/**********
****

Question 1: Write a C program that declares a static variable and a const variable within a function.

The program should increment the static variable each time the function is called and use a

switch case to check the value of the const variable. The function should handle at least three

different cases for the const variable and demonstrate the persistence of the static variable

across multiple calls.

****
****/

```
#include<stdio.h>

void fun();
int main(){



   for(int i=0;i<5;i++){
      fun();


   }



   return 0;
}

void fun(){
```

```
    static int a = 0;
    a++;
    const int b = 2;

    printf("static value = %d\n",a);

     switch(b){
        case 1:
        printf("Const variable is 1 \n");
        break;
        case 2 :
        printf("const variable is 2\n");
        break;
        case 3:
        printf("const variable is 3\n");
        break;
        default:
        printf(" invalid");
        break;


    }

}
```

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

*************************************************************************
****/



#include<stdio.h>
const int b = 3;

void fun();

int main(){

   for (int i=0;i<5;i++){

      fun();
   }

   return 0;
}
```

```
void fun(){

   static int a = 0;
   a++;



   if(a>b){

      printf("Max limit reached\n");
      return;
   }

   switch(a){

      case 1:
      printf("function called first time\n");
      break;
      case 2:
      printf("function is called second time\n");
      break;
      case 3:
      printf("the function is called more than two times\n");
      break;
      default:
      break;
   }

}
```

```
/**********************************************************************
****

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

**********************************************************************
****/

#include<stdio.h>

const int MAX = 5;

void fun(int operation , int value);

int main(){

   int choice,value;
   while(1){
   printf("1.Add value to elements\n2.subtract value to elements\n3.multiply values to
elements\n4.Exit\n");
   scanf("%d",&choice);

   if(choice ==4){
      printf("Exiting the program");
```

```
        break;
    }

    printf("Enter the value: ");
    scanf("%d",&value);

    fun(choice , value);
    printf("After operation performed\n");
    fun(0,0);
    }

    return 0;
}

void fun(int operation, int value){

    static int arr[5] = {0};

    switch(operation){

        case 1:
        for(int i=0;i<MAX;i++){
            arr[i]+=value;
        }
            printf("added %d to all elements",value);
        break;
        case 2:
        for(int i=0;i<MAX;i++){
```

```
        arr[i]-=value;
    }
        printf("subtracted %d to all elements",value);
    break;
    case 3:
    for(int i=0;i<MAX;i++){
        arr[i]*=value;
    }
        printf("multiplied %d to all elements",value);
    break;

    case 0:
    for(int i=0;i<MAX;i++){
        printf("%d ",arr[i]);
    }

    default:
    printf("invalid operation");


  }

}
```

```
/*************************************************************************
****

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

*************************************************************************
****/


#include<stdio.h>

const int MAX = 5;

void checkThreshold();

int main(){


for (int i=0;i<MAX;i++){

   checkThreshold();
}
```

```
return 0;
}

void checkThreshold(){

   static int a =0;
   a++;
   switch(a){
      case 1:
      printf("First case , static value = %d\n",a);
      break;
      case 2:
      printf("second case , static value = %d\n",a);
      break;
      case 3:
      printf("third case , static value = %d\n",a);
      break;
      case 4:
      printf("fourth case , static value = %d\n",a);
      break;
      case 5:
      printf("Threshold value reached = %d\n",a);
      break;

      default:
      printf("value reached greater than thresholld \n");
      break;
```

```
    }


}


/**********************************************************************
****

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

**********************************************************************
****/


#include<stdio.h>

const int MAX = 5;

void checkThreshold();

int main(){
```

```
for (int i=0;i<10;i++){

   checkThreshold();
}

return 0;
}

void checkThreshold(){

   static int a =0;
   a++;
   switch(a){
      case 1:
      printf("First case , counter = %d\n",a);
      break;
      case 2:
      printf("second case , counter = %d\n",a);
      break;
      case 3:
      printf("third case , counter = %d\n",a);
      break;
      case 4:
      printf("fourth case , counter = %d\n",a);
      break;
      case 5:
      printf("Fifith call  counter= %d\n",a);
```

```
        break;


        default:
        printf(" counter = %d\n",a);
        break;


    }
    if(a>=MAX){
        printf("maximum limit reached,resetting the counter \n");
        a = 0;
    }


}
```

Looping Statements, Pointers, Const with Pointers, Functions

Question 6: 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 *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
    }
}

void displayArray(const int *arr, int size) {
    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void modifyArray(int **arr, int size) {
    for (int i = 0; i < size; i++) {
        (*arr)[i] *= 2;
    }
}
```

```
int main() {
    int arr[5];
    int *ptr = arr;
    int **doublePtr = &ptr;


    initializeArray(ptr, 5);


    displayArray(ptr, 5);


    modifyArray(doublePtr, 5);


    displayArray(ptr, 5);

    return 0;
}
```

/**
****

Question 7: Develop a simple c 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 rows, int cols, int matrix[rows][cols]);
void transposeMatrix(const int * const *matrix, int rows, int cols, int **transposed);
void printMatrix(int rows, int cols, int matrix[rows][cols]);

int main() {
    int rows, cols;

    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &cols);
```

```
    int (*matrix)[cols] = malloc(rows * cols * sizeof(int));
    int (*transposed)[rows] = malloc(cols * rows * sizeof(int));

    printf("Enter the elements of the matrix:\n");
    readMatrix(rows, cols, matrix);

    transposeMatrix((const int * const *)matrix, rows, cols, (int **)transposed);

    printf("\nOriginal Matrix:\n");
    printMatrix(rows, cols, matrix);

    printf("\nTransposed Matrix:\n");
    printMatrix(cols, rows, transposed);

    free(matrix);
    free(transposed);

    return 0;
}

void readMatrix(int rows, int cols, int matrix[rows][cols]) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
```

```
            printf("Element [%d][%d]: ", i, j);
            scanf("%d", &matrix[i][j]);
        }
    }
}


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


void printMatrix(int rows, int cols, int matrix[rows][cols]) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

```
/*************************************************************************
****
Question 8: 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 *arr,int size);
void printArray(const int *arr,int size);

int main(){

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



   int *array = (int *)malloc(size*sizeof(int));
   if(array == NULL){
      printf("memmmroy allocation failed\n");
      return 1;
   }
```

```
    initializeArray(array,size);
    printArray(array,size);

    free(array);

    return 0;
}

void initializeArray(int *arr,int size){

    int i=0;
    while(i<size){
        arr[i]=i+1;
        i++;
    }

}

void printArray(const int *arr,int size){

    int i=0;
    while(i<size){
        printf("%d ",arr[i]);
        i++;
    }
    printf("\n");
}
```

```
/*************************************************************************
****
Question 9: 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 initializeArray(int *arr, int size);
void printArray(const int *arr, int size);
void swapArrays(int **arr1, int **arr2);
void swapArraysByValue(int *arr1[], int *arr2[], int size);

int main() {
    int size;

    printf("Enter the size of the arrays: ");
    scanf("%d", &size);



    int *array1 = (int *)malloc(size * sizeof(int));
    int *array2 = (int *)malloc(size * sizeof(int));
```

```
if (array1 == NULL || array2 == NULL) {
    printf("Memory allocation failed\n");
    return 1;
}


printf("Initializing Array 1:\n");
initializeArray(array1, size);

printf("Initializing Array 2:\n");
initializeArray(array2, size);


printf("Array 1 before swap:\n");
printArray(array1, size);

printf("Array 2 before swap:\n");
printArray(array2, size);

// Swap the arrays using double pointers
swapArrays(&array1, &array2);


printf("Array 1 after swap by reference:\n");
printArray(array1, size);

printf("Array 2 after swap by reference:\n");
```

```
    printArray(array2, size);

    // Swap the arrays back using single pointers
    swapArraysByValue(&array1, &array2, size);

    // Print the arrays after swapping back
    printf("Array 1 after swap back by value:\n");
    printArray(array1, size);

    printf("Array 2 after swap back by value:\n");
    printArray(array2, size);



    free(array1);
    free(array2);

    return 0;
}

// Function to initialize the array
void initializeArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
}

// Function to print the array using a const pointer to prevent modification
```

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

// Function to swap two arrays using double pointers (call by reference)
void swapArrays(int **arr1, int **arr2) {
    int *temp = *arr1;
    *arr1 = *arr2;
    *arr2 = temp;
}

// Function to swap two arrays using single pointers (call by value)
void swapArraysByValue(int *arr1[], int *arr2[], int size) {
    for (int i = 0; i < size; i++) {
        int temp = (*arr1)[i];
        (*arr1)[i] = (*arr2)[i];
        (*arr2)[i] = temp;
    }
}
```

```
/**********************************************************************
****
Question 10: 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 <string.h>



#define MAX_LENGTH 100



void readString(char *str, int maxLength);
void countFrequency(const char *str);

int main() {
   char str[MAX_LENGTH];



   readString(str, MAX_LENGTH);
```

```
    countFrequency(str);

    return 0;
}

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

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

    do {
        frequency[(unsigned char)str[i]]++;
        i++;
    } while (str[i] != '\0');

    printf("Character frequencies:\n");
    for (i = 0; i < 256; i++) {
        if (frequency[i] > 0) {
            printf("'%c': %d\n", i, frequency[i]);
```

```
      }
   }
}
```

```
/*************************************************************************
****
Arrays, Structures, Nested Structures, Unions, Nested Unions,
Strings, Typedef
Question 11: 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 <string.h>


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

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

void enterEmployeeData(Employee *emp, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nEnter details for employee %d:\n", i + 1);

        printf("Enter ID: ");
        scanf("%d", &emp[i].id);

        printf("Enter Name: ");
        scanf(" %[^\n]", emp[i].name);

        printf("Enter Salary: ");
        scanf("%f", &emp[i].salary);

        printf("Enter Address (Street): ");
        scanf(" %[^\n]", emp[i].address.street);

        printf("Enter Address (City): ");
        scanf(" %[^\n]", emp[i].address.city);
```

```
        printf("Enter Address (State): ");
        scanf(" %[^\n]", emp[i].address.state);

        printf("Enter Address (ZIP): ");
        scanf("%d", &emp[i].address.zip);
    }
}

void displayEmployeeData(const Employee *emp, int count) {
    printf("\nEmployee Details:\n");
    for (int i = 0; i < count; i++) {
        printf("\nEmployee %d:\n", i + 1);
        printf("ID: %d\n", emp[i].id);
        printf("Name: %s\n", emp[i].name);
        printf("Salary: %.2f\n", emp[i].salary);
        printf("Address: %s, %s, %s, %d\n", emp[i].address.street, emp[i].address.city,
emp[i].address.state, emp[i].address.zip);
    }
}

int main() {
    int n;

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

```
    Employee employees[n];



    enterEmployeeData(employees, n);

    displayEmployeeData(employees, n);

    return 0;
}



/**************************************************************************
****
Question 12: 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 struct {
    char type;
```

```
    union {
        int intValue;
        float floatValue;
        char stringValue[50];
    } data;
} DataUnion;


void enterData(DataUnion *data, int count);
void displayData(const DataUnion *data, int count);

int main() {
    int count;


    printf("Enter the number of data entries: ");
    scanf("%d", &count);


    DataUnion dataArray[count];


    enterData(dataArray, count);


    displayData(dataArray, count);

    return 0;
```

```
}



void enterData(DataUnion *data, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nEnter type for data %d (i: int, f: float, s: string): ", i + 1);
        scanf(" %c", &data[i].type);

        switch (data[i].type) {
            case 'i':
                printf("Enter integer value: ");
                scanf("%d", &data[i].data.intValue);
                break;
            case 'f':
                printf("Enter float value: ");
                scanf("%f", &data[i].data.floatValue);
                break;
            case 's':
                printf("Enter string value: ");
                scanf(" %[^\n]s", data[i].data.stringValue);
                break;
            default:
                printf("Invalid type!\n");
                i--;
                break;
        }
    }
}
```

```
void displayData(const DataUnion *data, int count) {
    printf("\nData Information:\n");
    for (int i = 0; i < count; i++) {
        printf("\nData %d:\n", i + 1);
        switch (data[i].type) {
            case 'i':
                printf("Type: Integer\n");
                printf("Value: %d\n", data[i].data.intValue);
                break;
            case 'f':
                printf("Type: Float\n");
                printf("Value: %f\n", data[i].data.floatValue);
                break;
            case 's':
                printf("Type: String\n");
                printf("Value: %s\n", data[i].data.stringValue);
                break;
            default:
                printf("Unknown type!\n");
                break;
        }
    }
}
```

```
/*************************************************************************
****
Question 13: Write a C program that uses an array of strings to
store names. Implement a structure containing a nested union to
store either the length of the string or the reversed string.
Use typedef to simplify the structure definition and display
the stored information.
*************************************************************************
****/

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



typedef struct {
   char name[50];
   union {
      int length;
      char reversed[50];
   } info;
   char infoType;
} NameInfo;



void enterNames(NameInfo *names, int count);
void processNames(NameInfo *names, int count);
void displayNames(const NameInfo *names, int count);
void reverseString(const char *src, char *dest);
```

```
int main() {
    int count;


    printf("Enter the number of names: ");
    scanf("%d", &count);


    NameInfo names[count];

    enterNames(names, count);


    processNames(names, count);


    displayNames(names, count);

    return 0;
}


void enterNames(NameInfo *names, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nEnter name %d: ", i + 1);
        scanf(" %[^\n]s", names[i].name);
    }
}
```

```
void processNames(NameInfo *names, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nProcess name %d (L for length, R for reversed): ", i + 1);
        scanf(" %c", &names[i].infoType);
        if (names[i].infoType == 'L') {
            names[i].info.length = strlen(names[i].name);
        } else if (names[i].infoType == 'R') {
            reverseString(names[i].name, names[i].info.reversed);
        } else {
            printf("Invalid option! Defaulting to length.\n");
            names[i].info.length = strlen(names[i].name);
            names[i].infoType = 'L';
        }
    }
}

void displayNames(const NameInfo *names, int count) {
    printf("\nNames and Information:\n");
    for (int i = 0; i < count; i++) {
        printf("\nName %d: %s\n", i + 1, names[i].name);
        if (names[i].infoType == 'L') {
            printf("Length: %d\n", names[i].info.length);
        } else if (names[i].infoType == 'R') {
            printf("Reversed: %s\n", names[i].info.reversed);
        } else {
            printf("Unknown information type!\n");
```

```
        }
    }
}



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


/**********************************************************************
****
Question 14: Develop a program that demonstrates the use of nested structures and unions.
Create a structure that contains a union, and within the union, define another structure. Use
an array to manage multiple instances of this complex structure and typedef to define the
structure.
**********************************************************************
****/
#include <stdio.h>



typedef struct {
    int day;
    int month;
    int year;
} Date;
```

```
typedef struct {
    char name[50];
    union {
        Date birthDate;
        struct {
            float salary;
            char department[50];
        } employmentDetails;
    } info;
    char infoType;
} Employee;


void enterEmployeeInfo(Employee *employees, int count);
void displayEmployeeInfo(const Employee *employees, int count);

int main() {
    int count;


    printf("Enter the number of employees: ");
    scanf("%d", &count);


    Employee employees[count];
```

```
    enterEmployeeInfo(employees, count);



    displayEmployeeInfo(employees, count);


    return 0;
}



void enterEmployeeInfo(Employee *employees, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nEnter information for employee %d:\n", i + 1);
        printf("Name: ");
        scanf(" %[^\n]s", employees[i].name);
        printf("Enter 'B' for birthDate or 'E' for employmentDetails: ");
        scanf(" %c", &employees[i].infoType);


        if (employees[i].infoType == 'B') {
            printf("Enter birth date (day month year): ");
            scanf("%d %d %d", &employees[i].info.birthDate.day,
&employees[i].info.birthDate.month, &employees[i].info.birthDate.year);
        } else if (employees[i].infoType == 'E') {
            printf("Enter salary: ");
            scanf("%f", &employees[i].info.employmentDetails.salary);
            printf("Enter department: ");
            scanf(" %[^\n]s", employees[i].info.employmentDetails.department);
        } else {
```

```
            printf("Invalid option! Defaulting to birthDate.\n");
            employees[i].infoType = 'B';
            printf("Enter birth date (day month year): ");
            scanf("%d %d %d", &employees[i].info.birthDate.day,
&employees[i].info.birthDate.month, &employees[i].info.birthDate.year);
        }
    }
}


void displayEmployeeInfo(const Employee *employees, int count) {
    printf("\nEmployee Information:\n");
    for (int i = 0; i < count; i++) {
        printf("\nEmployee %d:\n", i + 1);
        printf("Name: %s\n", employees[i].name);
        if (employees[i].infoType == 'B') {
            printf("Birth Date: %d-%d-%d\n", employees[i].info.birthDate.day,
employees[i].info.birthDate.month, employees[i].info.birthDate.year);
        } else if (employees[i].infoType == 'E') {
            printf("Salary: %.2f\n", employees[i].info.employmentDetails.salary);
            printf("Department: %s\n", employees[i].info.employmentDetails.department);
        } else {
            printf("Unknown information type!\n");
        }
    }
}
```

```c
/************************************************************************
****
Question 15: 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[50];
    char nationality[30];
} Author;


typedef struct {
    char title[100];
    Author author;
    union {
        int pages;
        int publicationYear;
    } details;
    char detailsType;
```

```
} Book;



void inputBookInfo(Book *books, int count);
void displayBookInfo(const Book *books, int count);

int main() {
    int count;



    printf("Enter the number of books: ");
    scanf("%d", &count);



    Book books[count];

    inputBookInfo(books, count);



    displayBookInfo(books, count);

    return 0;
}



void inputBookInfo(Book *books, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nEnter information for book %d:\n", i + 1);
```

```
        printf("Title: ");
        scanf(" %[^\n]s", books[i].title);
        printf("Author Name: ");
        scanf(" %[^\n]s", books[i].author.name);
        printf("Author Nationality: ");
        scanf(" %[^\n]s", books[i].author.nationality);
        printf("Enter 'P' for number of pages or 'Y' for publication year: ");
        scanf(" %c", &books[i].detailsType);

        if (books[i].detailsType == 'P') {
            printf("Number of pages: ");
            scanf("%d", &books[i].details.pages);
        } else if (books[i].detailsType == 'Y') {
            printf("Publication year: ");
            scanf("%d", &books[i].details.publicationYear);
        } else {
            printf("Invalid option! Defaulting to number of pages.\n");
            books[i].detailsType = 'P';
            printf("Number of pages: ");
            scanf("%d", &books[i].details.pages);
        }
    }
}

void displayBookInfo(const Book *books, int count) {
    printf("\nBook Information:\n");
    for (int i = 0; i < count; i++) {
```

```
        printf("\nBook %d:\n", i + 1);
        printf("Title: %s\n", books[i].title);
        printf("Author: %s\n", books[i].author.name);
        printf("Nationality: %s\n", books[i].author.nationality);
        if (books[i].detailsType == 'P') {
            printf("Number of Pages: %d\n", books[i].details.pages);
        } else if (books[i].detailsType == 'Y') {
            printf("Publication Year: %d\n", books[i].details.publicationYear);
        } else {
            printf("Unknown details type!\n");
        }
    }
}
```

/**
****

Question 16: 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 items[MAX];
   int top;
} Stack;


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

```
void displayStack(Stack *s);

int main() {
    Stack stack;
    initializeStack(&stack);


    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    push(&stack, 40);
    push(&stack, 50);


    printf("Stack after pushing 5 elements:\n");
    displayStack(&stack);


    printf("Popped element: %d\n", pop(&stack));


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


    printf("Stack after popping an element:\n");
    displayStack(&stack);

    return 0;
```

```
}

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

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

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

void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack is full. Cannot push %d\n", value);
        return;
    }
    s->items[++s->top] = value;
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot pop\n");
        return -1;
```

```
    }
    return s->items[s->top--];
}


int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot peek\n");
        return -1;
    }
    return s->items[s->top];
}


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

```
/*************************************************************************
****
```

Question 17: 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;



Node* createNode(int data);
void push(Node **top, int data);
int pop(Node **top);
int peek(Node *top);
int isEmpty(Node *top);
void displayStack(Node *top);

int main() {
    Node *top = NULL;
```

```
    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    push(&top, 40);
    push(&top, 50);

    printf("Stack after pushing 5 elements:\n");
    displayStack(top);

    printf("Popped element: %d\n", pop(&top));

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

    printf("Stack after popping an element:\n");
    displayStack(top);

    return 0;
}

// Function to create a new node
Node* createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
```

```
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to push an element
void push(Node **top, int data) {
    Node *newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
}

// Function to pop an element
int pop(Node **top) {
    if (isEmpty(*top)) {
        printf("Stack is empty. Cannot pop\n");
        return -1;
    }
    Node *temp = *top;
    int poppedData = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedData;
}
```

```
int peek(Node *top) {
    if (isEmpty(top)) {
        printf("Stack is empty. Cannot peek\n");
        return -1;
    }
    return top->data;
}

// Function to check if the stack is empty
int isEmpty(Node *top) {
    return top == NULL;
}

// Function to display the stack
void displayStack(Node *top) {
    if (isEmpty(top)) {
        printf("Stack is empty\n");
        return;
    }
    Node *current = top;
    while (current) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

/**
****

Question 18: 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>
#include <stdlib.h>

#define MAX 5


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


void initializeStack(Stack *s);
int isFull(Stack *s);
int isEmpty(Stack *s);
void push(Stack *s, int value);
int pop(Stack *s);
int peek(Stack *s);
void displayStack(Stack *s);
```

```
void reverseStack(Stack *s);

int main() {
    Stack stack;
    initializeStack(&stack);


    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    push(&stack, 40);
    push(&stack, 50);


    printf("Stack after pushing 5 elements:\n");
    displayStack(&stack);


    reverseStack(&stack);


    printf("Stack after reversal:\n");
    displayStack(&stack);

    return 0;
}
```

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


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


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


void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack is full. Cannot push %d\n", value);
        return;
    }
    s->items[++s->top] = value;
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot pop\n");
        return -1;
    }
```

```
    return s->items[s->top--];
}

int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot peek\n");
        return -1;
    }
    return s->items[s->top];
}



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



void reverseStack(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot reverse\n");
        return;
```

```
    }

    int start = 0;
    int end = s->top;

    while (start < end) {
        int temp = s->items[start];
        s->items[start] = s->items[end];
        s->items[end] = temp;
        start++;
        end--;
    }
}
```

/**
****

Question 19: 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;


Node* createNode(int data);
void push(Node **top, int data);
int pop(Node **top);
int peek(Node *top);
int isEmpty(Node *top);
void displayStack(Node *top);
void mergeStacks(Node **stack1, Node **stack2);

int main() {
```

```
Node *stack1 = NULL;
Node *stack2 = NULL;

push(&stack1, 10);
push(&stack1, 20);
push(&stack1, 30);

push(&stack2, 40);
push(&stack2, 50);
push(&stack2, 60);

printf("First stack:\n");
displayStack(stack1);

printf("Second stack:\n");
displayStack(stack2);

mergeStacks(&stack1, &stack2);

printf("Merged stack:\n");
displayStack(stack1);
```

```
    return 0;
}


Node* createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}


void push(Node **top, int data) {
    Node *newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
}


int pop(Node **top) {
    if (isEmpty(*top)) {
        printf("Stack is empty. Cannot pop\n");
        return -1;
    }
```

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


int peek(Node *top) {
    if (isEmpty(top)) {
        printf("Stack is empty. Cannot peek\n");
        return -1;
    }
    return top->data;
}


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


void displayStack(Node *top) {
    if (isEmpty(top)) {
        printf("Stack is empty\n");
        return;
    }
    Node *current = top;
```

```
    while (current) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

void mergeStacks(Node **stack1, Node **stack2) {
    if (isEmpty(*stack1)) {
        *stack1 = *stack2;
    } else {
        Node *temp = *stack1;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = *stack2;
    }
    *stack2 = NULL;
}
```

/**
****

Question 20: 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 expression

****
****/

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

#define MAX 100

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

void initializeStack(Stack *s);
int isFull(Stack *s);
int isEmpty(Stack *s);
void push(Stack *s, char value);
char pop(Stack *s);
char peek(Stack *s);
```

```
void displayStack(Stack *s);
int isBalancedParentheses(const char *expression);

int main() {
    const char *expression = "(a+b)*(c-(d/e))";

    if (isBalancedParentheses(expression)) {
        printf("The expression \"%s\" has balanced parentheses.\n", expression);
    } else {
        printf("The expression \"%s\" does not have balanced parentheses.\n", expression);
    }

    return 0;
}

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

// Function to check if the stack is full
int isFull(Stack *s) {
    return s->top == MAX - 1;
}

// Function to check if the stack is empty
int isEmpty(Stack *s) {
    return s->top == -1;
```

```
}

// Function to push an element onto the stack
void push(Stack *s, char value) {
    if (isFull(s)) {
        printf("Stack is full. Cannot push %c\n", value);
        return;
    }
    s->items[++s->top] = value;
}

// Function to pop an element from the stack
char pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot pop\n");
        return '\0';
    }
    return s->items[s->top--];
}

// Function to peek the top element of the stack
char peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty. Cannot peek\n");
        return '\0';
    }
    return s->items[s->top];
}
```

```
// Function to display the stack
void displayStack(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        return;
    }
    for (int i = s->top; i >= 0; i--) {
        printf("%c ", s->items[i]);
    }
    printf("\n");
}

// Function to check if the parentheses in an expression are balanced
int isBalancedParentheses(const char *expression) {
    Stack stack;
    initializeStack(&stack);

    for (int i = 0; expression[i] != '\0'; i++) {
        char ch = expression[i];

        if (ch == '(') {
            push(&stack, ch);
        } else if (ch == ')') {
            if (isEmpty(&stack)) {
                return 0;
            }
            pop(&stack);
```

```
    }
  }

  return isEmpty(&stack);
}
```

/**********
****

Question 21: 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>

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

```
Node* createNode(int data);
void push(Node **top, int data);
int pop(Node **top);
int peek(Node *top);
int isEmpty(Node *top);
void displayStack(Node *top);
int evaluatePostfix(const char *expression);

int main() {
   const char *expression = "23*54*+9-";

   printf("The result of the postfix expression \"%s\" is: %d\n", expression,
evaluatePostfix(expression));

   return 0;
}

Node* createNode(int data) {
   Node *newNode = (Node *)malloc(sizeof(Node));
   if (!newNode) {
      printf("Memory allocation failed\n");
      exit(1);
   }
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
```

```
}


void push(Node **top, int data) {
    Node *newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
}

int pop(Node **top) {
    if (isEmpty(*top)) {
        printf("Stack is empty. Cannot pop\n");
        return -1;
    }
    Node *temp = *top;
    int poppedData = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedData;
}


int peek(Node *top) {
    if (isEmpty(top)) {
        printf("Stack is empty. Cannot peek\n");
        return -1;
    }
    return top->data;
```

```
}

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



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

int evaluatePostfix(const char *expression) {
    Node *stack = NULL;
    int i = 0;
    while (expression[i] != '\0') {
        if (expression[i] >= '0' && expression[i] <= '9') {

            push(&stack, expression[i] - '0');
        } else {
```

```
            int operand2 = pop(&stack);
            int operand1 = pop(&stack);
            int result;
            switch (expression[i]) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                default:
                    printf("Invalid operator encountered: %c\n", expression[i]);
                    exit(1);
            }
            push(&stack, result);
        }
        i++;
    }

    return pop(&stack);
}
```

→ QUEUE enqueue and dequeue function

```
// QUEUE

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

struct Queue{
   int size;
   int front;
   int rear;
   int *Q;
};

void create(struct Queue *, int);
void enqueue (struct Queue *, int);
void display(struct Queue);
int dequeue(struct Queue *);


int main(){
   struct Queue q;
   create(&q,5);
   enqueue(&q,7);
   enqueue(&q,8);
   enqueue(&q,9);
   display(q);
   printf("%d ",dequeue(&q));
```

```
    printf("\n");
    display(q);
    return 0;
}

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

void enqueue (struct Queue *q, int x){
    if(q->rear == q->size-1){
        printf("Queue is full");
    }else{
        q->rear++;
        q->Q[q->rear] = x;
    }
}

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

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

```
/**********************************************************************
****

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>
#define MAX 50

typedef struct {
   int id;
   char name[20];
}Student;

typedef struct {

   int size;
   int front;
   int rear;
   Student students[MAX];

}Queue;
```

```
void create(Queue *,int size);
void enqueue(Queue *q,Student s);
void display(Queue);
Student dequeue(Queue *q);



int main(){
    Queue q;
    create(&q,MAX);



    Student s1 = {1,"Abhi"};
    Student s2 = {2,"milan"};
    Student s3 = {3,"Nalim"};

    enqueue(&q,s1);
    enqueue(&q,s2);
    enqueue(&q,s3);
    display(q);

    Student admittedStudent = dequeue(&q);
    printf("Admitted student: ID=%d, Name=%s\n", admittedStudent.id,
admittedStudent.name);




    return 0;
}
```

```
void create(Queue *q,int size){

    q->size = size;
    q->front = q->rear = -1;
    //q->students = (int *)malloc(sizeof(struct Student));
}

void enqueue(Queue *q,Student s){
    if(q->rear == q->size-1){
        printf("Queue i full ");
    }else {

        q->rear++;
        q->students[q->rear] = s;
    }



}

void display(Queue q){
    int i;
    for(i = q.front+1;i<=q.rear;i++){

      printf("ID -> %d, Name -> %s \n",q.students[i].id,q.students[i].name);
    }
    printf("\n");
}
```

```
Student dequeue(Queue *q) {
   Student s = {-1, ""};
   if (q->front == q->rear) {
      printf("Queue is empty. Cannot admit student.\n");
   } else {
      q->front++;
      s = q->students[q->front];
   }
   return s;
}
```

```
/*****************************************************************************
****

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>
```

```
typedef struct {
    int id;
    char name[50];
} Student;


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


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


void initializeQueue(Queue *q);
void addStudent(Queue *q, int id, char *name);
Student borrowBook(Queue *q);
void displayQueue(Queue *q);
int isEmpty(Queue *q);

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

```
    addStudent(&q, 1, "Alice");
    addStudent(&q, 2, "Bob");
    addStudent(&q, 3, "Charlie");

    printf("Current queue of students waiting to borrow books:\n");
    displayQueue(&q);


    Student borrower = borrowBook(&q);
    if (borrower.id != -1) {
        printf("Student borrowing a book: ID=%d, Name=%s\n", borrower.id, borrower.name);
    }


    printf("Queue after borrowing a book:\n");
    displayQueue(&q);

    return 0;
}


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


void addStudent(Queue *q, int id, char *name) {
```

```
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data.id = id;
    strcpy(newNode->data.name, name);
    newNode->next = NULL;

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


Student borrowBook(Queue *q) {
    Student s = {-1, ""};
    if (isEmpty(q)) {
        printf("Queue is empty. No students waiting to borrow a book.\n");
    } else {
        Node *temp = q->front;
        s = temp->data;
        q->front = q->front->next;
        if (q->front == NULL) {
            q->rear = NULL;
        }
        free(temp);
    }
    return s;
```

```
}

void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
    } else {
        Node *current = q->front;
        while (current != NULL) {
            printf("ID=%d, Name=%s -> ", current->data.id, current->data.name);
            current = current->next;
        }
        printf("\n");
    }
}


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

```
/**********************************************************************
****

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 the maximum size of the queue
#define MAX 100

// Define the structure for a student
typedef struct {
    int id;
    char name[50];
} Student;

// Define the structure for the queue
typedef struct {
    int size;
    int front;
```

```
    int rear;
    Student students[MAX];
} Queue;


void createQueue(Queue *q, int size);
void issueToken(Queue *q, int id, char *name);
Student serveStudent(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);


    issueToken(&q, 1, "Alice");
    issueToken(&q, 2, "Bob");
    issueToken(&q, 3, "Charlie");

    printf("Current queue of students waiting for their turn:\n");
    displayQueue(&q);

    Student served = serveStudent(&q);
    if (served.id != -1) {
        printf("Student served: ID=%d, Name=%s\n", served.id, served.name);
    }
```

```
    printf("Queue after serving a student:\n");
    displayQueue(&q);

    return 0;
}

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


void issueToken(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}


Student serveStudent(Queue *q) {
    Student s = {-1, ""};
    if (q->front == q->rear) {
        printf("Queue is empty. No students waiting for their turn.\n");
    } else {
```

```
        q->front++;
        s = q->students[q->front];
    }
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }

}
```

```
/**********************************************************************
****

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 100

typedef struct {
    int id;
    char name[50];
} Student;

typedef struct {
    int size;
    int front;
```

```
    int rear;
    Student students[MAX];
} Queue;


void createQueue(Queue *q, int size);
void addStudent(Queue *q, int id, char *name);
Student helpStudent(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);


    addStudent(&q, 1, "Alice");
    addStudent(&q, 2, "Bob");
    addStudent(&q, 3, "Charlie");


    printf("Current queue of students waiting for help:\n");
    displayQueue(&q);


    Student helped = helpStudent(&q);
    if (helped.id != -1) {
        printf("Student helped: ID=%d, Name=%s\n", helped.id, helped.name);
    }
```

```
    printf("Queue after helping a student:\n");
    displayQueue(&q);

    return 0;
}


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


void addStudent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}


Student helpStudent(Queue *q) {
    Student s = {-1, ""};
```

```
    if (q->front == q->rear) {
        printf("Queue is empty. No students waiting for help.\n");
    } else {
        q->front++;
        s = q->students[q->front];
    }
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }
}
```

```
/*************************************************************************
****

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 100


typedef struct {
    int id;
    char name[50];
} Student;


typedef struct {
    int size;
    int front;
```

```
    int rear;
    Student students[MAX];
} Queue;


void createQueue(Queue *q, int size);
void addStudent(Queue *q, int id, char *name);
Student processRegistration(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);


    addStudent(&q, 1, "Alice");
    addStudent(&q, 2, "Bob");
    addStudent(&q, 3, "Charlie");


    printf("Current queue of students waiting for registration:\n");
    displayQueue(&q);


    Student registered = processRegistration(&q);
    if (registered.id != -1) {
        printf("Student registered: ID=%d, Name=%s\n", registered.id, registered.name);
    }
```

```
    printf("Queue after processing a registration:\n");
    displayQueue(&q);

    return 0;
}


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


void addStudent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}


Student processRegistration(Queue *q) {
    Student s = {-1, ""};
```

```
    if (q->front == q->rear) {
        printf("Queue is empty. No students waiting for registration.\n");
    } else {
        q->front++;
        s = q->students[q->front];
    }
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }
}
```

```
/**********************************************************************
****

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 100


typedef struct {
   int id;
   char name[50];
} Student;


typedef struct {
   int size;
   int front;
   int rear;
   Student students[MAX];
```

```
} Queue;

void createQueue(Queue *q, int size);
void enqueueStudent(Queue *q, int id, char *name);
Student dequeueStudent(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);

    enqueueStudent(&q, 1, "Alice");
    enqueueStudent(&q, 2, "Bob");
    enqueueStudent(&q, 3, "Charlie");


    printf("Current queue of students waiting to board the bus:\n");
    displayQueue(&q);


    Student boarded = dequeueStudent(&q);
    if (boarded.id != -1) {
        printf("Student boarded the bus: ID=%d, Name=%s\n", boarded.id, boarded.name);
    }

    printf("Queue after boarding a student:\n");
    displayQueue(&q);
```

```
    return 0;
}



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

void enqueueStudent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}



Student dequeueStudent(Queue *q) {
    Student s = {-1, ""};
    if (q->front == q->rear) {
        printf("Queue is empty. No students waiting to board the bus.\n");
    } else {
        q->front++;
        s = q->students[q->front];
    }
```

```
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }
}
```

```
/**********************************************************************
****

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 100



typedef struct {
    int id;
    char name[50];
} Student;

typedef struct {
    int size;
    int front;
    int rear;
```

```
    Student students[MAX];
} Queue;



void createQueue(Queue *q, int size);
void enqueueStudent(Queue *q, int id, char *name);
Student dequeueStudent(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);



    enqueueStudent(&q, 1, "Alice");
    enqueueStudent(&q, 2, "Bob");
    enqueueStudent(&q, 3, "Charlie");



    printf("Current queue of students waiting for counseling:\n");
    displayQueue(&q);



    Student served = dequeueStudent(&q);
    if (served.id != -1) {
        printf("Student served: ID=%d, Name=%s\n", served.id, served.name);
    }
```

```
    printf("Queue after serving a student:\n");
    displayQueue(&q);

    return 0;
}


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


void enqueueStudent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}


Student dequeueStudent(Queue *q) {
    Student s = {-1, ""};
    if (q->front == q->rear) {
```

```
        printf("Queue is empty. No students waiting for counseling.\n");
    } else {
        q->front++;
        s = q->students[q->front];
    }
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }
}
```

```
/**************************************************************************
****

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 100


typedef struct {
    int id;
    char name[50];
} Student;


typedef struct {
    int size;
    int front;
    int rear;
```

```
    Student students[MAX];
} Queue;



void createQueue(Queue *q, int size);
void addStudent(Queue *q, int id, char *name);
Student processRegistration(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);



    addStudent(&q, 1, "Alice");
    addStudent(&q, 2, "Bob");
    addStudent(&q, 3, "Charlie");



    printf("Current queue of students waiting for registration:\n");
    displayQueue(&q);



    Student registered = processRegistration(&q);
    if (registered.id != -1) {
        printf("Student registered: ID=%d, Name=%s\n", registered.id, registered.name);
    }
```

```
    printf("Queue after processing a registration:\n");
    displayQueue(&q);

    return 0;
}



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

void addStudent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}



Student processRegistration(Queue *q) {
    Student s = {-1, ""};
    if (q->front == q->rear) {
        printf("Queue is empty. No students waiting for registration.\n");
```

```
    } else {
        q->front++;
        s = q->students[q->front];
    }
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }
}
```

```
/**********************************************************************
****

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 100

typedef struct {
    int id;
    char name[50];
} Student;

typedef struct {
    int size;
    int front;
    int rear;
```

```
    Student students[MAX];
} Queue;



void createQueue(Queue *q, int size);
void enqueueStudent(Queue *q, int id, char *name);
Student dequeueStudent(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);



    enqueueStudent(&q, 1, "Alice");
    enqueueStudent(&q, 2, "Bob");
    enqueueStudent(&q, 3, "Charlie");



    printf("Current queue of students waiting to check out equipment:\n");
    displayQueue(&q);



    Student served = dequeueStudent(&q);
    if (served.id != -1) {
        printf("Student checked out equipment: ID=%d, Name=%s\n", served.id, served.name);
    }
```

```
    printf("Queue after serving a student:\n");
    displayQueue(&q);

    return 0;
}


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


void enqueueStudent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add student ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->students[q->rear].id = id;
        strcpy(q->students[q->rear].name, name);
    }
}


Student dequeueStudent(Queue *q) {
    Student s = {-1, ""};
    if (q->front == q->rear) {
```

```
        printf("Queue is empty. No students waiting to check out equipment.\n");
    } else {
        q->front++;
        s = q->students[q->front];
    }
    return s;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->students[i].id, q->students[i].name);
        }
        printf("\n");
    }
}
```

```
/**********************************************************************
****
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 100


typedef struct {
    int id;
    char name[50];
} Parent;


typedef struct {
    int size;
    int front;
```

```
    int rear;
    Parent parents[MAX];
} Queue;


void createQueue(Queue *q, int size);
void enqueueParent(Queue *q, int id, char *name);
Parent dequeueParent(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    createQueue(&q, MAX);


    enqueueParent(&q, 1, "Alice");
    enqueueParent(&q, 2, "Bob");
    enqueueParent(&q, 3, "Charlie");


    printf("Current queue of parents waiting for the meeting:\n");
    displayQueue(&q);


    Parent served = dequeueParent(&q);
    if (served.id != -1) {
        printf("Parent met the teacher: ID=%d, Name=%s\n", served.id, served.name);
    }
```

```
    printf("Queue after serving a parent:\n");
    displayQueue(&q);

    return 0;
}



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



void enqueueParent(Queue *q, int id, char *name) {
    if (q->rear == q->size - 1) {
        printf("Queue is full. Cannot add parent ID=%d, Name=%s\n", id, name);
    } else {
        q->rear++;
        q->parents[q->rear].id = id;
        strcpy(q->parents[q->rear].name, name);
    }
}



Parent dequeueParent(Queue *q) {
    Parent p = {-1, ""};
```

```
    if (q->front == q->rear) {
        printf("Queue is empty. No parents waiting for the meeting.\n");
    } else {
        q->front++;
        p = q->parents[q->front];
    }
    return p;
}



void displayQueue(Queue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
    } else {
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("ID=%d, Name=%s -> ", q->parents[i].id, q->parents[i].name);
        }
        printf("\n");
    }
}
```

//// QUEUE using Linkedlist

```
#include <stdio.h>

#include <stdlib.h>

struct Node{

    int data;

    struct Node *next;

}*front=NULL,*rear=NULL;

void enqueue(int);

int dequeue();

void display();


int main(){

    enqueue(10);

    enqueue(20);
```

```
    enqueue(30);

    enqueue(40);

    enqueue(50);

    display();

    printf("%d \n",dequeue());

    display();

    return 0;

}

void enqueue(int x){

    struct Node *t;

    t = (struct Node*)malloc(sizeof(struct Node));

    if(t == NULL){

        printf("Queue is full \n");

    }else{
```

```
        t->data = x;

        t->next = NULL;

        if(front == NULL){

            front = rear = t;

        }else{

            rear->next = t;

            rear = t;

        }

    }

}

void display(){

    struct Node *p = front;

    while(p){

        printf("%d -> ",p->data);
```

```
        p = p->next;

    }

    printf("\n");

}

int dequeue(){

    int x = -1;

    struct Node *t;

    if(front == NULL){

        printf("Queue is already empty \n");

    }else{

        x = front->data;

        t = front;

        front = front->next;

        free(t);
```

```
    }

    return x;

}
```

```
/**********************************************************************
****
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>


typedef struct {
    char sensorType[20];
    float value;
    char timestamp[20];
```

```
} SensorReading;


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


void enqueueReading(char *sensorType, float value, char *timestamp);
SensorReading dequeueReading();
void displayQueue();
SensorReading searchReading(char *timestamp);

int main() {

    enqueueReading("Temperature", 22.5, "2025-01-20 10:00");
    enqueueReading("Pressure", 1013.25, "2025-01-20 10:01");
    enqueueReading("Temperature", 23.0, "2025-01-20 10:02");


    printf("Current queue of sensor readings:\n");
    displayQueue();


    SensorReading processed = dequeueReading();
    if (strcmp(processed.timestamp, "") != 0) {
        printf("Processed reading: SensorType=%s, Value=%.2f, Timestamp=%s\n",
```

```
            processed.sensorType, processed.value, processed.timestamp);
    }

    printf("Queue after processing a reading:\n");
    displayQueue();


    char searchTimestamp[] = "2025-01-20 10:01";
    SensorReading found = searchReading(searchTimestamp);
    if (strcmp(found.timestamp, "") != 0) {
        printf("Found reading: SensorType=%s, Value=%.2f, Timestamp=%s\n",
            found.sensorType, found.value, found.timestamp);
    } else {
        printf("Reading with timestamp %s not found.\n", searchTimestamp);
    }

    return 0;
}


void enqueueReading(char *sensorType, float value, char *timestamp) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.sensorType, sensorType);
        t->data.value = value;
```

```
        strcpy(t->data.timestamp, timestamp);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
}


SensorReading dequeueReading() {
    SensorReading reading = {"", 0, ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        reading = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return reading;
}
```

```
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("SensorType=%s, Value=%.2f, Timestamp=%s -> ",
              p->data.sensorType, p->data.value, p->data.timestamp);
        p = p->next;
    }
    printf("\n");
}



SensorReading searchReading(char *timestamp) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.timestamp, timestamp) == 0) {
            return p->data;
        }
        p = p->next;
    }
    SensorReading notFound = {"", 0, ""};
    return notFound;
}
```

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

*****************************************************************************
****/

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


typedef struct {
    int taskId;
    int priority;
    int executionTime;
} Task;


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

```
void enqueueTask(int taskId, int priority, int executionTime);
Task dequeueTask();
void displayQueue();
Task searchTaskByPriority(int priority);

int main() {

    enqueueTask(1, 2, 100);
    enqueueTask(2, 1, 200);
    enqueueTask(3, 3, 50);

    printf("Current queue of tasks:\n");
    displayQueue();

    Task executed = dequeueTask();
    if (executed.taskId != -1) {
        printf("Executed task: TaskID=%d, Priority=%d, ExecutionTime=%d\n",
               executed.taskId, executed.priority, executed.executionTime);
    }


    printf("Queue after executing a task:\n");
    displayQueue();


    int searchPriority = 1;
    Task found = searchTaskByPriority(searchPriority);
    if (found.taskId != -1) {
```

```
        printf("Found task with priority %d: TaskID=%d, ExecutionTime=%d\n",
            found.priority, found.taskId, found.executionTime);
    } else {
        printf("No task with priority %d found.\n", searchPriority);
    }

    return 0;
}



void enqueueTask(int taskId, int priority, int executionTime) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        t->data.taskId = taskId;
        t->data.priority = priority;
        t->data.executionTime = executionTime;
        t->next = NULL;

        if (front == NULL || front->data.priority > priority) {
            t->next = front;
            front = t;
            if (rear == NULL) {
                rear = t;
            }
        } else {
```

```
            p = front;
            while (p != NULL && p->data.priority <= priority) {
                q = p;
                p = p->next;
            }
            t->next = q->next;
            q->next = t;
            if (q == rear) {
                rear = t;
            }
        }
    }
}



Task dequeueTask() {
    Task task = {-1, -1, -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        task = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
```

```
    }
    return task;
}



void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("TaskID=%d, Priority=%d, ExecutionTime=%d -> ",
               p->data.taskId, p->data.priority, p->data.executionTime);
        p = p->next;
    }
    printf("\n");
}



Task searchTaskByPriority(int priority) {
    struct Node *p = front;
    while (p) {
        if (p->data.priority == priority) {
            return p->data;
        }
        p = p->next;
    }
    Task notFound = {-1, -1, -1};
    return notFound;
}
```

```
/**************************************************************************
****
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>


typedef struct {
    char source[50];
    int priority;
    void (*handler)(void);
} Interrupt;


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

```
void enqueueInterrupt(char *source, int priority, void (*handler)(void));
Interrupt dequeueInterrupt();
void displayQueue();
Interrupt searchInterruptBySource(char *source);
void sampleHandler(void);

int main() {

   enqueueInterrupt("Sensor A", 2, sampleHandler);
   enqueueInterrupt("Sensor B", 1, sampleHandler);
   enqueueInterrupt("Sensor C", 3, sampleHandler);



   printf("Current queue of interrupts:\n");
   displayQueue();



   Interrupt handled = dequeueInterrupt();
   if (strcmp(handled.source, "") != 0) {
      printf("Handled interrupt: Source=%s, Priority=%d\n", handled.source,
handled.priority);
      handled.handler();
   }



   printf("Queue after handling an interrupt:\n");
   displayQueue();
```

```
    char searchSource[] = "Sensor B";
    Interrupt found = searchInterruptBySource(searchSource);
    if (strcmp(found.source, "") != 0) {
        printf("Found interrupt: Source=%s, Priority=%d\n", found.source, found.priority);
        found.handler();
    } else {
        printf("Interrupt from source %s not found.\n", searchSource);
    }

    return 0;
}



void sampleHandler() {
    printf("Handling interrupt...\n");
}

void enqueueInterrupt(char *source, int priority, void (*handler)(void)) {
    struct Node *t, *p, *q;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.source, source);
        t->data.priority = priority;
        t->data.handler = handler;
        t->next = NULL;
```

```
        if (front == NULL || front->data.priority > priority) {
            t->next = front;
            front = t;
            if (rear == NULL) {
                rear = t;
            }
        } else {
            p = front;
            while (p != NULL && p->data.priority <= priority) {
                q = p;
                p = p->next;
            }
            t->next = q->next;
            q->next = t;
            if (q == rear) {
                rear = t;
            }
        }
    }
}



Interrupt dequeueInterrupt() {
    Interrupt interrupt = {"", -1, NULL};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
```

```
    } else {
        interrupt = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return interrupt;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("Source=%s, Priority=%d -> ", p->data.source, p->data.priority);
        p = p->next;
    }
    printf("\n");
}


Interrupt searchInterruptBySource(char *source) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.source, source) == 0) {
            return p->data;
```

```
        }
        p = p->next;
    }
    Interrupt notFound = {"", -1, NULL};
    return notFound;
}
```

```
/*************************************************************************
****
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>


typedef struct {
    int senderId;
    int receiverId;
    char payload[100];
} Message;


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

```
void enqueueMessage(int senderId, int receiverId, char *payload);
Message dequeueMessage();
void displayQueue();
Message searchMessageBySender(int senderId);

int main() {

   enqueueMessage(1, 2, "Message from 1 to 2");
   enqueueMessage(3, 4, "Message from 3 to 4");
   enqueueMessage(1, 5, "Message from 1 to 5");



   printf("Current queue of messages:\n");
   displayQueue();

   Message processed = dequeueMessage();
   if (processed.senderId != -1) {
      printf("Processed message: SenderID=%d, ReceiverID=%d, Payload=%s\n",
            processed.senderId, processed.receiverId, processed.payload);
   }



   printf("Queue after processing a message:\n");
   displayQueue();



   int searchSenderId = 1;
```

```
    Message found = searchMessageBySender(searchSenderId);
    if (found.senderId != -1) {
        printf("Found message from sender %d: ReceiverID=%d, Payload=%s\n",
            found.senderId, found.receiverId, found.payload);
    } else {
        printf("No message from sender %d found.\n", searchSenderId);
    }

    return 0;
}


void enqueueMessage(int senderId, int receiverId, char *payload) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        t->data.senderId = senderId;
        t->data.receiverId = receiverId;
        strcpy(t->data.payload, payload);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
```

```
    }
}


Message dequeueMessage() {
    Message message = {-1, -1, ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        message = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return message;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("SenderID=%d, ReceiverID=%d, Payload=%s -> ",
               p->data.senderId, p->data.receiverId, p->data.payload);
        p = p->next;
```

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


Message searchMessageBySender(int senderId) {
    struct Node *p = front;
    while (p) {
        if (p->data.senderId == senderId) {
            return p->data;
        }
        p = p->next;
    }
    Message notFound = {-1, -1, ""};
    return notFound;
}
```

/**
****

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>

typedef struct {
    char timestamp[20];
    char eventType[20];
    char description[100];
} LogEntry;

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

```
void enqueueLog(char *timestamp, char *eventType, char *description);
LogEntry dequeueLog();
void displayQueue();
LogEntry searchLogByEventType(char *eventType);
void freeMemoryIfLow();

int main() {

    enqueueLog("2025-01-20 10:00", "INFO", "System startup");
    enqueueLog("2025-01-20 10:01", "ERROR", "Sensor failure");
    enqueueLog("2025-01-20 10:02", "WARN", "Low battery");


    printf("Current queue of log entries:\n");
    displayQueue();


    freeMemoryIfLow();


    printf("Queue after memory check:\n");
    displayQueue();


    char searchEventType[] = "ERROR";
    LogEntry found = searchLogByEventType(searchEventType);
```

```
    if (strcmp(found.timestamp, "") != 0) {
        printf("Found log entry: Timestamp=%s, EventType=%s, Description=%s\n",
            found.timestamp, found.eventType, found.description);
    } else {
        printf("No log entry with event type %s found.\n", searchEventType);
    }

    return 0;
}


void enqueueLog(char *timestamp, char *eventType, char *description) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.timestamp, timestamp);
        strcpy(t->data.eventType, eventType);
        strcpy(t->data.description, description);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
```

```
}

LogEntry dequeueLog() {
    LogEntry log = {"", "", ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        log = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return log;
}

void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("Timestamp=%s, EventType=%s, Description=%s -> ",
               p->data.timestamp, p->data.eventType, p->data.description);
        p = p->next;
    }
```

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


LogEntry searchLogByEventType(char *eventType) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.eventType, eventType) == 0) {
            return p->data;
        }
        p = p->next;
    }
    LogEntry notFound = {"", "", ""};
    return notFound;
}


void freeMemoryIfLow() {

    int lowMemory = 1;
    if (lowMemory) {
        printf("Memory is low. Dequeuing the oldest log entry.\n");
        dequeueLog();
    }
}
```

```
/*************************************************************************
****
```

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>


typedef struct {
    char sourceIP[16];
    char destIP[16];
    char payload[100];
} Packet;


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

```
void enqueuePacket(char *sourceIP, char *destIP, char *payload);
Packet dequeuePacket();
void displayQueue();
Packet searchPacketByIP(char *ip);

int main() {

    enqueuePacket("192.168.1.1", "192.168.1.2", "Data from 192.168.1.1 to 192.168.1.2");
    enqueuePacket("192.168.1.3", "192.168.1.4", "Data from 192.168.1.3 to 192.168.1.4");
    enqueuePacket("192.168.1.5", "192.168.1.6", "Data from 192.168.1.5 to 192.168.1.6");


    printf("Current queue of network packets:\n");
    displayQueue();


    Packet transmitted = dequeuePacket();
    if (strcmp(transmitted.sourceIP, "") != 0) {
        printf("Transmitted packet: SourceIP=%s, DestIP=%s, Payload=%s\n",
               transmitted.sourceIP, transmitted.destIP, transmitted.payload);
    }


    printf("Queue after transmission:\n");
    displayQueue();


    char searchIP[] = "192.168.1.3";
```

```
    Packet found = searchPacketByIP(searchIP);
    if (strcmp(found.sourceIP, "") != 0) {
        printf("Found packet with source IP %s: DestIP=%s, Payload=%s\n",
            found.sourceIP, found.destIP, found.payload);
    } else {
        printf("No packet with source IP %s found.\n", searchIP);
    }

    return 0;
}


void enqueuePacket(char *sourceIP, char *destIP, char *payload) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.sourceIP, sourceIP);
        strcpy(t->data.destIP, destIP);
        strcpy(t->data.payload, payload);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
```

```
    }
}


Packet dequeuePacket() {
    Packet packet = {"", "", ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        packet = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return packet;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("SourceIP=%s, DestIP=%s, Payload=%s -> ",
               p->data.sourceIP, p->data.destIP, p->data.payload);
        p = p->next;
```

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



Packet searchPacketByIP(char *ip) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.sourceIP, ip) == 0 || strcmp(p->data.destIP, ip) == 0) {
            return p->data;
        }
        p = p->next;
    }
    Packet notFound = {"", "", ""};
    return notFound;
}
```

/**
****

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>


typedef struct {
    char version[20];
    char releaseNotes[100];
    char filePath[100];
} FirmwareUpdate;


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

```
void enqueueUpdate(char *version, char *releaseNotes, char *filePath);
FirmwareUpdate dequeueUpdate();
void displayQueue();
FirmwareUpdate searchUpdateByVersion(char *version);

int main() {

    enqueueUpdate("1.0.0", "Initial release", "/path/to/firmware1.bin");
    enqueueUpdate("1.1.0", "Bug fixes and improvements", "/path/to/firmware2.bin");
    enqueueUpdate("2.0.0", "Major update with new features", "/path/to/firmware3.bin");


    printf("Current queue of firmware updates:\n");
    displayQueue();


    FirmwareUpdate update = dequeueUpdate();
    if (strcmp(update.version, "") != 0) {
        printf("Installing update: Version=%s, ReleaseNotes=%s, FilePath=%s\n",
               update.version, update.releaseNotes, update.filePath);
    }


    printf("Queue after installing an update:\n");
    displayQueue();


    char searchVersion[] = "1.1.0";
```

```
    FirmwareUpdate found = searchUpdateByVersion(searchVersion);
    if (strcmp(found.version, "") != 0) {
        printf("Found firmware update: Version=%s, ReleaseNotes=%s, FilePath=%s\n",
            found.version, found.releaseNotes, found.filePath);
    } else {
        printf("No firmware update with version %s found.\n", searchVersion);
    }

    return 0;
}


void enqueueUpdate(char *version, char *releaseNotes, char *filePath) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.version, version);
        strcpy(t->data.releaseNotes, releaseNotes);
        strcpy(t->data.filePath, filePath);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
```

```
    }
}


FirmwareUpdate dequeueUpdate() {
    FirmwareUpdate update = {"", "", ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        update = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return update;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("Version=%s, ReleaseNotes=%s, FilePath=%s -> ",
                p->data.version, p->data.releaseNotes, p->data.filePath);
        p = p->next;
```

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

FirmwareUpdate searchUpdateByVersion(char *version) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.version, version) == 0) {
            return p->data;
        }
        p = p->next;
    }
    FirmwareUpdate notFound = {"", "", ""};
    return notFound;
}
```

/**
****

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>



typedef struct {

   char type[20];

   char timestamp[20];

   void (*action)(void);

} PowerEvent;



struct Node {

   PowerEvent data;

   struct Node *next;

} *front = NULL, *rear = NULL;



void enqueueEvent(char *type, char *timestamp, void (*action)(void));

PowerEvent dequeueEvent();
```

```
void displayQueue();
PowerEvent searchEventByType(char *type);
void powerOnAction();
void sleepAction();

int main() {

    enqueueEvent("Power On", "2025-01-20 08:00", powerOnAction);
    enqueueEvent("Sleep", "2025-01-20 22:00", sleepAction);
    enqueueEvent("Power Off", "2025-01-21 00:00", powerOnAction);



    printf("Current queue of power management events:\n");
    displayQueue();

    PowerEvent event = dequeueEvent();
    if (strcmp(event.type, "") != 0) {
        printf("Handling event: Type=%s, Timestamp=%s\n", event.type, event.timestamp);
        event.action();
    }



    printf("Queue after handling an event:\n");
    displayQueue();



    char searchType[] = "Sleep";
    PowerEvent found = searchEventByType(searchType);
```

```
    if (strcmp(found.type, "") != 0) {
        printf("Found event: Type=%s, Timestamp=%s\n", found.type, found.timestamp);
        found.action();
    } else {
        printf("No event with type %s found.\n", searchType);
    }

    return 0;
}

void powerOnAction() {
    printf("Powering on...\n");
}

void sleepAction() {
    printf("Going to sleep mode...\n");
}



void enqueueEvent(char *type, char *timestamp, void (*action)(void)) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.type, type);
        strcpy(t->data.timestamp, timestamp);
        t->data.action = action;
```

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


PowerEvent dequeueEvent() {
    PowerEvent event = {"", "", NULL};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        event = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return event;
}
```

```
void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("Type=%s, Timestamp=%s -> ", p->data.type, p->data.timestamp);
        p = p->next;
    }
    printf("\n");
}



PowerEvent searchEventByType(char *type) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.type, type) == 0) {
            return p->data;
        }
        p = p->next;
    }
    PowerEvent notFound = {"", "", NULL};
    return notFound;
}
```

/**
****

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>


typedef struct {
    int commandID;
    char commandType[20];
    char parameters[100];
} Command;


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


void enqueueCommand(int commandID, char *commandType, char *parameters);
Command dequeueCommand();
```

```
void displayQueue();
Command searchCommandByType(char *commandType);

int main() {

    enqueueCommand(1, "START", "Parameter 1");
    enqueueCommand(2, "STOP", "Parameter 2");
    enqueueCommand(3, "PAUSE", "Parameter 3");


    printf("Current queue of commands:\n");
    displayQueue();


    Command executed = dequeueCommand();
    if (executed.commandID != -1) {
        printf("Executed command: CommandID=%d, CommandType=%s, Parameters=%s\n",
               executed.commandID, executed.commandType, executed.parameters);
    }


    printf("Queue after executing a command:\n");
    displayQueue();


    char searchType[] = "STOP";
    Command found = searchCommandByType(searchType);
    if (found.commandID != -1) {
```

```
        printf("Found command with type %s: CommandID=%d, Parameters=%s\n",
            found.commandType, found.commandID, found.parameters);
    } else {
        printf("No command with type %s found.\n", searchType);
    }

    return 0;
}


void enqueueCommand(int commandID, char *commandType, char *parameters) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        t->data.commandID = commandID;
        strcpy(t->data.commandType, commandType);
        strcpy(t->data.parameters, parameters);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
}
```

```
Command dequeueCommand() {
    Command command = {-1, "", ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        command = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return command;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("CommandID=%d, CommandType=%s, Parameters=%s -> ",
                p->data.commandID, p->data.commandType, p->data.parameters);
        p = p->next;
    }
    printf("\n");
```

```
}



Command searchCommandByType(char *commandType) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.commandType, commandType) == 0) {
            return p->data;
        }
        p = p->next;
    }
    Command notFound = {-1, "", ""};
    return notFound;
}
```

/**
****

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>

typedef struct {
    char timestamp[20];
    char audioData[100];
} AudioBufferEntry;

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

void enqueueAudioSample(char *timestamp, char *audioData);
```

```
AudioBufferEntry dequeueAudioSample();
void displayQueue();
AudioBufferEntry searchSampleByTimestamp(char *timestamp);

int main() {

    enqueueAudioSample("2025-01-20 10:00", "Audio data 1");
    enqueueAudioSample("2025-01-20 10:01", "Audio data 2");
    enqueueAudioSample("2025-01-20 10:02", "Audio data 3");



    printf("Current queue of audio samples:\n");
    displayQueue();



    AudioBufferEntry playback = dequeueAudioSample();
    if (strcmp(playback.timestamp, "") != 0) {
        printf("Playing back audio sample: Timestamp=%s, AudioData=%s\n",
               playback.timestamp, playback.audioData);
    }



    printf("Queue after playback:\n");
    displayQueue();

    char searchTimestamp[] = "2025-01-20 10:01";
    AudioBufferEntry found = searchSampleByTimestamp(searchTimestamp);
    if (strcmp(found.timestamp, "") != 0) {
```

```
        printf("Found audio sample: Timestamp=%s, AudioData=%s\n",
            found.timestamp, found.audioData);
    } else {
        printf("No audio sample with timestamp %s found.\n", searchTimestamp);
    }

    return 0;
}



void enqueueAudioSample(char *timestamp, char *audioData) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.timestamp, timestamp);
        strcpy(t->data.audioData, audioData);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
}
```

```
AudioBufferEntry dequeueAudioSample() {
    AudioBufferEntry sample = {"", ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        sample = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return sample;
}



void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("Timestamp=%s, AudioData=%s -> ", p->data.timestamp, p->data.audioData);
        p = p->next;
    }
    printf("\n");
}
```

```
AudioBufferEntry searchSampleByTimestamp(char *timestamp) {
   struct Node *p = front;
   while (p) {
      if (strcmp(p->data.timestamp, timestamp) == 0) {
         return p->data;
      }
      p = p->next;
   }
   AudioBufferEntry notFound = {"", ""};
   return notFound;
}
```

```
/**********************************************************************
****
```

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>
```

```
typedef struct {
    int eventID;
    char eventType[20];
    char eventData[100];
} Event;


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


void enqueueEvent(int eventID, char *eventType, char *eventData);
Event dequeueEvent();
void displayQueue();
Event searchEventByID(int eventID);
Event searchEventByType(char *eventType);

int main() {

    enqueueEvent(1, "Button Press", "Button 1 pressed");
    enqueueEvent(2, "Sensor Trigger", "Temperature sensor triggered");
    enqueueEvent(3, "Timer Expiry", "Timer 1 expired");


    printf("Current queue of events:\n");
```

```
displayQueue();



Event handledEvent = dequeueEvent();
if (handledEvent.eventID != -1) {
    printf("Handled event: EventID=%d, EventType=%s, EventData=%s\n",
        handledEvent.eventID, handledEvent.eventType, handledEvent.eventData);
}



printf("Queue after handling an event:\n");
displayQueue();



int searchID = 2;
Event foundByID = searchEventByID(searchID);
if (foundByID.eventID != -1) {
    printf("Found event by ID %d: EventType=%s, EventData=%s\n",
        foundByID.eventID, foundByID.eventType, foundByID.eventData);
} else {
    printf("No event with ID %d found.\n", searchID);
}



char searchType[] = "Timer Expiry";
Event foundByType = searchEventByType(searchType);
if (foundByType.eventID != -1) {
    printf("Found event by type %s: EventID=%d, EventData=%s\n",
```

```
            foundByType.eventType, foundByType.eventID, foundByType.eventData);
    } else {
        printf("No event with type %s found.\n", searchType);
    }

    return 0;
}


void enqueueEvent(int eventID, char *eventType, char *eventData) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        t->data.eventID = eventID;
        strcpy(t->data.eventType, eventType);
        strcpy(t->data.eventData, eventData);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
}
```

```
Event dequeueEvent() {
   Event event = {-1, "", ""};
   struct Node *t;
   if (front == NULL) {
      printf("Queue is already empty \n");
   } else {
      event = front->data;
      t = front;
      front = front->next;
      free(t);
      if (front == NULL) {
         rear = NULL;
      }
   }
   return event;
}



void displayQueue() {
   struct Node *p = front;
   while (p) {
      printf("EventID=%d, EventType=%s, EventData=%s -> ",
            p->data.eventID, p->data.eventType, p->data.eventData);
      p = p->next;
   }
   printf("\n");
}
```

```
Event searchEventByID(int eventID) {
    struct Node *p = front;
    while (p) {
        if (p->data.eventID == eventID) {
            return p->data;
        }
        p = p->next;
    }
    Event notFound = {-1, "", ""};
    return notFound;
}



Event searchEventByType(char *eventType) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.eventType, eventType) == 0) {
            return p->data;
        }
        p = p->next;
    }
    Event notFound = {-1, "", ""};
    return notFound;
}
```

/**
****

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>

typedef struct {
    char eventType[20];
    int x, y;
    char timestamp[20];
} GUIEvent;

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

```
void enqueueEvent(char *eventType, int x, int y, char *timestamp);
GUIEvent dequeueEvent();
void displayQueue();
GUIEvent searchEventByType(char *eventType);

int main() {

    enqueueEvent("Button Click", 100, 200, "2025-01-20 10:00");
    enqueueEvent("Screen Touch", 150, 250, "2025-01-20 10:01");
    enqueueEvent("Button Click", 200, 300, "2025-01-20 10:02");



    printf("Current queue of GUI events:\n");
    displayQueue();



    GUIEvent handledEvent = dequeueEvent();
    if (strcmp(handledEvent.eventType, "") != 0) {
        printf("Handled event: EventType=%s, Coordinates=(%d, %d), Timestamp=%s\n",
            handledEvent.eventType, handledEvent.x, handledEvent.y,
handledEvent.timestamp);
    }



    printf("Queue after handling an event:\n");
    displayQueue();

    char searchType[] = "Button Click";
```

```
    GUIEvent found = searchEventByType(searchType);
    if (strcmp(found.eventType, "") != 0) {
        printf("Found event: EventType=%s, Coordinates=(%d, %d), Timestamp=%s\n",
            found.eventType, found.x, found.y, found.timestamp);
    } else {
        printf("No event with type %s found.\n", searchType);
    }

    return 0;
}


void enqueueEvent(char *eventType, int x, int y, char *timestamp) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.eventType, eventType);
        t->data.x = x;
        t->data.y = y;
        strcpy(t->data.timestamp, timestamp);
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
```

```
        }
    }
}

GUIEvent dequeueEvent() {
    GUIEvent event = {"", -1, -1, ""};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        event = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return event;
}


void displayQueue() {
    struct Node *p = front;
    while (p) {
        printf("EventType=%s, Coordinates=(%d, %d), Timestamp=%s -> ",
               p->data.eventType, p->data.x, p->data.y, p->data.timestamp);
        p = p->next;
```

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



GUIEvent searchEventByType(char *eventType) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.eventType, eventType) == 0) {
            return p->data;
        }
        p = p->next;
    }
    GUIEvent notFound = {"", -1, -1, ""};
    return notFound;
}
```

```
/*****************************************************************************
****
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>


typedef struct {
    char data[100];
    int length;
} BufferEntry;


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


void enqueueData(char *data, int length);
BufferEntry dequeueData();
```

```
void displayQueue();
BufferEntry searchDataPattern(char *pattern);

int main() {

   enqueueData("Hello", 5);
   enqueueData("World", 5);
   enqueueData("Embedded", 8);



   printf("Current queue of data chunks:\n");
   displayQueue();



   BufferEntry transmitted = dequeueData();
   if (transmitted.length != -1) {
      printf("Transmitted data: Data=%s, Length=%d\n", transmitted.data,
transmitted.length);
   }



   printf("Queue after transmission:\n");
   displayQueue();



   char searchPattern[] = "World";
   BufferEntry found = searchDataPattern(searchPattern);
   if (found.length != -1) {
```

```
        printf("Found data pattern: Data=%s, Length=%d\n", found.data, found.length);
    } else {
        printf("No data pattern \"%s\" found.\n", searchPattern);
    }

    return 0;
}



void enqueueData(char *data, int length) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strncpy(t->data.data, data, length);
        t->data.data[length] = '\0';
        t->data.length = length;
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
}
```

```
BufferEntry dequeueData() {
   BufferEntry entry = {"", -1};
   struct Node *t;
   if (front == NULL) {
      printf("Queue is already empty \n");
   } else {
      entry = front->data;
      t = front;
      front = front->next;
      free(t);
      if (front == NULL) {
         rear = NULL;
      }
   }
   return entry;
}



void displayQueue() {
   struct Node *p = front;
   while (p) {
      printf("Data=%s, Length=%d -> ", p->data.data, p->data.length);
      p = p->next;
   }
   printf("\n");
}
```

```
BufferEntry searchDataPattern(char *pattern) {
    struct Node *p = front;
    while (p) {
        if (strstr(p->data.data, pattern) != NULL) {
            return p->data;
        }
        p = p->next;
    }
    BufferEntry notFound = {"", -1};
    return notFound;
}
```

```
/*************************************************************************
****
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>

typedef struct {
    int messageID;
    int dataLength;
    char payload[8];
} CANMessage;


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


void enqueueMessage(int messageID, int dataLength, char *payload);
```

```
CANMessage dequeueMessage();
void displayQueue();
CANMessage searchMessageByID(int messageID);

int main() {

    enqueueMessage(1, 8, "ABCDEFGH");
    enqueueMessage(2, 5, "12345");
    enqueueMessage(3, 8, "HIJKLMNO");


    printf("Current queue of CAN bus messages:\n");
    displayQueue();


    CANMessage processed = dequeueMessage();
    if (processed.dataLength != -1) {
        printf("Processed message: MessageID=%d, DataLength=%d, Payload=%s\n",
            processed.messageID, processed.dataLength, processed.payload);
    }


    printf("Queue after processing a message:\n");
    displayQueue();


    int searchID = 2;
    CANMessage found = searchMessageByID(searchID);
```

```
    if (found.dataLength != -1) {
        printf("Found message: MessageID=%d, DataLength=%d, Payload=%s\n",
            found.messageID, found.dataLength, found.payload);
    } else {
        printf("No message with ID %d found.\n", searchID);
    }

    return 0;
}

void enqueueMessage(int messageID, int dataLength, char *payload) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        t->data.messageID = messageID;
        t->data.dataLength = dataLength;
        strncpy(t->data.payload, payload, dataLength);
        t->data.payload[dataLength] = '\0';
        t->next = NULL;
        if (front == NULL) {
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
```

```
}



CANMessage dequeueMessage() {
   CANMessage message = {-1, -1, ""};
   struct Node *t;
   if (front == NULL) {
      printf("Queue is already empty \n");
   } else {
      message = front->data;
      t = front;
      front = front->next;
      free(t);
      if (front == NULL) {
         rear = NULL;
      }
   }
   return message;
}



void displayQueue() {
   struct Node *p = front;
   while (p) {
      printf("MessageID=%d, DataLength=%d, Payload=%s -> ",
            p->data.messageID, p->data.dataLength, p->data.payload);
      p = p->next;
   }
```

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



CANMessage searchMessageByID(int messageID) {
    struct Node *p = front;
    while (p) {
        if (p->data.messageID == messageID) {
            return p->data;
        }
        p = p->next;
    }
    CANMessage notFound = {-1, -1, ""};
    return notFound;
}
```

/**
****

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>



typedef struct {
    char metadata[50];
    float inputFeatures[10];
    int featureCount;
} InputData;



struct Node {
    InputData data;
    struct Node *next;
```

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

void enqueueData(char *metadata, float *inputFeatures, int featureCount);
InputData dequeueData();
void displayQueue();
InputData searchDataByMetadata(char *metadata);

int main() {

   float features1[10] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
   float features2[10] = {11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0};
   float features3[10] = {21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0};

   enqueueData("Data1", features1, 10);
   enqueueData("Data2", features2, 10);
   enqueueData("Data3", features3, 10);


   printf("Current queue of input data:\n");
   displayQueue();

   InputData inferred = dequeueData();
   if (inferred.featureCount != -1) {
      printf("Processing input data: Metadata=%s, FeatureCount=%d\n",
            inferred.metadata, inferred.featureCount);
   }
```

```
    printf("Queue after processing input data:\n");
    displayQueue();


    char searchMetadata[] = "Data2";
    InputData found = searchDataByMetadata(searchMetadata);
    if (found.featureCount != -1) {
        printf("Found input data: Metadata=%s, FeatureCount=%d\n",
            found.metadata, found.featureCount);
    } else {
        printf("No input data with metadata \"%s\" found.\n", searchMetadata);
    }

    return 0;
}


void enqueueData(char *metadata, float *inputFeatures, int featureCount) {
    struct Node *t;
    t = (struct Node *)malloc(sizeof(struct Node));
    if (t == NULL) {
        printf("Queue is full \n");
    } else {
        strcpy(t->data.metadata, metadata);
        memcpy(t->data.inputFeatures, inputFeatures, featureCount * sizeof(float));
        t->data.featureCount = featureCount;
        t->next = NULL;
        if (front == NULL) {
```

```
            front = rear = t;
        } else {
            rear->next = t;
            rear = t;
        }
    }
}


InputData dequeueData() {
    InputData data = {"", {-1.0}, -1};
    struct Node *t;
    if (front == NULL) {
        printf("Queue is already empty \n");
    } else {
        data = front->data;
        t = front;
        front = front->next;
        free(t);
        if (front == NULL) {
            rear = NULL;
        }
    }
    return data;
}


void displayQueue() {
```

```
    struct Node *p = front;
    while (p) {
        printf("Metadata=%s, FeatureCount=%d -> ", p->data.metadata, p->data.featureCount);
        p = p->next;
    }
    printf("\n");
}


InputData searchDataByMetadata(char *metadata) {
    struct Node *p = front;
    while (p) {
        if (strcmp(p->data.metadata, metadata) == 0) {
            return p->data;
        }
        p = p->next;
    }
    InputData notFound = {"", {-1.0}, -1};
    return notFound;
}
```