```java
/*
 * HSR – Uebungen 'Algorithmen & Datenstrukturen 2'
 * Version: Sun Sep 22 13:31:09 CEST 2019
 */

package uebung02.ml.aufgabe01;

import java.util.Collection;
import java.util.LinkedList;

public class BinarySearchTree<K extends Comparable<? super K>, V> {

  protected Node root;

  public static class Entry<K, V> {

    private K key;
    private V value;

    public Entry(K key, V value) {
      this.key = key;
      this.value = value;
    }

    protected K setKey(K key) {
      K oldKey = this.key;
      this.key = key;
      return oldKey;
    }

    public K getKey() {
      return key;
    }

    public V setValue(V value) {
      V oldValue = this.value;
      this.value = value;
      return oldValue;
    }

    public V getValue() {
      return value;
    }

    @Override
    public String toString() {
      StringBuilder result = new StringBuilder();
      result.append("[").append(key).append("/").append(value).append("]");
      return result.toString();
    }

  } // End of class Entry

  protected class Node {

    private Entry<K, V> entry;
    private Node leftChild;
    private Node rightChild;

    public Node(Entry<K, V> entry) {
      this.entry = entry;
    }

    public Node(Entry<K, V> entry, Node leftChild, Node rightChild) {
      this.entry = entry;
      this.leftChild = leftChild;
      this.rightChild = rightChild;
    }
```



```java

    public Entry<K, V> getEntry() {
      return entry;
    }

    public Entry<K, V> setEntry(Entry<K, V> entry) {
      Entry<K, V> oldEntry = entry;
      this.entry = entry;
      return oldEntry;
    }

    public Node getLeftChild() {
      return leftChild;
    }

    public void setLeftChild(Node leftChild) {
      this.leftChild = leftChild;
    }

    public Node getRightChild() {
      return rightChild;
    }

    public void setRightChild(Node rightChild) {
      this.rightChild = rightChild;
    }

  } // End of class Node


  public Entry<K, V> insert(K key, V value) {
    Entry<K, V> newEntry = new Entry<>(key, value);
    root = insert(root, newEntry);
    return newEntry;
  }

  protected Node insert(Node node, Entry<K, V> entry) {
    if (node == null)
      return newNode(entry);
    else if (entry.getKey().compareTo(node.getEntry().getKey()) <= 0) {
      node.leftChild = insert(node.leftChild, entry);
    } else { /* if (entry.key > node.key) */
      node.rightChild = insert(node.rightChild, entry);
    }
    return node;
  }

  /**
   * Factory-Method: Creates a new node.
   *
   * @param entry
   *          The entry to be inserted in the new node.
   * @return The new created node.
   */
  protected Node newNode(Entry<K, V> entry) {
    return new Node(entry);
  }

  public void clear() {
    root = null;
  }

  public Entry<K, V> find(K key) {
    Node result = find(root, key);
    if (result == null) {
      return null;
    } else {
      return result.getEntry();
    }
  }
```

```
139
140   protected Node find(Node node, K key) {
141     if (node == null) {
142       return null;
143     }
144     if (key.compareTo(node.getEntry().getKey()) < 0) {
145       return find(node.leftChild, key);
146     }
147     if (key.compareTo(node.getEntry().getKey()) > 0) {
148       return find(node.rightChild, key);
149     }
150     return node;
151   };
152
153   /**
154    * Returns a collection with all entries with key.
155    *
156    * @param key
157    *          The key to be searched.
158    * @return Collection of all entries found. An empty collection is returned if
159    *         no entry with key is found.
160    */
161   public Collection<Entry<K, V>> findAll(K key) {
162     Collection<Entry<K, V>> entries = new LinkedList<Entry<K, V>>();
163     findAll(root, key, entries);
164     return entries;
165   }
166
167   protected void findAll(Node node, K key, Collection<Entry<K, V>> entries) {
168     if (node == null) {
169       return;
170     }
171     if (key.compareTo(node.getEntry().getKey()) == 0) {
172       entries.add(node.getEntry());
173     }
174     if (key.compareTo(node.getEntry().getKey()) <= 0) {
175       findAll(node.leftChild, key, entries);
176     }
177     if (key.compareTo(node.getEntry().getKey()) >= 0) {
178       findAll(node.rightChild, key, entries);
179     }
180   }
181
182   /**
183    * Returns a collection with all entries in inorder.
184    *
185    * @return Inorder-Collection of all entries.
186    */
187   public Collection<Entry<K, V>> inorder() {
188     Collection<Entry<K, V>> coll = new LinkedList<>();
189     inorder(root, coll);
190     return coll;
191   }
192
193   protected void inorder(Node node, Collection<Entry<K, V>> coll) {
194     if (node == null)
195       return;
196     inorder(node.leftChild, coll);
197     coll.add(node.getEntry());
198     inorder(node.rightChild, coll);
199   }
200
201   /**
202    * Prints the entries of the tree as a list in inorder to the console.
203    */
204   public void printInorder() {
205     inorder().stream().forEach(e -> {
206       System.out.print(e + " ");
207     });
208     System.out.println();
209   }
```



```
210
211   public Entry<K, V> remove(Entry<K, V> entry) {
212     if (entry == null) {
213       return null;
214     }
215     RemoveResult result = remove(root, entry);
216     root = result.node;
217     return result.entry;
218   }
219
220   protected class RemoveResult {
221
222     private Node node;
223     private Entry<K, V> entry;
224
225     public RemoveResult(Node node, Entry<K, V> entry) {
226       this.node = node;
227       this.entry = entry;
228     }
229
230     RemoveResult set(Node node) {
231       this.node = node;
232       return this;
233     }
234
235     public Node getNode() {
236       return node;
237     }
238
239     public Entry<K, V> getEntry() {
240       return entry;
241     }
242
243   }
```

```
244
245   protected RemoveResult remove(final Node node, final Entry<K, V> entry) {
246     RemoveResult result = null;
247     if (node == null) {
248       return new RemoveResult(null, null);
249     }
250     if (entry.getKey().compareTo(node.getEntry().getKey()) < 0) {
251       result = remove(node.leftChild, entry);
252       node.leftChild = result.node;
253       return result.set(node);
254     } else if (entry.getKey().compareTo(node.getEntry().getKey()) > 0) {
255       result = remove(node.rightChild, entry);
256       node.rightChild = result.node;
257       return result.set(node);
258     } else {
259       // Key found: is this the correct entry?
260       if (node.getEntry() != entry) {
261         // Searching for next entry with this key
262         result = remove(node.leftChild, entry);
263         node.leftChild = result.node;
264         if (result.entry == null) {
265           result = remove(node.rightChild, entry);
266           node.rightChild = result.node;
267         }
268         return result.set(node);
269       }
270       // We have reachted the correct node.
271       if (node.leftChild == null) {
272         return new RemoveResult(node.rightChild, node.getEntry());
273       }
274       if (node.rightChild == null) {
275         return new RemoveResult(node.leftChild, node.getEntry());
276       }
277       Entry<K, V> entryRemoved = node.getEntry();
278       Node q = getParentNext(node);
279       if (q == node) {
280         node.setEntry(node.rightChild.getEntry());
281         q.rightChild = q.rightChild.rightChild;
282       } else {
283         node.setEntry(q.leftChild.getEntry());
284         q.leftChild = q.leftChild.rightChild;
285       }
286       return new RemoveResult(node, entryRemoved);
287     }
288   }
289
290   /**
291    * Search for the inorder successor.
292    *
293    * @param p
294    *          The node for which the inorder successor shall be searched.
295    * @return The parent-node(!) of the inorder successor.
296    */
297   protected Node getParentNext(Node p) {
298     if (p.rightChild.leftChild != null) {
299       p = p.rightChild;
300       while (p.leftChild.leftChild != null)
301         p = p.leftChild;
302     }
303     return p;
304   }
305
306   /**
307    * The height of the tree.
308    *
309    * @return The actual height. -1 for an empty tree.
310    */
311   public int getHeight() {
312     return getHeight(root);
313   }
```



```
314
315   protected int getHeight(Node p) {
316     if (p == null)
317       return -1;
318     int rHeight = getHeight(p.rightChild);
319     int lHeight = getHeight(p.leftChild);
320     return (lHeight < rHeight ? rHeight : lHeight) + 1;
321   }
322
323   public int size() {
324     return size(root);
325   }
326
327   protected int size(Node n) {
328     if (n == null) {
329       return 0;
330     }
331     return size(n.leftChild) + size(n.rightChild) + 1;
332   }
333
334   public boolean isEmpty() {
335     return size() == 0;
336   }
337
338   public static void main(String[] args) {
339
340     // Example from lecture "L?schen (IV/IV)":
341     //BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
342     BinarySearchTree<Integer, String> bst = new BinarySearchTreeADV<>("L?schen (IV/IV)
    ", 0, 4);
343     System.out.println("Inserting:");
344     bst.insert(1, "Str1");
345     bst.printInorder();
346     bst.insert(3, "Str3");
347     bst.printInorder();
348     bst.insert(2, "Str2");
349     bst.printInorder();
350     bst.insert(8, "Str8");
351     bst.printInorder();
352     bst.insert(9, "Str9");
353     bst.insert(6, "Str6");
354     bst.insert(5, "Str5");
355     bst.printInorder();
356
357     System.out.println("Removeing 3:");
358     Entry<Integer, String> entry = bst.find(3);
359     System.out.println(entry);
360     bst.remove(entry);
361     bst.printInorder();
362
363   }
364
365   /* Session-Log:
366
367   Inserting:
368   [1/Str1]
369   [1/Str1] [3/Str3]
370   [1/Str1] [2/Str2] [3/Str3]
371   [1/Str1] [2/Str2] [3/Str3] [8/Str8]
372   [1/Str1] [2/Str2] [3/Str3] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
373   Removeing 3:
374   [3/Str3]
375   [1/Str1] [2/Str2] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
376
377   */
378
379 } // End of class BinarySearchTree
```

```java
/*
 * HSR − Uebungen 'Algorithmen & Datenstrukturen 2'
 * Version: Sun Sep 22 13:31:09 CEST 2019
 */

package uebung02.ml.aufgabe01;

import java.util.Iterator;
import java.util.Random;

import uebung02.ml.aufgabe01.BinarySearchTree.Entry;

public class BinarySearchTreeTest {

  private static Random randomGenerator = new Random(1);

  private static BinarySearchTree<Integer, String> generateTree(int nodes) {
    int key;
    BinarySearchTree<Integer, String> ret = new BinarySearchTree<>();
    for (int i = 0; i < nodes; i++) {
      key = randomGenerator.nextInt() * Integer.MAX_VALUE;
      ret.insert(key, "String_" + i);
    }
    return ret;
  }

  public static void main(String[] args) {
    System.out.println("BINARY TREE TEST");
    System.out
        .println("Please be patient, the following operations may take some time...");
    final int TESTRUNS = 100;
    final int BEGINSIZE = 10000;
    final int VARYSIZE = 10;
    long startTime = System.currentTimeMillis();

    BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
    double avgHeight = 0;
    double avgEntries = 0;
    double avgTime = 0;
    for (int i = 0; i < TESTRUNS; i++) {
      startTime = System.currentTimeMillis();
      bst = generateTree(BEGINSIZE + i * VARYSIZE);
      avgTime += System.currentTimeMillis() − startTime;
      avgHeight += bst.getHeight();
      avgEntries += BEGINSIZE + i * VARYSIZE;
    }
    avgTime /= TESTRUNS;
    avgEntries /= TESTRUNS;
    avgHeight /= TESTRUNS;
    System.out.println("Test successful, results are as follows:");
    System.out.println("Average time for generation is: " + avgTime + "ms");
    System.out.println("Average entries are: " + avgEntries);
    System.out.println("Average height is: " + avgHeight);
    System.out.println("In h=C*log2(n), C=h/log2(n) = " + avgHeight
        / (Math.log(avgEntries) / Math.log(2)));
    System.out.println();
```



```java

    bst = generateTree(20);
    int search = 15138431;
    Entry<Integer, String> searchResult;
    bst.insert(search, "String_" + search);
    searchResult = bst.find(search);
    if (searchResult == null) {
      System.err.println("Search for node " + search + " failed!");
    } else {
      System.out.println("Search for node " + search + " successful!");
    }
    System.out.println();
    bst.insert(search, "String_" + search);
    bst.insert(search, "String_" + search);
    bst.insert(search, "String_" + search);
    Iterator<Entry<Integer, String>> it = bst.findAll(search).iterator();
    int count = 0;
    while (it.hasNext()) {
      count++;
      it.next();
      System.out.println("Search for node " + search + " successful!");
    }
    System.out.println("Search for node " + search + ": " + count
        + " nodes found!");
    System.out.println();
    it = bst.findAll(search).iterator();
    count = 0;
    while (it.hasNext()) {
      bst.remove(it.next());
    }

    it = bst.findAll(search).iterator();
    count = 0;
    while (it.hasNext()) {
      count++;
      it.next();
      System.out.println("Search for node " + search + " successful!");
    }
    System.out.println("Search for node " + search + ": " + count
        + " nodes found!");
  }

}

/* Session−Log:

BINARY TREE TEST
Please be patient, the following operations may take some time...
Test successful, results are as follows:
Average time for generation is: 9.07ms
Average entries are: 10495.0
Average height is: 30.81
In h=C*log2(n), C=h/log2(n) = 2.306584099301782

Search for node 15138431 successful!

Search for node 15138431 successful!
Search for node 15138431 successful!
Search for node 15138431 successful!
Search for node 15138431 successful!
Search for node 15138431: 4 nodes found!

Search for node 15138431: 0 nodes found!

*/
```

```java
/*
 * HSR − Uebungen 'Algorithmen & Datenstrukturen 2'
 * Version: Sun Sep 22 13:31:09 CEST 2019
 */

package uebung02.ml.aufgabe01;

import static org.junit.Assert.*;

import java.util.Collection;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Random;

import org.junit.Before;
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

import uebung02.ml.aufgabe01.BinarySearchTree.Entry;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class BinarySearchTreeJUnitTest {

  BinarySearchTree<Integer, String> bst;

  @Before
  public void setUp() {
    bst = new BinarySearchTree<Integer, String>();
  }

  @Test
  public void test01EmptySizeInsertClear() {
    assertTrue(bst.isEmpty());
    assertEquals(0, bst.size());
    bst.insert(1, "String_1");
    assertEquals(1, bst.size());
    assertFalse(bst.isEmpty());
    bst.insert(2, "String_2");
    assertEquals(2, bst.size());
    bst.insert(2, "String_2");
    assertEquals(3, bst.size());
    bst.clear();
    assertTrue(bst.isEmpty());
    assertEquals(0, bst.size());
  }

  @Test
  public void test02Find() {
    Entry<Integer, String> entry;
    entry = bst.find(1);
    assertNull(entry);
    Entry<Integer, String> insertedEntry = bst.insert(1, "String_1");
    entry = bst.find(1);
    assertNotNull(entry);
    assertEquals(Integer.valueOf(1), entry.getKey());
    assertEquals("String_1", entry.getValue());
    assertSame(insertedEntry, entry);
  }
```



```java

  @Test
  public void test03FindAll() {
    Collection<Entry<Integer, String>> col;
    col = bst.findAll(1);
    assertEquals(0, col.size());
    bst.insert(1, "String_1");
    col = bst.findAll(2);
    assertEquals(0, col.size());
    bst.insert(2, "String_2");
    col = bst.findAll(2);
    assertEquals(1, col.size());
    bst.insert(2, "String_2");
    col = bst.findAll(2);
    assertEquals(2, col.size());
  }

  @Test
  public void test04GetHeight() {
    assertEquals(-1, bst.getHeight());
    bst.insert(1, "String_1");
    assertEquals(0, bst.getHeight());
    bst.insert(2, "String_2");
    assertEquals(1, bst.getHeight());
  }

  @Test
  public void test05Remove() {
    Entry<Integer, String> entry = new Entry<>(1, "String_1");
    entry = bst.remove(entry);
    assertNull(entry);
    final Entry<Integer, String> entry1 = bst.insert(1, "String_1");
    entry = bst.remove(entry1);
    assertSame(entry, entry1);
    assertEquals(0, bst.size());
    final Entry<Integer, String> entry1a = bst.insert(1, "String_1a");
    final Entry<Integer, String> entry1b = bst.insert(1, "String_1b");
    assertEquals(2, bst.size());
    entry = bst.remove(entry1a);
    assertSame(entry1a, entry);
    assertEquals(1, bst.size());
    entry = bst.remove(entry1b);
    assertSame(entry1b, entry);
    assertEquals(0, bst.size());
  }
```

```
107
108   @Test
109   public void test06RemoveCase3() {
110     bst.insert(1, "String_1");
111     Entry<Integer, String> entryToRemove = bst.insert(3, "String_3");
112     bst.insert(2, "String_2");
113     bst.insert(8, "String_8");
114     bst.insert(6, "String_6");
115     bst.insert(9, "String_9");
116     bst.insert(5, "String_5");
117     assertEquals(7, bst.size());
118     assertEquals(4, bst.getHeight());
119     Entry<Integer, String> removedEntry =  bst.remove(entryToRemove);
120     assertSame(entryToRemove, removedEntry);
121     assertEquals(6, bst.size());
122     assertEquals(3, bst.getHeight());
123     bst.remove(bst.find(6));
124     assertEquals(5, bst.size());
125     assertEquals(3, bst.getHeight());
126     bst.remove(bst.find(9));
127     assertEquals(4, bst.size());
128     assertEquals(2, bst.getHeight());
129   }
130
131   @Test
132   public void test07RemoveCase3Special() {
133     bst.insert(2, "String_2");
134     bst.insert(1, "String_1");
135     bst.insert(3, "String_3.1");
136     bst.insert(3, "String_3.2");
137     Collection<Entry<Integer, String>> col;
138     col = bst.findAll(3);
139     assertEquals(2, col.size());
140     Entry<Integer, String> removedEntry = bst.remove(bst.find(2));
141     assertNotNull(removedEntry);
142     assertEquals("String_2", removedEntry.getValue());
143     col = bst.findAll(3);
144     assertEquals(2, col.size());
145   }
146
147   @Test
148   public void test09StressTest() {
149     final int SIZE = 10000;
150     Random randomGenerator = new Random();
151     List<Entry<Integer, String>> entriesList = new LinkedList<>();
152     // key-Counters: count for every key how many time it was generated
153     Map<Integer, Integer> keyCounters = new HashMap<>();
154     // fill the Tree
155     for (int i = 0; i < SIZE; i++) {
156       int key = (int) (randomGenerator.nextFloat() * SIZE / 3);
157       Integer numberOfKeys = keyCounters.get(key);
158       if (numberOfKeys == null) {
159         numberOfKeys = 1;
160       } else {
161         numberOfKeys++;
162       }
163       keyCounters.put(key, numberOfKeys);
164       Entry<Integer, String> entry = bst.insert(key, "String_" + i);
165       entriesList.add(entry);
166       assertEquals(i + 1, bst.size());
167     }
168     // verify the number of entries per key
169     for (Map.Entry<Integer, Integer> keyEntry : keyCounters.entrySet()) {
170       int key = keyEntry.getKey();
171       int numberOfKeys = keyEntry.getValue();
172       assertEquals(numberOfKeys, bst.findAll(key).size());
173     }
```



```
174
175     // remove all entries
176     int size = bst.size();
177     for (Entry<Integer, String> entry : entriesList) {
178       Entry<Integer, String> deletedEntry = bst.remove(entry);
179       assertSame(entry, deletedEntry);
180       assertEquals(--size, bst.size());
181     }
182   }
183
184 }
185
```

```java
/*
 * HSR – Uebungen 'Algorithmen & Datenstrukturen 2'
 * Version: Sun Sep 22 13:31:09 CEST 2019
 */

package uebung02.ml.aufgabe01;

import ch.hsr.adv.commons.core.logic.domain.styles.ADVStyle;
import ch.hsr.adv.commons.core.logic.util.ADVException;
import ch.hsr.adv.commons.tree.logic.domain.ADVBinaryTreeNode;
import ch.hsr.adv.lib.bootstrapper.ADV;
import ch.hsr.adv.lib.tree.logic.binarytree.BinaryTreeModule;

@SuppressWarnings("unchecked")
public class BinarySearchTreeADV<K extends Comparable<? super K>, V>
    extends BinarySearchTree<K, V> {

  protected BinaryTreeModule advTree;

  protected class NodeADV extends BinarySearchTree<K, V>.Node
      implements ADVBinaryTreeNode<String> {

    protected NodeADV(Entry<K, V> entry) {
      super(entry);
    }

    @Override
    public String getContent() {
      return getEntry().getKey() + " / " + getEntry().getValue();
    }

    @Override
    public ADVStyle getStyle() {
      return null;
    }

    @Override
    public NodeADV getLeftChild() {
      return (NodeADV) super.getLeftChild();
    }

    @Override
    public NodeADV getRightChild() {
      return (NodeADV) super.getRightChild();
    }

  } // class BinaryTreeTestGVS.NodeGVS

  public BinarySearchTreeADV(String sessionName) {
    this(sessionName, -1, -1);
  }

  public BinarySearchTreeADV(String sessionName,
                             int maxLeftHeight, int maxRightHeight) {
    advTree = new BinaryTreeModule(sessionName);
    if ((maxLeftHeight != -1) && (maxLeftHeight != -1)) {
      advTree.setFixedTreeHeight(maxLeftHeight, maxRightHeight);
    }
    try {
      ADV.launch(null);
    } catch (ADVException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
```



```java

  @Override
  protected Node newNode(Entry<K, V> entry) {
    return new NodeADV(entry);
  }

  @Override
  public Entry<K, V> insert(K key, V value) {
    Entry<K, V> newEntry = super.insert(key, value);
    displayOnADV("insert(" + key + "," + value + ")");
    return newEntry;
  }

  @Override
  public Entry<K, V> remove(Entry<K, V> entry) {
    Entry<K, V> deletedEntry = super.remove(entry);
    displayOnADV("remove(" + entry + ")");
    return deletedEntry;
  }

  protected void displayOnADV(String advMessage) {
    advTree.setRoot((NodeADV) root);
    try {
      ADV.snapshot(advTree, "\n" + advMessage);
    } catch (ADVException e) {
      e.printStackTrace();
      System.exit(2);
    }
  }

}
```

```
1  /*
2   * HSR – Uebungen 'Algorithmen & Datenstrukturen 2'
3   * Version: Sun Sep 22 13:31:09 CEST 2019
4   */
5
6  package uebung02.ml.aufgabe01;
7
8  public class BinarySearchTreeTestADV {
9
10   public static void main(String[] args) {
11
12     BinarySearchTree<Integer, String> bts =
13         new BinarySearchTreeADV<>("Deleting internal node", 0, 4);
14
15     // Example from script: deleting internal node (slide 14):
16     int[] iarr = { 1, 3, 2, 8, 6, 9, 5 };
17     for (int i : iarr) {
18       bts.insert(i, "Str" + i);
19     }
20     bts.remove(bts.find(3));
21
22   }
23
24 }
25
```