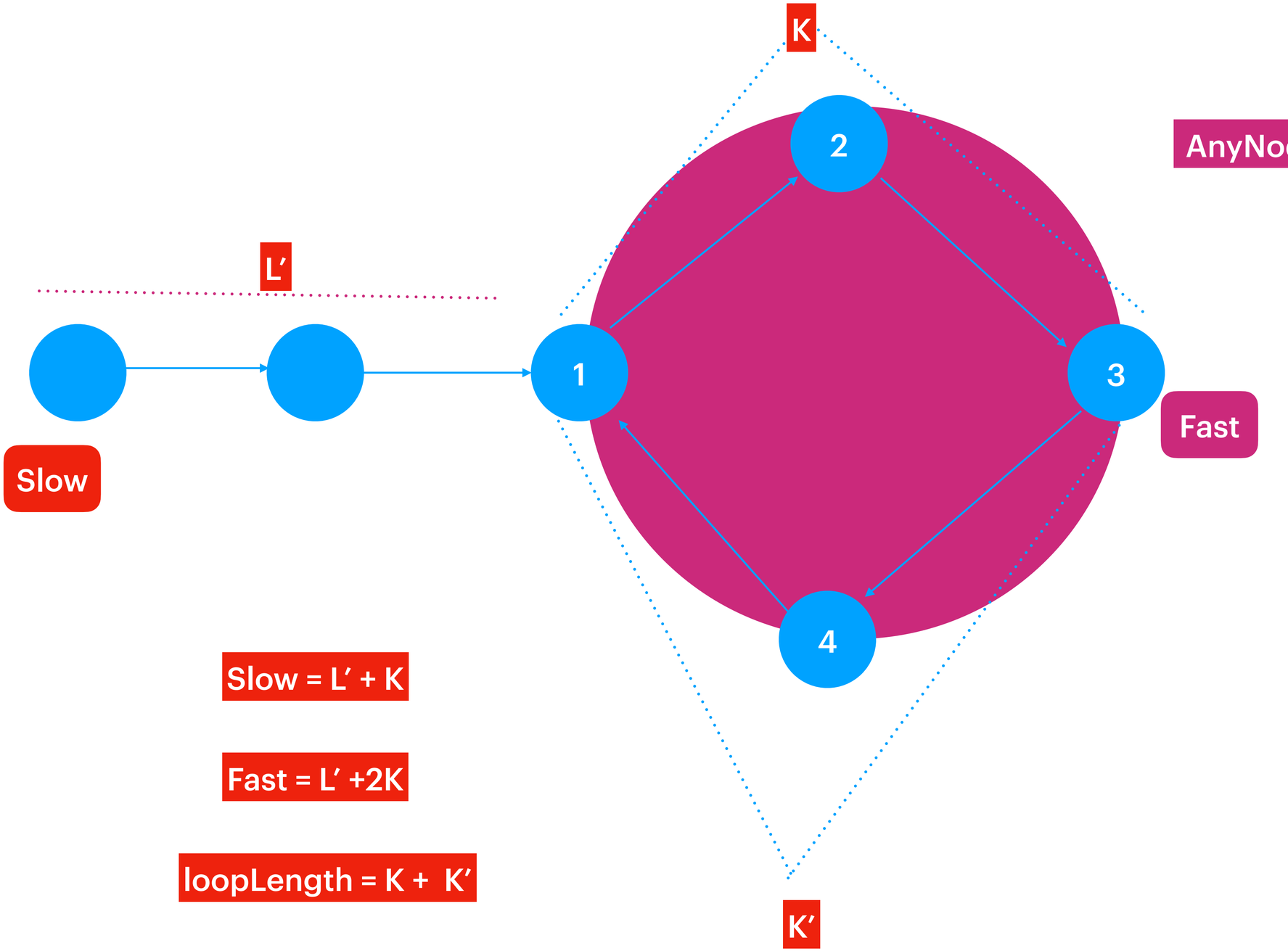


AnyNode = K Iterations

K' = L' ?
K + K' = L
2K + K' = L

K + K + K' = L

L' + K + K + K' = L + L'

SLOW + K + K' = L + L'

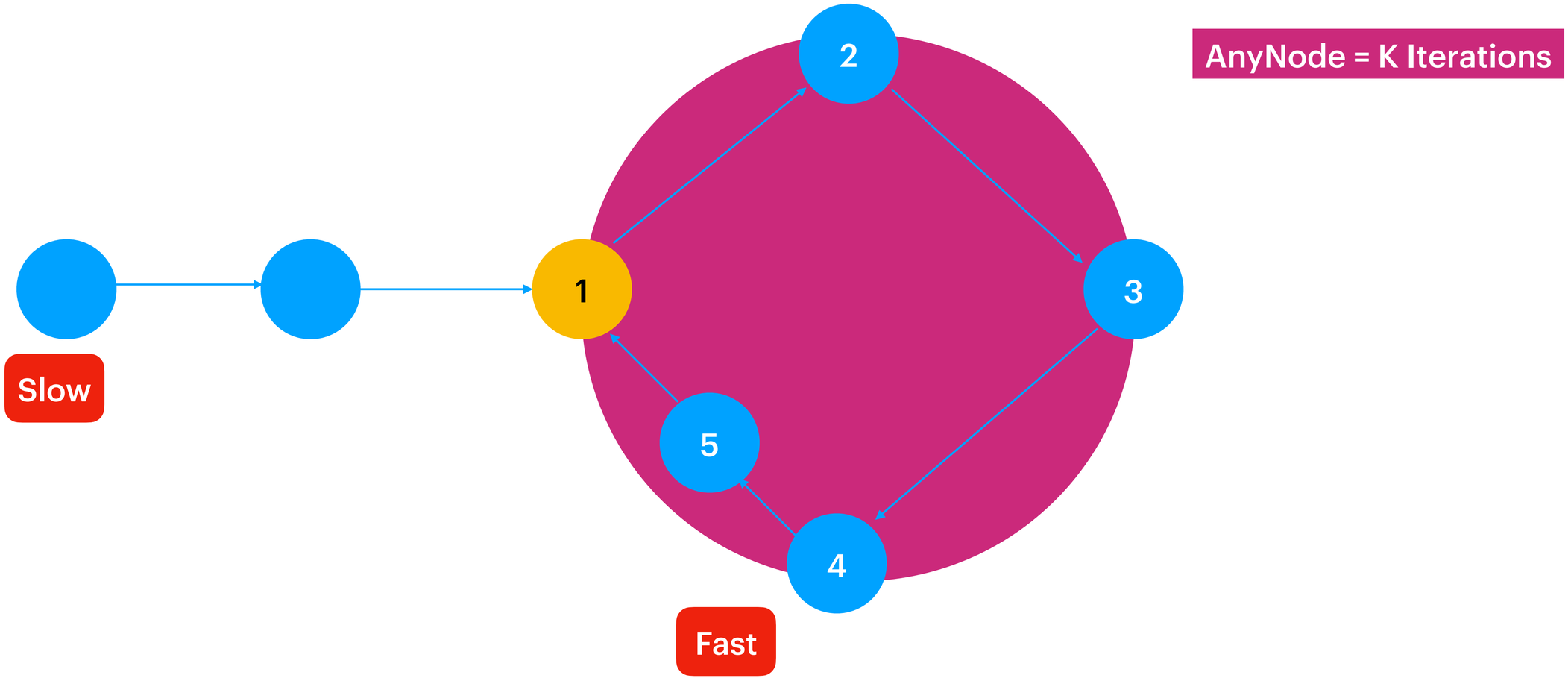
SLOW + L = L + L'

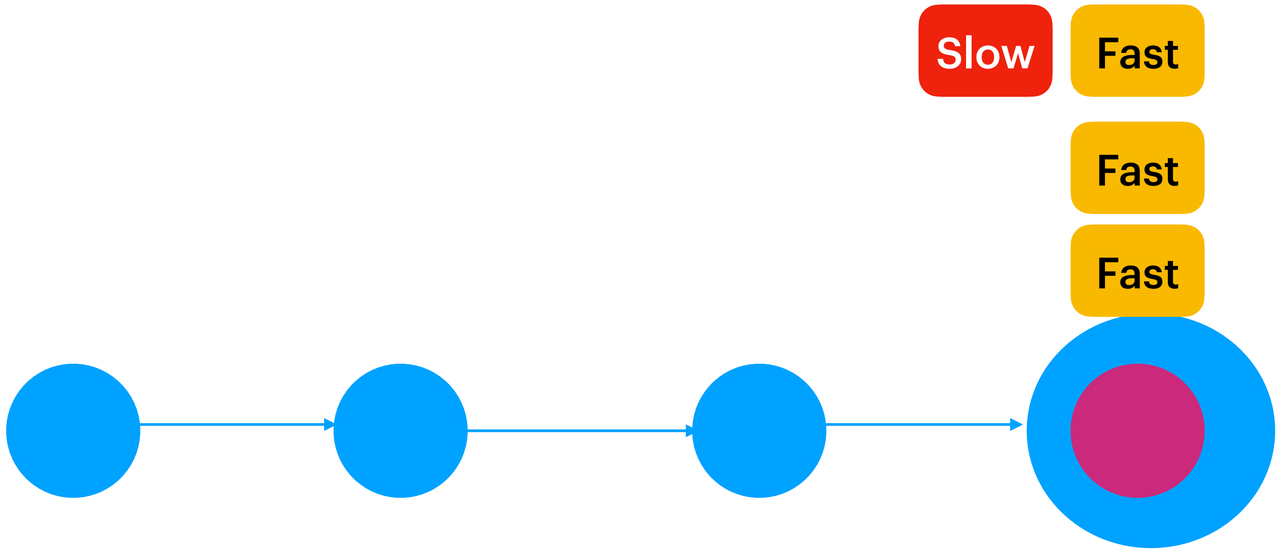
SLOW = L'

Slow = L' + K

Fast = L' +2K

loopLength = K + K'


AnyNode = K Iterations
2
1
3
5
4
Slow
Fast



Slow = L' + K

Fast = L' +2K

loopLength = K + K'

K' = L' ?

2K + K' = L
2K + K' = L

K+ K' = 0
K + 0 = 0
K = 0
5K = 0
10K = 0

K + K + K' = L

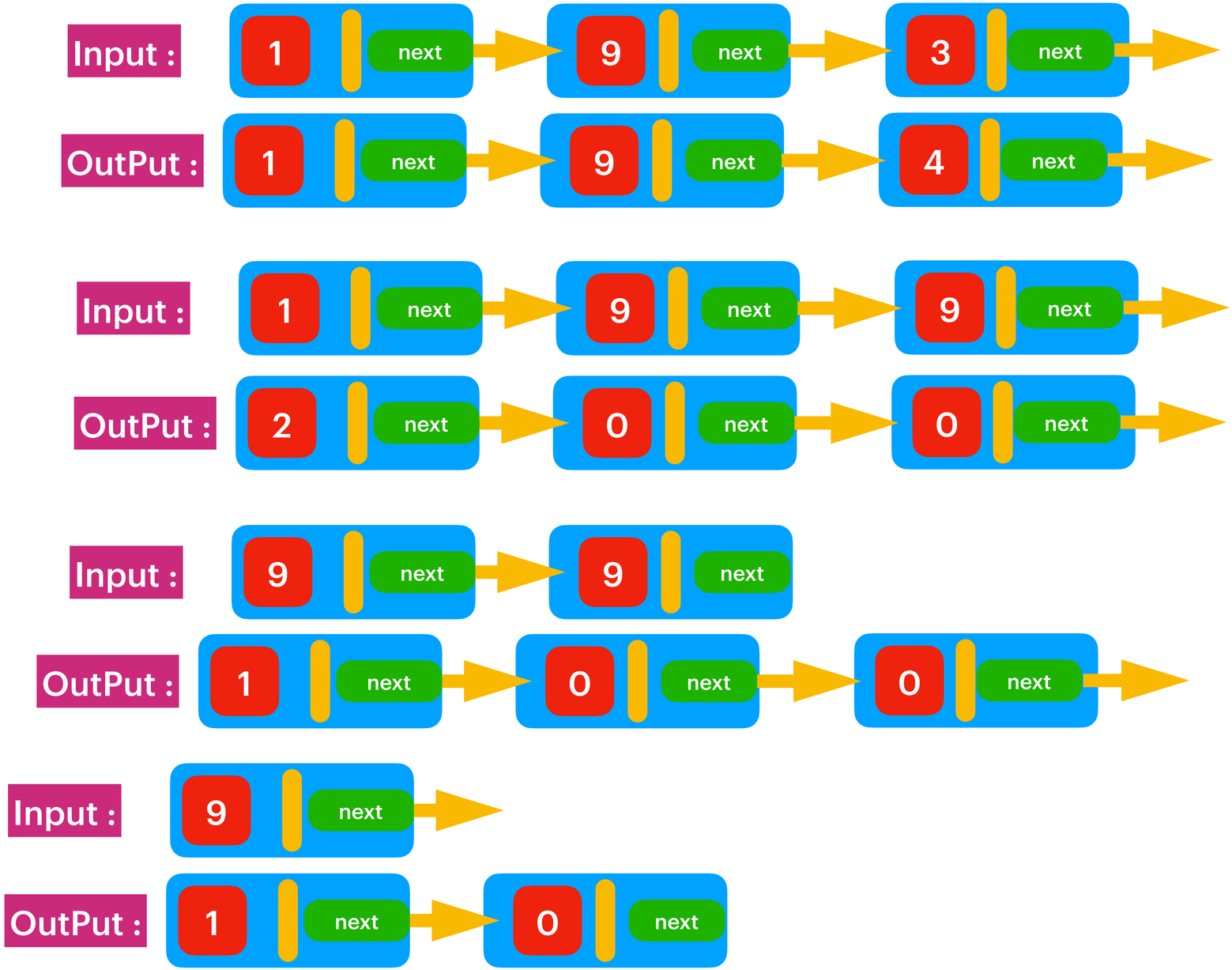
L' + K + K + K' = L + L'

SLOW + K + K' = L + L'

SLOW + L = L + L'

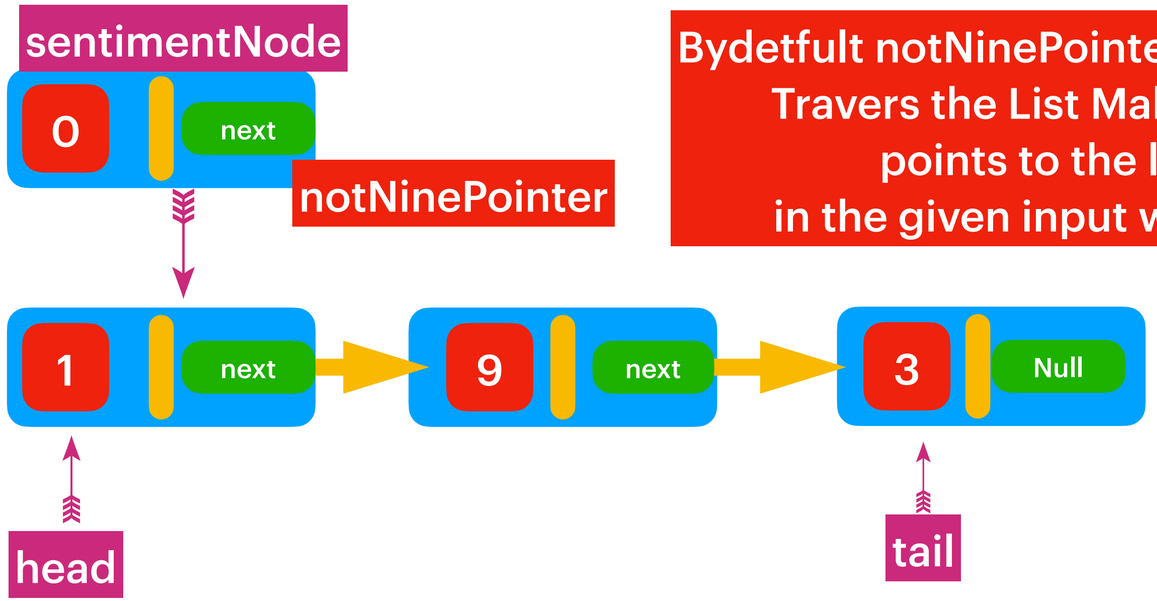
SLOW = L'

Plus One to List



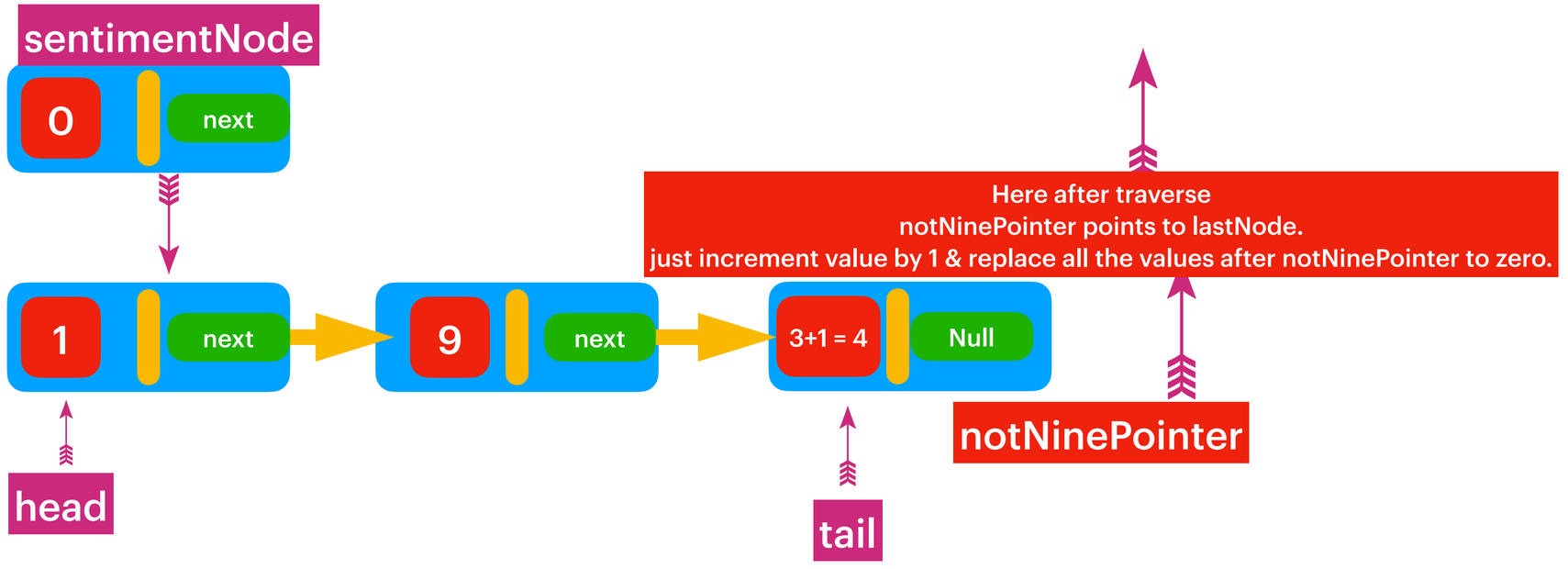
Plus One to List

Take sentimentalNode, which holds the value of '0' , mark sentimentalNode next to head.



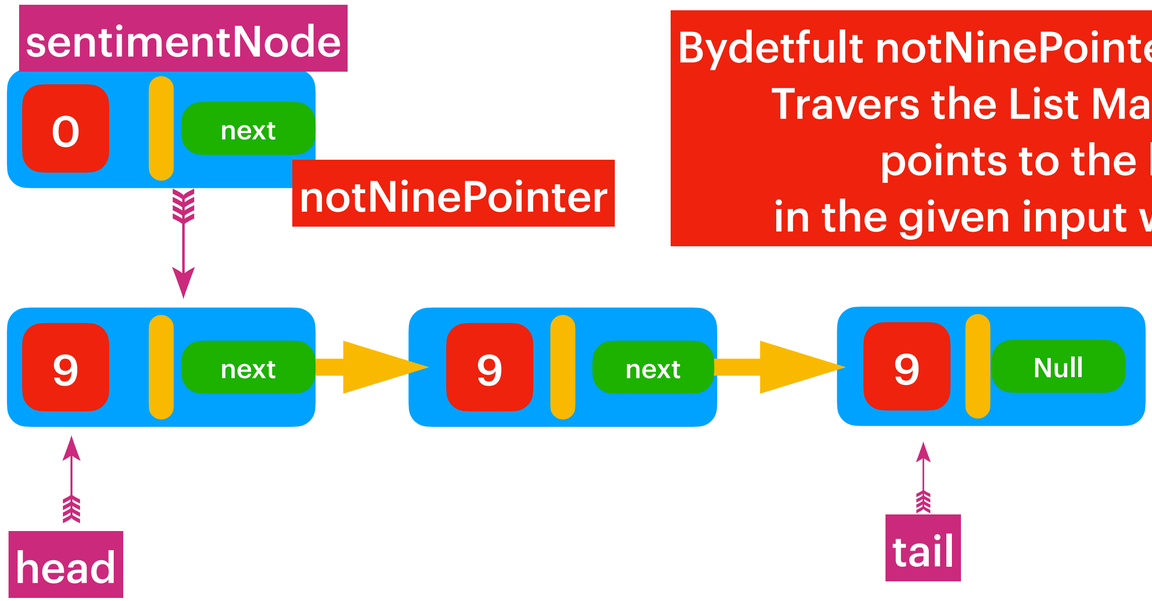
Bydetfult notNinePointer points to sentiment Node,
Travers the List Make sure, notNinePointer
points to the last possible Node
in the given input which is not equals to 9.

Key Point : while returning check
if the sentimentNode value is zero or 1,
If it is zero then return sentimentNode.next
else return sentimentNode.



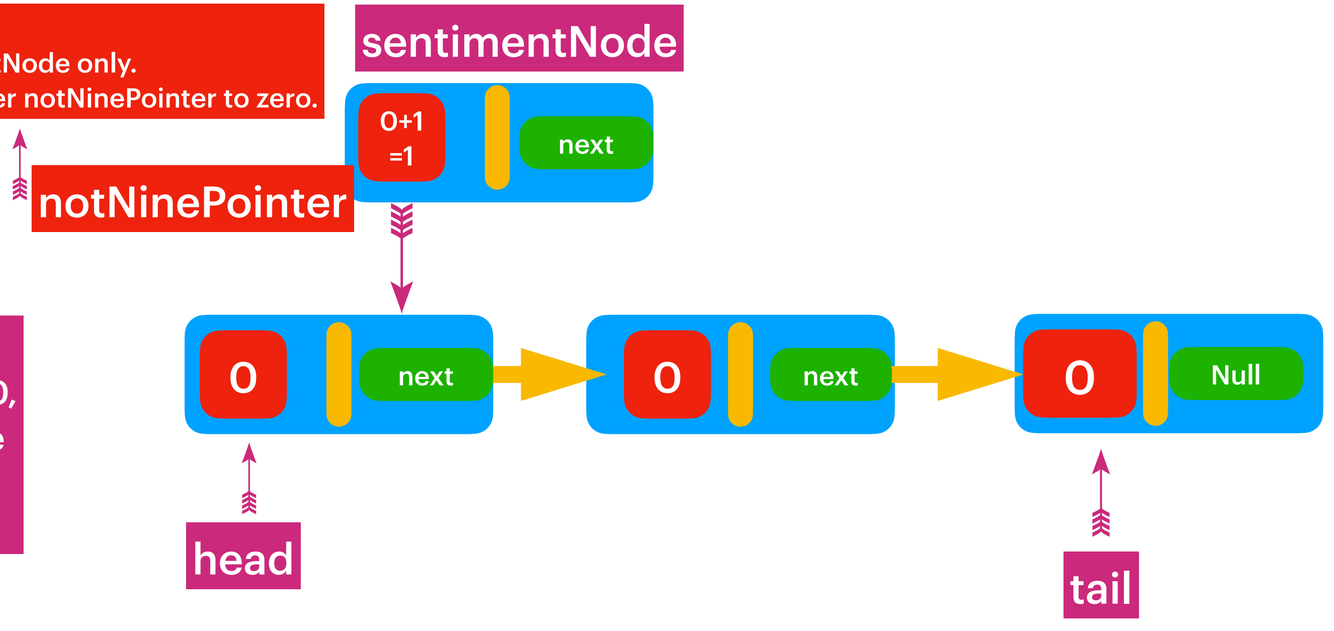
Plus One to List

Take sentimentalNode, which holds the value of '0' , mark sentimentalNode next to head.



Bydetfult notNinePointer points to sentiment Node,
Travers the List Make sure, notNinePointer
points to the last possible Node
in the given input which is not equals to 9.

Here after traverse also
notNinePointer points to sentimentNode only.
just increment by one & replace all the values after notNinePointer to zero.



Key Point : while returning check
if the sentimentNode value is 1 or 0,
If it is 1 then return sentimentNode
else return sentimentNode.next

Adding Numbers :: Inputs are in reverse order !!!

The inputs are in reverse order. => 342 + 465 = 807

Output :

Look at the output in reverse order :

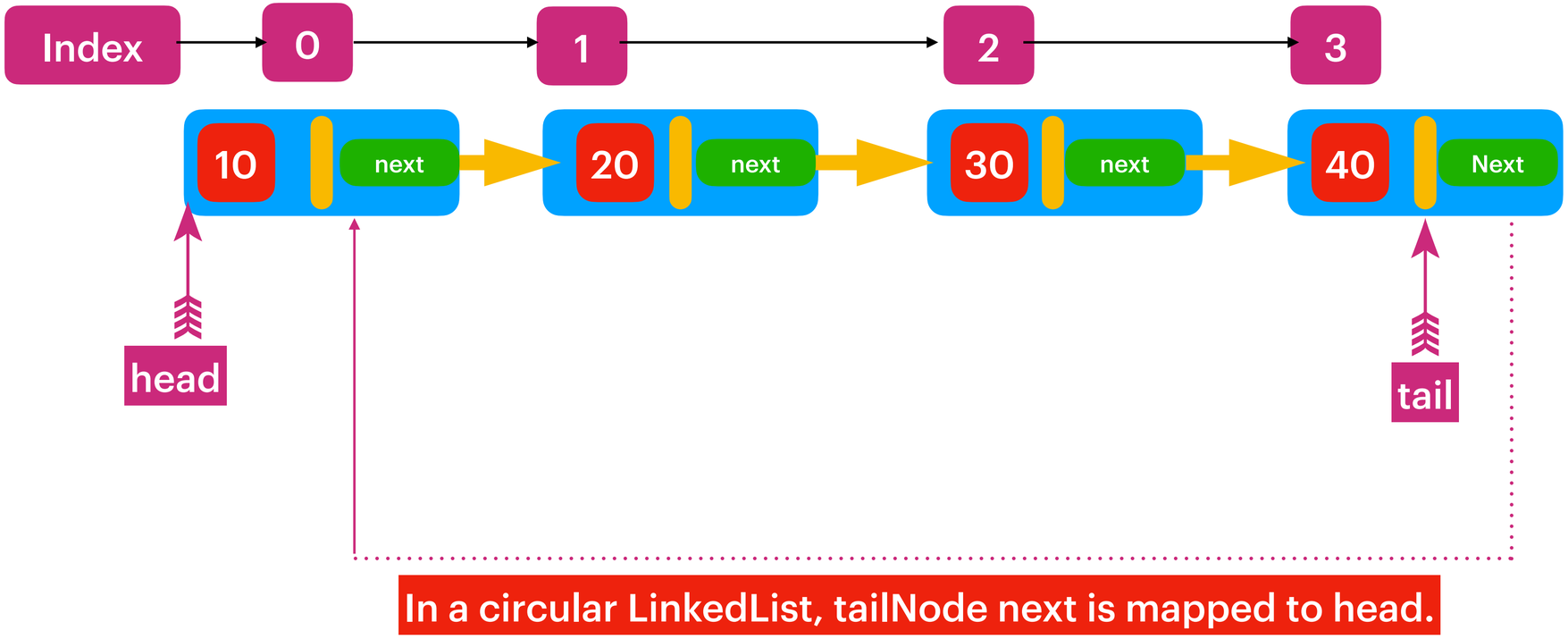
The inputs are in reverse order. => 99 + 1 = 100

Output :

Look at the output in reverse order :

CircularLinkedList

```
class  Node {
int data,
Node next;
}
```

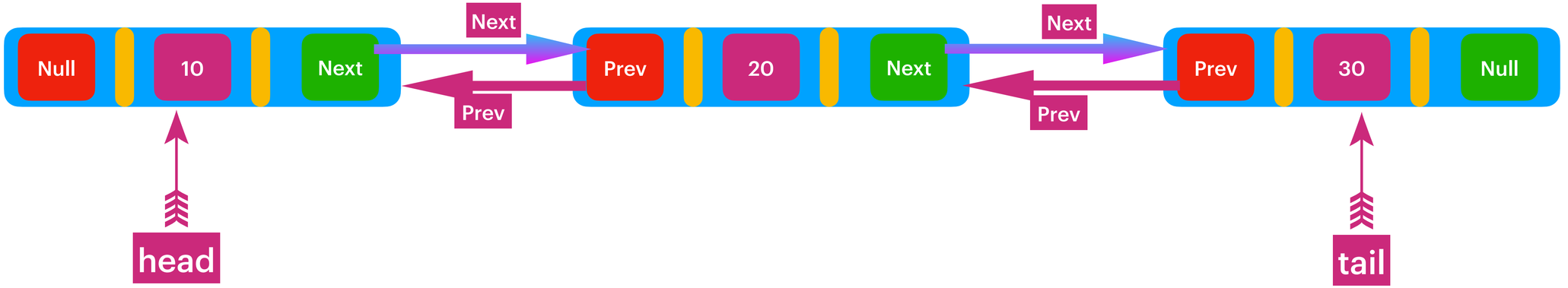


In a circular LinkedList, tailNode next is mapped to head.

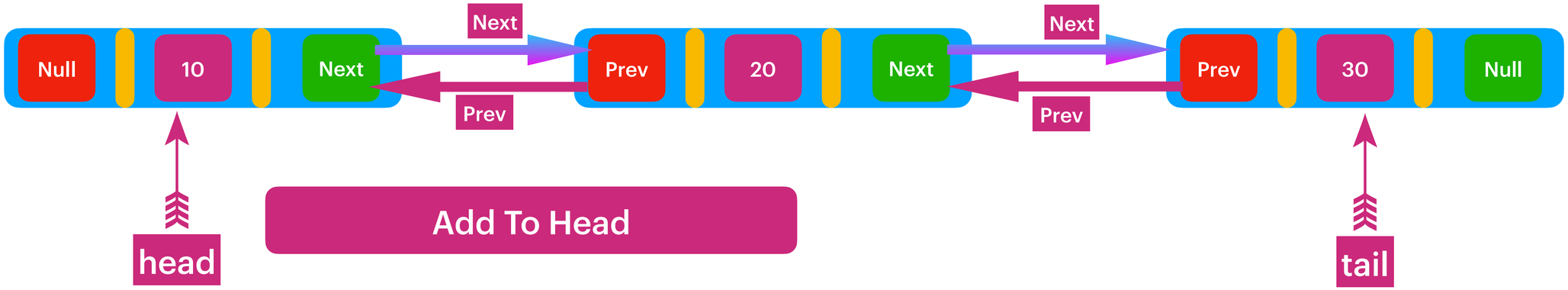
Double Linked List

Double Linked List has the reference of nextNode and its previous Node. So that we can traverse both in forward and reverse directions. Double Linked List simply fees the insert & delete operations.

Node

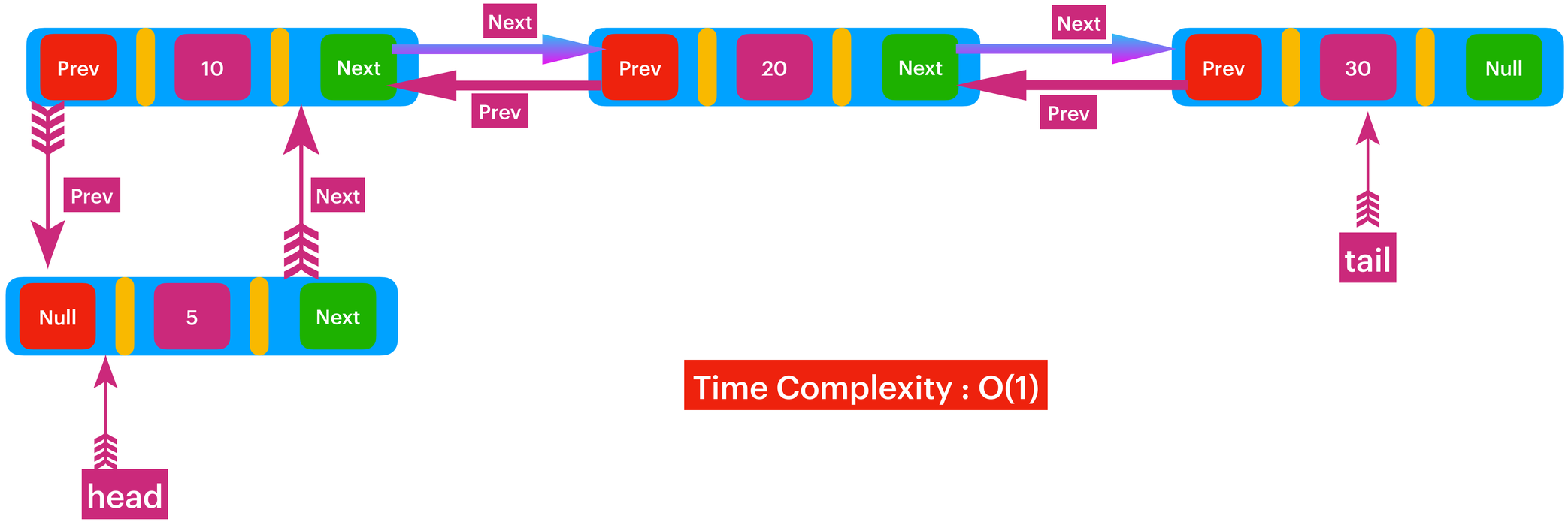



```
class  Node {
int data,
Node next;
Node prev;
}
```

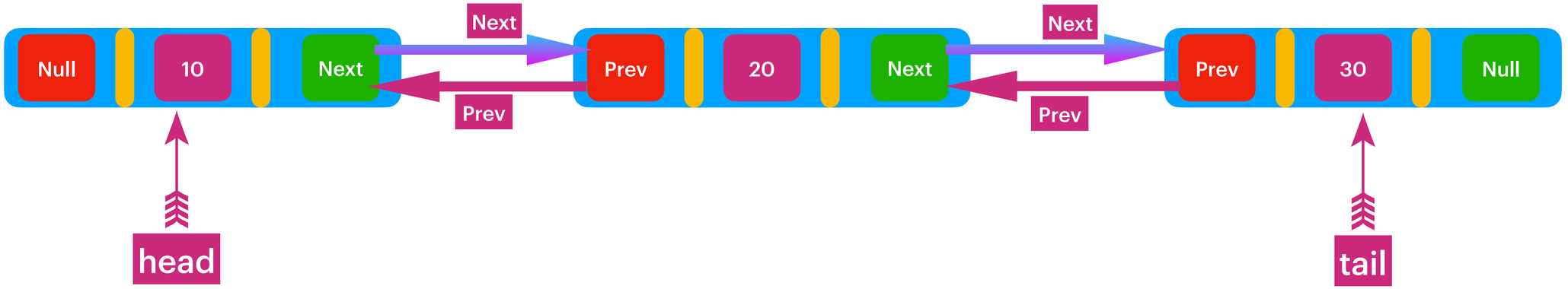


Add To Head

Make newNode next to head & head prev to new Node.
Mark newNode as head.

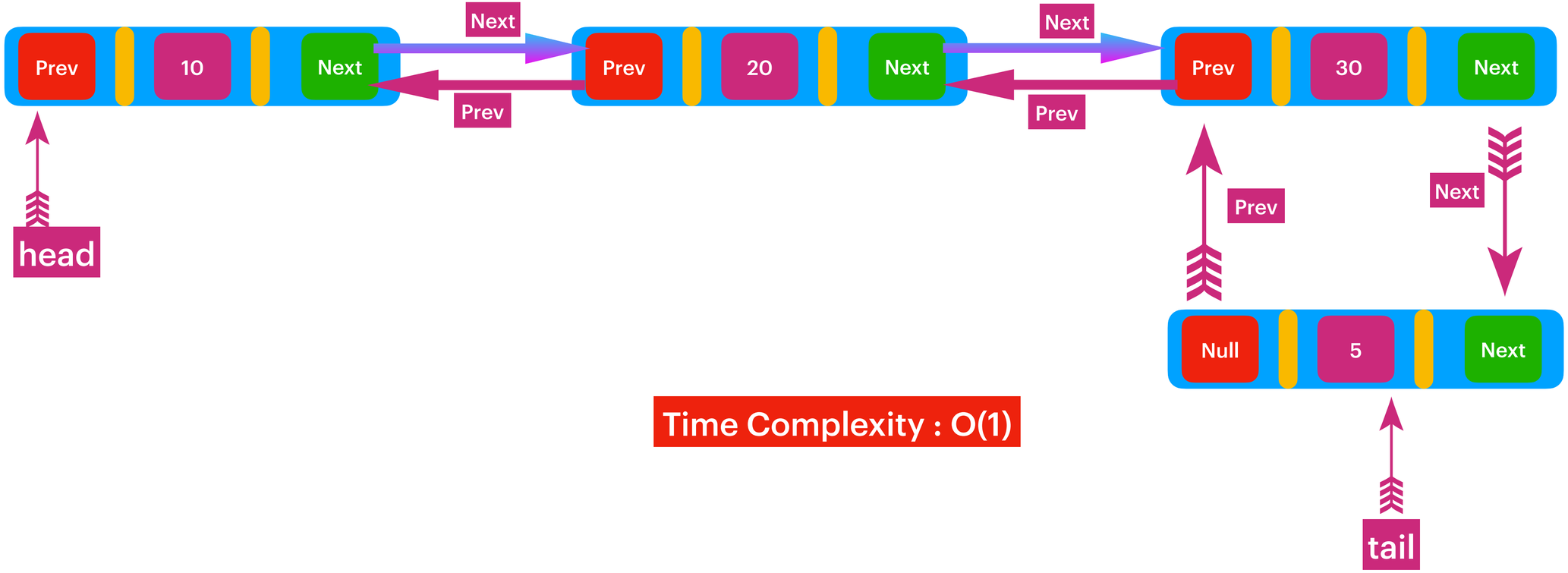


Time Complexity : O(1)

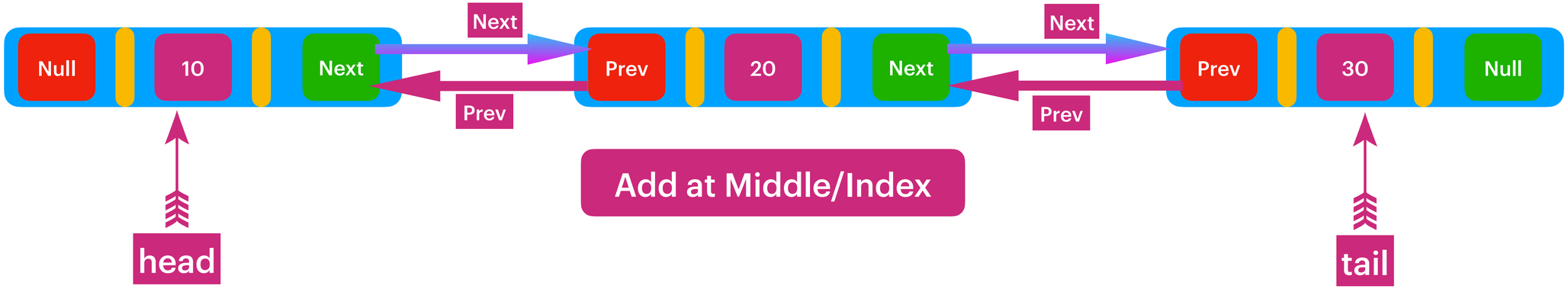


Add To Tail/Last

Make newNode prev to tail & tail next to new Node. Mark newNode as tail.

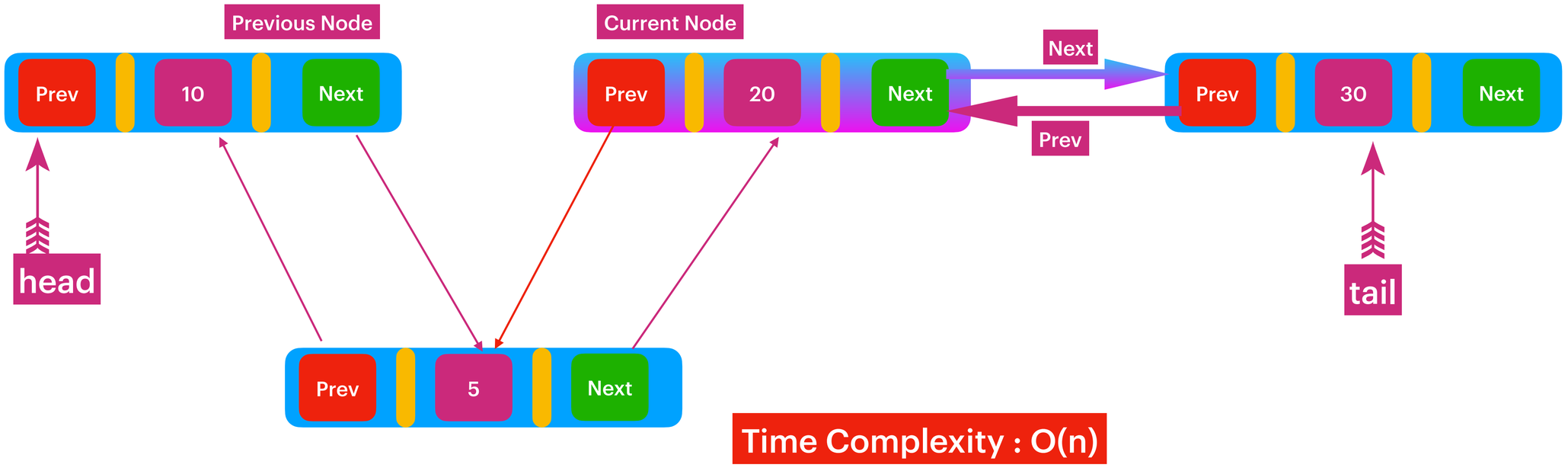


Time Complexity : O(1)

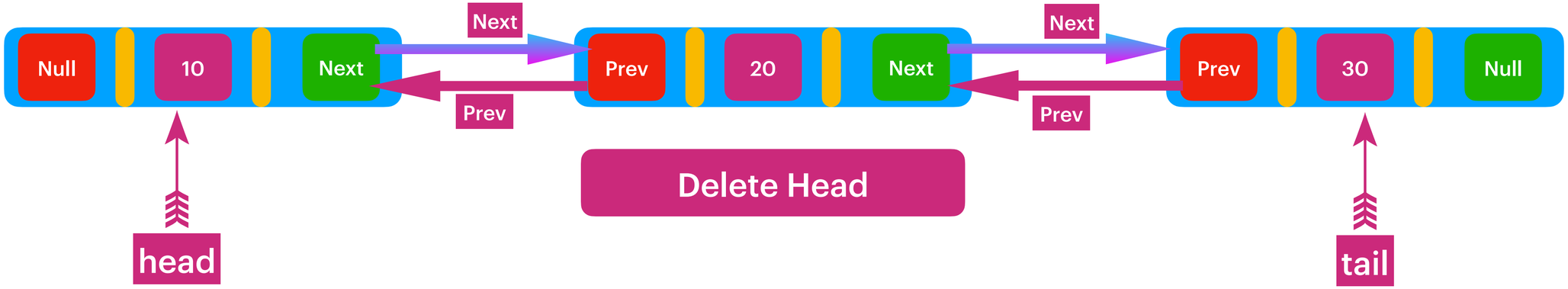


Add at Middle/Index

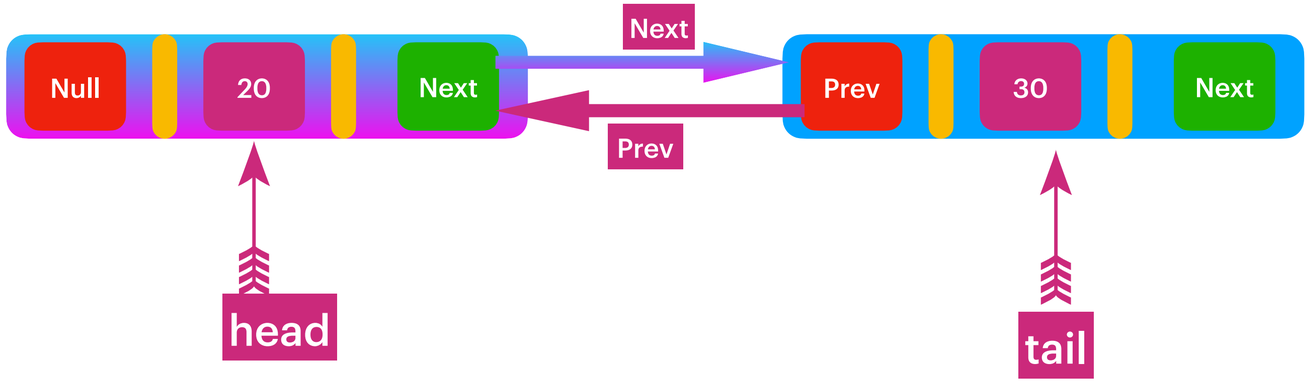
Traverse till currentNode.
Mark currentNode prev to newNode & previousNode next to newNode.
Mark newNode prev to previousNode & newNode next to currentNode.



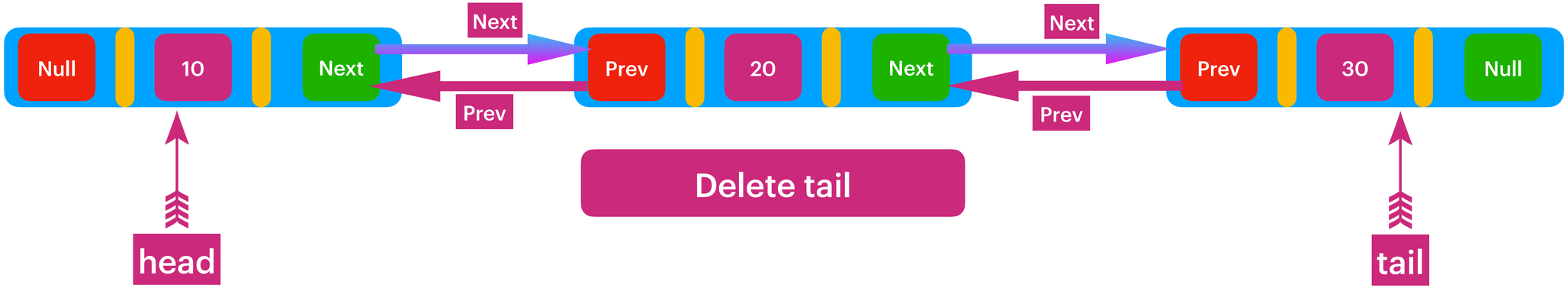
Time Complexity : O(n)



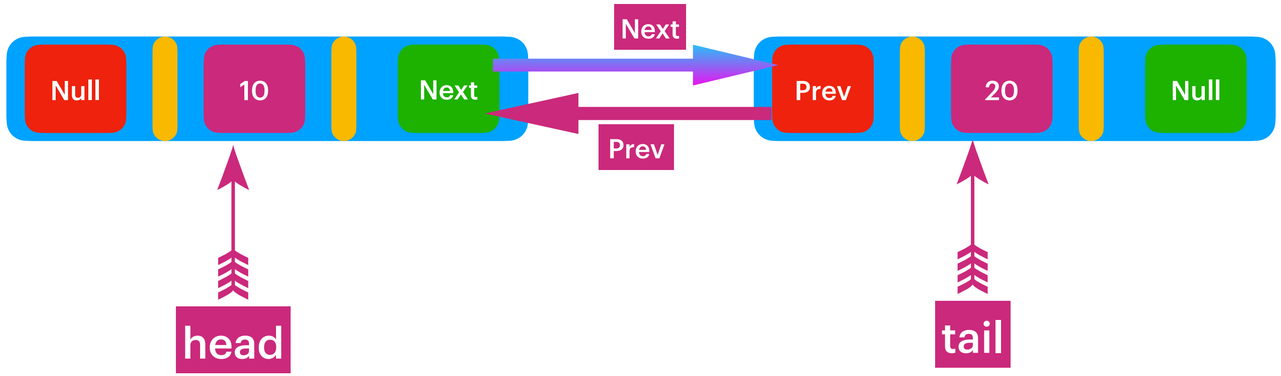
Mark head.next as newHead, then make newHead prev to null.



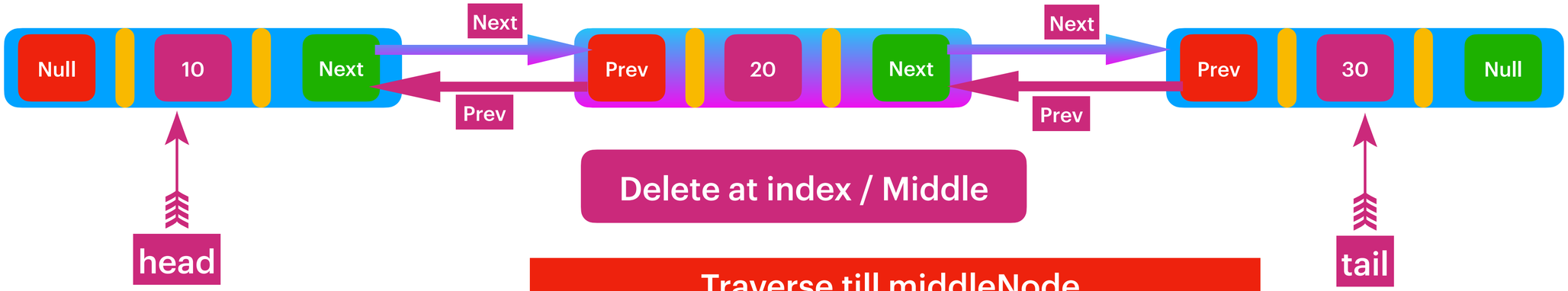
Time Complexity : O(1)



Mark tail.prev as new tail, then make new tail next as null.

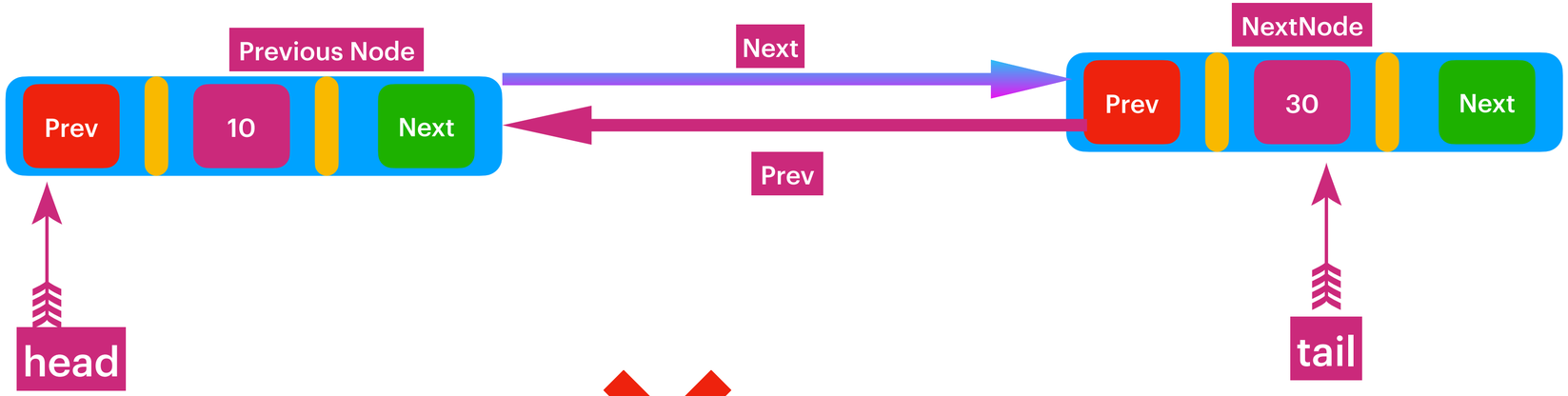


TimeComplexity : O(1)

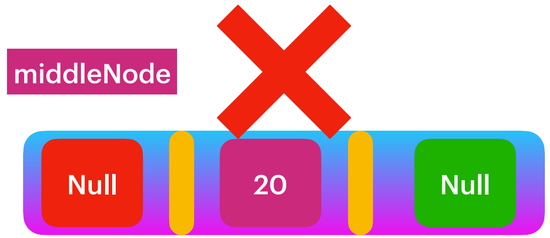


Delete at index / Middle

Traverse till middleNode.
Mark MiddleNode next to previousNode next.
Mark MiddleNode prev to nextNode prev.
Mark MiddleNode next & prev as null.



TimeComplexity : O(n)



Now no reference is pointing to middleNode so that middleNode will be garbage collected.