2074. Reverse Nodes in Even Length Groups

Medium | Linked List | Leetcode Link

Problem Description

The problem presents a scenario where we have a singly linked list, a data structure consisting of nodes, each with a value and a reference to the next node. The nodes in the list must be grouped according to the sequence of natural numbers i.e., the first node forms the first group, the next two nodes form the second group, the following three nodes form the third group, and so on. These groups' sizes increment by one, creating a naturally incremental grouping pattern across the linked list.

However, the issue involves a twist: we're asked to reverse the nodes only in those groups that have an even number of nodes. After executing these reversals, we need to return the new head of the modified linked list. It's crucial to notice that the last group's size may be smaller than or equal to the penultimate group, implying that the list may end before completing the size pattern of the sequence.

Intuition

The solution requires us to perform two primary operations:

1. Identify the groups within the linked list that are of even length.
2. Reverse the nodes in these even-length groups.

First, we note that the even-length groups are every other group, starting with the second one (sizes 2, 4, 6, etc.). To manage this, we can iterate through the list while keeping track of the current group size and the total nodes covered so far. Once we reach the end of a group, we review it. If the group size is even, and then proceed to the next group.

For reversing a group of given size k when present in the list, we temporarily detach this portion from the list, perform a standard linked list reversal, and then reconnect the reversed segment back to the main list.

To prevent issues with the final group (which may not conform to the natural number sequence if the list's length doesn't perfectly align with these group sizes), we check if there are enough nodes left for another group. If not, the remaining nodes form the last group, and we apply the even-length condition to decide if a reversal is needed.

The provided Python function `reverseEvenLengthGroups` follows this approach, utilizing a nested `reverse` helper function that specifically handles the reversal of k nodes in the sublist starting from `head`.

Lastly, to simplify handling edge cases such as the start and end of the list, a dummy node is employed, allowing for easy assignment and reference changes without additional null checks.

This approach requires updating node connections and occasionally counting nodes, both operations being $O(1)$, making the entire algorithm run in $O(n)$ time where n is the number of nodes in the list.

Solution Approach

The implementation employs a two-fold strategy—group detection and conditional reversal—based on simple list traversal and manipulation techniques.

To perform the list reversal operation, consider the `reverse` helper function defined within the `Solution` class. It takes two arguments: the `head` of the sublist to be reversed and the length `k` of the sublist. The function follows the classical linked list reversal approach:

1. Initialize three pointers: `prev` (set to `None`), `cur` (pointing to the `head` of the sublist), and `tail` (also pointing to the `head`). These pointers are for reversing the links of the sublist while keeping track of its tail.
2. Iterate `k` from 0 to `k - 1`. During each iteration, reverse the links by pointing the current node's `next` to its previous node (`prev`). Then, update `prev` to the current node and move `cur` to the next node in the original list.
3. Ensure the reversed sublist is reconnected to the main list by setting the `next` property of the `tail` to the current node `cur`, which now points to the sublist's successor in the main list.

The main part of the solution is structured as follows:

- The initial step involves determining the total number of nodes `n` in the original list. This is done via a while-loop that traverses the list and increments a counter for each node encountered.
- A `dummy` node is created with a value of 0 and its `next` reference pointing to the head of the list. This is a common trick to simplify edge case handling in linked list problems, particularly when modifications near the head are required.
- The actual traversal starts from the `dummy` node, using a pointer `prev` that keeps track of the end of the last processed group.
- The main while-loop executes as long as the starting index of the current group is within the bounds of the list. For each group iteration:
  - If `i` is even, indicating an even-length group, call the `reverse` function with `prev.next` as the starting node of the group. Then, update `prev.next` with the returned node, which is the new head of the reversed group. Independently of whether a group is reversed, a nested while-loop advances the `prev` pointer `i` times forward, which is equivalent to skipping over the current group.
  - Increment the group size `i` by 1 for the next iteration.
- Special attention is given for the last group, denoted by `i == 1`. If `left` is even and greater than zero, it implies that the last group of nodes also needs to be reversed. The same `reverse` function is called for this part.

Since the operations performed for each node are constant time, the overall time complexity of the solution is $O(n)$, where n is the number of nodes in the linked list.

Example Walkthrough

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

Suppose we have the following linked list:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7
```

According to the pattern described in the problem, the nodes should be grouped like so:

- Group 1: `1`
- Group 2: `2 -> 3` (even-length, should be reversed)
- Group 3: `4 -> 5 -> 6`
- Group 4: `7` (End of the list; this would be the part of the next group if we had more nodes)

Following the steps outlined:

1. We start by determining the group sizes. The first group has a size of `1`, so it remains unchanged. The linked list still looks like this: `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7`.
2. Next, we look at the second group which contains nodes `2` and `3`. Since it's even-length, we need to reverse it. After the reversal, our linked list becomes: `1 -> 3 -> 2 -> 4 -> 5 -> 6 -> 7`.
3. We move on to the third group, which contains `4`, `5`, `6`. Since this group has an odd number of nodes (`3`), we do not reverse them. The linked list remains: `1 -> 3 -> 2 -> 4 -> 5 -> 6 -> 7`.
4. Finally, we have the incomplete fourth group which would have had four members if the list were longer. Since it only contains `7`, and it's an odd-length group, we leave it as is.

Therefore, after applying the steps of the algorithm to the small example list, the modified linked list is:

```
1 -> 3 -> 2 -> 4 -> 5 -> 6 -> 7
```

From this example, we can see how the algorithm applies the group size pattern to decide where reversals are required and performs the reversal only on even-length groups using the `reverse` helper function. The use of pointers like `prev`, `cur`, and `tail` allows for the necessary list manipulations while traversing through the linked list only once, yielding an efficient $O(n)$ time complexity.

Python Solution

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

class Solution:
    def reverseEvenLengthGroups(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Helper function to reverse the linked list of a given length 'length'
        def reverse(head, length):
            previous, current = None, head
            count = 0
            while current and count < length:
                temp = current.next
                current.next = previous
                previous = current
                current = temp
                count += 1
            start.next = current
            return previous

        # Calculate the total number of nodes in the linked list
        node_count = 0
        temp_head = head
        while temp_head:
            temp_head = temp_head.next
            node_count += 1

        dummy = ListNode(-1, head)
        previous_group_end = dummy
        group_size = 1

        # Go through the list group by group
        while (1 + group_size) * group_size // 2 <= node_count and previous_group_end:
            # If the current group size is an even number, reverse the group
            if group_size % 2 == 0:
                previous_group_end.next = reverse(previous_group_end.next, group_size)

            # Move the pointer to the end of the current group
            for _ in range(group_size):
                if previous_group_end:
                    previous_group_end = previous_group_end.next

            # Increment the group size for the next iteration
            group_size += 1

        # Check if there are leftover nodes in the last group that form an even-length group
        remaining_nodes = node_count - group_size * (group_size - 1) // 2
        if remaining_nodes > 0 and remaining_nodes % 2 == 0:
            previous_group_end.next = reverse(previous_group_end.next, remaining_nodes)

        # Return the new head of the modified linked list
        return dummy.next
```

Java Solution

```java
class Solution {
    public ListNode reverseEvenLengthGroups(ListNode head) {
        int n = 0;
        // Calculate the total number of nodes in the linked list
        for (ListNode node = head; node != null; node = node.next) {
            ++n;
        }

        // Dummy node to simplify reversal operations
        ListNode dummy = new ListNode(0, head);
        ListNode groupPrev = dummy;

        // Group size starts from 1
        int currentGroupSize = 1;
        // Continue grouping and reversing while there are enough nodes remaining
        while ((currentGroupSize + 1) * currentGroupSize / 2 <= n && groupPrev != null) {
            // Reverse the group if its size is even
            if (currentGroupSize % 2 == 0) {
                ListNode nextGroupHead = groupPrev.next;
                groupPrev.next = reverse(nextGroupHead, currentGroupSize);
            }

            // Move the pointer to the end of the current group
            for (int i = 0; i < currentGroupSize && groupPrev != null; ++i) {
                groupPrev = groupPrev.next;
            }

            // Proceed to the next group
            currentGroupSize++;
        }

        // Calculate the number of nodes left for the last group
        int nodesLeft = n - currentGroupSize * (currentGroupSize - 1) / 2;
        // Perform group reversal if the number of nodes left is even
        if (nodesLeft > 0 && nodesLeft % 2 == 0) {
            ListNode nextGroupHead = groupPrev.next;
            groupPrev.next = reverse(nextGroupHead, nodesLeft);
        }

        // Return the head of the reversed list
        return dummy.next;
    }

    /**
     * Reverses a group of nodes in the linked list.
     * @param head The head node of the group to reverse.
     * @param groupSize The number of nodes to reverse.
     * @return new head of the reversed group.
     */
    private ListNode reverse(ListNode head, int groupSize) {
        ListNode prev = null;
        ListNode current = head;
        ListNode groupTail = current;
        int count = 0;

        // Perform reversal for groupSize nodes
        while (current != null && count < groupSize) {
            ListNode next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            ++count;
        }

        // Link the end of the reversed group to the rest of the list
        groupTail.next = current;
        return prev;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
};

// Function to reverse even length groups in a singly linked list.
ListNode* reverseEvenLengthGroups(ListNode* head) {
    // A vector to store the node values
    std::vector<int> nodeValues;
    ListNode* currentNode = head;

    // Traverse the linked list and collect node values.
    int totalNodes = 0;
    while (currentNode) {
        nodeValues.push_back(currentNode->val);
        currentNode = currentNode->next;
        totalNodes++;
    }

    int totalNodes = nodeValues.size();
    // Start processing groups.
    for (int groupSize = 1, startIndex = 0; startIndex < totalNodes; startIndex += groupSize, groupSize++) {
        // For the last group, adjust the size if it's shorter than expected.
        groupSize = std::min(totalNodes - startIndex, groupSize);
        // Check if the current group is of even length.
        if (groupSize % 2 == 0 && groupSize > 1) {
            // Reverse the even length group.
            std::reverse(nodeValues.begin() + startIndex, nodeValues.begin() + startIndex + groupSize);
        }
    }

    // Reset the current node to the head of the list.
    currentNode = head;
    // Update the list nodes with the reordered values.
    for (int val : nodeValues) {
        if (currentNode != nullptr) {
            // Set value to the current node.
            currentNode->val = val;
            // Move to the next node.
            currentNode = currentNode->next;
        }
    }

    // Return the modified linked list head.
    return head;
}
```

Typescript Solution

```typescript
// Function to reverse even length groups in a singly linked list.
function reverseEvenLengthGroups(head: ListNode | null): ListNode | null {
    // Node values are stored in this array.
    let nodeValues: number[] = [];
    let currentNode: ListNode | null = head;

    // Traverse the linked list and collect node values.
    while (currentNode) {
        nodeValues.push(currentNode.val);
        currentNode = currentNode.next;
    }

    const totalNodes: number = nodeValues.length;
    // Start processing groups.
    for (let groupSize = 1, startIndex = 0; startIndex < totalNodes; startIndex += groupSize, groupSize++) {
        // For the last group, adjust the size if it's shorter than expected.
        groupSize = Math.min(totalNodes - startIndex, groupSize);
        // Check if the current group is of even length.
        if (groupSize % 2 === 0 && groupSize > 1) {
            // Extract the subgroup of even length.
            let subGroup = nodeValues.slice(startIndex, groupSize);
            // Reverse the subgroup.
            subGroup.reverse();
            // Replace the original node values array.
            nodeValues.splice(startIndex, 0, ...subGroup);
        }
    }

    // Reset the current node to the head of the list.
    currentNode = head;
    // Update the list nodes with the reordered values.
    for (let val of nodeValues) {
        if (currentNode) {
            currentNode.val = val; // Set value to the current node.
            currentNode = currentNode.next; // Move to the next node.
        }
    }

    // Return the modified linked list head.
    return head;
}
```

Time and Space Complexity

Time Complexity

The given code consists of two main parts: first, finding the total number of nodes in the linked list, and second, reversing even-length groups.

1. Counting the nodes in the linked list has a time complexity of $O(n)$, where n is the total number of nodes. This is done with a simple while loop iterating through each node exactly once.
2. The second part involves iterating over the list again and for each group potentially reversing the nodes. Each node is involved in at most one reverse operation. The worst-case scenario is when all groups but the last are of even length. In this case, each node is touched twice: once for the actual reversal and once more when iterating through the groups. This implies a time complexity of $O(2n)$ for the worst case, simplified to $O(n)$.
3. The arithmetic calculation of $(i + 1) * i // 2$ is constant time for each group, since it doesn't depend on the list's length directly, but on the count of groups.

Combining these parts, the overall time complexity is $O(n + n)$, which simplifies to $O(n)$.

Space Complexity

The space complexity is determined by the amount of extra space used by the algorithm, not counting the input linked list.

1. We have a fixed number of variables (`prev`, `cur`, `i`, `tail`, `k`, `i`, `dummy`, and `left`) and a fixed-size stack for our recursive call to reverse the list. However, since the reversing is done iteratively within a while loop and not recursively, the space complexity is not directly affected by the size of the input list.
2. The `reverse` function uses a few temporary variables, but it does not allocate scaling additional space other than function call overhead, which is constant for each call since we are not using recursive calls.

Thus, the space complexity is $O(1)$ since it does not scale with the size of the input.

Overall, the code efficiently processes the list in linear time with constant extra space.