19 | Árbol rojinegro

Meta

Que el alumno domine el manejo de información almacenada en un *Árbol binario ordenado balanceado*, implementando un *Árbol Rojinegro*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar cómo se almacenan los datos en una estructura no lineal.
- Entender el comportamiento de un Árbol Rojinegro.
- Experimentar el uso de la herencia de orientación a objetos para reutilizar algoritmos, refactorizando el código de la práctica anterior según se requiera.

Código Auxiliar 19.1: Árbol rojinegro

https://github.com/computacion-ciencias/ed-arbol-rojinegro-cs

Antecedentes

Definición 19.1: Árbol Rojinegro

Un **Árbol Rojinegro** es un árbol binario de búsqueda que cumple con las propiedades siguientes:

1. Todo nodo tiene un atributo de color cuyo valor es rojo o negro.
2. La raíz es de color negro.
3. Todas las hojas (árboles vacíos) son también de color negro.
4. Un nodo rojo tiene 2 hijos negros.

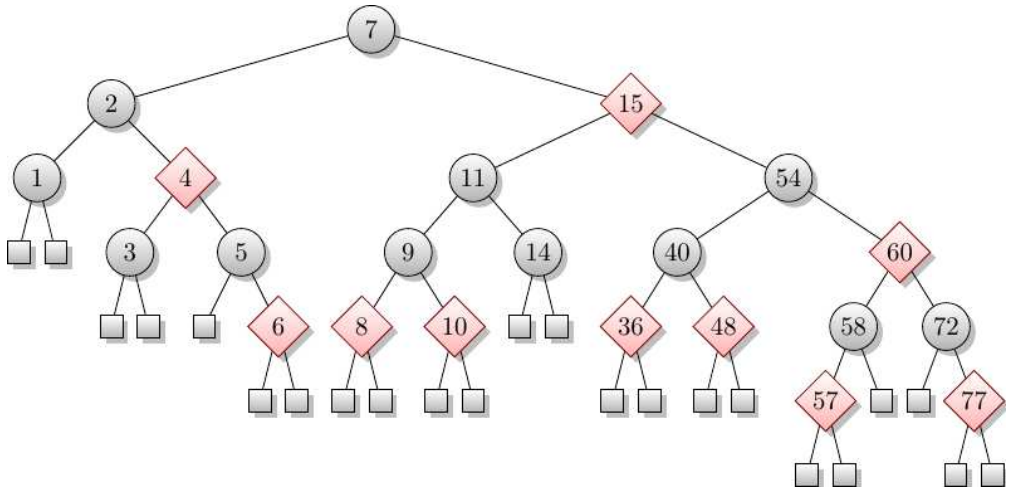

Figura 19.1 Ejemplo de Árbol Rojinegro

5. Cualquier camino de un nodo a cualquiera de sus hojas tiene el mismo número de nodos negros (*altura negra*).

Balanceo

Para mantener las propiedades de un árbol rojinegro después de agregar o remover algún nodo se utilizan dos algoritmos. Éstos dependen de los algoritmos de inserción y remoción de árboles binarios ordenados, así como de las operaciones de rotación explicadas para árboles AVL.

Insertar

Para insertar:

- El dato se inserta como en cualquier árbol binario ordenado (en una hoja).
- Al insertar el dato, el nuevo nodo siempre se pinta de rojo y tendrá como hijos dos nodos vacíos (negros).
- Sólo hay problemas si el nodo insertado queda como hijo de un nodo rojo, lo cual viola la propiedad 4. [1]

[1] Si un nodo es rojo, sus dos hijos son negros.

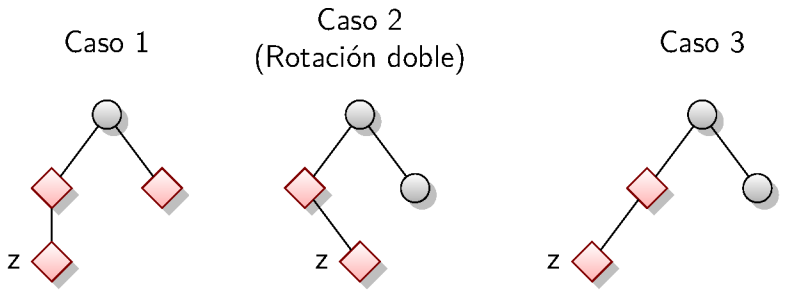

Figura 19.2 La letra z indica al nodo recién insertado. Cuando uno se dibuja justo debajo de su padre quiere decir que no importa si es hijo derecho o izquierdo.

Para recuperar la propiedad 4 se distinguen tres casos, con sus simétricos. Estos casos están dados por los colores y posiciones del nodo insertado, su padre, tío y abuelo. El esquema de la Figura 19.2 ayuda a reconocerlos fácilmente y las acciones a realizar se detallan en el Algoritmo 2.

Algoritmo 2 Balancear al insertar

```
1:  function BALANCEA(árbol, nodoAgregado)
2:      z ← nodoAgregado                                   ▷ rojo
3:      while El padre de z no sea negro do
4:          (El padre de z es un hijo izquierdo)
5:          if Caso 1: El padre y el tío del nodo nuevo son rojos. then
6:              Pintar al padre y al tío de negro.
7:              Pintar al abuelo de rojo.
8:              z ← abuelo(z).
9:          else
10:             if Caso 2: Padre rojo, tío negro y z hijo derecho. then
11:                 z ← padre(z)
12:                 Realizar rotación izquierda sobre z.
13:             end if
14:             Caso 3: Padre rojo, tío negro y z hijo izquierdo.
15:             Pintar al padre de negro.
16:             Pintar al abuelo de rojo.
17:             Realizar rotación a la derecha sobre el abuelo de z.
18:         end if
19:         (El padre de z es un hijo derecho) ...
20:     end while
21:     Pintar de negro la raíz.
22: end function
```

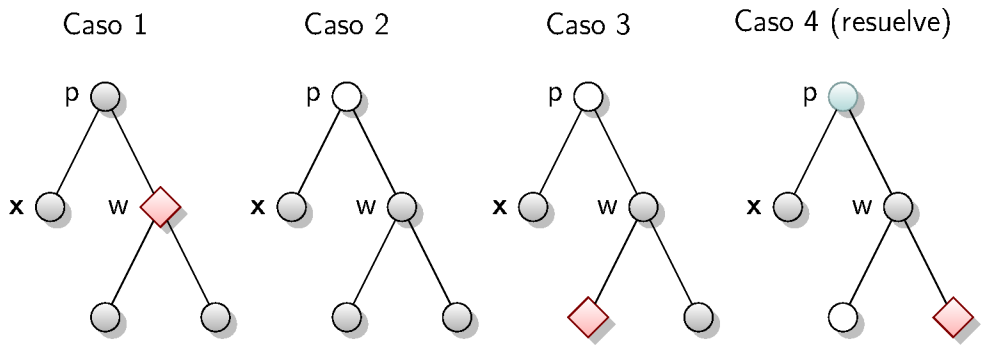

Figura 19.3 La letra x indica el color negro extra, inicialmente esta apunta al nodo removido, que en este momento sería una hoja vacía.

Remover

Para remover:

- El dato se remueve como en cualquier árbol binario ordenado.
- Sólo hay problemas si el nodo removido era negro, lo cual viola la propiedad 5. [2]
- Necesitaremos la referencia a la hoja a remover, aún conectada al árbol. Marcaremos a su padre como un nodo **doble negro**.
- Sean:

  p: el padre del nodo doble negro.

  x: el hijo de p que recibe el color $negro$ del nodo removido (nodo *doble negro*).

  w: el hermano de x.

Para eliminar al nodo *doble negro* se distinguen cuatro casos con sus simétricos. Para ello nos fijamos en los colores del padre, hermano y sobrinos del nodo removido. La Figura 19.3 indica cómo reconocerlos y el Algoritmo 3 nos dice qué hacer.

Desarrollo

Para comenzar a trabajar no olvides agregar el proyecto con los demás nodos y árboles que has programado en las prácticas anteriores. Ahí mismo agregarás el árbol rojinegro.

[2] Para cada nodo, todos los caminos del nodo a sus hijos descendientes contienen el mismo número de nodos **negros**.

Algoritmo 3 Balancear al remover

```
1:  function BALANCEA(árbol, nodoDobleNegro)
2:      x ← nodoDobleNegro                                  ▷ doble negro
3:      p ← x.padre
4:      while x no es la raíz ni es negro do
5:          if x es hijo izquierdo then
6:              w ← hermano de x
7:              if Caso 1: El hermano w es rojo. then
8:                  Pintar al hermano de negro.
9:                  Pintar al padre de rojo.
10:                 Realizar rotación izquierda sobre p.
11:                 w ← hijoDerecho(p).
12:             end if
13:             if Caso 2: Los dos sobrinos de x son negros. then
14:                 Pintar al hermano de rojo.
15:                 x ← p.
16:                 p ← padre(x).
17:             else
18:                 if Caso 3: El sobrino derecho es negro. then
19:                     Pintar al sobrino izquierdo de negro.
20:                     Pintar al hermano de rojo.
21:                     Realizar rotación derecha sobre w.
22:                     w ← hijoDerecho(p)
23:                 end if
24:                 Caso 4:
25:                 Pintar al hermano del color de p.
26:                 Pintar al padre y al sobrino derecho de negro.
27:                 Realizar rotación a la izquierda sobre p.
28:                 Sal del ciclo.
29:             end if
30:         else
31:             Casos simétricos.
32:         end if
33:     end while
34:     Si x no es null pintar de negro.
35: end function
```

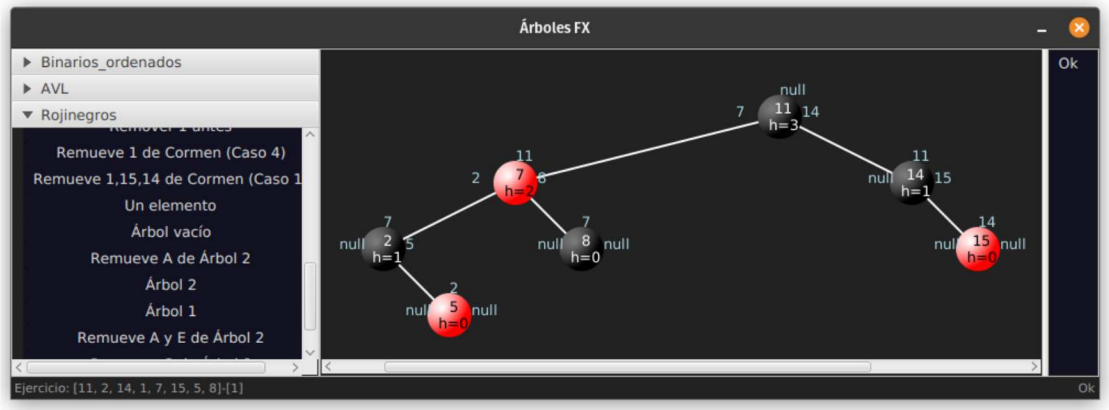

Figura 19.4 Ejemplo de árbol rojinegro.

Para implementar este tipo de árboles binarios se deberán completar las clases siguientes:

- `NodoRojinegro<C> where C : IComparable<C>`. Esta clase deberá implementar la interfaz `INodoBinarioOrdenado<C>` y extenderá `NodoBinarioOrdenado<C>`.
- `ÁrbolRojinegro<C> where C : IComparable<C>`-. Esta clase extiende `ÁrbolBinarioOrd`

1. Programa `NodoRojinegro<C>`. Una característica nueva de este tipo de nodos es la presencia del atributo **color**. Declara esta enumeración dentro del mismo espacio de nombres:

   `ED.Estructuras.NoLineales.ÁrbolBinario`

```
1 public enum Color
2 {
3   Rojo,
4   Negro
5 }
```

   Así la propiedad color será de tipo `Color`:

```
1 public Color Color { get; private set; }
```

   TIP: Agrega el método `ToString`, te ayudará a depurar tu código.

2. Agrega los constructores que necesites, llámalos desde los métodos `CreaNodo()` de la clase `ÁrbolRojinegro<C>`.

   Sobre escribe los métodos de lectura y escritura de los atributos con las referencias al padre, hijo izquierdo e hijo derecho, de modo que lancen `InvalidCastException`

si se intentan asignar nodos de una clase que no sea instancia de `NodoRojinegro<C>`. Igualmente haz que los métodos que devuelven nodos tengan `NodoRojinegro<C>` como tipo de regreso.

3. También haremos uso de orientación a objetos para aprovechar el trabajo de las prácticas pasadas. Harás una pequeña refactorización entre las clases `NodoBinarioOrdenado<C>`, `NodoAVL<C>` y la nueva `NodoRojinegro<C>`. Tras revisar los métodos para balancear árboles rojinegros, anota qué métodos de `NodoAVL<C>` también necesita `NodoRojinegro<C>`. Quita estos métodos de la clase `NodoAVL<C>` y ponlos en `NodoBinarioOrdenado<C>`. Observa que tu código debe compilar correctamente y las pruebas para árboles binarios y para árboles AVL deben funcionar exactamente igual que antes de hacer el cambio.

4. `ÁrbolRojinegro<C>`. Esta clase deberá extender

   `ÁrbolBinarioOrdenado<C>`.

   Ahora asegúrate de sobre escribir insertar/borrar/balancear en `NodoRojinegro<C>` y/o `ÁrbolRojinegro<C>` según corresponda. El árbol debe continuar siendo un árbol rojinegro válido y deben tener complejidad $\mathcal{O}(\log_2(n))$ [Cormen y col. 2009].

   TIP: En la clase árbol, en lugar de pedir el color de un nodo usando su propiedad, crea un método auxiliar que reciba el nodo y devuelva el color, de modo que, si el nodo es `null`, devuelva el color negro.

   TIP: Para balancear el árbol después de remover un nodo usa el auxiliar que devuelve la hoja que quieres quitar. Ejecuta el algoritmo de rebalanceo a partir de **su padre**, sin haberla quitado aún del árbol. Quítala al final. No olvides actualizar el atributo raíz del árbol si ésta fue modificada por alguna rotación.

   TIP: No olvides la propiedad `Count`.

Preguntas

1. ¿Qué ventajas encuentras sobre los árboles AVL? ¿Qué desventajas?

2. Desde el punto de vista de orientación a objetos ¿Por qué es válido poner los métodos para balancear nodos en `NodoBinarioOrdenado<C>`? ¿Por qué las pruebas para árboles binarios no balanceados siguen funcionando como antes?