1.

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

#define MAX_ORDERS 100

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

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

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

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

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

void enqueue(struct OrderQueue *q, struct Order order) {
    if (isFull(q)) {
        printf("Queue is full! Cannot place more orders.\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->orders[++q->rear] = order;
}

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

        if (buyOrder->price >= sellOrder->price) {
            int matchQuantity = buyOrder->quantity < sellOrder->quantity ? buyOrder->quantity : sellOrder->quantity;
            printf("Matched %d units at price %d\n", matchQuantity, sellOrder->price);
            buyOrder->quantity -= matchQuantity;
            sellOrder->quantity -= matchQuantity;
            if (buyOrder->quantity == 0) {
                buyQueue->front++;
            }
```

```
            if (sellOrder->quantity == 0) {
                sellQueue->front++;
            }
        } else {
            break;
        }
    }
}

int main() {
    struct OrderQueue buyQueue, sellQueue;
    initializeQueue(&buyQueue);
    initializeQueue(&sellQueue);

    struct Order buy1 = {"buy", 100, 10};
    struct Order buy2 = {"buy", 105, 5};
    struct Order sell1 = {"sell", 95, 8};
    struct Order sell2 = {"sell", 100, 7};

    enqueue(&buyQueue, buy1);
    enqueue(&buyQueue, buy2);
    enqueue(&sellQueue, sell1);
    enqueue(&sellQueue, sell2);

    printf("Matching orders:\n");
    matchOrders(&buyQueue, &sellQueue);

    return 0;
}
```

2.

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

struct Customer {
    char name[50];
    int priority;  // Higher number indicates higher priority
    struct Customer *next;
};

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

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

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

void enqueuePriority(struct Queue *q, char name[], int priority) {
    struct Customer *newCustomer = (struct Customer *)malloc(sizeof(struct Customer));
    strcpy(newCustomer->name, name);
    newCustomer->priority = priority;
```

```
    newCustomer->next = NULL;

    if (isQueueEmpty(q)) {
        q->front = q->rear = newCustomer;
    } else {
        struct Customer *current = q->front;
        struct Customer *previous = NULL;

        while (current != NULL && current->priority >= priority) {
            previous = current;
            current = current->next;
        }

        if (previous == NULL) {
            newCustomer->next = q->front;
            q->front = newCustomer;
        } else {
            previous->next = newCustomer;
            newCustomer->next = current;
        }

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

void serveCustomer(struct Queue *q) {
    if (isQueueEmpty(q)) {
```

```
        printf("No customers to serve!\n");
        return;
    }

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

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

    free(temp);
}

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

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

    printf("Customer service simulation:\n");
    serveCustomer(&queue);
    serveCustomer(&queue);
    serveCustomer(&queue);
    serveCustomer(&queue);
```

```
    return 0;
}
 3. 3. #include <stdio.h>
#include <string.h>

#define MAX_ATTENDEES 100

// General Attendee Queue
typedef struct {
    char names[MAX_ATTENDEES][50];
    int front;
    int rear;
} Queue;

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

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

int isQueueFull(Queue *q) {
    return q->rear == MAX_ATTENDEES - 1;
}

void enqueue(Queue *q, char name[]) {
    if (isQueueFull(q)) {
```

```
        printf("Queue is full. Cannot register more attendees.\n");
        return;
    }
    if (isQueueEmpty(q)) {
        q->front = 0;
    }
    q->rear++;
    strcpy(q->names[q->rear], name);
    printf("Registered: %s\n", name);
}

char* dequeue(Queue *q) {
    if (isQueueEmpty(q)) {
        printf("No attendees in queue.\n");
        return NULL;
    }
    char *name = q->names[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return name;
}

int main() {
    Queue generalQueue, vipQueue;
    initializeQueue(&generalQueue);
    initializeQueue(&vipQueue);
```

```
int choice;
char name[50];

while (1) {
    printf("\n--- Political Campaign Event Management ---\n");
    printf("1. Register General Attendee\n");
    printf("2. Register VIP Attendee\n");
    printf("3. Check-In Attendee\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    getchar();  // Consume newline character

    switch (choice) {
        case 1:
            if (!isQueueFull(&generalQueue)) {
                printf("Enter General Attendee Name: ");
                fgets(name, sizeof(name), stdin);
                name[strcspn(name, "\n")] = 0;  // Remove newline character
                enqueue(&generalQueue, name);
            }
            break;
        case 2:
            if (!isQueueFull(&vipQueue)) {
                printf("Enter VIP Attendee Name: ");
                fgets(name, sizeof(name), stdin);
                name[strcspn(name, "\n")] = 0;  // Remove newline character
                enqueue(&vipQueue, name);
```

```
            }
            break;
        case 3:
            if (!isQueueEmpty(&vipQueue)) {
                printf("VIP Attendee Checked-In: %s\n", dequeue(&vipQueue));
            } else if (!isQueueEmpty(&generalQueue)) {
                printf("General Attendee Checked-In: %s\n", dequeue(&generalQueue));
            } else {
                printf("No attendees in queue.\n");
            }
            break;
        case 4:
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}
```

4. 4.

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

#define MAX_TELLERS 3
#define MAX_TRANSACTION_TIME 10

// Customer structure for the linked list
```

```
typedef struct Customer {
    int id;
    int transactionTime;
    struct Customer* next;
} Customer;

// Queue structure for bank customers
typedef struct Queue {
    Customer* front;
    Customer* rear;
    int count;
} Queue;

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

int isQueueEmpty(Queue* q) {
    return q->count == 0;
}

void enqueue(Queue* q, int customerId, int transactionTime) {
    Customer* newCustomer = (Customer*)malloc(sizeof(Customer));
    newCustomer->id = customerId;
    newCustomer->transactionTime = transactionTime;
    newCustomer->next = NULL;
```

```
    if (isQueueEmpty(q)) {
        q->front = newCustomer;
    } else {
        q->rear->next = newCustomer;
    }
    q->rear = newCustomer;
    q->count++;
    printf("Customer %d added with transaction time %d minutes.\n", customerId, transactionTime);
}

Customer* dequeue(Queue* q) {
    if (isQueueEmpty(q)) {
        return NULL;
    }
    Customer* customer = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    q->count--;
    return customer;
}

void simulateBankQueue() {
    Queue queue;
    initializeQueue(&queue);
    srand(time(NULL));

    int customerCount = 0;
```

```
int tellers[MAX_TELLERS] = {0};  // Track transaction time remaining for each teller

for (int time = 0; time < 30; time++) {  // Simulate 30 minutes of bank operation
    printf("\nMinute %d:\n", time + 1);

    // Random customer arrival (30% chance of arrival)
    if (rand() % 100 < 30) {
        int transactionTime = rand() % MAX_TRANSACTION_TIME + 1;
        enqueue(&queue, ++customerCount, transactionTime);
    }

    // Process each teller
    for (int i = 0; i < MAX_TELLERS; i++) {
        if (tellers[i] == 0) {
            Customer* nextCustomer = dequeue(&queue);
            if (nextCustomer) {
                printf("Teller %d is now serving Customer %d (Transaction time: %d minutes).\n",
                       i + 1, nextCustomer->id, nextCustomer->transactionTime);
                tellers[i] = nextCustomer->transactionTime;
                free(nextCustomer);
            } else {
                printf("Teller %d is idle.\n", i + 1);
            }
        } else {
            tellers[i]--;  // Decrease the remaining transaction time
            printf("Teller %d is busy, %d minute(s) remaining.\n", i + 1, tellers[i]);
        }
    }
}
```

```
}

int main() {
   printf("Bank Teller Simulation\n");
   simulateBankQueue();
   return 0;
}
5. 5. #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define MAX_SIZE 1000  // Maximum size of the queue

// Structure to hold financial data feed
typedef struct {
   char symbol[10];     // Financial instrument symbol (e.g., "AAPL", "GOOGL")
   double price;        // Price of the instrument
   int volume;          // Volume traded
   char timestamp[20];   // Timestamp (e.g., "2025-01-21 14:35:00")
} DataFeed;

// Queue structure
typedef struct {
   DataFeed data[MAX_SIZE];
   int front;
   int rear;
   int size;
} Queue;
```

```
// Initialize the queue
void initializeQueue(Queue *q) {
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}

// Check if the queue is empty
bool isEmpty(Queue *q) {
    return q->size == 0;
}

// Check if the queue is full
bool isFull(Queue *q) {
    return q->size == MAX_SIZE;
}

// Enqueue a data feed into the queue
bool enqueue(Queue *q, DataFeed feed) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue data.\n");
        return false;
    }
    q->rear = (q->rear + 1) % MAX_SIZE;
    q->data[q->rear] = feed;
    q->size++;
    return true;
}
```

```
// Dequeue a data feed from the queue
bool dequeue(Queue *q, DataFeed *feed) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue data.\n");
        return false;
    }
    *feed = q->data[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    q->size--;
    return true;
}

// Display the queue contents (for debugging)
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Data feeds in queue:\n");
    int count = q->size;
    int index = q->front;
    while (count > 0) {
        DataFeed feed = q->data[index];
        printf("Symbol: %s, Price: %.2f, Volume: %d, Timestamp: %s\n",
               feed.symbol, feed.price, feed.volume, feed.timestamp);
        index = (index + 1) % MAX_SIZE;
        count--;
    }
```

```
}

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

    // Sample data feed inputs
    DataFeed feed1 = {"AAPL", 150.25, 1000, "2025-01-21 14:35:00"};
    DataFeed feed2 = {"GOOGL", 2750.50, 500, "2025-01-21 14:36:00"};
    DataFeed feed3 = {"MSFT", 300.00, 1200, "2025-01-21 14:37:00"};

    // Enqueue data feeds
    enqueue(&dataQueue, feed1);
    enqueue(&dataQueue, feed2);
    enqueue(&dataQueue, feed3);

    // Display queue
    displayQueue(&dataQueue);

    // Process (dequeue) data feeds
    DataFeed processedFeed;
    while (dequeue(&dataQueue, &processedFeed)) {
        printf("Processing feed - Symbol: %s, Price: %.2f, Volume: %d, Timestamp: %s\n",
            processedFeed.symbol, processedFeed.price, processedFeed.volume,
processedFeed.timestamp);
    }

    return 0;
}
```

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

// Define the structure for a car
struct Car {
   char plate_number[15];  // Car plate number
   struct Car* next;       // Pointer to next car in the queue
};

// Define the structure for the queue (traffic light queue)
struct Queue {
   struct Car* front;
   struct Car* rear;
};

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

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

// Enqueue (Add a car to the queue)
void enqueue(struct Queue* q, const char* plate_number) {
```

```
    struct Car* newCar = (struct Car*)malloc(sizeof(struct Car));
    strcpy(newCar->plate_number, plate_number);
    newCar->next = NULL;

    if (isEmpty(q)) {
        q->front = newCar;
        q->rear = newCar;
    } else {
        q->rear->next = newCar;
        q->rear = newCar;
    }
    printf("Car %s added to the queue.\n", plate_number);
}

// Dequeue (Process a car from the queue)
void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("No cars in the queue.\n");
        return;
    }

    struct Car* temp = q->front;
    printf("Car %s is passing through the green light.\n", temp->plate_number);
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL;  // If queue becomes empty, reset rear
    }
```

```
    free(temp);
}

// Simulate traffic light change (Red to Green)
void simulateTrafficLight(struct Queue* q) {
    printf("Traffic light is red. Waiting for cars to arrive...\n");
    while (!isEmpty(q)) {
        dequeue(q);  // Process cars one by one
    }
    printf("Traffic light is now green.\n");
}

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

    // Simulating car arrivals at different times
    enqueue(&trafficQueue, "ABC123");
    enqueue(&trafficQueue, "XYZ456");
    enqueue(&trafficQueue, "LMN789");

    // Simulate the traffic light change and process cars
    simulateTrafficLight(&trafficQueue);

    return 0;
}
```

7. 7, #include <stdio.h>

#include <stdlib.h>

```
#define MAX_VOTES 1000  // Maximum number of votes

// Define the structure for a vote
struct Vote {
    int station_id;  // Polling station ID
    char candidate[50];  // Candidate name
};

// Define the queue structure for vote counting
struct VoteQueue {
    struct Vote votes[MAX_VOTES];
    int front;
    int rear;
};

// Initialize the vote queue
void initializeQueue(struct VoteQueue* queue) {
    queue->front = 0;
    queue->rear = -1;
}

// Check if the queue is empty
int isEmpty(struct VoteQueue* queue) {
    return queue->rear < queue->front;
}

// Check if the queue is full
int isFull(struct VoteQueue* queue) {
    return queue->rear == MAX_VOTES - 1;
```

```
}

// Enqueue a vote into the queue
void enqueue(struct VoteQueue* queue, int station_id, const char* candidate) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue more votes.\n");
        return;
    }
    queue->rear++;
    queue->votes[queue->rear].station_id = station_id;
    snprintf(queue->votes[queue->rear].candidate, sizeof(queue->votes[queue->rear].candidate), "%s",
candidate);
    printf("Vote for %s from Station %d added.\n", candidate, station_id);
}

// Dequeue (count a vote)
void dequeue(struct VoteQueue* queue) {
    if (isEmpty(queue)) {
        printf("No votes to count.\n");
        return;
    }
    struct Vote vote = queue->votes[queue->front];
    printf("Counting vote for %s from Station %d\n", vote.candidate, vote.station_id);
    queue->front++;
}

// Simulate vote counting process
void processVotes(struct VoteQueue* queue) {
    while (!isEmpty(queue)) {
```

```
        dequeue(queue);  // Count each vote
    }
}

int main() {
    struct VoteQueue electionQueue;
    initializeQueue(&electionQueue);

    // Simulating votes from different polling stations
    enqueue(&electionQueue, 1, "Alice");
    enqueue(&electionQueue, 2, "Bob");
    enqueue(&electionQueue, 1, "Alice");
    enqueue(&electionQueue, 3, "Charlie");

    // Process the votes in order they were received
    printf("\nStarting vote count...\n");
    processVotes(&electionQueue);

    return 0;
}
```

8. 8. #include <stdio.h>

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

#define MAX_ID_LEN 10

// Define the structure for an airplane
struct Airplane {
    char id[MAX_ID_LEN];     // Airplane ID (e.g., flight number)
```

```
    char type[20];          // Type of operation (land or take off)
    int priority;           // Priority for emergency landings (higher means more priority)
    struct Airplane* next;   // Pointer to the next airplane in the queue
};

// Define the structure for the queue (runway queue)
struct Queue {
    struct Airplane* front;
    struct Airplane* rear;
};

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

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

// Enqueue (Add an airplane to the queue)
void enqueue(struct Queue* q, const char* id, const char* type, int priority) {
    struct Airplane* newAirplane = (struct Airplane*)malloc(sizeof(struct Airplane));
    strcpy(newAirplane->id, id);
    strcpy(newAirplane->type, type);
    newAirplane->priority = priority;
    newAirplane->next = NULL;
```

```
    // If queue is empty, set front and rear to the new airplane
    if (isEmpty(q)) {
        q->front = newAirplane;
        q->rear = newAirplane;
    } else {
        // Add the new airplane based on priority
        struct Airplane* current = q->front;
        struct Airplane* prev = NULL;

        while (current != NULL && current->priority >= priority) {
            prev = current;
            current = current->next;
        }

        if (prev == NULL) {
            newAirplane->next = q->front;
            q->front = newAirplane;
        } else if (current == NULL) {
            prev->next = newAirplane;
            q->rear = newAirplane;
        } else {
            newAirplane->next = current;
            prev->next = newAirplane;
        }
    }
    printf("Airplane %s with type %s added to the queue.\n", id, type);
}
```

```
// Dequeue (Allow an airplane to land or take off)
void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("No airplanes in the queue.\n");
        return;
    }

    struct Airplane* temp = q->front;
    printf("Airplane %s with type %s is now allowed to %s.\n", temp->id, temp->type, temp->type);
    q->front = q->front->next;

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

    free(temp);
}

// Simulate the runway management system
void manageRunway(struct Queue* q) {
    printf("\nStarting runway management...\n");

    while (!isEmpty(q)) {
        dequeue(q);  // Allow airplanes to land or take off based on priority
    }
}

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

```
    initializeQueue(&runwayQueue);

    // Simulating airplane arrivals
    enqueue(&runwayQueue, "AA123", "land", 1);  // Emergency landing (high priority)
    enqueue(&runwayQueue, "BB456", "take off", 3); // Regular take off
    enqueue(&runwayQueue, "CC789", "land", 2);  // Emergency landing (medium priority)

    // Manage the runway (process airplanes)
    manageRunway(&runwayQueue);

    return 0;
}
9. 9. #include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ORDERS 100  // Maximum number of orders in the queue

// Define the structure for a trading order
struct Order {
    char type[5];  // Type of order (buy or sell)
    char symbol[10]; // Stock symbol (e.g., AAPL, GOOGL)
    int quantity;    // Quantity of stocks
    float price;     // Price of each stock
};

// Define the queue structure for stock orders
struct OrderQueue {
    struct Order orders[MAX_ORDERS];
```

```
    int front;
    int rear;
};

// Initialize the order queue
void initializeQueue(struct OrderQueue* queue) {
    queue->front = 0;
    queue->rear = -1;
}

// Check if the queue is empty
int isEmpty(struct OrderQueue* queue) {
    return queue->rear < queue->front;
}

// Check if the queue is full
int isFull(struct OrderQueue* queue) {
    return queue->rear == MAX_ORDERS - 1;
}

// Enqueue an order
void enqueue(struct OrderQueue* queue, const char* type, const char* symbol, int quantity, float price)
{
    if (isFull(queue)) {
        printf("Order queue is full. Cannot enqueue more orders.\n");
        return;
    }
    queue->rear++;
    strcpy(queue->orders[queue->rear].type, type);
```

```
    strcpy(queue->orders[queue->rear].symbol, symbol);
    queue->orders[queue->rear].quantity = quantity;
    queue->orders[queue->rear].price = price;
    printf("Order to %s %d shares of %s at %.2f added.\n", type, quantity, symbol, price);
}

// Dequeue (process an order)
void dequeue(struct OrderQueue* queue) {
    if (isEmpty(queue)) {
        printf("No orders to process.\n");
        return;
    }

    struct Order order = queue->orders[queue->front];
    printf("Processing %s order: %d shares of %s at %.2f\n", order.type, order.quantity, order.symbol,
order.price);
    queue->front++;
}

// Cancel an order (remove an order from the queue)
void cancelOrder(struct OrderQueue* queue, int index) {
    if (index < queue->front || index > queue->rear) {
        printf("Invalid order index.\n");
        return;
    }

    for (int i = index; i < queue->rear; i++) {
        queue->orders[i] = queue->orders[i + 1];
    }
```

```
    queue->rear--;
    printf("Order at index %d has been cancelled.\n", index);
}

// Simulate the trading system
void processOrders(struct OrderQueue* queue) {
    printf("\nProcessing orders...\n");

    while (!isEmpty(queue)) {
        dequeue(queue);  // Process each order
    }
}

int main() {
    struct OrderQueue tradingQueue;
    initializeQueue(&tradingQueue);

    // Simulate adding orders
    enqueue(&tradingQueue, "buy", "AAPL", 100, 150.50);
    enqueue(&tradingQueue, "sell", "GOOGL", 50, 2750.75);
    enqueue(&tradingQueue, "buy", "TSLA", 30, 650.25);

    // Process orders
    processOrders(&tradingQueue);

    // Simulate order cancellation
    enqueue(&tradingQueue, "buy", "AAPL", 100, 150.50);
    cancelOrder(&tradingQueue, 0);  // Cancel the first order
```

```
    // Process remaining orders
    processOrders(&tradingQueue);

    return 0;
}
10. 10. #include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_NAME_LEN 50

// Define the structure for a registrant
struct Registrant {
    char name[MAX_NAME_LEN];  // Name of the registrant
    char type[15];           // Type of registration (walk-in or pre-registration)
    struct Registrant* next;  // Pointer to the next registrant in the queue
};

// Define the structure for the registration queue
struct Queue {
    struct Registrant* front;
    struct Registrant* rear;
};

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

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

// Enqueue (Add a registrant to the queue)
void enqueue(struct Queue* q, const char* name, const char* type) {
    struct Registrant* newRegistrant = (struct Registrant*)malloc(sizeof(struct Registrant));
    strcpy(newRegistrant->name, name);
    strcpy(newRegistrant->type, type);
    newRegistrant->next = NULL;

    // If the queue is empty, set front and rear to the new registrant
    if (isEmpty(q)) {
        q->front = newRegistrant;
        q->rear = newRegistrant;
    } else {
        q->rear->next = newRegistrant;
        q->rear = newRegistrant;
    }
    printf("%s (Type: %s) has been added to the registration queue.\n", name, type);
}

// Dequeue (Process a registrant from the queue)
void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("No registrants in the queue.\n");
        return;
```

```
    }

    struct Registrant* temp = q->front;
    printf("%s (Type: %s) has completed registration and is entering the conference.\n", temp->name, temp->type);
    q->front = q->front->next;

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

    free(temp);
}

// Cancel a registration (Remove a registrant from the queue)
void cancelRegistration(struct Queue* q, const char* name) {
    if (isEmpty(q)) {
        printf("No registrants to cancel.\n");
        return;
    }

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

    // Search for the registrant
    while (current != NULL && strcmp(current->name, name) != 0) {
        prev = current;
        current = current->next;
    }
```

```
    if (current == NULL) {
        printf("Registrant with name %s not found.\n", name);
        return;
    }

    // If the registrant is the front of the queue
    if (prev == NULL) {
        q->front = current->next;
    } else {
        prev->next = current->next;
    }

    if (current == q->rear) {
        q->rear = prev;  // If the registrant is the rear
    }

    printf("%s's registration has been cancelled.\n", current->name);
    free(current);
}

// Display all registrations in the queue
void displayRegistrants(struct Queue* q) {
    if (isEmpty(q)) {
        printf("No registrants in the queue.\n");
        return;
    }

    struct Registrant* current = q->front;
```

```
    printf("\nRegistrant List:\n");
    while (current != NULL) {
        printf("Name: %s, Type: %s\n", current->name, current->type);
        current = current->next;
    }
}

// Simulate the conference registration system
void manageRegistrations(struct Queue* q) {
    printf("\nProcessing registrations...\n");

    while (!isEmpty(q)) {
        dequeue(q);  // Process each registrant
    }
}

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

    // Simulate pre-registrations and walk-ins
    enqueue(&registrationQueue, "Alice", "Pre-Registration");
    enqueue(&registrationQueue, "Bob", "Walk-In");
    enqueue(&registrationQueue, "Charlie", "Pre-Registration");

    // Display all registrations
    displayRegistrants(&registrationQueue);

    // Simulate a cancellation
```

```
    cancelRegistration(&registrationQueue, "Bob");

    // Display remaining registrations
    displayRegistrants(&registrationQueue);

    // Process all remaining registrations
    manageRegistrations(&registrationQueue);

    return 0;
}
```

11.

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

#define MAX_AUDIENCE 100
#define MAX_NAME_LEN 50

struct Audience {
    char name[MAX_NAME_LEN];
    char role[20];  // Regular or Media
};

struct Queue {
    struct Audience audience[MAX_AUDIENCE];
    int front, rear;
};

void initializeQueue(struct Queue *q) {
    q->front = 0;
    q->rear = -1;
```

```
}

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

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

void enqueue(struct Queue *q, char *name, char *role) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    q->rear++;
    strcpy(q->audience[q->rear].name, name);
    strcpy(q->audience[q->rear].role, role);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("%s (Role: %s) is allowed into the debate room.\n",
          q->audience[q->front].name, q->audience[q->front].role);
    q->front++;
}
```

```
int main() {
    struct Queue debateQueue;
    initializeQueue(&debateQueue);

    enqueue(&debateQueue, "Alice", "Regular");
    enqueue(&debateQueue, "Bob", "Media");
    enqueue(&debateQueue, "Charlie", "Regular");

    dequeue(&debateQueue);  // Alice enters
    dequeue(&debateQueue);  // Bob enters (media personnel have priority)

    return 0;
}
```

12.
```
#include <stdio.h>
#include <stdlib.h>

struct LoanApplication {
    char name[50];
    int loanAmount;
    int creditScore;
    struct LoanApplication *next;
};

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

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

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

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

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

    while (current != NULL && current->loanAmount >= loanAmount && current->creditScore >=
creditScore) {
        prev = current;
        current = current->next;
    }

    if (prev == NULL) {
        newApp->next = q->front;
        q->front = newApp;
```

```
    } else {
        newApp->next = current;
        prev->next = newApp;
    }
}

void dequeue(struct Queue *q) {
    if (q->front == NULL) {
        printf("No loan applications to process.\n");
        return;
    }

    struct LoanApplication *temp = q->front;
    printf("Processing loan application for %s: Loan Amount: %d, Credit Score: %d\n",
           temp->name, temp->loanAmount, temp->creditScore);
    q->front = q->front->next;
    free(temp);
}

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

    enqueue(&loanQueue, "Alice", 100000, 750);
    enqueue(&loanQueue, "Bob", 50000, 650);
    enqueue(&loanQueue, "Charlie", 150000, 800);

    dequeue(&loanQueue);  // Process Charlie's application (highest priority)
    dequeue(&loanQueue);  // Process Alice's application
```

```
    dequeue(&loanQueue);  // Process Bob's application

    return 0;
}
```

13.
```
#include <stdio.h>
#include <string.h>

#define MAX_CUSTOMERS 5
#define MAX_NAME_LEN 50

struct Customer {
    char name[MAX_NAME_LEN];
    int cartValue;
};

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

void initializeQueue(struct Queue *q) {
    q->front = 0;
    q->rear = -1;
}

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

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

void enqueue(struct Queue *q, char *name, int cartValue) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    q->rear++;
    strcpy(q->customers[q->rear].name, name);
    q->customers[q->rear].cartValue = cartValue;
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("%s has completed checkout. Cart value: $%d\n",
           q->customers[q->front].name, q->customers[q->front].cartValue);
    q->front++;
}

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

    enqueue(&checkoutQueue, "Alice", 150);
```

```
    enqueue(&checkoutQueue, "Bob", 200);
    enqueue(&checkoutQueue, "Charlie", 120);

    dequeue(&checkoutQueue);  // Alice checks out
    dequeue(&checkoutQueue);  // Bob checks out

    return 0;
}
14. #include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Bus {
    char busName[50];
    int arrivalTime;  // Time in minutes (example: 830 for 8:30 AM)
    int isExpress;    // 1 for express, 0 for regular
    struct Bus *next;
};

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

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, char *busName, int arrivalTime, int isExpress) {
    struct Bus newBus = (struct Bus)malloc(sizeof(struct Bus));
    strcpy(newBus->busName, busName);
    newBus->arrivalTime = arrivalTime;
    newBus->isExpress = isExpress;
    newBus->next = NULL;

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

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No buses to process.\n");
        return;
    }
```

```
    struct Bus *temp = q->front;
    printf("Bus %s (Arrival: %d) is departing.\n", temp->busName, temp->arrivalTime);
    q->front = q->front->next;
    free(temp);
}

void printQueue(struct Queue *q) {
    struct Bus *current = q->front;
    while (current != NULL) {
        printf("Bus %s (Arrival: %d) - %s\n", current->busName, current->arrivalTime,
               current->isExpress ? "Express" : "Regular");
        current = current->next;
    }
}

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

    // Enqueue buses (Express buses prioritized during peak hours)
    enqueue(&busQueue, "Bus A", 800, 0); // Regular bus at 8:00 AM
    enqueue(&busQueue, "Bus B", 830, 1); // Express bus at 8:30 AM
    enqueue(&busQueue, "Bus C", 845, 0); // Regular bus at 8:45 AM
    enqueue(&busQueue, "Bus D", 900, 1); // Express bus at 9:00 AM

    // Print the current queue
    printf("Buses in the queue:\n");
    printQueue(&busQueue);
```

```
    // Process buses (departure)
    printf("\nProcessing buses:\n");
    dequeue(&busQueue);  // First express bus (Bus B)
    dequeue(&busQueue);  // Next bus (Bus A)

    return 0;
}
```

15.

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

#define MAX_CROWD 100
#define MAX_NAME_LEN 50

struct Person {
    char name[MAX_NAME_LEN];
    int isVIP;  // 1 for VIP, 0 for regular
};

struct Queue {
    struct Person crowd[MAX_CROWD];
    int front, rear;
};

void initializeQueue(struct Queue *q) {
    q->front = 0;
    q->rear = -1;
}
```

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

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

void enqueue(struct Queue *q, char *name, int isVIP) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    q->rear++;
    strcpy(q->crowd[q->rear].name, name);
    q->crowd[q->rear].isVIP = isVIP;
}

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

    struct Person person = q->crowd[q->front];
    printf("%s (VIP: %d) is exiting the rally.\n", person.name, person.isVIP);
    q->front++;
}
```

```
void printQueue(struct Queue *q) {
    printf("Current crowd in the rally:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%s (VIP: %d)\n", q->crowd[i].name, q->crowd[i].isVIP);
    }
}

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

    // Enqueue people (VIPs have priority in entry)
    enqueue(&rallyQueue, "Alice", 0);  // Regular person
    enqueue(&rallyQueue, "Bob", 1);    // VIP person
    enqueue(&rallyQueue, "Charlie", 0); // Regular person
    enqueue(&rallyQueue, "David", 1);  // VIP person

    // Print the current queue (before exit)
    printQueue(&rallyQueue);

    // Process exits (dequeue)
    printf("\nProcessing exits:\n");
    dequeue(&rallyQueue);  // Bob (VIP) exits first
    dequeue(&rallyQueue);  // Alice (Regular) exits
    dequeue(&rallyQueue);  // Charlie (Regular) exits

    return 0;
}
16. #include <stdio.h>
```

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

struct Transaction {
   char type[20];  // Type of transaction (deposit, withdrawal, transfer)
   float amount;   // Transaction amount
   char source[50]; // Source (for withdrawal or transfer)
   char destination[50]; // Destination (for transfer)
   struct Transaction *next;  // Pointer to the next transaction
};

struct Queue {
   struct Transaction *front;  // Front of the queue
   struct Transaction *rear;   // Rear of the queue
};

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, char *type, float amount, char *source, char *destination) {
   struct Transaction newTransaction = (struct Transaction)malloc(sizeof(struct Transaction));
   strcpy(newTransaction->type, type);
   newTransaction->amount = amount;
```

```
    strcpy(newTransaction->source, source);
    strcpy(newTransaction->destination, destination);
    newTransaction->next = NULL;

    if (isEmpty(q)) {
        q->front = 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: %s\n", temp->type);
    printf("Amount: %.2f\n", temp->amount);
    if (strcmp(temp->type, "transfer") == 0) {
        printf("From: %s\n", temp->source);
        printf("To: %s\n", temp->destination);
    } else {
        printf("Source: %s\n", temp->source);
    }

    q->front = q->front->next;
```

```
    free(temp);
}

void printQueue(struct Queue *q) {
    struct Transaction *current = q->front;
    while (current != NULL) {
        printf("Transaction Type: %s, Amount: %.2f\n", current->type, current->amount);
        current = current->next;
    }
}

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

    // Enqueue transactions
    enqueue(&transactionQueue, "deposit", 1000.00, "Account1", "");
    enqueue(&transactionQueue, "withdrawal", 500.00, "Account2", "");
    enqueue(&transactionQueue, "transfer", 300.00, "Account3", "Account4");

    // Print the queue of transactions
    printf("Queue of transactions:\n");
    printQueue(&transactionQueue);

    // Process the transactions
    printf("\nProcessing transactions:\n");
    dequeue(&transactionQueue);  // Process the first transaction (deposit)
    dequeue(&transactionQueue);  // Process the second transaction (withdrawal)
    dequeue(&transactionQueue);  // Process the third transaction (transfer)
```

```
    return 0;
}
17. #include <stdio.h>
#include <string.h>

#define MAX_VOTERS 100
#define MAX_NAME_LEN 50

struct Voter {
    char name[MAX_NAME_LEN];
    int id;  // Voter ID
};

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

void initializeQueue(struct Queue *q) {
    q->front = 0;
    q->rear = -1;
}

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

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

```
    return q->front > q->rear;
}

void enqueue(struct Queue *q, char *name, int id) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    q->rear++;
    strcpy(q->voters[q->rear].name, name);
    q->voters[q->rear].id = id;
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No voters in the queue.\n");
        return;
    }

    struct Voter voter = q->voters[q->front];
    printf("Voter %d (%s) is casting their vote.\n", voter.id, voter.name);
    q->front++;
}

void printQueue(struct Queue *q) {
    printf("Voter queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("Voter ID: %d, Name: %s\n", q->voters[i].id, q->voters[i].name);
    }
```

```
}

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

    // Register voters (enqueue)
    enqueue(&pollingQueue, "Alice", 1001);
    enqueue(&pollingQueue, "Bob", 1002);
    enqueue(&pollingQueue, "Charlie", 1003);

    // Print the voter queue
    printQueue(&pollingQueue);

    // Process voters (dequeue)
    printf("\nProcessing voters:\n");
    dequeue(&pollingQueue);  // Alice casts her vote
    dequeue(&pollingQueue);  // Bob casts his vote
    dequeue(&pollingQueue);  // Charlie casts his vote

    return 0;
}
```

18.
```
#include <stdio.h>
#include <string.h>

#define MAX_VOTERS 100
#define MAX_NAME_LEN 50

struct Voter {
```

```
    char name[MAX_NAME_LEN];
    int id;  // Voter ID
};

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

void initializeQueue(struct Queue *q) {
    q->front = 0;
    q->rear = -1;
}

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

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

void enqueue(struct Queue *q, char *name, int id) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    q->rear++;
    strcpy(q->voters[q->rear].name, name);
```

```
    q->voters[q->rear].id = id;
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No voters in the queue.\n");
        return;
    }

    struct Voter voter = q->voters[q->front];
    printf("Voter %d (%s) is casting their vote.\n", voter.id, voter.name);
    q->front++;
}

void printQueue(struct Queue *q) {
    printf("Voter queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("Voter ID: %d, Name: %s\n", q->voters[i].id, q->voters[i].name);
    }
}

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

    // Register voters (enqueue)
    enqueue(&pollingQueue, "Alice", 1001);
    enqueue(&pollingQueue, "Bob", 1002);
    enqueue(&pollingQueue, "Charlie", 1003);
```

```
    // Print the voter queue
    printQueue(&pollingQueue);

    // Process voters (dequeue)
    printf("\nProcessing voters:\n");
    dequeue(&pollingQueue);  // Alice casts her vote
    dequeue(&pollingQueue);  // Bob casts his vote
    dequeue(&pollingQueue);  // Charlie casts his vote

    return 0;
}
```

19.

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

typedef struct Patient {
    char name[50];
    int severity;
    struct Patient* next;
} Patient;

Patient* createPatient(char name[], int severity) {
    Patient* newPatient = (Patient*)malloc(sizeof(Patient));
    strcpy(newPatient->name, name);
    newPatient->severity = severity;
    newPatient->next = NULL;
    return newPatient;
}
```

```
void insertPatient(Patient** head, char name[], int severity) {
    Patient* newPatient = createPatient(name, severity);
    if (*head == NULL || (*head)->severity < severity) {
        newPatient->next = *head;
        *head = newPatient;
    } else {
        Patient* current = *head;
        while (current->next != NULL && current->next->severity >= severity) {
            current = current->next;
        }
        newPatient->next = current->next;
        current->next = newPatient;
    }
}

void servePatient(Patient** head) {
    if (*head != NULL) {
        Patient* temp = *head;
        *head = (*head)->next;
        printf("Serving patient: %s\n", temp->name);
        free(temp);
    } else {
        printf("No patients in the queue.\n");
    }
}

void printQueue(Patient* head) {
    if (head == NULL) {
```

```
        printf("No patients in the queue.\n");
    } else {
        Patient* current = head;
        while (current != NULL) {
            printf("Patient: %s, Severity: %d\n", current->name, current->severity);
            current = current->next;
        }
    }
}

int main() {
    Patient* queue = NULL;
    insertPatient(&queue, "John", 2);
    insertPatient(&queue, "Alice", 5);
    insertPatient(&queue, "Bob", 3);

    printQueue(queue);
    servePatient(&queue);
    servePatient(&queue);
    printQueue(queue);

    return 0;
}
```

20.
```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

typedef struct SurveyData {
```

```
    int surveyorID;
    char response[100];
} SurveyData;

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

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

int isFull(Queue* q) {
    return (q->rear + 1) % SIZE == q->front;
}

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

void enqueue(Queue* q, int surveyorID, char response[]) {
    if (isFull(q)) {
        printf("Queue is full. Unable to collect data.\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
```

```
    q->rear = (q->rear + 1) % SIZE;
    q->data[q->rear].surveyorID = surveyorID;
    snprintf(q->data[q->rear].response, sizeof(q->data[q->rear].response), "%s", response);
}

SurveyData dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No data to process.\n");
        SurveyData emptyData = {0, ""};
        return emptyData;
    }
    SurveyData temp = q->data[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % SIZE;
    }
    return temp;
}

void printQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    int i = q->front;
    while (i != q->rear) {
        printf("Surveyor %d: %s\n", q->data[i].surveyorID, q->data[i].response);
        i = (i + 1) % SIZE;
```

```
    }
    printf("Surveyor %d: %s\n", q->data[q->rear].surveyorID, q->data[q->rear].response);
}

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

    enqueue(&q, 1, "Yes");
    enqueue(&q, 2, "No");
    enqueue(&q, 3, "Maybe");

    printQueue(&q);

    SurveyData data = dequeue(&q);
    printf("Processed: Surveyor %d, Response: %s\n", data.surveyorID, data.response);

    printQueue(&q);

    return 0;
}
```

20.
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

```
} MarketData;

MarketData* createMarketData(char symbol[], double price) {
    MarketData* newData = (MarketData*)malloc(sizeof(MarketData));
    strcpy(newData->symbol, symbol);
    newData->price = price;
    newData->next = NULL;
    return newData;
}

void enqueue(MarketData** head, char symbol[], double price) {
    MarketData* newData = createMarketData(symbol, price);
    if (*head == NULL) {
        *head = newData;
    } else {
        MarketData* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newData;
    }
}

void analyzeData(MarketData* head) {
    if (head == NULL) {
        printf("No market data to analyze.\n");
        return;
    }
    MarketData* current = head;
```

```
    while (current != NULL) {
        printf("Analyzing Data: Symbol = %s, Price = %.2f\n", current->symbol, current->price);
        // Implement your market analysis logic here.
        current = current->next;
    }
}

int main() {
    MarketData* queue = NULL;

    enqueue(&queue, "AAPL", 145.30);
    enqueue(&queue, "GOOG", 2735.75);
    enqueue(&queue, "TSLA", 809.50);

    analyzeData(queue);

    return 0;
}
```