SET OF PROGRAMS
==========
1. **Stock Market Order Matching System**: Implement a queue using arrays to simulate a stock market's order matching system. Design a program where buy and sell orders are placed in a queue. The system should match and process orders based on price and time priority.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ORDERS 100

// Define the Order structure
typedef struct {
    char type[4];      // "buy" or "sell"
    float price;       // Price of the stock
    int quantity;      // Quantity of the stock
    int timestamp;     // Time of order, used for time priority
} Order;

// Define the Queue structure
typedef struct {
    Order orders[MAX_ORDERS];
    int front;
    int rear;
} Queue;

// Function to initialize a queue
void initQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return q->front == -1;
}

// Function to check if the queue is full
int isFull(Queue *q) {
    return q->rear == MAX_ORDERS - 1;
}

// Function to add an order to the queue
void enqueue(Queue *q, Order order) {
    if (isFull(q)) {
        printf("Queue is full, cannot add new order.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear++;
    q->orders[q->rear] = order;
```

```
}

// Function to remove an order from the queue
Order dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty, cannot dequeue.\n");
        exit(1);
    }
    Order order = q->orders[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
    return order;
}

// Function to process buy and sell orders based on price and time priority
void matchOrders(Queue *buyQueue, Queue *sellQueue) {
    while (!isEmpty(buyQueue) && !isEmpty(sellQueue)) {
        Order buyOrder = buyQueue->orders[buyQueue->front];
        Order sellOrder = sellQueue->orders[sellQueue->front];

        // Check if the buy price is greater than or equal to the sell price
        if (buyOrder.price >= sellOrder.price) {
            printf("Matching Buy Order: %d shares at %.2f with Sell Order: %d shares at %.2f\n",
                   buyOrder.quantity, buyOrder.price, sellOrder.quantity, sellOrder.price);

            // Process the orders (reduce quantity or remove completely)
            if (buyOrder.quantity > sellOrder.quantity) {
                buyOrder.quantity -= sellOrder.quantity;
                enqueue(buyQueue, buyOrder);
                dequeue(sellQueue); // Sell order completely matched
            } else if (buyOrder.quantity < sellOrder.quantity) {
                sellOrder.quantity -= buyOrder.quantity;
                enqueue(sellQueue, sellOrder);
                dequeue(buyQueue); // Buy order completely matched
            } else {
                dequeue(buyQueue); // Both orders completely matched
                dequeue(sellQueue);
            }
        } else {
            break; // No more matches possible (buy price too low)
        }
    }
}

int main() {
    Queue buyQueue, sellQueue;
    initQueue(&buyQueue);
    initQueue(&sellQueue);

    // Example orders (timestamp is just for demonstration)
    Order buyOrder1 = {"buy", 100.50, 10, 1};
    Order sellOrder1 = {"sell", 99.00, 5, 2};
    Order buyOrder2 = {"buy", 101.00, 20, 3};
```

```
    Order sellOrder2 = {"sell", 101.00, 15, 4};

    // Enqueue buy and sell orders
    enqueue(&buyQueue, buyOrder1);
    enqueue(&sellQueue, sellOrder1);
    enqueue(&buyQueue, buyOrder2);
    enqueue(&sellQueue, sellOrder2);

    // Match orders
    matchOrders(&buyQueue, &sellQueue);

    return 0;
}
```

2. **Customer Service Center Simulation**: Use a linked list to implement a queue for a customer service center. Each customer has a priority level based on their membership status, and the program should handle priority-based queueing and dynamic customer arrival.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to represent a customer
struct Customer {
    char name[100];
    int priority;
    struct Customer* next;
};

// Function to create a new customer node
struct Customer* createCustomer(char* name, int priority) {
    struct Customer* newCustomer = (struct Customer*)malloc(sizeof(struct Customer));
    strcpy(newCustomer->name, name);
    newCustomer->priority = priority;
    newCustomer->next = NULL;
    return newCustomer;
}

// Function to insert a customer into the priority queue based on priority
void insertCustomer(struct Customer** front, struct Customer** rear, char* name, int priority) {
    struct Customer* newCustomer = createCustomer(name, priority);

    if (*front == NULL || (*front)->priority < priority) {
        newCustomer->next = *front;
        *front = newCustomer;
    } else {
        struct Customer* temp = *front;
        while (temp->next != NULL && temp->next->priority >= priority) {
            temp = temp->next;
        }
        newCustomer->next = temp->next;
        temp->next = newCustomer;
    }

    if (*rear == NULL) {
```

```
        *rear = newCustomer;
    }
}

// Function to serve a customer (dequeue operation)
void serveCustomer(struct Customer** front, struct Customer** rear) {
    if (*front == NULL) {
        printf("No customers to serve.\n");
        return;
    }

    struct Customer* temp = *front;
    printf("Serving customer: %s (Priority: %d)\n", temp->name, temp->priority);
    *front = (*front)->next;

    if (*front == NULL) {
        *rear = NULL;
    }

    free(temp);
}

// Function to display the customers in the queue
void displayQueue(struct Customer* front) {
    if (front == NULL) {
        printf("No customers in the queue.\n");
        return;
    }

    printf("Customers in the queue (from highest to lowest priority):\n");
    struct Customer* temp = front;
    while (temp != NULL) {
        printf("Name: %s, Priority: %d\n", temp->name, temp->priority);
        temp = temp->next;
    }
}

int main() {
    struct Customer* front = NULL;
    struct Customer* rear = NULL;
    int choice;
    char name[100];
    int priority;

    while (1) {
        printf("\nCustomer Service Center Menu:\n");
        printf("1. Add customer to queue\n");
        printf("2. Serve customer\n");
        printf("3. Display queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar();  // To consume the newline character after entering the choice

        switch (choice) {
```

```
            case 1:
                printf("Enter customer's name: ");
                fgets(name, sizeof(name), stdin);
                name[strcspn(name, "\n")] = 0;  // Remove the newline character
                printf("Enter customer's priority (higher number means higher priority): ");
                scanf("%d", &priority);
                insertCustomer(&front, &rear, name, priority);
                break;
            case 2:
                serveCustomer(&front, &rear);
                break;
            case 3:
                displayQueue(front);
                break;
            case 4:
                printf("Exiting the program...\n");
                exit(0);
            default:
                printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```

3. **Political Campaign Event Management**: Implement a queue using arrays to manage attendees at a political campaign event. The system should handle registration, check-in, and priority access for VIP attendees.

```
#include <stdio.h>
#include <string.h>

#define MAX_ATTENDEES 100

// Define the structure for an attendee
typedef struct {
    char name[50];
    int isVIP;  // 1 for VIP, 0 for regular attendee
} Attendee;

// Queue structure
typedef struct {
    Attendee attendees[MAX_ATTENDEES];
    int front;
    int rear;
} Queue;

// Function prototypes
void initQueue(Queue* q);
int isEmpty(Queue* q);
int isFull(Queue* q);
void enqueue(Queue* q, Attendee a);
Attendee dequeue(Queue* q);
void displayQueue(Queue* q);
void registerAttendee(Queue* q, char* name, int isVIP);
```

```
void checkInAttendee(Queue* q);

int main() {
    Queue q;
    initQueue(&q);

    // Register some attendees
    registerAttendee(&q, "John Doe", 0); // Regular attendee
    registerAttendee(&q, "Jane Smith", 1); // VIP attendee
    registerAttendee(&q, "Alice Johnson", 0); // Regular attendee

    // Display the queue before check-in
    printf("Queue before check-in:\n");
    displayQueue(&q);

    // Check-in attendees (VIPs will have priority)
    checkInAttendee(&q);
    checkInAttendee(&q);
    checkInAttendee(&q);

    return 0;
}

// Initialize the queue
void initQueue(Queue* q) {
    q->front = 0;
    q->rear = 0;
}

// Check if the queue is empty
int isEmpty(Queue* q) {
    return q->front == q->rear;
}

// Check if the queue is full
int isFull(Queue* q) {
    return (q->rear + 1) % MAX_ATTENDEES == q->front;
}

// Enqueue an attendee
void enqueue(Queue* q, Attendee a) {
    if (isFull(q)) {
        printf("Queue is full! Cannot register more attendees.\n");
        return;
    }
    q->attendees[q->rear] = a;
    q->rear = (q->rear + 1) % MAX_ATTENDEES;
}

// Dequeue an attendee
Attendee dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! No attendees to check-in.\n");
        Attendee emptyAttendee = {"", 0};
        return emptyAttendee;
```

```
    }
    Attendee a = q->attendees[q->front];
    q->front = (q->front + 1) % MAX_ATTENDEES;
    return a;
}

// Display the queue
void displayQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Attendees in the queue:\n");
    int i = q->front;
    while (i != q->rear) {
        printf("%s (VIP: %d)\n", q->attendees[i].name, q->attendees[i].isVIP);
        i = (i + 1) % MAX_ATTENDEES;
    }
}

// Register an attendee (VIP or regular)
void registerAttendee(Queue* q, char* name, int isVIP) {
    Attendee a;
    strncpy(a.name, name, sizeof(a.name) - 1);
    a.isVIP = isVIP;

    // If VIP, enqueue at the front (priority)
    if (isVIP) {
        // Move regular attendees forward in the queue to make space for VIP
        for (int i = q->rear; i != q->front; i = (i - 1 + MAX_ATTENDEES) % MAX_ATTENDEES) {
            q->attendees[i] = q->attendees[(i - 1 + MAX_ATTENDEES) % MAX_ATTENDEES];
        }
        q->attendees[q->front] = a;
        q->rear = (q->rear + 1) % MAX_ATTENDEES;
    } else {
        enqueue(q, a); // For regular attendee, just enqueue
    }
}

// Check-in an attendee (dequeue operation)
void checkInAttendee(Queue* q) {
    if (isEmpty(q)) {
        printf("No attendees to check-in.\n");
        return;
    }
    Attendee a = dequeue(q);
    printf("Checked in: %s (VIP: %d)\n", a.name, a.isVIP);
}
```

4. **Bank Teller Simulation**: Develop a program using a linked list to simulate a queue at a bank. Customers arrive at random intervals, and each teller can handle one customer at a time. The program should simulate multiple tellers and different transaction times.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>

#define MAX_TELLERS 3 // Number of tellers
#define MAX_CUSTOMERS 10 // Maximum number of customers to simulate

// Customer structure (linked list node)
typedef struct Customer {
    int id;               // Unique ID for the customer
    int transaction_time;  // Time to process the customer
    struct Customer* next; // Pointer to the next customer in the queue
} Customer;

// Teller structure
typedef struct Teller {
    int id;                // Unique ID for the teller
    int time_remaining;      // Time remaining for the current customer to be served
} Teller;

// Queue structure (for customer queue)
typedef struct Queue {
    Customer* front; // Front of the queue
    Customer* rear;  // Rear of the queue
} Queue;

// Function declarations
void initQueue(Queue* q);
int isQueueEmpty(Queue* q);
void enqueue(Queue* q, int customer_id, int transaction_time);
Customer* dequeue(Queue* q);
void initTellers(Teller tellers[], int num_tellers);
void simulateBankQueue(Queue* queue, Teller tellers[], int num_tellers);
void printQueue(Queue* q);

// Main function
int main() {
    srand(time(NULL)); // Seed the random number generator

    Queue queue;
    Teller tellers[MAX_TELLERS];

    // Initialize the queue and tellers
    initQueue(&queue);
    initTellers(tellers, MAX_TELLERS);

    // Simulate customers arriving at random intervals
    for (int i = 0; i < MAX_CUSTOMERS; i++) {
        int transaction_time = rand() % 10 + 1; // Random transaction time between 1 and 10
        enqueue(&queue, i + 1, transaction_time); // Enqueue a new customer
        printf("Customer %d arrived with transaction time %d\n", i + 1, transaction_time);
        printQueue(&queue);

        // Simulate bank queue processing
        simulateBankQueue(&queue, tellers, MAX_TELLERS);

        // Sleep for a short time before the next customer arrives
```

```
        sleep(1);
    }

    return 0;
}

// Function to initialize the queue
void initQueue(Queue* q) {
    q->front = q->rear = NULL;
}

// Function to check if the queue is empty
int isQueueEmpty(Queue* q) {
    return q->front == NULL;
}

// Function to add a customer to the queue
void enqueue(Queue* q, int customer_id, int transaction_time) {
    Customer* new_customer = (Customer*)malloc(sizeof(Customer));
    new_customer->id = customer_id;
    new_customer->transaction_time = transaction_time;
    new_customer->next = NULL;

    if (isQueueEmpty(q)) {
        q->front = q->rear = new_customer;
    } else {
        q->rear->next = new_customer;
        q->rear = new_customer;
    }
}

// Function to remove a customer from the queue
Customer* dequeue(Queue* q) {
    if (isQueueEmpty(q)) {
        return NULL;
    }

    Customer* customer_to_serve = q->front;
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL;
    }

    return customer_to_serve;
}

// Function to initialize the tellers
void initTellers(Teller tellers[], int num_tellers) {
    for (int i = 0; i < num_tellers; i++) {
        tellers[i].id = i + 1;
        tellers[i].time_remaining = 0; // Initially, no customer is being served
    }
}
```

```
// Function to simulate processing by the tellers
void simulateBankQueue(Queue* queue, Teller tellers[], int num_tellers) {
    for (int i = 0; i < num_tellers; i++) {
        if (tellers[i].time_remaining == 0 && !isQueueEmpty(queue)) {
            // Teller is free, serve the next customer in the queue
            Customer* customer = dequeue(queue);
            tellers[i].time_remaining = customer->transaction_time;
            printf("Teller %d is now serving Customer %d with transaction time %d\n",
                   tellers[i].id, customer->id, customer->transaction_time);
            free(customer);
        } else if (tellers[i].time_remaining > 0) {
            // Teller is busy, decrease the remaining time
            tellers[i].time_remaining--;
            if (tellers[i].time_remaining == 0) {
                printf("Teller %d has finished serving a customer\n", tellers[i].id);
            }
        }
    }
}

// Function to print the queue
void printQueue(Queue* q) {
    if (isQueueEmpty(q)) {
        printf("The queue is empty.\n");
    } else {
        Customer* current = q->front;
        printf("Customers in queue: ");
        while (current != NULL) {
            printf("Customer %d (time: %d) -> ", current->id, current->transaction_time);
            current = current->next;
        }
        printf("END\n");
}
```

5. **Real-Time Data Feed Processing**: Implement a queue using arrays to process real-time data feeds from multiple financial instruments. The system should handle high-frequency data inputs and ensure data integrity and order.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_QUEUE_SIZE 1000  // Maximum size of the queue

// Define a structure for the financial instrument data (data feed)
typedef struct {
    char instrument_id[10];
    double price;
    int volume;
    long timestamp;  // Timestamp of the data feed
} DataFeed;

// Define a queue structure using arrays
typedef struct {
    DataFeed queue[MAX_QUEUE_SIZE];
```

```
    int front;
    int rear;
    int size;
} DataQueue;

// Function to initialize the queue
void initializeQueue(DataQueue* q) {
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}

// Function to check if the queue is full
int isQueueFull(DataQueue* q) {
    return q->size == MAX_QUEUE_SIZE;
}

// Function to check if the queue is empty
int isQueueEmpty(DataQueue* q) {
    return q->size == 0;
}

// Function to enqueue data into the queue
void enqueue(DataQueue* q, DataFeed data) {
    if (isQueueFull(q)) {
        printf("Queue is full, cannot enqueue data.\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->queue[q->rear] = data;
    q->size++;
}

// Function to dequeue data from the queue
DataFeed dequeue(DataQueue* q) {
    if (isQueueEmpty(q)) {
        printf("Queue is empty, cannot dequeue data.\n");
        exit(1);  // Exit if the queue is empty
    }
    DataFeed data = q->queue[q->front];
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    q->size--;
    return data;
}

// Function to display the queue contents
void displayQueue(DataQueue* q) {
    if (isQueueEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue contents:\n");
    int i = q->front;
    for (int j = 0; j < q->size; j++) {
        printf("Instrument: %s, Price: %.2f, Volume: %d, Timestamp: %ld\n",
```

```
            q->queue[i].instrument_id, q->queue[i].price, q->queue[i].volume, q->queue[i].timestamp);
        i = (i + 1) % MAX_QUEUE_SIZE;
    }
}

// Example of real-time data feed simulation
void processRealTimeData(DataQueue* q) {
    DataFeed newData;
    // Simulate data feed (In real scenario, data would come from an external source)
    strcpy(newData.instrument_id, "AAPL");
    newData.price = 150.75;
    newData.volume = 100;
    newData.timestamp = 1674249000;  // Example timestamp
    enqueue(q, newData);

    // More data can be processed here in real-time...
}

int main() {
    DataQueue queue;
    initializeQueue(&queue);

    // Simulate real-time data processing
    for (int i = 0; i < 5; i++) {
        processRealTimeData(&queue);
    }

    // Display the current queue state
    displayQueue(&queue);

    // Dequeue and process the data
    printf("\nDequeuing data...\n");
    while (!isQueueEmpty(&queue)) {
        DataFeed data = dequeue(&queue);
        printf("Processing: %s, Price: %.2f, Volume: %d, Timestamp: %ld\n",
               data.instrument_id, data.price, data.volume, data.timestamp);
    }

    return 0;
}
```

6. **Traffic Light Control System**: Use a linked list to implement a queue for cars at a traffic light. The system should manage cars arriving at different times and simulate the light changing from red to green.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // For sleep()

// Structure for a car in the queue
struct Car {
    int car_id;
    struct Car* next;
};

// Structure for the queue (linked list)
```

```
struct Queue {
    struct Car* front;
    struct Car* rear;
};

// Function to initialize the queue
void initQueue(struct Queue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to add a car to the queue
void enqueue(struct Queue* queue, int car_id) {
    struct Car* new_car = (struct Car*)malloc(sizeof(struct Car));
    new_car->car_id = car_id;
    new_car->next = NULL;

    if (queue->rear == NULL) {
        // If the queue is empty, both front and rear will point to the new car
        queue->front = queue->rear = new_car;
        return;
    }

    // Add the new car to the rear of the queue and update the rear pointer
    queue->rear->next = new_car;
    queue->rear = new_car;
}

// Function to remove a car from the front of the queue
int dequeue(struct Queue* queue) {
    if (queue->front == NULL) {
        printf("No cars in the queue.\n");
        return -1;
    }

    struct Car* temp = queue->front;
    int car_id = temp->car_id;

    queue->front = queue->front->next;

    // If the queue becomes empty, update the rear pointer to NULL
    if (queue->front == NULL) {
        queue->rear = NULL;
    }

    free(temp);
    return car_id;
}

// Function to simulate the traffic light system
void trafficLightControl(struct Queue* queue) {
    int light_timer = 5; // Time in seconds the light stays green
    int car_id;

    // Simulating traffic light changes
```

```
    while (1) {
        // Green light: Let cars pass through
        printf("\nGreen light: Cars can pass.\n");
        while (light_timer > 0 && queue->front != NULL) {
            car_id = dequeue(queue);
            printf("Car %d passed.\n", car_id);
            sleep(1);  // Simulate car passing (1 second per car)
            light_timer--;
        }

        if (queue->front == NULL) {
            printf("\nNo more cars waiting. The queue is empty.\n");
            break;
        }

        // Red light: Stop cars from passing
        printf("\nRed light: No cars are allowed to pass.\n");
        sleep(3);  // Simulate the red light duration (3 seconds)

        // Reset the light timer for the next green light cycle
        light_timer = 5;
    }
}

int main() {
    struct Queue queue;
    initQueue(&queue);

    // Simulating cars arriving at different times
    enqueue(&queue, 1);
    enqueue(&queue, 2);
    enqueue(&queue, 3);
    enqueue(&queue, 4);
    enqueue(&queue, 5);

    // Run the traffic light control system
    trafficLightControl(&queue);

    return 0;
}
```

7. **Election Vote Counting System**: Implement a queue using arrays to manage the vote counting process during an election. The system should handle multiple polling stations and ensure votes are counted in the order received.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_VOTES 100  // Maximum number of votes that can be held in the queue
#define MAX_CANDIDATES 5  // Number of candidates in the election

// Structure for the queue
typedef struct {
    int votes[MAX_VOTES];  // Array to store votes
```

```
    int front;  // Front index
    int rear;   // Rear index
    int size;   // Current size of the queue
} Queue;

// Function to initialize the queue
void initQueue(Queue *q) {
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return q->size == 0;
}

// Function to check if the queue is full
int isFull(Queue *q) {
    return q->size == MAX_VOTES;
}

// Function to enqueue a vote
void enqueue(Queue *q, int vote) {
    if (isFull(q)) {
        printf("Queue is full, cannot accept more votes.\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX_VOTES;  // Circular increment
    q->votes[q->rear] = vote;
    q->size++;
}

// Function to dequeue a vote and process it
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty, no votes to process.\n");
        return -1;  // Indicates no vote to process
    }
    int vote = q->votes[q->front];
    q->front = (q->front + 1) % MAX_VOTES;  // Circular increment
    q->size--;
    return vote;
}

// Function to count votes for each candidate
void countVotes(Queue *q) {
    int candidates[MAX_CANDIDATES] = {0};  // Initialize candidate votes to 0

    while (!isEmpty(q)) {
        int vote = dequeue(q);  // Get the next vote
        if (vote >= 0 && vote < MAX_CANDIDATES) {
            candidates[vote]++;  // Increment vote count for the candidate
        } else {
            printf("Invalid vote received: %d\n", vote);
```

```
        }
    }

    // Display vote counts for each candidate
    printf("Vote counts:\n");
    for (int i = 0; i < MAX_CANDIDATES; i++) {
        printf("Candidate %d: %d votes\n", i, candidates[i]);
    }
}

int main() {
    Queue q;
    initQueue(&q);

    // Simulate receiving votes (from polling stations)
    printf("Receiving votes...\n");
    enqueue(&q, 0);  // Vote for Candidate 0
    enqueue(&q, 1);  // Vote for Candidate 1
    enqueue(&q, 2);  // Vote for Candidate 2
    enqueue(&q, 0);  // Vote for Candidate 0
    enqueue(&q, 3);  // Vote for Candidate 3
    enqueue(&q, 1);  // Vote for Candidate 1
    enqueue(&q, 4);  // Vote for Candidate 4
    enqueue(&q, 2);  // Vote for Candidate 2
    enqueue(&q, 0);  // Vote for Candidate 0

    // Count and display the vote results
    countVotes(&q);

    return 0;
}
```

8. **Airport Runway Management**: Use a linked list to implement a queue for airplanes waiting to land or take off. The system should handle priority for emergency landings and manage runway allocation efficiently.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum {
    LANDING,     // Normal landing
    TAKEOFF,     // Normal takeoff
    EMERGENCY    // Emergency landing
} PlaneType;

// Structure to represent an airplane
typedef struct Airplane {
    char id[10];          // Unique ID for the airplane
    PlaneType type;      // Type of operation (landing, takeoff, emergency)
    struct Airplane* next; // Pointer to next airplane in the queue
} Airplane;

// Queue structure using linked list
typedef struct Queue {
```

```
    Airplane* front; // Front of the queue
    Airplane* rear;  // Rear of the queue
} Queue;

// Function to create a new airplane
Airplane* createAirplane(char id[], PlaneType type) {
    Airplane* newPlane = (Airplane*)malloc(sizeof(Airplane));
    strcpy(newPlane->id, id);
    newPlane->type = type;
    newPlane->next = NULL;
    return newPlane;
}

// Function to initialize the queue
void initQueue(Queue* q) {
    q->front = q->rear = NULL;
}

// Function to check if the queue is empty
int isEmpty(Queue* q) {
    return (q->front == NULL);
}

// Function to enqueue an airplane into the queue
void enqueue(Queue* q, Airplane* plane) {
    if (q->rear == NULL) {
        q->front = q->rear = plane;
        return;
    }
    q->rear->next = plane;
    q->rear = plane;
}

// Function to dequeue an airplane from the queue
Airplane* dequeue(Queue* q) {
    if (isEmpty(q)) {
        return NULL;
    }
    Airplane* temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    return temp;
}

// Function to print the airplane details
void printAirplane(Airplane* plane) {
    if (plane != NULL) {
        char* type = (plane->type == LANDING) ? "Landing" : (plane->type == TAKEOFF) ? "Takeoff" :
"Emergency Landing";
        printf("Airplane ID: %s, Type: %s\n", plane->id, type);
    }
}
```

```
// Function to process the runway (handle priority for emergency landings)
void processRunway(Queue* q) {
    // Process emergency landings first
    Airplane* temp;
    Airplane* prev = NULL;

    // Check for emergency landings in the queue and give them priority
    temp = q->front;
    while (temp != NULL) {
        if (temp->type == EMERGENCY) {
            // If we are at the front, process immediately
            if (prev == NULL) {
                temp = dequeue(q);
                printAirplane(temp);
                free(temp);
            } else {
                // Move emergency landing to the front
                prev->next = temp->next;
                temp->next = q->front;
                q->front = temp;
                printAirplane(temp);
                prev = NULL;
            }
        } else {
            prev = temp;
        }
        temp = temp->next;
    }

    // Process normal airplanes in the queue
    while (!isEmpty(q)) {
        Airplane* plane = dequeue(q);
        printAirplane(plane);
        free(plane);
    }
}

int main() {
    Queue runwayQueue;
    initQueue(&runwayQueue);

    // Add airplanes to the queue
    enqueue(&runwayQueue, createAirplane("A123", LANDING));
    enqueue(&runwayQueue, createAirplane("B456", TAKEOFF));
    enqueue(&runwayQueue, createAirplane("C789", EMERGENCY)); // Emergency landing has priority
    enqueue(&runwayQueue, createAirplane("D101", LANDING));

    // Process the queue with priority for emergency landings
    printf("Processing Runway Operations...\n");
    processRunway(&runwayQueue);

    return 0;
}
```

9. **Stock Trading Simulation**: Develop a program using arrays to simulate a queue for stock trading

orders. The system should manage buy and sell orders, handle order cancellations, and provide real-time updates.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ORDERS 10  // Maximum number of orders in the queue

// Enum for order types
typedef enum { BUY, SELL } OrderType;

// Enum for order status
typedef enum { PENDING, CANCELLED, COMPLETED } OrderStatus;

// Structure to represent an order
typedef struct {
    int orderId;
    OrderType type;
    int quantity;
    float price;
    OrderStatus status;
} StockOrder;

// Circular Queue to store orders
typedef struct {
    StockOrder orders[MAX_ORDERS];
    int front;
    int rear;
    int size;
} OrderQueue;

// Function prototypes
void initializeQueue(OrderQueue *q);
int isQueueFull(OrderQueue *q);
int isQueueEmpty(OrderQueue *q);
void addOrder(OrderQueue *q, int orderId, OrderType type, int quantity, float price);
void cancelOrder(OrderQueue *q, int orderId);
void processOrder(OrderQueue *q);
void displayOrders(OrderQueue *q);

int main() {
    OrderQueue orderQueue;
    initializeQueue(&orderQueue);

    // Simulate adding orders to the queue
    addOrder(&orderQueue, 101, BUY, 100, 150.5);
    addOrder(&orderQueue, 102, SELL, 50, 155.0);
    addOrder(&orderQueue, 103, BUY, 200, 145.0);

    // Display orders
    printf("Current Orders:\n");
    displayOrders(&orderQueue);

    // Cancel an order
```

```
    cancelOrder(&orderQueue, 102);

    // Process orders
    processOrder(&orderQueue);

    // Display updated orders
    printf("\nUpdated Orders:\n");
    displayOrders(&orderQueue);

    return 0;
}

// Function to initialize the queue
void initializeQueue(OrderQueue *q) {
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}

// Function to check if the queue is full
int isQueueFull(OrderQueue *q) {
    return q->size == MAX_ORDERS;
}

// Function to check if the queue is empty
int isQueueEmpty(OrderQueue *q) {
    return q->size == 0;
}

// Function to add an order to the queue
void addOrder(OrderQueue *q, int orderId, OrderType type, int quantity, float price) {
    if (isQueueFull(q)) {
        printf("Error: Order queue is full. Cannot add more orders.\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX_ORDERS;
    q->orders[q->rear].orderId = orderId;
    q->orders[q->rear].type = type;
    q->orders[q->rear].quantity = quantity;
    q->orders[q->rear].price = price;
    q->orders[q->rear].status = PENDING;
    q->size++;
    printf("Order %d added to the queue.\n", orderId);
}

// Function to cancel an order
void cancelOrder(OrderQueue *q, int orderId) {
    if (isQueueEmpty(q)) {
        printf("Error: No orders in the queue to cancel.\n");
        return;
    }

    int found = 0;
    for (int i = 0; i < q->size; i++) {
        int index = (q->front + i) % MAX_ORDERS;
```

```
        if (q->orders[index].orderId == orderId) {
            q->orders[index].status = CANCELLED;
            printf("Order %d has been cancelled.\n", orderId);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Error: Order ID %d not found in the queue.\n", orderId);
    }
}

// Function to process an order (simulating order completion)
void processOrder(OrderQueue *q) {
    if (isQueueEmpty(q)) {
        printf("Error: No orders to process.\n");
        return;
    }

    // Process the first order in the queue
    q->orders[q->front].status = COMPLETED;
    printf("Order %d has been processed and completed.\n", q->orders[q->front].orderId);

    // Move the front pointer to the next order
    q->front = (q->front + 1) % MAX_ORDERS;
    q->size--;
}

// Function to display all orders in the queue
void displayOrders(OrderQueue *q) {
    if (isQueueEmpty(q)) {
        printf("No orders to display.\n");
        return;
    }

    for (int i = 0; i < q->size; i++) {
        int index = (q->front + i) % MAX_ORDERS;
        StockOrder *order = &q->orders[index];

        char *type = order->type == BUY ? "BUY" : "SELL";
        char *status = order->status == PENDING ? "PENDING" : (order->status == CANCELLED ?
"CANCELLED" : "COMPLETED");

        printf("Order ID: %d, Type: %s, Quantity: %d, Price: %.2f, Status: %s\n",
               order->orderId, type, order->quantity, order->price, status);
    }
}
```

10. **Conference Registration System**: Implement a queue using linked lists for managing registrations at a conference. The system should handle walk-in registrations, pre-registrations, and cancellations.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
typedef struct Registrant {
    int id;
    char name[50];
    char registrationType[20]; // "Pre-registered" or "Walk-in"
    struct Registrant* next;
} Registrant;

typedef struct Queue {
    Registrant* front;
    Registrant* rear;
} Queue;

// Function to initialize an empty queue
void initQueue(Queue* queue) {
    queue->front = queue->rear = NULL;
}

// Function to create a new registrant node
Registrant* createRegistrant(int id, const char* name, const char* registrationType) {
    Registrant* newRegistrant = (Registrant*)malloc(sizeof(Registrant));
    newRegistrant->id = id;
    strcpy(newRegistrant->name, name);
    strcpy(newRegistrant->registrationType, registrationType);
    newRegistrant->next = NULL;
    return newRegistrant;
}

// Enqueue function to add a registrant to the queue
void enqueue(Queue* queue, int id, const char* name, const char* registrationType) {
    Registrant* newRegistrant = createRegistrant(id, name, registrationType);
    if (queue->rear == NULL) {
        queue->front = queue->rear = newRegistrant;
    } else {
        queue->rear->next = newRegistrant;
        queue->rear = newRegistrant;
    }
    printf("Registrant %s (ID: %d) added successfully!\n", name, id);
}

// Dequeue function to remove a registrant from the queue
void dequeue(Queue* queue) {
    if (queue->front == NULL) {
        printf("No registrants to dequeue!\n");
        return;
    }
    Registrant* temp = queue->front;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    printf("Registrant %s (ID: %d) removed successfully!\n", temp->name, temp->id);
    free(temp);
}
```

```
// Function to cancel a registrant’s registration
void cancelRegistrant(Queue* queue, int id) {
    if (queue->front == NULL) {
        printf("No registrants to cancel!\n");
        return;
    }
    Registrant* temp = queue->front;
    Registrant* prev = NULL;
    while (temp != NULL && temp->id != id) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Registrant with ID %d not found!\n", id);
        return;
    }

    if (prev == NULL) {
        queue->front = temp->next;
    } else {
        prev->next = temp->next;
    }

    if (temp == queue->rear) {
        queue->rear = prev;
    }

    printf("Registrant %s (ID: %d) canceled successfully!\n", temp->name, temp->id);
    free(temp);
}

// Function to display all registrants in the queue
void listRegistrants(Queue* queue) {
    if (queue->front == NULL) {
        printf("No registrants in the queue.\n");
        return;
    }
    Registrant* temp = queue->front;
    while (temp != NULL) {
        printf("ID: %d, Name: %s, Type: %s\n", temp->id, temp->name, temp->registrationType);
        temp = temp->next;
    }
}

// Main function
int main() {
    Queue queue;
    initQueue(&queue);

    // Adding some pre-registered and walk-in attendees
    enqueue(&queue, 101, "Alice", "Pre-registered");
    enqueue(&queue, 102, "Bob", "Walk-in");
    enqueue(&queue, 103, "Charlie", "Pre-registered");
```

```
    printf("Current Registrants:\n");
    listRegistrants(&queue);

    // Canceling a registrant
    cancelRegistrant(&queue, 102);

    printf("\nCurrent Registrants after cancellation:\n");
    listRegistrants(&queue);

    // Removing the first registrant (dequeue)
    dequeue(&queue);

    printf("\nCurrent Registrants after dequeue:\n");
    listRegistrants(&queue);

    return 0;
}
```

11. **Political Debate Audience Management**: Use arrays to implement a queue for managing the audience at a political debate. The system should handle entry, seating arrangements, and priority access for media personnel.

```
#include <stdio.h>
#include <string.h>

#define MAX_QUEUE_SIZE 100  // Maximum number of people in the queue
#define NAME_LENGTH 50  // Max length of a person’s name

// Structure to represent a person in the queue
typedef struct {
    char name[NAME_LENGTH];
    int isMedia;  // 1 for media, 0 for regular audience
} Person;

// Queue structure to manage the audience
typedef struct {
    Person queue[MAX_QUEUE_SIZE];
    int front;
    int rear;
} Queue;

// Initialize the queue
void initQueue(Queue* q) {
    q->front = 0;
    q->rear = 0;
}

// Check if the queue is full
int isFull(Queue* q) {
    return (q->rear == MAX_QUEUE_SIZE);
}

// Check if the queue is empty
int isEmpty(Queue* q) {
    return (q->front == q->rear);
```

```
}

// Add a new person to the queue (with priority for media)
void enqueue(Queue* q, char* name, int isMedia) {
    if (isFull(q)) {
        printf("Queue is full. Cannot add more people.\n");
        return;
    }

    // If the person is media, we place them at the front (priority access)
    if (isMedia) {
        // Shift all regular people one position back
        for (int i = q->rear; i > q->front; i--) {
            q->queue[i] = q->queue[i - 1];
        }
        // Insert the media person at the front
        strncpy(q->queue[q->front].name, name, NAME_LENGTH);
        q->queue[q->front].isMedia = 1;
        q->rear++;
    } else {
        // Regular audience - insert at the end of the queue
        strncpy(q->queue[q->rear].name, name, NAME_LENGTH);
        q->queue[q->rear].isMedia = 0;
        q->rear++;
    }
}

// Remove a person from the queue
void dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No one to remove.\n");
        return;
    }

    // Print the person being removed
    printf("%s is leaving the debate.\n", q->queue[q->front].name);

    // Shift the rest of the queue forward
    q->front++;
}

// Display the current queue
void displayQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("No one in the queue.\n");
        return;
    }

    printf("Current queue:\n");
    for (int i = q->front; i < q->rear; i++) {
        printf("%d. %s%s\n", i - q->front + 1, q->queue[i].name,
               q->queue[i].isMedia ? " (Media)" : "");
    }
}
```

```
// Main function to test the queue implementation
int main() {
    Queue q;
    initQueue(&q);

    // Test adding people to the queue
    enqueue(&q, "John Doe", 0);  // Regular audience
    enqueue(&q, "Jane Smith", 1); // Media
    enqueue(&q, "Alice Johnson", 0); // Regular audience
    enqueue(&q, "Media Reporter", 1); // Media

    // Display the current queue
    displayQueue(&q);

    // Dequeue some people
    dequeue(&q);
    dequeue(&q);

    // Display the queue again
    displayQueue(&q);

    return 0;
}
```

12. **Bank Loan Application Processing**: Develop a queue using linked lists to manage loan applications at a bank. The system should prioritize applications based on the loan amount and applicant's credit score.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to represent a loan application
typedef struct LoanApplication {
    char applicantName[100];
    float loanAmount;
    int creditScore;
    struct LoanApplication* next;  // Pointer to the next loan application
} LoanApplication;

// Function to create a new loan application node
LoanApplication* createLoanApplication(char* name, float loanAmount, int creditScore) {
    LoanApplication* newApp = (LoanApplication*)malloc(sizeof(LoanApplication));
    strcpy(newApp->applicantName, name);
    newApp->loanAmount = loanAmount;
    newApp->creditScore = creditScore;
    newApp->next = NULL;
    return newApp;
}

// Function to insert a loan application in the queue based on priority
void insertLoanApplication(LoanApplication** head, LoanApplication* newApp) {
    // If the queue is empty, insert as the first node
    if (*head == NULL) {
        *head = newApp;
```

```
        return;
    }

    LoanApplication* current = *head;
    LoanApplication* previous = NULL;

    // Traverse the list to find the right position based on loanAmount and creditScore
    while (current != NULL &&
           (current->loanAmount > newApp->loanAmount ||
           (current->loanAmount == newApp->loanAmount && current->creditScore >
newApp->creditScore))) {
        previous = current;
        current = current->next;
    }

    // Insert the new loan application in the found position
    if (previous == NULL) {
        newApp->next = *head;
        *head = newApp;
    } else {
        previous->next = newApp;
        newApp->next = current;
    }
}

// Function to process and display the loan applications in the queue
void processLoanApplications(LoanApplication* head) {
    if (head == NULL) {
        printf("No loan applications to process.\n");
        return;
    }

    printf("Processing loan applications:\n");
    LoanApplication* current = head;
    while (current != NULL) {
        printf("Applicant: %s, Loan Amount: %.2f, Credit Score: %d\n",
               current->applicantName, current->loanAmount, current->creditScore);
        current = current->next;
    }
}

// Main function to demonstrate the loan processing system
int main() {
    LoanApplication* loanQueue = NULL;

    // Example loan applications
    insertLoanApplication(&loanQueue, createLoanApplication("Alice", 50000, 720));
    insertLoanApplication(&loanQueue, createLoanApplication("Bob", 20000, 680));
    insertLoanApplication(&loanQueue, createLoanApplication("Charlie", 100000, 750));
    insertLoanApplication(&loanQueue, createLoanApplication("David", 30000, 710));

    // Process the loan applications in priority order
    processLoanApplications(loanQueue);

    return 0;
```

}

13. **Online Shopping Checkout System**: Implement a queue using arrays for an online shopping platform’s checkout system. The program should handle multiple customers checking out simultaneously and manage inventory updates.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Constants for maximum products and customers
#define MAX_PRODUCTS 5
#define MAX_CUSTOMERS 10

// Product structure
typedef struct {
    char name[30];
    int quantity;
    float price;
} Product;

// Customer structure
typedef struct {
    int customer_id;
    int product_id;
    int quantity;
} Customer;

// Queue structure to hold customers
typedef struct {
    Customer customers[MAX_CUSTOMERS];
    int front, rear;
} Queue;

// Initialize queue
void initQueue(Queue *q) {
    q->front = 0;
    q->rear = -1;
}

// Check if queue is empty
int isQueueEmpty(Queue *q) {
    return q->rear == -1;
}

// Check if queue is full
int isQueueFull(Queue *q) {
    return q->rear == MAX_CUSTOMERS - 1;
}

// Add a customer to the queue
void enqueue(Queue *q, Customer customer) {
    if (isQueueFull(q)) {
        printf("Queue is full! Cannot add customer.\n");
        return;
```

```
    }
    q->customers[++(q->rear)] = customer;
}

// Remove a customer from the queue
Customer dequeue(Queue *q) {
    if (isQueueEmpty(q)) {
        printf("Queue is empty! No customers to process.\n");
        exit(1);  // Exit if no customer is available
    }
    return q->customers[(q->front)++];
}

// Print product inventory
void printInventory(Product inventory[], int num_products) {
    printf("\nInventory:\n");
    for (int i = 0; i < num_products; i++) {
        printf("%d. %s - Quantity: %d, Price: %.2f\n", i+1, inventory[i].name, inventory[i].quantity,
inventory[i].price);
    }
}

// Process a customer's checkout
void processCheckout(Product inventory[], int num_products, Customer customer) {
    int product_id = customer.product_id - 1; // Adjust for 0-based index

    if (product_id < 0 || product_id >= num_products) {
        printf("Invalid product ID.\n");
        return;
    }

    // Check if there is enough inventory
    if (inventory[product_id].quantity < customer.quantity) {
        printf("Not enough stock for %s. Only %d items left.\n", inventory[product_id].name,
inventory[product_id].quantity);
    } else {
        // Deduct from inventory and print the purchase details
        inventory[product_id].quantity -= customer.quantity;
        printf("Customer %d purchased %d of %s. Total: %.2f\n", customer.customer_id, customer.quantity,
inventory[product_id].name, inventory[product_id].price * customer.quantity);
    }
}

int main() {
    // Define the inventory
    Product inventory[MAX_PRODUCTS] = {
        {"Laptop", 10, 999.99},
        {"Smartphone", 15, 599.99},
        {"Headphones", 25, 99.99},
        {"Smartwatch", 30, 199.99},
        {"Tablet", 12, 399.99}
    };

    // Define the queue for customers
    Queue queue;
```

```
    initQueue(&queue);

    // Sample customer data
    Customer customers[] = {
        {1, 1, 2},  // Customer 1 wants to buy 2 laptops
        {2, 3, 1},  // Customer 2 wants to buy 1 headphone
        {3, 2, 3},  // Customer 3 wants to buy 3 smartphones
        {4, 5, 1},  // Customer 4 wants to buy 1 tablet
    };

    // Enqueue customers
    for (int i = 0; i < 4; i++) {
        enqueue(&queue, customers[i]);
    }

    // Process customers in the queue
    while (!isQueueEmpty(&queue)) {
        Customer customer = dequeue(&queue);
        processCheckout(inventory, MAX_PRODUCTS, customer);
    }

    // Print final inventory after processing all customers
    printInventory(inventory, MAX_PRODUCTS);

    return 0;
}
```

14. **Public Transport Scheduling**: Use linked lists to implement a queue for managing bus arrivals and departures at a terminal. The system should handle peak hours, off-peak hours, and prioritize express buses.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PEAK_HOUR_START 7 // 7 AM
#define PEAK_HOUR_END 9   // 9 AM

// Define bus types
#define EXPRESS 1
#define REGULAR 0

// Bus structure
typedef struct Bus {
    int busID;
    int busType; // 1 for express, 0 for regular
    int arrivalTime; // In 24-hour format
    struct Bus* next; // Link to next bus
} Bus;

// Queue (linked list) structure
typedef struct Queue {
    Bus* front;
    Bus* rear;
} Queue;
```

```
// Function to create a new bus
Bus* createBus(int busID, int busType, int arrivalTime) {
    Bus* newBus = (Bus*)malloc(sizeof(Bus));
    newBus->busID = busID;
    newBus->busType = busType;
    newBus->arrivalTime = arrivalTime;
    newBus->next = NULL;
    return newBus;
}

// Function to initialize the queue
void initializeQueue(Queue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to check if it’s peak hour
int isPeakHour(int time) {
    return time >= PEAK_HOUR_START && time <= PEAK_HOUR_END;
}

// Function to enqueue a bus into the queue (with priority for express buses)
void enqueue(Queue* queue, int busID, int busType, int arrivalTime) {
    Bus* newBus = createBus(busID, busType, arrivalTime);

    // If the bus is express, add it to the front
    if (busType == EXPRESS) {
        if (queue->front == NULL) {
            queue->front = newBus;
            queue->rear = newBus;
        } else {
            newBus->next = queue->front;
            queue->front = newBus;
        }
    } else { // For regular buses, add to the rear
        if (queue->rear == NULL) {
            queue->front = newBus;
            queue->rear = newBus;
        } else {
            queue->rear->next = newBus;
            queue->rear = newBus;
        }
    }
}

// Function to dequeue a bus from the front of the queue
Bus* dequeue(Queue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty.\n");
        return NULL;
    }

    Bus* temp = queue->front;
    queue->front = queue->front->next;
```

```
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    return temp;
}

// Function to display the buses in the queue
void displayQueue(Queue* queue) {
    if (queue->front == NULL) {
        printf("No buses in the queue.\n");
        return;
    }

    Bus* temp = queue->front;
    printf("Bus ID | Type   | Arrival Time\n");
    while (temp != NULL) {
        printf("%6d | %s    | %02d:00\n", temp->busID,
            (temp->busType == EXPRESS) ? "Express" : "Regular", temp->arrivalTime);
        temp = temp->next;
    }
}

// Main function
int main() {
    Queue busQueue;
    initializeQueue(&busQueue);

    // Adding buses to the queue
    enqueue(&busQueue, 101, EXPRESS, 8);  // Express bus at 8 AM
    enqueue(&busQueue, 102, REGULAR, 9);  // Regular bus at 9 AM
    enqueue(&busQueue, 103, REGULAR, 6);  // Regular bus at 6 AM
    enqueue(&busQueue, 104, EXPRESS, 7);  // Express bus at 7 AM

    // Display the queue
    printf("Bus queue before any departure:\n");
    displayQueue(&busQueue);

    // Dequeue buses (simulate departures)
    printf("\nBuses departing from the terminal:\n");
    while (busQueue.front != NULL) {
        Bus* departingBus = dequeue(&busQueue);
        printf("Bus ID %d departed. Type: %s at %02d:00\n",
            departingBus->busID,
            (departingBus->busType == EXPRESS) ? "Express" : "Regular",
            departingBus->arrivalTime);
        free(departingBus);  // Free memory for the departed bus
    }

    return 0;
}
```

15. **Political Rally Crowd Control**: Develop a queue using arrays to manage the crowd at a political rally. The system should handle entry, exit, and VIP sections, ensuring safety and order

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

#define MAX_CAPACITY 10  // Maximum capacity of the rally

// Structure to represent a queue
typedef struct Queue {
    char people[MAX_CAPACITY][50];  // Array to store names of people
    int front;
    int rear;
} Queue;

// Function to initialize a queue
void initializeQueue(Queue *queue) {
    queue->front = 0;
    queue->rear = 0;
}

// Function to check if a queue is full
int isFull(Queue *queue) {
    return (queue->rear == MAX_CAPACITY);
}

// Function to check if a queue is empty
int isEmpty(Queue *queue) {
    return (queue->front == queue->rear);
}

// Function to add a person to the queue (general or VIP)
void enter(Queue *queue, char person[], int isVIP) {
    if (isFull(queue)) {
        printf("Rally is at full capacity. No more entry allowed.\n");
        return;
    }

    if (isVIP) {
        // Add VIP person to the front of the queue
        for (int i = queue->rear; i > queue->front; i--) {
            strcpy(queue->people[i], queue->people[i - 1]);
        }
        strcpy(queue->people[queue->front], person);
    } else {
        // Add general person to the rear of the queue
        strcpy(queue->people[queue->rear], person);
    }

    queue->rear++;
    printf("%s has entered the rally.\n", person);
}

// Function to remove a person from the queue (exit)
void exitQueue(Queue *queue) {
    if (isEmpty(queue)) {
        printf("No one is left in the rally.\n");
        return;
```

```
    }

    printf("%s has exited the rally.\n", queue->people[queue->front]);
    queue->front++;
}

// Function to process the next VIP in the queue
void processVIP(Queue *queue) {
    if (isEmpty(queue)) {
        printf("No VIPs in the queue.\n");
        return;
    }

    printf("Processing VIP: %s\n", queue->people[queue->front]);
    queue->front++;
}

// Function to process the next general person in the queue
void processGeneral(Queue *queue) {
    if (isEmpty(queue)) {
        printf("No general public in the queue.\n");
        return;
    }

    printf("Processing general public: %s\n", queue->people[queue->front]);
    queue->front++;
}

// Function to display the current state of the queue
void showQueues(Queue *queue) {
    printf("Current Queue:\n");
    if (isEmpty(queue)) {
        printf("No one is in the rally.\n");
        return;
    }

    for (int i = queue->front; i < queue->rear; i++) {
        printf("%s\n", queue->people[i]);
    }
}

int main() {
    Queue rallyQueue;
    initializeQueue(&rallyQueue);

    // Example of entering people into the rally
    enter(&rallyQueue, "Alice", 0);  // General
    enter(&rallyQueue, "Bob", 1);    // VIP
    enter(&rallyQueue, "Charlie", 0); // General
    enter(&rallyQueue, "David", 1);   // VIP

    // Show current state of the queue
    showQueues(&rallyQueue);

    // Process people (VIP first)
```

```
    processVIP(&rallyQueue);
    processGeneral(&rallyQueue);

    // Exit a person
    exitQueue(&rallyQueue);

    // Show current state after exit
    showQueues(&rallyQueue);

    return 0;
}
```

16. **Financial Transaction Processing**: Implement a queue using linked lists to process financial transactions. The system should handle deposits, withdrawals, and transfers, ensuring real-time processing and accuracy.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NAME_LENGTH 50

// Transaction types
typedef enum {
    DEPOSIT,
    WITHDRAWAL,
    TRANSFER
} TransactionType;

// Structure for a financial transaction
typedef struct Transaction {
    int transaction_id;
    TransactionType type;
    char account_holder[MAX_NAME_LENGTH];
    double amount;
    char target_account[MAX_NAME_LENGTH]; // For transfers
} Transaction;

// Node structure for the queue (linked list)
typedef struct QueueNode {
    Transaction transaction;
    struct QueueNode* next;
} QueueNode;

// Queue structure
typedef struct {
    QueueNode* front;
    QueueNode* rear;
} TransactionQueue;

// Function to create a new queue node
QueueNode* createQueueNode(Transaction transaction) {
    QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
    if (!newNode) {
        printf("Memory allocation failed!\n");
```

```
        exit(1);
    }
    newNode->transaction = transaction;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize the transaction queue
void initQueue(TransactionQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to check if the queue is empty
int isQueueEmpty(TransactionQueue* queue) {
    return queue->front == NULL;
}

// Function to enqueue a transaction into the queue
void enqueue(TransactionQueue* queue, Transaction transaction) {
    QueueNode* newNode = createQueueNode(transaction);
    if (isQueueEmpty(queue)) {
        queue->front = newNode;
        queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Function to dequeue a transaction from the queue
Transaction dequeue(TransactionQueue* queue) {
    if (isQueueEmpty(queue)) {
        printf("Queue is empty. No transactions to process.\n");
        exit(1);
    }

    QueueNode* temp = queue->front;
    Transaction transaction = temp->transaction;
    queue->front = queue->front->next;

    if (queue->front == NULL) {
        queue->rear = NULL;
    }

    free(temp);
    return transaction;
}

// Function to process a deposit transaction
void processDeposit(Transaction transaction) {
    printf("Processing Deposit: $%.2f for %s\n", transaction.amount, transaction.account_holder);
}

// Function to process a withdrawal transaction
```

```
void processWithdrawal(Transaction transaction) {
    printf("Processing Withdrawal: $%.2f for %s\n", transaction.amount, transaction.account_holder);
}

// Function to process a transfer transaction
void processTransfer(Transaction transaction) {
    printf("Processing Transfer: $%.2f from %s to %s\n", transaction.amount, transaction.account_holder,
transaction.target_account);
}

// Function to process the next transaction in the queue
void processTransaction(TransactionQueue* queue) {
    if (isQueueEmpty(queue)) {
        printf("No transactions to process.\n");
        return;
    }

    Transaction transaction = dequeue(queue);

    switch (transaction.type) {
        case DEPOSIT:
            processDeposit(transaction);
            break;
        case WITHDRAWAL:
            processWithdrawal(transaction);
            break;
        case TRANSFER:
            processTransfer(transaction);
            break;
    }
}

int main() {
    // Initialize the transaction queue
    TransactionQueue queue;
    initQueue(&queue);

    // Create some sample transactions
    Transaction t1 = {1, DEPOSIT, "Alice", 500.00, ""};
    Transaction t2 = {2, WITHDRAWAL, "Bob", 200.00, ""};
    Transaction t3 = {3, TRANSFER, "Charlie", 300.00, "David"};

    // Enqueue the transactions
    enqueue(&queue, t1);
    enqueue(&queue, t2);
    enqueue(&queue, t3);

    // Process the transactions
    processTransaction(&queue); // Process deposit
    processTransaction(&queue); // Process withdrawal
    processTransaction(&queue); // Process transfer

    return 0;
}
```

17. **Election Polling Booth Management**: Use arrays to implement a queue for managing voters at a polling booth. The system should handle voter registration, verification, and ensure smooth voting process.

```
#include <stdio.h>
#include <string.h>

#define MAX_VOTERS 100  // Max number of voters in the queue

// Queue structure to hold voter data
typedef struct {
    int id;            // Voter ID
    char name[50];      // Voter's name
} Voter;

// Polling Booth (Queue structure)
typedef struct {
    Voter voters[MAX_VOTERS];  // Array of voters
    int front, rear;           // Queue pointers
} PollingBooth;

// Function to initialize the polling booth (queue)
void initializePollingBooth(PollingBooth *booth) {
    booth->front = -1;
    booth->rear = -1;
}

// Function to check if the queue is full
int isQueueFull(PollingBooth *booth) {
    return booth->rear == MAX_VOTERS - 1;
}

// Function to check if the queue is empty
int isQueueEmpty(PollingBooth *booth) {
    return booth->front == -1 || booth->front > booth->rear;
}

// Function to register a new voter
void registerVoter(PollingBooth *booth, int id, const char *name) {
    if (isQueueFull(booth)) {
        printf("Polling booth is full. Cannot register more voters.\n");
        return;
    }
    booth->rear++;
    booth->voters[booth->rear].id = id;
    strcpy(booth->voters[booth->rear].name, name);
    if (booth->front == -1) {
        booth->front = 0; // First voter gets to be processed
    }
    printf("Voter %d (%s) registered successfully.\n", id, name);
}

// Function to verify and allow a voter to vote
void verifyAndVote(PollingBooth *booth) {
    if (isQueueEmpty(booth)) {
```

```
        printf("No voters in the queue.\n");
        return;
    }
    Voter voter = booth->voters[booth->front];
    printf("Voter %d (%s) is verified and allowed to vote.\n", voter.id, voter.name);
    booth->front++; // Remove the voter from the queue
    if (booth->front > booth->rear) {
        booth->front = booth->rear = -1; // Reset queue if it’s empty
    }
}

// Function to display all voters in the queue (for debugging)
void displayQueue(PollingBooth *booth) {
    if (isQueueEmpty(booth)) {
        printf("No voters in the queue.\n");
        return;
    }
    printf("Voters in the queue:\n");
    for (int i = booth->front; i <= booth->rear; i++) {
        printf("Voter ID: %d, Name: %s\n", booth->voters[i].id, booth->voters[i].name);
    }
}

int main() {
    PollingBooth booth;
    initializePollingBooth(&booth);

    int choice, id;
    char name[50];

    while (1) {
        printf("\nPolling Booth Management System\n");
        printf("1. Register Voter\n");
        printf("2. Verify and Allow Voter to Vote\n");
        printf("3. Display Voters in Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Register Voter
                printf("Enter Voter ID: ");
                scanf("%d", &id);
                printf("Enter Voter Name: ");
                scanf(" %[^\n]", name); // To accept spaces in name
                registerVoter(&booth, id, name);
                break;

            case 2: // Verify and Allow Voter to Vote
                verifyAndVote(&booth);
                break;

            case 3: // Display Voters in Queue
                displayQueue(&booth);
                break;
```

```
            case 4: // Exit
                printf("Exiting polling booth management system.\n");
                return 0;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

18. **Hospital Emergency Room Queue**: Develop a queue using linked lists to manage patients in a hospital emergency room. The system should prioritize patients based on the severity of their condition and manage multiple doctors.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NAME_LENGTH 100

// Enum for severity levels
typedef enum {
    CRITICAL = 1,
    HIGH = 2,
    MEDIUM = 3,
    LOW = 4
} SeverityLevel;

// Doctor structure
typedef struct {
    int doctorID;
    char name[MAX_NAME_LENGTH];
} Doctor;

// Patient structure
typedef struct Patient {
    int patientID;
    char name[MAX_NAME_LENGTH];
    SeverityLevel severity;
    struct Patient *next; // Pointer to the next patient in the queue
    Doctor *assignedDoctor; // Pointer to the doctor assigned to this patient
} Patient;

// Function prototypes
Patient* createPatient(int id, const char* name, SeverityLevel severity);
Doctor* createDoctor(int id, const char* name);
void enqueue(Patient **head, Patient *newPatient);
Patient* dequeue(Patient **head);
void assignDoctors(Patient *head, Doctor *doctors, int numDoctors);
void printQueue(Patient *head);
void freeQueue(Patient *head);
```

```
int main() {
    Patient *queue = NULL;

    // Create doctors
    Doctor doctors[] = {
        {1, "Dr. Smith"},
        {2, "Dr. Jones"},
        {3, "Dr. Adams"}
    };
    int numDoctors = sizeof(doctors) / sizeof(doctors[0]);

    // Create some patients
    enqueue(&queue, createPatient(1, "John Doe", CRITICAL));
    enqueue(&queue, createPatient(2, "Jane Smith", MEDIUM));
    enqueue(&queue, createPatient(3, "Emily Clark", HIGH));
    enqueue(&queue, createPatient(4, "Michael Brown", LOW));

    // Assign doctors to patients
    assignDoctors(queue, doctors, numDoctors);

    // Print the queue with doctor assignments
    printQueue(queue);

    // Dequeue and process patients
    printf("\nProcessing patients...\n");
    while (queue != NULL) {
        Patient *patient = dequeue(&queue);
        printf("Patient ID: %d, Name: %s, Severity: %d, Doctor: %s\n",
               patient->patientID, patient->name, patient->severity, patient->assignedDoctor->name);
        free(patient);
    }

    return 0;
}

// Function to create a new patient
Patient* createPatient(int id, const char* name, SeverityLevel severity) {
    Patient *newPatient = (Patient*)malloc(sizeof(Patient));
    newPatient->patientID = id;
    strcpy(newPatient->name, name);
    newPatient->severity = severity;
    newPatient->next = NULL;
    newPatient->assignedDoctor = NULL;
    return newPatient;
}

// Function to create a new doctor
Doctor* createDoctor(int id, const char* name) {
    Doctor *newDoctor = (Doctor*)malloc(sizeof(Doctor));
    newDoctor->doctorID = id;
    strcpy(newDoctor->name, name);
    return newDoctor;
}

// Function to enqueue a new patient in the queue
```

```
void enqueue(Patient **head, Patient *newPatient) {
    if (*head == NULL || newPatient->severity < (*head)->severity) {
        newPatient->next = *head;
        *head = newPatient;
    } else {
        Patient *current = *head;
        while (current->next != NULL && current->next->severity <= newPatient->severity) {
            current = current->next;
        }
        newPatient->next = current->next;
        current->next = newPatient;
    }
}

// Function to dequeue a patient from the queue
Patient* dequeue(Patient **head) {
    if (*head == NULL) {
        return NULL;
    }
    Patient *temp = *head;
    *head = (*head)->next;
    temp->next = NULL;
    return temp;
}

// Function to assign doctors to patients in a round-robin fashion
void assignDoctors(Patient *head, Doctor *doctors, int numDoctors) {
    int doctorIndex = 0;
    while (head != NULL) {
        head->assignedDoctor = &doctors[doctorIndex];
        doctorIndex = (doctorIndex + 1) % numDoctors;
        head = head->next;
    }
}

// Function to print the queue of patients
void printQueue(Patient *head) {
    printf("Hospital Emergency Room Queue:\n");
    while (head != NULL) {
        printf("Patient ID: %d, Name: %s, Severity: %d\n", head->patientID, head->name, head->severity);
        head = head->next;
    }
}

// Function to free the memory allocated for the queue
void freeQueue(Patient *head) {
    while (head != NULL) {
        Patient *temp = head;
        head = head->next;
        free(temp);
    }
}
```

19. **Political Survey Data Collection**: Implement a queue using arrays to manage data collection for a political survey. The system should handle multiple surveyors collecting data simultaneously and ensure

data consistency

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

#define MAX_QUEUE_SIZE 100
#define MAX_RESPONSE_LENGTH 256

// Survey response structure
typedef struct {
    char data[MAX_RESPONSE_LENGTH];
} SurveyResponse;

// Queue structure
typedef struct {
    SurveyResponse queue[MAX_QUEUE_SIZE];
    int front, rear;
    pthread_mutex_t lock; // Mutex for synchronization
} SurveyQueue;

// Initialize the queue
void initQueue(SurveyQueue *q) {
    q->front = -1;
    q->rear = -1;
    pthread_mutex_init(&q->lock, NULL);
}

// Check if the queue is full
int isFull(SurveyQueue *q) {
    return (q->rear == MAX_QUEUE_SIZE - 1);
}

// Check if the queue is empty
int isEmpty(SurveyQueue *q) {
    return (q->front == -1);
}

// Enqueue function to add a survey response to the queue
void enqueue(SurveyQueue *q, const char *response) {
    pthread_mutex_lock(&q->lock); // Lock the queue for thread safety

    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue new response.\n");
    } else {
        if (q->front == -1) {
            q->front = 0; // If it's the first element, set front to 0
        }
        q->rear++;
        strncpy(q->queue[q->rear].data, response, MAX_RESPONSE_LENGTH);
        printf("Survey response enqueued: %s\n", response);
    }

    pthread_mutex_unlock(&q->lock); // Unlock after operation
```

```
}

// Dequeue function to remove and return a survey response
SurveyResponse dequeue(SurveyQueue *q) {
    pthread_mutex_lock(&q->lock); // Lock the queue for thread safety

    SurveyResponse response = {""};

    if (isEmpty(q)) {
        printf("Queue is empty. No response to dequeue.\n");
    } else {
        strncpy(response.data, q->queue[q->front].data, MAX_RESPONSE_LENGTH);
        if (q->front == q->rear) {
            q->front = q->rear = -1; // Queue becomes empty
        } else {
            q->front++;
        }
        printf("Survey response dequeued: %s\n", response.data);
    }

    pthread_mutex_unlock(&q->lock); // Unlock after operation
    return response;
}

// Simulate a surveyor collecting data
void *surveyor(void *arg) {
    SurveyQueue *q = (SurveyQueue *)arg;
    // Simulating collecting survey responses
    const char *responses[] = {
        "Yes, I support the new policy.",
        "No, I don't support the new policy.",
        "I'm undecided on the policy."
    };

    for (int i = 0; i < 3; i++) {
        enqueue(q, responses[i]);
    }

    return NULL;
}

int main() {
    SurveyQueue q;
    initQueue(&q);

    // Create threads to simulate multiple surveyors
    pthread_t surveyors[3];
    for (int i = 0; i < 3; i++) {
        if (pthread_create(&surveyors[i], NULL, surveyor, (void *)&q) != 0) {
            perror("Failed to create thread");
            return 1;
        }
    }

    // Wait for all threads to finish
```

```
    for (int i = 0; i < 3; i++) {
        pthread_join(surveyors[i], NULL);
    }

    // Dequeue all responses (to simulate processing)
    while (!isEmpty(&q)) {
        SurveyResponse response = dequeue(&q);
    }

    pthread_mutex_destroy(&q.lock); // Clean up the mutex
    return 0;
}
```

20. **Financial Market Data Analysis**: Use linked lists to implement a queue for analyzing financial market data. The system should handle large volumes of data, perform real-time analysis, and generate insights for decision-making.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    float price;           // Market data: stock price or other financial metric
    struct Node* next;     // Pointer to the next node
} Node;

typedef struct Queue {
    Node* front;           // Points to the front of the queue
    Node* rear;            // Points to the rear of the queue
    int size;              // The number of elements in the queue
} Queue;

// Function to create an empty queue
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->front = queue->rear = NULL;
    queue->size = 0;
    return queue;
}

// Function to create a new node with market data (stock price)
Node* createNode(float price) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->price = price;
    newNode->next = NULL;
    return newNode;
}

// Enqueue function: Add new market data (price) to the queue
void enqueue(Queue* queue, float price) {
    Node* newNode = createNode(price);

    if (queue->rear == NULL) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
```

```
        queue->rear = newNode;
    }

    queue->size++;
}

// Dequeue function: Remove the front element from the queue
float dequeue(Queue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty!\n");
        return -1;
    }

    Node* temp = queue->front;
    float price = temp->price;
    queue->front = queue->front->next;

    if (queue->front == NULL) {
        queue->rear = NULL;
    }

    free(temp);
    queue->size--;
    return price;
}

// Function to display the current queue
void displayQueue(Queue* queue) {
    Node* temp = queue->front;
    while (temp != NULL) {
        printf("%.2f ", temp->price);
        temp = temp->next;
    }
    printf("\n");
}

// Function to calculate the moving average of the last ’n’ prices
float calculateMovingAverage(Queue* queue, int n) {
    if (queue->size < n) {
        printf("Not enough data to calculate moving average.\n");
        return -1;
    }

    Node* temp = queue->front;
    float sum = 0;
    int count = 0;

    while (temp != NULL && count < n) {
        sum += temp->price;
        temp = temp->next;
        count++;
    }

    return sum / n;
}
```

```
// Function to perform real-time analysis on the market data in the queue
void realTimeAnalysis(Queue* queue) {
    if (queue->size == 0) {
        printf("No data available for analysis.\n");
        return;
    }

    // Example analysis: Calculate the moving average of the last 5 prices
    int window = 5;
    float movingAvg = calculateMovingAverage(queue, window);

    if (movingAvg != -1) {
        printf("Moving average of the last %d prices: %.2f\n", window, movingAvg);
    }

    // You can add more complex analysis here, such as volatility, price trends, etc.
}

// Main function to simulate market data arrival and analysis
int main() {
    Queue* marketQueue = createQueue();

    // Simulate incoming market data (stock prices)
    enqueue(marketQueue, 100.5);
    enqueue(marketQueue, 102.3);
    enqueue(marketQueue, 104.8);
    enqueue(marketQueue, 101.2);
    enqueue(marketQueue, 106.5);
    enqueue(marketQueue, 107.0);

    // Perform real-time analysis
    realTimeAnalysis(marketQueue);

    // Display the current state of the queue
    printf("Current Queue: ");
    displayQueue(marketQueue);

    // Dequeue an element and re-analyze
    printf("Dequeueing: %.2f\n", dequeue(marketQueue));
    realTimeAnalysis(marketQueue);

    // Final state of the queue
    printf("Final Queue: ");
    displayQueue(marketQueue);

    return 0;
}
```