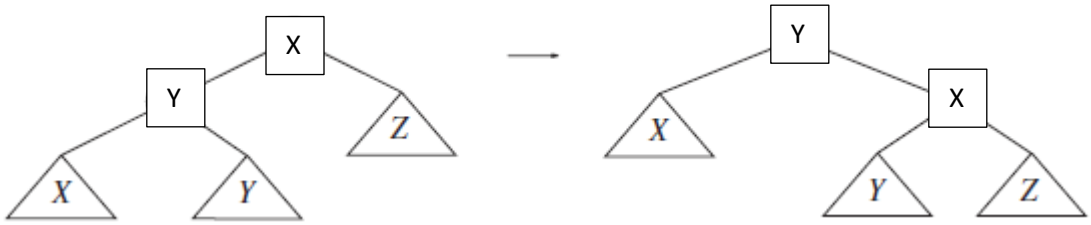
Práctica: AVL Trees
25 y 27 de octubre – Estructura de Datos

---

1. Ejercicio teórico: Ingresa los siguientes datos a un árbol AVL, dibuja el árbol cada vez que haya un ingreso, si se desequilibra, aplica la rotación adecuada, dibuja el árbol nuevamente e indica qué tipo de rotación hiciste y en qué nodo apareció el desequilibrio: 84, 10, 8, 92, 66, 88, 29, 27, 75, 72, 68, 62, 18, 80, 36, 1, 40.
2. **Crea la clase AVLTree**. Que deberá ser genérica. Utiliza la letra E como referencia a tipo de dato y no se te olvide que un AVL es un BST, para que limites los datos que pueden guardarse en un AVL.
3. **Crea la clase privada Node**, que tendrá como variables de instancia, el elemento. referencia a su hijo izquierdo y al derecho, además de contar con la información de la altura de cada nodo.
4. Crea el **constructor de la clase Node**, que reciba, el elemento y los hijos derecho e izquierdo (no olvides inicializar la altura de manera adecuada).
5. Crea un método **toString** que imprima el elemento y la altura de ese nodo en el siguiente formato [elem – height]
6. Declara la variable de instancia de AVL, llámala root.
7. Agrega **private static final int ALLOWED_IMBALANCE = 1**;
8. Crea el constructor que inicie un árbol a vacío.
9. Copia y adapta a esta clase el **método inOrder()** de tu clase ABB. O el iterador. Para que tengas alguna manera de recorrer e imprimir el árbol.
10. Agrega el **método privado Node insert(E elem, Node n)** que agregue un nuevo elemento al árbol. Adapta el método privado visto en clase y crea también el **método público void insert (E element)**. Este método privado utilizará el método balance que aún no tenemos, lo agregaremos en el siguiente paso.
11. Agrega el **método privado int height(Node n)**, este método debe devolver 0 si n es nodo nulo o la correspondiente altura del nodo n.
12. Agrega el **método privado Node balance(Node n).** Este método lo que hace es detectar si se ha roto la característica de un AVL (la diferencia de altura de sus subárboles, no debe ser mayor a 1) y en caso de ser necesario balancea. Además, este método actualiza la variable de instancia height del nodo. Este método ya lo tienes en tu presentación de clase. Este método llama a los métodos que hacen las rotaciones dobles y sencillas. Las cuales implementaremos más adelante.
13. Ahora empezaremos a agregar los métodos para las rotaciones simples y dobles necesarias. Crea el **método Node rotateWithLeftChild(Node x)**, este método realiza las rotación simple hacia la derecha con los siguietes pasos:
    a. En un nodo y guardamos la referencia del izquierdo de x
    b. El izquierdo de y, ahora debe referenciar al derecho de x.
    c. El derecho de y, ahora debe referenciar a x.
    d. Corregir la altura de x, tomando la altura máxima entre su subárbol derecho e izquierdo y luego sumando uno.

e. Corregir la altura de y, tomando la altura máxima entre su subárbol derecho e izquierdo y sumando uno.
f. Devolver el nodo y

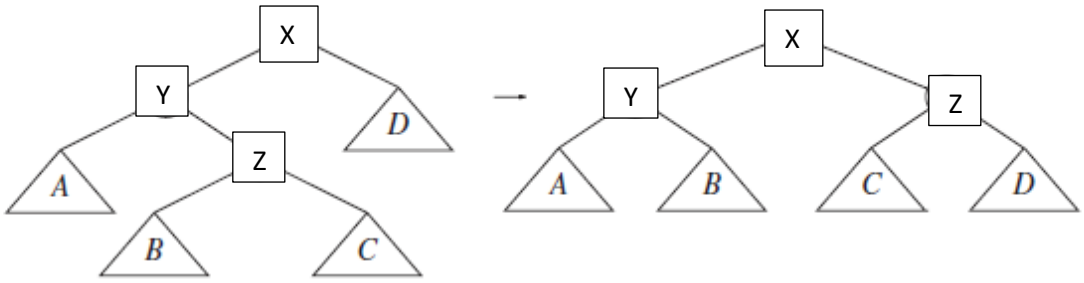


Weiss (pag. 135)

14. Crea el método **Node rotateWithRightChild(Node x)**, este método realiza las rotación simple hacia la izquierda, recuerda que este método es un método simétricamente opuesto al anterior:

15. Crea el método **Node doubleWithLeftChild(Node X)**, este método realiza una rotación doble derecha-izquierda, de acuerdo a los siguientes pasos:

    a. El nodo izquierdo de x, ahora referenciará al nodo obtenido al hacer una rotación hacia la derecha mandando como parámetro el hijo izquierdo.de X.
    b. Por último devolvemos el nodo obtenido al hacer una rotación hacia la izquierda mandando como parámetro el nodo x.



Weiss(pag, 135)

16. Crea el **método Node doubleWithRightNode(Node X)**, este método realiza una rotación doble izquierda-derecha, simétricamente opuesta al anterior.
17. Prueba tus clase.
18. Adicional: Genera un método que te imprima el árbol por niveles y acomodado de tal manera que se visualice como estructura jerárquica.

Nota: Documenta tipo javadoc y anexa comentarios de funcionalidad a tus métodos.