Circular Queue

Anthony Lupica

arl127@uakron.edu

Abstract

A queue for the purposes of dynamic allocation, where the user requires frequent dequeues, should be implemented as a circular queue. This method allows both enqueues and dequeues in constant time, while providing high locality of reference. Other methods compromise on these points of efficiency, namely in the form of list-based, or fixed front array-based queues.

Introduction

A queue is a kind of linear structure, which is FIFO sequenced, meaning elements may only be inserted at the rear and removed from the front. It follows from this limitation that elements nearer the front are older than elements nearer the rear. This ordering models relevant, every day, cases in which it is necessary to store some "things" for a holding period to be processed at a later time, acting as a buffer. Several implementation options exist for a queue, including static/dynamic array, and single/double linked list based solutions. A static array-based queue cannot accomplish both enqueue and dequeue operations in O(1) because elements will need to be copied and shifted down to accommodate rigid bounds.

Generally speaking, queues fall into 1 of 2 camps: fixed or unfixed. In that we mean, "is the queue's front held to a fixed position through its entire scope?" In order to design an unbounded array based queue, which by nature of using an array has good locality of reference and can perform insertions/deletes in O(1), we come to the focus of our research. A queue implemented as a circular array. Elements here have some range of indexes in [0, (capacity – 1)], but the front and rear elements are not fixed at the bounds. A circular queue, referred to also as a "ring buffer," allows the front and rear elements to "drift" within this range following a dequeue and enqueue operation respectively, but notably provides a way for them to drift indefinitely by permitting overflow back to index 0 when they would otherwise push beyond the valid range. This means that, while spatially index 0 is the front of the array, logically it loses all connection to that designation. All we concern ourselves with is tracking the front and back indexes to signal empty and full queues, and writing some method for adding capacity during runtime, which isn't too far removed from the same concern in a vector implementation.

We aim to study the implementation and show the advantages of the circular queue by comparing against a dynamic fixed version of a queue. A circular queue must prove it's worth, as it can be unintuitive and tricky to design.

Discussion

Hypothesis: in comparison to a linear, fixed, queue in which the position of the front element is constant, a circular queue more ideally optimizes space consumption. This method presents ideal circumstances for a queue where the number of contents to be enqueued is unknown, which is a common occurrence. It also a great option when the user expects to be performing deques quite often. The circular queue provides dequeue operations in O(1) time, whereas the shifting operation that a linear queue must use comes in at O(n).

Circular Queue

But, in high-level terms, how does a circular queue accomplish this goal of space optimization? Below is a queue built as a fixed array with 1 element of an arbitrary type enqueued.

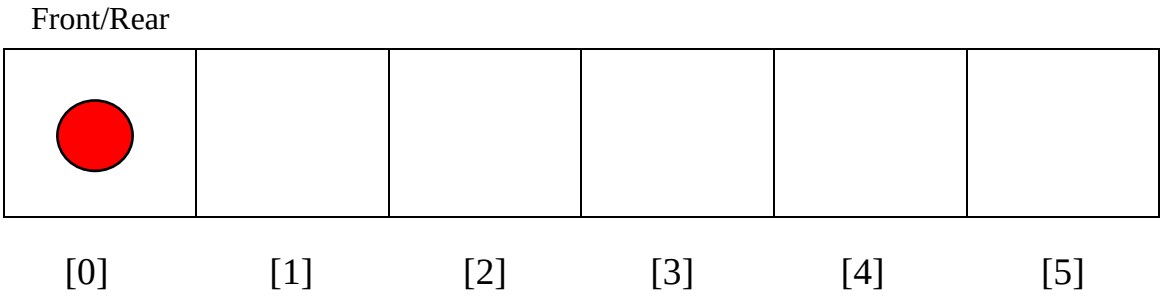


Figure 1

1 element is enqueued and the front/rear of this queue of capacity 6 share the same position (index 0). The element is colored red to aid in tracking it on its journey through the queue.

Continued enqueue operations will “stack” elements on top of our red dot toward the end of the array. Despite this somewhat counterintuitive phrasing, the front of the queue is simultaneously the oldest element in the array, considered the top element, but has the lowest numerical subscript.

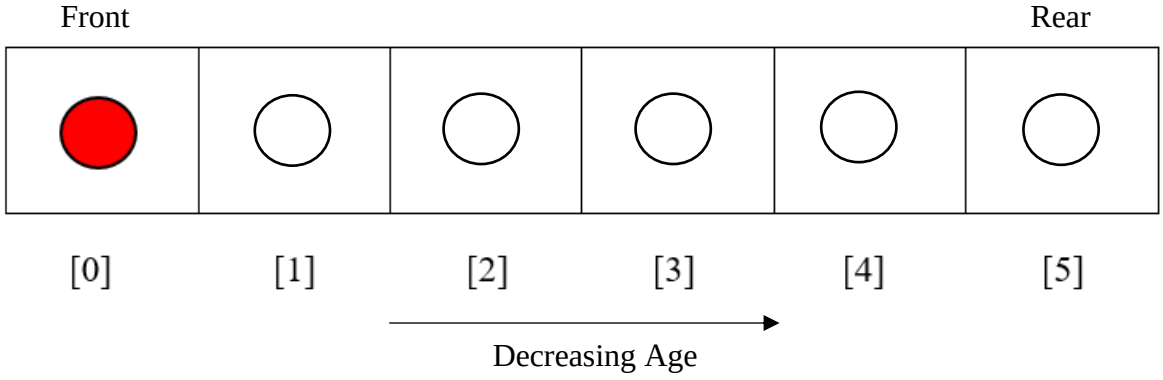


Figure 1.1

Following five enqueues, our original element holds firm on index 0 while younger members fill toward the rear. One more will cause a spill into unacceptable bounds. An arrow shows the orientation of elements from oldest to newest.

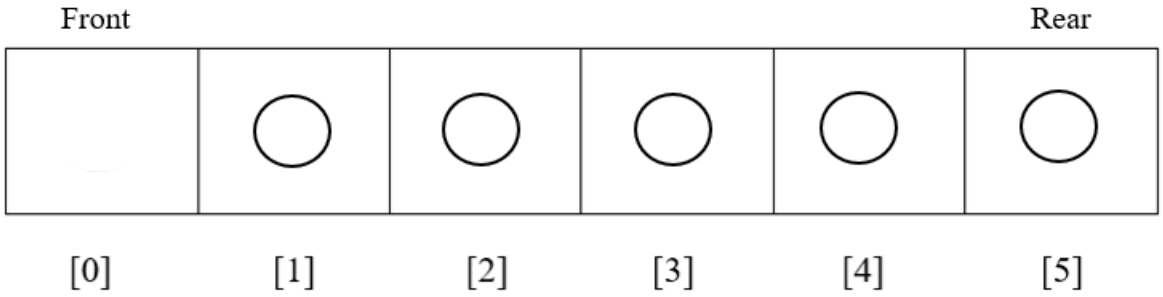


Figure 1.2

Following a single dequeue, our original element is gone. Yet, this illustrates that despite the array now having a vacant space, no additional elements can be enqueued. Our only options would be to increase the capacity of the array dynamically or copy and shift all elements down by 1, a costly move. Elements can only be enqueued at the rear, which is limited in position by the bounds of the array. This queue is capped out.

We require two things to solve the wasted space problem: allowing front to drift with dequeues just as back drifts with enqueues, and a solution to the bounds problem that drifting creates. If we configure the array such that an enqueue operation, which would otherwise violate acceptable bounds, loops back around to the “technical” front of the array, then we are able to ensure all available space is used maximally. Figure 1.3 shows this linear structure made circular. This will present some logistical concerns, including distinguishing an empty queue from a full one, which will be described in code examples.

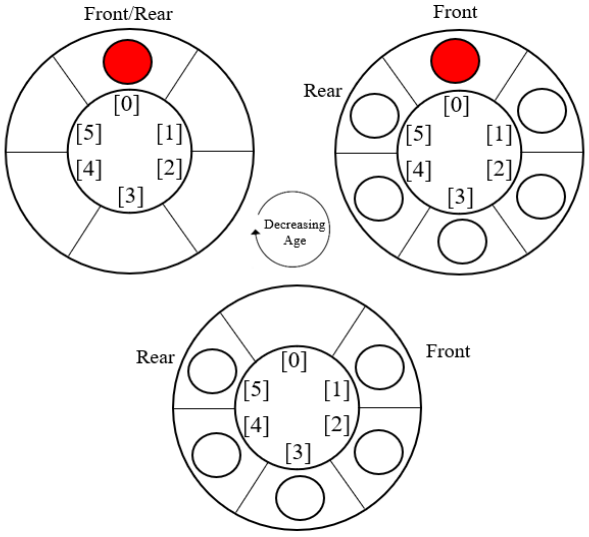


Figure 1.3

Shows the array, made circular, in three different states; initial enqueue (left), following five additional enqueues (right), and subsequent dequeue of the original element (bottom).

Now, facing the same space wastage problem as before, the indicated expression below within the enqueue method of class “CircularQueue” allows us to push another element into index 0, which will become the newest element. In this way, the positions of front and rear are made entirely arbitrary and given free rein to cycle anywhere within the subscript bounds.

Name Key

local parameters

queueVal – integer received for enqueue operation.

Data members

qArray – “queue” pointer to underlying dynamic array.

qCapacity – total number of elements the queue has space to store.

qSize – total number of elements in [0 – qCapacity) that are being stored.

iFront – tracks the index of the front element.

iBack – tracks the index of the back element.

```
void CircularQueue::enqueue(int queueVal)
{
    // if the queue is full, we call the grow helper method
    // to add capacity
    if (isFull())
    {
        grow();
    }

    // else if the queue currently has no elements, queueVal
    // becomes both the front and the rear element, and is inserted
    // at index 0
    else if (isEmpty())
    {
        iFront = iBack = 0;
        qArray[iBack] = queueVal;
        qSize = 1;

        return;
    }

    // if iBack + 1 is less than qCapacity - 1, iBack is incremented.
    // if iBack + 1 = qCapacity  - 1, iBack loops around to index 0
    iBack = (iBack + 1) % qCapacity;
    *(qArray + iBack) = queueVal;
    ++qSize;

    return;
}
```

Unlike the enqueue method for a fixed queue, which relies on a simple increment of the back index, this enqueue is concerned principally with providing iBack with the means to update circularly. The expression, iBack = (iBack + 1) % qCapacity, under two distinct input scenarios

further explains its significance. If iBack is some x: [0, qCapacity – 1) prior to enqueueing queueVal, and we assume the queue's capacity is 6 as before, then iBack = x % 6 = x + 1, and iBack increments for the next enqueue.

If, however, iBack = qCapacity – 1 prior to the enqueue, then iBack = 6 % 6 = 0. Thus, iBack will always return to index 0 at the right bound of defined subscripts, regardless of array capacity. The details of isEmpty and isFull are special in tackling how the queue keeps track of these states.

Upon uninitialized declaration of a queue object, both iFront and iBack take on the value -1 as a sort of flag for the empty state. When the first enqueue takes place, they will both be positioned at 0, since a singular element is both the front and the rear. If at any point the queue is emptied out from this point on, iFront and iBack are restored back to -1. Knowing this behavior, the bool function isEmpty uses the expression, return iFront = -1, to test for emptiness. isFull requires a slightly more complex test in the expression, return (iBack + 1) % qCapacity = iFront. With this, for example, a full queue of capacity 6 and an element in subscript 5 will produce the equivalency of 6 % 6 == iFront == 0. Notably, given the same full queue but a rear positioned at subscript 0, will mean front is at subscript 1. The same equivalence holds, with 1 % 6 == 1. In fact, this holds for all orientations seen clockwise with front directly adjacent to rear.

I.e.] (1 + 1) % 6 = 2

(2 + 1) % 6 = 3

(3 + 1) % 6 = 4

…

Then we say that if this equation holds, the queue is full and has no vacant space between front and rear, no matter how they are oriented.

Grow's solution is largely unimportant in our analysis. All that is needed is to know my version of grow doubles the capacity of the incoming array when called by a positive isFull reading, which will always begin with at least capacity 3 as defined by the default constructor. It accomplishes this by copying all elements into a temporary queue, which does make it costly; If the user new the problem size, and could guarantee it wouldn't grow, I would advise against using a dynamic queue for this reason. Both of my queue implementations exhibit this behavior for the grow method.

Now we take a look at our dequeue method, which is perhaps where the fixed/unfixed queues vary the most. The arrow highlights that the front index uses the modulus operator to attain circular motion just the same as the back index. Also, of note is how the dequeue method tests to see if the operation will result in an empty queue; if the front index has "caught up" with the back index, then we say they are the same element, and this means there is only one present. The most important aspect to take away, however, is that this design does not require that we copy all of our elements in a loop to shift them down in the array, making it the superior choice in terms of efficiency.

```
int CircularQueue::dequeue()
{
    // test if queue contains an element
    if (isEmpty() == 1)
    {
        std::cerr << "Queue is already empty\n";
        return -1;
    }

    // store value at front to be returned to the call
    int returnFront = *(qArray + iFront);

    // if front and back are equal, the element being dequeued is the only element.
    // Restore iFront and iBack to sentinel -1 to signal the empty queue.
    if (iFront == iBack)
    {
        iFront = iBack = -1;
    }
    // otherwise, increment front index
    else
    {
        iFront = (iFront + 1) % qCapacity;
    }

    --qSize; // update size member
    return returnFront;
}
```

The common case here is simply one update to iFront—no expensive looping and copying.

Fixed Queue

The enqueue method used for this version is largely identical, just opting for a simple increment in place of the modulus-based increment of the circular queue. However, the dequeue method, with significant changes, is shown below.

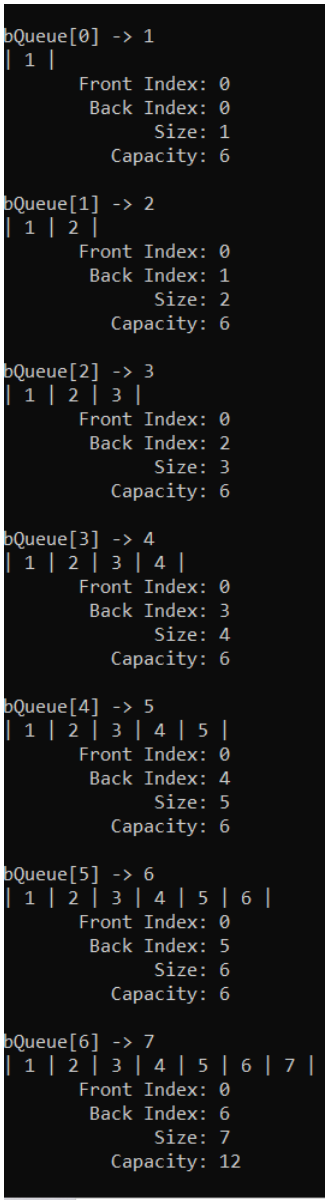
```
int BoundedQueue::dequeue()
{
    // test if queue contains an element
    if (isEmpty())
    {
        std::cerr << "Queue is already empty\n";
        return -1;
    }

    // store value at front to be returned to the call
    int returnFront = qArray[iFront];

    // if front and back are equal (and not at -1), the element being dequeued is the only element.
    // Restore iFront and iBack to sentinel -1 to signal the empty queue.
    if ((iFront == iBack) && (iBack != -1))
    {
        iFront = iBack = -1;
        --qSize;
        return returnFront;
    }
    // otherwise, perform shifting of elements
    else
    {
        for (int i = 0; i < qSize; ++i)
        {
            qArray[i] = qArray[i + 1];
        }
        std::cout << "shift performed!\n";
    }

    --qSize; // update size member
    --iBack;
    return returnFront;
}
```

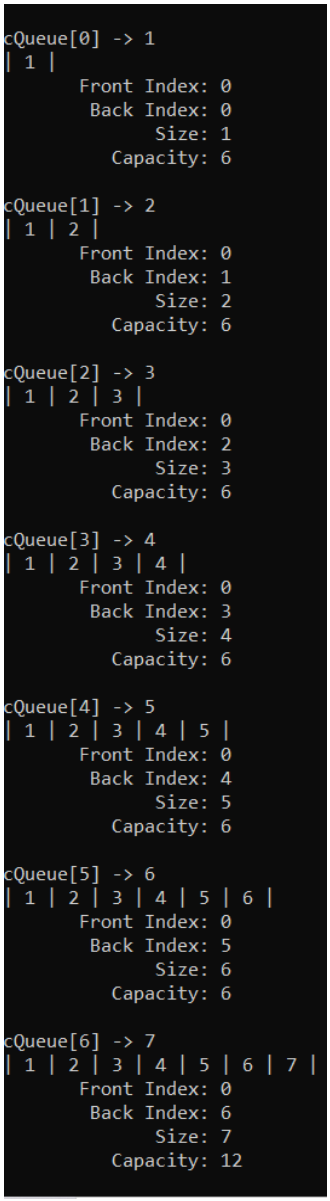
Shift

This queue wants to hold front rigid on index 0, so a dequeue operation necessitates a sliding of elements down to fill in that position. When this occurs, a message to standard output is issued to signal so.

Test Cases

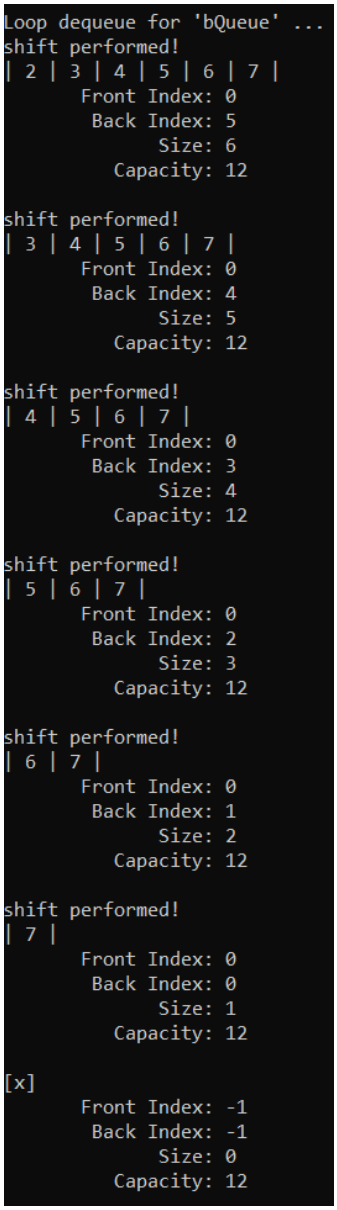
The driver code using these queue classes instantiates two objects: one a bounded queue named ‘bQueue’, and the other a circular queue named ‘cQueue’. The user is prompted to enter enqueue values in a loop for each (until they choose to stop), the result is displayed by each queue’s included display method, and all data members are revealed through accessor methods as they update.

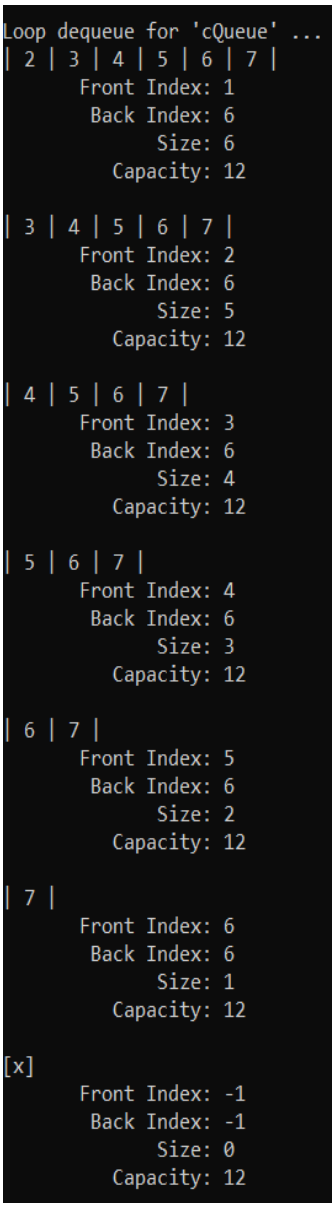




To this point it seems both operate identically, as cQueue's front index has not been moved by a dequeue yet. When the user enters a value that would push beyond the current capacity, both implementations call the grow member to double the capacity.

The user's chosen enqueues are then all dequeued in a loop, shown below.





An x is displayed afterward to show the queue has been fully emptied. In the case of bQueue, shifts are performed each time, the back index recedes to meet the front index, which is held constant. Much differently, cQueue advances the front index to meet the back index, which is held constant. This allows the circular queue to avoiding the shift altogether. In both cases, when the two indexes are equal, there is only one element left to be dequeued. In some combination of

intermixed enqueues and dequeues, the circular queue has the luxury of holding either front or back constant, which means both are done in O(1).

Conclusion

My findings conclude that a circular queue is not always the best choice but reigns supreme one specific application. If the user has a relatively small problem set, they might prefer to avoid the overhead of list nodes, and steer clear of the costly resizing of dynamic allocation in arrays. List-based queues are appealing for their ease of dynamic growth, and because queues rarely would require a search feature. They also provide a way to store aggregate data. However, if the user has a problem size that is not known ahead of time, using primitive types, and they foresee a healthy mix of enqueuing and dequeuing, they should consider a circular queue to enable both operations in O(1) time.

References

- Nikolaev, R. (n.d.). (rep.). *A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue* (pp. 1–16).
- *Difference between linear queue and circular queue*. GeeksforGeeks. (2021, January 6). Retrieved February 11, 2022, from https://www.geeksforgeeks.org/difference-between-linear-queue-and-circular-queue/
- *STD::Queue*. cplusplus.com. (n.d.). Retrieved February 11, 2022, from https://www.cplusplus.com/reference/queue/queue/
- Unit 1 DS04 Lists Stacks Queues
- Blue Tree Code. (2019, June 12). *Circular Queue Implementation - Array*. Youtube . Retrieved February 11, 2022, from https://www.youtube.com/watch?v=8sjFA-IX-Ww