2816. Double a Number Represented as a Linked List

Medium Stack Linked List Math

Leetcode Link

Problem Description

You are provided with the head node of a non-empty singly linked list, which represents a non-negative integer written in reverse order — the first node represents the least significant digit, and there are no leading zeroes. The task is to return the head node of the modified linked list that represents the original number multiplied by two.

Intuition

The operation of doubling a number represented by a linked list is analogous to the process of multiplying an integer by two. However, since linked lists organize digits in reverse order and do not allow random access like arrays, we must handle the multiplication and carryover processes from the least significant digit (at the head of the reversed list) to the most significant digit in a linear fashion.

The solution to this problem involves these steps:

1. **Reverse the Linked List**: We begin by reversing the original linked list. This rearrangement ensures the head of our new list represents the most significant digit of the number, thereby allowing us to perform multiplication from left to right as we normally would on paper.
2. **Multiply Each Digit**: Going through each digit (node) from left to right, we multiply by two (as stated in the problem) and handle the carryover. For each node, we calculate the product of the node's value and two and add any carried-over value from multiplying the previous node.
3. **Handle Carries and Create Nodes**: Since the result of multiplication for a single digit can be a two-digit number, we keep track of the carry, which can be 0 or 1 because the largest possible product of two digits is 18 when doubling 9 with a carry of 1). The single digit of the current product is stored as the value of the node, and the tens digit is carried over to the next multiplication.
4. **Adding the Last Carry**: After all nodes have been processed, if there's a remaining carry (above 0), a new node will be created to hold that value.
5. **Reverse the Result**: Reverse the resulting list back so that it's in the original format required by the problem, with the head node representing the least significant digit.

This approach efficiently carries out the doubling operation, despite the constraints of using a linked list data structure.

Solution Approach

The solution follows the intuition described earlier. Here's a step-by-step walk-through correlating with the code provided:

1. **Reverse the Input Linked List**: The `reverse` function takes the head of a list as an argument and iteratively reverses the list. It uses a dummy head node to simplify the edge cases and proceeds by iterating over the list, re-linking the nodes in reverse order. At each step, the current node's next pointer is pointed to the node following the dummy, and then the dummy's next pointer is updated to the current node. This process is repeated until all nodes have been relinked in reverse order.
2. **Multiply the digits and handle carries**: The `doubleIt` function then initializes `mul = 2` for doubling and `carry = 0` for initially no carryover. It iterates over the reversed linked list:
   - For each node, it calculates `x = head.val * mul + carry`, where `head.val` is the current digit.
   - The carry is updated to `carry = x // 10`, which represents the tens place of the result (can be either 0 or 1).
   - A new node is created with the digit `x % 10` (the ones place of the result) and is linked to the result list.
3. **Add final carry if exists**: After multiplying all digits, if there's any remaining carry, a new node with this carry value is appended to the result list.
4. **Reverse the result to the correct order**: Finally, the function calls `reverse` on the result list to bring back the desired order, with the head node representing the least significant digit.

The algorithm makes use of simple list manipulation techniques, requires no additional data structure other than the linked list nodes themselves, and follows a straightforward simulation of the multiplication process on paper. The reversing pattern is essential for dealing with the list's inherent reversed digit order, and the linear iteration enables handling of carries in a direct, intuitive manner.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Assume we have a linked list representing the number `123` in reverse order, that is, the linked list is `3 -> 2 -> 1`. We want to apply the solution approach to return a new list that represents the number `246` in reverse order, which should be `6 -> 4 -> 2`.

Step 1: Reverse the Linked List We reverse the original list. Before: `3 -> 2 -> 1` After reversal: `1 -> 2 -> 3`

Step 2: Multiply Each Digit and Handle Carries We now iterate through the reversed list, multiplying each digit by two and handling any carry.

1. Starting from the head, multiply `1 * 2 = 2`. No carry, so the new list is `2`.
2. Next, `2 * 2 = 4`. No carry from the previous node, so we just add `4` to the new list: `2 -> 4`.
3. Finally, `3 * 2 = 6`. Again, no carry, so we add `6` to the list: `2 -> 4 -> 6`.

Since we have no carry left after the final multiplication, we can proceed to the next step.

Step 3: Add Final Carry if Exists There is no final carry in our case, so we can skip this step.

Step 4: Reverse the Result to the Correct Order Lastly, we reverse the linked list to return the digits to their original representation order while keeping the result doubled. Before: `2 -> 4 -> 6` After reversal: `6 -> 4 -> 2`

The final linked list represents the number `246` in reverse order as required. This completes our example, demonstrating how the algorithm doubles a number represented as a linked list.

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 doubleIt(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Helper function to reverse the linked list.
        def reverse(node):
            dummy = ListNode()
            current = node
            while current:
                next_node = current.next
                current.next = dummy.next
                dummy.next = current
                current = next_node
            return dummy.next

        # Step 1: Reverse the input list to make it easier to double the digits from the least significant digit.
        head = reverse(head)

        # Initialize a new dummy head for the result list.
        dummy = current = ListNode()
        multiplier = 2  # The factor by which we want to multiply each node's value.
        carry = 0  # Initialize the carry for multiplication.

        # Step 2: Traverse the reversed list, doubling each digit and managing the carry.
        while head:
            product = head.val * multiplier + carry  # Multiply the current value and add the carry.
            carry = product // 10  # Update carry for the next iteration.
            current.next = ListNode(product % 10)  # Create a new node with the single digit.
            current = current.next  # Move to the next position in the result list.
            head = head.next  # Move to the next node in the input list.

        # Step 3: If there is a carry left after the loop, add it as a new node.
        if carry:
            current.next = ListNode(carry)

        # Step 4: Reverse the result list back to original order and return.
        return reverse(dummy.next)
```

Java Solution

```java
class Solution {

    // Method to double the value of each node in a linked list
    public ListNode doubleIt(ListNode head) {
        // First, reverse the given linked list to start multiplication from the least significant digit
        head = reverse(head);
        // Create a dummy node to simplify result list construction
        ListNode dummy = new ListNode();
        ListNode current = dummy; // This will be used to add new nodes to the result list
        int multiplier = 2; // The value by which we want to multiply each digit
        int carry = 0; // To hold the carry value that results from multiplication

        // Traverse the reversed list to multiply each digit
        while (head != null) {
            int product = head.val * multiplier + carry; // Multiply the current node's value and add carry
            carry = product / 10; // Compute carry for the next iteration
            current.next = new ListNode(product % 10); // Store the single digit result in the new list
            current = current.next; // Move to the next position in the result list
            head = head.next; // Move to the next node in the input list
        }

        // If carry remains after processing all digits, add a new node with the carry value
        if (carry > 0) {
            current.next = new ListNode(carry);
        }

        // Finally, reverse the result list to restore original order and return
        return reverse(dummy.next);
    }

    // Helper method to reverse a singly-linked list
    private ListNode reverse(ListNode head) {
        ListNode dummy = new ListNode(); // Dummy node to simplify list reversal
        ListNode current = dummy; // Pointer to traverse the original list
        // Iterate over the original list and reverse the links
        while (current != null) {
            ListNode next = current.next; // Remember the next node
            current.next = dummy.next; // Make current node point to the beginning of the new reversed list
            dummy.next = current; // Update the beginning of the reversed list to be the current node
            current = next; // Move to the next node in the original list
        }
        // Return the reversed list, excluding the dummy node
        return dummy.next;
    }
}
```

C++ Solution

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

class Solution {
public:
    ListNode* doubleIt(ListNode* head) {
        // Step 1: Reverse the input list to make it easier to handle carry operations.
        head = reverse(head);

        // Create a dummy head to simplify the node insertion process.
        ListNode* dummy = new ListNode();
        ListNode* current = dummy;

        // Initialize multiplier to 2 for doubling the list values.
        int multiplier = 2;
        int carry = 0; // To handle carry over during addition.

        // Step 2: Traverse the reversed list, doubling each node's value.
        while (head) {
            int product = head->val * multiplier + carry;
            carry = product / 10;
            current->next = new ListNode(product % 10);
            current = current->next;
            head = head->next; // Move to the next node.
        }

        // Step 3: If there's a carry after the last multiplication, add a new node for it.
        if (carry) {
            current->next = new ListNode(carry);
        }

        // Step 4: Reverse the list again to restore the original order.
        return reverse(dummy->next);
    }

    // Helper function to reverse a linked list.
    ListNode* reverse(ListNode* head) {
        // Create a dummy head to simplify the node insertion process.
        ListNode* dummy = new ListNode();
        ListNode* current = head;

        // Traverse the list and reverse the links.
        while (current) {
            ListNode* next = current->next; // Store next node.
            // Insert the current node at the beginning of the reversed list.
            current->next = dummy->next;
            dummy->next = current;
            // Move to the next node in the original list.
            current = next;
        }

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

Typescript Solution

```typescript
// Node definition for a singly-linked list.
class ListNode {
    val: number;
    next: ListNode | null;

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

/**
 * Takes the head of a linked list, doubles the number it represents,
 * and returns the head of the modified linked list.
 * @param {ListNode | null} head - The head of the linked list.
 * @return {ListNode | null} - The head of the modified linked list.
 */
function doubleIt(head: ListNode | null): ListNode | null {
    // Reverse the linked list to process its digits from the least significant digit.
    head = reverse(head);
    const dummyHead = new ListNode();
    let current = dummyHead;
    const multiplier = 2; // The number to multiply each digit by.
    let carry = 0; // Initialize carry to 0 for addition.

    // Iterate through the linked list, doubling each digit and handling the carry.
    while (head) {
        const product = head.val * multiplier + carry;
        carry = Math.floor(product / 10);
        current.next = new ListNode(product % 10);
        current = current.next;
        head = head.next;
    }

    // If there is a carry left after the last digit, add a new node for it.
    if (carry) {
        current.next = new ListNode(carry);
    }

    // Reverse the result to restore the original order and return.
    return reverse(dummyHead.next);
}

/**
 * Reverses a singly-linked list and returns the new head.
 * @param {ListNode | null} head - The head of the linked list to be reversed.
 * @return {ListNode | null} - The head of the reversed linked list.
 */
function reverse(head: ListNode | null): ListNode | null {
    const dummyHead = new ListNode();
    let current = head;

    // Iterate through the list and reverse the pointers.
    while (current) {
        const nextNode = current.next;
        current.next = dummyHead.next;
        dummyHead.next = current;
        current = nextNode;
    }

    // Return the new head of the reversed list.
    return dummyHead.next;
}
```

Time and Space Complexity

The given function `doubleIt` modifies a linked list by doubling each element within it. The algorithm can be analyzed for time and space complexity as follows:

Time Complexity

The time complexity of the code is `O(n)` where `n` is the number of nodes in the linked list. This complexity arises from several operations that each take linear time:

1. The `reverse` function traverses all the nodes of the linked list once to reverse it. This operation is `O(n)`.
2. The next section of the code involving multiplication and building the new result linked list also takes linear time since it processes each node exactly once. Each operation inside the while loop takes constant time, and the loop runs for `n` iterations: `O(n)`.
3. The `reverse` function is called again to reverse the modified list back to the original order. This is another `O(n)` operation.

Since these operations are sequential, the overall time complexity is the sum of individual complexities: `O(n) + O(n) + O(n)` which simplifies to `O(n)`.

Space Complexity

For space complexity, we exclude the space taken by the input and output from consideration, as is standard in complexity analysis.

1. The `reverse` function uses a constant amount of extra space (a few pointers) that does not depend on the size of the list, so it is `O(1)`.
2. In the main part of the function, apart from the input and the result, we use a constant amount of variables (`dummy`, `cur`, `mul`, `carry`, `next`, `x`). Hence, the space complexity remains `O(1)`.

The reference answer confirms the analysis of both time complexity `O(n)` and the space complexity `O(1)` for the given code.