1845. Seat Reservation Manager

Medium Design Heap (Priority Queue) Leetcode Link

Problem Description

The problem involves designing a system to manage seat reservations for a series of seats that are sequentially numbered from 1 through n, where n is the total number of seats available. The system should support two key operations: reserving a seat and unreserving a seat. When a seat is reserved, it should be the seat with the smallest number currently available. Unreserving will return a previously reserved seat back into the pool of available seats.

The operations that the system needs to support are as follows:

1. **Initialization (`SeatManager(int n)`):** The constructor of the system takes an integer `n` which represents the total number of seats. It should set up the initial state with all seats available for reservation.
2. **Reserve (`int reserve()`):** This operation should return the smallest-numbered seat that is currently unreserved, and then reserve it (making it no longer available for reservation).
3. **Unreserve (`void unreserve(int seatNumber)`):** This function takes an integer `seatNumber` and unreserves that specific seat, making it available again for future reservations.

Intuition

The solution to this problem requires an efficient data structure that can constantly provide the smallest-numbered available seat and also allow reserving and unreserving seats in a relatively fast manner. A min-heap is a suitable data structure for this problem as it can always provide the minimum element in constant time and supports insertion and deletion operations in logarithmic time. In Python, we have access to a min-heap implementation through the `heapq` module.

Here's the intuition behind each part of the solution:

- **Initialization:** We initialize the state by creating a list of all available seats. We then transform this list into a heap using the `heapify` function from the `heapq` module. This sets up our min-heap, ensuring that we always have quick access to the smallest available seat number.
- **Reserve:** When reserving a seat, we pop the smallest element from the heap using the `heappop` function, which gets us the smallest-numbered seat available. This operation also removes this seat from the pool of available seats.
- **Unreserve:** To unreserve a seat, we push the seat number back into the min-heap using the `heappush` function. This means the seat is again counted among the available seats and can be reserved in a future operation.

Overall, the min-heap maintains the state of available seats, ensuring the system can efficiently handle the reservation state and seat allocations.

Solution Approach

The solution to the problem uses a priority queue data structure, which is implemented using a heap in Python with the `heapq` module. The heap is a specialized tree-based data structure that satisfies the heap property — in the case of a min-heap, the smallest element is at the root, making it easy to access it quickly.

Here's a walkthrough of the algorithm and data structures used in each method of the `SeatManager` class:

- **Initialization (`__init__` method):** We initialize our `SeatManager` with an array (list) containing all possible seat numbers. This array is denoted as `self.q` (representing a queue) and holds integers from `1` to `n` inclusive. The `heapify` function is then called on `self.q` to transform it into a heap. This operation rearranges the array into a heap structure, so it's ready to serve the smallest element, which will be at index `0`.

```python
def __init__(self, n: int):
    self.q = list(range(1, n + 1))
    heapify(self.q)
```

- **Reserve (`reserve` method):** To reserve a seat, we use the `heappop` function on our heap `self.q`. This function removes and returns the smallest element from the heap, which corresponds to the smallest-numbered available seat as per our problem statement. Since the heap property is maintained after the pop operation, the next smallest element will come to the front.

```python
def reserve(self) -> int:
    return heappop(self.q)
```

- **Unreserve (`unreserve` method):** When a seat needs to be unreserved, the `heappush` function is used. It takes the seat number and adds it back to our heap `self.q`. The `heappush` function automatically rearranges elements in the heap to ensure the heap property is maintained after adding a new element.

```python
def unreserve(self, seatNumber: int) -> None:
    heappush(self.q, seatNumber)
```

The key insights in applying a min-heap for this solution are the constant time access to the minimum element and the logarithmic time complexity for insertion and deletion operations. The way the min-heap self-adjusts after a pop or push operation ensures that the sequence of available seats is always well-organized for the needs of the `reserve` and `unreserve` methods.

Example Walkthrough

Let us consider an example where `n = 5`, which means that the `SeatManager` is initialized with 5 seats available.

The `SeatManager` is set up by calling `SeatManager(5)`.

- When initialized (`__init__`), we start with `self.q` listing seat numbers `[1, 2, 3, 4, 5]`.
- After applying `heapify(self.q)`, `self.q` becomes a min-heap but remains `[1, 2, 3, 4, 5]` since it is already in ascending order and satisfies the heap property (the smallest element is at the root).

When a client calls `reserve()` for the first time:

- The `reserve` method calls `heappop(self.q)`, which removes the root of the heap, the smallest element: seat `1`.
- The `reserve` method then returns this value, so seat `1` is now reserved.
- The `self.q` looks like `[2, 3, 4, 5]` now, with `2` being the next available seat as the root.

If we reserve again:

- Calling `reserve()` pops the root, which is now seat `2`.
- The method returns `2`, making that seat reserved.
- The `self.q` is now `[3, 4, 5]` with `3` at the root.

Now, let's assume a client wants to unreserve seat `2`. They call `unreserve(2)`:

- The `unreserve` method calls `heappush(self.q, 2)`, which adds seat `2` back to the heap.
- The heap structure is maintained, and `self.q` automatically readjusts to `[2, 3, 4, 5]` as it becomes the root again.

Next, if another reserve request comes:

- Calling `reserve()` now would pop `2` from the heap again (even though we put it back earlier).
- The `reserve` method returns `2` once more, reserving it again.
- The `self.q` state is `[3, 4, 5]` with seat `3` ready to be the next one reserved.

This example demonstrates how a min-heap structure is essential for efficiently managing the smallest seat number reservation and is perfectly suited to the requirements of the problem described. The ordering of the heap ensures that the smallest number is always at the root and can be reserved or unreserved in a consistent and predictable manner.

Python Solution

```python
import heapq  # Importing heapq for heap operations

class SeatManager:
    def __init__(self, n: int):
        # Initializing a list of seat numbers starting from 1 to n
        self.available_seats = list(range(1, n + 1))
        # Heapifying the list to create a min-heap
        # In a min-heap, the smallest element is always at the root
        heapq.heapify(self.available_seats)

    def reserve(self) -> int:
        # Reserving the smallest available seat number
        # (i.e., popping the root element from the min-heap)
        return heapq.heappop(self.available_seats)

    def unreserve(self, seat_number: int) -> None:
        # Releasing a previously reserved seat by adding it back into the heap
        # and then rearranging the heap
        heapq.heappush(self.available_seats, seat_number)
        # The heappush function automatically maintains the heap property

# An example on how to use the SeatManager class
# obj = SeatManager(n)
# seat_number = obj.reserve()
# obj.unreserve(seat_number)

```

Java Solution

```java
import java.util.PriorityQueue;

// SeatManager manages the reservation and releasing of seats
public class SeatManager {

    // PriorityQueue to store available seat numbers in ascending order
    private PriorityQueue<Integer> availableSeats;

    // Constructor initializes the PriorityQueue with all seat numbers
    public SeatManager(int n) {
        availableSeats = new PriorityQueue<>();
        // Add all seats to the queue, seat numbers start from 1 to n
        for (int seatNumber = 1; seatNumber <= n; seatNumber++) {
            availableSeats.offer(seatNumber);
        }
    }

    // reserve() method to reserve the seat with the lowest number
    public int reserve() {
        // Polling gets and removes the smallest available seat number
        return availableSeats.poll();
    }

    // unreserve() method to put a seat number back into the queue
    public void unreserve(int seatNumber) {
        // Offering adds the seat number back to the available seats
        availableSeats.offer(seatNumber);
    }
}

// Example of how the SeatManager might be used:
// SeatManager manager = new SeatManager(n);
// int seatNumber = manager.reserve(); // Reserves the lowest available seat number
// manager.unreserve(seatNumber); // Unreserves a seat, making it available again

```

C++ Solution

```cpp
#include <queue>
#include <vector>

class SeatManager {
public:
    // Constructor that initializes the seat manager with a given number of seats.
    SeatManager(int n) {
        // Add all seats to the priority queue in ascending order.
        for (int i = 1; i <= n; ++i) {
            seats.push(i);
        }
    }

    // Reserves the lowest-numbered seat that is available and returns the seat number.
    int reserve() {
        // Get the smallest available seat number from the priority queue.
        int allocatedSeat = seats.top();
        // Remove the seat from the priority queue as it is now reserved.
        seats.pop();
        // Return the reserved seat number.
        return allocatedSeat;
    }

    // Unreserves a previously reserved seat so it can be used again in the future.
    void unreserve(int seatNumber) {
        // Add the seat back to the priority queue as it is now available.
        seats.push(seatNumber);
    }

private:
    // Priority queue to manage the available seats. Seats are sorted in ascending order.
    std::priority_queue<int, std::vector<int>, std::greater<int>> seats;
};

/**
 * This code snippet shows how to create an instance of the SeatManager and
 * use its reserve and unreserve methods.
 *
 * // Initialization
 * SeatManager* seatManager = new SeatManager(n);
 *
 * // Reserve a seat
 * int reservedSeatNumber = seatManager->reserve();
 *
 * // Unreserve a specific seat
 * seatManager->unreserve(seatNumber);
 *
 * // Remember to free allocated memory if it's no longer needed, to avoid memory leaks
 * delete seatManager;
 */

```

Typescript Solution

```typescript
// Initialize variables for seat management.
let seatsArray: number[] = [];

// Function that initializes the seat manager with a given number of seats.
function initializeSeatManager(n: number): void {
    // Ensure the seats array is empty before initialisation.
    seatsArray = [];
    // Add all seats to the array in ascending order.
    for (let i = 1; i <= n; ++i) {
        seatsArray.push(i);
    }
    // Sort the array to maintain the priority queue behavior.
    // This is to ensure the smallest number is always at the start of the array.
    seatsArray.sort((a, b) => a - b);
}

// Function that reserves the lowest-numbered seat that is available and returns the seat number.
function reserve(): number {
    // Shift the first element from the array and return it,
    // representing the reservation of the lowest available seat.
    let allocatedSeat = seatsArray.shift();
    // Return the reserved seat number.
    return allocatedSeat ?? -1; // Returns -1 if no seat is available (in a case where the array is empty).
}

// Function to unreserve a previously reserved seat so it can be used again in the future.
function unreserve(seatNumber: number): void {
    // Add the seat number back to the array.
    seatsArray.push(seatNumber);
    // Sort the array to re-establish priority queue order.
    seatsArray.sort((a, b) => a - b);
}

// Usage example:
// Initialize the seat manager with a number of seats.
initializeSeatManager(10);

// Reserve a seat.
let reservedSeatNumber = reserve();

// Unreserve a specific seat.
unreserve(reservedSeatNumber);

// Note: In a real implementation, care should be taken to handle errors
// or exceptional conditions, such as trying to unreserve a seat that has not been reserved.

```

Time and Space Complexity

Time Complexity

- **init method**: O(n) - Constructing the heap of size `n` takes `O(n)` time.
- **reserve method**: O(log n) - Popping the smallest element from the heap takes `O(log n)` time, where `n` is the number of unreserved seats.
- **unreserve method**: O(log n) - Pushing an element into the heap takes `O(log n)` time.

Space Complexity

- The space complexity for the entire `SeatManager` is O(n) where `n` is the number of seats. This accounts for the heap that stores the seat numbers.