

LAB FILE
OF
DATA STRUCTURE
M.C.A. 1st SEMESTER

An Initiative by “The Last Centre”



SUBMITTED TO:-

R. K. SHARMA

SUBMITTED BY:-

NAME: SWADEEP SINGH

ROLL NO:- 24-MCA-024

Program 1 : Write a program in C that prompts the user to enter their name and then displays a greeting message with their name.

```
#include <stdio.h>
int main() {
    char name[50];
    printf("Enter your name: ");
    scanf("%49s", name);
    printf("Hello, %s! Welcome!\n", name);
    return 0;
}
```

main.c	Run	Output
<pre>1 #include <stdio.h> 2 3 int main() { 4 char name[50]; 5 printf("Enter your name: "); 6 scanf("%49s", name); 7 printf("Hello, %s! Welcome!\n", name); 8 return 0; 9 } 10</pre>		<pre>Enter your name: swadeep Hello, swadeep! Welcome! === Code Execution Successful ===</pre>

Program 2 : Write a program in C to create a structure for a student with fields for name, age, and percentage. Prompt the user to enter details for two students and display the entered information.

```
#include <stdio.h>
struct Student {
    char name[50];
    int age;
    float percentage;
};
int main() {
    struct Student students[2];
    for (int i = 0; i < 2; i++) {
        printf("Enter details for Student %d:\n", i + 1);
        printf("Name: ");
        scanf(" %[^\\n]", students[i].name);
        printf("Age: ");
        scanf("%d", &students[i].age);
        printf("Percentage: ");
        scanf("%f", &students[i].percentage);
        printf("\\n");
    }
    printf("Entered details of the students:\\n");
    for (int i = 0; i < 2; i++) {
        printf("\\nStudent %d:\\n", i + 1);
        printf("Name: %s\\n", students[i].name);
        printf("Age: %d\\n", students[i].age);
        printf("Percentage: %.2f%%\\n", students[i].percentage);
    }
    return 0;
}
```

```
Enter details for Student 1:
Name: swadeep
Age: 21
Percentage: 21

Enter details for Student 2:
Name: rishabh
Age: 23
Percentage: 85

Entered details of the students:

Student 1:
Name: swadeep
Age: 21
Percentage: 21.00%

Student 2:
Name: rishabh
Age: 23
Percentage: 85.00%
```

Program 3 : Write a program in C to create a structure for a student with fields for name, age, and percentage. Use an array of structures to store details for multiple students, take input for each, and display the entered information.

```
#include <stdio.h>
struct Student {
    char name[50];
    int age;
    float percentage;
};

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

    struct Student students[n];
    for (int i = 0; i < n; i++) {
        printf("\nEnter details for Student %d:\n", i + 1);

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

        printf("Age: ");
        scanf("%d", &students[i].age);

        printf("Percentage: ");
        scanf("%f", &students[i].percentage);
    }

    printf("\nEntered details of the students:\n");
    for (int i = 0; i < n; i++) {
        printf("\nStudent %d:\n", i + 1);
        printf("Name: %s\\n", students[i].name);
        printf("Age: %d\\n", students[i].age);
        printf("Percentage: %.2f%%\\n", students[i].percentage);
    }

    return 0;
}
```

Enter the number of students: 2

Enter details for Student 1:

Name: rishabh

Age: 20

Percentage: 33

Enter details for Student 2:

Name: ankit

Age: 26

Percentage: 72

Entered details of the students:

Student 1:

Name: rishabh|

Age: 20

Percentage: 33.00%

Student 2:

Name: ankit

Age: 26

Percentage: 72.00%

Program 4 : Write a program in C to print the following pattern:

```
13
12 13
11 12 13
10 11 12 13
9 10 11 12 13
```

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

```
int main() {
    int start = 13;
    for (int i = 0; i < 5; i++) {
        for (int j = start - i; j <= 13; j++) {
            printf("%d ", j);
        }
        printf("\n");
    }
    return 0;
}
```

<pre>#include <stdio.h> int main() { int start = 13; for (int i = 0; i < 5; i++) { for (int j = start - i; j <= 13; j++) { printf("%d ", j); } printf("\n"); } return 0; }</pre>	<pre>13 12 13 11 12 13 10 11 12 13 9 10 11 12 13 === Code Execution Successful ===</pre>
---	---

Program 5 : Write a program in C to demonstrate pointer usage by printing addresses and values of two variables. Change the pointer's target and modify the value of a variable using the pointer, then display the updated values and addresses.

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;
    int *ptr;
    printf("Initial values and addresses:\n");
    printf("Address of a: %p, Value of a: %d\n", (void*)&a, a);
    printf("Address of b: %p, Value of b: %d\n", (void*)&b, b);

    ptr = &a;
    printf("\nPointer is now pointing to 'a'\n");
    printf("Address stored in ptr: %p, Value pointed by ptr: %d\n", (void*)ptr, *ptr);

    *ptr = 30;
    printf("Updated value of 'a' using the pointer: %d\n", a);

    ptr = &b;
    printf("\nPointer is now pointing to 'b'\n");
    printf("Address stored in ptr: %p, Value pointed by ptr: %d\n", (void*)ptr, *ptr);

    *ptr = 40;
    printf("Updated value of 'b' using the pointer: %d\n", b);

    printf("\nFinal values and addresses:\n");
    printf("Address of a: %p, Value of a: %d\n", (void*)&a, a);
    printf("Address of b: %p, Value of b: %d\n", (void*)&b, b);

    return 0;
}
```

```
Initial values and addresses:
Address of a: 0x7fffd4764ec4, Value of a: 10
Address of b: 0x7fffd4764ec0, Value of b: 20

Pointer is now pointing to 'a'
Address stored in ptr: 0x7fffd4764ec4, Value pointed by ptr: 10
Updated value of 'a' using the pointer: 30

Pointer is now pointing to 'b'
Address stored in ptr: 0x7fffd4764ec0, Value pointed by ptr: 20
Updated value of 'b' using the pointer: 40

Final values and addresses:
Address of a: 0x7fffd4764ec4, Value of a: 30
Address of b: 0x7fffd4764ec0, Value of b: 40

=== Code Execution Successful ===
```

Program 6 : Write a program in C to dynamically allocate memory for a structure using malloc to store a student's name and age. Take input for the student's details and display them using a pointer.

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

struct Student {
    char name[50];
    int age;
};

int main() {
    struct Student *student = (struct Student *)malloc(sizeof(struct Student));

    if (student == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter student's name: ");
    scanf(" %[^\\n]", student->name);
    printf("Enter student's age: ");
    scanf("%d", &student->age);

    printf("\\nStudent Details:\\n");
    printf("Name: %s\\n", student->name);
    printf("Age: %d\\n", student->age);

    free(student);

    return 0;
}
```

```
Enter student's name: swadeep
Enter student's age: 24
```

```
Student Details:
Name: swadeep
Age: 24
```

```
=== Code Execution Successful ===
```


Program 7 : Write a program in C to demonstrate pointer arithmetic by using a pointer to access and display the elements of an array.

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

```
int main() {  
    int arr[] = {10, 20, 30, 40, 50};  
    int *ptr = arr;  
    int n = sizeof(arr) / sizeof(arr[0]);  
  
    printf("Array elements using pointer arithmetic:\n");  
    for (int i = 0; i < n; i++) {  
        printf("Element %d: %d\n", i + 1, *(ptr + i));  
    }  
    return 0;  
}
```

```
Array elements using pointer arithmetic:
```

```
Element 1: 10
```

```
Element 2: 20
```

```
Element 3: 30
```

```
Element 4: 40
```

```
Element 5: 50
```

```
=== Code Execution Successful ===
```

Program 8 : Write a program in C to take 5 numbers as input, calculate their sum, and find the greatest number among them.

```
#include <stdio.h>

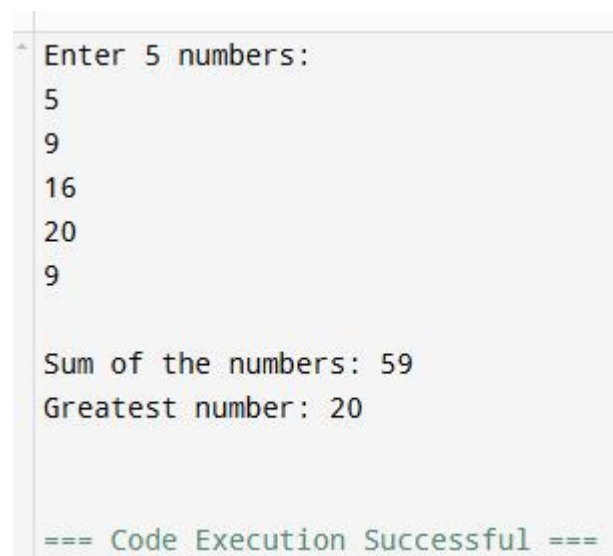
int main() {
    int numbers[5];
    int sum = 0;
    int greatest;
    printf("Enter 5 numbers:\n");
    for (int i = 0; i < 5; i++) {
        scanf("%d", &numbers[i]);
        sum += numbers[i];
    }

    greatest = numbers[0];

    for (int i = 1; i < 5; i++) {
        if (numbers[i] > greatest) {
            greatest = numbers[i];
        }
    }

    printf("\nSum of the numbers: %d\n", sum);
    printf("Greatest number: %d\n", greatest);

    return 0;
}
```



The screenshot shows a terminal window with the following text:

```
Enter 5 numbers:
5
9
16
20
9

Sum of the numbers: 59
Greatest number: 20

=== Code Execution Successful ===
```

Program 9 : Write a program in C to input 5 numbers, search for a specific number in the array, and print whether the number exists or not.

```
#include <stdio.h>

int main() {
    int numbers[5];
    int searchNum, found = 0;

    printf("Enter 5 numbers:\n");
    for (int i = 0; i < 5; i++) {
        scanf("%d", &numbers[i]);
    }

    printf("Enter the number to search: ");
    scanf("%d", &searchNum);

    for (int i = 0; i < 5; i++) {
        if (numbers[i] == searchNum) {
            found = 1;
            break;
        }
    }

    if (found) {
        printf("Number %d found in the array.\n", searchNum);
    } else {
        printf("Number %d not found in the array.\n", searchNum);
    }

    return 0;
}
```

```
Enter 5 numbers:
7
25
+2
62
17
Enter the number to search: 2
Number 2 found in the array.

=== Code Execution Successful ===
```

Program 10 : Write a program in C to implement a stack with operations to push, pop, and display elements. The program should handle stack overflow and underflow conditions.

```
#include <stdio.h>
#define MAX 5
struct Stack {
    int arr[MAX];
    int top;
};

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

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

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

void push(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow! Cannot push %d onto the stack.\n", value);
    } else {
        stack->arr[++stack->top] = value; // Increment top and add value
        printf("Pushed %d onto the stack.\n", value);
    }
}

int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow! No elements to pop.\n");
        return -1;
    } else {
        return stack->arr[stack->top--];
    }
}

void display(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements: ");
    }
}
```

```

        for (int i = 0; i <= stack->top; i++) {
            printf("%d ", stack->arr[i]);
        }
        printf("\n");
    }
}

int main() {
    struct Stack stack;
    int choice, value;
    initStack(&stack);
    do {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&stack, value);
                break;
            case 2:
                value = pop(&stack);
                if (value != -1) {
                    printf("Popped %d from the stack.\n", value);
                }
                break;
            case 3:
                display(&stack);
                break;
            case 4:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);
    return 0;
}

```

```
Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 222
Pushed 222 onto the stack.
```

```
Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 222 |
```

```
Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
```

```
Enter your choice: 2
Popped 222 from the stack.
```

```
Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting program.
```

```
=== Code Execution Successful ===
```

Program 11 : Implement a dynamic stack using a linked list in C, providing operations to push, pop, display the stack, and count the number of nodes, with a menu-driven interface.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};

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

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

int pop(struct Node** top) {
    if (isEmpty(*top)) {
        printf("Stack Underflow! No elements to pop.\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedValue;
}

void display(struct Node* top) {
    if (isEmpty(top)) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack elements: ");
    struct Node* current = top;
```

```

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

int countNodes(struct Node* top) {
    int count = 0;
    struct Node* current = top;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

int main() {
    struct Node* stack = NULL;
    int choice, value;
    do {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Count Nodes\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: // Push
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&stack, value);
                break;
            case 2: // Pop
                value = pop(&stack);
                if (value != -1) {
                    printf("Popped %d from the stack.\n", value);
                }
                break;
            case 3: // Display
                display(stack);
                break;
            case 4: // Count Nodes
                printf("Number of nodes in the stack: %d\n", countNodes(stack));
                break;
        }
    } while (choice != 5);
}

```



```

        case 5: // Exit
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
} while (choice != 5);
return 0;
}

```

```

Stack Operations:
1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit
Enter your choice: 1
Enter value to push: 20
Pushed 20 onto the stack.

```

```

Stack Operations:
1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit
Enter your choice: 1
Enter value to push: 40
Pushed 40 onto the stack.

```

```

Stack Operations:
1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit
Enter your choice: 1
Enter value to push: 60
Pushed 60 onto the stack.

```

```

Stack Operations:
1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit
Enter your choice: 3
Stack elements: 60 40 20

```

Stack Operations:

1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit

Enter your choice: 4

Number of nodes in the stack: 3

Stack Operations:

1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit

Enter your choice: 2

Popped 60 from the stack.

Stack Operations:

1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit

Enter your choice: 3

Stack elements: 40 20

Stack Operations:

1. Push
2. Pop
3. Display
4. Count Nodes
5. Exit

Enter your choice: 5

Exiting program.

Program 12 : Implement a queue using an array with operations to enqueue, dequeue, display the queue, and handle overflow and underflow conditions in a menu-driven interface.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5 // Maximum size of the queue
struct Queue {
    int arr[MAX]; // Array to store queue elements
    int front;    // Index of the front element
    int rear;     // Index of the rear element
};
void initQueue(struct Queue* queue) {
    queue->front = -1;
    queue->rear = -1;
}
int isEmpty(struct Queue* queue) {
    return queue->front == -1;
}
int isFull(struct Queue* queue) {
    return queue->rear == MAX - 1;
}
void enqueue(struct Queue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue Overflow! Cannot enqueue %d.\n", value);
    } else {
        if (queue->front == -1) // If queue is empty, set front to 0
            queue->front = 0;
        queue->rear++; // Increment rear index
        queue->arr[queue->rear] = value; // Add the element at the rear
        printf("Enqueued %d into the queue.\n", value);
    }
}
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! No elements to dequeue.\n");
        return -1;
    } else {
        int value = queue->arr[queue->front];
        if (queue->front == queue->rear) {
            queue->front = -1;
            queue->rear = -1;
        } else {
            queue->front++;
        }
    }
}
```

```

        return value;
    }
}

void display(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = queue->front; i <= queue->rear; i++) {
            printf("%d ", queue->arr[i]);
        }
        printf("\n");
    }
}

int main() {
    struct Queue queue;
    int choice, value;
    initQueue(&queue); // Initialize the queue
    do {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: // Enqueue
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&queue, value);
                break;
            case 2: // Dequeue
                value = dequeue(&queue);
                if (value != -1) {
                    printf("Dequeued %d from the queue.\n", value);
                }
                break;
            case 3: // Display
                display(&queue);
                break;
            case 4: // Exit
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

```

```
    } while (choice != 4);  
    return 0;  
}
```

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 1

Enter value to enqueue: 20

Enqueued 20 into the queue.

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 1

Enter value to enqueue: 40.

Enqueued 40 into the queue.

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 3

Queue elements: 20 40

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 2

Dequeued 20 from the queue.

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 4

Exiting program.

Program 13 : Implement a dynamic queue using a linked list with operations to enqueue, dequeue, display the queue, and count the number of nodes, with a menu-driven interface.

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

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

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

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

void enqueue(struct Node** front, struct Node** rear, int value) {
    struct Node* newNode = createNode(value);
    if (*rear == NULL) { // If queue is empty
        *front = *rear = newNode;
    } else {
        (*rear)->next = newNode;
        *rear = newNode;
    }
    printf("Enqueued %d into the queue.\n", value);
}

int dequeue(struct Node** front, struct Node** rear) {
    if (isEmpty(*front)) {
        printf("Queue Underflow! No elements to dequeue.\n");
        return -1;
    } else {
        struct Node* temp = *front;
        int value = temp->data;
        *front = (*front)->next;

        if (*front == NULL) { // If the queue becomes empty after dequeue
            *rear = NULL;
        }
        free(temp); // Free the memory of the dequeued node
        return value;
    }
}

void display(struct Node* front) {
```

```

    if (isEmpty(front)) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        struct Node* current = front;
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
}

int countNodes(struct Node* front) {
    int count = 0;
    struct Node* current = front;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

int main() {
    struct Node* front = NULL;
    struct Node* rear = NULL;
    int choice, value;
    do {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Count Nodes\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: // Enqueue
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&front, &rear, value);
                break;
            case 2: // Dequeue
                value = dequeue(&front, &rear);
                if (value != -1) {
                    printf("Dequeued %d from the queue.\n", value);
                }
                break;
            case 3: // Display
                display(front);
                break;
            case 4: // Count Nodes
                printf("Number of nodes in the queue: %d\n", countNodes(front));
                break;
            case 5: // Exit
                printf("Exiting program.\n");
                break;
        }
    } while (choice != 5);
}

```

```

        default:
            printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);
    return 0;
}

```

Queue Operations:

```

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit
Enter your choice: 1
Enter value to enqueue: 10
Enqueued 10 into the queue.

```

Queue Operations:

```

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit
Enter your choice: 1
Enter value to enqueue: 11
Enqueued 11 into the queue.

```

Queue Operations:

```

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit
Enter your choice: 1
Enter value to enqueue: 12
Enqueued 12 into the queue.

```

Queue Operations:

```

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit
Enter your choice: 3
Queue elements: 10 11 12

```


Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit

Enter your choice: 2

Dequeued 10 from the queue.

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit

Enter your choice: 4

Number of nodes in the queue: 2

Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Count Nodes
5. Exit

Enter your choice: 5

Exiting program.

Program 14 : Implement a singly linked list with operations to insert elements at the beginning, end, or any specific position, delete elements from the beginning, end, or any specified position, display the list, and count the number of nodes, with a menu-driven interface.

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

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

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

void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
    printf("Inserted %d at the beginning.\n", value);
}

void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("Inserted %d at the end.\n", value);
}
```

```

void insertAtPosition(struct Node** head, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
    }
    if (position == 1) {
        insertAtBeginning(head, value);
        return;
    }
    struct Node* newNode = createNode(value);
    struct Node* temp = *head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        free(newNode);
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
        printf("Inserted %d at position %d.\n", value, position);
    }
}

```

```

// Function to delete a node from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

```

```

// Function to delete a node from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    if (temp->next == NULL) {
        printf("Deleted %d from the end.\n", temp->data);
        free(temp);
    }
}

```

```

        *head = NULL;
        return;
    }
    struct Node* prev = NULL;
    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}

// Function to delete a node from a specific position
void deleteFromPosition(struct Node** head, int position) {
    if (*head == NULL || position < 1) {
        printf("Invalid position or list is empty!\n");
        return;
    }
    if (position == 1) {
        deleteFromBeginning(head);
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    for (int i = 1; i < position && temp != NULL; i++) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
    } else {
        prev->next = temp->next;
        printf("Deleted %d from position %d.\n", temp->data, position);
        free(temp);
    }
}

// Function to display the linked list
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Linked list: ");
    struct Node* temp = head;
    while (temp != NULL) {

```

```

        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to count the number of nodes in the linked list
int countNodes(struct Node* head) {
    int count = 0;
    struct Node* temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    return count;
}

// Main function with menu-driven interface
int main() {
    struct Node* head = NULL;
    int choice, value, position;

    do {
        printf("\nLinked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Position\n");
        printf("7. Display List\n");
        printf("8. Count Nodes\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtBeginning(&head, value);
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtEnd(&head, value);
                break;

```

```

    case 3:
        printf("Enter value to insert: ");
        scanf("%d", &value);
        printf("Enter position to insert: ");
        scanf("%d", &position);
        insertAtPosition(&head, value, position);
        break;
    case 4:
        deleteFromBeginning(&head);
        break;
    case 5:
        deleteFromEnd(&head);
        break;
    case 6:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        deleteFromPosition(&head, position);
        break;
    case 7:
        display(head);
        break;
    case 8:
        printf("Number of nodes: %d\n", countNodes(head));
        break;
    case 9:
        printf("Exiting program.\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 9);

return 0;
}

```

Output

Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 2

Enter value to insert: 2

Inserted 2 at the end.

Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 1

Enter value to insert: 1

Inserted 1 at the beginning.

Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 4

Deleted 1 from the beginning.

Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning

```
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit
Enter your choice: 2
Enter value to insert: 4
Inserted 4 at the end.
```

```
Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit
Enter your choice: 2
Enter value to insert: 6
Inserted 6 at the end.
```

```
Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit
Enter your choice: 7
Linked list: 2 -> 4 -> 6 -> NULL
```

```
Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit
```


Enter choice: 8
Number of nodes: 3

Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter choice: 9
Exiting program.

Program 15 : Implement a doubly linked list with operations to insert elements at the beginning, end, or any specific position, delete elements from the beginning, end, or any specified position, display the list in forward and backward direction, and count the number of nodes, with a menu-driven interface.

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

// Define the structure of a doubly linked list node
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning
void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head != NULL) {
        (*head)->prev = newNode;
    }
    newNode->next = *head;
    *head = newNode;
    printf("Inserted %d at the beginning.\n", value);
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    }
```

```

    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Inserted %d at the end.\n", value);
}

// Function to insert a node at a specific position
void insertAtPosition(struct Node** head, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
    }
    if (position == 1) {
        insertAtBeginning(head, value);
        return;
    }
    struct Node* newNode = createNode(value);
    struct Node* temp = *head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        free(newNode);
    } else {
        newNode->next = temp->next;
        newNode->prev = temp;
        if (temp->next != NULL) {
            temp->next->prev = newNode;
        }
        temp->next = newNode;
        printf("Inserted %d at position %d.\n", value, position);
    }
}

// Function to delete a node from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;

```

```

    *head = temp->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

```

```

// Function to delete a node from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    } else {
        *head = NULL;
    }
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}

```

```

// Function to delete a node from a specific position
void deleteFromPosition(struct Node** head, int position) {
    if (*head == NULL || position < 1) {
        printf("Invalid position or list is empty!\n");
        return;
    }
    struct Node* temp = *head;
    for (int i = 1; i < position && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range!\n");
        return;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        *head = temp->next;
    }
    if (temp->next != NULL) {

```

```

        temp->next->prev = temp->prev;
    }
    printf("Deleted %d from position %d.\n", temp->data, position);
    free(temp);
}

```

// Function to display the list in forward direction

```

void displayForward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("List in forward direction: ");
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

```

// Function to display the list in backward direction

```

void displayBackward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    printf("List in backward direction: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
}

```

// Function to count the number of nodes in the list

```

int countNodes(struct Node* head) {
    int count = 0;
    struct Node* temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
}

```

```

    return count;
}

// Main function with a menu-driven interface
int main() {
    struct Node* head = NULL;
    int choice, value, position;

    do {
        printf("\nDoubly Linked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Position\n");
        printf("7. Display Forward\n");
        printf("8. Display Backward\n");
        printf("9. Count Nodes\n");
        printf("10. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtBeginning(&head, value);
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtEnd(&head, value);
                break;
            case 3:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position to insert: ");
                scanf("%d", &position);
                insertAtPosition(&head, value, position);
                break;
            case 4:
                deleteFromBeginning(&head);
                break;
            case 5:
                deleteFromEnd(&head);
                break;
        }
    } while (choice != 10);
}

```

```

    case 6:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        deleteFromPosition(&head, position);
        break;
    case 7:
        displayForward(head);
        break;
    case 8:
        displayBackward(head);
        break;
    case 9:
        printf("Number of nodes: %d\n", countNodes(head));
        break;
    case 10:
        printf("Exiting program.\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 10);

return 0;
}

```

Output

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit

Enter your choice: 1

Enter value to insert: 10

Inserted 10 at the beginning.

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward

8. Display Backward

9. Count Nodes

10. Exit

Enter your choice: 1

Enter value to insert: 20

Inserted 20 at the beginning.

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit

Enter your choice: 1

Enter value to insert: 30

Inserted 30 at the beginning.

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit

Enter your choice: 3

Enter value to insert: 2

Enter position to insert: 2

Inserted 2 at position 2.

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward

8. Display Backward

9. Count Nodes

10. Exit

Enter your choice: 8

List in backward direction: 10 -> 20 -> 2 -> 30 -> NULL

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit

Enter your choice: 9

Number of nodes: 4

Doubly Linked List Operations:

1. Insert at Beginning
2. Insert at End

```
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit
Enter your choice: 4
Deleted 30 from the beginning.
```

```
Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit
Enter your choice: 7
```

```
List in forward direction: 2 -> 20 -> 10 -> NULL
```

```
Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display Forward
8. Display Backward
9. Count Nodes
10. Exit
Enter your choice: 10
Exiting program.
```

Program 16 : Implement a Circular Queue in C with operations to enqueue, dequeue, display, and handle overflow/underflow conditions.

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

#define MAX 5 // Define the maximum size of the circular queue

// Circular Queue structure
struct CircularQueue {
    int items[MAX];
    int front;
    int rear;
};

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

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

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

// Function to add an element to the queue (Enqueue)
void enqueue(struct CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow! Cannot enqueue %d.\n", value);
        return;
    }

    if (isEmpty(q)) { // If queue is initially empty
        q->front = 0;
    }
    q->rear = (q->rear + 1) % MAX;
    q->items[q->rear] = value;
    printf("Enqueued %d into the queue.\n", value);
}
```

```

// Function to remove an element from the queue (Dequeue)
void dequeue(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue Underflow! Cannot dequeue.\n");
        return;
    }

    int value = q->items[q->front];
    if (q->front == q->rear) { // Only one element was in the queue
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }
    printf("Dequeued %d from the queue.\n", value);
}

// Function to display the elements of the queue
void displayQueue(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Circular Queue elements: ");
    int i = q->front;
    while (1) {
        printf("%d ", q->items[i]);
        if (i == q->rear) break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

// Main function with a menu-driven interface
int main() {
    struct CircularQueue q;
    initQueue(&q);

    int choice, value;

    do {
        printf("\nCircular Queue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display Queue\n");
    }

```

```

printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter value to enqueue: ");
        scanf("%d", &value);
        enqueue(&q, value);
        break;
    case 2:
        dequeue(&q);
        break;
    case 3:
        displayQueue(&q);
        break;
    case 4:
        printf("Exiting...\n");
        break;
    default:
        printf("Invalid choice! Please try again.\n");
}
} while (choice != 4);

return 0;
}

```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
Enter value to enqueue: 10
Enqueued 10 into the queue.
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
Enter value to enqueue: 20
Enqueued 20 into the queue.
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
Enter value to enqueue: 25
Enqueued 25 into the queue.
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 1
Enter value to enqueue: 30
Enqueued 30 into the queue.
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 3
Circular Queue elements: 10 20 25 30
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 2
Dequeued 10 from the queue.
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Exit
Enter your choice: 4
Exiting...
```

Program 17 : Write a program to implement a Circular Linked List with the following operations: Insert at Beginning, Insert at End, Insert at any position, Delete from Beginning, Delete from End, Delete from any position, Display the list, and Count the number of nodes.

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

// Structure for a node in a circular linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = newNode; // Pointing to itself (circular nature)
    return newNode;
}

// Function to insert a node at the beginning
void insertAtBeginning(struct Node** tail, int value) {
    struct Node* newNode = createNode(value);
    if (*tail == NULL) {
        *tail = newNode;
    } else {
        newNode->next = (*tail)->next;
        (*tail)->next = newNode;
    }
    printf("Inserted %d at the beginning.\n", value);
}

// Function to insert a node at the end
void insertAtEnd(struct Node** tail, int value) {
    struct Node* newNode = createNode(value);
    if (*tail == NULL) {
        *tail = newNode;
    } else {
```

```

        newNode->next = (*tail)->next;
        (*tail)->next = newNode;
        *tail = newNode;
    }
    printf("Inserted %d at the end.\n", value);
}

// Function to insert a node at a specific position
void insertAtPosition(struct Node** tail, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
    }

    if (*tail == NULL || position == 1) {
        insertAtBeginning(tail, value);
        return;
    }

    struct Node* newNode = createNode(value);
    struct Node* current = (*tail)->next;
    for (int i = 1; i < position - 1 && current != *tail; i++) {
        current = current->next;
    }

    if (current == *tail && position > 2) {
        printf("Position out of range!\n");
        free(newNode);
    } else {
        newNode->next = current->next;
        current->next = newNode;
        if (current == *tail) {
            *tail = newNode;
        }
        printf("Inserted %d at position %d.\n", value, position);
    }
}

// Function to delete a node from the beginning
void deleteFromBeginning(struct Node** tail) {
    if (*tail == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = (*tail)->next;
    if (*tail == temp) { // Only one node in the list

```



```

        *tail = NULL;
    } else {
        (*tail)->next = temp->next;
    }
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

```

// Function to delete a node from the end

```

void deleteFromEnd(struct Node** tail) {
    if (*tail == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* current = (*tail)->next;
    if (current == *tail) { // Only one node
        printf("Deleted %d from the end.\n", current->data);
        free(current);
        *tail = NULL;
    } else {
        while (current->next != *tail) {
            current = current->next;
        }
        struct Node* temp = *tail;
        current->next = temp->next;
        *tail = current;
        printf("Deleted %d from the end.\n", temp->data);
        free(temp);
    }
}

```

// Function to delete a node from a specific position

```

void deleteFromPosition(struct Node** tail, int position) {
    if (*tail == NULL || position < 1) {
        printf("Invalid position or list is empty!\n");
        return;
    }

    struct Node* current = (*tail)->next;
    if (position == 1) {
        deleteFromBeginning(tail);
        return;
    }

```

```

    struct Node* prev = NULL;
    for (int i = 1; i < position && current != *tail; i++) {

```

```

        prev = current;
        current = current->next;
    }

    if (current == *tail && position > 1) {
        printf("Position out of range!\n");
    } else {
        prev->next = current->next;
        if (current == *tail) {
            *tail = prev;
        }
        printf("Deleted %d from position %d.\n", current->data, position);
        free(current);
    }
}

// Function to display the list
void displayList(struct Node* tail) {
    if (tail == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* current = tail->next;
    printf("Circular Linked List: ");
    do {
        printf("%d -> ", current->data);
        current = current->next;
    } while (current != tail->next);
    printf("(head)\n");
}

// Function to count the number of nodes
int countNodes(struct Node* tail) {
    if (tail == NULL) return 0;

    int count = 0;
    struct Node* current = tail->next;
    do {
        count++;
        current = current->next;
    } while (current != tail->next);

    return count;
}

// Main function with menu-driven interface

```

```

int main() {
    struct Node* tail = NULL;
    int choice, value, position;

    do {
        printf("\nCircular Linked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Position\n");
        printf("7. Display List\n");
        printf("8. Count Nodes\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtBeginning(&tail, value);
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertAtEnd(&tail, value);
                break;
            case 3:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position: ");
                scanf("%d", &position);
                insertAtPosition(&tail, value, position);
                break;
            case 4:
                deleteFromBeginning(&tail);
                break;
            case 5:
                deleteFromEnd(&tail);
                break;
            case 6:
                printf("Enter position to delete: ");
                scanf("%d", &position);
                deleteFromPosition(&tail, position);
                break;
        }
    } while (choice != 9);
}

```

```

        case 7:
            displayList(tail);
            break;
        case 8:
            printf("Number of nodes: %d\n", countNodes(tail));
            break;
        case 9:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Try again.\n");
    }
} while (choice != 9);

return 0;
}

```

Output

Circular Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 2

Enter value to insert: 10

Inserted 10 at the end.

Circular Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 1

Enter value to insert: 20

Inserted 20 at the beginning.

Circular Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 2

Enter value to insert: 30

Inserted 30 at the end.

Circular Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 7

Circular Linked List: 20 -> 10 -> 30 -> (head)

Circular Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

Enter your choice: 8

Number of nodes: 3

Circular Linked List Operations:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

```
Enter your choice: 5
Deleted 30 from the end.
```

```
Circular Linked List Operations:
```

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit

```
Enter your choice: 8
```

```
Number of nodes: 2
```

```
Circular Linked List Operations:
```

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Count Nodes
9. Exit