```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 Gabe Azzarita and Ty Fredrick
 *
 */
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";

        BinaryTree<T> left = t.newInstance();
        BinaryTree<T> right = t.newInstance();

        boolean contains = false;

        if (t.size() > 0) {
            T root = t.disassemble(left, right);

```

```
65              if (x.compareTo(root) == 0) {
66                  contains = true;
67              } else if (x.compareTo(root) < 0) {
68                  // If x < root we search left tree
69                  contains = isInTree(left, x);
70              } else {
71                  // If x > root we search right tree
72                  contains = isInTree(right, x);
73              }
74              t.assemble(root, left, right);
75          }
76
77          return contains;
78      }
79
80      /**
81       * Inserts {@code x} in {@code t}.
82       *
83       * @param <T>
84       *            type of {@code BinaryTree} labels
85       * @param t
86       *            the {@code BinaryTree} to be searched
87       * @param x
88       *            the label to be inserted
89       * @aliases reference {@code x}
90       * @updates t
91       * @requires IS_BST(t) and x is not in labels(t)
92       * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
93       */
94      private static <T extends Comparable<T>> void insertInTree(BinaryTree<T> t,
95              T x) {
96          assert t != null : "Violation of: t is not null";
97          assert x != null : "Violation of: x is not null";
98
99          BinaryTree<T> left = t.newInstance();
100         BinaryTree<T> right = t.newInstance();
101
102         if (t.size() == 0) {
103             // if t is empty, we create new tree with x as root
104             t.assemble(x, left, right);
105         } else {
106             T root = t.disassemble(left, right);
107             /*
108              * If x is smaller than root, we insert into left tree, else, we
109              * insert into right tree
110              */
111             if (x.compareTo(root) < 0) {
112                 insertInTree(left, x);
113             } else {
114                 insertInTree(right, x);
115             }
116             t.assemble(root, left, right);
117         }
118     }
119
120     /**
121      * Removes and returns the smallest (left-most) label in {@code t}.
122      *
123      * @param <T>
```

```
124      *            type of {@code BinaryTree} labels
125      * @param t
126      *            the {@code BinaryTree} from which to remove the label
127      * @return the smallest label in the given {@code BinaryTree}
128      * @updates t
129      * @requires IS_BST(t) and |t| > 0
130      * @ensures <pre>
131      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]  and
132      *  labels(t) = labels(#t) \ {removeSmallest}
133      * </pre>
134      */
135     private static <T> T removeSmallest(BinaryTree<T> t) {
136         assert t != null : "Violation of: t is not null";
137         assert t.size() > 0 : "Violation of: |t| > 0";
138
139         BinaryTree<T> left = t.newInstance();
140         BinaryTree<T> right = t.newInstance();
141
142         T root = t.disassemble(left, right);
143         T min;
144
145         // if left tree is empty, root is smallest
146         if (left.size() == 0) {
147             min = root;
148             t.transferFrom(right);
149         } else {
150             min = removeSmallest(left);
151             t.assemble(root, left, right);
152         }
153
154         return min;
155     }
156
157     /**
158      * Finds label {@code x} in {@code t}, removes it from {@code t}, and
159      * returns it.
160      *
161      * @param <T>
162      *            type of {@code BinaryTree} labels
163      * @param t
164      *            the {@code BinaryTree} from which to remove label {@code x}
165      * @param x
166      *            the label to be removed
167      * @return the removed label
168      * @updates t
169      * @requires IS_BST(t) and x is in labels(t)
170      * @ensures <pre>
171      * IS_BST(t)  and  removeFromTree = x  and
172      *  labels(t) = labels(#t) \ {x}
173      * </pre>
174      */
175     private static <T extends Comparable<T>> T removeFromTree(BinaryTree<T> t,
176             T x) {
177         assert t != null : "Violation of: t is not null";
178         assert x != null : "Violation of: x is not null";
179         assert t.size() > 0 : "Violation of: x is in labels(t)";
180
181         BinaryTree<T> left = t.newInstance();
182         BinaryTree<T> right = t.newInstance();
```

```
183
184         T root = t.disassemble(left, right);
185         T removed;
186
187         // If root = x, we remove root
188         if (root.equals(x)) {
189             removed = root;
190             /*
191              * If right tree is empty, we make the left tree the new tree, else
192              * we make the smallest node in the right tree the new root and
193              * assemble using left and new right tree
194              */
195             if (right.size() == 0) {
196                 t.transferFrom(left);
197             } else {
198                 t.assemble(removeSmallest(right), left, right);
199             }
200         } else if (x.compareTo(root) < 0) {
201             // If x < root, we remove x from the left tree
202             removed = removeFromTree(left, x);
203             t.assemble(root, left, right);
204         } else {
205             // If x > root, we remove x from right tree
206             removed = removeFromTree(right, x);
207             t.assemble(root, left, right);
208         }
209
210         return removed;
211     }
212
213     /**
214      * Creator of initial representation.
215      */
216     private void createNewRep() {
217         this.tree = new BinaryTree1<T>();
218     }
219
220     /*
221      * Constructors -----------------------------------------------------------
222      */
223
224     /**
225      * No-argument constructor.
226      */
227     public Set3a() {
228         this.createNewRep();
229     }
230
231     /*
232      * Standard methods -------------------------------------------------------
233      */
234
235     @SuppressWarnings("unchecked")
236     @Override
237     public final Set<T> newInstance() {
238         try {
239             return this.getClass().getConstructor().newInstance();
240         } catch (ReflectiveOperationException e) {
241             throw new AssertionError(
```

```
242                     "Cannot construct object of type " + this.getClass());
243         }
244     }
245
246     @Override
247     public final void clear() {
248         this.createNewRep();
249     }
250
251     @Override
252     public final void transferFrom(Set<T> source) {
253         assert source != null : "Violation of: source is not null";
254         assert source != this : "Violation of: source is not this";
255         assert source instanceof Set3a<?> : ""
256                 + "Violation of: source is of dynamic type Set3<?>";
257         /*
258          * This cast cannot fail since the assert above would have stopped
259          * execution in that case: source must be of dynamic type Set3a<?>, and
260          * the ? must be T or the call would not have compiled.
261          */
262         Set3a<T> localSource = (Set3a<T>) source;
263         this.tree = localSource.tree;
264         localSource.createNewRep();
265     }
266
267     /*
268      * Kernel methods ------------------------------------------------------
269      */
270
271     @Override
272     public final void add(T x) {
273         assert x != null : "Violation of: x is not null";
274         assert !this.contains(x) : "Violation of: x is not in this";
275
276         insertInTree(this.tree, x);
277     }
278
279     @Override
280     public final T remove(T x) {
281         assert x != null : "Violation of: x is not null";
282         assert this.contains(x) : "Violation of: x is in this";
283
284         return removeFromTree(this.tree, x);
285     }
286
287     @Override
288     public final T removeAny() {
289         assert this.size() > 0 : "Violation of: this /= empty_set";
290
291         return removeSmallest(this.tree);
292     }
293
294     @Override
295     public final boolean contains(T x) {
296         assert x != null : "Violation of: x is not null";
297
298         return isInTree(this.tree, x);
299     }
300
```

```
301     @Override
302     public final int size() {
303         return this.tree.size();
304     }
305 
306     @Override
307     public final Iterator<T> iterator() {
308         return this.tree.iterator();
309     }
310 
311 }
312 
```