Persistent Dynamic Sets

This preliminary handout contains some additional information on dynamic sets. It will later be merged with the information in Handout #18 to form a revised Handout #18'.

Persistence

A data type is said to be **persistent** (a.k.a. **immutable**) if its specification does not allow an existing instance of the data type to change over time. Operations exist to observe a given instance and create new instances, but there are no operations to change an existing instance.

Typically, implementations of persistent data types use only immutable data structures. However, the implementation of a persistent data type may use mutable operations (e.g., assignment to mutable variables) as long as all such operations are “under the hood” – the time-varying behavior of mutable operations cannot be observable outside the abstract barrier separating the specification and implementation.
Declaring Data Types in Haskell

Haskell’s `data` declaration allows the creation of new data types. For example:

```
-- Integer lists
data IntList = INil | ICons int IntList

ICons 3 (ICons 2 (ICons 7 Nil)) :: IntList

-- Lists of any type a:
data List a = Nil | Cons a (List a)

Cons 3 (Cons 2 (Cons 7 Nil)) :: List Int
Cons True (Cons False Nil) :: List Bool
Cons 'a' (Cons 'b' (Cons 'c' Nil)) :: List Char
Cons (Cons 'a' (Cons 'b' Nil))
    (Cons (Cons 'c Nil)
       Nil) :: List (List Char)

-- Built-in lists have a special notation:
3 : (2 : (7 : [])) :: [Int]
True : (False : []) :: [Bool]
'a' : ('b' : ('c' : [])) :: [Char]
['a', 'b', 'c'] :: [Char]
[['a', 'b'], ['c']] :: [[Char]]

-- The following is a standard data type for
-- distinguishing no result from some result
data Maybe a = Nothing | Just a

Nothing :: Maybe Int -- also has types Maybe Bool, Maybe Char
Just 3 :: Maybe Int
Just True :: Maybe Bool
Just 'a' :: Maybe Char
Just (Cons 'a' Nil) :: Maybe List Char
Haskell BST Implementation of Persistent Dynamic Sets: Part 1

```haskell
data BinTree a = Leaf | Node (BinTree a) a (BinTree a)

-- sample ops on BSTs
isLeaf Leaf = True
isLeaf _ = False

value Leaf = error "value of leaf"
value (Node l v r) = v

height Leaf = 0
height (Node l v r) = (max (height l) (height r)) + 1

-- exported functions

empty :: DynSet a
empty = Leaf

isEmpty :: DynSet a -> Bool
isEmpty = isLeaf

search :: Keyed a b => b -> DynSet a -> Maybe a
search key Leaf = Nothing
search key (Node l v r)
| key == keyOf(v) = Just(v)
| key < keyOf(v) = search key l
| key > keyOf(v) = search key r

insert :: Ord a => a -> DynSet a -> DynSet a
insert v Leaf = (Node Leaf v Leaf)
insert v (Node l x r)
| v == x = Node l v r
| v < x = Node (insert v l) x r
| v > x = Node l x (insert v r)

delete :: Ord a => a -> DynSet a -> DynSet a
delete v Leaf = Leaf
delete v (Node l x r)
| v < x = Node (delete v l) x r
| v > x = Node l x (delete v r)
| v == x = case (l,r) of
  (Leaf, _) -> r
  (_, Leaf) -> l
  _ -> let (Just mx) = maxElt l
        in Node (delete mx l) mx r
```
Haskell BST Implementation of Persistent Dynamic Sets: Part 2

toList :: DynSet a -> [a]
toList Leaf = []
toList (Node l v r) = (toList l) ++ [v] ++ (toList r)

minElt :: DynSet a -> Maybe a
minElt Leaf = Nothing
minElt (Node Leaf v _) = Just(v)
minElt (Node l _ _) = minElt l

maxElt :: DynSet a -> Maybe a
maxElt Leaf = Nothing
maxElt (Node _ v Leaf) = Just(v)
maxElt (Node _ _ r) = maxElt r

pred :: Ord a => a -> DynSet a -> Maybe a
pred v tr = predLoop Nothing tr
  where predLoop leftAncestor Leaf = Nothing
        predLoop leftAncestor (Node l x r)
          | v == x = if isLeaf l then leftAncestor else maxElt l
          | v < x = predLoop leftAncestor l
          | v > x = predLoop (Just x) r

succ :: Ord a => a -> DynSet a -> Maybe a
succ v tr = succLoop Nothing tr
  where succLoop rightAncestor Leaf = Nothing
        succLoop rightAncestor (Node l x r)
          | v == x = if isLeaf r then rightAncestor else minElt r
          | v < x = succLoop (Just x) l
          | v > x = succLoop rightAncestor r