The Ohio State University

PROJECT 4: SET ON BINARY SEARCH TREES

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

September 29, 2023

```
 1 import java.util.Iterator;
 2
 3 import components.binarytree.BinaryTree;
 4 import components.binarytree.BinaryTree1;
 5 import components.set.Set;
 6 import components.set.SetSecondary;
 7
 8 /**
 9  * {@code Set} represented as a {@code BinaryTree} (maintained as a binary
10  * search tree) of elements with implementations of primary methods.
11  *
12  * @param <T>
13  *            type of {@code Set} elements
14  * @mathdefinitions <pre>
15  * IS_BST(
16  *   tree: binary tree of T
17  *  ): boolean satisfies
18  *  [tree satisfies the binary search tree properties as described in the
19  *   slides with the ordering reported by compareTo for T, including that
20  *   it has no duplicate labels]
21  * </pre>
22  * @convention IS_BST($this.tree)
23  * @correspondence this = labels($this.tree)
24  *
25  * @author Daniil Gofman, Ansh Pachauri
26  *
27  */
28 public class Set3a<T extends Comparable<T>> extends SetSecondary<T> {
29
30     /*
31      * Private members
    ----------------------------------------------------------
32      */
33
34     /**
35      * Elements included in {@code this}.
36      */
37     private BinaryTree<T> tree;
38
39     /**
40      * Returns whether {@code x} is in {@code t}.
41      *
42      * @param <T>
43      *            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         // initialize variables
57         boolean result = false;
58         // check if tree is empty
59         if (t.size() > 0) {
60             BinaryTree<T> left = t.newInstance();
61             BinaryTree<T> right = t.newInstance();
62             T root = t.disassemble(left, right);
63             // compare x with the root and check for x in the
64             // appropriate node
65             if (root.equals(x)) {
66                 result = true;
67             } else if (root.compareTo(x) < 0) {
68                 result = isInTree(right, x);
69             } else {
70                 result = isInTree(left, x);
71             }
72             // reassemble tree
73             t.assemble(root, left, right);
74         }
75         // return if x is in the tree
76         return result;
77     }
78
79     /**
80      * Inserts {@code x} in {@code t}.
81      *
82      * @param <T>
83      *            type of {@code BinaryTree} labels
84      * @param t
85      *            the {@code BinaryTree} to be searched
86      * @param x
87      *            the label to be inserted
88      * @aliases reference {@code x}
89      * @updates t
90      * @requires IS_BST(t) and x is not in labels(t)
91      * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
92      */
93     private static <T extends Comparable<T>> void insertInTree
   (BinaryTree<T> t,
94             T x) {
95         assert t != null : "Violation of: t is not null";
96         assert x != null : "Violation of: x is not null";
97         // Initialize variables
```

```
 98         BinaryTree<T> left = t.newInstance();
 99         BinaryTree<T> right = t.newInstance();
100         // Check if the tree is empty
101         if (t.size() > 0) {
102             T root = t.disassemble(left, right);
103             // Check if the root is greater than,
104             // Or less than x and add to the appropriate node.
105             if (root.compareTo(x) > 0) {
106                 insertInTree(left, x);
107             } else {
108                 insertInTree(right, x);
109             }
110             // Reassemble the tree
111             t.assemble(root, left, right);
112             // If the tree is empty
113         } else {
114             // Reassemble the tree with x as the root
115             t.assemble(x, left, right);
116         }
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         // Initialize variables to store left and right children
140         BinaryTree<T> left = t.newInstance();
141         BinaryTree<T> right = t.newInstance();
142         // Get root of the tree
143         T root = t.disassemble(left, right);
144         T smallest = root;
145         if (left.size() > 0) {
146             smallest = removeSmallest(left);
147             t.assemble(root, left, right);
```

```
148         } else {
149             t.transferFrom(right);
150         }
151         // Return the smallest element
152         return smallest;
153     }
154
155     /**
156      * Finds label {@code x} in {@code t}, removes it from {@code t}, and
157      * returns it.
158      *
159      * @param <T>
160      *            type of {@code BinaryTree} labels
161      * @param t
162      *            the {@code BinaryTree} from which to remove label {@code
   x}
163      * @param x
164      *            the label to be removed
165      * @return the removed label
166      * @updates t
167      * @requires IS_BST(t) and x is in labels(t)
168      * @ensures <pre>
169      * IS_BST(t)  and  removeFromTree = x  and
170      *  labels(t) = labels(#t) \ {x}
171      * </pre>
172      */
173     private static <T extends Comparable<T>> T removeFromTree(BinaryTree<T>
   t,
174             T x) {
175         assert t != null : "Violation of: t is not null";
176         assert x != null : "Violation of: x is not null";
177         assert t.size() > 0 : "Violation of: x is in labels(t)";
178
179         // Create new binary trees for the left and right subtrees.
180         BinaryTree<T> right = t.newInstance();
181         BinaryTree<T> left = t.newInstance();
182
183         // Disassemble the original tree 't' and get the root element.
184         T root = t.disassemble(left, right);
185
186         // Create a variable to store the element that will be removed.
187         T remove = root;
188
189         // Check if the root element is equal to the element 'x'.
190         if (root.equals(x)) {
191             // If 'x' is found at the root, handle the removal case.
192             // If the right subtree is not empty, find the smallest element
193             // in the right subtree and set it as the new root.
194             if (right.size() > 0) {
195                 root = removeSmallest(right);
```

```
196                 t.assemble(root, left, right);
197             } else {
198                 // If the right subtree is empty, transfer the left subtree
    to 't'.
199                 t.transferFrom(left);
200             }
201         } else {
202             // If 'x' is not equal to the root element, recursively search
    for 'x'
203             // in either the left or right subtree based on the comparison.
204 
205             if (root.compareTo(x) < 0) {
206                 // If 'x' is greater than the root element, search in the
    right subtree.
207                 remove = removeFromTree(right, x);
208             } else {
209                 // If 'x' is less than the root element, search in the left
    subtree.
210                 remove = removeFromTree(left, x);
211             }
212 
213             // Assemble the tree after the recursive call.
214             t.assemble(root, left, right);
215         }
216 
217         // Return the element that was removed.
218         return remove;
219     }
220 
221     /**
222      * Creator of initial representation.
223      */
224     private void createNewRep() {
225 
226         // Create new representation
227         this.tree = new BinaryTree1<T>();
228     }
229 
230     /*
231      * Constructors
    -----------------------------------------------------------
232      */
233 
234     /**
235      * No-argument constructor.
236      */
237     public Set3a() {
238 
239         // Default constructor
240         this.createNewRep();
```

```
241     }
242
243     /*
244      * Standard methods
    -------------------------------------------------------
245      */
246
247     @SuppressWarnings("unchecked")
248     @Override
249     public final Set<T> newInstance() {
250         try {
251             return this.getClass().getConstructor().newInstance();
252         } catch (ReflectiveOperationException e) {
253             throw new AssertionError(
254                     "Cannot construct object of type " + this.getClass());
255         }
256     }
257
258     @Override
259     public final void clear() {
260         this.createNewRep();
261     }
262
263     @Override
264     public final void transferFrom(Set<T> source) {
265         assert source != null : "Violation of: source is not null";
266         assert source != this : "Violation of: source is not this";
267         assert source instanceof Set3a<?> : ""
268                 + "Violation of: source is of dynamic type Set3<?>";
269         /*
270          * This cast cannot fail since the assert above would have stopped
271          * execution in that case: source must be of dynamic type Set3a<?>,
    and
272          * the ? must be T or the call would not have compiled.
273          */
274         Set3a<T> localSource = (Set3a<T>) source;
275         this.tree = localSource.tree;
276         localSource.createNewRep();
277     }
278
279     /*
280      * Kernel methods
    -------------------------------------------------------
281      */
282
283     @Override
284     public final void add(T x) {
285         assert x != null : "Violation of: x is not null";
286         assert !this.contains(x) : "Violation of: x is not in this";
287
```

```
288         // Inserts element x into a BST
289         insertInTree(this.tree, x);
290     }
291 
292     @Override
293     public final T remove(T x) {
294         assert x != null : "Violation of: x is not null";
295         assert this.contains(x) : "Violation of: x is in this";
296 
297         // Remove and return element from the tree
298         return removeFromTree(this.tree, x);
299     }
300 
301     @Override
302     public final T removeAny() {
303         assert this.size() > 0 : "Violation of: this /= empty_set";
304         // Removes the smallest term from the tree
305         return removeSmallest(this.tree);
306     }
307 
308     @Override
309     public final boolean contains(T x) {
310         assert x != null : "Violation of: x is not null";
311         // checks if x is in the tree
312         return isInTree(this.tree, x);
313     }
314 
315     @Override
316     public final int size() {
317         // return the size of the tree
318         return this.tree.size();
319     }
320 
321     @Override
322     public final Iterator<T> iterator() {
323         return this.tree.iterator();
324     }
325 
326 }
327 
```