Detect cycle and find length

7/6/22

- To detect a cycle in a linked list, we must use slow and fast pointer approach.
- In this approach, slow will iterate by one step and fast will by 2 steps.
- If there is a cycle, then the two pointers will meet
- (slow and fast pointers will start at same beginning node)

```
def f(H: ListNode) -> bool:
    slow = fast = H
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            return True

    return False
```

head

- To find the length of the cycle, we need to start at the position where fast and slow pointers met, (from detecting a cycle) and use two pointers again.
start pointer iterate by one step
~~and step pointer iterate by the~~
~~it's then value which incrementing~~
~~until start and step pointers meet~~

---

and step increments until the start pointer cycles around the linked list back to end and then return the steps.

```
def detect_cycle(head: ListNode) -> ~~Optional[ListNode]~~ int
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            return calc_cycle_length(slow)
    return 0

def calc_cycle_length(end):
    start, step = end, 0
    while True:
        start = start.next
        step += 1
        if start is ~~step~~ end:
            return step
```

7/6/22 $\boxed{7.3}$

If space isn't a issue... (O(n) space)
Use hashset

```
def f(head: ListNode) -> ListNode:

    curr = head
    d = set()

    while curr:
        if curr in d:
            return curr
        ~~curr = curr.next~~
        d.add(curr)
        curr = curr.next

    return None
```

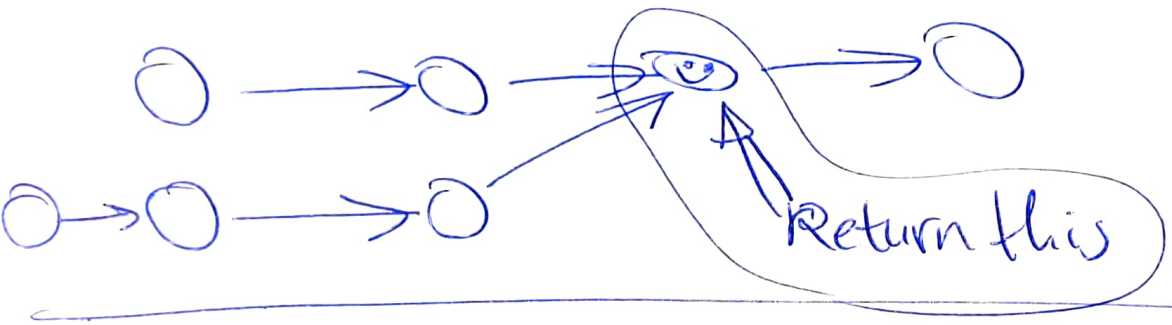
- If we want to do this $O(n), O(1)$, then we need to use slow and fast pointers approach. to detect if there is a cycle.
- Then find the cycle length.
- Then have a ~~another~~ pointer start at head and another start at the cycle length ~~point~~ ahead of the start and iterate the two pointers until they meet.
- They will meet at the start of the cycle (Floyd Cycle Chasing)

1 hr 46 min

7/6/22

7.4

Note: Did not understand the question. We are returning the node in which two LL intersect.



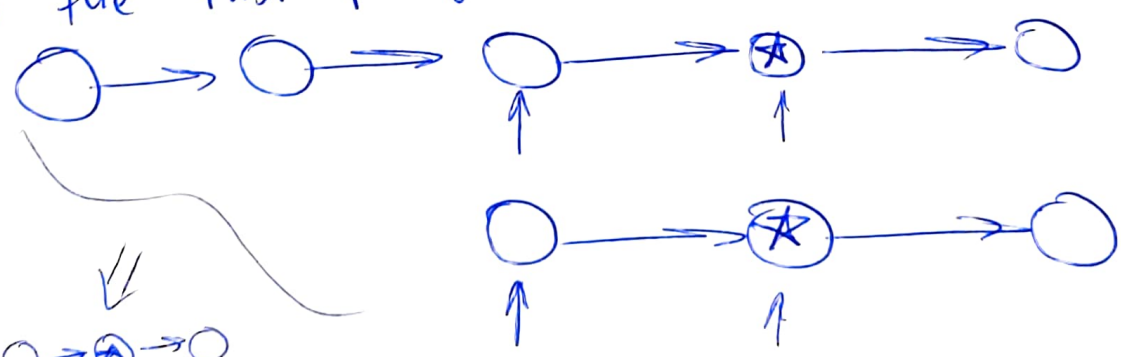
The core idea:

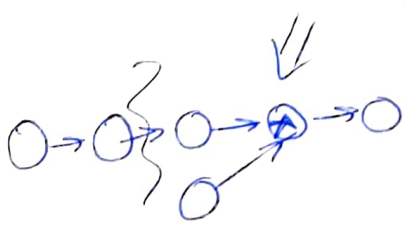
The lists overlap $\Longleftrightarrow$ Both lists have the same tail node.

~~Steps:~~

To solve this, we had to get the length of both nodes and start and compare each node in the same relative position.

(Skip the first |longerlist − shorterlist| nodes)

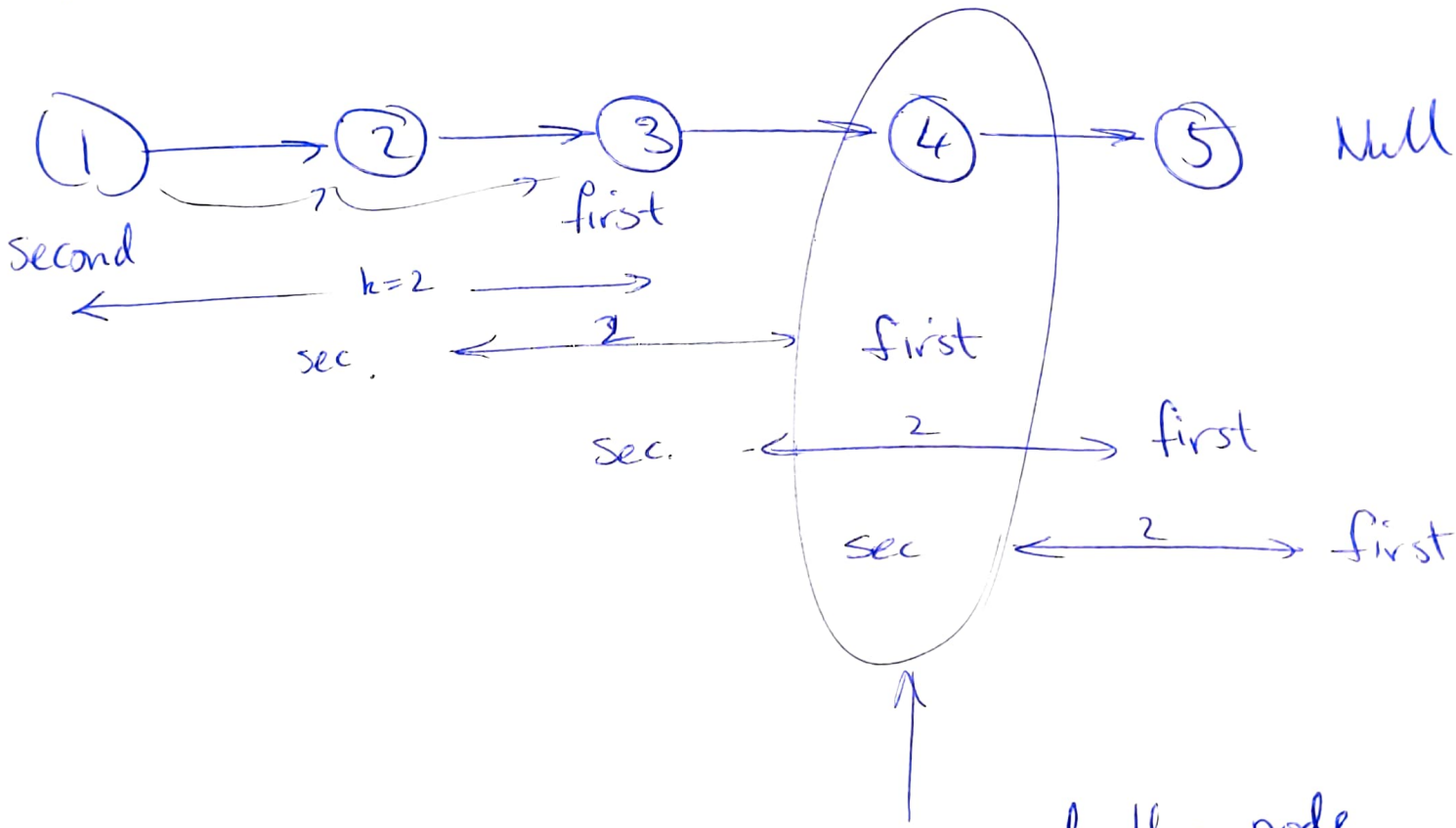




7/26/22 but Vague (it was on the right track) $\boxed{7.7}$

- The idea was correct but had no idea how to implement.

$k = 2$

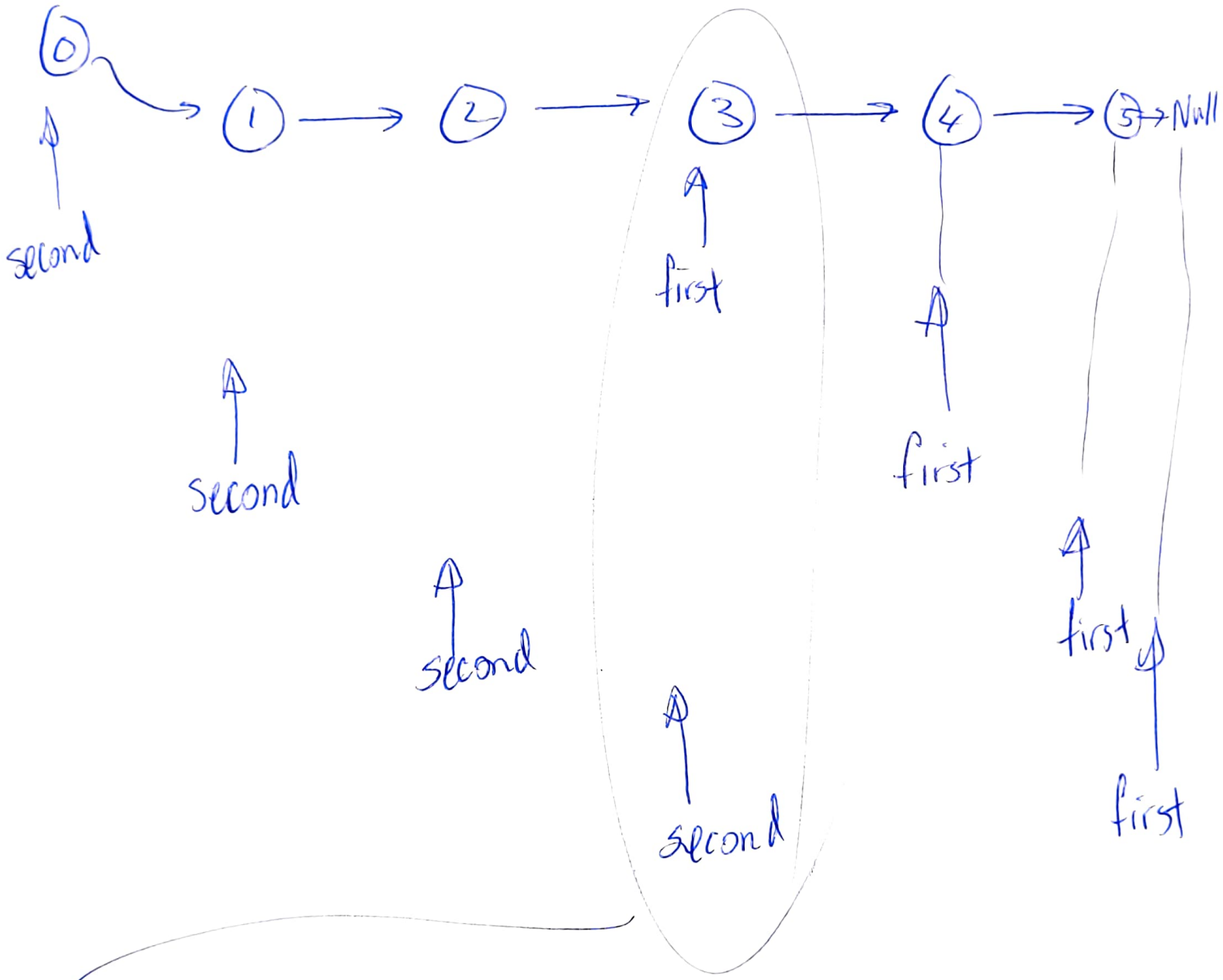


we are at the node we want to delete

But if we want to delete this node, then we have to be in k-1 th node to change pointers

use dummy node.

Same as before but the second pointer starts at 0. First pointer starts at same spot.



Now change pointer to k+1th node.

TOP "HARD" Interview problem from FANG 7/6/22

$$\begin{cases} A = [\text{'dog'}, \text{'dark'}, \text{'cat'}, \text{'door'}, \text{'dodge'}] \\ s = \text{'do'} \end{cases}$$

$\Rightarrow [\text{'dog'}, \text{'door'}, \text{'dodge'}]$

- Maybe use regular exp. ?

- The Naive approach that comes to mind: traverse the list and check item by item if 'do' is in the item string.

$\boxed{\text{Ex}}$

```
result = []
for item in A:
    if s in item
        result.append(item)

return result.
```

- This will take $O(n)$ time & space where n is $|A|$.

$\boxed{\text{5 min}}$

- Can I make it O(1) space?
  - ↳ Instead of append can I slice the list?
  - ↳ I can reorder and swap position of the items in the list?

- I can switch positions w/ pointers and put the items that is valid to the front. I can keep count and return the first "count" number of items.

```
def f(A, s):
    ~~left, cur~~
    ~~for item in A:~~
    left, curr = 0, 1
    while curr < len(A):
        if ~~A[curr]~~ s in A[curr]:
            A[left] = A[curr]
            A[curr] = A[left]
            left, curr++
        else: curr++
```

14min

My sol'n:

```
def f(A:List[str], s:str) -> List[str]:
    left, curr = 0, 0
    while curr < len(A):
        if s in A[curr]:
            A[left], A[curr] = A[curr], A[left]
            left, curr = left+1, curr+1
        else:
            curr += 1
    return A[:left]
```

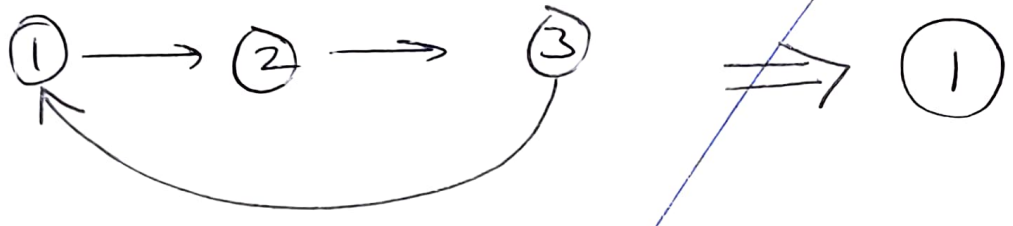
18 min

I realized that I should be asking if the input s can be anywhere in the ~~str~~ item string or only in the front.

7.3

Input: LL

Output: value at the start of cycle
or null if cycle doesn't exists.



The Naive approach is to iterate through the list and check to see if the next value is None.

↳ If None, return None

↳ ELSE, return the "next"

- We could a search method and iterate through the whole list.
- When the iteration reached the end, the next node will be the first node if it's a cycle.

1/2

Pseudocode

```
def f(L: ~~B~~ ListNode) → Optional[ListNode]:
    ~~current = ListNode~~
# (*) Should I create a dummy node? → No

    while L and L.data ≠ None:
        L = L.next

    ~~current~~
    return L.next
```

9min

How do I make test case for LL cycle?

Input: Cycle-free LL (two)

Output: bool?

74

7/6/22

Naive Approach:

- Iterate through both lists.
- Take an node from one LL and check w/ all nodes from the other LL.

→ $O(n^2)$, $O(1)$ (check node.next)

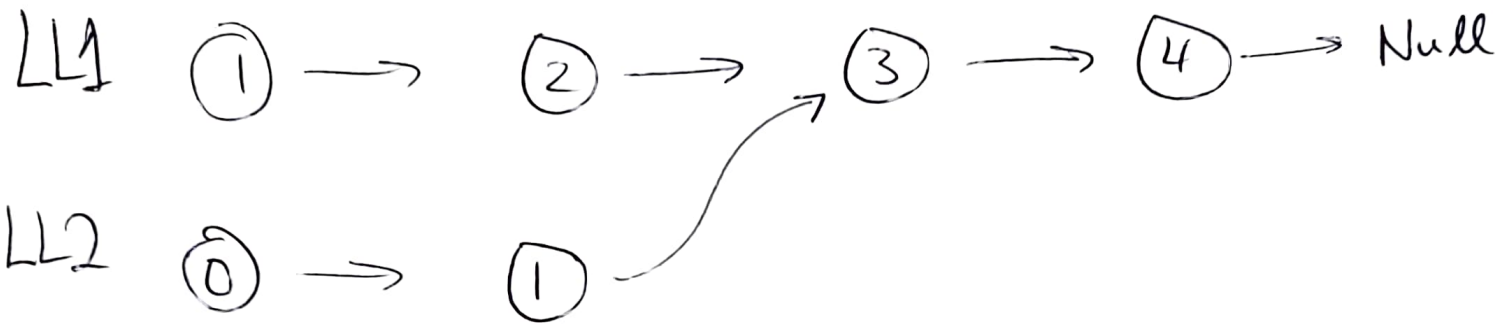
Use Hash set.

Add all nodes from LL1 to the hash set and check the other LL (LL2)

↳ $O(n + m)$, $O(n)$

- Is there a $O(n)$, $O(1)$ approach?
- Can I use pointers?
- Should I make them into cycles?
- Can I use the length of each LL?

6min



- If there is an overlap, that means one of the LL doesn't point to null at the tail?
- IF I reverse the LL and one node points to two other, does that tell me anything?

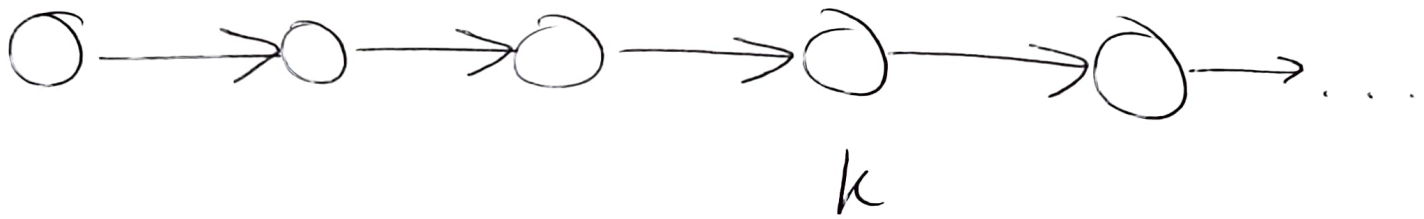
15 min Quit

7.7

7/6/22

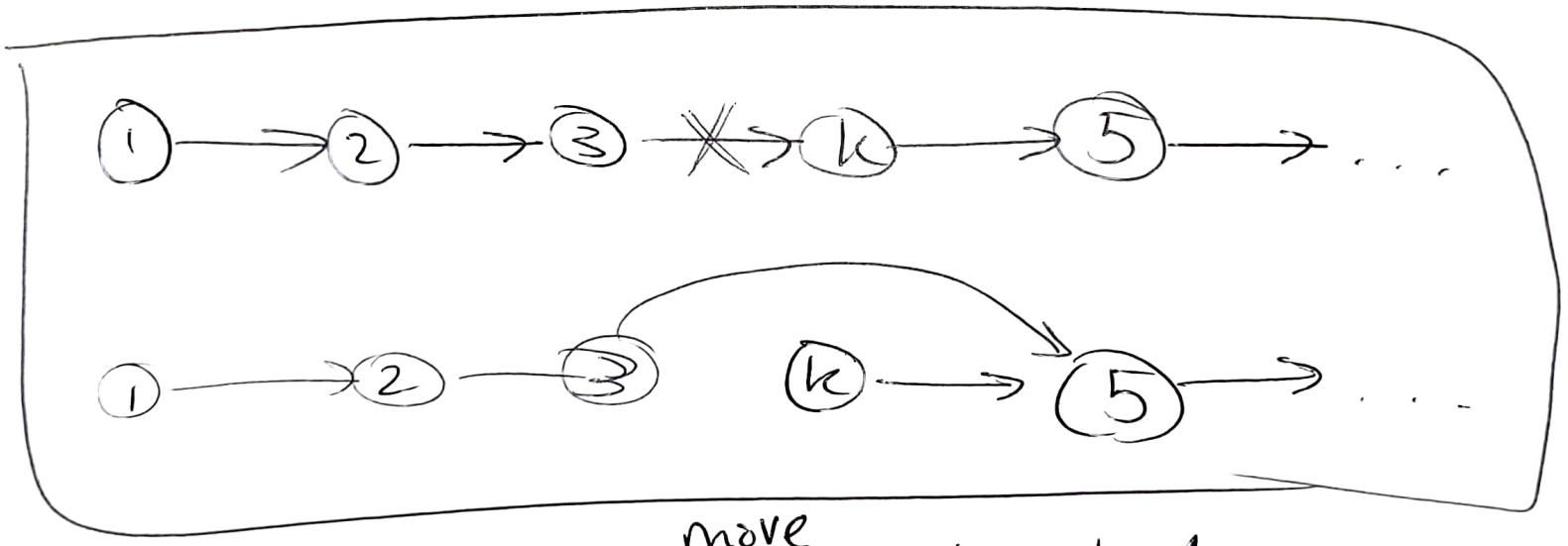
Input: LL and $k \in \mathbb{Z}$
Output: LL

Write a program that removes the kth last elem.



- Don't know the length of LL.

We have to do something like this



We would have to ~~count~~ move upto $k-1$ and make the elem. node point to $k+1^{th}$ node

This will take $O(n)$, $O(1)$.

Maybe have to set up a dummy / sentinel value.

~~Should I make a previous arg. in class ListNode?~~

~~x.prev.next = x.next~~
~~x.next.prev = x.prev.~~

10 min Quit

2/2