

Tema 15: El TAD de las colas

Informática (2011–12)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
 - Signatura del TAD de las colas
 - Propiedades del TAD de las colas
2. Implementaciones del TAD de las colas
 - Implementación de las colas mediante listas
 - Implementación de las colas mediante pares de listas
3. Comprobación de las implementaciones con QuickCheck
 - Librerías auxiliares
 - Generador de colas
 - Especificación de las propiedades de las colas
 - Comprobación de las propiedades

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
 - Signatura del TAD de las colas
 - Propiedades del TAD de las colas
2. Implementaciones del TAD de las colas
3. Comprobación de las implementaciones con QuickCheck

Descripción informal de las colas

- ▶ Una **cola** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo (el posterior o final) y la operación de extracción por el otro (el anterior o frente).
- ▶ Las colas también se llaman estructuras FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.
- ▶ Analogía con las colas del cine.

Signatura del TAD colas

► Signatura:

```
vacía    :: Cola a
inserta  :: a -> Cola a -> Cola a
primero  :: Cola a -> a
resto    :: Cola a -> Cola a
esVacía  :: Cola a -> Bool
válida   :: Cola a -> Bool
```

► Descripción de las operaciones:

- `vacía` es la cola vacía.
- `(inserta x c)` es la cola obtenida añadiendo `x` al final de `c`.
- `(primero c)` es el primero de la cola `c`.
- `(resto c)` es la cola obtenida eliminando el primero de `c`.
- `(esVacía c)` se verifica si `c` es la cola vacía.
- `(válida c)` se verifica si `c` representa una cola válida.

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas

 Signatura del TAD de las colas

 Propiedades del TAD de las colas

2. Implementaciones del TAD de las colas

3. Comprobación de las implementaciones con QuickCheck

Propiedades del TAD de las colas

1. `primero (inserta x vacia) == x`
2. Si `c` es una cola no vacía, entonces
`primero (inserta x c) == primero c,`
3. `resto (inserta x vacia) == vacia`
4. Si `c` es una cola no vacía, entonces
`resto (inserta x c) == inserta x (resto c)`
5. `esVacia vacia`
6. `not (esVacia (inserta x c))`

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
2. Implementaciones del TAD de las colas
 - Implementación de las colas mediante listas
 - Implementación de las colas mediante pares de listas
3. Comprobación de las implementaciones con QuickCheck

Implementación de las colas mediante listas

- ▶ Cabecera del módulo:

```
module ColaConListas
  (Cola,
   vacia,    -- Cola a
   inserta,  -- a -> Cola a -> Cola a
   primero,  -- Cola a -> a
   resto,    -- Cola a -> Cola a
   esVacia,  -- Cola a -> Bool
   valida    -- Cola a -> Bool
  ) where
```

- ▶ Representación de las colas mediante listas:

```
newtype Cola a = C [a] deriving (Show, Eq)
```

Implementación de las colas mediante listas

- ▶ Ejemplo de cola: `c1` es la cola obtenida añadiéndole a la cola vacía los números del 1 al 10. Por ejemplo,

```
| ghci> c1  
| C [10,9,8,7,6,5,4,3,2,1]
```

```
c1 = foldr inserta vacia [1..10]
```

- ▶ `vacía` es la cola vacía. Por ejemplo,

```
| ghci> vacia  
| C []
```

```
vacia :: Cola a  
vacia = C []
```

Implementación de las colas mediante listas

- ▶ Ejemplo de cola: `c1` es la cola obtenida añadiéndole a la cola vacía los números del 1 al 10. Por ejemplo,

```
| ghci> c1  
| C [10,9,8,7,6,5,4,3,2,1]
```

```
c1 = foldr inserta vacia [1..10]
```

- ▶ `vacía` es la cola vacía. Por ejemplo,

```
| ghci> vacia  
| C []
```

```
vacía :: Cola a  
vacía = C []
```

Implementación de las colas mediante listas

- ▶ `(inserta x c)` es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
inserta 12 c1  ~>  C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
inserta x (C c) = C (c ++ [x])
```

- ▶ `(primero c)` es el primer elemento de la cola `c`. Por ejemplo,

```
primero c1  ~>  10
```

```
primero :: Cola a -> a
primero (C (x:_)) = x
primero (C [])   = error "primero: cola vacia"
```

Implementación de las colas mediante listas

- ▶ (`inserta x c`) es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
| inserta 12 c1  ~>  C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
```

```
inserta x (C c) = C (c ++ [x])
```

- ▶ (`primero c`) es el primer elemento de la cola `c`. Por ejemplo,

```
| primero c1  ~>  10
```

```
primero :: Cola a -> a
```

```
primero (C (x:_)) = x
```

```
primero (C [])     = error "primero: cola vacia"
```

Implementación de las colas mediante listas

- ▶ (`inserta x c`) es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
| inserta 12 c1  ~>  C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
```

```
inserta x (C c) = C (c ++ [x])
```

- ▶ (`primero c`) es el primer elemento de la cola `c`. Por ejemplo,

```
| primero c1  ~>  10
```

```
primero :: Cola a -> a
```

```
primero (C (x:_)) = x
```

```
primero (C [])    = error "primero: cola vacia"
```

Implementación de las colas mediante listas

- ▶ `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ~>  C [9,8,7,6,5,4,3,2,1]
```

```
resto :: Cola a -> Cola a
```

```
resto (C (_:xs)) = C xs
```

```
resto (C [])      = error "resto: cola vacia"
```

- ▶ `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1      ~>  False
```

```
esVacia vacia  ~>  True
```

```
esVacia :: Cola a -> Bool
```

```
esVacia (C xs) = null xs
```

Implementación de las colas mediante listas

- ▶ `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ~>  C [9,8,7,6,5,4,3,2,1]
```

```
resto :: Cola a -> Cola a
```

```
resto (C (_:xs)) = C xs
```

```
resto (C [])     = error "resto: cola vacia"
```

- ▶ `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1      ~>  False
```

```
esVacia vacia  ~>  True
```

```
esVacia :: Cola a -> Bool
```

```
esVacia (C xs) = null xs
```

Implementación de las colas mediante listas

- ▶ (`resto c`) es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ~>  C [9,8,7,6,5,4,3,2,1]
```

```
resto :: Cola a -> Cola a
```

```
resto (C (_:xs)) = C xs
```

```
resto (C [])      = error "resto: cola vacia"
```

- ▶ (`esVacia c`) se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1      ~>  False
```

```
esVacia vacia  ~>  True
```

```
esVacia :: Cola a -> Bool
```

```
esVacia (C xs) = null xs
```

Implementación de las colas mediante listas

- ▶ `(valida c)` se verifica si `c` representa una cola válida. Con esta representación, todas las colas son válidas.

```
valida :: Cola a -> Bool  
valida c = True
```

Implementación de las colas mediante listas

- ▶ `(valida c)` se verifica si `c` representa una cola válida. Con esta representación, todas las colas son válidas.

```
valida :: Cola a -> Bool
```

```
valida c = True
```

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
2. Implementaciones del TAD de las colas
 - Implementación de las colas mediante listas
 - Implementación de las colas mediante pases de listas
3. Comprobación de las implementaciones con QuickCheck

Las colas como pares de listas

- ▶ En esta implementación, una cola c se representa mediante un par de listas (xs, ys) de modo que los elementos de c son, en ese orden, los elementos de la lista $xs++(\text{reverse } ys)$.
- ▶ Al dividir la lista en dos partes e invertir la segunda de ellas, esperamos hacer más eficiente las operaciones sobre las colas.
- ▶ Impondremos también una restricción adicional sobre la representación: las colas serán representadas mediante pares (xs, ys) tales que si xs es vacía, entonces ys será también vacía.
- ▶ Esta restricción ha de mantenerse por las operaciones que crean colas.

Implementación de las colas como pares de listas

► Cabecera del módulo

```
module ColaConDosListas
  (Cola,
   vacia,    -- Cola a
   inserta,  -- a -> Cola a -> Cola a
   primero,  -- Cola a -> a
   resto,    -- Cola a -> Cola a
   esVacia,  -- Cola a -> Bool
   valida    -- Cola a -> Bool
  ) where
```

► Las colas como pares de listas

```
newtype Cola a = C ([a],[a])
```

Implementación de las colas como pares de listas

- ▶ `(valida c)` se verifica si la cola `c` es válida; es decir, si su primer elemento es vacío entonces también lo es el segundo. Por ejemplo,

```
valida (C ([2], [5]))  ~>  True
valida (C ([2], []))  ~>  True
valida (C ([], [5]))  ~>  False
```

```
valida:: Cola a -> Bool
```

```
valida (C (xs,ys)) = not (null xs) || null ys
```

Implementación de las colas como pares de listas

- ▶ `(valida c)` se verifica si la cola `c` es válida; es decir, si su primer elemento es vacío entonces también lo es el segundo. Por ejemplo,

```
valida (C ([2], [5]))  ~>  True
valida (C ([2], []))  ~>  True
valida (C ([], [5]))  ~>  False
```

```
valida:: Cola a -> Bool
```

```
valida (C (xs,ys)) = not (null xs) || null ys
```

Implementación de las colas como pares de listas

- ▶ Procedimiento de escritura de colas como pares de listas.

```
instance Show a => Show (Cola a) where
    showsPrec p (C (xs,ys)) cad
        = showString "C " (showList (xs ++ (reverse ys)) c
```

- ▶ Ejemplo de cola: `c1` es la cola obtenida añadiéndole a la cola vacía los números del 1 al 10. Por ejemplo,

```
| ghci> c1
| C [10,9,8,7,6,5,4,3,2,1]
```

```
c1 :: Cola Int
c1 = foldr inserta vacia [1..10]
```

Implementación de las colas como pares de listas

- ▶ `vacía` es la cola vacía. Por ejemplo,

```
ghci> c1  
C [10,9,8,7,6,5,4,3,2,1]
```

```
vacía :: Cola a
```

```
vacía = C ([],[])
```

- ▶ `(inserta x c)` es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
inserta 12 c1 ~> C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
```

```
inserta y (C (xs,ys)) = C (normaliza (xs,y:ys))
```

Implementación de las colas como pares de listas

- ▶ `vacía` es la cola vacía. Por ejemplo,

```
ghci> c1  
C [10,9,8,7,6,5,4,3,2,1]
```

```
vacía :: Cola a
```

```
vacía = C ([],[])
```

- ▶ `(inserta x c)` es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
inserta 12 c1 ~> C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
```

```
inserta y (C (xs,ys)) = C (normaliza (xs,y:ys))
```

Implementación de las colas como pares de listas

- ▶ `vacía` es la cola vacía. Por ejemplo,

```
ghci> c1
C [10,9,8,7,6,5,4,3,2,1]
```

```
vacía :: Cola a
vacía = C ([],[])
```

- ▶ `(inserta x c)` es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
inserta 12 c1 ~> C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
inserta y (C (xs,ys)) = C (normaliza (xs,y:ys))
```

Implementación de las colas como pares de listas

- ▶ (`normaliza p`) es la cola obtenida al normalizar el par de listas `p`. Por ejemplo,

```
normaliza ([], [2,5,3])  ~> ([3,5,2], [])
normaliza ([4], [2,5,3]) ~> ([4], [2,5,3])
```

```
normaliza :: ([a],[a]) -> ([a],[a])
normaliza ([], ys) = (reverse ys, [])
normaliza p      = p
```

- ▶ (`primero c`) es el primer elemento de la cola `c`. Por ejemplo,
- ```
primero c1 ~> 10
```

---

```
primero :: Cola a -> a
primero (C (x:xs,ys)) = x
primero _ = error "primero: cola vacia"
```

---

## Implementación de las colas como pares de listas

- `(normaliza p)` es la cola obtenida al normalizar el par de listas `p`. Por ejemplo,

```
normaliza ([], [2,5,3]) ~> ([3,5,2], [])
normaliza ([4], [2,5,3]) ~> ([4], [2,5,3])
```

---

```
normaliza :: ([a],[a]) -> ([a],[a])
normaliza ([], ys) = (reverse ys, [])
normaliza p = p
```

---

- `(primero c)` es el primer elemento de la cola `c`. Por ejemplo,
- ```
primero c1 ~> 10
```

```
primero :: Cola a -> a
primero (C (x:xs,ys)) = x
primero _             = error "primero: cola vacia"
```

Implementación de las colas como pares de listas

- `(normaliza p)` es la cola obtenida al normalizar el par de listas `p`. Por ejemplo,

```
normaliza ([], [2,5,3])  ~> ([3,5,2], [])
normaliza ([4], [2,5,3]) ~> ([4], [2,5,3])
```

```
normaliza :: ([a], [a]) -> ([a], [a])
normaliza ([], ys) = (reverse ys, [])
normaliza p      = p
```

- `(primero c)` es el primer elemento de la cola `c`. Por ejemplo,

```
primero c1 ~> 10
```

```
primero :: Cola a -> a
primero (C (x:xs,ys)) = x
primero _              = error "primero: cola vacia"
```

Implementación de las colas como pares de listas

- ▶ `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
| resto c1  ~>  C [9,8,7,6,5,4,3,2,1]
```

```
resto  :: Cola a -> Cola a
resto (C (x:xs,ys)) = C (normaliza (xs,ys))
resto (C ([],[]))   = error "resto: cola vacia"
```

- ▶ `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
| esVacia c1      ~>  False
| esVacia vacia  ~>  True
```

```
esVacia :: Cola a -> Bool
esVacia (C (xs,_)) = null xs
```

Implementación de las colas como pares de listas

- ▶ `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ~>  C [9,8,7,6,5,4,3,2,1]
```

```
resto  :: Cola a -> Cola a
resto (C (x:xs,ys)) = C (normaliza (xs,ys))
resto (C ([],[]))   = error "resto: cola vacia"
```

- ▶ `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1      ~>  False
esVacia vacia   ~>  True
```

```
esVacia :: Cola a -> Bool
esVacia (C (xs,_)) = null xs
```

Implementación de las colas como pares de listas

- ▶ `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ~>  C [9,8,7,6,5,4,3,2,1]
```

```
resto  :: Cola a -> Cola a
resto (C (x:xs,ys)) = C (normaliza (xs,ys))
resto (C ([],[]))   = error "resto: cola vacia"
```

- ▶ `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1      ~>  False
esVacia vacia  ~>  True
```

```
esVacia :: Cola a -> Bool
esVacia (C (xs,_)) = null xs
```

Implementación de las colas como pares de listas

- ▶ `(elementos c)` es la lista de los elementos de la cola `c` en el orden de la cola. Por ejemplo,

```
| elementos (C ([3,2], [5,4,7])) ~> [3,2,7,4,5]
```

```
elementos :: Cola a -> [a]
```

```
elementos (C (xs,ys)) = xs ++ (reverse ys)
```

Implementación de las colas como pares de listas

- ▶ `(elementos c)` es la lista de los elementos de la cola `c` en el orden de la cola. Por ejemplo,

```
| elementos (C ([3,2], [5,4,7])) ~> [3,2,7,4,5]
```

```
elementos :: Cola a -> [a]
```

```
elementos (C (xs,ys)) = xs ++ (reverse ys)
```

Implementación de las colas como pares de listas

- ▶ `(igualColas c1 c2)` se verifica si las colas `c1` y `c2` son iguales.

```
ghci> igualColas (C ([3,2], [5,4,7])) (C ([3], [5,4,7,2]))
True
ghci> igualColas (C ([3,2], [5,4,7])) (C ([], [5,4,7,2,3]))
False
```

```
igualColas c1 c2 =
    valida c1 && valida c2 &&
    elementos c1 == elementos c2
```

- ▶ Extensión de la igualdad a las colas:

```
instance (Eq a) => Eq (Cola a) where
    (==) = igualColas
```

Implementación de las colas como pares de listas

- ▶ `(igualColas c1 c2)` se verifica si las colas `c1` y `c2` son iguales.

```
ghci> igualColas (C ([3,2], [5,4,7])) (C ([3], [5,4,7,2]))
True
ghci> igualColas (C ([3,2], [5,4,7])) (C ([], [5,4,7,2,3]))
False
```

```
igualColas c1 c2 =
    valida c1 && valida c2 &&
    elementos c1 == elementos c2
```

- ▶ Extensión de la igualdad a las colas:

```
instance (Eq a) => Eq (Cola a) where
    (==) = igualColas
```

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
2. Implementaciones del TAD de las colas
3. Comprobación de las implementaciones con QuickCheck
 - Librerías auxiliares
 - Generador de colas
 - Especificación de las propiedades de las colas
 - Comprobación de las propiedades

Importación de librerías

- ▶ Importación de la implementación de las colas que se desea comprobar.

```
import ColaConListas
-- import ColaConDosListas
```

- ▶ Importación de librerías auxiliares

```
import Data.List
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
2. Implementaciones del TAD de las colas
3. Comprobación de las implementaciones con QuickCheck
 - Librerías auxiliares
 - Generador de colas**
 - Especificación de las propiedades de las colas
 - Comprobación de las propiedades

Generador de colas

- `genCola` es un generador de colas de enteros. Por ejemplo,

```
ghci> sample genCola
C ([7,8,4,3,7],[5,3,3])
C ([1],[13])
...
```

```
genCola :: Gen (Cola Int)
genCola = frequency [(1, return vacia),
                    (30, do n <- choose (10,100)
                          xs <- vectorOf n arbitrary
                          return (creaCola xs))]
  where creaCola = foldr inserta vacia
```

```
instance Arbitrary (Cola Int) where
  arbitrary = genCola
```

Corrección del generador de colas

- ▶ Propiedad: Todo los elementos generados por `genCola` son colas válidas.

```
prop_genCola_correcto :: Cola Int -> Bool
prop_genCola_correcto c = valida c
```

- ▶ Comprobación.

```
| ghci> quickCheck prop_genCola_correcto
| +++ OK, passed 100 tests.
```

Corrección del generador de colas

- ▶ Propiedad: Todo los elementos generados por `genCola` son colas válidas.

```
prop_genCola_correcto :: Cola Int -> Bool
prop_genCola_correcto c = valida c
```

- ▶ Comprobación.

```
| ghci> quickCheck prop_genCola_correcto
| +++ OK, passed 100 tests.
```

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
2. Implementaciones del TAD de las colas
3. Comprobación de las implementaciones con QuickCheck
 - Librerías auxiliares
 - Generador de colas
 - Especificación de las propiedades de las colas
 - Comprobación de las propiedades

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ El primero de la cola obtenida añadiendo x a la cola vacía es x .

```
prop_primeros_inserta_vacia :: Int -> Bool
prop_primeros_inserta_vacia x =
  primero (inserta x vacia) == x
```

- ▶ Si una cola no está vacía, su primer elemento no varía al añadirle un elemento.

```
prop_primeros_inserta_no_vacia :: Cola Int -> Int -> Int
                                     -> Bool
prop_primeros_inserta_no_vacia c x y =
  primero (inserta x c') == primero c'
  where c' = inserta y vacia
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ El primero de la cola obtenida añadiendo x a la cola vacía es x .

```
prop_primeros_inserta_vacia :: Int -> Bool
prop_primeros_inserta_vacia x =
  primero (inserta x vacia) == x
```

- ▶ Si una cola no está vacía, su primer elemento no varía al añadirle un elemento.

```
prop_primeros_inserta_no_vacia :: Cola Int -> Int -> Int
                                     -> Bool
prop_primeros_inserta_no_vacia c x y =
  primero (inserta x c') == primero c'
  where c' = inserta y vacia
```

Especificación de las propiedades de las colas

- ▶ El primero de la cola obtenida añadiendo x a la cola vacía es x .

```
prop_primeros_inserta_vacia :: Int -> Bool
prop_primeros_inserta_vacia x =
    primero (inserta x vacia) == x
```

- ▶ Si una cola no está vacía, su primer elemento no varía al añadirle un elemento.

```
prop_primeros_inserta_no_vacia :: Cola Int -> Int -> Int
                                     -> Bool
prop_primeros_inserta_no_vacia c x y =
    primero (inserta x c') == primero c'
  where c' = inserta y vacia
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ El resto de la cola obtenida insertando un elemento en la cola vacía es la cola vacía.

```
prop_resto_inserta_vacia :: Int -> Bool
prop_resto_inserta_vacia x =
    resto (inserta x vacia) == vacia
```

- ▶ Las operaciones de encolar y desencolar conmutan.

```
prop_resto_inserta_en_no_vacia :: Cola Int -> Int -> Int
                                -> Bool
prop_resto_inserta_en_no_vacia c x y =
    resto (inserta x c') == inserta x (resto c')
    where c' = inserta y c
```

Especificación de las propiedades de las colas

- ▶ El resto de la cola obtenida insertando un elemento en la cola vacía es la cola vacía.

```
prop_resto_inserta_vacia :: Int -> Bool
prop_resto_inserta_vacia x =
  resto (inserta x vacia) == vacia
```

- ▶ Las operaciones de encolar y desencolar conmutan.

```
prop_resto_inserta_en_no_vacia :: Cola Int -> Int -> Int
                                -> Bool
prop_resto_inserta_en_no_vacia c x y =
  resto (inserta x c') == inserta x (resto c')
  where c' = inserta y c
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ El resto de la cola obtenida insertando un elemento en la cola vacía es la cola vacía.

```
prop_resto_inserta_vacia :: Int -> Bool
prop_resto_inserta_vacia x =
    resto (inserta x vacia) == vacia
```

- ▶ Las operaciones de encolar y desencolar conmutan.

```
prop_resto_inserta_en_no_vacia :: Cola Int -> Int -> Int
                                -> Bool
prop_resto_inserta_en_no_vacia c x y =
    resto (inserta x c') == inserta x (resto c')
    where c' = inserta y c
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ vacia es una cola vacía.

```
prop_vacia_es_vacia :: Bool
prop_vacia_es_vacia =
    esVacia vacia
```

- ▶ La cola obtenida insertando un elemento no es vacía.

```
prop_inserta_no_es_vacia :: Int -> Cola Int -> Bool
prop_inserta_no_es_vacia x c =
    not (esVacia (inserta x c))
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ vacia es una cola vacía.

```
prop_vacia_es_vacia :: Bool
prop_vacia_es_vacia =
    esVacia vacia
```

- ▶ La cola obtenida insertando un elemento no es vacía.

```
prop_inserta_no_es_vacia :: Int -> Cola Int -> Bool
prop_inserta_no_es_vacia x c =
    not (esVacia (inserta x c))
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ vacia es una cola vacía.

```
prop_vacia_es_vacia :: Bool
prop_vacia_es_vacia =
    esVacia vacia
```

- ▶ La cola obtenida insertando un elemento no es vacía.

```
prop_inserta_no_es_vacia :: Int -> Cola Int -> Bool
prop_inserta_no_es_vacia x c =
    not (esVacia (inserta x c))
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ La cola vacía es válida.

```
prop_valida_vacia :: Bool
prop_valida_vacia = valida vacia
```

- ▶ Al añadirle un elemento a una cola válida se obtiene otra válida.

```
prop_valida_inserta :: Cola Int -> Int -> Property
prop_valida_inserta c x =
    valida c ==> valida (inserta x c)
```

- ▶ El resto de una cola válida y no vacía es una cola válida.

```
prop_valida_resto :: Cola Int -> Property
prop_valida_resto c =
    valida c && not (esVacía c) ==> valida (resto c)
```

- └ Comprobación de las implementaciones con QuickCheck
- └ Especificación de las propiedades de las colas

Especificación de las propiedades de las colas

- ▶ La cola vacía es válida.

```
prop_valida_vacia :: Bool
prop_valida_vacia = valida vacia
```

- ▶ Al añadirle un elemento a una cola válida se obtiene otra válida.

```
prop_valida_inserta :: Cola Int -> Int -> Property
prop_valida_inserta c x =
    valida c ==> valida (inserta x c)
```

- ▶ El resto de una cola válida y no vacía es una cola válida.

```
prop_valida_resto :: Cola Int -> Property
prop_valida_resto c =
    valida c && not (esVacia c) ==> valida (resto c)
```

Especificación de las propiedades de las colas

- ▶ La cola vacía es válida.

```
prop_valida_vacia :: Bool
prop_valida_vacia = valida vacia
```

- ▶ Al añadirle un elemento a una cola válida se obtiene otra válida.

```
prop_valida_inserta :: Cola Int -> Int -> Property
prop_valida_inserta c x =
  valida c ==> valida (inserta x c)
```

- ▶ El resto de una cola válida y no vacía es una cola válida.

```
prop_valida_resto :: Cola Int -> Property
prop_valida_resto c =
  valida c && not (esVacia c) ==> valida (resto c)
```

Especificación de las propiedades de las colas

- ▶ La cola vacía es válida.

```
prop_valida_vacia :: Bool
prop_valida_vacia = valida vacia
```

- ▶ Al añadirle un elemento a una cola válida se obtiene otra válida.

```
prop_valida_inserta :: Cola Int -> Int -> Property
prop_valida_inserta c x =
  valida c ==> valida (inserta x c)
```

- ▶ El resto de una cola válida y no vacía es una cola válida.

```
prop_valida_resto :: Cola Int -> Property
prop_valida_resto c =
  valida c && not (esVacía c) ==> valida (resto c)
```

Tema 15: El TAD de las colas

1. Especificación del TAD de las colas
2. Implementaciones del TAD de las colas
3. Comprobación de las implementaciones con QuickCheck
 - Librerías auxiliares
 - Generador de colas
 - Especificación de las propiedades de las colas
 - Comprobación de las propiedades

Definición del procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =  
  defaultMain  
    [testGroup "Propiedades del TAD cola"  
     testGroup "Propiedades del TAD cola"  
       [testProperty "P1" prop_primero_inserta_vacia,  
        testProperty "P2" prop_primero_inserta_no_vacia,  
        testProperty "P3" prop_resto_inserta_vacia,  
        testProperty "P4" prop_resto_inserta_en_no_vacia,  
        testProperty "P5" prop_vacia_es_vacia,  
        testProperty "P6" prop_inserta_no_es_vacia,  
        testProperty "P7" prop_valida_vacia,  
        testProperty "P8" prop_valida_inserta,  
        testProperty "P9" prop_valida_resto]]
```

Comprobación de las propiedades de las colas

```
ghci> compruebaPropiedades
Propiedades del TAD cola
P1: [OK, passed 100 tests]
P2: [OK, passed 100 tests]
P3: [OK, passed 100 tests]
P4: [OK, passed 100 tests]
P5: [OK, passed 100 tests]
P6: [OK, passed 100 tests]
P7: [OK, passed 100 tests]
P8: [OK, passed 100 tests]
P9: [OK, passed 100 tests]
```

	Properties	Total
Passed	9	9
Failed	0	0
Total	9	9