# Question

Implement the RandomizedSet class:

* 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.

**Follow up:** Could you implement the functions of the class with each function works in **average** O(1) time?

**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:**

* -231 <= val <= 231 - 1
* At most 105 calls will be made to insert, remove, and getRandom.
* There will be **at least one** element in the data structure when getRandom is called.

# Solution

#### **Overview**

We're asked to implement the structure which provides the following operations in average \mathcal{O}(1)O(1) time:

* Insert
* Delete
* GetRandom

First of all - why this weird combination? The structure looks quite theoretical, but it's widely used in popular statistical algorithms like [Markov chain Monte Carlo](https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo) and [Metropolis–Hastings algorithm](https://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm). These algorithms are for sampling from a probability distribution when it's difficult to compute the distribution itself.

Let's figure out how to implement such a structure. Starting from the Insert, we immediately have two good candidates with \mathcal{O}(1)O(1) [average insert time](https://wiki.python.org/moin/TimeComplexity):

* Hashmap (or Hashset, the implementation is very similar): [Java HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) / [Python dictionary](https://docs.python.org/3/tutorial/datastructures.html#dictionaries)
* Array List: [Java ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) / [Python list](https://docs.python.org/3/tutorial/datastructures.html)

Let's consider them one by one.

Hashmap provides Insert and Delete in average constant time, although has problems with GetRandom.

The idea of GetRandom is to choose a random index and then to retrieve an element with that index. There is no indexes in hashmap, and hence to get true random value, one has first to convert hashmap keys in a list, that would take linear time. The solution here is to build a list of keys aside and to use this list to compute GetRandom in constant time.

Array List has indexes and could provide Insert and GetRandom in average constant time, though has problems with Delete.

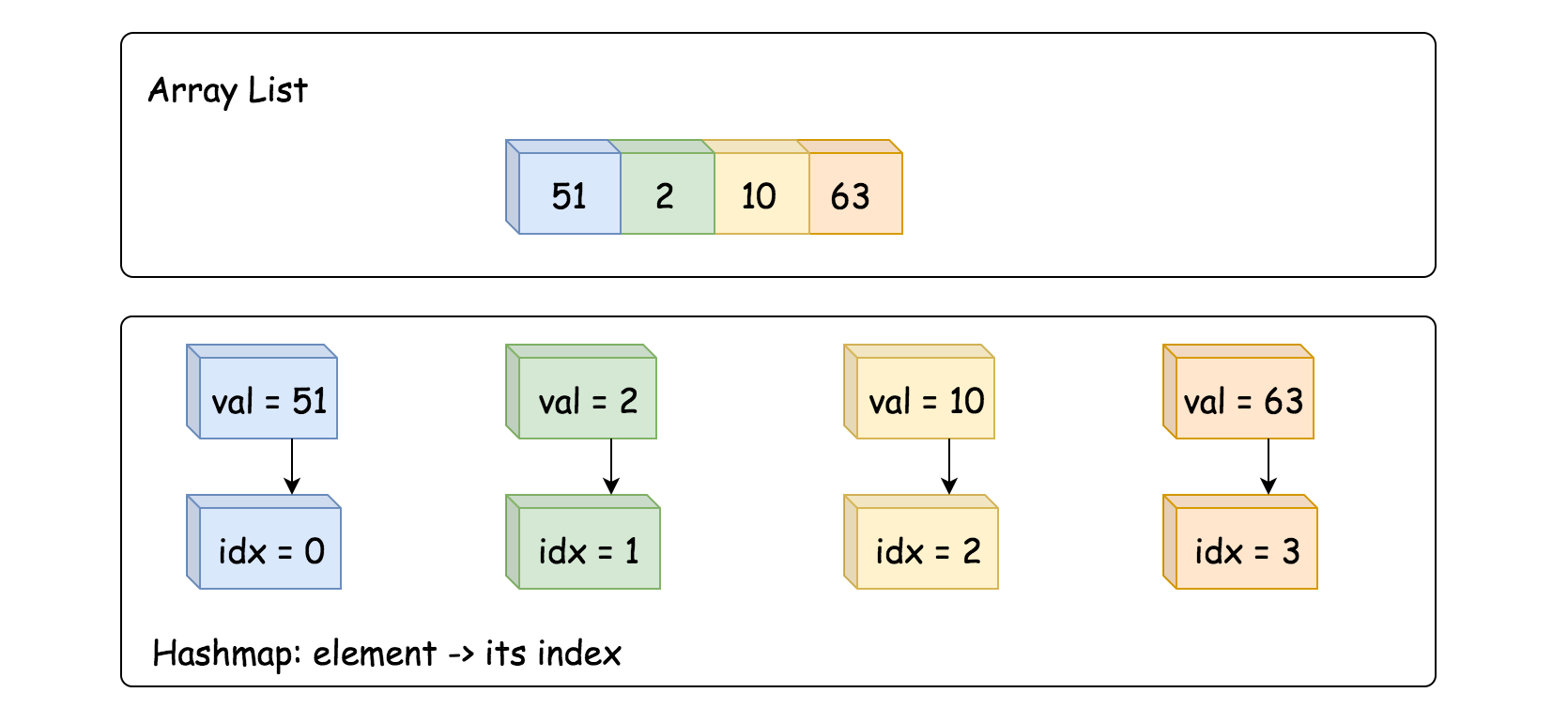
To delete a value at arbitrary index takes linear time. The solution here is to always delete the last value:

* Swap the element to delete with the last one.
* Pop the last element out.

For that, one has to compute an index of each element in constant time, and hence needs a hashmap which stores element -> its index dictionary.

Both ways converge into the same combination of data structures:

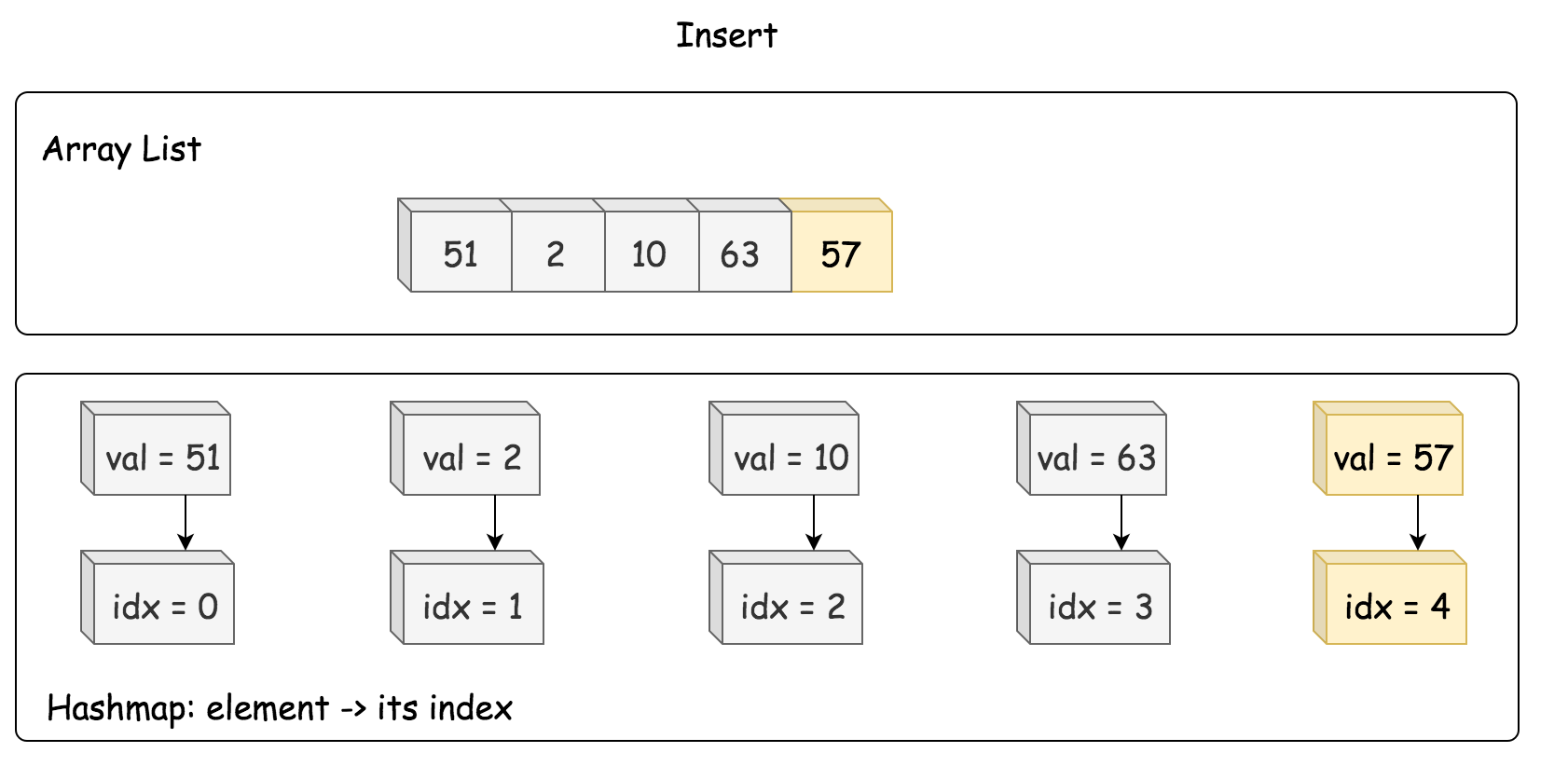
* Hashmap element -> its index.
* Array List of elements.



#### **Approach 1: HashMap + ArrayList**

**Insert**

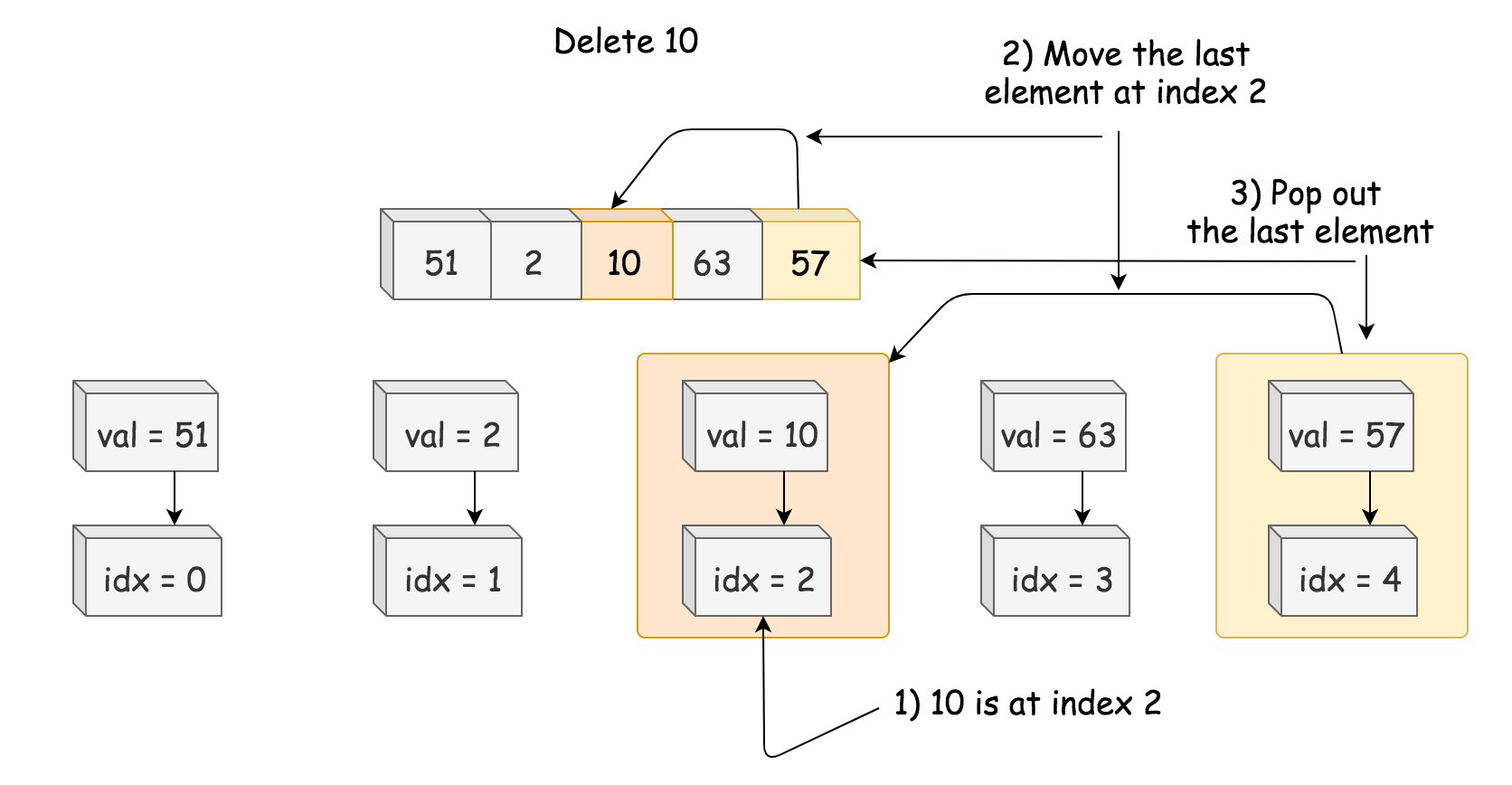
* Add value -> its index into dictionary, average \mathcal{O}(1)O(1) time.
* Append value to array list, average \mathcal{O}(1)O(1) time as well.



|  |
| --- |
| **/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/**  **public boolean insert(int val) {**  **if (dict.containsKey(val)) return false;**    **dict.put(val, list.size());**  **list.add(list.size(), val);**  **return true;**  **}** |

**Delete**

* Retrieve an index of element to delete from the hashmap.
* Move the last element to the place of the element to delete, \mathcal{O}(1)O(1) time.
* Pop the last element out, \mathcal{O}(1)O(1) time.



|  |
| --- |
| **/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/**  **public boolean remove(int val) {**  **if (! dict.containsKey(val)) return false;**  **// move the last element to the place idx of the element to delete**  **int lastElement = list.get(list.size() - 1);**  **int idx = dict.get(val);**  **list.set(idx, lastElement);**  **dict.put(lastElement, idx);**  **// delete the last element**  **list.remove(list.size() - 1);**  **dict.remove(val);**  **return true;**  **}** |

**GetRandom**

GetRandom could be implemented in \mathcal{O}(1)O(1) time with the help of standard random.choice in Python and Random object in Java.

|  |
| --- |
| /\*\* Get a random element from the set. \*/  public int getRandom() {  return list.get(rand.nextInt(list.size()));  } |

**Implementation**

|  |
| --- |
| class RandomizedSet {  Map<Integer, Integer> dict;  List<Integer> list;  Random rand = new Random();  /\*\* Initialize your data structure here. \*/  public RandomizedSet() {  dict = new HashMap();  list = new ArrayList();  }  /\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/  public boolean insert(int val) {  if (dict.containsKey(val)) return false;  dict.put(val, list.size());  list.add(list.size(), val);  return true;  }  /\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/  public boolean remove(int val) {  if (! dict.containsKey(val)) return false;  // move the last element to the place idx of the element to delete  int lastElement = list.get(list.size() - 1);  int idx = dict.get(val);  list.set(idx, lastElement);  dict.put(lastElement, idx);  // delete the last element  list.remove(list.size() - 1);  dict.remove(val);  return true;  }  /\*\* Get a random element from the set. \*/  public int getRandom() {  return list.get(rand.nextInt(list.size()));  }  } |

**Complexity Analysis**

* Time complexity. GetRandom is always \mathcal{O}(1)O(1). Insert and Delete both have \mathcal{O}(1)O(1) average time complexity, and \mathcal{O}(N)O(*N*) in the worst-case scenario when the operation exceeds the capacity of currently allocated array/hashmap and invokes space reallocation.
* Space complexity: \mathcal{O}(N)O(*N*), to store N elements.