Práctica 4

1. Si un árbol binario es dado como un nodo con dos subárboles idénticos se puede aplicar la técnica *sharing* para que los subárboles sean representados por el mismo árbol. Definir las siguientes funciones de manera que se puedan compartir la mayor cantidad posible de elementos de los árboles creados:

a) completo :: $a \to$ Int $\to$ Tree a, tal que dado un valor x de tipo a y un entero d, crea un árbol binario completo de altura d con el valor x en cada nodo.

b) balanceado :: $a \to$ Int $\to$ Tree a, tal que dado un valor x de tipo a y un entero n, crea un árbol binario balanceado de tamaño n, con el valor x en cada nodo.

2. Definir las siguientes funciones sobre árboles binarios de búsqueda (bst):

1. maximum :: Ord $a \Rightarrow$ BST $a \to a$, que calcula el máximo valor en un *bst*.
2. checkBST :: Ord $a \Rightarrow$ BST $a \to$ Bool, que chequea si un árbol binario es un *bst*.
3. splitBST :: Ord $a \Rightarrow$ BST $a \to a \to$ (BST a, BST a), que dado un árbol *bst* t y un elemento x, devuelva una tupla con un *bst* con los elementos de t menores o iguales a x y un *bst* con los elementos de t mayores a x.
4. join :: Ord $a \Rightarrow$ BST $a \to$ BST $a \to$ BST a, que una los elementos dos árboles *bst* en uno.

3. La definición de member dada en teoría (la cual determina si un elemento está en un bst) realiza en el peor caso $2 * d$ comparaciones, donde d es la altura del árbol. Dar una definición de member que realice a lo sumo $d + 1$ comparaciones. Para ello definir member en términos de una función auxiliar que tenga como parámetro el elemento candidato, el cual puede ser igual al elemento que se desea buscar (por ejemplo, el último elemento para el cual la comparación de $a \leqslant b$ retornó True) y que chequee que los elementos son iguales sólo cuando llega a una hoja del árbol.

4. La función insert dada en teoría para insertar un elemento en un rbt puede optimizarse eliminando comparaciones innecesarias hechas por la función balance. Por ejemplo, en la definición de la función ins cuando se aplica balance sobre el resultado de aplicar insert x sobre el subárbol izquierdo (l) y el subárbol derecho (r), los casos de balance para testear que se viola el invariante 1 en el subárbol derecho no son necesarios dado que r es un rbt.

a) Definir dos funciones lbalance y rbalance que chequeen que el invariante 1 se cumple en los subárboles izquierdo y derecho respectivamente.

b) Reemplazar las llamadas a balance en ins por llamadas a alguna de estas dos funciones.

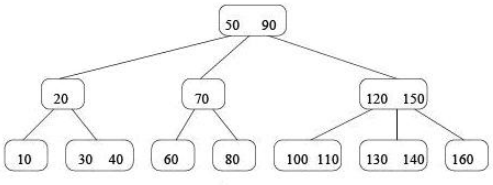
5. Los árboles 1-2-3 son árboles binarios de búsqueda donde los nodos pueden guardar múltiples valores y tener entre 2 y 4 hijos.
Específicamente, en un árbol 1-2-3 los nodos internos son de la forma:

2-node : Contienen un valor y dos hijos.

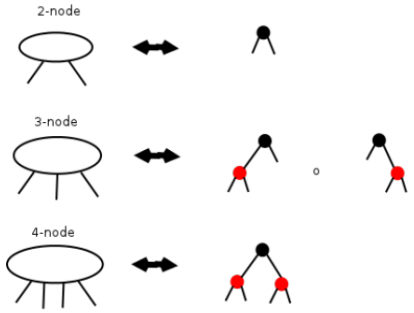
3-node : Contienen dos valores y tres hijos.

4-node : Contienen tres valores y cuatro hijos.

El siguiente es un ejemplo de un árbol 1-2-3:



Las siguientes correspondencias nos permiten representar un red-black tree como un árbol 1-2-3.



1. Definir un tipo de datos que represente árboles 1-2-3.
2. Definir una función que transforme red-black trees en árboles 1-2-3. Paralelizar cuando sea posible.

6. Definir una función fromList :: $[a] \to$ Heap a, que cree un *leftist heap* a partir de una lista, convirtiendo cada elemento de la lista en un *heap* de un solo elemento y aplicando la función merge hasta obtener un solo *heap*. Aplicar la función merge n veces, donde n es la longitud de la lista que recibe como argumento la función.

7. Un *pairing heap* es un árbol general que satisface el invariante de heap.
Para implementar *pairing heap* en Haskell definimos el siguiente tipo de datos:

data PHeaps a = Empty | Root a [PHeaps a]

Definir las siguientes funciones:

1. isPHeap :: Ord $a \Rightarrow$ PHeaps $a \to$ Bool, determina si un árbol es un *pairing heap*, es decir cumple con el invariante de *heap*.
2. merge :: Ord $a \Rightarrow$ PHeaps $a \to$ PHeaps $a \to$ PHeaps a, que una dos *pairing heap*. Para ello, comparar las raíces de ambos árboles y elegir la menor como raíz del nuevo *heap*, agregar el árbol con mayor raíz como hijo de éste.
3. insert :: Ord $a \Rightarrow$ PHeaps $a \to a \to$ PHeaps, que inserte un elemento en un *pairing heap*. Puede ser útil la función merge.
4. concatHeaps :: Ord $a \Rightarrow$ [PHeaps a] $\to$ PHeaps a, que dada una lista de *pairing heaps* construya otro con los elementos del mismo.
5. delMin :: Ord $a \Rightarrow$ PHeaps $a \to$ *Maybe* $(a,$ PHeaps $a)$, que dado un *pairing heap*, devuelva si el árbol no es vacío un par con el menor elemento y un *pairing heap* sin éste elemento, o *Nothing* en otro caso.