Tema: Árboles AVL

Asignatura: Métodos Algorítmicos de Resolución de Problemas
Profesor: Ricardo Peña

Curso 2013/14

1. Árboles AVL

La eficiencia de las operaciones de los árboles de búsqueda depende críticamente de poder mantener el árbol aproximadamente equilibrado mientras se realizan las inserciones y los borrados, sea cual sea el orden en el que se presenten los elementos. De no ser así, en el peor caso, el árbol construido degenera en una lista y el coste de la inserción y de la búsqueda pasaría a estar en $\Theta(n)$.

Existen diversas técnicas para equilibrar árboles de búsqueda y mantener, pese al trabajo adicional, un coste en $\Theta(\log n)$ para la inserción y el borrado. Una de las más conocidas son los árboles AVL[1]. Éstos son árboles de búsqueda con un invariante reforzado con respecto al de aquéllos, que especifica que la altura debe mantenerse dentro de ciertos límites. Las definiciones precisas son:

Definición 1 *Diremos que un árbol binario está equilibrado en altura si,*

- *O bien es vacío.*
- *O bien sus subárboles izquierdo i y derecho d son equilibrados en altura y, además,* $|altura(i) - altura(d)| \leq 1$.

Definición 2 *Diremos que un árbol t es AVL, denotado $AVL(t)$, si t es de búsqueda y t es equilibrado en altura.*

La condición de ser equilibrado en altura es más débil que la de ser de altura mínima. Por ejemplo, el árbol de la Figura 1 es equilibrado en altura, pero no es de altura mínima: el árbol de siete elementos de altura mínima tendría altura 3 en lugar de 4. Además, es AVL porque es de búsqueda.

A pesar de no ser de altura mínima, los árboles AVL siguen satisfaciendo la propiedad, que les hace interesantes como estructura de datos, de que su altura es proporcional al logaritmo en base dos de su cardinal. En adelante, si t es un árbol binario, h_t denotará la altura de t y n_t su cardinal.

Teorema 1 *Si t es equilibrado en altura, entonces $h_t \in \mathcal{O}(\log n_t)$.*

Demostración:

Denotaremos por n_h el cardinal mínimo de los árboles equilibrados en altura de altura h. Si demostramos que, para todo h, $h \in \mathcal{O}(\log n_h)$, entonces todo árbol t de altura h equilibrado en altura cumplirá $h \in \mathcal{O}(\log n_t)$ por ser $n_t \geq n_h$.

[1] Llamados así a partir de los nombres de sus autores G. M. Adelson-Velskii y E. M. Landis (1962).

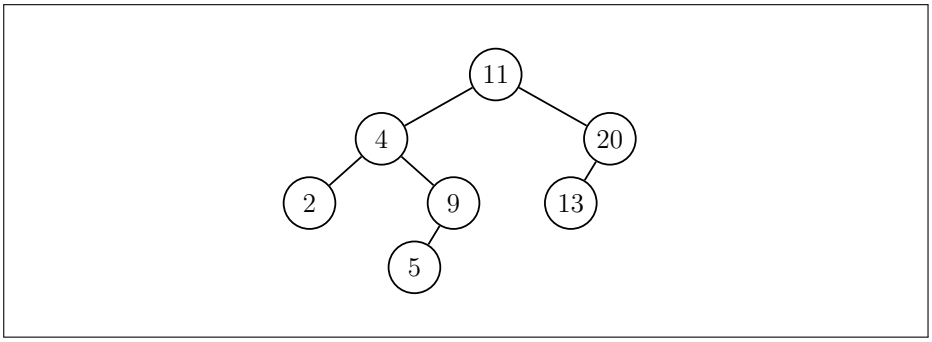

Figura 1: Ejemplo de árbol equilibrado en altura

Comenzaremos demostrando por inducción sobre h que:

$$\forall h.n_h = \mathit{fib}(h+2) - 1$$

donde $\mathit{fib}(n)$ denota el n-ésimo número de Fibonacci. Recordemos que:

$$\mathit{fib}(n) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \mathit{fib}(n-1) + \mathit{fib}(n-2) & \text{si } n \geq 2 \end{cases}$$

Casos base Es obvio que $n_0 = 0 = \mathit{fib}(2) - 1$ y $n_1 = 1 = \mathit{fib}(3) - 1$

Paso de inducción Si $h \geq 2$, por definición de cardinal, y de árbol equilibrado en altura de cardinal mínimo, se tiene $n_h = n_{h-1} + n_{h-2} + 1$. Por hipótesis de inducción se llega entonces al resultado deseado:

$$n_h = (\mathit{fib}(h+1) - 1) + (\mathit{fib}(h) - 1) + 1 = \mathit{fib}(h+2) - 1$$

Por otra parte, De Moivre demostró (por inducción sobre n) que los números de Fibonacci crecen exponencialmente según la fórmula:

$$\mathit{fib}(n) = \frac{1}{\sqrt{5}}(\phi^n - (-\phi)^{-n})$$

donde $\phi \stackrel{\text{def}}{=} \frac{1+\sqrt{5}}{2}$ es la llamada *razón áurea*. Cuando n se hace grande, el término $(-\phi)^{-n}$ se puede despreciar, con lo que podemos tomar $\mathit{fib}(n) = \frac{1}{\sqrt{5}}\phi^n$. Tenemos entonces:

$$n_h = \mathit{fib}(h+2) - 1 = \frac{1}{\sqrt{5}}\phi^{h+2} - 1 \;\Rightarrow\; h = \log_\phi(\sqrt{5}(n_h+1)) - 2$$

Es decir, $h \in \mathcal{O}(\log n_h)$ como queríamos demostrar. ∎

Vamos a desarrollar una función recursiva *esAVL* que, dado un árbol binario t, decide si t es o no AVL. Para ello, ha de determinar si t es de búsqueda y equilibrado en altura. Si t es vacío, ambas propiedades se satisfacen trivialmente. En caso contrario, se exige en primer lugar que los dos hijos las satisfagan y que, además, se respeten ciertas propiedades adicionales concernientes al orden que guarda la raíz con respecto a los hijos y a la relación que guardan las alturas de ambos.

Ello sugiere una inmersión de resultados en la que una función inmersora $esAVL'(t)$ devuelve no sólo si t es o no AVL sino, además, su altura y sus elementos mínimo y máximo. Evidentemente:

$$esAVL(t) = \textbf{sea } (b, h, mi, ma) = esAVL'(t) \textbf{ en } b$$

```
fun esAVL' (t : arbin) dev (b : bool; h : nat; mi, ma : elem) =
   caso vacio? (t)  → (T, 0, +∞, −∞)
    □  ¬vacio? (t)  → sea   (i, x, d) = desc (t);
                            (bi, hi, mii, mai) = esAVL' (i);
                            (bd, hd, mid, mad) = esAVL' (d);
                       en   (bi ∧ bd ∧ (mai < x < mid) ∧ |hi − hd| ≤ 1,
                            1 + max (hi, hd), min (mii, x), max (mad, x))
                      fcaso
   fcaso
ffun
```

Figura 2: Función que determina si un árbol binario es AVL

Cuando el árbol es vacío, los valores *mi* y *ma* no están definidos. Tras una breve reflexión, es fácil decidir que los valores apropiados son $+\infty$ para *mi* y $-\infty$ para *ma*. El algoritmo buscado se muestra en la Figura 2

Ejercicio 1 *Justificar que el coste de la llamada $esAVL'(t)$, y por tanto el de $esAVL(t)$, está en $\Theta(n_t)$.* ■

Aunque los AVL son simples árboles binarios con la implementación enlazada habitual, para obtener los costes deseados, es necesario añadir cierta información en los nodos sobre el estado de equilibrio del subárbol que tiene como raíz dicho nodo. La implementación de árboles AVL que presentamos aquí no es la original, en la que cada celda del árbol almacenaba un *factor de equilibrio* que tomaba valores en el conjunto $\{-1, 0, 1\}$, según el subárbol izquierdo fuera una unidad más alto que el derecho, ambos subárboles tuvieran la misma altura o el subárbol derecho fuera una unidad más alto que el izquierdo. En nuestro caso, conservamos directamente la altura del subárbol, es decir, nuestra representación de un árbol binario utilizado como AVL es la siguiente:

```
rep
   tipo arbin =  puntero a celda;
   tipo celda =  reg
                    e : elem; h : nat
                    izq, der : puntero a celda
                 freg
frep
```

donde h representa la altura del subárbol.

Con esta representación, redefinimos las operaciones *crear* y *altura* para árboles binarios con las implementaciones que se muestran en la Figura 3. Nótese que ahora el coste de $altura$ está en $\Theta(1)$ y el de *crear* se sigue manteniendo en $\Theta(1)$ a pesar de calcular la altura del nuevo árbol. En los algoritmos que siguen haremos uso de la notación h_t para abreviar la llamada a $altura(t)$, sabiendo que el coste va a ser constante.

Pasamos ahora a implementar las operaciones de búsqueda, inserción y borrado en AVLs. El código de la función *buscar* no cambia con respecto al de la misma función en los árboles de búsqueda no equilibrados. La diferencia es que ahora podemos garantizar que su coste está siempre en $\Theta(\log n)$ ya que, en el caso peor, recorre una altura del árbol.

```
fun altura (t : arbin) dev (h : nat) =
    caso vacio? (t)  → 0
    [] ¬vacio? (t)  → t↑.h
    fcaso
ffun

fun crear (i : arbin, x : elem, d : arbin) dev (t : arbin)
    reservar (t); t↑.izq := i; t↑.der := d; t↑.e := x;
    t↑.h := 1 + max (altura (i), altura (d)) dev t
ffun
```

Figura 3: Nueva implementación de *crear* y *altura* para árboles binarios

```
fun insertar (t : arbin, x : elem) dev (t' : arbin) =
    caso vacio? (t)  → crear(△, x, △)
    [] ¬vacio? (t)  → sea (i, y, d) = desc (t) en
                      caso x < y  → equil(insertar (i, x), y, d)
                        [] x = y  → t
                        [] x > y  → equil(i, y, insertar (d, x))
                      fcaso
    fcaso
ffun
```

Figura 4: Implementación de *insertar* para árboles AVL

1.1. Inserción en un AVL

La inserción en un AVL sigue, en principio, las mismas pautas que la función *insertar* de los árboles de búsqueda convencionales, ya que se ha de preservar el invariante de los árboles de búsqueda. El problema nuevo ahora es garantizar que el árbol resultante está equilibrado en altura si el de entrada lo está. Lo primero que haremos es especificar la nueva función *insertar* conjeturando que, tras la inserción, el AVL resultante ha crecido a lo sumo uno en altura con respecto al de entrada.

$\{AVL(t)\}$
fun *insertar* $(t : arbin, x : elem)$ **dev** $(t' : arbin)$
$\{AVL(t') \wedge elems(t') = elems(t) \cup \{x\} \wedge h_{t'} - h_t \leq 1\}$

La Figura 4 presenta el diseño de *insertar*. Nótese que la única diferencia con respecto a la versión convencional es que se han reemplazado dos llamadas a *crear* por llamadas a la función *equil*. La misión de estas llamadas es comprobar que, tras la inserción, todos los subárboles atravesados en el camino de descenso mantienen la propiedad de equilibrio. Si es así, la función *equil* simplemente llamará a *crear*. Si no es así, reorganizará el árbol para que se mantenga la propiedad de equilibrio.

Dejaremos el diseño de *equil* para más adelante, ya que esta función también será llamada durante el borrado. Para razonar sobre la corrección de *insertar*, nos bastará con su especificación.

Definimos para ello el siguiente predicado:

$$promete(i, x, d) \stackrel{\text{def}}{=} AVL(i) \wedge AVL(d) \wedge x \notin elems(i) \cup elems(d) \wedge ord(inorden(i) +\!\!+ [x] +\!\!+ inorden(d))$$

El significado intuitivo de $promete(i, x, d)$ es que la terna (i, x, d) reúne las condiciones idóneas para formar un árbol de búsqueda, si bien éste no sería necesariamente equilibrado. La especificación propuesta para *equil* es la siguiente:

$$\{promete(i, x, d) \wedge |h_i - h_d| \leq 2\}$$
fun $equil\ (i : arbin, x : elem, d : arbin)$ **dev** $(t : arbin)$
$$\{AVL(t) \wedge elems(t) = elems(i) \cup \{x\} \cup elems(d) \wedge max(h_i, h_d) \leq h_t \leq max(h_i, h_d) + 1\}$$

La corrección de *insertar* se deduce de los siguientes apartados:

1. Los casos trivial y no trivial cubren todas las posibilidades para t y para la relación entre el elemento insertado x y la raíz del árbol.

2. Las llamadas internas a *insertar* cumplen su precondición ya que, tanto i como d, son AVLs.

3. Las llamadas a *equil* cumplen su precondición ya que, por un lado, sus parámetros satisfacen el predicado *promete*, si hacemos hipótesis de inducción sobre las llamadas a *insertar*. En efecto, centrándonos en la primera rama, vemos que el elemento insertado x es menor que la raíz y y que el árbol devuelto por *insertar* es un AVL. Por otro lado, la altura tras la inserción, haciendo hipótesis de inducción, ha crecido a lo sumo en uno. Si el árbol i ya era uno más alto que d antes de insertar, ahora será a lo sumo dos unidades más alto.

4. La postcondición de *equil* garantiza la de *insertar*. En primer lugar, *equil* devuelve un AVL con los elementos requeridos. En cuanto a la altura, y centrándonos de nuevo en la primera rama, si antes de insertar en i teníamos $h_i = h + 1$ y $h_d = h$, tras la inserción y en el caso peor i tendrá altura $h + 2$ y el árbol devuelto por *equil* tendrá a lo sumo altura $h + 3$ que es una unidad más que la altura del árbol t recibido por *insertar*. El lector puede comprobar que también se satisfacen las cotas mínimas para la altura.

5. En el caso base en el que el árbol de entrada es vacío, el árbol devuelto por *insertar* es AVL y tiene altura 1, una unidad más que la del árbol de entrada. El otro caso base se produce cuando el elemento a insertar ya está en el árbol. El árbol devuelto t es el de entrada que, según la precondición, es AVL. En ese caso, la altura se mantiene igual.

El número de llamadas a *insertar* coincide, en el caso peor, con la altura del árbol de entrada, que sabemos está en $\mathcal{O}(\log n)$. El coste de cada llamada está en $\Theta(1)$, si anticipamos que ése será también el coste de cada llamada a *equil*. Por tanto, el coste de la inserción estará en $\mathcal{O}(\log n)$.

1.2. Borrado en un AVL

Al igual que ocurría con *insertar*, el borrado seguirá las mismas pautas que la función *borrar* de los árboles de búsqueda convencionales. Comenzamos por su especificación formal:

$$\{AVL(t)\}$$
fun $borrar\ (t : arbin, x : elem)$ **dev** $(t' : arbin)$
$$\{AVL(t') \wedge elems(t') = elems(t) - \{x\} \wedge h_t - h_{t'} \leq 1\}$$

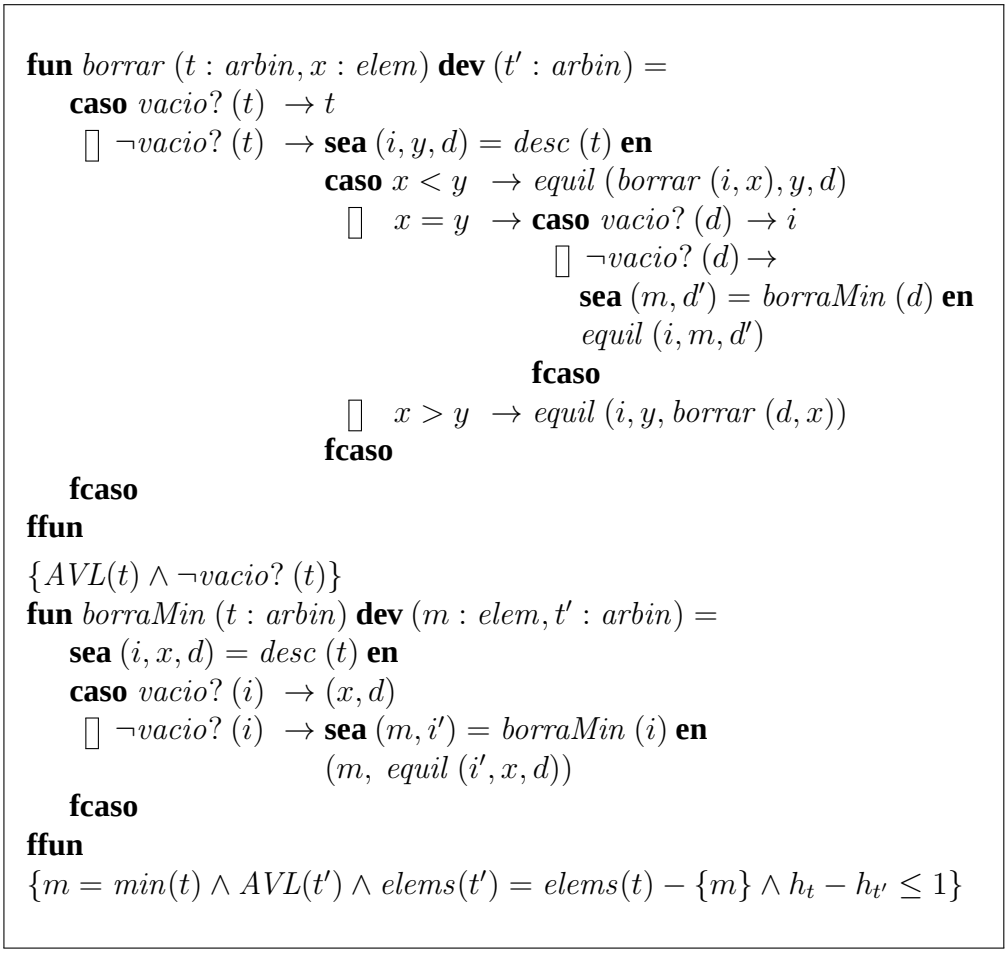

Figura 5: Implementación de *borrar* y *borraMin* para árboles AVL

El diseño se presenta en la Figura 5. De nuevo, la única diferencia de este diseño con respecto al del borrado en un árbol de búsqueda convencional es la sustitución de las llamadas a *crear* por llamadas a la función *equil* que se ocupa del reequilibrio. Razonamos la corrección de *borrar* del modo siguiente:

1. Los casos trivial y no trivial cubren todas las posibilidades para t y para la relación entre el elemento a borrar x y la raíz del árbol.

2. Las llamadas internas a *borrar* cumplen su precondición ya que, tanto i como d, son AVLs.

3. Las llamadas a *equil* cumplen su precondición ya que, si hacemos hipótesis de inducción sobre las llamadas a *borrar*, sus parámetros satisfacen el predicado *promete*. En efecto, centrándonos en la primera rama, vemos que el elemento borrado x es menor que la raíz y y que el árbol devuelto por *borrar* es un AVL. Por otro lado, la altura, tras el borrado y haciendo hipótesis de inducción, ha decrecido a lo sumo en uno. Si el árbol i antes de borrar fuera una unidad más bajo que d, ahora sería, a lo sumo, dos unidades más bajo. Lo mismo sucede en la llamada a *equil* después de llamar a *borraMin* ya que, según su especificación, el árbol devuelto d' es igual en altura, o una unidad más bajo, que d.

4. La llamada a *borraMin* cumple su precondición ya que el árbol d es AVL y no vacío.

5. La postcondición de *equil* garantiza la de *borrar*. En primer lugar, *equil* devuelve un AVL con los elementos requeridos. En cuanto a la altura y centrándonos de nuevo en la primera

rama, si antes de borrar en i teníamos $h_i = h - 1$ y $h_d = h$, tras el borrado y en el caso peor, i tendrá altura $h - 2$ y el árbol devuelto por *equil* tendrá como mínimo altura h, que es una unidad menos que la altura del árbol t recibido por *borrar*. El lector puede comprobar que el razonamiento es similar para la llamada a *equil* después de llamar a *borraMin* y también que se satisfacen las cotas máximas para la altura.

6. En el caso base en el que el árbol t de entrada es vacío, el árbol devuelto por *borrar* es el propio t, que es AVL y mantiene la altura. El otro caso base se produce cuando el elemento a borrar está en el árbol y su subárbol derecho d es vacío. Entonces se devuelve el izquierdo i, que es AVL y cuya altura es siempre una unidad menor que la del árbol de entrada (recuérdese que d es vacío).

La corrección de *borraMin* sigue las mismas pautas y se deja como ejercicio.

Ejercicio 2 *Demostrar formalmente la corrección de borraMin, utilizando la especificación de equil.* ■

La función *borrar* recorre una altura del árbol hasta encontrar el elemento a borrar. A partir de aquí, se producen tantas llamadas a *borraMin* como el valor del resto de la altura hasta llegar al elemento situado más a la izquierda del subárbol derecho del elemento borrado. En cada una de estas llamadas se realiza una llamada a *equil* y el resto de las operaciones tiene coste constante. Suponiendo un coste en $\Theta(1)$ para *equil*, concluimos que el coste de *borrar* está en $\mathcal{O}(\log n)$.

1.3. Funciones de reequilibrado

La función *equil* comprueba en primer lugar si es posible formar un árbol equilibrado con los tres componentes recibidos. Si es así, lo forma con una llamada a *crear*. En caso contrario, las únicas posibilidades son que, o bien el árbol izquierdo, o bien el árbol derecho, son dos unidades más alto que el otro subárbol, en cuyo caso *equil* delega el reequilibrio a dos funciones especializadas *desequilIzq* y *desequilDer*. En la Figura 6 puede verse el diseño de *equil* y la especificación de las dos funciones auxiliares.

En adelante, desrrollaremos *desequilIzd* y dejaremos la otra función como ejercicio para el lector, por ser los razonamientos sobre ambas funciones totalmente simétricos. El reequilibrio consistirá en producir *rotaciones* en las que ciertos subárboles pasan de ser subárboles de un nodo a serlo de otro. Cada rotación preserva el invariante de los árboles de búsqueda, es decir, el recorrido en inorden después de la rotación producirá la misma lista que antes de la misma.

El primer razonamiento sobre *desequilIzd* es que, por cumplirse $h_i = h_d + 2$, se tiene que $n_i \geq 2$ y por tanto i no es vacío. Llamaremos ii e id respectivamente a los subárboles izquierdo y derecho de i. Pueden darse dos situaciones excluyentes:

- $h_{ii} \geq h_{id}$. Es decir, el subárbol ii es el causante del desequilibrio. La rotación correspondiente se denomima LL (de *left-left*) y se ilustra gráficamente en la Figura 7.

- $h_{ii} < h_{id}$. Es decir, el subárbol id es el causante del desequilibrio. Además, por ser $n_i \geq 2$, se puede asegurar que el subárbol id no es vacío. Llamaremos idi e idd respectivamente a sus subárboles izquierdo y derecho. La rotación correspondiente se denomima LR (de *left-right*) y se ilustra gráficamente en la Figura 8.

La rotaciones simétricas correspondientes a los desequilibrios a la derecha se denominan RR y RL. Es fácil convencerse de que, en ambos casos LL y LR, se preserva el invariante de los árboles

fun $equil\ (i : arbin, x : elem, d : arbin)$ **dev** $(t : arbin) =$
  caso $|h_i - h_d| \leq 1 \ \rightarrow crear\ (i, x, d)$
  $\square$ $h_i = h_d + 2 \ \rightarrow desequilIzq\ (i, x, d)$
  $\square$ $h_i + 2 = h_d \ \rightarrow desequilDer\ (i, x, d)$
  fcaso
ffun

$\{promete(i, x, d) \wedge h_i = h_d + 2\}$
fun $desequilIzq\ (i : arbin, x : elem, d : arbin)$ **dev** $(t : arbin)$
$\{AVL(t) \wedge elems(t) = elems(i) \cup \{x\} \cup elems(d) \wedge$
$max(h_i, h_d) \leq h_t \leq max(h_i, h_d) + 1\}$

$\{promete(i, x, d) \wedge h_d = h_i + 2\}$
fun $desequilDer\ (i : arbin, x : elem, d : arbin)$ **dev** $(t : arbin)$
$\{AVL(t) \wedge elems(t) = elems(i) \cup \{x\} \cup elems(d) \wedge$
$max(h_i, h_d) \leq h_t \leq max(h_i, h_d) + 1\}$

Figura 6: Implementación de *equil* para árboles AVL

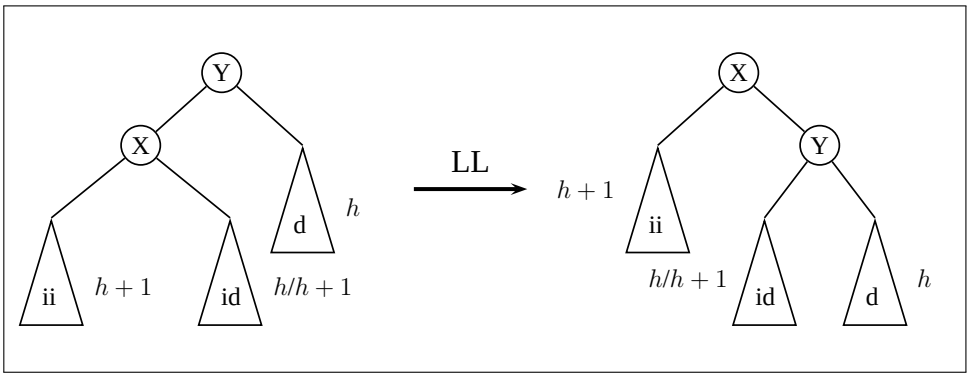

Figura 7: Rotación LL en un árbol AVL

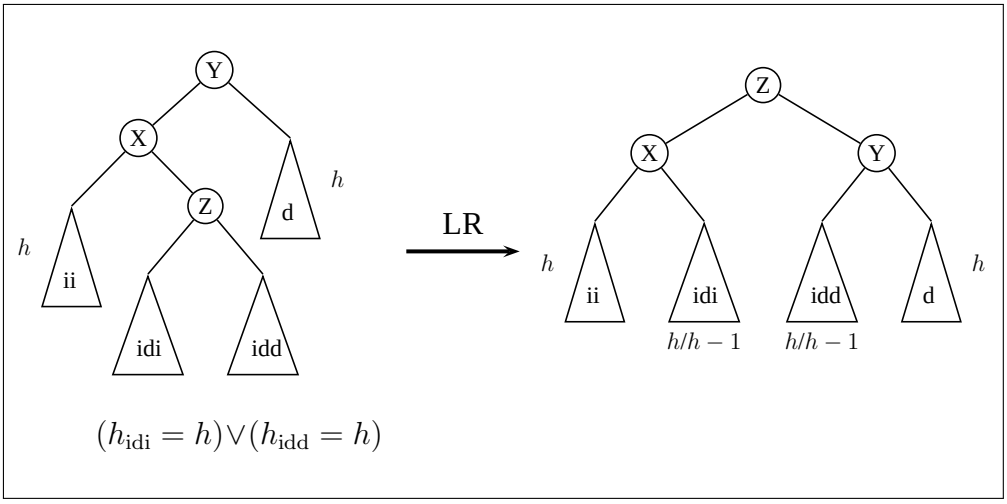

Figura 8: Rotación LR en un árbol AVL

```
fun desequilIzq (i : arbin, y : elem, d : arbin) dev (t : arbin) =
    sea (ii, x, id) = desc (i) en
    caso h_ii ≥ h_id  → crear (ii, x, crear (id, y, d))
      □  h_ii < h_id  → sea (idi, z, idd) = desc (id)
                        en crear (crear (ii, x, idi), z, crear (idd, y, d))
    fcaso
ffun
```

Figura 9: Implementación de *desequilIzq* para árboles AVL

de búsqueda. Si denotamos l_t a la lista en inorden de un árbol t, se cumplirán las siguientes igualdades:

$$(l_{ii} +\!\!+ [x] +\!\!+ l_{id}) +\!\!+ [y] +\!\!+ l_d = l_{ii} +\!\!+ [x] +\!\!+ (l_{id} +\!\!+ [y] +\!\!+ l_d)$$
$$(l_{ii} +\!\!+ [x] +\!\!+ (l_{idi} +\!\!+ [z] +\!\!+ l_{idd})) +\!\!+ [y] +\!\!+ l_d = (l_{ii} +\!\!+ [x] +\!\!+ l_{idi}) +\!\!+ [z] +\!\!+ (l_{idd} +\!\!+ [y] +\!\!+ l_d)$$

En cuanto a las alturas, en el caso LL la altura del árbol resultante es, o bien $h + 2$, o bien $h+3$, dependiendo de si la altura h_{id} es h o $h+1$. En cualquier caso, se satisface la postcondición de *desequilIzq*, es decir $max(h_i, h_d) \leq h_t \leq max(h_i, h_d) + 1\}$. En el caso LR, la altura del árbol resultante es siempre $h + 2$, que coincide con $h_i = max(h_i, h_d)$. Nótese también que, en ambos casos, el árbol resultante es AVL.

La implementación de *desequilIzq*, según el análisis que acabamos de realizar, se muestra en la Figura 9. Como habíamos anticipado, el coste de todas las operaciones involucradas es constante. Por tanto, los costes de *desqequilIzq*, de *desequilDer* y de *equil* están todos ellos en $\Theta(1)$.

Ejercicio 3 *Demostrar que, si se produce un desequilibrio LL (respectivamente, RR) durante la ejecución de insertar, entonces nunca se produce la situación en que $h_{ii} = h_{id}$ (respectivamente, $h_{dd} = h_{di}$). Como corolario, demostrar que a lo sumo una de las llamadas a equil produce a su vez una llamada a desequilIzq o desequilDer.* ■

Ejercicio 4 *Para familiarizarse con las rotaciones, se invita al lector a realizar la siguiente secuencia de inserciones a partir del árbol vacío: 5, 10, 20, 16, 13 ,7, 6, 11, 12 y 14. En el árbol resultante, realizar la siguiente secuencia de borrados: 13, 12, 6, 5, 10, 7, 14, 11, 16 y 20.* ■

Ejercicio 5 *Estudiar el consumo de memoria dinámica de insertar y borrar en un AVL. Proporcionar versiones de estas funciones y de sus auxiliares que tan sólo consuman memoria fresca cuando se inserta un elemento nuevo y que liberen una celda cuando se borra un elemento que ya estaba en el árbol.* ■

Lecturas complementarias

Estas notas están tomadas literalmente de la Sec. 7.3 del libro [1]. En el Cap. 7 del mismo, pueden encontrarse otros árboles equilibrados, como los 2-3, así como varias implementaciones de montículos.

Referencias

[1] R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.