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

import dataStructures.list.List;

public class TreeBitSet {
  private static final int BITS_PER_LEAF = LongBits.BITS_PER_LONG;

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

  private final Tree root;
  private final long capacity;

  // returns true if capacity is 64 * 2^n for some n >= 0
  private static boolean isValidCapacity(long capacity) {
    if (capacity <= 0) {
      return false;
    }
    while (capacity > BITS_PER_LEAF) {
      if (capacity % 2 != 0) {
        return false;
      }
      capacity /= 2;
    }
    return capacity == BITS_PER_LEAF;
  }

//---------------------------------------------------------------------
// DO NOT MODIFY ANY CODE ABOVE THIS LINE
//---------------------------------------------------------------------

  private static class Leaf implements Tree {
    private long bitset;

    public Leaf(long bitset) {
      this.bitset = bitset;
    }
    public long size() {
      return LongBits.countOnes(bitset);
    }
    public boolean contains(long element, long capacity) {
      return LongBits.contains(bitset, element);
    }
    public void add(long element, long capacity) {
      bitset = LongBits.set(bitset, element);
    }
    public List<Long> toList(long capacity) {
      List<Long> list = new LinkedList<>();
      for (long i = 0; i < BITS_PER_LEAF; i++)
        if (LongBits.contains(bitset, i))
          list.append(i);
      return list;
    }
    public Tree cloneTree() {
        return new Leaf(bitset);
    }
  }
```

```
 66
 67  private static class Node implements Tree {
 68    private final Tree left, right;
 69
 70    public Node(Tree left, Tree right) {
 71      this.left = left;
 72      this.right = right;
 73    }
 74    public long size() {
 75      return left.size() + right.size();
 76    }
 77    public boolean contains(long element, long capacity) {
 78      long half = capacity / 2;
 79      return element < half ? left.contains(element, half) : right.contains(element
   - half, half);
 80    }
 81    public void add(long element, long capacity) {
 82      long half = capacity / 2;
 83      if (element < half)
 84        left.add(element, half);
 85      else
 86        right.add(element - half, half);
 87    }
 88    public List<Long> toList(long capacity) {
 89      long half = capacity / 2;
 90      List<Long> leftList = left.toList(half);
 91      List<Long> rightList = right.toList(half);
 92      for (long element : rightList)
 93        leftList.append(element + half);
 94      return leftList;
 95    }
 96    public Tree cloneTree() {
 97        return new Node(left.cloneTree(), right.cloneTree());
 98    }
 99  }
100
101
102  // * Exercise 1 * -
103  private static Tree makeTree(long capacity) {
104    if (capacity <= BITS_PER_LEAF) {
105        return new Leaf(0);
106    } else {
107        long half = capacity / 2;
108        return new Node(makeTree(half), makeTree(half));
109    }
110  }
111
112  // * Exercise 2 * -
113  public TreeBitSet(long capacity) {
114    if (capacity <= 0)
115      throw new IllegalArgumentException("capacity must be positive");
116
117    if (!isValidCapacity(capacity))
118      throw new IllegalArgumentException("capacity must be 64 multiplied by a power
   of 2");
119
120    this.root = makeTree(capacity);
121    this.capacity = capacity;
122  }
123
124  // * Exercise 3 * -
125  public long capacity() {
126    return capacity;
127  }
```

```
128
129   // * Exercise 4 * -
130   private boolean outOfRange(long element) {
131     return element < 0 || element >= capacity;
132   }
133
134   // * Exercise 5 * -
135   public long size() {
136     return root.size();
137   }
138
139   // El método size en la clase TreeBitSet devuelve el tamaño total del conjunto,
    que es el número de elementos distintos en el conjunto.
140   //   root es el árbol subyacente que representa el conjunto en la implementación
    del árbol de bits.
141   //   root.size() invoca el método size del árbol root.
142   //   Este método es parte de la interfaz Tree (ya sea Leaf o Node), y su
    implementación depende de si el nodo es una hoja (Leaf) o un nodo interno (Node).
143
144   // Por lo tanto, el método size() en la clase TreeBitSet simplemente delega la
    llamada al método size del árbol raíz (root).
145   // La implementación específica de size en los nodos (Leaf o Node) se encargará
    de calcular el tamaño efectivo del conjunto dependiendo de la estructura del árbol
146   // y cómo se manejan las hojas y los nodos internos.
147
148   // * Exercise 6 * -
149   public boolean isEmpty() {
150     return size() == 0;
151   }
152
153   // * Exercise 7 * -
154   public boolean contains(long element) {
155     if (outOfRange(element))
156       return false;
157     return root.contains(element, capacity);
158   }
159
160   // * Exercise 8 * -
161   public void add(long element) {
162     if (outOfRange(element))
163       throw new IllegalArgumentException("element is out of range");
164     root.add(element, capacity);
165   }
166
167   // * Exercise 9 * -
168   public List<Long> toList() {
169     return root.toList(capacity);
170   }
171
172
173   //-----------------------------------------------------------------------
174   // Only for students without continuous assessment
175   //-----------------------------------------------------------------------
176
177   private TreeBitSet(long capacity, Tree root) {
178         this.capacity = capacity;
179         this.root = root;
180   }
181
182   // * Exercise 10 * -
183   public static TreeBitSet union(TreeBitSet set1, TreeBitSet set2) {
184         if (set1.capacity() != set2.capacity()) {
185           throw new IllegalArgumentException("sets have different capacities");
186         }
```

```
249     System.out.println(set.contains(270));
250
251     System.out.println(set.contains(11));
252     System.out.println(set.contains(272));
253
254     System.out.println(set.toList());
255   }
256 }
```