1171. Remove Zero Sum Consecutive Nodes from Linked List

Medium Hash Table Linked List

Leetcode Link

Problem Description

In this problem, we are provided with the head node of a singly linked list, which is a data structure where each node contains a value and a reference to the next node. Our task is to remove all consecutive sequences of nodes from this linked list that, when their values are added together, sum up to `0`.

The challenge lies in the possibility of such sequences being spread across the linked list and not necessarily being adjacent immediately. Furthermore, after removing a sequence that sums to `0`, new sequences can form, also leading to a sum of `0`, which should be removed as well. This process must be repeated until there are no more sequences with a sum of `0` remaining.

For example, if we have a linked list `1 -> 2 -> -1 -> 3 -> 1`, we can see that the nodes `2 -> -3 -> 3` sum up to `2 + (-3) + 3 = 0`. If we remove these nodes, the linked list becomes `1 -> 1`. The requirement is to keep removing such sequences until no such sequence exists, and in this case, we would finally end up with an empty linked list as `1 -> 1` also sums up to `0`.

Intuition

To solve this problem, we use a two-pass approach leveraging the concept of a prefix sum and a hash table. The prefix sum for a node in the linked list is the sum of all node values from the head up to and including that node. If at any point, we find two nodes with the same prefix sum, the nodes between them sum to `0`, and thus that sequence can be removed.

For example, suppose that for a linked list `1 -> 2 -> -3 -> 3 -> 1`, the prefix sum at the first `1` is `1`, then `3` at `2`, back to `0` after `-3`, up to `3` at the second `3`, and finally `4` at the last `1`. The appearance of `3` as a prefix sum implies that the subsequence from the beginning up to that point sums to `0`.

Here's our approach step by step:

1. We create a dummy node that we'll use as a new head, which helps us deal with edge cases where the beginning of the linked list might sum to `0` and be removed.
2. We iterate through the list, calculating the prefix sum for each node. We keep a track of the last seen node for each prefix sum in a hash table. If the same prefix sum is encountered again, we update it in the hash map with the most recent node. This is because any node sequences between the two nodes with the same sum can be removed, so we only care about the latest position for that prefix sum.
3. After populating the hash table, we make a second pass through the list, again calculating the prefix sum. Using our hash table, we can now update the `next` pointer of the current node to skip over any nodes that are part of a zero-sum sequence by setting it to the `next` of the last node remembered for the current prefix sum.

With this approach, we effectively go over the list, find all sequences summing to `0`, and remove them, then return the list with all those sequences gone.

Solution Approach

The provided solution makes use of the prefix sum concept and a hash table to efficiently find and remove zero-sum consecutive sequences in the linked list. Here's how the implementation unfolds:

1. We initialize a `dummy` node that acts as a pre-head, ensuring we can handle cases where the head itself might be part of a zero-sum sequence. This `dummy` node points to the original `head` of the list.

```
dummy = ListNode(next=head)
```

2. We declare a hash table `last` to record the last node for each unique prefix sum observed. The hash table is indexed by the prefix sum and contains the corresponding node as its value.
3. Starting from the `dummy` node (pre-head), we iterate through the linked list to calculate the prefix sum `s` for each node. As we compute the prefix sum, we update the `last` hash table with the current node. If the same sum occurs again later, it overwrites the previous node, since we only need the latest one.

```
s, cur = 0, dummy
while cur:
    s += cur.val
    last[s] = cur
    cur = cur.next
```

4. After populating the `last` table, we iterate the list again, starting from the `dummy` node, to update the `next` pointers. For each node, as we calculate the prefix sum `s`, we find the node corresponding to this sum in the `last` table, and we set the current node's `next` pointer to `last[s].next`. This effectively skips over and removes any nodes part of a zero-sum sequence found between the two nodes with equal prefix sums.

```
s, cur = 0, dummy
while cur:
    s += cur.val
    cur.next = last[s].next
    cur = cur.next
```

5. Finally, we return `dummy.next` — the head of the modified list, which no longer contains any sequence of nodes that sum up to `0`.

The two primary components used in the solution are:

- **Prefix Sum**: This technique is critical to discover sequences that total to `0`. By keeping track of the cumulative sum at each node, we can swiftly identify regions of the list that cancel each other out.
- **Hash Table**: By storing the last occurrence of a node for a given prefix sum, we have the ability to quickly jump over sequences that sum to `0`. This is because if a prefix sum repeats, the sum of the nodes between those repetitions is necessarily `0`.

Together, these structures allow the algorithm to achieve this goal with a linear time complexity relative to the number of nodes in the list, since each node is processed directly without the need for nested loops or recursion.

Example Walkthrough

Let's consider a simple linked list to illustrate the solution approach: `3 -> 4 -> -7 -> 5 -> -6 -> 6`. The goal is to remove sublists that sum to `0`.

1. **Initialize a `dummy` node and a `last` hash table**

   We create a `dummy` node and initialize our `last` hash table to store the last node associated with each unique prefix sum. Initially, the `dummy` node points to the head of our list (`3`).

2. **First pass: Compute prefix sums and populate the `last` table**

   Starting from the `dummy` node, the `prefix sum (s)` and `last` table will be updated as follows:

   ```
   1) s: 0 -> dummy node (There's always a dummy node associated with prefix sum 0)
   2) s: 3 -> 3 (first node with value 3)
   3) s: 7 -> 4 (then with value 4)
   4) s: 0 -> -7 (then with value -7, this gives us a prefix sum of 0, meaning everything from the dummy to this node sums to 0)
   5) s: 5 -> 5 (We continue the process from the node with value 5)
   6) s: -1 -> -6
   7) s: 5 -> 6 (The last node, summing up to 5)
   ```

   Notice how the prefix sum returned to `0` when we included the `-7` node, implying the sublist `3 -> 4 -> -7` sums to `0` and should be removed.

3. **Second pass: Update the `next` pointers using the `last` table**

   We iterate through the list again, using the `last` table to update `next` pointers. We perform the following updates, recalculating the prefix sum `s`:

   ```
   1) s: 0, dummy.next -> last[0].next (skipping to node with value 5)
   2) s: 5, 5.next -> last[5].next (skipping to node with value 6)
   ```

   After the second pass, the list becomes `5 -> 6`, since the first part `3 -> 4 -> -7` has been skipped. We then continue and find that `5 --> -6 --> 6` also sum to `0` and should be removed.

4. **Final list**

   After all updates, the prefix sums that led to a non-zero result have been removed, and we are left with an empty list, as the entire list was a combination of zero-sum sublists. The `dummy.next` now points to `None`.

By using the prefix sum and a hash table to keep track of the last occurrence of each prefix sum, we have efficiently removed consecutive sequences that sum to `0`. The resulting list would be returned as the modified list without these zero-sum sequences.

Python Solution

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, value=0, next_node=None):
        self.value = value
        self.next = next_node

class Solution:
    def removeZeroSumSublists(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Create a dummy node which points to the head of the list. This helps with edge cases.
        dummy_node = ListNode(next_node=head)

        # Dictionary to keep track of the prefix sums and their corresponding nodes.
        prefix_sums = {}

        # Calculate the prefix sums and store the most recent node that resulted in each sum.
        current_sum, current_node = 0, dummy_node
        while current_node:
            current_sum += current_node.value
            prefix_sums[current_sum] = current_node
            current_node = current_node.next

        # Reset the sum and traverse the list again to remove zero-sum sublists.
        current_sum, current_node = 0, dummy_node
        while current_node:
            current_sum += current_node.value
            # Set the next node to be the node that follows the last occurrence of the current sum.
            current_node.next = prefix_sums[current_sum].next
            current_node = current_node.next

        # The dummy node's next now points to the head of the adjusted list.
        return dummy_node.next
```

Java Solution

```
class Solution {
    public ListNode removeZeroSumSublists(ListNode head) {
        // Dummy node to serve as a new starting point for the linked list
        ListNode dummyHead = new ListNode(0, head);
        // HashMap to store the cumulative sum and corresponding node
        Map<Integer, ListNode> cumulativeSumMap = new HashMap<>();

        int sum = 0; // Variable to hold the cumulative sum of node values
        ListNode current = dummyHead; // Current node, starting from the dummy head

        // First pass: Calculate cumulative sums and save the last occurrence
        // of each sum in the HashMap
        while (current != null) {
            sum += current.val;
            cumulativeSumMap.put(sum, current);
            current = current.next;
        }

        sum = 0; // Reset the sum for the second pass
        current = dummyHead; // Reset current node to the dummy head

        // Second pass: Remove zero-sum sublists
        while (current != null) {
            sum += current.val; // Update the cumulative sum

            // If we have seen this sum before, it means the sublist between the
            // previous occurrence and this one sums to zero
            current.next = cumulativeSumMap.get(sum).next;

            current = current.next; // Move to the next node
        }

        return dummyHead.next; // Return the updated list without the dummy head
    }
}
```

C++ Solution

```
#include <unordered_map>

// 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* removeZeroSumSublists(ListNode* head) {
        // Create a dummy head node to handle edge cases seamlessly
        ListNode* dummyHead = new ListNode(0, head);
        std::unordered_map<int, ListNode*> lastSeenPrefixSum;
        int prefixSum = 0, sumToRemove;
        lastSeenPrefixSum[0] = dummyHead;

        // First pass to record the last occurrence of each prefix sum
        while (current) {
            prefixSum += current->val;
            lastSeenPrefixSum[prefixSum] = current;
            current = current->next;
        }

        // Reset the prefix sum and re-iterate from the dummy head
        prefixSum = 0;
        current = dummyHead;

        // Second pass to connect nodes bypassing the zero-sum sublists
        while (current) {
            prefixSum += current->val;
            // Link the current node to the node following the last occurrence of the same prefix sum
            // This effectively removes the zero-sum sublists
            current->next = lastSeenPrefixSum[prefixSum]->next;
            current = current->next;
        }

        // Return the next element of the dummy head which would be the new list head
        return dummyHead->next;
    }
};
```

Typescript Solution

```
// Type definition for a ListNode.
type ListNode = {
  val: number;
  next: ListNode | null;
};

/**
 * Removes all contiguous sublists with sum zero from a linked list.
 * @param head The head of the singly linked list.
 * @return The head of the modified linked list, with zero-sum sublists removed.
 */
function removeZeroSumSublists(head: ListNode | null): ListNode | null {
  // Dummy node at the start of the list to simplify edge cases.
  const dummy: ListNode = { val: 0, next: head };
  // A map to store the cumulative sum of nodes and their last occurrences.
  const lastOccurrenceOfSum: Map<number, ListNode> = new Map();
  let sum = 0;

  // First pass: Compute the cumulative sum and track the last occurrence of each sum.
  for (let currentNode: ListNode | null = dummy; currentNode; currentNode = currentNode.next) {
    sum += currentNode.val;
    lastOccurrenceOfSum.set(sum, currentNode);
  }

  sum = 0; // Reset sum for the second pass.

  // Second pass: Use the last occurrence map to skip over zero-sum sublists.
  for (let currentNode: ListNode | null = dummy; currentNode; currentNode = currentNode.next) {
    sum += currentNode.val;
    // The next node will be the one after the last occurrence of the current sum.
    // As we have removed zero-sum sublists, the sums will not repeat in the new list.
    currentNode.next = lastOccurrenceOfSum.get(sum)!.next;
  }

  // Return the modified list, sans the dummy node.
  return dummy.next;
}
```

Time and Space Complexity

The provided code has a time complexity of `O(n)` where `n` is the length of the linked list. This is because the code consists of two separate while loops that each iterate through the list once. The first while loop constructs a dictionary (last) mapping the cumulative sums of the nodes up to that point to the corresponding node. The second while loop uses the dictionary to skip over nodes that are part of a zero-sum sublist by setting the `next` pointer of previous non-zero-sum nodes to the next node in the last occurrence of that sum.

The space complexity of the code is `O(n)` as well. The primary factor contributing to the space complexity is the dictionary (`last`) which stores a value for each unique cumulative sum encountered while iterating through the list. In the worst-case scenario, there could be as many unique sums as there are nodes in the list (when no sublists sum to zero), which would require storing each node in the dictionary, hence `O(n)` space is needed.