Tipos de Datos Abstractos (TADs)

Una estructura de datos está definida por un conjunto de valores que puede asumir, un conjunto de operaciones que pueden realizarse con ellas y una serie de reglas que relaciona a ambas definiciones.

Tipos de Datos Concretos

Son estructuras de datos destinadas a mantener y recuperar conjuntos de tipos elementales. Habitualmente se les denomina tipos estructurados.

Listas

Primero recordemos una estructura de datos que ya fue definida, las *Lists.*
La Lista es una estructura de datos lineal de acceso secuencial, de longitud variable donde todos los elementos deben ser del mismo tipo.

```
i1 =  [1,2,3,4,5]
i1:: [Int]

b1 = [True,False,False,True]
b1:: [Bool]

fl1 = [[1.25, 3.01, 0.78], [2.41], [8.66, 47.3] ]
fl1:: [ [Float] ]

t1 = [( 1,"pepe", [2, 45]), (2, "Laura", [5,19]), ( 3, "Ramon", []) ]
t1:: [ ( Int, [Char], [Int] ) ]

e1 = []
e1:: [a]
```

Las listas pueden ser construidas por el agregado de un elemento al principio de una lista existente. Esto se lleva a cabo con el operador (**:**) , llamado '*cons*' por constructor, el cual toma dos argumentos. El primero de ellos denominado el *head* de la lista y el segundo denominado *tail*. Utilizado como operador infijo separa el *head* del *tail* de la lista.
La lista [e1, e2, e3,… ,en] resulta así ser una abreviación de la expresión e1 : (e2 : (e3 : (…(en : [])))) , y como el operador es asociativo por derecha pueden omitirse los paréntesis.

```
3 : [4,5]  =  [3,4,5]
True : []  =  [True]
```

La operación de concatenación de listas (++) puede ser escrita en función del operador cons.

```
(++):: [a]->[a]->[a]
(++) x [] = x
(++) [] y = y
(++) (x:t) y = x: ( t ++ y)
```

Tipos definidos por el usuario

1. Sinónimos de Tipos

Los sinónimos de tipo se utilizan para proporcionar abreviaciones para expresiones de tipo aumentando la legibilidad de los programas. Un sinónimo de tipo es introducido con una declaración de la forma:

type *Nombre a1 ... an = expresion_Tipo*

Donde

- *Nombre* es el nombre de un nuevo constructor de tipo de aridad `n>=0`
- *a1,..., an* son variables de tipo diferentes que representan los argumentos de *Nombre*
- *expresion_Tipo* es una expresión de tipo que sólo utiliza como variables de tipo las variables *a1,..., an* **.**

Ejemplo:

```
type Nombre = String
type Edad = Integer
type String = [Char]
type Dni  = Integer
type Persona = (Nombre, Edad)

tocayos::Persona -> Persona -> Bool
tocayos (nombre,_) (nombre',_) = (nombre == nombre')
```

Observaciones:

Ver que los nombres de los tipos de datos **van en Mayúsculas**.
En este caso el compilador no distingue entre el tipo original y su sinónimo, solo se utilizan para dar legibilidad.

2. Definiendo tipos de Datos

Además de los tipos provistos por Haskell, podemos crear nuestros propios tipos de datos mediante la sentencia **data**.
La definición de nuevos tipos de datos aumenta la seguridad de los programas ya que el sistema de inferencia de tipos distingue entre los tipos definidos por el usuario y los tipos predefinidos.
Estos tipos de datos pueden ser tipo Producto (entendemos el producto como un AND) o tipo Suma (entendiendo una suma como OR).
Así diremos que un tipo de dato producto está dado por *Expre1* **Y** *Expre2* **Y** … *Expre n.* O bien diremos que un tipo suma esta definido por *Expre 1* **O** *Expre 2* **O** … *Expre n*

Tipos Producto

Se utilizan para construir un nuevo tipo de datos formado a partir de otros.
Ejemplo:

(Una persona es un **Nombre, Y** un **Dni, Y** una **Edad**)

```
data Persona = Pers Nombre Dni Edad
juan::Persona
juan = Pers "Juan Lopez" 39235478 23
```

Se pueden definir funciones que manejen dichos tipos de datos:

```
esJoven:: Persona -> Bool
esJoven (Pers _ _ edad) = edad < 25
verPersona::Persona -> String
verPersona (Pers nombre docu edad) = "Persona, nombre: " ++ nombre ++ ", edad: " ++ show edad
++ ", DNI: " ++ show docu
```

También se pueden dar nombres a los campos de un tipo de datos producto:

```
data Persona = Pers {nombre::String, dni::Integer, edad::Integer} deriving Show
ej.: p1 = Pers {nombre="Pedro", dni=36458793, edad= 28}
     nombre p1 --→ "Pedro", edad p1 <30 :-→ True
```

Los nombres de dichos campos sirven como funciones selectoras del valor correspondiente.

Y esto nos soluciona el problema de tener que escribir funciones dedicadas a la
Por ejemplo:

```
tocayos:: Persona -> Persona -> Bool
tocayos p p' = nombre p == nombre p'
```

Obsérvese la diferencia de las tres definiciones de Persona

Como sinónimo de tipos:

```
type Persona = (Nombre, Edad)
```

No es un nuevo tipo de datos.

En realidad, si se definiera otro sinónimo tal como

```
type Numero    = Integer
type Direccion = (Nombre, Numero)
```

El sistema no daría error al aplicar una función que requiera un valor de tipo `Persona` con un valor de tipo `Direccion` .

La única ventaja (discutible) de la utilización de sinónimos de tipos de datos podría ser una mayor eficiencia (la definición de un nuevo tipo de datos puede requerir un mayor consumo de recursos).

Como Tipo de Datos:

```
data Persona = Pers Nombre Edad
```

El valor de tipo `Persona` es un nuevo tipo de datos

Y, si se definiera otro tipo de la forma

```
data Direccion = Dir Nombre Numero
```

El sistema daría error al utilizar una dirección en lugar de una persona.

Sin embargo, en la definición de una función por pattern matching, es necesario conocer el número de campos que definen una persona, por ejemplo:

```
esJoven (Pers _ edad) = edad < 25
```

Si se desea ampliar el valor persona añadiendo, por ejemplo, el "Dni", todas las definiciones de funciones que trabajen con datos de tipo Persona deberían modificarse.

Mediante campos con nombre:

```
data Persona = Pers { nombre::Nombre, edad::Edad }
ej.: p1 = Pers {nombre= "Fernando", edad =22}
```

El campo es un nuevo tipo de datos y ahora no es necesario modificar las funciones que trabajen con personas si se amplían los campos.

Tipos Enumerados

Se puede introducir un nuevo tipo de datos enumerando los posibles valores.

Ejemplos:

```
data Color = Rojo | Verde | Azul
data Temperatura = Frio | Caliente
data Estacion = Primavera | Verano | Otonio | Invierno
```

Se pueden definir funciones simples mediante pattern matching:

```
tiempo :: Estacion -> Temperatura
tiempo Primavera = Caliente
tiempo Verano = Caliente
tiempo _ = Frio
```

Las alternativas pueden contener a su vez productos:

```
data Forma = Circulo Float | Rectangulo Float Float
area :: Forma -> Float
area (Circulo radio) = pi * radio * radio
area (Rectangulo base altura) = base * altura
```

Tipos Recursivos

Los tipos de datos pueden autoreferenciarse consiguiendo valores recursivos, por ejemplo:

```
data Expr = Lit Integer | Suma Expr Expr | Multi Expr Expr
eval (Lit n) = n
eval (Suma e1 e2) = eval e1 + eval e2
eval (Multi e1 e2) = eval e1 * eval e2
```

Nuevos Tipos de Datos a partir de tipos existentes

En ciertas ocasiones puede desearse utilizar un tipo de datos ya existente (por motivos de eficiencia) pero etiquetarlo para que distinguirlo del tipo de datos original (Recordar que un sinónimo no se distingue del original).
Para ello, se puede emplear la declaración **`newtype`**.

```
newtype Persona = Per (Nombre, Edad)
esJoven (Per (_, edad)) = edad < 25
```

Internamente, el tipo persona se representa como una tupla con dos valores, pero un valor de tipo `Persona` se distingue mediante la etiqueta `Per` .

3. Introducción al Sistema de Clases en Haskell y la Sobrecarga

Cuando una función puede utilizarse con diferentes tipos de argumentos se dice que está sobrecargada. La función **`(+)`**, por ejemplo, puede utilizarse para sumar enteros o para sumar flotantes. La resolución de la sobrecarga por parte del sistema Haskell se basa en organizar los diferentes tipos en lo que se denominan ***clases de tipos.***

Considérese el operador de comparación `==`. Existen muchos tipos cuyos elementos pueden ser comparables, sin embargo, los elementos de otros tipos podrían no ser comparables.
Por ejemplo, comparar la igualdad de dos funciones es una tarea computacionalmente intratable, mientras que a menudo se desea comparar si dos listas son iguales (Ejercicio para el lector).
De esa forma, si se toma la definición de la función `elem` que chequea si un elemento pertenece a una lista:

```
x `elem` [] = False
x `elem` (y:ys) = x == y || (x `elem` ys)
```

Intuitivamente el tipo de la función `elem` debería ser `a->[a]->Bool`. Pero esto implicaría que la función `(==)` tuviese tipo `a->a->Bool`. Sin embargo, como ya se ha indicado, interesaría restringir la aplicación de `(==)` a los tipos cuyos elementos son ***comparables***.

Además, aunque `==` estuviese definida sobre todos los tipos, no sería lo mismo comparar la igualdad de dos listas que la de dos enteros.

Las ***clases de tipos*** solucionan ese problema permitiendo declarar qué tipos son instancias de unas clases determinadas y proporcionando definiciones de ciertas operaciones asociadas con cada clase de tipos. Por ejemplo, la clase de tipo que contiene el operador de igualdad se define en el *standar prelude* como:

```
class Eq a where
(==) :: a->a->Bool
```

`Eq` es el nombre de la clase que se está definiendo, **`(==)`** y **`(/=)`** son dos operaciones simples sobre esa clase. La declaración anterior podría leerse como: *"Un tipo* **`a`** *es una instancia de una clase* ***`Eq`*** *si hay una operación* ***`(==)`*** *definida sobre él"*.
La restricción de que un tipo **`a`** debe ser una instancia de una clase **`Eq`** se escribe **`Eq a`**.

Obsérvese que **`Eq a`** no es una expresión de tipo sino una restricción sobre el tipo de un objeto **`a`** (se denomina un *contexto*). Los contextos son insertados al principio de las expresiones de tipo. Por ejemplo, la operación == sería del tipo:

```
(==):: (Eq a) => a -> a -> Bool
```

Esa expresión podría leerse como:
"Para cualquier tipo `a` *que sea una instancia de la clase* `Eq`, == *tiene el tipo* `a->a->Bool`*"*.
La restricción se propagaría a la definición de `elem` que tendría el tipo:

```
elem:: (Eq a) => a -> [a] -> Bool
```

Las declaraciones de instancias permitirán declarar qué tipos son instancias de una determinada clase.
Por ejemplo:

```
instance Eq Integer where
        (==) x y = intEq x y
```

(Recordar que el tipo "`Integer`", es un TDA, no es el típico entero de tamaño registro de CPU, el cual se puede operar en ASM, por tanto las operaciones de un `Integer` deben implementarse como funciones, en este caso `intEq`). La definición de == se denomina **método.** `IntEq` es una función primitiva que compara si dos enteros son iguales, aunque podría haberse incluido cualquier otra expresión que definiese la igualdad entre enteros. La declaración se leería como: *"El tipo* `Integer` *es una instancia de la clase* `Eq` *y el método correspondiente a la operación == se define como..."*.
De la misma forma se podrían crear otras instancias:

```
instance Eq Float where
         x == y = FloatEq x y
```

La declaración anterior utiliza otra función primitiva que compara flotantes para indicar cómo comparar elementos de tipo `Float`.
Además, se podrían declarar instancias de la clase `Eq` tipos definidos por el usuario.
Por ejemplo para nuestro tipo de dato Persona, definido con anterioridad, podríamos definir cuál será nuestro criterio para indicar la igualdad de dos personas. Vamos a utilizar la comparación por número de DNI como criterio de igualdad.

```
data Persona = Pers {nombre::String, dni::Integer, edad::Integer} deriving Show

instance Eq Persona where
         (==) p1 p2 = dni p1 == dni p2
```

El *standar prelude* incluye un amplio conjunto de clases de tipos. De hecho, la clase `Eq` está definida con una definición ligeramente más larga que la anterior:

```
class Eq a where
       (==), (/=) :: a->a->Bool
       x /= y = not (x == y)
```

Se incluyen dos operaciones, una para igualdad `(==)` y otra para no igualdad `(/=)`. Se puede observar la utilización de un ***método por defecto*** para la operación (/=), el método por defecto es una definición de la clase. Si se omite la declaración de un método en una instancia entonces se utiliza la declaración del método por defecto de su clase. Por ejemplo, las tres instancias anteriores podrían utilizar la operación (/=) sin problemas utilizando el método por defecto (la negación de la igualdad).

Haskell también permite la *inclusión* de clases. Por ejemplo, podría ser interesante definir una clase `Ord` que *hereda* todas las operaciones de `Eq` pero que, además tuviese un conjunto nuevo de operaciones:

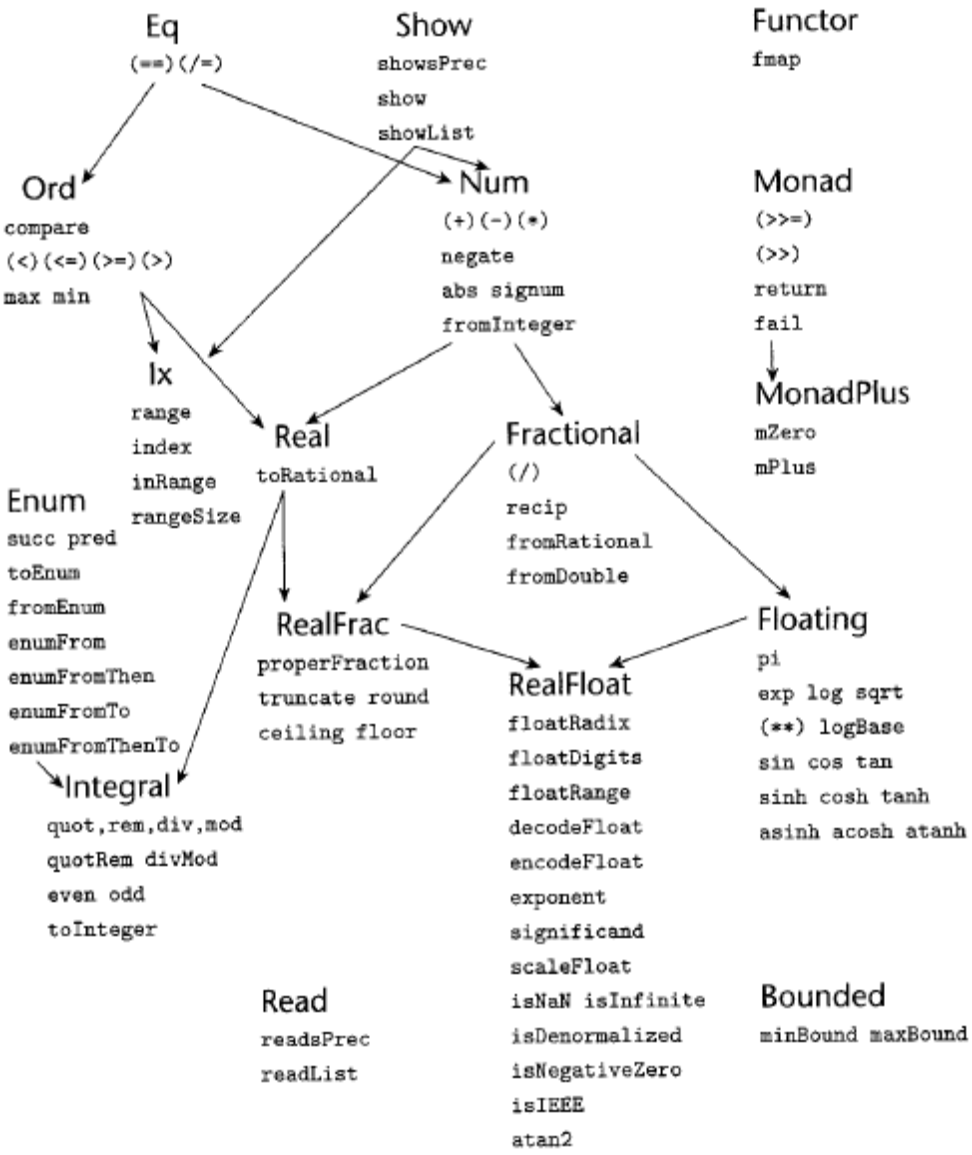
```
class (Eq a) => Ord a where
       (<), (<=), (>=), (>) :: a->a->Bool
       max, min :: a->a->a
```

El contexto en la declaración indica que `Eq` es una **superclase** de `Ord` (o que `Ord` es una **subclase** de `Eq`), y que cualquier instancia de `Ord` debe ser también una instancia de `Eq`.

Las inclusiones de clase permiten reducir el tamaño de los contextos: Una expresión de tipo para una función que utiliza operaciones tanto de las clases `Eq` como `Ord` podría utilizar el contexto `(Ord a)` en lugar de `(Eq a,Ord a)`, puesto que `Ord` implica `Eq`. Además, los métodos de las subclases pueden asumir la existencia de los métodos de la superclase. Por ejemplo, la declaración de `Ord` en el *standar prelude* incluye el siguiente método por defecto:

```
x < y = x <=y && x/=y
```

Jerarquia de Clases (*Standard Prelude*)



Tipos Algebraicos y Polimorfismo

Haskell nos permite definir nuestros propios tipos de datos, utilizando constructores. Un constructor es justamente como una función que espera unos argumentos y devuelve un valor del tipo que esta define.

Por ejemplo, supongamos que queremos un tipo de dato que nos proporcione la representación de un par de coordenadas en el plano cartesiano.
A este nuevo tipo de dato podemos darle el nombre de ***CoordType***, luego definimos una función constructor que llamamos ***Coord***, con dos argumentos, las componentes ***x*** e ***y*** , que devuelve un valor del tipo ***CoordType***.

La declaración de tal tipo se realiza con la expresión:

```
data CoordType = Coord Float Float  deriving Show
```

deriving Show al final de la declaración es necesario para que los tipos definidos por el usuario puedan ser mostrados de manera estándar.

Si en un programa nosotros queremos referirnos a una coordenada por ejemplo con componente x = 14.0 e y = 2, nos referiremos a un dato tipo **CoordType** de la siguiente manera.

```
c1 = Coord 14.0 2.0
c1:: CoordType
```

Una vez que se definió el nuevo tipo de dato, podemos escribir funciones de propósito general que tomen argumentos de este tipo.

```
getX  (Coord x _) = x
getY  (Coord _ y) = y
```

Observar que el *pattern matching* utilizado en las funciones es similar al que usamos con las listas, excepto que en aquel caso se usaba como operador infijo (**x:xs**), por ese motivo aparecía entre los identificadores de los argumentos formales.

La siguiente función toma una lista de coordenadas y determina si todos los puntos se encuentran en el semiplano del primer cuadrante (donde las coordenadas x e y son positivas o cero).

```
firstQuad [] = True
firstQuad ( (Coord x y) : cs) = ( x >=0) && (y >= 0) && (firstQuad cs)
```

Constructores Alternativos

Valores de un determinado tipo de datos no tienen por qué ser creados por un único constructor.
Consideremos un tipo de dato que represente a figuras geométricas las cuales pueden ser rectángulos, triángulos o círculos con la información que corresponda a cada figura. Por ejemplo el circulo necesita para quedar definido las coordenadas de su centro y el radio del mismo, el rectángulo las coordenadas de dos vértices no adyacentes y el triángulo las coordenadas de sus tres vértices.
Asi,

```
data Shape =   Rectangle CoordType CoordType
             |Circle CoordType Float
             |Triangle CoordType CoordType CoordType
             deriving Show
```

Definimos los constructores alternativos separados por la barra vertical *'pipe'* .

Una figura es así tanto un rectángulo, como un círculo o un triángulo. Algunos ejemplos:

```
f1 = Rectangle (Coord 4.9  2.1)  (Coord 7.25  6.8)
f2 = Circle (Coord 2.56 3.8)  4.5
f3 = Triangle (Coord 0.0  0.0) (Coord 44.88 0.0) (Coord 22.44 5.42)
```

Una función que acepta como argumento un tipo de dato que consta de constructores múltiples, debería tener al menos un pattern que se corresponda con cada uno de los constructores.

Por ejemplo, una función que calcula el área de una figura.

```
area (Rectangle corner1 corner2) = abs(getX corner1 - getX corner2) * abs(getY corner1 - getY corner2)

area (Circle center radius) = pi * radius * radius

area (Triangle vert1 vert2 vert3)= sqrt(h * (h-a) * (h-b) * (h-c))
                                   where
                                           h = (a+b+c)/2.0
                                           dist (Coord x1 y1) (Coord x2 y2)= sqrt((x1-x2)^2 +(y1-y2)^2)
                                           a = dist vert1 vert2
                                           b = dist vert1 vert3
                                           c = dist vert2 vert3
```

Polimorfismo Simple

Supongamos que deseamos conocer el tipo de la función `fst`, definida de la siguiente manera,

```
fst (x,_) = x
```

En Haskell el tipo de `fst` está dado por

```
fst :: (a,b) -> a
```

Esta definición de tipo nos dice que `fst` toma un argumento del tipo tupla esto lo indicamos con dos variables de tipo, y que el resultado es del mismo tipo que el primer elemento de la tupla (la misma variable a).
Cuando una función en su definición de tipo contiene variables, decimos que es una ***función polimórfica***.

Polimorfismo de Listas

Consideremos el tipo de dato List, si este no estuviera ya definido en Haskell, podríamos definirlo como un tipo de dato con las propiedades de la lista.

```
data List a = Cons a (List a) | Nil  deriving Show
```

Hemos parametrizado la definición del tipo `List`, mediante el uso de una variable de tipo `a`. Al menos un constructor para este tipo debería recibir como argumento a esta variable de tipo. En nuestro caso el constructor `Cons` requiere como primer argumento una variable del mismo tipo que `a`, y el segundo argumento será una Lista de `a`, indicando de este modo que todos los elementos deberán ser del mismo tipo.
A estas definiciones de tipo con variables de tipo, se las denominan ***definiciones de datos polimórficos.***

```
Cons 3 (Cons 12 (Cons 7 Nil))    => [3,12,7]
Cons False Nil                   => [False]
```

La mayoría de las funciones sobre listas son polimórficas, independientes del tipo de datos que componen la lista, ejemplo:

```
length::[a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

No todas las funciones sobre listas son polimórficas,

```
sum :: (Num p)=>[p] -> p
sum [] = 0
sum (x:xs) = x + sum xs
```

Tipos de Datos Abstractos (TDA)

Los TDA son independientes de la implementación y están implícitamente definidos a través de un conjunto de operaciones utilizadas para manipularlos.

Las definiciones de tipos de las operaciones y sus especificaciones conforman la interface del TDA, pero ni la estructura interna del TDA ni la implementación de sus operaciones son visibles al usuario.

Dada la especificación de un TDA, esas operaciones pueden tener una o varias implementaciones, dependiendo de la estructura de datos, o de los algoritmos, elegidos para la representación y operación del TDA.

El gran beneficio de la utilización de TDAs, es que refuerzan el concepto de modularidad en los programas. Así un cambio en la implementación de un conjunto de operaciones no afecta otras partes del programa mientras se conserve la interface.

Aunque no existe una construcción particular para soportar TDAs en Haskell, un TDA puede ser fácilmente implementable mediante un módulo Haskell. Un módulo en Haskell define una colección de funciones y definiciones de tipos de datos en un ambiente cerrado, y sólo exportar todas o sólo parte de sus definiciones. Un módulo se define utilizando la siguiente instrucción:

module *name (export list)* **where**

En nuestros ejemplos nombraremos al módulo con el nombre del TDA y la *export list* primero exportará el nombre del TDA seguido por las operaciones para usar el TDA.

Pila o Stack

Una pila o stack es una colección homogénea de elementos sobre la cual se definen dos operaciones.

`push` ***x s***, la cual coloca el elemento ***x*** en el tope de la pila ***s***, devolviendo una nueva pila con ***x*** en su tope.

`pop` ***s*** , la cual elimina el elemento tope de la pila, devolviendo una nueva pila que ya no lo contiene. El elemento tope de la pila es accesible mediante la operación `top` ***s***.

Los ítems son removidos de la pila utilizando la estrategia **LIFO** Last In First Out (último en entrar primero en salir).

Como la representación subyacente del TDA esta oculta al usuario, es necesario proveer una función `emptyStack`, la cual crea una pila vacía. La función `stackIsEmpty` sirve para verificar si la pila está vacía o no.

De este modo en Haskell la definición del Stack como un módulo TDA.

```
module Pila (Stack, pop, push, top, emptyStk, stackIsEmpty) where

push          :: a -> Stack a -> Stack a
pop           :: Stack a -> Stack a
top           :: Stack a -> a
emptyStk      :: Stack a
stackIsEmpty :: Stack a -> Bool
```

Las definiciones de tipos de las funciones no son necesarias, ya que Haskell las deduce de la implementación, pero resultan útiles como documentación para el usuario del TDA.

Una primera implementación puede hacerse mediante un tipo de dato definido por el usuario.

```
data Stack a = EmptyStk | Stk a (Stack a)  deriving Show

emptyStk = EmptyStk

push x s = Stk x s

pop EmptyStk = error "Stack Vacio"
pop (Stk _ s)  = s

top EmptyStk = error "Stack Vacio"
top (Stk x _) = x

stackIsEmpty EmptyStk = True
```

```
stackIsEmpty _ = False
```

Otra posible implementación podría ser utilizando la estructura de datos predefinida **`List`**, ya que `push` y `pop` se parecen a las operaciones `(:)` y **`tail`** .

Pero para asegurarse de que las únicas operaciones permitidas sobre la pila son las definidas en el TDA, creamos un nuevo tipo de dato utilizando **`newtype`** con el constructor **`Stk`** para diferenciar entre las listas predefinidas y las listas utilizadas en el TDA para la implementación de la pila.

La presencia de la palabra reservada **`newtype`** puede hacer que Haskell tenga en cuenta que el constructor solo se utiliza para la comprobación de tipos. Así este constructor no incurre en sobrecarga en tiempo de ejecución.

```
newtype Stack a       = Stk [a] deriving Show

emptyStack            = Stk []

push x (Stk xs)       = Stk (x:xs)

pop (Stk [])          = error "Pila Vacia"
pop (Stk (_:xs))      = Stk xs

top (Stk [])          = error "Pila Vacia"
top (Stk (x:_))       = x

stackIsEmpty (Stk []) = True
stackIsEmpty (Stk _ ) = False
```

Todas estas operaciones sobre Stacks resultan eficientes ya que operan en $O(1)$.

Cola o Queue

Una cola es una estructura de datos similar a la pila pero para remover elementos utiliza la estrategia **FIFO** (*First In First Out*), el primero en entrar es el primero en salir. El tipo de cada operación es similar a su contrapartida en el TDA Stack.

```
module Cola (Queue, emptyQueue, enQueue, deQueue, queueIsEmpty, front) where

emptyQueue   :: Queue a
enQueue      :: a -> Queue a -> Queue a
deQueue      :: Queue a -> Queue a
front        :: Queue a -> a
queueIsEmpty :: Queue a -> Bool
```

Como en el caso de la Pila, la implementación más sencilla es mediante listas

```
newtype Queue a      = Q [a] deriving Show

emptyQueue           = Q[]

enQueue x (Q s)      = Q (s ++ [x])

deQueue (Q [])       = error "ColaVacia"
dequeue (Q(x:t))     = Q t

front (Q [])         = error "ColaVacia"
front (Q(x:t))       = x

queueIsEmpty (Q [])  = True
queueIsEmpty (Q _ )  = False
```

El TDA Diccionario

Una estructura de diccionario es un tipo de dato abstracto (TDA) cuyo propósito es mantener un conjunto de datos, los cuales son identificables por un valor en particular que denominamos *"Clave"*. Esta clave es la que sirve para localizar e identificar al elemento dentro del conjunto. (Así como en un diccionario utilizamos el término como clave de búsqueda y este tiene asociado otro tipo de información, su significado).
Las operaciones de mantenimiento de este conjunto comprenden tres operaciones básicas:

- Insertar
- Eliminar
- Buscar

Con respecto a la implementación de este TDA, nos resultan de interés abordar tres posibles soluciones, mediante listas ordenadas, Arboles Binarios de Búsqueda y mediante Tablas de dispersión (Hashing).

Vamos a analizar brevemente las ventajas y desventajas de cada estrategia.

Listas enlazadas: Ya hemos visto que las listas son muy fáciles de implementar y manejar, sólo nos hace falta considerar algunos aspectos de eficiencia (tiempo de procesamiento) en las operaciones de inserción y búsqueda.
El tiempo optimista de búsqueda de un elemento dentro de una lista, se da cuando este se encuentra en la entrada de la misma, en este caso se llega a él en un tiempo igual a 1 operación.
$T_{min} = 1$
El tiempo máximo de búsqueda, o tiempo pesimista, se da en el caso de tener que recorrer la totalidad de la lista "secuencialmente", lo cual para un conjunto de N miembros nos dará N operaciones.
$T_{max} = N$
Por tanto podríamos decir que el tiempo promedio de búsqueda, para una estructura secuencial como las listas, está dado por:
$T = (T_{min} + T_{max}) / 2 = (1 + N)/2$
Es decir el tiempo está dado por un polinomio de grado1 en N (N: cantidad de datos), en este caso diremos que el tiempo promedio de búsqueda (o inserción si utilizo listas ordenadas) es ***"del orden de N" u O(n).***

Arboles binarios de búsqueda: Sabemos que un árbol binario, es una forma de organizar datos. Esta organización está dada por una regla simple y recursiva que expresa: *"Dado un nodo con un valor determinado x, el subárbol izquierdo contiene nodos con valores inferiores a x y el subárbol derecho contiene valores mayores a x"*. (Propiedad BST)
Si los valores que se insertaron en el árbol se proporcionaron con la suficiente aleatoriedad podemos pensar que el árbol se encuentra balanceado, esto significa que cuando nos paramos sobre la raíz del mismo, la profundidad del subárbol derecho es igual a la profundidad del subárbol izquierdo(o al menos se diferencian en 1 nivel).
Veamos si podemos deducir los tiempos o cantidad de operaciones en la inserción/búsqueda de un elemento en el árbol.
Observar que la profundidad del árbol es directamente proporcional a la cantidad de operaciones (consulta de etiqueta de nodo y bifurcación de camino).
Sea K, la cantidad de operaciones necesarias y n la cantidad de datos (nodos) acomodados en estas K operaciones.
$K = 0 , n = 1$
$K = 1 , n = 2$
$K = 2 , n = 4$
.
.
.
$K = i , n = 2^i$***(*)***

Si el conjunto posee N datos, ¿Cuántas operaciones serían necesarias para acomodar estos datos en el árbol?
De (*) tendríamos

$$N = 2^T$$

Aplicando Log_2 a ambos miembros obtenemos:

$$Log_2 (N) = T * Log_2(2)$$

$$\boldsymbol{Log_2(N) = T}$$

Es decir el tiempo máximo que lleva acomodar N elementos en un árbol binario de búsqueda es aproximadamente proporcional al Log_2 de N (diremos que es ***O(lg n)***).

Para tener una idea aproximada de lo que esto significa, consideremos lo siguiente. Comparemos cuanto es el tiempo que lleva acomodar en una lista ordenada y en un árbol binario de búsqueda un lote de 1024 (2^{10}) datos.

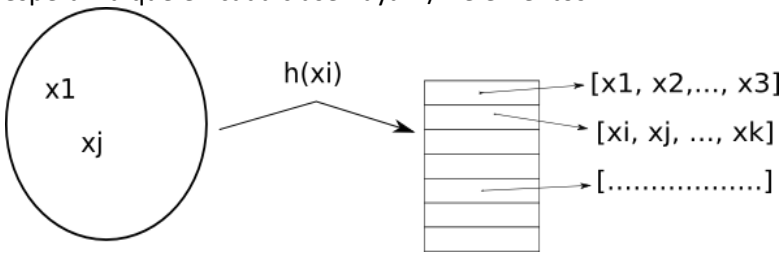
Lista ordenada: (1 + 1024)/2 ≈ 512 operaciones
Árbol Binario de búsqueda: Log_2 (1024) ≈Log_2 (2^{10}) ≈ 10 operaciones

Tabla de Dispersión (Hash Table)*:* Las tablas de dispersión requieren un tiempo promedio constante para acceder a un elemento del conjunto. Con un diseño cuidadoso podemos conseguir un tiempo de acceso lo suficientemente pequeño. En el peor caso tendremos un tiempo proporcional al tamaño del conjunto, como las listas.
Debemos considerar dos tipos de dispersión: Abierta y Cerrada.

La dispersión abierta o externa permite que el conjunto sea guardado en un espacio virtualmente ilimitado.
La idea es particionar el potencialmente infinito conjunto de miembros en un número finito de clases. Si tenemos **B** clases numeradas de **0** a **B-1**, luego tenemos una ***función de dispersión h()*** , tal que para cada **x** del tipo de dato del conjunto que se va a representar, ***h(x)*** es un entero de **0** a **B-1**. El valor de *h(x)* es naturalmente la clase a la x pertenece. Generalmente a x lo denominamos *clave* y a *h(x) valor de dispersión de x*.

Consideremos que esas clases la representamos con un arreglo de B componentes, numeradas de 0 a B-1. En este caso el valor *h(x),* número de clase corresponde a un número de componente.
En cada componente almacenaremos una lista con todos los elementos del conjunto que pertenecen a la misma clase es decir la clase *Bi = {x/ h(x) = i}*

Si el conjunto a clasificar tiene N elementos, es nuestra esperanza que en cada clase haya N/B elementos.
Este deseo sólo se hará realidad si la función de dispersión tiene una distribución uniforme en el intervalo 0, B-1.
El secreto del buen funcionamiento de la dispersión está en la elección de la función de dispersión h(x).
Para conjuntos particulares pueden elegirse determinado tipo de funciones que tienen en cuenta características particulares del dominio.



La dispersión cerrada, coloca todos los elementos del conjunto en el arreglo mismo, no crea listas, esto significa que sólo podemos acomodar conjuntos de N = B.
Cuando a dos elementos les corresponde la misma clase *h(x1) = h(xj) = m*, se aplica lo que denominamos "*estrategia de redispersión*".
Si tratamos de colocar el elemento x en la posición h(x) y encontramos que ya está ocupada, ocurre lo que se denomina una colisión, y la estrategia de redispersión elige una secuencia de lugares alternativos h1(x), h2(x), h3(x)... dentro de la tabla donde debemos colocar a x. Se intenta cada una de las ubicaciones en secuencia hasta encontrar una "Vacia". Sino, x NO PUEDE SER INSERTADO.

Inicialmente asumimos que la tabla está vacía, esto significa que en el arreglo tenemos guardado un valor especial "vacío", que debe ser distinto a cualquiera de los datos que se quieren insertar.

0	b
1	
2	
3	a
4	c
5	d
6	
7	

Supongamos que queremos insertar los valores a, b, c y d cuyos valores de dispersión son h(a) = 3, h(b) = 0, h(c) = 4 y h(d) =3.
Como estrategia de redispersión utilizaremos una forma sencilla llamada *"dispersión lineal"* , cuya ley sería $\boldsymbol{h_i(x) = (h(x) + i) \bmod B}$. Si nuestra tabla en cuestión tiene 8 componentes, B = 8.
Vemos que al insertar "a", se coloca en la posición 3, "b" en la cero y "c" en la 4.

Cuando tratamos de insertar “d” vemos que h(d)=3 está ocupado, entonces luego probamos h1(d)=4 y también está ocupado, h2(d) = 5 lo encontramos vacío entonces lo insertamos en esa posición.
El test de pertenencia de un elemento x (es lo que denominamos buscar(x)) requiere que se examinen h(x),h1(x),h2(x), …, hi(x), hasta que encontremos x o una posición “vacio”.
Primero supongamos que no se permiten las eliminaciones.
Digamos que buscando x, se da que h3(x) es la primera posición que encontramos vacía, entonces sería imposible que x estuviera en h4(x), h5(x) o cualquier posición subsiguiente en la secuencia, ya que esto sólo sería posible, si h3(x) estuvo lleno en el momento de insertar x.
Notar que si permitimos eliminaciones, nunca podremos estar seguros que x no está en el conjunto, al encontrar una cubeta vacía. Ya que ésta podría haber estado ocupada al momento de insertar x, y luego se eliminó el elemento que la ocupaba.
Para subsanar esta situación, cuando eliminamos un elemento del conjunto, la posición se marca con otro valor especial, “borrado”. Este valor tiene distinto significado según se esté insertando o buscando.
En el caso de una inserción la posición “vacio” y “borrado” tienen la misma significación, ya que significa que la posición dentro del array está disponible.
En el caso de la búsqueda, una posición marcada como “vacio” significa que nunca hubo nada en ese lugar, por lo tanto debe parar el proceso de búsqueda. Por otro lado una posición marcada con “borrado” significa que hubo elemento en esa posición, por lo tanto a los efectos de la búsqueda es como si estuviera ocupado, en consecuencia el proceso continua.

La dispersión abierta es utilizada, por ejemplo, como una de las etapas en la construcción de índices multiniveles en la tecnología de Bases de Datos.
La dispersión cerrada, como pone un límite a la cantidad de claves, es utilizada habitualmente como herramienta interna en aplicaciones que requieren mantener tablas de símbolos, tablas de variables, o en el almacenamiento de matrices ralas.

Definición del TDA Diccionario

El tipo Diccionario será de la forma **(*Dict a*)**.
Donde ***a*** es un tipo que deberá ser instancia de la clase *Ord* a los efectos de soportar operaciones de comparación e igualdad. El argumento ***a*** corresponde al tipo del elemento a almacenar en el diccionario.

Las operaciones necesarias para manipular el TDA diccionario serán:

`mkNewDict :: Dict a`

Retorna un diccionario vacío, listo para insertar elementos. Observar que aún no se sabe cuál será el tipo ***a***. (Lo mismo sucede con la lista vacía `[]`, la cual tiene tipo `[]::[a]`)

`insertDict :: (Ord a)=> a -> Dict a -> Dict a`

Dado un elemento de tipo ***a***, y un diccionario de elementos del tipo ***a***, inserta este elemento en el diccionario y retorna un nuevo diccionario que contiene al elemento.

`inDict :: (Ord a) => a -> Dict a -> Bool`

Verifica si el elemento proporcionado como argumento se encuentra en el diccionario.

`delDict :: (Ord a) => a -> Dict a -> Dict a`

Se proporciona un elemento a eliminar del diccionario. Retorna un diccionario donde se ha eliminado el elemento ***a*** del conjunto. Si la clave del elemento proporcionado no existe, se devuelve el mismo diccionario.

*Ejercicio para el lector: Escriba el módulo diccionario con una implementación de los mismos utilizando **List**.*

Árbol Binario de Búsqueda

Primero debemos definir el tipo de dato necesario para construir el árbol utilizando **`data`**, y construir las funciones que conforman su interface. Para ello, nos basaremos en la definición recursiva del árbol.

Regla base: *Un árbol vacío es un árbol binario.*

Ley inductiva: *Si T_1 y T_2 son árboles binarios, y **x** es un valor (tal que $\forall t_1 \in T_1 \wedge \forall t_2 \in T_2$ se verifica que : $t_1 < x < t_2$). Entonces podemos conformar otro árbol binario llamado **T** creando un nuevo nodo que contenga al valor **x**, el subárbol izquierdo T_1 y el subárbol derecho T_2. El nuevo nodo creado es la raíz de **T**.*

```
data  Bintree a = EmptyBT | NodoBT a (Bintree a) (Bintree a) deriving Show

mkNewTree   ::(Ord a)=> Bintree a
inTree      ::(Ord a)=> a -> Bintree a -> Bool
addTree     ::(Ord a)=> a -> Bintree a -> Bintree a
delTree     ::(Ord a)=> a -> Bintree a -> Bintree a
```

Creacion de una instancia de Bintree vacía

```
mkNewTree::(Ord a)=> Bintree a
mkNewTree = EmptyBT
```

Búsqueda:

En este caso construiremos una función que retornara verdadero o falso según halle o no el valor indicado dentro del árbol.

*La búsqueda de un elemento **x** dentro del árbol comenzará siempre en la raíz.*

*Caso Base: buscar **x** en un árbol **Vacio**, la búsqueda ha fallado, no es posible encontrar a **x** en el árbol.*

*Inducción: Si el dato a buscar **x** es menor que el valor del nodo se continuará buscando en el subárbol izquierdo, si es mayor en el subárbol derecho.*

```
inTree::(Ord a)=> a -> Bintree a -> Bool
inTree x EmptyBT = False
inTree x (NodoBT y lf rt) | x == y  = True
                          | x < y   = inTree x lf
                          | x > y   = inTree x rt
```

Inserción:

La función insertar, retornará un árbol binario en el cual se ha agregado un nodo con el valor indicado como parámetro (el nodo agregado siempre estará en una hoja o nodo externo del árbol).

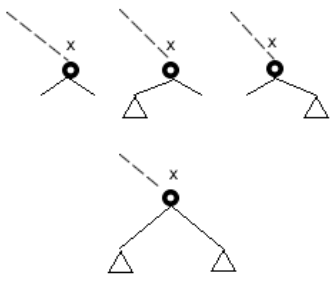
*Caso Base: Para insertar **x** en un árbol **Vacio**, se devolverá un nodo que contendrá a **x** y tendrá los subárboles izquierdo y derecho **Vacio**.*

*Inducción: Para insertar **x** en un árbol con valor en la raíz y, donde $x \neq y$, insertar **x** recursivamente en el subárbol izquierdo si $x<y$ o insertar **x** recursivamente en el subárbol derecho si $x>y$. Devolver la copia del árbol con sus respectivos subárboles izquierdo o derecho modificados.*

```
addTree::(Ord a)=> a -> Bintree a -> Bintree a
addTree x EmptyBT = NodoBT x EmptyBT EmptyBT
addTree x (NodoBT y lf rt) | x == y    = NodoBT y lf rt
                           | x < y     = NodoBT y (addTree x lf) rt
                           | x > y     = NodoBT y lf (addTree x rt)
```

Eliminación o Borrado:

Para escribir el proceso de borrado, debemos hacer algunas consideraciones con respecto a las distintas situaciones que nos podemos encontrar.



En este gráfico hemos representado las situaciones que se pueden dar con respecto a la eliminación del valor ***X***, del árbol binario.

Como vemos en el caso (a), *Nodo(x, Vacio, Vacio)*, la eliminación de **x** puede hacerse sencillamente eliminando el nodo que lo contiene y devolviendo el valor **Vacio** al nodo precedente (marcado con líneas de puntos).

En los casos (b) y (c), *Nodo(x, izquierdo, Vacio)* o *Nodo (x, Vacio, derecho)*, vemos que la eliminación de ***X*** significa que su lugar será ocupado por el subárbol izquierdo o por el derecho respectivamente.

En el caso (d), vemos que no se puede hacer desaparecer al nodo físicamente, ya que alteraría la estructura del árbol, por lo tanto se debe eliminar X lógicamente del árbol, y la manera adecuada es encontrar un valor de reemplazo para ocupar el nodo de X. Para ello observemos los siguiente, veamos el árbol que contiene a x en la raíz (*Nodo (x, T1, T2)*). Sean $T1$ y $T2$ sus subárboles, izquierdo y derecho respectivamente.

Sea t_{1max} el valor máximo contenido en $T1$ y sea t_{2min} el valor mínimo contenido en $T2$. En este caso se verifica que:

$$t_1 < t_{1max} < x < t_{2min} < t_2$$

$$\forall t_1 \in T1\text{-}\{t_{1max}\} \wedge t_2 \in T2\text{-}\{t_{2min}\}$$

En esta relación de orden sin alterar la estructura lógica de los datos vemos que:

$$t_1 < t_{1max} < t_{2min} < t_2$$

Esto significa que todo elemento de $T1$, es menor que t_{2min}, o que todo elemento de $T2$ es mayor que t_{1max}.

Por lo tanto podemos usar a cualquiera de esos valores para reemplazar a ***x***, sin que se altere la estructura del árbol, nos inclinaremos por la versión de encontrar el mínimo en el subárbol derecho.

Para esto necesitaremos desarrollar una función auxiliar, que denominaremos **`minTree`** , la cual aplicada a un árbol, devuelve el valor mínimo almacenado en el mismo, y el árbol modificado donde este mínimo se ha eliminado del conjunto (la función devolverá una tupla).

El valor más chico siempre se encuentra buscándolo recursivamente hacia la izquierda, y lo hallaremos cuando se arribe a un nodo cuyo subárbol izquierdo es **Vacio**.

```
minTree   :: (Ord a) => Bintree a -> (a, Bintree a)
minTree (NodoBT  v EmptyBT rt) = (v,rt)
minTree (NodoBT  v    lf   rt)=  let (x, new_lf) = minTree lf
                                 in
                                      (x, NodoBT v new_lf rt)
```

Ahora sí podemos analizar el algoritmo de eliminación de un valor que se encuentra en un árbol binario.

Caso Base: *No se necesita hacer nada para eliminar a **x** de un árbol **Vacio**, retornar el árbol tal cual esta.*
*Para borrar a **x** de un árbol cuya raíz contiene al valor **x**, modificar el árbol manteniendo la propiedad **BST**.*

- *Si la raíz tiene al menos un subárbol **Vacio**, reemplazar todo el árbol por el otro subárbol.*
- *Si la raíz tiene dos subárboles **no Vacios**:*
  *Reemplazar a **x** por el resultado de la función **minTree**, aplicada al subárbol derecho y retornar un árbol que tiene a **x** reemplazado convenientemente, el subárbol izquierdo como estaba y el derecho modificado.*

Inducción: *Para borrar a x de un árbol cuya raíz tiene el valor y, donde $x \neq y$, recursivamente borrar a x del subárbol izquierdo si $x < y$ o del derecho si $x > y$.*

```
delTree::(Ord a)=> a -> Bintree a -> Bintree a
delTree x EmptyBT = EmptyBT
delTree x (NodoBT y lf EmptyBT)
          | x == y  = lf
delTree x (NodoBT y EmptyBT rt)
          | x == y = rt
delTree x (NodoBT y lf rt)
        |x < y   = NodoBT y (delTree x lf) rt
        |x > y   = NodoBT y lf (delTree x rt)
        |x == y  = let (k,wt) = minTree (rt)
                   in (NodoBT k lf wt)
```

Nota y Ejercicios para el lector. A los efectos de practicar la manipulación de árboles, escribir las funciones inOrder, preOrder y postOrder que realizan un listado de los elementos del árbol, en Orden, en Pre Orden y Post Orden respectivamente. Y que poseen la siguiente definición de tipo.

```
InOrder:: (Ord a)=> Bintree a -> [a]
```

Ahora que ya hemos definido una estructura aceptable para ser utilizada como diccionario, veamos cómo sería la definición del TDA Diccionario utilizando estructura de Árbol binario.

```
newtype Dict a = Dicc (Bintree a) deriving Show

    mkDict:: (Ord a)=> Dict a
    mkDict = Dicc (mkNewTree)

    insertDict :: (Ord a)=> a -> Dict a -> Dict a
    insertDict x (Dicc t) = Dicc (addTree x t)

    inDict::(Ord a) => a -> Dict a -> Bool
    inDict x (Dicc t) = inTree x t

    delDict::(Ord a)=> a -> Dict a -> Dict a
    delDict x (Dicc t) = Dicc ( delTree x t)
```

El polimorfismo del TDA árbol binario de búsqueda, está garantizado para cualquier tipo ***a*** que sea una instancia de la clase **Ord**.

TDA Array

Un array se utiliza para almacenar y recuperar un conjunto de elementos (todos del mismo tipo), donde cada elemento tiene asociado un índice único.

En Haskell los arrays no forman parte de la biblioteca estándar Prelude, están provistos por un módulo de biblioteca denominado *Array* (Su implementación está fuera de los alcances de este curso).
La forma de incorporarlo a nuestro entorno de programa es mediante la directiva

import Data.Array

Creación de Arrays

Los arrays son creados mediante tres funciones predefinidas llamadas `array , listArray, accumArray`.

array *limites lista_de_asociaciones*

Este es el mecanismo fundamental de creación de arrays. El primer argumento *limites* indica el índice inferior y superior en el arreglo. Por ejemplo un vector con inicio en cero y con 5 elementos tendrá límites `(0,4)` una matriz bidimensional de 10 x 10, tendrá como límites `((1,1),(10,10))`.
El segundo parámetro es una lista de asociaciones, donde una *asociación* es de la forma `(i,x)` significando que el valor del elemento *i-esimo* del array es *x*.
Frecuentemente se utiliza listas por comprensión para definir la lista de asociaciones.
Ejemplos.

```
a' = array (1,4) [(3,'c'), (1,'a'),(2,'b'),(4,'d')]
m = array ((1,1),(2,3)) [((i,j),(i*j)) | i<-[1..2], j<-[1..3]]
```

El tipo de un array está denotado por `Array a b`, siendo `a` el tipo del índice, el que deberá ser una instancia de la clase *Ix*, y `b` el tipo de los elementos almacenados.

```
a:: Array Int Char
m:: Array (Int,Int) Int
```

Un array está indefinido si se especifica un índice fuera de los límites; si dos asociaciones hacen referencia al mismo índice, su valor queda indefinido.

listArray *limites lista_de_valores*

Se utiliza en los casos frecuentes donde la lista de elementos ya se encuentra en orden indexado.

```
b' = listArray (1,4) "face"
a'' = listArray (0,5) [12,6,15,11,3,2]
```

La última función

accumArray *función_de_acumulacion valor_inicial limites lista_de_asociaciones*

Esta función relaja la condición de conflictividad cuando en la lista de asociaciones hay índices repetidos.
Permite acumular en el mismo índice los elementos que colisionan, esa acumulación se hace utilizando la función de acumulación provista como argumento (generalmente es la adición normal `(+)`).
Por ejemplo

```
a = accumArray (+) 100 (1,3) [(2,2),(2,3),(1,67)]
```

Obtenemos

```
⇨  array (1,3) [(1,167),(2,105),(3,100)]
```

Como se ve, los conflictos se acumularon mediante suma (en el índice 2 tenemos 2+3=5. Y como arbitrariamente se eligió como valor inicial 100, a cada elemento del arreglo se le acumuló 100, incluso en el índice 3 donde no se proporcionó valor alguno).
Otro ejemplo

```
b = accumArray (++) "¡" (1,2)[(1,"Ho"),(1,"la")]
```

Se obtiene

```
⇨  array (1,2) [(1,"¡Hola"),(2,"¡")]
```

Usando los Arrays

Para acceder a un valor mediante un índice se utiliza el operador binario (**!**). Las funciones provistas permiten recuperar los límites (***bounds***), los índices (***indices***), los elementos (***elems***) y las asociaciones (***assocs***) de un array.

Por ejemplo consideremos el mismo array bidimensional definido con anterioridad.

```
m = array ((1,1),(2,3)) [((i,j),(i*j)) | i<-[1..2], j<-[1..3]]
```

Vamos a ilustrar el uso de las operaciones mencionadas

```
m!(1,2)      =>    2
bounds  m    =>    ((1,1),(2,3))
indices m    =>    [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
elems m      =>    [1,2,3,2,4,6]
assocs m     =>    [((1,1),1),((1,2),2),((1,3),3),((2,1),2),((2,2),4),((2,3),6)]
```

Es posible obtener un array actualizado (en la manera funcional, recordar no variables sólo etiquetas), en realidad se obtiene una copia del array original sustituyendo las asociaciones indicadas en la lista.

Array_original **//** *lista_de_asoc_a_sustituir*

Por ejemplo en

```
a' = array (1,4) [(3,'c'), (1,'a'),(2,'b'),(4,'d')]
            ⇨  array (1,4) [(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

La expresión

```
            a'''= a' // [(3,'r')]
```

Da como resultado un nuevo array similar a `a'`, donde su sustituyó o actualizó el elemento con índice 3 al valor `'r'`.

```
            ⇨  array (1,4) [(1,'a'),(2,'b'),(3,'r'),(4,'d')]
```

Otra función de utilidad es

accum *función_de_acumulacion array lista_de_asociaciones*

Por ejemplo si volvemos sobre el array bidimensional previamente definido

```
m = array ((1,1),(2,3)) [((i,j),(i*j)) | i<-[1..2], j<-[1..3]]
   ⇨    array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),((2,1),2),((2,2),4),((2,3),6)]

accum (+) m [((1,1),4), ((2,2),8)]
   ⇨    array ((1,1),(2,3)) [((1,1),5),((1,2),2),((1,3),3),((2,1),2),((2,2),12),((2,3),6)]
```

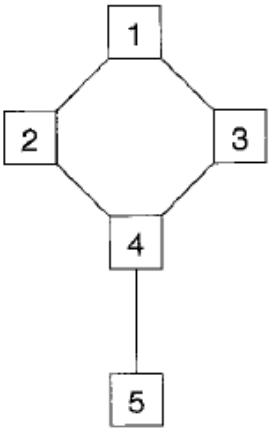
El resultado es una nueva matriz, idéntica a la original `m`, excepto los elementos (1,1) y el (2,2) a los cuales se les ha sumado 4 y 8 respectivamente.

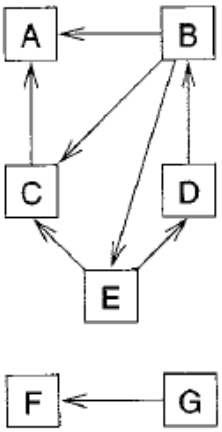
TDA Grafo

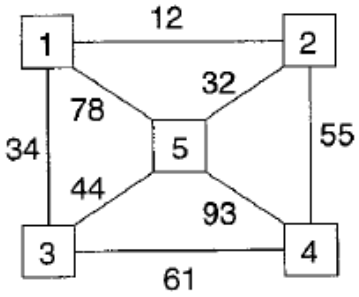
Definiciones y Terminología

Formalmente un grafo denotado como $G = \{V, E\}$, consiste en un conjunto finito de nodos o vértices $\boldsymbol{V}$, y un conjunto de aristas $\boldsymbol{E}$, donde una arista $\boldsymbol{i\ j}$ representa una conexión entre el vértice $\boldsymbol{i}$ y el vértice $\boldsymbol{j}$.
En otras palabras, un grafo puede ser definido *como un conjunto y una relación definida sobre ese conjunto*, donde cada elemento del conjunto corresponde a un vértice del grafo, y donde existe una relación entre dos elementos si existe una arista entre los correspondientes vértices. La relación es ***simétrica*** para el Grafo no dirigido.
Si la arista tiene dirección, se denomina **Arco**, y la relación es ***asimétrica*** (Grafo dirigido).
Nos referiremos a la cantidad de vértices y de aristas como **|V|** y **|E|** respectivamente.

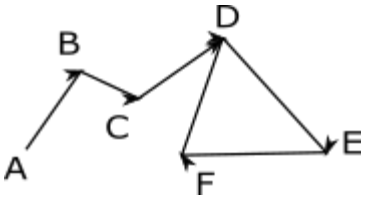
Un **grafo dirigido**, del mismo modo que un árbol, es una estructura de datos no lineal ya que cada nodo que puede tener uno o más antecesores. La diferencia con un árbol es que en éste último, salvo la raíz, cada nodo tiene un único ancestro.







Un ***camino*** $v_1\ v_n$, es una secuencia de vértices $v_1, v_2, \ldots, v_n$ donde cada par $v_{i-1}\ v_i$ es una arista o arco del grafo.
Un ***camino simple*** es un camino donde todos los v_i son distintos.
Un ***ciclo*** es un camino que al explorarlo presenta un arco de retroceso (se presenta un arco que vuelve sobre algún v_i del camino).

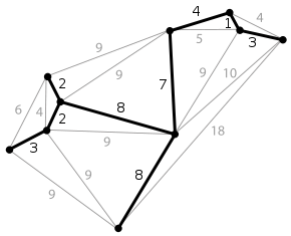


En este ejemplo, el camino actual es A, B, C, D, E, F, D . Al recorrerlo encontramos un arco que nos conduce a un vértice que ya pertenece al camino, F D. Esto es lo que se denomina un arco de retroceso, por tanto existe un ciclo D,E,F,D.

Decimos que un grafo es ***acíclico***, si para todo camino del mismo no contiene ciclos.
Si existe un camino cualquiera entre cualquier par de vértices del grafo, decimos que el grafo es ***conexo***.
(Para todo $v_i\ v_j$ pertenecientes a $\boldsymbol{V}$ existe un camino que va de v_i a v_j).Observar que para un grafo No Dirigido, la existencia de un camino $v_i\ v_j$ implica la existencia de un camino $v_j\ v_i$. No sucede lo mismo si el grafo es Dirigido.

Un conjunto de nodos conectados por una arista a un nodo en particular se dice que son ***adyacentes*** a ese nodo.
Se define el ***grado*** o ***valencia*** de un vértice $\boldsymbol{g(x)}$, como la cantidad de aristas que inciden a él.
En un ***Grafo Dirigido*** se diferencia el ***grado de entrada*** $\boldsymbol{g_{in}(x)}$, como el número de aristas que tiene al vértice $\boldsymbol{x}$ como vértice final, y el ***grado de salida*** $\boldsymbol{g_{out}(x)}$, como el número de aristas que tiene al vértice $\boldsymbol{x}$ como vértice inicial, de forma que $g(x) = g_{in}(x) + g_{out}(x)$.
Un ***grafo regular*** es un grafo donde cada vértice tiene el mismo grado o valencia. Un grafo regular con vértices de grado $\boldsymbol{k}$ es llamado grafo ***k-regular*** o grafo regular de grado $\boldsymbol{k}$.

Es común que las aristas tengan asociados valores o pesos, en ese caso decimos que se trata de un ***grafo ponderado***. El costo de un camino se define como la sumatoria de los pesos de las aristas que lo componen.



Dado un *grafo conexo y no dirigido*, un **árbol abarcador** de ese grafo es un subgrafo, que debe ser un árbol y vincular a todos los vértices del grafo inicial. El costo de dicho árbol está dado por la suma de los costos de cada rama del árbol.

El TDA Grafo

Vamos definir el TDA, para grafos ponderados, en el caso de grafos no ponderados es totalmente válida ignorando el peso.
Debemos encontrar una definición adecuada para el tipo (***Grafo n w***), donde:

- ***n*** es el tipo de los vértices: Asumimos que será una instancia de la clase Index (Class Ix). Esta suposición la hacemos para poder utilizar los vértices como índice de un array.
- ***w*** es el tipo de los pesos: Supondremos que los pesos son instancias de la clase Numero (Class Num).

Dados estos tipos, las operaciones necesarias para manipular el TDA Grafo serán:

```
mkGraph :: (Ix n , Num w) => Bool -> (n,n) -> [(n,n,w)] -> (Grafo n w)
```
Toma los limites inferior y superior del conjunto de índices, una lista de aristas (cada arista está dada por una tupla, origen - destino – peso) y retorna un Grafo. El primer argumento Booleano indica si el grafo es dirigido, False indica que se debe agregar un arco en ambas direcciones (Gafo no dirigido).

```
adjacente :: (Ix n , Num w) => (Grafo n w) -> n -> [n]
```
Retorna la lista de nodos adyacentes a un nodo dado.

```
nodos :: (Ix n , Num w) => (Grafo n w) -> [n]
```
Retorna la lista de nodos del grafo.

```
aristasD, aristasU :: (Ix n , Num w) => (Grafo n w) -> [(n,n,w)]
```
Retorna una lista de todas las aristas de un grafo dirigido y de uno no dirigido respectivamente.

```
aristaIn :: (Ix n , Num w) => (Grafo n w) -> (n,n) -> Bool
```
Retorna True si la arista dada existe en el grafo.

```
peso :: (Ix n , Num w) => (Grafo n w) -> n -> n -> w
```
Retorna el peso de la arista cuyo origen y destino se proporcionan.

Implementación del TDA Grafo

Existen varias formas de representar un grafo.
Una es mediante matrices de costos o conexión y mediante listas de adyacencia, elegiremos ésta última opción.
Vamos a obviar la posibilidad de utilizar Array para crear una lista de adyacencias y recurriremos a la lista.
Así definimos:

```
-- Peso de una arista, puede ser un entero o infinito
data Peso  = Infinito | Costo Float
     deriving (Show,Eq)


-- Definimos la relación de menor entre pesos
-- De aquí en adelante se puede hacer p < q, donde p y q tienen tipo Peso
instance Ord Peso  where
     Costo a < Costo b = a < b
     Infinito < Costo _ = False
     Costo _ < Infinito = True
     Infinito < Infinito = False
     a <= b = a < b || a == b
     a >  b = not (a < b)
     a >= b = not (a <=b)
---------------------------------
type Vertice  =  Int

--- Una lista de Adyacencia de un vértice es una tupla que tiene,
--- Vertice origen y lista de destinos con su respectivo Peso
type ListaAdy = (Vertice, [(Vertice, Peso)])

--- Entonces diremos que un grafo es una lista de listas de adyacencias
type Grafo    = [ListaAdy]
```

Ejercicio Propuesto

Desarrollar el módulo Grafo. Escribir el código de las funciones propuestas para la manipulación del TDA Grafo.

```
module Graph (Graph, mkGraph, adyacente, nodos, aristas, peso, aristaIn)where
```

Algoritmos sobre Grafos (Algunos)

Dirigidos (Caminos y recorridos)

- Dijkstra
- BPF
- Aciclidad

No Dirigidos (Árbol Abarcador de Costo Mínimo)

- Prim
- Kruskal

Bibliografia:

Algorithms - A Functional Programming Haskell Approach
Fethi Rabhi – Guy Lapalme

Real World Haskell
Bryan O´Sullivan, John Goerzen, Don Steward