**How does your solution swap the k-th node from the beginning with the k-th node from the end in a singly linked list?**

To solve this problem efficiently, I used a two-pointer technique, which allows us to find the k-th node from the beginning and the k-th node from the end in a single pass through the linked list. Here's how my solution works:

1. **Handling Base Cases**:
   * First, I handle some basic edge cases. If the linked list is empty (head == null) or contains only one node (head.next == null), I simply return the head because there's no swapping needed.
2. **Finding the k-th Node from the Beginning**:
   * I initialize two pointers: first and second, both pointing to the head of the linked list.
   * I move the first pointer k-1 steps forward to reach the k-th node from the beginning. At this point, first points to the node that will eventually swap its value with the k-th node from the end.
   * I store this node in a variable called kthFromBegin.
3. **Finding the k-th Node from the End**:
   * Now, I want to find the k-th node from the end. I keep moving the first pointer until it reaches the end of the list, while simultaneously moving the second pointer from the head.
   * By the time first reaches the last node, second will have moved to the k-th node from the end of the list. This works because there is a gap of k nodes between first and second.
4. **Swapping the Values**:
   * At this point, kthFromBegin points to the k-th node from the beginning, and second points to the k-th node from the end.
   * I swap the values of these two nodes directly. This approach avoids the complexity of rearranging the actual nodes, which simplifies the logic and reduces the risk of pointer manipulation errors.
5. **Returning the Modified List**:
   * Finally, I return the head of the modified list. The list is now updated with the k-th node from the beginning and the k-th node from the end having their values swapped.

**Complexity Analysis:**

* **Time Complexity**: The solution runs in O(n) time, where n is the number of nodes in the list. This is because we make a single pass through the list to identify both the k-th node from the beginning and the k-th node from the end.
* **Space Complexity**: The space complexity is O(1) since we are only using a few extra pointers and variables, regardless of the size of the input linked list.

**Example to Illustrate:**

For instance, given a list 1 -> 2 -> 3 -> 4 -> 5 and k = 2, the output should be 1 -> 4 -> 3 -> 2 -> 5. Here, 2 is the 2nd node from the beginning, and 4 is the 2nd node from the end. We swap these nodes, resulting in the modified list.