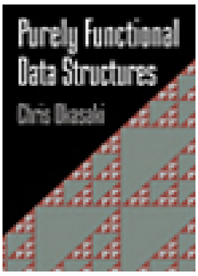

Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

A - Haskell Source Code pp. 185-206

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.013

Cambridge University Press

Appendix A
Haskell Source Code

Queues

```
module Queue (Queue(..)) where
  import Prelude hiding (head,tail)

  class Queue q where
    empty   :: q a
    isEmpty :: q a → Bool

    snoc    :: q a → a → q a
    head    :: q a → a
    tail    :: q a → q a
```

```
module BatchedQueue (BatchedQueue) where
  import Prelude hiding (head,tail)
  import Queue

  data BatchedQueue a = BQ [a] [a]

  check [] r = BQ (reverse r) []
  check f r = BQ f r

  instance Queue BatchedQueue where
    empty = BQ [] []
    isEmpty (BQ f r) = null f

    snoc (BQ f r) x = check f (x : r)

    head (BQ [] _) = error "empty queue"
    head (BQ (x : f) r) = x

    tail (BQ [] _) = error "empty queue"
    tail (BQ (x : f) r) = check f r
```

```
module BankersQueue (BankersQueue) where
  import Prelude hiding (head,tail)
  import Queue

  data BankersQueue a = BQ Int [a] Int [a]

  check lenf f lenr r =
    if lenr ≤ lenf then BQ lenf f lenr r
    else BQ (lenf+lenr) (f ++ reverse r) 0 []

  instance Queue BankersQueue where
    empty = BQ 0 [] 0 []
    isEmpty (BQ lenf f lenr r) = (lenf == 0)

    snoc (BQ lenf f lenr r) x = check lenf f (lenr+1) (x : r)
```

```
head (BQ lenf [] lenr r) = error "empty queue"
head (BQ lenf (x : f') lenr r) = x

tail (BQ lenf [] lenr r) = error "empty queue"
tail (BQ lenf (x : f') lenr r) = check (lenf-1) f' lenr r
```

```
module PhysicistsQueue (PhysicistsQueue) where
  import Prelude hiding (head,tail)
  import Queue

  data PhysicistsQueue a = PQ [a] Int [a] Int [a]

  check w lenf f lenr r =
    if lenr ≤ lenf then checkw w lenf f lenr r
    else checkw f (lenf+lenr) (f ++ reverse r) 0 []

  checkw [] lenf f lenr r = PQ f lenf f lenr r
  checkw w lenf f lenr r = PQ w lenf f lenr r

  instance Queue PhysicistsQueue where
    empty = PQ [] 0 [] 0 []
    isEmpty (PQ w lenf f lenr r) = (lenf == 0)

    snoc (PQ w lenf f lenr r) x = check w lenf f (lenr+1) (x : r)

    head (PQ [] lenf f lenr r) = error "empty queue"
    head (PQ (x : w) lenf f lenr r) = x

    tail (PQ [] lenf f lenr r) = error "empty queue"
    tail (PQ (x : w) lenf f lenr r) = check w (lenf-1) (Prelude.tail f) lenr r
```

```
module HoodMelvilleQueue (HoodMelvilleQueue) where
  import Prelude hiding (head,tail)
  import Queue

  data RotationState a =
           Idle
         | Reversing Int [a] [a] [a] [a]
         | Appending Int [a] [a]
         | Done [a]
  data HoodMelvilleQueue a = HM Int [a] (RotationState a) Int [a]

  exec (Reversing ok (x : f) f' (y : r) r') = Reversing (ok+1) f (x : f') r (y : r')
  exec (Reversing ok [] f' [y] r') = Appending ok f' (y : r')
  exec (Appending 0 f' r') = Done r'
  exec (Appending ok (x : f') r') = Appending (ok-1) f' (x : r')
  exec state = state

  invalidate (Reversing ok f f' r r') = Reversing (ok-1) f f' r r'
  invalidate (Appending 0 f' (x : r')) = Done r'
  invalidate (Appending ok f' r') = Appending (ok-1) f' r'
  invalidate state = state
```

```
exec2 lenf f state lenr r =
  case exec (exec state) of
    Done newf → HM lenf newf Idle lenr r
    newstate → HM lenf f newstate lenr r

check lenf f state lenr r =
  if lenr ≤ lenf then exec2 lenf f state lenr r
  else let newstate = Reversing 0 f [ ] r [ ]
       in exec2 (lenf+lenr) f newstate 0 [ ]

instance Queue HoodMelvilleQueue where
  empty = HM 0 [ ] Idle 0 [ ]
  isEmpty (HM lenf f state lenr r) = (lenf == 0)

  snoc (HM lenf f state lenr r) x = check lenf f state (lenr+1) (x : r)

  head (HM _ [] _ _ _) = error "empty queue"
  head (HM _ (x : f') _ _ _) = x

  tail (HM lenf [ ] state lenr r) = error "empty queue"
  tail (HM lenf (x : f') state lenr r) =
    check (lenf−1) f' (invalidate state) lenr r
```

```
module BootstrappedQueue (BootstrappedQueue) where
  import Prelude hiding (head,tail)
  import Queue

  data BootstrappedQueue a =
         E | Q Int [a] (BootstrappedQueue [a]) Int [a]

  checkQ,checkF :: Int → [a] → (BootstrappedQueue [a]) → Int → [a]
                           → BootstrappedQueue a
  checkQ lenfm f m lenr r =
    if lenr ≤ lenfm then checkF lenfm f m lenr r
    else checkF (lenfm+lenr) f (snoc m (reverse r)) 0 [ ]

  checkF lenfm [ ] E lenr f = E
  checkF lenfm [ ] m lenr r = Q lenfm (head m) (tail m) lenr r
  checkF lenfm f m lenr r = Q lenfm f m lenr r

  instance Queue BootstrappedQueue where
    empty = Q 0 [ ] E 0 [ ]
    isEmpty E = True
    isEmpty _ = False

    snoc E x = q 1 [x] E 0 [ ]
    snoc (Q lenfm f m lenr r) x = checkQ lenfm f m (lenr+1) (x : r)

    head E = error "empty queue"
    head (Q lenfm (x : f') m lenr r) = x

    tail E = error "empty queue"
    tail (Q lenfm (x : f') m lenr r) = checkQ (lenfm−1) f' m lenr r
```

```
module ImplicitQueue (ImplicitQueue) where
  import Prelude hiding (head,tail)
  import Queue

  data Digit a = ZERO | ONE a | TWO a a
  data ImplicitQueue a =
            SHALLOW (Digit a)
          | DEEP (Digit a) (ImplicitQueue (a, a)) (Digit a)

  instance Queue ImplicitQueue where
    empty = SHALLOW ZERO
    isEmpty (SHALLOW ZERO) = True
    isEmpty _ = False

    snoc (SHALLOW ZERO) y = SHALLOW (ONE y)
    snoc (SHALLOW (ONE x)) y = DEEP (TWO x y) empty ZERO
    snoc (DEEP f m ZERO) y = DEEP f m (ONE y)
    snoc (DEEP f m (ONE x)) y = DEEP f (snoc m (x,y)) ZERO

    head (SHALLOW ZERO) = error "empty queue"
    head (SHALLOW (ONE x)) = x
    head (DEEP (ONE x) m r) = x
    head (DEEP (TWO x y) m r) = x

    tail (SHALLOW ZERO) = error "empty queue"
    tail (SHALLOW (ONE x)) = empty
    tail (DEEP (TWO x y) m r) = DEEP (ONE y) m r
    tail (DEEP (ONE x) m r) =
        if isEmpty m then SHALLOW r else DEEP (TWO y z) (tail m) r
      where (y,z) = head m
```

Deques

```
module Deque (Deque(..)) where
  import Prelude hiding (head,tail,last,init)

  class Deque q where
    empty   :: q a
    isEmpty :: q a → Bool

    cons    :: a → q a → q a
    head    :: q a → a
    tail    :: q a → q a

    snoc    :: q a → a → q a
    last    :: q a → a
    init    :: q a → q a
```

```
module BankersDeque (BankersDeque) where
  import Prelude hiding (head,tail,last,init)
  import Deque
```

```
data BankersDeque a = BD Int [a] Int [a]

c = 3

check lenf f lenr r =
  if lenf > c*lenr + 1 then
    let i = (lenf+lenr) `div` 2
        j = lenf+lenr−i
        f' = take i f
        r' = r ++ reverse (drop i f)
    in BD i f' j r'
  else if lenr > c*lenf + 1 then
    let j = (lenf+lenr) `div` 2
        i = lenf+lenr−j
        r' = take j r
        f' = f ++ reverse (drop j r)
    in BD i f' j r'
  else BD lenf f lenr r

instance Deque BankersDeque where
  empty = BD 0 [] 0 []
  isEmpty (BD lenf f lenr r) = (lenf+lenr == 0)

  cons x (BD lenf f lenr r) = check (lenf+1) (x : f) lenr r

  head (BD lenf [] lenr r) = error "empty deque"
  head (BD lenf (x : f') lenr r) = x

  tail (BD lenf [] lenr r) = error "empty deque"
  tail (BD lenf (x : f') lenr r) = check (lenf−1) f' lenr r

  snoc (BD lenf f lenr r) x = check lenf f (lenr+1) (x : r)

  last (BD lenf f lenr []) = error "empty deque"
  last (BD lenf f lenr (x : r')) = x

  init (BD lenf f lenr []) = error "empty deque"
  init (BD lenf f lenr (x : r')) = check lenf f (lenr−1) r'
```

Catenable Lists

```
module CatenableList (CatenableList(..)) where
  import Prelude hiding (head,tail,(++))

  class CatenableList c where
    empty   :: c a
    isEmpty :: c a → Bool

    cons    :: a → c a → c a
    snoc    :: c a → a → c a
    (++)    :: c a → c a → c a

    head    :: c a → a
    tail    :: c a → c a
```

```
module CatList (CatList) where
  import Prelude hiding (head,tail,(++))
  import CatenableList
  import Queue (Queue)
  import qualified Queue

  data CatList q a = E | C a (q (CatList q a))

  link (C x q) s = C x (Queue.snoc q s)

  instance Queue q => CatenableList (CatList q) where
    empty = E
    isEmpty E = True
    isEmpty _ = False

    xs ++ E = xs
    E ++ xs = xs
    xs ++ ys = link xs ys

    cons x xs = C x Queue.empty ++ xs
    snoc xs x = xs ++ C x Queue.empty

    head E = error "empty list"
    head (C x q) = x

    tail E = error "empty list"
    tail (C x q) = if Queue.isEmpty q then E else linkAll q
      where linkAll q = if Queue.isEmpty q' then t else link t (linkAll q')
              where t = Queue.head q
                    q' = Queue.tail q
```

Catenable Deques

```
module CatenableDeque (CatenableDeque(..)) where
  import Prelude hiding (head,tail,last,init,(++))
  import Deque

  class Deque d => CatenableDeque d where
    (++) :: d a -> d a -> d a
```

```
module SimpleCatenableDeque (SimpleCatDeque) where
  import Prelude hiding (head,tail,last,init,(++))
  import CatenableDeque

  data SimpleCatDeque d a =
          SHALLOW (d a)
        | DEEP (d a) (SimpleCatDeque d (d a)) (d a)

  tooSmall d = isEmpty d || isEmpty (tail d)
```

```
dappendL d1 d2 = if isEmpty d1 then d2 else cons (head d1) d2
dappendR d1 d2 = if isEmpty d2 then d1 else snoc d1 (head d2)

instance Deque d => Deque (SimpleCatDeque d) where
  empty = SHALLOW empty
  isEmpty (SHALLOW d) = isEmpty d
  isEmpty _ = False

  cons x (SHALLOW d) = SHALLOW (cons x d)
  cons x (DEEP f m r) = DEEP (cons x f) m r

  head (SHALLOW d) = head d
  head (DEEP f m r) = head f

  tail (SHALLOW d) = SHALLOW (tail d)
  tail (DEEP f m r)
      | not (tooSmall f') = DEEP f' m r
      | isEmpty m = SHALLOW (dappendL f' r)
      | otherwise = DEEP (dappendL f' (head m)) (tail m) r
    where f' = tail f

  -- snoc, last, and init defined symmetrically...

instance Deque d => CatenableDeque (SimpleCatDeque d) where
  (SHALLOW d1) ++ (SHALLOW d2)
      | tooSmall d1 = SHALLOW (dappendL d1 d2)
      | tooSmall d2 = SHALLOW (dappendR d1 d2)
      | otherwise = DEEP d1 empty d2
  (SHALLOW d) ++ (DEEP f m r)
      | tooSmall d = DEEP (dappendL d f) m r
      | otherwise = DEEP d (cons f m) r
  (DEEP f m r) ++ (SHALLOW d)
      | tooSmall d = DEEP f m (dappendR r d)
      | otherwise = DEEP f (snoc m r) d
  (DEEP f1 m1 r1) ++ (DEEP f2 m2 r2) =
      DEEP f1 (snoc m1 r1 ++ cons f2 m2) r2
```

```
module ImplicitCatenableDeque (Sized(..), ImplicitCatDeque) where
  import Prelude hiding (head,tail,last,init,(++))
  import CatenableDeque

  class Sized d where
    size :: d a -> Int

  data ImplicitCatDeque d a =
           SHALLOW (d a)
         | DEEP (d a) (ImplicitCatDeque d (CmpdElem d a)) (d a)
                      (ImplicitCatDeque d (CmpdElem d a)) (d a)

  data CmpdElem d a =
           SIMPLE (d a)
         | CMPD (d a) (ImplicitCatDeque d (CmpdElem d a)) (d a)
```

```
share f r = (init f, m, tail r)
  where m = cons (last f) (cons (head r) empty)

dappendL d₁ d₂ =
  if isEmpty d₁ then d₂ else dappendL (init d₁) (cons (last d₁) d₂)
dappendR d₁ d₂ =
  if isEmpty d₂ then d₁ else dappendR (snoc d₁ (head d₂)) (tail d₂)

replaceHead x (SHALLOW d) = SHALLOW (cons x (tail d))
replaceHead x (DEEP f a m b r) = DEEP (cons x (tail f)) a m b r

instance (Deque d, Sized d) ⇒ Deque (ImplicitCatDeque d) where
  empty = SHALLOW empty
  isEmpty (SHALLOW d) = isEmpty d
  isEmpty _ = False

  cons x (SHALLOW d) = SHALLOW (cons x d)
  cons x (DEEP f a m b r) = DEEP (cons x f) a m b r

  head (SHALLOW d) = head d
  head (DEEP f a m b r) = head f

  tail (SHALLOW d) = SHALLOW (tail d)
  tail (DEEP f a m b r)
      | size f > 3 = DEEP (tail f) a m b r
      | not (isEmpty a) =
          case head a of
            SIMPLE d → DEEP f' (tail a) m b r
              where f' = dappendL (tail f) d
            CMPD f' c' r' → DEEP f'' a'' m b r
              where f'' = dappendL (tail f) f'
                    a'' = c' ++ replaceHead (SIMPLE r') a
      | not (isEmpty b) =
          case head b of
            SIMPLE d → DEEP f' empty d (tail b) r
              where f' = dappendL (tail f) m
            CMPD f' c' r' → DEEP f'' a'' r' (tail b) r
              where f'' = dappendL (tail f) m
                    a'' = cons (SIMPLE f') c'
      | otherwise = SHALLOW (dappendL (tail f) m) ++ SHALLOW r

  -- snoc, last, and init defined symmetrically...

instance (Deque d, Sized d) ⇒ CatenableDeque (ImplicitCatDeque d)
where
  (SHALLOW d₁) ++ (SHALLOW d₂)
      | size d₁ < 4 = SHALLOW (dappendL d₁ d₂)
      | size d₂ < 4 = SHALLOW (dappendR d₁ d₂)
      | otherwise = let (f, m, r) = share d₁ d₂ in DEEP f empty m empty r
  (SHALLOW d) ++ (DEEP f a m b r)
      | size d < 4 = DEEP (dappendL d f) a m b r
      | otherwise = DEEP d (cons (SIMPLE f) a) m b r
  (DEEP f a m b r) ++ (SHALLOW d)
      | size d < 4 = DEEP f a m b (dappendR r d)
      | otherwise = DEEP f a m (snoc b (SIMPLE r)) d
```

```
(DEEP f1 a1 m1 b1 r1) ++ (DEEP f2 a2 m2 b2 r2) = DEEP f1 a1' m b2' r2
  where (r1', m, f2') = share r1 f2
        a1' = snoc a1 (CMPD m1 b1 r1')
        b2' = cons (CMPD f2' a2 m2) b2
```

Random-Access Lists

```
module RandomAccessList (RandomAccessList(..)) where
  import Prelude hiding (head,tail,lookup)

  class RandomAccessList r where
    empty   :: r a
    isEmpty :: r a → Bool

    cons    :: a → r a → r a
    head    :: r a → a
    tail    :: r a → r a

    lookup  :: Int → r a → a
    update  :: Int → a → r a → r a
```

```
module BinaryRandomAccessList (BinaryList) where
  import Prelude hiding (head,tail,lookup)
  import RandomAccessList

  data Tree a = LEAF a | NODE Int (Tree a) (Tree a)
  data Digit a = ZERO | ONE (Tree a)
  newtype BinaryList a = BL [Digit a]

  size (LEAF x) = 1
  size (NODE w t1 t2) = w
  link t1 t2 = NODE (size t1 + size t2) t1 t2

  consTree t [] = [ONE t]
  consTree t (ZERO : ts) = ONE t : ts
  consTree t1 (ONE t2 : ts) = ZERO : consTree (link t1 t2) ts

  unconsTree [] = error "empty list"
  unconsTree [ONE t] = (t, [])
  unconsTree (ONE t : ts) = (t, ZERO : ts)
  unconsTree (ZERO : ts) = (t1, ONE t2 : ts')
    where (NODE _ t1 t2, ts') = unconsTree ts

  instance RandomAccessList BinaryList where
    empty = BL []
    isEmpty (BL ts) = null ts

    cons x (BL ts) = BL (consTree (LEAF x) ts)
    head (BL ts) = let (LEAF x, _) = unconsTree ts in x
    tail (BL ts) = let (_, ts') = unconsTree ts in BL ts'

    lookup i (BL ts) = look i ts
```

```
    where
      look i [] = error "bad subscript"
      look i (Zero : ts) = look i ts
      look i (One t : ts) =
        if i < size t then lookTree i t else look (i - size t) ts

      lookTree 0 (Leaf x) = x
      lookTree i (Leaf x) = error "bad subscript"
      lookTree i (Node w t1 t2) =
        if i < w `div` 2 then lookTree i t1 else lookTree (i - w `div` 2) t2

  update i y (BL ts) = BL (upd i ts)
    where
      upd i [] = error "bad subscript"
      upd i (Zero : ts) = Zero : upd i ts
      upd i (One t : ts) =
        if i < size t then One (updTree i t) : ts
        else One t : upd (i - size t) ts

      updTree 0 (Leaf x) = Leaf y
      updTree i (Leaf x) = error "bad subscript"
      updTree i (Node w t1 t2) =
        if i < w `div` 2 then Node w (updTree i t1) t2
        else Node w t1 (updTree (i - w `div` 2) t2)
```

```
module SkewBinaryRandomAccessList (SkewList) where
  import Prelude hiding (head,tail,lookup)
  import RandomAccessList

  data Tree a = Leaf a | Node a (Tree a) (Tree a)
  newtype SkewList a = SL [(Int, Tree a)]

  instance RandomAccessList SkewList where
    empty = SL []
    isEmpty (SL ts) = null ts

    cons x (SL ((w1,t1) : (w2,t2) : ts))
      | w1 == w2 = SL ((1+w1+w2, Node x t1 t2) : ts)
    cons x (SL ts) = SL ((1,Leaf x) : ts)

    head (SL []) = error "empty list"
    head (SL ((1, Leaf x) : ts)) = x
    head (SL ((w, Node x t1 t2) : ts)) = x

    tail (SL []) = error "empty list"
    tail (SL ((1, Leaf x) : ts)) = SL ts
    tail (SL ((w, Node x t1 t2) : ts)) = SL ((w `div` 2, t1) : (w `div` 2, t2) : ts)

    lookup i (SL ts) = look i ts
      where
        look i [] = error "bad subscript"
        look i ((w,t) : ts) =
          if i < w then lookTree w i t else look (i-w) ts
```

```
    lookTree 1 0 (LEAF x) = x
    lookTree 1 i (LEAF x) = error "bad subscript"
    lookTree w 0 (NODE x t1 t2) = x
    lookTree w i (NODE x t1 t2) =
        if i ≤ w' then lookTree w' (i−1) t1 else lookTree w' (i−1−w') t2
      where w' = w 'div' 2

  update i y (SL ts) = SL (upd i ts)
    where
    upd i [] = error "bad subscript"
    upd i ((w,t) : ts) =
      if i < w then (w,updTree w i t) : ts else (w,t) : upd (i−w) ts

    updTree 1 0 (LEAF x) = LEAF y
    updTree 1 i (LEAF x) = error "bad subscript"
    updTree w 0 (NODE x t1 t2) = NODE y t1 t2
    updTree w i (NODE x t1 t2) =
        if i ≤ w' then NODE x (updTree w' (i−1) t1) t2
        else NODE x t1 (updTree w' (i−1−w') t2)
      where w' = w 'div' 2
```

```
module AltBinaryRandomAccessList (BinaryList) where
  import Prelude hiding (head,tail,lookup)
  import RandomAccessList

  data BinaryList a =
    Nil | ZERO (BinaryList (a,a)) | ONE a (BinaryList (a,a))

  uncons :: BinaryList a → (a, BinaryList a)
  uncons Nil = error "empty list"
  uncons (ONE x Nil) = (x, Nil)
  uncons (ONE x ps) = (x, ZERO ps)
  uncons (ZERO ps) = let ((x,y), ps') = uncons ps in (x, ONE y ps')

  fupdate :: (a → a) → Int → BinaryList a → BinaryList a
  fupdate f i Nil = error "bad subscript"
  fupdate f 0 (ONE x ps) = ONE (f x) ps
  fupdate f i (ONE x ps) = cons x (fupdate f (i−1) (ZERO ps))
  fupdate f i (ZERO ps) = ZERO (fupdate f' (i 'div' 2) ps)
    where f' (x,y) = if i 'mod' 2 == 0 then (f x, y) else (x, f y)

  instance RandomAccessList BinaryList where
    empty = Nil
    isEmpty Nil = True
    isEmpty _ = False

    cons x Nil = ONE x Nil
    cons x (ZERO ps) = ONE x ps
    cons x (ONE y ps) = ZERO (cons (x,y) ps)

    head xs = fst (uncons xs)
    tail xs = snd (uncons xs)
```

```
lookup i Nil = error "bad subscript"
lookup 0 (ONE x ps) = x
lookup i (ONE x ps) = lookup (i-1) (ZERO ps)
lookup i (ZERO ps) = if i `mod` 2 == 0 then x else y
  where (x,y) = lookup (i `div` 2) ps

update i y xs = fupdate (λx → y) i xs
```

Heaps

```
module Heap (Heap(..)) where
  class Heap h where
    empty     :: Ord a ⇒ h a
    isEmpty   :: Ord a ⇒ h a → Bool

    insert    :: Ord a ⇒ a → h a → h a
    merge     :: Ord a ⇒ h a → h a → h a

    findMin   :: Ord a ⇒ h a → a
    deleteMin :: Ord a ⇒ h a → h a
```

```
module LeftistHeap (LeftistHeap) where
  import Heap

  data LeftistHeap a = E | T Int a (LeftistHeap a) (LeftistHeap a)

  rank E = 0
  rank (T r _ _ _) = r

  makeT x a b = if rank a ≥ rank b then T (rank b + 1) x a b
                else T (rank a + 1) x b a

  instance Heap LeftistHeap where
    empty = E
    isEmpty E = True
    isEmpty _ = False

    insert x h = merge (T 1 x E E) h

    merge h E = h
    merge E h = h
    merge h1@(T _ x a1 b1) h2@(T _ y a2 b2) =
      if x ≤ y then makeT x a1 (merge b1 h2)
      else makeT y a2 (merge h1 b2)

    findMin E = error "empty heap"
    findMin (T _ x a b) = x

    deleteMin E = error "empty heap"
    deleteMin (T _ x a b) = merge a b
```

```
module BinomialHeap (BinomialHeap) where
  import Heap

  data Tree a = Node Int a [Tree a]
  newtype BinomialHeap a = BH [Tree a]

  rank (Node r x c) = r
  root (Node r x c) = x

  link t1@(Node r x1 c1) t2@(Node _ x2 c2) =
    if x1 <= x2 then Node (r+1) x1 (t2 : c1) else Node (r+1) x2 (t1 : c2)

  insTree t [] = [t]
  insTree t ts@(t' : ts') =
    if rank t < rank t' then t : ts else insTree (link t t') ts'

  mrg ts1 [] = ts1
  mrg [] ts2 = ts2
  mrg ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1 : mrg ts1' ts2
    | rank t2 < rank t1 = t2 : mrg ts1 ts2'
    | otherwise = insTree (link t1 t2) (mrg ts1' ts2')

  removeMinTree [] = error "empty heap"
  removeMinTree [t] = (t, [])
  removeMinTree (t : ts) = if root t < root t' then (t, ts) else (t', t : ts')
    where (t', ts') = removeMinTree ts

  instance Heap BinomialHeap where
    empty = BH []
    isEmpty (BH ts) = null ts

    insert x (BH ts) = BH (insTree (Node 0 x []) ts)
    merge (BH ts1) (BH ts2) = BH (mrg ts1 ts2)

    findMin (BH ts) = root t
      where (t, _) = removeMinTree ts

    deleteMin (BH ts) = BH (mrg (reverse ts1) ts2)
      where (Node _ x ts1, ts2) = removeMinTree ts
```

```
module SplayHeap (SplayHeap) where
  import Heap

  data SplayHeap a = E | T (SplayHeap a) a (SplayHeap a)

  partition pivot E = (E, E)
  partition pivot t@(T a x b) =
    if x <= pivot then
      case b of
        E -> (t, E)
        T b1 y b2 ->
          if y <= pivot then
            let (small, big) = partition pivot b2
            in (T (T a x b) y small, big)
```

```
        else
          let (small, big) = partition pivot b₁
          in (T a x small, T big y b₂)
  else
    case a of
      E → (E, t)
      T a₁ y a₂ →
        if y ≤ pivot then
          let (small, big) = partition pivot a₂
          in (T a₁ y small, T big x b)
        else
          let (small, big) = partition pivot a₁
          in (small, T big y (T a₂ x b))

instance Heap SplayHeap where
  empty = E
  isEmpty E = True
  isEmpty _ = False

  insert x t = T a x b
    where (a, b) = partition x t

  merge E t = t
  merge (T a x b) t = T (merge ta a) x (merge tb b)
    where (ta, tb) = partition x t

  findMin E = error "empty heap"
  findMin (T E x b) = x
  findMin (T a x b) = findMin a

  deleteMin E = error "empty heap"
  deleteMin (T E x b) = b
  deleteMin (T (T E x b) y c) = T b y c
  deleteMin (T (T a x b) y c) = T (deleteMin a) x (T b y c)
```

```
module PairingHeap (PairingHeap) where
  import Heap

  data PairingHeap a = E | T a [PairingHeap a]

  mergePairs [ ] = E
  mergePairs [h] = h
  mergePairs (h₁ : h₂ : hs) = merge (merge h₁ h₂) (mergePairs hs)

  instance Heap PairingHeap where
    empty = E
    isEmpty E = True
    isEmpty _ = False

    insert x h = merge (T x [ ]) h

    merge h E = h
    merge E h = h
    merge h₁@(T x hs₁) h₂@(T y hs₂) =
      if x < y then T x (h₂ : hs₁) else T y (h₁ : hs₂)
```

```
    findMin E = error "empty heap"
    findMin (T x hs) = x

    deleteMin E = error "empty heap"
    deleteMin (T x hs) = mergePairs hs
```

```
module LazyPairingHeap (PairingHeap) where
  import Heap

  data PairingHeap a = E | T a (PairingHeap a) (PairingHeap a)

  link (T x E m) a = T x a m
  link (T x b m) a = T x E (merge (merge a b) m)

  instance Heap PairingHeap where
    empty = E
    isEmpty E = True
    isEmpty _ = False

    insert x a = merge (T x E E) a

    merge a E = a
    merge E b = b
    merge a@(T x _ _) b@(T y _ _) = if x ≤ y then link a b else link b a

    findMin E = error "empty heap"
    findMin (T x a m) = x

    deleteMin E = error "empty heap"
    deleteMin (T x a m) = merge a m
```

```
module SkewBinomialHeap (SkewBinomialHeap) where
  import Heap

  data Tree a = Node Int a [a] [Tree a]

  newtype SkewBinomialHeap a = SBH [Tree a]

  rank (Node r x xs c) = r
  root (Node r x xs c) = x

  link t1@(Node r x1 xs1 c1) t2@(Node _ x2 xs2 c2) =
    if x1 ≤ x2 then Node (r+1) x1 xs1 (t2 : c1)
    else Node (r+1) x2 xs2 (t1 : c2)

  skewLink x t1 t2 =
    let Node r y ys c = link t1 t2
    in if x ≤ y then Node r x (y : ys) c else Node r y (x : ys) c

  insTree t [] = [t]
  insTree t ts@(t' : ts') =
    if rank t < rank t' then t : ts else insTree (link t t') ts'
```

```
mrg ts₁ [ ] = ts₁
mrg [ ] ts₂ = ts₂
mrg ts₁@(t₁:ts₁') ts₂@(t₂:ts₂')
  | rank t₁ < rank t₂ = t₁ : mrg ts₁' ts₂
  | rank t₂ < rank t₁ = t₂ : mrg ts₁ ts₂'
  | otherwise = insTree (link t₁ t₂) (mrg ts₁' ts₂')

normalize [ ] = [ ]
normalize (t : ts) = insTree t ts

removeMinTree [ ] = error "empty heap"
removeMinTree [t] = (t, [ ])
removeMinTree (t : ts) = if root t < root t' then (t, ts) else (t', t : ts')
  where (t', ts') = removeMinTree ts

instance Heap SkewBinomialHeap where
  empty = SBH [ ]
  isEmpty (SBH ts) = null ts

  insert x (SBH (t₁ : t₂ : ts))
      | rank t₁ == rank t₂ = SBH (skewLink x t₁ t₂ : ts)
  insert x (SBH ts) = SBH (NODE 0 x [ ] [ ] : ts)

  merge (SBH ts₁) (SBH ts₂) = SBH (mrg (normalize ts₁) (normalize ts₂))

  findMin (SBH ts) = root t
    where (t, _) = removeMinTree ts

  deleteMin (SBH ts) = foldr insert (SBH ts') xs
    where (NODE _ x xs ts₁, ts₂) = removeMinTree ts
          ts' = mrg (reverse ts₁) (normalize ts₂)
```

```
module BootstrapHeap (BootstrapHeap) where
  import Heap

  data BootstrapHeap h a = E | H a (h (BootstrapHeap h a))

  instance Eq a ⇒ Eq (BootstrapHeap h a) where
    (H x _) == (H y _) = (x == y)
  instance Ord a ⇒ Ord (BootstrapHeap h a) where
    (H x _) ≤ (H y _) = (x ≤ y)

  instance Heap h ⇒ Heap (BootstrapHeap h) where
    empty = E
    isEmpty E = True
    isEmpty _ = False

    insert x h = merge (H x empty) h

    merge E h = h
    merge h E = h
    merge h₁@(H x p₁) h₂@(H y p₂) =
      if x ≤ y then H x (insert h₂ p₁) else H y (insert h₁ p₂)

    findMin E = error "empty heap"
    findMin (H x p) = x
```

```
deleteMin E = error "empty heap"
deleteMin (H x p) =
  if isEmpty p then E
  else let H y p1 = findMin p
           p2 = deleteMin p
       in H y (merge p1 p2)
```

Sortable Collections

```
module Sortable (Sortable(..)) where
 class Sortable s where
   empty :: Ord a ⇒ s a
   add   :: Ord a ⇒ a → s a → s a
   sort  :: Ord a ⇒ s a → [a]
```

```
module BottomUpMergeSort (MergeSort) where
 import Sortable

 data MergeSort a = MS Int [[a]]

 mrg [] ys = ys
 mrg xs [] = xs
 mrg xs@(x : xs') ys@(y : ys') =
   if x ≤ y then x : mrg xs' ys else y : mrg xs ys'

 instance Sortable MergeSort where
   empty = MS 0 []

   add x (MS size segs) = MS (size+1) (addSeg [x] segs size)
     where addSeg seg segs size =
              if size `mod` 2 == 0 then seg : segs
              else addSeg (mrg seg (head segs)) (tail segs) (size `div` 2)

   sort (MS size segs) = foldl mrg [] segs
```

Sets

```
module Set (Set(..)) where
 -- assumes multi-parameter type classes!

 class Set s a where
   empty  :: s a
   insert :: a → s a → s a
   member :: a → s a → Bool
```

```
module UnbalancedSet (UnbalancedSet) where
  import Set

  data UnbalancedSet a = E | T (UnbalancedSet a) a (UnbalancedSet a)

  instance Ord a ⇒ Set UnbalancedSet a where
    empty = E

    member x E = False
    member x (T a y b) =
      if x < y then member x a
      else if x > y then member x b
      else True

    insert x E = T E x E
    insert x s@(T a y b) =
      if x < y then T (insert x a) y b
      else if x > y then T a y (insert x b)
      else s
```

```
module RedBlackSet (RedBlackSet) where
  import Set

  data Color = R | B
  data RedBlackSet a = E | T Color (RedBlackSet a) a (RedBlackSet a)

  balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
  balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
  balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
  balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
  balance color a x b = T color a x b

  instance Ord a ⇒ Set RedBlackSet a where
    empty = E

    member x E = False
    member x (T _ a y b) =
      if x < y then member x a
      else if x > y then member x b
      else True

    insert x s = T B a y b
      where ins E = T R E x E
            ins s@(T color a y b) =
              if x < y then balance color (ins a) y b
              else if x > y then balance color a y (ins b)
              else s
            T _ a y b = ins s        -- guaranteed to be non-empty
```

Finite Maps

```
module FiniteMap (FiniteMap(..)) where
  -- assumes multi-parameter type classes!

  class FiniteMap m k where
    empty  :: m k a
    bind   :: k → a → m k a → m k a
    lookup :: k → m k a → Maybe a
```

```
module Trie (Trie) where
  import FiniteMap

  data Trie mk ks a = TRIE (Maybe a) (mk (Trie mk ks a))

  instance FiniteMap m k ⇒ FiniteMap (Trie (m k)) [k] where
    empty = TRIE NOTHING empty

    lookup [] (TRIE b m) = b
    lookup (k : ks) (TRIE b m) = lookup k m >>= λm' → lookup ks m'

    bind [] x (TRIE b m) = TRIE (JUST x) m
    bind (k : ks) x (TRIE b m) =
      let t = case lookup k m of
                JUST t → t
                NOTHING → empty
          t' = bind ks x t
      in TRIE b (bind k t' m)
```

```
module TrieOfTrees (Tree(..), Trie) where
  import FiniteMap

  data Tree a = E | T a (Tree a) (Tree a)
  data Trie mk ks a = TRIE (Maybe a) (mk (Trie mk ks (Trie mk ks a)))

  instance FiniteMap m k ⇒ FiniteMap (Trie (m k)) (Tree k) where
    empty = TRIE NOTHING empty

    lookup E (TRIE v m) = v
    lookup (T k a b) (TRIE v m) =
      lookup k m >>= λm' →
      lookup a m' >>= λm'' →
      lookup b m''

    bind E x (TRIE v m) = TRIE (JUST x) m
    bind (T k a b) x (TRIE v m) =
      let tt = case lookup k m of
                 JUST tt → tt
                 NOTHING → empty
```

```
    t = case lookup a tt of
            JUST t → t
            NOTHING → empty
    t' = bind b x t
    tt' = bind a t' tt
in TRIE v (bind k tt' m)
```