UNIT- 4

Queues: Introduction to queues: properties and operations, implementing queues using arrays and linked lists, Applications of queues in breadth-first search, scheduling, etc. Deques: Introduction to deques (double-ended queues), Operations on deques and their applications.

1. Define Queue.

A Queue is an ordered list in which all insertions take place at one end called the rear, while all deletions take place at the other end called the front. Rear is initialized to -1 and front is initialized to 0. Queue is also referred as First In First Out (FIFO) list.

2. What are the various operations performed on the Queue?

The various operations performed on the queue are

CREATE(Q) – Creates Q as an empty Queue.

Enqueue(Q,X) – Adds the element X to the Queue.

Dequeue(Q) – Deletes a element from the Queue.

ISEMTPTY(Q) – returns true if Queue is empty else false.

ISFULL(Q) - returns true if Queue is full else false

3. How do you test for an empty Queue?

The condition for testing an empty queue is rear=front-1. In linked list implementation of queue the condition for an empty queue is the header node link field is NULL.

4. What is enqueue operation?

Enqueue - adding an element to the queue at the rear end If the queue is not full, this function adds an element to the back of the queue, else it prints “OverFlow”.

5. What is dequeue operation?

Dequeue – removing or deleting an element from the queue at the front end If the queue is not empty, this function removes the element from the front of the queue, else it prints “UnderFlow”.

6. What are the types of queue

The following are the types of queue: ● Double ended queue ● Circular queue ● Priority queue

7. Define Circular Queue.

Another representation of a queue, which prevents an excessive use of memory by arranging elements/ nodes Q1,Q2,…Qn in a circular fashion. That is, it is the queue, which wraps around upon reaching the end of the queue

8. Define Dequeue.

Deque stands for Double ended queue. It is a linear list in which insertions and deletion

are made from either end of the queue structure.

9. Distinguish between stack and queue.

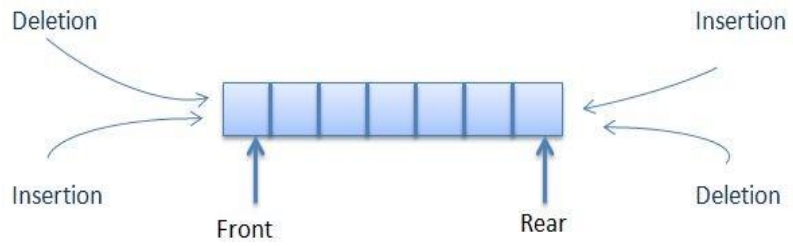
STACK	QUEUE
Insertion and deletion are made at one end.	Insertion at one end rear and deletion at other end front.
The element inserted last would be removed first. So LIFO structure.	The element inserted first would be removed first. So FIFO structure.
Full stack condition: If(top==Maxsize) Physically and Logically full stack	Full stack condition: If(rear = = Maxsize) Logically full. Physically may or may not be full.

10. How do you test for an empty queue?

To test for an empty queue, we must check whether REAR==HEAD where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that points to the dummy header. In the case of array implementation of queue, the condition to be checked for an empty queue is REAR<FRONT.

11. **Define double ended queue**

It is a special type of queue that allows insertion and deletion of elements at both Ends. It is also termed as DEQUE.



12. What are the methods to implement queue in C?

The methods to implement queues are: Array based, Linked list based

13. What are the applications of queue?

The following are the areas in which queues are applicable

a. Simulation

b. Batch processing in an operating system
c. Multiprogramming platform systems
d. Queuing theory
e. Printer server routines
f. Scheduling algorithms like disk scheduling , CPU scheduling
g. I/O buffer requests

14. What are the basic features of Queue?

1. Queue is an ordered list of elements of similar data types.
2. Queue is a FIFO (First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

1. What is queue and basic operations on queue?

A queue is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed. A queue is an Abstract Data Type (ADT). It is open at both its ends. The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages. Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: enqueue(), dequeue(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the queue. Queue uses two pointers − front and rear. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

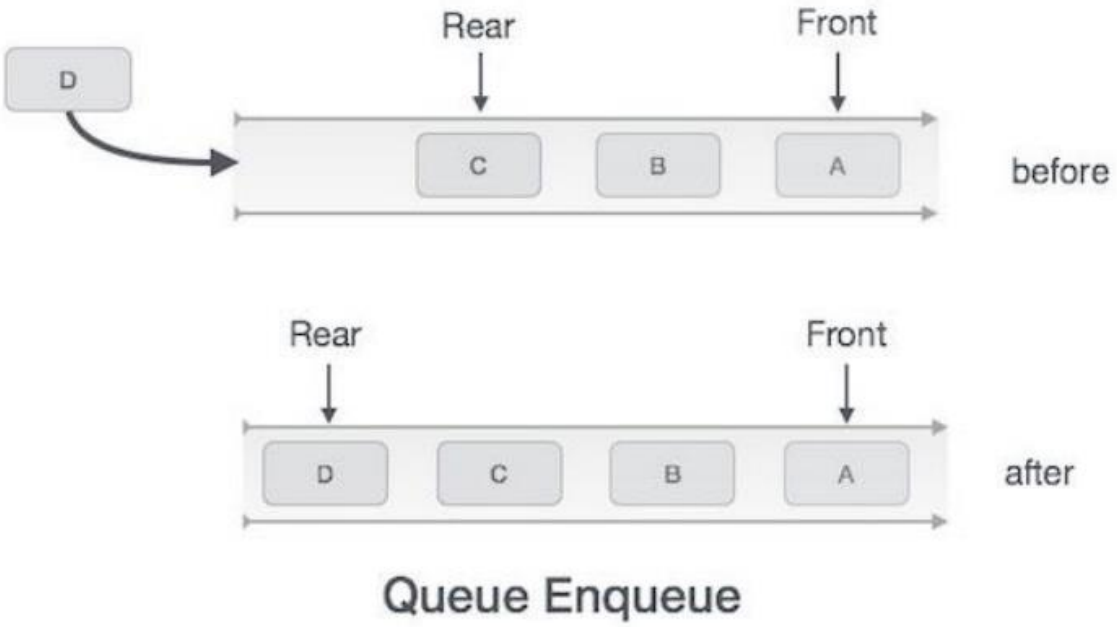
2. Write the algorithms for the insertion and deletion operations on queue.

Queue Insertion Operation: Enqueue()

The *enqueue()* is a data manipulation operation that is used to insert elements into the Queue. The following algorithm describes the **enqueue()** operation in a simpler way.

Algorithm

1. START
2. if(rear == length) Check if the queue is full .
3. If the queue is full, produce overflow error and exit.
4. rear++ (If the queue is not full, increment rear pointer to point the next empty space.)
5. q[rear] = item (Add data element to the queue location, where the rear is pointing.)
6. if(front==0) then front++
7. return success.
8. END



Queue Deletion Operation: dequeue()

The *dequeue()* is a data manipulation operation that is used to remove elements from the stack. The following algorithm describes the dequeue() operation in a simpler way.

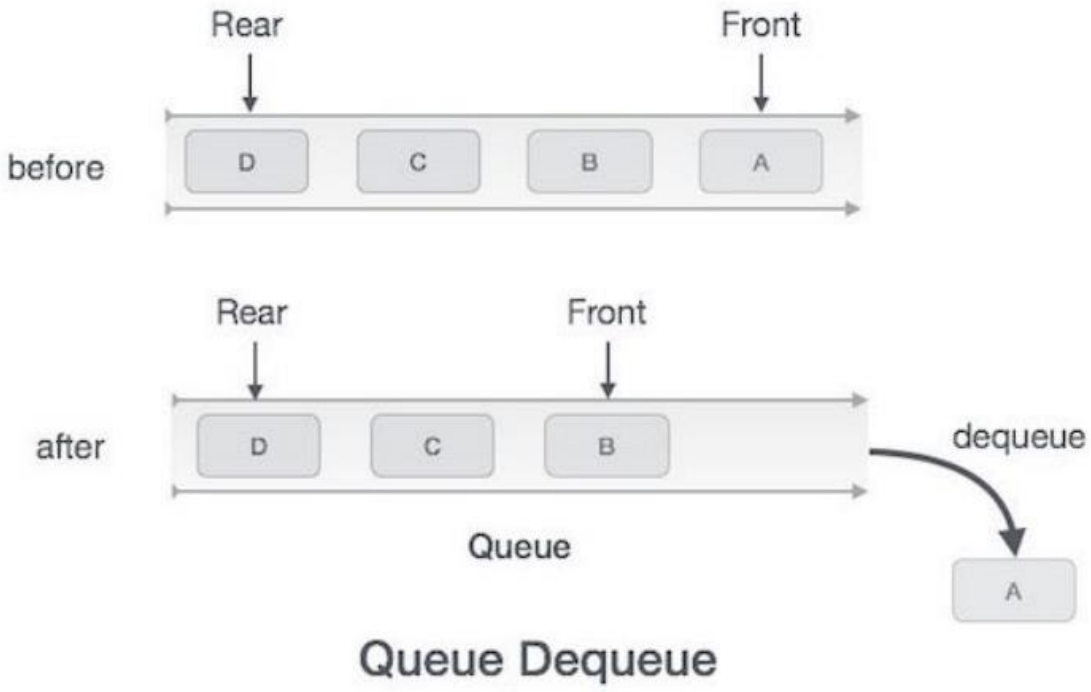
Algorithm

1. START
2. if((rear==0) and (front==0)) (Check if the queue is empty.)
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
4. item=q[front]
5. if (front==rear)
6. front=0, rear=0.
7. else

8. front ++ (Increment front pointer to point to the next available data element.)

9. Return success.

10. END



Queue Dequeue

3. Write the algorithms to check the status and to display the elements of the queue.

Algorithm for checking the status of the Queue

Steps:

1. if((rear==0) and (front==0))

2. print Queue is empty

3. else

4. if(rear==length)

5. print Queue is full

6. else

7. print no. of items in queue and free space available in queue(rear, length-rear)

8. end if

9. end if

10. stop

4. Write a c program to implement queue using array?

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

void enqueue();
void dequeue();
void display();
void status();
void front_element();
void rear_element();

int item, front=0, rear=0, length, q[50],i;

void main()
{

   int choice;
   printf("enter the length");
   scanf("%d", &length);

   while(1)
   {
      printf("\n Menu \n1.display \n2.enqueue \n3.dequeue\n4.Status \n5.first element \n5.last element \n7.exit ");

      printf("\nenter the choice: ");
      scanf("%d", &choice);
```

```
        switch(choice)
        {
           case 1: display();
                   break;

           case 2: enqueue();
                   break;

           case 3: dequeue();
                   break;

           case 4: status();
                   break;

           case 5: front_element();
                   break;

           case 6: rear_element();
                   break;

           case 7: exit(0);
        }
    }

}


void enqueue()
{
    if(rear ==length)
        printf("Queue is full. enqueue not possible");
    else{
        printf("enter the value");
        scanf("%d",&item);
        rear++;
        q[rear]=item;
        if(front==0)
            front++;
    }
}

void dequeue()
```

```
{
   if(front==0 && rear==0)
      printf("enque is empty. Dequeue not possible\n ");

else
{

   if(front==rear)
   {

      item=q[front];
      front=0;
      rear=0;

   }
   else
   {
      item=q[front];
      front++;

   }
   printf("the deleted item=%d", item);
}

}

void status()
{
   if(rear==0&&front==0)
      printf("queue is empty\n");

   else
   {
      if(rear==length)
         printf("queue is full\n");
      else
      {

         printf("the filled spaces=%d\n", rear );
         printf("the free spaces=%d\n", length-rear );
      }
   }
}
```

```
void display()
{
   i=front;
   while(i<=rear)
   {
      printf("\t%d", q[i]);
      i++;

   }
}

void front_element()
{
   if(front==0&&rear==0)
      printf("\n queue is empty");
   else
   {
      printf("\n First element is :%d", q[front]);
   }
}

void rear_element()
{
   if(front==0&&rear==0)
      printf("\n queue is empty");
   else
   {
      printf("\n Rear element is :%d", q[rear]);
   }
}
```

5. Write a c program to implement queue using Single Linked List?

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

struct node
{
   int data;
   struct node *next;
```

```
}*qheader, *front,*rear,*new, *ptr;

void creation();
void enqueue();
void dequeue();
void display();
void status();

int item;

void main()
{
   int choice;
   while(1)
   {
      printf("\n Menu \n1.creation \n2.enqueue \n3.dequeue\n4.Status \n5.Display \n6.exit ");
      printf("\nenter the choice: ");
      scanf("%d", &choice);

      switch(choice)
      {
         case 1: creation();
               break;

         case 2: enqueue();
               break;
         case 3: dequeue();
               break;
         case 4: status();
               break;
         case 5: display();
               break;
         case 6: exit(0);
               }
   }
}

void creation()
{
   qheader = (struct node*) malloc(sizeof(struct node));
   qheader ->data=NULL;
   qheader ->next=NULL;
   front=qheader;
   rear=qheader;
}
void enqueue()
```

```
{
   new= (struct node*) malloc(sizeof(struct node));
   if(new==NULL)
   {
      printf("\n insufficient memory, Push not possible");
   }


   else
   {
      printf("\n enter the item to insert: ");
      scanf("%d", &item);
      new->data=item;
      rear->next=new;
      new->next=NULL;
      if(front==qheader&&rear==qheader)
      {
         front=new;
         rear=new;
      }
      else
      {
         rear=new;
      }
   }
}

void dequeue()
{
 if(front==qheader&&rear==qheader)
 {
    printf("queue is empty, dequeue not possible");
 }
else
{
   ptr=front;
   if(front==rear)
   {
      front=qheader;
      rear=qheader;
      qheader->next=NULL;
   }
   else
   {
      qheader->next=ptr->next;
      front=ptr->next;
```

```
    }
    free(ptr);
}
}

void status()
{
    int count=0;
    if(front==qheader&&rear==qheader)
 {
     printf("queue is empty, dequeue not possible");
 }
    else
    {
        ptr=front;
        while(ptr->next!=NULL)
        {
            count++;
            ptr=ptr->next;
        }
     printf("Number of elements in queue=%d",count);
}
}

void display()
{
    if(front==qheader&&rear==qheader)
 {
     printf("queue is empty, dequeue not possible");
 }
 Else
{
printf("elements of the queue are: ");
    ptr=front;
    while(ptr->next!=NULL)
    {
        printf("\t%d", ptr->data);
        ptr=ptr->next;
    }
}
}
```

6. Write the applications of Queues.

Applications of Queue are

1. Breadth First Search (BFS) Algorithm

Breadth-first search (BFS) is a general technique for traversing a graph. One starts at some arbitrary node as the start node and expand shallowest unexpanded node before exploring deeper levels. The algorithm uses a queue to keep track of nodes to visit where new successors go to the end of the queue. Each entry in the queue stores all edges of the node processed and new nodes are added to the rear of the queue.

Step 1: Initially all nodes are undiscovered. Mark the start node as discovered Enqueue the start node S Front of queue which initially is the start node S.

Now enqueue all neighbours of node S into the Queue

Step2: Repeat Until queue is Empty

a. Select the node F in front of queue & mark it as discovered and process it by examining its neighbours.

b. Repeat until all neighbours of F have been processed

If a neighbour of F has not yet been discovered, add it to the rear of queue else do not add it to queue

c. Now delete the original node from the queue and go to step 2

2. Printing Job Management

Queues has the application in the management of printer jobs.

Many users send their printing jobs to a public printer. The printer will put them into a queue according to the arrival time and print the jobs one by one.

Assume that the documents are A.doc, B.doc, C.doc first arrive for printing. Therefore, the three jobs are enqueued and A.doc is sent for printing. Next A.doc finishes and is dequeued. Therefore B.doc starts printing. Meanwhile D.doc arrives and is enqueued. while B.doc is still printing. Next B.doc finishes and is dequeued. Now C.doc the document at the front of the queue is sent for printing. Next C.doc finishes and is dequeued. Now the final item in the queue, D.doc is sent for printing. This process will continue as long as documents arrive for printing.

3. **Handling Website Traffic**

Increasing website traffic is the primary goal for websites. However, extreme spikes in internet traffic frequently cause website infrastructure to fail. Particularly, the inventory systems and payment gateways are two of the most vulnerable systems needed to be preserved.

Another common concern is to ensure good user experience and fairness in terms of directing users from one webpage to another, in heavy-traffic situations.

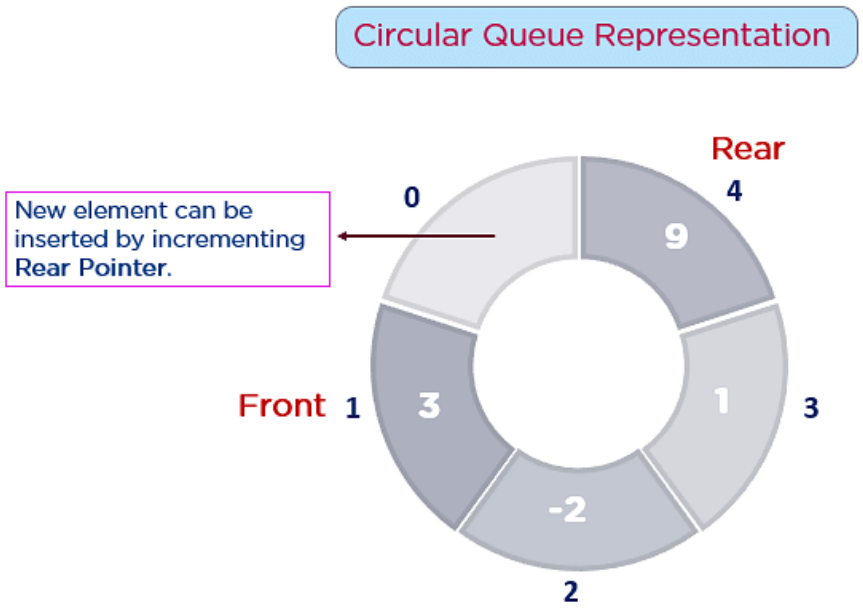
One of the best website traffic management solutions is by implementing a virtual HTTP request queue, thus making it one of the most used applications of queues.

CPU Scheduling

This is one of the most common applications of queues where a single resource is shared among multiple consumers, or asked to perform multiple tasks.
Operating system store all the processes in the memory in the form of a queue.
It makes sure that the tasks are performed based on the sequence of requests. When we need to replace a page, the oldest page is selected for removal, underlining the First-In-First-Out (FIFO) principle.

7. Explain circular queue and its implementation?

A circular queue is an extended version of a linear queue as it follows the First In First Out principle with the exception that it connects the last node of a queue to its first by forming a circular link. Hence, it is also called a Ring Buffer.



The Circular Queue is similar to a Linear Queue in the sense that it follows the FIFO (First In First Out) principle but differs in the fact that the last position is connected to the first position, replicating a circle.

Operations
Front - Used to get the starting element of the Circular Queue.
Rear - Used to get the end element of the Circular Queue.
enQueue(value) - Used to insert a new value in the Circular Queue. This operation takes place from the end of the Queue.
deQueue() - Used to delete a value from the Circular Queue. This operation takes place from the front of the Queue.

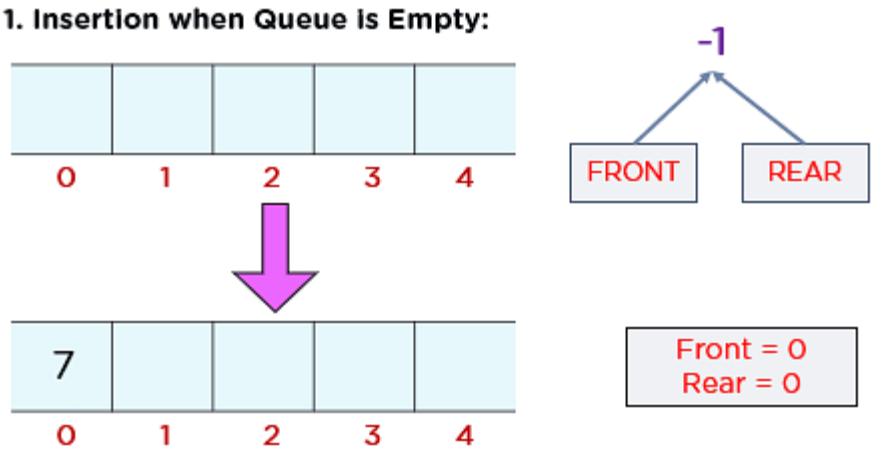
Steps for Implementing Queue Operations
Enqueue() and Dequeue() are the primary operations of the queue, which allow you to manipulate the data flow. These functions do not depend on the number of elements inside

the queue or its size; that is why these operations take constant execution time, i.e., O(1) [time-complexity].

1. Enqueue(x) Operation

Steps to insert (enqueue) a data element into a circular queue -

- Step 1: Check if the queue is full (Rear + 1 % Maxsize = Front)
- Step 2: If the queue is full, there will be an Overflow error
- Step 3: Check if the queue is empty, and set both Front and Rear to 0
- Step 4: If Rear = Maxsize - 1 & Front != 0 (rear pointer is at the end of the queue and front is not at 0th index), then set Rear = 0
- Step 5: Otherwise, set Rear = (Rear + 1) % Maxsize
- Step 6: Insert the element into the queue (Queue[Rear] = x)
- Step 7: Exit

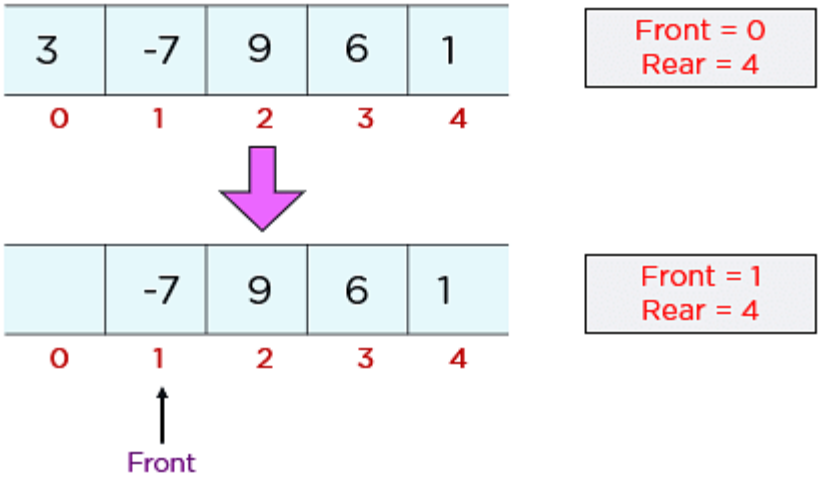


2. Dequeue() Operation

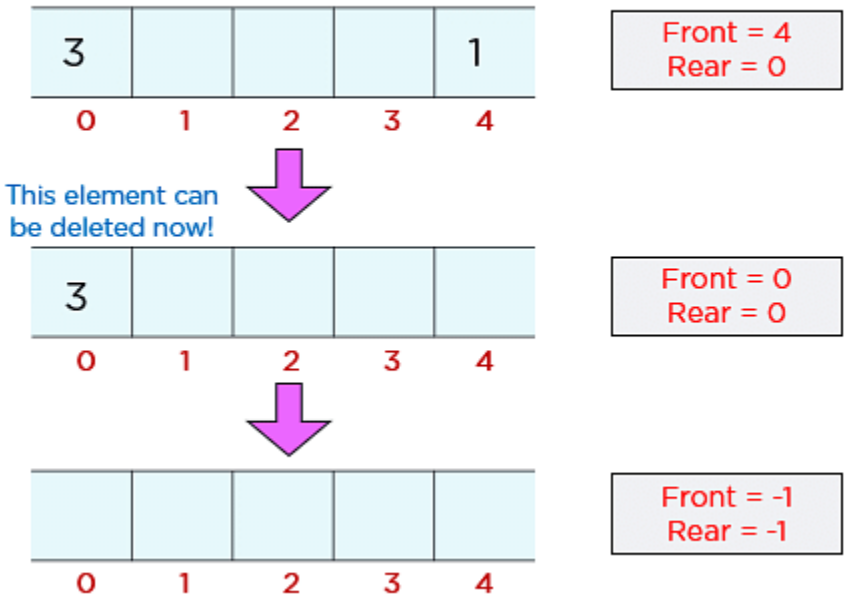
Obtaining data from the queue comprises two subtasks: access the data where the front is pointing and remove the data after access. You should take the following steps to remove data from a circular queue -

- Step 1: Check if the queue is empty (Front = -1 & Rear = -1)
- Step 2: If the queue is empty, Underflow error
- Step 3: Set Element = Queue[Front]
- Step 4: If there is only one element in a queue, set both Front and Rear to -1 (IF Front = Rear, set Front = Rear = -1)
- Step 5: And if Front = Maxsize -1 set Front = 0
- Step 6: Otherwise, set Front = Front + 1
- Step 7: Exit

1. Deletion when rear at the end of queue and front at the beginning of the queue



2. Deletion when front reached at end of queue but there is element rear is at beginning of queue



8. Explain double ended queue and its operations?

A double-ended queue, commonly referred to as a deque (pronounced "deck"), is a versatile data structure that **allows for the insertion and deletion of elements from both ends.** A double-ended queue in the data structure is an extension of the linear queue structure.

Deque provides greater flexibility in data handling. Unlike a standard queue, which follows the First-In-First-Out (FIFO) principle, a deque **can function in both FIFO and Last-In-First-Out (LIFO) modes**, making it suitable for various applications in programming and algorithms.

Double-ended queues in the data structure are especially useful in scenarios where you need to manage a collection of items with dynamic size and require efficient access to both ends. They can be implemented using arrays or linked lists, each offering different performance characteristics.

Types of Double ended queues Implementations

Deques can be implemented in several ways, each with its strengths and weaknesses. The most common implementations are:

1. Array-Based Deque

- In this implementation, a fixed-size array is used to store the elements of the deque. Two pointers (or indices) are maintained: one for the front and one for the rear.
- Advantages include constant time complexity (O(1)) for access and modification operations.
- However, a significant limitation is that resizing the array when it becomes full can be costly, requiring all elements to be copied to a new larger array.

2. Linked List-Based Deque

- This implementation utilizes nodes that contain data and pointers to the next and previous nodes.
- Elements can be added or removed from either end without needing to copy existing elements.
- The time complexity for all deque operations (insertion, deletion, and access) remains (O(1)).
- However, linked lists require extra memory for pointers and can be less cache-friendly compared to arrays.

9. Write the algorithm to implement Deque operations?

Operations of Deque

1. insertfront(): Adds an item at the front of Deque
2. insertrear(): Adds an item at the rear end of deque
3. deletefront(): Deletes from the front end of deque
4. **deleterear():** Delete an item from the rear end of Deque

Insertfront():

1. if(front==0) then
2. print insertion at front is not possible
3. else
4. if (front == -1) then
5. front = 0, rear=0
6. if(front>0) then
7. front –
8. deque[front] = item
9. end if
10. end if

11. end if
12. stop

Insert at rear:

Step:
1. if(rear==size)
2. print insert at rear end not possible
3. else
4. if(rear<size)
5. rear++
6. if(front == -1)
7. front =0, rear = 0
8. deque[rear] = item
9. end if
10. end if
11. end if
12. stop

Delete at front

Steps:
1. if(front == -1) then
2. print delete at front is not possible
3. else
4. print deleted item q[front]
5. if(front == rear) then
6. front=-1, rear=-1
7. else
8. front++
9. end if
10. end if
11. stop

Delete at rear:
Steps:
1. if(front == -1) and (rear==-1) then
2. print Delete at rear end is not possible
3. else
4. print deleted itemdeque[rear]
5. if(front==rear) then
6. front=-1, rear=-1
7. else
8. rear—
9. end if
10. end if

11. stop

Display
Algorithm for Display items
Steps:
1. if(front==-1) and (rear==-1) then
2. print Deque is empty
3. else
4. i=front
5. while(i<=rear) do
6. print deque[i]
7. i++
8. end while
9. end if
10. stop

10. Write a C program to implement Deque operations.

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


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

}*header=NULL,*ptr=NULL;


void create(int A[],int n)
{
 int i;
 struct Node *t,*last;
 header=(struct Node *)malloc(sizeof(struct Node));
 header->prev=NULL;
 header->data=A[0];
 header->next=NULL;
```

```
 last=header;
 for(i=1;i<=n;i++)
 {
 t=(struct Node*)malloc(sizeof(struct Node));
 t->data=A[i];
 t->next=last->next;
 t->prev=last;
 last->next=t;
 last=t;
 }
}

void Display( )
{
   ptr=header;
 while(ptr->next!=NULL)
 {
  printf("\n%d\n ",ptr->data);
  ptr=ptr->next;
 }
}
void insert_first()
{
   struct Node *t;
   t=(struct Node*)malloc(sizeof(struct Node));
   int x;
   printf("enter the value to insert at first position: ");
   scanf("%d", &x);
   t->data = x;
   t->prev = NULL;
   t->next= header;
   header->prev=t;
```

```
    header=t;

}

void insert_any()
{
    struct Node *t;
    t=(struct Node*)malloc(sizeof(struct Node));
    int x, pos,i;
    printf("enter the value to insert at given position: ");
    scanf("%d", &x);
    printf("enter the position to insert the given value: ");
    scanf("%d", &pos);
    ptr=header;
    t->data = x;
    for(i=0;i<pos-1;i++){
        ptr=ptr->next;}

    t->next = ptr->next;

    t->prev= ptr;
    ptr->next->prev=t;
    ptr->next=t;
}


int main()
{
 int a[5]={1,2,3,4,5};
 create(a,5);
 printf("\n After creating the Linked List: \n");
 Display();
```

```
 insert_first();
 printf("\n After inserting element at the first position in the Linked List: \n");
 Display();
 insert_any();
  printf("\n After inserting element at the given position in the Linked List: \n");
 Display();

 return 0;
}



//Double Linked List Delete Program

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

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

}*header=NULL,*ptr=NULL;

void create(int A[],int n)
{
 int i;
 struct Node *t,*last;
 header=(struct Node *)malloc(sizeof(struct Node));
 header->prev=NULL;
 header->data=A[0];
```

```
 header->next=NULL;
  last=header;
 for(i=1;i<=n;i++)
 {
 t=(struct Node*)malloc(sizeof(struct Node));
 t->data=A[i];
 t->next=last->next;
 t->prev=last;
 last->next=t;
 last=t;
 }
}

void Display( )
{
    ptr=header;
 while(ptr->next!=NULL)
 {
  printf("\n%d\n ",ptr->data);
  ptr=ptr->next;
 }
}

void delete_first()
{
    ptr=header;
    header=header->next;
    free(ptr);
    header->prev=NULL;
}

void delete_any()
```

```
{
   ptr=header;
   int i, pos;

   printf("enter the pos to delete");
   scanf("%d", &pos);
   for(i=0;i<pos-1;i++)
      ptr=ptr->next;

   ptr->prev->next=ptr->next;

   ptr->next->prev=ptr->prev;

   free(ptr);

}

int main()
{
 int a[5]={10,20,30,40,50};
 create(a,5);
 printf("\n After creating the Linked List: \n");
 Display();
 delete_first();
 printf("\n After deleting the first element from the Linked List: \n");
 Display();
 delete_any();
 printf("\n After deleting the element at given position from the Linked List: \n");
 Display();
 return 0;
}
```