Insert Delete GetRandom O(1) - Duplicates allowed

Design a data structure that supports all following operations in *average* **O(1)** time.

Note: Duplicate elements are allowed.

1. `insert(val)` : Inserts an item val to the collection.
2. `remove(val)` : Removes an item val from the collection if present.
3. `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 now contains [1,1].
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 proba
bility 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 1

The problem is a simple extension of the previous problem that did not have duplicates. Instead of storing a single index like in the previous problem, we simply store a collection of indices for all the times that a number appears in the array.

Insert() and random() are quite straightforward. For remove(), we take advantage of the fact that adding/removing from a HashSet is O(1) average time. The logic is otherwise similar - swap the index of any one instance of the item to be removed with the item in the very last place of the array. Update the sets after doing so, and then remove the last item.

Thanks to @yubad2000 for the wonderful idea of using a LinkedHashSet for O(1) iteration over large items. An iterator over a normal HashSet is actually O(h/n), where h is table capacity. So it is not a solution to our problem requiring O(1) time. Nor does an ArrayList instead of a HashSet work (I wasted some time on that for a while...).

```
public class RandomizedCollection {

    ArrayList<Integer> result;
    HashMap<Integer, LinkedHashSet<Integer>> map;

    public RandomizedCollection() {
        result = new ArrayList<Integer>();
        map = new HashMap<Integer, LinkedHashSet<Integer>>();
    }

    /** Inserts a value to the collection. Returns true if the collection did not
already contain the specified element. */
    public boolean insert(int val) {
        // Add item to map if it doesn't already exist.
        boolean alreadyExists = map.containsKey(val);
        if(!alreadyExists) {
            map.put(val, new LinkedHashSet<Integer>());
        }
        map.get(val).add(result.size());
        result.add(val);
        return !alreadyExists;
    }

    /** Removes a value from the collection. Returns true if the collection conta
ined the specified element. */
    public boolean remove(int val) {
        if(!map.containsKey(val)) {
            return false;
        }
        // Get arbitary index of the ArrayList that contains val
        LinkedHashSet<Integer> valSet = map.get(val);
        int indexToReplace = valSet.iterator().next();

        // Obtain the set of the number in the last place of the ArrayList
        int numAtLastPlace = result.get(result.size() - 1);
        LinkedHashSet<Integer> replaceWith = map.get(numAtLastPlace);
```

```
        // Replace val at arbitary index with very last number
        result.set(indexToReplace, numAtLastPlace);

        // Remove appropriate index
        valSet.remove(indexToReplace);

        // Don't change set if we were replacing the removed item with the same n
umber
        if(indexToReplace != result.size() - 1) {
            replaceWith.remove(result.size() - 1);
            replaceWith.add(indexToReplace);
        }
        result.remove(result.size() - 1);

        // Remove map entry if set is now empty, then return
        if(valSet.isEmpty()) {
            map.remove(val);
        }
        return true;
    }

    /** Get a random element from the collection. */
    public int getRandom() {
        // Get linearly random item
        return result.get((int)(Math.random() * result.size()));
    }
}
```

written by DeusVult original link here

Solution 2

This post is deleted!
written by haruhiku original link here

Solution 3

This solution assumes there is not ordering requirement for `remove`. `remove` will randomly remove an element among duplicates.

Similar to Problem 380 without duplicates, we use a list to contain all elements and use a dict to keep track of their indices. This time, we use a `set` to keep track of all indices for a value.

Just like Problem 380, when we remove a value, we randomly pop an index for that value from the set, swap the value with the last element of the list, and update the indices for the new value.

```
import random
class RandomizedCollection(object):

    def __init__(self):
        self.l, self.d = [], collections.defaultdict(set)

    def insert(self, val):
        self.d[val].add(len(self.l))
        self.l.append(val)
        return len(self.d[val])==1

    def remove(self, val):
        if val not in self.d:
            return False
        i, newVal = self.d[val].pop(), self.l[-1]
        len(self.d[val]) > 0 or self.d.pop(val, None)
        if newVal in self.d:
            self.d[newVal] = (self.d[newVal] | {i}) - {len(self.l)-1}
        self.l[i] = newVal
        self.l.pop()
        return True

    def getRandom(self):
        return random.choice(self.l)
```

Longer but clearer version:

```python
import random
class RandomizedCollection(object):
    def __init__(self):
        self.l = []
        self.d = collections.defaultdict(set)

    def insert(self, val):
        b = val not in self.d
        self.d[val].add(len(self.l))
        self.l.append(val)
        return b

    def remove(self, val):
        if val not in self.d:
            return False
        i, newVal = self.d[val].pop(), self.l[-1]
        if len(self.d[val]) == 0:
            del self.d[val]
        self.l[i] = newVal
        if newVal in self.d:
            self.d[newVal].add(i)
            self.d[newVal].discard(len(self.l)-1)
        self.l.pop()
        return True

    def getRandom(self):
        return random.choice(self.l)
```

written by o_sharp original link here