



## TP 5 - Árboles Binarios de Búsqueda y AVL

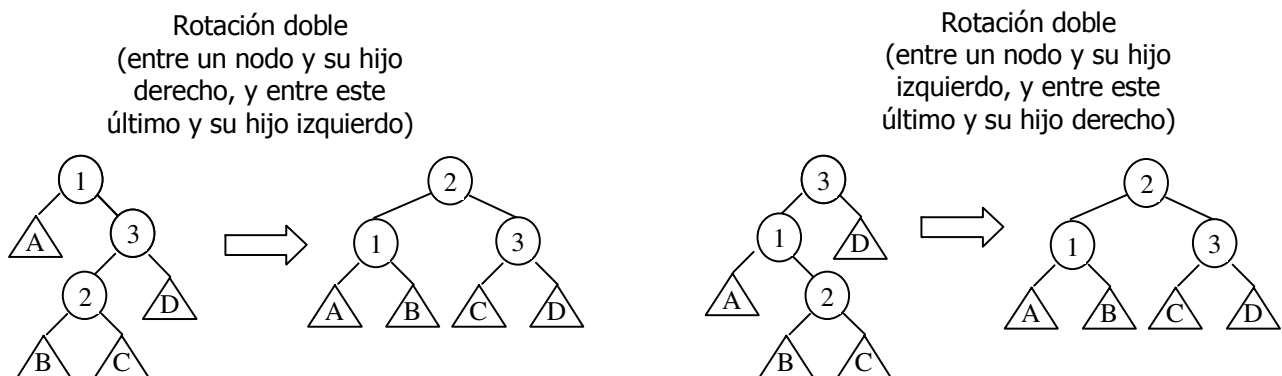
### Ejercicio 1.

- Muestre en papel (dibuje) las transformaciones que sufre un árbol binario de búsqueda (inicialmente vacío) al insertar cada uno de los siguientes elementos: 3, 1, 4, 6, 8, 2, 5, 7.
- Muestre como queda el árbol al eliminar los elementos: 7, 1 y 6.
- A partir de un árbol binario de búsqueda nuevamente vacío, muestre las transformaciones que sufre al insertar cada uno de los siguientes elementos: 5, 3, 7, 1, 8, 4, 6.
- Dibuje como queda el árbol al eliminar los elementos: 5, 3 y 7.
- ¿Qué puede concluir sobre la altura del árbol a partir de a) y c)?

### Ejercicio 2

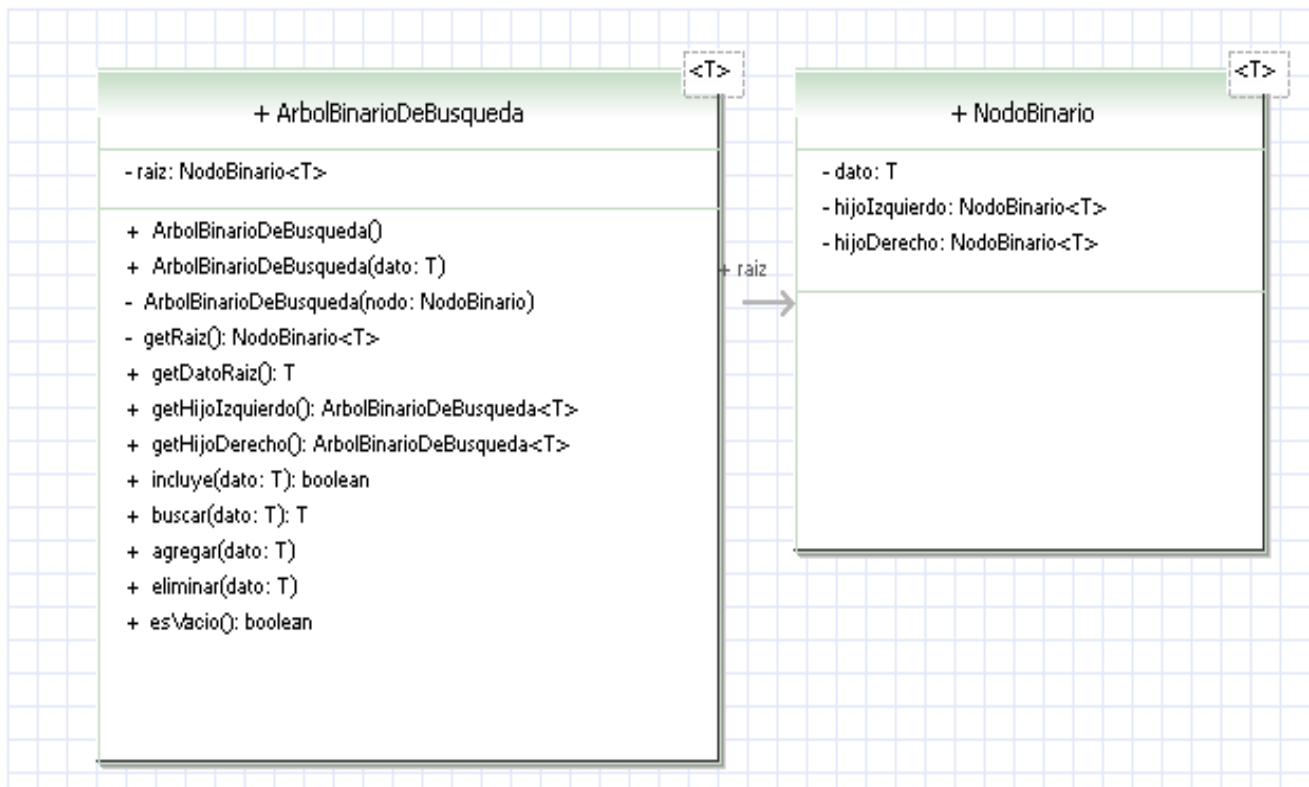
- Muestre las transformaciones que sufre un árbol AVL (inicialmente vacío) al insertar cada uno de los siguientes elementos: 40, 20, 30, 38, 33, 36, 34, 37. Indique el tipo de rotación empleado en cada balanceo.
- Dibuje como queda el árbol al eliminar los elementos: 20, 36, 37, 40. Considere que para eliminar un elemento con dos hijos lo reemplaza por el mayor de los más pequeños.

Como ayuda, se muestra a continuación las rotaciones dobles para este tipo de árbol.



### Ejercicio 3

Considere la siguiente especificación de la clase **ArbolBinarioDeBusqueda** (con la representación hijo izquierdo e hijo derecho):



#### Aclaración:

En la imagen no aparece la definición de la variable de tipo en la clase `ArbolBinarioDeBusqueda`. La misma es `<T extends Comparable<T>>`, lo cual indica que la clase representada por la variable `T` implemente la interface `Comparable`.

Las clases que implementan la interface `java.lang.Comparable<T>` permiten que sus instancias se puedan comparar entre sí. Para lograr esto, deben implementar el método `compareTo(T)`, el cual retorna el resultado de comparar el receptor del mensaje con el parámetro recibido. Este valor se codifica con un entero, el cual presenta la siguiente característica:

- = 0: si el objeto receptor es igual al pasado en el argumento.
- > 0: si el objeto receptor es mayor que el pasado como parámetro.
- < 0: si el objeto receptor es menor que el pasado como parámetro.

La descripción de cada método es la siguiente:

El constructor `ArbolBinarioDeBusqueda()` inicializa un árbol binario de búsqueda vacío con la raíz en null.

El constructor `ArbolBinarioDeBusqueda(T dato)` inicializa un árbol que tiene como raíz un nodo binario de búsqueda. Este nodo tiene el dato pasado como parámetro y ambos hijos nulos.

El constructor `ArbolBinarioDeBusqueda(NodoBinario<T> nodo)` inicializa un árbol donde el nodo pasado como parámetro es la raíz. Este método es privado y se podrá usar en la implementación de las operaciones sobre el árbol.

El método `getRaiz():NodoBinario <T>` retorna el nodo ubicado en la raíz del árbol.

El método `getDatoRaiz():T` retorna el dato almacenado en el `NodoBinario` raíz del árbol, sólo si el árbol no es vacío.

El método `esVacio(): boolean` indica si el árbol es vacío (no tiene dato cargado).



Los métodos `getHijoIzquierdo():ArbolBinarioDeBusqueda<T>` y `getHijoDerecho():ArbolBinarioDeBusqueda<T>` retornan los árboles hijos que se ubican a la izquierda y derecha del nodo raíz respectivamente. Están indefinidos para un árbol vacío.

El método `incluye (T dato)` retorna un valor booleano indicando si el dato recibido está incluido en el árbol.

El método `buscar (T dato):T` retorna el valor almacenado en el árbol que es igual al dato recibido.

El método `agregar (T dato)` agrega el dato indicado al árbol. En caso de encontrar un elemento igual dentro del árbol, reemplaza el existente por el recibido.

El método `eliminar (T dato)` elimina el dato del árbol.

Implemente la clase `ArbolBinarioDeBusqueda` con las operaciones descriptas.

**Nota:** Tener presente que a diferencia del TP anterior donde se utilizan tipos genéricos, en ABB y AVL no se pueden almacenar cualquier objeto, ya que estos necesitan ser comparables para poder ordenarlos dentro de la estructura.

## Ejercicio 5

Implemente la clase `Iterador<T>` que permite recorrer el contenido del árbol en orden. La misma tiene los métodos `hasNext():boolean` y `next():T` que permiten acceder al contenido del árbol siguiendo el orden de los elementos.

A continuación se muestra un método que imprime el contenido de un árbol binario de búsqueda de Strings referenciado por la variable `abb`:

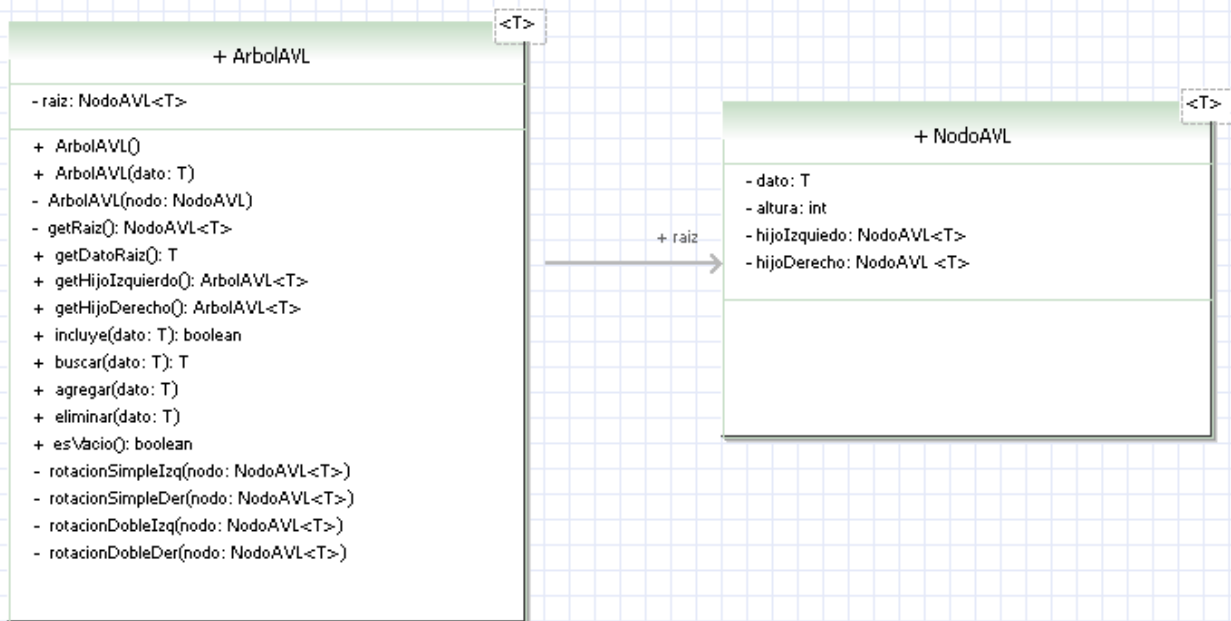
```
Iterador<String> it = new Iterador<String>(abb);  
while (it.hasNext()) {  
    String valor = it.next();  
    System.out.print(valor);  
}
```

## Ejercicio 6

Verifique las implementaciones realizadas en los ejercicios previos utilizando las pruebas de JUnit provistas.

## Ejercicio 7

Considere la siguiente especificación de la clase `ArbolAVL` (con la representación hijo izquierdo e hijo derecho):



La especificación de cada operación es análoga a las del ejercicio anterior, con la salvedad que en este caso se agregan las diferentes rotaciones que incluyen estos árboles.

Implemente las clases ArbolAVL y NodoAVL

## Ejercicio 8

Agregue a la clase **ArbolAVL** el método **esMinimal():boolean**. Un árbol AVL es minimal de altura  $h$  (es decir, si se saca algún nodo deja de ser AVL o de tener altura  $h$ ) si el número mínimo de nodos queda especificado por la siguiente recurrencia:

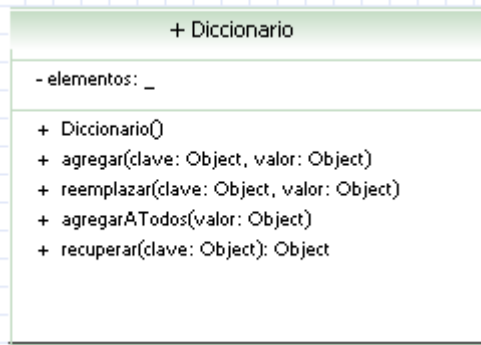
$$\min(h) = \begin{cases} 1 & \text{si } h = 0 \\ 2 & \text{si } h = 1 \\ 1 + \min(h-1) + \min(h-2) & \text{si } h > 1 \end{cases}$$

Si necesita, podría definir un método privado, en la clase ArbolAVL, **cantidadDeNodos():int** y otro **min(int h):int**, para resolver **esMinimal():boolean**.

## Ejercicio 9

Implementar en Java una solución para los siguientes problemas.

a) Defina una clase llamada **Diccionario**. El mismo contiene una colección de pares de elementos. Cada elemento está formado por una clave única y un valor asociado.



El mensaje **agregar(Object clave, Object valor)** agrega un nuevo elemento en el Diccionario, dado por el par **clave** y **valor**.

El mensaje **reemplazar(Object clave, Object valor)** reemplaza el valor del elemento con la clave dada por **clave**.

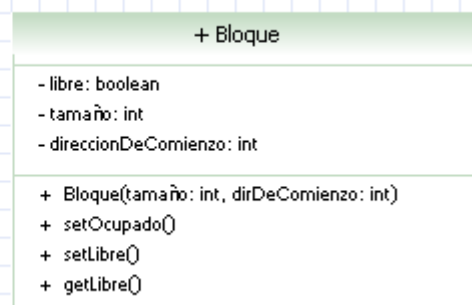
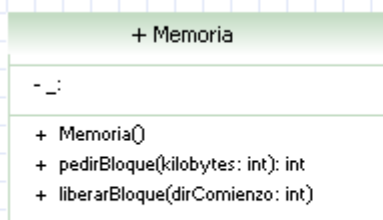
El mensaje **agregarATodos(Object valor)** reemplaza el valor de todos los elementos del Diccionario por el argumento **valor**.

El mensaje **recuperar(Object clave): Object** recupera el valor asociado a la **clave**.

Para la implementación de los métodos tenga en cuenta que **agregar()**, **reemplazar()** y **recuperar()** deben poseer tiempo de ejecución **O (log n)** en el **peor** de los casos. Determine que estructura de datos debe utilizar para poder cumplir con el orden solicitado, que cambios necesita realizar sobre ella y que estructuras adicionales necesita.

b) ¿Que cambios realizaría a la implementación del ejercicio previo para lograr que todas las operaciones tengan orden de ejecución **O (log n)** para el **peor** de los casos?

c) Se desea definir un esquema de organización de memoria dinámica, que consiste en organizar bloques de memoria en forma eficiente. Cada bloque de memoria se mide en kilobytes (en la memoria pueden existir muchos bloques de igual tamaño), se identifica por su dirección física de comienzo (representada por un número entero) y tiene un estado libre u ocupado.



El mensaje **pedirBloque(int kilobytes): int** devuelve la dirección de comienzo de un bloque de memoria libre de tamaño igual a Kilobytes. Si no hubiera devuelve -1. A su vez, cambia el estado del bloque a ocupado.

El mensaje **liberarBloque(int direccionDeComienzo)** cambia el estado del bloque a libre.

Debe elegir la o las estructuras de datos adecuadas para implementar la clase **Memoria**, de manera tal que las operaciones anteriores se resuelvan con complejidad **O (log n)** en el peor de los casos.



d) Implemente un programa que, ante un conjunto de palabras como entrada, imprima un listado ordenado de las mismas, incluyendo para cada una el número de veces que aparece.

Por ejemplo, ante la entrada:

```
hola datos algoritmos datos mundo hola
```

debe imprimir:

```
algoritmos 1  
datos 2  
hola 2  
mundo 1
```