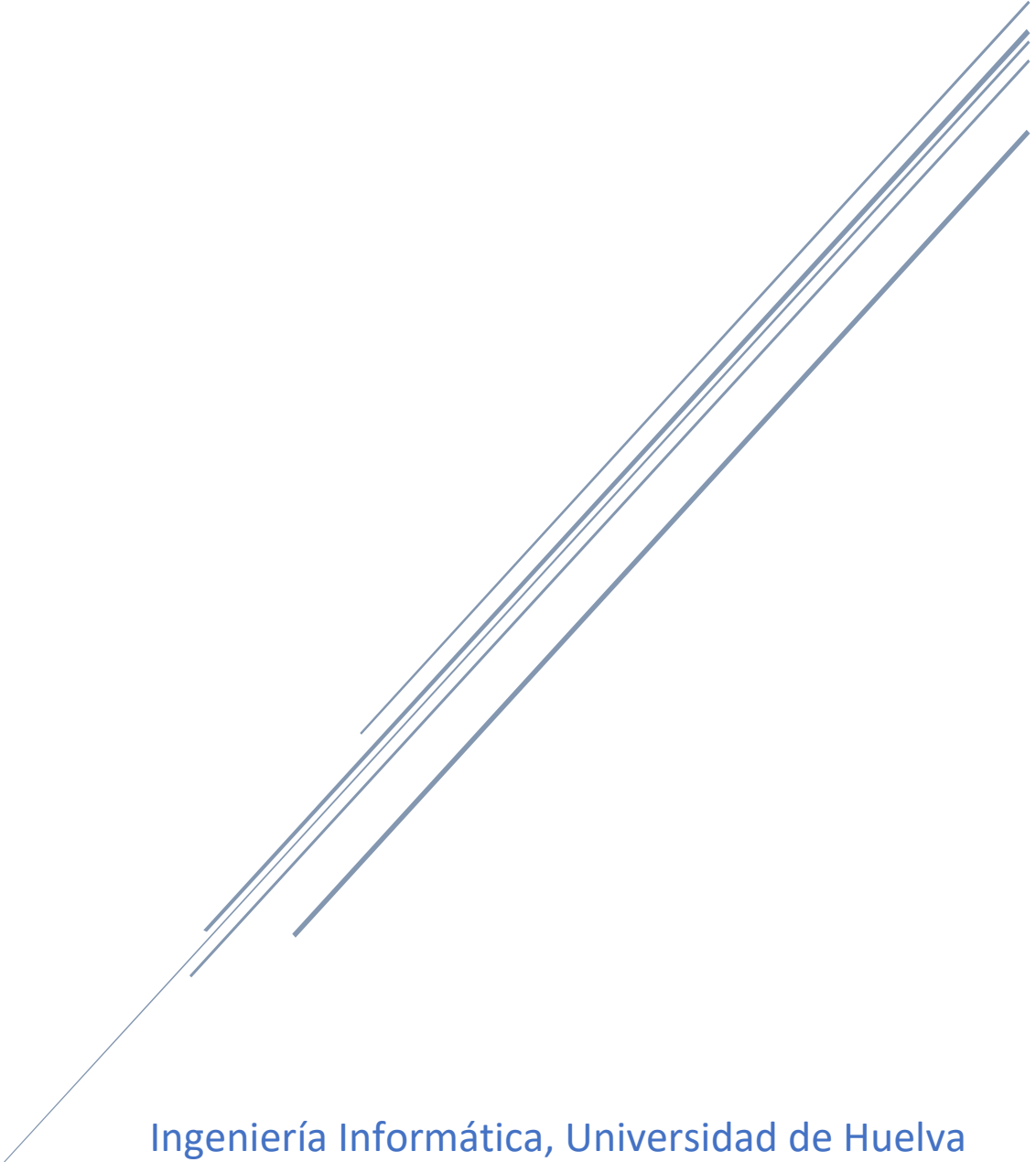


PRÁCTICA 4: DATA TYPES

Alba Márquez-Rodríguez



Ingeniería Informática, Universidad de Huelva
Modelos Avanzados de Computación

Contenido

Ejercicio 1: DROPPRECIO X 2

 Código Fuente 2

 Descripción del código 3

Ejercicio 2: GETLIST_DATE date criterio 4

 Código Fuente 4

 Descripción del código 5

Ejercicio 3: RECORRE ARBOL 6

 Código Fuente 6

 Descripción del código 7

Ejercicio 1: DROPPRECIO X

Dada una lista de Pizzas [lista de ingredientes y un precio], devuelve el segmento más largo de la lista que comienza con la pizza que tiene el menor precio superior a X euros.

Código Fuente

```
-- Tipos de Datos
-- Atributos de La Pizza
type Ingredients = [String]
type Price = Int
-- Dato Pizza formado por Los atributos previamente definidos
data Pizza = Pizza {ingredients :: Ingredients, price :: Price} deriving(Show, Eq) --Eq required

-- Ejemplos de Pizzas
-- declarado como pizza = atributo 1, atributo 2
margarita = Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Ham"], price=8}
formaggio = Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Blue Cheese","Gouda"], price=14}
barbecue = Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Bacon","Chicken","Barbacue Sauce"], price=12}
carbonara = Pizza {ingredients = ["Cream","Mozzarella","Onion","Mushroom"], price=13}
hawaian = Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Ham","Pineapple"], price=12}

-- Creación de Lista de Pizzas
pizzaList = [formaggio,barbecue,carbonara,margarita,hawaian]

-- devuelve el segmento más largo de la lista que comienza con la pizza que tiene el menor precio superior a X euros
-- 1. Filtrar por precios superiores a X€
higherPriceThan :: Price -> [Pizza] -> [Pizza]
higherPriceThan _ [] = [] -- base case
higherPriceThan x (head_element:rest_list) =
  if (price(head_element) > x) then
    [head_element]++(higherPriceThan x rest_list)
  else
    (higherPriceThan x rest_list)

testhigher :: Price -> [Pizza]
testhigher x = higherPriceThan x pizzaList

-- 2. Filtrar con Pizza con menor precio
lowerPricePizza :: [Pizza] -> Pizza
lowerPricePizza (head_element:[]) = head_element -- base case
lowerPricePizza (head_element:(head_element2:rest_list)) =
  if price(head_element) <= price(head_element2) then
    lowerPricePizza ([head_element]++rest_list)
  else lowerPricePizza ([head_element2]++rest_list)

testlower = lowerPricePizza pizzaList

-- Llama a las funciones necesarias
dropPrice :: Price -> [Pizza]
dropPrice x = dropPriceRec x pizzaList

dropPriceRec :: Price -> [Pizza] -> [Pizza]
dropPriceRec _ [] = []
dropPriceRec x (head_element:rest_list) =
  if head_element == lowerPricePizza(higherPriceThan x ([head_element]++rest_list)) then
    higherPriceThan x ([head_element]++rest_list)
  else
    dropPriceRec x rest_list
```

Descripción del código

Lo primero será crear el objeto tipo Pizza que estará compuesto por:

- Ingredients: ingredientes, lista de strings
- Price: Precio, que es un numero entero

Con esto podremos crear una pizza.

Lo siguiente será la declaración de las pizzas, hemos cogido 5 tipos de pizzas:

- Margarita
- Formaggio
- Barbacue
- Carbonara
- Hawaian

Una vez creados los objetos pizza, las introduciremos en una lista que será la lista pizza.

Para realizar lo que pide el ejercicio se dividirá en 2 partes:

1. **Filtrar los precios superiores al parámetro X** pasado por el usuario. El caso base será en el momento en el que la lista esté vacía. En otro caso cogerá la primera pizza y si su precio es superior a x lo concatenará con la lista de pizzas con mayor precio que x (quitándole la cabeza, que acabamos de verificar). Así podremos llamar a la función recursivamente y crear una lista de pizzas cuyo precio sea superior a X.
2. **Mostrar el segmento más largo a partir de la pizza con menor precio superior a X.** Esto lo hemos interpretado de manera que una vez filtradas las pizzas y encontrada la de menor precio, mostrará todas las pizzas a partir de esta que cumplan la primera condición. Esto se ha hecho, con el caso base en el que la lista está vacía, devuelve una lista vacía. En otro caso separa la lista en 3, cabecera1, cabecera2 y el resto de la lista. La función será llamada recursivamente sin la cabecera1, es decir, con cabecera2 y el resto de la lista. Así irá comparando los precios de las pizzas (1º y 2º elementos de la lista de pizzas de la llamada a función en la que nos encontramos). Compara las pizzas 1 y 2 y hace recursividad sobre la pizza más barata y el resto de las pizzas. Así al final devolverá la pizza más barata.

En la recursividad lo que hará será que si la cabecera de la lista es igual a la pizza mas barata que cumple la condición de ser mayor que X. Entonces devolverá toda la lista. En caso contrario, seguirá haciendo recursión con el resto de pizzas de la lista hasta que encuentre la más barata que cumpla la condición.

```
Main> dropPrice 10
[Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Bacon","Chicken","Barbacue Sauce"],
price = 12},Pizza {ingredients = ["Cream","Mozzarella","Onion","Mushroom"], price =
13},Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Ham","Pineapple"], price = 12}]
Main> dropPrice 8
[Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Bacon","Chicken","Barbacue Sauce"],
price = 12},Pizza {ingredients = ["Cream","Mozzarella","Onion","Mushroom"], price =
13},Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Ham","Pineapple"], price = 12}]
Main> dropPrice 7
[Pizza {ingredients = ["Tomatoe Sauce","Mozzarella","Ham"], price = 8},Pizza {ingredients =
["Tomatoe Sauce","Mozzarella","Ham","Pineapple"], price = 12}]
Main> |
```

Ejercicio 2: GETLIST_DATE date criterio

Dada una lista de Personas con los datos [nombre, apellido y fecha de nacimiento] ordenada ascendentemente por fecha de nacimiento (deberá crear el tipo fecha), devuelve el segmento más largo de la lista con las personas que nacieron antes, después o en esa fecha.

Código Fuente

```
--                                     Tipos de Datos
-- Atributos de la persona
type Name = String
type Surname = String
data Date = Date {day::Integer, month::Integer, year::Integer} deriving(Eq,Show)
-- Dato Persona formado por los atributos previamente definidos
data Person = Person {name::Name, surname::Surname, birth_date::Date} deriving(Show) --Eq required

--                                     Ejemplos de Pizzas
-- declarado como pizza = atributo 1, atributo 2
person1 = Person {name="Maria",surname="Garcia Correa",birth_date=(Date 18 10 2001)}
person2 = Person {name="Alba",surname="Marquez Rodriguez",birth_date=(Date 7 5 2001)}
person3 = Person {name="Jose",surname="Hernandez Gomez",birth_date=(Date 30 4 1997)}
person4 = Person {name="Fran", surname="Perea Rodriguez",birth_date=(Date 30 4 1984)}
person5 = Person {name="Antonio", surname="Gonzalez Rodriguez",birth_date=(Date 23 11 1963)}

-- Creación de lista de personas
peopleList = [person1,person2,person3,person4,person5]

-- Funciones Auxiliares
earlierDate :: Date -> Date -> Bool
earlierDate date1 date2 = ((year(date1)*10000) + (month(date1)*100) + day(date1)) < ((year(date2)*10000) + (month(date2)*100) + day(date2))

laterDate :: Date -> Date -> Bool
laterDate date1 date2 = ((year(date1)*10000) + (month(date1)*100) + day(date1)) > ((year(date2)*10000) + (month(date2)*100) + day(date2))

test :: Date -> [Person]
test date = [x | x<-peopleList, earlierDate (birth_date x) date]

-- Llama a las funciones necesarias

-- devuelve lista de personas nacidas antes, despues o con la misma date que la introducida
getListDate :: Date -> String -> [Person]
getListDate date what = case what of
    "antes"    -> [x | x<-peopleList, earlierDate (birth_date x) date]
    "despues"  -> [x | x<-peopleList, laterDate (birth_date x) date]
    "misma"    -> [x | x<-peopleList, (birth_date x) == date]
    otherwise  -> error "Error, introduzca: antes, despues o misma"
```

Descripción del código

Lo primero será crear el objeto tipo Persona que estará compuesto por:

- Name: string
- Surname: string
- Date: fecha aque está compuesta por 3 enteros (día, mes año)

Con esto podremos crear una persona.

Lo siguiente será la declaración de las personas, hemos creado 5 personas. Cada una con nombres, apellidos y fechas de nacimiento. Creadas en orden ascendente por fecha de nacimiento.

Una vez creadas las personas se crea una lista de personas.

Para realizar lo que pide el ejercicio se crearan dos funciones auxiliares para comparar los objetos Date creados. Como se puede comparar directamente si son iguales, no hará falta crear esta pero sí si una fecha es mayor que otra o si una fecha es menor que otra. Estas funciones serán booleanas.

Las dos funciones de comparar fecha funcionan de una manera parecida, así que explicaremos una de ellas. La forma de hacerlo es convertir la fecha en un entero y comparar estos dos enteros. El año será el entero con mayor valor por lo que se multiplicará por 10000 ($2000 \times 1000 = 20000000$) a esto se le sumará el mes que es el segundo con mayor valor por lo que se multiplicará por 100 $\rightarrow (12 \times 100) + 20001200$. Por último los días son los de menor valor y no se multiplican por ningún coeficiente $+(31) = 20001231$. Así cada dígito de la fecha tiene su propio lugar y no se pisa y se puede comparar con un número procesado de la misma manera.

La función principal, dependiendo del criterio seleccionado por la persona devolverá una lista de personas que cumplan la condición seleccionada que podrá ser:

- Antes
- Después
- Misma

```
Main> getListDate (Date 7 5 2001) "misma"
[Person {name = "Alba", surname = "Marquez Rodriguez", birth_date = Date {day = 7, month = 5, year = 2001}}]
Main> getListDate (Date 7 5 2000) "antes"
[Person {name = "Jose", surname = "Hernandez Gomez", birth_date = Date {day = 30, month = 4, year = 1997}}, Person {name = "Fran", surname = "Perea Rodriguez", birth_date = Date {day = 30, month = 4, year = 1984}}, Person {name = "Antonio", surname = "Gonzalez Rodriguez", birth_date = Date {day = 23, month = 11, year = 1963}}]
Main> getListDate (Date 7 5 2000) "despues"
[Person {name = "Maria", surname = "Garcia Correa", birth_date = Date {day = 18, month = 10, year = 2001}}, Person {name = "Alba", surname = "Marquez Rodriguez", birth_date = Date {day = 7, month = 5, year = 2001}}]
```

Ejercicio 3: RECORRE ARBOL

Crear un Node binario con 5 Leafs y realizar mostrar los elementos del árbol por pantalla, recorriendo el árbol en profundidad o en anchura. Debe ser especificado el método utilizado.

Código Fuente

```
-- Declaración del dato arbol Binario
data BinaryTree = Empty | Leaf {value::Int} | Node {value::Int, left::BinaryTree, right::BinaryTree} deriving(Eq,Show)

-- Declaración del ejemplo de arbol
tree = Node(value=0,
            left=(Node(value=1,
                        left=(Leaf(value=3)),
                        right=(Leaf(value=4)))),
            right=(Node(value=2,
                        left=(Leaf(value=5)),
                        right=Empty)))
```

```
-- Funciones Auxiliares

-- nivel y altura

-- Funcion Leveltree
leveltree :: BinaryTree -> Int -> [Int]
leveltree Empty _ = []
leveltree (Node value left right) 0 = [value]
leveltree (Leaf x) level = [x]
leveltree (Node value left right) level = (leveltree left (level-1))++(leveltree right (level-1))

-- Funcion getheight
getheight :: BinaryTree -> Int
getheight (Leaf _) = 1
getheight (Node _ left Empty) = 1 + (getheight left)
getheight (Node _ Empty right) = 1 + (getheight right)
getheight (Node _ left right) = 1 + (max (getheight left) (getheight right))
```

```
-- Funciones de Recorridos

-- Profundidad: preorden, inorden, postorden

-- preorden (current, left, right)
preorder :: BinaryTree -> [Int]
preorder Empty = []
preorder (Leaf x) = [x]
preorder (Node value left right) = [value] ++ (preorder left) ++ (preorder right)

-- inorden (left, current, right)
inorder :: BinaryTree -> [Int]
inorder Empty = []
inorder (Leaf x) = [x]
inorder (Node value left right) = (inorder left) ++ [value] ++ (inorder right)

-- postorden (left, right, current)
postorder :: BinaryTree -> [Int]
postorder Empty = []
postorder (Leaf x) = [x]
postorder (Node value left right) = (postorder left) ++ (postorder right) ++ [value]

-- Anchura (left to right)
width :: BinaryTree -> [Int]
width tree = widthRec tree ((getheight tree)-1)

widthRec :: BinaryTree -> Int -> [Int]
widthRec tree 0 = leveltree tree 0
widthRec tree level = (widthRec tree (level-1))++(leveltree tree level)
```

```
-- Funcion Principal

recorrearbol :: String -> [Int]
recorrearbol what = case what of
    "preorden" -> preorder tree
    "inorden" -> inorder tree
    "postorden" -> postorder tree
    "anchura" -> width tree
    otherwise -> error "Error"
```

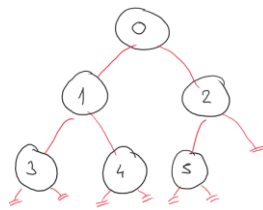
Descripción del código

Lo primero será crear el objeto tipo árbol que podrá ser de varias formas:

- Vacío
- Ser una hoja, con un valor
- Ser un Nodo compuesto por un valor, un objeto a la izquierda y otro a la derecha, ambos de tipo árbol.

Con esto podremos crear un árbol binario.

Lo siguiente será la declaración de un ejemplo de árbol, el creado en este caso es:



Una vez ya tenemos creado nuestro objeto de tipo árbol, podemos pasar a la creación de las funciones.

Primero crearemos los recorridos en profundidad: preorden, inorden y postorden ya que se pueden programar sin funciones auxiliares.

Preorden: (actual, izq, dcha): en el caso en el que el árbol sea vacío devolverá vacío. Eso será nuestro caso base. Otro caso es en el que sea una hoja, entonces devolverá el valor de esta hoja. Por último, si el árbol está compuesto por un nodo con dos subárboles entonces concatenerá el valor del nodo actual con el preorden del subárbol izquierdo y luego el preorden del subárbol derecho.

Inorden: (izq, actual, dcha): en el caso en el que el árbol sea vacío devolverá vacío. Eso será nuestro caso base. Otro caso es en el que sea una hoja, entonces devolverá el valor de esta hoja. Por último, si el árbol está compuesto por un nodo con dos subárboles entonces concatenerá el inorden del subárbol izquierdo, el valor del nodo actual y luego el inorden del subárbol derecho.

Postorden: (izq, dcha, actual): en el caso en el que el árbol sea vacío devolverá vacío. Eso será nuestro caso base. Otro caso es en el que sea una hoja, entonces devolverá el valor de esta hoja. Por último, si el árbol está compuesto por un nodo con dos subárboles entonces concatenerá el postorden del subárbol izquierdo, con el postorden del subárbol derecho y luego el valor del nodo en el que nos encontramos.

Para el recorrido en anchura será necesario crear una función que tome la altura del árbol y así leer los nodos que se encuentran en el nivel que pasemos cuando lo hagamos recursivamente. Esto se empezará a hacer desde la altura del árbol hasta la hoja del nivel más bajo. Por eso tenemos la función para leer los nodos del nivel pasado por parámetro: `leveltree`. Lo que haremos para leer por anchura será llamar recursivamente hasta leer todos los niveles del árbol con `leveltree`.

La función principal llamará a una función u otra según la instrucción del usuario.


```
Main> recorrearbol "anchura"  
[0,1,2,3,4,5]  
Main> recorrearbol "preorden"  
[0,1,3,4,2,5]  
Main> recorrearbol "inorden"  
[3,1,4,0,5,2]  
Main> recorrearbol "postorden"  
[3,4,1,5,2,0]
```