Práctica TAD e Inducción (Complementaria)

1. Un Array es una estructura de datos indexada, que tiene una dimensión fija y permite agregar elementos en posiciones aleatorias dentro del rango de indices. Por simplicidad se considera que los indices comienzan en 1. Las siguientes operaciones son soportadas por este TAD:

tad Array (A : Set) **where**
  import Int, Maybe
  ini : Int → Array A -- dado un entero retorna un vector de esa dimensión
  insert : Int → A → Array A → Array A -- inserta un elemento en la posición dada
  view : Int → Array A → Maybe A -- muestra un elemento en una posición dada
  size : Array A → Int -- muestra la dimensión del Vector

a) Dar las especificación algebraica del TAD Array.

b) Implementar en haskell el TAD Array, usando **class** e **instance** sobre el tipo lista.

2. Un multiconjunto o Bag difiere de un conjunto en que cada elemento del mismo puede ocurrir más de una vez. Por ejemplo, en el multiconjunto $\{1, 1, 2, 3, 3, 3\}$ el elemento 1 ocurre dos veces, el 2 una vez y el 3 ocurre tres veces. Las siguientes operaciones son implementadas por el *tad*:

tad Bag (A : Set) **where**
  import Bool
    vacio : Bag A
    insertar : A → Bag A → Bag A
    borrar : A → Bag A → Bag A -- borra una aparición del elemento A en el Bag
    union : Bag A → Bag A → Bag A
    pertenece : A → Bag A → Bool
    interseccion : Bag A → Bag A → Bag A
    esVacio : Bag A → Bool
    size : Bag A → Int
    resta : Bag A → Bag A → Bag A

a) Dar las especificación algebraica del *tad* Bag.

b) Suponiendo que en la implementación del *tad* se utiliza el tipo:

   type Bag $a = [(a, \mathsf{Int})]$

   para representar al tipo Bag a, donde a representa el elemneto y el entero la cantidad de veces que aparece en el multiconjunto. Ejemplo el Bag $\{a, b, b\}$ se implementa como $[(a, 1), (b, 2)]$), Implementar las operaciones *insertar*, *borrar* y *union*.

3. El TAD Mapa es una colección de pares (clave, valor), donde las claves son únicas pero los valores pueden repetirse. Ejemplos de Mapa son: $\{(1, a), (2, b), (7, a), (9, c)\}$, $\{(1, 3), (3, 3), (5, 6), (6, 6)\}$. Las siguientes operaciones son soportadas por este TAD:

tad Mapa (K : Ordered Set, V : Set)
  import List, Bool, *int*

$$
\begin{array}{ll}
\textit{empty} & : \mathsf{Mapa\ K\ V} \\
\textit{isKey} & : \mathsf{Mapa\ K\ V} \to \mathsf{K} \to \mathsf{Bool} \\
\textit{put} & : \mathsf{Mapa\ K\ V} \to (\mathsf{K}, \mathsf{V}) \to \mathsf{Mapa\ K\ V} \\
\textit{size} & : \mathsf{Mapa\ K\ V} \to \textit{int} \\
\textit{delete} & : \mathsf{Mapa\ K\ V} \to (\mathsf{K}, \mathsf{V}) \to \mathsf{Mapa\ K\ V} \\
\textit{minKey} & : \mathsf{Mapa\ K\ V} \to \mathsf{Maybe\ K} \\
\textit{asList} & : \mathsf{Mapa\ K\ V} \to \mathsf{List}\ (\mathsf{K}, \mathsf{V})
\end{array}
$$

Donde:

- *empty* crea un Mapa vacío
- *isKey* comprueba si una clave k esta en el Mapa
- *put* agrega un par (k, v) de un Mapa, si es posible
- *size* retorna la cantidad de pares (k, v) presentes en un Mapa
- *delete* elimina un par (k, v) de un Mapa
- *minKey* retorna la menor clave k de un Mapa, si el Mapa es *empty* retorna Nothing
- *asList* devuelve una lista **ORDENADA** por clave de todos los pares de un Map

ya se sabe que:

$$
\begin{array}{rl}
\textit{size empty} = & 0 \\
\textit{size}\ (\textit{put}\ m\ (a, b)) = & \textbf{if}\ \textit{isKey}\ m\ a\ \textbf{then}\ \textit{size}\ m\ \textbf{else}\ 1 + \textit{size}\ m \\
\textit{delete empty}\ (a, b) = & \textit{empty} \\
\textit{delete}\ (\textit{put}\ m\ (c, d))\ (a, b) = & \textbf{if}\ \textit{isKey}\ m\ c\ \textbf{then}\ \textit{delete}\ m\ (a, b) \\
 & \textbf{else if}\ a \equiv c \land d \equiv b\ \textbf{then}\ m\ \textbf{else}\ \textit{put}\ (\textit{delete}\ m\ (a, b))\ (c, b) \\
\vdots &
\end{array}
$$

Completar la especificación algebraica del TAD Mapa. Recordar que los constructores de listas son *cons* y *nil*.

4. El inventario de un juego permite que un jugador recolecte diferentes objetos para intercambiarlos dentro del juego. El tad Inv es una colección de tamaño limitado que almacena pares (ID, Cantidad) donde los ID son únicos y Cantidad es el número de objetos recolectados de tipo ID. El tamaño de un inventario se determina al crearse este y es la cantidad de pares (ID, Cantidad) que puede almacenar. Este *tad* soporta las siguientes operaciones:

$$
\begin{array}{l}
\textit{tad}\ \mathsf{Inv}\ (\mathsf{I} : \mathsf{Ordered\ Set}) \\
\quad \textbf{import}\ \mathsf{List}, \mathsf{Bool}, \mathsf{Nat} \\
\qquad \begin{array}{ll}
\textit{new} & : \mathsf{Nat} \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \\
\textit{collect} & : \mathsf{I} \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \\
\textit{del} & : \mathsf{I} \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \\
\textit{countI} & : \mathsf{I} \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \to \mathsf{Maybe\ Nat} \\
\textit{size} & : \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat}) \to \mathsf{Nat} \\
\textit{fromList} & : \mathsf{List\ I} \to \mathsf{Inv}\ (\mathsf{I}, \mathsf{Nat})
\end{array}
\end{array}
$$

donde

- *new* crea un inventario vacío de tamaño n.

- *collect* toma un id e incrementa en uno la cantidad asociada a este, si el id no está en el inventario lo agrega en la siguiente posición vacía si existe una, de lo contrario lo descarta.
- *del* toma un id y decrementa en uno su cantidad asociada.
- *countI* toma un id y un inventario retorna la *cantidad* asociada al id
- *size* dado un inventario nos dice que tamaño tiene.
- *fromList* dada una lista *ls* que contiene id (posiblemente repetidos), crea un inventario donde entren todos los elementos que están en la lista.
  Ejemplo: sea $<\texttt{'a'},\texttt{'b'},\texttt{'a'},\texttt{'v'},\texttt{'c'},\texttt{'b'}>$ la listas de *ids*, el inventario generado por *fromList* debería ser $\{(\texttt{'a'},2),(\texttt{'b'},2),(\texttt{'v'},1),(\texttt{'c'},1)\}$

Especifique el *tad* inventario.

5. Para implementar las operaciones del *tad* Inv se utilizará el tipo:

```
data Inv a = H | N (Inv a) (Maybe (a, Int)) (Inv a)
```

donde el tamaño del inventario queda definido por:

```
size :: Inv a → Int
size H = 0
size (N l j r) = 1 + size l + size r
```

y los elementos se completan según el recorrido *inorder* del inventario.
Implementar las operaciones *new* y *collect* del *tad* en haskell.

6. Considera el TAD de DList que especifica listas finitas con los constructores *empty*, *single* y *join*

```
tad DList (A : OrdSet) where
  import Maybe, List
  empty    : DList A                        -- lista vacía
  single   : A → DList A                    -- lista unitaria
  join     : DList A → DList A → DList A    -- join ls rs es la lista donde ls es prefijo de rs
  dhead    : DList A → Maybe A              -- devuelve la cabeza de la lista si es posible
  dtail    : DList A → Maybe (DList A)      -- devuelve la cola de la lista si es posible
  dreverse : DList A → DList A              -- devuelve la lista invertida
  toList   : DList A → List A               -- convierte una DList a un List
```

Dar la especificación algebraica del *tad* DList sin definir funciones auxiliares. Considerar definidos para el *tad* List, importado por DList, los constructores usuales *nil* y *cons*.
El constructor *join ls rs* tomas dos listas y retorna una nueva lista donde los elementos de *ls* están antes que los de *rs* en la lista:
Ejemplo *join* (*join* (*single* 'A') (*single* 'B')) (*single* 'C') representa la lista *"ABC"*.
Notar que una misma lista puede tener varias representaciones distintas.

7. Dado el *tad* Matriz con las siguientes operaciones:

```
tad Matriz (A : Set)
  import List, Bool, Nat, Maybe
    new : Nat → Nat → Matriz A   -- new i j : Crea una matriz con i filas y j columnas vacía
    set : Nat → Nat → A → Matriz A → Matriz A   -- set i j x m : inserta x en la fila i columna j de m
```

get : Nat $\to$ Nat $\to$ Matriz A $\to$ Maybe A -- obtiene el valor almacenado en un indice de la matriz
$size$: Matriz A $\to$ (Nat, Nat) -- size m : retorna un par con las dimensiones de la matriz

Especifique el *tad* Matriz, considere definidas las operaciones usuales para los *tad* importados.

8. Para implementar en haskell las operaciones del *tad* Matriz se utilizarán los tipos:

type Columna a = [Maybe a]
type Matriz a = [Columnas a]

donde una matriz queda definida por sus columnas. Implementar las funciones *new*, *set*, *get* y *size* especificadas en el *tad* Matriz

9. Dadas las siguientes definiciones:

$map :: (a \to b) \to [a] \to [b]$
$map\ f\ [\,] = [\,]$
$map\ f\ (x : xs) = f\ x : map\ f\ xs$

$[\,] +\!\!+\ ys = ys$
$xs +\!\!+ [\,] = xs$
$(x : xs) +\!\!+\ ys = x : (xs +\!\!+\ ys)$

a) Enunciar el principio de inducción para listas.

b) Probar por inducción estructural que:

i) $map\ (f \circ g) = map\ f \circ map\ g$

ii) $map\ f\ (xs +\!\!+\ ys) = map\ f\ xs +\!\!+\ map\ f\ ys$

10. Dadas las siguientes definiciones:

$concat\ [\,] = [\,]$
$concat\ (xs : xss) = xs +\!\!+\ concat\ xss$

$sum\ [\,] = 0$
$sum\ (x : xs) = x + sum\ xs$

Se quiere demostrar $(sum \circ concat)\ r = (sum \circ (map\ sum))\ r$ para esto:

a) Enunciar el principio de inducción para el tipo que tiene r.

b) Probar por inducción estructural que: $(sum \circ concat) = (sum \circ (map\ sum))$

11. Dado el siguiente tipo de dato:

data Bin a = E | N (Bin a) a (Bin a)

Probar por inducción estructural la siguiente propiedad:

$node = (sum \circ flatOne)$

y todos los lemas necesarios para su demostración. Donde

$flatOne$ E $= [\,]$
$flatOne$ (N $l\ x\ r$) $= 1 : (flatOne\ l +\!\!+\ flatOne\ r)$
$node$ E $= 0$

$node$ (N $l\ x\ r$) $= 1 + node\ l + node\ r$
$sum\ [\,] = 0$
$sum\ (x : xs) = x + sum\ xs$

12. Dado el siguiente tipo de datos para representar árboles generales:

data GTree a = N a [GTree a]

y las siguientes definiciones:

$$
\begin{aligned}
&elem\ x\ [\,] = \mathsf{False}\\
&elem\ x\ (y : ys) = x \equiv y\ ||\ elem\ x\ ys\\
&or\ [\,] = \mathsf{False}\\
&or\ (x : xs) = x\ ||\ or\ xs
\end{aligned}
$$

$$
\begin{aligned}
&concatMap\ f = concat \circ map\ f\\
&allNodes\ (\mathsf{N}\ x\ xs) = x : concatMap\ allNodes\ xs\\
&isNode\ y\ (\mathsf{N}\ x\ xs) = y \equiv x\ ||\ or\ (map\ (isNode\ y)\ xs)
\end{aligned}
$$

a) Enunciar el principio de inducción estructural para $\mathsf{GTree}\ a$

b) Probar por inducción estructural sobre t que $(elem\ n \circ allNodes)\ t = isNode\ n\ t$

Ayuda: Puede usar las siguientes propiedades dando su demostración:

a) Lema: $elem\ x\ (concat\ xss) = or\ (map\ (elem\ x)\ xss)$

b) Lema: $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$

13. Dadas las siguientes definiciones:

$$
\begin{aligned}
&atFirst\ x\ xss = [\,] : map\ (x{:})\ xss\\
&atLast\ [ys]\ x = [ys, ys +\!\!+ [x]]\\
&atLast\ ys : yss\ x = ys : atLast\ yss\ x
\end{aligned}
$$

enumerate

Determinar los tipos de las funciones $atFirst$ y $atLast$, siendo sus definiciones:

Demostrar que para toda lista yss y para todos elementos x e x', se cumple que:

$$atFirst\ x\ (atLast\ yss\ x') = atLast\ (atFirst\ x\ yss)\ x'$$