

K. J. Somaiya College of Engineering, Mumbai
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: C1 Roll No.:16010122257
Experiment No. 5
Grade: AA / AB / BB / BC / CC / CD /DD

Batch: Roll No.:

Experiment / assignment / tutorial No.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Implementation of Basic operations on queue for the assigned application using Array and Linked List- Create, Insert, Delete, Destroy

Objective: To implement Basic Operations on Queue i.e. Create, Push, Pop, Destroy for the given application

Expected Outcome of Experiment:

CO	Outcome
1	Explain the different data structures used in problem solving

Books/ Journals/ Websites referred:

1. *Fundamentals Of Data Structures In C* – Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications* – Jean Paul Tremblay, Paul G. Sorenson
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan
- 4.

Abstract

(Define Queue, enlist queue operations).

Definition:

An ordered collection of homogenous data items

Where elements are added at rear and removed from the front end.

Operations:

Create an empty queue

check if it is empty and/or full

Enqueue:

add an element at the rear

Dequeue:

remove the element in front

Destroy : remove all the elements one by one and destroy the data structure

List 5 Real Life applications of Queue:

Print Queue in a printer.

Task scheduling in operating systems.

Call center systems to manage customer calls.

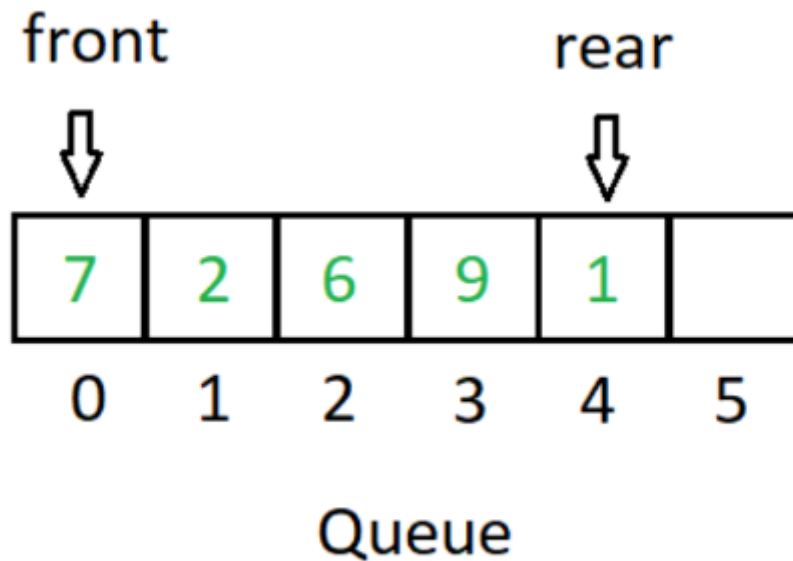
Breadth-first search algorithm in graph traversal.

Handling requests in web servers.

Define and explain various types of queue with suitable diagram and their application(s):

1) Simple queue- additions at rear and deletions from front

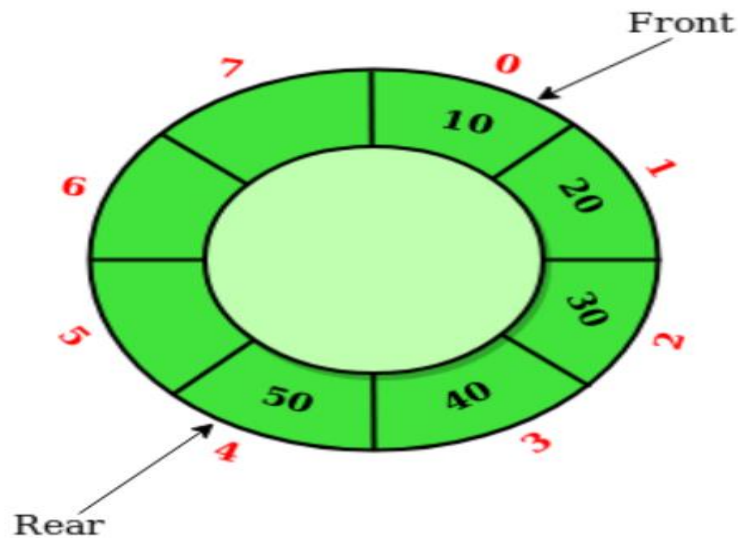
APPLICATIONS: process scheduling, disk scheduling, memory management, IO buffer, pipes, call center phone systems, and interrupt handling.



2) Circular queue- last node is connected to first node, deletions at front end while insertions are done at rear end.

APPLICATIONS: Memory management: circular queue is used in memory management.

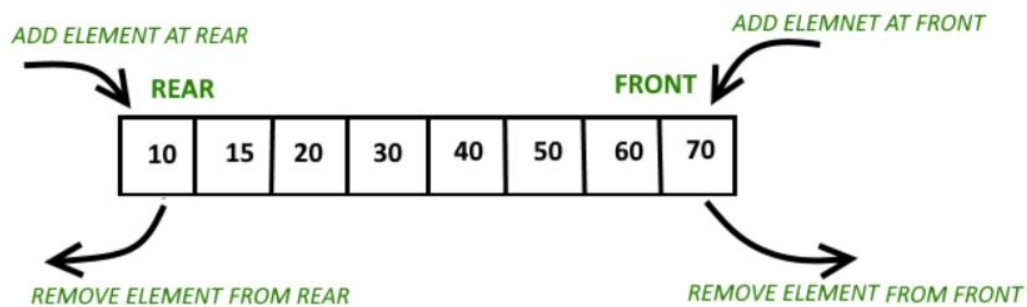
Traffic Systems: Queues are also used in traffic systems.



3) Doubly ended queue- deletions and insertions can be done at both the ends, has two pairs of fronts and rears, both.

APPLICATIONS: Job scheduling algorithm.

Storing a web browser's history.



4) Priority queue- every element has predefined priority

- Max priority : element with max priority is removed first
- min priority: element with min priority is removed first

K. J. Somaiya College of Engineering, Mumbai
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

APPLICATIONS: used in operating system like priority scheduling, load balancing and interrupt handling; used in heap sort.

Index	Front					Rear		
Data	10	5	3	98	12	36		
Priority	4	4	3	2	1	1		

Max Priority queue

Queue ADT:

Abstract typedef QueueType(ElementType ele)

Condition: none

1. Abstract QueueType CreateQueue()

Precondition: none

Postcondition: Empty Queue is created

2. Abstract QueueType Enqueue(QueueType Queue,
ElementType Element)

Precondition: Queue not full or NotFull(Queue)= True

Postcondition: Queue = Queue' + Element at the rear

Or Queue = original queue with new Element at the rear

3. Abstract ElementType dequeue(QueueType Queue)

Precondition: Queue not empty or NotEmpty(Queue)= True

Postcondition: Dequeue= element at the front

Queue= Queue - Element at the front

Or Queue = original queue with front element deleted

4. Abstract DestroyQueue(QueueType Queue)

Precondition: Queue not empty or NotEmpty(Queue)= True

Postcondition: Element from the Queue are removed one by one
starting from front to rear.

NotEmpty(Queue)= False

5. Abstract Boolean NotFull(QueueType Queue)

Precondition: none

Postcondition: NotFull(Queue)= true if Queue is not full

NotFull(Queue)= False if Queue is full.

6. Abstract Boolean NotEmpty(QueueType Queue)

Precondition: none

Postcondition: NotEmpty(Queue)= true if queue is not empty

NotEmpty(Queue)= False if Queue is empty.

Algorithm for Queue operations using array/Linked list : (Write only the algorithm for assigned type)

Struct NodeType{

ElementType Element;

NodeType *Next;

}

Algorithm QueueType CreateQueue()

//This Algorithm creates and returns an empty Queue, pointed by two pointers- front and rear

{ createNode(front);

```
createNode(rear);
```

```
Front=rear=NULL;
```

```
}
```

```
2. QueueType Enqueue(QueueType Queue, NodeType  
NewNode)
```

```
// This Algorithm adds a NewNode at the rear of 'queue'. rear is a pointer that points to  
the last node in the queue
```

```
{ if (front==rear==NULL) // first element in Queue
```

```
NewNode->Next = NULL;
```

```
front=NewNode;
```

```
rear=NewNode;
```

```
Else rear->Next=NewNode; // General case
```

```
rear=NewNode;
```

```
}
```


3. Algorithm ElementType DeQueue(QueueType Queue)

//This algorithm returns value of ElementType stored at the front of queue. Temp is a temporary node used in the dequeuer process.

```
{ if (front==rear==NULL)
```

```
    Print "Underflow"
```

```
    exit;
```

```
    Else
```

```
    {
```

```
        createNode(Temp);
```

```
        Temp=front;
```

```
        front= front->next;
```

```
        if (front=NULL)
```

```
            rear=NULL;
```

```
    Return(temp->Data);
```

}

}

4. Abstract DestroyQueue(QueueType Queue)

//This algorithm returns values stored in data structure and free the memory used in data

structure implementation.

{ { if front==NULL

Print "Underflow"

exit;

Else

// createNode(Temp);

while(NotEmpty(Queue))

{

return(Dequeue(Queue));

}

}

6. Abstract DisplayQueue(QueueType Queue)

//This algorithm Prints all the Elements stored in stack. Temp
purpose?

{ if front==NULL

Print "Error Message"

Else { createNode(Temp)

Temp=front;

While(Temp!=Null)

Print(Temp->Data);

Temp= Temp->next;

}

}

Implementation Details:

1) Mention the application assigned to you and explain how you implemented the solution using the assigned type of Queue.

I had to implement a simple queue using dynamic memory allocation in C.

- **CreateQueue:** We start by creating an empty queue by initializing pointers for the front and rear of the queue, setting the size of the queue, and initializing the count of elements to zero.
- **IsEmpty:** We implemented a function to check if the queue is empty by examining whether the front pointer is NULL.
- **IsFull:** Another function, IsFull, is used to check if the queue has reached its maximum size by comparing the count of elements to the size.
- **Enqueue:** The Enqueue function allows us to add elements to the queue. It checks if the queue is full before adding a new element to it. If the queue is not full, it dynamically allocates memory for a new node, stores the element, and adjusts the front and rear pointers accordingly.
- **Dequeue:** The Dequeue function removes elements from the front of the queue and returns their values. It also checks if the queue is empty before performing the dequeue operation. If the front and rear pointers coincide after dequeuing, the queue is empty.
- **DisplayQueue:** This function is used to display the elements in the queue, providing a visual representation of the current state of the queue.
- **DestroyQueue:** The DestroyQueue function removes all elements from the queue and frees the dynamically allocated memory, effectively destroying the queue.

Program source code:

LINKED LIST

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

```
#include <stdbool.h>
```

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

```
typedef int ElementType;
```

```
struct NodeType {  
    ElementType Element;  
    struct NodeType* Next;  
};
```

```
struct QueueType {  
    struct NodeType* front;  
    struct NodeType* rear;  
    int size;  
    int count;  
};
```

```
void CreateQueue(struct QueueType* queue, int size) {  
    queue->front = NULL;  
    queue->rear = NULL;  
    queue->size = size;  
    queue->count = 0;
```

```
}
```

```
bool IsEmpty(struct QueueType* queue) {  
    return (queue->front == NULL);  
}
```

```
bool IsFull(struct QueueType* queue) {  
    return (queue->count == queue->size);  
}
```

```
void Enqueue(struct QueueType* queue, ElementType item) {  
    if (IsFull(queue)) {  
        printf("OVERFLOW. Queue is full.\n");  
        return;  
    }  
    struct NodeType* newNode = (struct NodeType*)malloc(sizeof(struct NodeType));  
    if (newNode == NULL) {  
        printf("Memory allocation failed.\n");  
        exit(1);  
    }  
    newNode->Element = item;  
    newNode->Next = NULL;  
  
    if (IsEmpty(queue)) {  
        queue->front = newNode;
```

```
} else {  
    queue->rear->Next = newNode;  
}  
queue->rear = newNode;  
queue->count++;  
}
```

```
ElementType Dequeue(struct QueueType* queue) {  
    if (IsEmpty(queue)) {  
        printf("UNDERFLOW. Queue is empty.\n");  
        exit(1);  
    }  
    struct NodeType* temp = queue->front;  
    ElementType item = temp->Element;  
  
    if (queue->front == queue->rear) {  
        queue->front = NULL;  
        queue->rear = NULL;  
    } else {  
        queue->front = queue->front->Next;  
    }  
    free(temp);  
    queue->count--;  
    return item;  
}
```

```
void DisplayQueue(struct QueueType* queue) {  
    if (IsEmpty(queue)) {  
        printf("Queue is empty.\n");  
        return;  
    }  
    printf("Queue: ");  
    struct NodeType* temp = queue->front;  
    while (temp != NULL) {  
        printf("%d ", temp->Element);  
        temp = temp->Next;  
    }  
    printf("\n");  
}
```

```
void DestroyQueue(struct QueueType* queue) {  
    while (!IsEmpty(queue)) {  
        Dequeue(queue);  
    }  
}
```

```
int main() {  
    struct QueueType myQueue;  
    int choice, size;  
    ElementType item;
```



```
printf("Enter the size of the queue: ");
```

```
scanf("%d", &size);
```

```
CreateQueue(&myQueue, size);
```

```
printf("Queue Operations:\n");
```

```
printf("1. Enqueue\n");
```

```
printf("2. Dequeue\n");
```

```
printf("3. Display Queue\n");
```

```
printf("4. Destroy Queue\n");
```

```
printf("5. Exit\n");
```

```
while (1) {
```

```
    printf("Enter your choice: ");
```

```
    scanf("%d", &choice);
```

```
    if (choice == 1) {
```

```
        if (IsFull(&myQueue)) {
```

```
            printf("OVERFLOW.\n");
```

```
        } else {
```

```
            printf("Enter the element to enqueue: ");
```

```
            scanf("%d", &item);
```

```
            Enqueue(&myQueue, item);
```

```
        }
```

```
    } else if (choice == 2) {
```

```
if (!IsEmpty(&myQueue)) {  
    item = Dequeue(&myQueue);  
    printf("Dequeued element: %d\n", item);  
} else {  
    printf("UNDERFLOW.\n");  
}  
  
} else if (choice == 3) {  
    DisplayQueue(&myQueue);  
} else if (choice == 4) {  
    DestroyQueue(&myQueue);  
    printf("Queue destroyed.\n");  
} else if (choice == 5) {  
    printf("Exiting the program.\n");  
    return 0;  
} else {  
    printf("Invalid choice. Please try again.\n");  
}  
  
}  
  
return 0;  
}
```

ARRAY:

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

```
#include <stdbool.h>
```

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

```
typedef int ElementType;
```

```
struct QueueType {  
    ElementType* elements;  
    int front;  
    int rear;  
    int size;  
    int count;  
};
```

```
void CreateQueue(struct QueueType* queue, int size) {  
    queue->elements = (ElementType*)malloc(sizeof(ElementType) * size);  
    if (queue->elements == NULL) {  
        printf("Memory allocation failed.\n");  
        exit(1);  
    }  
    queue->front = 0;  
    queue->rear = -1;  
    queue->size = size;  
    queue->count = 0;
```

```
}
```

```
bool IsEmpty(struct QueueType* queue) {  
    return (queue->count == 0);  
}
```

```
bool IsFull(struct QueueType* queue) {  
    return (queue->count == queue->size);  
}
```

```
void Enqueue(struct QueueType* queue, ElementType item) {  
    if (IsFull(queue)) {  
        printf("OVERFLOW. Queue is full.\n");  
        return;  
    }  
    queue->rear = (queue->rear + 1) % queue->size;  
    queue->elements[queue->rear] = item;  
    queue->count++;  
}
```

```
ElementType Dequeue(struct QueueType* queue) {  
    if (IsEmpty(queue)) {  
        printf("UNDERFLOW. Queue is empty.\n");  
        exit(1);  
    }  
}
```

```
ElementType item = queue->elements[queue->front];  
  
queue->front = (queue->front + 1) % queue->size;  
  
queue->count--;  
  
return item;  
  
}
```

```
void DisplayQueue(struct QueueType* queue) {  
  
    if (IsEmpty(queue)) {  
  
        printf("Queue is empty.\n");  
  
        return;  
  
    }  
  
    printf("Queue: ");  
  
    int i = queue->front;  
  
    int elementsDisplayed = 0;  
  
    while (elementsDisplayed < queue->count) {  
  
        printf("%d ", queue->elements[i]);  
  
        i = (i + 1) % queue->size;  
  
        elementsDisplayed++;  
  
    }  
  
    printf("\n");  
  
}
```

```
void DestroyQueue(struct QueueType* queue) {  
  
    free(queue->elements);  
  
    queue->front = 0;
```

K. J. Somaiya College of Engineering, Mumbai
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

```
queue->rear = -1;

queue->size = 0;

queue->count = 0;
}

int main() {

    struct QueueType myQueue;

    int choice, size;

    ElementType item;

    printf("Enter the size of the queue: ");

    scanf("%d", &size);

    CreateQueue(&myQueue, size);

    printf("Queue Operations:\n");

    printf("1. Enqueue\n");

    printf("2. Dequeue\n");

    printf("3. Display Queue\n");

    printf("4. Destroy Queue\n");

    printf("5. Exit\n");

    while (1) {

        printf("Enter your choice: ");

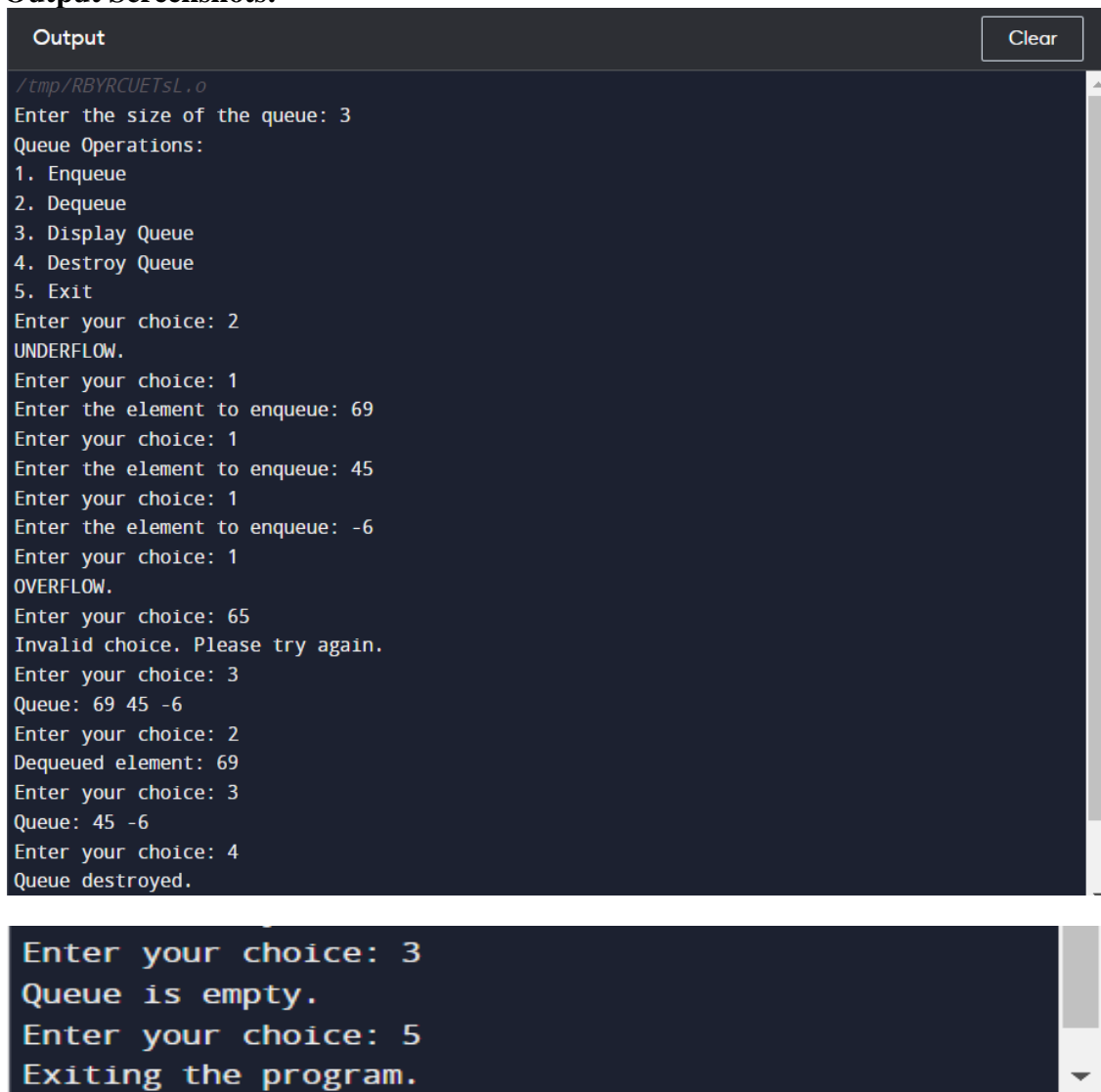
        scanf("%d", &choice);
```

```
if (choice == 1) {  
    if (IsFull(&myQueue)) {  
        printf("OVERFLOW\n");  
    } else {  
        printf("Enter the element to enqueue: ");  
        scanf("%d", &item);  
        Enqueue(&myQueue, item);  
    }  
} else if (choice == 2) {  
    if (!IsEmpty(&myQueue)) {  
        item = Dequeue(&myQueue);  
        printf("Dequeued element: %d\n", item);  
    } else {  
        printf("UNDERFLOW.\n");  
    }  
} else if (choice == 3) {  
    DisplayQueue(&myQueue);  
} else if (choice == 4) {  
    DestroyQueue(&myQueue);  
    printf("Queue destroyed.\n");  
} else if (choice == 5) {  
    printf("Exiting the program.\n");  
    return 0;  
} else {  
    printf("Invalid choice. Please try again.\n");  
}
```

K. J. Somaiya College of Engineering, Mumbai
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

```
    }  
}  
  
return 0;  
}
```

Output Screenshots:



```
Output  
/tmp/RBYRCUETsL.o  
Enter the size of the queue: 3  
Queue Operations:  
1. Enqueue  
2. Dequeue  
3. Display Queue  
4. Destroy Queue  
5. Exit  
Enter your choice: 2  
UNDERFLOW.  
Enter your choice: 1  
Enter the element to enqueue: 69  
Enter your choice: 1  
Enter the element to enqueue: 45  
Enter your choice: 1  
Enter the element to enqueue: -6  
Enter your choice: 1  
OVERFLOW.  
Enter your choice: 65  
Invalid choice. Please try again.  
Enter your choice: 3  
Queue: 69 45 -6  
Enter your choice: 2  
Dequeued element: 69  
Enter your choice: 3  
Queue: 45 -6  
Enter your choice: 4  
Queue destroyed.  
  
Enter your choice: 3  
Queue is empty.  
Enter your choice: 5  
Exiting the program.
```


Applications of Queue in computer science:

Queue data structures find applications in various computer science domains, including:

- Process scheduling in operating systems.
- Breadth-first search (BFS) algorithm in graph traversal.
- Handling requests in web servers.
- Managing tasks in parallel computing.
- Print spooling in printers.

Conclusion:-

In this experiment, we successfully implemented basic queue operations (Create, Enqueue, Dequeue, Display, Destroy) for a specific application. We also discussed the importance of queues in various real-life and computer science scenarios, highlighting their efficiency in managing and processing data in a FIFO manner. Understanding different data structures like queues is essential for efficient problem-solving.