Practical-5

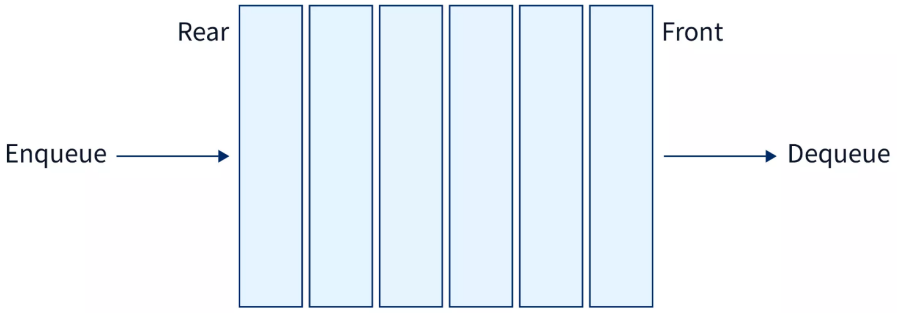
Queue Operations

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. A Queue is an abstract linear data structure serving as a collection of elements that are inserted (enqueue operation) and removed (dequeue operation) according to the First in First Out (FIFO) approach.

Insertion happens at the rear end of the queue whereas deletion happens at the front end of the queue. The front of the queue is returned using the peek operation.

A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The name "Queue" in data structure comes from the analogy to a set of physical elements lined up in a queue like people at a ticket counter. A real life example of this data structure would be a queue of cars at the toll booth: The first car (at the front of the queue) to reach the toll booth will be the first one to leave and the last car (at the back of the queue) will be the last one to leave. A queue of people waiting for their turn or a queue of airplanes waiting for landing instructions are also some real life examples of the queue data structure. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

To Summarize Queue in data structure is an ordered, linear sequence of items. It is a FIFO (First In First Out) data structure, which means that we can insert an item to the rear end of the queue and remove it from the front of the queue only. A Queue is a sequential data type, unlike an array, in an array, we can access any of its elements using indexing, but we can only access the element at the front of the queue at a time.



A Queue in data structure can be accessed from both of its sides (at the front for deletion and back for insertion). A Queue in data structure can be implemented using arrays, linked lists, or vectors.

Operations on Queue:

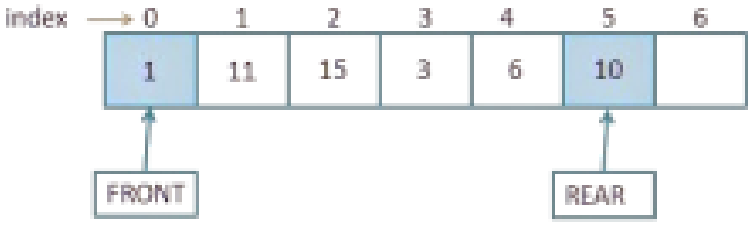
The Queue in data structure uses the FIFO (First In First Out) approach. Initially, we will set a front pointer to keep track of the front most element of the queue. Then the queue is initialized to -1 as its front, as we will add (enqueue) elements to the queue, the front gets updated to point to its front most element and if we remove (dequeue) elements from the queue, the front gets reduced. We can use queue to perform its main two operations: Enqueue and Dequeue, other operations being Peek, isEmpty and isFull.

Mainly the following four basic operations are performed on queue:

• **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

• **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition. • **Front:** Get the front item from the queue.

• **Rear:** Get the last item from the queue.



Enqueue
The Enqueue operation is used to add an element to the front of the queue.
Steps of the algorithm:

- Check if the Queue is full.
- Set the front as 0 for the first element.
- Increase rear by 1.
- Add the new element at the rear index.

Dequeue
The Dequeue operation is used to remove an element from the rear of the queue.
Steps of the algorithm:

- Check if the Queue is empty.
- Return the value at the front index.
- Increase front by 1.
- Set front and rear as -1 for the last element.

Peek
The Peek operation is used to return the front most element of the queue.

Steps of the algorithm:

- Check if the Queue is empty.
- Return the value at the front index.

isFull
The isFull operation is used to check if the queue is full or not.
Steps of the algorithm:

- Check if the number of elements in the queue (size) is equal to the capacity, if yes, return True.
- Return False.

isEmpty
The isEmpty operation is used to check if the queue is empty or not.
Steps of the algorithm:

- Check if the number of elements in the queue (size) is equal to 0, if yes, return True.
- Return False.

Implementation of Queue using Array Operations

- In queue insertion and deletion of elements takes place at two different ends.
- The addition of an element happens at an end known as REAR and deletion happens at the FRONT end.
- Implementation of a queue using an array starts with the creation of an array of size n.
- And initialize two variables FRONT and REAR with -1 which means the queue is empty.
- The REAR value represents the index up to which value is stored in the queue and the FRONT value represents the index of the first element to be dequeued

Enqueue
Insert an element from the rear end into the queue. Element is inserted into the queue after checking the overflow condition n-1==REAR to check whether the queue is full or not.

- If n-1==REAR then this means the queue is already full.
- But if REAR<n means that we can store an element in an array.
- So increment the REAR value by 1 and then insert an element at the REAR index.

Dequeue
Deleting an element from the FRONT end of the queue. Before deleting an element we need to check the underflow condition front == - 1 or front > rear to check whether there is at least one element available for the deletion or not.

- If front == - 1 or front > rear then no element is available to delete.
- Else delete FRONT index element
- If REAR==FRONT then we set -1 to both FRONT AND REAR
- Else we increment FRONT.

Front
Returns the FRONT value of the queue.

- First of all we need to check that the queue is not empty.
- If the queue is empty then we display that the queue is empty we simply return from the function and not execute further inside the function
- Otherwise, we will return the FRONT index value.

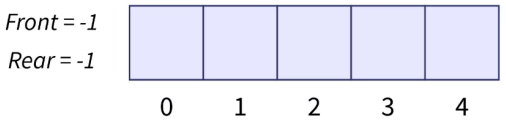
Display
It will traverse the queue and print all the elements of the queue.

- Firstly check whether the queue is not empty.
- If empty we display that the queue is empty we simply return from the function and not execute further inside the function.
- Else we will print all elements from FRONT to REAR.

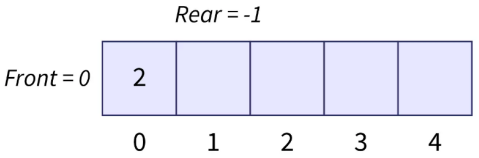
Example

Let us understand implementation of queue using array problem by an example *n=5*

Refer to the below image to show an empty array creation of size *5*

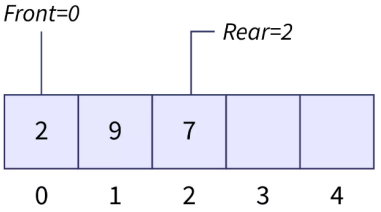


enqueue(2) Refer below image to show the enqueue operation



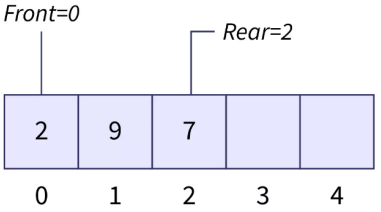
front() -> 2

enqueue(9) Refer below image to show the enqueue operation

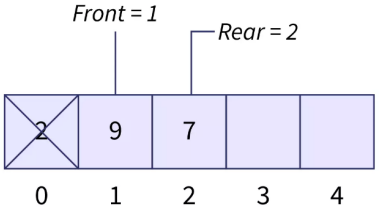


display() -> 2 9

enqueue(7) Refer below image to show the enqueue operation

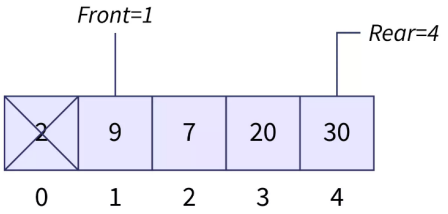


dequeue() Refer below image to show the dequeue operation



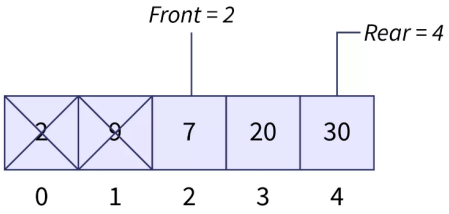
display() -> 9 7

enqueue(20) enqueue(30) Refer below image to show the enqueue operation

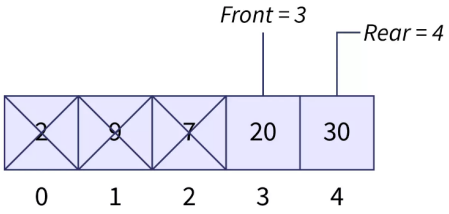


display() -> 9 7 20 30

dequeue() Refer below image to show the dequeue operation



dequeue() Refer below image to show the dequeue operation



display() -> 20 30

In the above example of implementation of the queue using array problem, first of all we are creating an array named arr[] of size n which means the capacity of the queue is n and initializing FRONT and REAR as -1. Now for enqueue operation we increment REAR and insert an element at REAR arr[REAR]=item. For the dequeue operation, we delete an element at the FRONT index and increment FRONT by 1.

enqueue(item)

```
Step 1:
      IF REAR = N - 1
          Print “OVERFLOW! QUEUE IS ALREADY FULL”
          TERMINATE
Step 2:
      IF FRONT = -1 and REAR = -1
          SET FRONT AND REAR AT 0 FRONT =  REAR=0
      ELSE
          INCREMENT REAR BY 1 SET REAR = REAR + 1
      [END OF IF]
Step 3:
      INSERT ELEMENT AT REAR Set QUEUE[REAR] = ITEM
Step 4:
      EXIT
```

dequeue()

```
Step 1:
      IF FRONT = -1 or FRONT > REAR
          Print “UNDERFLOW! QUEUE IS EMPTY”
          TERMINATE
      ELSE
          SET DELETE FRONT VALUE VAL =QUEUE[FRONT]
Step 2:
      IF FRONT==REAR
          SET BOTH FRONT AND REAR AT -1 SET FRONT=REAR=-1
      ELSE
          INCREMENT FRONT BY 1 SET FRONT = FRONT + 1
      [END OF IF]
Step 3:
      EXIT
```

front()

```
Step 1:
      IF FRONT = -1
          Print “QUEUE IS EMPTY!”
      ELSE
          SET VAL = QUEUE[FRONT]
          PRINT VAL
      [END OF IF]
Step 2:
      EXIT
```

display()

```
Step 1:
      IF FRONT = -1
          Print “QUEUE IS EMPTY!”
          TERMINATE
      ELSE
          FOR ALL ITEM FROM FRONT TO REAR
              PRINT ITEM
          [END OF FOR]
      [END OF IF]
Step 2:
      EXIT
```

Time Complexity

- enqueue(),dequeue(),front() can be performed in constant time. So time Complexity of enqueue, dequeue() , front() are O(1)
- display will print all the elements of the queue. So its time complexity is O(N).
- Overall worst case time complexity of implementation of queue using array is O(N).

Space Complexity

- As we are using an array of size n to store all the elements of the queue so space complexity using array implementation is O(N).

Stack

1. It follows the LIFO (Last In First Out) order to store the elements, which means the element that is inserted last will come out first.
2. It has only one end, known as the top, at which both insertion and deletion take place.
3. The insertion operation is known as push and the deletion operation is known as pop.
4. The condition for checking whether the stack is empty is top ==-1 as -1 refers to no element in the stack.
5. The condition for checking if the stack is full is top==max-1 as max refers to the maximum number of elements that can be in the stack.
6. There are no other variants or types of the stack.
7. It has a simple implementation compared to queues as no two pointers are involved.
8. It is used to solve recursion-based problems.
9. A real-life example of a stack can be the Undo/Redo operation in Word or Excel.

Queue

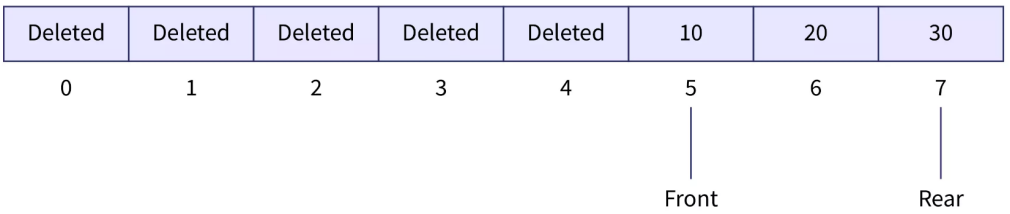
1. It follows the FIFO (First In First Out) order to store the elements, which means the element that is inserted first will come out first.
2. It has two ends, known as the rear and front, which are used for insertion and deletion. The rear end is used to insert the elements, whereas the front end is used to delete the elements from the queue.
3. The insertion operation is known as enqueue and the deletion operation is known as dequeue.
4. The condition for checking whether the queue is empty is front == -1
5. The condition for checking if the queue is full is rear==max-1 as max refers to the maximum number of elements that can be in the queue.
6. There are three types of queues known as circular, double-ended, and priority.
7. It has a complex implementation compared to stacks as two pointers front and rear are involved.
8. It is used to solve sequential processing-based problems.
9. A real-life example of a queue can be an operating system process scheduling queues.

Queue using Array

```
#define size 3
int que[size],front = -1, rear= -1;
void enqueue(int Q[],int y) {
      if(rear==size-1){
          printf("\nQueue overflow.");
       }
      else{
          rear=rear+1;
          Q[rear] = y;
      }
      if(front == -1)
      front=0;
 }
int dequeue(int Q[]) {
      int val;
      if(front == -1) {
          printf("Queue is underflow");
          return -1;
      }
      else {
          val = Q[front];
          Q[front] = 0;

           if (front == rear) //reset the queue
           {
                front= -1;
                rear = -1;
           }
           else
               front=front+1;
      }
      return val;
 }
```

Limitations of Queue

- A queue is not readily searchable: You might have to maintain another queue to store the dequeued elements in search of the wanted element.
- Traversal possible only once: The front most element needs to be dequeued to access the element behind it, this happens throughout the queue while traversing through it. In this process, the queue becomes empty.
- Memory Wastage: In a Static Queue, the array's size determines the queue capacity, the space occupied by the array remains the same, no matter how many elements are in the queue.



Applications of Queue

- A Queue data structure is used when things don't have to be accessed immediately but in FIFO (First In First Out) order. This property of the queue data structure makes queue applicable for the following situations:
- Job Scheduling The computer schedules the job execution one by one. There are many jobs like a key press, a mouse click, etc. in the system. The jobs are brought in the main memory and are assigned to the processor one by one which is organized using a queue.

  For eg, Round Robin processor scheduling in queues.

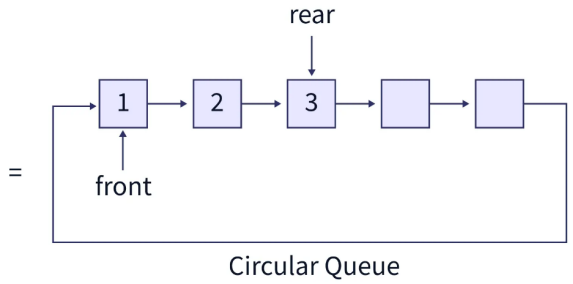
- Multiprogramming If there are multiple programs in the main memory, then that state is called multiprogramming. The programs in the main memory are organized in the form of queues, which are then called "Ready Queues". The processors will execute the programs by accessing them from the "Cache Memory" for simultaneous execution.
- Operation on data structures Certain operations like BFS (Breadth First Search), and tree traversal uses Queue. The sequence of traversal of inputs is set using queues.
- Buffer Space Queues are used in networking, during the transmission of data from the source machine to the destination.

There are four types of queues:

- Simple Queue
    - In a simple queue, we follow the First In, First Out (FIFO) rule. Insertion takes place at one end, i.e., the rear end or the tail of the queue, and deletion takes place at the other end, i.e., the front end or the head of the queue.
    - The time complexity of a simple queue is O(1) for insertion and deletion operations.
- Circular Queue
    - A circular queue exhibits similar characteristics to that of a simple queue, with the additional property of joining the front end to the rear end.
    - It follows the FIFO rule where the rear is connected to the end.
    - It is also known as the ring buffer.
    - The time complexity of a circular queue is O(1) for insertion and deletion.
- Priority Queue
    - A priority queue also exhibits similar characteristics to that of a simple queue where each element is assigned a particular priority value, where elements in the queue are assigned based on priority.
    - There are 2 types of priority queues:
        - Ascending Order: In this priority queue, the elements are arranged in ascending order of their priority, i.e., the element with the smallest priority comes at the start, and the element with the greatest priority comes at the end.
        - Descending Order: In this priority queue, the elements are arranged in descending order of their priority, i.e., the element with the greatest priority is at the start and the element with the smallest priority is present at the end of the queue.
    - For insertion and deletion, the priority queue has a time complexity of O(logn).
- Double-ended Queue
    - A Double-ended Queue, or Deque, is a different type of queue where enqueue (insertion) and dequeue (deletion) operations are performed at both the ends, i.e., the rear-end (tail) and the front-end (head). The Deque
    - can further be divided into 2 special queues, i.e., Input-restricted Deque and Output-Restricted Deque.
    - The time complexity of a double-ended queue is O(1) for insertion and deletion.

Circular Queue

A circular queue is a linear data structure used in programming to store data. It continues to adhere to the First In First Out insertion and deletion protocol. You may be familiar with the queue data structure, which similarly follows the FIFO principle and is a linear data structure; however, the difference between a circular queue and a normal queue is that the last element of the queue logically or physically remains linked to the first element, resulting in a circular connection. That's why it is called Circular + Queue. We can create the circular queue in C using either arrays or linked lists.

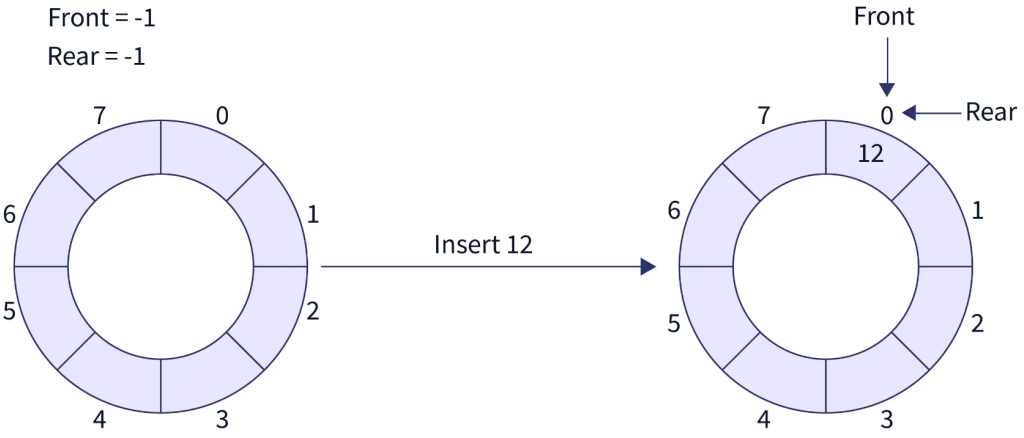


Operations on a Circular Queue

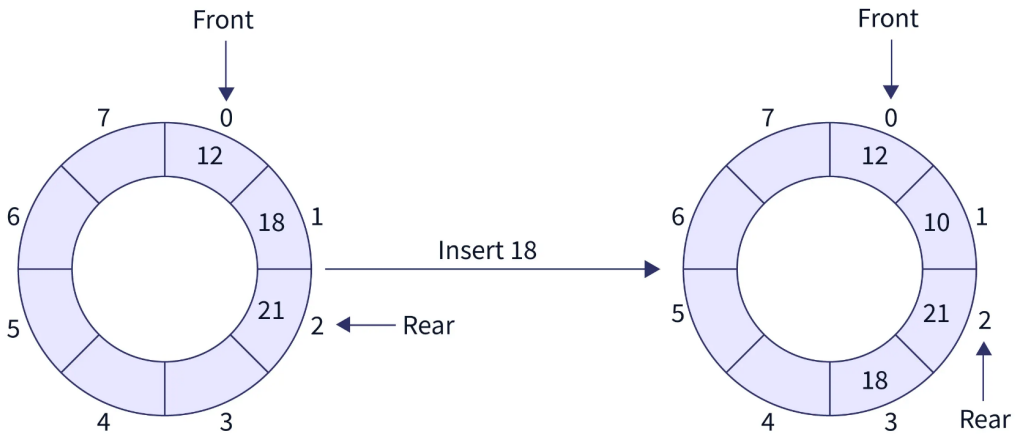
The Circular Queue supports two basic operations: inserting an element (called enqueue) and removing an element from the queue (called dequeue). Two pointers, front, and rear (sometimes called back), are needed to maintain details of where insertion and deletion will actually occur. When the queue is empty, both pointers either point to NULL or we assign -1 to indicate that the queue is empty.

enQueue Operation

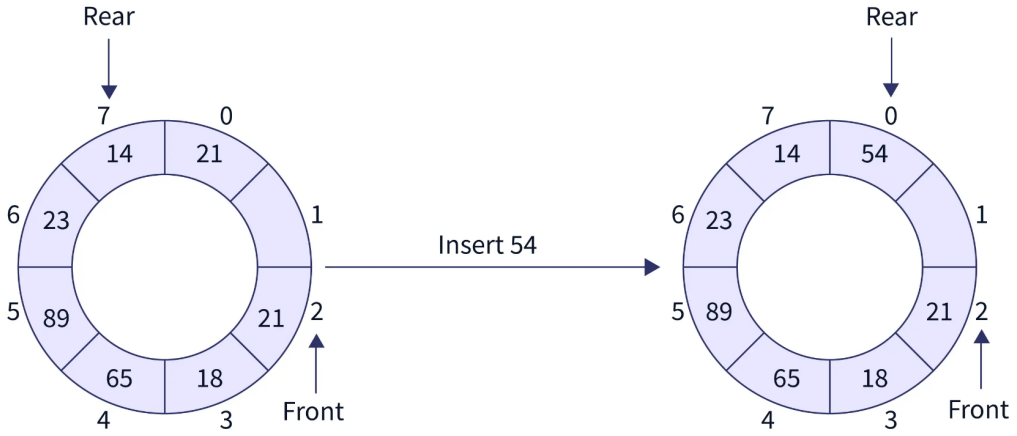
The enqueue operation means inserting some element into the queue, with every valid enqueue operation the rear pointer keeps incrementing.



If initially, the queue is empty then on the first insertion the front and rear will be set to the initial position and both will point to the same element. Else the rear pointer will keep incrementing for each insertion.



If increment of the rear pointer is not possible but there is enough space for insertion, the rear pointer shifts towards 0th index or first position.




```
//The parameter "element" is the element that
// will be inserted in the circular queue.
enQueue(element)
{
    // If the front is at the initial position
    // and the rear is at the last position of the queue
    // means the queue is full.
    if front == 0 AND rear == size - 1:
        printf("Queue is FULL")
        return

    // If there is only difference of 1 position
     // in between the front and rear pointers also then the queue is
full.
    if front == rear + 1:
        printf("Queue is FULL")
        return

    // If there was no element initially then initialize the front
    // as well as rear to point to the first position
    // and then add the element at the rear position.
    if front == -1 AND rear == -1:
```

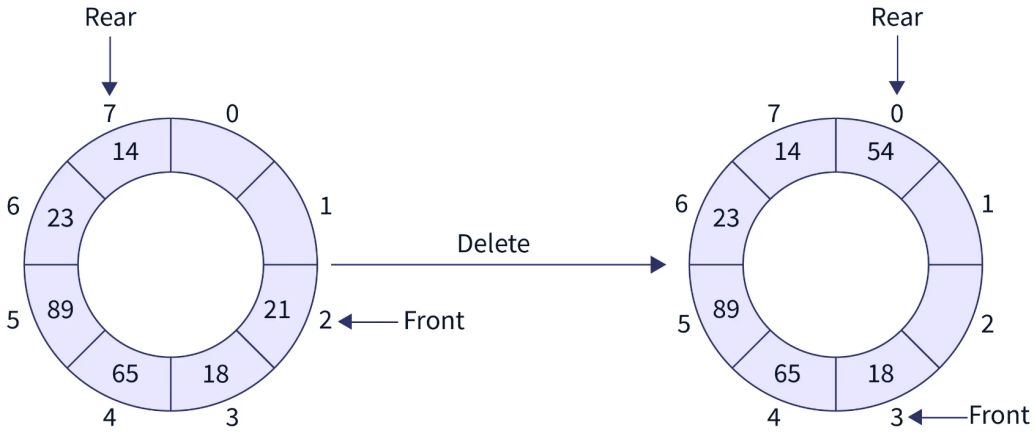
```
        front = 0
        rear = 0
        circularQueue -> rear = element
        return

    // If there were already one or more elements initially,
    // then just change the rear position.

    // If the rear was at the last position of the queue
    // then make it point to the first position
    // because the rear can take the position in a circular manner
    if rear == size - 1:
        rear = 0

    // Else, the rear is not at the last position of the queue,
    // so increment the rear pointer to store new element
    else
        rear = rear + 1

    // In the end insert the element at the rear position
    circularQueue -> rear = element
    return
}
```

Time & Space Complexity

- In the insertion, we just increment the rear pointer and assign a value to already allocated memory, these both sub-operations are constant hence, the time complexity of the insertion operation is constant O(1).
- Also, there is no need for extra space while insertion, so space complexity is also constant O(1).

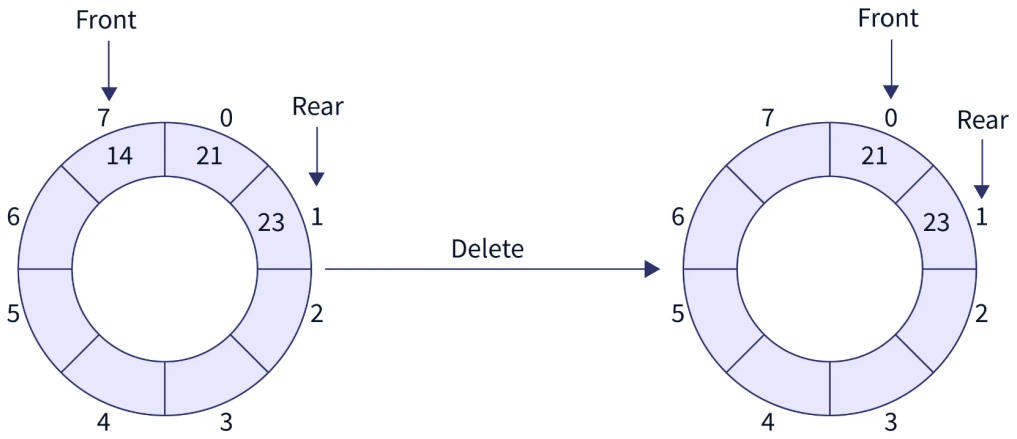
deQueue Operation

The dequeue operation means deleting some element from the queue, With every valid dequeue operation, the front pointer keeps incrementing. During deletion, the value corresponding to the front pointer is removed and the front pointer keeps incrementing.



Deletion of the element, when the front pointer is already at the last position of the non-empty Circular Queue, and further increment of position is not possible. In the circular queue, both pointers can

circularly take the positions so, the front pointer will shift to the first position of the circular queue.




```
deQueue()
{
    // If the front pointer is not initialized yet,
    // means the circular queue hasn't
    // even a single element till now
    if front == -1:
        printf("Queue is Empty")
        return

    // Store the deleted element locally
    currentElement = circularQueue -> front

    //Reset the value at that location, let's say by assigning 0.
    circularQueue -> front = 0

    //Now just change the position of the front pointer

    // If the front and rear both were pointing to the same index,
    // means there was only one element in the queue
    // Reset both pointers by assigning -1.
    if front == rear:
        front = -1
        rear = -1
        return currentElement

    // If the front is already at the last position then
    // increment will take it to the first position of the queue
    // because pointers can take any position circularly.
    if front == size - 1:
        front = 0

    // Else, just increment the front pointer
    else
        front = front + 1

    // Return the deleted element
    return currentElement
}
```

Time & Space Complexity

- In the algorithm, only the value at the memory space located by the front pointer is removed and the front pointer is incremented. The time complexity is constant O(1) because we are not performing any intensive steps, memory release and increment both are constant sub-operations.
- Also, there is no need to store or allocate some extra memory for deletion operation hence space complexity is also constant O(1).

Circular Queue using Array

```
#include<stdio.h>
# define MAX 5
int cqueue_arr[MAX];
int front = -1;
int rear = -1;
void insert(int item) {
      if((front == 0 && rear == MAX-1) || (front == rear+1))
     //check overflow  {
            printf("Queue Overflow \n");
            return;
      }
      if(front == -1)
      {
            front = 0;
            rear = 0;
      }
      else
      {
            if(rear == MAX-1) // increment rear pointer
                rear = 0;
            else
                 rear = rear+1;
      }
      cqueue_arr[rear] = item ;
}
void deletion()
{
      if(front == -1) //check underflow
      {
            printf("Queue Underflow\n");
            return ;
      }
      printf("Element deleted from queue is : %d
     \n",cqueue_arr[front]);
     if(front == rear) //reset pointers if last element
      {
            front = -1;
            rear=-1;
      }
      else //increment front pointer
      {
            if(front == MAX-1) //if front is at last element
                front = 0;
            else
                front = front+1;
      }
}
int main()
```

```
{
      int choice,item;
      do
      {
          printf("1.Insert\n");
           printf("2.Delete\n");
           printf("3.Quit\n");
           printf("Enter your choice : ");
           scanf("%d",&choice);
           switch(choice) {
               case 1 :
                    printf("Input the element for
               insertion in queue : ");  scanf("%d",
               &item);
                    insert(item);
                break;
                case 2 :
                    deletion();
                break;
                case 3:
                    break;
                default:
                    printf("Wrong choice\n");
           }
      }while(choice!=3);
      return 0;
}
```

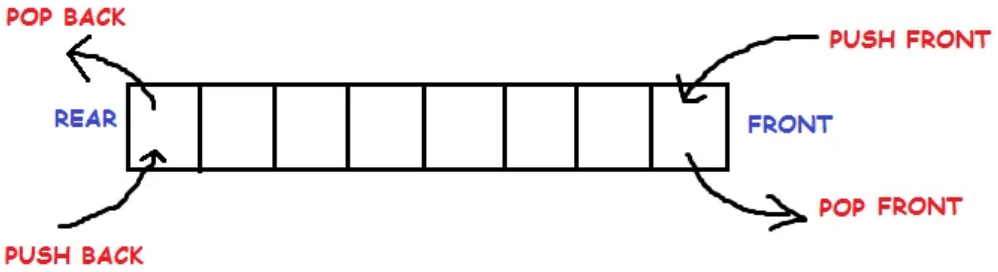
Sample - 2:

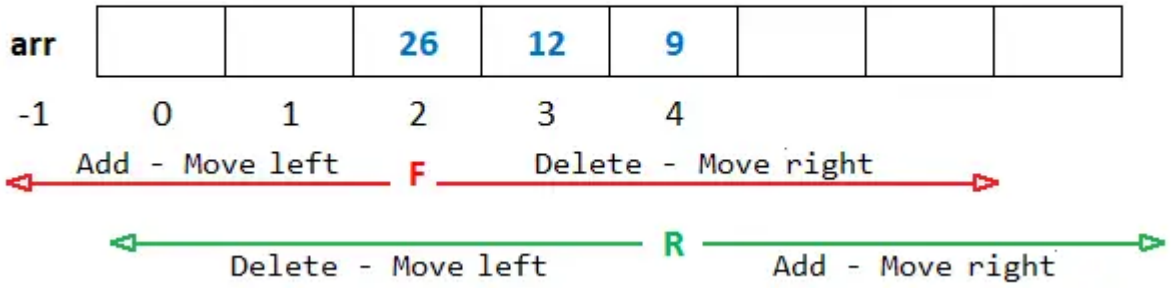
```
#include <stdio.h>
#include <stdlib.h>

#define info(x) printf("%s: %d\n", #x, x);

// We are declaring both pointer as global so that each function
// can access without passing as the parameter.
int front = -1;
int rear = -1;
int size = -1;

// Declaring a pointer that will store the base address of the
dynamically allocated array
int *circularQueue;

// Function to insert the element in the circular queue
void enQueue(int element)
{

}

// Function to delete element from the circular queue
int deQueue()
{
```

```
}

// Function to display the entire circular queue
void display(int front, int rear)
{

}
int main()
{
    size = 6;
    // Let's create a queue of size 6 and store the base address of the
array in the
    // variable named circularQueue
    circularQueue = (int *)malloc(sizeof(int) * size);

    info(front);
    info(rear);

    enQueue(8);
    info(front);
    info(rear);

    enQueue(5);
    info(front);
    info(rear);

    display(front, rear);

    enQueue(1);
    info(front);
    info(rear);

    enQueue(7);
    info(front);
    info(rear);

    display(front, rear);

    printf("Deleted Element: %d\n", deQueue());
    info(front);
    info(rear);

    printf("Deleted Element: %d\n", deQueue());
    info(front);
    info(rear);

    display(front, rear);
    return 0;
}
```

Double ended queue

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e , front and back. A Double Ended Queue is also known as Deque.





From the above image of the deque, we can see that when we add an element from the rear end, the R moves towards the right and, when we delete an element from the rear end, the R moves towards the left.

Similarly, when we delete an element from the front end, the F moves towards the right and when we add an element from the front end, the F moves towards the left.

Operation on Circular Queue

There are four operations possible on the double ended queue.

- Add Rear When we add an element from the rear end.
- Delete Rear When we delete an element from the rear end.
- Add Front When we add an element from the front end.
- Delete Front When we delete an element from the front end.

Add Rear Operation in Deque

For adding an element from the rear end first, we check if the value of te is equal to the size of the array then, we will display a message Queue is full, else we will increase the value of R by R=(R+1)%Size and add the element in the array at the new location of R and then increased the value of te by 1.

```
if(te==size){
     printf("Queue is full\n");
}
else{
```

```
        R=(R+1)%size;
        arr[R]=new_item;
        te=te+1;
    }
```

Here, te is a variable that keeps the total number of elements present in the array. When an element is added, the value of te is increased by 1. When an element has been deleted, the value of te is decreased by 1.

Delete Rear Operation in Deque

For deleting an element from the rear end first, we check if the value of te is equal to 0 then, we will display a message Queue is empty, else we will check if the value of R is -1, then we will initialize the value of R=size-1. After that, we will display the deleted element on the screen and decrease the value of R and te by 1.

```
    if(te==0){
        printf("Queue is empty\n");
    }
    else
    {
        if(R==-1){
            R=size-1;
        }
        printf("Number Deleted From Rear End = %d",arr[R]);
        R=R-1;
        te=te-1;
    }
```

Add Front Operation in Deque

For adding an element from the front end first, we check if the value of te is equal to the size of the array then, we will display a message Queue is full, else we will check if the value of F is 0 then we will initialize the value of F=size-1, else decrease the value of F by 1. After that, add the element in the array at the new location of F and then increase the value of te by 1.

```
    if(te==size){
        printf("Queue is full");
    }
    else
    {
        if(F==0){
            F=size-1;
        }
        else{
            F=F-1;
        }
        arr[F]=new_item;
        te=te+1;
    }
```

Delete Front Operation in Deque

For deleting an element from the front end first, we check if the value of te is equal to 0 then, we will display a message Queue is empty, else we will display the deleted element on the screen and then we will increase the value of F by F=(F+1)%Size and then decrease the value of te by 1.

```
if(te==0){
    printf("Queue is empty");
}
else{
    printf("Number Deleted From Front End = %d",arr[F]);
    F=(F+1)%size;
    te=te-1;
}
```

Double ended Queue using Array

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 7
int deque_arr[MAX];
int front=-1;
int rear=-1;

void insert_frontEnd(int item);
void insert_rearEnd(int item);
int delete_frontEnd();
int delete_rearEnd();
void display();
int isEmpty();
int isFull();

int main()
{ int choice,item;
     do
     { printf("\n\n1.Insert at the front end\n");
         printf("2.Insert at the rear end\n");
         printf("3.Delete from front end\n");
         printf("4.Delete from rear end\n");
         printf("5.Display\n");
         printf("6.Quit\n");
         printf("\nEnter your choice : ");
         scanf("%d",&choice);

         switch(choice)
         {
         case 1:
       printf("\nInput the element for adding in queue : ");
                scanf("%d",&item);
                insert_frontEnd(item);
                break;
         case 2:
       printf("\nInput the element for adding in queue : ");
                scanf("%d",&item);
                insert_rearEnd(item);
```

```
                break;
            case 3:
       printf("\nElement deleted from front end is : %d\n",
delete_frontEnd());
                break;
            case 4:
       printf("\nElement deleted from rear end is : %d\n",
delete_rearEnd());
                break;
            case 5:
                display();
                break;
            case 6:
                exit(1);
            default:
                printf("\nWrong choice\n");
           }/*End of switch*/
           printf("\nfront = %d, rear =%d\n", front , rear);
           display();
     }while(choice!=6);

     /*End of while*/
}/*End of main()*/

void insert_frontEnd(int item)
{
     if( isFull() )
     {
          printf("\nQueue Overflow\n");
          return;
     }
     if( front==-1 )/*If queue is initially empty*/
     {
          front=0;
          rear=0;
     }
     else if(front==0)
          front=MAX-1;
     else
          front=front-1;
     deque_arr[front]=item ;
}/*End of insert_frontEnd()*/

void insert_rearEnd(int item)
{
     if( isFull() )
     {
          printf("\nQueue Overflow\n");
          return;
     }
     if(front==-1) /*if queue is initially empty*/
     {
          front=0;
```

```
          rear=0;
     }
     else if(rear==MAX-1) /*rear is at last position
          of queue */ rear=0;
     else
          rear=rear+1;
     deque_arr[rear]=item ;
}/*End of insert_rearEnd()*/

int delete_frontEnd()
{
     int item;
     if( isEmpty() )
     {
          printf("\nQueue Underflow\n");
          exit(1);
     }
     item=deque_arr[front];
     if(front==rear) /*Queue has only one
     element */ {
          front=-1;
          rear=-1;
     }
     else
          if(front==MAX-1)
               front=0;
          else
               front=front+1;
     return item;
}/*End of delete_frontEnd()*/

int delete_rearEnd()
{
     int item;
     if( isEmpty() )
     {
          printf("\nQueue Underflow\n");
          exit(1);
     }
     item=deque_arr[rear];

     if(front==rear) /*queue has only one
     element*/ {
          front=-1;
          rear=-1;
     }
     else if(rear==0)
          rear=MAX-1;
     else
          rear=rear-1;
     return item;
}/*End of delete_rearEnd() */
```

```
int isFull()
{
     if ( (front==0 && rear==MAX-1) ||
          (front==rear+1) ) return 1;
     else
          return 0;
}/*End of isFull()*/

int isEmpty()
{
     if( front == -1)
          return 1;
     else
          return 0;
}/*End of isEmpty()*/

void display()
{ int i;
     if( isEmpty() )
     {
          printf("\nQueue is empty\n");
          return;
     }
     printf("\nQueue elements :\n");
     i=front;
     if( front<=rear )
     {
      for(i=front;i<=rear;i++)
      { if(i==front)
          printf("\n%d -->front",deque_arr[i]);
      else if(i==rear)
          printf("\n%d --->rear",deque_arr[i]);
          else
           printf("\n%d ",deque_arr[i]);
      }
     }
     else
     { while(i<=MAX-1)
               printf("%d ",deque_arr[i++]);
          i=0;
          while(i<=rear)
               printf("%d ",deque_arr[i++]);
     }
     printf("\n");

}/*End of display() */
```

Exercises

1. Write a C program to implement all operations of a normal queue using arrays.
    a. Demonstrate the operation for following elements.
        - ENQUEUE(4)
        - ENQUEUE(3)
        - DISPLAY()
        - DEQUEUE() //Show Element
        - ENQUEUE(7)
        - ENQUEUE(2)
        - ENQUEUE(5)
        - ENQUEUE(9)
        - DISPLAY()
        - DEQUEUE() //Show Element
        - ENQUEUE(3)
        - ENQUEUE(9)
        - DISPLAY()
2. Write a C program to implement all operations of a circular queue using arrays.
3. Write a C program to reverse a normal queue.

--

4. Write a C program to implement all operations of a normal queue using arrays.

```
Struct Person{
        char name;
        int amt;
};
```

    a. Demonstrate the operation for following elements of type structure.
        - ENQUEUE({A,500})
        - ENQUEUE({B,1000})
        - DISPLAY()
        - DEQUEUE() //Show Element
        - ENQUEUE({C,1500})
        - ENQUEUE({D,2000})
        - ENQUEUE({E,2200})
        - ENQUEUE({F,2500})
        - DISPLAY()
        - DEQUEUE() //Show Element
        - ENQUEUE({C,5000})
        - ENQUEUE({D,5500})
        - DISPLAY()