UNIVERSITY OF CALCUTTA

Four Year B.Sc. Semester - III Practical Examination, 2024

(Under CCF, 2022)

CMSM (B.Sc. Computer Science Four Year [H])

2024 – 2025

CU Roll No :	233611-21-0003
CU Registration No :	611-1111-0287-23
Semester :	III
Subject Code :	DSCC – 3
Subject Name :	Data Structures Lab using C

INDEX

INDEX

Sl. No	Date	Assignment Number	Page No.	Teacher's Signature
21		Assignment-21		
22		Assignment-22		
23		Assignment-23		
24		Assignment-24		
25		Assignment-25		
26		Assignment-26		
27		Assignment-27		
28		Assignment-28		

Assignment - 01

Date : __________

Objective :

Write a program in C to perform merging of two one-dimensional arrays, after taking all the necessary inputs from the user.

Algorithm :

Algorithm **MergeArray** :
Input : two arrays A1[L1...U1], A2[L2...U2]
Output : final merged array in A1 (considering A1 have enough space to accommodate A2)
Data Structure : Array

```
Start
I = U1 + 1
N = I + U2 + 1
While I <= N Do
    A1[I] = A2[I - U1 + 1]
    I = I + 1
EndWhile
Stop
```

Source Code :

```c
#include <stdio.h>
#include <assert.h>
#define MAX 10

int inputArray(int* arr, const char* prompt){
    int size = 0;
    printf("%s\n", prompt);
    printf("Enter Size : ");
    scanf("%d", &size);
    assert(size <= MAX);
    printf("Enter Array : ");
    for(int i = 0; i < size; i++) scanf("%d", &arr[i]);
    printf("\n");
    return size;
}

int mergeArray(int size1, int* arr1, int size2, int* arr2){
    assert(size1 + size2 <= MAX);
    for(int i = size1; i < size1 + size2; i++) arr1[i] = arr2[i - size1];
    return size1 + size2;
```

```
}

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

int main(){
    int arr1[MAX], arr2[MAX];
    int size1 = inputArray(arr1, "Array 1:");
    int size2 = inputArray(arr2, "Array 2:");
    printArray(size1, arr1, "Array 1 before Merging : ");
    printArray(size2, arr2, "Array 2 before Merging : ");
    size1 = mergeArray(size1, arr1, size2, arr2);
    printArray(size1, arr1, "Array 1 after Merging : ");
    printArray(size2, arr2, "Array 2 after Merging : ");
}
```

Code Output :

```
Array 1:
Enter Size : 4
Enter Array : 6 7 1 3

Array 2:
Enter Size : 5
Enter Array : 2 8 9 3 4

Array 1 before Merging : 6 7 1 3
Array 2 before Merging : 2 8 9 3 4
Array 1 after Merging : 6 7 1 3 2 8 9 3 4
Array 2 after Merging : 2 8 9 3 4
```

Conclusion :

The program efficiently demonstrates merging two arrays in-place while adhering to size constraints using modular functions. It ensures safety with assert checks and avoids additional memory allocation. The clean structure makes it reusable and practical for array manipulation tasks.

Teachers' Signature

Assignment - 02

Date : __________

Objective :

Write a program in C to implement the following matrix operations :

a. Multiplication of two matrices
b. Transpose of a given matrix.

Algorithm :

Algorithm **MultiplyMatrix**:
Input : three two-dimensional arrays M1[L1...U1][L2...U2], M2[L2...U2][L3...U3], M3[L1...U1][L3...U3]
Output : multiplication of M1 and M2 will be stored in M3
Data Structure : two-dimensional array

```
Start
I = L1, J = L3, K = L2
While I ⩽ U1 Do
    J = L3
    While J ⩽ U3 Do
        SUM = 0, K = L2
        While K ⩽ U2 Do
            SUM = SUM + (M1[I][K] * M2[K][J])
            K = K + 1
        EndWhile
        M3[I][J] = SUM
        J = J + 1
    EndWhile
    I = I + 1
EndWhile
Stop
```

Algorithm **TransposeMatrix**:
Input : two two-dimensional arrays M1[L1...U1][L2...U2], M2[L2...U2][L1...U1]
Output : transpose of M1 is stored in M2
Data Structure : two-dimensional array

```
Start
I = L1, J = L2
While I ⩽ U1 Do
    J = L2
    While J ⩽ U2 Do
        M2[J][I] = M1[I][J]
        J = J + 1
```

```
    EndWhile
    I = I + 1
EndWhile
Stop
```

Source Code :

```c
#include <stdio.h>
#include <assert.h>
#define MAX 20

typedef struct matrix{
    int col;
    int row;
    int mat[MAX][MAX];
} Matrix;

Matrix CreateMatrix(int row, int col){
    assert(row <= MAX && col <= MAX);
    Matrix m = {
        .col = col,
        .row = row
    };
    return m;
}

Matrix MultiplyMatrix(Matrix A, Matrix B){
    assert(A.col == B.row);
    Matrix C = CreateMatrix(A.row, B.col);
    for(int i = 0; i < A.row; i++){
        for(int j = 0; j < B.col; j++){
            int sum = 0;
            for(int k = 0; k < A.col; k++){
                sum += A.mat[i][k] * B.mat[k][j];
            }
            C.mat[i][j] = sum;
        }
    }
    return C;
}

Matrix TransposeMatrix(Matrix A){
    Matrix T = CreateMatrix(A.col, A.row);
    for(int i = 0; i < A.row; i++){
        for(int j = 0; j < A.col; j++){
            T.mat[j][i] = A.mat[i][j];
        }
    }
    return T;
}
```

```
Matrix CreateInputFilledMatrix(int row, int col){
    Matrix A = CreateMatrix(row, col);
    printf("Enter Matrix Elements : ");
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            scanf("%d", &(A.mat[i][j]));
        }
    }
    return A;
}

void DisplayMatrix(Matrix M){
    for(int i = 0; i < M.row; i++){
        for(int j = 0; j < M.col; j++){
            printf("%3d ", M.mat[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

int main(){
    Matrix A, B, T, C;
    int row, col;

    printf("Enter Row and Column size of Matrix A : ");
    scanf("%d%d", &row, &col);
    A = CreateInputFilledMatrix(row, col);

    printf("Enter Row and Column size of Matrix B : ");
    scanf("%d%d", &row, &col);
    B = CreateInputFilledMatrix(row, col);

    printf("Matrix A : \n");
    DisplayMatrix(A);

    printf("Matrix B : \n");
    DisplayMatrix(B);

    C = MultiplyMatrix(A, B);
    T = TransposeMatrix(A);

    printf("Matrix A * B : \n");
    DisplayMatrix(C);

    printf("Matrix A' : \n");
    DisplayMatrix(T);
}
```

Code Output :

```
Enter Row and Column size of Matrix A : 3 3
Enter Matrix Elements : 5 7 8 9 2 1 3 4 6
Enter Row and Column size of Matrix B : 3 3
Enter Matrix Elements : 1 0 0 0 1 0 0 0 1
Matrix A :
  5   7   8
  9   2   1
  3   4   6

Matrix B :
  1   0   0
  0   1   0
  0   0   1

Matrix A * B :
  5   7   8
  9   2   1
  3   4   6

Matrix A' :
  5   9   3
  7   2   4
  8   1   6
```

Conclusion :

This program efficiently implements matrix operations like creation, multiplication, transposition, and display using a structured Matrix type. It ensures dimension compatibility with assert and provides a user-friendly interface for input and output. The modular design makes the code clean, reusable, and a solid foundation for advanced matrix computations.

Teachers' Signature

Assignment - 03

Date : __________

Objective :

Write a program in C to implement stack operations using array and perform Insertion and Deletion operations. Show all possible exception/error cases.

Algorithm :

Algorithm **Push**:
Input : a stack STACK, pointer to top TOP, item to be inserted ITEM
Output : stack with newly pushed item
Data Structure : stack implemented with array with MAX capacity

```
Start
If TOP >= MAX Then
    Print "Stack Overflow"
    Exit
EndIf
TOP = TOP + 1
STACK[TOP] = ITEM
Stop
```

Algorithm **Pop**:
Input : a stack STACK, pointer to top TOP
Output : stack without the popped element ITEM
Data Structure : stack implemented with array with MAX capacity

```
Start
If TOP == -1 Then
    Print "Stack Underflow"
    Exit
EndIf
ITEM = STACK[TOP]
TOP = TOP - 1
Stop
```

Source Code :

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

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

```c
    int top;
} Stack;

Stack CreateStack(){
    Stack s;
    s.top = -1;
    return s;
}

void Push(Stack* s, int data){
    if(s->top == MAX - 1){
        printf("Stack Overflow\n\n");
        return;
    }
    s->top++;
    s->stack[s->top] = data;
}

int Pop(Stack* s){
    if(s->top == -1){
        printf("Stack Underflow\n\n");
        return -1;
    }
    return s->stack[s->top--];
}

void Display(Stack* s){
    if(s->top == -1){
        printf("|Empty|\n\n");
    }else{
        printf("|%4d| <- TOP\n", s->stack[s->top]);
        for(int i = s->top - 1; i >= 0; i--){
            printf("|%4d|\n", s->stack[i]);
        }
        printf("------\n\n");
    }
}

int main(){
    Stack stack = CreateStack();

    Push(&stack, 10);
    Display(&stack);
    Push(&stack, 20);
    Display(&stack);
    Push(&stack, 30);
    Display(&stack);
    Push(&stack, 40);
    Display(&stack);
    Push(&stack, 50);
    Display(&stack);
```

```
    Push(&stack, 60);

    printf("Popped Element : %d\n",  Pop(&stack));
    Display(&stack);
    printf("Popped Element : %d\n",  Pop(&stack));
    Display(&stack);
    printf("Popped Element : %d\n",  Pop(&stack));
    Display(&stack);
    printf("Popped Element : %d\n",  Pop(&stack));
    Display(&stack);
    printf("Popped Element : %d\n",  Pop(&stack));
    Display(&stack);
    Pop(&stack);
    Display(&stack);
}
```

Code Output :

```
|  10| ← TOP
———————

|  20| ← TOP
|  10|
———————

|  30| ← TOP
|  20|
|  10|
———————

|  40| ← TOP
|  30|
|  20|
|  10|
———————

|  50| ← TOP
|  40|
|  30|
|  20|
|  10|
———————

Stack Overflow

Popped Element : 50
|  40| ← TOP
|  30|
|  20|
|  10|
———————
```

```
Popped Element : 40
|  30| ← TOP
|  20|
|  10|
 ‾‾‾‾‾

Popped Element : 30
|  20| ← TOP
|  10|
 ‾‾‾‾‾

Popped Element : 20
|  10| ← TOP
 ‾‾‾‾‾

Popped Element : 10
|Empty|

Stack Underflow

|Empty|
```

Conclusion :

This program demonstrates the basic operations of a stack, including creation, push, pop, and display. The stack is implemented using a fixed-size array, ensuring simplicity and clarity. It handles edge cases like stack overflow and underflow with appropriate messages, making it robust. The modular design and straightforward functionality make it an excellent introduction to stack data structures and their use in C programming.

Teachers' Signature

Assignment - 04

Date : __________

Objective :

Write a program in C to take a string of brackets (all 3 types) as input, and check whether it is balanced or not. Show proper error messages.

Algorithm :

Algorithm **CheckBracketsBalance**:
Input : a stack STACK, pointer to top TOP, a string of brackets EXP
Output : Result Message if expression is balanced or not
Data Structure : stack implemented with array

```
Start
I = 0, U = 0
While EXP[I] ≠ NULL Do
    If EXP[I] == ‘(‘ or EXP[I] == ‘{‘ or EXP[I] == ‘[‘ Then
        Push(STACK, TOP, EXP[I])
    Else If EXP[I] == ‘)‘ Then
        OPEN = Pop(STACK, TOP)
        If OPEN ≠ ‘(‘ Then
            U = 1
        EndIf
    Else If EXP[I] == ‘}‘ Then
        OPEN = Pop(STACK, TOP)
        If OPEN ≠ ‘{‘ Then
            U = 1
        EndIf
    Else If EXP[I] == ‘]‘ Then
        OPEN = Pop(STACK, TOP)
        If OPEN ≠ ‘[‘ Then
            U = 1
        EndIf
    Else
        Print “Invalid Character, Not a Bracket”
        Exit
    EndIf
    If U == 1 Then
        Print “Brackets are not Balanced”
        Exit
    EndIf
    I = I + 1
EndWhile
If ISEMPTY(STACK, TOP) Then
    Print “Brackets are Balanced”
```

```
33. Else
34.     Print “Brackets are not Balanced”
35. EndIf
36. Stop
```

Source Code :

```
#include <stdio.h>

void ExpressionBalanceChecker(const char* string){
    char stack[100], open;
    int top = -1;

    for(int i = 0; string[i] != '\0'; i++){
        char c = string[i];
        int unbalanced = 0;
        switch(c){
            case '(':
            case '{':
            case '[':
                top++;
                stack[top] = c;
                break;
            case ')':
                open = stack[top--];
                if(open != '(') unbalanced = 1;
                break;
            case '}':
                open = stack[top--];
                if(open != '{') unbalanced = 1;
                break;
            case ']':
                open = stack[top--];
                if(open != '[') unbalanced = 1;
                break;
            default:
                printf("Error : Invalid character Encountered, Cannot determine balance!\n");
                return;
        }

        if(unbalanced){
            printf("Result : Brackets are not balanced!\n");
            return;
        }
    }

    if(top != -1){
        printf("Result : Brackets are not balanced!\n");
    }else{
        printf("Result : The Brackets are balanced!\n");
    }
```

```
}

int main(){
    char buffer[100];
    printf("**********Bracket Checker*********");
    printf("\nInput a string containing Brackets and check if its balanced or not :\nEnter String :
");
    scanf("%[^\n]s", buffer);

    ExpressionBalanceChecker(buffer);
}
```

Code Output :

```
**********Bracket Checker*********
Input a string containing Brackets and check if its balanced or not :
Enter String : (({[()]({})}))
Result : The Brackets are balanced!


**********Bracket Checker*********
Input a string containing Brackets and check if its balanced or not :
Enter String : (({])
Result : Brackets are not balanced!
```

Conclusion :

This program checks whether a given string containing brackets ((), {}, []) is balanced. It uses a stack to match opening and closing brackets, ensuring proper nesting and order. The program handles invalid characters gracefully and reports any unbalanced conditions. Its modular approach and clear logic make it an effective solution for understanding stack-based algorithms in C.

Teachers' Signature

Assignment - 05

Date : __________

Objective :

Write a menu driven program in C to perform insertion and deletion operations in a queue. Check for exception conditions.

Algorithm :

Algorithm **Enqueue**:
Input : a queue QUEUE, pointer to front F and rear R, item to be inserted ITEM
Output : queue with newly inserted item
Data Structure : linear queue implemented with array with MAX capacity

```
Start
If R == MAX - 1 Then
    Print “Queue FULL”
    Exit
Else
    If F == -1 Then
        F = 0
    EndIf
    R = R + 1
    QUEUE[R] = ITEM
EndIf
Stop
```

Algorithm **Dequeue**:
Input : a queue QUEUE, pointer to front F and rear R
Output : queue with deleted item in ITEM
Data Structure : linear queue implemented with array with MAX capacity

```
Start
If F == -1 Then
    Print “Queue EMPTY”
    Exit
Else
    ITEM = QUEUE[F]
    If F == R Then
        F = -1
        R = -1
    Else
        F = F + 1
    EndIf
EndIf
```

14. Stop

Source Code :

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 3

typedef struct queue{
    int queue[MAX];
    int front, rear;
} Queue;

Queue CreateQueue(){
    Queue q;
    q.front = -1;
    q.rear = -1;
    return q;
}

int Enqueue(Queue* q, int data){
    if(q->rear == MAX - 1){
        printf("EXCEPTION : Queue Full! Insertion Not Possible\n");
        return 0;
    }else{
        if(q->front == -1) q->front++;
        q->rear++;
        q->queue[q->rear] = data;
        return 1;
    }
}

int Dequeue(Queue* q){
    if(q->front == -1){
        printf("EXCEPTION : Queue Empty! Deletion Not Possible\n");
        return INT_MIN;
    }else{
        int data = q->queue[q->front];
        if(q->front == q->rear){
            q->front = -1;
            q->rear = -1;
        }else{
            q->front++;
        }
        return data;
    }
}

void Display(Queue* q){
    if(q->front == -1){
```

```c
        printf("Empty Queue\n");
    }else{
        printf("F -> ");
        for(int i = q->front; i <= q->rear; i++){
            printf("%d ", q->queue[i]);
        }
        printf("<- R\n");
    }
}

int main(){
    printf("************ Queue Operations **********\n");
    int choice, temp;
    Queue queue = CreateQueue();
    while(1){
        printf("\nMENU [max size %d] -\n1. Insert\n2. Delete\n3. Display\n4. Exit\n\nSelect
Operation : ", MAX);
        scanf("%d", &choice);

        switch(choice){
            case 1:
                printf("Enter Data : ");
                scanf("%d", &temp);
                if(Enqueue(&queue, temp)){
                    printf("INFO : Succesfully Inserted Data (%d)\n", temp);
                }
                break;
            case 2:
                temp = Dequeue(&queue);
                if(temp != INT_MIN){
                    printf("INFO : Succesfully Deleted Data (%d)\n", temp);
                }
                break;
            case 3:
                Display(&queue);
                break;
            case 4:
                printf("INFO : Program Exit!\n");
                exit(0);
            default:
                printf("ERROR : Invalid Operation Selected!");
        }
    }
}
```

Code Output :

```
************ Queue Operations **********

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 45
INFO : Succesfully Inserted Data (45)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 45 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 66
INFO : Succesfully Inserted Data (66)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 45 66 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 2
INFO : Succesfully Deleted Data (45)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 66 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 78
INFO : Succesfully Inserted Data (78)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 89
EXCEPTION : Queue Full! Insertion Not Possible

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 66 78 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 4
INFO : Program Exit!
```

Conclusion :

The program implements a linear queue using a static array, supporting operations such as enqueue (to add an element to the rear of the queue), dequeue (to remove an element from the front), and display (to print the current queue elements from front to rear). Exceptional cases like attempting to enqueue into a full queue or dequeue from an empty queue are handled gracefully, making the program robust and straightforward for demonstrating queue functionality.

However, the linear queue has some limitations. It suffers from wastage of space because once an element is dequeued, the freed space cannot be reused, even if the queue is not full. The fixed size of the array restricts the maximum number of elements, leading to potential overflow if the queue is too small or underutilization if it is too large. Additionally, inefficient memory utilization occurs as memory remains allocated for unused portions of the array when the queue appears empty (front > rear). Lastly, a manual reset is required to reuse the queue after it becomes "full" due to the linear progression of front and rear.

Teachers’ Signature

Assignment - 06

Date : __________

Objective :

Write a program in C to implement the standard circular queue operations (i.e. Insert, Delete, Display) using an array.

Algorithm :

Algorithm **Enqueue**:
Input : a circular queue CQUEUE, pointer to front F and rear R, item to be inserted ITEM
Output : queue with newly inserted item
Data Structure : circular queue implemented with array with MAX capacity

```
Start
If F == (R + 1) Mod MAX Then
    Print “Queue FULL”
    Exit
Else
    If F == -1 Then
        F = 0
    EndIf
    R = (R + 1) Mod MAX
    CQUEUE[R] = ITEM
EndIf
Stop
```

Algorithm **Dequeue**:
Input : a circular queue CQUEUE, pointer to front F and rear R
Output : queue with deleted item in ITEM
Data Structure : circular queue implemented with array with MAX capacity

```
Start
If F == -1 Then
    Print “Queue EMPTY”
    Exit
Else
    ITEM = CQUEUE[F]
    If F == R Then
        F = -1
        R = -1
    Else
        F = (F + 1) Mod MAX
    EndIf
EndIf
```

14. Stop

Source Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 3

typedef struct queue{
    int queue[MAX];
    int front, rear;
} Queue;

Queue CreateQueue(){
    Queue q;
    q.front = -1;
    q.rear = -1;
    return q;
}

int Enqueue(Queue* q, int data){
    if(q->front == (q->rear + 1) % MAX){
        printf("EXCEPTION : Queue Full! Insertion Not Possible\n");
        return 0;
    }else{
        if(q->front == -1) q->front++;
        q->rear = (q->rear + 1) % MAX;
        q->queue[q->rear] = data;
        return 1;
    }
}

int Dequeue(Queue* q){
    if(q->front == -1){
        printf("EXCEPTION : Queue Empty! Deletion Not Possible\n");
        return INT_MIN;
    }else{
        int data = q->queue[q->front];
        if(q->front == q->rear){
            q->front = -1;
            q->rear = -1;
        }else{
            q->front = (q->front + 1) % MAX;
        }
        return data;
    }
}

void Display(Queue* q){
    if(q->front == -1){
```

```
        printf("Empty Queue\n");
    }else{
        printf("F -> ");
        if(q->rear < q->front){
            for(int i = q->front; i < MAX; i++) printf("%d ", q->queue[i]);
            for(int i = 0; i <= q->rear; i++) printf("%d ", q->queue[i]);
        }else{
            for(int i = q->front; i <= q->rear; i++) printf("%d ", q->queue[i]);
        }
        printf("<- R\n");
    }
}

int main(){
    printf("************ Queue Operations **********\n");
    int choice, temp;
    Queue queue = CreateQueue();
    while(1){
        printf("\nMENU [max size %d] -\n1. Insert\n2. Delete\n3. Display\n4. Exit\n\nSelect
Operation : ", MAX);
        scanf("%d", &choice);

        switch(choice){
            case 1:
                printf("Enter Data : ");
                scanf("%d", &temp);
                if(Enqueue(&queue, temp)){
                    printf("INFO : Succesfully Inserted Data (%d)\n", temp);
                }
                break;
            case 2:
                temp = Dequeue(&queue);
                if(temp != INT_MIN){
                    printf("INFO : Succesfully Deleted Data (%d)\n", temp);
                }
                break;
            case 3:
                Display(&queue);
                break;
            case 4:
                printf("INFO : Program Exit!\n");
                exit(0);
            default:
                printf("ERROR : Invalid Operation Selected!");
        }
    }
}
```

Code Output :

```
************ Queue Operations **********

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 77
INFO : Succesfully Inserted Data (77)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 77 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 56
INFO : Succesfully Inserted Data (56)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 77 56 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 2
INFO : Succesfully Deleted Data (77)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 56 ← R

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 89
INFO : Succesfully Inserted Data (89)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 44
INFO : Succesfully Inserted Data (44)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 1
Enter Data : 78
EXCEPTION : Queue Full! Insertion Not Possible

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 3
F → 56 89 44 ← R
```

```
MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 2
INFO : Succesfully Deleted Data (56)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 2
INFO : Succesfully Deleted Data (89)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 2
INFO : Succesfully Deleted Data (44)

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 2
EXCEPTION : Queue Empty! Deletion Not Possible

MENU [max size 3] -
1. Insert
2. Delete
3. Display
4. Exit

Select Operation : 4
INFO : Program Exit!
```

Conclusion :

The program demonstrates the implementation of a circular queue using a static array, effectively utilizing the circular property to make full use of the allocated space. The operations include enqueue, which adds an element to the rear of the queue, dequeue, which removes an element from the front, and display, which traverses and prints the queue contents regardless of whether the elements are sequential or wrapped around. The circular queue ensures efficient memory utilization and avoids unused spaces, making it a reliable and effective solution for managing data in a queue structure.

Teachers' Signature