User Defined Types

and efficiency as well

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]


flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square :: Float → Shape
square n = Rect n n


area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Similarly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
return :: a → Maybe a
return x = Just x


(>>=) :: Maybe a → (a → Maybe b) → Maybe b
Nothing >>= _ = Nothing
Just x  >>= f = f x
```

RECURSIVE TYPES

In Haskell, new types can be defined in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)
```

:
:
:

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function (1 +).

- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))   =   3
```

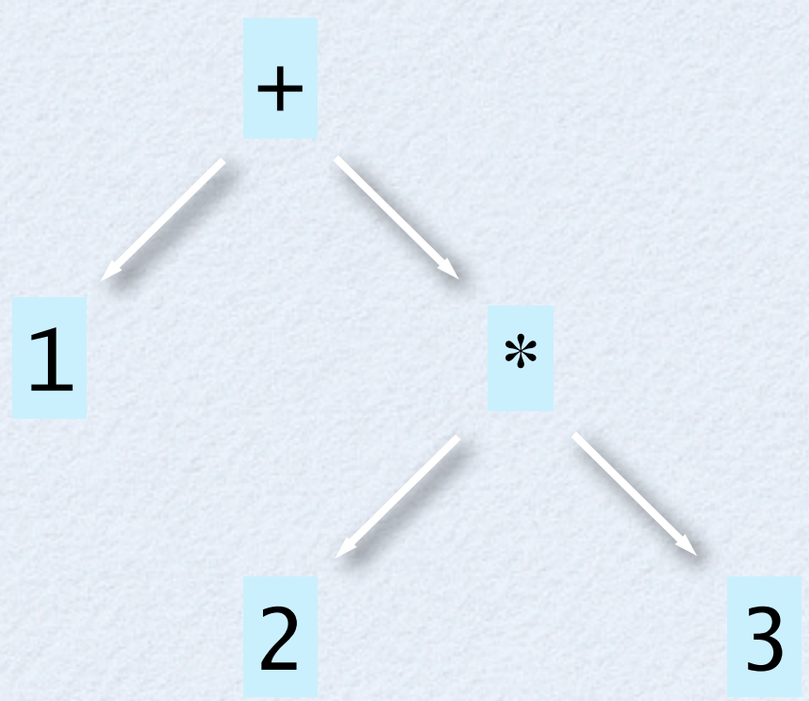
Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int :: Nat → Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n


int2nat :: Int → Nat
int2nat 0 = Zero
int2nat (n+1) = Succ (int2nat n)
```

ARITHMETIC EXPRESSIONS

Consider a simple form of expressions built up from integers using addition and multiplication.



Using recursion, a suitable new type to represent such expressions can be defined by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y


eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

EFFICIENCY

Naive definition of reversing a list

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

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

need |xs| calls to append the list

Calculate efficiency

- Reversing a list takes (length of xs) calls to reverse
- Each call to reverse takes (length of xs) calls to ++
- Reversing a list of length n requires (n-1) + (n-2) + . . . + 2 + 1 = O(n*n) steps

SPEEDIER REVERSE

Avoid using append - use an accumulating parameter

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

QUEUES

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

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

SLOW IMPLEMENTATION

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

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

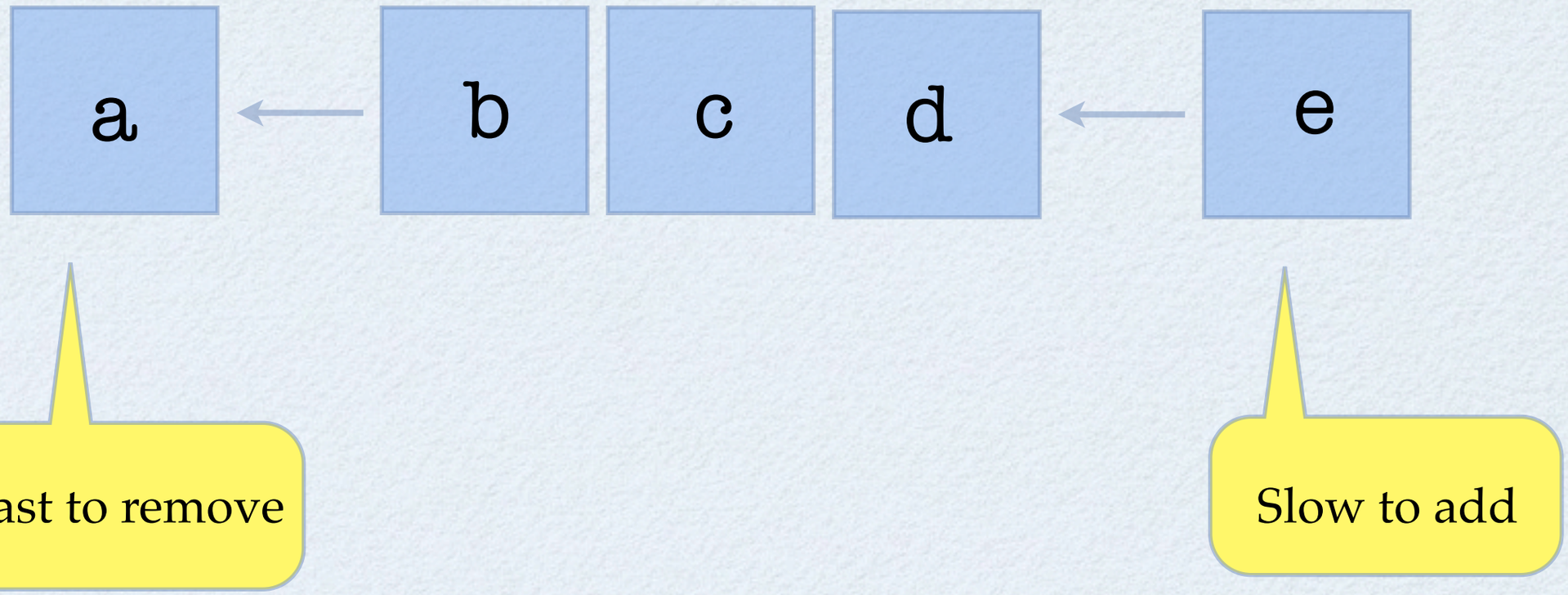
As many recursive calls as there are elements in xs

cost will be the square of the number of elements added

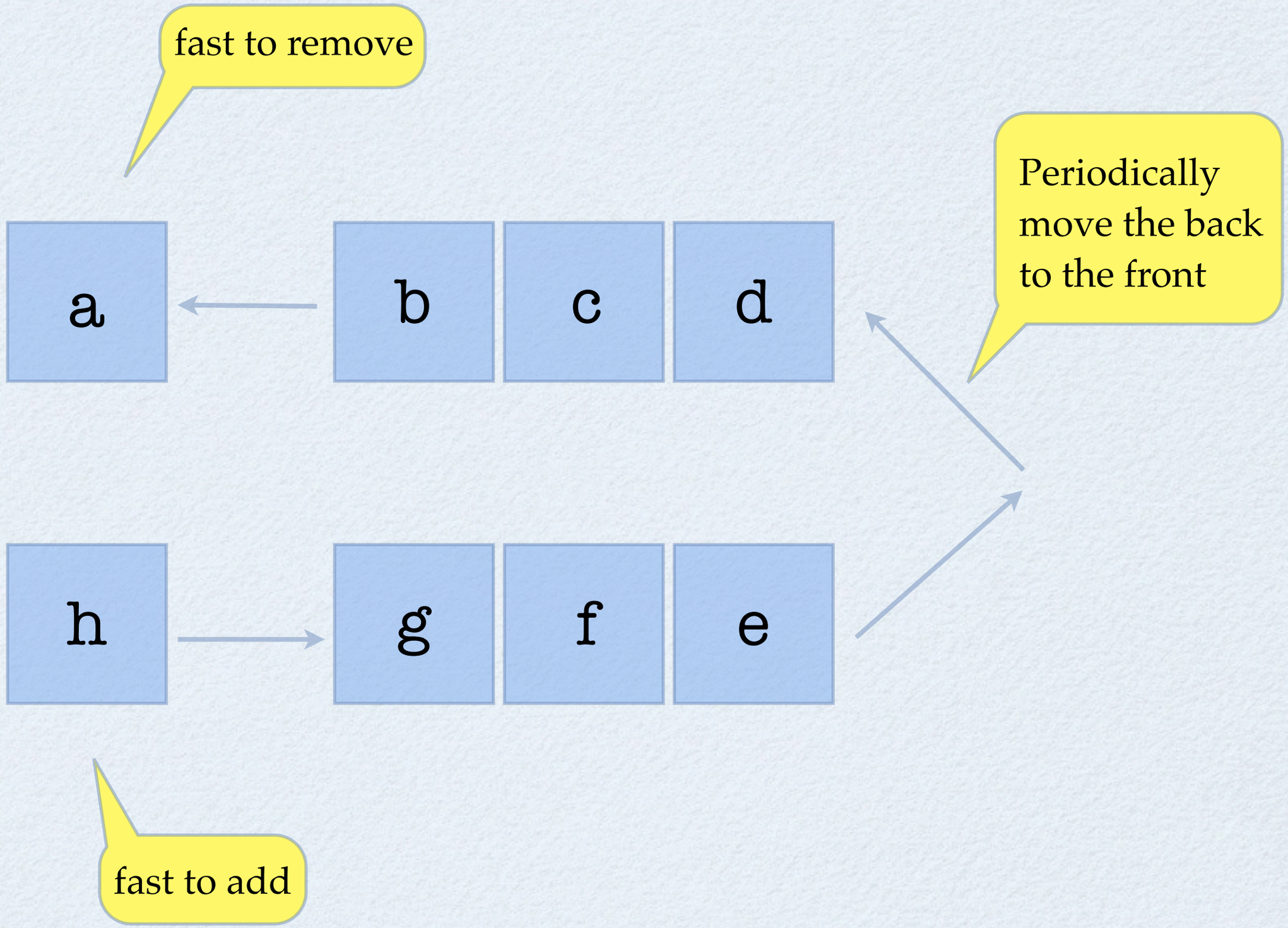
NEW IDEA

Model the back and the front of the queue separately

Old



New


fast to remove
a
b
c
d
Periodically move the back to the front
h
g
f
e
fast to add

SMART QUEUE

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

The front and the back of the queue

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

```
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 the 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 flip, O(1) to remove