380. Insert Delete GetRandom O(1)

Medium Design Array Hash Table Math Randomized

Leetcode Link

Problem Description

The `RandomizedSet` class is designed to perform operations on a collection of unique elements. It allows for the insertion and removal of elements and getting a random element from the set. The class methods which are required to be implemented are as follows:

- `insert(val)`: Adds the `val` to the set if it's not already present, and returns `true`; if `val` is already in the set, it returns `false`.
- `remove(val)`: Removes the `val` from the set if it's present and returns `true`; if `val` is not present in the set, it returns `false`.
- `getRandom()`: Returns a random element from the set, ensuring each element has an equal probability of being selected.

The constraint given is that each function should operate in average constant time, i.e., `O(1)` time complexity.

Intuition

The challenge lies in achieving `O(1)` time complexity for each operation - insert, remove, and getRandom. A standard set data structure wouldn't suffice for `getRandom()` to be `O(1)`. For efficient random access, we need to use a list structure where random access is `O(1)`. However, a list alone does not provide `O(1)` time complexity for insert and remove operations due to potential shifting of elements. To navigate this, we use a combination of a hash table (dictionary in Python) and a dynamic array (list in Python).

For insertion, a dynamic array (list) supports adding an element in `O(1)` time. To handle duplicates, we accompany the list with a hash table that stores elements as keys and their respective indices in the list as values. This simultaneously checks for duplicates and maintains the list of elements.

For removals, a list doesn't remove an element in `O(1)` time because it might have to shift elements. To circumvent this, we swap the element to be removed with the last element and then pop the last element from the list. This way, the removal doesn't require shifting all subsequent elements. After swapping and before popping, we must update the hash table accordingly to reflect the new index of the element that was swapped.

Getting a random element efficiently is accomplished with a list since we can access elements by an index in `O(1)` time. Since all elements in the set have an equal probability of being picked, we can select a random index and return the element at that index from the list.

Overall, the use of both data structures allows us to maintain the average constant time complexity constraint required by the problem for all operations, giving us an efficient solution that meets the problem requirements.

Solution Approach

The solution approach utilizes a blend of data structures and careful bookkeeping to ensure that each operation—insert, remove, and getRandom—executes in average constant `O(1)` time complexity.

Algorithms and Data Structures:

- **Hash Table/Dictionary (`self.d`)**: This hash table keeps track of the values as keys and their corresponding indices in the dynamic array as values. The hash table enables `O(1)` access time for checking if a value is already in the set and for providing values by looking up their index.
- **Dynamic Array/List (`self.q`)**: The dynamic array stores the elements of the set and allows us to utilize the `O(1)` access time to get a random element.

`insert` Implementation:

- Check if `val` is already in the `self.d` hash table. If it is, return `false` because no duplicate values are allowed in the set.
- If `val` is not present, add `val` as a key to `self.d` with the value being the current size of the list `self.q` (which will be the index of the inserted value).
- Then, append `val` to the list `self.q`.
- Return `true` because a new value has been successfully inserted into the set.

`remove` Implementation:

- Check if `val` is present in the `self.d` hash table. If not, return `false` because there's nothing to remove.
- If `val` is present, locate its index `i` in `self.q` using `self.d[val]`.
- Swap the value `val` in `self.q` with the last element in `self.q` to move `val` to the end of the list for O(1) removal.
- The hash table `self.d` needs to be updated to reflect the new index for the value that was swapped to the position previously held by `val`.
- Pop the last element from `self.q`, which is now `val`.
- Remove `val` from the hash table `self.d`.
- Return `true` because the value has been successfully removed from the set.

`getRandom` Implementation:

- Use Python's `choice` function from the `random` module to select a random element from the list `self.q`. The `choice` function inherently operates in `O(1)` time complexity because it selects an index randomly and returns the element at that index from the list.

This approach essentially provides an efficient and elegant solution to conduct insert, remove, and getRandom operations on a set with constant average time complexity, fulfilling the problem's constraints using the combination of a hash table with a dynamic array.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

1. Initialize:
   - We start by initializing our `RandomizedSet`. Both the dynamic array `self.q` and the hash table `self.d` are empty.
2. Insert 5:
   - Call `insert(5)`. Since 5 is not in `self.d`, we add it with its index to `self.d` (i.e., `self.d[5] = 0`) and append it to `self.q` (i.e., `self.q = [5]`).
   - The operation returns `true`.
3. Insert 10:
   - Call `insert(10)`. Similarly, since 10 is not in `self.d`, we add it with the next index to `self.d` (i.e., `self.d[10] = 1`) and append it to `self.q` (i.e., `self.q = [5, 10]`).
   - The operation returns `true`.
4. Insert 5 again:
   - Call `insert(5)`. Since 5 is already in `self.d`, we do not add it to `self.q` and return `false`.
5. Get a random element:
   - Call `getRandom()`. The function could return either 5 or 10, each with a 50% probability.
6. Remove 5:
   - Call `remove(5)`. We find the index of 5 from `self.d`, which is 0. We then swap `self.q[0]` (5) with the last element in `self.q` (which is 10), resulting in `self.q = [10, 5]`.
   - We update `self.d` to reflect the swap (now `self.d[10] = 0`), pop the last element in `self.q` (removing 5), and delete `self.d[5]`.
   - The operation returns `true`.
7. Current state:
   - The dynamic array `self.q` now holds `[10]`, and `self.d` holds `{10: 0}`.

This simple example demonstrates the processes of each operation and how the combination of a hash table and a dynamic array can achieve `O(1)` average time complexity for insertions, removals, and accessing a random element.

Python Solution

```python
from random import choice

class RandomizedSet:
    def __init__(self):
        self.index_dict = {}  # Mapping of values to their indices in the array
        self.values_list = []  # Dynamic array to hold the values

    def insert(self, val: int) -> bool:
        # Insert the value into the set if it's not already present, returning True if successful
        if val in self.index_dict:
            return False  # Value already exists
        self.index_dict[val] = len(self.values_list)  # Map value to its index in the array
        self.values_list.append(val)  # Add value to the array
        return True  # Insertion successful

    def remove(self, val: int) -> bool:
        # Remove the value from the set if present, returning True if successful
        if val not in self.index_dict:
            return False  # Value does not exist
        index_to_remove = self.index_dict.pop(val)  # Get the index of the value to remove
        last_element = self.values_list[-1]  # Get the last element in the array
        self.values_list[index_to_remove] = last_element  # Move the last element to the 'removed' position
        self.index_dict[last_element] = index_to_remove  # Update the last element's index
        self.values_list.pop()  # Remove the last element
        # Update the index of the moved value from the dictionary
        return True  # Removal successful

    def getRandom(self) -> int:
        # Return a random value from the set
        return choice(self.values_list)  # Randomly select and return a value from the array

# Example usage:
# randomized_set = RandomizedSet()
# param_1 = randomized_set.insert(val)
# param_2 = randomized_set.remove(val)
# param_3 = randomized_set.getRandom()
```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

// RandomizedSet design allows for O(1) time complexity for insertion, deletion and getting a random element.
class RandomizedSet {
    private Map<Integer, Integer> valueToIndexMap = new HashMap<>(); // Maps value to its index in 'valuesList'.
    private List<Integer> valuesList = new ArrayList<>(); // Stores the values.
    private Random randomGenerator = new Random(); // Random generator for getRandom() method.

    // Constructor of the RandomizedSet.
    public RandomizedSet() {
    }

    // Inserts a value to the set. Returns true if the set did not already contain the specified element.
    public boolean insert(int val) {
        if (valueToIndexMap.containsKey(val)) {
            // If the value is already present, return false.
            return false;
        }
        // Map the value to the size of the list which is the future index of this value.
        valueToIndexMap.put(val, valuesList.size());
        // Add the value to the end of the values list.
        valuesList.add(val);
        return true;
    }

    // Removes a value from the set. Returns true if the set contained the specified element.
    public boolean remove(int val) {
        if (!valueToIndexMap.containsKey(val)) {
            // If the value is not present, return false.
            return false;
        }
        // Get index of the element to remove.
        int indexToRemove = valueToIndexMap.get(val);
        // Get last element in the list.
        int lastElement = valuesList.get(valuesList.size() - 1);
        // Put the last element at the position of the element to remove.
        valuesList.set(indexToRemove, lastElement);
        // Update the map with the new index of lastElement.
        valueToIndexMap.put(lastElement, indexToRemove);
        // Remove the last element from the list.
        valuesList.remove(valuesList.size() - 1);
        // Remove the entry for the removed element from the map.
        valueToIndexMap.remove(val);
        return true;
    }

    // Get a random element from the set.
    public int getRandom() {
        // Returns a random element using the random generator.
        return valuesList.get(randomGenerator.nextInt(valuesList.size()));
    }
}

// The below comments describe how your RandomizedSet class could be used:
// boolean param_1 = obj.insert(val);
// boolean param_2 = obj.insert(val);
// boolean param_2 = obj.remove(val);
// int param_3 = obj.getRandom();
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <cstdlib>

class RandomizedSet {
public:
    RandomizedSet() {
        // Constructor doesn't need to do anything since the vector and
        // unordered_map are initialized by default
    }

    // Inserts a value to the set. Returns true if the set did not already contain the specified element
    bool insert(int val) {
        if (indexMap[val]) {
            // Value is already in the set, so insertion is not possible
            return false;
        }
        indexMap[val] = values.size(); // Map value to its index in 'values'
        values.push_back(val);         // Add value to the end of 'values'
        return true;
    }

    // Removes a value from the set. Returns true if the set contained the specified element
    bool remove(int val) {
        if (!indexMap.count(val)) {
            // Value is not in the set, so removal is not possible
            return false;
        }

        int index = indexMap[val];          // Get index of the element to remove
        indexMap[values.back()] = index;    // Map last element's index to the index of the one to be removed
        values[index] = values.back();      // Replace the element to remove with the last element
        values.pop_back();                  // Remove the last element
        indexMap.erase(val);                // Remove element from map

        return true;
    }

    // Gets a random element from the set
    int getRandom() {
        return values[rand() % values.size()]; // Return a random element by index
    }

private:
    std::unordered_map<int, int> indexMap; // Maps value to its index in 'values'
    std::vector<int> values;               // Stores the actual values
};

/**
 * Your RandomizedSet object will be instantiated and called as such:
 * RandomizedSet obj = new RandomizedSet();
 * bool param_1 = obj->insert(val);
 * bool param_2 = obj->remove(val);
 * int param_3 = obj->getRandom();
 */
```

Typescript Solution

```typescript
// Store the number and its corresponding index in the array
let valueToIndexMap: Map<number, number> = new Map();
// Store the numbers for random access
let valuesArray: number[] = [];

// Inserts a value to the set. Returns true if the set did not already contain the specified element.
function insert(value: number): boolean {
    if (valueToIndexMap.has(value)) {
        // Value already exists, so insertion is not done
        return false;
    }
    // Add value to the map and array
    valueToIndexMap.set(value, valuesArray.length);
    valuesArray.push(value);
    return true;
}

// Removes a value from the set. Returns true if the set contained the specified element.
function remove(value: number): boolean {
    if (!valueToIndexMap.has(value)) {
        // Value does not exist, hence, nothing to remove
        return false;
    }
    // Get index of the value to be removed
    const index = valueToIndexMap.get(value);
    // Move the last element to fill the gap of the removed element
    // Update the index of the last element in the map
    valueToIndexMap.set(valuesArray[valuesArray.length - 1], index);
    // Swap the last element with the one at the index
    valuesArray[index] = valuesArray[valuesArray.length - 1];
    // Remove the last element
    valuesArray.pop();
    // Delete the entry for the removed value from the map
    valueToIndexMap.delete(value);
    return true;
}

// Get a random element from the set.
function getRandom(): number {
    // Choose a random index and return the element at that index
    return valuesArray[Math.floor(Math.random() * valuesArray.length)];
}

// Usage example:
// var successfulInsert = insert(1); // returns true
// var successfulRemove = remove(2); // returns false
// var randomElement = getRandom(); // returns a random value from the set
```

Time and Space Complexity

Time Complexity:

- `insert(val: int) -> bool`: The insertion function consists of a dictionary check and insertion into a dictionary and a list, which are typically `O(1)` operations. Thus, the time complexity is `O(1)`.
- `remove(val: int) -> bool`: The remove function performs a dictionary check, index retrieval, and element swap in the list, as well as removal from both the dictionary and list. Dictionary and list operations involved here are usually `O(1)`. Therefore, the time complexity is `O(1)`.
- `getRandom() -> int`: The `getRandom` function makes a single call to the `choice()` function from the Python `random` module, which selects a random element from the list in `O(1)` time. Hence, the time complexity is `O(1)`.

Overall, each of the operations provided by the `RandomizedSet` class have `O(1)` time complexity.

Space Complexity:

- The RandomizedSet class maintains a dictionary (`self.d`) and a list (`self.q`). The dictionary maps element values to their indices in the list, and both data structures store up to `n` elements, where `n` is the total number of unique elements inserted into the set. Hence, the space complexity is `O(n)`, accounting for the storage of `n` elements in both the dictionary and the list.