```
/*****************************************************************************
****
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 100


typedef struct {
    int orderID;
    char type[4];
    float price;
    int quantity;
} Order;

typedef struct {
    Order orders[MAX];
    int front;
```

```
    int rear;
} Queue;


void initializeQueue(Queue *q);
void enqueueOrder(Queue *q, int orderID, char *type, float price, int quantity);
Order dequeueOrder(Queue *q);
void displayQueue(Queue *q);
void matchOrders(Queue *buyQueue, Queue *sellQueue);

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


    enqueueOrder(&buyQueue, 1, "BUY", 100.5, 10);
    enqueueOrder(&sellQueue, 2, "SELL", 100.5, 5);
    enqueueOrder(&buyQueue, 3, "BUY", 101.0, 20);
    enqueueOrder(&sellQueue, 4, "SELL", 101.0, 15);


    printf("Buy Orders:\n");
    displayQueue(&buyQueue);

    printf("Sell Orders:\n");
    displayQueue(&sellQueue);
```

```
    matchOrders(&buyQueue, &sellQueue);

    printf("Buy Orders after matching:\n");
    displayQueue(&buyQueue);

    printf("Sell Orders after matching:\n");
    displayQueue(&sellQueue);

    return 0;
}

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

void enqueueOrder(Queue *q, int orderID, char *type, float price, int quantity) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add order.\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;
```

```
    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;
}


Order dequeueOrder(Queue *q) {
    Order order = {0, "", 0.0, 0};

    if (q->front == -1) {
        printf("Queue is empty.\n");
        return order;
    }

    order = q->orders[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return order;
}
```

```
void displayQueue(Queue *q) {
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
        printf("OrderID=%d, Type=%s, Price=%.2f, Quantity=%d\n",
               q->orders[i].orderID, q->orders[i].type, q->orders[i].price, q->orders[i].quantity);
    }
    printf("\n");
}


void matchOrders(Queue *buyQueue, Queue *sellQueue) {
    while (buyQueue->front != -1 && sellQueue->front != -1) {
        Order buyOrder = buyQueue->orders[buyQueue->front];
        Order sellOrder = sellQueue->orders[sellQueue->front];

        if (buyOrder.price >= sellOrder.price) {
            int matchQuantity = (buyOrder.quantity < sellOrder.quantity) ? buyOrder.quantity :
sellOrder.quantity;
            printf("Matched Order: BuyOrderID=%d, SellOrderID=%d, Price=%.2f,
Quantity=%d\n",
                   buyOrder.orderID, sellOrder.orderID, sellOrder.price, matchQuantity);

            buyQueue->orders[buyQueue->front].quantity -= matchQuantity;
            sellQueue->orders[sellQueue->front].quantity -= matchQuantity;
```

```
        if (buyQueue->orders[buyQueue->front].quantity == 0)
            dequeueOrder(buyQueue);

        if (sellQueue->orders[sellQueue->front].quantity == 0)
            dequeueOrder(sellQueue);
      } else {
        break;
      }
   }
}
```

```
/**********************************************************************
****
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>
```

```
typedef struct {
    int customerID;
    char name[50];
    int priority;

} Customer;



struct Node {
    Customer data;
    struct Node *next;
} *front = NULL, *rear = NULL;



void enqueueCustomer(int customerID, char *name, int priority);
Customer dequeueCustomer();
void displayQueue();

int main() {

    enqueueCustomer(1, "Alice", 2);
    enqueueCustomer(2, "Bob", 3);
    enqueueCustomer(3, "Charlie", 1);



    printf("Current queue of customers:\n");
    displayQueue();
```

```
    Customer serviced = dequeueCustomer();
    if (serviced.priority != -1) {
        printf("Serviced customer: CustomerID=%d, Name=%s, Priority=%d\n",
            serviced.customerID, serviced.name, serviced.priority);
    }

    printf("Queue after servicing a customer:\n");
    displayQueue();

    return 0;
}

void enqueueCustomer(int customerID, char *name, int priority) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.customerID = customerID;
    strcpy(t->data.name, name);
    t->data.priority = priority;
    t->next = NULL;
```

```
    if (front == NULL || front->data.priority < priority) {
        t->next = front;
        front = t;
        if (rear == NULL) {
            rear = t;
        }
    } else {
        p = front;
        while (p != NULL && p->data.priority >= priority) {
            q = p;
            p = p->next;
        }
        t->next = q->next;
        q->next = t;
        if (q == rear) {
            rear = t;
        }
    }
}



Customer dequeueCustomer() {
    Customer customer = {-1, "", -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return customer;
    } else {
```

```
        customer = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return customer;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("CustomerID=%d, Name=%s, Priority=%d -> ",
              p->data.customerID, p->data.name, p->data.priority);
        p = p->next;
    }
    printf("\n");
}
```

/**
****

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>

#define MAX 100

typedef struct {
    int attendeeID;
    char name[50];
    int priority;
} Attendee;

typedef struct {
    Attendee attendees[MAX];
    int front;
```

```
    int rear;
} Queue;



void initializeQueue(Queue *q);
void enqueueAttendee(Queue *q, int attendeeID, char *name, int priority);
Attendee dequeueAttendee(Queue *q);
void displayQueue(Queue *q);

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



    enqueueAttendee(&queue, 1, "Alice", 1);
    enqueueAttendee(&queue, 2, "Bob", 2);
    enqueueAttendee(&queue, 3, "Charlie", 1);



    printf("Current queue of attendees:\n");
    displayQueue(&queue);



    Attendee checkedIn = dequeueAttendee(&queue);
    if (checkedIn.priority != -1) {
        printf("Checked-in attendee: AttendeeID=%d, Name=%s, Priority=%d\n",
               checkedIn.attendeeID, checkedIn.name, checkedIn.priority);
    }
```

```
    printf("Queue after check-in:\n");
    displayQueue(&queue);

    return 0;
}


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


void enqueueAttendee(Queue *q, int attendeeID, char *name, int priority) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add attendee.\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;

    int i;
    for (i = q->rear; i >= q->front && q->attendees[i].priority < priority; i--) {
        q->attendees[i + 1] = q->attendees[i];
    }
```

```
    q->attendees[i + 1].attendeeID = attendeeID;
    strcpy(q->attendees[i + 1].name, name);
    q->attendees[i + 1].priority = priority;
    q->rear++;
}


Attendee dequeueAttendee(Queue *q) {
    Attendee attendee = {-1, "", -1};

    if (q->front == -1) {
        printf("Queue is already empty \n");
        return attendee;
    }

    attendee = q->attendees[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return attendee;
}


void displayQueue(Queue *q) {
```

```
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
        printf("AttendeeID=%d, Name=%s, Priority=%d\n",
               q->attendees[i].attendeeID, q->attendees[i].name, q->attendees[i].priority);
    }
    printf("\n");
}

/**************************************************************************
****
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>
#include <time.h>


typedef struct {
```

```
    int customerID;
    int transactionTime;
} Customer;



struct Node {
    Customer data;
    struct Node *next;
} *front = NULL, *rear = NULL;



void enqueueCustomer(int customerID, int transactionTime);
Customer dequeueCustomer();
int isEmpty();
void simulateTeller(int tellerID);
void displayQueue();

int main() {
    srand(time(NULL));



    for (int i = 1; i <= 10; i++) {
        int transactionTime = rand() % 10 + 1;
        enqueueCustomer(i, transactionTime);
    }



    printf("Current queue of customers:\n");
```

```
    displayQueue();



    int numberOfTellers = 3;
    for (int i = 1; i <= numberOfTellers; i++) {
        simulateTeller(i);
    }



    printf("Queue after processing:\n");
    displayQueue();

    return 0;
}



void enqueueCustomer(int customerID, int transactionTime) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.customerID = customerID;
    t->data.transactionTime = transactionTime;
    t->next = NULL;
```

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


Customer dequeueCustomer() {
    Customer customer = {-1, -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return customer;
    } else {
        customer = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return customer;
}
```

```
int isEmpty() {
    return front == NULL;
}



void simulateTeller(int tellerID) {
    printf("Teller %d starting to serve customers...\n", tellerID);
    while (!isEmpty()) {
        Customer customer = dequeueCustomer();
        if (customer.customerID != -1) {
            printf("Teller %d is serving customer %d with transaction time %d\n",
                tellerID, customer.customerID, customer.transactionTime);


            for (int i = 0; i < customer.transactionTime; i++) {
                printf(".");
                fflush(stdout);
                sleep(1);
            }
            printf("\nTeller %d finished serving customer %d\n", tellerID, customer.customerID);
        }
    }
}



void displayQueue() {
    struct Node *p = front;
    while (p) {
```

```
        printf("CustomerID=%d, TransactionTime=%d -> ", p->data.customerID, p->data.transactionTime);
        p = p->next;
    }
    printf("\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 100

typedef struct {
    int instrumentID;
    char timestamp[20];
```

```
    float price;
    int volume;
} DataFeed;



typedef struct {
    DataFeed feeds[MAX];
    int front;
    int rear;
} Queue;



void initializeQueue(Queue *q);
void enqueueDataFeed(Queue *q, int instrumentID, char *timestamp, float price, int volume);
DataFeed dequeueDataFeed(Queue *q);
void displayQueue(Queue *q);

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



    enqueueDataFeed(&queue, 1, "2025-01-20 10:00:00", 100.5, 1000);
    enqueueDataFeed(&queue, 2, "2025-01-20 10:00:01", 101.0, 2000);
    enqueueDataFeed(&queue, 1, "2025-01-20 10:00:02", 100.8, 1500);
    enqueueDataFeed(&queue, 2, "2025-01-20 10:00:03", 101.2, 2500);
```

```
    printf("Current queue of data feeds:\n");
    displayQueue(&queue);



    DataFeed processed;
    while ((processed = dequeueDataFeed(&queue)).instrumentID != -1) {
        printf("Processed data feed: InstrumentID=%d, Timestamp=%s, Price=%.2f,
Volume=%d\n",
               processed.instrumentID, processed.timestamp, processed.price, processed.volume);
    }


    return 0;
}



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



void enqueueDataFeed(Queue *q, int instrumentID, char *timestamp, float price, int
volume) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add data feed.\n");
        return;
    }
```

```
    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->feeds[q->rear].instrumentID = instrumentID;
    strcpy(q->feeds[q->rear].timestamp, timestamp);
    q->feeds[q->rear].price = price;
    q->feeds[q->rear].volume = volume;
}


DataFeed dequeueDataFeed(Queue *q) {
    DataFeed feed = {-1, "", 0.0, 0};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return feed;
    }

    feed = q->feeds[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return feed;
}
```

```
void displayQueue(Queue *q) {
   if (q->front == -1) {
      printf("Queue is empty.\n");
      return;
   }

   for (int i = q->front; i <= q->rear; i++) {
      printf("InstrumentID=%d, Timestamp=%s, Price=%.2f, Volume=%d\n",
           q->feeds[i].instrumentID, q->feeds[i].timestamp, q->feeds[i].price, q->feeds[i].volume);
   }
   printf("\n");
}

/**
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>

// Define the structure for a car
```

```
typedef struct {
    int carID;
    int arrivalTime;
} Car;

// Define the structure for the queue node
struct Node {
    Car data;
    struct Node *next;
} *front = NULL, *rear = NULL;

// Function prototypes
void enqueueCar(int carID, int arrivalTime);
Car dequeueCar();
int isEmpty();
void simulateTrafficLight();
void displayQueue();

int main() {
    // Simulate cars arriving at different times
    enqueueCar(1, 0);  // Car 1 arrives at time 0
    sleep(1);  // Sleep for 1 second to simulate time passing
    enqueueCar(2, 1);  // Car 2 arrives at time 1
    sleep(2);  // Sleep for 2 seconds to simulate time passing
    enqueueCar(3, 3);  // Car 3 arrives at time 3
    sleep(1);  // Sleep for 1 second to simulate time passing
    enqueueCar(4, 4);  // Car 4 arrives at time 4
```

```
    // Display the queue
    printf("Current queue of cars:\n");
    displayQueue();

    // Simulate traffic light control
    simulateTrafficLight();

    // Display the queue after processing
    printf("Queue after processing:\n");
    displayQueue();

    return 0;
}

// Function to add a car to the queue (enqueue)
void enqueueCar(int carID, int arrivalTime) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.carID = carID;
    t->data.arrivalTime = arrivalTime;
    t->next = NULL;

    if (front == NULL) {
```

```
        front = rear = t;
    } else {
        rear->next = t;
        rear = t;
    }
}

// Function to remove (dequeue) the oldest car from the queue
Car dequeueCar() {
    Car car = {-1, -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return car;
    } else {
        car = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return car;
}

// Function to check if the queue is empty
int isEmpty() {
```

```
    return front == NULL;
}

// Function to simulate the traffic light control
void simulateTrafficLight() {
    int greenLightDuration = 5;  // Time duration for green light
    int elapsedTime = 0;

    while (!isEmpty() && elapsedTime < greenLightDuration) {
        Car car = dequeueCar();
        if (car.carID != -1) {
            printf("Car %d is passing the traffic light (arrival time: %d)\n", car.carID,
car.arrivalTime);
            elapsedTime += 1;  // Simulate 1 second for each car passing
            sleep(1);  // Sleep for 1 second to simulate car passing time
        }
    }
    printf("Traffic light is now red. No more cars can pass.\n");
}

// Function to display the current queue of cars
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("CarID=%d, ArrivalTime=%d -> ", p->data.carID, p->data.arrivalTime);
        p = p->next;
    }
    printf("\n");
```

```
}

/**********************************************************************
****

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>

#define MAX 100

typedef struct {
   int pollingStationID;
   int voteID;
} Vote;

typedef struct {
```

```
    Vote votes[MAX];
    int front;
    int rear;
} Queue;


void initializeQueue(Queue *q);
void enqueueVote(Queue *q, int pollingStationID, int voteID);
Vote dequeueVote(Queue *q);
void displayQueue(Queue *q);

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


    enqueueVote(&queue, 1, 101);
    enqueueVote(&queue, 2, 102);
    enqueueVote(&queue, 1, 103);
    enqueueVote(&queue, 3, 104);
    enqueueVote(&queue, 2, 105);


    printf("Current queue of votes:\n");
    displayQueue(&queue);


    Vote counted;
```

```
    while ((counted = dequeueVote(&queue)).voteID != -1) {
        printf("Counted vote: PollingStationID=%d, VoteID=%d\n",
               counted.pollingStationID, counted.voteID);
    }

    return 0;
}



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



void enqueueVote(Queue *q, int pollingStationID, int voteID) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add vote.\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->votes[q->rear].pollingStationID = pollingStationID;
    q->votes[q->rear].voteID = voteID;
}
```

```
Vote dequeueVote(Queue *q) {
    Vote vote = {-1, -1};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return vote;
    }

    vote = q->votes[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return vote;
}


void displayQueue(Queue *q) {
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
```

```
        printf("PollingStationID=%d, VoteID=%d\n",
            q->votes[i].pollingStationID, q->votes[i].voteID);
    }
    printf("\n");
}
```

```
/*************************************************************************
****
```

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 struct {
    int planeID;
    char type[10];
    int priority;
} Airplane;
```

```
struct Node {
    Airplane data;
    struct Node *next;
} *front = NULL, *rear = NULL;


void enqueueAirplane(int planeID, char *type, int priority);
Airplane dequeueAirplane();
int isEmpty();
void simulateRunwayAllocation();
void displayQueue();

int main() {

    enqueueAirplane(1, "LAND", 1);       // Regular landing
    enqueueAirplane(2, "TAKEOFF", 2);    // Regular takeoff
    enqueueAirplane(3, "LAND", 3);       // Emergency landing
    enqueueAirplane(4, "LAND", 1);       // Regular landing
    enqueueAirplane(5, "TAKEOFF", 2);    // Regular takeoff

    // Display the queue
    printf("Current queue of airplanes:\n");
    displayQueue();

    // Simulate runway allocation
    simulateRunwayAllocation();
```

```
    // Display the queue after processing
    printf("Queue after processing:\n");
    displayQueue();

    return 0;
}

// Function to add an airplane to the queue (enqueue) based on priority
void enqueueAirplane(int planeID, char *type, int priority) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.planeID = planeID;
    strcpy(t->data.type, type);
    t->data.priority = priority;
    t->next = NULL;

    if (front == NULL || front->data.priority < priority) {
        t->next = front;
        front = t;
        if (rear == NULL) {
            rear = t;
        }
```

```
    } else {
        p = front;
        while (p != NULL && p->data.priority >= priority) {
            q = p;
            p = p->next;
        }
        t->next = q->next;
        q->next = t;
        if (q == rear) {
            rear = t;
        }
    }
}

// Function to remove (dequeue) the highest-priority airplane from the queue
Airplane dequeueAirplane() {
    Airplane airplane = {-1, "", -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return airplane;
    } else {
        airplane = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
```

```
        }
    }
    return airplane;
}

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

// Function to simulate runway allocation
void simulateRunwayAllocation() {
    printf("Simulating runway allocation...\n");
    while (!isEmpty()) {
        Airplane airplane = dequeueAirplane();
        if (airplane.planeID != -1) {
            printf("Allocating runway to PlaneID=%d, Type=%s, Priority=%d\n",
                   airplane.planeID, airplane.type, airplane.priority);
            // Simulate the time for landing or takeoff (for demonstration purposes, we'll just print a message)
        }
    }
}

// Function to display the current queue of airplanes
void displayQueue() {
    struct Node *p = front;
    while (p) {
```

```
        printf("PlaneID=%d, Type=%s, Priority=%d -> ", p->data.planeID, p->data.type, p->data.priority);
        p = p->next;
    }
    printf("\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 100

typedef struct {
    int orderID;
    char type[5];
```

```
    float price;
    int quantity;
    int isCancelled;
} Order;


typedef struct {
    Order orders[MAX];
    int front;
    int rear;
} Queue;


void initializeQueue(Queue *q);
void enqueueOrder(Queue *q, int orderID, char *type, float price, int quantity);
Order dequeueOrder(Queue *q);
void cancelOrder(Queue *q, int orderID);
void displayQueue(Queue *q);

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


    enqueueOrder(&queue, 1, "BUY", 100.5, 10);
    enqueueOrder(&queue, 2, "SELL", 101.0, 20);
    enqueueOrder(&queue, 3, "BUY", 100.8, 15);
    enqueueOrder(&queue, 4, "SELL", 101.2, 25);
```

```
    printf("Current queue of trading orders:\n");
    displayQueue(&queue);

    cancelOrder(&queue, 2);

    printf("Queue after cancellation of order 2:\n");
    displayQueue(&queue);

    Order processed;
    while ((processed = dequeueOrder(&queue)).orderID != -1) {
        if (!processed.isCancelled) {
            printf("Processed order: OrderID=%d, Type=%s, Price=%.2f, Quantity=%d\n",
                processed.orderID, processed.type, processed.price, processed.quantity);
        }
    }

    return 0;
}

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

```
}

void enqueueOrder(Queue *q, int orderID, char *type, float price, int quantity) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add order.\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;

    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;
    q->orders[q->rear].isCancelled = 0;
}

Order dequeueOrder(Queue *q) {
    Order order = {-1, "", 0.0, 0, 0};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return order;
    }
```

```
    order = q->orders[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return order;
}


void cancelOrder(Queue *q, int orderID) {
    if (q->front == -1) {
        printf("Queue is empty. Cannot cancel order.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
        if (q->orders[i].orderID == orderID) {
            q->orders[i].isCancelled = 1;
            printf("Order %d is cancelled.\n", orderID);
            return;
        }
    }

    printf("Order %d not found.\n", orderID);
}
```

```
void displayQueue(Queue *q) {
   if (q->front == -1) {
      printf("Queue is empty.\n");
      return;
   }

   for (int i = q->front; i <= q->rear; i++) {
      printf("OrderID=%d, Type=%s, Price=%.2f, Quantity=%d, Cancelled=%d\n",
          q->orders[i].orderID, q->orders[i].type, q->orders[i].price, q->orders[i].quantity, q->orders[i].isCancelled);
   }
   printf("\n");
}

/*************************************************************************
****

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 {
    int registrationID;
    char name[50];
    char registrationType[15];
} Registration;


struct Node {
    Registration data;
    struct Node *next;
} *front = NULL, *rear = NULL;


void enqueueRegistration(int registrationID, char *name, char *registrationType);
Registration dequeueRegistration();
void cancelRegistration(int registrationID);
void displayQueue();

int main() {

    enqueueRegistration(1, "Alice", "WALK-IN");
    enqueueRegistration(2, "Bob", "PRE-REGISTRATION");
    enqueueRegistration(3, "Charlie", "WALK-IN");
    enqueueRegistration(4, "David", "PRE-REGISTRATION");
```

```
    printf("Current queue of registrations:\n");
    displayQueue();



    cancelRegistration(2);



    printf("Queue after cancellation of registration 2:\n");
    displayQueue();



    Registration processed;
    while ((processed = dequeueRegistration()).registrationID != -1) {
        printf("Processed registration: RegistrationID=%d, Name=%s, RegistrationType=%s\n",
                processed.registrationID, processed.name, processed.registrationType);
    }

    return 0;
}



void enqueueRegistration(int registrationID, char *name, char *registrationType) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }
```

```
    t->data.registrationID = registrationID;
    strcpy(t->data.name, name);
    strcpy(t->data.registrationType, registrationType);
    t->next = NULL;

    if (front == NULL) {
        front = rear = t;
    } else {
        rear->next = t;
        rear = t;
    }
}

// Function to remove  the oldest registration from the queue
Registration dequeueRegistration() {
    Registration registration = {-1, "", ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return registration;
    } else {
        registration = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
```

```
        }
    }
    return registration;
}

// Function to cancel a registration by registrationID
void cancelRegistration(int registrationID) {
    struct Node *p = front, *prev = NULL;
    while (p != NULL && p->data.registrationID != registrationID) {
        prev = p;
        p = p->next;
    }

    if (p == NULL) {
        printf("Registration %d not found.\n", registrationID);
        return;
    }

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

    if (p == rear) {
        rear = prev;
    }
```

```
    free(p);
    printf("Registration %d is cancelled.\n", registrationID);
}

// Function to display the current queue of registrations
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("RegistrationID=%d, Name=%s, RegistrationType=%s -> ",
                p->data.registrationID, p->data.name, p->data.registrationType);
        p = p->next;
    }
    printf("\n");
}

/**********************************************************************
****
```

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>
```

```
#include <string.h>

// Define the maximum size of the queue
#define MAX 100

// Define the structure for an audience member
typedef struct {
    int memberID;
    char name[50];
    char type[15];  // "GENERAL" or "MEDIA"
} AudienceMember;

// Define the structure for the queue
typedef struct {
    AudienceMember members[MAX];
    int front;
    int rear;
} Queue;

// Function prototypes
void initializeQueue(Queue *q);
void enqueueMember(Queue *q, int memberID, char *name, char *type);
AudienceMember dequeueMember(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue queue;
    initializeQueue(&queue);
```

```
    // Simulate audience entry
    enqueueMember(&queue, 1, "Alice", "GENERAL");
    enqueueMember(&queue, 2, "Bob", "MEDIA");
    enqueueMember(&queue, 3, "Charlie", "GENERAL");
    enqueueMember(&queue, 4, "David", "MEDIA");

    // Display the queue
    printf("Current queue of audience members:\n");
    displayQueue(&queue);

    // Dequeue and seat audience members
    AudienceMember seated;
    while ((seated = dequeueMember(&queue)).memberID != -1) {
        printf("Seated audience member: MemberID=%d, Name=%s, Type=%s\n",
               seated.memberID, seated.name, seated.type);
    }

    return 0;
}

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

// Function to add an audience member to the queue (enqueue)
```

```
void enqueueMember(Queue *q, int memberID, char *name, char *type) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add member.\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->members[q->rear].memberID = memberID;
    strcpy(q->members[q->rear].name, name);
    strcpy(q->members[q->rear].type, type);
}

// Function to remove (dequeue) the oldest audience member from the queue
AudienceMember dequeueMember(Queue *q) {
    AudienceMember member = {-1, "", ""};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return member;
    }

    member = q->members[q->front];
    q->front++;

    if (q->front > q->rear) {
```

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

    return member;
}

// Function to display the current queue of audience members
void displayQueue(Queue *q) {
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
        printf("MemberID=%d, Name=%s, Type=%s\n",
               q->members[i].memberID, q->members[i].name, q->members[i].type);
    }
    printf("\n");
}
```

/**
****

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>

// Define the structure for a loan application
typedef struct {
    int applicationID;
    char applicantName[50];
    float loanAmount;
    int creditScore;
} LoanApplication;

// Define the structure for the queue node
struct Node {
    LoanApplication data;
    struct Node *next;
} *front = NULL, *rear = NULL;
```

```
// Function prototypes
void enqueueApplication(int applicationID, char *applicantName, float loanAmount, int creditScore);
LoanApplication dequeueApplication();
void displayQueue();

int main() {
    // Add sample loan applications to the queue
    enqueueApplication(1, "Alice", 50000, 750);
    enqueueApplication(2, "Bob", 100000, 800);
    enqueueApplication(3, "Charlie", 75000, 700);
    enqueueApplication(4, "David", 150000, 680);

    // Display the queue
    printf("Current queue of loan applications:\n");
    displayQueue();

    // Dequeue and process loan applications
    LoanApplication processed;
    while ((processed = dequeueApplication()).applicationID != -1) {
        printf("Processed loan application: ApplicationID=%d, ApplicantName=%s, LoanAmount=%.2f, CreditScore=%d\n",
               processed.applicationID, processed.applicantName, processed.loanAmount, processed.creditScore);
    }

    return 0;
}
```

```
// Function to add a loan application to the queue (enqueue) based on priority
void enqueueApplication(int applicationID, char *applicantName, float loanAmount, int creditScore) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.applicationID = applicationID;
    strcpy(t->data.applicantName, applicantName);
    t->data.loanAmount = loanAmount;
    t->data.creditScore = creditScore;
    t->next = NULL;

    if (front == NULL || (loanAmount > front->data.loanAmount) ||
       (loanAmount == front->data.loanAmount && creditScore > front->data.creditScore)) {
        t->next = front;
        front = t;
        if (rear == NULL) {
            rear = t;
        }
    } else {
        p = front;
        while (p != NULL && ((loanAmount < p->data.loanAmount) ||
            (loanAmount == p->data.loanAmount && creditScore <= p->data.creditScore))) {
            q = p;
```

```
            p = p->next;
        }
        t->next = q->next;
        q->next = t;
        if (q == rear) {
            rear = t;
        }
    }
}

// Function to remove (dequeue) the highest-priority loan application from the queue
LoanApplication dequeueApplication() {
    LoanApplication application = {-1, "", 0.0, 0};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return application;
    } else {
        application = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return application;
}
```

```
// Function to display the current queue of loan applications
void displayQueue() {
   struct Node *p = front;
   while (p) {
      printf("ApplicationID=%d, ApplicantName=%s, LoanAmount=%.2f, CreditScore=%d -> ",
          p->data.applicationID, p->data.applicantName, p->data.loanAmount, p->data.creditScore);
      p = p->next;
   }
   printf("\n");
}
```

/**********

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>

// Define the maximum size of the queue
```

```
#define MAX 100

// Define the structure for a checkout
typedef struct {
    int customerID;
    char items[100][50]; // List of items with a maximum of 100 items
    int itemCount; // Number of items
} Checkout;

// Define the structure for the queue
typedef struct {
    Checkout checkouts[MAX];
    int front;
    int rear;
} Queue;

// Define the structure for inventory
typedef struct {
    char itemName[50];
    int quantity;
} Inventory;

// Function prototypes
void initializeQueue(Queue *q);
void enqueueCheckout(Queue *q, int customerID, char items[][50], int itemCount);
Checkout dequeueCheckout(Queue *q);
void displayQueue(Queue *q);
void updateInventory(Inventory *inventory, int inventorySize, Checkout checkout);
```

```
void displayInventory(Inventory *inventory, int inventorySize);

int main() {
    Queue queue;
    Inventory inventory[] = {
        {"Item1", 50},
        {"Item2", 100},
        {"Item3", 75},
        {"Item4", 200}
    };
    int inventorySize = 4;

    initializeQueue(&queue);

    // Simulate customers checking out
    char items1[][50] = {"Item1", "Item2"};
    enqueueCheckout(&queue, 1, items1, 2);

    char items2[][50] = {"Item3", "Item4", "Item1"};
    enqueueCheckout(&queue, 2, items2, 3);

    char items3[][50] = {"Item2", "Item3"};
    enqueueCheckout(&queue, 3, items3, 2);

    // Display the queue
    printf("Current queue of checkouts:\n");
    displayQueue(&queue);
```

```
    // Display the inventory before processing
    printf("Inventory before processing checkouts:\n");
    displayInventory(inventory, inventorySize);

    // Dequeue and process checkouts
    Checkout processed;
    while ((processed = dequeueCheckout(&queue)).customerID != -1) {
        printf("Processed checkout for CustomerID=%d\n", processed.customerID);
        updateInventory(inventory, inventorySize, processed);
    }

    // Display the inventory after processing
    printf("Inventory after processing checkouts:\n");
    displayInventory(inventory, inventorySize);

    return 0;
}

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

// Function to add a checkout to the queue (enqueue)
void enqueueCheckout(Queue *q, int customerID, char items[][50], int itemCount) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add checkout.\n");
```

```
        return;
    }

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->checkouts[q->rear].customerID = customerID;
    q->checkouts[q->rear].itemCount = itemCount;
    for (int i = 0; i < itemCount; i++) {
        strcpy(q->checkouts[q->rear].items[i], items[i]);
    }
}

// Function to remove (dequeue) the oldest checkout from the queue
Checkout dequeueCheckout(Queue *q) {
    Checkout checkout = {-1, {""}, 0};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return checkout;
    }

    checkout = q->checkouts[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
```

```
    }

    return checkout;
}

// Function to display the current queue of checkouts
void displayQueue(Queue *q) {
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
        printf("CustomerID=%d, ItemCount=%d, Items=", q->checkouts[i].customerID, q->checkouts[i].itemCount);
        for (int j = 0; j < q->checkouts[i].itemCount; j++) {
            printf("%s ", q->checkouts[i].items[j]);
        }
        printf("\n");
    }
    printf("\n");
}

// Function to update inventory based on processed checkout
void updateInventory(Inventory *inventory, int inventorySize, Checkout checkout) {
    for (int i = 0; i < checkout.itemCount; i++) {
        for (int j = 0; j < inventorySize; j++) {
            if (strcmp(inventory[j].itemName, checkout.items[i]) == 0) {
```

```
            if (inventory[j].quantity > 0) {
                inventory[j].quantity--;
            } else {
                printf("Item %s is out of stock!\n", inventory[j].itemName);
            }
            break;
        }
    }
  }
}

// Function to display the current inventory
void displayInventory(Inventory *inventory, int inventorySize) {
  for (int i = 0; i < inventorySize; i++) {
    printf("ItemName=%s, Quantity=%d\n", inventory[i].itemName, inventory[i].quantity);
  }
  printf("\n");
}
/**********************************************************************
****
```

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 the structure for a bus
typedef struct {
    int busID;
    char type[10];  // "REGULAR" or "EXPRESS"
    int arrivalTime; // Time of arrival in minutes since the start of the day
} Bus;

// Define the structure for the queue node
struct Node {
    Bus data;
    struct Node *next;
} *front = NULL, *rear = NULL;

// Function prototypes
void enqueueBus(int busID, char *type, int arrivalTime);
Bus dequeueBus();
int isEmpty();
void simulateTerminalOperations();
void displayQueue();

int main() {
    // Simulate bus arrivals at different times
    enqueueBus(1, "REGULAR", 480);  // Bus 1 arrives at 08:00 AM (480 minutes)
    enqueueBus(2, "EXPRESS", 485);  // Bus 2 arrives at 08:05 AM (485 minutes)
    enqueueBus(3, "REGULAR", 490);  // Bus 3 arrives at 08:10 AM (490 minutes)
```

```
    enqueueBus(4, "EXPRESS", 495);  // Bus 4 arrives at 08:15 AM (495 minutes)
    enqueueBus(5, "REGULAR", 500);  // Bus 5 arrives at 08:20 AM (500 minutes)

    // Display the queue
    printf("Current queue of buses:\n");
    displayQueue();

    // Simulate terminal operations
    simulateTerminalOperations();

    // Display the queue after processing
    printf("Queue after processing:\n");
    displayQueue();

    return 0;
}

// Function to add a bus to the queue (enqueue) based on priority
void enqueueBus(int busID, char *type, int arrivalTime) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.busID = busID;
    strcpy(t->data.type, type);
```

```
    t->data.arrivalTime = arrivalTime;
    t->next = NULL;

    // Priority: EXPRESS buses have higher priority
    if (front == NULL || (strcmp(type, "EXPRESS") == 0 && strcmp(front->data.type,
"REGULAR") == 0)) {
        t->next = front;
        front = t;
        if (rear == NULL) {
            rear = t;
        }
    } else {
        p = front;
        while (p != NULL && strcmp(p->data.type, "EXPRESS") == 0) {
            q = p;
            p = p->next;
        }
        while (p != NULL && p->data.arrivalTime <= arrivalTime) {
            q = p;
            p = p->next;
        }
        t->next = q->next;
        q->next = t;
        if (q == rear) {
            rear = t;
        }
    }
}
```

```
// Function to remove (dequeue) the oldest bus from the queue
Bus dequeueBus() {
    Bus bus = {-1, "", -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return bus;
    } else {
        bus = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return bus;
}

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

// Function to simulate terminal operations
void simulateTerminalOperations() {
    printf("Simulating terminal operations...\n");
```

```
    while (!isEmpty()) {
        Bus bus = dequeueBus();
        if (bus.busID != -1) {
            printf("BusID=%d, Type=%s, ArrivalTime=%d minutes is departing the terminal.\n",
                bus.busID, bus.type, bus.arrivalTime);
            // Simulate the time for departure (for demonstration purposes, we'll just print a
message)
        }
    }
}

// Function to display the current queue of buses
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("BusID=%d, Type=%s, ArrivalTime=%d -> ", p->data.busID, p->data.type, p-
>data.arrivalTime);
        p = p->next;
    }
    printf("\n");
}
```

```
/*************************************************************************
****

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 the maximum size of the queue
#define MAX 100

// Define the structure for an audience member
typedef struct {
    int memberID;
    char name[50];
    char type[10];  // "GENERAL" or "VIP"
} AudienceMember;

// Define the structure for the queue
typedef struct {
    AudienceMember members[MAX];
    int front;
    int rear;
```

```
} Queue;

// Function prototypes
void initializeQueue(Queue *q);
void enqueueMember(Queue *q, int memberID, char *name, char *type);
AudienceMember dequeueMember(Queue *q);
void displayQueue(Queue *q);

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

    // Simulate audience entry
    enqueueMember(&queue, 1, "Alice", "GENERAL");
    enqueueMember(&queue, 2, "Bob", "VIP");
    enqueueMember(&queue, 3, "Charlie", "GENERAL");
    enqueueMember(&queue, 4, "David", "VIP");

    // Display the queue
    printf("Current queue of audience members:\n");
    displayQueue(&queue);

    // Dequeue and seat audience members
    AudienceMember seated;
    while ((seated = dequeueMember(&queue)).memberID != -1) {
        printf("Seated audience member: MemberID=%d, Name=%s, Type=%s\n",
               seated.memberID, seated.name, seated.type);
    }
```

```
    return 0;
}

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

// Function to add an audience member to the queue (enqueue)
void enqueueMember(Queue *q, int memberID, char *name, char *type) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add member.\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->members[q->rear].memberID = memberID;
    strcpy(q->members[q->rear].name, name);
    strcpy(q->members[q->rear].type, type);
}

// Function to remove (dequeue) the oldest audience member from the queue
AudienceMember dequeueMember(Queue *q) {
```

```
    AudienceMember member = {-1, "", ""};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return member;
    }

    member = q->members[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return member;
}

// Function to display the current queue of audience members
void displayQueue(Queue *q) {
    if (q->front == -1) {
        printf("Queue is empty.\n");
        return;
    }

    for (int i = q->front; i <= q->rear; i++) {
        printf("MemberID=%d, Name=%s, Type=%s\n",
               q->members[i].memberID, q->members[i].name, q->members[i].type);
    }
```

```
    printf("\n");
}
```

/**
****

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 the structure for a financial transaction
typedef struct {
    int transactionID;
    char type[10];  // "DEPOSIT", "WITHDRAWAL", or "TRANSFER"
    float amount;
    int fromAccountID;
    int toAccountID; // Applicable only for transfers
} Transaction;

// Define the structure for the queue node
struct Node {
    Transaction data;
```

```
    struct Node *next;
} *front = NULL, *rear = NULL;

// Function prototypes
void enqueueTransaction(int transactionID, char *type, float amount, int fromAccountID, int toAccountID);
Transaction dequeueTransaction();
int isEmpty();
void processTransactions();
void displayQueue();

int main() {
    // Add sample transactions to the queue
    enqueueTransaction(1, "DEPOSIT", 500.0, 101, -1);
    enqueueTransaction(2, "WITHDRAWAL", 200.0, 102, -1);
    enqueueTransaction(3, "TRANSFER", 300.0, 103, 104);
    enqueueTransaction(4, "DEPOSIT", 700.0, 105, -1);

    // Display the queue
    printf("Current queue of transactions:\n");
    displayQueue();

    // Process transactions
    processTransactions();

    // Display the queue after processing
    printf("Queue after processing:\n");
    displayQueue();
```

```
    return 0;
}

// Function to add a transaction to the queue (enqueue)
void enqueueTransaction(int transactionID, char *type, float amount, int fromAccountID, int toAccountID) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.transactionID = transactionID;
    strcpy(t->data.type, type);
    t->data.amount = amount;
    t->data.fromAccountID = fromAccountID;
    t->data.toAccountID = toAccountID;
    t->next = NULL;

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

```
// Function to remove (dequeue) the oldest transaction from the queue
Transaction dequeueTransaction() {
    Transaction transaction = {-1, "", 0.0, -1, -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return transaction;
    } else {
        transaction = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return transaction;
}

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

// Function to process transactions
void processTransactions() {
    printf("Processing transactions...\n");
```

```
    while (!isEmpty()) {
        Transaction transaction = dequeueTransaction();
        if (transaction.transactionID != -1) {
            if (strcmp(transaction.type, "DEPOSIT") == 0) {
                printf("Processed DEPOSIT of %.2f to account %d\n", transaction.amount,
transaction.fromAccountID);
            } else if (strcmp(transaction.type, "WITHDRAWAL") == 0) {
                printf("Processed WITHDRAWAL of %.2f from account %d\n", transaction.amount,
transaction.fromAccountID);
            } else if (strcmp(transaction.type, "TRANSFER") == 0) {
                printf("Processed TRANSFER of %.2f from account %d to account %d\n",
                    transaction.amount, transaction.fromAccountID, transaction.toAccountID);
            }
        }
    }
}

// Function to display the current queue of transactions
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("TransactionID=%d, Type=%s, Amount=%.2f, FromAccountID=%d,
ToAccountID=%d -> ",
            p->data.transactionID, p->data.type, p->data.amount, p->data.fromAccountID, p-
>data.toAccountID);
        p = p->next;
    }
    printf("\n");
}
```

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 the structure for a patient
typedef struct {
   int patientID;
```

```
    char name[50];
    int severity;  // Higher value indicates higher severity
} Patient;

// Define the structure for the queue node
struct Node {
    Patient data;
    struct Node *next;
} *front = NULL, *rear = NULL;

// Function prototypes
void enqueuePatient(int patientID, char *name, int severity);
Patient dequeuePatient();
int isEmpty();
void simulateDoctorsHandlingPatients(int numberOfDoctors);
void displayQueue();

int main() {
    // Add sample patients to the queue
    enqueuePatient(1, "Alice", 5);   // High severity
    enqueuePatient(2, "Bob", 3);     // Medium severity
    enqueuePatient(3, "Charlie", 4); // Medium-high severity
    enqueuePatient(4, "David", 1);   // Low severity
    enqueuePatient(5, "Eva", 2);     // Low-medium severity

    // Display the queue
    printf("Current queue of patients:\n");
    displayQueue();
```

```
    // Simulate multiple doctors handling patients
    int numberOfDoctors = 2;
    simulateDoctorsHandlingPatients(numberOfDoctors);

    // Display the queue after processing
    printf("Queue after processing:\n");
    displayQueue();

    return 0;
}

// Function to add a patient to the queue (enqueue) based on priority
void enqueuePatient(int patientID, char *name, int severity) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.patientID = patientID;
    strcpy(t->data.name, name);
    t->data.severity = severity;
    t->next = NULL;

    if (front == NULL || front->data.severity < severity) {
        t->next = front;
```

```
        front = t;
        if (rear == NULL) {
            rear = t;
        }
    } else {
        p = front;
        while (p != NULL && p->data.severity >= severity) {
            q = p;
            p = p->next;
        }
        t->next = q->next;
        q->next = t;
        if (q == rear) {
            rear = t;
        }
    }
}

// Function to remove (dequeue) the highest-priority patient from the queue
Patient dequeuePatient() {
    Patient patient = {-1, "", -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return patient;
    } else {
        patient = front->data;
        t = front;
```

```
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return patient;
}

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

// Function to simulate doctors handling patients
void simulateDoctorsHandlingPatients(int numberOfDoctors) {
    printf("Simulating doctors handling patients...\n");
    for (int i = 0; i < numberOfDoctors; i++) {
        printf("Doctor %d is starting to see patients...\n", i + 1);
        while (!isEmpty()) {
            Patient patient = dequeuePatient();
            if (patient.patientID != -1) {
                printf("Doctor %d is treating patient %d (%s) with severity %d\n",
                       i + 1, patient.patientID, patient.name, patient.severity);
            }
        }
    }
}
```

```
// Function to display the current queue of patients
void displayQueue() {
   struct Node *p = front;
   while (p) {
      printf("PatientID=%d, Name=%s, Severity=%d -> ", p->data.patientID, p->data.name, p->data.severity);
      p = p->next;
   }
   printf("\n");
}
```

```
/*****************************************************************************
****
```

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 the maximum size of the queue
#define MAX 100
```

```
// Define the structure for a survey response
typedef struct {
    int surveyorID;
    int respondentID;
    char response[100];
} SurveyResponse;

// Define the structure for the queue
typedef struct {
    SurveyResponse responses[MAX];
    int front;
    int rear;
} Queue;

// Function prototypes
void initializeQueue(Queue *q);
void enqueueResponse(Queue *q, int surveyorID, int respondentID, char *response);
SurveyResponse dequeueResponse(Queue *q);
void displayQueue(Queue *q);

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

    // Simulate surveyors collecting data
    enqueueResponse(&queue, 1, 101, "Response from respondent 101 by surveyor 1");
    enqueueResponse(&queue, 2, 102, "Response from respondent 102 by surveyor 2");
    enqueueResponse(&queue, 1, 103, "Response from respondent 103 by surveyor 1");
```

```
    enqueueResponse(&queue, 3, 104, "Response from respondent 104 by surveyor 3");
    enqueueResponse(&queue, 2, 105, "Response from respondent 105 by surveyor 2");

    // Display the queue
    printf("Current queue of survey responses:\n");
    displayQueue(&queue);

    // Dequeue and process survey responses
    SurveyResponse processed;
    while ((processed = dequeueResponse(&queue)).surveyorID != -1) {
        printf("Processed response: SurveyorID=%d, RespondentID=%d, Response=%s\n",
               processed.surveyorID, processed.respondentID, processed.response);
    }

    return 0;
}

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

// Function to add a survey response to the queue (enqueue)
void enqueueResponse(Queue *q, int surveyorID, int respondentID, char *response) {
    if (q->rear == MAX - 1) {
        printf("Queue is full. Cannot add response.\n");
        return;
```

```
    }

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->responses[q->rear].surveyorID = surveyorID;
    q->responses[q->rear].respondentID = respondentID;
    strcpy(q->responses[q->rear].response, response);
}

// Function to remove (dequeue) the oldest survey response from the queue
SurveyResponse dequeueResponse(Queue *q) {
    SurveyResponse response = {-1, -1, ""};

    if (q->front == -1) {
        printf("Queue is already empty.\n");
        return response;
    }

    response = q->responses[q->front];
    q->front++;

    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }

    return response;
```

```
}

// Function to display the current queue of survey responses
void displayQueue(Queue *q) {
   if (q->front == -1) {
      printf("Queue is empty.\n");
      return;
   }

   for (int i = q->front; i <= q->rear; i++) {
      printf("SurveyorID=%d, RespondentID=%d, Response=%s\n",
           q->responses[i].surveyorID, q->responses[i].respondentID, q-
>responses[i].response);
   }
   printf("\n");
}
```

/**********

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>
#include <string.h>

// Define the structure for market data
typedef struct {
    int dataID;
    char timestamp[20];
    float price;
    int volume;
} MarketData;

// Define the structure for the queue node
struct Node {
    MarketData data;
    struct Node *next;
} *front = NULL, *rear = NULL;

// Function prototypes
void enqueueMarketData(int dataID, char *timestamp, float price, int volume);
MarketData dequeueMarketData();
int isEmpty();
void performRealTimeAnalysis();
void displayQueue();

int main() {
    // Add sample market data to the queue
    enqueueMarketData(1, "2025-01-20 10:00:00", 100.5, 1000);
    enqueueMarketData(2, "2025-01-20 10:00:01", 101.0, 2000);
```

```
    enqueueMarketData(3, "2025-01-20 10:00:02", 100.8, 1500);
    enqueueMarketData(4, "2025-01-20 10:00:03", 101.2, 2500);
    enqueueMarketData(5, "2025-01-20 10:00:04", 100.9, 3000);

    // Display the queue
    printf("Current queue of market data:\n");
    displayQueue();

    // Perform real-time analysis
    performRealTimeAnalysis();

    // Display the queue after processing
    printf("Queue after processing:\n");
    displayQueue();

    return 0;
}

// Function to add market data to the queue (enqueue)
void enqueueMarketData(int dataID, char *timestamp, float price, int volume) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
        return;
    }

    t->data.dataID = dataID;
```

```
    strcpy(t->data.timestamp, timestamp);
    t->data.price = price;
    t->data.volume = volume;
    t->next = NULL;

    if (front == NULL) {
        front = rear = t;
    } else {
        rear->next = t;
        rear = t;
    }
}

// Function to remove (dequeue) the oldest market data from the queue
MarketData dequeueMarketData() {
    MarketData data = {-1, "", 0.0, 0};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
        return data;
    } else {
        data = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
```

```
    }
    return data;
}

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

// Function to perform real-time analysis
void performRealTimeAnalysis() {
    printf("Performing real-time analysis...\n");
    int count = 0;
    float totalVolume = 0;
    float totalPrice = 0;

    while (!isEmpty()) {
        MarketData data = dequeueMarketData();
        if (data.dataID != -1) {
            printf("Analyzing data: DataID=%d, Timestamp=%s, Price=%.2f, Volume=%d\n",
                   data.dataID, data.timestamp, data.price, data.volume);

            // Aggregate data for insights
            totalVolume += data.volume;
            totalPrice += data.price;
            count++;
        }
    }
```

```
    // Generate insights
    if (count > 0) {
        float averagePrice = totalPrice / count;
        printf("Total Volume: %.2f\n", totalVolume);
        printf("Average Price: %.2f\n", averagePrice);
    } else {
        printf("No data to analyze.\n");
    }
}

// Function to display the current queue of market data
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("DataID=%d, Timestamp=%s, Price=%.2f, Volume=%d -> ",
               p->data.dataID, p->data.timestamp, p->data.price, p->data.volume);
        p = p->next;
    }
    printf("\n");
}
```