QUEUE
==========

CODE
==========

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

struct Queue{
    int size;
    int front;
    int rear;
    int *Q;
};
 void create(struct Queue *,int);
 void enqueue(struct Queue *,int);
 void display(struct Queue);
 int dequeue(struct Queue *);
int main()
{
   struct Queue q;
   create(&q,5);
   enqueue(&q,7);
   enqueue(&q,8);
   enqueue(&q,9);
   display(q);
   printf("%d ",dequeue(&q));
   printf("\n");
    display(q);
    return 0;
}
 void create(struct Queue *q,int size){
     q->size = size;
     q->front = q->rear = -1;
     q->Q = (int *) malloc(q->size * sizeof(int));
 }

void enqueue(struct Queue *q,int x){
    if(q->rear==q->size-1){
        printf("Queue is full");
    }
    else{
        q->rear++;
        q->Q[q->rear]=x;
    }
}
void display(struct Queue q){
    int i;
    for(i=q.front+1;i<=q.rear;i++){
        printf("%d->",q.Q[i]);
    }
    printf("\n");
}
```

```
int dequeue(struct Queue *q){
    int x = -1;
    if(q->front == q->rear){
        printf("Queue is Empty");
    }else{
        q->front++;
        x = q->Q[q->front];
    }
    return x;
}
```

SET OF PROBLEMS
==========

1.Student Admission Queue: Write a program to simulate a student admission process. Implement a queue using arrays to manage students waiting for admission. Include operations to enqueue (add a student), dequeue (admit a student), and display the current queue of students.

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

#define MAX 5  // Maximum number of students that can wait in the queue

// Structure to represent a student
struct Student {
    int id;
    char name[100];
};

// Queue structure
struct Queue {
    struct Student students[MAX];
    int front;
    int rear;
};

// 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 isFull(struct Queue* q) {
    if (q->rear == MAX - 1)
        return 1;
    return 0;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
    if (q->front == -1)
        return 1;
    return 0;
```

```
}

// Function to add a student to the queue (enqueue)
void enqueue(struct Queue* q, int id, const char* name) {
    if (isFull(q)) {
        printf("Queue is full. Cannot admit more students.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;  // First student in the queue
    }
    q->rear++;
    q->students[q->rear].id = id;
    strncpy(q->students[q->rear].name, name, sizeof(q->students[q->rear].name) - 1);
    q->students[q->rear].name[sizeof(q->students[q->rear].name) - 1] = '\0';  // Ensure null-termination
    printf("Student %s with ID %d added to the queue.\n", name, id);
}

// Function to remove a student from the queue (dequeue)
void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No students to admit.\n");
        return;
    }
    printf("Admitting student %s with ID %d.\n", q->students[q->front].name, q->students[q->front].id);
    // Move the front pointer to the next student in the queue
    for (int i = 0; i < q->rear; i++) {
        q->students[i] = q->students[i + 1];
    }
    q->rear--;
    if (q->rear == -1) {
        q->front = -1;  // Reset the queue if it's empty
    }
}

// Function to display the current queue
void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Current Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("ID: %d, Name: %s\n", q->students[i].id, q->students[i].name);
    }
}

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

    // Add students to the queue
    enqueue(&q, 101, "Alice");
    enqueue(&q, 102, "Bob");
    enqueue(&q, 103, "Charlie");
```

```
    // Display the queue
    displayQueue(&q);

    // Admit a student
    dequeue(&q);

    // Display the updated queue
    displayQueue(&q);

    // Add more students
    enqueue(&q, 104, "David");
    enqueue(&q, 105, "Eve");

    // Display the updated queue
    displayQueue(&q);

    return 0;
}
```

2.Library Book Borrowing Queue: Develop a program that simulates a library's book borrowing system. Use a queue to manage students waiting to borrow books. Include functions to add a student to the queue, remove a student after borrowing a book, and display the queue status

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

#define MAX_QUEUE_SIZE 5  // Define the maximum size of the queue

// Structure for a Student
typedef struct {
    int student_id;
    char name[50];
} Student;

// Structure for Queue
typedef struct {
    Student queue[MAX_QUEUE_SIZE];
    int front, rear;
} Queue;

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

// Check if the queue is full
int isFull(Queue* q) {
    return (q->rear + 1) % MAX_QUEUE_SIZE == q->front;
}

// Check if the queue is empty
int isEmpty(Queue* q) {
    return q->front == -1;
```

```
}

// Add a student to the queue
void enqueue(Queue* q, int student_id, const char* name) {
    if (isFull(q)) {
        printf("Queue is full! Cannot add more students.\n");
        return;
    }

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

    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->queue[q->rear].student_id = student_id;
    // Copy name using scanf
    snprintf(q->queue[q->rear].name, sizeof(q->queue[q->rear].name), "%s", name);
    printf("Student '%s' added to the queue.\n", name);
}

// Remove a student from the queue after borrowing the book
void dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! No students to remove.\n");
        return;
    }

    printf("Student '%s' (ID: %d) borrowed the book and left the queue.\n",
           q->queue[q->front].name, q->queue[q->front].student_id);

    if (q->front == q->rear) {
        q->front = q->rear = -1; // Queue is now empty
    } else {
        q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    }
}

// Display the status of the queue
void displayQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Students in the queue:\n");
    int i = q->front;
    while (i != q->rear) {
        printf("ID: %d, Name: %s\n", q->queue[i].student_id, q->queue[i].name);
        i = (i + 1) % MAX_QUEUE_SIZE;
    }
    // Display the last student
    printf("ID: %d, Name: %s\n", q->queue[q->rear].student_id, q->queue[q->rear].name);
}

int main() {
```

```
    Queue q;
    initQueue(&q);

    int choice, student_id;
    char name[50];

    while (1) {
        printf("\nLibrary Book Borrowing System\n");
        printf("1. Add student to queue\n");
        printf("2. Remove student from queue after borrowing book\n");
        printf("3. Display queue status\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Add student to the queue
                printf("Enter student ID: ");
                scanf("%d", &student_id);
                printf("Enter student name: ");
                // Using scanf to read a single word name (without spaces)
                scanf("%s", name);
                enqueue(&q, student_id, name);
                break;
            case 2:
                // Remove student from the queue
                dequeue(&q);
                break;
            case 3:
                // Display queue status
                displayQueue(&q);
                break;
            case 4:
                // Exit the program
                printf("Exiting the system...\n");
                exit(0);
                break;
            default:
                printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```

3.Cafeteria Token System: Create a program that simulates a cafeteria token system for students. Implement a queue using arrays to manage students waiting for their turn. Provide operations to issue tokens (enqueue), serve students (dequeue), and display the queue of students.

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

#define MAX_QUEUE_SIZE 10  // Maximum number of students in the queue
```

```
// Define a structure for the queue
typedef struct {
    int front, rear;
    int queue[MAX_QUEUE_SIZE];
} Queue;

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

// Function to check if the queue is full
int isFull(Queue *q) {
    return (q->rear == MAX_QUEUE_SIZE - 1);
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return (q->front == -1);
}

// Function to issue a token (enqueue)
void issueToken(Queue *q, int studentId) {
    if (isFull(q)) {
        printf("The queue is full. Cannot issue token.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;  // If the queue was empty, set front to 0
    }
    q->rear++;
    q->queue[q->rear] = studentId;
    printf("Token issued to student %d.\n", studentId);
}

// Function to serve a student (dequeue)
void serveStudent(Queue *q) {
    if (isEmpty(q)) {
        printf("No students in the queue to serve.\n");
        return;
    }
    int servedStudent = q->queue[q->front];
    printf("Serving student %d.\n", servedStudent);
    q->front++;
    if (q->front > q->rear) {  // If the queue becomes empty after serving
        q->front = q->rear = -1;
    }
}

// Function to display the queue of students
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("The queue is empty.\n");
        return;
```

```
    }
    printf("Current queue: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->queue[i]);
    }
    printf("\n");
}

// Main function to test the cafeteria token system
int main() {
    Queue q;
    initQueue(&q);

    int choice, studentId;

    while (1) {
        printf("\nCafeteria Token System\n");
        printf("1. Issue Token (enqueue)\n");
        printf("2. Serve Student (dequeue)\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter student ID to issue token: ");
                scanf("%d", &studentId);
                issueToken(&q, studentId);
                break;
            case 2:
                serveStudent(&q);
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                printf("Exiting system.\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

4.Classroom Help Desk Queue: Write a program to manage a help desk queue in a classroom. Use a queue to track students waiting for assistance. Include functions to add students to the queue, remove them once helped, and view the current queue

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

```
#define MAX_QUEUE_SIZE 10

// Structure to represent a student
typedef struct {
    int id;
    char name[50];
} Student;

// Structure to represent a queue
typedef struct {
    Student students[MAX_QUEUE_SIZE];
    int front;
    int rear;
} Queue;

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

// Function to check if the queue is empty
int isQueueEmpty(Queue *q) {
    return (q->front == -1);
}

// Function to check if the queue is full
int isQueueFull(Queue *q) {
    return (q->rear == MAX_QUEUE_SIZE - 1);
}

// Function to add a student to the queue
void enqueue(Queue *q, int id, const char *name) {
    if (isQueueFull(q)) {
        printf("Queue is full! Cannot add more students.\n");
    } else {
        if (q->front == -1) {
            q->front = 0; // If the queue is empty, initialize the front
        }
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
        printf("Student %s added to the queue.\n", name);
    }
}

// Function to remove a student from the queue (after helping them)
void dequeue(Queue *q) {
    if (isQueueEmpty(q)) {
        printf("Queue is empty! No students to help.\n");
    } else {
        printf("Helping student %s (ID: %d)...\n", q->students[q->front].name, q->students[q->front].id);
        for (int i = q->front; i < q->rear; i++) {
            q->students[i] = q->students[i + 1]; // Shift the students down
        }
```

```
        q->rear--;
        if (q->rear == -1) {
            q->front = -1; // Queue is now empty
        }
    }
}

// Function to display the current queue
void displayQueue(Queue *q) {
    if (isQueueEmpty(q)) {
        printf("The queue is empty.\n");
    } else {
        printf("Current Queue:\n");
        for (int i = q->front; i <= q->rear; i++) {
            printf("ID: %d, Name: %s\n", q->students[i].id, q->students[i].name);
        }
    }
}

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

    int choice, id;
    char name[50];

    do {
        printf("\nClassroom Help Desk Queue Management\n");
        printf("1. Add Student to Queue\n");
        printf("2. Help Student (Dequeue)\n");
        printf("3. View Current Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (isQueueFull(&q)) {
                    printf("Queue is full. Cannot add more students.\n");
                } else {
                    printf("Enter student ID: ");
                    scanf("%d", &id);
                    printf("Enter student name: ");
                    getchar(); // To clear the newline left by scanf
                    fgets(name, sizeof(name), stdin);
                    name[strcspn(name, "\n")] = 0; // Remove the newline character from input
                    enqueue(&q, id, name);
                }
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                displayQueue(&q);
                break;
```

```
            case 4:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

5.Exam Registration Queue: Develop a program to simulate the exam registration process. Use a queue to manage the order of student registrations. Implement operations to add students to the queue, process their registration, and display the queue status.

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

#define MAX_QUEUE_SIZE 5  // Maximum number of students in the queue

// Define the queue structure
typedef struct {
    int front, rear, size;
    int queue[MAX_QUEUE_SIZE];
} Queue;

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

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

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

// Add a student to the queue (registration)
void enqueue(Queue *q, int studentId) {
    if (isFull(q)) {
        printf("Queue is full. Cannot register more students.\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->queue[q->rear] = studentId;
    q->size++;
    printf("Student %d added to the registration queue.\n", studentId);
}
```

```
// Process (dequeue) a student’s registration
void dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No students to process.\n");
        return;
    }
    int studentId = q->queue[q->front];
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    q->size--;
    printf("Student %d registration processed.\n", studentId);
}

// Display the current queue status
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Current queue status: ");
    for (int i = 0; i < q->size; i++) {
        int index = (q->front + i) % MAX_QUEUE_SIZE;
        printf("%d ", q->queue[index]);
    }
    printf("\n");
}

// Main function to test the queue functionality
int main() {
    Queue q;
    initQueue(&q);

    int choice, studentId;

    while (1) {
        printf("\nExam Registration Queue Operations:\n");
        printf("1. Register a student\n");
        printf("2. Process a student’s registration\n");
        printf("3. Display queue status\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter student ID to register: ");
                scanf("%d", &studentId);
                enqueue(&q, studentId);
                break;

            case 2:
                dequeue(&q);
                break;

            case 3:
```

```
            displayQueue(&q);
            break;

        case 4:
            printf("Exiting program...\n");
            exit(0);

        default:
            printf("Invalid choice, please try again.\n");
      }
   }

   return 0;
}
```

6.School Bus Boarding Queue: Create a program that simulates the boarding process of a school bus. Implement a queue to manage the order in which students board the bus. Include functions to enqueue students as they arrive and dequeue them as they board.

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

// Define the maximum number of students
#define MAX_QUEUE_SIZE 10

// Structure to represent the Queue
struct Queue {
   int front, rear;
   int students[MAX_QUEUE_SIZE];
};

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

// Function to check if the queue is full
int isFull(struct Queue *q) {
   return q->rear == MAX_QUEUE_SIZE - 1;
}

// Function to check if the queue is empty
int isEmpty(struct Queue *q) {
   return q->front == -1 || q->front > q->rear;
}

// Function to enqueue a student
void enqueue(struct Queue *q, int studentId) {
   if (isFull(q)) {
      printf("Queue is full! No more students can board.\n");
   } else {
      if (q->front == -1) {
         q->front = 0;  // Set front to 0 if the queue is empty
      }
```

```
        q->rear++;
        q->students[q->rear] = studentId;
        printf("Student %d has arrived and is waiting to board.\n", studentId);
    }
}

// Function to dequeue a student (i.e., a student boards the bus)
void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No students in the queue to board the bus.\n");
    } else {
        int studentId = q->students[q->front];
        printf("Student %d is boarding the bus.\n", studentId);
        q->front++;

        // Reset the queue if all students have boarded
        if (q->front > q->rear) {
            initQueue(q);
        }
    }
}

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

    int choice, studentId;

    while (1) {
        printf("\nSchool Bus Boarding System\n");
        printf("1. Student arrives\n");
        printf("2. Student boards the bus\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter student ID: ");
                scanf("%d", &studentId);
                enqueue(&q, studentId);
                break;

            case 2:
                dequeue(&q);
                break;

            case 3:
                printf("Exiting the program.\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
```

```
    return 0;
}
```

7.Counseling Session Queue: Write a program to manage a queue for students waiting for a counseling session. Use an array-based queue to keep track of the students, with operations to add (enqueue) and serve (dequeue) students, and display the queue.

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

#define MAX_SIZE 5  // Maximum size of the queue

// Define the queue structure
struct Queue {
    int front, rear;
    int items[MAX_SIZE];
};

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

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
    return (q->front == -1 || q->front > q->rear);
}

// Function to add a student to the queue (enqueue)
void enqueue(struct Queue* q, int studentID) {
    if (isFull(q)) {
        printf("Queue is full! Cannot add more students.\n");
    } else {
        if (q->front == -1) {
            q->front = 0;  // First student being added
        }
        q->rear++;
        q->items[q->rear] = studentID;
        printf("Student %d added to the queue.\n", studentID);
    }
}

// Function to serve a student (dequeue)
void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! No students to serve.\n");
    } else {
        int servedStudent = q->items[q->front];
```

```
        printf("Serving student %d.\n", servedStudent);
        q->front++;
        if (q->front > q->rear) {
            q->front = q->rear = -1;  // Reset queue if empty
        }
    }
}

// Function to display the current queue
void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
    } else {
        printf("Current queue: ");
        for (int i = q->front; i <= q->rear; i++) {
            printf("%d ", q->items[i]);
        }
        printf("\n");
    }
}

// Main function to demonstrate the queue operations
int main() {
    struct Queue q;
    initializeQueue(&q);

    int choice, studentID;

    do {
        printf("\nCounseling Session Queue\n");
        printf("1. Add a student to the queue (enqueue)\n");
        printf("2. Serve a student (dequeue)\n");
        printf("3. Display the queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter student ID to add to the queue: ");
                scanf("%d", &studentID);
                enqueue(&q, studentID);
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
```

```
    } while (choice != 4);

    return 0;
}
```

8.Sports Event Registration Queue: Develop a program that manages the registration queue for a school sports event. Use a queue to handle the order of student registrations, with functions to add, process, and display the queue of registered students.

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

#define MAX 5  // Maximum number of students that can register

// Define the queue structure
typedef struct {
    char students[MAX][100];  // Array to hold student names
    int front;
    int rear;
} Queue;

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

// Function to check if the queue is full
int isFull(Queue *q) {
    return (q->rear == MAX - 1);
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return (q->front == -1);
}

// Function to add a student to the queue (enqueue)
void enqueue(Queue *q, const char *studentName) {
    if (isFull(q)) {
        printf("Queue is full. Cannot register more students.\n");
        return;
    }

    if (q->front == -1) {  // If the queue is empty, set front to 0
        q->front = 0;
    }

    q->rear++;
    strcpy(q->students[q->rear], studentName);  // Add student to the queue
    printf("%s has been successfully registered.\n", studentName);
}

// Function to remove a student from the queue (dequeue)
```

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

    printf("%s has been processed and removed from the queue.\n", q->students[q->front]);
    for (int i = 0; i < q->rear; i++) {
        strcpy(q->students[i], q->students[i + 1]);  // Shift all students one step ahead
    }

    q->rear--;  // Reduce the rear index
    if (q->rear == -1) {
        q->front = -1;  // If the queue is empty, reset the front pointer
    }
}

// Function to display the students in the queue
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No students to display.\n");
        return;
    }

    printf("Students in the registration queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%s\n", q->students[i]);
    }
}

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

    int choice;
    char studentName[100];

    do {
        printf("\nSports Event Registration Queue Menu:\n");
        printf("1. Register a student (enqueue)\n");
        printf("2. Process a registration (dequeue)\n");
        printf("3. Display queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar();  // To consume the newline character left by scanf

        switch (choice) {
            case 1:
                printf("Enter the student’s name: ");
                fgets(studentName, sizeof(studentName), stdin);
                studentName[strcspn(studentName, "\n")] = ’\0’;  // Remove the newline character
                enqueue(&q, studentName);
                break;
            case 2:
```

```
            dequeue(&q);
            break;
        case 3:
            displayQueue(&q);
            break;
        case 4:
            printf("Exiting the program.\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   } while (choice != 4);

   return 0;
}
```

9.Laboratory Equipment Checkout Queue: Create a program to simulate a queue for students waiting to check out laboratory equipment. Implement operations to add students to the queue, remove them once they receive equipment, and view the current queue.

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

// Define a struct for a student
typedef struct Student {
   int id;                  // Student ID
   char name[50];           // Student name
   struct Student *next;    // Pointer to the next student in the queue
} Student;

// Function prototypes
void enqueue(Student **front, Student **rear, int id, const char *name);
void dequeue(Student **front);
void displayQueue(Student *front);

int main() {
   Student *front = NULL, *rear = NULL;  // Initialize empty queue

   int choice, id;
   char name[50];

   while (1) {
      // Menu options
      printf("\nLaboratory Equipment Checkout Queue\n");
      printf("1. Add student to queue\n");
      printf("2. Remove student from queue\n");
      printf("3. View current queue\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            // Add student to the queue
```

```
            printf("Enter student ID: ");
            scanf("%d", &id);
            printf("Enter student name: ");
            getchar();  // To consume the newline character left by scanf
            fgets(name, sizeof(name), stdin);
            name[strcspn(name, "\n")] = '\0';  // Remove newline character from input
            enqueue(&front, &rear, id, name);
            break;

        case 2:
            // Remove student from the queue
            dequeue(&front);
            break;

        case 3:
            // Display the current queue
            displayQueue(front);
            break;

        case 4:
            // Exit the program
            printf("Exiting the program...\n");
            exit(0);
            break;

        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}

// Function to add a student to the queue
void enqueue(Student **front, Student **rear, int id, const char *name) {
   Student *newStudent = (Student *)malloc(sizeof(Student));
   if (newStudent == NULL) {
      printf("Memory allocation failed. Could not add student.\n");
      return;
   }

   newStudent->id = id;
   strcpy(newStudent->name, name);
   newStudent->next = NULL;

   if (*rear == NULL) {
      // If the queue is empty, both front and rear will point to the new student
      *front = *rear =
```

10.Parent-Teacher Meeting Queue: Write a program to manage a queue for a parent-teacher meeting. Use a queue to organize the order in which parents meet the teacher. Include functions to enqueue parents, dequeue them after the meeting, and display the queue status.

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

```
#define MAX_QUEUE_SIZE 5

// Queue structure
typedef struct {
    int front, rear;
    char *queue[MAX_QUEUE_SIZE];
} Queue;

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

// Function to check if the queue is full
int isFull(Queue *q) {
    return q->rear == MAX_QUEUE_SIZE - 1;
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function to enqueue a parent
void enqueue(Queue *q, char *parentName) {
    if (isFull(q)) {
        printf("Queue is full. Cannot add more parents.\n");
    } else {
        if (q->front == -1) {
            q->front = 0;  // Queue was empty, so front is set to 0
        }
        q->rear++;
        q->queue[q->rear] = parentName;
        printf("Added %s to the queue.\n", parentName);
    }
}

// Function to dequeue a parent after the meeting
void dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No parents to meet.\n");
    } else {
        printf("%s has completed the meeting and left the queue.\n", q->queue[q->front]);
        q->front++;
    }
}

// Function to display the current status of the queue
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
    } else {
        printf("Current Queue: ");
```

```
        for (int i = q->front; i <= q->rear; i++) {
            printf("%s ", q->queue[i]);
        }
        printf("\n");
    }
}

// Main function to test the program
int main() {
    Queue q;
    initQueue(&q);

    int choice;
    char name[50];

    while (1) {
        printf("\nParent-Teacher Meeting Queue System\n");
        printf("1. Enqueue a Parent\n");
        printf("2. Dequeue (Parent completed the meeting)\n");
        printf("3. Display Queue Status\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (isFull(&q)) {
                    printf("Queue is full. Please wait.\n");
                } else {
                    printf("Enter parent's name: ");
                    scanf("%s", name);
                    enqueue(&q, name);
                }
                break;

            case 2:
                dequeue(&q);
                break;

            case 3:
                displayQueue(&q);
                break;

            case 4:
                printf("Exiting program.\n");
                exit(0);
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    return 0;
}
```

QUEUE USING LINKED LIST

CODE
=======

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

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

void enqueue(int);
int dequeue();
void display();


int main(){
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    enqueue(50);
    display();
    printf("%d \n",dequeue());
    display();
    return 0;
}

void enqueue(int x){
    struct Node *t;
    t = (struct Node*)malloc(sizeof(struct Node));
    if(t == NULL){
        printf("Queue is full \n");
    }else{
        t->data = x;
        t->next = NULL;
        if(front == NULL){
            front = rear = t;
        }else{
            rear->next = t;
            rear = t;
        }
    }
}

void display(){
    struct Node *p = front;
    while(p){
        printf("%d -> ",p->data);
        p = p->next;
    }
```

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

int dequeue(){
    int x = -1;
    struct Node *t;
    if(front == NULL){
        printf("Queue is already empty \n");
    }else{
        x = front->data;
        t = front;
        front = front->next;
        free(t);
    }
    return x;
}
```

SET OF PROBLEMS

==========

1. Real-Time Sensor Data Processing:

Implement a queue using a linked list to store real-time data from various sensors (e.g., temperature, pressure). The system should enqueue sensor readings, process and dequeue the oldest data when a new reading arrives, and search for specific readings based on timestamps.

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

// Define the structure for the sensor reading.
typedef struct SensorReading {
    char sensorType[20]; // e.g., "Temperature" or "Pressure"
    float reading;       // The sensor's reading value
    time_t timestamp;    // Timestamp of the reading
    struct SensorReading *next; // Pointer to the next reading in the queue
} SensorReading;

// Queue structure to manage the linked list
typedef struct Queue {
    SensorReading *front; // Points to the front of the queue
    SensorReading *rear;  // Points to the rear of the queue
    int size;             // The current size of the queue
    int maxSize;          // The maximum size of the queue (to limit the data in the queue)
} Queue;

// Function to initialize the queue
Queue* initQueue(int maxSize) {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->front = q->rear = NULL;
    q->size = 0;
    q->maxSize = maxSize;
    return q;
}

// Function to create a new sensor reading node
```

```
SensorReading* createSensorReading(const char* sensorType, float reading) {
    SensorReading* newReading = (SensorReading*)malloc(sizeof(SensorReading));
    strcpy(newReading->sensorType, sensorType);
    newReading->reading = reading;
    newReading->timestamp = time(NULL); // Get current time as timestamp
    newReading->next = NULL;
    return newReading;
}

// Enqueue function: Add a new sensor reading to the queue
void enqueue(Queue* q, const char* sensorType, float reading) {
    if (q->size == q->maxSize) {
        printf("Queue is full. Dequeueing the oldest data.\n");
        dequeue(q); // Dequeue the oldest reading if the queue is full
    }

    SensorReading* newReading = createSensorReading(sensorType, reading);

    if (q->rear == NULL) {
        q->front = q->rear = newReading; // If the queue is empty, both front and rear point to the new node
    } else {
        q->rear->next = newReading;  // Add the new reading to the end of the queue
        q->rear = newReading;
    }

    q->size++;
    printf("Enqueued: %s = %.2f at %s", sensorType, reading, ctime(&(newReading->timestamp)));
}

// Dequeue function: Remove the oldest sensor reading from the queue
void dequeue(Queue* q) {
    if (q->size == 0) {
        printf("Queue is empty.\n");
        return;
    }

    SensorReading* temp = q->front;
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL; // If the queue becomes empty, set rear to NULL
    }

    printf("Dequeued: %s = %.2f at %s", temp->sensorType, temp->reading, ctime(&(temp->timestamp)));
    free(temp);
    q->size--;
}

// Function to search for a reading by timestamp
void searchByTimestamp(Queue* q, time_t timestamp) {
    SensorReading* current = q->front;
    while (current != NULL) {
        if (current->timestamp == timestamp) {
            printf("Found reading: %s = %.2f at %s", current->sensorType, current->reading,
ctime(&(current->timestamp)));
```

```
            return;
        }
        current = current->next;
    }
    printf("No reading found with the given timestamp.\n");
}

// Function to display the queue contents
void displayQueue(Queue* q) {
    if (q->size == 0) {
        printf("Queue is empty.\n");
        return;
    }

    SensorReading* current = q->front;
    while (current != NULL) {
        printf("Sensor: %s, Reading: %.2f, Timestamp: %s", current->sensorType, current->reading,
ctime(&(current->timestamp)));
        current = current->next;
    }
}

// Main function to demonstrate the queue operations
int main() {
    Queue* q = initQueue(5);  // Create a queue with a maximum size of 5

    // Enqueue some sensor data
    enqueue(q, "Temperature", 23.5);
    enqueue(q, "Pressure", 101.3);
    enqueue(q, "Temperature", 24.1);
    enqueue(q, "Pressure", 100.9);
    enqueue(q, "Temperature", 22.8);

    // Display the queue
    printf("\nQueue contents:\n");
    displayQueue(q);

    // Enqueue another reading to demonstrate dequeueing when the queue is full
    enqueue(q, "Pressure", 102.0);

    // Display the queue again
    printf("\nQueue contents after enqueueing another reading:\n");
    displayQueue(q);

    // Search for a reading by timestamp
    time_t searchTime = q->front->timestamp; // Example: Search for the first reading's timestamp
    printf("\nSearching for reading with timestamp: %s", ctime(&searchTime));
    searchByTimestamp(q, searchTime);

    // Dequeue and show the updated queue
    dequeue(q);
    printf("\nQueue contents after dequeueing:\n");
    displayQueue(q);

    return 0;
```

```
}
```

2. Task Scheduling in a Real-Time Operating System (RTOS):
Design a queue using a linked list to manage task scheduling in an RTOS. Each task should have a unique identifier, priority level, and execution time. Implement enqueue to add tasks, dequeue to remove the next task for execution, and search to find tasks by priority

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

typedef struct Task {
    int task_id;        // Unique identifier for the task
    int priority;       // Priority of the task (higher value = higher priority)
    int exec_time;      // Time required for task execution
    struct Task* next;  // Pointer to the next task in the queue
} Task;

// Queue represented as a linked list
typedef struct {
    Task* front;
    Task* rear;
} TaskQueue;

// Function to create a new task
Task* create_task(int task_id, int priority, int exec_time) {
    Task* new_task = (Task*)malloc(sizeof(Task));
    new_task->task_id = task_id;
    new_task->priority = priority;
    new_task->exec_time = exec_time;
    new_task->next = NULL;
    return new_task;
}

// Initialize the task queue
void init_queue(TaskQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Enqueue a task (insert at the rear of the queue, sorted by priority)
void enqueue(TaskQueue* queue, int task_id, int priority, int exec_time) {
    Task* new_task = create_task(task_id, priority, exec_time);

    // If the queue is empty, the new task will be both front and rear
    if (queue->rear == NULL) {
        queue->front = new_task;
        queue->rear = new_task;
        return;
    }

    // Insert the task in the correct position based on priority
    Task* temp = queue->front;
    Task* prev = NULL;

    while (temp != NULL && temp->priority >= new_task->priority) {
```

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

    // If insertion is at the front of the queue
    if (prev == NULL) {
        new_task->next = queue->front;
        queue->front = new_task;
    } else {
        prev->next = new_task;
        new_task->next = temp;
    }

    // Update the rear if necessary
    if (new_task->next == NULL) {
        queue->rear = new_task;
    }
}

// Dequeue a task (remove and return the highest priority task)
Task* dequeue(TaskQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty.\n");
        return NULL;
    }

    Task* task_to_run = queue->front;
    queue->front = queue->front->next;

    if (queue->front == NULL) {
        queue->rear = NULL; // If the queue becomes empty
    }

    return task_to_run;
}

// Search for a task by its priority level
Task* search(TaskQueue* queue, int priority) {
    Task* temp = queue->front;
    while (temp != NULL) {
        if (temp->priority == priority) {
            return temp;  // Return the first task with the matching priority
        }
        temp = temp->next;
    }
    return NULL; // Return NULL if no task with the specified priority is found
}

// Print the tasks in the queue
void print_queue(TaskQueue* queue) {
    Task* temp = queue->front;
    printf("Task Queue:\n");
    while (temp != NULL) {
        printf("Task ID: %d, Priority: %d, Exec Time: %d\n", temp->task_id, temp->priority,
temp->exec_time);
```

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

// Free the allocated memory
void free_queue(TaskQueue* queue) {
    Task* temp;
    while (queue->front != NULL) {
        temp = queue->front;
        queue->front = queue->front->next;
        free(temp);
    }
}

int main() {
    TaskQueue queue;
    init_queue(&queue);

    // Enqueue some tasks
    enqueue(&queue, 1, 5, 10);
    enqueue(&queue, 2, 8, 5);
    enqueue(&queue, 3, 2, 3);
    enqueue(&queue, 4, 10, 7);

    // Print the queue
    print_queue(&queue);

    // Dequeue a task (highest priority)
    Task* task = dequeue(&queue);
    if (task != NULL) {
        printf("\nDequeued Task: Task ID: %d, Priority: %d, Exec Time: %d\n", task->task_id, task->priority,
task->exec_time);
        free(task);
    }

    // Search for a task by priority
    task = search(&queue, 8);
    if (task != NULL) {
        printf("\nFound Task with Priority 8: Task ID: %d, Exec Time: %d\n", task->task_id,
task->exec_time);
    } else {
        printf("Task with priority 8 not found.\n");
    }

    // Print the queue after dequeue
    print_queue(&queue);

    // Free the remaining tasks
    free_queue(&queue);

    return 0;
}
```

3. Interrupt Handling Mechanism:
Create a queue using a linked list to manage interrupt requests (IRQs) in an embedded system. Each

interrupt should have a priority level and a handler function. Implement operations to enqueue new interrupts, dequeue the highest-priority interrupt, and search for interrupts by their source.

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

#define MAX_PRIORITY 10

// Define the structure for an interrupt request (IRQ)
typedef struct IRQ {
    int priority;              // Priority of the interrupt (higher number = higher priority)
    char source[20];           // Source of the interrupt (e.g., "Timer", "GPIO")
    void (*handler)(void);     // Pointer to the handler function
    struct IRQ *next;          // Pointer to the next IRQ in the list
} IRQ;

// Head of the linked list
IRQ *irqQueueHead = NULL;

// Function to create a new IRQ node
IRQ* createIRQ(int priority, const char *source, void (*handler)(void)) {
    IRQ *newIRQ = (IRQ*) malloc(sizeof(IRQ));
    if (newIRQ == NULL) {
        printf("Error: Memory allocation failed for IRQ.\n");
        exit(1);
    }
    newIRQ->priority = priority;
    snprintf(newIRQ->source, sizeof(newIRQ->source), "%s", source);
    newIRQ->handler = handler;
    newIRQ->next = NULL;
    return newIRQ;
}

// Function to enqueue a new interrupt (sorted by priority)
void enqueueIRQ(int priority, const char *source, void (*handler)(void)) {
    IRQ *newIRQ = createIRQ(priority, source, handler);

    if (irqQueueHead == NULL || irqQueueHead->priority < newIRQ->priority) {
        // Insert at the head if the list is empty or the new IRQ has higher priority
        newIRQ->next = irqQueueHead;
        irqQueueHead = newIRQ;
    } else {
        // Find the correct position to insert the IRQ based on priority
        IRQ *current = irqQueueHead;
        while (current->next != NULL && current->next->priority >= newIRQ->priority) {
            current = current->next;
        }
        newIRQ->next = current->next;
        current->next = newIRQ;
    }
}

// Function to dequeue the highest-priority interrupt (the first in the list)
IRQ* dequeueIRQ() {
    if (irqQueueHead == NULL) {
```

```
        printf("Error: IRQ queue is empty.\n");
        return NULL;
    }
    IRQ *highestPriorityIRQ = irqQueueHead;
    irqQueueHead = irqQueueHead->next;
    highestPriorityIRQ->next = NULL;
    return highestPriorityIRQ;
}

// Function to search for interrupts by their source
IRQ* searchIRQBySource(const char *source) {
    IRQ *current = irqQueueHead;
    while (current != NULL) {
        if (strcmp(current->source, source) == 0) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}

// Sample interrupt handler functions
void timerHandler() {
    printf("Handling Timer Interrupt\n");
}

void gpioHandler() {
    printf("Handling GPIO Interrupt\n");
}

void uartHandler() {
    printf("Handling UART Interrupt\n");
}

// Function to simulate handling the highest-priority IRQ
void handleHighestPriorityIRQ() {
    IRQ *irq = dequeueIRQ();
    if (irq != NULL) {
        printf("Handling IRQ from source: %s with priority: %d\n", irq->source, irq->priority);
        irq->handler();
        free(irq);  // Free the memory allocated for this IRQ
    }
}

// Main function to demonstrate the interrupt handling mechanism
int main() {
    // Enqueue some interrupts
    enqueueIRQ(5, "Timer", timerHandler);
    enqueueIRQ(8, "GPIO", gpioHandler);
    enqueueIRQ(3, "UART", uartHandler);
    enqueueIRQ(10, "Timer", timerHandler);

    // Handle interrupts in order of priority
    handleHighestPriorityIRQ();  // Should handle Timer interrupt with priority 10
    handleHighestPriorityIRQ();  // Should handle GPIO interrupt with priority 8
```

```
    handleHighestPriorityIRQ();  // Should handle Timer interrupt with priority 5
    handleHighestPriorityIRQ();  // Should handle UART interrupt with priority 3

    // Try searching for an IRQ by its source
    IRQ *foundIRQ = searchIRQBySource("GPIO");
    if (foundIRQ != NULL) {
        printf("Found IRQ from source: %s with priority: %d\n", foundIRQ->source, foundIRQ->priority);
    } else {
        printf("No IRQ found for the given source.\n");
    }

    return 0;
}
```

4. Message Passing in Embedded Communication Systems:
Implement a message queue using a linked list to handle inter-process communication in embedded systems. Each message should include a sender ID, receiver ID, and payload. Enqueue messages as they arrive, dequeue messages for processing, and search for messages from a specific sender.

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

// Define a maximum size for the payload (this can be changed as needed)
#define MAX_PAYLOAD_SIZE 100

// Structure to represent a message
typedef struct Message {
    int sender_id;
    int receiver_id;
    char payload[MAX_PAYLOAD_SIZE];
    struct Message* next; // Pointer to the next message in the queue
} Message;

// Structure to represent the message queue
typedef struct MessageQueue {
    Message* front; // Points to the front of the queue
    Message* rear;  // Points to the rear of the queue
} MessageQueue;

// Function to initialize a new message queue
void initQueue(MessageQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to create a new message
Message* createMessage(int sender_id, int receiver_id, const char* payload) {
    Message* newMessage = (Message*)malloc(sizeof(Message));
    if (newMessage == NULL) {
        printf("Error: Memory allocation failed!\n");
        return NULL;
    }
```

```
    newMessage->sender_id = sender_id;
    newMessage->receiver_id = receiver_id;
    strncpy(newMessage->payload, payload, MAX_PAYLOAD_SIZE);
    newMessage->next = NULL;

    return newMessage;
}

// Function to enqueue a message to the queue
void enqueue(MessageQueue* queue, int sender_id, int receiver_id, const char* payload) {
    Message* newMessage = createMessage(sender_id, receiver_id, payload);
    if (newMessage == NULL) {
        return;
    }

    // If the queue is empty, both front and rear will point to the new message
    if (queue->rear == NULL) {
        queue->front = queue->rear = newMessage;
        return;
    }

    // Otherwise, add the message at the end of the queue
    queue->rear->next = newMessage;
    queue->rear = newMessage;
}

// Function to dequeue a message from the queue
Message* dequeue(MessageQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty, no message to dequeue.\n");
        return NULL;
    }

    // Store the message at the front
    Message* temp = queue->front;
    queue->front = queue->front->next;

    // If the front becomes NULL, set rear to NULL (queue is empty)
    if (queue->front == NULL) {
        queue->rear = NULL;
    }

    // Return the dequeued message
    return temp;
}

// Function to search for messages from a specific sender
void searchMessagesFromSender(MessageQueue* queue, int sender_id) {
    if (queue->front == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    Message* current = queue->front;
    int found = 0;
```

```
    while (current != NULL) {
        if (current->sender_id == sender_id) {
            printf("Found message from sender %d: \n", sender_id);
            printf("Receiver ID: %d\n", current->receiver_id);
            printf("Payload: %s\n\n", current->payload);
            found = 1;
        }
        current = current->next;
    }

    if (!found) {
        printf("No messages found from sender %d.\n", sender_id);
    }
}

// Function to print the entire message queue (for debugging)
void printQueue(MessageQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    Message* current = queue->front;
    while (current != NULL) {
        printf("Sender: %d, Receiver: %d, Payload: %s\n", current->sender_id, current->receiver_id,
current->payload);
        current = current->next;
    }
}

int main() {
    MessageQueue queue;
    initQueue(&queue);

    // Enqueue some messages
    enqueue(&queue, 1, 2, "Hello from 1 to 2");
    enqueue(&queue, 2, 1, "Reply from 2 to 1");
    enqueue(&queue, 1, 3, "Message from 1 to 3");

    // Print all messages in the queue
    printf("All messages in the queue:\n");
    printQueue(&queue);

    // Search for messages from sender 1
    printf("\nSearching for messages from sender 1:\n");
    searchMessagesFromSender(&queue, 1);

    // Dequeue a message
    Message* message = dequeue(&queue);
    if (message != NULL) {
        printf("\nDequeued message: \n");
        printf("Sender: %d, Receiver: %d, Payload: %s\n", message->sender_id, message->receiver_id,
message->payload);
        free(message);  // Don’t forget to free the memory
```

```
    }

    // Print remaining messages in the queue
    printf("\nRemaining messages in the queue after dequeue:\n");
    printQueue(&queue);

    return 0;
}
```

5. Data Logging System for Embedded Devices:
Design a queue using a linked list to log data in an embedded system. Each log entry should contain a timestamp, event type, and description. Implement enqueue to add new logs, dequeue old logs when memory is low, and search for logs by event type.

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

#define MAX_DESCRIPTION_LENGTH 100
#define MAX_EVENT_TYPE_LENGTH 20

// Log Entry Structure
typedef struct LogEntry {
    time_t timestamp; // timestamp of when the event occurred
    char eventType[MAX_EVENT_TYPE_LENGTH]; // event type
    char description[MAX_DESCRIPTION_LENGTH]; // event description
    struct LogEntry* next; // pointer to the next log entry (for the linked list)
} LogEntry;

// Queue structure for the linked list
typedef struct LogQueue {
    LogEntry* front; // points to the first log entry
    LogEntry* rear;  // points to the last log entry
    int size;        // the number of log entries in the queue
} LogQueue;

// Function to create a new log entry
LogEntry* createLogEntry(const char* eventType, const char* description) {
    LogEntry* newEntry = (LogEntry*)malloc(sizeof(LogEntry));
    if (newEntry == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }

    // Set the timestamp for the log entry
    newEntry->timestamp = time(NULL);

    // Copy the event type and description
    strncpy(newEntry->eventType, eventType, MAX_EVENT_TYPE_LENGTH);
    strncpy(newEntry->description, description, MAX_DESCRIPTION_LENGTH);

    newEntry->next = NULL; // New log entry will have no next node
    return newEntry;
}
```

```
// Function to initialize the log queue
void initLogQueue(LogQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
    queue->size = 0;
}

// Function to add a log entry to the queue (enqueue)
void enqueue(LogQueue* queue, const char* eventType, const char* description) {
    LogEntry* newEntry = createLogEntry(eventType, description);
    if (newEntry == NULL) {
        return;
    }

    // If the queue is empty, set both front and rear to the new entry
    if (queue->rear == NULL) {
        queue->front = newEntry;
        queue->rear = newEntry;
    } else {
        // Otherwise, add the new entry to the end of the queue
        queue->rear->next = newEntry;
        queue->rear = newEntry;
    }

    queue->size++;
}

// Function to remove the oldest log entry from the queue (dequeue)
void dequeue(LogQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty!\n");
        return;
    }

    LogEntry* temp = queue->front;
    queue->front = queue->front->next;

    if (queue->front == NULL) {
        queue->rear = NULL; // If the queue is now empty, rear should also be NULL
    }

    free(temp); // Free the memory allocated for the log entry
    queue->size--;
}

// Function to search logs by event type
void searchLogsByEventType(LogQueue* queue, const char* eventType) {
    LogEntry* current = queue->front;
    while (current != NULL) {
        if (strcmp(current->eventType, eventType) == 0) {
            // Print log entry details if event type matches
            printf("Timestamp: %s", ctime(&current->timestamp));
            printf("Event Type: %s\n", current->eventType);
            printf("Description: %s\n\n", current->description);
```

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

// Function to print the entire queue (for debugging purposes)
void printQueue(LogQueue* queue) {
    LogEntry* current = queue->front;
    while (current != NULL) {
        printf("Timestamp: %s", ctime(&current->timestamp));
        printf("Event Type: %s\n", current->eventType);
        printf("Description: %s\n\n", current->description);
        current = current->next;
    }
}

// Main function to demonstrate the log queue system
int main() {
    LogQueue queue;
    initLogQueue(&queue);

    // Enqueue some log entries
    enqueue(&queue, "ERROR", "Failed to initialize sensor.");
    enqueue(&queue, "INFO", "Sensor initialized successfully.");
    enqueue(&queue, "WARNING", "Sensor temperature is too high.");

    // Print the queue
    printf("Current Log Queue:\n");
    printQueue(&queue);

    // Search for a specific event type
    printf("\nSearching for 'ERROR' events:\n");
    searchLogsByEventType(&queue, "ERROR");

    // Dequeue one log entry
    printf("\nDequeueing one log entry:\n");
    dequeue(&queue);
    printQueue(&queue);

    // Dequeue all logs (simulate memory cleanup)
    printf("\nDequeueing all log entries:\n");
    while (queue.size > 0) {
        dequeue(&queue);
    }
    printQueue(&queue); // Should print nothing as the queue is now empty

    return 0;
}
```

6. Network Packet Management:
Create a queue using a linked list to manage network packets in an embedded router. Each packet should have a source IP, destination IP, and payload. Implement enqueue for incoming packets, dequeue for packets ready for transmission, and search for packets by IP address.

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

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

// Define the Packet structure
typedef struct Packet {
    char source_ip[16];    // Source IP address (IPv4, e.g., "192.168.1.1")
    char dest_ip[16];      // Destination IP address (IPv4, e.g., "192.168.1.2")
    char payload[256];     // Payload data (can vary in size)
    struct Packet* next;   // Pointer to the next packet in the queue
} Packet;

// Define the Queue structure
typedef struct Queue {
    Packet* front;         // Pointer to the front of the queue
    Packet* rear;          // Pointer to the rear of the queue
} Queue;

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

// Function to create a new packet
Packet* createPacket(const char* source_ip, const char* dest_ip, const char* payload) {
    Packet* new_packet = (Packet*)malloc(sizeof(Packet));
    if (new_packet == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }

    // Set the packet details
    strncpy(new_packet->source_ip, source_ip, 16);
    strncpy(new_packet->dest_ip, dest_ip, 16);
    strncpy(new_packet->payload, payload, 256);
    new_packet->next = NULL;

    return new_packet;
}

// Function to enqueue a packet into the queue
void enqueue(Queue* q, const char* source_ip, const char* dest_ip, const char* payload) {
    Packet* new_packet = createPacket(source_ip, dest_ip, payload);
    if (new_packet == NULL) {
        return;
    }

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

// Function to dequeue a packet from the queue
```

```
Packet* dequeue(Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return NULL;
    }

    Packet* temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL; // Queue is now empty
    }

    return temp;
}

// Function to search for a packet by source or destination IP address
Packet* search(Queue* q, const char* ip_address) {
    Packet* current = q->front;
    while (current != NULL) {
        if (strcmp(current->source_ip, ip_address) == 0 || strcmp(current->dest_ip, ip_address) == 0) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}

// Function to print packet details (for debugging)
void printPacket(Packet* p) {
    if (p != NULL) {
        printf("Source IP: %s\n", p->source_ip);
        printf("Destination IP: %s\n", p->dest_ip);
        printf("Payload: %s\n", p->payload);
    } else {
        printf("Packet not found.\n");
    }
}

// Function to print the entire queue (for debugging)
void printQueue(Queue* q) {
    Packet* current = q->front;
    while (current != NULL) {
        printPacket(current);
        current = current->next;
    }
}

// Main function to demonstrate queue operations
int main() {
    Queue q;
    initQueue(&q);

    // Enqueue some packets
    enqueue(&q, "192.168.1.1", "192.168.1.2", "Packet 1 payload");
    enqueue(&q, "192.168.1.3", "192.168.1.4", "Packet 2 payload");
```

```
    enqueue(&q, "192.168.1.5", "192.168.1.6", "Packet 3 payload");

    // Print the queue
    printf("Queue after enqueueing packets:\n");
    printQueue(&q);

    // Search for a packet by IP address
    printf("\nSearching for packet with source IP 192.168.1.3:\n");
    Packet* p = search(&q, "192.168.1.3");
    printPacket(p);

    // Dequeue a packet
    printf("\nDequeuing a packet:\n");
    Packet* dequeued_packet = dequeue(&q);
    printPacket(dequeued_packet);
    free(dequeued_packet);

    // Print the queue after dequeueing
    printf("\nQueue after dequeueing a packet:\n");
    printQueue(&q);

    return 0;
}
```

7. Firmware Update Queue:
Implement a queue using a linked list to manage firmware updates in an embedded system. Each update should include a version number, release notes, and file path. Enqueue updates as they become available, dequeue them for installation, and search for updates by version number

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

// Define the structure for a firmware update
typedef struct FirmwareUpdate {
    int version_number;
    char release_notes[256];
    char file_path[256];
    struct FirmwareUpdate* next;  // Pointer to next firmware update in the queue
} FirmwareUpdate;

// Define the structure for the Queue
typedef struct {
    FirmwareUpdate* front;
    FirmwareUpdate* rear;
} FirmwareQueue;

// Function to initialize the queue
void initQueue(FirmwareQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to create a new firmware update node
FirmwareUpdate* createUpdate(int version, const char* release_notes, const char* file_path) {
```

```
    FirmwareUpdate* newUpdate = (FirmwareUpdate*)malloc(sizeof(FirmwareUpdate));
    newUpdate->version_number = version;
    strncpy(newUpdate->release_notes, release_notes, sizeof(newUpdate->release_notes) - 1);
    strncpy(newUpdate->file_path, file_path, sizeof(newUpdate->file_path) - 1);
    newUpdate->next = NULL;
    return newUpdate;
}

// Function to enqueue a firmware update into the queue
void enqueue(FirmwareQueue* queue, int version, const char* release_notes, const char* file_path) {
    FirmwareUpdate* newUpdate = createUpdate(version, release_notes, file_path);

    if (queue->rear == NULL) {
        queue->front = queue->rear = newUpdate;  // If the queue is empty, the new update is both front and
rear
    } else {
        queue->rear->next = newUpdate;  // Link the current rear node to the new node
        queue->rear = newUpdate;  // Update rear to the new node
    }
}

// Function to dequeue a firmware update from the queue
FirmwareUpdate* dequeue(FirmwareQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty.\n");
        return NULL;
    }

    FirmwareUpdate* updateToInstall = queue->front;  // Get the front update
    queue->front = queue->front->next;  // Move the front pointer to the next node

    if (queue->front == NULL) {
        queue->rear = NULL;  // If the queue is now empty, set the rear to NULL
    }

    return updateToInstall;
}

// Function to search for a firmware update by version number
FirmwareUpdate* searchByVersion(FirmwareQueue* queue, int version) {
    FirmwareUpdate* current = queue->front;
    while (current != NULL) {
        if (current->version_number == version) {
            return current;  // Found the update
        }
        current = current->next;
    }
    return NULL;  // Version not found
}

// Function to print firmware update details
void printUpdateDetails(FirmwareUpdate* update) {
    if (update != NULL) {
        printf("Version: %d\n", update->version_number);
        printf("Release Notes: %s\n", update->release_notes);
```

```
        printf("File Path: %s\n", update->file_path);
    }
}

// Function to free memory for all updates in the queue
void freeQueue(FirmwareQueue* queue) {
    FirmwareUpdate* current = queue->front;
    FirmwareUpdate* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
    queue->front = queue->rear = NULL;
}

int main() {
    FirmwareQueue queue;
    initQueue(&queue);

    // Enqueue some updates
    enqueue(&queue, 1, "Bug fixes and performance improvements", "/firmware/v1.0.bin");
    enqueue(&queue, 2, "New features added", "/firmware/v2.0.bin");
    enqueue(&queue, 3, "Security updates", "/firmware/v3.0.bin");

    // Search for a firmware update by version number
    int searchVersion = 2;
    FirmwareUpdate* update = searchByVersion(&queue, searchVersion);

    if (update != NULL) {
        printf("Found firmware update:\n");
        printUpdateDetails(update);
    } else {
        printf("No update found for version %d.\n", searchVersion);
    }

    // Dequeue and install firmware updates
    FirmwareUpdate* installedUpdate = dequeue(&queue);
    if (installedUpdate != NULL) {
        printf("Installing firmware update:\n");
        printUpdateDetails(installedUpdate);
        free(installedUpdate);  // Free memory after installation
    }

    installedUpdate = dequeue(&queue);
    if (installedUpdate != NULL) {
        printf("Installing firmware update:\n");
        printUpdateDetails(installedUpdate);
        free(installedUpdate);  // Free memory after installation
    }

    // Clean up the remaining queue
    freeQueue(&queue);
```

```
    return 0;
}
```

8. Power Management Events:
Design a queue using a linked list to handle power management events in an embedded device. Each event should have a type (e.g., power on, sleep), timestamp, and associated action. Implement operations to enqueue events, dequeue events as they are handled, and search for events by type.

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

// Define event types
#define POWER_ON 1
#define SLEEP    2
#define WAKEUP   3
#define POWER_OFF 4

// Define the event structure
typedef struct event {
    int event_type;       // Event type
    unsigned long timestamp; // Timestamp of the event
    char action[50];      // Associated action
    struct event *next;   // Pointer to the next event in the queue
} Event;

// Define the queue structure
typedef struct queue {
    Event *front;         // Pointer to the front of the queue
    Event *rear;          // Pointer to the rear of the queue
} Queue;

// Function to create a new event
Event* create_event(int event_type, unsigned long timestamp, const char* action) {
    Event *new_event = (Event *)malloc(sizeof(Event));
    if (new_event == NULL) {
        printf("Error: Memory allocation failed\n");
        return NULL;
    }
    new_event->event_type = event_type;
    new_event->timestamp = timestamp;
    strncpy(new_event->action, action, sizeof(new_event->action) - 1);
    new_event->action[sizeof(new_event->action) - 1] = '\0';
    new_event->next = NULL;
    return new_event;
}

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

// Function to enqueue an event into the queue
void enqueue(Queue *q, Event *new_event) {
```

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

// Function to dequeue an event from the queue
Event* dequeue(Queue *q) {
    if (q->front == NULL) {
        printf("Error: Queue is empty\n");
        return NULL;
    }
    Event *temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    return temp;
}

// Function to search for events by type
Event* search_event_by_type(Queue *q, int event_type) {
    Event *current = q->front;
    while (current != NULL) {
        if (current->event_type == event_type) {
            return current;  // Return the first match
        }
        current = current->next;
    }
    return NULL;  // No event found of the specified type
}

// Function to print an event
void print_event(Event *event) {
    if (event != NULL) {
        printf("Event Type: %d\n", event->event_type);
        printf("Timestamp: %lu\n", event->timestamp);
        printf("Action: %s\n", event->action);
    }
}

// Function to free the memory allocated for an event
void free_event(Event *event) {
    free(event);
}

// Main function demonstrating the queue usage
int main() {
    Queue queue;
    init_queue(&queue);

    // Create and enqueue some events
    Event *event1 = create_event(POWER_ON, 1000, "Power On");
```

```
    enqueue(&queue, event1);

    Event *event2 = create_event(SLEEP, 2000, "Sleep Mode");
    enqueue(&queue, event2);

    Event *event3 = create_event(WAKEUP, 3000, "Wakeup");
    enqueue(&queue, event3);

    Event *event4 = create_event(POWER_OFF, 4000, "Power Off");
    enqueue(&queue, event4);

    // Dequeue and print events as they are handled
    printf("Dequeue and handle events:\n");
    Event *event;
    while ((event = dequeue(&queue)) != NULL) {
        printf("Handling event...\n");
        print_event(event);
        free_event(event);  // Free the event after handling it
    }

    // Search for an event by type
    printf("\nSearching for event type %d (POWER_ON)...\n", POWER_ON);
    Event *found_event = search_event_by_type(&queue, POWER_ON);
    if (found_event != NULL) {
        print_event(found_event);
    } else {
        printf("No event found for the specified type\n");
    }

    return 0;
}
```

9. Command Queue for Embedded Systems:
Create a command queue using a linked list to handle user or system commands. Each command should have an ID, type, and parameters. Implement enqueue for new commands, dequeue for commands ready for execution, and search for commands by type.

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

// Define the maximum length of the command type and parameters
#define MAX_CMD_TYPE_LEN 20
#define MAX_CMD_PARAM_LEN 50

// Command structure
typedef struct Command {
    int id;
    char type[MAX_CMD_TYPE_LEN];
    char parameters[MAX_CMD_PARAM_LEN];
    struct Command* next;  // Pointer to the next command in the queue
} Command;

// Command Queue structure
typedef struct CommandQueue {
```

```
    Command* front;
    Command* rear;
} CommandQueue;

// Function to initialize the command queue
void initQueue(CommandQueue* queue) {
    queue->front = queue->rear = NULL;
}

// Function to create a new command
Command* createCommand(int id, const char* type, const char* parameters) {
    Command* newCommand = (Command*)malloc(sizeof(Command));
    if (!newCommand) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    newCommand->id = id;
    strncpy(newCommand->type, type, MAX_CMD_TYPE_LEN);
    strncpy(newCommand->parameters, parameters, MAX_CMD_PARAM_LEN);
    newCommand->next = NULL;
    return newCommand;
}

// Enqueue a new command into the command queue
void enqueue(CommandQueue* queue, int id, const char* type, const char* parameters) {
    Command* newCommand = createCommand(id, type, parameters);
    if (newCommand == NULL) return;

    if (queue->rear == NULL) {
        // If the queue is empty, the new command becomes both the front and rear
        queue->front = queue->rear = newCommand;
    } else {
        // Add the new command to the rear of the queue
        queue->rear->next = newCommand;
        queue->rear = newCommand;
    }
    printf("Command with ID %d enqueued: Type = %s, Params = %s\n", id, type, parameters);
}

// Dequeue a command from the front of the queue
Command* dequeue(CommandQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty. No command to dequeue.\n");
        return NULL;
    }

    Command* dequeuedCommand = queue->front;
    queue->front = queue->front->next;

    // If the queue is now empty, reset the rear pointer to NULL
    if (queue->front == NULL) {
        queue->rear = NULL;
    }

    return dequeuedCommand;
```

```
}

// Search for commands by type
void searchCommandsByType(CommandQueue* queue, const char* type) {
    Command* current = queue->front;
    int found = 0;

    while (current != NULL) {
        if (strncmp(current->type, type, MAX_CMD_TYPE_LEN) == 0) {
            printf("Found command ID %d with Type = %s, Params = %s\n", current->id, current->type,
current->parameters);
            found = 1;
        }
        current = current->next;
    }

    if (!found) {
        printf("No commands found with Type = %s\n", type);
    }
}

// Function to display all commands in the queue
void displayQueue(CommandQueue* queue) {
    Command* current = queue->front;
    if (current == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Commands in the queue:\n");
    while (current != NULL) {
        printf("ID = %d, Type = %s, Params = %s\n", current->id, current->type, current->parameters);
        current = current->next;
    }
}

int main() {
    // Create and initialize the command queue
    CommandQueue queue;
    initQueue(&queue);

    // Enqueue some commands
    enqueue(&queue, 1, "SET", "Pin 1 High");
    enqueue(&queue, 2, "GET", "Sensor Data");
    enqueue(&queue, 3, "SET", "Pin 2 Low");
    enqueue(&queue, 4, "RESET", "System");

    // Display all commands in the queue
    displayQueue(&queue);

    // Dequeue a command and execute it (for demo purposes)
    Command* cmd = dequeue(&queue);
    if (cmd != NULL) {
        printf("\nDequeued Command ID %d: Type = %s, Params = %s\n", cmd->id, cmd->type,
cmd->parameters);
```

```
        free(cmd);  // Don’t forget to free the memory of dequeued command
    }

    // Search for commands by type
    printf("\nSearching for ’SET’ commands:\n");
    searchCommandsByType(&queue, "SET");

    // Display all remaining commands in the queue
    displayQueue(&queue);

    // Dequeue all remaining commands
    while ((cmd = dequeue(&queue)) != NULL) {
        free(cmd);  // Free the memory of each dequeued command
    }

    return 0;
}
```

10. Audio Buffering in Embedded Audio Systems:
Implement a queue using a linked list to buffer audio samples in an embedded audio system. Each buffer entry should include a timestamp and audio data. Enqueue new audio samples, dequeue samples for playback, and search for samples by timestamp.

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

// Define the structure for an audio sample
typedef struct AudioSample {
    uint32_t timestamp;  // Timestamp for the sample
    int16_t *data;      // Pointer to the audio data (e.g., PCM sample)
    struct AudioSample *next;  // Pointer to the next sample in the queue
} AudioSample;

// Define the queue structure
typedef struct AudioQueue {
    AudioSample *head;  // Pointer to the first element in the queue
    AudioSample *tail;  // Pointer to the last element in the queue
} AudioQueue;

// Function to initialize the audio queue
void initQueue(AudioQueue *queue) {
    queue->head = NULL;
    queue->tail = NULL;
}

// Function to create a new audio sample
AudioSample* createAudioSample(uint32_t timestamp, int16_t *data) {
    AudioSample *newSample = (AudioSample*)malloc(sizeof(AudioSample));
    if (newSample == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    newSample->timestamp = timestamp;
    newSample->data = data;
```

```
    newSample->next = NULL;
    return newSample;
}

// Function to enqueue a new audio sample to the queue
void enqueue(AudioQueue *queue, uint32_t timestamp, int16_t *data) {
    AudioSample *newSample = createAudioSample(timestamp, data);
    if (newSample == NULL) return;

    if (queue->tail == NULL) {
        queue->head = queue->tail = newSample;
    } else {
        queue->tail->next = newSample;
        queue->tail = newSample;
    }
}

// Function to dequeue an audio sample from the queue for playback
AudioSample* dequeue(AudioQueue *queue) {
    if (queue->head == NULL) {
        printf("Queue is empty!\n");
        return NULL;
    }

    AudioSample *sampleToDequeue = queue->head;
    queue->head = queue->head->next;
    if (queue->head == NULL) {
        queue->tail = NULL;  // Queue is now empty
    }

    return sampleToDequeue;
}

// Function to search for an audio sample by timestamp
AudioSample* searchByTimestamp(AudioQueue *queue, uint32_t timestamp) {
    AudioSample *current = queue->head;
    while (current != NULL) {
        if (current->timestamp == timestamp) {
            return current;
        }
        current = current->next;
    }
    return NULL;  // Sample not found
}

// Function to print all samples in the queue (for debugging)
void printQueue(AudioQueue *queue) {
    AudioSample *current = queue->head;
    while (current != NULL) {
        printf("Timestamp: %u, Audio Data: %p\n", current->timestamp, (void*)current->data);
        current = current->next;
    }
}

int main() {
```

```
    AudioQueue queue;
    initQueue(&queue);

    // Enqueue some audio samples
    int16_t sample1[] = {100, 200, 300};
    int16_t sample2[] = {150, 250, 350};
    int16_t sample3[] = {200, 300, 400};

    enqueue(&queue, 1000, sample1);
    enqueue(&queue, 2000, sample2);
    enqueue(&queue, 3000, sample3);

    // Print the queue
    printf("Queue after enqueuing samples:\n");
    printQueue(&queue);

    // Dequeue a sample
    AudioSample *dequeuedSample = dequeue(&queue);
    if (dequeuedSample != NULL) {
        printf("Dequeued sample with timestamp %u\n", dequeuedSample->timestamp);
        free(dequeuedSample);
    }

    // Search for a sample by timestamp
    uint32_t timestampToSearch = 2000;
    AudioSample *foundSample = searchByTimestamp(&queue, timestampToSearch);
    if (foundSample != NULL) {
        printf("Found sample with timestamp %u\n", foundSample->timestamp);
    } else {
        printf("Sample with timestamp %u not found\n", timestampToSearch);
    }

    // Print the queue again
    printf("Queue after dequeuing one sample:\n");
    printQueue(&queue);

    // Clean up remaining samples
    while (queue.head != NULL) {
        AudioSample *sample = dequeue(&queue);
        free(sample);
    }

    return 0;
}
```

11. Event-Driven Programming in Embedded Systems:
Design a queue using a linked list to manage events in an event-driven embedded system. Each event should have an ID, type, and associated data. Implement enqueue for new events, dequeue for event handling, and search for events by type or ID.

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

// Define the event structure
```

```
typedef struct Event {
    int id;             // Event ID
    int type;           // Event type
    void *data;         // Pointer to associated data (can be any type)
    struct Event *next; // Pointer to the next event in the list
} Event;

// Define the event queue structure
typedef struct EventQueue {
    Event *front; // Front of the queue (oldest event)
    Event *rear;  // Rear of the queue (newest event)
} EventQueue;

// Function to initialize the event queue
void initQueue(EventQueue *queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to enqueue an event into the queue
void enqueue(EventQueue *queue, int id, int type, void *data) {
    Event *newEvent = (Event *)malloc(sizeof(Event));
    if (!newEvent) {
        printf("Memory allocation failed!\n");
        return;
    }

    newEvent->id = id;
    newEvent->type = type;
    newEvent->data = data;
    newEvent->next = NULL;

    if (queue->rear == NULL) {
        // If the queue is empty, both front and rear point to the new event
        queue->front = queue->rear = newEvent;
    } else {
        // Otherwise, add it to the end and update the rear pointer
        queue->rear->next = newEvent;
        queue->rear = newEvent;
    }
}

// Function to dequeue an event from the queue
Event *dequeue(EventQueue *queue) {
    if (queue->front == NULL) {
        printf("Queue is empty!\n");
        return NULL;
    }

    Event *eventToDequeue = queue->front;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL; // If the queue becomes empty, set rear to NULL
    }
```

```
    return eventToDequeue;
}

// Function to search for an event by ID
Event *searchEventByID(EventQueue *queue, int id) {
    Event *current = queue->front;
    while (current != NULL) {
        if (current->id == id) {
            return current; // Return the event if found
        }
        current = current->next;
    }
    return NULL; // Return NULL if not found
}

// Function to search for an event by Type
Event *searchEventByType(EventQueue *queue, int type) {
    Event *current = queue->front;
    while (current != NULL) {
        if (current->type == type) {
            return current; // Return the event if found
        }
        current = current->next;
    }
    return NULL; // Return NULL if not found
}

// Example usage of the event queue
int main() {
    EventQueue queue;
    initQueue(&queue);

    int data1 = 100;
    int data2 = 200;

    // Enqueue events
    enqueue(&queue, 1, 10, &data1); // Event with ID 1, Type 10, Data 100
    enqueue(&queue, 2, 20, &data2); // Event with ID 2, Type 20, Data 200

    // Dequeue an event and print details
    Event *event = dequeue(&queue);
    if (event) {
        printf("Dequeued Event: ID = %d, Type = %d, Data = %d\n", event->id, event->type, *(int
*)event->data);
        free(event); // Free the memory after processing
    }

    // Search for an event by ID
    event = searchEventByID(&queue, 2);
    if (event) {
        printf("Found Event by ID: ID = %d, Type = %d, Data = %d\n", event->id, event->type, *(int
*)event->data);
    } else {
        printf("Event with ID 2 not found!\n");
    }
```

```
    // Search for an event by Type
    event = searchEventByType(&queue, 10);
    if (event) {
        printf("Found Event by Type: ID = %d, Type = %d, Data = %d\n", event->id, event->type, *(int
*)event->data);
    } else {
        printf("Event with Type 10 not found!\n");
    }

    return 0;
}
```

12. Embedded GUI Event Queue:
Create a queue using a linked list to manage GUI events (e.g., button clicks, screen touches) in an embedded system. Each event should have an event type, coordinates, and timestamp. Implement enqueue for new GUI events, dequeue for event handling, and search for events by type.

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

// Define the Event Structure
typedef enum {
    BUTTON_CLICK,
    SCREEN_TOUCH,
    MOUSE_MOVE
} EventType;

typedef struct {
    EventType type;       // Type of the event
    int x, y;             // Coordinates (for touch or mouse events)
    time_t timestamp;     // Timestamp when the event occurred
} GUIEvent;

// Define the Node Structure for Linked List Queue
typedef struct Node {
    GUIEvent event;
    struct Node* next;   // Pointer to next node in the queue
} Node;

// Define the Queue Structure
typedef struct {
    Node* front;         // Front of the queue (for dequeuing)
    Node* rear;          // Rear of the queue (for enqueuing)
} EventQueue;

// Function to initialize the queue
void initializeQueue(EventQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Function to create a new event
GUIEvent createEvent(EventType type, int x, int y) {
```

```
    GUIEvent newEvent;
    newEvent.type = type;
    newEvent.x = x;
    newEvent.y = y;
    newEvent.timestamp = time(NULL);  // Capture current time as timestamp
    return newEvent;
}

// Enqueue a new event into the queue
void enqueue(EventQueue* queue, GUIEvent newEvent) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        return;
    }
    newNode->event = newEvent;
    newNode->next = NULL;

    if (queue->rear) {
        queue->rear->next = newNode;
    } else {
        queue->front = newNode;
    }
    queue->rear = newNode;
}

// Dequeue an event from the front of the queue
GUIEvent dequeue(EventQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty\n");
        return (GUIEvent){0};  // Return an empty event
    }
    Node* temp = queue->front;
    GUIEvent event = temp->event;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    free(temp);
    return event;
}

// Search for events by type (returns NULL if no events found)
Node* search_event_by_type(EventQueue* queue, EventType type) {
    Node* current = queue->front;
    while (current != NULL) {
        if (current->event.type == type) {
            return current;  // Return the first event with matching type
        }
        current = current->next;
    }
    return NULL;  // No event found with the given type
}

// Utility function to print event details (for debugging)
```

```
void printEvent(GUIEvent event) {
    char buffer[26];
    struct tm* tm_info;

    tm_info = localtime(&event.timestamp);
    strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", tm_info);

    printf("Event Type: %d, Coordinates: (%d, %d), Timestamp: %s\n", event.type, event.x, event.y,
buffer);
}

int main() {
    // Create and initialize event queue
    EventQueue queue;
    initializeQueue(&queue);

    // Enqueue some events
    enqueue(&queue, createEvent(BUTTON_CLICK, 100, 200));
    enqueue(&queue, createEvent(SCREEN_TOUCH, 150, 250));
    enqueue(&queue, createEvent(MOUSE_MOVE, 200, 300));
    enqueue(&queue, createEvent(BUTTON_CLICK, 250, 350));

    // Dequeue and print events
    GUIEvent event = dequeue(&queue);
    printEvent(event);

    // Search for a BUTTON_CLICK event and print it
    Node* searchResult = search_event_by_type(&queue, BUTTON_CLICK);
    if (searchResult != NULL) {
        printEvent(searchResult->event);
    } else {
        printf("No event found of type BUTTON_CLICK\n");
    }

    // Continue dequeuing the remaining events and print them
    while (queue.front != NULL) {
        event = dequeue(&queue);
        printEvent(event);
    }

    return 0;
}
```

13. Serial Communication Buffer:
Implement a queue using a linked list to buffer data in a serial communication system. Each buffer entry should include data and its length. Enqueue new data chunks, dequeue them for transmission, and search for specific data patterns.

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

// Define the maximum data size for simplicity.
#define MAX_DATA_SIZE 1024
```

```
// Define a node in the linked list (buffer entry).
typedef struct Node {
    char data[MAX_DATA_SIZE]; // Buffer for the data chunk
    size_t length;           // Length of the data chunk
    struct Node* next;       // Pointer to the next node
} Node;

// Define the Queue structure.
typedef struct Queue {
    Node* front; // Pointer to the front of the queue
    Node* rear;  // Pointer to the rear of the queue
} Queue;

// Function to create a new node
Node* createNode(const char* data, size_t length) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    strncpy(newNode->data, data, length);
    newNode->length = length;
    newNode->next = NULL;
    return newNode;
}

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

// Function to enqueue data into the queue
void enqueue(Queue* q, const char* data, size_t length) {
    Node* newNode = createNode(data, length);
    if (!newNode) return;

    if (q->rear == NULL) {
        q->front = q->rear = newNode;  // First element
        return;
    }

    q->rear->next = newNode;  // Add the new node at the end
    q->rear = newNode;        // Update rear pointer
}

// Function to dequeue data from the queue
Node* dequeue(Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty!\n");
        return NULL;
    }

    Node* temp = q->front;
    q->front = q->front->next;
```

```
    if (q->front == NULL) {
        q->rear = NULL;  // Queue is empty, reset rear
    }

    return temp;  // Return the dequeued node
}

// Function to search for a pattern in the queue
Node* searchQueue(Queue* q, const char* pattern) {
    Node* current = q->front;
    while (current != NULL) {
        if (strstr(current->data, pattern) != NULL) {
            return current;  // Pattern found
        }
        current = current->next;
    }
    return NULL;  // Pattern not found
}

// Function to print the contents of the queue (for debugging purposes)
void printQueue(Queue* q) {
    Node* current = q->front;
    while (current != NULL) {
        printf("Data: %s, Length: %zu\n", current->data, current->length);
        current = current->next;
    }
}

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

    // Enqueue some data chunks
    enqueue(&q, "Hello, World!", 13);
    enqueue(&q, "Serial data transmission", 25);
    enqueue(&q, "Buffering data chunks", 22);

    // Print queue contents
    printf("Queue contents before dequeue:\n");
    printQueue(&q);

    // Dequeue an element
    Node* dequeuedNode = dequeue(&q);
    if (dequeuedNode != NULL) {
        printf("\nDequeued Data: %s, Length: %zu\n", dequeuedNode->data, dequeuedNode->length);
        free(dequeuedNode);  // Free memory for dequeued node
    }

    // Search for a specific pattern in the queue
    const char* searchPattern = "Serial";
    Node* foundNode = searchQueue(&q, searchPattern);
    if (foundNode != NULL) {
        printf("\nFound pattern '%s' in data: %s\n", searchPattern, foundNode->data);
    } else {
```

```
        printf("\nPattern '%s' not found in the queue.\n", searchPattern);
    }

    // Print queue contents after dequeue
    printf("\nQueue contents after dequeue:\n");
    printQueue(&q);

    return 0;
}
```

14. CAN Bus Message Queue:
Design a queue using a linked list to manage CAN bus messages in an embedded automotive system. Each message should have an ID, data length, and payload. Implement enqueue for incoming messages, dequeue for processing, and search for messages by ID.

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

#define MAX_PAYLOAD_SIZE 8  // Assuming a maximum of 8 bytes for the payload

// Structure for a CAN bus message
typedef struct CAN_Message {
    unsigned int id;          // CAN message ID
    unsigned char length;     // Data length (in bytes)
    unsigned char payload[MAX_PAYLOAD_SIZE]; // Payload data
    struct CAN_Message* next; // Pointer to next message in the queue
} CAN_Message;

// Structure for the message queue
typedef struct CAN_MessageQueue {
    CAN_Message* front; // Front of the queue (dequeue from here)
    CAN_Message* rear;  // Rear of the queue (enqueue to here)
    unsigned int size;  // Number of messages in the queue
} CAN_MessageQueue;

// Function to initialize the queue
void initQueue(CAN_MessageQueue* queue) {
    queue->front = NULL;
    queue->rear = NULL;
    queue->size = 0;
}

// Function to create a new CAN message
CAN_Message* createMessage(unsigned int id, unsigned char* payload, unsigned char length) {
    CAN_Message* newMessage = (CAN_Message*)malloc(sizeof(CAN_Message));
    if (!newMessage) {
        printf("Memory allocation failed!\n");
        return NULL;
    }

    newMessage->id = id;
    newMessage->length = length;
    memcpy(newMessage->payload, payload, length);
    newMessage->next = NULL;
```

```
    return newMessage;
}

// Enqueue function to add a message to the queue
void enqueue(CAN_MessageQueue* queue, CAN_Message* newMessage) {
    if (queue->rear == NULL) {
        // Queue is empty
        queue->front = queue->rear = newMessage;
    } else {
        // Add the message at the end
        queue->rear->next = newMessage;
        queue->rear = newMessage;
    }
    queue->size++;
}

// Dequeue function to remove and return the front message
CAN_Message* dequeue(CAN_MessageQueue* queue) {
    if (queue->front == NULL) {
        printf("Queue is empty, cannot dequeue!\n");
        return NULL;
    }

    CAN_Message* temp = queue->front;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        // If the queue is now empty
        queue->rear = NULL;
    }
    queue->size--;

    return temp;
}

// Function to search for a message by its ID
CAN_Message* searchByID(CAN_MessageQueue* queue, unsigned int id) {
    CAN_Message* current = queue->front;
    while (current != NULL) {
        if (current->id == id) {
            return current; // Found the message
        }
        current = current->next;
    }
    return NULL; // Message with the given ID not found
}

// Function to display the message (for testing/debugging)
void displayMessage(CAN_Message* message) {
    if (message != NULL) {
        printf("Message ID: %u\n", message->id);
        printf("Data Length: %u\n", message->length);
        printf("Payload: ");
        for (unsigned char i = 0; i < message->length; i++) {
            printf("%02X ", message->payload[i]);
```

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

// Function to free the memory allocated for the message
void freeMessage(CAN_Message* message) {
    if (message != NULL) {
        free(message);
    }
}

int main() {
    // Initialize the queue
    CAN_MessageQueue queue;
    initQueue(&queue);

    // Create some messages
    unsigned char payload1[] = {0x01, 0x02, 0x03};
    CAN_Message* msg1 = createMessage(100, payload1, sizeof(payload1));
    enqueue(&queue, msg1);

    unsigned char payload2[] = {0x04, 0x05, 0x06};
    CAN_Message* msg2 = createMessage(200, payload2, sizeof(payload2));
    enqueue(&queue, msg2);

    unsigned char payload3[] = {0x07, 0x08, 0x09};
    CAN_Message* msg3 = createMessage(300, payload3, sizeof(payload3));
    enqueue(&queue, msg3);

    // Display all messages in the queue
    printf("Queue contents:\n");
    CAN_Message* current = queue.front;
    while (current != NULL) {
        displayMessage(current);
        current = current->next;
    }

    // Search for a message by ID
    unsigned int searchID = 200;
    CAN_Message* foundMessage = searchByID(&queue, searchID);
    if (foundMessage != NULL) {
        printf("Found message with ID %u:\n", searchID);
        displayMessage(foundMessage);
    } else {
        printf("Message with ID %u not found.\n", searchID);
    }

    // Dequeue and process messages
    CAN_Message* dequeuedMessage = dequeue(&queue);
    printf("Dequeued message:\n");
    displayMessage(dequeuedMessage);
    freeMessage(dequeuedMessage);

    // Dequeue another message
```

```
    dequeuedMessage = dequeue(&queue);
    printf("Dequeued message:\n");
    displayMessage(dequeuedMessage);
    freeMessage(dequeuedMessage);

    // Dequeue the last message
    dequeuedMessage = dequeue(&queue);
    printf("Dequeued message:\n");
    displayMessage(dequeuedMessage);
    freeMessage(dequeuedMessage);

    return 0;
}
```

15. Queue Management for Machine Learning Inference:
Create a queue using a linked list to manage input data for machine learning inference in an embedded system. Each entry should contain input features and metadata. Enqueue new data, dequeue it for inference, and search for specific input data by metadata.

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

#define MAX_FEATURE_SIZE 10  // Size of input features
#define MAX_METADATA_SIZE 50 // Size of metadata string

// Define the structure for each node in the linked list
typedef struct QueueNode {
    float features[MAX_FEATURE_SIZE];  // Array to hold the input features
    char metadata[MAX_METADATA_SIZE];  // Metadata (e.g., timestamp, ID, etc.)
    struct QueueNode* next;            // Pointer to the next node
} QueueNode;

// Define the Queue structure
typedef struct Queue {
    QueueNode* front;  // Pointer to the front of the queue
    QueueNode* rear;   // Pointer to the rear of the queue
    int size;          // Size of the queue
} Queue;

// Function to initialize the queue
void initQueue(Queue* q) {
    q->front = q->rear = NULL;
    q->size = 0;
}

// Function to create a new node with input features and metadata
QueueNode* createNode(float* features, const char* metadata) {
    QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
    if (!newNode) {
        printf("Memory allocation failed\n");
        return NULL;
    }

    // Copy the input features
```

```
    memcpy(newNode->features, features, sizeof(float) * MAX_FEATURE_SIZE);

    // Copy the metadata
    strncpy(newNode->metadata, metadata, MAX_METADATA_SIZE - 1);
    newNode->metadata[MAX_METADATA_SIZE - 1] = '\0';  // Ensure null-termination

    newNode->next = NULL;
    return newNode;
}

// Function to enqueue data into the queue
void enqueue(Queue* q, float* features, const char* metadata) {
    QueueNode* newNode = createNode(features, metadata);

    if (!newNode) return;  // If memory allocation failed

    if (q->rear == NULL) {
        q->front = q->rear = newNode;  // If the queue is empty, both front and rear point to the new node
    } else {
        q->rear->next = newNode;  // Add the new node to the end of the queue
        q->rear = newNode;
    }

    q->size++;
    printf("Enqueued: %s\n", metadata);
}

// Function to dequeue data for inference
QueueNode* dequeue(Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty, cannot dequeue\n");
        return NULL;
    }

    QueueNode* tempNode = q->front;
    q->front = q->front->next;

    // If the queue becomes empty, set the rear to NULL as well
    if (q->front == NULL) {
        q->rear = NULL;
    }

    q->size--;
    printf("Dequeued: %s\n", tempNode->metadata);

    return tempNode;  // Return the dequeued node (for inference)
}

// Function to search for input data by metadata
QueueNode* searchByMetadata(Queue* q, const char* metadata) {
    QueueNode* current = q->front;

    while (current != NULL) {
        if (strcmp(current->metadata, metadata) == 0) {
            return current;  // Return the node if metadata matches
```

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

    printf("Metadata not found: %s\n", metadata);
    return NULL;  // Return NULL if metadata is not found
}

// Function to display the queue contents (for debugging purposes)
void displayQueue(Queue* q) {
    QueueNode* current = q->front;

    if (current == NULL) {
        printf("Queue is empty\n");
        return;
    }

    while (current != NULL) {
        printf("Metadata: %s, Features: [", current->metadata);
        for (int i = 0; i < MAX_FEATURE_SIZE; i++) {
            printf("%.2f", current->features[i]);
            if (i < MAX_FEATURE_SIZE - 1) {
                printf(", ");
            }
        }
        printf("]\n");
        current = current->next;
    }
}

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

    float features1[MAX_FEATURE_SIZE] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0};
    float features2[MAX_FEATURE_SIZE] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0};
    float features3[MAX_FEATURE_SIZE] = {2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0};

    enqueue(&q, features1, "Data1");
    enqueue(&q, features2, "Data2");
    enqueue(&q, features3, "Data3");

    displayQueue(&q);

    // Dequeue and perform inference
    QueueNode* dequeuedNode = dequeue(&q);
    if (dequeuedNode != NULL) {
        printf("Performing inference on metadata: %s\n", dequeuedNode->metadata);
    }

    // Search for data by metadata
    QueueNode* foundNode = searchByMetadata(&q, "Data2");
    if (foundNode != NULL) {
        printf("Found data with metadata: %s\n", foundNode->metadata);
    }
```

```
    displayQueue(&q);

    return 0;
}
```