Chatovací server

```
import Control.Monad; import Control.Concurrent; import Data.List
import Network;       import System;             import System.IO

msg_new n   h = hPutStrLn h ("New user " ++ n)
msg_msg n m h = hPutStrLn h (n++": "++m)
msg_out n   h = hPutStrLn h ("User " ++ n ++ " logged out")

user_new n h hs =let hs'=       (n,h):hs in mapM_(msg_new n.snd)hs' >>return hs'
user_msg n m hs =                        mapM_(msg_msg n m.snd)hs>>return hs
user_out n h hs =let hs'=delete (n,h) hs in mapM_(msg_out n.snd)hs' >>return hs'
user_kic n hs = mapM_ hClose [h|(n',h)<-hs,n==n'] >> return hs

hGetLineNoCr h = liftM (delete '\r') $ hGetLine h
serve_client m h = do hSetBuffering h LineBuffering
                      n<-hGetLineNoCr h
                      putMVar m (user_new n h)
                      forever (hGetLineNoCr h >>= putMVar m.user_msg n) `catch`
                       \e->putMVar m $ user_out n h
server_listen m s = forever $ accept s >>= \(h,_,_)->forkIO $ serve_client m h
server_console m = forever $ getLine >>= \s->case words s of
                    ["quit"]  ->putMVar m (\hs->exitWith ExitSuccess)
                    ["kick",n]->putMVar m (user_kic n)
                    ["users"] ->putMVar m (\hs->mapM_(putStrLn.fst)hs>>return hs)
                    otherwise ->return ()
server port = do s<-listenOn $ PortNumber port
                 m<-newEmptyMVar
                 forkIO (server_listen m s)
                 forkIO (server_console m)
                 let work hs = takeMVar m >>= ($hs) >>= work
                 work []

main = do a<-getArgs
          case a of [p] -> withSocketsDo $ server $ fromInteger $ read p
                    otherwise -> putStrLn "Usage: chats port_number"
```

Interface 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

-- Implementation
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)
```

Použití fronty

```
module Test where
import Queue(Queue)
import qualified Queue; import qualified Classic; import qualified Amortized

pushOne::(Queue q)=>q Int->q Int
pushOne q = Queue.snoc 1 q

test::(Queue q,Enum a,Num a,Show (q a))=>q a->IO ()
test q = putStrLn $ show $ foldl (flip Queue.snoc) q [1..100]
-- Použití: test Classic.none
```

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

-- Implementation
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

-- Implementation
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}
```

Deque

```
module Deque (module Queue,Deque(..)) where
import Queue

class (Queue q)=>Deque q where
    last::q a->a
    init::q a->q a
    cons::a->q a->q a
```

Líně amortizovaná Deque

```
module AmortizedDE (AmortizedDE,none) where
import Deque(Queue,Deque); import qualified Deque

data AmortizedDE a = AmortizedDE !Int [a] !Int [a] deriving Show

none::AmortizedDE a; none=Deque.none

-- Implementation
check (AmortizedDE lh h lt t)
 | lh>2*lt+1 = let lh'=(lh+lt) `div` 2
           in AmortizedDE lh' (take lh' h) (lh+lt-lh') (t++reverse (drop lh' h))
 | lt>2*lh+1 = let lt'=(lh+lt) `div` 2
           in AmortizedDE (lh+lt-lt') (h++reverse (drop lt' t)) lt' (take lt' t)
check otherwise = otherwise

instance Queue AmortizedDE where
    none = AmortizedDE 0 [] 0 []
    empty (AmortizedDE lh _ lt _) = lh+lt==0

    head (AmortizedDE _  (h:_)  _  _ ) = h
    head (AmortizedDE _     _     _ [h]) = h
    tail (AmortizedDE lh (_:hs) lt t ) = check $ AmortizedDE (lh-1) hs lt t
    tail (AmortizedDE _     _     _ [t]) = none
    snoc x (AmortizedDE lh h    lt t ) = check $ AmortizedDE lh h (lt+1) (x:t)

instance Deque AmortizedDE where
    last (AmortizedDE _  _  _   (t:_)) = t
    last (AmortizedDE _ [t] _    _   ) = t
    init (AmortizedDE lh h  lt (_:ts)) = check $ AmortizedDE lh h (lt-1) ts
    init (AmortizedDE _ [h] _    _   ) = none
    cons x (AmortizedDE lh h lt t    ) = check $ AmortizedDE (lh+1) (x:h) lt t
```

Spojovatelné fronty

```
module CatenableQueue(module Queue,CatenableQueue(..)) where
import Queue

class Queue c=>CatenableQueue c where
    (++)::c a->c a->c a
```

Rekurzivní spojovatelná fronta

```
module RecCQueue(RecCQueue,none) where
import          Queue(         Queue); import qualified          Queue
import CatenableQueue(CatenableQueue); import qualified CatenableQueue

data Queue q=>RecCQueue q a = Empty | Node a (q (RecCQueue q a))

none::(Queue q)=>q z->RecCQueue q a; none _ = CatenableQueue.none

-- Implementation
link (Node x q) other = Node x (Queue.snoc other q)

instance Queue q=>Queue (RecCQueue q) where
    none = Empty
    empty Empty = True
    empty _ = False

    head (Node x q) = x
    tail (Node x q) = if Queue.empty q then Empty else linkAll q
      where linkAll q = let (q1,qs)=(Queue.head q, Queue.tail q)
                        in if Queue.empty qs then q1 else link q1 (linkAll qs)
    snoc x q = q CatenableQueue.++ (Node x Queue.none)
instance Queue q=>CatenableQueue (RecCQueue q) where
    Empty++x = x
    x++Empty = x
    x  ++  y = link x y
```

```
                              Random access lists
                              -------------------
module RAList(RAList(..)) where
class RAList l where
  none::l a;             empty::l a->Bool
  head::l a->a;          tail::l a->l a;          cons::a->l a->l a
  lookup::Int->l a->a;   update::Int->a->l a->l a

                            Binary random access list
                            -------------------------
module BinRAList(BinRAList,none) where
import RAList(RAList);  import qualified RAList;

data LTree a = Leaf a | Node (LTree a) (LTree a);  data Digit a = Zero | One (a)
newtype BinRAList a = BinRAList [Digit (LTree a)];
none::BinRAList a; none=RAList.none
--Implementation
consTree t []        = [One t]
consTree t (Zero:l)  = One t : l
consTree t (One u:l) = Zero : consTree (Node t u) l
unconsTree [One t]   = (t, [])
unconsTree (One t:l) = (t, Zero:l)
unconsTree (Zero:l)  = let (Node a b,l')=unconsTree l in (a,One b:l')
alterTree s i f (One t:l) | i<s = let (x,t')=alterBTree s i f t in (x,One t':l)
alterTree s i f (t:l) = let (x,l')=alterTree (s+s) (i-s) f l in (x,t:l')
alterBTree s 0 f (Leaf a) = (a, Leaf $ f a)
alterBTree s i f (Node x y) = let s2=s`div`2
                                  (al,x')=alterBTree s2 i      f x
                                  (ar,y')=alterBTree s2 (i-s2) f y
                              in if i<s2 then (al,Node x' y) else (ar,Node x y')
instance RAList BinRAList where
  none=BinRAList [];                    empty (BinRAList l) = null l

  cons x (BinRAList l) = BinRAList $ consTree (Leaf x) l
  head (BinRAList l) = let (Leaf x,_)=unconsTree l in x
  tail (BinRAList l) = let (_,l')=unconsTree l in BinRAList l

  lookup i (BinRAList l) = fst $ alterTree 1 i id l
  update i x (BinRAList l) = BinRAList $ snd $ alterTree 1 i (const x) l

                             Skew random access list
                             -----------------------
module SkewRAList(SkewRAList,none) where
import RAList(RAList);  import qualified RAList;

data Tree a = Leaf a | Node a (Tree a) (Tree a)
newtype SkewRAList a = SkewRAList [(Int, Tree a)]
none::SkewRAList a; none=RAList.none
--Implementation
alterTree i f ((s,t):l)|i<s       = let (x,t')=alterBTree s i f t in(x,(s,t'):l)
                       |otherwise = let (x,l')=alterTree(i-s)f l in (x,(s,t):l')
alterBTree s 0 f (Leaf a) = (a, Leaf $ f a)
alterBTree s 0 f (Node a x y) = (a, Node (f a) x y)
alterBTree s i f (Node a x y) = let s2=s`div`2
                                    (al,x')=alterBTree s2  i     f x
                                    (ar,y')=alterBTree s2 (i-s2) f y
                                in if i<s2 then (al,Node a x' y) else (ar,Node a x y')
instance RAList SkewRAList where
  none = SkewRAList [];                 empty (SkewRAList l) = null l

  cons x (SkewRAList ((a,u):(b,v):l)) | a==b = SkewRAList$ (1+a+b,Node x u v):l
  cons x (SkewRAList l) = SkewRAList$ (1,Leaf x):l
  head (SkewRAList ((_,Leaf x    ):_)) = x
  head (SkewRAList ((_,Node x _ _):_)) = x
  tail (SkewRAList ((_,Leaf x    ):l)) = SkewRAList l
  tail (SkewRAList ((a,Node x u v):l)) = SkewRAList$ (a`div`2,u):(a`div`2,v):l

  lookup i (SkewRAList l) = fst $ alterTree i id l
  update i x (SkewRAList l) = SkewRAList $ snd $ alterTree i (const x) l
```