Problem: Assigned Seating

Objective

Give practice with structs and dynamic memory in C.

Give practice with functions in C.

Give practice following a design specification with structs and function prototype requirements.

Background 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 Description

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

The input format consists of a sequence of actions and requests, specified in the order in which they occur. Instead of having a fixed number of lines, the input will be sentinel driven; this means that a special word on a line by itself will specify that there is no more input to process.

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 single word string of at most 50 characters *(use scanf with %s)*. Finally, there will never be more than 100 successful BUY operations executed on a single row.

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

Sample Explanation

In the first sample, ALICE's buy is successful but Bob's isn't because he tries to buy seat 7 in row 1, which ALICE already has bought. Carol's buy is also successful in row 2.

ALICE does have seats 7 and 10 on the first row (because she bought seats 5 through 10 on that row), but does not have seat one on row 1.

David follows this up by buying all the seats with lower numbers on row 1 next to Alice's block of seats. Right after this buy, David has seat 1 in row 1. Eric can't buy the seats he wants in row 1 because they are already bought by both David and ALICE. Finally, SAM can buy seats in row 10 because they are all open to buy.

In the second sample, we start with 2 look ups, which can not be successful because no one has seats. Then, all three instructors buy seats in different rows. The next three look ups are all for seats that are unowned, while the last look up is for Guha's seat.

Required Structs and Function Prototypes

Here are three #defines for important constants to use:

```
#define INITSIZE 10

#define MAXLEN 50

#define MAXROWS 100000
```

The first represents the initial size of any row (in terms of number of orders).

The second represents the maximum length of a name for an order.

The third represents the maximum number of rows in the theater.

For safety reasons (or if you want to use 1-based indexing), feel free to add 1 to the last two.

The following struct definition should be used for an order:

```
typedef struct order {
    int s_seat;
    int e_seat;
    char* name;
} order;
```

The reason the row isn't stored in the order is that each row number will implicitly be equal to the index in the array storing the rows for the movie theater.

A second required struct will manage a single row. A vast majority of the logic of the assignment will be done in the functions associated with this struct:

```
typedef struct theaterrow {
    order** list_orders;
    int max_size;
    int cur_size;
} theaterrow;
```

Here are functions you **<u>are required to write</u>**. What is given is both the function signature and a description of what the function does:

```
// Returns a pointer to a dynamically allocated order storing the given
// start row, end row, and the name this_name. Note: strcpy should be
// used to copy the contents into the struct after its name field is
// dynamically allocated.
order* make_order(int start, int end, char* this_name);

// This function will allocate the memory for one theaterrow, including
// allocating its array of orders to a default maximum size of 10, and
// setting its current size to 0.
theaterrow* make_empty_row();

// Assuming that order1 and order2 point to orders on the same row, this
// function returns 1 if the orders conflict, meaning that they share // at
least 1 seat in common, and returns 0 otherwise.
int conflict(order* order1, order* order2)

// Returns 1 if the order pointed to by this_order can be added to the
// row pointed to by this_row. Namely, 1 is returned if and only if
// this_order does NOT have any seats in it that are part of any order
// already on this_row.
int can_add_order(theaterrow* this_row, order* this_order);
```

```
// This function tries to add this_order to this_row. If  successful,
// the order is added and 1 is returned. Otherwise, 0 is returned and
// no change is made to this_row. If the memory for this_row is full,
// this function will double the maximum # of orders allocated for the
// row (via realloc), before adding this_order, and adjust max_size and
// cur_size of this_row.
void add_order(theaterrow* this_row, order* this_order);

// If seat_num seat number in row number row is occupied, return a
// pointer to the owner's name. Otherwise, return NULL.
char* get_owner(theaterrow** theater, int row, int seat_num);

// If seat_num in the row pointed to by this_row is occupied, return a
// pointer to the string storing the owner's name. If no one is sitting
// in this seat, return NULL.
char* get_row_owner(theaterrow* this_row, int seat_num);

// This function returns 1 if the seat number seat_no is contained in
// the range of seats pointed to by myorder, and returns 0 otherwise.
int contains(order* myorder, int seat_no);

// Frees all memory associated with this_order.
void free_order(order* this_order);

// Frees all memory associated with this_row.
void free_row(theaterrow* this_row);
```

Finally, in the main function, the whole theater will be declared as follows (you can choose any variable name you want):

```
theaterrow** amc_= calloc(MAXROWS+1, sizeof(theaterrow*));
```

This allocates an array, where each element in the array is a **pointer to a theater row**. The memory associated with this variable should be freed, in main, at the very end of the program.