1669. Merge In Between Linked Lists

Medium Linked List Leetcode Link

Problem Description

In this problem, we are given two singly-linked lists called `list1` and `list2`. The sizes of these lists are `n` and `m` respectively. The goal is to modify `list1` by removing a segment of its nodes, specifically the nodes from the `a`th position to the `b`th position (assuming the first node is at position 0). Following the removal of this segment, `list2` is then inserted into `list1` at the cut point. In other words, `list1` should be continued by `list2` starting at the `a`th node, and after the last node of `list2`, the continuation should be the rest of `list1` starting from the node right after the `b`th position. The task is to complete this operation and return the head of the updated `list1`.

To visualize, imagine `list1` as a chain of nodes and we are to clip out a section of this chain from `a` to `b`, then attach a new chain (`list2`) in its place, and finally reattach the remaining part of the original `list1` after `list2`.

Intuition

To achieve the merge described in the problem, the solution involves a few key steps executed in sequence. The first step is to find the node just before the `a`th node in `list1`; let's call this the `prevA` node. We also need to find the `b`th node itself because its next node is where we want to eventually connect the tail of `list2`. Let's refer to the `b`th node's next node as `postB`. To navigate to these nodes, we can start at the head of `list1` and traverse it while counting the nodes until we reach the desired positions.

Once we have `prevA` and `postB`, we disconnect the nodes from `prevA` until `postB`, effectively removing the segmented list between `a` and `b`. Now `prevA`'s next node is set to the head of `list2`, linking the start of `list2` to the front portion of `list1`.

Next, we traverse to the end of `list2` since we need to connect the tail of `list2` to the `postB` node. After reaching the end of `list2`, we set the next node to `postB`.

The merge is complete at this point, and we return the head of the modified `list1`. The essence of the solution is to splice the arrays by reassigning the `next` pointers of the nodes in `list1`, to incorporate the entirety of `list2` and then reconnect `list1`.

Handling the node connections properly and ensuring no nodes are lost in the process are crucial parts of the solution.

Solution Approach

The merger of the two lists is achieved via a step-by-step approach:

1. **Initialize Pointers**: We start by initializing two pointers `p` and `q` to the head of `list1`. These pointers will help us traverse the list.
2. **Find `prevA` Node**: The `p` pointer is used to find the node just before the `a`th position (the `prevA` node). We use a simple loop that traverses the list `a-1` times. The loop `for _ in range(a - 1)` moves the `p` pointer to the correct spot.
3. **Find `postB` Node**: Similarly, the `q` pointer is aimed at finding the node at the `b`th position. Because we're already at the head of `list1` (position 0), we only need to move `b` times to reach this node, hence the loop: `for _ in range(b)`.
4. **Detach & Connect**: The `next` pointer of `p` is then set to the head of `list2`, effectively detaching the `list1` segment between `a` and `b`, and linking the beginning of `list2` to `list1`.
5. **Traverse `list2`**: Now, we need to find the end of `list2`. We continue to move `p` forward with the loop `while p.next`. When this loop exits, `p` is at the last node of `list2`.
6. **Reattach Remaining `list1`**: The next pointer of the last node of `list2` (now at `p`) is connected to `q.next`, which is the node immediately following the `b`th node in `list1` (the `postB` node). This is done with `p.next = q.next`.
7. **Complete and Return**: The `q.next` is then pointed to `None` to detach the removed segment from the rest of the list, which is a good practice to avoid potential memory leaks in some environments. Finally, the head of the modified list (which is still `list1`) is returned.

Here's a breakdown of key patterns used:

- **Two-pointer technique**: Used to locate the nodes before and after the removed segment.
- **Traversal**: An essential operation for navigating linked lists.
- **Link manipulation**: The core logic revolves around correctly adjusting the `next` properties of the nodes to "stitch" the lists together.

This approach guarantees the merger without allocating new nodes, operating in-place within the given data structures. It also ensures we only traverse each list once, making the algorithm efficient with O(n + m) time complexity, where `n` and `m` are the lengths of `list1` and `list2`, respectively.

Example Walkthrough

Let's illustrate the solution approach with a small example where `list1` = 1 → 2 → 3 → 4 → 5 and `list2` = 100 → 101. Suppose we want to replace nodes in positions `a` = 1 to `b` = 2 of `list1` with `list2`.

1. **Initialize Pointers**: We start by setting `p` and `q` to the head of `list1`, which is the node with value `1`.
2. **Find `prevA` Node**: We need to find the node just before the `a`th position (the `prevA` node). We move `p` one step because `a - 1 = 0`. So, `p` now points to node `1`.
3. **Find `postB` Node**: To locate the `postB` node, we set `q` to the head of `list1` and move it `b` steps. After moving 2 steps, `q` points to node `3`.
4. **Detach & Connect**: We set the next of node `1` (`prevA.next`) to the head of `list2` (node `100`). Now `list1` starts as 1 → 100 → 101.
5. **Traverse `list2`**: We move `p` through `list2` to the end. As `list2` has two nodes, `p` will now point to node `101`.
6. **Reattach Remaining `list1`**: Set `p.next` (currently `p` is at `101` of `list2`) to `q.next` (`q` is at `3` of `list1`), so that `list1` now is 1 → 100 -> 101 -> 3 -> 4 -> 5.
7. **Complete and Return**: Set `q.next` to `None`, detaching the removed segment (in this case, not needed as `q.next` already points to the correct segment). The head of `list1` remains the first node with value `1`, so we return `list1`.

Following this example, `list1` will be transformed into 1 → 100 → 101 → 3 → 4 → 5 after the operation, which demonstrates the solution approach in action.

Python Solution

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeInBetween(self, list1: ListNode, a: int, b: int, list2: ListNode) -> ListNode:
        # Initialize two pointers to the head of list1
        prev_node_of_sublist = curr_node = list1

        # Move prev_node_of_sublist to the node just before position 'a'
        for _ in range(a - 1):
            prev_node_of_sublist = prev_node_of_sublist.next

        # Move curr_node to the node at position 'b'
        for _ in range(b):
            curr_node = curr_node.next

        # Connect the node before 'a' with the head of list2
        prev_node_of_sublist.next = list2

        # Traverse to the end of list2 to find the last node
        while prev_node_of_sublist.next:
            prev_node_of_sublist = prev_node_of_sublist.next

        # Connect the last node of list2 with the node after 'b' in list1
        prev_node_of_sublist.next = curr_node.next

        # The node at position 'b' no longer has any references and can be collected by garbage collector

        # Return the merged list starting with list1's head
        return list1
```

Java Solution

```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 mergeInBetween(ListNode list1, int a, int b, ListNode list2) {
        // Initial pointers to help with node traversal.
        ListNode beforeA = list1; // Pointer to the node just before position 'a'.
        ListNode afterB = list1; // Pointer to the node just after position 'b'.

        // Move the 'beforeA' pointer to the node just before the 'a' position.
        for (int i = 0; i < a - 1; i++) {
            beforeA = beforeA.next;
        }

        // Move the 'afterB' pointer to the node just after the 'b' position.
        for (int i = 0; i < b; i++) {
            afterB = afterB.next;
        }

        // Connect the 'beforeA' node to the start of list2.
        beforeA.next = list2;

        // Traverse list2 to the end.
        ListNode endOfList2 = beforeA.next; // Start from the first node of list2
        while (endOfList2.next != null) {
            endOfList2 = endOfList2.next;
        }

        // Connect the end of list2 to the 'afterB' node, effectively skipping 'a' to 'b' in list1.
        endOfList2.next = afterB.next;

        // 'afterB.next' should be null to ensure we don't retain unwanted references.
        afterB.next = null;

        return list1; // Return the modified list1.
    }
}
```

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* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
        // Pointers to manage the positions in list1
        ListNode* beforeA = list1; // Pointer to track the node before the 'a' position
        ListNode* nextNode = list1; // Pointer to track the node at the 'b' position

        // Move the beforeA pointer to the node just before the node at position 'a'
        for (int i = 1; i < a; ++i) {
            beforeA = beforeA->next;
        }

        // Move the nextNode pointer to the node at position 'b'
        for (int i = 0; i <= b; ++i) {
            nextNode = nextNode->next;
        }

        // Attach the start of list2 to where 'a' was in list1
        beforeA->next = list2;

        // Traverse list2 until the end
        while (beforeA->next) {
            beforeA = beforeA->next;
        }

        // Connect the end of list2 to the node just after 'b' in list1
        beforeA->next = nextNode;

        // The next node of 'b' position is now isolated, and we do not need to set it to nullptr

        // Return the modified list1 with list2 merged in between
        return list1;
    }
};
```

Typescript Solution

```typescript
/**
 * Merges one linked list into another between the indices `a` and `b`. The nodes after `b`
 * are reconnected to the end of `list2`.
 * @param {ListNode | null} list1 - The first linked list.
 * @param {number} a - The start index for the merge.
 * @param {number} b - The end index for the merge.
 * @param {ListNode | null} list2 - The second linked list to be merged.
 * @returns {ListNode | null} - The merged linked list.
 */
function mergeInBetween(
    list1: ListNode | null,
    a: number,
    b: number,
    list2: ListNode | null
): ListNode | null {
    // `prevMergeNode` will eventually point to the node just before `a`.
    let prevMergeNode = list1;
    // `postMergeNode` will eventually point to the node just after `b`.
    let postMergeNode = list1;

    // Find the `(a-1)`th node, to connect `list2` to its next.
    while (--a > 0) {
        prevMergeNode = prevMergeNode!.next;
    }

    // Find the `b`th node, which `list2` will be connected before.
    while (b-- > 0) {
        postMergeNode = postMergeNode!.next;
    }

    // Connect `list2` to the next of `prevMergeNode`.
    prevMergeNode!.next = list2;

    // Iterate to the last node of `list2`.
    while (prevMergeNode!.next) {
        prevMergeNode = prevMergeNode!.next;
    }

    // Connect the last node of `list2` to the node after `postMergeNode`.
    prevMergeNode!.next = postMergeNode!.next;
    // Not necessary to nullify `postMergeNode.next` as it will not affect the resultant list.
    return list1;
}
```

Time and Space Complexity

Time Complexity

The given code consists of a few steps. Here is the analysis of each:

1. Advanced `p` pointer `a - 1` times: The time complexity is `O(a)` because it requires one operation for each step until reaching the `a`-th node.
2. Advanced `q` pointer `b` times: The time complexity is `O(b)` because it traverses the linked list from the start until reaching the `b`-th node.
3. Connecting `list1` to `list2`: The operation is constant time, `O(1)`, since it's a matter of single assignments.
4. Traversing `list2` to find the end: In the worst case, `list2` has `n` nodes, making this operation `O(n)`, where `n` is the number of nodes in `list2`.
5. Connecting the end of `list2` to `q.next`: This is another constant time operation, `O(1)`.

Adding these up, assuming `n` is the number of nodes in the second list and `a` and `b` are the positions in the first list, the overall time complexity would be `O(a) + O(b) + O(n) + O(1) + O(1)`, which simplifies to `O(a + b + n)`.

Space Complexity

The space complexity is `O(1)` because the code only uses a fixed number of pointers (`p` and `q`) and does not allocate extra space that grows with the size of the input.