18 | Árbol AVL

Meta

Que el alumno domine el manejo de información almacenada en una *Árbol binario ordenado balanceado*, implementado un *Árbol AVL*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar cómo se almacenan los datos en una estructura no lineal.
- Programar dicha estructura en un lenguaje orientado a objetos, reutilizando los algoritmos implementados anteriormente.
- Analizar la eficiencia de un árbol autobalanceado.
- Dominar el uso de referencias para conectar los nodos de una estructura de datos.

Código Auxiliar 18.1: Árbol AVL

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

Antecedentes

Si conseguimos que un árbol binario ordenado mantenga la mayoría de sus datos cerca de la raíz, en lugar de que algunas ramas sean mucho más profundas que otras, la complejidad de las operaciones de búsqueda, inserción y remoción se mantendrán en el orden $\mathcal{O}(\log_2(n))$, en lugar de degenerar hasta $\mathcal{O}(n)$. Con este fin se han diseñado los árboles balanceados como el árbol AVL.

Definición 18.1: Árbol AVL

Un **Árbol AVL** es un árbol binario tal que, para cada nodo, el valor absoluto de la diferencia entre las alturas de los subárboles izquierdo y derecho es a lo más uno. En otras palabras:

1. Un árbol vacío es un árbol AVL.
2. Si A es un árbol no vacío y A_i y A_d sus subárboles, entonces A es AVL si y sólo si:
   a) A_i es AVL.
   b) A_d es AVL.
   c) Sea el *factor de balanceo* fb la diferencia de alturas entre los subárboles, entonces:

$$|fb| = |altura(A_i) - altura(A_d)| <= 1$$

Los árboles AVL toman su nombre de las iniciales de los apellidos de sus inventores, Georgii Adelson-Velskii y Yevgeniy Landis.

Balanceo

La forma más sencilla de construir y mantener árboles AVL es realizar una verficación sobre su condición de balanceo, justo después de cada inserción/remoción de un dato y corregir la estructura del árbol inmediatamente, de ser necesario. Para esto, se utilizan los algoritmos de inserción y remoción que vimos para árboles binarios ordenados, seguidos de un algoritmo de balanceo.

Este algoritmo depende de la definición de **factor de balanceo** dada arriba y de las operaciones de rotación. Hay dos operaciones de rotación y son simétricas, por lo que sólo se presenta la rotación a la derecha, siendo su simétrica la rotación a la izquierda. Esta rotación se ilustra en la Figura 18.1, donde decimos que la rotación se está aplicando sobre el nodo C. Los cambios que ocurren como resultado son:

1. El hijo derecho de B se convierte en hijo izquierdo de C. Si este hijo no es un árbol vacío, debe actualizar la referencia a su padre, que ahora es C.

2. C se convierte en el hijo derecho de B y simétricamente, B se convierte en padre de C.

3. Si C tenía padre, B se convierte en el hijo de ese nodo, en la misma posición que ocupaba C. Si C no tenía padre quiere decir que C era la raíz del árbol y ahora lo será B.

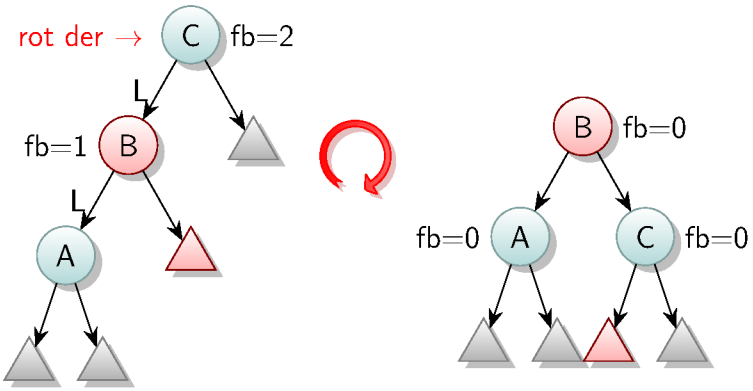

Figura 18.1 Rotación a la derecha (LL) sobre el nodo C. La nueva raíz del árbol que iniciaba en C ahora es B.

Al programar la rotación es importante no perder referencias a los nodos antes de asignarles su nueva posición, por lo que se debe recurrir a las referencias temporales pertienentes. En las precondiciones de la operación asumiremos que sólo se manda llamar sobre nodos cuyo factor de balanceo indique la necesidad de realizarla, según el algoritmo de balanceo que se detalla a continuación, de otro modo varias de las reasignaciones no podría ser llevadas a cabo.

El algoritmo de rebalanceo completo se muestra en el Algoritmo 1, que se ejecuta a partir del nodo que acaba de ser agregado o removido.

Desarrollo

Para implementar el **Árbol AVL** se deben implementar las siguientes clases:

- `NodoAVL<C>`. Esta clase debe implementar la interfaz `INodoBinarioOrdenado<C>`, hereda de `NodoBinarioOrdenado<C>`.

  Observa que, aunque lo segundo implica lo primero, es frecuente declarar ambas cosas explícitamente en una clase para remarcar el hecho de que se cumple con el contrato establecido por la interfaz.

- `ÁrbolAVL<C>`. Esta clase extiende `ÁrbolBinarioOrdenado<C>`. Obseva que no se define una interfaz nueva, pues `ÁrbolAVL<C>` no define un tipo de dato abstracto, por el contrario es una forma [eficiente] de implementar un árbol binario ordenado.

Programa tus componentes según las indicaciones siguientes:

1. Crea los constructores que necesites para `NodoAVL<C>` aunque, recuerda, no los llamarás más que desde los métodos `CreaNodo` de la clase `ÁrbolAVL<C>`.

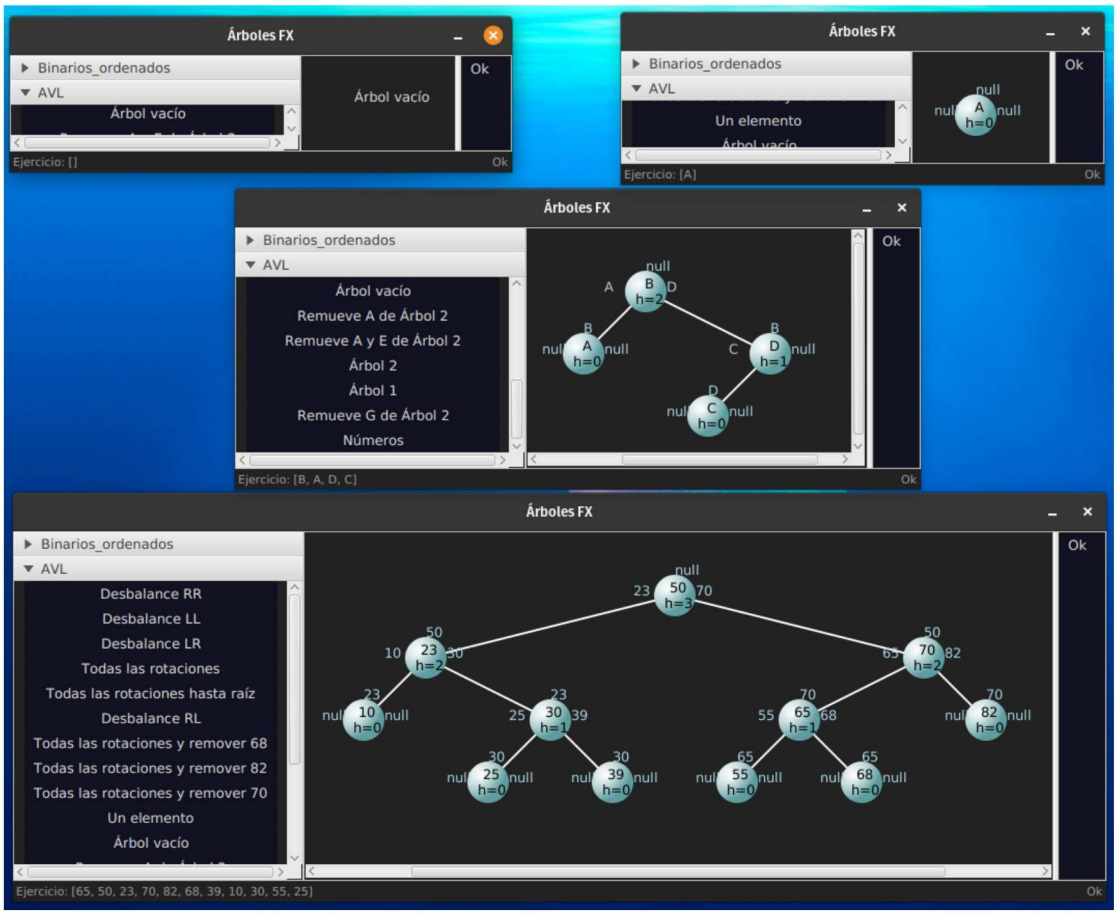

Figura 18.2 Muestra de algunos árboles AVL.

Algoritmo 1 Balancear

```
1:  function BALANCEA(árbol, nodoModificado)
2:      temp ← nodoModificado ▷ Rabalancear a partir de nodoModificado.padre
3:      while (temp ← temp.padre) = ∅ do
4:          ACTUALIZAALTURA(temp)
5:          fb ← FACTORDEBALANCEO(temp)
6:          if fb = −2 then                                   ▷ Lado derecho (R)
7:              if FACTORDEBALANCEO(temp.hijoD) = 1 then      ▷ Lado izquierdo (L)
8:                  ROTACIÓNDERECHA(temp.hijoD)
9:              end if
10:             nr ← ROTACIÓNIZQUIERDA(temp)
11:             if temp = raíz then
12:                 raíz = nr
13:             end if
14:             return
15:         else
16:             Caso simétrico
17:         end if
18:         if temp.alturaPrevia = temp.alturaNueva then
19:             return
20:         end if
21:     end while
22: end function
```

2. Como C# no soporta covarianza para propiedades de lectura y escritura, pero sí para métodos *virtuales* la interfaz `INodoBinarioOrdenado<C>` define este tipo de métodos de forma manual. Ejemplo:

```
1       INodoBinarioOrdenado<C>? Padre();
2       void Padre(INodoBinarioOrdenado<C>? padre);
```

   En tu clase `NodoBinarioOrdenado<C>` utiliza un atributo explícitamente y programa los métodos de lectura y escritura de la forma siguiente:

```
1       protected NodoBinarioOrdenado<C>? _padre;
2       public virtual NodoBinarioOrdenado<C>? Padre() => _padre;
3       public virtual void Padre(NodoBinarioOrdenado<C>? padre)
4       {
5           _padre = padre;
6       }
7       INodoBinarioOrdenado<C>? INodoBinarioOrdenado<C>.Padre() {
            ↪ return Padre(); }
8       void INodoBinarioOrdenado<C>.Padre(INodoBinarioOrdenado<C>?
            ↪  padre)
9       {
```

```
        if (padre == null) Padre(null);
        else if(padre.GetType() is NodoBinarioOrdenado<C>)
        {
            Padre((NodoBinarioOrdenado<C>?)padre);
        }
        throw new ArgumentException("Agumento de clase no admitida");
    }
```

Haz lo mismo para `HijoI` e `HijoD` y ajusta el resto de tu código para que funcione con estos cambios.

3. Para el árbol AVL sobreescribe una vez más los métodos de lectura y escritura, de modo que trabajen con el tipo más específico `NodoAVL<C>`. Observa que no se puede cambiar el tipo del argumento del método de lectura, porque entonces no se trataría de una sobreescritura, así que verificaremos que el tipo de nodo asignado sea correcto de forma dinámica.

```
    public override NodoAVL<C>? Padre() => (NodoAVL<C>?)_padre;
    public override void Padre(NodoBinarioOrdenado<C>? padre)
    {
        if (padre == null) Padre(null);
        else if(padre.GetType() is NodoAVL<C>)
        {
            Padre((NodoAVL<C>?)padre);
        }
        throw new InvalidCastException("Agumento de clase no admitida");
    }
```

Sobre escribe los métodos de escritura para los nodos `Padre`, `HijoI` e `HijoD`, de modo que lancen `InvalidCastException` si se intentan asignar nodos de una clase que no sea instancia de `NodoAVL<C>`.

Haz que los métodos que devuelven nodos tengan `NodoAVL<C>` como tipo de regreso. Para ello puedes simplemente llamar al método de la super clase o acceder directamente al atributo y hacer la conversión de tipo en tu método; te ahorrará muchas conversiones fuera de esta clase. Puedes hacer lo mismo con el método `raíz()` de `ÁrbolAVL<C>`, pero dejamos esto a tu elección.

4. Programa en `NodoAVL<C>` los métodos para:

   a) Realizar rotaciones izquierda y derecha sobre el nodo. Este nodo debe devolver al nodo que quede como raíz del subárbol. Actualiza aquí el valor del atributo altura para los nodos cuya altura se vea afectada por la rotación.

   b) Balancear un nodo, dado que su factor de balanceo es 2 ó -2. Este método debe devolver al nodo que quedó como raíz del subárbol después de rebalancear. No olvides actualizar las alturas de los nodos afectados después de balancear.

5. En `ÁrbolAVL<C>` sobreescribe los métodos para agregar y eliminar de tal modo que se agreguen los pasos para balancear el árbol. Asegúrate de que `Add` y `Remove` sean `virtual` en `ÁrbolBinarioOrdenado<C>` y `override` en `ÁrbolAVL<C>`. Observa que no es necesario eliminar el código que ya tenías para agregar y remover los nodos. Sólo falta recorrer el árbol desde el nodo modificado, hacia arriba, actualizando alturas y revisando los factores de balanceo. Programa entonces un método auxiliar que reciba el nodo recién agregado/removido y realice la actualización de alturas y verificación de balanceo a partir de su padre y hasta que ya no sea necesario hacer más modificaciones (este punto está determinado por el algoritmo).

   Recuerda que, si al balancear cambia la raíz del árbol, debes actualizar el atributo correspondiente en la clase `ÁrbolAVL<C>`.

   TIP: Programa un método `ToString` con recorrido en amplitud para que puedas darte una idea de dónde están acomodados tus nodos. Te servirá para depurar tu código. Puedes agregar temporalmente `Console.WriteLine(árbol.ToString())` en las pruebas unitarias para ver cómo cambia tu estructura cada vez que insertas o remueves un dato.

6. Observa que el iterador de la práctica pasada sigue funcionando.

Preguntas

1. Explica cómo se sabe si se hace una rotación izquierda, derecha o una doble rotación, para balancear el árbol.