

ESTRUCTURA DE DATOS.

APUNTES DE LA ASIGNATURA

ÁNGEL MANUEL SORIA GIL
UNIVERSIDAD DE MÁLAGA.
E.T.S. DE INGENIERÍA INFORMÁTICA.
CURSO 2022/2023

AMSG

ESTRUCTURA DE DATOS.....	1
CAPÍTULO 1. INTRODUCCIÓN A LA PROGRAMACIÓN FUNCIONAL	5
1. PROGRAMACIÓN FUNCIONAL. BREVE INTRODUCCIÓN.	5
2. HASKELL. INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN.....	5
CAPÍTULO 2. CARACTERÍSTICAS DE LA PROGRAMACIÓN FUNCIONAL	9
1. LISTAS.....	9
2. FUNCIONES DE ORDEN SUPERIOR.....	11
3. FUNCIONES Y SUS CARACTERÍSTICAS.	12
4. PLEGADOS. FOLDL Y FOLDR.	12
5. TIPOS ALGEBRAICOS.	13
CAPÍTULO 3. TIPOS ABSTRACTOS DE DATOS.....	15
1. INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS.	15
2. MÓDULOS, INTERFACES Y AXIOMAS.	15
3. EL TAD STACK (PILA).....	16
4. EL TAD QUEUE (COLA).....	18
5. EL TAD SET (CONJUNTO).....	19
6. LISTAS E ITERADORES EN JAVA.....	20
CAPÍTULO 4. ÁRBOLES.....	21
1. INTRODUCCIÓN A ÁRBOLES.	21
2. ÁRBOLES BINARIOS.....	22
3. COLAS CON PRIORIDAD Y MONTÍCULOS BINARIOS.	23
4. ÁRBOLES BINARIOS DE BUSQUEDA (BST).....	24
5. ÁRBOLES BALANCEADOS EN ALTURA (AVL).....	27
6. DICCIONARIOS.....	30
CAPÍTULO 5. GRAFOS.	31
1. DEFINICIONES BÁSICAS.....	31
2. REPRESENTACIÓN DE GRAFOS EN HASKELL	32
3. RECORRIDO DE GRAFOS.....	33
4. CAMINOS A VÉRTICES.....	35
5. GRAFOS DIRIGIDOS.....	37
6. REPRESENTACIÓN MATEMÁTICA.....	39
7. GRAFOS Y DIGRAFOS EN JAVA.....	40
8. ORDEN TOPOLOGICO.....	42
9. OTROS PROBLEMAS INTERESANTES.	43

AMSG

CAPÍTULO 1. INTRODUCCIÓN A LA PROGRAMACIÓN FUNCIONAL.

1. PROGRAMACIÓN FUNCIONAL. BREVE INTRODUCCIÓN.

1.1. PROGRAMACIÓN IMPERATIVA Y FUNCIONAL.

La programación que hemos visto hasta ahora es la llamada programación imperativa. En esta, manejamos variables mutables, es decir, que pueden cambiar, esto es que una misma variable puede tomar un valor x en un momento y un valor y en otro momento. Muestra típica de ello son códigos del tipo: $p = p+1$.

Observación: carecer de esta mutabilidad nos obliga a prescindir de los bucles y de la iteratividad.

La programación funcional es aquella en la que los programas son básicamente funciones. Tomamos unos datos de entrada, se realizan cálculos con estos datos y nos proporciona nuevos datos como resultado, pero ninguno de los datos proporcionados cambia. En programación funcional no hay variables mutables como sí ocurría en la imperativa.



Estas funciones pueden ser de dos tipos, puras o impuras. La diferencia entre ambas es que las primeras deben verificar la llamada Regla de Leibniz, es decir, que si $x = y$, entonces $f(x) = f(y)$.

Así podemos resumir básicamente en que la programación funcional consiste en escribir programas, los cuales son funciones escritos a partir de la combinación de otras funciones. Nosotros trabajaremos con un lenguaje funcional puro: Haskell.¹

2. HASKELL. INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN.

2.1. CARACTERÍSTICAS PRINCIPALES DE HASKELL.

En general, los lenguajes de programación funcional pura se caracterizan por la optimización, proveniente de transformaciones realizadas en el compilador; el paralelismo, al poder realizar distintos cálculos a la vez pues los datos no cambian en ningún momento; y memorización, aprovecha llamadas a las funciones y recuerda dichos valores para ahorrar cálculos.

Así podemos afirmar que las principales características de Haskell como lenguaje son la pureza funcional, la tipificación estática (esto es que incluye comprobación de tipos antes de ejecución),

¹ Más adelante también se trabajará con Java.

y la “pereza”, pues aprovecha cálculos hechos previamente y evita realizar otros mediante la no evaluación cuando no es necesario.

2.2. NOTACIÓN EN HASKELL. IDENTIFICADORES

Debemos tener en cuenta que Haskell es un lenguaje case sensitive, es decir, que distingue el uso de mayúsculas y minúsculas. Así se establece que las funciones y argumentos deben empezar por letras minúsculas o el carácter “_” (underscore). Por otro lado, los tipos deben empezar por mayúscula. Además, los operadores pueden contener uno o varios símbolos siempre y cuando el primero no sean dos puntos.

En Haskell también está la llamada notación parcializada, así para llamar a una función se escribe el identificador de esta y acto seguido sus argumentos separados por espacios, no emplea paréntesis como venimos acostumbrados a ver. El uso de estos últimos se utiliza únicamente para agrupar términos compuestos y modificar las prioridades de operaciones que veremos más adelante.

2.3. TIPOS BÁSICOS. NUMÉRICOS, BOOLEANOS Y CARACTERES.

Comenzaremos con los tipos numéricos. En ellos distinguimos: Int (conjunto acotado de enteros, puede ocurrir overflow al superar los 32 bits), Integer (conjunto de los enteros), Float (subconjunto de reales con precisión simple) y Double (subconjuntos de reales con precisión doble).

Continuamos con los booleanos. Se denotará por Bool y puede tomar únicamente los valores True y False. Además, para ellos se define los operadores típicos (&&), (||) y not..

Por último, los caracteres. Para ello tenemos el tipo Char que es un conjunto de caracteres Unicode y podemos además importar funciones específicas para este tipo con la sentencia “import Data.Char”.

2.4. DEFINICIÓN DE FUNCIONES.

En forma general una función en Haskell se define del siguiente modo:

nombre :: Tipo Dominio 1 -> ... -> Tipo Dominio n -> Tipo Codominio

nombre d1 ... dn = expresión

Veamos un ejemplo:

```
pythagoras :: Integer -> Integer -> Integer
pythagoras x y = square x + square y
```

Además, en estas definiciones se pueden añadir elementos condicionales, con las sentencias if, then y else; y elementos recursivos.

2.5. OPERADORES, ORDEN DE PREFERENCIA Y ASOCIATIVIDAD.

En Haskell podemos distinguir principalmente los siguientes operadores:

Operadores aritméticos:

- | | |
|---------------------------------------|--|
| - (+): Suma dos números. | - (-): Resta dos números. |
| - (*): Multiplica dos números. | - (/): Divide dos números. |
| - (div): Realiza una división entera. | - (^): Eleva un número a una potencia. |

Operadores de comparación:

- | | |
|---|--|
| - (==): Comprueba si dos valores son iguales. | - (/=): Comprueba si dos valores son diferentes. |
| - (<): Comparador menor que | - (<=): Comparador menor o igual que |
| - (>): Comparador mayor que | - (>=): Comparador mayor o igual que |

Operadores lógicos:

- (`&&`): Operador AND lógico, devuelve True si ambos valores son True.
- (`||`): Operador OR lógico, devuelve True si al menos uno de los valores es True.
- `not`: Operador NOT lógico, devuelve el valor opuesto (True si el valor es False y viceversa).

Operadores de listas:

- (`:`): Operador de construcción de lista, agrega un elemento al comienzo de una lista existente.
- (`++`): Concatena dos listas.
- (`!!`): Accede al elemento en la posición específica de una lista.

Operadores de funciones:

- `(.)`: Composición de funciones, permite encadenar funciones.
- `(\$)`: Aplicación de función, permite aplicar una función a un argumento.

Además, se asignan los siguientes órdenes de prioridad y asociatividad.

Priority	Associativity	Predefined Operators
10	left	Function application
9	left	<code>!!</code>
9	right	<code>.</code>
8	right	<code>^ ^ ^ ^ **</code>
7	left	<code>* /</code>
6	left	<code>+ -</code>
5	right	<code>: ++</code>
4	non associative	<code>== /= < <= > >=</code>
3	right	<code>&&</code>
2	right	<code> </code>
1	left	<code>>> >>=</code>
1	right	<code>=<<</code>
0	right	<code>\\$ \\$!</code>

Nota: debemos destacar que estos operadores pueden utilizar la notación prefija (x `operador` y) o la notación infija ((operador) x y)

2.6. EVALUACIÓN DE EXPRESIONES.

Haskell utiliza evaluación perezosa, esto es, aplica primera las definiciones de las aplicaciones ó funciones y después evalúa el mínimo número de veces aprovechando la pureza funcional. Un ejemplo de ello sería: `twice (10+2) → (10+2) + (10+2) → 12 + 12 → 24`²

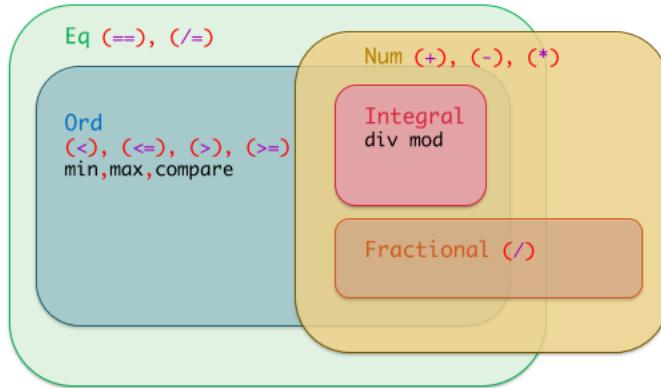
2.7. TUPLAS.

Las tuplas son estructuras de datos heterogéneas que básicamente son colecciones de valores del mismo tipo o no. Los valores se separan por comas y se agrupan entre paréntesis. No necesariamente se necesita un tipo fijo, puede definirse con tipos a y b genéricos.

² El primer paso consiste en aplicar la definición de `twice`, el segundo paso es la evaluación de `10+2`, que únicamente se realiza una vez por al aprovechar el valor por la pureza funcional. Por último, calcula `12 + 12`.

2.8. SOBRECARGA Y CLASES DE TIPOS.

Una clase de tipos es un conjunto de tipos que comparten alguna operación. Las principales clases son Integral (para los enteros), Fractional (para los racionales), Num (para cualquier tipo numérico), Ord (para los tipos ordenables), Eq (para tipos que pueden igualarse)... estos pueden organizarse en el siguiente mapa conceptual.



2.9. FUNCIONES A TROZOS. GUARDAS Y EXPRESIONES CONDICIONALES.

Al igual que con las funciones matemáticas, muchas veces es más sencillo expresarlas a trozos, así distinguimos distintos casos. Para ello es para lo que nos sirven las guardas. Estas se representan con una barra vertical |.

Para el funcionamiento de estas guardas también es fundamental el sangrado (recordamos que Haskell trabaja directamente con la sangría).

Así tenemos que las guardas suponen una alternativa a las clásicas expresiones condicionales de if-else. Además, tenemos el comando “otherwise” que sirve para indicar el comportamiento de la función en cualquier otro caso no contemplado anteriormente en las guardas. Veamos un ejemplo:

```

abs :: (Ord a, Num a) => a -> a
abs x | x >= 0 = x
      | otherwise = -x
  
```

Esto también nos puede llevar a funciones parcialmente definidas, como por ejemplo para definir el inverso de un elemento, en los racionales, para el cero este no puede ser definido, así para el caso en que x es 0 se lanzaría un error y para el resto de los casos se devolvería el inverso.

2.10. DEFINICIONES LOCALES.

En muchos lugares nos encontraremos con la necesidad de definir algunas variables en las funciones para acortarlas o facilitar su legibilidad. Muchas veces se definen a priori, luego empleamos la sentencia “where” ³que nos permite decir qué significa cada una de las variables definidas. Veamos mejor otro ejemplo.

```

(~=) :: Double -> Double -> Bool
x ~= y = abs (x-y) < epsilon
  where epsilon = 1/1000
  
```

³ Una alternativa a where sería let...in

CAPÍTULO 2.

CARACTERÍSTICAS DE LA

PROGRAMACIÓN

FUNCIONAL.

1. LISTAS.

Las listas son secuencias de datos homogéneos, es decir, se almacenan varios valores indexados de un mismo tipo. En Haskell, para escribir una lista hace con valores separados por comas encerrados entre corchetes. Veamos algunos ejemplos.

<code>[]</code>	la lista vacía
<code>[7, 2, 5]</code>	tiene tipo <code>[Integer]</code>
<code>[True, False]</code>	tiene tipo <code>[Bool]</code>
<code>[('a', True), ('a', False)]</code>	tiene tipo <code>[(Char, Bool)]</code>

Antes de nada, vamos a aclarar las principales diferencias de las listas y las tuplas. En las listas, todos los elementos deben ser del mismo tipo, en una tupla no necesariamente. Además, la longitud de las listas puede cambiar sin afectar a su tipo, en las tuplas sí se debería que modificar.

Nota: el tipo String queda definido como una lista de caracteres del siguiente modo: type String = [Char]. No obstante, hay una notación especial para él. Puede escribirse entre comillas y aplicar sobre ellos las mismas operaciones que para listas.

1.1. SECUENCIAS ARITMÉTICAS.

Una secuencia o sucesión aritmética matemática puede verse como una lista en la que la diferencia entre dos elementos consecutivos es constante. Para facilitar la notación, podremos ahorrarnos escribir toda la lista. Para ello Haskell utiliza la notación que se infiere de los siguientes ejemplos.

`[1..10]` → En este caso al hallarse únicamente el elemento inicial y el final, no se puede hallar dicha diferencia constante, luego se toma uno, así esta sería la lista de enteros del 1 al 10.

`[1, 3 .. 10]` → Si indicamos los elementos primero, segundo y último entonces podremos calcular dicha diferencia, en este caso sería la lista de enteros impares comprendidos entre el 1 y el 10.

`[1 ..]` → También podemos representar listas infinitas sin expresar el último elemento. En este caso se representa la lista de naturales.

1.2. LISTAS POR COMPRENSIÓN.

Haskell hasta el momento es un lenguaje que intenta trasladar el lenguaje matemático a la programación. En matemáticas esencial la definición de conjuntos. En Haskell algo muy parecido son las listas por comprensión. Básicamente se define una lista del siguiente modo: `[expresión de x | condiciones sobre x]`. Vamos a ver dos casos particulares como ejemplos.

$[x^2 \mid x <- [1..]] \rightarrow$ Es una lista de los cuadrados de los naturales.

$[(x^2, \text{even}(x^2)) \mid x <- [1, 3, ..]] \rightarrow$ Es una lista de duplas formadas por un número y si es par o no. Dichos números son los cuadrados de los números impares.⁴

En ocasiones, para evitar evaluar varias veces un mismo elemento, se permite definir dentro de la lista con la expresión let. Veamos otro ejemplo.

`sinVocales xs = [y | x <- xs, let y = toLower x, y `notElem` "aeiou"]`

1.3. FUNCIONES PREDEFINIDAS SOBRE LISTAS.

En esta sección vamos a descubrir algunas funciones predefinidas sobre listas que no nos harán falta importar. Son las siguientes:

- `null :: [a] → Bool` Dice si una lista es vacía o no
- `head :: [a] → a` Devuelve el primer elemento de una lista
- `last :: [a] → a` Devuelve el último elemento de una lista
- `tail :: [a] → [a]` Devuelve la lista pero sin el primer elemento.
- `init :: [a] → [a]` Devuelve la lista pero sin el último elemento.
- `take :: Int → [a] → [a]` Devuelve los primeros x elementos de una lista.
- `drop :: Int → [a] → [a]` Devuelve los últimos x elementos de una lista.
- `length :: [a] → Int` Devuelve la longitud de una lista.
- `elem :: Eq a => a → [a] → Bool` Devuelve si un elemento se encuentra en una lista.
- `notElem :: Eq a => a → [a] → Bool` Es la inversa de elem.
- `reverse :: [a] → [a]` Devuelve la lista en orden inverso.
- `(++) :: [a] → [a] → [a]` Devuelve la concatenación de dos listas.

1.4. CONSTRUCTORES DE LISTAS.

La forma en que hemos representado y escrito anteriormente las listas es azúcar sintáctico, es decir, una forma de escribir más fácil una lista. Estas en verdad se construyen a partir de la lista vacía [] y el operador (:).

Veamos en qué consiste este último. Se define del siguiente modo: `(:) :: a → [a] → [a]`. Dado un elemento del tipo a y una lista de elementos de ese mismo tipo, añade el elemento dado a la cabeza de la lista. Así podemos expresar también listas con dicho operador como prosigue.

`(x:xs), (x:y:xs)`

En la primera expresión damos un primer elemento x y el resto de la lista que sería xs. En la segunda expresión damos el primer elemento x, el segundo elemento y; y el resto de la lista xs. Estos patrones nos ayudarán mucho en la definición de funciones recursivas con listas. Un claro caso del uso de patrones y además recursividad en listas es el siguiente:

```
sorted :: (Ord a) => [a] -> Bool
sorted []      = True
sorted [_]     = True
sorted (x:y:zs) = x <= y && sorted (y:zs)
```

En ella [] define la lista vacía, [_] utiliza un patrón que define una lista con un único elemento y (x:y:zs) es el patrón explicado anteriormente

⁴ Puede haber varias condiciones sobre un mismo o varios elementos, en tal caso, cada condición ha de ser separada por una coma.

sobre el cual además se aplica recursividad. _ por sí solo denota cualquier otra cosa que encaje en ese sitio, es un poco como un hueco a completarse.

Hemos comentado que empleamos la recursividad. Vamos a indagar un poco más en este asunto.

1.5. RECUSIVIDAD EN LISTAS.

Por lo general para definir ciertas operaciones con listas vamos a hacerlo recursivamente como en el caso anterior. La idea es sencilla, hemos establecido unas soluciones fijas para algunos casos base y luego hemos dividido el problema. Hemos visto si los primeros dos elementos son ordenados y luego le añadimos que el resto de la lista lo sea con una llamada recursiva.

En general podemos seguir siempre este esquema. Necesitamos unos casos base, la solución a estos y una forma de añadir la solución de un elemento a la del resto de la lista. Esto se puede reflejar en el siguiente esquema con guardas.

$f [] = \text{solución caso base}$

$f (x:xs) = \text{añadir } x \text{ (f xs)}$

No obstante, esta no es siempre la mejor solución, en funciones como reverse por ejemplo alcanzaríamos una complejidad demasiado elevada usando este esquema, esto nos lleva a buscar una forma más eficiente de operar. En ello nos ayudarán los acumuladores.

1.6. RECUSIVIDAD EN LISTAS CON ACUMULADORES.⁵

Hasta ahora a la hora de programar casi siempre habíamos usado bucles donde podíamos tener variables que nos servían de acumulador. Aquí vamos a tratar de hacer lo mismo, pero aplicando recursividad.

Vamos a tener una función sobre una lista, esta se resolverá llamando a una función auxiliar sobre la misma lista y un parámetro acumulador inicial que por lo general será la solución para la lista vacía. Solo nos falta definir localmente dicha función auxiliar. Se hace de forma recursiva siguiendo un modelo similar al anterior. Se define para un caso base que suele ser la lista vacía y su solución es el acumulador. Despues se define para un caso general en que llamamos a la función de forma recursiva para la cola de la lista y añadimos la solución del primer elemento al acumulador. Veamos un ejemplo para sumar los elementos de una lista.

suma :: [Integer] → Integer

suma xs = sumaAc xs 0

where

2. FUNCIONES DE ORDEN SUPERIOR.

Las funciones de orden superior son básicamente funciones que toman otras funciones como argumento y devuelven el resultado como una transformación de algo al aplicarle dicha función.

Las principales relacionadas con listas son las funciones map y filter. Map básicamente dada una función f y una lista devuelve otra lista al aplicarle f a cada uno de sus elementos, es decir, dada la aplicación f y la lista $[x_1 \dots x_n]$ devuelve la lista $[f x_1 \dots f x_n]$. Por otro la función de filter es precisamente la de filtrado de una lista. Vemos a continuación la forma en que se definen.

⁵ Esta idea de acumuladores es válida en cualquier otra función que no implemente listas. Es la idea básica de emplear acumuladores igual que en programas iterativos.

```

map :: (a->b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : aplica f xs

filter :: (a->Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
| p x = x : filter p xs
| otherwise = filter p xs

```

3. FUNCIONES Y SUS CARACTERÍSTICAS.

Sabemos qué cosas podemos hacer con los tipos específicos como puede ser Int, pero ¿qué podemos hacer con las funciones? Exactamente lo mismo. Se le puede dar un nombre a una función al almacenarla en una variable, se pueden guardar en estructuras de datos como listas, se pueden pasar como parámetro a otras funciones, se pueden devolver como resultado de otras funciones, aplicarle operaciones específicas para funciones... Vamos a ver algunas particularidades.

3.1. FUNCIONES LAMBDA O ANÓNIMAS.

Como ya hemos dicho, podemos pasar una función como parámetro, pero para ello no es necesario ponerles un nombre, podemos hacerlo especificándolas directamente gracias a las funciones lambda o funciones anónimas. Se expresan generalmente de la siguiente forma:

(\ parámetro → cuerpo función) por ejemplo (\x → x+x) sería la función (2*).

3.2. SECCIONES Y PARCIALIZACIÓN.

Una sección es un operador binario que recibe un solo operando, que puede ser el izquierdo o el derecho, por ejemplo (+3) o (2*). Esto no es más que azúcar sintáctico para facilitar la escritura de ciertas funciones. De esto modo escribiendo (2*) nos ahorraremos escribir (\x → x+x)

Una parcialización es un concepto que generaliza el de las secciones. Una función de n argumentos puede invocarse únicamente con k argumentos para $k \leq n$.

Así se puede definir una función g (parcialización de f) como la función f a la que ya se le han pasado los k primeros argumentos y faltan los n-k restantes, sería tal que $g = f \text{ arg1} \dots \text{argk}$

3.3. COMPOSICIÓN DE FUNCIONES.

Al igual que en matemáticas existe la composición de funciones, aquí también. Se sigue la misma notación empleando un punto para ello. Así (f.g) x será f(g(x)).

4. PLEGADOS. FOLDL Y FOLDR.

En este apartado vamos a estudiar dos funciones en particular. Serán foldl y foldr, estas son funciones que hacen lo que su nombre indica, plegar una lista a la izquierda o a la derecha. Ahora bien, ¿a qué nos referimos con plegar? Entenderemos plegar como reducir todos los elementos de una lista a un solo valor a partir de un operador binario de plegado. Veamos cómo se definen.⁶



Nota: véase que a la hora de definir las funciones foldl y hacer la llamada recursiva se ha de tener en cuenta cómo está definida la función binaria para operar los elementos, esto es lo que diferencia entre plegar a la izquierda y plegar a la derecha.

⁶ Se recomienda encarecidamente leer, comprender y hacer los ejercicios habidos en el documento hs aportado en el campus virtual sobre este apartado.

5. TIPOS ALGEBRAICOS.

En Java además de los tipos predefinidos podíamos definir nosotros nuestros propios tipos de datos. Esto lo hacíamos con las clases. En Haskell también tendremos esta opción. Para ello declararemos los nuevos tipos algebraicos del siguiente modo como norma general:

```
data NombreDelNuevoTipo = ...
```

No obstante, hay mucha variedad de tipos. Vamos a profundizar en los tipos enumerados, unión, producto, general y parametrizados.

5.1. TIPOS ENUMERADOS.

Estos son los más sencillos. Simplemente constan de una declaración en la que se enumeran los posibles valores del tipo. Siguen el siguiente esquema.

```
data NombreDelTipo = Valor1 | Valor2 | ... | ValorN
```

Imaginemos que entre los n valores que hemos dado queremos compararlos. ¿Cómo lo hacemos? Para ello están las clases de tipo⁷. Estas son similares a las interfaces de Java y cuando un tipo de datos implementa una de estas clases lo que ocurre es que se definen una serie de operaciones y funciones para el tipo. Pueden definirse manual o automáticamente mediante la sentencia "deriving". Principalmente utilizaremos las clases de tipo Show, para mostrar por pantalla los valores del tipo; Eq, para establecer el operador de igualdad en el tipo; y Ord, para establecer una relación de orden total. Podremos hacerlo automáticamente con deriving (Ord) donde en este caso los valores quedarán ordenados tal y como se hayan declarado en el tipo de forma que Valor1<Valor2<Valor3<...<ValorN.

5.2. TIPO UNIÓN.

El tipo unión es aquel en el que se declaran varios constructores de datos y cada constructor toma un argumento. Sería del siguiente modo:

```
data NombreDelTipo = Constructor1 arg1 | Constructor2 arg2 | ... | ConstructorN argN
```

Aquí caben destacar dos detalles importantes. El primero de ellos es que podemos utilizar los constructores como patrones para definir funciones y distinguir casos. El segundo de ellos es que los constructores no son más que funciones que toman un argumento de un tipo y devuelven un valor del nuevo tipo definido, Constructor :: tipo → NombreDelTipoNuevo

5.3. TIPO PRODUCTO.

Este tipo es similar al productor cartesiano en matemáticas. Consta de un único constructor y varios argumentos que puede recibir. Sigue el modelo siguiente:

```
data NombreDelTipo = Constructor Arg1 Arg2 ... ArgN
```

En ocasiones puede que los argumentos coincidan en su tipo y no quede muy claro cual es el papel de cada argumento, por ejemplo, data Persona = P String String Int. En ella P es el constructor, String es un nombre, otro String es un apellido y el Int es la edad. Aquí viene la posible confusión ¿es nombre y apellido o apellido y nombre? Para ello podemos dar nombre a nuevos tipos más simples del siguiente modo type Nombre = String, type Apellido = String, type Edad = Int. Así podríamos redefinir data Persona = P Nombre Apellido Edad y eliminar las posibles confusiones.

Esto que acabamos de describir se conoce como sinónimo de tipos, consiste en definir un nuevo tipo asignándole un tipo ya existente como en este caso eran String e Int.

⁷ Es aplicable a todos los tipos algebraicos, no únicamente al enumerado.

5.4. TIPO GENERAL.

En el tipo general combinamos el tipo unión y el tipo producto, así tendremos varios constructores con varios argumentos. Seguirá el modelo siguiente y lo explicado en las secciones anteriores también le es aplicable.

```
data TipoGeneral = Constructor_1 Arg_1 .. Arg_n
                  | ...
                  | Constructor_m Arg_1 .. Arg_k
```

5.5. TIPO PARAMETRIZADO.

Un tipo parametrizado es aquel el cual recibe otro tipo como parámetro, es algo similar a las clases genéricas de Java. Por ejemplo en Java podíamos tener Map(<Integer, String>), aquí podemos tener data TipoParametrizado a b = Constructor a b.

Observación: Los argumentos recibidos por el constructor son del mismo tipo que el recibido en la parametrización.

Dos tipos parametrizados muy empleados en Haskell son Either y Maybe (el segundo es bastante más empleado que el primero). Either nos sirve para pasar un tipo a o un tipo b dependiendo de cuando nos convenga emplear uno u otro. Maybe nos es de gran ayuda para funciones que son definidas parcialmente (hay valores para los que no están definidas). Conseguimos evitar lanzar errores en funciones, así se puede devolver un elemento del tipo a o simplemente no devolver nada en los casos en que no se defina la función. Se definen del siguiente modo:

```
data Either a b = left a | right b

data Maybe a = Nothing
              | Just a
```

Aprovechamos para acabar este capítulo mostrando un ejemplo de la utilidad del tipo Maybe y además una función que también puede resultar interesante tener en mente para posibles ejercicios.

```
buscar :: Eq a => a -> [(a, b)] -> Maybe b -- predefinida como lookup
buscar _ [] = Nothing
buscar x ((k, v) : ps)
  | x == k = Just v
  | otherwise = buscar x ps
```

CAPÍTULO 3. TIPOS ABSTRACTOS DE DATOS.

1. INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS.

Un TAD (Tipo Abstracto de Datos) es un concepto empleado en programación centrado en separar la implementación interna del tipo y las operaciones que se pueden realizar con él. En Java, por ejemplo, creamos una clase y ocultamos los componentes internos haciéndolos privados. Lo que realmente importa de estos tipos abstractos de datos no es como han sido implementados, sino lo que podemos hacer con ellos y cómo utilizarlos.

Un ejemplo básico es el tipo Double. Conocemos las operaciones que se pueden realizar con el tipo Double y la complejidad de estas, pero no nos hace falta saber cómo se han implementado.

Así, un TAD se integra como un tipo de datos más cualquiera en el lenguaje y puede emplearse de la misma forma.

Para implementar un TAD se necesitan dos cosas. Una interfaz o signatura y unos axiomas. En la interfaz o signatura se definen y describen las operaciones que pueden realizarse con el tipo. Mientras, en los axiomas se definen ciertas relaciones que se deben cumplir entre los operadores.

2. MÓDULOS, INTERFACES Y AXIOMAS.

2.1. MÓDULOS.

Los módulos son propios de Haskell. Se entiende un módulo como un conjunto de tipos, clases, funciones relacionadas definidas en un mismo fichero. Así los módulos pueden utilizarse para dividir un programa en varios módulos facilitando el trabajo de organización y diseño de programas complejos, permite también el trabajo de varias personas a la vez en diferentes módulos y la utilización de módulos para la construcción de otros módulos.

El nombre del fichero y del módulo deben coincidir al igual que pasaba con las clases en Java. El fichero comienza con la declaración del módulo y las funciones que este va a exportar siguiendo el esquema del siguiente ejemplo. Una vez declarado el módulo se importan otros módulos que se vayan a utilizar en este y después se definen los tipos, funciones y demás elementos del módulo.

```
module DataStructures.Stack.LinearStack
( Stack
, empty
, isEmpty
, push
, pop
, top
) where

import Data.List(intercalate)
import Test.QuickCheck
```

Por otro lado, los clientes deberán importar los módulos que desean utilizar, para ello deberán hacer uso de la sentencia import, aunque en ocasiones, esto puede llevar a ciertas confusiones en caso de que dos módulos distintos tengan funciones que se llamen igual, para hacer distinciones entre módulos y evitar estas ambigüedades se importan añadiendo un nombre por el que se llamará a las funciones de dicho módulo. Vemos un ejemplo en el que podríamos llamar a S.función y Q.función de forma que no haya lugar a confusiones.

```
import qualified DataStructures.Stack.LinearStack as S
import qualified DataStructures.Queue.LinearQueue as Q
```

2.2. INTERFACES.

Las interfaces son propias de Java. Ya se han estudiado en Programación Orientada a Objetos y estas básicamente son una abstracción en la que se describen operaciones que serán soportadas por una clase que implemente dicha interfaz. Nos servirán en estos casos para implementar una misma estructura de datos de formas diferentes. Por ejemplo, una pila puede implementarse con un ArrayList o con LinkedList, así puede hacerse una interfaz Stack que describa las funciones que se implementarán en las clases ArrayStack y LinkedStack.

2.3. AXIOMAS.

Como antes hemos explicado, un TAD requiere también de unos axiomas. Vamos a ver brevemente cómo se construyen dichos axiomas en un caso genérico.

Lo primero de todo es distinguir qué elementos descritos en el módulo son constructores y qué son otros operadores como transformadores o selectores. Lo segundo que debemos hacer para construir los axiomas es resolver la pregunta siguiente para cada operador sobre cada constructor (siempre y cuando dicho operador sea aplicable sobre dicho constructor): ¿Qué devuelve ‘operador’ al aplicarlo a ‘constructor’? Veamos un ejemplo sencillo con las listas. Tenemos los constructores ($x:xs$) y $[]$. Sea el operador null que devuelve si una lista está vacía o no. null sobre $[]$ debe devolver True, null sobre $(x:xs)$ debe devolver falso.

Así podemos definir una función para comprobar con quickCheck cada una de estas propiedades y finalmente hacer una prueba para chequear que todas se verifican con un bloque do.

3. EL TAD STACK (PILA).

3.1. SIGNATURA O INTERFAZ.

Una pila (stack en inglés) es muy similar a una lista, pero tratándola como un objeto en vertical. Formalmente, es una estructura de datos contenedora que organiza los elementos con una política LIFO (Last In First Out) en la que se definen las siguientes operaciones básicas:

- push: coloca un nuevo elemento en la cima del Stack.
- pop: elimina el elemento situado en la cima del Stack.
- top: devuelve el elemento situado en la cima del Stack sin eliminarlo.
- isEmpty: comprueba si el Stack no contiene elementos.
- empty: construye una Stack vacía

3.2. AXIOMAS PARA STACK.

Ya hemos visto antes cómo definir los axiomas en general, para una Stack nos quedan los siguientes axiomas con los que podemos definir la siguiente prueba de chequeo.

```
ax_isEmpty_empty :: Bool
ax_isEmpty_empty = isEmpty empty == True

ax_isEmpty_push :: a -> Stack a -> Bool
ax_isEmpty_push x s = isEmpty (push x s) == False

ax_top_push :: Eq a => a -> Stack a -> Bool
ax_top_push x s = top (push x s) == x

ax_pop_push :: Eq a => a -> Stack a -> Bool
ax_pop_push x s = pop (push x s) == s
```

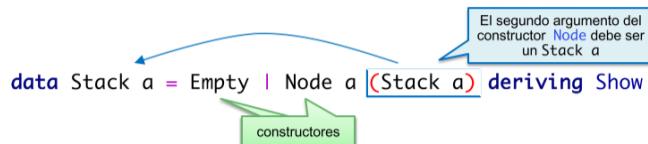
```
type T = Integer -- Char, String, etc.

check_isEmpty_empty = quickCheck (ax_isEmpty_empty :: Bool)
check_isEmpty_push = quickCheck (ax_isEmpty_push :: T -> Stack T -> Bool)
check_top_push = quickCheck (ax_top_push :: T -> Stack T -> Bool)
check_pop_push = quickCheck (ax_pop_push :: T -> Stack T -> Bool)

check_Stack = do
    check_isEmpty_empty
    check_isEmpty_push
    check_top_push
    check_pop_push
```

3.3. IMPLEMENTACIÓN.

Vamos a representar un Stack como un tipo de datos recursivo. La mayoría de TAD que representemos serán tipos de datos recursivos, pero ¿qué es un tipo de datos recursivo? La respuesta es intuitiva, son tipos de datos que se definen recursivamente, es decir, que en el constructor pueden llamarse a sí mismos, de otra manera, pueden aparecer como argumento de alguno de sus constructores. Es más sencillo de ver con la construcción del Stack.



Vamos a analizar un poco más en profundidad esta declaración. Comenzamos empleando la sentencia "data" para declarar el tipo "Stack" y además se le pasa un parámetro a. ¿Qué es este parámetro? Estamos construyendo una pila, pero ¿una pila de qué? El parámetro a es un tipo que indica el tipo de datos que se almacenará en la pila. Después vemos dos constructores: "Empty" y "Node". Con el primero de ellos simplemente definimos una pila vacía y no hace falta pasarle ningún argumento. Con el segundo construimos ya una pila con elementos. Para ello se le pasan dos argumentos, un elemento del tipo a y otro elemento que es otra pila de elementos del tipo a. Así el elemento que se le pasa se incorpora a la cabeza de la pila ya existente que se pasa también. Aquí observamos que pasamos como argumento al constructor de Stack otro tipo Stack. De ahí que digamos que es un tipo de datos recursivo.

```
module DataStructures.Stack.LinearStack
( Stack
, empty
, isEmpty
, push
, pop
, top
) where

import Data.List(intercalate)
import Test.QuickCheck

data Stack a = Empty | Node a (Stack a)

empty :: Stack a
empty = Empty

isEmpty :: Stack a -> Bool
isEmpty Empty = True
isEmpty _      = False

push :: a -> Stack a -> Stack a
push x s = Node x s

top :: Stack a -> a
top Empty     = error "top on empty stack"
top (Node x s) = x

pop :: Stack a -> Stack a
pop Empty     = error "pop on empty stack"
pop (Node x s) = s

-- Showing a stack
instance (Show a) -> Show (Stack a) where
    show s = "LinearStack(" ++ intercalate "," (aux s) ++ ")"
    where
        aux Empty      = []
        aux (Node x s) = show x : aux s

-- Stack equality
instance (Eq a) -> Eq (Stack a) where
    Empty      == Empty      = True
    (Node x s) == (Node x' s') = x==x' && s==s'
    _          == _          = False

-- This instance is used by QuickCheck to generate random stacks
instance (Arbitrary a) -> Arbitrary (Stack a) where
    arbitrary = do
        xs <- listOf arbitrary
        return (foldr push empty xs)
```

Vamos a añadir una imagen de la implementación del tipo LinearStack en Haskell para tener un ejemplo con la implementación completa de un tipo de datos abstracto con representación recursiva.

3.4. STACK EN JAVA.

Además de estudiarlo en Haskell, también podemos trasladar esta teoría equivalente a Java para su implementación. A modo de resumen tendremos una interfaz Stack y después dos clases que la implementarán. Estas son ArrayStack y LinkedStack. Se recomienda ver la implementación de estas dos en las diapositivas de la asignatura. En especial LinkedStack es de especial interés pues nos servirá de introducción para otros temas en los que tipos de datos enlazados toman protagonismo. Comprenden una explicación desde la diapositiva 21 hasta la 54.

4. EL TAD QUEUE (COLA).

4.1. SIGNATURA O INTERFAZ.

Una cola también es muy similar a una lista en cuanto a concepto. No obstante, formalmente una cola es una estructura de datos que organiza sus elementos siguiendo una política FIFO, es decir, el primero que entra, es el primero que sale. Se definen las siguientes funciones básicas:

- enqueue: añade un elemento al final de la cola.
- dequeue: extrae el elemento que se encuentra al principio de la cola.
- first: devuelve el elemento que se encuentra al principio de la cola sin extraerlo.
- isEmpty: dice si la cola está vacía.
- empty: construye una cola vacía.

4.2. AXIOMAS PARA QUEUE.

Como ya explicamos de modo general, debemos construir los axiomas en función de qué nos devuelvan los operadores al aplicarlos a los constructores. Así se obtienen:

```
module DataStructures.Queue.QueueAxioms
  (ax1,ax2,ax3,ax4,ax5,ax6,queueCheckAxioms) where

import DataStructures.Queue.LinearQueue    -- Implementation to use in tests
import Test.QuickCheck

ax1      =           isEmpty empty
ax2 x q =           not (isEmpty (enqueue x q))

ax3 x   =           first (enqueue x empty) == x
ax4 x q = not (isEmpty q) ==> first (enqueue x q) == first q

ax5 x   =           dequeue (enqueue x empty) == empty
ax6 x q = not (isEmpty q) ==> dequeue (enqueue x q) == enqueue x (dequeue q)

type Elem = Int  -- Tipo a usar en las pruebas

queueCheckAxioms = do
  quickCheck (ax1 :: Bool)
  quickCheck (ax2 :: Elem -> Queue Elem -> Bool)
  .
  .
  quickCheck (ax6 :: Elem -> Queue Elem -> Property)
```

4.3. IMPLEMENTACIÓN.

La implementación de Queue es sencilla y prácticamente calcada la de Stack. No hay más que ver que sus constructores siguen el mismo patrón.

```
data Queue a = Empty | Node a (Queue a)
```

Véase la implementación completa en las diapositivas de la asignatura (61-62).

4.4. QUEUE EN JAVA.

Al igual que con las Stack teníamos una interfaz y dos posibles implementaciones en Java, para las Queue ocurre igual, tenemos una interfaz y dos implementaciones, una con Arrays y otro con nodos enlazados. Para mayor comprensión véase la explicación e implementación existente en las diapositivas del tema. Proceden desde la número 64 hasta la 97.

5. EL TAD SET (CONJUNTO).

5.1. SIGNATURA O INTERFAZ.

Con este TAD Set se busca representar conjuntos finitos matemáticos. Así es una estructura de datos que almacena sus elementos sin que tengan un índice asociado y sin repetir ningún elemento. Implementa las siguientes funciones.

- empty: construye un conjunto vacío.
- isEmpty: dice si un conjunto es vacío.
- insert: dados un elemento y un conjunto devuelve el conjunto con el elemento añadido.
- delete: dados un elemento y un conjunto devuelve el conjunto con el elemento eliminado.
- isElem: dado un elemento y un conjunto dice si dicho elemento pertenece al conjunto.

5.2. AXIOMAS PARA SETS.

```
type Elem = Int -- Test axioms using sets of Ints

setAxioms = do
    quickCheck (ax1 :: Property)
    quickCheck (ax2 :: Elem -> Set Elem -> Property)
    quickCheck (ax3 :: Elem -> Set Elem -> Property)
    quickCheck (ax4 :: Elem -> Elem -> Set Elem -> Property)
    quickCheck (ax5 :: Elem -> Property)
    quickCheck (ax6 :: Elem -> Elem -> Set Elem -> Property)
    quickCheck (ax7 :: Elem -> Property)
    quickCheck (ax8 :: Elem -> Elem -> Set Elem -> Property)
    quickCheck (ax9 :: Elem -> Elem -> Set Elem -> Property)

    -- the empty set is empty
    ax1 = True ==> isEmpty empty

    -- insert always returns non-empty sets
    ax2 x s = True ==> not (isEmpty (insert x s))

    -- an element is only included once in a set
    ax3 x s = True ==> insert x (insert x s) == insert x s

    -- order of insertion is not important
    ax4 x y s = True ==> insert x (insert y s) == insert y (insert x s)

    -- no element is included in empty set
    ax5 x = True ==> not (isElem x empty)

    -- only elements previously inserted are included in set
    ax6 x y s = True ==> isElem y (insert x s) == (x==y) || isElem y s

    -- deleting a non-included element does not modify set
    ax7 x = True ==> delete x empty == empty

    -- deleting last inserted element returns set before insertion
    ax8 x y s = (x==y) ==> delete x (insert y s) == delete x s

    -- delete and insert commute
    ax9 x y s = (x==y) ==> delete x (insert y s) == insert y (delete x s)
```

5.3. IMPLEMENTACIÓN.

Los conjuntos pueden tener muchas implementaciones distintas, pueden ser lineales, donde se mantienen elementos sin repeticiones ni orden; lineales ordenados, donde no hay repeticiones, pero sí se sigue un orden; o por delegación sobre listas. Para nosotros la más fácil de ver será la lineal ordenada. No obstante, se verán más.

5.4. SETS EN JAVA.

Ocurre igual que en Haskell, aunque ya se ha visto en Java la interfaz Set y algunas de sus implementaciones.⁸

⁸ Se deja al lector oír las diapositivas del tema sobre las distintas implementaciones que no se añaden aquí al ser prácticamente todo código.

6. LISTAS E ITERADORES EN JAVA.

6.1. LISTAS.

Ya hemos visto las listas en Haskell en el tema anterior. Ahora vamos a verlas en Java. Aquí no vienen predefinidas, están implementadas como un TAD que hay que importar. El TAD List tiene una interfaz definida con las siguientes operaciones definidas. Así son las clases ArrayList y LinkedList principalmente quienes la implementan.⁹

```
public interface List<T> {
    boolean isEmpty();
    int size();
    T get(int i);
    void set(int i, T elem);
    void insert(int i, T elem);
    void remove(int i);
}
```

6.2. ITERADORES Y PLEGADOS.

Un iterador es un patrón de diseño que abstrae el proceso de recorrer los elementos de una colección, es decir, es una función que recorre uno por uno los elementos habidos en una lista, una pila, una cola... en general de cualquier estructura de datos en la cual tenga sentido e interés explorar sus elementos de esta manera.

En Haskell empleamos el concepto de plegado, este se realiza mediante la función fold asociada a la estructura de datos que queremos recorrer. Vemos un ejemplo de plegado para conjuntos

```
fold :: (a -> b -> b) -> b -> Set a -> b
fold f z s = fun s
  where
    fun Empty      = z
    fun (Node x s) = x `f` (fun s)
```

Básicamente se pasa una función, una solución para el caso base de la estructura vacía y la propia estructura, con esto fold nos devuelve un elemento del tipo b que es resultado de aplicar una transformación a cada elemento de la estructura y unirla al resultado de acumulado de aplicarla al resto de la lista.

Por su parte los iteradores en Java son objetos que permiten recorrer los elementos de una colección de objetos de cierto tipo. Hay dos interfaces principales para esto. La interfaz Iterator y la interfaz Iterable. La interfaz iterable hace que cualquier clase que la implemente deba añadir un método iterator que devuelva un objeto del tipo Iterator<T>, esto es, debe devolver un objeto de una clase que implemente la interfaz Iterator. Para ello podemos definir una clase privada dentro de nuestra clase Iterable que implemente Iterator. Esto le obliga a definir los métodos hasNext y next. Véasamos a verlo más fácil con el siguiente ejemplo.¹⁰

```
public class MyClass implements Iterable<Type> {
    ...
    public Iterator<Type> iterator() {
        return new MyClassIterator();
    }
}

private class MyClassIterator implements Iterator<Type> {
    ...
    public boolean hasNext() {
        ...
    }

    public Type next() {
        ...
    }
}
```

Este es el tipo concreto de los elementos devueltos al iterar

Método que crea un iterador (MyClassIterator implementa Iterator)

Clase anidada que implementa el iterador para recorrer los elementos de MyClass

La clase anidada puede acceder a los atributos de MyClass 😊

⁹ Estas implementaciones ya se vieron en la asignatura Programación Orientada a Objetos.

¹⁰ Termínese la implementación en Java del tipo ArraySet para aclarar el concepto siguiendo este esquema.

CAPÍTULO 4. ÁRBOLES.

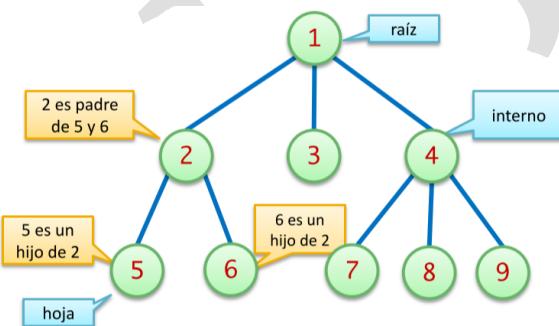
1. INTRODUCCIÓN A ÁRBOLES.

En este apartado vamos a introducir la idea de árbol y una idea de implementación básica en Haskell. Básicamente los árboles de los que hablamos son árboles matemáticos, esto es, un grafo con una única componente conexa y sin ciclos.

Los árboles constituyen una importante estructura de datos puesto que aportan más información que una relación lineal donde solamente se establecía un orden, aquí podemos establecer relaciones jerárquicas. Además, muchos algoritmos son más eficientes con esta estructura, por ejemplo los vistos en la asignatura de Análisis y Diseño de Algoritmos de vuelta atrás o ramificación y poda.

1.1. TERMINOLOGÍA.

En un árbol tenemos distintos elementos que se relacionan entre sí básicamente. A cada uno de estos elementos se le denomina nodo. Como ya hemos dicho, entre los nodos se establecen relaciones jerárquicas. Hay un nodo que se encuentra en la cima del árbol, a este lo llamaremos raíz. Así se dice que cada nodo tiene un parente y se emplea la terminología “padre-hijo”. Por último, hay algunos nodos que no tienen hijos, a estos se les llaman hojas. Vamos a verlo en un ejemplo.



También se definen un arco o arista como el par de nodos (u,v) tales que u es parente de v , y un camino como una secuencia de nodos tales que la secuencia es vacía, hay un único nodo o entre cada par de nodos existe una arista.

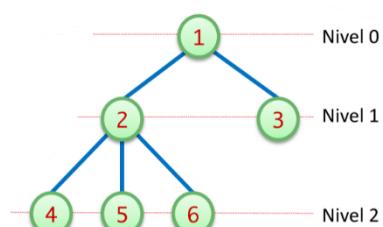
1.2. IDEA DE IMPLEMENTACIÓN DE ÁRBOL.

Tal y como hemos dicho antes, podemos entender un árbol como un conjunto de nodos de forma que o bien el árbol es vacío o bien hay un nodo raíz y el resto de nodos pueden entenderse como subárboles que “cuelgan” de la raíz. Esto nos lleva a la siguiente definición en Haskell:

```
data Tree a = Empty | Node a [Tree a] deriving Show
```

Constructor para árbol vacío Valor en raíz Lista de subárboles hijos

Así se estructuran los árboles en niveles, se parte de nivel 0 en la raíz hasta nivel n en las hojas. De esta forma se define la altura de un árbol como el número de niveles, es decir, si se alcanza hasta el nivel n , tenemos altura $n+1$ (puesto que también hay nivel 0).



2. ÁRBOLES BINARIOS.

Un árbol binario es un árbol en el que cada nodo tiene a lo sumo dos hijos de forma que se distinguirán hijo izquierdo e hijo derecho. Además, dentro de estos árboles binarios se distinguen los siguientes tipos:

- Árbol binario auténtico: es aquel en que cada nodo tiene dos hijos excepto los nodos hoja.
- Árbol binario completo: es aquel en que todos los niveles están completos salvo el último en que quizás haya un único nodo hoja en el lado izquierdo, en ellos se debe dar que la diferencia entre altura y niveles sea menor o igual a dos.
- Árbol binario perfecto: es aquel árbol binario que es auténtico y que todas sus hojas están al mismo nivel.

Esto nos lleva a definir en Haskell un árbol binario de la siguiente forma.

```
data TreeB a = EmptyB | NodeB a (TreeB a) (TreeB a) deriving Show
```

Además, los árboles binarios definen funciones como `atLevel` o `PathsTo`¹¹ que son muy empleadas en operaciones con dichas estructuras de datos. No obstante, estas pueden quedarse cortas, como hicimos en TAD's y en listas, una opción muy importante es la de recorrer elemento a elemento la estructura. Para ello, disponemos de las siguientes opciones.

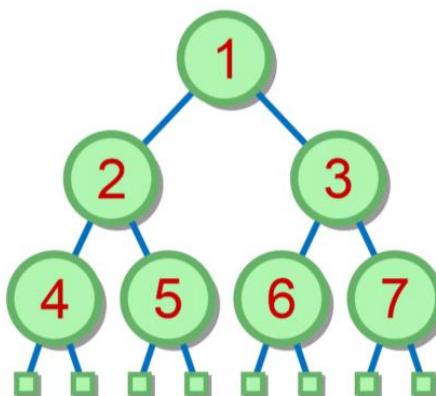
2.1. RECORRIDO DE ÁRBOLES BINARIOS.

Por recorrido de árboles binarios se entiende, como es obvio, la iteración elemento a elemento del árbol. Un árbol se puede recorrer de distintas formas. Se distinguen distintos tipos de recorridos según el orden en que se visiten los elementos. Podemos encontrar:

- Pre-order: se recorre el árbol de arriba a abajo y de izquierda a derecha.
- In-order: es un recorrido en profundidad, esto es, se empieza desde los nodos hoja de la izquierda y se recorren el resto de nodos pasando primero por el que más a la izquierda esté independientemente del nivel.
- Post-order: se recorre el árbol de izquierda a derecha y de abajo a arriba.
- En anchura: se recorre el árbol por niveles.

Las tres primeras se entenderán mejor con el siguiente ejemplo gráfico:

```
Main> preOrderB tree4
[1,2,4,5,3,6,7]
Main> inOrderB tree4
[4,2,5,1,6,3,7]
Main> postOrderB tree4
[4,5,2,6,7,3,1]
```



¹¹ Véase la implementación en diapositivas 13 y 14 del tema 4.

3. COLAS CON PRIORIDAD Y MONTÍCULOS BINARIOS.

3.1. COLAS CON PRIORIDAD LINEALES.

En el capítulo 3 ya vimos las colas. Ahora vamos a añadirles la coletilla “con prioridad”. Ya vimos que las colas seguían una política FIFO, ahora vamos a modificar esta política y vamos a darle prioridad a ciertos elementos según se establezca una relación entre ellos. Así considerando lo que sería una relación de orden como puede ser distinguir elementos por edad, antigüedad, valor... En general, siempre tienen prioridad los que menos tienen, es decir, los menores en la relación de orden serán los que tengan mayor prioridad, por tanto, se insertarán primero los menores y en caso de igualdad en la relación se seguirá el criterio FIFO para desempate.

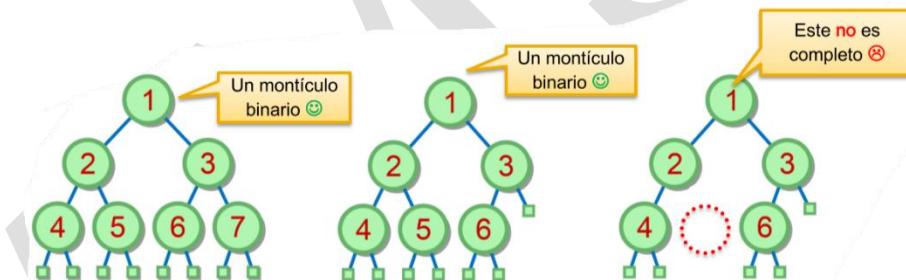
Así el principal cambio de la implementación en colas con prioridad lineales es el método enqueue. Para este, en Haskell, se sigue el código de la imagen.

```
enqueue :: (Ord a) => a -> PQueue a -> PQueue a  
enqueue x Empty = Node x Empty  
enqueue x (Node y q)  
| x < y      = Node x (Node y q)  
| otherwise    = Node y (enqueue x q)
```

Este método tiene un orden lineal. Puede mejorarse con la implementación con árboles. Vamos para ello a ver antes los montículos binarios.

3.2. MONTÍCULOS BINARIOS.

Se define un montículo binario como un árbol binario completo en que el valor de cada nodo hijo es mayor o igual al de su padre, es decir, cualquier camino desde la raíz hasta una hoja será una cadena ascendente.



Así, cuando vayamos a insertar, eliminar o realizar cualquier operación sobre un montículo binario debemos hacerlo de forma que se siga manteniendo esta propiedad de que el valor de cualquier hijo sea mayor o igual al del padre¹². Vamos a ver para ello entonces cómo ha de hacerse la inserción de un elemento en un montículo binario.

INSERCIÓN EN MONTÍCULOS BINARIOS.

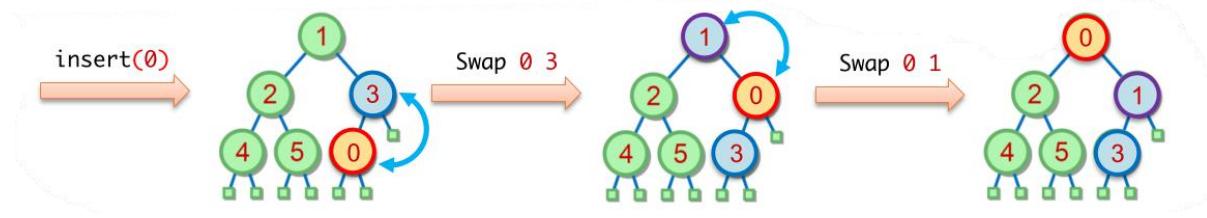
La inserción constará de dos fases. En cada una de ellas se asegura mantener una propiedad. En la primera aseguraremos que el árbol binario siga siendo completo. En la segunda aseguraremos la propiedad HOP.

En la primera fase colocamos el nuevo elemento en la primera posición libre que haya, es decir, en el último nivel, a la derecha del último nodo que haya, o en caso de que este último nivel esté completo, a la izquierda en un nuevo nivel.

En la segunda fase debemos mantener el orden, así mientras el padre sea mayor al hijo, vamos a intercambiar sus posiciones.

¹² A partir de ahora, a esta propiedad la llamaremos HOP (Heap-Order property)

Vemos un ejemplo gráfico de cómo sería el proceso:

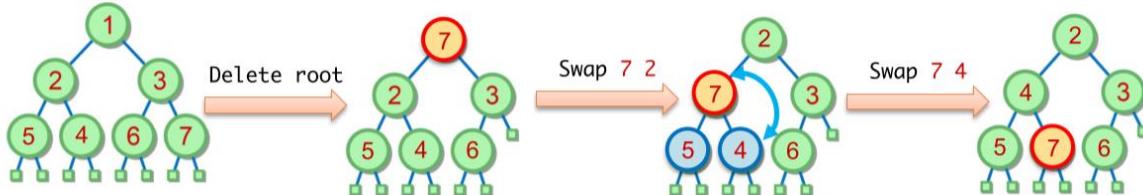


ELIMINACIÓN DE LA RAÍZ EN MONTÍCULOS BINARIOS.

Para eliminar la raíz en montículos binarios también debemos asegurar que se sigan manteniendo las dos propiedades de ser completo y HOP. Para ello también se distinguen dos fases en la eliminación.

La primera fase consiste en pasar último nodo a la raíz reescribiendo el valor que hubiese en ella, es decir, pasar a la raíz el nodo que más a la derecha esté en el último nivel. De esta forma conseguimos garantizar que el resultado sigue siendo un árbol binario completo.

La segunda fase consiste en hundir ahora la raíz en el árbol. Para ello tomamos los dos hijos que tenga el nodo que ahora hemos colocado en la raíz (al cual llamaremos u), de ellos nos quedamos con el menor (al cual llamaremos mu). Mientras u sea mayor a mu, vamos a intercambiar sus papeles y así hasta que finalice. Un ejemplo gráfico sería el siguiente.



CONVERSIÓN DE MONTÍCULOS EN LISTAS.

Esta función, aunque es sencilla de explicar, tiene cierta importancia. Para colocar un montículo en una lista simplemente nos dedicamos a hacerlo por niveles, así primero se pasa el nivel 0, el nivel 1, el nivel 2, ..., así hasta el nivel n. De esta forma podemos localizar padres e hijos fácilmente. La raíz siempre tendrá el índice 0. Luego, para el resto de nodos, si estamos en la posición i del array; su parent se localizará en la posición $(i-1)/2$, su hijo izquierdo en la posición $2*i+1$ y su hijo derecho será el siguiente, es decir, estará en la posición $2*i+2$.

Gracias a esto último podemos implementar montículos binarios de forma sencilla en Java vía Arrays. Véase la implementación en las diapositivas (38-45).

De esta forma podemos implementar en Java colas con prioridad basadas en montículos binarios de forma que la implementación es trivial pues delega todas las operaciones en las operaciones del montículo y reducimos los costes lineales que pasan a ser logarítmicos. Véase la implementación de estas en las diapositivas (46-47).

4. ÁRBOLES BINARIOS DE BUSQUEDA (BST).

Un árbol binario de búsqueda se define como un árbol binario en el cual para cada nodo v se verifican:

- i) todos los nodos del subárbol izquierdo son menores o iguales que v
- ii) todos los nodos del subárbol derecho son mayores o iguales que v

Estas dos propiedades son muy importantes para facilitarnos la búsqueda en árboles. No obstante, al igual que ocurría con los montículos, debemos asegurarnos de que cuando insertamos, eliminamos o hacemos alguna modificación sobre el BST siga satisfaciendo las propiedades i) y ii). Veamos cómo hacer una inserción, eliminación, búsqueda o recorrido en un BST.

INSERCIÓN EN UN BST.

Es sencillo. Simplemente debemos ir comparando el elemento a insertar con la raíz del árbol y después llamar recursivamente para insertar a la izquierda o derecha. Vamos a razonarlo un poco más paso a paso. Sean X el elemento a insertar, A la raíz del árbol, LT el subárbol de la izquierda y RT el subárbol de la derecha. Distinguimos varios casos. Si X es igual que A, entonces basta con cambiar A por X y mantener los subárboles. Si X es menor que A, entonces X debe estar en el subárbol izquierdo, para ello llamamos recursivamente a insertar X en el subárbol LT, en caso contrario en que X sea mayor que A, el elemento se insertará en el subárbol RT. Así nos quedaría un código como este.

```
insert :: (Ord a) => a -> BST a -> BST a
insert x' Empty = Node x' Empty Empty
insert x' (Node x lt rt)
| x'==x      = Node x' lt rt
| x'<x       = Node x (insert x' lt) rt
| otherwise   = Node x lt (insert x' rt)
```

Observación: en caso de hacer una inserción de elementos ordenados nos quedaría un árbol degenerado en el que solamente se inserta en una de las ramas, pero nunca en la otra.

Observación: al recorrer el árbol con un inOrder nos devolverá una lista de sus elementos ordenados de menor a mayor.

BÚSQUEDA EN BST.

Para realizar una búsqueda en un árbol BST es muy sencilla. Simplemente nos guiamos según el nodo raíz, si el elemento que buscamos es mayor que él lo buscamos recursivamente a la derecha, en caso de ser menor, se busca a la izquierda y en caso de ser igual ya lo habríamos encontrado. Sigue el siguiente código.

De igual forma, para encontrar el mayor o menor elemento del árbol lo tendríamos muy fácil, el menor será el más a la izquierda y el mayor el más a la derecha. Se localizarían con los

```
minim :: BST a -> a
minim Empty = error "minim on empty tree"
minim (Node x Empty rt) = x
minim (Node x lt rt) = minim lt
```

```
search :: (Ord a) => a -> BST a -> Maybe a
search x' Empty = Nothing
search x' (Node x lt rt)
| x'==x      = Just x
| x'<x       = search x' lt
| otherwise   = search x' rt
```

```
isElem :: (Ord a) => a -> BST a -> Bool
isElem x t = isJust (search x t)
```

```
maxim :: BST a -> a
maxim Empty = error "maxim on empty tree"
maxim (Node x lt Empty) = x
maxim (Node x lt rt) = maxim rt
```

ELIMINACIÓN DE UN ELEMENTO EN UN BST.

Para eliminar un elemento podemos encontrar más problemas. Aquí dependerá la forma de proceder según el nodo a eliminar tenga dos hijos, uno o ninguno. Comenzamos localizando el nodo a eliminar de forma parecida a cómo hacíamos en la inserción. En caso de ser una hoja (no tener hijos) simplemente se elimina y ya está. En caso de que solo tenga un hijo simplemente podemos sustituir al padre por el hijo. Por último, en caso de tener dos hijos deberá ser el mínimo elemento del hijo derecho el que sustituya al padre ó bien el máximo elemento del hijo izquierdo. Vemos un ejemplo gráfico.



Vemos ahora el código necesario para implementar esta función. Necesitará de dos funciones auxiliares: split y combine.

```

split :: BST a -> (a,BST a)
split (Node x Empty rt) = (x,rt)
split (Node x lt  rt)  = (m,Node x lt' rt)
  where (m,lt') = split lt

combine :: BST a -> BST a -> BST a
combine Empty rt      = rt
combine lt   Empty    = lt
combine lt   rt       = Node x' lt rt'
  where (x',rt') = split rt

delete :: (Ord a) => a -> BST a -> BST a
delete x' Empty     = Empty
delete x' (Node x lt rt)
| x'==x      = combine lt rt
| x'<x       = Node x (delete x' lt) rt
| otherwise   = Node x lt (delete x' rt)

```

PLEGADOS PARA BST.

Al igual que podíamos recorrer un árbol en pre-order, in-order, post-order y en profundidad podemos hacer plegados de las mismas formas. Nos interesan especialmente las tres primeras y en particular in-order. La implementación utiliza una función auxiliar traversal. Veámosla.

```

traversal :: ((b -> b) -> (b -> b) -> (b -> b) -> (b -> b)) ->
            (a -> b -> b) -> b -> BST a -> b
traversal order f z t = aux t z
  where
    aux Empty      = id
    aux (Node x lt rt) = order (f x) (aux lt) (aux rt)

foldInOrder :: (a -> b -> b) -> b -> BST a -> b
foldInOrder = traversal (\xf lf rf -> lf . xf . rf)

```

4.1. BST EN JAVA.

En este apartado hemos explicado el razonamiento genérico y en algunos casos hemos visto la implementación en Haskell. Ahora vamos a hacerlo en Java.

Utilizamos una clase privada Tree para apoyarnos y luego trabajar recursivamente.

```

private static class Tree<C,D> {
  private C key;
  private D value;
  private Tree<C,D> left, right;

  public Tree(C k, D v) {
    key = k;
    value = v;
    left = null;
    right = null;
  }

  private Tree<K,V> root;
  private int size;

  public BST() {
    root = null;
    size = 0;
  }
}

```

Dejamos esta imagen en la que se establece la clase privada mencionada y el constructor de la clase BST. El resto de implementación se deja al lector como ejercicio pues se basa exclusivamente en llamar a una función auxiliar que se implementará de forma privada utilizando recursividad siguiendo el mismo razonamiento que en Haskell. Dejamos el siguiente ejemplo de esto.

```

public V search(K key) {
  return BST.searchRec(root, key);
}

private static <C extends Comparable<? Super C>, D>
  D searchRec(Tree<C,D> tree, C key) {
  if (tree == null)
    return null;
  else if (key.compareTo(tree.key) == 0)
    return tree.value;
  else if (key.compareTo(tree.key) < 0)
    return searchRec(tree.left, key);
  else
    return searchRec(tree.right, key);
}

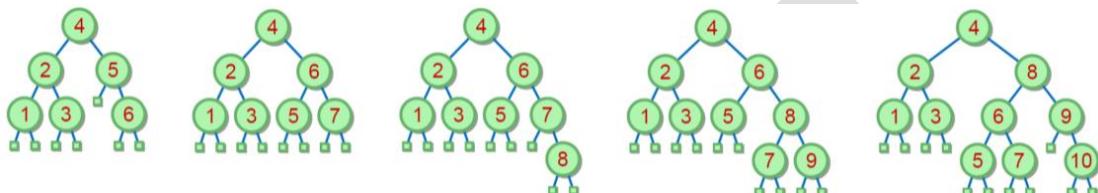
```

5. ÁRBOLES BALANCEADOS EN ALTURA (AVL).

Se llama árbol AVL a todos aquellos árboles binarios de búsqueda que además satisfacen la propiedad de estar balanceado en altura, esto es que, para cada nodo, las alturas de sus hijos difieren a lo sumo en 1. Es por este motivo la altura se convierte en una propiedad vital en los árboles AVL. Debido esa gran importancia vamos a tenerla en cuenta en la construcción del TAD y vamos a considerar esta altura como un parámetro más.

```
data AVL a = Empty | Node a Int (AVL a) (AVL a)
```

En cada nodo mantenemos su altura

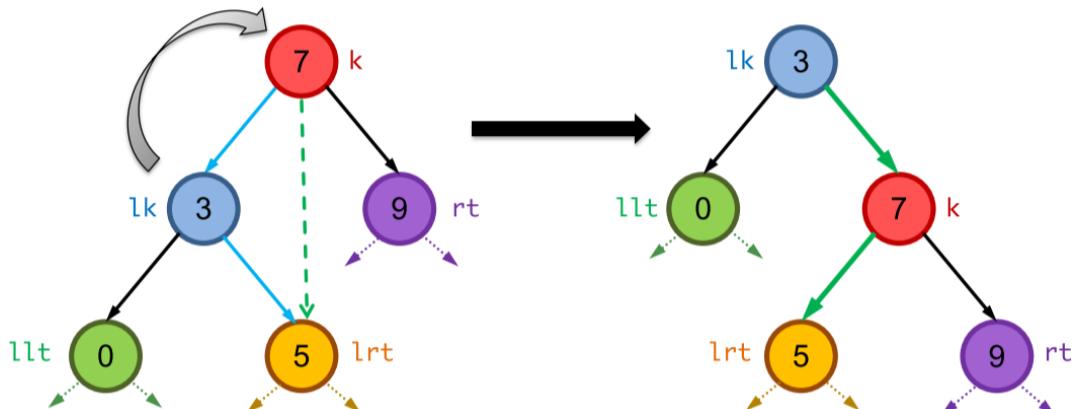


En esta imagen podemos apreciar distintos árboles AVL. Es importante destacar que para que un árbol sea un AVL, cada subárbol debe ser también un AVL puesto que la propiedad de ser balanceado se ha de verificar para cada nodo.

Observación: esta nueva propiedad añadida a los BST nos garantiza que la complejidad operacional de insert y delete sea de orden logarítmico. Así conseguimos una mayor eficiencia.

5.1. ROTACIONES EN AVLs.

Una rotación de un árbol AVL es una operación que consiste en mover un nodo hacia arriba de forma que arrastra a una rama hacia arriba, otra la arrastra hacia abajo y hay un subárbol que se desconecta de un nodo para conectarse a otro. Gráficamente el proceso se entiende mucho mejor.



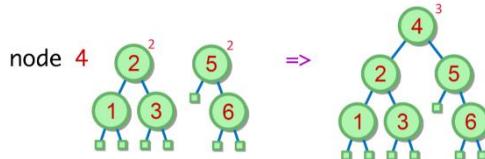
¹³El nodo que sube hacia arriba es el 3. Arrastra consigo hacia arriba a la rama del 0. A su vez, se arrastra hacia abajo a la rama del siete. Por último, es el subárbol del 5 el que se desconecta del nodo del 3 para conectarse al del 7. En este caso la rotación ha sido hacia la derecha, pero puede darse también una rotación a la izquierda. Para implementar estos métodos nos basta con abstraernos del ejemplo anterior y reescribirlo para un caso general. Así obtendríamos:

```
rotR :: AVL a -> AVL a
rotR (Node k h (Node lk lh llt lrt) rt) = node lk llt (node k lrt rt)
rotL :: AVL a -> AVL a
rotL (Node k h lt (Node rk rh rlt rrt)) = node rk (node k lt rlt) rrt
```

¹³ Para facilitar la comprensión de estas funciones se recomienda seguir el proceso gráfico de rotación con varios ejemplos para finalmente lograr abstraerse.

En los códigos anteriores node es un constructor auxiliar para árboles AVL que consiste en lo siguiente:

```
node :: a -> AVL a -> AVL a -> AVL a
node k lt rt = Node k h lt rt
  where h = 1 + max (height lt) (height rt)
```



5.2. INSERCIÓN EN ÁRBOLES AVL.

Cuando nos proponemos insertar un elemento en un árbol AVL debemos tener mucho cuidado. Debemos asegurar que siga siendo un AVL tras la inserción. Para ello debemos garantizar es un BST y que está balanceado. Esto implica mantener el orden y la diferencia de alturas entre subárboles. Por tanto, al hacer una inserción normal como hacíamos en BST se descompensaría seguramente el árbol. Así que hay que añadir algún método para reestablecer esta propiedad. Este método se apoya en las rotaciones antes vistas, nos sirven para reestablecer el balanceo en un AVL. Podemos saber si el árbol se inclina a derecha o a izquierda gracias a las siguientes funciones:

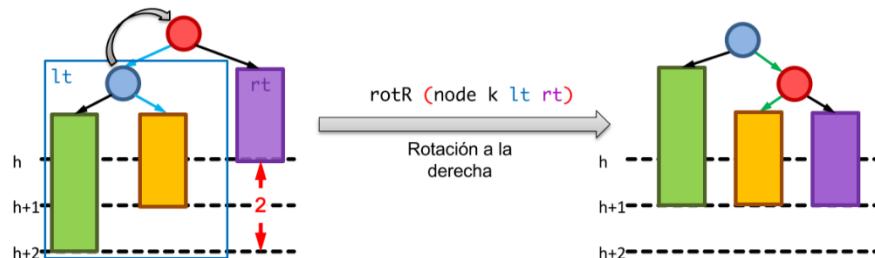
```
leftLeaning :: AVL a -> Bool
leftLeaning (Node x h lt rt) = height lt >= height rt

rightLeaning :: AVL a -> Bool
rightLeaning (Node x h lt rt) = height lt <= height rt
```

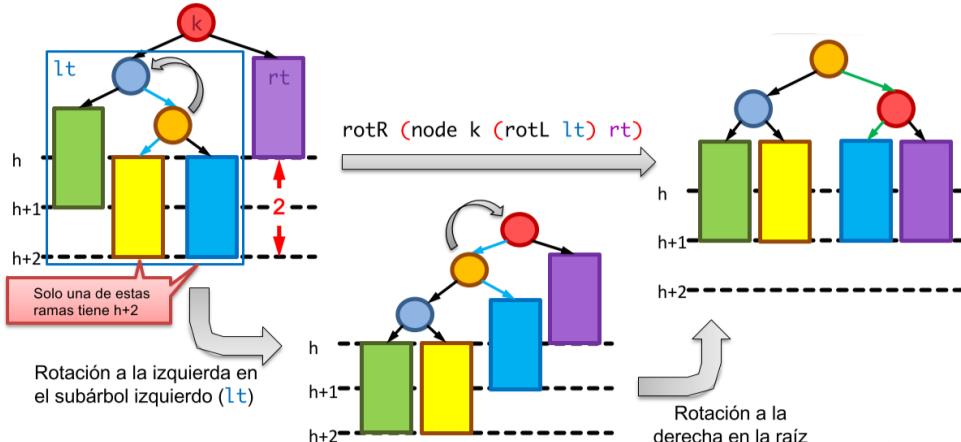
Podremos utilizarlas para restaurar el balanceo en la siguiente función: balance.

```
balance :: a -> AVL a -> AVL a -> AVL a
balance k lt rt
| (lh-rh > 1) && leftLeaning lt = rotR (node k lt rt)
| (lh-rh > 1)                  = rotR (node k (rotL lt) rt)
| (rh-lh > 1) && rightLeaning rt = rotL (node k lt rt)
| (rh-lh > 1)                  = rotL (node k lt (rotR rt))
| otherwise                      = node k lt rt
  where lh = height lt
        rh = height rt
```

Vamos a analizar los casos para entenderlo correctamente. En el primero de ellos, hay diferencia de alturas mayor a uno y además está inclinado a la izquierda. ¿Qué quiere decir esto? Que el problema está entre el subárbol izquierdo y el subárbol derecho y que con una sola rotación hacia la derecha bastaría para solucionarlo. Esto no se ve así de rápido. Hay que pararse a pensar. Que $lh-rh > 1$ nos indica que el árbol está balanceado hacia la izquierda. Por otro lado, $leftLeaning lt$ nos indica que dentro del subárbol izquierdo, también hay inclinación a la izquierda, por tanto estamos en la siguiente situación.



Ahora bien, en el segundo caso, si no hay $leftLeaning$ a la izquierda quiere decir el subárbol de la izquierda está "más bajo" que el de la derecha, pero que dentro de este subárbol de la izquierda, es el lado derecho el más bajo, es decir, estaríamos en un situación cómo la siguiente:



Observación: la opción de hacer un único giro a la derecha no serviría pues provocaría una nueva inclinación, pero esta vez en el lado derecho del árbol.

El razonamiento para los casos de la derecha es análogo y finalmente, cualquier otro caso implicaría que el árbol que se le ha pasado está balanceado y es un AVL, por tanto, no hay que hacer ninguna modificación.

De esta forma conseguimos que hacer la inserción en un AVL sea igual que en un BST pero asegurando que se verifique la propiedad de balanceo con la función balance. Se obtendría el siguiente código.

```
insert :: (Ord a) => a -> AVL a -> AVL a
insert k' Empty = node k' Empty Empty
insert k' (Node k h lt rt)
| k' == k      = Node k' h lt rt
| k' < k       = balance k (insert k' lt) rt
| otherwise     = balance k lt (insert k' rt)
```

Así hemos conseguido unificar en una función las dos fases para mantener las propiedades de un AVL. La primera fase sería la inserción como en un BST y la segunda restablecer el balanceo gracias a la llamada a balance.

Por último, recalcar que la función búsqueda es exactamente igual que en BST y la eliminación con la función delete sigue el mismo esquema que insert. Primero se realiza la eliminación como en BST y después se reestablece el balanceo con las llamadas a balance. Queda el código siguiente:

```
delete :: (Ord a) => a -> AVL a -> AVL a
delete k' Empty = Empty
delete k' (Node k h lt rt)
| k' == k      = combine lt rt
| k' < k       = balance k (delete k' lt) rt
| otherwise     = balance k lt (delete k' rt)

combine :: AVL a -> AVL a -> AVL a
combine Empty rt = rt
combine lt Empty = lt
combine lt rt   = balance k' lt rt'
where (k',rt') = split rt

-- Elimina y devuelve el mínimo elemento de un árbol
split :: AVL a -> (a, AVL a)
split (Node k h Empty rt) = (k, rt)
split (Node k h lt rt)   = (k', balance k lt' rt')
where (k',lt') = split lt
```

Observación: las funciones combine y split también han tenido que ser redefinidas para asegurar que se verifique la propiedad de balanceo.

6. DICCIONARIOS.

Un diccionario es un TAD que nos permite establecer relaciones entre un conjunto del tipo A y otro conjunto del tipo B. Básicamente estas relaciones son asignar a los elementos de A (a los que llamaremos claves, (Key)) un elemento del tipo B (a los que llamaremos valores, (Value)).

Nota: esta notación de key, value nos debería recordar a cómo se hacían mapas en Java.

Se puede definir un diccionario fácilmente mediante árboles AVL en los que cada nodo guarda una relación de elementos del tipo A con otro del tipo B. Así en Haskell, deberemos definir este tipo de datos en el módulo del diccionario. Nos quedaría algo como el siguiente código.

```
module DataStructures.Dictionary.AVLDictionary
  ( Dict
  , empty
  , isEmpty
  , insert
  , valueOf
  ) where

import qualified DataStructures.BinarySearchTree.AVL as T

data Rel a b = a :-> b

instance (Eq a) => Eq (Rel a b) where
  (k :-> _) == (k' :-> _) = (k == k')

instance (Ord a) => Ord (Rel a b) where
  (k :-> _) <= (k' :-> _) = (k <= k')

data Dict a b = D (T.AVL (Rel a b))

empty :: Dict a b
empty = D T.empty

isEmpty :: Dict a b -> Bool
isEmpty (D avl) = T.isEmpty avl

insert :: (Ord a) => a -> b -> Dict a b -> Dict a b
insert k v (D avl) = D (T.insert (k :-> v) avl)

valueOf :: (Ord a) => a -> Dict a b -> Maybe b
valueOf k (D avl) =
  case T.search (k :-> undefined) avl of
    Nothing      -> Nothing
    Just (_ :-> v) -> Just v
```

Además del código del lateral (que básicamente define el tipo de datos Rel para la relación entre tipos y el tipo Diccionario que delega todas sus operaciones en las operaciones para árboles AVL vistas en el apartado anterior) necesitamos como vimos en el capítulo 3 unos axiomas. Son los siguientes:

```
isEmpty empty
not (isEmpty (insert k v d))
valueOf k empty == Nothing
k==k' ==> valueOf k (insert k' v' d) == Just v'
k/=k' ==> valueOf k (insert k' v' d) == valueOf k d
```

CAPÍTULO 5. GRAFOS.

1. DEFINICIONES BÁSICAS.

A lo largo de este capítulo trataremos con grafos y operaciones que implementen. Para ello comenzamos definiendo qué es un grafo.

Definición: Llamamos grafo simple G a un par ordenado $G = (V, E)$ donde V es un conjunto cuyos elementos se denominan vértices y E es un conjunto formado por subconjuntos de dos elementos de V , a dichos elementos se les denomina aristas.

Observación: en el caso de grafos simples tenemos que $E \subseteq \{\{u, v\} | u, v \in V, u \neq v\}$, es decir; no hay aristas simples ni de un vértice en sí mismo. Además, son no dirigidas.

Definición: Llamamos multígrafo o grafo múltiple al par ordenado $G = (V, E)$ donde V es un conjunto de elementos denominados vértices y E es una familia de aristas. De esta forma, se admiten aristas múltiples y también lazos o bucles.

Definición: Llamamos dígrafo al par ordenado $G = (V, E)$ donde V es un conjunto de elementos denominados vértices y E es un conjunto de aristas orientadas, es decir, un dígrafo es un grafo simple con aristas dirigidas.

Definición: Sea $G = (V, E)$ un grafo. Sean u, v pertenecientes a V . Diremos que u y v son vértices adyacentes si existe una arista $e = \{u, v\}$. También diremos que e es incidente con u y v . También nos encontraremos más frecuentemente con el término sucesor en lugar de incidente. Así se dirá que u es sucesor de v y recíprocamente que v es sucesor de u .

Observación + Definición: En caso de ser G un dígrafo diremos que u es el vértice inicial y v es el vértice final. Además, se dirá que e es incidente a v (puesto que entra a él) y que e es saliente a u (puesto que sale de él). Por último, dada la arista $\{u, v\}$ se dirá que v es sucesor de u , pero no se dará el recíproco.

Definición: Sea G un grafo y sea u perteneciente a V . Llamaremos grados de u y lo denotaremos por $gr(u)$ ó $deg(u)$ al número de aristas incidentes con u .

Observación + Definición: En caso de ser G un dígrafo llamaremos grado de entrada al número de aristas incidentes a u y se llamará grado de salida al número de aristas salientes a u .

1.1. OTROS CONCEPTOS IMPORTANTES. CONEXIDAD.

Definición: Llamamos camino a un subgrafo en que todos sus vértices están conectados por aristas, podría decirse así que un camino es una secuencia de vértices conectados por aristas.

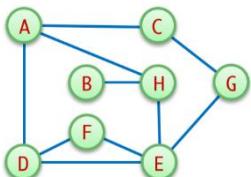
Definición: Llamaremos ciclo a un camino cerrado, es decir, el vértice inicial y final es el mismo y todas las aristas recorridas son distintas.

Definición: Decimos que un grafo es conexo si $\forall u, v \in V$ existe un camino entre ellos que los conecta.

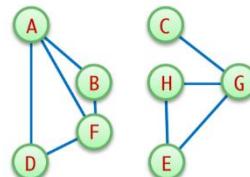
A continuación, vamos a definir componentes conexas. Vamos a prescindir aquí rigurosidad matemática pues nos haría falta definir el concepto de paseo y definir una relación de equivalencia, cosa algo larga y tediosa para la sencillez del concepto.

Definición alternativa: diremos que una componente de G conexa será todo aquel subgrafo maximal y conexo de G .

Vemos un ejemplo visual muy sencillo:



Tenemos el grafo L (de la izquierda) el cual posee una única componente conexa, por tanto, es conexo. Por otro lado, tenemos el grafo R (de la derecha) el cual posee dos componentes conexas y, por tanto, no es conexo.



1.2. ÁRBOLES DE EXPANSIÓN O ÁRBOLES RECUBRIDORES.

Observación: un árbol no es más que un grafo conexo y sin ciclos.

Definición: Sea G un grafo no dirigido y conexo. Decimos que T es un árbol de expansión o recubridor de G si T es un subgrafo conexo y sin ciclos de G . Dicho de otro modo, T es un árbol que contiene a todos los vértices de G .

Observación: Un árbol de expansión T de un grafo G ha de contener necesariamente a todos sus vértices, pero puede o no contener a todas sus aristas.

Definición: En caso de ser G un grafo no conexo, llamamos bosque recubridor o bosque de expansión al conjunto de n árboles de expansión, uno por cada una de las diferentes componentes conexas de G .

1.3. DENSIDAD.

Se define la densidad de un árbol como la proporción de aristas respecto al número de vértices.

Definición: Sea $G = (V, E)$ un grafo. Se dice que G es disperso si tiene una densidad baja. Se establece que un grafo tiene una densidad baja si $\#E \leq \#V * \log (\#V)$

Definición: Se dice que un grafo G es denso si no es disperso.

2. REPRESENTACIÓN DE GRAFOS EN HASKELL.

Vamos a comenzar a ver cómo se representa un grafo en Haskell. Comenzamos con la idea básica. Matemáticamente es un par ordenado (V, E) . Así necesitaremos un conjunto de vértices y otro de aristas. El problema lo tenemos principalmente para representar las aristas... Para ello haremos uso de los sucesores o adyacentes. Así tendremos una lista de elementos que almacene los vértices del grafo y una función que dado uno de esos vértices nos de quienes son sus sucesores. Así podemos empezar con la siguiente especificación.

```

data Graph a = G [a] (a -> [a])

module DataStructures.Graph.Graph
  ( Graph
  , Edge
  , Path
  , mkGraphSuc
  , mkGraphEdges
  , successors
  , vertices
  , edges
  , degree
  ) where
  
```

Además, en el módulo se definirán las siguientes funciones:

Dado un grafo G lo más sencillo de implementar son las últimas funciones: successors, vertices y degree. Dado un grafo (G vertex sucesores), la función successors aplicada a un elemento v simplemente devolverá sucesores v . En el caso de vertices devolverá vertex y en el de degree nos devolverá la longitud de la lista que devuelve successors. No obstante, para edges necesitaremos algo más. Hay que definir un nuevo tipo... Lo veremos después.

Ahora vamos a ver los constructores para un grafo. Hablamos de dos constructores en concreto: mkGraphSuc que será un poco más sencillo y mkGraphEdges, que no será tan directa.

Al primer constructor simplemente se le pasará una lista de vértices y una función suc. Así la definición de este constructor es la siguiente:

`mkGraphSuc :: [a] -> (a -> [a]) -> Graph a` Podría decirse que se veía venir. Es la definición trivial.
`mkGraphSuc vs suc = G vs suc`

Ahora vamos con mkGraphEdges. A este constructor se le aporta una lista de vértices y luego una lista con las aristas que serán del tipo Edge. Así se ha de definir una función suc a partir de la lista de aristas. Se hará como sigue:

```
type Edge a = (a,a)

mkGraphEdges :: (Eq a) => [a] -> [Edge a] -> Graph a
mkGraphEdges vs es = G vs suc
where
    suc v = nub ( [ y | (x,y) <- es, x==v]
                  ++
                  [ x | (x,y) <- es, y==v] )
```

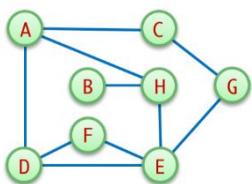
Observación: se añaden los vértices adyacentes como salientes, por tanto, estamos ante un grafo con aristas no dirigidas.

3. RECORRIDO DE GRAFOS.

Los recorridos, al igual que en otras estructuras de datos, son operaciones vitales y que intervienen en otras funciones más complejas. Un recorrido básicamente se dedica a recorrer todo el grafo visitando cada vértice. Existen varias formas de visitar los vértices. Las principales son en profundidad (Depth First Traversal) o en anchura (Breadth First Traversal). Vamos a verlos.

3.1. DEPTH FIRST TRAVERSAL (DFT)

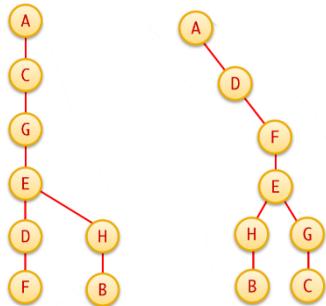
El algoritmo para recorrer un grafo en profundidad es sencillo. Se comienza visitando el primer vértice disponible y después para cada vértice aún no visitado en la lista de sucesores, se le visita. Así puede construirse recursivamente haciendo la función sobre un vértice, en lugar de sobre el árbol. Cabe destacar que estamos hablando de visitar vértices como concepto abstracto... visitar podría ser añadir el vértice a una lista, modificarlo, numerarlo... Vamos a ver un ejemplo de ejecución del algoritmo sobre el grafo anterior.



Supongamos que lo que hacemos en el recorrido (lo que hemos llamado visitar) es añadir a una lista. Comenzamos visitando el nodo A. Así comenzaríamos con la siguiente lista [A]. Ahora consideramos la lista de sucesores de A, es decir [C, H, D]. Para cada uno de ellos se visita si no han sido visitados anteriormente. Así proseguimos con C pues no se ha visitado. Se añade a la lista obteniendo [A, C]. La lista de sucesores de C es [A, G]. Como A ya ha sido visitado pasamos a G. Lo añadimos, nuestro resultado será [A, C, G]. Ahora seguimos con G. Sus sucesores son {C, E}. C ya ha sido visitado, pasamos a E. Se añade, [A, C, G, E]. Los sucesores de E son [G, H, D, F]. C ya ha sido visitado, luego se añade H. Tenemos [A, C, G, E, H]. Los sucesores de H son {B, A}. Se añade B. Tenemos [A, C, G, E, H, B]. B no tiene sucesores así que volvemos con backtracking a los sucesores de H. A también fue ya visitado, así que volvemos también con backtracking a los sucesores de E. Se añade ahora D. Sus sucesores son [A, F, E]. A ya fue visitado, se añade F. Tenemos [A, C, G, E, H, B, D, F]. Los sucesores de F ya han sido todos visitados, volvemos ahora por backtracking hacia atrás y como todos han sido ya visitados se acaba la ejecución.

Observación: a la vez que se visita cada nodo puede añadirse a un árbol de expansión del grafo en construcción.

Observación: el resultado de la función DFT puede ser distinto en función de cómo se asigne u ordene la lista de sucesores.



Aquí tenemos un ejemplo. En el grafo anterior se podía haber construido un árbol mínimo de expansión como el más a la izquierda, pero si hubiésemos hecho el recorrido con otro orden en las listas de sucesores, empezando por D en lugar de por C en este caso, se podría haber obtenido un árbol como el que hay más a la derecha.

En cualquier caso, DFT siempre es llamada una vez por cada vértice y que cada arista se hace dos veces, una por vértices.

IMPLEMENTACIÓN CON STACK.

Para la implementación de esta función necesitamos una estructura donde almacenar los vértices que no han sido visitados aún, esta estructura puede ser una pila. Así el algoritmo será el siguiente:

- Apilar el vértice inicial en la pila S
- Mientras S no sea vacía
 - Quitar v de la pila S
 - Si v no ha sido visitado
 - Visitar v
 - Apilar sus sucesores no visitados.

Además de este pseudocódigo vamos a añadir a continuación la implementación en Haskell de la función DFT para la construcción de una lista con todos los vértices del grafo

```

import DataStructures.Stack.LinearStack
import qualified DataStructures.Set.BSTSet as S
import DataStructures.Graph.Graph

pushAll :: Stack a -> [a] -> Stack a
pushAll s xs = foldr push s xs

dft :: (Ord a) => Graph a -> a -> [a]
dft g v0 = aux S.empty (push v0 empty)
  where
    aux visited stack
      | isEmpty stack = [] -- fin de recorrido
      | v `S.isElem` visited = aux visited stack' -- v ha sido visitado
      | otherwise         = v : aux visited' (pushAll stack' us)
    where
      v = top stack
      stack' = pop stack
      visited' = S.insert v visited
      us = [ u | u <- successors g v, u `S.notIsElem` visited ]
  
```

3.2. BREADTH FIRST TRAVERSAL BFT. (RECORRIDO EN AMPLITUD).

La diferencia con DFT es esencial. Mientras en DFT seguimos avanzando siempre que podíamos en "niveles"¹⁴ del grafo. Ahora vamos a avanzar a través los sucesores, no en niveles.

¹⁴ En grafos no se han definido niveles pues no se puede para cualquier grafo, si se hizo para árboles. Entiéndase la idea de nivel referido a árboles en lugar de grafos.

El algoritmo será muy similar, se visitará un primer vértice, y una vez sea visitado, se añadirán a la estructura correspondiente todos los sucesores del último visitado. Ahora en vez de visitar vértice a vértice e ir añadiendo sucesores, vamos a visitar todos los vértices habidos y después se añadirán los sucesores. Así se prosigue hasta visitar todos los vértices. Veamos el algoritmo.

- Inicialmente encolamos el vértice A en la cola Q
- Mientras Q no sea vacía
 - Desencolar el vértice v.
 - Si v no ha sido visitado
 - Visitarlo
 - Encolar sus sucesores no visitados

Observación: el algoritmo es básicamente igual que el DFT, sin embargo, cambia la estructura de datos utilizada, antes utilizamos una pila, ahora estamos usando una cola. Esta diferencia es la que nos proporciona la diferencia entre recorrer en profundidad o en amplitud.

Vamos ahora a ver el código en Haskell.

```
import import DataStructures.Queue.TwoListsQueue
import qualified DataStructures.Set.BSTSet as S
import DataStructures.Graph.Graph

enqueueAll :: Queue a -> [a] -> Queue a
enqueueAll s xs = foldr enqueue s xs

bft :: (Ord a) => Graph a -> a -> [a]
bft g v0 = aux S.empty (enqueue v0 empty)
  where
    aux visited queue
      | isEmpty queue           = []
      | v `S.isElem` visited    = aux visited queue' -- v fue visitado
      | otherwise                = v : aux visited' (enqueueAll queue' us)
    where
      v = first queue
      queue' = dequeue queue
      visited' = S.insert v visited
      us = [ u | u <- successors g v, u `S.notIsElem` visited ]
```

OBSERVACIÓN IMPORTANTE: el recorrido en BFT recorre toda una componente conexa del grafo donde esté el primer elemento que se añade. Esto nos facilita que el árbol de búsqueda formado en el recorrido tenga los caminos más cortos desde v a cualquier otro vértice. Así nos interesaría hacer caminos a partir de BFT y el vértice del origen del camino (o el final, en verdad solo vamos a obtener un camino mínimo, por eso da igual).

4. CAMINOS A VÉRTICES.

Hemos visto en el apartado anterior como recorrer un grafo. No obstante, es un recorrido un poco pobre, empezamos y acabamos en un lugar que nosotros no decidimos... nos gustaría recorrer el grafo desde un vértice que elijamos hasta otro vértice, es decir, nos gustaría poder construir caminos en el grafo. Para esto nos puede venir muy bien un diccionario, que para cada arista nos guarde cual es la siguiente en visitar.

4.1. CAMINOS A VÉRTICES CON DFT.

Para recorrer un árbol en profundidad necesitamos solamente una pila, ahora vamos a añadir el diccionario y alguna que otra modificación sobre el algoritmo para conseguir los caminos. Pensamos primero ¿cómo podemos cambiar el algoritmo? Ahora nos interesa tener nodos y su sucesor o un nodo y su antecesor para ir construyendo el camino más fácilmente. Así que donde

antes apilábamos vértices, ahora serán aristas. Después de esto básicamente vamos a hacer un DFT pero con aristas. Así podremos llegar a completar un diccionario que nos dirá como podemos llegar a cada vértice y a partir de cuál llegamos, así construiremos recursivamente hacia atrás el camino. Así dividiremos el proceso en dos partes: construcción del diccionario y reconstrucción del camino. Vamos a ver un pseudocódigo para la construcción del diccionario y el código completo en Haskell.

- Comenzamos apilando en la pila S una arista inicial $A \rightarrow A$
- Mientras S no sea vacía
 - Quitar $w \rightarrow v$ de S
 - Si v no ha sido visitado
 - Visitar v
 - Añadir $v \leftarrow w$ al diccionario D
 - Apilar en S las aristas de sucesores no visitados.

Una vez se acabe comienza la reconstrucción del camino, para ello nos bastará con buscar recursivamente hasta que el nodo que tenemos y el buscábamos eran el mismo. Vemos el código completo, aunque en mayor tamaño la última parte de reconstrucción, puesto que no se ha suministrado pseudocódigo de ella.

```

import DataStructures.Stack.LinearStack
import qualified DataStructures.Set.BSTSet as S
import qualified DataStructures.Dictionary.BSTDictionary as D
import DataStructures.Graph.Graph

data AnEdge a = a :> a -- w :> v significa que llegamos a v desde w
-- la siguiente devuelve los caminos desde un vértice
-- en una visita DFT
dftPaths :: (Ord a) => Graph a -> a -> [Path a]
dftPaths g v0 = aux S.empty (push (v0 :> v0) empty) D.empty
where
  aux visited stack dict
    | isEmpty stack = [] -- fin de recorrido
    | v `S.isElem` visited = aux visited stack' dict -- v fue visitado
    | otherwise =
        pathFromTo v0 v dict' :-- devolvemos un camino desde v0 hasta v
        aux visited' (pushAll stack' es) dict'
  where
    w :> v = top stack
    stack' = pop stack
    visited' = S.insert v visited
    dict' = D.insert v w dict -- el parent de v es w
    es = [ v :> u | u <- successors g v, u `S.notIsElem` visited ]

-- reconstrucción de un camino desde v0 hasta w
-- a través de un diccionario
pathFromTo :: (Ord a) => a -> a -> D.Dictionary a a -> [a]
pathFromTo v0 w dict = reverse (aux w)
  where
    aux w
      | w == v0 = [w]
      | otherwise = w : aux u
      where Just u = D.valueOf w dict

```

4.2. CAMINOS A VÉRTICES CON BFT

Para hacer caminos a partir de un BFT en lugar de un DFT se hace exactamente igual. Construimos un diccionario repitiendo el algoritmo del BFT, pero almacenando aristas en la cola. Después hay que recomponer el camino a partir del diccionario construido. La función pathFromTo va a reutilizarse igual. Vamos a ver el código en Haskell.

```

import import DataStructures.Queue.TwoListsQueue
import qualified DataStructures.Set.BSTSet as S
import DataStructures.Graph.Graph
import qualified DataStructures.Dictionary.BSTDDictionary as D

data BiEdge a = a :> a -- w :> v vamos a v desde w

bftPaths :: (Ord a) => Graph a -> a -> [Path a]
bftPaths g v0 = aux S.empty (enqueue (v0 :> v0) empty) D.empty
where
  aux visited queue dict
    | isEmpty queue = [] -- fin de recorrido
    | v `S.isElem` visited = aux visited queue' dict -- v fue visitado
    | otherwise =
        pathFromTo v0 v dict' :-- v fue visitado: devolvemos un camino
        aux visited' (enqueueAll queue' es) dict'
  where
    w :> v = first queue
    queue' = dequeue queue
    visited' = S.insert v visited
    dict' = D.insert v w dict -- parent of v is w
    es = [ v :> u | u <- successors g v, u `S.notIsElem` visited ]

```

4.3. OTRAS UTILIDADES DE BFT Y DFT.

A continuación, se van a mostrar algunas aplicaciones útiles de los recorridos BFT y DFT.

Podemos asegurar la existencia de un camino entre dos vértices de un grafo viendo si está en la lista que devuelve la función, sea con DFT o BFT. Además, podemos devolver el propio camino con la función pathFromTo, o si en concreto utilizamos pathFromTo a partir de un BFT, podemos devolver el camino más corto entre dos vértices.

Otra utilidad es encontrar todas las componentes conexas de un grafo aplicando reiteradamente un DFT sobre el grafo y distintos elementos de él.

Por último, vamos a hablar de la detección de ciclos. Podemos detectar ciclos si detectamos algún camino entre v y otro vértice conectado a v. Así podemos pasar a hablar de grafos bipartitos.

Definición: se define un grafo bipartito como aquél en el que se puede establecer una partición en dos subconjuntos de forma que no existe ninguna arista entre vértices del mismo subconjunto.

Nota: esta última definición es equivalente a un grafo en el que no existen ciclos de longitud impar.

5. GRAFOS DIRIGIDOS

A lo largo de este tema hemos estado hablando sobre grafos simples, en ningún momento hemos hablado de dirección de las aristas. Vamos ahora a trabajar con ello. Recordamos la definición dada al comienzo del tema de grafo dirigido.

Definición: Llamamos dígrafo al par ordenado $G = (V, E)$ donde V es un conjunto de elementos denominados vértices y E es un conjunto de aristas orientadas, es decir, un dígrafo es un grafo simple con aristas dirigidas.

Simplificando mucho la definición, las aristas ahora tienen origen y destino. Así dada una arista $\{v, w\}$ o también representada como $v \rightarrow w$ se define v como la fuente de la arista, w como el destino y como ya anticipamos al comienzo del tema, se dice que w es sucesor de v y que v es predecesor o antecesor de w.

También se recuerdan las definiciones de grado de salida, grado de entrada. Además, notamos que seguimos trabajando con caminos y ciclos con la única modificación de que ahora las aristas son dirigidas y eso cambia a los sucesores de un vértice, pero no cambia nada más.

5.1. REPRESENTACIÓN DE GRAFOS DIRIGIDOS O DIGRAFOS EN HASKELL.

La representación de grafos dirigidos es muy similar a la que ya teníamos para grafos simples. Comenzamos con los constructores. Ya vimos que solamente necesitábamos una lista con los vértices que hay en el grafo y una función sucesores. Así, lo que cambiarán no son los constructores, si no la función sucesores, que ahora deberá vigilar el sentido de las aristas.

Por ello tenemos de nuevo los dos constructores ya conocidos: el primero para una lista de vértices y la función sucesores, y el segundo en el que se pase una lista de vértices y otra de aristas dirigidas.

También habrá más funciones definidas casi trivialmente de forma casi calcada a como ya se definían para grafos simples. Estas funciones son: successors, que dado un grafo y un vértice de este nos devolverá el resultado que devuelva la función sucesores del grafo aplicada al vértice v; predecesors, esta es nueva, dado un grafo y un elemento v devolverá los elementos w que aparezcan en el grafo y tales que v aparezca en la lista de sucesores de w; vertices, ya hablamos de ella al comienzo del tema; outDegree, es el grado de salida, devolverá la longitud de la lista de sucesores; por último, inDegree, grado de entrada, devuelve la longitud de los predecesores.

Vemos ahora la implementación de estos.

```

data DiGraph a = DG [a] (a -> [a])

mkDiGraphSuc :: [a] -> (a -> [a]) -> DiGraph a
mkDiGraphSuc vs suc = DG vs suc

data DiEdge a = a :-> a deriving Show

mkDiGraphEdges :: (Eq a) => [a] -> [DiEdge a] -> DiGraph a
mkDiGraphEdges vs es = DG vs suc
where
  suc v = [ y | x :-> y <- es, x==v ]
  successors :: DiGraph a -> a -> [a]
  successors (DG vs suc) v = suc v

  predecesors :: (Eq a) => DiGraph a -> a -> [a]
  predecesors (DG vs suc) v =
    [ w | w <- vs, v `elem` suc w ]

  vertices :: DiGraph a -> [a]
  vertices (DG vs suc) = vs

  outDegree :: DiGraph a -> a -> Int
  outDegree g v = length (successors g v)

  inDegree :: (Eq a) => DiGraph a -> a -> Int
  inDegree g v = length (predecesors g v)

```

5.2. RECORRIDOS Y EN DIGRAFOS.

Si nos paramos a pensar un poco observamos que lo único que cambia es cómo se dan los sucesores de un vértice... Esto quiere decir que nuestro algoritmo es totalmente válido, lo único que cambia es como se ha de implementar la función successors.

Observación: en verdad la implementación de successors es igual, la que cambia es la función suc dada para la construcción del grafo a partir de las aristas, es decir, la función que se le pasa al constructor trivial del grafo.

Por tanto, como los caminos se construían a partir de los recorridos tampoco cambian, se siguen haciendo a partir de un DFT o BFT si se buscan los caminos mínimos.

5.3. SER FUERTEMENTE CONEXO.

Recordamos que para grafos simples definimos un grafo conexo como aquel en el que para cada par de vértices del grafo existe un camino que los conecta, ¿qué diferencia hay con los grafos dirigidos? Esta diferencia está en un pequeño matiz. En grafos simples si un vértice v estaba conectado con otro vértice w , entonces w estaba conectado con v , ahora esto no se tiene por qué cumplir necesariamente. Puede ser que exista un camino de v a w pero que no se pueda recorrer el camino en sentido contrario. Podemos hacer un símil con las carreteras. En grafos simples las carreteras son de doble sentido, mientras que en los grafos dirigidos las carreteras tienen sentido único. Así pasamos a definir otro concepto usado para grafos dirigidos.

Definición: se dice que dos vértices u, v , estás fuertemente conectados si existe un camino dirigido que una u con v y otro que una v con u .

Definición: se llama componente fuertemente conexa al máximo conjunto de vértices en que todos están fuertemente conectados entre sí.

6. REPRESENTACIÓN MATEMÁTICA.

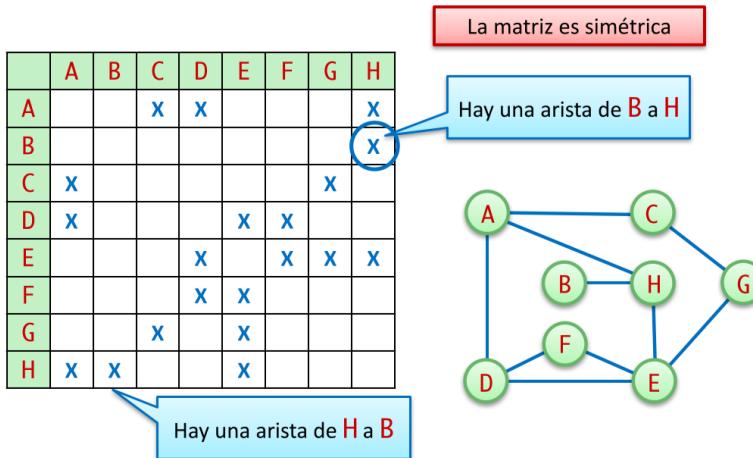
6.1. REPRESENTACIÓN CON MATRICES DE ADYACENCIA.

Hasta ahora hemos si pensábamos en un grafo nos imaginábamos un dibujito con vértices y aristas que unían algunos de ellos. Vamos a ver otra manera de representarlos más formalmente.

Definición: Sea $G = (V, E)$ un grafo simple con n vértices. Establecemos un orden en V y expresamos $V = \{v_1, \dots, v_n\}$. Definimos la matriz de adyacencia de G respecto del orden establecido en V como:

$$A_G = (a_{ij})_{i,j=1}^n$$

donde a_{ij} es 1 si la arista $\{v_i, v_j\}$ pertenece a E ó 0 en caso contrario.



Vemos un ejemplo de ello en la imagen de la izquierda.

Observación: como bien se indica en la imagen, para un grafo simple se tiene que la matriz de adyacencia es una matriz simétrica.

La hemos definido para un grafo simple, pero no para un digrafo. Vamos a ver cómo cambia la definición.

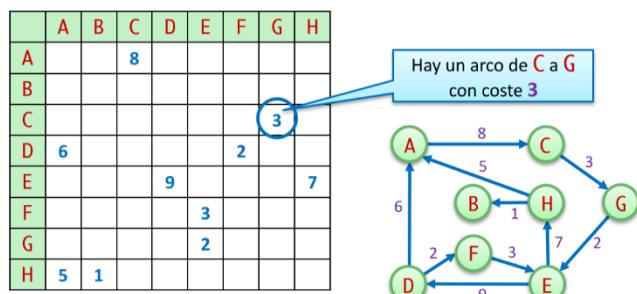
Definición: Sea $G = (V, E)$ un digrafo ó grafo dirigido con n vértices. Establecemos un orden en V y expresamos $V = \{v_1, \dots, v_n\}$. Definimos la matriz de adyacencia de G respecto del orden establecido en V como:

$$A_G = (a_{ij})_{i,j=1}^n$$

donde a_{ij} es 1 si la arista $(v_i \rightarrow v_j)$ pertenece a E ó 0 en caso contrario.

Observación + Definición: estamos representando aristas con valores numéricos, 1 si existe dicha arista o 0 si no existe, pero... ¿y si extendemos ese rango de definición a más valores posibles? De esta forma podemos añadir información sobre la arista, esta información añadida se conoce como peso o coste de una arista.

Ejemplo: imaginamos que, con el grafo, cada vértice representa una ciudad, las aristas son carreteras disponibles para viajar de una a ciudad a otra. Además, son aristas dirigidas. El peso de cada arista podría representar la distancia que se recorre para ir de una ciudad a la otra. Veamos un ejemplo de matriz de adyacencia en dígrafos con pesos.



Observación: esta matriz ya no es simétrica, esto es puesto que, al ser un grafo dirigido, la simetría en las aristas no es necesaria, es decir, que si existe $(u \rightarrow v)$ no implica que tenga que existir $(v \rightarrow u)$.

6.2. ANÁLISIS DE LA REPRESENTACIÓN CON MATRICES DE ADYACENCIA.

En este apartado vamos a ver si esta representación con matrices de adyacencia nos conviene o no. Destacamos primero que su eficiencia depende mucho de si el grafo es disperso o denso. Imaginamos un grafo con 1000 vértices, pero solamente 1 arista... esa matriz es una matriz que en su gran mayoría no contiene información útil y por tanto nos lleva a un desperdicio de espacio de memoria y tiempo al tener que recorrerla. No obstante, si el grafo es denso, sí estaríamos haciendo un uso eficiente de recursos.

Otro gran aspecto que destacar es que la representación no es tan sencilla como en las tablas que se han representado anteriormente. La mayoría de los lenguajes a la hora de indexar solamente admiten números enteros, en concreto naturales ó el 0. Así que no podremos tener indexando directamente a los vértices. Necesitamos establecer como dijimos en la definición un orden en V de forma que podamos abstraernos del valor que haya en el vértice y simplemente lo llamemos como vértice n-ésimo, así será el número n quién represente a ese vértice. Todo esto implica que además de la matriz de adyacencia deberemos tener un diccionario de vértices a enteros.

Por último, destacamos que las operaciones para determinar si un arco está en el grafo tiene orden 1 pero las operaciones que accedan a todos los sucesores de un vértice tendrán a lo poco orden n.

6.3. REPRESENTACIÓN CON LISTAS DE ADYACENCIA.

La representación con listas de adyacencia ya no es tan matemática como la anterior. Vamos simplemente a describir cómo se hace, no entraremos a definiciones formales. Un grafo se puede representar con listas de adyacencia fácilmente, para cada vértice se crea una lista donde se almacenan sus vértices adyacentes. Esta era más o menos la idea de la implementación con la función suc. Por su parte, para un digrafo en esta lista solamente se almacenarán los sucesores, no los antecesores. También se pueden representar grafos con pesos si en la lista en vez de almacenar únicamente vértices almacenamos pares (vértice, peso) o hagamos lo propio con alguna estructura de datos que nos lo permita.

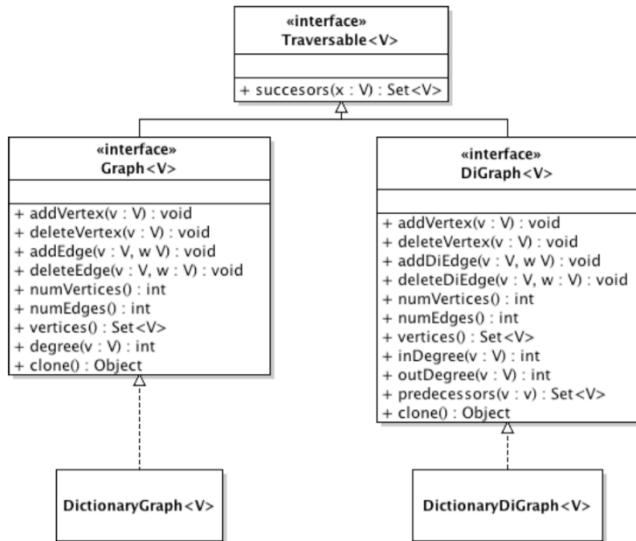


6.4. ANÁLISIS DE LA REPRESENTACIÓN CON LISTAS DE ADYACENCIA.

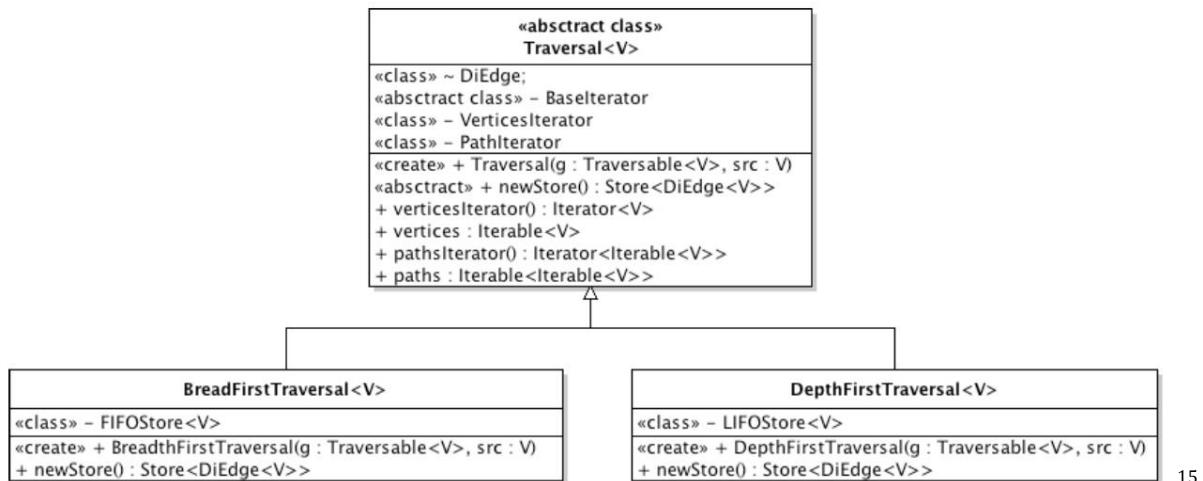
Destacamos que hace buen uso de memoria incluso con grafos dispersos, aunque si nos vamos al extremo contrario y utilizamos grafos muy densos y sin pesos se obtienen peores resultados que con matrices de adyacencia. Además, se invierten resultados para las funciones de existencia de aristas y de obtención de sucesores. A la hora de consultar si una arista existe debemos recorrer la lista, por tanto, puede llegar a tener orden lineal (n). Sin embargo, ahora para obtener los sucesores simplemente tenemos que acceder a una lista, luego, tiene orden 1.

7. GRAFOS Y DIGRAFOS EN JAVA.

Teóricamente ya hemos visto todo lo necesario para implementar grafos y dígrafos en Java, pues lo hemos ido explicando antes de implementar en Haskell. Ahora vamos a ver cómo hacer la implementación en Java. Lo primero que destacaremos es el diagrama UML. En él nos vamos a encontrar con varias interfaces. La primera de ellas es Traversable, generalmente esta suele servir para el recorrido, para ello nos pide que las clases finales que la implementen tengan en ellas la función successors. Después tendremos otras dos interfaces que extenderán a Traversable. La primera de ella es Graph para grafos simples y la segunda es Digraph para grafos dirigidos. Vemos por último las clases DictionaryGraph y DictionaryDiGraph donde faremos la implementación. Vamos a aportar dicho diagrama UML.



A la hora de realizar recorridos también va a seguir una idea similar, aunque en Java ya sabemos que no es tan simple como en Haskell, aquí hay multitud de clases e interfaces, vamos a ver también el diagrama UML para ello.



Así tenemos que tanto el BFT como el DFT implementan Traversal (que no traversable). Vértices es un método que almacena todos los vértices que se han visitado durante el recorrido, para recorrer todos esos vértices tenemos verticesIterator. De igual forma que con vertices pasa para los caminos. Tenemos paths que son los distintos caminos que se encuentran en un recorrido, por otro lado, pathsIterator nos sirve para recorrer dichos la lista con dichos caminos.

Además, cuando hablamos de recorridos pasamos a hablar de las componentes conexas del grafo. Ya vimos que podíamos utilizar dft o bft de un grafo y un vértice para devolver su componente conexa... pero si queremos todas las componentes en Haskell tenemos el siguiente código que en su momento no dijimos.

```

type ConnectedComponent a = [a]
dftConnectedComps :: (Ord a) => Graph a -> [ConnectedComponent a]
dftConnectedComps g = aux (vertices g)
where
    aux [] = []
    aux (v:vs) = comp : aux (vs \\ comp)
    where
        comp = dft g v

```

Lo que hacemos es coger la lista de vértices, calcular su componente conexa y llamar recursivamente para calcular componentes conexas, pero eliminando los vértices que ya estén en alguna componente.

¹⁵ No puedo dar más información pues tampoco hay mucho más en las diapositivas sobre esto.

Ahora vamos a ver cómo hacerlo en Java.

```
Set<V> component = new HashSet<O>;
for(V v : new BreadthFirstTraversal<O>(g, src).vertices<O>) {
    component.insert(v); // add to component
}
```

Para obtener la componente conexa en Java hay que hacer un poquito más. Para cada vértice del BFT (o DFT) se añade a un conjunto.

Pero para obtener todas las componentes conexas del grafo si es verdad que hay que trabajar un poco más de código, no es tan compacto como en Haskell.

```
public class ConnectedComponents<V> {
    private Set<Set<V>> components; // Components as sets of vertices
    private Dictionary<V, Integer> inComponent; // in which component is a vertex

    public ConnectedComponents(Graph<V> g) {
        components = new HashSet<O>();
        inComponent = new HashDictionary<O>();

        Set<V> unvisited = new HashSet<V>(); // no vertex has been visited yet
        for(V v : g.vertices<O>)
            unvisited.insert(v);

        for(int c = 0; !unvisited.isEmpty<O>(); c++) {
            V src = unvisited.iterator<O>.next(); // an unvisited vertex
            inComponent.insert(src, c); // store component number for src

            Set<V> component = new HashSet<O>;
            for(V v : new DepthFirstTraversal<O>(g, src).vertices<O>)
                component.insert(v); // add to component
            inComponent.insert(v, c); // store component number for v

            components.insert(component); // add component to set of components
            for(V v : component)
                unvisited.delete(v); // all vertex in component have been visited
        }
    }
}
```

Tendremos un conjunto de conjuntos donde almacenaremos todas las componentes conexas. Comenzamos creando un conjunto de vértices no visitados aún. Para cada no visitado se añade la componente al diccionario, se añaden los vértices a la componente y se acaban eliminando del conjunto de no visitados a los vértices de la componente ya añadida.

8. ORDEN TOPOLOGICO.

Definición: Sea $G = (V, E)$ un digrafo. Se define un orden topológico en él como la relación de orden total entre vértices tal que si existe la arista $(v \rightarrow w)$ entonces se tiene que $v < w$.

Observación: en un digrafo pueden haber varios y distintos órdenes topológicos. Además, en caso de ser cíclico el digrafo, no puede existir el orden topológico puesto que en tal caso se tendría que $v < w$ y que $w < v$.

Definición: llamamos fuente a aquel nodo que tiene grado de entrada 0, es decir, de él solo pueden salir aristas o ser un vértice aislado.

Definición: llamamos sumidero a aquel nodo que tiene grado de salida 0, es decir, a él solo pueden entrar aristas o ser un vértice aislado.

Así para construir un orden topológico lo que haremos será buscar fuentes en el grafo reiteradamente. El algoritmo será el siguiente

- Buscar fuente
- Añadir la fuente al orden
- Eliminar esa fuente y sus correspondientes aristas
- Repetir recursivamente hasta acabar o no poder seguir.

Así en Java para implementar el orden topológico tendríamos el siguiente código:

```

public class TopologicalSorting<V> {
    private List<V> order;
    private boolean cycle;
    public TopologicalSorting(DiGraph<V> graph) {
        order = new ArrayList<V>();
        cycle = false;
        DiGraph<V> g = (DiGraph<V>) graph.clone();
        while(!cycle && !g.vertices().isEmpty()) {
            V next = null;
            for(V v : g.vertices())
                if(g.inDegree(v)==0) {
                    next = v;
                    break;
                }
            if(next!=null) {
                order.append(next);
                //also deletes corresponding edges
                g.deleteVertex(next);
            } else
                cycle = true;
        }
    }
}

public boolean hasCycle() {
    return cycle;
}

public List<V> order() {
    return cycle ? null : order;
}

```

Lo que hacemos es que mientras aún nos queden vértices en el grafo por visitar y mientras no haya ciclos vamos a buscar una fuente, cuando la tengamos comprobamos si es nula (es decir, que en verdad no hemos encontrado fuente). En este caso hemos encontrado un ciclo. En caso contrario, añadimos al orden la fuente y eliminamos este vértice y sus aristas del grafo.

9. OTROS PROBLEMAS INTERESANTES.

Aquí simplemente vamos a hacer una lista de problemas interesantes y algunos teoremas o algoritmos que pueden solucionarlo.

- Caminos entre dos vértices: se explicó con el BFT.
- Camino más corto: también se explicó con el BFT.
- Detección de ciclos: una posible solución con el orden topológico.
- Ciclos eulerianos: se resuelve con la caracterización de ciclos eulerianos. Un grafo tiene ciclos eulerianos si y solo si es conexo y todos sus vértices tienen grado par.
- Ciclos hamiltonianos: no existe de momento ninguna caracterización que lo resuelva fácilmente
- 2-coloreable: es dos colooreable si es bipartito.
- Árbol Mínimo de expansión: con el Algoritmo de Prim o Jarnick.
- Biconexo: si al eliminar alguno de los vértices un grafo conexo queda dividido en dos componentes conexas.
- Planar: con el teorema o caracterización de Kuratowski.
- Grafos isomorfos: hay teoremas que nos dicen si no son isomorfos, si si lo son solamente podemos comprobar la definición.