Functional Datastructures

Efficiency

Consider a naive reverse definition

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Note: reverse and (++) are part of the Prelude

How many (++) calls needed to produce all elements of xs ++ ys?

O(length xs)

Efficiency

- Reversing a list takes (length xs) calls to reverse
- Each call to reverse costs

  O(length (reverse xs)) = O(length xs)
- So reversing a list of length n requires approx (n-1) +(n-2) + … + 1 = O(n*n) steps

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Fast Reverse

- Quicker reverse avoids using append. Idea: use an accumulating parameter

```
reverse :: [a] -> [a]
reverse xs = revInto [] xs
     where revInto ys []     = ys
           revInto ys (x:xs) = revInto (x:ys) xs
```

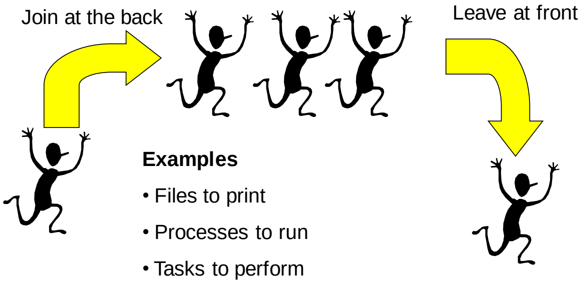
A helper function

accumulating parameter – it accumulates the answer

Data Structures

- Datatype
  - A model of something that we want to represent in our program
- Data structure
  - A particular way of *storing* data
  - How? Depending on what we want to do with the data
- Today: one example
  - Queue

What is a Queue?



Examples

- Files to print
- Processes to run
- Tasks to perform

What is a Queue?

A *queue* contains a sequence of values. We can add elements at the back, and remove elements from the front.

We'll implement the following operations:

```
empty    :: Q a               -- an empty queue
add      :: a -> Q a -> Q a   -- add element at back
remove   :: Q a -> Q a        -- remove an element from front
front    :: Q a -> a          -- inspect the front element
isEmpty  :: Q a -> Bool       -- check if the queue is empty
```

First Try

data Q a = Q [a] **deriving** (Eq, Show)

```
empty              = Q []
add x (Q xs)       = Q (xs++[x])
remove (Q (x:xs))  = Q xs
front (Q (x:xs))   = x
isEmpty (Q xs)     = null xs
```

Works, but slow

```
add x (Q xs) = Q (xs++[x])

[]     ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

As many recursive calls as there are elements in xs

Add 1, add 2, add 3, add 4, add 5…

Time is the *square* of the number of additions

A Module

- Implement the result in a *module*
- Use as specification
- Hides the internals (representation)
- Allows the re-use
  - By other programmers
  - Of the same names

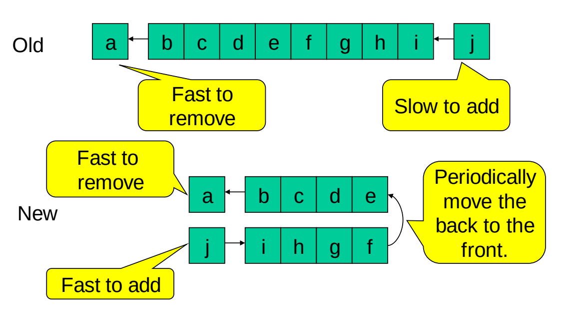
SlowQueue Module

module SlowQueue **where**

data Q a = Q [a] **deriving** (Eq, Show)

```
empty              = Q []
add x (Q xs)       = Q (xs++[x])
remove (Q (x:xs))  = Q xs
front (Q (x:xs))   = x
isEmpty (Q xs)     = null xs
```

New Idea: Store the Front and Back Separately



Smart Datatype

data Q a = Q [a] [a]
deriving (Eq, Show)

The front and the back part of the queue.

Invariant: front is empty only when the back is also empty

Smart Operations

```
empty                      = Q [] []
isEmpty q                  = q == empty
add x (Q front back)       = fixQ (Q front (x:back))
front (Q (x:front) back)   = x
remove (Q (x:front) back)  = fixQ (Q front back)
```

Move the back of the queue to the front when front becomes empty

Flipping

```
fixQ (Q [] back) = Q (reverse back) []
fixQ q           = q
```

- fixQ takes one call per element
- Each element is flipped exactly once, so
  - O(1) to add, O(1) to fixQ, O(1) to remove.

Wrapping it up

```
module Queue (Q,
              empty, add, remove,
              front, isEmpty
             ) where
```

Exports type Q but not the constructor

```
*Main> :i Q
data Q a    -- Defined at Queue.hs:11:5
*Main> front (Q [1,2] [3])
<interactive>:1:0: Not in scope: data constructor `Q'
```

Exported Constructors

```
module Queue (Q(Q),
              empty, ...
              front, ...
             ) where
```

Exports type Q and the constructor Q

Not a good idea here: allows client to
- become dependent on internal implementation details
- break datatype invariants

```
*Main> :i Q
data Q a = Q [a] [a]  -- Defined at Queue.hs:11:5
*Main> Q [] [3]
Q [] [3]
```

How can we test the smart functions?

- By using the original implementation as a *reference*
- The behaviour should be ”the same”
  - Check results
- First version is an *abstract model* that is ”obviously correct”

Later we will see:

- How to make QuickCheck work for our own datatypes
  - We need to tell it how to generate random values
- How to test the equivalence of the reference and efficient implementations
  - we need to add conversion functions
- How to test the intended invariants