Day 20 programs

---

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

```
#include <stdio.h>

#define MAX_ORDERS 100


typedef struct {
    char type; // 'B' for buy, 'S' for sell
    int price;
    int quantity;
} Order;


Order queue[MAX_ORDERS];
int front = -1, rear = -1;


void enqueue(char type, int price, int quantity) {
    if (rear == MAX_ORDERS - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
    queue[rear].type = type;
    queue[rear].price = price;
    queue[rear].quantity = quantity;
}
```

```
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Processed Order: %c %d %d\n", queue[front].type, queue[front].price,
queue[front].quantity);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Orders in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%c %d %d\n", queue[i].type, queue[i].price, queue[i].quantity);
    }
}

int main() {
    enqueue('B', 100, 10);
    enqueue('S', 101, 5);
    enqueue('B', 99, 20);

    display();

    dequeue();
```

```
    display();

    return 0;
}
```

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

```
#include <stdio.h>
#define MAX_CUSTOMERS 100

typedef struct {
    char name[20];
    int priority; // 1 for VIP, 2 for Regular
} Customer;

Customer queue[MAX_CUSTOMERS];
int front = -1, rear = -1;

void enqueue(char *name, int priority) {
    if (rear == MAX_CUSTOMERS - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
    for (int i = rear; i > front && queue[i - 1].priority > priority; i--) {
        queue[i] = queue[i - 1];
```

```
    }
    queue[rear].priority = priority;
    snprintf(queue[rear].name, sizeof(queue[rear].name), "%s", name);
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Serving Customer: %s (Priority: %d)\n", queue[front].name,
queue[front].priority);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Customers in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s (Priority: %d)\n", queue[i].name, queue[i].priority);
    }
}

int main() {
    enqueue("Alice", 2);
    enqueue("Bob", 1);
```

```
    enqueue("Charlie", 2);

    display();

    dequeue();
    display();

    return 0;
}
```

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

```
#include <stdio.h>
#define MAX_ATTENDEES 100

typedef struct {
    char name[20];
    char type; // 'V' for VIP, 'R' for Regular
} Attendee;

Attendee queue[MAX_ATTENDEES];
int front = -1, rear = -1;

void enqueue(char *name, char type) {
    if (rear == MAX_ATTENDEES - 1) {
        printf("Queue is full!\n");
        return;
    }
```

```
    if (front == -1) front = 0;
    rear++;
    queue[rear].type = type;
    snprintf(queue[rear].name, sizeof(queue[rear].name), "%s", name);
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Checking in: %s (%c)\n", queue[front].name, queue[front].type);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Attendees in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s (%c)\n", queue[i].name, queue[i].type);
    }
}

int main() {
    enqueue("Alice", 'V');
    enqueue("Bob", 'R');
```

```
    enqueue("Charlie", 'V');

    display();

    dequeue();
    display();

    return 0;
}
```

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>

typedef struct Customer {
    char name[20];
    int transactionTime; // Time required for the transaction
    struct Customer *next;
} Customer;

Customer *front = NULL, *rear = NULL;

void enqueue(char *name, int transactionTime) {
    Customer *newCustomer = (Customer *)malloc(sizeof(Customer));
    if (!newCustomer) {
```

```
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newCustomer->name, name);
    newCustomer->transactionTime = transactionTime;
    newCustomer->next = NULL;

    if (rear == NULL) {
        front = rear = newCustomer;
    } else {
        rear->next = newCustomer;
        rear = newCustomer;
    }
    printf("Customer added: %s (Transaction Time: %d mins)\n", name,
transactionTime);
}

void dequeue(int tellerId) {
    if (front == NULL) {
        printf("Teller %d: No customers in the queue!\n", tellerId);
        return;
    }
    Customer *temp = front;
    printf("Teller %d is serving: %s (Transaction Time: %d mins)\n", tellerId, front-
>name, front->transactionTime);
    front = front->next;

    if (front == NULL) rear = NULL;

    free(temp);
```

```
}

void displayQueue() {
    if (front == NULL) {
        printf("No customers in the queue!\n");
        return;
    }
    printf("Customers in queue:\n");
    Customer *temp = front;
    while (temp != NULL) {
        printf("%s (Transaction Time: %d mins)\n", temp->name, temp-
>transactionTime);
        temp = temp->next;
    }
}

int main() {
    // Adding customers to the queue
    enqueue("Alice", 5);
    enqueue("Bob", 10);
    enqueue("Charlie", 8);

    displayQueue();

    // Simulating tellers serving customers
    dequeue(1); // Teller 1
    displayQueue();

    dequeue(2); // Teller 2
```

```
    displayQueue();

    enqueue("David", 6);
    enqueue("Eve", 4);

    displayQueue();

    dequeue(1); // Teller 1
    displayQueue();

    return 0;
}
```

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>
#define MAX_DATA 100

int dataQueue[MAX_DATA];
int front = -1, rear = -1;

void enqueue(int data) {
    if (rear == MAX_DATA - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
```

```
    dataQueue[rear] = data;
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Processing Data: %d\n", dataQueue[front]);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Data in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%d ", dataQueue[i]);
    }
    printf("\n");
}

int main() {
    enqueue(101);
    enqueue(102);
    enqueue(103);
```

```
    display();

    dequeue();
    display();

    return 0;
}
```

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

```
#include <stdio.h>
#define MAX_CARS 100

typedef struct {
    char licensePlate[10];
} Car;

Car carQueue[MAX_CARS];
int front = -1, rear = -1;

void enqueue(char *licensePlate) {
    if (rear == MAX_CARS - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
```

```
    snprintf(carQueue[rear].licensePlate, sizeof(carQueue[rear].licensePlate), "%s",
licensePlate);
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Car crossing: %s\n", carQueue[front].licensePlate);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Cars in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s ", carQueue[i].licensePlate);
    }
    printf("\n");
}

int main() {
    enqueue("ABC123");
    enqueue("XYZ789");
    enqueue("LMN456");
```

```
    display();

    dequeue();
    display();

    return 0;
}
```

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

```
#include <stdio.h>
#define MAX_VOTES 100

int voteQueue[MAX_VOTES];
int front = -1, rear = -1;

void enqueue(int vote) {
    if (rear == MAX_VOTES - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
    voteQueue[rear] = vote;
}

void dequeue() {
```

```
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Counting Vote: Candidate %d\n", voteQueue[front]);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Votes in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("Candidate %d ", voteQueue[i]);
    }
    printf("\n");
}

int main() {
    enqueue(1);
    enqueue(2);
    enqueue(1);
    enqueue(3);

    display();

    dequeue();
```

```
    display();

    return 0;
}
```

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

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

typedef struct Plane {
    char id[10];
    char type; // 'E' for emergency, 'R' for regular
    struct Plane *next;
} Plane;

Plane *front = NULL, *rear = NULL;

void enqueue(char *id, char type) {
    Plane *newPlane = (Plane *)malloc(sizeof(Plane));
    if (!newPlane) {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newPlane->id, id);
    newPlane->type = type;
    newPlane->next = NULL;
```

```
    if (type == 'E') {
        // Emergency landings are added to the front
        if (front == NULL) {
            front = rear = newPlane;
        } else {
            newPlane->next = front;
            front = newPlane;
        }
    } else {
        // Regular planes are added to the end
        if (rear == NULL) {
            front = rear = newPlane;
        } else {
            rear->next = newPlane;
            rear = newPlane;
        }
    }
    printf("Plane added: %s (%c)\n", id, type);
}

void dequeue() {
    if (front == NULL) {
        printf("No planes in the queue!\n");
        return;
    }
    Plane *temp = front;
    printf("Allocating runway to: %s (%c)\n", front->id, front->type);
    front = front->next;
```

```
    if (front == NULL) rear = NULL;

    free(temp);
}

void display() {
    if (front == NULL) {
        printf("No planes in the queue!\n");
        return;
    }
    printf("Planes in queue:\n");
    Plane *temp = front;
    while (temp != NULL) {
        printf("%s (%c)\n", temp->id, temp->type);
        temp = temp->next;
    }
}

int main() {
    // Adding planes to the queue
    enqueue("Flight101", 'R');
    enqueue("Flight202", 'E');
    enqueue("Flight303", 'R');
    enqueue("Flight404", 'E');

    display();

    // Allocating runway
```

```
    dequeue();
    display();

    dequeue();
    display();

    enqueue("Flight505", 'R');
    display();

    return 0;
}
```

9. **Stock Trading Simulation**: Develop a program using arrays to simulate a queue for stock trading orders. The system should manage buy and sell orders, handle order cancellations, and provide real-time updates.

```
#include <stdio.h>
#define MAX_ORDERS 100

typedef struct {
    char type; // 'B' for buy, 'S' for sell
    int price;
    int quantity;
} Order;

Order orders[MAX_ORDERS];
int front = -1, rear = -1;

void enqueue(char type, int price, int quantity) {
```

```
    if (rear == MAX_ORDERS - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
    orders[rear].type = type;
    orders[rear].price = price;
    orders[rear].quantity = quantity;
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Processing Order: %c %d %d\n", orders[front].type, orders[front].price,
orders[front].quantity);
    front++;
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Orders in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%c %d %d\n", orders[i].type, orders[i].price, orders[i].quantity);
```

```
    }
}

int main() {
    enqueue('B', 100, 10);
    enqueue('S', 110, 5);
    enqueue('B', 90, 20);

    display();

    dequeue();
    display();

    return 0;
}
```

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 Attendee {
    char name[20];
    char type; // 'P' for pre-registered, 'W' for walk-in
    struct Attendee *next;
} Attendee;
```

```
Attendee *front = NULL, *rear = NULL;

void enqueue(char *name, char type) {
    Attendee *newAttendee = (Attendee *)malloc(sizeof(Attendee));
    if (!newAttendee) {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newAttendee->name, name);
    newAttendee->type = type;
    newAttendee->next = NULL;

    if (rear == NULL) {
        front = rear = newAttendee;
    } else {
        rear->next = newAttendee;
        rear = newAttendee;
    }
    printf("Registered: %s (%c)\n", name, type);
}

void dequeue() {
    if (front == NULL) {
        printf("Queue is empty!\n");
        return;
    }
    Attendee *temp = front;
    printf("Processing: %s (%c)\n", front->name, front->type);
    front = front->next;
```

```
    if (front == NULL) rear = NULL;

    free(temp);
}

void display() {
    if (front == NULL) {
        printf("Queue is empty!\n");
        return;
    }
    Attendee *temp = front;
    printf("Attendees in queue:\n");
    while (temp != NULL) {
        printf("%s (%c)\n", temp->name, temp->type);
        temp = temp->next;
    }
}

int main() {
    // Sample registrations
    enqueue("Alice", 'P');
    enqueue("Bob", 'W');
    enqueue("Charlie", 'P');

    display();

    dequeue();
    display();
```

```
    enqueue("David", 'W');
    display();

    return 0;
}
```

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

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

#define MAX 100

typedef struct {
    char name[20];
    char type; // 'M' for Media, 'A' for Audience
} Person;

Person queue[MAX];
int front = 0, rear = -1;

void enqueue(char *name, char type) {
    if (rear >= MAX - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (type == 'M') {
```

```
        // Shift everyone to make room for Media personnel
        for (int i = ++rear; i > front; i--) {
            queue[i] = queue[i - 1];
        }
        strcpy(queue[front].name, name);
        queue[front].type = type;
    } else {
        rear++;
        strcpy(queue[rear].name, name);
        queue[rear].type = type;
    }
    printf("Added: %s (%c)\n", name, type);
}

void dequeue() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Seating: %s (%c)\n", queue[front].name, queue[front].type);
    front++;
}

void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Current Queue:\n");
```

```
    for (int i = front; i <= rear; i++) {
        printf("%s (%c)\n", queue[i].name, queue[i].type);
    }
}

int main() {
    enqueue("Alice", 'A');
    enqueue("Bob", 'M');
    enqueue("Charlie", 'A');
    display();
    dequeue();
    display();
    return 0;
}
```

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

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

typedef struct Application {
    char name[20];
    int loanAmount;
    int creditScore;
    struct Application *next;
} Application;
```

```
Application *front = NULL;

void enqueue(char *name, int loanAmount, int creditScore) {
    Application *newApp = (Application *)malloc(sizeof(Application));
    strcpy(newApp->name, name);
    newApp->loanAmount = loanAmount;
    newApp->creditScore = creditScore;
    newApp->next = NULL;

    if (front == NULL || (loanAmount > front->loanAmount) ||
        (loanAmount == front->loanAmount && creditScore > front->creditScore)) {
        newApp->next = front;
        front = newApp;
    } else {
        Application *current = front;
        while (current->next != NULL &&
               ((loanAmount < current->next->loanAmount) ||
                (loanAmount == current->next->loanAmount && creditScore <= current-
>next->creditScore))) {
            current = current->next;
        }
        newApp->next = current->next;
        current->next = newApp;
    }
    printf("Application added: %s (Loan: %d, Credit Score: %d)\n", name,
loanAmount, creditScore);
}

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

```
        printf("No applications in the queue!\n");
        return;
    }
    Application *temp = front;
    printf("Processing: %s (Loan: %d, Credit Score: %d)\n", front->name, front->loanAmount, front->creditScore);
    front = front->next;
    free(temp);
}

void display() {
    if (front == NULL) {
        printf("No applications in the queue!\n");
        return;
    }
    Application *temp = front;
    printf("Applications in queue:\n");
    while (temp != NULL) {
        printf("%s (Loan: %d, Credit Score: %d)\n", temp->name, temp->loanAmount, temp->creditScore);
        temp = temp->next;
    }
}

int main() {
    enqueue("Alice", 50000, 750);
    enqueue("Bob", 100000, 800);
    enqueue("Charlie", 75000, 780);
    display();
    dequeue();
```

```
    display();
    return 0;
}
```

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

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

#define MAX 100

typedef struct {
    char name[20];
    int cartTotal; // Number of items in the cart
} Customer;

Customer queue[MAX];
int front = 0, rear = -1;

int inventory = 1000; // Total items available in inventory

void enqueue(char *name, int cartTotal) {
    if (rear >= MAX - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (cartTotal > inventory) {
        printf("Not enough inventory for %s's cart!\n", name);
```

```
        return;
    }
    rear++;
    strcpy(queue[rear].name, name);
    queue[rear].cartTotal = cartTotal;
    printf("Customer added: %s (Cart Total: %d)\n", name, cartTotal);
}

void dequeue() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Processing checkout for: %s (Cart Total: %d)\n", queue[front].name,
queue[front].cartTotal);
    inventory -= queue[front].cartTotal;
    printf("Remaining Inventory: %d\n", inventory);
    front++;
}

void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Customers in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s (Cart Total: %d)\n", queue[i].name, queue[i].cartTotal);
    }
```

```
}

int main() {
    enqueue("Alice", 200);
    enqueue("Bob", 300);
    enqueue("Charlie", 150);
    display();

    dequeue();
    display();

    enqueue("David", 500);
    display();

    return 0;
}
```

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

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

typedef struct Bus {
    char id[10];
    char type; // 'E' for express, 'R' for regular
    struct Bus *next;
} Bus;
```

```
Bus *front = NULL, *rear = NULL;

void enqueue(char *id, char type) {
    Bus *newBus = (Bus *)malloc(sizeof(Bus));
    if (!newBus) {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newBus->id, id);
    newBus->type = type;
    newBus->next = NULL;

    if (type == 'E') {
        // Priority insertion for express buses
        if (front == NULL) {
            front = rear = newBus;
        } else {
            newBus->next = front;
            front = newBus;
        }
    } else {
        // Regular buses added to the end
        if (rear == NULL) {
            front = rear = newBus;
        } else {
            rear->next = newBus;
            rear = newBus;
        }
```

```
    }
    printf("Bus added: %s (%c)\n", id, type);
}

void dequeue() {
    if (front == NULL) {
        printf("No buses in the queue!\n");
        return;
    }
    Bus *temp = front;
    printf("Departing: %s (%c)\n", front->id, front->type);
    front = front->next;

    if (front == NULL) rear = NULL;

    free(temp);
}

void display() {
    if (front == NULL) {
        printf("No buses in the queue!\n");
        return;
    }
    printf("Buses in queue:\n");
    Bus *temp = front;
    while (temp != NULL) {
        printf("%s (%c)\n", temp->id, temp->type);
        temp = temp->next;
    }
```

```
}

int main() {
    enqueue("Bus101", 'R');
    enqueue("Bus202", 'E');
    enqueue("Bus303", 'R');
    enqueue("Bus404", 'E');

    display();

    dequeue();
    display();

    enqueue("Bus505", 'R');
    display();

    return 0;
}
```

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

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

#define MAX 100

typedef struct {
    char name[20];
```

```
    char type; // 'V' for VIP, 'G' for General
} Attendee;

Attendee queue[MAX];
int front = 0, rear = -1;

void enqueue(char *name, char type) {
    if (rear >= MAX - 1) {
        printf("Queue is full!\n");
        return;
    }
    if (type == 'V') {
        // Shift everyone to make room for VIP
        for (int i = ++rear; i > front; i--) {
            queue[i] = queue[i - 1];
        }
        strcpy(queue[front].name, name);
        queue[front].type = type;
    } else {
        rear++;
        strcpy(queue[rear].name, name);
        queue[rear].type = type;
    }
    printf("Added: %s (%c)\n", name, type);
}

void dequeue() {
    if (front > rear) {
        printf("Queue is empty!\n");
```

```
        return;
    }
    printf("Seating: %s (%c)\n", queue[front].name, queue[front].type);
    front++;
}

void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Current Queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s (%c)\n", queue[i].name, queue[i].type);
    }
}

int main() {
    enqueue("Alice", 'G');
    enqueue("Bob", 'V');
    enqueue("Charlie", 'G');
    display();

    dequeue();
    display();

    enqueue("David", 'V');
    display();
```

```
    return 0;
}
```

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

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

typedef struct Transaction {
    char type[20]; // Deposit, Withdrawal, or Transfer
    double amount;
    struct Transaction *next;
} Transaction;

Transaction *front = NULL, *rear = NULL;

void enqueue(char *type, double amount) {
    Transaction *newTrans = (Transaction *)malloc(sizeof(Transaction));
    if (!newTrans) {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newTrans->type, type);
    newTrans->amount = amount;
    newTrans->next = NULL;

    if (rear == NULL) {
```

```
        front = rear = newTrans;
    } else {
        rear->next = newTrans;
        rear = newTrans;
    }
    printf("Transaction added: %s (Amount: %.2f)\n", type, amount);
}

void dequeue() {
    if (front == NULL) {
        printf("No transactions in the queue!\n");
        return;
    }
    Transaction *temp = front;
    printf("Processing: %s (Amount: %.2f)\n", front->type, front->amount);
    front = front->next;

    if (front == NULL) rear = NULL;

    free(temp);
}

void display() {
    if (front == NULL) {
        printf("No transactions in the queue!\n");
        return;
    }
    printf("Transactions in queue:\n");
    Transaction *temp = front;
```

```
    while (temp != NULL) {
        printf("%s (Amount: %.2f)\n", temp->type, temp->amount);
        temp = temp->next;
    }
}

int main() {
    enqueue("Deposit", 1000.00);
    enqueue("Withdrawal", 500.00);
    enqueue("Transfer", 250.00);
    display();

    dequeue();
    display();

    enqueue("Deposit", 2000.00);
    display();

    return 0;
}
```

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

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

#define MAX 100
```

```
typedef struct {
    char name[20];
    int voterId;
} Voter;

Voter queue[MAX];
int front = 0, rear = -1;

void enqueue(char *name, int voterId) {
    if (rear >= MAX - 1) {
        printf("Queue is full!\n");
        return;
    }
    rear++;
    strcpy(queue[rear].name, name);
    queue[rear].voterId = voterId;
    printf("Voter registered: %s (ID: %d)\n", name, voterId);
}

void dequeue() {
    if (front > rear) {
        printf("No voters in the queue!\n");
        return;
    }
    printf("Processing vote for: %s (ID: %d)\n", queue[front].name,
queue[front].voterId);
    front++;
}
```

```
void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Voters in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s (ID: %d)\n", queue[i].name, queue[i].voterId);
    }
}

int main() {
    enqueue("Alice", 1001);
    enqueue("Bob", 1002);
    enqueue("Charlie", 1003);
    display();

    dequeue();
    display();

    enqueue("David", 1004);
    display();

    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.

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

typedef struct Patient {
    char name[20];
    int severity; // Higher value means higher severity
    struct Patient *next;
} Patient;

Patient *front = NULL, *rear = NULL;

void enqueue(char *name, int severity) {
    Patient *newPatient = (Patient *)malloc(sizeof(Patient));
    if (!newPatient) {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newPatient->name, name);
    newPatient->severity = severity;
    newPatient->next = NULL;

    if (front == NULL || severity > front->severity) {
        // Higher severity patients go to the front
        newPatient->next = front;
        front = newPatient;
    } else {
        Patient *current = front;
        while (current->next != NULL && current->next->severity >= severity) {
```

```
            current = current->next;
        }
        newPatient->next = current->next;
        current->next = newPatient;
    }

    printf("Patient added: %s (Severity: %d)\n", name, severity);
}

void dequeue() {
    if (front == NULL) {
        printf("No patients in the queue!\n");
        return;
    }
    Patient *temp = front;
    printf("Treating: %s (Severity: %d)\n", front->name, front->severity);
    front = front->next;

    if (front == NULL) rear = NULL;

    free(temp);
}

void display() {
    if (front == NULL) {
        printf("No patients in the queue!\n");
        return;
    }
    printf("Patients in queue:\n");
```

```
    Patient *temp = front;
    while (temp != NULL) {
        printf("%s (Severity: %d)\n", temp->name, temp->severity);
        temp = temp->next;
    }
}

int main() {
    enqueue("Alice", 3); // Critical
    enqueue("Bob", 1);   // Non-Critical
    enqueue("Charlie", 5); // Critical
    display();

    dequeue();
    display();

    enqueue("David", 2); // Less critical
    display();

    return 0;
}
```

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 <string.h>

#define MAX 100
```

```
typedef struct {
    char surveyorName[20];
    char candidate[20];
    int votes;
} SurveyData;

SurveyData queue[MAX];
int front = 0, rear = -1;

void enqueue(char *surveyorName, char *candidate, int votes) {
    if (rear >= MAX - 1) {
        printf("Queue is full!\n");
        return;
    }
    rear++;
    strcpy(queue[rear].surveyorName, surveyorName);
    strcpy(queue[rear].candidate, candidate);
    queue[rear].votes = votes;
    printf("Survey data collected by: %s (Candidate: %s, Votes: %d)\n",
surveyorName, candidate, votes);
}

void dequeue() {
    if (front > rear) {
        printf("No survey data to process!\n");
        return;
    }
    printf("Processing: %s collected data for %s (Votes: %d)\n",
queue[front].surveyorName, queue[front].candidate, queue[front].votes);
```

```
    front++;
}

void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Survey data in queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("%s collected data for %s (Votes: %d)\n", queue[i].surveyorName,
queue[i].candidate, queue[i].votes);
    }
}

int main() {
    enqueue("Surveyor1", "CandidateA", 100);
    enqueue("Surveyor2", "CandidateB", 150);
    enqueue("Surveyor3", "CandidateA", 200);
    display();

    dequeue();
    display();

    enqueue("Surveyor4", "CandidateC", 50);
    display();

    return 0;
}
```

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

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

typedef struct MarketData {
    char symbol[10];
    double price;
    struct MarketData *next;
} MarketData;

MarketData *front = NULL, *rear = NULL;

void enqueue(char *symbol, double price) {
    MarketData *newData = (MarketData *)malloc(sizeof(MarketData));
    if (!newData) {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newData->symbol, symbol);
    newData->price = price;
    newData->next = NULL;

    if (rear == NULL) {
        front = rear = newData;
    } else {
```

```
        rear->next = newData;
        rear = newData;
    }
    printf("Market data added: %s (Price: %.2f)\n", symbol, price);
}

void dequeue() {
    if (front == NULL) {
        printf("No market data to analyze!\n");
        return;
    }
    MarketData *temp = front;
    printf("Analyzing: %s (Price: %.2f)\n", front->symbol, front->price);
    front = front->next;

    if (front == NULL) rear = NULL;

    free(temp);
}

void display() {
    if (front == NULL) {
        printf("No market data in the queue!\n");
        return;
    }
    printf("Market data in queue:\n");
    MarketData *temp = front;
    while (temp != NULL) {
        printf("%s (Price: %.2f)\n", temp->symbol, temp->price);
```

```
        temp = temp->next;
    }
}

int main() {
    enqueue("AAPL", 150.25);
    enqueue("GOOG", 2800.50);
    enqueue("AMZN", 3400.75);
    display();

    dequeue();
    display();

    enqueue("TSLA", 720.30);
    display();

    return 0;
}
```