Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza
1542

# Práctica 6

**Tecnología de Programación**
Grado en Ingeniería Informática

## Objetivos

- Comprender la definición y manejo de tipos y estructuras de datos en Haskell.
- Introducir los mecanismos de las clases y la programación genérica.

**Tarea**
En esta práctica, vas a programar en Haskell una serie de funciones que permitan trabajar con árboles binarios de búsqueda (`https://en.wikipedia.org/wiki/Binary_tree`).
Diseña en implementa en Haskell las funciones que se piden a lo largo de la práctica.

## 1. Árboles binarios

Define en Haskell los tipos necesarios para implementar una estructura de datos que represente un árbol binario genérico pero homogéneo (es decir, todos los datos almacenados en el árbol son del mismo tipo, pero éste puede ser entero, real, cadena de caracteres, o incluso cualquier tipo de datos definido por el usuario).

Para construir un árbol de forma transparente, se deberán implementar las siguientes funciones:

- **`empty`** – Esta función devuelve un árbol vacío, sin ningún elemento.
- **`leaf x`** – Esta función devuelve un árbol que consta de una única hoja que contiene el elemento x.
- **`tree x lc rc`** – Esta función devuelve un árbol que contiene en la raíz el elemento x, con hijo izquierdo `lc` e hijo derecho `rc`. Tanto el hijo izquierdo como el hijo derecho son sub–árboles, no elementos del tipo de datos genérico.
- **`size t`** – Esta función devuelve el número de elementos del árbol.

También deberás permitir que los árboles puedan ser visualizados por pantalla (con la alineación y formato que estimes oportuno, pero de forma que quede claro el contenido del árbol –los datos– y su distribución y estructura) utilizando la función de Haskell `print`. Para ello deberás hacer que tu árbol binario instancie la clase Show, por lo que necesita implementar la función `show`.

Implementa toda la funcionalidad en un módulo independiente llamado **BinaryTree.hs**. Para probar tu módulo, puedes o bien utilizar el intérprete de Haskell `ghci` o bien hacer un programa principal. Prueba que el árbol es realmente genérico, es decir, prueba que el árbol funciona con diferentes tipos de datos.

(continúa en la siguiente página...)

Mediante las funciones que has definido se puede construir y visualizar un árbol de forma manual, con código como el siguiente:

```
testTree = tree 1 (tree 2 (leaf 3) (leaf 4)) (tree 5 empty (leaf 6))
main = print testTree
```

```
1
|- 2
   |- 3
   |- 4
|- 5
   |- <>
   |- 6
```

O bien:

```
testTree = tree "R" (tree "HI" (leaf "NII") (leaf "NID"))
                    (tree "HD" (leaf "NDI") (leaf "NDD"))
main = print testTree
```

```
"R"
  |-"HI"
    |- "NII"
    |- "NID"
  |- "HD"
    |- "NDI"
    |- "NDD"
```

## 2. Construcción de árboles binarios de búsqueda (BST)

Con las funciones disponibles hasta el momento en tu módulo el proceso para construir un árbol es pesado y no garantiza que el árbol esté ordenado como requiere un árbol binario de búsqueda.

Implementa una función para añadir elementos a un árbol, de tal forma que se inserten en la posición correspondiente según el criterio de ordenación para el árbol:

- **`add t x`** – Añade el elemento x al árbol t, devolviendo el árbol binario de búsqueda resultante. Considera que el árbol binario de búsqueda puede contener valores repetidos, pero el valor en cada nodo siempre será mayor que todos los valores de los elementos del subárbol de la izquierda, y menor o igual que todos los elementos del subárbol de la derecha. Para nuestro caso, el criterio de ordenación se define simplemente mediante el uso de los operadores `'<'` y `'>'` (o `'<='` y `'>='`).

A partir de ella, implementa una función que construya un árbol binario de búsqueda a partir de los elementos de una lista:

- **`build xs`** – Construye un árbol binario de búsqueda, comenzando con un árbol vacío e insertando sucesivamente los elementos de la lista xs.

(continúa en la siguiente página...)

Ahora puedes construir un árbol de la siguiente forma:

```
build [3, 2, 2, 5, 1, 4, 4]

3
|- 2
   |- 1
   |- 2
|-5
   |- 4
      |- <>
      |- 4
   |- <>
```

Para poder implementar estas funciones, dado que el árbol binario de búsqueda es genérico, necesitarás asegurar que el tipo de dato de x tenga definidos los operadores de comparación. Esto se consigue añadiendo una restricción para que x sea instancia de la clase **`Ord`** de Haskell.

## 3. Arboles binarios de búsqueda equilibrados

El proceso anterior genera un árbol que, aunque siendo un árbol binario de búsqueda correcto, puede presentar una distribución y altura que dependen del orden en que se inserten los elementos:

```
build [3, 2, 5, 1, 4]
  3
  |- 2
     |- 1
     |- <>
  |- 5
     |- 4
     |- <>

build [4, 3, 2, 5, 1]
  4
  |- 3
     |- 2
        |- 1
        |- <>
     |- <>
  |- 5
```

El caso peor (patológico) aparecería al utilizar para construir el árbol una lista ordenada, que daría como resultado un árbol degenerado:

```
build [1..6]
  1
  |- <>
  |- 2
     |- <>
     |- 3
        |- <>
        |- 4
           |- <>
           |- 5
              |- <>
              |- 6
```

Implementa una función que construya un árbol equilibrado a partir de los elementos de una lista:

- **`buildBalanced xs`** – Construye un árbol equilibrado, ordenando la lista y dividiéndola en dos por la mediana.

```
buildBalanced [1..6]

  4
  |- 2
     |- 1
     |- 3
  |- 6
     |- 5
     |- <>
```

Para la implementación puedes buscar información de funciones de Haskell como **sort** o **splitAt**.

## 4. Recorrido de árboles binarios

Como ya sabes, hay tres tipos de recorridos en profundidad que se pueden utilizar sobre árboles binarios: pre–orden, post–orden e in–orden.

Implementa tres funciones (**`preorder t`**, **`postorder t`**, **`inorder t`**) que, dado un árbol **t**, devuelva una lista con todos los elementos del árbol en el orden de recorrido correspondiente. Comprueba que las tres funciones funcionan correctamente.

A partir de todas las funciones que ya tienes definidas, implementa una función que equilibre un árbol:

- **`balance t`** – Construye un árbol equilibrado (es decir, de altura mínima) a partir de otro cualquiera.

```
names  = build ["Adolfo","Diego","Juan","Pedro","Tomas"]

print names
"Adolfo"
|- <>
|- "Diego"
   |- <>
   |- "Juan"
      |- <>
      |- "Pedro"
         |- <>
         |- "Tomas"

print (balance names)
"Juan"
|- "Diego"
   |- "Adolfo"
   |- <>
|- "Tomas"
   |- "Pedro"
   |- <>
```

## 5. Búsquedas

El objetivo final de la práctica es poder realizar búsquedas en la estructura de datos que hemos definido. Para ello deberás implementar la siguiente función:

- **`between t xmin xmax`** – Busca en el árbol binario de búsqueda `t` y devuelve una lista con todos los elementos del árbol que están entre los valores `xmin` y `xmax` (ambos inclusive).
  La búsqueda debe evitar recorrer los subárboles que no sea necesario inspeccionar, y no es necesario devolver la lista ordenada.

## 6. Pruebas

Te proporcionamos un fichero **BinaryTreeTest.hs** que prueba todas las funciones solicitadas en la práctica. Además de las pruebas que tú mismo hagas con el intérprete de Haskell o con tu propio programa principal, tus funciones deberán compilar con dicho archivo de pruebas. De lo contrario la calificación de la práctica será de 0.

## Entrega

Como resultado de esta práctica deberás entregar **sólo el siguiente archivo**:

- **BinaryTree.hs** – que contenga la definición de la estructura de datos junto con todas las funciones arriba mencionadas.

Todas las funciones deben tener exactamente el nombre e interfaz definidos en este guión.
Todos los archivos de código fuente solicitados en este guión deberán ser comprimidos en un único archivo `zip` con el siguiente nombre:

- `practica6_<nip1>_<nip2>.zip` (donde `<nip1>` y `<nip2>` son los NIPs de los estudiantes involucrados) si el trabajo ha sido realizado por parejas. En este caso sólo uno de los dos estudiantes deberá hacer la entrega.

- `practica6_<nip>.zip` (donde `<nip>` es el NIP del estudiante involucrado) si el trabajo ha sido realizado de forma individual.

El archivo comprimido a entregar no debe contener ningún fichero aparte de los fuentes que te pedimos: ningún fichero ejecutable o de objeto, ni ningún otro fichero adicional. La entrega se hará en la tarea correspondiente a través de la plataforma Moodle:
`http::://moodle.unizar.es`