Arboles binarios

¿De qué se trata?
Podría considerarse como una extensión del algoritmo de búsqueda binaria.
Es un árbol que cumple con las siguientes propiedades:

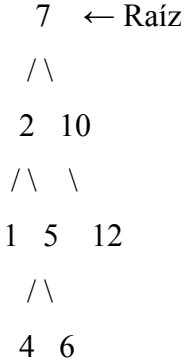
- Cada nodo interno tiene como máximo 2 hijos. Si hablamos de **árboles binarios completos**, entonces cada nodo interno tiene **exactamente 2 hijos**.
- Los hijos de un nodo son un par ordenado.
- Los hijos se denominan **izquierdo** y **derecho**.

Operaciones

Al ser utilizados para buscar, es posible definir las funciones buscarMin y buscarMax.
En la primera de estas, se parte desde la raíz del árbol, y se viaja continuamente por la izquierda del árbol, hasta que se llegue al valor más mínimo, que es el que se encuentra a la izquierda del todo.
En el caso de buscarMax, es análogo, pero en vez de ir a la izquierda se va a la derecha.
Estas operaciones tienen un costo general de **O(Log(n))**, pero en el peor de los casos podríamos estar hablando de **O(n)**, osea **lineal**.

Paréntesis - ¿Cuál es la raíz de un árbol binario (BST)?

Buena pregunta, pues la **raíz** de un árbol se trata del **nodo superior** del mismo. Es aquel que **no tiene padre**, y es el punto de **entrada por el cual se accede a todos los demás nodos**. Si trazamos su recorrido hacia la izquierda, todos los nodos que derivan son menores que la raíz, mientras que si lo hacemos hacia la derecha, todos los nodos son mayores a esta.



Operación complicada - Eliminar

¿Por qué es difícil? - Se pueden dar 3 casos.

. Nodo hoja (sin hijos): Se elimina directamente.

. Nodo con un hijo: El padre del nodo eliminado se enlaza con su único hijo.

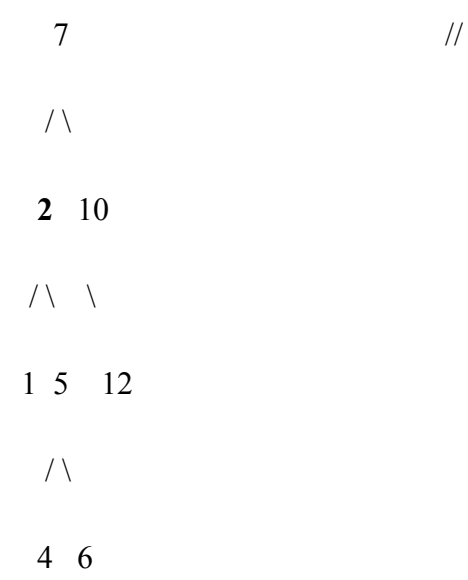
. Nodo con dos hijos: Requiere un reemplazo adecuado para mantener la propiedad del BST. Aquí radica la complejidad.

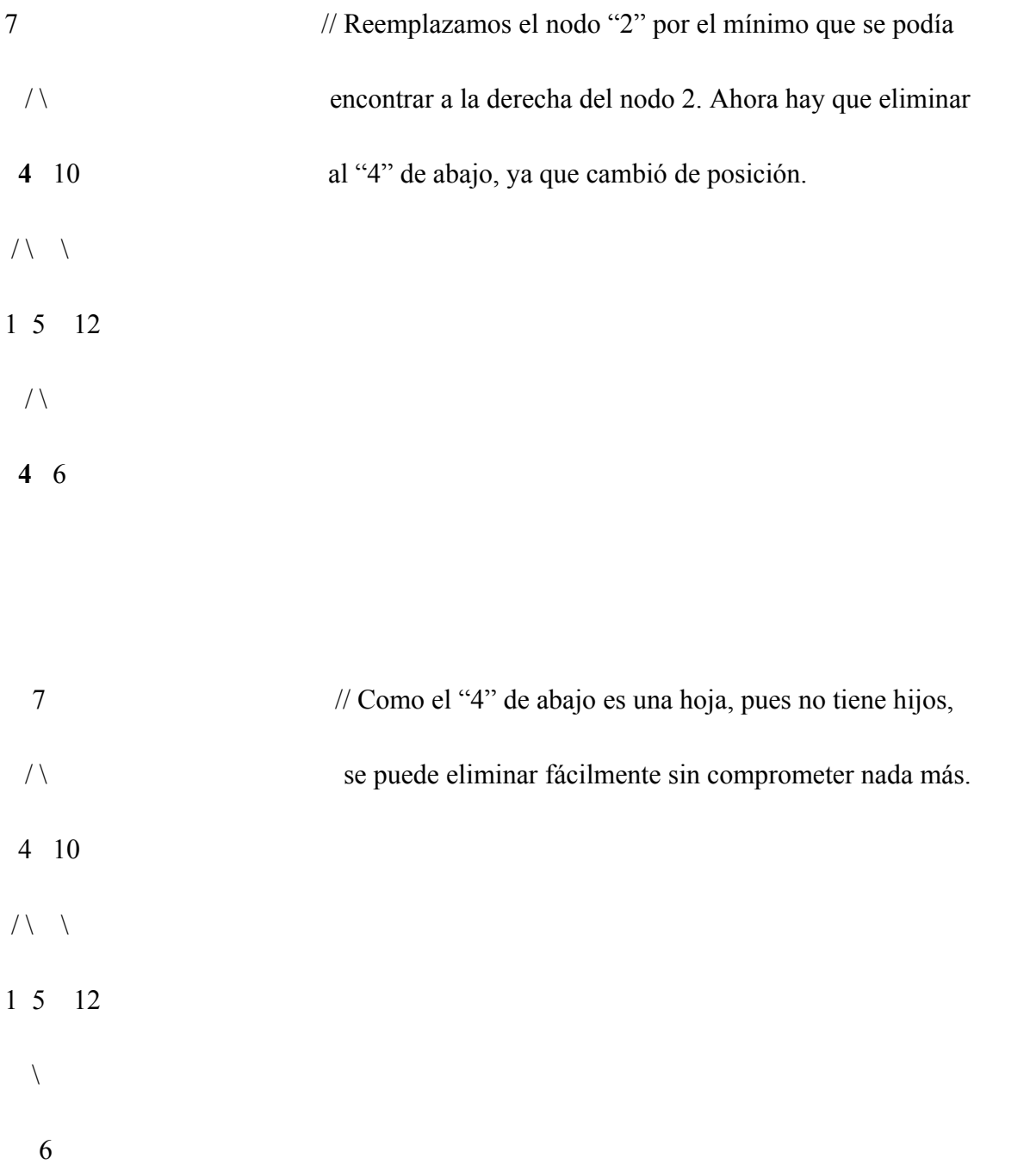
Si eliminamos un nodo con dos hijos, dejamos dos subárboles "huérfanos". Para resolver esto, se busca un nodo que pueda reemplazar al eliminado sin alterar el orden:

Sucesor inmediato: El mínimo valor del subárbol derecho del nodo a eliminar.

Predecesor inmediato: El máximo valor del subárbol izquierdo (también válido, pero menos común).

Caso:




7
/ \
4 10
/ \ \
1 5 12
/ \
4 6
// Reemplazamos el nodo “2” por el mínimo que se podía encontrar a la derecha del nodo 2. Ahora hay que eliminar al “4” de abajo, ya que cambió de posición.
7
/ \
4 10
/ \ \
1 5 12
\
6
// Como el “4” de abajo es una hoja, pues no tiene hijos, se puede eliminar fácilmente sin comprometer nada más.

18.2

Explicación de la búsqueda por posición en árboles binarios de búsqueda

Objetivo

Encontrar el K-ésimo menor elemento de un árbol binario de búsqueda (ABB) de manera eficiente, utilizando información sobre el tamaño de los subárboles.

Conceptos clave

Tamaño de un nodo:

- Es el número total de nodos en su subárbol, incluyéndose a sí mismo.

Ejemplo: Si un nodo tiene 3 nodos en su subárbol izquierdo y 2 en el derecho, su tamaño es 1+3+2=6.

K-ésimo menor elemento:

- El elemento que ocuparía la posición K si todos los elementos del árbol estuvieran ordenados de menor a mayor.

Algoritmo para buscar el K-ésimo elemento

Dado un nodo raíz t con tamaño $S = S_L + S_R + 1$ (donde S_L es el tamaño del subárbol izquierdo y S_R el del derecho):

Caso 1:

$K = S_L + 1$

El K-ésimo elemento es la raíz del subárbol actual.

Ejemplo: Si $K = 4$ y $S_L = 3$, entonces $K = 3 + 1 = 4 \rightarrow$ la raíz es el 4º elemento.

Caso 2:

$K \leq S_L$

El K-ésimo elemento está en el subárbol izquierdo.

Ejemplo: Si $K = 2$ y $S_L = 5$, buscamos el 2º elemento en el subárbol izquierdo.

Caso 3:

K > SL+1

El K-ésimo elemento está en el subárbol derecho.

Se busca el (K−SL−1)-ésimo elemento en el subárbol derecho.

Ejemplo: Si K = 7 y SL = 3, buscamos el 7−3−1= 3-er elemento en el subárbol derecho.

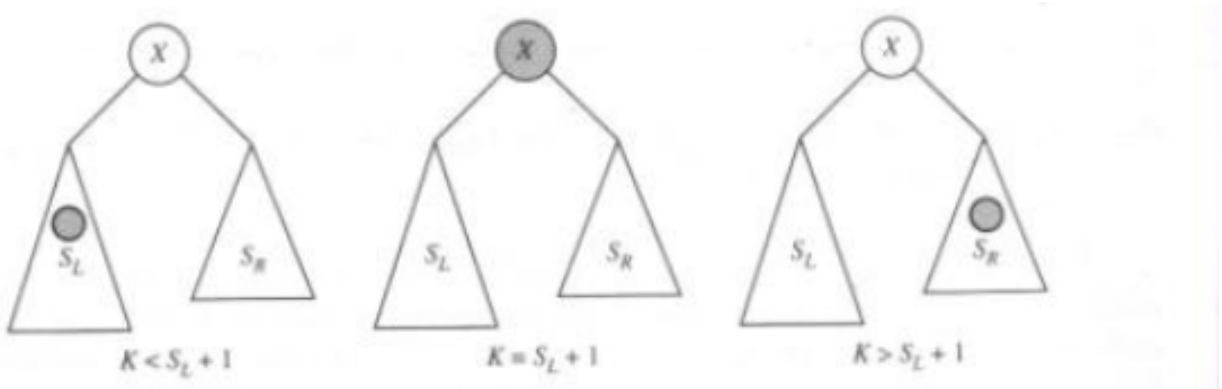


Figura 18.13 Empleo del atributo `tamanyo` para implementar el método `buscarKesimo`.

Mantenimiento del tamaño

Para que el algoritmo funcione, el tamaño de cada nodo debe actualizarse durante las operaciones de inserción y eliminación:

Inserción:

Cada nodo en el camino desde la raíz hasta el punto de inserción incrementa su tamaño en 1.

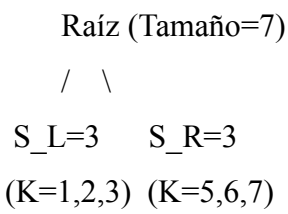
El nuevo nodo insertado tiene tamaño 1.

Eliminación:

Cada nodo en el camino desde la raíz hasta el nodo eliminado decrementa su tamaño en 1.

En operaciones como eliminarMin, se reduce el tamaño de los nodos afectados.

Ejemplo visual



Si K=4, la raíz es el 4º elemento.

Si K=2, se busca en el subárbol izquierdo.

Si K=5, se busca el 5−3−1=1-er elemento en el subárbol derecho.

18.3 Análisis de las operaciones de los árboles binarios de búsqueda

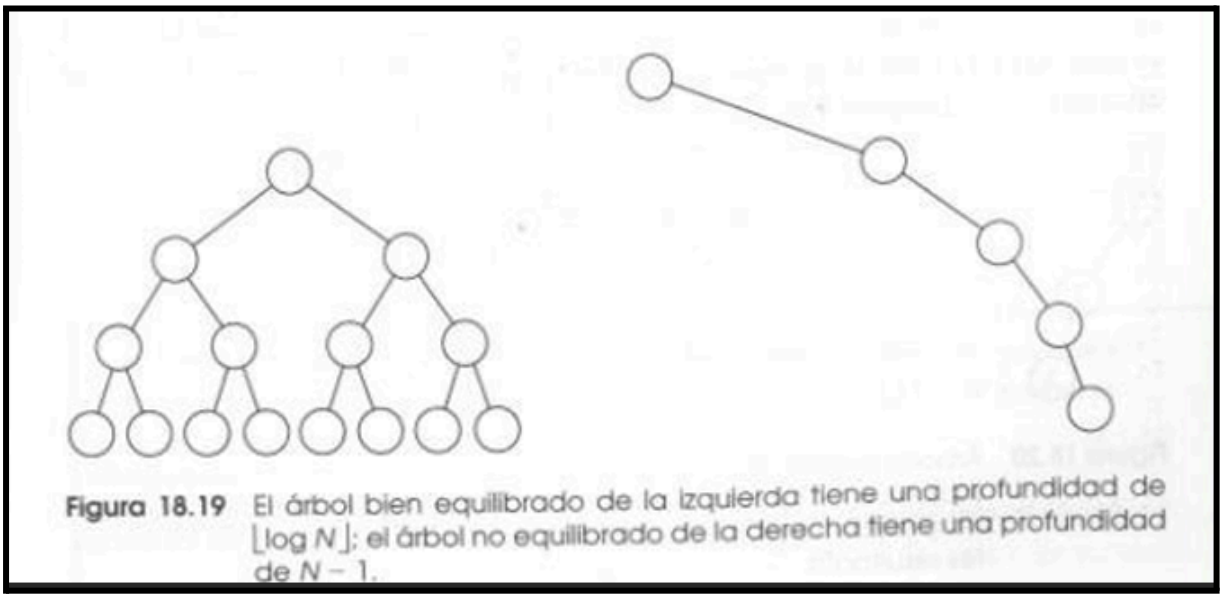


Figura 18.19 El árbol bien equilibrado de la izquierda tiene una profundidad de $\lfloor \log N \rfloor$; el árbol no equilibrado de la derecha tiene una profundidad de $N - 1$.

Es correcto decir que el árbol no equilibrado de la derecha funciona como una LinkedList convencional, que para acceder a cualquiera de sus elementos, sin ser el primero, tiene una complejidad temporal en el peor caso de O(n).

En un árbol binario, el costo de cada una de las operaciones (insertar, buscar y eliminar) es proporcional a la cantidad de nodos consultados en el proceso. Es por eso, que si el árbol está bien equilibrado, la complejidad es de Log(n). Si no lo está, como en el segundo caso de la foto, la complejidad es de O(n).

Podríamos decir, que en un árbol binario equilibrado, tanto la profundidad de los nodos, como el coste de operaciones es de Log(n) (figura.1 18.19). Mientras que en uno degenerado, como el de la derecha, en ambos casos es de O(n).

Caso promedio

¿Hay un caso medio? Según Allan en el libro, la profundidad media de los nodos de un árbol binario de búsqueda es logarítmica, asumiendo que cada árbol se genera como resultado de secuencias aleatorias de inserción; osea, sin operaciones de tipo *eliminar*.
Es decir, que con inserciones aleatorias, la profundidad media es de Log(n).

Longitud del camino interno (IPL):
Suma de las profundidades de todos los nodos.
Media de esta suma: *1.38 x N x Log(N)*, lo cual es similar al análisis de quicksort que se cita en el libro de Allan.
Costo de búsqueda exitosa: O(Log N).

DEFINICIÓN: La *longitud del camino interno* de un árbol binario es igual a la suma de las profundidades de sus nodos.

Peor caso: inserción en orden ascendente o descendente, osea, tener un árbol degenerado como el de la figura 18.19, en ese caso el peor caso es de O(N).

Longitud del camino interno y externo

. Del IPL ya hablamos, mide la suma de la profundidad de todos los nodos. El mismo, está relacionado con el coste de búsquedas exitosas.

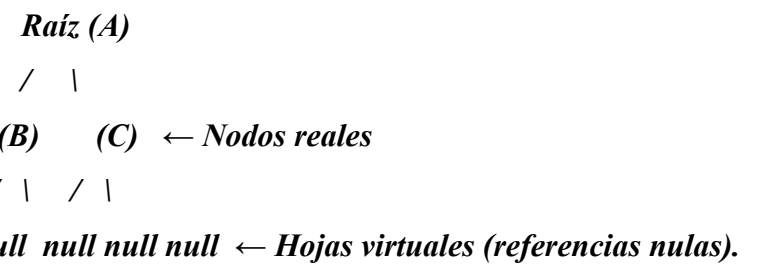
. Por otro lado, está la longitud del camino externo (EPL), que mide la suma de las profundidades de todas las referencias nulas (hojas virtuales).
. Tiene una relación con IPL. Ya que *EPL = IPL + 2N.*
. Está relacionado con el costo de inserciones y búsquedas sin éxito.

¿Qué son las "Hojas virtuales"?
. En un árbol binario, cada nodo tiene dos hijos (zurdo y diestro).
. Los nodos *hoja reales*, son aquellos sin *hijos reales.* Pero técnicamente tienen dos referencias nulas (zurda = null; diestra = null).
Estas referencias nulas, se consideran **hojas virtuales** en el análisis teórico.

Propósito de las hojas virtuales con el EPL:

La profundidad del camino externo suma las profundidades de todas las hojas virtuales. Esto ayuda a medir el costo de:

- *Búsqueda sin éxito*: Cuando se busca un elemento que no está en el árbol, la búsqueda termina en una hoja virtual, puesto a que vendría a ser el final del camino.
- *Inserciones*: Una inserción siempre ocurre en una hoja virtual (se reemplaza null por un nuevo nodo).



Relación con el IPL

IPL = suma de la profundidad de los nodos reales

En el ejemplo de arriba, el IPL: A(profundidad = 0) + B(1) + C(1) = 2

Por el teorema falopero, se sabe que EPL = IPL + 2N (N = Número de nodos reales).

EPL = 2 + 2 x 3 = **8**

Esto tiene sentido, ya que hay 4 hojas virtuales, y cada una de ellas tiene una profundidad de 2, => 4 x 2 = **8** = **EPL**.

lqqd.

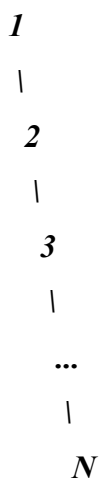
Problema garrafal con la operación de *eliminar*

En nuestro árbol está todo correcto y precioso, con un tiempo de ejecución promedio de Log(n), siempre que el árbol está ordenado aleatoriamente. Pero qué pasa si aplicamos la operación (infame) de *eliminar*?

Como es más que sabido, al eliminar un nodo que tiene dos hijos, el mismo se reemplaza por el mínimo del subárbol derecho. En el caso de que las eliminaciones no sean aleatorias, este método puede desequilibrar el árbol a la izquierda.

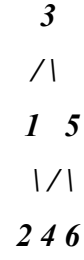
Con operaciones aleatorias de inserción y eliminación, el árbol tiende a mantenerse equilibrado. Mientras que en secuencias ordenadas, el desequilibrio es inevitable. Esto provoca, que en el peor caso, el costo de las operaciones (buscar, insertar y eliminar) sea de O(n). También implica que la profundidad del árbol en el peor caso sea de O(n).

Un mamarracho que daría ejemplo de un árbol desequilibrado, sería insertar elementos en orden ascendente:



Aca hay un claro problema, pues el número mayor siempre se inserta a la derecha, formando una especie de lista enlazada, la cual no hace falta decir que para recorrerla tenemos que pasar todos sus elementos.

Mientras en un caso equilibrado (con los nodos insertados NO EN ORDEN):



Costo de búsqueda: Log(n).
Profundidad: Log(n) (como un balance aproximado).

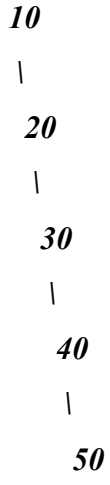
Propondremos un ejemplo numérico

Árbol desequilibrado (elementos a insertar = 10, 20, 30, 40, 50).

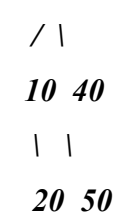
Se nos genera un mamarracho.

Profundidad: igual a N.

Costo de buscar, por ejemplo al nodo = 50: 5, pues hay cinco nodos a recorrer = O(n).



Árbol equilibrado con inserción aleatoria (mismos nodos a insertar)



Profundidad: 2 —-> Para N = 5, la profundidad promedio es Log5 =~ 2,32…

Costo de buscar 50: Hay que recorrer 3 nodos, no 5 como antes, lo que hace que el costo promedio sea de Log(n).

¡APARECE FIBO! Arboles binarios AVL

DEFINICIÓN: Un *árbol AVL* es un árbol binario de búsqueda con una propiedad adicional de equilibrio, según la cual, las alturas de los hijos derecho e izquierdo sólo pueden diferir, a lo sumo, en una unidad. Como es usual, tomamos -1 como la altura del árbol vacío.

Se podría decir que es un árbol binario de búsqueda auto-balanceado.
Esto implica que, para cada nodo, la diferencia de altura entre su subárbol izquierdo y derecho es <= 1.
Si se desequilibra (diferencia > 1), entonces se aplican rotaciones para rebalancearlo.

Cual es su objetivo? - Garantizar mantener una altura O(Log(n)), garantizando operaciones eficientes (O(Log(n))), aun en el peor de los casos. Suena una joyeta.

Aspecto interesante que se menciona en el libro del buen amigo Allan

La condición de equilibrio AVL implica que el árbol tiene siempre una profundidad logarítmica. Para demostrarlo, basta mostrar que un árbol de altura H debe tener, por lo menos, C^H nodos, donde C es una constante mayor que 1. En otras palabras, el número mínimo de nodos de un árbol crece exponencialmente con la altura. Así, la profundidad máxima de un árbol con N elementos es $\log_C N$. El Teorema 18.3 muestra que todo árbol AVL de altura H tiene *muchos* nodos.

VER FOTOS DE LAS SIGUIENTES 3 PÁGINAS
LUEGO RETOMAMOS CON ROTACIONES

Arboles Binarios AVL

El número mínimo de nodos $S(H)$ para una altura H crece exponencialmente ($S(H) \geq c^H$)
Esto garantiza que la profundidad máxima sea $O(Log(n))$

$S(H)$ sigue la recurrencia:
$S(H) = S(H-1) + S(H-2) + 1$ //
con $S(0) = 1$ y $S(1) = 2$

Esta secuencia es similar a los números de fibonacci, pero con un termino adicional (+1)

$$S(H) = F_{H+3} - 1$$

→ Donde F_i es el i-ésimo numero de fibonacci

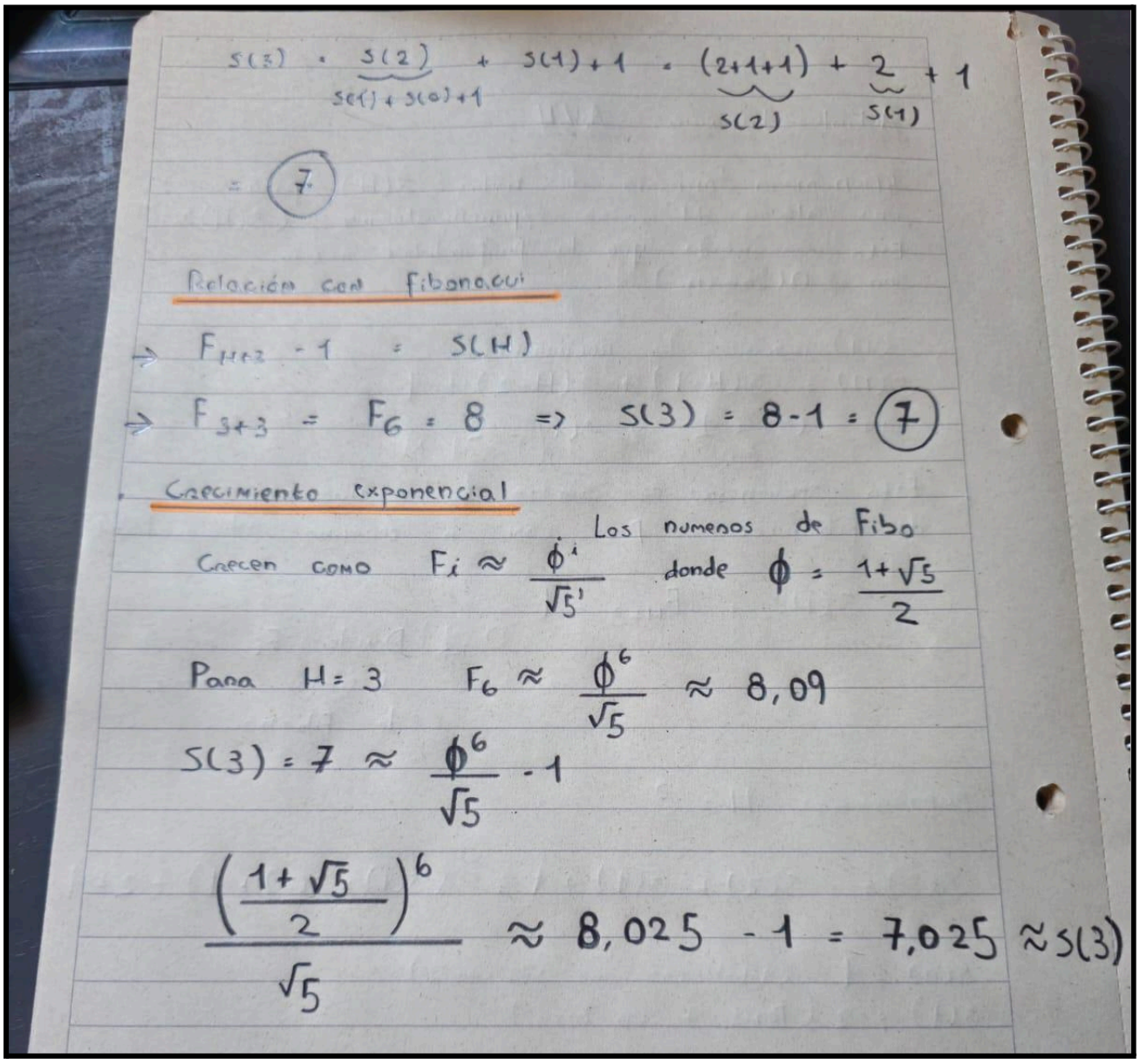
Supongamos $H = 3$

$$S(3) = S(2) + S(1) + 1 = (S(1) + S(0) + 1) + 2 + 1$$

$S(0) = 1$ (Arbol con una sola raíz)
$S(1) = 2$ (Raíz + un hijo)

$S(3) = S(2) + S(1) + 1$
$S(2) = S(1) + S(0) + 1 = 2 + 1 + 1 = 4$

$S(1) = 2$
$S(2) = 4$



$S(3) = \underbrace{S(2)}_{S(1)+S(0)+1} + S(1) + 1 = \underbrace{(2+1+1)}_{S(2)} + \underbrace{2}_{S(1)} + 1$

$= (7)$

Relación con Fibonacci

→ $F_{H+2} - 1 = S(H)$

→ $F_{3+3} = F_6 = 8 \Rightarrow S(3) = 8 - 1 = (7)$

Crecimiento exponencial

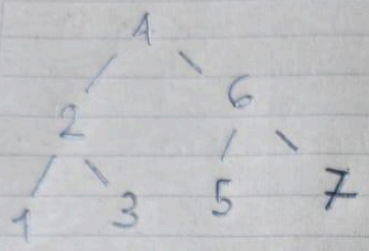
Crecen como $F_i \approx \frac{\phi^i}{\sqrt{5}}$ Los numeros de Fibo donde $\phi = \frac{1+\sqrt{5}}{2}$

Para $H = 3$ $F_6 \approx \frac{\phi^6}{\sqrt{5}} \approx 8,09$

$S(3) = 7 \approx \frac{\phi^6}{\sqrt{5}} - 1$

$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^6}{\sqrt{5}} \approx 8,025 - 1 = 7,025 \approx S(3)$

Árbol AVL mínimo para H = 3



Altura : 3

Nodos : 7

Cumple que $7 \geq C^3$

$C \approx 1,913$ $(\sqrt[3]{7})$

- La condición AVL fuerza que el número de nodos crezca exponencialmente con la altura, lo que garantiza que la altura crezca logaritmicamente con N. Esto asegura operaciones eficientes (O(Log(n)) incluso en el peor caso.

Rotación simple

Se aplica en casos donde el desequilibrio ocurre en una rama externa (izq - izq) o (der - der).

Antes de la rotación:

```
      k2 (desequilibrado)
     /
   k1
  /
A
```

Después de rotación simple derecha:

```
      k1
     / \
  A   k2
```

Pasos: k1 se convierte en la nueva raíz y k2 se desplaza a la derecha de k1.
El subárbol derecho de k1 (si existiese) se convierte en el subárbol izquierdo de k2.

Vamos a proponer un ejemplo con números. k1 = 8 y k2 = 4.

```
    8 (desequilibrado)  // Un mamarracho.
   /
  4
 /
1
```

Va a pasar a estar asi:

```
   4
  / \
1   8
```

Es más que evidente que, si 4 tuviese algún hijo derecho, ese mismo nodo pasaría a ser hijo izquierdo de 8. Es decir que 4.derecho = 8.izquierdo, una vez hecha la rotación.

Código que repasa la solución

```
static NodoBinario comBigDerecho(NodoBinario k1) {
   NodoBinario k2 = k1.izquierdo; // k2 es el hijo izquierdo de k1 (ej: 4)
   k1.izquierdo = k2.derecho;     // El subárbol derecho de k2 se convierte en el izquierdo de k1
   k2.derecho = k1;               // k1 (ej: 8) se convierte en hijo derecho de k2
   return k2;                     // k2 (ej: 4) es la nueva raíz
}
```

Rotación doble

1. Una inserción en el subárbol izquierdo del hijo izquierdo de *X*.
2. Una inserción en el subárbol derecho del hijo izquierdo de *X*.
3. Una inserción en el subárbol izquierdo del hijo derecho de *X*.
4. Una inserción en el subárbol derecho del hijo derecho de *X*.

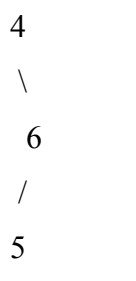
Esos son los 4 casos que se pueden dar, la rotación simple satisface el caso 1 y 4, pero no puede decir lo mismo con los casos 2 y 3.
Para estos últimos es que existe la rotación doble. Esta se utiliza cuando el desequilibrio está en una rama interna (izquierda - derecha o derecha - izquierda).

Caso posible (LR)

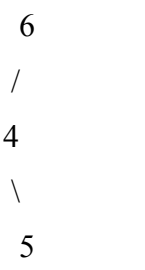
```
      8 (desequilibrado: altura izquierda=2, derecha=0)
     /
    4
     \
      6
     /
    5
```

Tenemos en ese árbol un mamarracho, puesto a que la altura del subárbol izquierdo de 8 = 4 es 2, mientras que la del subárbol derecho de 8 es 0.
Vamos a proceder a hacer dos rotaciones, primero una RR (derecha - derecha), cambiando de lugar el 4 con el 6.

ANTES:



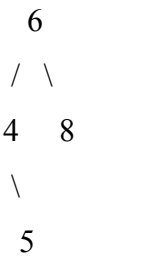
DESPUÉS:



Ahora hacemos la segunda rotación simple, esta vez una LL (izquierda - izquierda), cambiando de lugar el 6 con el 8, el cual es la raíz del árbol inicial.

ANTES:

DESPUÉS:



Es más que claro, que si 6 tuviera un subárbol derecho, es decir, algún hijo derecho, este pasaría a ser subárbol izquierdo de 8.

De este modo, se cumple con la condición de que la altura entre subárboles no puede ser >= 1 entre ellos.

¿Por qué se hacen dos rotaciones?

. La primera convierte el caso complejo (LR o RL) en un caso simple (LL o RR).

. La segunda aplica la rotación simple correspondiente para balancear.

. Eficiencia: Ambas rotaciones se ejecutan en un tiempo de O(1).

Se podría decir, que una rotación doble son dos rotaciones simples consecutivas.

Código de cómo se vería la solución en Java

```
static Nodo dobleRotacionIzquierdaDerecha(Nodo k3) {
   k3.izquierdo = rotacionIzquierda(k3.izquierdo); // Primera rotación (RR)
   return rotacionDerecha(k3);                    // Segunda rotación (LL)
}
```

Caso 2: RL

```
      4 (desequilibrado: altura derecha=2, izquierda=0)
       \
        8
       /
      6
       \
        7
```

¿Qué hay que hacer acá? - Bueno, primero habría que tomar el subárbol derecho de 4, que es 8. Este tiene una altura de 2. Al 8 lo cambiamos por el 6, de modo que el 8 ahora pasa a estar a la derecha del 6.

Luego aplicamos rotación simple para que nos quede así:

```
      6
     / \
    4   8
       /
      7
```

Código de cómo se vería la solución en Java

```
static Nodo dobleRotacionDerechaIzquierda(Nodo k1) {
   k1.derecho = rotacionDerecha(k1.derecho); // Primera rotación (LL)
   return rotacionIzquierda(k1);             // Segunda rotación (RR)
}
```