Programación funcional con Haskell
Estructuras Persistentes - BST

Juan Manuel Rabasedas

Estructuras de Datos

- Qué es una estructura de datos?
- Las listas , colas , pilas , árboles , etc.
- Una estructura de datos queda definida si damos:
  - El conjunto de valores que puede tomar,
  - Un conjunto de operaciones definidas sobre estos valores,
  - Un conjunto de propiedades que relacionan todo lo anterior.

Estructuras Funcionales e imperativos

Muchos de los algoritmos más difundidos están escritos para estructuras efímeras.

- En las estructuras efímeras, cualquier cambio significa eliminar información.
- Las estructuras efímeras soportan una sola versión.
- Las estructuras persistentes, soportan un historial de versiones.
- La flexibilidad de las estructuras persistentes tienen un costo:
  - Debemos adaptar las estructuras y algoritmos al modelo persistente.A veces no es posible.
  - La eficiencia de algunos algoritmos para estructuras efímeras, no se van a poder alcanzar con modelos persistentes.

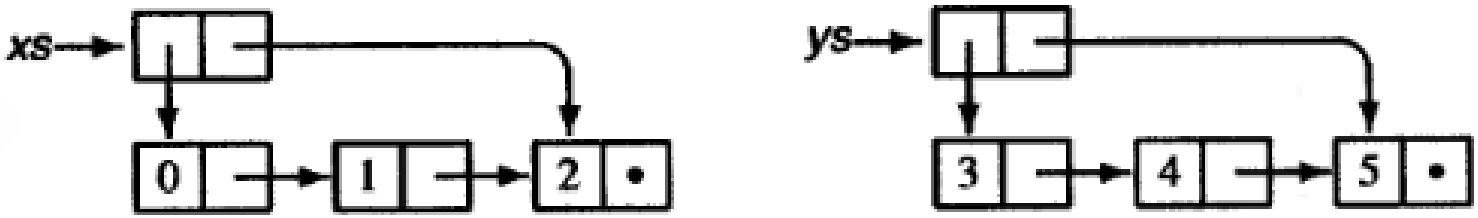
Implementar Persistencia

En un lenguaje funcional puro, todas las estructuras son persistentes.

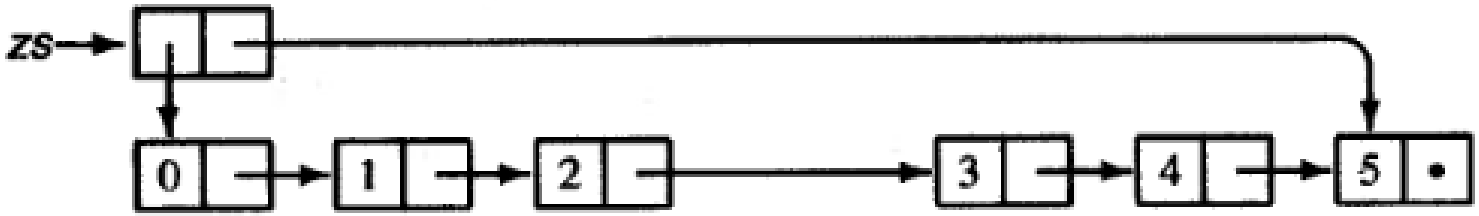
- Las estructuras persistentes al modificarse no se sobrescribe.
- Se duplican los datos y se modifica la copia.
- Las diferentes versiones comparten la información.
  A esto se lo llama sharing.
- El manejo automático de la memoria es imprescindible.
  Para esto es necesario implementar un garbage collection.

Listas simplemente enlazadas efímeras

- Dadas dos listas xs e ys



- Al concatenarlas $zs = xs +\!\!+ ys$ obtenemos:

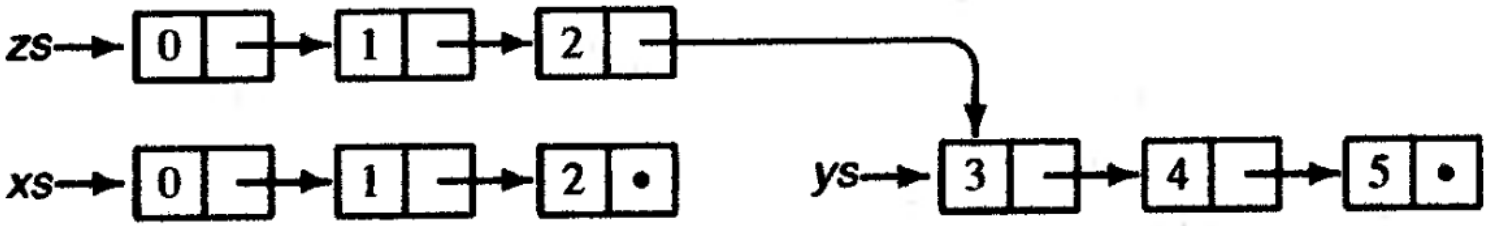


- La operación elimina las listas xs e ys.
- La operación $zs = xs +\!\!+ ys$ no recorrio ningún elemento de las listas.

Listas simplemente enlazadas persistentes

- Dadas dos listas xs e ys

- Al concatenarlas $zs = xs +\!\!+ ys$ obtenemos:



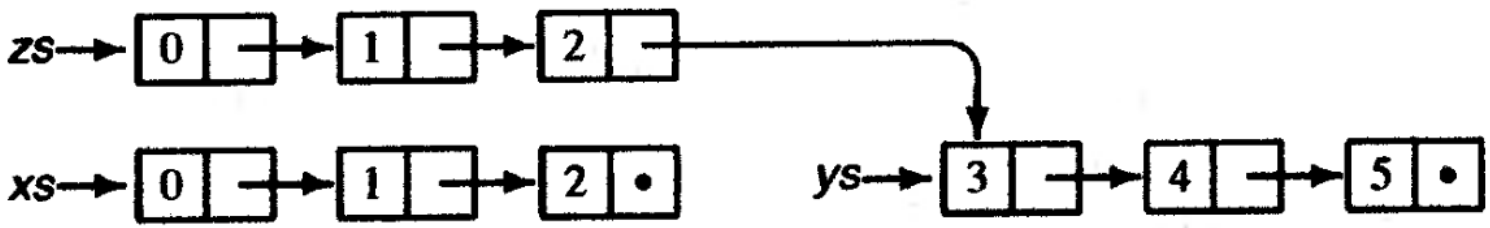
- La operación mantiene las listas xs e ys y crea una nueva zs.
- Se utilizo tiempo para copiar todos los elementos de xs
- La operación $zs = xs +\!\!+ ys$ tomo un tiempo proporcional a la longitud de xs

En Haskell

La concatenación la definimos como:

$$
\begin{aligned}
(+\!\!+) &:: [a] \to [a] \to [a] \\
[\,] +\!\!+ ys &= ys \\
(x : xs) +\!\!+ ys &= x : (xs +\!\!+ ys)
\end{aligned}
$$

Si observamos el resultado de la concatenación $zs = xs +\!\!+ ys$ de las listas persistentes:

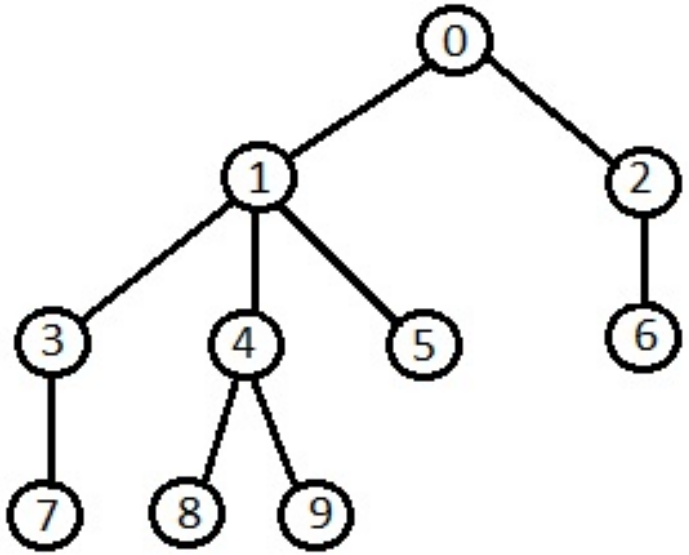


vemos que la copia de los datos de xs en zs coincide con la recursión y la aplicación del $(:)$ *cons*

Estructuras de datos Árboles

Definición (Árboles)

Un árbol es una estructura de dato donde la información almacenada se conecta sin formar ciclos mediante ramas. Esas conexiones cumple la relación padre-hijo. El primer nodo del árbol se lo denomina raíz y a los últimos se los llama hojas. Todos los demás puntos de ramificación se denominarán nodos.



Árboles binarios

Un árbol binario es un árbol en el que cada nodo puede tener como máximo dos hijos.
En Haskell representamos un árbol binario con la siguiente tipo de datos:

$$\textbf{data}\ \text{Bin}\ a = \text{Hoja} \mid \text{Nodo}\ (\text{Bin}\ a)\ a\ (\text{Bin}\ a)$$

Definimos funciones sobre los árboles mediante pattern-matching y recursión:

$$\begin{aligned}
&member :: \text{Eq}\ a \Rightarrow a \rightarrow \text{Bin}\ a \rightarrow \text{Bool} \\
&member\ a\ \text{Hoja} = \text{False} \\
&member\ a\ (\text{Nodo}\ l\ b\ r) = a \equiv b \vee member\ a\ l \vee member\ a\ r
\end{aligned}$$

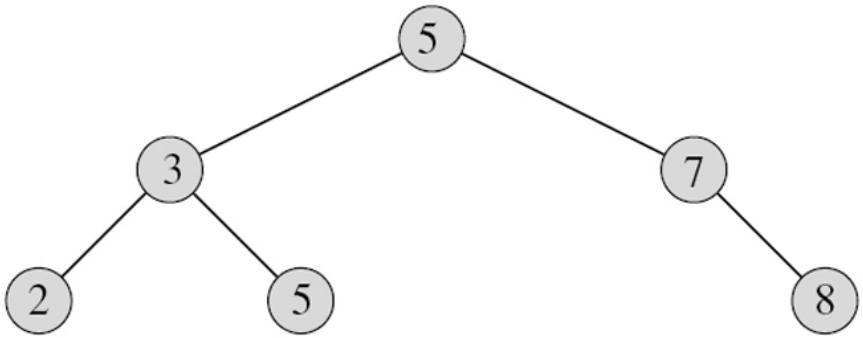
¿Cuantos elementos visita member para encontrar el elemento buscado?

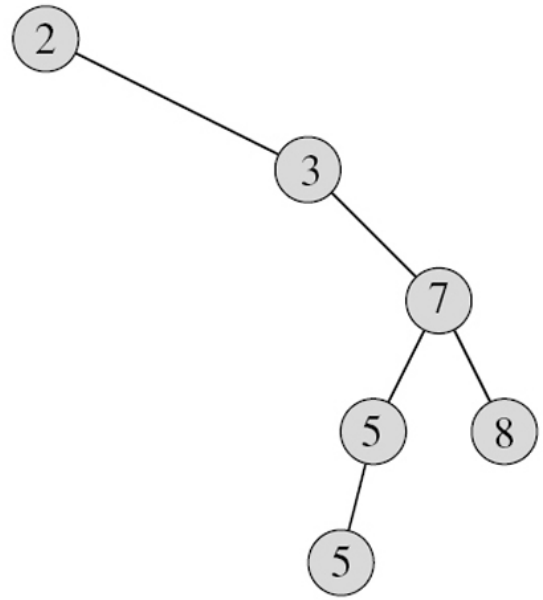
Árboles binarios de Búsqueda

- Los árboles binarios de búsqueda son árboles ordenados.
- El elemento de cada nodo es mayor que los elementos del subárbol izquierdo y menor que los elementos del subárbol derecho.
- Supone definida una relación de orden entre los elementos de sus nodos.
- Puede haber distintos BST para un mismo conjunto de elementos

Árboles binarios de Búsqueda

- Con más precisión, un árbol binario de búsqueda (BST) es un árbol binario t tal que
  - Si t es una Hoja es un BST
  - Si t es un Nodo $l\ a\ r$, tanto l como r tienen que ser BST, y se tiene que cumplir que:
    - Si y es una valor en algún nodo de l entonces $y \leqslant a$
    - Si y es una valor en algún nodo de r entonces $a < y$





Operaciones sobre BSTs

- Recorrido inorder de un BST

$$
\begin{aligned}
&inorder :: \text{Bin } a \to [a] \\
&inorder\ \text{Hoja} &&= [\,] \\
&inorder\ (\text{Nodo } l\ a\ r) &&= inorder\ l +\!\!+ [a] +\!\!+ inorder\ r
\end{aligned}
$$

- Re-implementamos member para BSTs.

$$
\begin{aligned}
&member :: \text{Ord } a \Rightarrow a \to \text{Bin } a \to \text{Bool} \\
&member\ a\ \text{Hoja} = \text{False} \\
&member\ a\ (\text{Nodo } l\ b\ r) \mid a \equiv b = \text{True} \\
&\qquad\qquad\qquad\qquad\quad \mid a < b = member\ a\ l \\
&\qquad\qquad\qquad\qquad\quad \mid otherwise = member\ a\ r
\end{aligned}
$$

¿Cuantos elementos visita member para encontrar el elemento buscado?

Operaciones sobre BSTs

- el mínimo valor en un BST:

$$
\begin{aligned}
&minimum :: \text{Bin } a \to a \\
&minimum\ (\text{Nodo Hoja } a\ r) = a \\
&minimum\ (\text{Nodo } l\ a\ r) \quad\ = minimum\ l
\end{aligned}
$$

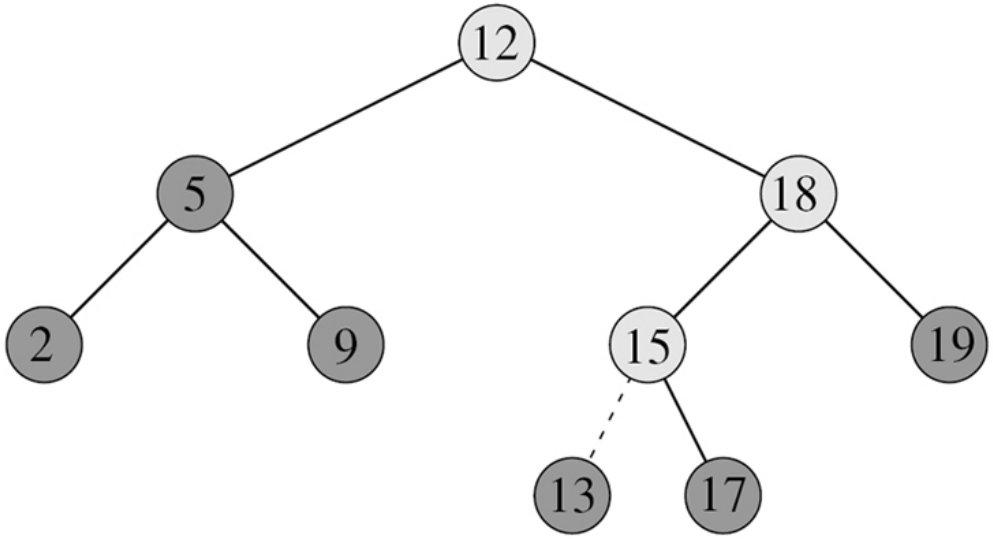
- Ejercicio: implementar maximun.
- Ejercicio: implementar $checkBST :: \text{Bin } a \to \text{Bool}$
- En member, minimun, maximun sólo recorremos (a lo sumo) un camino entre la raíz y una hoja.

Las operaciones $member$, $minimum$ y $maximum$ toman un tiempo en relación a la altura del árbol para ejecutarse.
Sobre árboles más bajos se ejecutan más rápido.

Inserción en BSTs

- Para insertar, recorremos el árbol hasta encontrar una hoja, que transformamos en un nuevo nodo.

$$
\begin{aligned}
&insert :: \text{Ord } a \Rightarrow a \to \text{Bin } a \to \text{Bin } a \\
&insert\ a\ \text{Hoja} &&= \text{Nodo Hoja } a \text{ Hoja} \\
&insert\ a\ (\text{Nodo } l\ b\ r) \mid a \leqslant b &&= \text{Nodo } (insert\ a\ l)\ b\ r \\
&\qquad\qquad\qquad\quad \mid otherwise &&= \text{Nodo } l\ b\ (insert\ a\ r)
\end{aligned}
$$



Borrado en BSTs

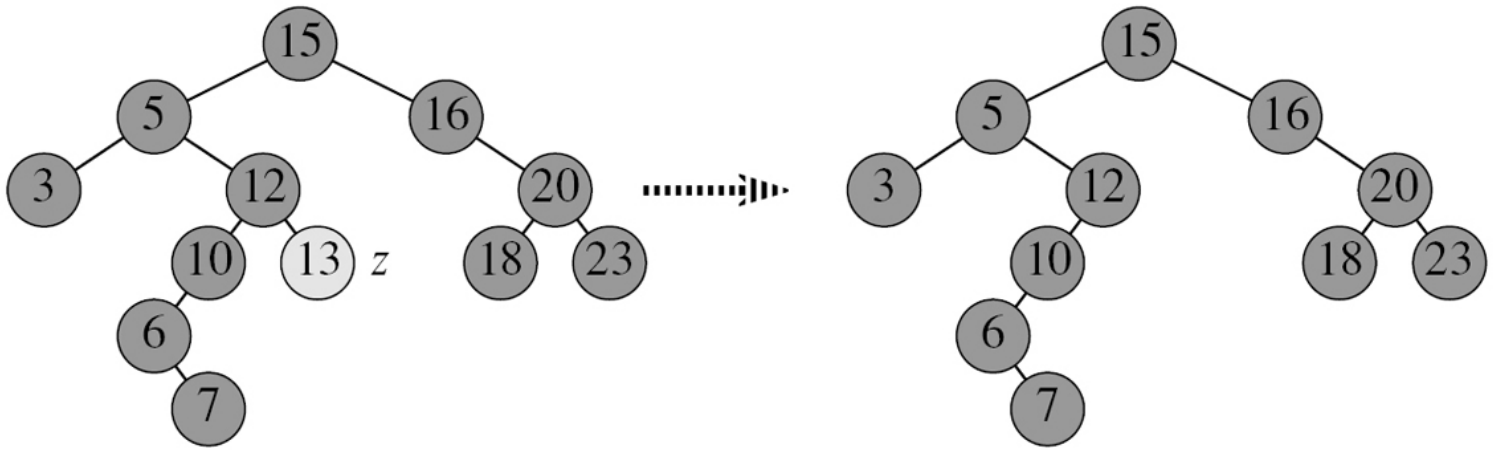
Para borrar un elemento en un BST primero lo tengo que encontrar:

$$
\begin{aligned}
&delete :: \text{Ord } a \Rightarrow a \rightarrow \text{Bin } a \rightarrow \text{Bin } a \\
&delete \ \_ \ \text{Hoja} = \text{Hoja} \\
&delete \ z \ (\text{Nodo } l \ b \ r) \mid z < b = \text{Nodo } (delete \ z \ l) \ b \ r \\
&delete \ z \ (\text{Nodo } l \ b \ r) \mid z > b = \text{Nodo } l \ b \ (delete \ z \ r) \\
&delete \ z \ (\text{Nodo } l \ b \ r) \mid z \equiv b = ...
\end{aligned}
$$

- Una vez encontrado el elemento tenemos que considerar tres casos.

Borrado en BSTs

- El nodo tiene hojas como subárboles

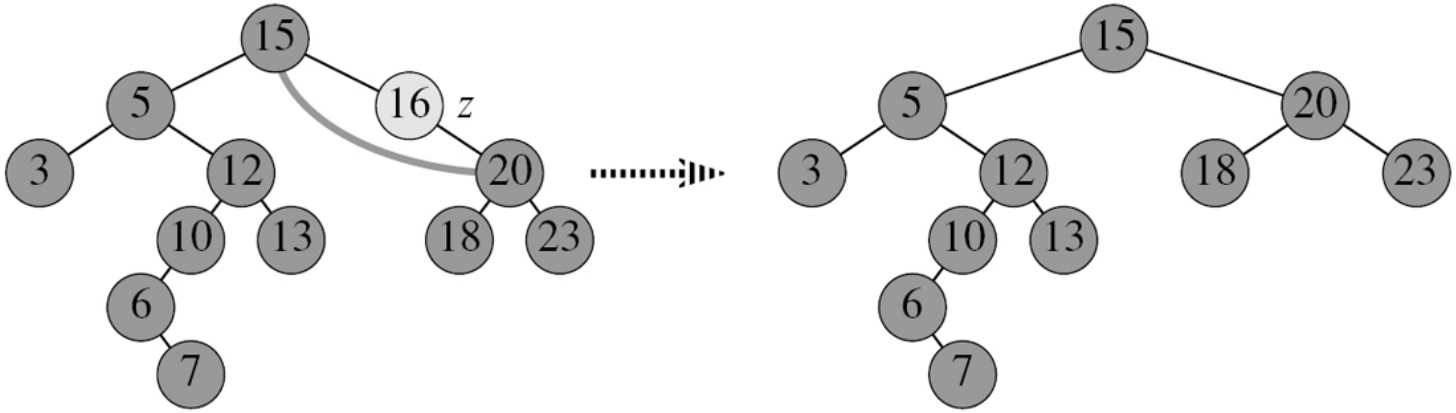


(a)

- $delete\ z$ (Nodo Hoja b Hoja) $\mid z \equiv b =$ Hoja

Borrado en BSTs

- El nodo tiene un sólo subárbol con datos

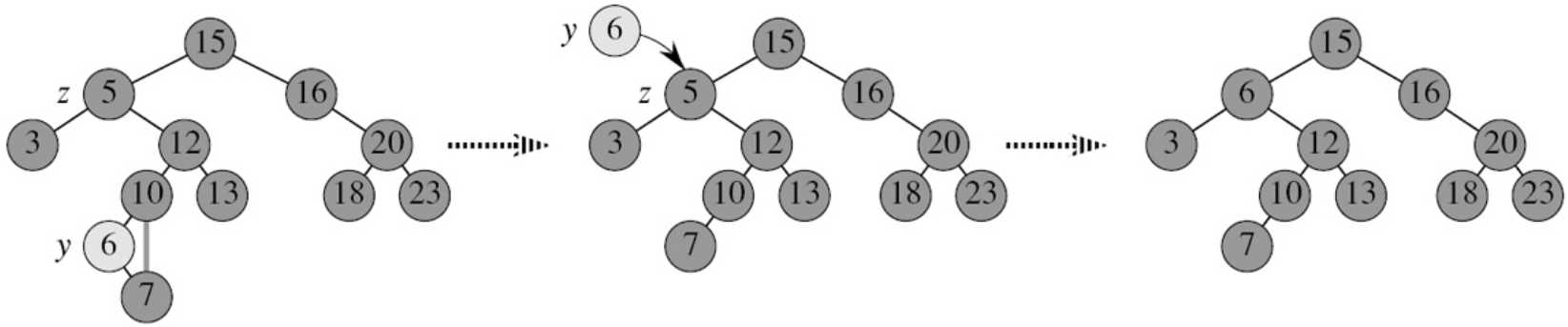


(b)

$$delete\ z\ (\text{Nodo Hoja}\ b\ r) \mid z \equiv b = r$$
$$delete\ z\ (\text{Nodo}\ l\ b\ \text{Hoja}) \mid z \equiv b = l$$

Borrado en BSTs

- El nodo tiene dos subárboles con datos



- $delete\ z\ (\text{Nodo}\ l\ b\ r) \mid z \equiv b =$ **let** $y = minimun\ r$
  in Nodo $l\ y\ (delete\ y\ r)$

Ventajas y Limitaciones

- El tiempo de ejecución de las operaciones depende de la altura del árbol.
- Los BST pueden degenerar en una lista si los datos se insertan en orden.
- Si esto ocurre su rendimiento no será mejor que una lista.
- Existen implementaciones de BST que nos aseguran árboles bajos (balaceados): AVL y Árboles Rojos y Negros.

Resumen

Hoy estudiamos la representación de árboles binarios de busqueda:

- Definimos la estructura BST a partir de una relación de orden.
- Vimos que a un mismo conjunto de datos le pueden corresponder BST diferentes.
- Dimos un recorrido que nos devuelve los datos del árbol en orden.
- Definimos las operaciones básicas $member$, $minimum$, $maximum$, $insert$ y $delete$
- Observamos que estas operaciones son del orden de la altura del árbol.
- Discutimos las ventajas y las limitaciones de la estructura.

Referencias

- Programming in Haskell. Graham Hutton, CUP 2007.
- Introducción a la Programación Funcional con Haskell. Richard Bird, Prentice Hall 1997.
- Purely Functional Data Structures. Chris Okasaki. CUP 1998.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein
- Apuntes de clases "Estructuras persistentes" Mauro Jaskelioff.