```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Tue Sep 24 09:34:25 CEST 2024
 */

package ex02.solution.task01;

import java.util.Collection;
import java.util.LinkedList;
import java.util.stream.Collectors;

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

  public 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;
    }
    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<>();
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<Node> inorderNodeList = new LinkedList<>();
189     inorder(root, inorderNodeList);
190     Collection<Entry<K, V>> inorderEntryList = inorderNodeList.stream()
191         .map(node -> node.getEntry()).collect(Collectors.toList());
192     return inorderEntryList;
193   }
194
195   protected void inorder(Node node, Collection<Node> inorderList) {
196     if (node == null)
197       return;
198     inorder(node.leftChild, inorderList);
199     inorderList.add(node);
200     inorder(node.rightChild, inorderList);
201   }
```



```
202
203   /**
204    * Prints the entries of the tree as a list in inorder to the console.
205    */
206   public void printInorder() {
207     inorder().stream().forEach(e -> {
208       System.out.print(e + " ");
209     });
210     System.out.println();
211   }
212
213   public Entry<K, V> remove(Entry<K, V> entry) {
214     if (entry == null) {
215       return null;
216     }
217     RemoveResult result = remove(root, entry);
218     root = result.node;
219     return result.entry;
220   }
221
222   protected class RemoveResult {
223
224     private Node node;
225     private Entry<K, V> entry;
226
227     public RemoveResult(Node node, Entry<K, V> entry) {
228       this.node = node;
229       this.entry = entry;
230     }
231
232     RemoveResult set(Node node) {
233       this.node = node;
234       return this;
235     }
236
237     public Node getNode() {
238       return node;
239     }
240
241     public Entry<K, V> getEntry() {
242       return entry;
243     }
244
245   }
```

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



```
308
309   /**
310    * The height of the tree.
311    *
312    * @return The current height. -1 for an empty tree.
313    */
314   public int getHeight() {
315     return getHeight(root);
316   }
317
318   protected int getHeight(Node p) {
319     if (p == null)
320       return -1;
321     int rHeight = getHeight(p.rightChild);
322     int lHeight = getHeight(p.leftChild);
323     return (lHeight < rHeight ? rHeight : lHeight) + 1;
324   }
325
326   public int size() {
327     return size(root);
328   }
329
330   protected int size(Node n) {
331     if (n == null) {
332       return 0;
333     }
334     return size(n.leftChild) + size(n.rightChild) + 1;
335   }
336
337   public boolean isEmpty() {
338     return size() == 0;
339   }
```

```
340
341   public static void main(String[] args) {
342
343     // Example from lecture "Löschen (IV/IV)":
344     BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
345     //BinarySearchTree<Integer, String> bst = new BinarySearchTreeADV<>("Loeschen (IV/
    IV)");
346     //BinarySearchTree<Integer, String> bst = new BinarySearchTreeADV<>("Loeschen (IV/
    IV)", 0, 4);
347     System.out.println("Inserting:");
348     bst.insert(1, "Str1");
349     bst.printInorder();
350     bst.insert(3, "Str3");
351     bst.printInorder();
352     bst.insert(2, "Str2");
353     bst.printInorder();
354     bst.insert(8, "Str8");
355     bst.printInorder();
356     bst.insert(9, "Str9");
357     bst.insert(6, "Str6");
358     bst.insert(5, "Str5");
359     bst.printInorder();
360
361     System.out.println("Removeing 3:");
362     Entry<Integer, String> entry = bst.find(3);
363     System.out.println(entry);
364     bst.remove(entry);
365     bst.printInorder();
366
367   }
368
369   /* Session-Log:
370
371   Inserting:
372   [1/Str1]
373   [1/Str1] [3/Str3]
374   [1/Str1] [2/Str2] [3/Str3]
375   [1/Str1] [2/Str2] [3/Str3] [8/Str8]
376   [1/Str1] [2/Str2] [3/Str3] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
377   Removeing 3:
378   [3/Str3]
379   [1/Str1] [2/Str2] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
380
381   */
382
383 } // End of class BinarySearchTree
```



```
1  /*
2   * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
3   * Version: Tue Sep 24 09:34:25 CEST 2024
4   */
5
6  package ex02.solution.task01;
7
8  import java.util.Iterator;
9  import java.util.Random;
10
11 import ex02.solution.task01.BinarySearchTree.Entry;
12
13 public class BinarySearchTreeTest {
14
15   private static Random randomGenerator = new Random(1);
16
17   private static BinarySearchTree<Integer, String> generateTree(int nodes) {
18     int key;
19     BinarySearchTree<Integer, String> ret = new BinarySearchTree<>();
20     for (int i = 0; i < nodes; i++) {
21       key = randomGenerator.nextInt() * Integer.MAX_VALUE;
22       ret.insert(key, "String_" + i);
23     }
24     return ret;
25   }
26
27   public static void main(String[] args) {
28     System.out.println("BINARY TREE TEST");
29     System.out
30         .println("Please be patient, the following operations may take some time...");
31     final int TESTRUNS = 100;
32     final int BEGINSIZE = 10000;
33     final int VARYSIZE = 10;
34     long startTime = System.currentTimeMillis();
35
36     BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
37     double avgHeight = 0;
38     double avgEntries = 0;
39     double avgTime = 0;
40     for (int i = 0; i < TESTRUNS; i++) {
41       startTime = System.currentTimeMillis();
42       bst = generateTree(BEGINSIZE + i * VARYSIZE);
43       avgTime += System.currentTimeMillis() - startTime;
44       avgHeight += bst.getHeight();
45       avgEntries += BEGINSIZE + i * VARYSIZE;
46     }
47     avgTime /= TESTRUNS;
48     avgEntries /= TESTRUNS;
49     avgHeight /= TESTRUNS;
50     System.out.println("Test successful, results are as follows:");
51     System.out.println("Average time for generation is: " + avgTime + " ms");
52     System.out.println("Average entries are: " + avgEntries);
53     System.out.println("Average height is: " + avgHeight);
54     System.out.println("In h=C*log2(n), C=h/log2(n) = " + avgHeight
55         / (Math.log(avgEntries) / Math.log(2)));
56     System.out.println();
```

```
57
58      bst = generateTree(20);
59      int search = 15138431;
60      Entry<Integer, String> searchResult;
61      bst.insert(search, "String_" + search);
62      searchResult = bst.find(search);
63      if (searchResult == null) {
64        System.err.println("Search for node " + search + " failed!");
65      } else {
66        System.out.println("Search for node " + search + " successful!");
67      }
68      System.out.println();
69      bst.insert(search, "String_" + search);
70      bst.insert(search, "String_" + search);
71      bst.insert(search, "String_" + search);
72      Iterator<Entry<Integer, String>> it = bst.findAll(search).iterator();
73      int count = 0;
74      while (it.hasNext()) {
75        count++;
76        it.next();
77        System.out.println("Search for node " + search + " successful!");
78      }
79      System.out.println("Search for node " + search + ": " + count
80          + " nodes found!");
81      System.out.println();
82      it = bst.findAll(search).iterator();
83      count = 0;
84      while (it.hasNext()) {
85        bst.remove(it.next());
86      }
87
88      it = bst.findAll(search).iterator();
89      count = 0;
90      while (it.hasNext()) {
91        count++;
92        it.next();
93        System.out.println("Search for node " + search + " successful!");
94      }
95      System.out.println("Search for node " + search + ": " + count
96          + " nodes found!");
97    }
98
99  }
100
101 /* Session-Log:
102
103 BINARY TREE TEST
104 Please be patient, the following operations may take some time...
105 Test successful, results are as follows:
106 Average time for generation is: 4.12 ms
107 Average entries are: 10495.0
108 Average height is: 30.25
109 In h=C*log2(n), C=h/log2(n) = 2.2646598183667286
110
111 Search for node 15138431 successful!
112
113 Search for node 15138431 successful!
114 Search for node 15138431 successful!
115 Search for node 15138431 successful!
116 Search for node 15138431 successful!
117 Search for node 15138431: 4 nodes found!
118
119 Search for node 15138431: 0 nodes found!
120
121 */
122
```



```
1   /*
2    * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
3    * Version: Tue Sep 24 09:34:25 CEST 2024
4    */
5
6   package ex02.solution.task01;
7
8   import static org.junit.Assert.*;
9
10  import java.util.Collection;
11  import java.util.HashMap;
12  import java.util.LinkedList;
13  import java.util.List;
14  import java.util.Map;
15  import java.util.Random;
16
17  import org.junit.Before;
18  import org.junit.FixMethodOrder;
19  import org.junit.Test;
20  import org.junit.runners.MethodSorters;
21
22  import ex02.solution.task01.BinarySearchTree.Entry;
23
24  @FixMethodOrder(MethodSorters.NAME_ASCENDING)
25  public class BinarySearchTreeJUnitTest {
26
27    BinarySearchTree<Integer, String> bst;
28
29    @Before
30    public void setUp() {
31      bst = new BinarySearchTree<>();
32    }
33
34    @Test
35    public void test01EmptySizeInsertClear() {
36      assertTrue(bst.isEmpty());
37      assertEquals(0, bst.size());
38      bst.insert(1, "String_1");
39      assertEquals(1, bst.size());
40      assertFalse(bst.isEmpty());
41      bst.insert(2, "String_2");
42      assertEquals(2, bst.size());
43      bst.insert(2, "String_2");
44      assertEquals(3, bst.size());
45      bst.clear();
46      assertTrue(bst.isEmpty());
47      assertEquals(0, bst.size());
48    }
49
50    @Test
51    public void test02Find() {
52      Entry<Integer, String> entry;
53      entry = bst.find(1);
54      assertNull(entry);
55      Entry<Integer, String> insertedEntry = bst.insert(1, "String_1");
56      entry = bst.find(1);
57      assertNotNull(entry);
58      assertEquals(Integer.valueOf(1), entry.getKey());
59      assertEquals("String_1", entry.getValue());
60      assertSame(insertedEntry, entry);
61    }
```

```
62
63    @Test
64    public void test03FindAll() {
65      Collection<Entry<Integer, String>> col;
66      col = bst.findAll(1);
67      assertEquals(0, col.size());
68      bst.insert(1, "String_1");
69      col = bst.findAll(2);
70      assertEquals(0, col.size());
71      bst.insert(2, "String_2");
72      col = bst.findAll(2);
73      assertEquals(1, col.size());
74      bst.insert(2, "String_2");
75      col = bst.findAll(2);
76      assertEquals(2, col.size());
77    }
78
79    @Test
80    public void test04GetHeight() {
81      assertEquals(-1, bst.getHeight());
82      bst.insert(1, "String_1");
83      assertEquals(0, bst.getHeight());
84      bst.insert(2, "String_2");
85      assertEquals(1, bst.getHeight());
86    }
87
88    @Test
89    public void test05Remove() {
90      Entry<Integer, String> entry = new Entry<>(1, "String_1");
91      entry = bst.remove(entry);
92      assertNull(entry);
93      final Entry<Integer, String> entry1 = bst.insert(1, "String_1");
94      entry = bst.remove(entry1);
95      assertSame(entry, entry1);
96      assertEquals(0, bst.size());
97      final Entry<Integer, String> entry1a = bst.insert(1, "String_1a");
98      final Entry<Integer, String> entry1b = bst.insert(1, "String_1b");
99      assertEquals(2, bst.size());
100     entry = bst.remove(entry1a);
101     assertSame(entry1a, entry);
102     assertEquals(1, bst.size());
103     entry = bst.remove(entry1b);
104     assertSame(entry1b, entry);
105     assertEquals(0, bst.size());
106   }
```



```
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(1);
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
```



```
1   /*
2    * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
3    * Version: Tue Sep 24 09:34:25 CEST 2024
4    */
5
6   package ex02.solution.task01;
7
8   import ch.hsr.adv.commons.core.logic.domain.styles.ADVStyle;
9   import ch.hsr.adv.commons.core.logic.util.ADVException;
10  import ch.hsr.adv.commons.tree.logic.domain.ADVBinaryTreeNode;
11  import ch.hsr.adv.lib.bootstrapper.ADV;
12  import ch.hsr.adv.lib.tree.logic.binarytree.BinaryTreeModule;
13
14  @SuppressWarnings("unchecked")
15  public class BinarySearchTreeADV<K extends Comparable<? super K>, V>
16      extends BinarySearchTree<K, V> {
17
18    protected BinaryTreeModule advTree;
19
20    protected class NodeADV extends BinarySearchTree<K, V>.Node
21        implements ADVBinaryTreeNode<String> {
22
23      protected NodeADV(Entry<K, V> entry) {
24        super(entry);
25      }
26
27      @Override
28      public String getContent() {
29        return getEntry().getKey() + " / " + getEntry().getValue();
30      }
31
32      @Override
33      public ADVStyle getStyle() {
34        return null;
35      }
36
37      @Override
38      public NodeADV getLeftChild() {
39        return (NodeADV) super.getLeftChild();
40      }
41
42      @Override
43      public NodeADV getRightChild() {
44        return (NodeADV) super.getRightChild();
45      }
46
47    } // class BinaryTreeTestADV.NodeADV
48
49    public BinarySearchTreeADV(String sessionName) {
50      this(sessionName, -1, -1);
51    }
52
53    public BinarySearchTreeADV(String sessionName,
54                               int maxLeftHeight, int maxRightHeight) {
55      advTree = new BinaryTreeModule(sessionName);
56      if ((maxLeftHeight != -1) && (maxLeftHeight != -1)) {
57        advTree.setFixedTreeHeight(maxLeftHeight, maxRightHeight);
58      }
59      try {
60        ADV.launch(null);
61      } catch (ADVException e) {
62        e.printStackTrace();
63        System.exit(1);
64      }
65    }
```

```
66
67    @Override
68    protected Node newNode(Entry<K, V> entry) {
69      return new NodeADV(entry);
70    }
71
72    @Override
73    public Entry<K, V> insert(K key, V value) {
74      Entry<K, V> newEntry = super.insert(key, value);
75      displayOnADV("insert(" + key + "," + value + ")");
76      return newEntry;
77    }
78
79    @Override
80    public Entry<K, V> remove(Entry<K, V> entry) {
81      Entry<K, V> deletedEntry = super.remove(entry);
82      displayOnADV("remove(" + entry + ")");
83      return deletedEntry;
84    }
85
86    protected void displayOnADV(String advMessage) {
87      advTree.setRoot((NodeADV) root);
88      try {
89        ADV.snapshot(advTree, "\n" + advMessage);
90      } catch (ADVException e) {
91        e.printStackTrace();
92        System.exit(2);
93      }
94    }
95
96  }
```



```
1  /*
2   * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
3   * Version: Tue Sep 24 09:34:25 CEST 2024
4   */
5
6  package ex02.solution.task01;
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");
14         new BinarySearchTreeADV<>("Deleting internal node", 0, 4);
15
16     // Example from script: deleting internal node (slide 14):
17     int[] iarr = { 1, 3, 2, 8, 6, 9, 5 };
18     for (int i : iarr) {
19       bts.insert(i, "Str" + i);
20     }
21     bts.remove(bts.find(3));
22
23   }
24
25 }
26
```