Prácticas de laboratorio de LTP
(Parte II : Programación Funcional)

Práctica 5: Listas y **tipos algebraicos (II)**

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Jose Luis Pérez
jlperez@dsic.upv.es

Introducción

En esta segunda sesión de la práctica veremos como se pueden definir nuevos tipos, en el lenguaje Haskell, con sus valores asociados, empleando los llamados **tipos algebraicos**.

3. Tipos algebraicos

3.1 Enumeraciones y tipos renombrados

En un lenguaje funcional como Haskell se pueden definir nuevos tipos con sus valores asociados, empleando **enumeraciones** y También se pueden declarar **renombramientos de tipos** ya definidos, denominados tipos "sinónimos".

3.2 Tipos recursivos. Arboles

En Haskell podemos crear tipos cuyos campos de constructor sean el propio tipo. De esta forma, podemos crear **estructuras de datos recursivas**. Utilizaremos este tipo de estructuras para representar y manipular **árboles**.

Nota: En Poliformat se dispone de un enlace a un libro de Haskell en castellano. Y también el fichero **codigoEnPdf_P5** que podéis utilizar para copiar y pegar los ejemplos que se presentan durante la sesión.

3.1 Enumeraciones

Hasta ahora hemos utilizado varios tipos: **Bool**, **Int**, **Char**, etc. Para crear un nuevo tipo debemos usar la palabra clave **data** para definir un tipo.

Vamos a ver como está definido el tipo **Bool** en la librería estándar de Haskell:

```
data Bool = False | True
```

La palabra reservada data indica que vamos a definir un nuevo tipo de dato.

La **parte a la izquierda** del **=** denota el tipo, que es Bool.

La **parte a la derecha** son los **constructores de datos**, que especifican los diferentes valores que puede tener un tipo (**True** o **False**) separados por **|**.
(Tanto el nombre del tipo como el de los constructores de datos deben tener la primera letra en mayúsculas).

Veamos otro ejemplo:

```
data Color = Red | Green | Blue
```

se establece un nuevo tipo de datos Color. El tipo Color contiene solo tres valores denotados por los correspondientes constructores de datos constantes **Red**, **Green** y **Blue**.

3.1 Enumeraciones

Ahora vamos a pensar en como definiríamos una figura en Haskell. La mejor solución mejor consiste en crear nuestro propio tipo (Shape) que represente una figura.

Consideraremos que una figura solo puede ser un **círculo** o un **rectángulo**:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

El constructor **Circle** tiene 3 campos

El constructor **Rectangle** tiene 4 campos

Los constructores de datos (**Circle** y **Rectangle**) son en realidad funciones que devuelven un valor del tipo para el que fueron definidos. Vamos a ver la declaración de tipo de estos dos constructores de datos.

```
Prelude> :t Circle
Circle :: Float -> Float -> Float -> Shape

Prelude> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Podemos hacer una función que tome una figura y devuelva su **superficie**:

```
surface :: Shape -> Float
surface (Circle x y r) = pr * r ^ 2
surface (Rectangle x y b h) = 2 * b + 2 * h
```

3.1 Tipos renombrados

Los **tipos renombrados** (o tipos **sinónimos**) simplemente dan a algún tipo un nombre diferente, de forma que obtenga algún significado para alguien que está leyendo nuestro código o documentación.

En la práctica 4 mencionamos que los tipos **[Char]** y **String** eran equivalentes. Aquí tienes como define la librería estándar **String** como **tipo renombrado** de **[Char]**:

```
type String = [Char]
```

Veamos otro ejemplo:

```
module Figure where
  type Xpos = Float
  type Ypos = Float
  type Pos  = (Xpos,Ypos)
  type Radius = Float
  type Base = Float
  type Height = Float
  data Shape = Circle Pos Radius | Rectangle Pos Base Height  deriving Show
  surface (Circle  position  r)  = pi * r ^ 2
  surface (Rectangle position b h) = 2 * b + 2 * h
```

```
Prelude> :l Figure
*Figure> circle1 =Circle  (3,4)  5
*Figure > circle1
Circle (3.0,4.0) 5.0
*Figure > rectangle1 =Rectangle  (3,4)  6  7
*Figure > rectangle1
 Rectangle (3.0,4.0) 6.0 7.0
```

3.1 Tipos renombrados

Para el ejercicio que se plantea en esta sección, se han de definir, previamente, los siguientes tipos renombrados:

```
module Biblio where
  type Person = String
  type Book =  String
  type Database = [(Person,Book)]
```

El tipo Database define una base de datos de una biblioteca como una lista de pares (Person,Book) donde Person es el nombre de la persona que tiene en prestamo el libro Book. Un ejemplo de base de datos es:

```
  exampleBase :: Database
  exampleBase = [("Alicia","El nombre de la rosa"),("Juan",
    "La hija del canibal"),("Pepe","Odesa"),("Alicia","La ciudad de las bestias")]
```

A partir de esta base de datos ejemplo se pueden definir funciones para obtener los libros que tiene en préstamo una persona dada, **obtain**, para realizar un préstamo, **borrow**, y para realizar una devolución, **return'**.

3.1 Tipos renombrados

Por ejemplo, la función **obtain** se puede definir así:

```
module Biblio where
  ...
  obtain :: Database -> Person -> [Book]
  obtain dBase thisPerson = [book | (person,book) <- dBase, person==thisPerson]
```

que significa que la función devuelve la lista de todos los libros tales que hay un par (person,book) en la base de datos y person es igual a la persona cuyos libros se está buscando.

Este es un ejemplo de uso de la función **obtain**:

```
*Biblio> obtain exampleBase "Alicia"
["El nombre de la rosa","La ciudad de las bestias"]
```

Ejercicio 11: Crear un módulo (**Biblio**) con el anterior código y completar el programa con las definiciones para las funciones **borrow** y **return'**:

```
borrow :: Database -> Book -> Person -> Database
return'  :: Database -> (Person,Book) -> Database
```

3.2 Tipos recursivos. Árboles:

Árboles que almacenan valores en las hojas (**TreeInt, Tree a**)

Podemos crear tipos algebraicos cuyos campos de constructor sean el propio tipo. De esta forma, podemos crear **estructuras de datos recursivas**. Y con ellas podemos representar **árboles**:

Consideraremos dos clases de árboles en los siguientes apartados. La primera de ellas **almacena los valores en la hojas**:

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

establece un nuevo tipo de datos TreeInt que consta de un número infinito de valores definidos recursivamente con la ayuda de los símbolos constructores de datos Leaf (unario) y Branch (binario), que toman como argumentos un número entero y dos árboles TreeInt, respectivamente.

Nada impide definir tipos genéricos empleando estos recursos expresivos. El tipo:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

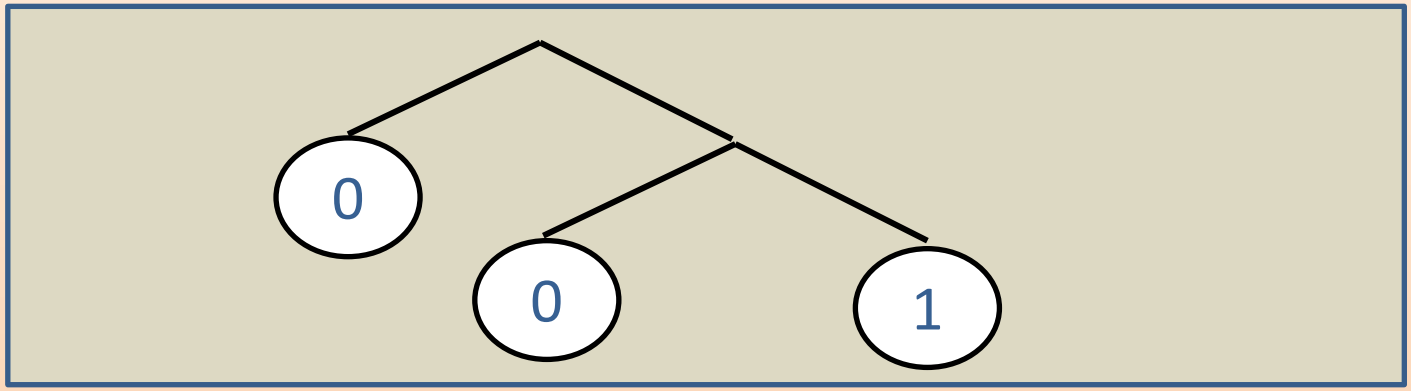
3.2 Tipos recursivos. Árboles:

Árboles que almacenan valores en las hojas (**TreeInt, Tree a**)

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

Ejemplo de un valor TreeInt sería:

```
Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))
```



Veamos un ejemplo de función recursiva aplicada sobre (Tree a), que calcula el número hojas del árbol:

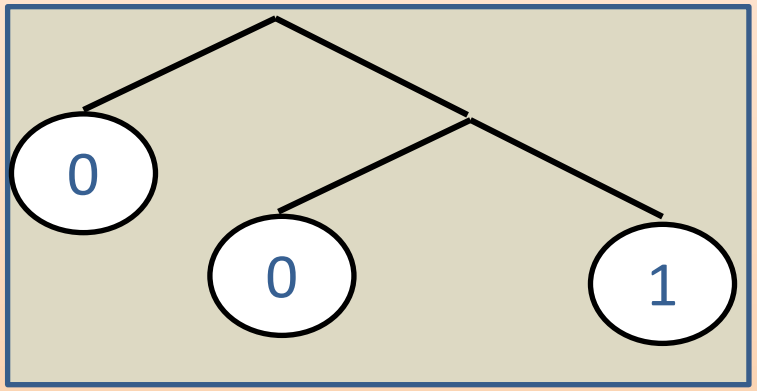
```
module  Arboles where
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  numleaves :: Tree a -> Int
  numleaves (Leaf x) = 1
  numleaves (Branch a b) = numleaves a + numleaves b
```

3.2 Tipos recursivos. Árboles: Ejercicios (Tree a)

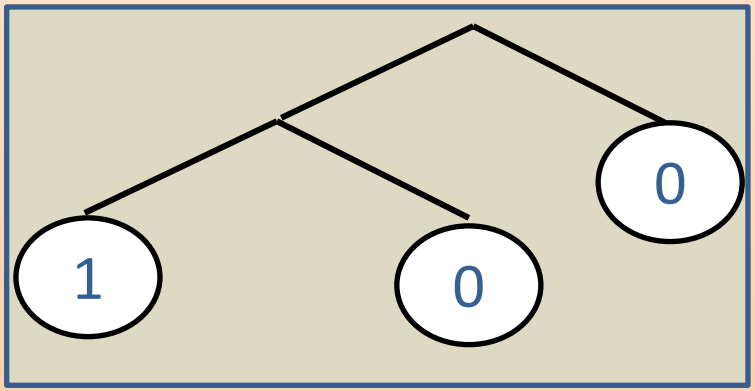
Ejercicio 12: Definir una función que obtenga el árbol simétrico al que se le pasa como parámetro:

```
symmetric :: Tree a -> Tree a
```

Dado: Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))



Devuelve: Branch (Branch (Leaf 1) (Leaf 0)) (Leaf 0)



Ejercicio 13: Definir las funciones

```
listToTree :: [a] -> Tree a
treeToList :: Tree a -> [a]
```

la primera de las cuales convierte una lista no vacía en un árbol, realizando la segunda de ellas la transformación contraria.

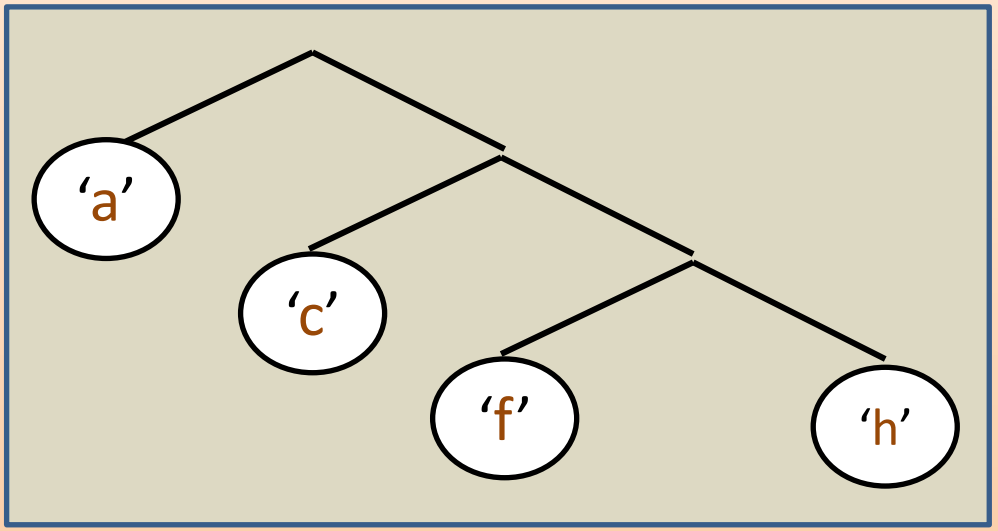
3.2 Tipos recursivos. Árboles: Ejercicios (Tree a)

Ejercicio 13:

listToTree :: [a] -> Tree a

Dado: [‘a’,’c’,’f’,’h’] o “acfd”

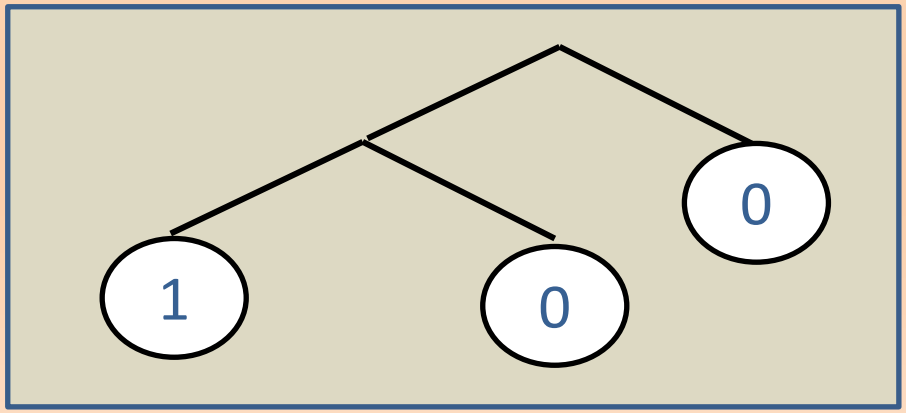
Devuelve: Branch (Leaf ‘a’) (Branch (Leaf ‘c’) (Branch (Leaf ‘f’) (Leaf ‘h’)))



treeToList :: Tree a -> [a]

Dado:

Branch (Branch (Leaf 1) (Leaf 0)) (Leaf 0)



Devuelve: [1,0,0]

Árboles que almacenan valores en los nodos (**BinTreeInt**)

Considérese ahora la siguiente definición alternativa, **BinTreeInt**, para el tipo de datos de los árboles binarios de enteros:

```
data BinTreeInt = Void | Node Int BinTreeInt BinTreeInt
```

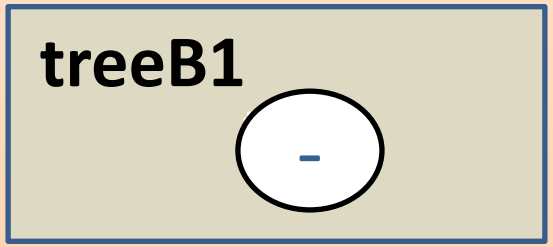
en el que los valores enteros se almacenan en los nodos (denotados por el constructor Node) y donde las hojas son subárboles que tienen tan solo raiz (denotados por el símbolo constructor de datos Void).

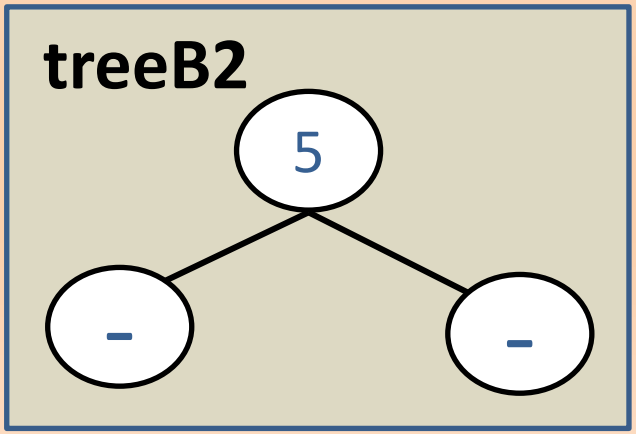
A continuación, se muestran algunos ejemplos de árboles binarios de enteros:

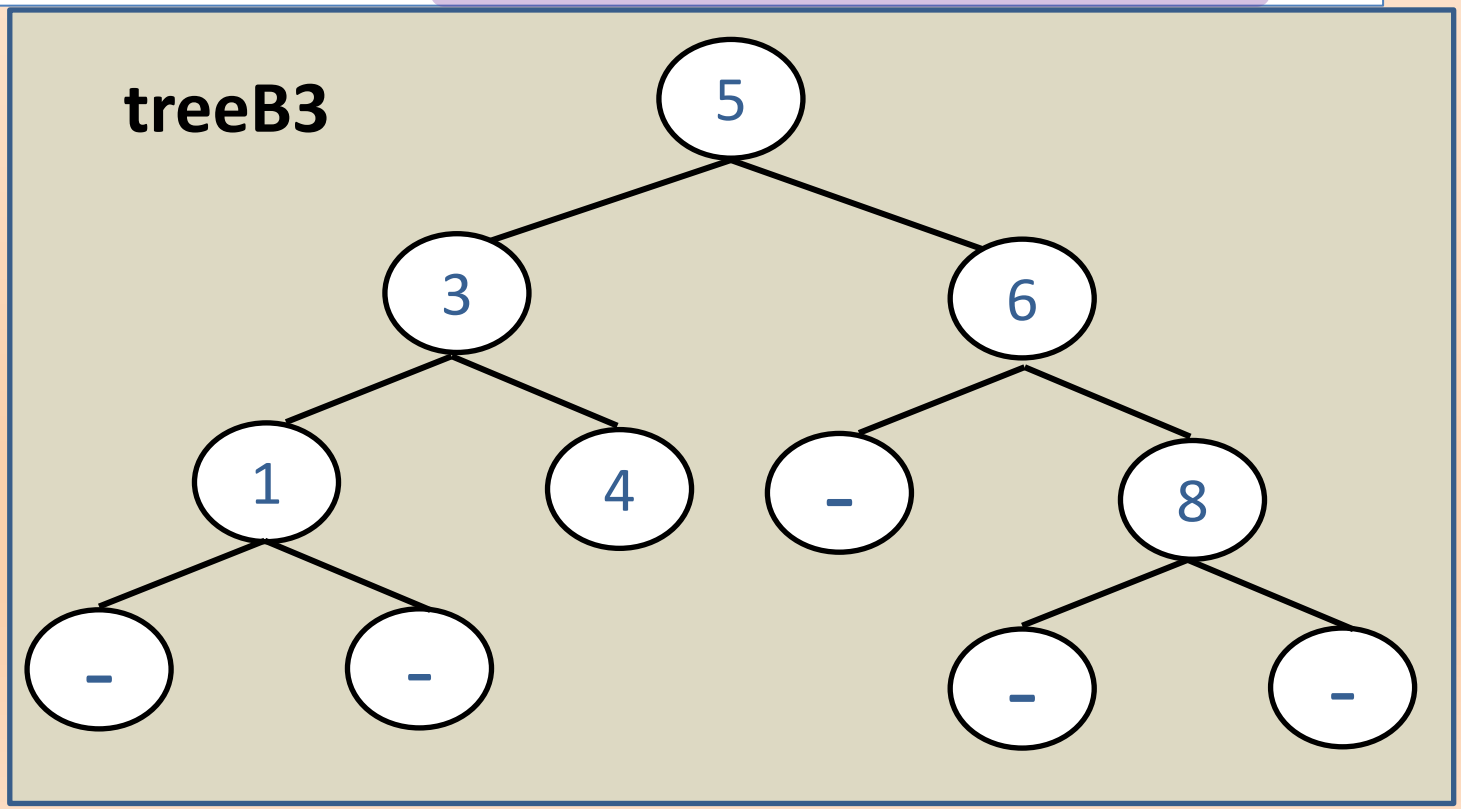
```
module  Arboles where
  ...
  data BinTreeInt = Void | Node Int  BinTreeInt  BinTreeInt
  treeB1 = Void
  treeB2 = (Node 5 Void Void)
  treeB3 = (Node 5 (Node 3 (Node 1 Void Void) (Node 4 Void Void))
                                              (Node 6 Void (Node 8 Void Void)))
```

Árboles que almacenan valores en los nodos (**BinTreeInt**)

```
module  Arboles where
...
  data BinTreeInt = Void | Node Int  BinTreeInt  BinTreeInt deriving Show
  treeB1 = Void
  treeB2 = (Node 5 Void Void)
  treeB3 = (Node 5 (Node 3 (Node 1 Void Void) (Node 4 Void Void))
                   (Node 6 Void (Node 8 Void Void)))
```







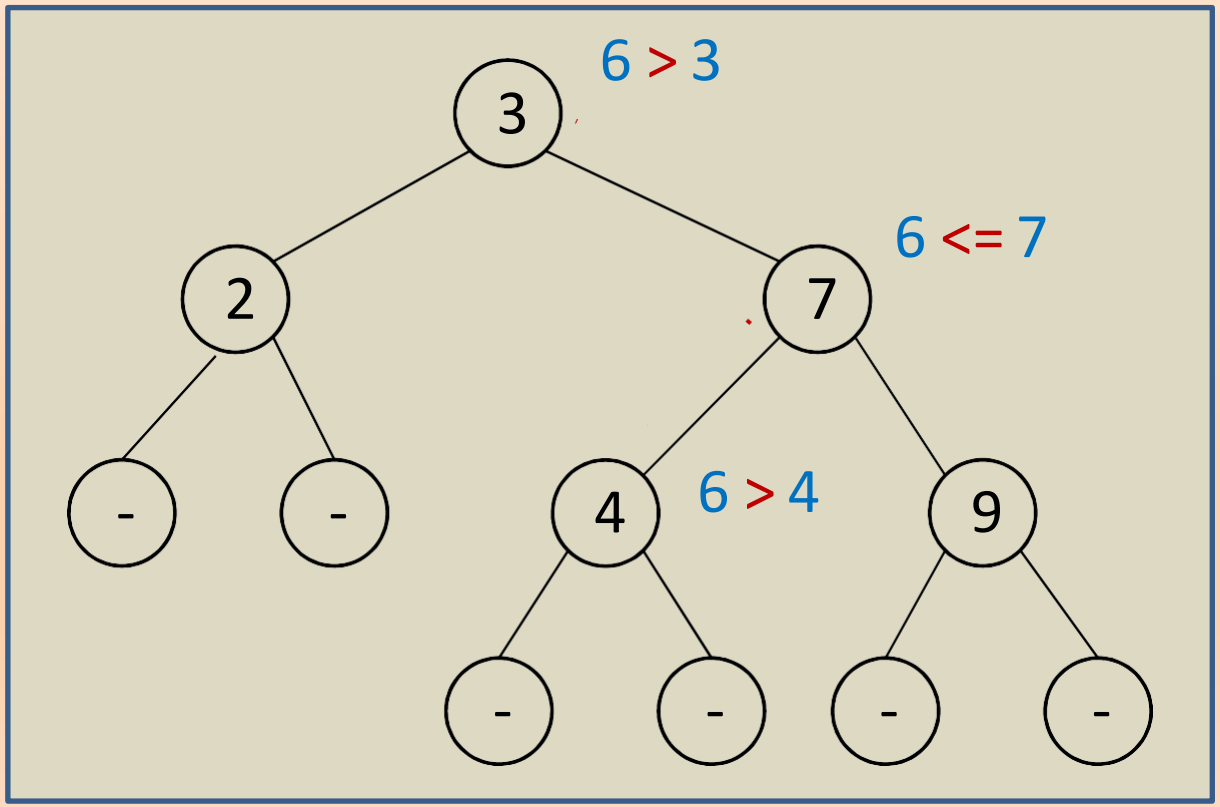
3.2 Tipos recursivos. Árboles: Ejercicios (BinTreeInt)

Ejercicio 14: Definir una función:

```
insTree :: Int -> BinTreeInt -> BinTreeInt
```

para insertar un valor entero en su lugar en un árbol binario ordenado.

```
insTree  6  (Node 3 (Node 2 Void Void) (Node 7 (Node 4 Void Void)
                                               (Node 9 Void Void)))
```



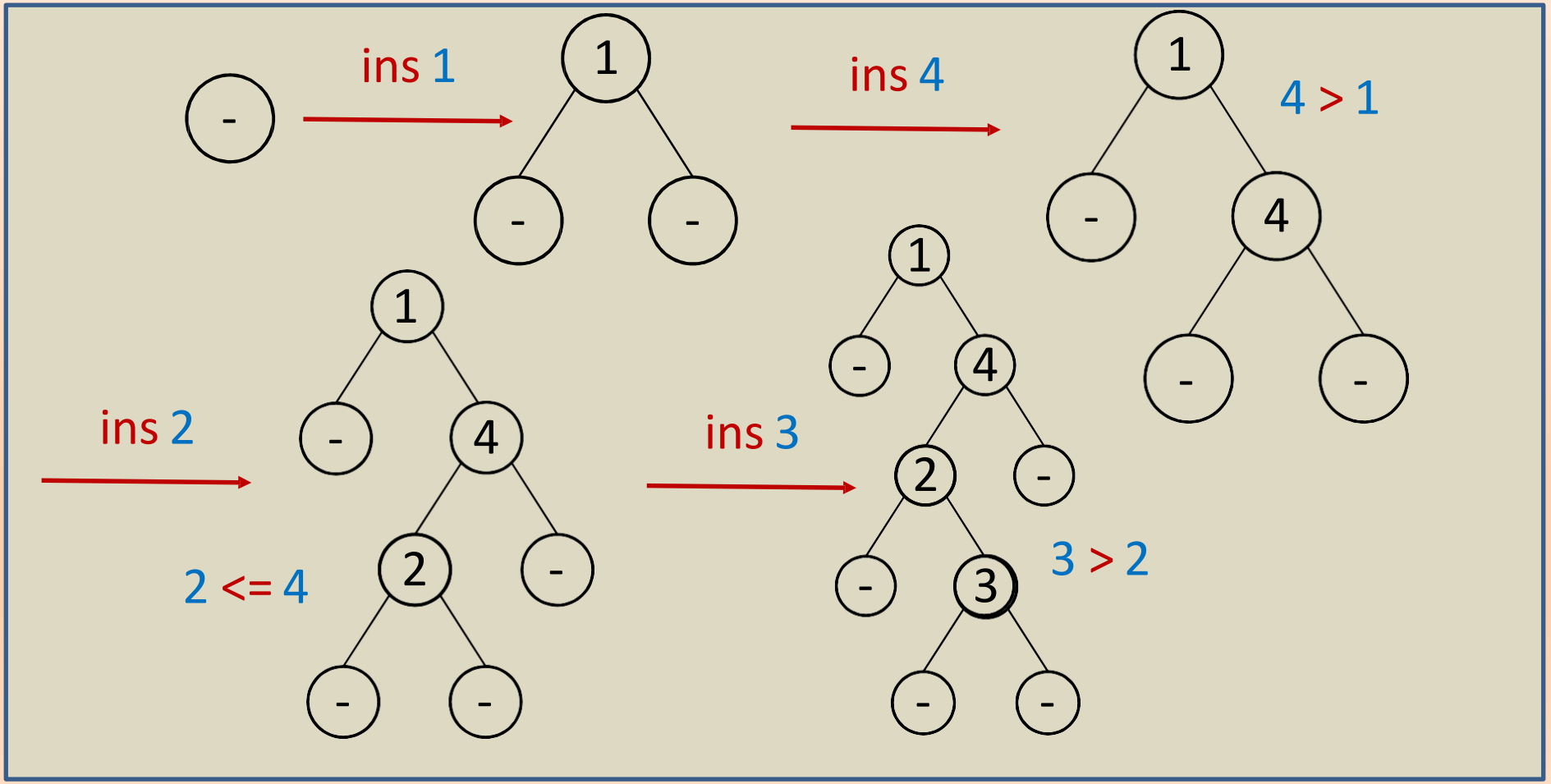
3.2 Tipos recursivos. Árboles: Ejercicios (BinTreeInt)

Ejercicio 15: Dada una lista no ordenada de enteros, definir una función:

```
creaTree :: [Int] -> BinTreeInt
```

que construya un árbol binario ordenado a partir de la misma.

```
creaTree [3, 2, 4, 1]
```



3.2 Tipos recursivos. Árboles: Ejercicios (BinTreeInt)

Ejercicio 16: Definir una función:

```
treeElem :: Int -> BinTreeInt -> Bool
```

que determine “de forma eficiente” si un valor entero pertenece o no a un árbol binario ordenado.

```
treeElem  6   (Node 3 (Node 2 Void Void) (Node 7 (Node 4 Void Void)
                                                 (Node 9 Void Void)))
```