61. Rotate List

Medium Linked List Two Pointers

Leetcode Link

Problem Description

The problem presents a singly linked list and an integer `k`. The task is to move the last `k` nodes of the list to the front, essentially rotating the list to the right by `k` places. If the list has `n` nodes and `k` is greater than `n`, the rotation should be effective after `k modulo n` moves. If a list is empty or has a single node, it remains unchanged after rotation.

Intuition

To address this problem, we can follow a series of logical steps:

1. Determine the length of the list. This helps to understand how many rotations actually need to be performed since rotating a list by its length or multiples of `n` results in the same list.
2. Since rotating by `k` places where `k` is greater than the length of the list (let's call it `n`) is the same as rotating by `k modulo n` places, we calculate `k = k % n`. This simplifies the problem by ensuring that we don't perform unnecessary rotations.
3. Identify the `k-th` node from the end (or `n - k)-th` from the beginning after adjusting `k`) which after rotation will become the new head of the list. We use two pointers, `fast` and `slow`. We initially set both to the head of the list and move `fast` `k` steps forward.
4. We then advance both `fast` and `slow` pointers until `fast` reaches the end of the list. At this point, `slow` will be pointing to the node right before the `k-th` node from the end.
5. We update the pointers such that `fast`'s next points to the old head, making the old tail the new neighbor of the old head. The `slow`'s next becomes the new head of the rotated list, and we also need to set `slow`'s next to `None` to indicate the new end of the list.

Following this approach leads us to the rotated list which is required by the problem.

Solution Approach

The implementation of the solution uses the `two-pointer technique` along with an understanding of `linked list` traversal to achieve the rotation. Here is the step-by-step walk-through:

1. **Check for Edge Cases**: If the head of the list is `None` or if there is only one node (`head.next is None`), there is nothing to rotate, so we simply return the `head`.
2. **Calculate the Length (`n`)**: We traverse the entire list to count the number of nodes, storing this count in `n`. This traversal is done using a `while` loop that continues until the current node (`cur`) is `None`.
3. **Adjust `k` by Modulo**: Since rotating the list `n` times results in the same list, we can reduce `k` by taking `k` modulo `n` (`k %= n`). This simplifies the number of rotations needed to the minimum effective amount.
4. **Early Exit for `k = 0`**: If `k` becomes `0` after the modulo operation, this means the list should not be rotated as it would remain the same. Thus, we can return the `head` without any modifications.
5. **Initialize Two Pointers**: Start with two pointers, `fast` and `slow`, both referencing the `head` of the list. These pointers are going to help us find the new head after the rotations.
6. **Move `fast` Pointer**: Forward the `fast` pointer by `k` steps. Doing this will place `fast` exactly `k` nodes ahead of `slow` in the list.
7. **Move Both Pointers Until `fast` Reaches the End**: Now, move both `fast` and `slow` pointers simultaneously one step at a time until `fast` is at the last node of the list. Due to the initial `k` steps taken, `slow` will now be pointing to the `n-k-1-th` node or the one right before the new head of the rotated list.
8. **Perform the Rotation**:
   - The node following `slow` (`slow.next`) is the new head of the rotated list (`ans`).
   - To complete the rotation, we set the next node of the current last node (`fast.next`) to the old head (`head`).
   - To mark the new end of the list, we assign `None` to the next of `slow` (`slow.next = None`).

By following the above steps, we have rotated the list to the right by `k` places effectively and efficiently. As a result, the `ans` pointer, now referring to the new head of the list, is then returned as the final rotated list.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have a linked list `[1 -> 2 -> 3 -> 4 -> 5]` and we are given `k = 2`.

1. **Check for Edge Cases**: The list is not empty and has more than one node, so we can proceed.
2. **Calculate the Length (`n`)**: By traversing the list, we determine the length `n = 5`.
3. **Adjust `k` by Modulo**: Since `k = 2` is not greater than `n`, `k % n` is still `k`. Thus, `k` remains `2`.
4. **Early Exit for `k = 0`**: `k` is not `0`, so we do need to perform a rotation.
5. **Initialize Two Pointers**: We set both `fast` and `slow` to the head of the list. Currently, `fast` and `slow` are pointing to `1`.
6. **Move `fast` Pointer**: We advance `fast` by `k` steps: from `1` to `2`, then `2` to `3`. Now, `fast` is pointing to `3`, and `slow` is still at `1`.
7. **Move Both Pointers Until `fast` Reaches the End**: We advance both pointers until `fast` is at the last node:
   - Move `slow` to `2` and `fast` to `4`.
   - Move `slow` to `3` and `fast` to `5`.
   - Now `fast` is at `5`, the end of the list, and `slow` is at `3`.
8. **Perform the Rotation**:
   - The node after `slow` (`slow.next`), which is `4`, will become the new head.
   - We set `fast.next` (which is `5.next`) to the old head (`1`), forming a connection from `5` to `1`.
   - We update `slow.next` to `None` to indicate the new end of the list.

After performing rotation, the list now becomes `[4 -> 5 -> 1 -> 2 -> 3]` because the last two nodes (`4` and `3`) have been moved to the front of the list.

By returning the new head (`4` in this case), we conclude the rotation process and the modified list is correctly rotated by `k` places.

Python Solution

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        # If the list is empty or has just one element, no rotation is needed
        if head is None or head.next is None:
            return head

        # Count the number of elements in the linked list
        current, length = head, 0
        while current:
            length += 1
            current = current.next

        # Compute the actual number of rotations needed as k could be larger than the length of the list
        k %= length
        if k == 0:  # If no rotation is needed
            return head

        # Use two pointers, fast and slow. Start both at the beginning of the list
        fast = slow = head

        # Move the fast pointer k steps ahead
        for _ in range(k):
            fast = fast.next

        # Move both pointers at the same speed until fast reaches the end of the list
        while fast.next:
            fast, slow = fast.next, slow.next

        # At this point, slow is at the node before the new head after rotation
        # We can now adjust the pointers to complete the rotation
        new_head = slow.next
        slow.next = None  # The next pointer of the new tail should point to None
        fast.next = head  # The next pointer of the old tail should point to the old head

        return new_head  # Return the new head of the list

# Note: The Optional[ListNode] type hint should be imported from typing
# if you want to use it for type checking in Python 3.
# Otherwise, you can omit it for the function signatures.
```

Java Solution

```java
// Definition for singly-linked list.
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 rotateRight(ListNode head, int k) {
        // If the list is empty or has one node, no rotation needed
        if (head == null || head.next == null) {
            return head;
        }

        // Find the length of the linked list
        ListNode current = head;
        int length = 0;
        while (current != null) {
            length++;
            current = current.next;
        }

        // Normalize k in case it's larger than the list's length
        k %= length;

        // If k is 0, the list remains unchanged
        if (k == 0) {
            return head;
        }

        // Use two pointers: 'fast' will lead 'slow' by 'k' nodes
        ListNode fast = head;
        ListNode slow = head;
        while (k-- > 0) {
            fast = fast.next;
        }

        // Move both at the same pace. When 'fast' reaches the end, 'slow' will be at the k-th node from the end
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }

        // 'slow' is now at the node after which the rotation will occur
        ListNode newHead = slow.next;

        // Break the list at the node 'slow' and make 'fast' point to the original head to rotate
        slow.next = null;
        fast.next = head;

        // Return the new head of the rotated list
        return newHead;
    }
}
```

C++ Solution

```cpp
// 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* rotateRight(ListNode* head, int k) {
        // If the head is null or there is only one node, return the head as rotation isn't needed
        if (!head || !head->next) {
            return head;
        }

        // Find the length of the linked list
        ListNode* current = head;
        int length = 0;
        while (current) {
            length++;
            current = current->next;
        }

        // Normalize k to prevent unnecessary rotations if k >= length
        k %= length;

        // If k is 0 after modulo operation, no rotation is needed; return the original head
        if (k == 0) {
            return head;
        }

        // Set two pointers, fast and slow initially at head
        ListNode* fast = head;
        ListNode* slow = head;

        // Move fast pointer k steps ahead
        while (k--) {
            fast = fast->next;
        }

        // Move both slow and fast pointers until fast reaches the end of the list
        while (fast->next) {
            fast = fast->next;
            slow = slow->next;
        }

        // The slow pointer now points to the node just before rotation point
        // The fast pointer points to the last node of the list
        ListNode* newHead = slow->next; // This will be the new head after rotation
        slow->next = nullptr; // Break the chain to form a new end of the list
        fast->next = head; // Connect the original end of the list to the original head

        // Return the new head of the rotated list
        return newHead;
    }
};
```

Typescript Solution

```typescript
// Definition for singly-linked list.
class ListNode {
    val: number
    next: ListNode | null
    constructor(val: number = 0, next: ListNode | null = null) {
        this.val = val;
        this.next = next;
    }
}

/**
 * Rotates the list to the right by k places.
 * @param {ListNode | null} head - The head of the linked list.
 * @param {number} k - The number of places to rotate the list.
 * @return {ListNode | null} - The head of the rotated linked list.
 */
function rotateRight(head: ListNode | null, k: number): ListNode | null {
    // If the head is null or there is only one node, return the head as no rotation is needed.
    if (!head || !head.next) {
        return head;
    }

    // Find the length of the linked list.
    let current: ListNode = head;
    let length: number = 0;
    while (current) {
        length++;
        current = current.next;
    }

    // Normalize k to prevent unnecessary rotations if k is greater than or equal to length.
    k %= length;

    // If k is 0 after the modulo operation, no rotation is needed; return the original head.
    if (k === 0) {
        return head;
    }

    // Set two pointers, fast and slow, initially at the head.
    let fast: ListNode = head;
    let slow: ListNode = head;

    // Move the fast pointer k steps ahead.
    while (k-- > 0) {
        fast = fast.next;
    }

    // Move both slow and fast pointers until the fast reaches the end of the list.
    while (fast.next) {
        fast = fast.next;
        slow = slow.next;
    }

    // The slow pointer now points to the node just before the rotation point.
    // The fast pointer points to the last node in the list.
    let newHead = slow.next; // This will be the new head after rotation.
    slow.next = null; // Break the chain to form a new end of the list.
    fast.next = head; // Connect the original end of the list to the original head.

    // Return the new head of the rotated list.
    return newHead;
};
```

Time and Space Complexity

Time Complexity

The time complexity of the given Python code can be broken down into the following steps:

1. Iterate through the linked list to find out the length `n`. This process takes `O(n)` time as it goes through all elements of the linked list exactly once.
2. Calculate `k %= n`. This calculation is done in constant time, `O(1)`.
3. Moving the `fast` pointer `k` steps ahead. This again takes `O(k)` time, but since `k` in this case is always less than `n` after the modulo operation, we can say this operation takes at most `O(n)` time.
4. Moving both `fast` and `slow` pointers to find the new head of the rotated list: This has to traverse the remainder of the list, which takes at most `O(n-k)` time. However, in worst-case scenarios where `k` is `0`, this would result in `O(n)` time. Since the previous steps ensure that `k < n`, the combined operations will still be `O(n)`.
5. Re-link the end of the list to the previous head and set the `next` of the new tail to `None`: These operations are done in constant time, `O(1)`.

In all, considering that `O(n)` is the dominating factor, the overall time complexity of the code is `O(n)`.

Space Complexity

The space complexity of the code can also be analyzed:

1. The given algorithm only uses a fixed amount of extra space for variables `cur`, `n`, `fast`, and `slow`, regardless of the size of the input linked list.
2. No additional data structures are used that depend on the size of the input.
3. All pointer moves and assignments are done in-place.

Hence, the space complexity is `O(1)`, which means it requires constant extra space.