Data Structure Design...

video-24

Leetcode - 1912

Hard

∞ Instagram	codestorywithmik
X	CSwithMIK
WhatsApp	codestorywithMIK

WeekendWithMIK

WeekendWithMIK

@WeekendWithMIK [redacted]

Welcome to WeekendWithMIK 🎥✨ >

[redacted]

LIFE BEHIND THE SCENES...

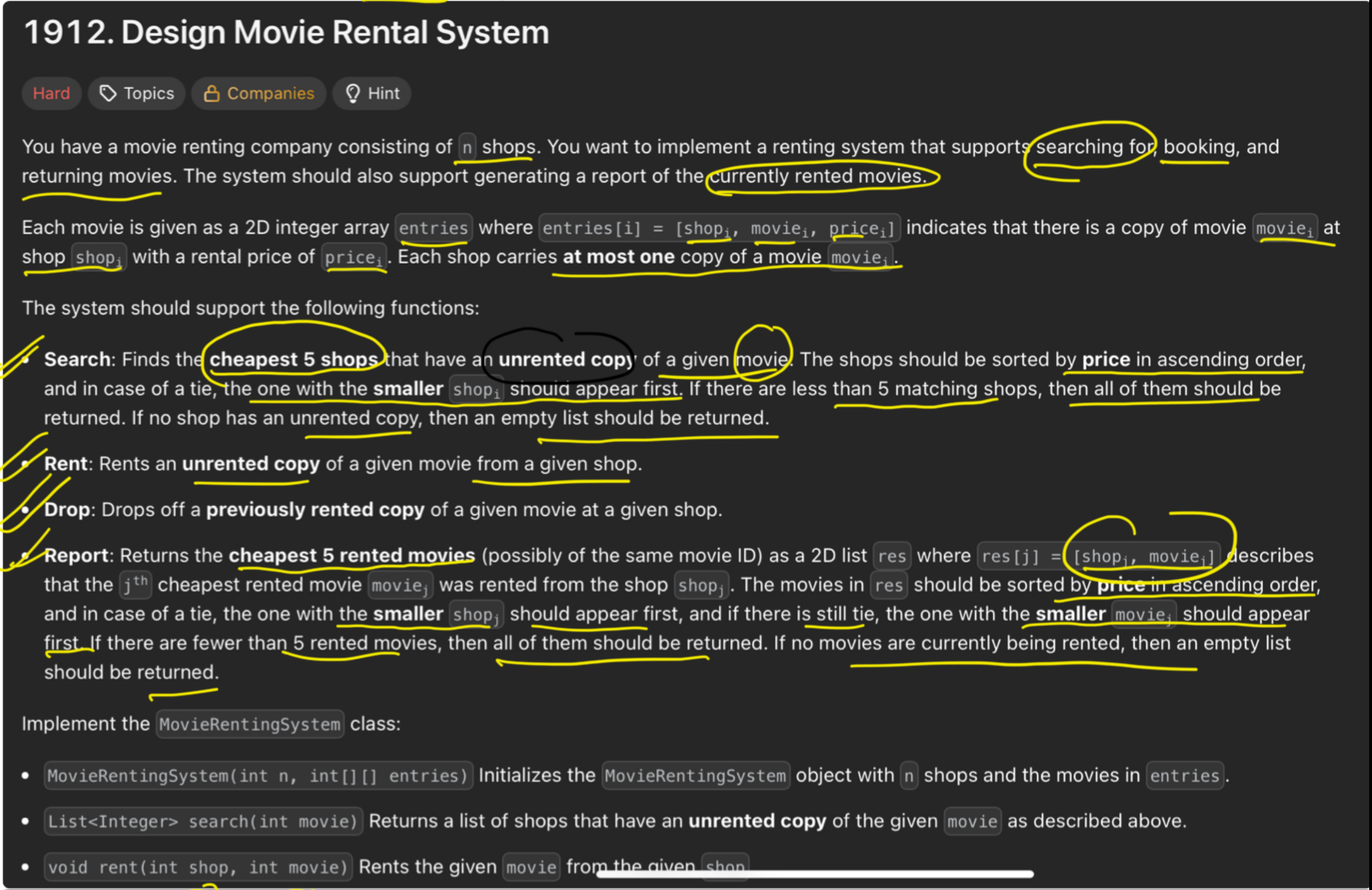
News

+ Tech

Motivation :-

There will be times when you will think nothing is working, you can't do it, you loose confidence. Know that even a successful person also went through such phase. Things take time, provided you don't quit..

MIK...



1912. Design Movie Rental System

Hard · Topics · Companies · Hint

You have a movie renting company consisting of `n` shops. You want to implement a renting system that supports searching for, booking, and returning movies. The system should also support generating a report of the currently rented movies.

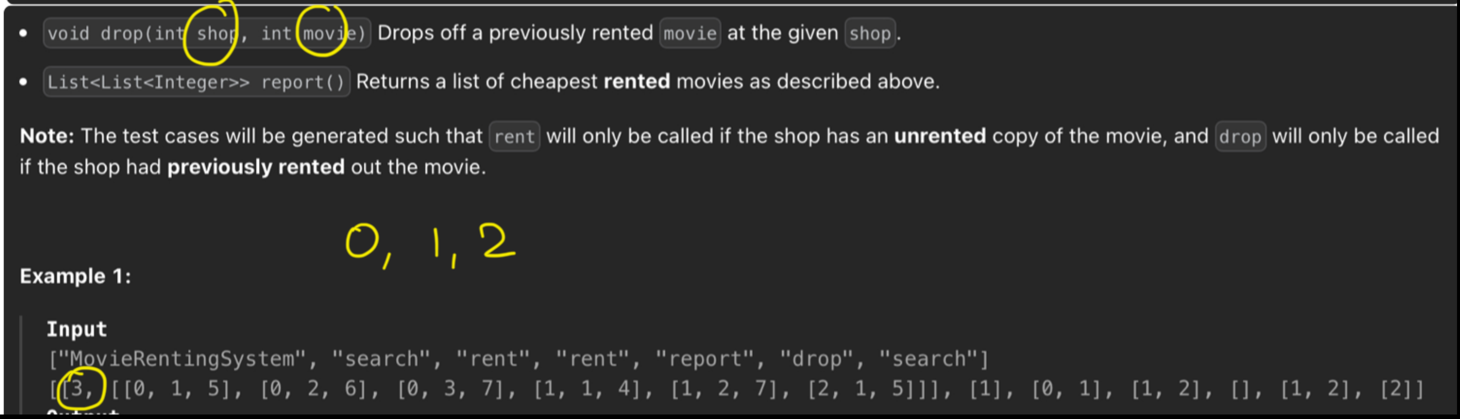
Each movie is given as a 2D integer array `entries` where `entries[i]` = [shop$_i$, movie$_i$, price$_i$] indicates that there is a copy of movie movie$_i$ at shop shop$_i$ with a rental price of price$_i$. Each shop carries **at most one** copy of a movie movie$_i$.

The system should support the following functions:

- **Search**: Finds the **cheapest 5 shops** that have an **unrented copy** of a given movie. The shops should be sorted by **price** in ascending order, and in case of a tie, the one with the **smaller** shop$_i$ should appear first. If there are less than 5 matching shops, then all of them should be returned. If no shop has an unrented copy, then an empty list should be returned.
- **Rent**: Rents an **unrented copy** of a given movie from a given shop.
- **Drop**: Drops off a **previously rented copy** of a given movie at a given shop.
- **Report**: Returns the **cheapest 5 rented movies** (possibly of the same movie ID) as a 2D list `res` where `res[j]` = [shop$_j$, movie$_j$] describes that the j^{th} cheapest rented movie movie$_j$ was rented from the shop shop$_j$. The movies in `res` should be sorted by **price** in ascending order, and in case of a tie, the one with the **smaller** shop$_j$ should appear first, and if there is still tie, the one with the **smaller** movie$_j$ should appear first. If there are fewer than 5 rented movies, then all of them should be returned. If no movies are currently being rented, then an empty list should be returned.

Implement the `MovieRentingSystem` class:

- `MovieRentingSystem(int n, int[][] entries)` Initializes the `MovieRentingSystem` object with `n` shops and the movies in `entries`.
- `List<Integer> search(int movie)` Returns a list of shops that have an **unrented copy** of the given `movie` as described above.
- `void rent(int shop, int movie)` Rents the given `movie` from the given `shop`



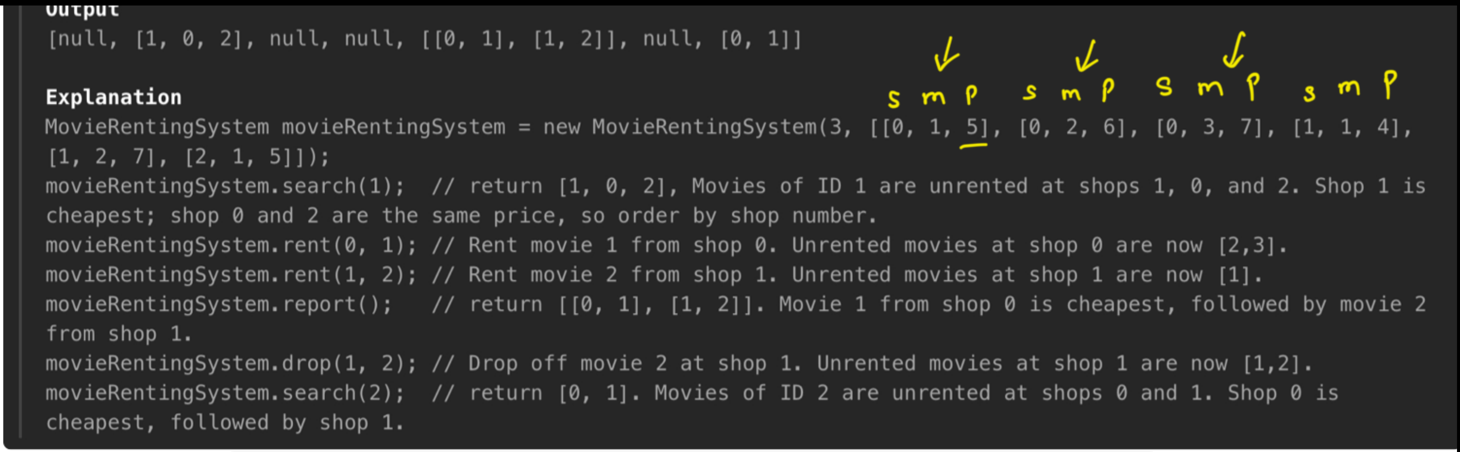
- `void drop(int shop, int movie)` Drops off a previously rented `movie` at the given `shop`.
- `List<List<Integer>> report()` Returns a list of cheapest **rented** movies as described above.

Note: The test cases will be generated such that `rent` will only be called if the shop has an **unrented** copy of the movie, and `drop` will only be called if the shop had **previously rented** out the movie.

0, 1, 2

Example 1:

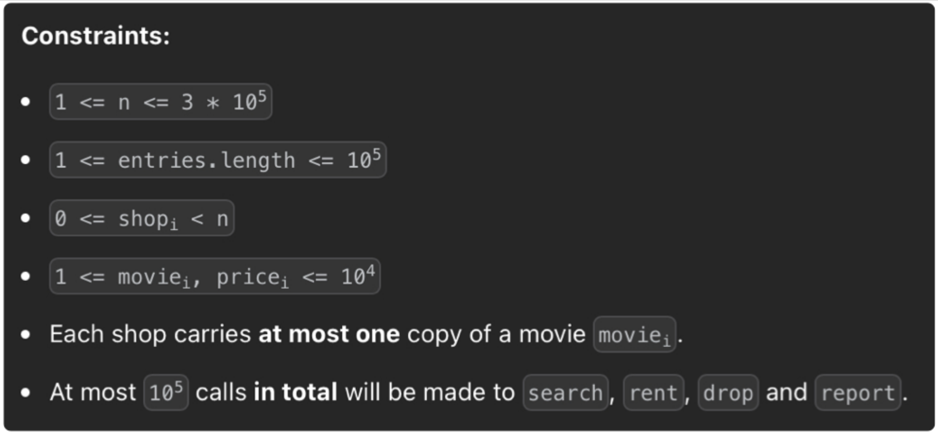
```
Input
["MovieRentingSystem", "search", "rent", "rent", "report", "drop", "search"]
[[3, [[0, 1, 5], [0, 2, 6], [0, 3, 7], [1, 1, 4], [1, 2, 7], [2, 1, 5]]], [1], [0, 1], [1, 2], [], [1, 2], [2]]
```



s m p s m p s m p s m p

```
Output
[null, [1, 0, 2], null, null, [[0, 1], [1, 2]], null, [0, 1]]

Explanation
MovieRentingSystem movieRentingSystem = new MovieRentingSystem(3, [[0, 1, 5], [0, 2, 6], [0, 3, 7], [1, 1, 4],
[1, 2, 7], [2, 1, 5]]);
movieRentingSystem.search(1);  // return [1, 0, 2], Movies of ID 1 are unrented at shops 1, 0, and 2. Shop 1 is
cheapest; shop 0 and 2 are the same price, so order by shop number.
movieRentingSystem.rent(0, 1); // Rent movie 1 from shop 0. Unrented movies at shop 0 are now [2,3].
movieRentingSystem.rent(1, 2); // Rent movie 2 from shop 1. Unrented movies at shop 1 are now [1].
movieRentingSystem.report();   // return [[0, 1], [1, 2]]. Movie 1 from shop 0 is cheapest, followed by movie 2
from shop 1.
movieRentingSystem.drop(1, 2); // Drop off movie 2 at shop 1. Unrented movies at shop 1 are now [1,2].
movieRentingSystem.search(2);  // return [0, 1]. Movies of ID 2 are unrented at shops 0 and 1. Shop 0 is
cheapest, followed by shop 1.
```



Constraints:

- `1 <= n <= 3 * 10^5`
- `1 <= entries.length <= 10^5`
- `0 <= shop_i < n`
- `1 <= movie_i, price_i <= 10^4`
- Each shop carries **at most one** copy of a movie `movie_i`.
- At most `10^5` calls **in total** will be made to `search`, `rent`, `drop` and `report`.

Thought Process

→ Search (movie)

Cheapest 5 shops having this movie as unrented

sorted by price → shop

movie → {(price1, shop1) (price2, shop2)}

ascending order

```
unordered_map
< movie ,   set < pair<int,int>>> available;
                       ↑    ↑
                       p    s
```

✓ MovieRentingSystem (int n, entries) {

```
        // Parse entries
              [shop , movie, price]

        available[movie].insert({price, shop});

}
```

✓ Search (movie) {

```
        result;
        count = 0;
        if (available.count(movie)) {

                for ( auto &[price, shop] : available[movie]){

                            result.push_back (shop);
                             count++;
                             if(count == 5) {
                                  break;
                             }
                }
        }
}     return result;
```

report()

" Cheapest 5 rented movies {shop, movie}
sorted by price → shop → movie "

Data structure

↓

ordr Set :- { (Price, shop, movie),
(Price, shope, m2), }

set<tuple<int, int, int>> rented;

```
// vect<vect<int>> report ( ) {

          result;
          int count =0;
          for (auto & [price, shop, movie] : rented){

                result.push_back ({shop, movie});
                Count ++;
                 if (cu >= 5)
                        brk;
          }

     retn result;
}
```

rent(movie , shop)

"rent an unrented copy of movie from the shop"

available : movie → set { (price, shop), (Price , shop). }

↑ ↑ ↑ ↑

O(n) search finding the shop :.

movie → set { (Shop, price) (shop price) .. }

↓

search a give shop in log time using Binary Search.

```
set.lower_bound( shop);
```

Custom Binary Search

>= Shop

movie ↑   shop ↑   price ↑

```
unordered_map< int , set<pair<int,int>> ; movietoShop;
```

```
void rent ( shop , movie ) {

auto it = movietoShop[movie].lower_bound( {shop, INT_MIN})
```

>= Shop

```
price = it→second;

// remove it from available map.

available [movie].erase({price, shop});
```

```
}          rented.insert ({price, shop,movie})


void drop ( shop,  movie )  {
```