DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Balanceo de árboles y árboles AVL

Guido Tagliavini Ponce
Universidad de Buenos Aires
guido.tag@gmail.com

Índice

1. Introducción

La estructura de árbol es una idea fundamental en la computación. Su importancia radica en su capacidad para estructurar información de manera dinámica, es decir, posibilitando la fácil realización de cambios en esa información a lo largo del tiempo.

El concepto de balanceo aparece como una solución al crecimiento desmesurado de la altura de los mismos, lo cual imposibilita realizar operaciones que requieran atravesar el árbol desde la raíz hasta una hoja en forma eficiente.

En este trabajo revisamos el concepto de balanceo, y estudiamos diversas familias de árboles que son balanceados. Una de estas familias derivará en el concepto de árbol AVL. Estos árboles son balanceados y permiten ejecutar eficientemente las operaciones de diccionario, lo cual los hace una representación perfecta para estos tipos abstractos de datos. Por esta razón, en la segunda parte del trabajo nos concentramos en la implementación detallada de las operaciones de un AVL.

2. Balanceo de árboles

2.1. Motivación

Intuitivamente, que un árbol esté balanceado significa que no hay subárboles que sean mucho más grandes (en algún sentido de *grande*) que otros subárboles. Una interpretación gráfica de esto es que un árbol balanceado es *llano*. Los árboles por excelencia que cumplen esta noción intuitiva de balanceo son los completos (aquellos que tienen todos los niveles completos), puesto que su altura es chica en relación a la cantidad de nodos que concentra entre todos sus niveles. El hecho de que un árbol binario completo de n nodos tiene altura aproximadamente $\lg n$, motiva la definición central de todo este apunte.

Definición. Un árbol de n nodos se dice *balanceado* si su altura es $O(\lg n)$

Retomando la discusión de la introducción, la utilidad práctica de los árboles balanceados aparece cuando se combina este concepto con el de árbol de búsqueda. Recordemos este concepto.

Definición. Un árbol binario con claves en los nodos se dice *de búsqueda* (y escribimos ABB) si:

- toda clave en el subárbol izquierdo es menor que la clave de la raíz;
- toda clave en el subárbol derecho es mayor que la clave de la raíz;
- tanto el subárbol izquierdo como el derecho son árboles binarios de búsqueda.

Un ABB permite determinar la existencia de una clave en el árbol en tiempo proporcional a la altura del árbol [1]. En un ABB de n nodos cualquiera, esto puede ser, en el peor caso, $O(n)$. Sin embargo, si el árbol es balanceado, su altura será $O(\lg n)$, con lo cual el costo de la búsqueda baja a $O(\lg n)$.

Nuestro objetivo será definir familias de árboles binarios que sean balanceados. Como hemos mostrado, nuestro interés se centra en árboles que puedan ser usados como estructuras de representación de diccionarios, por lo que una propiedad deseada para estas familias es que al ser restringidas al conjunto de ABBs, soporten la inserción

y el borrado de nodos (es decir, la actualización a lo largo del tiempo), de modo tal que al efectuar tales operaciones se obtenga otro árbol de la familia. Esta última propiedad es lo que nos permitirá asegurar que a lo largo de una secuencia de operaciones, los costos de inserción, borrado y búsqueda se mantienen. Esto quedará más claro y resultará evidente al ver ejemplos concretos.

2.2. Árboles binarios completos

Como hemos visto, los árboles binarios completos son una familia de árboles balanceados elemental. El hecho de que sean balanceados se fundamenta en el siguiente resultado, al que ya hemos hecho alusión.

Teorema. *Un árbol binario completo* T *de* n *nodos tiene altura* $\lg(n+1)$.

Demostración. Sea h la altura de T. Como T es completo, los h niveles de T están completos, es decir que

$$n = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

□

Corolario. *Un árbol binario completo es balanceado.*

El problema con esta clase de árboles, es que son extremadamente rígidos, en el sentido que no es posible realizar una inserción o borrado de un nodo y obtener otro árbol completo, como muestra la figura 1.

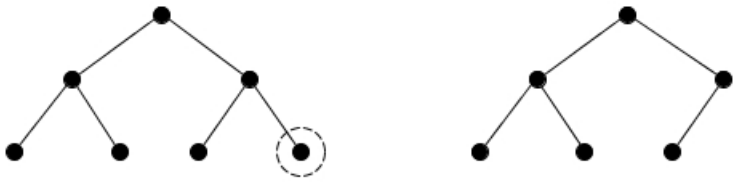


Figura 1: un árbol binario completo deja de ser completo al eliminarle un nodo

2.3. Balanceo perfecto

Necesitamos relajar un poco la rigidez de los árboles completos. Observemos que una característica de un arbol completo es que todas sus hojas estan en el ultimo o en el anteultimo nivel. Obviamente, esta propiedad caracteriza a muchos otros árboles que no son completos, como muestra la figura 2. Para evitar casos de extremo desbalanceo, como el de la figura, pediremos que no haya nodos que tengan un sólo hijo, que se

puede ver que son los que abundan en el ejemplo, y que causan que existan ramas largas.

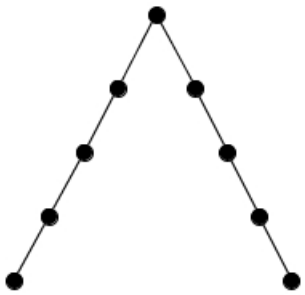


Figura 2: árbol que tiene sus hojas al mismo nivel, pero no es balanceado

Con todo esto en mente, vamos a definir un nuevo criterio de balanceo, que no es más que una serie de restricciones sobre árboles binarios que definen una familia.

Definición. Un árbol binario se dice *lleno* si todo nodo tiene 0 o 2 hijos.

Definición. Un árbol binario se dice *perfectamente balanceado* si:

- es un árbol lleno;
- todas sus hojas se encuentran en el último ó anteúltimo nivel.

La figura 3 muestra un árbol binario perfectamente balanceado. Notar que tiene forma achatada, condiciéndose, intuitivamente, con la noción de balanceo. El siguiente resultado expresa formalmente esto último.

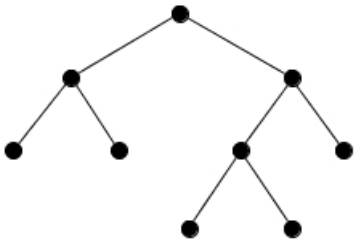


Figura 3: árbol perfectamente balanceado

Teorema. *Un árbol binario perfectamente balanceado* T *de* n *nodos tiene altura* $\lceil \lg(n+1) \rceil$.

Demostración. Sea h la alura de T. Lo primero que veremos es que cada nivel $0 \leq i < h-1$ está completo. En efecto, si suponemos que $i > 0$ es el primer nivel que no está completo, entonces existe un nodo v en el nivel $i-1$ que no tiene dos hijos, dado que $i-1$ está completo. Como T es un árbol lleno, entonces v debe tener 0 hijos, es decir que es una hoja. Luego, T tiene una hoja en el nivel $i-1 < h-2$, lo cual contradice al hecho de que todas las hojas de dicho árbol están en el nivel $h-1$ ó $h-2$.

Por lo anterior, debe ser

$$n > \sum_{i=0}^{h-2} 2^i = 2^{h-1} - 1$$

La desigualdad es estricta debido a que T tiene al menos un nodo en el nivel $h-1$. Por otro lado, como T tiene altura h, se tiene

$$n \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Juntando ambas desigualdades,

$$2^{h-1} - 1 < n \leq 2^h - 1$$

Sumando 1 y tomando logaritmos,

$$h - 1 < \lg(n+1) \leq h$$

Es decir que $h = \lceil \lg(n+1) \rceil$.

□

Corolario. *Un árbol binario perfectamente balanceado es balanceado.*

Lamentablemente, esta familia de árboles balanceados sigue padeciendo algunos de los problemas que tienen los árboles binarios completos. Particularmente sufre el problema de que no hay árboles perfectamente balanceados para cada cantidad de nodos posible. La figura 4 muestra un ejemplo.

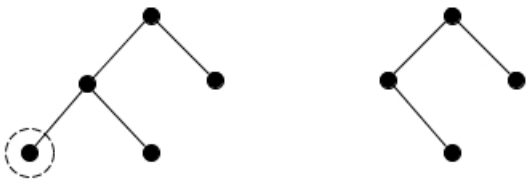


Figura 4: árbol binario perfectamente balanceado deja de ser perfectamente balanceado al eliminar un nodo

2.4. Balanceo en altura

Definición. Un árbol binario se dice *balanceado en altura* si para cada nodo, la altura de su subárbol derecho e izquierdo difieren a lo sumo en una unidad.

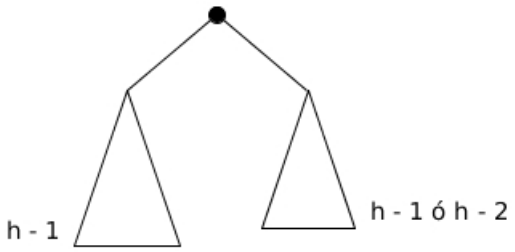
Teorema. *Un árbol binario balanceado en altura es balanceado.*

Demostración. La pregunta clave de la demostración es,

¿cuál es la mínima cantidad de nodos que puede tener un árbol binario balanceado en altura, de altura h*?*

Intuitivamente, responder a esta pregunta es útil porque fijado un árbol balanceado en altura de n nodos y altura h, dicho árbol no puede ser demasiado ralo (es decir n debe ser suficientemente grande) debido a que el balance debería obligar a que haya una gran cantidad de nodos concentrados a lo largo de los h niveles.

Llamemos $n_{min}(h)$ a la mínima cantidad de nodos de un árbol binario balanceado de altura h. Es claro que $n_{min}(1) = 1$ y $n_{min}(2) = 2$. Si $h > 2$, ¿cuánto vale $n_{min}(h)$? Un árbol binario balanceado en altura de n nodos y altura $h > 2$ debe tener dos subárboles, izquierdo y derecho. De la definición de balanceo en altura se deduce que estos dos también son balanceados en altura. Como T tiene altura h, entonces uno de ellos tiene altura $h-1$, y el otro altura $h-1$ o $h-2$, puesto que la diferencia entre las alturas de los mismos es a lo sumo 1. Gráficamente, la situación es la siguiente:



Luego,

$$n \geq 1 + n_{min}(h-1) + \text{mín}(n_{min}(h-1), n_{min}(h-2))$$

Como T es arbitrario de altura h, resulta que $n_{min}(h) \geq 1 + n_{min}(h-1) + \text{mín}(n_{min}(h-1), n_{min}(h-2))$. De esta desigualdad, más los casos base, se sigue inmediatamente que $n_{min}(h) > n_{min}(h-1)$ para todo $h > 1$, con lo cual la expresión anterior se simplifica a

$$n_{min}(h) \geq 1 + n_{min}(h-1) + n_{min}(h-2)$$

Para probar la igualdad, basta ver que hay algún árbol binario balanceado en altura, de altura h, con cantidad de nodos $1+n_{min}(h-1)+n_{min}(h-2)$. Este árbol se puede construir fácilmente en forma inductiva, razonando igual que antes. En definitiva,

$$n_{min}(h) = 1 + n_{min}(h-1) + n_{min}(h-2)$$

Observemos que esta recurrencia tiene un parecido con la sucesión de Fibonacci. La siguiente tabla muestra los primeros valores de n_{min} y Fib.

h	$n_{min}(h)$	$\mathrm{Fib}(h)$
1	1	1
2	2	1
3	4	2
4	7	3
5	12	5
6	20	8
7	33	13
8	54	21

Queda como ejercicio para el lector probar que $n_{min}(h) = \mathrm{Fib}(h+2)-1$. Dado que $\mathrm{Fib}(n) = \Omega(2^n)$, tenemos $n_{min}(h) = \Omega(2^h)$.

Finalmente, tomemos un árbol binario balanceado en altura de n nodos y altura h, y veamos que $h = O(\lg n)$. Se tiene $n \geq n_{min}(h) = \Omega(2^h)$, con lo cual $n = \Omega(2^h)$. Pero entonces $h = O(\lg n)$. □

Como veremos en la segunda parte de este trabajo, esta familia de árboles binarios balanceados ha probado ser útil.

2.5. Balanceo en peso

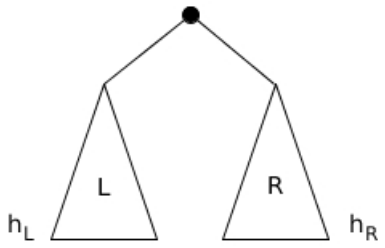
El último criterio de balanceo que estudiaremos no es una propuesta superadora del balanceo en altura. Aún así, es interesante conocerlo, sencillamente con el fin de reforzar la idea de que hay diversas propiedades que se le pueden exigir a un árbol de manera tal de asegurar su balance, aunque no todas son igualmente útiles debido a su mayor o menor flexibilidad.

Definición. Un árbol se dice *balanceado en peso* si para cada nodo, la cantidad de nodos en su subárbol derecho e izquierdo difieren a lo sumo en una unidad.

Teorema. *Un árbol binario balanceado en peso* T *de* n *nodos tiene altura* $\lceil \lg(n+1) \rceil$.

Demostración. Sea h la altura de T. Veamos, primero, que al igual que los árboles binarios perfectamente balanceados, esta clase de árboles tienen cada nivel $0 \leq i < h-1$ completo. Procedemos por inducción en h. El caso $h = 1$ es trivial. Sea $h > 1$. Sea r la raíz de T. Si r tiene un único hijo entonces T sólo puede estar compuesto por r y su hijo, puesto que es balanceado en peso, con lo cual vale el resultado. Supongamos

que r tiene dos hijos. Entonces, r tiene un subárbol izquierdo L y uno derecho R, con alturas h_L y h_R respectivamente, y tamaños n_L y n_R respectivamente. Gráficamente este árbol tiene la siguiente forma:



Como T es balanceado, se tiene

$$|n_R - n_L| \leq 1 \tag{1}$$

Más aún, tanto L como R son balanceados en peso. Como además ambos tienen altura menor que h, vale la hipótesis inductiva sobre ellos, con lo cual L y R tienen todo nivel completo, excepto, quizás, el último.

Como T tiene altura h entonces podemos suponer, sin pérdida de generalidad, que $h_L = h - 1$. Como L tiene los primeros $h - 2$ niveles completos, vale

$$n_L > \sum_{i=0}^{h-3} 2^i = 2^{h-2} - 1$$

La desigualdad es estricta porque L tiene al menos un nodo en el nivel $h - 2$. En definitiva

$$n_L \geq 2^{h-2} \tag{2}$$

De la ecuación (1) se deduce que $n_R \geq n_L - 1$. Juntando esto con (2) resulta que

$$n_R \geq 2^{h-2} - 1 \tag{3}$$

Como R tiene altura h_R, entonces $n_R \leq 2^{h_R - 1} - 1$. Combinando esto último con (3),

$$2^{h-2} - 1 \leq 2^{h_R - 1} - 1$$

Luego $h_R \geq h - 1$, pero como h_R tiene altura a lo sumo $h - 1$, debe ser $h_R = h - 1$. En definitiva, ambos subárboles de r tienen la misma altura $h - 1$ y tienen todos sus niveles completos, salvo quizás el último, lo cual prueba lo que queríamos y concluye la inducción.

A partir de aquí se procede igual que en el caso del balanceo perfecto.

□

3. Árboles AVL

Definición. Un árbol AVL es un ABB balanceado en altura.

El nombre de estos árboles proviene de las iniciales de sus dos creadores, Adelson-Velskii y Landis, y fue el primer ABB balanceado publicado en la historia. En esta segunda parte del trabajo, veremos cómo estos árboles permiten implementar las operaciones de diccionario, todas en tiempo logarítmico, gracias a la noción de balanceo que utilizan.

Un concepto clave a la hora de hablar de balanceo en altura es el siguiente.

Definición. Se define el factor de balanceo de un nodo v de un árbol binario como

$$\mathrm{fb}(v) = \text{altura del subárbol derecho de } v - \text{altura del subárbol izquierdo de } v$$

Por lo tanto, un árbol binario es balanceado en altura si para cada nodo v, vale $|\mathrm{fb}(v)| \leq 1$. En particular, cada nodo de un AVL tendrá un factor de balanceo 0, 1 ó -1.

Como ya hemos dicho, las dos operaciones que buscamos soportar son las de búsqueda, inserción y borrado de claves. Dado que un AVL es, en particular, un ABB, la búsqueda sobre estos árboles es exactamente igual a la búsqueda sobre ABBs, y tendremos asegurado un costo logarítmico gracias al balanceo. Para la inserción y búsqueda, el esquema de las operaciones será el siguiente:

1. Realizar la operación como en un ABB clásico.
2. Rebalancear aquellos nodos desbalanceados.

En este documento no profundizamos sobre las operaciones clásicas sobre ABBs, de modo que el lector interesado puede remitirse a [1]. De cualquier manera, a efectos de entender el funcionamiento de un AVL no podemos abstraernos completamente del funcionamiento de las operaciones clásicas de ABB. Necesitaremos hacer uso de algunos hechos que enunciaremos en su debido momento.

Las figuras 5 y 6 muestran por qué es necesario realizar el paso 2, luego de una inserción o un borrado, respectivamente. En la figura 5, el ABB inicialmente posee todos sus nodos balanceados, pero después de insertar la clave -15, el nodo de clave -1 pasa a tener factor de balanceo -2. En la figura 6, partimos del mismo árbol AVL y al borrar la clave 5, el nodo de clave 2 pasa a tener factor de balanceo -2.

3.1. Rotaciones en ABBs

Para remendar los nodos desbalanceados, realizaremos sobre ciertos nodos, una clase de operaciones que se conocen como *rotaciones*. Dado que nuestro objetivo es recuperar la condición de AVL, necesitamos no sólo arreglar los factores de balanceo, sino también mantener el invariante de ABB.

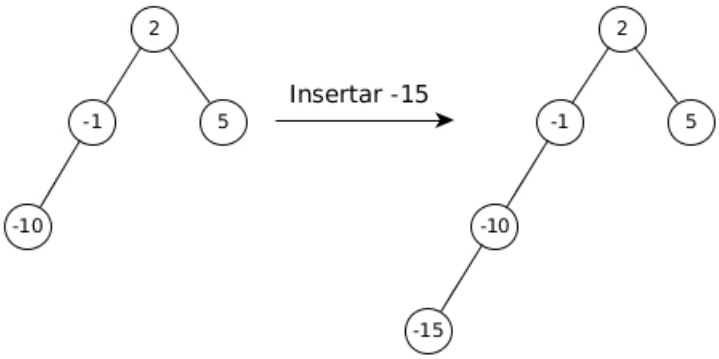


Figura 5: desbalanceo de un nodo en una inserción

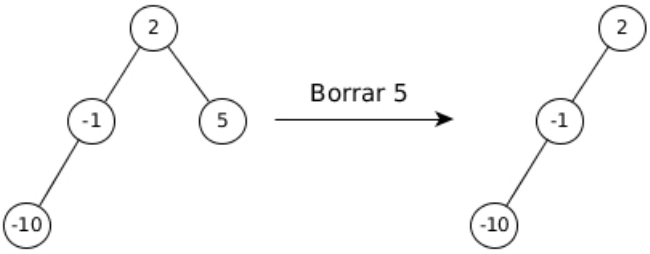


Figura 6: desbalanceo de un nodo en un borrado

Para restaurar el balanceo, utilizaremos únicamente dos operaciones de rotación, que se ilustran en la figura 7. Notar que estas operaciones son independientes de cualquier noción de balanceo, y que son aplicables en cualquier ABB, puesto que preservan el invariante de representación de estos árboles. En otras palabras, al aplicar cualquiera de las rotaciones a un ABB, el resultado es otro ABB.

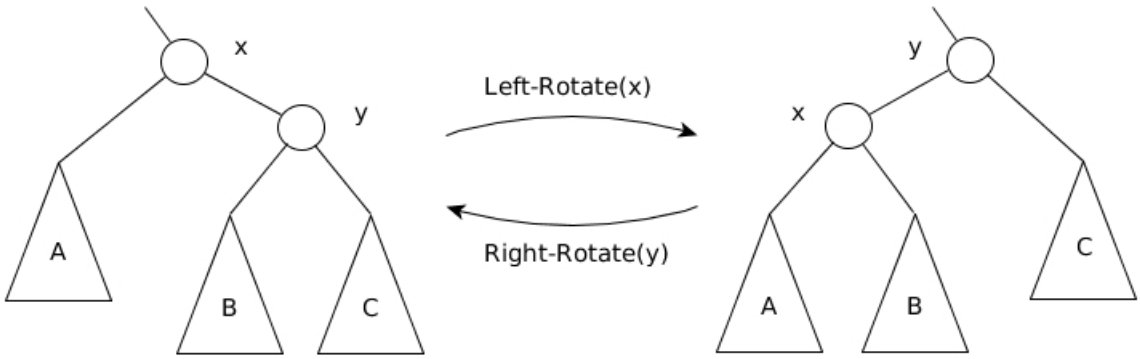


Figura 7: rotaciones en ABBs

Las dos operaciones son una inversa de la otra. Por un lado, LEFT-ROTATE(x) toma un nodo x, que debe tener un hijo derecho y, rotando hacia la izquierda a x, dejando a y como raíz del subárbol, y colgando a la derecha de x el subárbol izquierdo de y (si hubiera uno). Observar que al rotar a x a la izquierda, estamos *bajando* a este nodo y *levantando* al nodo y.

Por otro lado, RIGHT-ROTATE(y) exige que y tenga un hijo izquierdo, y realiza una operación análoga. En este caso, es al nodo y que rotamos hacia la derecha, haciendo que *baje* y, consecuentemente, que *suba* su hijo izquierdo x.

Intuitivamente, utilizaremos LEFT-ROTATE(x) para trasladar un poco del *peso* del subárbol derecho hacia el izquierdo, y RIGHT-ROTATE(y), contrariamente, para llevar peso del subárbol izquierdo hacia el derecho. Aquí cuando hablamos de peso, no nos referimos a una cantidad de nodos sino a una altura, que es aquello que intentamos balancear.

3.2. Casos de desbalanceo

Al realizar una inserción o borrado y encontrarnos con que algunos factores de balanceo son estrictamente mayores que 1, resulta conveniente resolver cada uno de estos problemas progresivamente, rebalanceando los nodos más profundos, continuando hacia los más cercanos a la raíz. Por esta razón, es útil plantearse los distintos escenarios en los que tenemos un subárbol cuya raíz está desbalanceada en una unidad (es decir, tiene factor de balanceo -2 ó 2), pero tal que todo otro nodo debajo de dicho nodo está balanceado.

Llamemos p a la raíz de un tal subárbol. Concentrémonos, primero, en el caso $fb(p) = 2$. Supongamos que el subárbol izquierdo de p tiene altura h. Entonces, el subárbol derecho de p tiene altura $h + 2$. Este subárbol derecho debe tener al menos dos nodos, con lo cual podemos nombrar q a su raíz. La figura 8 muestra la estructura

de este árbol. A partir de ahora, los números dentro de cada nodo representarán los factores de balanceo de los mismos.

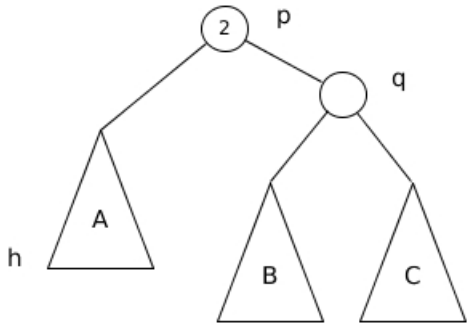


Figura 8: $\mathrm{fb}(p) = 2$

Dividimos nuevamente en casos, según el valor de $\mathrm{fb}(q)$. Como asumimos que todos los nodos debajo de p están balanceados, debe ser $\mathrm{fb}(q) \in \{-1, 0, 1\}$.

Supongamos $\mathrm{fb}(q) = 1$. Observar que esto determina las alturas de ambos subárboles de q. Como el subárbol derecho de p es mucho más alto que el izquierdo, debemos realizar algún tipo de rotación que traslade peso desde la derecha hacia la izquierda. Por este motivo, parece razonable ejecutar una LEFT-ROTATE(p), y como muestra la figura 9, esta operación reestablece el balance.

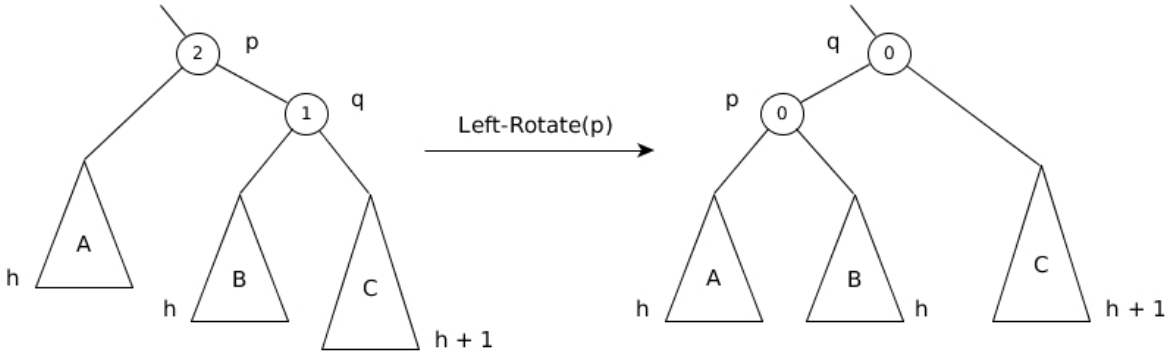


Figura 9: $\mathrm{fb}(p) = 2$ y $\mathrm{fb}(q) = 1$ **(caso A)**

Supongamos $\mathrm{fb}(q) = 0$. Utilizando la misma estrategia de rotar p hacia la izquierda, vemos que nuevamente logramos rebalancear el subárbol, como muestra la figura

10.

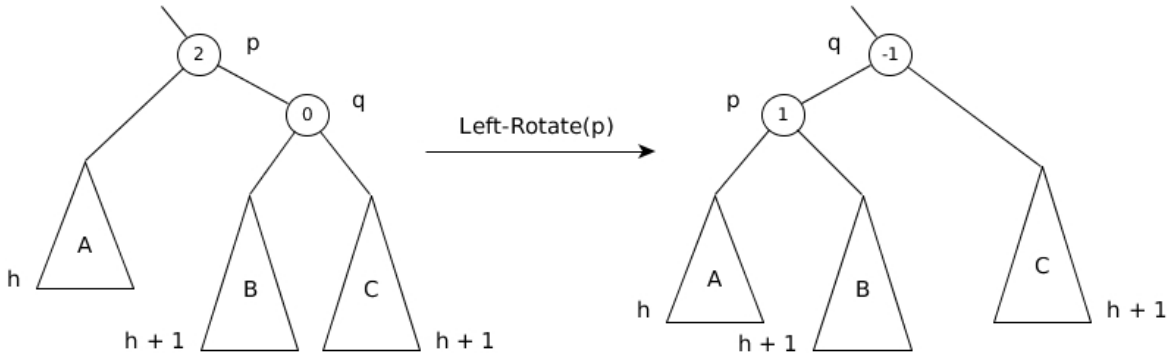


Figura 10: $\mathtt{fb}(p) = 2$ y $\mathtt{fb}(q) = 0$ **(caso B)**

Finalmente, supongamos $\mathtt{fb}(q) = -1$. Podemos intentar otra vez la rotación de p hacia la izquierda, aunque lamentablemente esto funciona, pues terminaremos con $\mathtt{fb}(q) = -2$, como se ve en la figura 11. Necesitamos rebalancear con más cuidado.

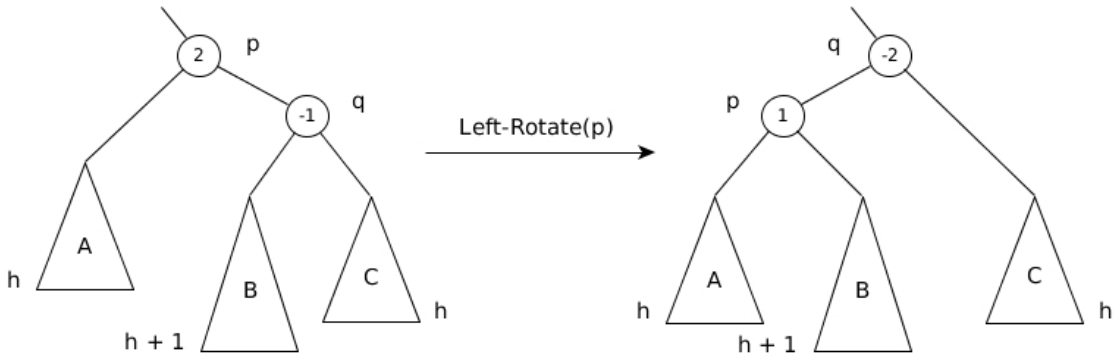


Figura 11: rotar p hacia la izquierda falla

Notemos que el subárbol B, que es el más pesado de los subárboles de q, es el que impide llegar al balance. Podemos intentar *desarmar* este árbol a través de otras rotaciones, antes de intentar balancear p. Como B tiene altura $h+1$, tiene al menos un nodo, con lo cual podemos llamar r a su raíz. La estructura que estamos considerando es la que muestra la figura 12.

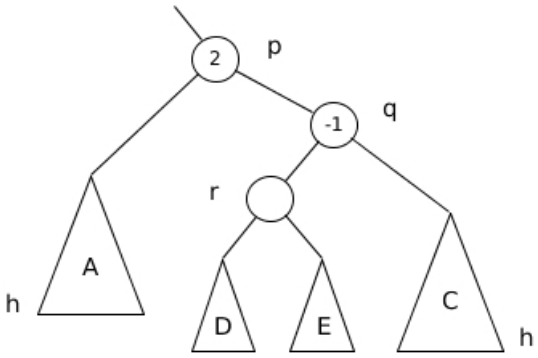


Figura 12: $\mathrm{fb}(p) = 2$ y $\mathrm{fb}(q) = -1$

Para distribuir el peso de B, necesitamos trasladar una parte de él hacia el subárbol derecho de q. Como podemos intuir a esta altura, es posible conseguir esto vía una RIGHT-ROTATE(q). Dejamos como tarea para el lector verificar que esta sola esta rotación puede no ser suficiente en ciertos casos, dependiendo del valor de $\mathrm{fb}(r)$. Si ejecutamos la LEFT-ROTATE(p) pendiente, ahora sí habremos restaurado el balanceo. El proceso se muestra en la figura 13.

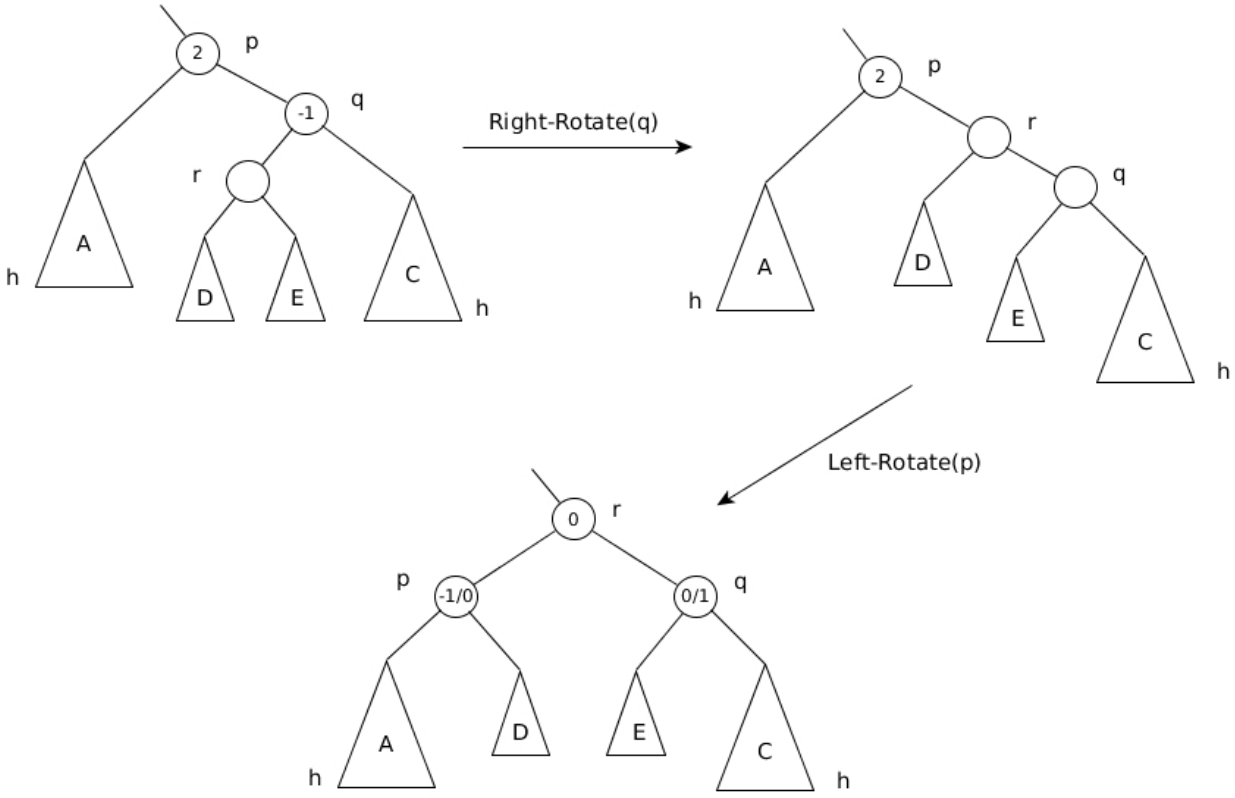


Figura 13: $\mathrm{fb}(p) = 2$ y $\mathrm{fb}(q) = -1$ **(caso C)**

Notemos que los factores de balanceo de p y q dependerán de las alturas de los subárboles D y E. Si miramos el árbol original, el subárbol con raíz en r tenía altura $h + 1$, con lo cual las alturas de D y E son algunas de $h - 1$ ó h.

Hasta ahora estuvimos analizando el caso $\mathrm{fb}(p) = 2$. El caso $\mathrm{fb}(p) = -2$ es completamente simétrico, y no repetiremos el mismo análisis detallado. Las figuras 14, 15 y 16 muestran las rotaciones necesarias para cada uno de los casos, que son, siguiendo el mismo orden que antes, análogos.

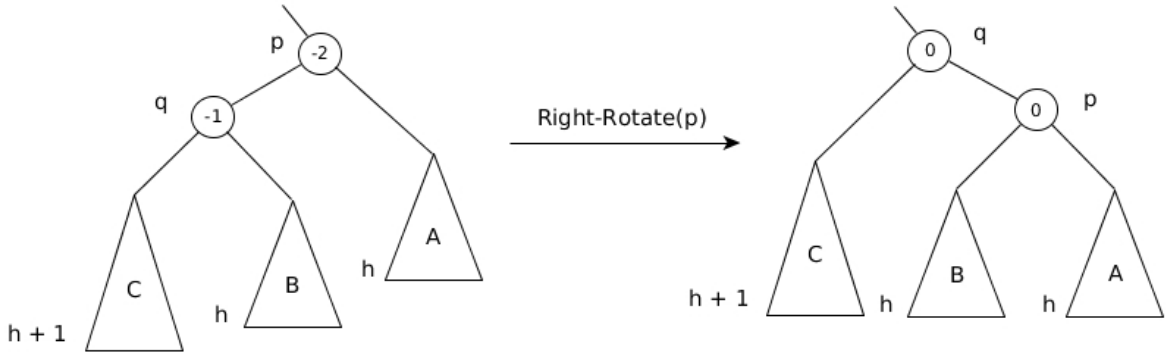


Figura 14: $fb(p) = -2$ y $fb(q) = -1$ **(caso D)**

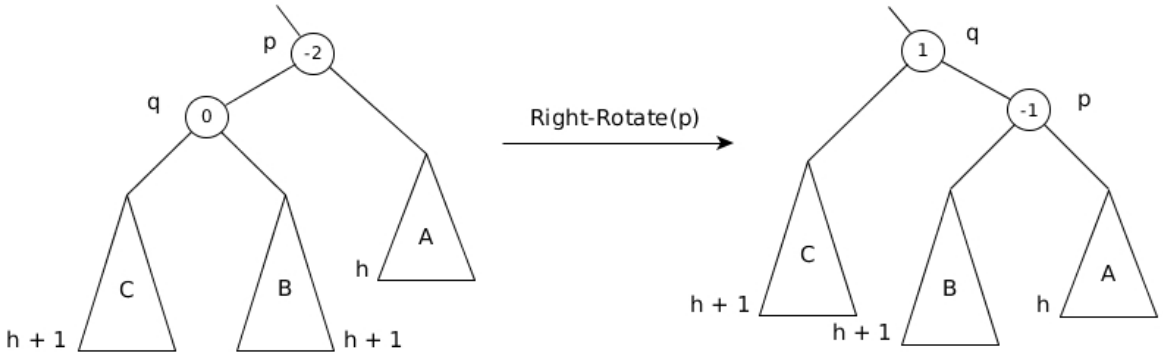


Figura 15: $fb(p) = -2$ y $fb(q) = 0$ **(caso E)**

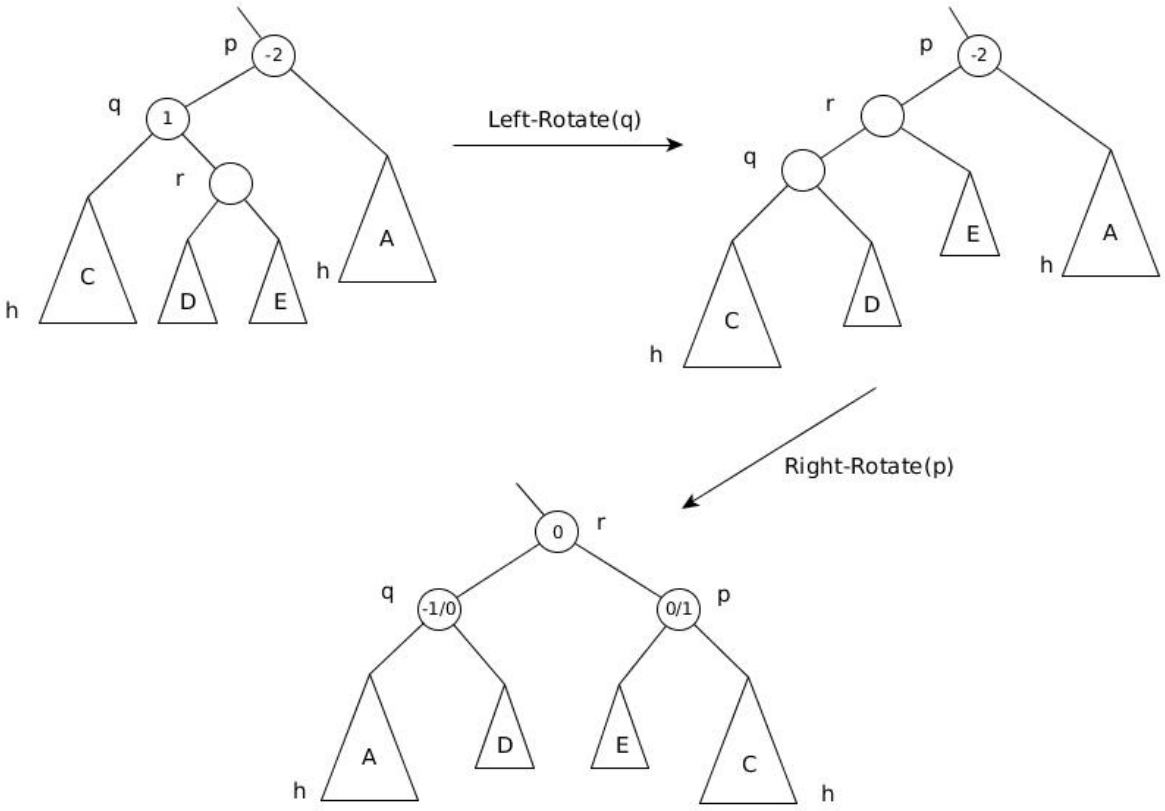


Figura 16: $fb(p) = -2$ y $fb(q) = 1$ **(caso F)**

3.3. Inserción en un AVL

Recordemos que la inserción en un AVL agrega un nuevo nodo como una hoja del árbol. Esto implica que sólo pueden cambiar los factores de balanceo de nodos que se encuentren en el camino que entre la raíz y el nodo agregado. Si logramos rebalancear cada factor desbalanceado a lo largo de este camino, habremos reestablecido el balance en todo el árbol. Esta idea motiva el algoritmo de inserción sobre AVLs.

```
1 AVL-Insert(T, k)
2 begin
3     Insertar k en T como en un ABB
4     Sea x el nodo agregado
5     AVL-Rebalance(T, x)
6 end
```

```
1 AVL-Rebalance(T, x)
2 begin
3     foreach p nodo en el camino entre x y la raíz de T do
4         if |fb(p)| > 1 then
5             Rebalancear el subárbol con raíz en p como en alguno de los casos A,
              ..., F
6         end
7     end
8 end
```

La complejidad del algoritmo es claramente proporcional a la altura del árbol, es decir, $O(\lg n)$.

La correctitud de este algoritmo se basa fuertemente en que cualquier desbalanceo que nos encontremos tendrá la forma de alguno de los clasificados anteriormente. Es evidente que el primero de todos que nos encontremos, recorriendo el camino desde abajo, será alguno de los vistos, puesto que el factor de balanceo no puede tener módulo mayor que 2, que a su vez se debe a que como sólo insertamos un nodo, entonces a lo sumo aumentamos la altura de una rama en uno. Pero luego de hacer el primer rebalanceo, ¿por qué los factores de nodos superiores no empeoran aún más? ¿No podría pasar que algún factor de balanceo pase a tener módulo mayor que 2?

Para responder negativamente a esto, debemos analizar cómo se propagan los cambios en la altura luego de efectuar el primer rebalanceo. Para esto, debemos tener en cuenta cómo era la altura del subárbol a rebalancear, en el instante previo a la inserción, y cuál es su altura luego del rebalanceo.

Hacemos el análisis para el rebalanceo del caso A, y dejamos el resto como ejercicio para el lector.

Teorema. *Durante una inserción, si el primer rebalanceo es de tipo A, entonces, luego del mismo, ningún nodo en el camino entre la raíz del subárbol rebalanceado y la raíz de todo el árbol tendrá un factor de balanceo de módulo mayor que dos.*

Demostración. Utilizamos la notación de la figura 9. Sea P el subárbol con raíz en p, previo a la inserción. Sea Q el resultado de insertar y efectuar el primer rebalanceo (de tipo A) en ese subárbol. En este caso, la inserción se debe haber producido necesariamente en el subárbol C (¿por qué?), y la altura de P es $h+2$. Como la altura de Q también es $h+2$, entonces ningún factor de balanceo de un nodo en el camino entre la raíz de Q y la raíz del árbol pudo haber cambiado respecto de su valor previo a la inserción. □

Observar que este resultado muestra algo más fuerte que lo que queríamos. Hemos probado que, en una inserción, si el primer rebalanceo es de tipo A, luego del mismo todo el árbol queda balanceado. Se puede demostrar que, luego de una inserción, los únicos casos de rebalanceo que se pueden dar son A, C, D y F, y que luego de realizar alguno de ellos, todos los factores de balanceo quedan reestablecidos.

3.4. Borrado en un AVL

Recordemos que los casos de borrado en un ABB son tres, que dependen de la cantidad de hijos del nodo a borrar. Llamemos z al nodo a borrar.

Si z no tiene hijos, entonces es borrado y el resto del árbol no cambia. Esto sólo puede generar variaciones de a lo sumo una unidad en los factores de balanceo de los nodos del camino entre el padre de z y la raíz.

Si z tiene un único hijo, entonces ese hijo es conectado con el padre de z. Al igual que antes, sólo pueden cambiar en uno, los factores de nodos entre el padre de z y la raíz.

Por último, si z tiene dos hijos, z es reemplazado por el nodo antecesor (o el sucesor) en el subárbol del cual es raíz. En otras palabras, se reemplaza z por el nodo de clave máxima en su subárbol izquierdo, al que llamamos x, y al único hijo de x se lo conecta con el padre de x, salvo cuando x es el hijo izquierdo de z, en cuyo caso el hijo de x conservará su relación con x. Por ende, sólo pueden cambiar los factores de balanceo de los nodos entre el antiguo padre de x (si x no es hijo de z) o bien x mismo (en caso contrario), y la raíz del árbol.

Este argumento muestra que, al igual que en la inserción, el borrado de nodos sólo modifica, a lo sumo en una unidad, los factores de balanceo de nodos entre cierto nodo particular (que depende del caso de borrado) y la raíz.

```
1 AVL-Delete(T, k)
2 begin
3     Borrar k de T como en un ABB
4     Sea y el primer nodo posiblemente desbalanceado
5     AVL-Rebalance(T, y)
6 end
```

El costo de este algoritmo también es proporcional a la altura del árbol, o sea, $O(\lg n)$.

Al igual que en la inserción, el algoritmo se basa en que nunca aparecerá un nodo desbalanceado con un factor de balanceo de módulo mayor que 2, pero no daremos una prueba formal de que esto efectivamente ocurre.

En el caso del borrado, cualquiera de los seis casos de rebalanceo puede darse, y el desbalance puede propagarse hacia nodos superiores, aún luego de haber efectuado una operación de rebalanceo. Por este motivo, no es posible cortar anticipadamente el recorrido hacia la raíz, luego del primer rebalanceo, como sí era posible en la inserción.

4. Conclusión

Organizar la información utilizando ABBs balanceados permite acceder a ella eficientemente. Este rápido acceso se torna imprescindible cuando el volúmen de información que debemos manejar es enorme. Sin embargo, lograr el balanceo no es una

tarea sencilla, y para conseguirlo necesitamos imponer ciertas restricciones que sean sostenibles a lo largo de las modificaciones que se efectúen sobre el árbol.

Hemos explorado distintos criterios para forzar el balanceo de un árbol, observando que algunos de ellos son demasiado rígidos como para poder ser mantenidos. El balanceo en altura probó ser efectivo en ese sentido. Queda claro que podría haber otras alternativas para llegar al preciado balanceo, y de hecho las hay, como por ejemplo aquellas utilizadas por los árboles red-black, árboles B o árboles 2-3-4. Si bien todas estas estructuras cumplen con el objetivo de almacenar en orden y en forma balanceada toda la información, no todos son igualmente complejos: la distinta rigidez entre los criterios de balanceo que utilizan hace que la dificultad de los algoritmos de cada estructura y, a nivel de eficiencia, la cantidad de rebalanceos que necesitan, varíe entre ellas. El balanceo de árboles, por lo tanto, no es sólo un concepto utilizado por los árboles AVL, sino que tiene la jerarquía de un campo de estudio en computación.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.