# Tema 19: El TAD de los árboles binarios de búsqueda

## Informática (2017–18)

José A. Alonso Jiménez

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

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda
   - Signatura del TAD de los árboles binarios de búsqueda
   - Propiedades del TAD de los árboles binarios de búsqueda

2. Implementación del TAD de los árboles binarios de búsqueda
   - Los ABB como tipo de dato algebraico

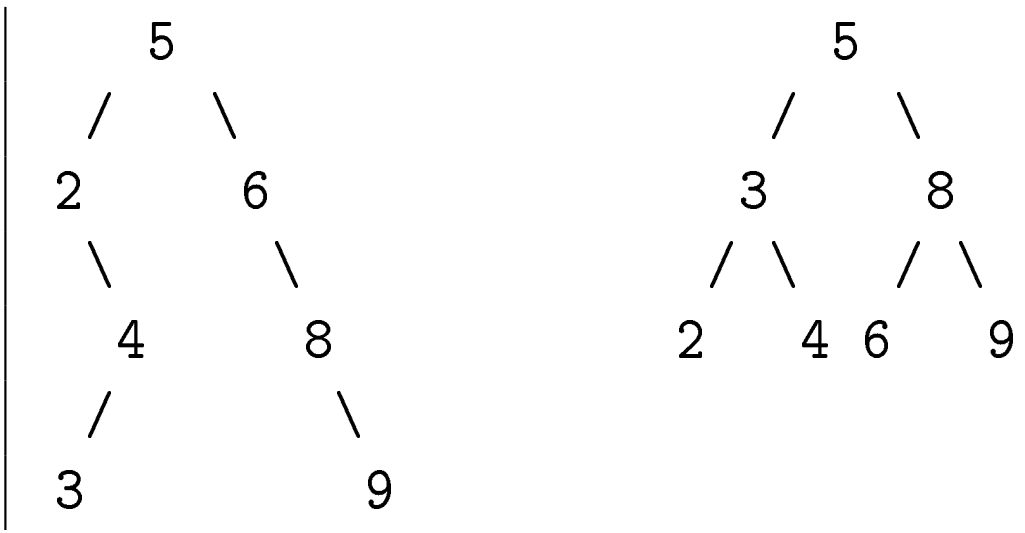
3. Comprobación de la implementación con QuickCheck
   - Librerías auxiliares
   - Generador de árboles binarios de búsqueda
   - Especificación de las propiedades de los árboles de búsqueda
   - Comprobación de las propiedades

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda
   - Signatura del TAD de los árboles binarios de búsqueda
   - Propiedades del TAD de los árboles binarios de búsqueda
2. Implementación del TAD de los árboles binarios de búsqueda
3. Comprobación de la implementación con QuickCheck

# Descripción de los árboles binarios de búsqueda

- Un árbol binario de búsqueda (ABB) es un árbol binario tal que el valor de cada nodo es mayor que los valores de su subárbol izquierdo y es menor que los valores de su subárbol derecho y, además, ambos subárboles son árboles binarios de búsqueda.
- Por ejemplo, al almacenar los valores de [2,3,4,5,6,8,9] en un ABB se puede obtener los siguientes ABB:




- El objetivo principal de los ABB es reducir el tiempo de acceso a los valores.

# Signatura del TAD de los árboles binarios de búsqueda

Signatura:

```
vacio     :: ABB
inserta   :: (Ord a,Show a) => a -> ABB a -> ABB a
elimina   :: (Ord a,Show a) => a -> ABB a -> ABB a
crea      :: (Ord a,Show a) => [a] -> ABB a
menor     :: Ord a => ABB a -> a
elementos :: (Ord a,Show a) => ABB a -> [a]
pertenece :: (Ord a,Show a) => a -> ABB a -> Bool
valido    :: (Ord a,Show a) => ABB a -> Bool
```

# Signatura del TAD de los árboles binarios de búsqueda

Descripción de las operaciones:

- `vacio` es el ABB vacío.
- `(pertenece v a)` se verifica si v es el valor de algún nodo del ABB a.
- `(inserta v a)` es el árbol obtenido añadiendo el valor v al ABB a, si no es uno de sus valores.
- `(crea vs)` es el ABB cuyos valores son vs.
- `(elementos a)` es la lista de los valores de los nodos del ABB en el recorrido inorden.
- `(elimina v a)` es el ABB obtenido eliminando el valor v del ABB a.
- `(menor a)` es el mínimo valor del ABB a.
- `(valido a)` se verifica si a es un ABB correcto.

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda
   - Signatura del TAD de los árboles binarios de búsqueda
   - Propiedades del TAD de los árboles binarios de búsqueda
2. Implementación del TAD de los árboles binarios de búsqueda
3. Comprobación de la implementación con QuickCheck

# Propiedades del TAD de los árboles binarios de búsqueda

1. `valido vacio`
2. `valido (inserta v a)`
3. `inserta x a /= vacio`
4. `pertenece x (inserta x a)`
5. `not (pertenece x vacio)`
6. `pertenece y (inserta x a)`
   `== (x == y) || pertenece y a`
7. `valido (elimina v a)`
8. `elimina x (inserta x a) == elimina x a`
9. `valido (crea xs)`
10. `elementos (crea xs) == sort (nub xs)`
11. `pertenece v a == elem v (elementos a)`
12. $\forall$ `x` $\in$ `elementos a (menor a <= x)`

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda

2. Implementación del TAD de los árboles binarios de búsqueda
   Los ABB como tipo de dato algebraico

3. Comprobación de la implementación con QuickCheck

# Los ABB como tipo de dato algebraico

Cabecera del módulo:

```
module ArbolBin
    (ABB,
     vacio,      -- ABB
     inserta,    -- (Ord a,Show a) => a -> ABB a -> ABB a
     elimina,    -- (Ord a,Show a) => a -> ABB a -> ABB a
     crea,       -- (Ord a,Show a) => [a] -> ABB a
     crea',      -- (Ord a,Show a) => [a] -> ABB a
     menor,      -- Ord a => ABB a -> a
     elementos,  -- (Ord a,Show a) => ABB a -> [a]
     pertenece,  -- (Ord a,Show a) => a -> ABB a -> Bool
     valido      -- (Ord a,Show a) => ABB a -> Bool
    ) where
```

# Los ABB como tipo de dato algebraico

- Los ABB como tipo de dato algebraico.

```
data Ord a => ABB a = Vacio
                    | Nodo a (ABB a) (ABB a)
                    deriving (Show, Eq)
```

- Procedimiento de escritura de árboles binarios de búsqueda.

```
instance (Show a, Ord a) => Show (ABB a) where
    show Vacio        = " _"
    show (Nodo x i d) =
       " (" ++ show x ++ show i ++ show d ++ ")"
```

# Los ABB como tipo de dato algebraico

- abb1 y abb2 son árboles de búsqueda binarios.

```
ghci> abb1
 (5 (2 - (4 (3 - -) -)) (6 - (8 - (9 - -))))
ghci> abb2
 (5 (2 - (4 (3 - -) -)) (8 (6 - (7 - -)) (10 (9 - -) (11
```

```
abb1, abb2 :: ABB Int
abb1 = crea (reverse [5,2,6,4,8,3,9])
abb2 = foldr inserta vacio
            (reverse [5,2,4,3,8,6,7,10,9,11])
```

# Los ABB como tipo de dato algebraico

- `vacio` es el ABB vacío.

```
vacio :: ABB a
vacio = Vacio
```

- `(pertenece v a)` se verifica si `v` es el valor de algún nodo del ABB a. Por ejemplo,

```
pertenece 3 abb1  ⇝  True
pertenece 7 abb1  ⇝  False
```

```
pertenece :: (Ord a,Show a) => a -> ABB a -> Bool
pertenece v' Vacio                 = False
pertenece v' (Nodo v i d) | v' == v = True
                          | v' <  v = pertenece v' i
                          | v' >  v = pertenece v' d
```

# Los ABB como tipo de dato algebraico

- `vacio` es el ABB vacío.

```
vacio :: ABB a
vacio = Vacio
```

- `(pertenece v a)` se verifica si `v` es el valor de algún nodo del ABB `a`. Por ejemplo,

```
pertenece 3 abb1  ⇝  True
pertenece 7 abb1  ⇝  False
```

```
pertenece :: (Ord a,Show a) => a -> ABB a -> Bool
pertenece v' Vacio                 = False
pertenece v' (Nodo v i d) | v' == v = True
                          | v' <  v = pertenece v' i
                          | v' >  v = pertenece v' d
```

# Los ABB como tipo de dato algebraico

- `vacio` es el ABB vacío.

```
vacio :: ABB a
vacio = Vacio
```

- `(pertenece v a)` se verifica si `v` es el valor de algún nodo del ABB `a`. Por ejemplo,

```
pertenece 3 abb1  ⇝  True
pertenece 7 abb1  ⇝  False
```

```
pertenece :: (Ord a,Show a) => a -> ABB a -> Bool
pertenece v' Vacio                  = False
pertenece v' (Nodo v i d) | v' == v = True
                          | v' <  v = pertenece v' i
                          | v' >  v = pertenece v' d
```

# Los ABB como tipo de dato algebraico

- `(inserta v a)` es el árbol obtenido añadiendo el valor `v` al ABB `a`, si no es uno de sus valores. Por ejemplo,

```
ghci> inserta 7 abb1
 (5 (2 - (4 (3 - -) -)) (6 - (8 (7 - -) (9 - -))))
```

```
inserta :: (Ord a,Show a) => a -> ABB a -> ABB a
inserta v' Vacio = Nodo v' Vacio Vacio
inserta v' (Nodo v i d)
    | v' == v   = Nodo v i d
    | v' <  v   = Nodo v (inserta v' i) d
    | otherwise = Nodo v i (inserta v' d)
```

# Los ABB como tipo de dato algebraico

- `(inserta v a)` es el árbol obtenido añadiendo el valor `v` al ABB a, si no es uno de sus valores. Por ejemplo,

```
ghci> inserta 7 abb1
 (5 (2 - (4 (3 - -) -)) (6 - (8 (7 - -) (9 - -))))
```

```
inserta :: (Ord a,Show a) => a -> ABB a -> ABB a
inserta v' Vacio = Nodo v' Vacio Vacio
inserta v' (Nodo v i d)
    | v' == v   = Nodo v i d
    | v' <  v   = Nodo v (inserta v' i) d
    | otherwise = Nodo v i (inserta v' d)
```

# Los ABB como tipo de dato algebraico

- `(crea vs)` es el ABB cuyos valores son `vs`. Por ejemplo

```
ghci> crea [3,7,2]
 (2 - (7 (3 - -) -))
```

```
crea :: (Ord a,Show a) => [a] -> ABB a
crea = foldr inserta Vacio
```

# Los ABB como tipo de dato algebraico

- `(crea vs)` es el ABB cuyos valores son `vs`. Por ejemplo

```
ghci> crea [3,7,2]
 (2 - (7 (3 - -) -))
```

```
crea :: (Ord a,Show a) => [a] -> ABB a
crea = foldr inserta Vacio
```

# Los ABB como tipo de dato algebraico

- `(crea' vs)` es el ABB de menor profundidad cuyos valores son los de la lista ordenada `vs`. Por ejemplo,

```
ghci> crea' [2,3,7]
 (3 (2 - -) (7 - -))
```

```
crea' :: (Ord a,Show a) => [a] -> ABB a
crea' [] = Vacio
crea' vs = Nodo x (crea' l1) (crea' l2)
    where n      = length vs `div` 2
          l1     = take n vs
          (x:l2) = drop n vs
```

# Los ABB como tipo de dato algebraico

- `(crea' vs)` es el ABB de menor profundidad cuyos valores son los de la lista ordenada `vs`. Por ejemplo,

```
ghci> crea' [2,3,7]
 (3 (2 - -) (7 - -))
```

```
crea' :: (Ord a,Show a) => [a] -> ABB a
crea' [] = Vacio
crea' vs = Nodo x (crea' l1) (crea' l2)
    where n       = length vs `div` 2
          l1      = take n vs
          (x:l2) = drop n vs
```

# Los ABB como tipo de dato algebraico

- `(elementos a)` es la lista de los valores de los nodos del ABB a en el recorrido inorden. Por ejemplo,

```
elementos abb1  ⇝  [2,3,4,5,6,8,9]
elementos abb2  ⇝  [2,3,4,5,6,7,8,9,10,11]
```

```
elementos :: (Ord a,Show a) => ABB a -> [a]
elementos Vacio = []
elementos (Nodo v i d) =
   elementos i ++ [v] ++ elementos d
```

# Los ABB como tipo de dato algebraico

- `(elementos a)` es la lista de los valores de los nodos del ABB a en el recorrido inorden. Por ejemplo,

```
elementos abb1  ⇝  [2,3,4,5,6,8,9]
elementos abb2  ⇝  [2,3,4,5,6,7,8,9,10,11]
```

```
elementos :: (Ord a,Show a) => ABB a -> [a]
elementos Vacio = []
elementos (Nodo v i d) =
   elementos i ++ [v] ++ elementos d
```

# Los ABB como tipo de dato algebraico

- `(elimina v a)` el ABB obtenido eliminando el valor `v` del ABB `a`.

```
ghci> elimina 3 abb1
 (5 (2 - (4 - -)) (6 - (8 - (9 - -))))
ghci> elimina 2 abb1
 (5 (4 (3 - -) -) (6 - (8 - (9 - -))))
```

```
elimina :: (Ord a,Show a) => a -> ABB a -> ABB a
elimina v' Vacio = Vacio
elimina v' (Nodo v i Vacio) | v' == v = i
elimina v' (Nodo v Vacio d) | v' == v = d
elimina v' (Nodo v i d)
    | v' <  v = Nodo v (elimina v' i) d
    | v' >  v = Nodo v i (elimina v' d)
    | v' == v = Nodo k i (elimina k d)
                where k = menor d
```

# Los ABB como tipo de dato algebraico

- `(elimina v a)` el ABB obtenido eliminando el valor `v` del ABB `a`.

```
ghci> elimina 3 abb1
 (5 (2 - (4 - -)) (6 - (8 - (9 - -))))
ghci> elimina 2 abb1
 (5 (4 (3 - -) -) (6 - (8 - (9 - -))))
```

```
elimina :: (Ord a,Show a) => a -> ABB a -> ABB a
elimina v' Vacio = Vacio
elimina v' (Nodo v i Vacio) | v' == v = i
elimina v' (Nodo v Vacio d) | v' == v = d
elimina v' (Nodo v i d)
    | v' <  v = Nodo v (elimina v' i) d
    | v' >  v = Nodo v i (elimina v' d)
    | v' == v = Nodo k i (elimina k d)
                where k = menor d
```

# Los ABB como tipo de dato algebraico

- `(menor a)` es el mínimo valor del ABB a. Por ejemplo,

```
menor abb1  ⇝  2
```

```
menor :: Ord a => ABB a -> a
menor (Nodo v Vacio _) = v
menor (Nodo _ i     _) = menor i
```

- `(menorTodos v a)` se verifica si `v` es menor que todos los elementos del ABB a.

```
menorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
menorTodos v Vacio = True
menorTodos v a     = v < minimum (elementos a)
```

# Los ABB como tipo de dato algebraico

- `(menor a)` es el mínimo valor del ABB a. Por ejemplo,

```
menor abb1  ⇝  2
```

```
menor :: Ord a => ABB a -> a
menor (Nodo v Vacio _) = v
menor (Nodo _ i     _) = menor i
```

- `(menorTodos v a)` se verifica si v es menor que todos los elementos del ABB a.

```
menorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
menorTodos v Vacio = True
menorTodos v a     = v < minimum (elementos a)
```

# Los ABB como tipo de dato algebraico

- `(menor a)` es el mínimo valor del ABB a. Por ejemplo,

```
menor abb1  ⇝  2
```

```
menor :: Ord a => ABB a -> a
menor (Nodo v Vacio _) = v
menor (Nodo _ i     _) = menor i
```

- `(menorTodos v a)` se verifica si `v` es menor que todos los elementos del ABB a.

```
menorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
menorTodos v Vacio = True
menorTodos v a     = v < minimum (elementos a)
```

# Los ABB como tipo de dato algebraico

- `(mayorTodos v a)` se verifica si `v` es mayor que todos los elementos del ABB `a`.

```
mayorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
mayorTodos v Vacio = True
mayorTodos v a     = v > maximum (elementos a)
```

- `(valido a)` se verifica si `a` es un ABB correcto. Por ejemplo,

```
valido abb1 ⇝ True
```

```
valido :: (Ord a, Show a) => ABB a -> Bool
valido Vacio          = True
valido (Nodo v i d) = mayorTodos v i && menorTodos v d
                      && valido i && valido d
```

# Los ABB como tipo de dato algebraico

- `(mayorTodos v a)` se verifica si `v` es mayor que todos los elementos del ABB `a`.

```
mayorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
mayorTodos v Vacio = True
mayorTodos v a     = v > maximum (elementos a)
```

- `(valido a)` se verifica si `a` es un ABB correcto. Por ejemplo,

```
valido abb1 ⇝ True
```

```
valido :: (Ord a, Show a) => ABB a -> Bool
valido Vacio          = True
valido (Nodo v i d) = mayorTodos v i && menorTodos v d
                      && valido i && valido d
```

# Los ABB como tipo de dato algebraico

- `(mayorTodos v a)` se verifica si `v` es mayor que todos los elementos del ABB `a`.

```
mayorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
mayorTodos v Vacio = True
mayorTodos v a     = v > maximum (elementos a)
```

- `(valido a)` se verifica si `a` es un ABB correcto. Por ejemplo,

```
valido abb1 ⇝ True
```

```
valido :: (Ord a, Show a) => ABB a -> Bool
valido Vacio          = True
valido (Nodo v i d) = mayorTodos v i && menorTodos v d
                      && valido i && valido d
```

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda

2. Implementación del TAD de los árboles binarios de búsqueda

3. Comprobación de la implementación con QuickCheck
   - Librerías auxiliares
   - Generador de árboles binarios de búsqueda
   - Especificación de las propiedades de los árboles de búsqueda
   - Comprobación de las propiedades

# Librerías auxiliares

- Importación de la implementación de ABB.

```
import ArbolBin
```

- Importación de librerías auxiliares.

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

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda

2. Implementación del TAD de los árboles binarios de búsqueda

3. Comprobación de la implementación con QuickCheck
   - Librerías auxiliares
   - Generador de árboles binarios de búsqueda
   - Especificación de las propiedades de los árboles de búsqueda
   - Comprobación de las propiedades

# Generador de árboles binarios de búsqueda

- genABB es un generador de árboles binarios de búsqueda. Por ejemplo,

```
ghci> sample genABB
 -
(1 (-1 - -) -)
(1 - -)
(-1 (-3 - -) (1 - (4 - -)))
```

```
genABB :: Gen (ABB Int)
genABB = do xs <- listOf arbitrary
            return (foldr inserta vacio xs)

instance Arbitrary (ABB Int) where
    arbitrary = genABB
```

# Generador de árboles binarios de búsqueda

- Propiedad. Todo los elementos generados por genABB son árboles binarios de búsqueda.

```
prop_genABB_correcto :: ABB Int -> Bool
prop_genABB_correcto = valido
```

# Generador de árboles binarios de búsqueda

- Propiedad. Todo los elementos generados por genABB son árboles binarios de búsqueda.

```
prop_genABB_correcto :: ABB Int -> Bool
prop_genABB_correcto = valido
```

# Generador de árboles binarios de búsqueda

- `listaOrdenada` es un generador de listas ordenadas de números enteros. Por ejemplo,

```
ghci> sample listaOrdenada
[1]
[-2,-1,0]
```

```
listaOrdenada :: Gen [Int]
listaOrdenada =
    frequency [(1,return []),
               (4,do xs <- orderedList
                     n <- arbitrary
                     return (nub ((case xs of
                                     []  -> n
                                     x:_ -> n `min` x)
                                  :xs)))]
```

# Generador de árboles binarios de búsqueda

- `(ordenada xs)` se verifica si `xs` es una lista ordenada creciente. Por ejemplo,

```
ordenada [3,5,9]  ⇝  True
ordenada [3,9,5]  ⇝  False
```

```
ordenada :: [Int] -> Bool
ordenada xs = and [x<y | (x,y) <- zip xs (tail xs)]
```

- Propiedad. El generador `listaOrdenada` produce listas ordenadas.

```
prop_listaOrdenada_correcta :: [Int] -> Property
prop_listaOrdenada_correcta xs =
    forAll listaOrdenada ordenada
```

# Generador de árboles binarios de búsqueda

- `(ordenada xs)` se verifica si `xs` es una lista ordenada creciente. Por ejemplo,

```
ordenada [3,5,9]  ⇝  True
ordenada [3,9,5]  ⇝  False
```

```
ordenada :: [Int] -> Bool
ordenada xs = and [x<y | (x,y) <- zip xs (tail xs)]
```

- Propiedad. El generador `listaOrdenada` produce listas ordenadas.

```
prop_listaOrdenada_correcta :: [Int] -> Property
prop_listaOrdenada_correcta xs =
    forAll listaOrdenada ordenada
```

# Generador de árboles binarios de búsqueda

- `(ordenada xs)` se verifica si `xs` es una lista ordenada creciente. Por ejemplo,

```
ordenada [3,5,9]  ⇝  True
ordenada [3,9,5]  ⇝  False
```

```
ordenada :: [Int] -> Bool
ordenada xs = and [x<y | (x,y) <- zip xs (tail xs)]
```

- Propiedad. El generador `listaOrdenada` produce listas ordenadas.

```
prop_listaOrdenada_correcta :: [Int] -> Property
prop_listaOrdenada_correcta xs =
    forAll listaOrdenada ordenada
```

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda

2. Implementación del TAD de los árboles binarios de búsqueda

3. Comprobación de la implementación con QuickCheck
   - Librerías auxiliares
   - Generador de árboles binarios de búsqueda
   - Especificación de las propiedades de los árboles de búsqueda
   - Comprobación de las propiedades

# Especificación de las propiedades de los ABB

- `vacio` es un ABB.

```
prop_vacio_es_ABB :: Bool
prop_vacio_es_ABB =
    valido (vacio :: ABB Int)
```

- Si a es un ABB, entonces `(inserta v a)` también lo es.

```
prop_inserta_es_valida :: Int -> ABB Int -> Bool
prop_inserta_es_valida v a =
    valido (inserta v a)
```

# Especificación de las propiedades de los ABB

- `vacio` es un ABB.

```
prop_vacio_es_ABB :: Bool
prop_vacio_es_ABB =
    valido (vacio :: ABB Int)
```

- Si `a` es un ABB, entonces `(inserta v a)` también lo es.

```
prop_inserta_es_valida :: Int -> ABB Int -> Bool
prop_inserta_es_valida v a =
    valido (inserta v a)
```

# Especificación de las propiedades de los ABB

- `vacio` es un ABB.

```
prop_vacio_es_ABB :: Bool
prop_vacio_es_ABB =
    valido (vacio :: ABB Int)
```

- Si `a` es un ABB, entonces `(inserta v a)` también lo es.

```
prop_inserta_es_valida :: Int -> ABB Int -> Bool
prop_inserta_es_valida v a =
    valido (inserta v a)
```

# Especificación de las propiedades de los ABB

- El árbol que resulta de añadir un elemento a un ABB es no vacío.

```
prop_inserta_es_no_vacio :: Int -> ABB Int -> Bool
prop_inserta_es_no_vacio x a =
    inserta x a /= vacio
```

- Para todo `x` y `a`, `x` es un elemento de `(inserta x a)`.

```
prop_elemento_de_inserta :: Int -> ABB Int -> Bool
prop_elemento_de_inserta x a =
    pertenece x (inserta x a)
```

# Especificación de las propiedades de los ABB

- El árbol que resulta de añadir un elemento a un ABB es no vacío.

```
prop_inserta_es_no_vacio :: Int -> ABB Int -> Bool
prop_inserta_es_no_vacio x a =
    inserta x a /= vacio
```

- Para todo `x` y `a`, `x` es un elemento de `(inserta x a)`.

```
prop_elemento_de_inserta :: Int -> ABB Int -> Bool
prop_elemento_de_inserta x a =
    pertenece x (inserta x a)
```

# Especificación de las propiedades de los ABB

- El árbol que resulta de añadir un elemento a un ABB es no vacío.

```
prop_inserta_es_no_vacio :: Int -> ABB Int -> Bool
prop_inserta_es_no_vacio x a =
    inserta x a /= vacio
```

- Para todo `x` y `a`, `x` es un elemento de `(inserta x a)`.

```
prop_elemento_de_inserta :: Int -> ABB Int -> Bool
prop_elemento_de_inserta x a =
    pertenece x (inserta x a)
```

# Especificación de las propiedades de los ABB

- En un árbol vacio no hay ningún elemento.

```
prop_vacio_sin_elementos :: Int -> Bool
prop_vacio_sin_elementos x =
    not (pertenece x vacio)
```

- Los elementos de `(inserta x a)` son `x` y los elementos de `a`.

```
prop_elementos_de_inserta :: Int -> Int
                          -> ABB Int -> Bool
prop_elementos_de_inserta x y a =
    pertenece y (inserta x a)
    == (x == y) || pertenece y a
```

# Especificación de las propiedades de los ABB

- En un árbol vacio no hay ningún elemento.

```
prop_vacio_sin_elementos :: Int -> Bool
prop_vacio_sin_elementos x =
    not (pertenece x vacio)
```

- Los elementos de `(inserta x a)` son `x` y los elementos de `a`.

```
prop_elementos_de_inserta :: Int -> Int
                          -> ABB Int -> Bool
prop_elementos_de_inserta x y a =
    pertenece y (inserta x a)
    == (x == y) || pertenece y a
```

# Especificación de las propiedades de los ABB

- En un árbol vacio no hay ningún elemento.

```
prop_vacio_sin_elementos :: Int -> Bool
prop_vacio_sin_elementos x =
    not (pertenece x vacio)
```

- Los elementos de `(inserta x a)` son `x` y los elementos de `a`.

```
prop_elementos_de_inserta :: Int -> Int
                          -> ABB Int -> Bool
prop_elementos_de_inserta x y a =
    pertenece y (inserta x a)
    == (x == y) || pertenece y a
```

# Especificación de las propiedades de los ABB

- Si `a` es un ABB, entonces `(elimina v a)` también lo es.

```
prop_elimina_es_valida :: Int -> ABB Int -> Bool
prop_elimina_es_valida v a =
    valido (elimina v a)
```

- El resultado de eliminar el elemento `x` en `(inserta x a)` es `(elimina x a)`.

```
prop_elimina_agrega :: Int -> ABB Int -> Bool
prop_elimina_agrega x a =
    elimina (inserta x a) == elimina x a
```

# Especificación de las propiedades de los ABB

- Si `a` es un ABB, entonces `(elimina v a)` también lo es.

```
prop_elimina_es_valida :: Int -> ABB Int -> Bool
prop_elimina_es_valida v a =
    valido (elimina v a)
```

- El resultado de eliminar el elemento `x` en `(inserta x a)` es `(elimina x a)`.

```
prop_elimina_agrega :: Int -> ABB Int -> Bool
prop_elimina_agrega x a =
    elimina (inserta x a) == elimina x a
```

# Especificación de las propiedades de los ABB

- Si `a` es un ABB, entonces `(elimina v a)` también lo es.

```
prop_elimina_es_valida :: Int -> ABB Int -> Bool
prop_elimina_es_valida v a =
    valido (elimina v a)
```

- El resultado de eliminar el elemento `x` en `(inserta x a)` es `(elimina x a)`.

```
prop_elimina_agrega :: Int -> ABB Int -> Bool
prop_elimina_agrega x a =
    elimina (inserta x a) == elimina x a
```

# Especificación de las propiedades de los ABB

- `(crea xs)` es un ABB.

```
prop_crea_es_valida :: [Int] -> Bool
prop_crea_es_valida xs =
    valido (crea xs)
```

- Para todas las listas ordenadas `xs`, se tiene que `(crea' xs)` es un ABB.

```
prop_crea'_es_valida :: [Int] -> Property
prop_crea'_es_valida xs =
    forAll listaOrdenada (valido . crea')
```

# Especificación de las propiedades de los ABB

- `(crea xs)` es un ABB.

```
prop_crea_es_valida :: [Int] -> Bool
prop_crea_es_valida xs =
    valido (crea xs)
```

- Para todas las listas ordenadas `xs`, se tiene que `(crea' xs)` es un ABB.

```
prop_crea'_es_valida :: [Int] -> Property
prop_crea'_es_valida xs =
    forAll listaOrdenada (valido . crea')
```

# Especificación de las propiedades de los ABB

- `(crea xs)` es un ABB.

```
prop_crea_es_valida :: [Int] -> Bool
prop_crea_es_valida xs =
    valido (crea xs)
```

- Para todas las listas ordenadas `xs`, se tiene que `(crea' xs)` es un ABB.

```
prop_crea'_es_valida :: [Int] -> Property
prop_crea'_es_valida xs =
    forAll listaOrdenada (valido . crea')
```

# Especificación de las propiedades de los ABB

- `(elementos (crea xs))` es igual a la lista `xs` ordenada y sin repeticiones.

```
prop_elementos_crea :: [Int] -> Bool
prop_elementos_crea xs =
    elementos (crea xs) == sort (nub xs)
```

- Si `ys` es una lista ordenada sin repeticiones, entonces `(elementos (crea' ys))` es igual `ys`.

```
prop_elementos_crea' :: [Int] -> Bool
prop_elementos_crea' xs =
    elementos (crea' ys) == ys
    where ys = sort (nub xs)
```

# Especificación de las propiedades de los ABB

- `(elementos (crea xs))` es igual a la lista `xs` ordenada y sin repeticiones.

```
prop_elementos_crea :: [Int] -> Bool
prop_elementos_crea xs =
    elementos (crea xs) == sort (nub xs)
```

- Si `ys` es una lista ordenada sin repeticiones, entonces `(elementos (crea' ys))` es igual `ys`.

```
prop_elementos_crea' :: [Int] -> Bool
prop_elementos_crea' xs =
    elementos (crea' ys) == ys
    where ys = sort (nub xs)
```

# Especificación de las propiedades de los ABB

- `(elementos (crea xs))` es igual a la lista `xs` ordenada y sin repeticiones.

```
prop_elementos_crea :: [Int] -> Bool
prop_elementos_crea xs =
    elementos (crea xs) == sort (nub xs)
```

- Si `ys` es una lista ordenada sin repeticiones, entonces `(elementos (crea' ys))` es igual `ys`.

```
prop_elementos_crea' :: [Int] -> Bool
prop_elementos_crea' xs =
    elementos (crea' ys) == ys
    where ys = sort (nub xs)
```

# Especificación de las propiedades de los ABB

- Un elemento pertenece a `(elementos a)` syss es un valor de `a`.

```
prop_en_elementos :: Int -> ABB Int -> Bool
prop_en_elementos v a =
    pertenece v a == elem v (elementos a)
```

- `(menor a)` es menor o igual que todos los elementos de ABB `a`.

```
prop_menoresMinimo ::Int -> ABB Int -> Bool
prop_menoresMinimo v a =
    and [menor a <= v | v <- elementos a]
```

# Especificación de las propiedades de los ABB

- Un elemento pertenece a `(elementos a)` syss es un valor de a.

```
prop_en_elementos :: Int -> ABB Int -> Bool
prop_en_elementos v a =
    pertenece v a == elem v (elementos a)
```

- `(menor a)` es menor o igual que todos los elementos de ABB a.

```
prop_menoresMinimo ::Int -> ABB Int -> Bool
prop_menoresMinimo v a =
    and [menor a <= v | v <- elementos a]
```

# Especificación de las propiedades de los ABB

- Un elemento pertenece a `(elementos a)` syss es un valor de a.

```
prop_en_elementos :: Int -> ABB Int -> Bool
prop_en_elementos v a =
    pertenece v a == elem v (elementos a)
```

- `(menor a)` es menor o igual que todos los elementos de ABB a.

```
prop_menoresMinimo ::Int -> ABB Int -> Bool
prop_menoresMinimo v a =
    and [menor a <= v | v <- elementos a]
```

# Tema 19: El TAD de los árboles binarios de búsqueda

1. Especificación del TAD de los árboles binarios de búsqueda

2. Implementación del TAD de los árboles binarios de búsqueda

3. Comprobación de la implementación con QuickCheck
   - Librerías auxiliares
   - Generador de árboles binarios de búsqueda
   - Especificación de las propiedades de los árboles de búsqueda
   - Comprobación de las propiedades

# Definición del procedimiento de comprobación

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

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del tipo ABB"
          [testProperty "P1"  prop_listaOrdenada_correcta,
           testProperty "P2"  prop_orderedList_correcta,
           testProperty "P3"  prop_vacio_es_ABB,
           testProperty "P4"  prop_inserta_es_valida,
           testProperty "P5"  prop_inserta_es_no_vacio,
           testProperty "P6"  prop_elemento_de_inserta,
           testProperty "P7"  prop_vacio_sin_elementos,
           testProperty "P8"  prop_elementos_de_inserta,
           testProperty "P9"  prop_elimina_es_valida,
           testProperty "P10" prop_elimina_agrega,
           testProperty "P11" prop_crea_es_valida,
           testProperty "P12" prop_crea'_es_valida,
           testProperty "P13" prop_elementos_crea,
           testProperty "P14" prop_elementos_crea',
           testProperty "P15" prop_en_elementos,
           testProperty "P16" prop_menoresMinimo],
         testGroup "Corrección del generador"
          [testProperty "P18" prop_genABB_correcto]]
```

# Comprobación de las propiedades de los ABB

```
ghci> compruebaPropiedades
Propiedades del tipo ABB:
  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]
  P10: [OK, passed 100 tests]
  P11: [OK, passed 100 tests]
  P12: [OK, passed 100 tests]
  P13: [OK, passed 100 tests]
  P14: [OK, passed 100 tests]
  P15: [OK, passed 100 tests]
  P16: [OK, passed 100 tests]
Corrección del generador:
  P18: [OK, passed 100 tests]

         Properties   Total
 Passed  17           17
 Failed  0            0
 Total   17           17
```