```
module M19functionaldata where
```

Purely Functional Data Structures

This lecture is based on the book

Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998.

A purely functional data structure is built and manipulated without the use of side effects. The main consequence of eschewing side effects is the persistence of functional data structures. Persistence means that after a modification, the previous version of the structure is still available. Semantically, persistence is a nice property, but it can lead to memory leaks if code unadvertently holds on to previous versions that are no longer used. In (typical) single-threaded use, the new modified version of a structure shares a significant part of its representation with the previous version. Garbage collection guarantees that inaccessible parts of the previous version are reclaimed for other uses.

Binary Search Tree

The standard binary search tree serves as a very simple example for a representation of a set. A map can be implemented very similarly. The representation needs to provide efficient implementations for the member and insert operations.

```
data Tree a = E | T (Tree a) a (Tree a)
            deriving (Show)
```

```
empty = E
```

```
instance Member Tree where
  member x E = False
  member x (T l y r) =
    case compare x y of
      EQ -> True
      LT -> member x l
      GT -> member x r
```

```
instance Insert Tree where
  insert x E = T E x E
  insert x t@(T l y r) =
    case compare x y of
      EQ -> t
      LT -> T (insert x l) y r
      GT -> T l y (insert x r)
```

Persistent data structure. Easy correctness proof. Known deficiency: Not balanced.

Leftist Heap

A heap or a priority queue is a special representation of an ordered set where access to a "minimal" element is most efficient. A typical representation of a heap is a tree with the minimum element at the root and with every node fulfilling the "heap condition" that the elements at its children are greater than (or equal to) the element at the node. A consequence of the heap condition is that the elements along any path in the tree are sorted ascendingly.

A *leftist heap* is a heap-ordered binary tree that satisfies the *leftist property*: The rank of any left child is at least as large as the rank of its right sibling. Here, the rank of a node is the length of its right spine, i.e., the rightmost path to an empty node. Consequence: in a leftist heap, the right spine is always the shortest path to an empty node.

Proposition:

The right spine of a leftist heap of size n contains at most $\lfloor \log_2(n+1) \rfloor$ elements.

Our representation of leftist heaps directly includes rank information:

```haskell
data Heap a = EH | TH Int a (Heap a) (Heap a)
            deriving (Show)
```

Key observation: Two leftist heaps can be merged by merging their right spines as with sorted lists and then swapping children along this path to restore the leftist property.

```haskell
instance Merge Heap where
  merge h1 EH = h1
  merge EH h2 = h2
  merge h1@(TH _ x1 l1 r1) h2@(TH _ x2 l2 r2) =
    if x1 <= x2 then
      makeT x1 l1 (merge r1 h2)
    else
      makeT x2 l2 (merge h1 r2)

instance Rank Heap where
  rank EH = 0
  rank (TH r _ _ _) = r

makeT x l r =
  let rankl = rank l
      rankr = rank r
  in  if rankr <= rankl then
        TH (rankr + 1) x l r
      else
        TH (rankl + 1) x r l
```

Merge runs in $O(\log n)$ time because the length of each right spine is at most logarithmic in the number of elements.

Given the efficient merge function, the remaining functions are trivial.

```
instance Insert Heap where
  insert x h = merge h (TH 1 x EH EH)
instance FindMin Heap where
  findMin (TH _ x _ _) = x
instance DeleteMin Heap where
  deleteMin (TH _ x l r) = merge l r
```

What's their complexity?

Binomial Heaps

Binomial heaps are another common implementation of heaps. They are composed of primitive objects, the "binomial trees". They are defined inductively:

- A binomial tree of rank 0 is a singleton node.
- A binomial tree of rank $(r + 1)$ is formed by linking two binomial trees of rank r, which makes one tree the leftmost child of the other.

A binomial tree of rank r contains exactly 2^r nodes. Here is an alternative definition: A binomial tree of rank r is a node with r children $t_1, \ldots, t_r$ where each t_i is a binomial tree of rank $(r - i)$. The first field of a tree node contains its rank.

```
data BinTree a = Node Int a [BinTree a]
```

```
instance Rank BinTree where
  rank (Node r _ _) = r
```

```
instance Root BinTree where
  root (Node _ x _) = x
```

```
subtrees (Node _ _ ts) = ts
```

Linking one tree to the other maintains heap order by attaching trees with larger roots below trees with smaller roots. This code assumes that only trees of equal rank are linked.

```
link :: (Ord a) => BinTree a -> BinTree a -> BinTree a
link t1@(Node r1 x1 c1) t2@(Node r2 x2 c2) | r1 == r2 =
  if x1 <= x2 then
     Node (r1+1) x1 (t2 : c1)
  else
     Node (r1+1) x2 (t1 : c2)
```

A "binomial heap" is a collection of heap-ordered binomial trees in which no two trees have the same rank. This collection is represented as a list of trees in rank-increasing order.

```
newtype BinHeap a = BinHeap [BinTree a]
```

Because each binomial tree has 2^r Elements and because no two trees in a binomial heap have the same rank, the trees in a binomial heap of size n correspond to the ones in the binary representation of n. Hence, a heap of size n contains at most $\lfloor \log_2(n+1) \rfloor$ trees.

The insert and merge operations on binomial heaps are defined in analogy to binary addition with the link operation corresponding to processing the carry bit.

```
insTree :: (Ord a) => BinTree a -> [BinTree a] -> [BinTree a]
insTree t [] = [t]
insTree t h@(t1:ts) =
  if rank t < rank t1 then
    t : h
  else
    insTree (link t t1) ts

instance Insert BinHeap where
  insert x (BinHeap ts) = BinHeap (insTree (Node 0 x []) ts)
```

The worst case for the insert operation occurs when the heap contains $n = 2^k - 1$ elements. In this case, $O(k) = O(\log n)$ link operations are necessary.

The merge operation is also straightforward. It traverses both lists and links the trees of equal rank.

```
instance Merge BinHeap where
  merge (BinHeap ts1) (BinHeap ts2) = BinHeap $ mergeTrees ts1 ts2
mergeTrees [] ts = ts
mergeTrees ts [] = ts
mergeTrees h1@(t1:ts1) h2@ (t2:ts2) =
  case compare (rank t1) (rank t2) of
    LT -> t1 : mergeTrees ts1 h2
    GT -> t2 : mergeTrees h1 ts2
    EQ -> insTree (link t1 t2) (mergeTrees ts1 ts2)
```

Complexity?

For the remaining operations, we need an auxiliary function that extracts the minimum tree from a heap. It clearly takes $O(\log n)$ time

```
removeMinTree :: (Ord a) => [BinTree a] -> (BinTree a, [BinTree a])
removeMinTree [t] =
  (t, [])
removeMinTree (t:ts) =
  if root t <= root t1 then
    (t, ts)
  else
    (t1, t : ts1)
  where (t1, ts1) = removeMinTree ts
```

Finding the minimum element returns just the root of the minimum tree.

```
instance FindMin BinHeap where
  findMin (BinHeap ts) =
    root (fst (removeMinTree ts))
```

Deleting the minimum element requires to merge the list of subtrees with the remaining binomial heap. However, the list of subtrees is arranged in decreasing rank order, so it needs to be reversed before merging.

```
instance DeleteMin BinHeap where
  deleteMin (BinHeap ts) =
    BinHeap (mergeTrees ts1 (reverse (subtrees t1)))
    where (t1, ts1) = removeMinTree ts
```

Red-Black Trees

Binary search trees may become unbalanced, which degrades their potential $\log n$ performance for find and insert to linear time. The red-black tree is an instance of a balanced binary search tree structure that guarantees worst case $O(\log n)$ time for find and insert. In such a tree, every node is colored either red or black where empty nodes are considered black.

```
data RBTree a = RBE | RBT Color (RBTree a) a (RBTree a)
data Color = R | B
```

A red-black tree must satisfy the following invariants.

1. No red node has a red child.
2. Every path from the root to a empty node contains the same number of black nodes.

These invariants guarantee that the longest path in a red-black tree (with alternating red and black nodes) is never more than twice as long as the shortest path (with black nodes, only).

Proposition:

The maximum depth of a node in a red-black tree of size n is less than or equal to $2 * \lfloor \log(n+1) \rfloor$.

The implementation of member ignores the colors.

```
instance Member RBTree where
  member x RBE = False
  member x (RBT _ l y r) =
    case compare x y of
      EQ -> True
      LT -> member x l
      GT -> member x r
```

```
instance Insert RBTree where
  insert x s = RBT B a y b
    where
    RBT _ a y b = ins s
    ins RBE = RBT R RBE x RBE
    ins t@(RBT c l y r) =
      case compare x y of
        EQ -> t
        LT -> balance c (ins l) y r
        GT -> balance c l y (ins r)
```

Coloring the new node red maintains *2* but violates *1* if the parent of the new node is red. During insertion, a single red-red violation is allowed, which is percolated along the search path to the root by the balance function.

```
balance B (RBT R (RBT R a x b) y c) z d = RBT R (RBT B a x b) y (RBT B c z d)
balance B (RBT R a x (RBT R b y c)) z d = RBT R (RBT B a x b) y (RBT B c z d)
balance B a x (RBT R (RBT R b y c) z d) = RBT R (RBT B a x b) y (RBT B c z d)
balance B a x (RBT R b y (RBT R c z d)) = RBT R (RBT B a x b) y (RBT B c z d)
balance c a x b = RBT c a x b
```

The balance function performs several unnecessary tests. When ins recurses on the left child, then there is no need to expect a red-red violation in the riht child. In fact, it is possible to only ever test the color of nodes on the search path.

With these optimizations, this implementation of binary search trees is very fast.

Stack

Queues

A common functional implementation of a queue consists of a pair of lists where one list contains the front elements of the queue and the other the rear elements in *reverse*.

```
data Queue a = Queue [a] [a]
             deriving (Show)
```

Moreover, we maintain the invariant that the front list is only empty if the rear list is also empty. Whenever the front list becomes empty, we reverse the rear list and put it in front.

```
checkf (Queue [] xs) = Queue (reverse xs) []
checkf q = q

instance QueueOps Queue where
  headq (Queue (x:_) _) = x
  tailq (Queue (_:xs) ys) = checkf (Queue xs ys)
  enq x (Queue xs ys) = checkf (Queue xs (x:ys))
  nullq (Queue xs _) = null xs
  emptyq = (Queue [] [])
```

Complexity: headq and nullq run in O(1) worst case. tailq and enq run in O(n) worst case, but in O(1) amortized time.

There is also a real-time version of the queue where each operation has O(1) worst case time complexity. It exploits lazy evaluation.

```
data RTQ a = RTQ [a] [a] [a]
```

Its representation consists of the front part of the queue, the reversed rear part, and a *schedule*. The schedule serves to incrementalize the execution of the reverse operation so that it can be evenly spread over the enq and tailq operations.

```
instance QueueOps RTQ where
  emptyq = RTQ [] [] []
  nullq (RTQ xs _ _) = null xs
  enq x (RTQ f r s) = exec (RTQ f (x:r) s)
  headq (RTQ (x:f) _ _) = x
  tailq (RTQ (x:f) r s) = exec (RTQ f r s)

exec (RTQ f r (_:s)) = RTQ f r s
exec (RTQ f r []) = RTQ f' [] f'
  where f' = rotate f r []

rotate [] (y:_) a = y:a
rotate (x:xs) (y:ys) a = x : rotate xs ys (y:a)
```

With this implementation, all operations take O(1) worst-case time.

enq x1 (RTQ [] [] []) -> exec (RTQ [] (x1:[]) []) -> RTQ f1 [] f1 where f1 = rotate [] (x1:[]) []

enq x2

exec (RTQ f1 (x2:[]) f1) where f1 = rotate [] (x1:[]) [] -> exec (RTQ f1 (x2:[]) f1) where { f1 = x1:f1'; f1' = [] } -> RTQ f1 (x2:[]) f1'

enq x3

exec (RTQ f1 (x3:x2:[]) f1') where { f1 = x1:f1'; f1' = [] } -> RTQ (f2 [] f2) where { f1 = x1:f1'; f1' = []; f2 = rotate f1 (x3:x2:[]) [] }

Auxiliary definitions: interface for set and heap manipulation

```
class Rank f where
  rank :: f a -> Int
```

```
rankBinTree :: BinTree a -> Int
rankBinTree (Node r _ _) = r

instance Rank [] where
  rank xs = length xs

class Root f where
  root :: f a -> a

class Member f where
  member :: (Ord a) => a -> f a -> Bool

class Insert f where
  insert :: (Ord a) => a -> f a -> f a

class Merge f where
  merge :: (Ord a) => f a -> f a -> f a

class FindMin f where
  findMin :: (Ord a) => f a -> a

class DeleteMin f where
  deleteMin :: (Ord a) => f a -> f a

class QueueOps q where
  nullq :: q a -> Bool
  emptyq :: q a
  enq :: a -> q a -> q a
  tailq :: q a -> q a
  headq :: q a -> a
```