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 func();

int main()
{
   for (int i = 0; i < 3; i++)
      func();
   return 0;
}

void func()
{
   static int s = 0;
   s++;
   const int c = s / 2;
   switch (c)
   {
      case 0: printf("1) Static Variable Value = %d, Const Variable Value =
                                                %d\n", s, c);
```

```
            break;
        case 1: printf("2) Static Variable Value = %d, Const Variable Value =
                                                                %d\n", s, c);
            break;
        case 2: printf("3) Static Variable Value = %d, Const Variable Value =
                                                                %d\n", s, c);
            break;
        default: printf("Garbage Value\n");
             break;
    }
}
```

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 func();

int main()
{
    for (int i = 0; i < 3; i++)
        func();
```

```
    return 0;
}

void func()
{
    static int s = 0;
    const int c = 2;
    if (s >= c)
    {
        printf("Limit reached maximum calls\n", c);
        return;
    }
    s++;
    switch (s)
    {
        case 1: printf("Call number: %d\n", s);
                break;
        case 2: printf("Call number: %d\n", s);
                break;
        default: printf("Call number: %d\n", s);
    }
}
```

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 func(char ch);

int main()
{
    char op;
    for(int i = 0; i < 4; i++)
    {
        printf("Enter the op\n+ add\n- sub\n* mul\n");
        scanf(" %c",&op);
        func(op);
    }
    return 0;
}

void func(char ch)
{
    static int a[5] = {5, 3, 6, 8, 4};
    const int n = 5;
    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    switch (ch)
    {
        case '+': for (int i = 0; i < n; i++)
```

```
                a[i] += 3;
            break;
        case '-': for (int i = 0; i < n; i++)
                a[i] -= 2;
            break;
        case '*': for (int i = 0; i < n; i++)
                a[i] *= 5;
            break;
        default: printf("Invalid option\n");
    }
    printf("Updated array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
```

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.

```
#include <stdio.h>

void func();

int main()
{
```

```
    for (int i = 0; i < 4; i++)
        func();
    return 0;
}

void func()
{
    static int c = 0;
    const int t = 3;
    c++;
    switch (c)
    {
        case 1: printf("Counter = %d\n", c);
                break;
        case 3: printf("Counter = %d\n", c);
                break;
        default:if (c < t)
                    printf("Counter = %d\n", c);
                else
                    printf("Counter = %d > threshold %d\n", c, t);
                break;
    }
}
```

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 func();

int main()
{
   for (int i = 0; i < 5; i++)
      func();
   return 0;
}

void func()
{
   static int c = 0;
   const int l = 5;
   c++;
   switch (c)
   {
      case 1: printf("Counter = %d\n", c);
            break;
      case 5: printf("Counter = %d\n", c);
            c = 0;
            break;
      default:printf("Counter = %d\n", c);
            break;
```

```
    }
    //l = 6; // Compilation error
}
```

Looping Statements, Pointers, Const with Pointers, Functions

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(int *a, int n);
void modifyArray(int **a, int n);
void printArray(const int *a, int n);

int main()
{
    const int n = 5;
    int a[n];
    initializeArray(a, n);
    printf("Initialized array:\n");
    printArray(a, n);
    int *p = a;
    modifyArray(&p, n);
    printf("\nModified array:\n");
```

```
    printArray(a, n);
    return 0;
}

void initializeArray(int *a, int n)
{
    for (int i = 0; i < n; i++)
        a[i] = i + 1;
}

void modifyArray(int **a, int n)
{
    for (int i = 0; i < n; i++)
        (*a)[i] += 1;
}

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

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>
#include <stdlib.h>

void readMatrix(int **m, int r, int c);
void transposeMatrix(const int **m, int **t, int r, int c);
void printMatrix(const int **m, int r, int c);

int main()
{
    int r, c;
    printf("Enter the number of rows and columns: ");
    scanf("%d %d", &r, &c);

    int **m = (int **)malloc(r * sizeof(int *));
    for (int i = 0; i < r; i++)
        m[i] = (int *)malloc(c * sizeof(int));

    int **t = (int **)malloc(c * sizeof(int *));
    for (int i = 0; i < c; i++)
        t[i] = (int *)malloc(r * sizeof(int));

    printf("Enter the elements of the matrix:\n");
    readMatrix(m, r, c);

    printf("Original matrix:\n");
    printMatrix((const int **)m, r, c);
```

```
    transposeMatrix((const int **)m, t, r, c);

    printf("Transposed matrix:\n");
    printMatrix((const int **)t, c, r);

    for (int i = 0; i < r; i++)
        free(m[i]);
    free(m);

    for (int i = 0; i < c; i++)
        free(t[i]);
    free(t);

    return 0;
}

void readMatrix(int **m, int r, int c)
{
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
            scanf("%d", &m[i][j]);
    }
}

void transposeMatrix(const int **m, int **t, int r, int c)
{
```

```
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
            t[j][i] = m[i][j];
    }
}

void printMatrix(const int **m, int r, int c)
{
    for (int i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
            printf("%d ", m[i][j]);
        printf("\n");
    }
}
```

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 *a, int size);
void printArray(const int *a, int size);
```

```
int main()
{
    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);
    int *a = (int *)malloc(size * sizeof(int));
    if (a == NULL)
    {
        printf("Memory allocation failed\n");
        return 1;
    }
    initializeArray(a, size);
    printf("Array elements:\n");
    printArray(a, size);
    free(a);
    return 0;
}

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

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

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>
#include <stdlib.h>

void swapArraysCallByValue(int *a1, int *a2, int size1, int size2);
void swapArraysCallByReference(int **a1, int **a2);
void printArray(const int *a, int size);

int main()
{
   int size1, size2;
   printf("Enter the sizes of the first and second array: ");
```

```
scanf("%d %d", &size1, &size2);
int *a1 = (int *)malloc(size1 * sizeof(int));
int *a2 = (int *)malloc(size2 * sizeof(int));
if (a1 == NULL || a2 == NULL)
{
   printf("Memory allocation failed\n");
   return 1;
}
printf("Enter elements of the first array:\n");
for (int i = 0; i < size1; i++)
   scanf("%d", &a1[i]);
printf("Enter elements of the second array:\n");
for (int i = 0; i < size2; i++)
   scanf("%d", &a2[i]);
printf("\nOriginal Arrays\n");
printf("Array 1: ");
printArray(a1, size1);
printf("Array 2: ");
printArray(a2, size2);

swapArraysCallByValue(a1, a2, size1, size2);
printf("\nSwapped Arrays Using Call By Value\n");
printf("Array 1: ");
printArray(a1, size2);
printf("Array 2: ");
printArray(a2, size1);
swapArraysCallByReference(&a1, &a2);
```

```
    printf("\nSwapped Arrays Using Call By Reference\n");
    printf("Array 1: ");
    printArray(a1, size2);
    printf("Array 2: ");
    printArray(a2, size1);
    free(a1);
    free(a2);
    return 0;
}

void swapArraysCallByValue(int *a1, int *a2, int size1, int size2)
{
    int size = (size1 < size2) ? size1 : size2;
    for (int i = 0; i < size; i++)
    {
        int t = a1[i];
        a1[i] = a2[i];
        a2[i] = t;
    }
}

void swapArraysCallByReference(int **a1, int **a2)
{
    int *t = *a1;
    *a1 = *a2;
    *a2 = t;
}
```

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

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

#define MAX_CHARACTERS 256

void readString(char *s);
void countCharacterFrequency(const char *s);

int main()
{
   char s[100];
   readString(s);
   countCharacterFrequency(s);
   return 0;
```

```
}

void readString(char *str)
{
    printf("Enter a string: ");
    scanf("%s", str);
}

void countCharacterFrequency(const char *s)
{
    int f[MAX_CHARACTERS] = {0};
    int i = 0;
    do
    {
        char ch = s[i];
        if (ch != '\0')
            f[(unsigned char)ch]++;
        i++;
    } while (s[i - 1] != '\0');

    printf("\nCharacter frequencies:\n");
    for (int i = 0; i < MAX_CHARACTERS; i++)
    {
        if (f[i] > 0)
            printf("'%c': %d\n", i, f[i]);
    }
}
```

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>
#include <stdlib.h>

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

typedef struct
{
   char name[50];
   int age;
   float salary;
   Address address;
} Employee;

void enterEmployeeInfo(Employee *emp);
void displayEmployeeInfo(const Employee *emp);

int main()
```

```
{
    int n;
    printf("Enter the number of employees: ");
    scanf("%d", &n);
    Employee *employees = (Employee *)malloc(n * sizeof(Employee));
    if (employees == NULL)
    {
        printf("Memory allocation failed\n");
        return 1;
    }
    for (int i = 0; i < n; i++)
    {
        printf("\nEnter information for employee %d\n", i + 1);
        enterEmployeeInfo(&employees[i]);
    }
    printf("\nEmployee Information\n");
    for (int i = 0; i < n; i++)
        displayEmployeeInfo(&employees[i]);
    free(employees);
    return 0;
}

void enterEmployeeInfo(Employee *emp)
{
    printf("Enter name: ");
    scanf(" %[^\n]", emp->name);
```

```
    printf("Enter age: ");
    scanf("%d", &emp->age);
    printf("Enter salary: ");
    scanf("%f", &emp->salary);
    printf("Enter city: ");
    scanf(" %[^\n]", emp->address.city);
    printf("Enter state: ");
    scanf(" %[^\n]", emp->address.state);
}

void displayEmployeeInfo(const Employee *emp)
{
    printf("Name: %s\n", emp->name);
    printf("Age: %d\n", emp->age);
    printf("Salary: %.2f\n", emp->salary);
    printf("Address -\n");
    printf("City: %s\n", emp->address.city);
    printf("State: %s\n", emp->address.state);
}
```

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>
#include <string.h>
```

```
typedef union
{
   int i;
   float f;
   char s[100];
} Data;

typedef struct
{
   char type;   // i for int, f for float, s for string
   Data data;
} Data1;

void displayData(const Data1 *d);

int main()
{
   Data1 a[3];
   a[0].type = 'i';
   a[0].data.i = 8;
   a[1].type = 'f';
   a[1].data.f = 23.4;
   a[2].type = 's';
   strcpy(a[2].data.s, "Good Day");
   for (int i = 0; i < 3; i++)
      displayData(&a[i]);
   return 0;
```

```
}

void displayData(const Data1 *d)
{
   printf("Type: ");
   switch (d->type)
   {
      case 'i': printf("Integer\n");
                printf("Value: %d\n", d->data.i);
                break;
      case 'f': printf("Float\n");
                printf("Value: %.2f\n", d->data.f);
                break;
      case 's': printf("String\n");
                printf("Value: %s\n", d->data.s);
                break;
      default:printf("Unknown type\n");
   }
   printf("\n");
}
```

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>
#include <stdlib.h>

#define MAX_NAME_LENGTH 100
#define MAX_NAMES 2

typedef union
{
   int length;
   char reverse[MAX_NAME_LENGTH];
} String;

typedef struct
{
   char name[MAX_NAME_LENGTH];
   String info;
   int s; // length-1 and reversed-0
} Name;

void reverseString(char *s);
void displayNameInfo(const Name *n);

int main()
{
   Name names[MAX_NAMES];
   int option;
   for (int i = 0; i < MAX_NAMES; i++)
```

```
    {
        printf("Enter name %d: ", i + 1);
        scanf("%s", names[i].name);
        printf("Select the option\n");
        printf("1. Store length\n");
        printf("2. Store reverse string\n");
        printf("Enter the option: ");
        scanf("%d", &option);
        switch(option)
        {
            case 1: names[i].info.length = strlen(names[i].name);
                    printf("Length = %d\n", names[i].info.length);
                    break;
            case 2: reverseString(names[i].name);
                    strcpy(names[i].info.reverse, names[i].name);
                    break;
            default: printf("Invalid option\n");
        }
    }
    printf("\nStored Information: \n");
    for (int i = 0; i < MAX_NAMES; i++)
        displayNameInfo(&names[i]);
    return 0;
}

void reverseString(char *s)
{
```

```
    int start = 0;
    int end = strlen(s) - 1;
    while (start < end)
    {
        char t = s[start];
        s[start] = s[end];
        s[end] = t;
        start++;
        end--;
    }
}

void displayNameInfo(const Name *n)
{
    if (n->s)
        printf("Length = %d\n", n->info.length);
    else
        printf("Reversed String = %s\n", n->info.reverse);
}
```

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>
#include <string.h>
```

```
#define MAX_NAME_LENGTH 100
#define MAX_ENTRIES 3

typedef struct
{
   int id;
   char name[MAX_NAME_LENGTH];
} Employee;

typedef union
{
   int age;
   float salary;
   Employee employeeData;
} Data;

typedef struct
{
   char type; //'i' for age, 'f' for salary, 'e' for Employee
   Data data;
} Info;

void inputData(Info *info, int index);
void displayData(const Info *info);

int main()
{
```

```
    Info entries[MAX_ENTRIES];
    for (int i = 0; i < MAX_ENTRIES; i++)
    {
        inputData(&entries[i], i);
        displayData(&entries[i]);
    }
    return 0;
}

void inputData(Info *info, int index)
{
    printf("\nEnter information for entry %d:\n", index + 1);
    printf("Choose the type of data:\n");
    printf("1. Age\n");
    printf("2. Salary\n");
    printf("3. Employee Info\n");
    int option;
    printf("Enter the option: ");
    scanf("%d", &option);
    switch (option)
    {
        case 1: info->type = 'i';
                printf("Enter age: ");
                scanf(" %d", &info->data.age);
                break;
        case 2: info->type = 'f';
                printf("Enter salary: ");
```

```
            scanf(" %f", &info->data.salary);
            break;
        case 3: info->type = 'e';
            printf("Enter employee ID: ");
            scanf(" %d", &info->data.employeeData.id);
            printf("Enter employee name: ");
            scanf(" %[^\n]", info->data.employeeData.name);
            break;
        default: printf("Invalid option\n");
    }
}

void displayData(const Info *info)
{
    printf("\nStored Information\n");
    switch (info->type)
    {
        case 'i': printf("Type: Age: %d\n", info->data.age);
                  break;
        case 'f': printf("Type: Salary = %f\n", info->data.salary);
                  break;
        case 'e': printf("Type: Employee\n");
                  printf("Employee ID: %d\n", info->data.employeeData.id);
                  printf("Employee Name: %s\n",
                                        info->data.employeeData.name);
                  break;
        default: printf("Unknown data type\n");
```

```
    }
}
```

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>
#include <string.h>

typedef struct
{
    char name[100];
} Author;

typedef union
{
    int pages;
    int publicationYear;
} Book;

typedef struct
{
    char title[100];
    Author author;
    Book details;
```

```
    int type; // 1-pages and 2-publication year
} BookDetails;

void inputBookInfo(BookDetails *book);
void displayBookInfo(const BookDetails *book);

int main()
{
    BookDetails book;
    inputBookInfo(&book);
    displayBookInfo(&book);
    return 0;
}

void inputBookInfo(BookDetails *book)
{
    printf("Enter the book title: ");
    scanf(" %[^\n]", book->title);
    printf("Enter the author's name: ");
    scanf(" %[^\n]", book->author.name);
    printf("Choose the type\n");
    printf("1. Number of Pages\n");
    printf("2. Publication Year\n");
    printf("Enter the option: ");
    scanf("%d", &book->type);
    switch(book->type)
    {
```

```
        case 1: printf("Enter the number of pages: ");
                scanf("%d", &book->details.pages);
                break;
        case 2: printf("Enter the publication year: ");
                scanf("%d", &book->details.publicationYear);
                break;
        default: printf("Invalid option\n");
    }
}

void displayBookInfo(const BookDetails *book)
{
    printf("\nBook Information\n");
    printf("Title: %s\n", book->title);
    printf("Author: %s\n", book->author.name);
    if (book->type == 1)
        printf("Number of Pages: %d\n", book->details.pages);
    else if (book->type == 2)
        printf("Publication Year: %d\n", book->details.publicationYear);
}
```

Stacks Using Arrays and Linked List

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.

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

#define MAX 5

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

int isFull(const Stack *s);
int isEmpty(const Stack *s);
void push(Stack *s, int value);
int pop(Stack *s);
int peek(const Stack *s);

int main()
{
   Stack s;
   s.top = -1;
   printf("Pushing elements to the stack:\n");
   push(&s, 10);
   push(&s, 20);
   push(&s, 30);
   push(&s, 40);
   push(&s, 50);
```

```
    printf("\nPeeking the top element: %d\n", peek(&s));
    printf("\nPopping elements from the stack:\n");
    printf("Popped: %d\n", pop(&s));
    printf("Popped: %d\n", pop(&s));
    printf("\nPeeking the top element after popping: %d\n", peek(&s));
    pop(&s);
    pop(&s);
    pop(&s);
    printf("\nPopping from empty stack: %d\n", pop(&s));
    return 0;
}

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

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

void push(Stack *s, int value)
{
    if (isFull(s))
    {
        printf("Stack is full\n");
```

```
        return;
    }
    s->a[++(s->top)] = value;
    printf("Pushed: %d\n", value);
}

int pop(Stack *s)
{
    if (isEmpty(s))
    {
        printf("Stack is empty\n");
        return -1;
    }
    return s->a[(s->top)--];
}

int peek(const Stack *s)
{
    if (isEmpty(s))
    {
        printf("Stack is empty\n");
        return -1;
    }
    return s->a[s->top];
}
```

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;

int push(Node **top, int value);
int pop(Node **top);
int peek(Node *top);
int isEmpty(Node *top);
int isFull();

int main()
{
   Node *stack = NULL;
   printf("Pushing elements to the stack:\n");
   push(&stack, 10);
   push(&stack, 20);
   push(&stack, 30);
   printf("\nPeeking the top element: %d\n", peek(stack));
```

```
    printf("\nPopping elements from the stack:\n");
    printf("Popped: %d\n", pop(&stack));
    printf("Popped: %d\n", pop(&stack));
    printf("\nPeeking the top element after popping: %d\n", peek(stack));
    printf("Popped: %d\n", pop(&stack));
    printf("\nPopping from empty stack: %d\n", pop(&stack));
    return 0;
}

int push(Node **top, int value)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed\n");
        return 0;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
    printf("Pushed: %d\n", value);
    return 1;
}

int pop(Node **top)
{
    if (isEmpty(*top))
```

```
    {
        printf("Stack is empty\n");
        return -1;
    }
    Node *temp = *top;
    int value = temp->data;
    *top = temp->next;
    free(temp);
    return value;
}

int peek(Node *top)
{
    if (isEmpty(top))
    {
        printf("Stack is empty\n");
        return -1;
    }
    return top->data;
}

int isEmpty(Node *top)
{
    return top == NULL;
}

int isFull()
```

```
{
    return 0;
}
```

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 100

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

int push(Stack *s, int value);
int isEmpty(Stack *s);
int isFull();
void reverseStack(Stack *s);
void display(Stack *s);

int main()
{
    Stack stack;
```

```
    stack.top = -1;
    printf("Pushing elements to the stack:\n");
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    push(&stack, 40);
    printf("\nStack before reversal:\n");
    display(&stack);
    reverseStack(&stack);
    printf("\nStack after reversal:\n");
    display(&stack);
    return 0;
}

int push(Stack *s, int value)
{
    if (isFull(s))
    {
        printf("Stack overflow\n");
        return 0;
    }
    s->a[++(s->top)] = value;
    printf("Pushed: %d\n", value);
    return 1;
}

int isEmpty(Stack *s)
```

```
{
    return s->top == -1;
}

int isFull()
{
    return 0;
}

void reverseStack(Stack *s)
{
    if (isEmpty(s))
    {
        printf("Stack is empty\n");
        return;
    }
    int start = 0;
    int end = s->top;
    while (start < end)
    {
        int t = s->a[start];
        s->a[start] = s->a[end];
        s->a[end] = t;
        start++;
        end--;
    }
    printf("Stack has been reversed\n");
```

```
}

void display(Stack *s)
{
   if (isEmpty(s))
   {
      printf("Stack is empty\n");
      return;
   }
   for (int i = s->top; i >= 0; i--)
      printf("%d ", s->a[i]);
   printf("\n");
}
```

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;
```

```
void push(Node **top, int value);
int isEmpty(Node *top);
void mergeStacks(Node **stack1, Node **stack2);
void display(Node *top);

int main()
{
    Node *stack1 = NULL;
    Node *stack2 = NULL;
    printf("Pushing elements to stack1:\n");
    push(&stack1, 10);
    push(&stack1, 20);
    push(&stack1, 30);
    printf("Stack1 elements:\n");
    display(stack1);
    printf("\nPushing elements to stack2:\n");
    push(&stack2, 40);
    push(&stack2, 50);
    push(&stack2, 60);
    printf("Stack2 elements:\n");
    display(stack2);
    printf("\nMerging stack2 into stack1\n");
    mergeStacks(&stack1, &stack2);
    printf("Merged stack elements:\n");
    display(stack1);
    return 0;
}
```

```
void push(Node **top, int value)
{
   Node *newNode = (Node *)malloc(sizeof(Node));
   if (newNode == NULL)
   {
      printf("Memory allocation failed\n");
      return;
   }
   newNode->data = value;
   newNode->next = *top;
   *top = newNode;
   printf("Pushed: %d\n", value);
}

int isEmpty(Node *top)
{
   return top == NULL;
}

void mergeStacks(Node **stack1, Node **stack2)
{
   if (isEmpty(*stack2))
   {
      printf("Stack2 is empty\n");
      return;
   }
   if (isEmpty(*stack1))
```

```
    {
        *stack1 = *stack2;
        *stack2 = NULL;
        printf("Stack1 is empty so stack2 merged into stack1\n");
        return;
    }
    Node *t = *stack2;
    while (t->next != NULL)
        t = t->next;
    t->next = *stack1;
    *stack1 = *stack2;
    *stack2 = NULL;
    printf("Successfully merged stack2 into stack1\n");
}

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

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

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

#define MAX 100

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

int push(Stack *s, char value);
char pop(Stack *s);
int isEmpty(Stack *s);
int isFull(Stack *s);
int checkBalancedParentheses(const char *exp);

int main()
```

```
{
    char exp1[] = "(a + b) * (c - d)";
    char exp2[] = "{[a + b] * (c - d)}";
    char exp3[] = "(a + b))";
    printf("Checking balanced parentheses:\n");
    printf("Expression: %s -> %s\n", exp1,
          checkBalancedParentheses(exp1) ? "Balanced" : "Not Balanced");
    printf("Expression: %s -> %s\n", exp2,
           checkBalancedParentheses(exp2) ? "Balanced" : "Not Balanced");
    printf("Expression: %s -> %s\n", exp3,
          checkBalancedParentheses(exp3) ? "Balanced" : "Not Balanced");
    return 0;
}

int push(Stack *s, char value)
{
    if (isFull(s))
    {
        printf("Stack overflow\n");
        return 0;
    }
    s->a[++(s->top)] = value;
    return 1;
}

char pop(Stack *s)
{
```

```
    if (isEmpty(s))
    {
        printf("Stack underflow\n");
        return '\0';
    }
    return s->a[(s->top)--];
}

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

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

int checkBalancedParentheses(const char *exp)
{
    Stack stack;
    stack.top = -1;
    for (int i = 0; exp[i] != '\0'; i++)
    {
        char ch = exp[i];
        if (ch == '(' || ch == '{' || ch == '['
            push(&stack, ch);
```

```
        else if (ch == ')' || ch == '}' || ch == ']')
        {
            if (isEmpty(&stack))
                return 0;
            char top = pop(&stack);
            if ((ch == ')' && top != '(') || (ch == '}' && top != '{') ||
                (ch == ']' && top != '['))
                    return 0;
        }
    }
    return isEmpty(&stack);
}
```

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

```
void push(Node **top, int value);
int pop(Node **top);
int isEmpty(Node *top);
int evaluatePostfix(const char *exp);

int main()
{
   char exp1[] = "24*19*+";
   char exp2[] = "8+15-/";
   char exp3[] = "18+45*6-";
   printf("Postfix Expression Evaluation\n");
   printf("Expression: %s -> Result: %d\n", exp1, evaluatePostfix(exp1));
   printf("Expression: %s -> Result: %d\n", exp2, evaluatePostfix(exp2));
   printf("Expression: %s -> Result: %d\n", exp3, evaluatePostfix(exp3));
   return 0;
}

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

```
    *top = newNode;
}

int pop(Node **top)
{
    if (isEmpty(*top))
    {
        printf("Stack is empty\n");
        return -1;
    }
    Node *temp = *top;
    int value = temp->data;
    *top = temp->next;
    free(temp);
    return value;
}

int isEmpty(Node *top) {
    return top == NULL;
}

int evaluatePostfix(const char *exp)
{
    Node *stack = NULL;
    for (int i = 0; exp[i] != '\0'; i++)
    {
        char ch = exp[i];
```

```
        if (isdigit(ch))
            push(&stack, ch - '0');
        else
        {
            int op2 = pop(&stack);
            int op1 = pop(&stack);
            switch (ch) {
                case '+': push(&stack, op1 + op2);
                          break;
                case '-': push(&stack, op1 - op2);
                          break;
                case '*': push(&stack, op1 * op2);
                          break;
                case '/': if (op2 == 0)
                          {
                              printf("Division by zero\n");
                              return -1;
                          }
                          push(&stack, op1 / op2);
                          break;
                default: printf("Invalid operator\n");
                         return -1;
            }
        }
    }
    return pop(&stack);
}
```

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>

struct Queue
{
   int size;
   int front;
   int rear;
   char **students;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue q;
   createQueue(&q, 3);
```

```
    enqueue(&q, "ABC");
    enqueue(&q, "DEF");
    enqueue(&q, "GHI");
    displayQueue(q);
    printf("\nAdmitting student: %s\n", dequeue(&q));
    displayQueue(q);
    enqueue(&q, "JKL");
    displayQueue(q);
    printf("\nAdmitting student: %s\n", dequeue(&q));
    displayQueue(q);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->students = (char **)malloc(q->size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
```

```
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Added student: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *admittedStudent = q->students[q->front];
        return admittedStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
```

```
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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 MAX_NAME_LENGTH 50

struct Queue
{
    int size;
    int front;
    int rear;
    char **students;
};

void createQueue(struct Queue *, int);
```

```
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int totalBooks = 5;
   createQueue(&queue, totalBooks);
   enqueue(&queue, "ABC");
   enqueue(&queue, "DEF");
   enqueue(&queue, "GHI");
   displayQueue(queue);
   printf("\nStudent borrowing a book: %s\n", dequeue(&queue));
   displayQueue(queue);
   enqueue(&queue, "JKL");
   enqueue(&queue, "MNO");
   displayQueue(queue);
   printf("\nStudent borrowing a book: %s\n", dequeue(&queue));
   displayQueue(queue);
   return 0;
}

void createQueue(struct Queue *q, int size)
{
   q->size = size;
   q->front = q->rear = -1;
```

```
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student added: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *borrowedStudent = q->students[q->front];
```

```
        return borrowedStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
```

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>
#include <string.h>

#define MAX_NAME_LENGTH 50
```

```
struct Queue
{
   int size;
   int front;
   int rear;
   char **students;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
   createQueue(&queue, total);
   enqueue(&queue, "ABC");
   enqueue(&queue, "DEF");
   enqueue(&queue, "GHI");
   displayQueue(queue);
   printf("\nServing student: %s\n", dequeue(&queue));
   displayQueue(queue);
   enqueue(&queue, "JKL");
   displayQueue(queue);
   printf("\nServing student: %s\n", dequeue(&queue));
```

```
    displayQueue(queue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Issued token to student: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
```

```
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *servedStudent = q->students[q->front];
        return servedStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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_NAME_LENGTH 50

struct Queue
{
   int size;
   int front;
   int rear;
   char **students;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
   createQueue(&queue, total);
   enqueue(&queue, "ABC");
```

```
    enqueue(&queue, "DEF");
    enqueue(&queue, "GHI");
    displayQueue(queue);
    printf("\nHelping student: %s\n", dequeue(&queue));
    displayQueue(queue);
    enqueue(&queue, "JKL");
    enqueue(&queue, "MNO");
    displayQueue(queue);
    printf("\nHelping student: %s\n", dequeue(&queue));
    displayQueue(queue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
```

```
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student added: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *helpedStudent = q->students[q->front];
        return helpedStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
```

```
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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 MAX_NAME_LENGTH 50

struct Queue
{
    int size;
    int front;
    int rear;
    char **students;
};

void createQueue(struct Queue *, int);
```

```
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
   createQueue(&queue, total);
   enqueue(&queue, "ABC");
   enqueue(&queue, "DEF");
   enqueue(&queue, "GHI");
   displayQueue(queue);
   printf("\nProcessing registration for student: %s\n", dequeue(&queue));
   displayQueue(queue);
   enqueue(&queue, "JKL");
   displayQueue(queue);
   return 0;
}

void createQueue(struct Queue *q, int size)
{
   q->size = size;
   q->front = q->rear = -1;
   q->students = (char **)malloc(size * sizeof(char *));
}
```

```
void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student added: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *registeredStudent = q->students[q->front];
        return registeredStudent;
    }
}
```

```
void displayQueue(struct Queue q)
{
   if (q.front == q.rear)
      printf("Queue is empty\n");
   else
   {
      printf("Students in the registration queue:\n");
      for (int i = q.front + 1; i <= q.rear; i++)
         printf("%s -> ", q.students[i]);
      printf("NULL\n");
   }
}
```

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>
#include <string.h>

#define MAX_NAME_LENGTH 50

struct Queue
{
   int size;
```

```
    int front;
    int rear;
    char **students;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
    struct Queue queue;
    int total = 5;
    createQueue(&queue, total);
    enqueue(&queue, "ABC");
    enqueue(&queue, "DEF");
    enqueue(&queue, "GHI");
    displayQueue(queue);
    printf("\nStudent boarding the bus: %s\n", dequeue(&queue));
    displayQueue(queue);
    enqueue(&queue, "JKL");
    displayQueue(queue);
    printf("\nStudent boarding the bus: %s\n", dequeue(&queue));
    displayQueue(queue);
    return 0;
}
```

```
void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student arrived: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty. No student to board.\n");
        return NULL;
    }
```

```
    else
    {
        q->front++;
        char *boardingStudent = q->students[q->front];
        return boardingStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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>
#include <string.h>

#define MAX_NAME_LENGTH 50

struct Queue
{
   int size;
   int front;
   int rear;
   char **students;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
   createQueue(&queue, total);
   enqueue(&queue, "ABC");
   enqueue(&queue, "DEF");
   enqueue(&queue, "GHI");
   displayQueue(queue);
```

```
    printf("\nServing student for counseling: %s\n", dequeue(&queue));
    displayQueue(queue);
    enqueue(&queue, "JKL");
    displayQueue(queue);
    printf("\nServing student for counseling: %s\n", dequeue(&queue));
    displayQueue(queue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student arrived: %s\n", studentName);
    }
```

```
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear) {
        printf("Queue is empty\n");
        return NULL;
    }
    else {
        q->front++;
        char *servingStudent = q->students[q->front];
        return servingStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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_NAME_LENGTH 50

struct Queue
{
   int size;
   int front;
   int rear;
   char **students;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
```

```
    createQueue(&queue, total);
    enqueue(&queue, "ABC");
    enqueue(&queue, "DEF");
    enqueue(&queue, "GHI");
    displayQueue(queue);
    printf("\nProcessing registration for student: %s\n", dequeue(&queue));
    displayQueue(queue);
    enqueue(&queue, "JKL");
    displayQueue(queue);
    printf("\nProcessing registration for student: %s\n", dequeue(&queue));
    displayQueue(queue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
```

```
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student registered: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *processingStudent = q->students[q->front];
        return processingStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
```

```
    {
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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 MAX_NAME_LENGTH 50

struct Queue
{
    int size;
    int front;
    int rear;
    char **students;
};
```

```
void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
   createQueue(&queue, total);
   enqueue(&queue, "ABC");
   enqueue(&queue, "DEF");
   enqueue(&queue, "GHI");
   displayQueue(queue);
   printf("\nServing student: %s\n", dequeue(&queue));
   displayQueue(queue);
   enqueue(&queue, "JKL");
   displayQueue(queue);
   printf("\nServing student: %s\n", dequeue(&queue));
   displayQueue(queue);
   return 0;
}

void createQueue(struct Queue *q, int size)
{
   q->size = size;
   q->front = q->rear = -1;
```

```
    q->students = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *studentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->students[q->rear] = (char *)malloc(strlen(studentName) + 1);
        strcpy(q->students[q->rear], studentName);
        printf("Student waiting: %s\n", studentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *servingStudent = q->students[q->front];
```

```
        return servingStudent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
        printf("Students in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.students[i]);
        printf("NULL\n");
    }
}
```

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>
#include <string.h>

#define MAX_NAME_LENGTH 50
```

```
struct Queue
{
   int size;
   int front;
   int rear;
   char **parents;
};

void createQueue(struct Queue *, int);
void enqueue(struct Queue *, const char *);
char *dequeue(struct Queue *);
void displayQueue(struct Queue);

int main()
{
   struct Queue queue;
   int total = 5;
   createQueue(&queue, total);
   enqueue(&queue, "Mr. ABC");
   enqueue(&queue, "Mrs. DEF");
   enqueue(&queue, "Mr. GHI");
   displayQueue(queue);
   printf("\nServing parent: %s\n", dequeue(&queue));
   displayQueue(queue);
   enqueue(&queue, "Mrs. JKL");
   displayQueue(queue);
   printf("\nServing parent: %s\n", dequeue(&queue));
```

```
    displayQueue(queue);
    return 0;
}

void createQueue(struct Queue *q, int size)
{
    q->size = size;
    q->front = q->rear = -1;
    q->parents = (char **)malloc(size * sizeof(char *));
}

void enqueue(struct Queue *q, const char *parentName)
{
    if (q->rear == q->size - 1)
        printf("Queue is full\n");
    else
    {
        q->rear++;
        q->parents[q->rear] = (char *)malloc(strlen(parentName) + 1);
        strcpy(q->parents[q->rear], parentName);
        printf("Parent waiting: %s\n", parentName);
    }
}

char *dequeue(struct Queue *q)
{
    if (q->front == q->rear)
```

```
    {
        printf("Queue is empty\n");
        return NULL;
    }
    else
    {
        q->front++;
        char *servingParent = q->parents[q->front];
        return servingParent;
    }
}

void displayQueue(struct Queue q)
{
    if (q.front == q.rear)
        printf("Queue is empty\n");
    else
    {
        printf("Parents in the queue:\n");
        for (int i = q.front + 1; i <= q.rear; i++)
            printf("%s -> ", q.parents[i]);
        printf("NULL\n");
    }
}
```

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>

struct SensorData
{
   int temperature;
   int pressure;
   char timestamp[20];
   struct SensorData *next;
} *front = NULL, *rear = NULL;

void enqueue(int temp, int pressure, const char *timestamp);
void display();
int dequeue();
void searchByTimestamp(const char *timestamp);

int main()
{
   enqueue(23, 1002, "2025-01-20 10:00");
```

```
    enqueue(28, 1300, "2025-01-20 11:05");
    enqueue(30, 1655, "2025-01-20 12:10");
    printf("Sensor Readings in the Queue:\n");
    display();
    printf("\nDequeuing the oldest reading: %d\n", dequeue());
    printf("\nSensor Readings after Dequeue:\n");
    display();
    printf("\nSearching for a reading with timestamp:\n");
    searchByTimestamp("2025-01-20 11:05");
    return 0;
}

void enqueue(int temp, int pressure, const char *timestamp)
{
    struct SensorData *newData = (struct SensorData *)malloc
                                            (sizeof(struct SensorData));
    if (newData == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newData->temperature = temp;
    newData->pressure = pressure;
   strncpy(newData->timestamp, timestamp,sizeof(newData->timestamp) - 1);
    newData->timestamp[sizeof(newData->timestamp) - 1] = '\0';
    newData->next = NULL;
    if (front == NULL)
```

```
        front = rear = newData;
    else
    {
        rear->next = newData;
        rear = newData;
    }
    printf("Enqueued data: Temp = %d, Pressure = %d, Timestamp = %s\n",
temp, pressure, timestamp);
}

void display()
{
    struct SensorData *current = front;
    if (current == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    while (current != NULL)
    {
        printf("Temp = %d, Pressure = %d, Timestamp = %s\n",
                current->temperature, current->pressure, current->timestamp);
        current = current->next;
    }
}

int dequeue()
{
```

```
    if (front == NULL)
    {
        printf("Queue is empty, no data to dequeue\n");
        return -1;
    }
    struct SensorData *temp = front;
    int tempData = front->temperature;
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(temp);
    return tempData;
}

void searchByTimestamp(const char *timestamp)
{
    struct SensorData *current = front;
    while (current != NULL)
    {
        if (strcmp(current->timestamp, timestamp) == 0)
        {
            printf("Found reading with Timestamp = %s : Temp = %d, Pressure =
                    %d\n", timestamp, current->temperature, current->pressure);
            return;
        }
        current = current->next;
    }
```

```
    printf("No reading found with Timestamp = %s\n", timestamp);
}
```

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;
} *front = NULL, *rear = NULL;

void enqueue(int taskID, int priority, int executionTime);
void dequeue();
void display();
void searchByPriority(int priority);

int main()
{
```

```
    enqueue(1, 4, 8);
    enqueue(2, 3, 15);
    enqueue(3, 2, 1);
    printf("Tasks in the Queue:\n");
    display();
    printf("\nDequeueing task...\n");
    dequeue();
    display();
    printf("\nSearching for tasks with priority:\n");
    searchByPriority(4);
    return 0;
}

void enqueue(int taskID, int priority, int executionTime)
{
    struct Task *newTask = (struct Task *)malloc(sizeof(struct Task));
    if (newTask == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newTask->taskID = taskID;
    newTask->priority = priority;
    newTask->executionTime = executionTime;
    newTask->next = NULL;
    if (front == NULL)
        front = rear = newTask;
```

```
    else
    {
        struct Task *current = front;
        struct Task *prev = NULL;
        while (current != NULL && current->priority >= priority)
        {
            prev = current;
            current = current->next;
        }
        if (prev == NULL)
        {
            newTask->next = front;
            front = newTask;
        }
        else
        {
            prev->next = newTask;
            newTask->next = current;
            if (current == NULL)
                rear = newTask;
        }
    }
    printf("Enqueued task: ID = %d, Priority = %d, Execution Time = %d\n",
taskID, priority, executionTime);
}

void dequeue()
{
```

```
    if (front == NULL)
    {
        printf("Queue is empty, no task to dequeue\n");
        return;
    }
    struct Task *temp = front;
    printf("Dequeued task: ID = %d, Priority = %d, Execution Time = %d\n",
                                temp->taskID, temp->priority, temp->executionTime);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(temp);
}

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

```
    }
}

void searchByPriority(int priority)
{
    struct Task *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (current->priority == priority)
        {
            printf("Found task with Priority = %d: ID = %d, Execution Time =
                    %d\n", priority, current->taskID, current->executionTime);
            found = 1;
        }
        current = current->next;
    }
    if (!found)
        printf("No tasks found with Priority = %d\n", priority);
}
```

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>
#include <string.h>

struct IRQ
{
    int priority;
    char source[50];
    void (*handler)(void);
    struct IRQ *next;
} *front = NULL, *rear = NULL;

void enqueue(int priority, const char *source, void (*handler)(void));
void dequeue();
void display();
void searchBySource(const char *source);
void handlerA();
void handlerB();
void handlerC();

int main()
{
    enqueue(5, "Sensor", handlerA);
    enqueue(8, "Timer", handlerB);
    enqueue(3, "Network", handlerC);
    printf("Interrupt Queue:\n");
    display();
    printf("\nDequeueing highest-priority interrupt:\n");
```

```
    dequeue();
    display();
    printf("\nSearching for interrupt from:\n");
    searchBySource("Sensor");
    return 0;
}

void enqueue(int priority, const char *source, void (*handler)(void))
{
    struct IRQ *newIRQ = (struct IRQ *)malloc(sizeof(struct IRQ));
    if (newIRQ == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newIRQ->priority = priority;
    strcpy(newIRQ->source, source);
    newIRQ->handler = handler;
    newIRQ->next = NULL;
    if (front == NULL || front->priority < priority)
    {
        newIRQ->next = front;
        front = newIRQ;
        if (rear == NULL) rear = newIRQ;
    }
    else
    {
```

```
        struct IRQ *current = front;
        while (current->next != NULL && current->next->priority >= priority)
            current = current->next;
        newIRQ->next = current->next;
        current->next = newIRQ;
        if (newIRQ->next == NULL)
            rear = newIRQ;
    }
    printf("Enqueued IRQ: Priority = %d, Source = %s\n", priority, source);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no IRQ to dequeue.\n");
        return;
    }
    struct IRQ *t = front;
    printf("Dequeued IRQ: Priority = %d, Source = %s\n", t->priority,
                                                        t->source);
    t->handler();
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}
```

```
void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct IRQ *current = front;
    while (current != NULL)
    {
        printf("Priority = %d, Source = %s\n", current->priority,
                                                    current->source);
        current = current->next;
    }
}

void searchBySource(const char *source)
{
    struct IRQ *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strcmp(current->source, source) == 0)
        {
            printf("Found IRQ: Priority = %d, Source = %s\n", current->priority,
                                                        current->source);
            found = 1;
```

```
        }
        current = current->next;
    }
    if (!found)
        printf("No IRQs found from source: %s\n", source);
}

void handlerA()
{
    printf("Executing Handler A\n");
}

void handlerB()
{
    printf("Executing Handler B\n");
}

void handlerC()
{
    printf("Executing Handler C\n");
}
```

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>

struct Message
{
    int senderID;
    int receiverID;
    char payload[100];
    struct Message *next;
}*front = NULL, *rear = NULL;

void enqueue(int senderID, int receiverID, const char *payload);
void dequeue();
void display();
void searchBySenderID(int senderID);

int main()
{
    enqueue(1, 2, "One");
    enqueue(2, 3, "Good");
    enqueue(1, 4, "Day");
    printf("Message Queue:\n");
    display();
    printf("\nDequeuing the oldest message:\n");
    dequeue();
    display();
```

```
    printf("\nSearching for messages from Sender ID:\n");
    searchBySenderID(1);
    return 0;
}

void enqueue(int senderID, int receiverID, const char *payload)
{
     struct Message *newMessage = (struct Message *)malloc(sizeof(struct
Message));
    if (newMessage == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newMessage->senderID = senderID;
    newMessage->receiverID = receiverID;
    strcpy(newMessage->payload, payload);
    newMessage->next = NULL;
    if (rear == NULL)
        front = rear = newMessage;
    else
    {
        rear->next = newMessage;
        rear = newMessage;
    }
    printf("Enqueued Message: Sender ID = %d, Receiver ID = %d,
            Payload = \"%s\"\n", senderID, receiverID, payload);
}
```

```
void dequeue()
{
   if (front == NULL)
   {
      printf("Queue is empty, no message to dequeue\n");
      return;
   }
   struct Message *t = front;
   printf("Dequeued Message: Sender ID = %d, Receiver ID = %d,
                  Payload = \"%s\"\n", t->senderID, t->receiverID, t->payload);
   front = front->next;
   if (front == NULL)
      rear = NULL;
   free(t);
}

void display()
{
   if (front == NULL)
   {
      printf("Queue is empty\n");
      return;
   }
   struct Message *current = front;
   while (current != NULL)
   {
      printf("Sender ID = %d, Receiver ID = %d, Payload = \"%s\"\n",
```

```
            current->senderID, current->receiverID, current->payload);
        current = current->next;
    }
}

void searchBySenderID(int senderID)
{
    struct Message *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (current->senderID == senderID)
        {
            printf("Found Message: Sender ID = %d, Receiver ID = %d,
                    Payload = \"%s\"\n", current->senderID, current->receiverID,
                                                        current->payload);
            found = 1;
        }
        current = current->next;
    }
    if (!found)
        printf("No messages found from Sender ID: %d\n", senderID);
}
```

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
{
   int timestamp;
   char eventType[20];
   char description[100];
   struct LogEntry *next;
}*front = NULL, *rear = NULL;

void enqueue(int timestamp, const char *eventType, const char *description);
void dequeue();
void display();
void searchByEventType(const char *eventType);

int main()
{
   enqueue(11, "INFO", "XYZ completed");
   enqueue(12, "ERROR", "ABC detected");
   printf("Log Queue:\n");
   display();
   printf("\nDequeuing the oldest log:\n");
   dequeue();
```

```
    display();
    printf("\nSearching for logs with event type:\n");
    searchByEventType("ERROR");
    return 0;
}

void enqueue(int timestamp, const char *eventType, const char *description)
{
    struct LogEntry *newEntry = (struct LogEntry *)malloc
                                            (sizeof(struct LogEntry));
    if (newEntry == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newEntry->timestamp = timestamp;
    strcpy(newEntry->eventType, eventType);
    strcpy(newEntry->description, description);
    newEntry->next = NULL;
    if (rear == NULL)
        front = rear = newEntry;
    else
    {
        rear->next = newEntry;
        rear = newEntry;
    }
    printf("Enqueued Log: Timestamp = %d, Event Type = %s,
```

```
            Description = \"%s\"\n", timestamp, eventType, description);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no log to dequeue\n");
        return;
    }
    struct LogEntry *t = front;
    printf("Dequeued Log: Timestamp = %d, Event Type = %s,
     Description = \"%s\"\n",t->timestamp, t->eventType, t->description);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct LogEntry *current = front;
```

```
    while (current != NULL)
    {
        printf("Timestamp = %d, Event Type = %s, Description = \"%s\"\n",
               current->timestamp, current->eventType, current->description);
        current = current->next;
    }
}

void searchByEventType(const char *eventType)
{
    struct LogEntry *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strcmp(current->eventType, eventType) == 0)
        {
            printf("Found Log: Timestamp = %d, Event Type = %s, Description =
     \"%s\"\n",current->timestamp, current->eventType, current->description);
            found = 1;
        }
        current = current->next;
    }
    if (!found)
        printf("No logs found with Event Type: %s\n", eventType);
}
```

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[20];
    char destIP[20];
    char payload[100];
    struct Packet *next;
} *front = NULL, *rear = NULL;

void enqueue(const char *sourceIP, const char *destIP, const char *payload);
void dequeue();
void display();
void searchByIP(const char *ipAddress);

int main()
{
    enqueue("192.168.0.1", "192.168.0.2", "Packet 1 Payload");
    enqueue("192.168.0.3", "192.168.0.4", "Packet 2 Payload");
    enqueue("192.168.0.1", "192.168.0.5", "Packet 3 Payload");
```

```
    printf("Packet Queue:\n");
    display();
    printf("\nDequeuing the oldest packet:\n");
    dequeue();
    display();
    printf("\nSearching for packets with source or destination IP:\n");
    searchByIP("192.168.0.1");
    return 0;
}

void enqueue(const char *sourceIP, const char *destIP, const char *payload)
{
    struct Packet *newPacket = (struct Packet *)malloc(sizeof(struct Packet));
    if (newPacket == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    strcpy(newPacket->sourceIP, sourceIP);
    strcpy(newPacket->destIP, destIP);
    strcpy(newPacket->payload, payload);
    newPacket->next = NULL;
    if (rear == NULL)
        front = rear = newPacket;
    else
    {
        rear->next = newPacket;
```

```
        rear = newPacket;
    }
    printf("Enqueued Packet: Source IP = %s, Destination IP = %s,
                                Payload = \"%s\"\n", sourceIP, destIP, payload);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no packet to dequeue\n");
        return;
    }
    struct Packet *t = front;
    printf("Dequeued Packet: Source IP = %s, Destination IP = %s,
                        Payload = \"%s\"\n", t->sourceIP, t->destIP, t->payload);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
```

```
        return;
    }
    struct Packet *current = front;
    while (current != NULL)
    {
        printf("Source IP = %s, Destination IP = %s, Payload = \"%s\"\n",
               current->sourceIP, current->destIP, current->payload);
        current = current->next;
    }
}

void searchByIP(const char *ipAddress)
{
    struct Packet *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strcmp(current->sourceIP, ipAddress) == 0 || strcmp(current->destIP,
                                                                ipAddress) == 0)
        {
            printf("Found Packet: Source IP = %s, Destination IP = %s, Payload =
                \"%s\"\n", current->sourceIP, current->destIP, current->payload);
            found = 1;
        }
        current = current->next;
    }
    if (!found)
```

```
        printf("No packets found with IP address: %s\n", ipAddress);
}
```

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[20];
   char releaseNotes[100];
   char filePath[100];
   struct FirmwareUpdate *next;
}*front = NULL, *rear = NULL;

void enqueue(const char *version, const char *releaseNotes, const char *filePath);
void dequeue();
void display();
void searchByVersion(const char *version);

int main()
```

```
{
    enqueue("v1.0.0", "Initial release", "/path/to/firmware_v1.0.0.bin");
    enqueue("v1.1.0", "Bug fixes and performance improvements",
                                                "/path/to/firmware_v1.1.0.bin");
    enqueue("v1.2.0", "Security patch", "/path/to/firmware_v1.2.0.bin");
    printf("Firmware Update Queue:\n");
    display();
    printf("\nDequeuing the oldest firmware update for installation:\n");
    dequeue();
    display();
    printf("\nSearching for firmware update with version:\n");
    searchByVersion("v1.1.0");
    return 0;
}

void enqueue(const char *version, const char *releaseNotes, const char
*filePath)
{
    struct FirmwareUpdate *newUpdate = (struct FirmwareUpdate*)
                                    malloc(sizeof(struct FirmwareUpdate));
    if (newUpdate == NULL)
    {
        printf("Memory allocation failed!\n");
        return;
    }
    strcpy(newUpdate->version, version);
    strcpy(newUpdate->releaseNotes, releaseNotes);
    strcpy(newUpdate->filePath, filePath);
```

```
    newUpdate->next = NULL;
    if (rear == NULL)
        front = rear = newUpdate;
    else
    {
        rear->next = newUpdate;
        rear = newUpdate;
    }
    printf("Enqueued Firmware Update: Version = %s, Release Notes =
                    \"%s\", File Path = %s\n",version, releaseNotes, filePath);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no firmware update to dequeue\n");
        return;
    }
    struct FirmwareUpdate *t = front;
    printf("Dequeued Firmware Update: Version = %s, Release Notes =
            \"%s\", File Path = %s\n", t->version, t->releaseNotes, t->filePath);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}
```

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

void searchByVersion(const char *version)
{
    struct FirmwareUpdate *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strcmp(current->version, version) == 0)
        {
            printf("Found Firmware Update: Version = %s, Release Notes =
                \"%s\", File Path = %s\n", current->version, current->releaseNotes,
                                current->filePath);
```

```
            found = 1;
        }
        current = current->next;
    }
    if (!found)
        printf("No firmware update found with version: %s\n", version);
}
```

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 type[50];
    char timestamp[30];
    char action[100];
    struct PowerEvent *next;
}*front = NULL, *rear = NULL;

void enqueue(const char *type, const char *timestamp, const char *action);
```

```
void dequeue();
void display();
void searchByType(const char *type);

int main()
{
    enqueue("Power On", "2025-01-20 08:00:00", "Initialize all systems");
    enqueue("Sleep", "2025-01-20 22:00:00", "Suspend non-critical systems");
    enqueue("Power Off", "2025-01-21 00:00:00", "Shut down all systems
                                                        safely");
    printf("Power Management Event Queue:\n");
    display();
    printf("\nDequeuing the oldest power event:\n");
    dequeue();
    display();
    printf("\nSearching for events with type 'Sleep':\n");
    searchByType("Sleep");
    return 0;
}

void enqueue(const char *type, const char *timestamp, const char *action)
{
    struct PowerEvent *newEvent = (struct PowerEvent *)malloc
                                            (sizeof(struct PowerEvent));
    if (newEvent == NULL)
    {
        printf("Memory allocation failed\n");
```

```
        return;
    }
    strcpy(newEvent->type, type);
    strcpy(newEvent->timestamp, timestamp);
    strcpy(newEvent->action, action);
    newEvent->next = NULL;
    if (rear == NULL)
        front = rear = newEvent;
    else
    {
        rear->next = newEvent;
        rear = newEvent;
    }
    printf("Enqueued Power Event: Type = %s, Timestamp = %s,
                                    Action = \"%s\"\n", type, timestamp, action);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no event to dequeue\n");
        return;
    }
    struct PowerEvent *t = front;
    printf("Dequeued Power Event: Type = %s, Timestamp = %s,
                        Action = \"%s\"\n", t->type, t->timestamp, t->action);
```

```
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct PowerEvent *current = front;
    while (current != NULL)
    {
        printf("Type = %s, Timestamp = %s, Action = \"%s\"\n",
               current->type, current->timestamp, current->action);
        current = current->next;
    }
}

void searchByType(const char *type)
{
    struct PowerEvent *current = front;
    int found = 0;
    while (current != NULL)
```

```
    {
        if (strcmp(current->type, type) == 0)
        {
            printf("Found Power Event: Type = %s, Timestamp = %s, Action =
                \"%s\"\n", current->type, current->timestamp, current->action);
            found = 1;
        }
        current = current->next;
    }
    if (!found)
        printf("No power event found with type: %s\n", type);
}
```

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 id;
    char type[50];
```

```
    char parameters[100];
    struct Command *next;
}*front = NULL, *rear = NULL;

void enqueue(int id, const char *type, const char *parameters);
void dequeue();
void display();
void searchByType(const char *type);

int main()
{
    enqueue(1, "System", "Restart device");
    enqueue(2, "User", "Change display brightness");
    enqueue(3, "System", "Check battery status");
    enqueue(4, "User", "Enable Wi-Fi");
    printf("Command Queue:\n");
    display();
    printf("\nDequeuing the oldest command:\n");
    dequeue();
    display();
    printf("\nSearching for commands of type:\n");
    searchByType("User");
    return 0;
}

void enqueue(int id, const char *type, const char *parameters)
{
```

```
    struct Command *newCommand = (struct Command *)malloc
                                                (sizeof(struct Command));
    if (newCommand == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newCommand->id = id;
    strcpy(newCommand->type, type);
    strcpy(newCommand->parameters, parameters);
    newCommand->next = NULL;
    if (rear == NULL)
        front = rear = newCommand;
    else
    {
        rear->next = newCommand;
        rear = newCommand;
    }
    printf("Enqueued Command: ID = %d, Type = %s, Parameters = \"%s\"\n",
           id, type, parameters);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no command to dequeue\n");
```

```
        return;
    }
    struct Command *t = front;
    printf("Dequeued Command: ID = %d, Type = %s, Parameters = \"%s\"\n",
           t->id, t->type, t->parameters);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct Command *current = front;
    while (current != NULL)
    {
        printf("ID = %d, Type = %s, Parameters = \"%s\"\n",
               current->id, current->type, current->parameters);
        current = current->next;
    }
}
```

```
void searchByType(const char *type)
{
   struct Command *current = front;
   int found = 0;
   while (current != NULL)
   {
      if (strcmp(current->type, type) == 0)
      {
        printf("Found Command: ID = %d, Type = %s, Parameters = \"%s\"\n",
              current->id, current->type, current->parameters);
         found = 1;
      }
      current = current->next;
   }
   if (!found)
      printf("No commands found with type: %s\n", type);
}
```

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 <string.h>
```

```
struct AudioSample
{
    unsigned int timestamp;
    short *audioData;
    struct AudioSample *next;
} *front = NULL, *rear = NULL;

void enqueue(unsigned int timestamp, short *audioData);
void dequeue();
void display();
void searchByTimestamp(unsigned int timestamp);

int main()
{
    short audioData1[] = {100, 150, 200, 250};
    short audioData2[] = {300, 350, 400, 450};
    short audioData3[] = {500, 550, 600, 650};
    enqueue(1000, audioData1);
    enqueue(2000, audioData2);
    enqueue(3000, audioData3);
    printf("Audio Buffer Queue:\n");
    display();
    printf("\nDequeuing the oldest audio sample:\n");
    dequeue();
    display();
    printf("\nSearching for sample with timestamp:\n");
    searchByTimestamp(2000);
```

```
    return 0;
}

void enqueue(unsigned int timestamp, short *audioData)
{
    struct AudioSample *newSample = (struct AudioSample *)
                                    malloc(sizeof(struct AudioSample));
    if (newSample == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newSample->timestamp = timestamp;
    newSample->audioData = audioData;
    newSample->next = NULL;
    if (rear == NULL)
        front = rear = newSample;
    else
    {
        rear->next = newSample;
        rear = newSample;
    }
    printf("Enqueued Audio Sample: Timestamp = %u\n", timestamp);
}

void dequeue()
{
```

```
    if (front == NULL)
    {
        printf("Queue is empty, no sample to dequeue\n");
        return;
    }
    struct AudioSample *t = front;
    printf("Dequeued Audio Sample: Timestamp = %u\n", t->timestamp);
    printf("Audio Data: ");
    for (int i = 0; i < 4; i++)
        printf("%d ", t->audioData[i]);
    printf("\n");
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct AudioSample *current = front;
    while (current != NULL)
    {
```

```
        printf("Timestamp = %u, Audio Data = ", current->timestamp);
        for (int i = 0; i < 4; i++)
            printf("%d ", current->audioData[i]);
        printf("\n");
        current = current->next;
    }
}

void searchByTimestamp(unsigned int timestamp)
{
    struct AudioSample *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (current->timestamp == timestamp)
        {
            printf("Found Audio Sample: Timestamp = %u, Audio Data = ",
                                                    current->timestamp);
            for (int i = 0; i < 4; i++)
                printf("%d ", current->audioData[i]);
            printf("\n");
            found = 1;
            break;
        }
        current = current->next;
    }
    if (!found)
```

```
        printf("No audio sample found with timestamp %u\n", timestamp);
}
```

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 id;
   char type[50];
   char data[100];
   struct Event *next;
} *front = NULL, *rear = NULL;

void enqueue(int id, const char *type, const char *data);
void dequeue();
void display();
void searchByType(const char *type);
void searchById(int id);
```

```
int main()
{
    enqueue(1, "Sensor", "Temperature: 25°C");
    enqueue(2, "Button", "Pressed");
    printf("Event Queue:\n");
    display();
    printf("\nDequeuing the oldest event:\n");
    dequeue();
    display();
    printf("\nSearching for events of type:\n");
    searchByType("Sensor");
    printf("\nSearching for event with ID:\n");
    searchById(3);
    return 0;
}

void enqueue(int id, const char *type, const char *data)
{
    struct Event *newEvent = (struct Event *)malloc(sizeof(struct Event));
    if (newEvent == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newEvent->id = id;
    strcpy(newEvent->type, type);
    strcpy(newEvent->data, data);
```

```
    newEvent->next = NULL;
    if (rear == NULL)
        front = rear = newEvent;
    else
    {
        rear->next = newEvent;
        rear = newEvent;
    }
    printf("Enqueued Event: ID = %d, Type = %s, Data = \"%s\"\n", id, type,
                                                                    data);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no event to dequeue\n");
        return;
    }
    struct Event *t = front;
    printf("Dequeued Event: ID = %d, Type = %s, Data = \"%s\"\n", t->id,
                                                        t->type, t->data);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}
```

```
void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct Event *current = front;
    while (current != NULL)
    {
       printf("ID = %d, Type = %s, Data = \"%s\"\n", current->id, current->type,
                                                          current->data);
        current = current->next;
    }
}

void searchByType(const char *type)
{
    struct Event *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strcmp(current->type, type) == 0)
        {
            printf("Found Event: ID = %d, Type = %s, Data = \"%s\"\n",
                                    current->id, current->type, current->data);
            found = 1;
```

```
        }
        current = current->next;
    }
    if (!found)
        printf("No events found with type: %s\n", type);
}

void searchById(int id)
{
    struct Event *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (current->id == id)
        {
            printf("Found Event: ID = %d, Type = %s, Data = \"%s\"\n",
                                current->id, current->type, current->data);
            found = 1;
            break;
        }
        current = current->next;
    }
    if (!found)
        printf("No event found with ID: %d\n", id);
}
```

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>
#include <time.h>

struct GUIEvent
{
   char type[50];
   int x, y;
   time_t timestamp;
   struct GUIEvent *next;
} *front = NULL, *rear = NULL;

void enqueue(const char *type, int x, int y);
void dequeue();
void display();
void searchByType(const char *type);
time_t getTimestamp();

int main()
{
   enqueue("Button Click", 100, 150);
```

```
    enqueue("Screen Touch", 200, 250);
    enqueue("Button Click", 300, 350);
    printf("GUI Event Queue:\n");
    display();
    printf("\nDequeuing the oldest event:\n");
    dequeue();
    display();
    printf("\nSearching for events of type:\n");
    searchByType("Button Click");
    return 0;
}

void enqueue(const char *type, int x, int y)
{
    struct GUIEvent *newEvent = (struct GUIEvent *)malloc
                                            (sizeof(struct GUIEvent));
    if (newEvent == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    strcpy(newEvent->type, type);
    newEvent->x = x;
    newEvent->y = y;
    newEvent->timestamp = getTimestamp();
    newEvent->next = NULL;
    if (rear == NULL)
```

```
        front = rear = newEvent;
    else
    {
        rear->next = newEvent;
        rear = newEvent;
    }
    printf("Enqueued Event: Type = %s, Coordinates = (%d, %d),
                    Timestamp = %ld\n", type, x, y, newEvent->timestamp);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no event to dequeue\n");
        return;
    }
    struct GUIEvent *t = front;
    printf("Dequeued Event: Type = %s, Coordinates = (%d, %d),
                    Timestamp = %ld\n", t->type, t->x, t->y, t->timestamp);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t);
}

void display()
```

```
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct GUIEvent *current = front;
    while (current != NULL)
    {
        printf("Type = %s, Coordinates = (%d, %d), Timestamp = %ld\n",
                    current->type, current->x, current->y, current->timestamp);
        current = current->next;
    }
}

void searchByType(const char *type)
{
    struct GUIEvent *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strcmp(current->type, type) == 0)
        {
            printf("Found Event: Type = %s, Coordinates = (%d, %d), Timestamp
          = %ld\n", current->type, current->x, current->y, current->timestamp);
            found = 1;
        }
        current = current->next;
```

```
    }
    if (!found)
        printf("No events found with type: %s\n", type);
}

time_t getTimestamp()
{
    return time(NULL);
}
```

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 DataChunk
{
    char *data;
    int length;
    struct DataChunk *next;
} *front = NULL, *rear = NULL;
```

```
void enqueue(const char *data, int length);
void dequeue();
void display();
void searchForPattern(const char *pattern);

int main()
{
   enqueue("Hello", 5);
   enqueue("World", 5);
   printf("Serial Communication Buffer:\n");
   display();
   printf("\nDequeueing the oldest data chunk:\n");
   dequeue();
   display();
   printf("\nSearching for the pattern:\n");
   searchForPattern("World");
   return 0;
}

void enqueue(const char *data, int length)
{
   struct DataChunk *newChunk = (struct DataChunk *)malloc
                                    (sizeof(struct DataChunk));
   if (newChunk == NULL)
   {
      printf("Memory allocation failed\n");
      return;
```

```
    }
    newChunk->data = (char *)malloc(length + 1);
    if (newChunk->data == NULL) {
        printf("Memory allocation for data failed\n");
        free(newChunk);
        return;
    }
    strncpy(newChunk->data, data, length);
    newChunk->data[length] = '\0';
    newChunk->length = length;
    newChunk->next = NULL;
    if (rear == NULL)
        front = rear = newChunk;
    else
    {
        rear->next = newChunk;
        rear = newChunk;
    }
    printf("Enqueued Data: \"%s\" (Length = %d)\n", data, length);
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no data to dequeue\n");
        return;
```

```
    }
    struct DataChunk *t = front;
    printf("Dequeued Data: \"%s\" (Length = %d)\n", t->data, t->length);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t->data);
    free(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Buffer is empty\n");
        return;
    }
    struct DataChunk *current = front;
    while (current != NULL)
    {
        printf("Data: \"%s\" (Length = %d)\n", current->data, current->length);
        current = current->next;
    }
}

void searchForPattern(const char *pattern)
{
```

```
    struct DataChunk *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (strstr(current->data, pattern) != NULL) {
            printf("Pattern \"%s\" found in Data: \"%s\" (Length = %d)\n",
                                        pattern, current->data, current->length);
            found = 1;
        }
        current = current->next;
    }
    if (!found)
        printf("Pattern \"%s\" not found in any data chunks\n", pattern);
}
```

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 id;
    int dataLength;
    unsigned char *payload;
    struct CANMessage *next;
} *front = NULL, *rear = NULL;

void enqueue(int id, const unsigned char *payload, int dataLength);
void dequeue();
void display();
void searchByID(int id);
void freeMessage(struct CANMessage *msg);

int main()
{
    unsigned char payload1[] = {0x01, 0x02, 0x03, 0x04};
    unsigned char payload2[] = {0x05, 0x06, 0x07};
    unsigned char payload3[] = {0x08, 0x09, 0x0A, 0x0B, 0x0C};
    enqueue(1, payload1, sizeof(payload1));
    enqueue(2, payload2, sizeof(payload2));
    enqueue(3, payload3, sizeof(payload3));
    printf("CAN Bus Message Queue:\n");
    display();
    printf("\nDequeuing the oldest message:\n");
    dequeue();
    display();
    printf("\nSearching for message with ID:\n");
    searchByID(2);
```

```
    return 0;
}

void enqueue(int id, const unsigned char *payload, int dataLength)
{
    struct CANMessage *newMessage = (struct CANMessage *)
                                    malloc(sizeof(struct CANMessage));
    if (newMessage == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newMessage->payload = (unsigned char *)malloc(dataLength);
    if (newMessage->payload == NULL)
    {
        printf("Memory allocation for payload failed\n");
        free(newMessage);
        return;
    }
    memcpy(newMessage->payload, payload, dataLength);
    newMessage->id = id;
    newMessage->dataLength = dataLength;
    newMessage->next = NULL;
    if (rear == NULL)
        front = rear = newMessage;
    else
    {
```

```
        rear->next = newMessage;
        rear = newMessage;
    }
    printf("Enqueued CAN Message: ID = %d, Data Length = %d,
                                                Payload = {", id, dataLength);
    for (int i = 0; i < dataLength; i++)
    {
        printf("0x%02X", newMessage->payload[i]);
        if (i < dataLength - 1)
            printf(", ");
    }
    printf("}\n");
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no message to dequeue\n");
        return;
    }
    struct CANMessage *t = front;
    printf("Dequeued CAN Message: ID = %d, Data Length = %d,
                                        Payload = {", t->id, t->dataLength);
    for (int i = 0; i < t->dataLength; i++)
    {
        printf("0x%02X", t->payload[i]);
```

```
        if (i < t->dataLength - 1)
            printf(", ");
    }
    printf("}\n");
    front = front->next;
    if (front == NULL)
        rear = NULL;
    freeMessage(t);
}

void display()
{
    if (front == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    struct CANMessage *current = front;
    while (current != NULL)
    {
        printf("ID = %d, Data Length = %d, Payload = {", current->id,
                                                    current->dataLength);
        for (int i = 0; i < current->dataLength; i++)
        {
            printf("0x%02X", current->payload[i]);
            if (i < current->dataLength - 1)
                printf(", ");
```

```
        }
        printf("}\n");
        current = current->next;
    }
}

void searchByID(int id)
{
    struct CANMessage *current = front;
    int found = 0;
    while (current != NULL)
    {
        if (current->id == id)
        {
            printf("Found CAN Message: ID = %d, Data Length = %d,
                                Payload = {", current->id, current->dataLength);
            for (int i = 0; i < current->dataLength; i++)
            {
                printf("0x%02X", current->payload[i]);
                if (i < current->dataLength - 1)
                    printf(", ");
            }
            printf("}\n");
            found = 1;
            break;
        }
        current = current->next;
```

```
    }
    if (!found)
        printf("No CAN message found with ID: %d\n", id);
}

void freeMessage(struct CANMessage *msg)
{
    if (msg)
    {
        free(msg->payload);
        free(msg);
    }
}
```

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 MLInputData
{
    int id;
    float *features;
    int featureCount;
    struct MLInputData *next;
} *front = NULL, *rear = NULL;

void enqueue(int id, float *features, int featureCount);
void dequeue();
void display();
void searchByMetadata(int id);

int main()
{
    float features1[] = {1.2f, 3.4f, 5.6f};
    float features2[] = {2.1f, 4.3f, 6.5f};
    float features3[] = {7.8f, 9.1f, 0.4f};
    enqueue(1, features1, 3);
    enqueue(2, features2, 3);
    enqueue(3, features3, 3);
    printf("Queue of Machine Learning Inference Data:\n");
    display();
    printf("\nDequeuing the oldest data entry:\n");
    dequeue();
```

```
    display();
    printf("\nSearching for data entry with ID:\n");
    searchByMetadata(2);
    return 0;
}

void enqueue(int id, float *features, int featureCount)
{
    struct MLInputData *newData = (struct MLInputData *)
                                        malloc(sizeof(struct MLInputData));
    if (newData == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }
    newData->features = (float *)malloc(sizeof(float) * featureCount);
    if (newData->features == NULL)
    {
        printf("Memory allocation for features failed\n");
        free(newData);
        return;
    }
    for (int i = 0; i < featureCount; i++)
        newData->features[i] = features[i];
    newData->id = id;
    newData->featureCount = featureCount;
    newData->next = NULL;
```

```
    if (rear == NULL)
        front = rear = newData;
    else
    {
        rear->next = newData;
        rear = newData;
    }
    printf("Enqueued Data: ID = %d, Features = {", id);
    for (int i = 0; i < featureCount; i++)
    {
        printf("%.2f", newData->features[i]);
        if (i < featureCount - 1)
            printf(", ");
    }
    printf("}\n");
}

void dequeue()
{
    if (front == NULL)
    {
        printf("Queue is empty, no data to dequeue\n");
        return;
    }
    struct MLInputData *t = front;
    printf("Dequeued Data: ID = %d, Features = {", t->id);
    for (int i = 0; i < t->featureCount; i++)
```

```
    {
        printf("%.2f", t->features[i]);
        if (i < t->featureCount - 1)
            printf(", ");
    }
    printf("}\n");
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(t->features);
    free(t);
}

void display() {
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct MLInputData *current = front;
    while (current != NULL) {
        printf("ID = %d, Features = {", current->id);
        for (int i = 0; i < current->featureCount; i++)
        {
            printf("%.2f", current->features[i]);
            if (i < current->featureCount - 1)
                printf(", ");
        }
```

```
        printf("}\n");
        current = current->next;
    }
}

void searchByMetadata(int id)
{
    struct MLInputData *current = front;
    int found = 0;
    while (current != NULL) {
        if (current->id == id) {
            printf("Found Data with ID = %d, Features = {", current->id);
            for (int i = 0; i < current->featureCount; i++)
            {
                printf("%.2f", current->features[i]);
                if (i < current->featureCount - 1)
                    printf(", ");
            }
            printf("}\n");
            found = 1;
            break;
        }
        current = current->next;
    }
    if (!found)
        printf("No data found with ID = %d\n", id);
}
```