Insert, Delete and Get Random in O(1)

Matei-Alexandru Gardus

September 27, 2020

Contents

1 Problem

1.1 Directions

Problem Link

> Design a data structure that supports all following operations in average O(1) time.
>
> Note: Duplicate elements are allowed.
>
> - insert(val): Inserts an item val to the collection.
> - remove(val): Removes an item val from the collection if present.
> - getRandom: Returns a random element from current collection of elements. The probability of each element being returned is linearly related to the number of same value the collection contains.

2 Notes

This is evidently a problem related to time complexity and remembering specifically which data structure is relevant for a specific situation. Here, the best data structure for O(1) amortized insertion and accessing is a dynamic array. The problem is the removal, since we need to traverse the array, which is O(n). However, as we know with Leetcode, O(n) become O(1) if we add a Hashtable in there somehow.

3 Solution

Considering that we are tasked with the design of a data structure, we'll need to figure out what underlying components to achieve the desired time complexity for each solution.

Insertion and access, as mentioned in the Notes, are O(1) amortized in dynamic arrays, which points us to a viable candidate for the data structure. The issue is removal in O(1) time. Luckily for us, removal in O(1) time can be achieved by remembering where a specific element is in the array before removing it. This can be done by storing a Hashtable which has the keys as the values in the array. Each key would point to an array of the indices at which this specific value can be found.

We'll begin by bringing up the standard Leetcode boilerplate, along with two imports:

- `defaultdict` is required to initialize any empty key with an empty set; this way, we can immediately append indices to each key for any found values
- `random.choice` is used to extract elements from the array randomly.

```
from collections import defaultdict
from random import choice

class RandomizedCollection:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        # collection is the array storing each element
        self.collection = []
        # indices is the hashtable described in the solution
        self.indices = defaultdict(set)
```

To insert an element, we'll need to simply append it to the array, which is an O(1) operation. We'll also need to store the index of the newly added value in the hashtable for later removal, if necessary. The `return` statement is required to implement the requested method. To fulfill the requested return value, we'll check if there were any indices in the hashtable stored at the added value before this function call.

```
    def insert(self, val: int) -> bool:
        """
        Inserts a value to the collection. Returns true if the collection did not alre
        """
        self.collection.append(val)
        self.indices[val].add(len(self.collection) - 1)
        return len(self.indices[val]) == 1
```

Removal is a bit more difficult, but to achieve the intended effect it's actually simple in Python: lists are given a `pop` method, which is O(1) time. If, in some way, the value of the element we're trying to remove were at the end of the array, we'd have O(1) time. However, we also know where any given value is in the array using the `indices` hashtable. Therefore, we can just swap the provided value with the one at the end of the array, pop the value we want to remove and adjust the `indices` hashtable accordingly.

```
    def remove(self, val: int) -> bool:
        """
        Removes a value from the collection. Returns true if the collection contained 
        """
        if self.indices[val]:
            indexOfVal = self.indices[val].pop()
            lastElement = self.collection[-1]
            # Swap them!
            self.collection[indexOfVal] = lastElement
            if self.indices[lastElement]:
                # Add the new index for the last element
                self.indices[lastElement].add(indexOfVal)
                # Remove the old index
                self.indices[lastElement].discard(len(self.collection) - 1)
                # Note the add and pop, because self.indices contains Sets, not Lists
            self.collection.pop()
            return True
        return False
```

Now for the last function, `getRandom`. We can just return a value located at random inside our collection.

```
    def getRandom(self) -> int:
        """
        Get a random element from the collection.
        """
        return choice(self.collection)
```

Note the mention of the probability of element being *"linearly related to the number of same value the collection contains"* in the directions. This is actually a bit confusing and added to surprise you; technically, the more an element appears in our collection, it's linearly more likely we'll randomly extract it automatically, with no need to modify our code to account for it. The added sentence there is most likely added since the underlying data structure made in a submission may end up not being an array, which isn't the case here.