

UNIVERSIDAD NACIONAL DE INGENIERÍA

Estructuras de Datos

Árboles AVL

Rotaciones, altura y reparar un poco en cada operación

Gerson Garrido

Semana 11

Hoy

1. El problema: una altura de n
2. La condición AVL: en *cada* nodo
3. Por qué la altura es O(log n): Fibonacci
4. Rotación: cambia la forma y conserva el orden
5. Los cuatro casos: simple y doble
6. Insertar: basta una reparación
7. Borrar: pueden hacer falta varias
8. La invariante en las pruebas
9. Cuánto se gana, medido

> **Lectura**
>
> Apuntes de la semana 11, que son el texto principal. Código en `code/week-11`.

SECCIÓN 01

El problema

01

Lo que quedó pendiente

- Un árbol de búsqueda cuesta lo que mide su **altura**.

Lo que quedó pendiente

- Un árbol de búsqueda cuesta lo que mide su **altura**.
- Con claves ordenadas, la altura es $n - 1$: una lista.

Lo que quedó pendiente

- Un árbol de búsqueda cuesta lo que mide su **altura**.
- Con claves ordenadas, la altura es $n - 1$: una lista.
- `palabras.txt` viene ordenado: estimamos cerca de una hora sin balance.

Lo que quedó pendiente

- Un árbol de búsqueda cuesta lo que mide su **altura**.
- Con claves ordenadas, la altura es $n - 1$: una lista.
- `palabras.txt` viene ordenado: estimamos cerca de una hora sin balance.
- `std::map` recibió el mismo archivo, y tardó menos de un segundo.

Lo que quedó pendiente

- Un árbol de búsqueda cuesta lo que mide su **altura**.
- Con claves ordenadas, la altura es $n - 1$: una lista.
- `palabras.txt` viene ordenado: estimamos cerca de una hora sin balance.
- `std::map` recibió el mismo archivo, y tardó menos de un segundo.

La idea de hoy

Después de cada operación, revisar el camino recorrido, y arreglar la forma donde se haya desequilibrado demasiado. Con una operación local y barata: la **rotación**.

¿Por qué no exigir la altura mínima?

- Con $n = 2^k - 1$ claves, la única forma de altura mínima es el árbol perfecto.

¿Por qué no exigir la altura mínima?

- Con $n = 2^k - 1$ claves, la única forma de altura mínima es el árbol perfecto.
- Inserten una clave menor que todas, y borren la mayor.

¿Por qué no exigir la altura mínima?

- Con $n = 2^k - 1$ claves, la única forma de altura mínima es el árbol perfecto.
- Inserten una clave menor que todas, y borren la mayor.
- Otra vez $2^k - 1$ claves, y otra vez debe ser el perfecto.

¿Por qué no exigir la altura mínima?

- Con $n = 2^k - 1$ claves, la única forma de altura mínima es el árbol perfecto.
- Inserten una clave menor que todas, y borren la mayor.
- Otra vez $2^k - 1$ claves, y otra vez debe ser el perfecto.
- Pero cada posición del recorrido tiene ahora la clave de su vecina: $\Theta(n)$ trabajo.

¿Por qué no exigir la altura mínima?

- Con $n = 2^k - 1$ claves, la única forma de altura mínima es el árbol perfecto.
- Inserten una clave menor que todas, y borren la mayor.
- Otra vez $2^k - 1$ claves, y otra vez debe ser el perfecto.
- Pero cada posición del recorrido tiene ahora la clave de su vecina: $\Theta(n)$ trabajo.

Pedir menos

Una condición:

1. que dé altura O(log n);
2. que se restaure tocando pocos nodos, solo en el camino recorrido.

SECCIÓN 02

La condición AVL

02



La condición AVL

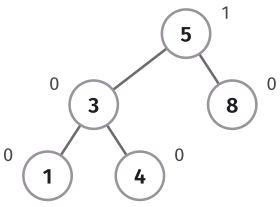
Árbol AVL

Balance de un nodo: altura del subárbol izquierdo menos altura del derecho. Un AVL es un árbol binario de búsqueda en el que **cada nodo** tiene balance -1, 0 o 1.

Altura: un nodo solo tiene 0, y un subárbol vacío tiene -1. Sin ese -1 no salen las cuentas.

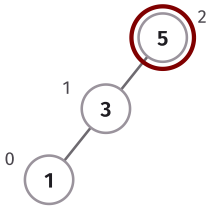
Tres árboles, con su balance

JUNTO A CADA NODO, SU BALANCE



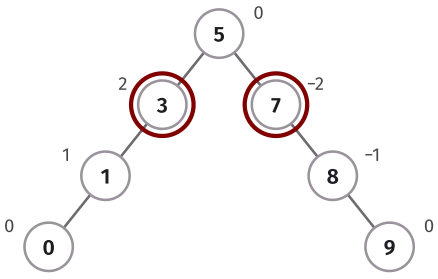
5(3(1,4),8)

es AVL



5(3(1,_),_)

no es AVL



5(3(1(0,_),_),7(_,8(_,9)))

no es AVL

En *cada* nodo, no solo en la raíz

- En el tercer árbol, la raíz tiene balance 0: sus dos lados miden lo mismo.

En *cada* nodo, no solo en la raíz

- En el tercer árbol, la raíz tiene balance 0: sus dos lados miden lo mismo.
- Pero el 3 tiene balance 2, y el 7 tiene -2. No es AVL.

En *cada* nodo, no solo en la raíz

- En el tercer árbol, la raíz tiene balance 0: sus dos lados miden lo mismo.
- Pero el 3 tiene balance 2, y el 7 tiene -2. No es AVL.
- Si la condición fuera solo en la raíz, dos cadenas largas la cumplirían, con altura de unos $n/2$.

En *cada* nodo, no solo en la raíz

- En el tercer árbol, la raíz tiene balance 0: sus dos lados miden lo mismo.
- Pero el 3 tiene balance 2, y el 7 tiene -2. No es AVL.
- Si la condición fuera solo en la raíz, dos cadenas largas la cumplirían, con altura de unos $n/2$.

Error común

`check_invariants` tiene que revisar **todos** los nodos, como con el orden.

Cada nodo guarda su altura

avl_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5      int height = 0;
6  };
```

avl_set.h

```
1  static void update_height(Node* node) {
2      node->height = 1 + std::max(height_of(node->left),
           height_of(node->right));
3  }
```

- Calcularla bajando costaría Θ(tamaño del subárbol).
- Un puntero nulo cuenta como altura -1: de eso se encarga `height_of`.

Cada nodo guarda su altura

avl_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5      int height = 0;
6  };
```

avl_set.h

```
1  static void update_height(Node* node) {
2      node->height = 1 + std::max(height_of(node->left),
         height_of(node->right));
3  }
```

- Calcularla bajando costaría Θ(tamaño del subárbol).
- Un puntero nulo cuenta como altura -1: de eso se encarga `height_of`.
- `update_height` confía en lo que guardan los hijos: $O(1)$.

Cada nodo guarda su altura

avl_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5      int height = 0;
6  };
```

avl_set.h

```
1  static void update_height(Node* node) {
2      node->height = 1 + std::max(height_of(node->left),
         height_of(node->right));
3  }
```

- Calcularla bajando costaría Θ(tamaño del subárbol).
- Un puntero nulo cuenta como altura -1: de eso se encarga `height_of`.
- `update_height` confía en lo que guardan los hijos: O(1).
- Por eso **el orden importa**: primero los hijos.

Cada nodo guarda su altura

avl_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5      int height = 0;
6  };
```

avl_set.h

```
1  static void update_height(Node* node) {
2      node->height = 1 + std::max(height_of(node->left),
           height_of(node->right));
3  }
```

- Calcularla bajando costaría Θ(tamaño del subárbol).
- Un puntero nulo cuenta como altura -1: de eso se encarga `height_of`.
- `update_height` confía en lo que guardan los hijos: O(1).
- Por eso **el orden importa**: primero los hijos.
- La altura guardada es una invariante más.

SECCIÓN 03

Por qué la altura es O(log n)

03

Darle la vuelta a la pregunta

La pregunta

Para una altura h, ¿cuál es la **menor** cantidad de nodos $N(h)$ de un AVL?

Darle la vuelta a la pregunta

La pregunta

Para una altura h, ¿cuál es la **menor** cantidad de nodos $N(h)$ de un AVL?

- $N(0) = 1$, y $N(1) = 2$.

Darle la vuelta a la pregunta

La pregunta

Para una altura h, ¿cuál es la **menor** cantidad de nodos $N(h)$ de un AVL?

- $N(0) = 1$, y $N(1) = 2$.
- Con $h \geq 2$: una raíz, un subárbol de altura $h - 1$, y el otro lo más bajo que se permita: $h - 2$.

Darle la vuelta a la pregunta

La pregunta

Para una altura h, ¿cuál es la **menor** cantidad de nodos $N(h)$ de un AVL?

- $N(0) = 1$, y $N(1) = 2$.
- Con $h \geq 2$: una raíz, un subárbol de altura $h - 1$, y el otro lo más bajo que se permita: $h - 2$.
- Y cada uno, con la menor cantidad posible de nodos.

Darle la vuelta a la pregunta

La pregunta

Para una altura h, ¿cuál es la **menor** cantidad de nodos $N(h)$ de un AVL?

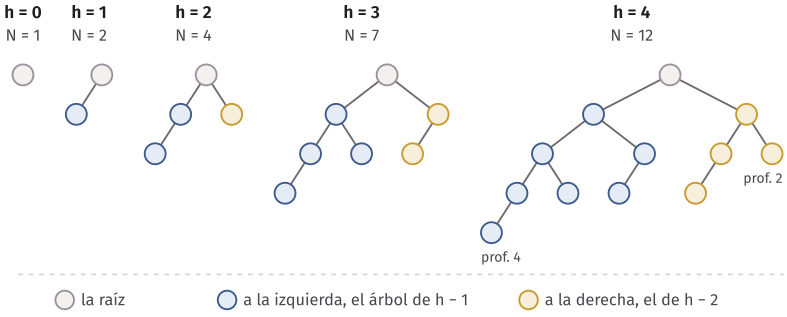
- $N(0) = 1$, y $N(1) = 2$.
- Con $h \geq 2$: una raíz, un subárbol de altura $h - 1$, y el otro lo más bajo que se permita: $h - 2$.
- Y cada uno, con la menor cantidad posible de nodos.

La recurrencia

$$N(h) = N(h-1) + N(h-2) + 1$$

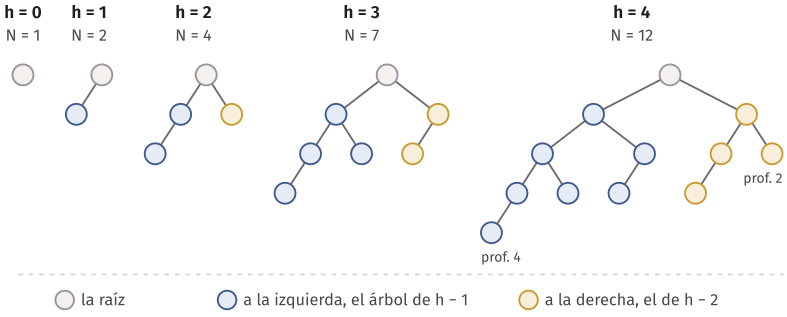
$N(2) = 4$, $N(3) = 7$, $N(4) = \mathbf{12}$.

Los árboles de Fibonacci



N(h) = N(h − 1) + N(h − 2) + 1, y por ejemplo N(4) = 7 + 4 + 1 = 12

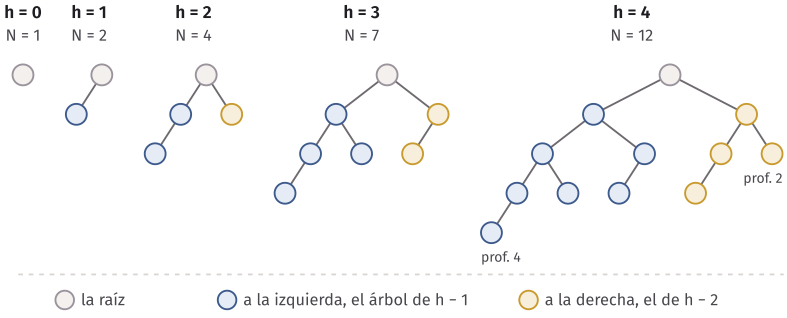
Los árboles de Fibonacci



N(h) = N(h − 1) + N(h − 2) + 1, y por ejemplo N(4) = 7 + 4 + 1 = 12

- Esas dos hojas a profundidades tan distintas las permite la condición: por eso la altura no es $\log_2 n$, sino algo más.

Los árboles de Fibonacci



N(h) = N(h − 1) + N(h − 2) + 1, y por ejemplo N(4) = 7 + 4 + 1 = 12

- Esas dos hojas a profundidades tan distintas las permite la condición: por eso la altura no es $\log_2 n$, sino algo más.
- Si sus claves se insertan por niveles, no hace falta ninguna rotación.

Una cota sencilla

N crece con h, así que $N(h-1) > N(h-2)$. Entonces:

$$N(h) = N(h-1) + N(h-2) + 1 > 2\,N(h-2)$$

Una cota sencilla

N crece con h, así que $N(h-1) > N(h-2)$. Entonces:

$$N(h) = N(h-1) + N(h-2) + 1 > 2\,N(h-2)$$

Cada vez que la altura sube en 2, los nodos más que se duplican:

$$N(h) > 2N(h-2) > 4N(h-4) > 8N(h-6) > \dots$$

y se llega a $N(0) = 1$ o a $N(1) = 2$, en unos $h/2$ pasos. Por lo tanto, $N(h) \geq 2^{h/2}$.

Una cota sencilla

N crece con h, así que $N(h-1) > N(h-2)$. Entonces:

$$N(h) = N(h-1) + N(h-2) + 1 > 2\,N(h-2)$$

Cada vez que la altura sube en 2, los nodos más que se duplican:

$$N(h) > 2N(h-2) > 4N(h-4) > 8N(h-6) > \dots$$

y se llega a $N(0) = 1$ o a $N(1) = 2$, en unos $h/2$ pasos. Por lo tanto, $N(h) \geq 2^{h/2}$.

Ya está resuelto el problema

Con n nodos y altura h: $n \geq N(h) \geq 2^{h/2}$. Es decir, $h \leq 2 \log_2 n$.
Con un millón de claves, la altura es como máximo 39, y no 999 999.

La cota exacta

Sumen 1 a los dos lados: $N(h) + 1 = \big(N(h-1) + 1\big) + \big(N(h-2) + 1\big)$

La cota exacta

Sumen 1 a los dos lados: $N(h) + 1 = \big(N(h-1) + 1\big) + \big(N(h-2) + 1\big)$

- Es Fibonacci: $N(h) = F(h+3) - 1$, con $F(1) = F(2) = 1$.

La cota exacta

Sumen 1 a los dos lados: $N(h) + 1 = \big(N(h-1) + 1\big) + \big(N(h-2) + 1\big)$

- Es Fibonacci: $N(h) = F(h+3) - 1$, con $F(1) = F(2) = 1$.
- Fórmula de Binet: $F(k)$ es el entero más cercano a $\varphi^k/\sqrt{5}$, con $\varphi \approx 1.618$. La usamos sin demostrarla.

La cota exacta

Sumen 1 a los dos lados: $N(h) + 1 = \big(N(h-1) + 1\big) + \big(N(h-2) + 1\big)$

- Es Fibonacci: $N(h) = F(h+3) - 1$, con $F(1) = F(2) = 1$.
- Fórmula de Binet: $F(k)$ es el entero más cercano a $\varphi^k/\sqrt{5}$, con $\varphi \approx 1.618$. La usamos sin demostrarla.
- Entonces $n + 2 > \varphi^{h+3}/\sqrt{5}$, y tomando logaritmos:

La cota exacta

Sumen 1 a los dos lados: $N(h) + 1 = \big(N(h-1)+1\big) + \big(N(h-2)+1\big)$

- Es Fibonacci: $N(h) = F(h+3) - 1$, con $F(1) = F(2) = 1$.
- Fórmula de Binet: $F(k)$ es el entero más cercano a $\varphi^k/\sqrt{5}$, con $\varphi \approx 1.618$. La usamos sin demostrarla.
- Entonces $n + 2 > \varphi^{h+3}/\sqrt{5}$, y tomando logaritmos:

La forma exacta, contando aristas

$$h < \log_\varphi(n+2) + \log_\varphi \sqrt{5} - 3, \text{ es decir, } h < 1.4405 \log_2(n+2) - 1.3277$$

Los libros que cuentan nodos tienen una altura 1 mayor, y escriben -0.3277. Es la misma cota. En corto: **unas 1.44 veces** $\log_2 n$.

Comprobado, con árboles de verdad

terminal

```
$ build/min_nodes
  h    N(h) F(h+3)-1 medida  rot  log2n 1.44log    cota
  0      1        1      0    0   0.00    0.00    0.96
  1      2        2      1    0   1.00    1.44    1.55
  2      4        4      2    0   2.00    2.88    2.40
  3      7        7      3    0   2.81    4.04    3.24
  4     12       12      4    0   3.58    5.16    4.16
  8     88       88      8    0   6.46    9.30    8.02
 12    609      609     12    0   9.25   13.32   12.00
 16   4180     4180     16    0  12.03   17.32   16.00
 24 196417   196417     24    0  17.58   25.32   24.00
```

Comprobado, con árboles de verdad

terminal

```
$ build/min_nodes
  h     N(h) F(h+3)-1 medida  rot   log2n 1.44log    cota
  0       1        1      0    0    0.00    0.00    0.96
  1       2        2      1    0    1.00    1.44    1.55
  2       4        4      2    0    2.00    2.88    2.40
  3       7        7      3    0    2.81    4.04    3.24
  4      12       12      4    0    3.58    5.16    4.16
  8      88       88      8    0    6.46    9.30    8.02
 12     609      609     12    0    9.25   13.32   12.00
 16    4180     4180     16    0   12.03   17.32   16.00
 24  196417   196417     24    0   17.58   25.32   24.00
```

- `medida`: se armó el árbol y se midió. `rot`: sin ninguna rotación.

Comprobado, con árboles de verdad

terminal

```
$ build/min_nodes
  h     N(h) F(h+3)-1 medida  rot   log2n 1.44log    cota
  0       1        1      0    0    0.00    0.00    0.96
  1       2        2      1    0    1.00    1.44    1.55
  2       4        4      2    0    2.00    2.88    2.40
  3       7        7      3    0    2.81    4.04    3.24
  4      12       12      4    0    3.58    5.16    4.16
  8      88       88      8    0    6.46    9.30    8.02
 12     609      609     12    0    9.25   13.32   12.00
 16    4180     4180     16    0   12.03   17.32   16.00
 24  196417   196417     24    0   17.58   25.32   24.00
```

- `medida`: se armó el árbol y se midió. `rot`: sin ninguna rotación.
- `cota` se pega a *h*: esos árboles casi la tocan.

Tres precisiones sobre el 1.44

- **Es un redondeo hacia abajo.** La constante es 1.44042…, así que "$h \leq 1.44 \log_2 n$" termina fallando: la primera vez en $h = 4552$, con un árbol de un número de nodos de 952 dígitos. Lo comprueban las pruebas, con enteros exactos.

Tres precisiones sobre el 1.44

- **Es un redondeo hacia abajo.** La constante es 1.44042…, así que "$h \leq 1.44 \log_2 n$" termina fallando: la primera vez en $h = 4552$, con un árbol de un número de nodos de 952 dígitos. Lo comprueban las pruebas, con enteros exactos.
- **Con $n = 1$ no es "por debajo".** $h = 0$ y $1.44 \log_2 1 = 0$. Lo correcto es $\leq$, con $n \geq 1$.

Tres precisiones sobre el 1.44

- **Es un redondeo hacia abajo.** La constante es 1.44042…, así que "$h \leq 1.44 \log_2 n$" termina fallando: la primera vez en $h = 4552$, con un árbol de un número de nodos de 952 dígitos. Lo comprueban las pruebas, con enteros exactos.
- **Con $n = 1$ no es "por debajo".** $h = 0$ y $1.44 \log_2 1 = 0$. Lo correcto es $\leq$, con $n \geq 1$.
- **Es el peor caso, y es raro.** Hay que armar un árbol de Fibonacci a propósito. Al azar y con claves ordenadas se queda bastante más abajo.

Tres precisiones sobre el 1.44

- **Es un redondeo hacia abajo.** La constante es 1.44042…, así que "$h \leq 1.44 \log_2 n$" termina fallando: la primera vez en $h = 4552$, con un árbol de un número de nodos de 952 dígitos. Lo comprueban las pruebas, con enteros exactos.
- **Con $n = 1$ no es "por debajo".** $h = 0$ y $1.44 \log_2 1 = 0$. Lo correcto es $\leq$, con $n \geq 1$.
- **Es el peor caso, y es raro.** Hay que armar un árbol de Fibonacci a propósito. Al azar y con claves ordenadas se queda bastante más abajo.

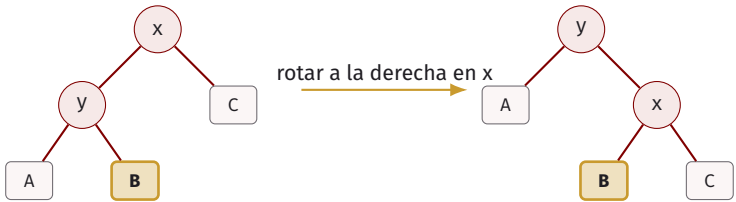
> **Idea clave**
>
> Una condición local, en cada nodo, limita la altura total: un AVL alto necesita muchos nodos.

SECCIÓN 04

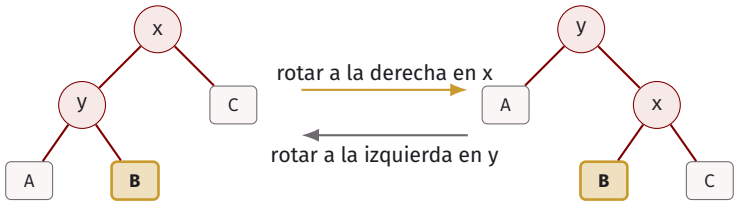
Rotación

04

Rotación: la operación

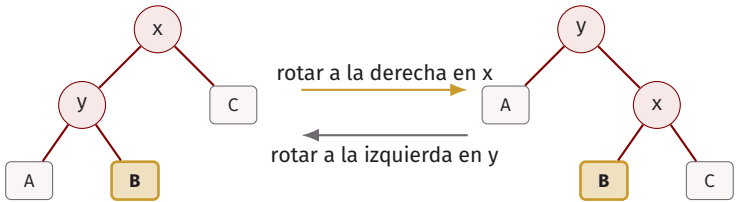


Rotación: la operación



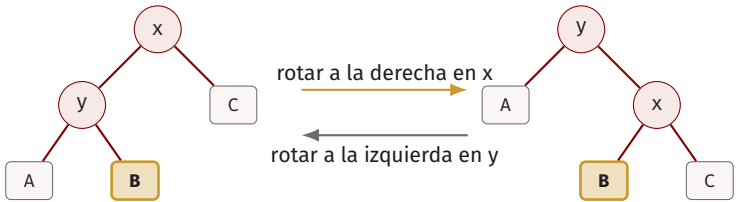
- Son inversas una de otra. A, B y C pueden estar vacíos.

Rotación: la operación



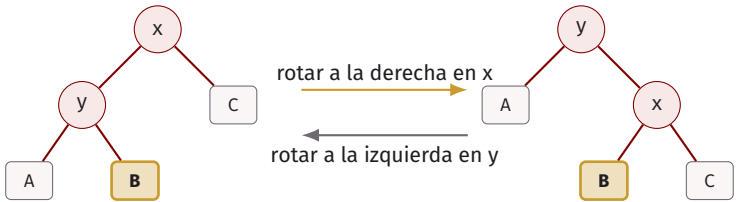
- Son inversas una de otra. A, B y C pueden estar vacíos.
- x era padre de y, y ahora es su hijo.

Rotación: la operación



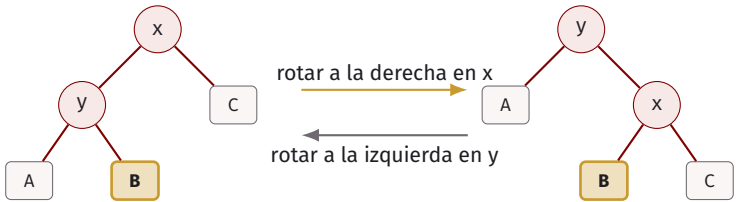
- Son inversas una de otra. A, B y C pueden estar vacíos.
- x era padre de y, y ahora es su hijo.
- **B, el subárbol del medio, cambia de padre.** A y C no.

Rotación: la operación



- Son inversas una de otra. A, B y C pueden estar vacíos.
- x era padre de y, y ahora es su hijo.
- **B, el subárbol del medio, cambia de padre.** A y C no.
- Lo que colgaba del padre de x ahora es y.

Rotación: la operación



- Son inversas una de otra. A, B y C pueden estar vacíos.
- x era padre de y, y ahora es su hijo.
- **B, el subárbol del medio, cambia de padre.** A y C no.
- Lo que colgaba del padre de x ahora es y.
- Tres enlaces hacia abajo, sin importar cuántos nodos haya: **O(1)**.

Una rotación no rompe el orden



- Recorrido en orden, antes: en orden de (y con A y B), x, C. Es decir, **A, y, B, x, C**.

Una rotación no rompe el orden



- Recorrido en orden, antes: en orden de (y con A y B), x, C. Es decir, **A, y, B, x, C**.
- Después: A, y, en orden de (x con B y C). Es decir, **A, y, B, x, C**.

Una rotación no rompe el orden



- Recorrido en orden, antes: en orden de (y con A y B), x, C. Es decir, **A, y, B, x, C**.
- Después: A, y, en orden de (x con B y C). Es decir, **A, y, B, x, C**.
- Si el recorrido sale igual y estaba ordenado, sigue ordenado. Y lo de afuera no se entera.

Una rotación no rompe el orden



- Recorrido en orden, antes: en orden de (y con A y B), x, C. Es decir, **A, y, B, x, C**.
- Después: A, y, en orden de (x con B y C). Es decir, **A, y, B, x, C**.
- Si el recorrido sale igual y estaba ordenado, sigue ordenado. Y lo de afuera no se entera.
- Con desigualdades: $A < y < B < x < C$. Por eso B es lo único que cabe entre y y x.

Una rotación, a mano

terminal

```
$ build/rotation_demo
inicio: 40(20(10,30),60)
  en orden: 10 20 30 40 60
rotar a la derecha en 40: 20(10,40(30,60))
  en orden: 10 20 30 40 60
rotar a la izquierda en 20: 40(20(10,30),60)
  en orden: 10 20 30 40 60
```

Una rotación, a mano

terminal

```
$ build/rotation_demo
inicio: 40(20(10,30),60)
  en orden: 10 20 30 40 60
rotar a la derecha en 40: 20(10,40(30,60))
  en orden: 10 20 30 40 60
rotar a la izquierda en 20: 40(20(10,30),60)
  en orden: 10 20 30 40 60
```

- La forma cambia, y la línea `en orden` no.

Una rotación, a mano

terminal

```
$ build/rotation_demo
inicio: 40(20(10,30),60)
  en orden: 10 20 30 40 60
rotar a la derecha en 40: 20(10,40(30,60))
  en orden: 10 20 30 40 60
rotar a la izquierda en 20: 40(20(10,30),60)
  en orden: 10 20 30 40 60
```

- La forma cambia, y la línea `en orden` no.
- A es el 10, B es el 30, C es el 60.

Una rotación, a mano

terminal

```
$ build/rotation_demo
inicio: 40(20(10,30),60)
  en orden: 10 20 30 40 60
rotar a la derecha en 40: 20(10,40(30,60))
  en orden: 10 20 30 40 60
rotar a la izquierda en 20: 40(20(10,30),60)
  en orden: 10 20 30 40 60
```

- La forma cambia, y la línea `en orden` no.
- A es el 10, B es el 30, C es el 60.
- Rotar y volver a rotar deja el árbol como estaba.

Una rotación cambia las alturas

- Después de rotar a la derecha: A sube un nivel, C baja uno, y B se queda igual.

Una rotación cambia las alturas

- Después de rotar a la derecha: A sube un nivel, C baja uno, y B se queda igual.
- Si A era el subárbol alto, eso arregla el desequilibrio. Para eso rotamos.

Una rotación cambia las alturas

- Después de rotar a la derecha: A sube un nivel, C baja uno, y B se queda igual.
- Si A era el subárbol alto, eso arregla el desequilibrio. Para eso rotamos.
- Solo pueden cambiar las alturas de **x e y**: A, B y C no cambiaron por dentro.

Una rotación cambia las alturas

- Después de rotar a la derecha: A sube un nivel, C baja uno, y B se queda igual.
- Si A era el subárbol alto, eso arregla el desequilibrio. Para eso rotamos.
- Solo pueden cambiar las alturas de **x e y**: A, B y C no cambiaron por dentro.

El orden

Primero x, que quedó abajo. Después y, que quedó arriba: se calcula con la altura de x.

Con puntero al padre: lo que escribirán

pseudocódigo

```
algoritmo rotar_izquierda(arbol, x)
    y ← hijo derecho de x
    medio ← hijo izquierdo de y
    hijo derecho de x ← medio
    si medio ≠ NULO entonces
        padre de medio ← x
    fin si
    poner a y en el lugar de x: en el enlace
        del padre de x, o en la raíz de arbol
    padre de y ← padre de x
    hijo izquierdo de y ← x
    padre de x ← y
    devolver y
fin algoritmo
```

- Tres enlaces hacia abajo, y sus tres padres.

Con puntero al padre: lo que escribirán

pseudocódigo

```
algoritmo rotar_izquierda(arbol, x)
    y ← hijo derecho de x
    medio ← hijo izquierdo de y
    hijo derecho de x ← medio
    si medio ≠ NULO entonces
        padre de medio ← x
    fin si
    poner a y en el lugar de x: en el enlace
        del padre de x, o en la raíz de arbol
    padre de y ← padre de x
    hijo izquierdo de y ← x
    padre de x ← y
    devolver y
fin algoritmo
```

- Tres enlaces hacia abajo, y sus tres padres.
- La raíz no es un enlace de ningún nodo: de eso se encarga `replace_child`.

Con puntero al padre: lo que escribirán

pseudocódigo

```
algoritmo rotar_izquierda(arbol, x)
    y ← hijo derecho de x
    medio ← hijo izquierdo de y
    hijo derecho de x ← medio
    si medio ≠ NULO entonces
        padre de medio ← x
    fin si
    poner a y en el lugar de x: en el enlace
        del padre de x, o en la raíz de arbol
    padre de y ← padre de x
    hijo izquierdo de y ← x
    padre de x ← y
    devolver y
fin algoritmo
```

- Tres enlaces hacia abajo, y sus tres padres.
- La raíz no es un enlace de ningún nodo: de eso se encarga `replace_child`.

Error común

Olvidar el `parent` de B. El recorrido en orden sale bien, porque baja. Falla después, al subir.

Sin padre, en la clase de los apuntes

avl_set.h

```
1  Node* rotate_left(Node* x) {
2      Node* y = x->right;
3      x->right = y->left;
4      y->left = x;
5      rotations_++;
6      return y;
7  }
```

- Quien llama escribe el resultado en su enlace.

Sin padre, en la clase de los apuntes

avl_set.h

```
1  Node* rotate_left(Node* x) {
2      Node* y = x->right;
3      x->right = y->left;
4      y->left = x;
5      rotations_++;
6      return y;
7  }
```

- Quien llama escribe el resultado en su enlace.
- **Solo cambia enlaces.** Ningún nodo cambia de clave ni de dirección.

Sin padre, en la clase de los apuntes

avl_set.h

```
1  Node* rotate_left(Node* x) {
2      Node* y = x->right;
3      x->right = y->left;
4      y->left = x;
5      rotations_++;
6      return y;
7  }
```

- Quien llama escribe el resultado en su enlace.
- **Solo cambia enlaces.** Ningún nodo cambia de clave ni de dirección.
- Por eso un puntero a un valor, o un iterador, sobrevive a una rotación.

Sin padre, en la clase de los apuntes

avl_set.h

```
1  Node* rotate_left(Node* x) {
2      Node* y = x->right;
3      x->right = y->left;
4      y->left = x;
5      rotations_++;
6      return y;
7  }
```

- Quien llama escribe el resultado en su enlace.
- **Solo cambia enlaces.** Ningún nodo cambia de clave ni de dirección.
- Por eso un puntero a un valor, o un iterador, sobrevive a una rotación.
- Y no toca las alturas: la semana 12 usa rotaciones en un árbol que no las tiene.

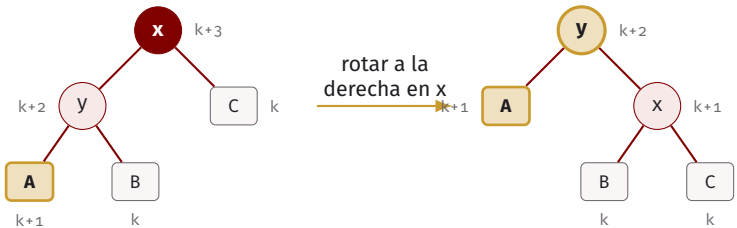
SECCIÓN 05

Los cuatro casos

05

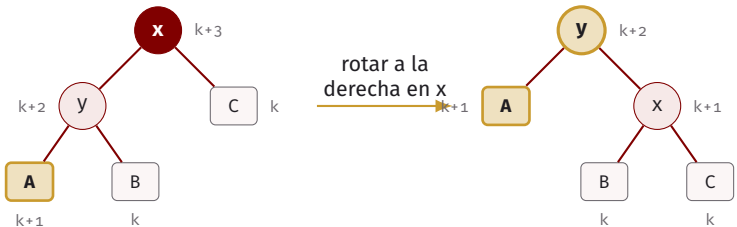
Izquierda-izquierda: una rotación simple

x tiene balance 2, y su hijo y tiene balance 1.



Izquierda-izquierda: una rotación simple

x tiene balance 2, y su hijo y tiene balance 1.



- A sube un nivel: justo lo que le sobraba. x e y quedan con balance 0.

Izquierda-izquierda: una rotación simple

x tiene balance 2, y su hijo y tiene balance 1.

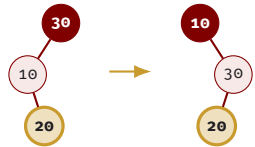


- A sube un nivel: justo lo que le sobraba. x e y quedan con balance 0.
- `derecha-derecha` es el espejo: balance -2 y -1, y se rota a la izquierda.

Izquierda-derecha: una rotación no alcanza

```
terminal

$ build/single_on_zigzag
insertar 30: 30
insertar 10: 30(10,_)
insertar 20: 10(_,30(20,_))
```

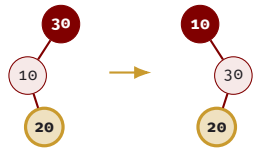


Y luego `check_invariants` detiene el programa:
`abs(left - right) <= 1` no se cumple.

Izquierda-derecha: una rotación no alcanza

terminal

```
$ build/single_on_zigzag
insertar 30: 30
insertar 10: 30(10,_)
insertar 20: 10(_,30(20,_))
```



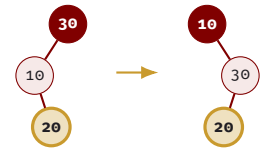
Y luego `check_invariants` detiene el programa:
`abs(left - right) <= 1` no se cumple.

- El zigzag se volvió otro zigzag, el espejo, igual de desbalanceado.

Izquierda-derecha: una rotación no alcanza

```
terminal

$ build/single_on_zigzag
insertar 30: 30
insertar 10: 30(10,_)
insertar 20: 10(_,30(20,_))
```

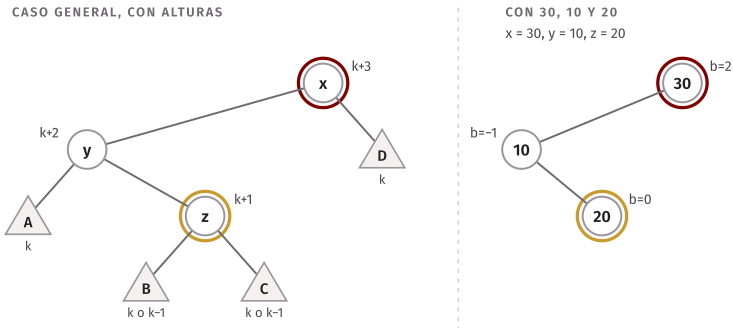


Y luego `check_invariants` detiene el programa:
`abs(left - right) <= 1` no se cumple.

- El zigzag se volvió otro zigzag, el espejo, igual de desbalanceado.
- Lo pesado es el subárbol **del medio**, y el del medio no sube al rotar: solo se muda.

Izquierda-derecha: una rotación doble

x tiene balance 2, y su hijo y tiene balance -1.



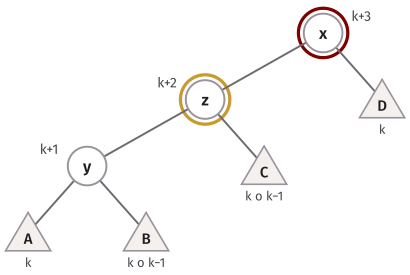
x tiene balance 2 y su hijo y tiene −1. Signos opuestos: caso izquierda-derecha.

Una sola idea en dos rotaciones: **el del medio sube hasta arriba**. `derecha-izquierda` es el espejo.

Izquierda-derecha: una rotación doble

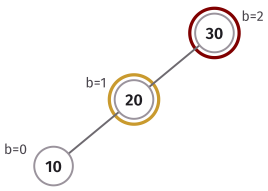
x tiene balance 2, y su hijo y tiene balance −1.

CASO GENERAL, CON ALTURAS



CON 30, 10 Y 20

x = 30, y = 10, z = 20



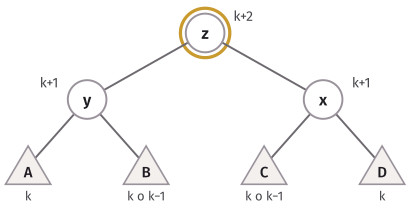
Rotar a la izquierda en y: z sube, y queda el caso izquierda-izquierda.

Una sola idea en dos rotaciones: **el del medio sube hasta arriba**. `derecha-izquierda` es el espejo.

Izquierda-derecha: una rotación doble

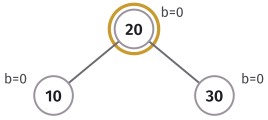
x tiene balance 2, y su hijo y tiene balance −1.

CASO GENERAL, CON ALTURAS



CON 30, 10 Y 20

x = 30, y = 10, z = 20



Rotar a la derecha en x: z queda arriba, con altura k + 2, y B y C se reparten entre y y x.

Una sola idea en dos rotaciones: **el del medio sube hasta arriba**. `derecha-izquierda` es el espejo.

Cómo se decide

x	hijo pesado	Caso	Reparación
2	1	izquierda-izquierda	derecha en x
2	0	igual, solo al borrar	derecha en x
2	−1	izquierda-derecha	izquierda en el hijo, y derecha en x
−2	−1	derecha-derecha	izquierda en x
−2	0	igual, solo al borrar	izquierda en x
−2	1	derecha-izquierda	derecha en el hijo, e izquierda en x

Cómo se decide

x	hijo pesado	Caso	Reparación
2	1	izquierda-izquierda	derecha en x
2	0	igual, solo al borrar	derecha en x
2	−1	izquierda-derecha	izquierda en el hijo, y derecha en x
−2	−1	derecha-derecha	izquierda en x
−2	0	igual, solo al borrar	izquierda en x
−2	1	derecha-izquierda	derecha en el hijo, e izquierda en x

Para recordarlo

Si los dos signos coinciden, una rotación. Si son opuestos, dos.

Los cuatro, sobre 3 claves

terminal

```
$ build/avl_trace 30 20 10
insertar 30: 30
insertar 20: 30(20,_)
insertar 10: 20(10,30)
    reparo en 30: izquierda-izquierda
altura 1, rotaciones 1
$ build/avl_trace 10 20 30
insertar 10: 10
insertar 20: 10(_,20)
insertar 30: 20(10,30)
    reparo en 10: derecha-derecha
altura 1, rotaciones 1
```

terminal

```
$ build/avl_trace 30 10 20
insertar 30: 30
insertar 10: 30(10,_)
insertar 20: 20(10,30)
    reparo en 30: izquierda-derecha
altura 1, rotaciones 2
$ build/avl_trace 10 30 20
insertar 10: 10
insertar 30: 10(_,30)
insertar 20: 20(10,30)
    reparo en 10: derecha-izquierda
altura 1, rotaciones 2
```

Los cuatro terminan en `20(10,30)`: la clave del medio arriba.

La reparación

pseudocódigo

```
algoritmo reparar(arbol, x)
    si balance(x) > 0 entonces
        y ← hijo izquierdo de x
        si balance(y) < 0 entonces
            z ← rotar_izquierda(arbol, y)
            alturas: primero y, luego z
        fin si
        arriba ← rotar_derecha(arbol, x)
    sino
        el espejo de lo anterior
    fin si
    alturas: primero x, luego arriba
    devolver arriba
fin algoritmo
```

avl_set.h

```
1  Node* repair_left_heavy(Node* x) {
2      Node* y = x->left;
3      if (balance_of(y) < 0) {
4          x->left = rotate_left(y);
5          update_height(y);
6          update_height(x->left);
7      }
8      Node* top = rotate_right(x);
9      update_height(x);
10     update_height(top);
11     return top;
12 }
```

Después de `rotate_right(x)`, el hijo es x: va primero.

SECCIÓN 06

Insertar

06

Insertar: subir reparando

pseudocódigo

```
algoritmo subir_reparando(arbol, nodo)
    mientras nodo ≠ NULO hacer
        actualizar la altura de nodo
        si balance(nodo) > 1 o balance(nodo) < -1 entonces
            nodo ← reparar(arbol, nodo)
        fin si
        nodo ← padre de nodo
    fin mientras
fin algoritmo
```

- Primero, insertar como la semana pasada: se cuelga una hoja.

Insertar: subir reparando

pseudocódigo

```
algoritmo subir_reparando(arbol, nodo)
    mientras nodo ≠ NULO hacer
        actualizar la altura de nodo
        si balance(nodo) > 1 o balance(nodo) < -1 entonces
            nodo ← reparar(arbol, nodo)
        fin si
        nodo ← padre de nodo
    fin mientras
fin algoritmo
```

- Primero, insertar como la semana pasada: se cuelga una hoja.
- Solo cambian las alturas de los antepasados del nodo nuevo.

Insertar: subir reparando

pseudocódigo

```
algoritmo subir_reparando(arbol, nodo)
    mientras nodo ≠ NULO hacer
        actualizar la altura de nodo
        si balance(nodo) > 1 o balance(nodo) < -1 entonces
            nodo ← reparar(arbol, nodo)
        fin si
        nodo ← padre de nodo
    fin mientras
fin algoritmo
```

- Primero, insertar como la semana pasada: se cuelga una hoja.
- Solo cambian las alturas de los antepasados del nodo nuevo.
- Se sube desde su padre, hasta la raíz.

Con recursión: la vuelta es la subida

avl_set.h

```
1  Node* insert_at(Node* node, const K& key, bool&
       added) {
2      if (node == nullptr) {
3          added = true;
4          return new Node{key};
5      }
6      if (key < node->key) {
7          node->left = insert_at(node->left, key,
       added);
8      } else if (node->key < key) {
9          node->right = insert_at(node->right, key,
       added);
10     } else {
11         return node;
12     }
13     return rebalance(node);
14 }
```

- Cada nivel escribe en su enlace lo que le devuelve el de abajo, que puede ser otro nodo.

Con recursión: la vuelta es la subida

avl_set.h

```
1  Node* insert_at(Node* node, const K& key, bool&
       added) {
2      if (node == nullptr) {
3          added = true;
4          return new Node{key};
5      }
6      if (key < node->key) {
7          node->left = insert_at(node->left, key,
       added);
8      } else if (node->key < key) {
9          node->right = insert_at(node->right, key,
       added);
10     } else {
11         return node;
12     }
13     return rebalance(node);
14 }
```

- Cada nivel escribe en su enlace lo que le devuelve el de abajo, que puede ser otro nodo.
- La semana pasada, la recursión podía agotar la pila en un árbol alto.

Con recursión: la vuelta es la subida

avl_set.h

```
1  Node* insert_at(Node* node, const K& key, bool&
       added) {
2      if (node == nullptr) {
3          added = true;
4          return new Node{key};
5      }
6      if (key < node->key) {
7          node->left = insert_at(node->left, key,
       added);
8      } else if (node->key < key) {
9          node->right = insert_at(node->right, key,
       added);
10     } else {
11         return node;
12     }
13     return rebalance(node);
14 }
```

- Cada nivel escribe en su enlace lo que le devuelve el de abajo, que puede ser otro nodo.
- La semana pasada, la recursión podía agotar la pila en un árbol alto.
- Aquí, con un millón de claves, la altura es como máximo 27: menos de 30 marcos.

Con recursión: la vuelta es la subida

avl_set.h

```
1  Node* insert_at(Node* node, const K& key, bool&
       added) {
2      if (node == nullptr) {
3          added = true;
4          return new Node{key};
5      }
6      if (key < node->key) {
7          node->left = insert_at(node->left, key,
       added);
8      } else if (node->key < key) {
9          node->right = insert_at(node->right, key,
       added);
10     } else {
11         return node;
12     }
13     return rebalance(node);
14 }
```

- Cada nivel escribe en su enlace lo que le devuelve el de abajo, que puede ser otro nodo.
- La semana pasada, la recursión podía agotar la pila en un árbol alto.
- Aquí, con un millón de claves, la altura es como máximo 27: menos de 30 marcos.

Mantener la invariante paga

El balance es lo que hace segura la recursión.

Paso a paso: 1, 2, 3, 4, 5, 6, 7



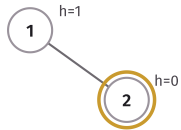
rotaciones: 0

Insertar 1: una hoja, de altura 0.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

rotaciones: 0

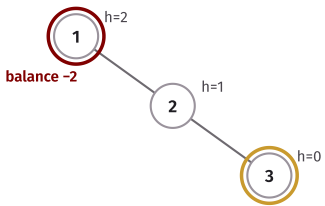


Insertar 2: el 1 queda con balance −1. Todavía cumple la condición.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

rotaciones: 0

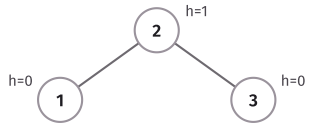


Insertar 3: el 1 queda con balance −2, y su hijo pesado 2 con −1. Caso derecha-derecha.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

rotaciones: 1

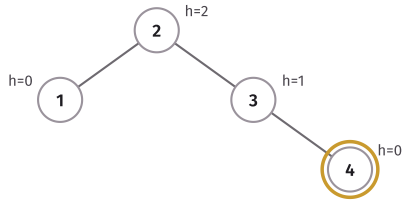


Reparar en 1: rotar a la izquierda. Sube el 2, y los dos lados quedan iguales.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

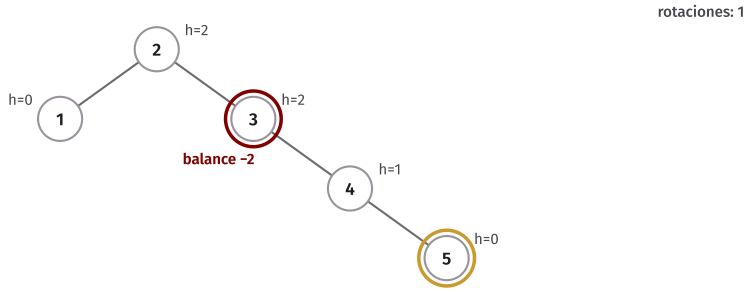
rotaciones: 1



Insertar 4: el 3 y el 2 quedan con balance −1. No hay nada que reparar.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

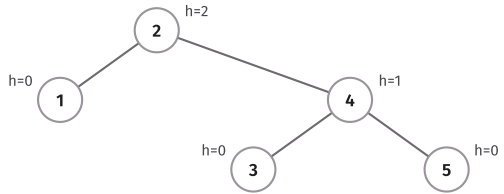


Insertar 5: al subir, el 3 es el primero con balance −2. La raíz aún guarda su altura de antes.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

rotaciones: 2

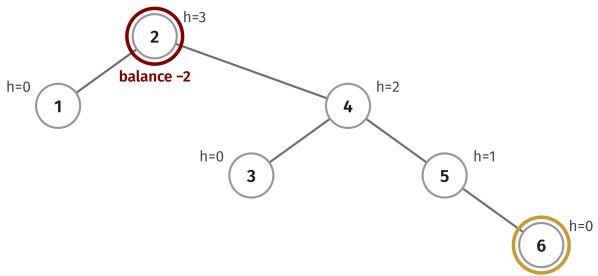


Reparar en 3: rotar a la izquierda. El subárbol recupera su altura, y la raíz sigue en h=2.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

rotaciones: 2

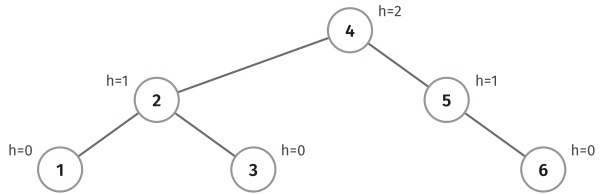


Insertar 6: el 5 y el 4 quedan en −1, y la raíz 2 en −2. Otra vez derecha-derecha.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7

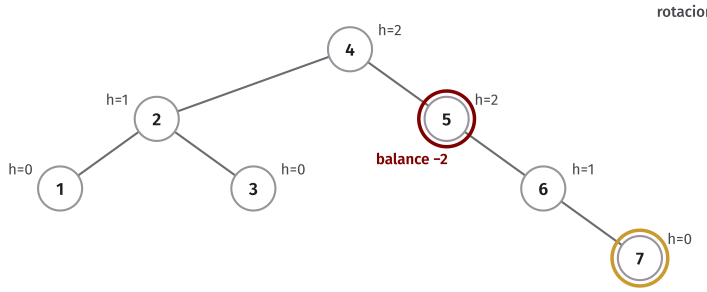
rotaciones: 3



Reparar en 2: rotar a la izquierda. Sube el 4, y el 3, el del medio, se muda bajo el 2.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

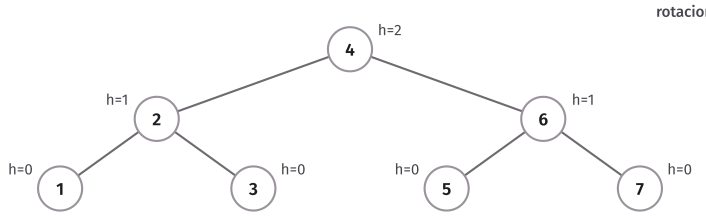
Paso a paso: 1, 2, 3, 4, 5, 6, 7



Insertar 7: al subir, el 5 es el primero con balance −2.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Paso a paso: 1, 2, 3, 4, 5, 6, 7



Reparar en 5: rotar a la izquierda. Árbol perfecto: altura 2, con 4 rotaciones en 7 inserciones.

Se repara al insertar el 3 (en 1), el 5 (en 3), el 6 (en 2) y el 7 (en 5), siempre derecha-derecha. Sin balance, estas claves dan una cadena.

Basta una reparación

Afirmación

En una inserción, basta **una** reparación, simple o doble: como máximo, 2 rotaciones.

Caso	Antes de insertar	Sin reparar	Tras reparar
izquierda-izquierda	k + 2	k + 3	k + 2
izquierda-derecha	k + 2	k + 3	k + 2

Basta una reparación

Afirmación

En una inserción, basta **una** reparación, simple o doble: como máximo, 2 rotaciones.

Caso	Antes de insertar	Sin reparar	Tras reparar
izquierda-izquierda	k + 2	k + 3	k + 2
izquierda-derecha	k + 2	k + 3	k + 2

- **La reparación devuelve la altura de antes.** Los antepasados estaban bien con esa altura, y para ellos no cambió nada.

Basta una reparación

Afirmación

En una inserción, basta **una** reparación, simple o doble: como máximo, 2 rotaciones.

Caso	Antes de insertar	Sin reparar	Tras reparar
izquierda-izquierda	k + 2	k + 3	k + 2
izquierda-derecha	k + 2	k + 3	k + 2

- **La reparación devuelve la altura de antes.** Los antepasados estaban bien con esa altura, y para ellos no cambió nada.
- Antes de insertar, el hijo pesado tenía balance 0. Después tiene ± 1: al insertar, nunca hay balance 0 en él.

Todos los órdenes

terminal

```
$ build/all_orders
  n ordenes  h.min  h.max  max.reparaciones  max.rotaciones
  1       1      0      0                 0               0
  2       2      1      1                 0               0
  3       6      1      1                 1               2
  4      24      2      2                 1               2
  5     120      2      2                 1               2
  6     720      2      2                 1               2
  7    5040      2      3                 1               2
  8   40320      3      3                 1               2
  9  362880      3      3                 1               2
```

Todos los órdenes

terminal

```
$ build/all_orders
  n ordenes  h.min  h.max  max.reparaciones  max.rotaciones
  1       1      0      0                 0               0
  2       2      1      1                 0               0
  3       6      1      1                 1               2
  4      24      2      2                 1               2
  5     120      2      2                 1               2
  6     720      2      2                 1               2
  7    5040      2      3                 1               2
  8   40320      3      3                 1               2
  9  362880      3      3                 1               2
```

- Ninguna inserción pasó de 1 reparación. Con 6 claves: la semana pasada, alturas de 2 a 5. Aquí, todas 2.

Todos los órdenes

terminal

```
$ build/all_orders
  n ordenes  h.min  h.max  max.reparaciones  max.rotaciones
  1       1      0      0                 0               0
  2       2      1      1                 0               0
  3       6      1      1                 1               2
  4      24      2      2                 1               2
  5     120      2      2                 1               2
  6     720      2      2                 1               2
  7    5040      2      3                 1               2
  8   40320      3      3                 1               2
  9  362880      3      3                 1               2
```

- Ninguna inserción pasó de 1 reparación. Con 6 claves: la semana pasada, alturas de 2 a 5. Aquí, todas 2.
- Esto no lo demuestra para n grande: eso lo hace el argumento de las alturas.

Qué dice “basta una”, y qué no

Dice

Se hace como máximo una reparación, de 1 o 2 rotaciones.

No dice

Que la subida sea corta: para encontrar el nodo hay que subir, O(log n).

Que haya una en cada inserción: muchas no reparan nada.

Que valga al borrar. No vale.

Qué dice “basta una”, y qué no

Dice

Se hace como máximo una reparación, de 1 o 2 rotaciones.

Costo de insertar

Bajar O(log n), subir O(log n), reparar O(1): **O(log n) en el peor caso**, para cada inserción. No es un promedio ni un costo amortizado.

No dice

Que la subida sea corta: para encontrar el nodo hay que subir, O(log n).

Que haya una en cada inserción: muchas no reparan nada.

Que valga al borrar. No vale.

SECCIÓN 07

Borrar

07

Borrar: la misma idea, y dos diferencias

pseudocódigo

```
algoritmo borrar(arbol, clave)
    nodo ← el nodo con esa clave, o devolver falso
    si nodo tiene dos hijos entonces
        sucesor ← el mínimo del subárbol derecho de nodo
        copiar a nodo la clave de sucesor
        nodo ← sucesor
    fin si
    padre ← padre de nodo
    poner al único hijo de nodo, o NULO, en su lugar
    liberar nodo
    subir_reparando(arbol, padre)
    devolver verdadero
fin algoritmo
```

Error común

Subir desde el padre del nodo con la clave borrada. Con dos hijos, el que salió es el sucesor, y quedarían alturas viejas.

Borrar: la misma idea, y dos diferencias

pseudocódigo

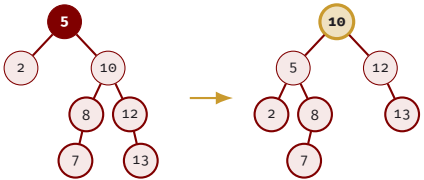
```
algoritmo borrar(arbol, clave)
    nodo ← el nodo con esa clave, o devolver falso
    si nodo tiene dos hijos entonces
        sucesor ← el mínimo del subárbol derecho de nodo
        copiar a nodo la clave de sucesor
        nodo ← sucesor
    fin si
    padre ← padre de nodo
    poner al único hijo de nodo, o NULO, en su lugar
    liberar nodo
    subir_reparando(arbol, padre)
    devolver verdadero
fin algoritmo
```

Error común

Subir desde el padre del nodo con la clave borrada. Con dos hijos, el que salió es el sucesor, y quedarían alturas viejas.

Por eso `remove_node` devuelve el padre del nodo que salió.

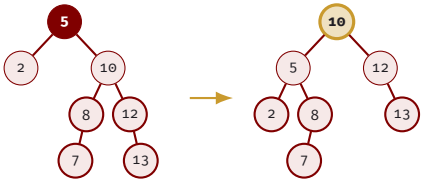
Primera diferencia: balance 0 en el hijo pesado



Ya sin el 1. El hijo 10 tiene balance 0.

- Al borrar, baja el otro lado: el hijo pesado no se tocó.

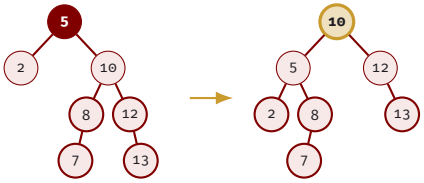
Primera diferencia: balance 0 en el hijo pesado



Ya sin el 1. El hijo 10 tiene balance 0.

- Al borrar, baja el otro lado: el hijo pesado no se tocó.
- Sus dos subárboles miden lo mismo. **Va la simple.**

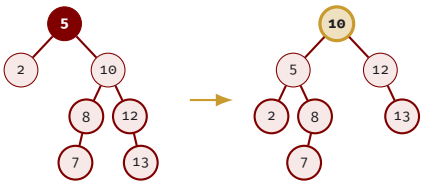
Primera diferencia: balance 0 en el hijo pesado



Ya sin el 1. El hijo 10 tiene balance 0.

- Al borrar, baja el otro lado: el hijo pesado no se tocó.
- Sus dos subárboles miden lo mismo. **Va la simple.**
- Con la doble subiría el 8, y el 10 quedaría con balance -2.

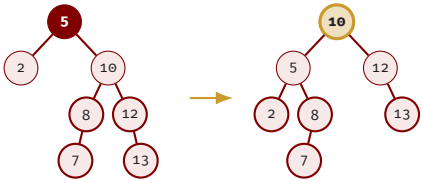
Primera diferencia: balance 0 en el hijo pesado



Ya sin el 1. El hijo 10 tiene balance 0.

- Al borrar, baja el otro lado: el hijo pesado no se tocó.
- Sus dos subárboles miden lo mismo. **Va la simple.**
- Con la doble subiría el 8, y el 10 quedaría con balance -2.
- En otros árboles la doble da por casualidad un AVL: no siempre se nota.

Primera diferencia: balance 0 en el hijo pesado



Ya sin el 1. El hijo 10 tiene balance 0.

- Al borrar, baja el otro lado: el hijo pesado no se tocó.
- Sus dos subárboles miden lo mismo. **Va la simple.**
- Con la doble subiría el 8, y el 10 quedaría con balance -2.
- En otros árboles la doble da por casualidad un AVL: no siempre se nota.

Error común

`balance_of(y) <= 0` en vez de `< 0`.

Segunda diferencia: la altura puede seguir bajando

Antes de borrar, x mide k + 2. El hijo pesado mide k + 1, y el otro lado baja de k a k − 1.

Balance del hijo pesado	Reparación	Altura tras reparar	Frente a antes
1 (mismo signo)	simple	k + 1	bajó
0	simple	k + 2	igual: se termina
−1 (signo opuesto)	doble	k + 1	bajó

Segunda diferencia: la altura puede seguir bajando

Antes de borrar, x mide k + 2. El hijo pesado mide k + 1, y el otro lado baja de k a k − 1.

Balance del hijo pesado	Reparación	Altura tras reparar	Frente a antes
1 (mismo signo)	simple	k + 1	bajó
0	simple	k + 2	igual: se termina
−1 (signo opuesto)	doble	k + 1	bajó

- Si bajó, el padre de x ve un lado más bajo: el mismo problema, un nivel más arriba.

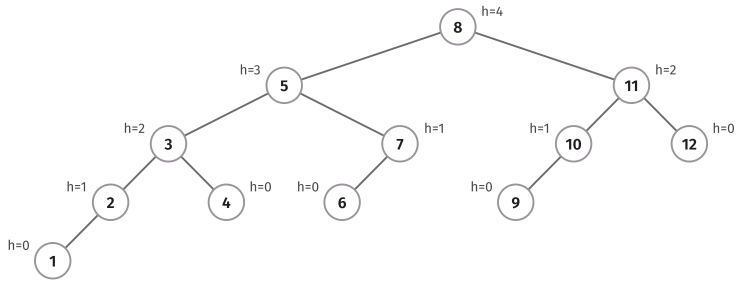
Segunda diferencia: la altura puede seguir bajando

Antes de borrar, x mide k + 2. El hijo pesado mide k + 1, y el otro lado baja de k a k − 1.

Balance del hijo pesado	Reparación	Altura tras reparar	Frente a antes
1 (mismo signo)	simple	k + 1	bajó
0	simple	k + 2	igual: se termina
−1 (signo opuesto)	doble	k + 1	bajó

- Si bajó, el padre de x ve un lado más bajo: el mismo problema, un nivel más arriba.
- Al insertar, la reparación devolvía la altura de antes. Aquí, solo en una de cada 4, más o menos.

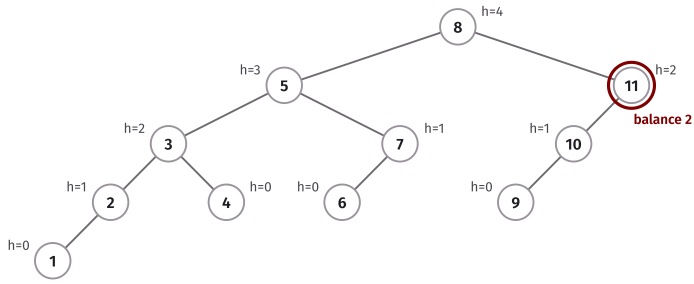
Una cascada: borrar 12 en un árbol de Fibonacci



El árbol de Fibonacci de altura 4. Vamos a borrar su clave mayor, el 12.

Sin el 12, el 11 queda con balance 2. Al repararlo, su subárbol baja de altura 2 a 1, y la raíz queda con balance 2: hay que reparar también. **Dos reparaciones**, en un árbol de altura 4.

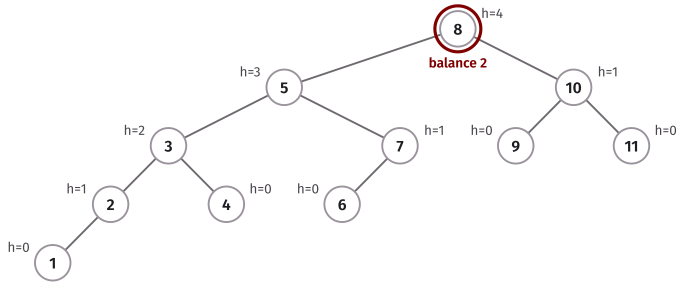
Una cascada: borrar 12 en un árbol de Fibonacci



Sale el 12, y el 11 queda con balance 2. Su hijo pesado, el 10, tiene 1: izquierda-izquierda.

Sin el 12, el 11 queda con balance 2. Al repararlo, su subárbol baja de altura 2 a 1, y la raíz queda con balance 2: hay que reparar también. **Dos reparaciones**, en un árbol de altura 4.

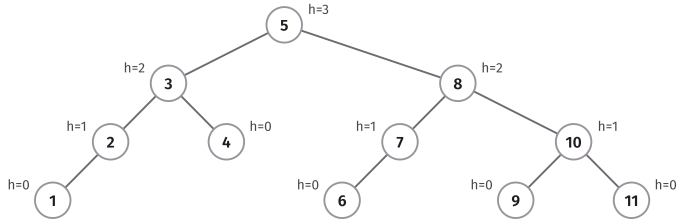
Una cascada: borrar 12 en un árbol de Fibonacci



Reparar en 11: su subárbol baja de altura 2 a 1, y ahora la raíz 8 queda con balance 2.

Sin el 12, el 11 queda con balance 2. Al repararlo, su subárbol baja de altura 2 a 1, y la raíz queda con balance 2: hay que reparar también. **Dos reparaciones**, en un árbol de altura 4.

Una cascada: borrar 12 en un árbol de Fibonacci



Reparar en 8: sube el 5. Dos reparaciones en un borrado, y el árbol queda de altura 3.

Sin el 12, el 11 queda con balance 2. Al repararlo, su subárbol baja de altura 2 a 1, y la raíz queda con balance 2: hay que reparar también. **Dos reparaciones**, en un árbol de altura 4.

¿Cuántas reparaciones puede hacer un borrado?

- Si se repara x, el hijo por donde se subió media 2 menos que x antes de borrar.

¿Cuántas reparaciones puede hacer un borrado?

- Si se repara x, el hijo por donde se subió medía 2 menos que x antes de borrar.
- **Cada reparación cuesta 2 de altura**, y no 1.

¿Cuántas reparaciones puede hacer un borrado?

- Si se repara x, el hijo por donde se subió medía 2 menos que x antes de borrar.
- **Cada reparación cuesta 2 de altura**, y no 1.
- Con k reparaciones, el último nodo reparado tiene altura de al menos 2k, que no supera a la del árbol.

¿Cuántas reparaciones puede hacer un borrado?

- Si se repara x, el hijo por donde se subió medía 2 menos que x antes de borrar.
- **Cada reparación cuesta 2 de altura**, y no 1.
- Con k reparaciones, el último nodo reparado tiene altura de al menos 2k, que no supera a la del árbol.

Máximo

Como máximo $\lfloor h/2 \rfloor$ reparaciones, con h la altura antes de borrar. Cada una es de 1 o 2 rotaciones: **O(log n)**.

¿Cuántas reparaciones puede hacer un borrado?

- Si se repara x, el hijo por donde se subió medía 2 menos que x antes de borrar.
- **Cada reparación cuesta 2 de altura**, y no 1.
- Con k reparaciones, el último nodo reparado tiene altura de al menos 2k, que no supera a la del árbol.

Máximo

Como máximo $\lfloor h/2 \rfloor$ reparaciones, con h la altura antes de borrar. Cada una es de 1 o 2 rotaciones: **O(log n)**.

"Una reparación en cada nivel" es cierto para **cada nivel de ese camino**. Y ese camino, en el peor caso, es corto: mide unos $h/2$. No hay un borrado con h reparaciones.

Medido

terminal

```
$ build/repair_counts
     operacion        n    rep/op    rot/op   max    h/2
insertar orden  1024000     1.000     1.000     1      9
 insertar azar  1024000     0.466     0.700     1     11
   borrar azar  1024000     0.266     0.373     5     11
Borrar la clave mayor de un arbol de Fibonacci
  h        n reparaciones rotaciones    h/2  h despues
  4       12            2          2      2          3
 10      232            5          5      5          9
 18    10945            9          9      9         17
 26   514228           13         13     13         25
```

Medido

terminal

```
$ build/repair_counts
     operacion        n    rep/op    rot/op   max    h/2
insertar orden  1024000     1.000     1.000     1      9
 insertar azar  1024000     0.466     0.700     1     11
   borrar azar  1024000     0.266     0.373     5     11
Borrar la clave mayor de un arbol de Fibonacci
  h        n reparaciones rotaciones    h/2  h despues
  4       12            2          2      2          3
 10      232            5          5      5          9
 18    10945            9          9      9         17
 26   514228           13         13     13         25
```

- Árboles de Fibonacci: exactamente $\lfloor h/2 \rfloor$. Ese máximo se alcanza.

Medido

terminal

```
$ build/repair_counts
     operacion        n    rep/op    rot/op   max    h/2
insertar orden  1024000     1.000     1.000     1      9
 insertar azar  1024000     0.466     0.700     1     11
   borrar azar  1024000     0.266     0.373     5     11
Borrar la clave mayor de un arbol de Fibonacci
  h        n reparaciones rotaciones    h/2  h despues
  4       12            2          2      2          3
 10      232            5          5      5          9
 18    10945            9          9      9         17
 26   514228           13         13     13         25
```

- Árboles de Fibonacci: exactamente $\lfloor h/2 \rfloor$. Ese máximo se alcanza.
- Al azar, menos de la mitad de las inserciones repara algo. Son medidas, con una semilla.

Medido

terminal

```
$ build/repair_counts
     operacion        n    rep/op    rot/op   max    h/2
insertar orden  1024000     1.000     1.000     1      9
 insertar azar  1024000     0.466     0.700     1     11
   borrar azar  1024000     0.266     0.373     5     11
Borrar la clave mayor de un arbol de Fibonacci
  h        n reparaciones rotaciones    h/2  h despues
  4       12            2          2      2          3
 10      232            5          5      5          9
 18    10945            9          9      9         17
 26   514228           13         13     13         25
```

- Árboles de Fibonacci: exactamente $\lfloor h/2 \rfloor$. Ese máximo se alcanza.
- Al azar, menos de la mitad de las inserciones repara algo. Son medidas, con una semilla.
- Al borrar se repara menos seguido, pero una vez llegó a 5.

Error común: “basta una”, también al borrar

terminal

```
$ build/stop_after_first_repair
5000 inserciones al azar: todas pasan check_invariants
antes de borrar 12: 8(5(3(2(1,_),4),7(6,_)),11(10(9,_),12))
despues de borrar 12: 8(5(3(2(1,_),4),7(6,_)),10(9,11))
```

Error común: “basta una”, también al borrar

terminal

```
$ build/stop_after_first_repair
5000 inserciones al azar: todas pasan check_invariants
antes de borrar 12: 8(5(3(2(1,_),4),7(6,_)),11(10(9,_),12))
despues de borrar 12: 8(5(3(2(1,_),4),7(6,_)),10(9,11))
```

- Y luego `check_invariants` detiene el programa.

Error común: “basta una”, también al borrar

terminal

```
$ build/stop_after_first_repair
5000 inserciones al azar: todas pasan check_invariants
antes de borrar 12: 8(5(3(2(1,_),4),7(6,_)),11(10(9,_),12))
despues de borrar 12: 8(5(3(2(1,_),4),7(6,_)),10(9,11))
```

- Y luego `check_invariants` detiene el programa.
- Pasa todas las inserciones. Pasa muchos borrados. El árbol de Fibonacci lo atrapa: nadie repara la raíz.

Error común: “basta una”, también al borrar

terminal

```
$ build/stop_after_first_repair
5000 inserciones al azar: todas pasan check_invariants
antes de borrar 12: 8(5(3(2(1,_),4),7(6,_)),11(10(9,_),12))
despues de borrar 12: 8(5(3(2(1,_),4),7(6,_)),10(9,11))
```

- Y luego `check_invariants` detiene el programa.
- Pasa todas las inserciones. Pasa muchos borrados. El árbol de Fibonacci lo atrapa: nadie repara la raíz.

Idea clave

Al insertar, una reparación devuelve la altura de antes. Al borrar, suele dejar el subárbol 1 más bajo. Las dos operaciones cuestan O(log n).

SECCIÓN 08

La invariante en las pruebas

08



Dos revisiones nuevas

avl_set.h

```
1  assert(node->height == 1 + std::max(height_of(node->left), height_of(node->right)));
2  assert(std::abs(balance_of(node)) <= 1);
```

Dos revisiones nuevas

avl_set.h

```
1  assert(node->height == 1 + std::max(height_of(node->left), height_of(node->right)));
2  assert(std::abs(balance_of(node)) <= 1);
```

- **La primera** detecta una altura que miente. Si cada nodo calza con sus hijos, todas son las reales: no hace falta volver a bajar.

Dos revisiones nuevas

avl_set.h

```
1  assert(node->height == 1 + std::max(height_of(node->left), height_of(node->right)));
2  assert(std::abs(balance_of(node)) <= 1);
```

- **La primera** detecta una altura que miente. Si cada nodo calza con sus hijos, todas son las reales: no hace falta volver a bajar.
- **La segunda** detecta un árbol de verdad desbalanceado. Sin la primera no sirve: con alturas falsas, un balance perfecto no quiere decir nada.

Dos revisiones nuevas

avl_set.h

```
1  assert(node->height == 1 + std::max(height_of(node->left), height_of(node->right)));
2  assert(std::abs(balance_of(node)) <= 1);
```

- **La primera** detecta una altura que miente. Si cada nodo calza con sus hijos, todas son las reales: no hace falta volver a bajar.
- **La segunda** detecta un árbol de verdad desbalanceado. Sin la primera no sirve: con alturas falsas, un balance perfecto no quiere decir nada.
- Un error aquí no se nota enseguida: el orden no se rompe, solo la promesa de ser bajo. Por eso se llama tras **cada** operación.

Rotar y no actualizar las alturas

terminal

```
$ build/no_height_update
con 1, 2, 3: 2(1,3), altura guardada 1
con 1000 claves ordenadas: altura real 17, altura guardada 24
```

Rotar y no actualizar las alturas

terminal

```
$ build/no_height_update
con 1, 2, 3: 2(1,3), altura guardada 1
con 1000 claves ordenadas: altura real 17, altura guardada 24
```

- Con 3 claves, la forma es correcta. La raíz guarda 1 por casualidad, y la hoja 1 sigue guardando 2.

Rotar y no actualizar las alturas

terminal

```
$ build/no_height_update
con 1, 2, 3: 2(1,3), altura guardada 1
con 1000 claves ordenadas: altura real 17, altura guardada 24
```

- Con 3 claves, la forma es correcta. La raíz guarda 1 por casualidad, y la hoja 1 sigue guardando 2.
- Con 1000 claves, debía ser 9. Las decisiones se toman con números falsos.

Rotar y no actualizar las alturas

terminal

```
$ build/no_height_update
con 1, 2, 3: 2(1,3), altura guardada 1
con 1000 claves ordenadas: altura real 17, altura guardada 24
```

- Con 3 claves, la forma es correcta. La raíz guarda 1 por casualidad, y la hoja 1 sigue guardando 2.
- Con 1000 claves, debía ser 9. Las decisiones se toman con números falsos.
- Sigue siendo un árbol de búsqueda: todas las búsquedas responden bien. Una prueba que solo mire el contenido no lo ve.

Actualizar primero al padre

terminal

```
$ build/parent_before_child
con 1, 2, 3: 2(1,3), altura real 1, altura guardada 3
con 1000 claves ordenadas: altura real 13, altura guardada 15
```

Actualizar primero al padre

terminal

```
$ build/parent_before_child
con 1, 2, 3: 2(1,3), altura real 1, altura guardada 3
con 1000 claves ordenadas: altura real 13, altura guardada 15
```

- Sube el 2, y baja el 1. El 1 todavía guarda altura 2, de cuando era raíz de la cadena.

Actualizar primero al padre

terminal

```
$ build/parent_before_child
con 1, 2, 3: 2(1,3), altura real 1, altura guardada 3
con 1000 claves ordenadas: altura real 13, altura guardada 15
```

- Sube el 2, y baja el 1. El 1 todavía guarda altura 2, de cuando era raíz de la cadena.
- El 2 se calcula primero: $1 + \text{máx}(2, 0) = 3$, con la altura vieja de su hijo.

Actualizar primero al padre

terminal

```
$ build/parent_before_child
con 1, 2, 3: 2(1,3), altura real 1, altura guardada 3
con 1000 claves ordenadas: altura real 13, altura guardada 15
```

- Sube el 2, y baja el 1. El 1 todavía guarda altura 2, de cuando era raíz de la cadena.
- El 2 se calcula primero: $1 + \text{máx}(2, 0) = 3$, con la altura vieja de su hijo.
- Luego el 1 se corrige a 0. Tarde.

Actualizar primero al padre

terminal

```
$ build/parent_before_child
con 1, 2, 3: 2(1,3), altura real 1, altura guardada 3
con 1000 claves ordenadas: altura real 13, altura guardada 15
```

- Sube el 2, y baja el 1. El 1 todavía guarda altura 2, de cuando era raíz de la cadena.
- El 2 se calcula primero: $1 + \text{máx}(2, 0) = 3$, con la altura vieja de su hijo.
- Luego el 1 se corrige a 0. Tarde.

Idea clave

Toda altura se calcula con las de sus hijos: de abajo hacia arriba, en la subida y dentro de una rotación.

SECCIÓN 09

Cuánto se gana

09

La altura: no depende de la máquina

terminal

```
Claves ordenadas
        n  log2n 1.44log   avl  sin.bal
     1000   10.0    14.4     9      999
    16000   14.0    20.1    13    15999
  1024000   20.0    28.8    19        -
Claves al azar, tres semillas
        n  log2n 1.44log   a1   a2   a3 prof.avl   s1   s2   s3 prof.sin
     1000   10.0    14.4   11   11   11     8.20   22   19   23    11.12
  1024000   20.0    28.8   23   23   23    18.35   49   51   49    24.73
```

La altura: no depende de la máquina

terminal

```
Claves ordenadas
        n  log2n 1.44log   avl  sin.bal
     1000   10.0    14.4     9      999
    16000   14.0    20.1    13    15999
  1024000   20.0    28.8    19        -
Claves al azar, tres semillas
        n  log2n 1.44log   a1   a2   a3 prof.avl   s1   s2   s3 prof.sin
     1000   10.0    14.4   11   11   11     8.20   22   19   23    11.12
  1024000   20.0    28.8   23   23   23    18.35   49   51   49    24.73
```

- **Ordenadas:** altura medida $\lfloor \log_2 n \rfloor$, la mínima. Era el peor caso sin balance.

La altura: no depende de la máquina

terminal

```
Claves ordenadas
        n  log2n 1.44log   avl  sin.bal
     1000   10.0    14.4     9      999
    16000   14.0    20.1    13    15999
  1024000   20.0    28.8    19        -
Claves al azar, tres semillas
        n  log2n 1.44log   a1   a2   a3 prof.avl   s1   s2   s3 prof.sin
     1000   10.0    14.4   11   11   11     8.20   22   19   23    11.12
  1024000   20.0    28.8   23   23   23    18.35   49   51   49    24.73
```

- **Ordenadas:** altura medida $\lfloor \log_2 n \rfloor$, la mínima. Era el peor caso sin balance.
- **Al azar:** entre 1.1 y 1.2 veces $\log_2 n$ en esta tabla. La profundidad media, unos 25 % menor.

La altura: no depende de la máquina

terminal

```
Claves ordenadas
        n  log2n 1.44log   avl  sin.bal
     1000   10.0    14.4     9      999
    16000   14.0    20.1    13    15999
  1024000   20.0    28.8    19        -
Claves al azar, tres semillas
        n  log2n 1.44log   a1   a2   a3 prof.avl   s1   s2   s3 prof.sin
     1000   10.0    14.4   11   11   11     8.20   22   19   23    11.12
  1024000   20.0    28.8   23   23   23    18.35   49   51   49    24.73
```

- **Ordenadas:** altura medida $\lfloor \log_2 n \rfloor$, la mínima. Era el peor caso sin balance.
- **Al azar:** entre 1.1 y 1.2 veces $\log_2 n$ en esta tabla. La profundidad media, unos 25 % menor.
- Nada llega a 1.44 $\log_2 n$, salvo un árbol de Fibonacci armado a propósito.

El reloj: lo que se gana, y lo que no

terminal

```
$ build/avl_times
Claves ordenadas: insertar (s)
        n       avl   sin.bal        x
     4000   0.00020   0.01432     70.1
     8000   0.00046   0.06139    134.6
    16000   0.00095   0.24263    254.9
Claves al azar: insertar y buscar todas (s)
        n   ins.avl   ins.sin  avl/sin   bus.avl   bus.sin  sin/avl
    16000   0.00270   0.00184     1.47   0.00150   0.00150     1.00
   128000   0.03328   0.02897     1.15   0.02228   0.02268     1.02
   512000   0.39233   0.26606     1.47   0.18000   0.18255     1.01
```

El reloj: lo que se gana, y lo que no

terminal

```
$ build/avl_times
Claves ordenadas: insertar (s)
        n       avl   sin.bal        x
     4000   0.00020   0.01432     70.1
     8000   0.00046   0.06139    134.6
    16000   0.00095   0.24263    254.9
Claves al azar: insertar y buscar todas (s)
        n   ins.avl   ins.sin  avl/sin   bus.avl   bus.sin  sin/avl
    16000   0.00270   0.00184     1.47   0.00150   0.00150     1.00
   128000   0.03328   0.02897     1.15   0.02228   0.02268     1.02
   512000   0.39233   0.26606     1.47   0.18000   0.18255     1.01
```

- **Ordenadas:** la razón x se duplica, aproximadamente, en cada fila.

El reloj: lo que se gana, y lo que no

terminal

```
$ build/avl_times
Claves ordenadas: insertar (s)
        n       avl   sin.bal        x
     4000   0.00020   0.01432     70.1
     8000   0.00046   0.06139    134.6
    16000   0.00095   0.24263    254.9
Claves al azar: insertar y buscar todas (s)
        n   ins.avl   ins.sin  avl/sin   bus.avl   bus.sin  sin/avl
    16000   0.00270   0.00184     1.47   0.00150   0.00150     1.00
   128000   0.03328   0.02897     1.15   0.02228   0.02268     1.02
   512000   0.39233   0.26606     1.47   0.18000   0.18255     1.01
```

- **Ordenadas:** la razón x se duplica, aproximadamente, en cada fila.
- **Al azar, el AVL no gana.** Insertar costó entre 1.1 y 1.6 veces lo de sin balance, y buscar salió empatado, aunque compare menos. No sabemos por qué.

El reloj: lo que se gana, y lo que no

terminal

```
$ build/avl_times
Claves ordenadas: insertar (s)
        n       avl   sin.bal        x
     4000   0.00020   0.01432     70.1
     8000   0.00046   0.06139    134.6
    16000   0.00095   0.24263    254.9
Claves al azar: insertar y buscar todas (s)
        n   ins.avl   ins.sin  avl/sin   bus.avl   bus.sin  sin/avl
    16000   0.00270   0.00184     1.47   0.00150   0.00150     1.00
   128000   0.03328   0.02897     1.15   0.02228   0.02268     1.02
   512000   0.39233   0.26606     1.47   0.18000   0.18255     1.01
```

- **Ordenadas:** la razón x se duplica, aproximadamente, en cada fila.
- **Al azar, el AVL no gana.** Insertar costó entre 1.1 y 1.6 veces lo de sin balance, y buscar salió empatado, aunque compare menos. No sabemos por qué.
- Lo que compra es **una garantía**. Con -O2, mediana de 3. Dependen de la máquina.

El caso de la semana pasada

terminal

```
$ build/dictionary_avl ../../data/palabras.txt
orden de bytes: si
AVL, tal como vienen: 1045757 palabras, altura 19, rotaciones 1045737, 0.196784 s
std::set, tal como vienen: 1045757 palabras, 0.161904 s
AVL, barajadas: 1045757 palabras, altura 23, rotaciones 729058, 1.30662 s
std::set, barajadas: 1045757 palabras, 1.29662 s
sin balance, barajadas: 1045757 palabras, altura 49, 1.51672 s
```

El caso de la semana pasada

terminal

```
$ build/dictionary_avl ../../data/palabras.txt
orden de bytes: si
AVL, tal como vienen: 1045757 palabras, altura 19, rotaciones 1045737, 0.196784 s
std::set, tal como vienen: 1045757 palabras, 0.161904 s
AVL, barajadas: 1045757 palabras, altura 23, rotaciones 729058, 1.30662 s
std::set, barajadas: 1045757 palabras, 1.29662 s
sin balance, barajadas: 1045757 palabras, altura 49, 1.51672 s
```

- El archivo completo, en el orden malo: altura 19, y una fracción de segundo.

El caso de la semana pasada

terminal

```
$ build/dictionary_avl ../../data/palabras.txt
orden de bytes: si
AVL, tal como vienen: 1045757 palabras, altura 19, rotaciones 1045737, 0.196784 s
std::set, tal como vienen: 1045757 palabras, 0.161904 s
AVL, barajadas: 1045757 palabras, altura 23, rotaciones 729058, 1.30662 s
std::set, barajadas: 1045757 palabras, 1.29662 s
sin balance, barajadas: 1045757 palabras, altura 49, 1.51672 s
```

- El archivo completo, en el orden malo: altura 19, y una fracción de segundo.
- La semana pasada estimamos cerca de una hora sin balance.

El caso de la semana pasada

terminal

```
$ build/dictionary_avl ../../data/palabras.txt
orden de bytes: si
AVL, tal como vienen: 1045757 palabras, altura 19, rotaciones 1045737, 0.196784 s
std::set, tal como vienen: 1045757 palabras, 0.161904 s
AVL, barajadas: 1045757 palabras, altura 23, rotaciones 729058, 1.30662 s
std::set, barajadas: 1045757 palabras, 1.29662 s
sin balance, barajadas: 1045757 palabras, altura 49, 1.51672 s
```

- El archivo completo, en el orden malo: altura 19, y una fracción de segundo.
- La semana pasada estimamos cerca de una hora sin balance.
- En esta tabla, `std::set` y `AvlSet` salen parecidos. La semana 12 los compara con cuidado.

SECCIÓN 10

El laboratorio

10

Una sesión: balancear el mapa

La otra sesión es para la **Práctica 3**.

terminal

```
make test-base
make test-rotations
make test-insert
make test-erase
make test-lab
make bench
```

Una sesión: balancear el mapa

La otra sesión es para la **Práctica 3**.

1. Altura y rotaciones, con pruebas directas. 30 min

terminal

```
make test-base
make test-rotations
make test-insert
make test-erase
make test-lab
make bench
```

Una sesión: balancear el mapa

La otra sesión es para la **Práctica 3**.

1. Altura y rotaciones, con pruebas directas. 30 min
2. Inserción con balance, y `check_invariants`. 40 min

terminal

```
make test-base
make test-rotations
make test-insert
make test-erase
make test-lab
make bench
```

Una sesión: balancear el mapa

La otra sesión es para la **Práctica 3**.

1. Altura y rotaciones, con pruebas directas. 30 min
2. Inserción con balance, y `check_invariants`. 40 min
3. Borrado con balance, contra `std::map`. 25 min

terminal

```
make test-base
make test-rotations
make test-insert
make test-erase
make test-lab
make bench
```

Una sesión: balancear el mapa

La otra sesión es para la **Práctica 3**.

1. Altura y rotaciones, con pruebas directas. 30 min
2. Inserción con balance, y `check_invariants`. 40 min
3. Borrado con balance, contra `std::map`. 25 min
4. `make bench` y `lab/report.md`. 15 min

terminal

```
make test-base
make test-rotations
make test-insert
make test-erase
make test-lab
make bench
```

Una sesión: balancear el mapa

La otra sesión es para la **Práctica 3**.

1. Altura y rotaciones, con pruebas directas. 30 min
2. Inserción con balance, y `check_invariants`. 40 min
3. Borrado con balance, contra `std::map`. 25 min
4. `make bench` y `lab/report.md`. 15 min

Viene hecho: el mapa de la semana 10, completo y funcionando, con `height` en cada nodo; `height_of`, `balance_of`, y el programa que mide. Si prefieren su `BstMap`, pueden usarlo: `make test-base` les dice si sigue sano.

terminal

```
make test-base
make test-rotations
make test-insert
make test-erase
make test-lab
make bench
```

Las rotaciones se quedan

lab/avl_map.h

```
1  Node* rotate_left(Node*) { not_implemented("rotate_left"); }
2  Node* rotate_right(Node*) { not_implemented("rotate_right"); }
```

Las rotaciones se quedan

lab/avl_map.h

```
1  Node* rotate_left(Node*) { not_implemented("rotate_left"); }
2  Node* rotate_right(Node*) { not_implemented("rotate_right"); }
```

- Reciben el nodo que baja, y devuelven el que sube.

Las rotaciones se quedan

lab/avl_map.h

```
1  Node* rotate_left(Node*) { not_implemented("rotate_left"); }
2  Node* rotate_right(Node*) { not_implemented("rotate_right"); }
```

- Reciben el nodo que baja, y devuelven el que sube.
- Arreglan enlaces, padres y `root_`, y suman 1 a `rotations_`.

Las rotaciones se quedan

lab/avl_map.h

```
1  Node* rotate_left(Node*) { not_implemented("rotate_left"); }
2  Node* rotate_right(Node*) { not_implemented("rotate_right"); }
```

- Reciben el nodo que baja, y devuelven el que sube.
- Arreglan enlaces, padres y `root_`, y suman 1 a `rotations_`.
- **No tocan las alturas.** La semana 12 las usa tal como están, en un árbol sin alturas.

Las rotaciones se quedan

lab/avl_map.h

```
1  Node* rotate_left(Node*) { not_implemented("rotate_left"); }
2  Node* rotate_right(Node*) { not_implemented("rotate_right"); }
```

- Reciben el nodo que baja, y devuelven el que sube.
- Arreglan enlaces, padres y `root_`, y suman 1 a `rotations_`.
- **No tocan las alturas.** La semana 12 las usa tal como están, en un árbol sin alturas.
- Llamen a `rebalance_up` en `insert_node`, y no en `insert`: `operator[]` también inserta.

Resumen

- **Condición AVL:** en cada nodo, las alturas de los subárboles difieren en 1 como máximo.
- **Altura:** $N(h) = N(h-1) + N(h-2) + 1$ da $h \leq 2\log_2 n$, y exacto, $h < 1.4405\log_2(n+2) - 1.3277$.
- **Rotación:** O(1), cambia la forma y conserva el recorrido en orden.
- **Cuatro casos:** signos iguales, simple; opuestos, doble. Con balance 0, la simple.
- **Insertar:** basta una reparación, que devuelve la altura de antes. La subida es O(log n).
- **Borrar:** hasta $\lfloor h/2 \rfloor$ reparaciones, O(log n).
- **Alturas:** de abajo hacia arriba. Tras rotar, primero el que bajó.

Para la próxima clase

- Leer los apuntes de la semana 12: árboles rojinegros.
- Laboratorio: `AvlMap`. Guarden sus rotaciones: la semana 12 parte de ellas.
- Práctica 3 en la otra sesión: tablas hash y árboles binarios de búsqueda.

Preguntas de repaso

¿Cuántas reparaciones puede necesitar una inserción? ¿Y un borrado? ¿Cuál es la menor cantidad de nodos de un AVL de altura 4?
Y para pensar: ¿qué pasaría si un nodo pudiera tener más de una clave?