

Queue

A queue is an ordered set of objects (a list) where the ordering is determined by *when* each object was inserted. With a stack, one accesses/removes the newest element (most recently added) whereas with a queue, one accesses/removes the oldest element (least recently inserted).

You are familiar with queues in daily life. You know that when you have a single resource such as a cashier in the cafeteria, you need to “join the end of the line” and the person at the front of the line is the one being served. The key property of a queue is that, among those objects/persons/etc currently in the queue, the one being served/removed is the one who first entered the queue.

The queue abstract data type (ADT) has two basic operations associated with it: `enqueue(e)` which adds an element to the queue, and `dequeue()` which removes an element from the queue. We could also have operations `isEmpty()` which checks if there is an element in the queue, and `peek()` which returns the first element in the queue (but does not remove it), and `size()` which returns the number of items in the queue. But these are not necessary for a core queue. [ASIDE: Java has a `Queue` interface, but uses the keyword `poll` instead of `dequeue` and `offer` instead of `enqueue`.]

Often one writes `add` and `remove` instead of `enqueue` and `dequeue`. Of course, these operations are very different for a queue than they are for a stack. Removing an element from a queue removes the least recently added element, whereas removing an element from a stack removes the most recently added. We say that queues implement “first come, first served” policy (also called FIFO, first in first out), whereas stacks implement a LIFO policy, namely last in, first out.

Example Suppose we add (and remove) items `a,b,c,d,e,f,g` in the following order, shown on left. On the right is show the corresponding state of the queue *after* the operation.

OPERATION	STATE
	-
<code>add(a)</code>	<code>a</code>
<code>add(b)</code>	<code>ab</code>
<code>remove()</code>	<code>b</code>
<code>add(c)</code>	<code>bc</code>
<code>add(d)</code>	<code>bcd</code>
<code>add(e)</code>	<code>bcde</code>
<code>remove()</code>	<code>cde</code>
<code>add(f)</code>	<code>cdef</code>
<code>remove()</code>	<code>def</code>
<code>add(g)</code>	<code>defg</code>

Data structures for implementing a queue

Singly linked list

One way to implement a queue is with a singly linked list. Just as you join a line at the back, when you add an element to a singly linked list queue, you manipulate the `tail` reference. Similarly, just as you serve the person at the front the queue, when you remove an item from a singly linked list queue, you manipulate the `head` reference. The `enqueue(E)` and `dequeue()` operations are equivalent to `addLast(E)` and `removeFirst()` operations from a singly linked list.

Array

Can we implement a queue using an array? Yes. But we need to be clever about it. Let’s suppose there are `size` elements in the queue and the array has `length` slots. One *inefficient* way to use an array would be to enforce that the elements are in positions 0 to position `size-1`: When we remove an element, we would remove it from position 0 and when we add an element, we would add it at position `size` (and then increment `size`). Adding an element can be done in only a few operations (assuming `size < length`). The inefficiency comes when we remove an element. We remove from position 0, so when we remove we have to shift the remaining elements from positions 1 to `size-1` by one position, so they would go from positions 0 to `size-2`. This is obviously inefficient.

A second attempt is to relax the requirement that the front of the queue is at position 0. Instead of shifting when we dequeue, we just keep track of an index `head` (and `size`) where `head` is the index of the next item to be removed.¹⁰ Note that, with this approach, both `add` and `remove` require only a few operations (independent of the length of the array). Below is the state of the array queue for the same example as above.

```

-----
0123456789.....      head   size
                        0       0
a                        0       1
ab                       0       2
 b                       1       1
bc                       1       2
bcd                      1       3
bcde                     1       4
  cde                     2       3
   cdef                    2       4
    def                    3       3
     defg                   3       4
-----

```

The problem, of course, is that when `head + size >= length` then we will exceed the length of the array. Moreover, once we remove an element from the array, we never use that array position again. This is clearly inefficient.

Circular array

To take advantage of the empty positions, we treat the array as *circular*, so that the last array position (`length-1`) is followed by position 0. The next available position is thus `(head + size) % length`. Note that this only holds when `size < length`. If this doesn’t hold, i.e. if `size == length`, then the length of the array needs to be increased (see below) if we are to add another element. So the algorithm for adding and deleting an element would go like this:

¹⁰In the context of linked lists, `head` was a reference variable. In the context of arrays, we can treat `head` as an integer index.

```

enqueue( element ){    // array implementation
    if ( size == length)
        increase length of array // *** SEE BELOW **
    a[ (head + size) % length ] = element
    size = size + 1
}

dequeue(){
    out = a[head]
    head = (head + 1) % length
    size = size - 1
    return out
}

```

Take the above example and suppose that the array has `length = 4`:

```

-----
0123...   head   size

          0       0
a          0       1
ab         0       2
 b         1       1
 bc        1       2
 bcd       1       3
ebcd       1       4
e cd       2       3
efcd       2       4
ef d       3       3
efgd       3       4
-----

```

At any time that `size` is 4, if we were to add another element then we would need to increase the length of the array.

Increasing the length of the (circular) array

To increase the length of the circular array, we create another array (say twice as big) and then copy the `length` elements to the new array. In the case of a queue, you need to be careful how you copy the elements, namely you need to copy the `head` element of the small array to position 0 in the new array, etc.

```

// copy the length elements to a new bigger array
create a bigger array
for i = 0 to length-1
    big[i] = small[ (head + i) % small.length ]

```