C Code For Circular Queue & Operations on Circular Queue in Hindi

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-44

We have already finished learning about the implementation of circular queues using arrays. We saw the algorithms behind enqueuing and dequeuing elements in a circular queue. We saw the conditions for circular queues to be declared full and empty. Before proceeding, if you somehow missed the last lecture, I would recommend seeing that first, since we discussed key concepts there. Today's lecture will be solely focusing on the programming part.

Note: In circular queues, the *f* is always an index behind the first element which means there is always a vacant index in circular queues.

Let's get your editors involved. I have attached the source code below. Keep it handy while understanding the code.

Understanding the code snippet below:

1. First of all, I would like you all to copy everything from the queue implementation program since things are more or less the same, and circular queues are just a variation of normal queues. So, we would just make subtle modifications and things will work well.

2. Now, since it was a queue, replace queues with circular queues. Start by changing the struct named *queue* to a struct named *circularQueue*, and all the four members remain the same as queue. (An integer variable *size* to store the size of the array, another integer variable *f* to store the index of the front end, an integer variable *r* to store the index of the rear end. And an integer pointer *arr* to store the address of the dynamically allocated array.)

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

Code Snippet 1: Declaring struct circularQueue

3. In main, we had declared a struct circularQueue *q,* and initialized its instances. Here is a subtle change, we don't initialize circular queues' *f and r* with -1, rather 0. Since -1 is unreachable in circular incrementation. Leave everything as it is.

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

Code Snippet 2: Defining and initialising a struct element *q*

4. Modifying isEmpty:

If you remember, the condition for isEmpty remains the same for both queues and circular queues. So, no modifications are needed here. Leave this as well.

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

Code Snippet 3: Modifying the *isEmpty* function

5. Modifying isFull:

Earlier, isFull checked if our rear has reached the limit of the array. And if it did, we returned the overflow statement. But now, the queue isn't full until technically. So, just see if the index next to the rear becomes front or not. Use circular increment (modulus) to pursue any increment in a circular queue.

So, check if (*r element of q) +1* is equal to the (*f element of q)*. If it is, then there is no space left in the queue to insert anymore elements, hence return 1, else 0.

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

Code Snippet 4: Modifying the *isFull* function

6. Modifying Enqueue:

In the function *enqueue*, first of all, check if the queue is full by calling the *isFull* function. If it returns 1, then print the condition of the queue overflow and return. Else, increase the *r* value of *q* circularly using the arrow operator and modulus, and insert the new value at the increased index r of the array *arr*.

```
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);
    }
}
```

Code Snippet 5: Modifying the *enqueue* function

7. Modifying Dequeue:

Earlier when we dequeued in a queue, we would simply increase the value of *f* by 1. We would now increase but circularly, and that would be it.

In the function *dequeue,* first, check whether the circular queue is already not empty by calling *isEmpty*. If it returns 1, then print the condition of the queue underflow and return. Else, increase the *f* value of *q* using the arrow operator circularly, and store the value at the index *f* of the array in some integer variable *a*. Later, return *a*.

```
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;
}
```

Code Snippet 6: Modifying the *dequeue* function

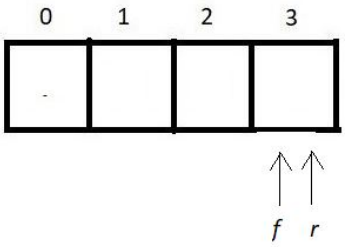
Here is the whole source code:

```c
#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, 12);
    enqueue(&q, 15);
```

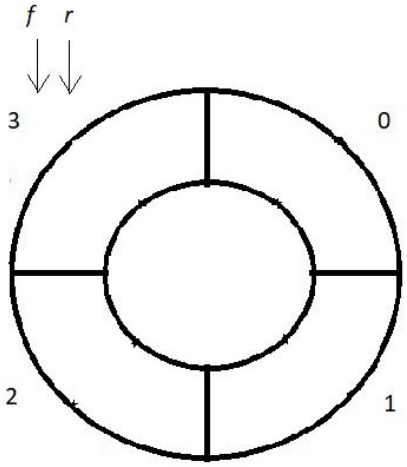
```
    enqueue(&q, 1);
    printf("Dequeuing element %d\n", dequeue(&q));
    printf("Dequeuing element %d\n", dequeue(&q));
    printf("Dequeuing element %d\n", dequeue(&q));
    enqueue(&q, 45);
    enqueue(&q, 45);
    enqueue(&q, 45);

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

    return 0;
}
```

Code Snippet 7: Implementing a circular queue and its operations using arrays

Now you can actually see what happened here. Earlier when we enqueued elements to the full and dequeued everything again as seen in the illustration below, the queue still remained full.



You can even use the queue implementation code to see that. But now when you enqueue elements to its full and delete everything, the circular queue becomes empty, unlike the normal queue. You can very easily now insert at (3+1)%4 index which is the zeroth index. I'll show you that using the code.



Let us now insert/enqueue three elements inside the queue. And see if it reverts the **full** message.

```
    enqueue(&q, 12);
    enqueue(&q, 15);
    enqueue(&q, 1);
    if(isFull(&q)){
        printf("Queue is full\n");
    }
```

Code Snippet 8: Using the *enqueue* function

Our terminal had the following output:

```
Enqued element: 12
Enqued element: 15
Enqued element: 1
Queue is full
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

Now let's dequeue everything and see if the queue again becomes empty or not.

```
    printf("Dequeuing element %d\n", dequeue(&q));
    printf("Dequeuing element %d\n", dequeue(&q));
    printf("Dequeuing element %d\n", dequeue(&q));
    if(isEmpty(&q)){
        printf("Queue is empty\n");
    }
```

Code Snippet 9: Using the *dequeue* function

And the output we received was:

```
Dequeuing element 12
Dequeuing element 15
Dequeuing element 1
Queue is empty
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above program

But had it been a normal queue, it would have still been full. I hope you understood everything. Make sure you read through the lectures again if you are having trouble understanding certain concepts. Trust me, you'll be able to crack things yourself.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial where we'll try implementing queues using linked lists. Till then keep coding.