TIPOS ABSTRACTOS DE DATOS EN HASKELL

Haskell λ

A Purely Functional Language

Fernando Benito

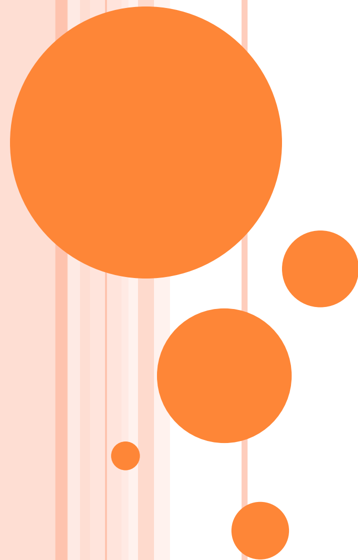
25726083-P

Programación Declarativa Avanzada

Ingeniería Informática

Dpto. LCC – UMA

Índice General



1. Introducción a Haskell
2. Tipos abstractos de datos en Haskell
3. TAD Grafo
4. TAD Montículos
5. Bibliografía

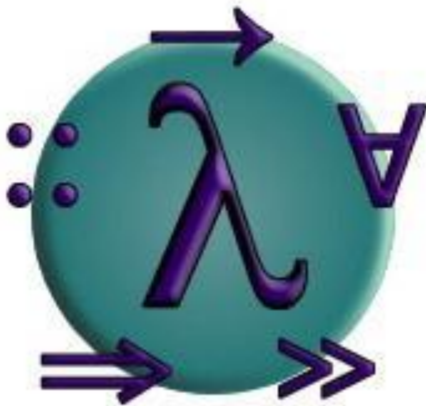
Introducción a Haskell

1. **Introducción**
2. **Historia**
3. **Variantes**

Introducción a Haskell

1. Lenguaje funcional puro de propósito general
2. Incorpora:
   1. Facilidad para la definición de TADs
   2. Utilización de módulos
3. Mayor ventaja: soporte para tipos de datos y funciones recursivas, listas, tuplas, patrones, etc.

Introducción a Haskell
historia



1. Miranda (1985)
2. Haskell 1.0 (1990)
3. Haskell 98 (1997)
4. Publicación del estándar Haskell 98 (1999)
5. Versión revisada de Haskell 98 (2003)
6. Haskell Prime (2006-Actualidad)

Introducción a Haskell
Variantes

1. Versiones paralelas
   - Parallel Haskell (MIT, Glasgow)
   - Distributed Haskell y Eden

2. Versiones orientadas a objetos
   - Haskell++
   - Mondrian

3. Versión educativa
   - Gofer: desarrollada por Mark Jones que fue suplantada por Hugs.

Índice General

1. Introducción a Haskell
2. **Tipos abstractos de datos en Haskell**
3. TAD Grafo
4. TAD Montículos
5. Bibliografía

Tipos abstractos de datos en Haskell

1. **Introducción**
2. **TADs**

Tipos abstractos de datos en Haskell
Introducción

- Una parte importante del lenguaje Haskell lo forma el sistema de tipos que es utilizado para detectar errores en expresiones y definiciones de función.
- Cada tipo tiene asociadas un conjunto de operaciones que no tienen significado para otros tipos.
- Es posible asociar un único tipo a toda expresión bien formada ⟶ lenguaje fuertemente tipado.

Tipos abstractos de datos en Haskell
TADs

- Un TAD es una abstracción para destacar las operaciones que podemos realizar con los datos, ocultando los detalles de su representación.

- Para un TAD se determinan:
  - Las operaciones y el tipo de cada una (interfaz).
  - Propiedades que interrelacionan las operaciones (axiomas).

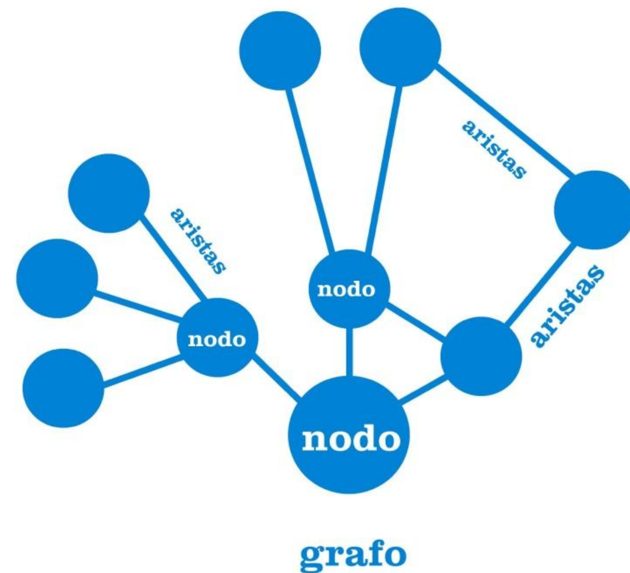
- La signatura permite decidir qué mezcla de operaciones tiene sentido y los axiomas permiten operar con los datos en forma abstracta.
  - signaturas + axiomas = especificación

Índice General

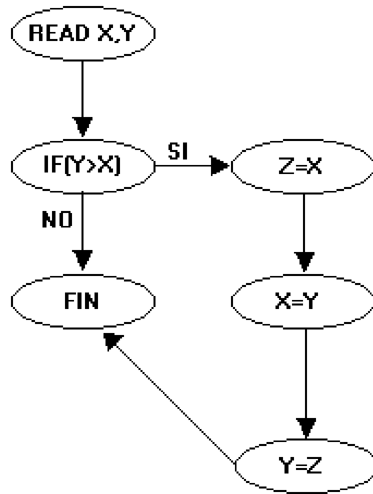
1. Introducción a Haskell
2. Tipos abstractos de datos en Haskell
3. **TAD Grafo**
4. TAD Montículos
5. Bibliografía

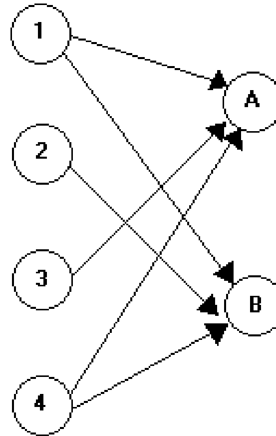
TAD GRAFO

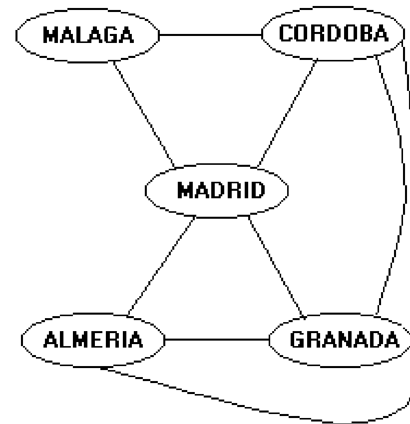
1. **Definiciones y terminología**
2. **Signatura del TAD grafos**
3. **Implementación**
   - Vectores de adyacencia
   - Matrices de adyacencia
4. **El algoritmo de Kruskal**
   - Ejemplo
   - Implementación
   - Complejidad



TAD Grafo
Definiciones y terminología







- Un grafo en el ámbito de las ciencias de la computación es una estructura de datos, en concreto un TAD, que consiste en un conjunto de nodos (vértices) y un conjunto de arcos (aristas) que establecen relaciones entre los nodos.

TAD GRAFO
DEFINICIONES Y TERMINOLOGÍA

- Un **grafo** G es un par (V,A) donde V es el conjunto de los vértices (o nodos) y A el conjunto las aristas.
- Una **arista** del grafo es un par de vértices.
- Un **arco** es una arista dirigida.
- **V** es el número de vértices.
- **A** es el número de aristas.
- Un **camino** de v1 a vn es una sucesión de vértices v1, v2, . . . , vn tal que para todo i, vi-1 vi es una arista del grafo.

TAD Grafo
Definiciones y terminología

- Un **camino simple** es un camino tal que todos sus vértices son distintos.
- Un **ciclo** es un camino tal que v1 = vn y todos los restantes vértices son distintos.
- Un **grafo acíclico** es un grafo sin ciclos.
- Un **grafo conexo** es un grafo tal que para cualquier par de vértices existe un camino del primero al segundo.
  - Un árbol es un grafo acíclico conexo.
- Un **grafo ponderado** es un grafo cuyas aristas tienen un peso.

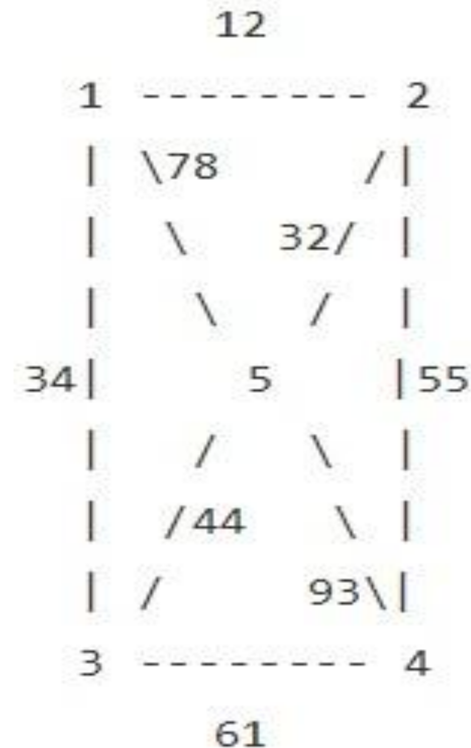
TAD Grafo
Signatura del TAD grafos

```
creaGrafo  :: (Ix v,Num p) => Bool -> (v,v) -> [(v,v,p)]
                                  -> Grafo v p
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
nodos      :: (Ix v,Num p) => (Grafo v p) -> [v]
aristasND  :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristasD   :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristaEn   :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
peso       :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
```

TAD Grafo
Signatura del TAD grafos

- Ejemplo

creaGrafo False
 (1,5)
[(1,2,12),(1,3,34),
(1,5,78),(2,4,55),
(2,5,32),(3,4,61),
(3,5,44),(4,5,93)]



TAD Grafo
Implementación
Vectores de adyacencia

```haskell
module GrafoConVectorDeAdyacencia
    (Grafo,
     creaGrafo,  -- (Ix v,Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristasND,  -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristasD,   -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where

-- ------------------------------------------------------------------------
-- Librerías auxiliares                                                 --
-- ------------------------------------------------------------------------

import Data.Array
```

TAD Grafo
Implementación
Vectores de adyacencia

(Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p

```
type Grafo v p = Array v [(v,p)]
```

TAD Grafo
Implementación

Vectores de adyacencia

```
-- grafoVA es el grafo
--              12
--        1 -------- 2
--        | \78     /|
--        |  \   32/ |
--        |   \   /  |
--      34|     5    |55
--        |   /   \  |
--        |  /44   \ |
--        | /     93\|
--        3 -------- 4
--              61
-- representado mediante un vector de adyacencia.
grafoVA = array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                       (2,[(1,12),(4,55),(5,32)]),
                       (3,[(1,34),(4,61),(5,44)]),
                       (4,[(2,55),(3,61),(5,93)]),
                       (5,[(1,78),(2,32),(3,44),(4,93)])]
```

TAD Grafo Implementación
Vectores de adyacencia

- (creaGrafo d cs as) es un grafo (dirigido si d es True y no dirigido en caso contrario), con el par de cotas cs y listas de aristas as (cada arista es una terna formada por los dos vértices y su peso).

```
creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo d cs vs =
    accumArray
     (\xs x -> xs++[x]) [] cs
     ((if d then []
       else [(x2,(x1,p))|(x1,x2,p) <- vs, x1 /= x2]) ++
      [(x1,(x2,p)) | (x1,x2,p) <- vs])
```

TAD Grafo
Implementación
Vectores de adyacencia

- grafoVA' es el mismo grafo que grafoVA pero creado con creaGrafo.

```
grafoVA' = creaGrafo False (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                  (2,4,55),(2,5,32),
                                  (3,4,61),(3,5,44),
                                  (4,5,93)]
```

TAD Grafo
Implementación
Vectores de adyacencia (operaciones)

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes g v = map fst (g!v)
```

- adyacentes grafoVA' 4 == [2,3,5]

```
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos g = indices g
```

- nodos grafoVA' == [1,2,3,4,5]

```
peso :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
peso x y g = head [c | (a,c) <- g!x , a == y]
```

- peso 1 5 grafoVA' == 78

TAD Grafo
Implementación
Vectores de adyacencia (operaciones)

```
aristaEn :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn g (x,y) = elem y (adyacentes g x)
```

- aristaEn grafoVA' (5,1) == True
- aristaEn grafoVA' (4,1) == False

```
aristasD :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristasD g = [(v1,v2,w) | v1 <- nodos g , (v2,w) <- g!v1]
```

```
aristasND :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristasND g =
    [(v1,v2,w) | v1 <- nodos g , (v2,w) <- g!v1 , v1 < v2]
```

TAD Grafo
Implementación
Matrices de adyacencia

```haskell
module GrafoConMatrizDeAdyacencia
    (Grafo,
     creaGrafo,  -- (Ix v,Num p) => Bool -> (v,v) -> [(v,v,p)] -> Grafo v
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristasND,  -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristasD,   -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where

-- ---------------------------------------------------------------------
-- Librerías auxiliares                                               --
-- ---------------------------------------------------------------------

import Data.Array
```

TAD Grafo
Implementación
Matrices de adyacencia

(Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p

```
type Grafo v p = Array (v,v) (Maybe p)
```

TAD Grafo
Implementación
Matrices de adyacencia

grafoMA es la representación del grafo del ejemplo anterior mediante una matriz de adyacencia.

```
grafoMA = array ((1, 1), (5, 5))
               [((1, 1),Nothing), ((1, 2),Just 10), ((1, 3),Just 20),
                ((1, 4),Nothing), ((1, 5),Nothing), ((2, 1),Nothing),
                ((2, 2),Nothing), ((2, 3),Nothing), ((2, 4),Just 30),
                ((2, 5),Nothing), ((3, 1),Nothing), ((3, 2),Nothing),
                ((3, 3),Nothing), ((3, 4),Just 40), ((3, 5),Nothing),
                ((4, 1),Nothing), ((4, 2),Nothing), ((4, 3),Nothing),
                ((4, 4),Nothing), ((4, 5),Just 50), ((5, 1),Nothing),
                ((5, 2),Nothing), ((5, 3),Nothing), ((5, 4),Nothing),
                ((5, 5),Nothing)]
```

TAD Grafo Implementación

Matrices de adyacencia

- (creaGrafo d cs as) es un grafo (dirigido si d es True y no dirigido en caso contrario), con el par de cotas cs y listas de aristas as (cada arista es una terna formada por los dos vértices y su peso).

```
creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)] -> (Grafo v p)
creaGrafo dir cs@(l,u) as
    = matrizVacia //
      ([((x1,x2),Just w) | (x1,x2,w) <- as] ++
       if dir then []
       else [((x2,x1),Just w) | (x1,x2,w) <- as, x1 /= x2])
    where
      matrizVacia = array ((l,l),(u,u))
                          [((x1,x2),Nothing) | x1 <- range cs,
                                               x2 <- range cs]
```

TAD Grafo Implementación

Matrices de adyacencia

- grafoMA' es el mismo grafo que grafoMA pero creado con creaGrafo.

```
grafoMA' = creaGrafo False (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                  (2,4,55),(2,5,32),
                                  (3,4,61),(3,5,44),
                                  (4,5,93)]
```

TAD Grafo
Implementación
Matrices de adyacencia (operaciones)

```haskell
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes g v1 =
    [v2 | v2 <- nodos g, (g!(v1,v2)) /= Nothing]
```

- adyacentes grafoMA' 4 == [2,3,5]

```haskell
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos g = range (l,u)
    where ((l,_),(u,_)) = bounds g
```

- nodos grafoMA' == [1,2,3,4,5]

```haskell
peso :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
peso x y g  = w where (Just w) = g!(x,y)
```

- peso 1 5 grafoVA' == 78

TAD Grafo
Implementación
Matrices de adyacencia (operaciones)

```
aristaEn :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn g (x,y)= (g!(x,y)) /= Nothing
```

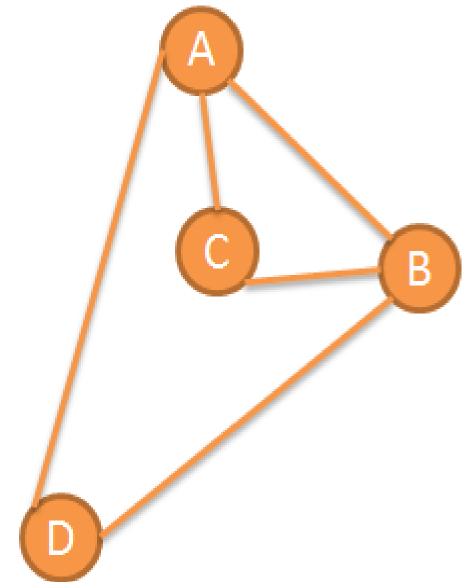
- aristaEn grafoMA' (5,1) == True
- aristaEn grafoMA' (4,1) == False

```
aristasD :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristasD g = [(v1,v2,extrae(g!(v1,v2)))
             | v1 <- nodos g,
               v2 <- nodos g,
               aristaEn g (v1,v2)]
    where extrae (Just w) = w
```

```
aristasND :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristasND g = [(v1,v2,extrae(g!(v1,v2)))
              | v1 <- nodos g,
                v2 <- range (v1,u),
                aristaEn g (v1,v2)]
    where (_,(u,_)) = bounds g
          extrae (Just w) = w
```

TAD Grafo
Algoritmo de Kruskal

- El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado.
- Si el grafo no es conexo, entonces busca un bosque expandido mínimo.
- Busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y el valor total de todas las aristas del árbol es el mínimo.

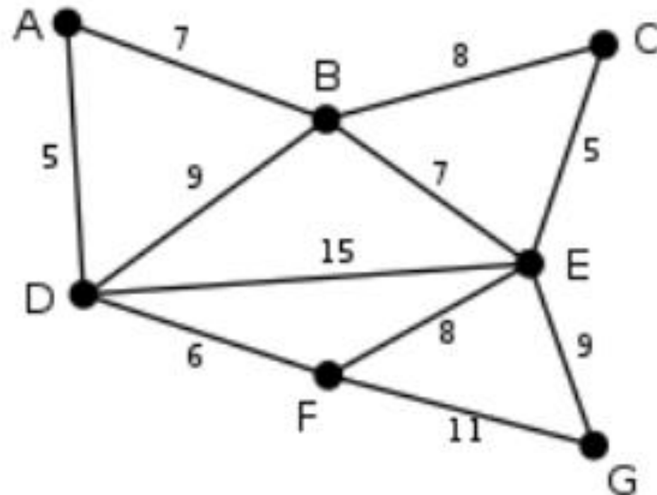


TAD Grafo

Algoritmo de Kruskal (Pseudocódigo)

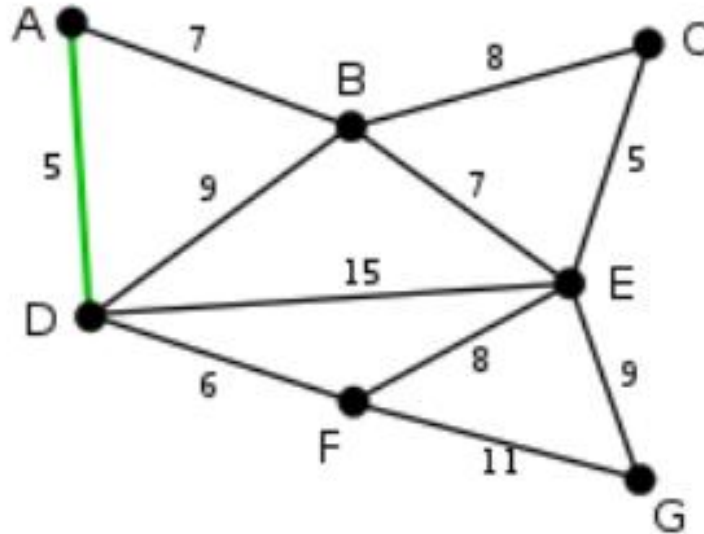
- Se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado
- Se crea un conjunto C que contenga a todas las aristas del grafo
- Mientras C es no vacío
  - eliminar una arista de peso mínimo de C
  - si esa arista conecta dos árboles diferentes
    - se añade al bosque, combinando los dos árboles en un solo árbol
  - en caso contrario
    - se desecha la arista
- Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

TAD Grafo
Algoritmo de Kruskal (Ejemplo I)



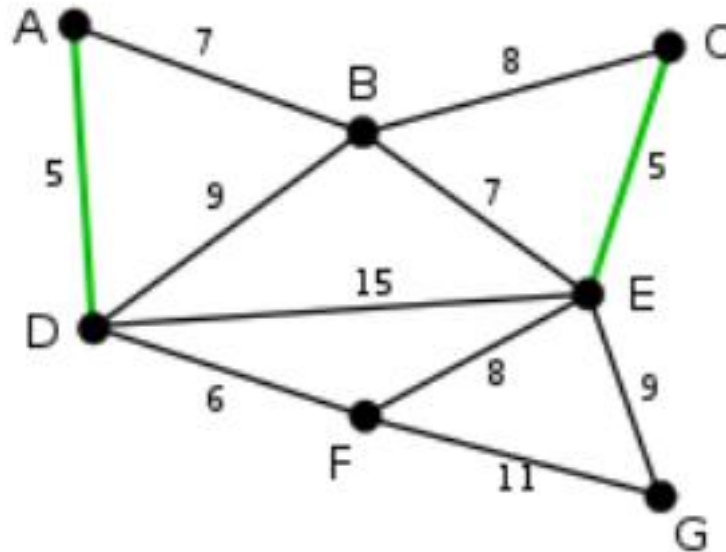
- Este es el grafo original. Los números de las aristas indican su peso. Ninguna de las aristas está resaltada.

TAD GRAFO
ALGORITMO DE KRUSKAL (EJEMPLO II)



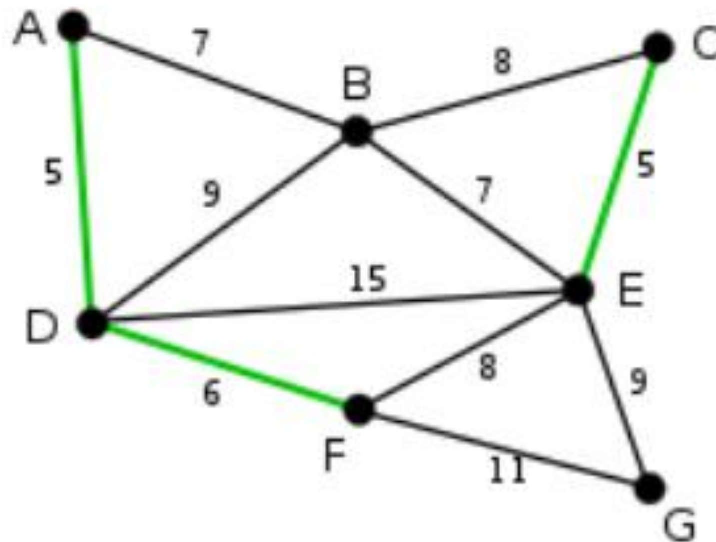
- **AD** y **CE** son las aristas más cortas, con peso 5, y **AD** se ha elegido arbitrariamente, por tanto se resalta.

TAD Grafo
Algoritmo de Kruskal (Ejemplo III)



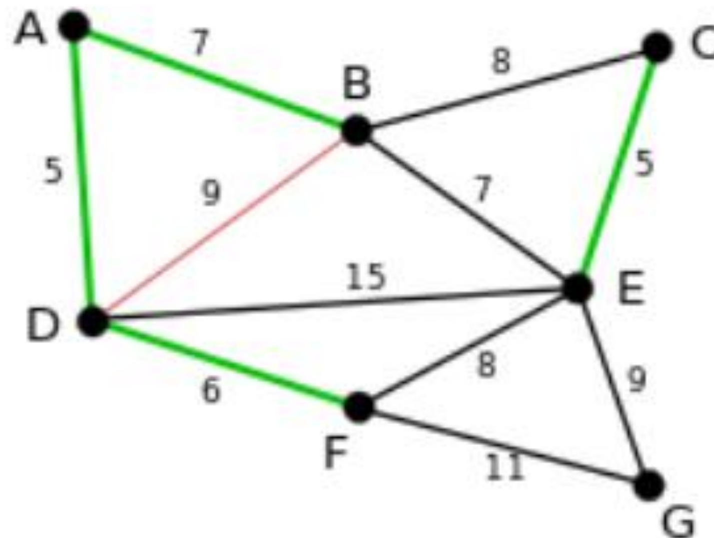
- Sin embargo, ahora es **CE** la arista más pequeña que no forma ciclos, con peso 5, por lo que se resalta como segunda arista.

TAD Grafo
Algoritmo de Kruskal (Ejemplo IV)



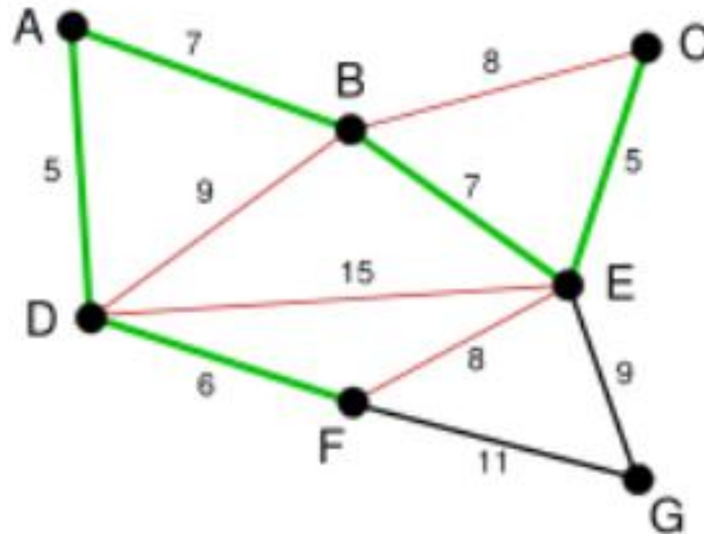
- La siguiente arista, **DF** con peso 6, ha sido resaltada utilizando el mismo método.

TAD Grafo
Algoritmo de Kruskal (Ejemplo V)



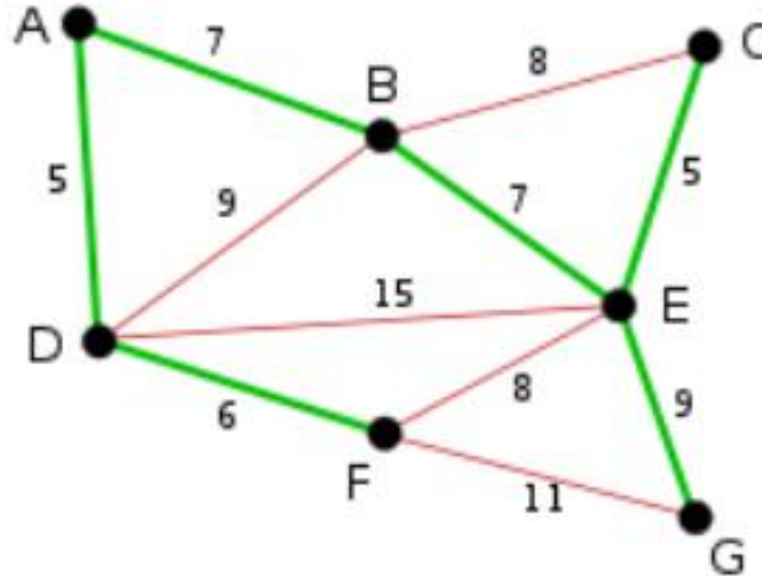
- La siguientes aristas más pequeñas son **AB** y **BE**, ambas con peso 7. **AB** se elige arbitrariamente, y se resalta. La arista **BD** se resalta en rojo, porque formaría un ciclo **ABD** si se hubiera elegido

TAD Grafo
Algoritmo de Kruskal (Ejemplo VI)



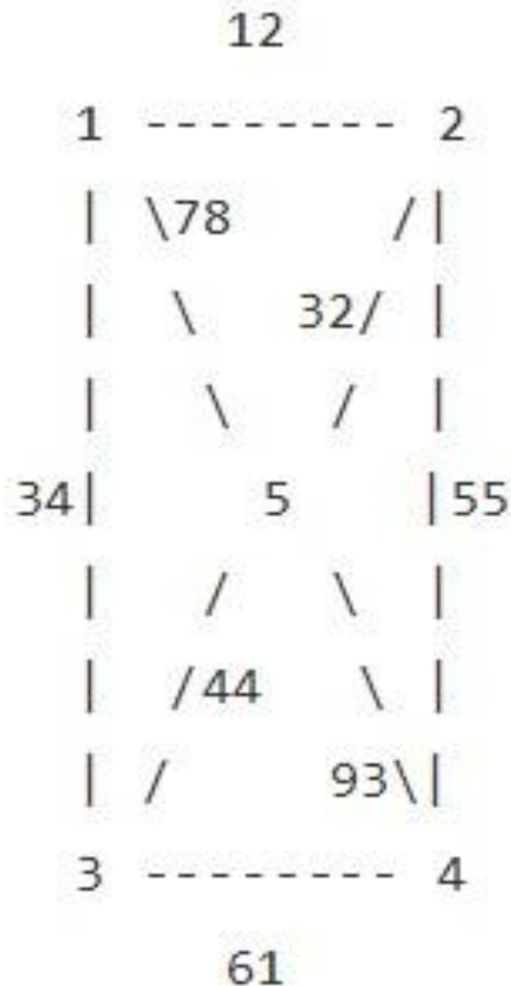
- El proceso continúa marcando las aristas, **BE** con peso 7. Muchas otras aristas se marcan en rojo en este paso: **BC** (ciclo BCE), **DE** (ciclo DEBA), y **FE** (ciclo FEBAD).

TAD Grafo
Algoritmo de Kruskal (Ejemplo VII)



- Finalmente, el proceso termina con la arista **EG** de peso 9, y se ha encontrado el **árbol expandido mínimo**.

TAD Grafo
Algoritmo de Kruskal (Ejemplo Implementado)



kruskal(creaGrafo True (1,5)
[(1,2,12),(1,3,34),(1,5,78),(2,4,55),
(2,5,32), (3,4,61),(3,5,44),(4,5,93)])

Resultado:

[(55,2,4),(34,1,3),(32,2,5),(12,1,2)]

TAD Grafo
Algoritmo de Kruskal (Complejidad)

- Sea m el número de aristas del grafo y n el número de vértices, el algoritmo de Kruskal muestra una complejidad **O(m log n),** cuando se ejecuta sobre estructuras de datos simples.

TAD Grafo
Algoritmo de Kruskal (Complejidad)

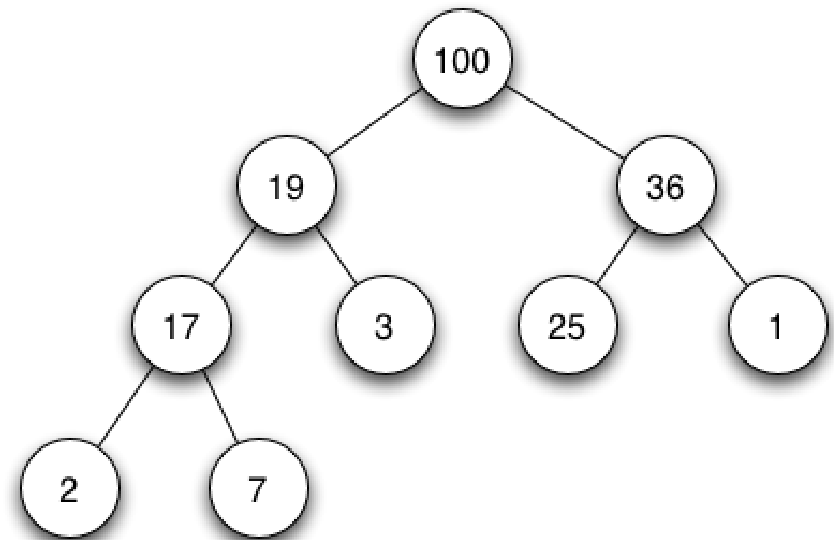
- Primero se ordenan las aristas por su peso usando una ordenación por comparación con una complejidad del orden de **O(m log m)**
- Lo siguiente es usar una estructura de datos sobre conjuntos disjuntos para controlar qué vertices están en qué componentes.
- Es necesario O(m) operaciones ya que por cada arista hay dos operaciones de búsqueda y posiblemente una unión de conjuntos.
- Por tanto, la complejidad total es del orden de **O(m log n).**

Índice General

1. Introducción a Haskell
2. Tipos abstractos de datos en Haskell
3. TAD Grafo
4. **TAD Montículos**
5. Bibliografía

TAD MONTÍCULO

1. **Definición**
2. **Signatura del TAD montículo**
3. **Propiedades del TAD montículo**
4. **Ejemplo**
5. **Implementación**
    1. Funciones auxiliares



TAD Montículo
Definición

- En computación, un montículo (heap en inglés) es una estructura de datos del tipo árbol con información perteneciente a un conjunto ordenado.
- Los montículos son útiles para implementar colas de prioridad.
- Otra ventaja de los montículos es que, por ser árboles completos, simplica su codicación y libera al programador del uso de punteros.

TAD Montículo
Signatura

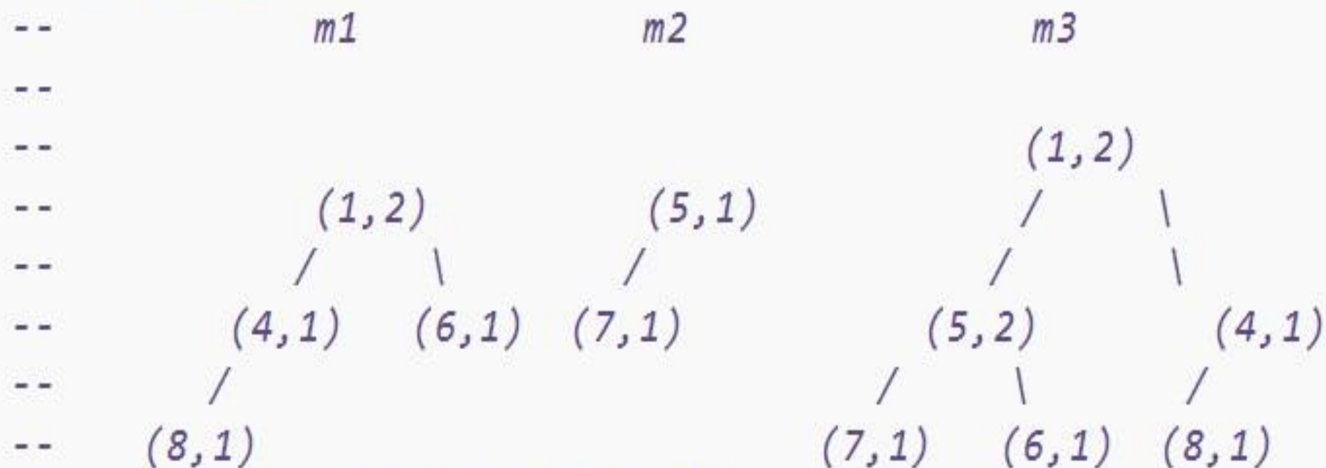
- La signatura de las operaciones del TAD montículo es la siguiente:

```
vacio   :: Ord a => Monticulo a
inserta :: Ord a => a -> Monticulo a -> Monticulo a
menor   :: Ord a => Monticulo a -> a
resto   :: Ord a => Monticulo a -> Monticulo a
esVacio :: Ord a => Monticulo a -> Bool
valido  :: Ord a => Monticulo a -> Bool
```

TAD Montículo
Propiedades

1. esVacio vacio
2. valido (inserta x m)
3. not (esVacio (inserta x m))
4. not (esVacio m) == valido (resto m)
5. resto (inserta x vacio) == vacio
6. x <= menor m -> resto (inserta x m) == m
7. Si m es no vacío y x > menor m, entonces resto (inserta x m) == inserta x (resto m)
8. esVacio m || esVacio (resto m) || menor m <= menor (resto m)

TAD Montículo
Ejemplo



```
-- Gráficamente
--             m1                 m2                    m3
--
--                                                     (1,2)
--              (1,2)            (5,1)                 /     \
--             /     \          /                     /       \
--         (4,1)   (6,1)   (7,1)                  (5,2)          (4,1)
--         /                                     /     \         /
--     (8,1)                                (7,1)   (6,1)  (8,1)
m1, m1', m2, m3 :: Monticulo Int
m1  = foldr inserta vacio [6,1,4,8]
m1' = foldr inserta vacio [6,8,4,1]
m2  = foldr inserta vacio [7,5]
m3 = mezcla m1 m2
```

TAD Montículo Implementación

```
module Monticulo
    (Monticulo,
     vacio,    -- Ord a => Monticulo a
     inserta,  -- Ord a => a -> Monticulo a -> Monticulo a
     menor,    -- Ord a => Monticulo a -> a
     resto,    -- Ord a => Monticulo a -> Monticulo a
     esVacio,  -- Ord a => Monticulo a -> Bool
     valido    -- Ord a => Monticulo a -> Bool
    ) where

import Data.List (sort)
```

TAD Montículo
Implementación (Operaciones I)

```haskell
vacio :: Ord a => Monticulo a
vacio = Vacio
```

```haskell
creaM :: Ord a => a -> Monticulo a -> Monticulo a -> Monticulo a
creaM x a b | rango a >= rango b = M x (rango b + 1) a b
            | otherwise          = M x (rango a + 1) b a
```

```haskell
mezcla :: Ord a =>  Monticulo a -> Monticulo a -> Monticulo a
mezcla m Vacio = m
mezcla Vacio m = m
mezcla m1@(M x _ a1 b1) m2@(M y _ a2 b2)
     | x <= y    = creaM x a1 (mezcla b1 m2)
     | otherwise = creaM y a2 (mezcla m1 b2)
```

TAD Montículo
Implementación (Operaciones II)

```haskell
inserta :: Ord a => a -> Monticulo a -> Monticulo a
inserta x m = mezcla (M x 1 Vacio Vacio) m
```

```haskell
menor  :: Ord a => Monticulo a -> a
menor (M x _ _ _) = x
menor Vacio       = error "menor: monticulo vacio"
```

```haskell
resto :: Ord a => Monticulo a -> Monticulo a
resto Vacio       = error "resto: monticulo vacio"
resto (M x _ a b) = mezcla a b
```

```haskell
esVacio :: Ord a => Monticulo a -> Bool
esVacio Vacio = True
esVacio _     = False
```

TAD Montículo
Implementación (Operaciones III)

```
valido :: Ord a => Monticulo a -> Bool
valido Vacio = True
valido (M x _ Vacio Vacio) = True
valido (M x _ m1@(M x1 n1 a1 b1) Vacio) =
    x <= x1 && valido m1
valido (M x _ Vacio m2@(M x2 n2 a2 b2)) =
    x <= x2 && valido m2
valido (M x _ m1@(M x1 n1 a1 b1) m2@(M x2 n2 a2 b2)) =
    x <= x1 && valido m1 &&
    x <= x2 && valido m2
```

- **(valido m)** se verifica si m es un montículo; es decir, es un árbol binario en el que los valores de cada nodo es menor o igual que los valores de sus hijos.

TAD Montículo
Implementación (Funciones auxiliares I)

```haskell
menorTodos:: Ord a => a -> Monticulo a -> Bool
menorTodos x Vacio       = True
menorTodos x (M y n a b) = x <= y && valido (M y n a b)
```

```haskell
enMonticulo x Vacio        = False
enMonticulo x (M y _ a b)
    | x < y       = False
    | x == y      = True
    | otherwise = enMonticulo x a || enMonticulo x b
```

```haskell
numeroDeNodos :: Ord a => Monticulo a -> Int
numeroDeNodos m
    | esVacio m = 0
    | otherwise = 1 + numeroDeNodos (resto m)
```

TAD Montículo
Implementación (Funciones auxiliares II)

```haskell
filtra :: Ord a => (a -> Bool) -> Monticulo a -> Monticulo a
filtra p m
    | esVacio m = vacio
    | p mm      = inserta mm (filtra p rm)
    | otherwise = filtra p rm
    where mm = menor m
          rm = resto m
```

```haskell
menores :: Ord a => Int -> Monticulo a -> [a]
menores 0 m  = []
menores (n+1) m | esVacio m = []
                | otherwise = menor m : menores n (resto m)
```

```haskell
restantes :: Ord a => Int -> Monticulo a -> Monticulo a
restantes 0 m  = m
restantes (n+1) m | esVacio m = vacio
                  | otherwise = restantes n (resto m)
```

Índice General

1. Introducción a Haskell
2. Tipos abstractos de datos en Haskell
3. TAD Grafo
4. TAD Montículos
5. **Bibliografía**

BIBLIOGRAFÍA

[1] R.S. Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[2] Simon Peyton Jones (editor). Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language. Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106, Feb 1999.

[3] P. Wadler. Monads for Functional Programming In Advanced Functional Programming , Springer Verlag, LNCS 925, 1995.

[4] R. Bird. Introduction to Functional Programming using Haskell. Prentice Hall, New York, 1998.

[5] G.L. Steele Jr. Common Lisp: The Language. Digital Press, Burlington, Mass., 1984.

[6] A.Davie. Introduction to Functional Programming System Using Haskell. Cambridge University Press, 1992.

[7] Ruiz BC., Gutiérrez F., Guerrero P., Gallardo JE. Razonando con Haskell. Una Introducción a la Programación Funcional. Thomson. ISBN: 84-607-1218-4.

[8] Material docente utilizado por José A. Alonso para la docencia de programación funcional. Universidad de Sevilla. http://www.glc.us.es/jalonso

[9] http://es.wikipedia.org/wiki/Haskell