2671. Frequency Tracker

Medium | Design | Hash Table

Leetcode Link

Problem Description

The problem requires us to design a data structure called `FrequencyTracker` that can track the number of occurrences, or *frequencies*, of numbers added to it. The `FrequencyTracker` should be able to perform the following operations:

- Initialization with an empty data structure.
- Adding a number to the data structure using the `add` method.
- Deleting a single occurrence of a number from the data structure with the `deleteOne` method. If the number is not present, no action is taken.
- Checking if there is any number in the data structure that appears exactly a given number of times with the `hasFrequency` method.

A key challenge of this problem is efficiently updating and querying the frequency of numbers in the data structure since the operations can occur numerous times.

Intuition

The intuition behind the solution is to use two hash-based data structures, where we map numbers to their frequencies and frequencies to the count of numbers having them.

When adding a number, we increment its frequency count in the `cnt` dictionary and update the frequency counts in the `freq` dictionary accordingly. If a frequency count goes to zero, it's important to decrement the count of numbers that previously had that frequency.

Similarly, when deleting a number, we decrease the number's frequency in `cnt` and update the frequency counts in the `freq`.

The `hasFrequency` method is straightforward: it directly checks if there is any number with the given frequency by looking it up in the `freq` dictionary.

The primary goal is to keep these operations at constant time complexity, `O(1)`, which is attainable through effective use of the hash maps provided by the `defaultdict` from Python's collections module. By keeping direct mappings of frequencies and their counts, we avoid the need to iterate through our data structure to find frequencies, which would increase the time complexity significantly.

Solution Approach

In the given solution, we utilize two dictionaries from the Python `collections` library. The `defaultdict` type is used to automatically create dictionary entries with a default value of `0` for new keys.

Let's examine each method of the `FrequencyTracker` class to understand its inner workings:

- The `__init__` method sets up our data structure with two `defaultdict` instances named `cnt` and `freq`. `cnt` keeps track of the frequency for each number while `freq` tracks how many numbers have a particular frequency.
- The `add` method increases the frequency of the given `number`. It performs the following steps:
  1. Checks if the current frequency of the number is greater than 0. This means we have some numbers with this frequency, and since we're about to increase this `number`'s frequency, we need to decrease the count of numbers with the current frequency. Hence, we do `self.freq[self.cnt[number]] -= 1`.
  2. Increases the frequency of the number by one in the `cnt` dictionary with `self.cnt[number] += 1`.
  3. Accounts for the new frequency by incrementing its count in the `freq` dictionary with `self.freq[self.cnt[number]] += 1`.
- The `deleteOne` method removes one occurrence of the given `number`. Its steps are:
  1. It checks whether the `number` is present in `cnt`. If its count is at `0`, we do nothing as there's nothing to delete.
  2. If the `number` exists, it decreases the count of numbers with the current frequency by updating `self.freq[self.cnt[number]]`.
  3. It then decrements the count of the `number` in `cnt` with `self.cnt[number] -= 1`.
  4. Finally, it increments the count of numbers with the new (decreased) frequency since the `number` now belongs to this new frequency category.
- The `hasFrequency` method simply returns whether the given `frequency` has a non-zero count in the `freq` dictionary with the expression `self.freq[frequency] > 0`. If the count is greater than 0, then there's at least one number in our data structure that has the queried frequency, so it returns `True`. Otherwise, it returns `False`.

This solution leverages the power of hash maps for efficient lookups and updates. The operations—`add`, `deleteOne`, and `hasFrequency`—all work in constant time, `O(1)`, since they involve a fixed number of hash map updates and lookups, making this solution highly optimal for the problem at hand.

Example Walkthrough

Let's illustrate the solution approach by walking through a small example that demonstrates how the `FrequencyTracker` would operate:

1. We start by initializing our `FrequencyTracker`, which sets up our two `defaultdict` instances: `cnt` and `freq`.
2. Next, we call the `add` method with the number `5`. Since `5` is not in `cnt`, it gets a default value of `0`. We then increment this to `1` in `cnt[5]`. Our dictionaries now look like this:
   - `cnt`: `{5: 1}`
   - `freq`: `{1: 1}` (meaning there is 1 number with a frequency of 1)
3. We add the number `5` again. The `freq[cnt[5]]` which was `1` is decremented to `0`, as we're about to increment the frequency of `5`. After incrementing `cnt[5]`, we also increment `freq[2]` to `1`. The dictionaries are now:
   - `cnt`: `{5: 2}`
   - `freq`: `{1: 0, 2: 1}`
4. Suppose we add a different number, say `3`. Just like when we first added `5`, `cnt` and `freq` are updated respectively:
   - `cnt`: `{5: 2, 3: 1}`
   - `freq`: `{1: 1, 2: 1}`
5. Now we want to delete one occurrence of the number `5` using `deleteOne`. This results in decreasing `cnt[5]` and also updating `freq` accordingly:
   - `cnt`: `{5: 1, 3: 1}`
   - `freq`: `{1: 2, 2: 0}` (since we now have two numbers with a frequency of 1)
6. Let's try to delete a number that hasn't been added, for instance `deleteOne(7)`. Since `7` is not in `cnt`, nothing changes, and `cnt` and `freq` remain the same.
7. Finally, using `hasFrequency`, we check if there is a number with a frequency of `1`. Since `freq[1]` is `2`, we return `True` because there are two numbers (3 and 5 after the deletion happened) with a frequency of 1.

This small example walks us through a series of operations that fulfill the objectives of the `FrequencyTracker`. Each step involves a certain number of constant-time hash map operations, thus maintaining the `O(1)` time complexity for all operations in the data structure.

Python Solution

```python
from collections import defaultdict

class FrequencyTracker:
    def __init__(self):
        # Initialize a dictionary to store the count of each number
        self.number_count = defaultdict(int)
        # Initialize a dictionary to store the count of frequencies
        self.frequency_count = defaultdict(int)

    def add(self, number: int) -> None:
        # If the number is already present, decrease its old frequency count
        if self.frequency_count[self.number_count[number]] > 0:
            self.frequency_count[self.number_count[number]] -= 1

        # Increment the count of the number
        self.number_count[number] += 1
        # Increment the count of the new frequency
        self.frequency_count[self.number_count[number]] += 1

    def deleteOne(self, number: int) -> None:
        # If the number is not present, there is nothing to delete
        if self.number_count[number] == 0:
            return

        # Decrease the frequency count of the number's current count
        self.frequency_count[self.number_count[number]] -= 1
        # Decrement the count of the number
        self.number_count[number] -= 1
        # If the new count is not zero, increment the frequency count of the new count
        self.frequency_count[self.number_count[number]] += 1

    def hasFrequency(self, frequency: int) -> bool:
        # Check if the frequency count is greater than 0,
        # which means there is at least one number with the given frequency
        return self.frequency_count[frequency] > 0

```

Java Solution

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

public class FrequencyTracker {

    // This map maintains the count of occurrences for each number.
    private Map<Integer, Integer> numberFrequencyMap = new HashMap<>();

    // This map maintains the frequency of frequencies.
    private Map<Integer, Integer> frequencyCountMap = new HashMap<>();

    // Constructor for FrequencyTracker class
    public FrequencyTracker() {
        // No initialization is needed since we're using HashMaps.
    }

    // Method to add a number and update frequency tracking.
    public void add(int number) {
        // First, get the current frequency of the number, defaulting to zero if not present.
        int currentFrequency = numberFrequencyMap.getOrDefault(number, 0);

        // Decrease the count of the frequency that the number currently has, if it is greater than 0.
        if (frequencyCountMap.getOrDefault(currentFrequency, 0) > 0) {
            frequencyCountMap.merge(currentFrequency, -1, Integer::sum);
        }

        // Increase the frequency of the number by 1.
        numberFrequencyMap.merge(number, 1, Integer::sum);

        // Increase the count of the new frequency the number now has.
        frequencyCountMap.merge(currentFrequency + 1, 1, Integer::sum);
    }

    // Method to delete a single occurrence of a number and update frequency tracking.
    public void deleteOne(int number) {
        // Retrieve the current frequency of the number, defaulting to zero if not present.
        int currentFrequency = numberFrequencyMap.getOrDefault(number, 0);

        // If the number is not tracked or its count is zero, do nothing.
        if (currentFrequency == 0) {
            return;
        }

        // Decrease the count of the frequency that the number currently has.
        frequencyCountMap.merge(currentFrequency, -1, Integer::sum);

        // Decrease the frequency of the number by 1.
        numberFrequencyMap.merge(number, -1, Integer::sum);

        // If the frequency becomes zero, remove it from the map.
        if (numberFrequencyMap.get(number) == 0) {
            numberFrequencyMap.remove(number);
        } else {
            // Increase the count of the new frequency the number now has.
            frequencyCountMap.merge(currentFrequency - 1, 1, Integer::sum);
        }
    }

    // Method to check if there is any number that has exactly the given frequency.
    public boolean hasFrequency(int frequency) {
        // Return true if the frequency is present and there are numbers with that frequency; otherwise, false.
        return frequencyCountMap.getOrDefault(frequency, 0) > 0;
    }
}

// Sample usage:
// FrequencyTracker tracker = new FrequencyTracker();
// tracker.add(number);           // Adds a number to the tracker.
// tracker.deleteOne(number);    // Removes one occurrence of the number.
// boolean hasFreq = tracker.hasFrequency(frequency);  // Checks if any number has the given frequency.

```

C++ Solution

```cpp
#include <unordered_map>
using namespace std;

class FrequencyTracker {
public:
    FrequencyTracker() {
        // Constructor, no initialization needed since the maps are empty
    }

    // Adds a number to the tracker
    void add(int number) {
        // Fetch the current frequency of the number
        int currentFrequency = numberCount[number];
        // If the number is already present, decrement its count in the frequency map
        if (currentFrequency > 0) {
            frequencyCount[currentFrequency]--;
        }
        // Increment the count of the number
        numberCount[number]++;
        // Increment the count of the new frequency
        frequencyCount[currentFrequency + 1]++;
    }

    // Deletes a single instance of a number from the tracker
    void deleteOne(int number) {
        // Fetch the current frequency of the number
        int currentFrequency = numberCount[number];
        // Return early if the number does not exist in the tracker
        if (currentFrequency == 0) {
            return;
        }
        // Decrement the frequency count of the number's current frequency
        frequencyCount[currentFrequency]--;
        // Decrement the number's count
        numberCount[number]--;
        // Increment the count of the new frequency
        frequencyCount[currentFrequency - 1]++;
    }

    // Checks if any number has the specified frequency
    bool hasFrequency(int frequency) {
        // Return true if the frequency count is greater than 0
        return frequencyCount[frequency] > 0;
    }

private:
    // A map to store the counts of individual numbers
    unordered_map<int, int> numberCount;
    // A map to store the frequencies of counts
    unordered_map<int, int> frequencyCount;
};

/**
 * Your FrequencyTracker object will be instantiated and called as such:
 * FrequencyTracker* obj = new FrequencyTracker();
 * obj->add(number);
 * obj->deleteOne(number);
 * bool freqExists = obj->hasFrequency(frequency);
 */

```

Typescript Solution

```typescript
// This map keeps track of the count of each number.
const countMap: Map<number, number> = new Map();

// This map keeps track of the frequency of each count.
const frequencyMap: Map<number, number> = new Map();

/**
 * Adds a number to the tracker, increasing its count.
 * @param {number} number - The number to add.
 */
function add(number: number): void {
    // Retrieve the current count for the number, default to 0.
    const currentCount = countMap.get(number) || 0;

    // Decrease the frequency of the current count, if any.
    if (currentCount > 0) {
        frequencyMap.set(currentCount, (frequencyMap.get(currentCount) || 0) - 1);
    }

    // Increase the count of the number.
    countMap.set(number, currentCount + 1);

    // Increase the frequency of the new count.
    frequencyMap.set(currentCount + 1, (frequencyMap.get(currentCount + 1) || 0) + 1);
}

/**
 * Deletes one instance of a number from the tracker.
 * @param {number} number - The number to delete.
 */
function deleteOne(number: number): void {
    // Retrieve the current count for the number, default to 0.
    const currentCount = countMap.get(number) || 0;

    // If the count is zero, exit the function as there is nothing to delete.
    if (currentCount === 0) {
        return;
    }

    // Decrease the frequency of the current count.
    frequencyMap.set(currentCount, (frequencyMap.get(currentCount) || 0) - 1);

    // Update the count map to reflect one less of the number.
    if (currentCount - 1 > 0) {
        countMap.set(number, currentCount - 1);
        frequencyMap.set(currentCount - 1, (frequencyMap.get(currentCount - 1) || 0) + 1);
    } else {
        // If the new count is 0, remove the number from the countMap.
        countMap.delete(number);
    }
}

/**
 * Checks if any number has a specific frequency count.
 * @param {number} frequency - The frequency to check.
 * @returns {boolean} - True if any number has the specified frequency count.
 */
function hasFrequency(frequency: number): boolean {
    return (frequencyMap.get(frequency) || 0) > 0;
}

```

Time and Space Complexity

Time Complexity

- `__init__`: The initialization method's time complexity is `O(1)` since it only involves initializing two `defaultdict` data structures.
- `add` method: The `add` method has a time complexity of `O(1)`. This is because it performs a constant amount of work - updating dictionaries - regardless of the size of the input data.
- `deleteOne` method: Similar to the `add` method, `deleteOne` also has a time complexity of `O(1)` for the same reasons.
- `hasFrequency` method: The `hasFrequency` method again has a time complexity of `O(1)` as it only checks the value associated with a key in a dictionary.

Space Complexity

- The space complexity of the `FrequencyTracker` class is `O(n)` where `n` is the number of unique numbers that have been added to the tracker. This is due to the fact that each unique number requires space in both the `cnt` and `freq` dictionaries. The space taken by these dictionaries scales with the number of unique numbers added.