

Figura 1: Rotación de árboles binarios de búsqueda

## 1. Árboles Rojo-Negros

Un árbol rojo-negro resulta de la representación de un árbol 2-3-4 mediante un árbol binario.

Cada nodo de un árbol rojo negro contiene la siguiente información: color, clave, hijo izquierdo, hijo derecho y padre. Si un hijo o el padre de un nodo no existe, el apuntador correspondiente contiene el valor NULL, el cual consideraremos como un nodo cuyo color es negro. En lo sucesivo nos referiremos a los nodos distintos a las hojas (NULL) como nodos internos del árbol y a las hojas y al padre de la raíz como nodos externos.

Todo árbol rojo-negro satisface las siguientes propiedades:

1. Todo nodo es rojo o negro
2. La raíz es negra
3. Toda hoja (NULL) es negra.
4. Si un nodo es rojo, entonces sus dos hijos son negros.
5. Para cada nodo, todas las rutas de un nodo a las hojas (NULLs) contienen el mismo número de nodos negros. El número de nodos negros en esta ruta se conoce como *altura-negra* del nodo.

Una posible representación de un árbol rojo negro y sus nodos en Java sería:

### 1.1. Rotaciones

Las operaciones de inserción y eliminación de un árbol binario de búsqueda, si se aplican a un árbol rojo negro pueden modificar las propiedades enumeradas en la sección anterior. Para restaurar estas propiedades es necesario cambiar el color de algunos nodos así como también la estructura de los apuntadores.

La estructura de los apuntadores se cambia mediante *rotación*, la cual es una operación que preserva las propiedades de un árbol binario de búsqueda. Existen dos tipos de rotaciones: a la izquierda y a la derecha.

La Figura 1 muestra las operaciones de rotación sobre árboles binarios de búsqueda.

A continuación se muestra el código Java de las rutinas de rotación:

La Figura 2 muestra cómo se modifica un árbol binario por efecto de una rotación a la izquierda en uno de sus nodos.

---

**Algorithm 1** Estructura de un árbol rojo-negro

---

```
public class ArbolRojoNegro
```

```
{
    static class Nodo
    {
        Comparable clave;
        Nodo izq;
        Nodo der;
        Nodo padre;
        boolean color;
    }
    private static final boolean NEGRO = false;
    private static final boolean ROJO = true;
    private static final Nodo NULL;
    static
    {
        NULL = new Node();
        NULL.clave = null;
        NULL.padre = NULL;
        NULL.izq = NULL;
        NULL.der = NULL;
        NULL.color = NEGRO;
    }
    void insertar(Nodo z) {...}
    void eliminar(Nodo z) {...}
    Nodo buscar(Comparable clave)
}
```

---

Fig.

Figura 2: Efecto de la rotación a la izquierda en un nodo de un árbol binario de búsqueda

---

**Algorithm 2** Rotaciones a la izquierda y a la derecha

---

```
public rotarIzq(Nodo x)
{
    Nodo y = x.der;
    x.der = y.izq;
    y.izq.padre = x;
    y.padre = x.padre;
    if (x.padre==NULL)
        root = y;
    else if (x==x.padre.izq)
        x.padre.izq = y;
    else
        x.padre.der = y;
    y.izq = x;
    x.padre = y;
}
public rotarDer(Nodo y)
{
    x = y.izq;
    y.izq = x.der;
    x.der.padre = y;
    x.padre = y.padre;
    if (y.padre==NULL)
        root = x;
    else if (y==y.padre.izq)
        y.padre.izq = x;
    else
        y.padre.der = x;
    x.der = y;
    y.padre = x;
}
```

---

---

**Algorithm 3** Inserción de un nodo en un árbol rojo-negro

---

```
public void insertar(Nodo z) {
    Nodo y = NULL;
    Nodo x = root;
    while (x!=NULL) {
        y = x;
        if (z.clave < x.clave)
            x = x.izq;
        else
            x = x.der;
    }
    z.padre = y;
    if (y == NULL)
        root = z;
    else if (z.clave < y.clave)
        y.izq = z;
    else
        y.der = z;
    z.izq = z.der = NULL;
    z.color = ROJO;
    corregirInsercion(z)
}
```

---

## 1.2. Inserción

La inserción de un nodo en un árbol rojo-negro con  $n$  elementos puede realizarse en un tiempo  $O(\lg n)$ . Utilizamos una versión modificada de la rutina de inserción en un árbol binario de búsqueda ordinario:

Puede observarse que el código es casi igual a la rutina de inserción en un árbol binario de búsqueda normal, salvo por tres diferencias fundamentales:

1. En lugar de usar el `null` ordinario, utilizamos la constante `NULL` ya que necesitamos que las hojas tengan color NEGRO.
2. Se establece el color del nodo  $z$  a insertar como ROJO.
3. Se invoca a la rutina `corregirInserción(z)` para restaurar las propiedades de árbol rojo-negro.

Nótese que una vez insertado y marcado como rojo el nodo  $z$  (justo antes de llamar a `corregirInsercion`) se presenta la siguiente situación:

- Las propiedades 1 y 3 indudablemente se cumplen, ya que los hijos del nuevo nodo rojo insertado son `NULL` y por lo tanto son negros.

- La propiedad 5 se cumple debido a que no se ha insertado ningún nodo negro y por lo tanto todas las rutas de la raíz a las hojas siguen teniendo el mismo número de hijos negros.
- Las únicas propiedades que podrían ser violadas son la 2, que requiere que la raíz sea negra y la 4, que dice que un nodo rojo no puede tener ningún hijo rojo. Ambas violaciones se deben a que el nuevo nodo es rojo. La propiedad 2 se viola si  $z$  es la raíz y la 4 si el padre de  $z$  es rojo.

Para estudiar cómo `corregirInsercion` permite restaurar las propiedades de árbol rojo-negro examinaremos el código en tres partes: inicialización, terminación y mantenimiento.

### 1.2.1. Inicialización

Antes de la primera iteración, comenzamos con un árbol rojo-negro sin violaciones y añadimos un nodo rojo  $z$ .

Si hay una violación a la propiedad 2 (raíz negra), entonces la raíz roja tiene que ser el nodo recién añadido  $z$ , el cual sería el único nodo interno del árbol. Debido a que tanto el padre como los hijos de  $z$  son NULL, el cual es negro, no hay violación de la propiedad 4. Así, esta sería la única violación en el árbol.

Si hay una violación de la propiedad 4, entonces, considerando que los hijos del nodo  $z$  son NULL negros y que el árbol no tenía violaciones antes de la inserción de  $z$ , la violación tiene que ser porque tanto  $z$  como  $z.padre$  son rojos. Es imposible que haya alguna otra violación de las propiedades.

### 1.2.2. Terminación

Cuando el ciclo termina, lo hace porque  $z.padre$  es negro. Así, no hay violación de la propiedad 4 al terminar el ciclo. La única propiedad que podría fallar es la propiedad 2, la cual es restaurada en la línea 27.

### 1.2.3. Mantenimiento

Hay seis casos a considerar dentro del ciclo while, pero tres de ellos son simétricos; dependiendo de si  $z.padre$  es un hijo izquierdo o un hijo derecho del abuelo de  $z$  ( $z.padre.padre$ ), lo cual se determina en la línea 2. Estudiaremos únicamente la primera posibilidad, correspondiente a las líneas 3-14 de `corregirInsercion` (nótese que el código entre las líneas 15 y 26 es simétrico).

Nótese que  $z.padre.padre$  existe, ya que la condición del ciclo es que  $z.padre$  sea rojo y por lo tanto  $z.padre$  no puede ser la raíz.

El primero de los tres casos a considerar se diferencia de los casos 2 y 3 por el color del tío de  $z$  ( $z.padre.padre.der$ ). Si el tío ( $y$ ) es rojo entonces se ejecuta el caso 1. De otro modo se transfiere el control a los casos 2 y 3. En los tres casos el abuelo de  $z$  ( $z.padre.padre$ ) es negro, puesto que el padre  $z.padre$  es rojo y la propiedad 4 sólo puede ser violada entre  $z$  y  $z.padre$ .

---

**Algorithm 4** Corrección de las propiedades de un árbol rojo negro inmediatamente después de la inserción de un nodo.

---

```
void corregirInsercion(Nodo z){
1   while (z.padre.color == RED){
2       if (z.padre == z.padre.padre.izq) {
3           y = z.padre.padre.der;
4           if (y.color==RED) {
5               z.padre.color = NEGRO;
6               y.color = NEGRO;
7               z.padre.padre.color = ROJO;
8               z = z.padre.padre;
9           } else {
10              if (z == z.padre.der) {
11                  z = z.padre;
12                  rotarIzq(z);
13              }
14              z.padre.color = NEGRO;
15              z.padre.padre.color = ROJO;
16              rotarDer(z.padre.padre);
17          }
18      } else {
19          y = z.padre.padre.izq;
20          if (y.color == ROJO) {
21              z.padre.color = NEGRO;
22              y.color = NEGRO;
23              z.padre.padre.color = ROJO;
24              z = z.padre.padre;
25          } else {
26              if (z == z.padre.izq) {
27                  z = z.padre;
28                  rotarDer(z);
29              }
30              z.padre.color = NEGRO;
31              z.padre.padre.color = ROJO;
32              rotarIzq(z.padre.padre);
33          }
34      }
35  }
36  root.color = NEGRO;
37 }
```

---

Figura 3: Ejemplo de caso 1 de inserción en árbol rojo negro

Figura 4: Ejemplo de caso 2 de inserción en árbol rojo negro

**Caso 1: El tío de  $z$  ( $y$ ) es rojo** El caso 1 se ejecuta cuando tanto  $z.padre$  e  $y$  son rojos. Puesto que  $z.padre.padre$  es negro, se pueden colorear  $z.padre$  e  $y$  como negros, corrigiendo así el problema de que  $z$  y  $z.padre$  sean ambos rojos, y colorear  $z.padre.padre$  como rojo, manteniendo de esta manera la propiedad 5.

Luego se repite el ciclo con  $z.padre.padre$  como el nuevo nodo  $z$ .

La Figura 3 muestra un ejemplo de inserción en árbol rojo-negro donde se presenta el caso 1 dos veces en el ciclo.

Nótese que si al final del ciclo la raíz queda roja, la violación es corregida en la línea 27 estableciendo que la raíz debe ser negra.

**Caso 3: El tío de  $z$  ( $y$ ) es negro y  $z$  es el hijo izquierdo de su padre** En el caso 3 tanto  $z$  como  $z.padre$  son rojos y  $z.padre.padre$  es negro. La acción a realizar en este caso consiste en hacer negro a  $z.padre$  y rojo a  $z.padre.padre$  y realizar una rotación a la derecha de  $z.padre.padre$ . Como  $z$  y  $z.padre$  originalmente eran rojos, la rotación realizada no introduce una violación de la propiedad 5 ya que la altura-negra de los nodos no resulta afectada y como ya no quedan nodos rojos consecutivos el procedimiento está terminado.

La Figura 4 muestra un ejemplo de inserción en árbol rojo-negro donde se presenta el caso 3.

**Caso 2: El tío de  $z$  ( $y$ ) es negro y  $z$  es el hijo derecho de su padre** En el caso 2  $z$  es el hijo derecho de  $z.padre$ . En este caso se procede simplemente a realizar una rotación a la izquierda para transformar el problema en el caso 3. Al terminar la rotación se considerará  $z$  al antiguo  $z.padre$  que fue rotado.

### 1.3. Eliminación

La eliminación de un nodo en un árbol rojo-negro con  $n$  elementos puede realizarse en un tiempo  $O(\lg n)$ . Utilizamos una versión modificada de la rutina de eliminación en un árbol binario de búsqueda ordinario:

Puede observarse que el código es casi igual a la rutina de inserción en un árbol binario de búsqueda normal, salvo por las siguientes diferencias:

1. En lugar de usar el `null` ordinario, utilizamos la constante `NULL` ya que necesitamos que las hojas tengan color NEGRO.
2. Se hace una asignación incondicional en la línea 10 (en un árbol binario de búsqueda normal sólo se puede hacer esta asignación si  $x$  no es nulo). En caso de que  $x$  sea la constante `NULL`, esta asignación facilitará la codificación de `corregirEliminar()`.

---

**Algorithm 5** Eliminación de un nodo en un árbol rojo-negro

---

```
public Nodo eliminar(Nodo z){
1   Nodo x,y;
2   if (z.izq!=NULL && z.der!=NULL)
3       y=buscarMax(z.izq); //también sirve buscarMin(z.der)
4   else
5       y=z;
6   if (y.izq != NULL)
7       x = y.izq;
8   else
9       x = y.right;
10  x.padre = y.padre;
11  if (y.padre == NULL)
12      root = x;
13  else if (y == y.padre.izq)
14      y.padre.izq = x;
15  else
16      y.padre.der = x;
17  if (y != z)
18      z.clave = y.clave; //copiar datos adicionales si aplica
19  if (y.color == NEGRO) {
20      corregirEliminar(x);
21  return y;
}
```

---



3. Se invoca a la rutina `corregirEliminación(z)` para restaurar las propiedades de árbol rojo-negro en caso de que el nodo “desaparecido” sea negro.

Nótese que si el nodo desaparecido es negro, pueden presentarse tres problemas:

1. Si  $y$  era la raíz y un hijo rojo de  $y$  se convierte en la nueva raíz, se viola la propiedad 2.
2. Si tanto  $x$  como  $y.padre$  (que también es  $x.padre$ ) eran rojos, entonces se viola la propiedad 4.
3. La remoción de  $y$  hace que cualquier ruta que previamente contenía a  $y$  ahora tenga un nodo negro menos. Por lo tanto se viola la propiedad 5. Para evitar lidiar con este problema, asumiremos que la negrura del nodo desaparecido  $y$  se le transfiere a su hijo  $x$ . El problema ahora es que  $x$  no es ni negro ni rojo (violando la propiedad 1), sino que es *doble negro* o *rojo-negro* y contribuye 2 o 1, respectivamente, al conteo de nodos negros en las rutas que lo contengan. El atributo *color* de  $x$  seguirá siendo ROJO (si es rojo-negro) o NEGRO (si es negro-negro). En otras palabras, el atributo negro extra de un nodo se refleja por el hecho de que  $x$  apunte a él y no en el atributo *color*.

A continuación examinaremos cómo el procedimiento `corregirEliminar` restaura las propiedades de árbol rojo-negro.

Dentro del ciclo **while**,  $x$  siempre apunta a un nodo doble-negro distinto a la raíz. En la segunda línea determinamos si  $x$  es un hijo izquierdo de su padre  $x.padre$ . La situación cuando  $x$  es el hijo derecho es simétrica, así que nos concentraremos sólo en el primer caso.

Nótese que el propósito del ciclo **while** es mover el negro extra hasta que:

1.  $x$  apunte a un nodo rojo-negro, en cuyo caso se colorea como negro (no doble) fuera del ciclo, en la última línea,
2.  $x$  apunta a la raíz, en cuyo caso la negrura extra puede ser simplemente “desaparecida”, o
3. se puedan ejecutar rotaciones y cambios de colores apropiados.

En primer lugar mantenemos un apuntador  $w$  al hermano de  $x$ . Puesto que  $x$  es doble negro,  $w$  no puede ser NULL; de otro modo el número de nodos negros en la ruta de  $x.padre$  a la hoja  $w$  sería menor que el número en la ruta de  $x.padre$  a  $x$ .

En el código se indican cuatro casos. La idea clave en cada caso es observar que el número de nodos negros (incluyendo el negro extra de  $x$ ) desde la raíz del subárbol a cada uno de sus subárboles es preservado por la transformación. Así, si la propiedad 5 se cumple antes de la transformación, también se cumple después. Por ejemplo, para el Caso 1, puede verse en la Figura 5 que el número de negros desde la raíz del subárbol a los subárboles  $\alpha$  y  $\beta$  es 3, la cual es preservada después de la transformación. Lo mismo ocurre con los otros subárboles y en los otros casos.

---

**Algorithm 6** Restauración de propiedades de árbol rojo-negro

---

```
void corregirEliminar(Nodo x) {
    Nodo w;
    while (x!=root && x.color == NEGRO) {
        if (x == x.padre.left) {
            w = x.padre.right;
            if (w.color == ROJO) {
1                w.color = NEGRO;
1                x.padre.color = ROJO;
1                rotarIzq(x.padre);
1                w = x.padre.der;
            }
            if (w.izq.color == NEGRO && w.der.color == NEGRO) {
2                w.color = ROJO;
2                x = x.padre;
            } else {
                if (w.der.color == NEGRO) {
3                    w.izq.color = NEGRO;
3                    w.color = ROJO;
3                    rotarDer(w);
3                    w = x.padre.der;
                }
4                w.color = x.padre.color;
4                x.padre.color = NEGRO;
4                w.der.color = NEGRO;
4                rotarIzq(x.padre);
4                x = root;
            }
        } else {
            //lo mismo, pero intercambiando izq y der
        }
        x.color = NEGRO;
    }
}
```

---

Figura

Figura 5: Casos dentro del ciclo **while** de `corregirEliminar()`

Figura

Figura 6: Ejemplo del Caso 4 de `corregirEliminar()`

Figura

Figura 7: Ejemplo del Caso 3 de `corregirEliminar()`

**Caso 4: El hermano ( $w$ ) de  $x$  es negro y el hijo derecho de  $w$  es rojo** En este caso intercambiando algunos colores (haciendo que  $w$  tenga el color del padre y que  $w.padre$  y  $w.der$  sean negros) y haciendo una rotación a la izquierda podemos eliminar la negrura extra de  $x$  sin violar ninguna de las propiedades de árbol rojo negro. Nótese que por la manera en que se asignaron los colores es imposible que se viole la propiedad 4 ya que aunque  $w$  podría convertirse en rojo, su hijo derecho se convierte en negro. El hijo izquierdo de  $w$  es potencialmente rojo, pero su nuevo padre después de la rotación será  $w.padre$  el cual previamente había se había coloreado de negro. Finalmente, podría ser que  $w$  quede rojo y que quede como raíz del árbol, pero fuera del ciclo se corrige esta situación.

La Figura 6 muestra un ejemplo de una eliminación que produce el Caso 4 y cómo el resultado final es un árbol rojo-negro válido.

**Caso 3: El hermano ( $w$ ) de  $x$  es negro, el hijo izquierdo de  $w$  es rojo y el hijo derecho de  $w$  es negro** En este caso se pueden intercambiar los colores de  $w$  y su hijo izquierdo  $w.izq$  y luego ejecutar un rotación a la derecha de  $w$  sin violar las propiedades. Esto no nos permite desaparecer la negrura extra de  $x$ , pero nos permite transformar el problema al Caso 4, que resolvimos previamente.

La Figura 7 muestra un ejemplo de una eliminación que produce el Caso 4 y cómo el resultado final es un árbol rojo-negro válido.

**Caso 2: El hermano ( $w$ ) de  $x$  es negro, los dos hijos de  $w$  son negros** En este caso quitamos negrura tanto de  $x$  (moviendo el apuntador) como de  $w$  (convirtiéndolo en negro) y movemos la negrura al padre de ambos (haciendo que  $x$  apunte ahora  $x.padre$ ). Después de esto, el ciclo se repite, a menos que la negrura extra se haya movido a la raíz del árbol, de donde se puede desaparecer sin problemas.

La Figura 8 muestra un ejemplo de una eliminación que produce el Caso 4 y cómo el resultado final es un árbol rojo-negro válido.

**Caso 1: El hermano ( $w$ ) de  $x$  es rojo** Como los dos hijos de  $w$  tienen que ser negros, se pueden intercambiar los colores de  $w$  y  $x.padre$  (que es negro, ya que tiene un hijo rojo) y luego realizar

Figura

Figura 8: Ejemplo del Caso 2 de `corregirEliminar()`

Figura

Figura 9: Ejemplo del Caso 1 de `corregirEliminar()`

una rotación a la izquierda de  $x.padre$ . Después de la rotación, el hermano de  $x$  será uno de los antiguos hijos negros de  $w$ , convirtiéndose así el problema en uno de los casos 2, 3 o 4.

La Figura 9 muestra un ejemplo de una eliminación que produce el Caso 4 y cómo el resultado final es un árbol rojo-negro válido.