Fronty

```
module Queue (Queue(..)) where
class Queue q where
    none::q a
    empty::q a->Bool

    head::q a->a
    tail::q a->q a
    snoc::a->q a->q a
```

Klasická fronta

```
module Classic (Classic,none) where
import Queue(Queue); import qualified Queue

data Classic a = Classic [a] [a] deriving Show
none::Classic a; none=Queue.none

check (Classic [] t) = Classic (reverse t) []
check otherwise = otherwise

instance Queue Classic where
    none = Classic [] []
    empty (Classic h _) = null h

    head (Classic (h:_)  _) = h
    tail (Classic (_:hs) t) = check $ Classic hs t
    snoc x (Classic h t)    = check $ Classic h (x:t)
```

Líně amortizovaná fronta

```
module Amortized (Amortized,none) where
import Queue(Queue);  import qualified Queue

data Amortized a = Amortized !Int [a] !Int [a] deriving Show
none::Amortized a; none=Queue.none

check (Amortized lh h lt t) | lh<lt = Amortized (lh+lt) (h++reverse t) 0 []
check otherwise = otherwise

instance Queue Amortized where
    none = Amortized 0 [] 0 []
    empty (Amortized _ h _ _) = null h

    head (Amortized _ (h:_) _ _) = h
    tail (Amortized lenh (_:hs) lent t) = check $ Amortized (lenh-1) hs lent t
    snoc x (Amortized lenh h lent t) = check $ Amortized lenh h (lent+1) (x:t)
```

Realtime fronta

```
module Realtime (Realtime,none) where
import Queue(Queue); import qualified Queue

data Realtime a = Realtime {getH,getT,getS::[a]} deriving Show
none::Realtime a; none=Queue.none

rotate [] [y] acc = y:acc
rotate (x:xs) (y:ys) acc = x:rotate xs ys (y:acc)
work (Realtime h t (s:ss)) = Realtime h t ss
work (Realtime h t [])     = let h'=rotate h t [] in Realtime h' [] h'

instance Queue Realtime where
    none = Realtime [] [] []
    empty q = null (getH q)

    head q = head (getH q)
    tail q = work $ q {getH=tail (getH q)}
    snoc x q = work $ q {getT=x:getT q}
```

Obecná pole Data.Array.IArray

```
class IArray a e where ...
array :: (IArray a e, Ix i) => (i, i) -> [(i, e)] -> a i e
listArray :: (IArray a e, Ix i) => (i, i) -> [e] -> a i e
accumArray :: (IArray a e, Ix i) => (e->f->e) -> e -> (i, i) -> [(i,f)] -> a i e
(!) :: (IArray a e, Ix i) => a i e -> i -> e
indices :: (IArray a e, Ix i) => a i e -> [i]
elems :: (IArray a e, Ix i) => a i e -> [e]
assocs :: (IArray a e, Ix i) => a i e -> [(i, e)]
(//) :: (IArray a e, Ix i) => a i e -> [(i, e)] -> a i e
accum :: (IArray a e, Ix i) => (e -> e' -> e) -> a i e -> [(i, e')] -> a i e
amap :: (IArray a e', IArray a e, Ix i) => (e' -> e) -> a i e' -> a i e
ixmap :: (IArray a e, Ix i, Ix j) => (i, i) -> (i -> j) -> a j e -> a i e

instance IArray Array e
v Data.Array.Unboxed je instance IArray UArray boxed
boxed může být Bool Double Float Char Int Int{8,16,32,64} Word Word{8,16,32,64}
```

♣ Rozdílová pole v Data.Array.Diff

```
instance IArray DiffArray e
instance IArray DiffUArray boxed
```

přístup a modifikace nejnovější verze je v O(1)

Rychlá pole v monádě

```
class (Monad m) => MArray a e m where
    getBounds :: Ix i => a i e -> m (i,i)
    newArray  :: Ix i => (i,i) -> e -> m (a i e)
    newArray_ :: Ix i => (i,i) -> m (a i e)
    -- the following are private
    getNumElements  :: Ix i => a i e -> m Int
    unsafeNewArray_ :: Ix i => (i,i) -> m (a i e)
    unsafeRead  :: Ix i => a i e -> Int -> m e
    unsafeWrite :: Ix i => a i e -> Int -> e -> m ()

newListArray :: (MArray a e m, Ix i) => (i, i) -> [e] -> m (a i e)
readArray :: (MArray a e m, Ix i) => a i e -> i -> m e
writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()
getElems :: (MArray a e m, Ix i) => a i e -> m [e]
getAssocs :: (MArray a e m, Ix i) => a i e -> m [(i, e)]
mapArray ::(MArray a e' m, MArray a e m, Ix i) => (e'->e) -> a i e' -> m (a i e)
mapIndices ::(MArray a e m, Ix i, Ix j) => (i,i) -> (i->j) -> a j e -> m (a i e)
freeze,unsafeFreeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
thaw, unsafeThaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)

instance MArray IOArray e IO
instance MArray IOUArray boxed IO
boxed může být Bool Double Float Char Int Int{8,16,32,64} Word Word{8,16,32,64}

permute :: [Int] -> IOArray Int e -> IO (IOArray Int e)
permute p a = do res <- getBounds a >>= newArray_
                 permute' 0 p res
                 return res
 where
  permute' _ []    _   = return ()
  permute' i (j:p) res = do readArray a j >>= writeArray res i
                            permute' (i+1) p res

permute::(MArray a e m) => [Int] -> a Int e -> m (a Int e)
permute p arr =
  let pa = array (0, length p - 1) (zip p [0..])
      pi = (pa !)
  in do bnds <- getBounds arr
        mapIndices bnds pi arr
```

```
                         Odkazy v monádě IO
                         ------------------
module Data.IORef
data IORef a
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
modifyIORef ref f = readIORef ref >>= writeIORef ref . f
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
```