Capítulo 2

Árboles rojinegros

2.1. Propiedades de los árboles rojinegros

Un árbol rojinegro es un árbol binario de búsqueda, con un bit extra de almacenamiento por nodo: su color, que bien puede ser rojo (`RED`) o negro (`BLACK`). Restringiendo la forma en que los nodos pueden ser coloreados en cualquier camino desde la raíz hasta una hoja, los árboles rojinegros aseguran que no hay camino tal que sea mas del doble de largo que cualquier otro, así que, el árbol está aproximadamente balanceado.

Cada nodo del árbol ahora contiene los campos $color$, key, $left$, $right$ y p. Si un hijo del padre de un nodo no existe, el puntero correspondiente al nodo, contiene el valor `NIL`. Estos valores `NIL` serán punteros a nodos externos (hojas) del árbol binario de búsqueda y el normal, donde los nodos con clave de soporte serán nodos internos del árbol.

Un árbol binario de búsqueda es un árbol rojinegro si satisface las siguientes propiedades:

1. Cada nodo puede ser de color rojo o negro.
2. La raíz es de color negro.
3. Cada nodo `NIL`(hoja `NIL`) es de color negro.
4. Si un nodo es de color rojo, entonces sus dos nodos hijos son de color negro.
5. Para cada nodo, todos los caminos desde el nodo a las hojas descendientes contienen el mismo número de nodos de color negro.

Se le llama al número de nodos negros en cualquier camino desde, pero sin incluir, un nodo x hasta una hoja, la **altura de nodos negros** denotada por $bh(x)$. Por la propiedad 5, la noción de la altura de los nodos negros está bien definida, desde que

todos los caminos descendientes desde el nodo tengan el mismo número de nodos negros. Se define la altura de nodos negros de un árbol rojinero como la altura de nodos negros de su raíz.

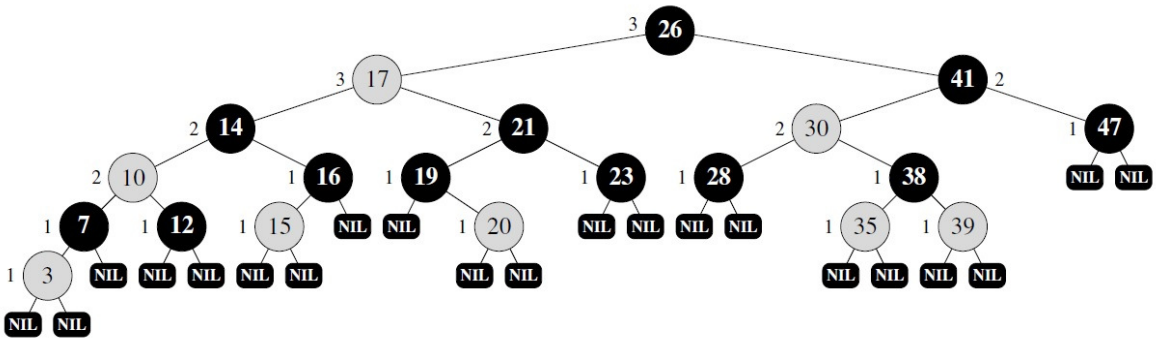

Figura 2.1: Un árbol rojinegro, con los nodos negros ocurecidos, y los nodos rojos sombreados. Cada nodo en un árbol rojinegro es rojo o bien negro, los hijos de un nodo rojo son ambos negros, y cada camino simple desde el nodo hasta una hoja descendiende contiene el mismo número de nodos negros. Cada hoja, mostrada como `NIL`, es negra. Cada nodo no nulo es marcado con su altura de nodos negros; los nodos `NIL` tienen una altura de nodos negros 0.

2.2. Rotaciones

Se cambia la estructura del puntero durante la rotación, la cual es una operación local dentro del árbol que preserva la propiedad del árbol binario de búsqueda. La Figura 2.2 muestra las dos clases de rotación: la rotación a la izquierda, y a la derecha. Cuando se hace una rotación a la izquierda de un nodo x, se asume que su hijo derecho y no es `NIL`$[T]$; x puede ser cualquier nodo en el árbol donde su hijo derecho no sea `NIL`$[T]$. La rotación a la izquierda pivotea alrededor del enlace de x a y. Esto hace que y sea la nueva raíz del subárbol, donde x es el hijo izquierdo de y y el hijo izquierdo de y es el hijo derecho de x.

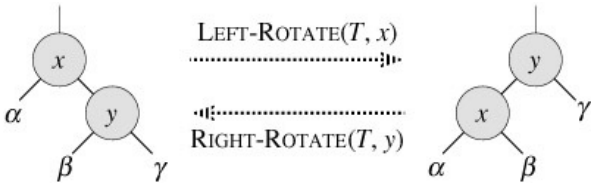

Figura 2.2: Las operaciones de rotación en un árbol binario de búsqueda. La operación `LEFT-ROTATE`(T, x) transforma la configuración de los dos nodos a la izquierda en la configuración de los de la derecha, cambiando un número constante de punteros. La configuración de la derecha puede ser transformada en la configuración de la izquierda por la operación inversa `RIGHT-ROTATE`(T, y). Las letras α, β y γ representan árboles arbitrarios. Una operación de rotación, preserva la propiedad del árbol binario de búsqueda: Las claves en α preceden la clave $key[x]$ la cual precede las claves en β, que a su vez precede a $key[\gamma]$, que precede las claves en γ.

La Figura 2.3 muestra como se trabaja la rotación a la izquierda. El código para `RIGHT-ROTATE` es simétrico. Sólo los punteros son cambiados por la rotación; todos los demás campos en un nodo permanecen iguales.

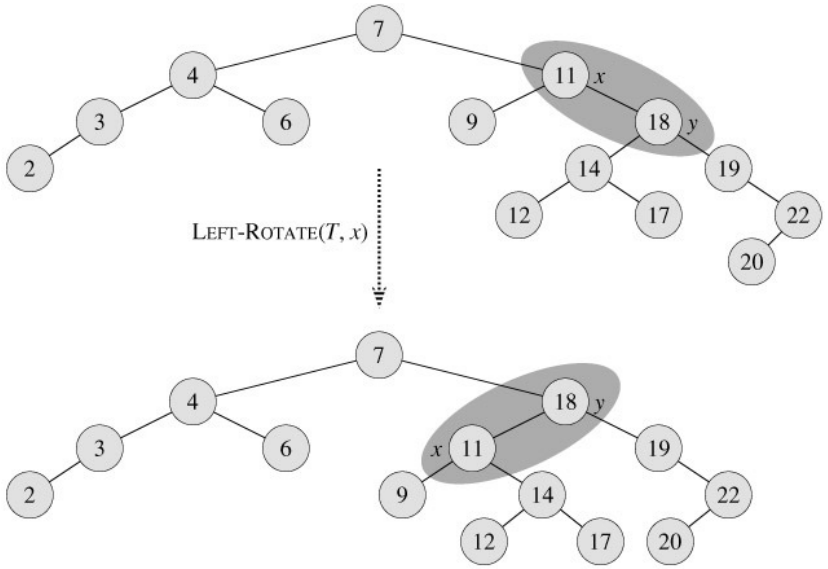

Figura 2.3: Un ejemplo de como el procedimiento `LEFT-ROTATE`(T, x) modifica un árbol binario de búsqueda. el recorrido inorden del árbol de entrada y el árbol modificado, producen la misma lista de las llaves

El pseudocódigo para `LEFT-ROTATE` asume que $right[x] \neq \texttt{NIL}[T]$ y que el padre de la raíz es el nodo $\texttt{NIL}[T]$.

Rotación a la izquierda

function `LEFT-ROTATE`(T, x)

1 $y \leftarrow right[x]$ ▷ Set y.

```
2   right[x] ← left[y]              ▷ Turn y's left subtree into x's right subtree.
3   p[left[y]] ← x
4   p[y] ← p[x]                     ▷ Link x's parent to y.
5   if p[x] == NIL[T] then
6       T ← y
7   else
8       if x == left[p[x]] then
9           left[p[x]] ← y
10      else
11          right[p[x]] ← y
12  left[y] ← x                     ▷ Put x on y's left.
13  p[x] ← y
```

El pseudocódigo para `RIGHT-ROTATE` asume que $left[x] \neq \texttt{NIL}[T]$ y que el padre de la raíz es nodo $\texttt{NIL}[T]$.

Rotación a la derecha

function `RIGHT-ROTATE`(T, x)

```
1   y ← left[x]                     ▷ Set y.
2   left[x] ← right[y]              ▷ Turn y's right subtree into x's left subtree.
3   p[right[y]] ← x
4   p[y] ← p[x]                     ▷ Link x's parent to y.
5   if p[x] == NIL[T] then
6       T ← y
7   else
8       if x == right[p[x]] then
9           right[p[x]] ← y
10      else
11          left[p[x]] ← y
```

12 $right[y] \leftarrow x$ ▷ Put x on y's right.

13 $p[x] \leftarrow y$

2.3. Inserción

Se utiliza una versión ligeramente modificada del procedimiento `TREE-INSERT` para insertar un nuevo valor k en un árbol T como si este fuese un árbol binario de búsqueda ordinario. Se crea un nuevo nodo z, en el cual se almacena el valor k y se colorea de rojo. Para garantizar que las propiedades de un árbol rojinegro se preservan, entonces se llama un procedimiento auxilar `RB-INSERT-FIXUP` para recolorear los nodos y realizar las rotaciones que sean del caso.

Algoritmo RB-Insert

function `RB-INSERT`(T, k)

1 $key[z] \leftarrow k$

2 $left[z] \leftarrow$ `NIL`$[T]$

3 $right[z] \leftarrow$ `NIL`$[T]$

4 $color[z] \leftarrow$ `RED`

5 **if** $T \neq$ `NIL`$[T]$ **then**

6 $y \leftarrow p[T]$

7 **else**

8 $y \leftarrow T$

9 $x \leftarrow T$

10 **while** $x \neq$ `NIL`$[T]$ **do**

11 $y \leftarrow x$

12 **if** $key[z] < key[x]$ **then**

13 $x \leftarrow left[x]$

14 **else**

15 $x \leftarrow right[x]$

16 $p[z] \leftarrow y$

17 **if** $y ==$ `NIL`$[T]$ **then**

```
18      T ← z
19  else
20      if key[z] < key[y] then
21          left[y] ← z
22      else
23          right[y] ← z
24  RB-INSERT-FIXUP( T, z )
```

Hay 4 diferencias entre los procedimientos `TREE-INSERT` y `RB-INSERT`. Primera, todas las instancias de `NIL` en el `TREE-INSERT` son reemplazadas por el nodo `NIL`$[T]$. Segunda, se cambia a $left[z]$ y $right[z]$ al nodo `NIL`$[T]$ en las líneas 2-3 de `RB-INSERT`, para mantener la estructura adecuada del árbol. Tercera, se colorea de rojo al nodo z en la línea 4. Cuarta, debido a que colorear z puede causar la violación a una de las propiedades del árbol rojinegro, se llama a `RB-INSERT-FIXUP`(T, z) en la línea 24 de `RB-INSERT` para restaurar estas propiedades.

Algoritmo RB-INSERT-FIXUP

```
function RB-INSERT-FIXUP( T, z )
1   while color[p[z]] == RED do
2       if p[z] == left[p[p[z]]] then
3           y ← right[p[p[z]]]
4           if color[y] == RED then
5               color[p[z]] ← BLACK                ▷ Case 1
6               color[y] ← BLACK                   ▷ Case 1
7               color[p[p[z]]] ← RED               ▷ Case 1
8               z ← p[p[z]]                        ▷ Case 1
9           else
10              if z == right[p[z]] then
11                  z ← p[z]                       ▷ Case 2
12                  LEFT-ROTATE( T, z )            ▷ Case 2
13              color[p[z]] ← BLACK                ▷ Case 3
```

```
14                  color[p[p[z]]] ← RED                    ▷ Case 3
15                  RIGHT-ROTATE( T, p[p[z]] )              ▷ Case 3
16          else
17              y ← left[p[p[z]]]
18              if color[y] == RED then
19                  color[p[z]] ← BLACK                     ▷ Case 1
20                  color[y] ← BLACK                        ▷ Case 1
21                  color[p[p[z]]] ← RED                    ▷ Case 1
22                  z ← p[p[z]]                             ▷ Case 1
23              else
24                  if z == left[p[z]] then
25                      z ← p[z]                            ▷ Case 2
26                      RIGHT-ROTATE( T, z )                ▷ Case 2
27                  color[p[z]] ← BLACK                     ▷ Case 3
28                  color[p[p[z]]] ← RED                    ▷ Case 3
29                  LEFT-ROTATE( T, p[p[z]] )               ▷ Case 3
30  color[T] ← BLACK
```

Para entender como funciona RB-INSERT-FIXUP se debe examinar el código en 3 pasos. Primero, se determinan cuales violaciones de las propiedades del árbol rojinegro fueron causadas por RB-INSERT cuando el nodo z es insertado y coloreado a rojo. Segundo, se examina el objetivo general del ciclo while en las líneas 1-29. Finalmente, se explora en cual de los 3 casos se rompe el ciclo while y se logra cumplir la meta. La Figura 2.4 muestra como RB-INSERT-FIXUP opera en un árbol rojinegro de ejemplo.

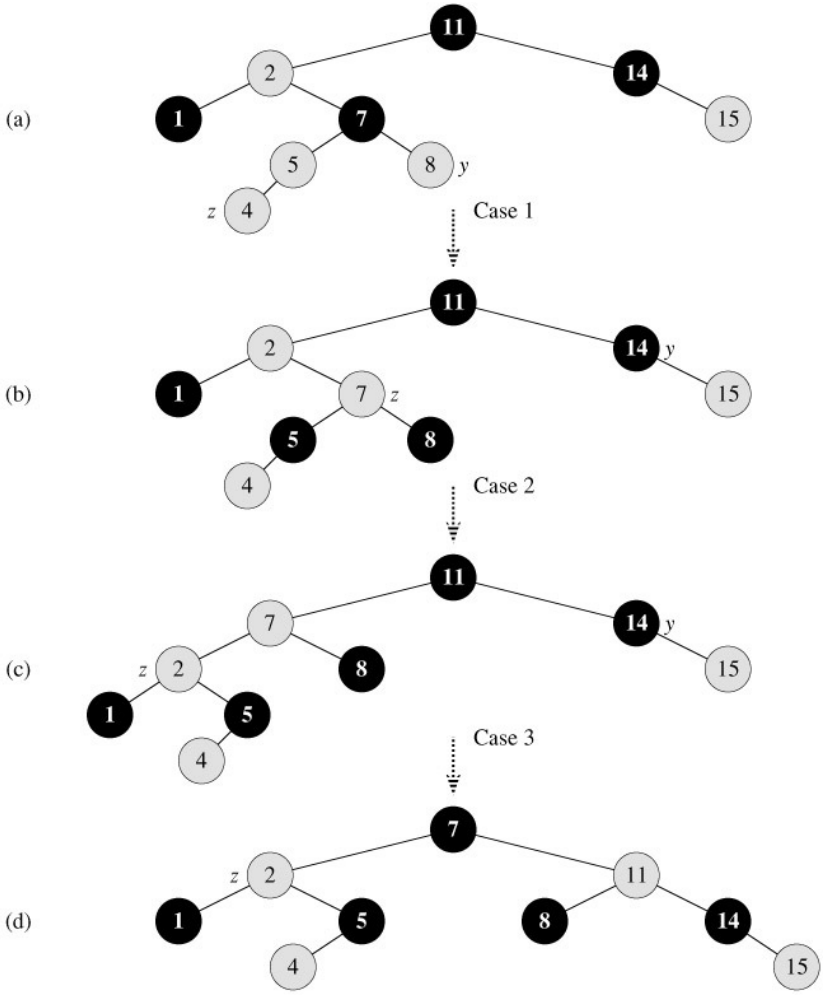

Figura 2.4: Funcionamiento de la función `RB-INSERT-FIXUP`. (a) Un nodo z después de la inserción. Como z y su padre $p[z]$ son ambos rojos, una violación de la propiedad 4 ocurre. Como el tío y de z es rojo, el caso 1 en el código puede ser aplicado. Los nodos son recoloreados y el puntero z se mueve hacia arriba del árbol, resultando en el árbol mostrado en (b). Una vez más, z y su padre son ambos rojos, pero el tío y de z es negro. Como z es el hijo derecho de $p[z]$, el caso 2 puede ser aplicado. Una rotación a la izquierda es desarrollada, y el árbol que resulta es mostrado en (c). Ahora z es el hijo izquierdo de su padre, y el caso 3 puede ser aplicado. Una rotación a la derecha produce el árbol en (d), el cual ya es un árbol rojinegro al cumplir las cinco propiedades.

2.4. Borrado

El procedimiento `RB-DELETE` es una pequeña modificación del procedimiento `TREE-DELETE`. Después de borrar un nodo, es llamado el procedimiento auxiliar `RB-DELETE-FIXUP` que ajusta los colores de los nodos y realiza tantas rotaciones como sean necesarias para restaurar las propiedades del árboles rojinegros.

Algoritmo RB-DELETE

function RB-DELETE(T, z)

```
1   if left[z] == NIL[T]  or right[z] == NIL[T] then
2       y ← z
3   else
4       y ←TREE-SUCCESSOR( z )
5   if left[y] ≠ NIL[T] then
6       x ← left[y]
7   else
8       x ← right[y]
9   p[x] ← p[y]
10  if p[y] == NIL[T] then
11      T ← x
12  else
13      if y == left[p[y]] then
14          left[p[y]] ← x
15      else
16          right[p[y]] ← x
17  if y ≠ z then
18      key[z] ← key[y]
19      Copy all information fields from y to z
20  if color[y] == BLACK then
21      RB-DELETE-FIXUP( T, x )
22  return y
```

Hay 3 diferencias entre los procedimientos TREE-DELETE y RB-DELETE. La primera, todas las referencias a NIL en TREE-DELETE son reemplazadas por las referencias al nodo NIL[T] en RB-DELETE. Segunda, la prueba de que si x es NIL en la línea 9 de TREE-DELETE es removida, y la asignación $p[x] \leftarrow p[y]$ es realizada sin condiciones en la línea 9 de RB-DELETE. Por lo tanto, si x es el nodo NIL[T], el apuntador a su padre, apunta al padre del nodo y. Tercera, un llamado a RB-DELETE-FIXUP es realizado en

las líneas 20-21 si y es negro. Si y es rojo, las propiedades se mantienen cuando y es borrado, debido a las siguientes razones:

- Ninguna altura negra en el árbol ha cambiado.
- No hay nodos rojos adyacentes, y
- Desde que y no sea la raíz, si este es rojo, la raíz permanecerá negra.

El nodo x utilizado en el llamado a `RB-DELETE-FIXUP` es uno de dos nodos: o el nodo que era hijo único de y antes de que y fuese borrado si y tuviese un hijo que no fuese el nodo `NIL`$[T]$, o si y no tiene hijos, x es el nodo `NIL`$[T]$. En éste último caso, la asignación incondicional en la línea 9 garantiza que el padre de x es ahora el nodo que antes fue el padre de y.

Algoritmo RB-DELETE-FIXUP

```
function RB-DELETE-FIXUP( T, x )
1   while x ≠ T and color[x] == BLACK do
2       if x == left[p[x]] then
3           w ← right[p[x]]
4           if color[w] == RED then
5               color[w] ← BLACK                          ▷ Case 1
6               color[p[x]] ← RED                         ▷ Case 1
7               LEFT-ROTATE( T, p[x] )                    ▷ Case 1
8               w ← right[p[x]]                           ▷ Case 1
9           if color[left[w]] == BLACK and color[right[w]] == BLACK then
10              color[w] ← RED                            ▷ Case 2
11              x ← p[x]                                  ▷ Case 2
12          else
13              if color[right[w]] == BLACK then
14                  color[left[w]] ← BLACK                ▷ Case 3
15                  color[w] ←RED                         ▷ Case 3
16                  RIGHT-ROTATE( T, w )                  ▷ Case 3
17                  w ← right[p[x]]                       ▷ Case 3
```

```
18                  color[w] ← color[p[x]]                    ▷ Case 4
19                  color[p[x]] ← BLACK                       ▷ Case 4
20                  color[right[w]] ← BLACK                   ▷ Case 4
21                  LEFT-ROTATE( T, p[x] )                    ▷ Case 4
22                  x ← T                                     ▷ Case 4
23          else
24              w ← left[p[x]]
25              if color[w] == RED  then
26                  color[w] ← BLACK                          ▷ Case 1
27                  color[p[x]] ← RED                         ▷ Case 1
28                  RIGHT-ROTATE( T, p[x] )                   ▷ Case 1
29                  w ← left[p[x]]                            ▷ Case 1
30              if color[right[w]] == BLACK  and color[left[w]] == BLACK then
31                  color[w] ← RED                            ▷ Case 2
32                  x ← p[x]                                  ▷ Case 2
33              else
34                  if color[left[w]] == BLACK then
35                      color[right[w]] ← BLACK               ▷ Case 3
36                      color[w] ←RED                         ▷ Case 3
37                      LEFT-ROTATE( T, w )                   ▷ Case 3
38                      w ← left[p[x]]                        ▷ Case 3
39                  color[w] ← color[p[x]]                    ▷ Case 4
40                  color[p[x]] ← BLACK                       ▷ Case 4
41                  color[left[w]] ← BLACK                    ▷ Case 4
42                  RIGHT-ROTATE( T, p[x] )                   ▷ Case 4
43                  x ← T                                     ▷ Case 4
44  color[x] ← BLACK
```

Si el nodo a borrar y del árbol en `RB-DELETE` es negro, 3 problemas se pueden generar. Primero, si y es la raíz y un hijo rojo de y se vuelve la nueva raíz, entonces se está violado la propiedad 2. Segundo, si x y $p[y]$ (el cual ahora es $p[x]$) son rojos, entonces se está violado la propiedad 4. Tercero, remover a y causa que cualquier camino que previamente haya contenido a y tenga un nodo menos de color negro. Por lo tanto, la propiedad 5 se ha violado por algún ancestro de y en el árbol.