# Queue

Last lecture we looked at a abstract data type called a "stack". The two main operations were `push` and `pop`. I introduced the idea of a stack by saying that it was a kind of list, but with a restricted set of operations, namely we can think of `push` and `pop` as `addFirst` and `removeFirst`, respectively (or instead, `addLast` and `removeLast` – it depends on the data structure we choose).

Today we will consider another kinds of abstract data type, called a queue. A queue can also be thought of a list. However, a queue is again a very restricted type of list, since we are much more restricted in the operations that we can apply.

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 that gets served next. The fundamental property of a queue is that, among those things currently in the queue, the one that is removed next is the one who first entered the queue, i.e. the one that was least recently added. This is different from a stack, where the one that is removed next is the newest one, that is, 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.

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.

### 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 |
|---|---|
| | - |
| add(a) | a |
| add(b) | ab |
| remove() | b |
| add(c) | bc |
| add(d) | bcd |
| add(e) | bcde |
| remove() | cde |
| add(f) | cdef |
| remove() | def |
| add(g) | defg |

## 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 when the queue is implemented with 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`.

A second attempt at using an array to implement a queue 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`) which is the index of the next item to be removed.[1] Then, `tail = head + size - 1`. 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. **[EDIT Oct 16.]** Note that when the queue is initialized, `tail == -1`.

```
-------------------------------------------
01234567               head    tail   size

--------                  0    -1      0     //  to deal with size == 0
a-------                  0     0      1
ab------                  0     1      2
-b------                  1     1      1
-bc-----                  1     2      2
-bcd----                  1     3      3
-bcde---                  1     4      4
--cde---                  2     4      3
--cdef--                  2     5      4
---def--                  3     5      3
---defg-                  3     6      4
-------------------------------------------
```

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

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

A better way to use an array to implement a queue is treat the array as *circular*, so that the last array position (`length-1`) is followed by position 0. The relationship between indices would now be `tail = (head + size - 1) % length` .

[**EDIT: Oct 16**] Note that when `size == 0`, the formula implies that `tail = (head - 1) % length`, so in the special case that `head == 0`, we would have `tail == length - 1`. This is the state when the queue is initialized, for example.

The algorithm for dequeueing is as follows. The only subtlety there is that if there is just one element in the queue, then `head == tail` before we remove that element. So when we advance the `head` index, this puts the `head` one position in front of `tail`.

```
dequeue(){          //  check that size > 0  (omitted)
   element = queue[head]
   head = (head + 1) % length
   size = size - 1
   return element
}
```

What about enqueueing? The next available position is `(head + size) % length`, but only if `size < length` since if `size == length` then the array is full. In that case, the length of the array needs to be increased (see below) before we can add another element. So the algorithm for enqueueing would go like this:

```
enqueue( element ){
   if ( size == length)
      increase length of array and copy // SEE BELOW
   size = size + 1
   tail = (tail + 1) % length
   queue[tail] = element
}
```

To increase the length of the array, we create another array (say twice as big) and then copy the `length` elements to the new array. We 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 called tmpQ
for i = 0 to length-1
   tmpQ[i] = queue[ (head + i) %  queue.length ]
queue = tmpQ
```

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

```
-------------------------------------------
01234567            head    tail   size

----                  0      3      0       tail == -1 % 4 == 3
a---                  0      0      1
ab--                  0      1      2
-b--                  1      1      1
-bc-                  1      2      2
-bcd                  1      3      3
ebcd                  1      0      4
```

Suppose the next instruction were `enqueue(f)`. If we were simply to increase the length of the array and add the element to position 4, then we would get

```
ebcdf---              1      0      5
```

which would be incorrect since this would be inserting `f` between `d` and `e`.

[**BEGIN ASIDE:** In the lecture video, I discussed a slightly different example in which the full array was as follows:

```
                     head   tail   size
  efgd                 3      2      4
```

I said that the four elements should be copied to the bigger array as:

```
  defg----             0      3      4
```

and so the new element would be enqueued after the `g`. One student asked if one could keep the `efg` in the same positions in the new array, but copy the `d` to the end of the array as follows:

```
  efg----d             7      2      4
```

Again the new enqueued element would be added after the `g`. In the lecture, I responded that this would be ok, except that one must keep in mind that the `efg` still need to be copied since the bigger array really is a new array; its not just an extension of the previous array.

One more observation about this suggested alternative: Consider a slightly different example, and again consider enqueing a new element (after `g`):

```
  gdef                 1      0      4
```

This new suggested method would have to copy `def` as follows:

```
  g----def             5      0      4
```

That would be fine. [**END ASIDE**]

### Exercises

In the lecture, I briefly discussed a fun problem from the exercises of using stacks to implement a queue or vice-versa.

## ADT's, the Java API's, and interfaces

For the last several lectures we have discussed several types of abstract data types (ADT)s We began by saying what a list is in an abstract sense: a set of things and a set of methods or operations that are applied to these things. In this lecture and the last, we have seen two more ADT's: the stack and the queue. These can be thought of as lists, in the sense that they define a finite set of ordered elements.

I emphasize that ADT's are not programming language specific. They are abstract entities that exist in your head. We can generally agree on what these entities are, and we can write algorithms in pseudocode that describe how to compute things with these ADTs, and often this is helpful to do because we don't need to specify certain implementation details that don't matter.[2]

ADTs and the Java API are similar in that that the Java API for a class also specifies what are the methods for the class, and what the methods do, but it doesn't specify the full details on the implementation. The "I" in API stands for interface (application program interface).

In Java, however, the word `interface` has another meaning as well. An `interface` is set of method definitions, namely each method is defined formally by a return type, and by a method signature (a method name, and method argument types in a particular order). An `interface` does not and cannot include an implementation of the methods. Rather, in Java one needs to define a class that `implements` the interface. (Again, `implements` is a reserved word in Java. For example, there is a `List` interface (look it up in the Java API). This interface is implemented by the `ArrayList` and `LinkedList` classes, namely for each method defined in the `List` interface, there is an implemented method in the `ArrayList` and `LinkedList` class. These classes have other methods as well, which are not defined in the `List` interface.

Java has a separate `Stack` class that is not directly related to the `List` interface.
`https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html`

It has the usual stack methods (push, pop) as well as common ones that make it more useful (isEmpty and peek). It also has one which is frankly not stacklike, namely `search`. This method checks the stack for a particular object and if it is there then it returns the position in the stack. (Frankly I don't know why the people who invented the Java language chose to include this method.)

Java does not have a `Queue` class. Rather, `Queue` is an interface
`https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html`

which is implemented by many different queue classes. I mention this only for your curiosity. We will not be discussing these particular Java `Stack` and `Queue` details further in the course.

[2] We have also seen that sometimes the implementation details do matter when it comes to computational complexity. Certain operations might be $O(N)$ with one implementation of an ADT but $O(1)$ with another implementation. Sometimes we care about this, sometimes we don't. You will develop a sense for when these things matter as you get more experience.