→ Linked - List

↳ linear data structure where elements (called Nodes) are stored in non-contiguous memory location.

↳ Each Node contains :-

① Data → the value stored in the node

② pointer (next) → a reference to the next node in the sequence.

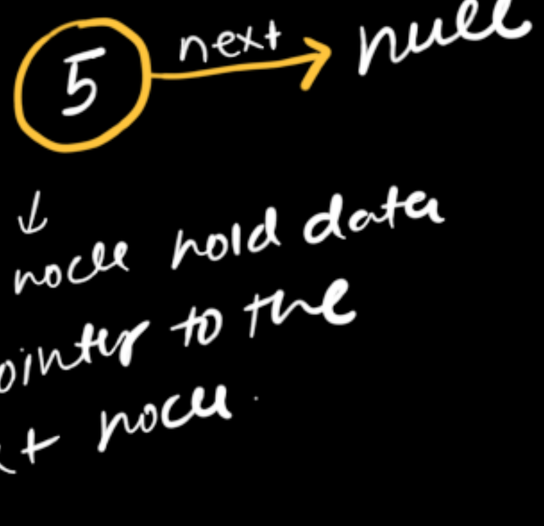
→ Types of Linked List :-

① Singly -LL : Node points to the next node only

② Doubly -LL : Nodes have pointers to both the next & previous node.

③ Circular-LL : The last node points back to the first node; can be singly or doubly.

→ Node - structure



↓

Each node hold data & pointer to the next node.

```
public class Node<T> {
    public T data;
    public Node<T> next;
    Node(T data) {
        this.data = data;
        this.next = null;
    }
}
```

→ Time & Space complexity

	with Head	with Tail
Insertion		
↳ • addFirst	O(1)	O(1)
• addLast	O(N)	O(1)
Deletion		
↳ • deleteFirst	O(1)	O(1)
• deleteLast	O(N)	O(N)
Searching / Traversing	O(N)	O(N)
Insertion / deletion at Index	O(N)	O(N)

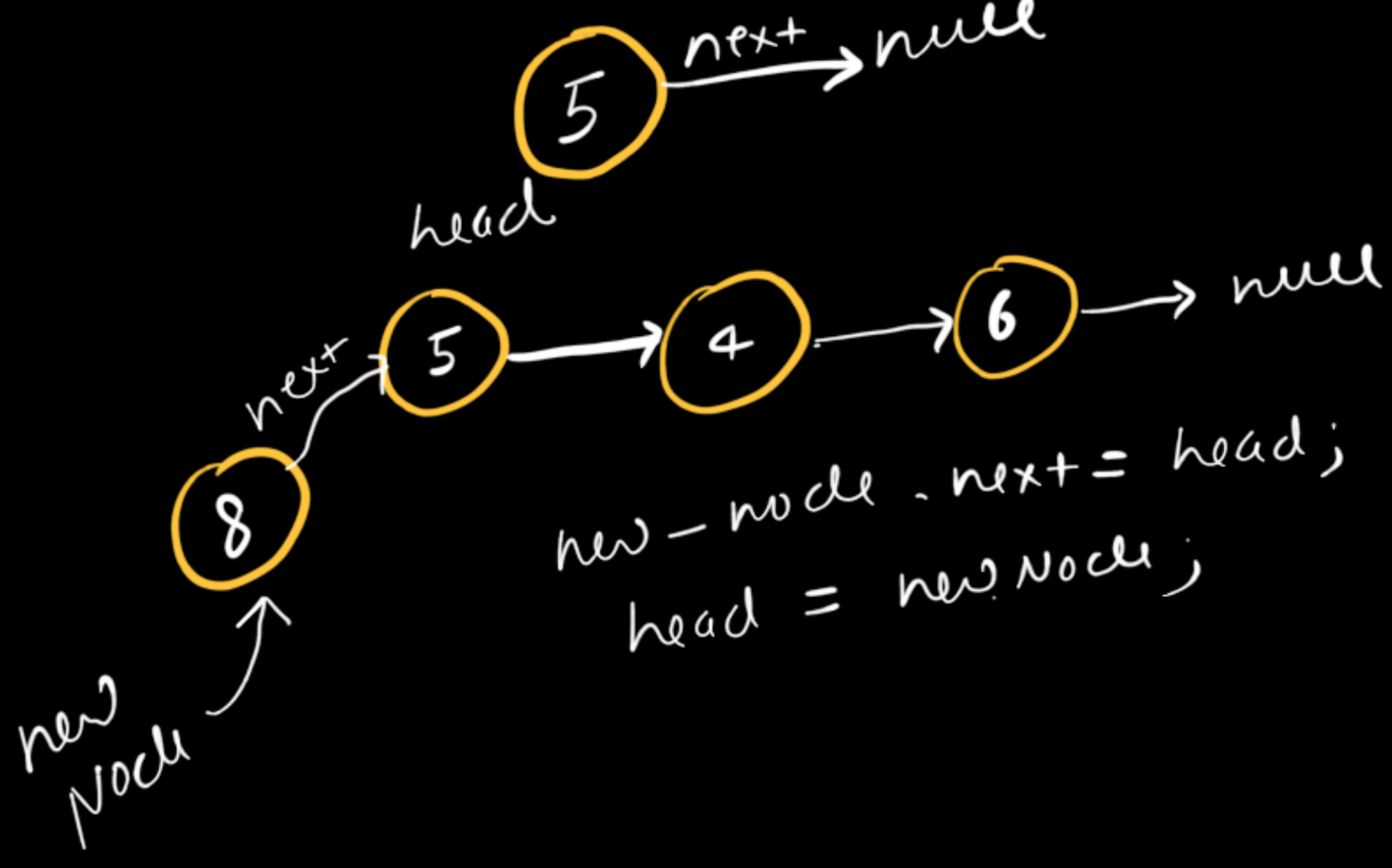
Space - complexity: O(1)

```java
LLNode.java
public class LLNode<T> {  8 usages  new *
    public T value;
    public LLNode<T> next;  6 usages

    public LLNode() {  no usages  new *
    }

    public LLNode(T value) {  3 usages  new *
        this.value = value;
        this.next = null;
    }

    public LLNode(T value, LLNode<T> next) {  n
        this(value);
        this.next = next;
    }

    public T getValue() { return value; }

    public LLNode<T> getHead() { return next; }

}
```

→ Insertion

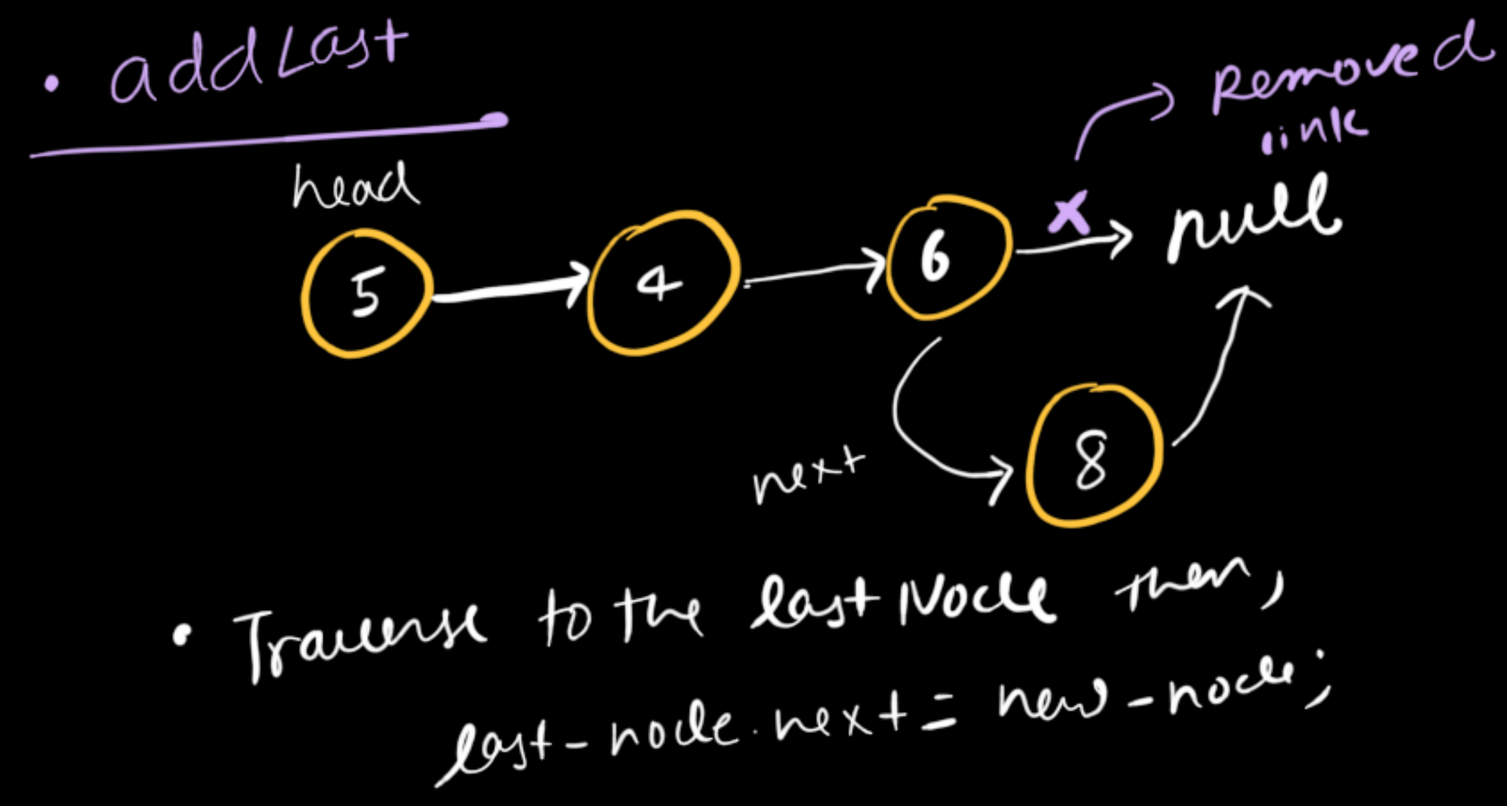
• addFirst

↳ Initially head is null → if(head == null){ head = new Node(5); return; }




```java
LinkedList.java
public void addFirst(T value) {  1 usage  ne
    if (head == null) {
        head = new LLNode<T>(value);
        return;
    }

    LLNode<T> node = new LLNode<T>(value);
    node.next = head;
    head = node;
}
```

• addLast

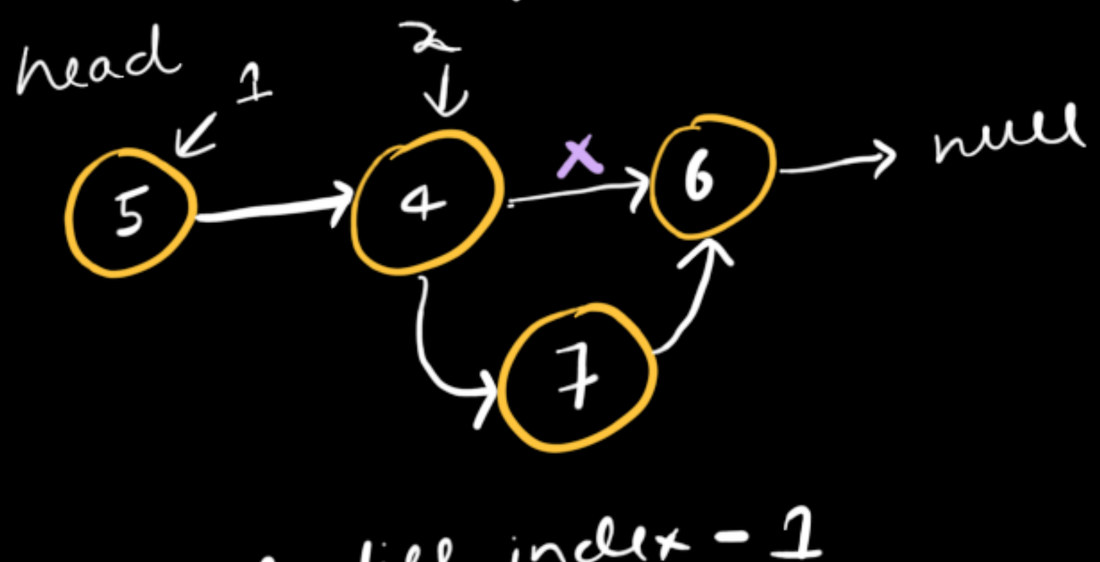


• Traverse to the last Node then,
last-node.next = new-node;

```java
LinkedList.java
public void addLast(T value) {  2 usag
    if (head == null) {
        addFirst(value);
        size++;
        return;
    }

    LLNode<T> temp = head;
    while (temp.next != null) {
        temp = temp.next;
    }
    temp.next = new LLNode<T>(value);
    size++;
}
```

→ Insertion at index

index = 3; value = 7



• Traverse till index - 1
↳ Front-node = prev-node.next;
prev-node.next = new-node;
new-node.next = front-node;

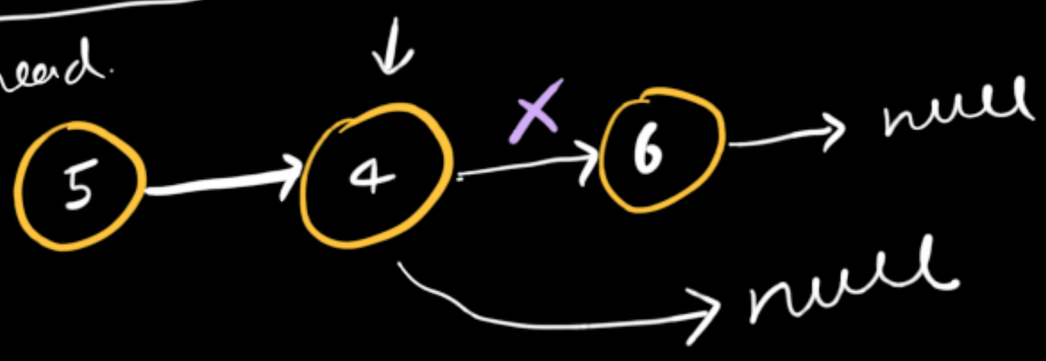
```java
LinkedList.java
public void insertAtIndex(int index, T value) {
    if (index == 0) {
        addFirst(value);
        return;
    }

    if (index == this.size) {
        addLast(value);
        return;
    }

    LLNode<T> temp = head;
    for (int i = 1; i < index; i++) {
        temp = temp.next;
    }

    LLNode<T> front = temp.next;
    LLNode<T> node = new LLNode<T>(value);
    temp.next = node;
    node.next = front;
    size++;
}
```

→ Deletion

• delete First

↳ head = head.next;

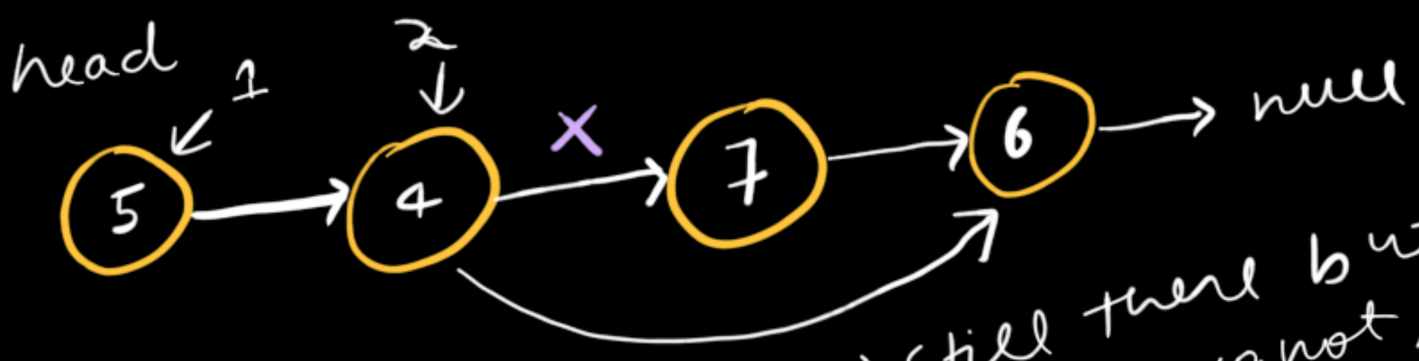
• deleteLast

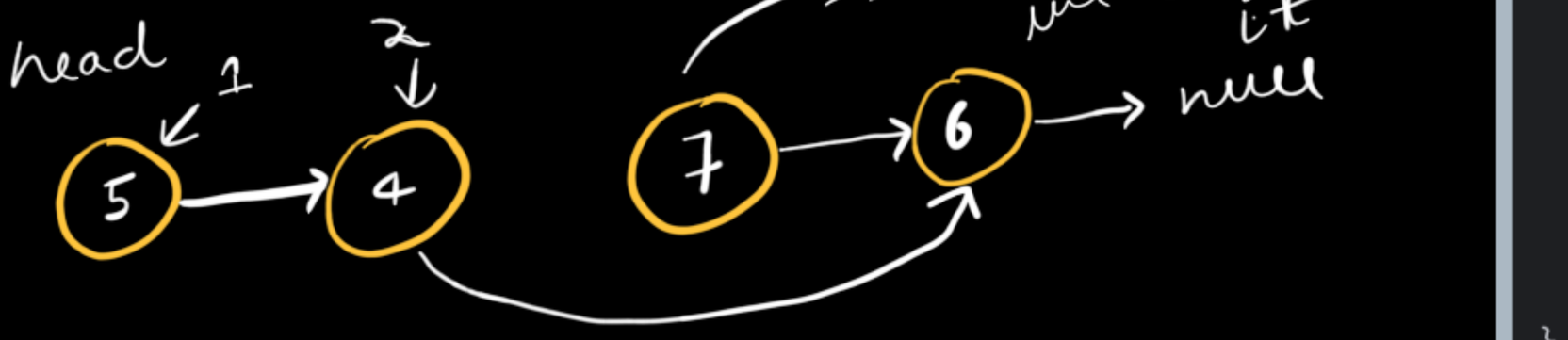


Traverse to second Last node
↳ while (temp.next.next != null)
↳ traverse
second-last-node.next = null;

→ deletion at index

index = 3; value = 7



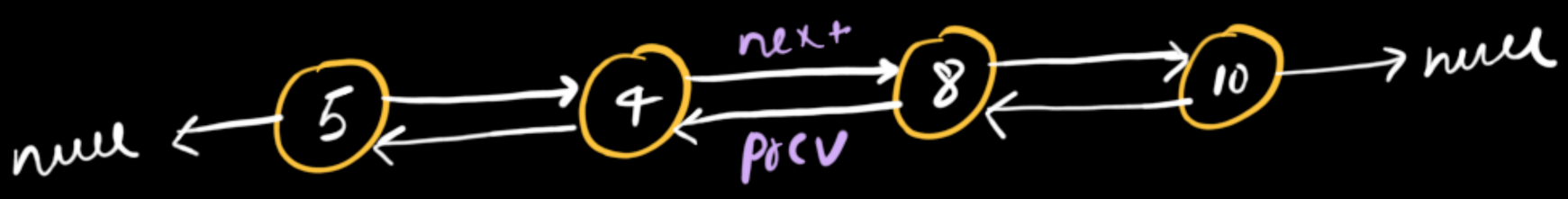


- Traverse till index - 1
  - ↳ Front-node = prev-node.next.next;
    del = prev-node.next;
    prev-node.next = null;
    return del;

LinkedList.java

```java
public LLNode<T> deleteFirst() {  2 usages  n
    if (head == null) {
        return null;
    }

    LLNode<T> temp = head;
    head = head.next;
    size--;
    return temp;
}

public LLNode<T> deleteLast() {  3 usages  ne
    if (head == null) {
        return null;
    }
    LLNode<T> temp = head;

    while (temp.next.next != null) {
        temp = temp.next;
    }
    LLNode<T> del = temp.next;
    temp.next = null;
    size--;
    return del;
}

public LLNode<T> deleteAtIndex(int index) {
    if (index == 0) {
        return deleteFirst();
    }

    if (index == this.size) {
        return deleteLast();
    }

    LLNode<T> temp = head;
    for (int i = 1; i < index; i++) {
        temp = temp.next;
    }

    LLNode<T> front = temp.next.next;
    LLNode<T> del = temp.next;
    temp.next = front;
    size--;
    return del;
}
```

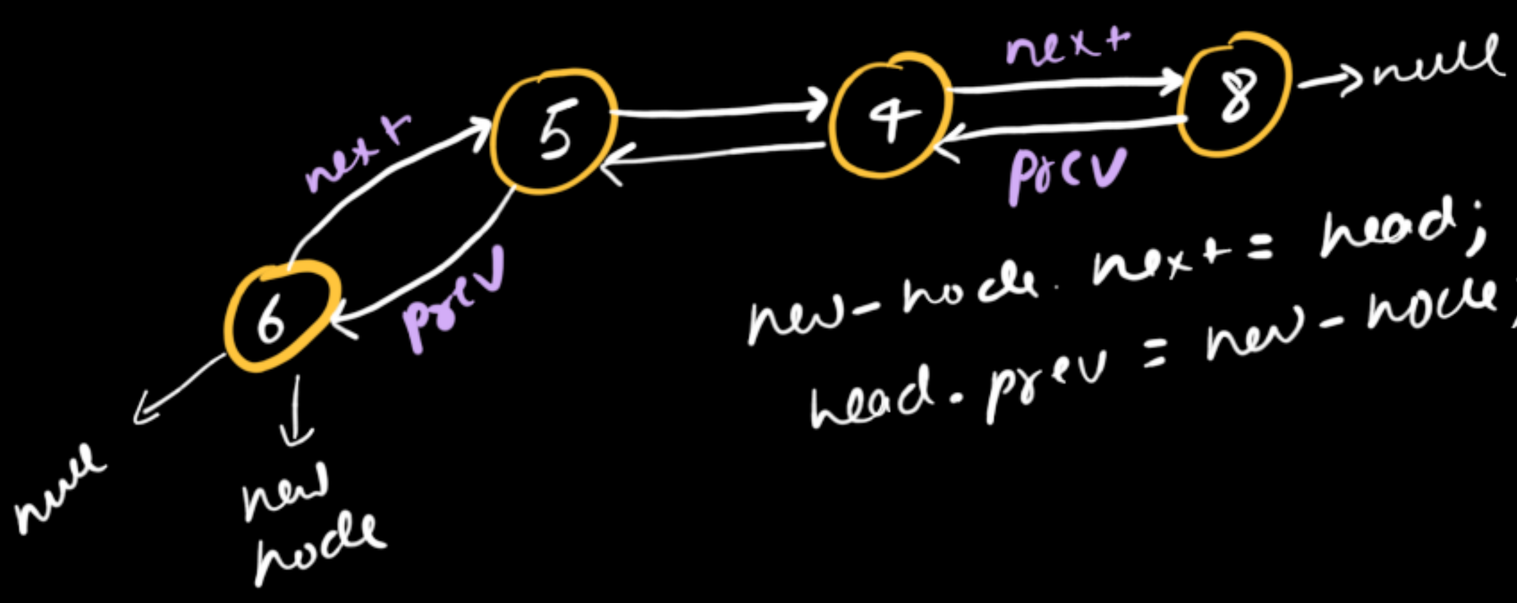
→ Doubly Linked List



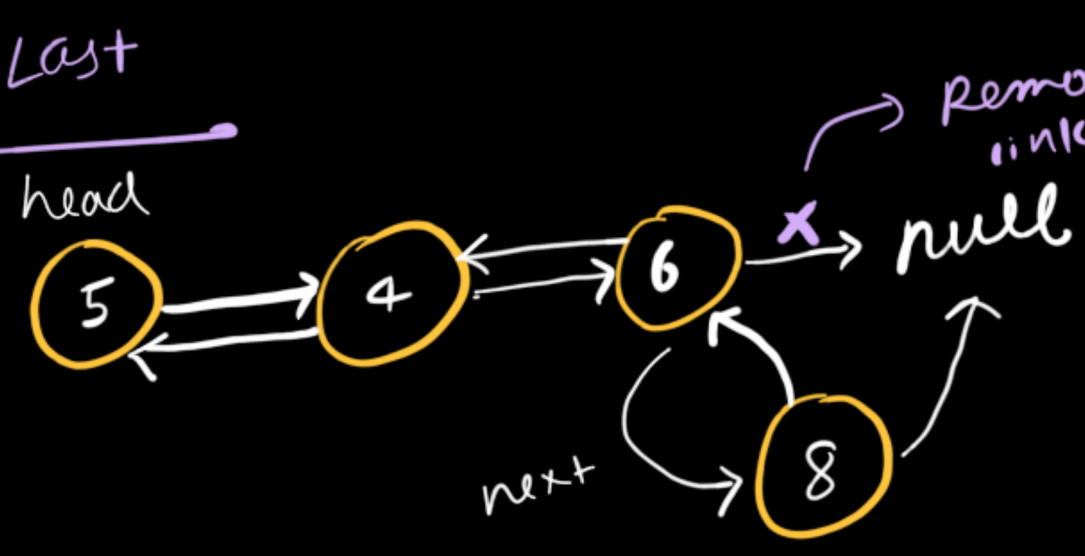
→ Insertion

- addFirst

↳ head is null initially.



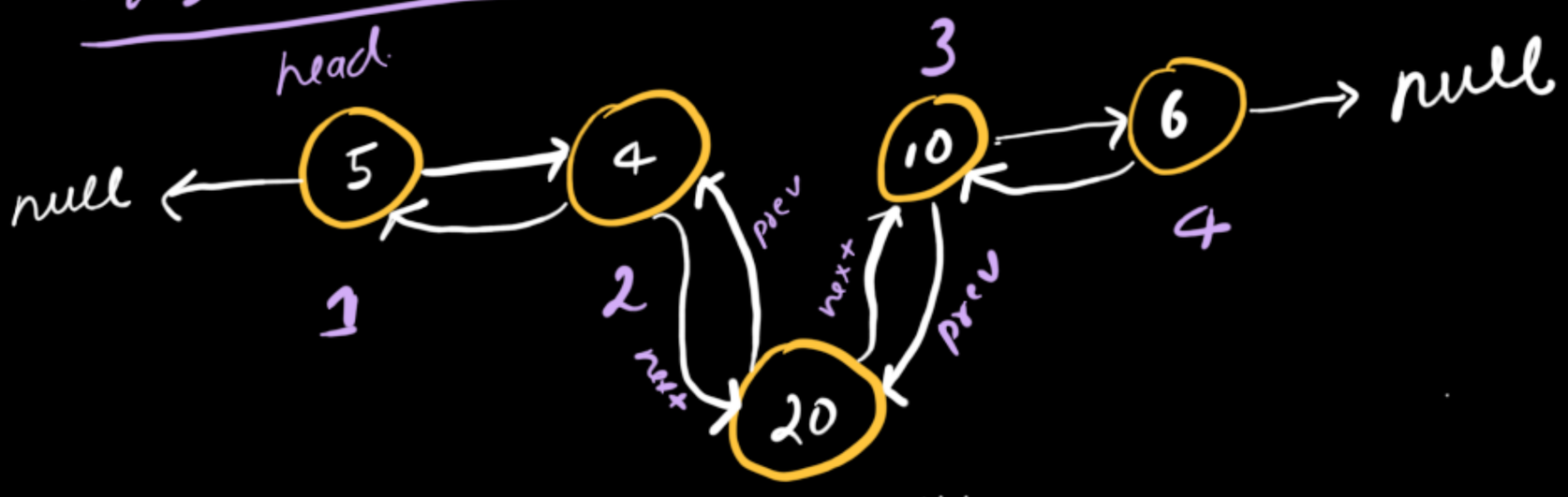
- addLast



- Traverse to the last Node then,

last_node.next = new_node;
new_node.prev = last_node;

- insertAtIndex

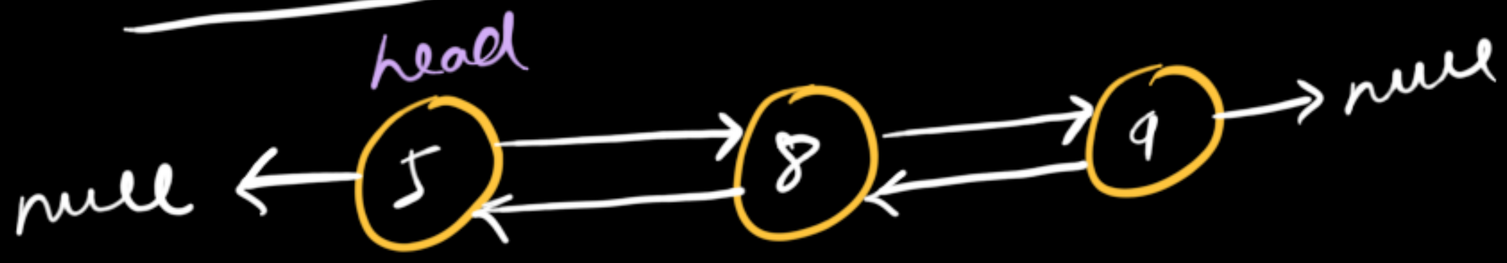


front = temp.next;
new_node.next = front;
front.prev = new_node;
temp.next = new_node;
new_node.prev = temp;

```java
DoublyLinkedList.java
public void addFirst(T value) {  6 usages  new *
    if (head == null) {
        head = new DLLNode<T>(value);
        size++;
        return;
    }

    DLLNode<T> temp = head;
    DLLNode<T> node = new DLLNode<>(value);
    node.next = temp;
    temp.prev = node;
    head = node;
    size++;
}

public void addLast(T value) {  3 usages  new *
    if (head == null) {
        addFirst(value);
        return;
    }

    DLLNode<T> temp = head;
    while (temp.next != null) {
        temp = temp.next;
    }

    DLLNode<T> node = new DLLNode<>(value);
    temp.next = node;
    node.prev = temp;
    size++;
}

public void insertAtIndex(int index, T value) {
    if (index < 0 || index > size) return;
    if (index == 0) {
        addFirst(value);
        return;
    }

    if (index == this.size) {
        addLast(value);
        return;
    }

    DLLNode<T> temp = head;
    for (int i = 0; i < index - 1; i++) {
        temp = temp.next;
    }

    DLLNode<T> front = temp.next;
    DLLNode<T> node = new DLLNode<>(value);
    node.next = front;
    front.prev = node;
    temp.next = node;
    node.prev = temp;
    size++;
}
```

→ Deletion

• deleteFirst



head = head.next;

• deleteLast

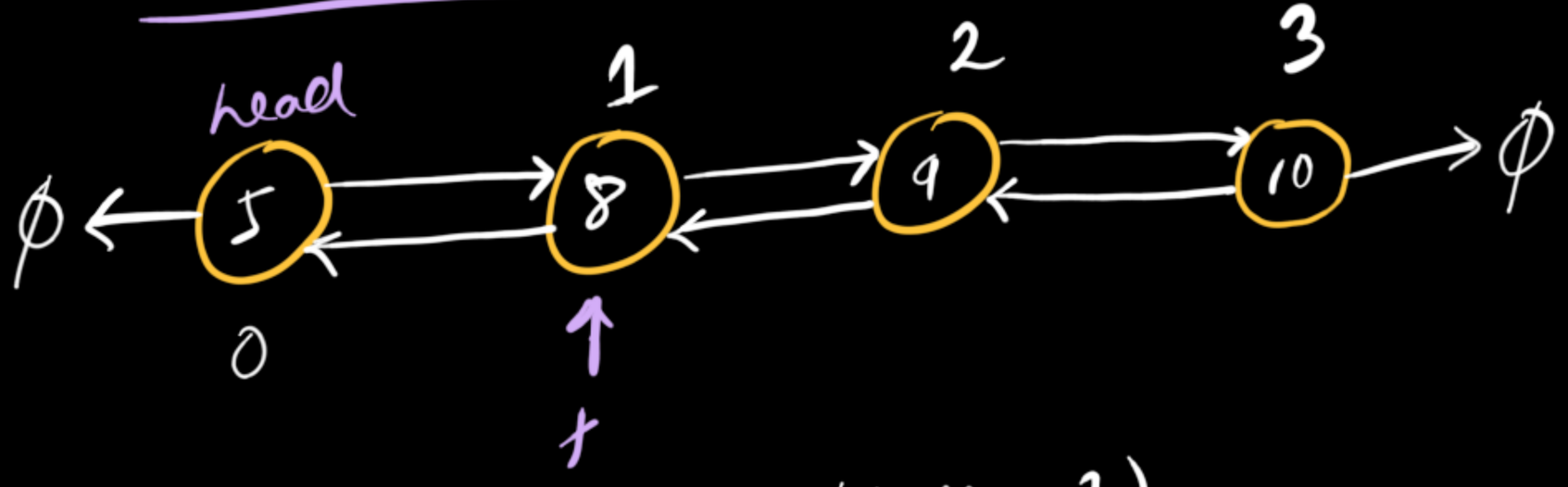
Traverse till temp.next.next != null

front = temp.next;

temp.next = null;

front.prev = null

• deleteAtIndex



traverse till (index - 1)

front_sec = temp.next.next;

front = temp.next;

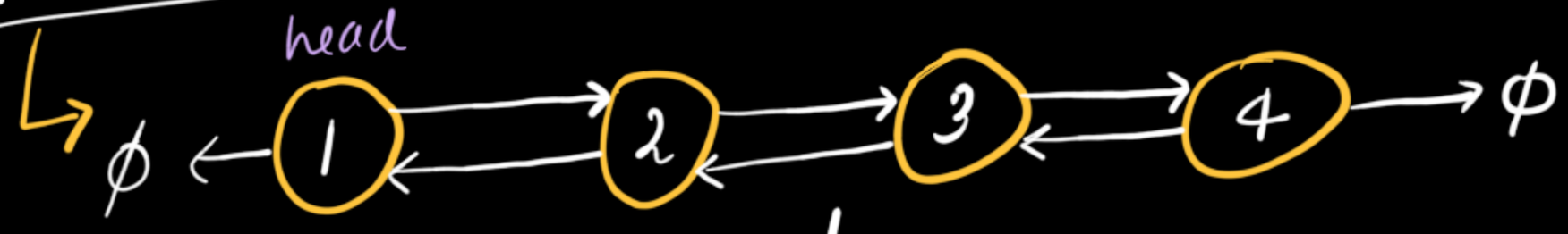
temp.next = front_second;

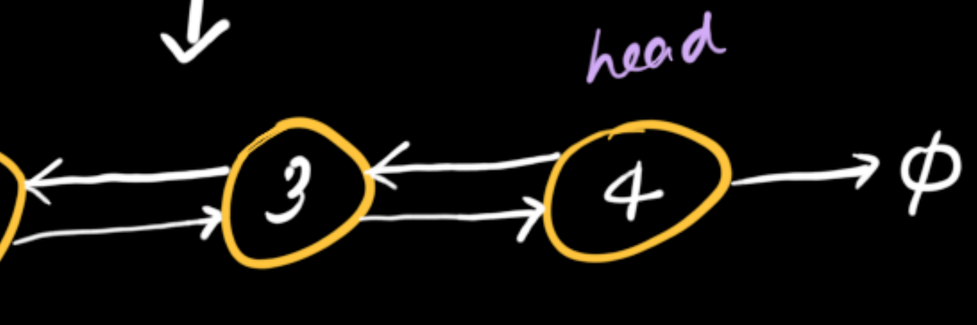
front.prev = null;

front_sec.prev = temp;

```java
DoublyLinkedList.java
public DLLNode<T> deleteFirst() {  2 usages  new *
    if (head == null) return null;

    DLLNode<T> temp = head;
    head = head.next;

    if (head != null) head.prev = null;

    temp.next = null;
    size--;
    return temp;
}

public DLLNode<T> deleteLast() {  2 usages  new *
    if (head == null) return null;

    if (head.next == null) {
        DLLNode<T> temp = head;
        head = null;
        size--;
        return temp;
    }

    DLLNode<T> temp = head;
    while (temp.next.next != null) {
        temp = temp.next;
    }

    DLLNode<T> del = temp.next;
    temp.next = null;
    del.prev = null;

    size--;
    return del;
}

public DLLNode<T> deleteAtIndex(int index) {  1 u
    if (index < 0 || index >= size) return null;

    if (index == 0) return deleteFirst();
    if (index == size - 1) return deleteLast();

    DLLNode<T> temp = head;
    for (int i = 0; i < index - 1; i++) {
        temp = temp.next;
    }

    DLLNode<T> del = temp.next;
    DLLNode<T> front = del.next;

    temp.next = front;
    if (front != null) front.prev = temp;

    del.prev = null;
    del.next = null;

    size--;
    return del;
}
```

→ Reverse a doubly Linked List

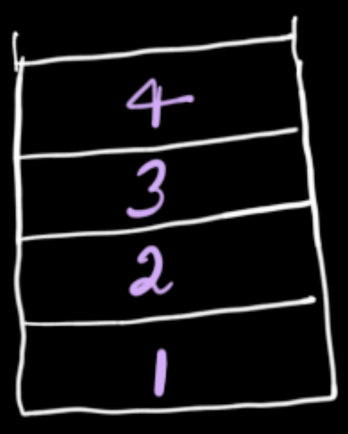




→ Approach-1

- we traverse the linked list & push element in stack.

```
while(temp != null){
  stack.push(temp.value)
  temp = temp.next;
}
```



Stack
(LIFO)

- Again we start from head, update the value of each node by popping the stack.

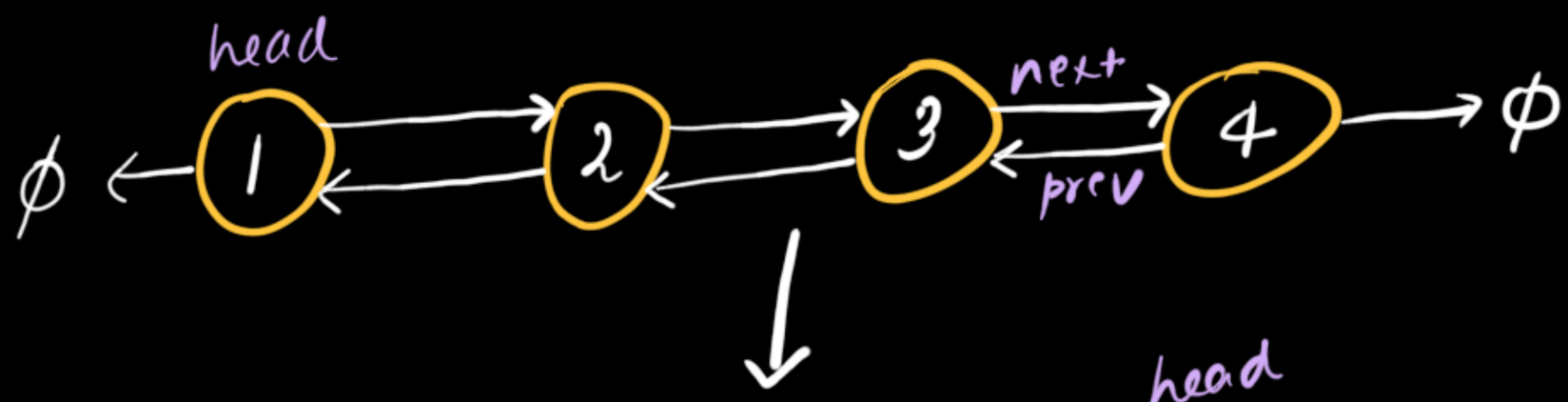
```
temp = head
while(temp != null){ // !stack.isEmpty()
  temp.value = stack.pop();
  temp = temp.next;
}
```

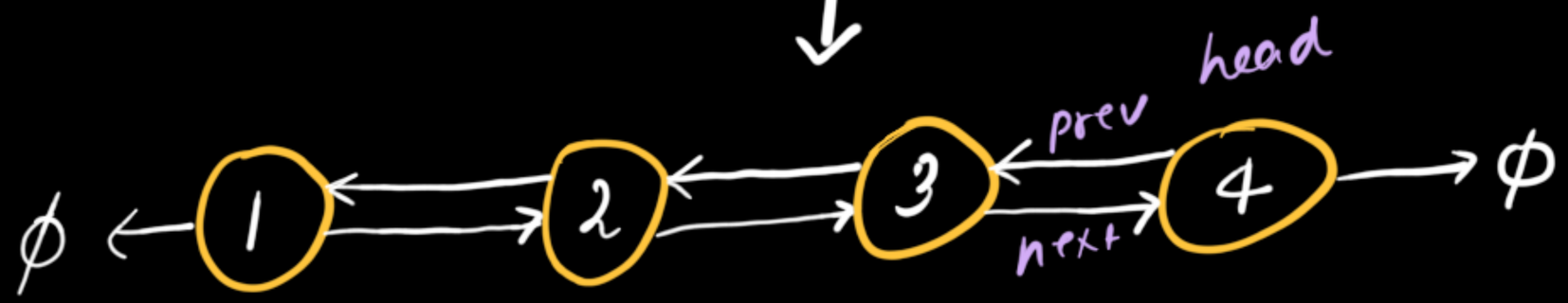
TC: O(2N)
SC: O(N)

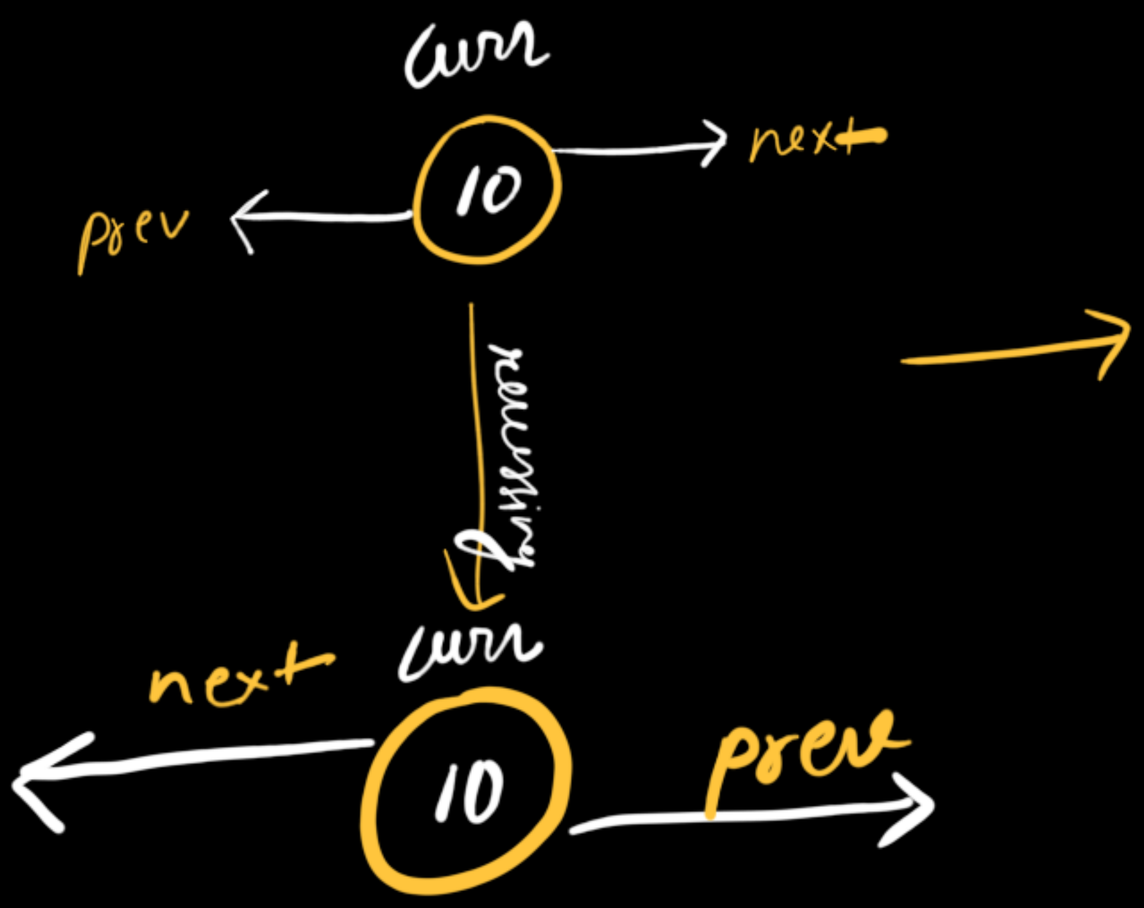
Part1.java

```java
public static DLLNode<Integer> reverseDLL(DLLNode<Integer> head) {
    if (head == null || head.next == null) {
        return head;
    }

    Stack<Integer> stack = new Stack<>();
    DLLNode<Integer> temp = head;

    while (temp != null) {
        stack.push(temp.value);
        temp = temp.next;
    }
    temp = head;

    while (!stack.isEmpty()) {
        temp.value = stack.pop();
        temp = temp.next;
    }

    return head;
}
```

→ Approach - 2



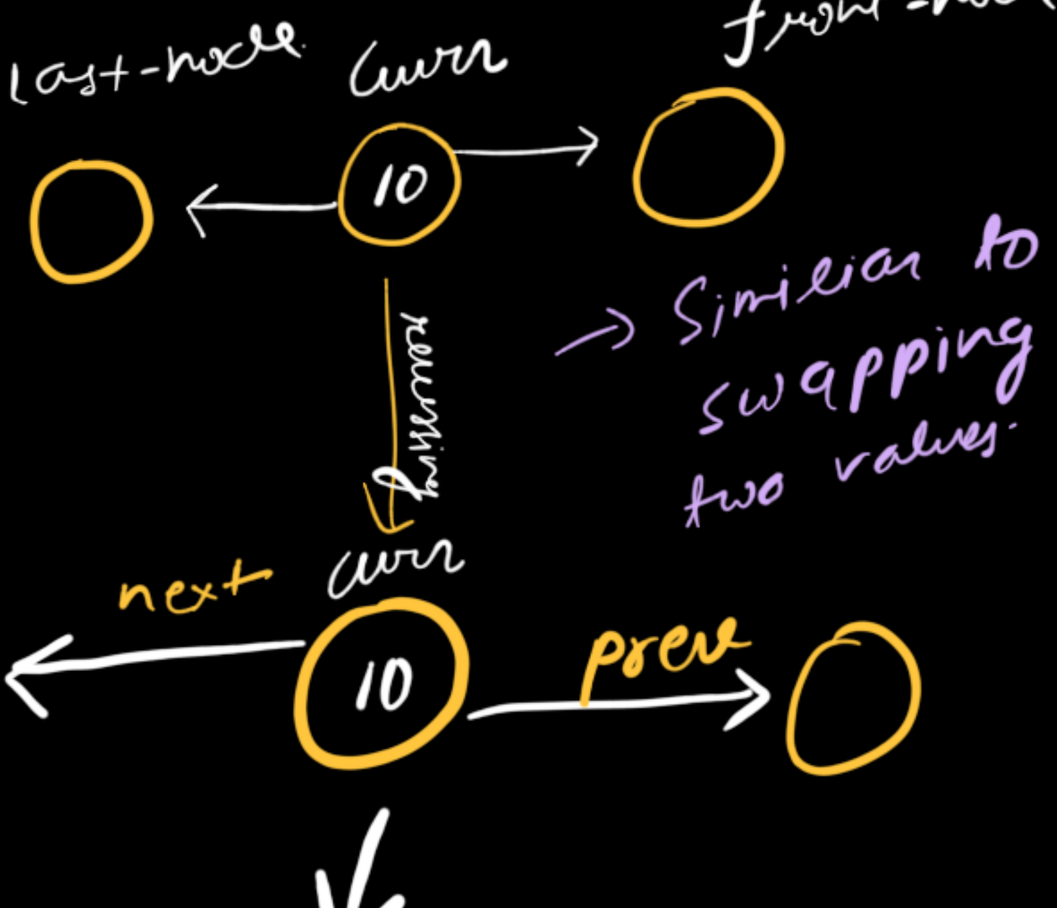




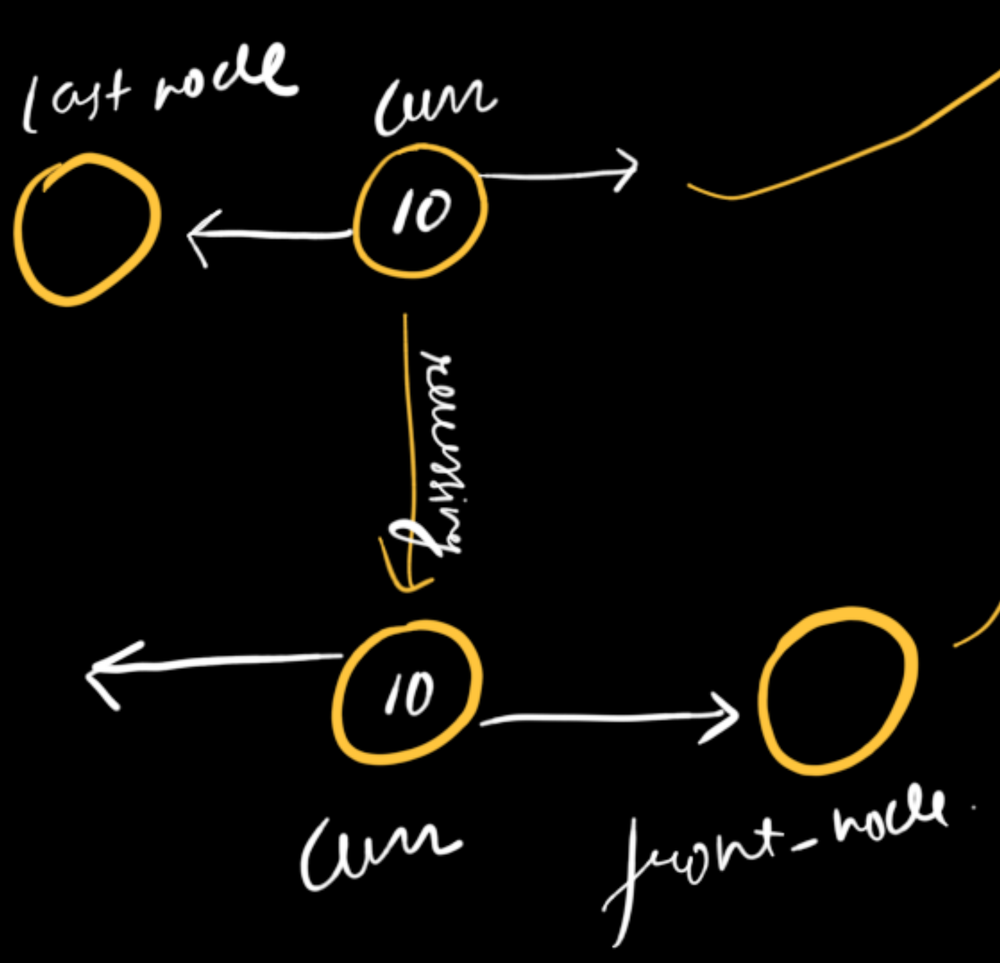
This would need two variable to perform the operation.

↓

But for swapping we need one.




```
curr.next = last-node;
curr.prev = front-node;
```



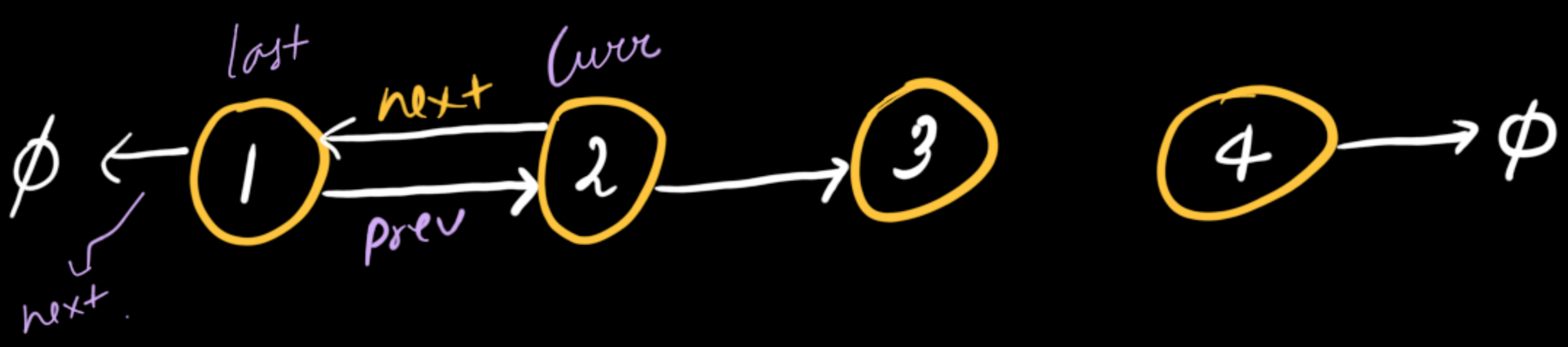

```
last-node = curr.prev;
curr.prev = curr.next;
curr.next = last-node;
```

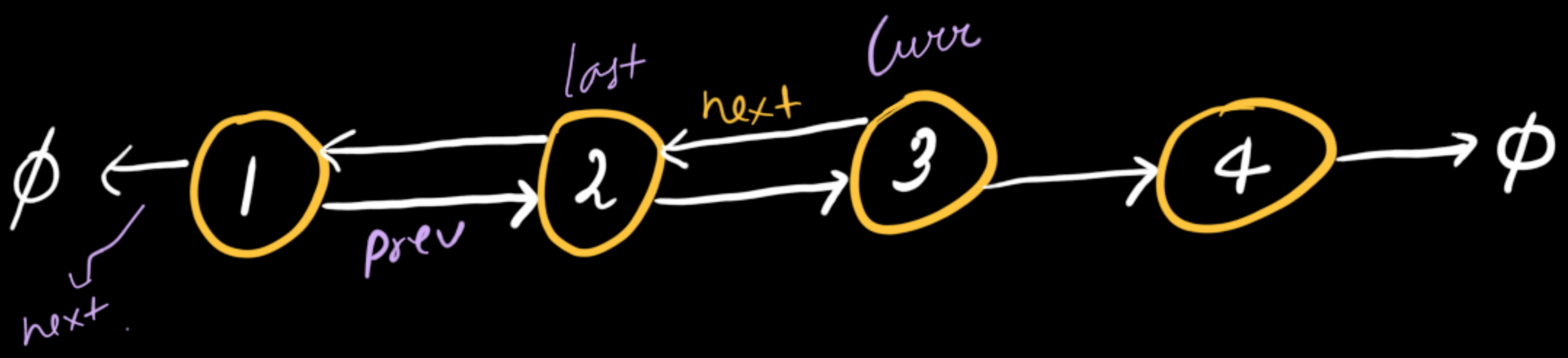
↓

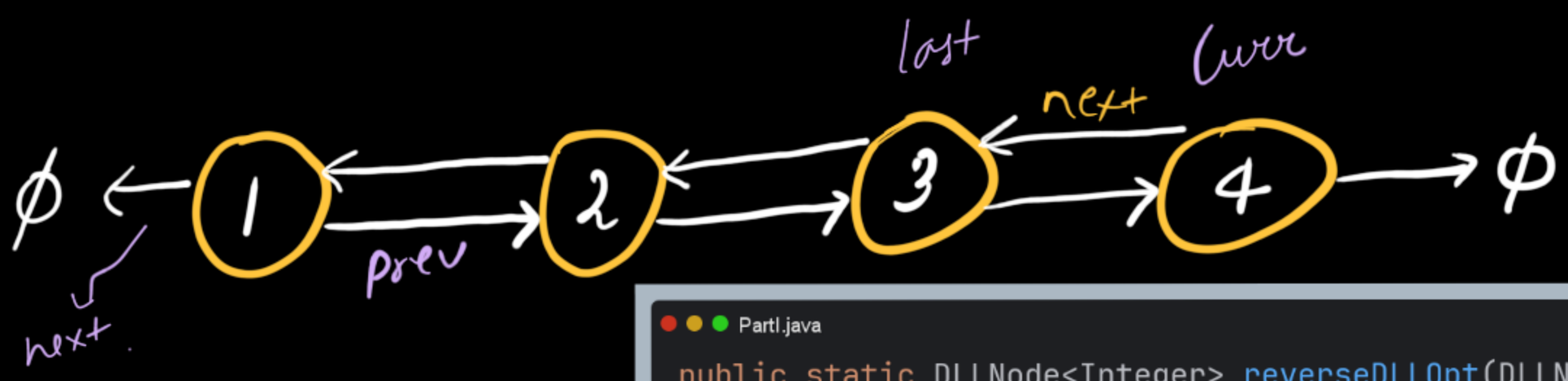
Similar to swapping we are taking a temp variable last-node.

```
last_node = cur.prev;
cur.prev = cur.next;
cur.next = last_node;
cur = cur.prev;
```

to move forward since next become previous & prev become next







new_head = last.prev;

TC: O(N)

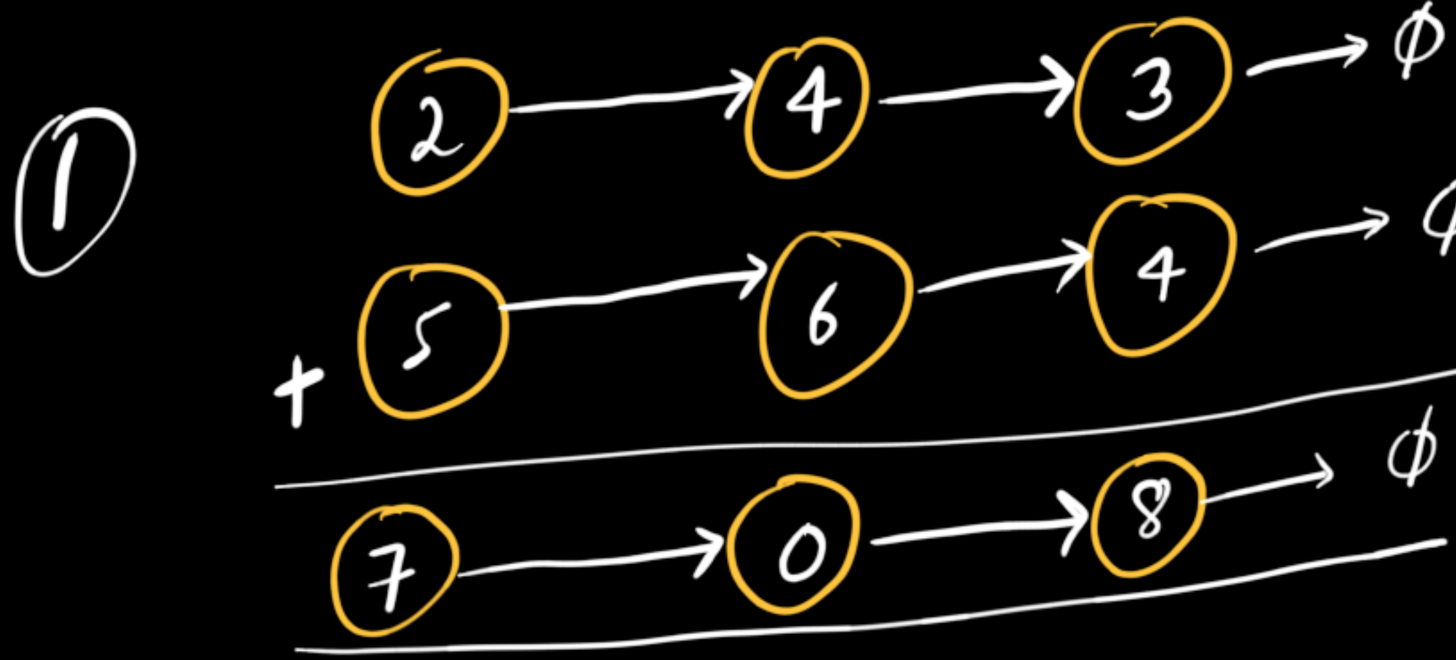
SC: O(1)

Part1.java

```java
public static DLLNode<Integer> reverseDLLOpt(DLLNode<Integer> head) {
    if (head == null || head.next == null) {
        return head;
    }
    DLLNode<Integer> last = null;
    DLLNode<Integer> curr = head;

    while (curr != null){
      last = curr.prev;
      curr.prev = curr.next;
      curr.next = last;

      curr = curr.prev;

    }
    head = last.prev;

    return head;
}
```

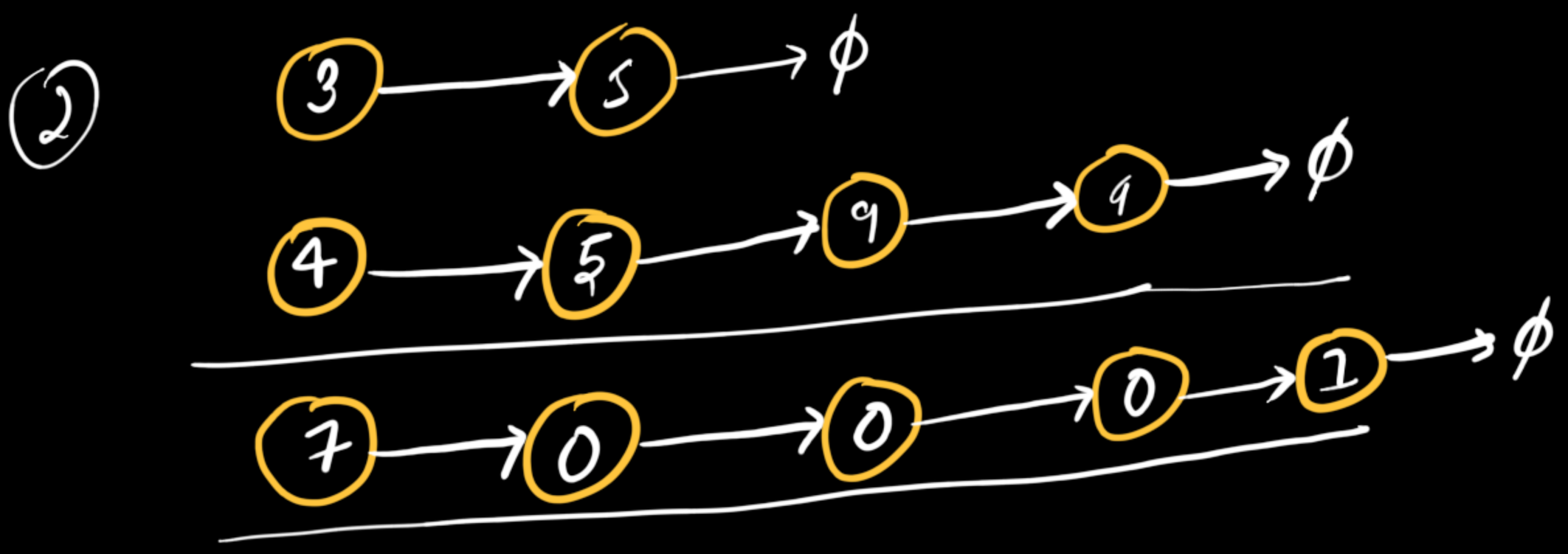
→ Add two numbers

↳ • Given two linked list l1 & l2, return sum of these two list in a new linked list.
• The digits are stored in reverse order.




```
  1
  342
+ 465
-----
  807
-----
  1
```

return → 7 → 0 → 8



→ Dummy node approach

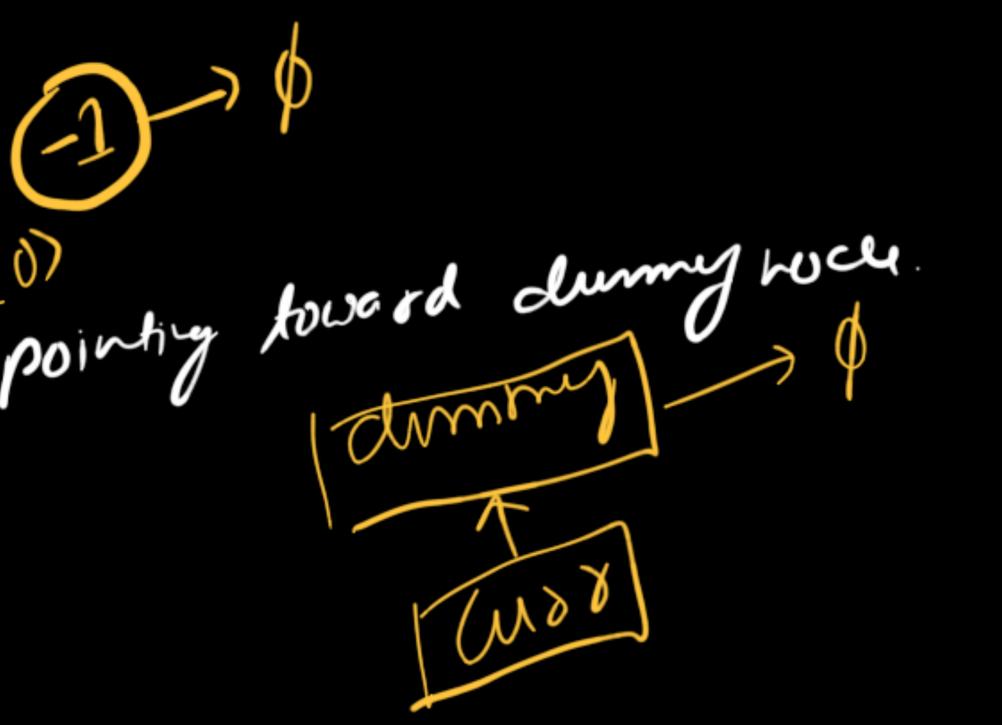
↳ Common linked list pattern used to simplify operations like insertion, merging, addition etc.

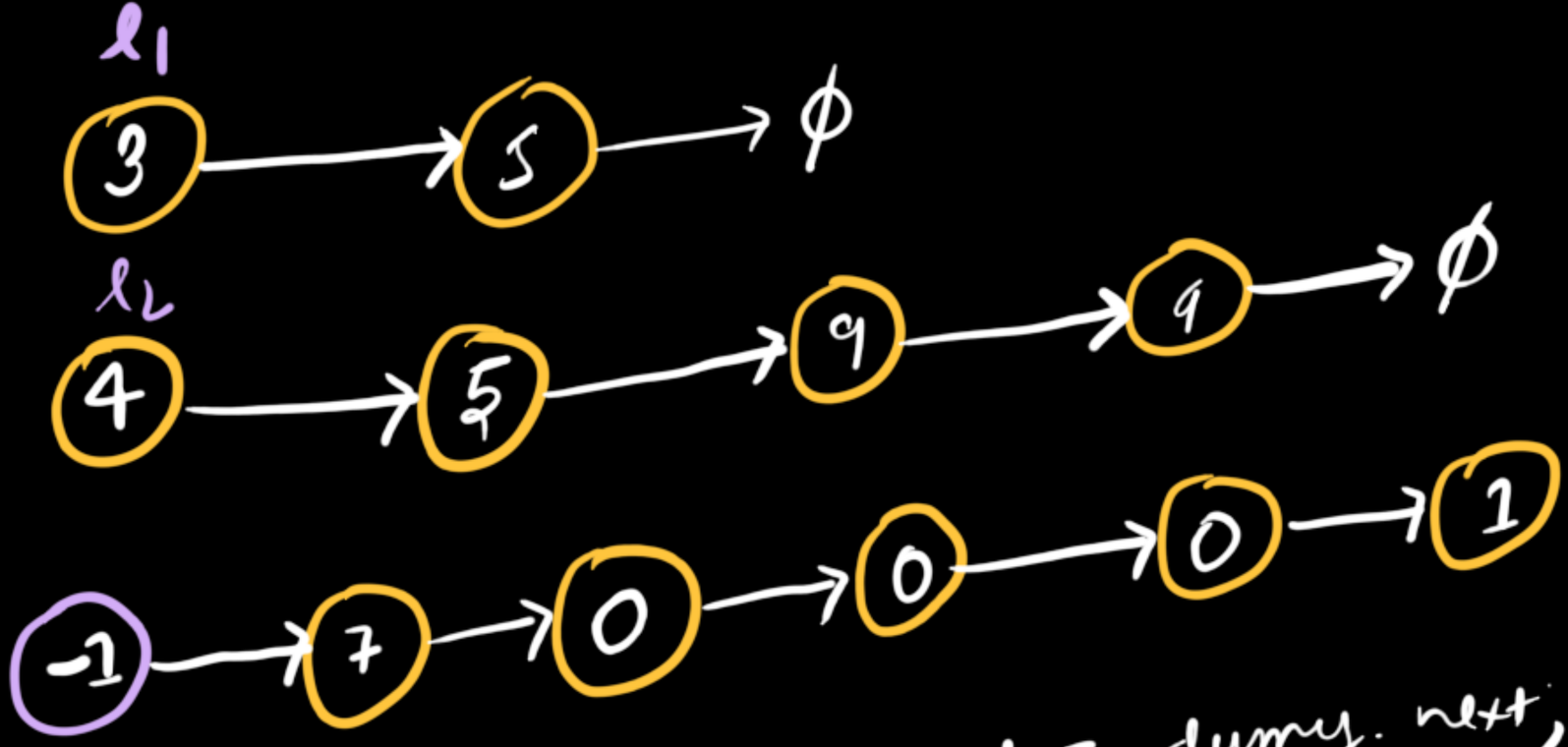
↳ helps by avoiding special handling for the head of the list.

↳ Benefits:

- No edge-case for first-node
- Avoid checking (head == null)
- makes code more readable.

• Initially:
  - A dummy node
  - carry value (0)
  - current node pointing toward dummy node.





carry = 0

1 + 1

dummy node

↳ new head = dummy.next;

carry = 0; temp = dummy

while (l1 != null || l2 != null) {

int sum = carry;

if (l1 != null) {

sum += l1.value;

l1 = l1.next;

}

if (l2 != null) {

sum += l2.value;

l2 = l2.next;

}

temp.next = new Node(sum % 10);

carry = sum / 10

temp = temp.next;

}

if (carry != 0){

temp.next = new Node(carry);

}

return dummy.next;

```java
public static LLNode<Integer> addTwoNumbers(LLNode<Integer> l1, LLNode<Integer> l2) {
    if (l1 == null && l2 == null) return null;
    if (l1 == null) return l2;
    if (l2 == null) return l1;

    LLNode<Integer> dummy = new LLNode<>( value: -1);
    LLNode<Integer> temp = dummy;
    int carry = 0;

    while (l1 != null || l2 != null) {
        int sum = carry;

        if (l1 != null) {
            sum += l1.value;
            l1 = l1.next;
        }

        if (l2 != null) {
            sum += l2.value;
            l2 = l2.next;
        }

        temp.next = new LLNode<>( value: sum % 10);
        carry = sum / 10;
        temp = temp.next;
    }

    if (carry != 0) {
        temp.next = new LLNode<>(carry);
    }

    return dummy.next;
}
```

TC: O(N)

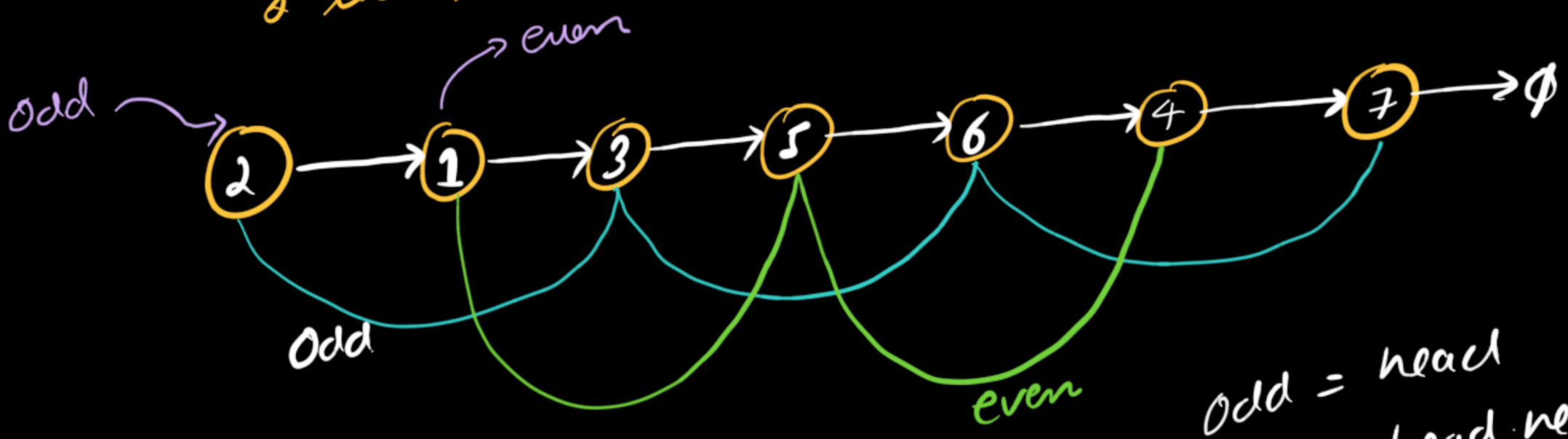
SC: O(N)

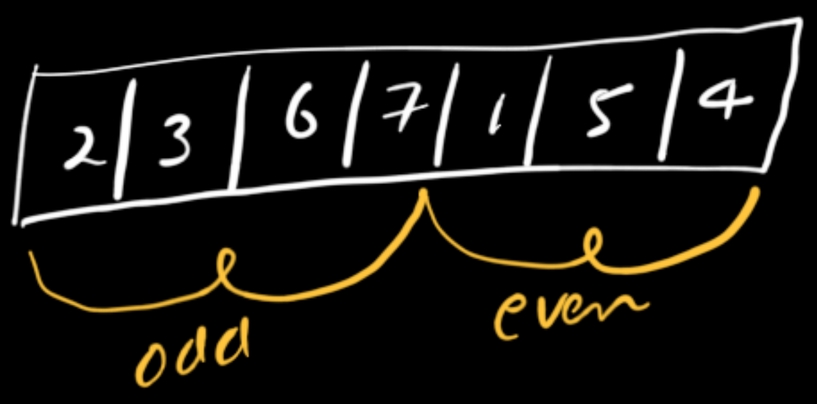
↳ but only for output

if (carry remain even after finishing both list, just add a new Node with value carry.

→ Odd - Even Linked List

↳ Given the head of the linked list, group all the node with odd indices together followed by even indices, & then return the reordered list.





→ list

odd = head
even = head.next
odd.next = odd.next.next;
even.next = even.next.next;

→ Approach - 1

↳ take a list & traverse on odd indices add it to the list & then traverse at even indices & then add it to the list

- After building list, iterate over it & assign the value.

↳ For odd length linked list, (odd.next != null) condition will hit & it will miss that value, so we make another check if (odd != null) add it to the list

↓

Same goes for even part

TC: O(2N)
SC: O(N)

PartIII.java

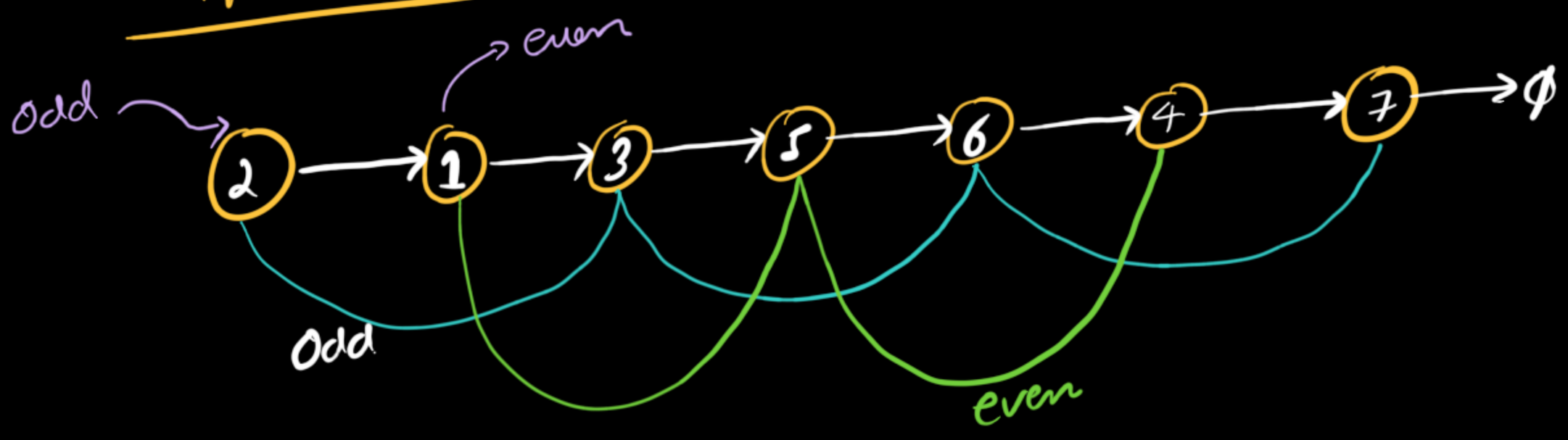
```java
public static LLNode<Integer> oddEvenList(LLNode<Integer> head) {
    if (head == null || head.next == null) {
        return head;
    }

    List<Integer> list = new ArrayList<>();

    LLNode<Integer> odd = head;
    LLNode<Integer> even = head.next;

    while (odd != null && odd.next != null){
        list.add(odd.value);
        odd = odd.next.next;
    }
    if (odd != null) list.add(odd.value);

    while (even != null && even.next != null){
        list.add(even.value);
        even = even.next.next;
    }
    if (even != null) list.add(even.value);

    LLNode<Integer> temp = head;

    int index = 0;
    while (temp != null){
        temp.value = list.get(index);
        index++;
        temp = temp.next;
    }

    return head;
}
```

O(N/2)

O(N/2)

O(N)

→ Approach - 2

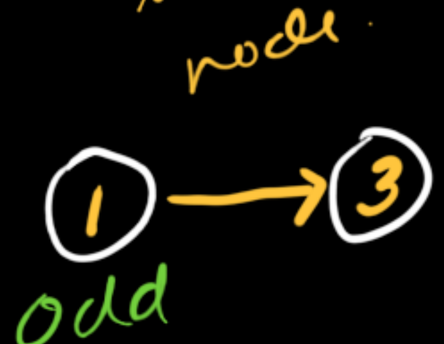


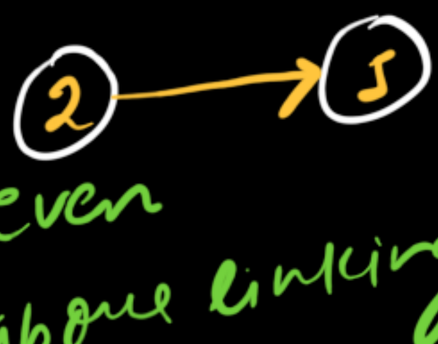
Links:

odd.next = odd.next.next; } connecting links
even.next = even.next.next;

Now after connecting odd & even link to the next odd & even node.

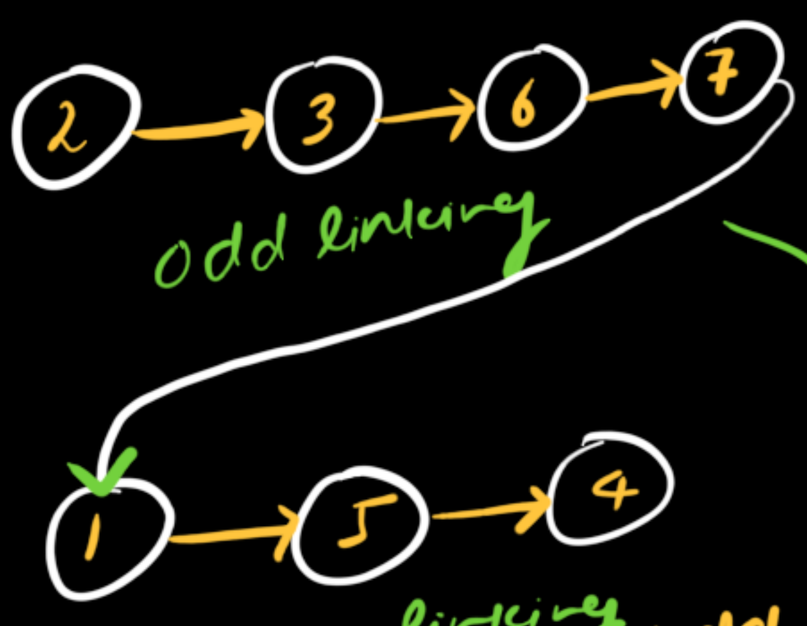
↳ To keep on performing these operation we need to move to next odd & even node.





↳ After performing above linking operation, the new odd & even :-

odd = odd.next;
even = even.next;



To connect ~~the~~ odd - Even part we need to keep reference to first even node & then :-

odd.next = first_even_node;

PartIII.java

```java
public static LLNode<Integer> oddEvenListOpt(LLNode<Integer> head) {
    if (head == null || head.next == null) {
        return head;
    }

    LLNode<Integer> odd = head;
    LLNode<Integer> even = head.next;
    LLNode<Integer> evenHead = head.next;

    while (even != null && even.next != null){
        odd.next = odd.next.next;
        even.next = even.next.next;

        odd = odd.next;
        even = even.next;
    }

    odd.next = evenHead;

    return head;
}
```

TC: O(N)

SC: O(1)

→ Sort 0's, 1's & 2's list

↳ Sort the linked list containing 0's, 1's & 2's.

→ Approach - 1

① Count the frequency of 0's, 1's & 2's

② Iterate over the linked list & overwrite the value.

① Overwrite 0 until count0 is != 0.

② Overwrite 1 until count1 is != 0

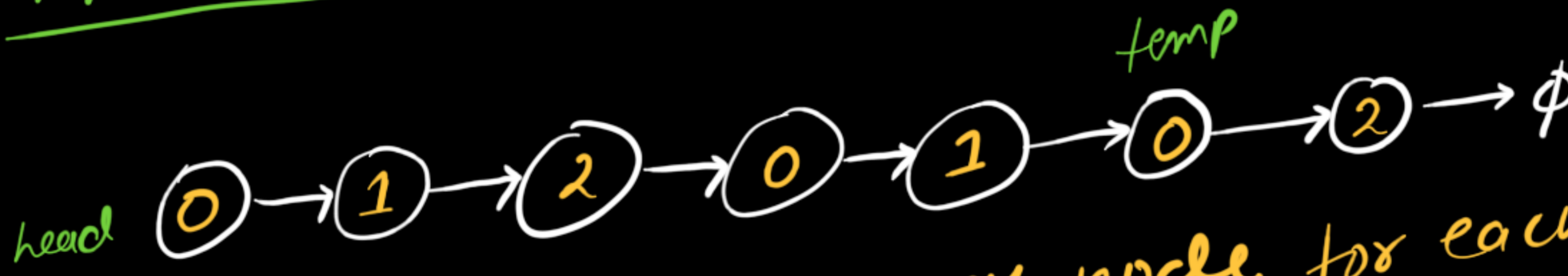
③ Overwrite 2 until count2 is != 0

TC: O(2N)
SC: O(1)

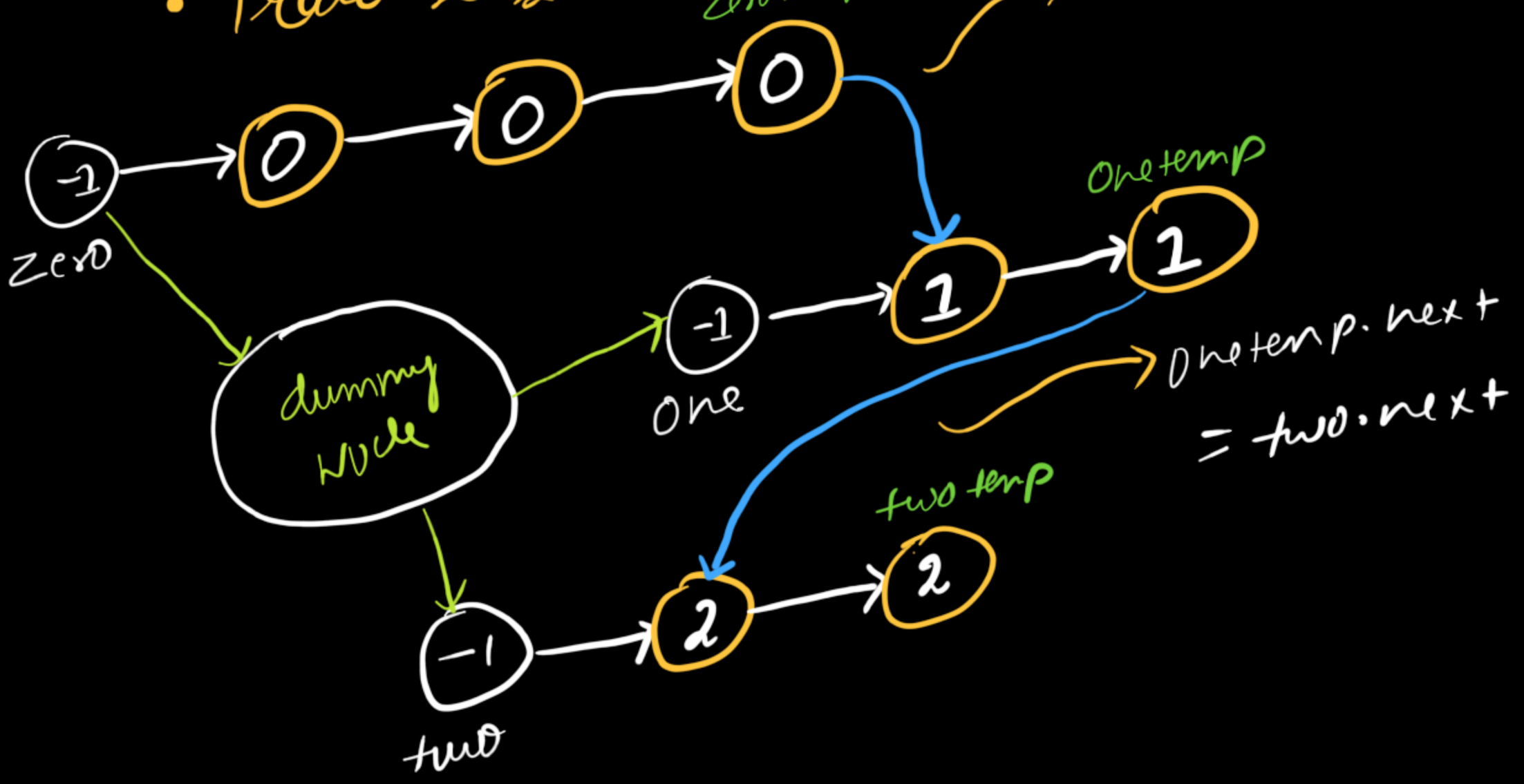
PartIV.java

```java
public static LLNode<Integer> sortList(LLNode<Integer> head) {
    int count0 = 0;
    int coun1 = 0;
    int count2 = 0;

    LLNode<Integer> temp = head;

    while (temp != null){
        if (temp.value == 0) count0++;
        else if (temp.value == 1) coun1++;
        else  count2++;

        temp = temp.next;
    }

    temp = head;

    while (temp != null){
        if (count0 != 0) {
            temp.value = 0;
            count0--;
        } else if (coun1 != 0) {
            temp.value = 1;
            coun1--;
        }else {
            temp.value = 2;
            count2--;
        }
        temp = temp.next;
    }
    return head;
}
```

→ Approach-2



- Create three dummy node for each value.
- Traverse & connect to each node (0,1,2)



Edge-Case:

① what if there are only 0's & 2's?

↳ check if ones.next != null
↳ if null, point 0's to 2's else point 0's to one's.

i.e if(ones.next != null){
tempZero.next = ones.next;
} else {
tempZero.next = twos.next;
}

② last of temptwo will always point to null.
↳ temptwo.next = null;

TC: O(N)
SC: O(1)

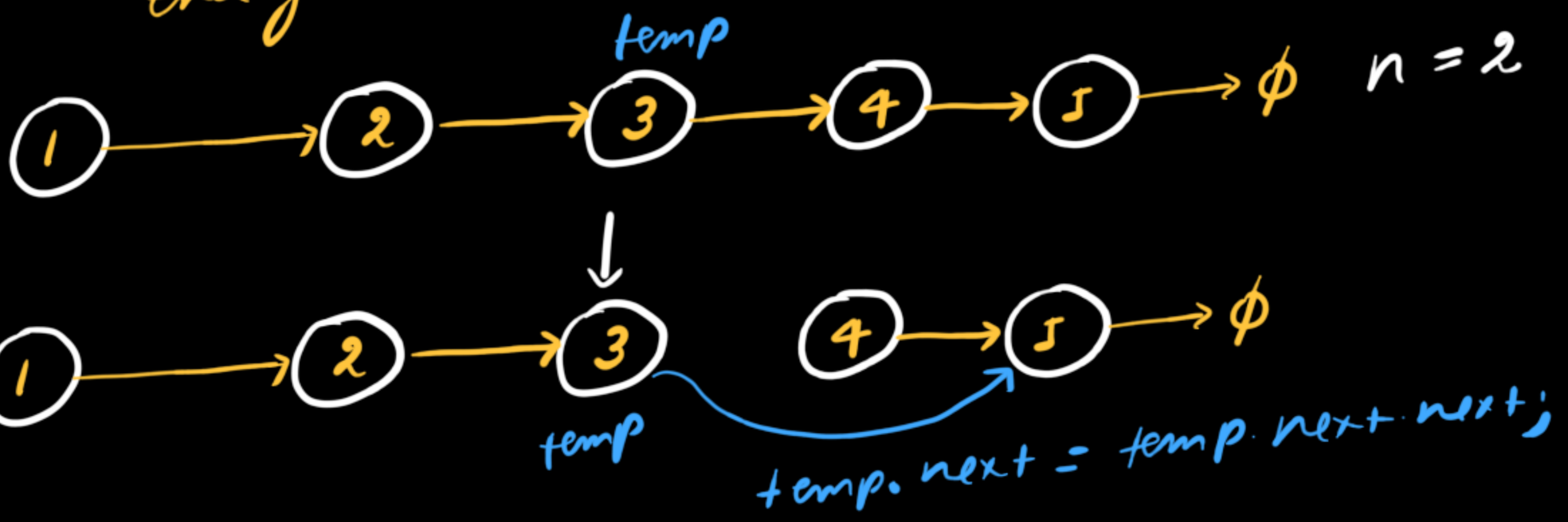
PartIV.java

```java
public static LLNode<Integer> sortListOpt(LLNode<Integer> head) {
    if (head == null || head.next == null) return head;

                                    ↳ dummy nodes
    LLNode<Integer> zeroes = new LLNode<>( value: -1);
    LLNode<Integer> ones = new LLNode<>( value: -1);
    LLNode<Integer> twos = new LLNode<>( value: -1);

    LLNode<Integer> tempZero = zeroes;
    LLNode<Integer> tempone = ones;
    LLNode<Integer> temptwo = twos;

    LLNode<Integer> temp = head;

    while (temp != null) {
        if (temp.value == 0) {
            tempZero.next = temp;
            tempZero = tempZero.next;
        } else if (temp.value == 1) {
            tempone.next = temp;
            tempone = tempone.next;
        } else {
            temptwo.next = temp;
            temptwo = temptwo.next;
        }
        temp = temp.next;
    }

    // Break the last node
    temptwo.next = null;

    // Connect 0s → 1s → 2s
    tempZero.next = ones.next != null ? ones.next : twos.next;
    tempone.next = twos.next;

    return zeroes.next;
}
```

↳ pointers

→ Remove nth node from list

↳ Given the head of a linked list, remove nth node from the end of the list & return its head.



→ Approach - 1

① Find the length of the list & iterate till (length - n - 1)

② perform temp.next = temp.next.next;

③ Edge case:
↳ what if length of the ll == n ?
↳ return head.next;

TC: O(len) + O(len - n)
SC: O(1)

PartV.java

```java
public LLNode<Integer> removeNthFromEnd(LLNode<Integer> head, int n) {
    int length = len(head);

    if (n == length) {
        return head.next;
    }

    LLNode<Integer> temp = head;
    for (int i = 0; i < length - n - 1; i++) {
        temp = temp.next;
    }

    temp.next = temp.next.next;
    return head;
}

public static int len(LLNode<Integer> head) {  1 usage  new *
    int len = 0;
    LLNode<Integer> temp = head;
    while (temp != null) {
        len++;
        temp = temp.next;
    }
    return len;
}
```

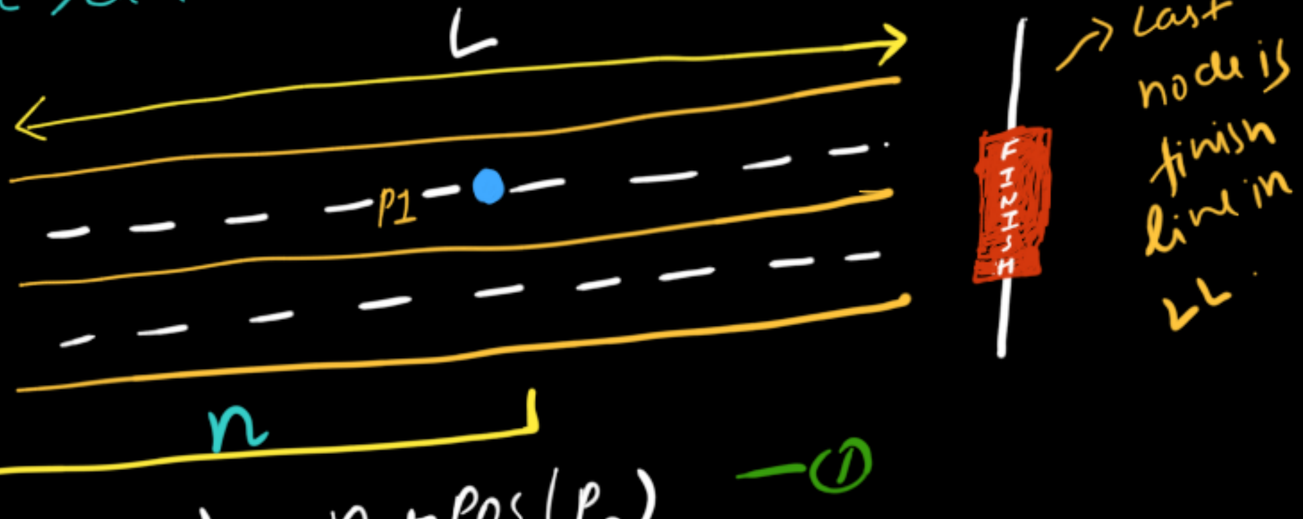
→ Approach-2

↳ How can we reach just before nth node ?

↳ Imagine there are two racer's P1 & P2.

P1 → cheats & runs n steps before the race starts

P2 → runs when race start.



- The relative gap between P1 & P2 is n initially.

↓

- Now both racer run at equal pace i.e 1 step

↳ Let P1 takes x steps, so P2 also takes x steps.

↳ So, $pos(P_1) + x = L$ — (2)

$pos(P_2) = x$ — (3)

$pos(P_1) + pos(P_2) = L$ — (2) & (3)

$pos(P_2) = L - pos(P_1)$ → (1)

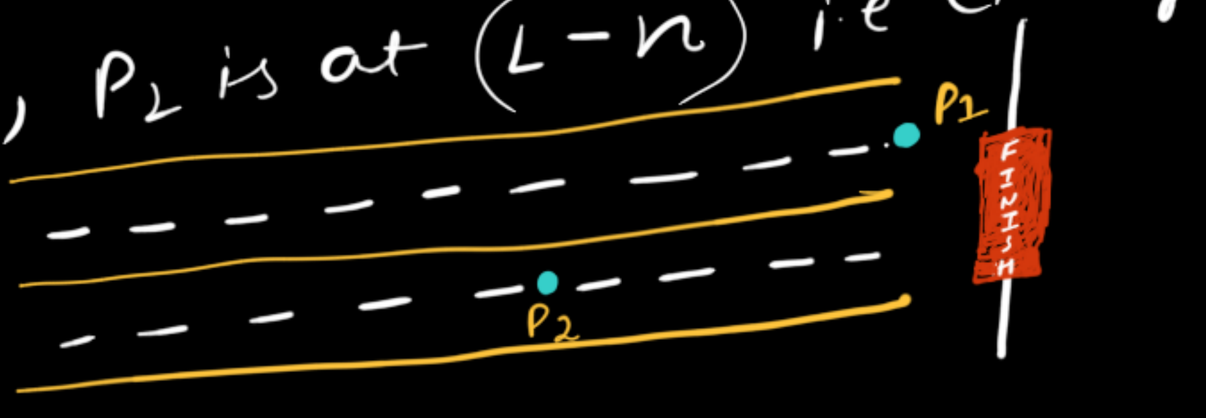
$pos(P_2) = L - n + pos(P_1)$ { $pos(P_1) = 0$, initially }

$\boxed{pos(P_2) = L - n}$

$pos(P_1) = n + pos(P_2)$ — (1)

↳ Initially it is 0

- when P_1 finishes the race, P_2 is at $(L-n)$ i.e exactly before the nth node.



PartV.java

```java
public static LLNode<Integer> removeNthFromEndOpt(LLNode<Integer> head, int n) {
    if (head == null) return null;
    LLNode<Integer> fast = head;
    LLNode<Integer> slow = head;

    for (int i = 0; i < n; i++) {
        fast = fast.next;
    }

    // when length == n i.e FAST becomes null, and we are req to delete first node
    if (fast == null) return head.next;

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

    slow.next = slow.next.next;
    return head;
}
```

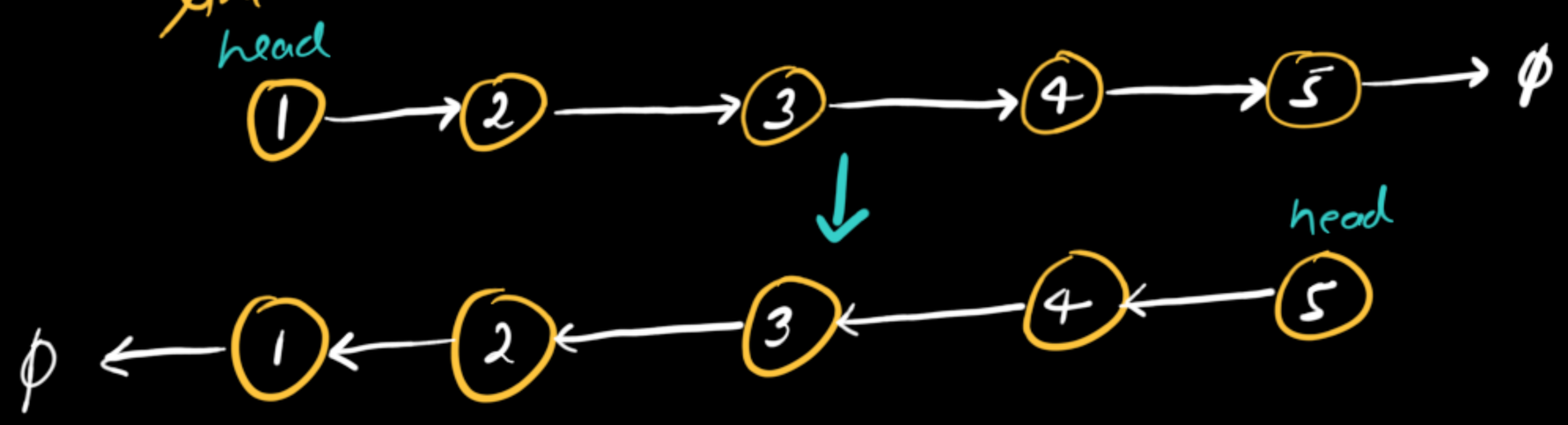
→ move fast pointer n steps ahead

TC : O(N)

SC : O(1)

→ Reverse a Linked List

↳ Given the head of Linked List, reverse & return the head of linked list.



→ Approach - 1

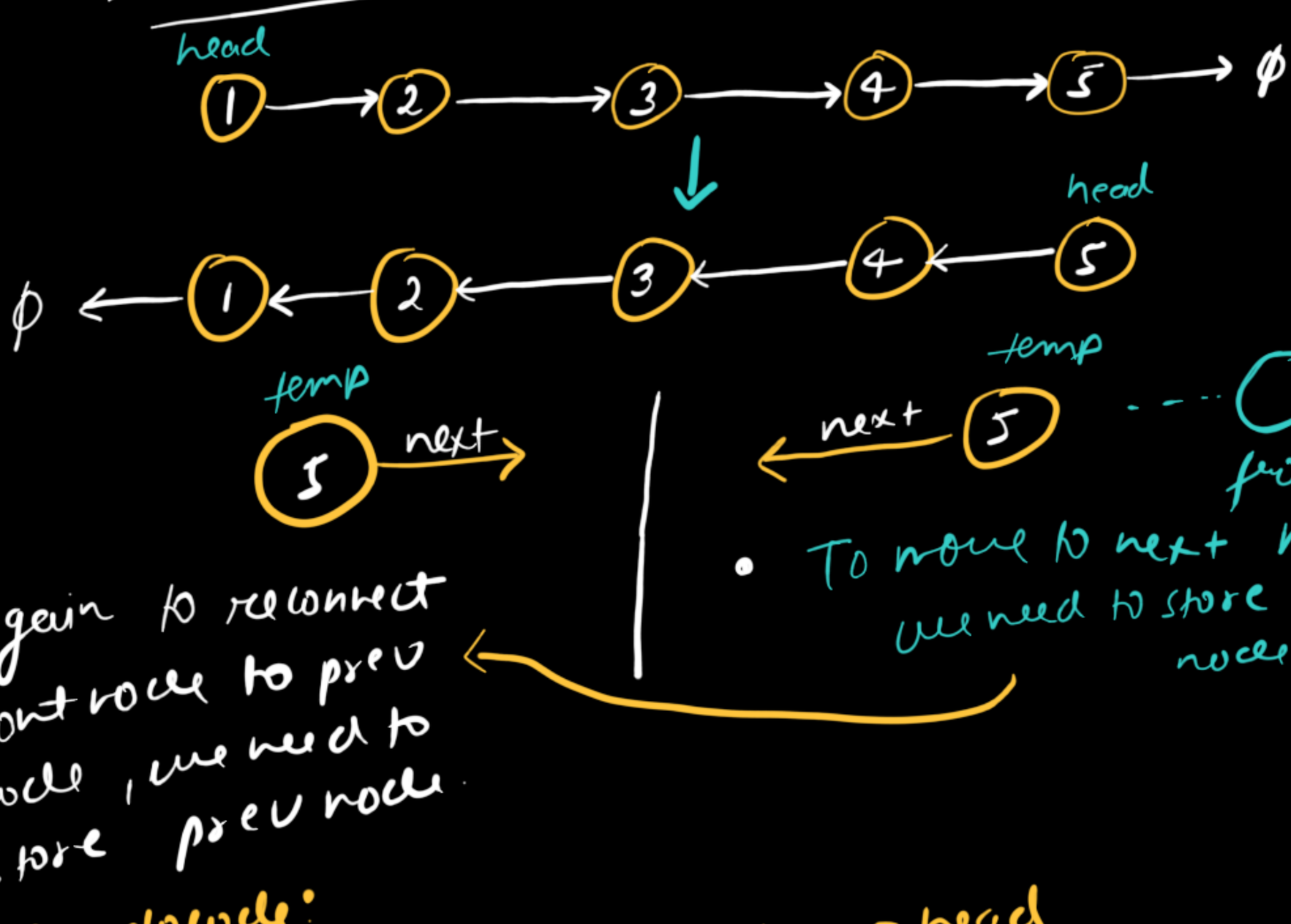
↳ ① Use stack, push element into stack

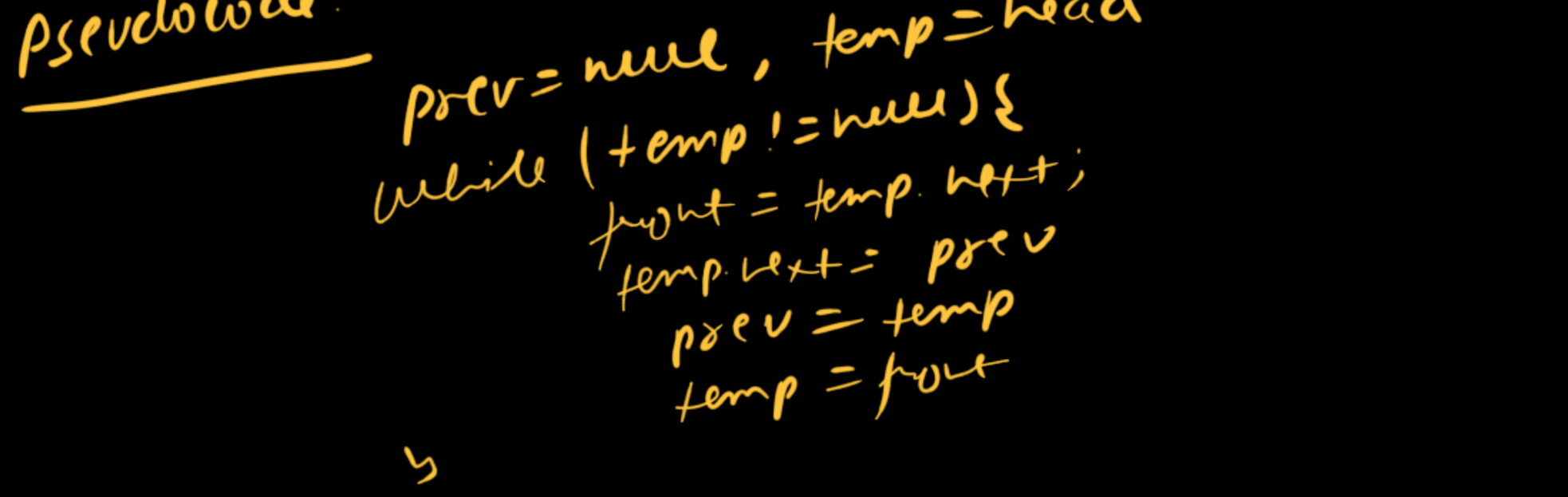
② Again, iterate from start, & overside the value of temp by popping off from the stack

Tc: O(2N) : Sc: O(N)

→ Approach - 2

↳ • Iteratively

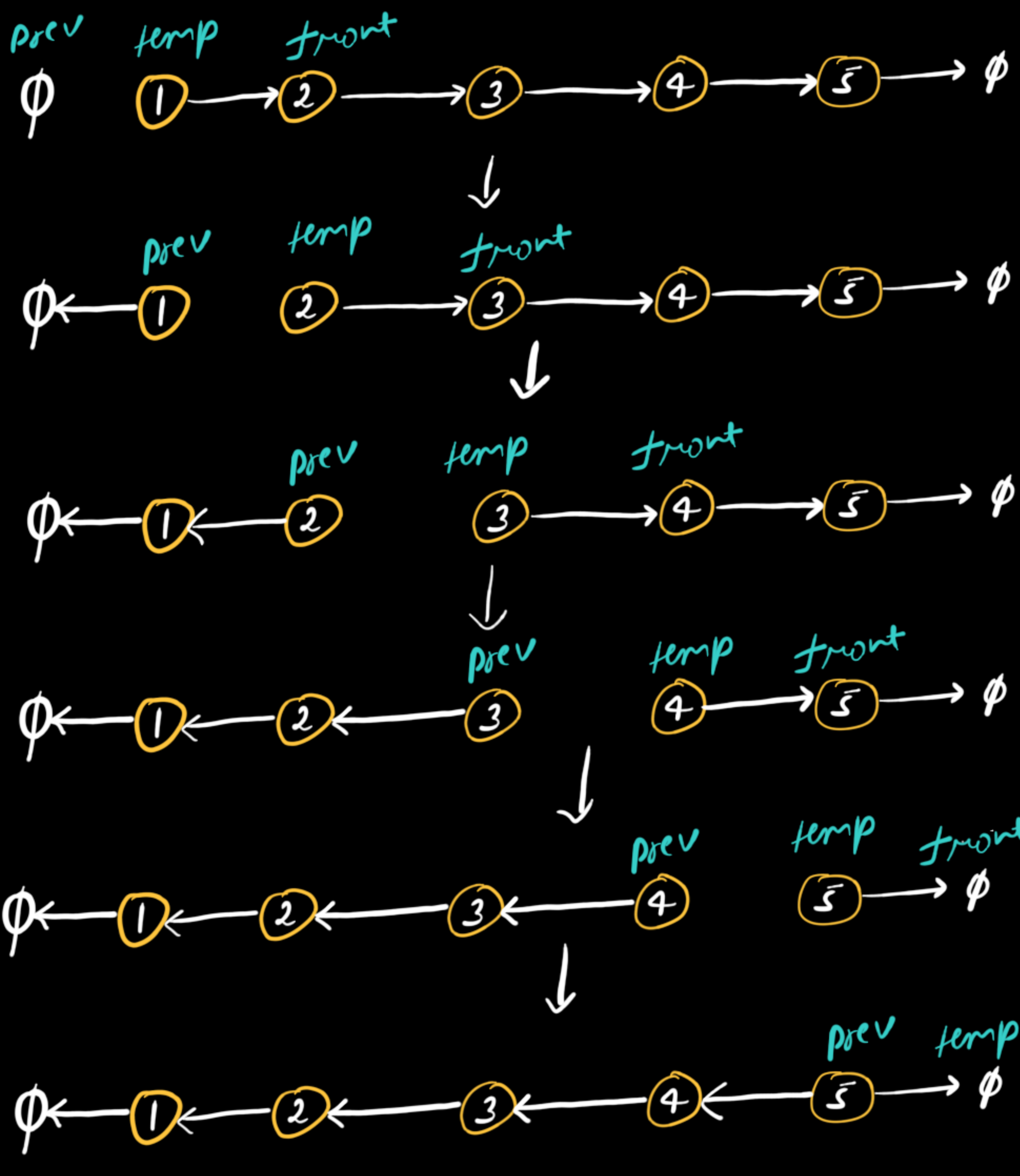




- Again to reconnect front node to prev node, we need to store prev node.
- To move to next node we need to store front node.

Pseudocode:

```
prev = null, temp = head
while (temp != null) {
    front = temp.next;
    temp.next = prev
    prev = temp
    temp = front
}
```



- when temp reaches null our linked list is reversed & new head is prev.

PartVI.java

```java
public static LLNode<Integer> reverseListOpt(LLNode<Integer> head) {

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

    LLNode<Integer> temp = head;
    LLNode<Integer> prev = null;

    while (temp != null){
        LLNode<Integer> front = temp.next;
        temp.next = prev;
        prev = temp;
        temp = front;
    }

    return prev;
}
```

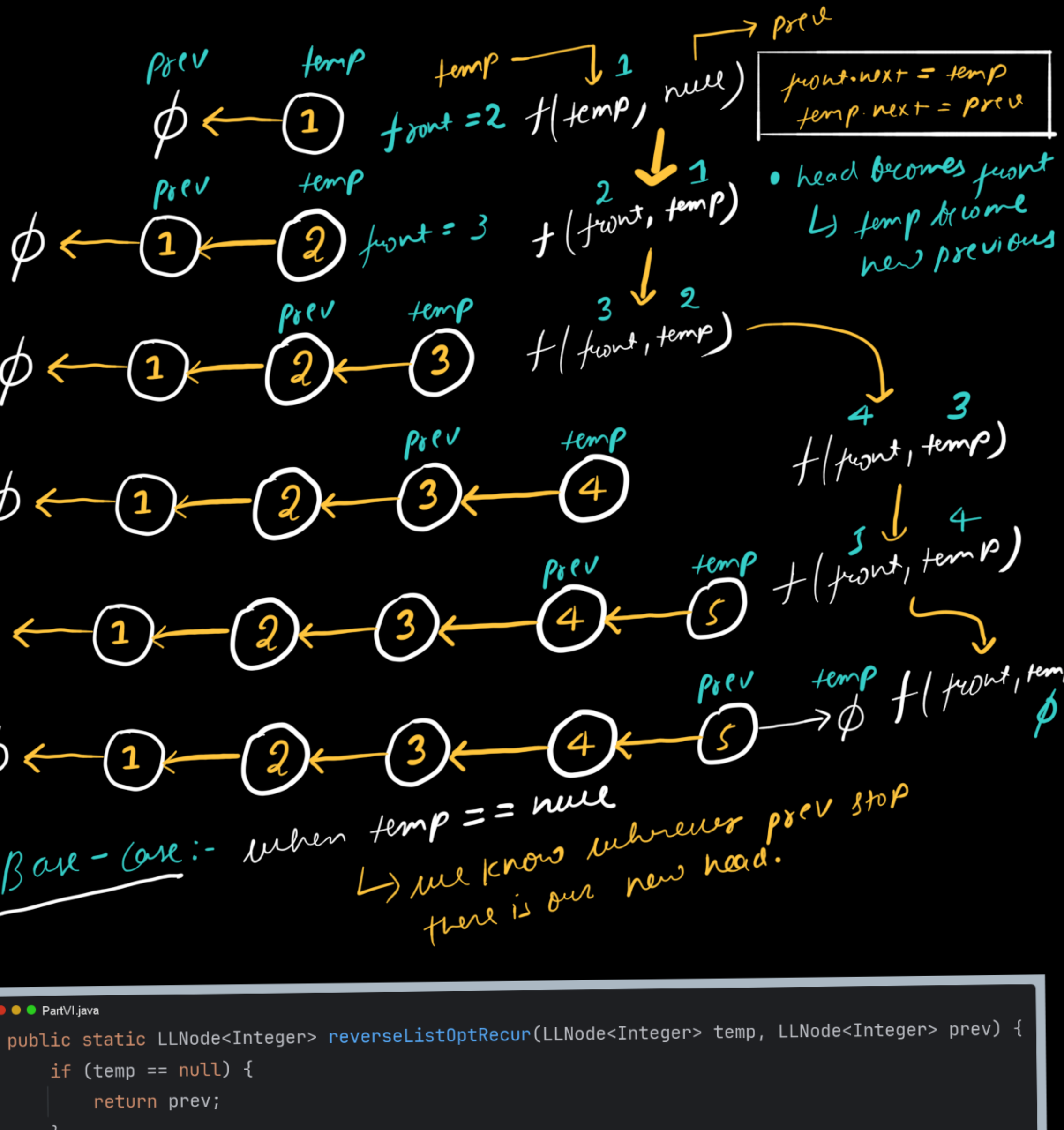
TC : O(N)

SC : O(1)

→ Recursive

↳ only two variables are changing externally

(1) temp (2) prev



Base-case :- when temp == null

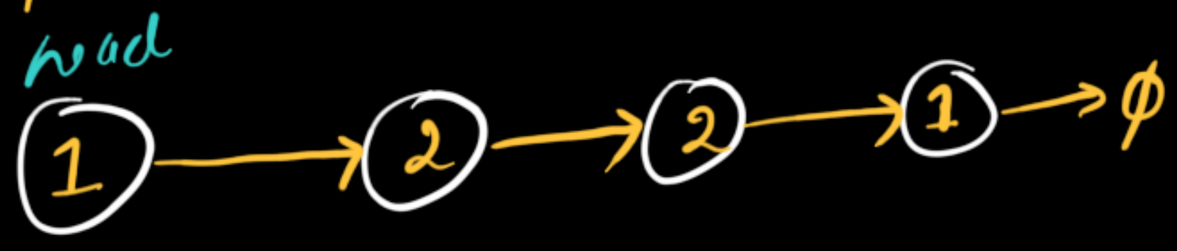
↳ we know whenever prev stop there is our new head.

```java
PartVI.java
public static LLNode<Integer> reverseListOptRecur(LLNode<Integer> temp, LLNode<Integer> prev) {
    if (temp == null) {
        return prev;
    }

    LLNode<Integer> front = temp.next;
    temp.next = prev;

    return reverseListOptRecur(front, temp);
}
```

TC: O(N)
SC: O(N) → stack space

→ Check if linked list is palindrome

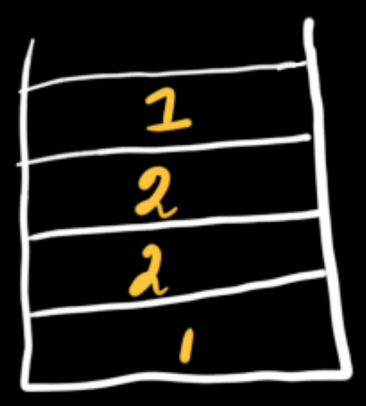
↳ given the head of the linked list, return true if LL is palindrome.



→ Approach - 1

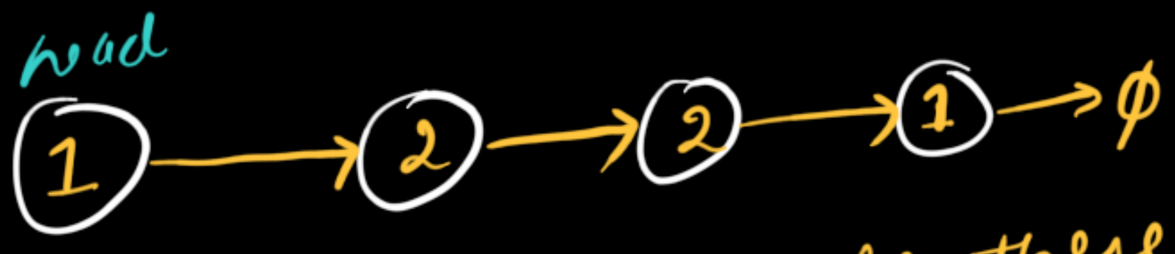
(1) iterate our linked list & push each element in stack

(2) now again iterate from head, & compare the value of each node by popping off the stack.

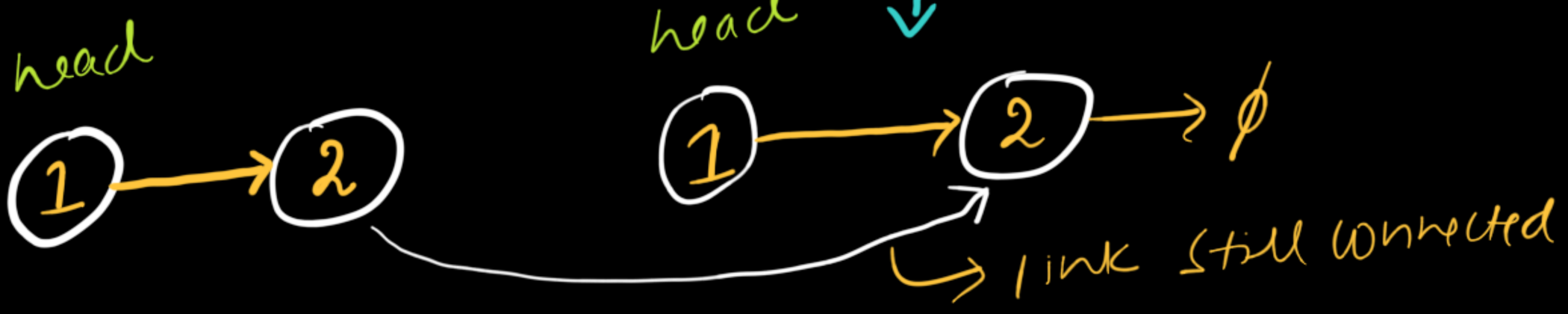


TC: O(2N) SC: O(N)

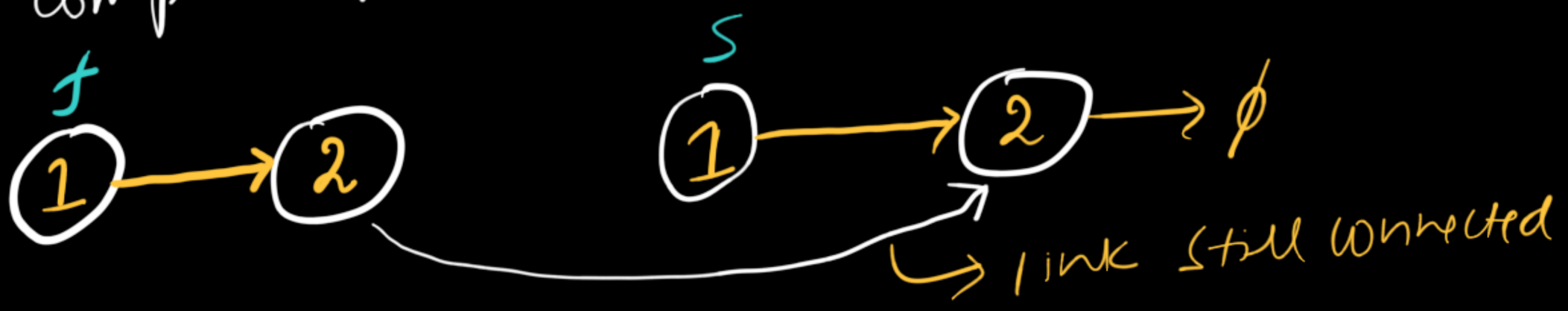
→ Approach - 2



- if we somehow divide these linked list into two list, & then reverse the 2nd divided list, then we can compare each node, if != we return false else true.



- Now if we put two pointer f & s, compare & keep on moving




```
while ( s != null ) {
    if ( f.value != s.value ) {
        return false;
    }
    f = f.next;
    s = s.next;
}
return true.
```

Always restore original list.

- do not distort the original linked list.

TC: O(2N)

SC: O(1)

```java
public static boolean isPalindromeOpt(LLNode<Integer> head) {  1 usage
    LLNode<Integer> mid = getMid(head);
    LLNode<Integer> r = reverseListOpt(mid);

    LLNode<Integer> f = head;
    LLNode<Integer> s = r;

    while (s != null) {
        if (!f.value.equals(s.value)) {
            reverseListOpt(r);  // restore original list
            return false;
        }

        f = f.next;
        s = s.next;
    }
    reverseListOpt(r);  // restore original list
    return true;
}

@Contract(value = "null -> null", pure = true)
private static LLNode<Integer> getMid(LLNode<Integer> head) {  1 usage
    if (head == null || head.next == null) return head;

    LLNode<Integer> slow = head;
    LLNode<Integer> fast = head;

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

    return slow;
}

@Contract("null -> null")
private static LLNode<Integer> reverseListOpt(LLNode<Integer> head) {
    if (head == null || head.next == null) return head;

    LLNode<Integer> temp = head;
    LLNode<Integer> prev = null;

    while (temp != null) {
        LLNode<Integer> front = temp.next;
        temp.next = prev;
        prev = temp;
        temp = front;
    }

    return prev;
}
```

→ Add 1 to the linked list

↳ Given +ve integer represented in the form of singly Linked List.

↳ Add one to the number i.e increment the given number by one.

Ex-1: head → 1 → 5 → 2 → ϕ + 1 → head → 1 → 5 → 3 → ϕ

Ex-2: head → 9 → 9 → 9 → ϕ + 1 → head → 1 → 0 → 0 → 0 → ϕ

→ Approach-1

(1) Reverse the LL → 2 → 5 → 1 → ϕ (head)

(2) Add 1 to it → 3 → 5 → 1 → ϕ (head)

(3) Again reverse the LL → 1 → 5 → 3 → ϕ (head)

(4) what it?

head → 9 → 9 → 9 → ϕ + 1

↳ In this case we need to add one more node to first.

↳ if (carry == 0) break;

↳ we don't need to perform further operation

↳ (1) Reverse the LL

9 → 9 → 9 → ϕ

(2) Add 1 to it.

carry 1 | 1 | 1

0 → 0 → 0 → ϕ (temp)

(3) carry = 1 & temp is at null

↳ (1) Reverse the LL

0 → 0 → 0 → ϕ

(2) Add 1 carry node at head.

head → 1 → 0 → 0 → 0 → ϕ

i.e

```
New_node = new Node(carry)
new_node.next = head;
head = new_node.
```

PartVIII.java

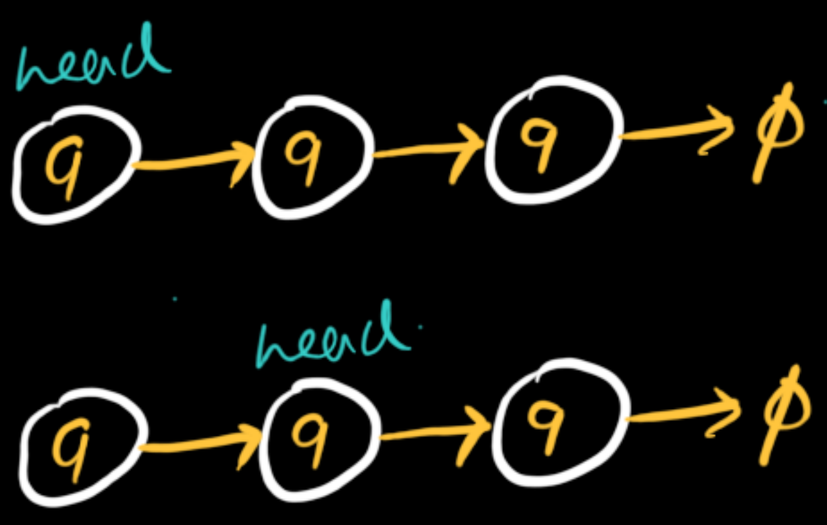
```java
public static LLNode<Integer> addOne(LLNode<Integer> head) {
    head = reverseListOpt(head);
    LLNode<Integer> temp = head;
    int carry = 1;

    while (temp != null) {
        int sum = temp.value + carry;
        temp.value = sum % 10;
        carry = sum / 10;

        if (carry == 0) break;
        temp = temp.next;
    }

    head = reverseListOpt(head);

    if (carry > 0) {
        LLNode<Integer> c = new LLNode<>(carry);
        c.next = head;
        head = c;
    }

    return head;
}
```

TC: O(3N)

SC: O(1)

→ Approach-2 → can we do it in single pass

↳ use recursion



f(head)

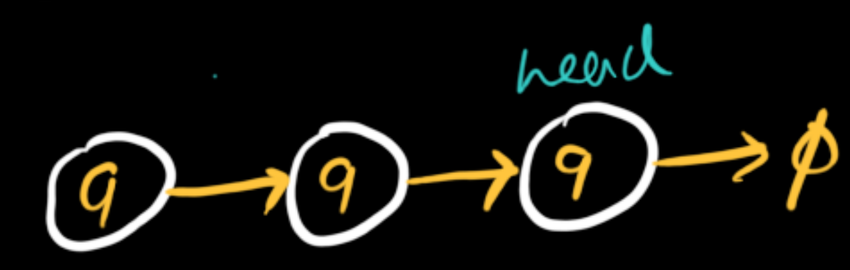
↓

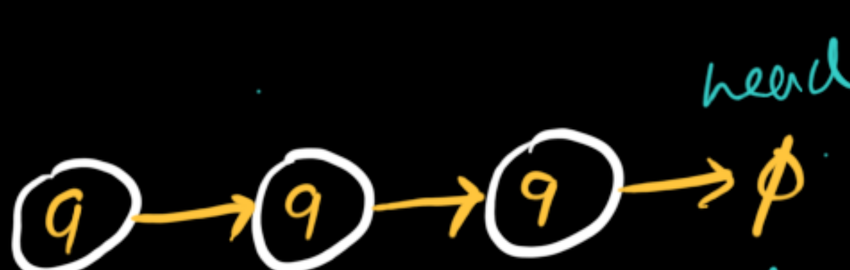
f(head.next)

Base-case:

↳ head == null ?

↳ return 1



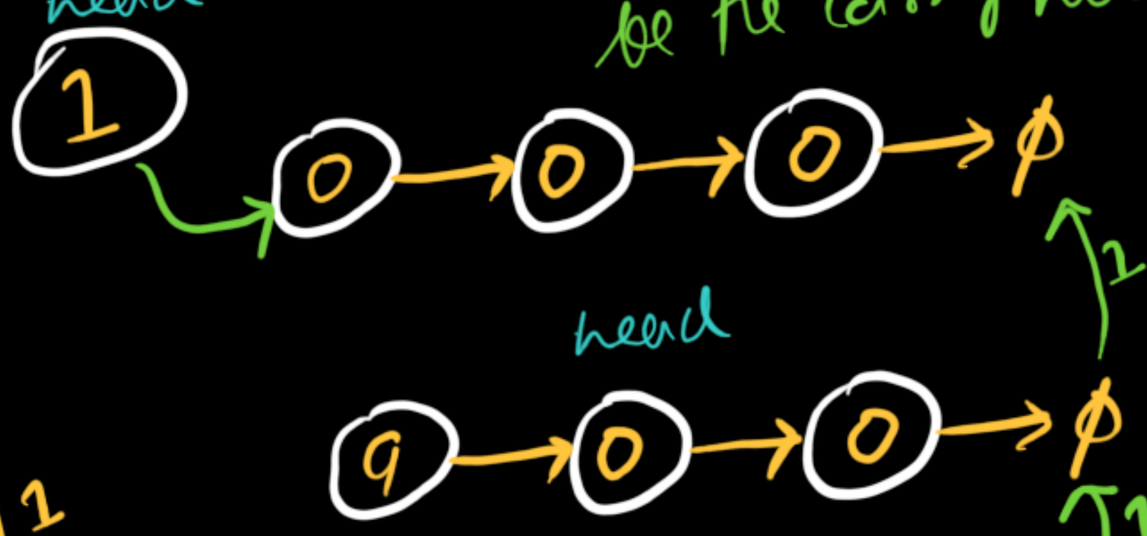


f(head)

↓

f(head.next)

↑1



new head will be the carry node.

f(head)

↓

f(head.next)

↑1

f(head.next)

↓

f(head.next)

↑1

f(head.next)

↓

f(head.next)

↑1

Still carry = 1

if (carry != 0)

add new node at head.

head.value = sum % 10

carry = sum / 10

{sum = head.val + carry}

(2)

We need to add one to the LL.

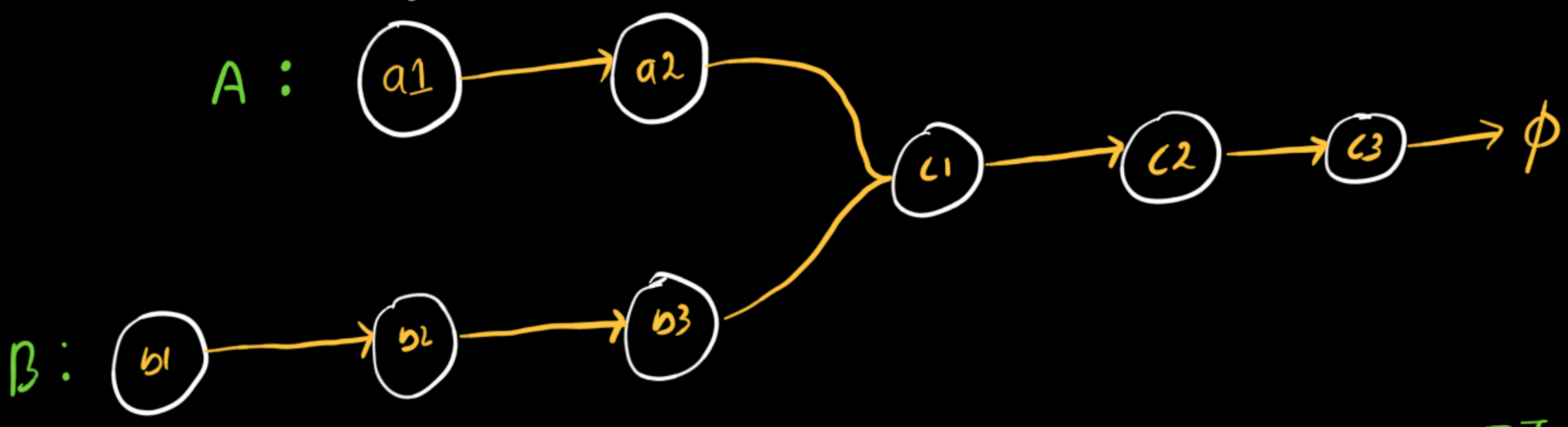
TC: O(N)

SC: O(N)

PartVIII.java

```java
public static LLNode<Integer> add1(LLNode<Integer> head){
    int carry = addOneOpt(head);
    if (carry > 0){
        LLNode<Integer> c = new LLNode<Integer>(carry);
        c.next = head;
        head = c;
    }
    return head;
}

private static int addOneOpt(LLNode<Integer> head) {  2 us
    if (head == null) return 1;

    int carry = addOneOpt(head.next);
    int sum = head.value + carry;

    head.value = sum % 10;
    return sum / 10;
}
```

→ Intersection of two linked List



→ Approach - 1

↳ • we can add one of the Linked List in SET & then Traverse the second list, & the moment we find a node already in set i.e our intersection point.

PartIX.java

```java
public static @Nullable LLNode<Integer> getIntersectionNode(LLNode<Integer> headA, LLNode<Integer> headB) {
    Set<LLNode<Integer>> set = new HashSet<>();
    LLNode<Integer> ll = headA;

    while (ll != null) {
        set.add(ll);
        ll = ll.next;
    }
    LLNode<Integer> temp = headB;

    while (temp != null) {
        if (set.contains(temp)) {
            return temp;
        }
        temp = temp.next;
    }

    return null;
}
```

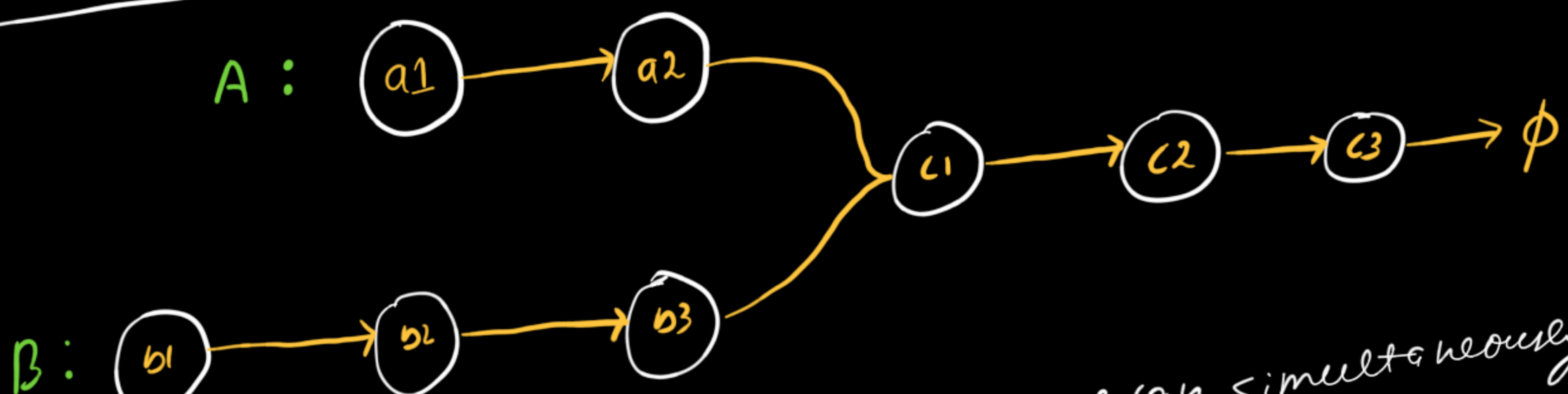
$O(l_1)$ → length of 1st LL

TC: $O(l_1 + l_2)$

SC: $O(l_1)$

$O(l_2)$ ↳ length of second LL

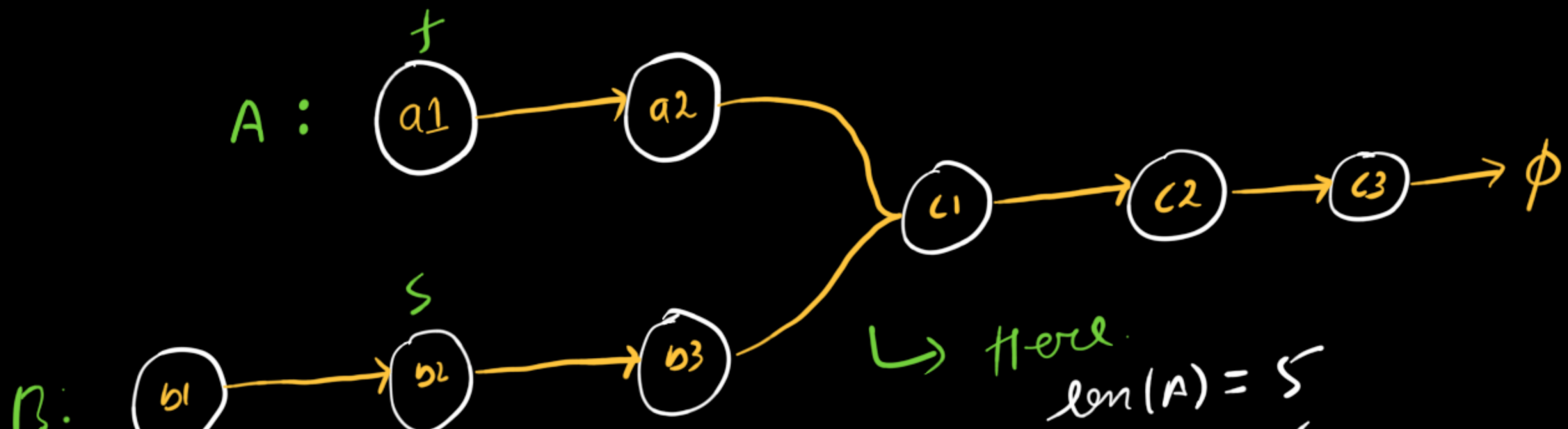
→ Approach - 2



① if length of A == length of B, we can simultaneously traverse & if we find common value i.e our intersection point.

② what if len(A) > len(B) || len(A) < len(B) ?

↳ traverse the length difference in longer part of the LL & then traverse & find the intersection point.



↳ Here:
len(A) = 5
len(B) = 6

- So traverse the [len(B) − len(A)] before hand in the longer linked list i.e B_i & then start traversing from equal length, & whenever node becomes equal we will return that node.

PartIX.java

```java
public static LLNode<Integer> getInter(LLNode<Integer> a, LLNode<Integer> b) {  1 usa
    if (a == null || b == null) return null;
    int lenOfA = getLen(a);
    int lenOfB = getLen(b);

    if (lenOfA > lenOfB) {
        int diff = lenOfA - lenOfB;
        while (diff-- > 0) a = a.next;
    } else {
        int diff = lenOfB - lenOfA;
        while (diff-- > 0) b = b.next;
    }
    return intersection(a, b);
}

@Contract(...)
private static LLNode<Integer> intersection(LLNode<Integer> f, LLNode<Integer> s) {

    while (f != null && s != null) {
        if (f == s) {
            return f;
        }
        f = f.next;
        s = s.next;
    }
    return null;
}

@Contract(pure = true)
private static int getLen(LLNode<Integer> head) {  2 usages  new *
    int len = 0;

    LLNode<Integer> temp = head;
    while (temp != null) {
        len++;
        temp = temp.next;
    }

    return len;
}
```

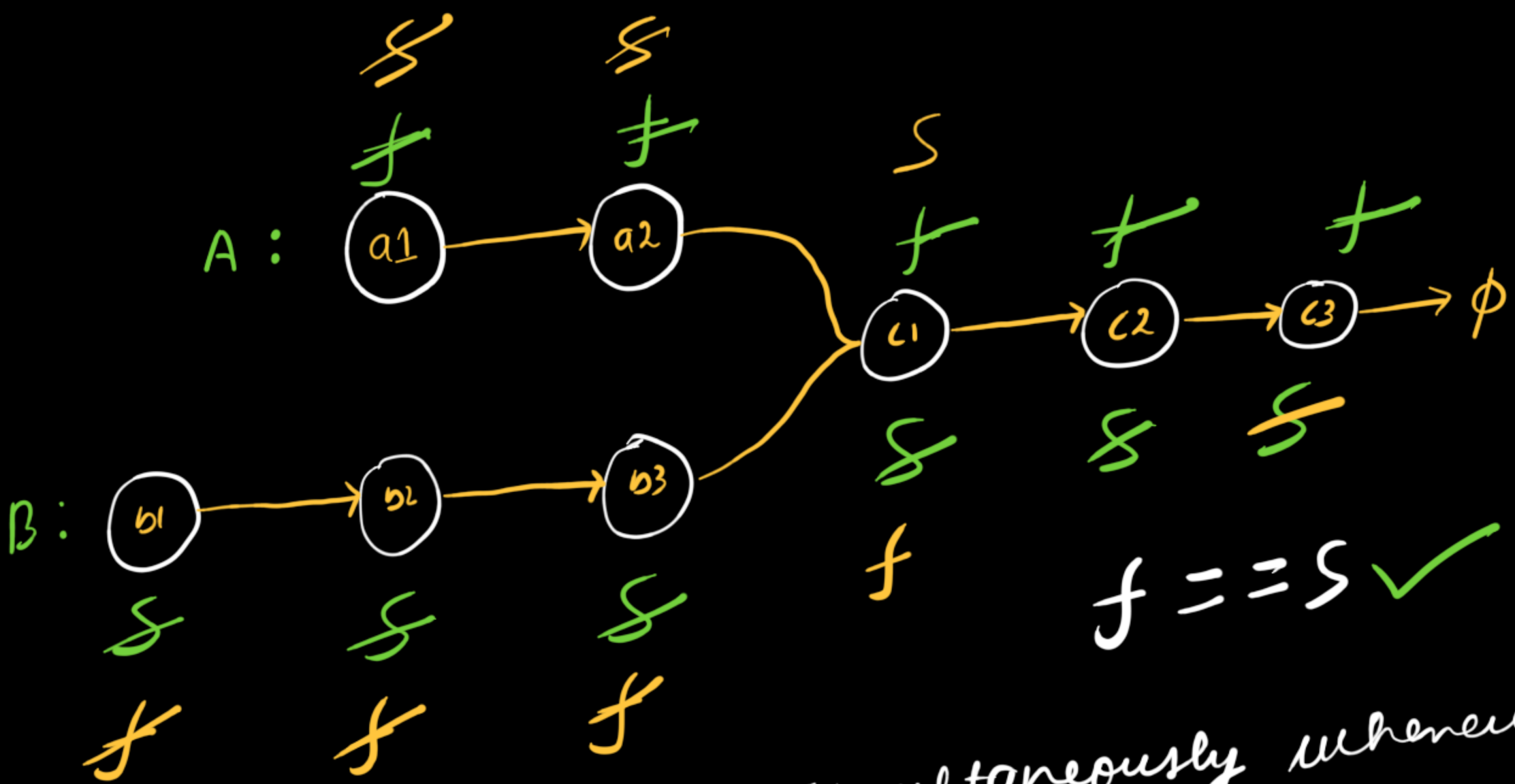
O(a+b)

O(b-a)

O(min(a,b))

TC: a+b+b-a+min(a,b)
TC: O(2b+min(a,b))
SC: O(1)

→ Approach-3



1) Traverse both list simultaneously whenever one list reaches null switch to the second list, & vice versa. At one point they will intersect & we will return the node.

PartIX.java

```java
public static LLNode<Integer> interOpt(LLNode<Integer> a , LLNode<Integer> b){
    if (a == null || b == null) return null;
    LLNode<Integer> f = a;
    LLNode<Integer> s = b;

    while (f != s){
        f = f == null ? b : f.next;
        s = s == null ? a : s.next;
    }

    return f;
}
```

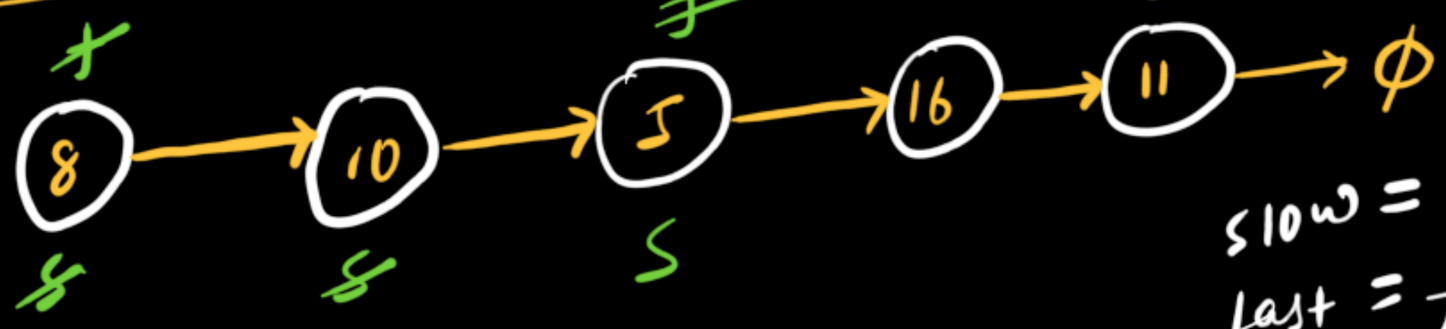
TC: O(N)

SC: O(1)

→ Find middle of the linked list (Tortoise & H

↳ Given the head of the singly linked list, return the middle node.

- For odd number of nodes



- while (fast.next != null)
  ↳ slow is the middle node

- For even numbers of Nodes

- while ((fast != null))
  ↳ slow is the middle node.

PartX.java

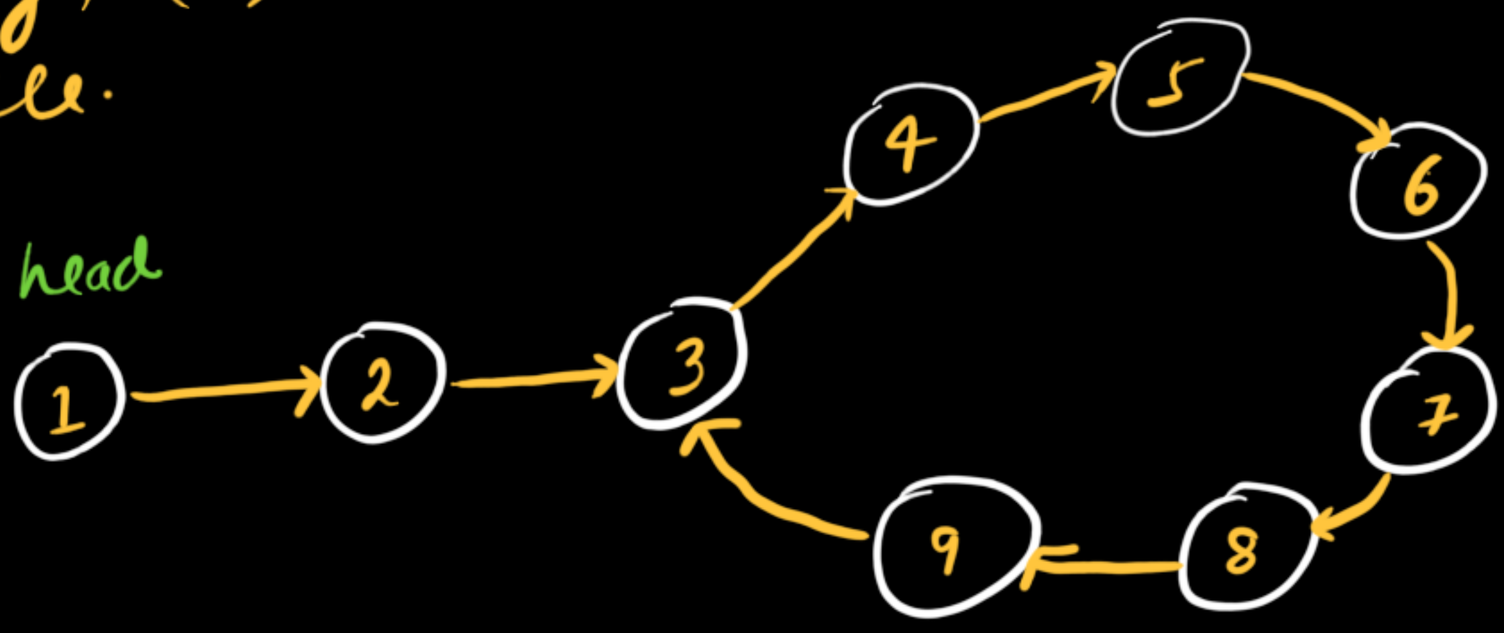
```java
public static LLNode<Integer> middleNode(LLNode<Integer> head) {

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

    LLNode<Integer> fast = head;
    LLNode<Integer> slow = head;

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

    return slow;
}
```

TC: O(N/2)

SC: O(1)

→ Detect a Loop or Cycle in Linked List

↳ Given head of the linked list, return true if it contain a cycle.



→ Approach-1

↳. use set to check if new Node being iterated is already visited or not?

↳ if visited → return true (↳ does have cycle.)

! visited → return false.

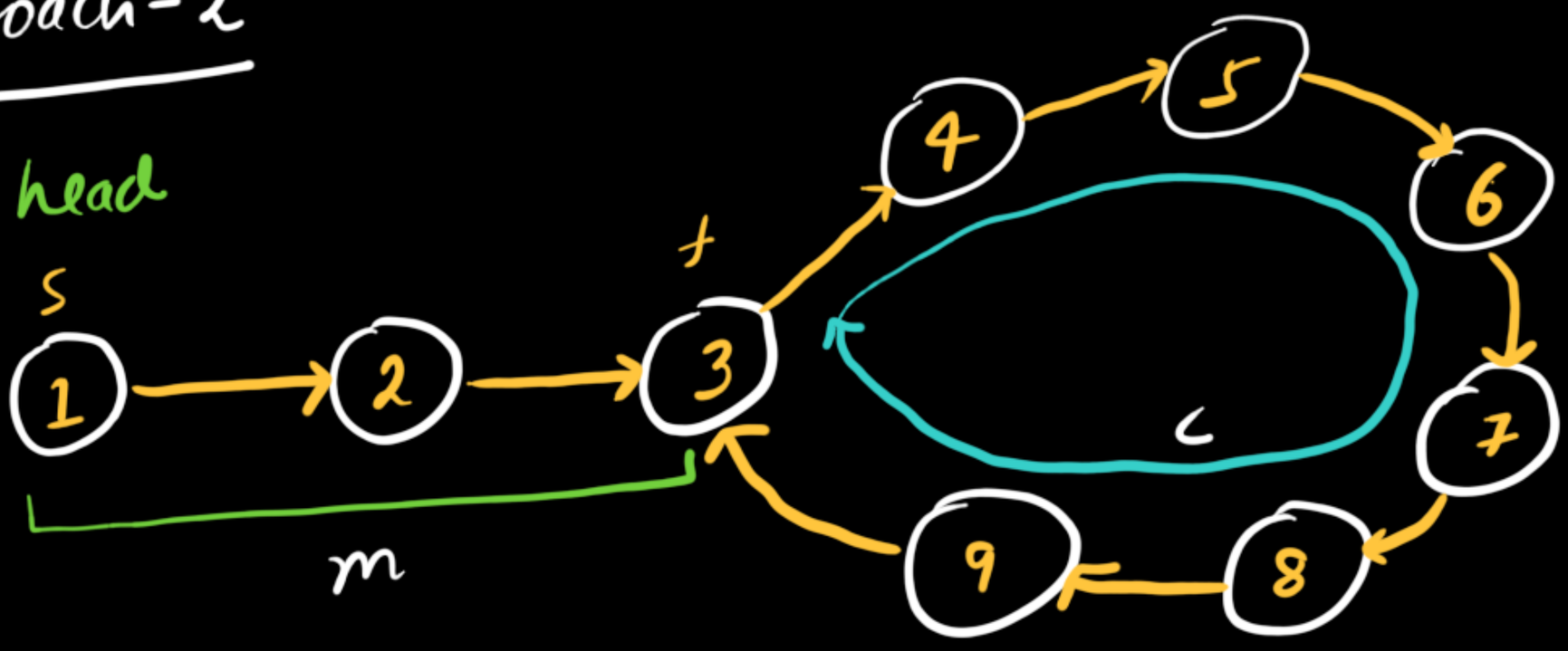
PartXI.java

```java
public static boolean containsCycle(LLNode<Integer> head) {
    if (head == null || head.next == null) return false;

    Set<LLNode<Integer>> set = new HashSet<>();
    LLNode<Integer> temp = head;

    while (temp != null) {
        if (set.contains(temp)) return true;
        else set.add(temp);
        temp = temp.next;
    }
    return false;
}
```

TC: O(N)

SC: O(N)

→ Approach-2



m = number of nodes before the cycle starts
c = length of the cycle.
s = slow pointer moves one step i.e { slow = slow.next }
f = fast pointer moves two step i.e { fast = fast.next.next }

$$f = 2s \quad \text{①}$$

- They both started at head. The distance travelled by both pointers must differ by a multiple of the cycle length (since once inside the cycle, any multiple of c brings you back to the same position)

$$f - s = nc \quad \text{②} \quad \text{for } n \geq 1$$

- slow pointer has moved nc steps
- fast pointer has moved 2nc steps.

putting ① in ②;

$$2s - s = nc$$

$$s = nc$$

→ Interpretation

- out of nc steps, the first m steps are before the cycle begins.
- The remaining (nc−k) steps are within the cycle.

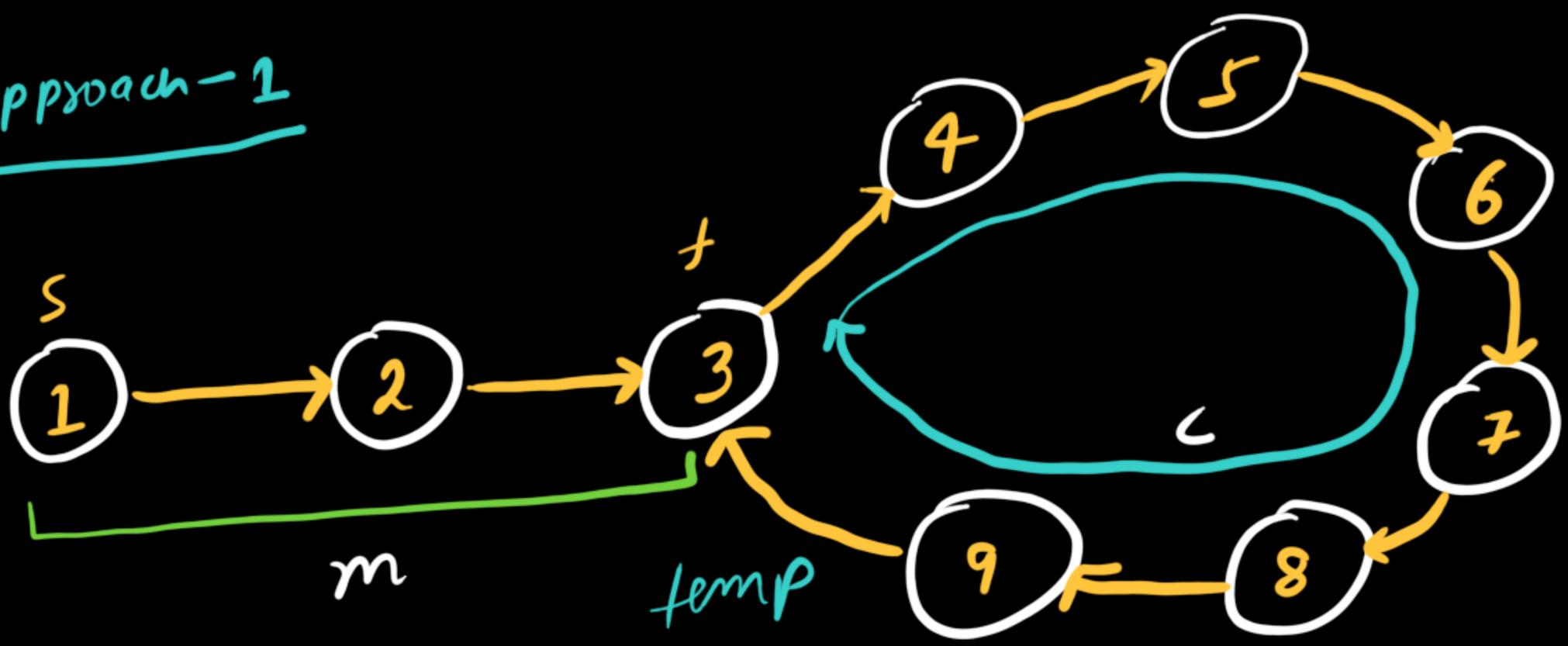
TC : O(N/2)
SC : O(1)

PartXI.java

```java
public static boolean containsCycleOpt(LLNode<Integer> head) {
    if (head == null || head.next == null) return false;

    LLNode<Integer> slow = head;
    LLNode<Integer> fast = head;

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

        if (slow == fast) return true;
    }
    return false;
}
```

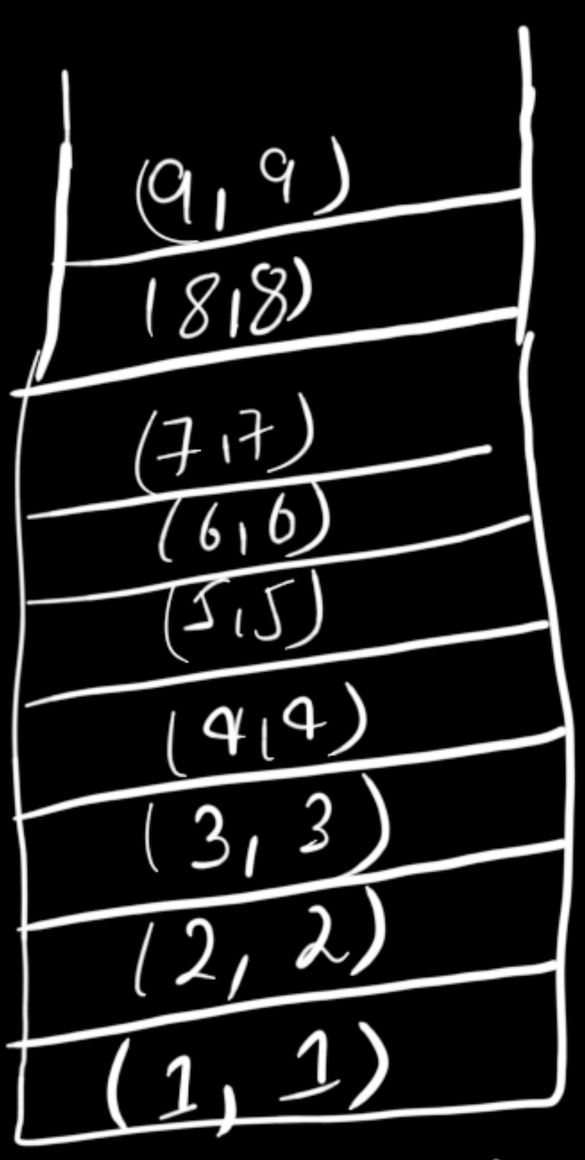
→ Find the length of the Loop in LinkedList

→ Approach-1



count = 0

1
2
3
4
5
6
7
8
9



<Node, count>

→ Again we reach at Node → (3) so length of the Linked List is

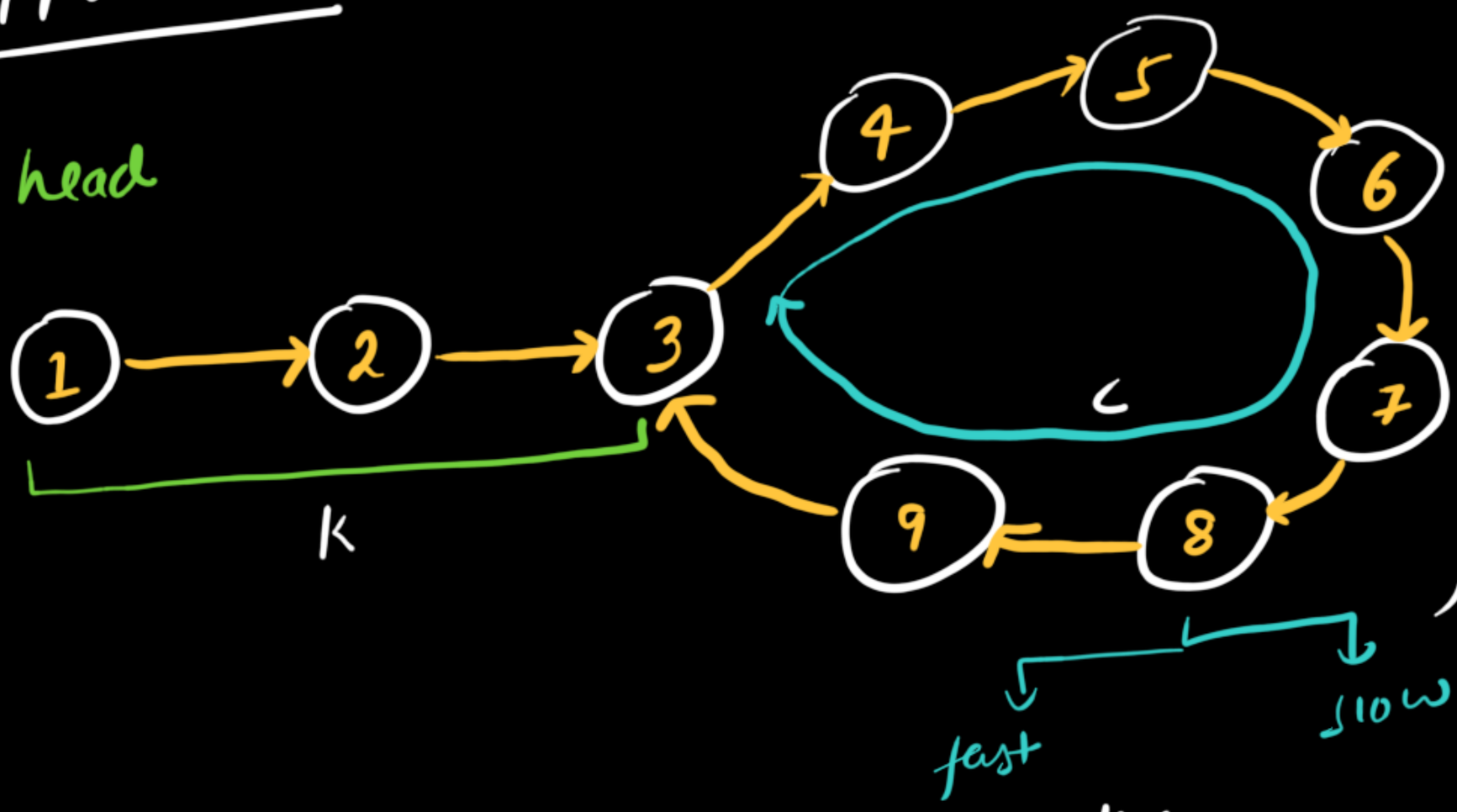
len = current_count - map.get(temp)

PartXII.java

```java
public static int lengthOfLL(LLNode<Integer> head) {  no
    if (head == null || head.next == null) return 0;

    Map<LLNode<Integer>, Integer> map = new HashMap<>();
    LLNode<Integer> temp = head;
    int count = 0;

    while (temp != null) {
        if (map.containsKey(temp)) {
            return count - map.get(temp);
        }
        map.put(temp, count);
        temp = temp.next;
        count++;
    }
    return 0;
}
```

TC : O(N)
SC : O(N)

→ Approach-2



(1) Find the point where both pointers fast & slow meets.

(2) Now again iterate through the cycle, while fast again meets the slow keep counting.

↳ Initially BOTH fast & slow will be equal, so move fast by 1 & then start the iteration by keeping a count variable, keep on incrementing until fast reaches slow.

TC: $O(K+C) + O(C) \approx O(N)$

SC: $O(1)$

PartXII.java

```java
public static int lengthOfLLOpt(LLNode<Integer> head) {  1 usage  new *
    if (head == null || head.next == null) return 0;

    LLNode<Integer> slow = head;
    LLNode<Integer> fast = head;

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

        if (slow == fast) {
            return len(slow, fast);
        }
    }
    return 0;
}

@Contract(pure = true)
private static int len(LLNode<Integer> slow, LLNode<Integer> fast) {
    int count = 1;
    fast = fast.next;

    while (slow != fast){
        count++;
        fast = fast.next;
    }
    return count;
}
```

$O(K+C)$ → no of nodes before cycle starts.

$O(C)$ → length of the cycle.

→ delete middle node of the Linked List

→ Approach-1

- Find the length of the linked list & divide it by two
- traverse one time less than the (len/2)
- current_node.next = current.next.next

TC: O(N) + O(N/2)

SC: O(1)

→ Approach-2

↳ • Apply tortoise & hare Algorithm to find middle node.
- keep a prev node before the middle node.
  ↳ prev.next = slow.next;

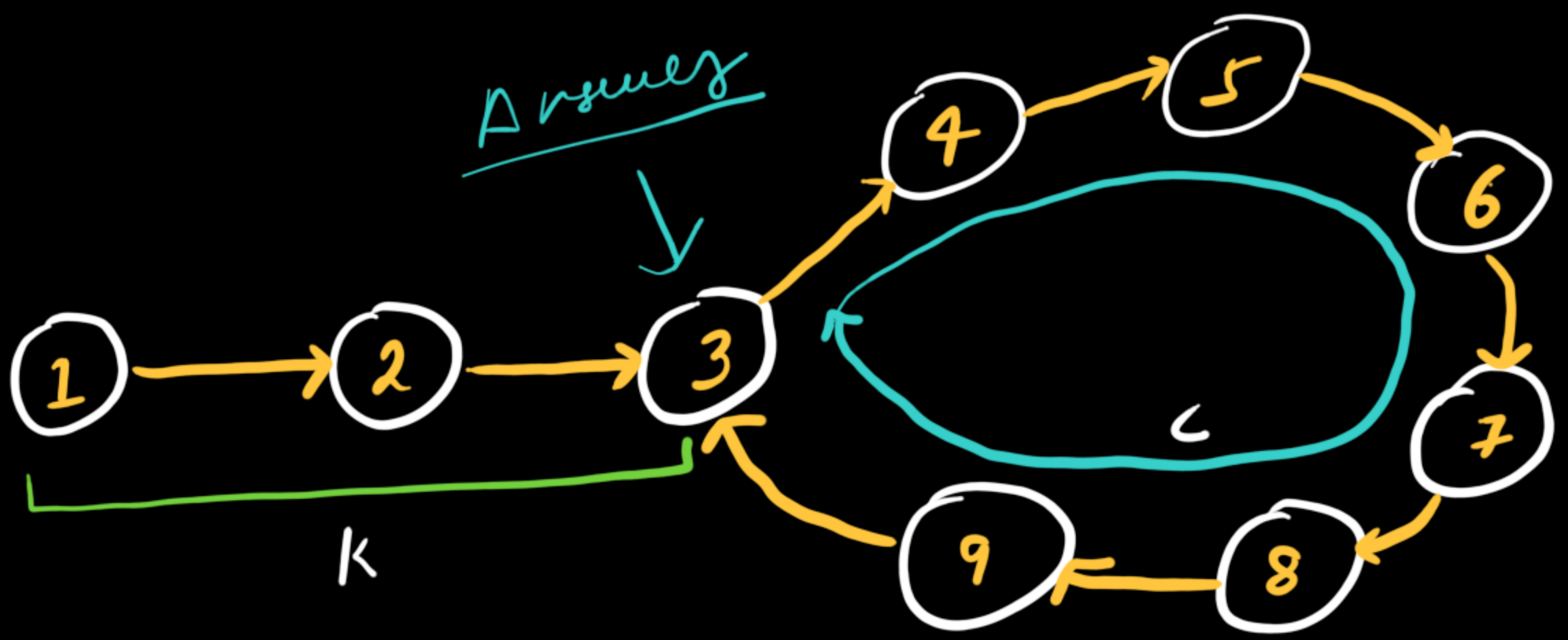
PartXIII.java

```java
public static LLNode<Integer> deleteMiddle(LLNode<Integer> head) {
    if (head == null || head.next == null) return head;

    LLNode<Integer> prev = null;
    LLNode<Integer> slow = head;
    LLNode<Integer> fast = head;

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

TC: O(N/2)

SC: O(1)

→ Find Start of a cycle in LL



→ Approach - 1

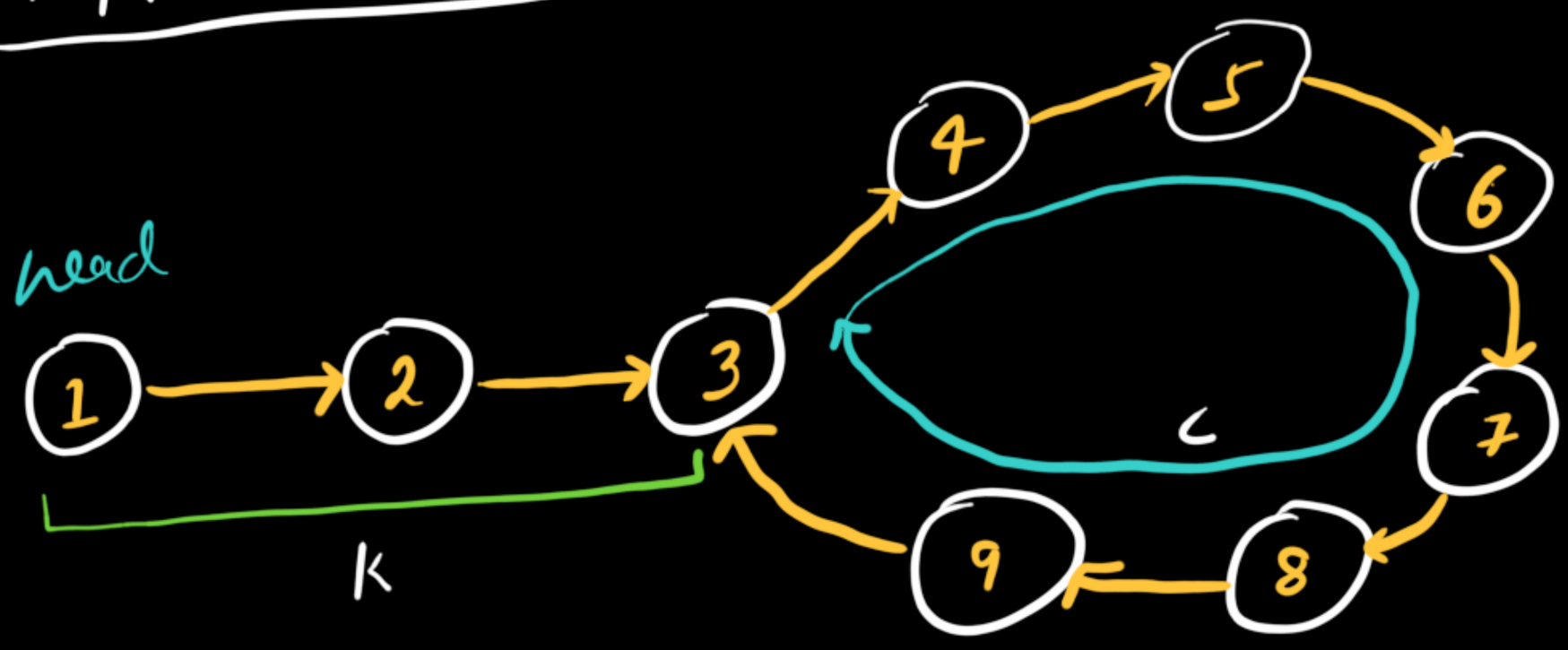
- Use Set data structure, whenever we encounter the same node, we simply return that node.

PartXIV.java

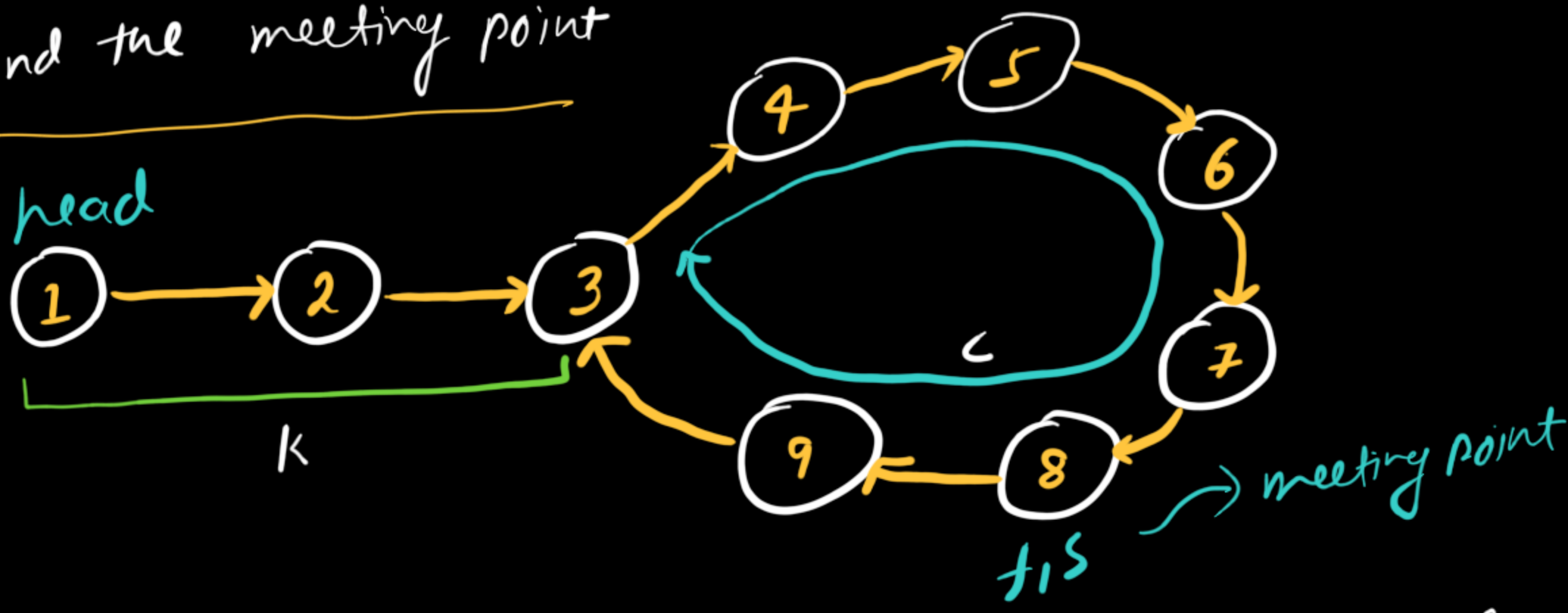
```java
public static LLNode<Integer> startOfCycle(LLNode<Integer> head) {
    if (head == null || head.next == null) return head;

    Set<LLNode<Integer>> set = new HashSet<>();
    LLNode<Integer> temp = head;

    while (temp != null) {
        if (set.contains(temp)) {
            return temp;
        }
        set.add(temp);
        temp = temp.next;
    }
    return null;
}
```

TC : O(N)
SC : O(N)

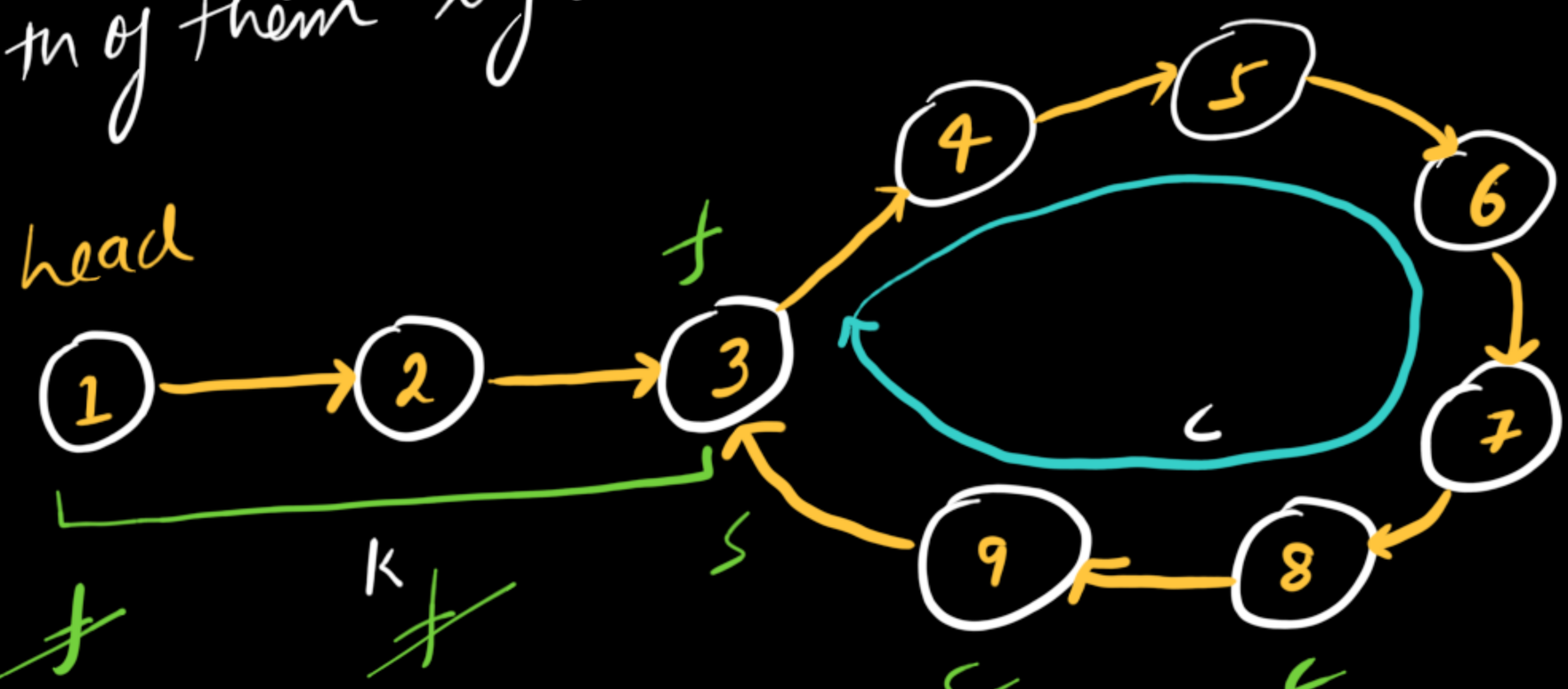
→ Approach - 2



① Find the meeting point



② Reset one of the pointers to head & then traverse both of them by one step.



↳ They will meet at the starting point of the cycle; return any one of them

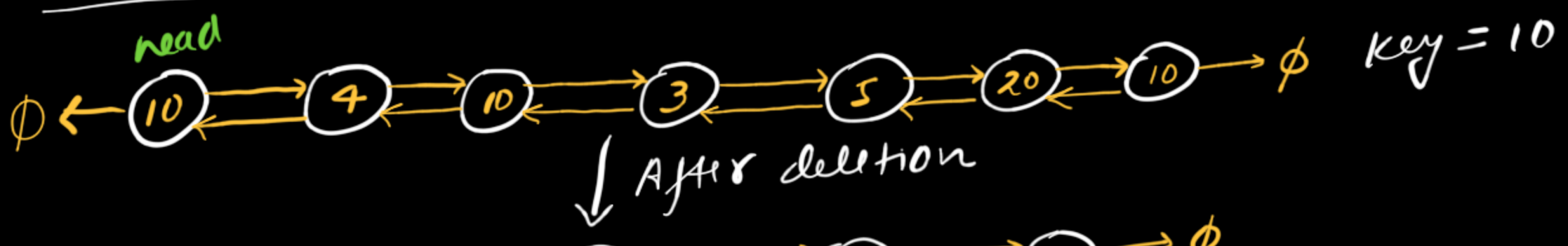
Tc: O(N)

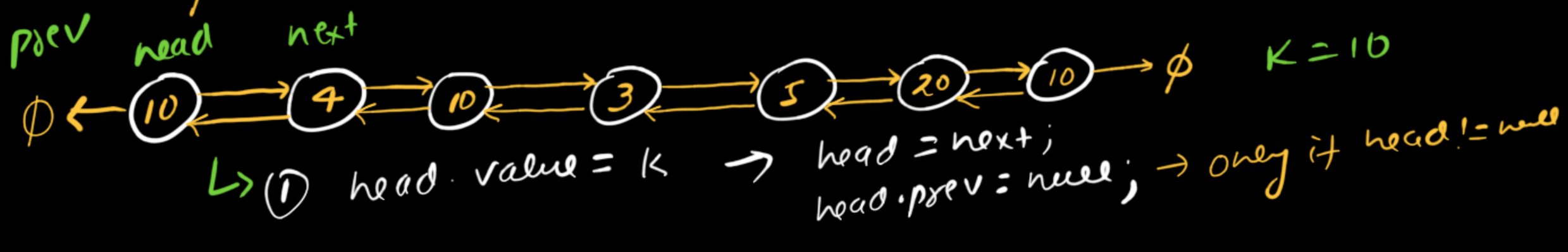
Sc: O(1)

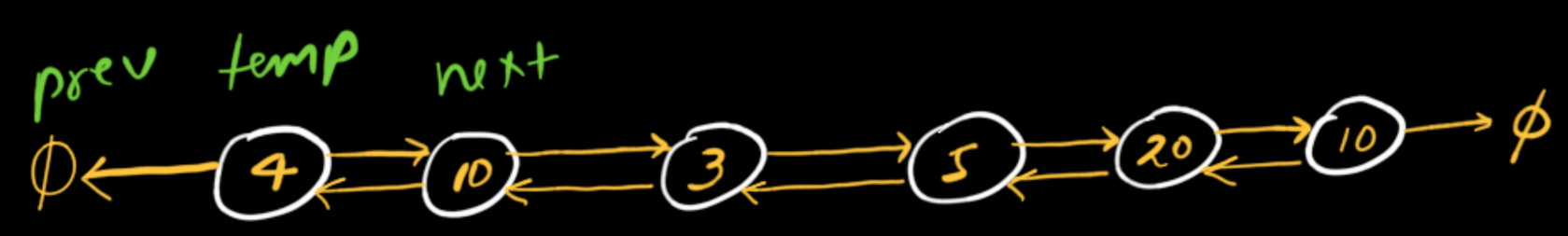
PartXIV.java

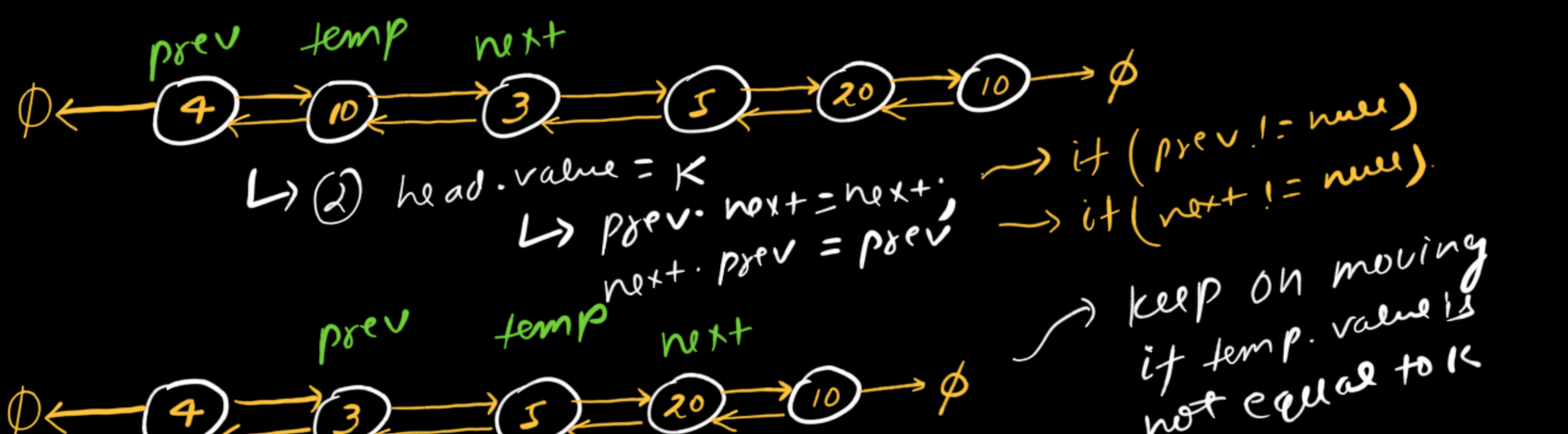
```java
public static LLNode<Integer> startOfCycleOpt(LLNode<Integer> head){
    if (head == null || head.next == null) return head;

    LLNode<Integer> slow = head;
    LLNode<Integer> fast = head;

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

        if (slow == fast) break;
    }

    fast = head;

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

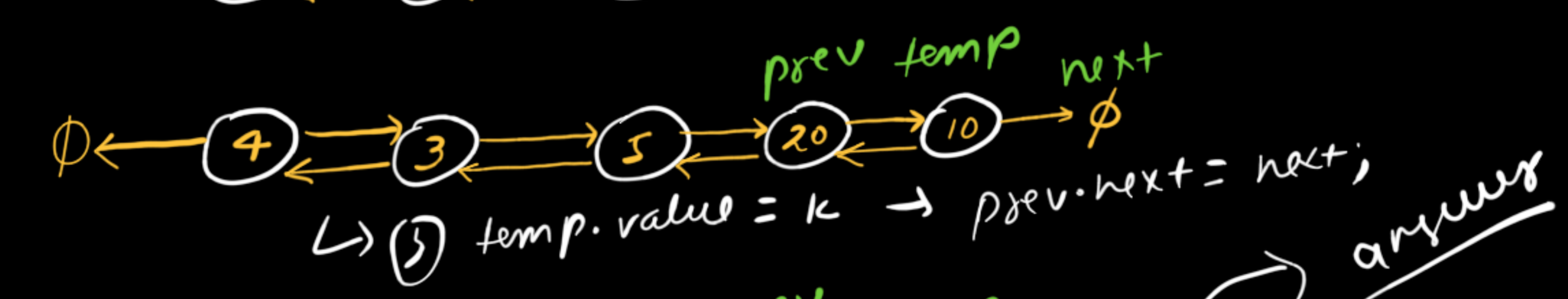
→ Delete all occurrences of a key in a DLL











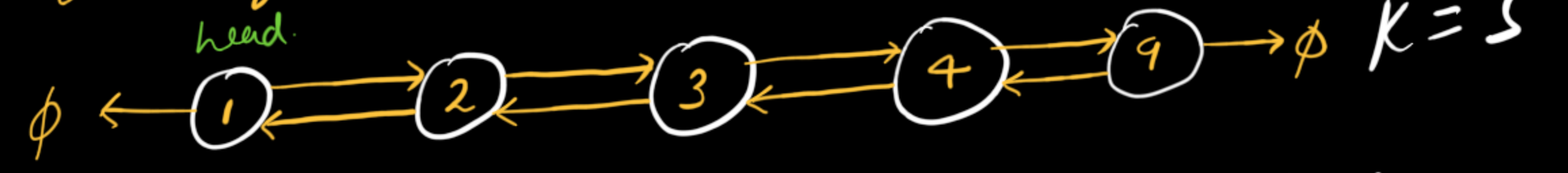

```java
public static DLLNode<Integer> deleteAllOccurrences(DLLNode<Integer> head, int k) {
    if (head == null) return null;

    DLLNode<Integer> temp = head;

    while (temp != null) {
        if (temp.value.equals(k)) {
            DLLNode<Integer> prev = temp.prev;
            DLLNode<Integer> next = temp.next;

            if (temp == head) {
                head = next;
                if (head != null) head.prev = null;
            } else {
                if (prev != null) prev.next = next;
                if (next != null) next.prev = prev;
            }

            temp = next;
        } else {
            temp = temp.next;
        }
    }

    return head;
}
```

TC: O(N)

SC: O(1)

→ Find pairs with given sum in sorted DLL

↳ given sorted doubly linked list of positive distinct elements, the task is to find pairs in a doubly-linked list whose sum is equal to given value target.



→ Approach-1

- iterate over the doubly linked list & check each time if t.value + s.value <= k, add it to the data structure

TC : O(N²)

SC : O(1)

PartXVI.java

```java
public static @NotNull List<Integer[]> findPair(DLLNode<Integer> head, int k) {
    DLLNode<Integer> temp = head;
    List<Integer[]> ans = new ArrayList<>();

    while (temp != null){
        DLLNode<Integer> s = temp.next;

        while (s != null && temp.value + s.value <= k){
            if (temp.value + s.value == k){
                ans.add(new Integer[]{temp.value , s.value});
            }
            s = s.next;
        }
        temp = temp.next;
    }
    return ans;
}
```

→ Approach-2

↳ • insert node value as key & temp node as value in hashmap.

- target = k - temp.value, if target is present inside map, then we have one of our pair.

PartXVI.java

```java
public static @NotNull List<Integer[]> findPairBetter(DLLNode<Integer> head, int k) {
    DLLNode<Integer> temp = head;
    Map<Integer, DLLNode<Integer>> map = new HashMap<>();
    List<Integer[]> ans = new ArrayList<>();

    while (temp != null) {
        Integer target = k - temp.value;

        if (map.containsKey(target)) {
            ans.add(new Integer[]{temp.value, target});
        }
        map.put(temp.value, temp);
        temp = temp.next;
    }
    return ans;
}
```

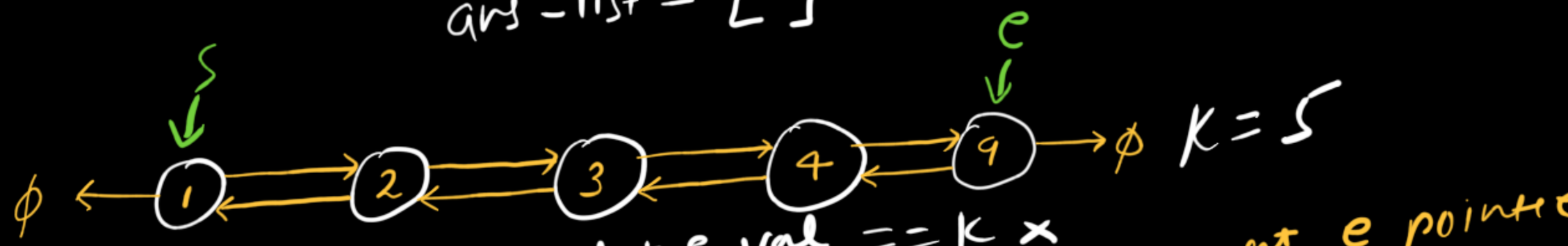
TC : O(N)

SC : O(N)

→ Approach-3

- Since array is sorted we can take advantage of its properties.

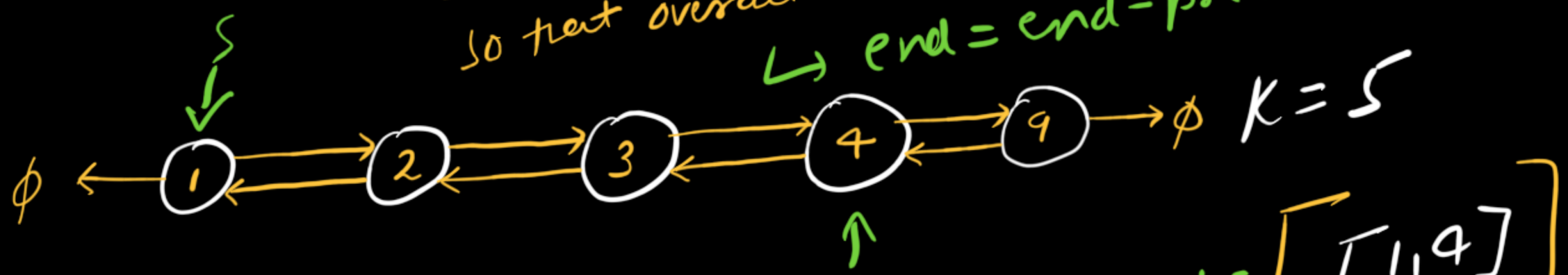
ans-list = [ ]



↳ sum = s.val + e.val == K ✗

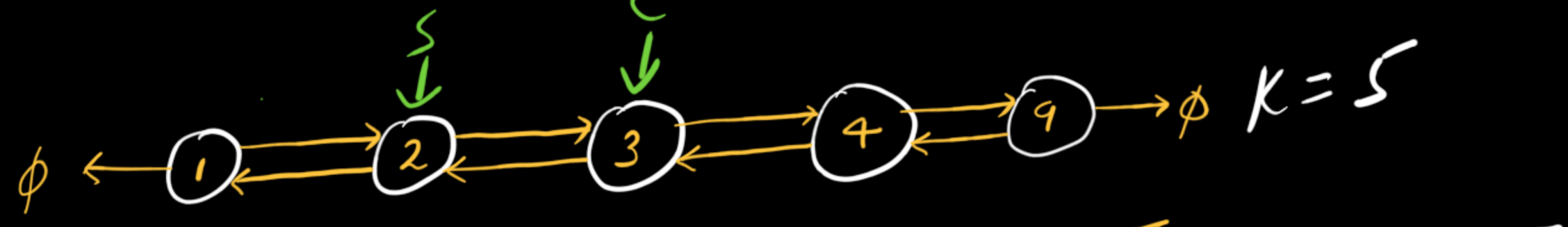
- Sin sum > K, we need to decrement e pointer so that overall sum decreases.

↳ end = end.prev



- sum = s.val + e.val == K ans-list = [[1,4]]

↳ s = s.next, end = end.prev



- sum = s.val + e.val == K ans = [[1,4],[2,3]]

↳ s = s.next, end = end.prev

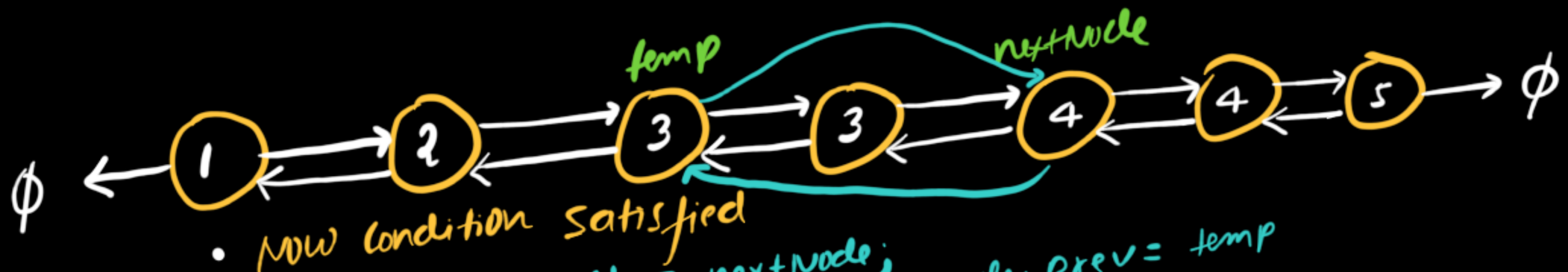
TC: O(N)
SC: O(1)

PartXVI.java

```java
public static @NotNull List<Integer[]> findPairOpt(DLLNode<Integer> head, int k) {
    DLLNode<Integer> start = head;
    DLLNode<Integer> end = head;
    List<Integer[]> ans = new ArrayList<>();

    while (end.next != null) end = end.next;

    while (start.value < end.value) {
        int sum = start.value + end.value;

        if (sum == k){
            ans.add(new Integer[]{start.value , end.value});
            start = start.next;
            end = end.prev;
        }
        else if (sum > k){
            end = end.prev;
        }
        else start = start.next;
    }
    return ans;
}
```

→ Remove duplicates from sorted DLL

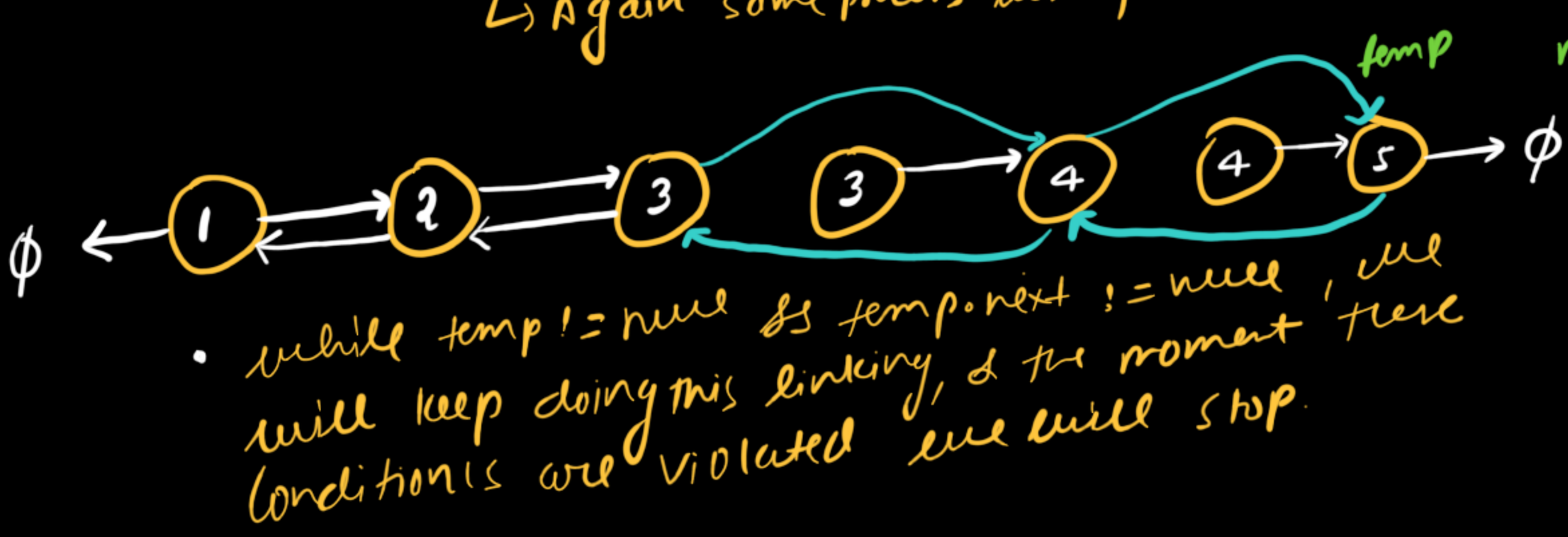
- move nextNode until temp.value != nextNod.value.
- move nextnode one by one until ↳ this condition is not satisfied.



- now condition satisfied

```
temp.next = nextNode;
if( nextNode != null) nextNode.prev = temp
temp = nextNode;
```

↳ Again same process will follow



- while temp != null && temp.next != null we will keep doing this linking, & the moment, these conditions are violated we will stop.

PartXVII.java

```java
public static DLLNode<Integer> removeDuplicates(DLLNode<Integer> head){
    if (head == null || head.next == null) return head;

    DLLNode<Integer> temp = head;

    while (temp != null && temp.next != null){
        DLLNode<Integer> nextNode = temp.next;

        while (nextNode != null && nextNode.value.equals(temp.value)){
            nextNode = nextNode.next;
        }

        temp.next = nextNode;
        if (nextNode != null) nextNode.prev = temp;
        temp = temp.next;
    }

    return head;
}
```

TC : O(N)
SC : O(1)

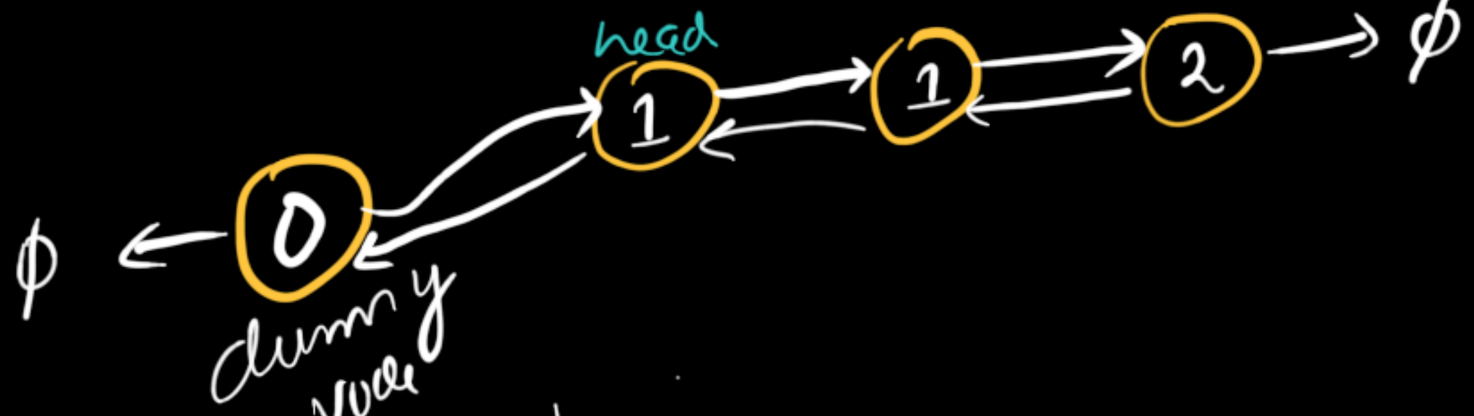
→ Follow up:

→ what if we have to remove all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

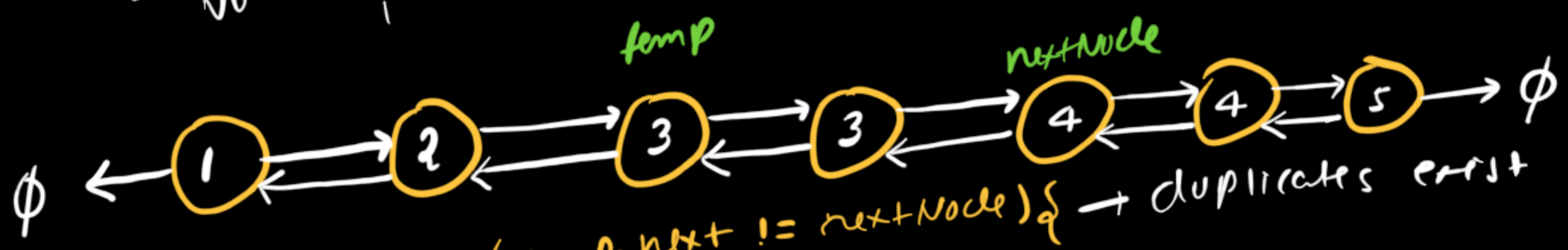
① what if head may get removed?

↳

• To remove head, connect dummy node to the head




```
dummy = new node(0);
dummy.next = head;
head.prev = dummy;
```



→ duplicates exist

```
while(temp.next != nextNode){
prevNode = temp.prev
prevNode.next = nextNode;
if(nextNode != null) nextNode.prev = prevNode.
temp = nextNode
} else {
temp = temp.next;
}
```

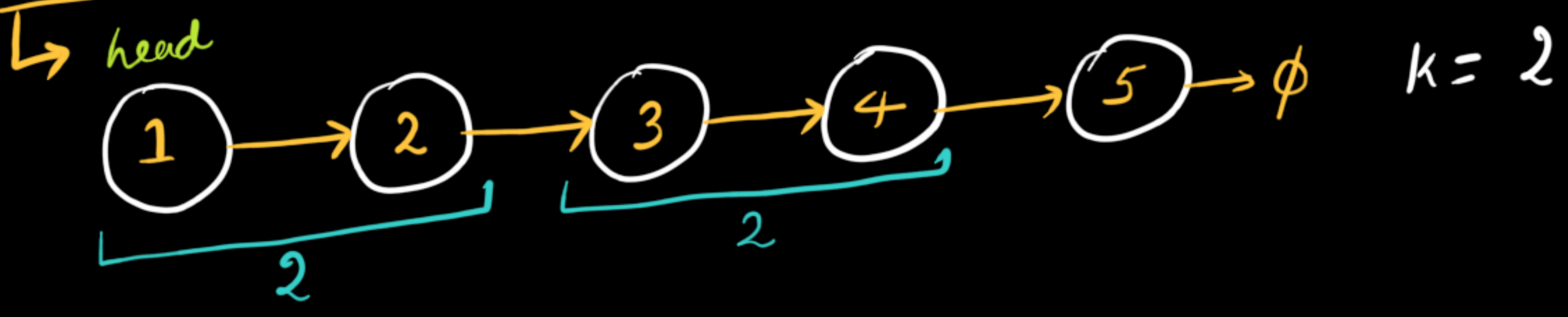
PartXVII.java

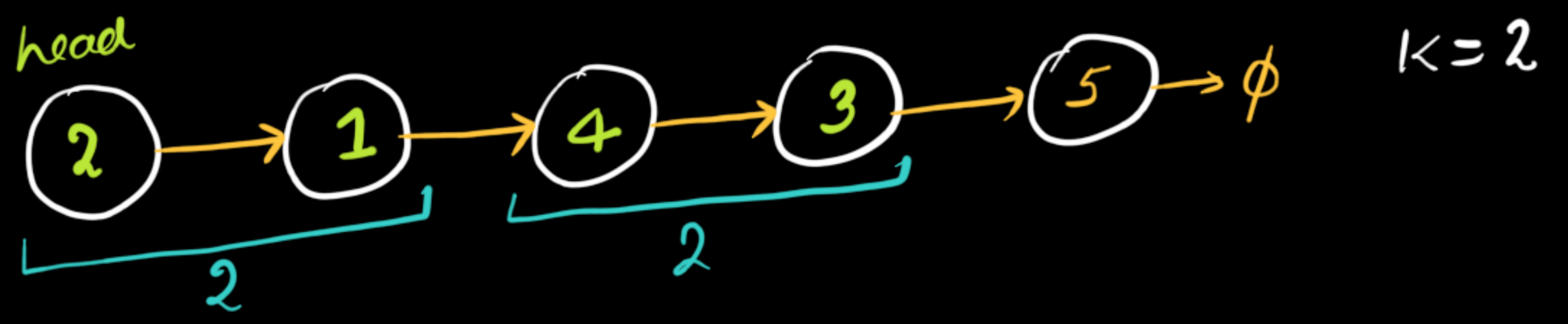
```java
public static DLLNode<Integer> removeDuplicatesII(DLLNode<Integer> head) {
    if (head == null || head.next == null) return head;

    DLLNode<Integer> dummy = new DLLNode<>( value: 0);
    dummy.next = head;
    head.prev = dummy;

    DLLNode<Integer> temp = head;

    while (temp != null) {
        DLLNode<Integer> nextNode = temp.next;

        while (nextNode != null && nextNode.value.equals(temp.value)) {
            nextNode = nextNode.next;
        }

        if (temp.next != nextNode){ // duplicates detected
            DLLNode<Integer> prevNode = temp.prev;
            prevNode.next = nextNode;
            if (nextNode != null) nextNode.prev = temp;
            temp = nextNode;
        }
        else temp = temp.next; // no duplicates
    }
    return head;
}
```

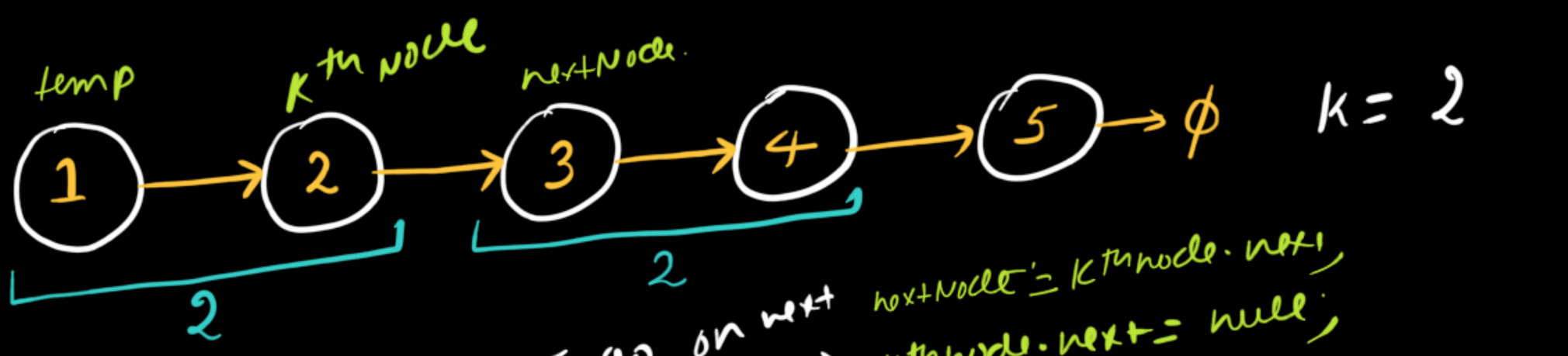
TC: O(N)

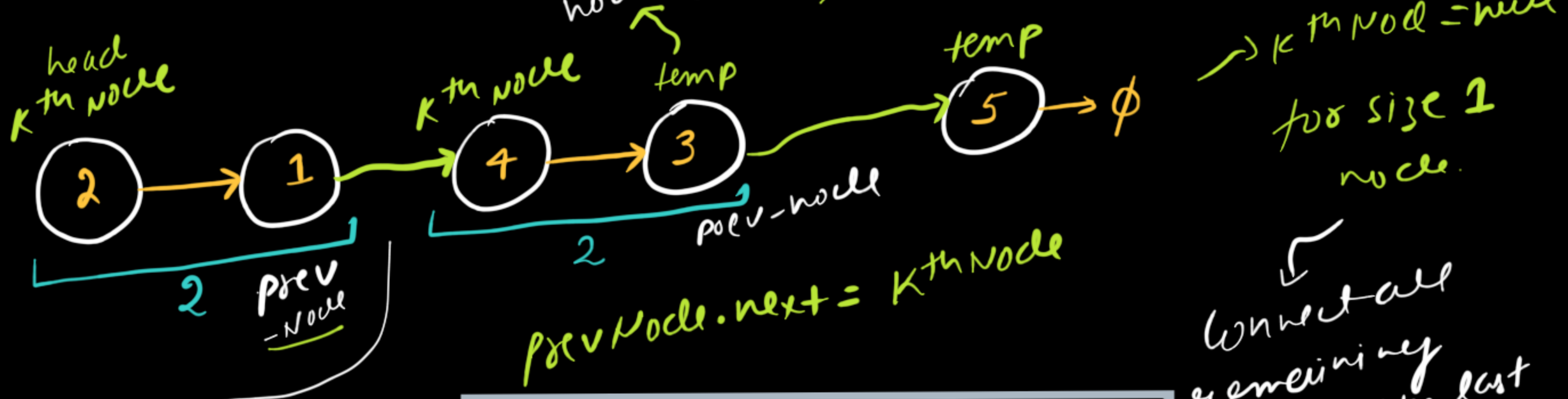
SC: O(1)

→ Reverse nodes in k-group










```java
public static LLNode<Integer> reverseKGroup(LLNode<Integer> head, int k) {
    if (head == null || head.next == null || k <= 1) {
        return head;
    }

    LLNode<Integer> temp = head;
    LLNode<Integer> prev = null;

    while (temp != null) {
        LLNode<Integer> kthNode = getKthNode(temp, k);
        if (kthNode == null) {
            // connect remainder if less than k nodes left
            if (prev != null) prev.next = temp;
            break;
        }

        LLNode<Integer> nextNode = kthNode.next;
        kthNode.next = null;

        LLNode<Integer> reversedHead = reverseListOpt(temp);

        if (temp == head) head = reversedHead;
        else prev.next = reversedHead;

        // temp becomes the tail of the reversed group
        prev = temp;
        temp = nextNode;
    }

    return head;
}

@Contract(pure = true)
private static LLNode<Integer> getKthNode(LLNode<Integer> head, int k) {
    LLNode<Integer> temp = head;
    while (--k > 0 && temp != null) {
        temp = temp.next;
    }
    return temp;
}
```

TC: O(n/k) * O(k)

outer loop — for finding kth node

SC: O(1)

→ Rotate a List

Steps:

- Count length & point the tail to head to form circle.

- Traverse (len - k) nodes i.e (5-2) = 3 nodes

new head = temp.next
temp.next = null

Our list is reversed now.

TC: O(N)
SC: O(1)

```java
public static LLNode<Integer> rotate(LLNode<Integer> head, int k) {
    if (head == null || head.next == null) return head;

    int len = 1;
    LLNode<Integer> temp = head;

    while (temp.next != null) {
        len++;
        temp = temp.next;
    }
    temp.next = head;

    k = k % len;

    if (k == 0) {
        temp.next = null;
        return head;
    }
    int diff = len - k;
    temp = head;

    while (temp != null && --diff > 0) {
        temp = temp.next;
    }
    LLNode<Integer> nextHead = temp.next;
    temp.next = null;

    return nextHead;
}
```

→ merge two sorted list

while (list1 != null && list2 != null){

if(list1.val ≥ list2.val) {

dummy.next = list2;

list2 = list2.next;

} else {

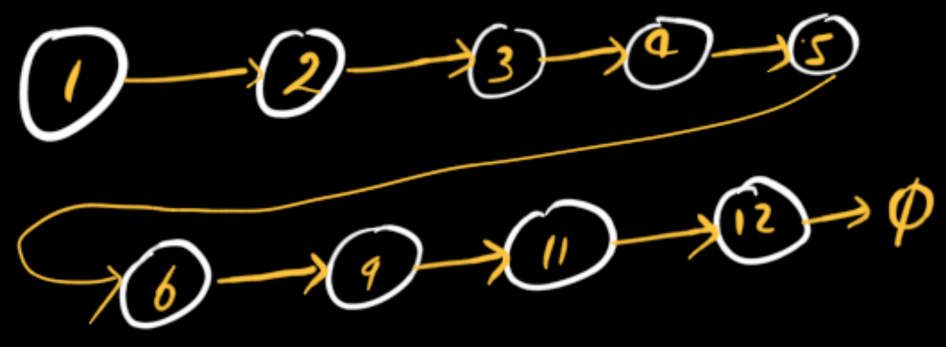
dummy.next = list1;

list1 = list1.next;

}

dummy = dummy.next;

}



if any list reaches null, & any pointer is not at null connect that list attached.

dummy.next = list1 != null ? list1 : list2;

```java
public static LLNode<Integer> mergeTwoLists(LLNode<Integer> list1, LLNode<Integer> list2) {
    if (list1 == null) return list2;
    if (list2 == null) return list1;

    return merge(list1, list2);
}

private static LLNode<Integer> merge(LLNode<Integer> list1, LLNode<Integer> list2) {  1 usage
    LLNode<Integer> newList = new LLNode<~>( value: 0);
    LLNode<Integer> temp = newList;

    while (list1 != null && list2 != null) {
        if (list1.value >= list2.value) {
            temp.next = list2;
            list2 = list2.next;
        } else {
            temp.next = list1;
            list1 = list1.next;
        }
        temp = temp.next;
    }

    temp.next = list1 != null ? list1 : list2;
    return newList.next;
}
```

TC : O(n+m)

SC : O(1)

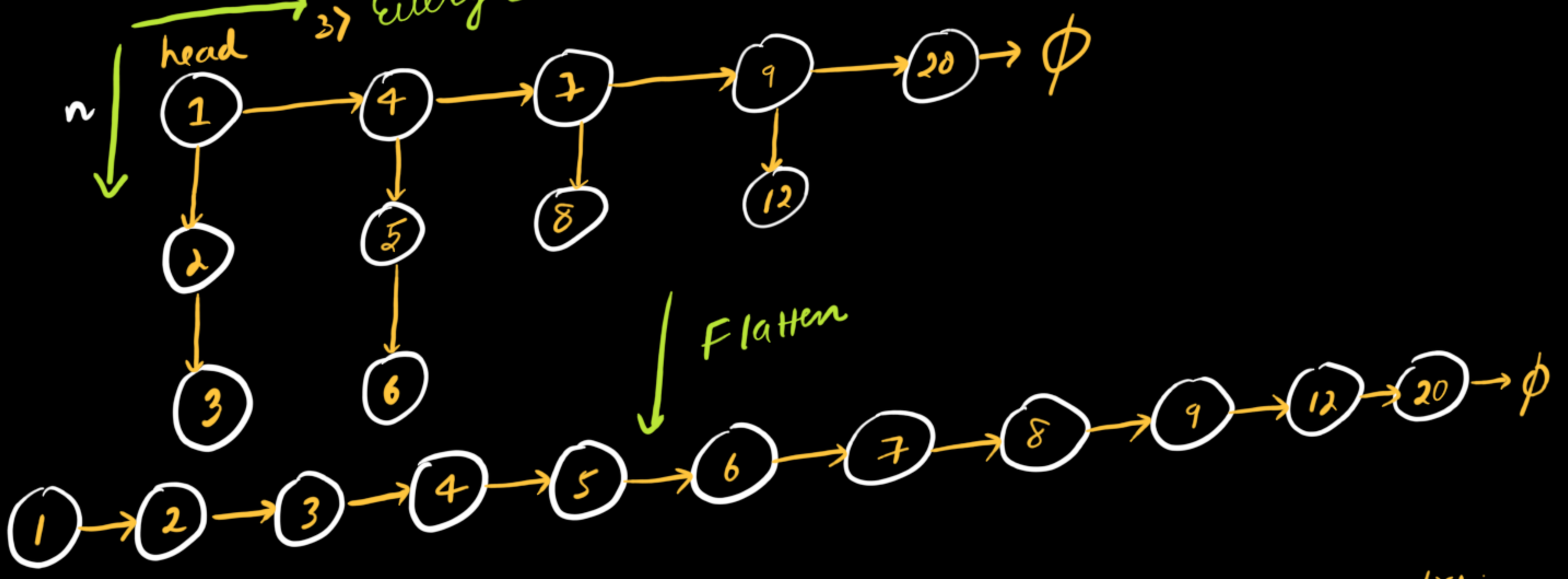
↓

ones than a singer node.

→ Flatten a Linked List

↳ • Given a linked list containing n head nodes, where every node in the linked list contains two pointers.

1) next → points to the next node in the list
2) child → points to a linked list where the current node is head
3) Every child node is sorted in increasing order



→ Approach - 1

- Traverse through child & next pointed add it in array list;
- Sort the array list
- Convert the sorted list into a flattened linked list.

```
PartXXI.java
public static Node flattenLinkedList(Node head) {  1 usage  new *
    if (head == null || (head.next == null && head.child == null)) {
        return head;
    }

    Node temp = head;
    List<Integer> list = new ArrayList<>();
    while (temp != null) {
        Node childTemp = temp;

        while (childTemp != null) {
            list.add(childTemp.data);
            childTemp = childTemp.child;
        }
        temp = temp.next;
    }
    Collections.sort(list);
    return convert(list);
}

private static @Nullable Node convert( @NotNull List<Integer> ls) {  1
    if (ls.isEmpty()) return null;

    Node dummy = new Node( data: -1);
    Node temp = dummy;

    for (int e : ls) {
        temp.next = new Node(e);
        temp = temp.next;
    }
    return dummy.next;
}
```

O(m×n)

O(m×n log(m×n))

O(m×n)

TC: 2×O(m×n) + m×n log(m×n)

TC = m×n(2 + log(m×n))

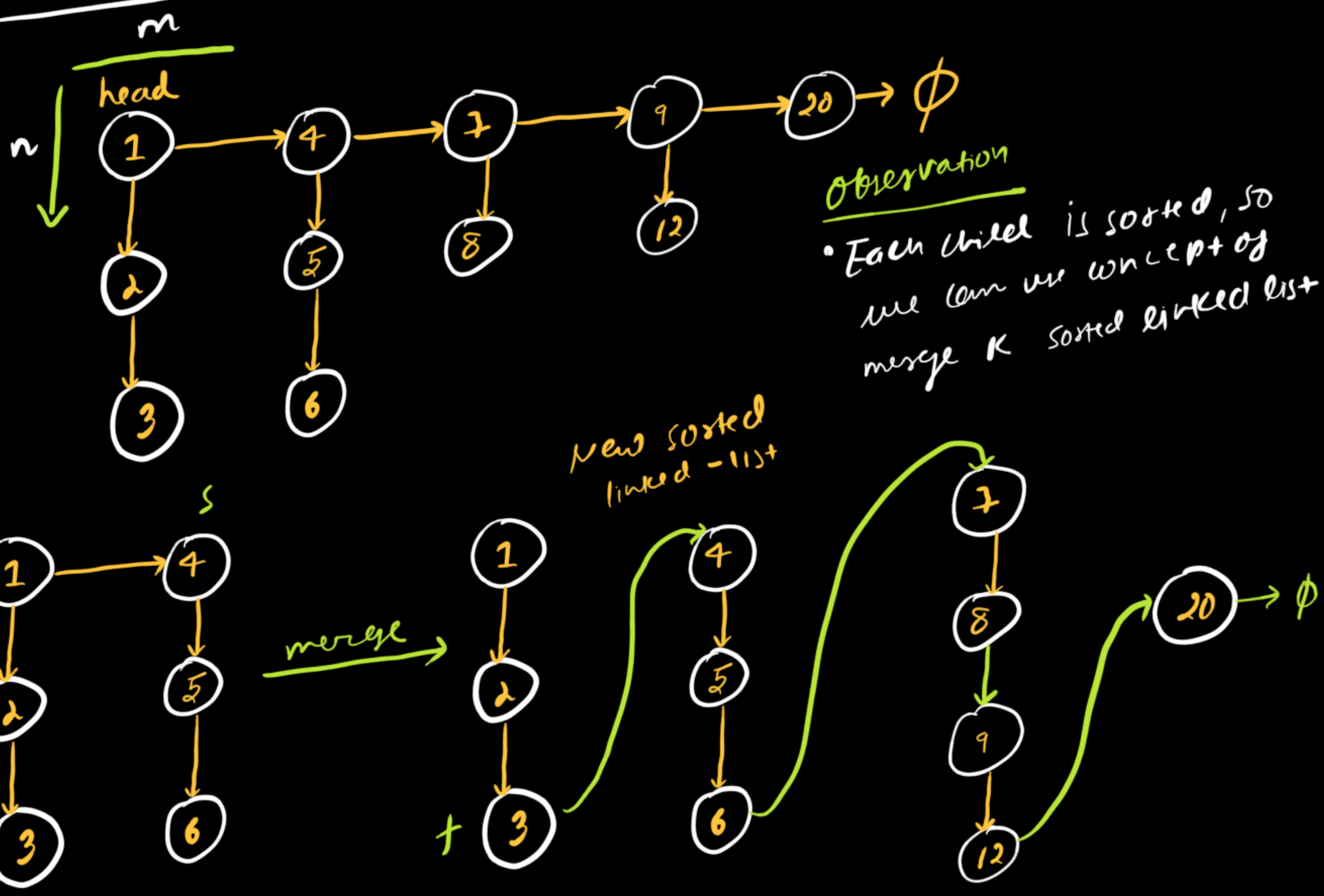
TC = (m×n) log(m×n)

SC: O(m×n)

m = no. of nodes using next pointer

n = no of nodes using child pointer

→ Approach-2



- Recursively traverse to the last list
- pass the current head & returned merged head to the merge function

Tc: $m \times (m+n)$ {m == n}
$= m \times (2N) \approx (2mN)$

SC: O(N)
↳ Recursive Stack Space.

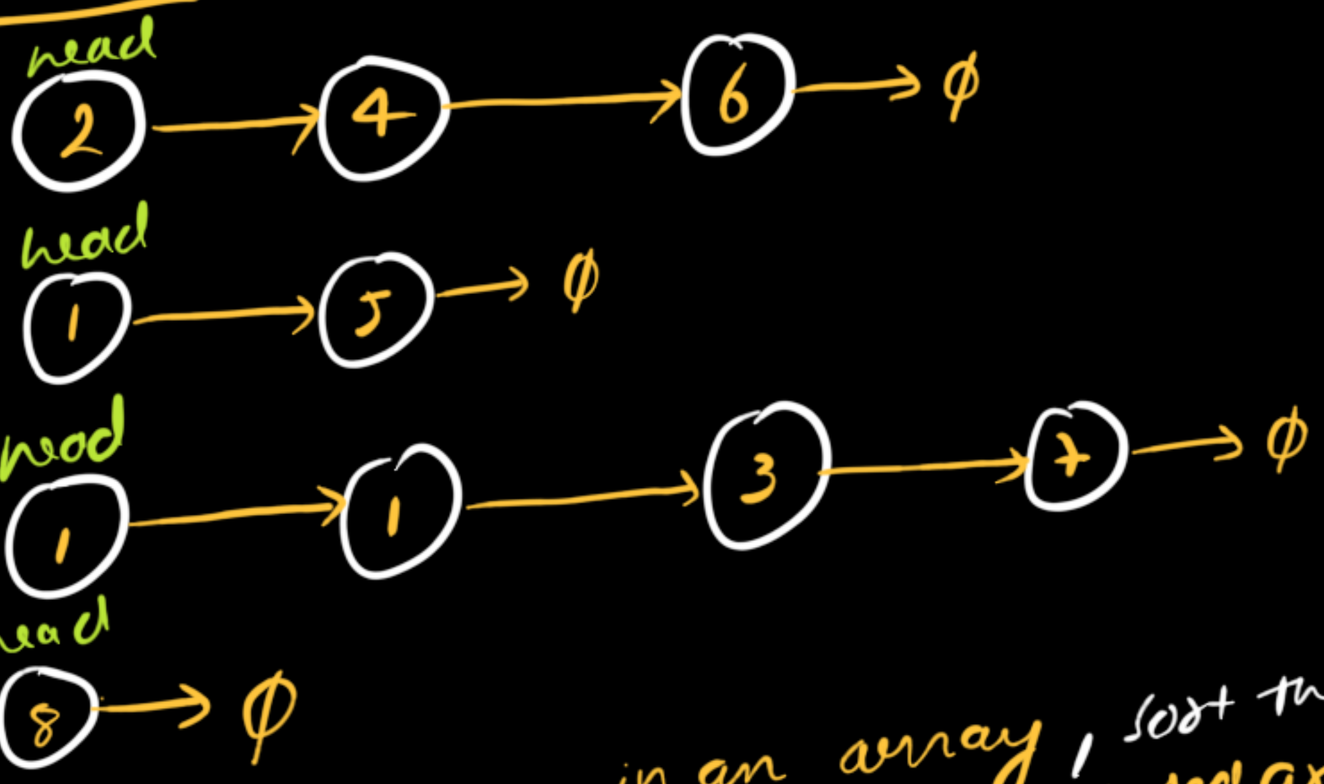
↳ length of horizontal chain

```
PartXXI.java
public static Node flattenLinkedListOpt(Node head) {
    if (head == null || head.next == null){
        return head;
    }
    Node mergedHead = flattenLinkedListOpt(head.next);
    head.next = null; // break the horizontal chain

    return merge(head , mergedHead);
}
private static Node merge(Node list1, Node list2) {
    Node dummy = new Node( data: 0);
    Node temp = dummy;

    while (list1 != null && list2 != null) {
        if (list1.data <= list2.data) {
            temp.child = list1;
            list1 = list1.child;
        } else {
            temp.child = list2;
            list2 = list2.child;
        }
        temp = temp.child;
    }

    temp.child = (list1 != null) ? list1 : list2;

    return dummy.child;
}
```

Tc: O(m+n)

→ Merge K sorted Linked List

→ Approach-1



- Traverse & Store them in an array, sort them
- Again build the linked list from the sorted array

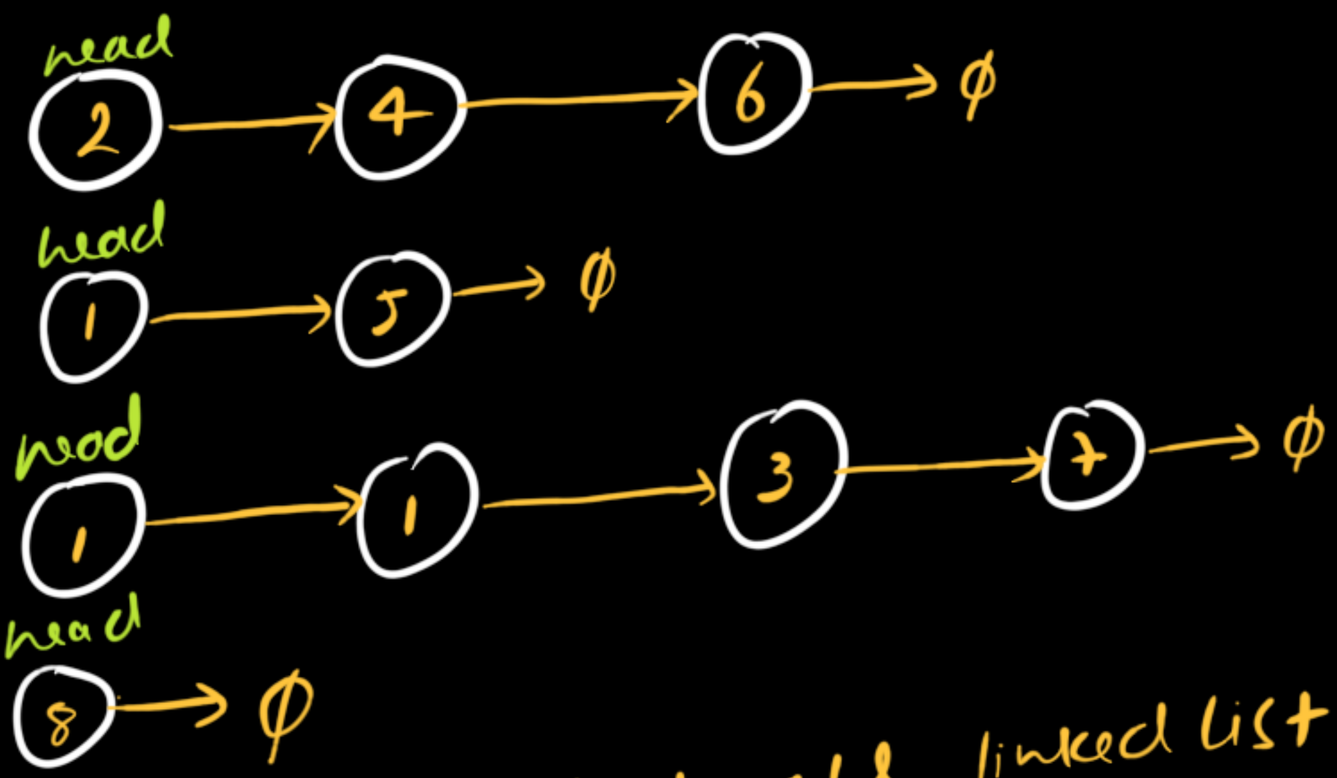
Tc: $O(N \times K) + O(N \times K \log(N \times K)) + O(N \times K)$

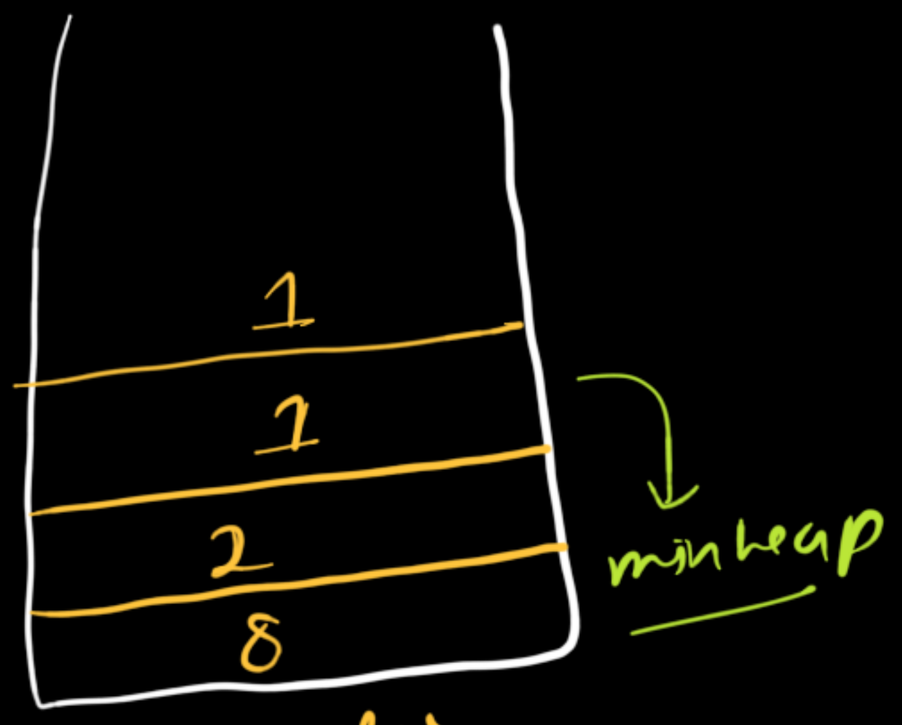
$N \times K$: Each LL have length K; N: no of LL given

Sc: $O(N \times m)$

↳ for storing the result

→ Approach-2





PQ<Node>

(1) Insert the head of all linked list in PQ
(2) Create a dummy node (-1) → ϕ
(3) pop of the minimum node from priority queue i.e 1 dummy
(4) Connect the popped off node to the dummy.next
(5) check if the popped node next != null
↳ if true → insert next to the PQ
↳ keep doing this until PQ is empty.

XXII.java

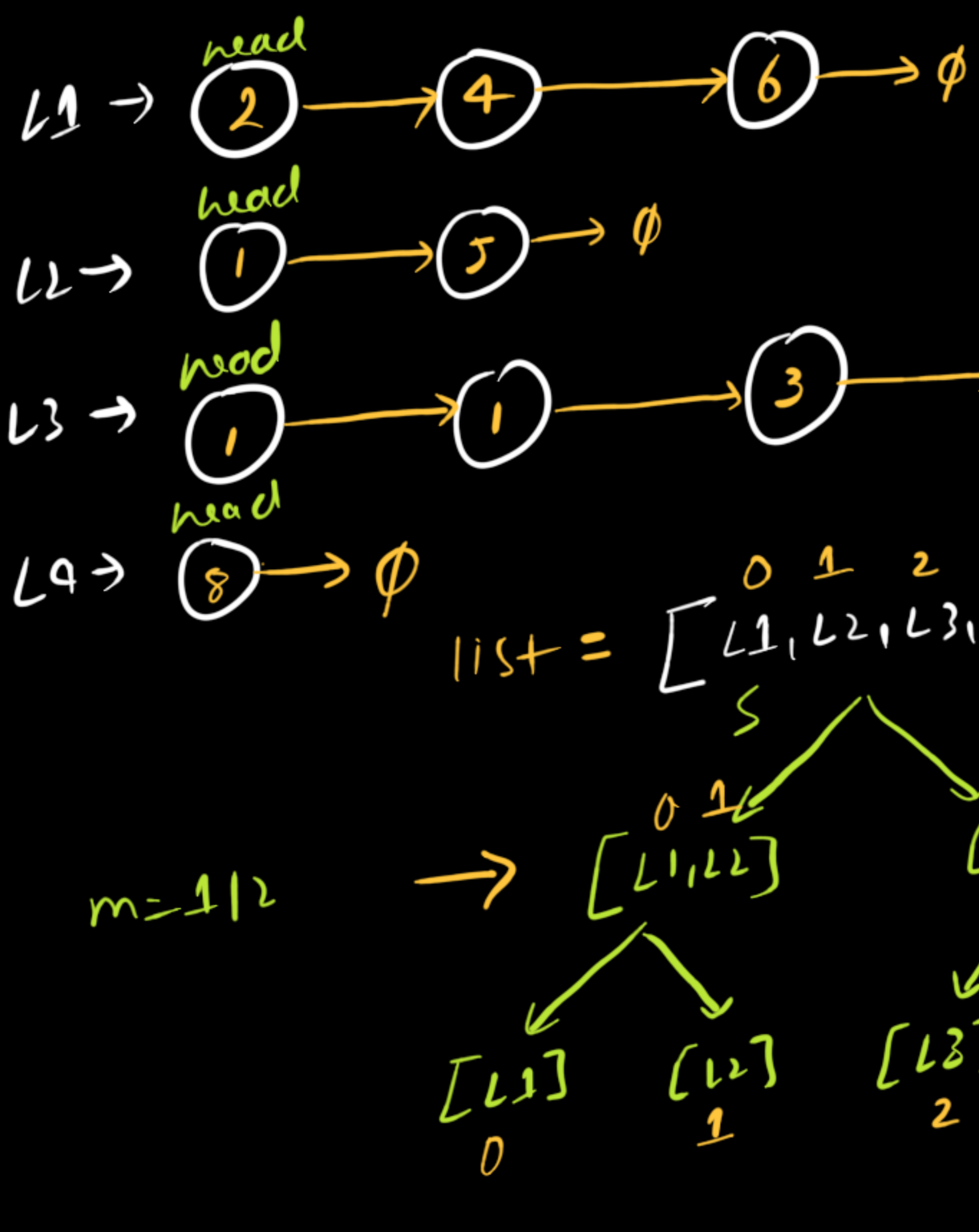
```java
public static @Nullable LLNode<Integer> mergeKLists(LLNode<Integer> @NotNull [] lists) {

    if (lists.length == 0) return null;

    PriorityQueue<LLNode<Integer>> pq = new PriorityQueue<>(
            Comparator.comparingInt( LLNode<Integer> a -> a.value)
    );

    for (LLNode<Integer> head : lists) {
        if (head != null) {
            pq.offer(head);
        }
    }

    LLNode<Integer> dummy = new LLNode<Integer>( value: -1);
    dummy.next = null;
    LLNode<Integer> temp = dummy;

    while (!pq.isEmpty()) {
        LLNode<Integer> min = pq.poll();
        temp.next = min;
        temp = temp.next;

        if (min.next != null) {
            pq.offer(min.next);
        }
    }
    return dummy.next;

}
```

→ total number of Nodes

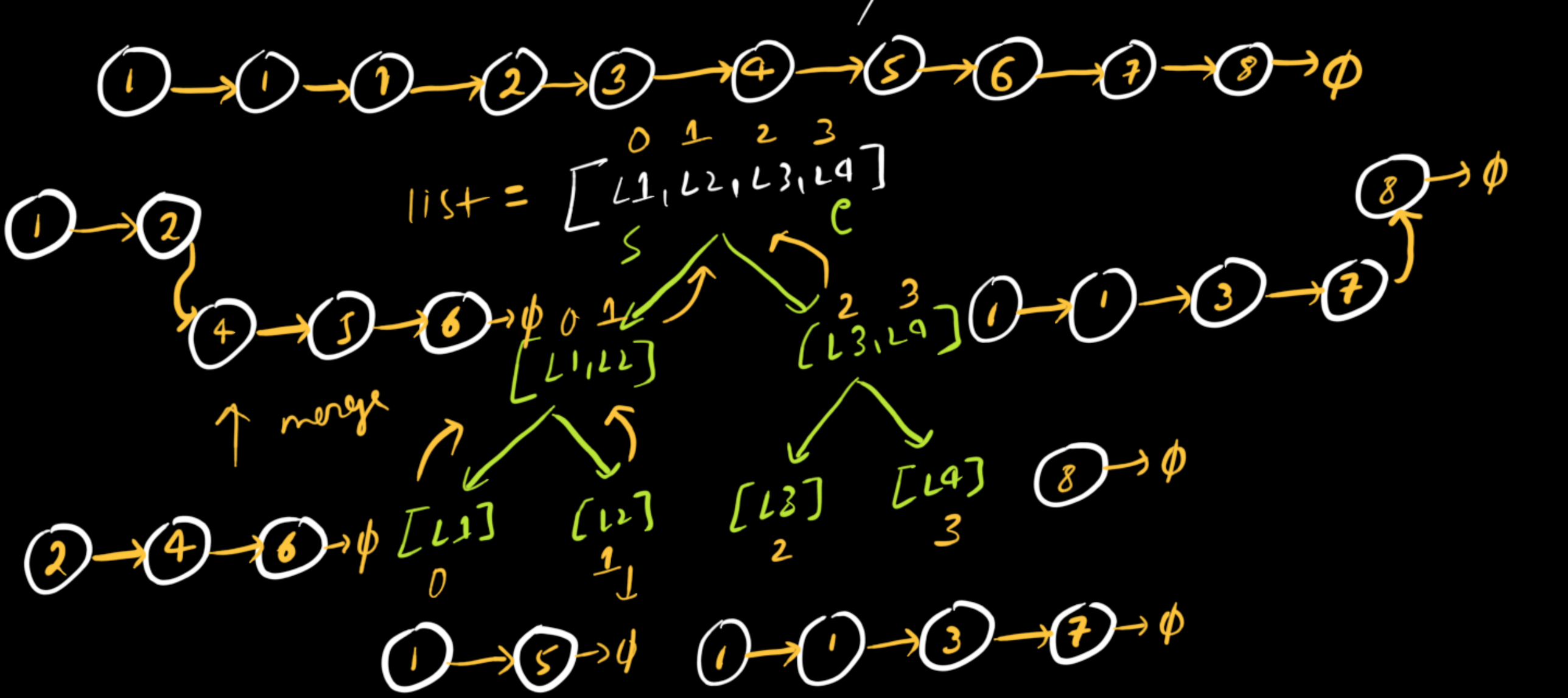
TC: N log(K)

SC: O(K)

→ Approach-3



→ Final K merged list.



XXII.java

```java
public static @Nullable LLNode<Integer> mergeKList(LLNode<Integer> @NotNull [] lists) {
    if (lists.length == 0) {
        return null;
    }

    return mergeK(lists, s: 0, e: lists.length - 1);
}

private static LLNode<Integer> mergeK(LLNode<Integer>[] lists, int s, int e) {  3 usage
    if (s == e) return lists[s];
    int mid = (s + e) / 2;

    LLNode<Integer> left = mergeK(lists, s, mid);
    LLNode<Integer> right = mergeK(lists, s: mid + 1, e);

    return merge(left, right);
}

private static LLNode<Integer> merge(LLNode<Integer> list1, LLNode<Integer> list2) {
    LLNode<Integer> newList = new LLNode<~>( value: 0);
    LLNode<Integer> temp = newList;

    while (list1 != null && list2 != null) {
        if (list1.value >= list2.value) {
            temp.next = list2;
            list2 = list2.next;
        } else {
            temp.next = list1;
            list1 = list1.next;
        }
        temp = temp.next;
    }

    temp.next = list1 != null ? list1 : list2;
    return newList.next;
}
```

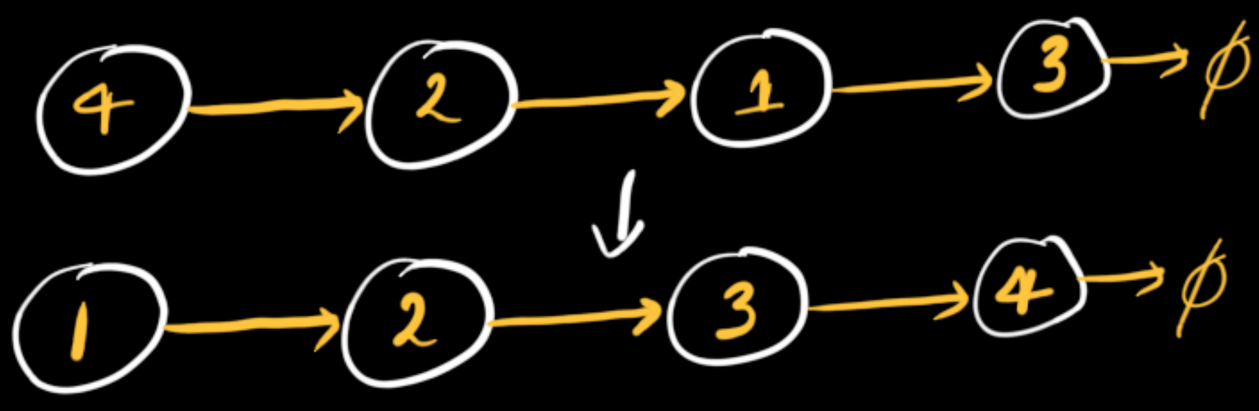
TC: O(N log K)

SC: O(N)

↓

recursive stack space.

→ Sort - List



→ Approach-1 (Brute)

↳ Apply Bubble- sort

Tc: $O(n^2)$

Sc: $O(1)$

```java
public static LLNode<Integer> sortList(LLNode<Integer> head) {
    if (head == null || head.next == null) return head;

    LLNode<Integer> temp = head;

    while (temp != null){
        LLNode<Integer> secTemp = temp.next;

        while (secTemp != null){
            if (temp.value > secTemp.value){
                Integer t = temp.value;
                temp.value = secTemp.value;
                secTemp.value = t;
            }
            secTemp = secTemp.next;
        }

        temp = temp.next;
    }
    return head;
}
```

→ Approach-2 (Better)

① Traverse the list, put all element into an ArrayList

② Sort the list & override the value of the original list;

```java
public static LLNode<Integer> sortListBetter(LLNode<Integer> head) {
    if (head == null || head.next == null) return head;

    LLNode<Integer> temp = head;
    List<Integer> l = new ArrayList<>();

    while (temp != null){
        l.add(temp.value);
        temp = temp.next;
    }

    Collections.sort(l);
    temp = head;
    for (Integer e : l){
        temp.value = e;
        temp = temp.next;
    }
    return head;
}
```

O(N)

O(N log N)

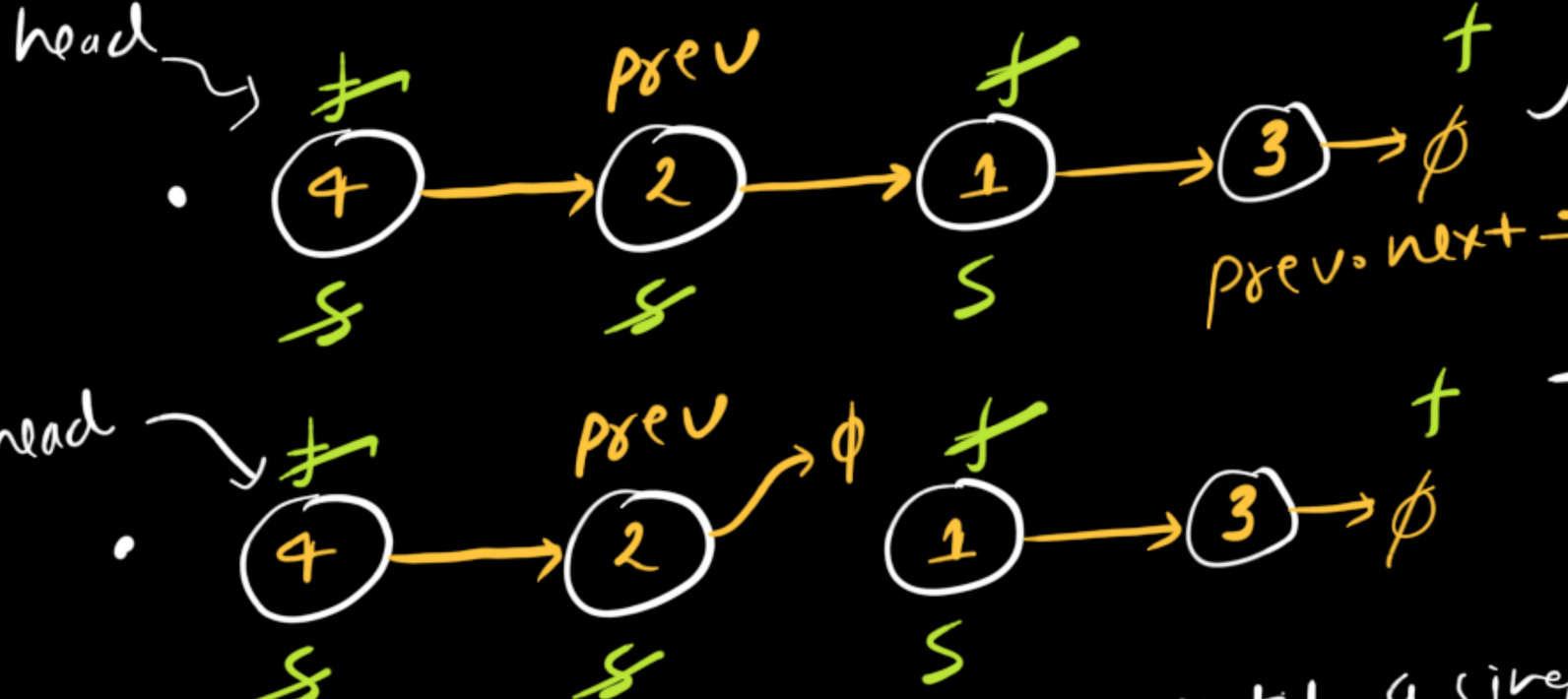
O(N)

Tc: $O(2N) + N \log N$
$\approx O(N \log N)$

Sc: $O(N)$

→ Approach-3 (merge-sort)

- How do we divide the array into two equal part
  - ↳ Tortoise & hare Algorithm



- recursively keep doing this until a single node remains, that will be our base case.
  - ↳ when single node remains head.next == null
  - Base-case

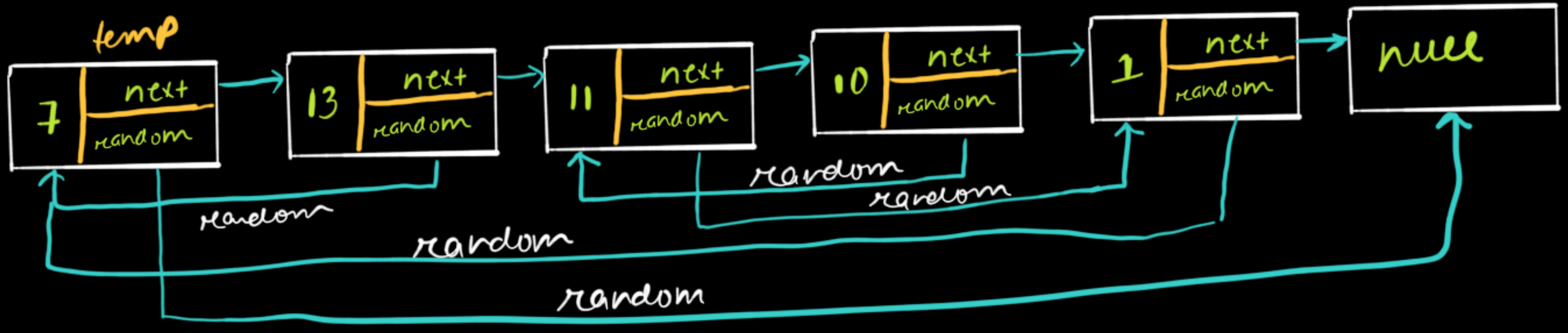
T.C: O(nlogn)

S.C: O(N) ↓ stackspace.

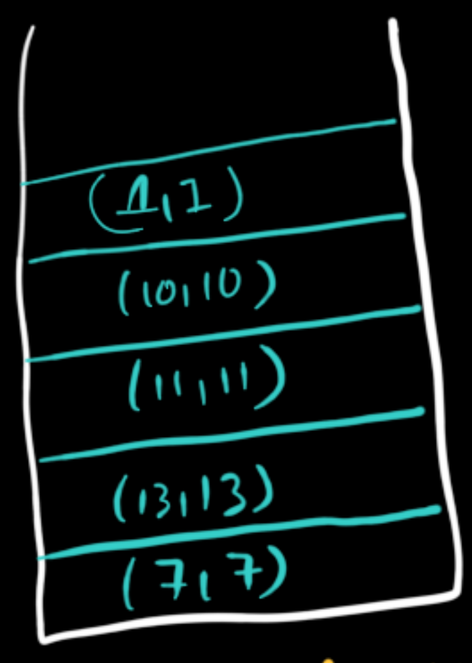
```java
PartXXIII.java
public static LLNode<Integer> sortListOpt( @NotNull LLNode<Integer> head) {  3 usages
    if (head.next == null) return head;

    LLNode<Integer> f = head;
    LLNode<Integer> s = head;
    LLNode<Integer> prev = null;

    while (f != null && f.next != null){
        prev  = s;
        s = s.next;
        f = f.next.next;
    }
    prev.next = null;

    LLNode<Integer> left = sortListOpt(head);
    LLNode<Integer> right = sortListOpt(s);


    return merge(left , right);
}
private static LLNode<Integer> merge(LLNode<Integer> list1, LLNode<Integer> list2) {
    LLNode<Integer> newList = new LLNode<~>( value: 0);
    LLNode<Integer> temp = newList;

    while (list1 != null && list2 != null) {
        if (list1.value >= list2.value) {
            temp.next = list2;
            list2 = list2.next;
        } else {
            temp.next = list1;
            list1 = list1.next;
        }
        temp = temp.next;
    }

    temp.next = list1 != null ? list1 : list2;
    return newList.next;
}
```

→ Copy list with Random pointer

↳ Next: points to the next node
random: points to any random node or null



→ Approach-1



Map<Node, Node>

Original Node

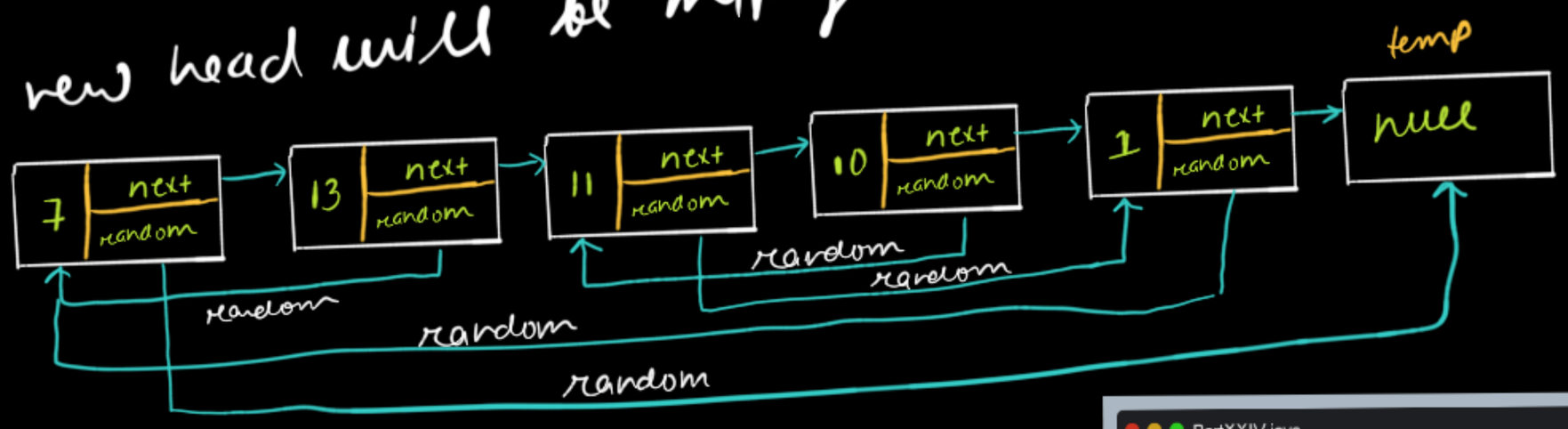
Copied Node

① place temp, push each node into the hashmap.

② Again place temp over the head, get the copied node using org-node as key.

```
Node org = temp;
Node cpy = map.get(temp);
cpy.next = map.get(org.next);
cpy.random = map.get(org.random);
```

③ The new head will be map.get(head);



TC: O(2N)

SC: O(N) + O(N)

For map

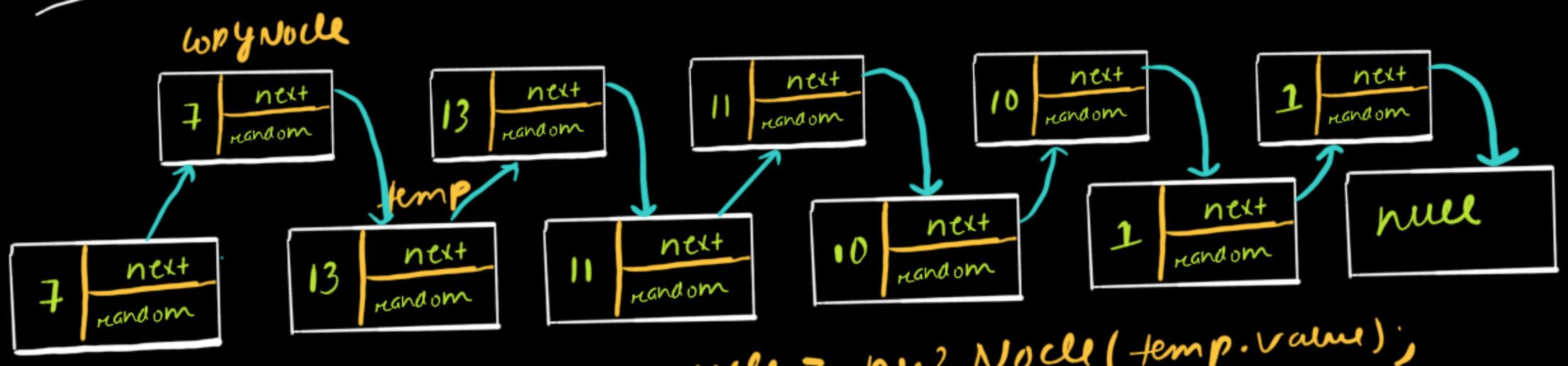
For creating the new list.

PartXXIV.java

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

    Map<Node, Node> map = new HashMap<>();
    Node temp = head;

    while (temp != null) {                    // O(N)
        map.put(temp, new Node(temp.val));
        temp = temp.next;
    }

    temp = head;
    while (temp != null) {                    // O(N)
        Node copy = map.get(temp);
        copy.next = map.get(temp.next);
        copy.random = map.get(temp.random);
        temp = temp.next;
    }

    return map.get(head);
}
```

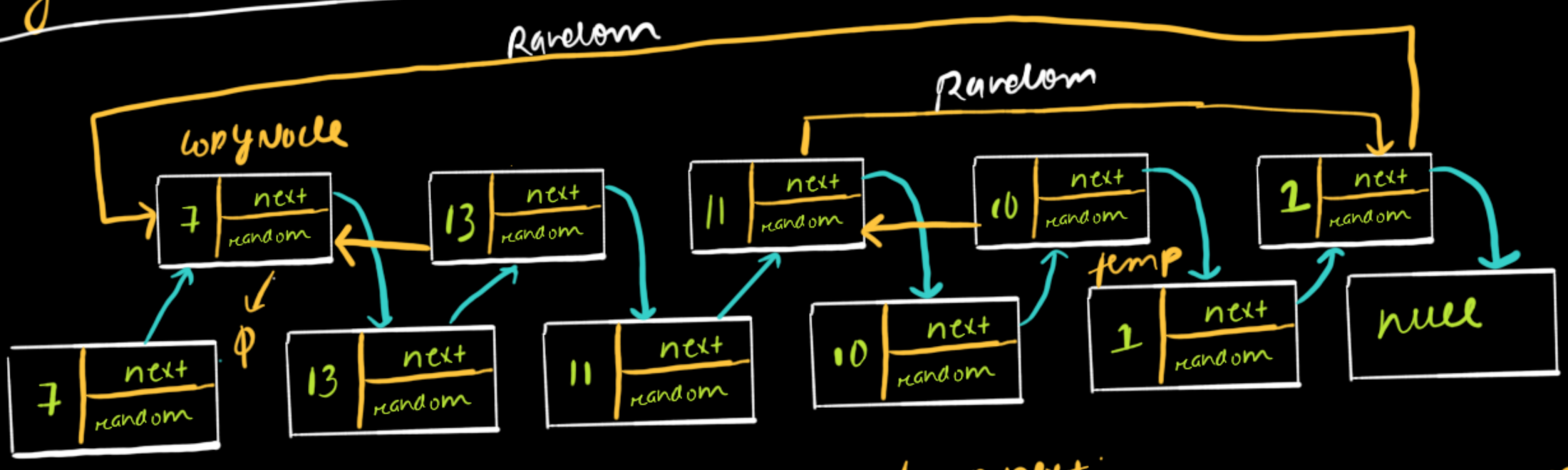
→ Approach-2

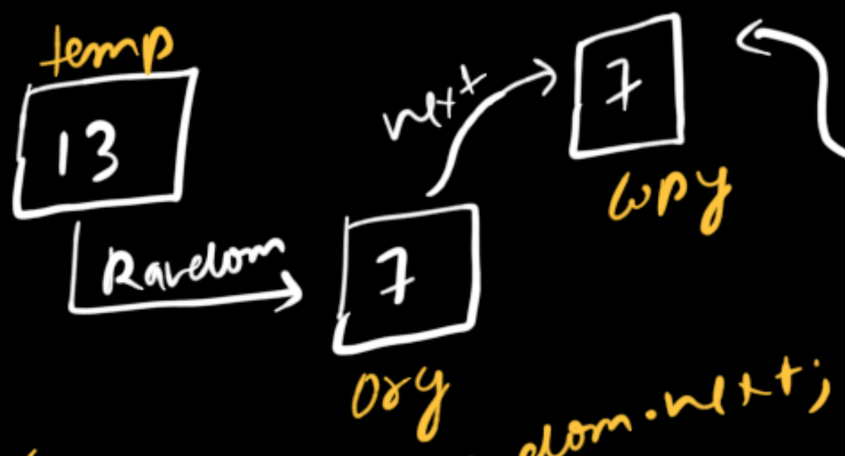
- Insert copy-Node in between the current node & current.next node.




```
Node copyNode = new Node(temp.value);
copyNode.next = temp.next;
temp.next = copyNode;
temp = temp.next.next;
```

- Again point temp to head & Connect random pointer

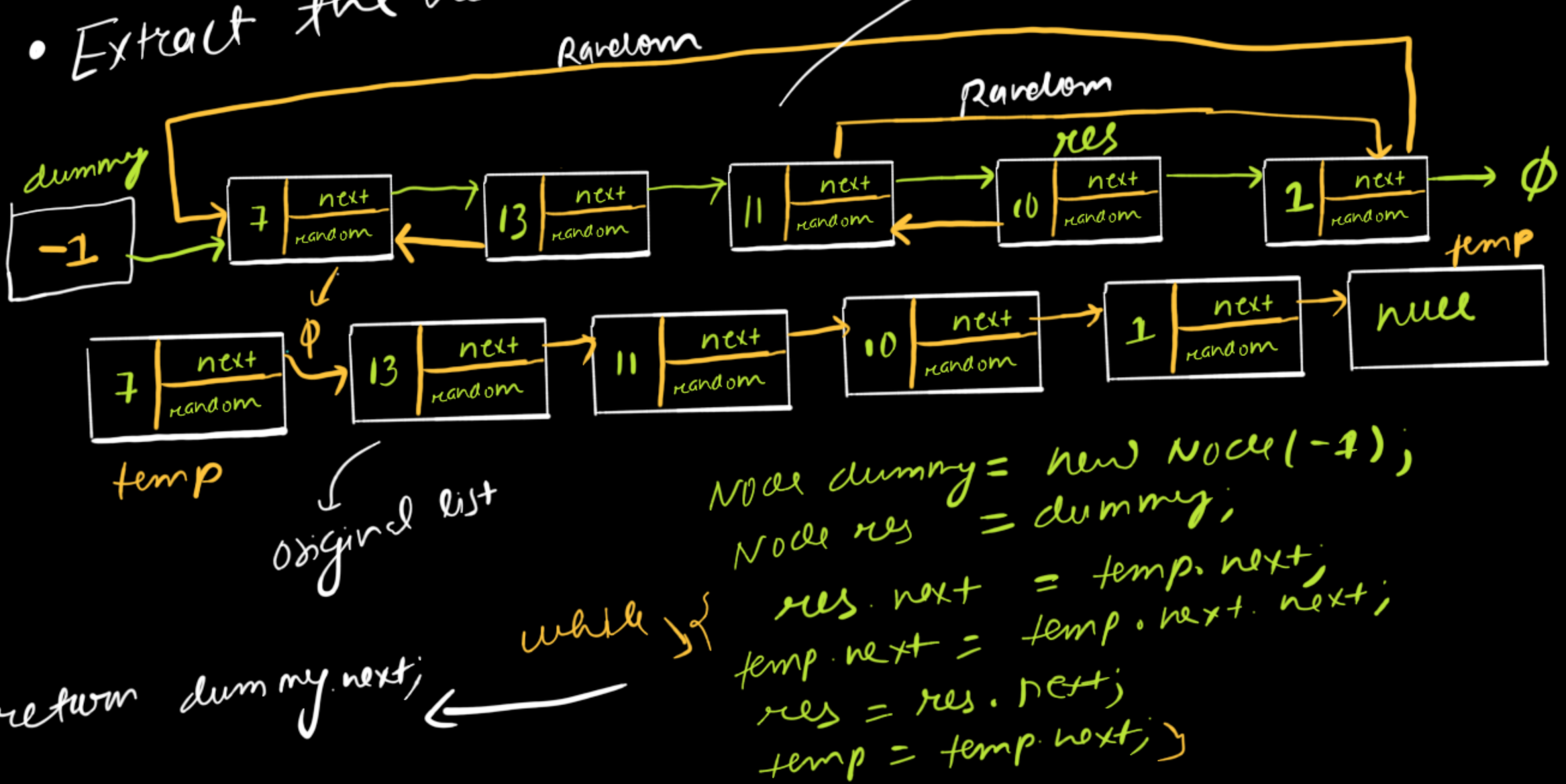





```
Node copy = temp.next;
copy.random = temp.random != null ? temp.random.
                                    next : null;
temp = temp.next.next;
```

So ⇝ temp.random.next;

- Extract the created copy Linked List → Copyed list




```
Node dummy = new Node(-1);
Node res = dummy;
while(){
res.next = temp.next;
temp.next = temp.next.next;
res = res.next;
temp = temp.next;}
```

return dummy.next;

PartXXIV.java

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

    Node temp = head;

    while (temp != null) {
        Node copy = new Node(temp.val);
        copy.next = temp.next;
        temp.next = copy;
        temp = copy.next;
    }

    temp = head;
    while (temp != null) {
        if (temp.random != null) {
            temp.next.random = temp.random.next;
        }
        temp = temp.next.next;
    }

    Node dummy = new Node( val: 0);
    Node res = dummy;
    temp = head;

    while (temp != null) {
        res.next = temp.next;
        res = res.next;

        temp.next = temp.next.next;
        temp = temp.next;
    }

    return dummy.next;
}
```

O(N)

O(N)

O(N)

TC: O(3N)

SC: O(N)

↓

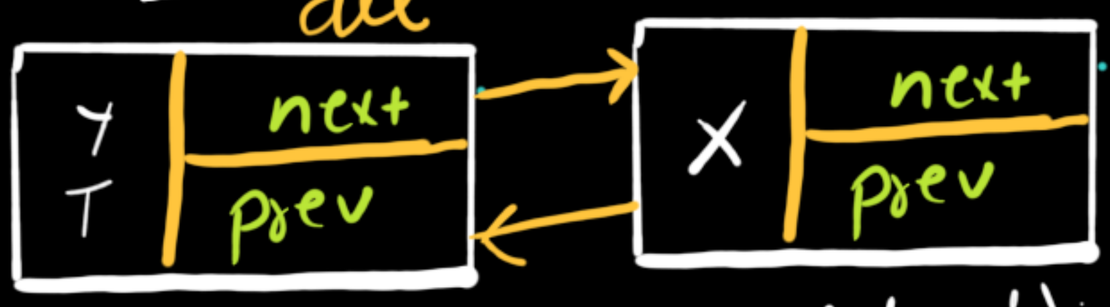
for creating the new copyed Linked List

→ Design Browser History

↳
- BrowserHistory(homepage); → visit homepage
- visit(url); → visit the url
- back(steps); → go back steps time in history
- forward(steps); → go forward steps time in history.

Doubly linked List

→ visit()




```
newNode = new Node(url);
dll.next = newNode;
newNode.prev = dll;
dll = dll.next;
```

TC: O(2steps)

SC: O(N)

[No. of visit() called]

can call adds one node.

BrowserHistory.java

```java
public class BrowserHistory {  no usages  new *
    private Node dll;  12 usages
    @Contract(pure = true)
    public BrowserHistory(String homepage) {  no u
        this.dll = new Node(homepage);
    }

    public void visit(String url) {  no usages  new
        Node node = new Node(url);
        dll.next = node;
        node.prev = dll;
        dll = node;
    }

    public String back(int steps) {  no usages  new

        while (steps > 0){
            if (dll.prev != null) dll = dll.prev;
            else break;
            steps--;
        }

        return dll.url;
    }

    public String forward(int steps) {  no usages

        while (steps > 0){
            if (dll.next != null) dll = dll.next;
            else break;
            steps--;
        }
        return dll.url;

    }
    private class Node {  6 usages  new *
        public String url;  3 usages
        public Node next;  4 usages
        public Node prev;  4 usages

        @Contract(pure = true)
        public Node(String url){  2 usages  new *
            this.url = url;
            this.next = null;
            this.prev = null;
        }
    }
}
```

O(1)

O(steps)

O(steps)