```java
import java.util.Comparator;
import java.util.NoSuchElementException;

import org.w3c.dom.Node;

/**
 * Implementation of Binary Search Tree data structure
 *
 * @author Andry Arthur
 *
 */
public class BSTree<E>
{
    private class Node
    {
        E data;
        Node left;
        Node right;
        Node parent;

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

    private Node root;

    private Comparator<E> comparator;

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

    /**
     * Returns the root of the tree.
     *
     * @return  the root of the tree
     */
    public Node getNode()
    {
        return root;
    }

    /**
     * Adds the given item to the tree.
     *
     * @param item      item to be added on the tree
     */
    public void addLoop(E item)
    {
        Node nodeNew = new Node(item);
```

```
58          Node curr = null;
59          if(root == null) {
60              root = nodeNew;
61          }
62          else {
63              curr = root;
64              while(curr != nodeNew) {
65                  if(comparator.compare(item, curr.data) < 0) {
66                      if(curr.left == null) {
67                          nodeNew.parent = curr;
68                          curr.left = nodeNew;
69                      }
70                      else {
71                          curr = curr.left;
72                      }
73                  }
74                  else {
75                      if(curr.right == null) {
76                          nodeNew.parent = curr;
77                          curr.right = nodeNew;
78                      }
79                      else {
80                          curr = curr.right;
81                      }
82                  }
83              }
84          }
85          return;
86      }
87
88      /**
89       * Checks if tree is empty.
90       *
91       * @return  true if tree is empty
92       */
93      public boolean isEmpty()
94      {
95          return root == null;
96      }
97
98      //methods that use loops
99
100     /**
101      * Returns the maximum value in the tree using loops.
102      *
103      * @return  the maximum value in the tree
104      * @throws  NoSuchElementException when tree is empty
105      */
106     public E maxValueLoop()
107     {
108         if(isEmpty()) {
109             throw new NoSuchElementException();
110         }
111
112         Node container = findMaxNodeLoop(root);
113
114         return container.data;
```

```
115     }
116
117     /**
118      * Returns the node with the maximum value in subtree using loops.
119      *
120      * @param curr      root of subtree
121      * @return  node with the maximum value
122      */
123     private Node findMaxNodeLoop(Node curr)
124     {
125         Node currMax = curr;
126
127         while(currMax.right != null) {
128             currMax = currMax.right;
129         }
130
131         return currMax;
132     }
133
134     /**
135      * Returns the minimum value using loops
136      *
137      * @return  the minimum value
138      * @throws  NoSuchElementException when tree is empty
139      */
140     public E minValueLoop()
141     {
142         if(isEmpty()) {
143             throw new NoSuchElementException();
144         }
145
146         Node container = findMinNodeLoop(root);
147
148         return container.data;
149     }
150
151     /**
152      * Returns node with the minimum value in substree using loops.
153      *
154      * @param curr      root of subtree
155      * @return  node with minimum value
156      */
157     private Node findMinNodeLoop(Node curr)
158     {
159         Node currMin = curr;
160
161         while(currMin.left != null) {
162             currMin = currMin.left;
163         }
164
165         return currMin;
166     }
167
168     /**
169      * Checks whether tree contains item
170      *
171      * @param item      item to be found in tree
```

```
172      * @return  true if tree contains item
173      */
174     public boolean containsLoop(E item)
175     {
176         if(findNodeLoop(root, item) == null) {
177             return false;
178         }
179         else {
180             return true;
181         }
182     }
183 
184     /**
185      * Returns the Node with the given item within the given subtree.
186      *
187      * @param curr      root of subtree
188      * @param item      item to be found
189      * @return  node with the given item within subtree
190      */
191     private Node findNodeLoop(Node curr, E item)
192     {
193         Node currNode = curr;
194 
195         while(currNode != null) {
196             if(comparator.compare(item, currNode.data) == 0) {
197                 return currNode;
198             }
199             else if(comparator.compare(item, currNode.data) < 0) {
200                 currNode = currNode.left;
201             }
202             else {
203                 currNode = currNode.right;
204             }
205         }
206 
207         return currNode;
208     }
209 
210     //recursive methods
211 
212     /**
213      * Adds item onto the tree.
214      *
215      * @param item      item to be added
216      */
217     public void add(E item)
218     {
219         //root = new BSTreeUtils<E>().add(root, comparator, new Node(item));   // helper
    version;
220         add(root, item);
221     }
222 
223     /**
224      * Adds the given item onto the tree using recursion
225      *
226      * @param curr      root of subtree
227      * @param item      item to be added
```

```
228      */
229     private void add(Node curr, E item)
230     {
231         if(this.isEmpty()) {
232             root = new Node(item);
233         }
234         else if (comparator.compare(curr.data, item) < 0) {
235             if(curr.right == null) {
236             curr.right = new Node(item);
237             curr.right.parent = curr;
238             }
239             else {
240                 add(curr.right, item);
241             }
242         }
243         else {
244             if(curr.left == null) {
245                 curr.left = new Node(item);
246                 curr.left.parent = curr;
247                 }
248                 else {
249                     add(curr.left, item);
250                 }
251         }
252     }
253 
254     /**
255      * Returns the maximum value in the tree using recursion.
256      *
257      * @return  the maximum value in the tree
258      */
259     public E maxValue()
260     {
261         Node max = findMaxNode(root);
262 
263         if(max == null) {
264             throw new NoSuchElementException();
265         }
266 
267         return max.data;
268     }
269 
270     /**
271      * Returns the node with the maximum value in subtree using recursion.
272      *
273      * @param curr      root of subtree
274      * @return  node with the maximum value
275      */
276     private Node findMaxNode(Node curr)
277     {
278         if(curr == null) {
279             return null;
280         }
281         else if(curr.right == null) {
282             return curr;
283         }
284         else {
```

```
285             return findMaxNode(curr.right);
286         }
287     }
288 
289     /**
290      * Returns the minimum value using recursion.
291      *
292      * @return  the minimum value
293      * @throws  NoSuchElementException when the tree is empty
294      */
295     public E minValue()
296     {
297         Node min = findMinNode(root);
298 
299         if(min == null) {
300             throw new NoSuchElementException();
301         }
302 
303         return min.data;
304     }
305 
306     /**
307      * Returns node with the minimum value in substree using recursion.
308      *
309      * @param curr      root of subtree
310      * @return  node with minimum value
311      */
312     private Node findMinNode(Node curr)
313     {
314         if(curr == null) {
315             return null;
316         }
317         else if(curr.left == null) {
318             return curr;
319         }
320         else {
321             return findMinNode(curr.left);
322         }
323     }
324 
325     /**
326      * Checks whether tree contains item
327      *
328      * @param item      item to be found in tree
329      * @return  true if tree contains item
330      */
331     public boolean contains(E item)
332     {
333         return findNode(root, item) != null;
334     }
335 
336     /**
337      * Returns the Node with the given item within the given subtree.
338      *
339      * @param curr      root of subtree
340      * @param item      item to be found
341      * @return  node with the given item within subtree
```

```
342      */
343     private Node findNode(Node curr, E item)
344     {
345         Node nodeFind = null;
346 
347         if(curr == null) {
348             return null;
349         }
350         else if(comparator.compare(curr.data, item) == 0) {
351             nodeFind = curr;
352             return nodeFind;
353         }
354         else {
355             if(comparator.compare(curr.data, item) < 0) {
356                 return findNode(curr.right, item);
357             }
358             else {
359                 return findNode(curr.left, item);
360             }
361         }
362     }
363 
364     /**
365      * Removes given item from the tree.
366      *
367      * @param item      item to be removed
368      * @return  given item from the tree
369      */
370     public boolean remove(E item)
371     {
372         Node nodeToRemove = findNode(root, item);
373         if(nodeToRemove == null) {
374             return false;
375         }
376         else if(nodeToRemove == root) {
377             root = null;
378         }
379         else if(nodeToRemove.left != null && nodeToRemove.right != null) {
380             removeHasBoth(nodeToRemove);
381         }
382         else if(nodeToRemove.left == null || nodeToRemove.right == null) {
383             removeMissing(nodeToRemove);
384         }
385         return true;
386     }
387 
388     /**
389      * Removes given node that is parent to either one or no children.
390      *
391      * @param node      node to be removed
392      */
393     private void removeMissing(Node node)
394     {
395         Node parent = node.parent;
396 
397         if(node.left == null && node.right == null) {
398             if(comparator.compare(parent.left.data, node.data) == 0) {
```

```
399                 parent.left = null;
400             }
401             else {
402                 parent.right = null;
403             }
404         }
405         else if (node.left != null) {
406             if(comparator.compare(parent.left.data, node.data) == 0) {
407                 parent.left = node.left;
408             }
409             else {
410                 parent.right = node.left;
411             }
412             node.left.parent = parent;
413         }
414         else {
415             if(comparator.compare(parent.right.data, node.data) == 0) {
416                 parent.right = node.right;
417             }
418             else {
419                 parent.left = node.right;
420             }
421             node.right.parent = parent;
422         }
423         //node.parent = null;
424         return;
425     }
426 
427     /**
428      * Removes given node that is parent to two other nodes
429      *
430      * @param node      node to be removed
431      */
432     private void removeHasBoth(Node node)
433     {
434         Node parent = node.parent;
435 
436         parent.right = node.right;
437         node.right.parent = parent;
438         node.right.left = node.left;
439         node.left.parent = node.right;
440     }
441 
442     /**
443      * Returns a string version of the tree.
444      *
445      * @return  string version of the tree
446      */
447     public String toString()
448     {
449         return new BSTreeUtils<E>().toString(root);
450     }
451 }
452 
```