

# Estructuras Inmutables

Mauro Jaskelioff

18/4/2022

# Estructuras de Datos Funcionales vs. Imperativas

- ▶ Muchos de los algoritmos tradicionales están pensados para estructuras **efímeras**.
  - ▶ En las estructuras efímeras, los cambios son destructivos.
- ▶ Las estructuras efímeras soportan una sola versión y son coherentes con un modelo secuencial.
- ▶ Las estructuras **inmutables**, soportan varias versiones y son más fácilmente paralelizables.
- ▶ La flexibilidad de las estructuras inmutables tienen un costo:
  - ▶ Debemos adaptar las estructuras y algoritmos al modelo inmutable (de ser posible).
  - ▶ Hay ciertas cotas de las estructuras efímeras que no siempre se van a poder alcanzar.

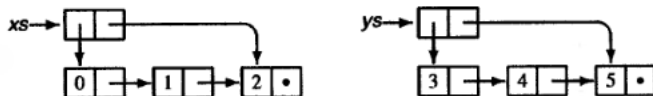
# Inmutabilidad y Sharing

- ▶ En un lenguaje funcional puro, todas las estructuras son inmutables.
- ▶ Las estructuras inmutables no se destruyen al hacer un cambio.
- ▶ Mas bien, se copian los datos y se modifica la copia.
- ▶ Los nodos que no cambian pueden ser compartidos por las diferentes versiones ([sharing](#)).
- ▶ Notar que el manejo automático de la memoria (garbage collection) es prácticamente esencial.

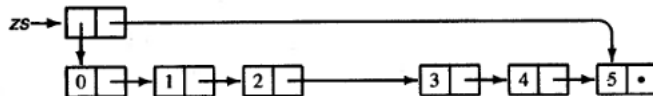
# Ejemplo: Listas simplemente enlazadas efímeras

- Concatenación  $zs = xs \mathbin{++} ys$ .

- Antes



- Después



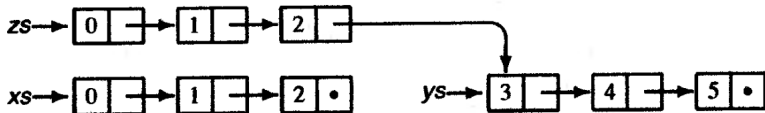
- La operación destruye las listas *xs* e *ys*.
- La operación  $zs = xs \mathbin{++} ys$  es  $O(1)$ .

# Ejemplo: Listas simplemente enlazadas inmutables

- Concatenación  $zs = xs \mathbin{++} ys$ .
- Antes



- Después



- Luego de la concatenación tenemos las tres listas: *xs*, *ys*, y *zs*.
- Hubo que copiar todos los nodos de *xs*.
  - La operación  $zs = xs \mathbin{++} ys$  es  $O(|xs|)$ .

# Listas en Haskell

- ▶ Las listas en Haskell vienen predefinidas, pero bien podríamos definirlas nosotros:

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

- ▶ Preferimos usar la versión predefinida con `[]` para la lista vacía, y `(:)` para la operación cons.
- ▶ La concatenación es

$$\begin{aligned} (++) & \quad :: [a] \rightarrow [a] \rightarrow [a] \\ [] & \quad ++ ys = ys \\ (x : xs) & ++ ys = x : (xs ++ ys) \end{aligned}$$

# Ejercicio

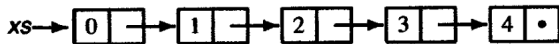
- Considere la siguiente función que modifica un sólo elemento de la lista:

$$\begin{aligned} \text{update} & \quad :: [a] \rightarrow \text{Int} \rightarrow a \rightarrow [a] \\ \text{update } [] & \quad \_ \_ = [] \\ \text{update } (x : xs) \ 0 & \ x' = x' : xs \\ \text{update } (x : xs) \ i & \ x' = x : \text{update } xs \ (i - 1) \ x' \end{aligned}$$

- Dibujar la memoria luego de ejecutar

$$ys = \text{update } xs \ 2 \ 7 \quad \text{y} \quad zs = \text{update } xs \ 0 \ 8$$

donde



# Árboles Binarios en Haskell

- ▶ Un **árbol binario** es un árbol en el que cada nodo tiene exactamente dos hijos.
- ▶ En Haskell representamos un árbol binario con la siguiente definición recursiva:

**data** *Bin a* = *Hoja* | *Nodo* (*Bin a*) *a* (*Bin a*)

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

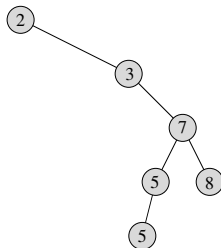
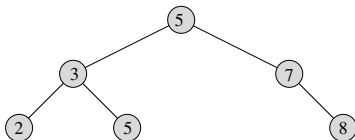
*member* :: *Eq a*  $\Rightarrow$  *a*  $\rightarrow$  *Bin a*  $\rightarrow$  *Bool*  
*member a Hoja* = *False*  
*member a (Nodo l b r)* = (*a* == *b*)  $\vee$  *member a l*  $\vee$  *member a r*

- ▶ ¿Cuál es la complejidad de *member*?



# Árboles Binarios de Búsqueda

- ▶ Un **árbol binario de búsqueda** es un árbol binario  $t$  tal que
  - ▶ o bien  $t$  es una hoja,
  - ▶ o bien  $t$  es un *Nodo*  $l$  a  $r$ , y se cumple que
    - ▶  $l$  y  $r$  son árboles binarios de búsqueda, y
    - ▶ si  $y$  es una clave en algún nodo de  $l$  entonces  $y \leq a$ .
    - ▶ Si  $y$  es una clave en algún nodo de  $r$  entonces  $a < y$ .



# Operaciones sobre BSTs

- Re-implementamos *member* para BSTs.

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

- Recorrido *inorder* en un BST

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

# Operaciones sobre BSTs

- ▶ El mínimo valor en un BST:

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

- ▶ Ejercicio: implementar *maximum*.
- ▶ Ejercicio: implementar *checkBST*  $:: \text{Bin } a \rightarrow \text{Bool}$ .
- ▶ En *member*, *minimum* y *maximum* sólo recorreremos (a lo sumo) un camino entre la raíz y una hoja.

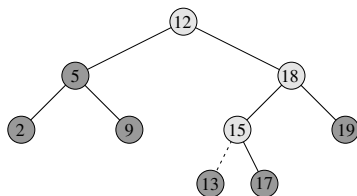
## Teorema

Las operaciones *member*, *minimum* y *maximum* son  $O(h)$ , donde  $h$  es la altura del árbol.

# Inserción en BSTs

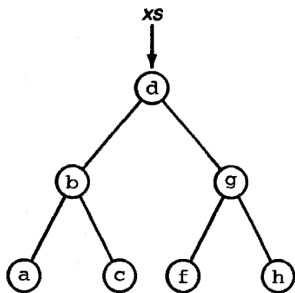
- Para insertar, recorremos el árbol hasta encontrar una hoja, que transformamos en un nuevo nodo.

$insert :: Ord\ a \Rightarrow a \rightarrow Bin\ a \rightarrow Bin\ a$   
 $insert\ a\ Hoja = Nodo\ Hoja\ a\ Hoja$   
 $insert\ a\ (Nodo\ l\ b\ r) \mid a \leq b = Nodo\ (insert\ a\ l)\ b\ r$   
 $\mid otherwise = Nodo\ l\ b\ (insert\ a\ r)$

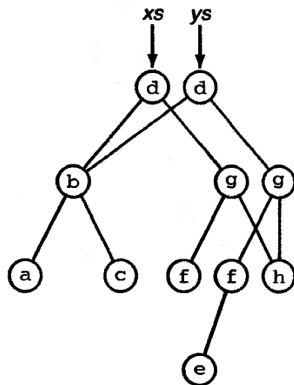


# Sharing en BSTs

- Veamos qué sucede en memoria al insertar un nodo a un BST.



$ys = \text{insert } "e" \text{ } xs$



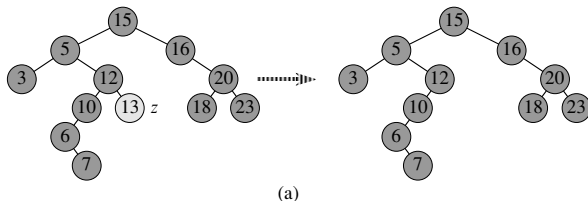
# Borrado en BSTs

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

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

# Borrado en BSTs

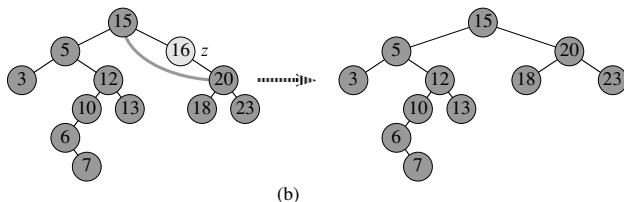
a) El nodo tiene hojas como subárboles



*delete z (Nodo Hoja b Hoja) | z == b = Hoja*

# Borrado en BSTs

b) El nodo tiene un sólo subárbol con datos



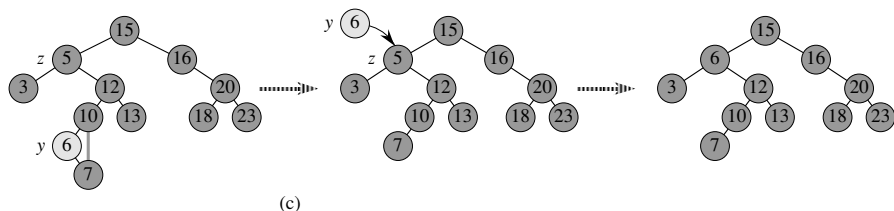
*delete z (Nodo Hoja b r) | z == b = r*

*delete z (Nodo l b Hoja) | z == b = l*



# Borrado en BSTs

c) El nodo tiene dos subárboles con datos



*delete*  $z$  (Nodo  $l$   $b$   $r$ ) |  $z == b =$  **let**  $y = \text{minimum } r$   
**in** *Nodo*  $l$   $y$  (*delete*  $y$   $r$ )

# Árboles balanceados

- ▶ Las operaciones de búsqueda, inserción y borrado son del orden de la altura del árbol.
- ▶ En el mejor caso son  $O(\lg n)$ ,
- ▶ pero en el peor caso pueden ser  $O(n)$ 
  - ▶ Por ejemplo, al insertar datos ordenados, el árbol degenera en una lista.
- ▶ La solución es mantener el árbol balanceado
  - ▶ Ejemplos: AVL, Red-Black Trees.

# Red-Black Trees

- ▶ Es un árbol binario de búsqueda con nodo “coloreados” rojos o negros,

**data** *Color* = *R* | *B*

**data** *RBT* *a* = *E* | *T Color (RBT a) a (RBT a)*

- ▶ y además se cumplen las siguiente invariantes:
  - INV1* Ningún nodo rojo tiene hijos rojos.
  - INV2* Todos los caminos de la raíz a una hoja tienen el mismo número de nodos negros (altura negra).
- ▶ En un RBT, el camino más largo es a lo sumo el *doble* que el camino más corto.
- ▶ En un RBT la altura es  $O(\lg n)$  (o sea, es un árbol balanceado).

# Operaciones sobre RBTs

- Implementamos *member* para RBTs.

$$\begin{array}{ll} \text{member}_{RBT} & :: \text{Ord } a \Rightarrow a \rightarrow RBT\ a \rightarrow Bool \\ \text{member}_{RBT}\ a\ E & = False \\ \text{member}_{RBT}\ a\ (T\ -\ l\ b\ r) \mid & \begin{array}{l} a == b = True \\ a < b = \text{member}_{RBT}\ a\ l \\ a > b = \text{member}_{RBT}\ a\ r \end{array} \end{array}$$

- El código es el mismo que para BSTs
  - Simplemente ignoramos el color.

# Inserción en RBTs

$insert \quad :: \text{Ord } a \Rightarrow a \rightarrow \text{RBT } a \rightarrow \text{RBT } a$

$insert\ x\ t = makeBlack\ (ins\ x\ t)$

**where**  $ins\ x\ E \quad \quad \quad = T\ \textcolor{red}{R}\ E\ x\ E$

$ins\ x\ (T\ c\ l\ y\ r) \mid x < y \quad = balance\ c\ (ins\ x\ l)\ y\ r$

$\mid x > y \quad = balance\ c\ l\ y\ (ins\ x\ r)$

$\mid otherwise = T\ c\ l\ y\ r$

$makeBlack\ E \quad \quad \quad = E$

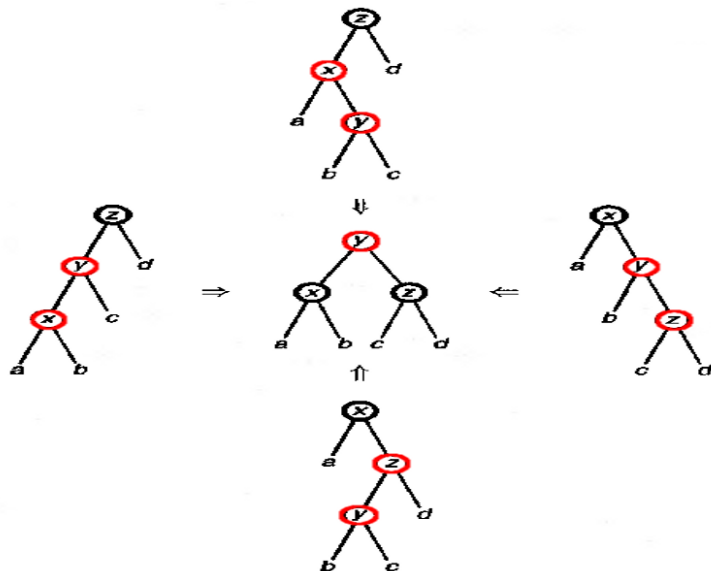
$makeBlack\ (T\ _\ l\ x\ r) = T\ \textcolor{black}{B}\ l\ x\ r$

- Notar que el nodo nuevo se inserta como un nodo rojo, por lo que se mantiene la altura negra (INV2).
- Pero se puede violar INV1, por lo que hay que rebalancear.
- Luego de rebalanceo puede quedar una raíz roja, por lo que se colorea de negro la raíz.

# Rebalanceo de RBTs

- ▶ Luego de una insertar el nuevo nodo rojo hay a lo sumo **una única** violación de **INV1** que ocurre cuando el padre es rojo.
- ▶ Por lo tanto la violación siempre ocurre en un camino **B-R-R**.
- ▶ La función *balance* va arreglando y propagando hacia arriba esta violación.
- ▶ La (única) violación, puede aparecer en cuatro configuraciones.
- ▶ En todos los casos la solución es la misma: reescribir el nodo como un padre rojo con dos hijos negros.

# Configuraciones de violación de invariante en RBTs



# Implementación de *balance*

- ▶ La implementación de *balance* se puede hacer fácilmente mediante pattern-matching.

$balance :: Color \rightarrow RBT\ a \rightarrow a \rightarrow RBT\ a \rightarrow RBT\ a$

$balance\ \mathbf{B}\ (T\ \mathbf{R}\ (T\ \mathbf{R}\ a\ x\ b)\ y\ c)\ z\ d = T\ \mathbf{R}\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d)$

$balance\ \mathbf{B}\ (T\ \mathbf{R}\ a\ x\ (T\ \mathbf{R}\ b\ y\ c))\ z\ d = T\ \mathbf{R}\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d)$

$balance\ \mathbf{B}\ a\ x\ (T\ \mathbf{R}\ (T\ \mathbf{R}\ b\ y\ c)\ z\ d) = T\ \mathbf{R}\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d)$

$balance\ \mathbf{B}\ a\ x\ (T\ \mathbf{R}\ b\ y\ (T\ \mathbf{R}\ c\ z\ d)) = T\ \mathbf{R}\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d)$

$balance\ c\ l\ a\ r = T\ c\ l\ a\ r$

- ▶  $W_{balance} \in O(1)$ . Como el árbol está balanceado  $W_{insert} \in O(\lg n)$ .
- ▶ La implementación es simple.
  - ▶ Comparar con las implementaciones imperativas.
  - ▶ De yapa, esta implementación es inmutable.



# Heaps

- ▶ Los **heaps** (o montículos) son árboles que permiten un acceso eficiente al mínimo elemento.
- ▶ Mantienen la invariante de que todo nodo es menor a todos los valores de sus hijos.
- ▶ Por lo tanto, el mínimo está siempre en la raíz.
- ▶ Hay diferentes variantes de heaps:
  - ▶ Heaps tradicionales, Leftist, Binomial, Splay, y Pairing Heaps.
- ▶ Un heap debe soportar eficientemente las operaciones

*insert*        ::  $Ord\ a \Rightarrow a \rightarrow Heap\ a \rightarrow Heap\ a$

*findMin*     ::  $Ord\ a \Rightarrow Heap\ a \rightarrow a$

*deleteMin* ::  $Ord\ a \Rightarrow Heap\ a \rightarrow Heap\ a$

- ▶ Algunas variantes también soportan eficientemente la unión de dos heaps: *merge* ::  $Ord\ a \Rightarrow Heap\ a \rightarrow Heap\ a \rightarrow Heap\ a$ .

# Leftist Heaps

- ▶ Variante de heap que es fácil de implementar en forma inmutable.
- ▶ El **rango** de un heap es la longitud de la **espina derecha** (el camino hacia la derecha hasta un nodo vacío.)
- ▶ Invariante Leftist: el **rango** de cualquier hijo izquierdo es mayor o igual que el de su hermano de la derecha.
- ▶ Consecuencias:
  - ▶ La espina derecha es el camino más corto a un nodo vacío.
  - ▶ La longitud de la espina derecha es a lo sumo  $\lg(n + 1)$ .
  - ▶ Los elementos de la espina derecha están ordenados (como consecuencia de la invariante de heap.)

# Implementación de leftist heaps

- Definimos el siguiente tipo de datos

```
type Rank    = Int
data Heap a = E | N Rank a (Heap a) (Heap a)
```

- La operación más importante es *merge*:

```
merge :: Ord a => Heap a -> Heap a -> Heap a
merge h1 E = h1
merge E h2 = h2
merge h1@(N - x a1 b1) h2@(N - y a2 b2) =
    if x ≤ y then makeH x a1 (merge b1 h2)
    else makeH y a2 (merge h1 b2)
```

- Las espigas derechas se mezclan para seguir ordenadas y preservar la invariante leftist.
- La función *makeH* se encarga de preservarla.

## Implementación de leftist heaps (cont.)

- Definimos la función que devuelve el rango

$$\begin{aligned} \text{rank} &:: \text{Heap } a \rightarrow \text{Rank} \\ \text{rank } E &= 0 \\ \text{rank } (N \ r \ \_ \ \_) &= r \end{aligned}$$

- Definimos  $\text{makeH}$

$$\text{makeH } x \ a \ b = \text{if } \text{rank } a \geq \text{rank } b \text{ then } N (\text{rank } b + 1) \ x \ a \ b \\ \text{else } N (\text{rank } a + 1) \ x \ b \ a$$

- Tanto  $W_{\text{rank}}$  como  $W_{\text{makeH}}$  están en  $O(1)$ .
- Como la espina derecha es a lo sumo logarítmica,

$$W_{\text{merge}} \in O(\lg n).$$

# Implementación de leftist heaps (cont.)

- Una vez definido un *merge* eficiente, el resto de las operaciones son sencillas

<i>insert</i>	$:: Ord\ a \Rightarrow a \rightarrow Heap\ a \rightarrow Heap\ a$
<i>insert</i> $x\ h$	$= merge\ (N\ 1\ x\ E\ E)\ h$
<i>findMin</i>	$:: Ord\ a \Rightarrow Heap\ a \rightarrow a$
<i>findMin</i> $(N\ \_ x\ a\ b)$	$= x$
<i>deleteMin</i>	$:: Ord\ a \Rightarrow Heap\ a \rightarrow Heap\ a$
<i>deleteMin</i> $(N\ \_ x\ a\ b)$	$= merge\ a\ b$

- Dado que  $W_{merge} \in O(\lg n)$ , tenemos que  $W_{insert}$  y  $W_{deleteMin}$  están en  $O(\lg n)$ .
- $W_{findMin} \in O(1)$ .

- ▶ *Programming in Haskell*. Graham Hutton, CUP 1ed (2007), o 2da ed (2016).
- ▶ *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, Clifford Stein