895. Maximum Frequency Stack

Hard Stack Design Hash Table Ordered Set [Leetcode Link]

Problem Description

The problem asks for the design of a special kind of stack that keeps track of the frequency of elements as they are pushed. Unlike a regular stack, which simply adds elements to the top and removes them from the top in a LIFO (Last In First Out) manner, this stack has the additional functionality of popping the most frequently occurring element. In case of a tie in frequency, the element closest to the top should be popped.

Intuition

To solve this problem, we must maintain a frequency map and a map for elements by frequency. For this, two hashmaps are used: one to count the instances (`self.cnt`) of each value and another (`self.d`) to store values grouped by their instance count. Additionally, we keep track of the maximum frequency (`self.mx`) among all elements in the stack at any given time.

- When `push` is called, we increase the count of the element and update the element's group by frequency. We also update the `self.mx` to ensure it's always the highest frequency we've encountered so far.
- Conversely, when `pop` is called, we need to retrieve and return the most frequent element, which is at the end of the list in the `self.d` dictionary for the maximum frequency. After removing the element, we decrease its count in `self.cnt`. Should the list for this frequency be empty afterward, it means we no longer have elements with this highest frequency, so we decrement `self.mx`.

The key to the solution is making sure that both push and pop operations are done in O(1) average time complexity, which is achieved through the use of the hashmaps and keeping track of the maximum frequency dynamically.

Solution Approach

The implementation of the `FreqStack` utilizes hash maps, a commonly used data structure in algorithm design for achieving efficient, average O(1) time access and manipulation of data.

The `FreqStack` class is structured as follows:

1. **Initialization (`__init__` method):**
   - `self.cnt`: A `defaultdict(int)` which maps each value to its frequency of occurrence in the stack.
   - `self.d`: A `defaultdict(list)` which maps frequencies to a list of values that have that frequency.
   - `self.mx`: An integer keeping track of the current maximum frequency of any element in the `FreqStack`.
2. **Push Operation (`push` method):**
   - Increment the frequency count of the value `val` being pushed in `self.cnt`.
   - Append `val` to the list in `self.d` that corresponds to this new frequency count.
   - Update `self.mx` to reflect the maximum frequency count if the frequency of `val` is the new maximum.
3. **Pop Operation (`pop` method):**
   - Identify the value `val` that needs to be popped, which is the last element in the list at `self.d[self.mx]`.
   - Pop this value from the list.
   - Decrement the frequency count of the value `val` in `self.cnt`.
   - If the list at the current maximum frequency is now empty, decrement `self.mx` as there are no longer any elements with this frequency.
   - Return the value `val`.

By using these structures, we can ensure that the push operation is conducted by simply incrementing the count and appending to a list, and the pop operation is a matter of popping from a list and updating counts. Both operations avoid any time-consuming searches or iterations, allowing for fast execution.

This approach elegantly handles the requirements of popping the most frequent element, and in the case of a tie, the element nearest to the top, all while maintaining average O(1) time complexity for both push and pop operations.

Example Walkthrough

Let's consider a scenario where we operate on a `FreqStack` object to illustrate the solution approach described above.

1. **Initialize the `FreqStack` object.**:
   - `self.cnt` would be an empty `defaultdict(int)`
   - `self.d` would be an empty `defaultdict(list)`
   - `self.mx` would be initialized to `0`, indicating no elements are in the stack yet
2. **Perform several `push` operations**:
   - `push(5)`: `self.cnt[5]` becomes `1` since 5 is now pushed once. `self.d[1]` now contains `[5]`. `self.mx` is updated to `1`.
   - `push(7)`: `self.cnt[7]` becomes `1`. `self.d[1]` becomes `[5, 7]`. `self.mx` stays `1`.
   - `push(5)`: `self.cnt[5]` is incremented to `2`. `self.d[2]` now has `[5]`. `self.mx` is updated to `2`.
   - `push(7)`: `self.cnt[7]` is incremented to `2`. `self.d[2]` becomes `[5, 7]`. `self.mx` stays `2`.
   - `push(5)`: `self.cnt[5]` is incremented to `3`. `self.d[3]` now has `[5]`. `self.mx` is updated to `3`.

   At this point, our frequency map `self.cnt` looks like this: `{5: 3, 7: 2}` and our elements by frequency map `self.d` looks like this: `{1: [5, 7], 2: [5, 7], 3: [5]}`. The current maximum frequency `self.mx` is `3`.
3. **Perform a `pop` operation**:
   - We need to pop the most frequent element. According to `self.mx`, the most frequent elements are in the list `self.d[3]`, which currently has `[5]`.
   - We pop `5` from `self.d[3]`, which then becomes an empty list. Now, `self.cnt[5]` is decremented to `2`.
   - Since `self.d[3]` is empty, we decrement `self.mx` to `2`.

   The popped element is `5`. After the pop, the structures are:
   - `self.cnt`: `{5: 2, 7: 2}`
   - `self.d`: `{1: [5, 7], 2: [5, 7]}`
   - `self.mx`: `2`

Continuing to push and pop using the same approach, the `FreqStack` will maintain the frequency of elements and allow us to pop the most frequent element quickly, or the latest pushed element among the most frequent when there's a tie, achieving the average O(1) time complexity for both operations.

Python Solution

```python
from collections import defaultdict

class FreqStack:
    def __init__(self):
        # Initialize a dictionary to count the frequency of elements
        self.freq_counter = defaultdict(int)
        # A dictionary that maps frequencies to a list of elements with that frequency
        self.freq_dict = defaultdict(list)
        # Variable to keep track of the maximum frequency observed so far
        self.max_freq = 0

    def push(self, val: int) -> None:
        """
        Pushes an integer onto the stack and updates the structures tracking element frequency.
        """
        # Increment the frequency count for the given value
        self.freq_counter[val] += 1
        # Add the value to the list of values that have the new frequency count
        self.freq_dict[self.freq_counter[val]].append(val)
        # Update the maximum frequency if it's exceeded by this value's frequency
        self.max_freq = max(self.max_freq, self.freq_counter[val])

    def pop(self) -> int:
        """
        Pops and returns the most frequent integer from the stack. If there is a tie,
        it returns the integer closest to the top of the stack.
        """
        # Pop the value from the list corresponding to the maximum frequency
        val = self.freq_dict[self.max_freq].pop()
        # Decrement the frequency count for that value
        self.freq_counter[val] -= 1
        # If there are no more elements with the current maximum frequency, decrease the maximum frequency
        if not self.freq_dict[self.max_freq]:
            self.max_freq -= 1
        # Return the value
        return val


# How to use the FreqStack class
# obj = FreqStack()
# obj.push(val)
# param_2 = obj.pop()

```

Java Solution

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

// Class to define a stack-like data structure that supports push and pop
// operations based on the frequency of elements.
class FreqStack {
    // A map to store the frequency of each element.
    private Map<Integer, Integer> frequencyMap = new HashMap<>();
    // A map to store stacks corresponding to each frequency.
    private Map<Integer, Deque<Integer>> frequencyStackMap = new HashMap<>();
    // Variable to store the current maximum frequency.
    private int maxFrequency;

    // Constructor for the FreqStack class.
    public FreqStack() {
        // Initialize the maxFrequency to 0.
        maxFrequency = 0;
    }

    // Method to push an integer onto the stack.
    public void push(int val) {
        // Increase the frequency of the value by 1.
        frequencyMap.put(val, frequencyMap.getOrDefault(val, 0) + 1);
        // Get the updated frequency of the value.
        int currentFrequency = frequencyMap.get(val);
        // Add the value to the stack corresponding to its frequency.
        frequencyStackMap.computeIfAbsent(currentFrequency, k -> new ArrayDeque<>()).push(val);
        // Update the maxFrequency if necessary.
        maxFrequency = Math.max(maxFrequency, currentFrequency);
    }

    // Method to pop and return the most frequent element from the stack.
    // If there is a tie, it returns the element closest to the stack's top.
    public int pop() {
        // Pop the element from the stack with the maximum frequency.
        int value = frequencyStackMap.get(maxFrequency).pop();
        // Decrement the frequency count of the popped element.
        frequencyMap.put(value, frequencyMap.get(value) - 1);
        // If the stack corresponding to the maximum frequency is empty,
        // then reduce the maximum frequency.
        if (frequencyStackMap.get(maxFrequency).isEmpty()) {
            maxFrequency--;
        }
        // Return the popped element.
        return value;
    }
}

/**
 * The FreqStack class is used like this:
 * FreqStack obj = new FreqStack();
 * obj.push(val);
 * int param_2 = obj.pop();
 */

```

C++ Solution

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

class FreqStack {
public:
    FreqStack() {
        // Constructor initializes the FreqStack object.
    }

    void push(int val) {
        // Increment the frequency count of the pushed value.
        frequencyMap[val]++;

        // Push the value into the corresponding frequency stack.
        frequencyStackMap[frequencyMap[val]].push(val);

        // Update the maximum frequency.
        maxFrequency = max(maxFrequency, frequencyMap[val]);
    }

    int pop() {
        // Get the value from the top of the maximum frequency stack and
        // pop it from the stack.
        int value = frequencyStackMap[maxFrequency].top();
        frequencyStackMap[maxFrequency].pop();

        // Decrement the frequency count of the popped value.
        frequencyMap[value]--;

        // If the current maximum frequency stack is empty, decrement the
        // maximum frequency.
        if (frequencyStackMap[maxFrequency].empty()) {
            maxFrequency--;
        }

        // Return the popped value.
        return value;
    }

private:
    unordered_map<int, int> frequencyMap; // Maps value to its frequency.
    unordered_map<int, stack<int>> frequencyStackMap; // Maps frequency to a stack containing values with that frequency.
    int maxFrequency = 0; // The current maximum frequency among all values.
};

/**
 * The FreqStack object is instantiated and used as shown below:
 * FreqStack* obj = new FreqStack();
 * obj->push(val); // Pushes an element onto the stack.
 * int param_2 = obj->pop(); // Pops and returns the most frequent element. If there is a tie, it returns the element closest to the
 */

```

Typescript Solution

```typescript
// Importing the necessary data structures from JavaScript's standard library
import { Stack } from 'stack-typescript';

// Define a global variable to track the frequency of each value.
const frequencyMap: Record<number, number> = {};

// Define a global variable to map frequencies to stacks that hold values with those frequencies.
const frequencyStackMap: Record<number, Stack<number>> = {};

// Define a global variable to keep track of the current maximum frequency.
let maxFrequency: number = 0;

// Initializes the global data structures.
function initializeFreqStack(): void {
  // Reset frequencyMap and frequencyStackMap for a new FreqStack instance.
  for (const key in frequencyMap) delete frequencyMap[key];
  for (const key in frequencyStackMap) delete frequencyStackMap[key];
  maxFrequency = 0;
}

// Defines the push method to add a value to the FreqStack.
function push(val: number): void {
  // Check if the value is already in frequencyMap, increment its count, otherwise add it with a count of 1.
  frequencyMap[val] = (frequencyMap[val] || 0) + 1;

  // If frequencyStackMap doesn't already have a stack for the new frequency, create one.
  const frequency = frequencyMap[val];
  if (!frequencyStackMap[frequency]) {
    frequencyStackMap[frequency] = new Stack<number>();
  }

  // Push the value onto the appropriate frequency stack.
  frequencyStackMap[frequency].push(val);

  // Update maxFrequency if necessary.
  maxFrequency = Math.max(maxFrequency, frequency);
}

// Defines the pop method to remove and return the most frequent value from the FreqStack.
function pop(): number {
  // Retrieve the stack with the current max frequency.
  const maxFreqStack = frequencyStackMap[maxFrequency];

  // Pop the most frequent value from this stack.
  const value = maxFreqStack.top();
  maxFreqStack.pop();

  // Decrease the frequency of the popped value in the frequencyMap.
  frequencyMap[value] -= 1;

  // If the max frequency stack is now empty, decrement maxFrequency.
  if (maxFreqStack.size() === 0) {
    delete frequencyStackMap[maxFrequency];
    maxFrequency -= 1;
  }

  // Return the popped value.
  return value;
}

// To mimic class instantiation with global scope we invoke initializeFreqStack to start.
initializeFreqStack();

```

Time and Space Complexity

Time Complexity

- `__init__()`: The time complexity is `O(1)` as it only initializes variables.
- `push(val)`: The time complexity is `O(1)`. The operation increments a counter, appends a value to a list, and updates a maximum value. Each of these operations is constant time as it doesn't depend on the size of the data structure.
- `pop()`: The time complexity is `O(1)`. It pops a value from the list corresponding to the maximum frequency, decrements the counter for the value, and decreases the maximum frequency if necessary. These operations are all constant time since the pop operation removes the last element of the list which is a constant time operation.

Space Complexity

- Overall space complexity for the `FreqStack` class is `O(N)`, where `N` is the number of elements pushed into the stack. This is because the stack keeps track of all elements inserted, their counts, and the lists corresponding to each frequency.