Práctica 4: Árboles Binarios de Búsqueda y árboles AVL.

Programación III - UNLP

Cursada 2013

Todos los ejercicios de la presente práctica deberán hacerse en `Java` dentro de un mismo proyecto llamado "`Prog3_2013`". Se recomienda realizar cada ejercicio en un paquete diferente, y se recomienda el uso de `Eclipse`.

Además, se recomienda realizar todas las practicas de la presente materia dentro de un mismo proyecto de `Eclipse`, a fin de poder reutilizar código escrito entre diferentes ejercicios y prácticas.

1. a) 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.

   b) Muestre como queda el árbol al eliminar los elementos: 7, 1 y 6.

   c) 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.

   d) Dibuje como queda el árbol al eliminar los elementos: 5, 3 y 7.

   e) ¿Qué puede concluir sobre la altura del árbol a partir de a) y c)?

2. a) 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.

   b) 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.

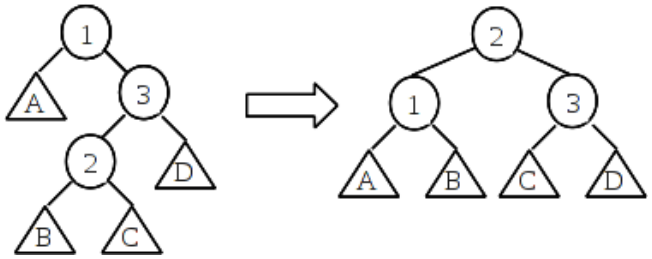

Figura 1: Ejercicio 1. Rotación doble, entre un nodo y su hijo derecho, y entre este último y su hijo izquierdo.

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

   En la imagen no aparece la definición del tipo genérico 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`.

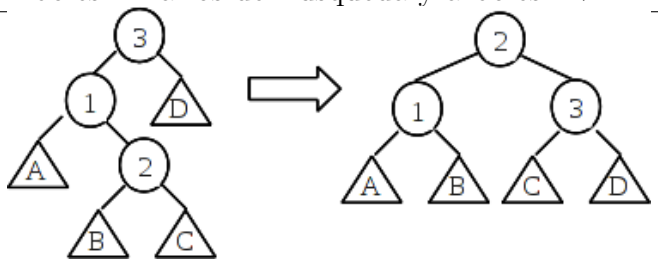

Figura 2: Ejercicio 1. Rotación doble, entre un nodo y su hijo izquierdo, y entre este último y su hijo derecho.

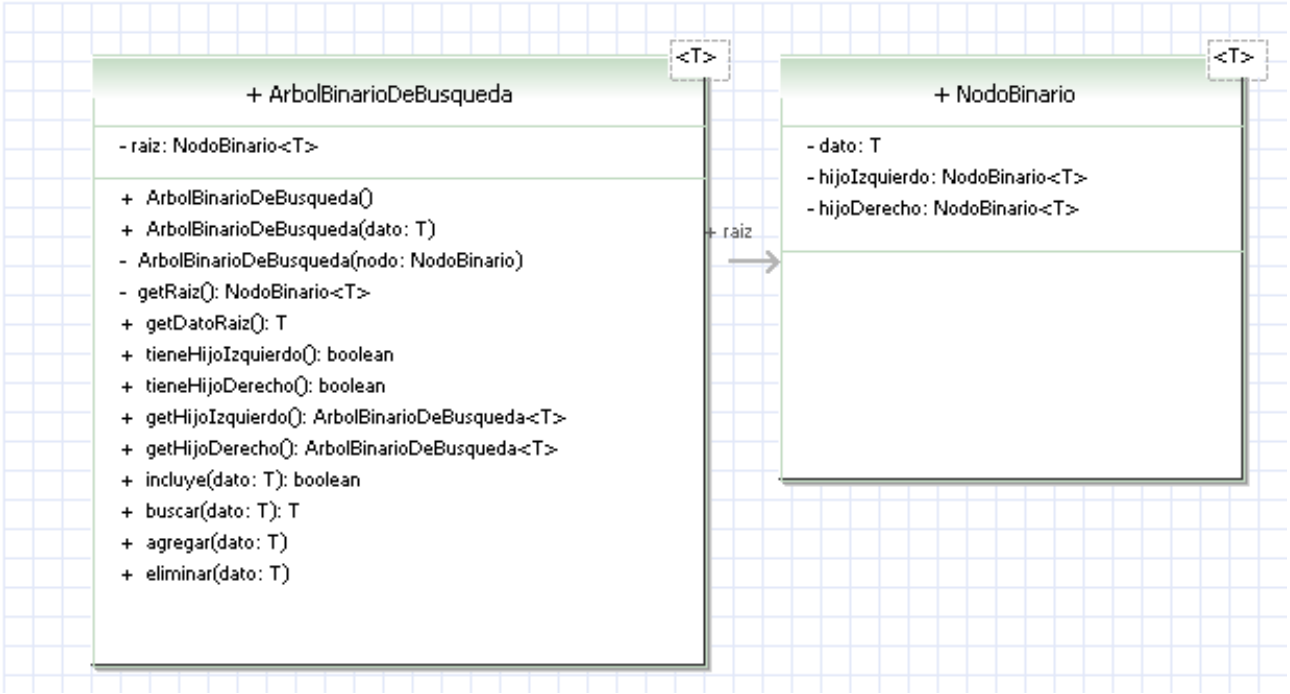

Figura 3: Ejercicio 2. `ArbolBinarioDeBusqueda`.

```
1 class ArbolBinarioDeBusqueda<T extends Comparable<T>> {
2   // ...
3 }
```

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:

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

b) 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.

c) 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.

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

e) El método `getDatoRaiz():T` retorna el dato almacenado en el NodoBinario raíz del árbol.

f) Los métodos `tieneHijoIzquierdo():boolean` y `tieneHijoDerecho():boolean` indican si existen los respectivos hijos del árbol. Ambos retornan `false` para los árboles vacíos.

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

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

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

j) 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.

k) 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.

4. 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 agregar las diferentes rotaciones que incluyen estos árboles. Implemente las clases `ArbolAVL` y `NodoAVL`.

5. 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.

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

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

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

   d) 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 $\mathcal{O}(\log n)$[1] en el peor de los casos. Determine

[1] Una operación $\mathcal{O}(\log n)$ se dice que es de tiempo de ejecución logarítmico, si el tiempo de ejecución es, en el peor caso, proporcional al logaritmo del tamaño (cantidad de elementos) de la estructura. Por lo tanto, es un tiempo mejor que el lineal, pero peor que el constante.

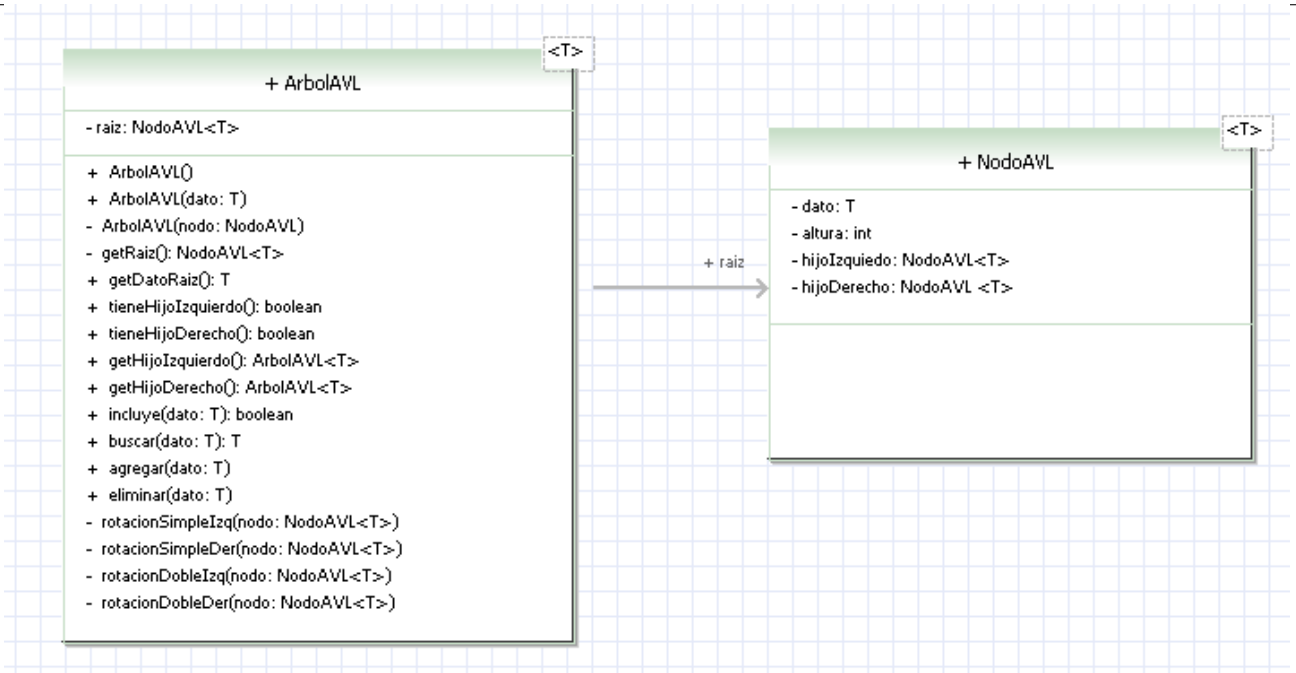

Figura 4: Ejercicio 4. `ArbolAVL` y `NodoAVL`

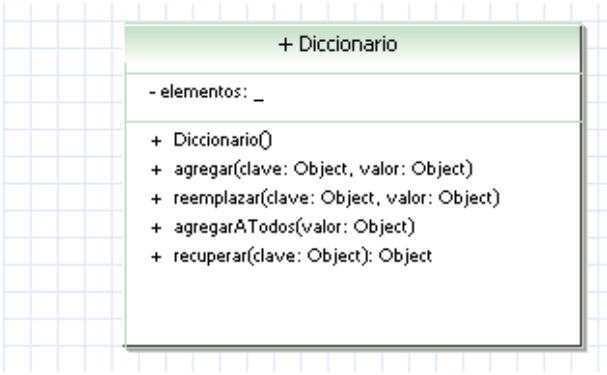

Figura 5: Ejercicio 4. `ArbolAVL` y `NodoAVL`

que estructura de datos debe utilizar para poder cumplir con el orden solicitado, qué cambios necesita realizar sobre ella, y qué estructuras adicionales necesita.

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

6. 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)` que que cambia el estado del bloque a libre.

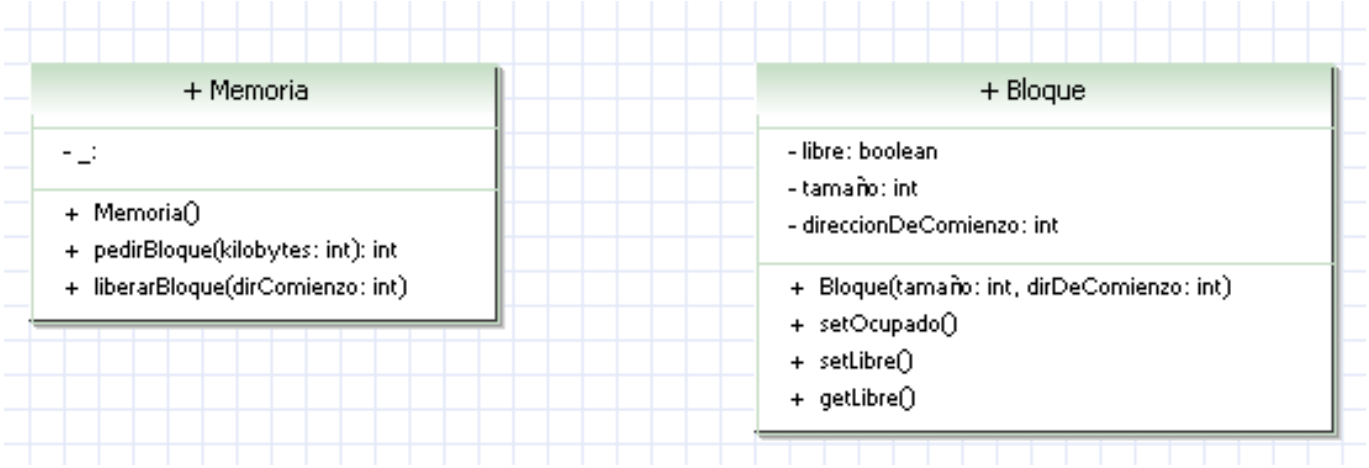

Figura 6: Ejercicio 5

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 $\mathcal{O}(\log n)$ en el peor de los casos.

7. Implemente un programa que, ante un conjunto de palabras como entrada, imprima un listado ordenado de las mismas, incluyendo para cada un 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