```java
import java.util.Iterator;

import components.binarytree.BinaryTree;
import components.binarytree.BinaryTree1;
import components.set.Set;
import components.set.SetSecondary;

/**
 * {@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 David P. and Zach B.
 *
 */
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 found = false;
```

```
58
59          // If the tree is not empty, proceed to check
60          if (t.size() > 0) {
61              BinaryTree<T> left = t.newInstance();
62              BinaryTree<T> right = t.newInstance();
63              T root = t.root();
64              t.disassemble(left, right);
65              // Compare the element with the root
66              int compareResult = x.compareTo(root);
67              if (compareResult == 0) {
68                  found = true; // Element found
69              } else if (compareResult < 0) {
70                  found = isInTree(left, x); // Search in the left subtree
71              } else {
72                  found = isInTree(right, x); // Search in the right subtree
73              }
74              // Reassemble the tree after checking
75              t.assemble(root, left, right);
76          }
77
78          return found;
79      }
80
81      /**
82       * Inserts {@code x} in {@code t}.
83       *
84       * @param <T>
85       *            type of {@code BinaryTree} labels
86       * @param t
87       *            the {@code BinaryTree} to be searched
88       * @param x
89       *            the label to be inserted
90       * @aliases reference {@code x}
91       * @updates t
92       * @requires IS_BST(t) and x is not in labels(t)
93       * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
94       */
95      private static <T extends Comparable<T>> void insertInTree(BinaryTree<T> t,
96              T x) {
97          assert t != null : "Violation of: t is not null";
98          assert x != null : "Violation of: x is not null";
99          // If the tree is empty, insert the element as the root
100         if (t.size() == 0) {
101             t.assemble(x, t.newInstance(), t.newInstance());
102         } else {
103             BinaryTree<T> left = t.newInstance();
104             BinaryTree<T> right = t.newInstance();
105             T root = t.root();
106             t.disassemble(left, right);
107             // Compare the element to be inserted with the root
108             int compareResult = x.compareTo(root);
109             // Insert into the appropriate subtree
110             if (compareResult < 0) {
111                 insertInTree(left, x);
112             } else {
113                 insertInTree(right, x);
114             }
```

```
115             // Reassemble the tree with the updated subtree
116
117             t.assemble(root, left, right);
118         }
119     }
120
121     /**
122      * Removes and returns the smallest (left-most) label in {@code t}.
123      *
124      * @param <T>
125      *            type of {@code BinaryTree} labels
126      * @param t
127      *            the {@code BinaryTree} from which to remove the label
128      * @return the smallest label in the given {@code BinaryTree}
129      * @updates t
130      * @requires IS_BST(t) and |t| > 0
131      * @ensures <pre>
132      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]  and
133      *  labels(t) = labels(#t) \ {removeSmallest}
134      * </pre>
135      */
136     private static <T> T removeSmallest(BinaryTree<T> t) {
137         assert t != null : "Violation of: t is not null";
138         assert t.size() > 0 : "Violation of: |t| > 0";
139
140         T smallest;
141         BinaryTree<T> left = t.newInstance();
142         BinaryTree<T> right = t.newInstance();
143         T root = t.root();
144         t.disassemble(left, right);
145
146         // If the left subtree is empty, the root is the smallest element
147         if (left.size() == 0) {
148             smallest = root;
149             t.transferFrom(right);
150         } else {
151             // Recursively find the smallest element in the left subtree
152             smallest = removeSmallest(left);
153             t.assemble(root, left, right);
154         }
155
156         return smallest;
157     }
158
159     /**
160      * Finds label {@code x} in {@code t}, removes it from {@code t}, and
161      * returns it.
162      *
163      * @param <T>
164      *            type of {@code BinaryTree} labels
165      * @param t
166      *            the {@code BinaryTree} from which to remove label {@code x}
167      * @param x
168      *            the label to be removed
169      * @return the removed label
170      * @updates t
171      * @requires IS_BST(t) and x is in labels(t)
```

```
172      * @ensures <pre>
173      * IS_BST(t)  and  removeFromTree = x  and
174      *  labels(t) = labels(#t) \ {x}
175      * </pre>
176      */
177     private static <T extends Comparable<T>> T removeFromTree(BinaryTree<T> t,
178             T x) {
179         assert t != null : "Violation of: t is not null";
180         assert x != null : "Violation of: x is not null";
181         assert t.size() > 0 : "Violation of: x is in labels(t)";
182 
183         // type of variable to hold removed element
184         T removed;
185         // create new instances of left and right subtrees
186         BinaryTree<T> left = t.newInstance();
187         BinaryTree<T> right = t.newInstance();
188 
189         // get root of tree
190         T root = t.root();
191         t.disassemble(left, right);
192 
193         int compareResult = x.compareTo(root);
194         // if element is equal to the root
195         if (compareResult == 0) {
196             removed = root;
197             // if right subtree is empty, transfer left to the tree.
198             if (right.size() == 0) {
199                 t.transferFrom(left);
200             } else if (left.size() == 0) {
201                 // if left subtree is empty, transfer the right subtree to the tree.
202                 t.transferFrom(right);
203                 // if both are not empty
204             } else {
205                 // remove smallest element from right subtree
206                 T smallestInRight = removeSmallest(right);
207                 t.assemble(smallestInRight, left, right);
208             }
209             // If the element to be removed is less than the root
210         } else if (compareResult < 0) {
211             // recursively remove element from left tree
212             removed = removeFromTree(left, x);
213             // reassemble tree
214             t.assemble(root, left, right);
215         } else {
216             // recursively remove element from right tree.
217             removed = removeFromTree(right, x);
218             t.assemble(root, left, right);
219         }
220 
221         return removed;
222     }
223 
224     /**
225      * Creator of initial representation.
226      */
227     private void createNewRep() {
228         this.tree = new BinaryTree1<>();
```

```
229     }
230
231     /*
232      * Constructors ------------------------------------------------------------
233      */
234
235     /**
236      * No-argument constructor.
237      */
238     public Set3a() {
239         this.createNewRep();
240     }
241
242     /*
243      * Standard methods --------------------------------------------------------
244      */
245
246     @SuppressWarnings("unchecked")
247     @Override
248     public final Set<T> newInstance() {
249         try {
250             return this.getClass().getConstructor().newInstance();
251         } catch (ReflectiveOperationException e) {
252             throw new AssertionError(
253                     "Cannot construct object of type " + this.getClass());
254         }
255     }
256
257     @Override
258     public final void clear() {
259         this.createNewRep();
260     }
261
262     @Override
263     public final void transferFrom(Set<T> source) {
264         assert source != null : "Violation of: source is not null";
265         assert source != this : "Violation of: source is not this";
266         assert source instanceof Set3a<?> : ""
267                 + "Violation of: source is of dynamic type Set3<?>";
268         /*
269          * This cast cannot fail since the assert above would have stopped
270          * execution in that case: source must be of dynamic type Set3a<?>, and
271          * the ? must be T or the call would not have compiled.
272          */
273         Set3a<T> localSource = (Set3a<T>) source;
274         this.tree = localSource.tree;
275         localSource.createNewRep();
276     }
277
278     /*
279      * Kernel methods ----------------------------------------------------------
280      */
281
282     @Override
283     public final void add(T x) {
284         assert x != null : "Violation of: x is not null";
285         assert !this.contains(x) : "Violation of: x is not in this";
```

```
286
287         insertInTree(this.tree, x);
288     }
289
290     @Override
291     public final T remove(T x) {
292         assert x != null : "Violation of: x is not null";
293         assert this.contains(x) : "Violation of: x is in this";
294
295         return removeFromTree(this.tree, x);
296     }
297
298     @Override
299     public final T removeAny() {
300         assert this.size() > 0 : "Violation of: this /= empty_set";
301
302         return removeSmallest(this.tree);
303     }
304
305     @Override
306     public final boolean contains(T x) {
307         assert x != null : "Violation of: x is not null";
308
309         return isInTree(this.tree, x);
310     }
311
312     @Override
313     public final int size() {
314
315         return this.tree.size();
316
317     }
318
319     @Override
320     public final Iterator<T> iterator() {
321         return this.tree.iterator();
322     }
323
324 }
325
```