Solution Review: Problem Challenge 3

We'll cover the following

Frequency Stack (hard)

Design a class that simulates a Stack data structure, implementing the following two operations:

1. `push(int num)`: Pushes the number 'num' on the stack.
2. `pop()`: Returns the most frequent number in the stack. If there is a tie, return the number which was pushed later.

Example:

```
After following push operations: push(1), push(2), push(3), push(2), push(1), push(2), push(5)

1. pop() should return 2, as it is the most frequent number
2. Next pop() should return 1
3. Next pop() should return 2
```

Solution

This problem follows the Top 'K' Elements pattern, and shares similarities with Top 'K' Frequent Numbers.

We can use a **Max Heap** to store the numbers. Instead of comparing the numbers we will compare their frequencies so that the root of the heap is always the most frequently occurring number. There are two issues that need to be resolved though:

1. How can we keep track of the frequencies of numbers in the heap? When we are pushing a new number to the **Max Heap**, we don't know how many times the number has already appeared in the **Max Heap**. To resolve this, we will maintain a **HashMap** to store the current frequency of each number. Thus whenever we push a new number in the heap, we will increment its frequency in the HashMap and when we pop, we will decrement its frequency.
2. If two numbers have the same frequency, we will need to return the number which was pushed later while popping. To resolve this, we need to attach a sequence number to every number to know which number came first.

In short, we will keep three things with every number that we push to the heap:

```
1. number // value of the number
2. frequency // current frequency of the number when it was pushed to the heap
3. sequenceNumber // a sequence number, to know what number came first
```

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
const Heap = require('./collections/heap'); //http://www.collectionsjs.com


class Element {
  constructor(number, frequency, sequenceNumber) {
    this.number = number;
    this.frequency = frequency;
    this.sequenceNumber = sequenceNumber;
  }

  compare(other) {
    // higher frequency wins
    if (this.frequency !== other.frequency) {
      return this.frequency - other.frequency;
    }
    // if both elements have same frequency, return the element that was pushed later
    return this.sequenceNumber - other.sequenceNumber;
  }
}

class FrequencyStack {
  constructor() {
    this.sequenceNumber = 0;
    this.frequencyMap = {};
    this.maxHeap = new Heap([], null, ((a, b) => a.compare(b)));
  }
```

Time complexity

The time complexity of `push()` and `pop()` is $O(logN)$ where 'N' is the current number of elements in the heap.

Space complexity

We will need $O(N)$ space for the heap and the map, so the overall space complexity of the algorithm is $O(N)$.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. See how

Back
Problem Challenge 3
Next
Introduction
Mark as Completed
Report an Issue
Ask a Question