Approach 2: Iterative O(1) space

Intuition

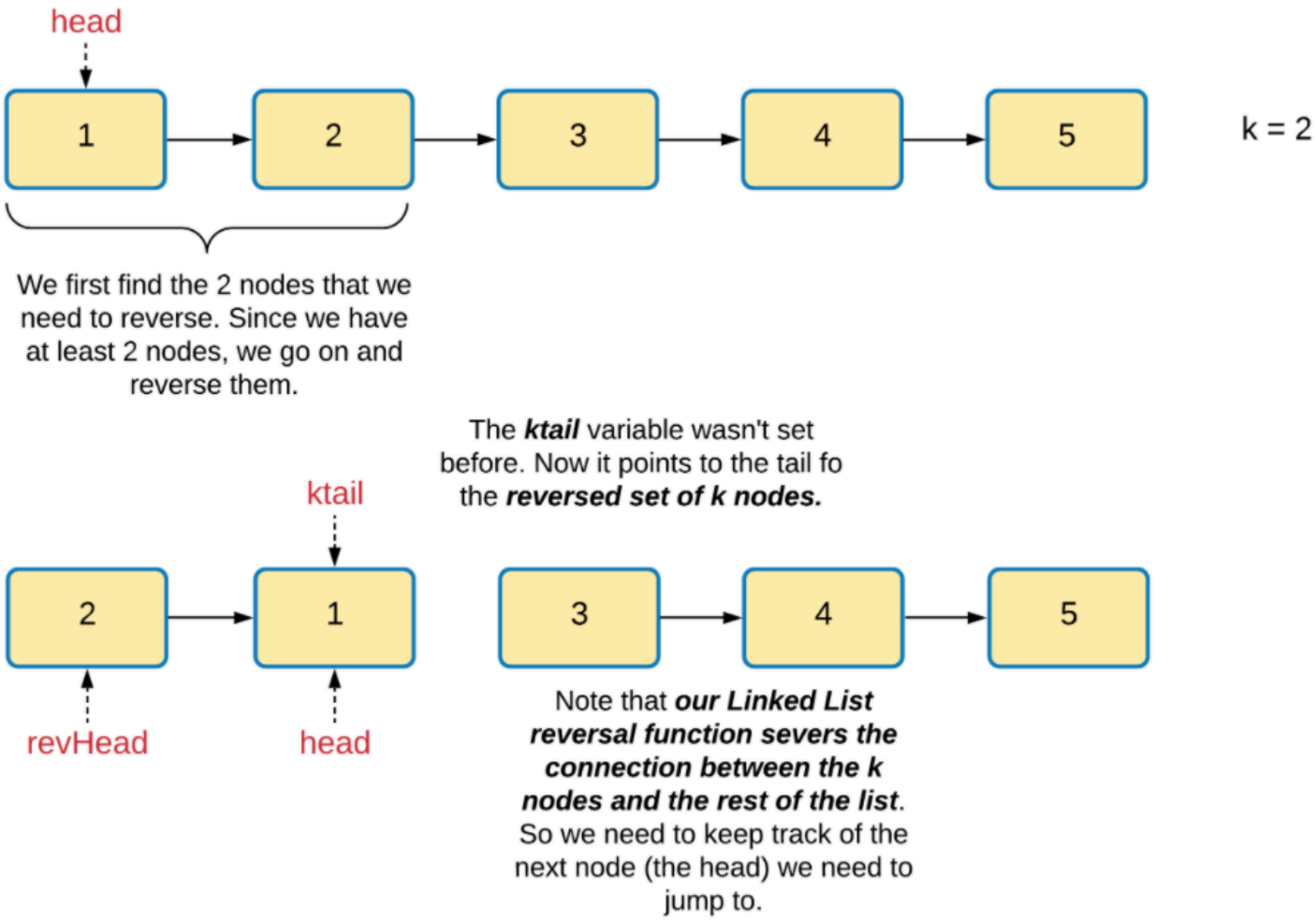
The idea here is the same as before except that we won't be making use of the stack here and rather use a couple additional variables to maintain the proper connections along the way. We still count `k` nodes at a time. If we find `k` nodes, then we reverse them.

> In addition to the "head" and "rev_head" variables from before, we need to know the "tail" node of the previous set of k nodes as well. The recursive approach reverses k nodes from left to right, but it establishes the connections from right to left or back to front. In this approach we will be reversing and establishing the connections while going from front to back.

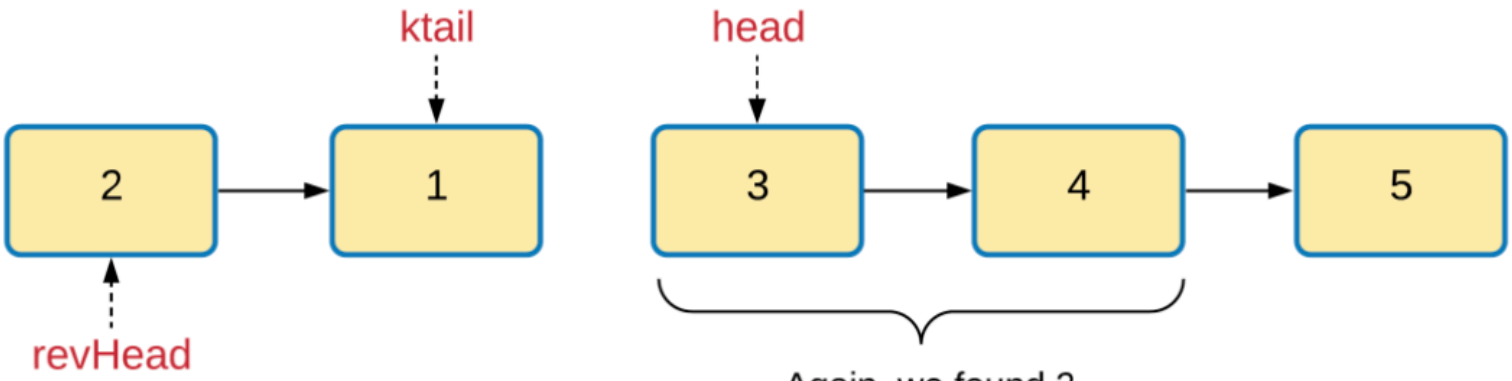
Algorithm

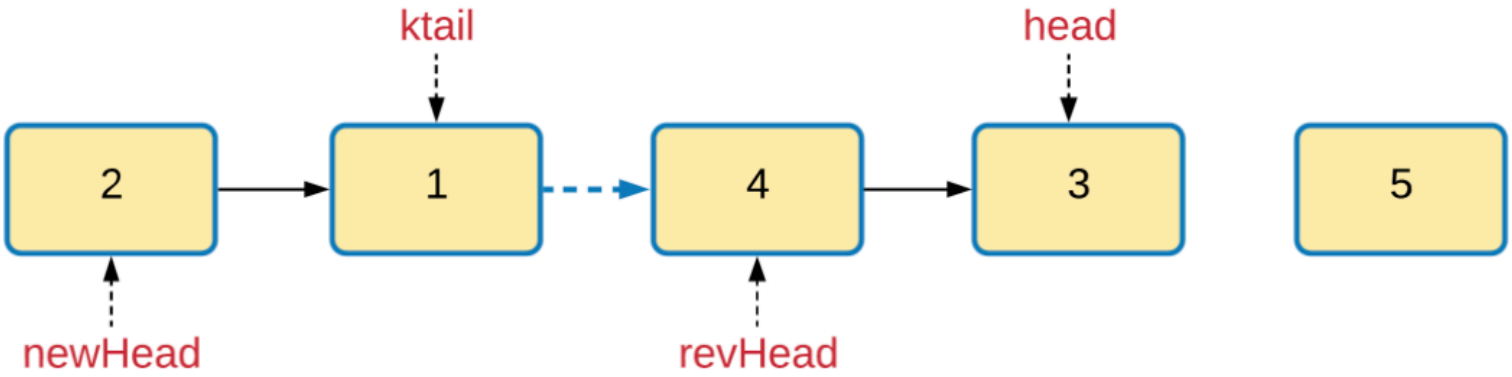
1. Assuming we have a `reverse()` function already defined for a linked list. This function would take the head of the linked list and also an integer value representing `k`. We don't have to reverse till the end of the linked list. Only `k` nodes are to be touched at a time.
2. We need to maintain four different variables in this algorithm as we chug along:
    1. *head* ~ which will always point to the original head of the next set of `k` nodes.
    2. *revHead* ~ which is basically the tail node of the original set of `k` nodes. Hence, this becomes the new head after reversal.
    3. *ktail* ~ is the tail node of the previous set of `k` nodes after reversal.
    4. *newHead* ~ acts as the head of the final list that we need to return as the output. Basically, this is the k^{th} node from the beginning of the original list.
3. The core algorithm remains the same as before. Given the `head`, we first count `k` nodes. If we are able to find at least `k` nodes, we reverse them and get our `revHead`.
4. At this point we check if we already have the variable `ktail` set or not. It won't be set when we reverse the very first set of `k` nodes. However, if this variable is set, then we attach `ktail.next` to the `revHead`. Also, we need to update `ktail` to point to the tail of the reversed set of `k` nodes that we just processed. Remember, the `head` node becomes the new tail and hence, we set `ktail = head`.
5. We keep doing this until we reach the end of the list or we encounter that there are `< k` nodes left in the list.

Let's look at the same linked list that we use for a dry run in the first approach. The first step simply assigns all the relevant pointers and reverses the first two nodes.



This step is really important since it highlights the use case of the `ktail` pointer here.





It's at this point that ***we need to connect the previous set of reversed k nodes and the current one***. That's why we have the ktail vaiable. ***We set it's next pointer to revHead and then set ktail to point to head***.