Estructuras de Datos

Examen 2.ª Convocatoria Ordinaria
Curso 2019-2020

Tenemos una secuencia de *n* objetos cada uno de ellos con cierto peso (w_1, w_2, ..., w_n) y disponemos de un número ilimitado de cubos (todos de la misma capacidad, *W*), cada uno de los cuales puede albergar uno o más objetos siempre que la suma de sus pesos no supere su capacidad (*W*). Supondremos, además, que el peso de cada objeto no es superior a *W* ($w_i \leq W$, $1 \leq i \leq n$). La **capacidad restante** de un cubo será *W* menos la suma de los pesos de los objetos que contiene.

El problema del *Bin Packing* consiste en ir rellenando cubos con los objetos de forma que, para cualquier cubo, la suma de los pesos de los objetos que contenga no sobrepase la capacidad del cubo (*W*). El objetivo es colocar todos los objetos en cubos, minimizando el número de cubos utilizados.

Un algoritmo heurístico (es decir, que no garantiza la solución óptima, pero suele obtener buenas soluciones en la práctica) para resolver este problema es *First Fit*, que se describe de la siguiente forma:

- Se mantiene una secuencia de cubos (inicialmente vacía).
- Se van colocando los objetos, uno a uno, de la siguiente forma:
  - Para cada objeto a colocar, se recorre la secuencia de cubos de izquierda a derecha y se ubica el objeto en el **primer** cubo que tenga capacidad restante suficiente.
  - Si no hay ningún cubo que permita ubicar el objeto, se añade un nuevo cubo al **final** de la secuencia y se ubica el objeto en él.

La implementación inmediata del heurístico (que llamaremos *implementación lineal*) consiste en usar una estructura de datos lineal que represente una secuencia de cubos y buscar, de izquierda a derecha, el primer cubo donde se pueda insertar cada objeto, o añadir un cubo al final de la secuencia si ninguno puede albergar el nuevo objeto.

Representaremos los cubos y su contenido por la siguiente clase Java:

```
class Bin {
    private int remainingCapacity;
    private List<Integer> weights;
}
```

donde **`remainingCapacity`** representa la capacidad restante del cubo y **`weights`** es una lista con los pesos de los objetos incluidos en el cubo.

El problema de la *implementación lineal* es que su complejidad es $O(n^2)$. Una implementación más eficiente usaría un árbol balanceado AVL de cubos para representar la secuencia, con sus nodos ordenados según la posición en la secuencia de los cubos (es decir, si recorremos el árbol en en-orden, obtendríamos la secuencia de cubos como si la recorriéramos de izquierda a derecha).
Dado un árbol AVL, definimos su **capacidad restante máxima** como la capacidad restante del cubo del árbol con mayor capacidad restante.

Cada nodo del árbol AVL almacenará:

- un cubo (**`Bin`**)
- la altura del nodo en el árbol (**`int`**)
- la capacidad restante máxima del AVL del que es raíz (**`int`**)

- los hijos izquierdo y derecho (`AVL`)

Recordemos que en un árbol AVL, la diferencia entre las alturas del hijo izquierdo y derecho de un nodo no puede ser nunca superior a una unidad y que, si dicha condición se viola, es necesario restaurarla realizando rotaciones. Para este problema, el árbol AVL se puede equilibrar utilizando solo rotaciones simples a la izquierda.
Dado que la altura del árbol AVL es O(log n), es posible tanto encontrar el **primer** cubo de la secuencia donde ubicar el objeto como añadir un nuevo cubo al **final** en O(log n), con lo que la implementación del heurístico sería ahora O(n log n).

Sea la siguiente clase Java para representar un árbol AVL tal como se ha descrito:

```
public class AVL {
    static private class Node {
        Bin bin;
        int height;
        int maxRemainingCapacity;
        Node left, right;
    }
    private Node root;
}
```

La clase `Node` representa un nodo del AVL, donde `bin` es el cubo almacenado en el nodo, `height` es la altura del nodo en el árbol, `maxRemainingCapacity` es la capacidad restante máxima del árbol, `left` y `right` son los hijos izquierdo y derecho del nodo. El atributo `root` representa la raíz del AVL.

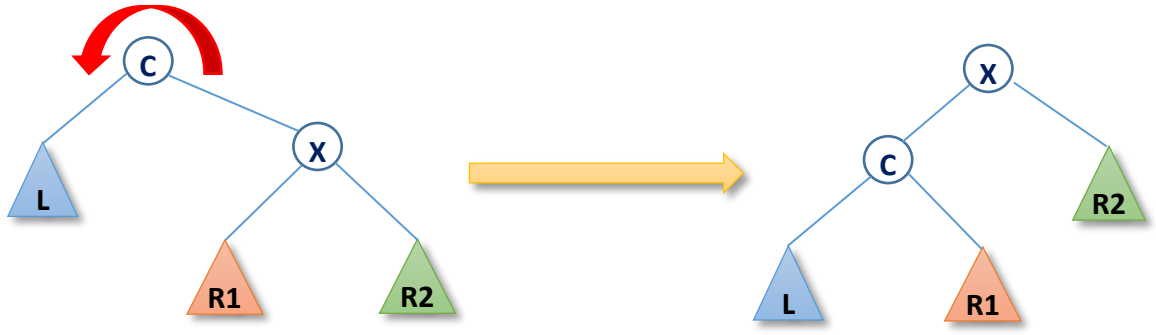
a) (0,5 puntos) Define los siguientes métodos de la clase `Bin`:
- constructor `Bin(int initialCapacity)` que dada una capacidad inicial construye un cubo vacío con tal capacidad
- `int remainingCapacity()` que devuelve la capacidad restante del cubo
- `void addObject(int weight)` que dado el peso de un objeto lo añade al cubo; si el objeto no cabe en el cubo eleva una excepción

Define los siguientes métodos privados estáticos de la clase `AVL`:
- `int maxRemainingCapacity(Node n)` que dado un nodo devuelve su capacidad restante máxima
- `int height(Node n)` que dado un nodo devuelve su altura

b) (0,5 puntos) Define el método `void setHeight()` de la clase `Node` que ajusta la altura del nodo

c) (0,5 puntos) Define el método `void setMaxRemainingCapacity()` de la clase `Node` que ajusta la capacidad restante máxima del nodo

d) (1 punto) Define el método `Node rotateLeft()` de la clase `Node` que aplica una rotación simple a la izquierda y devuelve el nodo raíz del árbol rotado, según se muestra en la figura de la siguiente página (puedes suponer que el hijo derecho del nodo no está vacío; se muestran su raíz, *x*, y sus hijos *R1* y *R2*).



e) (1,5 puntos) Define el método `void addNewBin(Bin bin)` de la clase `AVL` que toma un cubo y lo añade al final de la espina derecha del AVL. Para mantener el invariante de los árboles AVL, en cada nodo de la espina derecha, si la altura resultante del hijo derecho es más de una unidad superior a la del hijo izquierdo, habrá que aplicar una rotación simple a la izquierda.

f) (2.0 puntos) Define el método `void addFirst(int initialCapacity, int weight)` de la clase `AVL` que tome la capacidad de los cubos del problema (*W*), el peso de un objeto a añadir y que añada dicho objeto al primer cubo que pueda albergarlo o añada un nuevo cubo al final de la espina derecha si el nuevo objeto no cabe en ningún cubo. El algoritmo será el siguiente:

- Si el AVL está vacío o no cabe en ningún cubo, se añadirá un nuevo nodo con un cubo con el objeto al final de la espina derecha.
- En otro caso, si la capacidad restante máxima del hijo izquierdo es mayor o igual al peso del objeto, se añadirá el objeto al primer cubo posible del hijo izquierdo.
- En otro caso, si la capacidad restante del cubo en el nodo raíz es mayor o igual al peso del objeto, se añadirá el objeto al cubo en la raíz.
- En otro caso, se añadirá el objeto al primer cubo posible del hijo derecho.

g) (0,75 puntos) Define el método `void addAll(int initialCapacity, int[] weights)` de la clase `AVL` que tome el valor de la capacidad máxima de los cubos (*W*) y un *array* con los pesos de los objetos, y que añada al AVL todos los objetos del *array*, según se ha descrito anteriormente.

h) (0,75 puntos) Define el método `List<Bin> toList()` de la clase `AVL` que devuelva una lista de cubos con su recorrido en en-orden.

SOLO PARA ALUMNOS QUE RENUNCIAN A LA EVALUACIÓN CONTINUA

a) (1,25 puntos) Define el método `List<Bin> linearBinPacking(int initialCapacity, List<Integer> weights)` de la clase `LinearBinPacking` para resolver el problema usando la *implementación lineal* del algoritmo descrita al principio del enunciado. Este método toma como argumentos la capacidad máxima (*W*) común a todos los cubos y una lista con los pesos de los objetos. Devolverá una secuencia de cubos con los objetos incluidos tras aplicar la implementación lineal. Para representar la secuencia de cubos se usará una lista.

b) (0,5 puntos) Define el método `Iterable<Integer> objects()` de la clase `Bin` que devuelve un iterable con los pesos de los objetos contenidos en el cubo.

c) (0,75 puntos) Define el método `Iterable<Integer> allWeights(Iterable<Bin> bins)` de la clase `LinearBinPacking` que tome un iterable de cubos y devuelva un iterable de pesos. Al iterar el resultado, deberá obtenerse, en primer lugar, el primer peso incluido en el primer cubo

del argumento; a continuación, su segundo peso y así hasta que se obtengan todos los pesos del primer cubo. Luego, se obtendrán de la misma forma los pesos de los restantes cubos. La iteración estará completa cuando se hayan obtenido todos los pesos de todos los cubos.