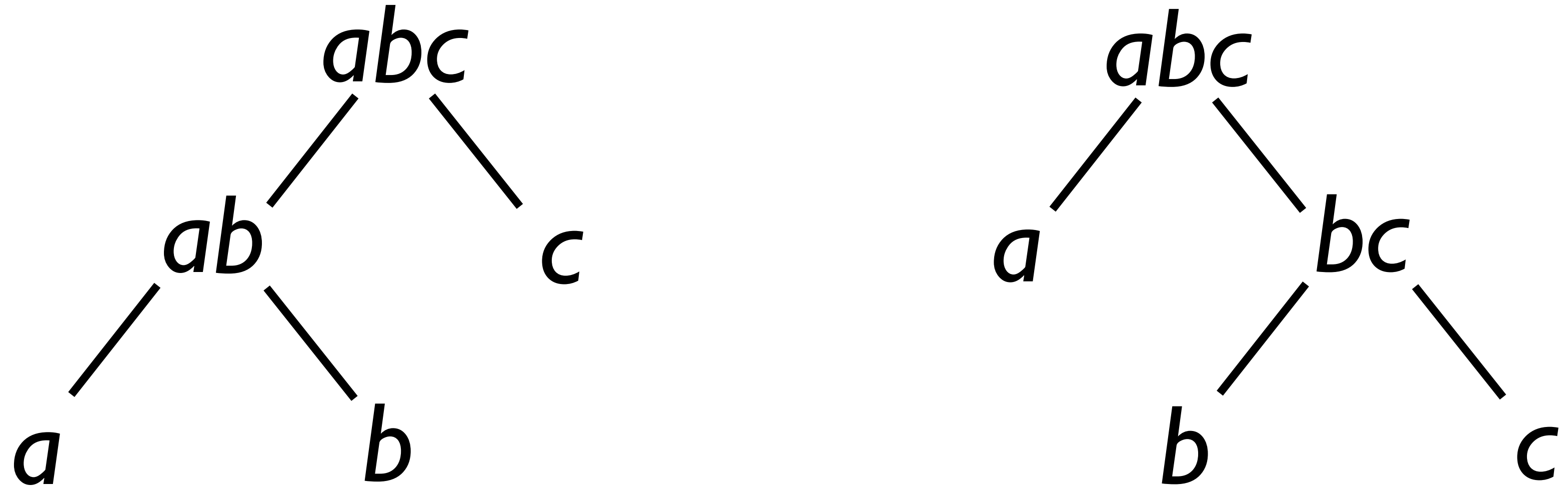
Semigroups Monoids & Trees



Matthew Brecknell

`m.brck.nl/bfpg7`

abstraction

consideration of a general quality or characteristic apart from specific instances

semigroup

an algebraic structure ⟨*S*,✧⟩

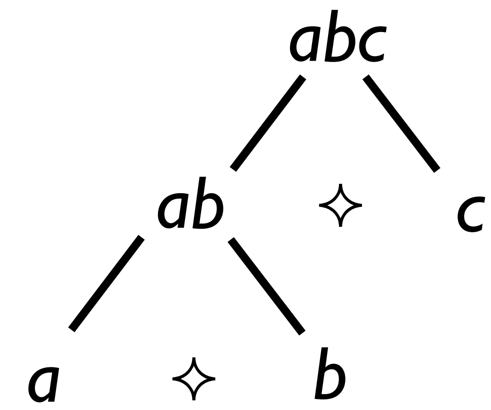
S : type

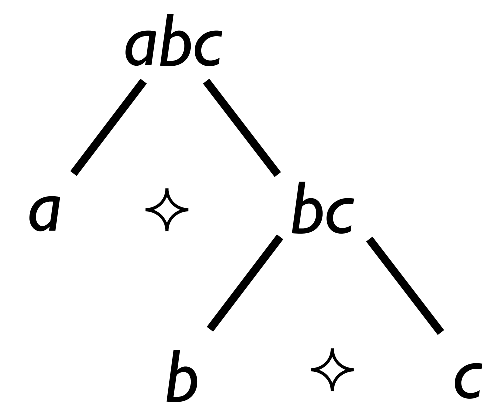
✧ : $S \to S \to S$

where ✧ is associative

$\forall\ a\ b\ c : S,$

$(a$ ✧ $b)$ ✧ $c = a$ ✧ $(b$ ✧ $c)$





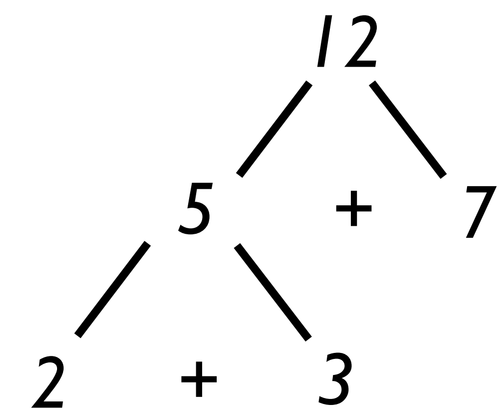
semigroup examples

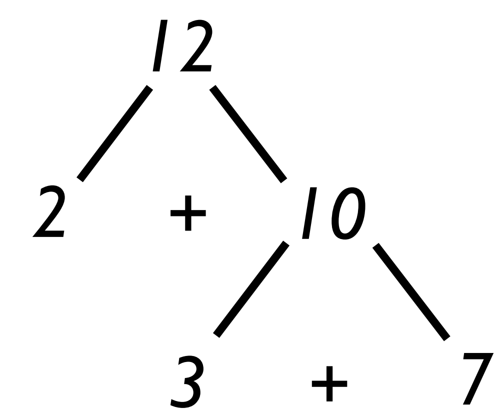
positive numbers with addition

$\langle P, + \rangle$

$P \; : \;$ type

$+ \; : \; P \to P \to P$





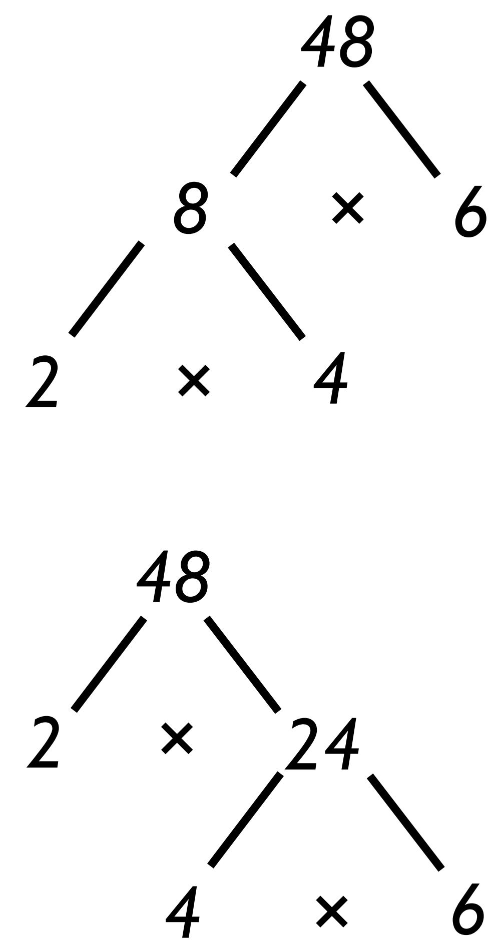
semigroup examples

even numbers with multiplication

$\langle E, \times \rangle$

E : type

$\times$: $E \rightarrow E \rightarrow E$



monoid

an algebraic structure ⟨*M*,✧,e⟩

M : type

✧ : $M \to M \to M$

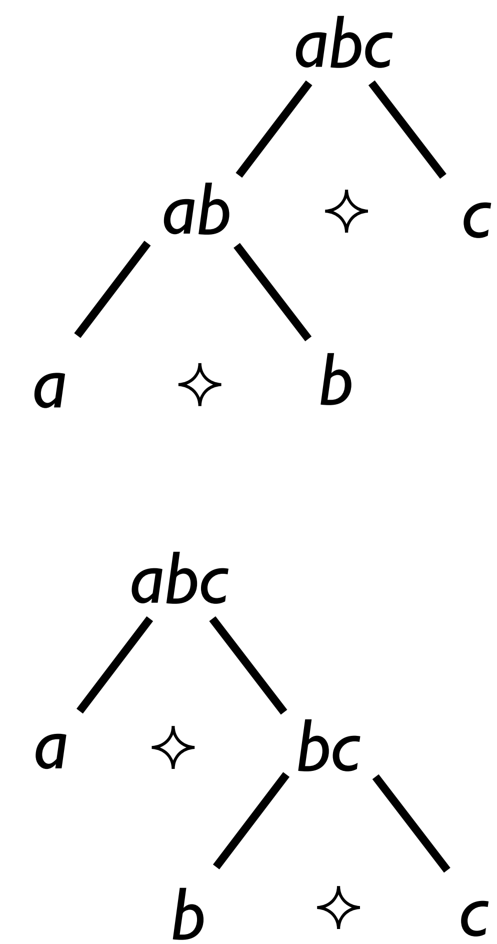
e : M

where ✧ is associative

and e is identity for ✧

∀ *m* : *M*,

  e ✧ *m* = *m* = *m* ✧ e



monoid examples

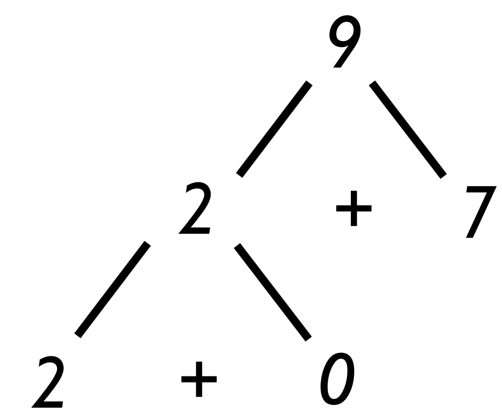
natural numbers with addition

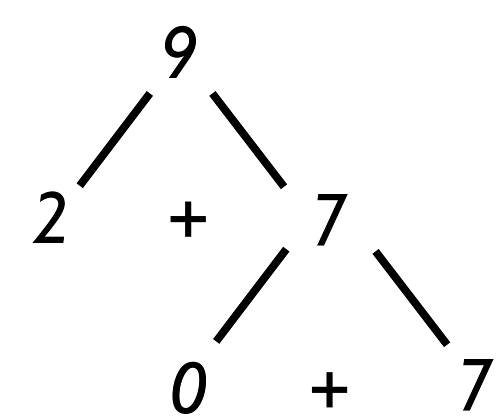
$\langle N, +, 0 \rangle$

N : type

$+$: $N \rightarrow N \rightarrow N$

0 : N





monoid examples

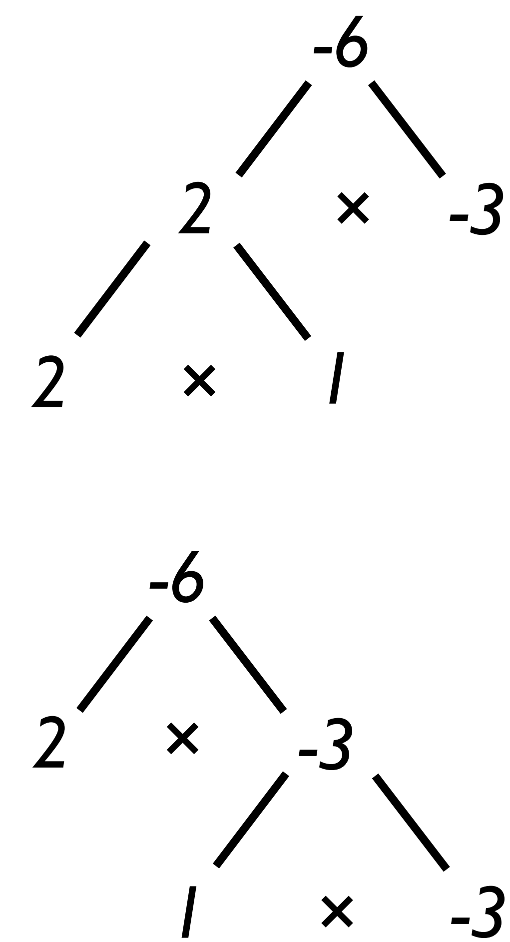
integers with multiplication

$\langle Z, \times, 1 \rangle$

$Z \ : \ \text{type}$

$\times \ : \ Z \rightarrow Z \rightarrow Z$

$1 \ : \ Z$



notation

in mathematics

- a monoid $\langle M, \diamond, e \rangle$ is identified by a *triple*,
- a semigroup $\langle S, \diamond \rangle$ is identified by a *pair*,
- we may have *many* monoids and semigroups based on the *same* underlying type.

in programming

- we want abstractions automatically selected and checked by their types,
- we have *at most one* semigroup and *one* monoid instance per type.

Haskell type classes

a disciplined approach to overloading

```
class Semigroup s where
  ✧ :: s → s → s
```

- defines a type class `Semigroup`, so that types may be subsequently declared as instances of this class,
- declares an operation ✧ which must be implemented for each instance.

Haskell type classes

keyword introducing type class definition

```
class Semigroup s where
  ✧ :: s → s → s
```

Haskell type classes

name* of type class being defined

```
class Semigroup s where
  ✧ :: s → s → s
```

* a type class name must begin with a capital letter.

Haskell type classes

introduction of type variable* ranging over the instances† of the class

```
class Semigroup s where
  ✧ :: s → s → s
```

* a type variable must begin with a lower-case letter.

† instances yet to be declared.

Haskell type classes

keyword introducing the body
of the class definition

```
class Semigroup s where
  ✧ :: s → s → s
```

Haskell type classes

```
class Semigroup s where
  ✧ :: s → s → s
```

declaration of an operation which must be defined for each instance of the class

Haskell type classes

```
class Semigroup s where
  ✧ :: s → s → s
```

“has the type”...

Haskell type classes

```
class Semigroup s where
  ✧ :: s → s → s
```

... of a binary operation...

Haskell type classes

```
class Semigroup s where
  ✧ :: s → s → s
```

... on the type of declared instances.

Haskell type class instances

```
class Semigroup s where
  ✧ :: s → s → s
```

keyword introducing type class
instance declaration

```
instance Semigroup [a] where
  ✧ = ++
```

Haskell type class instances

```
class Semigroup s where
  ✧ :: s → s → s
```

reference to class being instantiated

```
instance Semigroup [a] where
  ✧ = ++
```

Haskell type class instances

```
class Semigroup s where
  ✧ :: s → s → s
```

type* being declared an instance of
class `Semigroup`

```
instance Semigroup [a] where
  ✧ = ++
```

* lists of anything, since ‘a’ is a type variable ranging over all types.

Haskell type class instances

```
class Semigroup s where
  ✧ :: s → s → s


instance Semigroup [a] where
  ✧ = ++
```

for lists, define ✧ as list concatenation

Haskell type classes

subclasses

as before, name of type class being defined

```
class (Semigroup m) ⇒ Monoid m where
  mempty :: m
```

Haskell type classes

subclasses

context constrains `Monoid` instances to types which are already `Semigroup` instances

```
class (Semigroup m) ⇒ Monoid m where
  mempty :: m
```

Haskell type class instances

```
instance Semigroup [a] where
  ✧ = ++

instance Monoid [a] where
  mempty = []
```

the empty list is identity with respect to list concatenation

using type classes

what are the types of operations?

```
ghci> :type (✧)
(✧) :: Semigroup s ⇒ s → s → s

ghci> :type mempty
mempty :: Monoid m ⇒ m
```

context constrains usage to types which are declared as instances

using type classes

generalising algorithms over all instances of a class

constraint inherited from `Monoid` operations

```
mconcat :: Monoid m ⇒ [m] → m
mconcat = foldr (✧) mempty
```

```
foldr :: (a → b → b) → b → [a] → b
foldr f z = rec where
  rec [] = z
  rec (x:xs) = f x (rec xs)
```

monoid as a fold

generalising folds to arbitrary structures

```
class Foldable t where
  fold :: Monoid m ⇒ t m → m
```

```
ghci> :type fold
fold :: (Foldable t, Monoid m) ⇒ t m → m
```

monoid as a fold

generalising folds to arbitrary structures

```
class Foldable t where
  fold :: Monoid m ⇒ t m → m

instance Foldable [] where
  fold = foldr (✧) mempty

instance Foldable Set where
  fold = Set.foldr (✧) mempty
```

monoid as a fold

when elements are not the monoid we want to fold

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m
```

```
instance Foldable [] where
  foldMap f = foldr ((✧).f) mempty
```

```
instance Foldable Set where
  foldMap = Set.foldr ((✧).f) mempty
```

semigroup as a fold

for non-empty structures

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → t a → m
```

non-empty structures may be regarded as possibly-empty structures!

```
class Foldable t ⇒ Foldable1 t where
  foldMap1 :: Semigroup m ⇒ (a → m) → t m → m
```

constraint in contra-variant position

`Semigroup` > `Monoid` $\Longrightarrow$ `Foldable1` < `Foldable`

semigroup as a fold

for non-empty structures

```
class Foldable t ⇒ Foldable1 t where
  foldMap1 :: Semigroup m ⇒ (a → m) → t m → m

data Tree a = Node a [Tree a]
```

always contains at least one element

```
instance Foldable1 Tree where
  foldMap1 f (Node x ts) = rec x ts where
    rec x [] = f x
    rec x (Node y ts : r) = f x ✧ rec y (ts ++ r)
```

arrange recursion so that we always have an element handy, since the types prevent us from using `mempty`

more examples

booleans with logical conjunction

```
newtype All = All { getAll :: Bool }

instance Semigroup All where
  All x ◇ All y = All (x && y)

instance Monoid All where
  mempty = True

all :: Foldable t ⇒ (a → Bool) → t a → Bool
all p = getAll . foldMap (All . p)
```

why make a new type `All`?

why not make `Bool` a `Monoid` instance directly?

more examples

booleans with logical disjunction

```
newtype Any = Any { getAny :: Bool }

instance Semigroup Any where
  Any x ✧ Any y = Any (x || y)

instance Monoid Any where
  mempty = False

all :: Foldable t ⇒ (a → Bool) → t a → Bool
all p = getAny . foldMap (Any . p)
```

… because `Bool` is a `Monoid` in two different ways

more examples

numbers with addition

```
newtype Sum a = Sum { getSum :: a }

instance Num a ⇒ Semigroup (Sum a) where
  Sum x ✧ Sum y = Sum (x + y)

instance Num a ⇒ Monoid (Sum a) where
  mempty = 0

sum :: (Foldable t, Num a) ⇒ t a → a
sum = getSum . foldMap Sum
```

why make a new type `Sum`?

more examples

numbers with multiplication

```
newtype Product a = Product { getProduct :: a }

instance Num a ⇒ Semigroup (Product a) where
  Product x ✧ Product y = Product (x × y)

instance Num a ⇒ Monoid (Product a) where
  mempty = 1

product :: (Foldable t, Num a) ⇒ t a → a
product = getProduct . foldMap Product
```

… because `Num` makes a `Monoid` in two different ways

more examples

left and right bias

```
newtype First a = First { getFirst :: a }
newtype Last  a = Last  { getLast  :: a }

instance Semigroup (First a) where
  First x ✧ First y = First x

instance Semigroup (Last a) where
  Last x ✧ Last y = Last y

head1 :: Foldable1 t ⇒ t a → a
head1 = getFirst . foldMap1 First

last1 :: Foldable1 t ⇒ t a → a
last1 = getLast . foldMap1 Last
```

no `Monoid` instances?

more examples

minimum and maximum

```
newtype Min a = Min { getMin :: a }
newtype Max a = Max { getMax :: a }

instance Ord a ⇒ Semigroup (Min a) where
  Min x ✧ Min y = Min (min x y)

instance Ord a ⇒ Semigroup (Max a) where
  Max x ✧ Max y = Max (max x y)

min :: (Ord a, Foldable1 t) ⇒ t a → a
min = getMin . foldMap1 Min

max :: (Ord a, Foldable1 t) ⇒ t a → a
max = getMax . foldMap1 Max
```

more examples

constructing monoids from semigroups

```
data Maybe a = Nothing | Just a

instance Semigroup s ⇒ Semigroup (Maybe s) where
  Just x ✧ Just y = Just (x ✧ y)
  x ✧ Nothing = x
  Nothing ✧ y = y

instance Semigroup s ⇒ Monoid (Maybe s) where
  mempty = Nothing

last :: Foldable t ⇒ t a → Maybe a
last = fmap getLast . foldMap (Just . Last)

findFirst :: Foldable t ⇒ (a → Maybe b) → t a → Maybe b
findFirst p = fmap getFirst . foldMap (fmap First . f)
```

more examples

composing semigroups and monoids

```
instance (Semigroup a, Semigroup b) ⇒ Semigroup (a,b) where
  (a1,b1) ✧ (a2,b2) = (a1 ✧ a2, b1 ✧ b2)

instance (Monoid a, Monoid b) ⇒ Monoid (a,b) where
  mempty = (mempty, mempty)
```

```
instance Semigroup b ⇒ Semigroup (a → b) where
  (f ✧ g) x = f x ✧ g x

instance Monoid b ⇒ Monoid (a → b) where
  mempty = const mempty
```

more examples

going backwards

```
newtype Dual a = Dual { getDual :: a }

instance Semigroup a ⇒ Semigroup (Dual a) where
  Dual x ✧ Dual y = Dual (y ✧ x)

instance Monoid a ⇒ Monoid (Dual a) where
  mempty = Dual mempty
```

we could have defined `Last` using `Dual` and `First`

more examples

endomorphisms (functions to self)

```
newtype Endo a = Endo { appEndo :: a → a }

instance Semigroup (Endo a) where
  Endo f ✧ Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty = Endo id


foldr :: Foldable t ⇒ (a → b → b) → b → t a → b
foldr f z t = appEndo (foldMap (Endo . f) t) z

foldl :: Foldable t ⇒ (b → a → b) → t a → b
foldl f t =
  appEndo (getDual (foldMap (Dual . Endo . flip f) t) z
```

caching measures in trees

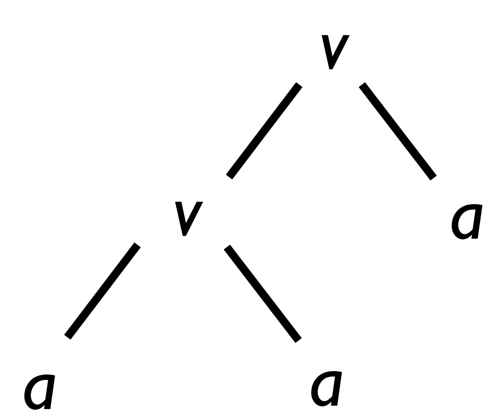
we have seen how semigroups and monoids nicely capture folds over arbitrary structures

but what happens if we build semigroup values right into the structure?

caching measures in trees

- build a balanced tree* for efficient access
- put values in the leaves
- annotate each node with a measure of its subtree

```
data Tree1 v a
  = Leaf a | Branch v (Tree1 v a) (Tree1 v a)
```



* for simplicity, we omit balancing operations from tree constructions

caching measures in trees

constructing measured trees

```
class Semigroup v ⇒ Measured a v where
  measure :: a → v

instance Measured a v ⇒ Measured (Tree1 v a) v where
  measure (Branch v l r) = v
  measure (Leaf a) = measure a

instance Measured a v => Semigroup (Tree1 v a) where
  l ✧ r = Branch (measure l ✧ measure r) l r
```

since ✧ is associative, the measure for a tree does not change if the tree is rebalanced!

caching measures in trees

deconstructing measured trees

```
type Tree v a = Maybe (Tree1 v a)

instance (Monoid v, Measured a v) ⇒ Measured (Maybe a) v
  where measure = maybe mempty id
```

find element that causes measure predicate to become `True`

```
search :: Measured a v
  ⇒ (v → Bool) → Tree v a → Maybe a
```

split at element that causes measure predicate to become `True`

```
split :: Measured a v
  ⇒ (v → Bool) → Tree v a → (Tree v a, Tree v a)
```

caching measures in trees

deconstructing measured trees

find element that causes measure predicate to become `True`

```
search :: Measured a v
  ⇒ (v → Bool) → Tree v a → Maybe a
search p (Just t) | p (measure t) = Just (left t)
  where
    left (Leaf a) = a
    left (Branch _ l r)
      | p (measure l) = left l
      | otherwise = mid (measure l) r
    mid i (Leaf a) = a
    mid i (Branch _ l r)
      | p (i ✧ measure l) = mid i l
      | otherwise = mid (i ✧ measure l) r
search _ _ = Nothing
```

caching measures in trees

lists with logarithmic operations

```
newtype Elem a = Elem a

type List a = Tree (Sum Int) (Elem a)

instance Measured (Elem a) (Sum Int) where
  measure _ = 1

(!!) :: List a → Int → Maybe a
xs !! n = search p xs
  where
    p (Sum i) < n
```

caching measures in trees

minimum priority queue

```
newtype Elem p a = Elem p a

type PQueue p a = Tree (Min Int) (Elem p a)

instance Ord p ⇒ Measured (Elem p a) (Min Int) where
  measure (Elem p _) = p

insert :: Ord p ⇒ p → a → PQueue p a → PQueue p a
insert p x q = q ✧ Just (Leaf (Elem p x))


findMin :: PQueue p a → Maybe a
findMin q = search p q
  where
    p (Min i) = i <= getSize (measure q)
```