1836. Remove Duplicates From an Unsorted Linked List

Medium Hash Table Linked List Leetcode Link

Problem Description

In this problem, we are given the head of a singly linked list. Our task is to find all the values within this linked list that appear more than once. Once we have identified such values, we are then to delete all the nodes containing any of those values from the linked list. The goal is to return the linked list after we've made all the necessary deletions.

Intuition

To solve this problem, we need a way to track the frequency of each value present in the linked list. A common and efficient way to do this is by using a hash table, also known as a dictionary in Python, where the keys are the values from the linked list and the corresponding values are the counts of occurrences for each key.

The solution approach involves two main steps. First, we need to traverse the entire linked list to populate the hash table with the correct counts for each value. With the completed hash table, we can then identify which values appear more than once in the linked list.

In the second step, we need to traverse the linked list again and this time, delete nodes that have a count greater than one. This amounts to checking the hash table for the count of the current node's value. If it's greater than one, it means this value appears multiple times and hence the node should be deleted. We need to carefully update the next pointers of the nodes that are not deleted to ensure we have a properly linked list at the end.

A dummy node is typically used as an anchor to manage the head of the list during deletion, especially in cases where the head node itself might need to be deleted. This dummy node initially points to the head of the list, and we start our iteration from the head while keeping track of the previous node as well. If a node needs to be deleted, we can bypass it by setting the `next` pointer of the previous node to the current node's `next`. If a node doesn't need to be deleted, we just move the previous pointer to the current node.

After iterating through the entire list and making necessary deletions, the dummy's `next` pointer points to the head of the modified list, which we return as the final result.

Solution Approach

The implementation of the solution begins with importing `Counter` from the `collections` module in Python. The `Counter` class provides a convenient way to count hashable objects. It is essentially a dictionary where elements are stored as dictionary keys and their counts are stored as dictionary values.

Here's the step-by-step breakdown of the implementation:

1. **Initialize the Counter**: An instance of the `Counter` is created, which will keep track of the number of occurrences of each element in the list. This is done by the line `cnt = Counter()`.

2. **First Pass - Count the Occurrences**: The first traversal of the list occurs here, fulfilling the responsibility of counting occurrences of each value. We continue traversing the list until we reach the end (`cur` is `None`). During the traverse, we increment the count of the current value `cur.val` by doing `cnt[cur.val] += 1`. This forms the frequency mapping required to identify duplicates.

3. **Setup the Dummy Node and Pointers**: A dummy node is created with `ListNode(0, head)`. This node is a placeholder to help manage deletions, especially when the head of the list might need to be deleted. The `pre` pointer is set to the dummy node and `cur` is reset to `head`.

4. **Second Pass - Delete Duplicates**: The list is traversed again. This time we have our frequency map ready, and hence for each node, we check if its value appears more than once by verifying `cnt[cur.val] > 1`. If this condition holds, it means the node is a duplicate and should be removed from the list. To delete the current node `cur`, we set the `next` pointer of the previous node `pre` to the `next` of the current, effectively bypassing the current node in the list.

   However, if the current node's value does not appear more than once, we need to keep it, and simply update the `pre` pointer to reference the current node. After either of these checks is performed, we move the current pointer `cur` to the next node in the list (`cur.next`).

5. **Return Modified List**: After the traversal is complete and we have erased all the nodes that had duplicate values, the list is now modified. The `dummy.next` holds the reference to the new head of the list, which is returned as the final result.

This algorithm effectively solves the problem using O(n) time complexity for the two traversals and O(n) space complexity for the counter.

Example Walkthrough

Let's assume we have a singly linked list with the following values: 3 → 4 → 4 → 2 → 3 → 1 → 2.

Our objective is to identify all values that appear more than once and then remove all nodes containing any such values. Following our solution approach, here's an example walkthrough:

1. **Initialize the Counter**: We start by creating an empty `Counter` object: `cnt = Counter()`.

2. **First Pass - Count the Occurrences**: We traverse the list and count the occurrences of each value. The counter after this pass will look like this: `cnt = {3: 2, 4: 2, 2: 2, 1:1}`.

3. **Setup the Dummy Node and Pointers**: We create the dummy node and set up our pointers. Now, we have `dummy -> 3 -> 4 -> 4 -> 2 -> 3 -> 1 -> 2`, with `pre` pointing to `dummy` and `cur` pointing to the first node with value `3`.

4. **Second Pass - Delete Duplicates**: We begin traversing the list again. Here's how we process each node:

   - Look at `cur` node with value `3`, `cnt[3] > 1`, it is a duplicate, so we update `pre.next` to `cur.next`, bypassing the current `3`.
   - `cur` now points to the first `4`.
   - Look at `cur` node with value `4`, `cnt[4] > 1`, it is a duplicate, so we bypass it.
   - `cur` now points to the second `4`, we repeat the above step.
   - `cur` now points to the first `2`, and we bypass it, as `cnt[2] > 1`.
   - `cur` now points to the second `3`, and we bypass it, as `cnt[3] > 1`.
   - `cur` now points to `1`, since `cnt[1] == 1`, it's not a duplicate, `pre` is updated to reference this node.
   - `cur` moves to the second `2`, which is bypassed, as `cnt[2] > 1`.
   - List traversal is now complete.

5. **Return Modified List**: At this point of the walkthrough, only the node with value `1` remains, and the updated list points to it. So, the `dummy.next` is pointing to the node with value `1`, which is now the head of our resulting list.

Hence, after the algorithm finishes, the linked list that we return will only contain the node with value `1`.

Python Solution

```python
from collections import Counter

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

class Solution:
    def deleteDuplicatesUnsorted(self, head: ListNode) -> ListNode:
        # Create a Counter to keep track of the frequency of each value in the linked list.
        value_counter = Counter()

        # Traverse the linked list to populate the counter with the frequencies of each value.
        current_node = head
        while current_node:
            value_counter[current_node.val] += 1
            current_node = current_node.next

        # Start with a dummy node that points to the head of the list.
        # This simplifies edge cases such as deleting the head node.
        dummy_node = ListNode(0, head)

        # Initialize two pointers, 'previous' and 'current'.
        # 'previous' will lag one behind 'current' as we traverse the list.
        previous = dummy_node
        current = head

        # Traverse the linked list again, this time to remove duplicates.
        while current:
            # If the current node's value has a count greater than 1, it's a duplicate.
            if value_counter[current.val] > 1:
                # Therefore, skip this node by setting the previous node's next to be the next node.
                previous.next = current.next
            else:
                # If it's not a duplicate, move the 'previous' pointer up to be the 'current' node.
                previous = current
            # Move the current pointer to the next node in the list.
            current = current.next

        # Return the modified list, starting from the dummy head's next value
        return dummy_node.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 {
    public ListNode deleteDuplicatesUnsorted(ListNode head) {
        // HashMap to store the frequency of each value in the list
        Map<Integer, Integer> valueCount = new HashMap<>();

        // First pass: count the occurrences of each value
        ListNode current = head;
        while (current != null) {
            valueCount.put(current.val, valueCount.getOrDefault(current.val, 0) + 1);
            current = current.next;
        }

        // Dummy node to simplify edge cases at the head of the list
        ListNode dummy = new ListNode(0, head);

        // Second pass: remove nodes with values that appear more than once
        ListNode previous = dummy; // Maintain the node before the current node
        current = head; // Start again from the head of the list
        while (current != null) {
            // If current node's value count is more than 1, skip it
            if (valueCount.get(current.val) > 1) {
                previous.next = current.next;
            } else {
                // Only move the previous pointer if current node is unique
                previous = current;
            }
            current = current.next; // Move to the next node in the list
        }

        // Return the next node of the dummy, which is the new head of the modified list
        return dummy.next;
    }
}
```

C++ Solution

```cpp
// Definition for a singly-linked list node.
struct ListNode {
    int val;
    ListNode *next;
    // Constructor to initialize a node with a value and next pointer (default nullptr).
    ListNode(int x = 0, ListNode *nextNode = nullptr) : val(x), next(nextNode) {}
};

class Solution {
public:
    // Function to delete nodes with duplicate values from an unsorted linked list.
    ListNode* deleteDuplicatesUnsorted(ListNode* head) {
        // An unordered map to store the count of each value in the list.
        unordered_map<int, int> valueCounts;

        // First pass: count occurrences of each value in the list.
        for (ListNode* currentNode = head; currentNode != nullptr; currentNode = currentNode->next) {
            valueCounts[currentNode->val]++;
        }

        // Dummy node to facilitate deletion from the head of the list.
        ListNode dummyNode;

        // Previous node pointer starts from dummy node; current node starts from head.
        ListNode *previousNode = &dummyNode, *currentNode = head;

        // Second pass: remove nodes with values that have more than one occurrence.
        while (currentNode != nullptr) {
            // If the current value exists more than once, skip the current node.
            if (valueCounts[currentNode->val] > 1) {
                previousNode->next = currentNode->next;
            } else {
                // Only move the previous node pointer if the current value isn't duplicated.
                previousNode = currentNode;
            }
            // Move to the next node in the list.
            currentNode = currentNode->next;
        }

        // Return the new list starting at the node after the dummy node.
        return dummyNode.next;
    }
};
```

Typescript Solution

```typescript
/**
 * Function to delete all duplicates from an unsorted singly-linked list
 * @param {ListNode | null} head - The head of the singly-linked list
 * @returns {ListNode | null} - The modified list with duplicates removed
 */
function deleteDuplicatesUnsorted(head: ListNode | null): ListNode | null {
    // Map to store the frequency count of each value in the list
    const frequencyCount: Map<number, number> = new Map();

    // Count the occurrences of each value by traversing the list
    for (let currentNode = head; currentNode !== null; currentNode = currentNode.next) {
        const value = currentNode.val;
        frequencyCount.set(value, (frequencyCount.get(value) ?? 0) + 1);
    }

    // Create a dummy node that points to the head of the list
    const dummyHead = new ListNode(0, head);

    // Traverse the list with two pointers, `previousNode` and `currentNode`, connected as: previousNode -> currentNode
    for (
        let previousNode = dummyHead, currentNode = head;
        currentNode !== null;
        currentNode = currentNode.next
    ) {
        // Check the frequency count of the currentNode's value
        if (frequencyCount.get(currentNode.val)! > 1) {
            // If count is more than 1, it is a duplicate, remove it by updating the next pointer of the previous node
            previousNode.next = currentNode.next;
        } else {
            // If current value is not a duplicate, move previousNode pointer to the current node
            previousNode = currentNode;
        }
    }

    // Return the modified list, omitting the dummy head
    return dummyHead.next;
}

// ListNode class definition for reference
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);
    }
}
```

Time and Space Complexity

The provided code aims to remove all nodes that have duplicate values from an unsorted singly-linked list. The algorithm works in two passes. The first pass counts the occurrences of each value using a counter (`cnt`), and the second pass removes nodes with values that occur more than once.

- **Time complexity**: The time complexity of the code is `O(n)` where `n` is the length of the linked list. This is because each node in the list is visited exactly twice – once while counting the occurrences (first `while` loop) and once while removing duplicates (second `while` loop). Both operations for each node take constant time, so the total time is linear with respect to the number of nodes in the list.

- **Space complexity**: The space complexity of the code is also `O(n)`. This is due to the use of a counter (`cnt`) to store the occurrence count for each value present in the linked list. In the worst case, if all `n` nodes have unique values, the counter will need to store an entry for each value, resulting in `O(n)` space used.