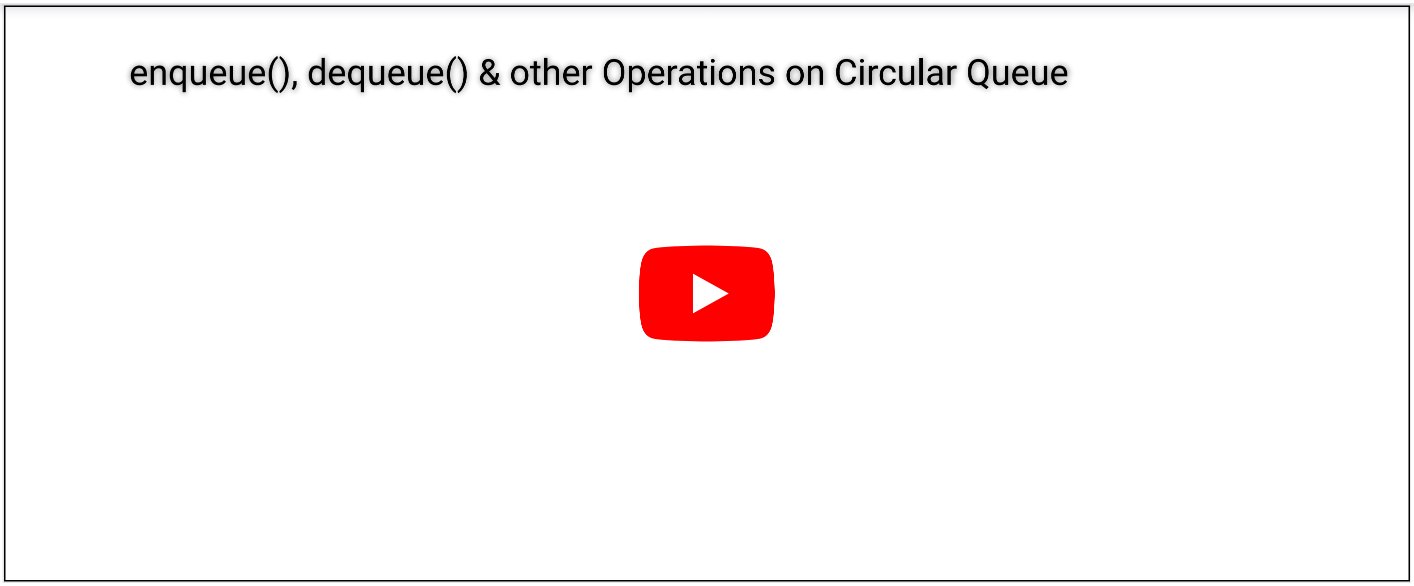
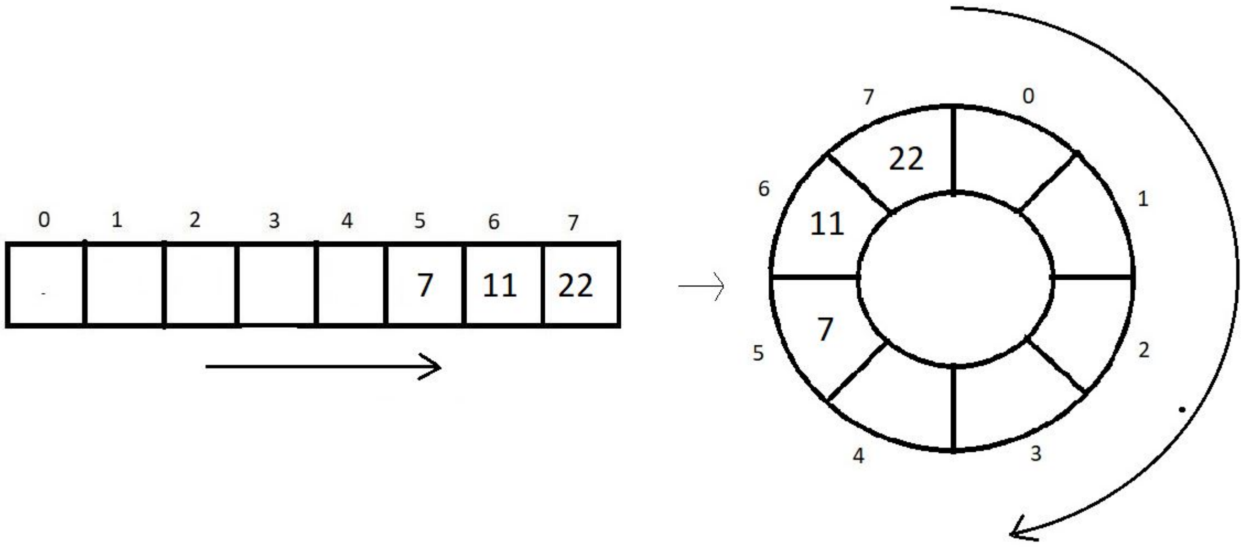


enqueue(), dequeue() & other Operations on Circular Queue

In the last tutorial, we gave you a basic introduction to circular queues and their necessity. We made you visualize the differences between a linear and a circular queue. We saw the advantages of a circular queue over a linear/normal queue. Today, we'll finish the implementation part of a circular queue and its operations using arrays.

If you remember, we converted a linear queue into a circular queue using a mathematical tool called **modulus**. This enables the feature of incrementing the indices circularly, where 0 comes after every *size -1* index. See this illustration below, and realize how a queue implemented using a linear array of size 8 couldn't utilize the memory space efficiently. Once the rear index variable reaches the limit, the queue disables further enqueuing even if the spaces behind go unused.

But once you convert this linear/normal queue into a circular one, it enables further enqueuing until the queue actually gets full. We could now reuse the vacant spaces left after a dequeue operation by going through them again and again circularly.



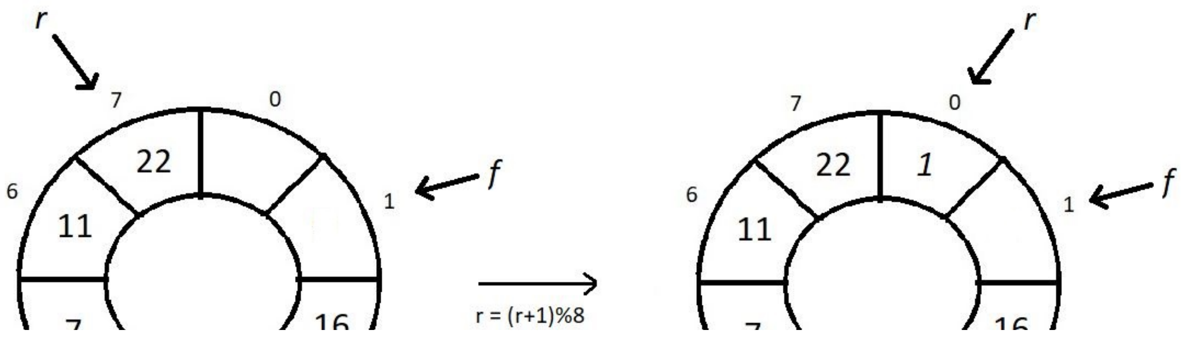
Note: Circular increment lets us access the queue indices circularly, which means, after we finish visiting the 7th index in the above illustration, we again come at the zeroth index.

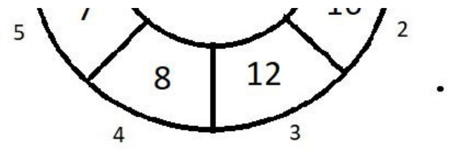
Let us now see the operations one by one.

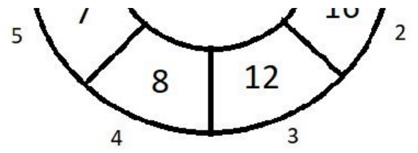
Enqueue:

Inserting a new element in a queue requires the user to input a value that we would pass into the *queue* function. Before inserting,

1. First, check if the queue is already not full. Here, the usual method to check the full condition wouldn't work. We will now check if the next index to the rear is whether the front or not.
2. If it is, it means the queue is full. Because front *f* represents the starting of the queue, and rear *r* represents the end. And the front coming next to the rear indicates that the queue is full. Therefore this is the case of queue overflow. Else just increment the rear by 1 and take its modulus by the queue's size. This is called the **circular increment**. Insert the new element there. Follow the illustration below.





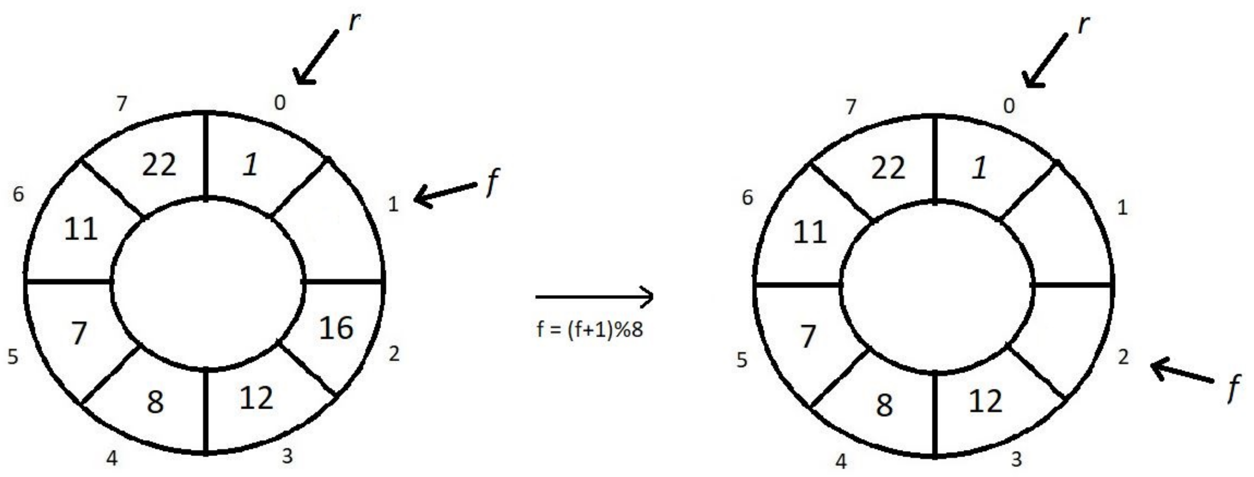


Now, since the *f* is just next to the *r,* the queue is full, and no more elements can get pushed.

Dequeue:

Dequeuing is deleting the element in a queue which is the first among all the elements to get inserted. And since the front *f* holds the index of that element, we can just remove that. But before doing that,

1. First, check if the queue is already not empty. Previously, we would just check if our front equals the rear, and if it did, we declared the queue empty. And you'll be amazed to know that it works here as well. There are zero modifications here.
2. So, if the front *f* equals the rear *r*, it is the case of queue underflow, else just increment *f* by 1 and take its modulus by the queue's size*.* While dequeuing, we store the element being removed and return it at the end. Follow the illustration below.



Condition for *isEmpty*:

1. If our *f* equals *r,* then there is no element in our queue, and this is the case of an empty queue.

Condition for *isFull*:

1. If our *(r+1)%size* equals *f,* then there is no space left in our queue, and this is the

case of a full queue.

So, we could successfully transform all our operations to work with the same efficiency for our circular queue as well. We will see the programming part in the next lecture. Don't hesitate to go over things again if you couldn't digest everything at once. Our materials are available for your review at any time.
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 learn to program the circular queue and its operations in C. Till then, keep coding.

Previous Next

CodeWithHarry