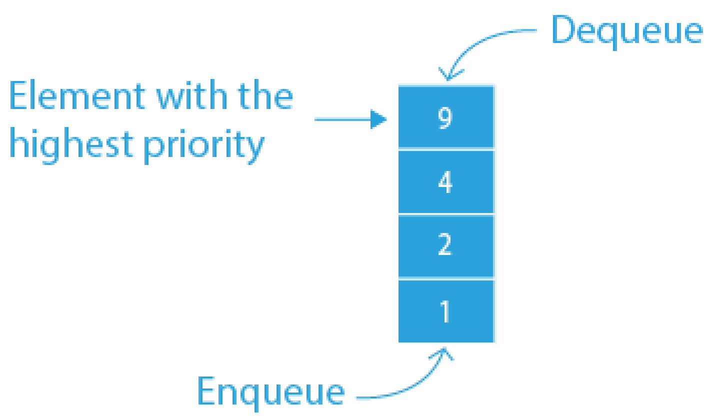
Priority Queue

Priority queues are abstract data structures where each element in the queue has a priority value. For example, in any airline, baggage under the “First-Class” or “Business” arrives before other baggage.

A priority Queue is a type of queue that follows the given below properties:

- An item with higher priority will be dequeued before the item with lower priority.
- If two elements present in the priority queue are having the same priority, then they will be served according to the order in which they are present in the queue.

The priority queue is widely used in many applications like job scheduling algorithms, CPU and Disk scheduling, and managing various resources shared between different processes, etc.



Removing Highest Priority Element

How is the Priority Value assigned in the Priority Queue?

In c, priority value will be given at the time of insertion of element. And on the basis of that priority value elements will be chosen for performing the operations. Elements with higher priority value will be served first for the operations as compared to elements with lower priority value.

The main difference between a queue and a priority queue:

In the queue, the element inserted first will be dequeued first. But in the case of a priority queue, the element which is having highest priority will be dequeued first.
When an element is popped out of the priority queue, the result will be in the sorted order, it can be either increasing or decreasing. While in queue elements are popped out in the order of FIFO (First in First out).

Priority queue can be implemented using the following data structures:

Arrays

Linked list

Heap data structure

Binary search tree

1) Implement Priority Queue Using Array:

A simple implementation is to use an array of the following structure.

```
struct item {
        int item;
        int priority;
}
```

enqueue(): This function is used to insert new data into the queue.

dequeue(): This function removes the element with the highest priority from the queue.

peek()/top(): This function is used to get the highest priority element in the queue without removing it from the queue.

Code Implementation

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

```
#include<limits.h>
#define MAX 100

int idx = -1;
int pqVal[MAX];
int pqPriority[MAX];

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

int isFull(){
return idx == MAX - 1;
}

void enqueue(int data, int priority)
{
if(!isFull()){
idx++;
pqVal[idx] = data;
pqPriority[idx] = priority;
}
}
int peek()
{
int maxPriority = INT_MIN;
int indexPos = -1;
for (int i = 0; i <= idx; i++) {
```

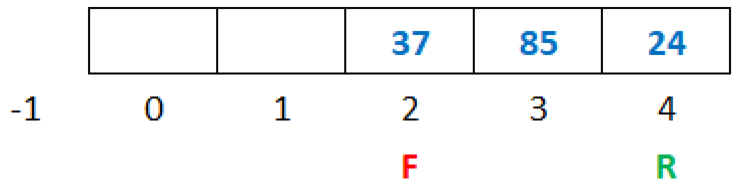
```
if (maxPriority == pqPriority[i] && indexPos > -1 && pqVal[indexPos] <
pqVal[i])
{
maxPriority = pqPriority[i];
indexPos = i;
}
else if (maxPriority < pqPriority[i]) {
maxPriority = pqPriority[i];
indexPos = i;
}
}
return indexPos;
}
void dequeue()
{
if(!isEmpty())
{
int indexPos = peek();
for (int i = indexPos; i < idx; i++) {
pqVal[i] = pqVal[i + 1];
pqPriority[i] = pqPriority[i + 1];
}
idx--;
}
}
void display(){
for (int i = 0; i <= idx; i++) {
printf("(%d, %d)\n",pqVal[i], pqPriority[i]);
}
```

```
}
int main()
{
enqueue(5, 1);
enqueue(10, 3);
enqueue(15, 4);
enqueue(20, 5);
enqueue(25, 2);
printf("Priority Queue Before Dequeue : \n");
display();
dequeue();
dequeue();
printf("\nPriority Queue After Dequeue : \n");
display();
return 0;
}
```

What is Circular Queue in C

A Circular Queue in C is a data structure in which elements are stored in a circular manner. In Circular Queue, after the last element, the first element occurs.

A Circular Queue is used to overcome the limitation we face in the array implementation of a Queue. The problem is that when the rear reaches the end and if we delete some elements from the front and then try to add a new element in the queue, it says "Queue is full", but still there are spaces available in the queue. See the example given below.



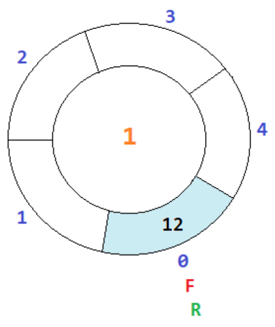
In the above image, the queue is full because the rear R reached the end of the queue. We have deleted two elements from the queue, so the front F is at index 2. We can see that there are spaces available in the queue, but we can't add a new element because the rear can't go back to index 0.

Operation on Circular Queue

There are two operations possible on the circular queue.

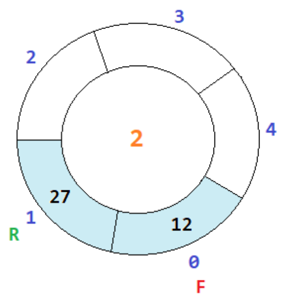
- Add - When we add an element in the circular queue.
- Delete - When we delete an element from the circular queue.

To understand how the above operations work on a circular queue. See the example given below.



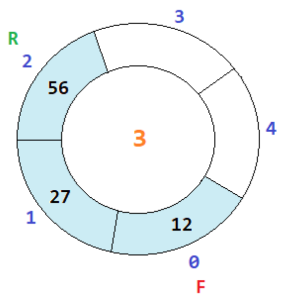
Size = 5
F = 0
R = -1
te = 0 (Total Elements)

Add 12
R = (R+1) % Size = 0
te = 1



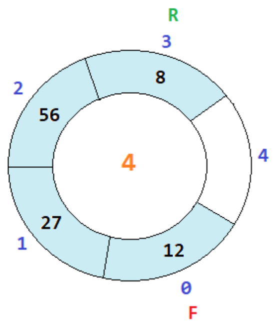
Size = 5
F = 0
R = 0
te = 1 (Total Elements)

Add 27
R = (R+1) % Size = 1
te = 2



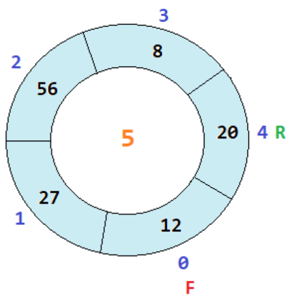
Size = 5
F = 0
R = 1
te = 2 (Total Elements)

Add 56
R = (R+1) % Size = 2
te = 3



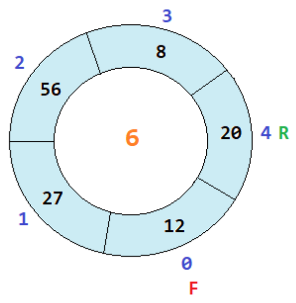
Size = 5
F = 0
R = 2
te = 3 (Total Elements)

Add 8
R = (R+1) % Size = 3
te = 4



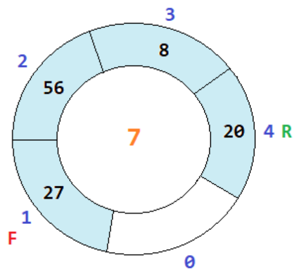
Size = 5
F = 0
R = 3
te = 4 (Total Elements)

Add 20
R = (R+1) % Size = 4
te = 5



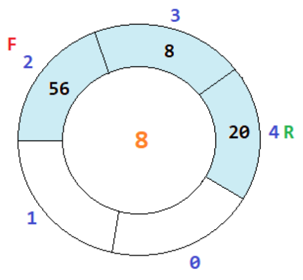
Size = 5
F = 0
R = 4
te = 5 (Total Elements)

Add 15
te == Size
Queue is full



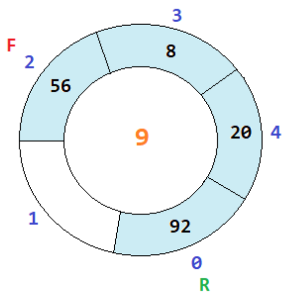
Size = 5
F = 0
R = 4
te = 5 (Total Elements)

Delete 12
F = (F+1) % Size = 1
te = 4



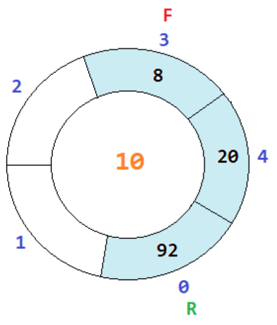
Size = 5
F = 1
R = 4
te = 4 (Total Elements)

Delete 27
F = (F+1) % Size = 2
te = 3



Size = 5
F = 2
R = 4
te = 3 (Total Elements)

Add 92
R = (R+1) % Size = 0
te = 4



Size = 5
F = 2
R = 0
te = 4 (Total Elements)

Delete 56
F = (F+1) % Size = 3
te = 3

From the above image, we can see that when we add a new element in the circular queue, the variable R is increased by R=(R+1)%Size, and the new element is added at the new position of R and te is increased by 1. Similarly, when we delete an element from the circular queue, the variable F is increased by F=(F+1)%Size and te is decreased by 1.

Add Operation in Circular Queue

For add operation in the circular queue first, we check if the value of te is equal to the value of size 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.

Example

```
if(te==size)
{
        printf("Queue is full\n");
}
else
{
        R=(R+1)%size;
        arr[R]=new_item;
   te=te+1;
}
```

Copy

Delete Operation in Circular Queue

For delete operation in the circular queue first, we check if the value of te is 0 then, we will display a message Queue is empty, else we will display the deleted element on the screen and then increase the value of F by F=(F+1)%Size and then decrease the value of te by 1.

Example

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

Copy

Program of Circular Queue using Array

Below is the complete program of circular queue in C using an array having size 5.

Circular Queue Program in C using Array

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define size 5

int main()
{
    int arr[size],R=-1,F=0,te=0,ch,n,i,x;

    for(;;)                        // An infinite loop
```

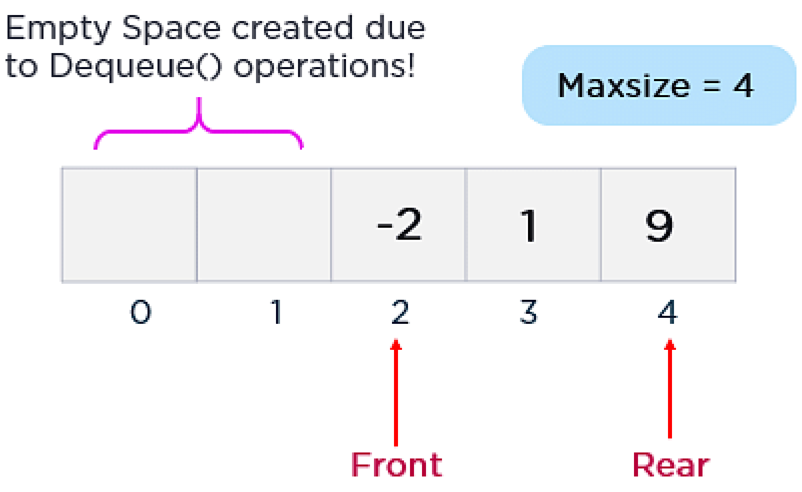
```
{
    system("cls");                  // for clearing the screen
    printf("1. Add\n");
    printf("2. Delete\n");
    printf("3. Display\n");
    printf("4. Exit\n");
    printf("Enter Choice: ");
    scanf("%d",&ch);

    switch(ch)
    {
        case 1:
            if(te==size)
            {
                printf("Queue is full");
                getch();     // pause the loop to see the message
            }
            else
            {
                printf("Enter a number ");
                scanf("%d",&n);
                R=(R+1)%size;
                arr[R]=n;
                te=te+1;
            }
            break;

        case 2:
            if(te==0)
            {
                printf("Queue is empty");
                getch();      // pause the loop to see the message
            }
            else
            {
                printf("Number Deleted = %d",arr[F]);
                F=(F+1)%size;
                te=te-1;
                getch();      // pause the loop to see the number
            }
            break;

        case 3:
            if(te==0)
            {
                printf("Queue is empty");
                getch();      // pause the loop to see the message
            }
            else
            {
                x=F;
                for(i=1; i<=te; i++)
                {
                    printf("%d ",arr[x]);
                    x=(x+1)%size;
                }
                getch();      // pause the loop to see the numbers
            }
            break;

        case 4:
            exit(0);
```

```
                break;

            default:
                printf("Wrong Choice");
                getch();            // pause the loop to see the message
        }
    }
    return 0;
}
```
Copy

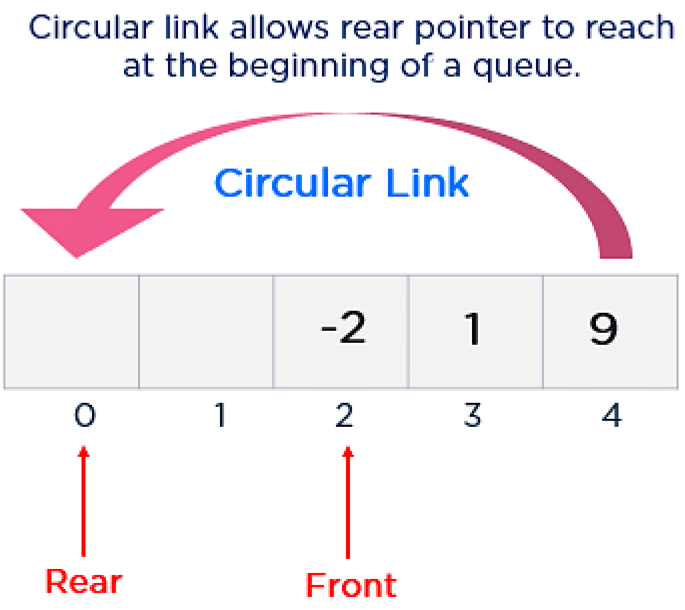
In the above program, we have defined a macro named size having value 5 using the statement #define. We can use the word size to declare the size of the array as int arr[size]. When we run the above program, the word size will be replaced by its value 5. So the size of the array will be 5.

Why Was the Concept of Circular Queue Introduced?

Implementation of a linear queue brings the drawback of memory wastage. However, memory is a crucial resource that you should always protect by analyzing all the implications while designing algorithms or solutions. In the case of a linear queue, when the rear pointer reaches the MaxSize of a queue, there might be a possibility that after a certain number of dequeue() operations, it will create an empty space at the start of a queue.



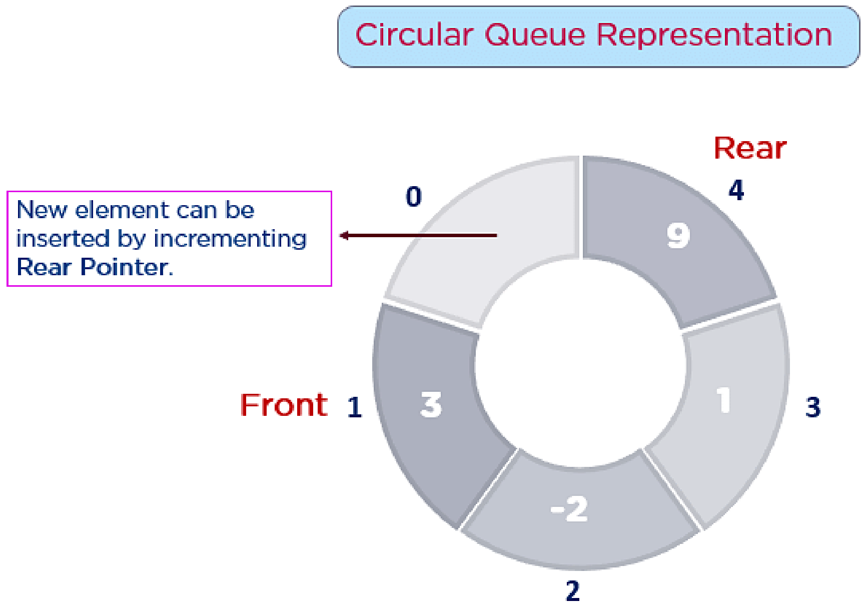
Additionally, this newly created empty space can never be re-utilized as the rear pointer reaches the end of a queue. Hence, experts introduced the concept of the circular queue to overcome this limitation.


Circular link allows rear pointer to reach at the beginning of a queue.
Circular Link
-2
1
9
0
1
2
3
4
Rear
Front

As shown in the figure above, the rear pointer arrives at the beginning of a queue with the help of a circular link to re-utilize the empty space to insert a new element. This simple addition of a circular link resolves the problem of memory wastage in the case of queue implementation. Thus, this particular type of queue is considered the best version of a queue data structure.

What is Circular Queue in a Data Structure?

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.


Circular Queue Representation
New element can be inserted by incrementing Rear Pointer.
Rear
Front
0
4
1
3
2
9
3
1
-2

As shown in the illustration above, the circular queue resolves the memory wastage problem with the help of a circular link.

How Does the Circular Queue Work?

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.

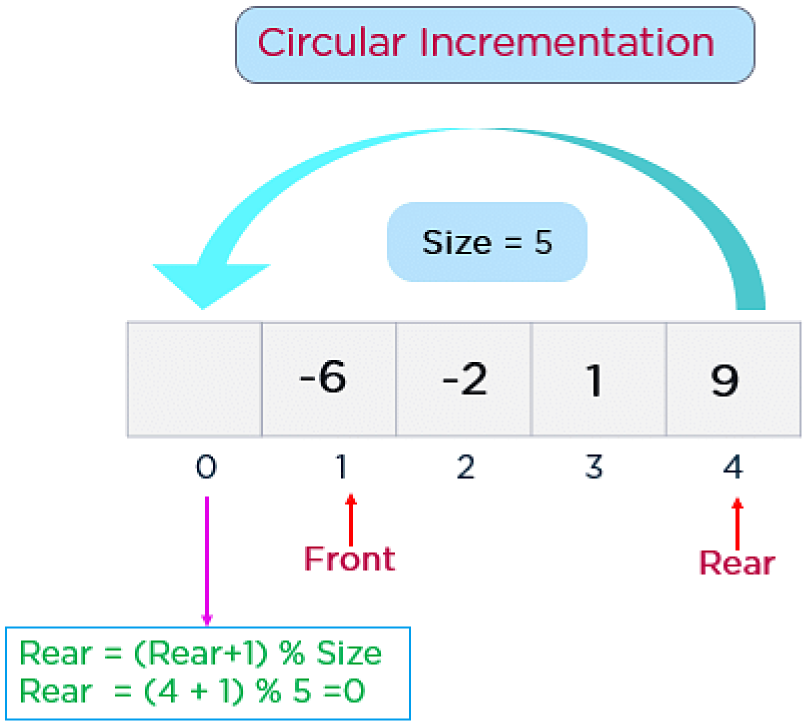
- Rear - Used to get the end element of the Circular Queue.
- enQueue(value) - Used to insert a new value in the Circular Queue. This operation takes place from the end of the Queue.
- deQueue() - Used to delete a value from the Circular Queue. This operation takes place from the front of the Queue.

Representation of Circular Queue using Arrays and a Linked List

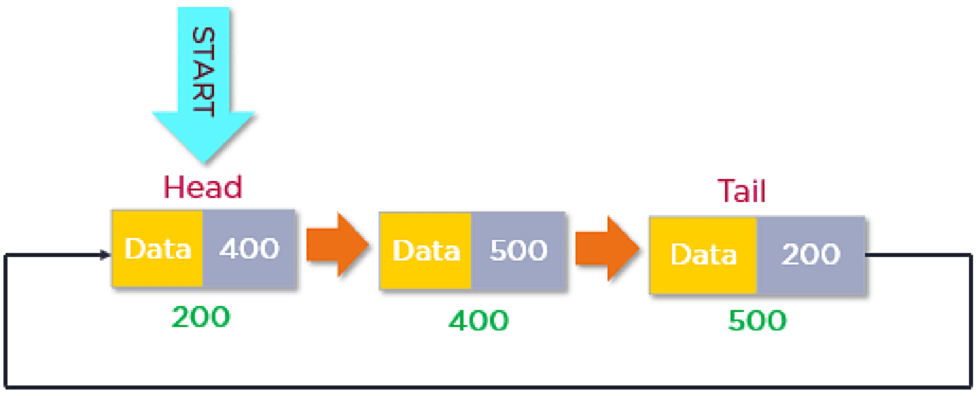
You can implement the circular queue using both the 1-D array and the Linked list. However, implementing a circular link is a new addition that you need to execute. Additionally, this queue works by the process of circular incrementation. That is, when you reach the end of a queue, you start from the beginning of a queue. The circular incrementation is achievable with the help of the modulo division.

Now you will understand how you can achieve circular incrementation, with the help of an example. Let's say the MaxSize of your queue is 5, and the rear pointer has already reached the end of a queue. There is one empty space at the beginning of a queue, which means that the front pointer is pointing to location 1.

```
Rear + 1 = 4 + 1 = 5 (Overflow Error)

Rear = (Rear + 1)% MaxSize = 0 (Reached loc. 0 / Beginning of queue)
```



Similarly, the tail node of a linked list can be connected to its head node by adding the address value of a head node in the reference field of the tail node.



Circular Link in Linked List Representation of Circular Queue

Steps for Implementing Queue Operations

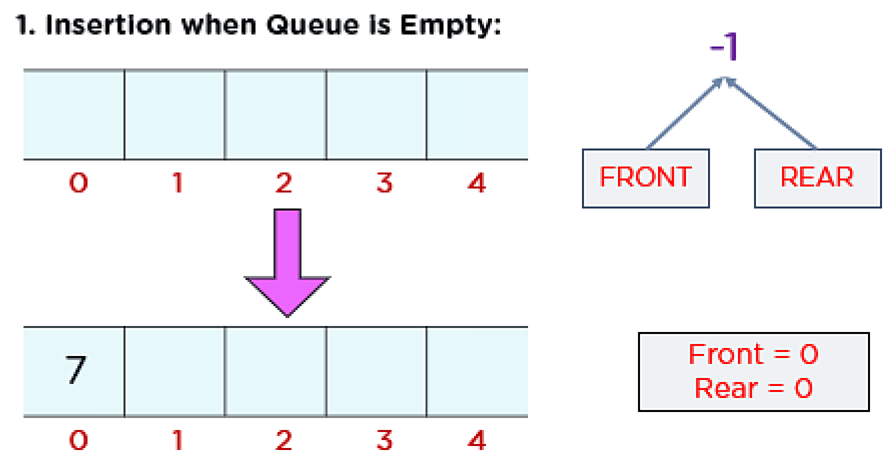
Enqueue() and Dequeue() are the primary operations of the queue, which allow you to manipulate the data flow. These functions do not depend on the number of elements inside the queue or its size; that is why these operations take constant execution time, i.e., O(1) [time-complexity]. Here, you will deal with steps to implement queue operations:

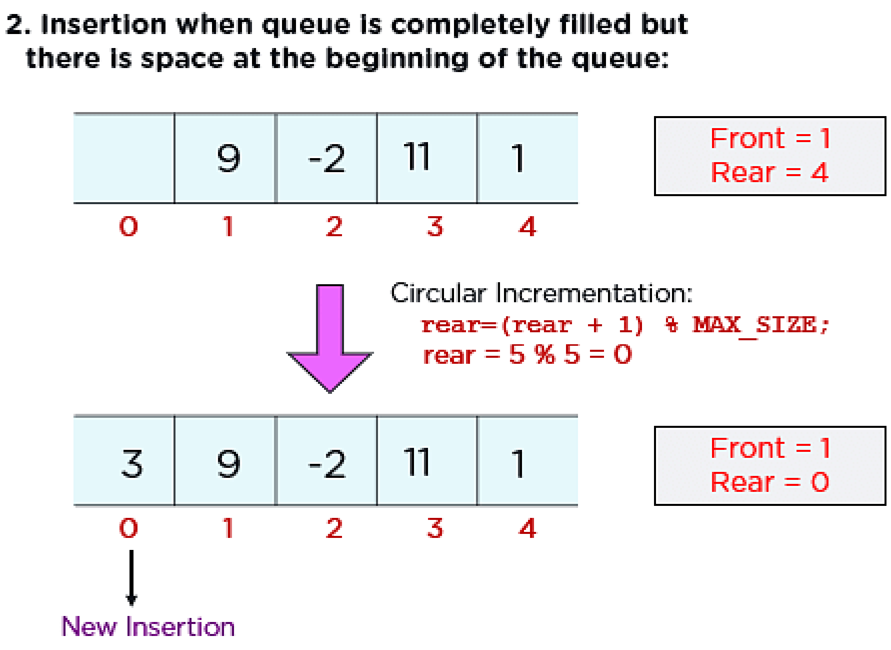
1. Enqueue(x) Operation

You should follow the following 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

Now, you will explore the Enqueue() operation by analyzing different cases of insertion in the circular queue:





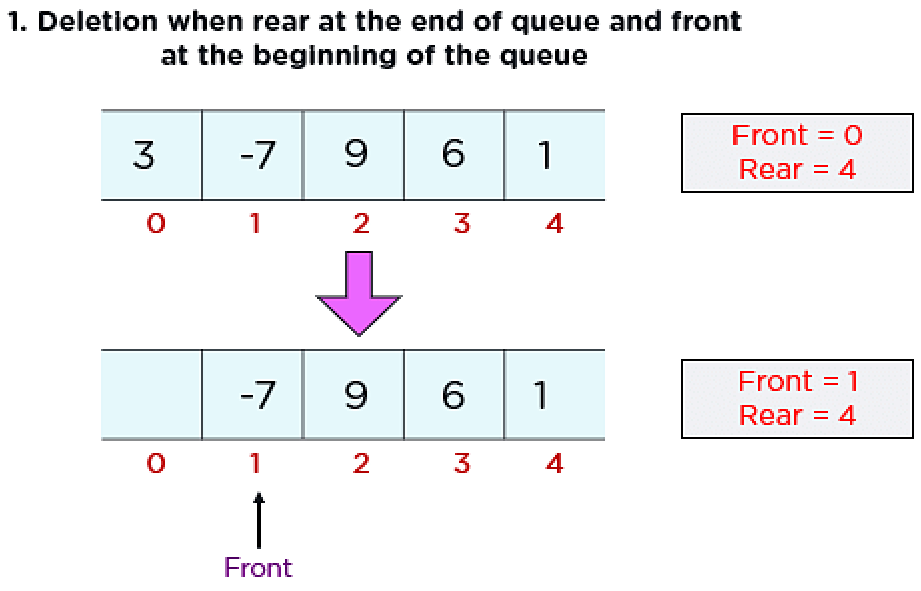
2. Dequeue() Operation

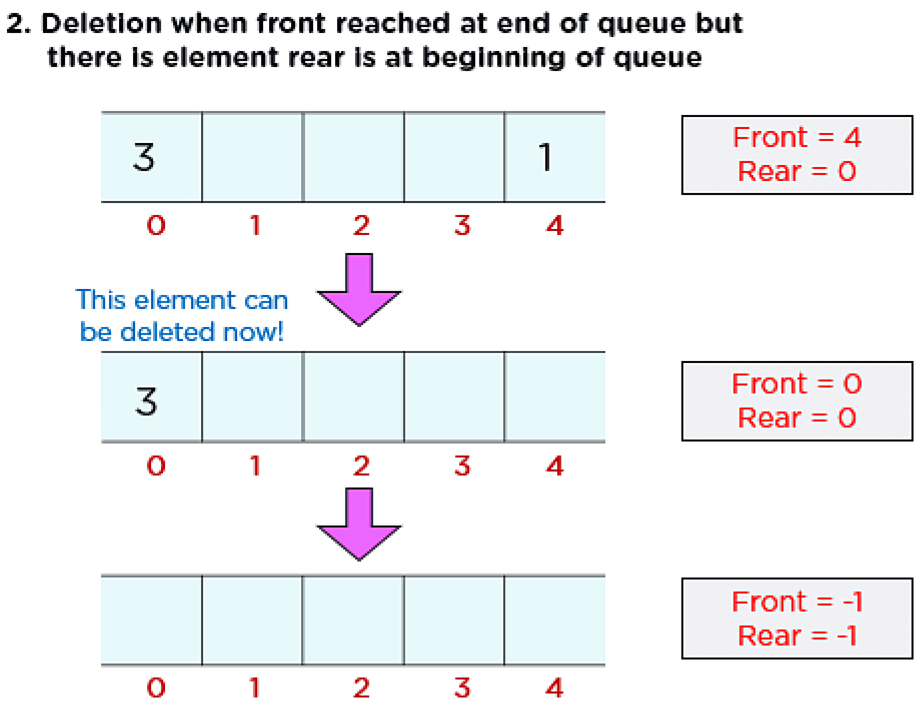
Obtaining data from the queue comprises two subtasks: access the data where the front is pointing and remove the data after access. You should take the following steps to remove data from a circular queue -

- 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

Let's understand Dequeue() operation through a diagrammatic representation:


1. Deletion when rear at the end of queue and front at the beginning of the queue
3
-7
9
6
1
0
1
2
3
4
Front = 0
Rear = 4
-7
9
6
1
0
1
2
3
4
Front = 1
Rear = 4
Front


2. Deletion when front reached at end of queue but there is element rear is at beginning of queue
3
1
0
1
2
3
4
Front = 4
Rear = 0
This element can be deleted now!
3
Front = 0
Rear = 0
Front = -1
Rear = -1

Implementation of Circular Queue using an Array

The process of array implementation starts with the declaration of array and pointer variables. You use the following statements for it:

```
#define MAX_SIZE 5    //global value assignment to max_size  variable

 int a[MAX_SIZE];

 int front = -1;

 int rear = -1;
```

After this, you will begin with the implementation of circular queue operations.

1. The first primary queue operation that allows you to manage data flow in a queue is Enqueue(). For implementing this function, you have to check if the queue is empty. If a circular queue is empty, you must set both the front and rear pointer to 0 manually. This condition can be implemented as follows:

```
void enqueue(int x)

 {

if (front == -1 && rear == -1)

{

front = rear = 0;

}
```

Later, you will check if the queue is full. If it's full, you will return an overflow error (i.e., no empty space for insertion).

```
else if ((rear + 1) % MAX_SIZE == front)

{

printf("queue is full\n");

return;

}
```

Otherwise, you must increment the rear pointer using the technique of circular incrementation. And as the rear pointer is incremented now, you can insert an element at the new location.

```
else

rear = (rear + 1) % MAX_SIZE;

a[rear] = x;

}
```

2. The next operation is Dequeue(). The Dequeue() operation removes the element from the front node of a queue. Further, an element can only be deleted when there is at least an element to delete, i.e., Rear > 0 (queue shouldn't be empty). If the queue is empty, then you must write an underflow error and exit.

```
void dequeue()

 {

if (front == -1 && rear == -1)

{

printf("queue is empty \n");

return;

}
```

Additionally, if there is only one element in the queue, you should set both the front and rear pointer to NULL.

```
else if (front == rear)

{

front = rear = -1;

}
```

Otherwise, you will simply increment the front pointer using the technique of circular incrementation. And as the front pointer is incremented, the link to the previous element's memory address gets removed, resulting in the deletion of a previous element.

```
else

front = (front + 1) % MAX_SIZE;

}
```

The complete program for the array implementation of a circular queue in programming language C is given below. The code consists of two additional functions Peek() and Print(). The Peek() function returns the element at the front node of a queue without deleting it. Meanwhile, the Print() function is created to visualize the state of a queue after each operation.

Program in C:

```
#include <stdio.h>

#include <Windows.h>
```

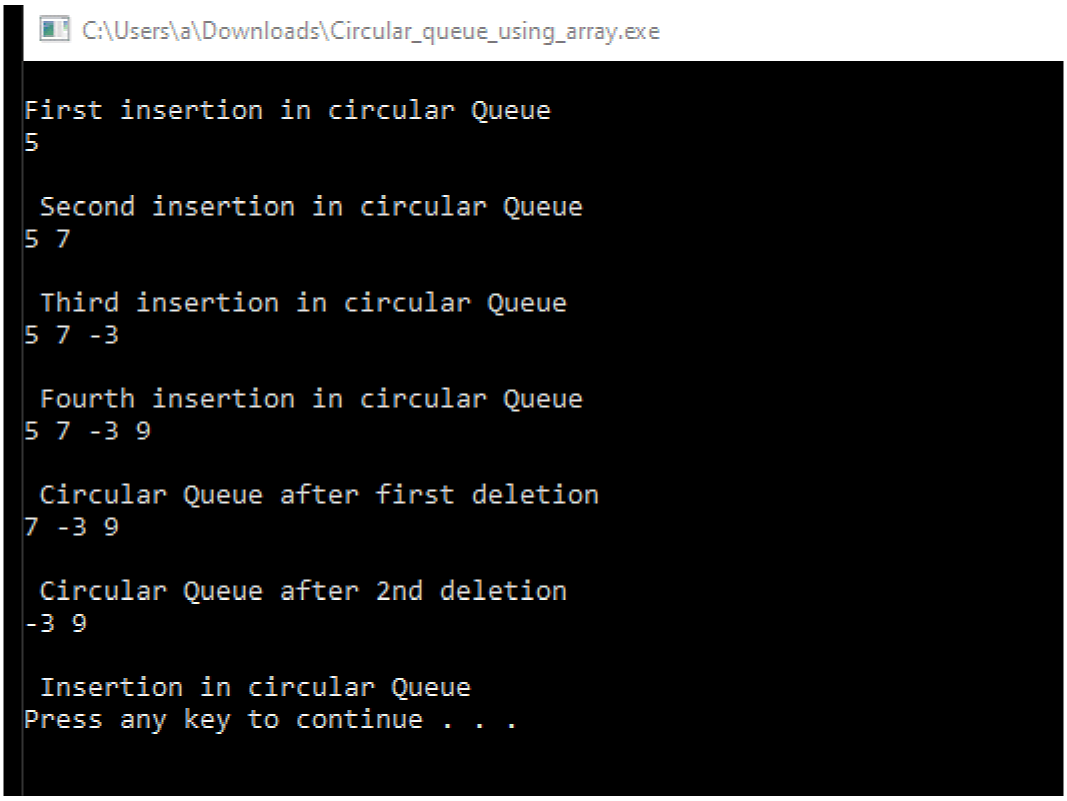
```
#define MAX_SIZE 5

int a[MAX_SIZE];

int front = -1;

int rear = -1;

void enqueue(int x)

{

if (front == -1 && rear == -1)

{

front = rear = 0;

}

else if ((rear + 1) % MAX_SIZE == front)

{

printf("queue is full\n");

return;

}

else

rear = (rear + 1) % MAX_SIZE;
a[rear] = x;

}

void dequeue()

{

if (front == -1 && rear == -1)

{

printf("queue is empty \n");

return;

}

else if (front == rear)

{

front = rear = -1;

}

else

front = (front + 1) % MAX_SIZE;

}

int Peek()
```

```
{

if (a[front] == -1)

return -1;

return a[front];

}

void Print()

{

int count = ((rear + MAX_SIZE - front) % MAX_SIZE)+1;

int i;

for (i = 0; i < count; i++)

{

printf("%d ", a[(front+i)%MAX_SIZE]);

}

printf("\n");

}

int main(void)

{
```

```
enqueue(5);

printf("\nFirst insertion in circular Queue\n");

Print();

printf("\n Second insertion in circular Queue\n");

enqueue(7);

Print();

printf("\n Third insertion in circular Queue\n");

enqueue(-3);

Print();

printf("\n Fourth insertion in circular Queue\n");

enqueue(9);

Print();

printf("\n Circular Queue after first deletion\n");

dequeue();

Print();

printf("\n Circular Queue after 2nd deletion\n");

dequeue();
```

```
Print();

printf("\n Insertion in circular Queue\n");

enqueue(14);

system("pause");

return 0;

}
```

Output:



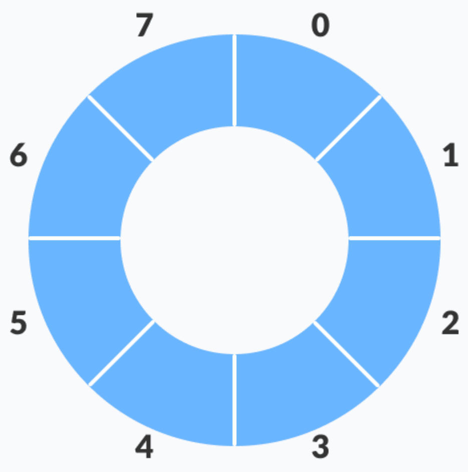
C:\Users\a\Downloads\Circular_queue_using_array.exe

```
First insertion in circular Queue
5

 Second insertion in circular Queue
5 7

 Third insertion in circular Queue
5 7 -3

 Fourth insertion in circular Queue
5 7 -3 9

 Circular Queue after first deletion
7 -3 9

 Circular Queue after 2nd deletion
-3 9

 Insertion in circular Queue
Press any key to continue . . .
```

Circular Queue Data Structure

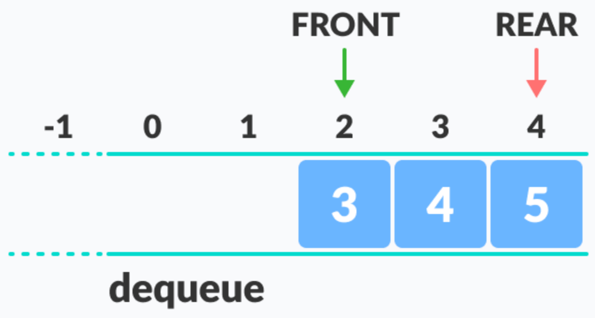
In this tutorial, you will learn what a circular queue is. Also, you will find implementation of circular queue in C, C++, Java and Python.

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus forming a circle-like structure.



Circular queue representation

The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.



Limitation of the regular Queue

Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)
```

Circular Queue Operations

The circular queue work as follows:

- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- initially, set value of FRONT and REAR to -1

1. Enqueue Operation

- check if the queue is full
- for the first element, set value of FRONT to 0
- circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by REAR

2. Dequeue Operation

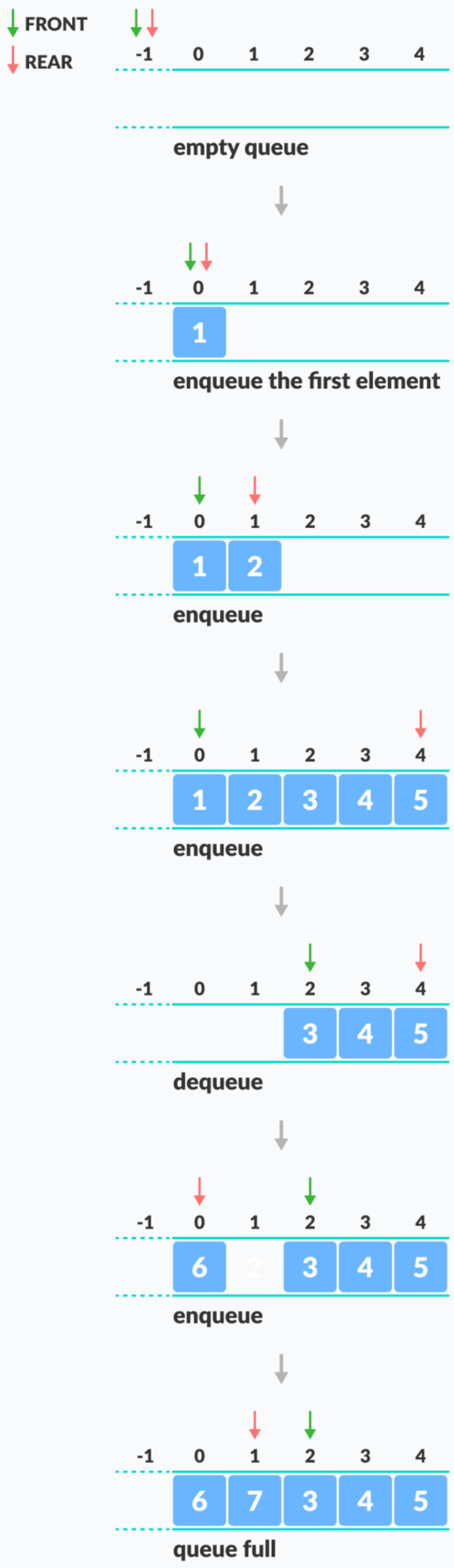
- check if the queue is empty

- return the value pointed by `FRONT`
- circularly increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1

However, the check for full queue has a new additional case:

- Case 1: `FRONT` = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`

The second case happens when `REAR` starts from 0 due to circular increment and when its value is just 1 less than `FRONT`, the queue is full.



Enque and Deque Operations

Circular Queue Implementations in Python, Java, C, and C++

The most common queue implementation is using arrays, but it can also be implemented using lists.

```c
// Circular Queue implementation in C

#include <stdio.h>

#define SIZE 5

int items[SIZE];
int front = -1, rear = -1;

// Check if the queue is full
int isFull() {
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
  return 0;
}

// Check if the queue is empty
int isEmpty() {
  if (front == -1) return 1;
  return 0;
}

// Adding an element
void enQueue(int element) {
  if (isFull())
    printf("\n Queue is full!! \n");
  else {
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    printf("\n Inserted -> %d", element);
  }
}

// Removing an element
```

```c
int deQueue() {
  int element;
  if (isEmpty()) {
    printf("\n Queue is empty !! \n");
    return (-1);
  } else {
    element = items[front];
    if (front == rear) {
      front = -1;
      rear = -1;
    }
    // Q has only one element, so we reset the
    // queue after dequeing it. ?
    else {
      front = (front + 1) % SIZE;
    }
    printf("\n Deleted element -> %d \n", element);
    return (element);
  }
}

// Display the queue
void display() {
  int i;
  if (isEmpty())
    printf(" \n Empty Queue\n");
  else {
    printf("\n Front -> %d ", front);
    printf("\n Items -> ");
    for (i = front; i != rear; i = (i + 1) % SIZE) {
      printf("%d ", items[i]);
    }
    printf("%d ", items[i]);
    printf("\n Rear -> %d \n", rear);
  }
}

int main() {
  // Fails because front = -1
  deQueue();

  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);

  // Fails to enqueue because front == 0 && rear == SIZE - 1
  enQueue(6);

  display();
  deQueue();

  display();

  enQueue(7);
  display();

  // Fails to enqueue because front == rear + 1
  enQueue(8);

  return 0;
}
```