

# 381. Insert Delete GetRandom O(1) - Duplicates allowed

June 25, 2019 | 20.2K views

Average Rating: 4.19 (31 votes)

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.

**Example:**

```
// Init an empty collection.
RandomizedCollection collection = new RandomizedCollection();

// Inserts 1 to the collection. Returns true as the collection did not contain 1.
collection.insert(1);

// Inserts another 1 to the collection. Returns false as the collection contained 1.
collection.insert(1);

// Inserts 2 to the collection, returns true. Collection now contains [1,1,2].
collection.insert(2);

// getRandom should return 1 with the probability 2/3, and returns 2 with the probability 1/3.
collection.getRandom();

// Removes 1 from the collection, returns true. Collection now contains [1,2].
collection.remove(1);

// getRandom should return 1 and 2 both equally likely.
collection.getRandom();
```

## Solution

### Intuition

We must support three operations with duplicates:

- insert**
- remove**
- getRandom**

To **getRandom** in  $O(1)$  and have it scale linearly with the number of copies of a value. The simplest solution is to store all values in a list. Once all values are stored, all we have to do is pick a random index.

We don't care about the order of our elements, so **insert** can be done in  $O(1)$  using a dynamic array (**ArrayList** in Java or **list** in Python).

The issue we run into is how to go about an  $O(1)$  remove. Generally we learn that removing an element from an array takes a place in  $O(N)$ , unless it is the last element in which case it is  $O(1)$ .

The key here is that *we don't care about order*. For the purposes of this problem, if we want to remove the element at the **i**th index, we can simply swap the **i**th element and the last element, and perform an  $O(1)$  pop (*technically* we don't have to swap, we just have to copy the last element into index **i** because it's popped anyway).

With this in mind, the most difficult part of the problem becomes *finding* the index of the element we have to remove. All we have to do is have an accompanying data structure that maps the element values to their index.

### Approach 1: ArrayList + HashMap

#### Algorithm

We will keep a **list** to store all our elements. In order to make finding the index of elements we want to remove  $O(1)$ , we will use a **HashMap** or dictionary to map values to all indices that have those values. To make this work each value will be mapped to a set of indices. The tricky part is properly updating the **HashMap** as we modify the **list**.

- insert**: Append the element to the **list** and add the index to **HashMap[element]**.
- remove**: This is the tricky part. We find the index of the element using the **HashMap**. We use the trick discussed in the intuition to remove the element from the **list** in  $O(1)$ . Since the last element in the list gets moved around, we have to update its value in the **HashMap**. We also have to get rid of the index of the element we removed from the **HashMap**.
- getRandom**: Sample a random element from the list.

#### Implementation

JavaPythonCopy

```
1 from collections import defaultdict
2 from random import choice
3
4 class RandomizedCollection:
5
6     def __init__(self):
7         """
8         Initialize your data structure here.
9         """
10        self.lst = []
11        self.idx = defaultdict(set)
12
13
14        def insert(self, val: int) -> bool:
15            """
16            Inserts a value to the collection. Returns true if the collection did not already contain the
17            specified element.
18            """
19            self.idx[val].add(len(self.lst))
20            self.lst.append(val)
21            return len(self.idx[val]) == 1
22
23
24        def remove(self, val: int) -> bool:
25            """
26            Removes a value from the collection. Returns true if the collection contained the specified
27            element.
```

#### Complexity Analysis


- Time complexity:  $O(N)$ , with  $N$  being the number of operations. All of our operations are  $O(1)$ , giving  $N * O(1) = O(N)$ .
- Space complexity:  $O(N)$ , with  $N$  being the number of operations. The worst case scenario is if we get  $N$  **add** operations, in which case our **ArrayList** and our **HashMap** grow to size  $N$ .

Rate this article: ★★★★★


PreviousNext


Comments: 6

Sort By ▾



Type comment here... (Markdown is supported)

 Preview Post







dvornikovdev


★ 10

🕒 June 27, 2019 1:45 PM

Is there more than one approaches?

10   |  Share |  Reply

SHOW 4 REPLIES







themimalist


★ 5

🕒 July 12, 2020 4:20 AM

How is remove() O(1) with set operations being O(log(n))?

1   |  Share |  Reply

SHOW 1 REPLY




tkblackbelt





★ 13

🕒 May 1, 2020 5:59 PM


I did it by having a deque for each hash map entry. So when I add an index I append to the end of the queue. Then when removing I pop the next index off, swap the index with the end of the array, and then pop the right element off the queue and append the new index to the left.

Runtime: 96 ms, faster than 95.92% of Python

 Read More

1   |  Share |  Reply

SHOW 1 REPLY







dpforever


★ 1

🕒 August 23, 2019 11:41 PM

Worst case time complexities for python set can be O(N). The provided remove operation is not a O(1) solution.

0   |  Share |  Reply

SHOW 3 REPLIES







person\_random

★ 7


🕒 June 7, 2020 12:56 AM

I think this is wrong solution.

getRandom() is not going to return linearly related probability. choice() simply return a uniform probability.

0   |  Share |  Reply

SHOW 1 REPLY







rahulkun

★ 454

🕒 May 15, 2020 5:36 AM

I don't believe set has O(1) complexity in C++. Above implementation isn't O(1).

0   |  Share |  Reply

SHOW 1 REPLY