```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 Put your name here
 *
 */
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
     *            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";

        boolean inTree = false;

        if (t.height() > 0) {
            if (t.root().compareTo(x) != 0) { // x != root
                BinaryTree<T> tL = t.newInstance();
                BinaryTree<T> tR = t.newInstance();
```

```
63                T root = t.disassemble(tL, tR);
64                if (root.compareTo(x) < 0) { // x > root
65                    inTree = isInTree(tR, x);
66                } else { // x < root
67                    inTree = isInTree(tL, x);
68                }
69                t.assemble(root, tL, tR);
70            } else { // x == root
71                inTree = true;
72            }
73        }
74
75        return inTree;
76    }
77
78    /**
79     * Inserts {@code x} in {@code t}.
80     *
81     * @param <T>
82     *            type of {@code BinaryTree} labels
83     * @param t
84     *            the {@code BinaryTree} to be searched
85     * @param x
86     *            the label to be inserted
87     * @aliases reference {@code x}
88     * @updates t
89     * @requires IS_BST(t) and x is not in labels(t)
90     * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
91     */
92    private static <T extends Comparable<T>> void insertInTree(BinaryTree<T> t,
93            T x) {
94        assert t != null : "Violation of: t is not null";
95        assert x != null : "Violation of: x is not null";
96
97        if (t.height() > 0) {
98            if (t.root().compareTo(x) != 0) { // x != root
99                BinaryTree<T> tL = t.newInstance();
100               BinaryTree<T> tR = t.newInstance();
101               T root = t.disassemble(tL, tR);
102               if (root.compareTo(x) < 0) { // x > root
103                   insertInTree(tR, x);
104               } else { // x < root
105                   insertInTree(tL, x);
106               }
107               t.assemble(root, tL, tR);
108           }
109       } else {
110           t.assemble(x, t.newInstance(), t.newInstance());
111       }
112
113   }
114
115   /**
116    * Removes and returns the smallest (left-most) label in {@code t}.
117    *
118    * @param <T>
119    *            type of {@code BinaryTree} labels
```

```
120      * @param t
121      *            the {@code BinaryTree} from which to remove the label
122      * @return the smallest label in the given {@code BinaryTree}
123      * @updates t
124      * @requires IS_BST(t) and |t| > 0
125      * @ensures <pre>
126      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]  and
127      *  labels(t) = labels(#t) \ {removeSmallest}
128      * </pre>
129      */
130     private static <T> T removeSmallest(BinaryTree<T> t) {
131         assert t != null : "Violation of: t is not null";
132         assert t.size() > 0 : "Violation of: |t| > 0";
133
134         T smallest;
135
136         if (t.height() > 1) { // t.height > 1; smallest is not root
137             BinaryTree<T> tL = t.newInstance();
138             BinaryTree<T> tR = t.newInstance();
139             T root = t.root();
140             t.disassemble(tL, tR);
141             if (tL.height() > 0) { // search left tree for smallest
142                 smallest = removeSmallest(tL);
143             } else { // else root is smallest; new root = smallest in right tree
144                 smallest = root;
145                 root = removeSmallest(tR);
146             }
147             t.assemble(root, tL, tR);
148         } else { // t.height() == 1; root is smallest
149             smallest = t.root();
150             t.clear();
151         }
152
153         return smallest;
154     }
155
156     /**
157      * Finds label {@code x} in {@code t}, removes it from {@code t}, and
158      * returns it.
159      *
160      * @param <T>
161      *            type of {@code BinaryTree} labels
162      * @param t
163      *            the {@code BinaryTree} from which to remove label {@code x}
164      * @param x
165      *            the label to be removed
166      * @return the removed label
167      * @updates t
168      * @requires IS_BST(t) and x is in labels(t)
169      * @ensures <pre>
170      * IS_BST(t)  and  removeFromTree = x  and
171      *  labels(t) = labels(#t) \ {x}
172      * </pre>
173      */
174     private static <T extends Comparable<T>> T removeFromTree(BinaryTree<T> t,
175             T x) {
176         assert t != null : "Violation of: t is not null";
```

```
177         assert x != null : "Violation of: x is not null";
178         assert t.size() > 0 : "Violation of: x is in labels(t)";
179
180         if (t.height() > 0) {
181             BinaryTree<T> tL = t.newInstance();
182             BinaryTree<T> tR = t.newInstance();
183             T root = t.disassemble(tL, tR);
184             if (root.compareTo(x) != 0) { // x != root
185                 if (root.compareTo(x) < 0) { // x > root
186                     removeFromTree(tR, x);
187                 } else { // x > root
188                     removeFromTree(tL, x);
189                 }
190                 t.assemble(root, tL, tR);
191             } else { // x == root; reconstruct tree with new root
192                 if (t.height() > 1) {
193                     T newRoot;
194                     if (tR.height() > 0) { // new rt = smallest from right tree
195                         newRoot = removeSmallest(tR);
196                     } else { // make the root of the left tree the new root
197                         newRoot = tL.root();
198                         tL.disassemble(tL, tR);
199                     }
200                     // reassemble with new root
201                     t.assemble(newRoot, tL, tR);
202                 } else { // if x == root and t.height() == 1; clear t
203                     t.clear();
204                 }
205             }
206         }
207         return x; // we can do this because of requires x is in t
208     }
209
210     /**
211      * Creator of initial representation.
212      */
213     private void createNewRep() {
214         this.tree = new BinaryTree1<T>();
215     }
216
217     /*
218      * Constructors ------------------------------------------------------------
219      */
220
221     /**
222      * No-argument constructor.
223      */
224     public Set3a() {
225         this.createNewRep();
226     }
227
228     /*
229      * Standard methods --------------------------------------------------------
230      */
231
232     @SuppressWarnings("unchecked")
233     @Override
```

```
234     public final Set<T> newInstance() {
235         try {
236             return this.getClass().getConstructor().newInstance();
237         } catch (ReflectiveOperationException e) {
238             throw new AssertionError(
239                     "Cannot construct object of type " + this.getClass());
240         }
241     }
242
243     @Override
244     public final void clear() {
245         this.createNewRep();
246     }
247
248     @Override
249     public final void transferFrom(Set<T> source) {
250         assert source != null : "Violation of: source is not null";
251         assert source != this : "Violation of: source is not this";
252         assert source instanceof Set3a<?> : ""
253                 + "Violation of: source is of dynamic type Set3<?>";
254         /*
255          * This cast cannot fail since the assert above would have stopped
256          * execution in that case: source must be of dynamic type Set3a<?>, and
257          * the ? must be T or the call would not have compiled.
258          */
259         Set3a<T> localSource = (Set3a<T>) source;
260         this.tree = localSource.tree;
261         localSource.createNewRep();
262     }
263
264     /*
265      * Kernel methods ----------------------------------------------------------
266      */
267
268     @Override
269     public final void add(T x) {
270         assert x != null : "Violation of: x is not null";
271         assert !this.contains(x) : "Violation of: x is not in this";
272
273         insertInTree(this.tree, x);
274     }
275
276     @Override
277     public final T remove(T x) {
278         assert x != null : "Violation of: x is not null";
279         assert this.contains(x) : "Violation of: x is in this";
280
281         return removeFromTree(this.tree, x);
282     }
283
284     @Override
285     public final T removeAny() {
286         assert this.size() > 0 : "Violation of: this /= empty_set";
287
288         return removeSmallest(this.tree);
289     }
290
```

```
291     @Override
292     public final boolean contains(T x) {
293         assert x != null : "Violation of: x is not null";
294
295         return isInTree(this.tree, x);
296     }
297
298     @Override
299     public final int size() {
300
301         return this.tree.size();
302     }
303
304     @Override
305     public final Iterator<T> iterator() {
306         return this.tree.iterator();
307     }
308
309 }
310
```