2349. Design a Number Container System

Medium | Design | Hash Table | Ordered Set | Heap (Priority Queue) — Leetcode Link

Problem Description

The LeetCode problem entails designing a system that can manage a set of numbers each located at a unique index. The system needs to support two primary operations:

1. `change`: This operation must insert or replace a number at a specified index. If an index already contains a number, it should be replaced with the new number.
2. `find`: This operation should return the smallest index at which a specific number is located. If the number is not present in any container, the system should return `-1`.

The challenge is to implement these operations efficiently, meaning we need to optimize for both insert/replace and search operations, ensuring that `find` queries are performed in an optimized manner, especially in a dataset where searches are frequent compared to updates.

Intuition

To arrive at the solution, we need a data structure that can efficiently keep track of each number's indices and quickly retrieve the smallest index of any given number. We use two main data structures to achieve this:

1. A hash map (`self.mp`), where the key is the index and the value is the number at that index. This allows us to quickly check if an index already has a number and to update the number at any index.
2. A default dictionary of `SortedSet` (`self.t`), indexed by numbers. `SortedSet` is a data structure that maintains the elements in sorted order. This allows us to quickly retrieve the smallest index for a given number as it will always be the first element.

When we perform the `change` operation, we update the index's number in the hash map. If the index already had a number, we remove the index from the SortedSet of that old number. Then we add the index to the SortedSet of the new number.

During the `find` operation, we look up the SortedSet for the given number. If the SortedSet is not empty, we return the first element (since it's the smallest index due to the properties of the SortedSet). If the SortedSet is empty, it means the number is not present at any index, so we return `-1`.

This approach allows us to efficiently update the indices and retrieve the smallest index for any number, making the NumberContainers class a fast and reliable system for the operations required.

Solution Approach

The implementation of the NumberContainers class uses a hash map and a default dictionary of SortedSet structures. The `SortedSet` is chosen for its properties of maintaining the elements in sorted order, which is essential for efficient retrieval of indices.

Here's a step-by-step breakdown of the implementation:

1. **Initialization (`__init__` method):**
   - A hash map `self.mp` is initialized to keep track of which number is at which index.
   - A default dictionary `self.t` of `SortedSet` is initialized to store sets of indices for each number. The `defaultdict` from the Python `collections` module ensures that each number key in the dictionary will automatically be associated with an empty `SortedSet` if it does not already exist in the dictionary.
2. **Change operation (`change` method):**
   - Given an `index` and a `number`:
     - If the `index` already exists in the hash map (indicating a number is already present at that index), remove the index from the `SortedSet` of the current number associated with that index.
     - Update the hash map with the new `number` at the `index`.
     - Add the `index` to the `SortedSet` of the new number. This operation automatically keeps the `SortedSet` in sorted order.

   Python code snippet for the change operation:

```python
def change(self, index: int, number: int) -> None:
    if index in self.mp:
        v = self.mp[index]
        self.t[v].remove(index)
    self.mp[index] = number
    self.t[number].add(index)
```

3. **Find operation (`find` method):**
   - To find the smallest index for a given `number`:
     - Retrieve the `SortedSet` associated with the `number` from `self.t`.
     - If the `SortedSet` is not empty, return the first element (smallest index) since the indices are maintained in sorted order.
     - If the `SortedSet` is empty, return `-1` indicating the `number` is not present at any index.

   Python code snippet for the find operation:

```python
def find(self, number: int) -> int:
    s = self.t[number]
    return s[0] if s else -1
```

The choice of `SortedSet` is crucial for the solution's efficiency. `SortedSet` allows for fast addition and removal of index elements while maintaining a sort order, making the `find` operation a simple and quick retrieval of the smallest element. This solution thus capably balances the need for dynamic updates with frequent and fast queries, delivering the functionalities required by the NumberContainers system.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we initialize our `NumberContainers` class and the following operations are performed sequentially:

1. `change(2, 1)`: We want to insert number `1` at index `2`.
   - Since index `2` is not yet mapped, we simply insert it into `self.mp`: `self.mp` becomes `{2: 1}`.
   - We add the index to the `SortedSet` of number `1` in `self.t`: `self.t[1]` now contains `{2}`.
2. `change(1, 1)`: Now, we insert number `1` at index `1`.
   - We add the new key-value pair to `self.mp`: `self.mp` becomes `{2: 1, 1: 1}`.
   - Index `1` is added into the `SortedSet` of number `1` in `self.t`: `self.t[1]` now contains `{1, 2}`, automatically sorted.
3. `change(2, 2)`: Number `2` is now inserted at index `2`, which already has a number `1`.
   - We remove index `2` from the `SortedSet` of the old number, `self.t[1]`, which then becomes `{1}`.
   - `self.mp` is updated to reflect the change: `self.mp` becomes `{2: 2, 1: 1}`.
   - Index `2` is added to the `SortedSet` of number `2` (which was an empty `SortedSet` before): `self.t[2]` now contains `{2}`.
4. `find(1)`: We want to find the smallest index where number `1` is located.
   - Lookup `self.t[1]`, which contains `{1}`.
   - The smallest index in the sorted set is the first element, which is `1`, thus we return `1`.
5. `find(3)`: We want to find the smallest index where number `3` is located.
   - Since number `3` has never been inserted, `self.t[3]` is an empty `SortedSet`.
   - We return `-1`, indicating that number `3` is not present at any index.

The sequence of these operations would result in the following internal states for our `NumberContainers` class:

- Hash map of index to numbers: `self.mp = {1: 1, 2: 2}`
- Default dictionary of number to SortedSets of indices: `self.t = {1: SortedSet([1]), 2: SortedSet([2])}`

This example demonstrates how each `change` and `find` operation works as expected, providing efficient updates and lookups as per the problem requirements.

Python Solution

```python
# Import the defaultdict from collections and SortedSet from sortedcontainers
from collections import defaultdict
from sortedcontainers import SortedSet

class NumberContainers:
    def __init__(self):
        # Dictionary to hold the current number at each index
        self.index_to_number = {}
        # Defaultdict to hold SortedSets which will contain indices for each number
        # SortedSets are used for maintaining the indices in sorted order
        self.number_to_indices = defaultdict(SortedSet)

    def change(self, index: int, number: int) -> None:
        """Update the number at the given index and maintain the indices sorted."""
        # If the index is already in our mapping, remove it from the current number's SortedSet
        if index in self.index_to_number:
            current_number = self.index_to_number[index]
            self.number_to_indices[current_number].remove(index)

        # Update the number at the index in our index_to_number mapping
        self.index_to_number[index] = number
        # Add the index to the new number's SortedSet
        self.number_to_indices[number].add(index)

    def find(self, number: int) -> int:
        """Returns the smallest index for the given number; if not found, returns -1."""
        # Get the SortedSet of indices for the given number
        indices = self.number_to_indices[number]
        # If there are any indices, return the smallest one (first element)
        # If not, return -1 indicating the number is not assigned to any index
        return indices[0] if indices else -1

# The NumberContainers class can now be instantiated and methods called as described.
# obj = NumberContainers()
# obj.change(index, number)
# param_2 = obj.find(number)

```

Java Solution

```java
import java.util.HashMap;
import java.util.Map;
import java.util.TreeSet;

class NumberContainers {
    private final Map<Integer, Integer> indexToNumberMap = new HashMap<>();
    private final Map<Integer, TreeSet<Integer>> numberToIndicesMap = new HashMap<>();

    // Constructor
    public NumberContainers() {
        // Intentionally left blank, no initialization needed here
    }

    /**
     * Updates the number at a given index and maintains the mapping of numbers to a sorted set of indices.
     *
     * @param index  The index to change.
     * @param number The new number to associate with the index.
     */
    public void change(int index, int number) {
        // If index already contains a number, update the mapping
        if (indexToNumberMap.containsKey(index)) {
            int currentNumber = indexToNumberMap.get(index);
            // Remove the index from the current number's set
            TreeSet<Integer> indicesSet = numberToIndicesMap.get(currentNumber);
            indicesSet.remove(index);
            // If the set is empty after removal, remove it from the map
            if (indicesSet.isEmpty()) {
                numberToIndicesMap.remove(currentNumber);
            }
        }
        // Add or update the index-to-number mapping
        indexToNumberMap.put(index, number);
        // Add index to the new number's set, creating the set if it doesn't exist
        numberToIndicesMap.computeIfAbsent(number, k -> new TreeSet<>()).add(index);
    }

    /**
     * Finds the lowest index for a number.
     * If the number is not associated with any index, returns -1.
     *
     * @param number The number to find the lowest index for.
     * @return The lowest index of the given number or -1 if not found.
     */
    public int find(int number) {
        // Check if number exists in the map and return the first (lowest) index
        return numberToIndicesMap.containsKey(number) ? numberToIndicesMap.get(number).first() : -1;
    }
}

// The NumberContainers object usage remains the same; example of instantiation and method calls:
// NumberContainers obj = new NumberContainers();
// obj.change(index,number);
// int lowestIndex = obj.find(number);

```

C++ Solution

```cpp
#include <map>
#include <set>

class NumberContainers {
public:
    // Maps each index to the number it contains.
    std::map<int, int> indexToNumberMap;

    // Maps each number to a set of indices that contain this number.
    std::map<int, std::set<int>> numberToIndicesMap;

    // Default constructor.
    NumberContainers() {
    }

    // Changes the number at the given index.
    void change(int index, int number) {
        // Check if the index is already in the map.
        auto it = indexToNumberMap.find(index);
        if (it != indexToNumberMap.end()) {
            // If the index is already there, remove the index from the set corresponding
            // to its current number, as it's about to be reassigned.
            numberToIndicesMap[it->second].erase(index);
            // Update the index to the new number.
            it->second = number;
        } else {
            // If the index is new, just add it to the map.
            indexToNumberMap[index] = number;
        }
        // Insert the index to the set corresponding to the new number.
        numberToIndicesMap[number].insert(index);
    }

    // Finds the smallest index that contains the given number. Returns -1 if such an index cannot be found.
    int find(int number) {
        // Attempt to find the set of indices for the given number.
        auto it = numberToIndicesMap.find(number);
        // If the number is not found or the set is empty, return -1.
        return (it == numberToIndicesMap.end() || it->second.empty()) ? -1 : *it->second.begin();
    }
};

/**
 * The NumberContainers object will be instantiated and called like this:
 * NumberContainers* obj = new NumberContainers();
 * obj->change(index, number);
 * int param_2 = obj->find(number);
 */

```

Typescript Solution

```typescript
// Mapping from index to number.
const indexToNumberMap: Map<number, number> = new Map();

// Mapping from number to a set of indices containing that number.
const numberToIndicesMap: Map<number, Set<number>> = new Map();

/**
 * Changes the number at the given index.
 * @param index The index of the number to change.
 * @param number The new number to set at the index.
 */
function change(index: number, number: number): void {
    // Check if the index already maps to a number.
    if (indexToNumberMap.has(index)) {
        // Retrieve the current number at the index.
        const currentNumber = indexToNumberMap.get(index);
        // Remove index from the set of the current number.
        numberToIndicesMap.get(currentNumber)?.delete(index);
    }
    // Update the index to the new number.
    indexToNumberMap.set(index, number);

    // If there's no existing set for this number, create one.
    if (!numberToIndicesMap.has(number)) {
        numberToIndicesMap.set(number, new Set());
    }
    // Add the index to the set corresponding to the new number.
    numberToIndicesMap.get(number)?.add(index);
}

/**
 * Finds the smallest index that contains the given number.
 * @param number The number to find the smallest index for.
 * @returns The smallest index containing the given number, or -1 if not found.
 */
function find(number: number): number {
    // Attempt to find the set of indices for the given number.
    const indicesSet = numberToIndicesMap.get(number);
    // If there's no set for the number or the set is empty, return -1.
    if (!indicesSet || indicesSet.size === 0) {
        return -1;
    }
    // Return the smallest index (first value) in the sorted set.
    return Math.min(...indicesSet);
}

// Example usage:
// change(1, 10);
// let smallestIndex = find(10);
// console.log(smallestIndex); // Outputs the smallest index where the number 10 is located, if present.

```

Time and Space Complexity

The given Python code defines a class `NumberContainers` that manages a mapping between indices and numbers and provides two methods: `change` to change the number at a given index, and `find` to find the smallest index with a given number.

Time complexity:

- `__init__` function: The initialization function has a constant time complexity of $O(1)$, as it only involves the initialization of two data structures: a dictionary `self.mp` and a defaultdict of SortedSet `self.t`.
- `change` function: The `change` method has a time complexity of $O(\log n)$ for updating the SortedSet in the case where the index already exists in `self.mp`, since SortedSet removes the element in $O(\log n)$ and adds the element in $O(\log n)$ time complexity. If an element is not present, adding to a SortedSet is $O(\log n)$ as well. Updating the dictionary `self.mp` has a time complexity of $O(1)$.
- `find` function: The `find` method has a time complexity of $O(1)$ for accessing the first element of the SortedSet since SortedSet maintains the elements in sorted order, and accessing the elements by index is done in constant time.

Space complexity:

- `__init__` function: The space complexity for the initialization function is $O(1)$ as it initializes empty data structures.
- `change` and `find` functions: The space complexity is $O(m + k)$ where m is the number of unique indices and k is the total number of unique numbers present in the NumberContainers object. This is because `self.mp` can potentially hold a mapping for each unique index and `self.t` holds unique numbers each associated with a SortedSet which in turn contains indices.

Overall, this data structure is optimized for quick updates and lookups, with the SortedSet providing efficient ordering for the indices.