```haskell
module Tree where

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

data Tree a = Node a [Tree a]
            | EmptyTree
            deriving (Eq)

-- :k Show, Show :: * -> Constraint
-- {-# MINIMAL showsPrec | show #-}
instance Show a => Show (Tree a) where
    -- | Show the tree
    show EmptyTree = "Empty"
    show (Node a []) = show a
    show (Node a ts) = show a ++ " -> [" ++ show ts ++ "]"

-- :k FromList, FromList :: (* -> *) -> Constraint
-- {-# MINIMAL fromList #-}
instance FromList Tree where
    fromList [] = EmptyTree
    fromList (x:xs) = Node x (map (\y -> Node y []) xs)

-- :k Functor, Functor :: (* -> *) -> Constraint
-- {-# MINIMAL fmap #-}
instance Functor Tree where
    fmap _ EmptyTree = EmptyTree
    fmap f (Node x ts) = Node (f x) (map (fmap f) ts)
    (<$) :: b -> Tree a -> Tree b
    (<$) x EmptyTree = EmptyTree
    (<$) x (Node _ ts) = Node x (x <$ ts)

-- k: ToList, ToList :: (* -> *) -> Constraint
-- {-# MINIMAL toList #-}
instance ToList Tree where
    toList EmptyTree = []
    toList (Node x ts) = x : concatMap toList ts

-- :k Foldable, Foldable :: (* -> *) -> Constraint
-- {-# MINIMAL foldMap | foldr #-}
instance Foldable Tree where
    foldr _ z EmptyTree = z
    foldr f z (Node x ts) = f x (foldr (\t acc -> foldr f acc t) z ts)
    foldl _ z EmptyTree = z
    foldl f z (Node x ts) = foldl (\acc t -> foldl f acc t) (f z x) ts
    foldMap _ EmptyTree = mempty
    foldMap f (Node x ts) = f x `mappend` foldMap (foldMap f) ts
    foldr1 _ EmptyTree = error "foldr1: empty structure"
    foldr1 f (Node x ts) = foldr f x (concatMap toList ts)
    foldl1 _ EmptyTree = error "foldl1: empty structure"
    foldl1 f (Node x ts) = foldl f x (concatMap toList ts)
    null EmptyTree = True
    null _ = False
    length EmptyTree = 0
    length (Node _ ts) = 1 + sum (map length ts)
    elem _ EmptyTree = False
    elem e (Node x ts) = (e == x) || any (elem e) ts
    maximum EmptyTree = error "maximum: empty structure"
    maximum (Node x ts) = foldl max x (concatMap toList ts)
    minimum EmptyTree = error "minimum: empty structure"
    minimum (Node x ts) = foldl min x (concatMap toList ts)
```

```haskell
    sum EmptyTree = 0
    sum (Node x ts) = x + sum (concatMap toList ts)
    product EmptyTree = 1
    product (Node x ts) = x * product (concatMap toList ts)

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

-- :k Semigroup, Semigroup :: (* -> *) -> Constraint
-- {-# MINIMAL (<>) #-}
instance Semigroup (Tree a) where
    EmptyTree <> t = t
    t <> EmptyTree = t
    Node a ts <> Node b us = Node a (ts ++ [Node b us])

-- :k Monoid, Monoid :: * -> Constraint
-- {-# MINIMAL mempty #-}
instance Monoid (Tree a) where
    mempty = EmptyTree
    mappend = (<>)
    mconcat = foldr mappend mempty

-- :k Applicative, Applicative :: (* -> *) -> Constraint
-- {-# MINIMAL pure, ((<*>) | liftA2) #-}
instance Applicative Tree where
    pure x = Node x []
    EmptyTree <*> _ = EmptyTree
    _ <*> EmptyTree = EmptyTree
    (Node f fs) <*> (Node x xs) = Node (f x) (fs <*> xs)

-- :k Monad, Monad :: (* -> *) -> Constraint
-- {-# MINIMAL (>>=) #-}
instance Monad Tree where
    EmptyTree >>= _ = EmptyTree
    (Node a ts) >>= f = case f a of
        Node b us -> Node b (us ++ concatMap (>>= f) ts)
    return = pure

insert :: a -> Tree a -> Tree a
insert x EmptyTree = Node x []
insert x (Node a ts) = Node a (Node x [] : ts)

delete :: Eq a => a -> Tree a -> Tree a
delete _ EmptyTree = EmptyTree
delete y (Node x ts)
    | x == y = EmptyTree
    | otherwise = Node x (map (delete y) ts)

search :: Eq a => a -> Tree a -> Bool
search _ EmptyTree = False
search y (Node x ts) = (x == y) || any (search y) ts

exampleTree :: Tree Int
exampleTree = insert 3 (insert 2 (insert 1 EmptyTree))

tf = Node (+1) [Node (*2) []] :: Tree (Int -> Int)
tx = Node 1 [Node 2 [Node 3 []]] :: Tree Int
```