```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 Qinuo Shi & Yiming Cheng
 *
 */
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";

        // TODO - fill in body
```

```
58          boolean b = false;
59          for (T a : t) {
60              if (a.equals(x)) {
61                  b = true;
62              }
63          }
64          return b;
65      }
66
67      /**
68       * Inserts {@code x} in {@code t}.
69       *
70       * @param <T>
71       *            type of {@code BinaryTree} labels
72       * @param t
73       *            the {@code BinaryTree} to be searched
74       * @param x
75       *            the label to be inserted
76       * @aliases reference {@code x}
77       * @updates t
78       * @requires IS_BST(t) and x is not in labels(t)
79       * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
80       */
81      private static <T extends Comparable<T>> void insertInTree(BinaryTree<T> t,
82              T x) {
83          assert t != null : "Violation of: t is not null";
84          assert x != null : "Violation of: x is not null";
85
86          // TODO - fill in body
87          BinaryTree<T> left = t.newInstance();
88          BinaryTree<T> right = t.newInstance();
89          if (t.size() > 0) {
90              T content = t.disassemble(left, right);
91              if (x.compareTo(content) > 0) {
92                  insertInTree(right, x);
93              } else {
94                  insertInTree(left, x);
95              }
96              t.assemble(content, left, right);
97          } else {
98              t.assemble(x, left, right);
99          }
100
101     }
102
103     /**
104      * Removes and returns the smallest (left-most) label in {@code t}.
105      *
106      * @param <T>
107      *            type of {@code BinaryTree} labels
108      * @param t
109      *            the {@code BinaryTree} from which to remove the label
110      * @return the smallest label in the given {@code BinaryTree}
111      * @updates t
112      * @requires IS_BST(t) and |t| > 0
113      * @ensures <pre>
114      * IS_BST(t)  and  removeSmallest = [the smallest label in #t]  and
```

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

```
172             } else {
173                 t.transferFrom(left);
174             }
175         } else if (x.compareTo(content) < 0) {
176             removeContent = removeFromTree(left, x);
177             t.assemble(content, left, right);
178         } else if (x.compareTo(content) > 0) {
179             removeContent = removeFromTree(right, x);
180             t.assemble(content, left, right);
181         }
182 
183         return removeContent;
184     }
185 
186     /**
187      * Creator of initial representation.
188      */
189     private void createNewRep() {
190 
191         // TODO - fill in body
192         this.tree = new BinaryTree1<T>();
193 
194     }
195 
196     /*
197      * Constructors -----------------------------------------------------------
198      */
199 
200     /**
201      * No-argument constructor.
202      */
203     public Set3a() {
204 
205         // TODO - fill in body
206         this.createNewRep();
207 
208     }
209 
210     /*
211      * Standard methods -------------------------------------------------------
212      */
213 
214     @SuppressWarnings("unchecked")
215     @Override
216     public final Set<T> newInstance() {
217         try {
218             return this.getClass().getConstructor().newInstance();
219         } catch (ReflectiveOperationException e) {
220             throw new AssertionError(
221                     "Cannot construct object of type " + this.getClass());
222         }
223     }
224 
225     @Override
226     public final void clear() {
227         this.createNewRep();
228     }
```

```
229
230     @Override
231     public final void transferFrom(Set<T> source) {
232         assert source != null : "Violation of: source is not null";
233         assert source != this : "Violation of: source is not this";
234         assert source instanceof Set3a<?> : ""
235                 + "Violation of: source is of dynamic type Set3<?>";
236         /*
237          * This cast cannot fail since the assert above would have stopped
238          * execution in that case: source must be of dynamic type Set3a<?>, and
239          * the ? must be T or the call would not have compiled.
240          */
241         Set3a<T> localSource = (Set3a<T>) source;
242         this.tree = localSource.tree;
243         localSource.createNewRep();
244     }
245
246     /*
247      * Kernel methods ------------------------------------------------------
248      */
249
250     @Override
251     public final void add(T x) {
252         assert x != null : "Violation of: x is not null";
253         assert !this.contains(x) : "Violation of: x is not in this";
254
255         // TODO - fill in body
256         insertInTree(this.tree, x);
257
258     }
259
260     @Override
261     public final T remove(T x) {
262         assert x != null : "Violation of: x is not null";
263         assert this.contains(x) : "Violation of: x is in this";
264
265         // TODO - fill in body
266         return removeFromTree(this.tree, x);
267
268     }
269
270     @Override
271     public final T removeAny() {
272         assert this.size() > 0 : "Violation of: this /= empty_set";
273
274         // TODO - fill in body
275         return removeSmallest(this.tree);
276
277     }
278
279     @Override
280     public final boolean contains(T x) {
281         assert x != null : "Violation of: x is not null";
282
283         // TODO - fill in body
284         return isInTree(this.tree, x);
285
```

```
286     }
287
288     @Override
289     public final int size() {
290
291         // TODO - fill in body
292         return this.tree.size();
293
294     }
295
296     @Override
297     public final Iterator<T> iterator() {
298         return this.tree.iterator();
299     }
300
301 }
302
```