Scenario: You are developing a car rental system for a rental agency. The system needs to manage customers who are waiting to rent or return cars. Customers are managed using a queue (implemented with a linked list), where they wait in line to either rent or return a car. The cars are stored in a stack (implemented with an array) that handles the available cars for rent. Each car has a unique ID. When a customer rents a car, the car is popped from the stack. When a customer returns a car, it is pushed back onto the stack.

Task Definition

The task involves managing a car rental system where customers can either rent or return cars. This system uses two primary data structures:

1. **A queue implemented using a linked list** to handle customer actions (renting or returning a car).
2. **A stack implemented using a single-dimensional array** to keep track of available car IDs.

Main

Create a Queue for Customer Actions

- **Objective:** Initialize a new queue to manage customer actions. This queue will be used to store actions such as renting or returning cars.

```
Queue customerQueue = createQueue();
```

Initialize the Car Stack

- **Objective:** Set up a stack to manage available car IDs. This stack will track which cars are available for rent and which ones are currently returned.\

```
CarStack carStack;

initializeCarStack(&carStack);
```

Add Cars to the Stack

- **Objective:** Populate the stack with car IDs that represent cars available for rent. This step simulates having a set of cars ready for customers.

  ```
  pushCar(&carStack, "Ca123");

  pushCar(&carStack, "Ca456");

  pushCar(&carStack, "Ca789");

  pushCar(&carStack, "Ca439");

  pushCar(&carStack, "Ca956");

  pushCar(&carStack, "Ca829");
  ```

Enqueue Customer Actions

- **Objective:** Record customer requests to rent or return cars. Each request specifies the action and the customer's name.

  ```
  enqueue(customerQueue, "pop", "Alice");

  enqueue(customerQueue, "pop", "Charlie");

  enqueue(customerQueue, "push", "Bob");

  enqueue(customerQueue, "push", "Andy");

  enqueue(customerQueue, "pop", "Robert");
  ```

Process Customer Actions

- **Objective:** Handle each customer request by processing actions in the order they were received. Determine whether the request is to rent or return a car and perform the corresponding stack operation.
- **Action:**
  - Continuously check if the queue is empty.
  - Dequeue actions from the front of the queue.
  - Based on the action type, call the appropriate stack function:
    - If the action is to rent a car, pop a car ID from the stack and handle the result.
    - If the action is to return a car, push a car ID onto the stack and handle the result.
  - After processing each action, free the memory allocated for the node to prevent memory leaks.

Functions

Queue Functions

`Queue createQueue()`

Description: Initializes a new queue by allocating memory and setting the `front` and `rear` pointers to `NULL`.

`void enqueue(Queue q, char *action, char *customerName)`

Description: Adds a new action (rent or return) to the end of the queue. It creates a new node, sets its `action` and `customerName`, and updates the `rear` pointer.

`struct Node* dequeue(Queue q)`

Description: Removes and returns the action node from the front of the queue. Updates the `front` pointer to the next node. If the queue becomes empty, the `rear` pointer is also set to `NULL`.

`int isQueueEmpty(Queue q)`

Description: Checks if the queue is empty by verifying if the `front` pointer is `NULL`.

Stack Functions

`void initializeCarStack(CarStack* s)`

Description: Initializes the stack by setting the `top` index to `-1`, indicating that the stack is empty.

`int pushCar(CarStack* s, const char *carId)`

Description: Adds a car ID to the top of the stack. It checks if the stack is full before adding the ID and updates the `top` index.

`char* popCar(CarStack* s)`

Description: Removes and returns the car ID from the top of the stack. It checks if the stack is empty before removing the ID and updates the `top` index.

`int isCarStackEmpty(CarStack* s)`

Description: Checks if the stack is empty by verifying if the `top` index is `-1`.

`int isCarStackFull(CarStack* s)`

Description: Checks if the stack is full by comparing the `top` index with `MAX_CARS - 1`.

Structures

1. Node Structure for the Queue

```
// Node structure for the queue (linked list)
struct Node {
    char action[5];          // Action to perform: "push" (return car) or "pop" (rent car)
    char customerName[100];  // Customer's name
    struct Node* next;       // Pointer to the next node in the queue
};
```

2. Stack Structure for Car IDs

```
// Stack structure implemented using a single-dimensional array
typedef struct {
    char carIds[MAX_CARS * CAR_ID_LENGTH]; // Array to hold car IDs contiguously
    int top;                // Index of the top element in the stack
} CarStack;
```

3. Queue Structure

```
// Queue structure with pointers to the front and rear
typedef struct {
    struct Node *front;     // Pointer to the front of the queue
    struct Node *rear;      // Pointer to the rear of the queue
} *Queue;
```

Example Array:

c	**a**	**1**	**1**	**2**		**c**	**a**	**1**	**……**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**……**