14 | Árbol rojinegro

Meta

Que el alumno domine el manejo de información almacenada en un *Árbol binario ordenado*, 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.
- Programar dicha estructura en un lenguaje orientado a objetos, reutilizando los algoritmos implementados anteriormente.

Antecedentes

Definición 14.1: 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.
5. Cualquier camino de un nodo a cualquiera de sus hojas tiene el mismo número de nodos negros (*altura negra*).

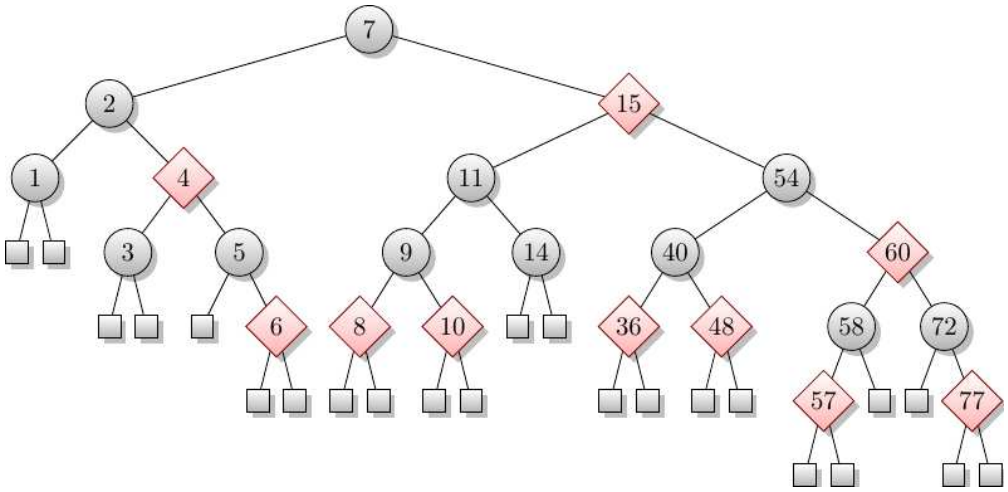

Figura 14.1 Ejemplo de Árbol Rojinegro

Desarrollo

Para comenzar a trabajar no olvides agregar tus clases `ColeccionAbstracta<E>`, `NodoBOLigado<C>` y las demás de árboles que has programado en las prácticas anteriores. Entonces deberás poder ejecutar el visualizador con los comandos:

```
$ make compile
$ make run
```

Si no funciona, revisa que no hayas omitido nada, como editar el `Makefile` para poner la dirección donde descargaste `JavaFX`.

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

- `NodoRojinegro<C extends Comparable<C> >`. Esta clase deberá implementar la interfaz `NodoBinario<E>` y extenderá `NodoBOLigado<C>`.
- `ÁrbolRojiengro<C extends Comparable<C> >`. Esta clase extiende `ÁrbolBOLigado<C>`.

Ya se te dan estos archivos, para que puedas correr las visualizaciones, pero falta completarlos.

1. Completa `NodoRojinegro<C extends Comparable<C> >` agregando los constructores que necesites, llámalos desde el método `creaNodo` de la clase `ÁrbolRojinegro<C>`.

   Sobre escribe los métodos de escritura, de modo que lancen `ClassCastException` 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.

2. También haremos uso de orientación a objetos para aprovechar el trabajo de las prácticas pasadas. Harás una pequeña refactorización entre las clases `NodoBOLigado<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 `NodoBinario<C>`. Observa que tu código debe compilar correctamente y los demos para árboles binarios y para árboles AVL deben funcionar exactamente igual que antes de hacer el cambio.

   Ahora asegúrate de implementar insertar/borrar/balancear en `NodoRojinegro<C>` según corresponda y agrega:

   - `public Color color();`
   - `public void color(Color c);`

   Observa que, para declarar el color de los nodos se utilizará una enumeración que declararás dentro de `NodoRojinegro<C>`:

```
public enum Color{
  ROJO,
  NEGRO
}
```

   Así nuestro atributo color será de tipo `Color`: `private Color color;`

3. `ÁrbolRojinegro<C extends Comparable<C>>`.

   Esta clase deberá extender `ÁrbolBOLigado<E>`. Se añade el requisito de que al agregar o remover nodos, el árbol debe continuar siendo un árbol rojinegro válido por lo que estos métodos deberán ser sobre-escritos y deben tener complejidad $\mathcal{O}(\log n)$ Cormen y col. 2009.

Para ver los árboles de manera gráfica, se provee un paquete que facilita mostrarlos.

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 `NodoBinario`? ¿Por qué los demos para árboles binarios no balanceados siguen funcionando como antes?