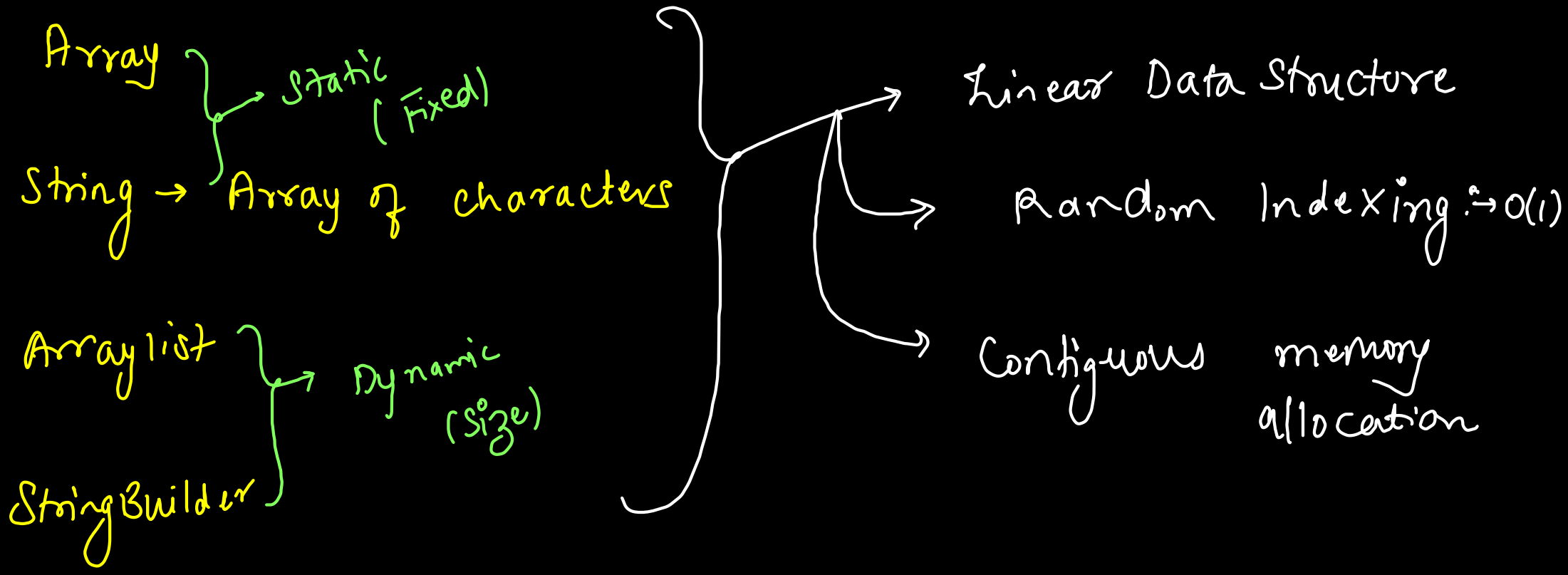
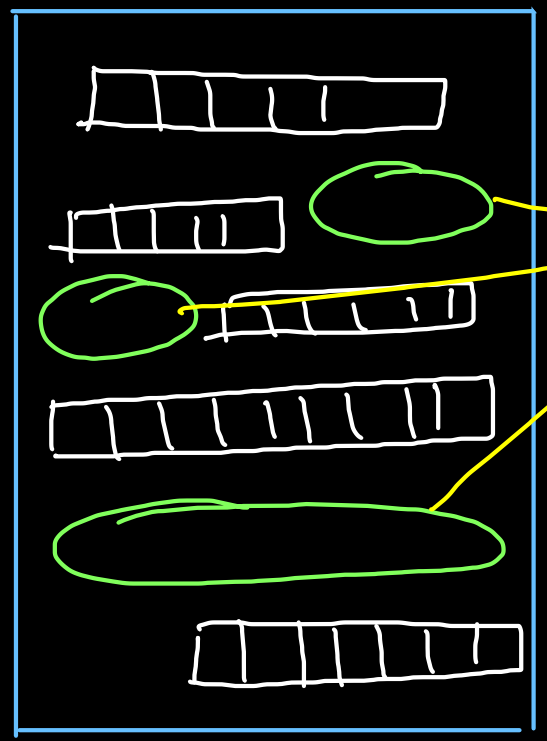
LinkedList


Array
String → Array of characters
Static (Fixed)
ArrayList
StringBuilder
Dynamic (Size)
Linear Data Structure
Random Indexing :→ O(1)
Contiguous memory allocation



Heap

Discontiguous memory

"Fragmentation"

Disadvantage of Array!


```
class ListNode {

    int data;

    ListNode next;

}
```

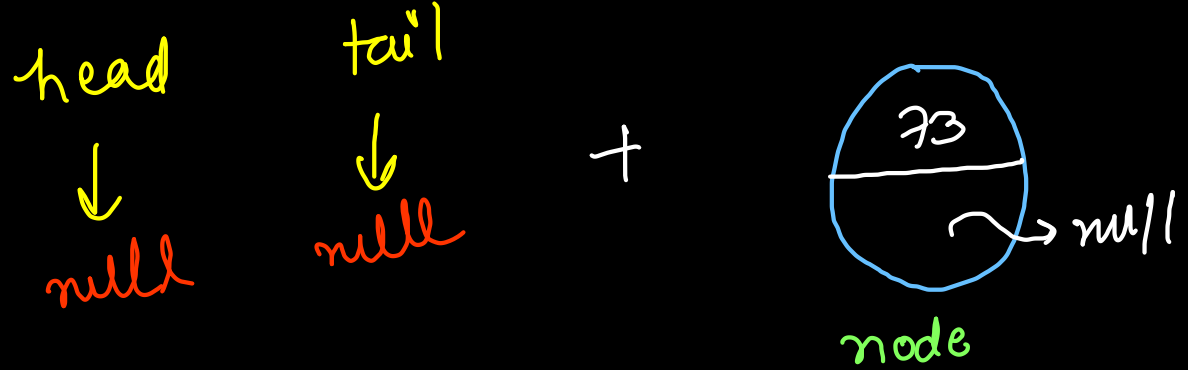
Collection
of nodes
⇓
linkedlist

⇒ Singly

⇒ forward iteration

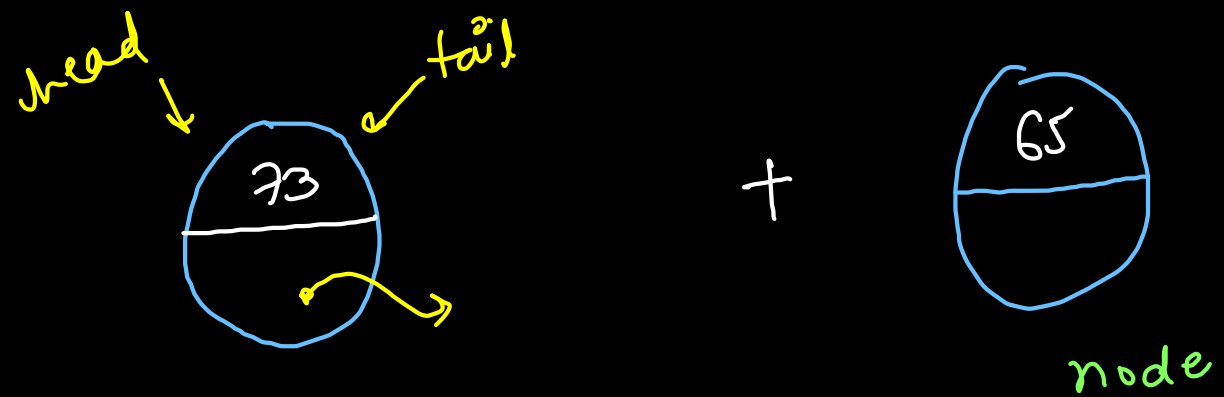
Design linked list

Forward/Singly linked list

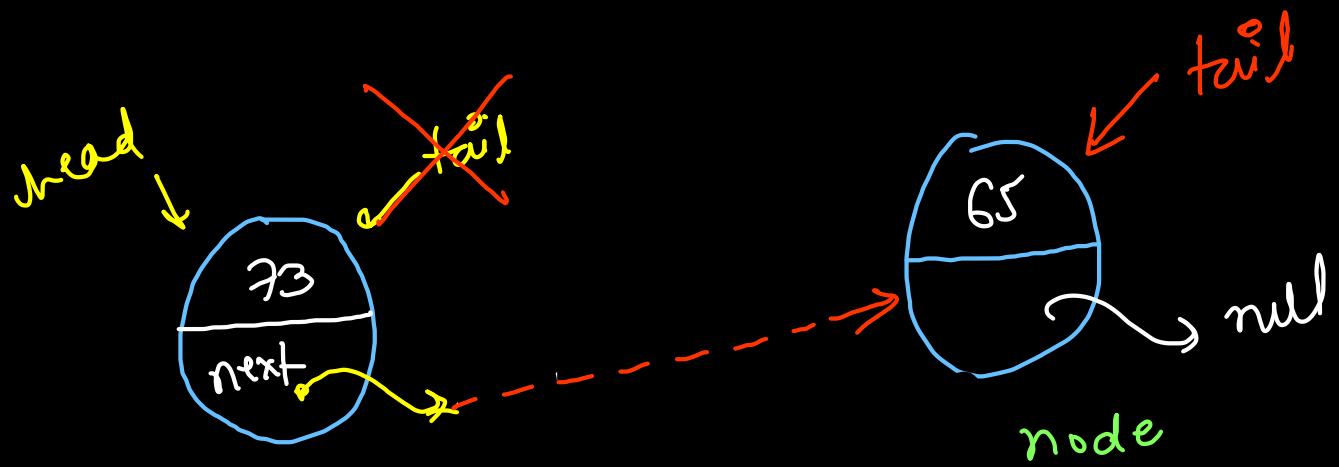


add(73)

Insert At tail



add(65)



① tail.next = node;

② tail = node;

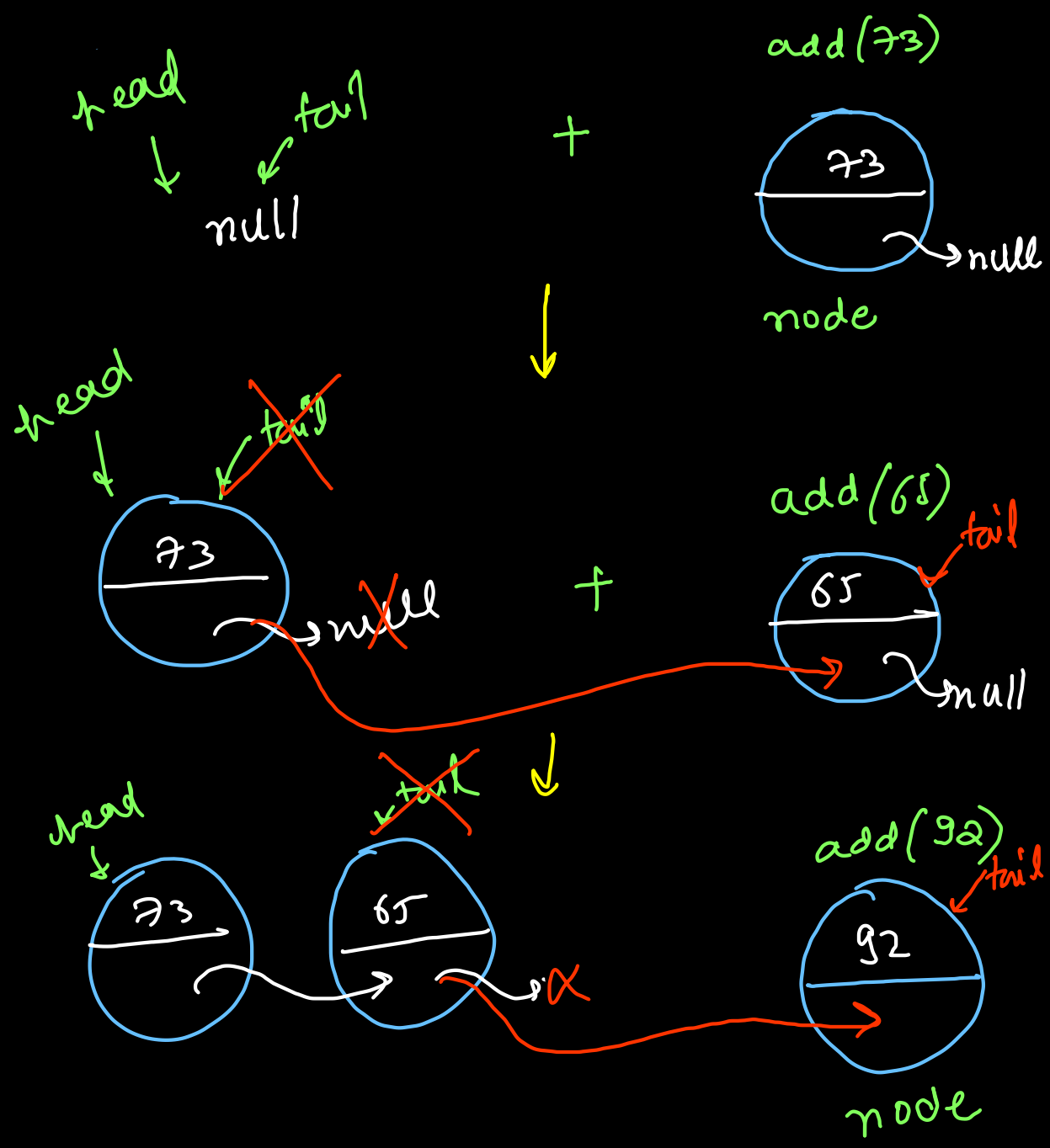
```
public void addAtTail(int val) {
    ListNode node = new ListNode();
    node.val = val;

    if(head == null){
        head = tail = node;
    } else {
        tail.next = node;
        tail = node;
    }
}
```

tail pointer

```java
public void addAtTail(int val) {
    ListNode node = new ListNode();
    node.val = val;

    if(head == null){
        head = tail = node;
    } else {
        tail.next = node;
        tail = node;
    }
}
```

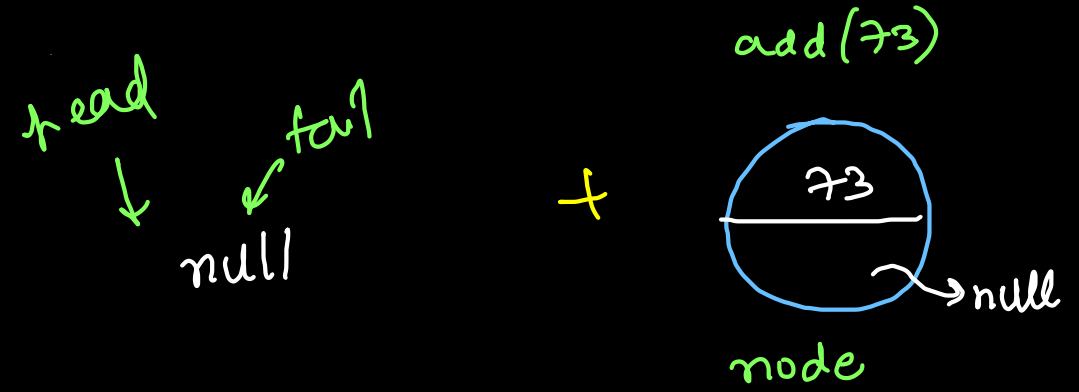
O(1)

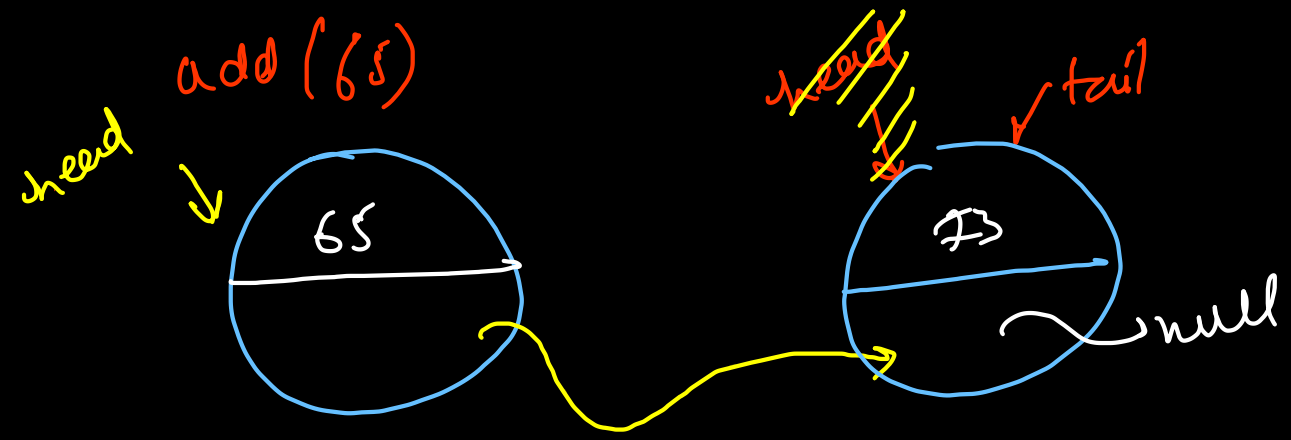
If tail is not available
=> O(n) addAtTail

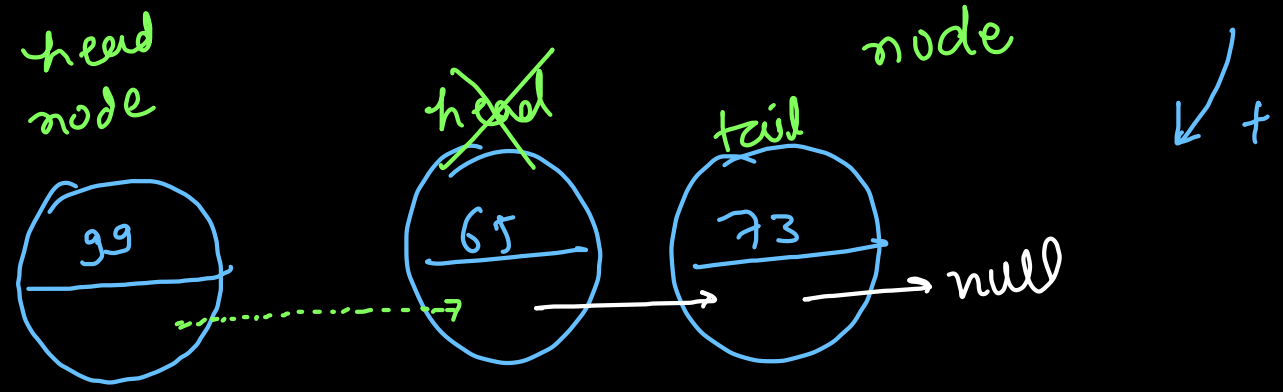


```java
public void addAtHead(int val) {
    ListNode node = new ListNode();
    node.val = val;

    if(head == null){
        head = tail = node;
    } else {
        node.next = head;
        head = node;
    }
}
```

O(1)



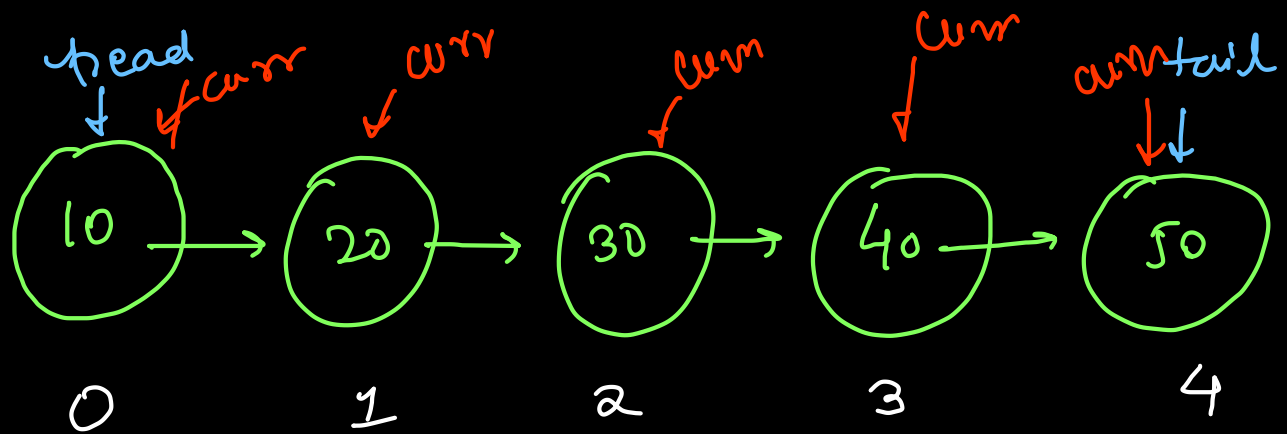




```java
class ListNode{
    int val;
    ListNode next;
}
```

```java
class MyLinkedList {
    ListNode head, tail;
```

```java
public void addAtHead(int val) {
    ListNode node = new ListNode();
    node.val = val;

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

public void addAtTail(int val) {
    ListNode node = new ListNode();
    node.val = val;

    if(head == null){
        head = tail = node;
    } else {
        tail.next = node;
        tail = node;
    }
}
```



O(1) best case

list.get(0) ⇒ 10 0 jump

list.get(4) ⇒ 50 0 jump

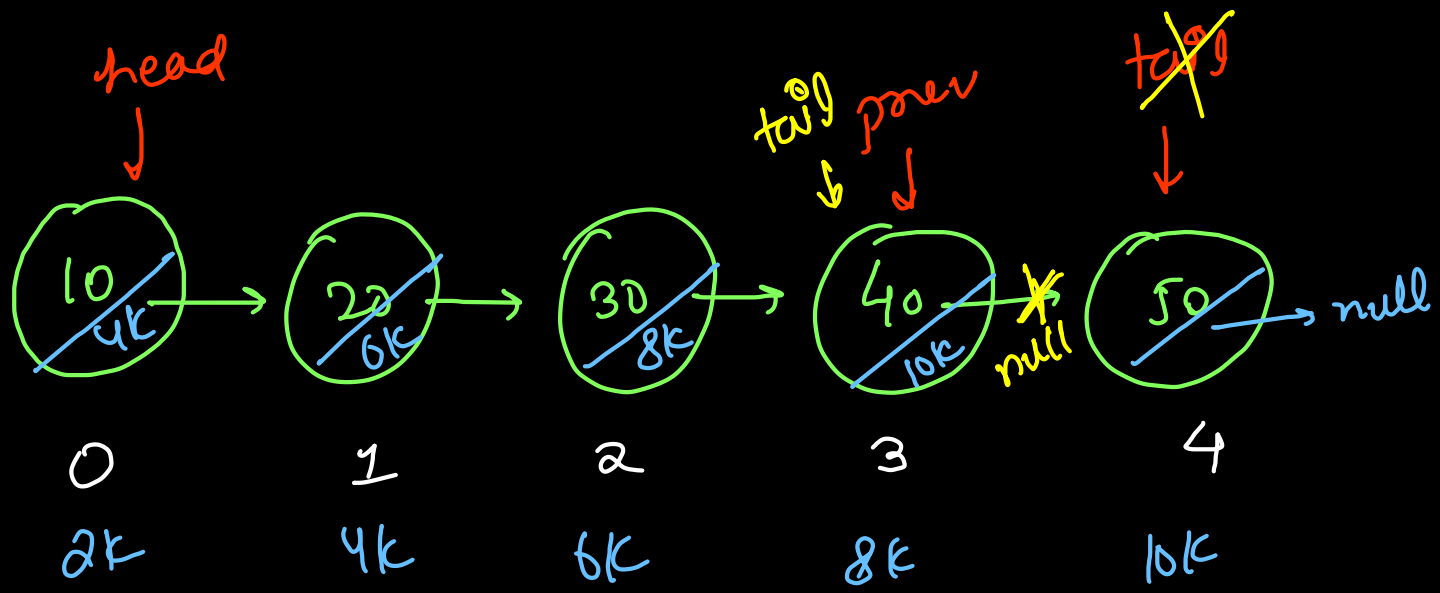
LinkedList doesn't support random indexing

O(N) worst case

list.get(1) ⇒ 20 1 jump

list.get(2) ⇒ 30 2 jumps

list.get(3) ⇒ 40 3 jumps




```java
public void deleteAtHead(){
    head = head.next;
    if(head == null) tail = null;
}

public void deleteAtTail(){

}
```

```java
public int get(int index) {
    ListNode curr = head;
    for(int idx = 0; idx < index; idx++){
        curr = curr.next;
    }
    return curr.val;
}
```

Best case : O(1)

Average Case: O(n)

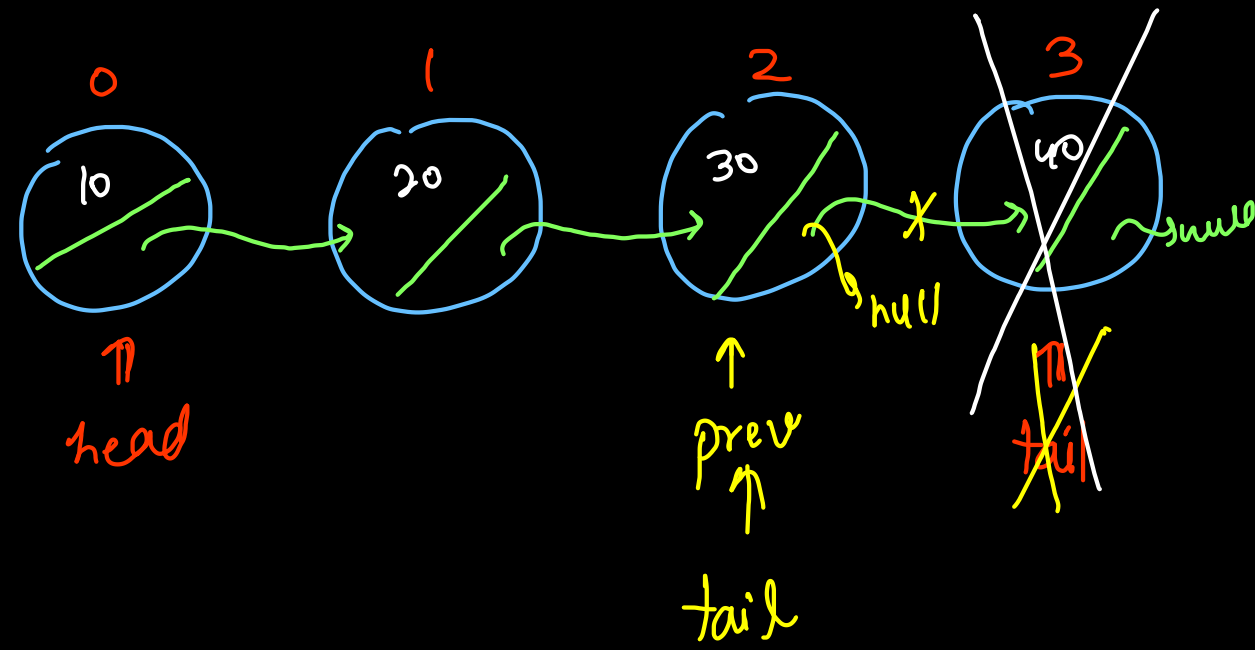
Worst case: O(n)

```java
public void deleteAtHead(){
    size--;
    head = head.next;
}

public ListNode getNode(int index){
    ListNode curr = head;
    for(int idx = 0; idx < index; idx++){
        curr = curr.next;
    }
    return curr;
}

public void deleteAtTail(){
    size--;
    ListNode prev = getNode(size - 1);
    prev.next = null;
    prev = tail;
}
```

O(1)

secondlast node

O(n)



Singly linked list

If tail pointer

addAtHead ⇒ O(1)

removeAtHead ⇒ O(1)

addAtTail ⇒ O(1)

* removeAtTail ⇒ O(n)

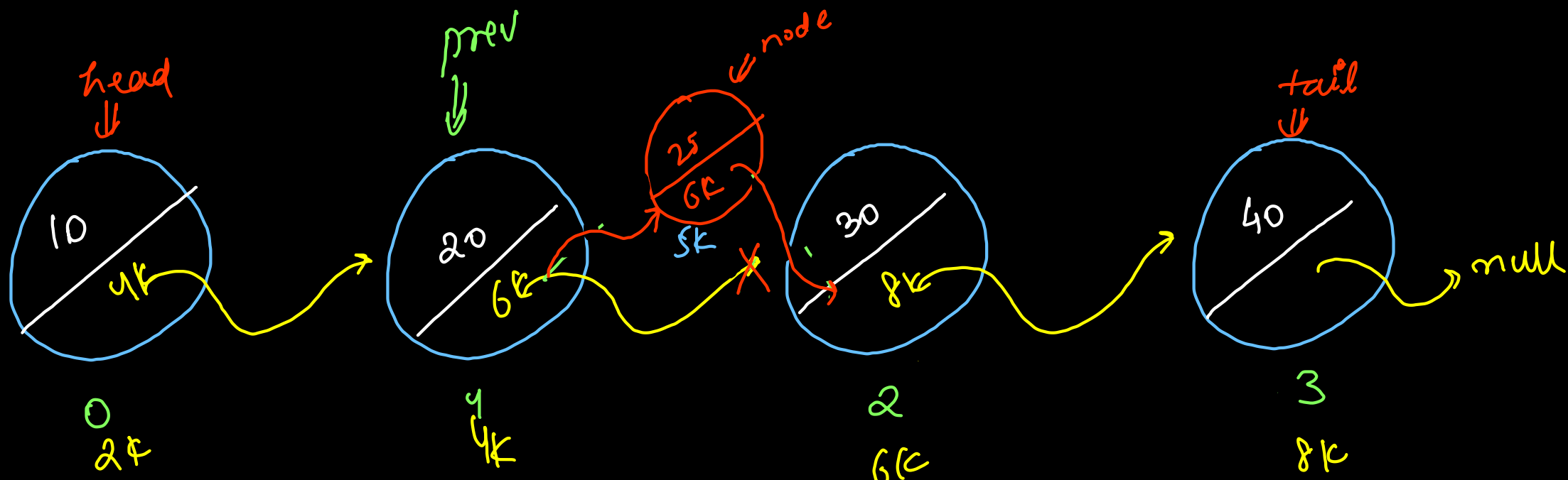
If tail pointer not available

addAtHead ⇒ O(1)

removedHead ⇒ O(1)

addAtTail ⇒ O(n)

* removeAtTail ⇒ O(n)



addAtIndex(5, 0) ⇒ head

addAtIndex(52, 4) ⇒ tail
size

addAtIndex(25, 2)

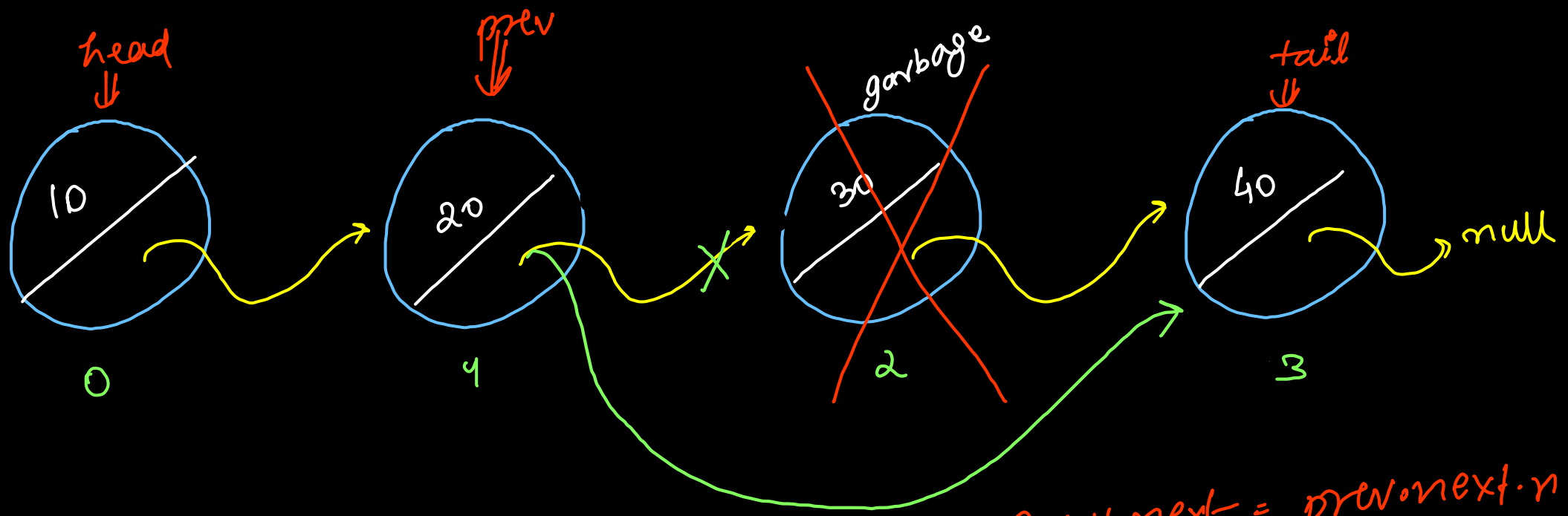
```
node.next = prev.next;
prev.next = node;
```

```java
public void addAtIndex(int index, int val) {
    if(index == 0){
        addAtHead(val);
        return;
    }

    if(index == size){
        addAtTail(val);
        return;
    }

    ListNode node = new ListNode();
    node.val = val;
    ListNode prev = getNode(index - 1);
    node.next = prev.next;
    prev.next = node;
}
```

addAtHead(val); → O(1)

addAtTail(val); → O(1)

getNode(index - 1); → O(n)



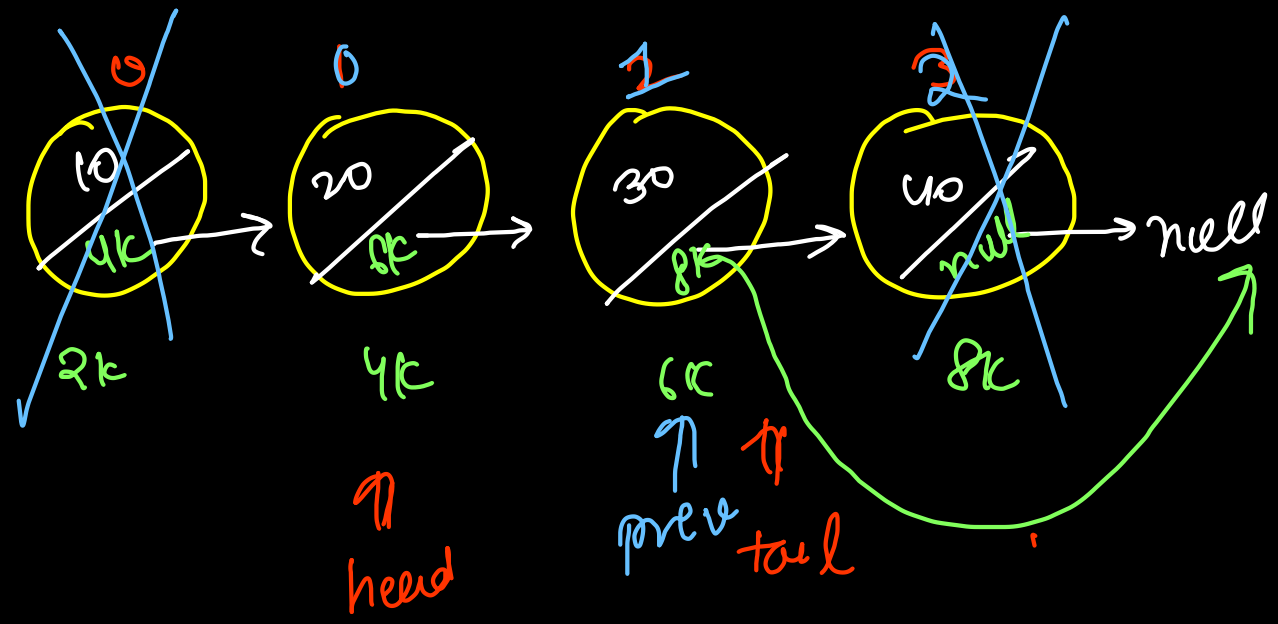
prev.next = prev.next.next;

deleteAt(0); ⇒ delete head

deleteAt(3); ⇒ delete tail

deleteAt(2);

```java
public void deleteAtIndex(int index) {
    if(index < 0 || index >= size)
        return;

    size--;
    if(size == 0){
        // empty linked list
        head = tail = null;
        return;
    }

    if(index == 0){
        // delete head node
        head = head.next;
        return;
    }

    ListNode prev = getNode(index - 1);
    prev.next = prev.next.next;

    if(index == size){
        // delete tail node
        tail = prev;
    }
}
```



deleteAt(-1) deleteAt(4) : invalid

size = 4 / 3 2

= size

deleteAt(0)

head

deleteAt(2)

```
class ListNode{
    int val;
    ListNode next;
}

class MyLinkedList {
    ListNode head, tail;
    int size = 0;

    public ListNode getNode(int index){
        if(index < 0 || index >= size) return null;

        ListNode curr = head;
        for(int idx = 0; idx < index; idx++){
            curr = curr.next;
        }
        return curr;
    }

    public int get(int index) {
        if(index < 0 || index >= size) return -1;
        return getNode(index).val;
    }

    public void addAtHead(int val) {
        ListNode node = new ListNode();
        node.val = val;
        size++;

        if(head == null){
            head = tail = node;
        } else {
            node.next = head;
            head = node;
        }
```

O(n)

O(1)

O(1)

```
    public void addAtTail(int val) {
        ListNode node = new ListNode();
        node.val = val;
        size++;

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

    public void addAtIndex(int index, int val) {
        if(index < 0 || index > size)
            return;

        if(index == 0){
            addAtHead(val);
            return;
        }

        if(index == size){
            addAtTail(val);
            return;
        }

        ListNode node = new ListNode();
        node.val = val;
        ListNode prev = getNode(index - 1);
        size++;
        node.next = prev.next;
        prev.next = node;
    }
```

O(1)

O(1)

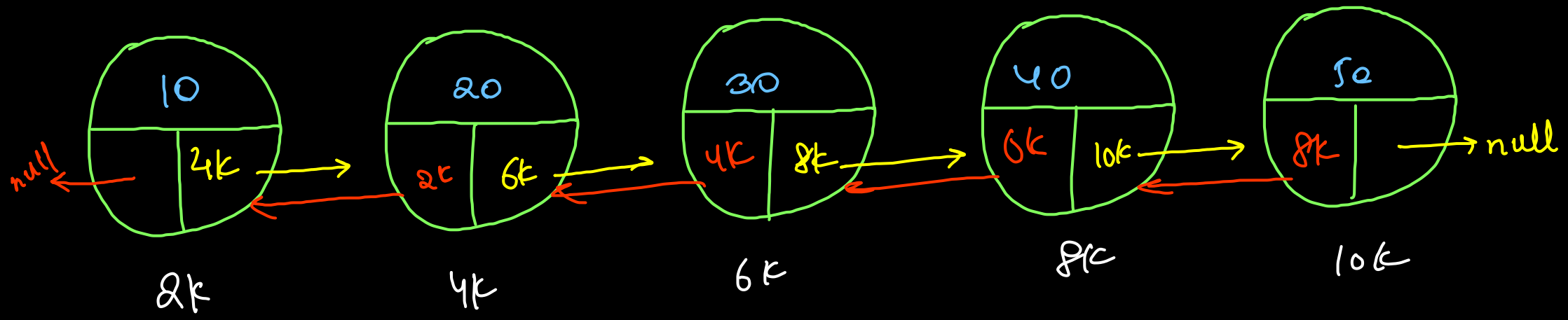
O(n)

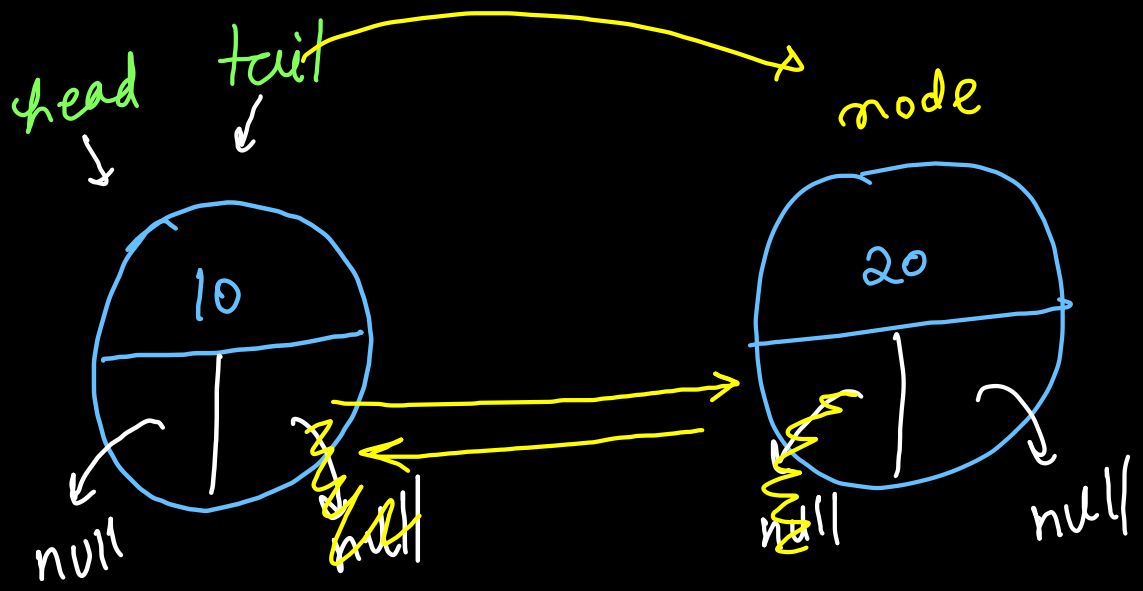
```
    public void deleteAtIndex(int index) {
        if(index < 0 || index >= size)
            return;

        size--;
        if(size == 0){
            // empty linked list
            head = tail = null;
            return;
        }

        if(index == 0){
            // delete head node
            head = head.next;
            return;
        }

        ListNode prev = getNode(index - 1);
        prev.next = prev.next.next;

        if(index == size){
            // delete tail node
            tail = prev;
        }
    }
```

O(1)

O(1)

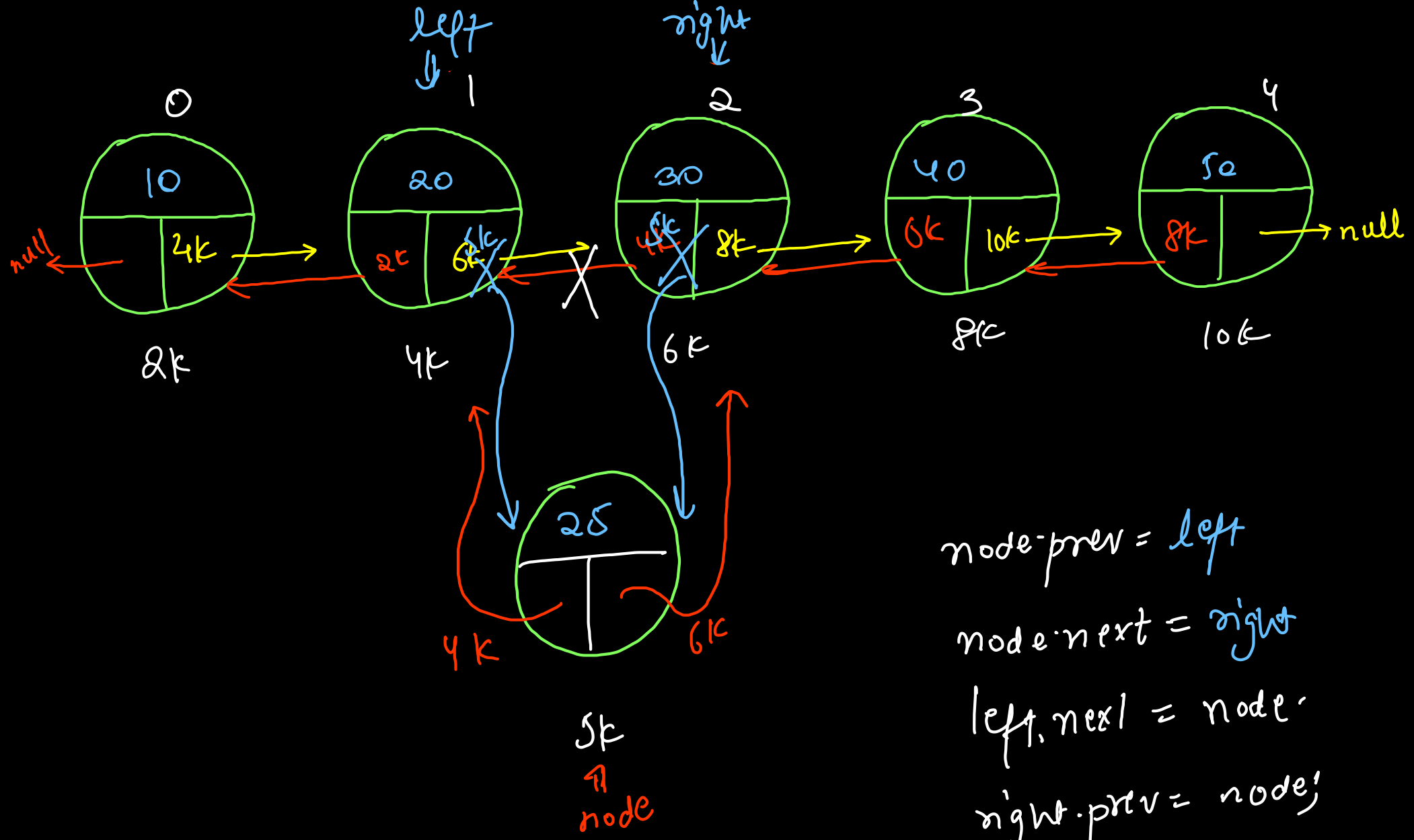
O(n)





addAttail(10)

addAttail(20)



node.prev = left

node.next = right

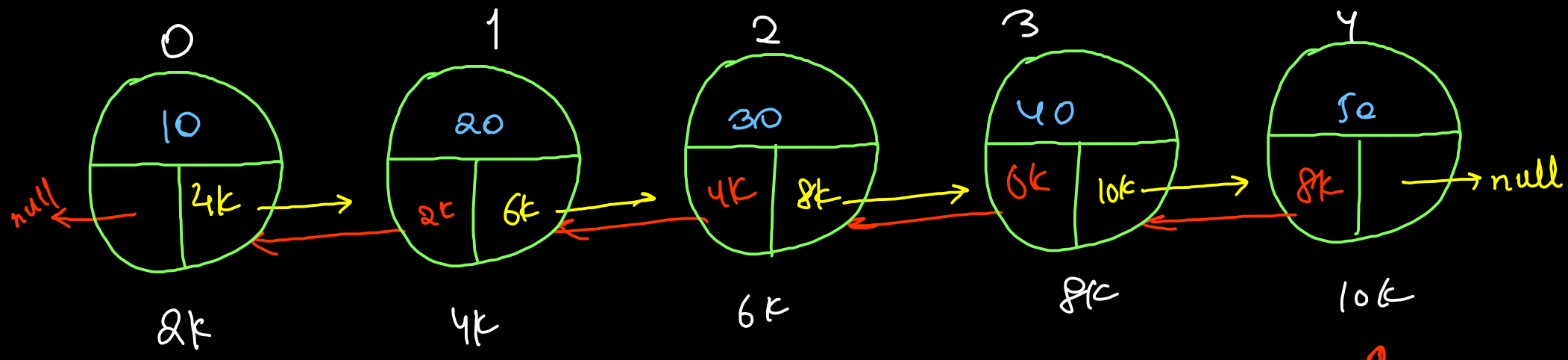
left.next = node;

right.prev = node;

```java
public void addAtHead(int val) {
    ListNode node = new ListNode();
    node.val = val;
    size++;

    if(head == null){
        head = tail = node;
    } else {
        node.next = head;
        head.prev = node;
        head = node;
    }
}

public void addAtTail(int val) {
    ListNode node = new ListNode();
    node.val = val;
    size++;

    if(head == null){
        head = tail = node;
    } else {
        tail.next = node;
        node.prev = tail;
        tail = node;
    }
}
```

```java
public void addAtIndex(int index, int val) {
    if(index < 0 || index > size)
        return;

    if(index == 0){
        addAtHead(val);
        return;
    }

    if(index == size){
        addAtTail(val);
        return;
    }

    ListNode node = new ListNode();
    node.val = val;

    ListNode left = getNode(index - 1);
    ListNode right = left.next;

    size++;
    node.prev = left;
    node.next = right;
    left.next = node;
    right.prev = node;
}
```

```java
class ListNode{
    int val;
    ListNode next, prev;
}
```



deleteAt(head)

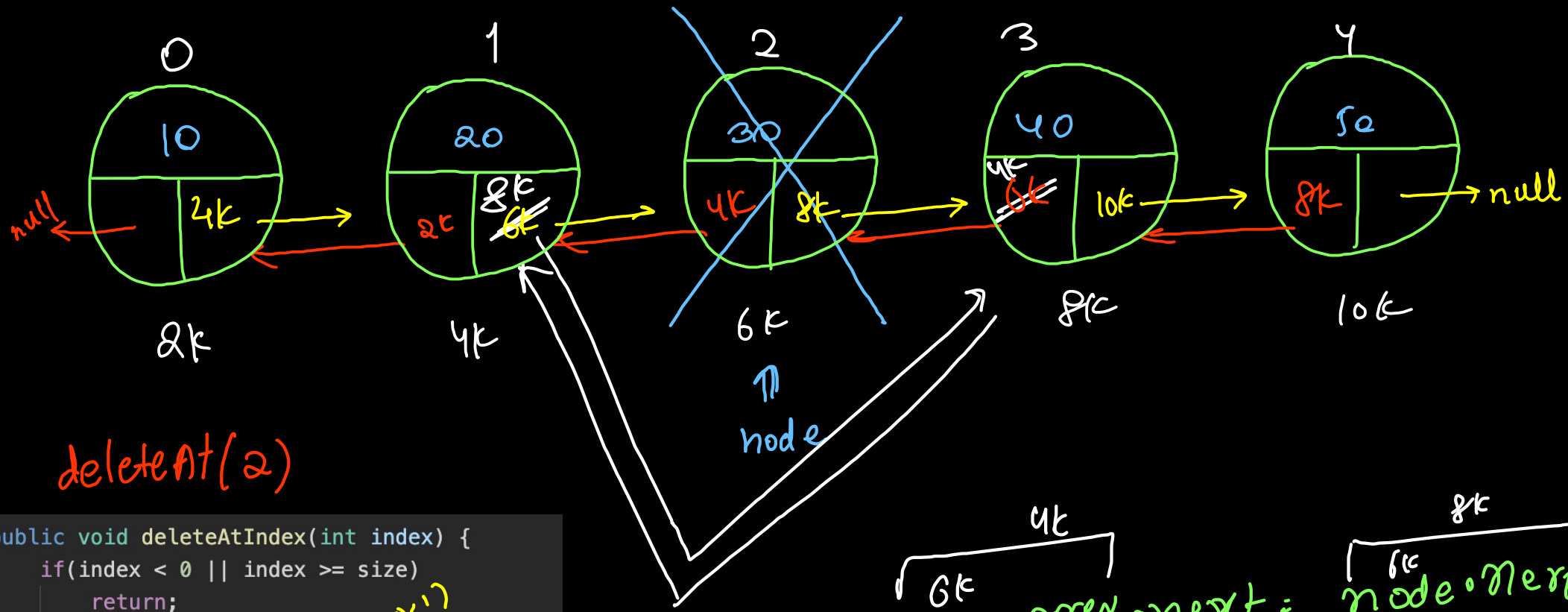
deleteAt(tail)

```
public void deleteHead(){
    size--;
    if(size == 0){
        head = tail = null;
        return;
    }

    head = head.next;
    head.prev = null;

}
```

O(1)

```
public void deleteTail(){
    size--;
    if(size == 0){
        head = tail = null;
        return;
    }

    tail = tail.prev;
    tail.next = null;

}
```

O(1)



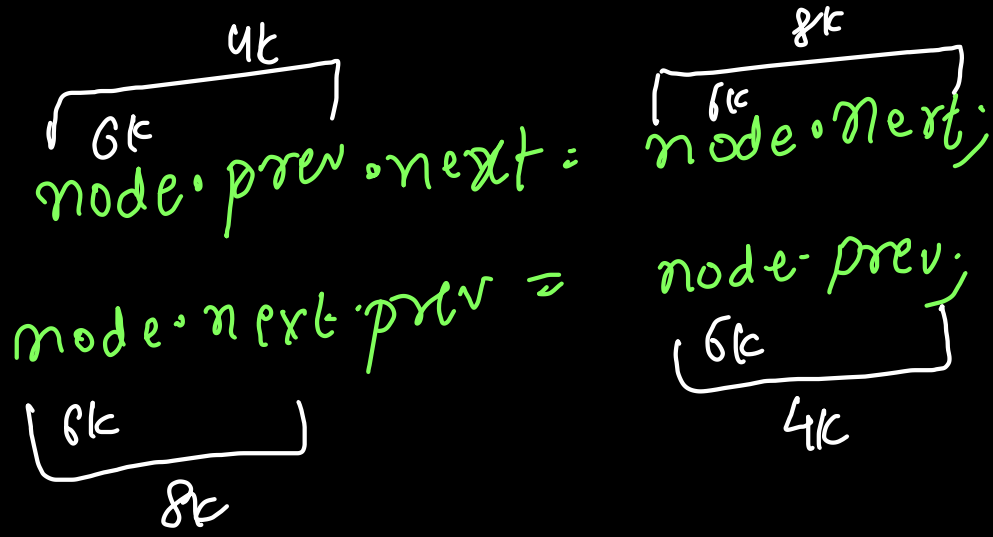
deleteAt(2)

```java
public void deleteAtIndex(int index) {
    if(index < 0 || index >= size)
        return;

    if(index == 0) deleteHead();
    else if(index == size - 1) deleteTail();
    else {
        size--;
        ListNode node = getNode(index);
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }
}
```

O(1)

O(1)

O(n)



```
class ListNode{
    int val;
    ListNode next, prev;
}

class MyLinkedList {
    ListNode head, tail;
    int size = 0;

    public ListNode getNode(int index){
        if(index < 0 || index >= size) return null;

        ListNode curr = head;
        for(int idx = 0; idx < index; idx++){
            curr = curr.next;
        }
        return curr;
    }

    public int get(int index) {
        if(index < 0 || index >= size) return -1;
        return getNode(index).val;
    }
```

Doubly Linked List!

O(n)

O(n)

```
    public void addAtHead(int val) {
        ListNode node = new ListNode();
        node.val = val;
        size++;

        if(head == null){
            head = tail = node;
        } else {
            node.next = head;
            head.prev = node;
            head = node;
        }
    }

    public void addAtTail(int val) {
        ListNode node = new ListNode();
        node.val = val;
        size++;

        if(head == null){
            head = tail = node;
        } else {
            tail.next = node;
            node.prev = tail;
            tail = node;
        }
    }
```

O(1)

O(1)

```
    public void addAtIndex(int index, int val) {
        if(index < 0 || index > size)
            return;

        if(index == 0){
            addAtHead(val);
            return;
        }

        if(index == size){
            addAtTail(val);
            return;
        }

        ListNode node = new ListNode();
        node.val = val;

        ListNode left = getNode(index - 1);
        ListNode right = left.next;

        size++;
        node.prev = left;
        node.next = right;
        left.next = node;
        right.prev = node;
    }
```

O(1)

O(1)

O(n)

```
    public void deleteHead(){
        size--;
        if(size == 0){
            head = tail = null;
            return;
        }

        head = head.next;
        head.prev = null;
    }
```

O(1)

```
    public void deleteTail(){
        size--;
        if(size == 0){
            head = tail = null;
            return;
        }

        tail = tail.prev;
        tail.next = null;
    }
```

O(1)

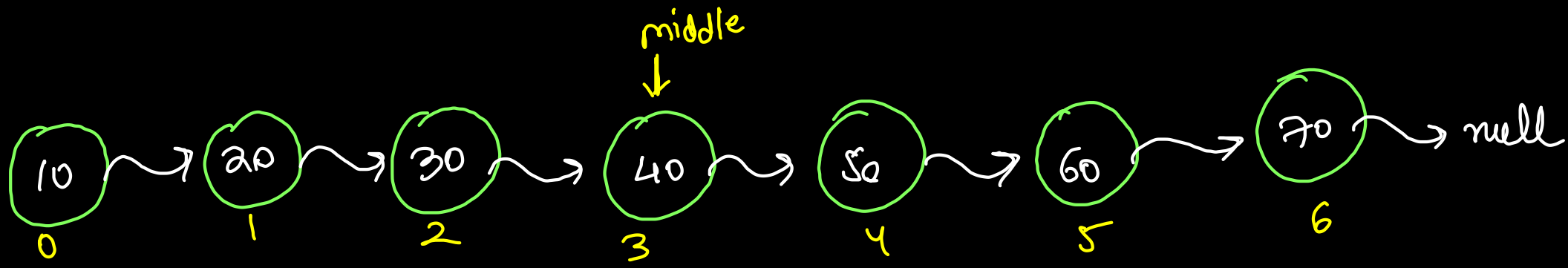
```
    public void deleteAtIndex(int index) {
        if(index < 0 || index >= size)
            return;

        if(index == 0) deleteHead();
        else if(index == size - 1) deleteTail();
        else {
            size--;
            ListNode node = getNode(index);
            node.prev.next = node.next;
            node.next.prev = node.prev;
        }
    }
```

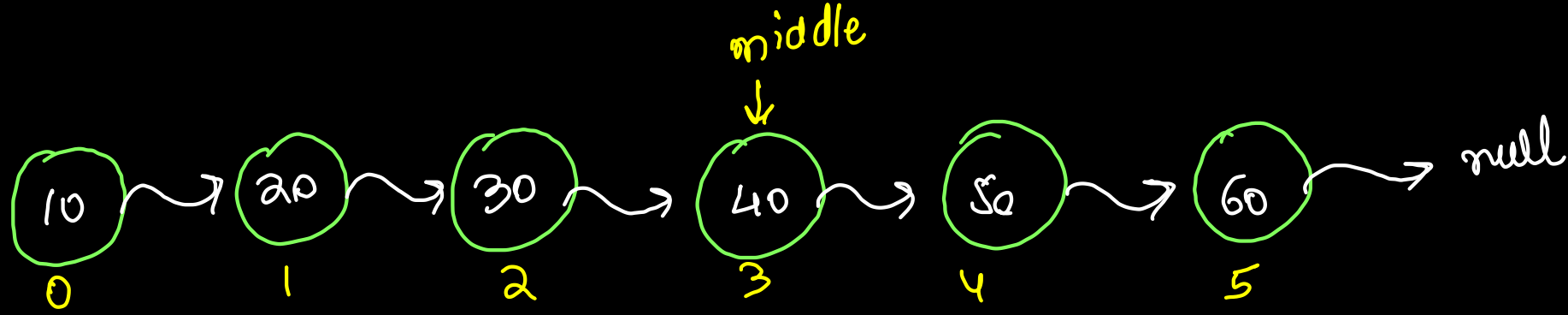
O(1)

O(1)

O(n)

Middle of LL LC 876

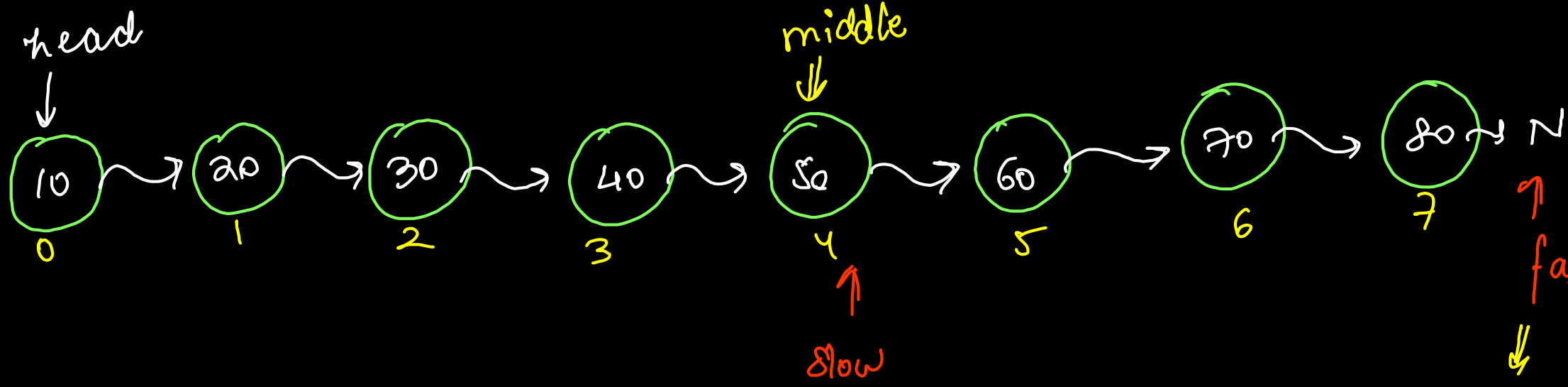




Two Pass Algorithm

1. Count no of nodes in LL
2. Return the $n/2^{th}$ node of the LL

```java
// Total Time = O(N), Space = O(1)
public ListNode twoPassAlgo(ListNode head){
    // Pass 1: Count Number of Nodes
    ListNode curr = head;
    int count = 0;
    while(curr != null){
        curr = curr.next;
        count++;
    }

    // Pass 2: Return the N/2th Node of LL
    curr = head;
    for(int i = 0; i < count / 2; i++)
        curr = curr.next;
    return curr;
}
```

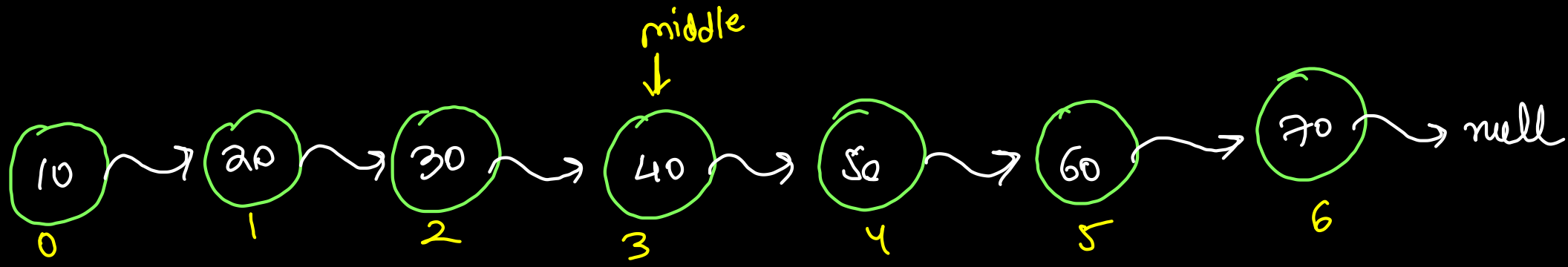


slow → middle → n/2 dist
fast → null → n dist
} → time taken is same → 1 node jump, 2 node jump

Single Pass Algorithm / Two-Pointer Algorithm / Slow-Fast Pt Algo / Hare-Tortoise Algorithm

```
slow = fast = head;

while (fast != null) {
    slow = slow.next;          // n/2 } O(n) time
    fast = fast.next.next;     // n
}
return slow;
```

Space
$\Rightarrow O(1)$




```
slow = fast = head;
while ( fast != null && fast.next != null)
    slow = slow.next;
    fast = fast.next.next;
}
return slow;
```

n/2 } O(n) time

n

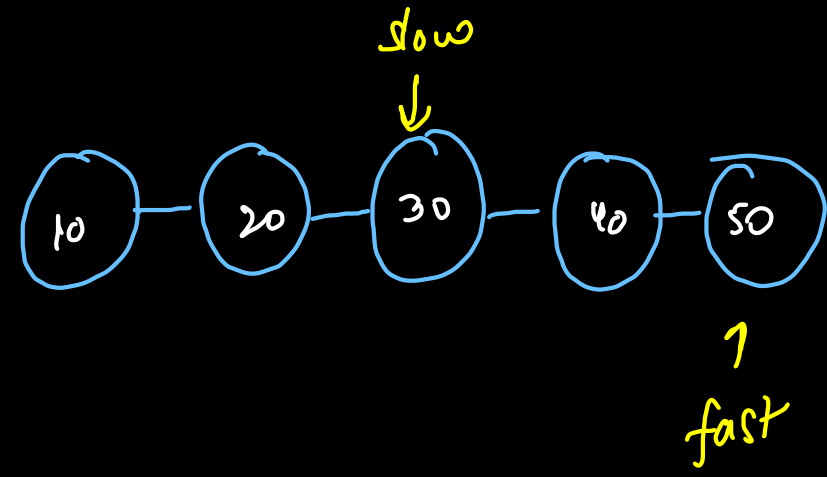
Space ⇒ O(1)

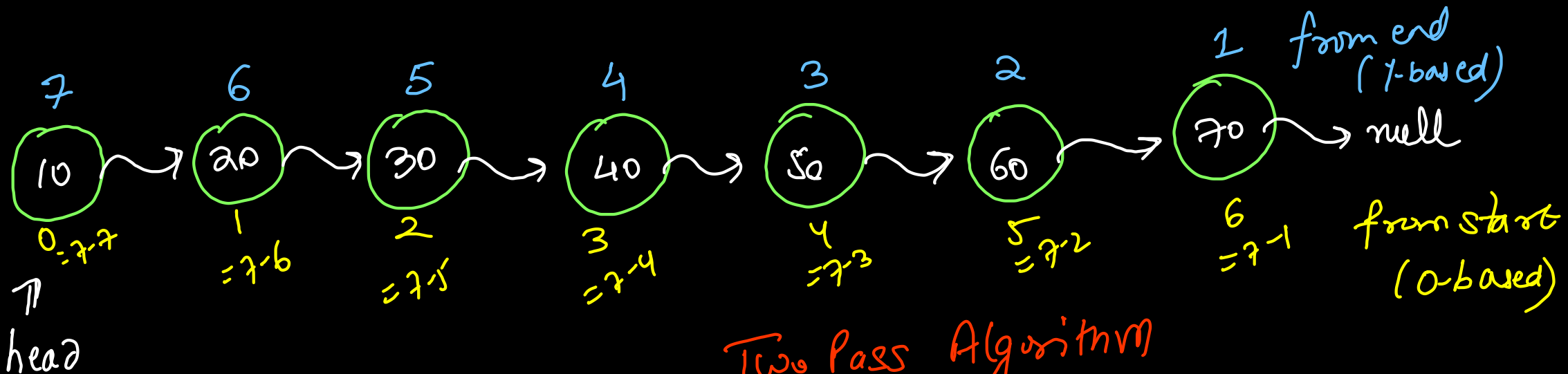
fast.next = null

⇓

odd nodes

```java
// Total Time = O(N), Space = O(1)
public ListNode onePassAlgo(ListNode head){
    ListNode slow = head, fast = head;
    while(fast != null && fast.next != null){
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```





Two Pass Algorithm

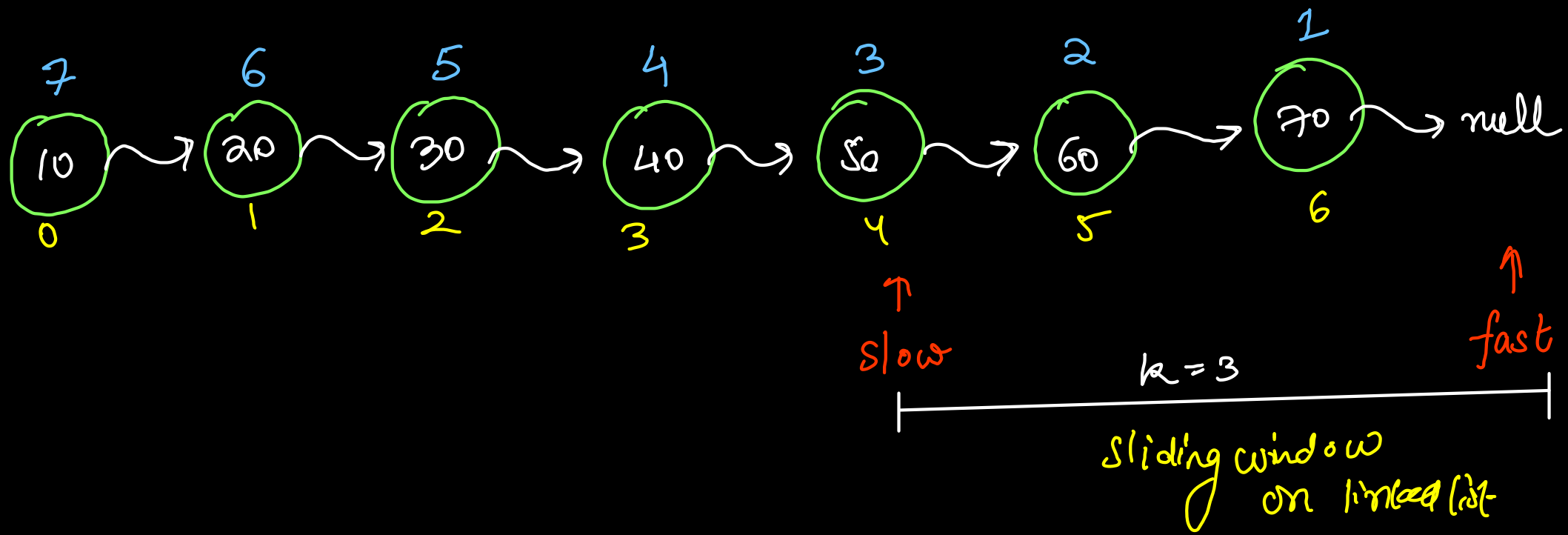
1) Count no of nodes

2) get ($n-k^{th}$) node from starting

get(3rd from end) = 50

N = 7, k = 3 } index = n - k = 7 - 3 = 4

```
int getNthFromLast(Node head, int k)
{
    // pass 1: count number of nodes
    int n = 0;
    Node curr = head;

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

    if(k <= 0 || k > n)
        return -1;

    // pass 2:
    // kth node from end = (n - k)th node from start
    curr = head;
    for(int i = 0; i < n - k; i++){
        curr = curr.next;
    }
    return curr.data;
}
```



slow = (50)
fast = null
→ dist should be fixed ⇒ k=3

k = 3rd node from end

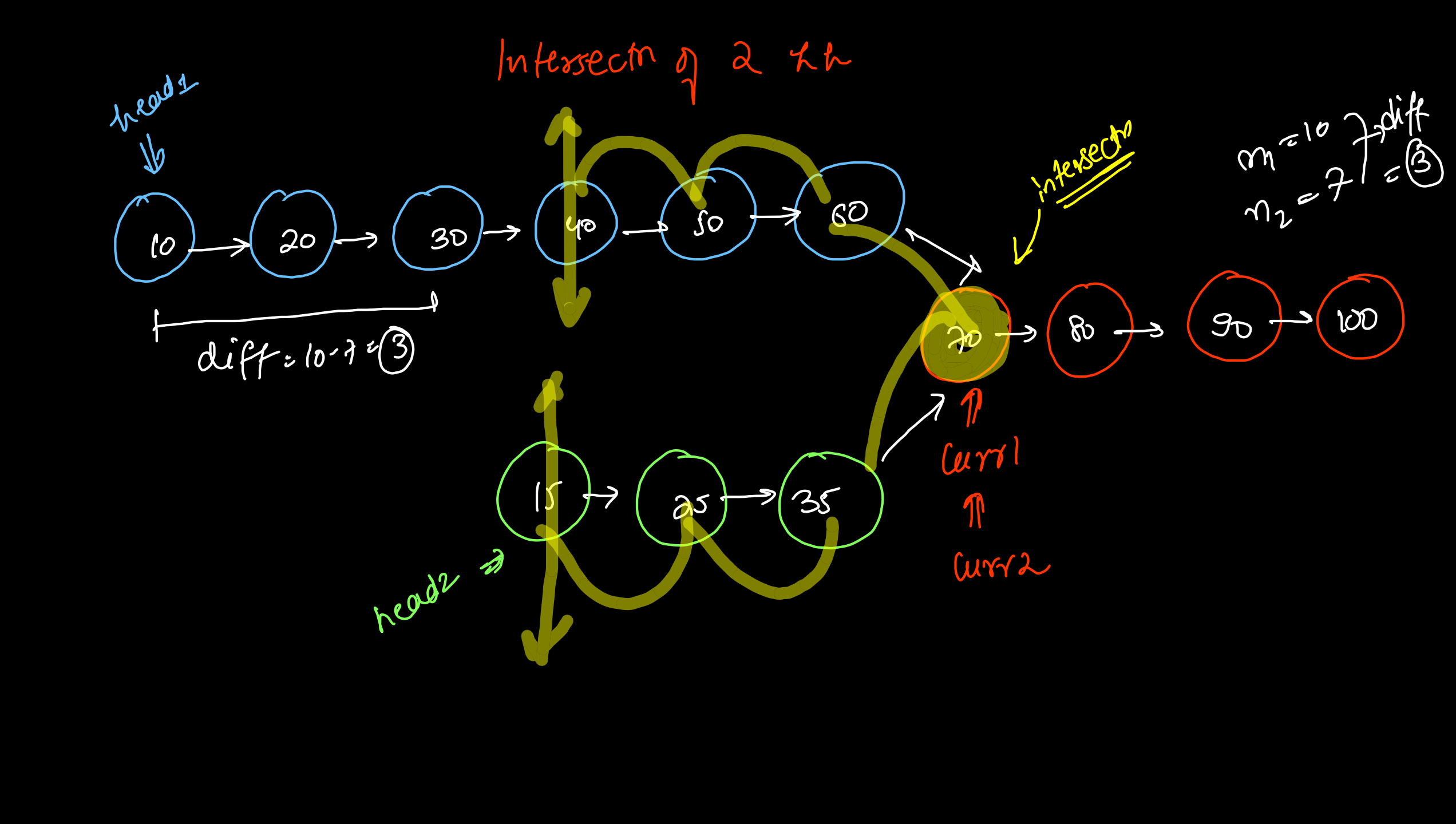
slow = (20)
fast = null
→ dist = k = 6

k = 6th node from end

```java
int onePassAlgo(Node head, int k){
    Node slow = head, fast = head;

    // Fast should be k dist ahead of slow
    for(int i = 0; i < k; i++){
        if(fast == null) return -1;
        fast = fast.next;
    }

    // Sliding Window
    while(fast != null){
        slow = slow.next;
        fast = fast.next;
    }
    return slow.data;
}
```

Total time = O(n)

Space = O(1)


Intersection of 2 LL
head1
10
20
30
40
50
60
70
80
90
100
diff = 10 - 7 = 3
intersect
m = 10
n2 = 7
diff = 3
head2
15
25
35
curr1
curr2

① Brute force

Time ⇒ $O(n^2)$ pairs ⇒ find first common ancestor

Space ⇒ $O(1)$

② Hashmap

Insert all nodes of one tree in hashmap ⇒ $O(n_1)$ } $O(n_1+n_2)$

Search for other tree in hashmap ⇒ $O(n_2)$

Extra space ⇒ $O(n)$

```java
public int size(ListNode curr){
    int count = 0;
    while(curr != null){
        curr = curr.next;
        count++;
    }
    return count;
}

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    // Figure out the lengths of linked list
    int sizeA = size(headA);
    int sizeB = size(headB);

    ListNode currA = headA, currB = headB;

    // Maintain the Lead by Diff of Size
    for(int i = 0; i < (sizeA - sizeB); i++)
        currA = currA.next;

    for(int i = 0; i < (sizeB - sizeA); i++)
        currB = currB.next;

    // meet at the intersection point
    while(currA != currB){
        currA = currA.next;
        currB = currB.next;
    }

    return currA;
}
```

$O(n_1)$

$O(n_2)$

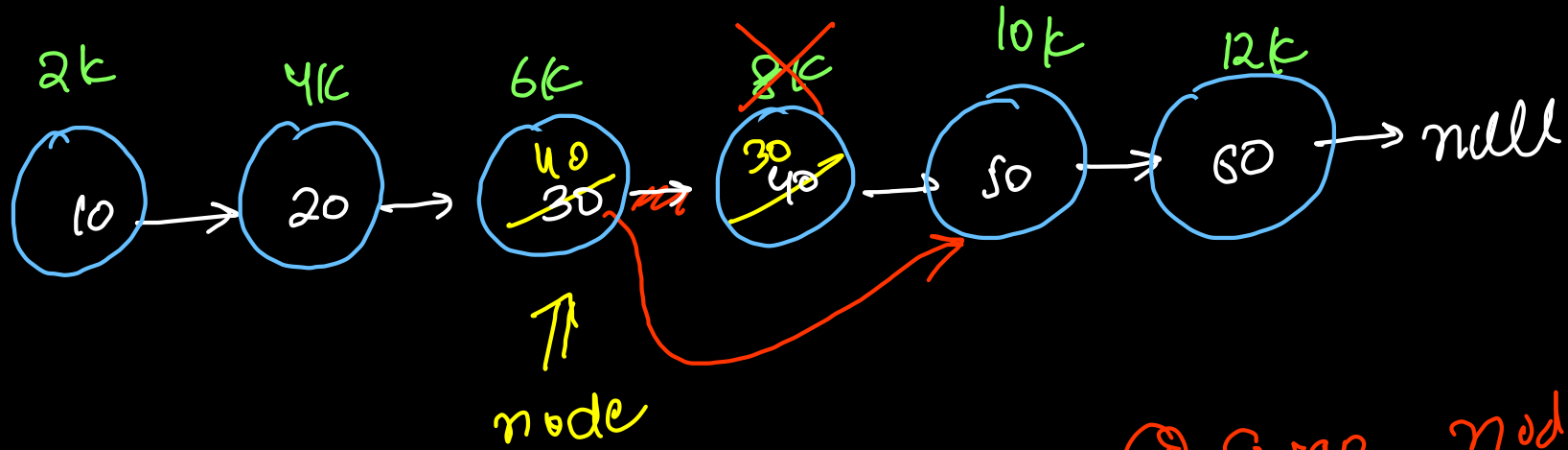
$O(|n_1 - n_2|)$

$O(n_1 + n_2)$

Total time $\Rightarrow O(n)$

Extra Space $\Rightarrow O(1)$

Delete Node W/o head node LC 237




```
class Solution {
    public void deleteNode(ListNode node) {
        // Store Data of Next Node in Node
        node.val = node.next.val;

        // Delete Next Node
        node.next = node.next.next;
    }
}
```

① Swap node & node.next data

② Delete node.next

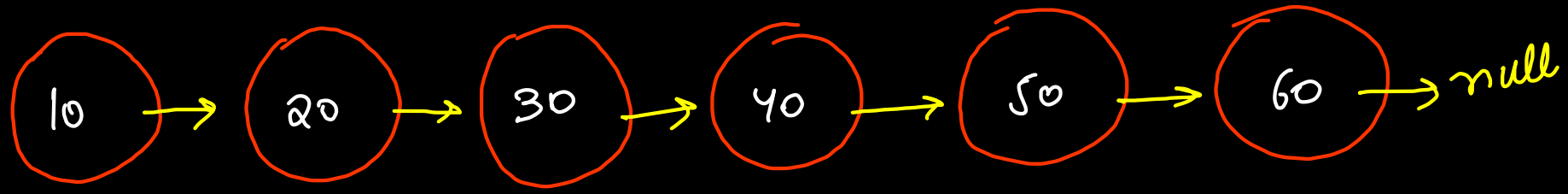
node.next = node.next.next

Linked list Cycle - I

LC 141

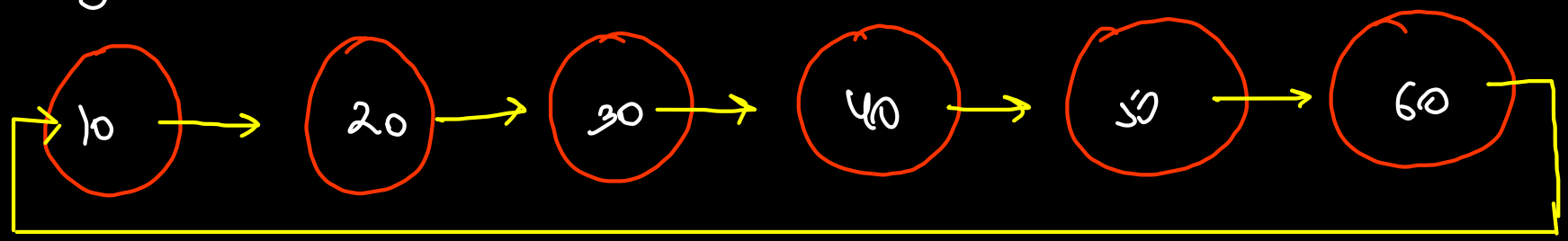
w/o cycle (singly linked list)

false



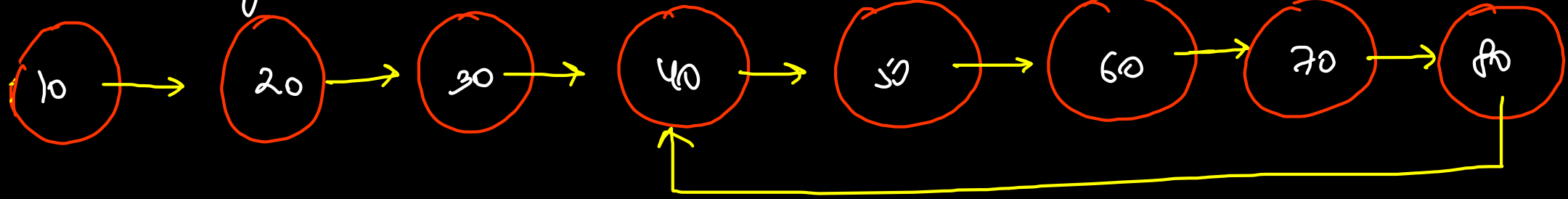
with cycle (circular singly double list)

true

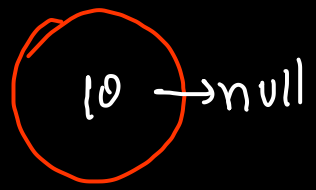


with cycle/ loop

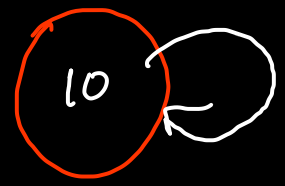
true



Corner case



false



true

null

false

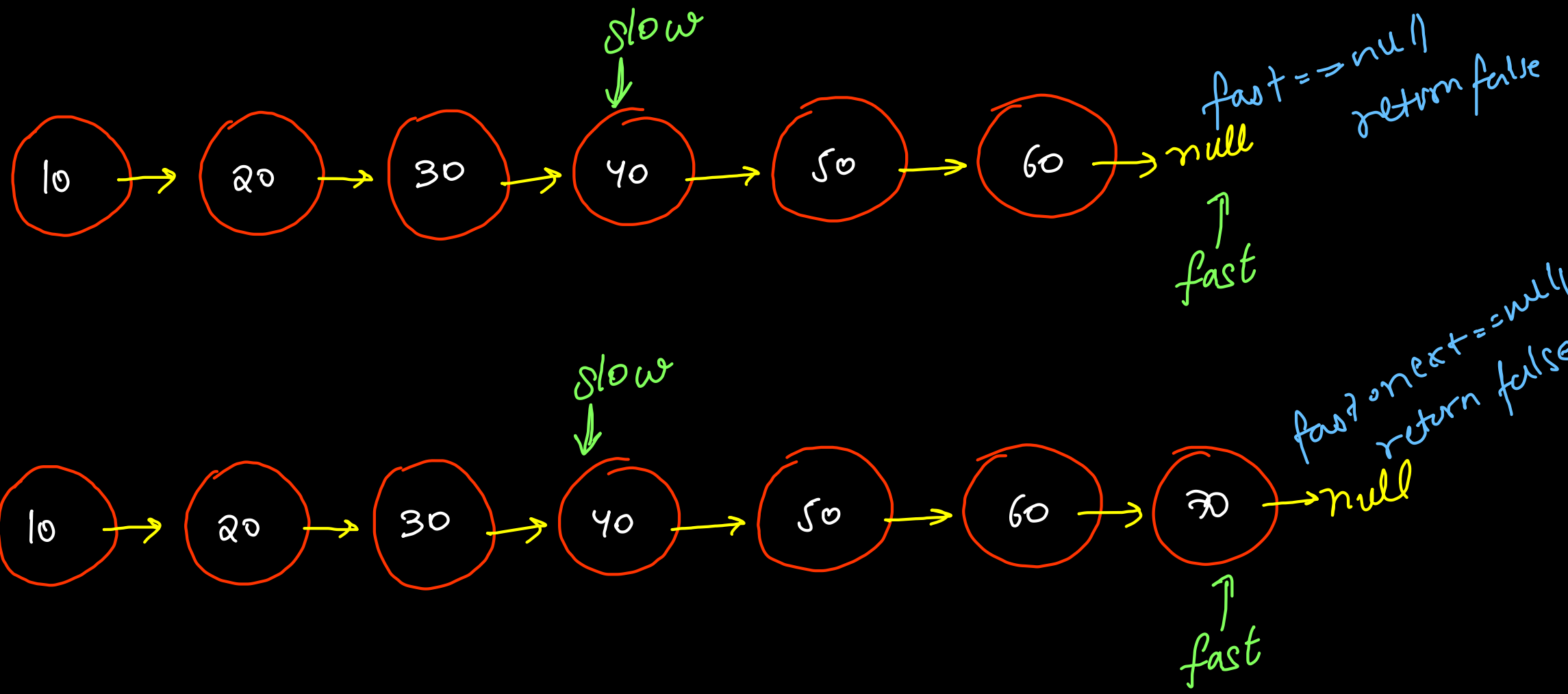
Approach ① Extra Space

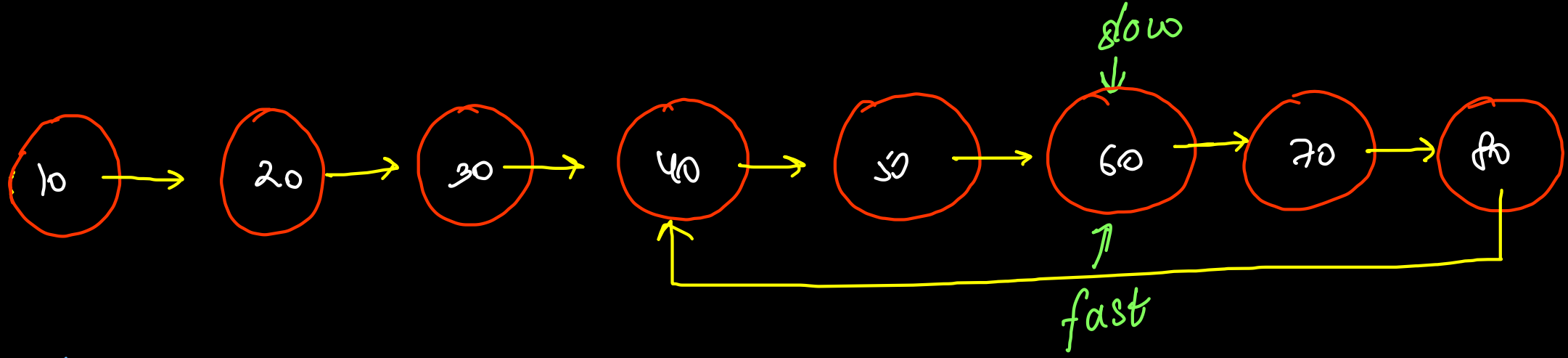
① Traverse & store each node in hashmap

② If any node is already present, return true

③ If reached null, return false

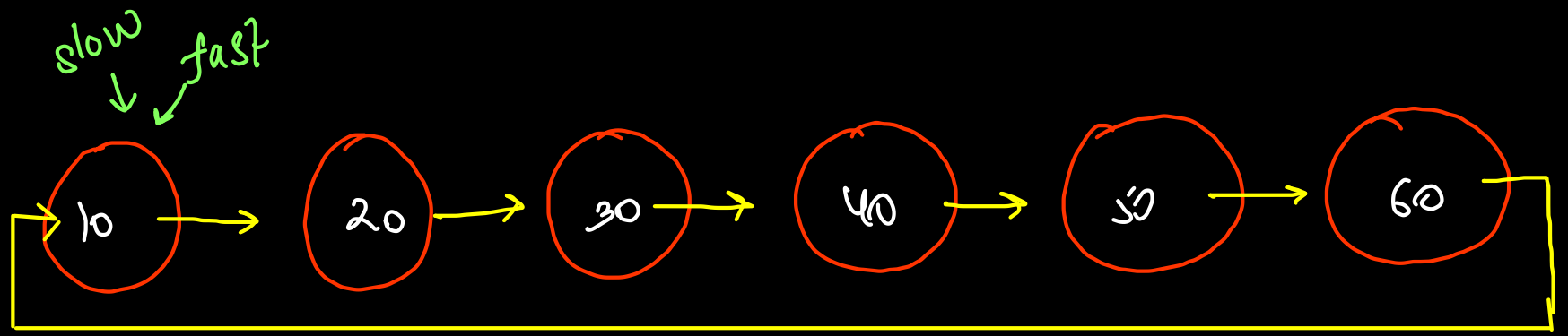
Time ⇒ $O(n)$ Space ⇒ $O(n)$ hashmap


slow
10
20
30
40
50
60
null
fast
fast == null
return false
slow
10
20
30
40
50
60
70
null
fast
fast?.next == null
return false.

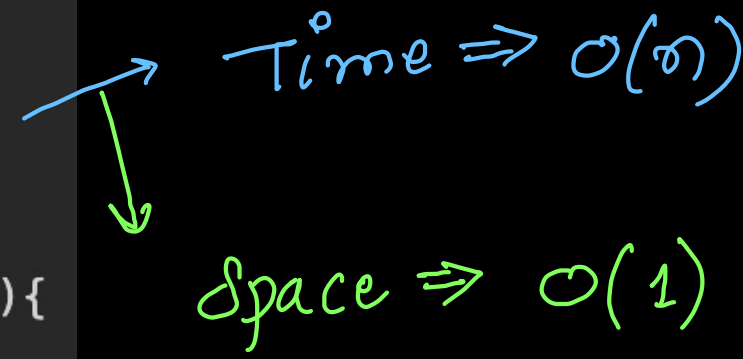


true

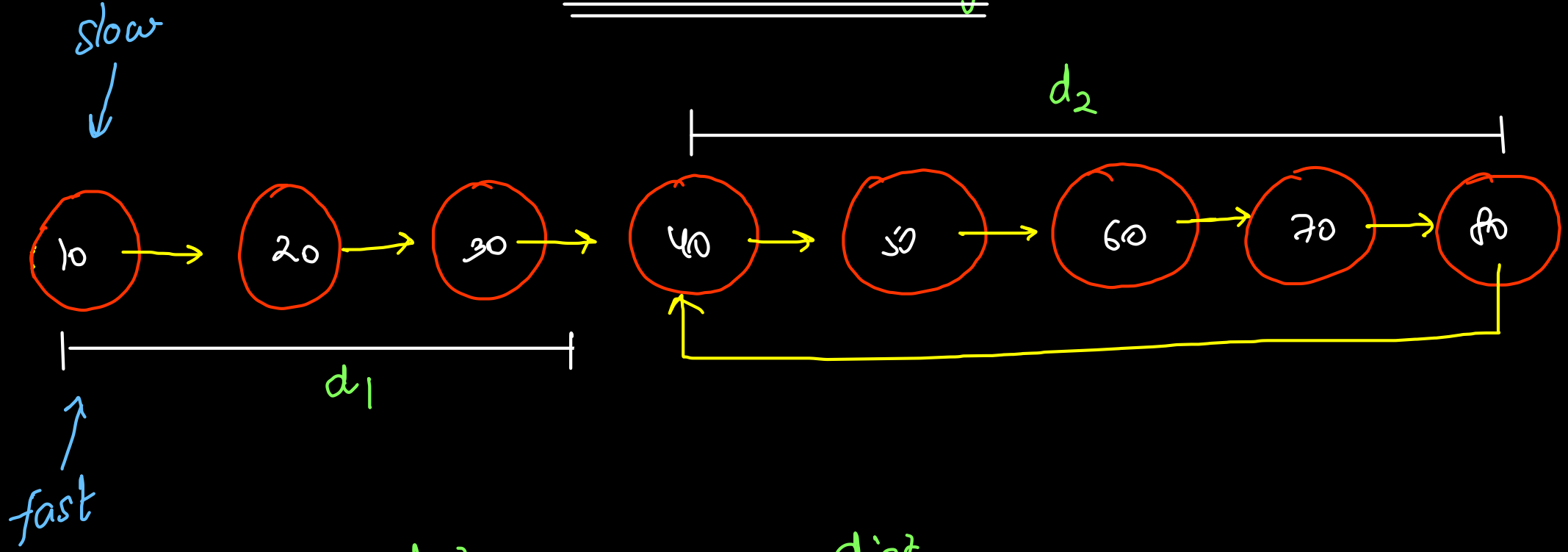
slow == fast ⇒ return true;



```java
public class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode slow = head, fast = head;

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

        return false;
    }
}
```



Mathematical Proof



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

$$2 * (d_1 + n_1' * d_2 + x) = d_1 + n_2 * d_2 + x$$

$$2d_1 + n_1 * d_2 + 2x = d_1 + n_2 d_2 + x$$

$$d_1 + x + (n_1 - n_2) d_2 = 0$$

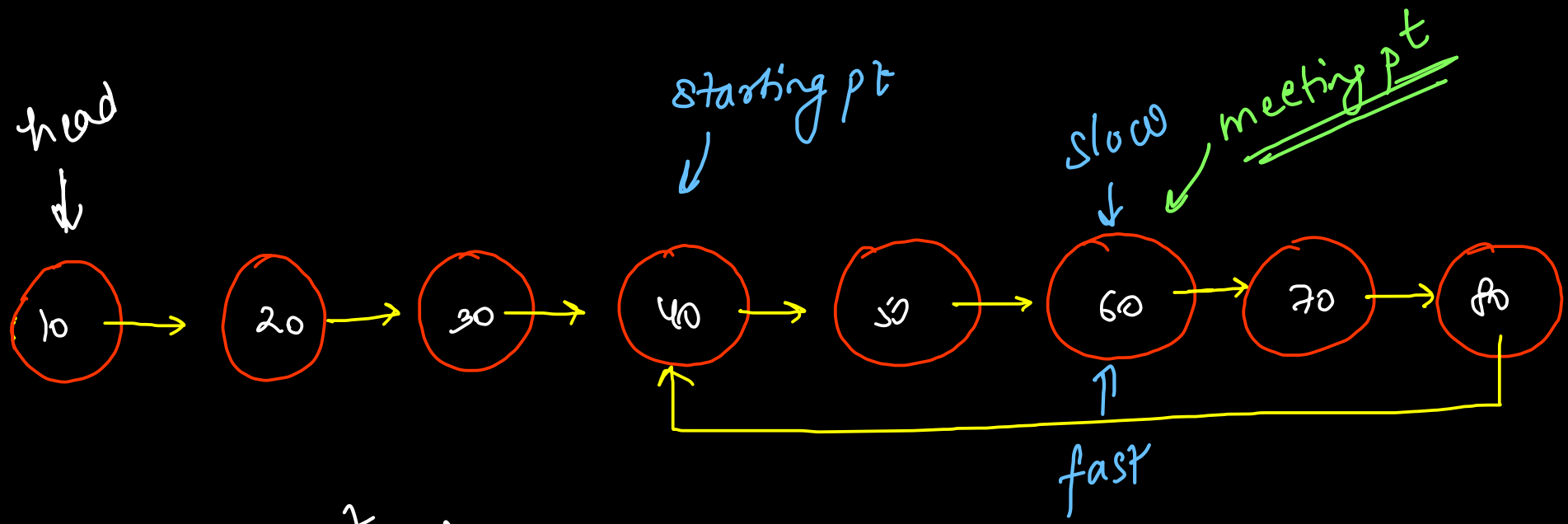
$$d_2 = \frac{d_1 + x}{n_2 - n_1} \qquad n_2 \neq n_1$$

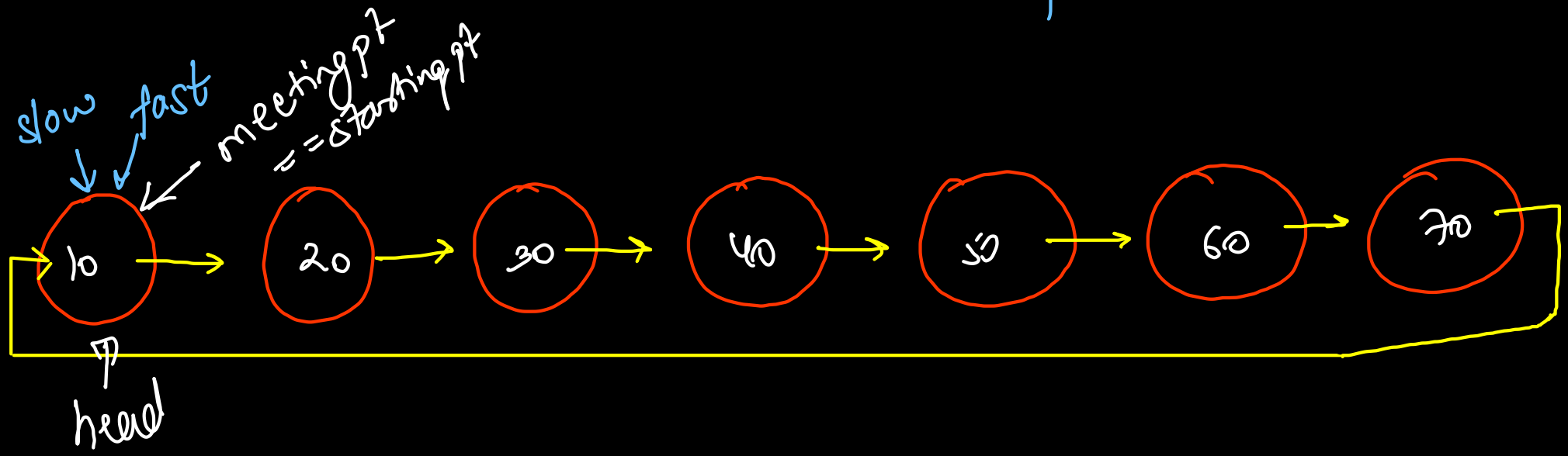
$\Downarrow$

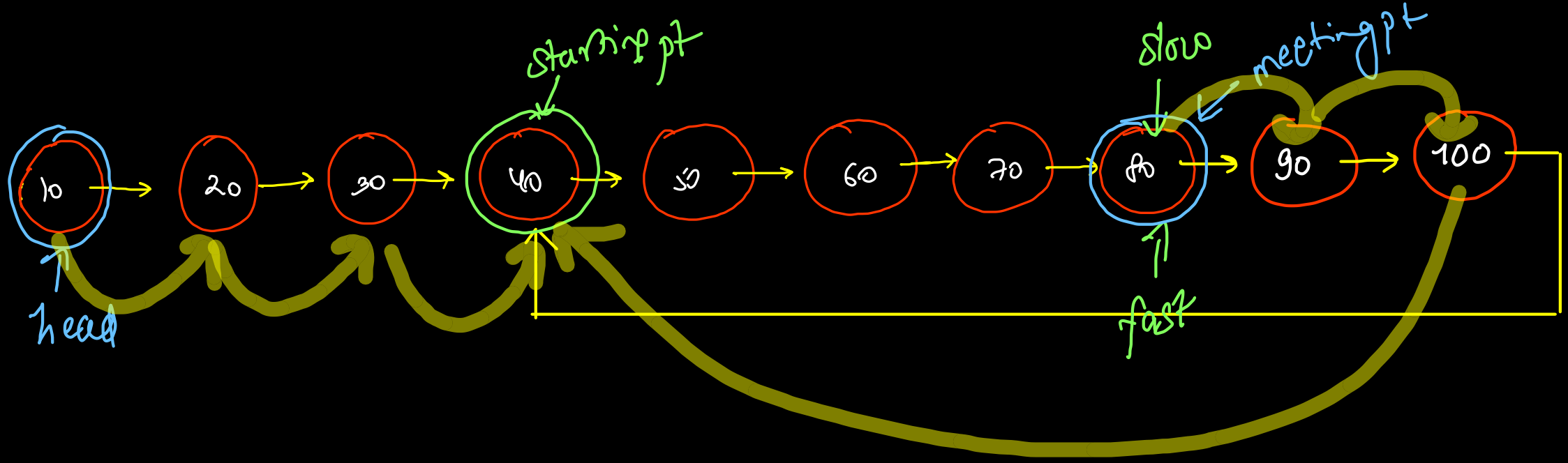
Integral value $\Rightarrow$ meeting pt always exists.

$\Downarrow$ never infinite loop

linked list Cycle - II







distance (b/w head & starting pt)

= distance (b/w meeting pt & starting pt)

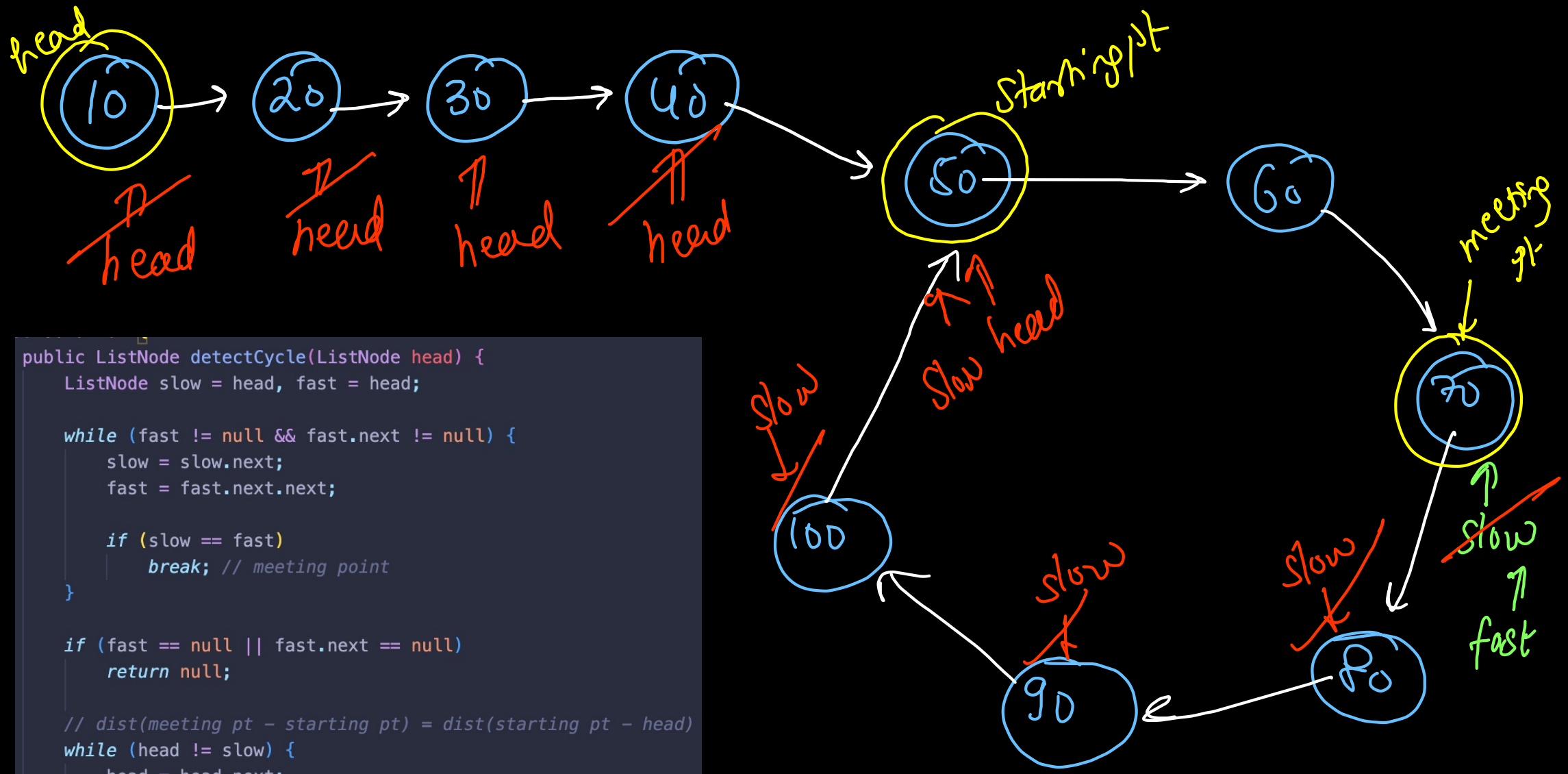
```java
public ListNode detectCycle(ListNode head) {
    ListNode slow = head, fast = head;

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

        if(slow == fast) break; // meeting point
    }

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

    // dist(meeting pt - starting pt) = dist(starting pt - head)
    while(head != slow){
        head = head.next;
        slow = slow.next;
    }

    return slow;  // starting point
}
```

O(n)

O(n)

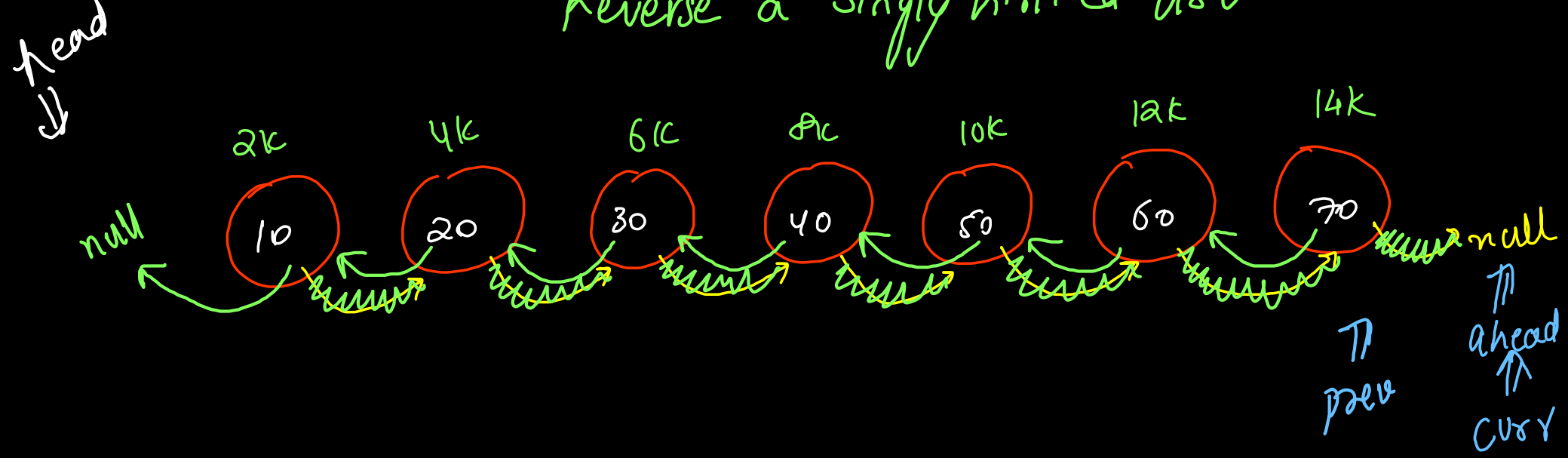
Total time ⇒ O(n)

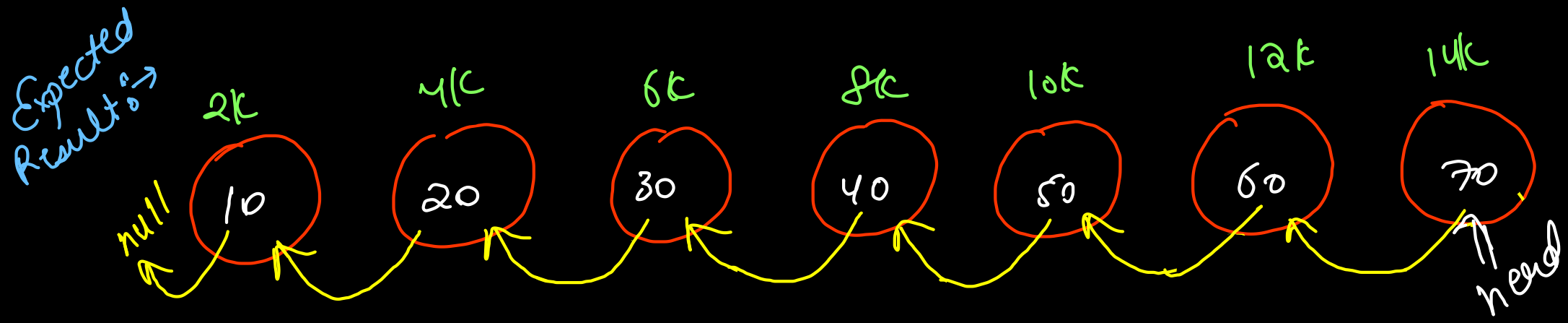
Space ⇒ O(1)




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

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

        if (slow == fast)
            break; // meeting point
    }

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

    // dist(meeting pt - starting pt) = dist(starting pt - head)
    while (head != slow) {
        head = head.next;
        slow = slow.next;
    }

    return slow; // starting point
}
```

Reverse a Singly linked list





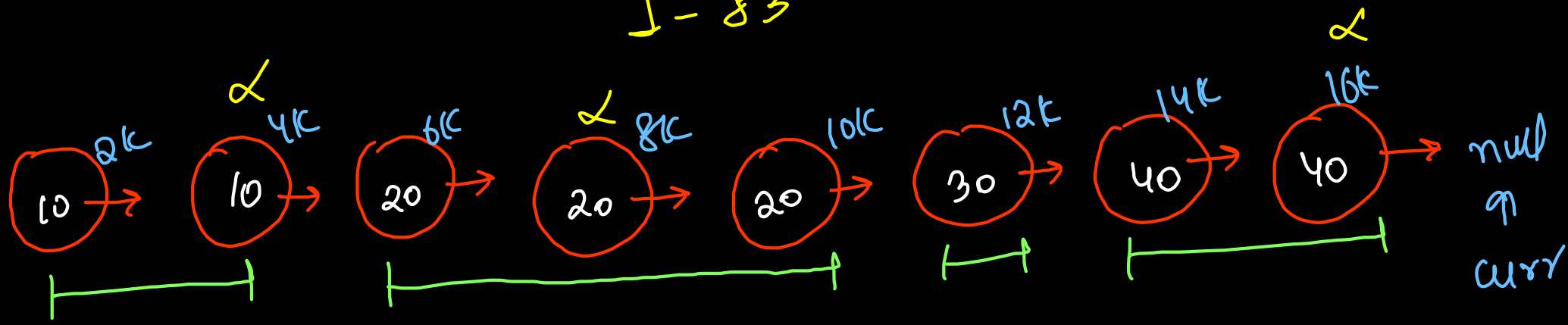
```java
public ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head;

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

    return prev;
}
```

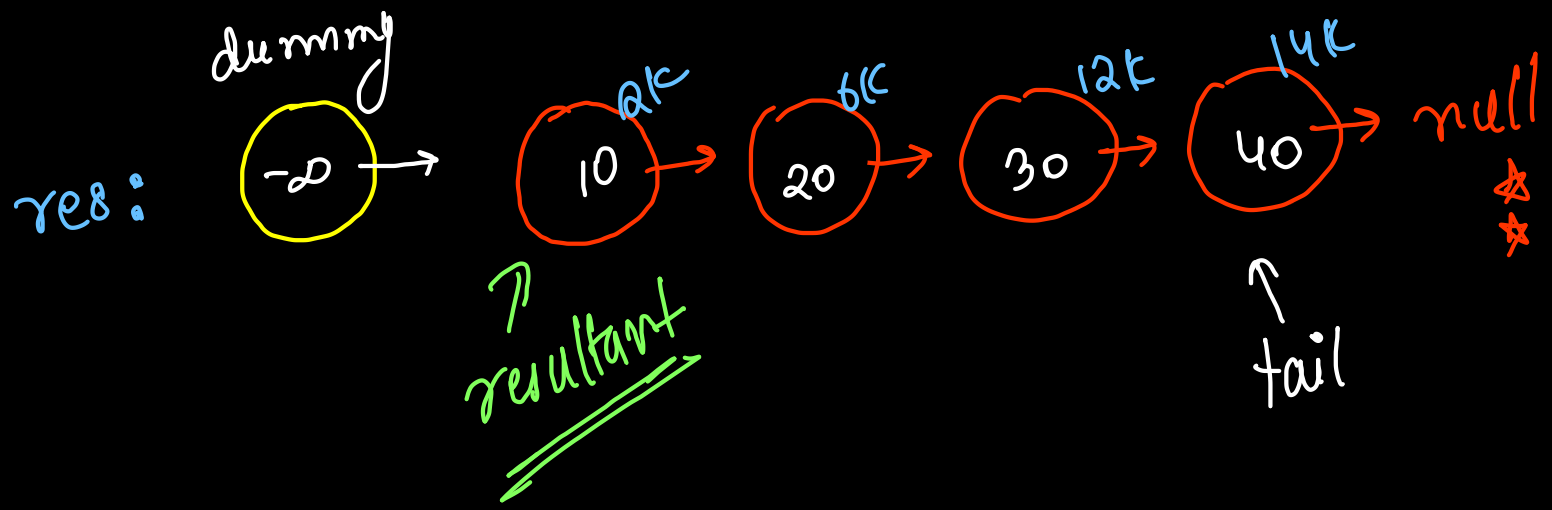
Total time ⇒ O(n)

Space ⇒ O(1)

Remove Duplicates from Sorted List

I - 83



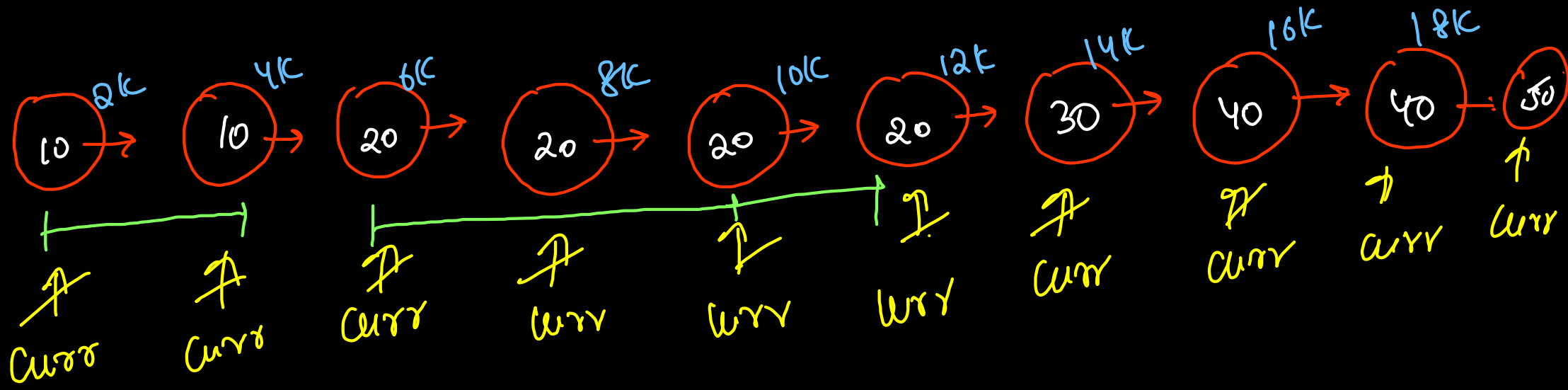


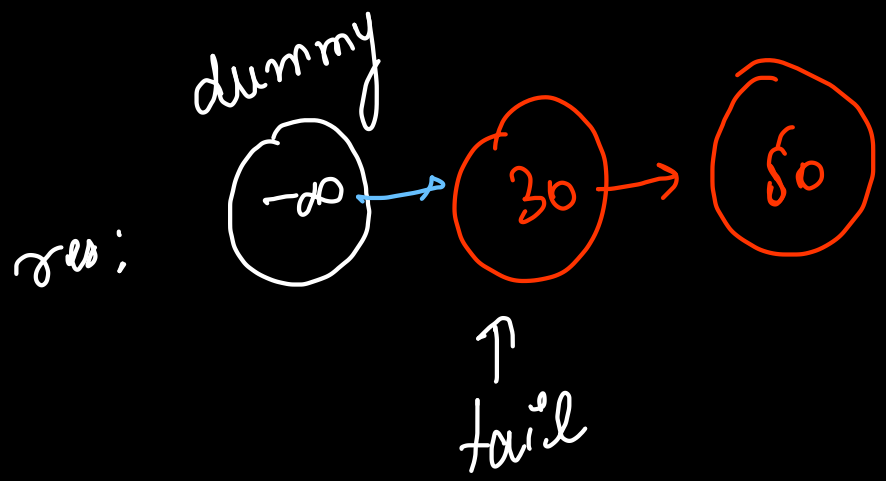
```java
public ListNode deleteDuplicates(ListNode curr) {
    ListNode dummy = new ListNode(Integer.MIN_VALUE);
    ListNode tail = dummy;

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

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

Total time
↳ O(n)

Extra Space
↳ O(1)

II - 82


2K
4K
6K
8K
10K
12K
14K
16K
18K
10
10
20
20
20
20
30
40
40
50
curr
curr
curr
curr
curr
curr
curr
curr
curr
curr


dummy
-40
30
50
tail

ans:

```java
public ListNode deleteDuplicates(ListNode curr) {
    ListNode dummy = new ListNode(Integer.MIN_VALUE);
    ListNode tail = dummy;

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

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

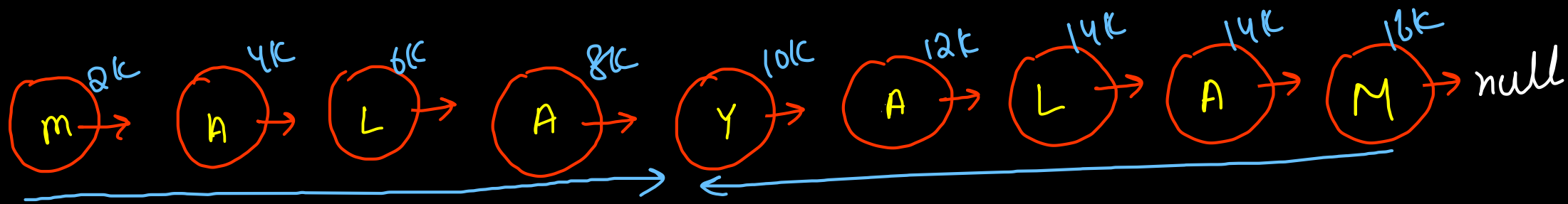
duplicate node (skip all)

unique node (take it)

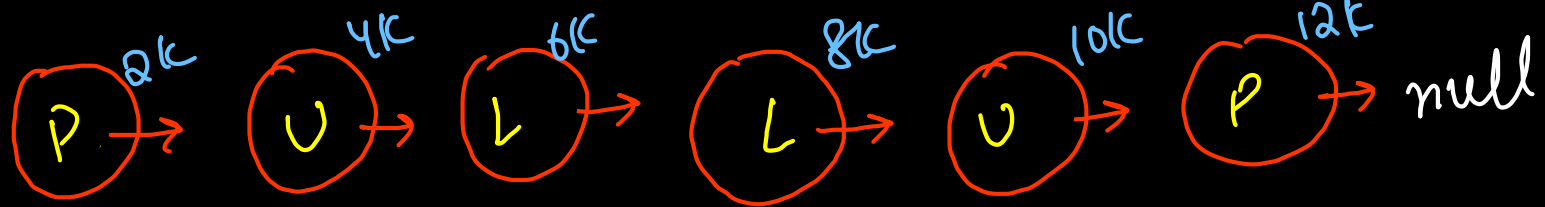
Time ⇒ O(n) linear

Space ⇒ O(1)

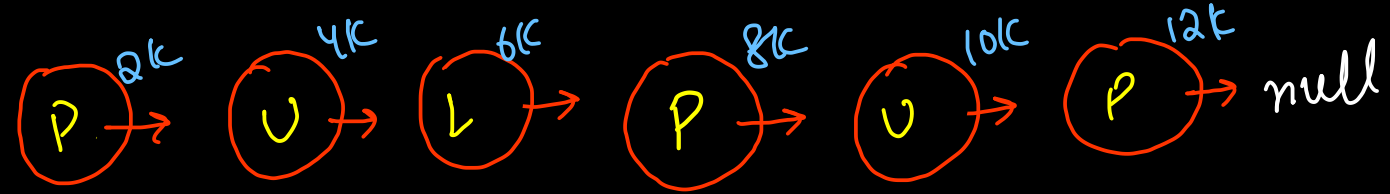
Palindrome LL



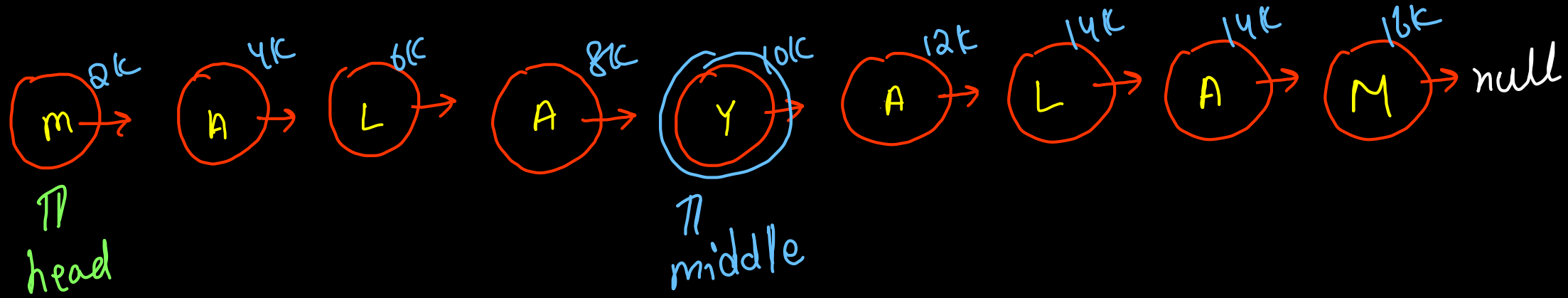
true

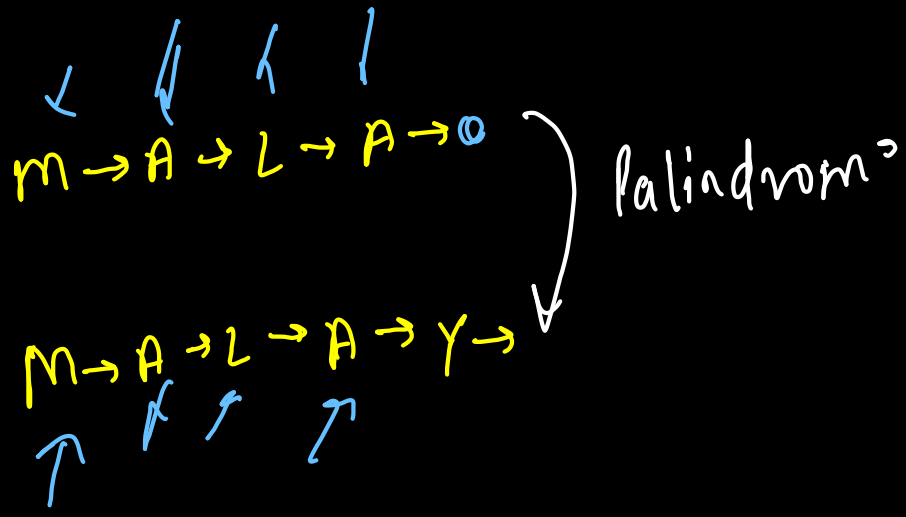


true



false





1. Find middle node
2. Reverse the second half
3. Compare both the halfs.

```java
public boolean isPalindrome(ListNode first) {
    ListNode mid = middle(first);
    ListNode second = reverse(mid);

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

    return true;
}
```

→ O(n)

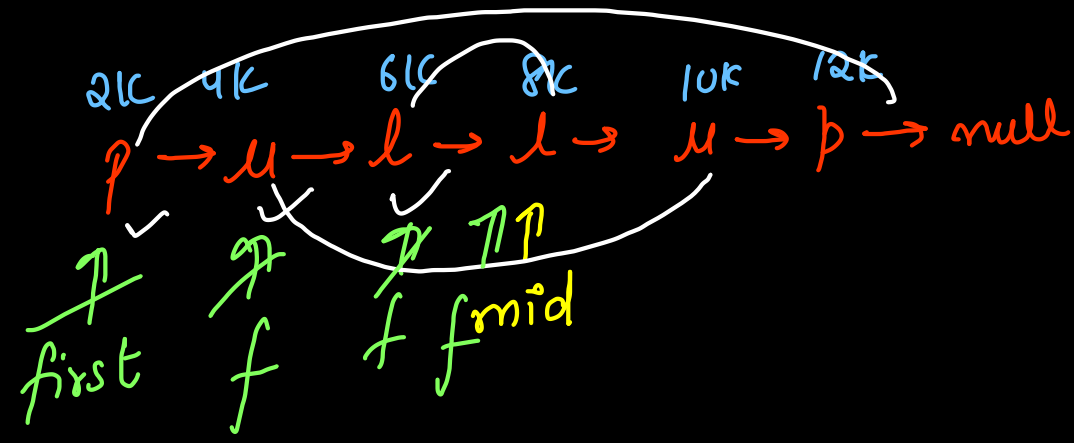
→ O(n/2)

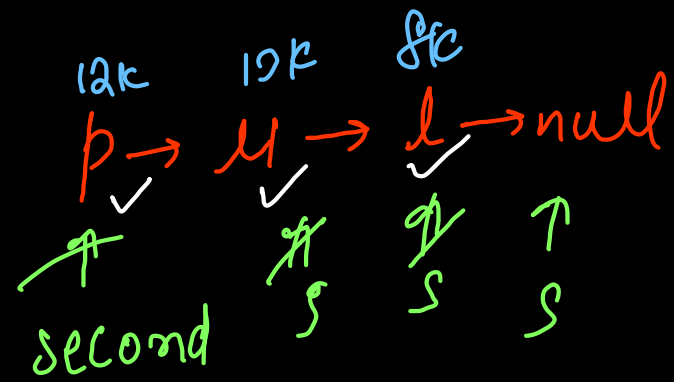
→ O(n/2)

Total time
⇒ O(n)

Space ⇒ O(1)

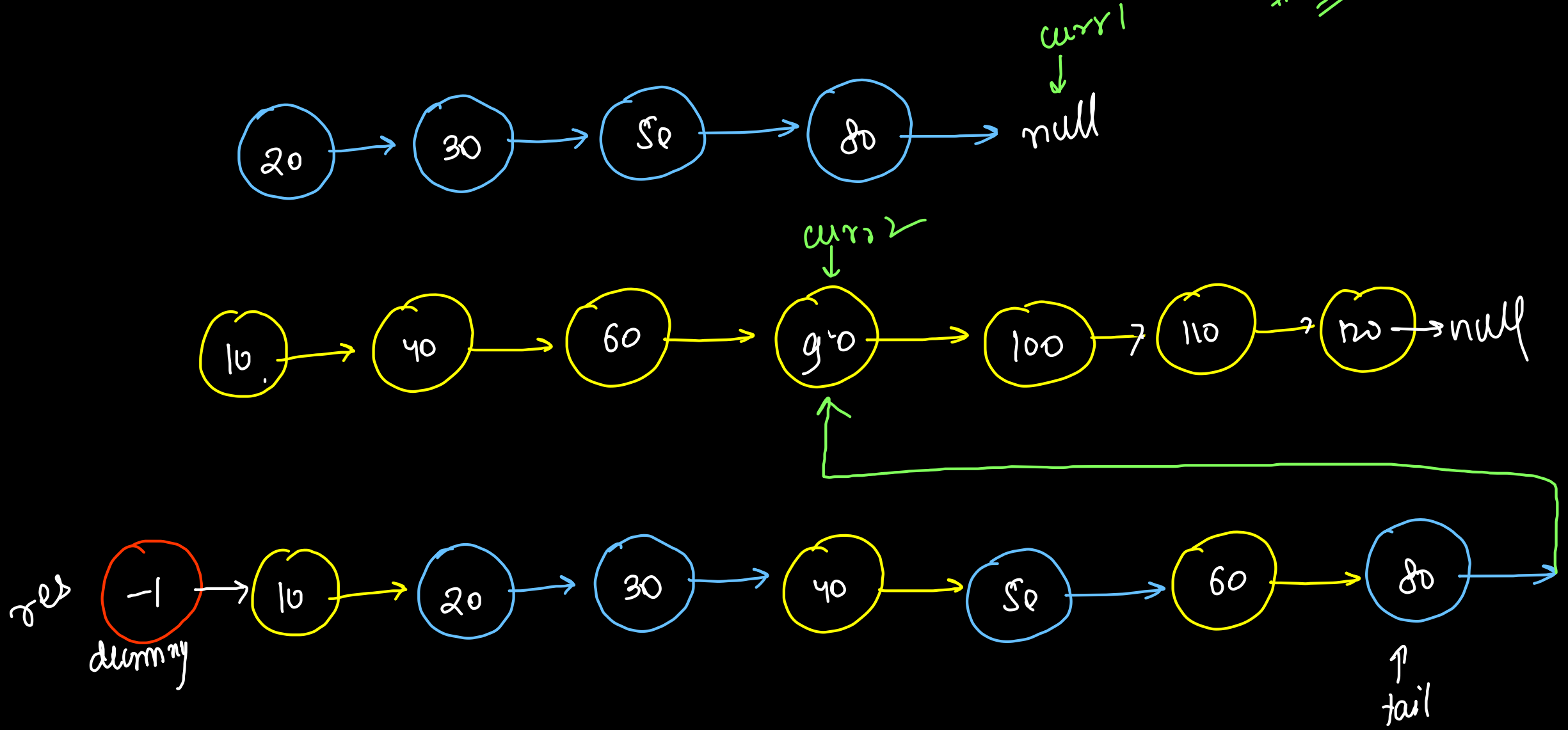
```java
public boolean isPalindrome(ListNode first) {
    ListNode mid = middle(first);
    ListNode second = reverse(mid);

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

    return true;
}
```





Merge 2 Sorted Lists

#In-place



```java
public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(list1 != null && list2 != null){
        if(list1.val < list2.val){
            tail.next = list1;
            tail = tail.next;
            list1 = list1.next;
        } else {
            tail.next = list2;
            tail = tail.next;
            list2 = list2.next;
        }
    }

    if(list1 != null) tail.next = list1;
    else tail.next = list2;
    return dummy.next;
}
```

Time

$\Rightarrow O(n_1 + n_2)$

Space

$\Rightarrow O(1)$

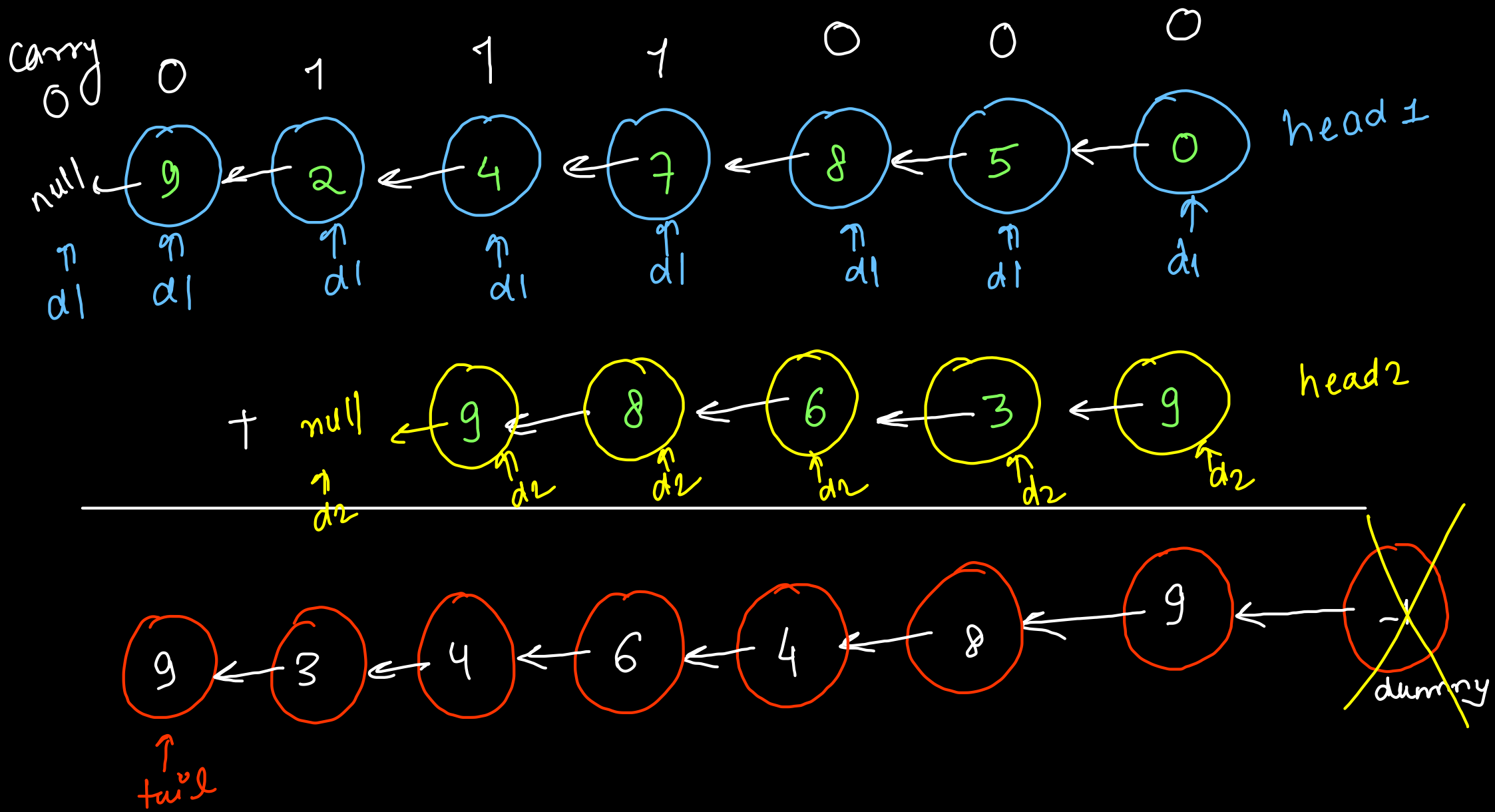
inplace

Why quicksort is preferred for arrays & merge sort is preferred for linkedlist ?

locality of reference

↳ merging → inplace → LL

Addition of Two Linked List



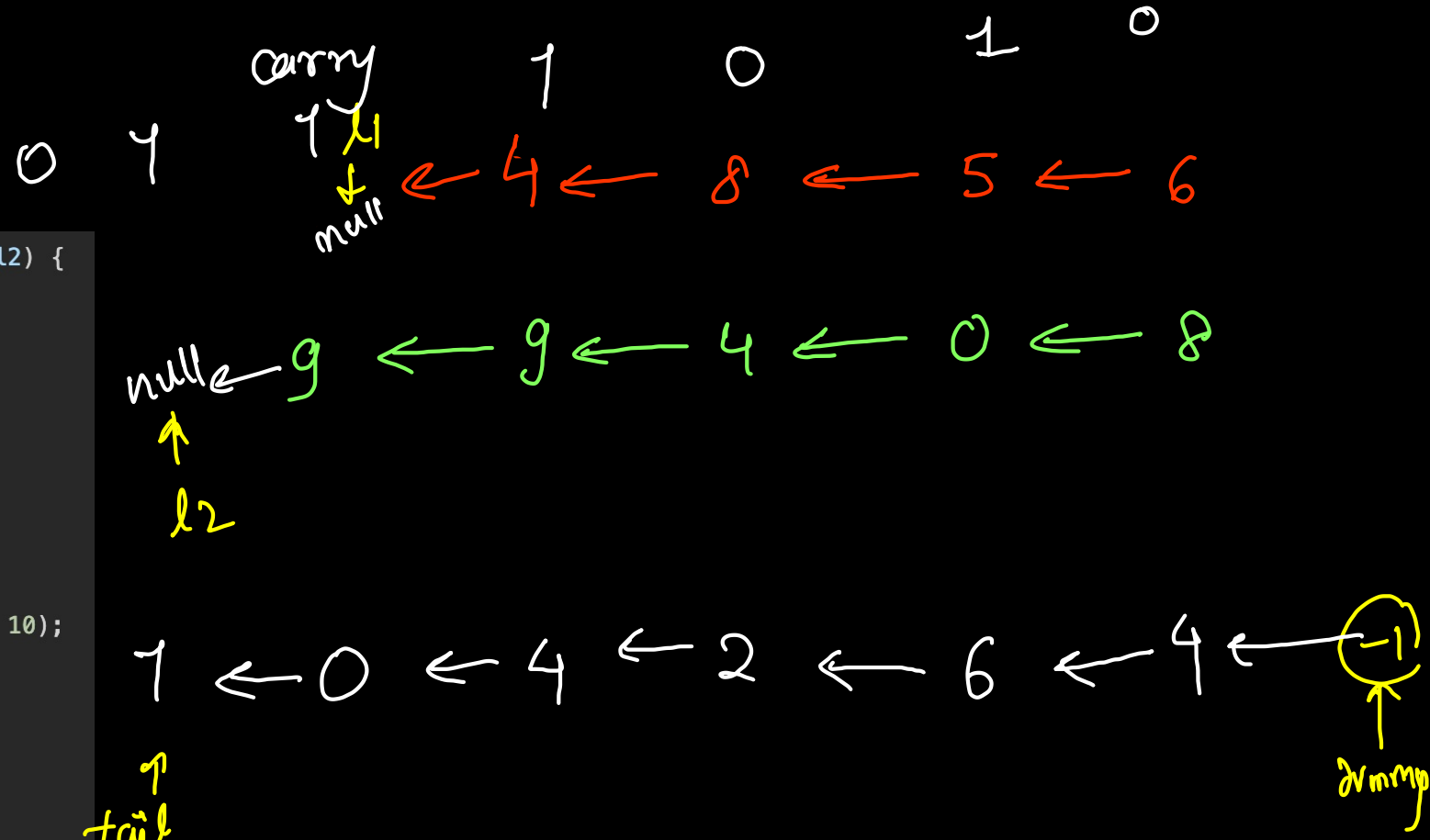
```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;

    int carry = 0;

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

        tail.next = new ListNode((d1 + d2 + carry) % 10);
        carry = (d1 + d2 + carry) / 10;

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

    return dummy.next;
}
```

Time $\Rightarrow O(n_1 + n_2)$

Space $\Rightarrow$

- Input $\rightarrow O(n_1 + n_2)$
- Extra $\rightarrow O(1)$
- Output $\rightarrow O(n)$ {resultant}

```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;

    int carry = 0;

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

        tail.next = new ListNode((d1 + d2 + carry) % 10);
        carry = (d1 + d2 + carry) / 10;

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

    return dummy.next;
}
```



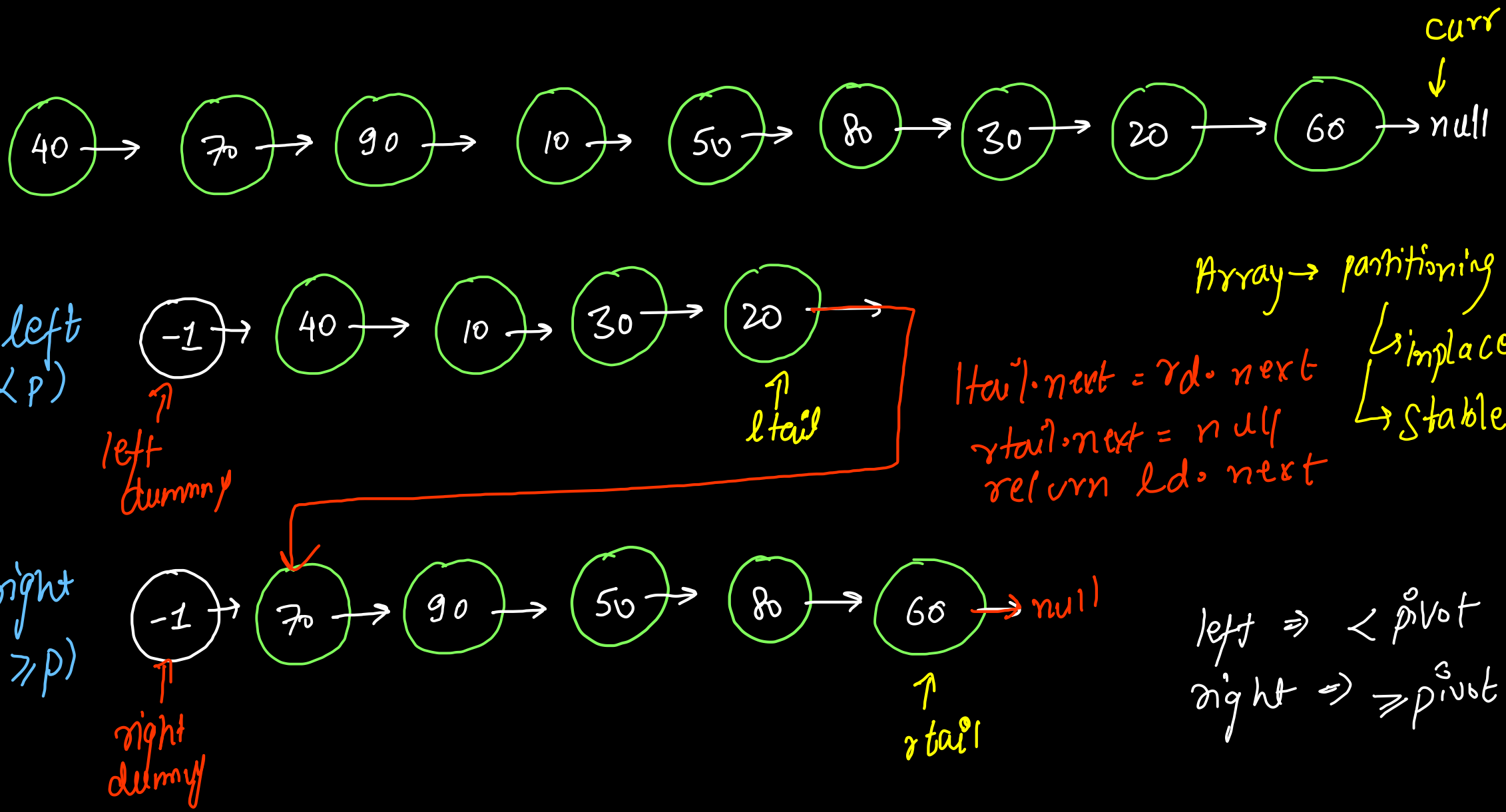
```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    l1 = reverse(l1);
    l2 = reverse(l2);

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

    int carry = 0;

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

        tail.next = new ListNode((d1 + d2 + carry) % 10);
        carry = (d1 + d2 + carry) / 10;

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

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

Add Two Linked List Nos

Leetcode 445

Partition a linked list

pivot = 50



```java
public ListNode partition(ListNode curr, int pivot) {
    ListNode ldummy = new ListNode(-1);
    ListNode rdummy = new ListNode(-1);
    ListNode left = ldummy, right = rdummy;

    while(curr != null){
        if(curr.val < pivot){
            left.next = curr;
            left = curr;
            curr = curr.next;
        } else {
            right.next = curr;
            right = curr;
            curr = curr.next;
        }
    }

    left.next = rdummy.next;
    right.next = null;
    return ldummy.next;
}
```

Time ⇒ O(n)

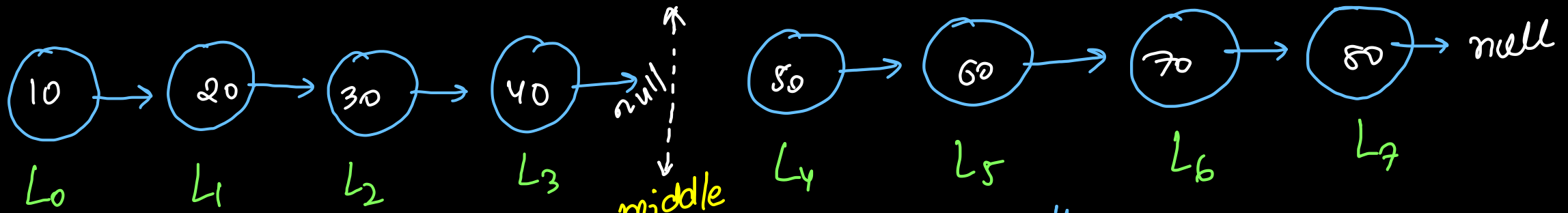
Space ⇒ O(1)

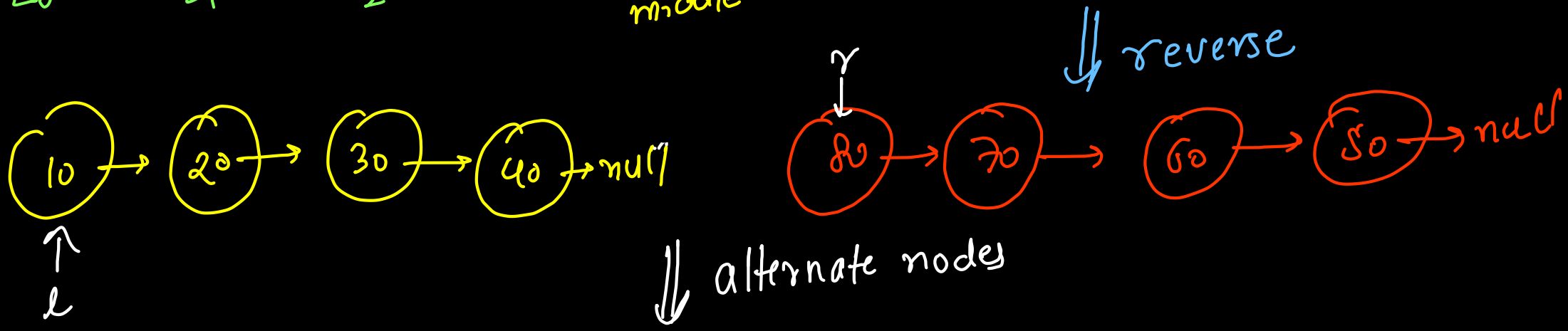
⇒ Inplace

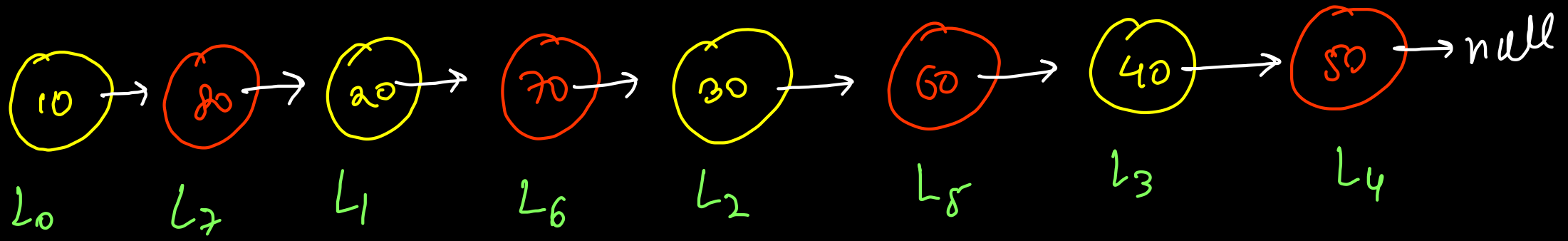
⇒ Stable

(relative order is preserved)

Reorder list LC 143







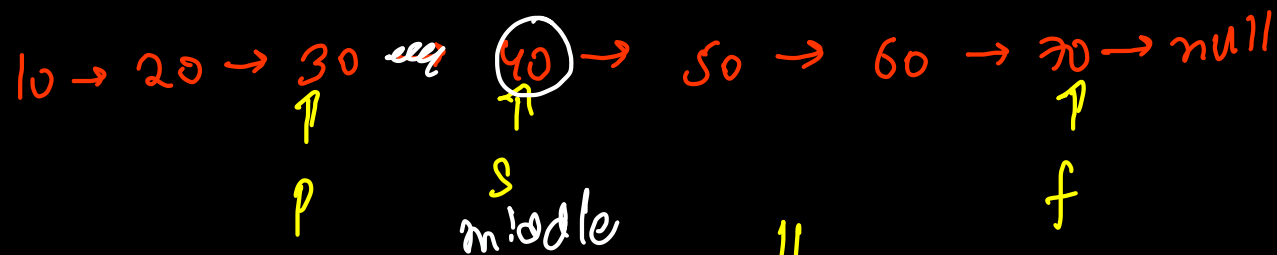
⭐ Procedure

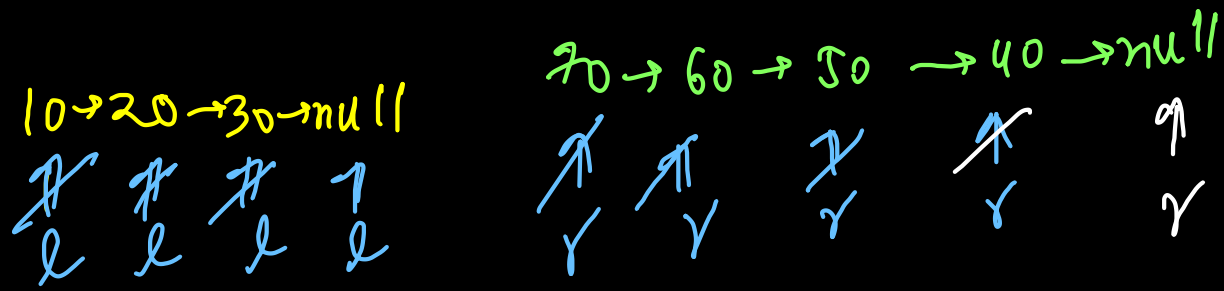
1. get middle → leftTail.next = null
2. reverse the right part
3. take alternate nodes from left & right half

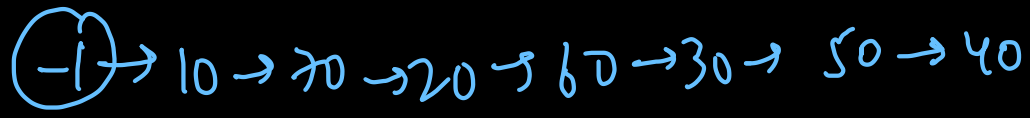
```java
public ListNode middle(ListNode head){
    ListNode prev = null, slow = head, fast = head;

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

    prev.next = null;
    return slow;
}
```

Time
⇒ O(n)

Space
⇒ O(1)








```java
public void reorderList(ListNode head) {
    if(head == null || head.next == null) return;

    // Get the Middle Node
    ListNode mid = middle(head);

    // Reverse the Right Half
    ListNode l1 = head;
    ListNode l2 = reverse(mid);

    // Take Alternate Nodes
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(l1 != null || l2 != null){
        if(l1 != null){
            tail.next = l1;
            tail = l1;
            l1 = l1.next;
        }

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

    head = dummy.next;
}
```

Merge K Sorted Lists

arr[0] = 10 → 50 → 70 → null

arr[1] = 20 → 40 → null

arr[2] = 55 → null

arr[3] = 45 → 65 → null

arr[4] = null

arr[5] = 30 → 35 → 90 → 95 → null

arr.length = 6

⇒ resultant

10 → 20 → 30 → 35 → 40 → 45 → 50 → 65 → 70 → 90 → 95

Approach 1)

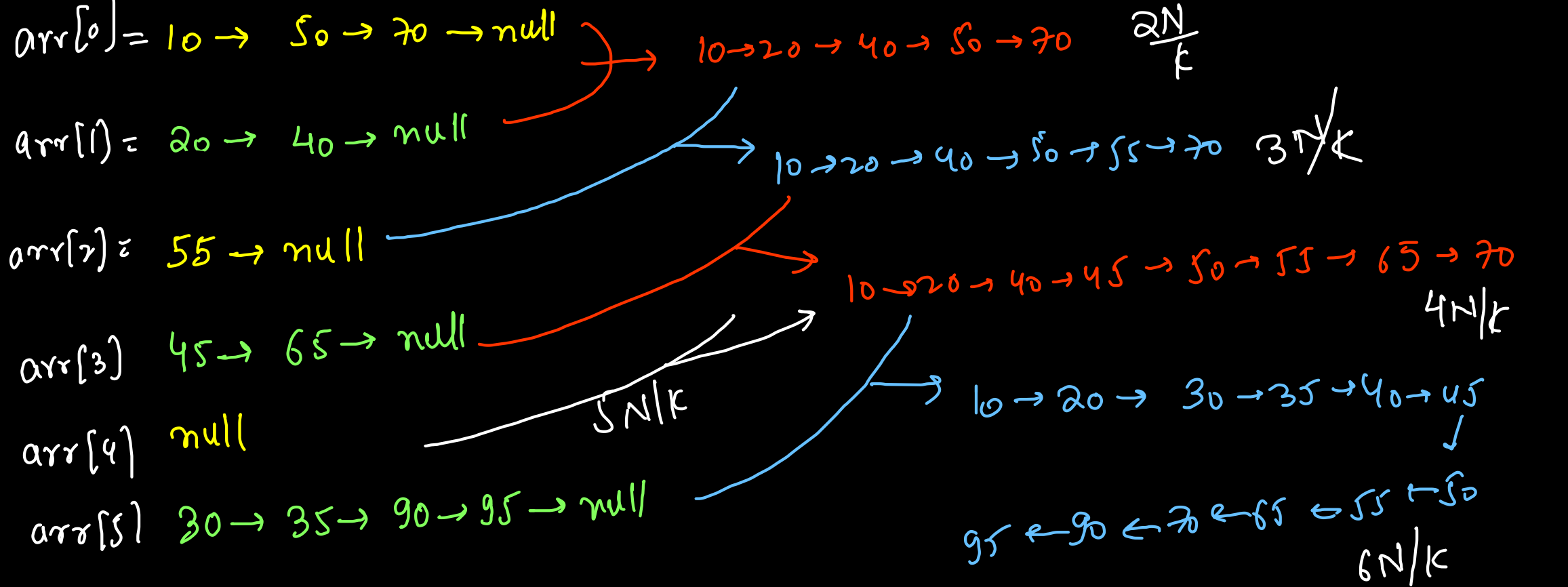
Take minm out of all linkedlists

O(N*K)

Approach 2)

Merging two at a time.

O(N*K)



req linkedlist = N

onelinkedlist = N/k

Total time = $\frac{N}{k}(1+2+3+\cdots k)$

$= \frac{N}{k} * k^2 = N \times k$ (10×10)

```java
public ListNode merge(ListNode list1, ListNode list2){
    ListNode dummy = new ListNode(-1);
    ListNode tail = dummy;

    while(list1 != null && list2 != null){
        if(list1.val < list2.val){
            tail.next = list1;
            tail = tail.next;
            list1 = list1.next;
        } else {
            tail.next = list2;
            tail = tail.next;
            list2 = list2.next;
        }
    }

    if(list1 != null) tail.next = list1;
    else tail.next = list2;
    return dummy.next;
}

public ListNode mergeKLists(ListNode[] arr) {
    ListNode res = null;
    for(ListNode head: arr){
        res = merge(res, head);
    }
    return res;
}
```

Total time $\Rightarrow O(N \times K)$ (total number of nodes, no. of linked list)

Space $\Rightarrow O(1)$ extra space

arr[0] = 10 → 50 → 70 → null

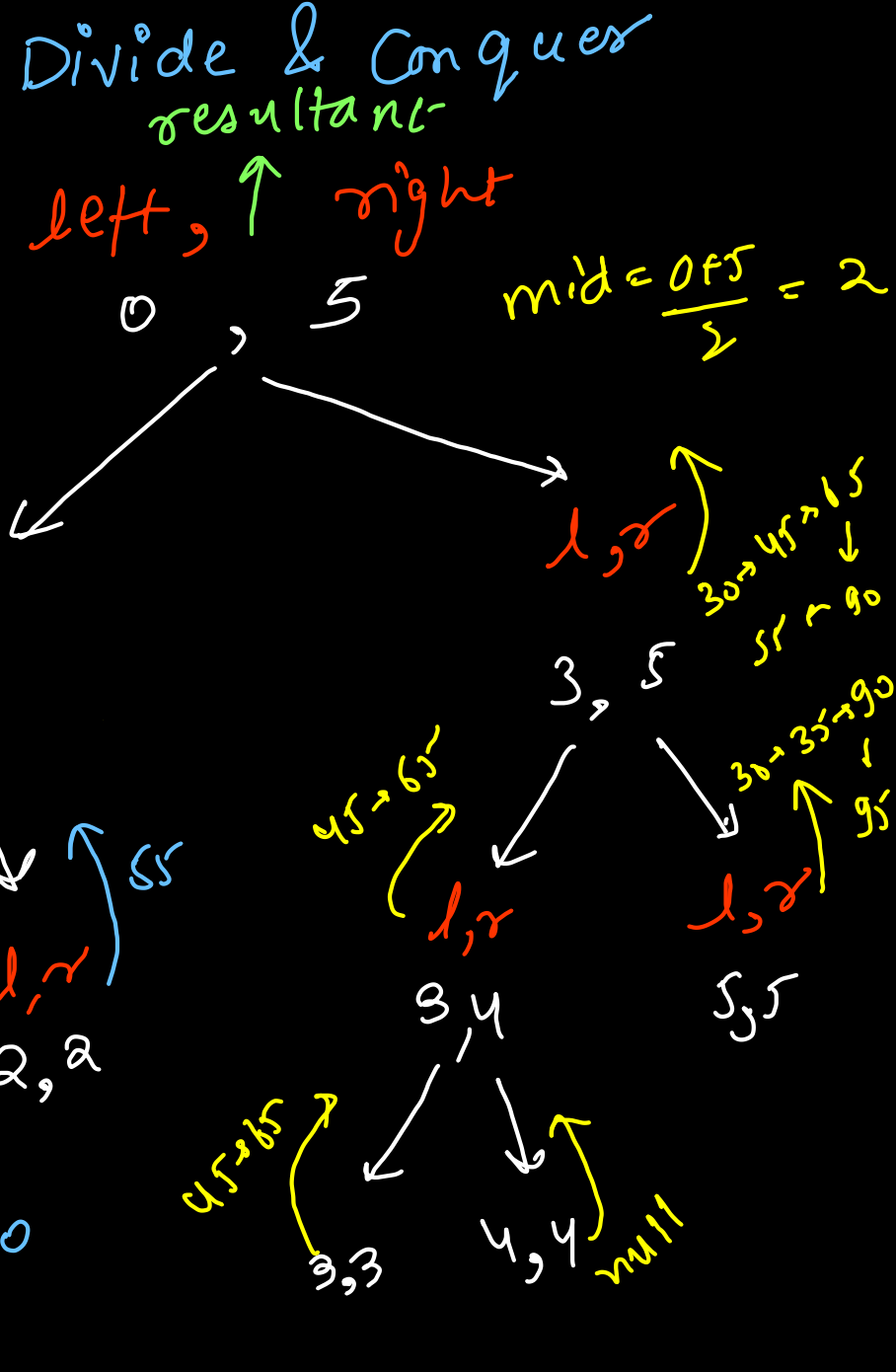
arr[1] = 20 → 40 → null

arr[2] = 55 → null

arr[3] 45 → 65 → null

arr[4] null

arr[5] 30 → 35 → 90 → 95 → null



```java
public ListNode helper(ListNode[] arr, int left, int right){
    if(left == right) return arr[left];
    int mid = (left + right) / 2;
    ListNode l1 = helper(arr, left, mid);
    ListNode l2 = helper(arr, mid + 1, right);
    return merge(l1, l2);
}

public ListNode mergeKLists(ListNode[] arr) {
    if(arr.length == 0) return null;
    return helper(arr, 0, arr.length - 1);
}
```

→ base case (only 1 list)

faith

↳ meeting expectations (postorder work)

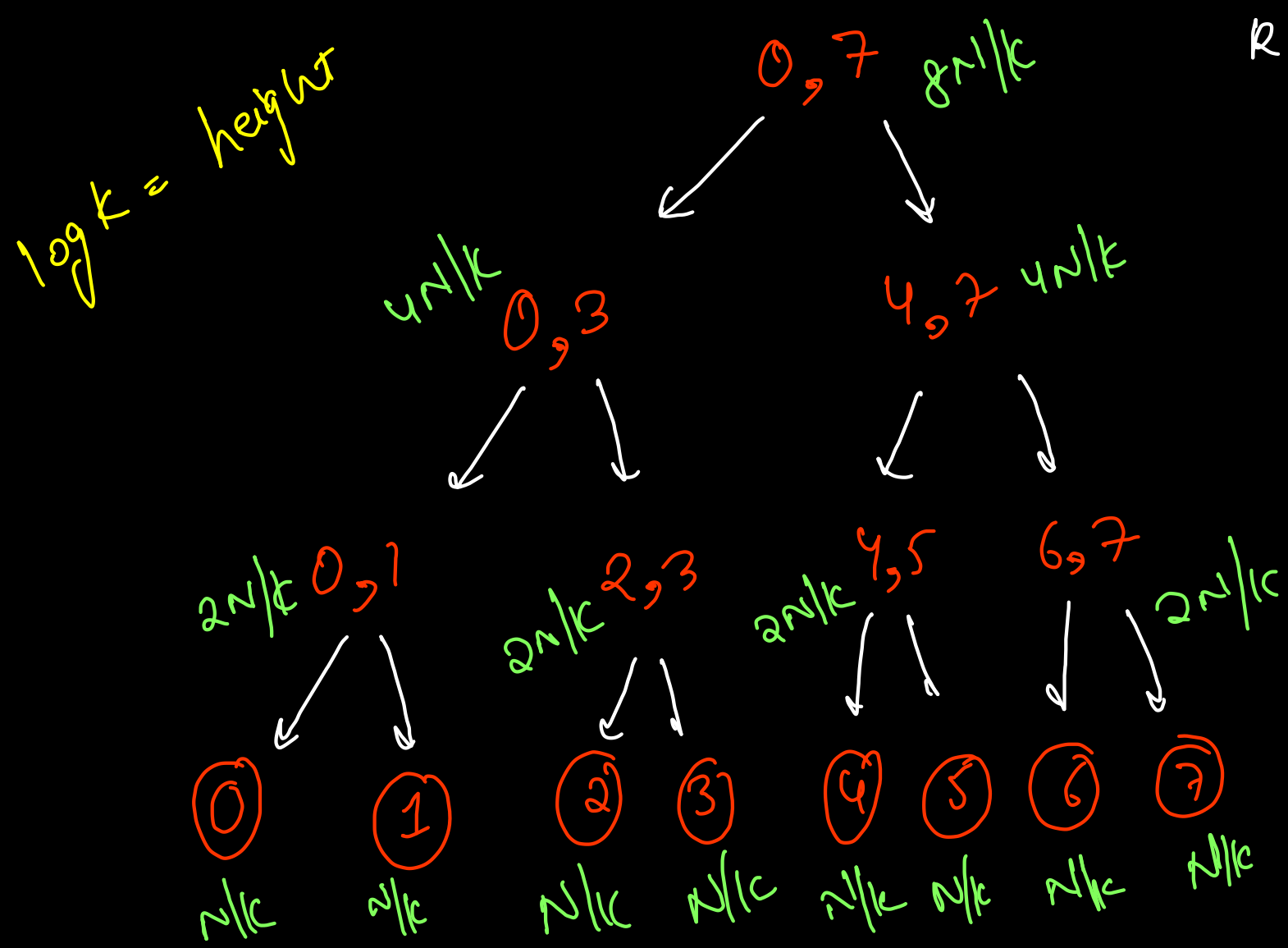
when k = 0 (no lists)

↳ divide & conquer

Time ⇒ O(N)

Space

↳ Recursive call stack Space ≠ O(log k)



$K = 8$ N = total no. of nodes

$n = N/K$

↳ 1 list size

Total time

$$= \frac{N}{K}\left(1+2+4+\cdots K\right)$$

$$= \frac{N}{K} * 2^{\log_2 K}$$

$$= \frac{N}{K} * K$$

$$= O(N)$$

Codeforces

⇒ 2-3 din

⇒ Peer connect pressure

Service Based

↳ cheap labour

Coder ∝

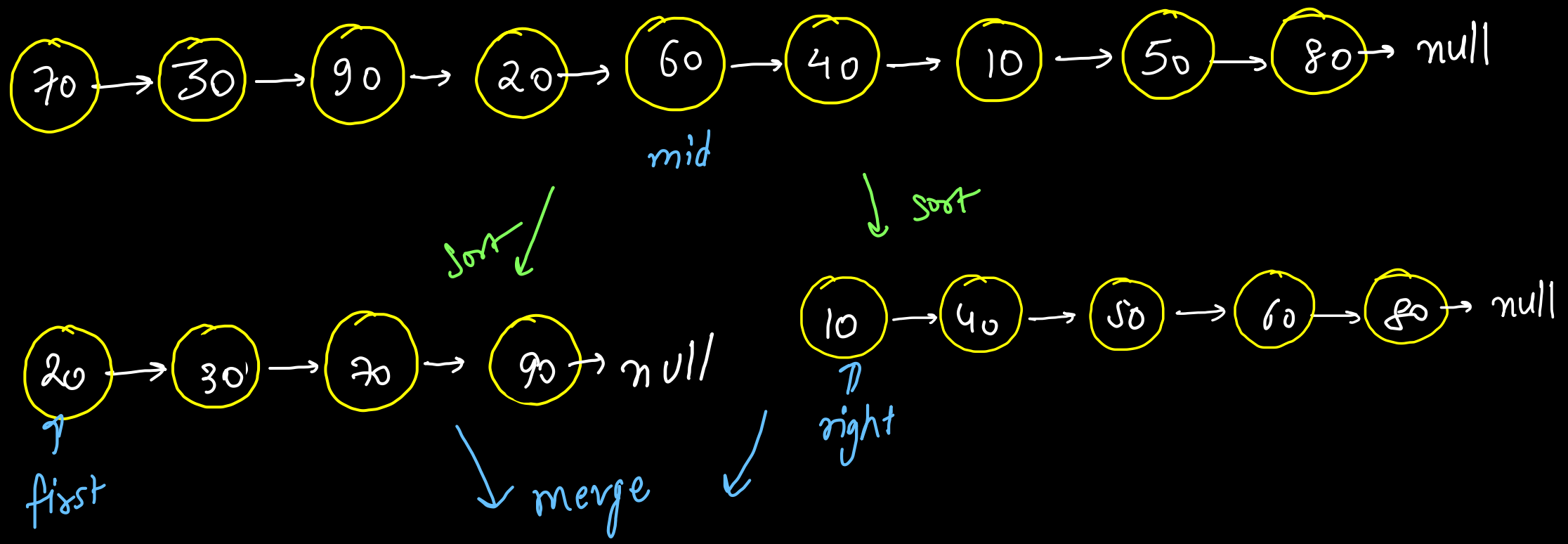
Recoding FJP-2 / JSP-4

Advanced

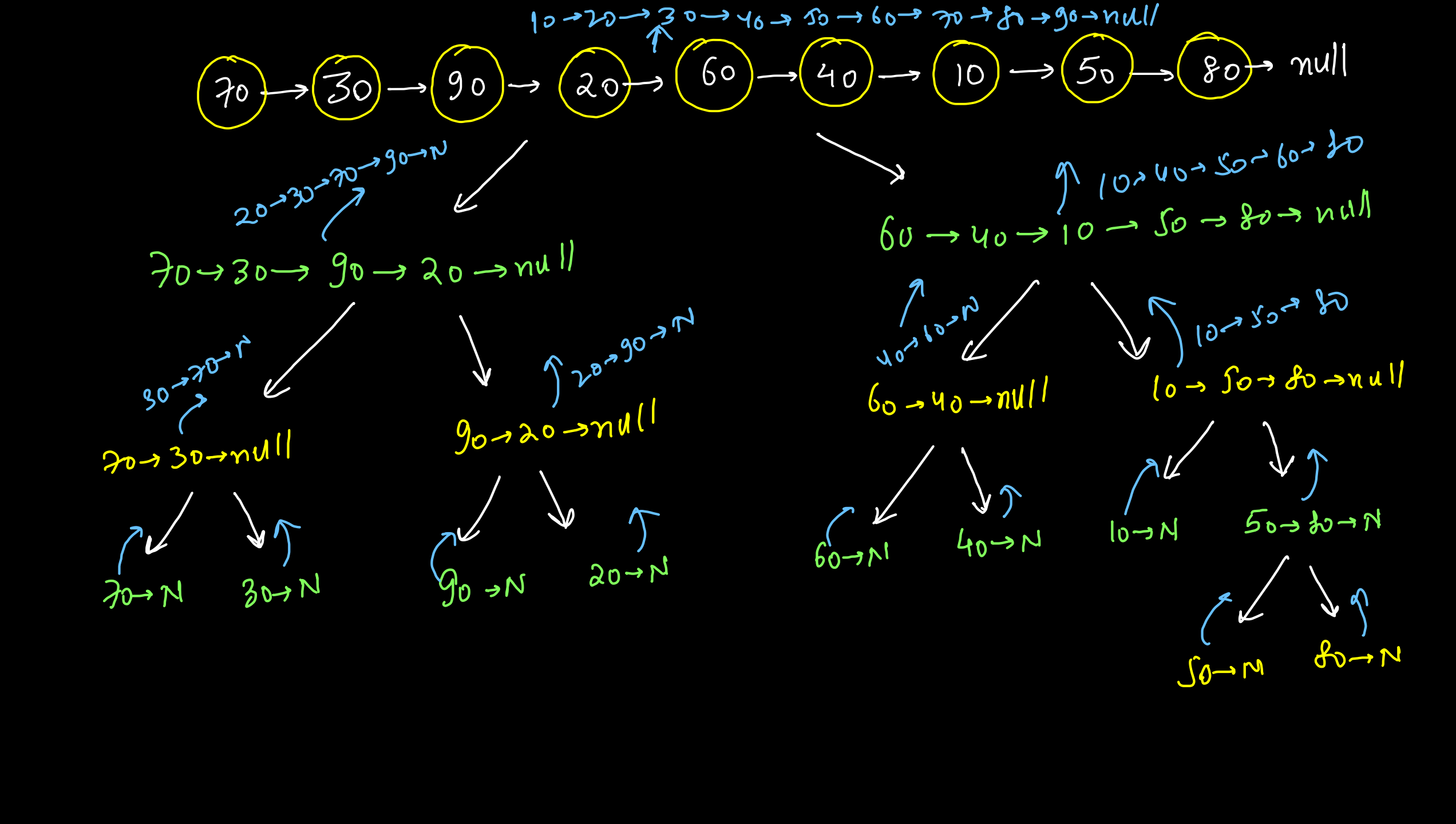
DSA 2.0

Scratch to Adv

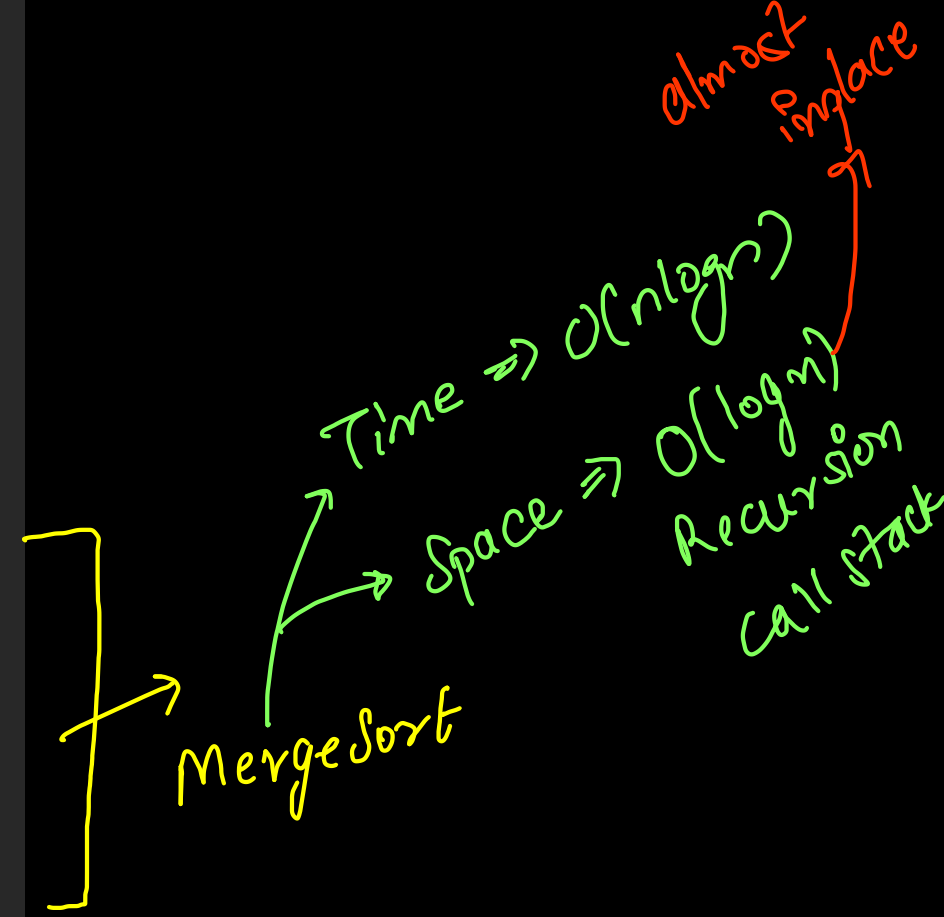
Sort Singly linked List
Merge Sort



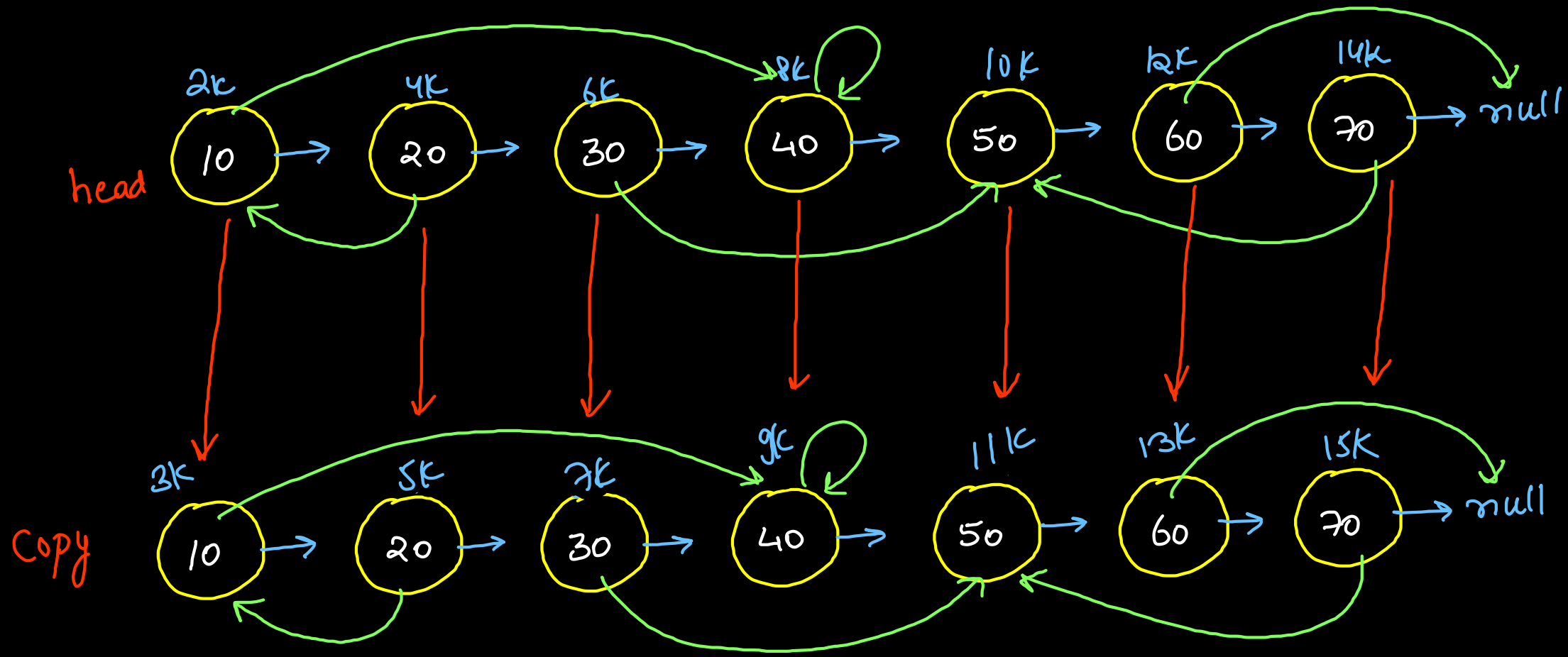
10 → 20 → 30 → 40 → 50 → 60 → 70 → 80 → 90

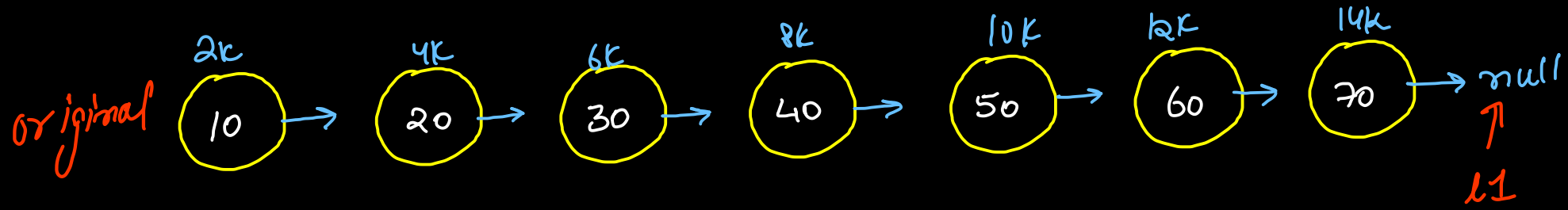

10→20→30→40→50→60→70→80→90→null
70→30→90→20→60→40→10→50→80→null
20→30→70→90→N
70→30→90→20→null
30→70→N
70→30→null
20→90→N
90→20→null
70→N
30→N
90→N
20→N
10→40→50→60→70
60→40→10→50→80→null
40→10→N
60→40→null
10→50→80
10→50→80→null
60→N
40→N
10→N
50→80→N
50→N
80→N

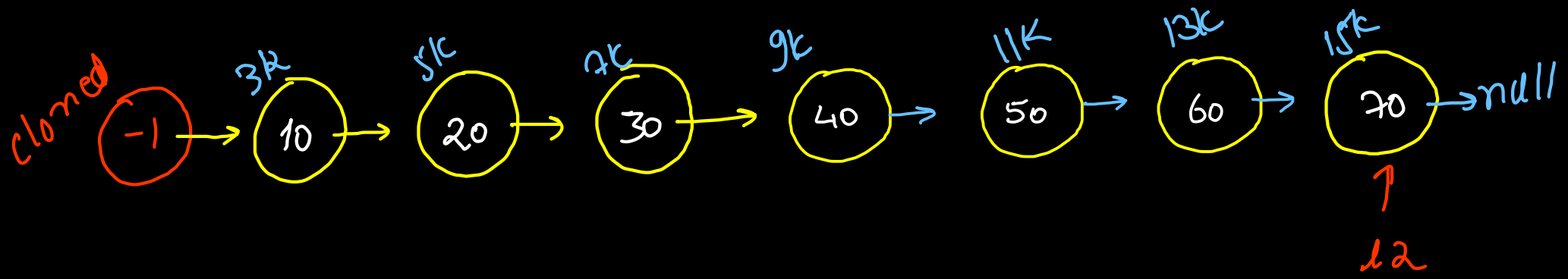
```java
public ListNode middle(ListNode head){
    if(head == null || head.next == null) return head;
    ListNode slow = head, fast = head, prev = null;

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

    prev.next = null;
    return slow;
}

public ListNode sortList(ListNode head) {
    if(head == null || head.next == null) return head;
    ListNode mid = middle(head);
    ListNode l1 = sortList(head);
    ListNode l2 = sortList(mid);
    return merge(l1, l2);
}
```



Copy List with Random Pointer

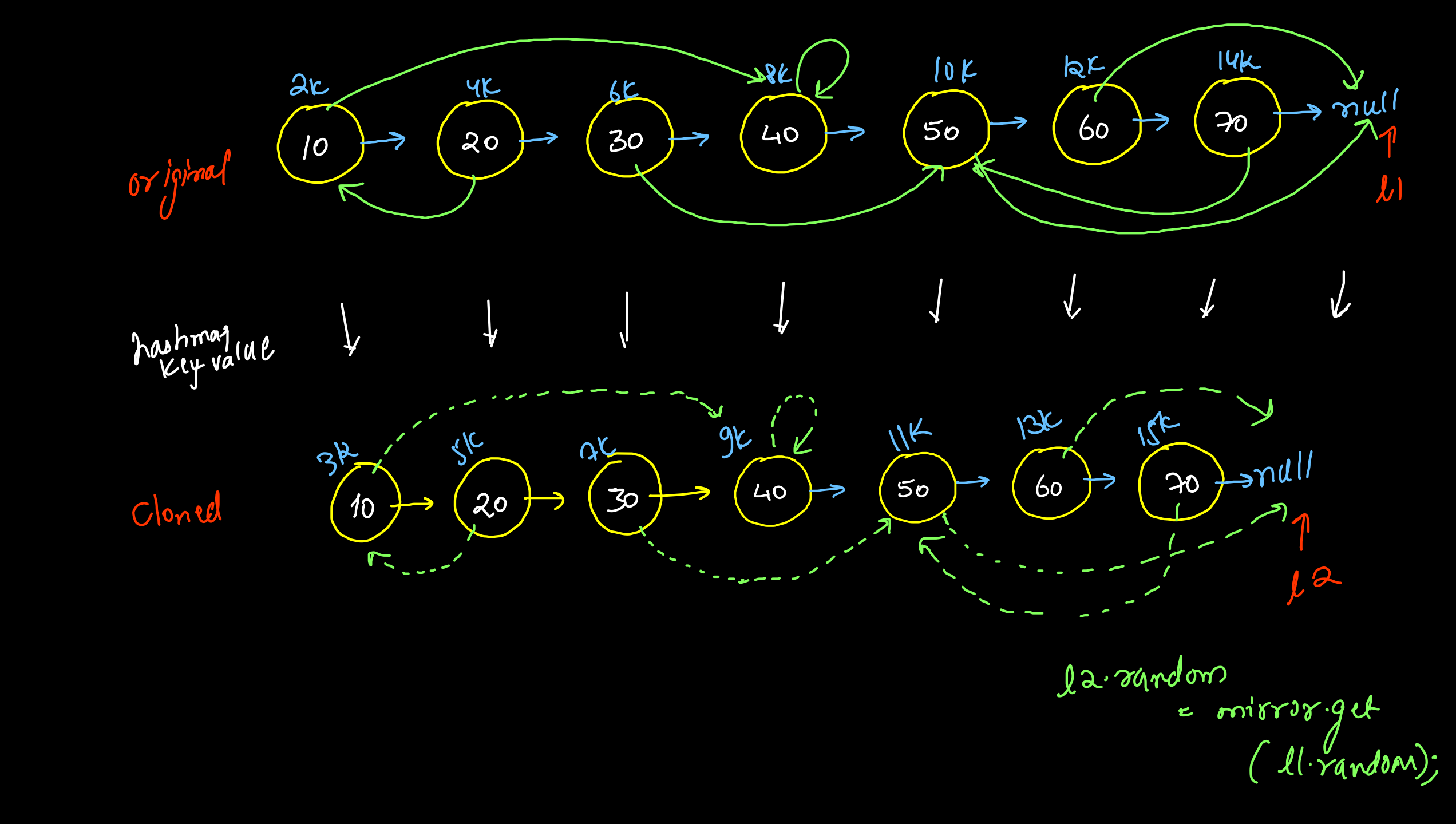






Step 1) Clone the linkedlist ignoring random pointers!

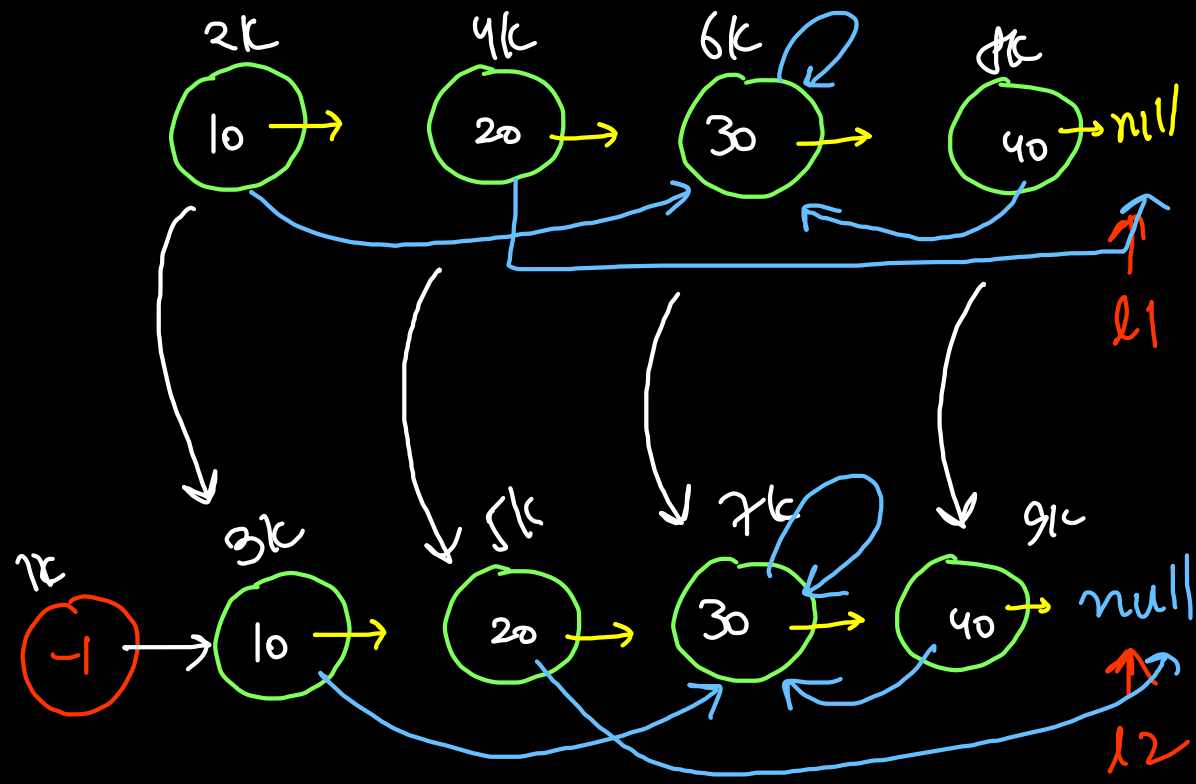
```
public Node copyRandomList(Node head) {
    Node dummy = new Node(-1);
    Node l1 = head, l2 = dummy;

    while(l1 != null){
        l2.next = new Node(l1.val);
        l1 = l1.next;
        l2 = l2.next;
    }
```


original
2K
4K
6K
8K
10K
12K
14K
10
20
30
40
50
60
70
null
l1
hashmap
key value
Cloned
3K
5K
7K
9K
11K
13K
15K
10
20
30
40
50
60
70
null
l2
l2.random
= mirror.get
(l1.random);

```java
public Node copyRandomList(Node head) {
    Node dummy = new Node(-1);
    Node l1 = head, l2 = dummy;
    HashMap<Node, Node> mirror = new HashMap<>();

    while(l1 != null){
        l2.next = new Node(l1.val);
        mirror.put(l1, l2.next);
        l1 = l1.next;
        l2 = l2.next;
    }

    l1 = head; l2 = dummy.next;
    while(l1 != null){
        l2.random = mirror.get(l1.random);
        l1 = l1.next;
        l2 = l2.next;
    }

    return dummy.next;
}
```

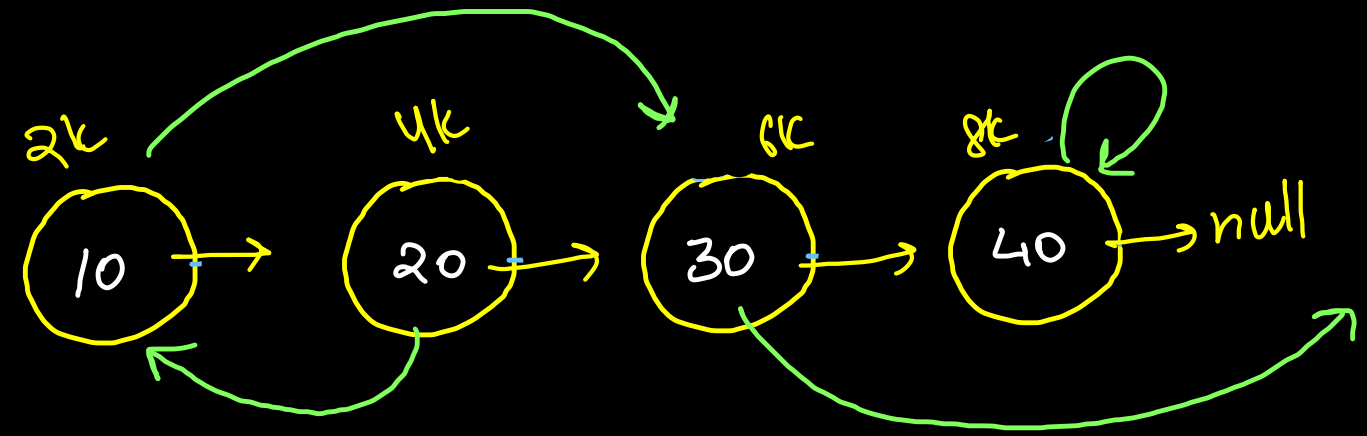
O(n)

O(n)

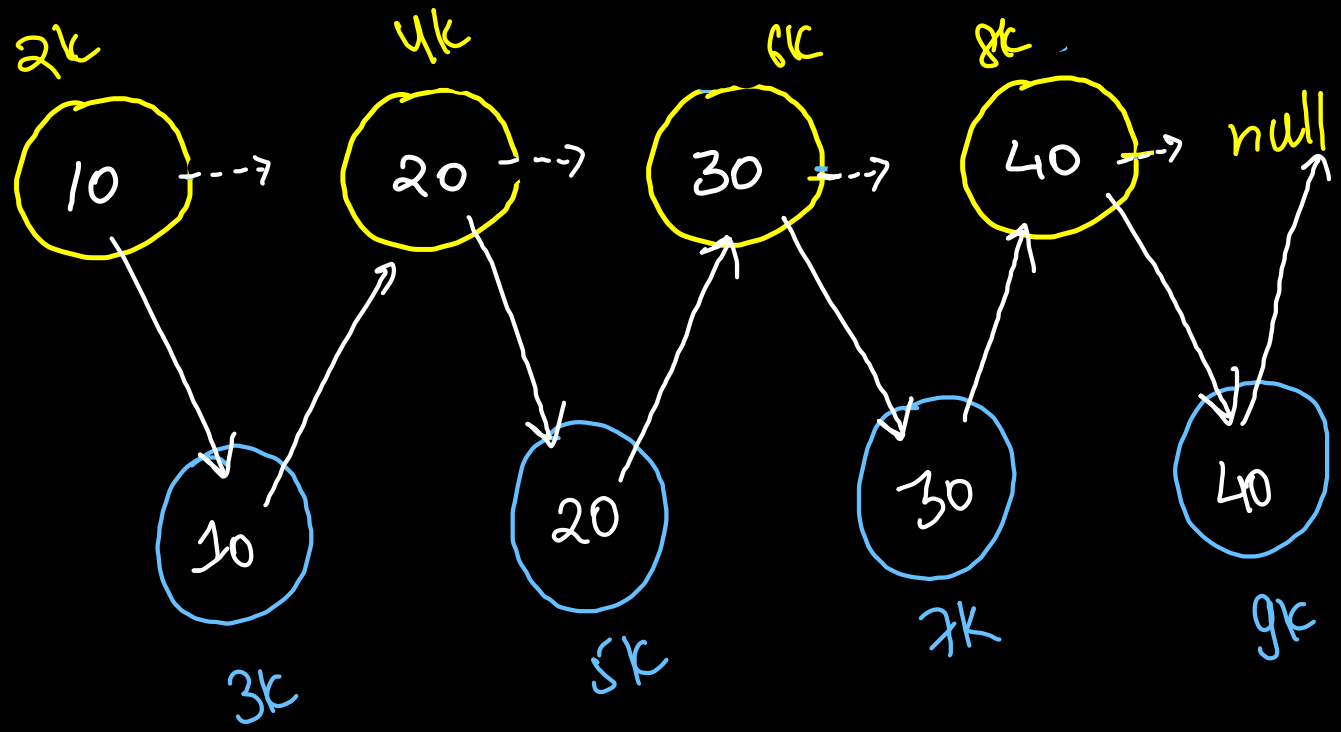


Time ⇒ O(n)

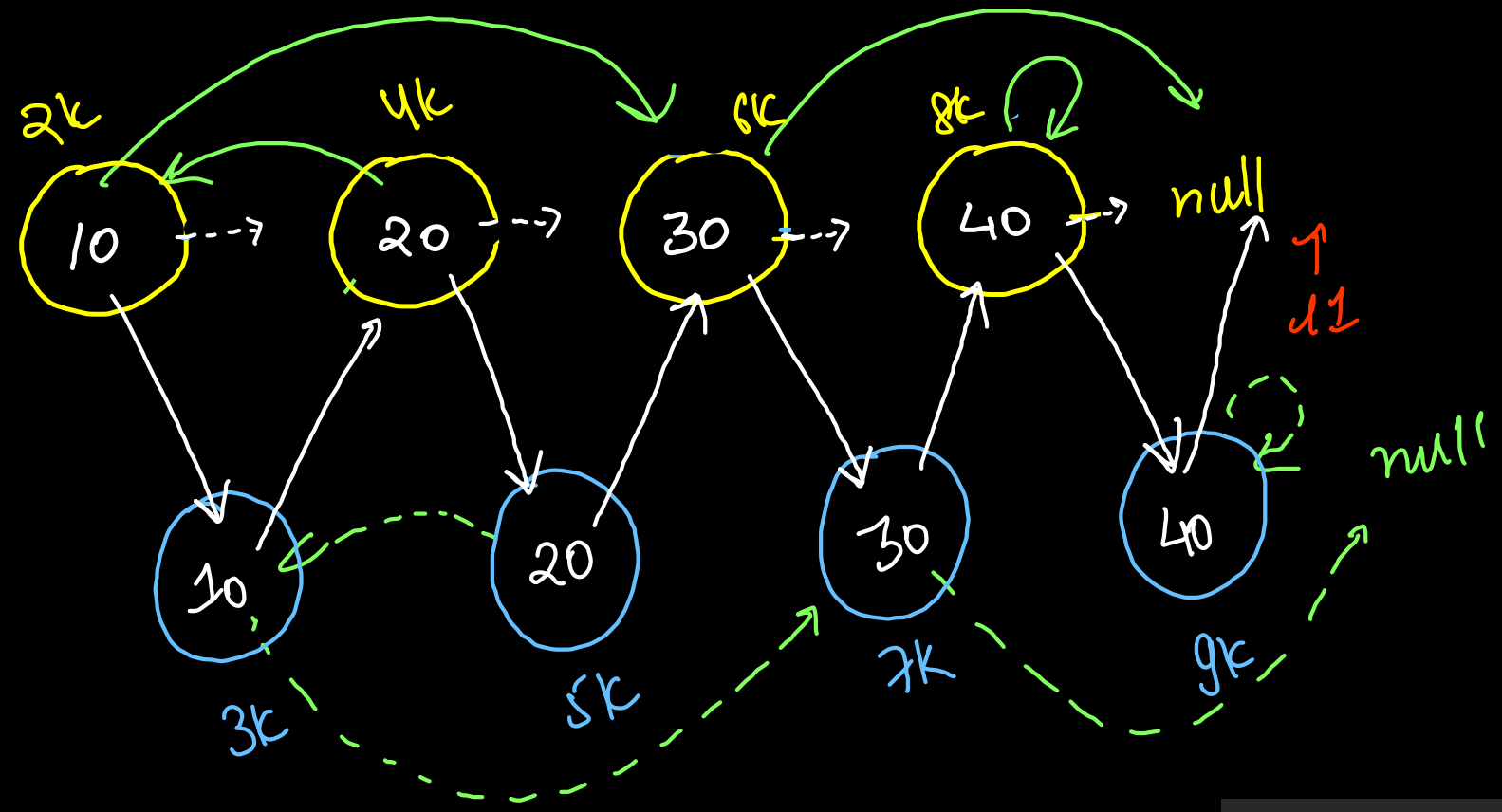
Space ⇒ O(n)
hashmap




```
Node l1 = head;
while(l1 != null){
    Node l2 = new Node(l1.val);
    l2.next = l1.next;
    l1.next = l2;
    l1 = l1.next.next;
}
```



↑
l1

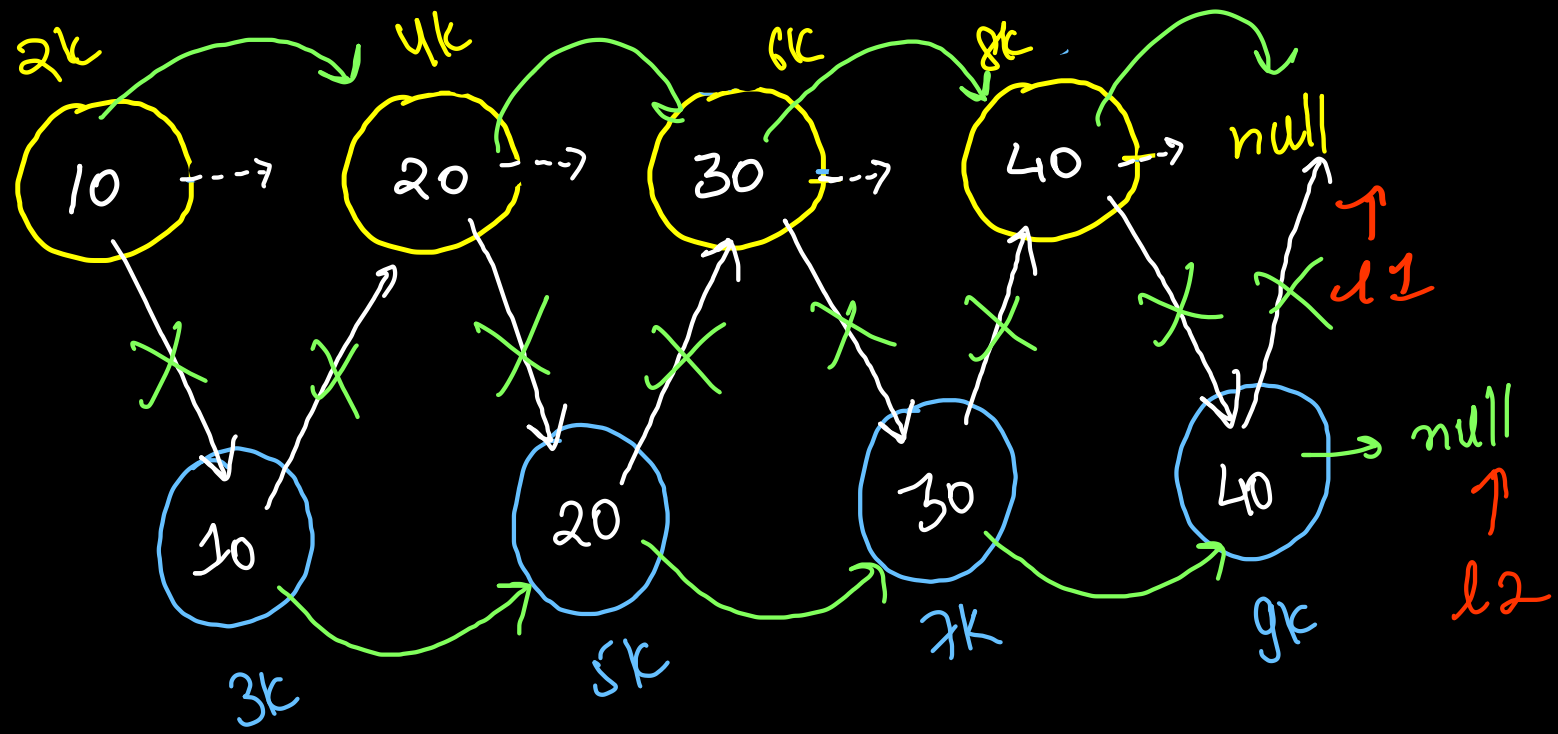


l1.next.random = l1.random.next

l1.next: duplicate of l1

l1.random.next: duplicate of random of l1

```
l1 = head;
// Clone Random Pointers
while(l1 != null){
    if(l1.random != null)
        l1.next.random = l1.random.next;
    l1 = l1.next.next;
}
```


2k
4k
6k
8k
10
20
30
40
null
l1
10
20
30
40
null
l2
3k
5k
7k
9k

```java
public Node copyRandomList(Node head) {
    if(head == null) return null;

    Node l1 = head;
    // Create Cloned Nodes
    while(l1 != null){
        Node l2 = new Node(l1.val);
        l2.next = l1.next;
        l1.next = l2;
        l1 = l1.next.next;
    }

    l1 = head;
    // Clone Random Pointers
    while(l1 != null){
        if(l1.random != null)
            l1.next.random = l1.random.next;
        l1 = l1.next.next;
    }

    // Detach Two Lists
    Node ans = head.next;
    l1 = head;
    Node l2 = head.next;
    while(l1 != null){
        if(l1.next != null)
            l1.next = l1.next.next;
        if(l2.next != null)
            l2.next = l2.next.next;
        l1 = l1.next;
        l2 = l2.next;
    }

    return ans;
```

O(n)

O(n)

O(n)

Time ⇒ O(n)

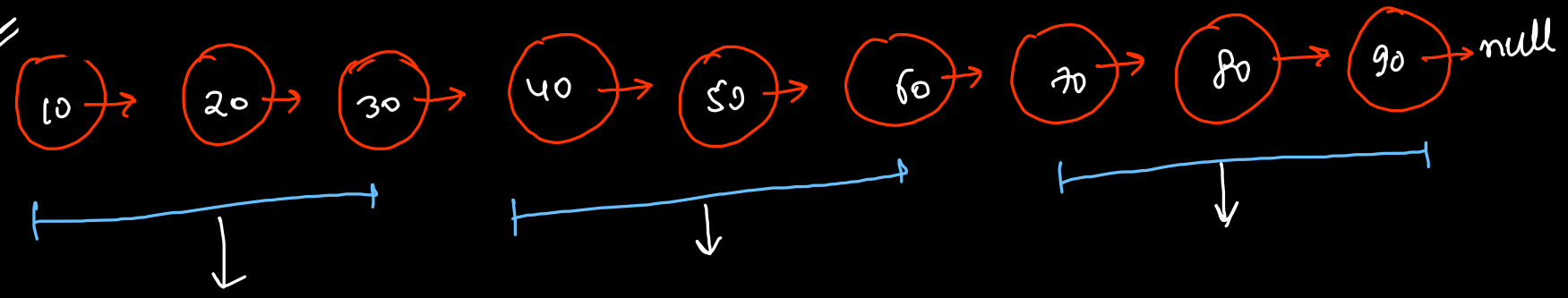
Space ⇒ O(1)
(Extra)

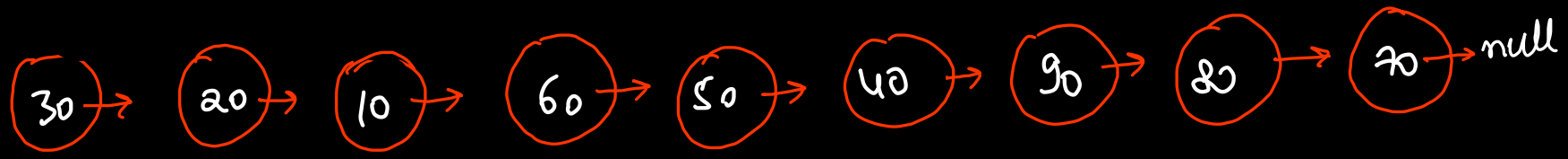
Reverse Linked list in Groups of K

$k \leq n$

$k = 3$

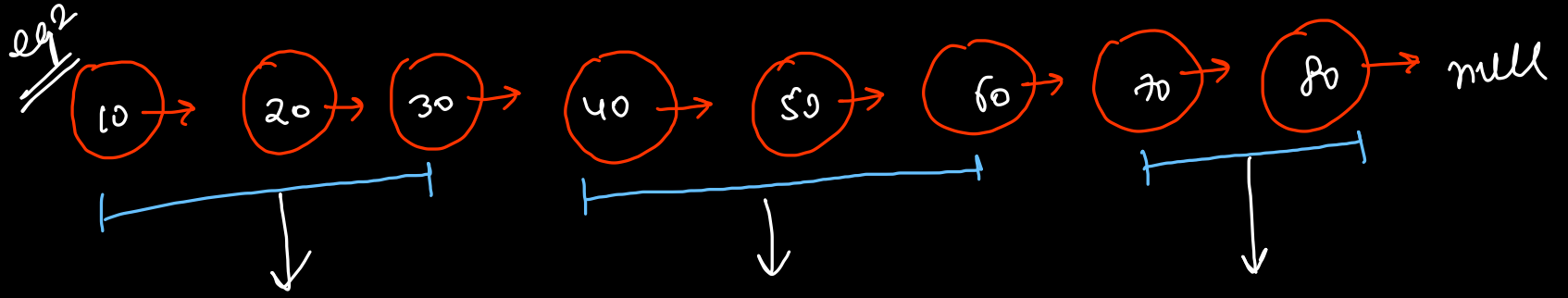
eg 1

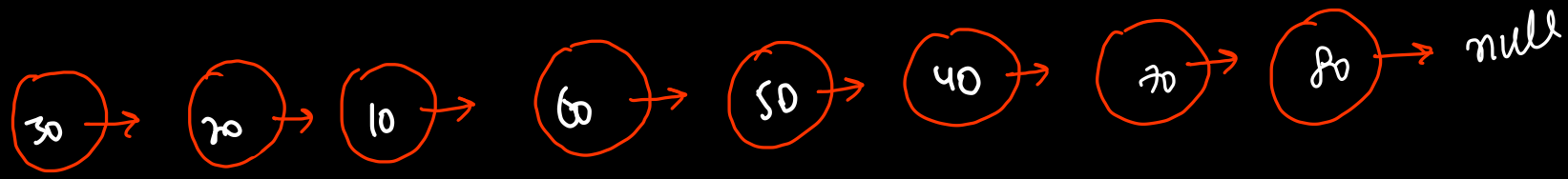


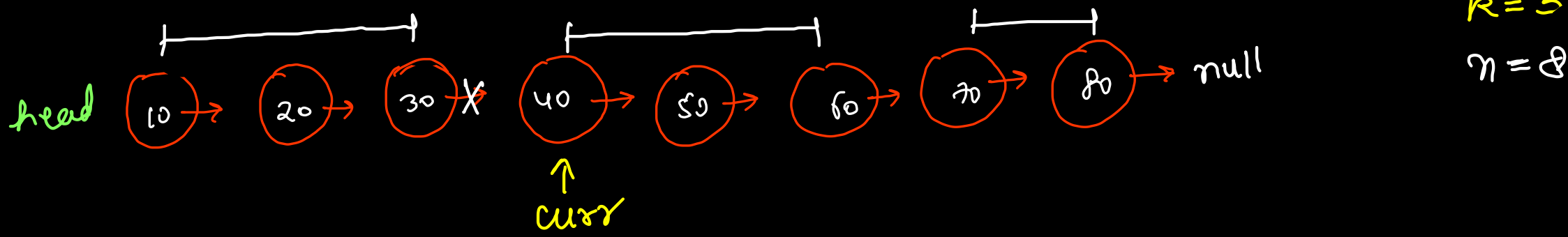


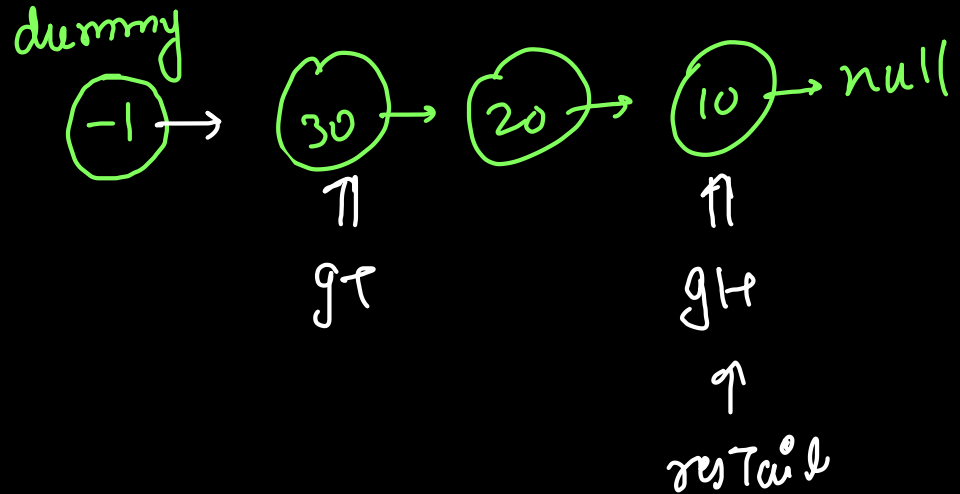
eg2

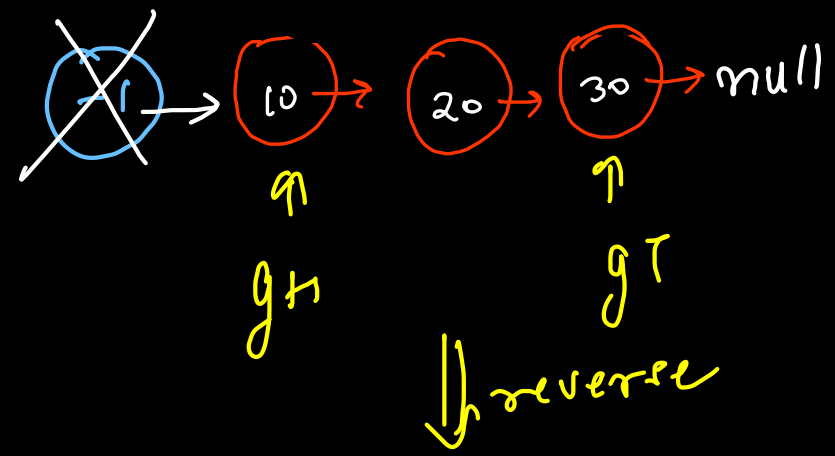
n=8 k=3



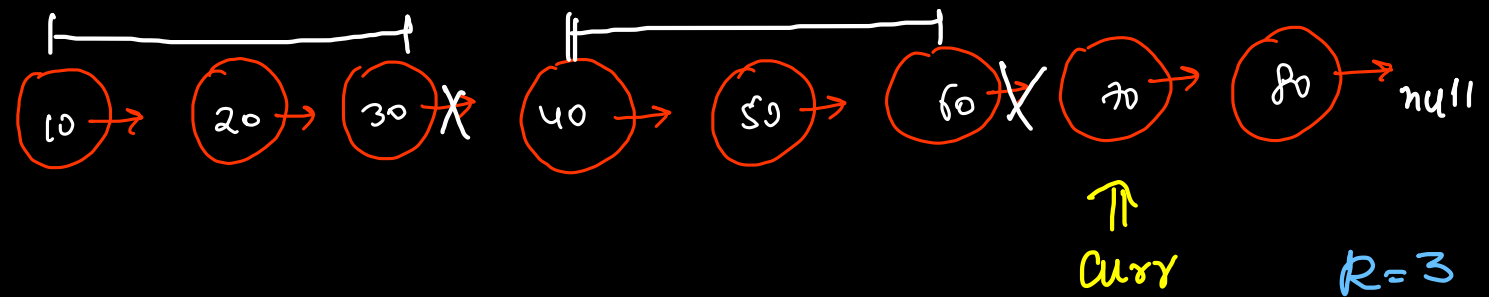


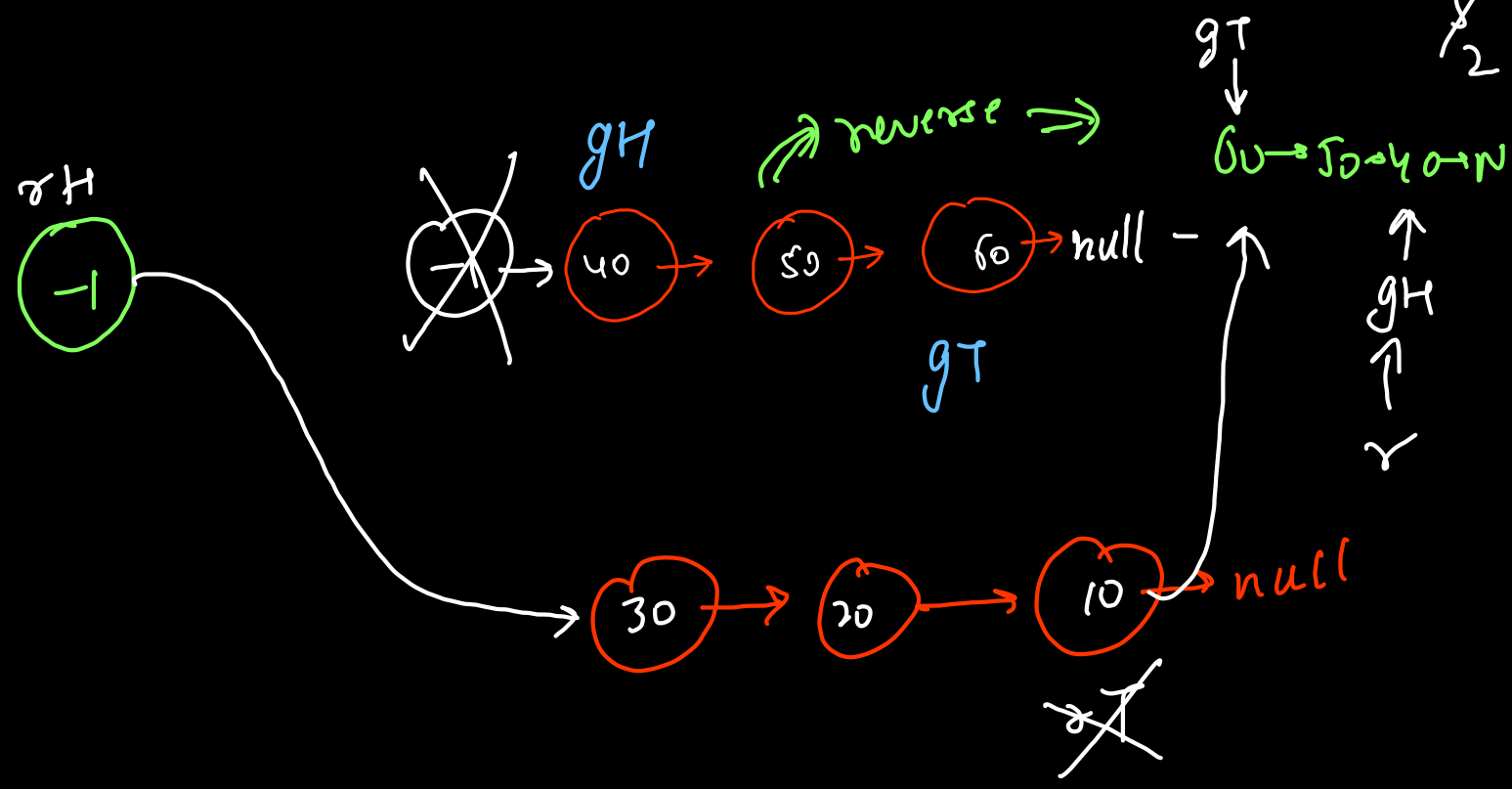

k = 3
n = 8
head
10
20
30
40
50
60
70
80
null
curr


dummy
-1
30
20
10
null
gT
gH
resTail


-1
10
20
30
null
gH
gT
reverse

```java
int n = size(curr);
ListNode resHead = new ListNode(-1);
ListNode resTail = resHead;

while(curr != null){
    if(n < k){
        // Group Length < K: Do Not Reverse
        resTail.next = curr;
        break;
    }

    n -= k;
    ListNode groupHead = new ListNode(-1);
    ListNode groupTail = groupHead;

    for(int idx = 0; idx < k; idx++){
        groupTail.next = curr;
        groupTail = curr;
        curr = curr.next;
    }

    groupTail.next = null;
    groupHead = groupHead.next;
    reverse(groupHead);
    resTail.next = groupTail;
    resTail = groupHead;
}

return resHead.next;
```





```java
public int size(ListNode curr){
    int count = 0;
    while(curr != null) {
        count++;
        curr = curr.next;
    }
    return count;
}

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;
}
```

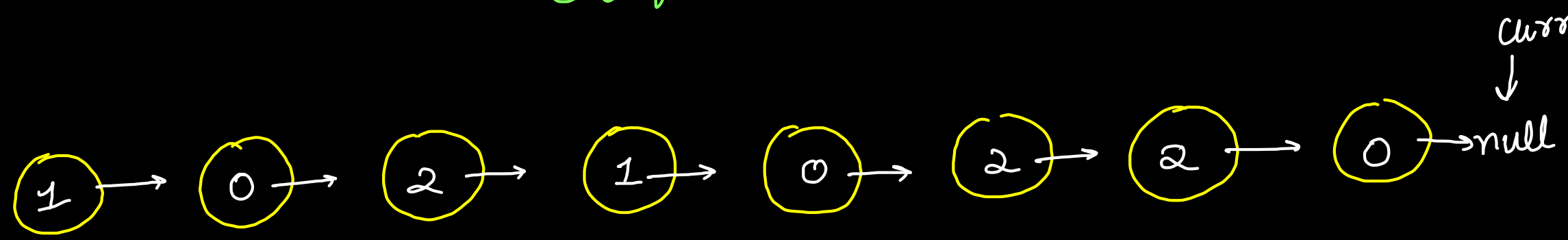
```java
public ListNode reverseKGroup(ListNode curr, int k) {
    int n = size(curr);
    ListNode resHead = new ListNode(-1);
    ListNode resTail = resHead;

    while(curr != null){
        if(n < k){
            // Group Length < K: Do Not Reverse
            resTail.next = curr;
            break;
        }

        n -= k;
        ListNode groupHead = new ListNode(-1);
        ListNode groupTail = groupHead;

        for(int idx = 0; idx < k; idx++){
            groupTail.next = curr;
            groupTail = curr;
            curr = curr.next;
        }

        groupTail.next = null;
        groupHead = groupHead.next;
        reverse(groupHead);
        resTail.next = groupTail;
        resTail = groupHead;
    }

    return resHead.next;
}
```

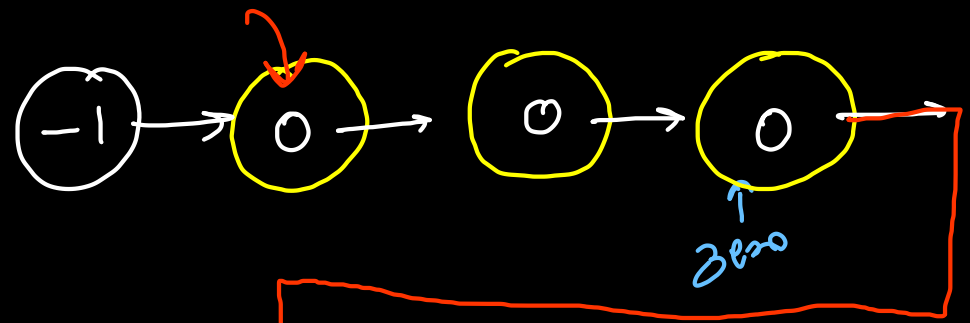
Total time
↳ O(n)

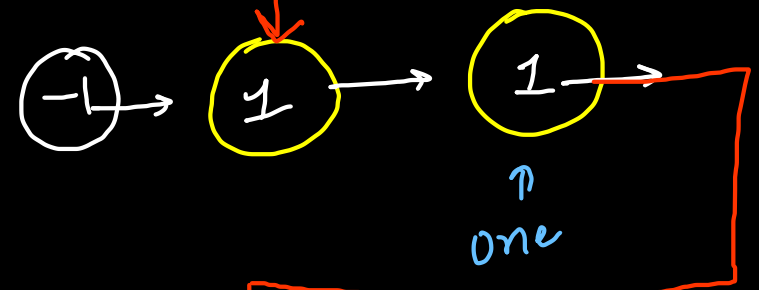
Space
↳ O(1)

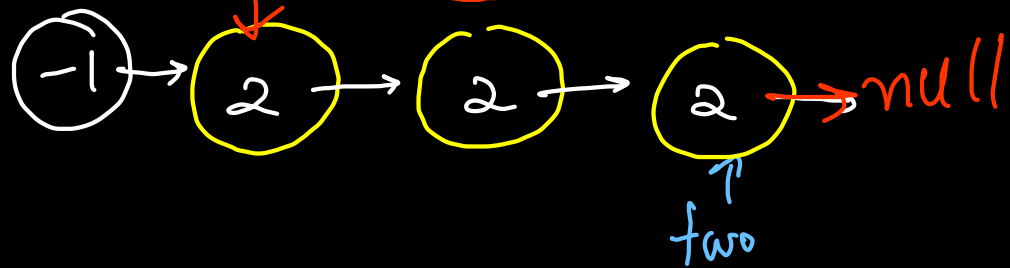
Note: groupHead & groupTail are not swapped!

Sort 0 1 2 {Inplace & stable}



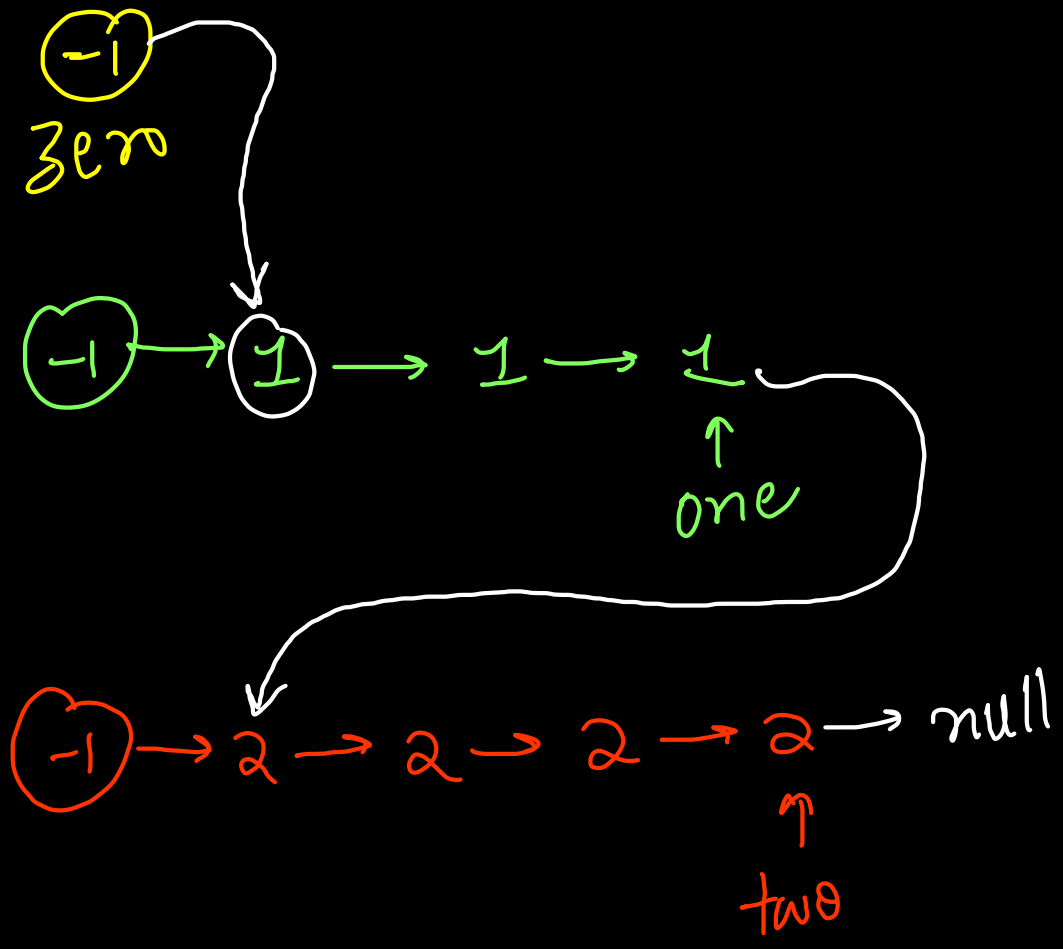







```
zeroTail.next = oneHead.next
oneTail.next = twoHead.next
twoTail.next = null
return zeroHead.next;
```

1 → 1 → 2 → 2 → 1 → 2 → 2 → null



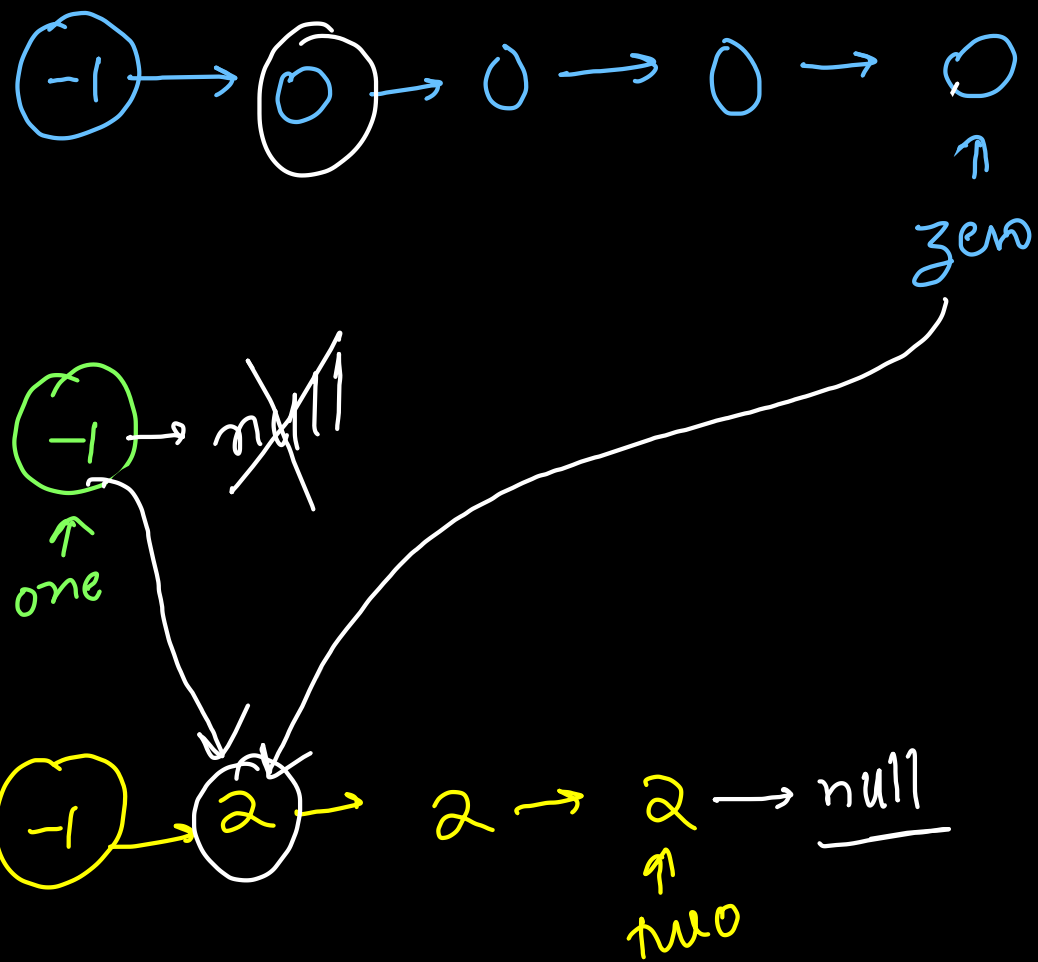
zeroTail.next = oneHead.next

oneTail.next = twoHead.next

twoTail.next = null

return zeroHead.next;

0 → 2 → 2 → 0 → 0 → 2 → 0 → null



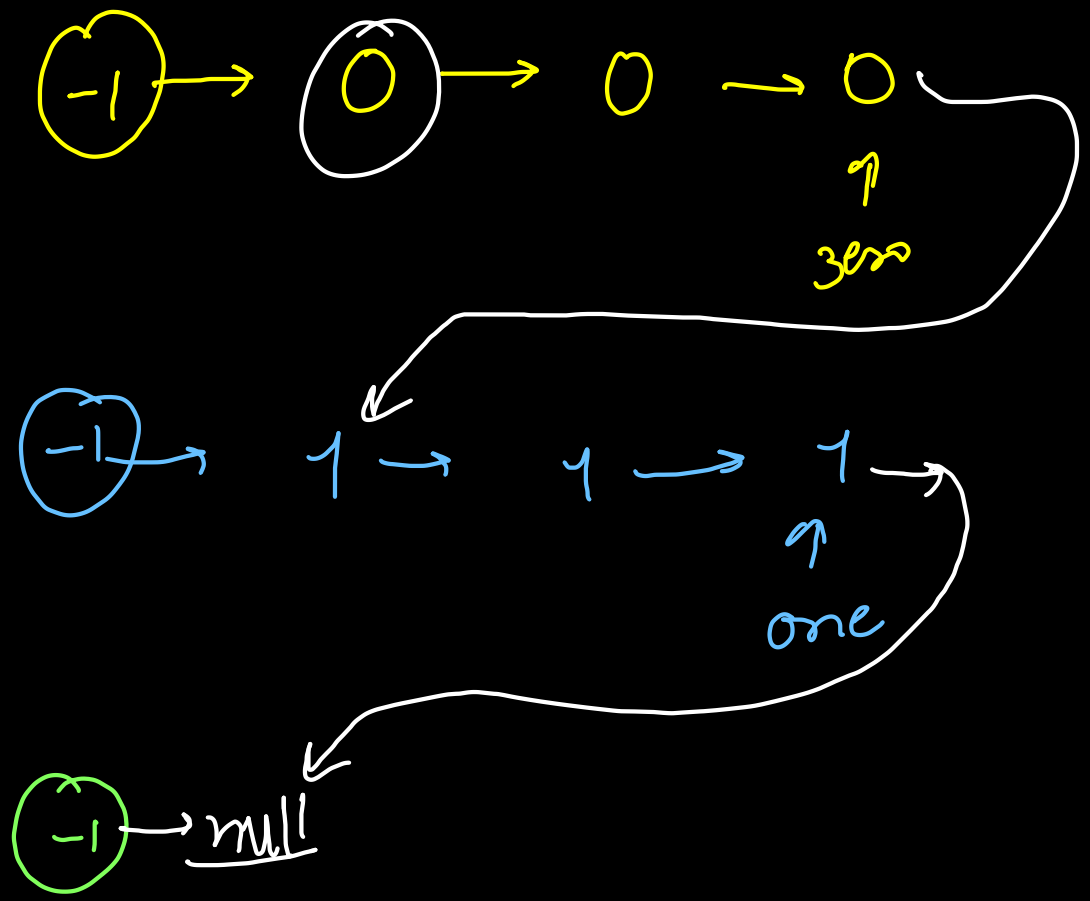
oneTail.next = twoHead.next

zeroTail.next = oneHead.next

twoTail.next = null

return zeroHead.next;

0 → 1 → 0 → 1 → 1 → 0 → null



- ✓ oneTail.next = twoHead.next
- ✓ zeroTail.next = oneHead.next
- ✓ twoTail.next = null
- ✓ return zeroHead.next;

```java
static Node segregate(Node head)
{
    Node zeroDummy = new Node(-1);
    Node zero = zeroDummy;
    Node oneDummy = new Node(-1);
    Node one = oneDummy;
    Node twoDummy = new Node(-1);
    Node two = twoDummy;

    while(head != null){
        if(head.data == 0){
            zero.next = head;
            zero = zero.next;
        } else if(head.data == 1){
            one.next = head;
            one = one.next;
        } else if(head.data == 2){
            two.next = head;
            two = two.next;
        }
        head = head.next;
    }

    two.next = null;
    one.next = twoDummy.next;
    zero.next = oneDummy.next;
    return zeroDummy.next;
}
```

Total time $\Rightarrow O(n)$

space $\Rightarrow O(1)$

inplace/stable