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 can be modeled using two priority queues: one to keep track of currently occupied chairs and their leave times, and another to keep track of available (free) chairs.

1. **Sorting by Arrival Time**: First, we sort the times based on the arrival time of friends. Each friend gets assigned a chair when they arrive.
2. **Handling Chair Assignments**: As friends arrive, we check if there are any free chairs from friends who have already left. If so, we assign the smallest available chair. If no chair is free, the friend gets a new chair.
3. **Target Friend**: If the current friend is the target friend, we return the index of the chair they get assigned, whether it's a newly assigned one or one that was just freed.

Complexity

Time Complexity:

1. **Sorting**: Sorting the times takes O(n log n), where n is the number of friends.
2. **Priority Queues**: For each friend's arrival and departure:
   - Pushing and popping from the `occupied` and `available` priority queues takes O(log n).

Thus, the overall time complexity is:

- **O(n log n)** for sorting,
- **O(n log n)** for the operations on priority queues.

Therefore, the total time complexity is **O(n log n)**.

Space Complexity:

1. **Priority Queues**: We use two priority queues (`occupied` and `available`), each holding at most n elements.

Thus, the space complexity is **O(n)** for the priority queues.

Code

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

        sort(times.begin(), times.end());

        int newChair = 0;
        for (int i = 0; i < n; ++i) {
            int arrival = times[i][0];
            int leaveTime = times[i][1];
            int friendIndex = times[i][2];

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

            if (available.empty()) {
                if (friendIndex == targetFriend) return newChair;
                occupied.push({leaveTime, newChair});
                newChair++;
            } else {
                int chairIndex = available.top();
                available.pop();
                if (friendIndex == targetFriend) return chairIndex;
                occupied.push({leaveTime, chairIndex});
            }
        }
        return -1;
    }
};
```