1474. Delete N Nodes After M Nodes of a Linked List

Easy Linked List Leetcode Link

Problem Description

In this problem, you are given a singly linked list and two integers `m` and `n`. The task is to modify the linked list by iterating through it and alternating between keeping and removing nodes. To be precise, you have to keep the first `m` nodes, then remove the next `n` nodes, and continue this pattern until the end of the list. The outcome should be the head of the linked list after these operations have been performed.

Intuition

The intuition behind the solution is to effectively use two pointers to traverse the linked list: one to mark the boundary of the nodes that will be retained (`pre`) and the other to find the boundary where the removal will stop (`cur`). The general approach is:

1. Keep a pointer to the current node we're looking at, starting with the head of the list.
2. Move this pointer `m-1` times forward (since we want to keep `m` nodes, we move `m-1` times to stay on the last node that we want to keep).
3. Establish another pointer at the same position and move it `n` times forward to find the last node that we want to remove.
4. Connect the `m`th node to the node right after the last removed node (effectively skipping over `n` nodes).
5. Continue this process until we reach the end of the list.

This algorithm is an in-place transformation which means we modify the original linked list without using extra space for another list.

Solution Approach

The solution to this problem follows a straightforward iterative approach, using a simple while loop to traverse the linked list, removing unwanted nodes as it goes. Below are the steps involved in the implementation:

1. Initialize a pointer named `pre` to point to the head of the list. This pointer will be used to track the node after which node deletion will start.

2. Use a while loop that continues until `pre` is not `None` (meaning we have not reached the end of the list).

3. Within the while loop, process keeping `m` nodes intact by iterating `m-1` times with a for loop. The `-1` is used because we are starting from the node `pre` which is already considered the first of the `m` nodes. If during this process `pre` becomes `None`, we exit the loop because we've reached the end of the list.

4. Now, we need a second pointer called `cur` which will start at the same position as `pre` and move `n` times forward to mark the count of nodes to be deleted.

5. After the for loop to remove `n` nodes, set `pre.next` to `cur.next`, this effectively skips over `n` nodes that are to be deleted. If `cur` is `None` at the end of the for loop, it means we reached the end of the list, and thus, we can safely set `pre.next` to `None`.

6. Finally, move `pre` to `pre.next` to process the next batch of nodes, beginning the m-n cycle again.

This solution employs no additional data structures, effectively making it an in-place operation with `O(1)` additional space complexity. The time complexity is `O(L)` where `L` is the number of nodes in the linked list, as each node is visited at most once.

The pattern used is the two-pointer technique, where one pointer (`pre`) is used to keep track of the node at the border between kept and removed nodes, and the other (`cur`) used to find the next node that `pre` should point to after removing `n` nodes.

A side note is that we have to manage the case when we reach the end of the list correctly. If `cur` becomes `None` after the removal loop, it means that we do not have any more nodes to process, and we should break out of the loop.

Example Walkthrough

Let's consider a linked list and use a small example to illustrate the solution approach. Suppose our linked list is `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8` and the integers `m` and `n` are given as `m = 2` and `n = 3`. This means we want to keep 2 nodes and then remove the next 3 nodes, repeating this process until the end of the list.

- Start with the head of the list. The list is `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8`.

- The pointer `pre` starts at the head node `1`. We iterate `m-1` times forward, which in this case is `1` time (since we are keeping two nodes, `1` and `2`).

- Now `pre` is at node `2`. Next, we set up `cur` to point to the same node as `pre`.

- We then move `cur` `n` times forward. Since `n` is `3`, we move `cur` to node `5`.

- The list now looks like this, where the brackets indicate the nodes that will remain: `[1 -> 2] 3 -> 4 -> 5 -> 6 -> 7 -> 8`.

- We connect the node at `pre` (which is node `2`) to `cur.next` (which is node `6`). Our list is now `1 -> 2 -> 6 -> 7 -> 8`.

- Now, we move `pre` to `pre.next` which points to node `6`. Then we repeat our process. Since there are fewer than `m` nodes left after `pre`, we do not continue and our final output is the list `1 -> 2 -> 6 -> 7 -> 8`.

Here are the steps in detail:

1. **Initialization**: `pre` points to `1`, and the list is `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8`.

2. **Iteration 1**:

   - Keep `m` nodes: `pre` traverses `1` node and still points to `2`.
   - Remove `n` nodes: `cur` starts at `2` and traverses `3` nodes, ending up at `5`.
   - Skip `n` nodes: Connect `2` to `6`.
   - The list is now `[1 -> 2] -> 6 -> 7 -> 8`.

3. **Move `pre` to `pre.next`**: `pre` now points to `6`.

4. **Iteration 2**:

   - Since there are fewer than `m` nodes left after `pre`, and we cannot remove more nodes, the process stops.

The final list, after the in-place modifications, is `1 -> 2 -> 6 -> 7 -> 8`.

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 deleteNodes(self, head: ListNode, m: int, n: int) -> ListNode:
        current_node = head

        # Iterate over the entire linked list
        while current_node:
            # Skip m nodes, these nodes will be retained
            for _ in range(m - 1):
                if current_node:
                    current_node = current_node.next
            # If we reach the end of the list, return the head as we don't have
            # more nodes to delete
            if current_node is None:
                return head

            # Now current_node points to the last node before the deletion begins
            to_delete = current_node.next

            # Skip n nodes to find the last node which needs to be deleted
            for _ in range(n):
                if to_delete:
                    to_delete = to_delete.next

            # Connect the current_node to the node following the last deleted node
            current_node.next = to_delete

            # Move to the next set of nodes
            current_node = to_delete

        return head  # Return the modified list
```

Java Solution

```java
class Solution {

    /**
     * Given a linked list, its head, and two integers m and n,
     * this function deletes every n nodes in the list after keeping m nodes.
     *
     * @param head The head of the singly-linked list.
     * @param m    The number of nodes to keep before deletion starts.
     * @param n    The number of nodes to delete.
     * @return The head of the modified list.
     */
    public ListNode deleteNodes(ListNode head, int m, int n) {
        // 'currentNode' is used to traverse the linked list starting from the head.
        ListNode currentNode = head;

        // Continue loop until 'currentNode' is null, which means end of the list.
        while (currentNode != null) {
            // Skip 'm' nodes but stop if the end of the list is reached.
            for (int i = 0; i < m - 1 && currentNode != null; ++i) {
                currentNode = currentNode.next;
            }

            // If 'currentNode' is null after the for loop, we return the head
            // as we reached the end of the list and cannot delete further nodes.
            if (currentNode == null) {
                return head;
            }

            // 'nodeToBeDeleted' points to the node from where we start deletion.
            ListNode nodeToBeDeleted = currentNode;

            // Advance 'nodeToBeDeleted' 'n' times or until the end of the list is reached.
            for (int i = 0; i < n && nodeToBeDeleted != null; ++i) {
                nodeToBeDeleted = nodeToBeDeleted.next;
            }

            // Connect the 'currentNode.next' to the node after the last deleted node.
            // If 'nodeToBeDeleted' is null, we've reached the end and thus set next to null.
            currentNode.next = (nodeToBeDeleted == null) ? null : nodeToBeDeleted.next;

            // Move 'currentNode' ahead to process the next chunk of nodes.
            currentNode = currentNode.next;
        }

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

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* deleteNodes(ListNode* head, int m, int n) {
        // previous_node will point to the last node before the sequence to be deleted
        ListNode* previous_node = head;

        // Continue iterating through the linked list until we reach the end
        while (previous_node) {
            // Skip m nodes, but retain the last one before deletion begins
            for (int i = 0; i < m - 1 && previous_node; ++i) {
                previous_node = previous_node->next;
            }

            // If we've reached the end, return the head as no deletion is needed
            if (!previous_node) {
                return head;
            }

            // Skip n nodes starting from previous_node->next for deletion
            ListNode* current = previous_node;
            for (int i = 0; i < n && current; ++i) {
                current = current->next;
            }

            // Connect the previous_node to the node after the n nodes to be deleted
            // If current is not nullptr, link to the node after the deleted sequence
            // If current is nullptr, it means we've reached the end of the list, so we set it to nullptr
            previous_node->next = (current ? current->next : nullptr);

            // Move the previous_node forward to start a new sequence
            previous_node = previous_node->next;
        }

        // Return the head of the modified list
        return head;
    }
};
```

Typescript Solution

```typescript
// Definition for singly-linked list node
class ListNode {
    val: number;
    next: ListNode | null;

    constructor(val: number = 0, next: ListNode | null = null) {
        this.val = val;
        this.next = next;
    }
}

/**
 * Deletes nodes in a linked list following a pattern: keep 'm' nodes then delete 'n' nodes, repeating this process.
 * @param head - The head of the singly-linked list.
 * @param m - The number of nodes to keep.
 * @param n - The number of nodes to delete.
 * @returns The head of the modified linked list.
 */
function deleteNodes(head: ListNode | null, m: number, n: number): ListNode | null {
    // previousNode will point to the last node before the sequence to be deleted
    let previousNode: ListNode | null = head;

    // Continue iterating through the linked list until the end is reached
    while (previousNode) {
        // Skip 'm' nodes, but retain the last one before deletion begins
        for (let i = 0; i < m - 1 && previousNode; i++) {
            previousNode = previousNode.next;
        }

        // If the end is reached, no further deletion is needed
        if (!previousNode) {
            return head;
        }

        // Skip 'n' nodes starting from previousNode.next for deletion
        let currentNode: ListNode | null = previousNode;
        for (let i = 0; i < n && currentNode != null; i++) {
            currentNode = currentNode.next;
        }

        // Connect previousNode to the node after the 'n' nodes to delete
        // If currentNode is not null, link to the node after the deleted sequence
        // If currentNode is null, set the next of previousNode to null (end of list)
        previousNode.next = currentNode ? currentNode.next : null;

        // Move previousNode forward to start a new sequence
        previousNode = previousNode.next;
    }

    // Return the head of the modified list
    return head;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code involves iterating through each node of the linked list using two nested loops controlled by `m` and `n`. The outer while loop goes through each node, but the iteration is controlled by the logic that deletes every `n` following `m` nodes. Each node is visited at most once due to the nature of the single pass through the linked list. The for loop running `m` times and the for loop running `n` times are sequential and do not multiply their number of operations since they operate on different sets of nodes. Therefore, the time complexity of this function is `O(m + n)`, for each group of `m+n` nodes, with a total of `O(T/m+n * (m + n))` = `O(T)` where `T` is the total number of nodes in the linked list.

Space Complexity

As there are no additional data structures that grow with the input size, and the given code only uses a fixed number of variables to keep track of the current and previous nodes, the space complexity is constant. Therefore, the space complexity is `O(1)`.