The Number Of The Smallest Unoccupied Chair

solved by	Senan
Platform	LeetCode
difficulty	Medium
# Serial	1942
tags	Heaps Simulation
language	C++
solved on	@11/10/2024
link	https://leetcode.com/problems/the-number-of-the-smallest-unoccupied-chair/
Completion	☑

Intuition

The problem involves assigning chairs to friends based on their arrival and leaving times, with the goal of finding which chair will be assigned to a specific friend (`targetFriend`). We maintain two priority queues: one to track currently occupied chairs and their release times, and another to track available chairs in the smallest-first order. As friends arrive, we free up chairs that are no longer occupied and assign the smallest available chair to the new arrival. The solution proceeds in a greedy fashion by always assigning the smallest possible chair.

Approach

1. Parse the input `times`, where each element represents a friend's arrival and departure time.
2. Append the index of each friend to the input times to keep track of their identities.
3. Sort the times by arrival time to process friends in chronological order.
4. Use two priority queues:
    - One for managing currently occupied chairs, sorted by when they will be freed.
    - Another for available chairs, sorted by chair number.
5. For each friend, first free up any chairs from friends who have already left (before the current friend's arrival).
6. Assign the smallest available chair to the current friend.
7. If the current friend is the target friend, return the chair number assigned to them.

Complexity

Time Complexity:

- Sorting the `times` array takes $O(n \log n)$, where n is the number of friends.
- For each friend, we potentially pop from both priority queues, which takes $O(n \log n)$ for each operation.
- Thus, the overall time complexity is $O(n \log n)$.

Space Complexity:

- We use two priority queues, both of which will hold up to n elements.
- Thus, the space complexity is $\mathrm{O}(n)$.

Code

```
class Solution {
public:
    int smallestChair(vector<vector<int>>& times, int targetFriend) {
        int n = times.size();

        for (int i = 0; i < n; i++) times[i].push_back(i);
        sort(times.begin(), times.end());

        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>
> occupied;
        priority_queue<int, vector<int>, greater<int>> available;

        for (int i = 0; i < n; i++) available.push(i);

        for (int i = 0; i < n; i++) {
            int arrivalTime = times[i][0];
            int leavingTime = times[i][1];

            while (!occupied.empty() && occupied.top().second <= arrivalTime) {
                available.push(occupied.top().first);
                occupied.pop();
            }

            int assignedChair = available.top();
            available.pop();

            if (times[i][2] == targetFriend)
                return assignedChair;

            occupied.push({assignedChair, leavingTime});
        }
        return -1;
    }
};
```