Solution: Insert Delete GetRandom O(1)

Let's solve the Insert Delete GetRandom O(1) problem using the Custom Data Structures pattern.

We'll cover the following

Statement

Implement a **Random Set** data structure that can perform the following operations:

- **Constructor()**: This initializes the **Random Set** object.
- **Insert()**: This function takes an integer, **data**, as its parameter and, if it does not already exist in the set, add it to the set, returning TRUE. If the integer already exists in the set, the function returns FALSE.
- **Delete()**: This function takes an integer, **data**, as its parameter and, if it exists in the set, removes it, returning TRUE. If the integer does not exist in the set, the function returns FALSE.
- **GetRandom()**: This function takes no parameters. It returns an integer chosen at random from the set.

> **Note:** Your implementation should aim to have a running time of $O(1)$ (on average) for each operation.

Constraints:

- $-2^{31} \leq \textbf{data} \leq 2^{31}$
- No more than 2×10^5 calls will be made to the **Insert()**, **Delete()** and **GetRandom()** functions.
- There will be at least one element in the data structure when the **GetRandom()** function is called.

Solution

So far, you've probably brainstormed some approaches on how to solve this problem. Let's explore some of these approaches and figure out which one to follow while considering time complexity and any implementation constraints.

Naive approach

The naive approach to solve this problem is by using an array. Elements can be inserted and deleted from an array. Random values from an array can also be fetched by getting a random index first and then retrieving the array element.

The time complexity for this approach will be $O(n)$, but our approach should have a running time of $O(1)$ (on average) for each operation. Although the array supports constant time lookups, the deletion takes $O(n)$

time. Clearly, we need to look for a better approach.

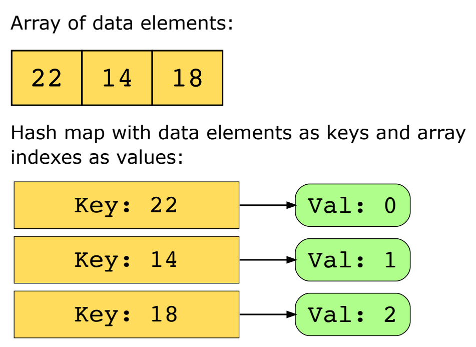
Optimized approach using an array and a hash map

This problem can’t be solved using a standard data structure, so we’ll need to design a custom data structure that meets these requirements. Arrays support constant time lookups, but deletion takes $O(n)$. A data structure that supports constant time deletion is the hash map. So, we’ll use arrays and hash maps to design our custom data structure.

Let’s consider the data structures with constant-time lookups, such as arrays and hash maps. Both of these data structures have their own advantages. To benefit from both, we’ll create a hybrid data structure to store the elements.

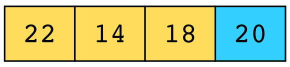
- **Insert():** When inserting data, we will store the new element in our array. And then, insert a key-value pair in our hash map, where the key will be the new element and the value will be its index in the array. Both of these operations have a time complexity of $O(1)$.
- **Delete():** Using the hash map, we find the index in the array at which the element is located. Swap the last element in the array with the one to be removed. Then, in the hash map, update the location of the element we just swapped with the target element, and remove the entry for the target element from the hash map. Lastly, pop out the target element from the array. Each of these five operations has a time complexity of $O(1)$.
- **GetRandom():** For this operation, we can choose a number at random between 0 and $n - 1$, where n is the size of the array. We return the integer located at this random index.

The slides below illustrate how our custom data structure will work:

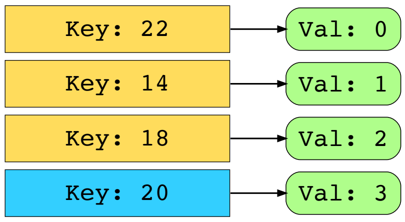


1 of 8

Array of data elements:



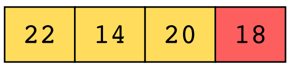
Hash map with data elements as keys and array indexes as values:



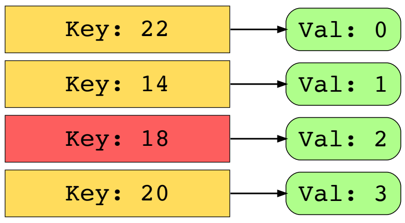
Insert(20) results in the value 20 being appended to the array, and the key-value pair of {20, 3} being added to the hash map.

2 of 8

Array of data elements:



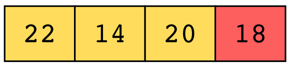
Hash map with data elements as keys and array indexes as values:



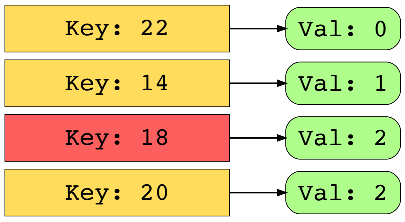
Delete(18) results in the last value in the array being swapped with the value at the original position of 18 (index 2).

3 of 8

Array of data elements:



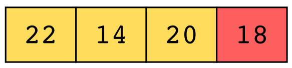
Hash map with data elements as keys and array indexes as values:



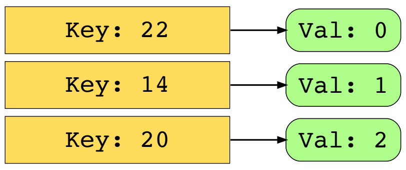
We will then update the value in the hash map for key=20, as its array index has been changed.

4 of 8

Array of data elements:



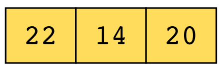
Hash map with data elements as keys and array indexes as values:



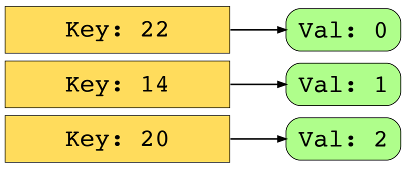
Remove the key-value pair from the hash map.

5 of 8

Array of data elements:



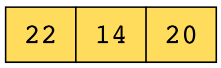
Hash map with data elements as keys and array indexes as values:



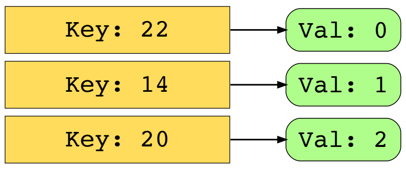
Lastly, remove the element from the array.

6 of 8

Array of data elements:

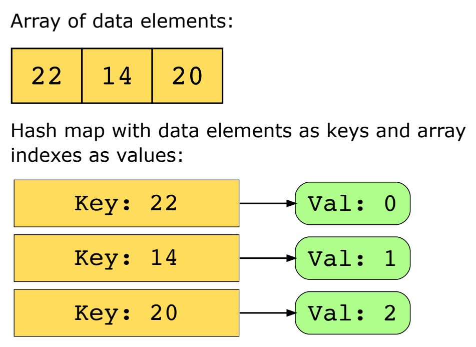


Hash map with data elements as keys and array indexes as values:



After removing 18, this will be the state of array and hash map.

7 of 8



When we call the **GetRandom()** function, it returns any one of the elements in the array.

8 of 8

Let's take a look at the code for this solution below:

Java

```java
class RandomSet{
    Map<Integer, Integer> dict;
    List<Integer> list;
    Random rand = new Random();

    /** Initialize your data structure here. */
    public RandomSet() {
        dict = new HashMap<>(); // maps actual value to its index
        list = new ArrayList<>(); // store actual values in an array
    }

    /** Inserts a value to the dataset. Returns true if the dataset did not already contain the specified
    public boolean insert(int val) {

        // Inserts a value in the data structure.
        // Returns True if it did not already contain the specified element.

        if (dict.containsKey(val))
            return false;
        dict.put(val, list.size());
        list.add(list.size(), val);
        return true;
    }

    /** Deletes a value from the dataset. Returns true if the dataset contained the specified value. */
    public boolean delete(int val) {
        // Deletes a value from the data structure.
```

Insert Delete GetRandom O(1)

Solution summary

To recap, the solution to this problem is:

For our setup, we store our data in an array and the location of each data element in a hash map.

- For insertion:
    - We append an element to the array.

- For deletion:
    - We use a hash map to find the index at which the element is located in our array.
    - After finding the position of the element in an array, swap the last element in the array with the one to be deleted.
    - Update the relevant key-value pair in the hash map so that its value is the new index of the element we just swapped with the target element.
    - Delete the key-value pair of the target element from the hash map and then delete the target element from our array.
- To get a random element, generate a random number in the range up to the amount of elements stored, and return the element in the array at this random index.

Time complexity

- **Constructor:** The time complexity of the constructor is $O(1)$.
- **Insert()** has an average time complexity of $O(1)$. We append the element to the end of the array and insert a key-value pair in a hash map, both of which are, on average, $O(1)$ operations.
- **Delete()** has an average time complexity of $O(1)$. We swap the value to delete with the value at the last index of the array. We update the index of the moved element in the hash map, and then pop out the value to remove from the end of the array. Each of these three operations has an average time complexity of $O(1)$.
- **GetRandom()** has an average time complexity of $O(1)$. We pick a number at random in the range between the lowest and highest indices of the array and use it as an index in the array. Both of these are $O(1)$ operations.

> Please note that the worst-case scenario for **Insert()** is $O(n)$, which occurs when space needs to be reallocated. This solution fulfills the stated requirements because the average time complexity remains $O(1)$.