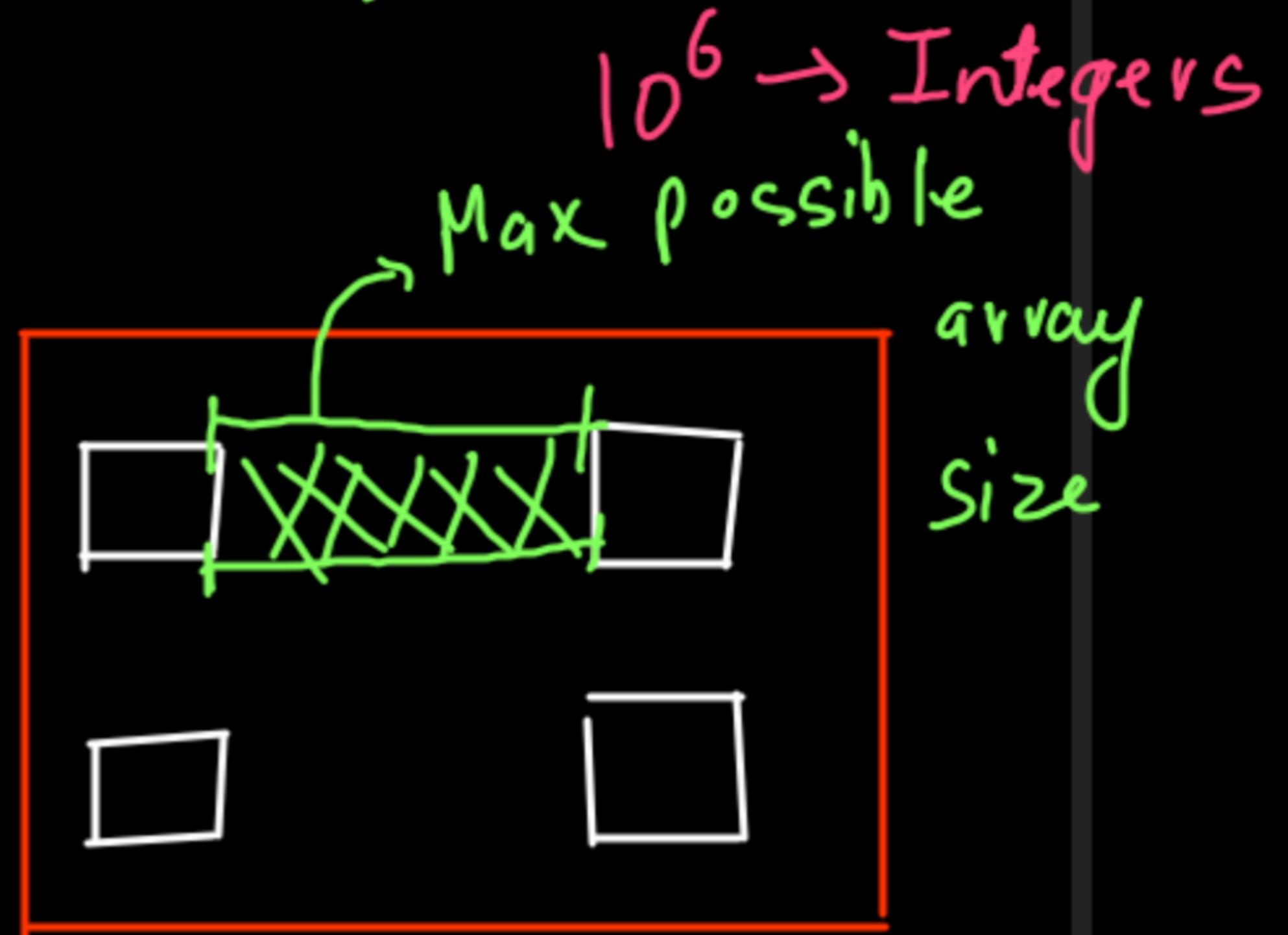
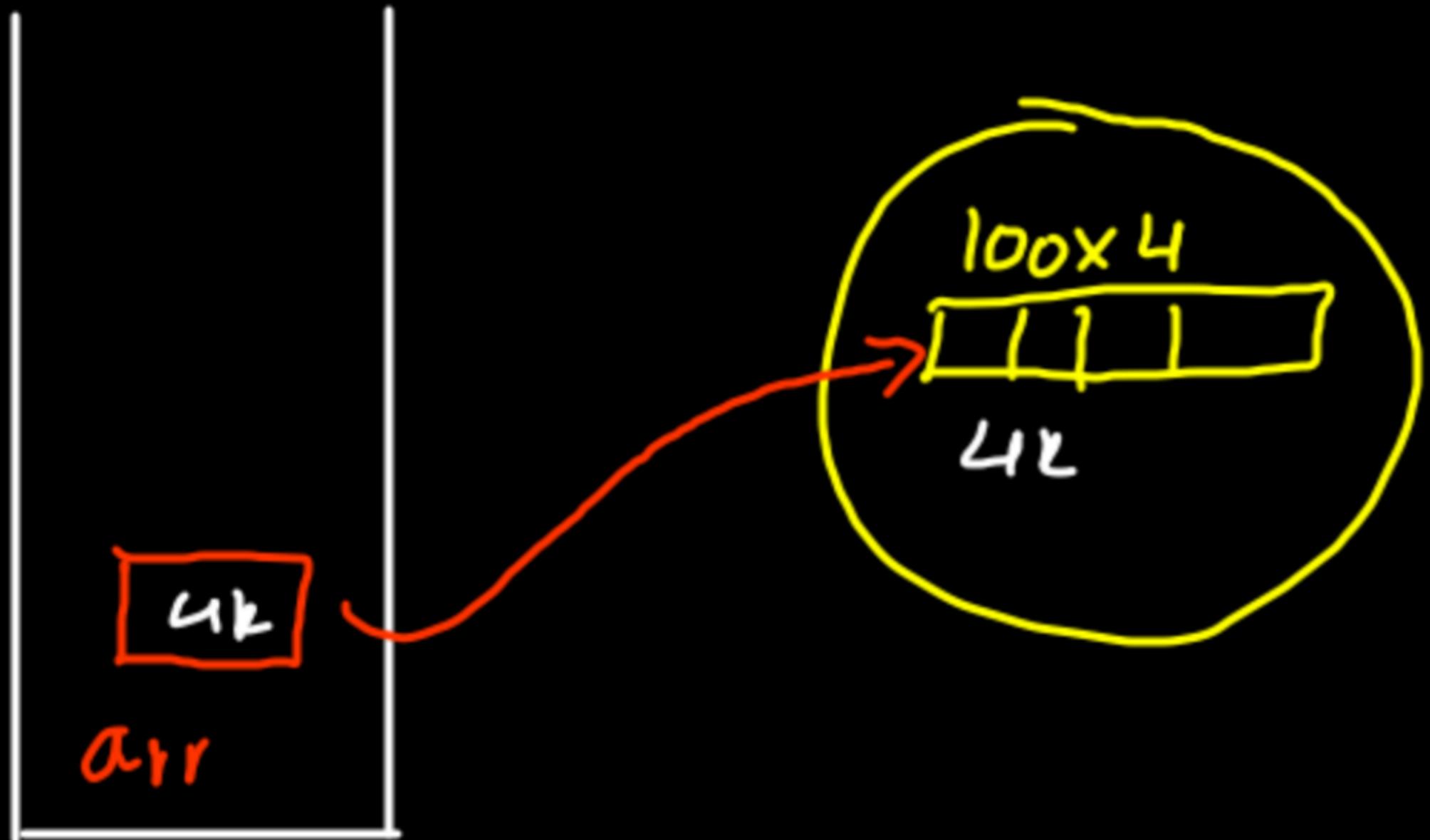


Linked List (Level 1+2)

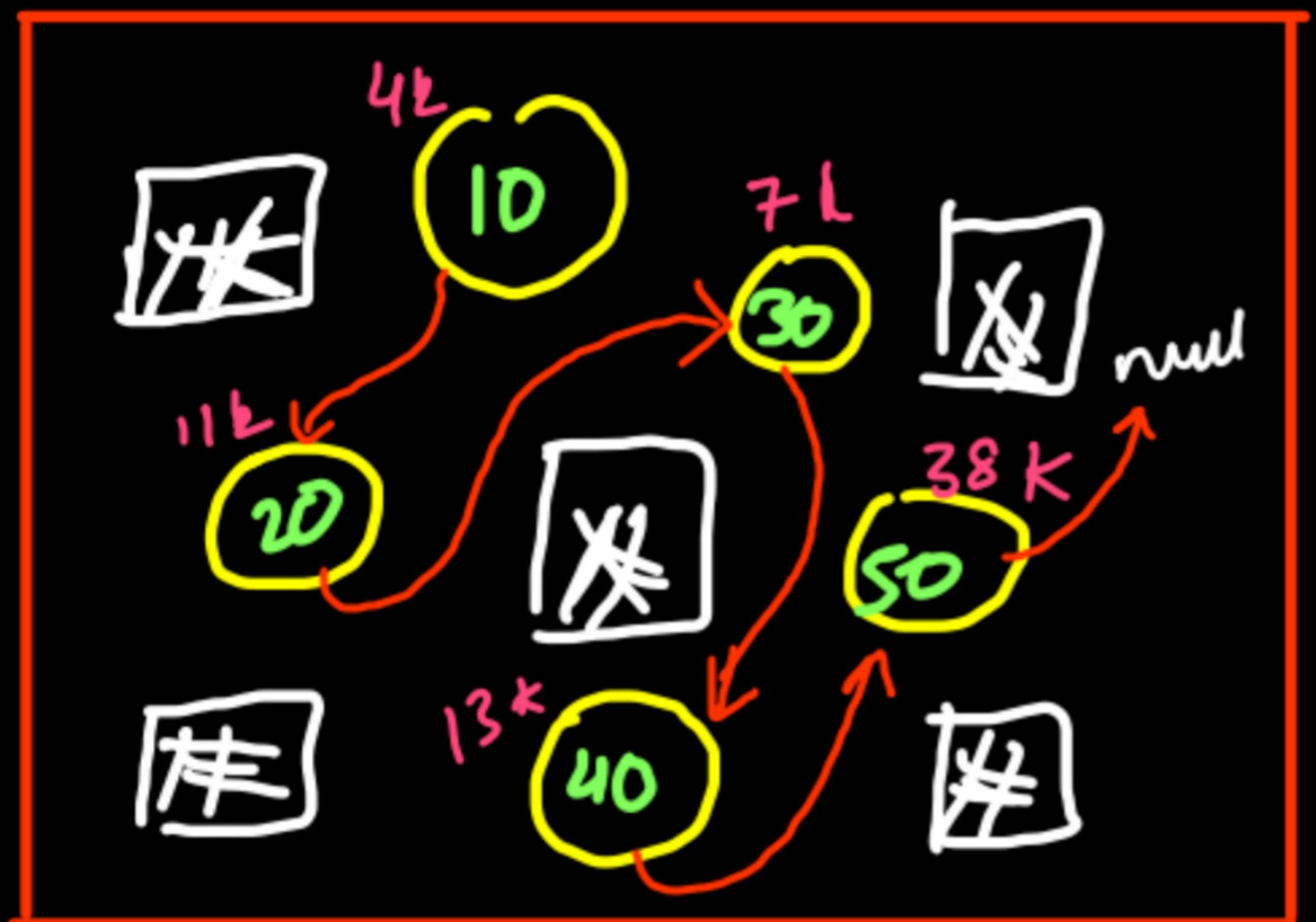
Arrays

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



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

Linked List: Non contiguous memory allocation.



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

We need links to connect data-

10, 20, 30, 40, 50
4k 11k 7k 13k 38k

Node { object } {

int data;

Node next; → Address of
Next

3

Node

Inner class ko static banana hai.

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

class
↓
data members
functions

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.

```
public static class LinkedList {  
  
    Node head;  
  
    public void display() {}  
  
    public void addFirst(int data) {}  
  
    public void addLast(int data) {}  
  
    public int get(int index) {}  
  
}
```

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

```
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();

3

list = 2k
Stack

head = null
tail = null
size = 0

2k

Heap

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

list = 2k

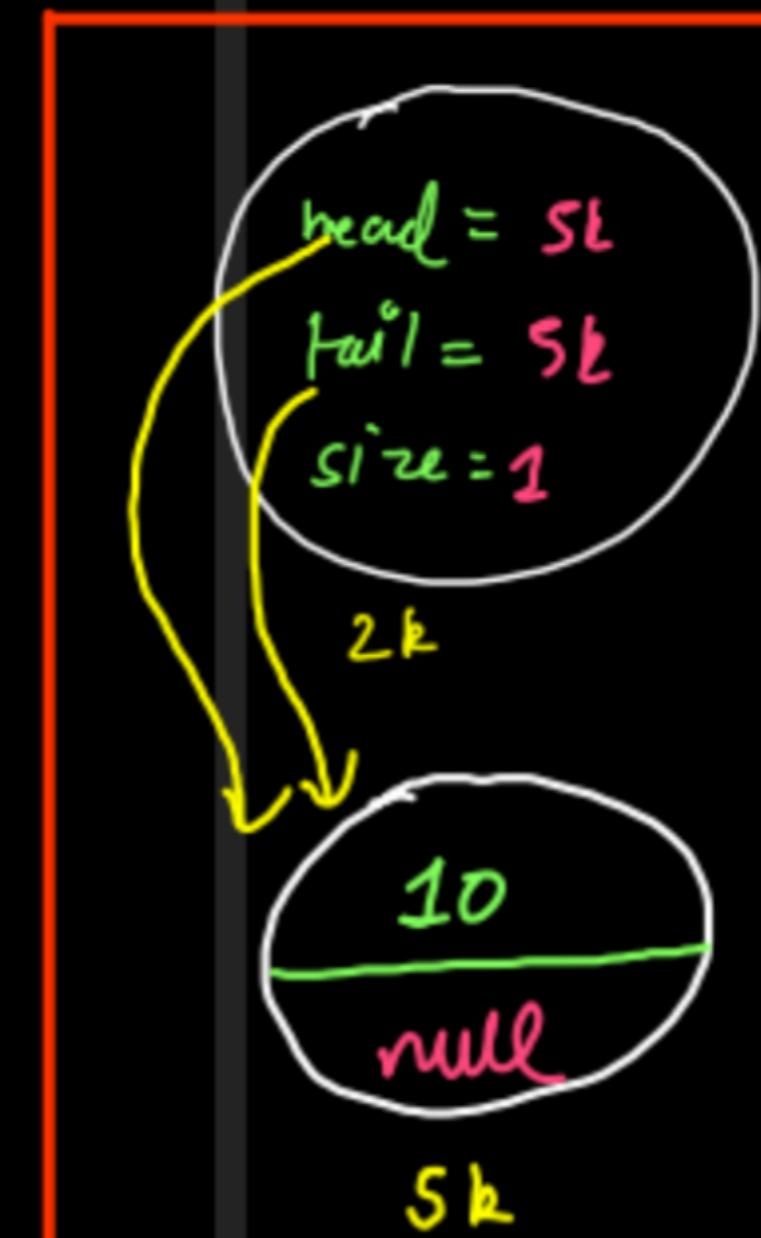
Stack

psv main() {

LinkedList list = new LinkedList();

list.addLast(10);

3



Heap

```

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

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) {
    }
}

```

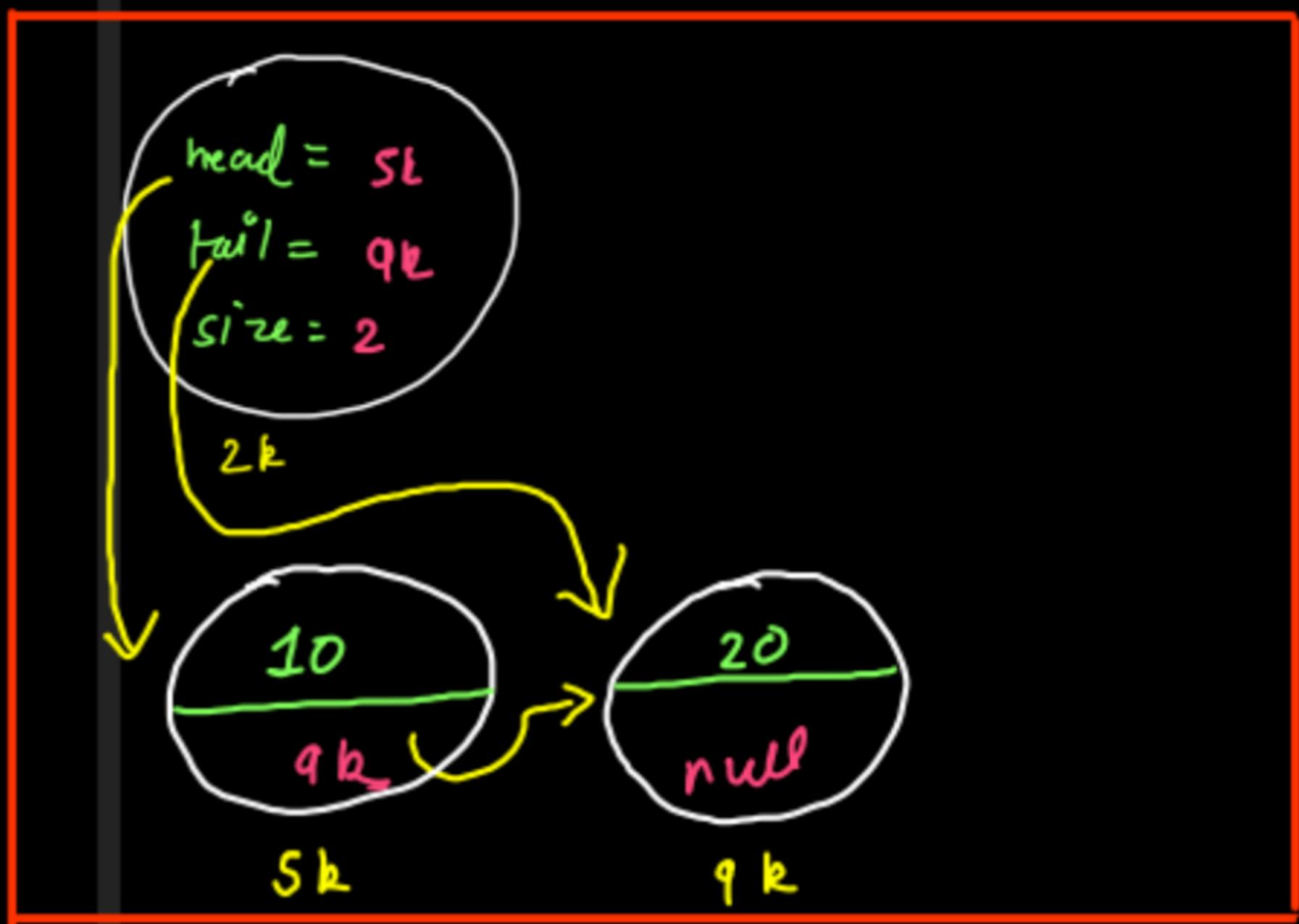
list = 2k

Stack

```

psv main() {
    LinkedList list = new LinkedList();
    list.addLast(10);
    list.addLast(20);
}

```



Heap

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

```
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);

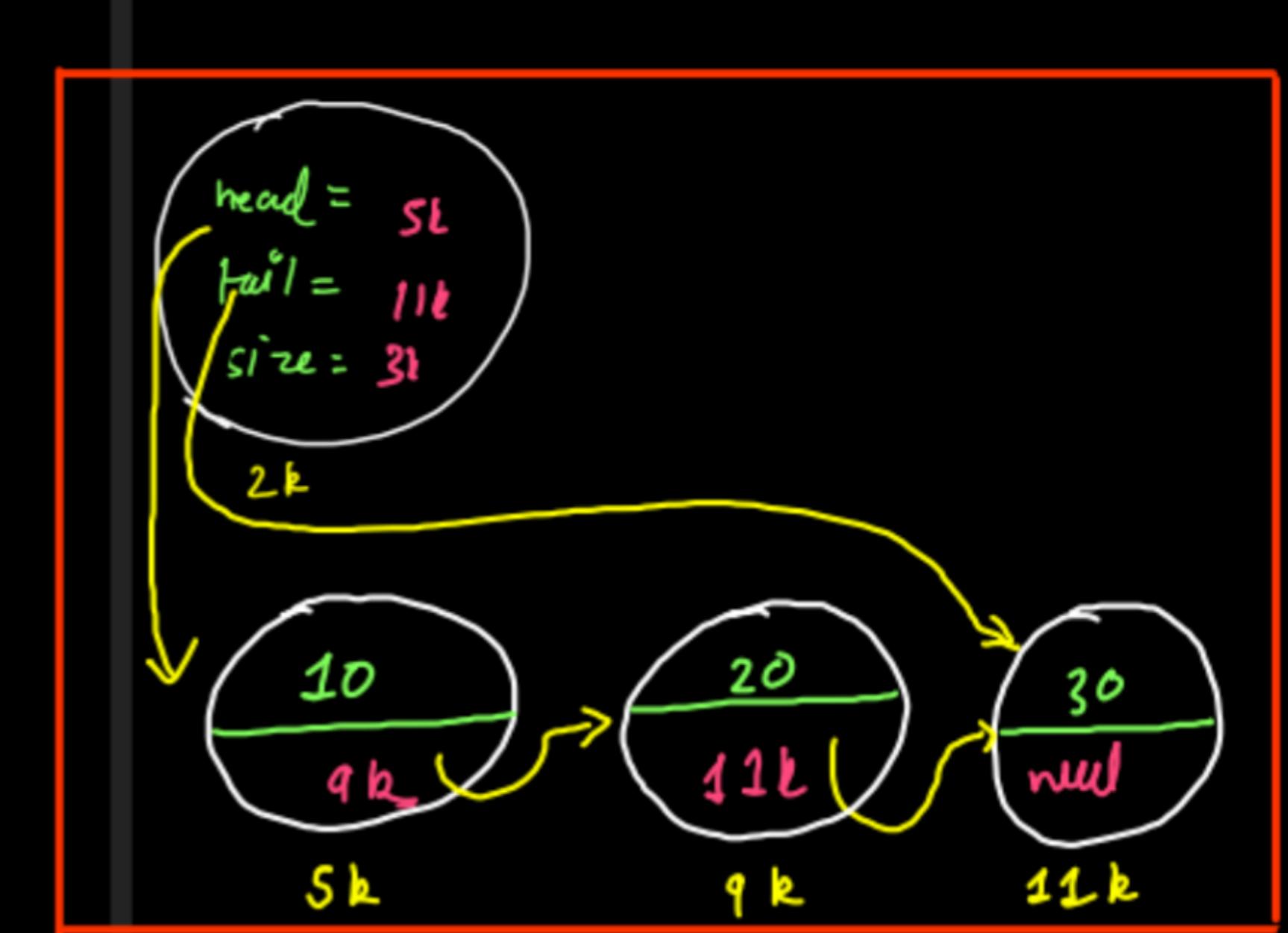
list.addLast(30);

}

list = 2k

Stack

++ Size of reference is
unknown in LL.



Heap

LL is taking lot of extra space

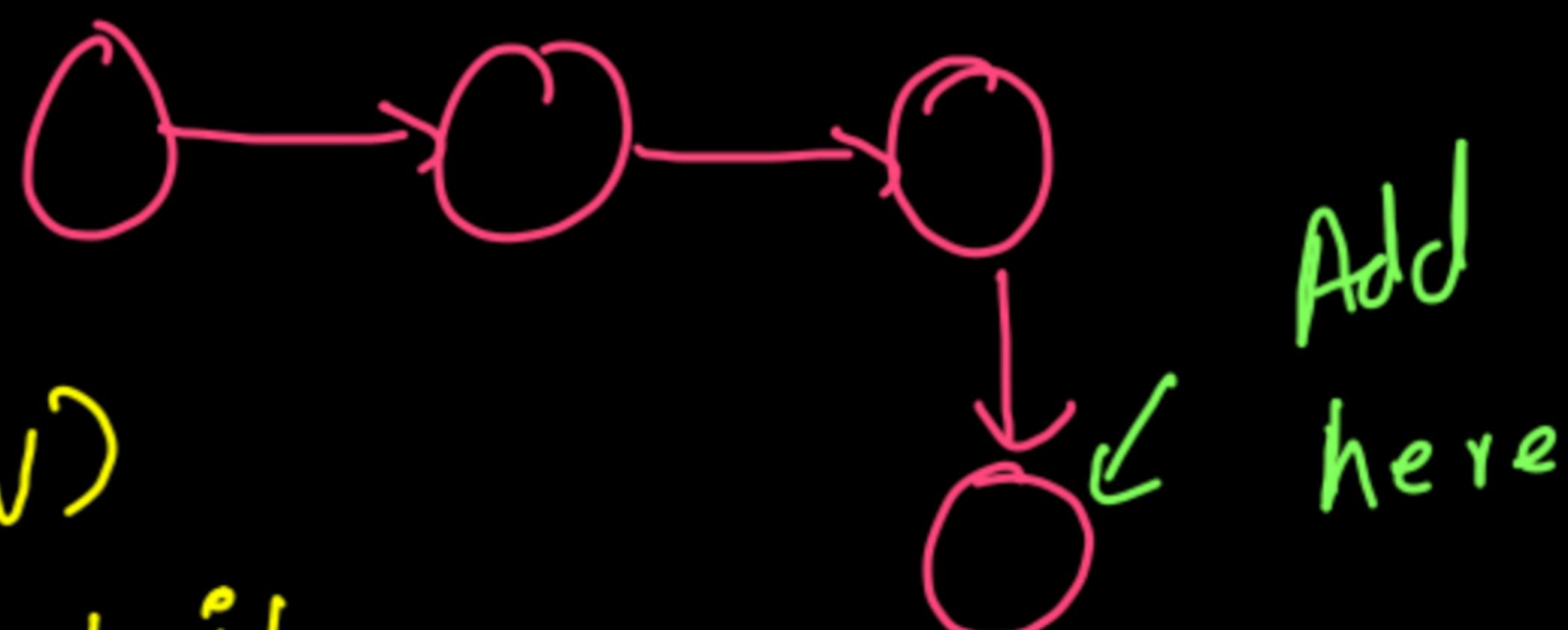
Random Access is not allowed
in LL

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++;  
}
```

$TC = O(1)$ \rightarrow using tail
addLast is a constant operation

#If tail is not given



$TC = O(N)$
without tail

Add first

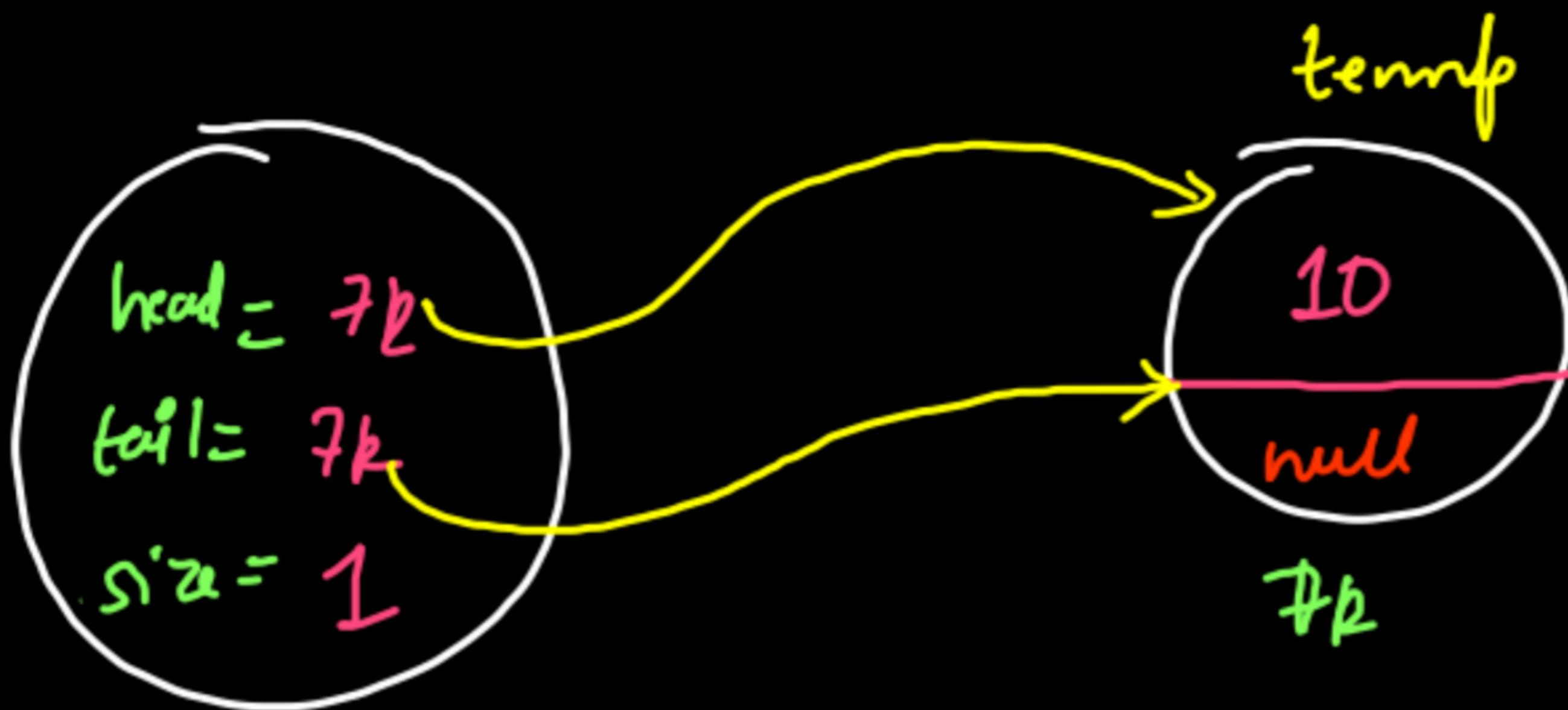
```
LinkedList list = new LinkedList();
```



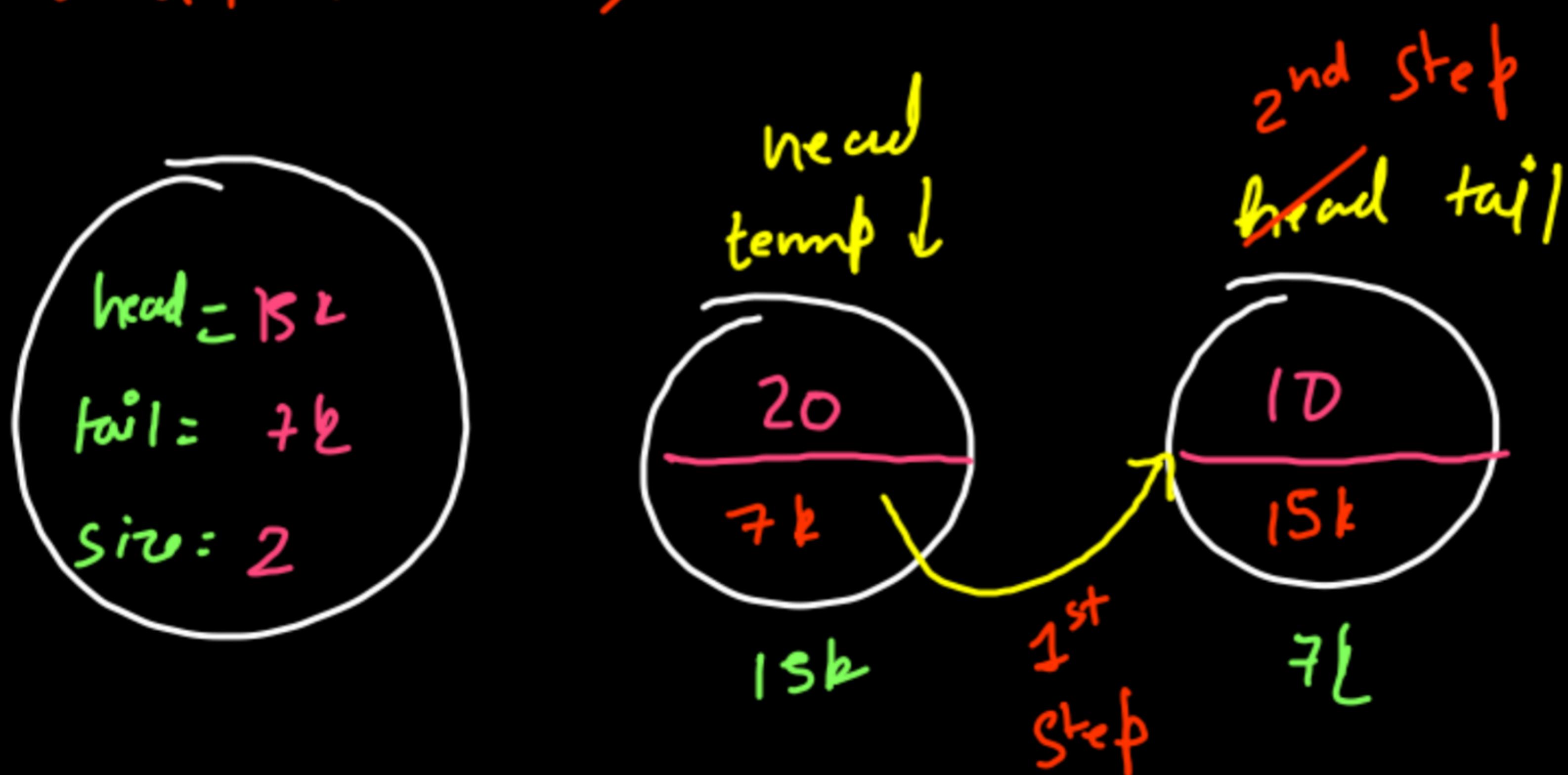
addFirst(10);



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



`addFirst(20);`



if `size == 0`
 1 `temp.next = head;`
 2 `head = temp;`

Add first Code

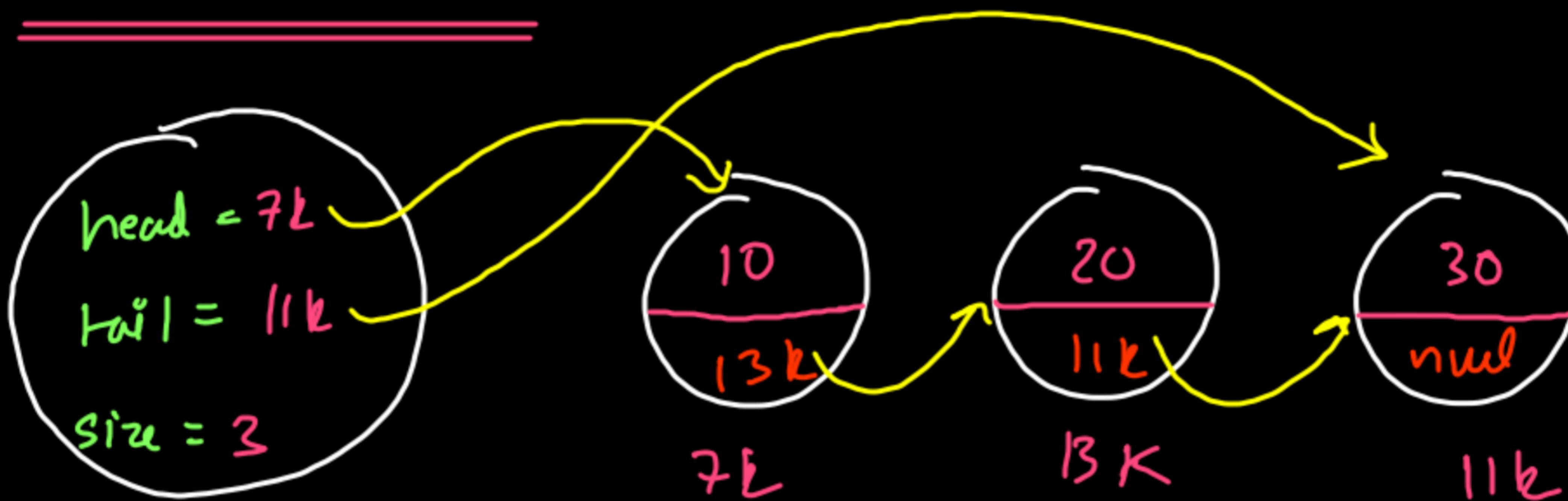
```
public void addFirst(int val) {  
    // write your code here  
  
    Node temp = new Node();  
    temp.data = val;  
  
    if(size == 0) {  
        head = temp;  
        tail = temp;  
    } else {  
        temp.next = head;  
        head = temp;  
    }  
  
    size++;  
}
```

$T C = O(1)$



always constant because
head will always be
there.

Remove first



removeFirst();

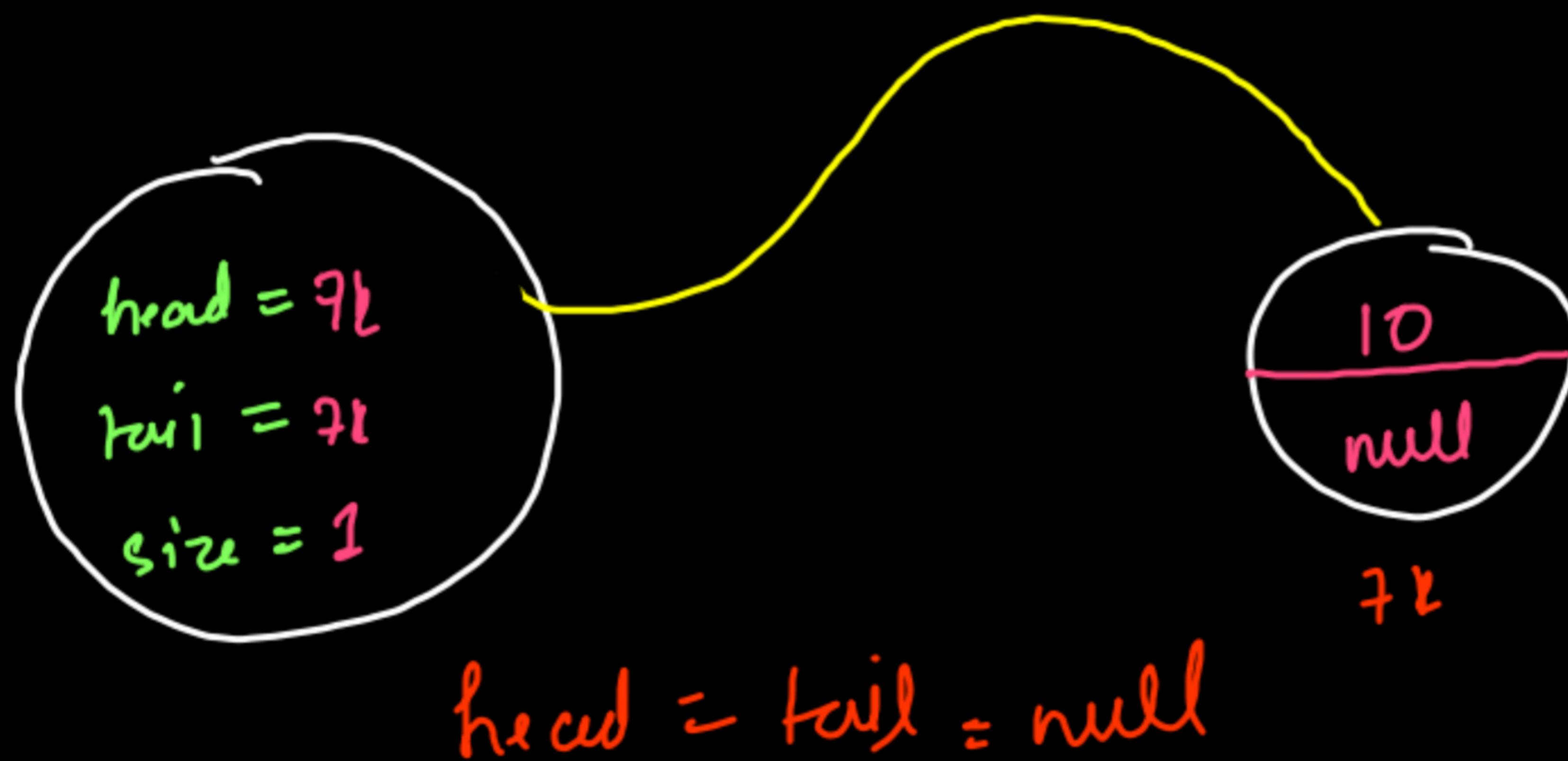


if (size > 1) head = head.next;

Now this is garbage
collectible.

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

when LL is empty, head = tail = null



Removefirst Code

```
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--;
}
```

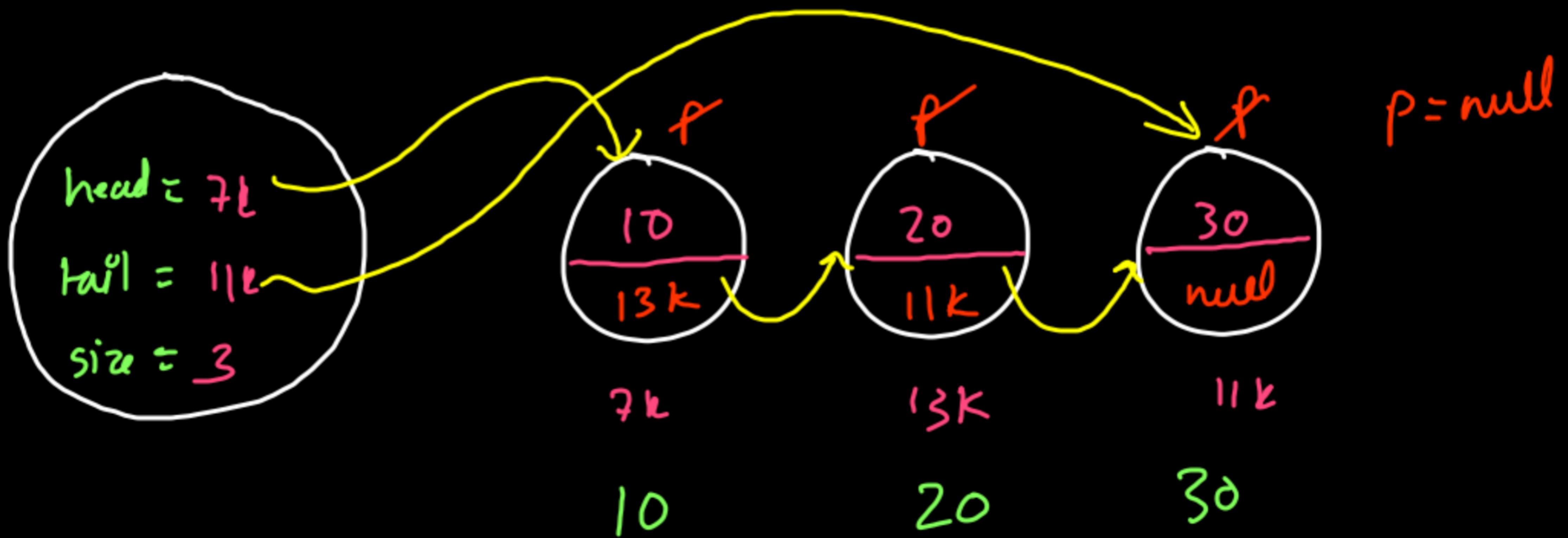
Display List

→ Using
size

```
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.

```
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; } In case of invalid index  
else return -1; } print "Invalid arguments"
```

If list is empty print "List is empty"

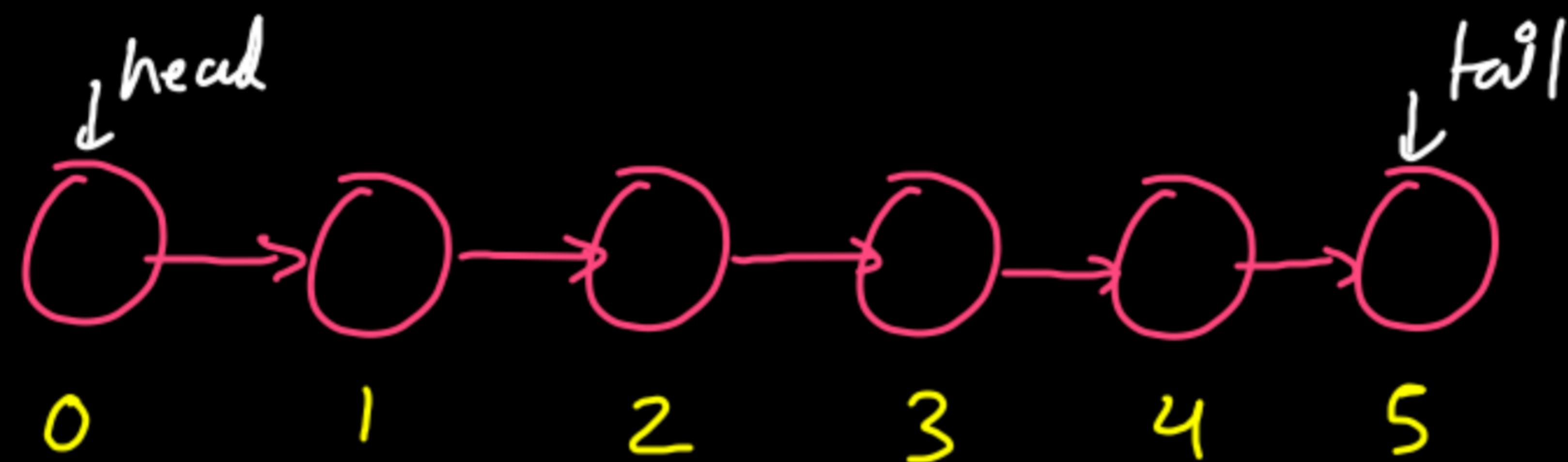
GetLast()

```
return tail.data; } In case of invalid index  
else return -1; } 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

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

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

```
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();

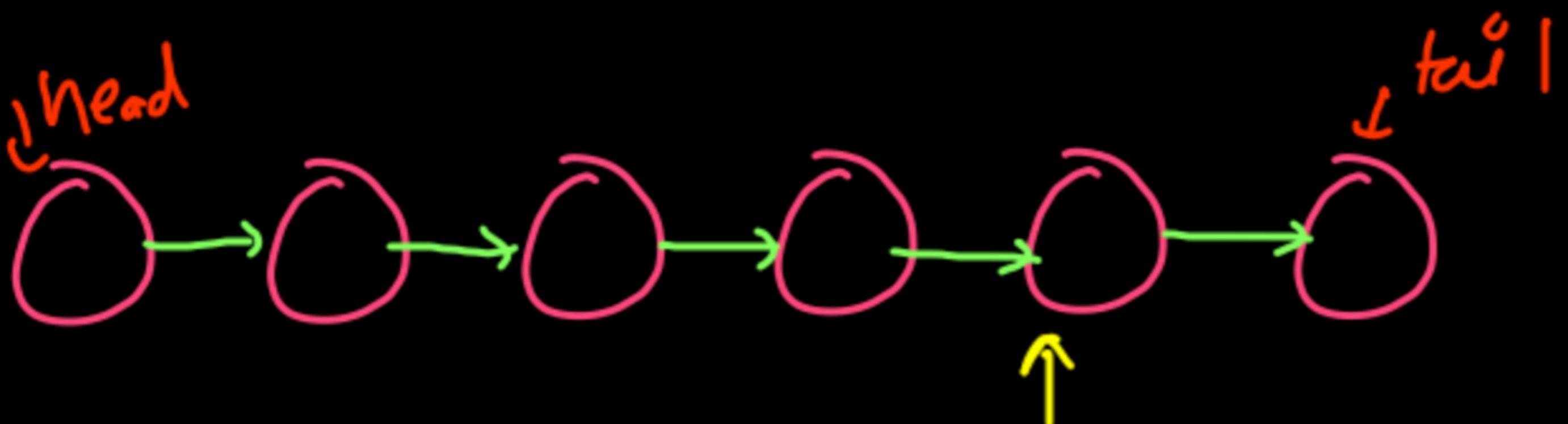
    if(idx == size - 1) return getLast();

    Node curr = head;

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

    return curr.data;
}
```

Remove last



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;

if (size == 1)

head = tail = null;

Remove last (ode)

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

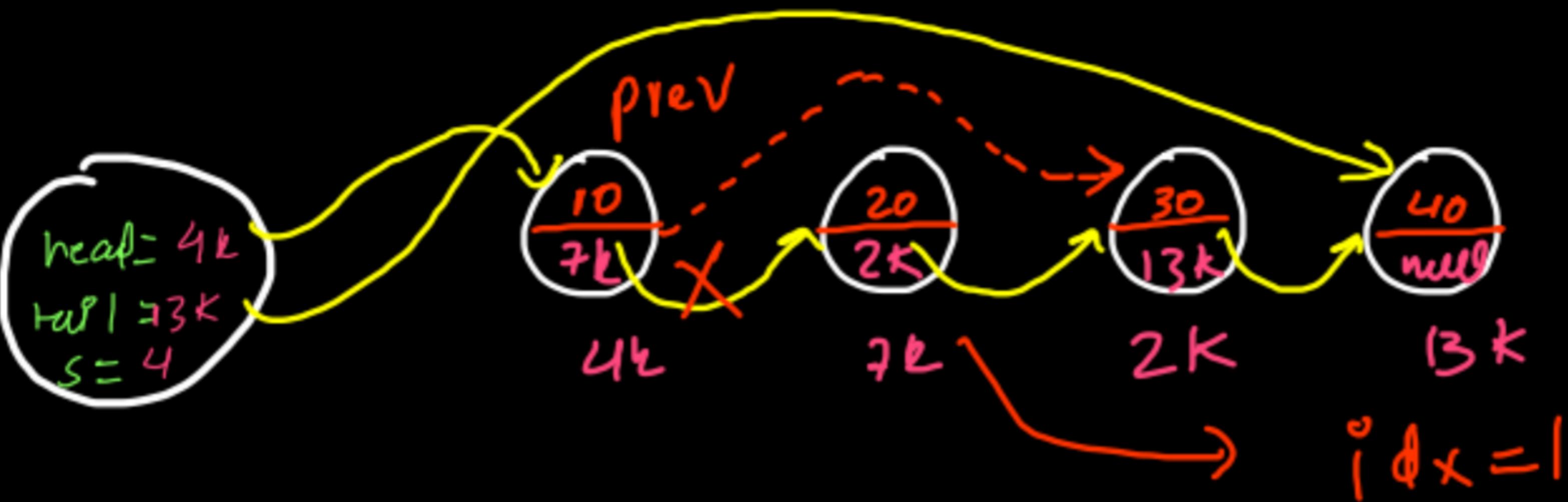
    if(size == 1) {
        head = tail = null;
        size--;
        return;
    }

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

    curr.next = null;
    tail = curr;

    size--;
}
```

Remove At



- 1) $idx == 0 \rightarrow \text{removeFirst}();$
 - 2) $idx == size - 1 \rightarrow \text{removeLast}();$
 - 3) remove At idx \rightarrow ① Get $(idx - 1)^{\text{th}}$ node
 $0 < idx < size - 1$ ② $prev.next = prev.next.next$
 ③ $size--;$
- $idx < 0 \text{ || } idx >= size \} \text{ invalid args}$

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

Remove At Code

```
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;
    }

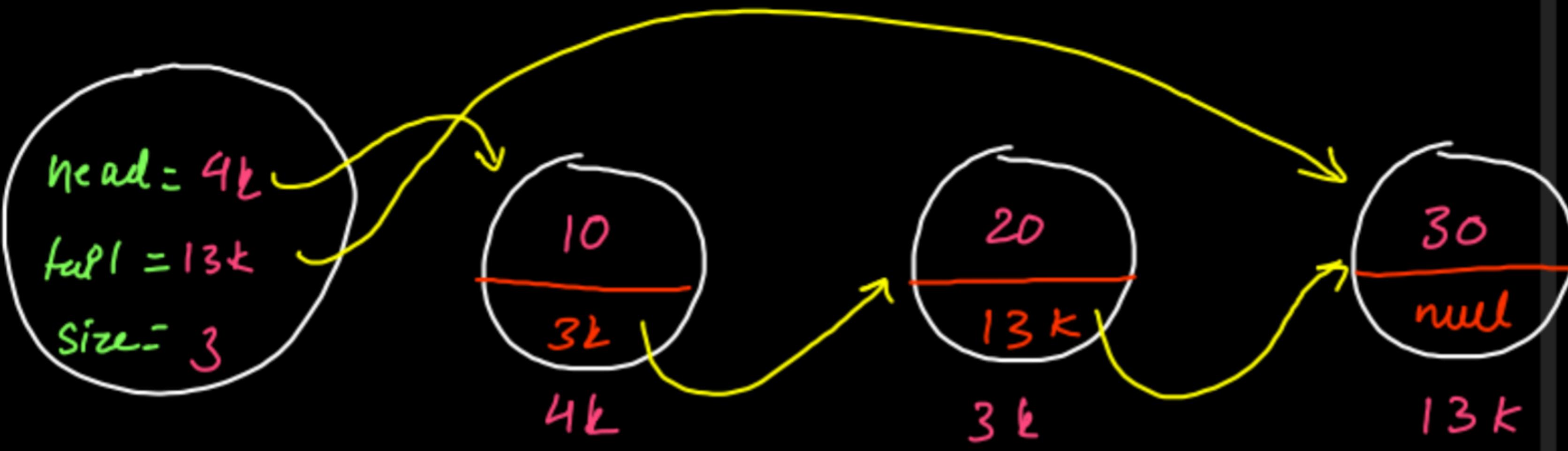
    if(index == 0) {
        removeFirst();
        return;
    }

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

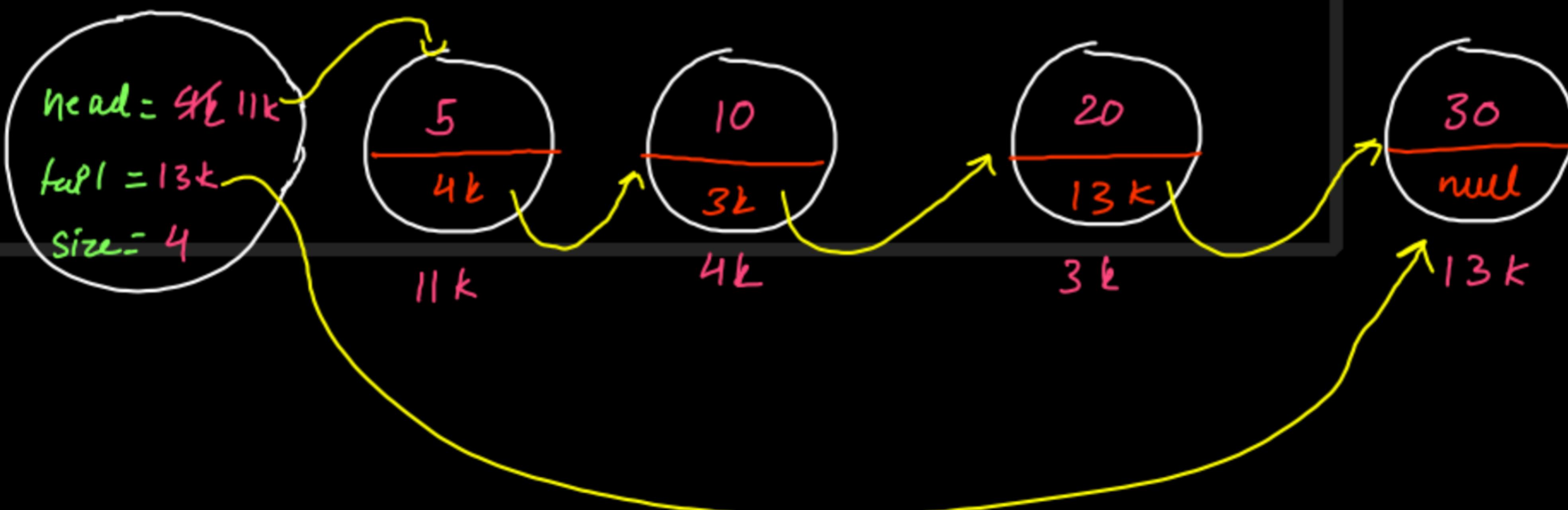
    Node curr = head;
    for(int i=1;i<index;i++) {
        curr = curr.next;
    }

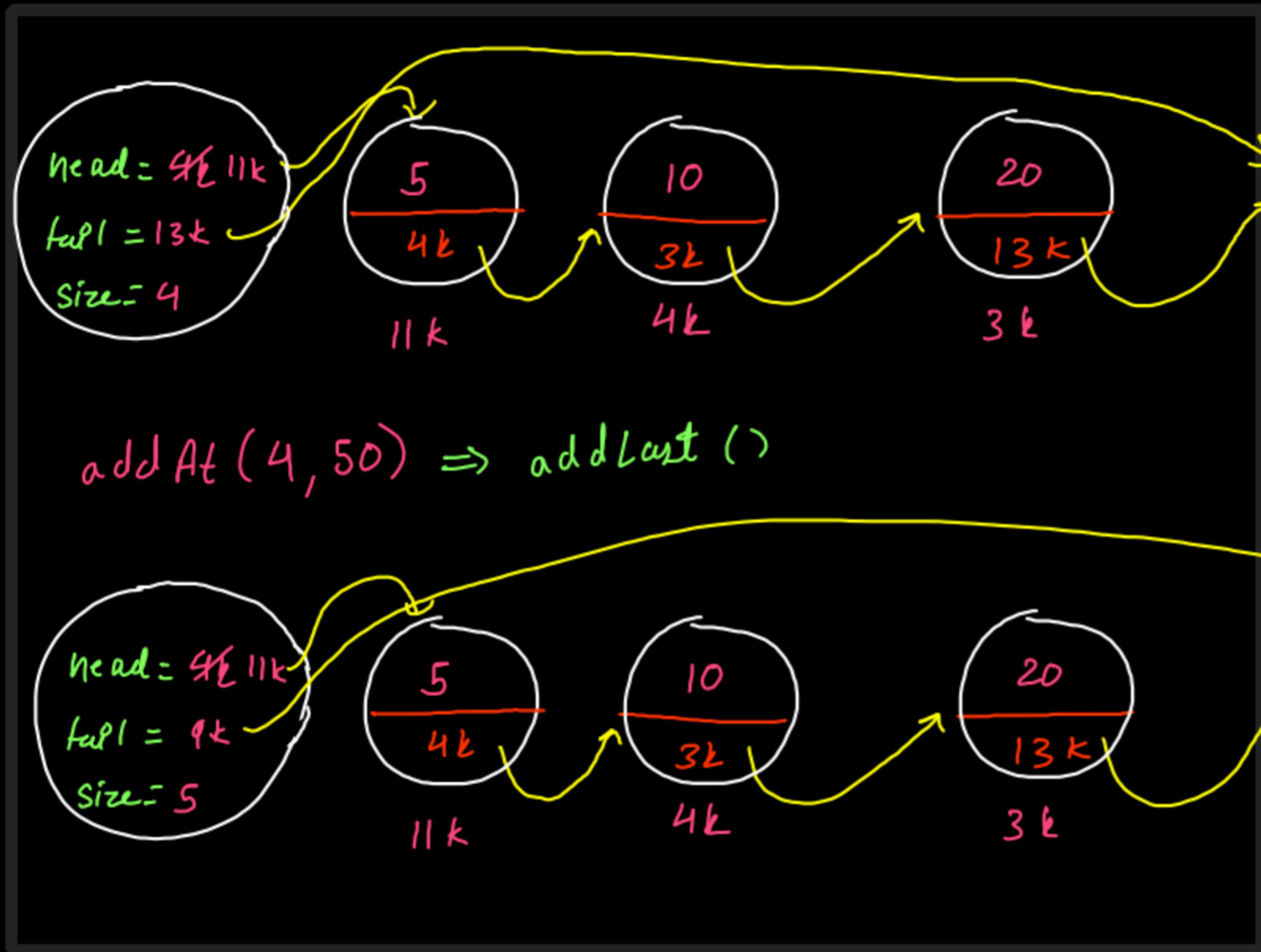
    curr.next = curr.next.next;
    size--;
}
```

Add At

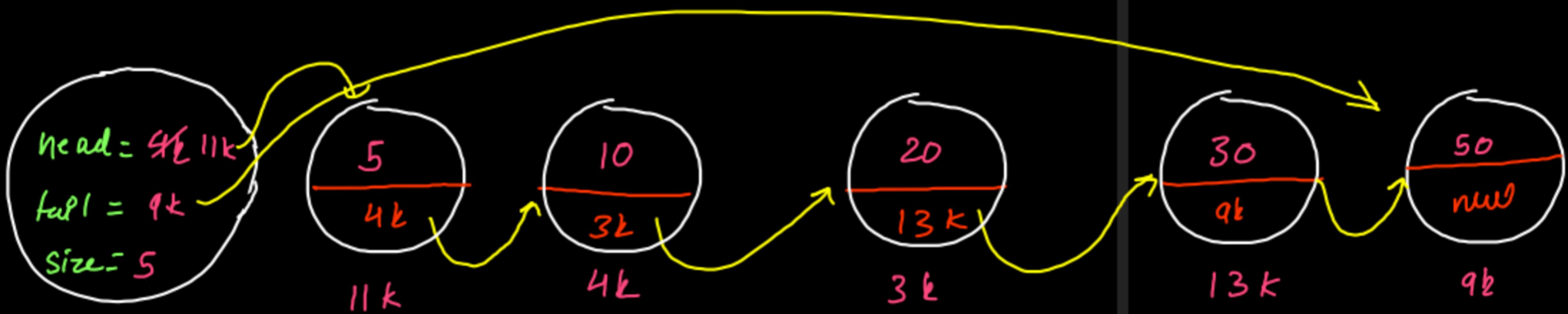


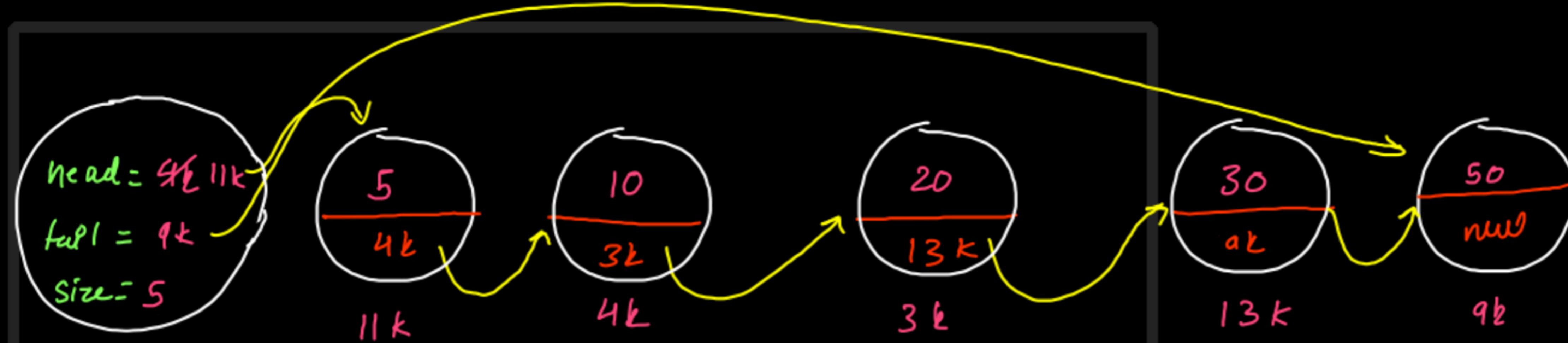
add At (0,5) → add first();



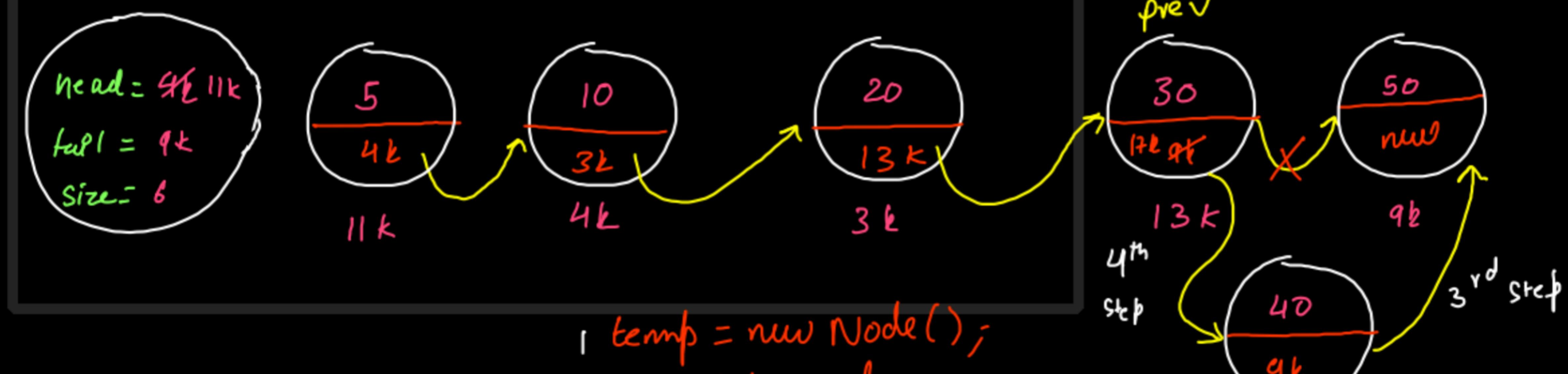


addAt(4, 50) \Rightarrow addLast()





add At (4, 40); \Rightarrow add At (size-1) is not add At tail



```

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

Add At Code

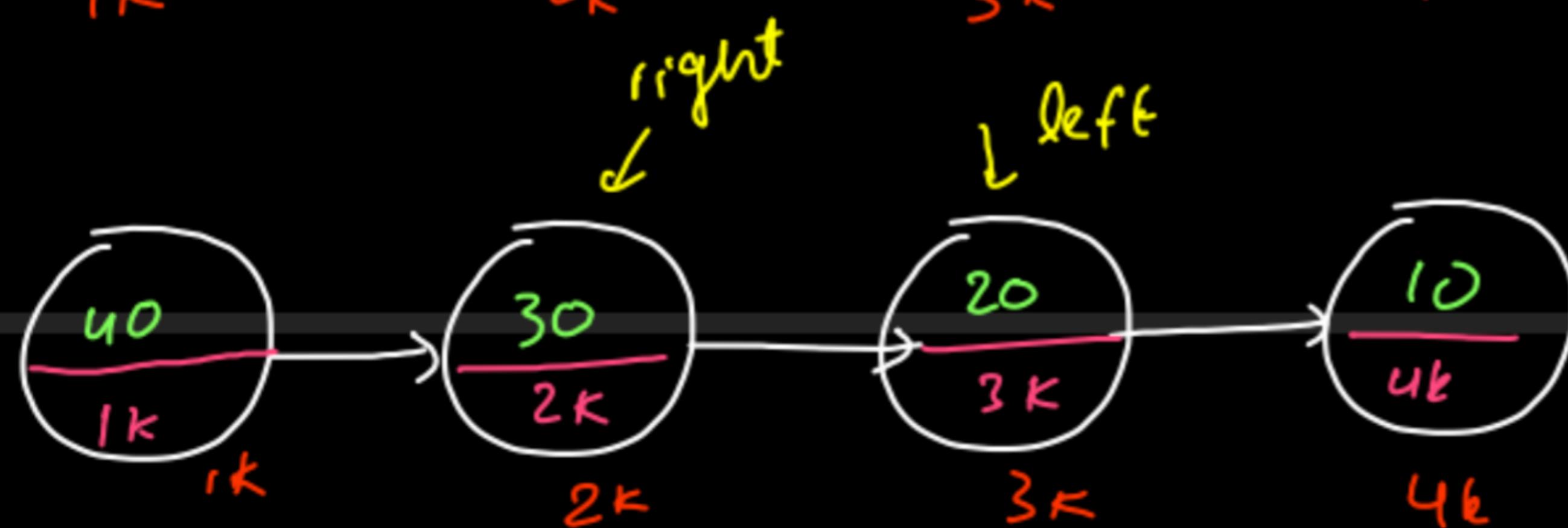
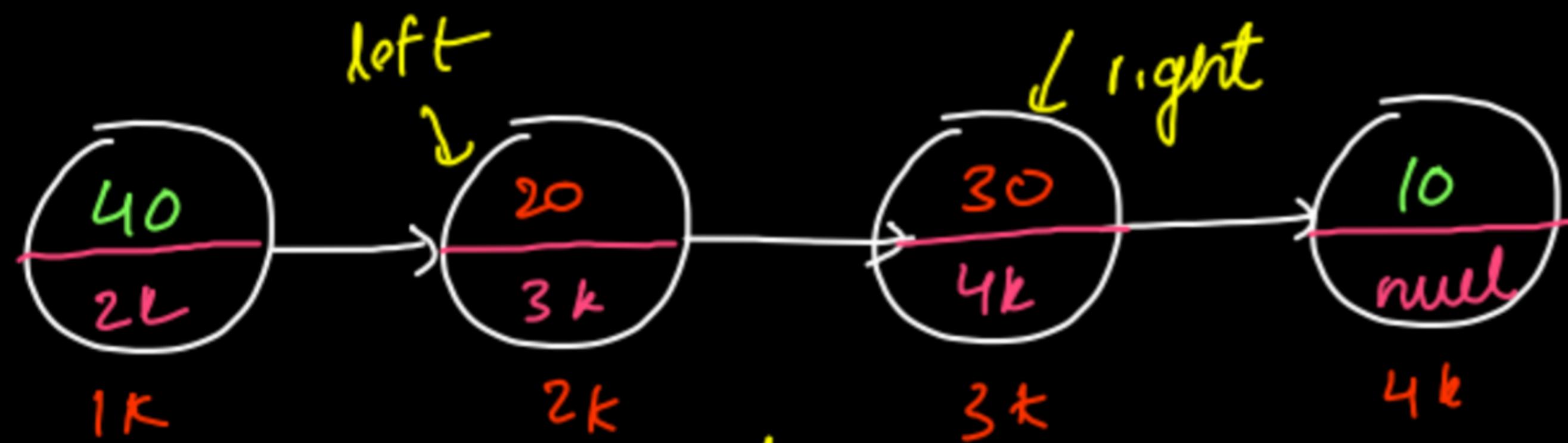
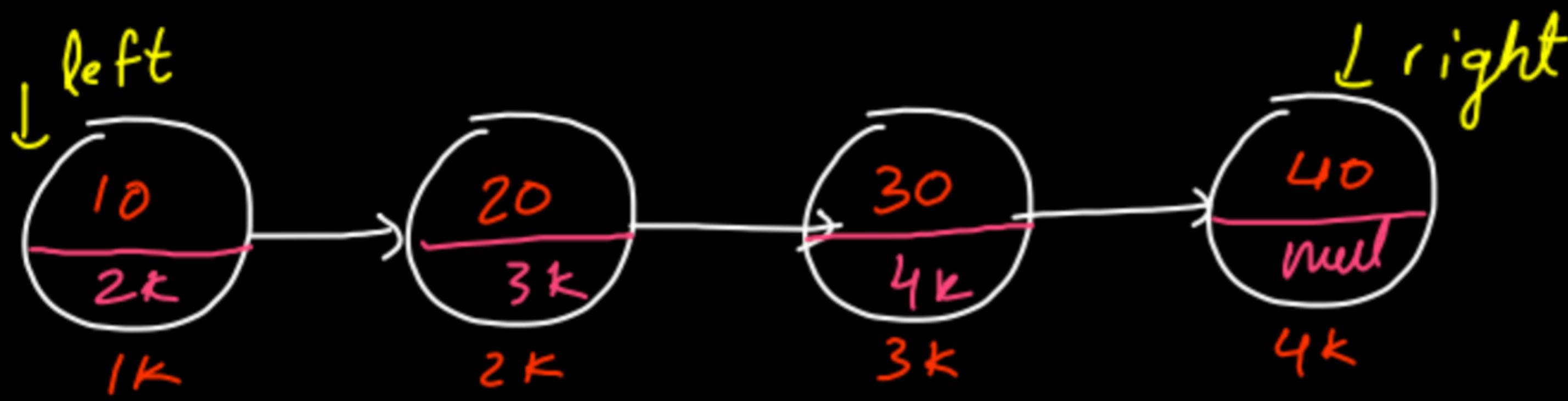
```
public void addAt(int idx, int val){  
    // write your code here  
  
    if(idx < 0 || idx > size) {  
        System.out.println("Invalid arguments");  
        return;  
    }  
  
    if(idx == 0) {  
        addFirst(val);  
        return;  
    }  
  
    if(idx == size) {  
        addLast(val);  
        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 $\frac{N}{2}$ times

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

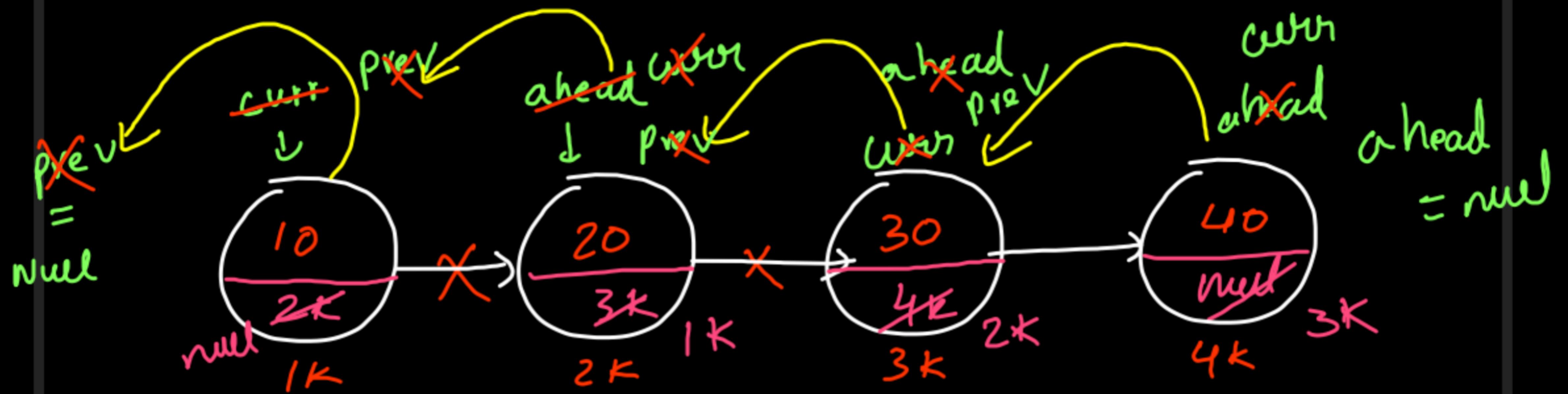
$$\Rightarrow TC = O(N^2)$$

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

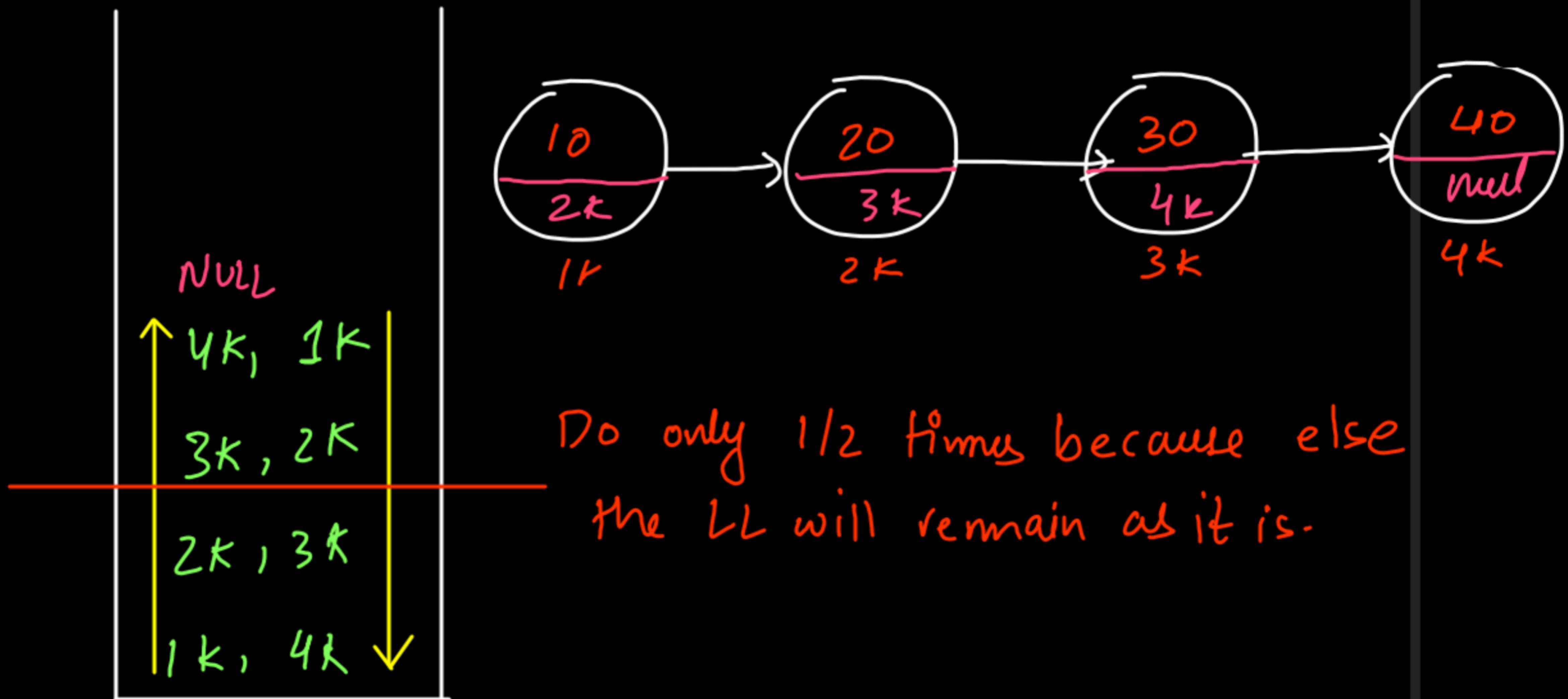
Reverse a LL Pointer Iterative Code

```
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



$l = null$	
$l = 6k \quad r = l = 1k \quad c = 5$	
$l = 5k \quad r = l = 1k \quad c = 4$	
$l = 4k \quad r = l = 1k \quad c = 3$	
$l = 3k \quad r = l = 1k \quad c = 2$	
$l = 2k \quad r = l = 1k \quad c = 1$	
$l = 1k \quad r = l = 1k \quad c = 0$	

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

This is wrong as $r = right.next$ 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 Rewrite using Return

~~$l = \text{null}$~~
 ~~$l = 6k \quad r = 1k$~~
 ~~$l = 5k \quad r = 1k \ 2k$~~
 ~~$l = 4k \quad r = 1k \ 3k$~~
 ~~$l = 3k \quad r = 1k \ 4k$~~
 ~~$l = 2k \quad r = 1k \ 5k$~~
 ~~$l = 1k \quad r = 1k \ 6k$~~

if (counter > size(z))
 ~~$l = 60$~~
 ~~$10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow 60 \rightarrow \text{null}$~~
 ~~1k 2k 3k 4k 5k 6k~~

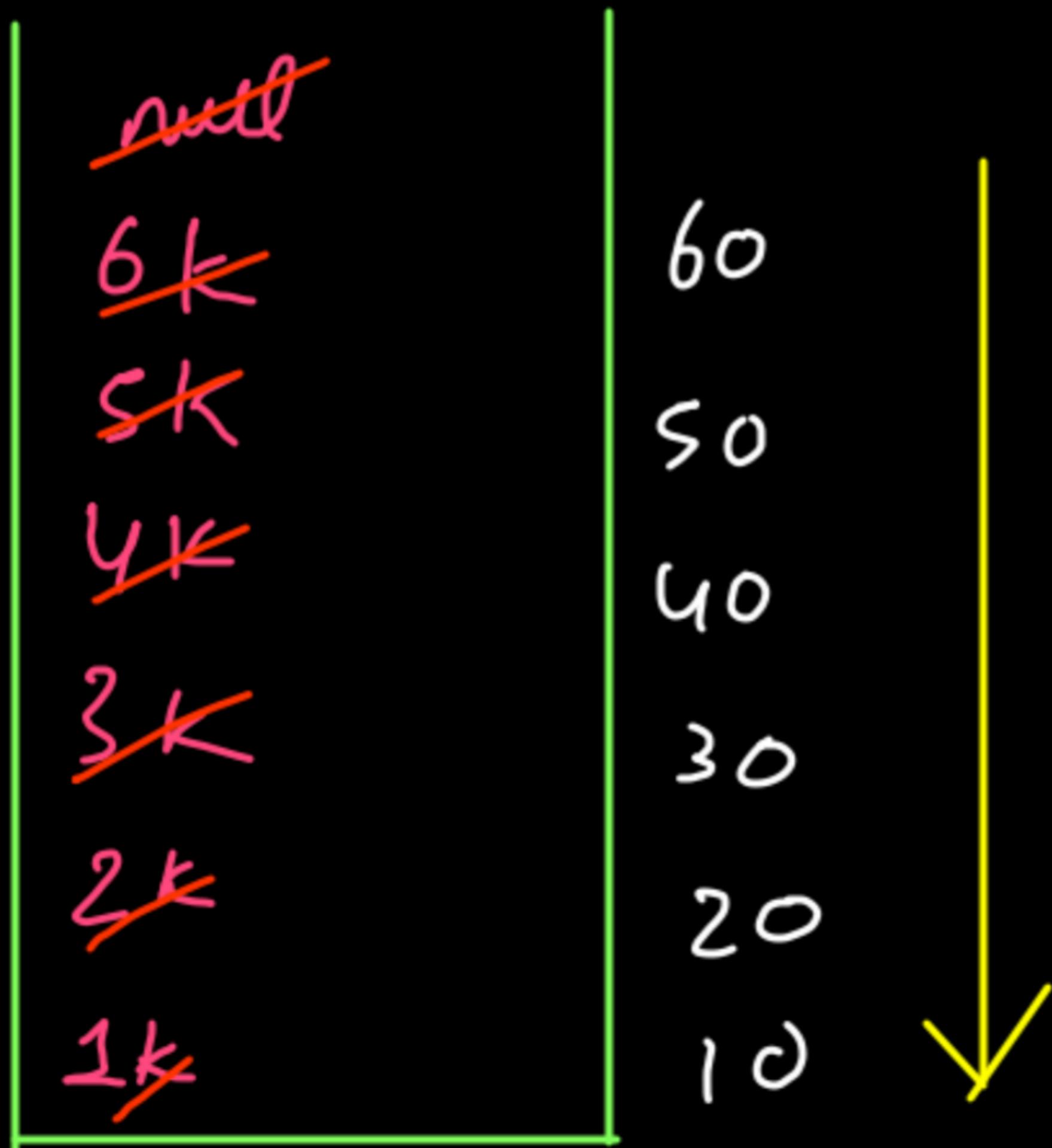
 → ~~$60 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow l = 20 \rightarrow 10 \rightarrow \text{null}$~~
 ~~1k 2k 3k 4k 5k 6k~~

 → ~~$60 \rightarrow 50 \rightarrow 30 \rightarrow 40 \rightarrow l = 30 \rightarrow 20 \rightarrow 10 \rightarrow \text{null}$~~
 ~~1k 2k 3k 4k 5k 6k~~

 → ~~$60 \rightarrow 50 \rightarrow 40 \rightarrow 30 \rightarrow 20 \rightarrow 10 \rightarrow \text{null}$~~
 ~~1k 2k 3k 4k 5k 6k~~

Display Reverse

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



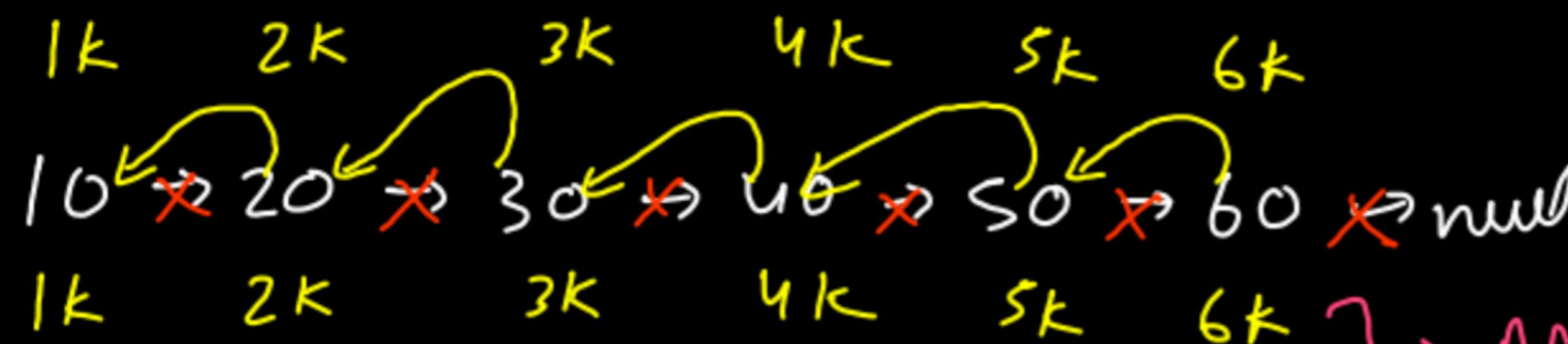
print in postorder

```
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();  
}
```

$T.C = O(N)$

Reverse Linked List Pointer Recursive

10 → 20 → 30 → 40 → 50 → 60 → null



After complete recursion

1K's next still points at 2K.

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

cwr	null
cwr	6K
cwr	5K
cwr	4K
cwr	3K
cwr	2K
cwr	1K

→ meri next node hai 6K. uska next mujhe bana do.
node.next = node



Reverse LL Pointer Recursive Code

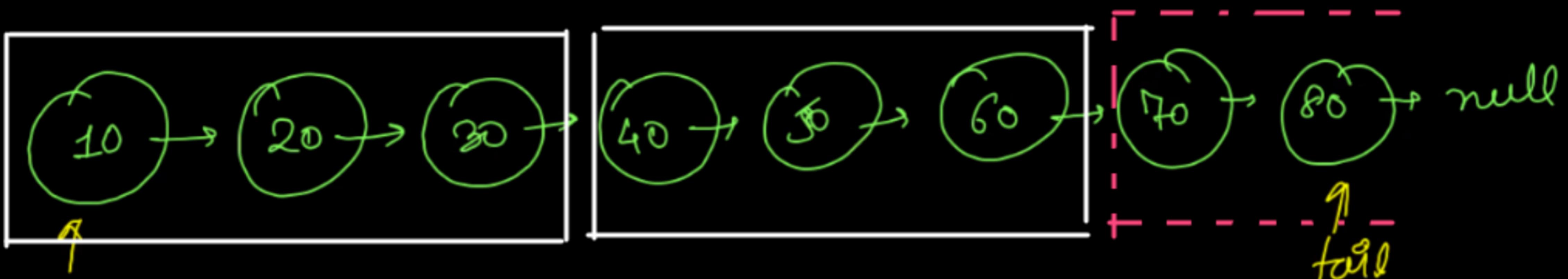
```
private void reversePRHelper(Node node){  
    // write your code here  
  
    if(node == null || node.next == null) {  
        return;  
    }  
    reversePRHelper(node.next);  
  
    //khud pe khade hoke apne next ka kaam karo  
    node.next.next = node;  
}  
  
public void reversePR(){  
    // 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;  
}
```

$T.C = O(N)$

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

Ye zavoori hai taaki agar head null ho to null pointer exception na aye .

K - Reverse in Linked List



$k = 3$

↓ After K Reverse

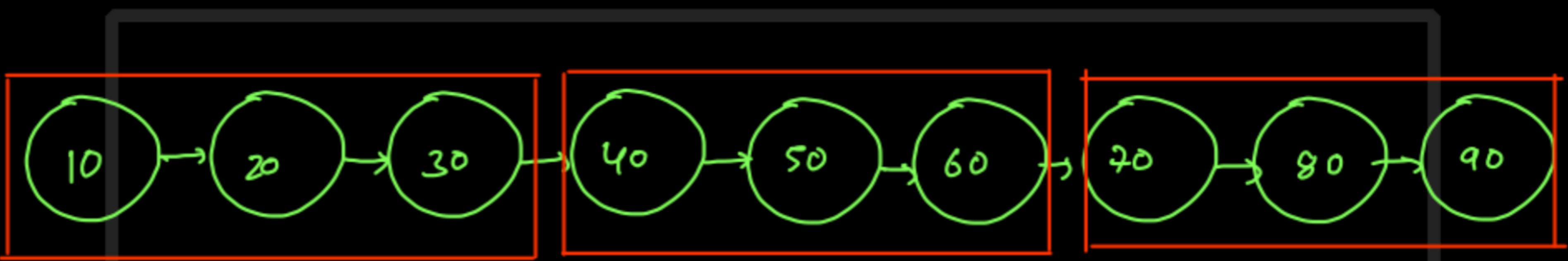


for each group

- Remove first from group
- Add first to curr LL

Initial state

input

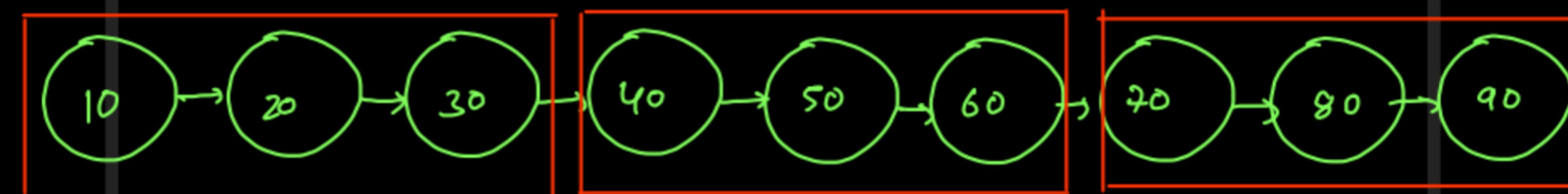


prev: empty list

curr: empty list

Reversing
1st group

input

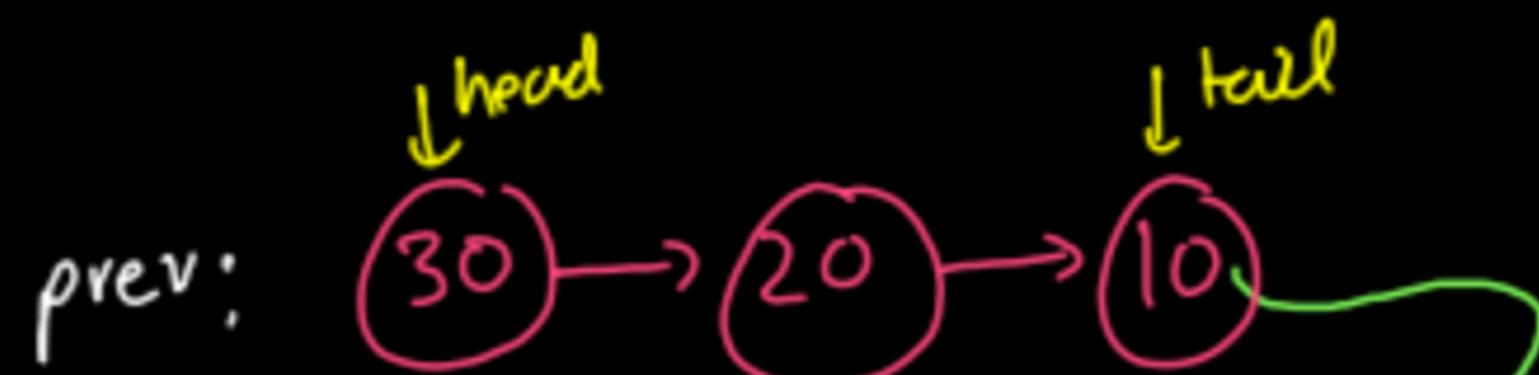


prev: empty list

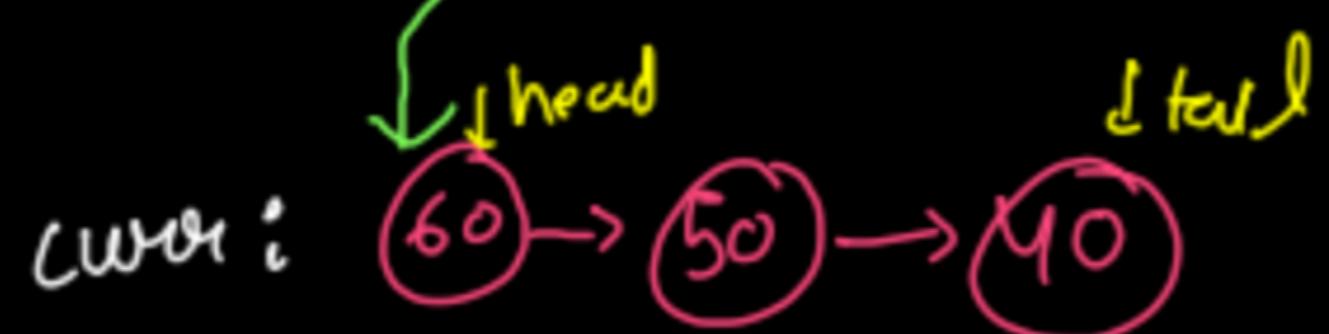
curr:
head
↓
30((30)) --> 20((20)) --> tail
↓
10((10))

Reversing
2nd Group

input

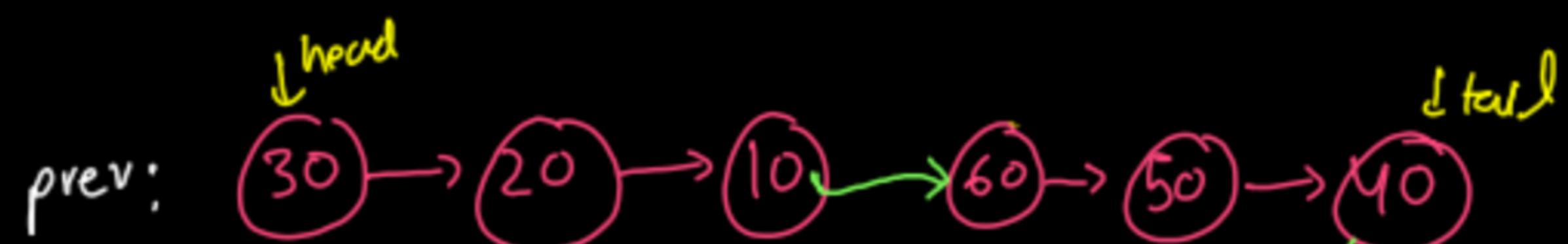


prev, tail-next
= curr, head



Reversing
3rd Group

input



prev, tail-next
= curr, head



Code for K Reverse

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

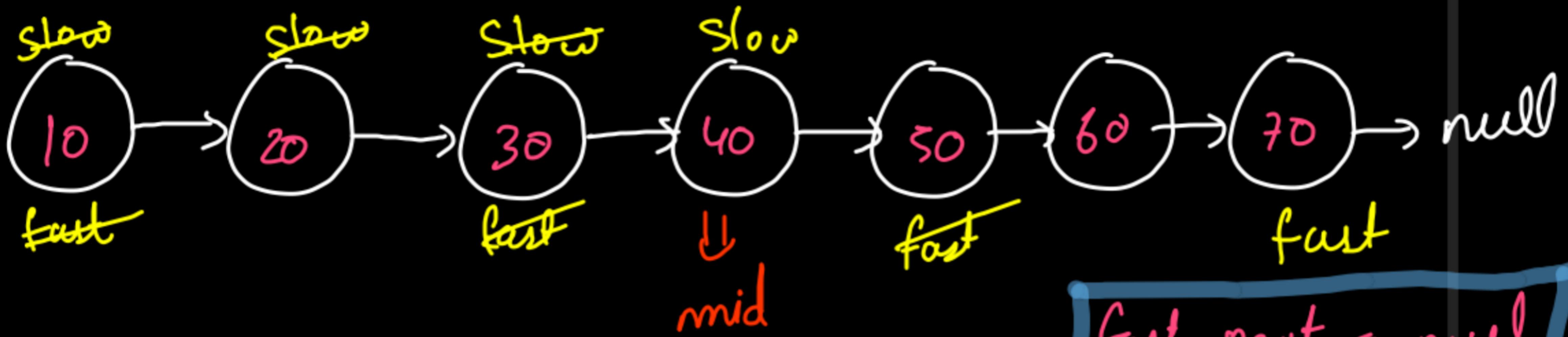
    while(size > 0) {
        LinkedList curr = new LinkedList();

        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);
            }
        }

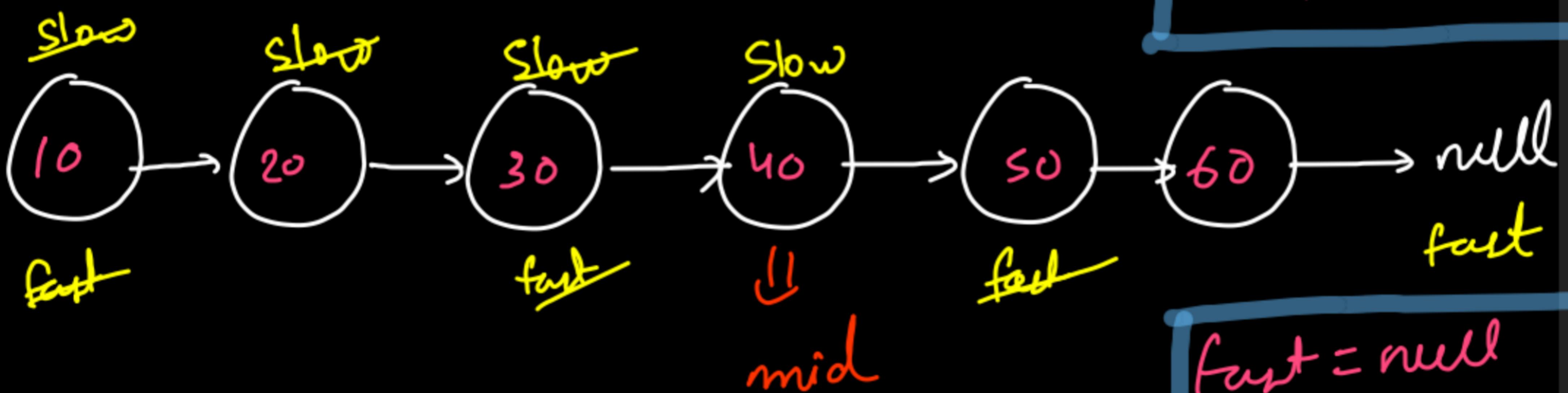
        if(prev.head == null) {
            prev = curr;
        } else {
            prev.tail.next = curr.head;
            prev.tail = curr.tail;
            prev.size += curr.size;
        }
    }

    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

```
'  
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

$$\text{Speed} = \text{Nodes/sec}$$

$$\text{Time} = t$$

$$\text{Dist} = ? \text{ say } x$$

fast

$$\text{Speed} = 2\text{Nodes/sec}$$

$$\text{Time} = t$$

$$\text{Dist} = N$$

$$\Rightarrow \text{Time} = \frac{\text{Dist}}{\text{Speed}}$$

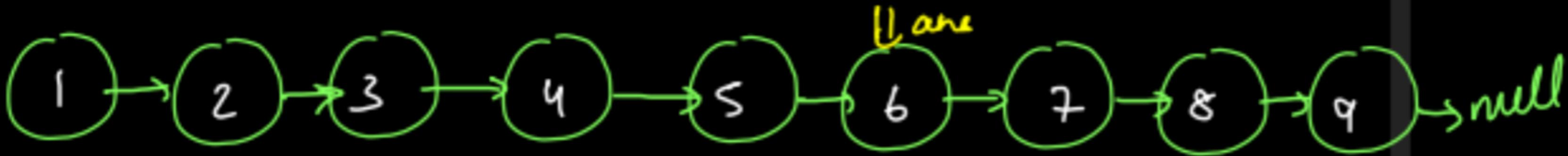
$$\frac{x}{1} = \frac{N}{2}$$

\Rightarrow

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

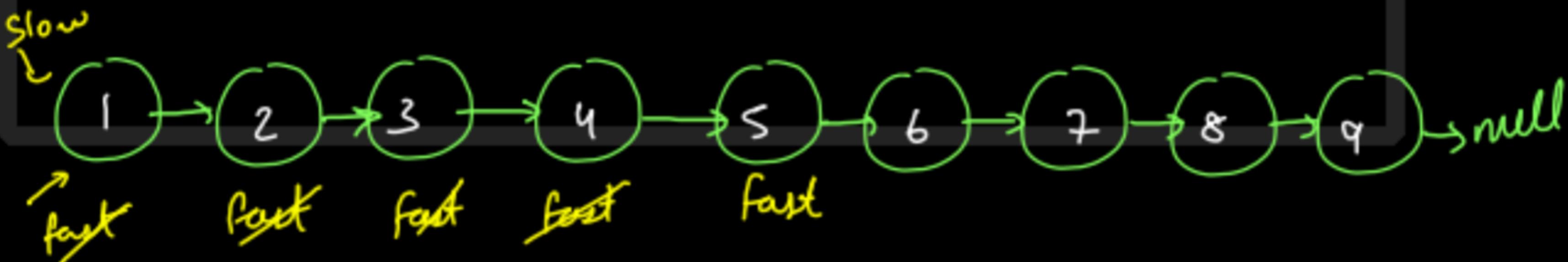
Kth Node from End

$k = 4$

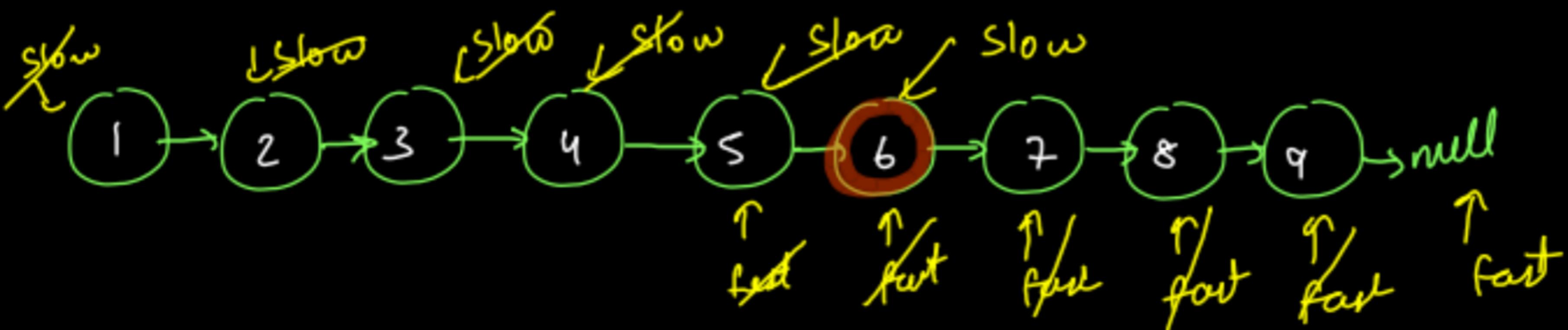


Idea is to have slow at kth node when fast becomes null. To do that, have 2 pointers with some speed but keep some initial distance b/w them.

→ Move fast k times initially



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



Kth 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

odd:  (T)

even:  (T)

 (f)

Approach:

- ① find middle node
- ② Reverse the second part
- ③ Compare both the parts

equal Yes (Palindrome)
not equal No

```
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;  
}
```

```
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;  
}
```