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.

Sean los siguientes tipos en Haskell para representar cubos y su contenido:

```
type Capacity = Int
type Weight = Int
data Bin = B Capacity [Weight]
```

Asumimos que capacidades y pesos son valores enteros. El tipo **`Bin`** representa un cubo: el primer argumento representa la capacidad restante del cubo y el segundo será 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 (**`Capacity`**)

- 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 el siguiente tipo Haskell para representar un árbol AVL tal como se ha descrito:

```
data AVL = Empty | Node Bin Int Capacity AVL AVL deriving Show
```

donde:

- **`Empty`** representa un AVL vacío
- **`Node`** representa un nodo donde el valor de tipo **`Bin`** es el cubo almacenado en el nodo, el valor de tipo **`Int`** es la altura del nodo en el árbol, el valor de tipo **`Capacity`** es la capacidad restante máxima del árbol y los valores de tipo **`AVL`** son los hijos izquierdo y derecho del nodo.

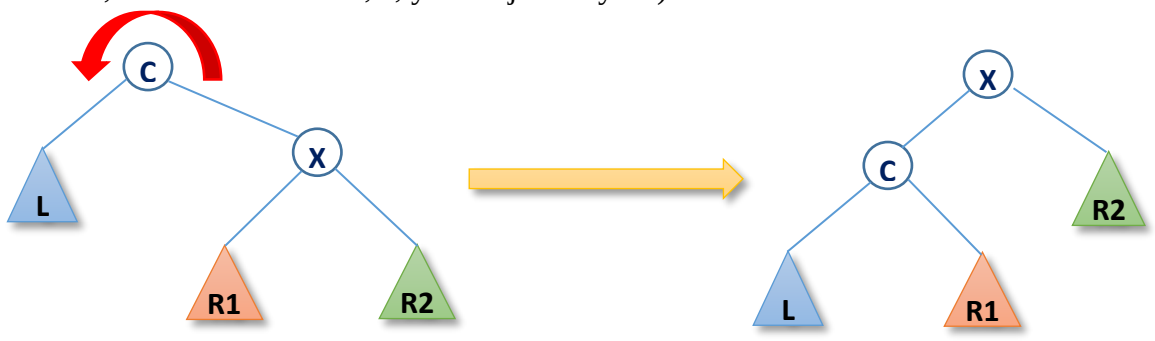
a) (0,5 puntos) Define las siguientes funciones auxiliares:

- **`emptyBin`** que dada una capacidad devuelve un cubo vacío con tal capacidad
- **`remainingCapacity`** que dado un cubo devuelve su capacidad restante
- **`addObject`** que dados un cubo y un objeto lo añade al cubo; si el objeto no cabe en el cubo señala un error
- **`maxRemainingCapacity`** que dado un AVL devuelva su capacidad restante máxima
- **`height`** que dado un AVL devuelve su altura

b) (0,5 puntos) Define una función **`nodeWithHeight`** que tome un cubo, una altura, un AVL (hijo izquierdo) y otro AVL (hijo derecho) y que devuelva un AVL con dicha información, **calculando su capacidad restante máxima**.

c) (0,5 puntos) Define una función **`node`** que tome un cubo, un AVL (hijo izquierdo) y otro AVL (hijo derecho) y que devuelva un AVL con dicha información, **calculando su capacidad restante máxima y la correspondiente altura.**

d) (1 punto) Define una función **`rotateLeft`** que tome un cubo *c*, un AVL (hijo izquierdo) *L* y otro AVL no vacío (hijo derecho) *R* y que devuelva un AVL creado con dicha información tras aplicar, además, una rotación simple a la izquierda tal como se ve en la siguiente figura (dado que el árbol *R* no está vacío, se muestran su raíz, *x*, y sus hijos *R1* y *R2*):



e) (1,5 puntos) Define una función **`addNewBin`** que tome un cubo y un árbol AVL y que añada un nuevo nodo con dicho cubo 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 una función **`addFirst`** que tome la capacidad de los cubos del problema (*W*), el peso de un objeto a añadir y un AVL 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 una función **`addAll`** que tome el valor de la capacidad máxima de los cubos (*W*), una lista de pesos de objetos y que construya un AVL con cubos que contenga todos los objetos de la lista, según se ha descrito anteriormente.

h) (0,75 puntos) Define una función **`toList`** que tome un AVL y devuelva una lista de cubos con su recorrido en en-orden.

SOLO PARA ALUMNOS QUE RENUNCIAN A LA EVALUACIÓN CONTINUA

a) (1,25 puntos) Escribe una función **`linearBinPacking`** para resolver el problema usando la *implementación lineal* del algoritmo descrita al principio del enunciado. La función tomará 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 el siguiente tipo Haskell:

```
data Sequence = SEmpty | SNode Bin Sequence deriving Show
```

b) (0,5 puntos) Define una función **`seqToList`** que tome una **`Sequence`** y devuelva una lista de cubos.

c) (0,75 puntos) Define una función **`addAllFold`** que resuelva el apartado *g* (función **`addAll`**) utilizando un plegado.