**Exp No. 2**

## Title:

Implementation of Linear Queue ADT and Circular Queue ADT using Array

## Aim:

To implement (a) Linear Queue ADT using array and (b) Circular Queue ADT using array, and demonstrate basic queue operations such as Enqueue, Dequeue, Peek, and Display.

## Objectives:

• Understand queue concepts and FIFO (First In, First Out) principle.  
• Implement Linear Queue using static array with front and rear pointers.  
• Implement Circular Queue using static array with modulo arithmetic to efficiently utilize space.  
• Compare limitations of Linear Queue vs. benefits of Circular Queue.

## Theory:

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. Two primary operations are: Enqueue (insert at rear) and Dequeue (remove from front). A Linear Queue implemented with a static array suffers from the 'false overflow' problem when front advances; even if there is free space at the beginning, rear may reach the array end. A Circular Queue treats the array as circular using modulo arithmetic so that after the last index, the next index is 0, thus efficiently reusing freed slots.

## Algorithms:

Algorithm A: Linear Queue using Array

1. Initialize front = -1, rear = -1, MAX = size of array.  
2. isEmpty(): return (front == -1 || front > rear).  
3. isFull(): return (rear == MAX - 1).  
4. Enqueue(x): if isFull() -> Overflow; else if front == -1 set front = 0; set queue[++rear] = x.  
5. Dequeue(): if isEmpty() -> Underflow; else return queue[front++] (optionally reset front, rear to -1 when front > rear).  
6. Peek(): if isEmpty() -> no element; else return queue[front].  
7. Display(): print elements from front to rear.

Algorithm B: Circular Queue using Array

1. Initialize front = -1, rear = -1, MAX = size of array.  
2. isEmpty(): return (front == -1).  
3. isFull(): return ((rear + 1) % MAX == front).  
4. Enqueue(x): if isFull() -> Overflow; else if isEmpty() set front = rear = 0; else rear = (rear + 1) % MAX; set cq[rear] = x.  
5. Dequeue(): if isEmpty() -> Underflow; else x = cq[front]; if front == rear set front = rear = -1; else front = (front + 1) % MAX; return x.  
6. Peek(): if isEmpty() -> no element; else return cq[front].  
7. Display(): traverse from front to rear using modulo until you wrap back.

## Program Code A (C Language): Linear Queue using Array

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 5 // Fixed size for the queue

int queue[MAX\_SIZE];

int front = -1;

int rear = -1;

// Function to check if the queue is empty

bool is\_empty() {

    return (front == -1 || front > rear);

}

// Function to check if the queue is full (False Overflow can occur here)

bool is\_full() {

    return (rear == MAX\_SIZE - 1);

}

// Function to insert an element at the rear (Enqueue)

void enqueue(int data) {

    if (is\_full()) {

        printf("Overflow! Cannot enqueue %d. Queue is full.\n", data);

    } else {

        if (front == -1) {

            front = 0; // Initialize front when the first element is added

        }

        queue[++rear] = data;

        printf("Enqueued: %d\n", data);

    }

}

// Function to remove an element from the front (Dequeue)

int dequeue() {

    if (is\_empty()) {

        printf("Underflow! Cannot dequeue. Queue is empty.\n");

        return -1; // Sentinel value for error

    } else {

        int data = queue[front++];

        // Check if the queue became empty after dequeue

        if (front > rear) {

            front = -1;

            rear = -1;

        }

        return data;

    }

}

// Function to get the front element without removing it (Peek)

int peek() {

    if (is\_empty()) {

        printf("No element to peek. Queue is empty.\n");

        return -1; // Sentinel value for error

    } else {

        return queue[front];

    }

}

// Function to display all elements in the queue

void display() {

    if (is\_empty()) {

        printf("Queue: Empty\n");

    } else {

        printf("Linear Queue: ");

        for (int i = front; i <= rear; i++) {

            printf("%d ", queue[i]);

        }

        printf("\n");

    }

}

int main() {

    printf("--- Linear Queue Operations ---\n");

    // Sample Operations

    enqueue(10);

    enqueue(20);

    enqueue(30);

    display(); // Output: 10 20 30

    int dequeued\_val = dequeue();

    if (dequeued\_val != -1) {

        printf("Dequeued: %d\n", dequeued\_val); // Output: 10

    }

    int peek\_val = peek();

    if (peek\_val != -1) {

        printf("Front: %d\n", peek\_val); // Output: 20

    }

    enqueue(40);

    enqueue(50);

    enqueue(60); // This will cause Overflow (MAX\_SIZE is 5, front is 1, rear is 4, 0-indexing)

    display(); // Output: 20 30 40 50

    return 0;

}

## Program Code B (C Language): Circular Queue using Array

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 5 // Fixed size for the circular queue

int cq[MAX\_SIZE];

int front = -1;

int rear = -1;

// Function to check if the circular queue is empty

bool is\_empty\_cq() {

    return (front == -1);

}

// Function to check if the circular queue is full

bool is\_full\_cq() {

    return ((rear + 1) % MAX\_SIZE == front);

}

// Function to insert an element at the rear (Enqueue)

void enqueue\_cq(int data) {

    if (is\_full\_cq()) {

        printf("Overflow! Cannot enqueue %d. Circular Queue is full.\n", data);

    } else {

        if (is\_empty\_cq()) {

            front = rear = 0; // Initialize front and rear for the first element

        } else {

            rear = (rear + 1) % MAX\_SIZE; // Advance rear circularly

        }

        cq[rear] = data;

        printf("Enqueued: %d\n", data);

    }

}

// Function to remove an element from the front (Dequeue)

int dequeue\_cq() {

    if (is\_empty\_cq()) {

        printf("Underflow! Cannot dequeue. Circular Queue is empty.\n");

        return -1; // Sentinel value for error

    } else {

        int data = cq[front];

        if (front == rear) {

            front = rear = -1; // Reset when the last element is removed

        } else {

            front = (front + 1) % MAX\_SIZE; // Advance front circularly

        }

        return data;

    }

}

// Function to get the front element without removing it (Peek)

int peek\_cq() {

    if (is\_empty\_cq()) {

        printf("No element to peek. Circular Queue is empty.\n");

        return -1; // Sentinel value for error

    } else {

        return cq[front];

    }

}

// Function to display all elements in the circular queue

void display\_cq() {

    if (is\_empty\_cq()) {

        printf("Circular Queue: Empty\n");

        return;

    }

    printf("Circular Queue: ");

    int i = front;

    do {

        printf("%d ", cq[i]);

        if (i == rear) break; // Stop after printing the last element

        i = (i + 1) % MAX\_SIZE;

    } while (i != (rear + 1) % MAX\_SIZE);

    printf("\n");

}

int main() {

    printf("--- Circular Queue Operations ---\n");

    // Sample Operations

    enqueue\_cq(1);

    enqueue\_cq(2);

    enqueue\_cq(3);

    enqueue\_cq(4);

    display\_cq(); // Output: 1 2 3 4

    int dequeued\_val = dequeue\_cq();

    if (dequeued\_val != -1) {

        printf("Dequeued: %d\n", dequeued\_val); // Output: 1

    }

    enqueue\_cq(5); // Now the space left by '1' is reused

    display\_cq(); // Output: 2 3 4 5

    enqueue\_cq(6); // This will cause Overflow (Size 5, max elements 5: 2, 3, 4, 5, 6)

    dequeued\_val = dequeue\_cq();

    if (dequeued\_val != -1) {

        printf("Dequeued: %d\n", dequeued\_val); // Output: 2

    }

    enqueue\_cq(7); // Space left by '2' is reused

    display\_cq(); // Output: 3 4 5 6 7

    return 0;

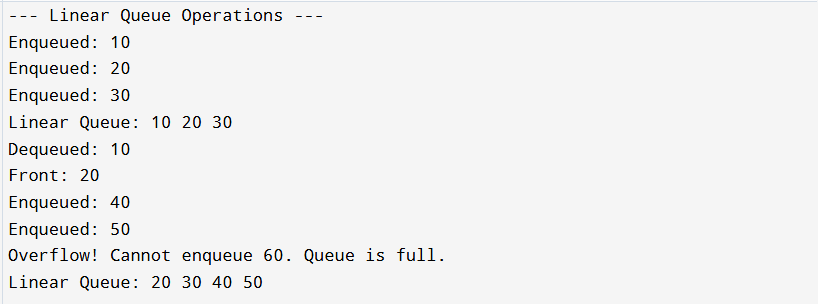
}

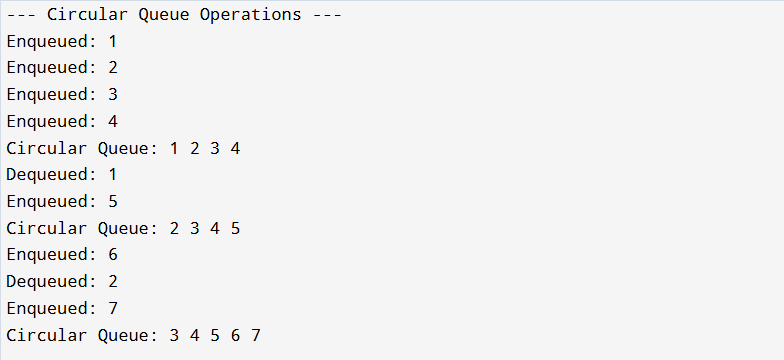
## Sample Input/Output:

Linear Queue Sample:  
Operations: Enqueue 10, Enqueue 20, Enqueue 30, Dequeue, Peek, Display  
Output:  
Enqueued: 10  
Enqueued: 20  
Enqueued: 30  
Dequeued: 10  
Front: 20  
Queue: 20 30  
  
Circular Queue Sample:  
Operations: Enqueue 1, Enqueue 2, Enqueue 3, Enqueue 4, Dequeue, Enqueue 5, Display  
Output:  
Enqueued: 1  
Enqueued: 2  
Enqueued: 3  
Enqueued: 4  
Dequeued: 1  
Enqueued: 5  
Circular Queue: 2 3 4 5

## Result:

Successfully implemented Linear Queue and Circular Queue ADTs using arrays and demonstrated queue operations with illustrative examples.





## Conclusion:

Linear Queue suffers from false overflow due to fixed rear movement, whereas Circular Queue overcomes this by wrapping indices using modulo arithmetic, enabling efficient space utilization in fixed-size arrays.

## Post-Lab Problem : Queue-based Ticket Counter Simulator

Objective:

Use your queue implementations to simulate a simple ticket counter where customers arrive and are served in FIFO order.

Description:

Write a program that accepts a stream of commands:  
• A x -> A new customer with token x arrives (enqueue x)  
• S -> Serve the next customer (dequeue and print token)  
• P -> Print/peek the current customer at the front  
• D -> Display the queue  
Implement this using both Linear Queue and Circular Queue to observe behavior when capacity is reached.

Input:

First line: N (number of commands). Next N lines: commands as described (e.g., A 101, S, P, D). Assume MAX = 5.

Output:

For each command, print the outcome (e.g., "Enqueued 101", "Served 101", "Front 102", "Queue: ...", "Overflow", "Underflow").

Constraints:

• Use your array-based queue ADTs only (no STL/collections).  
• Number of commands N ≤ 50.  
• Token values are positive integers.  
• For Linear Queue, show Overflow when rear reaches MAX-1.  
• For Circular Queue, show Overflow when (rear+1)%MAX == front.

Sample I/O:

Input:  
7  
A 10  
A 20  
A 30  
S  
A 40  
A 50  
D  
  
Expected Output (Circular Queue):  
Enqueued 10  
Enqueued 20  
Enqueued 30  
Served 10  
Enqueued 40  
Enqueued 50  
Circular Queue: 20 30 40 50

Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define MAX\_SIZE 5 // Constraint: MAX = 5

int cq[MAX\_SIZE];

int cq\_front = -1;

int cq\_rear = -1;

bool cq\_is\_empty() {

    return (cq\_front == -1);

}

bool cq\_is\_full() {

    return ((cq\_rear + 1) % MAX\_SIZE == cq\_front);

}

void cq\_enqueue(int token) {

    if (cq\_is\_full()) {

        printf("Overflow! Customer token %d denied (Queue full).\n", token);

    } else {

        if (cq\_is\_empty()) {

            cq\_front = cq\_rear = 0;

        } else {

            cq\_rear = (cq\_rear + 1) % MAX\_SIZE;

        }

        cq[cq\_rear] = token;

        printf("Enqueued %d\n", token);

    }

}

int cq\_dequeue() {

    if (cq\_is\_empty()) {

        printf("Underflow! No customer to serve (Queue empty).\n");

        return -1;

    } else {

        int token = cq[cq\_front];

        if (cq\_front == cq\_rear) {

            cq\_front = cq\_rear = -1; // Reset

        } else {

            cq\_front = (cq\_front + 1) % MAX\_SIZE;

        }

        return token;

    }

}

void cq\_peek() {

    if (cq\_is\_empty()) {

        printf("Front: Queue empty\n");

    } else {

        printf("Front %d\n", cq[cq\_front]);

    }

}

void cq\_display() {

    if (cq\_is\_empty()) {

        printf("Circular Queue: Empty\n");

        return;

    }

    printf("Circular Queue: ");

    int i = cq\_front;

    do {

        printf("%d ", cq[i]);

        if (i == cq\_rear) break;

        i = (i + 1) % MAX\_SIZE;

    } while (true);

    printf("\n");

}

int main() {

    int N;

    char command[5]; // Sufficient for 'A x' or 'S\0'

    int token\_value;

    printf("--- Ticket Counter Simulator (Circular Queue ADT) ---\n");

    printf("Enter number of commands (N <= 50): ");

    if (scanf("%d", &N) != 1 || N <= 0) {

        fprintf(stderr, "Invalid N.\n");

        return 1;

    }

    printf("Enter N commands (A x, S, P, D):\n");

    for (int i = 0; i < N; i++) {

        if (scanf("%s", command) != 1) {

            break; // Stop on read error

        }

        if (strcmp(command, "A") == 0) {

            if (scanf("%d", &token\_value) == 1) {

                cq\_enqueue(token\_value); // Use cq\_enqueue

            } else {

                fprintf(stderr, "Invalid 'A' command format.\n");

                break;

            }

        } else if (strcmp(command, "S") == 0) {

            int served\_token = cq\_dequeue(); // Use cq\_dequeue

            if (served\_token != -1) {

                printf("Served %d\n", served\_token);

            }

        } else if (strcmp(command, "P") == 0) {

            cq\_peek(); // Use cq\_peek

        } else if (strcmp(command, "D") == 0) {

            cq\_display(); // Use cq\_display

        } else {

            printf("Unknown command: %s\n", command);

        }

    }

    return 0;

}

Actual Output:

