Queue

```
#include<stdio.h>
#include<stdlib.h>
# define SIZE 3
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
int main()
{
        int ch;
        while (1)
        {
        printf("1.Enqueue Operation\n");
        printf("2.Dequeue Operation\n");
        printf("3.Display the Queue\n");
        printf("4.Exit\n");
        printf("Enter your choice of operations : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
        enqueue();
        break;
        case 2:
        dequeue();
        break;
        case 3:
        show();
        break;
        case 4:
        exit(0);
        default:
        printf("Incorrect choice \n");
        }
        }
}

void enqueue()
{
        int insert_item;
        if (Rear == SIZE - 1)
        printf("Overflow \n");
        else
        {
        if (Front == - 1)

        Front = 0;
        printf("Element to be inserted in the Queue\n : ");
        scanf("%d", &insert_item);
        Rear = Rear + 1;
        inp_arr[Rear] = insert_item;
        }
}

void dequeue()
{
        if (Front == - 1 || Front > Rear)
        {
        printf("Underflow \n");
        return ;
        }
        else
        {
        printf("Element deleted from the Queue: %d\n", inp_arr[Front]);
        Front = Front + 1;
        }
}

void show()
{

        if (Front == - 1)
        printf("Empty Queue \n");
        else
        {
        printf("Queue: \n");
        for (int i = Front; i <= Rear; i++)
        printf("%d ", inp_arr[i]);
        printf("\n");
        }
}
```

<u>Queue Program 2</u>

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

struct queue
{
	int size;
	int f;
	int r;
	int* arr;
};

int isEmpty(struct queue *q){
	if(q->r==q->f){
	return 1;
	}
	return 0;
}

int isFull(struct queue *q){
	if(q->r==q->size-1){
	return 1;
	}
	return 0;
}

void enqueue(struct queue *q, int val){
	if(isFull(q)){
	printf("This Queue is full\n");
	}
	else{
	q->r++;
	q->arr[q->r] = val;
	printf("Enqued element: %d\n", val);
	}
}

int dequeue(struct queue *q){
	int a = -1;
	if(isEmpty(q)){
	printf("This Queue is empty\n");
	}
	else{
        q->f++;
        a = q->arr[q->f];
        }
        return a;
}

int main(){
        struct queue q;
        q.size = 4;
        q.f = q.r = -1;
        q.arr = (int*) malloc(q.size*sizeof(int));

        // Enqueue few elements
        enqueue(&q, 14);
        enqueue(&q, 5);
        enqueue(&q, 1);
        enqueue(&q, 17);
        enqueue(&q, 19);
        printf("Dequeuing element %d\n", dequeue(&q));
        printf("Dequeuing element %d\n", dequeue(&q));
        printf("Dequeuing element %d\n", dequeue(&q));
        enqueue(&q, 2);
        enqueue(&q, 4);
        enqueue(&q, 7);

        if(isEmpty(&q)){
        printf("Queue is empty\n");
        }
        if(isFull(&q)){
        printf("Queue is full\n");
        }

        return 0;
}
```

<u>Circular Queue</u>

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

struct circularQueue
{
	int size;
	int f;
	int r;
	int* arr;
};


int isEmpty(struct circularQueue *q){
	if(q->r==q->f){
	return 1;
	}
	return 0;
}

int isFull(struct circularQueue *q){
	if((q->r+1)%q->size == q->f){
	return 1;
	}
	return 0;
}

void enqueue(struct circularQueue *q, int val){
	if(isFull(q)){
	printf("This Queue is full");
	}
	else{
	q->r = (q->r +1)%q->size;
	q->arr[q->r] = val;
	printf("Enqued element: %d\n", val);
	}
}

int dequeue(struct circularQueue *q){
	int a = -1;
	if(isEmpty(q)){
	printf("This Queue is empty");
	}
	else{
	q->f = (q->f +1)%q->size;
	a = q->arr[q->f];
	}
	return a;
}
```

```
int main(){
        struct circularQueue q;
        q.size = 4;
        q.f = q.r = 0;
        q.arr = (int*) malloc(q.size*sizeof(int));

        // Enqueue few elements
        enqueue(&q, 5);
        enqueue(&q, 15);
        enqueue(&q, 47);
        enqueue(&q, 4);
        enqueue(&q, 7);
        printf("Dequeuing element %d\n", dequeue(&q));
        printf("Dequeuing element %d\n", dequeue(&q));
        printf("Dequeuing element %d\n", dequeue(&q));
        enqueue(&q, 10);
        enqueue(&q, 11);
        enqueue(&q, 12);

        if(isEmpty(&q)){
        printf("Queue is empty\n");
        }
        if(isFull(&q)){
        printf("Queue is full\n");
        }

        return 0;
}
```

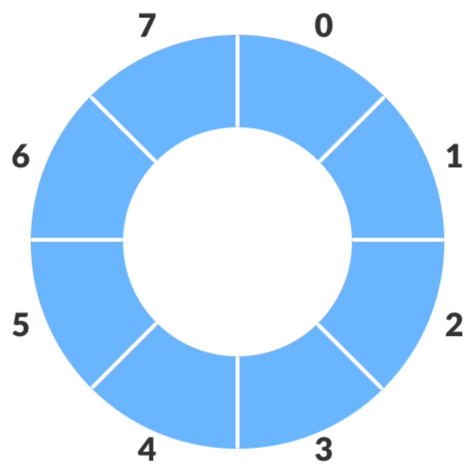
How Circular Queue Works

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,

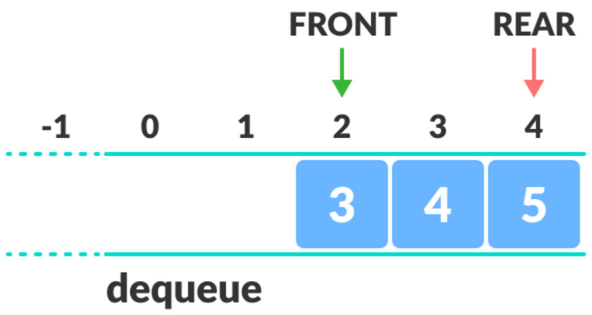
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

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.

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`

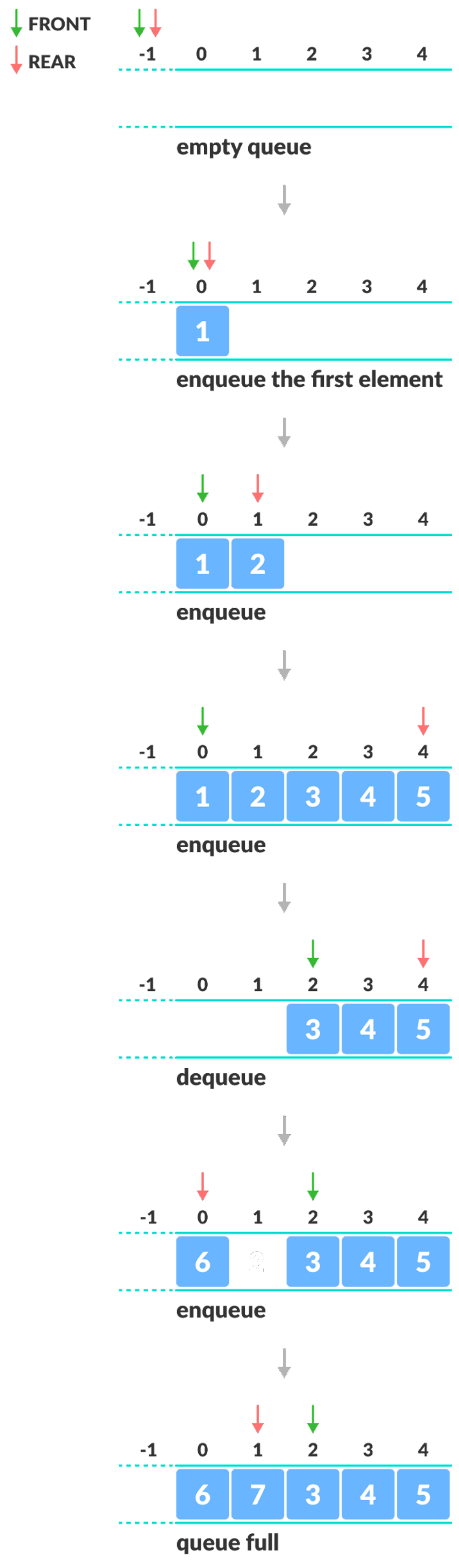
The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.

// 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
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;
}
```



Enque and Deque Operations

Applications of Circular Queue

- CPU scheduling
- Memory management
- Traffic Management