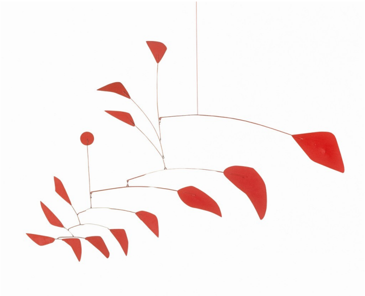
Programación funcional con Haskell
Árboles Balanceados

Juan Manuel Rabasedas



Árboles Balanceados

En los arboles binarios de búsqueda:

- Las operaciones de búsqueda, inserción y borrado son del orden de la altura del árbol.
- Si insertamos datos en orden, la altura del árbol es máxima y los rendimientos son los de una lista.
- Si mantenemos la altura lo más baja posible nuestros algoritmos son más rápidos
- Para lograr esto debemos mantener nuestros árboles balanceados.
- La altura del árbol debe estar en el orden del log n.
- Red Balck Tree, Leftist Heaps, Maxiphobic Heaps, ...

Red Black Tree

Es un BST cuyos nodos están coloreados de Rojo o de **Negro**:

> **data** Color $\;= \mathbf{R} \mid \mathbf{B}$
> **data** RBT $a = \mathrm{E} \mid \mathrm{T}$ Color (RBT a) a (RBT a)

y se cumplen los siguientes invariantes sobre los colores:

1. Todos los nodos rojos tienen un padre negro. (Local)
2. Todos los caminos de la raíz a una hoja tienen el mismo número de nodos negros (altura negra). (Global)

Esto significa que la altura siempre esta en el orden del log n.

Red Black Tree

Implementamos *member* para RBTs.

$$
\begin{array}{l}
\mathit{memberRBT} :: \mathrm{Ord}\ a \Rightarrow a \rightarrow \mathrm{RBT}\ a \rightarrow \mathrm{Bool} \\
\mathit{memberRBT}\ a\ \mathrm{E} = \mathrm{False} \\
\mathit{memberRBT}\ a\ (\mathrm{T}\ \_\ l\ b\ r) \mid a \equiv b \quad = \mathrm{True} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid a < b \quad = \mathit{memberRBT}\ a\ l \\
\qquad\qquad\qquad\qquad\qquad\quad \mid j\ a > b = \mathit{memberRBT}\ a\ r
\end{array}
$$

Si ignoramos el color el código es el mismo que para BSTs

Red Black Tree

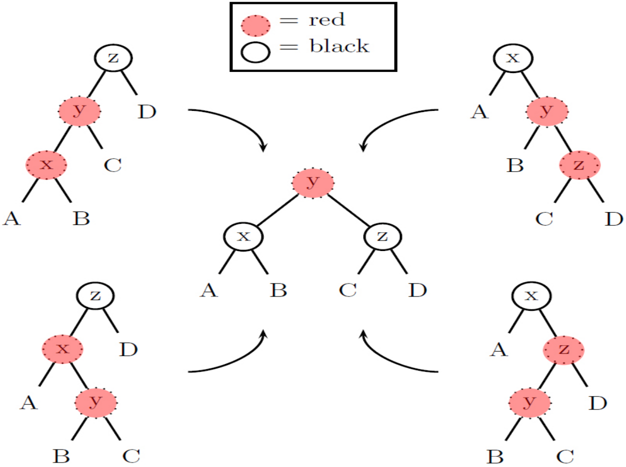
Implementamos *insert* para RBTs:

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

- Para mantener la (altura negra) el nuevo nodo se inserta rojo. (Global)
- Pero esto puede romper la Propiedad 1,para solucionar esto vamos a rebalancear. (Local)
- El rebalanceo puede dejarnos una raíz roja, así que la coloremos de negro.

Red Black Tree

Luego de insertar el nuevo nodo rojo hay a lo sumo una única violación de la propiedad 1, que ocurre cuando el padre es rojo.
Esta violación, puede aparecer en cuatro configuraciones:



Red Black Tree

Implementamos *balance* para RBTs:

$$
\begin{aligned}
&\mathit{balance} :: \text{Color} \to \text{RBT}\ a \to a \to \text{RBT}\ a \to \text{RBT}\ a\\
&\mathit{balance}\ \mathbf{B}\ (\text{T}\ \mathbf{R}\ (\text{T}\ \mathbf{R}\ a\ x\ b)\ y\ c)\ z\ d = \text{T}\ \mathbf{R}\ (\text{T}\ \mathbf{B}\ a\ x\ b)\ y\ (\text{T}\ \mathbf{B}\ c\ z\ d)\\
&\mathit{balance}\ \mathbf{B}\ (\text{T}\ \mathbf{R}\ a\ x\ (\text{T}\ \mathbf{R}\ b\ y\ c))\ z\ d = \text{T}\ \mathbf{R}\ (\text{T}\ \mathbf{B}\ a\ x\ b)\ y\ (\text{T}\ \mathbf{B}\ c\ z\ d)\\
&\mathit{balance}\ \mathbf{B}\ a\ x\ (\text{T}\ \mathbf{R}\ (\text{T}\ \mathbf{R}\ b\ y\ c)\ z\ d) = \text{T}\ \mathbf{R}\ (\text{T}\ \mathbf{B}\ a\ x\ b)\ y\ (\text{T}\ \mathbf{B}\ c\ z\ d)\\
&\mathit{balance}\ \mathbf{B}\ a\ x\ (\text{T}\ \mathbf{R}\ b\ y\ (\text{T}\ \mathbf{R}\ c\ z\ d)) = \text{T}\ \mathbf{R}\ (\text{T}\ \mathbf{B}\ a\ x\ b)\ y\ (\text{T}\ \mathbf{B}\ c\ z\ d)\\
&\mathit{balance}\ c\ l\ a\ r = \text{T}\ c\ l\ a\ r
\end{aligned}
$$

- Es una implementación no recursiva.
- El costo de *insert* es logarítmico.

Haps

Los heaps (o montículos) son árboles que permiten un acceso eficiente al mínimo elemento:

- Invariante Heap: todo nodo es menor a todos los valores de sus hijos.
- El mínimo está siempre en la raíz.
- Las operaciones de un heap son:

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

Leftist Heaps

Los Leftist Heaps son estructuras que cumplen con el invariante Heap y a demás:

- Implementa eficientemente la unión de dos heaps:

$$\mathit{merge} :: \mathrm{Ord}\ a \Rightarrow \mathrm{Heap}\ a \rightarrow \mathrm{Heap}\ a \rightarrow \mathrm{Heap}\ a$$

- Invariante Leftist: el rango de cualquier hijo izquierdo es mayor o igual que el de su hermano de la derecha.
- Rango es la longitud de la espina derecha de cada nodo.

Luego tenemos que:

- La espina derecha es la ruta más corta a una hoja.
- Los elementos de la espina derecha están ordenados.

Leftist Heaps

Definimos el siguiente tipo de datos:

$$\textbf{type}\ \text{Rank} = \text{Int}$$
$$\textbf{data}\ \text{Heap}\ a = \text{E} \mid \text{N Rank}\ a\ (\text{Heap}\ a)\ (\text{Heap}\ a)$$

Definimos la función *merge*:

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

Decidido quien es la nueva raíz, debemos determinar los dos sub-árboles que la acompañen:

- Las espinas derechas (b) se mezclan (*merge*) para seguir ordenadas y
- preservar la invariante leftist. Para preservarla definimos *makeH*.

Leftist Heaps

Definimos la función que devuelve el rango

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

Definimos *makeH*:

$$
\begin{aligned}
\textit{makeH}\ x\ a\ b = \mathbf{if}\ \textit{rank}\ a \geqslant \textit{rank}\ b\ &\mathbf{then}\ \mathsf{N}\ (\textit{rank}\ b + 1)\ x\ a\ b\\
&\mathbf{else}\ \ \mathsf{N}\ (\textit{rank}\ a + 1)\ x\ b\ a
\end{aligned}
$$

- Tanto *rank* como *makeH* no son recursivas.
- Como la espina derecha es logarítmica, *merge* es logarítmica.

Leftist Heaps

Una vez definido un *merge* eficiente, el resto de las operaciones son simples:

$$\mathit{insert} :: \text{Ord } a \Rightarrow a \rightarrow \text{Heap } a \rightarrow \text{Heap } a$$
$$\mathit{insert}\ x\ h = \mathit{merge}\ (\text{N } 1\ x \text{ E E})\ h$$

$$\mathit{findMin} :: \text{Ord } a \Rightarrow \text{Heap } a \rightarrow a$$
$$\mathit{findMin}\ (\text{N } x\ a\ b) = x$$

$$\mathit{deleteMin} :: \text{Ord } a \Rightarrow \text{Heap } a \rightarrow \text{Heap } a$$
$$\mathit{deleteMin}\ (\text{N } x\ a\ b) = \mathit{merge}\ a\ b$$

- Dado que *merge* es logaritmica entonces *insert* y *deleteMin* también.
- *findMin* no es recursiva.

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.
- Alternatives to Two Classic Data Structures. Chris Okasaki.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein
- Apuntes de clases "Estructuras persistentes" Mauro Jaskelioff.