**Importante:** Al final del examen, solo debes subir los siguientes archivos sin comprimir: TreeBitSet.hs (ubicado en la carpeta DataStructures/Set) y TreeBitSet.java (ubicado en la carpeta dataStructures/set). Asegúrate de que los archivos que subas sean los que corresponden a tus soluciones, incluye tu nombre y número de identidad (número de pasaporte para los estudiantes extranjeros) en el comentario inicial de ambos archivos, y que se compilen sin errores. Solo se calificarán los archivos sin errores de compilación. Además de la corrección del programa, se tendrá en cuenta la claridad, simplicidad y eficiencia de tus soluciones.

# TreeBitSet

Un conjunto de bits es una secuencia de bits que representa un conjunto de enteros no negativos. Cada bit indica si un elemento está presente (el bit está puesto a 1) o ausente (el bit está puesto a 0) en el conjunto. Por ejemplo, el conjunto de bits `00001010` representa el conjunto `{1, 3}`, ya que los bits en las posiciones 1 y 3 son 1, y el resto de los bits son 0 (ten en cuenta que el bit más a la derecha tiene la posición 0).

Un **TreeBitSet** es una estructura de datos que representa un conjunto de enteros no negativos usando un árbol binario que almacena conjuntos de bits en sus hojas (a partir de ahora llamaremos a este árbol un "árbol de conjuntos de bits"). Un **TreeBitSet** puede almacenar un número máximo de elementos, por lo que consta de dos componentes: una capacidad (número máximo de elementos que se pueden almacenar en el conjunto que estarán en el rango $[0, \mathit{capacidad} - 1]$), y un árbol de conjuntos de bits, que almacena los elementos reales en sus hojas. El árbol de conjuntos de bits es un árbol binario perfecto, donde cada nodo es una hoja o un nodo interno con dos hijos. En nuestra implementación, cada nodo hoja contendrá un conjunto de bits de 64 bits, que representa un subconjunto de los elementos en el rango $[0, \mathit{capacidad} - 1]$ de los elementos posibles. Cada nodo interno representa la unión de los conjuntos de bits de sus dos hijos.

La principal ventaja de usar un `TreeBitSet` es reducir la complejidad espacial y temporal de las operaciones de conjuntos, como agregar, eliminar y consultar elementos. Un `TreeBitSet` puede almacenar hasta $64 * 2^{h-1}$ elementos, donde `h` es la altura del árbol, usando solo $2^h$ conjuntos de bits de 64 bits en las hojas. Además, un `TreeBitSet` puede realizar la mayoría de las operaciones de conjunto en tiempo logarítmico, recorriendo el árbol y aplicando operaciones a nivel de bits en los conjuntos de bits relevantes.

La capacidad de un `TreeBitSet` debe ser un múltiplo de 64, ya que cada nodo hoja puede almacenar hasta 64 elementos. La capacidad de cada subárbol es la mitad de la capacidad de su nodo padre. El rango de elementos que cada subárbol puede almacenar está determinado por su posición en el árbol. Por ejemplo, el hijo izquierdo de un nodo puede almacenar elementos desde 0 hasta la mitad de la capacidad del nodo menos 1, y el hijo derecho de un nodo puede almacenar elementos desde la mitad de la capacidad del nodo hasta la capacidad del nodo menos 1. Este proceso donde cada nodo se divide en dos subárboles se repite hasta que la capacidad de los subárboles sea igual a 64, momento en el que los subárboles son hojas.

Para ilustrar cómo funciona el `TreeBitSet`, consideremos un ejemplo de `TreeBitSet` con capacidad 256, que puede almacenar elementos en el rango [0,255]. El árbol de conjuntos de bits correspondiente tiene dos niveles de nodos internos y un nivel de hojas. El nodo raíz tiene capacidad 256, y sus dos hijos, que son nodos internos, tienen capacidad 128. El hijo izquierdo del nodo raíz tiene dos hijos, que son hojas, con capacidad 64. La primera hoja almacena los elementos en el rango [0,63], y la segunda hoja almacena los elementos en el rango [64,127]. El hijo derecho del nodo raíz tiene dos hijos, que son hojas, con capacidad 64. La primera hoja almacena los elementos en el rango [128,191], y la segunda hoja almacena los elementos en el rango [192,255].

Veamos cómo agregar algunos elementos a este conjunto de bits en árbol. Empezamos con un árbol de conjuntos de bits vacío, donde todos los conjuntos de bits son 0, que se ve así (solo mostramos 5 bits por hoja por simplicidad, pero la implementación real usa 64 bits por hoja):

```
          Node
        /     \
     Node     Node
    /   \     /   \
  Leaf  Leaf  Leaf  Leaf
  00000 00000 00000 00000
```

Ahora agregaremos el elemento 4 al árbol de conjuntos de bits. Como 4 es menor que la mitad de la capacidad del nodo raíz (256/2 = 128), el elemento 4 pertenece al hijo izquierdo. Como el hijo izquierdo es un nodo interno, repetimos el mismo procedimiento para este nodo. Como 4 es menor que la mitad de la capacidad del hijo izquierdo (128/2 = 64), el elemento 4 pertenece al hijo izquierdo del hijo izquierdo. Como el hijo izquierdo del hijo izquierdo es una hoja, ponemos el bit en la posición 4 del conjunto de bits de esta hoja a 1. El conjunto de bits se convierte en `10000`. El árbol de conjuntos de bits se ve así:

```
          Node
        /     \
     Node     Node
    /   \     /   \
  Leaf  Leaf  Leaf  Leaf
  10000 00000 00000 00000
```

Ahora agregaremos el elemento 0 a este árbol de conjuntos de bits. El procedimiento es similar, pero terminamos poniendo el bit en la posición 0 de la hoja más a la izquierda, que se convierte en `10001`:

```
          Node
        /     \
     Node     Node
    /   \     /   \
  Leaf  Leaf  Leaf  Leaf
  10001 00000 00000 00000
```

Finalmente, agregaremos el elemento 195 al árbol de conjuntos de bits. Como 195 es mayor o igual que la mitad de la capacidad del nodo raíz (256/2 = 128), el elemento 195 pertenece al hijo derecho. Sin embargo, como los bits en el hijo derecho corresponden a valores en el rango [128,255], restamos la mitad de la capacidad del nodo raíz al elemento, y agregamos 67 (195 - 128) al hijo derecho en su lugar. Como el hijo derecho es un nodo interno, repetimos el mismo procedimiento para este nodo. Como 67 es mayor o igual que la mitad de la capacidad del hijo derecho (128/2 = 64), el elemento 67 pertenece al hijo derecho del hijo derecho. Sin embargo, como los bits en el hijo derecho del hijo derecho corresponden a valores en el rango [192,255], restamos la mitad de la capacidad del hijo derecho al elemento, y agregamos 3 (67 - 64) al hijo derecho del hijo derecho en su lugar. Como el hijo derecho del hijo derecho es una hoja, ponemos el bit en la posición 3 del conjunto de bits de esta hoja a 1. El conjunto de bits se convierte en `01000`:

```
          Node
        /     \
     Node     Node
    /   \     /   \
  Leaf  Leaf  Leaf  Leaf
  10001 00000 00000 01000
```

Para comprobar si un elemento está contenido en el árbol de conjuntos de bits, seguimos un procedimiento similar al de agregar un elemento, pero en lugar de poner el bit en la hoja correspondiente a 1, comprobamos si el bit es 1 o 0. Para eliminar un elemento del árbol de conjuntos de bits, ponemos el bit en la hoja correspondiente a 0.

# Conjuntos de bits en árbol en Haskell

En Haskell, podemos representar esta estructura de datos usando los siguientes tipos de datos:

```
data Tree = Leaf Int | Node Tree Tree deriving Show

data TreeBitSet = TBS Int Tree deriving Show

bitsPerLeaf :: Int
bitsPerLeaf = 64
```

El tipo de datos `Tree` representa un árbol de conjuntos de bits. Un `Tree` puede ser una `Leaf`, que contiene un conjunto de bits de tipo `Int` (cada valor `Int` en nuestra implementación consta de 64 bits), o un `Node`, que contiene dos `Tree`s como sus hijos.

El tipo de datos `TreeBitSet` representa un conjunto de bits en árbol. Un `TreeBitSet` consta de un `Int`, que es su capacidad, y un `Tree`, que es el árbol de conjuntos de bits.

La constante `bitsPerLeaf` es el número de bits en un `Int`, que es 64.

También proporcionamos un módulo llamado `IntBits` que implementa funciones para manipular conjuntos de bits representados como `Int`s (ver funciones y comentarios en el código proporcionado en el módulo `DataStructures.Set.IntBits`).

## Ejercicios

1. **(1 punto)** Implementa la función `makeTree`, que toma un `Int` correspondiente a una capacidad válida ($64 \cdot 2^n$ para algún $n \geq 0$), y devuelve un `Tree` de conjuntos de bits con esa capacidad. La función debe crear un árbol perfecto para dicha capacidad, donde todos los conjuntos de bits sean 0. La función debe usar un algoritmo recursivo, que crea una `Leaf` con conjunto de bits 0 si la capacidad es menor o igual que `bitsPerLeaf`, o un `Node` con dos subárboles de la mitad de la capacidad en caso contrario.
2. **(0.25 puntos)** Se proporciona ya implementada con los códigos, la función `isValidCapacity`, que toma un `Int` como la capacidad de un `TreeBitSet` y devuelve `True` si dicha capacidad es $64 \cdot 2^n$ para algún $n \geq 0$, y `False` en caso contrario. Usando esta función, implementa la función `empty`, que toma un `Int` como la capacidad de un `TreeBitSet` y devuelve un `TreeBitSet` vacío con esa capacidad. La función debe comprobar si la capacidad es positiva y válida, y usar la función `makeTree` para crear el árbol de conjuntos de bits. Si la capacidad es negativa o inválida, la función debe lanzar un error con el mensaje "capacity must be positive" o "capacity must be 64 multiplied by a power of 2" respectivamente.
3. **(0.25 puntos)** Implementa la función `capacity`, que toma un `TreeBitSet` y devuelve la capacidad del conjunto. La función debe simplemente devolver el primer componente del valor `TreeBitSet`.
4. **(0.25 puntos)** Implementa la función `outOfRange`, que toma un `TreeBitSet` y un `Int` como un elemento, y devuelve `True` si el elemento está fuera del rango del conjunto, y `False` en caso contrario. La función debe comprobar si el elemento es negativo o mayor o igual que la capacidad del conjunto.
5. **(1.25 puntos)** Implementa la función `size`, que toma un `TreeBitSet` y devuelve el número total de elementos almacenados en el conjunto. La función debe usar una función auxiliar que toma un `Tree` y devuelve el número total de bits que están puestos a 1 en las hojas del árbol. La función auxiliar puede contar el

```
          Node
        /     \
     Node     Node              Node
    /   \     /   \            /   \
  Leaf  Leaf  Leaf  Leaf       Leaf  Leaf
  10001 00000 00000 01000      00011 01000
```

debería ser el siguiente árbol de conjuntos de bits con capacidad 256:

```
          Node
        /     \
     Node     Node
    /   \     /   \
  Leaf  Leaf  Leaf  Leaf
  10011 01000 00000 01000
```

# Conjuntos de bits en árbol en Java

En Java, podemos representar esta estructura de datos usando la siguiente interfaz y clases:

```
  private interface Tree {
    boolean contains(long element, long capacity);
    void add(long element, long capacity);
    long size();
    List<Long> toList(long capacity);
  }

  private static class Leaf implements Tree {
    private long bitset;

    public Leaf(long bitset) {
      this.bitset = bitset;
    }

    ...
  }

  private static class Node implements Tree {
    private final Tree left, right;

    public Node(Tree left, Tree right) {
      this.left = left;
      this.right = right;
    }

    ...
  }
```

La interfaz `Tree` representa un árbol de conjuntos de bits. Un `Tree` puede ser una `Leaf`, que contiene un conjunto de bits de tipo `long` (cada valor `long` en nuestra implementación consta de 64 bits), o un `Node`, que contiene dos `Tree`s como sus hijos.

La clase `TreeBitSet` representa un conjunto de bits en árbol. Un `TreeBitSet` consta de un `long`, que es su capacidad, y un `Tree`, que es una referencia a la raíz del árbol de conjuntos de bits.

```
public class TreeBitSet {
  private static final int BITS_PER_LEAF = LongBits.bitsPerLong;
  private final long capacity;
  private final Tree root;
```

La constante `BITS_PER_LEAF` es el número de bits en un `long`, que es 64.

También proporcionamos un módulo llamado `LongBits` que implementa métodos para manipular conjuntos de bits representados como `long`s (ver métodos y comentarios en el código proporcionado en la clase `dataStructures.set.LongBits`).

## Ejercicios

1. **(1 punto)** Implementa el método estático privado `makeTree`, que toma un `long` correspondiente a una capacidad válida ($64 \cdot 2^n$ para algún $n \geq 0$), y devuelve un `Tree` de conjuntos de bits con esa capacidad. El método debe crear un árbol perfecto para dicha capacidad, donde todos los conjuntos de bits sean 0. El método debe usar un algoritmo recursivo, que crea una `Leaf` con conjunto de bits 0 si la capacidad es menor o igual que `BITS_PER_LEAF`, o un `Node` con dos subárboles de la mitad de la capacidad en caso contrario.
2. **(0.25 puntos)** Se proporciona ya implementado con los códigos el método estático privado `isValidCapacity`, que toma un `long` como la capacidad de un `TreeBitSet` y devuelve `true` si dicha capacidad es $64 \cdot 2^n$ para algún $n \geq 0$, y `false` en caso contrario. Usando este método, implementa el constructor público `TreeBitSet`, que toma un `long` como la capacidad de un `TreeBitSet` y devuelve un `TreeBitSet` vacío con esa capacidad. El constructor debe comprobar si la capacidad es positiva y válida, y usar el método `makeTree` para crear el árbol de conjuntos de bits. Si la capacidad es negativa o inválida, el constructor debe lanzar un error con el mensaje "capacity must be positive" o "capacity must be 64 multiplied by a power of 2" respectivamente.
3. **(0.25 puntos)** Implementa el método público `capacity` en la clase `TreeBitSet` que devuelve la capacidad del conjunto. El método debe simplemente devolver el valor del atributo correspondiente en la clase.
4. **(0.25 puntos)** Implementa el método privado `outOfRange` en la clase `TreeBitSet` que toma un `long` como un elemento, y devuelve `true` si el elemento está fuera del rango del conjunto, y `false` en caso contrario. El método debe comprobar si el elemento es negativo o mayor o igual que la capacidad del conjunto.
5. **(1.25 puntos)** Implementa el método público `size` en la clase `TreeBitSet` que devuelve el número total de elementos almacenados en el conjunto. Para implementar este método, implementa primero los métodos `size` en las clases `Leaf` y `Node`. Puedes contar el número de bits que están puestos a 1 en el conjunto de bits de una hoja usando el método `LongBits.countOnes`.
6. **(0.25 puntos)** Implementa el método público `isEmpty` en la clase `TreeBitSet` que devuelve `true` si el conjunto está vacío, y `false` en caso contrario.
7. **(1.25 puntos)** Implementa el método público `contains` en la clase `TreeBitSet` que toma un `long` como un elemento, y devuelve `true` si el conjunto contiene el elemento, y `false` en cualquier otro caso. Si el elemento está dentro de la capacidad del conjunto, para implementar este método, implementa primero los métodos auxiliares `contains` en las clases `Leaf` y `Node`, que toman un `long` como un elemento, y un `long` como la capacidad del árbol, y devuelven `true` si el árbol contiene el elemento, y `false` en caso contrario. Los métodos auxiliares deben recorrer recursivamente el subárbol izquierdo o derecho dependiendo de la posición del elemento en el rango. Una vez que se llega a una hoja, el método debe usar el método `LongBits.contains` para comprobar si el bit correspondiente al elemento está puesto en el conjunto de bits.
8. **(1.25 puntos)** Implementa el método público `add` en la clase `TreeBitSet` que toma un `long` como un elemento, y agrega ese elemento al `TreeBitSet`. El método debe comprobar si el elemento proporcionado está fuera del rango del conjunto y lanzar un error con el mensaje "element is out of range" si ese es el caso. En otro caso, para definir este método, implementa primero los métodos auxiliares `add` en las clases `Leaf` y `Node`, que toman un `long` como un elemento, y un `long` como la capacidad del árbol, y agregan ese elemento al `Tree`. Los métodos auxiliares

deben recorrer recursivamente el subárbol izquierdo o derecho dependiendo de la posición del elemento en el rango. Una vez que se llega a una hoja, el método debe usar el método `LongBits.set` para poner a 1 el bit correspondiente al elemento en el conjunto de bits.

9. **(1.25 puntos)** Implementa el método público `toList` en la clase `TreeBitSet` que devuelve una lista con los elementos en el conjunto. Para definir este método, implementa primero los métodos auxiliares `toList` en las clases `Leaf` y `Node`. El método debe usar el método auxiliar recursivo que toma un `long` como la capacidad del árbol y devuelve una lista con los elementos en el árbol. El método auxiliar debe recorrer recursivamente los subárboles izquierdo y derecho. Una vez que se llega a una hoja, el método debe usar el método `LongBits.contains` para obtener una lista con los elementos en el conjunto de bits. Las listas están implementadas en el paquete `dataStructures.list`.

## Solo para estudiantes sin evaluación continua

Para resolver estos ejercicios, es posible que primero necesites definir un constructor privado para la clase `TreeBitSet` que tome un `long` como la capacidad de un `TreeBitSet` y un `Tree` como la raíz del árbol de conjuntos de bits. También puedes necesitar definir un método estático privado para clonar un `Tree`.

10. **(1.5 puntos)** Implementa el método estático público `union`, que toma dos `TreeBitSet`s y devuelve un nuevo `TreeBitSet` obtenido después de calcular la unión de los dos conjuntos. El método debe comprobar si los dos conjuntos tienen la misma capacidad y lanzar un error con el mensaje "sets have different capacities" si no es así. De lo contrario, el método debe usar un método auxiliar privado recursivo que toma dos `Tree`s y devuelve el nuevo `Tree`. El método auxiliar debe recorrer los subárboles izquierdo y derecho. Una vez que se llega a una hoja, el método debe usar el método `LongBits.or` para calcular la unión de los dos conjuntos de bits. El método debe construir su resultado recorriendo directamente los árboles de los dos conjuntos, y no convirtiendo los conjuntos en listas u otra estructura de datos y luego de vuelta a árboles.
11. **(1.5 puntos)** Implementa el método estático público `extendedUnion`, que toma dos `TreeBitSet`s como conjuntos, y devuelve un nuevo `TreeBitSet` obtenido después de calcular la unión de los dos conjuntos. El método debe ser capaz de calcular la unión de ambos conjuntos incluso si sus capacidades son diferentes. El método debe construir su resultado recorriendo directamente los árboles de los dos conjuntos, y no convirtiendo los conjuntos en listas u otra estructura de datos y luego de vuelta a árboles.