729. My Calendar I

Medium Design Segment Tree Binary Search Ordered Set

Problem Description

In this problem, you are tasked with creating a program that acts as a calendar. The primary function of this calendar is to ensure that when a new event is added, it doesn't clash with any existing events (a situation referred to as a "double booking"). An event is defined by a start time and an end time, which are represented by a pair of integers. These times create a half-open interval [start, end), meaning it includes the start time up to, but not including, the end time.

MyCalendar() is a constructor that initializes the calendar object.
 book(int start, int end) is a method that adds an event to the calendar if it does not conflict with any existing events. If the event can be

Your job is to implement a MyCalendar class that will hold the events and has the following capabilities:

- added without causing a double booking, it returns true; otherwise, it returns false and does not add the event.
- The goal is to ensure that no two events overlap in time.

The key to solving this problem is to efficiently determine whether the newly requested booking overlaps with any existing

bookings. One way to approach this is by maintaining a sorted list of events, allowing for quick searches and insertions.

That the new event's start time does not conflict with the end time of the previous event.

The intuition is to search for the correct location to insert a new event such that the list remains sorted. We must check two

That the new event's end time does not conflict with the start time of the next event.
 We can accomplish this by:

- We can accomplish this by:

 1. Using the sortedcontainers. SortedDict class, which keeps keys in a sorted order. This allows us to quickly find the position where the new
- event could be inserted.

 2. Applying the bisect_right method to find the index of the smallest event end that is greater than the new event's start time.

3. Checking if the new event's end time conflicts with the next event's start time in the sorted dictionary.

4. If there is no conflict, we insert the new event into the "sorted" dictionary, with the end time as the key and the start time as the value.

adjacent events in the sorted list of bookings.

The implementation of the book method in our solution proceeds with this intuition, allowing for efficient booking operations. Each

5. By keeping the dictionary sorted by the end time, this ensures that we can always quickly check for potential overlap with the immediately

booking can be processed in logarithmic time with respect to the number of existing bookings, therefore making the solution

stored in the self.sd attribute of the class instances, ready to keep track of the booked events.

- scalable for a large number of events.

 Solution Approach
- The implementation of MyCalendar relies on the sortedcontainers Python library, which offers a SortedDict data structure to maintain the events sorted by their end times. Here's a walkthrough of how the solution is implemented using this SortedDict:

 1. Initialization: When the MyCalendar class is instantiated, it initializes a SortedDict in the constructor. This SortedDict is

self.sd = SortedDict() 2. **Booking an Event:** The book method is where the logic to check for double bookings and add events takes place.

def __init__(self):

the start time.
idx = self.sd.bisect_right(start)

Now, we need to ensure that the new event does not conflict with the next event in the SortedDict. We check if the found index is within the bounds of the SortedDict and if the new event's end time is greater than the start time of the event at that index.
if idx < len(self.sd) and end > self.sd.values()[idx]:

First, we find the index (position) where the new event's end time would be inserted into the SortedDict. We use the

bisect_right method, which returns an index pointing to the first element in the SortedDict's values that is greater than

times.
self.sd[end] = start

If at any point we detect an overlap (a potential double booking), we return False without adding the event.

are efficient due to the nature of the SortedDict, which maintains the order of keys and allows for binary search insertions and

If there is no conflict, it means the new event does not cause a double booking, and we insert it into the dictionary. Here,

the event's end time is used as the key and the start time as the value. This ensures the events are sorted by their end

return True

The book method performs at most two key operations: finding where to insert and actually inserting the event. Both operations

return False

self.sd[end] = start

When we try to book the first event:

Now, MyCalendar looks like this:

• self.sd contains {20: 10}

def book(self, start: int, end: int) -> bool:

if idx < len(self.sd) and end > self.sd.values()[idx]:

idx = self.sd.bisect_right(start)

return False

Example Walkthrough

lookups. This is how the provided solution ensures no double bookings occur while adding events to the MyCalendar.

After successfully adding the event without conflicts, the method returns True.

- Imagine we have a MyCalendar instance and we want to book two events. The first event is from time 10 to 20, and the second event is from time 15 to 25.
- We initialize our MyCalendar and its underlying SortedDict, currently empty.
 We attempt to book an event with start=10 and end=20.
 self.sd.bisect_right(10) will return 0 since there are no keys greater than 10 (as the dictionary is empty).
 Since the index 0 is within bounds and there are no events, there are no conflicts.
 We add the event to the SortedDict with key 20 and value 10.

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

than 10, there is a potential overlap with the existing event.

4. Since end time 25 of the new event is greater than start time 10 of the existing event, which would result in a double booking, we return False.

Solution Implementation

Python

class MyCalendar:

When we try to book the second event:

Thus, the attempt to book an event from time 15 to 25 fails, preserving the non-overlapping constraint of the calendar. The SortedDict remains unchanged with the single event {20: 10} and no double booking occurs.

def book(self, start: int, end: int) -> bool:

otherwise returns False.

1. We attempt to book the second event with start=15 and end=25.

from sortedcontainers import SortedDict

2. self.sd.bisect_right(15) will return index 0 since 20 is the first key greater than 15.

def __init__(self):
 # Create a sorted dictionary to store the end time of each event
 # as the key and the start time as the value
 self.sorted_events = SortedDict()

3. We check if idx is within bounds and if end (25) is greater than the start time of the event at index 0 (which is 10). Since 25 is indeed greater

:param start: The start time of the event :param end: The end time of the event :return: Boolean indicating whether the event was successfully booked

Attempts to book an event in the calendar based on the provided start and end times.

check if the requested end time is greater than the start time of the next event.

Returns True if the event can be booked (doesn't overlap with existing events);

Find the index of the first event that ends after the requested start time

- If next_event_index is within the bounds of the sorted dictionary,

next_event_index = self.sorted_events.bisect_right(start)

Check if there is a conflict with the next event:

Example of how to instantiate the MyCalendar class and book events:

return True # Event booked successfully

if next_event_index < len(self.sorted_events) and end > self.sorted_events.values()[next_event_index]: return False # Event cannot be booked due to overlap # If there is no conflict, insert the new event into the sorted dictionary. self.sorted_events[end] = start

calendar = MyCalendar()

print(is_booked)

Java

is_booked = calendar.book(start, end)

```
import java.util.Map;
import java.util.TreeMap;
// The class MyCalendar is designed to store bookings as intervals.
// It uses a TreeMap to keep the intervals sorted by start time.
class MyCalendar {
    // Using TreeMap to maintain the intervals sorted by the start key.
    private final TreeMap<Integer, Integer> calendar;
    // Constructor initializes the TreeMap.
    public MyCalendar() {
        calendar = new TreeMap<>();
    /**
     * Tries to book an interval from start to end.
     * @param start the starting time of the interval
     * @param end the ending time of the interval
     * @return true if the booking does not conflict with existing bookings, false otherwise
     */
    public boolean book(int start, int end) {
        // Retrieves the maximum entry whose key is less than or equal to start.
        Map.Entry<Integer, Integer> floorEntry = calendar.floorEntry(start);
        // If there is an overlap with the previous interval, return false.
        if (floorEntry != null && floorEntry.getValue() > start) {
            return false;
        // Retrieves the minimum entry whose key is greater than or equal to start.
        Map.Entry<Integer, Integer> ceilingEntry = calendar.ceilingEntry(start);
        // If there is an overlap with the next interval, return false.
        if (ceilingEntry != null && ceilingEntry.getKey() < end) {</pre>
            return false;
        // If there is no overlap, add the interval to the TreeMap and return true.
        calendar.put(start, end);
        return true;
// Usage example:
```

```
// Increment the count for the start time of the new event
events[start]++;
// Decrement the count for the end time of the new event
events[end]--;
int currentEvents = 0; // counter for overlapping events
```

C++

private:

public:

#include <map>

using namespace std;

MyCalendar() {

map<int, int> events;

class MyCalendar {

// MyCalendar obj = new MyCalendar();

// boolean isBooked = obj.book(start, end);

otherwise returns false. */

for (auto& keyValue : events) {

if (currentEvents > 1) {

events[start]--;

events[end]++;

return false;

bool book(int start, int end) {

// map to keep track of event starts (+1) and ends (-1)

/* Function to book a new event from 'start' to 'end' time.

// Iterate through all time points in the map

currentEvents += keyValue.second;

// Add up the values to check for overlaps

Returns true if the event can be booked without conflicts,

// Constructor initializes the MyCalendar object

```
// No overlap found, event is successfully booked
        return true;
};
// Example usage:
// MyCalendar* calendar = new MyCalendar();
// bool canBook = calendar->book(start,end);
TypeScript
// A global array to keep track of booked time slots as an array of start-end pairs.
let calendar: number[][] = [];
/**
 * Attempts to book a new event in the calendar.
 * @param {number} start The start time of the event.
 * @param {number} end The end time of the event.
 * @return {boolean} True if the event can be successfully booked without conflicts; otherwise, false.
 */
function book(start: number, end: number): boolean {
    // Iterate through each already booked event in the calendar.
    for (const item of calendar) {
        // If the new event overlaps with an existing event, return false.
        if (end > item[0] && start < item[1]) {</pre>
            return false;
    // If there is no overlap, add the new event to the calendar and return true.
```

// If more than one event is happening at the same time, revert changes and return false

```
Attempts to book an event in the calendar based on the provided start and end times. Returns True if the event can be booked (doesn't overlap with existing events); otherwise returns False.

:param start: The start time of the event
:param end: The end time of the event
:return: Boolean indicating whether the event was successfully booked
```

self.sorted_events[end] = start

is_booked = calendar.book(start, end)

SortedDict if there is no overlap.

return True # Event booked successfully

self.sorted_events = SortedDict()

def book(self, start: int, end: int) -> bool:

as the key and the start time as the value

calendar.push([start, end]);

from sortedcontainers import SortedDict

return true;

def __init__(self):

calendar = MyCalendar()

print(is_booked)

Time Complexity

class MyCalendar:

```
Time and Space Complexity

The provided code implements a class MyCalendar that stores the start time of the events as values and the end time as keys in a SortedDict. When a new event is booked, it checks if there is any overlap with existing events and then stores the event in the
```

Create a sorted dictionary to store the end time of each event

next_event_index = self.sorted_events.bisect_right(start)

return False # Event cannot be booked due to overlap

Check if there is a conflict with the next event:

Example of how to instantiate the MyCalendar class and book events:

Find the index of the first event that ends after the requested start time

If there is no conflict, insert the new event into the sorted dictionary.

check if the requested end time is greater than the start time of the next event.

if next_event_index < len(self.sorted_events) and end > self.sorted_events.values()[next_event_index]:

- If next_event_index is within the bounds of the sorted dictionary,

__init__: The constructor simply creates a new SortedDict, which is an operation taking 0(1) time.
 book: This method involves two main actions. First, it performs a binary search to find the right position where the new event should be inserted. The bisect_right method in SortedDict runs in 0(log n) time where n is the number of keys in the

dictionary. Secondly, the code inserts the new (end, start) pair into the dictionary. Inserting into a SortedDict also takes

O(log n) time. Therefore, the overall time complexity for each book operation is O(log n). Space Complexity

With no events booked, the space complexity is 0(1) as only an empty SortedDict is maintained.
 As events are added, the space complexity grows linearly with the number of non-overlapping events stored. Therefore, in the worst-case

The space complexity of the code is mainly dictated by the storage requirements of the SortedDict.

- As events are added, the space complexity grows linearly with the number of non-overlapping events stored. Therefore, in the worst-case scenario, where the calendar has n non-overlapping events, the space complexity would be 0(n).

 Overall, the space complexity of the MyCalendar data structure is 0(n) where n is the number of non-overlapping events booked
- in the calendar.