Problem 1500: Design a File Sharing System

Problem Information

Difficulty: Medium
Acceptance Rate: 41.89%
Paid Only: Yes
Tags: Hash Table, Design, Sorting, Heap (Priority Queue), Data Stream

Problem Description

We will use a file-sharing system to share a very large file which consists of `m` small **chunks** with IDs from `1` to `m`.

When users join the system, the system should assign **a unique** ID to them. The unique ID should be used **once** for each user, but when a user leaves the system, the ID can be **reused** again.

Users can request a certain chunk of the file, the system should return a list of IDs of all the users who own this chunk. If the user receives a non-empty list of IDs, they receive the requested chunk successfully.

Implement the `FileSharing` class:

* `FileSharing(int m)` Initializes the object with a file of `m` chunks. * `int join(int[] ownedChunks)`: A new user joined the system owning some chunks of the file, the system should assign an id to the user which is the **smallest positive integer** not taken by any other user. Return the assigned id. * `void leave(int userID)`: The user with `userID` will leave the system, you cannot take file chunks from them anymore. * `int[] request(int userID, int chunkID)`: The user `userID` requested the file chunk with `chunkID`. Return a list of the IDs of all users that own this chunk sorted in ascending order.

Example:

Input: ["FileSharing","join","join","join","request","request","leave","request","leave","join"] [[4],[[1,2]],[[2,3]],[[4]],[1,3],[2,2],[1],[2,1],[2],[[]]] **Output:** [null,1,2,3,[2],[1,2],null,[],null,1] **Explanation:** FileSharing fileSharing = new FileSharing(4); // We use the system to share a file of 4 chunks. fileSharing.join([1, 2]); // A user who has chunks [1,2] joined the system, assign id = 1 to them and return 1. fileSharing.join([2, 3]); // A user who has chunks [2,3] joined the system, assign id = 2 to them and return 2. fileSharing.join([4]); // A user who has chunk [4] joined the system, assign id = 3 to them and return 3. fileSharing.request(1, 3); // The user with id = 1 requested the third file chunk, as only the user with id = 2 has the file, return [2] . Notice that user 1 now has chunks [1,2,3]. fileSharing.request(2, 2); // The user with id = 2 requested the second file chunk, users with ids [1,2] have this chunk, thus we return [1,2]. fileSharing.leave(1); // The user with id = 1 left the system, all the file chunks with them are no longer available for other users. fileSharing.request(2, 1); // The user with id = 2 requested the first file chunk, no one in the system has this chunk, we return empty list []. fileSharing.leave(2); // The user with id = 2 left the system. fileSharing.join([]); // A user who doesn't have any chunks joined the system, assign id = 1 to them and return 1. Notice that ids 1 and 2 are free and we can reuse them.

Constraints:

* `1 <= m <= 105` * `0 <= ownedChunks.length <= min(100, m)` * `1 <= ownedChunks[i] <= m` * Values of `ownedChunks` are unique. * `1 <= chunkID <= m` * `userID` is guaranteed to be a user in the system if you **assign** the IDs **correctly**. * At most `104` calls will be made to `join`, `leave` and `request`. * Each call to `leave` will have a matching call for `join`.

Follow-up:

* What happens if the system identifies the user by their IP address instead of their unique ID and users disconnect and connect from the system with the same IP? * If the users in the system join and leave the system frequently without requesting any chunks, will your solution still be efficient? * If all users join the system one time, request all files, and then leave, will your solution still be efficient? * If the system will be used to share `n` files where the `ith` file consists of `m[i]`, what are the changes you have to make?

Code Snippets

C++:

```
class FileSharing {
public:
FileSharing(int m) {

}
```

```
int join(vector<int> ownedChunks) {

}

void leave(int userID) {

}

vector<int> request(int userID, int chunkID) {

}
};

/**
* Your FileSharing object will be instantiated and called as such:
* FileSharing* obj = new FileSharing(m);
* int param_1 = obj->join(ownedChunks);
* obj->leave(userID);
* vector<int> param_3 = obj->request(userID,chunkID);
*/
```

Java:

```
class FileSharing {

public FileSharing(int m) {

}

public int join(List<Integer> ownedChunks) {

}

public void leave(int userID) {

}

public List<Integer> request(int userID, int chunkID) {

}
}
```

```
/**
* Your FileSharing object will be instantiated and called as such:
* FileSharing obj = new FileSharing(m);
* int param_1 = obj.join(ownedChunks);
* obj.leave(userID);
* List<Integer> param_3 = obj.request(userID,chunkID);
*/
```

Python3:

```
class FileSharing:

def __init__(self, m: int):



def join(self, ownedChunks: List[int]) -> int:



def leave(self, userID: int) -> None:



def request(self, userID: int, chunkID: int) -> List[int]:




# Your FileSharing object will be instantiated and called as such:
# obj = FileSharing(m)
# param_1 = obj.join(ownedChunks)
# obj.leave(userID)
# param_3 = obj.request(userID,chunkID)
```