```java
package dataStructures;

public class BinarySearchTree<K extends Comparable<K>, V>
implements
        OrderedDictionary<K, V> {

    static final long serialVersionUID = 0L;

    // The root of the tree.
    protected BSTNode<K, V> root;

    // Number of entries in the tree.
    protected int currentSize;

    protected static class PathStep<K, V> {

        // The parent of the node.
        public BSTNode<K, V> parent;

        // The node is the left or the right child of parent.
        public boolean isLeftChild;

        public PathStep(BSTNode<K, V> theParent, boolean toTheLeft)
{
            parent = theParent;
            isLeftChild = toTheLeft;
        }

        public void set(BSTNode<K, V> newParent, boolean toTheLeft)
{
            parent = newParent;
            isLeftChild = toTheLeft;
        }

    }

    public BinarySearchTree() {
        root = null;
        currentSize = 0;
    }

```

```
39      // Returs true iff the dictionary contains no entries.
40      public boolean isEmpty() {
41          return root == null;
42      }
43
44      // Returns the number of entries in the dictionary.
45      public int size() {
46          return currentSize;
47      }
48
49      // If there is an entry in the dictionary whose key is the
    specified key,
50      // returns its value; otherwise, returns null.
51      public V find(K key) {
52          BSTNode<K, V> node = this.findNode(root, key);
53          if (node == null)
54              return null;
55          else
56              return node.getValue();
57      }
58
59      // Returns the node whose key is the specified key;
60      // or null if no such node exists.
61      protected BSTNode<K, V> findNode(BSTNode<K, V> node, K key) {
62          if (node == null)
63              return null;
64          else {
65              int compResult = key.compareTo(node.getKey());
66              if (compResult == 0)
67                  return node;
68              else if (compResult < 0)
69                  return this.findNode(node.getLeft(), key);
70              else
71                  return this.findNode(node.getRight(), key);
72          }
73      }
74
75      // Returns the entry with the smallest key in the dictionary.
76      public Entry<K, V> minEntry() throws EmptyDictionaryException {
77          if (this.isEmpty())
78              throw new EmptyDictionaryException();
```

```
79
80          return this.minNode(root).getEntry();
81      }
82
83      // Returns the node with the smallest key
84      // in the tree rooted at the specified node.
85      // Precondition: node != null.
86      protected BSTNode<K, V> minNode(BSTNode<K, V> node) {
87          if (node.getLeft() == null)
88              return node;
89          else
90              return this.minNode(node.getLeft());
91      }
92
93      // Returns the entry with the largest key in the dictionary.
94      public Entry<K, V> maxEntry() throws EmptyDictionaryException {
95          if (this.isEmpty())
96              throw new EmptyDictionaryException();
97
98          return this.maxNode(root).getEntry();
99      }
100
101     // Returns the node with the largest key
102     // in the tree rooted at the specified node.
103     // Precondition: node != null.
104     protected BSTNode<K, V> maxNode(BSTNode<K, V> node) {
105         if (node.getRight() == null)
106             return node;
107         else
108             return this.maxNode(node.getRight());
109     }
110
111     // Returns the node whose key is the specified key;
112     // or null if no such node exists.
113     // Moreover, stores the last step of the path in lastStep.
114     protected BSTNode<K, V> findNode(K key, PathStep<K, V>
    lastStep) {
115         BSTNode<K, V> node = root;
116         while (node != null) {
117             int compResult = key.compareTo(node.getKey());
118             if (compResult == 0)
```

```
119                 return node;
120             else if (compResult < 0) {
121                 lastStep.set(node, true);
122                 node = node.getLeft();
123             } else {
124                 lastStep.set(node, false);
125                 node = node.getRight();
126             }
127         }
128         return null;
129     }
130 
131     // If there is an entry in the dictionary whose key is the
    specified key,
132     // replaces its value by the specified value and returns the
    old value;
133     // otherwise, inserts the entry (key, value) and returns null.
134     public V insert(K key, V value) {
135         PathStep<K, V> lastStep = new PathStep<K, V>(null, false);
136         BSTNode<K, V> node = this.findNode(key, lastStep);
137         if (node == null) {
138             BSTNode<K, V> newLeaf = new BSTNode<K, V>(key, value);
139             this.linkSubtree(newLeaf, lastStep);
140             currentSize++;
141             return null;
142         } else {
143             V oldValue = node.getValue();
144             node.setValue(value);
145             return oldValue;
146         }
147     }
148 
149     // Links a new subtree, rooted at the specified node, to the
    tree.
150     // The parent of the old subtree is stored in lastStep.
151     protected void linkSubtree(BSTNode<K, V> node, PathStep<K, V>
    lastStep) {
152         if (lastStep.parent == null)
153             // Change the root of the tree.
154             root = node;
155         else
```

```
156         // Change a child of parent.
157         if (lastStep.isLeftChild)
158             lastStep.parent.setLeft(node);
159         else
160             lastStep.parent.setRight(node);
161     }
162 
163     // Returns the node with the smallest key
164     // in the tree rooted at the specified node.
165     // Moreover, stores the last step of the path in lastStep.
166     // Precondition: theRoot != null.
167     protected BSTNode<K, V> minNode(BSTNode<K, V> theRoot,
168             PathStep<K, V> lastStep) {
169         BSTNode<K, V> node = theRoot;
170         while (node.getLeft() != null) {
171             lastStep.set(node, true);
172             node = node.getLeft();
173         }
174         return node;
175     }
176 
177     // If there is an entry in the dictionary whose key is the
    specified key,
178     // removes it from the dictionary and returns its value;
179     // otherwise, returns null.
180     public V remove(K key) {
181         PathStep<K, V> lastStep = new PathStep<K, V>(null, false);
182         BSTNode<K, V> node = this.findNode(key, lastStep);
183         if (node == null)
184             return null;
185         else {
186             V oldValue = node.getValue();
187             if (node.getLeft() == null)
188                 // The left subtree is empty.
189                 this.linkSubtree(node.getRight(), lastStep);
190             else if (node.getRight() == null)
191                 // The right subtree is empty.
192                 this.linkSubtree(node.getLeft(), lastStep);
193             else {
194                 // Node has 2 children. Replace the node's entry
    with
```

```
195                 // the 'minEntry' of the right subtree.
196                 lastStep.set(node, false);
197                 BSTNode<K, V> minNode =
    this.minNode(node.getRight(), lastStep);
198                 node.setEntry(minNode.getEntry());
199                 // Remove the 'minEntry' of the right subtree.
200                 this.linkSubtree(minNode.getRight(), lastStep);
201             }
202             currentSize--;
203             return oldValue;
204         }
205     }
206
207     // Returns an iterator of the entries in the dictionary
208     // which preserves the key order relation.
209     public Iterator<Entry<K, V>> iterator() {
210         return new BSTInorderIterator<K, V>(root);
211     }
212
213 }
214
```