2349. Design a Number Container System

Design a number container system that can do the following:

- **Insert** or **Replace** a number at the given index in the system.
- **Return** the smallest index for the given number in the system.

Implement the `NumberContainers` class:

- `NumberContainers()` Initializes the number container system.
- `void change(int index, int number)` Fills the container at `index` with the `number`. If there is already a number at that `index`, replace it.
- `int find(int number)` Returns the smallest index for the given `number`, or `-1` if there is no index that is filled by `number` in the system.

Example 1:

Input
```
["NumberContainers", "find", "change", "change", "change", "change", "find", "change", "find"]
[[], [10], [2, 10], [1, 10], [3, 10], [5, 10], [10], [1, 20], [10]]
```
Output
```
[null, -1, null, null, null, null, 1, null, 2]
```

Explanation
```
NumberContainers nc = new NumberContainers();
nc.find(10); // There is no index that is filled with number 10. Therefore, we return -1.
nc.change(2, 10); // Your container at index 2 will be filled with number 10.
nc.change(1, 10); // Your container at index 1 will be filled with number 10.
nc.change(3, 10); // Your container at index 3 will be filled with number 10.
nc.change(5, 10); // Your container at index 5 will be filled with number 10.
nc.find(10); // Number 10 is at the indices 1, 2, 3, and 5. Since the smallest index that is filled with 10 is 1, we return 1.
nc.change(1, 20); // Your container at index 1 will be filled with number 20. Note that index 1 was filled with 10 and then replaced with 20.
nc.find(10); // Number 10 is at the indices 2, 3, and 5. The smallest index that is filled with 10 is 2. Therefore, we return 2.
```

Constraints:

- `1 <= index, number <=` 10^9
- At most 10^5 calls will be made **in total** to `change` and `find`.

2349. Design a Number Container System

Overview

We need to design a number container system to efficiently manage and query numbers based on their indices. This system should support two primary operations:

1. **Inserting or Replacing a Number:** We can insert a number at a specific index, or replace the number already present at that index.
2. **Finding the Smallest Index of a Number:** We need to retrieve the smallest index where a given number is present. If the number doesn't exist in the system, we should return `-1`.

To achieve this, we need to implement a class `NumberContainers` with the following methods:

- **`NumberContainers()`**: Initializes the container system. This involves setting up the internal data structures to store the mappings between numbers and indices.
- **`void change(int index, int number)`**: Updates the system by associating the given `number` with the provided `index`. If the index already contains a number, it should be replaced. If this operation introduces new data or modifies existing mappings, the system must ensure consistency for subsequent queries.
- **`int find(int number)`**: Returns the smallest index where the specified `number` exists. If the number is not present, it returns `-1`.

Approach 1: Two Maps

Intuition

We need to focus on two main operations: `change`, which allows us to insert or replace a number at a specific index, and `find`, which retrieves the smallest index associated with a given number.

The key to implementing these operations efficiently lies in using map data structures:

1. **`indexToNumber`**: This map holds the relationship between an index and the number currently stored at that index. It allows us to quickly check if an index already contains a number and enables efficient replacement during the `change` operation.
2. **`numberToIndices`**: This map keeps track of the indices where each number is present. By using a set to store these indices, we ensure that they remain automatically sorted, enabling efficient insertion and retrieval of the smallest index for a number.

With these structures in mind, let’s break the solution into two parts: first, the `change` operation, and second, the `find` operation.

1. Change Operation (Insertion and Replacement)

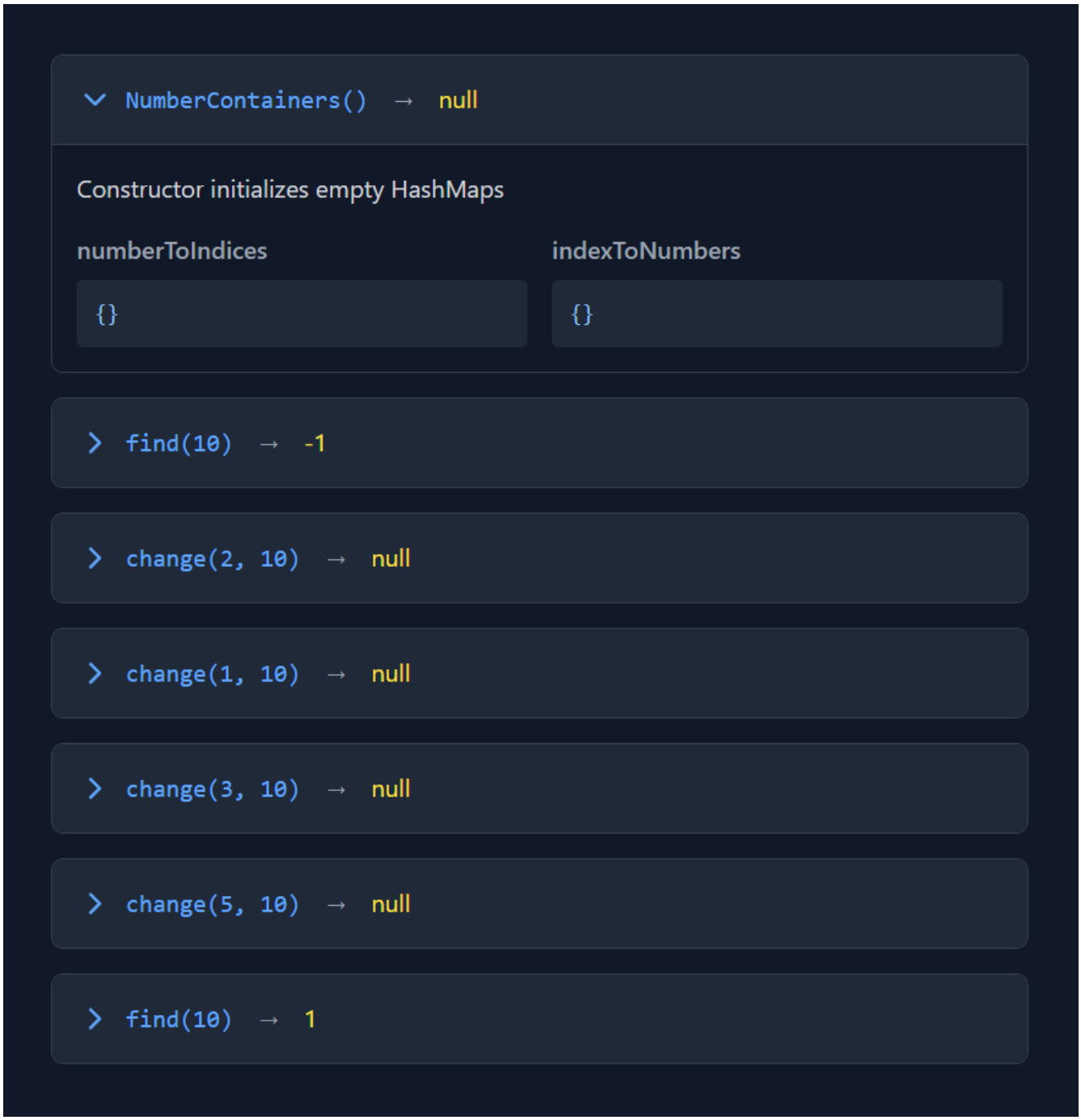
The `change` operation begins by checking if the given index already holds a number. If the index does contain a number, we first remove this index from the set of indices associated with the old number in the `numberToIndices` map. This step ensures that the old number no longer references the index after the replacement. Once the index is removed, we check whether the set for the old number has become empty. If it has, we remove the old number entirely from the map to maintain a clean and efficient data structure.

After handling the removal, we proceed to insert the new number at the given index. This involves adding the index to the set of indices for the new number in `numberToIndices`. Because we are using a set, the indices remain sorted automatically, allowing us to avoid any additional effort to manage their order. This also prepares us for the `find` operation, where the smallest index will always be readily accessible.

2. Find Operation (Retrieve Smallest Index)

For the `find` operation, we need to return the smallest index that contains the given number. To achieve this, we check the `numberToIndices` map. If the number isn't found, we return `-1`, indicating that the number is not present. If the number exists, the smallest index will always be the first element in the set of indices (since sets store elements in ascending order). This allows us to quickly return the result with minimal effort.

The algorithm is visualized below:


NumberContainers() → null
Constructor initializes empty HashMaps
numberToIndices
{}
indexToNumbers
{}
find(10) → -1
change(2, 10) → null
change(1, 10) → null
change(3, 10) → null
change(5, 10) → null
find(10) → 1


NumberContainers() → null
find(10) → -1
No entries for number 10
numberToIndices
{}
indexToNumbers
{}
change(2, 10) → null
change(1, 10) → null
change(3, 10) → null
change(5, 10) → null
find(10) → 1

> `NumberContainers()` → `null`

> `find(10)` → `-1`

v `change(2, 10)` → `null`

Added first mapping for 10

numberToIndices

```
{
  "10": [
    2
  ]
}
```

indexToNumbers

```
{
  "2": 10
}
```

> `change(1, 10)` → `null`

> `change(3, 10)` → `null`

> `change(5, 10)` → `null`


change(1, 10) → null
Added index 1 to number 10
numberToIndices
{
"10": [
1,
2
]
}
indexToNumbers
{
"1": 10,
"2": 10
}
change(3, 10) → null
change(5, 10) → null
find(10) → 1
change(1, 20) → null
find(10) → 2

change(3, 10) → null

Added index 3 to number 10

numberToIndices

```
{
  "10": [
    1,
    2,
    3
  ]
}
```

indexToNumbers

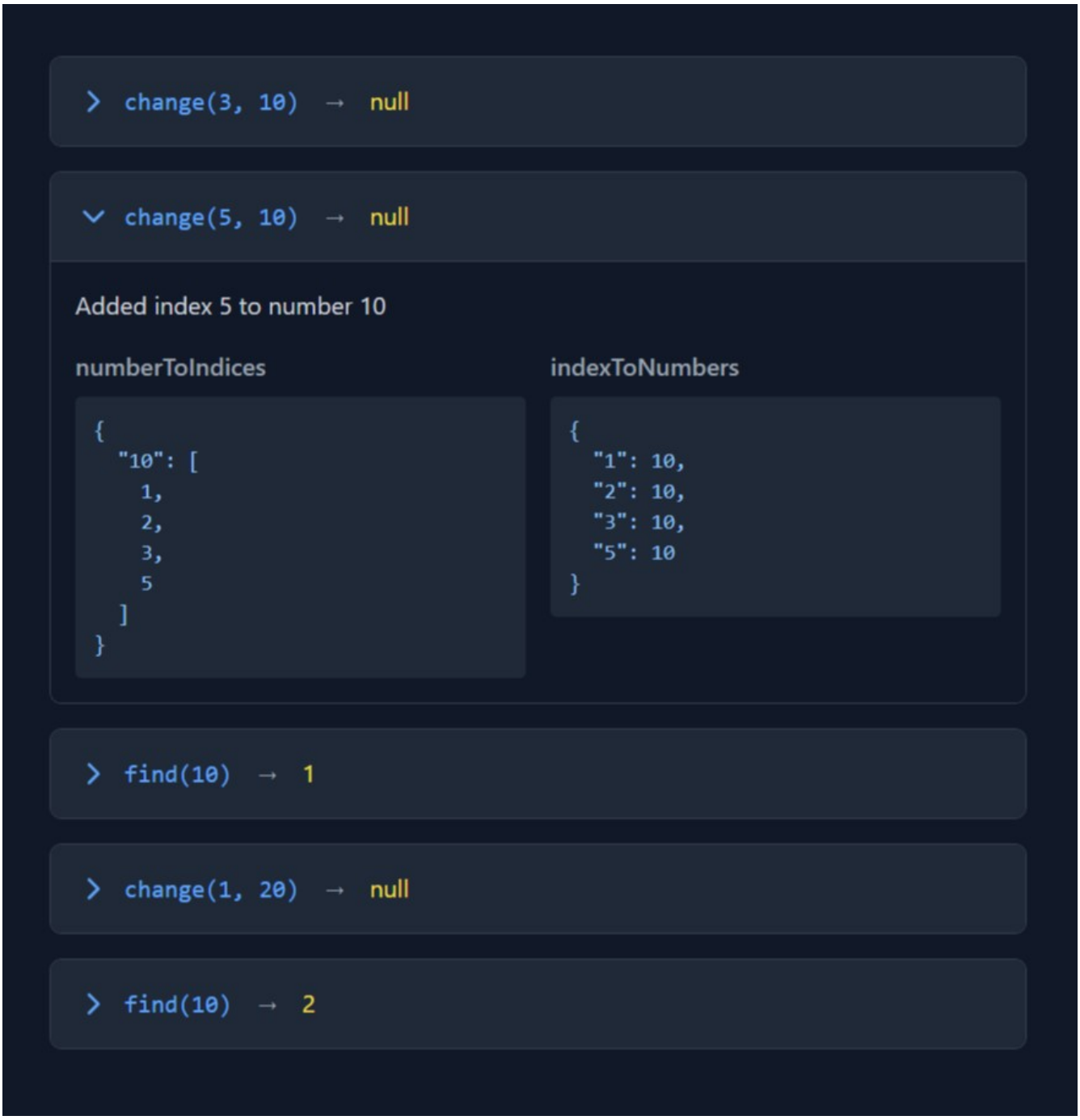
```
{
  "1": 10,
  "2": 10,
  "3": 10
}
```

change(5, 10) → null

find(10) → 1

change(1, 20) → null

find(10) → 2


> change(3, 10) → null
change(5, 10) → null
Added index 5 to number 10
numberToIndices
{
"10": [
1,
2,
3,
5
]
}
indexToNumbers
{
"1": 10,
"2": 10,
"3": 10,
"5": 10
}
> find(10) → 1
> change(1, 20) → null
> find(10) → 2


> change(3, 10) → null
> change(5, 10) → null
find(10) → 1
Returns smallest index for number 10
numberToIndices
{
"10": [
1,
2,
3,
5
]
}
indexToNumbers
{
"1": 10,
"2": 10,
"3": 10,
"5": 10
}
> change(1, 20) → null
> find(10) → 2

> change(5, 10) → null

> find(10) → 1

> change(1, 20) → null

Moved index 1 from number 10 to 20

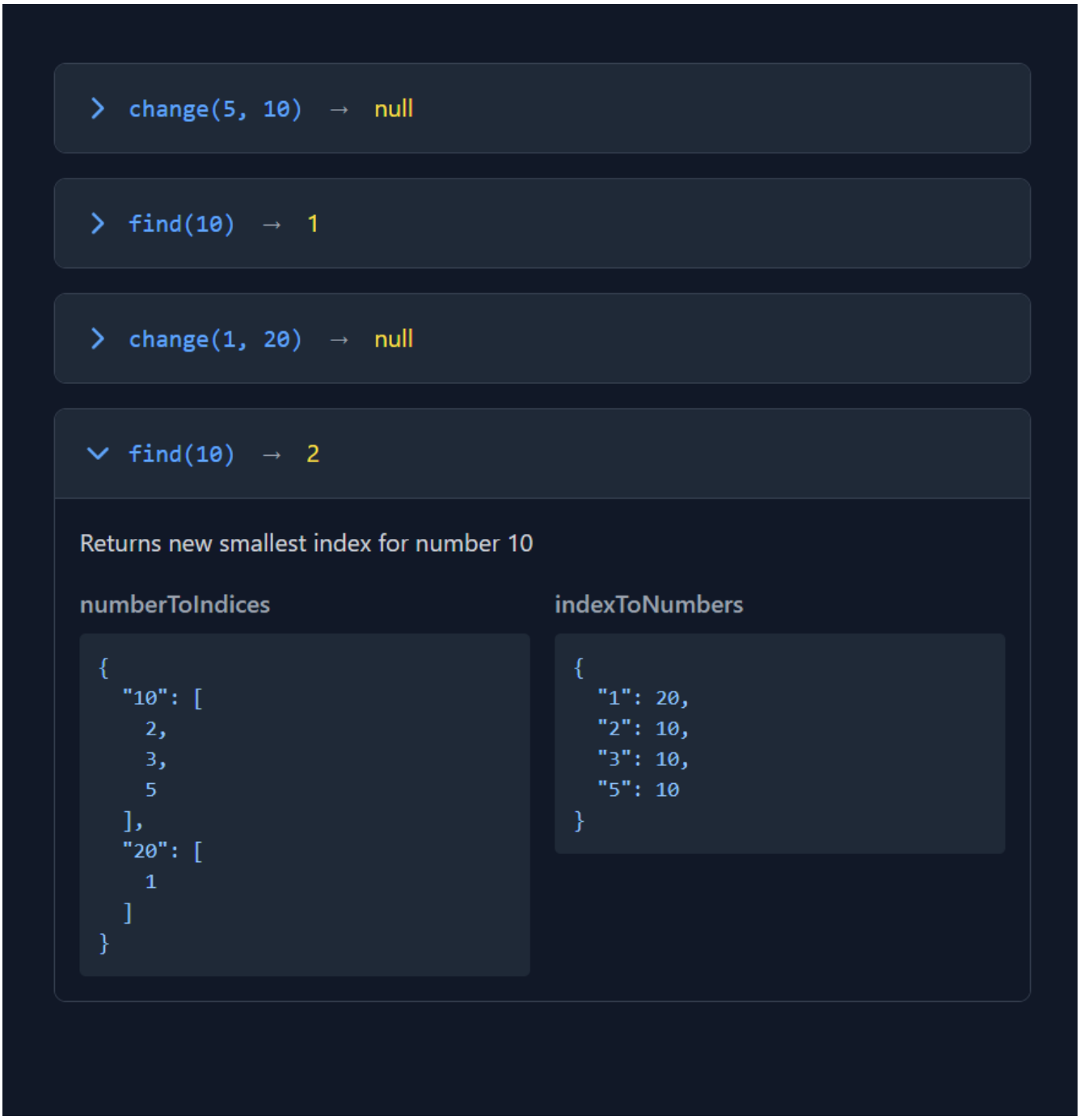
numberToIndices

```
{
  "10": [
    2,
    3,
    5
  ],
  "20": [
    1
  ]
}
```

indexToNumbers

```
{
  "1": 20,
  "2": 10,
  "3": 10,
  "5": 10
}
```

> find(10) → 2


> change(5, 10) → null
> find(10) → 1
> change(1, 20) → null
find(10) → 2
Returns new smallest index for number 10
numberToIndices
{
"10": [
2,
3,
5
],
"20": [
1
]
}
indexToNumbers
{
"1": 20,
"2": 10,
"3": 10,
"5": 10
}

2349. Design a Number Container System

```
/*
   Hash maps+ordered set
   Time complexity:
   ***change(int,int): O(logn)
   ***find(int): O(1)
   ***overall: O(q.logn), q:#queries
   Overall space complexity: O(n)
*/
class NumberContainers {
   private:
      std::unordered_map<int,std::set<int>> number_indexes;
      std::unordered_map<int,int> index_number;

   public:
      NumberContainers() {

      }

      // O(logn)
      void change(int index, int number) {
         if(index_number.find(index)!=index_number.end()){
            int old_number=index_number[index];
            auto it=number_indexes[old_number].find(index);
            if(it!=number_indexes[old_number].end()) number_indexes[old_number].erase(it);
         }
         index_number[index]=number;
         number_indexes[number].insert(index);
      }


      // O(1)
      int find(int number) {
         return !number_indexes[number].empty()?*number_indexes[number].begin():-1;
      }
};
```