```
data Buffer a = Empty | Write Int a (Buffer a) | Read Int (Buffer a) deriving (Show)

buf = Write 1 "a" $ Write 2 "b" $ Write 1 "c" $ Empty

recBuffer :: b -> (Int -> a -> Buffer a -> b -> b) ->  (Int -> Buffer a -> b -> b) -> Buffer a -> b

recBuffer cEmpty cWrite cRead b = case b of
   Empty -> cEmpty
   Write n x b -> cWrite n x b (rec b)
   Read n b -> cRead n b (rec b)
   where rec = recBuffer cEmpty cWrite cRead

foldBuffer :: b -> (Int -> a -> b -> b) ->  (Int -> b -> b) -> Buffer a -> b
{-
foldBuffer cEmpty cWrite cRead b = case b of
   Empty -> cEmpty
   Write n x b -> cWrite n x (rec b)
   Read n b -> cRead n (rec b)
   where rec = foldBuffer cEmpty cWrite cRead
-}
foldBuffer cEmpty cWrite cRead = recBuffer cEmpty (\n x _ -> cWrite n x) (\n _ -> cRead n)

posicionesOcupadas::Buffer a -> [Int]
posicionesOcupadas = foldBuffer [] (\n _ rec ->  [n] ++ rec ) (\n rec -> filter (\e -> e /= n) rec )

contenido::Int -> Buffer a -> Maybe a
contenido entrada = foldBuffer Nothing (\n x rec -> if n == entrada then Just x else rec) (\n rec -> if n == entrada
then Nothing else rec)

puedeCompletarLecturas:: Buffer a -> Bool
puedeCompletarLecturas = recBuffer True (\n _ _ rec -> rec ) (\n b rec -> (hayUnWrite n b) && rec)

hayUnWrite :: Int -> Buffer a -> Bool
hayUnWrite entrada = foldBuffer False (\n _ rec -> if n==entrada then True else rec) (\n rec -> if n==entrada then
False else rec)

deshacer::Buffer a -> Int -> Buffer a
deshacer = recBuffer (\_ -> Empty)
            (\_ _ b rec -> \i -> if i == 1 then b else rec (i-1))
            (\_ b rec -> \i -> if i == 1 then b else rec (i-1) )




data Prop = Var String | No Prop | Y Prop Prop | O Prop Prop | Imp Prop Prop

type Valuacion = String -> Bool
```

```
recProp :: (String -> b) -> (Prop -> b -> b) -> (Prop -> Prop -> b -> b ->b ) -> (Prop -> Prop -> b -> b ->b ) ->
(Prop -> Prop -> b -> b ->b ) -> Prop -> b

recProp cVar cNo cY cO cImp p= case p of
  Var s -> cVar s
  No p -> cNo p (rec p)
  Y p1 p2 -> cY p1 p2 (rec p1) (rec p2)
  O p1 p2 -> cO p1 p2 (rec p1) (rec p2)
  Imp p1 p2 -> cImp p1 p2 (rec p1) (rec p2)
  where rec = recProp cVar cNo cY cO cImp

foldProp :: (String -> b) -> (b -> b) -> (b -> b -> b ) -> (b -> b -> b ) -> (b -> b -> b ) -> Prop -> b
foldProp cVar cNo cY cO cImp = recProp cVar (\_-> cNo) (\_ _ -> cY) (\_ _ -> cO) (\_ _ -> cImp)
{- por alguna razon no funciona el nub
variables :: Prop -> [String]
variables p = nub (variablesAux p)
-}
variablesAux :: Prop -> [String]
variablesAux = foldProp (\s -> [s]) (\rec -> rec) (\rec1 rec2 -> rec1 ++ rec2) (\rec1 rec2 -> rec1 ++ rec2) (\rec1
rec2 -> rec1 ++ rec2)

evaluar :: Valuacion -> Prop -> Bool
evaluar v = foldProp (\s -> v s)
                (\rec -> not rec)
                (\rec1 rec2 -> rec1 && rec2)
                (\rec1 rec2 -> rec1 || rec2)
                (\rec1 rec2 -> not rec1 ||  rec2)

estaEnFNN :: Prop -> Bool
estaEnFNN = recProp (\s -> True)
                (\p _ -> case p of
                  Var _ -> True
                  _-> False
                  )
                (\_ _ rec1 rec2 -> rec1 && rec2)
                (\_ _ rec1 rec2 -> rec1 && rec2)
                (\_ _ _ _ -> False)


data RoseTree a = Rose a [RoseTree a]

foldRT :: (a -> [b] -> b) -> RoseTree a -> b
foldRT f (Rose x hijos) = f x (map (foldRT f) hijos)

tamaño :: RoseTree a -> Int
tamaño = foldRT (\_ recs -> 1 + sum recs)

hojas :: RoseTree a -> [a]
```

```
hojas = foldRT (\x recs -> [x] ++ concat recs)

distancias :: RoseTree a -> [Int]
distancias = foldRT (\_ hijos -> if null hijos then [0] else map (+1) (concat hijos))

ejemplo :: RoseTree Int
ejemplo = Rose 1
  [ Rose 2 []
  , Rose 3
      [ Rose 4 []
      , Rose 5 []
      ]
  , Rose 6 []
  ]


data AT a = NilT | Tri a (AT a ) (AT a) (AT a) deriving (Show)

foldAT :: b -> (a -> b -> b -> b -> b) -> AT a -> b
foldAT cNilT cTri a = case a of
    NilT -> cNilT
    Tri x i m d -> cTri x (rec i) (rec m) (rec d)
    where rec = foldAT cNilT cTri

preorder :: AT a -> [a]

preorder = foldAT [] (\x reci recm recd -> [x] ++ reci ++ recm ++ recd)

at1 = Tri 1 (Tri 2 NilT NilT NilT) (Tri 3 (Tri 4 NilT NilT NilT) NilT NilT) (Tri 5 NilT NilT NilT)


mapAT :: (a -> b) -> AT a -> AT b
mapAT f = foldAT NilT (\x reci recm recd -> Tri (f x) reci recm recd)

nivel :: AT a -> Int -> [a]
nivel = foldAT (\_ -> []) (\x reci recm recd -> \i -> if i == 0 then [x] else ( reci (i-1) ++ recm (i-1) ) ++ recd(i-1))
```