2336. Smallest Number in Infinite Set

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

Problem Description

In this LeetCode problem, we are required to implement a class called `SmallestInfiniteSet`. This class simulates a set that contains all positive integers starting from 1 and extending to infinity. The class has to provide two operations:

1. `popSmallest()`: This operation should remove the smallest number currently in the set and return it. The initial call to this method would return 1, the second call would return 2, and so on. After a number is popped, it is no longer present in the set unless it is added back.

2. `addBack(num)`: This operation allows one to add a number back into the set. However, a number can be added back only if it has been previously popped from the set and is not currently present.

The challenge is to code these operations efficiently, keeping in mind that there can be an infinite number of integers.

Intuition

The solution to this problem relies on maintaining a auxiliary set to keep track of the numbers that have been popped. We can call this auxiliary set "blacklist" or `black` in the code sample given. When `popSmallest()` is called, we start from 1 and check if that number is in the `black` set. If it is not, we pop it (remove and return) and add it to the `black` set. If it is, we increment the number and check again until we find the smallest number not in the `black` set.

When `addBack()` is called with a number argument, we remove it from the `black` set, which represents adding it back into the set of available numbers to be popped. This is because a number can only be re-added if it has been removed before, and removing from the `black` will allow `popSmallest()` to find it again as the smallest available number.

This approach is intuitive in that it treats the infinite set of positive integers as an implicit list that we only modify by noting which elements are currently not in the set (which are in `black`). It uses set operations which are typically O(1) for existence check and removal, making the operations efficient.

Solution Approach

Let's discuss the implementation details of the `SmallestInfiniteSet` class and walk through each method provided in the solution:

Initializer `__init__(self)`:

In Python, the `__init__` method serves as an initializer or a constructor for a class. It is automatically invoked when a new instance of the class is created.

For our `SmallestInfiniteSet` class:

```python
def __init__(self):
    self.black = set()
```

This initializer sets up an empty set called `self.black`. This set will hold all numbers that have been popped and removed from the infinite set of positive integers.

- Data Structure: `set()` is used because it allows fast addition, removal, and membership check operations with an average time complexity of O(1).

Method `popSmallest(self) -> int`:

The `popSmallest` method is used to pop and return the smallest integer from the infinite set. The pseudo-algorithm for this method is:

1. Start with `i = 1` since 1 is the smallest positive integer.
2. Increment `i` while `i` is in `self.black`. This is because if `i` is in `self.black`, it means that `i` has been previously popped.
3. Once an `i` is found that is not in `self.black`, add it to `self.black` and return it.

```python
def popSmallest(self) -> int:
    i = 1
    while i in self.black:
        i += 1
    self.black.add(i)
    return i
```

- Algorithm: Linear search to find the smallest integer not in `self.black`.
- Pattern Used: Iteration starting from 1 and incrementing until an integer not in `self.black` is found.

This method ensures that we always return the next smallest number, as per the requirement of the problem.

Method `addBack(self, num: int) -> None`:

The `addBack` method is meant to add a number back into the set if it isn't currently in the set:

```python
def addBack(self, num: int) -> None:
    self.black.discard(num)
```

- Data Structure: We use the `discard` method of the set, which removes `num` if present in `self.black`. This operation does not raise an error even if `num` is not present.
- Pattern Used: Conditional removal, letting the set data structure handle the existence check implicitly.

The situation where `addBack` is called with a number that has never been popped or is already in the set doesn't change the `self.black` set, as per the problem's instructions.

These methods collectively allow us to efficiently simulate the infinite set operations of popping the smallest element and adding numbers back into the infinite set.

Example Walkthrough

Let's use a small example to illustrate how the `SmallestInfiniteSet` class works according to the solution approach. Assume we create an instance of `SmallestInfiniteSet` and then perform a series of operations:

1. Call `popSmallest()` four times.
2. Call `addBack(2)`.
3. Call `popSmallest()` two more times.

Let's go step by step:

Step 1

We first instantiate our `SmallestInfiniteSet` class. The `black` set is initialized as empty.

Step 2

We call `popSmallest()` for the first time. Since `black` is empty, 1 is not in `black` and is returned. `black` is now `{1}`.

Step 3

We call `popSmallest()` for the second time. Here `i` starts at 1 again but since 1 is in `black`, increment `i` to 2. Now 2 is not in `black`, so 2 is returned and `black` becomes `{1, 2}`.

Step 4

We call `popSmallest()` for the third time. `i` starts at 1, but both 1 and 2 are in `black`, so we increment `i` to 3. Three is returned and `black` becomes `{1, 2, 3}`.

Step 5

We call `popSmallest()` for the fourth time. `i` starts at 1, but numbers 1, 2, and 3 are in `black`, so we increment `i` to 4. Four is returned and `black` becomes `{1, 2, 3, 4}`.

Step 6

Now, we call `addBack(2)`. This removes 2 from `black`. Therefore, `black` is now `{1, 3, 4}`.

Step 7

We call `popSmallest()` again. We start at 1, which is in `black`. We find that 2 is now not in `black` (because we just added it back), so we return 2, and `black` becomes `{1, 2, 3, 4}`.

Step 8

One final call to `popSmallest()` starts at 1 again, but we must increment until we find 5, which is not in `black`. We return 5 and `black` becomes `{1, 2, 3, 4, 5}`.

This walkthrough exemplifies how an implied infinite set of positive integers can be manipulated using a `black` set to track which elements have been 'popped' and are not currently available unless added back. By incrementing the minimal candidate and checking against `black`, we ensure that `popSmallest()` always returns the smallest available integer, and by using `discard` in `addBack`, we maintain an efficient model of this infinite integer set.

Python Solution

```python
class SmallestInfiniteSet:
    def __init__(self):
        # Initialize an empty set to keep track of popped elements
        self.popped_elements = set()

    def popSmallest(self) -> int:
        # Starting from 1, find the smallest integer not yet popped
        smallest = 1
        while smallest in self.popped_elements:
            smallest += 1
        # Add the found integer to the popped elements set
        self.popped_elements.add(smallest)
        # Return the smallest integer
        return smallest

    def addBack(self, num: int) -> None:
        # If the number is in the popped elements, remove it to make it available again
        if num in self.popped_elements:
            self.popped_elements.remove(num)
        # Note that if the number is not in the popped elements, no action is taken

# Example of how to use the SmallestInfiniteSet class
# obj = SmallestInfiniteSet() # Instantiate the class
# param_1 = obj.popSmallest() # Pop the smallest element available
# obj.addBack(num) # Add back a specific number into the set of available numbers

```

Java Solution

```java
import java.util.Set;
import java.util.HashSet;

class SmallestInfiniteSet {
    // A HashSet to store numbers that have been popped.
    private Set<Integer> poppedNumbers = new HashSet<>();

    // Constructor
    public SmallestInfiniteSet() {
        // Nothing to initialize since HashSet is already initialized.
    }

    // Method to pop the smallest number that has not been popped yet.
    public int popSmallest() {
        // Starting from 1, as it's the smallest positive integer.
        int smallest = 1;
        // Loop over the set to find the smallest non-popped number (the ones not in poppedNumbers).
        while (poppedNumbers.contains(smallest)) {
            // If current number is in the set, increase it to check the next one.
            smallest++;
        }
        // Once the smallest number is found, add it to the set to indicate it has been popped.
        poppedNumbers.add(smallest);
        // Return the smallest number that hasn't been popped before.
        return smallest;
    }

    // Method to add back a number to the set of available numbers.
    public void addBack(int num) {
        // Remove the specified number from the set of popped numbers.
        poppedNumbers.remove(num);
    }
}

/**
 * The SmallestInfiniteSet object will be instantiated and called as follows:
 * SmallestInfiniteSet obj = new SmallestInfiniteSet();
 * int param_1 = obj.popSmallest();      // Pops the smallest available number
 * obj.addBack(num);                     // Adds back a number to the available set
 */

```

C++ Solution

```cpp
#include <unordered_set>
using namespace std;

// Class representing an infinite set with functionality to get the smallest element and add elements back
class SmallestInfiniteSet {
private:
    // 'removedNumbers' holds all numbers that have been popped from the set
    unordered_set<int> removedNumbers;

public:
    SmallestInfiniteSet() {
        // Constructor does not need to initialize anything for this implementation.
        // An alternative could be initializing an internal data structure if needed.
    }

    // Function to pop the smallest number from the set that hasn't been popped yet
    int popSmallest() {
        int current = 1; // We start from 1 as it is the smallest possible positive integer for the infinite set
        // Loop to find the first number that hasn't been removed yet
        while (removedNumbers.count(current)) {
            current++; // If current number is in 'removedNumbers', increment and check the next
        }
        // Once we find the smallest number not in 'removedNumbers', add it to the set to mark it as removed
        removedNumbers.insert(current);
        return current; // Return the smallest number
    }

    // Function to add back a number to the set, making it available to be popped again
    void addBack(int num) {
        // Erase the number from 'removedNumbers' to mark it as not removed
        removedNumbers.erase(num);
    }
};

// Example usage:
// SmallestInfiniteSet* obj = new SmallestInfiniteSet();
// int param_1 = obj->popSmallest();
// obj->addBack(num);

```

Typescript Solution

```typescript
// Initialize a variable to store the smallest available numbers,
// using an array to represent a set of initialized values to true.
let smallestAvailable: boolean[] = new Array(1001).fill(true);

/**
 * Retrieves and removes the smallest number available from the set.
 * @returns {number} The smallest number that was available.
 */
function popSmallest(): number {
    for (let i = 1; i <= 1001; i++) {
        if (smallestAvailable[i]) {
            smallestAvailable[i] = false; // Mark the number as unavailable.
            return i; // Return the number that's now popped out of the set.
        }
    }
    return -1;
}

/**
 * Adds a number back into the set if it's not already marked as available.
 * @param {number} num - The number to add back to the set.
 */
function addBack(num: number): void {
    if (num >= 1 && num <= 1000 && !smallestAvailable[num]) {
        smallestAvailable[num] = true; // Mark the number as available again.
    }
}

```

Time and Space Complexity

The given Python code defines a class `SmallestInfiniteSet` that maintains a set of integers from which the smallest number can be "popped" (i.e., returned and removed from the set) and to which specific numbers can be "added back" to the set if they have been previously removed.

Time Complexity

1. **`popSmallest` Method**: The time complexity for `popSmallest` is `O(n)`, where `n` is the number of consecutive integers starting from 1 that have been added to the `black` set. In the worst-case scenario, the method iterates through all elements that have been added to the set to find the smallest one that is not in the set.

2. **`addBack` Method**: The time complexity for `addBack` is `O(1)`. This is because the `discard` method of a set in Python, which is used to remove an element if it exists in the set, operates in constant time.

Space Complexity

The space complexity of the entire class is `O(m)`, where `m` is the number of unique elements that have been popped and not added back. The `black` set will grow as more unique elements are popped and retained in the set. It will not grow larger than the number of elements that have been popped and not added back, hence the space complexity of `O(m)`.