Day 19 programs

Variable, Static, Const, and Switch Case

Question 1: Write a C program that declares a static variable and a const variable within a function. The program should increment the static variable each time the function is called and use a switch case to check the value of the const variable. The function should handle at least three different cases for the const variable and demonstrate the persistence of the static variable across multiple calls.

```
#include <stdio.h>

void demoFunction() {
    static int staticVar = 0;
    const int constVar = 2;

    staticVar++;
    printf("Static Variable: %d\n", staticVar);

    switch (constVar) {
        case 1:
            printf("Const Variable is 1\n");
            break;
        case 2:
            printf("Const Variable is 2\n");
            break;
        case 3:
            printf("Const Variable is 3\n");
            break;
        default:
```

```
            printf("Const Variable is unknown\n");
            break;
    }
}

int main() {
    printf("First call to demoFunction:\n");
    demoFunction();

    printf("\nSecond call to demoFunction:\n");
    demoFunction();

    printf("\nThird call to demoFunction:\n");
    demoFunction();

    return 0;
}
```

Question 2: Create a C program where a static variable is used to keep track of the number of times a function has been called. Implement a switch case to print a different message based on the number of times the function has been invoked (e.g., first call, second call, more than two calls). Ensure that a const variable is used to define a maximum call limit and terminate further calls once the limit is reached.

```
#include <stdio.h>

void trackCalls() {
    static int callCount = 0;
    const int maxCalls = 5;
```

```
    if (callCount >= maxCalls) {
        printf("Maximum call limit (%d) reached.\n", maxCalls);
        return;
    }

    callCount++;
    printf("Call Number: %d - ", callCount);

    switch (callCount) {
        case 1:
            printf("This is the first call.\n");
            break;
        case 2:
            printf("This is the second call.\n");
            break;
        default:
            printf("This is call number %d.\n", callCount);
            break;
    }
}

int main() {
    printf("Function calls tracking program:\n\n");

    for (int i = 0; i < 7; i++) {
        trackCalls();
    }

    return 0;
```

}

Question 3: Develop a C program that utilizes a static array inside a function to store values across multiple calls. Use a const variable to define the size of the array. Implement a switch case to perform different operations on the array elements (e.g., add, subtract, multiply) based on user input. Ensure the array values persist between function calls.

```
#include <stdio.h>

void modifyArray(int operation, int value) {
    const int ARRAY_SIZE = 5;           // Const variable defining the array size
    static int array[5] = {0};          // Static array to persist values
    static int initialized = 0;         // Flag to initialize array only once

    // Initialize array values to 1 on the first call
    if (!initialized) {
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = 1;
        }
        initialized = 1;
    }

    printf("Operation: ");
    switch (operation) {
        case 1: // Add value to each element
            printf("Addition\n");
            for (int i = 0; i < ARRAY_SIZE; i++) {
                array[i] += value;
            }
```

```
            break;
        case 2: // Subtract value from each element
            printf("Subtraction\n");
            for (int i = 0; i < ARRAY_SIZE; i++) {
                array[i] -= value;
            }
            break;
        case 3: // Multiply each element by value
            printf("Multiplication\n");
            for (int i = 0; i < ARRAY_SIZE; i++) {
                array[i] *= value;
            }
            break;
        default:
            printf("Invalid operation. No changes made.\n");
            return;
    }

    // Print the current state of the array
    printf("Current array values: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int choice, value;
```

```
    printf("Array Modification Program:\n");
    printf("1. Add\n2. Subtract\n3. Multiply\n0. Exit\n");

    while (1) {
        printf("\nEnter your choice (0 to exit): ");
        scanf("%d", &choice);

        if (choice == 0) {
            printf("Exiting program.\n");
            break;
        }

        if (choice >= 1 && choice <= 3) {
            printf("Enter a value to apply: ");
            scanf("%d", &value);
            modifyArray(choice, value);
        } else {
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Question 4: Write a program that demonstrates the difference between const and static variables. Use a static variable to count the number of times a specific switch case is executed, and a const variable to define a threshold value for triggering a specific case. The program should execute different actions based on the value of the static counter compared to the const threshold.

```c
#include <stdio.h>

void demonstrateVariables(int input) {
    static int counter = 0;           // Static variable to count specific case executions
    const int threshold = 3;          // Const variable defining a threshold value

    printf("Input: %d\n", input);
    switch (input) {
        case 1:
            counter++;                // Increment counter for case 1
            printf("Case 1 executed. Counter: %d\n", counter);

            if (counter >= threshold) {
                printf("Counter has exceeded the threshold (%d).\n", threshold);
            }
            break;

        case 2:
            printf("Case 2 executed.\n");
            break;

        default:
            printf("Default case executed.\n");
            break;
    }
}

int main() {
    printf("Program to demonstrate const and static variables:\n");
```

```
    // Simulate different inputs
    demonstrateVariables(1);
    demonstrateVariables(2);
    demonstrateVariables(1);
    demonstrateVariables(1);
    demonstrateVariables(3);

    return 0;
}
```

Question 5: Create a C program with a static counter and a const limit. The program should include a switch case to print different messages based on the value of the counter. After every 5 calls, reset the counter using the const limit. The program should also demonstrate the immutability of the const variable by attempting to modify it and showing the compilation error.

```
#include <stdio.h>

void demonstrateStaticAndConst() {
    static int counter = 0;
    const int limit = 5;

    counter++;
    printf("Call Number: %d\n", counter);

    switch (counter) {
        case 1:
            printf("First call.\n");
            break;
```

```
        case 2:
            printf("Second call.\n");
            break;
        case 3:
            printf("Third call.\n");
            break;
        case 4:
            printf("Fourth call.\n");
            break;
        case 5:
            printf("Fifth call. Counter resetting.\n");
            break;
        default:
            printf("Counter exceeded cases.\n");
            break;
    }

    if (counter >= limit) {
        counter = 0;
    }
}

int main() {
    for (int i = 0; i < 12; i++) {
        demonstrateStaticAndConst();
    }
    return 0;
}
```

Looping Statements, Pointers, Const with Pointers, Functions

Question 1: Write a C program that demonstrates the use of both single and double pointers. Implement a function that uses a for loop to initialize an array and a second function that modifies the array elements using a double pointer. Use the const keyword to prevent modification of the array elements in one of the functions.

```
#include <stdio.h>

void initializeArray(const int *arr, int size) {
    printf("Initial array values:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", *(arr + i));
    }
    printf("\n");
}

void modifyArray(int **arr, int size) {
    for (int i = 0; i < size; i++) {
        *(*arr + i) *= 2; // Double each element
    }
}

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    int *ptr = array;

    initializeArray(ptr, 5);

    modifyArray(&ptr, 5);
```

```
    printf("Modified array values:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Question 2: Develop a program that reads a matrix from the user and uses a function to transpose the matrix. The function should use a double pointer to manipulate the matrix. Demonstrate both call by value and call by reference in the program. Use a const pointer to ensure the original matrix is not modified during the transpose operation.

```
#include <stdio.h>

void transposeMatrix(const int **matrix, int **transposed, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            transposed[j][i] = matrix[i][j];
        }
    }
}

int main() {
    int rows, cols;
    printf("Enter rows and columns of the matrix: ");
    scanf("%d %d", &rows, &cols);
```

```
    int matrix[rows][cols], transposed[cols][rows];
    printf("Enter elements of the matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    int *matrixPtr[rows], *transposedPtr[cols];
    for (int i = 0; i < rows; i++) matrixPtr[i] = matrix[i];
    for (int i = 0; i < cols; i++) transposedPtr[i] = transposed[i];

    transposeMatrix((const int **)matrixPtr, transposedPtr, rows, cols);

    printf("Transposed Matrix:\n");
    for (int i = 0; i < cols; i++) {
        for (int j = 0; j < rows; j++) {
            printf("%d ", transposed[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Question 3: Create a C program that uses a single pointer to dynamically allocate memory for an array. Write a function to initialize the array using a while loop, and

another function to print the array. Use a const pointer to ensure the printing function does not modify the array.

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

void initializeArray(int *array, int size) {
    int i = 0;
    while (i < size) {
        array[i] = i + 1;
        i++;
    }
}

void printArray(const int *array, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int size;
    printf("Enter size of the array: ");
    scanf("%d", &size);

    int *array = (int *)malloc(size * sizeof(int));
    if (!array) {
        printf("Memory allocation failed.\n");
        return 1;
```

```
    }

    initializeArray(array, size);
    printf("Array elements: ");
    printArray(array, size);

    free(array);
    return 0;
}
```

Question 4: Write a program that demonstrates the use of double pointers to swap two arrays. Implement functions using both call by value and call by reference. Use a for loop to print the swapped arrays and apply the const keyword appropriately to ensure no modification occurs in certain operations.

```
#include <stdio.h>

void swapArrays(int **a, int **b) {
    int *temp = *a;
    *a = *b;
    *b = temp;
}

void printArray(const int *array, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}
```

```
int main() {
    int array1[] = {1, 2, 3, 4};
    int array2[] = {5, 6, 7, 8};
    int size = 4;

    int *ptr1 = array1, *ptr2 = array2;

    printf("Before swapping:\n");
    printf("Array 1: ");
    printArray(ptr1, size);
    printf("Array 2: ");
    printArray(ptr2, size);

    swapArrays(&ptr1, &ptr2);

    printf("After swapping:\n");
    printf("Array 1: ");
    printArray(ptr1, size);
    printf("Array 2: ");
    printArray(ptr2, size);

    return 0;
}
```

Question 5: Develop a C program that demonstrates the application of const with pointers. Create a function to read a string from the user and another function to count the frequency of each character using a do-while loop. Use a const pointer to ensure the original string is not modified during character frequency calculation.

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

```c
#include <string.h>

void countFrequency(const char *str) {
    int freq[256] = {0};
    int i = 0;

    do {
        freq[(int)str[i]]++;
        i++;
    } while (str[i] != '\0');

    printf("Character frequencies:\n");
    for (int j = 0; j < 256; j++) {
        if (freq[j] > 0) {
            printf("'%c': %d\n", j, freq[j]);
        }
    }
}

int main() {
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';

    countFrequency(str);

    return 0;
}
```

Arrays, Structures, Nested Structures, Unions, Nested Unions, Strings, Typedef

---

Question 1: Write a C program that uses an array of structures to store information about employees. Each structure should contain a nested structure for the address. Use typedef to simplify the structure definitions. The program should allow the user to enter and display employee information.

```
#include <stdio.h>

typedef struct {
    char street[50];
    char city[50];
    char state[50];
    int zip;
} Address;

typedef struct {
    int id;
    char name[50];
    Address address;
} Employee;

void inputEmployee(Employee *emp) {
    printf("Enter Employee ID: ");
    scanf("%d", &emp->id);
    printf("Enter Name: ");
    scanf("%s", emp->name);
    printf("Enter Street: ");
    scanf("%s", emp->address.street);
```

```
    printf("Enter City: ");
    scanf("%s", emp->address.city);
    printf("Enter State: ");
    scanf("%s", emp->address.state);
    printf("Enter ZIP Code: ");
    scanf("%d", &emp->address.zip);
}

void displayEmployee(const Employee *emp) {
    printf("Employee ID: %d\n", emp->id);
    printf("Name: %s\n", emp->name);
    printf("Address: %s, %s, %s - %d\n", emp->address.street, emp->address.city,
emp->address.state, emp->address.zip);
}

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

    Employee employees[n];

    for (int i = 0; i < n; i++) {
        printf("\nEntering details for Employee %d:\n", i + 1);
        inputEmployee(&employees[i]);
    }

    printf("\nEmployee Details:\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("\nDetails of Employee %d:\n", i + 1);
        displayEmployee(&employees[i]);
    }

    return 0;
}
```

Question 2: Create a program that demonstrates the use of a union to store different types of data. Implement a nested union within a structure and use a typedef to define the structure. Use an array of this structure to store and display information about different data types (e.g., integer, float, string).

```
#include <stdio.h>

typedef union {
    int intValue;
    float floatValue;
    char stringValue[50];
} Data;

typedef struct {
    char type;
    Data data;
} Element;

void inputElement(Element *elem) {
    printf("Enter type (i for int, f for float, s for string): ");
    scanf(" %c", &elem->type);
    if (elem->type == 'i') {
        printf("Enter integer value: ");
```

```
        scanf("%d", &elem->data.intValue);
    } else if (elem->type == 'f') {
        printf("Enter float value: ");
        scanf("%f", &elem->data.floatValue);
    } else if (elem->type == 's') {
        printf("Enter string value: ");
        scanf("%s", elem->data.stringValue);
    }
}

void displayElement(const Element *elem) {
    if (elem->type == 'i') {
        printf("Integer: %d\n", elem->data.intValue);
    } else if (elem->type == 'f') {
        printf("Float: %.2f\n", elem->data.floatValue);
    } else if (elem->type == 's') {
        printf("String: %s\n", elem->data.stringValue);
    }
}

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

    Element elements[n];

    for (int i = 0; i < n; i++) {
        printf("\nEntering details for Element %d:\n", i + 1);
```

```
        inputElement(&elements[i]);
    }

    printf("\nElement Details:\n");
    for (int i = 0; i < n; i++) {
        printf("\nDetails of Element %d:\n", i + 1);
        displayElement(&elements[i]);
    }

    return 0;
}
```

Question 3: Write a C program that uses an array of strings to store names. Implement a structure containing a nested union to store either the length of the string or the reversed string. Use typedef to simplify the structure definition and display the stored information.

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

typedef union {
    int length;
    char reversed[50];
} StringInfo;

typedef struct {
    char name[50];
    StringInfo info;
} StringData;
```

```
void inputString(StringData *strData) {
    printf("Enter a name: ");
    scanf("%s", strData->name);
    strData->info.length = strlen(strData->name);
}

void reverseString(char *dest, const char *src) {
    int len = strlen(src);
    for (int i = 0; i < len; i++) {
        dest[i] = src[len - i - 1];
    }
    dest[len] = '\0';
}

void displayString(const StringData *strData) {
    reverseString(strData->info.reversed, strData->name);
    printf("Name: %s\n", strData->name);
    printf("Length: %d\n", strData->info.length);
    printf("Reversed: %s\n", strData->info.reversed);
}

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

    StringData names[n];

    for (int i = 0; i < n; i++) {
```

```
        printf("\nEntering details for Name %d:\n", i + 1);
        inputString(&names[i]);
    }

    printf("\nString Details:\n");
    for (int i = 0; i < n; i++) {
        printf("\nDetails of Name %d:\n", i + 1);
        displayString(&names[i]);
    }

    return 0;
}
```

Question 4: Develop a program that demonstrates the use of nested structures and unions. Create a structure that contains a union, and within the union, define another structure. Use an array to manage multiple instances of this complex structure and typedef to define the structure.

```
#include <stdio.h>

typedef struct {
    char title[50];
    int id;
} Book;

typedef union {
    Book bookInfo;
    int pageCount;
} LibraryItem;
```

```
typedef struct {
    char type;
    LibraryItem item;
} Library;

void inputLibrary(Library *lib) {
    printf("Enter type (b for book, p for pages): ");
    scanf(" %c", &lib->type);
    if (lib->type == 'b') {
        printf("Enter book title: ");
        scanf("%s", lib->item.bookInfo.title);
        printf("Enter book ID: ");
        scanf("%d", &lib->item.bookInfo.id);
    } else if (lib->type == 'p') {
        printf("Enter page count: ");
        scanf("%d", &lib->item.pageCount);
    }
}

void displayLibrary(const Library *lib) {
    if (lib->type == 'b') {
        printf("Book Title: %s, ID: %d\n", lib->item.bookInfo.title, lib->item.bookInfo.id);
    } else if (lib->type == 'p') {
        printf("Page Count: %d\n", lib->item.pageCount);
    }
}

int main() {
    int n;
```

```
    printf("Enter the number of library items: ");
    scanf("%d", &n);

    Library library[n];

    for (int i = 0; i < n; i++) {
        printf("\nEntering details for Library Item %d:\n", i + 1);
        inputLibrary(&library[i]);
    }

    printf("\nLibrary Item Details:\n");
    for (int i = 0; i < n; i++) {
        printf("\nDetails of Library Item %d:\n", i + 1);
        displayLibrary(&library[i]);
    }

    return 0;
}
```

Question 5: Write a C program that defines a structure to store information about books. Use a nested structure to store the author's details and a union to store either the number of pages or the publication year. Use typedef to simplify the structure and implement functions to input and display the information.

```
#include <stdio.h>

typedef struct {
    char name[50];
    char email[50];
} Author;
```

```
typedef union {
    int pages;
    int year;
} BookInfo;

typedef struct {
    char title[50];
    Author author;
    BookInfo info;
} Book;

void inputBook(Book *book) {
    printf("Enter book title: ");
    scanf("%s", book->title);
    printf("Enter author name: ");
    scanf("%s", book->author.name);
    printf("Enter author email: ");
    scanf("%s", book->author.email);
    printf("Enter number of pages: ");
    scanf("%d", &book->info.pages);
    printf("Enter publication year: ");
    scanf("%d", &book->info.year);
}

void displayBook(const Book *book) {
    printf("Title: %s\n", book->title);
    printf("Author: %s, Email: %s\n", book->author.name, book->author.email);
    printf("Pages: %d, Year: %d\n", book->info.pages, book->info.year);
```

```
}

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

    Book books[n];

    for (int i = 0; i < n; i++) {
        printf("\nEntering details for Book %d:\n", i + 1);
        inputBook(&books[i]);
    }

    printf("\nBook Details:\n");
    for (int i = 0; i < n; i++) {
        printf("\nDetails of Book %d:\n", i + 1);
        displayBook(&books[i]);
    }

    return 0;
}
```

Stacks Using Arrays and Linked List

Question 1: Write a C program to implement a stack using arrays. The program should include functions for all stack operations: push, pop, peek, isEmpty, and isFull. Demonstrate the working of the stack with sample data.

```c
#include <stdio.h>
#define MAX 5

typedef struct {
    int arr[MAX];
    int top;
} Stack;

void initStack(Stack *s) {
    s->top = -1;
}

int isFull(Stack *s) {
    return s->top == MAX - 1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow!\n");
    } else {
        s->arr[++(s->top)] = value;
    }
}

int pop(Stack *s) {
```

```
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    } else {
        return s->arr[(s->top)--];
    }
}

int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is Empty!\n");
        return -1;
    } else {
        return s->arr[s->top];
    }
}

void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is Empty!\n");
    } else {
        for (int i = 0; i <= s->top; i++) {
            printf("%d ", s->arr[i]);
        }
        printf("\n");
    }
}

int main() {
```

```
    Stack s;
    initStack(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    printf("Stack elements: ");
    display(&s);

    printf("Top element: %d\n", peek(&s));

    pop(&s);
    printf("Stack after pop: ");
    display(&s);

    return 0;
}
```

Question 2: Develop a program to implement a stack using a linked list. Include functions for all stack operations: push, pop, peek, isEmpty, and isFull. Ensure proper memory management by handling dynamic allocation and deallocation.

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

typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

```
typedef struct {
    Node* top;
} Stack;

void initStack(Stack *s) {
    s->top = NULL;
}

int isEmpty(Stack *s) {
    return s->top == NULL;
}

void push(Stack *s, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
    newNode->data = value;
    newNode->next = s->top;
    s->top = newNode;
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    } else {
```

```
        Node* temp = s->top;
        int value = temp->data;
        s->top = s->top->next;
        free(temp);
        return value;
    }
}

int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is Empty!\n");
        return -1;
    } else {
        return s->top->data;
    }
}

void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is Empty!\n");
    } else {
        Node* temp = s->top;
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

```
int main() {
    Stack s;
    initStack(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    printf("Stack elements: ");
    display(&s);

    printf("Top element: %d\n", peek(&s));

    pop(&s);
    printf("Stack after pop: ");
    display(&s);

    return 0;
}
```

Question 3: Create a C program to implement a stack using arrays. Include an additional operation to reverse the contents of the stack. Demonstrate the reversal operation with sample data.

```
#include <stdio.h>
#define MAX 5

typedef struct {
    int arr[MAX];
```

```
    int top;
} Stack;

void initStack(Stack *s) {
    s->top = -1;
}

int isFull(Stack *s) {
    return s->top == MAX - 1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow!\n");
    } else {
        s->arr[++(s->top)] = value;
    }
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    } else {
        return s->arr[(s->top)--];
```

```
    }
}

void reverse(Stack *s) {
    int start = 0;
    int end = s->top;
    while (start < end) {
        int temp = s->arr[start];
        s->arr[start] = s->arr[end];
        s->arr[end] = temp;
        start++;
        end--;
    }
}

void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is Empty!\n");
    } else {
        for (int i = 0; i <= s->top; i++) {
            printf("%d ", s->arr[i]);
        }
        printf("\n");
    }
}

int main() {
    Stack s;
    initStack(&s);
```

```
    push(&s, 10);
    push(&s, 20);
    push(&s, 30);

    printf("Stack before reversal: ");
    display(&s);

    reverse(&s);

    printf("Stack after reversal: ");
    display(&s);

    return 0;
}
```

Question 4: Write a program to implement a stack using a linked list. Extend the program to include an operation to merge two stacks. Demonstrate the merging operation by combining two stacks and displaying the resulting stack.

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

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct {
    Node* top;
```

```
} Stack;

void initStack(Stack *s) {
    s->top = NULL;
}

int isEmpty(Stack *s) {
    return s->top == NULL;
}

void push(Stack *s, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
    newNode->data = value;
    newNode->next = s->top;
    s->top = newNode;
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    } else {
        Node* temp = s->top;
        int value = temp->data;
        s->top = s->top->next;
```

```
        free(temp);
        return value;
    }
}

void mergeStacks(Stack *s1, Stack *s2) {
    while (!isEmpty(s2)) {
        push(s1, pop(s2));
    }
}

void display(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is Empty!\n");
    } else {
        Node* temp = s->top;
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    Stack s1, s2;
    initStack(&s1);
    initStack(&s2);
```

```
    push(&s1, 10);
    push(&s1, 20);
    push(&s2, 30);
    push(&s2, 40);

    printf("Stack 1 before merge: ");
    display(&s1);

    printf("Stack 2 before merge: ");
    display(&s2);

    mergeStacks(&s1, &s2);

    printf("Stack 1 after merge: ");
    display(&s1);

    return 0;
}
```

Question 5: Develop a program that implements a stack using arrays. Add functionality to check for balanced parentheses in an expression using the stack. Demonstrate this with sample expressions.

```
#include <stdio.h>
#define MAX 50

typedef struct {
    char arr[MAX];
    int top;
} Stack;
```

```
void initStack(Stack *s) {
    s->top = -1;
}

int isFull(Stack *s) {
    return s->top == MAX - 1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, char value) {
    if (isFull(s)) {
        printf("Stack Overflow!\n");
    } else {
        s->arr[++(s->top)] = value;
    }
}

char pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    } else {
        return s->arr[(s->top)--];
    }
}
```

```
int isBalanced(char *expr) {
    Stack s;
    initStack(&s);

    for (int i = 0; expr[i] != '\0'; i++) {
        if (expr[i] == '(') {
            push(&s, '(');
        } else if (expr[i] == ')') {
            if (isEmpty(&s)) {
                return 0;  // Unbalanced
            }
            pop(&s);
        }
    }
    return isEmpty(&s);
}

int main() {
    char expr[] = "(a + b) * (c + d)";

    if (isBalanced(expr)) {
        printf("The parentheses are balanced.\n");
    } else {
        printf("The parentheses are unbalanced.\n");
    }

    return 0;
}
```

Question 6: Create a C program to implement a stack using a linked list. Extend the program to implement a stack-based evaluation of postfix expressions. Include all necessary stack operations and demonstrate the evaluation with sample expressions.

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

typedef struct {
    int arr[50];
    int top;
} Stack;

void initStack(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, int value) {
    s->arr[++(s->top)] = value;
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
```

```
        return -1;
    } else {
        return s->arr[(s->top)--];
    }
}

int evaluatePostfix(char *expr) {
    Stack s;
    initStack(&s);

    for (int i = 0; expr[i] != '\0'; i++) {
        if (isdigit(expr[i])) {
            push(&s, expr[i] - '0');
        } else {
            int val2 = pop(&s);
            int val1 = pop(&s);
            switch (expr[i]) {
                case '+': push(&s, val1 + val2); break;
                case '-': push(&s, val1 - val2); break;
                case '*': push(&s, val1 * val2); break;
                case '/': push(&s, val1 / val2); break;
            }
        }
    }
    return pop(&s);
}

int main() {
    char expr[] = "23*45*+";
```

```
    printf("Result of Postfix Evaluation: %d\n", evaluatePostfix(expr));
    return 0;
}
```

Queue using array Programs

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

#define SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50];
};

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

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

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

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

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    q->front++;
    printf("Admitted Student: %s\n", q->students[q->front]);
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Current Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
```

```
        printf("%s ", q->students[i]);
    }
    printf("\n");
}

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

    enqueue(&q, "Alice");
    enqueue(&q, "Bob");
    enqueue(&q, "Charlie");

    display(&q);

    dequeue(&q);
    display(&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>
#include <string.h>
```

```
#define SIZE 10

struct Queue {
    int front, rear;
    char borrowers[SIZE][50]; // Array to store student names
};

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

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

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

void enqueue(struct Queue *q, char name[]) {
    if (isFull(q)) {
        printf("Queue is Full\n");
        return;
    }
    q->rear++;
    strcpy(q->borrowers[q->rear], name); // Add student to the queue
    printf("Added to queue: %s\n", name);
}
```

```
void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    q->front++;
    printf("Borrowing Book: %s\n", q->borrowers[q->front]); // Remove student from the queue
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Current Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->borrowers[i]);
    }
    printf("\n");
}

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

    // Adding students to the queue
    enqueue(&q, "Alice");
    enqueue(&q, "Bob");
```

```
    enqueue(&q, "Charlie");

    // Displaying the queue
    display(&q);

    // Processing students for borrowing books
    dequeue(&q);
    dequeue(&q);

    // Displaying the queue after serving students
    display(&q);

    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 SIZE 10

struct Queue {
    int front, rear;
    int tokens[SIZE];
};
```

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

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

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

void enqueue(struct Queue *q, int token) {
    if (isFull(q)) {
        printf("Queue is Full\n");
        return;
    }
    q->rear++;
    q->tokens[q->rear] = token;
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    q->front++;
    printf("Served Token: %d\n", q->tokens[q->front]);
}
```

```
void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Current Tokens: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%d ", q->tokens[i]);
    }
    printf("\n");
}

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

    enqueue(&q, 101);
    enqueue(&q, 102);
    enqueue(&q, 103);

    display(&q);

    dequeue(&q);
    display(&q);

    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 SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50]; // Array to store student names
};

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

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

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

void enqueue(struct Queue *q, char name[]) {
    if (isFull(q)) {
        printf("Help Desk Queue is Full\n");
```

```
        return;
    }
    q->rear++;
    strcpy(q->students[q->rear], name); // Add student to the queue
    printf("Added to help desk queue: %s\n", name);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Help Desk Queue is Empty\n");
        return;
    }
    q->front++;
    printf("Helping student: %s\n", q->students[q->front]); // Remove student from the
queue
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Help Desk Queue is Empty\n");
        return;
    }
    printf("Current Help Desk Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->students[i]);
    }
    printf("\n");
}
```

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

    // Adding students to the help desk queue
    enqueue(&q, "Alice");
    enqueue(&q, "Bob");
    enqueue(&q, "Charlie");

    // Displaying the current queue
    display(&q);

    // Helping students
    dequeue(&q);
    dequeue(&q);

    // Displaying the queue after helping students
    display(&q);

    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>
#include <string.h>
```

```
#define SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50]; // Array to store student names
};

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

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

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

void enqueue(struct Queue *q, char name[]) {
    if (isFull(q)) {
        printf("Registration Queue is Full\n");
        return;
    }
    q->rear++;
    strcpy(q->students[q->rear], name); // Add student to the queue
    printf("Added to registration queue: %s\n", name);
}
```

```
void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Registration Queue is Empty\n");
        return;
    }
    q->front++;
    printf("Processing registration for: %s\n", q->students[q->front]); // Process student registration
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Registration Queue is Empty\n");
        return;
    }
    printf("Current Registration Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->students[i]);
    }
    printf("\n");
}

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

    // Adding students to the registration queue
    enqueue(&q, "Alice");
```

```
    enqueue(&q, "Bob");
    enqueue(&q, "Charlie");

    // Displaying the queue
    display(&q);

    // Processing registrations
    dequeue(&q);
    dequeue(&q);

    // Displaying the queue after processing some students
    display(&q);

    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 <string.h>
#define SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50];
};
```

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

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

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

void enqueue(struct Queue *q, char name[]) {
    if (isFull(q)) {
        printf("Queue is full, no more students can board.\n");
        return;
    }
    q->rear++;
    strcpy(q->students[q->rear], name);
    printf("%s added to the bus boarding queue.\n", name);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No students waiting to board.\n");
        return;
    }
    q->front++;
    printf("%s boarded the bus.\n", q->students[q->front]);
```

```
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("The bus boarding queue is empty.\n");
        return;
    }
    printf("Bus Boarding Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->students[i]);
    }
    printf("\n");
}

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

    enqueue(&q, "Alice");
    enqueue(&q, "Bob");
    enqueue(&q, "Charlie");
    display(&q);

    dequeue(&q);
    dequeue(&q);
    display(&q);

    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 <string.h>
#define SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50];
};

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

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

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

void enqueue(struct Queue *q, char name[]) {
    if (isFull(q)) {
        printf("Queue is full, cannot add more students.\n");
```

```
        return;
    }
    q->rear++;
    strcpy(q->students[q->rear], name);
    printf("%s added to counseling queue.\n", name);
}

void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No students waiting for counseling.\n");
        return;
    }
    q->front++;
    printf("Counseling session served for: %s\n", q->students[q->front]);
}

void display(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Counseling queue is empty.\n");
        return;
    }
    printf("Counseling Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->students[i]);
    }
    printf("\n");
}

int main() {
```

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

    enqueue(&q, "Eve");
    enqueue(&q, "Dan");
    display(&q);

    dequeue(&q);
    display(&q);

    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 <string.h>
#define SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50];
};

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

```
void enqueue(struct Queue *q, char name[]) {
    if (q->rear == SIZE - 1) {
        printf("Registration queue is full.\n");
        return;
    }
    q->rear++;
    strcpy(q->students[q->rear], name);
    printf("%s registered for the event.\n", name);
}

void dequeue(struct Queue *q) {
    if (q->front == q->rear) {
        printf("No students in the queue.\n");
        return;
    }
    q->front++;
    printf("%s has completed registration.\n", q->students[q->front]);
}

void display(struct Queue *q) {
    if (q->front == q->rear) {
        printf("No students in the registration queue.\n");
        return;
    }
    printf("Registration Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->students[i]);
    }
```

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

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

    enqueue(&q, "John");
    enqueue(&q, "Mike");
    display(&q);

    dequeue(&q);
    display(&q);

    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 <string.h>
#define SIZE 10

struct Queue {
    int front, rear;
    char students[SIZE][50];
};
```

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

void enqueue(struct Queue *q, char name[]) {
    if (q->rear == SIZE - 1) {
        printf("Equipment checkout queue is full.\n");
        return;
    }
    q->rear++;
    strcpy(q->students[q->rear], name);
    printf("%s added to the checkout queue.\n", name);
}

void dequeue(struct Queue *q) {
    if (q->front == q->rear) {
        printf("No students waiting for equipment.\n");
        return;
    }
    q->front++;
    printf("%s received the equipment.\n", q->students[q->front]);
}

void display(struct Queue *q) {
    if (q->front == q->rear) {
        printf("No students in the checkout queue.\n");
        return;
    }
```

```
    printf("Equipment Checkout Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->students[i]);
    }
    printf("\n");
}

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

    enqueue(&q, "Sara");
    enqueue(&q, "Tom");
    display(&q);

    dequeue(&q);
    display(&q);

    return 0;
}
```

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 <string.h>
#define SIZE 10
```

```
struct Queue {
    int front, rear;
    char parents[SIZE][50];
};

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

void enqueue(struct Queue *q, char name[]) {
    if (q->rear == SIZE - 1) {
        printf("Meeting queue is full.\n");
        return;
    }
    q->rear++;
    strcpy(q->parents[q->rear], name);
    printf("%s added to the parent-teacher meeting queue.\n", name);
}

void dequeue(struct Queue *q) {
    if (q->front == q->rear) {
        printf("No parents waiting for a meeting.\n");
        return;
    }
    q->front++;
    printf("Meeting completed for: %s\n", q->parents[q->front]);
}

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

```
    if (q->front == q->rear) {
        printf("No parents in the meeting queue.\n");
        return;
    }
    printf("Meeting Queue: ");
    for (int i = q->front + 1; i <= q->rear; i++) {
        printf("%s ", q->parents[i]);
    }
    printf("\n");
}

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

    enqueue(&q, "Mr. Smith");
    enqueue(&q, "Mrs. Johnson");
    display(&q);

    dequeue(&q);
    display(&q);

    return 0;
}
```

Queue using limked list

---

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>

struct Node {
    int timestamp;
    float data;
    struct Node *next;
};

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

void enqueue(struct Queue *q, int timestamp, float data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->timestamp = timestamp;
    newNode->data = data;
    newNode->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}
```

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

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Node *temp = q->front;
    while (temp) {
        printf("Timestamp: %d, Data: %.2f\n", temp->timestamp, temp->data);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, 25.5);
    enqueue(&q, 2, 26.0);
    enqueue(&q, 3, 27.2);
    display(&q);
```

```
    dequeue(&q);
    display(&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>

struct Task {
    int taskId;
    int priority;
    int executionTime;
    struct Task *next;
};

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

void enqueue(struct Queue *q, int taskId, int priority, int executionTime) {
    struct Task *newTask = (struct Task *)malloc(sizeof(struct Task));
    newTask->taskId = taskId;
    newTask->priority = priority;
```

```
    newTask->executionTime = executionTime;
    newTask->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newTask;
        return;
    }
    q->rear->next = newTask;
    q->rear = newTask;
}

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

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Task *temp = q->front;
    while (temp) {
        printf("Task ID: %d, Priority: %d, Execution Time: %d\n", temp->taskId, temp-
>priority, temp->executionTime);
```

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

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, 5, 10);
    enqueue(&q, 2, 3, 8);
    enqueue(&q, 3, 4, 15);
    display(&q);
    dequeue(&q);
    display(&q);
    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>

struct Interrupt {
    int priority;
    char handler[50];
    struct Interrupt *next;
};
```

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

void enqueue(struct Queue *q, int priority, char *handler) {
    struct Interrupt *newInterrupt = (struct Interrupt *)malloc(sizeof(struct Interrupt));
    newInterrupt->priority = priority;
    strcpy(newInterrupt->handler, handler);
    newInterrupt->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newInterrupt;
        return;
    }
    q->rear->next = newInterrupt;
    q->rear = newInterrupt;
}

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

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

```
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Interrupt *temp = q->front;
    while (temp) {
        printf("Priority: %d, Handler: %s\n", temp->priority, temp->handler);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, "Handler A");
    enqueue(&q, 2, "Handler B");
    enqueue(&q, 3, "Handler C");
    display(&q);
    dequeue(&q);
    display(&q);
    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>
```

```
struct Message {
    int senderId;
    int receiverId;
    char payload[100];
    struct Message *next;
};

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

void enqueue(struct Queue *q, int senderId, int receiverId, char *payload) {
    struct Message *newMessage = (struct Message *)malloc(sizeof(struct Message));
    newMessage->senderId = senderId;
    newMessage->receiverId = receiverId;
    strcpy(newMessage->payload, payload);
    newMessage->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newMessage;
        return;
    }
    q->rear->next = newMessage;
    q->rear = newMessage;
}

void dequeue(struct Queue *q) {
    if (q->front == NULL) {
```

```
        printf("Queue is empty\n");
        return;
    }
    struct Message *temp = q->front;
    q->front = q->front->next;
    free(temp);
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Message *temp = q->front;
    while (temp) {
        printf("Sender ID: %d, Receiver ID: %d, Payload: %s\n", temp->senderId, temp->receiverId, temp->payload);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, 2, "Hello, World!");
    enqueue(&q, 2, 3, "Goodbye, World!");
    display(&q);
    dequeue(&q);
    display(&q);
    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>

struct LogEntry {
    char timestamp[50];
    char eventType[50];
    char description[100];
    struct LogEntry *next;
};

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

void enqueue(struct Queue *q, char *timestamp, char *eventType, char *description)
{
    struct LogEntry *newLog = (struct LogEntry *)malloc(sizeof(struct LogEntry));
    strcpy(newLog->timestamp, timestamp);
    strcpy(newLog->eventType, eventType);
    strcpy(newLog->description, description);
```

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

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

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct LogEntry *temp = q->front;
    while (temp) {
        printf("Timestamp: %s, Event Type: %s, Description: %s\n", temp->timestamp,
temp->eventType, temp->description);
        temp = temp->next;
```

```
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, "2022-03-01 12:00:00", "Error", "Device Overheated");
    enqueue(&q, "2022-03-01 12:10:00", "Warning", "Low Battery");
    display(&q);
    dequeue(&q);
    display(&q);
    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>
#include <stdlib.h>
#include <string.h>

struct Packet {
    char sourceIP[16];
    char destIP[16];
    char payload[100];
    struct Packet *next;
};
```

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

void enqueue(struct Queue *q, char *sourceIP, char *destIP, char *payload) {
    struct Packet *newPacket = (struct Packet *)malloc(sizeof(struct Packet));
    strcpy(newPacket->sourceIP, sourceIP);
    strcpy(newPacket->destIP, destIP);
    strcpy(newPacket->payload, payload);
    newPacket->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newPacket;
        return;
    }
    q->rear->next = newPacket;
    q->rear = newPacket;
}

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

```
void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Packet *temp = q->front;
    while (temp) {
        printf("Source IP: %s, Destination IP: %s, Payload: %s\n", temp->sourceIP,
temp->destIP, temp->payload);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, "192.168.1.1", "192.168.1.2", "Data Packet 1");
    enqueue(&q, "192.168.1.2", "192.168.1.3", "Data Packet 2");
    display(&q);
    dequeue(&q);
    display(&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>

struct FirmwareUpdate {
    char version[10];
    char releaseNotes[100];
    char filePath[100];
    struct FirmwareUpdate *next;
};

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

void enqueue(struct Queue *q, char *version, char *releaseNotes, char *filePath) {
    struct FirmwareUpdate *newUpdate = (struct FirmwareUpdate
*)malloc(sizeof(struct FirmwareUpdate));
    strcpy(newUpdate->version, version);
    strcpy(newUpdate->releaseNotes, releaseNotes);
    strcpy(newUpdate->filePath, filePath);
    newUpdate->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newUpdate;
        return;
    }
    q->rear->next = newUpdate;
    q->rear = newUpdate;
}
```

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

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct FirmwareUpdate *temp = q->front;
    while (temp) {
        printf("Version: %s, Release Notes: %s, File Path: %s\n", temp->version, temp->releaseNotes, temp->filePath);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, "1.0", "Initial Release", "/path/to/update1");
    enqueue(&q, "1.1", "Bug Fixes", "/path/to/update2");
    display(&q);
```

```
    dequeue(&q);
    display(&q);
    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>

struct PowerEvent {
    char eventType[50];
    char timestamp[50];
    char action[100];
    struct PowerEvent *next;
};

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

void enqueue(struct Queue *q, char *eventType, char *timestamp, char *action) {
    struct PowerEvent *newEvent = (struct PowerEvent *)malloc(sizeof(struct
PowerEvent));
```

```
    strcpy(newEvent->eventType, eventType);
    strcpy(newEvent->timestamp, timestamp);
    strcpy(newEvent->action, action);
    newEvent->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newEvent;
        return;
    }
    q->rear->next = newEvent;
    q->rear = newEvent;
}

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

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct PowerEvent *temp = q->front;
    while (temp) {
```

```
        printf("Event Type: %s, Timestamp: %s, Action: %s\n", temp->eventType, temp->timestamp, temp->action);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, "Power On", "2022-01-01 08:00:00", "Start Device");
    enqueue(&q, "Sleep", "2022-01-01 12:00:00", "Sleep Mode");
    display(&q);
    dequeue(&q);
    display(&q);
    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>

struct Command {
    int commandId;
    char commandType[50];
    char parameters[100];
```

```
    struct Command *next;
};

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

void enqueue(struct Queue *q, int commandId, char *commandType, char
*parameters) {
    struct Command *newCommand = (struct Command *)malloc(sizeof(struct
Command));
    newCommand->commandId = commandId;
    strcpy(newCommand->commandType, commandType);
    strcpy(newCommand->parameters, parameters);
    newCommand->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newCommand;
        return;
    }
    q->rear->next = newCommand;
    q->rear = newCommand;
}

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

```
    q->front = q->front->next;
    free(temp);
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Command *temp = q->front;
    while (temp) {
        printf("Command ID: %d, Command Type: %s, Parameters: %s\n", temp->commandId, temp->commandType, temp->parameters);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, "Power On", "None");
    enqueue(&q, 2, "Reboot", "None");
    display(&q);
    dequeue(&q);
    display(&q);
    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>

struct AudioSample {
    int timestamp;
    char data[100];
    struct AudioSample *next;
};

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

void enqueue(struct Queue *q, int timestamp, char *data) {
    struct AudioSample *newSample = (struct AudioSample *)malloc(sizeof(struct AudioSample));
    newSample->timestamp = timestamp;
    strcpy(newSample->data, data);
    newSample->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newSample;
        return;
    }
    q->rear->next = newSample;
    q->rear = newSample;
```

```
}

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

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct AudioSample *temp = q->front;
    while (temp) {
        printf("Timestamp: %d, Data: %s\n", temp->timestamp, temp->data);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, "Sample 1");
    enqueue(&q, 2, "Sample 2");
    display(&q);
```

```
    dequeue(&q);
    display(&q);
    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>

struct Event {
    int eventId;
    char eventType[50];
    char eventData[100];
    struct Event *next;
};

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

void enqueue(struct Queue *q, int eventId, char *eventType, char *eventData) {
    struct Event *newEvent = (struct Event *)malloc(sizeof(struct Event));
    newEvent->eventId = eventId;
```

```
    strcpy(newEvent->eventType, eventType);
    strcpy(newEvent->eventData, eventData);
    newEvent->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newEvent;
        return;
    }
    q->rear->next = newEvent;
    q->rear = newEvent;
}

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

void search(struct Queue *q, int eventId) {
    struct Event *temp = q->front;
    while (temp) {
        if (temp->eventId == eventId) {
            printf("Event Found: ID = %d, Type = %s, Data = %s\n", temp->eventId,
temp->eventType, temp->eventData);
            return;
        }
```

```
        temp = temp->next;
    }
    printf("Event not found\n");
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Event *temp = q->front;
    while (temp) {
        printf("Event ID: %d, Event Type: %s, Event Data: %s\n", temp->eventId, temp->eventType, temp->eventData);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, "Button Press", "User pressed the button");
    enqueue(&q, 2, "Sensor Triggered", "Temperature sensor crossed threshold");
    display(&q);
    dequeue(&q);
    display(&q);
    search(&q, 1);
    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 <string.h>


struct GUIEvent {

    int eventId;

    char eventType[50];

    int x, y; // Coordinates for screen touch or button click

    long timestamp; // Timestamp of the event

    struct GUIEvent *next;

};


struct Queue {

    struct GUIEvent *front;

    struct GUIEvent *rear;

};


void enqueue(struct Queue *q, int eventId, char *eventType, int x, int y, long
timestamp) {

    struct GUIEvent *newEvent = (struct GUIEvent *)malloc(sizeof(struct GUIEvent));

    newEvent->eventId = eventId;

    strcpy(newEvent->eventType, eventType);

    newEvent->x = x;

    newEvent->y = y;
```

```
    newEvent->timestamp = timestamp;
    newEvent->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newEvent;
        return;
    }
    q->rear->next = newEvent;
    q->rear = newEvent;
}

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

void search(struct Queue *q, int eventId) {
    struct GUIEvent *temp = q->front;
    while (temp) {
        if (temp->eventId == eventId) {
            printf("Event Found: ID = %d, Type = %s, Coordinates = (%d, %d),
Timestamp = %ld\n",
                temp->eventId, temp->eventType, temp->x, temp->y, temp->timestamp);
            return;
        }
```

```
        temp = temp->next;
    }
    printf("Event not found\n");
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct GUIEvent *temp = q->front;
    while (temp) {
        printf("Event ID: %d, Event Type: %s, Coordinates: (%d, %d), Timestamp:
%ld\n",
               temp->eventId, temp->eventType, temp->x, temp->y, temp->timestamp);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, "Button Click", 100, 200, 1617179712);
    enqueue(&q, 2, "Screen Touch", 150, 250, 1617179730);
    display(&q);
    dequeue(&q);
    display(&q);
    search(&q, 1);
    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>

struct SerialData {
    int dataLength;
    char data[256];
    struct SerialData *next;
};

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

void enqueue(struct Queue *q, int dataLength, char *data) {
    struct SerialData *newData = (struct SerialData *)malloc(sizeof(struct SerialData));
    newData->dataLength = dataLength;
    strcpy(newData->data, data);
    newData->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newData;
        return;
    }
```

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

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

void search(struct Queue *q, char *dataPattern) {
    struct SerialData *temp = q->front;
    while (temp) {
        if (strstr(temp->data, dataPattern)) {
            printf("Data Found: %s\n", temp->data);
            return;
        }
        temp = temp->next;
    }
    printf("Data not found\n");
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
```

```
        return;
    }
    struct SerialData *temp = q->front;
    while (temp) {
        printf("Data Length: %d, Data: %s\n", temp->dataLength, temp->data);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 10, "Hello World");
    enqueue(&q, 8, "Data Chunk");
    display(&q);
    dequeue(&q);
    display(&q);
    search(&q, "Data");
    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>
```

```
struct CANMessage {
    int messageId;
    int dataLength;
    char payload[256];
    struct CANMessage *next;
};

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

void enqueue(struct Queue *q, int messageId, int dataLength, char *payload) {
    struct CANMessage *newMessage = (struct CANMessage *)malloc(sizeof(struct
CANMessage));
    newMessage->messageId = messageId;
    newMessage->dataLength = dataLength;
    strcpy(newMessage->payload, payload);
    newMessage->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newMessage;
        return;
    }
    q->rear->next = newMessage;
    q->rear = newMessage;
}

void dequeue(struct Queue *q) {
    if (q->front == NULL) {
```

```
        printf("Queue is empty\n");
        return;
    }
    struct CANMessage *temp = q->front;
    q->front = q->front->next;
    free(temp);
}

void search(struct Queue *q, int messageId) {
    struct CANMessage *temp = q->front;
    while (temp) {
        if (temp->messageId == messageId) {
            printf("Message Found: ID = %d, Data Length = %d, Payload = %s\n",
                temp->messageId, temp->dataLength, temp->payload);
            return;
        }
        temp = temp->next;
    }
    printf("Message not found\n");
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct CANMessage *temp = q->front;
    while (temp) {
        printf("Message ID: %d, Data Length: %d, Payload: %s\n",
```

```
            temp->messageId, temp->dataLength, temp->payload);
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    enqueue(&q, 1, 8, "Hello CAN");
    enqueue(&q, 2, 10, "Message Payload");
    display(&q);
    dequeue(&q);
    display(&q);
    search(&q, 2);
    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.

Each problem requires creating a queue with the following operations using a linked list:

enqueue: Add new elements to the queue.

dequeue: Remove and process elements from the queue.

search: Find elements based on specific criteria.

display: Show all elements in the queue.

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

```
#include <string.h>

struct MLData {
    int featureCount;
    float features[100];  // Assuming a maximum of 100 features
    char metadata[100];
    struct MLData *next;
};

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

void enqueue(struct Queue *q, int featureCount, float *features, char *metadata) {
    struct MLData *newData = (struct MLData *)malloc(sizeof(struct MLData));
    newData->featureCount = featureCount;
    memcpy(newData->features, features, sizeof(float) * featureCount);
    strcpy(newData->metadata, metadata);
    newData->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = newData;
        return;
    }
    q->rear->next = newData;
    q->rear = newData;
}

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

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

void search(struct Queue *q, char *metadata) {
    struct MLData *temp = q->front;
    while (temp) {
        if (strcmp(temp->metadata, metadata) == 0) {
            printf("Data Found: Metadata = %s, Features = [", temp->metadata);
            for (int i = 0; i < temp->featureCount; i++) {
                printf("%f", temp->features[i]);
                if (i != temp->featureCount - 1) printf(", ");
            }
            printf("]\n");
            return;
        }
        temp = temp->next;
    }
    printf("Data not found\n");
}

void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
```

```
        return;
    }
    struct MLData *temp = q->front;
    while (temp) {
        printf("Metadata: %s, Features: [", temp->metadata);
        for (int i = 0; i < temp->featureCount; i++) {
            printf("%f", temp->features[i]);
            if (i != temp->featureCount - 1) printf(", ");
        }
        printf("]\n");
        temp = temp->next;
    }
}

int main() {
    struct Queue q = {NULL, NULL};
    float features[] = {1.0, 2.0, 3.0};
    enqueue(&q, 3, features, "Sample 1");
    float features2[] = {4.0, 5.0, 6.0};
    enqueue(&q, 3, features2, "Sample 2");
    display(&q);
    dequeue(&q);
    display(&q);
    search(&q, "Sample 1");
    return 0;
}
```