```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 Bashir Ali and Kwasi Fosu
 *
 */
public class Set3a<T extends Comparable<T>> extends SetSecondary<T> {

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

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

```

```
39      /**
40       * Returns whether {@code x} is in {@code t}.
41       *
42       * @param <T>
43       *            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          boolean inTree = false;
55          if (t.size() > 0) {
56              BinaryTree<T> left = t.newInstance();
57              BinaryTree<T> right = t.newInstance();
58              T root = t.disassemble(left, right);
59              if (x.compareTo(root) < 0) {
60                  inTree = isInTree(left, x);
61              } else if (x.compareTo(root) > 0) {
62                  inTree = isInTree(right, x);
63              } else {
64                  inTree = root.equals(x);
65              }
66              t.assemble(root, left, right);
67          }
68          return inTree;
69      }
70
71      /**
72       * Inserts {@code x} in {@code t}.
73       *
74       * @param <T>
75       *            type of {@code BinaryTree} labels
76       * @param t
```

```
 77      *            the {@code BinaryTree} to be searched
 78      * @param x
 79      *            the label to be inserted
 80      * @aliases reference {@code x}
 81      * @updates t
 82      * @requires IS_BST(t) and x is not in labels(t)
 83      * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
 84      */
 85     private static <T extends Comparable<T>> void
    insertInTree(BinaryTree<T> t,
 86             T x) {
 87         assert t != null : "Violation of: t is not null";
 88         assert x != null : "Violation of: x is not null";
 89
 90         BinaryTree<T> left = t.newInstance();
 91         BinaryTree<T> right = t.newInstance();
 92         if (t.size() == 0) {
 93             t.assemble(x, left, right);
 94         } else {
 95             T root = t.disassemble(left, right);
 96             if (x.compareTo(root) < 0) {
 97                 insertInTree(left, x);
 98             } else {
 99                 insertInTree(right, x);
100             }
101             t.assemble(root, left, right);
102         }
103     }
104
105     /**
106      * Removes and returns the smallest (left-most) label in
    {@code t}.
107      *
108      * @param <T>
109      *            type of {@code BinaryTree} labels
110      * @param t
111      *            the {@code BinaryTree} from which to remove
    the label
112      * @return the smallest label in the given {@code
```

```
    BinaryTree}
113      * @updates t
114      * @requires IS_BST(t) and |t| > 0
115      * @ensures <pre>
116      * IS_BST(t)  and  removeSmallest = [the smallest label in
    #t]  and
117      *  labels(t) = labels(#t) \ {removeSmallest}
118      * </pre>
119      */
120     private static <T> T removeSmallest(BinaryTree<T> t) {
121         T smallest = t.root();
122         BinaryTree<T> left = t.newInstance();
123         BinaryTree<T> right = t.newInstance();
124         T root = t.disassemble(left, right);
125
126         if (left.size() > 0) {
127             smallest = removeSmallest(left);
128             t.assemble(root, left, right);
129         } else {
130             smallest = root;
131             t.transferFrom(right);
132         }
133         return smallest;
134     }
135
136     /**
137      * Finds label {@code x} in {@code t}, removes it from
    {@code t}, and
138      * returns it.
139      *
140      * @param <T>
141      *            type of {@code BinaryTree} labels
142      * @param t
143      *            the {@code BinaryTree} from which to remove
    label {@code x}
144      * @param x
145      *            the label to be removed
146      * @return the removed label
147      * @updates t
```

```
148      * @requires IS_BST(t) and x is in labels(t)
149      * @ensures <pre>
150      * IS_BST(t)  and  removeFromTree = x  and
151      *  labels(t) = labels(#t) \ {x}
152      * </pre>
153      */
154     private static <T extends Comparable<T>> T
    removeFromTree(BinaryTree<T> t,
155             T x) {
156         assert t != null : "Violation of: t is not null";
157         assert x != null : "Violation of: x is not null";
158         assert t.size() > 0 : "Violation of: x is in
    labels(t)";
159 
160         T removed = t.root();
161         BinaryTree<T> left = t.newInstance();
162         BinaryTree<T> right = t.newInstance();
163         T root = t.disassemble(left, right);
164 
165         if (x.compareTo(root) == 0) {
166             if (right.size() > 0) {
167                 removed = removeSmallest(right);
168                 t.assemble(removed, left, right);
169             } else {
170                 t.transferFrom(left);
171             }
172         } else {
173             T oldRoot = root;
174             if (x.compareTo(root) < 0) {
175                 root = removeFromTree(left, x);
176                 t.assemble(oldRoot, left, right);
177             } else {
178                 root = removeFromTree(right, x);
179                 t.assemble(oldRoot, left, right);
180             }
181 
182         }
183         return root;
184     }
```

```
185
186     /**
187      * Creator of initial representation.
188      */
189     private void createNewRep() {
190
191         this.tree = new BinaryTree1<T>();
192
193     }
194
195     /*
196      * Constructors
    -----------------------------------------------------------------
197      */
198
199     /**
200      * No-argument constructor.
201      */
202     public Set3a() {
203
204         this.createNewRep();
205
206     }
207
208     /*
209      * Standard methods
    -------------------------------------------------------------
210      */
211
212     @SuppressWarnings("unchecked")
213     @Override
214     public final Set<T> newInstance() {
215         try {
216             return
    this.getClass().getConstructor().newInstance();
217         } catch (ReflectiveOperationException e) {
218             throw new AssertionError(
219                     "Cannot construct object of type " +
    this.getClass());
```

```
220         }
221     }
222
223     @Override
224     public final void clear() {
225         this.createNewRep();
226     }
227
228     @Override
229     public final void transferFrom(Set<T> source) {
230         assert source != null : "Violation of: source is not
    null";
231         assert source != this : "Violation of: source is not
    this";
232         assert source instanceof Set3a<?> : ""
233                 + "Violation of: source is of dynamic type
    Set3<?>";
234         /*
235          * This cast cannot fail since the assert above would
    have stopped
236          * execution in that case: source must be of dynamic
    type Set3a<?>, and
237          * the ? must be T or the call would not have compiled.
238          */
239         Set3a<T> localSource = (Set3a<T>) source;
240         this.tree = localSource.tree;
241         localSource.createNewRep();
242     }
243
244     /*
245      * Kernel methods
    ------------------------------------------------------------
246      */
247
248     @Override
249     public final void add(T x) {
250         assert x != null : "Violation of: x is not null";
251         assert !this.contains(x) : "Violation of: x is not in
    this";
```

```
252
253         insertInTree(this.tree, x);
254
255     }
256
257     @Override
258     public final T remove(T x) {
259         assert x != null : "Violation of: x is not null";
260         assert this.contains(x) : "Violation of: x is in this";
261
262         return removeFromTree(this.tree, x);
263     }
264
265     @Override
266     public final T removeAny() {
267         assert this.size() > 0 : "Violation of: this /=
    empty_set";
268
269         return removeSmallest(this.tree);
270     }
271
272     @Override
273     public final boolean contains(T x) {
274         assert x != null : "Violation of: x is not null";
275
276         return isInTree(this.tree, x);
277     }
278
279     @Override
280     public final int size() {
281
282         return this.tree.size();
283     }
284
285     @Override
286     public final Iterator<T> iterator() {
287         return this.tree.iterator();
288     }
289
```

```
290 }
291
```