Abstract Data Types - Stack and Queue

Two important abstract data types in computer science are *stacks* and *queues*. They show up all over the place.

A `Stack` is a linear data structure which is LIFO (last in first out). Think of a stack of plates. When you add a plate to the stack, you will place it on the top of the stack. When you remove a plate from the stack, you will remove the plate at the top – the last one that was added. The operations for a stack are usually called `push` to add an item and `pop` to remove an item.

A `Queue` is FIFO (first in first out). Think of a queue of people waiting at a store check-out. The first person who entered the queue will be the first one at the check-out, therefore the first to leave the queue.

We define the operations on queues which include:

Function	Explanation
`empty :: Queue a`	gives an empty queue
`isEmpty :: Queue a -> Bool`	determines if a queue is empty
`front :: Queue a -> a`	gives the item that is at the front of the queue
`enqueue :: a -> Queue a -> Queue a`	adds an item to the rear of the queue
`dequeue :: Queue a -> Queue a`	removes an item from the front of the queue

An obvious way to represent a queue is with a list. Here is a module with this representation:

```
module Queue (Queue, empty, isEmpty, front, enqueue, dequeue) where

newtype Queue a = MakeQ([a])

empty :: Queue a
empty = MakeQ([])

isEmpty :: Queue a -> Bool
isEmpty (MakeQ(q)) = null q

front :: Queue a -> a
front (MakeQ(x:q)) = x

enqueue :: a -> Queue a -> Queue a
enqueue x (MakeQ(q)) = MakeQ(q ++ [x])

dequeue :: Queue a -> Queue a
dequeue (MakeQ(x:q)) = MakeQ(q)
```

What is the efficiency for each of the operations? Let's try another representation and see if we can do better.

The problem with lists is that we have easy access to the front but to access the rear we have to traverse the entire list. So to make access to the rear easier, we will split up the queue into two lists, front and rear. The rear list holds its elements in reverse order. For example, a queue containing items [1,2,3,4,5,6] might look like front=[1,2,3] and rear = [6,5,4]. Now, adding an element to the rear of the queue is easy, resulting in front=[1,2,3] and rear =[7,6,5,4].

Removing an item from the front of the queue is easy too. Just remove the first item from the front list: front=[2,3] and rear=[7,6,5,4].

That all works fine until you dequeue all the elements from the front list. Then what can you do? Well, you can reverse the rear list and make it the front list. So after removing two more items from our example we would get front=[4,5,6,7] and rear=[].

Here is our second representation with the requirement that the rear list will never have items when the front list is empty:

```
module Queue (Queue, empty, isEmpty, front, enqueue, dequeue)where

newtype Queue a = MakeQ([a],[a])

empty :: Queue a
empty = MakeQ([],[])

isEmpty :: Queue a -> Bool
isEmpty (MakeQ(xs,ys)) = null xs

front :: Queue a -> a
front (MakeQ(x:xs,ys)) = x

enqueue :: a -> Queue a -> Queue a
enqueue x (MakeQ(xs,ys)) = makeValid (xs, x:ys)

dequeue :: Queue a -> Queue a
dequeue (MakeQ(x:xs,ys)) = makeValid (xs,ys)

makeValid :: ([a],[a]) -> Queue a
makeValid (xs,ys) =
     if null xs then MakeQ(reverse ys,[])
     else MakeQ(xs,ys)
```

Notice that the `makeValid` function enforces our requirement so that when the front list becomes empty, we reverse the rear list and make it the front. All the operations are $O(1)$ except the reverse function which is $O(n)$. Have we gained anything with this new repre-

sentation?

Well we have if we talk about *amortised* cost. This is the average cost over a worst case sequence of operations. Consider the behavior of this queue over time as we add and then remove some items. Say we add N items and then remove N items. At the point when the front list is empty and we need to reverse the rear list we have already performed N steps, over time. The reversing of the list takes N steps. So when you average the cost over time we get $O(1)$ amortised cost, even though an individual step took $O(N)$ steps.

There is another implementation of a queue which takes $O(1)$ true cost for all operations, but it is beyond the scope of this course.