```java
import java.util.Iterator;
import java.util.Random;

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 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}.
```

```
42      *
43      * @param <T>
44      *            type of {@code BinaryTree} labels
45      * @param t
46      *            the {@code BinaryTree} to be searched
47      * @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             result = removeSmallest(left); // left tree is
    less than root
140             t.assemble(r, left, right);
141         }
142         return result;
143     }
144 
145     /**
146      * Finds label {@code x} in {@code t}, removes it from
    {@code t}, and
147      * returns it.
148      *
149      * @param <T>
150      *            type of {@code BinaryTree} labels
151      * @param t
152      *            the {@code BinaryTree} from which to remove
    label {@code x}
153      * @param x
154      *            the label to be removed
155      * @return the removed label
156      * @updates t
157      * @requires IS_BST(t) and x is in labels(t)
158      * @ensures <pre>
159      * IS_BST(t)  and  removeFromTree = x  and
160      *  labels(t) = labels(#t) \ {x}
161      * </pre>
162      */
163     private static <T extends Comparable<T>> T
    removeFromTree(BinaryTree<T> t,
164             T x) {
165         assert t != null : "Violation of: t is not null";
166         assert x != null : "Violation of: x is not null";
```

```
167         assert t.size() > 0 : "Violation of: x is in
    labels(t)";
168
169         T result = t.root();
170         if (t.size() == 1) {
171             t.clear(); // delete the label
172         } else {
173             BinaryTree<T> left = t.newInstance();
174             BinaryTree<T> right = t.newInstance();
175             T r = t.disassemble(left, right);
176             if (r.compareTo(x) > 0) {
177                 result = removeFromTree(left, x); // x is in
    the left tree
178                 t.assemble(r, left, right);
179             } else if (r.compareTo(x) < 0) {
180                 result = removeFromTree(right, x); // x is in
    the right tree
181                 t.assemble(r, left, right);
182             } else {
183                 // x is the root node
184                 result = t.root();
185                 if (right.size() == 0) {
186                     t.transferFrom(left);
187                 } else {
188                     T newRoot = right.root();
189                     //left tree is less than the new root
190                     t.assemble(newRoot, left, right);
191                 }
192
193             }
194         }
195
196         return result;
197     }
198
199     /**
200      * Creator of initial representation.
201      */
202     private void createNewRep() {
203
204         this.tree = new BinaryTree1<T>();
205
206     }
207
208     /*
209      * Constructors
    ------------------------------------------------------------
```

```
210      */
211
212     /**
213      * No-argument constructor.
214      */
215     public Set3a() {
216
217         this.createNewRep();
218
219     }
220
221     /*
222      * Standard methods
    ---------------------------------------------------------
223      */
224
225     @SuppressWarnings("unchecked")
226     @Override
227     public final Set<T> newInstance() {
228         try {
229             return
    this.getClass().getConstructor().newInstance();
230         } catch (ReflectiveOperationException e) {
231             throw new AssertionError(
232                     "Cannot construct object of type " +
    this.getClass());
233         }
234     }
235
236     @Override
237     public final void clear() {
238         this.createNewRep();
239     }
240
241     @Override
242     public final void transferFrom(Set<T> source) {
243         assert source != null : "Violation of: source is not
    null";
244         assert source != this : "Violation of: source is not
    this";
245         assert source instanceof Set3a<?> : ""
246                 + "Violation of: source is of dynamic type
    Set3<?>";
247         /*
248          * This cast cannot fail since the assert above would
    have stopped
249          * execution in that case: source must be of dynamic
```

```
   type Set3a<?>, and
250          * the ? must be T or the call would not have
   compiled.
251          */
252         Set3a<T> localSource = (Set3a<T>) source;
253         this.tree = localSource.tree;
254         localSource.createNewRep();
255     }
256
257     /*
258      * Kernel methods
   ------------------------------------------------------------
259      */
260
261     @Override
262     public final void add(T x) {
263         assert x != null : "Violation of: x is not null";
264         assert !this.contains(x) : "Violation of: x is not in
   this";
265
266         insertInTree(this.tree, x);
267
268     }
269
270     @Override
271     public final T remove(T x) {
272         assert x != null : "Violation of: x is not null";
273         assert this.contains(x) : "Violation of: x is in
   this";
274
275         return removeFromTree(this.tree, x);
276
277     }
278
279     @Override
280     public final T removeAny() {
281         assert this.size() > 0 : "Violation of: this /=
   empty_set";
282
283         return this.removeAnyHelper(this.tree);
284     }
285
286     /**
287      * Helper method to remove a random element from the
   tree.
288      *
289      * @param tree
```

```
290      *            the binary tree from which to remove an element
291      * @return the value removed
292      */
293     private T removeAnyHelper(BinaryTree<T> tree) {
294         Random rand = new Random();
295         BinaryTree<T> left = tree.newInstance();
296         BinaryTree<T> right = tree.newInstance();
297         T value = tree.disassemble(left, right);
298 
299         int direction = tree.size() == 1 ? -1 : rand.nextInt(3);
300         // 0: left, 1: right, -1 or 2: current node
301         T removedValue;
302 
303         if (direction == 0 && left.size() > 0) {
304             // Recursively remove from left
305             removedValue = this.removeAnyHelper(left);
306             tree.assemble(value, left, right);
307         } else if (direction == 1 && right.size() > 0) {
308             // Recursively remove from right
309             removedValue = this.removeAnyHelper(right);
310             tree.assemble(value, left, right);
311         } else {
312             // Remove current node
313             removedValue = value;
314             if (left.size() > 0 && right.size() > 0) {
315                 // Node has two children, find successor
316                 T successor = removeSmallest(right);
317                 tree.assemble(successor, left, right);
318             } else if (left.size() > 0) {
319                 // Node has only left child
320                 tree.transferFrom(left);
321             } else if (right.size() > 0) {
322                 // Node has only right child
323                 tree.transferFrom(right);
324             } else {
325                 // Node is a leaf
326                 tree.clear();
327             }
328         }
329 
330         return removedValue;
331 
332     }
333 
334     @Override
```

```
335     public final boolean contains(T x) {
336         assert x != null : "Violation of: x is not null";
337
338         return isInTree(this.tree, x);
339     }
340
341     @Override
342     public final int size() {
343
344         return this.tree.size();
345     }
346
347     @Override
348     public final Iterator<T> iterator() {
349         return this.tree.iterator();
350     }
351
352 }
353
```