

Estructuras de Datos

Árboles rojinegros

De los árboles 2-3-4 a la inserción

Gerson Garrido

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Nodos con más de una clave: árboles 2-3-4
2. Insertar: dividir un nodo lleno
3. La idea de los B-trees
4. El nodo 2-3-4 escrito en binario: negro con hijos rojos
5. Las cinco propiedades y la altura
6. Insertar en un rojinegro, caso por caso
7. ¿De verdad son el mismo árbol? Lo comprobamos
8. La eliminación, solo la idea
9. AVL, rojinegro y `std::map`, medidos

> **Lectura**
>
> Apuntes de la semana 12. Open Data Structures, secs. 9.1 y 9.2 (usa la variante inclinada a la izquierda). CLRS, *Red-Black Trees*, y *B-Trees* solo para la idea.

SECCIÓN 01

Árboles 2-3-4

01



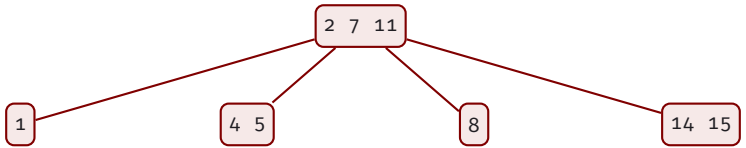
Otra forma de balancear

- Un AVL permite hojas a distintas profundidades, y limita la diferencia.
- Pidamos algo que parece imposible: **todas las hojas a la misma profundidad**, siempre.
- Con nodos binarios no se puede. Pero con **más de una clave por nodo**, una clave nueva puede entrar en un nodo que ya existe.

Árbol 2-3-4 *(2-3-4 tree)*

Árbol de búsqueda en el que cada nodo tiene 1, 2 o 3 claves; un nodo interno con k claves tiene $k + 1$ hijos; y **todas las hojas están a la misma profundidad**.

Cómo se ve




```
[2 7 11]([1],[4 5],[8],[14 15])
```

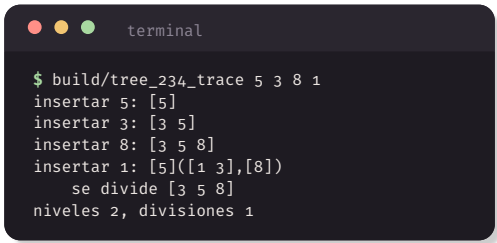
- L niveles, y cada nodo interno con al menos 2 hijos:

$$n \geq 1 + 2 + 4 + \cdots + 2^{L-1} = 2^L - 1 \quad \Rightarrow \quad L \leq \log_2(n+1)$$

- Una condición simple, sin alturas guardadas. Todo el problema es mantenerla.

Insertar: dividir un nodo lleno

- Una clave nueva **siempre entra en una hoja que ya existe**.
- Si tiene lugar, listo.
- Si está llena, se **divide**: la del medio de sus 3 claves sube al padre, y la nueva entra en su mitad.
- Eso puede llenar al padre: el problema sube.
- Si se divide la raíz, el árbol gana un nivel. **Es la única forma en que crece.**



Idea clave

El árbol crece **desde arriba**: por eso las hojas siguen a la misma profundidad.

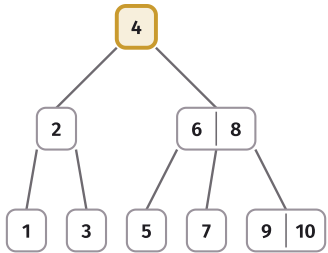
Una cascada

terminal

```
$ build/tree_234_trace 1 2 3 4 5 6 7 8 9 10 | tail -7
insertar 8: [2 4 6]([1],[3],[5],[7 8])
    se divide [5 6 7]
insertar 9: [2 4 6]([1],[3],[5],[7 8 9])
insertar 10: [4]([2]([1],[3]),[6 8]([5],[7],[9 10]))
    se divide [7 8 9]
    se divide [2 4 6]
niveles 3, divisiones 5
```

- Al insertar el 10: se divide la hoja, el 8 sube, la raíz está llena, se divide, y aparece una raíz nueva.

La cascada, dibujada



Se divide [2 4 6]: el 4 sube a una raíz nueva. Todas las hojas bajan juntas

Hay más de una regla de división

1. **Al hacer overflow.** Solo si hace falta. Sube la del medio de las 3 que tenía el nodo. La de este curso.
2. **Juntar las 4 claves**, y subir siempre la segunda. Es como lo describe Open Data Structures, sec. 9.1.
3. **Al bajar.** Dividir todo nodo lleno que se encuentre, haga falta o no. Es como lo hace CLRS en los B-trees.

> **Importa**
>
> Las tres dan árboles 2-3-4 correctos y cuestan O(log n). Pero no dan **el mismo** árbol. Volveremos a esto.

SECCIÓN 02

La idea de los B-trees

02



B-trees: la misma idea, con nodos enormes

- Un disco entrega **páginas** enteras, de unos pocos KB.
- Leer una página cuesta mucho más que comparar dentro de una que ya está en memoria.
- Se cuenta entonces **páginas leídas**: una por nivel.
- Un nodo del tamaño de una página, con cientos de claves. Mil millones de claves: 3 niveles, no 30.



Bases de Datos

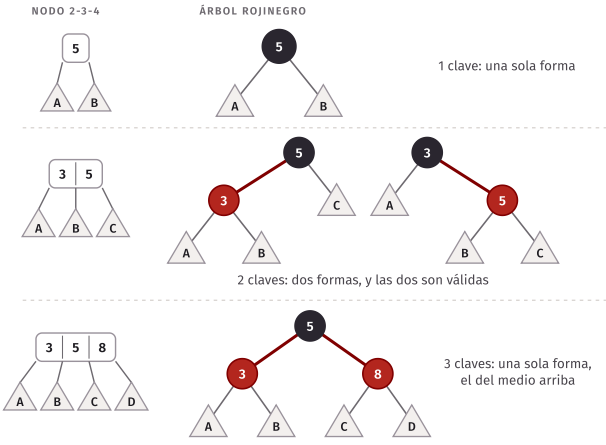
Los construye de verdad. Aquí nos quedamos con la idea: lo que se ahorra son lecturas de disco, no comparaciones.

SECCIÓN 03

Del 2-3-4 al rojinegro

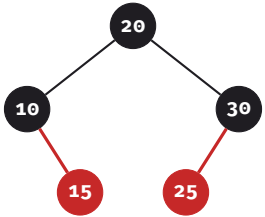
03

La idea central: rojo es “soy parte de mi padre”

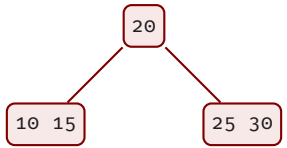


- Cada nodo 2-3-4 es un **negro** arriba, con sus demás claves como **hijos rojos**.
- Una cadena negro-rojo-rojo **no** es una forma de escribir 3 claves: es un rojo con hijo rojo.

Colapsar



`20(10(_,15r),30(25r,_))`



`[20]([10 15],[25 30])`

- **Colapsar:** juntar cada negro con sus hijos rojos. Reaparece el árbol 2-3-4.
- Fíjense: `10` y `30` son nodos de 2 claves, y están escritos de formas **distintas**.
- Un rojinegro colapsa a **exactamente uno**. Un 2-3-4 con t nodos de 2 claves viene de 2^t rojinegros.

Lo comprobamos: se generan todos los rojinegros válidos

terminal

```
$ build/all_shapes
  n rojinegros  2-3-4  suma.2^t  alcanzables h.todos  h.ins   cota
  1          1      1         1            1       0      0   1.00
  2          2      1         2            2       1      1   2.17
  3          2      2         2            1       1      1   3.00
  4          4      2         4            4       2      2   3.64
  5          8      4         8            6       2      2   4.17
  6         16      5        16           12       3      3   4.61
  7         33      9        33           29       3      3   5.00
  8         56     15        56           48       3      3   5.34
  9         90     28        90           58       3      3   5.64
```

- `rojinegros` y `suma.2^t` coinciden, siempre.
- **No todo árbol válido se puede obtener insertando:** `alcanzables` es menor. Con 3 claves, `20(10,30)` es válido, y la inserción no lo produce.

Este curso usa el árbol clásico

Clásico

CLRS, la biblioteca del curso, estos apuntes.

- Dos formas de nodo de 2 claves.
- Un 2-3-4 tiene varios rojinegros.
- `10(_,15r)` es correcto.

Inclinado a la izquierda

Open Data Structures, sec. 9.2.

- Una sola forma: el rojo a la izquierda.
- Un 2-3-4 tiene un único rojinegro.
- `10(_,15r)` sería un error.

No los mezclen

Lean Open Data Structures para el puente con los árboles 2-4. Pero sus casos de inserción no son los nuestros.

Las cinco propiedades

Propiedad	En el árbol 2-3-4
1. Rojo o negro	Parte de arriba de un nodo, o una de sus otras claves
2. La raíz es negra	Arriba de ella no hay nadie a quien pertenecer
3. Las hojas nulas son negras	Marcan donde se acaba el árbol
4. Un rojo solo tiene hijos negros	Un nodo tiene como máximo un negro y 2 rojos: 3 claves
5. Desde cada nodo, misma cantidad de negros hasta sus hojas nulas	**Todas las hojas a la misma profundidad**

- Una **hoja nula** es cada `nullptr`. No ocupa memoria: `is_red` dice que no es rojo.
- Contar negros **es** contar niveles. Esa cantidad es la **altura negra** b.

Altura máxima: dos cuentas, sin inducción

1. b es pequeño

Colapsado, son b niveles, con al menos 2^i nodos en el nivel i: $n \geq 2^b - 1$, es decir, $b \leq \log_2(n+1)$.

2. Un camino tiene como máximo $2b$ nodos

La raíz es negra, y cada rojo va justo después de un negro distinto: como máximo b rojos.

Resultado

Contando aristas: $h \leq 2b - 1 \leq 2\log_2(n+1) - 1$
La forma usual, $h \leq 2\log_2(n+1)$, cuenta también la arista a la hoja nula.

El camino más largo y el más corto, desde un mismo nodo

Dos caminos desde x hasta hojas nulas pasan por la misma cantidad de negros, b_x.

- x **negro:** el corto tiene al menos b_x nodos. El largo, como máximo $2b_x$, alternando.
- x **rojo:** se cuenta a sí mismo en los dos. Al menos $b_x + 1$, y como máximo $2b_x + 1 \leq 2(b_x + 1)$.

Respuesta

Desde un mismo nodo, un camino hasta una hoja nula tiene como máximo el doble de nodos que otro. Solo los rojos hacen la diferencia, y no puede haber más rojos que negros, salvo uno si empieza en rojo.

¿Qué tan cerca se llega?

terminal

```
$ build/height_bound
  h  min.nodos negros     cota h/log2n
  0          1      1     1.00    0.00
  1          2      1     2.17    1.00
  2          4      2     3.64    1.00
  3          6      2     4.61    1.16
  4         10      3     5.92    1.20
  6         22      4     8.05    1.35
  8         46      5    10.11    1.45
 10         94      6    12.14    1.53
 12        190      7    14.15    1.59
 16        766      9    18.17    1.67
 20       3070     11    22.17    1.73
 24      12286     13    26.17    1.77
 32     196606     17    34.17    1.82
 40    3145726     21    42.17    1.85
```

- El rojinegro **válido** con menos nodos para cada altura.
- La cota queda a unas 2 unidades de h.
- `h/log2n` sube despacio hacia 2.

La constante

El 2 no se puede bajar, si solo se sabe que cumple las propiedades. Un AVL tiene 1.44.

SECCIÓN 04

Insertar

04

¿De qué color es el nodo nuevo?

Si fuera negro

Ese camino tiene un negro de más: la propiedad 5 se rompe en **todos** los antepasados.

En 2-3-4: un nodo nuevo debajo de una hoja. Justo lo que no se hace.

Es rojo

La 5 no cambia. Solo se puede romper la 4, si su padre es rojo, o la 2, si es la raíz. Son problemas **locales**.

En 2-3-4: una clave que entra en un nodo que ya existe.

- Llamamos *z* al rojo en conflicto. Si su padre es rojo, ese padre no es la raíz: tiene **abuelo**, negro.
- El **tío** es el otro hijo del abuelo. **Un tío nulo cuenta como negro.**

La inserción, en pseudocódigo

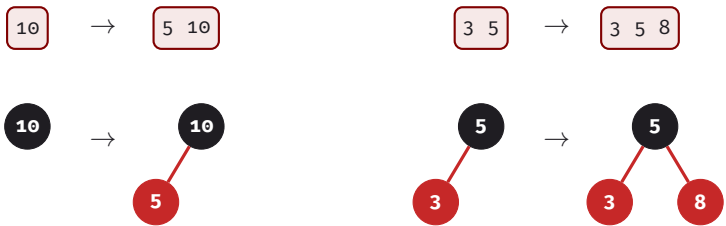
pseudocódigo

```
algoritmo reparar_insercion(arbol, z)
  mientras el padre de z es rojo hacer
    padre ← padre de z
    abuelo ← padre de padre
    si padre es hijo izq. de abuelo entonces
      tio ← hijo der. de abuelo
      si tio es rojo entonces
        pintar de negro a padre y a tio
        pintar de rojo a abuelo
        z ← abuelo
```

pseudocódigo

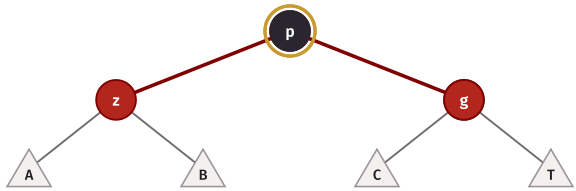
```
sino
        si z es hijo der. de padre entonces
          z ← padre
          padre ← rotar_izquierda(arbol, z)
        fin si
        pintar de negro a padre
        pintar de rojo a abuelo
        rotar_derecha(arbol, abuelo)
      fin si
    sino
      el espejo: cambiar izquierdo por derecho
    fin si
  fin mientras
  pintar de negro a la raiz de arbol
fin algoritmo
```

Padre negro: la clave entró en un nodo con lugar



- No se rompe ninguna propiedad. No hay nada que hacer.
- A la derecha: un nodo de 2 claves pasa a 3 sin ningún arreglo, porque el nuevo cayó del lado libre del negro.

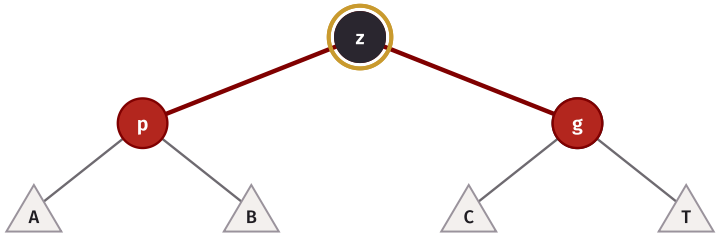
Tío negro, en línea: una rotación



Rotar a la derecha en g y cambiar los dos colores. Un negro con dos hijos rojos

- En 2-3-4, el grupo tiene 2 claves, g y p: el tío no pertenece a él. La nueva tiene lugar: **pasa a 3 claves**, y ningún nodo se divide.
- El árbol 2-3-4 ya quedó bien. Lo que está mal es cómo está escrito: negro-rojo-rojo.
- El tío es la raíz de T, y puede ser nulo: en la primera vuelta del ciclo, si es negro, es nulo.

Tío negro, en zigzag: dos rotaciones



Rotar a la derecha en g y cambiar los colores: z queda arriba, negro

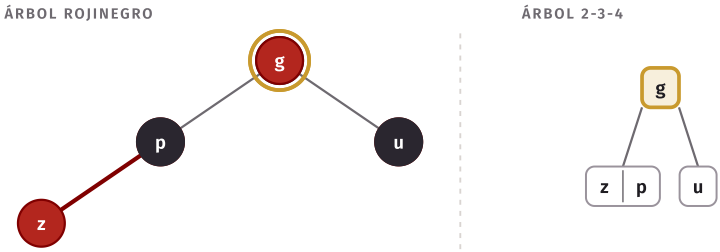
En zigzag: lo que hay que recordar

- En 2-3-4, lo mismo que antes: un nodo de 2 claves pasa a 3. El del medio es z, y sube.

Error común

Olvidar `z ← padre`. Tras rotar, el z que quedó es el negro que subió, su padre puede ser rojo, y el ciclo sigue arreglando lo que no estaba roto. En árboles pequeños no se nota.

Tío rojo: un nodo lleno se divide, solo con colores



Tres colores, sin mover enlaces: el nodo se divide. g es rojo y sube: z ← g

- Subir g es como insertarlo arriba: g es ahora un rojo que puede tener padre rojo. **El problema sube**.
- Cada vuelta sube 2 niveles: como máximo $\lfloor h/2 \rfloor$ recoloreos.

Al final: pintar la raíz de negro

- Si el ciclo llegó a la raíz recoloreando, la raíz quedó roja.
- Se pinta de negro. **Es dividir la raíz del árbol 2-3-4.**
- Es el único momento en que la altura negra aumenta: el árbol 2-3-4 gana un nivel.

Error común

Olvidarlo. Las búsquedas responden bien, de vez en cuando la raíz queda roja, y en el mapa con puntero al padre el ciclo busca un abuelo que no existe.

La propiedad 4 rota

Un rojo con hijo rojo es un estado de paso: un nodo lleno que no puede recibir otra clave, si el tío es rojo, o un nodo de 3 claves bien formado y mal escrito, si es negro.

La tabla de casos

Rojinegro	Grupo	En 2-3-4	Trabajo
Padre negro	1 o 2 claves	La clave entra	Nada
Tío negro, en línea	2 claves	Pasa a 3 claves	1 rotación, 2 colores
Tío negro, zigzag	2 claves	Pasa a 3 claves	2 rotaciones, 2 colores
Tío rojo	3 claves	**Se divide**, la del medio sube	3 colores. Sube
Raíz roja al final	La raíz se dividió	**Gana un nivel**	1 color

Como máximo 2 rotaciones

Tío rojo no rota. Con tío negro, se rota 1 o 2 veces, y el que queda arriba es negro: el ciclo termina. Comprobado en todos los órdenes de hasta 9 claves.

Paso a paso: 10, 20, 30, 15, 25

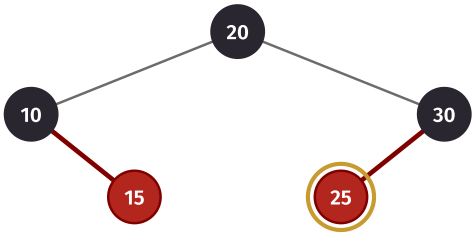
terminal

```
$ build/rb_trace 10 20 30 15 25
insertar 10: 10
    en 10: raiz roja: pintar de negro
    2-3-4: [10]
insertar 20: 10(_,20r)
    2-3-4: [10 20]
insertar 30: 20(10r,30r)
    en 10: tio negro, en linea: una rotacion
    2-3-4: [10 20 30]
insertar 15: 20(10(_,15r),30)
    en 20: tio rojo: recolorear
    en 20: raiz roja: pintar de negro
    2-3-4: [20]([10 15],[30])
insertar 25: 20(10(_,15r),30(25r,_))
    2-3-4: [20]([10 15],[25 30])
altura 2, negros 2, rotaciones 1, recoloreos 1
```

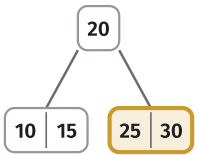
- 30: tío nulo, en línea, 1 rotación.
- 15: `[10 20 30]` se divide, y es la raíz: el árbol 2-3-4 gana un nivel.
- 25: su padre es negro. Nada.
- Al final aparecen las **dos** formas de un nodo de 2 claves.

La misma inserción, en los dos árboles

ÁRBOL ROJINEGRO



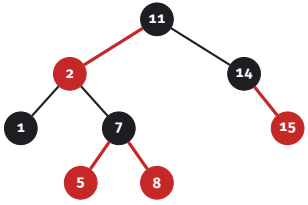
AL COLAPSARLO: ÁRBOL 2-3-4



Insertar 25: su padre es negro. Las dos formas de un nodo de 2 claves

Un caso con todo: el problema sube

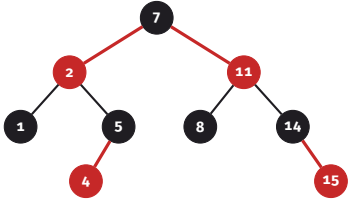
Antes de insertar el 4:



`[2 11]([1],[5 7 8],[14 15])`

→

Después:



`[2 7 11]([1],[4 5],[8],[14 15])`

- 4 llega bajo 5. Tío 8, rojo: `[5 7 8]` se divide. 7 sube, rojo.
- Ahora z es 7. Padre 2 rojo. **Tío 14, negro de verdad, no nulo**: `[2 11]` pasa a `[2 7 11]`.
- Zigzag: rotar en 2, rotar en 11. El del medio, 7, queda arriba, negro, con 2 hijos rojos.

En C++: `RedBlackSet`, recursivo

rb_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 repair_from_grandparent(node);
14 }
```

- A la vuelta de la recursión, cada nivel se pregunta **como abuelo** si debajo hay un rojo con hijo rojo. Mismas decisiones que el pseudocódigo.

SECCIÓN 05

¿De verdad es el mismo árbol?

05

La comparación de verdad

terminal

```
$ build/collapse_check
  n ordenes    medio  segunda al.bajar
  1       1        0        0        0
  2       2        0        0        0
  3       6        0        0        0
  4      24        0       12        0
  5     120        0       60        0
  6     720        0      432        0
  7    5040        0     3216        0
  8   40320        0    28032        0
  9  362880        0   267840    31104
```

- Cuántos órdenes **no** calzan, con la forma completa: claves de cada nodo y orden de los hijos.
- **Dividir al hacer overflow, subiendo la del medio de las 3: ninguno.** Y tantas divisiones como recoloreos.

Dónde se separan

terminal

```
$ build/collapse_check
Primer orden en que no calza
  al hacer overflow, sube la del medio de las 3: ninguno
  al hacer overflow, sube siempre la segunda de las 4: 1 3 4 2
    rojinegro: [3]([1 2],[4])
    2-3-4:     [2]([1],[3 4])
  al bajar, se divide todo nodo lleno: 1 2 3 4 5 6 7 8 9
    rojinegro: [2 4 6]([1],[3],[5],[7 8 9])
    2-3-4:     [4]([2]([1],[3]),[6]([5],[7 8 9]))
```

Idea clave

Todo rojinegro **colapsa** a un 2-3-4 válido. La inserción clásica hace **exactamente** lo mismo que un 2-3-4 que se divide al hacer overflow, subiendo la del medio de las 3.

SECCIÓN 06

La invariante en las pruebas

06



check_invariants

rb_set.h

```
1  if (is_red(node)) {
2      assert(!is_red(node->left) && !is_red(node->right));
3  }
4  int left = check_node(node->left, low, &node->key, count);
5  int right = check_node(node->right, &node->key, high, count);
6  assert(left == right);
7  return left + (is_red(node) ? 0 : 1);
```

1: solo se rompe con un cast o memoria dañada. **2:** una línea aparte, en la raíz. **3:** se cumple sola, `nullptr` no es rojo. **4 y 5:** en **cada** nodo.

Error común

Contar negros solo por los extremos. En `20(10(5,15r),30(25,35))`, los extremos tienen 3, y el medio, 2.

Antes de usar el verificador, verifíquenlo

- Un `check_invariants` con errores es peor que ninguno: da confianza.
- El laboratorio trae **17 árboles rotos**, armados a mano, sin pasar por `insert`. Cada uno rompe una sola cosa.
- Raíz roja, rojos seguidos, negros desiguales en los extremos y por dentro, orden roto solo contra un antepasado lejano, padres, tamaño, un color inválido.
- Cada prueba revisa antes un árbol sano: un verificador que siempre falla tampoco pasa.

SECCIÓN 07

Eliminación

07

Eliminación: solo la idea

- No se implementa. `RedBlackMap` no ofrece `erase`.
- Sale un rojo, o un negro con hijo rojo: fácil. Su nodo 2-3-4 tenía otra clave.
- Sale un negro sin hijos: **su nodo 2-3-4 queda vacío**. Hay que mirar al hermano.

Se ve	En 2-3-4	Se hace
Hermano rojo	Nada todavía	1 rotación, cambia la forma
Hermano negro, hijos negros	**Fusión**	Recolorear. Puede subir
Hermano negro, rojo lejano	**Préstamo**	1 rotación. Termina
Hermano negro, rojo cercano	Préstamo, otra forma	1 rotación más

Por qué tantos casos

En 2-3-4 es solo fusión o préstamo. Los 8 casos salen de las dos formas de un nodo de 2 claves. Como máximo 3 rotaciones, y AVL podía necesitar $\lfloor h/2 \rfloor$ reparaciones.

SECCIÓN 08

Medido

08

La altura

terminal

```
$ build/height_table
Claves ordenadas, de menor a mayor
        n  log2n   cota   avl    rn  negros  prof.avl  prof.rn
     1000   10.0   18.9     9    16       9      7.99     8.41
  1024000   20.0   38.9    19    36      19     17.98    18.39
Claves al azar, tres semillas
        n  log2n   cota   a1  a2  a3   r1  r2  r3  prof.avl  prof.rn
     1000   10.0   18.9   11  11  11   11  11  12      8.24     8.28
  1024000   20.0   38.9   23  23  23   23  24  23     18.36    18.41
```

- **Al azar: empate.** La altura difiere en 1 como máximo, y la profundidad media en menos de 1 %.
- **En orden: casi el doble.** 19 contra 36. Es el mejor caso de AVL, y queda cerca de la cota del rojinegro.
- Pero la profundidad media difiere en **2 %**. No confundan el camino más largo con la búsqueda típica.

Rotaciones y recoloreos

terminal

```
$ build/count_table
     claves        n  rot.avl   rot.rn recol.rn  max.rot  max.rec
 crecientes  1024000    1.000    1.000    1.000        1       18
decrecientes  1024000    1.000    1.000    1.000        1       18
       azar  1024000    0.697    0.582    0.513        2       11
```

- **Al azar**, el rojinegro hace unas 17 % menos rotaciones.
- **En orden, empate**: casi una rotación por inserción en los dos.
- `max.rot` no pasó de 2. `max.rec` llegó a 18, la mitad de la altura.

Lo que se suele repetir

"Rojinegro hace menos rotaciones" es cierto al azar, y no en orden.

El reloj: un millón de claves

```
make bench-times
```

```
avl
         n ins.orden bus.orden  ins.azar  bus.azar
   1024000    0.0695    0.4802    0.9906    0.7676
rn
         n ins.orden bus.orden  ins.azar  bus.azar
   1024000    0.0560    0.4569    0.8611    0.7604
std
         n ins.orden bus.orden  ins.azar  bus.azar
   1024000    0.2345    0.6758    0.5561    0.9037
```

`-O2`, sin sanitizers, mediana de 7, cada mapa y cada n en su proceso. Laptop arm64. Dependen de la máquina.

- **Buscar: AVL y rojinegro empatan**, también en orden. La altura no se convirtió en tiempo.
- **Insertar al azar: gana `std::map`.**
- **Insertar en orden: gana nuestro rojinegro.**
- **Trampa:** la 2.ª repetición de AVL tarda 0.11 s, y la 4.ª, 0.07. Con mediana de 3 parecía que el rojinegro lo doblaba.

¿Por qué? Se cambió una sola cosa, 3 veces

¿Por qué `std::map` inserta rápido al azar?

Nuestra bajada hace **dos** comparaciones por nivel. La de `std::map`, una. Con una sola, nuestro rojinegro pasó de unos 0.86 s a **0.53 s**, y `std::map` da 0.55 s.

¿Por qué nuestro rojinegro le gana a AVL?

Nuestro `AvlMap` sube siempre hasta la raíz. Con parada temprana, en orden pasó de 0.07 a **0.038 s**, y le ganó al rojinegro, con 0.06 s. Al azar, el rojinegro sigue delante.

Lo que queda abierto, y la lección

¿Por qué `std::map` es casi 4 veces más lento en orden?

No del todo. Se queda en 0.23 s repetición tras repetición, y justo después de otro mapa, 0.09 s una sola vez. Probablemente, por el orden en que libera sus nodos. No lo medimos.

Error común

Atribuir a la estructura la diferencia de reloj entre dos programas. Aquí, la misma estructura varió casi 2 veces por una comparación. Cambien una sola cosa y vuelvan a medir.

¿Qué elegir?

Situación	Elección	Por qué, con estos números
Sin más información	`std::map`	Ya existe, y solo perdió al insertar `int` en orden y al buscar `int`
Muchas lecturas	Cualquiera	No medimos ventaja de AVL. Por clave exacta: `std::unordered_map`
Muchas escrituras al azar	`std::map`	Lo más rápido en estas mediciones
Importa el camino más largo	AVL	1.44 contra 2. En orden, 19 contra 36
Rotaciones caras	Rojinegro	Al borrar, hasta 3 contra O(log n)

En la biblioteca

El estándar no nombra ningún árbol. En libstdc++ 14, `bits/stl_tree.h` dice *Red-black tree class*, basado en Cormen, Leiserson y Rivest. `std::map` no expone su altura.

SECCIÓN 09

El laboratorio

09

Sesión 1: `RedBlackMap`

1. **Rotaciones** *(5 min)*. Copien las suyas de la semana 11, sin cambiar nada.
2. **`check_invariants`** *(35 min)*. Antes que la inserción. Debe detectar los 17 árboles rotos.
3. **`fix_insert`** *(60 min)*. A partir del pseudocódigo. Formas exactas, y al azar contra `std::map`.

Comandos

`make test-base`, `test-rotations`, `test-check`, `test-insert`, `test-lab`.
Para revisar a mano: `build/rb_trace`, que da la forma correcta y cada paso.

Sesión 2: medir, y decidir

4. **Contar** *(15 min)*. `make bench-counts`.
5. **Medir** *(45 min)*. Escriban `time_inserts` y `time_finds`. `make bench-times`, 3 veces, y `make bench-words`.
6. **Concluir** *(60 min)*. Qué elegirían para muchas lecturas, para muchas escrituras, y cuándo no vale la pena dejar `std::map`.

> **Si los números no alcanzan**
>
> Escribir que no alcanzan para decidir también es una buena conclusión.

Resumen

- **2-3-4:** nodos de 1 a 3 claves, hojas a la misma profundidad. Un nodo lleno se divide, y la del medio sube.
- **Rojinegro:** un negro con hijos rojos es un nodo 2-3-4. 3 claves: una forma. 2 claves: **dos**.
- **Altura:** $h \leq 2b - 1 \leq 2\log_2(n+1) - 1$, contando aristas.
- **Nuevo rojo.** Padre negro: nada. Tío negro: 1 o 2 rotaciones. Tío rojo: se divide con colores, y sube. Raíz roja: se pinta.
- Como máximo **2 rotaciones** por inserción. Hasta $\lfloor h/2 \rfloor$ recoloreos.
- **Calza** con un 2-3-4 que se divide al hacer overflow, subiendo la del medio de las 3. Con otras reglas, no.
- **Medido:** AVL más bajo, sobre todo en orden; rojinegro con menos rotaciones, al azar. Buscar empató, y al insertar pesó más cómo está escrito cada uno.

Para la próxima clase

- Leer los apuntes de la semana 13: heaps y colas de prioridad.
- Laboratorio: `RedBlackMap`, y el informe con sus números.

Preguntas de repaso

Dibujen un nodo 2-3-4 con tres claves como un rojinegro. Un rojo tiene un hijo rojo: ¿qué propiedad se rompe, y qué es en 2-3-4? ¿Por qué un camino tiene como máximo el doble de nodos que otro? Inserten 10, 20, 30, 15 y 25, y digan cada caso. ¿Por qué el nuevo es rojo? ¿Por qué una base de datos prefiere muchas claves por nodo?