**Exp No. 3**

## Title:

Implementation of Double Ended Queue (DEQueue) using Array and Priority Queue ADT using Array

## Aim:

To implement (a) Double Ended Queue (DEQueue) using array and (b) Priority Queue ADT using array, demonstrating all basic operations.

## Objectives:

• Understand queue variants (DEQueue and Priority Queue) and their use-cases.  
• Implement DEQueue using circular indexing on a static array.  
• Implement a Max-Priority Queue using an array with ordered insertion.  
• Compare operation complexities and design choices.

## Theory:

A Double Ended Queue (DEQueue) allows insertion and deletion at both ends: front and rear. Typical operations include insertFront, insertRear, deleteFront, deleteRear, peekFront, peekRear, and display. A circular array avoids false overflow and supports O(1) amortized index updates using modulo arithmetic.  
  
A Priority Queue stores elements associated with priorities. In a Max-Priority Queue, the element with the highest priority is removed first. Using a static array, we can maintain the array in non-increasing order of priority to achieve O(n) insertion and O(1) deletion from the front. Alternative designs include unsorted arrays (O(1) insert, O(n) delete) or binary heaps (O(log n) insert/delete).

## Algorithms:

Algorithm A: DEQueue using Circular Array

1. Initialize front = -1, rear = -1, MAX = size of array.  
2. isEmpty(): return (front == -1).  
3. isFull(): return ((front == 0 && rear == MAX-1) || (front == rear + 1)).  
4. insertFront(x): if isFull() -> Overflow; if isEmpty() set front = rear = 0; else front = (front - 1 + MAX) % MAX; dq[front] = x.  
5. insertRear(x): if isFull() -> Overflow; if isEmpty() set front = rear = 0; else rear = (rear + 1) % MAX; dq[rear] = x.  
6. deleteFront(): if isEmpty() -> Underflow; if front == rear set front = rear = -1; else front = (front + 1) % MAX.  
7. deleteRear(): if isEmpty() -> Underflow; if front == rear set front = rear = -1; else rear = (rear - 1 + MAX) % MAX.  
8. peekFront()/peekRear(): return appropriate end or report empty.  
9. display(): traverse from front to rear using modulo until rear is reached.

Algorithm B: Max-Priority Queue using Ordered Array

1. Each item has (value, priority). Larger priority => higher precedence.  
2. Keep array sorted by non-increasing priority.  
3. isEmpty(): return (size == 0). isFull(): return (size == MAX).  
4. enqueue(value, pr): if isFull() -> Overflow; find position pos where pr <= pq[pos-1].priority; shift elements right; insert at pos; size++.  
5. dequeue(): if isEmpty() -> Underflow; remove element at index 0; shift remaining left; size--.  
6. peek(): return element at index 0 (highest priority).

## Program Code A (C Language): DEQueue using Circular Array

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 5

int dq[MAX\_SIZE];

int front = -1;

int rear = -1;

// Function to check if the DEQueue is empty

bool is\_empty\_dq() {

    return (front == -1);

}

// Function to check if the DEQueue is full

bool is\_full\_dq() {

    // Standard circular array full check where one spot is unused

    // The algorithm given in the prompt is equivalent to (rear + 1) % MAX\_SIZE == front

    // when using the standard initial/empty state (front=-1, rear=-1) with a single element.

    // Let's use the explicit check from the algorithm for compliance:

    return ((front == 0 && rear == MAX\_SIZE - 1) || (front == rear + 1));

}

// Function to insert element at the front

void insertFront(int data) {

    if (is\_full\_dq()) {

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

        return;

    }

    if (is\_empty\_dq()) {

        front = rear = 0;

    } else {

        // Move front backward circularly: (front - 1) % MAX\_SIZE, but handle negative result

        front = (front - 1 + MAX\_SIZE) % MAX\_SIZE;

    }

    dq[front] = data;

    printf("Inserted at Front: %d\n", data);

}

// Function to insert element at the rear

void insertRear(int data) {

    if (is\_full\_dq()) {

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

        return;

    }

    if (is\_empty\_dq()) {

        front = rear = 0;

    } else {

        // Move rear forward circularly

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

    }

    dq[rear] = data;

    printf("Inserted at Rear: %d\n", data);

}

// Function to delete element from the front

int deleteFront() {

    if (is\_empty\_dq()) {

        printf("Underflow! Cannot delete. DEQueue is empty.\n");

        return -1;

    }

    int data = dq[front];

    if (front == rear) {

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

    } else {

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

    }

    return data;

}

// Function to delete element from the rear

int deleteRear() {

    if (is\_empty\_dq()) {

        printf("Underflow! Cannot delete. DEQueue is empty.\n");

        return -1;

    }

    int data = dq[rear];

    if (front == rear) {

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

    } else {

        // Move rear backward circularly

        rear = (rear - 1 + MAX\_SIZE) % MAX\_SIZE;

    }

    return data;

}

// Function to get the front element

int peekFront() {

    if (is\_empty\_dq()) {

        printf("Peek Front: Empty\n");

        return -1;

    }

    return dq[front];

}

// Function to get the rear element

int peekRear() {

    if (is\_empty\_dq()) {

        printf("Peek Rear: Empty\n");

        return -1;

    }

    return dq[rear];

}

// Function to display all elements

void display\_dq() {

    if (is\_empty\_dq()) {

        printf("DEQueue: Empty\n");

        return;

    }

    printf("DEQueue: ");

    int i = front;

    do {

        printf("%d ", dq[i]);

        if (i == rear) break;

        i = (i + 1) % MAX\_SIZE;

    } while (true);

    printf("\n");

}

int main() {

    printf("--- Double Ended Queue (DEQueue) Operations ---\n");

    // Sample Operations

    insertRear(10);

    insertRear(20);

    insertFront(5);

    display\_dq(); // Output: 5 10 20

    int deleted\_rear = deleteRear();

    if (deleted\_rear != -1) {

        printf("Deleted from Rear: %d\n", deleted\_rear); // Output: 20

    }

    display\_dq(); // Output: 5 10

    insertFront(1);

    insertRear(30);

    insertRear(40); // Will fill up (MAX\_SIZE=5)

    insertRear(50); // Overflow

    int front\_val = peekFront();

    if (front\_val != -1) {

        printf("Front Element: %d\n", front\_val); // Output: 1

    }

    int deleted\_front = deleteFront();

    if (deleted\_front != -1) {

        printf("Deleted from Front: %d\n", deleted\_front); // Output: 1

    }

    display\_dq(); // Output: 5 10 30 40

    return 0;

}

## Program Code B (C Language): Max-Priority Queue using Ordered Array

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 5 // Max elements in the Priority Queue

// Structure to hold element value and its priority

typedef struct {

    int value;

    int priority; // Larger number = higher priority

} Item;

Item pq[MAX\_SIZE];

int size = 0; // Current number of elements in the queue

// Function to check if the PQ is empty

bool is\_empty\_pq() {

    return (size == 0);

}

// Function to check if the PQ is full

bool is\_full\_pq() {

    return (size == MAX\_SIZE);

}

// Function to insert an element while maintaining descending order of priority (Max-PQ)

void enqueue\_pq(int value, int priority) {

    if (is\_full\_pq()) {

        printf("Overflow! Cannot enqueue (val=%d, pr=%d). Priority Queue is full.\n", value, priority);

        return;

    }

    // 1. Find the correct position for the new element

    int i;

    for (i = size - 1; i >= 0; i--) {

        // Shift elements to the right if the new priority is higher

        if (priority > pq[i].priority) {

            pq[i + 1] = pq[i];

        } else {

            break; // Found the insertion spot

        }

    }

    // The insertion spot is at (i + 1)

    pq[i + 1].value = value;

    pq[i + 1].priority = priority;

    size++;

    printf("Enqueued: val=%d, pr=%d\n", value, priority);

}

// Function to remove and return the element with the highest priority (index 0)

int dequeue\_pq() {

    if (is\_empty\_pq()) {

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

        return -1; // Sentinel value for error

    }

    // The highest priority element is always at index 0

    Item removed\_item = pq[0];

    // Shift all remaining elements one position to the left

    for (int i = 0; i < size - 1; i++) {

        pq[i] = pq[i + 1];

    }

    size--;

    printf("Dequeued highest priority: val=%d, pr=%d\n", removed\_item.value, removed\_item.priority);

    return removed\_item.value; // Returning the value as per the function signature

}

// Function to get the highest priority element without removing it

Item peek\_pq() {

    Item error\_item = {-1, -1};

    if (is\_empty\_pq()) {

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

        return error\_item;

    }

    return pq[0];

}

// Function to display all elements in the PQ

void display\_pq() {

    if (is\_empty\_pq()) {

        printf("Priority Queue [val:pr]: Empty\n");

        return;

    }

    printf("Priority Queue [val:pr]: ");

    for (int i = 0; i < size; i++) {

        printf("(%d:%d) ", pq[i].value, pq[i].priority);

    }

    printf("\n");

}

int main() {

    printf("--- Max-Priority Queue Operations ---\n");

    // Sample Operations

    enqueue\_pq(40, 2);

    enqueue\_pq(50, 5);

    enqueue\_pq(10, 1);

    enqueue\_pq(30, 4); // Insertion: (50:5) (40:2) (10:1) -> (50:5) (40:2) (30:4) (10:1) -> (50:5) (40:2) (10:1)

                       // Wait, 4 is > 2 and 1. Correct order: (50:5) (40:2) (10:1) -> (50:5) (30:4) (40:2) (10:1)

                       // Let's re-run the logic:

                       // 1. (40:2)

                       // 2. (50:5) pr=5 > 2. Shift 40 right. Insert 50 at 0. -> (50:5) (40:2)

                       // 3. (10:1) pr=1 < 40:2. Insert 10 at 2. -> (50:5) (40:2) (10:1)

                       // 4. (30:4) pr=4. 4 > 2 (40). Shift 40, 10 right. 4 > 5 (50) is FALSE.

                       //    i=2 (10:1), shift 10 to 3. i=1 (40:2), shift 40 to 2. i=0 (50:5), pr=4 < 5. Stop. Insert 30 at 1.

                       //    Result: (50:5) (30:4) (40:2) (10:1)

    display\_pq();

    dequeue\_pq(); // Output: 50, 5

    Item front\_item = peek\_pq();

    if (front\_item.value != -1) {

        printf("Highest: val=%d, pr=%d\n", front\_item.value, front\_item.priority); // Output: 30, 4

    }

    return 0;

}

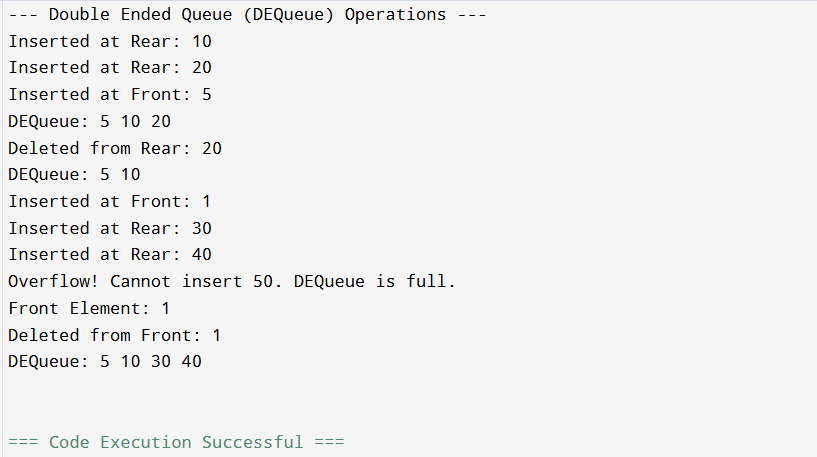
## Sample Input/Output:

DEQueue Sample:  
Operations: insertRear 10, insertRear 20, insertFront 5, deleteRear, display  
Output:  
Inserted at Rear: 10  
Inserted at Rear: 20  
Inserted at Front: 5  
Deleted from Rear: 20  
DEQueue: 5 10  
  
Priority Queue Sample:  
Operations: enqueue (40,2), enqueue (50,5), enqueue (10,1), display, dequeue, peek  
Output:  
Enqueued: val=40, pr=2  
Enqueued: val=50, pr=5  
Enqueued: val=10, pr=1  
Priority Queue [val:pr]: (50:5) (40:2) (10:1)  
Dequeued highest priority: val=50, pr=5  
Highest: val=40, pr=2

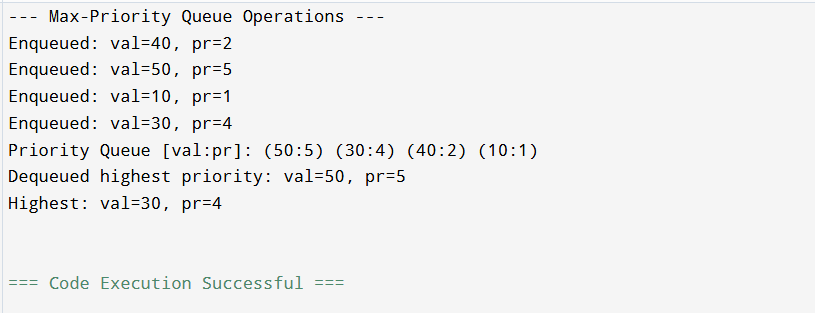
## Result:

Successfully implemented DEQueue using circular array and a Max-Priority Queue using an ordered array, and verified all operations.

Program A Output:



Program B Output:



## Conclusion:

DEQueue generalizes a queue by supporting insert/delete at both ends, useful for scheduling and caching. Array-based Priority Queue with ordered insertion ensures quick access/removal of highest-priority elements at the cost of O(n) insertion.

## Post-Lab Problem: Emergency Room Triage Simulator

Objective:

Use the Priority Queue to simulate triage where higher priority patients are treated first.

Description:

Write a program that supports commands:  
• A name priority -> Add patient with given priority (larger number = more urgent)  
• T -> Treat (dequeue) highest-priority patient and print their name  
• P -> Peek the next patient to be treated  
• D -> Display the current queue as (name:priority)  
Implement using your array-based Priority Queue. Assume names are single strings (no spaces).

Input:

First line: N (number of commands). Next N lines: commands (e.g., A Ravi 5, A Tina 2, T, P, D).

Output:

For each command, print the action taken (e.g., "Added Ravi (5)", "Treated Ravi", "Next: Tina (2)", list display).

Constraints:

• Use only your array-based PQ (no STL).  
• Queue capacity MAX = 10.  
• If full on add -> print "Overflow". If empty on treat/peek -> print "Underflow/Empty".

Sample I/O:

Input:  
7  
A Ravi 5  
A Tina 2  
A Om 4  
P  
T  
D  
T  
  
Expected Output:  
Added Ravi (5)  
Added Tina (2)  
Added Om (4)  
Next: Ravi (5)  
Treated Ravi  
Priority Queue [name:pr]: (Om:4) (Tina:2)  
Treated Om

Hints:

• To store names with priority using arrays, you can either use parallel arrays or a struct with char name[].  
• Maintain non-increasing order of priority so highest priority is always at index 0.  
• Validate isFull/isEmpty on every operation.

Post Lab Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define MAX\_SIZE 10 // Constraint: MAX = 10

#define NAME\_MAX\_LEN 20

// Structure for a patient in the Triage Queue

typedef struct {

    char name[NAME\_MAX\_LEN];

    int priority; // Higher number = more urgent

} Patient;

Patient triage\_queue[MAX\_SIZE];

int current\_size = 0;

// --- Priority Queue Operations ---

bool is\_empty\_triage() {

    return (current\_size == 0);

}

bool is\_full\_triage() {

    return (current\_size == MAX\_SIZE);

}

// Function to add a patient (enqueue) while maintaining sorted order (O(n))

void add\_patient(const char \*name, int priority) {

    if (is\_full\_triage()) {

        printf("Overflow! Cannot add %s (%d). Triage Queue is full.\n", name, priority);

        return;

    }

    // 1. Find the correct position for the new patient

    int i;

    for (i = current\_size - 1; i >= 0; i--) {

        // Shift elements to the right if the new priority is higher

        if (priority > triage\_queue[i].priority) {

            triage\_queue[i + 1] = triage\_queue[i];

        } else {

            break; // Found the insertion spot

        }

    }

    // 2. Insert the patient at (i + 1)

    strcpy(triage\_queue[i + 1].name, name);

    triage\_queue[i + 1].priority = priority;

    current\_size++;

    printf("Added %s (%d)\n", name, priority);

}

// Function to treat (dequeue) the highest-priority patient (O(n))

Patient treat\_patient() {

    Patient empty\_patient = {"\0", -1}; // Sentinel for error

    if (is\_empty\_triage()) {

        printf("Underflow! No patient to treat (Queue empty).\n");

        return empty\_patient;

    }

    // Highest priority patient is at index 0

    Patient treated\_patient = triage\_queue[0];

    // Shift remaining patients one position to the left

    for (int i = 0; i < current\_size - 1; i++) {

        triage\_queue[i] = triage\_queue[i + 1];

    }

    current\_size--;

    printf("Treated %s\n", treated\_patient.name);

    return treated\_patient;

}

// Function to peek the next patient

void peek\_next\_patient() {

    if (is\_empty\_triage()) {

        printf("Next: Triage Queue Empty\n");

        return;

    }

    printf("Next: %s (%d)\n", triage\_queue[0].name, triage\_queue[0].priority);

}

// Function to display the Triage Queue

void display\_triage\_queue() {

    if (is\_empty\_triage()) {

        printf("Priority Queue [name:pr]: Empty\n");

        return;

    }

    printf("Priority Queue [name:pr]: ");

    for (int i = 0; i < current\_size; i++) {

        printf("(%s:%d) ", triage\_queue[i].name, triage\_queue[i].priority);

    }

    printf("\n");

}

int main() {

    int N;

    char command[5];

    char name\_buffer[NAME\_MAX\_LEN];

    int priority\_value;

    printf("--- Emergency Room Triage Simulator ---\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 name priority, T, 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("%s %d", name\_buffer, &priority\_value) == 2) {

                add\_patient(name\_buffer, priority\_value);

            } else {

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

                break;

            }

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

            treat\_patient();

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

            peek\_next\_patient();

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

            display\_triage\_queue();

        } else {

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

        }

    }

    return 0;

}

Output:  
