Solution: Maximum Frequency Stack

Let's solve the Maximum Frequency Stack problem using the Knowing What to Track pattern.

We'll cover the following

Statement

Design a stack-like data structure. You should be able to push elements to this data structure and pop elements with maximum frequency.

You'll need to implement the **FreqStack** class that should consist of the following:

- **FreqStack**: This is a class used to declare a frequency stack.
- **Push(value)**: This is used to push an integer data onto the top of the stack.
- **Pop()**: This is used to remove and return the most frequent element in the stack.

> **Note:** If there is a tie for the most frequent element, then the most recently pushed element is removed and returned.

Constraints:

- $0 \leq$ `value` $\leq 10^9$
- At most, 2×10^3 calls will be made to **Push()** and **Pop()**.
- It is guaranteed that there will be at least one element in the stack before calling **Pop()**.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

One possible approach to solve the maximum frequency stack problem is using a heap data structure, specifically a max heap. This approach involves maintaining a hash map to keep track of the frequency of each element in the stack and a counter variable to keep track of the last inserted element.

To push elements onto the max-heap, we need to follow these steps:

- Check if the element is already present in a heap. If it is, increment its frequency in the hash map. Otherwise, set its frequency to 1.
- Increment the counter by 1.
- Add an array containing the element, its frequency, and the counter to the max-heap.

To pop elements from the max-heap, we need to follow these steps:

- Remove the most frequently occurring element from the max-heap.
- Decrement the frequency of that element in the hash map.

The time complexity of the push and pop operations using this approach is $O(\log n)$, where n is the number of elements in the stack. The space complexity is $O(n)$, as we may need to store up to n elements in the hash map in the worst case.

Optimized approach using frequency counting

The naive approach discussed above introduces additional time complexity for both operations because of the heap data structure. We can, however, reduce this time by using a stack to keep track of the elements with a particular frequency. The following variables and data structures are used to facilitate these operations:

`group`: A hash map where the keys represent the frequency of each element, and the values are stacks that store all elements with that frequency. The stack helps in maintaining the elements in the order of their most recent addition.

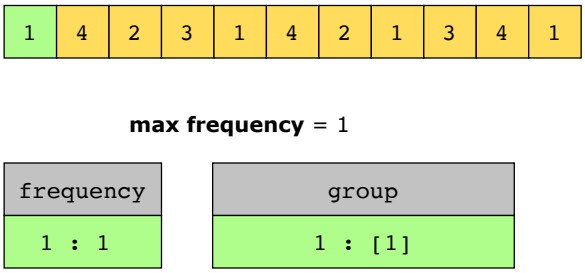
`frequency`: A hash map storing the element as the key. The frequency of the element is its corresponding value.

`maxFrequency`: A variable storing count of the element(s) with the most occurrences in the stack. It is initialized to 0.

Here's how we'll implement the algorithm to find the most frequent element from the stack:

- When pushing an element, perform the following steps:
  - Increment the frequency of that element in the `frequency` hash map. Store this frequency in a variable, `freq`.
  - If `freq` is greater than the value stored in `maxFrequency`, set `maxFrequency` equal to `freq`.
  - In the `group` hashmap, add the element to the end of the array which corresponds to the `freq` key.
- When popping an element, perform the following steps:
  - If `maxFrequency` is greater than `0`, our stack is not empty, so we can pop elements from it. We do the following:
    - From the `group` hash map, we access the array that corresponds to the `maxFrequency` key. This array contains all the greatest occurring elements sorted by least recent (leftmost) to most recent (rightmost). We pop the last element from the array and store it in a variable, `value`.
    - In the `frequency` hash map, we decrement the frequency by 1 which corresponds to the `show` key.
    - In the `group` hash map, if the array with key `maxFrequency` is now empty, there are no elements with our maximum occurring frequency. So we update `maxFrequency` by decrementing it by 1.
- Otherwise, if the `maxFrequency` is less than `0`, the `group` hash map is empty, so there are no elements to pop from it. Therefore, we return `-1`.
- If the above condition is not satisfied, we return the value stored in the `show` variable.

Let's make a stack from the below elements. We'll first need to insert all of the elements into our custom data structure. We start by inserting the element, **1**, through the **Push(1)** method. We increment the **max frequency** variable (initially 0) by 1.



Let's look at the code for this solution below:

Java

```
class FreqStack {
    // Declare a FreqStack class containing frequency and group hashmaps
    // and maxFrequency integer
    Map<Integer, Integer> frequency;
    Map<Integer, Stack <Integer>> group;
    int maxFrequency;

    // Use constructor to initialize the FreqStack object
    public FreqStack() {
        frequency = new HashMap < > ();
        group = new HashMap < > ();
        maxFrequency = 0;
    }

    // Use push function to push the value into the FreqStack
    public void push(int value) {
        // Get the frequency for the given value and
        // increment the frequency for the given value
        int freq = frequency.getOrDefault(value, 0) + 1;
        frequency.put(value, freq);

        // Check if the maximum frequency is lower that the new frequency
        // of the given show
        // Save the given showName for the new calculated frequency
```

Maximum Frequency Stack

Solution summary

This code solves the problem by maintaining a stack that tracks the frequency of elements and retrieves the most frequently occurring element efficiently. It uses two dictionaries to keep track of the frequency of each element and the elements associated with a given frequency. When a new element is pushed onto the stack, its frequency is incremented, and it is added to the list of elements corresponding to that frequency. The maximum frequency is updated as needed. When a pop operation is requested, the stack retrieves the element with the highest frequency and decrements its frequency. If there are multiple elements with the same highest frequency, the most recently added element is removed first.

Time complexity

The algorithm has $O(1)$ time complexity for both **Pop()** and **Push()** functionality.

Space complexity

The algorithm has $O(n)$ space complexity, where n is the number of elements in the **FreqStack**.

Back

Maximum Frequency ...

Next

First Unique Character...

Mark as Completed