```haskell
module Queue where

import Lib (CanBeEmpty (..), FromList(..), ToList(..))

data Queue a = Queue a (Queue a)
             | EmptyQueue
             deriving (Eq)

-- :k Show, Show :: * -> Constraint
-- {-# MINIMAL showsPrec | show #-}
instance Show a => Show (Queue a) where
  -- | Show the queue
  show EmptyQueue = "<-|"
  show (Queue a b) = "<-" ++ show a ++ (show b) -- first show shows a
charracter, seccond show shows a list like reccursion
  -- showList konvertiert eine Liste von Werten in eine String-Darstellung der
Queue
  --showList [] = "<-|"
  --showList xs = "<-" ++ concatMap (show . Queue) xs ++ "<-"



-- instance Show a => Show (Queue a) where
-- -- | Show the queue
-- show EmptyQueue = "EmptyQueue"
-- show (Queue a b) = "Queue " ++ show a ++ "(" ++ show b ++ ")" -- first show
shows a charracter, seccond show shows a list like reccursion


-- :k FromList, FromList :: (* -> *) -> Constraint
-- {-# MINIMAL fromList #-}
instance FromList Queue where
  fromList = foldr Queue EmptyQueue

-- :k Functor, Functor :: (* -> *) -> Constraint
-- {-# MINIMAL fmap #-}
instance Functor Queue where
  fmap _ EmptyQueue = EmptyQueue
  fmap f (Queue x xs) = Queue (f x) (fmap f xs)
  (<$) :: b -> Queue a -> Queue b
  (<$) x EmptyQueue = EmptyQueue
  (<$) x (Queue _ xs) = Queue x (x <$ xs) -- ersetzt alle Werte in der Queue
mit x

-- k: ToList, ToList :: (* -> *) -> Constraint
-- {-# MINIMAL toList #-}
instance ToList Queue where
  toList = foldr (:) []
```

```haskell
-- :k Foldable, Foldable :: (* -> *) -> Constraint
-- {-# MINIMAL foldMap | foldr #-}
instance Foldable Queue where
  foldr _ z EmptyQueue = z
  foldr f z (Queue x xs) = f x (foldr f z xs)
  foldl _ z EmptyQueue = z
  foldl f z (Queue x xs) = foldl f (f z x) xs
  -- foldMap ist eine Methode der Foldable-Typklasse in Haskell, die es
erlaubt,
  -- eine Funktion auf jedes Element einer Struktur anzuwenden und die
Ergebnisse
  -- unter Verwendung einer Monoid-Instanz zu kombinieren.
  foldMap _ EmptyQueue = mempty
  foldMap f (Queue x xs) = f x `mappend` foldMap f xs
  -- foldr1 :: (a -> a -> a) -> Queue a -> a
  foldr1 _ EmptyQueue = error "foldr1: empty structure"
  foldr1 f (Queue x xs) = foldr f x xs
  -- foldl1 :: (a -> a -> a) -> Queue a -> a
  foldl1 _ EmptyQueue = error "foldl1: empty structure"
  foldl1 f (Queue x xs) = foldl f x xs
  -- null :: Queue a -> Bool
  null EmptyQueue = True
  null _ = False
  -- length :: Queue a -> Int
  length EmptyQueue = 0
  length (Queue _ xs) = 1 + length xs
  -- elem :: Eq a => a -> Queue a -> Bool
  elem _ EmptyQueue = False
  elem e (Queue x xs) = (e == x) || elem e xs
  -- maximum :: Ord a => Queue a -> a
  maximum EmptyQueue = error "maximum: empty structure"
  maximum (Queue x xs) = foldl max x xs
  -- minimum :: Ord a => Queue a -> a
  minimum EmptyQueue = error "minimum: empty structure"
  minimum (Queue x xs) = foldl min x xs
  -- sum :: Num a => Queue a -> a
  sum EmptyQueue = 0
  sum (Queue x xs) = foldl (+) x xs
  -- product :: Num a => Queue a -> a
  product EmptyQueue = 1
  product (Queue x xs) = foldl (*) x xs




-- :k CanBeEmpty, CanBeEmpty :: * -> Constraint
-- {-# MINIMAL isEmpty #-}
instance CanBeEmpty (Queue a) where
  isEmpty EmptyQueue = True
  isEmpty _ = False
```

```haskell
-- :k Semigroup , Semigroup :: (* -> *) -> Constraint7
-- {-# MINIMAL (<>) #-}
instance Semigroup (Queue a) where
  EmptyQueue <> EmptyQueue = EmptyQueue
  EmptyQueue <> x = x
  x <> EmptyQueue = x
  Queue a as <> Queue x xs = (enqueue x (Queue a as)) <> xs


-- :k Monoid, Monoid :: * -> Constraint
-- {-# MINIMAL mempty #-}
-- instance Semigroup a => Monoid (Queue a) where wenn a vom Typ semigroup
sein Muss, dann so machen sonst wie unten
instance Monoid (Queue a) where
  mempty = EmptyQueue
  -- Defining mappend using (<>)
  mappend = (<>)
  -- Defining mconcat using foldr and mappend
  mconcat = foldr mappend mempty



-- :k Applicative, Applicative :: (* -> *) -> Constraint
-- {-# MINIMAL pure, ((<*>) | liftA2) #-}
instance Applicative Queue where
  pure x = Queue x EmptyQueue
  EmptyQueue <*> _ = EmptyQueue
  _ <*> EmptyQueue = EmptyQueue
  (<*>)  (Queue f fs) (Queue y ys) = (fmap f (Queue y ys)) <> (fs <*> (Queue y
ys))
  -- (*>) :: Queue a -> Queue b -> Queue b
  (*>) EmptyQueue x = x
  (*>) _ EmptyQueue = EmptyQueue
  (*>) (Queue _ fs) ys@(Queue _ ys') = ys
  -- (<*) :: Queue a -> Queue b -> Queue a
  (<*) EmptyQueue _ = EmptyQueue
  (<*) q1 EmptyQueue = EmptyQueue
  (<*) (Queue x xs) q2 = Queue x xs
```

```haskell
-- (>>=) :: m a -> (a -> m b) -> m b
-- :k Monad, Monad :: (* -> *) -> Constraint
-- {-# MINIMAL (>>=) #-}
instance Monad Queue where
  -- Bind (>>=)
  EmptyQueue >>= _ = EmptyQueue
  (Queue a as) >>= f = (f a) <> (as >>= f)
  -- Then (>>)
  EmptyQueue >> _ = EmptyQueue
  x >> EmptyQueue = EmptyQueue
  x >> (Queue b bs) = bs
  -- return ist äquivalent zu pure
  return = pure



enqueue :: a -> Queue a -> Queue a
enqueue x EmptyQueue = Queue x EmptyQueue
enqueue b (Queue x xs) = Queue x (enqueue b xs)

dequeue :: Queue a -> Queue a
dequeue EmptyQueue = EmptyQueue
dequeue (Queue x xs) = xs

front :: Queue a -> a
front EmptyQueue = error "Empty queue so no front"
front (Queue a xs) = a

-- zum Texten erstellte Queues
exampleQueue :: Queue Int
exampleQueue = enqueue 3 (enqueue 2 (enqueue 1 EmptyQueue))
qf = Queue (+1) (Queue (*2) EmptyQueue) :: Queue (Int -> Int)
qx = Queue 1 (Queue 2 (Queue 3 EmptyQueue)) :: Queue Int
```