92. Reverse Linked List II

Medium Linked List Leetcode Link

Problem Description

The problem presents us with a singly linked list and two integers, `left` and `right`, with the condition that `left <= right`. The goal is to reverse the nodes in the linked list that fall between the positions `left` and `right` (inclusive). The positions are 1-indexed, not 0-indexed, so the first node in the list would be position 1, the second node would be position 2, and so on. Ultimately, the modified linked list should be returned with the specified portion reversed while keeping the rest of the list's original structure intact.

Intuition

To tackle this problem, we need to understand the concept of reversing a linked list and also keeping track of the nodes at the boundaries of the section we want to reverse. The core idea is to iterate through the linked list to locate the node just before the `left` position (the start of our reversal) and the node at the `right` position (the end of our reversal).

Upon reaching the `left` position, we will begin the process of reversing the links between the nodes until we reach the `right` position. The key points are:

- We need a reference to the node just before the `left` position to reconnect the reversed sublist back to the preceding part of the list.
- We need to store the node at the `left` position as it will become the tail of the reversed sublist and connect to the node following the `right` position.

This can be achieved with a few pointers and careful reassignments of the `next` pointers within the sublist. By keeping track of the current node being processed and the previous node within the reversal range, we can reverse the links one by one. Finally, we must ensure we reattach the reversed sublist to the non-reversed parts properly to maintain a functioning linked list.

Solution Approach

The solution employs two essential steps: iterating to the specified nodes and reversing the sublist. Here, we use a dummy node to simplify edge case handling, such as when reversing from the first node.

1. **Initialization**
   - A dummy node is created with its `next` pointing to the head of the list. This helps manage the edge case where the `left` is 1, indicating the reversal starting from the head.
   - Two pointers, `pre` and `cur`, are initially set to the dummy and head nodes, respectively.
2. **Locating the Start Point**
   - We move the `pre` pointer `left - 1` times forward to reach the node just before where the reversal is supposed to start.
   - At this point, `pre.next` points to the first node to be reversed.
   - We also set a pointer `q` to mark the beginning of the sublist to be reversed (`pre.next`).
3. **Reversal Process**
   - A loop runs `right - left + 1` times, which corresponds to the length of the sublist to be reversed.
   - Within the loop, we perform the reversal. We constantly update the `cur.next` pointer to point to `pre`, effectively reversing the link between the current pair of nodes.
   - After reversing the link, we need to update the `pre` and `cur` pointers. `pre` moves to where `cur` used to be, and `cur` shifts to the next node in the original sequence using the temporary pointer `t` which holds the unreversed remainder of the list.
4. **Final Connections**
   - After the loop, the sublist is reversed. However, we still need to connect the reversed sublist back into the main list.
   - The pointer `p.next` is set to `pre`, which, after the loop termination, points to the `right` node that is now the head of the reversed sublist.
   - The initial `left` node, which is now at the end of the reversed sublist and pointed to by `q`, should point to the `cur` node, which is the node right after the `right` position or `None` if `right` was at the end of the list.
5. **Return the Result**
   - The function returns `dummy.next` as the new head of the list, ensuring that whether we reversed from the head or any other part of the list, we have the right starting point.

In summary, the solution takes a careful approach to change pointers and reconnect the nodes to achieve the desired reversal between the `left` and `right` indices, while keeping the rest of the list intact.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we have a linked list with elements 1 → 2 → 3 → 4 → 5, and we are asked to reverse the nodes between `left = 2` and `right = 4`. The positions of these elements in the list are: 1 → 2 → 3 → 4 → 5.

Following the solution approach step by step:

1. **Initialization**
   - Create a dummy node (`pre`) which points to the head of the list.
   - Initialize another pointer (`cur`) to the head which is the first node (`1`).
2. **Locating the Start Point**
   - Since `left = 2`, we move the `pre` pointer `left - 1` (1 time) forward, and it now points to node `1`.
   - The `pre.next` then points to node `2`, which is the start of the sublist we want to reverse.
   - We set a pointer `q` to mark the beginning of the sublist to be reversed (`pre.next`), so `q` points to node `2`.
3. **Reversal Process**
   - We need to reverse the sublist from position 2 to 4. The loop will run `right - left + 1` (`4 - 2 + 1 = 3` times).
   - In the first loop iteration, `cur` points to `2` and its `next` is `3`. We temporarily store the node following `cur` in pointer `t` (node `3`), then set `cur.next` to `pre` (node `1`), and update `pre` to `cur` (node `2`). Now `cur` points to `t` (node `3`).
   - We repeat this process for node `3` and `4`. Upon completion of the loop, our list looks like 1 → 2 ← 3 ← 4 with `pre` at `4` and `cur` pointing to `5`.
4. **Final Connections**
   - We need to make the final connections to integrate the reversed sublist with the rest of the list.
   - Connect `p.next` (the original start node `1`) to `pre` (which is now `4`), and `q.next` (which holds the start of the reversed sublist `2`) to `cur` (node `5` which is the remaining part of the list)
5. **Return the Result**
   - Return the `dummy.next`, which points to the new head of the list (node `1`).

The modified linked list will now look like 1 → 4 → 3 → 2 → 5, with the nodes between position `2` and `4` reversed.

Python Solution

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

class Solution:
    def reverseBetween(self, head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:
        # If the list only contains one node or no reversal is needed, return the head as is.
        if head.next is None or left == right:
            return head

        # Initialize a dummy node to simplify edge cases
        # where the reversal might include the head of the list.
        dummy = ListNode(0)
        dummy.next = head

        # This node will eventually point to the node right before
        # the reversal starts. Initialize it to the dummy node.
        predecessor = dummy

        # Move the predecessor to the node right before where
        # the reversal is supposed to start.
        for _ in range(left - 1):
            predecessor = predecessor.next

        # Initialize the 'reverse_start' node, which will eventually point to the
        # first node in the sequence that needs to be reversed.
        reverse_start = predecessor.next

        # The 'current' node will traverse the sublist that needs to be reversed.
        current = reverse_start

        # This loop reverses the nodes between 'left' and 'right'.
        # 'next_temp' is used to temporarily store the next node as we
        # rearrange pointers.
        for _ in range(right - left + 1):
            next_temp = current.next
            current.next = predecessor
            predecessor, current = current, next_temp

        # Link the nodes preceding the reversed sublist to the first node
        # in the reversed sequence.
        predecessor.next = predecessor

        # Link the last node in the reversed sublist to the remaining
        # part of the list that was not reversed.
        reverse_start.next = current

        # Return the new head of the list, which is the next of dummy node.
        return dummy.next

```

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 {

    /**
     * Reverses a section of a singly-linked list between the given positions.
     *
     * @param head The head of the linked list.
     * @param left The position from where to start the reversal (1-indexed).
     * @param right The position where to end the reversal (1-indexed).
     * @return The head of the modified linked list.
     */
    public ListNode reverseBetween(ListNode head, int left, int right) {
        // If there is only one node or no need to reverse, return the original list.
        if (head.next == null || left == right) {
            return head;
        }

        // Dummy node to simplify the handling of the head node.
        ListNode dummyNode = new ListNode(0, head);

        // Pointer to track the node before the reversal section.
        ListNode nodeBeforeReverse = dummyNode;
        for (int i = 0; i < left - 1; ++i) {
            nodeBeforeReverse = nodeBeforeReverse.next;
        }

        // 'firstReversed' will become the last node after the reversal.
        ListNode firstReversed = nodeBeforeReverse.next;
        // 'current' is used to track the current node being processed.
        ListNode current = firstReversed;
        ListNode prev = null;

        // Perform the actual reversal between 'left' and 'right'.
        for (int i = 0; i < right - left + 1; ++i) {
            ListNode nextTemp = current.next;
            current.next = prev;
            prev = current;
            current = nextTemp;
        }

        // Reconnect the reversed section back to the list.
        nodeBeforeReverse.next = prev;      // Connect with node before reversed part.
        firstReversed.next = current;       // Connect the last reversed node to the remainder of the list.

        // Return the new head of the list.
        return dummyNode.next;
    }
}

```

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* reverseBetween(ListNode* head, int left, int right) {
        // If there is only one node or no node to reverse
        if (!head || left == right) {
            return head;
        }

        // Create a dummy node to handle edge cases, such as reversing the head node
        ListNode* dummyNode = new ListNode(0);
        dummyNode->next = head;

        // Pointers for the node before the reversing part and the first node to reverse
        ListNode* preReverse = dummyNode;

        // Iterate to find the node before the left position
        for (int i = 0; i < left - 1; ++i) {
            preReverse = preReverse->next;
        }

        // Start reversing from the left position
        ListNode* current = preReverse->next;
        ListNode* nextNode = nullptr;
        ListNode* prev = nullptr;

        // Apply the reverse from left to right positions
        for (int i = 0; i < right - left + 1; ++i) {
            nextNode = current->next; // Save the next node to move on
            current->next = prev; // Reverse the link
            prev = current; // Move prev one step forward for the next iteration
            current = nextNode; // Move to the next node in the list
        }

        // Adjust the links for the node before left and the node right after the reversed part
        preReverse->next->next = current; // Connect the reversed part with the rest of the list
        preReverse->next = prev;          // Connect the start of the reversed list to the previous part

        // Return the new head of the list
        ListNode* newHead = dummyNode->next;
        delete dummyNode; // Clean up the memory used by dummyNode
        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;
    }
}

/**
 * Reverses a portion of the singly-linked list between positions 'left' and 'right'.
 *
 * @param {ListNode | null} head The head of the linked list.
 * @param {number} left The position to start reversing from (1-indexed).
 * @param {number} right The position to stop reversing at (1-indexed).
 * @return {ListNode | null} The head of the modified list.
 */
function reverseBetween(head: ListNode | null, left: number, right: number): ListNode | null {
    // Base case: If the sublist to reverse is of size 0, return the original list.
    const sublistLength = right - left;
    if (sublistLength === 0) {
        return head;
    }

    // Create a dummy node to handle edge cases seamlessly.
    const dummyNode = new ListNode(0, head);
    let previousNode: ListNode | null = null;
    let currentNode: ListNode | null = dummyNode;

    // Move the currentNode to the position right before where reversal begins.
    for (let i = 0; i < left; i++) {
        previousNode = currentNode;
        currentNode = currentNode.next;
    }

    // The previousNode now points to the node right before the start of the sublist.
    const sublistHeadPrev = previousNode;
    previousNode = null;

    // Reverse the sublist from the 'left' to 'right' position.
    for (let i = 0; i <= sublistLength; i++) {
        const nextNode = currentNode.next;
        currentNode.next = previousNode;
        previousNode = currentNode;
        currentNode = nextNode;
    }

    // Connect the reversed sublist back to the unchanged part of the original list.
    sublistHeadPrev.next.next = currentNode;
    sublistHeadPrev.next = previousNode;

    // Return the dummy node's next, which is the new head of the linked list.
    return dummyNode.next;
}

```

Time and Space Complexity

The time complexity of the given code can be determined by analyzing the number of individual operations that are performed as the input size (the size of the linked list) grows. The reversal operation within the section of the linked list bounded by `left` and `right` is the most significant part of the function.

- The code iterates from the dummy node to the node just before where the reversal starts (`left - 1` iterations), which is `O(left)`.
- Then it reverses the nodes between the `left` and `right` position, taking `right - left + 1` iterations, which is `O(right - left)`.

Assuming `n` is the total number of nodes in the linked list, the time complexity is the sum of the two:

`O(left) + O(right - left)`, which is equivalent to `O(right)`. Since `right` is at most `n`, the upper bound for the time complexity is `O(n)`.

For space complexity, the code only uses a fixed number of extra variables (`dummy`, `pre`, `p`, `q`, `cur`, and `t`), irrespective of the input size. These variables hold references to nodes in the list but do not themselves result in additional space that scales with the input size. Therefore, the space complexity is `O(1)`, meaning it is constant.