Estructuras Inmutables

Mauro Jaskelioff

13/04/2018

Estructuras de Datos Funcionales vs. Imperativas

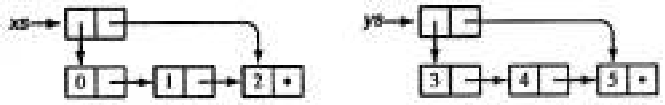
- Muchos de los algoritmos tradicionales están pensados para estructuras efímeras.
  - En las estructuras efímeras, los cambios son destructivos.
- Las estructuras efímeras soportan una sola versión y son coherentes con un modelo secuencial.
- Las estructuras inmutables, soportan varias versiones y son 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

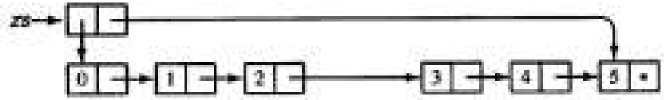
- En un lenguaje funcional puro, todas las estructuras son inmutables.
- Las estructuras inmutables no se destruyen al hacer un cambio.
- Mas bien, se copian los datos y se modifica la copia.
- Los nodos que no cambian pueden ser compartidos por las diferentes versiones (sharing).
- Notar que el manejo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

Ejemplo: Listas simplemente enlazadas efímeras

- Concatenación $zs = xs +\!\!+\ ys$.
- Antes



- Después

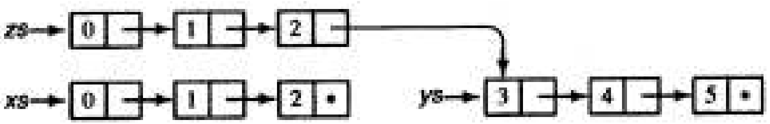


- La operación destruye las listas *xs* e *ys*.
- La operación $zs = xs +\!\!+\ ys$ es $O(1)$.

Ejemplo: Listas simplemente enlazadas inmutables

- Concatenación $zs = xs +\!\!+\ ys$.
- Antes

- Después



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

Listas en Haskell

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

$$\begin{aligned} \textbf{data}\ \mathit{List}\ a = {} & \mathit{Nil} \\ \mid {} & \mathit{Cons}\ a\ (\mathit{List}\ a) \end{aligned}$$

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

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

Ejercicio

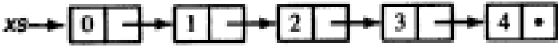
- Considere la siguiente función que modifica un sólo elemento de la lista:

$$
\begin{array}{lcl}
update & & :: [a] \to Int \to a \to [a] \\
update\ [\,] & \_\ \_ & = [\,] \\
update\ (x : xs)\ 0\ x' & & = x' : xs \\
update\ (x : xs)\ i\ \ x' & & = x\ \ : update\ xs\ (i - 1)\ x'
\end{array}
$$

- Dibujar la memoria luego de ejecutar

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

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:

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

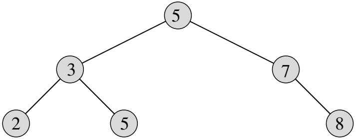
- Definimos funciones sobre los árboles mediante pattern-matching y recursión:

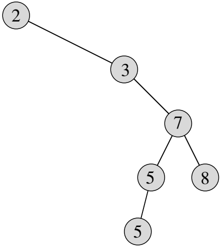
$$\begin{aligned}
&member && :: Eq\ a \Rightarrow a \rightarrow Bin\ a \rightarrow Bool \\
&member\ a\ Hoja && = False \\
&member\ a\ (Nodo\ l\ b\ r) && = (a == b) \vee member\ a\ l \vee member\ a\ r
\end{aligned}$$

- ¿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 $y \leqslant a$.
    - Si y es una clave en algún nodo de r entonces $a < y$.





Operaciones sobre BSTs

- Re-implementamos *member* para BSTs.

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

- Recorrido inorder en un BST

$$
\begin{array}{ll}
inorder & :: Bin\ a \rightarrow [a] \\
inorder\ Hoja & = [\,] \\
inorder\ (Nodo\ l\ a\ r) & = inorder\ l +\!\!+ [a] +\!\!+ inorder\ r
\end{array}
$$

Operaciones sobre BSTs

- El mínimo valor en un BST:

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

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

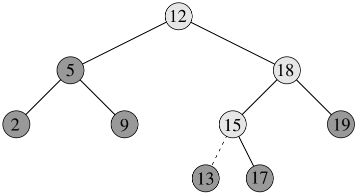
Teorema

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

Inserción en BSTs

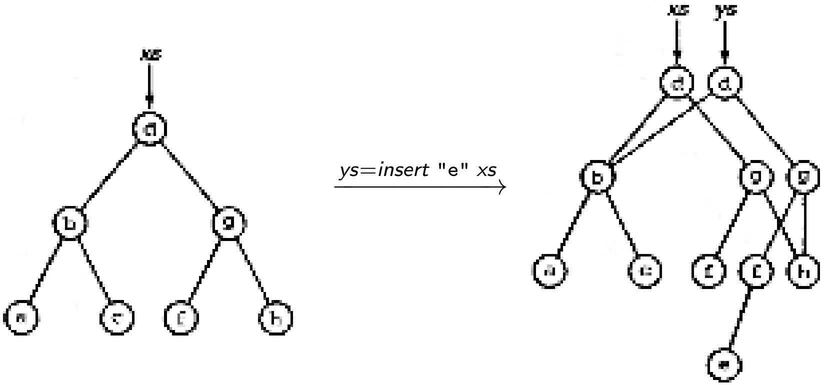
- Para insertar, recorremos el árbol hasta encontrar una hoja, que transformamos en un nuevo nodo.

$$
\begin{array}{lll}
\mathit{insert} & :: & \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Bin}\ a \rightarrow \mathit{Bin}\ a \\
\mathit{insert}\ a\ \mathit{Hoja} & = & \mathit{Nodo}\ \mathit{Hoja}\ a\ \mathit{Hoja} \\
\mathit{insert}\ a\ (\mathit{Nodo}\ l\ b\ r) \mid a \leqslant b & = & \mathit{Nodo}\ (\mathit{insert}\ a\ l)\ b\ r \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} & = & \mathit{Nodo}\ l\ b\ (\mathit{insert}\ a\ r)
\end{array}
$$



Sharing en BSTs

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



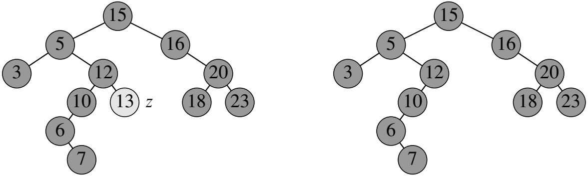
Borrado en BSTs

$$
\begin{array}{lcl}
\mathit{delete} & :: & \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Bin}\ a \rightarrow \mathit{Bin}\ a \\
\mathit{delete}\ \_\ \mathit{Hoja} & = & \mathit{Hoja} \\
\mathit{delete}\ z\ (\mathit{Nodo}\ l\ b\ r) \mid z < b & = & \mathit{Nodo}\ (\mathit{delete}\ z\ l)\ b\ r \\
\mathit{delete}\ z\ (\mathit{Nodo}\ l\ b\ r) \mid z > b & = & \mathit{Nodo}\ l\ b\ (\mathit{delete}\ z\ r) \\
\mathit{delete}\ z\ (\mathit{Nodo}\ l\ b\ r) \mid z == b & = & \ldots
\end{array}
$$

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

Borrado en BSTs

a) El nodo tiene hojas como subárboles

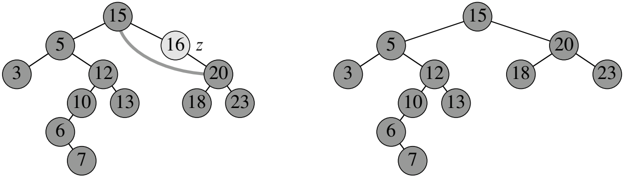


(a)

$$\textit{delete } z \ (\textit{Nodo Hoja } b \textit{ Hoja}) \mid z == b = \textit{Hoja}$$

Borrado en BSTs

b) El nodo tiene un sólo subárbol con datos

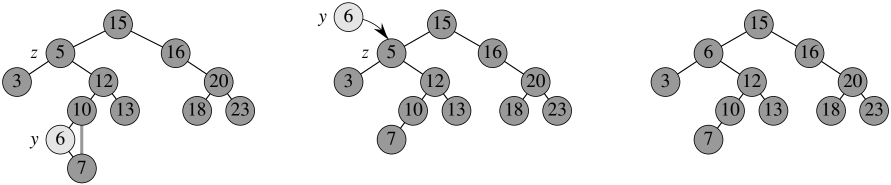


(b)

$$delete\ z\ (Nodo\ Hoja\ b\ r) \mid z == b = r$$
$$delete\ z\ (Nodo\ l\ b\ Hoja) \mid z == b = l$$

Borrado en BSTs

c) El nodo tiene dos subárboles con datos



(c)

$$
\begin{aligned}
&\mathit{delete}\ z\ (\mathit{Nodo}\ l\ b\ r) \mid z == b = \textbf{let}\ y = \mathit{minimum}\ r \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{Nodo}\ l\ y\ (\mathit{delete}\ y\ r)
\end{aligned}
$$

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,

$$\begin{aligned}
&\textbf{data}\ \mathit{Color} &&= R \mid \mathbf{B} \\
&\textbf{data}\ \mathit{RBT}\ a &&= E \mid T\ \mathit{Color}\ (\mathit{RBT}\ a)\ a\ (\mathit{RBT}\ a)
\end{aligned}$$

- 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.
- Esto significa que la altura está siempre en $O(\lg n)$.

Operaciones sobre RBTs

- Implementamos *member* para RBTs.

$$
\begin{aligned}
&member_{RBT} && :: Ord\ a \Rightarrow a \rightarrow RBT\ a \rightarrow Bool \\
&member_{RBT}\ a\ E && = False \\
&member_{RBT}\ a\ (T\ \_\ l\ b\ r) \mid a == b && = True \\
&\qquad\qquad\qquad\qquad\quad \mid a < b && = member_{RBT}\ a\ l \\
&\qquad\qquad\qquad\qquad\quad \mid a > b && = member_{RBT}\ a\ r
\end{aligned}
$$

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

Inserción en RBTs

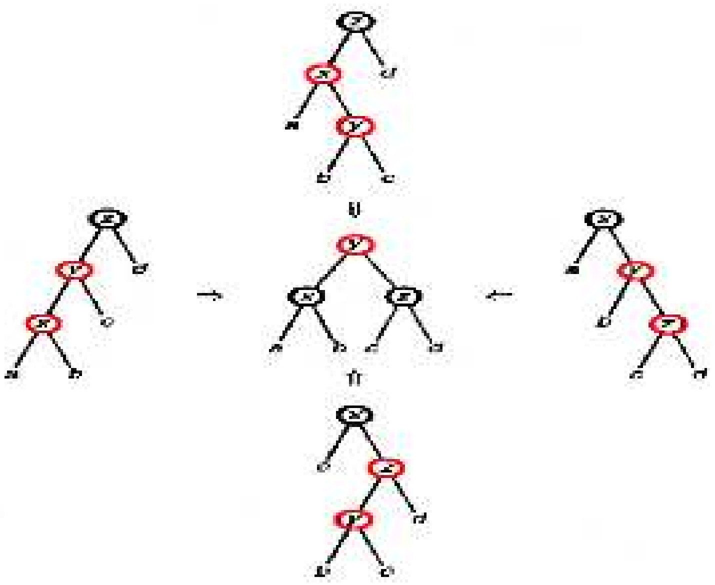
$$
\begin{array}{l}
insert \quad :: Ord\ a \Rightarrow a \rightarrow RBT\ a \rightarrow RBT\ a \\
insert\ x\ t = makeBlack\ (ins\ x\ t) \\
\quad \mathbf{where}\ ins\ x\ E \qquad\qquad\qquad\qquad = T\ \color{red}{R}\ E\ x\ E \\
\qquad\qquad ins\ x\ (T\ c\ l\ y\ r)\ |\ x < y \qquad = balance\ c\ (ins\ x\ l)\ y\ r \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ x > y \qquad = balance\ c\ l\ y\ (ins\ x\ r) \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ otherwise = T\ c\ l\ y\ r \\
\qquad\qquad makeBlack\ E \qquad\qquad = E \\
\qquad\qquad makeBlack\ (T\ \_\ l\ x\ r) = T\ \mathbf{B}\ l\ x\ r
\end{array}
$$

- 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.

$$
\begin{array}{ll}
balance :: Color \rightarrow RBT\ a \rightarrow a \rightarrow RBT\ a \rightarrow RBT\ a & \\
balance\ \mathbf{B}\ (T\ R\ (T\ R\ a\ x\ b)\ y\ c)\ z\ d & = T\ R\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d) \\
balance\ \mathbf{B}\ (T\ R\ a\ x\ (T\ R\ b\ y\ c))\ z\ d & = T\ R\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d) \\
balance\ \mathbf{B}\ a\ x\ (T\ R\ (T\ R\ b\ y\ c)\ z\ d) & = T\ R\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d) \\
balance\ \mathbf{B}\ a\ x\ (T\ R\ b\ y\ (T\ R\ c\ z\ d)) & = T\ R\ (T\ \mathbf{B}\ a\ x\ b)\ y\ (T\ \mathbf{B}\ c\ z\ d) \\
balance\ c\ l\ a\ r & = T\ c\ l\ a\ r
\end{array}
$$

- $W_{balance} \in O(1)$. 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

$$
\begin{aligned}
&\mathit{insert} &&:: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Heap}\ a \rightarrow \mathit{Heap}\ a\\
&\mathit{findMin} &&:: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \rightarrow a\\
&\mathit{deleteMin} &&:: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \rightarrow \mathit{Heap}\ a
\end{aligned}
$$

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

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

$$\begin{array}{l} \textbf{type}\ \mathit{Rank} \quad = \mathit{Int} \\ \textbf{data}\ \mathit{Heap}\ a = E \mid N\ \mathit{Rank}\ a\ (\mathit{Heap}\ a)\ (\mathit{Heap}\ a) \end{array}$$

- La operación más importante es *merge*:

$$\begin{array}{l} \mathit{merge} :: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \to \mathit{Heap}\ a \to \mathit{Heap}\ a \\ \mathit{merge}\ h1\ E = h1 \\ \mathit{merge}\ E\ h2 = h2 \\ \mathit{merge}\ h1@(N\ \_\ x\ a1\ b1)\ h2@(N\ \_\ y\ a2\ b2) = \\ \qquad\qquad \textbf{if}\ x \leqslant y\ \textbf{then}\ \mathit{makeH}\ x\ a1\ (\mathit{merge}\ b1\ h2) \\ \qquad\qquad\qquad\qquad \textbf{else}\ \ \mathit{makeH}\ y\ a2\ (\mathit{merge}\ h1\ b2) \end{array}$$

- Las espinas 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

$$\begin{array}{lll} rank & :: & Heap\ a \to Rank \\ rank\ E = & & 0 \\ rank\ (N\ r\ \_\ \_\ \_) = r & & \end{array}$$

- Definimos *makeH*

$$\begin{array}{rl} makeH\ x\ a\ b = \textbf{if}\ rank\ a \geqslant rank\ b & \textbf{then}\ N\ (rank\ b + 1)\ x\ a\ b \\ & \textbf{else}\ \ N\ (rank\ a + 1)\ x\ b\ a \end{array}$$

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

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

Implementación de leftist heaps (cont.)

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

$$
\begin{array}{ll}
\mathit{insert} & :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Heap}\ a \rightarrow \mathit{Heap}\ a \\
\mathit{insert}\ x\ h & = \mathit{merge}\ (N\ 1\ x\ E\ E)\ h \\
\mathit{findMin} & :: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \rightarrow a \\
\mathit{findMin}\ (N\ \_\ x\ a\ b) & = x \\
\mathit{deleteMin} & :: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \rightarrow \mathit{Heap}\ a \\
\mathit{deleteMin}\ (N\ \_\ x\ a\ b) & = \mathit{merge}\ a\ b
\end{array}
$$

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

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, Clifford Stein