Introduction to Circular Queue in Data Structures

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-42

We have completed learning the basics of queues and its operations so far. Before proceeding to the next section, let's catch up on what we covered in queues. Using a real-life example, we explored the meaning of queue. We learn that it follows the FIFO principle. We implemented a queue ADT and its basic operations using arrays. We wrote their code in C.

When we discussed queues, we decided to have two index variables *f and r,* which would maintain the two ends of the queue. If we follow the illustration below, we would see that our queue gets full when element 8 is pushed in the queue. In other words, we can only enqueue in a queue until the queue isn't full.

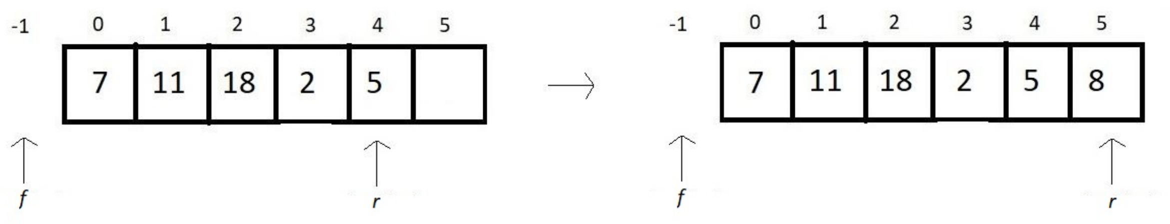


Figure 1: Using two integer variables to maintain the ends of a queue

Now, we start dequeuing some elements. Let's remove the first three elements. And now, if you carefully observe, our queue is still full since the rear end is at the array's threshold. But technically, it has space worth three elements left. And this is one characteristic cum drawback of a linear/normal queue when implemented using arrays. We don't get to efficiently utilize the space acquired by the array in the heap. Here the remaining three spaces remain unused for the whole time.

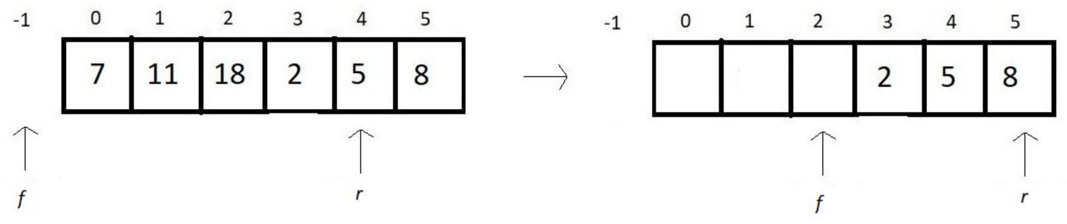


Figure 2: Dequeuing leaves vacant spaces behind

When we talk about utilizing these spaces rather than letting them go unused, we introduce circular queues.

Let's now see how we can eliminate this drawback and what modifications this situation calls for.

1. One optimizing call would be to reset *f and r* to -1 whenever the queue becomes empty, or in other words, they both become equal. This makes all the space in the array reusable. Here, the queue was full since *r* equals the *size* - 1 of the array. But resetting both the index variables to -1 empties the queue.

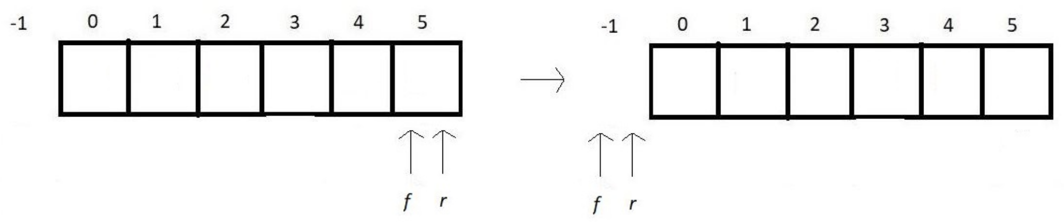


Figure 3: Resetting the index variables to -1

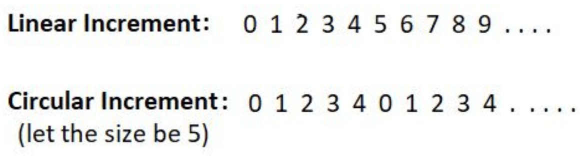
However, the efficiency of this method is limited by the requirement that the front and rear be the same. It is ineffective in the case of figure 2. Therefore we need a more optimized solution. This is when **circular queues** come to the rescue.

Circular queues:

In circular queues, we mainly focus on the point that we don't increment our indices linearly. Linearly increasing indices cause the case of overflow when our index reaches the limit, which is *size-1*.

In linear increment, i becomes i+1.

But in a circular increment ; i becomes (i+1)%size. This gives an upper cap to the maximum value making the index repeat itself.



Linear Increment: 0 1 2 3 4 5 6 7 8 9 . . . .

Circular Increment: 0 1 2 3 4 0 1 2 3 4 . . . . .
(let the size be 5)

And this makes us start from the beginning once we reach the threshold of the array. Refer to the illustration below to visualize the movement of the cursor.

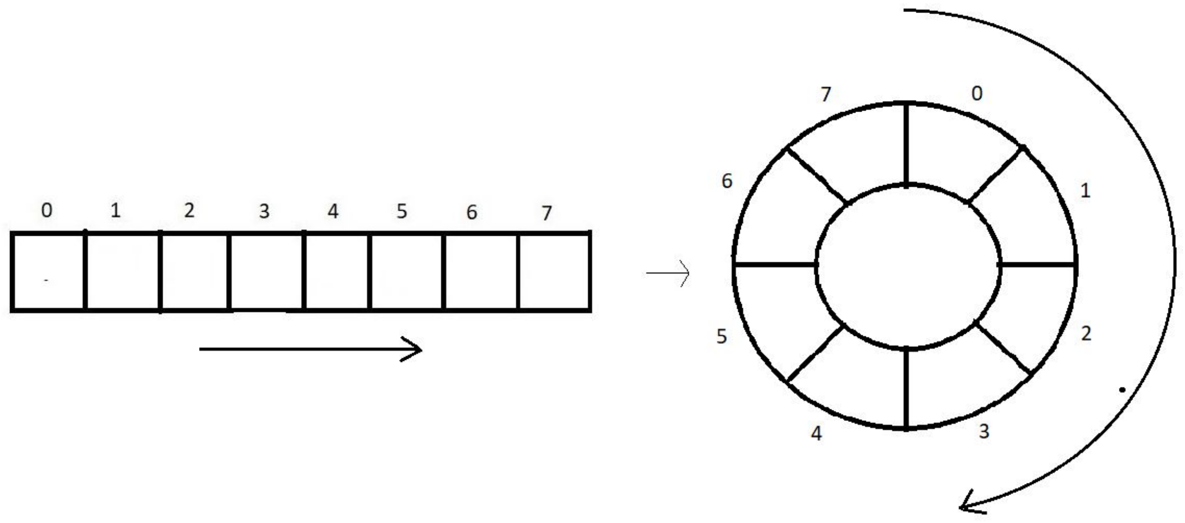


Figure 4: Conversion of a linear queue to a circular queue

And this is the circular implementation of the same array we used to implement linearly. This allows the leftover spaces to be used again. This wheel type array is called the **circular queue**.

I hope the concept behind using this circular queue is clear to you. Simply put, implementing the array circularly enables us to access the vacant spaces we left behind while deleting the elements. And now we can enqueue until the queue is actually full, and not just the way we did earlier. We'll see the operations in the next segment.

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 the operations of a linear queue in a circular queue. Till then, keep coding.