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 <string.h>

#define MAX_ORDERS 10

typedef struct{
    int orderID;
    char type[5]; // buy or sell
    float price;
    int quantity;
}Order;

Order buy[MAX_ORDERS];
int front_buy =0,rear_buy =-1;

Order sell[MAX_ORDERS];
int front_sell =0, rear_sell =-1;

void add_order(int id, const char* type, float price, int quantity);
void match_orders();
void display_orders();

int main(){
    add_order(1, "buy", 100.5, 10);
    add_order(2, "buy", 101.0, 5);
    add_order(3, "sell", 99.5, 7);
    add_order(4, "sell", 100.0, 12);

    printf("Before Matching : \n");
    display_orders();

    printf("\n Matching orders: \n");
    match_orders();

    printf("\nAfter Matching: \n");
    display_orders();
```

```
    return 0;
}

void add_order(int id, const char* type, float price, int quantity){
    Order new ={id,"",price,quantity};
    strcpy(new.type,type);
    if(strcmp(type,"buy")==0){
        if(rear_buy<MAX_ORDERS){
            rear_buy++;
            buy[rear_buy] = new;
        }else{
            printf("Buy queue is full\n");
        }
    }else if(strcmp(type,"sell")==0){
        if(rear_sell<MAX_ORDERS){
            rear_sell++;
            sell[rear_sell] =new;
        }else{
            printf("Sell queuee is full");
        }
    }
}
void match_orders(){
    while(front_buy <= rear_buy && front_sell <= rear_sell){
        if(buy[front_buy].price >= sell[front_sell].price){
            int matched = (buy[front_buy].quantity < sell[front_sell].quantity) ?
buy[front_buy].quantity : sell[front_sell].quantity;

            printf("Matched : Buy order %d and sell order %d, Quantity : %d, Price : %.2f
\n",buy[front_buy].orderID,sell[front_buy].orderID,matched,sell[front_sell].price);

            buy[front_buy].quantity -= matched;
            sell[front_sell].quantity -= matched;

            if(buy[front_buy].quantity == 0){
                front_buy++;
            }
            if(sell[front_sell].quantity ==0){
                front_sell++;
            }else{
```

```
            printf("No match\n");
            break;
        }
    }
  }
}
void display_orders(){
  printf("\nBuy Orders: \n");
  for(int i= front_buy;i<=rear_buy;i++){
    printf("OrderID: %d, Price: %.2f, Quantity: %d\n", buy[i].orderID, buy[i].price,
buy[i].quantity);
  }
  printf("\nSell Orders:\n");
  for (int i = front_sell; i <= rear_sell; i++) {
    printf("OrderID: %d, Price: %.2f, Quantity: %d\n", sell[i].orderID, sell[i].price,
sell[i].quantity);
  }
}
```

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

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

typedef struct Customer
{
  int id;
  char name[50];
  int priority;
  struct Customer *next;
}Customer;
```

```
Customer *front = NULL;

Customer *create(int id,const char *name,int priority){
    Customer *new = (Customer *)malloc (sizeof(Customer));
    new->id = id;
    strcpy(new->name,name);
    new->priority = priority;
    new->next =NULL;
    return new;
}
void enqueue(int id, const char* name, int priority);
void dequeue();
void display_queue();

int main(){
    enqueue(1, "Alice", 2); // Medium priority
    enqueue(2, "Bob", 1);   // High priority
    enqueue(3, "Charlie", 3); // Low priority
    enqueue(4, "Daisy", 1);  // High priority

    printf("\nBefore Serving:\n");
    display_queue();
    printf("\nServing Customers:\n");
    dequeue();
    dequeue();

    printf("\nAfter Serving:\n");
    display_queue();

    return 0;

}

void enqueue(int id, const char* name, int priority){
    Customer *new = create(id,name,priority);

    if(front == NULL || front->priority >priority){
        new->next = front;
        front = new;
    }
    else{
```

```
        Customer *temp = front;
        while(temp->next != NULL && temp->next->priority <= priority){
            temp = temp->next;
        }
        new->next = temp->next;
        temp->next = new;
    }
    printf("Customer %s with priority %d added to the queue.\n", name, priority);
}

void dequeue(){
    if (front == NULL) {
        printf("No customers in the queue.\n");
        return;
    }

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


void display_queue(){
    if (front == NULL) {
        printf("No customers in the queue.\n");
        return;
    }

    printf("Current Queue:\n");
    Customer* temp = front;
    while (temp != NULL) {
        printf("ID: %d, Name: %s, Priority: %d\n", temp->id, temp->name, temp->priority);
        temp = temp->next;
    }
}
```

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.

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

#define MAX_QUEUE_SIZE 100

typedef struct {
    int id;
    char name[50];
    int is_vip;
} Attendee;

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

int is_full();
int is_empty();
void register_attendee(int id, const char* name, int is_vip);
void check_in();
void display_queue();

int main() {
    register_attendee(1, "Alice", 0);
    register_attendee(2, "Bob", 1);
    register_attendee(3, "Charlie", 0);
    register_attendee(4, "Daisy", 1);

    printf("\nBefore Check-In:\n");
    display_queue();

    printf("\nCheck-In Process:\n");
    check_in();
    check_in();

    printf("\nAfter Check-In:\n");
    display_queue();

    return 0;
}

int is_full() {
    return rear == MAX_QUEUE_SIZE - 1;
```

```
}

int is_empty() {
    return front == -1 || front > rear;
}

void register_attendee(int id, const char* name, int is_vip) {
    if (is_full()) {
        printf("Registration failed: The queue is full.\n");
        return;
    }

    if (is_vip) {
        if (front == -1) {
            front = 0;
        }
        for (int i = ++rear; i > front; i--) {
            queue[i] = queue[i - 1];
        }
        queue[front].id = id;
        strcpy(queue[front].name, name);
        queue[front].is_vip = is_vip;
    } else {
        if (front == -1) {
            front = 0;
        }
        queue[++rear].id = id;
        strcpy(queue[rear].name, name);
        queue[rear].is_vip = is_vip;
    }

    printf("Attendee %s (ID: %d, VIP: %s) registered successfully.\n", name, id, is_vip ?
"Yes" : "No");
}

void check_in() {
    if (is_empty()) {
        printf("No attendees in the queue for check-in.\n");
        return;
    }
```

```
    printf("Checked in: %s (ID: %d, VIP: %s)\n",
           queue[front].name, queue[front].id, queue[front].is_vip ? "Yes" : "No");
    front++;
}

void display_queue() {
    if (is_empty()) {
        printf("No attendees in the queue.\n");
        return;
    }

    printf("Current Queue:\n");
    for (int i = front; i <= rear; i++) {
        printf("ID: %d, Name: %s, VIP: %s\n", queue[i].id, queue[i].name, queue[i].is_vip ?
"Yes" : "No");
    }
}
```

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

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

#define MAX_TELLERS 3

typedef struct Customer {
    int id;
    char name[50];
    int transaction_time; // Time required for transaction in seconds
    struct Customer* next;
} Customer;

typedef struct Teller {
    int id;
    Customer* current_customer;
} Teller;
```

```
Customer* front = NULL;
Customer* rear = NULL;

Teller tellers[MAX_TELLERS];

int is_full();
int is_empty();
void enqueue(int id, const char* name, int transaction_time);
Customer* dequeue();
void initialize_tellers();
int is_all_tellers_free();
void assign_customer_to_teller();
void process_transactions();
void simulate_bank_operations(int num_customers);

int main() {
    initialize_tellers();

    int num_customers;
    printf("Enter the number of customers to simulate: ");
    scanf("%d", &num_customers);

    simulate_bank_operations(num_customers);

    return 0;
}

int is_full() {
    return rear != NULL && rear->next == NULL;
}

int is_empty() {
    return front == NULL;
}

void enqueue(int id, const char* name, int transaction_time) {
    Customer* new_customer = (Customer*)malloc(sizeof(Customer));
    new_customer->id = id;
    strcpy(new_customer->name, name);
    new_customer->transaction_time = transaction_time;
```

```
    new_customer->next = NULL;

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

Customer* dequeue() {
    if (front == NULL) return NULL;

    Customer* temp = front;
    front = front->next;

    if (front == NULL) rear = NULL;

    return temp;
}

void initialize_tellers() {
    for (int i = 0; i < MAX_TELLERS; i++) {
        tellers[i].id = i + 1;
        tellers[i].current_customer = NULL;
    }
}

int is_all_tellers_free() {
    for (int i = 0; i < MAX_TELLERS; i++) {
        if (tellers[i].current_customer == NULL) return 1;
    }
    return 0;
}

void assign_customer_to_teller() {
    if (!front) return;

    for (int i = 0; i < MAX_TELLERS; i++) {
        if (tellers[i].current_customer == NULL) {
            Customer* customer = dequeue();
```

```
            tellers[i].current_customer = customer;
            printf("Teller %d is serving Customer %d: %s (Transaction Time: %d
seconds)\n",
                tellers[i].id, customer->id, customer->name, customer->transaction_time);
            break;
        }
    }
}

void process_transactions() {
    for (int i = 0; i < MAX_TELLERS; i++) {
        if (tellers[i].current_customer != NULL) {
            tellers[i].current_customer->transaction_time--;
            if (tellers[i].current_customer->transaction_time == 0) {
                printf("Teller %d finished serving Customer %d: %s\n",
                    tellers[i].id, tellers[i].current_customer->id,
tellers[i].current_customer->name);
                free(tellers[i].current_customer);
                tellers[i].current_customer = NULL;
            }
        }
    }
}

void simulate_bank_operations(int num_customers) {
    srand(time(0));

    for (int i = 0; i < num_customers; i++) {
        int transaction_time = rand() % 10 + 1;
        char name[50];
        sprintf(name, "Customer %d", i + 1);
        enqueue(i + 1, name, transaction_time);
        printf("Customer %d added to queue with transaction time %d seconds\n", i + 1,
transaction_time);
    }

    while (front != NULL || is_all_tellers_free()) {
        assign_customer_to_teller();
        process_transactions();
        sleep(1);
    }
```

```
    printf("All customers have been served.\n");
}
```

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

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

#define MAX_QUEUE_SIZE 100
#define MAX_INSTRUMENT_NAME 50

typedef struct DataFeed {
    char instrument_name[MAX_INSTRUMENT_NAME];
    double price;
    long timestamp; // Timestamp in milliseconds
} DataFeed;

DataFeed queue[MAX_QUEUE_SIZE];
int front = -1, rear = -1;

int is_full() {
    return rear == MAX_QUEUE_SIZE - 1;
}

int is_empty() {
    return front == -1;
}

void enqueue(DataFeed data) {
    if (is_full()) {
        printf("Queue is full. Cannot process more data.\n");
        return;
    }

    if (front == -1) {
        front = 0;
```

```
    }

    queue[++rear] = data;
    printf("Data added to queue: %s | Price: %.2f | Timestamp: %ld\n",
           data.instrument_name, data.price, data.timestamp);
}

DataFeed dequeue() {
    if (is_empty()) {
        printf("Queue is empty. No data to process.\n");
        DataFeed empty_data = {"", 0.0, 0};
        return empty_data;
    }

    DataFeed data = queue[front];

    // Move the front pointer
    if (front == rear) {
        front = rear = -1;
    } else {
        front++;
    }

    return data;
}

void process_data_feed() {
    while (!is_empty()) {
        DataFeed data = dequeue();
        printf("Processing Data: %s | Price: %.2f | Timestamp: %ld\n",
               data.instrument_name, data.price, data.timestamp);
    }
}

int main() {
    DataFeed data1 = {"AAPL", 150.25, 1674186701000};
    DataFeed data2 = {"GOOG", 2805.65, 1674186702000};
    DataFeed data3 = {"AMZN", 3450.50, 1674186703000};

    enqueue(data1);
    enqueue(data2);
```

```
    enqueue(data3);

    printf("\nProcessing Data Feed:\n");
    process_data_feed();

    return 0;
}
```

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

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

#define MAX_CAR_NAME_LENGTH 50

int is_empty();
void enqueue(int id, const char* name, int arrival_time);
Car* dequeue();
void change_traffic_light();
void process_cars();
void simulate_traffic_light_system(int num_cars);

typedef struct Car {
    int id;
    char name[MAX_CAR_NAME_LENGTH];
    int arrival_time;
    struct Car* next;
} Car;

Car* front = NULL;
Car* rear = NULL;

int main() {
    int num_cars;
    printf("Enter the number of cars arriving at the traffic light: ");
    scanf("%d", &num_cars);
```

```
    simulate_traffic_light_system(num_cars);

    return 0;
}

int is_empty() {
    return front == NULL;
}

void enqueue(int id, const char* name, int arrival_time) {
    Car* new_car = (Car*)malloc(sizeof(Car));
    new_car->id = id;
    strcpy(new_car->name, name);
    new_car->arrival_time = arrival_time;
    new_car->next = NULL;

    if (rear == NULL) {
        front = rear = new_car;
    } else {
        rear->next = new_car;
        rear = new_car;
    }
    printf("Car %s (ID: %d) arrived at the traffic light at time %d seconds.\n", name, id,
arrival_time);
}

Car* dequeue() {
    if (is_empty()) {
        printf("No cars to process.\n");
        return NULL;
    }

    Car* car_to_process = front;
    front = front->next;

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

    return car_to_process;
}
```

```
void change_traffic_light() {
    static int red_time = 5;
    static int green_time = 3;

    printf("\nTraffic light is now RED for %d seconds.\n", red_time);
    printf("Traffic light is now GREEN for %d seconds.\n", green_time);
}

void process_cars() {
    while (!is_empty()) {
        Car* car = dequeue();
        printf("Processing car %s (ID: %d) during GREEN light.\n", car->name, car->id);
        free(car);
    }
}

void simulate_traffic_light_system(int num_cars) {
    int arrival_time;
    char car_name[MAX_CAR_NAME_LENGTH];

    for (int i = 0; i < num_cars; i++) {
        printf("Enter car name: ");
        scanf("%s", car_name);
        printf("Enter car arrival time (in seconds): ");
        scanf("%d", &arrival_time);
        enqueue(i + 1, car_name, arrival_time);
    }

    while (!is_empty()) {
        change_traffic_light();
        process_cars();
    }

    printf("All cars have passed the traffic light.\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_VOTES 100

int front = -1, rear = -1;

int is_empty();
int is_full();
void enqueue(int polling_station_id, int candidate_id);
Vote dequeue();
void count_votes();
void simulate_vote_counting_system(int num_votes);

typedef struct Vote {
    int polling_station_id;
    int candidate_id;
} Vote;

Vote vote_queue[MAX_VOTES];

int main() {
    int num_votes;
    printf("Enter the number of votes: ");
    scanf("%d", &num_votes);

    simulate_vote_counting_system(num_votes);

    return 0;
}

int is_empty() {
    return front == -1;
}

int is_full() {
    return rear == MAX_VOTES - 1;
}
```

```
void enqueue(int polling_station_id, int candidate_id) {
    if (is_full()) {
        printf("Vote queue is full, cannot accept more votes.\n");
        return;
    }

    if (front == -1) {
        front = 0;
    }

    rear++;
    vote_queue[rear].polling_station_id = polling_station_id;
    vote_queue[rear].candidate_id = candidate_id;

    printf("Vote added from Polling Station %d for Candidate %d.\n", polling_station_id,
candidate_id);
}

Vote dequeue() {
    Vote vote;
    if (is_empty()) {
        printf("No votes to process.\n");
        vote.polling_station_id = -1;
        vote.candidate_id = -1;
        return vote;
    }

    vote = vote_queue[front];
    front++;

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

    return vote;
}

void count_votes() {
    int candidate_votes[10] = {0};

    while (!is_empty()) {
```

```
        Vote vote = dequeue();
        if (vote.polling_station_id != -1) {
            candidate_votes[vote.candidate_id - 1]++;
        }
    }

    printf("\nVote Count Results:\n");
    for (int i = 0; i < 10; i++) {
        printf("Candidate %d: %d votes\n", i + 1, candidate_votes[i]);
    }
}

void simulate_vote_counting_system(int num_votes) {
    int polling_station_id, candidate_id;

    for (int i = 0; i < num_votes; i++) {
        printf("Enter Polling Station ID: ");
        scanf("%d", &polling_station_id);
        printf("Enter Candidate ID (1 to 10): ");
        scanf("%d", &candidate_id);
        enqueue(polling_station_id, candidate_id);
    }

    count_votes();
}
```

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>

#define MAX_AIRPLANE_NAME_LENGTH 50

int is_empty();
void enqueue(int airplane_id, const char* airplane_name, int priority, const char*
operation);
Airplane* dequeue();
```

```
void manage_runway();
void simulate_airport_runway_management(int num_airplanes);

typedef struct Airplane {
    int airplane_id;
    char airplane_name[MAX_AIRPLANE_NAME_LENGTH];
    int priority; // 1 for emergency, 0 for normal
    char operation[MAX_AIRPLANE_NAME_LENGTH]; // "land" or "take off"
    struct Airplane* next;
} Airplane;

Airplane* front = NULL;
Airplane* rear = NULL;

int is_empty() {
    return front == NULL;
}

void enqueue(int airplane_id, const char* airplane_name, int priority, const char*
operation) {
    Airplane* new_airplane = (Airplane*)malloc(sizeof(Airplane));
    new_airplane->airplane_id = airplane_id;
    strcpy(new_airplane->airplane_name, airplane_name);
    new_airplane->priority = priority;
    strcpy(new_airplane->operation, operation);
    new_airplane->next = NULL;

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

    printf("Airplane %s (ID: %d) is waiting to %s with priority %d.\n", airplane_name,
airplane_id, operation, priority);
}

Airplane* dequeue() {
    if (is_empty()) {
        printf("No airplanes waiting.\n");
```

```
        return NULL;
    }

    Airplane* airplane_to_process = front;
    front = front->next;

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

    return airplane_to_process;
}

void manage_runway() {
    if (is_empty()) {
        return;
    }

    Airplane* airplane = dequeue();

    if (airplane != NULL) {
        printf("Airplane %s (ID: %d) is granted runway for %s.\n",
airplane->airplane_name, airplane->airplane_id, airplane->operation);
        free(airplane);
    }
}

void simulate_airport_runway_management(int num_airplanes) {
    int airplane_id, priority;
    char airplane_name[MAX_AIRPLANE_NAME_LENGTH],
operation[MAX_AIRPLANE_NAME_LENGTH];

    for (int i = 0; i < num_airplanes; i++) {
        printf("Enter Airplane Name: ");
        scanf("%s", airplane_name);
        printf("Enter Airplane ID: ");
        scanf("%d", &airplane_id);
        printf("Enter Operation (land/take off): ");
        scanf("%s", operation);
        printf("Enter Priority (1 for emergency, 0 for normal): ");
        scanf("%d", &priority);
```

```
        enqueue(airplane_id, airplane_name, priority, operation);
    }

    printf("\nManaging runway...\n");
    while (!is_empty()) {
        manage_runway();
    }

    printf("All airplanes have been processed.\n");
}

int main() {
    int num_airplanes;
    printf("Enter the number of airplanes: ");
    scanf("%d", &num_airplanes);

    simulate_airport_runway_management(num_airplanes);

    return 0;
}
```

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

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

#define MAX_ORDERS 100
#define MAX_STOCK_NAME_LENGTH 50

int front_buy = -1, rear_buy = -1;
int front_sell = -1, rear_sell = -1;

int is_empty_buy();
int is_empty_sell();
int is_full_buy();
```

```
int is_full_sell();
void enqueue_buy(int order_id, const char* stock_name, int quantity, float price);
void enqueue_sell(int order_id, const char* stock_name, int quantity, float price);
void cancel_order(int order_id, char order_type);
void process_orders();
void show_orders();
void simulate_stock_trading(int num_orders);

typedef struct StockOrder {
    int order_id;
    char stock_name[MAX_STOCK_NAME_LENGTH];
    int quantity;
    float price;
} StockOrder;

StockOrder buy_orders[MAX_ORDERS];
StockOrder sell_orders[MAX_ORDERS];

int is_empty_buy() {
    return front_buy == -1;
}

int is_empty_sell() {
    return front_sell == -1;
}

int is_full_buy() {
    return rear_buy == MAX_ORDERS - 1;
}

int is_full_sell() {
    return rear_sell == MAX_ORDERS - 1;
}

void enqueue_buy(int order_id, const char* stock_name, int quantity, float price) {
    if (is_full_buy()) {
        printf("Buy order queue is full.\n");
        return;
    }

    if (front_buy == -1) front_buy = 0;
```

```
    rear_buy++;

    buy_orders[rear_buy].order_id = order_id;
    strcpy(buy_orders[rear_buy].stock_name, stock_name);
    buy_orders[rear_buy].quantity = quantity;
    buy_orders[rear_buy].price = price;

    printf("Buy order %d added: Stock: %s, Quantity: %d, Price: %.2f\n", order_id,
stock_name, quantity, price);
}

void enqueue_sell(int order_id, const char* stock_name, int quantity, float price) {
    if (is_full_sell()) {
        printf("Sell order queue is full.\n");
        return;
    }

    if (front_sell == -1) front_sell = 0;
    rear_sell++;

    sell_orders[rear_sell].order_id = order_id;
    strcpy(sell_orders[rear_sell].stock_name, stock_name);
    sell_orders[rear_sell].quantity = quantity;
    sell_orders[rear_sell].price = price;

    printf("Sell order %d added: Stock: %s, Quantity: %d, Price: %.2f\n", order_id,
stock_name, quantity, price);
}

void cancel_order(int order_id, char order_type) {
    if (order_type == 'B') {
        int i;
        for (i = front_buy; i <= rear_buy; i++) {
            if (buy_orders[i].order_id == order_id) {
                printf("Cancelling Buy order %d: Stock: %s\n", buy_orders[i].order_id,
buy_orders[i].stock_name);
                for (int j = i; j < rear_buy; j++) {
                    buy_orders[j] = buy_orders[j + 1];
                }
                rear_buy--;
                if (rear_buy == -1) front_buy = -1;
```

```
            return;
        }
    }
    printf("Buy order %d not found.\n", order_id);
} else if (order_type == 'S') {
    int i;
    for (i = front_sell; i <= rear_sell; i++) {
        if (sell_orders[i].order_id == order_id) {
            printf("Cancelling Sell order %d: Stock: %s\n", sell_orders[i].order_id,
sell_orders[i].stock_name);
            for (int j = i; j < rear_sell; j++) {
                sell_orders[j] = sell_orders[j + 1];
            }
            rear_sell--;
            if (rear_sell == -1) front_sell = -1;
            return;
        }
    }
    printf("Sell order %d not found.\n", order_id);
}
}

void process_orders() {
  while (front_buy <= rear_buy && front_sell <= rear_sell) {
    if (buy_orders[front_buy].price >= sell_orders[front_sell].price) {
        int matched_quantity = (buy_orders[front_buy].quantity <
sell_orders[front_sell].quantity)
                               ? buy_orders[front_buy].quantity : sell_orders[front_sell].quantity;

        printf("Matched Order: Buy Order %d and Sell Order %d, Quantity: %d, Price:
%.2f\n",
               buy_orders[front_buy].order_id, sell_orders[front_sell].order_id,
matched_quantity, sell_orders[front_sell].price);

        buy_orders[front_buy].quantity -= matched_quantity;
        sell_orders[front_sell].quantity -= matched_quantity;

        if (buy_orders[front_buy].quantity == 0) front_buy++; // Remove processed buy
order
        if (sell_orders[front_sell].quantity == 0) front_sell++; // Remove processed sell
order
```

```
        } else {
            break; // No match possible
        }
    }
}

void show_orders() {
    printf("\nBuy Orders:\n");
    for (int i = front_buy; i <= rear_buy; i++) {
        printf("Order ID: %d, Stock: %s, Quantity: %d, Price: %.2f\n",
buy_orders[i].order_id, buy_orders[i].stock_name, buy_orders[i].quantity,
buy_orders[i].price);
    }

    printf("\nSell Orders:\n");
    for (int i = front_sell; i <= rear_sell; i++) {
        printf("Order ID: %d, Stock: %s, Quantity: %d, Price: %.2f\n",
sell_orders[i].order_id, sell_orders[i].stock_name, sell_orders[i].quantity,
sell_orders[i].price);
    }
}

void simulate_stock_trading(int num_orders) {
    int order_id, quantity;
    float price;
    char stock_name[MAX_STOCK_NAME_LENGTH], order_type;

    for (int i = 0; i < num_orders; i++) {
        printf("Enter Order Type (B for Buy, S for Sell): ");
        scanf(" %c", &order_type);
        printf("Enter Stock Name: ");
        scanf("%s", stock_name);
        printf("Enter Order ID: ");
        scanf("%d", &order_id);
        printf("Enter Quantity: ");
        scanf("%d", &quantity);
        printf("Enter Price: ");
        scanf("%f", &price);

        if (order_type == 'B') {
            enqueue_buy(order_id, stock_name, quantity, price);
```

```
        } else if (order_type == 'S') {
            enqueue_sell(order_id, stock_name, quantity, price);
        }

        show_orders();
        process_orders();
    }

    printf("Final Order Status:\n");
    show_orders();
}

int main() {
    int num_orders;
    printf("Enter the number of orders: ");
    scanf("%d", &num_orders);

    simulate_stock_trading(num_orders);

    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>

#define MAX_NAME_LENGTH 100

struct Participant {
    int id;
    char name[MAX_NAME_LENGTH];
    char registration_type; // 'W' for walk-in, 'P' for pre-registration
    struct Participant* next;
};

struct Participant* front = NULL;
```

```
struct Participant* rear = NULL;

void enqueue(int id, const char* name, char registration_type);
void cancel_registration(int id);
void process_registrations();
void show_registrations();
int is_empty();
void register_participant();

int is_empty() {
    return front == NULL;
}

void enqueue(int id, const char* name, char registration_type) {
    struct Participant* new_participant = (struct Participant*)malloc(sizeof(struct
Participant));
    new_participant->id = id;
    strcpy(new_participant->name, name);
    new_participant->registration_type = registration_type;
    new_participant->next = NULL;

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

    printf("Participant Registered: ID: %d, Name: %s, Type: %c\n", id, name,
registration_type);
}

void cancel_registration(int id) {
    struct Participant* temp = front;
    struct Participant* prev = NULL;

    if (temp != NULL && temp->id == id) {
        front = temp->next;
        free(temp);
        printf("Cancelled Registration for Participant ID: %d\n", id);
        return;
```

```
    }

    while (temp != NULL && temp->id != id) {
        prev = temp;
        temp = temp->next;
    }

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

    prev->next = temp->next;
    if (temp == rear) {
        rear = prev;
    }
    free(temp);
    printf("Cancelled Registration for Participant ID: %d\n", id);
}

void process_registrations() {
    if (is_empty()) {
        printf("No registrations to process.\n");
        return;
    }

    struct Participant* current = front;
    while (current != NULL) {
        printf("Processing Participant ID: %d, Name: %s, Type: %c\n", current->id,
current->name, current->registration_type);
        current = current->next;
    }
}

void show_registrations() {
    if (is_empty()) {
        printf("No participants registered.\n");
        return;
    }

    struct Participant* current = front;
```

```
    printf("Registered Participants:\n");
    while (current != NULL) {
        printf("ID: %d, Name: %s, Type: %c\n", current->id, current->name,
current->registration_type);
        current = current->next;
    }
}

void register_participant() {
    int id;
    char name[MAX_NAME_LENGTH];
    char registration_type;

    printf("Enter Participant ID: ");
    scanf("%d", &id);
    printf("Enter Participant Name: ");
    getchar(); // To consume the newline character left by scanf
    fgets(name, MAX_NAME_LENGTH, stdin);
    name[strcspn(name, "\n")] = '\0'; // Remove the newline character
    printf("Enter Registration Type (W for Walk-in, P for Pre-registration): ");
    scanf(" %c", &registration_type);

    enqueue(id, name, registration_type);
}

int main() {
    int choice;

    while (1) {
        printf("\nConference Registration System\n");
        printf("1. Register Participant\n");
        printf("2. Cancel Registration\n");
        printf("3. Process Registrations\n");
        printf("4. Show Registered Participants\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                register_participant();
```

```
            break;
        case 2:
            {
                int id;
                printf("Enter Participant ID to cancel: ");
                scanf("%d", &id);
                cancel_registration(id);
            }
            break;
        case 3:
            process_registrations();
            break;
        case 4:
            show_registrations();
            break;
        case 5:
            printf("Exiting the system.\n");
            exit(0);
            break;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

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

#define MAX_NAME_LENGTH 100
#define MAX_AUDIENCE_SIZE 100
```

```
struct Person {
    int id;
    char name[MAX_NAME_LENGTH];
    char role; // 'M' for Media, 'A' for Audience
};

struct Queue {
    struct Person audience[MAX_AUDIENCE_SIZE];
    int front;
    int rear;
};

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

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

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

void enqueue(struct Queue* q, int id, const char* name, char role) {
    if (is_full(q)) {
        printf("Queue is full! Cannot add more people.\n");
        return;
    }

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

    q->rear++;
    q->audience[q->rear].id = id;
    strcpy(q->audience[q->rear].name, name);
    q->audience[q->rear].role = role;
    printf("Added: ID: %d, Name: %s, Role: %c\n", id, name, role);
}
```

```
void dequeue(struct Queue* q) {
    if (is_empty(q)) {
        printf("Queue is empty! No one to remove.\n");
        return;
    }

    printf("Removed: ID: %d, Name: %s, Role: %c\n", q->audience[q->front].id,
q->audience[q->front].name, q->audience[q->front].role);
    q->front++;

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

void show_queue(struct Queue* q) {
    if (is_empty(q)) {
        printf("No one is in the queue.\n");
        return;
    }

    printf("Current Audience Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("ID: %d, Name: %s, Role: %c\n", q->audience[i].id, q->audience[i].name,
q->audience[i].role);
    }
}

void handle_entry(struct Queue* q) {
    int id;
    char name[MAX_NAME_LENGTH];
    char role;

    printf("Enter Person ID: ");
    scanf("%d", &id);
    printf("Enter Person Name: ");
    getchar(); // To consume the newline character left by scanf
    fgets(name, MAX_NAME_LENGTH, stdin);
    name[strcspn(name, "\n")] = '\0'; // Remove the newline character
    printf("Enter Role (A for Audience, M for Media): ");
```

```
    scanf(" %c", &role);

    enqueue(q, id, name, role);
}

void handle_seating(struct Queue* q) {
    if (is_empty(q)) {
        printf("Queue is empty! No one to seat.\n");
        return;
    }

    struct Queue tempQueue;
    initialize_queue(&tempQueue);

    for (int i = q->front; i <= q->rear; i++) {
        if (q->audience[i].role == 'M') {
            enqueue(&tempQueue, q->audience[i].id, q->audience[i].name,
q->audience[i].role);
        }
    }

    for (int i = q->front; i <= q->rear; i++) {
        if (q->audience[i].role == 'A') {
            enqueue(&tempQueue, q->audience[i].id, q->audience[i].name,
q->audience[i].role);
        }
    }

    *q = tempQueue;
    printf("Seating arrangement done: Media first, then Audience.\n");
}

int main() {
    struct Queue q;
    initialize_queue(&q);

    int choice;

    while (1) {
        printf("\nPolitical Debate Audience Management System\n");
        printf("1. Handle Entry\n");
```

```
        printf("2. Handle Seating\n");
        printf("3. Show Queue\n");
        printf("4. Remove Person (Dequeue)\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                handle_entry(&q);
                break;
            case 2:
                handle_seating(&q);
                break;
            case 3:
                show_queue(&q);
                break;
            case 4:
                dequeue(&q);
                break;
            case 5:
                printf("Exiting the system.\n");
                exit(0);
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    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>

int main();
```

```
void initialize_queue(struct Queue* q);
int is_empty(struct Queue* q);
void enqueue(struct Queue* q, int id, float loanAmount, int creditScore);
void dequeue(struct Queue* q);
void process_applications(struct Queue* q);
void show_queue(struct Queue* q);

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

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

int main() {
    struct Queue q;
    initialize_queue(&q);

    int choice;

    while (1) {
        printf("\nBank Loan Application Processing System\n");
        printf("1. Add Loan Application\n");
        printf("2. Process Loan Applications\n");
        printf("3. Show Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                int id;
                float loanAmount;
                int creditScore;

                printf("Enter Loan Application ID: ");
```

```
            scanf("%d", &id);
            printf("Enter Loan Amount: ");
            scanf("%f", &loanAmount);
            printf("Enter Credit Score: ");
            scanf("%d", &creditScore);

            enqueue(&q, id, loanAmount, creditScore);
            break;
        }
        case 2:
            process_applications(&q);
            break;
        case 3:
            show_queue(&q);
            break;
        case 4:
            printf("Exiting the system.\n");
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
    }
  }

  return 0;
}

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

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

void enqueue(struct Queue* q, int id, float loanAmount, int creditScore) {
  struct LoanApplication* newApplication = (struct
LoanApplication*)malloc(sizeof(struct LoanApplication));
  newApplication->id = id;
  newApplication->loanAmount = loanAmount;
  newApplication->creditScore = creditScore;
```

```
    newApplication->next = NULL;

    if (is_empty(q)) {
        q->front = newApplication;
        q->rear = newApplication;
    } else {
        q->rear->next = newApplication;
        q->rear = newApplication;
    }
}

void dequeue(struct Queue* q) {
    if (is_empty(q)) {
        printf("Queue is empty! No loan applications to process.\n");
        return;
    }

    struct LoanApplication* temp = q->front;
    printf("Processing Loan Application ID: %d, Amount: %.2f, Credit Score: %d\n",
           temp->id, temp->loanAmount, temp->creditScore);
    q->front = q->front->next;
    free(temp);

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

void process_applications(struct Queue* q) {
    if (is_empty(q)) {
        printf("Queue is empty! No loan applications to process.\n");
        return;
    }

    struct Queue sortedQueue;
    initialize_queue(&sortedQueue);

    while (!is_empty(q)) {
        struct LoanApplication* current = q->front;
        dequeue(q);
```

```
        if (is_empty(&sortedQueue)) {
            enqueue(&sortedQueue, current->id, current->loanAmount,
current->creditScore);
        } else {
            struct LoanApplication* temp = sortedQueue.front;
            struct LoanApplication* prev = NULL;
            while (temp != NULL && (temp->loanAmount > current->loanAmount ||
                                    (temp->loanAmount == current->loanAmount &&
temp->creditScore > current->creditScore))) {
                prev = temp;
                temp = temp->next;
            }

            if (prev == NULL) {
                current->next = sortedQueue.front;
                sortedQueue.front = current;
                if (sortedQueue.rear == NULL) {
                    sortedQueue.rear = current;
                }
            } else {
                current->next = prev->next;
                prev->next = current;
                if (current->next == NULL) {
                    sortedQueue.rear = current;
                }
            }
        }
    }

    *q = sortedQueue;
}

void show_queue(struct Queue* q) {
    if (is_empty(q)) {
        printf("No loan applications in the queue.\n");
        return;
    }

    struct LoanApplication* temp = q->front;
    printf("Loan Applications Queue (Sorted by Amount and Credit Score):\n");
    while (temp != NULL) {
```

```
        printf("ID: %d, Loan Amount: %.2f, Credit Score: %d\n",
               temp->id, temp->loanAmount, temp->creditScore);
        temp = temp->next;
    }
}
```

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

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

struct Checkout {
    int customerID;
    int itemID;
    int quantity;
    struct Checkout *next;
};

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

void create(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueueQueue(struct Queue *q, int customerID, int itemID, int quantity); //
Renamed function for Queue
void enqueueRegistration(int id, const char* name, char registration_type); // Renamed
for Registration
void dequeue(struct Queue *q);
void process_checkout(struct Queue *q, int inventory[], int size);
void display(struct Queue *q);

int main() {
    struct Queue q;
    create(&q);
```

```
    int inventory[5] = {100, 50, 75, 30, 200}; // Example inventory for 5 items
    int size = 5;
    int choice;

    while (1) {
        printf("\n1. Enqueue\n2. Process Checkout\n3. Display Queue\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                int customerId, itemId, quantity;
                printf("Enter Customer ID: ");
                scanf("%d", &customerId);
                printf("Enter Item ID (0-4): ");
                scanf("%d", &itemId);
                printf("Enter Quantity: ");
                scanf("%d", &quantity);

                if (itemId >= 0 && itemId < size) {
                    enqueueQueue(&q, customerId, itemId, quantity); // Using the renamed
function
                } else {
                    printf("Invalid Item ID.\n");
                }
                break;
            }
            case 2:
                process_checkout(&q, inventory, size);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting the system.\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
```

```
    return 0;
}

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

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

void enqueueQueue(struct Queue *q, int customerID, int itemID, int quantity) { // 
Renamed function
    struct Checkout *new = (struct Checkout *)malloc(sizeof(struct Checkout));
    new->customerID = customerID;
    new->itemID = itemID;
    new->quantity = quantity;
    new->next = NULL;

    if (isEmpty(q)) {
        q->front = new;
        q->rear = new;
    } else {
        q->rear->next = new;
        q->rear = new;
    }
}

void enqueueRegistration(int id, const char* name, char registration_type) { // Renamed 
function for Registration
    printf("Registered: %d, %s, Type: %c\n", id, name, registration_type);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }
    struct Checkout *temp = q->front;
    q->front = q->front->next;
    free(temp);
```

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

void process_checkout(struct Queue *q, int inventory[], int size) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Checkout *current = q->front;
    while (current != NULL) {
        if (inventory[current->itemID] >= current->quantity) {
            inventory[current->itemID] -= current->quantity;
            printf("Processed Checkout for Customer ID: %d, Item ID: %d, Quantity: %d\n",
                   current->customerID, current->itemID, current->quantity);
            dequeue(q);
        } else {
            printf("Insufficient stock for Customer ID: %d, Item ID: %d. Only %d items
available.\n",
                   current->customerID, current->itemID, inventory[current->itemID]);
            break;
        }
        current = q->front;
    }
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Checkout *temp = q->front;
    printf("Checkout Queue:\n");
    while (temp != NULL) {
        printf("Customer ID: %d, Item ID: %d, Quantity: %d\n",
               temp->customerID, temp->itemID, temp->quantity);
        temp = temp->next;
```

```
    }
}
```

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

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

struct Bus {
    int busID;
    char busType[10]; // Express or Regular
    int arrivalTime;  // Time in minutes since midnight
    int departureTime; // Time in minutes since midnight
    struct Bus *next;
};

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

void create(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, int busID, char busType[], int arrivalTime, int
departureTime);
void dequeue(struct Queue *q);
void display(struct Queue *q);
void prioritizeExpress(struct Queue *q);
void processBusArrival(struct Queue *q);

int main() {
    struct Queue q;
    create(&q);

    int choice;
```

```
    while (1) {
        printf("\n1. Add Bus Arrival\n2. Process Bus Departure\n3. Display Bus Queue\n4.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                int busID, arrivalTime, departureTime;
                char busType[10];

                printf("Enter Bus ID: ");
                scanf("%d", &busID);
                printf("Enter Bus Type (Express/Regular): ");
                scanf("%s", busType);
                printf("Enter Arrival Time (minutes since midnight): ");
                scanf("%d", &arrivalTime);
                printf("Enter Departure Time (minutes since midnight): ");
                scanf("%d", &departureTime);

                enqueue(&q, busID, busType, arrivalTime, departureTime);
                break;
            }
            case 2:
                processBusArrival(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting the system.\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}

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

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

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

void enqueue(struct Queue *q, int busID, char busType[], int arrivalTime, int
departureTime) {
    struct Bus* newBus = (struct Bus*)malloc(sizeof(struct Bus));
    newBus->busID = busID;
    strcpy(newBus->busType, busType);
    newBus->arrivalTime = arrivalTime;
    newBus->departureTime = departureTime;
    newBus->next = NULL;

    if (isEmpty(q)) {
        q->front = newBus;
        q->rear = newBus;
    } else {
        q->rear->next = newBus;
        q->rear = newBus;
    }

    // Prioritize Express buses
    prioritizeExpress(q);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Bus* temp = q->front;
    q->front = q->front->next;
    free(temp);

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

```
}

void prioritizeExpress(struct Queue *q) {
    // If the first bus in the queue is a regular bus and there's an express bus,
    // we should move the express bus to the front
    if (isEmpty(q)) return;

    struct Bus* current = q->front;
    struct Bus* prev = NULL;

    while (current != NULL) {
        if (strcmp(current->busType, "Express") == 0) {
            if (prev != NULL) {
                prev->next = current->next;
                current->next = q->front;
                q->front = current;
                if (current->next == NULL) {
                    q->rear = current;
                }
            }
            break;
        }
        prev = current;
        current = current->next;
    }
}

void processBusArrival(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Bus* bus = q->front;
    printf("Processing Bus ID: %d, Type: %s, Arrival Time: %d, Departure Time: %d\n",
           bus->busID, bus->busType, bus->arrivalTime, bus->departureTime);

    dequeue(q);
}

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

```
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Bus* temp = q->front;
    printf("Current Bus Queue:\n");
    while (temp != NULL) {
        printf("Bus ID: %d, Type: %s, Arrival Time: %d, Departure Time: %d\n",
               temp->busID, temp->busType, temp->arrivalTime, temp->departureTime);
        temp = temp->next;
    }
}
```

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

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

#define SIZE 10

struct Person {
    int id;
    char name[50];
    char type[10];
};

struct Queue {
    struct Person person[SIZE];
    int front;
    int rear;
};

void create(struct Queue *q);
int isFull(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, struct Person p);
```

```
void dequeue(struct Queue *q);
void display(struct Queue *q);
void processVIP(struct Queue *vipQueue, struct Queue *generalQueue);

int main() {
    struct Queue vipQueue, generalQueue;
    create(&vipQueue);
    create(&generalQueue);
    int choice;

    while (1) {
        printf("\n1. Add Person (Entry)\n2. Process Exit\n3. Display Queue\n4. Exit
System\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                struct Person p;
                printf("Enter Person ID: ");
                scanf("%d", &p.id);
                printf("Enter Person Name: ");
                scanf(" %[^\n]", p.name);
                printf("Enter Person Type (VIP/General): ");
                scanf("%s", p.type);

                if (strcmp(p.type, "VIP") == 0) {
                    enqueue(&vipQueue, p);
                } else {
                    enqueue(&generalQueue, p);
                }
                break;
            }
            case 2: {
                processVIP(&vipQueue, &generalQueue);
                break;
            }
            case 3: {
                printf("\nVIP Queue:\n");
                display(&vipQueue);
                printf("\nGeneral Queue:\n");
```

```
            display(&generalQueue);
            break;
        }
        case 4:
            printf("Exiting the system.\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}

void create(struct Queue *q) {
   q->front = 0;
   q->rear = 0;
}

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

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

void enqueue(struct Queue *q, struct Person p) {
   if (isFull(q)) {
      printf("Queue is full. Cannot add more people.\n");
      return;
   }
   q->person[q->rear++] = p;
}

void dequeue(struct Queue *q) {
   if (isEmpty(q)) {
      printf("Queue is empty. No one to exit.\n");
      return;
   }
   printf("Person exiting: %s (ID: %d, Type: %s)\n", q->person[q->front].name,
```

```
q->person[q->front].id, q->person[q->front].type);
    q->front++;
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Current Queue:\n");
    for (int i = q->front; i < q->rear; i++) {
        printf("ID: %d, Name: %s, Type: %s\n", q->person[i].id, q->person[i].name,
q->person[i].type);
    }
}

void processVIP(struct Queue *vipQueue, struct Queue *generalQueue) {
    if (!isEmpty(vipQueue)) {
        dequeue(vipQueue);
    } else if (!isEmpty(generalQueue)) {
        dequeue(generalQueue);
    } else {
        printf("No one is left in the rally.\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>

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

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

void initializeQueue(struct Queue* q);
int isEmpty(struct Queue* q);
void enqueue(struct Queue* q, int transactionID, char* type, double amount);
void dequeue(struct Queue* q);
void display(struct Queue* q);
void processTransaction(struct Queue* q);

int main() {
    struct Queue transactionQueue;
    initializeQueue(&transactionQueue);

    int choice;
    while (1) {
        printf("\n1. Add Transaction\n2. Process Transaction\n3. Display Transactions\n4.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                int transactionID;
                char type[20];
                double amount;

                printf("Enter Transaction ID: ");
                scanf("%d", &transactionID);
                printf("Enter Transaction Type (Deposit/Withdrawal/Transfer): ");
                scanf("%s", type);
                printf("Enter Amount: ");
                scanf("%lf", &amount);

                enqueue(&transactionQueue, transactionID, type, amount);
                break;
            }
            case 2:
                processTransaction(&transactionQueue);
```

```
            break;
        case 3:
            display(&transactionQueue);
            break;
        case 4:
            printf("Exiting the system.\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}

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

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

void enqueue(struct Queue* q, int transactionID, char* type, double amount) {
   struct Transaction* newTransaction = (struct Transaction*)malloc(sizeof(struct
Transaction));
   newTransaction->transactionID = transactionID;
   snprintf(newTransaction->type, sizeof(newTransaction->type), "%s", type);
   newTransaction->amount = amount;
   newTransaction->next = NULL;

   if (isEmpty(q)) {
      q->front = newTransaction;
      q->rear = newTransaction;
   } else {
      q->rear->next = newTransaction;
      q->rear = newTransaction;
   }
}
```

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

void display(struct Queue* q) {
    if (isEmpty(q)) {
        printf("No transactions in the queue.\n");
        return;
    }
    struct Transaction* temp = q->front;
    printf("Transactions in Queue:\n");
    while (temp != NULL) {
        printf("ID: %d, Type: %s, Amount: %.2f\n", temp->transactionID, temp->type,
temp->amount);
        temp = temp->next;
    }
}

void processTransaction(struct Queue* q) {
    if (!isEmpty(q)) {
        dequeue(q);
    } else {
        printf("No transactions to process.\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 <stdlib.h>
```

```
#include <string.h>

#define SIZE 10

struct Voter {
    int id;
    char name[50];
    int age;
    char hasVoted;  // 'Y' for voted, 'N' for not voted
};

struct Queue {
    struct Voter voters[SIZE];
    int front;
    int rear;
};

void initializeQueue(struct Queue *q);
int isFull(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, struct Voter v);
void dequeue(struct Queue *q);
void display(struct Queue *q);
void verifyVoter(struct Queue *q);

int main() {
    struct Queue voterQueue;
    initializeQueue(&voterQueue);

    int choice;
    while (1) {
        printf("\n1. Register Voter\n2. Verify Voter\n3. Display Voter Queue\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                struct Voter v;
                printf("Enter Voter ID: ");
                scanf("%d", &v.id);
                printf("Enter Voter Name: ");
```

```
            scanf(" %[^\n]", v.name);
            printf("Enter Voter Age: ");
            scanf("%d", &v.age);
            v.hasVoted = 'N';

            enqueue(&voterQueue, v);
            break;
        }
        case 2:
            verifyVoter(&voterQueue);
            break;
        case 3:
            display(&voterQueue);
            break;
        case 4:
            printf("Exiting the system.\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}

void initializeQueue(struct Queue *q) {
   q->front = 0;
   q->rear = 0;
}

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

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

void enqueue(struct Queue *q, struct Voter v) {
   if (isFull(q)) {
      printf("Queue is full. Cannot register more voters.\n");
```

```
        return;
    }
    q->voters[q->rear++] = v;
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No voters to verify.\n");
        return;
    }
    printf("Voter exiting: %s (ID: %d, Age: %d)\n", q->voters[q->front].name,
q->voters[q->front].id, q->voters[q->front].age);
    q->front++;
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Current Voter Queue:\n");
    for (int i = q->front; i < q->rear; i++) {
        printf("ID: %d, Name: %s, Age: %d, Voted: %c\n", q->voters[i].id,
q->voters[i].name, q->voters[i].age, q->voters[i].hasVoted);
    }
}

void verifyVoter(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No voters in the queue.\n");
        return;
    }
    struct Voter v = q->voters[q->front];
    if (v.hasVoted == 'N') {
        printf("Voter verified: %s (ID: %d, Age: %d)\n", v.name, v.id, v.age);
        q->voters[q->front].hasVoted = 'Y';  // Mark the voter as having voted
        dequeue(q);  // Remove the verified voter from the queue
    } else {
        printf("This voter has already voted: %s (ID: %d, Age: %d)\n", v.name, v.id, v.age);
        dequeue(q);  // Remove the voter from the queue
    }
```

```
}
```

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

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

struct Patient {
    int id;
    char name[50];
    int severity;  // Higher number means more severe
    struct Patient *next;
};

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

void initializeQueue(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, struct Patient p);
void dequeue(struct Queue *q);
void display(struct Queue *q);

int main() {
    struct Queue erQueue;
    initializeQueue(&erQueue);

    int choice;
    while (1) {
        printf("\n1. Add Patient\n2. Process Patient\n3. Display Queue\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
```

```
            struct Patient p;
            printf("Enter Patient ID: ");
            scanf("%d", &p.id);
            printf("Enter Patient Name: ");
            scanf(" %[^\n]", p.name);
            printf("Enter Severity (1-10, 10 being most severe): ");
            scanf("%d", &p.severity);
            p.next = NULL;
            enqueue(&erQueue, p);
            break;
        }
        case 2:
            dequeue(&erQueue);
            break;
        case 3:
            display(&erQueue);
            break;
        case 4:
            printf("Exiting the system.\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  }

  return 0;
}

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

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

void enqueue(struct Queue *q, struct Patient p) {
  struct Patient *newPatient = (struct Patient *)malloc(sizeof(struct Patient));
  *newPatient = p;
```

```
    if (isEmpty(q)) {
        q->front = q->rear = newPatient;
    } else {
        struct Patient *temp = q->front;
        struct Patient *prev = NULL;
        while (temp != NULL && temp->severity >= p.severity) {
            prev = temp;
            temp = temp->next;
        }
        if (prev == NULL) {
            newPatient->next = q->front;
            q->front = newPatient;
        } else {
            prev->next = newPatient;
            newPatient->next = temp;
            if (temp == NULL) {
                q->rear = newPatient;
            }
        }
    }
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No patients in the queue.\n");
        return;
    }
    struct Patient *temp = q->front;
    printf("Processing Patient: %s (ID: %d, Severity: %d)\n", temp->name, temp->id,
temp->severity);
    q->front = q->front->next;
    free(temp);
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    struct Patient *temp = q->front;
    printf("Patients in Queue (Highest Severity First):\n");
```

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

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

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

#define SIZE 10

struct SurveyData {
    int id;
    char name[50];
    char response[200];
};

struct Queue {
    struct SurveyData data[SIZE];
    int front;
    int rear;
};

void initializeQueue(struct Queue *q);
int isFull(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, struct SurveyData s);
void dequeue(struct Queue *q);
void display(struct Queue *q);

int main() {
```

```
    struct Queue surveyQueue;
    initializeQueue(&surveyQueue);

    int choice;
    while (1) {
        printf("\n1. Add Survey Data\n2. Process Data\n3. Display Queue\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                struct SurveyData s;
                printf("Enter Survey ID: ");
                scanf("%d", &s.id);
                printf("Enter Surveyor Name: ");
                scanf(" %[^\n]", s.name);
                printf("Enter Survey Response: ");
                scanf(" %[^\n]", s.response);
                enqueue(&surveyQueue, s);
                break;
            }
            case 2:
                dequeue(&surveyQueue);
                break;
            case 3:
                display(&surveyQueue);
                break;
            case 4:
                printf("Exiting the system.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}

void initializeQueue(struct Queue *q) {
    q->front = 0;
    q->rear = 0;
```

```
}

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

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

void enqueue(struct Queue *q, struct SurveyData s) {
    if (isFull(q)) {
        printf("Queue is full. Cannot add more survey data.\n");
        return;
    }
    q->data[q->rear++] = s;
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Processing Survey Data: ID: %d, Name: %s, Response: %s\n",
q->data[q->front].id, q->data[q->front].name, q->data[q->front].response);
    q->front++;
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Survey Data Queue:\n");
    for (int i = q->front; i < q->rear; i++) {
        printf("ID: %d, Name: %s, Response: %s\n", q->data[i].id, q->data[i].name,
q->data[i].response);
    }
}
```

20. *Financial Market Data Analysis*: Use linked lists to implement a queue for

analyzing financial market data. The system should handle large volumes of data, perform real-time analysis, and generate insights for decision-making.

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

struct MarketData {
    int id;
    float price;
    char symbol[10];
    struct MarketData *next;
};

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

void initializeQueue(struct Queue *q);
int isEmpty(struct Queue *q);
void enqueue(struct Queue *q, struct MarketData m);
void dequeue(struct Queue *q);
void display(struct Queue *q);

int main() {
    struct Queue marketQueue;
    initializeQueue(&marketQueue);

    int choice;
    while (1) {
        printf("\n1. Add Market Data\n2. Process Data\n3. Display Data\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: {
                struct MarketData m;
                printf("Enter Market Data ID: ");
                scanf("%d", &m.id);
                printf("Enter Symbol: ");
```

```
            scanf("%s", m.symbol);
            printf("Enter Price: ");
            scanf("%f", &m.price);
            m.next = NULL;
            enqueue(&marketQueue, m);
            break;
        }
        case 2:
            dequeue(&marketQueue);
            break;
        case 3:
            display(&marketQueue);
            break;
        case 4:
            printf("Exiting the system.\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  }

  return 0;
}

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

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

void enqueue(struct Queue *q, struct MarketData m) {
  struct MarketData *newData = (struct MarketData *)malloc(sizeof(struct
MarketData));
  *newData = m;

  if (isEmpty(q)) {
    q->front = q->rear = newData;
  } else {
```

```
        q->rear->next = newData;
        q->rear = newData;
    }
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No data in the queue.\n");
        return;
    }
    struct MarketData *temp = q->front;
    printf("Processing Market Data: ID: %d, Symbol: %s, Price: %.2f\n", temp->id,
temp->symbol, temp->price);
    q->front = q->front->next;
    free(temp);
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    struct MarketData *temp = q->front;
    printf("Market Data Queue:\n");
    while (temp != NULL) {
        printf("ID: %d, Symbol: %s, Price: %.2f\n", temp->id, temp->symbol,
temp->price);
        temp = temp->next;
    }
}
```

Vaild parenthese

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 50

typedef struct{
```

```
    char type[SIZE];
    int top;
}Stack;


void push(Stack *stack,char type);
char pop(Stack *stack);
int isEmpty(Stack *stack);
int isValidParenthese(const char *str);

int main(){
    char str[100];
    printf("Enter a string witj Parenthses \n");
    scanf("%s",str);

    if(isValidParenthese(str)){
        printf("The parenthese is valid \n");
    }
    else{
        printf("Not valid \n");
    }
    return 0;
}

void push(Stack *stack,char type){
    if(stack->top >= SIZE-1){
        printf("Stack overflow\n");
    }
    stack->type[++stack->top] = type;
}
char pop(Stack *stack){
    if (isEmpty(stack)) {
        return '\0';
    }
    return stack->type[stack->top--];
}
int isEmpty(Stack *stack){

    return stack->top == -1;
}
int isValidParenthese(const char *str){
```

```
    Stack stack;
    stack.top = -1;

    for(int i=0;str[i] != '\0';i++){
        char ch = str[i];
        if(ch == '(' || ch == '{' || ch =='['){
            push(&stack,ch);
        }
        else if (ch == ')' || ch == '}' || ch == ']')
        {
            char top = pop(&stack);
            if((ch == ')' && top != '(')|| (ch == '}' && top != '{') || (ch == ']' && top != '[')){
                return 0;
            }


        }

    }
    return isEmpty(&stack);
}
```