```
--Student's name: Juan Diaz-Flores Merino
--Student's group:
--Identity number (DNI if Spanish/passport if Erasmus):

module DataStructures.Set.TreeBitSet (
  TreeBitSet,
  empty,
  size,
  capacity,
  isEmpty,
  contains,
  add,
  toList

  ) where

import Test.QuickCheck
import qualified DataStructures.Set.IntBits as IntBits
import Data.List((\\))


data Tree = Leaf Int | Node Tree Tree deriving Show

data TreeBitSet = TBS Int Tree deriving Show

bitsPerLeaf :: Int
bitsPerLeaf = 64

-- returns true if capacity is 64 * 2^n for some n >= 0
isValidCapacity :: Int -> Bool
isValidCapacity capacity
  | capacity <= 0           = False
  | capacity == bitsPerLeaf = True
  | m == 0                  = isValidCapacity half
  | otherwise               = False
  where
    (half, m) = divMod capacity 2

-- ================================================================

-- | Función para crear un árbol con la capacidad dada
makeTree :: Int -> Tree
makeTree capacidad
  | capacidad <= bitsPerLeaf = Leaf 0    -- Si la capacidad es pequeña
o igual a bitsPerLeaf, crea una hoja con todos los bits en 0.
  | otherwise               = Node (makeTree mitad) (makeTree mitad)
-- Si la capacidad es mayor que bitsPerLeaf, crea un nodo con dos
subárboles, cada uno con la mitad de la capacidad.
  where
    mitad = div capacidad 2   -- Calcula la mitad de la capacidad para
las llamadas recursivas.
```

```
-- | Función para crear un TreeBitSet vacío con la capacidad dada
empty :: Int -> TreeBitSet
empty capacidad
  | capacidad < 0                    = error "capacity must be positive"
-- Asegura que la capacidad sea un valor positivo.
  | not (isValidCapacity capacidad) = error "capacity must be 64
multiplied by a power of 2"   -- Asegura que la capacidad sea una
potencia válida de 2 multiplicada por 64.
  | otherwise                       = TBS capacidad (makeTree capacidad)
-- Crea un TreeBitSet con la capacidad especificada y un árbol asociado
usando makeTree.


-- | Función para obtener el tamaño de un TreeBitSet
size :: TreeBitSet -> Int
size (TBS c t) = aux t
  where
    -- Función auxiliar para calcular el tamaño de un árbol
    aux (Leaf b) = IntBits.countOnes b  -- Si el nodo es una hoja,
devuelve la cantidad de unos en el conjunto de bits.
    aux (Node lt rt) = aux lt + aux rt  -- Si el nodo es un nodo
interno, suma los tamaños de los subárboles izquierdo y derecho.


-- | Función para obtener la capacidad de un TreeBitSet
capacity :: TreeBitSet -> Int
capacity (TBS c _) = c


-- | Función para verificar si un TreeBitSet está vacío
isEmpty :: TreeBitSet -> Bool
isEmpty tbs = (size tbs == 0)


-- | Función para verificar si un elemento está fuera del rango de un
TreeBitSet
outOfRange :: TreeBitSet -> Int -> Bool
outOfRange (TBS c _) x = x < 0 || x >= c


-- | Función para verificar si un elemento está contenido en un
TreeBitSet
contains :: Int -> TreeBitSet -> Bool
contains x tbs@(TBS capacidad arbol)
  | outOfRange tbs x = False   -- Si el elemento está fuera del rango
del conjunto, devuelve False.
  | otherwise   = aux arbol x capacidad
  where
```

```
    -- Función auxiliar para verificar la presencia de un elemento en
un árbol
    aux :: Tree -> Int -> Int -> Bool
    aux (Leaf bits) x capacidad = IntBits.contains bits x   -- Si el
nodo es una hoja, verifica si el bit correspondiente al elemento está
presente.
    aux (Node izquierdo derecho) x capacidad
      | x < mitad = aux izquierdo x mitad   -- Si el elemento es menor
que la mitad de la capacidad, busca en el subárbol izquierdo.
      | otherwise = aux derecho (x - mitad) mitad   -- Si el elemento
es mayor o igual a la mitad de la capacidad, busca en el subárbol
derecho.
      where
        mitad = div capacidad 2   -- Calcula la mitad de la capacidad
para determinar el rango de búsqueda.


-- | Función para agregar un elemento a un TreeBitSet
add :: Int -> TreeBitSet -> TreeBitSet
add x tbs@(TBS capacidad arbol)
  | outOfRange tbs x = error "element is out of range"  -- Si el
elemento está fuera del rango del conjunto, genera un error.
  | otherwise   = TBS capacidad (aux arbol x capacidad)
  where
    -- Función auxiliar para agregar un elemento a un árbol
    aux :: Tree -> Int -> Int -> Tree
    aux (Leaf bits) x capacidad = Leaf (IntBits.set bits x)  -- Si el
nodo es una hoja, establece el bit correspondiente al elemento.
    aux (Node izquierdo derecho) x capacidad
      | x < mitad = Node (aux izquierdo x mitad) derecho   -- Si el
elemento es menor que la mitad de la capacidad, agrega al subárbol
izquierdo.
      | otherwise = Node izquierdo (aux derecho (x - mitad) mitad)   --
Si el elemento es mayor o igual a la mitad de la capacidad, agrega al
subárbol derecho.
      where
        mitad = div capacidad 2   -- Calcula la mitad de la capacidad
para determinar el rango de inserción.



-- | Función para convertir un TreeBitSet a una lista de elementos
toList :: TreeBitSet -> [Int]
toList (TBS capacidad arbol) = aux arbol capacidad
  where
    -- Función auxiliar para realizar la conversión a lista
    aux :: Tree -> Int -> [Int]
    aux (Leaf bits) _ = filter (IntBits.contains bits)
[0..bitsPerLeaf-1]  -- Si el nodo es una hoja, filtra los elementos
presentes en el conjunto de bits.
```

```
-- ========================================

instance Arbitrary TreeBitSet where
    arbitrary = do
        exponent <- chooseInt (0, 2)
        let capacity = bitsPerLeaf * 2^exponent
        elements <- listOf (chooseInt (0, capacity-1))
        return (foldr add (empty capacity) elements)
            where
                chooseInt :: (Int, Int ) -> Gen Int
                chooseInt (x,y) = choose (x,y)

instance Eq TreeBitSet where
  tbs1 == tbs2 = toInts tbs1 == toInts tbs2
    where
      toInts (TBS _ t) = reverse . dropWhile (==0) . reverse $ toInts'
t
      toInts' (Leaf b) = [b]
      toInts' (Node lt rt) = toInts' lt ++ toInts' rt

-- ========================================


validExponent e = e >= 0 && e <= 5

validElement e (TBS c t) = e >= 0 && e < c

-- Axioms for basic set of operations
ax1 e = validExponent e ==> isEmpty emptySet
  where emptySet = empty (64*2^e)

ax2 x s = validElement x s ==> not (isEmpty (add x s))

ax3 e x = validExponent e ==> not (contains x emptySet)
  where emptySet = empty (64*2^e)

ax4 x y s = validElement y s ==> contains x (add y s) == (x==y) ||
contains x s

ax5 e = validExponent e ==> size emptySet == 0
  where emptySet = empty (64*2^e)

ax6 x s = contains x s  ==> size (add x s) == size s
ax7 x s = validElement x s && not (contains x s) ==> size (add x s) ==
1 + size s

ax11 e = validExponent e ==> null (toList emptySet)
  where emptySet = empty (64*2^e)
```

```
ax12 x s = contains x s  ==> toList (add x s) == toList s

ax13 x s = validElement x s && not (contains x s) ==> x `elem` xs &&
(xs \\ [x]) `sameElements` toList s
  where
    xs = toList (add x s)
    sameElements xs ys = null (xs \\ ys) && null (ys \\ xs)


type Elem = Int

setAxioms = do
  quickCheck (ax1 :: Int -> Property)
  quickCheck (ax2 :: Elem -> TreeBitSet -> Property)
  quickCheck (ax3 :: Int -> Elem -> Property)
  quickCheck (ax4 :: Elem -> Elem -> TreeBitSet -> Property)
  quickCheck (ax5 :: Int -> Property)
  quickCheck (ax6 :: Elem -> TreeBitSet -> Property)
  quickCheck (ax7 :: Elem -> TreeBitSet -> Property)
  quickCheck (ax11 :: Int -> Property)
  quickCheck (ax12 :: Elem -> TreeBitSet -> Property)
  quickCheck (ax13 :: Elem -> TreeBitSet -> Property)
```