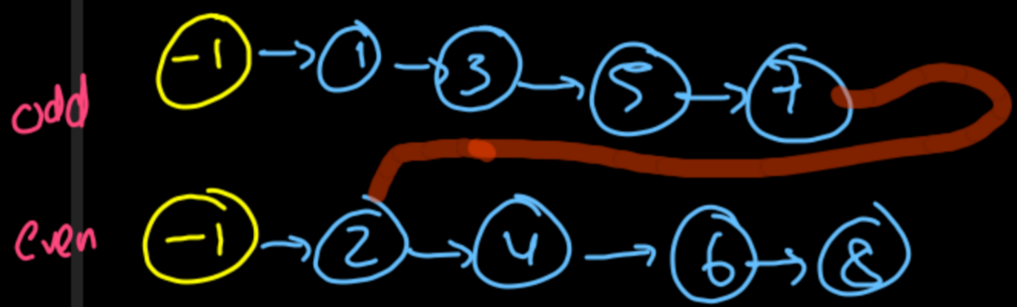
Odd Even List {By indices} {Unfold Variation}



Odd Even List {By Values} {Sort 01 Variation}







Code

```
Node divide(int N, Node head){
    // code here
    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;
}
```