<https://leetcode.com/problems/insert-delete-getrandom-o1/description/>

Implement the RandomizedSet class:

RandomizedSet() Initializes the RandomizedSet object.

bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.

bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.

int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.You must implement the functions of the class such that each function works in **average** O(1) time complexity.

**Example 1:**

Input

["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]

[[], [1], [2], [2], [], [1], [2], []]

Output

[null, true, false, true, 2, true, false, 2]

Explanation

RandomizedSet randomizedSet = new RandomizedSet();

randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomizedSet.remove(2); // Returns false as 2 does not exist in the set.

randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.

randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 2.

**Constraints:**

-2^31 <= val <= 2^31 - 1

At most 2 \* 10^5 calls will be made to insert, remove, and getRandom.

There will be **at least one** element in the data structure when getRandom is called.

**Attempt 1: 2023-02-09**

**Solution 1:  Hash Table + Swap item with last item and remove as O(1) (30 min)**

class RandomizedSet {

    List<Integer> list;

    Map<Integer, Integer> map;

    Random random = new Random();

    public RandomizedSet() {

        list = new ArrayList<Integer>();

        // {key, value} -> {val, val position at list}

        map = new HashMap<Integer, Integer>();

    }

    public boolean insert(int val) {

        if(list.contains(val)) {

            return false;

        }

        // Put index before add val, because index start from 0,

        // if add val first, list size will increase 1, equal to

        // index start from 1, which is wrong coordinate system

        map.put(val, list.size());

        list.add(val);

        return true;

    }

    public boolean remove(int val) {

        if(!list.contains(val)) {

            return false;

        }

        int index = map.get(val);

        if(index < list.size() - 1) {

            int lastItem = list.get(list.size() - 1);

            list.set(index, lastItem);

            map.put(lastItem, index);

        }

        list.remove(list.size() - 1);

        map.remove(val);

        return true;

    }

    public int getRandom() {

        return list.get(random.nextInt(list.size()));

    }

}

/\*\*

\* Your RandomizedSet object will be instantiated and called as such:

\* RandomizedSet obj = new RandomizedSet();

\* boolean param\_1 = obj.insert(val);

\* boolean param\_2 = obj.remove(val);

\* int param\_3 = obj.getRandom();

\*/

**Refer to**

<https://leetcode.com/problems/insert-delete-getrandom-o1/solutions/1532391/javascript-easy-to-understand-map-array/>

For this problem, we need to make insert, remove, and getRandom all to O(1) time complexity, so it's straightforward to think about using a map. With a map, we could implement the insert and remove easily with O(1) time complexity. But how about getRandom?

Let's take a look at what operations do we need for the getRandom method:

Get a random number.

Get the real value according to that random number.

Looks like it's pretty easy, right? We could get a random number by Math.random, and use an array to store index-based values.

**But when we try to finish the code, we may find the remove method could not be easy, since if we use something like splice to remove the value in an array, it could be not O(1) depends on the implementation, and all the indexes after this element need to be updated.**

**So, here comes the final key point for this problem, how do we make it O(1) with steady indexes? Let's list some clues:**

**If we want steady indexes, then we can't remove this element from the list. We must put a value here.**

**If we wanna remove a value with O(1) in a list, it's straightforward to think about removing the last value.**

**Combined with these clues, it is not difficult for us to find out that we could swap the target value and the last value, then remove it. This could meet our two needs at the same time.**

At this moment, the whole strategy is clear, so the next step is just coding:

class RandomizedSet {

  constructor() {

    this.map = new Map();

    this.list = [];

  }

  insert(val) {

    if (this.map.has(val)) return false;

    this.map.set(val, this.list.length);

    this.list.push(val);

    return true;

  }

  remove(val) {

    if (!this.map.has(val)) return false;

    const idx = this.map.get(val);

    this.\_swap(idx, this.list.length - 1);

    this.list.pop();

    this.map.set(this.list[idx], idx);

    this.map.delete(val);

    return true;

  }

  getRandom() {

    return this.list[Math.floor(Math.random() \* this.list.length)];

  }

  \_swap(a, b) {

    const tmp = this.list[a];

    this.list[a] = this.list[b];

    this.list[b] = tmp;

  }

}

**Refer to**

<https://leetcode.com/problems/insert-delete-getrandom-o1/solutions/85401/java-solution-using-a-hashmap-and-an-arraylist-along-with-a-follow-up-131-ms/>

public class RandomizedSet {

    ArrayList<Integer> nums;

    HashMap<Integer, Integer> locs;

    java.util.Random rand = new java.util.Random();

    /\*\* Initialize your data structure here. \*/

    public RandomizedSet() {

        nums = new ArrayList<Integer>();

        locs = new HashMap<Integer, Integer>();

    }

    /\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/

    public boolean insert(int val) {

        boolean contain = locs.containsKey(val);

        if ( contain ) return false;

        locs.put( val, nums.size());

        nums.add(val);

        return true;

    }

    /\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/

    public boolean remove(int val) {

        boolean contain = locs.containsKey(val);

        if ( ! contain ) return false;

        int loc = locs.get(val);

        if (loc < nums.size() - 1 ) { // not the last one than swap the last one with this val

            int lastone = nums.get(nums.size() - 1 );

            nums.set( loc , lastone );

            locs.put(lastone, loc);

        }

        locs.remove(val);

        nums.remove(nums.size() - 1);

        return true;

    }

    /\*\* Get a random element from the set. \*/

    public int getRandom() {

        return nums.get( rand.nextInt(nums.size()) );

    }

}

**The follow-up: Allowing duplications**

class RandomizedSet {

    List<Integer> list;

    Map<Integer, Set<Integer>> map;

    Random random = new Random();

    public RandomizedSet() {

        list = new ArrayList<Integer>();

        // {key, value} -> {val, all val positions at list}

        map = new HashMap<Integer, Set<Integer>>();

    }

    public boolean insert(int val) {

        if(list.contains(val)) {

            return false;

        }

        // Put index before add val, becuase index start from 0,

        // if add val first, list size will increase 1, equal to

        // index start from 1, which is wrong coordinate system

        map.put(val, new HashSet<Integer>());

        map.get(val).add(list.size());

        list.add(val);

        return true;

    }

    public boolean remove(int val) {

        if(!list.contains(val)) {

            return false;

        }

        int index = map.get(val).iterator().next();

        if(index < list.size() - 1) {

            int lastItem = list.get(list.size() - 1);

            list.set(index, lastItem);

            map.get(lastItem).remove(list.size() - 1);

            map.get(lastItem).add(index);

        }

        list.remove(list.size() - 1);

        map.get(val).remove(index);

        return true;

    }

    public int getRandom() {

        return list.get(random.nextInt(list.size()));

    }

}

/\*\*

\* Your RandomizedSet object will be instantiated and called as such:

\* RandomizedSet obj = new RandomizedSet();

\* boolean param\_1 = obj.insert(val);

\* boolean param\_2 = obj.remove(val);

\* int param\_3 = obj.getRandom();

\*/

**Refer to**

<https://leetcode.com/problems/insert-delete-getrandom-o1/solutions/85401/java-solution-using-a-hashmap-and-an-arraylist-along-with-a-follow-up-131-ms/comments/90382>

**For example, after insert(1), insert(1), insert(2), getRandom() should have 2/3 chance return 1 and 1/3 chance return 2. Then, remove(1), 1 and 2 should have an equal chance of being selected by getRandom().**

**The idea is to add a set to the hashMap to remember all the locations of a duplicated number.**

    public class RandomizedSet {

    ArrayList<Integer> nums;

    HashMap<Integer, Set<Integer>> locs;

    java.util.Random rand = new java.util.Random();

    /\*\* Initialize your data structure here. \*/

    public RandomizedSet() {

        nums = new ArrayList<Integer>();

        locs = new HashMap<Integer, Set<Integer>>();

    }

    /\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/

    public boolean insert(int val) {

        boolean contain = locs.containsKey(val);

        if ( ! contain ) locs.put( val, new HashSet<Integer>() );

        locs.get(val).add(nums.size());

        nums.add(val);

        return ! contain ;

    }

    /\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/

    public boolean remove(int val) {

        boolean contain = locs.containsKey(val);

        if ( ! contain ) return false;

        int loc = locs.get(val).iterator().next();

            locs.get(val).remove(loc);

        if (loc < nums.size() - 1 ) {

            int lastone = nums.get(nums.size() - 1 );

            nums.set( loc , lastone );

            locs.get(lastone).remove(nums.size() - 1);

            locs.get(lastone).add(loc);

        }

        nums.remove(nums.size() - 1);

        if (locs.get(val).isEmpty()) locs.remove(val);

        return true;

    }

    /\*\* Get a random element from the set. \*/

    public int getRandom() {

        return nums.get( rand.nextInt(nums.size()) );

    }

}