Code (Tuesday Week 4)

Dictionary (data invariants)

```haskell
module Dictionary
  ( Word
  , Definition
  , Dict
  , emptyDict
  , insertWord
  , lookup
  ) where

import Prelude hiding (Word, lookup)
import Test.QuickCheck
import Test.QuickCheck.Modifiers
-- lookup :: [(a,b)] -> a -> Maybe b
type Word = String
type Definition = String

newtype Dict = D [DictEntry]
             deriving (Show, Eq)

emptyDict :: Dict
emptyDict = D []

insertWord :: Word -> Definition -> Dict -> Dict
insertWord w def (D defs) = D (insertEntry (Entry w def) defs)
  where
    insertEntry wd (x:xs) = case compare (word wd) (word x)
                              of GT -> x : (insertEntry wd xs)
                                 EQ -> wd : xs
                                 LT -> wd : x : xs
    insertEntry wd [] = [wd]

lookup :: Word -> Dict -> Maybe Definition
lookup w (D es) = search w es
  where
    search w [] = Nothing
    search w (e:es) = case compare w (word e) of
       LT -> Nothing
       EQ -> Just (defn e)
       GT -> search w es
```

```haskell
sorted :: (Ord a) => [a] -> Bool
sorted []  = True
sorted [x] = True
sorted (x:y:xs) = x <= y && sorted (y:xs)

wellformed :: Dict -> Bool
wellformed (D es) = sorted es

prop_insert_wf dict w d = wellformed dict ==>
                          wellformed (insertWord w d dict)

data DictEntry
  = Entry { word :: Word
          , defn :: Definition
          } deriving (Eq, Show)

instance Ord DictEntry where
  Entry w1 d1 <= Entry w2 d2 = w1 <= w2


instance Arbitrary DictEntry where
  arbitrary = Entry <$> arbitrary <*> arbitrary

instance Arbitrary Dict where
  arbitrary = do
    Ordered ds <- arbitrary
    pure (D ds)

prop_arbitrary_wf dict = wellformed dict
```

Queue (data refinement)

```haskell
import Test.QuickCheck


emptyQueueL = []
enqueueL a  = (++ [a])
frontL      = head
dequeueL    = tail
sizeL       = length


toAbstract :: Queue -> [Int]
toAbstract (Q f sf r sr) = f ++ reverse r

prop_empty_ref = toAbstract emptyQueue == emptyQueueL
```

```haskell
prop_enqueue_ref fq x = toAbstract (enqueue x fq)
                     == enqueueL x (toAbstract fq)

prop_size_ref fq = size fq == sizeL (toAbstract fq)

prop_front_ref fq = size fq > 0 ==> front fq == frontL (toAbstract fq)
prop_deq_ref fq = size fq > 0 ==>  toAbstract (dequeue fq)
                                == dequeueL (toAbstract fq)

prop_wf_empty = wellformed emptyQueue
prop_wf_enq x q = wellformed q ==> wellformed (enqueue x q)
prop_wf_deq x q = wellformed q && size q > 0 ==> wellformed (dequeue q)

data Queue = Q [Int] -- front of the queue
               Int   -- size of the front
               [Int] -- rear of the queue
               Int   -- size of the rear
             deriving (Show, Eq)

wellformed :: Queue -> Bool
wellformed (Q f sf r sr) = length f == sf && length r == sr
                        && sf >= sr

instance Arbitrary Queue where
  arbitrary = do
    NonNegative sf' <- arbitrary
    NonNegative sr <- arbitrary
    let sf = sf' + sr
    f <- vectorOf sf arbitrary
    r <- vectorOf sr arbitrary
    pure (Q f sf r sr)


inv3 :: Queue -> Queue
inv3 (Q f sf r sr)
   | sf < sr   = Q (f ++ reverse r) (sf + sr) [] 0
   | otherwise = Q f sf r sr

emptyQueue :: Queue
emptyQueue = Q [] 0 [] 0

enqueue :: Int -> Queue -> Queue
enqueue x (Q f sf r sr) = inv3 (Q f sf (x:r) (sr+1))

front :: Queue -> Int   -- partial
front (Q (x:f) sf r sr) = x

dequeue :: Queue -> Queue -- partial
```

```
dequeue (Q (x:f) sf r sr) = inv3 (Q f (sf -1) r sr)

size    :: Queue -> Int
size (Q f sf r sr) = sf + sr
```