Queues

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.*

There are many ways to implement queues, even in everyday life. Take a queue you are familiar with, namely customers waiting to be served in a shop. Many shops give a numbered ticket that you take when you come in, and this determines the order of service. This involves a different "data structure" than the lineup queue mentioned above. (Using numbered tickets allows you to move about the shop and examine the products e.g. in a bakery.) There is a third "implementation" that you are familiar with. Sometimes you enter the shop and notice that there is a set of people waiting to be served, but there is no lineup and there are not tickets to take. You don't know the order of the customers that you see. What do you do? As long as you remember which set of people was in the shop at the time you arrived, then you will be able to take your turn in proper order. (This policy works provided that everyone uses it.)

The queue abstract data type (ADT) has two basic operations associated with it: `add(e)` which adds an element to the queue, and `remove()` which removes an element from the queue. We could also have operations `isEmpty()` which checks if there is an element in the queue, and `front()` which returns the first element in the queue (but does not remove it), and `size` which returns the number of items in the queue. Notice that these operations are basically the same as those of a stack, except that their meaning is different: 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:

```
add(a)
add(b)
remove()
add(c)
add(d)
add(e)
remove()
add(f)
remove()
add(g)
```

Here is the resulting sequence of states of the queue:

```
a
ab
b
bc
bcd
bcde
cde
cdef
def
defg
```

Data structures (and algorithms) for implementing a queue

singly linked list

One simple 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. This can be done in $O(1)$ time. 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. Again this can be done in $O(1)$ time. Here the add and remove operations are equivalent to `addLast(E)` and `removeFirst()` operations from a singly linked list.

array

Can we implement a queue using an array? Yes we can! But we need to be clever about it. One *inefficient* way to do it would be to arranged the items from position 0 to position `size-1` and to remove the item from position 0. To add an element to the queue, we would add it at position `size`. This can be done in $O(1)$ time. 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 way of removing requires $\Theta($`size`$)$ operations, which is clearly undesirable.

A better solution which avoids the shift is to relax the requirement that the front of the queue is at position 0. Instead of shifting, we just keep track of two positions: `front` and `back`, `front` is the position of the next item to be removed and `back` is the position where the next item is to added. Note that, with this approach, both `add` and `remove` require $O(1)$ time.

Below is the state of the array queue for the same example as above.

```
---------------------------------------------
0123456789.........       front   back   size

                            0      0      0
a                           0      1      1
ab                          0      2      2
 b                          1      2      1
```

```
 bc                          1      3      2
 bcd                         1      4      3
 bcde                        1      5      4
  cde                        2      5      3
  cdef                       2      6      4
   def                       3      6      3
   defg                      3      7      4
---------------------------------------
```

One problem arises, of course, when we do so many adds that `back >=` N, that is, we have exceeded the capacity N of the array. If we have also done removes, then there will be positions empty from position 0 to `front - 1`. How can we take advantage of these empty positions?

(circular) array

To take advantage of the empty positions, we treat the array as *circular*, so that the last array position $(N-1)$ is "followed by" position 0. The variable `back` is still defined as the next "available" position, provided that `size <` N. Otherwise, if `size ==` N, then `back` will be the same as `front`. In this case (which is easily tested for, obviously), if an `add` occurs then the array will need to be expanded.

Take the above example and suppose that the array has $N = 4$ four positions:

```
------------------------------
0123       front   back   size

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

Notice that there is no simple relationship between the variables `back`, `front` and `size`. You might try to be clever and try defining size in terms of "`back - front` mod `N`", but notice that its a bit tricky since the case `back` = `front` occurs when `size` = 0 and also when `size` = N. When coding up an algorithm for a circular array, you will probably find it is best to update these variables independently: `back` and `front` are incremented ("mod N", to keep them in the range 0 to N-1) for operations `add` and `remove` respectively, and size is incremented and decremented, respectively. [ASIDE: The GT textbook has another solution, which is to limit the array to $N - 1$ elements

rather than N elements. This ensures that `size = 0` only happens when the array is empty (see p. 208).]

When the number of elements (`size`) exceeds the capacity N of the circular array queue, we either give an error or we expand the array. We have seen array expansions before – recall `ArrayList` (and `Vector`) from lecture 19. To expand a circular array, we do the following:

```
create a larger array aNew   //  capacity = 2N, for example
for i = 0 to N-1
   aNew[i] = aOld[ (front + i) % N]
```

Note that this copies the N elements to the new array, such that the front element is moved to position 0 in the new array. In the last example, the new array would be `defg____`.

double ended queues (called a "deque")

I briefly mentioned a related ADT, called a double ended queue. This ADT is more general than a queue, in that it allows you to add an element to the front a queue and to remove an item from the back of the queue. (It doesn't allow you to add or remove from any position other than the front or back. The the deque is less general than a "list".) Note that when I use the terms "front" and "back" here, I am not refering to any particular data structure. I am simply assuming that there is an ordering of elements, so that the front (first) position is well defined, and the back (last) position is well defined.

Also note that there would be no difference between a double ended queue and a double ended stack i.e. a double ended stack would allow you to push or pop at either end. Interestingly, one talks about double ended queues but one almost never hears about double ended stacks – even though the two are the same.

Java `Queue` interface

For your information, Java does not have a class `Queue`. Rather, it has an interface `Queue`. There are six methods defined in the interface, consisting of three pairs:

- Insert: `add(e), offer(e)`
- Remove: `remove(), poll()`
- Examine: `element(), peek()`

Within each pair, the first method throws an exception if the queue is empty; the second does not throw an exception. Feel free to check out the details, though you are not required to know them.

One final point: the `Queue` interface extends the `Collection` interface. As such, it has many non-queue-like methods. For example, you can add or remove an element from an arbitrary position. Since any class that implements `Queue` must also implement `Collection`, this means that you have arbitrary access to elements at any position in the queue. As with the Java `Stack`, this would seem to be an inappropriate use of inheritance.