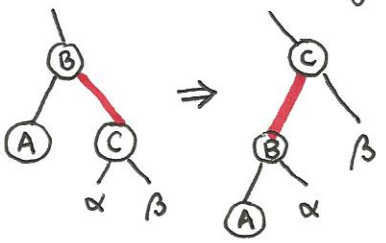
# ACLARACIONES A TODO COLOR SOBRE ARBOLES ROJINEGROS

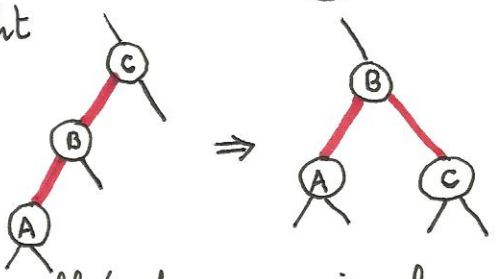
## INSERT (transparencia 42)

- La "coletilla" final tras la llamada recursiva a insert corrige

"right-leaning reds" vía rotateLeft

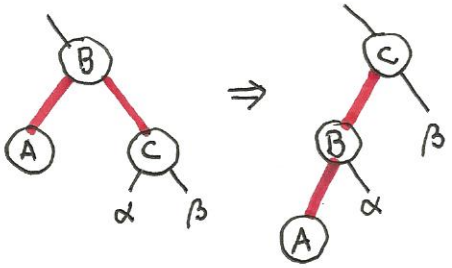



"two reds in a row" vía rotateRight




Observaciones Importantes

1) El efecto de la primera operación va más allá de corregir los desajustes /\. En realidad lo que se consigue siempre es que el hijo derecho se vuelva negro. La llamada también hay que hacerla cuando los dos hijos son **rojos**. En tal caso tenemos




de manera que se genera la secuencia indeseada de dos **rojos** por la **izqda** que se corregiría con la siguiente llamada.

2) En efecto, podemos limitarnos a corregir las secuencias de dos **rojos** por la **izqda**. No pueden detectarse problemas tales siendo uno de los punteros **rojos** por la **dcha**, pues el mismo habría sido "ennegrecido" en la etapa anterior

## INSERT (transparencias 45 y 46)

- Los 4-nodos sólo pueden generarse al insertar un elemento, pero al pasar al final de la coletilla la llamada colorFlip, los rompemos, por lo que en efecto tendríamos capturados (sólo) los árboles 2-3.

(transparencia 54)

- Esta "coletilla" final triple es el procedimiento **fixUp** que se utilizará para "apañar" el camino tras una eliminación. No sería necesario eliminar los 4-nodos si quisiéramos tener 2-3-4 árboles.

## DELETEMAX (transparencia 57)

- Cada llamada recursiva se hace con un nodo del árbol **h**. Originalmente h sería la **raíz**. Nos ocuparemos de garantizar que h "forma parte" de un 3-nodo o de un 4-nodo **antes** de hacer cada llamada recursiva vía **moveRedRight**.
- No hace falta "tomar precauciones" en la raíz pues en los demás casos lo hacemos para no generar desequilibrios al eliminar un elemento de un 2-nodo. Obsérvese que si h es la raíz y tenemos h.right == null, forzosamente h.left == null, pues en otro caso deberíamos tener originalmente **isRed(h.left)** por lo que habríamos rotado y ya no sería verdad h.right == null. Por tanto, la raíz sólo se elimina cuando era el único elemento, lo que nos llevaría al caso degenerado (pero permitido) vacío.
- Elimino la posibilidad de que h apunte al **máximo** sin que sea **hoja** vía la **rotateRight**. Además al llegar (¡después!) en tales condiciones a dicho nodo lo podremos eliminar sin problemas al estar unido a su padre con un puntero **rojo** (en otras palabras, es el elemento más a la **dcha** de un 3-nodo)
- Antes de "bajar hacia la drcha" haciendo una nueva llamada anidada a **deleteMax** con **h.right** (recordad la errata en la transparencia 57) se garantiza que no es la "raíz de un 2-nodo. Tal cosa sucede sii **h.right** y **h.right.left** son los dos negros, y se arregla con los procedimientos indicados en la transparencia 56, distinguiendo dos casos según el valor de **h.left.left**.

### Observaciones

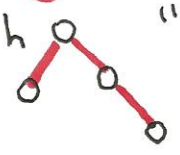
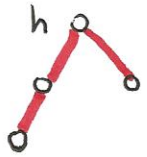
1) El "invariante" **h** o **h.right** es **rojo** se obtiene en realidad "a la mitad" de cada ejecución recursiva, tras llamar a **rotateRight** para inclinar a la **dcha** el 3-nodo del que

forma parte **h**. Como quiera que también nos ocupamos de que tal cosa suceda <u>siempre</u> (salvo en la raíz) parecería "casi" innecesaria la pregunta condicional. Pero obsérvese que el "reequilibrado" que hace **moveRedRight** en el caso en el que **h.left.left** es **rojo** ya ha girado "por adelantado" en puntero **rojo** hacia la **drcha.**

2) El "invariante" anterior no se conseguiría en el caso de que el árbol contuviera un único elemento, pero en tal caso si tratáramos de eliminarlo todo funciona correctamente, pues el puntero externo deviene correctamente **null**.

3) En el primer paso del example 1 (transparencia 58) el 2-nodo se convierte en 4-nodo debido a que los sucesores **L** y **J** son negros, por lo que "toca" aplicar el caso fácil de **moveRedRight** ya que **I** también es negro. Como indicamos antes, nada se ocupa "sistemáticamente" de lograr que la raíz deje de ser un 2-nodo, aquí sólo pasa "por culpa" de que el hijo **L** también pertenece a un 2-nodo, lo que sí ha de ser corregido antes de "bajar" a él. Por ejemplo, si **J** hubiese sido **rojo** habríamos "bajado" tranquilamente a **L** sin alterar el color negro de **D** y **L**.

<u>DELETEMIN</u> (transparencia 61)

- La asimetría de los **LLRB trees** elimina la necesidad de rotar antes de chequear **h.left** pues si **h.left == null** será también negro, por lo que en tal caso **h.right** también ha de ser negro y por tanto **null**, pues en caso contrario el árbol no estaría bien equilibrado.
- En cambio, el procedimiento **moveRedLeft** en el caso "difícil" **h.right.left rojo** exige una rotación más para conseguir la situación "simétrica" a la que teníamos tras el primer flip  en **moveRedRight**  (transparencia 56)

DELETE (transparencia 66)

- Durante la "bajada" en busca del elemento a eliminar nos aseguramos de llegar a un 3-nodo aplicando **moveRedLeft** o **moveRedRight** según corresponda, cuando veamos que debajo tenemos un 2-nodo.
- Pero cuando corresponde bajar por la **dcha**, antes de hacerlo aplicamos **rotateRight** (hay una errata, leanRight debe ser rotateRight) para que no haya problemas si nos encontramos con el elemento a eliminar en una hoja.
- Obsérvese que incluso si hemos encontrado al elemento, pero no en una hoja, "preparamos el terreno" para seguir bajando por la **dcha**, pues tendremos que hacerlo para encontrar su sucesor y eliminarlo vía **deleteMin**.

MUY IMPORTANTE: Recordemos que los procedimientos de eliminación no se ocupaban de formar 3-nodos en la raíz (¡pues de hecho ello sería imposible cuando el árbol tiene un sólo elemento!), pues "están pensados" para ser aplicados sobre la raíz de un árbol. Pero aquí llamamos a **deleteMin** con un **subárbol**: ¡si hacemos la llamada sin preocuparnos de que su raíz forme parte de un **3-nodo**, o sea sea **roja** o alguno de sus hijos lo sean, como se trate de un subárbol con un sólo nodo y lo eliminemos sin más, nos cargamos el equilibrio del árbol!
Es a través de la llamada previa a comprobar **cmp == 0** al procedimiento **moveRedRight** que se garantiza que la raíz del subárbol al que se aplicará **deleteMin** es un **3-nodo**

- Obsérvese que las respectivas llamadas finales a **fixUp** se ocupan de "apañar" el camino desde la raíz hasta el nodo eliminado (en su caso) por **deleteMin** en dos "etapas": desde la raíz al nodo interno donde estaba el elemento eliminado a través de las llamadas que hace **delete**, y desde allí hasta el nodo físico eliminado, por medio de las llamadas que hace **deleteMin**.