Parcial – Estructuras de Datos – UNQ

Aclaraciones:

- *Esta evaluación es a libro abierto. Se pueden usar todas las funciones y tipos de datos vistos en la práctica y en la teórica, salvo que el enunciado indique lo contrario.*
- *No se olvide de poner nombre, nro. de alumno, nro. de hoja y cantidad total de hojas en cada una de las hojas.*
- *Le recomendamos leer el enunciado en su totalidad y organizar sus ideas antes de comenzar la resolución.*
- *Recuerde que la intención es medir cuánto comprende usted del tema. Por ello, no dude en escribir todo lo que sabe, en explicar lo que se propone antes de escribir código, en probar sus funciones con ejemplos, etc.*

Un disco duro consiste de una serie de bloques de tamaño fijo, en donde se almacena información. Por simplicidad representamos un bloque únicamente con un número o *id*:

```
type Block = Int
```

Para facilitar la manipulación de grandes volumenes de bloques, el disco trabajará internamente con segmentos de bloques libres. Un segmento es, entonces, un grupo de bloques contiguos. Para trabajar con segmentos se define el tipo abstracto `Segment` definido con la siguiente interfaz:

- `new :: Block -> Int -> Segment`: crea un segmento que se inicia en el bloque `b` y tiene tamaño `n` (o sea, contiene `n` bloques). PRECONDICIÓN: `b` $\geq$ `0` y `n` $>$ `0`. $O(1)$
- `use :: Int -> Segment -> Segment`: retorna un nuevo segmento que resulta de usar los primeros `n` bloques de `s` (o sea, un segmento que tiene menos bloques que el original). PRECONDICIÓN: `n` $<$ `size s`. $O(1)$
- `base :: Segment -> Block`: dado un segmento retorna su bloque de inicio. $O(1)$
- `size :: Segment -> Int`: dado un segmento retorna su tamaño (cantidad de bloques que lo componen). $O(1)$
- `higher :: Segment -> Segment -> Bool`: decide si el 1er segmento se inicia en un bloque con id más alto que el 2do. $O(1)$
- `bigger :: Segment -> Segment -> Bool`: decide si el 1er segmento contiene más bloques que el 2do. $O(1)$
- `toBlocks :: Segment -> [Block]`: dado un segmento `s` retorna la lista de bloques que lo componen. $O(s)$

Los bloques que conforman un segmento son contiguos, de modo que:

```
toBlocks (new 1 10) = [1,2,3,4,5,6,7,8,9,10]
```

El tipo abstracto que representa discos, `Disc`, tiene la siguiente interfaz:

- `nuevo :: Int -> Disc`: genera un disco vacío de tamaño `n`. $O(1)$
- `tamaño :: Disc -> Int`: dado un disco retorna su tamaño. $O(1)$
- `libre :: Block -> Disc -> Bool`: decide si el bloque `b` está libre en el disco `d`. $O(log(d))$
- `ocupado :: Block -> Disc -> Bool`: decide si el bloque `b` está ocupado en el disco `d`. $O(log(d))$
- `liberar :: [Block] -> Disc -> Disc`: retorna el disco que resulta de liberar todos los bloques de la lista `bs` en `d`. PRECONDICIÓN: los bloques de `bs` están ocupados en `d`.
- `ocupar :: Int -> Disc -> ([Block], Disc)`: dados `n` y `d`, retorna un par `(bs,d')` donde `d'` es el disco que resulta de ocupar `n` bloques en `d`, y `bs` es la lista de bloques usados. PRECONDICIÓN: hay el menos `n` bloques libres en `d`. $O(log(d))$
- `espacioLibre :: Disc -> Int`: dado un disco `d` calcula el espacio libre disponible. $O(d)$

Para lograr una mejor eficiencia, al momento de implementar el TAD `Disc`, representamos el disco con dos AVL's de segmentos donde se lleva cuenta de los bloques disponibles y el tamaño del mismo (para poder consultarlo en orden constante).

```
data Disc = D Int (Tree Segment) (Tree Segment)
```

El primero de los árboles contendrá los segmentos disponibles ordenados por bloque de inicio (comparando por `higher`), para poder localizar fácilmente un bloque en el disco; mientras que el segundo los ordenará por tamaño (usando `bigger`) para facilitar la asignación de espacio para archivos. Esto fuerza el uso de ciertos invariantes de representación en el tipo `Disc`, para mantener la consistencia de la estructura. Dado `D size bs ts`:

- `bs` y `ts` son AVL's y contienen los mismos elementos.

- `bs` está ordenado por bloque de inicio de segmento.
- `ts` está ordenado por tamaño de segmento.
- Los segmentos de `bs` (y `ts`) entran en el disco (es decir `init s + size s` $\leq$ `size` para cada `s` de `bs`).
- No hay segmentos solapados (que compartan bloques, por ejemplo `[1,2,3,4]` y `[3,4,5,6]`).
- No hay segmentos contiguos (que comienzen uno a continuación del otro, por ejemplo `[1,2,3]` y `[4,5,6]` deberían ser un único segmento `[1,2,3,4,5,6]`).

Pueden considerar ya definidas todas las funciones vistas en clase, así como las siguientes funciones:

- `agregarPorBloque :: Segment -> Tree Segment -> Tree Segment` y
  `agregarPorTamaño :: Segment -> Tree Segment -> Tree Segment`:
  que agregan el segmento `s` en el AVL `t` comparando por `higher` y `bigger` respectivamente. $O(log(t))$
- `borrarPorBloque :: Segment -> Tree Segment -> Tree Segment` y
  `borrarPorTamaño :: Segment -> Tree Segment -> Tree Segment`:
  que eliminan el segmento `s` del AVL `t` comparando por `higher` y `bigger` respectivamente. $O(log(t))$

ATENCIÓN: solo deben implementarse las funciones pedidas en los ejercicios (por ejemplo, `liberar` ¡NO DEBE ser definida!)

Ejercicio 1 Implementar las funciones `nuevo`, `tamaño`, `libre`, `ocupado` y `espacioLibre` de la interfaz del tipo `Disc` con sus complejidades correspondientes.

AYUDA: para `libre` implementar una funcion auxiliar `inSegment :: Block -> Segment -> Bool` que decida si el bloque pertenece al segmento. ¿Qué complejidad debe tener para garantizar el orden $O(log(d))$ pedido en `libre`?

Para la implementación de `ocupar` tenemos el requisito adicional de que, al asignar espacio para un nuevo archivo, éste debe quedar lo menos fragmentado posible. Es decir, si hay algún segmento que pueda contener el archivo entero, deben usarse los bloques de ese segmento prioritariamente. Además, cuanto más ajustada es esa elección, mejor, para evitar fragmentaciones innecesarias en futuros archivos. Si el archivo debe ser fragmentado obligatoriamente, se busca usar la menor cantidad de segmentos posibles.

Ejercicio 2 Implementar, con el orden de complejidad pedido, las siguientes funciones auxiliares que trabajan directamente sobre árboles AVL:

a) `entraSinFragmentar :: Int -> Tree Segment -> Bool`: que dado número `n` decide si hay al menos un segmento que contenga esa cantidad de bloques en el AVL `t`. $O(log(t))$

b) `buscarSegmentoMasGrande :: Int -> Tree Segment -> (Segment, Maybe Segment)`: que dado un número `n` y un AVL `t`, devuelve el par `(s, ms)` donde `s` es primer segmento de `t` con al menos `n` bloques (idealmente el más chico). Si el segmento encontrado contiene más de `n` bloques, `ms` contiene el segmento con los bloques restantes de `s`. PRECONDICIÓN: hay al menos un segmento con `n` bloques en `t`. $O(log(t))$

c) `buscarSegmentos :: Int -> Tree Segment -> ([Segment], Maybe Segment)`: que dado un número `n` y un AVL `t`, devuelve el par `(ss, ms)` donde `ss` es la lista de segmentos reservados y `ms` contiene el (posible) remanente del último segmento usado. PRECONDICIÓN: hay al menos `n` bloques libres en `t`. $O(log(t))$

Finalmente, se propone una función `ocuparRep` como sigue:

```
ocuparRep :: Int -> Tree Segment -> Tree Segment -> ([Segment], Maybe Segment) -> ([Block], Disc)
ocuparRep n bs ts p = (segsToBlocks (fst p), actualizarDisco n bs ts p)
```

donde `segsToBlocks :: [Segment] -> [Block]` es la extensión a listas de `toBlocks` del módulo `Segment`.

Ejercicio 3 Implementar las siguientes funciones que completan la definición de `ocupar`, planteando las precondiciones correspondientes y asegurando que las estructuras resultantes cumplan los invariantes de represetación debidos:

a) `actualizarPorBloque :: [Segment] -> Maybe Segment -> Tree Segment -> Tree Segment` y
`actualizarPorTamaño :: [Segment] -> Maybe Segment -> Tree Segment -> Tree Segment`, $O(log(t))$.

b) `actualizarDisco :: Int -> Tree Segment -> Tree Segment -> ([Segment], Maybe Segment) -> Disc`, $O(log(t))$.

c) `ocupar :: Int -> Disc -> ([Block], Disc)`, de la interfaz. AYUDA: use la función `ocuparRep`.

Puede ser que le sean útiles las siguientes funciones, que deberá definir en caso de usarlas:

- `borrarSegsPorBloque :: [Segment] -> Tree Segment -> Tree Segment` y
  `borrarSegsPorTamaño :: [Segment] -> Tree Segment -> Tree Segment`:
  las extensiones para operar sobre listas de `borrarPorBloque` y `borrarPorTamaño` respectivamente.