**Queue 21-01-2025**

* 1**. Stock Market Order Matching System\*\*: Implement a queue using arrays to simulate a stock market's order matching system. Design a program where buy and sell orders are placed in a queue. The system should match and process orders based on price and time priority.**
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* #define MAX 100
* typedef struct {
* char type[5];
* int quantity;
* float price;
* } Order;
* typedef struct {
* Order orders[MAX];
* int front, rear;
* } Queue;
* void initQueue(Queue \*q) {
* q->front = q->rear = -1;
* }
* int isEmpty(Queue \*q) {
* return q->front == -1;
* }
* int isFull(Queue \*q) {
* return q->rear == MAX - 1;
* }
* // Enqueue an order
* void enqueue(Queue \*q, char \*type, int quantity, float price) {
* if (isFull(q)) {
* printf("Queue is full! Cannot add more orders.\n");
* return;
* }
* if (isEmpty(q)) {
* q->front = 0;
* }
* q->rear++;
* strcpy(q->orders[q->rear].type, type);
* q->orders[q->rear].quantity = quantity;
* q->orders[q->rear].price = price;
* printf("%s order added: %d shares at $%.2f\n", type, quantity, price);
* }
* // Dequeue an order
* void dequeue(Queue \*q) {
* if (isEmpty(q)) {
* printf("No orders to process.\n");
* return;
* }
* printf("Processing %s order: %d shares at $%.2f\n",
* q->orders[q->front].type,
* q->orders[q->front].quantity,
* q->orders[q->front].price);
* if (q->front == q->rear) {
* q->front = q->rear = -1;
* } else {
* q->front++;
* }
* }
* // Display all orders in the queue
* void display(Queue \*q) {
* if (isEmpty(q)) {
* printf("No orders in the queue.\n");
* return;
* }
* printf("Current Orders:\n");
* for (int i = q->front; i <= q->rear; i++) {
* printf("%s: %d shares at $%.2f\n",
* q->orders[i].type,
* q->orders[i].quantity,
* q->orders[i].price);
* }
* }
* int main() {
* Queue buyQueue, sellQueue;
* initQueue(&buyQueue);
* initQueue(&sellQueue);
* int choice;
* char type[5];
* int quantity;
* float price;
* while (1) {
* printf("\nMenu:\n");
* printf("1. Add Buy Order\n");
* printf("2. Add Sell Order\n");
* printf("3. Process Buy Order\n");
* printf("4. Process Sell Order\n");
* printf("5. Display Orders\n");
* printf("6. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter quantity and price for BUY order: ");
* scanf("%d %f", &quantity, &price);
* enqueue(&buyQueue, "BUY", quantity, price);
* break;
* case 2:
* printf("Enter quantity and price for SELL order: ");
* scanf("%d %f", &quantity, &price);
* enqueue(&sellQueue, "SELL", quantity, price);
* break;
* case 3:
* printf("Processing a BUY order...\n");
* dequeue(&buyQueue);
* break;
* case 4:
* printf("Processing a SELL order...\n");
* dequeue(&sellQueue);
* break;
* case 5:
* printf("\nBuy Orders:\n");
* display(&buyQueue);
* printf("\nSell Orders:\n");
* display(&sellQueue);
* break;
* case 6:
* exit(0);
* default:
* printf("Invalid choice. Please try again.\n");
* }
* }
* return 0;
* }

* 2. \*\*Customer Service Center Simulation\*\*: Use a linked list to implement a queue for a customer service center. Each customer has a priority level based on their membership status, and the program should handle priority-based queueing and dynamic customer arrival.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* struct Customer {
* int id;
* char name[50];
* int priority; // Higher value = higher priority
* struct Customer \*next;
* };
* struct Customer \*front = NULL;
* // Function prototypes
* void enqueue(int id, const char \*name, int priority);
* void dequeue();
* void display();
* struct Customer \*createCustomer(int id, const char \*name, int priority);
* int main() {
* int choice, id, priority;
* char name[50];
* do {
* printf("\nCustomer Service Center Menu:\n");
* printf("1. Add Customer\n");
* printf("2. Serve Customer\n");
* printf("3. Display Queue\n");
* printf("4. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter Customer ID: ");
* scanf("%d", &id);
* printf("Enter Customer Name: ");
* scanf("%s", name);
* printf("Enter Priority (1=Low, 2=Medium, 3=High): ");
* scanf("%d", &priority);
* enqueue(id, name, priority);
* break;
* case 2:
* dequeue();
* break;
* case 3:
* display();
* break;
* case 4:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 4);
* return 0;
* }
* struct Customer \*createCustomer(int id, const char \*name, int priority) {
* struct Customer \*newCustomer = (struct Customer \*)malloc(sizeof(struct Customer));
* newCustomer->id = id;
* strcpy(newCustomer->name, name);
* newCustomer->priority = priority;
* newCustomer->next = NULL;
* return newCustomer;
* }
* void enqueue(int id, const char \*name, int priority) {
* struct Customer \*newCustomer = createCustomer(id, name, priority);
* if (front == NULL || front->priority < priority) {
* newCustomer->next = front;
* front = newCustomer;
* } else {
* struct Customer \*temp = front;
* while (temp->next != NULL && temp->next->priority >= priority) {
* temp = temp->next;
* }
* newCustomer->next = temp->next;
* temp->next = newCustomer;
* }
* printf("Customer added: ID=%d, Name=%s, Priority=%d\n", id, name, priority);
* }
* void dequeue() {
* if (front == NULL) {
* printf("Queue is empty. No customers to serve.\n");
* return;
* }
* struct Customer \*temp = front;
* printf("Serving customer: ID=%d, Name=%s, Priority=%d\n", temp->id, temp->name, temp->priority);
* front = front->next;
* free(temp);
* }
* void display() {
* if (front == NULL) {
* printf("Queue is empty.\n");
* return;
* }
* struct Customer \*temp = front;
* printf("Customer Queue:\n");
* 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.
* #include <stdio.h>
* #include <string.h>
* #define MAX 100
* typedef struct {
* int id;
* char name[50];
* int isVIP; // 1 for VIP, 0 for regular
* } Attendee;
* typedef struct {
* Attendee queue[MAX];
* int front, rear;
* } Queue;
* void initializeQueue(Queue \*q) {
* q->front = q->rear = -1;
* }
* int isFull(Queue \*q) {
* return q->rear == MAX - 1;
* }
* int isEmpty(Queue \*q) {
* return q->front == q->rear;
* }
* void enqueue(Queue \*q, int id, const char \*name, int isVIP) {
* if (isFull(q)) {
* printf("Queue is full!\n");
* return;
* }
* Attendee newAttendee = {id, "", isVIP};
* strcpy(newAttendee.name, name);
* q->rear++;
* if (isVIP) {
* for (int i = q->rear; i > q->front + 1; i--) {
* q->queue[i] = q->queue[i - 1];
* }
* q->queue[q->front + 1] = newAttendee;
* } else {
* q->queue[q->rear] = newAttendee;
* }
* printf("Registered: %s (ID=%d, VIP=%s)\n", name, id, isVIP ? "Yes" : "No");
* }
* void dequeue(Queue \*q) {
* if (isEmpty(q)) {
* printf("Queue is empty!\n");
* return;
* }
* q->front++;
* printf("Checked-in: %s (ID=%d)\n", q->queue[q->front].name, q->queue[q->front].id);
* }
* void display(Queue \*q) {
* if (isEmpty(q)) {
* printf("No attendees in the queue.\n");
* return;
* }
* printf("Attendees:\n");
* for (int i = q->front + 1; i <= q->rear; i++) {
* printf("%s (ID=%d, VIP=%s)\n", q->queue[i].name, q->queue[i].id, q->queue[i].isVIP ? "Yes" : "No");
* }
* }
* int main() {
* Queue q;
* initializeQueue(&q);
* int choice, id, isVIP;
* char name[50];
* do {
* printf("\n1. Register Attendee\n2. Check-in Attendee\n3. Display Attendees\n4. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter ID, Name, and VIP (1/0): ");
* scanf("%d %s %d", &id, name, &isVIP);
* enqueue(&q, id, name, isVIP);
* break;
* case 2:
* dequeue(&q);
* break;
* case 3:
* display(&q);
* break;
* case 4:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice!\n");
* }
* } while (choice != 4);
* return 0;
* }
* 4. \*\*Bank Teller Simulation\*\*: Develop a program using a linked list to simulate a queue at a bank. Customers arrive at random intervals, and each teller can handle one customer at a time. The program should simulate multiple tellers and different transaction times.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* #include <time.h>
* struct Customer {
* int id;
* int transactionTime;
* struct Customer \*next;
* };
* struct Queue {
* struct Customer \*front, \*rear;
* };
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = NULL;
* }
* int isEmpty(struct Queue \*q) {
* return q->front == NULL;
* }
* void enqueue(struct Queue \*q, int id, int transactionTime) {
* struct Customer \*newCustomer = (struct Customer \*)malloc(sizeof(struct Customer));
* newCustomer->id = id;
* newCustomer->transactionTime = transactionTime;
* newCustomer->next = NULL;
* if (isEmpty(q)) {
* q->front = q->rear = newCustomer;
* } else {
* q->rear->next = newCustomer;
* q->rear = newCustomer;
* }
* printf("Customer %d joined the queue (Transaction Time: %d minutes).\n", id, transactionTime);
* }
* void dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No customers in the queue.\n");
* return;
* }
* struct Customer \*temp = q->front;
* printf("Serving Customer %d (Transaction Time: %d minutes).\n", temp->id, temp->transactionTime);
* q->front = q->front->next;
* free(temp);
* if (q->front == NULL) {
* q->rear = NULL;
* }
* }
* void displayQueue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("Queue is empty.\n");
* return;
* }
* struct Customer \*temp = q->front;
* printf("Current Queue:\n");
* while (temp) {
* printf("Customer %d (Transaction Time: %d minutes)\n", temp->id, temp->transactionTime);
* temp = temp->next;
* }
* }
* int main() {
* struct Queue bankQueue;
* initializeQueue(&bankQueue);
* int choice, id = 1, transactionTime;
* srand(time(0)); // Seed for random transaction times
* do {
* printf("\n1. Add Customer\n2. Serve Customer\n3. Display Queue\n4. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* transactionTime = rand() % 10 + 1;
* enqueue(&bankQueue, id++, transactionTime);
* break;
* case 2:
* dequeue(&bankQueue);
* break;
* case 3:
* displayQueue(&bankQueue);
* break;
* case 4:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 4);
* return 0;
* }

* 5. \*\*Real-Time Data Feed Processing\*\*: Implement a queue using arrays to process real-time data feeds from multiple financial instruments. The system should handle high-frequency data inputs and ensure data integrity and order.
* #include <stdio.h>
* #include <stdlib.h>
* #define MAX 100
* struct DataFeed {
* int id;          // Instrument ID
* float price;     // Current price
* char timestamp[20]; // Timestamp
* };
* struct Queue {
* struct DataFeed data[MAX];
* int front, rear;
* };
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = -1;
* }
* int isFull(struct Queue \*q) {
* return q->rear == MAX - 1;
* }
* int isEmpty(struct Queue \*q) {
* return q->front == q->rear;
* }
* void enqueue(struct Queue \*q, int id, float price, char \*timestamp) {
* if (isFull(q)) {
* printf("Queue is full. Cannot add more data.\n");
* return;
* }
* q->rear++;
* q->data[q->rear].id = id;
* q->data[q->rear].price = price;
* snprintf(q->data[q->rear].timestamp, 20, "%s", timestamp);
* printf("Data added: Instrument ID: %d, Price: %.2f, Timestamp: %s\n", id, price, timestamp);
* }
* void dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("Queue is empty. No data to process.\n");
* return;
* }
* q->front++;
* printf("Processing Data: Instrument ID: %d, Price: %.2f, Timestamp: %s\n",
* q->data[q->front].id, q->data[q->front].price, q->data[q->front].timestamp);
* }
* void display(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("Queue is empty.\n");
* return;
* }
* printf("Current Queue:\n");
* for (int i = q->front + 1; i <= q->rear; i++) {
* printf("Instrument ID: %d, Price: %.2f, Timestamp: %s\n",
* q->data[i].id, q->data[i].price, q->data[i].timestamp);
* }
* }
* int main() {
* struct Queue dataQueue;
* initializeQueue(&dataQueue);
* int choice, id;
* float price;
* char timestamp[20];
* do {
* printf("\n1. Add Data\n2. Process Data\n3. Display Queue\n4. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* if (!isFull(&dataQueue)) {
* printf("Enter Instrument ID: ");
* scanf("%d", &id);
* printf("Enter Price: ");
* scanf("%f", &price);
* printf("Enter Timestamp: ");
* scanf("%s", timestamp);
* enqueue(&dataQueue, id, price, timestamp);
* } else {
* printf("Queue is full. Cannot add more data.\n");
* }
* break;
* case 2:
* dequeue(&dataQueue);
* break;
* case 3:
* display(&dataQueue);
* break;
* case 4:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 4);
* 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>
* struct Car {
* int carID;              // Car identifier
* struct Car \*next;       // Pointer to next car
* };
* struct Queue {
* struct Car \*front, \*rear;
* };
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = NULL;
* }
* int isEmpty(struct Queue \*q) {
* return q->front == NULL;
* }
* void enqueue(struct Queue \*q, int carID) {
* struct Car \*newCar = (struct Car \*)malloc(sizeof(struct Car));
* newCar->carID = carID;
* newCar->next = NULL;
* if (q->rear == NULL) {
* q->front = q->rear = newCar;
* } else {
* q->rear->next = newCar;
* q->rear = newCar;
* }
* printf("Car %d added to the queue.\n", carID);
* }
* int dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No cars in the queue.\n");
* return -1;
* }
* struct Car \*temp = q->front;
* int carID = temp->carID;
* q->front = q->front->next;
* if (q->front == NULL) {
* q->rear = NULL;
* }
* free(temp);
* return carID;
* }
* void displayQueue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No cars in the queue.\n");
* return;
* }
* printf("Cars in the queue: ");
* struct Car \*temp = q->front;
* while (temp != NULL) {
* printf("%d ", temp->carID);
* temp = temp->next;
* }
* printf("\n");
* }
* void changeTrafficLight(int \*lightState) {
* if (\*lightState == 1) {
* printf("Traffic Light: Green\n");
* \*lightState = 0;  // Change to red after green light
* } else {
* printf("Traffic Light: Red\n");
* \*lightState = 1;  // Change to green after red light
* }
* }
* int main() {
* struct Queue q;
* initializeQueue(&q);
* int choice, carID, lightState = 1;  // 1 for Red, 0 for Green
* do {
* printf("\n1. Add Car to Queue\n2. Change Traffic Light\n3. Process Car\n4. Display Queue\n5. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter Car ID: ");
* scanf("%d", &carID);
* enqueue(&q, carID);
* break;
* case 2:
* changeTrafficLight(&lightState);
* break;
* case 3:
* if (lightState == 0) {
* carID = dequeue(&q);
* if (carID != -1) {
* printf("Car %d has passed through the green light.\n", carID);
* }
* } else {
* printf("Traffic light is red. No cars can pass.\n");
* }
* break;
* case 4:
* displayQueue(&q);
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* return 0;
* }
* 7. \*\*Election Vote Counting System\*\*: Implement a queue using arrays to manage the vote counting process during an election. The system should handle multiple polling stations and ensure votes are counted in the order received.
* #include <stdio.h>
* #include <stdlib.h>
* #define MAX 5
* struct Queue {
* int votes[MAX];
* int front, rear;
* };
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = -1;
* }
* int isFull(struct Queue \*q) {
* return q->rear == MAX - 1;
* }
* int isEmpty(struct Queue \*q) {
* return q->front == -1;
* }
* void enqueue(struct Queue \*q, int vote) {
* if (isFull(q)) {
* printf("Queue is full, cannot accept more votes.\n");
* } else {
* if (q->front == -1) {
* q->front = 0;
* }
* q->rear++;
* q->votes[q->rear] = vote;
* printf("Vote for candidate %d added.\n", vote);
* }
* }
* int dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No votes to count.\n");
* return -1;
* } else {
* int vote = q->votes[q->front];
* q->front++;
* if (q->front > q->rear) {
* q->front = q->rear = -1;  // Reset queue when all votes are counted
* }
* return vote;
* }
* }
* void displayQueue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No votes in the queue.\n");
* return;
* }
* printf("Votes in the queue: ");
* for (int i = q->front; i <= q->rear; i++) {
* printf("%d ", q->votes[i]);
* }
* printf("\n");
* }
* int main() {
* struct Queue q;
* initializeQueue(&q);
* int choice, vote;
* do {
* printf("\n1. Add Vote\n2. Count Vote\n3. Display Votes\n4. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter vote (Candidate ID): ");
* scanf("%d", &vote);
* enqueue(&q, vote);
* break;
* case 2:
* vote = dequeue(&q);
* if (vote != -1) {
* printf("Vote counted for candidate %d\n", vote);
* }
* break;
* case 3:
* displayQueue(&q);
* break;
* case 4:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 4);
* return 0;
* }

* 8. \*\*Airport Runway Management\*\*: Use a linked list to implement a queue for airplanes waiting to land or take off. The system should handle priority for emergency landings and manage runway allocation efficiently.
* #include <stdio.h>
* #include <stdlib.h>
* struct Plane {
* int id;
* char type; // 'L' for landing, 'T' for take-off, 'E' for emergency landing
* struct Plane \*next;
* };
* struct Queue {
* struct Plane \*front, \*rear;
* };
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = NULL;
* }
* int isEmpty(struct Queue \*q) {
* return q->front == NULL;
* }
* void enqueue(struct Queue \*q, int id, char type) {
* struct Plane \*newPlane = (struct Plane \*)malloc(sizeof(struct Plane));
* newPlane->id = id;
* newPlane->type = type;
* newPlane->next = NULL;
* if (isEmpty(q)) {
* q->front = q->rear = newPlane;
* } else {
* q->rear->next = newPlane;
* q->rear = newPlane;
* }
* printf("Plane %d (%c) added to queue.\n", id, type);
* }
* void emergencyLanding(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No planes in the queue.\n");
* return;
* }
* struct Plane \*temp = q->front;
* struct Plane \*prev = NULL;
* // Search for the first plane with an emergency landing
* while (temp != NULL && temp->type != 'E') {
* prev = temp;
* temp = temp->next;
* }
* if (temp != NULL) {
* if (prev == NULL) {
* // Emergency plane is at the front
* q->front = q->front->next;
* } else {
* prev->next = temp->next;
* }
* printf("Emergency plane %d (%c) is being processed.\n", temp->id, temp->type);
* free(temp);
* } else {
* printf("No emergency landing planes found in queue.\n");
* }
* }
* void dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No planes in the queue.\n");
* return;
* }
* struct Plane \*temp = q->front;
* q->front = q->front->next;
* printf("Plane %d (%c) processed.\n", temp->id, temp->type);
* free(temp);
* }
* void displayQueue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No planes in the queue.\n");
* return;
* }
* struct Plane \*temp = q->front;
* printf("Planes in the queue: ");
* while (temp != NULL) {
* printf("%d(%c) ", temp->id, temp->type);
* temp = temp->next;
* }
* printf("\n");
* }
* int main() {
* struct Queue q;
* initializeQueue(&q);
* int choice, id;
* char type;
* do {
* printf("\n1. Add Plane to Queue\n2. Emergency Landing\n3. Process Plane\n4. Display Queue\n5. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter plane ID: ");
* scanf("%d", &id);
* printf("Enter plane type (L for landing, T for take-off, E for emergency landing): ");
* scanf(" %c", &type); // Space before %c to clear newline
* enqueue(&q, id, type);
* break;
* case 2:
* emergencyLanding(&q);
* break;
* case 3:
* dequeue(&q);
* break;
* case 4:
* displayQueue(&q);
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* 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\_SIZE 10
* struct Order {
* int id;
* char type; // 'B' for buy, 'S' for sell
* int quantity;
* float price;
* };
* struct OrderQueue {
* struct Order orders[MAX\_SIZE];
* int front, rear;
* };
* void initializeQueue(struct OrderQueue \*q) {
* q->front = -1;
* q->rear = -1;
* }
* int isFull(struct OrderQueue \*q) {
* return q->rear == MAX\_SIZE - 1;
* }
* int isEmpty(struct OrderQueue \*q) {
* return q->front == -1 || q->front > q->rear;
* }
* void enqueue(struct OrderQueue \*q, int id, char type, int quantity, float price) {
* if (isFull(q)) {
* printf("Order queue is full! Cannot add more orders.\n");
* return;
* }
* struct Order newOrder = {id, type, quantity, price};
* if (isEmpty(q)) {
* q->front = 0;
* }
* q->rear++;
* q->orders[q->rear] = newOrder;
* printf("Order ID %d (%c) with %d shares at %.2f added.\n", id, type, quantity, price);
* }
* void dequeue(struct OrderQueue \*q) {
* if (isEmpty(q)) {
* printf("No orders to process.\n");
* return;
* }
* struct Order processedOrder = q->orders[q->front];
* printf("Order ID %d (%c) with %d shares at %.2f processed.\n", processedOrder.id, processedOrder.type, processedOrder.quantity, processedOrder.price);
* q->front++;
* }
* void cancelOrder(struct OrderQueue \*q, int id) {
* if (isEmpty(q)) {
* printf("No orders to cancel.\n");
* return;
* }
* int found = 0;
* for (int i = q->front; i <= q->rear; i++) {
* if (q->orders[i].id == id) {
* found = 1;
* printf("Order ID %d canceled.\n", id);
* for (int j = i; j < q->rear; j++) {
* q->orders[j] = q->orders[j + 1];
* }
* q->rear--;
* if (q->rear < q->front) {
* q->front = -1;
* }
* break;
* }
* }
* if (!found) {
* printf("Order ID %d not found.\n", id);
* }
* }
* void displayQueue(struct OrderQueue \*q) {
* if (isEmpty(q)) {
* printf("No orders in the queue.\n");
* return;
* }
* printf("Orders in the queue:\n");
* for (int i = q->front; i <= q->rear; i++) {
* struct Order order = q->orders[i];
* printf("Order ID %d (%c) - %d shares at %.2f\n", order.id, order.type, order.quantity, order.price);
* }
* }
* int main() {
* struct OrderQueue queue;
* initializeQueue(&queue);
* int choice, id, quantity;
* float price;
* char type;
* do {
* printf("\n1. Add Order\n2. Process Order\n3. Cancel Order\n4. Display Orders\n5. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter order ID: ");
* scanf("%d", &id);
* printf("Enter order type (B for buy, S for sell): ");
* scanf(" %c", &type);
* printf("Enter quantity: ");
* scanf("%d", &quantity);
* printf("Enter price: ");
* scanf("%f", &price);
* enqueue(&queue, id, type, quantity, price);
* break;
* case 2:
* dequeue(&queue);
* break;
* case 3:
* printf("Enter order ID to cancel: ");
* scanf("%d", &id);
* cancelOrder(&queue, id);
* break;
* case 4:
* displayQueue(&queue);
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* 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>
* struct Participant {
* int id;
* char name[50];
* char registrationType[20]; // "Pre-registration" or "Walk-in"
* struct Participant \*next;
* };
* struct Queue {
* struct Participant \*front, \*rear;
* };
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = NULL;
* }
* int isEmpty(struct Queue \*q) {
* return q->front == NULL;
* }
* void enqueue(struct Queue \*q, int id, const char \*name, const char \*registrationType) {
* struct Participant \*newParticipant = (struct Participant\*)malloc(sizeof(struct Participant));
* if (!newParticipant) {
* printf("Memory allocation failed.\n");
* return;
* }
* newParticipant->id = id;
* strcpy(newParticipant->name, name);
* strcpy(newParticipant->registrationType, registrationType);
* newParticipant->next = NULL;
* if (isEmpty(q)) {
* q->front = q->rear = newParticipant;
* } else {
* q->rear->next = newParticipant;
* q->rear = newParticipant;
* }
* printf("Participant %s (ID: %d) added successfully.\n", name, id);
* }
* void dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No participants to process.\n");
* return;
* }
* struct Participant \*temp = q->front;
* printf("Participant %s (ID: %d) processed.\n", temp->name, temp->id);
* q->front = q->front->next;
* free(temp);
* if (q->front == NULL) {
* q->rear = NULL;
* }
* }
* void cancelRegistration(struct Queue \*q, int id) {
* if (isEmpty(q)) {
* printf("No participants to cancel.\n");
* return;
* }
* struct Participant \*temp = q->front, \*prev = NULL;
* while (temp != NULL) {
* if (temp->id == id) {
* if (prev == NULL) {
* q->front = temp->next;
* } else {
* prev->next = temp->next;
* }
* if (temp == q->rear) {
* q->rear = prev;
* }
* printf("Participant %s (ID: %d) canceled successfully.\n", temp->name, id);
* free(temp);
* return;
* }
* prev = temp;
* temp = temp->next;
* }
* printf("Participant with ID %d not found.\n", id);
* }
* void displayQueue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No participants registered.\n");
* return;
* }
* struct Participant \*temp = q->front;
* printf("Current registered participants:\n");
* while (temp != NULL) {
* printf("ID: %d, Name: %s, Type: %s\n", temp->id, temp->name, temp->registrationType);
* temp = temp->next;
* }
* }
* int main() {
* struct Queue q;
* initializeQueue(&q);
* int choice, id;
* char name[50], type[20];
* do {
* printf("\n1. Register Participant\n2. Process Registration\n3. Cancel Registration\n4. Display Registrations\n5. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter participant ID: ");
* scanf("%d", &id);
* printf("Enter participant name: ");
* scanf(" %[^\n]", name);  // To read name with spaces
* printf("Enter registration type (Pre-registration / Walk-in): ");
* scanf(" %[^\n]", type);
* enqueue(&q, id, name, type);
* break;
* case 2:
* dequeue(&q);
* break;
* case 3:
* printf("Enter participant ID to cancel: ");
* scanf("%d", &id);
* cancelRegistration(&q, id);
* break;
* case 4:
* displayQueue(&q);
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* return 0;
* }
* 11. \*\*Political Debate Audience Management\*\*: Use arrays to implement a queue for managing the audience at a political debate. The system should handle entry, seating arrangements, and priority access for media personnel.
* #include <stdio.h>
* #include <string.h>
* #define MAX\_SIZE 100
* // Structure for each audience member
* struct Audience {
* int id;
* char name[50];
* char role[20]; // "Media" or "General"
* };
* // Queue structure using an array
* struct Queue {
* struct Audience arr[MAX\_SIZE];
* int front, rear;
* };
* // Function to initialize the queue
* void initializeQueue(struct Queue \*q) {
* q->front = q->rear = -1;
* }
* // Function to check if the queue is empty
* int isEmpty(struct Queue \*q) {
* return q->front == -1;
* }
* // Function to check if the queue is full
* int isFull(struct Queue \*q) {
* return q->rear == MAX\_SIZE - 1;
* }
* // Function to enqueue (add) a participant
* void enqueue(struct Queue \*q, int id, const char \*name, const char \*role) {
* if (isFull(q)) {
* printf("Queue is full. Cannot add more participants.\n");
* return;
* }
* struct Audience newAudience;
* newAudience.id = id;
* strcpy(newAudience.name, name);
* strcpy(newAudience.role, role);
* if (q->front == -1) {
* q->front = 0;
* }
* q->arr[++q->rear] = newAudience;
* printf("Participant %s (ID: %d) added to the queue.\n", name, id);
* }
* // Function to dequeue (remove) a participant from the front of the queue
* void dequeue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("Queue is empty. No participant to remove.\n");
* return;
* }
* struct Audience removed = q->arr[q->front];
* printf("Participant %s (ID: %d) has entered the debate hall.\n", removed.name, removed.id);
* if (q->front == q->rear) {
* q->front = q->rear = -1;
* } else {
* q->front++;
* }
* }
* // Function to display the current queue of participants
* void displayQueue(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("No participants in the queue.\n");
* return;
* }
* printf("Current queue of participants:\n");
* for (int i = q->front; i <= q->rear; i++) {
* printf("ID: %d, Name: %s, Role: %s\n", q->arr[i].id, q->arr[i].name, q->arr[i].role);
* }
* }
* // Function to handle priority access for media personnel
* void priorityMedia(struct Queue \*q) {
* if (isEmpty(q)) {
* printf("Queue is empty.\n");
* return;
* }
* printf("Processing media personnel with priority...\n");
* // Process media participants first
* for (int i = q->front; i <= q->rear; i++) {
* if (strcmp(q->arr[i].role, "Media") == 0) {
* printf("Priority Access: %s (ID: %d) enters the debate hall.\n", q->arr[i].name, q->arr[i].id);
* // Remove this participant from the queue
* for (int j = i; j < q->rear; j++) {
* q->arr[j] = q->arr[j + 1];
* }
* q->rear--;
* i--; // To check the same position again after shifting
* }
* }
* }
* int main() {
* struct Queue q;
* initializeQueue(&q);
* int choice, id;
* char name[50], role[20];
* do {
* printf("\n1. Add Participant to Queue\n2. Enter Debate Hall\n3. Display Queue\n4. Process Media Priority\n5. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter participant ID: ");
* scanf("%d", &id);
* printf("Enter participant name: ");
* scanf(" %[^\n]", name);  // Read name with spaces
* printf("Enter participant role (Media / General): ");
* scanf(" %[^\n]", role);
* enqueue(&q, id, name, role);
* break;
* case 2:
* dequeue(&q);
* break;
* case 3:
* displayQueue(&q);
* break;
* case 4:
* priorityMedia(&q);
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* return 0;
* }

* 12. \*\*Bank Loan Application Processing\*\*: Develop a queue using linked lists to manage loan applications at a bank. The system should prioritize applications based on the loan amount and applicant's credit score.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* // Define structure for Loan Application
* struct LoanApplication {
* int id;
* float loanAmount;
* int creditScore;
* char applicantName[50];
* struct LoanApplication \*next;
* };
* // Define Queue structure
* struct LoanQueue {
* struct LoanApplication \*front, \*rear;
* };
* // Function to initialize the queue
* void initializeQueue(struct LoanQueue \*q) {
* q->front = q->rear = NULL;
* }
* // Function to check if the queue is empty
* int isEmpty(struct LoanQueue \*q) {
* return q->front == NULL;
* }
* // Function to create a new loan application
* struct LoanApplication\* createLoanApplication(int id, float loanAmount, int creditScore, const char \*applicantName) {
* struct LoanApplication \*newApp = (struct LoanApplication\*) malloc(sizeof(struct LoanApplication));
* newApp->id = id;
* newApp->loanAmount = loanAmount;
* newApp->creditScore = creditScore;
* strcpy(newApp->applicantName, applicantName);
* newApp->next = NULL;
* return newApp;
* }
* // Function to enqueue a new loan application, prioritize by loanAmount and creditScore
* void enqueue(struct LoanQueue \*q, int id, float loanAmount, int creditScore, const char \*applicantName) {
* struct LoanApplication \*newApp = createLoanApplication(id, loanAmount, creditScore, applicantName);
* // If the queue is empty, add the new application at the front
* if (isEmpty(q)) {
* q->front = q->rear = newApp;
* } else {
* // Insert the new application in the correct position based on priority
* struct LoanApplication \*current = q->front, \*previous = NULL;
* while (current != NULL && (current->loanAmount > loanAmount || (current->loanAmount == loanAmount && current->creditScore > creditScore))) {
* previous = current;
* current = current->next;
* }
* if (previous == NULL) {
* // Insert at the front
* newApp->next = q->front;
* q->front = newApp;
* } else {
* // Insert in between or at the end
* previous->next = newApp;
* newApp->next = current;
* if (current == NULL) {
* q->rear = newApp;
* }
* }
* }
* printf("Loan application for %s (ID: %d) has been added.\n", applicantName, id);
* }
* // Function to dequeue (process) the loan application with the highest priority
* void dequeue(struct LoanQueue \*q) {
* if (isEmpty(q)) {
* printf("No loan applications in the queue.\n");
* return;
* }
* struct LoanApplication \*temp = q->front;
* q->front = q->front->next;
* printf("Loan application for %s (ID: %d) processed.\n", temp->applicantName, temp->id);
* free(temp);
* }
* // Function to display all loan applications in the queue
* void displayQueue(struct LoanQueue \*q) {
* if (isEmpty(q)) {
* printf("No loan applications in the queue.\n");
* return;
* }
* struct LoanApplication \*current = q->front;
* printf("Loan applications in the queue (highest priority first):\n");
* while (current != NULL) {
* printf("ID: %d, Applicant: %s, Loan Amount: %.2f, Credit Score: %d\n", current->id, current->applicantName, current->loanAmount, current->creditScore);
* current = current->next;
* }
* }
* int main() {
* struct LoanQueue queue;
* initializeQueue(&queue);
* int choice, id, creditScore;
* float loanAmount;
* char name[50];
* do {
* printf("\n1. Add Loan Application\n2. Process Loan Application\n3. Display Queue\n4. Exit\nEnter your choice: ");
* scanf("%d", &choice);
* switch (choice) {
* case 1:
* printf("Enter loan application ID: ");
* scanf("%d", &id);
* printf("Enter applicant name: ");
* scanf(" %[^\n]", name);  // Read name with spaces
* printf("Enter loan amount: ");
* scanf("%f", &loanAmount);
* printf("Enter credit score: ");
* scanf("%d", &creditScore);
* enqueue(&queue, id, loanAmount, creditScore, name);
* break;
* case 2:
* dequeue(&queue);
* break;
* case 3:
* displayQueue(&queue);
* break;
* case 4:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Please try again.\n");
* }
* } while (choice != 4);
* return 0;
* }

* 13. \*\*Online Shopping Checkout System\*\*: Implement a queue using arrays for an online shopping platform's checkout system. The program should handle multiple customers checking out simultaneously and manage inventory updates.
* #include <stdio.h>
* #define MAX 5  // Maximum number of customers in the queue
* // Structure for Inventory Item
* struct Item {
* int id;
* char name[50];
* int quantity;
* float price;
* };
* // Structure for Customer Checkout
* struct Customer {
* int customerId;
* int itemId;
* int quantityBought;
* };
* // Declare inventory and queue
* struct Item inventory[3] = {{1, "Laptop", 10, 50000}, {2, "Smartphone", 20, 20000}, {3, "Headphones", 50, 3000}};
* struct Customer queue[MAX];
* int front = -1, rear = -1;
* void enqueue(int customerId, int itemId, int quantityBought);
* void dequeue();
* void displayQueue();
* void updateInventory(int itemId, int quantityBought);
* void displayInventory();
* int main() {
* int choice, customerId, itemId, quantityBought;
* do {
* printf("\n1. Add Customer to Checkout Queue\n");
* printf("2. Process Checkout\n");
* printf("3. Display Queue\n");
* printf("4. Display Inventory\n");
* printf("5. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* if (rear == MAX - 1) {
* printf("Queue is full!\n");
* } else {
* printf("Enter customer ID: ");
* scanf("%d", &customerId);
* printf("Enter item ID to checkout (1: Laptop, 2: Smartphone, 3: Headphones): ");
* scanf("%d", &itemId);
* printf("Enter quantity to buy: ");
* scanf("%d", &quantityBought);
* enqueue(customerId, itemId, quantityBought);
* }
* break;
* case 2:
* if (front == -1) {
* printf("No customers in the queue!\n");
* } else {
* dequeue();
* }
* break;
* case 3:
* displayQueue();
* break;
* case 4:
* displayInventory();
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* return 0;
* }
* // Function to add customer to checkout queue
* void enqueue(int customerId, int itemId, int quantityBought) {
* if (rear == -1) {
* front = rear = 0;
* } else {
* rear++;
* }
* queue[rear].customerId = customerId;
* queue[rear].itemId = itemId;
* queue[rear].quantityBought = quantityBought;
* // Update the inventory
* updateInventory(itemId, quantityBought);
* printf("Customer %d added to checkout queue.\n", customerId);
* }
* // Function to remove customer from checkout queue and process their order
* void dequeue() {
* int customerId = queue[front].customerId;
* int itemId = queue[front].itemId;
* int quantityBought = queue[front].quantityBought;
* printf("\nProcessing customer %d's order:\n", customerId);
* printf("Item ID: %d, Quantity: %d\n", itemId, quantityBought);
* // Update inventory after processing checkout
* updateInventory(itemId, -quantityBought);
* // Remove customer from the queue
* if (front == rear) {
* front = rear = -1;
* } else {
* front++;
* }
* }
* // Function to display the queue
* void displayQueue() {
* if (front == -1) {
* printf("Queue is empty!\n");
* } else {
* printf("\nCustomers in checkout queue:\n");
* for (int i = front; i <= rear; i++) {
* printf("Customer ID: %d, Item ID: %d, Quantity: %d\n", queue[i].customerId, queue[i].itemId, queue[i].quantityBought);
* }
* }
* }
* // Function to update the inventory after checkout
* void updateInventory(int itemId, int quantityBought) {
* for (int i = 0; i < 3; i++) {
* if (inventory[i].id == itemId) {
* inventory[i].quantity -= quantityBought;
* if (inventory[i].quantity < 0) {
* inventory[i].quantity = 0;
* }
* break;
* }
* }
* }
* // Function to display current inventory status
* void displayInventory() {
* printf("\nInventory status:\n");
* for (int i = 0; i < 3; i++) {
* printf("Item ID: %d, Name: %s, Quantity: %d, Price: %.2f\n", inventory[i].id, inventory[i].name, inventory[i].quantity, inventory[i].price);
* }
* }

* 14. \*\*Public Transport Scheduling\*\*: Use linked lists to implement a queue for managing bus arrivals and departures at a terminal. The system should handle peak hours, off-peak hours, and prioritize express buses.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* #define MAX\_PEAK\_HOURS 18
* #define MIN\_PEAK\_HOURS 9
* // Bus structure
* struct Bus {
* int busId;
* char type[20];    // "express" or "regular"
* int arrivalTime;  // 24-hour format (0-23)
* struct Bus\* next;
* };
* // Queue operations
* struct Bus\* front = NULL, \* rear = NULL;
* void enqueue(int busId, char\* type, int arrivalTime);
* void dequeue();
* void displayQueue();
* void processBusArrival(int currentTime);
* int isPeakHours(int currentTime);
* int main() {
* int choice, busId, arrivalTime, currentTime;
* char type[20];
* do {
* printf("\n1. Add Bus to Queue\n");
* printf("2. Process Bus Departure\n");
* printf("3. Display Bus Queue\n");
* printf("4. Process Bus Arrivals\n");
* printf("5. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* printf("Enter bus ID: ");
* scanf("%d", &busId);
* printf("Enter bus type (express/regular): ");
* scanf("%s", type);
* printf("Enter arrival time (24-hour format): ");
* scanf("%d", &arrivalTime);
* enqueue(busId, type, arrivalTime);
* break;
* case 2:
* dequeue();
* break;
* case 3:
* displayQueue();
* break;
* case 4:
* printf("Enter current time (24-hour format): ");
* scanf("%d", &currentTime);
* processBusArrival(currentTime);
* break;
* case 5:
* printf("Exiting...\n");
* break;
* default:
* printf("Invalid choice! Try again.\n");
* }
* } while (choice != 5);
* return 0;
* }
* // Function to add bus to queue
* void enqueue(int busId, char\* type, int arrivalTime) {
* struct Bus\* newBus = (struct Bus\*)malloc(sizeof(struct Bus));
* newBus->busId = busId;
* strcpy(newBus->type, type);
* newBus->arrivalTime = arrivalTime;
* newBus->next = NULL;
* if (rear == NULL) {
* front = rear = newBus;
* } else {
* rear->next = newBus;
* rear = newBus;
* }
* printf("Bus %d added to queue.\n", busId);
* }
* // Function to remove bus from queue
* void dequeue() {
* if (front == NULL) {
* printf("Queue is empty, no buses to process.\n");
* return;
* }
* struct Bus\* temp = front;
* printf("Processing bus %d (%s) at arrival time %d\n", front->busId, front->type, front->arrivalTime);
* front = front->next;
* free(temp);
* }
* // Function to display the queue
* void displayQueue() {
* if (front == NULL) {
* printf("Queue is empty.\n");
* return;
* }
* struct Bus\* temp = front;
* printf("Current bus queue:\n");
* while (temp != NULL) {
* printf("Bus ID: %d, Type: %s, Arrival Time: %d\n", temp->busId, temp->type, temp->arrivalTime);
* temp = temp->next;
* }
* }
* // Function to process bus arrivals
* void processBusArrival(int currentTime) {
* if (front == NULL) {
* printf("No buses to process.\n");
* return;
* }
* if (isPeakHours(currentTime)) {
* printf("Peak hours! Prioritizing express buses.\n");
* struct Bus\* temp = front;
* struct Bus\* expressBus = NULL;
* struct Bus\* regularBus = NULL;
* while (temp != NULL) {
* if (strcmp(temp->type, "express") == 0) {
* if (expressBus == NULL) expressBus = temp;
* } else {
* if (regularBus == NULL) regularBus = temp;
* }
* temp = temp->next;
* }
* // Process express bus first, if exists
* if (expressBus != NULL) {
* printf("Processing express bus %d at arrival time %d\n", expressBus->busId, expressBus->arrivalTime);
* } else {
* printf("No express buses in the queue.\n");
* }
* // Process regular bus if no express bus
* if (regularBus != NULL) {
* printf("Processing regular bus %d at arrival time %d\n", regularBus->busId, regularBus->arrivalTime);
* }
* } else {
* printf("Off-peak hours. Processing buses in the order of arrival.\n");
* dequeue();
* }
* }
* // Function to check if it's peak hours (9 AM to 6 PM)
* int isPeakHours(int currentTime) {
* return currentTime >= MIN\_PEAK\_HOURS && currentTime <= MAX\_PEAK\_HOURS;
* }

* 15. \*\*Political Rally Crowd Control\*\*: Develop a queue using arrays to manage the crowd at a political rally. The system should handle entry, exit, and VIP sections, ensuring safety and order.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* #define MAX\_QUEUE\_SIZE 10
* // Structure to hold attendee information
* struct Attendee {
* int id;
* char name[50];
* char type[10]; // "VIP" or "Regular"
* };
* // Queue to manage attendees
* struct Queue {
* struct Attendee queue[MAX\_QUEUE\_SIZE];
* int front;
* int rear;
* };
* // Function prototypes
* void initializeQueue(struct Queue\* q);
* int isQueueFull(struct Queue\* q);
* int isQueueEmpty(struct Queue\* q);
* void enqueue(struct Queue\* q, int id, const char\* name, const char\* type);
* void dequeue(struct Queue\* q);
* void displayQueue(struct Queue\* q);
* int main() {
* struct Queue rallyQueue;
* initializeQueue(&rallyQueue);
* int choice, id;
* char name[50], type[10];
* do {
* printf("\n1. Enter Rally (VIP or Regular)\n");
* printf("2. Exit Rally\n");
* printf("3. Display Queue\n");
* printf("4. Exit Program\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* if (isQueueFull(&rallyQueue)) {
* printf("Queue is full. Cannot add more attendees.\n");
* } else {
* printf("Enter attendee ID: ");
* scanf("%d", &id);
* printf("Enter name: ");
* scanf(" %[^\n]", name); // To allow spaces in name
* printf("Enter type (VIP/Regular): ");
* scanf("%s", type);
* enqueue(&rallyQueue, id, name, type);
* }
* break;
* case 2:
* if (isQueueEmpty(&rallyQueue)) {
* printf("No one is in the queue to exit.\n");
* } else {
* dequeue(&rallyQueue);
* }
* break;
* case 3:
* displayQueue(&rallyQueue);
* break;
* case 4:
* printf("Exiting program...\n");
* break;
* default:
* printf("Invalid choice. Try again.\n");
* }
* } while(choice != 4);
* return 0;
* }
* // Function to initialize the queue
* void initializeQueue(struct Queue\* q) {
* q->front = -1;
* q->rear = -1;
* }
* // Function to check if the queue is full
* int isQueueFull(struct Queue\* q) {
* return q->rear == MAX\_QUEUE\_SIZE - 1;
* }
* // Function to check if the queue is empty
* int isQueueEmpty(struct Queue\* q) {
* return q->front == -1;
* }
* // Function to add a new attendee to the queue
* void enqueue(struct Queue\* q, int id, const char\* name, const char\* type) {
* if (q->front == -1) {
* q->front = 0;
* }
* q->rear++;
* q->queue[q->rear].id = id;
* strcpy(q->queue[q->rear].name, name);
* strcpy(q->queue[q->rear].type, type);
* printf("Attendee %s added to the rally queue.\n", name);
* }
* // Function to remove an attendee from the queue
* void dequeue(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("Queue is empty. No one to exit.\n");
* } else {
* printf("Attendee %s is exiting the rally.\n", q->queue[q->front].name);
* for (int i = 0; i < q->rear; i++) {
* q->queue[i] = q->queue[i + 1]; // Shift the queue
* }
* q->rear--;
* if (q->rear == -1) {
* q->front = -1;
* }
* }
* }
* // Function to display the queue of attendees
* void displayQueue(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("The rally queue is empty.\n");
* return;
* }
* printf("Rally Queue:\n");
* for (int i = q->front; i <= q->rear; i++) {
* printf("ID: %d, Name: %s, Type: %s\n", q->queue[i].id, q->queue[i].name, q->queue[i].type);
* }
* }

* 16. \*\*Financial Transaction Processing\*\*: Implement a queue using linked lists to process financial transactions. The system should handle deposits, withdrawals, and transfers, ensuring real-time processing and accuracy.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* struct Transaction {
* int id;
* char type[10]; // "Deposit", "Withdrawal", or "Transfer"
* float amount;
* int account\_id;
* struct Transaction\* next;
* };
* struct Queue {
* struct Transaction\* front;
* struct Transaction\* rear;
* };
* // Function prototypes
* void initializeQueue(struct Queue\* q);
* int isQueueEmpty(struct Queue\* q);
* void enqueue(struct Queue\* q, int id, const char\* type, float amount, int account\_id);
* void dequeue(struct Queue\* q);
* void displayQueue(struct Queue\* q);
* void processTransaction(struct Transaction\* t);
* int main() {
* struct Queue transactionQueue;
* initializeQueue(&transactionQueue);
* int choice, id, account\_id;
* float amount;
* char type[10];
* do {
* printf("\n1. Add Transaction (Deposit, Withdrawal, Transfer)\n");
* printf("2. Process Transaction\n");
* printf("3. Display Queue\n");
* printf("4. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* printf("Enter Transaction ID: ");
* scanf("%d", &id);
* printf("Enter Transaction Type (Deposit/Withdrawal/Transfer): ");
* scanf("%s", type);
* printf("Enter Amount: ");
* scanf("%f", &amount);
* printf("Enter Account ID: ");
* scanf("%d", &account\_id);
* enqueue(&transactionQueue, id, type, amount, account\_id);
* break;
* case 2:
* if (isQueueEmpty(&transactionQueue)) {
* printf("No transactions to process.\n");
* } else {
* dequeue(&transactionQueue);
* }
* break;
* case 3:
* displayQueue(&transactionQueue);
* break;
* case 4:
* printf("Exiting program...\n");
* break;
* default:
* printf("Invalid choice. Try again.\n");
* }
* } while(choice != 4);
* return 0;
* }
* // Initialize the queue
* void initializeQueue(struct Queue\* q) {
* q->front = NULL;
* q->rear = NULL;
* }
* // Check if the queue is empty
* int isQueueEmpty(struct Queue\* q) {
* return (q->front == NULL);
* }
* // Enqueue a new transaction into the queue
* void enqueue(struct Queue\* q, int id, const char\* type, float amount, int account\_id) {
* struct Transaction\* t = (struct Transaction\*)malloc(sizeof(struct Transaction));
* if (!t) {
* printf("Memory allocation failed!\n");
* return;
* }
* t->id = id;
* strcpy(t->type, type);
* t->amount = amount;
* t->account\_id = account\_id;
* t->next = NULL;
* if (isQueueEmpty(q)) {
* q->front = q->rear = t;
* } else {
* q->rear->next = t;
* q->rear = t;
* }
* printf("Transaction added to queue.\n");
* }
* // Dequeue and process the next transaction in the queue
* void dequeue(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("Queue is empty, no transaction to process.\n");
* return;
* }
* struct Transaction\* t = q->front;
* q->front = q->front->next;
* if (q->front == NULL) {
* q->rear = NULL;
* }
* printf("Processing Transaction ID: %d\n", t->id);
* processTransaction(t);
* free(t);
* }
* // Display all transactions in the queue
* void displayQueue(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("Queue is empty.\n");
* return;
* }
* struct Transaction\* temp = q->front;
* printf("Transaction Queue:\n");
* while (temp) {
* printf("ID: %d, Type: %s, Amount: %.2f, Account ID: %d\n",
* temp->id, temp->type, temp->amount, temp->account\_id);
* temp = temp->next;
* }
* }
* // Process a transaction based on its type
* void processTransaction(struct Transaction\* t) {
* if (strcmp(t->type, "Deposit") == 0) {
* printf("Depositing %.2f to Account %d\n", t->amount, t->account\_id);
* } else if (strcmp(t->type, "Withdrawal") == 0) {
* printf("Withdrawing %.2f from Account %d\n", t->amount, t->account\_id);
* } else if (strcmp(t->type, "Transfer") == 0) {
* printf("Transferring %.2f from Account %d\n", t->amount, t->account\_id);
* } else {
* printf("Invalid transaction type.\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 MAX\_VOTERS 100
* // Structure to represent a voter
* struct Voter {
* int id;
* char name[50];
* int age;
* int verified; // 1 if verified, 0 if not verified
* };
* // Queue structure for managing voters
* struct Queue {
* struct Voter voters[MAX\_VOTERS];
* int front;
* int rear;
* };
* // Function prototypes
* void initializeQueue(struct Queue\* q);
* int isQueueEmpty(struct Queue\* q);
* int isQueueFull(struct Queue\* q);
* void enqueue(struct Queue\* q, int id, const char\* name, int age);
* void dequeue(struct Queue\* q);
* void verifyVoter(struct Queue\* q);
* void displayQueue(struct Queue\* q);
* int main() {
* struct Queue pollingQueue;
* initializeQueue(&pollingQueue);
* int choice, id, age;
* char name[50];
* do {
* printf("\n1. Register Voter\n");
* printf("2. Verify Voter\n");
* printf("3. Voter Vote\n");
* printf("4. Display Voter Queue\n");
* printf("5. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* if (isQueueFull(&pollingQueue)) {
* printf("Polling booth is full. Cannot register more voters.\n");
* } else {
* printf("Enter Voter ID: ");
* scanf("%d", &id);
* printf("Enter Voter Name: ");
* getchar();  // To consume newline left by previous input
* fgets(name, sizeof(name), stdin);
* name[strcspn(name, "\n")] = 0; // Remove newline character
* printf("Enter Voter Age: ");
* scanf("%d", &age);
* enqueue(&pollingQueue, id, name, age);
* }
* break;
* case 2:
* verifyVoter(&pollingQueue);
* break;
* case 3:
* if (isQueueEmpty(&pollingQueue)) {
* printf("No voters left to vote.\n");
* } else {
* dequeue(&pollingQueue);
* }
* break;
* case 4:
* displayQueue(&pollingQueue);
* break;
* case 5:
* printf("Exiting system...\n");
* break;
* default:
* printf("Invalid choice. Try again.\n");
* }
* } while(choice != 5);
* return 0;
* }
* // Initialize the queue
* void initializeQueue(struct Queue\* q) {
* q->front = 0;
* q->rear = -1;
* }
* // Check if the queue is empty
* int isQueueEmpty(struct Queue\* q) {
* return (q->front > q->rear);
* }
* // Check if the queue is full
* int isQueueFull(struct Queue\* q) {
* return (q->rear == MAX\_VOTERS - 1);
* }
* // Enqueue a voter into the queue
* void enqueue(struct Queue\* q, int id, const char\* name, int age) {
* if (isQueueFull(q)) {
* printf("Polling booth is full, cannot register voter.\n");
* return;
* }
* q->rear++;
* q->voters[q->rear].id = id;
* strcpy(q->voters[q->rear].name, name);
* q->voters[q->rear].age = age;
* q->voters[q->rear].verified = 0;  // Not verified initially
* printf("Voter registered: %s\n", name);
* }
* // Dequeue a voter and simulate voting
* void dequeue(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("No voters left in the queue.\n");
* return;
* }
* struct Voter voter = q->voters[q->front];
* printf("Voter %s (ID: %d) is voting now.\n", voter.name, voter.id);
* q->front++;
* }
* // Verify the voter by checking the age
* void verifyVoter(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("No voters to verify.\n");
* return;
* }
* printf("Enter Voter ID to verify: ");
* int id;
* scanf("%d", &id);
* for (int i = q->front; i <= q->rear; i++) {
* if (q->voters[i].id == id) {
* if (q->voters[i].age >= 18 && q->voters[i].verified == 0) {
* q->voters[i].verified = 1;
* printf("Voter %s (ID: %d) is verified.\n", q->voters[i].name, id);
* return;
* } else if (q->voters[i].verified == 1) {
* printf("Voter %s (ID: %d) has already been verified.\n", q->voters[i].name, id);
* return;
* } else {
* printf("Voter %s (ID: %d) is not eligible to vote.\n", q->voters[i].name, id);
* return;
* }
* }
* }
* printf("Voter ID not found.\n");
* }
* // Display the current queue of voters
* void displayQueue(struct Queue\* q) {
* if (isQueueEmpty(q)) {
* printf("No voters in the queue.\n");
* return;
* }
* printf("Current Voter Queue:\n");
* for (int i = q->front; i <= q->rear; i++) {
* printf("ID: %d, Name: %s, Age: %d, Verified: %s\n",
* q->voters[i].id, q->voters[i].name, q->voters[i].age,
* q->voters[i].verified ? "Yes" : "No");
* }
* }

* 18. \*\*Hospital Emergency Room Queue\*\*: Develop a queue using linked lists to manage patients in a hospital emergency room. The system should prioritize patients based on the severity of their condition and manage multiple doctors.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* // Define maximum number of doctors
* #define MAX\_DOCTORS 3
* // Structure to represent a patient
* struct Patient {
* int id;
* char name[50];
* int severity;  // Higher value means more severe
* struct Patient\* next;
* };
* // Structure to represent the emergency room queue
* struct ERQueue {
* struct Patient\* front;
* };
* // Function prototypes
* void initializeQueue(struct ERQueue\* q);
* int isQueueEmpty(struct ERQueue\* q);
* void enqueue(struct ERQueue\* q, int id, const char\* name, int severity);
* void dequeue(struct ERQueue\* q, int doctorId);
* void displayQueue(struct ERQueue\* q);
* void handlePatients(struct ERQueue\* q);
* int main() {
* struct ERQueue erQueue;
* initializeQueue(&erQueue);
* int choice, id, severity;
* char name[50];
* do {
* printf("\n1. Register Patient\n");
* printf("2. Assign Doctor and Treat Patient\n");
* printf("3. Display ER Queue\n");
* printf("4. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* printf("Enter Patient ID: ");
* scanf("%d", &id);
* printf("Enter Patient Name: ");
* getchar();  // To consume newline left by previous input
* fgets(name, sizeof(name), stdin);
* name[strcspn(name, "\n")] = 0;  // Remove newline character
* printf("Enter Severity (1 to 10, 10 being most severe): ");
* scanf("%d", &severity);
* enqueue(&erQueue, id, name, severity);
* break;
* case 2:
* handlePatients(&erQueue);
* break;
* case 3:
* displayQueue(&erQueue);
* break;
* case 4:
* printf("Exiting system...\n");
* break;
* default:
* printf("Invalid choice. Try again.\n");
* }
* } while(choice != 4);
* return 0;
* }
* // Initialize the ER queue
* void initializeQueue(struct ERQueue\* q) {
* q->front = NULL;
* }
* // Check if the ER queue is empty
* int isQueueEmpty(struct ERQueue\* q) {
* return q->front == NULL;
* }
* // Enqueue a patient in the ER queue (priority based on severity)
* void enqueue(struct ERQueue\* q, int id, const char\* name, int severity) {
* struct Patient\* newPatient = (struct Patient\*)malloc(sizeof(struct Patient));
* if (!newPatient) {
* printf("Memory allocation failed.\n");
* return;
* }
* newPatient->id = id;
* strcpy(newPatient->name, name);
* newPatient->severity = severity;
* newPatient->next = NULL;
* if (isQueueEmpty(q) || q->front->severity < severity) {
* newPatient->next = q->front;
* q->front = newPatient;
* } else {
* struct Patient\* temp = q->front;
* while (temp->next != NULL && temp->next->severity >= severity) {
* temp = temp->next;
* }
* newPatient->next = temp->next;
* temp->next = newPatient;
* }
* printf("Patient %s (ID: %d) registered with severity %d.\n", name, id, severity);
* }
* // Dequeue a patient for treatment (by doctor)
* void dequeue(struct ERQueue\* q, int doctorId) {
* if (isQueueEmpty(q)) {
* printf("No patients in the queue.\n");
* return;
* }
* struct Patient\* patientToTreat = q->front;
* printf("Doctor %d is treating Patient %s (ID: %d) with severity %d.\n", doctorId, patientToTreat->name, patientToTreat->id, patientToTreat->severity);
* q->front = q->front->next;
* free(patientToTreat);
* }
* // Simulate treating patients for multiple doctors
* void handlePatients(struct ERQueue\* q) {
* int doctorId;
* printf("Enter Doctor ID (1 to %d): ", MAX\_DOCTORS);
* scanf("%d", &doctorId);
* if (doctorId < 1 || doctorId > MAX\_DOCTORS) {
* printf("Invalid doctor ID.\n");
* return;
* }
* dequeue(q, doctorId);
* }
* // Display all patients in the ER queue
* void displayQueue(struct ERQueue\* q) {
* if (isQueueEmpty(q)) {
* printf("No patients in the queue.\n");
* return;
* }
* struct Patient\* temp = q->front;
* printf("Patients in the ER Queue (Priority based on Severity):\n");
* while (temp) {
* printf("ID: %d, Name: %s, Severity: %d\n", temp->id, temp->name, temp->severity);
* temp = temp->next;
* }
* }
* 19. \*\*Political Survey Data Collection\*\*: Implement a queue using arrays to manage data collection for a political survey. The system should handle multiple surveyors collecting data simultaneously and ensure data consistency.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* #define MAX\_SIZE 100  // Maximum size of the queue
* // Structure to represent survey data
* struct SurveyData {
* int surveyorId;
* char respondentName[50];
* int age;
* char response[100];
* };
* // Structure to represent the queue
* struct SurveyQueue {
* struct SurveyData data[MAX\_SIZE];
* int front;
* int rear;
* };
* // Function prototypes
* void initializeQueue(struct SurveyQueue\* q);
* int isQueueEmpty(struct SurveyQueue\* q);
* int isQueueFull(struct SurveyQueue\* q);
* void enqueue(struct SurveyQueue\* q, int surveyorId, const char\* respondentName, int age, const char\* response);
* void dequeue(struct SurveyQueue\* q);
* void displayQueue(struct SurveyQueue\* q);
* int main() {
* struct SurveyQueue surveyQueue;
* initializeQueue(&surveyQueue);
* int choice, surveyorId, age;
* char respondentName[50], response[100];
* do {
* printf("\n1. Collect Survey Data\n");
* printf("2. Process Survey Data\n");
* printf("3. Display Survey Data\n");
* printf("4. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* printf("Enter Surveyor ID: ");
* scanf("%d", &surveyorId);
* printf("Enter Respondent Name: ");
* getchar();  // To consume newline character
* fgets(respondentName, sizeof(respondentName), stdin);
* respondentName[strcspn(respondentName, "\n")] = 0;  // Remove newline character
* printf("Enter Respondent Age: ");
* scanf("%d", &age);
* printf("Enter Survey Response: ");
* getchar();  // To consume newline character
* fgets(response, sizeof(response), stdin);
* response[strcspn(response, "\n")] = 0;  // Remove newline character
* enqueue(&surveyQueue, surveyorId, respondentName, age, response);
* break;
* case 2:
* dequeue(&surveyQueue);
* break;
* case 3:
* displayQueue(&surveyQueue);
* break;
* case 4:
* printf("Exiting system...\n");
* break;
* default:
* printf("Invalid choice. Try again.\n");
* }
* } while(choice != 4);
* return 0;
* }
* // Initialize the queue
* void initializeQueue(struct SurveyQueue\* q) {
* q->front = 0;
* q->rear = -1;
* }
* // Check if the queue is empty
* int isQueueEmpty(struct SurveyQueue\* q) {
* return q->front > q->rear;
* }
* // Check if the queue is full
* int isQueueFull(struct SurveyQueue\* q) {
* return q->rear == MAX\_SIZE - 1;
* }
* // Enqueue a new survey data entry
* void enqueue(struct SurveyQueue\* q, int surveyorId, const char\* respondentName, int age, const char\* response) {
* if (isQueueFull(q)) {
* printf("Queue is full. Cannot collect more survey data.\n");
* } else {
* q->rear++;
* q->data[q->rear].surveyorId = surveyorId;
* strcpy(q->data[q->rear].respondentName, respondentName);
* q->data[q->rear].age = age;
* strcpy(q->data[q->rear].response, response);
* printf("Survey data collected successfully.\n");
* }
* }
* // Dequeue a survey data entry (process the data)
* void dequeue(struct SurveyQueue\* q) {
* if (isQueueEmpty(q)) {
* printf("No survey data to process.\n");
* } else {
* struct SurveyData processedData = q->data[q->front];
* printf("\nProcessing Survey Data...\n");
* printf("Surveyor ID: %d\n", processedData.surveyorId);
* printf("Respondent Name: %s\n", processedData.respondentName);
* printf("Respondent Age: %d\n", processedData.age);
* printf("Response: %s\n", processedData.response);
* q->front++;
* }
* }
* // Display all survey data in the queue
* void displayQueue(struct SurveyQueue\* q) {
* if (isQueueEmpty(q)) {
* printf("No survey data to display.\n");
* } else {
* printf("\nSurvey Data:\n");
* for (int i = q->front; i <= q->rear; i++) {
* printf("\nSurveyor ID: %d\n", q->data[i].surveyorId);
* printf("Respondent Name: %s\n", q->data[i].respondentName);
* printf("Respondent Age: %d\n", q->data[i].age);
* printf("Response: %s\n", q->data[i].response);
* printf("-----------------------------\n");
* }
* }
* }

* 20. \*\*Financial Market Data Analysis\*\*: Use linked lists to implement a queue for analyzing financial market data. The system should handle large volumes of data, perform real-time analysis, and generate insights for decision-making.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* // Structure to represent a market data entry
* struct MarketData {
* char symbol[10];
* double price;
* int volume;
* char timestamp[20];
* struct MarketData\* next;
* };
* // Queue structure to store market data
* struct Queue {
* struct MarketData\* front;
* struct MarketData\* rear;
* };
* // Function prototypes
* void initializeQueue(struct Queue\* q);
* int isQueueEmpty(struct Queue\* q);
* void enqueue(struct Queue\* q, const char\* symbol, double price, int volume, const char\* timestamp);
* void dequeue(struct Queue\* q);
* void displayQueue(struct Queue\* q);
* void analyzeData(struct Queue\* q);
* int main() {
* struct Queue marketDataQueue;
* initializeQueue(&marketDataQueue);
* int choice;
* char symbol[10], timestamp[20];
* double price;
* int volume;
* do {
* printf("\n1. Add Market Data\n");
* printf("2. Process Market Data\n");
* printf("3. Display Market Data\n");
* printf("4. Analyze Data\n");
* printf("5. Exit\n");
* printf("Enter your choice: ");
* scanf("%d", &choice);
* switch(choice) {
* case 1:
* printf("Enter Stock Symbol: ");
* scanf("%s", symbol);
* printf("Enter Price: ");
* scanf("%lf", &price);
* printf("Enter Volume: ");
* scanf("%d", &volume);
* printf("Enter Timestamp: ");
* scanf("%s", timestamp);
* enqueue(&marketDataQueue, symbol, price, volume, timestamp);
* break;
* case 2:
* dequeue(&marketDataQueue);
* break;
* case 3:
* displayQueue(&marketDataQueue);
* break;
* case 4:
* analyzeData(&marketDataQueue);
* break;
* case 5:
* printf("Exiting system...\n");
* break;
* default:
* printf("Invalid choice. Try again.\n");
* }
* } while(choice != 5);
* return 0;
* }
* // Initialize the queue
* void initializeQueue(struct Queue\* q) {
* q->front = q->rear = NULL;
* }
* // Check if the queue is empty
* int isQueueEmpty(struct Queue\* q) {
* return q->front == NULL;
* }
* // Enqueue market data
* void enqueue(struct Queue\* q, const char\* symbol, double price, int volume, const char\* timestamp) {
* struct MarketData\* newData = (struct MarketData\*)malloc(sizeof(struct MarketData));
* if(newData == NULL) {
* printf("Memory allocation failed\n");
* return;
* }
* strcpy(newData->symbol, symbol);
* newData->price = price;
* newData->volume = volume;
* strcpy(newData->timestamp, timestamp);
* newData->next = NULL;
* if(isQueueEmpty(q)) {
* q->front = q->rear = newData;
* } else {
* q->rear->next = newData;
* q->rear = newData;
* }
* printf("Market data added successfully.\n");
* }
* // Dequeue market data (process the oldest data)
* void dequeue(struct Queue\* q) {
* if(isQueueEmpty(q)) {
* printf("No data to process.\n");
* return;
* }
* struct MarketData\* temp = q->front;
* q->front = q->front->next;
* printf("Processing data: %s %lf %d %s\n", temp->symbol, temp->price, temp->volume, temp->timestamp);
* free(temp);
* }
* // Display all market data in the queue
* void displayQueue(struct Queue\* q) {
* if(isQueueEmpty(q)) {
* printf("No data in the queue.\n");
* return;
* }
* struct MarketData\* temp = q->front;
* printf("\nMarket Data in Queue:\n");
* while(temp != NULL) {
* printf("Symbol: %s, Price: %.2f, Volume: %d, Timestamp: %s\n", temp->symbol, temp->price, temp->volume, temp->timestamp);
* temp = temp->next;
* }
* }
* // Perform real-time analysis of the market data (calculate average price)
* void analyzeData(struct Queue\* q) {
* if(isQueueEmpty(q)) {
* printf("No data to analyze.\n");
* return;
* }
* struct MarketData\* temp = q->front;
* double totalPrice = 0.0;
* int count = 0;
* while(temp != NULL) {
* totalPrice += temp->price;
* count++;
* temp = temp->next;
* }
* if(count > 0) {
* double averagePrice = totalPrice / count;
* printf("Average Price of Market Data: %.2f\n", averagePrice);
* }
* }