```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 Shyam Sai Bethina and Yihone Chu
 *
 */
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
```

```
44      * @param t
45      *            the {@code BinaryTree} to be searched
46      * @param x
47      *            the label to be searched for
48      * @return true if t contains x, false otherwise
49      * @requires IS_BST(t)
50      * @ensures isInTree = (x is in labels(t))
51      */
52     private static <T extends Comparable<T>> boolean
   isInTree(BinaryTree<T> t,
53             T x) {
54         assert t != null : "Violation of: t is not null";
55         assert x != null : "Violation of: x is not null";
56 
57         /*
58          * Initializes answer value.
59          */
60         boolean answer = false;
61 
62         /*
63          * Checks if t is not empty
64          */
65         if (t.size() > 0) {
66             /*
67              * If t is not empty, then disassembles the tree and
   sets it to the
68              * correct trees and root variables.
69              */
70             BinaryTree<T> lhs = t.newInstance();
71             BinaryTree<T> rhs = t.newInstance();
72 
73             T root = t.disassemble(lhs, rhs);
74 
75             /*
76              * If the root equals x, then x is in the tree and
   answer is true.
77              * Otherwise, if x is greater than the root, the
   statement
78              * recursively checks if x is in the right tree, or if
   x is less
79              * than the root, the statement recursively checks if
   x is in the
80              * left tree.
81              */
```

```
 82            if (x.equals(root)) {
 83                answer = true;
 84            } else if (x.compareTo(root) > 0) {
 85                answer = isInTree(rhs, x);
 86            } else {
 87                answer = isInTree(lhs, x);
 88            }
 89
 90            /*
 91             * Reassembles the original tree to preserve it.
 92             */
 93            t.assemble(root, lhs, rhs);
 94        }
 95        return answer;
 96    }
 97
 98    /**
 99     * Inserts {@code x} in {@code t}.
100     *
101     * @param <T>
102     *            type of {@code BinaryTree} labels
103     * @param t
104     *            the {@code BinaryTree} to be searched
105     * @param x
106     *            the label to be inserted
107     * @aliases reference {@code x}
108     * @updates t
109     * @requires IS_BST(t) and x is not in labels(t)
110     * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
111     */
112    private static <T extends Comparable<T>> void
    insertInTree(BinaryTree<T> t,
113            T x) {
114        assert t != null : "Violation of: t is not null";
115        assert x != null : "Violation of: x is not null";
116
117        /*
118         * If the tree is not empty, then it inserts x in the
    correct spot. If
119         * the tree is empty, then a new tree is constructed with
    x being the
120         * root and empty left and right trees.
121         */
122        if (t.size() > 0) {
```

```
123             /*
124              * If t is not empty, then disassembles the tree and
    sets it to the
125              * correct trees and root variables.
126              */
127             BinaryTree<T> lhs = t.newInstance();
128             BinaryTree<T> rhs = t.newInstance();
129 
130             T root = t.disassemble(lhs, rhs);
131 
132             /*
133              * If the root equals x, then x is in the tree and
    answer is true.
134              * Otherwise, if x is greater than the root, the
    statement
135              * recursively adds in x in the right tree, or if x is
    less than the
136              * root, the statement recursively adds in x in the
    left tree.
137              */
138             if (x.compareTo(root) > 0) {
139                 insertInTree(rhs, x);
140             } else {
141                 insertInTree(lhs, x);
142             }
143 
144             /*
145              * Reassembles the original tree to preserve it.
146              */
147             t.assemble(root, lhs, rhs);
148         } else {
149             t.assemble(x, t.newInstance(), t.newInstance());
150         }
151 
152     }
153 
154     /**
155      * Removes and returns the smallest (left-most) label in
    {@code t}.
156      *
157      * @param <T>
158      *            type of {@code BinaryTree} labels
159      * @param t
160      *            the {@code BinaryTree} from which to remove the
```

```
    label
161      * @return the smallest label in the given {@code BinaryTree}
162      * @updates t
163      * @requires IS_BST(t) and |t| > 0
164      * @ensures <pre>
165      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]
    and
166      *  labels(t) = labels(#t) \ {removeSmallest}
167      * </pre>
168      */
169     private static <T> T removeSmallest(BinaryTree<T> t) {
170         assert t != null : "Violation of: t is not null";
171         assert t.size() > 0 : "Violation of: |t| > 0";
172 
173         /*
174          * Initializes answer as the root.
175          */
176         T answer = t.root();
177 
178         if (t.size() > 0) {
179             /*
180              * If t is not empty, then disassembles the tree and
    sets it to the
181              * correct trees and root variables.
182              */
183             BinaryTree<T> lhs = t.newInstance();
184             BinaryTree<T> rhs = t.newInstance();
185 
186             T root = t.disassemble(lhs, rhs);
187 
188             /*
189              * Since smaller values than the root belong in the
    left tree, we
190              * first check if it is not empty. If it is not, then
    we recursively
191              * get the smallest value in the left tree and
    reassemble the tree.
192              * The variable answer becomes the smallest value.
193              */
194             if (lhs.size() != 0) {
195                 answer = removeSmallest(lhs);
196                 t.assemble(root, lhs, rhs);
197             } else {
198                 /*
```

```
199                  * If a left tree doesn't exist, then the root
    becomes the right
200                  * tree, and the answer stays as the root.
201                  */
202                 t.transferFrom(rhs);
203             }
204         }
205         return answer;
206 
207     }
208 
209     /**
210      * Finds label {@code x} in {@code t}, removes it from {@code
    t}, and
211      * returns it.
212      *
213      * @param <T>
214      *            type of {@code BinaryTree} labels
215      * @param t
216      *            the {@code BinaryTree} from which to remove
    label {@code x}
217      * @param x
218      *            the label to be removed
219      * @return the removed label
220      * @updates t
221      * @requires IS_BST(t) and x is in labels(t)
222      * @ensures <pre>
223      * IS_BST(t)  and  removeFromTree = x  and
224      *  labels(t) = labels(#t) \ {x}
225      * </pre>
226      */
227     private static <T extends Comparable<T>> T
    removeFromTree(BinaryTree<T> t,
228             T x) {
229         assert t != null : "Violation of: t is not null";
230         assert x != null : "Violation of: x is not null";
231         assert t.size() > 0 : "Violation of: x is in labels(t)";
232 
233         /*
234          * The removed value is initialized as the root value.
235          */
236         T removed = t.root();
237 
238         if (t.size() > 0) {
```

```
239             /*
240              * If t is not empty, then disassembles the tree and
    sets it to the
241              * correct trees and root variables.
242              */
243             BinaryTree<T> lhs = t.newInstance();
244             BinaryTree<T> rhs = t.newInstance();
245 
246             T root = t.disassemble(lhs, rhs);
247 
248             if (x.compareTo(root) == 0 && rhs.size() != 0) {
249                 /*
250                  * If the removed value is the root, then the tree
    is
251                  * reassembled with the smallest value of the
    right hand tree
252                  * becoming the new root, the old left tree being
    the new left
253                  * tree, and the remaining right hand tree
    becoming the new
254                  * right hand tree.
255                  */
256                 t.assemble(removeSmallest(rhs), lhs, rhs);
257             } else if (x.compareTo(root) == 0 && rhs.size() == 0)
    {
258                 /*
259                  * If the removed value is the root and the right
    hand tree is
260                  * empty, then the left hand tree becomes the new
    tree, with the
261                  * root of the left hand tree becoming the root of
    the entire
262                  * tree.
263                  */
264                 t.transferFrom(lhs);
265             } else if (x.compareTo(root) > 0) {
266                 /*
267                  * If the removed value is greater than the root,
    then we
268                  * recursively remove the value from the right
    hand tree.
269                  */
270                 removed = removeFromTree(rhs, x);
271             } else {
```

```
272                 /*
273                  * If the removed value is less than the root, then we
274                  * recursively remove the value from the right hand tree.
275                  */
276                 removed = removeFromTree(lhs, x);
277             }
278             /*
279              *
280              * Reassembles the tree without the removed value.
281              */
282
283             t.assemble(root, lhs, rhs);
284         }
285
286         return removed;
287     }
288
289     /**
290      * Creator of initial representation.
291      */
292     private void createNewRep() {
293
294         /*
295          * Creates a representation which is a binary tree.
296          */
297         this.tree = new BinaryTree1<T>();
298
299     }
300
301     /*
302      * Constructors
    -----------------------------------------------------------
303      */
304
305     /**
306      * No-argument constructor.
307      */
308     public Set3a() {
309
310         this.createNewRep();
311
312     }
```

```
313
314     /*
315      * Standard methods
    ------------------------------------------------------------
316      */
317
318     @SuppressWarnings("unchecked")
319     @Override
320     public final Set<T> newInstance() {
321         try {
322             return this.getClass().getConstructor().newInstance();
323         } catch (ReflectiveOperationException e) {
324             throw new AssertionError(
325                     "Cannot construct object of type " +
    this.getClass());
326         }
327     }
328
329     @Override
330     public final void clear() {
331         this.createNewRep();
332     }
333
334     @Override
335     public final void transferFrom(Set<T> source) {
336         assert source != null : "Violation of: source is not
    null";
337         assert source != this : "Violation of: source is not
    this";
338         assert source instanceof Set3a<?> : ""
339                 + "Violation of: source is of dynamic type Set3<?
    >";
340         /*
341          * This cast cannot fail since the assert above would have
    stopped
342          * execution in that case: source must be of dynamic type
    Set3a<?>, and
343          * the ? must be T or the call would not have compiled.
344          */
345         Set3a<T> localSource = (Set3a<T>) source;
346         this.tree = localSource.tree;
347         localSource.createNewRep();
348     }
349
```

```
350     /*
351      * Kernel methods
    ------------------------------------------------------------
352      */
353
354     @Override
355     public final void add(T x) {
356         assert x != null : "Violation of: x is not null";
357         assert !this.contains(x) : "Violation of: x is not in
    this";
358
359         /*
360          * Uses insertInTree to insert x into the right spot in
    the binary tree.
361          */
362         insertInTree(this.tree, x);
363
364     }
365
366     @Override
367     public final T remove(T x) {
368         assert x != null : "Violation of: x is not null";
369         assert this.contains(x) : "Violation of: x is in this";
370
371         /*
372          * Removes x from the tree using removeFromTree and
    returning the
373          * removed value.
374          */
375         return removeFromTree(this.tree, x);
376     }
377
378     @Override
379     public final T removeAny() {
380         assert this.size() > 0 : "Violation of: this /=
    empty_set";
381
382         /*
383          * Removes and returns the smallest value from the tree.
384          */
385         return removeSmallest(this.tree);
386     }
387
388     @Override
```

```
389     public final boolean contains(T x) {
390         assert x != null : "Violation of: x is not null";
391 
392         /*
393          * Returns the boolean value returned when calling
    isInTree with the
394          * tree variable and x.
395          */
396         return isInTree(this.tree, x);
397     }
398 
399     @Override
400     public final int size() {
401         /*
402          * Returns the size value returned from size method from
    kernel class.
403          */
404         return this.tree.size();
405     }
406 
407     @Override
408     public final Iterator<T> iterator() {
409         /*
410          * Returns the iterator of the tree representation.
411          */
412         return this.tree.iterator();
413     }
414 
415 }
416 
```