1721. Swapping Nodes in a Linked List

```
Medium Linked List Two Pointers
```

Problem Description

In this problem, you are provided with the head of a singly <u>linked list</u> and an integer k. The goal is to return the modified linked list after swapping the values of two specific nodes: the kth node from the beginning and the kth node from the end. The linked list is indexed starting from 1, which means that the first node is considered the 1st node from both the beginning and the end in the case of a single-node list. Note that it's not the nodes themselves that are being swapped, but rather their values.

Intuition

To solve this problem, we think about how we can access the kth node from the beginning and the end of a singly linked list. A singly linked list does not support random access in constant time; meaning we can't directly access an element based on its position without traversing the list.

track of this node.

Finding the kth node from the end is trickier because we don't know the length of the <u>linked list</u> beforehand. However, by using two-pointers – fast and slow – and initially set both at the head, we can move fast k-1 steps ahead so that it points to the kth

Given we only need to swap the values and not the nodes themselves, a two-pointer approach is perfect. The first step is to

locate the kth node from the beginning which can be done by traversing k-1 nodes from the head. We use a pointer p to keep

node. Then, we move both fast and slow at the same pace. When fast reaches the last node of the list, slow will be pointing at the kth node from the end. We use another pointer q to mark this node.

Once we have located both nodes to be swapped, p and q, we simply exchange their values and return the (modified) head of the linked list as the result.

Solution Approach

The implementation of the solution follows a two-pointer approach that is commonly used when dealing with linked list problems.

Initialize two pointers fast and slow to reference the head of the linked list. Move the fast pointer k-1 steps ahead. After this loop, fast will be pointing

2. Move the fast pointer k-1 steps ahead. After this loop, fast will be pointing to the kth node from the beginning of the list. We use pointer p to mark this node.

3. Once the fast pointer is in position, we start moving both fast and slow pointers one step at a time until fast is pointing to the last node of the

linked list. Now, the slow pointer will be at the kth node from the end of the list. We use another pointer q to mark this node.

4. Swap the values of nodes p and q by exchanging their val property.

The algorithm is simple yet powerful and can be broken down into the following steps:

5. Finally, return the modified linked list's head.

This is a classical algorithm that does not require additional data structures for its execution and fully utilizes the <u>linked list</u>'s

kth node from the end, resulting in O(n) time complexity, where n is the number of nodes in the linked list.

- sequential access nature. It's efficient because it only requires a single pass to find both the kth node from the beginning and the
 - The only tricky part that needs careful consideration is handling edge cases, such as k being 1 (swapping the first and last elements), k being equal to half the list's length (in the case of an even number of nodes), or the list having k or fewer nodes.

Example Walkthrough

However, since the problem only asks to swap values, the provided solution handles all these cases without any additional

Let's walk through an example to illustrate the solution approach. Consider a linked list and an integer k:

Linked List: 1 -> 2 -> 3 -> 4 -> 5

k: 2

Initialize two pointers fast and slow to reference the head of the linked list. Move the fast pointer k-1 steps ahead. So, after this step:

slow: 2

Step 2:

Step 3:

fast: 5 (Last node)

checks.

```
fast: 2 (Second node)
slow: 1 (Head of the list)
```

Step 1:
fast: 3

Return the head of the modified linked list, which points to the node with value 1.

def swapNodes(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:

Store the kth node from the beginning in a temporary variable 'first_k_node'.

Initialize two pointers that will start at the head of the list.

// This pointer will eventually point to the kth node from the end

Start moving both fast and slow one step at a time until fast points to the last node:

Our task is to swap the 2nd node from the beginning with the 2nd node from the end.

fast: 4 slow: 3

```
4. At this point, we have:
p (kth from the beginning): Node with value 2
q (kth from the end): Node with value 4
5. Swap the values of nodes p and q:
Node `p` value before swap: 2
Node `q` value before swap: 4
Swap their values.
```

slow: 4 (This becomes our `k`th node from the end)

values from the start and end of a singly linked list in O(n) time complexity without any additional space required.

Solution Implementation

Python

class Solution:

Node `p` value after swap: 4

Node `q` value after swap: 2

The modified linked list now looks like this:

Modified Linked List: 1 -> 4 -> 3 -> 2 -> 5

fast_pointer = slow_pointer = head

first_k_node = fast_pointer

ListNode kthFromEnd = head;

fast = fast->next;

slow = slow->next;

let fast: ListNode | null = head;

let slow: ListNode | null = head;

// Move the 'fast' pointer to the kth node

fast = fast ? fast.next : null;

return head;

};

TypeScript

while (--k) {

// Now, slow points to the kth node from the end

std::swap(firstNode->val, secondNode->val);

// Return the head of the modified list

ListNode* secondNode = slow; // Keep reference to the kth node from the end

// Function to swap the kth node from the beginning with the kth node from the end in a singly—linked list

// Swap the values of the kth nodes from the beginning and end

function swapNodes(head: ListNode | null, k: number): ListNode | null {

// Initialize two pointers, both starting at the head of the list

class ListNode:
 def __init__(self, value=0, next_node=None):
 self.value = value
 self.next_node = next_node

This example illustrates the power of the two-pointer approach where pointer manipulation allows us to swap the kth nodes'

Move the fast pointer k-1 steps ahead, pointing to the kth node from the beginning. for _ in range(k - 1): fast_pointer = fast_pointer.next_node

```
# Move both pointers until the fast pointer reaches the end.
       # At that point, slow pointer will point to the kth node from the end.
       while fast_pointer.next_node:
            fast_pointer = fast_pointer.next_node
            slow_pointer = slow_pointer.next_node
       # Store the kth node from the end in a temporary variable 'second_k_node'.
        second_k_node = slow_pointer
       # Swap the values of the kth node from the beginning and the kth node from the end.
        first_k_node.value, second_k_node.value = second_k_node.value, first_k_node.value
       # Return the modified linked list head.
        return head
Java
/**
* Definition for singly-linked list.
 * public class ListNode {
      int val;
      ListNode next;
      ListNode() {}
 *
      ListNode(int val) { this.val = val; }
 *
      ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 *
* }
*/
class Solution {
    public ListNode swapNodes(ListNode head, int k) {
       // Fast pointer that will be used to locate the kth node from the beginning
       ListNode kthFromStart = head;
       // Move the kthFromStart pointer to the kth node
       while (--k > 0) {
            kthFromStart = kthFromStart.next;
```

```
// Fast pointer that will be used to reach the end of the list
       ListNode current = kthFromStart;
       // Move the current pointer to the end, maintaining the gap between
       // kthFromEnd and current, so that when current reaches the end,
       // kthFromEnd is at the kth node from the end
       while (current.next != null) {
           current = current.next;
           kthFromEnd = kthFromEnd.next;
       // Swap the values of the kth node from the start and end
       int tempValue = kthFromStart.val;
       kthFromStart.val = kthFromEnd.val;
       kthFromEnd.val = tempValue;
       // Return the modified list
       return head;
/**
* Definition for singly-linked list.
* struct ListNode {
      int val;
      ListNode *next;
      ListNode(): val(0), next(nullptr) {}
      ListNode(int x) : val(x), next(nullptr) {}
*
      ListNode(int x, ListNode *next): val(x), next(next) {}
*
* };
*/
class Solution {
public:
   ListNode* swapNodes(ListNode* head, int k) {
       // Initialize a pointer to move k nodes into the list
       ListNode* fast = head;
       while (--k) { // Move the fast pointer k-1 times
            fast = fast->next;
       // At this point, fast points to the kth node from the beginning
       // Initialize two pointers to find the kth node from the end
       ListNode* slow = head; // This will point to the kth node from the end
       ListNode* firstNode = fast; // Keep a reference to the kth node from the beginning
       // Move both pointers until the fast pointer reaches the end of the list
       while (fast->next) {
```

```
// 'starting' points to the kth node from the beginning
      // This node will later be swapped
      let starting = fast;
      // Move both 'fast' and 'slow' until 'fast' reaches the end of the list
      // 'slow' will then point to the kth node from the end
      while (fast && fast.next) {
          fast = fast.next;
          slow = slow ? slow.next : null;
      // 'ending' points to the kth node from the end
      let ending = slow;
      // If both nodes to swap have been found, swap their values
      if (starting && ending) {
          let temp = starting.val;
          starting.val = ending.val;
          ending.val = temp;
      // Return the head of the modified list
      return head;
class ListNode:
   def __init__(self, value=0, next_node=None):
        self.value = value
        self.next_node = next_node
class Solution:
   def swapNodes(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
       # Initialize two pointers that will start at the head of the list.
        fast_pointer = slow_pointer = head
       # Move the fast pointer k-1 steps ahead, pointing to the kth node from the beginning.
        for _ in range(k - 1):
            fast_pointer = fast_pointer.next_node
       # Store the kth node from the beginning in a temporary variable 'first_k_node'.
        first_k_node = fast_pointer
       # Move both pointers until the fast pointer reaches the end.
       # At that point, slow pointer will point to the kth node from the end.
       while fast pointer.next node:
            fast_pointer = fast_pointer.next_node
            slow_pointer = slow_pointer.next_node
       # Store the kth node from the end in a temporary variable 'second_k_node'.
        second_k_node = slow_pointer
       # Swap the values of the kth node from the beginning and the kth node from the end.
```

Time and Space Complexity The time complexity of the provided code snippet is O(n), where n is the length of the linked list. This is because the code

Return the modified linked list head.

return head

iterates over the list twice: once to find the k-th node from the beginning, and once more to find the k-th node from the end while simultaneously finding the end of the list.

The space complexity is 0(1) since only a constant amount of additional space is used, regardless of the input size. No extra data structures are created that depend on the size of the list; only a fixed number of pointers (fast, slow, and p) are used.

first k node.value, second k node.value = second k node.value, first k node.value