```java
package dataStructures;

public class AVLTree<K extends Comparable<K>, V> extends AdvancedBSTree<K, V> {

    static final long serialVersionUID = 0L;

    // If there is an entry in the dictionary whose key is the specified key,
    // replaces its value by the specified value and returns the old value;
    // otherwise, inserts the entry (key, value) and returns null.
    public V insert(K key, V value) {
        Stack<PathStep<K, V>> path = new StackInList<PathStep<K, V>>();
        BSTNode<K, V> node = this.findNode(key, path);
        if (node == null) {
            AVLNode<K, V> newLeaf = new AVLNode<K, V>(key, value);
            this.linkSubtree(newLeaf, path.top());
            currentSize++;
            this.reorganizeIns(path);
            return null;
        } else {
            V oldValue = node.getValue();
            node.setValue(value);
            return oldValue;
        }
    }

    // Every ancestor of the new leaf is stored in the stack,
    // which is not empty.
    protected void reorganizeIns(Stack<PathStep<K, V>> path) {
        boolean grew = true;
        PathStep<K, V> lastStep = path.pop();
        AVLNode<K, V> parent = (AVLNode<K, V>) lastStep.parent;
        while (grew && parent != null) {
            if (lastStep.isLeftChild)
                // parent's left subtree has grown.
                switch (parent.getBalance()) {
                case 'L':
                    this.rebalanceInsLeft(parent, path);
```

```
38                      grew = false;
39                      break;
40                  case 'E':
41                      parent.setBalance('L');
42                      break;
43                  case 'R':
44                      parent.setBalance('E');
45                      grew = false;
46                      break;
47                  }
48              else
49                  // parent's right subtree has grown.
50                  switch (parent.getBalance()) {
51                  case 'L':
52                      parent.setBalance('E');
53                      grew = false;
54                      break;
55                  case 'E':
56                      parent.setBalance('R');
57                      break;
58                  case 'R':
59                      this.rebalanceInsRight(parent, path);
60                      grew = false;
61                      break;
62                  }
63              lastStep = path.pop();
64              parent = (AVLNode<K, V>) lastStep.parent;
65          }
66      }
67
68      // Every ancestor of node is stored in the stack, which is not
    empty.
69      // height( node.getLeft() ) - height( node.getRight() ) = 2.
70      protected void rebalanceInsLeft(AVLNode<K, V> node,
71              Stack<PathStep<K, V>> path) {
72          AVLNode<K, V> leftChild = (AVLNode<K, V>) node.getLeft();
73          switch (leftChild.getBalance()) {
74          case 'L':
75              this.rotateLeft1L(node, leftChild, path);
76              break;
77          // case 'E':
```

```
78          // Impossible.
79          case 'R':
80              this.rotateLeft2(node, leftChild, path);
81              break;
82          }
83      }
84
85      // Every ancestor of node is stored in the stack, which is not
    empty.
86      // height( node.getRight() ) - height( node.getLeft() ) = 2.
87      protected void rebalanceInsRight(AVLNode<K, V> node,
88              Stack<PathStep<K, V>> path) {
89          AVLNode<K, V> rightChild = (AVLNode<K, V>) node.getRight();
90          switch (rightChild.getBalance()) {
91          case 'L':
92              this.rotateRight2(node, rightChild, path);
93              break;
94          // case 'E':
95          // Impossible.
96          case 'R':
97              this.rotateRight1R(node, rightChild, path);
98              break;
99          }
100     }
101
102     // If there is an entry in the dictionary whose key is the
    specified key,
103     // removes it from the dictionary and returns its value;
104     // otherwise, returns null.
105     public V remove(K key) {
106         Stack<PathStep<K, V>> path = new StackInList<PathStep<K,
    V>>();
107         BSTNode<K, V> node = this.findNode(key, path);
108         if (node == null)
109             return null;
110         else {
111             V oldValue = node.getValue();
112             if (node.getLeft() == null)
113                 // The left subtree is empty.
114                 this.linkSubtree(node.getRight(), path.top());
115             else if (node.getRight() == null)
```

```
116                 // The right subtree is empty.
117                 this.linkSubtree(node.getLeft(), path.top());
118             else {
119                 // Node has 2 children. Replace the node's entry
    with
120                 // the 'minEntry' of the right subtree.
121                 path.push(new PathStep<K, V>(node, false));
122                 BSTNode<K, V> minNode =
    this.minNode(node.getRight(), path);
123                 node.setEntry(minNode.getEntry());
124                 // Remove the 'minEntry' of the right subtree.
125                 this.linkSubtree(minNode.getRight(), path.top());
126             }
127             currentSize--;
128             this.reorganizeRem(path);
129             return oldValue;
130         }
131     }
132 
133     // Every ancestor of the removed node is stored in the stack,
134     // which is not empty.
135     /**
136      * Reorganiza após uma remoção.
137      *
138      * @param path
139      *             - caminho até ao nó onde ocorreu a remoção.
140      */
141     protected void reorganizeRem(Stack<PathStep<K, V>> path) {
142 
143         boolean decreased = true;
144         PathStep<K, V> lastStep = path.pop();
145         AVLNode<K, V> parent = (AVLNode<K, V>) lastStep.parent;
146         while (decreased && parent != null) {
147             if (lastStep.isLeftChild)
148                 // parent's left subtree has decreased.
149                 switch (parent.getBalance()) {
150                 case 'L':
151                     parent.setBalance('E');
152                     break;
153                 case 'E':
154                     parent.setBalance('R');
```

```
155                     decreased = false;
156                     break;
157                 case 'R':
158                     this.rebalanceRemLeft(parent, path);
159                     decreased = false;
160                     break;
161                 }
162             else
163                 // parent's right subtree has decreased.
164                 switch (parent.getBalance()) {
165                 case 'L':
166                     this.rebalanceRemRight(parent, path);
167                     decreased = false;
168                     break;
169                 case 'E':
170                     parent.setBalance('L');
171                     decreased = false;
172                     break;
173                 case 'R':
174                     parent.setBalance('E');
175                     break;
176                 }
177             lastStep = path.pop();
178             parent = (AVLNode<K, V>) lastStep.parent;
179         }
180 
181     }
182 
183     // Every ancestor of node is stored in the stack, which is not
    empty.
184     // height( node.getLeft() ) - height( node.getRight() ) = 2.
185     /**
186      * Reequilibra a árvore quando ocorre uma remoção à esquerda de
    um Nó (Pai)
187      * que estava com um desequilíbrio para a direita.
188      *
189      * @param node
190      *             Nó (pai) onde ocorreu a remoção.
191      * @param path
192      *             Caminho até ao nó (pai) dado.
193      */
```

```
194     protected void rebalanceRemLeft(AVLNode<K, V> node,
195             Stack<PathStep<K, V>> path) {
196         AVLNode<K, V> rightChild = (AVLNode<K, V>) node.getRight();
197         switch (rightChild.getBalance()) {
198         case 'L':
199             this.rotateRight2(node, rightChild, path);
200             break;
201         case 'E':
202             this.rotateRight1R(node, rightChild, path);
203             break;
204         case 'R':
205             this.rotateRight1R(node, rightChild, path);
206             break;
207         }
208     }
209 
210     // Every ancestor of node is stored in the stack, which is not
    empty.
211     // height( node.getLeft() ) - height( node.getRight() ) = 2.
212     /**
213      * Reequilibra a árvore quando ocorre uma remoção à direita de
    um Nó (Pai)
214      * que estava com um desequilíbrio para a esquerda.
215      *
216      * @param node
217      *             Nó (pai) onde ocorreu a remoção.
218      * @param path
219      *             Caminho até ao nó (pai) dado.
220      */
221     protected void rebalanceRemRight(AVLNode<K, V> node,
222             Stack<PathStep<K, V>> path) {
223         AVLNode<K, V> leftChild = (AVLNode<K, V>) node.getLeft();
224         switch (leftChild.getBalance()) {
225         case 'L':
226             this.rotateLeft1L(node, leftChild, path);
227             break;
228         case 'E':
229             this.rotateLeft1L(node, leftChild, path);
230             break;
231         case 'R':
232             this.rotateLeft2(node, leftChild, path);
```

```
233             break;
234         }
235     }
236 
237     // Performs a single left rotation rooted at theRoot,
238     // when the balance factor of its leftChild is 'L'.
239     //
240     // Every ancestor of theRoot is stored in the stack, which is
    not empty.
241     // height( node.getLeft() ) - height( node.getRight() ) = 2.
242     protected void rotateLeft1L(AVLNode<K, V> theRoot, AVLNode<K,
    V> leftChild,
243             Stack<PathStep<K, V>> path) {
244         theRoot.setBalance('E');
245         leftChild.setBalance('E');
246         this.rotateLeft(theRoot, leftChild, path);
247     }
248 
249     // Performs a single left rotation rooted at theRoot,
250     // when the balance factor of its leftChild is 'E'.
251     //
252     // Every ancestor of theRoot is stored in the stack, which is
    not empty.
253     // height( node.getLeft() ) - height( node.getRight() ) = 2.
254     protected void rotateLeft1E(AVLNode<K, V> theRoot, AVLNode<K,
    V> leftChild,
255             Stack<PathStep<K, V>> path) {
256         // theRoot.setBalance('L');
257         leftChild.setBalance('R');
258         this.rotateLeft(theRoot, leftChild, path);
259     }
260 
261     // Performs a single right rotation rooted at theRoot,
262     // when the balance factor of its rightChild is 'R'.
263     //
264     // Every ancestor of theRoot is stored in the stack, which is
    not empty.
265     // height( node.getRight() ) - height( node.getLeft() ) = 2.
266     protected void rotateRight1R(AVLNode<K, V> theRoot,
267             AVLNode<K, V> rightChild, Stack<PathStep<K, V>> path) {
268         theRoot.setBalance('E');
```

```
269         rightChild.setBalance('E');
270         this.rotateRight(theRoot, rightChild, path);
271     }
272 
273     // Performs a single right rotation rooted at theRoot,
274     // when the balance factor of its rightChild is 'E'.
275     //
276     // Every ancestor of theRoot is stored in the stack, which is
    not empty.
277     // height( node.getRight() ) - height( node.getLeft() ) = 2.
278     protected void rotateRight1E(AVLNode<K, V> theRoot,
279             AVLNode<K, V> rightChild, Stack<PathStep<K, V>> path) {
280         // theRoot.setBalance('R');
281         rightChild.setBalance('L');
282         this.rotateRight(theRoot, rightChild, path);
283     }
284 
285     // Performs a double left rotation rooted at theRoot.
286     //
287     // Every ancestor of theRoot is stored in the stack, which is
    not empty.
288     // height( node.getLeft() ) - height( node.getRight() ) = 2.
289     protected void rotateLeft2(AVLNode<K, V> theRoot, AVLNode<K, V>
    leftChild,
290             Stack<PathStep<K, V>> path) {
291         AVLNode<K, V> rightGrandchild = (AVLNode<K, V>)
    leftChild.getRight();
292         switch (rightGrandchild.getBalance()) {
293         case 'L':
294             leftChild.setBalance('E');
295             theRoot.setBalance('R');
296             break;
297         case 'E':
298             leftChild.setBalance('E');
299             theRoot.setBalance('E');
300             break;
301         case 'R':
302             leftChild.setBalance('L');
303             theRoot.setBalance('E');
304             break;
305         }
```

```
306         rightGrandchild.setBalance('E');
307         this.rotateLeft(theRoot, leftChild, rightGrandchild, path);
308     }
309 
310     // Performs a double right rotation rooted at theRoot.
311     //
312     // Every ancestor of theRoot is stored in the stack, which is
    not empty.
313     // height( node.getRight() ) - height( node.getLeft() ) = 2.
314     protected void rotateRight2(AVLNode<K, V> theRoot,
315             AVLNode<K, V> rightChild, Stack<PathStep<K, V>> path) {
316         AVLNode<K, V> leftGrandchild = (AVLNode<K, V>)
    rightChild.getLeft();
317         switch (leftGrandchild.getBalance()) {
318         case 'L':
319             theRoot.setBalance('E');
320             rightChild.setBalance('R');
321             break;
322         case 'E':
323             theRoot.setBalance('E');
324             rightChild.setBalance('E');
325             break;
326         case 'R':
327             theRoot.setBalance('L');
328             rightChild.setBalance('E');
329             break;
330         }
331         leftGrandchild.setBalance('E');
332         this.rotateRight(theRoot, rightChild, leftGrandchild,
    path);
333     }
334 
335 }
336 
```