Previous Next

25. Reverse Nodes in k-Group

March 15, 2020 | 17.5K views

Average Rating: 4.75 (16 votes)

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.

Example:

Given this linked list: `1->2->3->4->5`

For $k = 2$, you should return: `2->1->4->3->5`

For $k = 3$, you should return: `3->2->1->4->5`

Note:

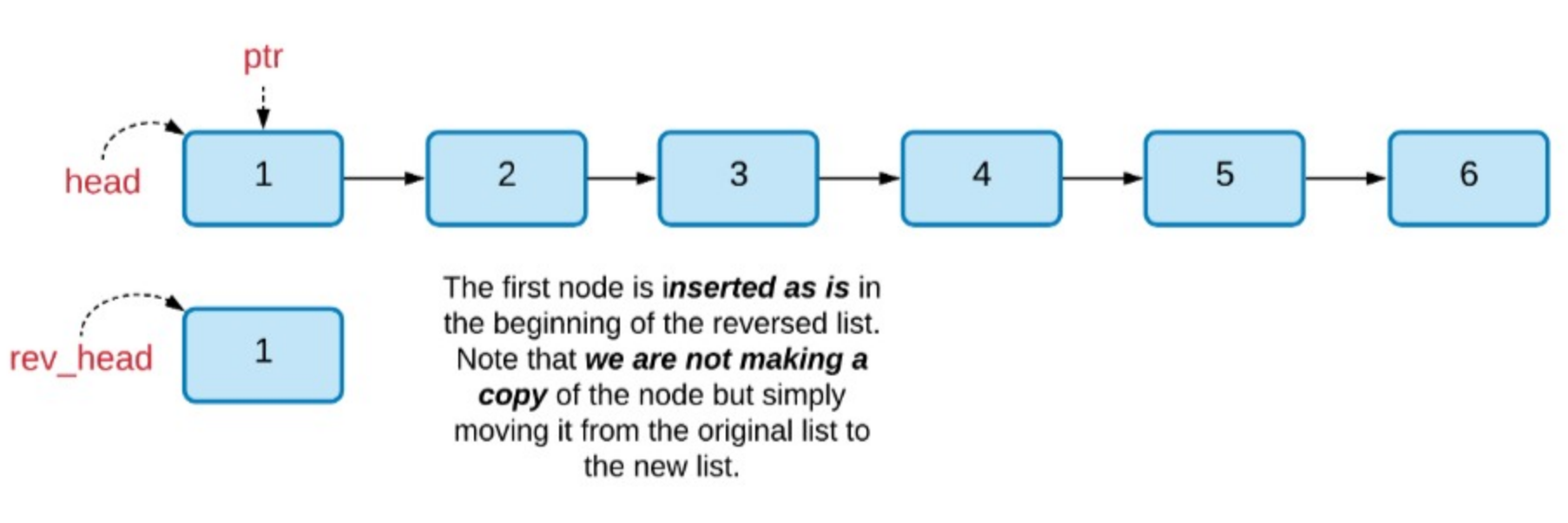
- Only constant extra memory is allowed.
- You may not alter the values in the list's nodes, only nodes itself may be changed.

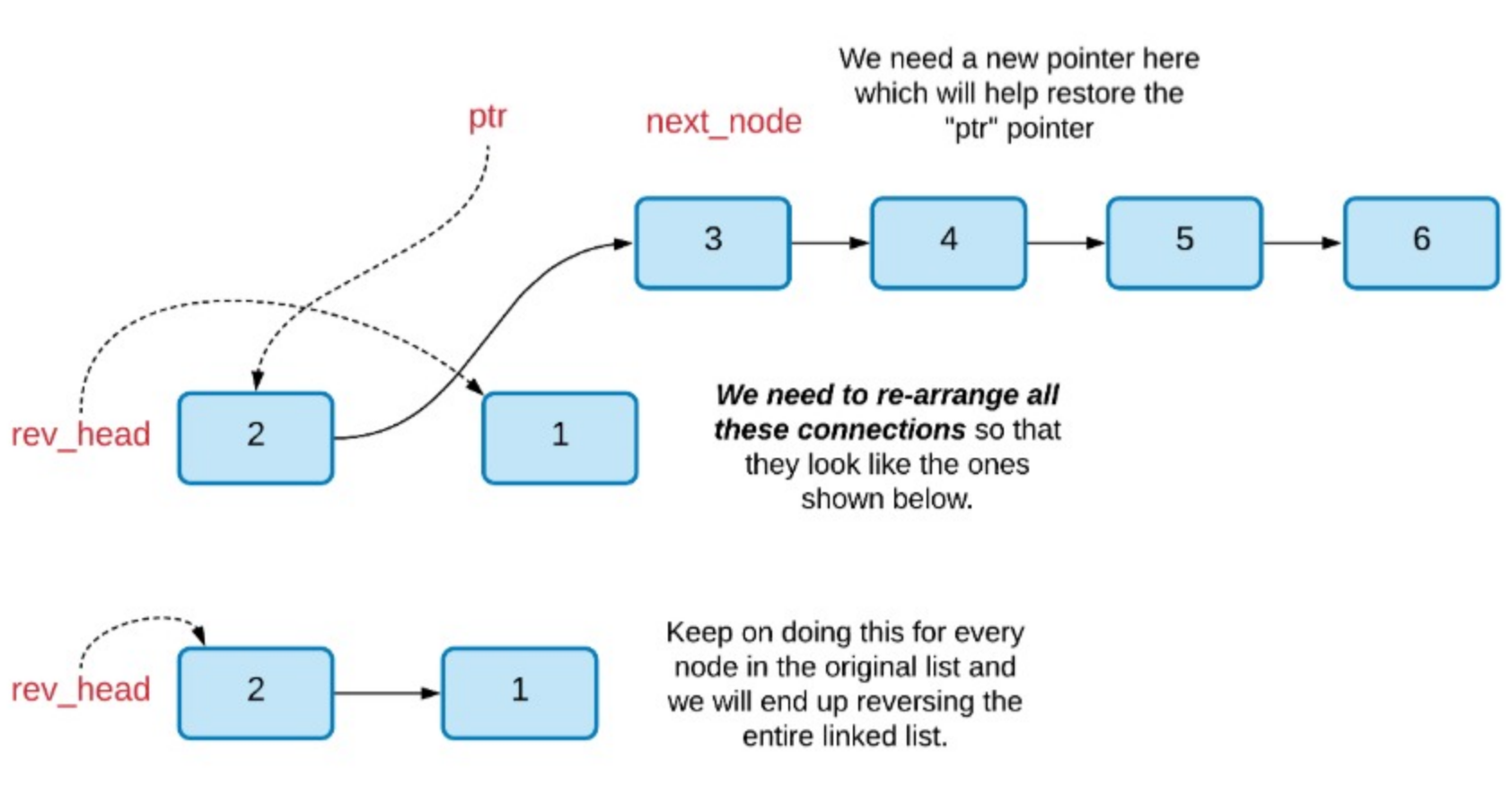
Solution

The problem statement clearly mentions that we are not to use any additional space for our solution. So naturally, a recursive solution is not acceptable here because of the space utilized by the recursion stack. However, for the sake of completeness, we shall go over the recursive approach first before moving on to the iterative approach. The interviewer may not specify the space constraint initially and so, a recursive solution would be a quick first approach followed by the iterative version.

A Linked list is a recursive structure. A sub-list in itself is a linked list. So, if you think about it, reversing a list consisting of k nodes is simply a linked list reversal algorithm. So, before we look at our actual approaches, we need to know that we will essentially be making use of a linked list reversal function here. There are many ways of reversing a linked list. A lot of programmers like to reverse the links themselves for reversing a linked list. What I personally like to do is to combine linked list traversal with insertion in beginning.

- Say the linked list we need to reverse has the starting node called `head`.
- Now, we will consider another pointer which will act as the head of the reversed linked list. Let's call this `rev_head`.
- We will use a pointer, `ptr` to traverse the original list.
- For every element, we basically insert it at the beginning of the reverse list which has `rev_head` as its head.
- That's it! We keep on moving `ptr` one step forward and keep inserting the nodes in the beginning of our reverse list and we will end up reversing the entire list.





Now that we have the basic linked list reversal stuff out of the way, we can move on with our actual problem which is to reverse the linked list, `k` nodes at a time. The basic idea is to make use of our reversal function for a linked list. Usually, we start with the head of the list and keep running the reversal algorithm all the way to the end. However, in this case, we will only process `k nodes`.

However, the problem statement also mentions that if there are `< k nodes` left in the linked list, then we don't have to reverse them. This implies that we first need to `count k nodes` before we get on with our reversal. If at any point, we find that we don't have `k nodes`, then we don't reverse that portion of the linked list. Right off the bat, this implies at least two traversals of the list overall. One for counting, and the next, for reversals.

Approach 1: Recursion

Intuition

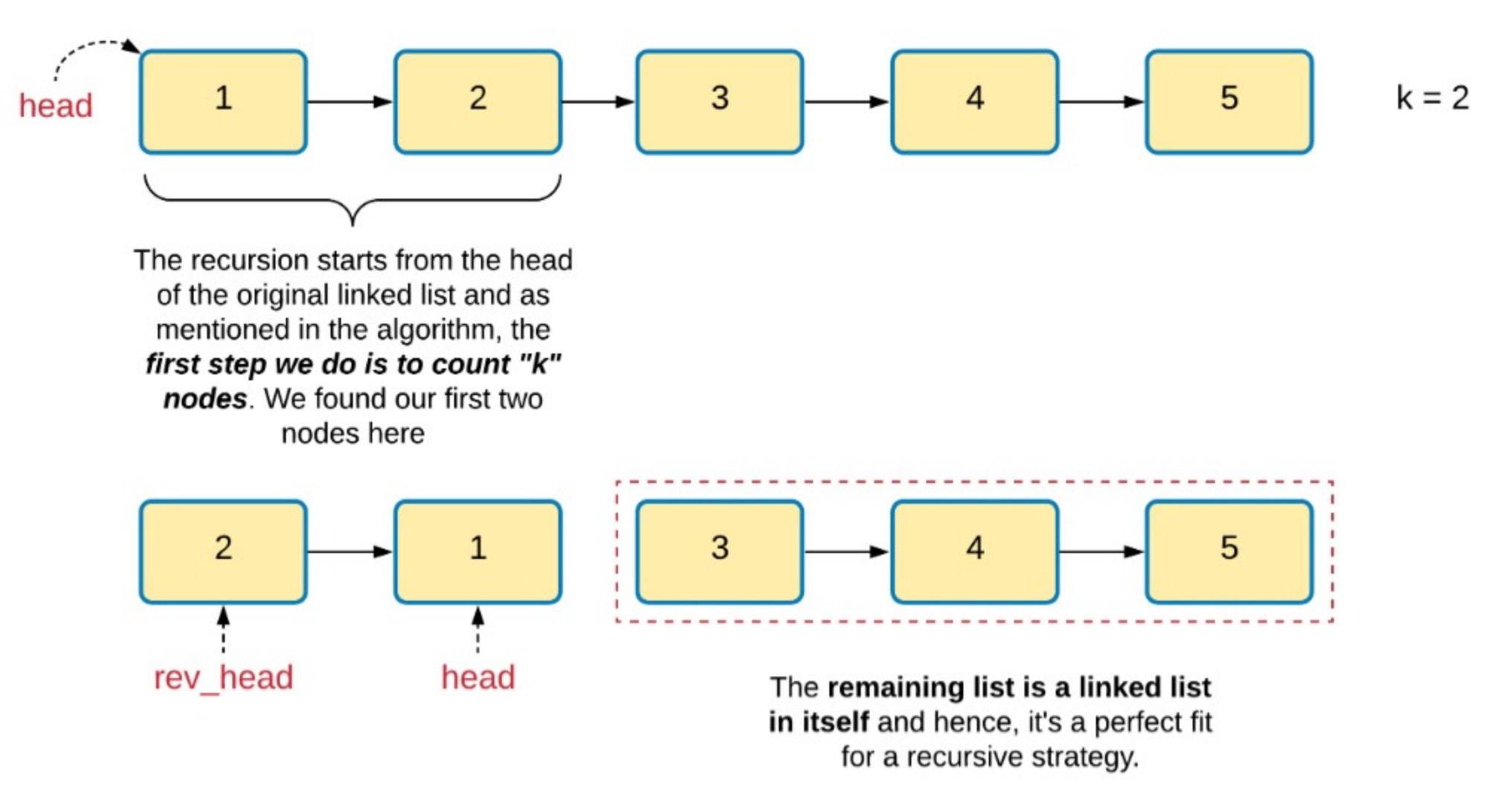
The recursive approach is a natural fit for this problem since the problem asks us to perform a modification operation on a fixed portion of the linked list, one portion at a time. Since a sub-list of a linked list is a linked list in itself, we can make use of recursion to do the heavy lifting for us. All we need to focus here is how we are going to reverse those k nodes. This part is sorted because we already discussed how general linked list reversal works.

We also need to make sure we are hooking up the right connections as recursion backtracks. For e.g. say we are given a linked list `1,2,3,4,5` and we are to reverse two nodes at a time. In the recursive approach, we will first reverse the first two nodes thus getting `2,1`. When recursion backtracks, we assume that we will have `4,3,5`. Now, we need to ensure that we hookup `1->4` correctly so that the overall list is what we expect.

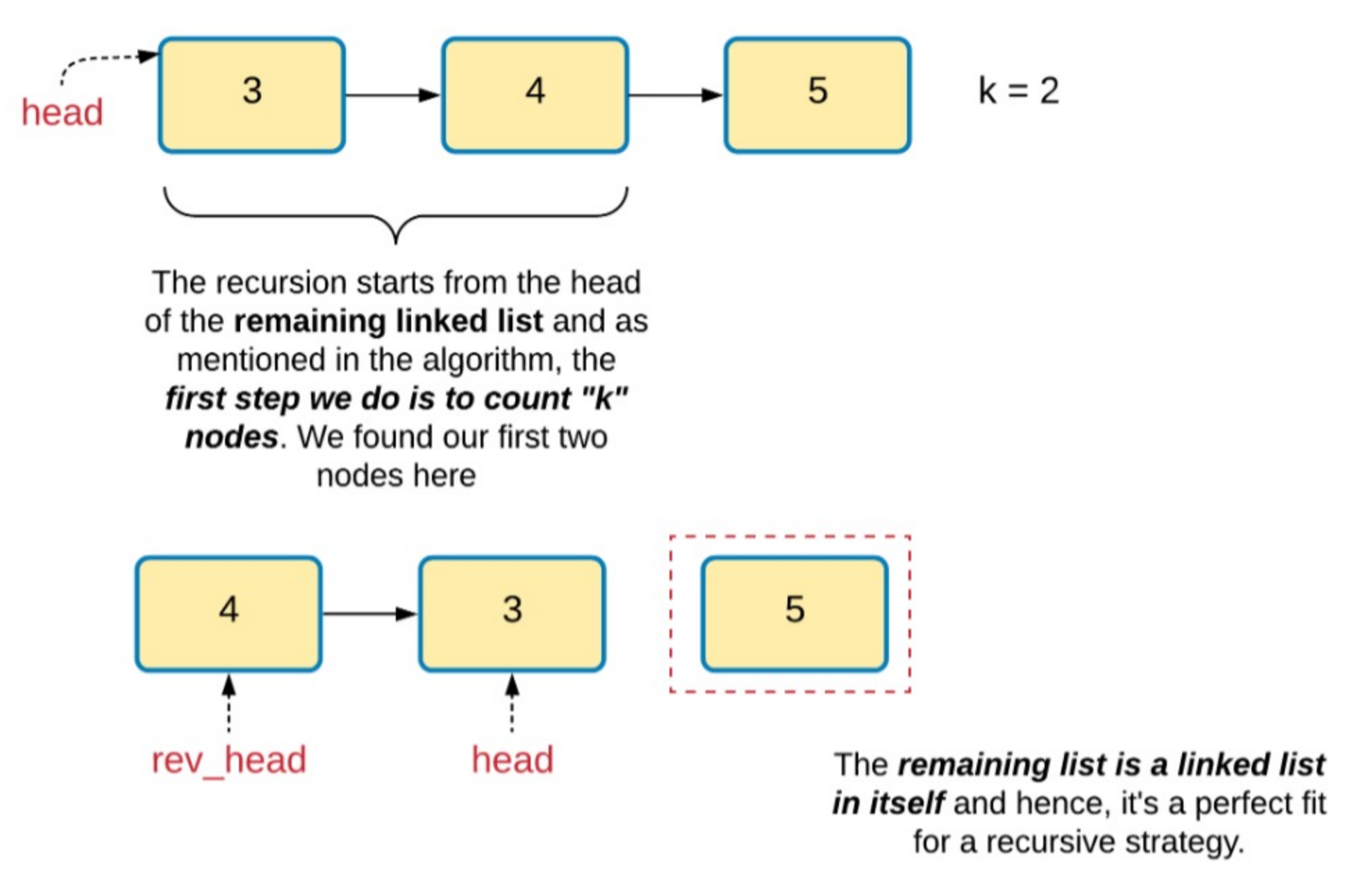
Algorithm

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. In every recursive call, we first count the number of nodes in the linked list. As soon as the count reaches `k`, we break.
3. If there are less than `k` nodes left in the list, we return the head of the list.
4. However, if there are at least `k` nodes in the list, then we reverse these nodes by calling our `reverse()` function defined in the first step.
5. Our recursion function needs to return the head of the reversed linked list. This would simply be the k^th nodes in the list passed to the recursion function because after reversing the first `k` nodes, the k^th node will become the new head and so on.
6. So, in every recursive call, we first reverse `k` nodes, then recurse on the rest of the linked list. When recursion returns, we establish the proper connections.

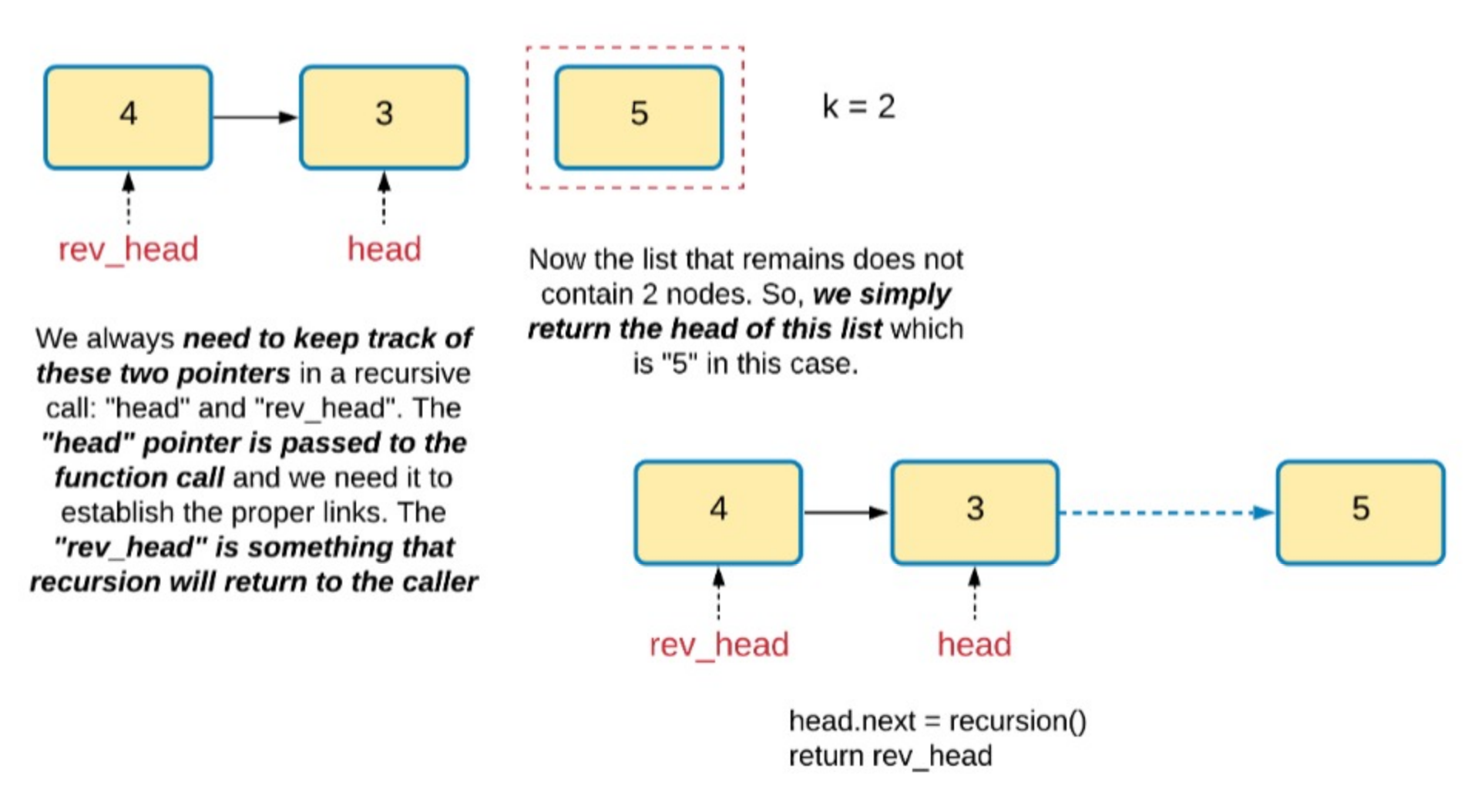
Let's look at a quick example of the algorithm's dry run. So, in the first recursive step, we process the first two nodes of the list and then make a recursive call.



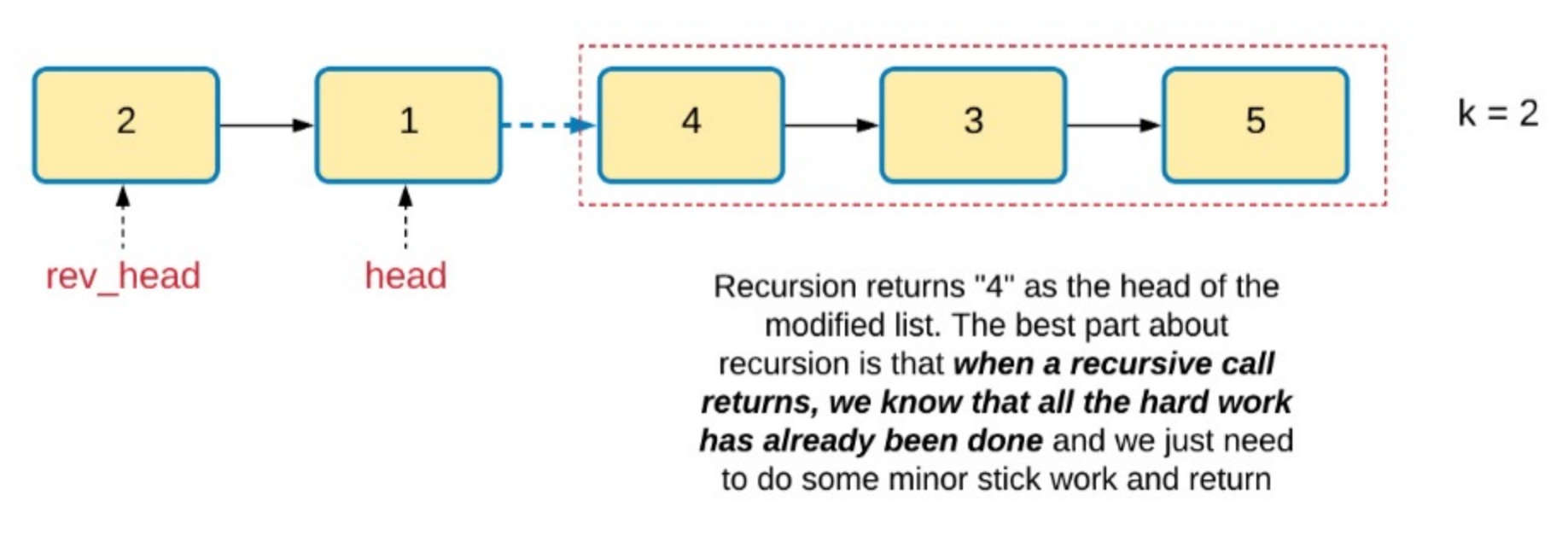
Here again, we process the two nodes and then make the final recursive call for this example linked list.



Now here we don't have enough nodes to reverse. So, in the recursive call we simply return the only remaining node here which is "5". Once that node is returned from the recursive call, we need to establish the proper connections i.e. from 3->5.



Similarly, recursion would return `4` as the new head node of the modified list ahead. We need to establish the connection from `1` to `4` and then return `2` as the head of the modified list.



Java Python Copy

```python
class Solution:

    def reverseLinkedList(self, head, k):

        # Reverse k nodes of the given linked list.
        # This function assumes that the list contains
        # atleast k nodes.
        new_head, ptr = None, head
        while k:

            # Keep track of the next node to process in the
            # original list
            next_node = ptr.next

            # Insert the node pointed to by "ptr"
            # at the beginning of the reversed list
            ptr.next = new_head
            new_head = ptr

            # Move on to the next node
            ptr = next_node

            # Decrement the count of nodes to be reversed by 1
            k -= 1

        # Return the head of the reversed list
        return new_head
```

Complexity Analysis

- Time Complexity: $O(N)$ since we process each node exactly twice. Once when we are counting the number of nodes in each recursive call, and then once when we are actually reversing the sub-list. A slightly optimized implementation here could be that we don't count the number of nodes at all and simply reverse k nodes. If at any point we find that we didn't have enough nodes, we can re-reverse the last set of nodes so as to keep the original structure as required by the problem statement. That ways, we can get rid of the extra counting.
- Space Complexity: $O(N/k)$ used up by the recursion stack. The number of recursion calls is determined by both k and N. In every recursive call, we process k nodes and then make a recursive call to process the rest.

Approach 2: Iterative O(1) space

Intuition

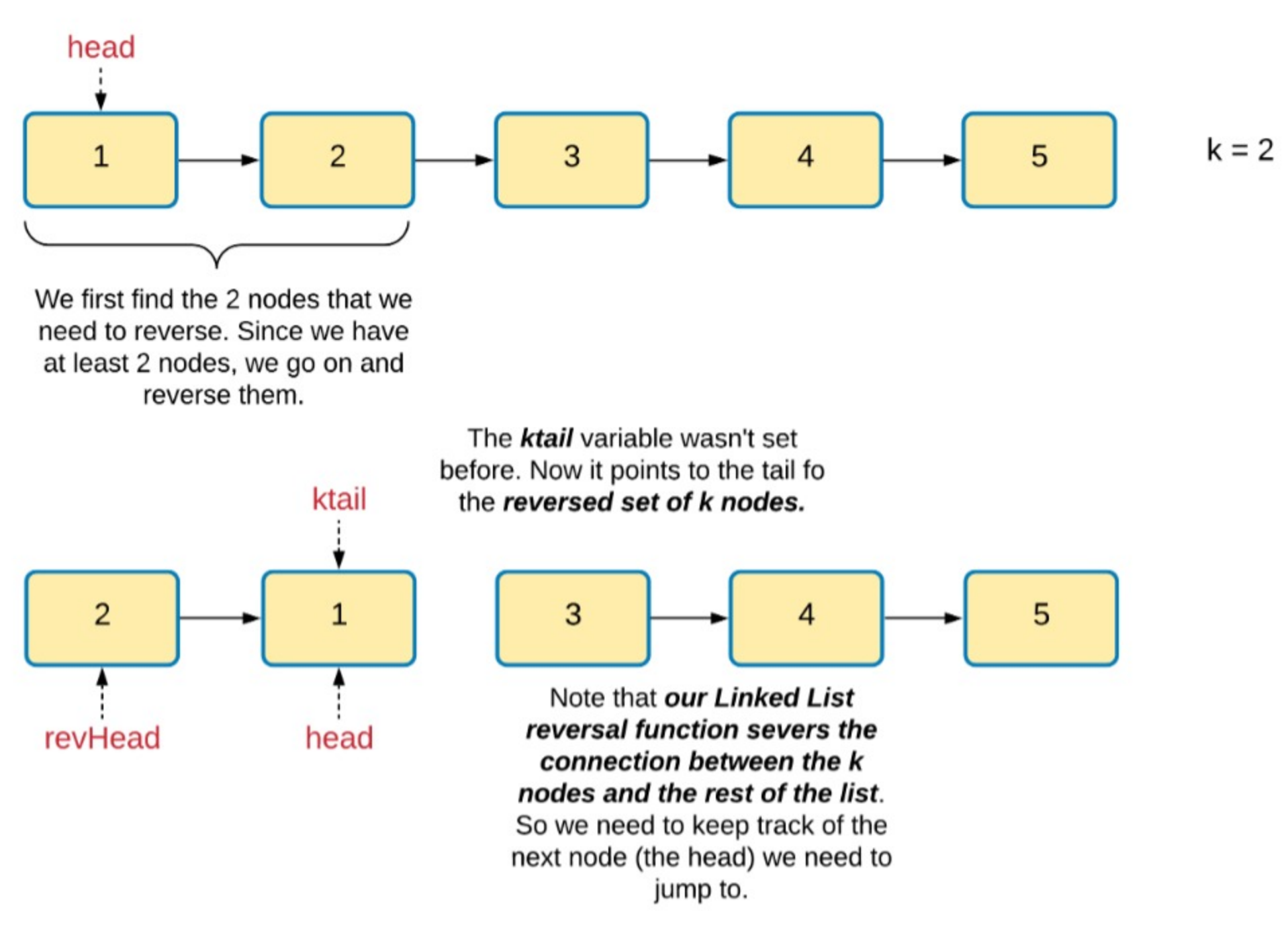
The idea here is the same as before except that we won't be making use of the stack here and rather use a couple additional variables to maintain the proper connections along the way. We still count `k` nodes at a time. If we find `k` nodes, then we reverse them.

> 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.

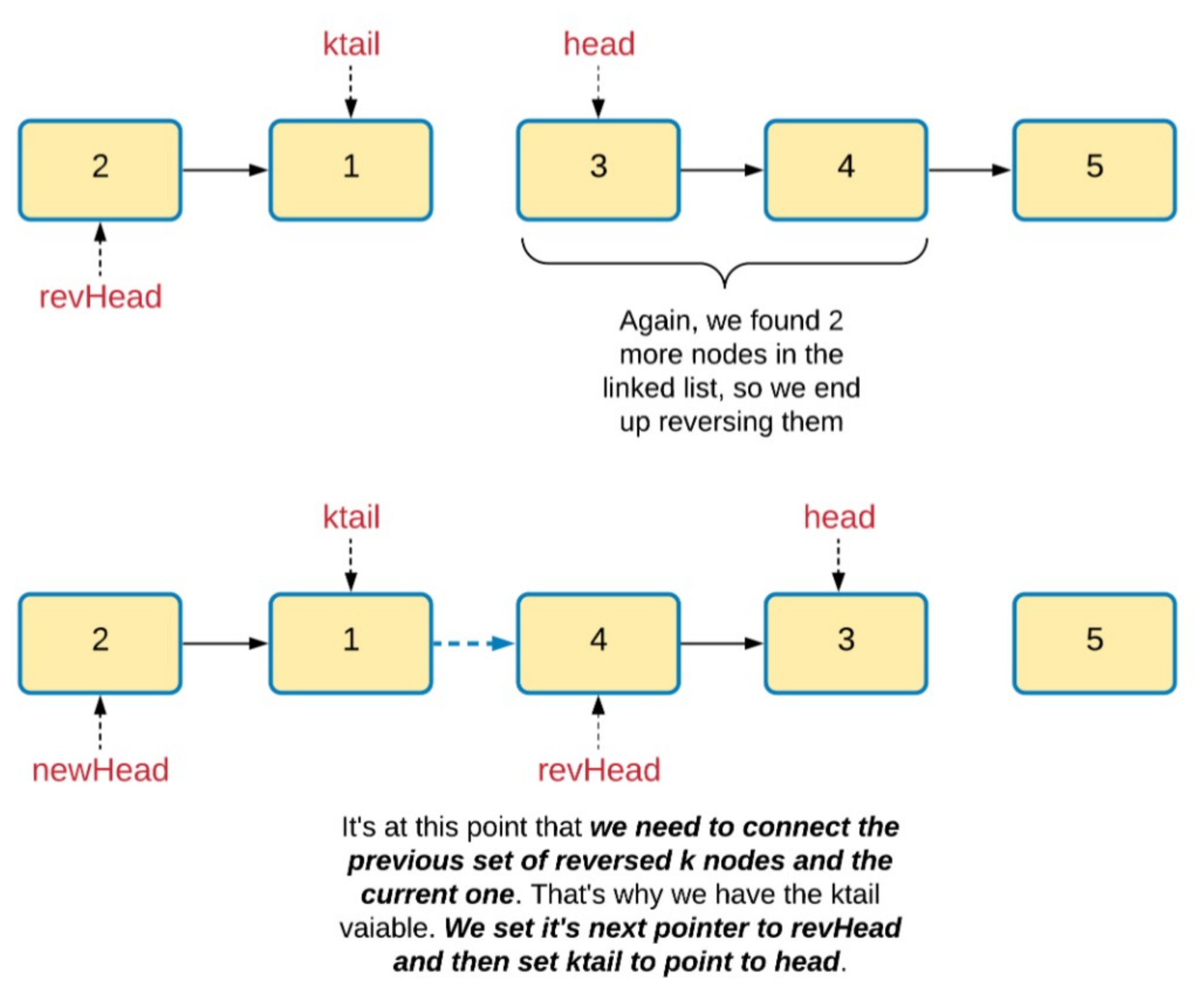
Algorithm

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.



Java Python Copy

```python
class Solution:

    def reverseLinkedList(self, head, k):

        # Reverse k nodes of the given linked list.
        # This function assumes that the list contains
        # atleast k nodes.
        new_head, ptr = None, head
        while k:

            # Keep track of the next node to process in the
            # original list
            next_node = ptr.next

            # Insert the node pointed to by "ptr"
            # at the beginning of the reversed list
            ptr.next = new_head
            new_head = ptr

            # Move on to the next node
            ptr = next_node

            # Decrement the count of nodes to be reversed by 1
            k -= 1

        # Return the head of the reversed list
        return new_head
```

- Time Complexity: $O(N)$ since we process each node exactly twice. Once when we are counting the number of nodes in each recursive call, and then once when we are actually reversing the sub-list.
- Space Complexity: $O(1)$.

Rate this article:

Previous Next

Comments: 4 Sort By

Type comment here... (Markdown is supported)

Preview Post

willye 861 April 2, 2020 10:20 AM Report
Managed to do the iterative version but took me too long... over 30 min.
6 Share Reply
SHOW 1 REPLY

derreck503 104 March 29, 2020 8:37 AM
Damn the recursive method makes it so easy. Ran into a lot of problems trying it iteratively.
3 Share Reply

SuperX 0 June 27, 2020 11:23 AM Report
Another recursive version. Reverse the sub-link list Recursively without checking the illegal length k.

```
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        dumb_head = pointer = ListNode()
```

Read More
0 Share Reply
SHOW 1 REPLY

FattyPiKa 19 May 12, 2020 9:48 AM
Great article. It helps a lot. The iterative way is much more trivial than the recursive way.
0 Share Reply
SHOW 1 REPLY