Algoritmos
y
Estructuras de Datos

Cursada 2011

Ejercicio

Agregue a la clase ArbolBinarioDeBusqueda el método de instancia **cumpleAVL():boolean,** el cual determina si el ArbolBinarioDeBusqueda es un árbol AVL, es decir si para cada nodo se cumple que el alto de sus subárboles difieren en a lo sumo 1.

¿Como se puede implementar?

Ejercicio

Siguiendo el enunciado se puede implementar trivialmente...

```
public boolean cumpleAVL() {
    if (this.esVacio()) {
        return true;
     }
    else  {
        return
            Math.abs(
          this.getHijoIzquierdo().altura() -
          this.getHijoDerecho().altura()) <= 1);
    }
}
```

Si definimos la altura de un árbol vacío como -1...
¿Funciona para todos los casos?

Ejercicio

Faltaba chequear que a su vez, los hijos sean también AVL.
Corrección:

```
public boolean cumpleAVL() {
    if (this.esVacio()) {
        return true;
     }
    else  {
        return
            this.getHijoIzquierdo().cumpleAVL() &&
                this.getHijoDerecho().cumpleAVL()  &&
                  Math.abs(
            this.getHijoIzquierdo().altura() -
            this.getHijoDerecho().altura()) <= 1);
    }
}
```

Funciona en todos los casos, pero con varios recorridos...

Ejercicio

Podemos definir un método que retorne la altura y haga la verificación en el mismo paso. Y si detecta que el árbol no es AVL, retorne un valor que no indique una altura válida.

Quedaría de esta forma:

```
public boolean cumpleAVL(){
    if (this.isEmpty())
            return true;
    else
            return this.chequearAlturas() >=0;
}
```

Ejercicio

```
private int chequearAlturas(){
        int altura;

        if (this.esVacio())
                altura = 0;
        else {
                int hd = 0, hi = 0;
                altura = -1;  //por defecto, inválido

                hi = this.getHijoIzquierdo().chequearAlturas();
                hd = this.getHijoDerecho().chequearAlturas();

                if (hd != -1 && hi != -1 && Math.abs( hd - hi) < 2)
                        altura = Math.max(hi, hd) + 1;
                }
        return altura;
        }
```

Ejercicio 9.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.

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.

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.

Ejercicio 8.b

La clase Memoria, contiene dos AVLs. Uno de bloques de memoria ordenados por tamaño, para liberar bloques en O(Log n). Otro de colas de bloques de igual tamaño, para extraer del comienzo un bloque libre y colocarlo como ocupado al final. De esta forma, la operacion pedir bloque también se realiza en O (Log n).

```
public class Memoria{
ArbolAVL <BloqueDeMemoria> avlOrdenadoPorDireccion
ArbolAVL <ColaDeBloques> avlOrdenadoPorTamaño}

public class ColaDeBloques{
int tamaño;
Cola <BloqueDeMemoria> colaDeBloquesConLibresAlPrincipio}

public class BloqueDeMemoria {
  private int tamaño;
  private int direccionDeComienzo;
  private bool ocupado;}
```

Ejercicio 8.b

```
public int pedirBloque(int kb){
int direccionDeComienzo = - 1;
Cola<BloqueDeMemoria> colaDeBloques = avlOrdenadoPorTamaño.buscar(kb);
BloqueDeMemoria bloque = colaDeBloques.desencolar();
If (bloque.libre()) {
        direccionComienzo = bloque.direccionDeComienzo;
        colaDeBloques.encolar(bloque);
        bloque.marcarOcupado();
}
return direccionDeComienzo;
}
```

Ejercicio 8.b

```
public void liberarBloque(int direccionDeComienzo){
BloqueDeMemoria bloque =
    avlOrdenadoPorDireccion.buscar(direccionDeComienzo);
bloque.marcarLibre();
}
```

Ejercicio 8.b

¿Los dos métodos presentados, aseguran que las operaciones se resuelvan en O(Log n)?

Ejercicio 8.b

¿Qué sucede si se libera un bloque que esta entre medio de dos bloques ocupados en la cola de bloques de igual tamaño? ¿El método pedir bloque sigue funcionando?

Ejercicio 8.b

El método pedir bloque, necesita que los bloques estén ordenados primero los libres y luego los ocupados, por lo cual no funcionaría. ¿Cómo lo corregimos?

Ejercicio 8.b

Propuesta: que el bloque de memoria tenga en realidad enlaces al bloque previo y siguiente, de forma que la cosa pasa a ser una lista doblemente enlazada, por lo cual, se puede ubicar directamente el bloque y hacer los ajustes necesarios.