Classical Queue under Persistence

If we use the classical queue persistently, the cost amortization will break:

1. Say we start with an empty queue, insert m elements.

   (m operations, $O(1)$ time each)

   `[1], [n, ... , 2]`

2. Set `q` to it so that we can reuse it many times.

3. Perform `tl q` m times.

   (m operations, $O(m)$ time each)

There are only $2m$ operations, but the time to amortize over it is $O(m^2+m)$. Reuse breaks amortization here.

Lazy Evaluation

We will use lazy evaluation to solve the above problem. First let's study lazy evaluation in detail. Lazy evaluation means:

- call by need: expression not evaluation until "needed":
  - displayed to user
  - used by body of a function being evaluated
  - pattern-matched by a function being evaluated
  - (generalization of all of the above): used by an expression being evaluated
- memoization: result of evaluation stored for later use—in fact at the place that used to hold the expression

Evaluation Example 1

Let `f = \n->n*n`. To square a thousand numbers and then just keep the first one:

```
  head (map f [1, 2, ..., 1000])
= head (f 1 : map f [2, ..., 1000])
= f 1
= 1
```

Note:

- No unnecessary squaring or list construction.
- No deep, $O(n)$ recursion.
- `map` is not tail-recursive, but it turns out to be a Good Thing!

Evaluation Example 2

Let us take the first two numbers from the squaring. This is aided by:

```
take 0 _ = []
take n (x:xs) | n>0 = x : take (n-1) xs
```

The evaluation:

```
  take 2 (map f [1, 2, ..., 1000])
= take 2 (f 1 : map f [2, ..., 1000])
= f 1 : take 1 (map f [2, ..., 1000])
= 1 : take 1 (f 2 : map f [3, ..., 1000])
= 1 : f 2 : take 0 (map f [3, ..., 1000])
= 1 : 4 : []
= [1,4]
```

Evaluation Example 3

Recall that ++ and `foldr` are not tail recursive. In fact we often define:

```
xs ++ ys = foldr (:) ys xs
```

Now we know maybe it is a Good Thing: do `[1,2,3]++[4,5,6]` and just take the head:

```
  head ([1,2,3]++[4,5,6])
= head (1 : ([2,3]++[4,5,6]))
= 1
```

There is no deep recursion, and the actual combined list is not constructed if we just ask for the first few elements!

Evaluation Example 4

We sometimes want to redefine ++ with `reverse`, which in turn is defined with `foldl` and is tail-recursive. Let us see what that costs us. To even take one element out of a reversal:

```
  head (reverse [1,2,3])
= head (foldl (flip (:)) [] [1,2,3])
= head (foldl (flip (:)) [1] [2,3])
= head (foldl (flip (:)) [2,1] [3])
= head (foldl (flip (:)) [3,2,1] [])
= head [3,2,1]
= 3
```

We have to wait for the $O(n)$ loop to finish before we can proceed, and the entire list is constructed no matter how much we need.

Lazy Evaluation Summary

Let us summarize what we learned from the examples:

- Tail recursion and `foldl` exhibit *monolithic laziness*:
  - all or nothing, flat rate
  - useful when the main operator is computational, e.g., summing a list of numbers with `(+)`, which is all-or-nothing by nature
- `foldr`, `map`, `take` exhibit *incremental laziness*:
  - one at a time, pay as you go
  - useful when the main operator is constructional, e.g., constructing a list with `(:)`, where partial results are possible

Now we can consider queues with lazy evaluation in mind.

Reflections on Classical Queue

The most expensive operation in the queue is reversing the back list and appending it to the front. Let's call it *rotation* for short.

The gist of the problem with the classical queue:

- Rotations occur too rarely and too late, when the back list is too long.
- On the other hand, rotating too frequently is not nice either, even with lazy concatenation:

  ```
  ((([1]++[2])++[3])++[4])++[5]   --inefficient
  ```

So we want to rotate when the back list is not too short and not too long, say, as soon as it becomes longer than the front list.

Lazy Queue for Persistence

Declaration:

```
data PQueue a = PQ Int [a] Int [a]
```

The parameters are: length of front list, front list, length of back list, back list.

```
empty = PQ 0 [] 0 []
hd (PQ _ (x:_) _ _) = x
```

To add an element to the queue, we add it to the back list, but then we may need to rotate. Similarly for removing an element.

```
snoc x (PQ fl fs bl bs) = rotate fl fs (bl+1) (x:bs)
tl (PQ fl (_:fs) bl bs) = rotate (fl-1) fs bl bs
```


Lazy Queue for Persistence

We rotate when the back list is longer than the front list. In other words, we make sure that the front list is always at least as long as the back list.

```
rotate fl fs bl bs
  | bl > fl = PQ (fl+bl) (fs ++ reverse bs) 0 []
  | otherwise = PQ fl fs bl bs
```


Performance of Lazy Queue

We will not prove the time complexity in full, but we will look at a few pathological examples and see that it is $O(1)$ amortized.

Just before rotation occurs, the queue may look like:

```
q = [1,...,m], [m+1,...,2m]
```

If we do `snoc` or `tl` now, a reversal will be created but left unexecuted due to laziness. It will take at least another m `tl`'s before the $O(m)$ reversal is executed. So we have $m+1$ operations to share the $O(m)$ cost. Good.

Note: `(++)` is incremental, so it adds only $O(1)$ cost to each `tl`. The worst thing that could happen to it is a lot of `snoc`'s that lead to a tree of unevaluated `(++)`'s, but note that its depth is logarithmic and decreases quickly.

Performance of Lazy Queue

If we do `tl q` m times, each take $O(1)$ time due to laziness. Good.

If we do $m+1$ `tl`'s on `q` and repeat this m times: potentially $O(m^2)$ cost, but amortized over m^2 operations. Still good.

Finally, do m `tl`'s on `q` and set the result to `r`, yielding

```
r = reverse [m+1,...,2m], []
```

so that `tl r` evaluates the reversal.

Now do `tl r` m times. What is the cost? The first `tl r` spends $O(m)$ time for the reversal, and the result is memoized:

```
r = [2m,...,m+1], []
```

So subsequent `tl r`'s take $O(1)$ each. We still have $O(2m)$ cost amortized over $2m$ operations.