1429. First Unique Number

Array

Queue

Design

Problem Description

Medium

The problem at hand requires us to keep track of integers as they arrive in a sequence, so we can always query for the first unique integer in this sequence. To clarify, a unique integer is an integer that appears exactly once in the current sequence. We're given a sequence of integers as an array and need to do the following: Initialize our data structure with the array of integers.

- - Be able to add new integers to the queue.

• Be able to return the value of the first unique integer in the queue at any time.

Hash Table

Data Stream

The problem specifies that if there isn't a unique integer, the showFirstUnique method should return -1. For the purpose of this

where elements are strictly processed in a FIFO (First-In-First-Out) manner, but rather a sequence where we can inspect and count occurrences of integers. Intuition

Navigating to a solution requires us to address two key operations efficiently: query for the first unique integer, and add new

problem, the 'queue' is a conceptual ordering of the integers; it's not necessarily a queue data structure in the traditional sense

integers to our dataset. A straightforward approach to find unique integers might involve iterating through our data and counting occurrences every time we want to find the first unique integer, but this would be inefficient, particularly as the amount of data

grows. Instead, we want to optimize this process. The given solution takes advantage of two Python data structures—Counter and deque—to handle these operations efficiently: • Counter: This is a special dictionary provided by Python's collections module that stores elements as keys and their counts as values. This allows us to efficiently keep track of the number of times each integer has been seen.

• deque: A double-ended queue that provides an efficient way to insert and remove elements from both ends, with O(1) time complexity for these operations. It is important here for tracking the order of the integers as they appear and for quick removal of non-unique integers from the front.

For the showFirstUnique method, the implementation uses the deque to keep the integers in order and Counter to know the counts. It pops elements from the deque that are no longer unique (i.e., have a count higher than 1), until it finds an integer that is unique, or if the deque is empty, in which case it returns -1.

count of the integer in the Counter, and appends the integer to the deque. This keeps our count accurate, and maintains the correct order for when we need to find the first unique integer. In summary, the Counter keeps track of occurrences allowing for quick updates and checks, while the deque maintains the order

The add method needs to take an integer and add it to our data set, which consists of both the Counter and deque. It updates the

The problem is addressed by implementing a class FirstUnique with two methods and an initializer: <u>__init__(self, nums: List[int])</u>: This is the initializer for our class. It takes a list of integers and initializes two attributes:

showFirstUnique(self) -> int: This method is used to retrieve the value of the first unique integer in the queue. It does so by:

added:

Example Walkthrough

using popleft().

of unique integers efficiently.

Solution Approach

1. Using a while loop to check the front element of the queue (using self.q[0]).

complexity that avoids the need for repeated scanning of the list of integers.

Now, when showFirstUnique() is called the first time, it performs the following steps:

• self.cnt[3] becomes 2 as 3 was already present in the sequence.

1. self.cnt: A Counter object which records the number of occurrences of each integer in the list.

2. If the count of the front element in the queue (as stored in self.cnt) is greater than 1, it is not unique; thus, it is removed from the deque

2. self.q: A deque which represents our queue and stores the integers in the same order as they appear in the original list.

- 3. This process repeats until a unique integer is found (an integer with a count of 1) or the deque becomes empty. 4. The method returns -1 if no unique integer is found; otherwise, it returns the first unique integer.
- 1. The Counter is updated to increment the count of the particular value (self.cnt[value] += 1). 2. The value is appended to the end of the deque (using self.q.append(value)). This ensures the order of integers is maintained.

By maintaining a count of occurrences and the order of inserts, the class efficiently supports querying for unique values and

add(self, value: int) -> None: This method handles adding a new integer to our data structure. When a new value is

extending the sequence with new values. The Counter allows for constant-time complexity for incrementing and checking counts, and the deque allows for constant-time complexity for adding and removing elements. Together, these data structures enable the FirstUnique class to perform the necessary operations with an efficient time

• self.cnt is initialized as Counter({4: 2, 10: 1, 5: 2, 3: 1}) showing the count of each number in the sequence. • self.q is initialized as deque([4, 10, 5, 3, 5, 4]) keeping the original order of numbers.

1. The method starts a loop and examines self.q[0], which is 4. 2. It then checks self.cnt[4], which is 2. Since this is greater than 1, 4 is not unique, so it is removed from self.q using popleft().

3. The next element at the front is 10. The count self.cnt[10] is checked and found to be 1. Therefore, 10 is the first unique number.

Later, if the add() method is called with the value 3, the following occurs:

4. showFirstUnique() returns 10.

Solution Implementation

from collections import Counter, deque

def __init__(self, nums: List[int]):

def showFirstUnique(self) -> int:

self.queue.popleft()

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

Loop until we find the first unique number or the queue is empty

If the first element in the queue is not unique, remove it

If the queue is empty return -1, otherwise return the first unique number

while self.queue and self.counter[self.queue[0]] != 1:

return -1 if not self.queue else self.queue[0]

Create an object of FirstUnique with a list of numbers

Call showFirstUnique to get the first unique number

Python

class FirstUnique:

• 3 is added to self.q, which is now deque([10, 5, 3, 5, 4, 3]).

Let's say we are given the following sequence of integers for our FirstUnique class: [4, 10, 5, 3, 5, 4].

When the FirstUnique object is created with this sequence, the following operations occur during initialization:

After this addition, calling showFirstUnique() again starts a check from the front: 1. The front element is 10 with self.cnt[10] still 1, so showFirstUnique() would again return 10.

2. If we had added another 10, then the count would change and the showFirstUnique() process would remove 10 from the queue since its count

Through this example, we understand how the FirstUnique class efficiently manages the data with Counter and deque to provide quick access to the first unique integer and accommodates additions to the sequence.

would be increased, and the next first unique number in the queue would be returned or -1 if there are no unique numbers left.

Initialize a counter to keep the count of each number self.counter = Counter(nums) # Initialize a queue to store the unique numbers in order self.queue = deque(nums)

Increment the count of the added value self.counter[value] += 1 # Add the value to the queue self.queue.append(value)

obj = FirstUnique(nums)

obj.add(value)

#include <vector>

class FirstUnique {

#include <unordered_map>

#include <deque>

public:

};

/**

* obj->add(value);

TypeScript

});

Java

Usage of the FirstUnique class

param_1 = obj.showFirstUnique()

Add a new value to the queue

```
import java.util.Map;
import java.util.HashMap;
import java.util.Deque;
import java.util.ArrayDeque;
class FirstUnique {
   // Maps each number to its occurrence count
    private Map<Integer, Integer> countMap = new HashMap<>();
    // Queue to maintain the order of elements
   private Deque<Integer> queue = new ArrayDeque<>();
   // Constructor that initializes the data structure with the given array of numbers
   public FirstUnique(int[] nums) {
        for (int num : nums) {
            add(num); // Use the add method to handle the addition of numbers
   // Returns the value of the first unique number
   public int showFirstUnique() {
       // While the queue is not empty and the front of the queue is not unique
       while (!queue.isEmpty() && countMap.get(queue.peekFirst()) != 1) {
            queue.pollFirst(); // Remove it from the queue
       // Return the first element in the queue if the queue is not empty, else return -1
       return queue.isEmpty() ? -1 : queue.peekFirst();
   // Adds a new number into the data structure
    public void add(int value) {
       // Update the occurrence count of the value
       countMap.put(value, countMap.getOrDefault(value, 0) + 1);
       // If it is the first time the value is added, add it to the queue
       if (countMap.get(value) == 1) {
           queue.offer(value);
/**
* The FirstUnique class can be used by creating an instance with an array of integers
* and calling the instance methods to show the first unique number or add new numbers to the structure.
* Example:
* FirstUnique firstUnique = new FirstUnique(new int[]{2, 3, 5});
* System.out.println(firstUnique.showFirstUnique()); // outputs the first unique number
* firstUnique.add(5); // adds a number to the data structure
*/
C++
```

```
++frequencyCount[value]; // Increment the frequency count of the added number
       uniqueQueue.push_back(value); // Add the new number to the back of the deque
private:
   std::unordered_map<int, int> frequencyCount; // Maps each number to its frequency count in the stream
```

* FirstUnique* obj = new FirstUnique(nums);

* int firstUniqueNumber = obj->showFirstUnique();

// @param nums - The array of integers to process.

function initializeFirstUnique(nums: number[]) {

std::deque<int> uniqueQueue;

if (!uniqueQueue.empty()) {

frequencyCount = {}; // Reset frequency count uniqueQueue = []; // Reset the unique queue nums.forEach(num => { frequencyCount[num] = (frequencyCount[num] || 0) + 1; // Increment the frequency count of each number

uniqueQueue.push(num); // Add the number to the queue

return uniqueQueue[0]; // Return the first unique number

// Function that returns the value of the first unique integer of the current list.

// Function that adds a value to the stream of integers the functions are tracking.

uniqueQueue.push(value); // Add the new number to the back of the queue

// Function that initializes the object with an array of integers.

/// Constructor that takes a vector of integers and initializes the object.

uniqueQueue.push_back(num); // Add the number to the deque

/// @return - The first unique integer in the list, or -1 if there isn't one.

return uniqueQueue.front(); // Return the first unique number

while (!uniqueQueue.empty() && frequencyCount[uniqueQueue.front()] != 1) {

uniqueQueue.pop_front(); // Remove non-unique elements from the front

// A deque maintaining the order of incoming numbers

let frequencyCount: { [key: number]: number } = {}; // Maps each number to its frequency count in the stream

frequencyCount[value] = (frequencyCount[value] | | 0) + 1; // Increment the frequency count of the added number

let uniqueQueue: number[] = []; // An array representing a queue maintaining the order of incoming numbers

/// Returns the value of the first unique integer of the current list.

return -1; // Return -1 if there's no unique number

/// @param value - The integer to add to the list.

/// Adds value to the stream of integers the class is tracking.

* Your FirstUnique object will be instantiated and called as such:

++frequencyCount[num]; // Increment the frequency count of each number

/// @param nums - The vector of integers to process.

FirstUnique(std::vector<int>& nums) {

for (int num : nums) {

int showFirstUnique() {

void add(int value) {

// @return - The first unique integer in the list, or -1 if there isn't one. function showFirstUnique(): number { while (uniqueQueue.length > 0 && frequencyCount[uniqueQueue[0]] !== 1) { uniqueQueue.shift(); // Remove non-unique elements from the front

// @param value - The integer to add to the list.

return -1; // Return -1 if there's no unique number

if (uniqueQueue.length > 0) {

function add(value: number) {

from collections import Counter, deque

def __init__(self, nums: List[int]):

self.counter = Counter(nums)

self.queue.popleft()

Usage of the FirstUnique class

param_1 = obj.showFirstUnique()

Add a new value to the queue

obj = FirstUnique(nums)

obj.add(value)

Time Complexity

class FirstUnique:

- // Example usage: // initializeFirstUnique([2, 3, 5]); // console.log(showFirstUnique()); // Outputs the first unique number // add(5); // Add number 5 to the queue // console.log(showFirstUnique()); // Outputs the new first unique number
- self.queue = deque(nums) def showFirstUnique(self) -> int: # Loop until we find the first unique number or the queue is empty while self.queue and self.counter[self.queue[0]] != 1:

If the first element in the queue is not unique, remove it

If the queue is empty return -1, otherwise return the first unique number

Initialize a counter to keep the count of each number

Initialize a queue to store the unique numbers in order

def add(self, value: int) -> None: # Increment the count of the added value self.counter[value] += 1 # Add the value to the queue self.queue.append(value)

return -1 if not self.queue else self.queue[0]

Time and Space Complexity

Create an object of FirstUnique with a list of numbers

Call showFirstUnique to get the first unique number

For the <u>__init__</u> method:

where n is the number of elements in nums. • Initializing the deque with nums has a time complexity of O(n) for copying all elements into the deque.

```
For the showFirstUnique method:
```

• The while loop in this method can be deceiving, but in the amortized analysis, each element gets dequeued only once due to the nature of the "first unique" constraint. Although in the worst case of a single showFirstUnique call, it could be O(n) where n is the number of elements in the deque, the total operation across all calls is bounded by the number of add operations. Therefore, we consider the amortized time complexity to

Constructing the Counter from the nums list involves counting the frequency of each integer in the list, which has a time complexity of O(n)

be 0(1). For the add method:

 Appending the value to the deque is also 0(1). **Space Complexity**

Incrementing the counter for a value is 0(1).

• The space complexity is O(n) for storing the elements in both the Counter and the deque, where n is the total number of elements that have been added (including duplicates). There is no extra space used that grows with respect to the input size or operations other than what is used to store the numbers and their counts.