Iterative Link Reversal.

Intuition

In the previous approach, we looked at an algorithm for reversing a portion of the given linked list such that the underlying structure doesn't change. We only modified the values of the nodes for achieving the reversal. However, it may so happen that you cannot change the data available in the nodes. In that scenario, we have to modify the links themselves to achieve the reversal.

Essentially, starting from the node at position `m` and all the way up to `n`, we reverse the `next` pointers for all the nodes in between. Let's look at the algorithm for achieving this.

Algorithm

Before looking at the algorithm, it's important to understand how the link reversal will work and what set of pointers will be required for the same. Let's say we have a linked list consisting of three different nodes, `A → B → C` and we want to reverse the links between the nodes and obtain `A ← B ← C`.

Suppose we have at our disposal, two pointers. One of them points to the node `A` and the other one points to the node `B`. Let's call these pointers `prev` and `cur` respectively. We can simply use these two pointers to reverse the link between `A and B`.

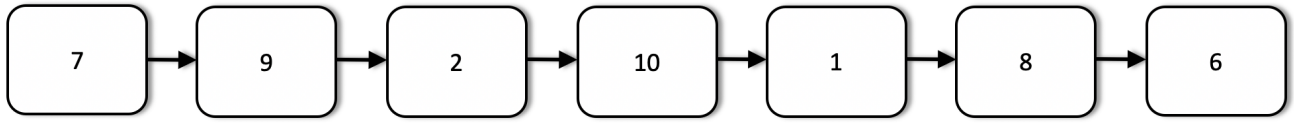
```
cur.next = prev
```

The only problem with this is, we don't have a way of progressing further i.e. once we do this, we can't reach the node `C`. That's why we need a third pointer that will help us continue the link reversal process. So, we do the following instead.

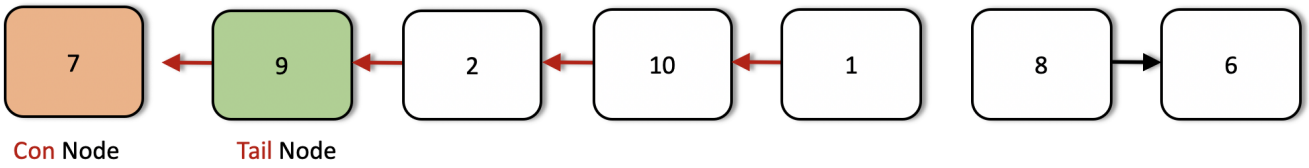
```
third = cur.next
cur.next = prev
prev = cur
cur = third
```

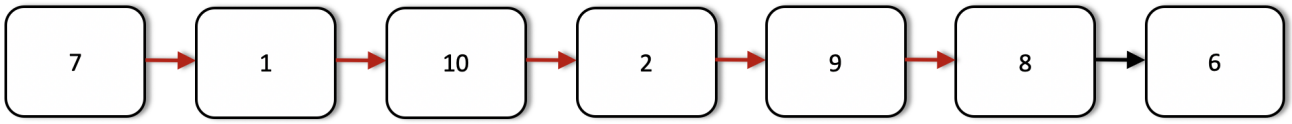
We do the above *iteratively* and we will achieve what the question asks us to do. Let's look at the steps for the algorithm now.

1. We need two pointers, `prev` and `cur` as explained above.
2. The `prev` pointer should be initialized to `None` initially while `cur` is initialized to the `head` of the linked list.
3. We progress the `cur` pointer one step at a time and the `prev` pointer follows it.
4. We keep progressing the two pointers in this way until the `cur` pointer reaches the m^{th} node from the beginning of the list. This is the point from where we start reversing our linked list.
5. An important thing to note here is the usage of two additional pointers which we will call as `tail` and `con`. The `tail` pointer points to the m^{th} node from the beginning of the linked list and we call it a *tail* pointer since this node becomes the tail of the reverse sublist. The `con` points to the node one before m^{th} node and this connects to the new head of the reversed sublist. Let's take a look at a figure to understand these two pointers better.



Suppose given this linked list, we want to reverse the sub list between nodes 2 and 5. That means we want the final list to be 7 → 1 → 10 → 2 → 9 → 8 → 6



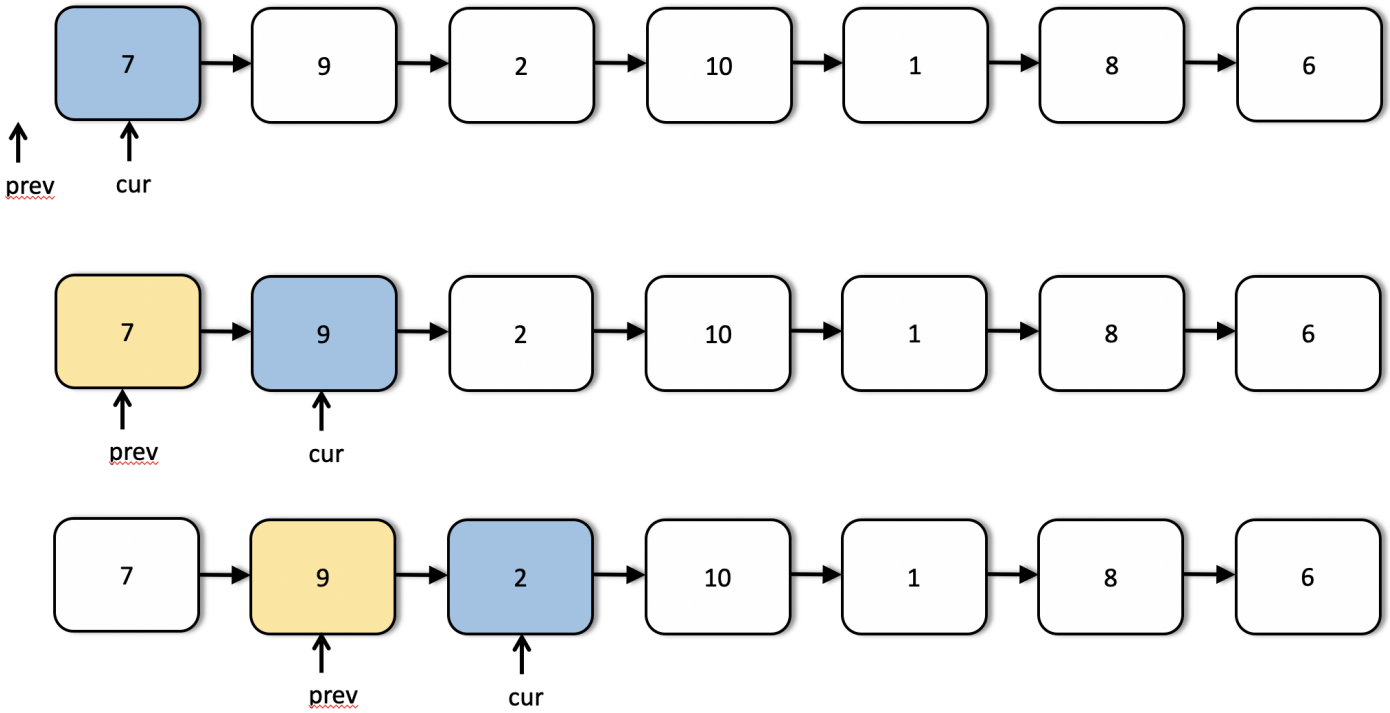


6. The `tail` and the `con` pointers are set once initially and then used in the end to finish the linked list reversal.

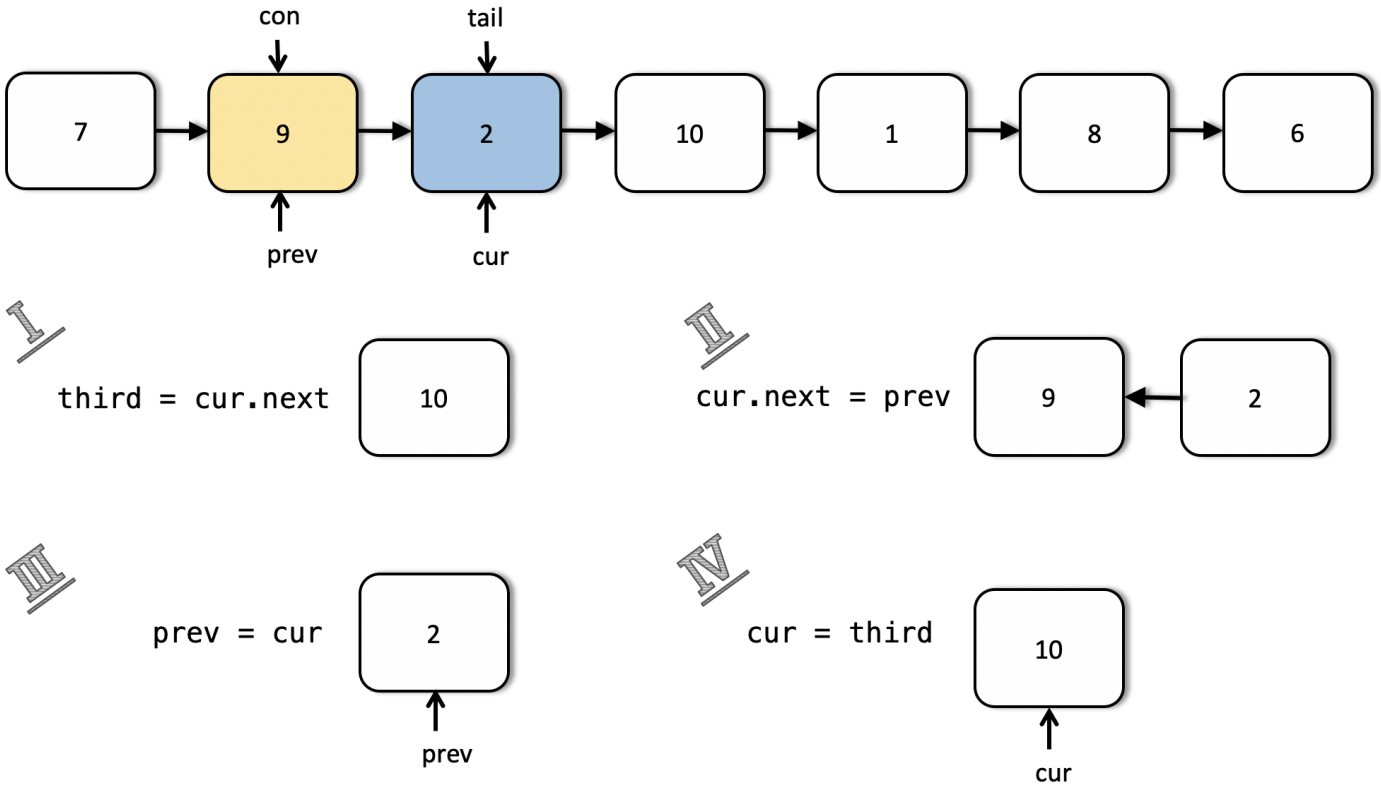
7. Once we reach the m^{th} node, we iteratively reverse the links as explained before using the two pointers. We keep on doing this until we are done reversing the link (next pointer) for the n^{th} node. At that point, the `prev` pointer would point to the n^{th} node.

8. We use the `con` pointer to attach to the `prev` pointer since the node now pointed to by the `prev` pointer (the n^{th} node from the beginning) will come in place of the m^{th} node due after the reversal. Similarly, we will make use of the `tail` pointer to connect to the node next to the `prev` node i.e. $(n+1)^{th}$ node from the beginning.

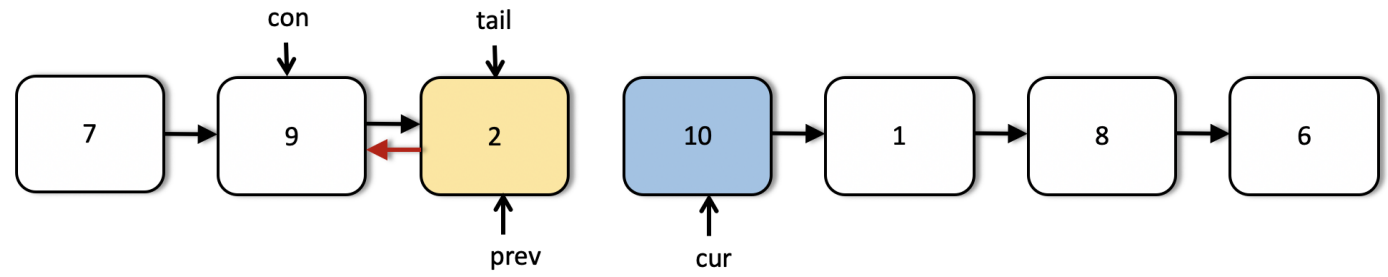
Let's have a look at the algorithm execute on a sample linked list to make the use case for all these pointers clearer. We are given a linked list initially with elements `7 → 9 → 2 → 10 → 1 → 8 → 6` and we need to reverse the list from node 3 through 6.

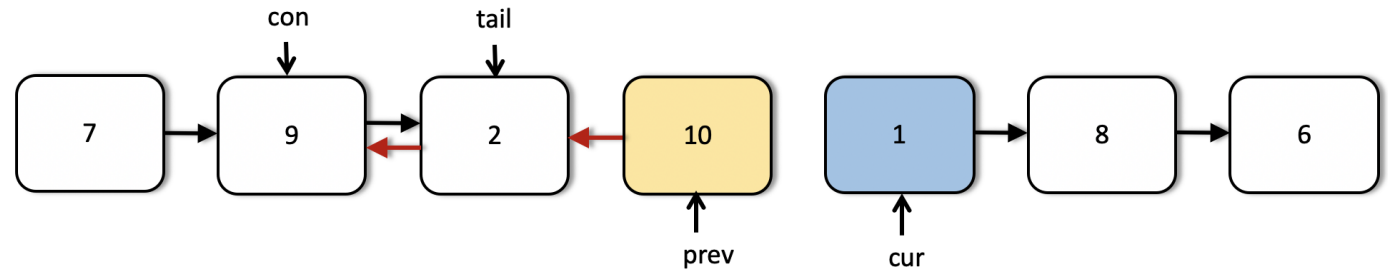


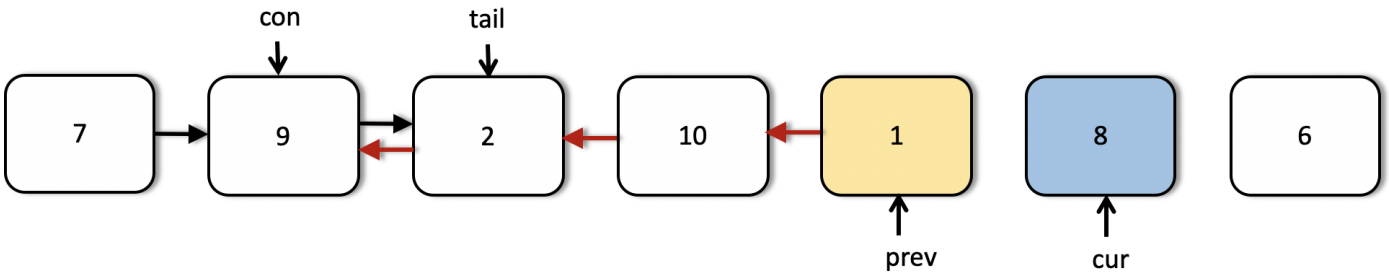
We can see the first few steps of our iterative solution above. The first step shows the initialization of the two pointers and the third step shows us the starting point for the list reversal process.

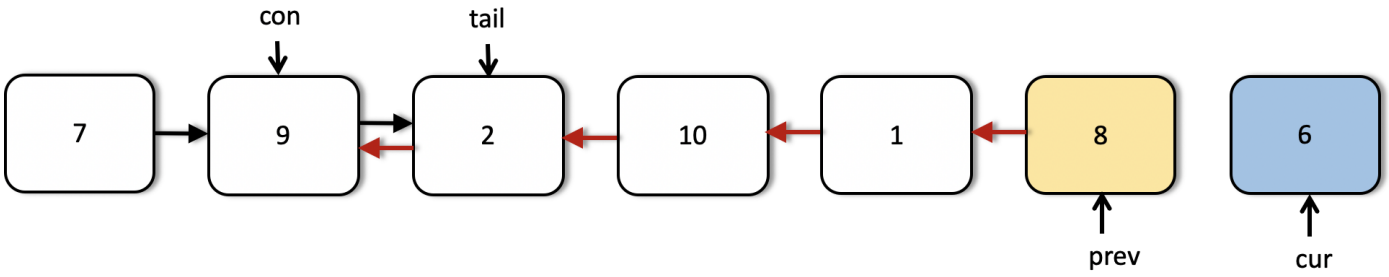


This shows us in detail how the links are reversed and how we move forward after reversing the links between two nodes. This step is done multiple times as shown in the following images.

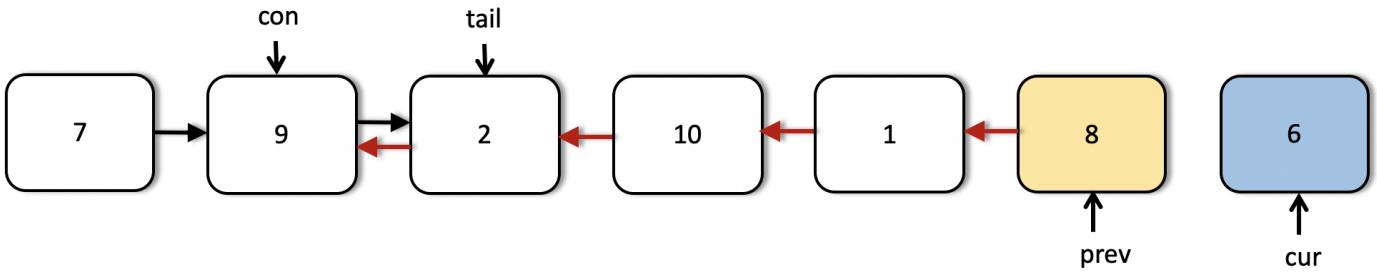


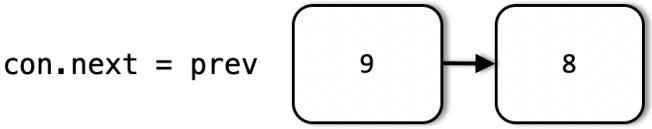


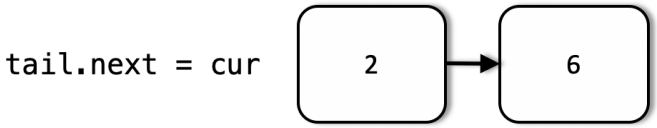


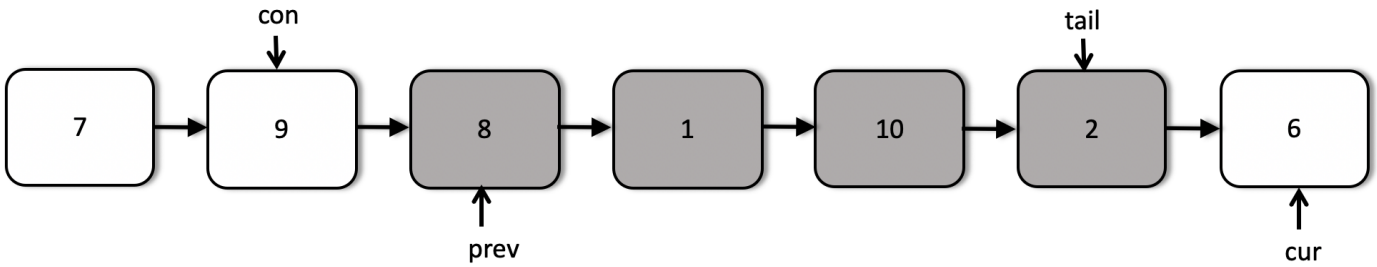


As we can see from the above images, now the two pointers have reached their final positions. We are done reversing the sublist that we were required to do i.e. nodes 3 through 6. However, we still have to fix some connections. The next image explains how we use the `tail` and `con` pointers to make the final connections.


con
tail
7
9
2
10
1
8
6
prev
cur


con.next = prev
9
8


tail.next = cur
2
6


con
tail
7
9
8
1
10
2
6
prev
cur