

Assigned Seating

Objective

Give practice with structs and dynamic memory in C.
Give practice with functions in C.

Story

The demand for movies is higher than ever, and you are going to use this to your advantage to make some decent cash. You need your theater to stand out from its competitors, so you are going to introduce your Unacceptably Luminescent Tech Igniting Many Airplanes Though Eco-friendly (ULTIMATE) projector.

The projector in question can be seen for miles, and it works in the sunlight too. You are going to set up a large theater that can fit many people. For now you need some software that can track who is in what seat efficiently. It seems the software you got from buying out a theater could only handle up to 400-seat theaters. You have *much* larger ambitions.

Problem

Write a program that can allow for people to buy consecutive seats in a row (if available) and look up who owns the seat at a particular spot in a row. You should assume that the theater starts with every seat available for purchase.

Input

Each line of input will be in one of the following formats

- BUY <ROW> <START> <END> <NAME>
 - This means the user with the given <NAME> will try to purchase the seats in the given <ROW>. They want the seats with numbers ranging from <START> to <END> inclusive.
 - You are guaranteed that <START> will be at most <END>.
 - If any other user owns at least one seat in this range, the purchase should **NOT** go through.
- LOOKUP <ROW> <SEAT>
 - This means that we need to determine which user, if any, purchased the seat with the number <SEAT> in the given <ROW>.
- QUIT
 - This should terminate the program and clean up any memory.

<ROW>, <START>, <END>, and <SEAT> will all be positive integers in the range of 1 to 100,000, inclusive. The <NAME> will be an alphabetic string of at most 50 characters.

Output

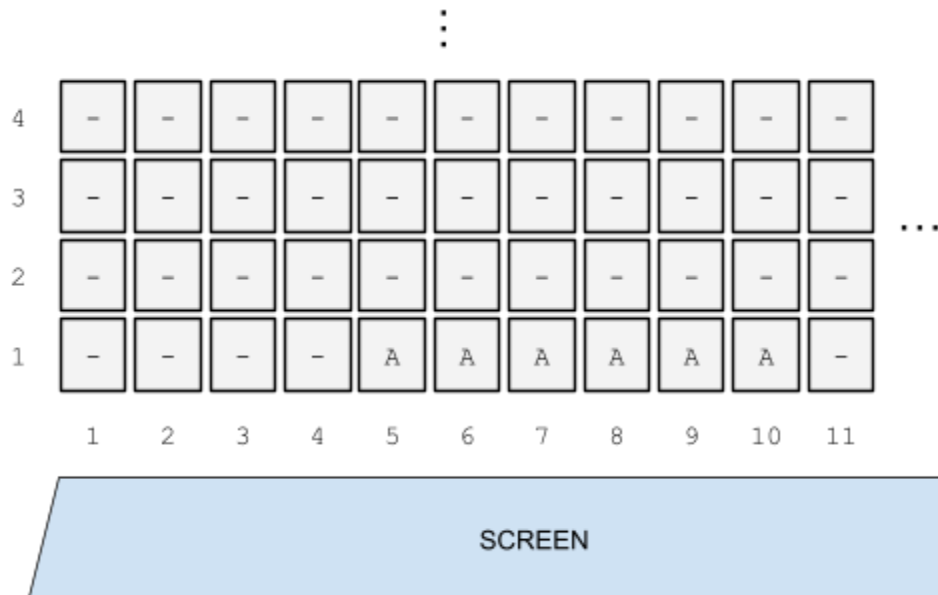
For each BUY input, output a single line containing the word "SUCCESS" if the purchase was successful, and "FAILURE" otherwise. For each LOOKUP input, output a single line containing solely the name of the user that owns the seat, if the seat is owned and "No one", if the seat is unowned.

Sample Input	Sample Output
BUY 1 5 10 ALICE BUY 1 7 7 Bob BUY 2 5 10 Carol LOOKUP 1 7 LOOKUP 1 10 LOOKUP 1 1 BUY 1 1 4 David LOOKUP 1 1 BUY 1 2 5 Eric BUY 10 1000 1000 SAM QUIT	SUCCESS FAILURE SUCCESS ALICE ALICE No one SUCCESS David FAILURE SUCCESS
LOOKUP 1 1 LOOKUP 1 2 BUY 3 49998 50003 Guha BUY 1 50000 50000 Meade BUY 5 49999 50001 Ahmed LOOKUP 1 1 LOOKUP 1 49999 LOOKUP 50000 1 LOOKUP 3 50000 QUIT	No one No one SUCCESS SUCCESS SUCCESS No one No one No one Guha

Explanation

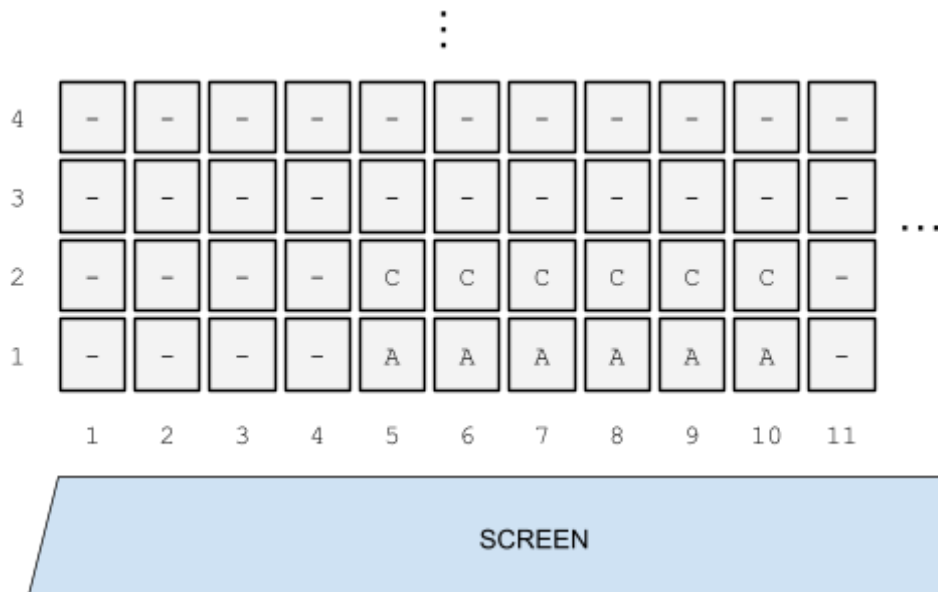
Case 1

In the first line ALICE (all caps) buys 6 seats in the first row (marked with A in the figure below)



Next Bob tries to buy seat 7 in the first row, but cannot, because ALICE already bought it.

The Carol Tries to buy the same 6 seats as ALICE, but in row 2. This is possible because none have been bought yet.



The next line of input looks up the person who owns seat 7 in row 1. This is ALICE.


The next line of input looks up the person who owns seat 10 in row 1. This is also ALICE.

The next line of input ("LOOKUP 1 1") looks up the person who owns seat 1 in row 1. This is not ALICE. No one actually owns this seat.

In the next line David tries to buy the seats 1 through 4 in the first row. He does so successfully. The current theater is as follows.

⋮

4	-	-	-	-	-	-	-	-	-	-	-	
3	-	-	-	-	-	-	-	-	-	-	-	...
2	-	-	-	-	C	C	C	C	C	C	-	
1	D	D	D	D	A	A	A	A	A	A	-	
	1	2	3	4	5	6	7	8	9	10	11	



SCREEN

We then look up who owns the first seat of the first row again. This time David owns it.

Eric then tries to buy the seats from 2 to 5 in row 1. Both David and ALICE have some of those seats, so Eric fails his purchase.

Lastly, SAM buys the 1000th seat in the 10th row. This is successful. Since no one is remotely close to that location.

After this is the QUIT command which exits the program.

Case 2

The first 2 lines of input try to look up values in a theater that has had no seats purchased.

The third line of input is when Guha purchases 6 seats from 49,998 to 50,003 in the third row.

Meade on the fourth line of input purchases a single seat at position 50,000 in the first row.

Ahmed on the fifth line of input purchases 3 seats from 49,999 to 50,001 in the fifth row.

All purchases were successful since they were in different rows.

There is a lookup in the first row in the first seat, but no one purchased a seat there.

The next lookup is in the first row at position 49,999. Meade is the only person in that row, but his reservation starts at 50,000, so no one is in spot 49,999.

The next lookup is in row 50,000 at seat 1. There is no one in that row.

The next and last lookup is requesting the owner of the 50,000 seat on the third row. This is Guha!

The following line is QUIT and exits the program.

Hints

Dynamic Memory: You need to use dynamic memory.

Required Structs: For full credit you are required to use the following structs

```
struct Reservation {
    int start, end;    // Start and end of the reservation
    char * name;       // Name of the person who reserved this range
};

struct Row {
    Struct Reservation * array;    // The array of reservations
    int max_reservations;          // the spots in the array
    int num_reservations;          // occupied spots in the array
};

struct Theater {
    struct Row * row;    // The array of rows in the theater
};
```

Varying Token Count: You should use the first token of the line to determine how many additional tokens can be read. When you read a token using something like scanf, unless your format specifier indicates to read to the end of the line the scanf will stop after reading the given token. That is scanf can read part way through the line and leave the remainder of the line for processing by a future scanf call.

strcmp: You can use the strcmp to check if a string is the same as another string.

AVOID 2D GRID: Do NOT store a 2D to represent the theater. That would take up too much memory. Our business is not doing well enough to constitute filling every seat in the theater, especially since there are more seats than human beings on the planet.

Instead the Row will be sparsely allocated. In the worst case the Row will be an array with a number of spots equal to around twice the actual number of reservations.

Do Not Split Reservations: Keep the reservation as one struct. Do not make a separate piece of memory for each chair in a reservation.

Output: You should print out your output as your program generates it. Do not try to hold onto your output until the end. That much memory will probably burst your limit on heap memory in the large cases.

I BETTER NOT FIND THIS ON THE INTERNET (OUTSIDE OF WEBCOURSES)!!!

Grading Criteria

- Good comments, whitespace, and variable names
 - 15 points
- No extra input output (e.g. input prompts, "Please enter the original sign's message:")
 - 5 points
- Standard IO
 - 5 points
- Use a Theater, Row, and Reservation struct as described in the hints section
 - 15 points total [5 points each]
- Implement a initializeRow function
 - 5 points
- Implement a makePurchase function that returns an int representing a success or failure
 - 5 points
- Programs will be tested on 10 cases
 - 5 points each

No points will be awarded to programs that do not compile using "gcc -std=gnu11 -lm".

*Sometimes a requested technique will be given, and solutions without the requested technique will have their maximum points total reduced. For this problem use dynamic memory. **Without this programs will earn at most 50 points!***

Any case that causes a program to return a non-zero return code will be treated as wrong. Additionally, any case that takes longer than the maximum allowed time (the max of {5 times my solutions time, 10 seconds}) will also be treated as wrong.

No partial credit will be awarded for an incorrect case.