LAB 3

Experiment 1

OBJECTIVE: To perform various operations in a queue using array implementation.

THEORY:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. The element inserted first is removed first. The main operations on a queue are:

1. Enqueue (Insertion): Adds an element to the rear of the queue.
2. Dequeue (Deletion): Removes an element from the front of the queue.
3. Display: Shows all elements in the queue.
4. isEmpty: Checks if the queue is empty.
5. isFull: Checks if the queue is full.

Queue Representation in Array

- front points to the first element.
- rear points to the last inserted element.
- If rear == MAX - 1, the queue is full.
- If front > rear or front == -1, the queue is empty.

ALGORITHM:

Enqueue Operation:

1. Check if the queue is full.
2. If not full, increment rear and insert the element.
3. If inserting the first element, set front = 0.

Dequeue Operation:

1. Check if the queue is empty.
2. If not empty, print and remove the front element.
3. Increment front.

Display Operation:

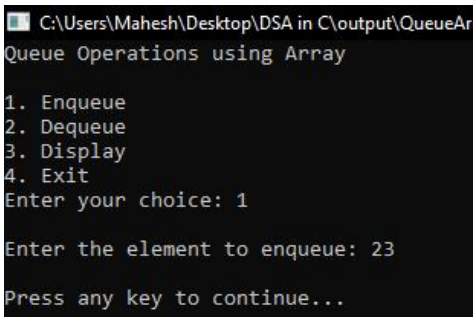
1. If empty, print "Queue is empty".
2. Otherwise, print elements from front to rear.

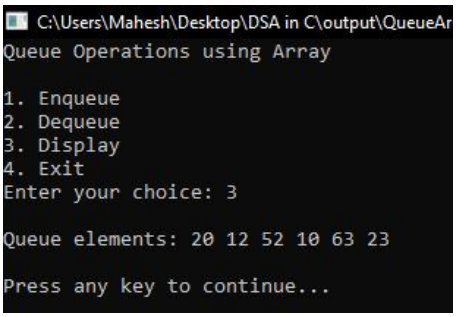
PROGRAMS

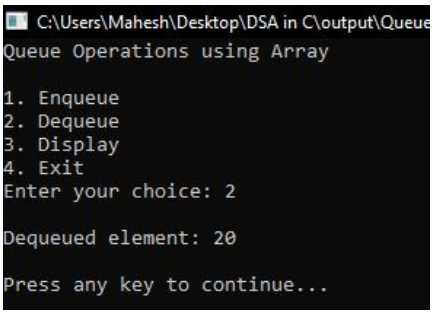
```
#include<stdio.h>
#include<conio.h>
#define MAX 10

int queue[MAX], front = -1, rear = -1;

int isFull() {
    return rear == MAX - 1;
}
int isEmpty() {
    return front == -1 || front > rear;
}
void enqueue(int item) {
    if (isFull()) {
        printf("Queue is full\n");
    }
    else {
        if (front == -1) front = 0;
        rear++;
        queue[rear] = item;
    }
}
void dequeue() {
```

```
    if (isEmpty()) {
        printf("Queue is empty\n");
    }
    else {
        printf("Dequeued element: %d\n",
queue[front]);
        front++;
    }
}
void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    }
    else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, item;

    while (1) {
        system("cls");
        printf("Queue Operations using
Array\n");
        printf("\n1. Enqueue\n2. Dequeue\n3.
Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice) {
        case 1:
            printf("Enter the element to enqueue:
");
            scanf("%d", &item);
            enqueue(item);
            break;
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
    getch();
}
```

Output:







CONCLUSION:

In this lab, we implemented queue operations using an array. The enqueue and dequeue functions worked as expected, following FIFO order. This demonstrates how queues can be efficiently managed using arrays.

Experiment 2

OBJECTIVE:

To perform various operations on a Queue with Linked List implementation.

THEORY:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. The element inserted first is removed first. The main operations on a queue are:

1. Enqueue (Insertion): Adds an element to the rear of the queue.
2. Dequeue (Deletion): Removes an element from the front of the queue.
3. Display: Shows all elements in the queue.
4. isEmpty: Checks if the queue is empty.

Queue Representation using Linked List

- The queue is implemented using a linked list where each node contains data and a pointer to the next node.
- front points to the first node in the queue.
- rear points to the last node in the queue.
- If front == NULL, the queue is empty.

Algorithm

Enqueue Operation:

1. Create a new node.
2. If the queue is empty, set front = rear = new node.
3. Else, set rear->next = new node and update rear.

Dequeue Operation:

1. Check if the queue is empty.
2. If not empty, print and remove the front node.
3. Update front to the next node.

Display Operation:

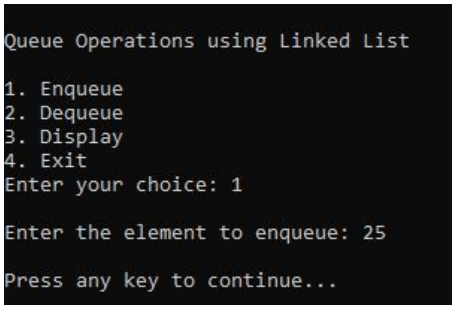
1. If empty, print "Queue is empty".
2. Otherwise, traverse from front to rear and print elements.

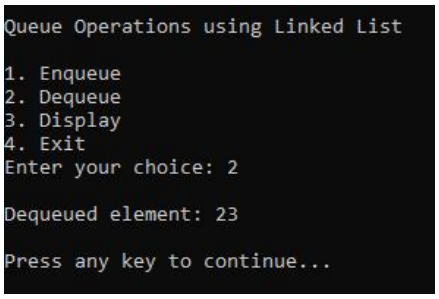
PROGRAMS

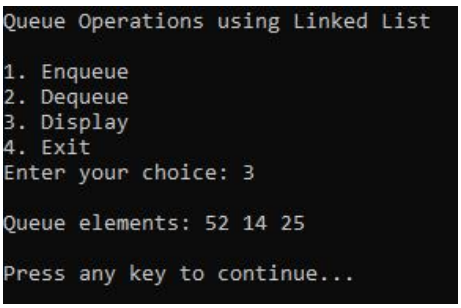
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

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

typedef struct Node Node;
Node* front = NULL;
Node* rear = NULL;

int isEmpty() {
    return front == NULL;
}

void enqueue(int item) {
    Node* newNode =
(Node*)malloc(sizeof(Node));
    newNode->data = item;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
```

```
        rear = newNode;
    }
}

void dequeue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    Node* temp = front;
    printf("Dequeued element: %d\n", front->data);
    front = front->next;
    if (front == NULL) rear = NULL;
    free(temp);
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, item;
    while (1) {
        system("cls");
        printf("\nQueue Operations using Linked List\n\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice) {
        case 1:
            printf("Enter the element to enqueue: ");
            scanf("%d", &item);
            enqueue(item);
            break;
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
}
```

Output:







CONCLUSION:

In this lab, we implemented queue operations using a linked list. The enqueue and dequeue functions worked as expected, following FIFO order. This demonstrates how queues can be efficiently managed using linked lists.

Experiment 3

OBJECTIVE:

To perform various operations in a circular queue using array implementation.

THEORY:

A circular queue is a linear data structure that follows the First In, First Out (FIFO) principle but connects the end of the queue back to the front to utilize unused space efficiently.

Features of Circular Queue:

1. Efficient Space Utilization: Unlike a linear queue, it does not waste space after elements are dequeued.
2. Circular Structure: The rear wraps around when it reaches the last index.
3. Key Operations:
   - Enqueue: Adds an element at the rear.
   - Dequeue: Removes an element from the front.
   - Display: Shows all elements in the queue.
   - isFull: Checks if the queue is full.
   - isEmpty: Checks if the queue is empty.

Algorithm

Enqueue Operation:

1. Check if the queue is full.
2. If empty, set front = 0.
3. Increment rear circularly using (rear + 1) % MAX.
4. Insert the new element at rear.

Dequeue Operation:

1. Check if the queue is empty.
2. Print and remove the front element.
3. If only one element was left, reset front and rear to -1.
4. Otherwise, update front using (front + 1) % MAX.

Display Operation:

1. If empty, print "Queue is empty".
2. Traverse from front to rear circularly and print elements.

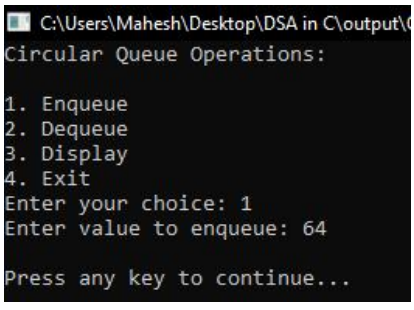
PROGRAMS

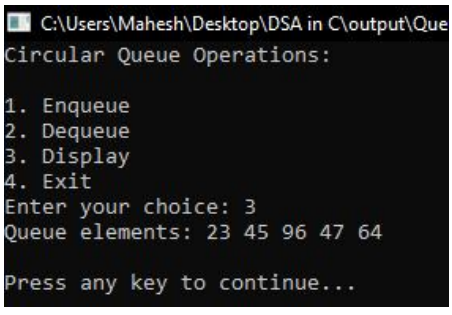
```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define MAX 10

int queue[MAX], front = -1, rear = -1;

int isFull() {
    return (front == (rear + 1) % MAX);
}

int isEmpty() {
    return (front == -1);
}

void enqueue(int value) {
    if (isFull()) {
        printf("Queue is full\n");
        return;
    }
    if (isEmpty()) {
        front = 0;
```

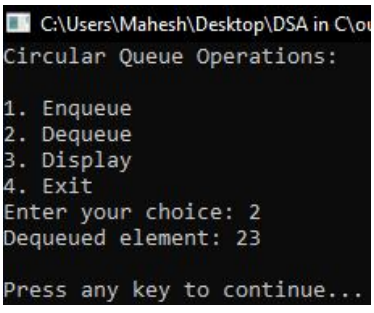
```
    }
    rear = (rear + 1) % MAX;
    queue[rear] = value;
}

void dequeue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Dequeued element: %d\n",
queue[front]);
    if (front == rear) {
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % MAX;
    }
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i != rear; i = (i + 1) %
MAX) {
        printf("%d ", queue[i]);
    }
    printf("%d\n", queue[rear]); // Print the last
element
}
```

```
int main() {
    int choice, value;
    while (1) {
        system("cls");
        printf("Circular Queue
Operations:\n\n");
        printf("1. Enqueue\n2. Dequeue\n3.
Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to enqueue:
");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
}
```

Output:







CONCLUSION:

In this lab, we implemented a circular queue using an array. The enqueue and dequeue operations worked efficiently, demonstrating the advantage of circular queues in avoiding wasted space compared to linear queues.

Experiment 4

OBJECTIVE: To perform various operations in a priority queue, implementing both Min and Max Priority Queues using arrays.

THEORY:

A Priority Queue is a special type of queue in which elements are inserted based on their priority. The two types of priority queues are:

1. Min Priority Queue: The element with the lowest value has the highest priority and is dequeued first.
2. Max Priority Queue: The element with the highest value has the highest priority and is dequeued first.

Key Operations:

- Enqueue: Inserts an element in the correct position based on priority.
- Dequeue: Removes the highest (or lowest) priority element.
- Display: Shows elements in priority order.
- isEmpty: Checks if the queue is empty.

Algorithm

Enqueue (Insertion):

- Insert the element at the rear.
- Sort the queue based on priority (ascending for Min Queue, descending for Max Queue).

Dequeue (Deletion):

- Remove the first element (highest priority).

Display:

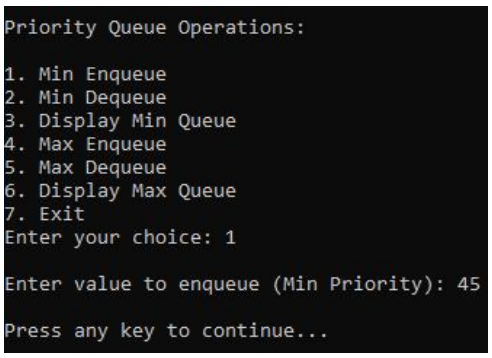
- Print elements from front to rear.

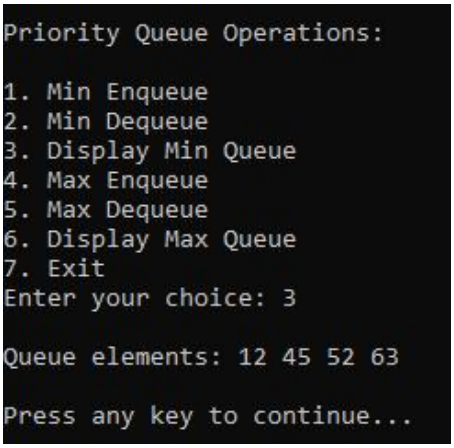
PROGRAMS

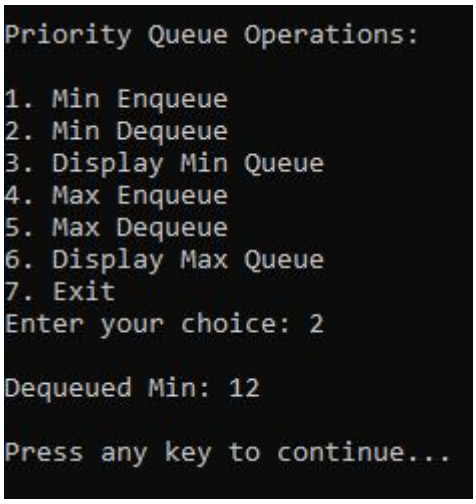
```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 10

int minQueue[MAX], maxQueue[MAX];
int minFront = -1, minRear = -1;
int maxFront = -1, maxRear = -1;

int isEmpty(int front) {
    return front == -1;
}

void minEnqueue(int value) {
    if (minRear == MAX - 1) {
        printf("Min Priority Queue is full\n");
        return;
    }
    if (isEmpty(minFront)) {
        minFront = 0;
    }
    minRear++;
    minQueue[minRear] = value;
    for (int i = minRear; i > minFront &&
minQueue[i] < minQueue[i - 1]; i--) {
        int temp = minQueue[i];
        minQueue[i] = minQueue[i - 1];
        minQueue[i - 1] = temp;
    }
}

void minDequeue() {
    if (isEmpty(minFront)) {
        printf("Min Priority Queue is empty\n");
        return;
    }
    printf("Dequeued Min: %d\n",
minQueue[minFront]);
    if (minFront == minRear) minFront =
minRear = -1;
```

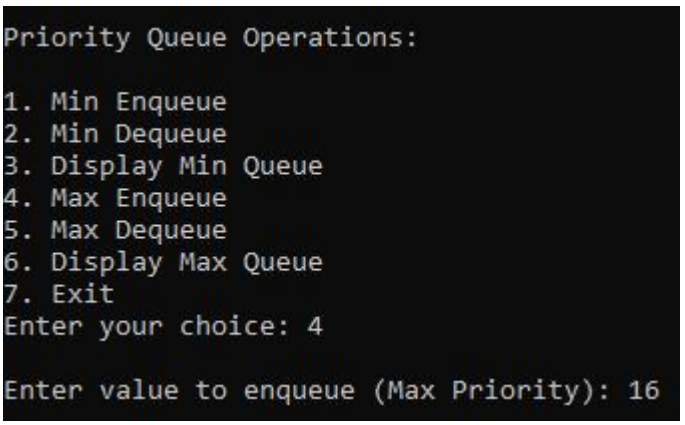
```
    else minFront++;
}

void maxEnqueue(int value) {
    if (maxRear == MAX - 1) {
        printf("Max Priority Queue is full\n");
        return;
    }
    if (isEmpty(maxFront)) {
        maxFront = 0;
    }
    maxRear++;
    maxQueue[maxRear] = value;
    for (int i = maxRear; i > maxFront &&
maxQueue[i] > maxQueue[i - 1]; i--) {
        int temp = maxQueue[i];
        maxQueue[i] = maxQueue[i - 1];
        maxQueue[i - 1] = temp;
    }
}

void maxDequeue() {
    if (isEmpty(maxFront)) {
        printf("Max Priority Queue is empty\n");
        return;
    }
    printf("Dequeued Max: %d\n",
maxQueue[maxFront]);
    if (maxFront == maxRear) maxFront =
maxRear = -1;
    else maxFront++;
}

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

int main() {
    int choice, value;
    while (1) {
        system("cls");
        printf("\nPriority Queue
Operations:\n\n");
        printf("1. Min Enqueue\n2. Min
Dequeue\n3. Display Min Queue\n");
        printf("4. Max Enqueue\n5. Max
Dequeue\n6. Display Max Queue\n7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        printf("\n");
        switch (choice) {
            case 1:
                printf("Enter value to enqueue
(Min Priority): ");
                scanf("%d", &value);
                minEnqueue(value);
                break;
            case 2:
                minDequeue();
                break;
            case 3:
                display(minQueue, minFront,
minRear);
                break;
            case 4:
                printf("Enter value to enqueue
(Max Priority): ");
                scanf("%d", &value);
                maxEnqueue(value);
                break;
            case 5:
                maxDequeue();
                break;
            case 6:
                display(maxQueue, maxFront,
maxRear);
                break;
            case 7:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
        printf("\nPress any key to continue...\n");
        getch();
    }
    return 0;
}
```

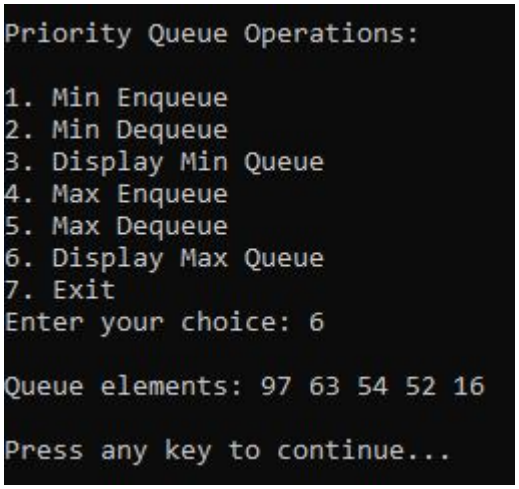
Output:

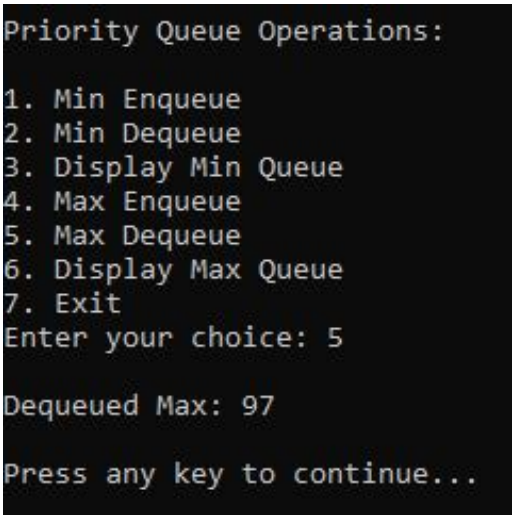












CONCLUSION:

In this lab, we implemented a Priority Queue using arrays. The Min Priority Queue removes the smallest element first, while the Max Priority Queue removes the largest element first. This approach is useful in scheduling and real-time processing applications.