CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Árboles rojo negro

© 2016 Blai Bonet

Objetivos

- Árboles rojo negro
- Operaciones de inserción y eliminación

© 2016 Blai Bonet

Árboles rojo negro

Un árbol rojo negro es un tipo especial de árbol binario de búsqueda el cual está garantizado que su altura sea $O(\log n)$ cuando almacena n elementos

Adicional a los apuntadores para los hijos y el padre, los nodos en un árbol rojo negro tienen un **1 bit de color**

En lugar de asignar valor null a los apuntadores, utilizamos un **único nodo sentinela** denotado por nil: el papá del nodo raíz apunta al sentinela, así como los apuntadores hijos de las hojas, y el apuntador correspondiente de los nodos internos con un solo hijo

Todos los nodos tienen padre bien definido excepto el sentinela

© 2016 Blai Bonet

Ejemplo de árbol rojo negro

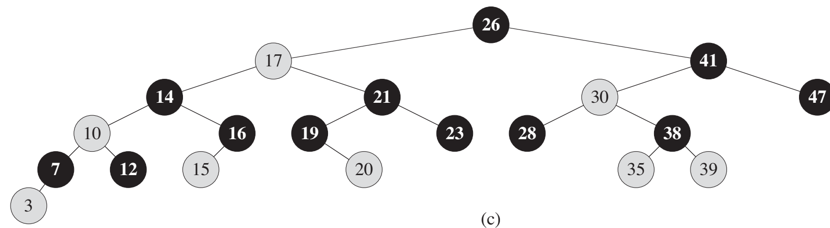

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Ejemplo de árbol rojo negro con sentinela

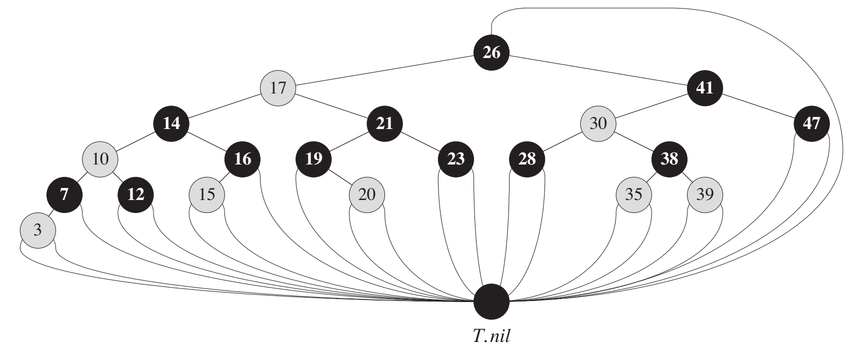

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Claves duplicadas

En un árbol binario de búsqueda, las claves duplicadas no representan ningún problema y por lo tanto son permitidas

En un árbol rojo negro, las claves duplicadas representan un problema al momento de insertar/eliminar nodos. Por lo tanto, en un árbol rojo negro no permitimos la existencia de claves duplicadas

© 2016 Blai Bonet

Propiedades de árboles rojo negro

Los colores son utilizados para **acotar** las estructuras permitidas. En particular, en un árbol rojo negro el camino más largo puede ser a lo sumo 2 veces más largo que el camino más corto

Las propiedades que definen a los árboles rojo y negro son:

P1. Todo nodo es rojo o negro

P2. La raíz es negra

P3. El sentinela nil es negro

P4. Si un nodo es rojo, sus hijos son negros

P5. Para cada nodo, todos los caminos simples desde el nodo hasta las hojas tienen el **mismo número de nodos negros**

© 2016 Blai Bonet

Altura negra

Definimos la **altura negra** de un nodo x, denotada por $bh(x)$, como el número de nodos negros, sin incluir x, en un camino simple desde x hasta las hojas

Por la propiedad P5, la altura negra está bien definida

© 2016 Blai Bonet

Ejemplo de árbol rojo negro con alturas negras

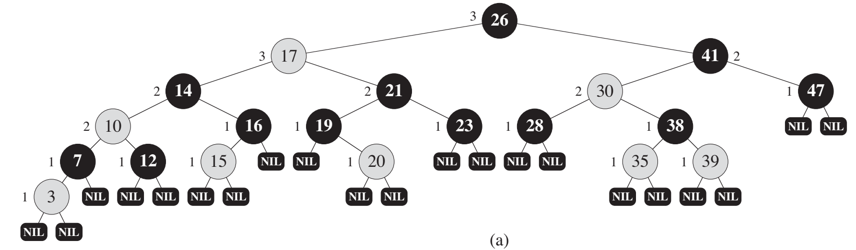

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Altura negra: resultado fundamental

Un árbol rojo negro con n **claves (nodos internos)** tiene altura h menor o igual a $2\log_2(n+1)$

Prueba:

Primero mostramos por inducción en $h(x)$ que el subárbol enraizado en x tiene un **número de claves** $N(x) \geq 2^{bh(x)} - 1$

• Para $h(x) = 0$, $x = \text{nil}$, $bh(x) = 0$ y $N(x) = 0 = 2^0 - 1 = 2^{bh(x)} - 1$

• Para el paso inductivo, considere un nodo x con $h(x) > 0$. Sus dos hijos y y z tienen altura negra igual a $bh(x)$ ó $bh(x) - 1$

Usando la hipótesis inductiva

$$\begin{aligned} N(x) &= 1 + N(y) + N(z) \ \geq\ 1 + (2^{bh(y)} - 1) + (2^{bh(z)} - 1) \\ &\geq\ 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \ =\ 2^{bh(x)} - 1 \end{aligned} \qquad \square$$

© 2016 Blai Bonet

Altura negra: resultado fundamental

Un árbol rojo negro con n **claves (nodos internos)** tiene altura h menor o igual a $2\log_2(n+1)$

Prueba:

Considere ahora un árbol rojo negro con raíz r y de altura h con n claves

Por la propiedad P4, **a lo sumo la mitad de los nodos** en un camino simple desde la raíz hasta las hojas son **rojos**

Por lo tanto, la altura negra de la raíz satisface $bh(r) \geq h/2$

Entonces, $n \ \geq\ 2^{bh(r)} - 1 \ \geq\ 2^{h/2} - 1 \implies n + 1 \ \geq\ 2^{h/2}$

Calculamos:

$$\log_2(n+1) \ \geq\ h/2 \quad \implies \quad h \ \leq\ 2\log_2(n+1) \qquad \square$$

© 2016 Blai Bonet

Operaciones tipo query sobre árboles rojo negro

Como un árbol rojo negro con n claves es un árbol binario de búsqueda de altura $O(\log n)$, entonces las operaciones de query:

- `Search`
- `Minimum`
- `Maximum`
- `Predecessor`
- `Successor`

corren en tiempo $O(\log n)$ sobre árboles rojo negro

Las operaciones de inserción y eliminación deben modificarse para que preserven las propiedades P1–P5 que definen a los árboles rojo negro

© 2016 Blai Bonet

Rotaciones

Operación fundamental para **balancear el árbol** durante inserciones y eliminaciones. Existen rotaciones a la izquierda y a la derecha:

- En una rotación a la izquierda con pivote x se asume $x.right \neq$ nil
- En una rotación a la derecha con pivote x se asume $x.left \neq$ nil

También, como se dijo antes, se asume que el padre de la raíz es nil

© 2016 Blai Bonet

Rotaciones

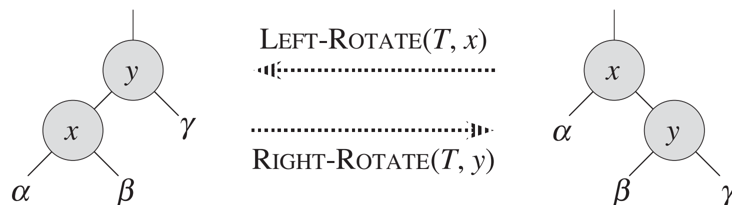

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Ambas operaciones toman **tiempo constante**

Ambas operaciones mantienen la propiedad de **árbol de búsqueda**

© 2016 Blai Bonet

Rotación a la izquierda

```
1  Left-Rotate(RBtree T, pointer x)
2      y = x.right
3      x.right = y.left
4      if y.left != T.nil then y.left.p = x
5
6      y.p = x.p
7      if x.p == T.nil
8          T.root = y
9      else if x == x.p.left
10         x.p.left = y
11     else
12         x.p.right = y
13     y.left = x
14     x.p = y
```

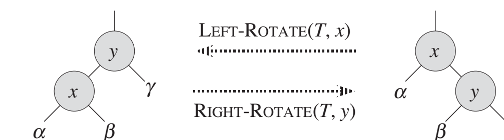

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Ejemplo de rotación a la izquierda

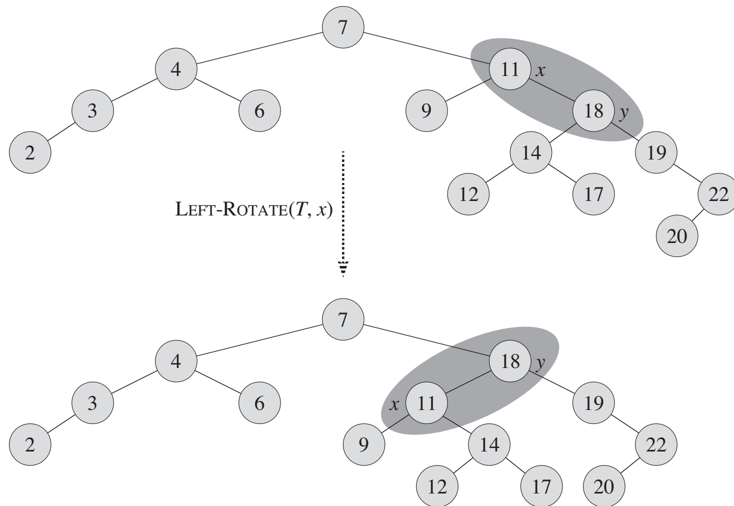

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Inserción

Las inserciones se hacen de manera similar a las inserciones en un árbol binario de búsqueda:

– primero se busca en donde realizar la inserción

– segundo, el nodo z se inserta a nivel de las hojas y se **pinta de rojo**

– luego, de ser necesario, se pintan y se hacen rotaciones sobre los ancestros de z para restablecer las propiedades P1–P5

© 2016 Blai Bonet

Inserción

```
1  RB-Insert(RBtree T, pointer z)
2      y = T.nil                      % inicializamos x a la raíz y y al padre de x
3      x = T.root
4      while x != T.nil                     % busacamos en donde hacer la inserción
5          y = x
6          if z.key < x.key
7              x = x.left
8          else
9              x = x.right
10
11     % insertamos el nodo z como hijo de y (a nivel de las hojas)
12     z.p = y
13     if y.p == T.nil
14         T.root = z
15     else if z.key < y.key
16         y.left = z
17     else
18         y.right = z
19     z.left = T.nil
20     z.right = T.nil
21     z.color = RED                                   % el nodo z se pinta de rojo
22
23     RB-Insert-Fixup(T, z)                     % restablecemos propiedades P1--P5
```

© 2016 Blai Bonet

Corrección en inserción

```
1  RB-Insert-Fixup(RBtree T, pointer z)
2      while z.p.color == RED
3          if z.p == z.p.p.left
4              y = z.p.p.right
5              if y.color == RED                                       % caso 1
6                  z.p.color = BLACK                                   % caso 1
7                  y.color = BLACK                                     % caso 1
8                  z.p.p.color = RED                                   % caso 1
9                  z = z.p.p                                           % caso 1
10             else
11                 if z == z.p.right                                   % caso 2
12                     z = z.p                                         % caso 2
13                     Left-Rotate(T, z)                               % caso 2
14
15                 z.p.color = BLACK                                   % caso 3
16                 z.p.p.color = RED                                   % caso 3
17                 Right-Rotate(T, z.p.p)                              % caso 3
18         else
19             % como el bloque de arriba pero intercambiando left
20             % por right, y Left-Rotate por Right-Rotate
21
22     T.root.color = BLACK
```

© 2016 Blai Bonet

Ejemplo de inserción

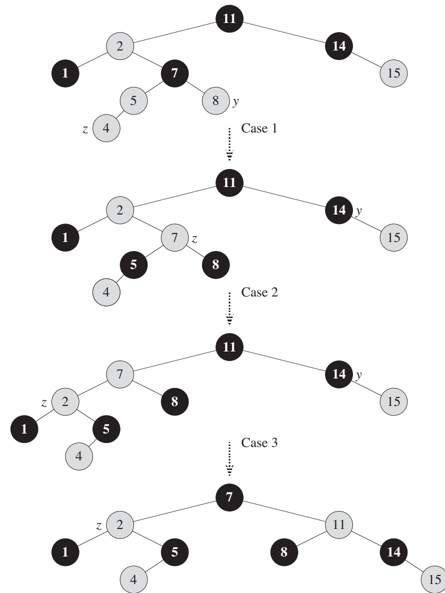

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Análisis de la inserción

Como en árboles binarios, la búsqueda del lugar para la inserción toma tiempo $O(h) = O(\log n)$ donde h es la altura del árbol y n es el número de claves en el árbol

Luego, la inserción llama a `RB-Insert-Fixup(T, z)`:

– en el caso 3, el padre de z se pinta de rojo y el lazo termina

– en el caso 2, a continuación se pasa al caso 3

– en el caso 1, z se asigna al "abuelo" de z

Entonces, `RB-Insert-Fixup(T, z)` toma tiempo $O(h) = O(\log n)$ en el peor caso, y la inserción toma tiempo $O(\log n)$

© 2016 Blai Bonet

Correctitud de la inserción

¿Qué propiedades pueden ser violadas después de insertar z y antes de llamar a `RB-Insert-Fixup`?

P1. Todo nodo es rojo o negro **[CIERTA]**

P2. La raíz es negra

P3. El sentinela nil es negro **[CIERTA]**

P4. Si un nodo es rojo, sus hijos son negros

P5. Para cada nodo, todos los caminos simples desde el nodo hasta las hojas tienen el **mismo número de nodos negros** **[CIERTA]**

Solamente P2 ó P4 pudieran ser violadas

Ambas violaciones vendrían de pintar de rojo al nuevo nodo z

© 2016 Blai Bonet

Restablecimiento de las propiedades

`RB-Insert-Fixup` restablece las propiedades de forma iterativa

La correctitud se establece con el siguiente **invariante de tres partes** que es cierto al inicio de cada iteración del ciclo:

– el nodo z es rojo

– si $z.p$ es la raíz del árbol, entonces $z.p$ es negro

– si el árbol viola alguna de las propiedades, entonces se viola una sola de ellas que es P2 o P4. Si el árbol viola P2 es porque z es la raíz y z es rojo. Si el árbol viola P4 es porque ambos z y $z.p$ son rojos

Cuando el ciclo termina, $z.p$ es negro y por lo tanto, por el invariante, la unica propiedad que pudiera estar violándose sería P2. La línea 22 restablece P2 al pintar la raíz de negro

© 2016 Blai Bonet

Caso 1 durante inserción

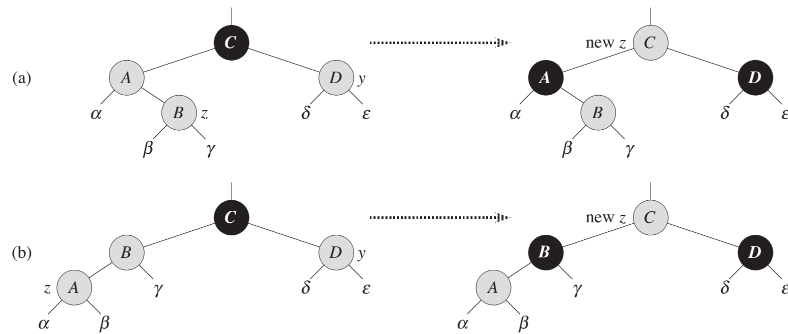

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Casos 2 y 3 durante inserción

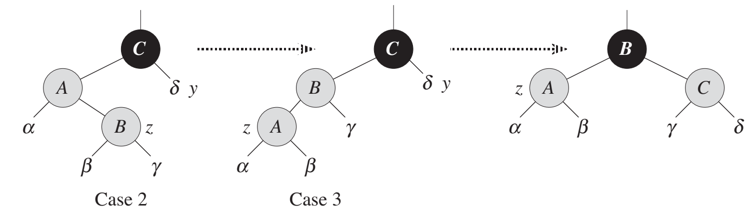

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Eliminación

Como en la inserción, la eliminación se basa en la eliminación de nodos para árboles binarios de búsqueda

La eliminación recibe un apuntador al nodo z que se quiere eliminar

Como lo hicimos en árboles binarios de búsqueda, usamos una rutina de **transplante de nodos**

© 2016 Blai Bonet

Transplante de nodos en árboles rojo negro

```
1 RB-Transplant(RBtree T, pointer u, pointer v)
2     if u.p == T.nil                          % es u es raíz?
3         T.root = v
4     else if u = u.p.left                     % es u hijo izquierdo?
5         u.p.left = v
6     else                                     % u es hijo derecho
7         u.p.right = v
8     v.p = u.p
```

Un poco más sencilla que el transplante en árboles binarios de búsqueda ya que el nodo v siempre es distinto de null

© 2016 Blai Bonet

Eliminación en árboles rojo negro

```
1   RB-Delete(RBtree T, pointer z)
2       y = z
3       y-original-color = y.color
4       if z.left == T.nil
5           x = z.right
6           RB-Transplant(T, z, z.right)
7       else if z.right == T.nil
8           x = z.left
9           RB-Transplant(T, z, z.left)
10      else
11          y = Tree-Minimum(z.right)
12          y-roginal-color = y.color
13          x = y.right
14
15          if y.p == z
16              x.p = y
17          else
18              RB-Transplant(T, y, y.right)
19              y.right = z.right
20              y.right.p = y
21
22          RB-Transplant(T, z, y)
23          y.left = z.left
24          y.left.p = y
25          y.color = z.color
26
27      if y-original-color == BLACK
28          RB-Delete-Fixup(T, x)
```

© 2016 Blai Bonet

Corrección en eliminación

```
1   RB-Delete-Fixup(RBtree T, pointer x)
2       while x != T.root && x.color == BLACK
3           if x == x.p.left
4               w = x.p.right
5               if w.color == RED                                    % caso 1
6                   w.color = BLACK                                  % caso 1
7                   x.p.color = RED                                  % caso 1
8                   Left-Rotate(T, x.p)                              % case 1
9                   w = x.p.right                                    % caso 1
10
11              if w.left.color == BLACK && w.right.color == BLACK
12                  w.color = RED                                    % caso 2
13                  x = x.p                                          % caso 2
14              else
15                  if w.right.color == BLACK
16                      w.left.color = BLACK                         % caso 3
17                      w.color = RED                                % caso 3
18                      Right-Rotate(T, w)                           % caso 3
19                      w = x.p.right
20
21                  w.color = x.p.color                              % caso 4
22                  x.p.color = BLACK                                % caso 4
23                  w.right.color = BLACK                            % caso 4
24                  Left-Rotate(T, x.p)                              % caso 4
25                  x = T.root
26          else
27              % como el bloque de arriba pero intercambiando left
28              % por right, y Left-Rotate por Right-Rotate
29
30      x.color = BLACK
```

© 2016 Blai Bonet

Análisis de la eliminación

RB-Delete(T, z) toma tiempo $O(h) = O(\log n)$ sin contar el tiempo de RB-Delete-Fixup(T, x) por la llamada a Tree-Minimum

- En el caso 1 se realizan operaciones de tiempo constante y luego se pasa al caso 2, 3 o 4
- Los casos 3 y 4 terminan por la asignación de la raíz a x en línea 25
- El caso 2 termina con la asignación $x = x.p$ por lo que el número de iteraciones está acotado por $O(h) = O(\log n)$

Entonces, RB-Delete(T, z) toma tiempo $O(\log n)$

© 2016 Blai Bonet

Correctitud de la eliminación

Ver el libro!

© 2016 Blai Bonet

Resumen

- Los árboles rojo negro implementan todas las operaciones sobre conjuntos dinámicos en tiempo $O(\log n)$ en el peor caso donde n es el número de elementos almacenados en el árbol
- Los árboles rojo negro asignan colores a los nodos (rojo o negro) para mantener los árboles aproximadamente balanceados
- Las operaciones de inserción y eliminación de nodos autobalancean el árbol utilizando los colores y las rotaciones
- Las operaciones de query son las mismas que para los árboles binarios de búsqueda ya que un árbol rojo negro es un tipo especial de árbol binario de búsqueda

© 2016 Blai Bonet

Ejercicios (1 de 5)

1. (13.1-1) Dibuje un árbol binario de búsqueda completo de altura 3 para las claves $\{1, 2, \ldots, 15\}$. Agregue el sentinela nil de color negro y luego pinte los nodos de rojo y negro de forma que resulten diferentes árboles rojo negro de alturas negra 2, 3 y 4
2. (13.1-3) Considere un árbol rojo negro que satisface todas las propiedades excepto P2 (i.e. la raíz es roja en vez de negra). Si cambiamos el color de la raíz de rojo a negro, ¿es el árbol resultante un árbol rojo negro?
3. (13.1-5) Muestre que el camino simple más largo desde la raíz a una hoja en un árbol rojo negro tiene longitud que es a lo sumo el doble a la longitud del camino simple más corto de la raíz a una hoja
4. (13.1-6) ¿Cuál es el número mayor de nodos internos que un árbol rojo negro de altura negra k puede tener? ¿Cuál es el menor número posible?

© 2016 Blai Bonet

Ejercicios (2 de 5)

5. (13.1-7) Describa un árbol rojo negro que tenga la mayor proporcion de nodos rojos internos con respecto a nodos negros internos. ¿Cuál es dicha proporción? ¿Cuál es la menor tal proporción y cuál es su valor?
6. (13.2-1) Escriba la rutina `Right-Rotate`
7. (13.2-2) Argumente que en un **árbol binario de búsqueda** con n nodos existen exactamente $n - 1$ rotaciones posibles
8. Muestre que las rotaciones preservan la propiedad de árbol binario de búsqueda

© 2016 Blai Bonet

Ejercicios (3 de 5)

9. (13.2-3) Sean a, b y c nodos arbitrarios en subárboles α, β y γ respectivamente en siguiente figura. ¿Cómo cambian las profundidades de a, b y c cuando una rotación a la izquierda es aplicada al nodo x?

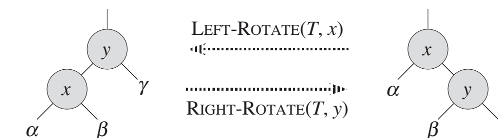

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

10. (13.2-4) Muestre que un **árbol binario de búsqueda** cualquiera con n nodos puede ser transformado a otro árbol binario de búsqueda cualquiera mediante la aplicación de $O(n)$ rotaciones (Ayuda: primero muestra que bastan $n - 1$ rotaciones para transformar cualquier árbol binario de búsqueda en un árbol binario que consiste de una única cadena más a la derecha)

© 2016 Blai Bonet

Ejercicios (4 de 5)

11. (13.3-1) En la línea 21 de RB-Insert, se pinta de rojo el nuevo nodo insertado. Si lo hubieramos pintado de negro, entonces las propiedades P2 y P4 no serían violadas. ¿Por qué se escoge pintar de rojo el nuevo nodo y no de negro?

12. (13.3-2) Muestre el árbol rojo negro que resulta de hacer inserciones sucesivas de las claves 41, 38, 31, 12, 19 y 8 sobre un árbol rojo negro inicialmente vacío

13. (13.3-3) Suponga que la altura negra de cada uno de los subárboles α, β, γ, δ y ε en las figuras para los casos 1, 2 y 3 de la inserción es k. Verifique que la transformación indicada preserva la propiedad P5

14. (13.3-4) Muestre que RB-Insert-Fixup no cambia el color del nodo nil

© 2016 Blai Bonet

Ejercicios (5 de 5)

15. (13.3-5) Considere un árbol rojo negro de n nodos construido a partir de una secuencia de n llamadas a RB-Insert. Argumente que si $n > 1$, el árbol tiene al menos un nodo rojo

16. (13.4-3) En el ejercicio 12 se le pide calcular el árbol rojo negro que resulta de una secuencia de inserciones. Partiendo de ese árbol, muestre los árboles que resultan de eliminar de forma sucesiva las claves 8, 12, 19, 31, 38 y 41

17. (13.4-7) Suponga que un nodo x es insertado en un árbol rojo y negro con RB-Insert y luego eliminado immediatamente con RB-Delete. ¿És el árbol resultante el mismo antes de la inserción? Justifique

© 2016 Blai Bonet