2487. Remove Nodes From Linked List

Medium | Stack | Recursion | Linked List | Monotonic Stack | Leetcode Link

Problem Description

In this problem, we are working with a singly linked list, where each node has an integer value. Our task is to modify the linked list by removing all nodes that are followed by nodes with greater values. To put it in another way, for every node in the linked list, if there exists a node to its right (i.e., further along the linked list) with a larger value, we must remove that node. The final linked list should only consist of nodes that do not have any higher valued nodes to their right. We are required to return the head of the modified linked list after performing the deletions.

Intuition

The intuition behind the solution is to process the nodes of the original linked list not from the beginning to the end but in reverse order. When traversing the list from right to left, we can keep track of the maximum value seen so far, and only keep nodes with values greater than this maximum. This way, we ensure that all remaining nodes do not have any nodes with greater values to their right by the definition of our traversal.

To implement this, we first convert the linked list to an array (`nums`), allowing us to access elements in reverse order. Then we use a stack (`stk`) to help us identify the nodes to keep. When traversing the array in reverse, we compare the current node's value with the top element of the stack (if the stack is not empty). If the current node's value is larger, we pop elements from the stack until we find a larger value or the stack becomes empty. This enforces that all elements left in the stack do not have greater values to their right. After that, we add the current value to the stack.

Ultimately, we use the values in the stack to rebuild the linked list from left to right. The stack itself now acts as a blueprint for our final modified linked list, preserving the order of the nodes that have been kept. We start with an empty sentinel node (`dummy`) and then append all nodes from the stack to this sentinel node. Finally, we return the `next` node of the sentinel node, which represents the head of our modified list.

Solution Approach

The solution follows a two-step approach: transforming the linked list into an array and then processing that array using a stack to build our final linked list.

Firstly, we initialize an empty array named `nums`. We then iterate over the linked list starting from the `head` and append each node value to the `nums` array. This conversion allows easier manipulation of nodes in reverse order, which is essential for solving the problem.

Next, we use a stack to keep track of the nodes that will remain in the linked list after removal of the others. Starting with an empty list named `stk`, we iterate over the `nums` array. For each value `v`, we perform the following steps:

- We check if the stack is not empty and the top element is less than `v`.
- If so, we continuously pop elements from the stack until we find an element greater than `v` or the stack is empty. This ensures that `v` is the greatest element thus far and will be included in the final list.
- After ensuring that `stk` only contains elements greater or equal to `v`, we append `v` to `stk`.

By this process, we effectively reverse the `nums` array and filter out values such that no element has a greater value following it.

After processing the entire `nums` array in this way, `stk` contains all the values that should appear in the modified linked list, in reverse order. To turn this stack into a linked list, we create a dummy head node `dummy` to serve as a non-value holding starting point. We then iterate over the `stk` array, and for each value `v`, we append a new node with value `v` to the end of the list starting at the `dummy` node.

Thus, we are able to create the new linked list with all nodes not followed by a node with greater value—to the right—all in one pass through the `stk`. The final step is to return `dummy.next`, which points to the head of our new modified linked list.

The overall process utilizes a conversion to array for ease of traversal, a stack for filtering necessary nodes, and a re-conversion to linked list. This approach is efficient since it traverses the list of nodes only twice, regardless of the number of removals necessary.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider the following singly linked list:

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

According to the problem statement, we want to keep only the nodes that do not have a higher valued node to their right. In this case, since the list is ordered in ascending fashion, all nodes but the last one would be removed, leaving us with:

`5`

Let's apply the solution approach step by step.

Converting Linked List to Array

First, we initialize an empty array `nums` and iterate from the head to the end of the linked list, appending each node's value to `nums`.

After this step, `nums` will be: `[1, 2, 3, 4, 5]`.

Processing the Array Using a Stack

We initialize a stack `stk` and iterate over `nums` in reverse order (from right to left):

1. Start with `5`. Since `stk` is empty, we add `5` to `stk`.
2. Move to `4`, compare with the top element of `stk` (which is `5`). Since `4` is less than `5`, we discard `4`.
3. The same logic applies to `3`, `2`, and `1`. Since all are less than the top element `5`, they are all discarded.

After this process, `stk` contains just `[5]`.

Rebuilding the Linked List

Now, we create a sentinel node `dummy` and start creating new linked list nodes using the values in `stk`.

Since `stk` has one element, `5`, we create a node with the value `5` and link it to the `dummy`.

Finally, we return `dummy.next` which points to the head of the modified list.

The modified list will be a single node with the value:

`5`

Thus, our example list has been modified in place to remove all nodes that have a larger valued node to their right, leaving us with the correct result according to the problem description.

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 removeNodes(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # List to store the values of the linked list nodes
        node_values = []

        # Traverse the linked list and store the values in node_values list
        while head:
            node_values.append(head.val)
            head = head.next

        # Stack to maintain the required condition and store the final values
        stack = []
        for value in node_values:
            # If the current value is greater than the last value in the stack,
            # pop the last value to ensure monotonically increasing order
            while stack and stack[-1] < value:
                stack.pop()
            # Append the current value to the stack
            stack.append(value)

        # Create a dummy node to serve as a starting point for the new linked list
        dummy = ListNode()
        current = dummy

        # Convert the stack to a linked list
        for value in stack:
            current.next = ListNode(value)
            current = current.next

        # Return the head of the newly formed linked list
        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 {
    // Function to remove nodes from the linked list that have a value greater to the right
    public ListNode removeNodes(ListNode head) {
        // Initialize an array to store the node values
        List<Integer> nodeValues = new ArrayList<>();

        // Iterate through linked list to collect values
        while (head != null) {
            nodeValues.add(head.val);
            head = head.next;
        }

        // Use a stack to keep track of nodes to be preserved
        Deque<Integer> stack = new ArrayDeque<>();
        for (int value : nodeValues) {
            // Remove all elements from stack that are smaller than current value
            while (!stack.isEmpty() && stack.peek() < value) {
                stack.pop();
            }
            // Push the current value onto the stack
            stack.push(value);
        }

        // Create a dummy node to build the resulting linked list
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        // Build the new linked list by popping values from the stack
        while (!stack.isEmpty()) {
            // Since stack is LIFO, use stack.pollLast() to maintain original order of remaining nodes
            current.next = new ListNode(stack.pollLast());
            current = current.next;
        }

        // Return the next node of dummy since first node is a placeholder
        return dummy.next;
    }
}

```

C++ Solution

```cpp
// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}            // Default constructor
    ListNode(int x) : val(x), next(nullptr) {}       // Constructor with value parameter
    ListNode(int x, ListNode *next) : val(x), next(next) {}  // Constructor with value and next node parameters
};

class Solution {
public:
    ListNode* removeNodes(ListNode* head) {
        vector<int> values; // This will hold the values of the nodes of the linked list

        // Traverse the linked list and fill the vector with node values
        while (head) {
            values.emplace_back(head->val);
            head = head->next;
        }

        vector<int> stack; // This is used to keep track of the nodes to keep

        // Iterate over the values and use the stack to filter the values
        for (int value : values) {
            // If the last value in the stack is less than the current value, pop from stack
            while (!stack.empty() && stack.back() < value) {
                stack.pop_back();
            }
            // Push the current value onto the stack
            stack.push_back(value);
        }

        // Create a dummy node to build the new linked list
        ListNode* dummy_head = new ListNode();
        ListNode* current = dummy_head; // Pointer to iterate through the new list

        // Construct the new linked list from the stack values
        for (int value : stack) {
            current->next = new ListNode(value);
            current = current->next;
        }

        // Return the head of the new linked list, which is after the dummy node
        return dummy_head->next;
    }
};

```

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;
    }
}

// Function to remove nodes according to the specific criteria described in the original C++ function.
function removeNodes(head: ListNode | null): ListNode | null {
    // This array will hold the values of the nodes of the linked list.
    const values: number[] = [];

    // Traverse the linked list and fill the array with node values.
    while (head) {
        values.push(head.val);
        head = head.next;
    }

    // This array is used to keep track of the nodes to keep.
    const stack: number[] = [];

    // Iterate over the values and use the stack to filter the values.
    for (let value of values) {
        // If the last value in the stack is less than the current value, pop from stack.
        while (stack.length > 0 && stack[stack.length - 1] < value) {
            stack.pop();
        }
        // Push the current value onto the stack.
        stack.push(value);
    }

    // Create a dummy node to build the new linked list.
    const dummyHead: ListNode = new ListNode();
    // Pointer to iterate through the new list.
    let current: ListNode = dummyHead;

    // Construct the new linked list from the stack values.
    for (let value of stack) {
        current.next = new ListNode(value);
        current = current.next;
    }

    // Return the head of the new linked list, which is after the dummy node.
    return dummyHead.next;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several loops over the input list:

1. The first loop that constructs the `nums` list is `O(n)` where `n` is the size of the linked list since it visits each node exactly once.
2. The second loop uses a while loop inside a for loop to process each element and possibly multiple elements within the stack (`stk`). In the worst case, each element is pushed and popped exactly once, which still yields `O(n)` because each element is handled twice at most (once for pushing and once for popping).
3. The third loop creates the new linked list from the stack, and this is also `O(n)` since each element in the stack is visited once.

Thus, the overall time complexity is `O(n) + O(n) + O(n) = O(n)`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm:

1. The `nums` list holds all the values from the linked list at once, which requires `O(n)` space.
2. The `stk` stack can potentially hold all the values (if they are sorted in increasing order), so it also requires `O(n)` space.
3. The new linked list that is constructed is also counted toward the space complexity, but since the original list is traversed as well, this does not contribute additional space overhead in terms of complexity analysis.

The total space complexity is `O(n) + O(n) = O(n)` since the space required for the stack and the space required for the nums list are additive.