2671. Frequency Tracker Design Hash Table Medium

Leetcode Link

The problem requires us to design a data structure called FrequencyTracker that can track the number of occurrences, or

Problem Description

frequencies, of numbers added to it. The FrequencyTracker should be able to perform the following operations: Initialization with an empty data structure.

 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.

frequencies to the count of numbers having them.

Adding a number to the data structure using the add method.

method. A key challenge of this problem is efficiently updating and querying the frequency of numbers in the data structure since the

Checking if there is any number in the data structure that appears exactly a given number of times with the hasFrequency

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

When adding a number, we increment its frequency count in the cnt dictionary and update the frequency counts in the frequency

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, 0(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.

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.

self.freq[self.cnt[number]].

solution highly optimal for the problem at hand.

cnt [5]. Our dictionaries now look like this:

Solution Approach

• 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

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.

3. It then decrements the count of the number in cnt with self.cnt[number] -= 1.

has the queried frequency, so it returns True. Otherwise, it returns False.

frequency. Hence, we do 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.

4. Finally, it increments the count of numbers with the new (decreased) frequency since the number now belongs to this new

expression self.freq[frequency] > 0. If the count is greater than 0, then there's at least one number in our data structure that

- 2. If the number exists, it decreases the count of numbers with the current frequency by updating
- frequency category. • The hasFrequency method simply returns whether the given frequency has a non-zero count in the freq dictionary with the

This solution leverages the power of hash maps for efficient lookups and updates. The operations—add, deleteone, and

1. We start by initializing our FrequencyTracker, which sets up our two defaultdict instances: cnt and freq.

hasFrequency—all work in constant time, 0(1), since they involve a fixed number of hash map updates and lookups, making this

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

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

o 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:

4. Suppose we add a different number, say 3. Just like when we first added 5, cnt and freq are updated respectively: o cnt: {5: 2, 3: 1}

freq accordingly:

structure.

10

11

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

64

65

71

// Sample usage:

C++ Solution

public:

12

13

14

15

16

17

68 // tracker.add(number);

#include <unordered_map>

class FrequencyTracker {

FrequencyTracker() {

void add(int number) {

// Adds a number to the tracker

if (currentFrequency > 0) {

// Fetch the current frequency of the number

int currentFrequency = numberCount[number];

frequencyCount[currentFrequency]--;

// Increment the count of the number

using namespace std;

69 // tracker.deleteOne(number);

67 // FrequencyTracker tracker = new FrequencyTracker();

// Adds a number to the tracker.

// Constructor, no initialization needed since the maps are empty

// If the number is already present, decrement its count in the frequency map

70 // boolean hasFreq = tracker.hasFrequency(frequency); // Checks if any number has the given frequency.

// Removes one occurrence of the number.

o cnt: {5: 2}

o freq: {1: 0, 2: 1}

o freq: {1: 1, 2: 1}

o cnt: {5: 1, 3: 1}

and freq remain the same.

from collections import defaultdict

self.number count = defaultdict(int)

def add(self, number: int) -> None:

self.frequency count = defaultdict(int)

Increment the count of the new frequency

def deleteOne(self, number: int) -> None:

if self.number_count[number] == 0:

Decrement the count of the number

def hasFrequency(self, frequency: int) -> bool:

self.number_count[number] -= 1

self.frequency_count[self.number_count[number]] += 1

self.frequency_count[self.number_count[number]] -= 1

self.frequency_count[self.number_count[number]] += 1

Check if the frequency count is greater than 0,

If the number is not present, there is nothing to delete

Decrease the frequency count of the number's current count

which means there is at least one number with the given frequency

// This map maintains the count of occurrences for each number.

If the new count is not zero, increment the frequency count of the new count

class FrequencyTracker:

def __init__(self):

return

 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

7. Finally, using has Frequency, we check if there is a number with a frequency of 1. Since freq[1] is 2, we return True because

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 0(1) time complexity for all operations in the data

5. Now we want to delete one occurrence of the number 5 using deleteone. This results in decreasing cnt [5] and also updating

Python Solution

If the number is already present, decrease its old frequency count

Initialize a dictionary to store the count of each number

Initialize a dictionary to store the count of frequencies

there are two numbers (3 and 5 after the deletion happened) with a frequency of 1.

if self.frequency_count[self.number_count[number]] > 0: self.frequency count[self.number count[number]] -= 1 13 14 15 # Increment the count of the number 16 self.number_count[number] += 1

```
35
            return self.frequency_count[frequency] > 0
36
```

Java Solution

import java.util.HashMap;

public class FrequencyTracker {

2 import java.util.Map;

```
private Map<Integer, Integer> numberFrequencyMap = new HashMap<>();
 8
       // This map maintains the frequency of frequencies.
 9
10
        private Map<Integer, Integer> frequencyCountMap = new HashMap<>();
11
12
       // Constructor for FrequencyTracker class
13
        public FrequencyTracker() {
14
           // No initialization is needed since we're using HashMaps.
15
16
17
       // Method to add a number and update frequency tracking.
18
        public void add(int number) {
19
           // First, get the current frequency of the number, defaulting to zero if not present.
20
            int currentFrequency = numberFrequencyMap.getOrDefault(number, 0);
21
22
           // Decrease the count of the frequency that the number currently has, if it is greater than 0.
23
           if (frequencyCountMap.getOrDefault(currentFrequency, 0) > 0) {
24
                frequencyCountMap.merge(currentFrequency, -1, Integer::sum);
25
26
27
           // Increase the frequency of the number by 1.
28
           numberFrequencyMap.merge(number, 1, Integer::sum);
29
30
           // Increase the count of the new frequency the number now has.
31
            frequencyCountMap.merge(currentFrequency + 1, 1, Integer::sum);
32
33
34
       // Method to delete a single occurrence of a number and update frequency tracking.
35
        public void deleteOne(int number) {
36
           // Retrieve the current frequency of the number, defaulting to zero if not present.
37
            int currentFrequency = numberFrequencyMap.getOrDefault(number, 0);
38
39
            // If the number is not tracked or its count is zero, do nothing.
            if (currentFrequency == 0) {
40
41
                return;
42
43
44
            // Decrease the count of the frequency that the number currently has.
45
            frequencyCountMap.merge(currentFrequency, -1, Integer::sum);
46
47
            // Decrease the frequency of the number by 1.
           numberFrequencyMap.merge(number, -1, Integer::sum);
48
49
50
            // If the frequency becomes zero, remove it from the map.
51
           if (numberFrequencyMap.get(number) == 0) {
52
                numberFrequencyMap.remove(number);
53
            } else {
54
                // Increase the count of the new frequency the number now has.
55
                frequencyCountMap.merge(currentFrequency - 1, 1, Integer::sum);
56
57
58
59
       // Method to check if there is any number that has exactly the given frequency.
        public boolean hasFrequency(int frequency) {
60
           // Return true if the frequency is present and there are numbers with that frequency; otherwise, false.
61
62
            return frequencyCountMap.getOrDefault(frequency, 0) > 0;
63
```

53

30

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

57

59

60

61

/**

*/

*/

function deleteOne(number: number): void {

if (currentCount === 0) {

if (currentCount - 1 > 0) {

countMap.delete(number);

return;

} else {

// Retrieve the current count for the number, default to 0.

// Update the count map to reflect one less of the number.

// If the count is zero, exit the function as there is nothing to delete.

frequencyMap.set(currentCount, (frequencyMap.get(currentCount) || 0) - 1);

// If the new count is 0, remove the number from the countMap.

* @returns {boolean} - True if any number has the specified frequency count.

frequencyMap.set(currentCount - 1, (frequencyMap.get(currentCount - 1) || 0) + 1);

const currentCount = countMap.get(number) || 0;

// Decrease the frequency of the current count.

countMap.set(number, currentCount - 1);

* Checks if any number has a specific frequency count.

* @param {number} frequency - The frequency to check.

return (frequencyMap.get(frequency) || 0) > 0;

function hasFrequency(frequency: number): boolean {

```
numberCount[number]++;
19
20
           // Increment the count of the new frequency
21
           frequencyCount[currentFrequency + 1]++;
22
23
24
       // Deletes a single instance of a number from the tracker
       void deleteOne(int number) {
25
26
           // Fetch the current frequency of the number
           int currentFrequency = numberCount[number];
           // Return early if the number does not exist in the tracker
28
29
           if (currentFrequency == 0) {
30
               return;
31
32
           // Decrement the frequency count of the number's current frequency
33
           frequencyCount[currentFrequency]--;
           // Decrement the number's count
34
35
           numberCount[number]--;
36
           // Increment the count of the new frequency
37
           frequencyCount[currentFrequency - 1]++;
38
39
       // Checks if any number has the specified frequency
40
       bool hasFrequency(int frequency) {
41
           // Return true if the frequency count is greater than 0
42
43
           return frequencyCount[frequency] > 0;
44
45
  private:
       // A map to store the counts of individual numbers
       unordered_map<int, int> numberCount;
48
       // A map to store the frequencies of counts
49
       unordered_map<int, int> frequencyCount;
50
51 };
52
   /**
    * 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);
59
60
Typescript Solution
    // 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.
 10
     */
     function add(number: number): void {
         // Retrieve the current count for the number, default to 0.
 12
 13
         const currentCount = countMap.get(number) || 0;
 14
 15
         // Decrease the frequency of the current count, if any.
 16
         if (currentCount > 0) {
             frequencyMap.set(currentCount, (frequencyMap.get(currentCount) | | 0) - 1);
 17
 18
 19
 20
         // Increase the count of the number.
         countMap.set(number, currentCount + 1);
 21
 22
 23
         // Increase the frequency of the new count.
         frequencyMap.set(currentCount + 1, (frequencyMap.get(currentCount + 1) || 0) + 1);
 24
 25
 26
 27 /**
     * Deletes one instance of a number from the tracker.
     * @param {number} number - The number to delete.
```

Time and Space Complexity

• __init__: The initialization method's time complexity is 0(1) since it only involves initializing two defaultdict data structures. • add method: The add method has a time complexity of 0(1). This is because it performs a constant amount of work - updating

Time Complexity

- 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 0(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.