21st – Jan

Queues using arrays and linked list

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>

struct Order
{
    char type[5]; int
    price;
    int quantity;
};

struct Queue
{
    int size; int
    front; int
    rear;
    struct Order *orders;
};

void createQueue(struct Queue *, int);
int enqueue(struct Queue *, const char *, int, int);
```

```
int dequeue(struct Queue *, struct Order *);
void matchOrders(struct Queue *, struct Queue *); void
displayQueue(struct Queue, const char *);

int main()
{
    struct Queue buyQueue, sellQueue;
    createQueue(&buyQueue, 5);
    createQueue(&sellQueue, 5);
    enqueue(&buyQueue, "Buy", 100, 4);
    enqueue(&buyQueue, "Buy", 150, 5);
    enqueue(&sellQueue, "Sell", 75, 3);
    enqueue(&sellQueue, "Sell", 50, 2); printf("\nInitial
    Orders\n"); displayQueue(buyQueue, "Buy");
    displayQueue(sellQueue, "Sell"); printf("\nMatching
    Orders\n"); matchOrders(&buyQueue, &sellQueue);
    printf("\nRemaining Orders\n");
    displayQueue(buyQueue, "Buy");
    displayQueue(sellQueue, "Sell");
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
```

```
    q->front = q->rear = -1;
    q->orders = (struct Order *)malloc(size * sizeof(struct Order));
}

int enqueue(struct Queue *q, const char *type, int price, int quantity)
{
    if (q->rear == q->size - 1)
    {
        printf("Queue is full\n"); return -1;
    }
    else
    {
        q->rear++;
        strcpy(q->orders[q->rear].type, type); q-
        >orders[q->rear].price = price;
        q->orders[q->rear].quantity = quantity;
        printf("Added %s order: Price=%d, Quantity=%d\n", type, price, quantity); return 0;
    }
}

int dequeue(struct Queue *q, struct Order *order)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
```

```
        return -1;
    }
    else
    {
        q->front++;
        *order = q->orders[q->front];
        return 0;
    }
}

void matchOrders(struct Queue *buyQueue, struct Queue *sellQueue)
{
    while (buyQueue->front != buyQueue->rear && sellQueue->front !=
                                                        sellQueue->rear)
    {
        struct Order buyOrder = buyQueue->orders[buyQueue->front + 1]; struct
        Order sellOrder = sellQueue->orders[sellQueue->front + 1]; if (buyOrder.price
        >= sellOrder.price)
        {
            int matchedQuantity = (buyOrder.quantity < sellOrder.quantity) ?
                                                    buyOrder.quantity : sellOrder.quantity;
          printf("Matched Order: BUY Price=%d, SELL Price=%d, Quantity=%d\n",
          buyOrder.price, sellOrder.price, matchedQuantity); buyQueue->orders[buyQueue-
          >front + 1].quantity -= matchedQuantity; sellQueue->orders[sellQueue->front +
          1].quantity -= matchedQuantity; if (buyQueue->orders[buyQueue->front +
          1].quantity == 0)
                dequeue(buyQueue, &buyOrder);
```

```
            else if (sellQueue->orders[sellQueue->front + 1].quantity == 0) dequeue(sellQueue,
                &sellOrder);
        }
        else
            break;
    }
}

void displayQueue(struct Queue q, const char *type)
{
    if (q.front == q.rear) printf("Queue is
        empty\n");
    else
    {
        printf("%s Orders:\n", type);
        for (int i = q.front + 1; i <= q.rear; i++)
          printf("Price=%d, Quantity=%d\n", q.orders[i].price, q.orders[i].quantity);
    }
}
```

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>

struct CustomerData
{
    char name[50];
    int priority;  // 1 = Gold, 2 = Silver, 3 = Bronze
    char timestamp[20];
    struct CustomerData *next;
} *front = NULL, *rear = NULL;

void enqueue(const char *name, int priority, const char *timestamp);
void display();
void dequeue();
void processQueue();

int main() {
    enqueue("XYZ", 1, "2025-01-21 11:00");
    enqueue("ABC", 2, "2025-01-21 11:15");
    enqueue("DEF", 3, "2025-01-21 11:30");
    enqueue("MNO", 1, "2025-01-21 11:45");
    printf("Customer Queue before processing:\n");
    display();
    processQueue();
    printf("\nCustomer Queue after processing:\n");
    display();
    return 0;
}
```

```
void enqueue(const char *name, int priority, const char *timestamp)
{
    struct CustomerData *newCustomer = (struct CustomerData *)
                                malloc(sizeof(struct CustomerData));
    if (newCustomer == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    strncpy(newCustomer->name, name, sizeof(newCustomer->name) - 1);
    newCustomer->name[sizeof(newCustomer->name) - 1] = '\0';
    newCustomer->priority = priority;
    strncpy(newCustomer->timestamp, timestamp,
                                sizeof(newCustomer->timestamp) - 1);
    newCustomer->timestamp[sizeof(newCustomer->timestamp) - 1] = '\0';
    newCustomer->next = NULL;
    if (front == NULL || front->priority > priority)
    {
        newCustomer->next = front;
        front = newCustomer;
        if (rear == NULL)
            rear = newCustomer;
    }
    else
    {
        struct CustomerData *current = front;
        while (current->next != NULL && current->next->priority <= priority)
```

```
            current = current->next;
        newCustomer->next = current->next;
        current->next = newCustomer;
        if (newCustomer->next == NULL)
            rear = newCustomer;
    }
    printf("Enqueued customer: Name = %s, Priority = %d, Timestamp = %s\n",
                                                    name, priority, timestamp);
}

void display()
{
    struct CustomerData *current = front;
    if (current == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    while (current != NULL)
    {
        printf("Name = %s, Priority = %d, Timestamp = %s\n", current->name,
                                        current->priority, current->timestamp);
        current = current->next;
    }
}
void dequeue()
{
    if (front == NULL)
```

```
    {
        printf("Queue is empty, no customer to serve\n");
        return;
    }
    struct CustomerData *temp = front;
    printf("Serving customer: Name = %s, Priority = %d, Timestamp = %s\n",
                                front->name, front->priority, front->timestamp);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(temp);
}

void processQueue()
{
    while (front != NULL)
        dequeue();
}
```

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

```
struct Attendee
{
    char name[50];
    int type;      // 1 for VIP, 0 for non VIP
};

struct Queue
{
    int size;
    int front;
    int rear;
    struct Attendee *attendees;
};

void createQueue(struct Queue *, int);
int enqueue(struct Queue *, const char *, int);
int dequeue(struct Queue *, struct Attendee *);
void checkIn(struct Queue *, struct Queue *);
void displayQueue(struct Queue, const char *);

int main()
{
    struct Queue nonVipQueue, vipQueue;
    createQueue(&nonVipQueue, 3);
    createQueue(&vipQueue, 3);
    enqueue(&nonVipQueue, "ABC", 0);
    enqueue(&vipQueue, "DEF", 1);
```

```
    enqueue(&nonVipQueue, "GHI", 0);
    enqueue(&vipQueue, "JKL", 1);
    enqueue(&nonVipQueue, "MNO", 0);
    printf("\nInitial Registration:\n");
    displayQueue(vipQueue, "VIP Queue");
    displayQueue(nonVipQueue, "Non-VIP Queue");
    printf("\nChecking In Attendees:\n");
    checkIn(&vipQueue, &nonVipQueue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->attendees = (struct Attendee *)malloc(size * sizeof(struct Attendee));
}

int enqueue(struct Queue *q, const char *name, int type)
{
    if (q->rear == q->size - 1)
    {
        printf("Queue is full\n");
        return -1;
    }
    else
    {
```

```
        q->rear++;
        strcpy(q->attendees[q->rear].name, name);
        q->attendees[q->rear].type = type;
        printf("Registered attendee: %s (VIP: %d)\n", name, type);
        return 0;
    }
}

int dequeue(struct Queue *q, struct Attendee *attendee)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return -1;
    }
    else
    {
        q->front++;
        *attendee = q->attendees[q->front];
        return 0;
    }
}

void checkIn(struct Queue *vipQueue, struct Queue *nonVipQueue)
{
    struct Attendee attendee;
    while (vipQueue->front != vipQueue->rear)
```

```
    {
        dequeue(vipQueue, &attendee);
        printf("VIP Attendee Checked In: %s\n", attendee.name);
    }
    while (nonVipQueue->front != nonVipQueue->rear)
    {
        dequeue(nonVipQueue, &attendee);
        printf("Non-VIP Attendee Checked In: %s\n", attendee.name);
    }
}

void displayQueue(struct Queue q, const char *type1)
{
    if (q.front == q.rear)
        printf("%s is empty\n", type1);
    else
    {
        printf("%s:\n", type1);
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("Name: %s\n", q.attendees[i].name);
    }
}
```

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 <string.h>

struct CustomerData
{
    char name[50];
    int transactionTime;
    struct CustomerData *next;
} *front = NULL, *rear = NULL;

struct Teller
{
    int id;
    int availableTime;
};

void enqueue(const char *name, int transactionTime);
void dequeue();
void processTellers(struct Teller tellers[], int numTellers);
void simulateBank(int numTellers, int numCustomers);

int main()
{
    simulateBank(2, 4);
    return 0;
}
```

```
void enqueue(const char *name, int transactionTime)
{
    struct CustomerData *newCustomer = (struct CustomerData *)
                                        malloc(sizeof(struct CustomerData));
    if (newCustomer == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    strncpy(newCustomer->name, name, sizeof(newCustomer->name) - 1);
    newCustomer->name[sizeof(newCustomer->name) - 1] = '\0';
    newCustomer->transactionTime = transactionTime;
    newCustomer->next = NULL;
    if (front == NULL)
        front = rear = newCustomer;
    else
    {
        rear->next = newCustomer;
        rear = newCustomer;
    }
    printf("Customer enqueued: Name = %s, Transaction Time = %d seconds\n",
                                            name, transactionTime);
}

void dequeue()
{
    if (front == NULL)
```

```
    {
        printf("Queue is empty, no customer to serve\n");
        return;
    }
    struct CustomerData *temp = front;
    printf("Serving customer: Name = %s, Transaction Time = %d seconds\n",
                                        front->name, front->transactionTime);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(temp);
}

void processTellers(struct Teller tellers[], int numTellers)
{
    for (int i = 0; i < numTellers; i++)
    {
        if (tellers[i].availableTime == 0 && front != NULL)
        {
            printf("Teller %d is serving customer: Name = %s,
                                        Transaction Time = %d seconds\n",
                tellers[i].id, front->name, front->transactionTime);
            tellers[i].availableTime = front->transactionTime;
            dequeue();
        }
    }
}
```

```
void simulateBank(int numTellers, int numCustomers)
{
    struct Teller tellers[numTellers];
    for (int i = 0; i < numTellers; i++)
    {
        tellers[i].id = i + 1;
        tellers[i].availableTime = 0;
    }
    for (int i = 0; i < numCustomers; i++)
    {
        char customerName[50];
        customerName[0] = 'C';
        customerName[1] = 'u';
        customerName[2] = 's';
        customerName[3] = 't';
        customerName[4] = 'o';
        customerName[5] = 'm';
        customerName[6] = 'e';
        customerName[7] = 'r';
        customerName[8] = i + '1';
        customerName[9] = '\0';
        int transactionTime = 3 + (i % 5);
        enqueue(customerName, transactionTime);
    }
    while (front != NULL)
    {
        processTellers(tellers, numTellers);
```

```
        for (int i = 0; i < numTellers; i++)
        {
            if (tellers[i].availableTime > 0)
                tellers[i].availableTime--;
        }
    }
    printf("\nAll customers have been served\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>

struct DataFeed
{
    char instrument[50];
    float price;
    int sequence;
};

struct Queue
{
    int size;
```

```
    int front;
    int rear;
    struct DataFeed *feeds;
};

void createQueue(struct Queue *, int);
int enqueue(struct Queue *, const char *, float, int);
int dequeue(struct Queue *, struct DataFeed *);
void processDataFeeds(struct Queue *);
void displayQueue(struct Queue);

int main()
{
    struct Queue dataQueue;
    createQueue(&dataQueue, 5);
    enqueue(&dataQueue, "ABC", 1500.50, 1);
    enqueue(&dataQueue, "JKL", 1800.80, 2);
    enqueue(&dataQueue, "MNO", 8800.00, 3);
    printf("\nInitial Data Feed Queue:\n");
    displayQueue(dataQueue);
    printf("\nProcessing Data Feed:\n");
    processDataFeeds(&dataQueue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
```

```
    q->size = size;
    q->front = q->rear = -1;
    q->feeds = (struct DataFeed *)malloc(size * sizeof(struct DataFeed));
}

int enqueue(struct Queue *q, const char *instrument, float price, int sequence)
{
    if (q->rear == q->size - 1)
    {
        printf("Queue is full\n");
        return -1;
    }
    else
    {
        q->rear++;
        strcpy(q->feeds[q->rear].instrument, instrument);
        q->feeds[q->rear].price = price;
        q->feeds[q->rear].sequence = sequence;
        printf("Added Data Feed: %s - Price=%.2f, Sequence=%d\n", instrument,
                                                    price, sequence);
        return 0;
    }
}

int dequeue(struct Queue *q, struct DataFeed *feed)
{
    if (q->front == q->rear)
```

```
    {
        printf("Queue is empty\n");
        return -1;
    }
    else
    {
        q->front++;
        *feed = q->feeds[q->front];
        return 0;
    }
}

void processDataFeeds(struct Queue *q)
{
    struct DataFeed feed;
    while (q->front != q->rear)
    {
        dequeue(q, &feed);
        printf("Processing Data Feed: %s - Price=%.2f, Sequence=%d\n",
                                    feed.instrument, feed.price, feed.sequence);
    }
}

// Function to display the contents of the queue (for monitoring)
void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
```

```
        printf("Queue is empty\n");
    else
    {
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s - Price=%.2f, Sequence=%d\n", q.feeds[i].instrument,
                                    q.feeds[i].price, q.feeds[i].sequence);
    }
}
```

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  <string.h>
#include <unistd.h>

struct Car
{
    int id;
    struct Car* next;
};

struct Queue
{
    struct Car* front;
```

```
    struct Car* rear;
};

void initializeQueue(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int carId);
void dequeue(struct Queue* q);
void trafficLightSystem(struct Queue* q);

int main()
{
    struct Queue carQueue;
    initializeQueue(&carQueue);
    enqueue(&carQueue, 1);
    enqueue(&carQueue, 2);
    enqueue(&carQueue, 3);
    trafficLightSystem(&carQueue);
    while (!isQueueEmpty(&carQueue))
        dequeue(&carQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = NULL;
    q->rear = NULL;
}
```

```
int isQueueEmpty(struct Queue* q)
{
    return q->front == NULL;
}

void enqueue(struct Queue* q, int carId)
{
    struct Car* newCar = (struct Car*)malloc(sizeof(struct Car));
    if (!newCar)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newCar->id = carId;
    newCar->next = NULL;
    if (q->rear == NULL)
        q->front = q->rear = newCar;
    else
    {
        q->rear->next = newCar;
        q->rear = newCar;
    }
    printf("Car %d arrived at the traffic light\n", carId);
}

void dequeue(struct Queue* q)
{
```

```
    if (isQueueEmpty(q))
    {
        printf("No cars at the traffic light\n");
        return;
    }
    struct Car* temp = q->front;
    printf("Car %d is passing through the green light.\n", temp->id);
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
}

void trafficLightSystem(struct Queue* q)
{
    char light = 'R';
    int time = 0;
    while (time < 30)
    {
        printf("\nTime: %d seconds\n", time);
        if (light == 'R')
            printf("Traffic light: RED\n");
        else
        {
            printf("Traffic light: GREEN\n");
            if (!isQueueEmpty(q))
                dequeue(q);
```

```
            else
                printf("No cars waiting\n");
        }
        time += 5;
        sleep(1);
        light = (light == 'R') ? 'G' : 'R';
    }
}
```

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>

struct Vote
{
    int pollingStationId;
    char candidate[50];
    int voteCount;
};

struct Queue
{
    int size;
```

```
    int front;
    int rear;
    struct Vote *votes;
};

void createQueue(struct Queue *, int);
int enqueue(struct Queue *, int, const char *, int);
int dequeue(struct Queue *, struct Vote *);
void processVotes(struct Queue *);
void displayQueue(struct Queue);

int main()
{
    struct Queue voteQueue;
    createQueue(&voteQueue, 4);
    enqueue(&voteQueue, 1, "ABC", 400);
    enqueue(&voteQueue, 2, "DEF", 500);
    enqueue(&voteQueue, 1, "HIJ", 300);
    enqueue(&voteQueue, 3, "KLM", 100);
    printf("\nInitial Votes in Queue:\n");
    displayQueue(voteQueue);
    printf("\nProcessing Votes:\n");
    processVotes(&voteQueue);
    return 0;
}

void createQueue(struct Queue *q, int size)
```

```
{
    q->size = size;
    q->front = q->rear = -1;
    q->votes = (struct Vote *)malloc(size * sizeof(struct Vote));
}

int enqueue(struct Queue *q, int pollingStationId, const char *candidate, int voteCount)
{
    if (q->rear == q->size - 1)
    {
        printf("Queue is full\n");
        return -1;
    }
    else
    {
        q->rear++;
        q->votes[q->rear].pollingStationId = pollingStationId;
        strcpy(q->votes[q->rear].candidate, candidate);
        q->votes[q->rear].voteCount = voteCount;
        printf("Added Vote: Polling Station %d, Candidate: %s, Votes: %d\n",
                pollingStationId, candidate, voteCount);
        return 0;
    }
}

int dequeue(struct Queue *q, struct Vote *vote)
{
```

```
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return -1;
    }
    else
    {
        q->front++;
        *vote = q->votes[q->front];
        return 0;
    }
}

void processVotes(struct Queue *q)
{
    struct Vote vote;
    while (q->front != q->rear)
    {
        dequeue(q, &vote);
        printf("Processing Vote: Polling Station %d, Candidate: %s, Votes: %d\n",
                vote.pollingStationId, vote.candidate, vote.voteCount);
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
```

```
        printf("Queue is empty\n");
    else
    {
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("Polling Station %d, Candidate: %s, Votes: %d\n",
          q.votes[i].pollingStationId, q.votes[i].candidate, q.votes[i].voteCount);
    }
}
```

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>

struct Airplane
{
    int id;
    char type[10];
    int priority;
    struct Airplane* next;
};

struct Queue
{
```

```
    struct Airplane* front;
    struct Airplane* rear;
};

void initializeQueue(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* type, int priority);
void dequeue(struct Queue* q);
void manageRunway(struct Queue* q, int timeSlots);

int main()
{
    struct Queue runwayQueue;
    initializeQueue(&runwayQueue);
    enqueue(&runwayQueue, 101, "land", 0);
    enqueue(&runwayQueue, 102, "takeoff", 0);
    enqueue(&runwayQueue, 103, "land", 1);
    enqueue(&runwayQueue, 104, "land", 0);
    enqueue(&runwayQueue, 105, "takeoff", 1);
    manageRunway(&runwayQueue, 5);
    while (!isQueueEmpty(&runwayQueue))
        dequeue(&runwayQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
```

```
    q->front = NULL;
    q->rear = NULL;
}

int isQueueEmpty(struct Queue* q)
{
    return q->front == NULL;
}

void enqueue(struct Queue* q, int id, const char* type, int priority)
{
    struct Airplane* newPlane = (struct Airplane*)malloc(sizeof(struct Airplane));
    if (!newPlane)
    {
        printf("Memory allocation failed!\n");
        return;
    }
    newPlane->id = id;
    strcpy(newPlane->type, type);
    newPlane->priority = priority;
    newPlane->next = NULL;
    if (q->front == NULL || priority == 1)
    {
        newPlane->next = q->front;
        q->front = newPlane;
        if (q->rear == NULL)
            q->rear = newPlane;
```

```
    }
    else
    {
        q->rear->next = newPlane;
        q->rear = newPlane;
    }
     printf("Airplane %d (%s, priority: %d) added to the queue.\n", id, type, priority);
}

void dequeue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("No airplanes in the queue\n");
        return;
    }
    struct Airplane* temp = q->front;
    printf("Airplane %d (%s) is using the runway\n", temp->id, temp->type);
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
}

void manageRunway(struct Queue* q, int timeSlots)
{
    for (int i = 0; i < timeSlots; i++)
```

```
    {
        printf("\nTime Slot %d:\n", i + 1);
        if (!isQueueEmpty(q))
            dequeue(q);
        else
            printf("Runway is idle\n");
    }
}
```

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 5

struct Order
{
    int orderId;
    char type[5];
    int price;
    int quantity;
};
```

```
struct Queue
{
    struct Order orders[MAX_ORDERS];
    int front;
    int rear;
};

int isFull(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, int orderId, const char *type, int price, int quantity);
void dequeue(struct Queue *q);
void cancelOrder(struct Queue *q, int orderId);
void displayQueue(struct Queue *q);

int main()
{
    struct Queue orderQueue;
    orderQueue.front = -1;
    orderQueue.rear = -1;
    enqueue(&orderQueue, 1, "Buy", 100, 10);
    enqueue(&orderQueue, 2, "Sell", 150, 5);
    enqueue(&orderQueue, 3, "Buy", 120, 20);
    printf("\nCurrent Orders in Queue:\n");
    displayQueue(&orderQueue);
    printf("\nProcessing Orders:\n");
    dequeue(&orderQueue);
    printf("\nAttempting to cancel Order ID:\n");
```

```
    cancelOrder(&orderQueue, 2);
    printf("\nRemaining Orders in Queue:\n");
    displayQueue(&orderQueue);
    printf("\nProcessing Remaining Orders:\n");
    dequeue(&orderQueue);
    return 0;
}

int isFull(struct Queue *q)
{
    return q->rear == MAX_ORDERS - 1;
}

int isEmpty(struct Queue *q)
{
    return q->front == q->rear;
}

void enqueue(struct Queue *q, int orderId, const char *type, int price, int quantity)
{
    if (isFull(q))
    {
        printf("Queue is full\n");
        return;
    }
    q->rear++;
    q->orders[q->rear].orderId = orderId;
```

```
    strcpy(q->orders[q->rear].type, type);
    q->orders[q->rear].price = price;
    q->orders[q->rear].quantity = quantity;
    printf("Order added: ID=%d, Type=%s, Price=%d, Quantity=%d\n", orderId,
                                    type, price, quantity);
}

void dequeue(struct Queue *q)
{
    if (isEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    q->front++;
    struct Order order = q->orders[q->front];
    printf("Processing Order: ID=%d, Type=%s, Price=%d, Quantity=%d\n",
                order.orderId, order.type, order.price, order.quantity);
}

void cancelOrder(struct Queue *q, int orderId)
{
    if (isEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
```

```
    int found = 0;
    for (int i = q->front + 1; i <= q->rear; i++) {
        if (q->orders[i].orderId == orderId)
        {
            for (int j = i; j < q->rear; j++)
                q->orders[j] = q->orders[j + 1];
            q->rear--;
            printf("Order with ID=%d has been cancelled successfully\n", orderId);
            found = 1;
            break;
        }
    }
    if (!found)
        printf("Order with ID=%d not found.\n", orderId);
}

void displayQueue(struct Queue *q)
{
    if (isEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    for (int i = q->front + 1; i <= q->rear; i++)
        printf("ID=%d, Type=%s, Price=%d, Quantity=%d\n", q->orders[i].orderId,
                q->orders[i].type, q->orders[i].price, q->orders[i].quantity);
}
```

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>

struct Attendee
{
   int id;
   char name[50];
   struct Attendee* next;
};

struct Queue
{
   struct Attendee* front;
   struct Attendee* rear;
};

void initializeQueue(struct Queue* q);
void registerAttendee(struct Queue* q, int id, const char* name);
void cancelRegistration(struct Queue* q, int id);
void displayQueue(struct Queue* q);

int main()
{
```

```
    struct Queue queue;
    initializeQueue(&queue);
    registerAttendee(&queue, 1, "ABC");
    registerAttendee(&queue, 2, "MNO");
    registerAttendee(&queue, 3, "XYZ");
    displayQueue(&queue);
    cancelRegistration(&queue, 2);
    displayQueue(&queue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = NULL;
    q->rear = NULL;
}

void registerAttendee(struct Queue* q, int id, const char* name)
{
    struct Attendee* newAttendee = (struct Attendee*)malloc
                                                (sizeof(struct Attendee));
    if (!newAttendee)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newAttendee->id = id;
```

```
    strcpy(newAttendee->name, name);
    newAttendee->next = NULL;
    if (q->rear == NULL)
        q->front = q->rear = newAttendee;
    else
    {
        q->rear->next = newAttendee;
        q->rear = newAttendee;
    }
    printf("Attendee %s (ID: %d) registered\n", name, id);
}

void cancelRegistration(struct Queue* q, int id)
{
    struct Attendee *temp = q->front, *prev = NULL;
    while (temp != NULL && temp->id != id)
    {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL)
    {
        printf("Attendee with ID %d not found.\n", id);
        return;
    }
    if (prev == NULL)
        q->front = temp->next;
```

```
    else
        prev->next = temp->next;
    if (temp == q->rear)
        q->rear = prev;
    free(temp);
    printf("Registration for Attendee ID %d canceled\n", id);
}

void displayQueue(struct Queue* q)
{
    struct Attendee* temp = q->front;
    printf("Current  registrations:\n");
    while (temp != NULL)
    {
        printf("ID: %d, Name: %s\n", temp->id, temp->name);
        temp = temp->next;
    }
}
```

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 <stdlib.h>

#define MAX_CAPACITY 100
```

```
struct Queue
{
    int data[MAX_CAPACITY];
    int front, rear;
};

void initializeQueue(struct Queue* q);
int isQueueFull(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int id);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue audienceQueue;
    initializeQueue(&audienceQueue);
    enqueue(&audienceQueue, 1);
    enqueue(&audienceQueue, 2);
    enqueue(&audienceQueue, 3);
    displayQueue(&audienceQueue);
    dequeue(&audienceQueue);
    displayQueue(&audienceQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
```

```
{
    q->front = q->rear = -1;
}

int isQueueFull(struct Queue* q)
{
    return (q->rear + 1) % MAX_CAPACITY == q->front;
}

int isQueueEmpty(struct Queue* q)
{
    return q->front == -1;
}

void enqueue(struct Queue* q, int id)
{
    if (isQueueFull(q))
    {
        printf("Queue is full\n");
        return;
    }
    if (q->front == -1)
        q->front = 0;
    q->rear = (q->rear + 1) % MAX_CAPACITY;
    q->data[q->rear] = id;
    printf("Audience member %d entered the queue\n", id);
}
```

```
void dequeue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    int id = q->data[q->front];
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front = (q->front + 1) % MAX_CAPACITY;
    printf("Audience member %d seated\n", id);
}

void displayQueue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Current queue: ");
    for (int i = q->front; i != q->rear; i = (i + 1) % MAX_CAPACITY)
        printf("%d ", q->data[i]);
    printf("%d\n", q->data[q->rear]);
}
```

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>

struct LoanApplication
{
   int id;
   float amount;
   int creditScore;
   struct LoanApplication* next;
};

struct Queue
{
   struct LoanApplication* front;
};

void initializeQueue(struct Queue* q);
void enqueue(struct Queue* q, int id, float amount, int creditScore);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
```

```
    struct Queue loanQueue;
    initializeQueue(&loanQueue);
    enqueue(&loanQueue, 1, 50000, 750);
    enqueue(&loanQueue, 2, 100000, 800);
    enqueue(&loanQueue, 3, 70000, 750);
    displayQueue(&loanQueue);
    dequeue(&loanQueue);
    displayQueue(&loanQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = NULL;
}

void enqueue(struct Queue* q, int id, float amount, int creditScore)
{
     struct LoanApplication* newApplication = (struct LoanApplication*)
                                        malloc(sizeof(struct LoanApplication));
    newApplication->id = id;
    newApplication->amount = amount;
    newApplication->creditScore = creditScore;
    newApplication->next = NULL;
    if (q->front == NULL || creditScore > q->front->creditScore ||
        (creditScore == q->front->creditScore && amount > q->front->amount))
    {
```

```
        newApplication->next = q->front;
        q->front = newApplication;
    }
    else
    {
        struct LoanApplication* temp = q->front;
        while (temp->next != NULL &&
            (temp->next->creditScore > creditScore ||
            (temp->next->creditScore == creditScore && temp->next->amount >=
                                                                amount)))
                temp = temp->next;
        newApplication->next = temp->next;
        temp->next = newApplication;
    }
    printf("Loan Application ID %d (Amount: %.2f, Credit Score: %d) added\n",
                                id, amount, creditScore);
}

void dequeue(struct Queue* q)
{
    if (q->front == NULL)
    {
        printf("No loan applications to process\n");
        return;
    }
    struct LoanApplication* temp = q->front;
    printf("Processing Loan Application ID %d (Amount: %.2f, Credit Score:
```

```
                        %d)\n", temp->id, temp->amount, temp->creditScore);
    q->front = q->front->next;
    free(temp);
}

void displayQueue(struct Queue* q)
{
    struct LoanApplication* temp = q->front;
    printf("Loan Applications:\n");
    while (temp != NULL)
    {
        printf("ID: %d, Amount: %.2f, Credit Score: %d\n", temp->id,
                                        temp->amount, temp->creditScore);
       temp = temp->next;
    }
}
```

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>

#define MAX_CUSTOMERS 100

struct Customer
```

```
{
    int id;
    int items;
};

struct Queue
{
    struct Customer data[MAX_CUSTOMERS];
    int front, rear;
};

void initializeQueue(struct Queue* q);
int isQueueFull(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int id, int items);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue checkoutQueue;
    initializeQueue(&checkoutQueue);
    enqueue(&checkoutQueue, 1, 5);
    enqueue(&checkoutQueue, 2, 3);
    enqueue(&checkoutQueue, 3, 7);
    displayQueue(&checkoutQueue);
    dequeue(&checkoutQueue);
```

```
    displayQueue(&checkoutQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = q->rear = -1;
}

int isQueueFull(struct Queue* q)
{
    return (q->rear + 1) % MAX_CUSTOMERS == q->front;
}

int isQueueEmpty(struct Queue* q)
{
    return q->front == -1;
}

void enqueue(struct Queue* q, int id, int items)
{
    if (isQueueFull(q))
    {
        printf("Queue is full\n");
        return;
    }
    if (q->front == -1) q->front = 0;
```

```
    q->rear = (q->rear + 1) % MAX_CUSTOMERS;
    q->data[q->rear].id = id;
    q->data[q->rear].items = items;
    printf("Customer %d with %d items added to the queue\n", id, items);
}

void dequeue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("No customers to process\n");
        return;
    }
    struct Customer customer = q->data[q->front];
    printf("Processing Customer %d with %d items\n", customer.id,
                                                    customer.items);
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front = (q->front + 1) % MAX_CUSTOMERS;
}

void displayQueue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("No customers in the queue\n");
```

```
        return;
    }
    printf("Current queue:\n");
    for (int i = q->front; i != q->rear; i = (i + 1) % MAX_CUSTOMERS)
        printf("Customer ID: %d, Items: %d\n", q->data[i].id, q->data[i].items);
    printf("Customer ID: %d, Items: %d\n", q->data[q->rear].id,
                                        q->data[q->rear].items);
}
```

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>

struct Bus
{
    int id;
    char type[10];
    struct Bus* next;
};

struct Queue
{
    struct Bus* front;
```

```
    struct Bus* rear;
};

void initializeQueue(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* type);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue busQueue;
    initializeQueue(&busQueue);
    enqueue(&busQueue, 1, "regular");
    enqueue(&busQueue, 2, "express");
    enqueue(&busQueue, 3, "regular");
    displayQueue(&busQueue);
    dequeue(&busQueue);
    displayQueue(&busQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = q->rear = NULL;
}

void enqueue(struct Queue* q, int id, const char* type)
```

```
{
    struct Bus* newBus = (struct Bus*)malloc(sizeof(struct Bus));
    newBus->id = id;
    strcpy(newBus->type, type);
    newBus->next = NULL;
    if (strcmp(type, "express") == 0)
    {
        newBus->next = q->front;
        q->front = newBus;
        if (q->rear == NULL) q->rear = newBus;
    }
    else
    {
        if (q->rear == NULL)
            q->front = q->rear = newBus;
        else
        {
            q->rear->next = newBus;
            q->rear = newBus;
        }
    }
    printf("Bus %d (%s) added to the queue\n", id, type);
}

void dequeue(struct Queue* q)
{
    if (q->front == NULL)
```

```
    {
        printf("No buses to process\n");
        return;
    }
    struct Bus* temp = q->front;
    printf("Bus %d (%s) is departing\n", temp->id, temp->type);
    q->front = q->front->next;
    if (q->front == NULL) q->rear = NULL;
    free(temp);
}

void displayQueue(struct Queue* q)
{
    struct Bus* temp = q->front;
    printf("Buses in queue:\n");
    while (temp != NULL)
    {
        printf("ID: %d, Type: %s\n", temp->id, temp->type);
        temp = temp->next;
    }
}
```

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>
```

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

#define MAX_CAPACITY 200

struct Person
{
    int id;
    char type[10];
};

struct Queue
{
    struct Person data[MAX_CAPACITY];
    int front, rear;
};

void initializeQueue(struct Queue* q);
int isQueueFull(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* type);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue rallyQueue;
```

```
    initializeQueue(&rallyQueue);
    enqueue(&rallyQueue, 1, "VIP");
    enqueue(&rallyQueue, 2, "General");
    enqueue(&rallyQueue, 3, "VIP");
    displayQueue(&rallyQueue);
    dequeue(&rallyQueue);
    displayQueue(&rallyQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = q->rear = -1;
}

int isQueueFull(struct Queue* q)
{
    return (q->rear + 1) % MAX_CAPACITY == q->front;
}

int isQueueEmpty(struct Queue* q)
{
    return q->front == -1;
}

void enqueue(struct Queue* q, int id, const char* type)
{
```

```
    if (isQueueFull(q))
    {
        printf("Queue is full\n");
        return;
    }
    if (q->front == -1) q->front = 0;
    q->rear = (q->rear + 1) % MAX_CAPACITY;
    q->data[q->rear].id = id;
    strcpy(q->data[q->rear].type,  type);
    printf("%s Person ID %d added to the queue.\n", type, id);
}

void dequeue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("No one in the queue\n");
        return;
    }
    struct Person person = q->data[q->front];
    printf("%s Person ID %d is entering the rally\n", person.type, person.id);
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front = (q->front + 1) % MAX_CAPACITY;
}
```

```
void displayQueue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Current queue:\n");
    for (int i = q->front; i != q->rear; i = (i + 1) % MAX_CAPACITY)
        printf("ID: %d, Type: %s\n", q->data[i].id, q->data[i].type);
    printf("ID: %d, Type: %s\n", q->data[q->rear].id, q->data[q->rear].type);
}
```

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>

struct Transaction
{
    int id;
    char type[20];
    float amount;
    struct Transaction* next;
```

```
};

struct Queue
{
    struct Transaction* front;
    struct Transaction* rear;
};

void initializeQueue(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* type, float amount);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue transactionQueue;
    initializeQueue(&transactionQueue);
    enqueue(&transactionQueue, 1, "Deposit", 5000);
    enqueue(&transactionQueue, 2, "Withdrawal", 2000);
    enqueue(&transactionQueue, 3, "Transfer", 3000);
    displayQueue(&transactionQueue);
    dequeue(&transactionQueue);
    displayQueue(&transactionQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
```

```
{
    q->front = q->rear = NULL;
}

void enqueue(struct Queue* q, int id, const char* type, float amount)
{
    struct Transaction* newTransaction = (struct Transaction*)malloc
                                        (sizeof(struct Transaction));
    if (!newTransaction)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newTransaction->id = id;
    strcpy(newTransaction->type, type);
    newTransaction->amount = amount;
    newTransaction->next = NULL;
    if (q->rear == NULL)
        q->front = q->rear = newTransaction;
    else
    {
        q->rear->next = newTransaction;
        q->rear = newTransaction;
    }
    printf("Transaction ID %d (%s, Amount: %.2f) added\n", id, type, amount);
}
```

```
void dequeue(struct Queue* q)
{
    if (q->front == NULL)
    {
        printf("No transactions to process\n");
        return;
    }
    struct Transaction* temp = q->front;
    printf("Processing Transaction ID %d (%s, Amount: %.2f)\n", temp->id,
                                                temp->type, temp->amount);
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
}

void displayQueue(struct Queue* q)
{
    struct Transaction* temp = q->front;
    printf("Pending Transactions:\n");
    while (temp != NULL)
    {
        printf("ID: %d, Type: %s, Amount: %.2f\n", temp->id, temp->type,
                                                temp->amount);
        temp = temp->next;
    }
}
```

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 <stdlib.h>

#define MAX_VOTERS 100

struct Voter
{
    int id;
    char name[50];
};

struct Queue
{
    struct Voter data[MAX_VOTERS];
    int front, rear;
};

void initializeQueue(struct Queue* q);
int isQueueFull(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* name);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);
```

```
int main()
{
    struct Queue voterQueue;
    initializeQueue(&voterQueue);
    enqueue(&voterQueue, 1, "XYZ");
    enqueue(&voterQueue, 2, "MNO");
    enqueue(&voterQueue, 3, "PQR");
    displayQueue(&voterQueue);
    dequeue(&voterQueue);
    displayQueue(&voterQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = q->rear = -1;
}

int isQueueFull(struct Queue* q)
{
    return (q->rear + 1) % MAX_VOTERS == q->front;
}

int isQueueEmpty(struct Queue* q)
{
    return q->front == -1;
}
```

```
void enqueue(struct Queue* q, int id, const char* name)
{
    if (isQueueFull(q))
    {
        printf("Queue is full\n");
        return;
    }
    if (q->front == -1) q->front = 0;
    q->rear = (q->rear + 1) % MAX_VOTERS;
    q->data[q->rear].id = id;
    strcpy(q->data[q->rear].name, name);
    printf("Voter ID %d (%s) added to the queue\n", id, name);
}

void dequeue(struct Queue* q)
{
    if (isQueueEmpty(q)) {
        printf("No voters to process\n");
        return;
    }
    struct Voter voter = q->data[q->front];
    printf("Processing Voter ID %d (%s)\n", voter.id, voter.name);
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front = (q->front + 1) % MAX_VOTERS;
}
```

```
void displayQueue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Voters in queue:\n");
    for (int i = q->front; i != q->rear; i = (i + 1) % MAX_VOTERS)
        printf("ID: %d, Name: %s\n", q->data[i].id, q->data[i].name);
    printf("ID: %d, Name: %s\n", q->data[q->rear].id, q->data[q->rear].name);
}
```

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>

struct Patient
{
    int id;
    char name[50];
    int severity;
    struct Patient* next;
```

```
};

struct Queue
{
    struct Patient* front;
};

void initializeQueue(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* name, int severity);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue erQueue;
    initializeQueue(&erQueue);
    enqueue(&erQueue, 1, "XYZ", 5);
    enqueue(&erQueue, 2, "PQR", 8);
    enqueue(&erQueue, 3, "ABC", 4);
    displayQueue(&erQueue);
    dequeue(&erQueue);
    displayQueue(&erQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
```

```
    q->front = NULL;
}

void enqueue(struct Queue* q, int id, const char* name, int severity)
{
    struct Patient* newPatient = (struct Patient*)malloc(sizeof(struct Patient));
    newPatient->id = id;
    strcpy(newPatient->name, name);
    newPatient->severity = severity;
    newPatient->next = NULL;
    if (q->front == NULL || severity > q->front->severity)
    {
        newPatient->next = q->front;
        q->front = newPatient;
    }
    else
    {
        struct Patient* temp = q->front;
        while (temp->next != NULL && temp->next->severity >= severity)
            temp = temp->next;
        newPatient->next = temp->next;
        temp->next = newPatient;
    }
    printf("Patient ID %d (%s, Severity: %d) added\n", id, name, severity);
}

void dequeue(struct Queue* q)
```

```
{
    if (q->front == NULL)
    {
        printf("No patients to process\n");
        return;
    }
    struct Patient* temp = q->front;
    printf("Processing Patient ID %d (%s, Severity: %d)\n", temp->id,
                                                    temp->name, temp->severity);
    q->front = q->front->next;
    free(temp);
}

void displayQueue(struct Queue* q)
{
    struct Patient* temp = q->front;
    printf("Patients in queue:\n");
    while (temp != NULL)
    {
        printf("ID: %d, Name: %s, Severity: %d\n", temp->id, temp->name,
                                                        temp->severity);
        temp = temp->next;
    }
}
```

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 <string.h>

#define MAX_SURVEYORS 50

struct Surveyor
{
   int id;
   char name[50];
   int collectedData;
};

struct Queue
{
   struct Surveyor data[MAX_SURVEYORS];
   int front, rear;
};

void initializeQueue(struct Queue* q);
int isQueueFull(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int id, const char* name, int collectedData);
void dequeue(struct Queue* q);
```

```
void displayQueue(struct Queue* q);

int main()
{
    struct Queue surveyQueue;
    initializeQueue(&surveyQueue);
    enqueue(&surveyQueue, 1, "PQR", 10);
    enqueue(&surveyQueue, 2, "MNO", 15);
    enqueue(&surveyQueue, 3, "DEF", 8);
    displayQueue(&surveyQueue);
    dequeue(&surveyQueue);
    displayQueue(&surveyQueue);
    return 0;
}

void initializeQueue(struct Queue* q)
{
    q->front = q->rear = -1;
}

int isQueueFull(struct Queue* q)
{
    return (q->rear + 1) % MAX_SURVEYORS == q->front;
}

int isQueueEmpty(struct Queue* q)
{
```

```
    return q->front == -1;
}

void enqueue(struct Queue* q, int id, const char* name, int collectedData)
{
    if (isQueueFull(q))
    {
        printf("Queue is full\n");
        return;
    }
    if (q->front == -1) q->front = 0;
    q->rear = (q->rear + 1) % MAX_SURVEYORS;
    q->data[q->rear].id = id;
    strcpy(q->data[q->rear].name, name);
    q->data[q->rear].collectedData = collectedData;
    printf("Surveyor ID %d (%s) added with %d surveys collected\n", id, name,
                                                        collectedData);
}

void dequeue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("No surveyors to process\n");
        return;
    }
    struct Surveyor surveyor = q->data[q->front];
```

```
    printf("Processing Surveyor ID %d (%s, Data Collected: %d)\n", surveyor.id,
                                        surveyor.name, surveyor.collectedData);
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front = (q->front + 1) % MAX_SURVEYORS;
}

void displayQueue(struct Queue* q)
{
    if (isQueueEmpty(q))
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Surveyors in queue:\n");
    for (int i = q->front; i != q->rear; i = (i + 1) % MAX_SURVEYORS)
        printf("ID: %d, Name: %s, Collected Data: %d\n", q->data[i].id,
                                    q->data[i].name, q->data[i].collectedData);
    printf("ID: %d, Name: %s, Collected Data: %d\n", q->data[q->rear].id,
                q->data[q->rear].name,  q->data[q->rear].collectedData);
}
```

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>

struct MarketData
{
    int id;
    float price;
    int volume;
    struct MarketData* next;
};

struct Queue
{
    struct MarketData* front;
    struct MarketData* rear;
};

void initializeQueue(struct Queue* q);
void enqueue(struct Queue* q, int id, float price, int volume);
void dequeue(struct Queue* q);
void displayQueue(struct Queue* q);

int main()
{
    struct Queue marketQueue;
    initializeQueue(&marketQueue);
    enqueue(&marketQueue, 1, 150.5, 1000);
```

```
    enqueue(&marketQueue, 2, 200.75, 500);
    enqueue(&marketQueue, 3, 175.25, 700);
    displayQueue(&marketQueue);
    dequeue(&marketQueue);
    displayQueue(&marketQueue);
    return 0;
}

void initializeQueue(struct Queue* q) {
    q->front = q->rear = NULL;
}

void enqueue(struct Queue* q, int id, float price, int volume) {
    struct MarketData* newData = (struct MarketData*)malloc
                                        (sizeof(struct MarketData));
    newData->id = id;
    newData->price = price;
    newData->volume = volume;
    newData->next = NULL;
    if (q->rear == NULL)
        q->front = q->rear = newData;
    else {
        q->rear->next = newData;
        q->rear = newData;
    }
    printf("Market Data ID %d added (Price: %.2f, Volume: %d)\n", id,
                                                price, volume);
```

```
}

void dequeue(struct Queue* q)
{
    if (q->front == NULL) {
        printf("No data to process\n");
        return;
    }
    struct MarketData* temp = q->front;
    printf("Processing Data ID %d (Price: %.2f, Volume: %d)\n", temp->id,
                                                temp->price, temp->volume);
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
}

void displayQueue(struct Queue* q)
{
    struct MarketData* temp = q->front;
    printf("Market Data Queue:\n");
    while (temp != NULL) {
        printf("ID: %d, Price: %.2f, Volume: %d\n", temp->id, temp->price,
                                                        temp->volume);
        temp = temp->next;
    }
}
```