```java
import java.util.Iterator;

/**
 * {@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 Alex Honigford and Jonny Pater
 *
 */
public class Set3a<T extends Comparable<T>> extends SetSecondary<T> {

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

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

    /**
     * Returns whether {@code x} is in {@code t}.
     *
     * @param <T>
     *            type of {@code BinaryTree} labels
```

```
44      * @param t
45      *            the {@code BinaryTree} to be searched
46      * @param x
47      *            the label to be searched for
48      * @return true if t contains x, false otherwise
49      * @requires IS_BST(t)
50      * @ensures isInTree = (x is in labels(t))
51      */
52     private static <T extends Comparable<T>> boolean
   isInTree(BinaryTree<T> t,
53             T x) {
54         assert t != null : "Violation of: t is not null";
55         assert x != null : "Violation of: x is not null";
56         boolean inTree = false;
57         if (t.size() > 0) {
58             BinaryTree<T> left = t.newInstance();
59             BinaryTree<T> right = t.newInstance();
60             T root = t.disassemble(left, right);
61             inTree = root.equals(x) || isInTree(left, x) ||
   isInTree(right, x);
62             t.assemble(root, left, right);
63         }
64         return inTree;
65     }
66
67     /**
68      * Inserts {@code x} in {@code t}.
69      *
70      * @param <T>
71      *            type of {@code BinaryTree} labels
72      * @param t
73      *            the {@code BinaryTree} to be searched
74      * @param x
75      *            the label to be inserted
76      * @aliases reference {@code x}
77      * @updates t
78      * @requires IS_BST(t) and x is not in labels(t)
79      * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
80      */
81     private static <T extends Comparable<T>> void
   insertInTree(BinaryTree<T> t,
82             T x) {
83         assert t != null : "Violation of: t is not null";
84         assert x != null : "Violation of: x is not null";
```

```
 85         BinaryTree<T> left = t.newInstance();
 86         BinaryTree<T> right = t.newInstance();
 87         if (t.size() == 0) {
 88             t.assemble(x, left, right);
 89         } else {
 90             T root = t.disassemble(left, right);
 91             int compare = x.compareTo(root);
 92             if (compare < 0) {
 93                 insertInTree(left, x);
 94             } else if (compare > 0) {
 95                 insertInTree(right, x);
 96             }
 97             t.assemble(root, left, right);
 98         }
 99     }
100
101     /**
102      * Removes and returns the smallest (left-most) label in
    {@code t}.
103      *
104      * @param <T>
105      *            type of {@code BinaryTree} labels
106      * @param t
107      *            the {@code BinaryTree} from which to remove the
    label
108      * @return the smallest label in the given {@code BinaryTree}
109      * @updates t
110      * @requires IS_BST(t) and |t| > 0
111      * @ensures <pre>
112      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]
    and
113      *  labels(t) = labels(#t) \ {removeSmallest}
114      * </pre>
115      */
116     private static <T> T removeSmallest(BinaryTree<T> t) {
117         assert t != null : "Violation of: t is not null";
118         assert t.size() > 0 : "Violation of: |t| > 0";
119         BinaryTree<T> left = t.newInstance();
120         BinaryTree<T> right = t.newInstance();
121         T root = t.disassemble(left, right);
122         T smallest;
123         if (left.size() > 0) {
124             smallest = removeSmallest(left);
125             t.assemble(root, left, right);
```

```
126         } else {
127             smallest = root;
128             t.transferFrom(right);
129         }
130         return smallest;
131     }
132
133     /**
134      * Finds label {@code x} in {@code t}, removes it from {@code
    t}, and
135      * returns it.
136      *
137      * @param <T>
138      *            type of {@code BinaryTree} labels
139      * @param t
140      *            the {@code BinaryTree} from which to remove
    label {@code x}
141      * @param x
142      *            the label to be removed
143      * @return the removed label
144      * @updates t
145      * @requires IS_BST(t) and x is in labels(t)
146      * @ensures <pre>
147      * IS_BST(t)  and  removeFromTree = x  and
148      *  labels(t) = labels(#t) \ {x}
149      * </pre>
150      */
151     private static <T extends Comparable<T>> T
    removeFromTree(BinaryTree<T> t,
152             T x) {
153         assert t != null : "Violation of: t is not null";
154         assert x != null : "Violation of: x is not null";
155         assert t.size() > 0 : "Violation of: x is in labels(t)";
156
157         BinaryTree<T> left = t.newInstance();
158         BinaryTree<T> right = t.newInstance();
159         T removed;
160         T root = t.disassemble(left, right);
161         int compare = x.compareTo(root);
162         if (compare < 0) {
163             removed = removeFromTree(left, x);
164             t.assemble(root, left, right);
165         } else if (compare > 0) {
166             removed = removeFromTree(right, x);
```

```
167             t.assemble(root, left, right);
168         } else {
169             removed = root;
170             if (right.size() > 0) {
171                 T nextSmallest = removeSmallest(right);
172                 t.assemble(root, left, right);
173                 t.replaceRoot(nextSmallest);
174             } else {
175                 t.transferFrom(left);
176             }
177         }
178 
179         return removed;
180     }
181 
182     /**
183      * Creator of initial representation.
184      */
185     private void createNewRep() {
186 
187         this.tree = new BinaryTree1<>();
188 
189     }
190 
191     /*
192      * Constructors
    -----------------------------------------------------------------
193      */
194 
195     /**
196      * No-argument constructor.
197      */
198     public Set3a() {
199 
200         this.createNewRep();
201     }
202 
203     /*
204      * Standard methods
    -----------------------------------------------------------------
205      */
206 
207     @SuppressWarnings("unchecked")
208     @Override
```

```
209     public final Set<T> newInstance() {
210         try {
211             return this.getClass().getConstructor().newInstance();
212         } catch (ReflectiveOperationException e) {
213             throw new AssertionError(
214                     "Cannot construct object of type " +
    this.getClass());
215         }
216     }
217
218     @Override
219     public final void clear() {
220         this.createNewRep();
221     }
222
223     @Override
224     public final void transferFrom(Set<T> source) {
225         assert source != null : "Violation of: source is not
    null";
226         assert source != this : "Violation of: source is not
    this";
227         assert source instanceof Set3a<?> : ""
228                 + "Violation of: source is of dynamic type Set3<?
    >";
229         /*
230          * This cast cannot fail since the assert above would have
    stopped
231          * execution in that case: source must be of dynamic type
    Set3a<?>, and
232          * the ? must be T or the call would not have compiled.
233          */
234         Set3a<T> localSource = (Set3a<T>) source;
235         this.tree = localSource.tree;
236         localSource.createNewRep();
237     }
238
239     /*
240      * Kernel methods
    ------------------------------------------------------------
241      */
242
243     @Override
244     public final void add(T x) {
245         assert x != null : "Violation of: x is not null";
```

```
246         assert !this.contains(x) : "Violation of: x is not in
    this";
247
248         insertInTree(this.tree, x);
249     }
250
251     @Override
252     public final T remove(T x) {
253         assert x != null : "Violation of: x is not null";
254         assert this.contains(x) : "Violation of: x is in this";
255
256         return removeFromTree(this.tree, x);
257     }
258
259     @Override
260     public final T removeAny() {
261         assert this.size() > 0 : "Violation of: this /=
    empty_set";
262
263         return removeSmallest(this.tree);
264     }
265
266     @Override
267     public final boolean contains(T x) {
268         assert x != null : "Violation of: x is not null";
269
270         return isInTree(this.tree, x);
271     }
272
273     @Override
274     public final int size() {
275
276         return this.tree.size();
277     }
278
279     @Override
280     public final Iterator<T> iterator() {
281         return this.tree.iterator();
282     }
283
284 }
285
```