```java
import java.util.Iterator;

import components.binarytree.BinaryTree;
import components.binarytree.BinaryTree1;
import components.set.Set;
import components.set.SetSecondary;

/**
 * {@code Set} represented as a {@code BinaryTree} (maintained as a binary
 * search tree) of elements with implementations of primary methods.
 *
 * @param <T>
 *            type of {@code Set} elements
 * @mathdefinitions <pre>
 * IS_BST(
 *   tree: binary tree of T
 *  ): boolean satisfies
 *  [tree satisfies the binary search tree properties as described in the
 *   slides with the ordering reported by compareTo for T, including that
 *   it has no duplicate labels]
 * </pre>
 * @convention IS_BST($this.tree)
 * @correspondence this = labels($this.tree)
 *
 * @author Evan Frisbie & Charan Nanduri
 *
 */
public class Set3a<T extends Comparable<T>> extends SetSecondary<T> {

    /*
     * Private members
     * --------------------------------------------------------------
     */

    /**
     * Elements included in {@code this}.
     */
    private BinaryTree<T> tree;

    /**
```

```java
     * Returns whether {@code x} is in {@code t}.
     *
     * @param <T>
     *            type of {@code BinaryTree} labels
     * @param t
     *            the {@code BinaryTree} to be searched
     * @param x
     *            the label to be searched for
     * @return true if t contains x, false otherwise
     * @requires IS_BST(t)
     * @ensures isInTree = (x is in labels(t))
     */
    private static <T extends Comparable<T>> boolean isInTree(BinaryTree<T> t,
            T x) {
        assert t != null : "Violation of: t is not null";
        assert x != null : "Violation of: x is not null";

        //create a boolean to hold our resulting value
        boolean result = false;

        //split into a left and right tree and get size
        BinaryTree<T> l = t.newInstance(), r = t.newInstance();
        int size = t.size();

        //check if we have found our value or recursively call this function to
        //check again
        if (size > 0) {
            T root = t.disassemble(l, r);

            if (root.equals(x)) {
                result = true;
            } else if (root.compareTo(x) < 0) {
                result = isInTree(r, x);
            } else if (root.compareTo(x) > 0) {
                result = isInTree(l, x);
            }

            t.assemble(root, l, r);
        }

        //return our result
        return result;
    }
```

```
 83
 84      /**
 85       * Inserts {@code x} in {@code t}.
 86       *
 87       * @param <T>
 88       *            type of {@code BinaryTree} labels
 89       * @param t
 90       *            the {@code BinaryTree} to be searched
 91       * @param x
 92       *            the label to be inserted
 93       * @aliases reference {@code x}
 94       * @updates t
 95       * @requires IS_BST(t) and x is not in labels(t)
 96       * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
 97       */
 98      private static <T extends Comparable<T>> void
    insertInTree(BinaryTree<T> t,
 99              T x) {
100          assert t != null : "Violation of: t is not null";
101          assert x != null : "Violation of: x is not null";
102
103          // start by getting the tree's size
104          int size = t.size();
105
106          //Break down the tree, check values, insert the x in the
    correct position
107          //then reassemble the tree
108          if (size != 0) {
109
110              BinaryTree<T> l = t.newInstance(), r = t.newInstance();
111              T root = t.disassemble(l, r);
112
113              if (x.compareTo(root) < 0) {
114                  insertInTree(l, x);
115              } else {
116                  insertInTree(r, x);
117              }
118
119              t.assemble(root, l, r);
120          } else if (size == 0) {
121              //if the tree is empty then create a new tree with just
    a root x
122
123              BinaryTree<T> lnull = t.newInstance();
124              BinaryTree<T> rnull = t.newInstance();
```

```
125
126             t.assemble(x, lnull, rnull);
127         }
128
129     }
130
131     /**
132      * Removes and returns the smallest (left-most) label in {@code
    t}.
133      *
134      * @param <T>
135      *            type of {@code BinaryTree} labels
136      * @param t
137      *            the {@code BinaryTree} from which to remove the
    label
138      * @return the smallest label in the given {@code BinaryTree}
139      * @updates t
140      * @requires IS_BST(t) and |t| > 0
141      * @ensures <pre>
142      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]
    and
143      *  labels(t) = labels(#t) \ {removeSmallest}
144      * </pre>
145      */
146     private static <T> T removeSmallest(BinaryTree<T> t) {
147         assert t != null : "Violation of: t is not null";
148         assert t.size() > 0 : "Violation of: |t| > 0";
149
150         // Create a result of type T
151         T result = null;
152
153         //split the tree and disassemble to the root
154         BinaryTree<T> l = t.newInstance(), r = t.newInstance();
155         T root = t.disassemble(l, r);
156         int leftTreeSize = l.size();
157
158         //Until the tree is empty, move down recursively and
    reassemble on the
159         //way out. If it is empty, remove it and set it as the
    result.
160         if (leftTreeSize > 0) {
161             result = removeSmallest(l);
162             t.assemble(root, l, r);
163         } else if (leftTreeSize == 0) {
164             result = root;
```

```
165             t.transferFrom(r);
166         }
167 
168         //Finally, return the node that was removed.
169         return result;
170     }
171 
172     /**
173      * Finds label {@code x} in {@code t}, removes it from {@code
    t}, and
174      * returns it.
175      *
176      * @param <T>
177      *            type of {@code BinaryTree} labels
178      * @param t
179      *            the {@code BinaryTree} from which to remove label
    {@code x}
180      * @param x
181      *            the label to be removed
182      * @return the removed label
183      * @updates t
184      * @requires IS_BST(t) and x is in labels(t)
185      * @ensures <pre>
186      * IS_BST(t)  and  removeFromTree = x  and
187      *  labels(t) = labels(#t) \ {x}
188      * </pre>
189      */
190     private static <T extends Comparable<T>> T
    removeFromTree(BinaryTree<T> t,
191             T x) {
192         assert t != null : "Violation of: t is not null";
193         assert x != null : "Violation of: x is not null";
194         assert t.size() > 0 : "Violation of: x is in labels(t)";
195 
196         T removed;
197         BinaryTree<T> left = t.newInstance();
198         BinaryTree<T> right = t.newInstance();
199         T rootNode = t.disassemble(left, right);
200 
201         if (x.compareTo(rootNode) < 0) {
202             // search through left side of tree
203             removed = removeFromTree(left, x);
204             // assemble tree
205             t.assemble(rootNode, left, right);
206         } else if (x.compareTo(rootNode) > 0) {
```

```
207             // search through right side of tree
208             removed = removeFromTree(right, x);
209             // reassemble tree
210             t.assemble(rootNode, left, right);
211         } else { // base case
212             removed = rootNode;
213             // reassemble tree
214             // left most node on right tree becomes root node
215             // if right tree is empty, then reassembled tree is
    just left side
216             if (right.size() > 0) {
217                 t.assemble(removeSmallest(right), left, right);
218             } else {
219                 t.transferFrom(left);
220             }
221         }
222 
223         return removed;
224     }
225 
226     /**
227      * Creator of initial representation.
228      */
229     private void createNewRep() {
230 
231         //just create a representation and save it to the tree
    value of this
232         this.tree = new BinaryTree1<T>();
233 
234     }
235 
236     /*
237      * Constructors
    -----------------------------------------------------------
238      */
239 
240     /**
241      * No-argument constructor.
242      */
243     public Set3a() {
244 
245         this.createNewRep();
246 
247     }
248 
```

```
249     /*
250      * Standard methods
    ------------------------------------------------------
251      */
252
253     @SuppressWarnings("unchecked")
254     @Override
255     public final Set<T> newInstance() {
256         try {
257             return this.getClass().getConstructor().newInstance();
258         } catch (ReflectiveOperationException e) {
259             throw new AssertionError(
260                     "Cannot construct object of type " +
    this.getClass());
261         }
262     }
263
264     @Override
265     public final void clear() {
266         this.createNewRep();
267     }
268
269     @Override
270     public final void transferFrom(Set<T> source) {
271         assert source != null : "Violation of: source is not null";
272         assert source != this : "Violation of: source is not this";
273         assert source instanceof Set3a<?> : ""
274                 + "Violation of: source is of dynamic type Set3<?
    >";
275         /*
276          * This cast cannot fail since the assert above would have
    stopped
277          * execution in that case: source must be of dynamic type
    Set3a<?>, and
278          * the ? must be T or the call would not have compiled.
279          */
280         Set3a<T> localSource = (Set3a<T>) source;
281         this.tree = localSource.tree;
282         localSource.createNewRep();
283     }
284
285     /*
286      * Kernel methods
    ------------------------------------------------------
287      */
```

```
288
289     @Override
290     public final void add(T x) {
291         assert x != null : "Violation of: x is not null";
292         assert !this.contains(x) : "Violation of: x is not in
    this";
293
294         //Just call our function that will sort and place
295         insertInTree(this.tree, x);
296
297     }
298
299     @Override
300     public final T remove(T x) {
301         assert x != null : "Violation of: x is not null";
302         assert this.contains(x) : "Violation of: x is in this";
303         /*
304          * //Create a T value to hold the result T result = null;
305          *
306          * //Remove x from the tree and assign result =
307          * removeFromTree(this.tree, x);
308          */
309         //return our value
310         return removeFromTree(this.tree, x);
311     }
312
313     @Override
314     public final T removeAny() {
315         assert this.size() > 0 : "Violation of: this /= empty_set";
316
317         //Create a T value to hold the result
318         T result = null;
319
320         //Remove the smallest value from the tree and assign
321         result = removeSmallest(this.tree);
322
323         //return our value
324         return result;
325     }
326
327     @Override
328     public final boolean contains(T x) {
329         assert x != null : "Violation of: x is not null";
330
331         //Create a boolean value to hold the result, initialize to
```

```
   false
        boolean result = false;

        //use isInTree to check if the value is contained.
        result = isInTree(this.tree, x);

        //return our value
        return result;
    }

    @Override
    public final int size() {

        int size = this.tree.size();

        return size;
    }

    @Override
    public final Iterator<T> iterator() {
        return this.tree.iterator();
    }

}

```