3063. Linked List Frequency

Easy Hash Table Linked List Counting

Problem Description

In this problem, we are given the head of a singly linked list that contains an unspecified number of nodes. Each node in the linked list contains an integer value, and all the elements in the linked list are said to be distinct. Our objective is to create a new linked list that represents the frequency of each distinct element found in the original linked list. The length of the new linked list will be equal to the number of distinct elements (`n`) in the original list, and the values of nodes in the new list are the frequencies of the corresponding elements, unordered.

For instance, if the original list contained the elements [1,2,1,2,1,3], there are 3 distinct elements, and the new list should contain [3,2,1] to represent the frequencies of elements 1, 2, and 3, respectively. It is important to note that the elements in the new list output do not need to be in any particular order and represent the count of how many times each element occurs in the original list.

Intuition

To solve this problem, we utilize a hash table (dictionary in Python) to keep track of the frequency or count of each element as we traverse the original linked list. Since the elements in the linked list are distinct, each key in the hash table uniquely identifies an element, and the associated value represents the frequency of that element.

The process begins by initializing an empty hash table and then iterating through each node in the linked list until we reach the end. For each node, we increment the count of the node's value in the hash table. After completing the traversal, we will have a complete record of the frequencies of all the distinct elements in the original list.

Next, we generate a new linked list from the frequency counts stored in the hash table. For each frequency count in the hash table, we create a new node with that frequency as its value and insert it at the beginning of the new linked list. This approach conveniently constructs the new linked list in reverse order, which is acceptable because the problem statement does not require the frequencies to be in a specific order.

In summary, the solution involves two primary steps:

1. Traverse the original linked list and record the frequency of each element.
2. Traverse the hash table and construct the new linked list using the frequencies recorded.

The time complexity of this algorithm is linear, O(n), because we iterate through each element of the original linked list once. The space complexity is also linear, O(k), where k is the number of distinct elements in the list, for storing the frequency counts in the hash table.

Solution Approach

The implementation of the solution follows a straightforward approach, making use of two core concepts—hash tables and linked lists. Below is an explanation of how each part of the solution is brought together:

Hash Table to Record Frequencies

A hash table is used to map unique elements to their frequency. In Python, the `collections.Counter` class is an excellent fit for this purpose, as it automatically initializes dictionary keys with a default count of 0.

As we iterate over the original linked list, we increment the count for the value of each node:

```
cnt = Counter()
while head:
    cnt[head.val] += 1
    head = head.next
```

For every node in the linked list, `head.val` is used to reference the node's value, which serves as the key in the hash table `cnt`. The corresponding value in the hash table is incremented to keep track of the frequency of the element.

Constructing the New Linked List

The next step is to construct the new linked list using the frequencies stored in the hash table. We create a dummy head node to simplify the process of adding new nodes to the linked list. This allows us to insert nodes without checking if the list is empty:

```
dummy = ListNode()
```

We iterate over the hash table's values, creating a new node for each frequency:

```
for val in cnt.values():
    dummy.next = ListNode(val, dummy.next)
```

Here, we're inserting each new node at the beginning of the new linked list so that the dummy's `next` always points to the latest node created. The order doesn't matter according to the problem statement.

By the end of this loop, we have a new linked list that starts right after the dummy node. The new list contains the frequencies of distinct elements as its node values.

Finally, we return the `next` node of `dummy`, which is the head of the new linked list containing the frequencies:

```
return dummy.next
```

In conclusion, the solution combines the efficient counting using a hash table with simple list manipulation to generate the required output. The elegance of the solution lies in its utilization of Python's built-in `Counter` to handle the frequency tracking and the dummy node that circumvents the need for special cases while constructing the new list.

Example Walkthrough

Let's go through an example to illustrate the solution approach with a small sample linked list. Suppose our original linked list is `4 -> 2 -> 4 -> 3`, representing the elements [4, 2, 4, 3]. We want to create a new linked list that shows how frequently each distinct element occurs.

Step 1: Using a Hash Table to Record Frequencies

First, we initialize an empty hash table (Counter) and begin iterating over the linked list. Here's how the counter changes as we traverse the list:

1. After encountering the first element (4), the counter is `{4: 1}`.
2. The second element is (2), updating the counter to `{4: 1, 2: 1}`.
3. The third element is again (4), so we increment its count: `{4: 2, 2: 1}`.
4. Finally, we encounter (3), and the counter becomes `{4: 2, 2: 1, 3: 1}`.

Now we have a hash table that contains the frequency of each distinct element from the original list.

Step 2: Constructing the New Linked List

With the frequency counts stored, we proceed to create the new linked list:

1. We initialize a new linked list with a dummy head.
2. Iterate over the counter values: (2, 1, 1).
3. For each frequency, we create a new node and add it to the front of the list.

Here are the steps visualized:

- New node with value `2` → new list is: `dummy -> 2`
- New node with value `1` → new list is: `dummy -> 1 -> 2`
- New node with value `1` → new list is: `dummy -> 1 -> 1 -> 2`

We've inserted each new node at the beginning, so after we're done, our new list (ignoring dummy) looks like: `1 -> 1 -> 2`.

Remembering the order is irrelevant; this new list represents the frequencies of elements 4, 2, and 3 respectively: [2, 1, 1]. Now, we return the first actual node after the dummy, which is the head of the new linked list containing the frequencies.

Therefore, given the original linked list `4 -> 2 -> 4 -> 3`, our algorithm correctly outputs a new linked list with elements representing the frequencies `[2, 1, 1]`. The number 4 appears twice, and numbers 2 and 3 appear once in the original list.

Solution Implementation

Python

```
from collections import Counter

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

class Solution:
    def frequenciesOfElements(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Use Counter to count occurrences of each element
        count = Counter()

        # Iterate through the linked list and count frequencies
        current_node = head
        while current_node:
            count[current_node.val] += 1
            current_node = current_node.next

        # Create a dummy node to make adding new nodes easier
        dummy_node = ListNode()
        tail = dummy_node  # 'tail' will be used to keep track of the last node

        # Iterate through the frequencies and create a new linked list
        # with the frequency values of each element in the same order they appear in the count
        for frequency in count.values():
            # Add a new node with the frequency to the new list
            tail.next = ListNode(frequency)
            tail = tail.next  # Move the tail to the new last node

        # Return the head of the new linked list which contains the frequencies
        return dummy_node.next
```

Java

```
import java.util.HashMap;
import java.util.Map;

/**
 * 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 {
    /**
     * Calculates frequency of each element in the linked list
     * and returns a linked list of frequencies.
     *
     * @param head The head of the input linked list.
     * @return The head of a new linked list representing frequencies of elements.
     */
    public ListNode frequenciesOfElements(ListNode head) {
        // HashMap to store the frequency count of each element in the list
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        // Traverse the linked list and update the frequency count for each element
        ListNode current = head;
        while (current != null) {
            // If the element is already in the map, increment the frequency,
            // otherwise add the element with frequency 1
            frequencyMap.merge(current.val, 1, Integer::sum);
            // Move to the next node
            current = current.next;
        }

        // Dummy node as a placeholder to start building the result list
        ListNode dummy = new ListNode();
        ListNode resultTail = dummy;

        // Iterate over the frequencies in the map to create a new list
        for (int frequency : frequencyMap.values()) {
            // Insert each frequency at the front of the list
            resultTail.next = new ListNode(frequency);
            resultTail = resultTail.next;
        }

        // Return the head of the new list which contains the frequencies
        return dummy.next;
    }
}
```

C++

```
#include <unordered_map>

// Definition for singly-linked list.
struct ListNode {
    int val;  // The value stored within the node
    ListNode *next;  // Pointer to the next node in the list
    ListNode() : val(0), next(nullptr) {} // Default constructor initializing 'val' to 0 and 'next' to nullptr
    ListNode(int x) : val(x), next(nullptr) {} // Constructor initializing 'val' with given value and 'next' to nullptr
    ListNode(int x, ListNode *next) : val(x), next(next) {} // Constructor initializing both 'val' and 'next' with given values
};

class Solution {
public:
    ListNode* frequenciesOfElements(ListNode* head) {
        // An unordered map that will store frequency of each element
        unordered_map<int, int> elementFrequencies;

        // Iterate over the linked list and count the occurrences of each element
        for (ListNode* cur = head; cur != nullptr; cur = cur->next) {
            elementFrequencies[cur->val]++;
        }

        // A dummy node that simplifies list operations by removing the necessity to handle special cases for head.
        ListNode* dummyHead = new ListNode();

        // Adding the frequency values to a new list, maintaining the order of the elements
        // Looping through the map and creating a new node with a frequency value for each element
        for (auto& [elementFrequency, a] : elementFrequencies) {
            dummyHead->next = new ListNode(elementFrequency, a, dummyHead->next);
        }

        // The actual head of the new list is the next of dummy node
        ListNode* resultList = dummyHead->next;

        // Cleaning up the dummy head node to prevent memory leak
        delete dummyHead;

        // Return the head of the list containing the frequencies
        return resultList;
    }
};
```

TypeScript

```
/**
 * Definition for 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);
    }
}

/**
 * Calculates the frequency of each element in a linked list and
 * creates a new linked list with the count of each element.
 * @param head The head of the input linked list.
 * @returns The head of the new linked list containing frequencies.
 */
function frequenciesOfElements(head: ListNode | null): ListNode | null {
    // Create a map to keep track of the frequency count of each element.
    const frequencyCount: Map<number, number> = new Map();

    // Iterate over the linked list and update the count of each element.
    let currentNode = head;
    while (currentNode !== null) {
        const currentValue = currentNode.val;
        frequencyCount.set(currentValue, (frequencyCount.get(currentValue) || 0)
            + 1);
        currentNode = currentNode.next;
    }

    // Initialize a dummy node to simplify insertions.
    const dummyHead = new ListNode();
    let tail = dummyHead;

    // Iterate over the frequency count map and add the frequencies to the new linked list.
    for (const frequency of frequencyCount.values()) {
        tail.next = new ListNode(frequency);
        tail = tail.next;
    }

    // Return the next of dummy node which points to head of the frequency list.
    return dummyHead.next;
}
```

Python

```
from collections import Counter

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

class Solution:
    def frequenciesOfElements(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # Use Counter to count occurrences of each element
        count = Counter()

        # Iterate through the linked list and count frequencies
        current_node = head
        while current_node:
            count[current_node.val] += 1
            current_node = current_node.next

        # Create a dummy node to make adding new nodes easier
        dummy_node = ListNode()
        tail = dummy_node  # 'tail' will be used to keep track of the last node

        # Iterate through the frequencies and create a new linked list
        # with the frequency values of each element in the same order they appear in the count
        for frequency in count.values():
            # Add a new node with the frequency to the new list
            tail.next = ListNode(frequency)
            tail = tail.next  # Move the tail to the new last node

        # Return the head of the new linked list which contains the frequencies
        return dummy_node.next
```

Time and Space Complexity

The time complexity of the code is $O(n)$, where n is the number of nodes in the linked list. This is because the algorithm must traverse each node once to count the frequency of its value, which is done in a single pass.

The space complexity is also $O(n)$ because a `Counter` is used to store the frequency of each distinct element in the linked list. In the worst case, all elements are unique, and the counter will have n key-value pairs.