```java
import java.util.Comparator;

public class BSTree<E>
{
    class Node
    {
        E data;
        Node left;
        Node right;
        Node parent;

        Node (E d)
        {
            this.data = d;
            this.left = null;
            this.right = null;
            this.parent = null;
        }
    }

    private Node root;
    private Comparator<E> comp;

    public BSTree(Comparator<E> theComp)
    {
        root = null;
        comp = theComp;
    }

    public Node getRoot()
    {
        return root;
    }

    /**
     * The item is added to the tree.
     * @param item item that is added to the tree.
     */
    public void addLoop(E item)
    {
        Node addNode = new Node(item);
        if(this.isEmpty()) {
            root = addNode;
```

```
45              root.parent = null;
46          }
47          else {
48              E currData = null;
49              Node curr = root;
50              while(curr != null && curr.data != item) { //no
    duplicates
51                  currData = curr.data;
52                  if(comp.compare(currData, item) > 0) {
53                      if(curr.left == null) {
54                          curr.left = addNode;
55                          addNode.parent = curr;
56                      }
57                      else {
58                          curr = curr.left;
59                      }
60                  }
61                  if(comp.compare(currData, item) < 0) {
62                      if(curr.right == null) {
63                          curr.right = addNode;
64                          addNode.parent = curr;
65
66                      }
67                      else {
68                          curr = curr.right;
69                      }
70                  }
71              }
72          }
73      }
74
75      /**
76       * Checks if the tree is empty.
77       * @return If the tree is empty.
78       */
79      public boolean isEmpty()
80      {
81          if(root == null) {
82              return true;
83          }
84          else {
85              return false;
86          }
```

```
 87     }
 88
 89     /**
 90      * The largest value in the tree is returned.
 91      * @return The largest value in the tree.
 92      */
 93     public E maxValueLoop()
 94     {
 95         return findMaxNodeLoop(root).data;
 96     }
 97
 98     /**
 99      * The node with the highest value in the sub-tree rooted
    (starting) at the current node is returned.
100      * @param curr Node to start from.
101      * @return The node with the highest value in the sub-tree
    rooted (starting) at the current node.
102      */
103     public Node findMaxNodeLoop(Node curr)
104     {
105         if(this.isEmpty()) {
106             throw new NoSuchElementException();
107         }
108         while(curr.right!= null) {
109             curr = curr.right;
110         }
111         return curr;
112     }
113
114     /**
115      * The smallest value in the tree is returned.
116      * @return The smallest value in the tree.
117      */
118     public E minValueLoop()
119     {
120         return findMinNodeLoop(root).data;
121     }
122
123     /**
124      * The node with the smallest value in the sub-tree rooted
    (starting) at the current node is returned.
125      * @param curr Node to start from.
126      * @return The node with the smallest value in the sub-tree
```

```
    rooted (starting) at the current node.
127      */
128     public Node findMinNodeLoop(Node curr)
129     {
130         if(this.isEmpty()) {
131             throw new NoSuchElementException();
132         }
133         while(curr.left!= null) {
134             curr = curr.left;
135         }
136         return curr;
137     }
138 
139     /**
140      * Checks if the given item is found in the tree.
141      * @param item Item to search for in the tree.
142      * @return If the given item is found in the tree.
143      */
144     public boolean containsLoop(E item)
145     {
146         if(this.isEmpty()) {
147             return false;
148         }
149         else if (findNodeLoop(root, item) != null) {
150             return true;
151         }
152         else {
153             return false;
154         }
155     }
156 
157     /**
158      * The Node that contains the item given is returned. Looked
    for in the sub-tree rooted (starting) at the current node.
159      * @param curr Node to start from.
160      * @param item Item to search for in the tree of Nodes.
161      * @return The Node that contains the item given is returned.
162      */
163     public Node findNodeLoop(Node curr, E item)
164     {
165         while(curr != null) {
166             E currData = curr.data;
167             if(comp.compare(currData, item) == 0) {
```

```
168                 return curr;
169             }
170             else if(comp.compare(currData, item) > 0 ) {
171                 curr = curr.left;
172             }
173             else if(comp.compare(currData, item) < 0 ) {
174                 curr = curr.right;
175             }
176         }
177         return curr;
178     }
179 
180     /**
181      * The item given is added to the tree.
182      * @param item Item to add accordingly in the tree.
183      */
184     public void add(E item)
185     {
186         add(root, item);
187     }
188 
189     /**
190      * Using Recursion, adds the given item to the sub-tree rooted (starting) at the given curr node.
191      * @param curr Node to start from.
192      * @param item Item to add accordingly to the tree.
193      */
194     public void add(Node curr, E item)
195     {
196         Node addNode = new Node(item);
197         if(this.isEmpty()) {
198             root = addNode;
199             root.parent = null;
200         }
201         else {
202             E currData = null;
203             if(curr != null && curr.data != item) { //no duplicates said in class
204                 currData = curr.data;
205                 if(comp.compare(currData, item) > 0) {
206                     if(curr.left == null) {
207                         curr.left = addNode;
208                         addNode.parent = curr;
```

```
209                     }
210                     else {
211                         add(curr.left, item);
212                     }
213                 }
214                 if(comp.compare(currData, item) < 0) {
215                     if(curr.right == null) {
216                         curr.right = addNode;
217                         addNode.parent = curr;
218 
219                     }
220                     else {
221                         add(curr.right, item);
222                     }
223                 }
224             }
225         }
226     }
227 
228     /**
229      * The largest value in the tree is returned.
230      * @return The largest value in the tree.
231      */
232     public E maxValue()
233     {
234         return findMaxNode(root).data;
235     }
236 
237     /**
238      * Using recursion, the highest value in the sub-tree rooted
    (starting) at the current node is returned.
239      * @param curr Node to start from.
240      * @return The highest value in the sub-tree rooted (starting)
    at the current node.
241      */
242     public Node findMaxNode(Node curr)
243     {
244         if(this.isEmpty()) {
245             throw new NoSuchElementException();
246         }
247         if(curr.right != null) {
248             return findMaxNode(curr.right);
249         }
```

```
250         return curr;
251     }
252 
253     /**
254      * The smallest value in the tree is returned.
255      * @return The smallest value in the tree.
256      */
257     public E minValue()
258     {
259         return findMinNode(root).data;
260     }
261 
262     /**
263      * Using recursion, the smallest value in the sub-tree rooted (starting) at the current node is returned.
264      * @param curr Node to start from.
265      * @return The smallest value in the sub-tree rooted (starting) at the current node.
266      */
267     public Node findMinNode(Node curr)
268     {
269         if(this.isEmpty()) {
270             throw new NoSuchElementException();
271         }
272         if(curr.left != null) {
273             return findMinNode(curr.left);
274         }
275         return curr;
276     }
277 
278     /**
279      * Checks if the given item is found within the tree.
280      * @param item Item to search for in the tree.
281      * @return If the given item is found within the tree.
282      */
283     public boolean contains(E item)
284     {
285         if(this.isEmpty()) {
286             return false;
287         }
288         else if (findNode(root, item) != null) {
289             return true;
290         }
```

```
291         else {
292             return false;
293         }
294     }
295 
296     /**
297      * Using recursion, the node that contains the item given is returned. Looked for in the sub-tree rooted (starting) at the current node.
298      * @param curr Node to start from.
299      * @param item Item to search for stored in a Node found in the tree.
300      * @return The node that contains the item given is returned.
301      */
302     public Node findNode(Node curr, E item)
303     {
304         if(curr != null) {
305             E currData = curr.data;
306             if(comp.compare(currData, item) == 0) {
307                 return curr;
308             }
309             else if(comp.compare(currData, item) > 0 ) {
310                 return findNode(curr.left, item);
311             }
312             else if(comp.compare(currData, item) < 0 ) {
313                 return findNode(curr.right, item);
314             }
315         }
316         return curr;
317     }
318 
319     /**
320      * The given item is removed from the tree.
321      * @param item Item to search and remove from the tree.
322      * @return If the given item is removed from the tree.
323      */
324     public boolean remove(E item)
325     {
326         Node curr = findNode(root, item);
327         if(curr != null) {
328             if(curr.left != null && curr.right != null) {
329                 removeHasBoth(curr);
330             }
```

```
331             else if (curr.left == null || curr.right == null) {
332                 removeMissing(curr);
333             }
334             return true;
335         }
336         return false;
337     }
338 
339     /**
340      * Removes the given node even if it is missing one or both children.
341      * @param node Node to remove from the tree.
342      */
343     public void removeMissing(Node node)
344     {
345 
346         if(node == root) {
347             if(node.left == null && node.right == null) { //missing both
348                 root = null;
349             }
350             else if(node.left == null) { //missing left
351                 root = node.right;
352                 node.right.parent = null;
353                 node = null;
354             }
355             else if(node.right == null) { //missing right
356                 root = node.left;
357                 node.left.parent = null;
358                 node = null;
359             }
360         }
361         else if(node.parent.left == node) { // left side of the tree PARENT
362             if(node.left == null && node.right == null) { //missing both
363                 node.parent.left = null;
364                 node = null;
365             }
366             else if(node.left == null) { //missing left
367                 node.parent.left = node.right;
368                 node.right.parent = node.parent;
369                 node = null;
```

```
370             }
371             else if(node.right == null) { //missing right
372                 node.parent.left = node.left;
373                 node.left.parent = node.parent;
374                 node = null;
375             }
376         }
377         else if(node.parent.right == node) { // right side of the tree PARENT
378             if(node.left == null && node.right == null) { //missing both
379                 node.parent.right = null;
380                 node = null;
381             }
382             else if(node.left == null) {//missing left
383                 node.parent.right = node.right;
384                 node.right.parent = node.parent;
385                 node = null;
386             }
387             else if(node.right == null) {   //missing right
388                 node.parent.right = node.left;
389                 node.left.parent = node.parent;
390                 node = null;
391             }
392         }
393     }
394
395     /**
396      * Removes the node while assuming it has two children (exact).
397      * @param node Node to remove from the tree.
398      */
399     public void removeHasBoth(Node node)
400     {
401         E store = null;
402         store = findMaxNode(node.left).data;
403         node.data = store;
404         removeMissing(findMaxNode(node.left));
405     }
406
407     /**
408      * String representation is returned for the tree.
409      */
410     public String toString()
```

```
411     {
412         return new BSTreeUtils<E>().toString(root);
413     }
414 }
415
```