```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 Zhuoyang Li + Xinci Ma
 *
 */
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
     * @param t
     *            the {@code BinaryTree} to be searched
     * @param x
```

```
48      *            the label to be searched for
49      * @return true if t contains x, false otherwise
50      * @requires IS_BST(t)
51      * @ensures isInTree = (x is in labels(t))
52      */
53     private static <T extends Comparable<T>> boolean
   isInTree(BinaryTree<T> t,
54             T x) {
55         assert t != null : "Violation of: t is not null";
56         assert x != null : "Violation of: x is not null";
57
58         boolean result = false;
59         BinaryTree<T> left = t.newInstance();
60         BinaryTree<T> right = t.newInstance();
61         if (t.size() == 0) {
62             result = false; // x is not in an empty tree t
63         } else {
64             T r = t.disassemble(left, right);
65             if (r.equals(x)) {
66                 result = true; // x is in the root of t
67             } else if (r.compareTo(x) > 0) {
68                 result = isInTree(left, x); // x might be in
   the left tree
69             } else {
70                 result = isInTree(right, x); // x might be in
   the right tree
71             }
72             t.assemble(r, left, right); // restore t
73         }
74         return result;
75     }
76
77     /**
78      * Inserts {@code x} in {@code t}.
79      *
80      * @param <T>
81      *            type of {@code BinaryTree} labels
82      * @param t
83      *            the {@code BinaryTree} to be searched
84      * @param x
85      *            the label to be inserted
86      * @aliases reference {@code x}
87      * @updates t
88      * @requires IS_BST(t) and x is not in labels(t)
89      * @ensures IS_BST(t) and labels(t) = labels(#t) union
   {x}
90      */
```

```
 91    private static <T extends Comparable<T>> void
   insertInTree(BinaryTree<T> t,
 92            T x) {
 93        assert t != null : "Violation of: t is not null";
 94        assert x != null : "Violation of: x is not null";
 95
 96        BinaryTree<T> left = t.newInstance();
 97        BinaryTree<T> right = t.newInstance();
 98        if (t.size() == 0) {
 99            t.assemble(x, left, right); // insert x in an
   empty tree t
100        } else {
101            T r = t.disassemble(left, right);
102            if (r.compareTo(x) > 0) {
103                insertInTree(left, x); // insert x in the
   left tree
104            } else {
105                insertInTree(right, x); // insert x in the
   right tree
106            }
107            t.assemble(r, left, right); // restore t
108        }
109
110    }
111
112    /**
113     * Removes and returns the smallest (left-most) label in
   {@code t}.
114     *
115     * @param <T>
116     *            type of {@code BinaryTree} labels
117     * @param t
118     *            the {@code BinaryTree} from which to remove
   the label
119     * @return the smallest label in the given {@code
   BinaryTree}
120     * @updates t
121     * @requires IS_BST(t) and |t| > 0
122     * @ensures <pre>
123     * IS_BST(t)  and  removeSmallest = [the smallest label
   in #t]  and
124     *  labels(t) = labels(#t) \ {removeSmallest}
125     * </pre>
126     */
127    private static <T> T removeSmallest(BinaryTree<T> t) {
128        assert t != null : "Violation of: t is not null";
129        assert t.size() > 0 : "Violation of: |t| > 0";
```

```
130
131         T result = t.root();
132         BinaryTree<T> left = t.newInstance();
133         BinaryTree<T> right = t.newInstance();
134         if (t.size() == 1) {
135             result = t.disassemble(left, right);
136             // the smallest label is the root itself
137         } else {
138             T r = t.disassemble(left, right);
139
140             if (left.size() > 0) {
141                 result = removeSmallest(left);
142                 t.assemble(r, left, right);
143             } else {
144                 t.transferFrom(right);
145             }
146         }
147         return result;
148     }
149
150     /**
151      * Finds label {@code x} in {@code t}, removes it from
    {@code t}, and
152      * returns it.
153      *
154      * @param <T>
155      *            type of {@code BinaryTree} labels
156      * @param t
157      *            the {@code BinaryTree} from which to remove
    label {@code x}
158      * @param x
159      *            the label to be removed
160      * @return the removed label
161      * @updates t
162      * @requires IS_BST(t) and x is in labels(t)
163      * @ensures <pre>
164      * IS_BST(t)  and  removeFromTree = x  and
165      *  labels(t) = labels(#t) \ {x}
166      * </pre>
167      */
168     private static <T extends Comparable<T>> T
    removeFromTree(BinaryTree<T> t,
169             T x) {
170         assert t != null : "Violation of: t is not null";
171         assert x != null : "Violation of: x is not null";
172         assert t.size() > 0 : "Violation of: x is in
    labels(t)";
```

```
173
174         T result = t.root();
175         if (t.size() == 1) {
176             t.clear(); // delete the label
177         } else {
178             BinaryTree<T> left = t.newInstance();
179             BinaryTree<T> right = t.newInstance();
180             T r = t.disassemble(left, right);
181             if (r.compareTo(x) > 0) {
182                 result = removeFromTree(left, x); // x is in
    the left tree
183                 t.assemble(r, left, right);
184             } else if (r.compareTo(x) < 0) {
185                 result = removeFromTree(right, x); // x is in
    the right tree
186                 t.assemble(r, left, right);
187             } else {
188                 // x is the root node
189                 result = r;
190                 if (right.size() == 0) {
191                     t.transferFrom(left);
192                 } else {
193                     T smallest = removeSmallest(right);
194                     t.assemble(smallest, left, right);
195                 }
196
197             }
198         }
199
200         return result;
201     }
202
203     /**
204      * Creator of initial representation.
205      */
206     private void createNewRep() {
207
208         this.tree = new BinaryTree1<T>();
209
210     }
211
212     /*
213      * Constructors
    -----------------------------------------------------------
214      */
215
216     /**
```

```
217      * No-argument constructor.
218      */
219     public Set3a() {
220 
221         this.createNewRep();
222 
223     }
224 
225     /*
226      * Standard methods
    ---------------------------------------------------------
227      */
228 
229     @SuppressWarnings("unchecked")
230     @Override
231     public final Set<T> newInstance() {
232         try {
233             return
    this.getClass().getConstructor().newInstance();
234         } catch (ReflectiveOperationException e) {
235             throw new AssertionError(
236                     "Cannot construct object of type " +
    this.getClass());
237         }
238     }
239 
240     @Override
241     public final void clear() {
242         this.createNewRep();
243     }
244 
245     @Override
246     public final void transferFrom(Set<T> source) {
247         assert source != null : "Violation of: source is not
    null";
248         assert source != this : "Violation of: source is not
    this";
249         assert source instanceof Set3a<?> : ""
250                 + "Violation of: source is of dynamic type
    Set3<?>";
251         /*
252          * This cast cannot fail since the assert above would
    have stopped
253          * execution in that case: source must be of dynamic
    type Set3a<?>, and
254          * the ? must be T or the call would not have
    compiled.
```

```
255          */
256         Set3a<T> localSource = (Set3a<T>) source;
257         this.tree = localSource.tree;
258         localSource.createNewRep();
259     }
260
261     /*
262      * Kernel methods
    ------------------------------------------------------------
263      */
264
265     @Override
266     public final void add(T x) {
267         assert x != null : "Violation of: x is not null";
268         assert !this.contains(x) : "Violation of: x is not in
    this";
269
270         insertInTree(this.tree, x);
271
272     }
273
274     @Override
275     public final T remove(T x) {
276         assert x != null : "Violation of: x is not null";
277         assert this.contains(x) : "Violation of: x is in
    this";
278
279         return removeFromTree(this.tree, x);
280
281     }
282
283     @Override
284     public final T removeAny() {
285         assert this.size() > 0 : "Violation of: this /=
    empty_set";
286
287         return this.removeAnyHelper(this.tree);
288     }
289
290     /**
291      * Helper method to remove a random element from the
    tree.
292      *
293      * @param tree
294      *            the binary tree from which to remove an
    element
295      * @return the value removed
```

```
296      */
297     private T removeAnyHelper(BinaryTree<T> tree) {
298         Random rand = new Random();
299         BinaryTree<T> left = tree.newInstance();
300         BinaryTree<T> right = tree.newInstance();
301         T value = tree.disassemble(left, right);
302 
303         int direction = tree.size() == 1 ? -1 :
    rand.nextInt(3);
304         // 0: left, 1: right, -1 or 2: current node
305         T removedValue;
306 
307         if (direction == 0 && left.size() > 0) {
308             // Recursively remove from left
309             removedValue = this.removeAnyHelper(left);
310             tree.assemble(value, left, right);
311         } else if (direction == 1 && right.size() > 0) {
312             // Recursively remove from right
313             removedValue = this.removeAnyHelper(right);
314             tree.assemble(value, left, right);
315         } else {
316             // Remove current node
317             removedValue = value;
318             if (left.size() > 0 && right.size() > 0) {
319                 // Node has two children, find successor
320                 T successor = removeSmallest(right);
321                 tree.assemble(successor, left, right);
322             } else if (left.size() > 0) {
323                 // Node has only left child
324                 tree.transferFrom(left);
325             } else if (right.size() > 0) {
326                 // Node has only right child
327                 tree.transferFrom(right);
328             } else {
329                 // Node is a leaf
330                 tree.clear();
331             }
332         }
333 
334         return removedValue;
335 
336     }
337 
338     @Override
339     public final boolean contains(T x) {
340         assert x != null : "Violation of: x is not null";
341 
```

```
342         return isInTree(this.tree, x);
343     }
344 
345     @Override
346     public final int size() {
347 
348         return this.tree.size();
349     }
350 
351     @Override
352     public final Iterator<T> iterator() {
353         return this.tree.iterator();
354     }
355 
356 }
357 
```