SWE 4x	**Übung zu Softwareentwicklung mit modernen Plattformen 4**	**SS 2015, Übung 5**

Abgabetermin: SA in der KW 22

Gr. 1, E. Pitzer
Gr. 2, F. Gruber-Leitner

Name Roman Lumetsberger **Aufwand in h** 10

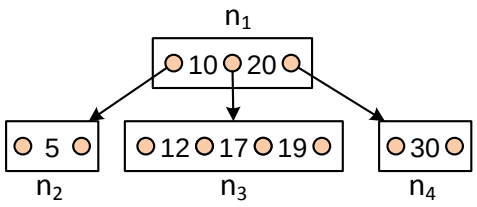
Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

2-3-4-Bäume (6 + 18 Punkte)

Mengen *(sets)* und Wörterbücher *(dictionaries oder maps)* sind in der Praxis häufig benötigte Behälterklassen. Sie sind daher auch in jedem ernst zu nehmenden Behälter-Framework enthalten (so auch im JDK). Sollen die Elemente in sortierter Reihenfolge gehalten werden, werden zur Realisierung dieser Behältertypen meistens binäre Suchbäume eingesetzt. Die in der Übung behandelte Implementierung eines binären Suchbaums hat leider den Nachteil, dass der Baum zu einer linearen Liste entarten kann. Das hat zur Konsequenz, dass alle Operationen auf dem Suchbaum nicht mehr logarithmische, sondern lineare Laufzeitkomplexität aufweisen.

Diesem Problem kann man beikommen, indem man den Suchbaum bei jeder Einfüge- und Löschoperation ausbalanziert. Ein Baum ist balanziert, wenn der linke und der rechte Unterbaum im Wesentlichen dieselbe Höhe aufweisen und diese Eigenschaft auch für die Unterbäume der Unterbäume gilt.

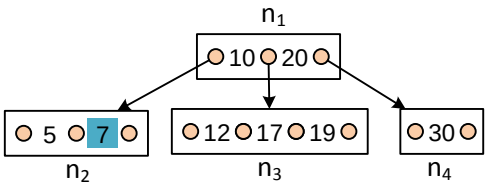
Mit so genannten 2-3-4-Bäumen lassen sich alle Baumoperationen so realisieren, dass der Baum immer ausbalanziert bleibt. Im Gegensatz zu Binärbäumen, bei denen jeder Knoten zwei Zeiger auf die Nachfolgerknoten aufweisen kann, können 2-3-4-Bäume Knoten mit zwei, drei oder vier Zeigern auf Nachfolgerknoten besitzen (siehe nachfolgende Abbildung).



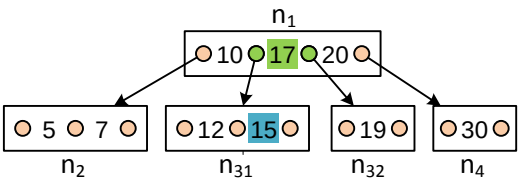
In jedem Knoten des Baums werden bis zu drei aufsteigend sortierte Schlüssel gespeichert. Damit in derartigen Bäumen effizient gesucht werden kann, sind die Elemente in Unterbäumen eines Knotens folgendermaßen angeordnet: Alle Schlüssel im ersten Unterbaum (jener, welcher am weitesten links liegt) sind kleiner als er erste Schlüsselwert, alle Schlüssel im zweiten Unterbaum sind größer oder gleich wie der erste, aber kleiner als der zweite Schlüssel, usw.

Die Suche nach einem Element in einem 2-3-4-Baum kann daher folgendermaßen implementiert werden: Zunächst wird in den Schlüsseln des Wurzelknotens nach dem Element gesucht. Wird dieses hier nicht gefunden, wird ermittelt, zwischen welchen Schlüsselwerten sich das Element befindet und die Suche beim entsprechenden Nachfolgerknoten fortgesetzt. Dies wird so lange wiederholt bis man das Element gefunden hat oder die Suche erfolglos bei einem Blatt des Baumes abgebrochen werden muss.

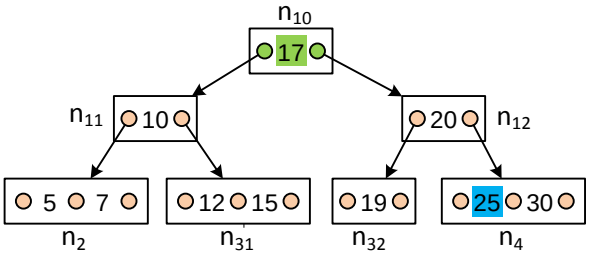
Das Einfügen eines neuen Elements gestaltet sich hingegen etwas komplizierter. Da neue Elemente nur in Blättern eingefügt werden, muss zunächst mit der oben beschriebenen Suchstrategie jenes Blatt bestimmt werden, in welches das Element gehört. In dieses Blatt wird das Element sortiert eingefügt, sofern es sich beim Blatt um einen 2- oder 3-Knoten handelt. Will man beispielsweise in den obigen Baum das Element 7 einfügen, so kann dieses problemlos in den Knoten n_2 aufgenommen werden:



Handelt es sich hingegen beim betroffenen Blatt um einen 4-Knoten, muss dieser Knoten vor dem Einfügen in zwei 2-Knoten aufgespalten werden. Dazu wird der mittlere der drei Schlüssel im Vorgängerknoten eingefügt und aus dem linken und rechten Schlüssel zwei 2-Knoten gebildet, die an den passenden Stellen in den Vorgängerknoten gehängt werden. Ein Beispiel soll dieses Vorgehen verdeutlichen. Will man in obigen Baum das Element 15 einfügen, so muss dies im Knoten n_3 erfolgen. Da dieser bereits vollständig aufgefüllt ist, wird der mittlere Schlüssel 17 in den Vorgängerknoten n_1 verschoben und n_3 in n_{31} und n_{32} zerlegt. Anschließend wird 15 in n_{31} eingefügt.



Durch das Aufteilen eines Knotens und dem damit verbundenen Einfügen eines neuen Wertes in den Vorgängerknoten könnte auch dieser überlaufen. Um dies von Vornherein zu verhindern, werden beim Durchwandern des Baums von der Wurzel bis zum Blatt, in das eingefügt werden soll, alle angetroffenen 4-Knoten aufgeteilt. Soll beispielsweise im obigen Baum 25 eingefügt werden, muss zunächst der Wurzelknoten n_1 in die 2-Knoten n_{11} und n_{12} geteilt werden. Da der Wurzelknoten keine Vorgänger hat, muss für den mittleren Schlüssel ein neuer Wurzelknoten n_{10} geschaffen werden:



Ihre Aufgabe ist es nun, zwei Implementierungen für das Interface `SortedTreeSet<T>` sowie deren Basisinterfaces `SortedSet<T>` und `Iterable<T>` zu erstellen:

```
package swe4.collections;

public interface SortedSet<T> extends Iterable<T> {
  boolean        add(T elem);      // Fügt elem in den Set ein, falls elem noch nicht
                                   // im Set enthalten war. In diesem Fall wird
                                   // true zurückgegeben. Sonst false.
  T              get(T elem);      // Gibt eine Referenz auf das Element im Set
                                   // zurück das gleich zu elem ist und null, wenn
                                   // ein derartiges Element nicht existiert.
  boolean        contains(T elem); // Gibt zurück, ob ein zu elem gleiches Element
                                   // im Set existiert.
  int            size();           // Gibt die Anzahl der Element im Set zurück.
  T              first();          // Gibt das kleinstes Element im Set zurück.
  T              last();           // Gibt das größtes Element im Set zurück.
  Comparator<T>  comparator();     // Liefert den Comparator oder null, wenn
                                   // „natürliche Sortierung“ verwendet wird.
  Iterator<T>    iterator();
}

public interface SortedTreeSet<T> extends SortedSet<T> {
  int height();                    // Gibt die Höhe des Baums zurück.
}
```

Beachten Sie, dass Implementierungen von `SortedSet<T>` Mengen im mathematischen Sinne realisieren, d. h., dass gleiche Elemente nur einmal in der Menge enthalten sein dürfen. Die Methode `add()` gibt daher auch zurück, ob ein Element eingefügt worden ist (`true`) oder ob es sich bereits in der Menge befunden hat (`false`).

a) Implementieren Sie zunächst die Klasse `BSTSet<T>`, welche die gegebenen Interfaces in Form eines binären Suchbaums realisiert. Passen Sie dazu den in der Übung erstellten binären Suchbaums so an, dass die angeführten Anforderungen erfüllt sind und ergänzen Sie die noch fehlenden Operationen.

b) Implementieren Sie die Klasse `TwoThreeFourTreeSet<T>` unter Verwendung eines 2-3-4-Baums als interne Datenstruktur.

Die beiden Klassen müssen einen Standard-Konstruktor und einen Konstruktor, an den ein Vergleichsobjekt übergeben werden kann, das `java.util.Comparator<T>` implementiert, zur Verfügung stellen. Wird ein Vergleichsobjekt übergeben, wird dieses zum Vergleichen von Elementen herangezogen. Ist kein Vergleichsobjekt vorhanden, wird angenommen, dass die eingefügten Elemente das Interface `Comparable<T>` unterstützen und der Vergleich auf dieser Basis durchgeführt („natürliche Sortierung“).

Testen Sie Ihre Implementierung ausführlich. Auf der Lernplattform stehen Ihnen die Klassen `TwoThreeFourTreeSetTest`, `TwoThreeFourTreeSetTest` und ihre Basisklasse `SortedTreeSetTestBase` zur Verfügung, die Unittests enthalten, welche die Korrektheit Ihrer Implementierung überprüfen. Ihre Implementierung muss diese Tests bestehen. Erweitern Sie die Testsuite um zumindest 10 weitere sinnvolle Testfälle, die sich signifikant von den bestehenden Tests unterscheiden.

1 Binäre Suchbäume

1.1 Lösungsidee

Für die Implementierung des *SortedTreeSet* als binären Suchbaum muss die Lösung der Übung nur um die geforderten Methoden erweitert werden.
Beim Ausführen der Testfälle wurde festgestellt, dass bei großen Datenmengen eine *StackOverflowException* geworfen wird. Dies hatte zur Folge, dass die Implementierung der Methode *get* von rekursiv auf iterativ umgebaut werden musste.

1.1.1 Höhe des Baums

Laut Definition ist die Höhe eines Baumes die Anzahl der Kanten von jenem Knoten, der am weitesten von der Wurzel entfernt ist, bis zur Wurzel. D.h.:

- Ein leerer Baum hat Höhe -1.
- Ein Baum mit nur einem Knoten hat Höhe 0.
- ...

Die Höhe des Suchbaums kann gleich beim Einfügen mitgerechnet werden und erfordert somit keine spezielle Implementierung.

2 2-3-4 Bäume

2.1 Lösungsidee

Die grundsätzliche Idee eines **2-3-4 Baumes** ist bereits in der Angabe beschrieben und wird hier nicht mehr extra angeführt.
Diese Implementierung benötigt eine eigene *Node* Klasse, die Datenkomponenten aufnehmen kann.

- Liste von Werten (max 3).
- Liste der Kindknoten (max 4).

2.1.1 Einfügen (*add*)

Beim Einfügen wird der Baum durchlaufen, um ein Blatt zu finden, indem der Wert eingefügt werden kann.
Dabei wird der Wert immer mit dem in den Knoten gespeicherten Werten verglichen und somit der richtige Kindknoten bestimmt.
Bei dieser Art des Einfügens in einen **2-3-4 Baum** ist zu beachten, dass jene Konten, die schon 3 Werte gespeichert haben, gleich beim Besuchen aufgespalten werden.

2.1.2 Suchen (*get*)

Um einen Wert in einem **2-3-4 Baum** zu suchen, muss der Baum durchlaufen werden und bei jedem Knoten anhand eines Vergleichs der richtige Kindknoten ermittelt werden.

2.1.3 Iterator

Beim *Iterator* müssen alle Knoten und deren Werte und Kinder in der korrekten Reihenfolge durchlaufen werden.
Eine einfache Möglichkeit dies zu bewerkstelligen ist es, eine Liste der Werte rekursiv zu ermitteln und dann dessen *Iterator* nach außen weiterzugeben.

2.1.4 Höhe des Baums

Die Höhe des Suchbaums kann auch hier gleich beim Einfügen mitgerechnet werden und erfordert somit keine spezielle Implementierung.

2.2 Sourcecode

SortedSet.java

```java
package at.lumetsnet.swe4.collections;

import java.util.Comparator;
import java.util.Iterator;

public interface SortedSet<T> extends Iterable<T> {

  /***
   * Adds an element to the set
   *
   * @param elem
   * @return true if the item was added, otherwise false
   */
  boolean add(T elem);

  /***
   * Searches for an element in the set
   *
   * @param elem
   * @return found element or null if the element was not found
   */
  T get(T elem);

  /***
   * checks if the set contains the element
   *
   * @param elem
   * @return true if element found, false if not
   */
  boolean contains(T elem);

  /***
   * Gets the size of the set
   *
   * @return
   */
  int size();

  /***
   * Gets the smallest element of the set
   *
   * @return
   */
  T first();

```

```
46   /***
47    * Gets the biggest element of the set
48    *
49    * @return
50    */
51   T last();
52
53   /***
54    * Gets the comparator
55    *
56    * @return comparator or null
57    */
58   Comparator<T> comparator();
59
60   /***
61    * Gets the iterator
62    */
63   Iterator<T> iterator();
64 }
```

SortedTreeSet.java

```
1  package at.lumetsnet.swe4.collections;
2
3  public interface SortedTreeSet<T> extends SortedSet<T> {
4    /***
5     * Gets the height of the tree
6     * height starts with 0, this means that a tree
7     * with only one item has height 0
8     *
9     * @return
10    */
11   int height();
12
13 }
```

AbstractSortedTreeSet.java

```
1  package at.lumetsnet.swe4.collections;
2
3  import java.util.Comparator;
4
5  /***
6   * Class used as base class for binary-search-tree and thwo-three-four tree
7   *
8   * @author romanlum
9   *
10  * @param <T>
```

```
11  */
12 public abstract class AbstractSortedTreeSet<T> implements SortedTreeSet<T> {
13
14   protected Comparator<T> comparator;
15   protected int size;
16   protected int level;
17
18   public AbstractSortedTreeSet(Comparator<T> comparator) {
19     this.comparator = comparator;
20     this.size = 0;
21     this.level = -1;
22   }
23
24   /***
25    * Compares two elements using either the comparator or comparable
26    *
27    * @param left
28    * @param right
29    * @return
30    */
31   protected int compareElements(T left, T right) {
32     return Util.compareElements(left, right, comparator);
33   }
34
35   /***
36    * Gets the comparator
37    *
38    * @return comparator or null
39    */
40   @Override
41   public Comparator<T> comparator() {
42     return comparator;
43   }
44
45   /***
46    * Gets the size of the set
47    *
48    * @return
49    */
50   @Override
51   public int size() {
52     return size;
53   }
54
55   /***
56    * checks if the set contains the element
57    *
58    * @param elem
59    * @return true if element found, false if not
```

```
60     */
61   @Override
62   public boolean contains(T elem) {
63     return get(elem) != null;
64   }
65
66   /***
67    * Gets the height of the tree height starts with 0, this means that a tree
68    * with only one item has height 0
69    *
70    * @return
71    */
72   @Override
73   public int height() {
74     return level;
75   }
76 }
```

BSTSet.java

```
1  package at.lumetsnet.swe4.collections;
2
3  import java.util.Comparator;
4  import java.util.Iterator;
5  import java.util.NoSuchElementException;
6  import java.util.Stack;
7
8  public class BSTSet<T> extends AbstractSortedTreeSet<T> {
9
10   /***
11    * Node helper class
12    *
13    * @author romanlum
14    *
15    * @param <T>
16    */
17   private static class Node<T> {
18     private T value;
19     private Node<T> left, right;
20
21     Node(T val, Node<T> left, Node<T> right) {
22       this.left = left;
23       this.right = right;
24       this.value = val;
25     }
26   }
27
28   /***
29    * Iterator class
```

```
30    *
31    * @author romanlum
32    *
33    * @param <T>
34    */
35   private static class BSTIterator<T> implements Iterator<T> {
36
37     private Stack<Node<T>> unvisitedParents = new Stack<>();
38
39     public BSTIterator(Node<T> root) {
40       Node<T> next = root;
41       while (next != null) {
42         unvisitedParents.push(next);
43         next = next.left;
44       }
45     }
46
47     @Override
48     public boolean hasNext() {
49       return !unvisitedParents.isEmpty();
50     }
51
52     @Override
53     public T next() {
54       if (!hasNext()) {
55         throw new NoSuchElementException("Stack is empty");
56       }
57
58       Node<T> cur = unvisitedParents.pop();
59       Node<T> next = cur.right;
60       while (next != null) {
61         unvisitedParents.add(next);
62         next = next.left;
63       }
64       return cur.value;
65     }
66
67   }
68
69   private Node<T> root;
70
71   public BSTSet() {
72     this(null);
73   }
74
75   public BSTSet(Comparator<T> comparator) {
76     super(comparator);
77     root = null;
78     level = -1;
```

```
79    }
80
81    @Override
82    public Iterator<T> iterator() {
83      return new BSTIterator<>(root);
84    }
85
86    /***
87     * Adds an element to the set
88     *
89     * @param elem
90     * @return true if the item was added, otherwise false
91     */
92    @Override
93    public boolean add(T elem) {
94      int curLevel = -1;
95
96      Node<T> newNode = new Node<>(elem, null, null);
97
98      if (root == null) {
99        root = newNode;
100     } else {
101       Node<T> current = root;
102       curLevel++;
103       while (current != null) {
104         int cmpResult = compareElements(current.value, elem);
105         if (cmpResult == 0)
106           return false; // duplicate element
107
108         if (cmpResult > 0) {
109           if (current.left == null) {
110             current.left = newNode;
111             break;
112           } else {
113             current = current.left;
114           }
115         } else {
116           if (current.right == null) {
117
118             current.right = newNode;
119             break;
120           } else {
121             current = current.right;
122           }
123         }
124         curLevel++;
125       }
126     }
127     size++;
```

```
128       curLevel++;
129
130       if (curLevel > level)
131         level = curLevel; // update current level
132       return true;
133     }
134
135     /***
136      * Searches for an element in the set
137      *
138      * @param elem
139      * @return found element or null if the element was not found
140      */
141     @Override
142     public T get(T elem) {
143       Node<T> t = root;
144       while (t != null) {
145         int cmpRes = compareElements(t.value, elem);
146         if (cmpRes == 0) {
147           return t.value;
148         } else if (cmpRes > 0) {
149           t = t.left;
150         } else {
151           t = t.right;
152         }
153       }
154       return null;
155     }
156
157     /***
158      * Gets the smallest element of the set
159      *
160      * @return
161      */
162     @Override
163     public T first() {
164       if (root == null) {
165         throw new NoSuchElementException("Set is empty");
166       }
167       Node<T> tmp = root;
168       while (tmp.left != null) {
169         tmp = tmp.left;
170       }
171       return tmp.value;
172     }
173
174     /***
175      * Gets the biggest element of the set
176      *
```

```
177     * @return
178     */
179   @Override
180   public T last() {
181     if (root == null) {
182       throw new NoSuchElementException("Set is empty");
183     }
184     Node<T> tmp = root;
185     while (tmp.right != null) {
186       tmp = tmp.right;
187     }
188     return tmp.value;
189   }
190 
191 }
```

TTFNode.java

```
1  package at.lumetsnet.swe4.collections;
2  
3  import java.util.ArrayList;
4  import java.util.Comparator;
5  import java.util.List;
6  import java.util.NoSuchElementException;
7  
8  /***
9   * Node class used for 2-3-4 tree
10  *
11  * @author romanlum
12  *
13  * @param <T>
14  */
15 public class TTFNode<T> {
16   private ArrayList<T> values;
17   private ArrayList<TTFNode<T>> children;
18   private Comparator<T> comparator;
19   private TTFNode<T> parent;
20 
21   public TTFNode(Comparator<T> comparator, TTFNode<T> parent) {
22     this.values = new ArrayList<>(3);
23     this.children = new ArrayList<>(4);
24     this.comparator = comparator;
25     this.parent = parent;
26   }
27 
28   public TTFNode(Comparator<T> comparator, TTFNode<T> parent, T value) {
29     this(comparator, parent);
30     this.values.add(0, value);
31   }
```

```
32
33  public TTFNode(Comparator<T> comparator, TTFNode<T> parent, T value,
34      TTFNode<T> left, TTFNode<T> right) {
35    this(comparator, parent);
36    this.values.add(0, value);
37    // update parent
38    if (left != null) {
39      left.parent = this;
40      children.add(0, left);
41    }
42    // update parent
43    if (right != null) {
44      right.parent = this;
45      children.add(1, right);
46    }
47  }
48
49  /***
50   * Returns all the values used for the iterator
51   *
52   * @param iteratorList
53   */
54  void getValues(List<T> iteratorList) {
55    // Check if we have children
56    if (children.size() != 0) {
57      for (int i = 0; i < values.size(); i++) {
58        // add child values
59        children.get(i).getValues(iteratorList);
60        // add node value
61        iteratorList.add(values.get(i));
62      }
63      children.get(children.size() - 1).getValues(iteratorList);
64    } else {
65      iteratorList.addAll(values);
66    }
67  }
68
69  /***
70   * Gets if the node is full (4-Node)
71   *
72   * @return
73   */
74  public boolean isFull() {
75    return values.size() == 3;
76  }
77
78  /**
79   * Gets if the node has children
80   *
```

```
81    * @return
82    */
83   public boolean hasChildren() {
84     return children.size() != 0;
85   }
86
87   /***
88    * Gets the child in which the element should be in
89    *
90    * @param elem
91    * @return
92    */
93   public TTFNode<T> getChild(T elem) {
94     if (children.size() == 0)
95       return null;
96     return (children.get(getChildIndex(elem)));
97   }
98
99   /***
100   * splits the node
101   *
102   * @return
103   */
104  public TTFNode<T> split() {
105    TTFNode<T> tmpParent = parent;
106    if (tmpParent == null) {
107      // create new parent (used for splitting root)
108      tmpParent = new TTFNode<T>(comparator, null);
109    }
110    // add the value to the node
111    tmpParent.addValue(values.get(1));
112
113    TTFNode<T> left = new TTFNode<T>(comparator, tmpParent, values.get(0),
114        getChildByPosition(0), getChildByPosition(1));
115
116    TTFNode<T> right = new TTFNode<T>(comparator, tmpParent, values.get(2),
117        getChildByPosition(2), getChildByPosition(3));
118
119    // get the correct child index
120    int childIndex = tmpParent.getChildIndex(this.values.get(0));
121    if (childIndex < tmpParent.children.size())
122      tmpParent.children.remove(childIndex); // remove old child
123    // insert childs
124    tmpParent.children.add(childIndex, right);
125    tmpParent.children.add(childIndex, left);
126
127    return tmpParent;
128  }
129
```

```
130   /***
131    * Gets the element
132    *
133    * @param elem
134    * @return
135    */
136   public T get(T elem) {
137     int idx = values.indexOf(elem);
138     if (idx == -1)
139       throw new NoSuchElementException("Element " + elem + " not found");
140
141     return values.get(idx);
142   }
143
144   /***
145    * Checks if the node contains the value
146    *
147    * @param elem
148    * @return
149    */
150   public boolean contains(T elem) {
151     return values.contains(elem);
152   }
153
154   /***
155    * Adds a value to the node
156    *
157    * @param value
158    */
159   public void addValue(T value) {
160     values.add(value);
161     // sort the values
162     values.sort(comparator);
163   }
164
165   /***
166    * Gets the parent
167    *
168    * @return
169    */
170   public TTFNode<T> getParent() {
171     return parent;
172   }
173
174   /***
175    * Gets the first value
176    *
177    * @return
178    */
```

```
179   public T getFirstValue() {
180     if (values.isEmpty()) {
181       throw new NoSuchElementException("Node is empty");
182     }
183     return values.get(0);
184   }
185
186   /***
187    * Gets the last value according to node size
188    *
189    * @return
190    */
191   public T getLastValue() {
192     if (values.isEmpty()) {
193       throw new NoSuchElementException("Node is empty");
194     }
195     return values.get(values.size() - 1);
196   }
197
198   /***
199    * Gets the first child
200    *
201    * @return
202    */
203   public TTFNode<T> getFirstChild() {
204     if (children.isEmpty())
205       return null;
206     return children.get(0);
207   }
208
209   /***
210    * Gets the last child
211    *
212    * @return
213    */
214   public TTFNode<T> getLastChild() {
215     if (children.isEmpty())
216       return null;
217     return children.get(children.size() - 1);
218   }
219
220   /***
221    * gets the child if available
222    *
223    * @param index
224    * @return
225    */
226   private TTFNode<T> getChildByPosition(int index) {
227     if (index < children.size())
```

```
228       return children.get(index);
229     return null;
230   }
231 
232   /***
233    * calculates the correct child index for the given element
234    *
235    * @param elem
236    * @return
237    */
238   private int getChildIndex(T elem) {
239     if (Util.compareElements(values.get(0), elem, comparator) > 0)
240       return 0;
241     else if (values.size() == 1
242         || Util.compareElements(values.get(1), elem, comparator) > 0)
243       return 1;
244     else if (values.size() == 2
245         || Util.compareElements(values.get(2), elem, comparator) > 0)
246       return 2;
247     else
248       return 3;
249   }
250 }
```

TwoThreeFourTreeSet.java

```
1  package at.lumetsnet.swe4.collections;
2  
3  import java.util.ArrayList;
4  import java.util.Comparator;
5  import java.util.Iterator;
6  import java.util.List;
7  import java.util.NoSuchElementException;
8  
9  public class TwoThreeFourTreeSet<T> extends AbstractSortedTreeSet<T> {
10 
11   private TTFNode<T> root;
12 
13   /***
14    * Constructor with natural sorting
15    */
16   public TwoThreeFourTreeSet() {
17     this(null);
18   }
19 
20   /***
21    * Constructor with special sorting
22    *
23    * @param comparator
```

```
24     */
25    public TwoThreeFourTreeSet(Comparator<T> comparator) {
26      super(comparator);
27      root = null;
28    }
29
30    /***
31     * Adds an element to the set
32     *
33     * @param elem
34     * @return true if the item was added, otherwise false
35     */
36    @Override
37    public boolean add(T elem) {
38      if (root == null) {
39        root = new TTFNode<>(comparator, null, elem);
40        size++;
41        level = 0;
42        return true;
43      }
44
45      int currentLevel = 0;
46      TTFNode<T> tmp = root;
47      while (tmp != null) {
48
49        // split if full
50        if (tmp.isFull()) {
51          TTFNode<T> result = tmp.split();
52          if (result.getParent() == null) {
53            root = result; // set new root
54            currentLevel = 0;
55          } else {
56            currentLevel--;
57          }
58          tmp = result;
59          // we start again one level above
60
61        } else {
62          // element already added
63          if (tmp.contains(elem))
64            return false;
65
66          if (tmp.hasChildren()) {
67            tmp = tmp.getChild(elem);
68            currentLevel++;
69          } else {
70            tmp.addValue(elem);
71            size++;
72            if (currentLevel > level)
```

```
73              level = currentLevel;
74            return true;
75          }
76        }
77      }
78      return false;
79    }
80
81    /***
82     * Searches for an element in the set
83     *
84     * @param elem
85     * @return found element or null if the element was not found
86     */
87    @Override
88    public T get(T elem) {
89      TTFNode<T> tmp = root;
90      while (tmp != null) {
91        if (tmp.contains(elem)) {
92          return tmp.get(elem);
93        }
94        tmp = tmp.getChild(elem);
95      }
96      return null;
97    }
98
99    /***
100    * Gets the smallest element of the set
101    *
102    * @return
103    */
104   @Override
105   public T first() {
106     if (root == null) {
107       throw new NoSuchElementException("Set is empty");
108     }
109     TTFNode<T> tmp = root;
110     while (tmp != null && tmp.getFirstChild() != null) {
111       tmp = tmp.getFirstChild();
112     }
113     return tmp.getFirstValue();
114   }
115
116   /***
117    * Gets the biggest element of the set
118    *
119    * @return
120    */
121   @Override
```

```
122   public T last() {
123     if (root == null) {
124       throw new NoSuchElementException("Set is empty");
125     }
126     TTFNode<T> tmp = root;
127     while (tmp != null && tmp.getLastChild() != null) {
128       tmp = tmp.getLastChild();
129     }
130     return tmp.getLastValue();
131   }
132 
133   /***
134    * Gets the iterator
135    */
136   @Override
137   public Iterator<T> iterator() {
138     List<T> iteratorList = new ArrayList<T>();
139     if (root != null)
140       root.getValues(iteratorList);
141     return iteratorList.iterator();
142   }
143 }
```

Util.java

```
1  package at.lumetsnet.swe4.collections;
2  
3  import java.util.Comparator;
4  
5  public class Util {
6    /***
7     * compares the elements using the comparator if not null otherwise it is
8     * assumed that T implements comparable
9     *
10    * @param left
11    * @param right
12    * @param comparator
13    * @return
14    */
15   @SuppressWarnings("unchecked")
16   public static <T> int compareElements(T left, T right,
17       Comparator<T> comparator) {
18 
19     if (comparator != null) {
20       return comparator.compare(left, right);
21     }
22 
23     if (!(left instanceof Comparable<?>)) {
24       throw new IllegalArgumentException("Elements are not comparable");
```

```
25      }
26      return ((Comparable<T>) left).compareTo(right);
27    }
28  }
```

2.3 Test - Sourcecode

SortedTreeSetTestBase.java

```
1  package at.lumetsnet.swe4.collections.test;
2
3  import java.util.Comparator;
4  import java.util.Iterator;
5  import java.util.NoSuchElementException;
6  import java.util.Random;
7
8  import org.junit.Test;
9
10 import at.lumetsnet.swe4.collections.SortedSet;
11 import at.lumetsnet.swe4.collections.SortedTreeSet;
12 import static org.junit.Assert.*;
13
14 public abstract class SortedTreeSetTestBase {
15
16   protected abstract <T> SortedTreeSet<T> createSet(Comparator<T> comparator);
17
18   protected <T> SortedTreeSet<T> createSet() {
19     return createSet(null);
20   }
21
22   @Test
23   public void testSizeSimple() {
24     SortedSet<Integer> set = createSet();
25     set.add(1);
26     assertEquals(1, set.size());
27
28     set.add(2);
29     assertEquals(2, set.size());
30     set.add(3);
31     assertEquals(3, set.size());
32     set.add(4);
33     assertEquals(4, set.size());
34   }
35
36   @Test
37   public void testAddSimple() {
38     SortedSet<Integer> set = createSet();
39
```

```
40      set.add(2);
41      assertTrue(set.contains(2));
42      set.add(1);
43      assertTrue(set.contains(1));
44      set.add(3);
45      assertTrue(set.contains(3));
46
47      assertTrue(set.contains(1));
48      assertTrue(set.contains(2));
49    }
50
51    @Test
52    public void testSize() throws Exception {
53      final int NELEMS = 100;
54      SortedSet<Integer> set = createSet();
55      for (int i = 1; i <= NELEMS; i++) {
56        set.add(i);
57        assertEquals(i, set.size());
58      }
59    }
60
61    @Test
62    public void testLinearAdd() {
63      final int NELEMS = 10000;
64      SortedSet<Integer> set = createSet();
65      for (int i = 0; i < NELEMS; i++)
66        set.add(i);
67
68      for (int i = 0; i < NELEMS; i++)
69        assertTrue(set.contains(i));
70
71      for (int i = NELEMS; i < NELEMS + 100; i++)
72        assertFalse(set.contains(i));
73    }
74
75    @Test
76    public void testRandomAdd() {
77      final int NELEMS = 100000;
78      Random rand = new Random();
79      SortedSet<Integer> set = createSet();
80
81      int[] numbers = new int[NELEMS];
82      for (int i = 0; i < numbers.length; i++)
83        numbers[i] = rand.nextInt();
84
85      for (int i = 0; i < NELEMS; i++)
86        set.add(numbers[i]);
87
88      for (int i = 0; i < NELEMS; i++)
```

```
89        assertTrue(set.contains(numbers[i]));
90    }
91
92    @Test
93    public void testSorted() {
94      final int NELEMS = 1000;
95      Random rand = new Random();
96      SortedSet<Integer> set = createSet();
97
98      for (int i = 1; i <= NELEMS; i++) {
99        set.add(rand.nextInt());
100       assertTrue(isSorted(set));
101       assertEquals(i, set.size());
102     }
103   }
104
105   @Test
106   public void testAddMultipleSimple() {
107     SortedSet<Integer> set = createSet();
108
109     assertTrue(set.add(10));
110     assertTrue(set.add(15));
111
112     assertFalse(set.add(10));
113     assertFalse(set.add(15));
114
115     assertTrue(set.add(5));
116     assertFalse(set.add(5));
117
118     assertEquals(3, set.size());
119   }
120
121   @Test
122   public void testAddMultiple() {
123     final int NELEMS = 1000;
124
125     SortedSet<Integer> set = createSet();
126     for (int i = 0; i < NELEMS; i++)
127       assertTrue(set.add(i));
128
129     int size = set.size();
130
131     for (int i = 0; i < NELEMS; i++)
132       assertFalse(set.add(i));
133
134     assertEquals(size, set.size());
135   }
136
137   @Test
```

```
138   public void testGetSimple() {
139     SortedSet<Integer> set = createSet();
140
141     assertNull(set.get(5));
142     set.add(5);
143     assertEquals(5, set.get(5).intValue());
144     assertNull(set.get(99));
145   }
146
147   @Test
148   public void testGet() {
149     final int NELEMS = 1000;
150
151     SortedSet<Integer> set = createSet();
152     for (int i = 0; i < NELEMS; i++)
153       set.add(i);
154
155     for (int i = 0; i < NELEMS; i++)
156       assertEquals(i, set.get(i).intValue());
157   }
158
159   @Test
160   public void testContainsSimple() {
161     SortedSet<Integer> set = createSet();
162     set.add(3);
163     set.add(1);
164     set.add(5);
165
166     assertTrue(set.contains(1));
167     assertTrue(set.contains(3));
168     assertTrue(set.contains(5));
169
170     assertFalse(set.contains(0));
171     assertFalse(set.contains(2));
172     assertFalse(set.contains(4));
173     assertFalse(set.contains(6));
174   }
175
176   @Test
177   public void testContains() {
178     final int NELEMS = 1000;
179
180     SortedSet<Integer> set = createSet();
181     for (int i = 0; i < NELEMS; i++)
182       set.add(i);
183
184     for (int i = 0; i < NELEMS; i++)
185       assertTrue(set.contains(i));
186   }
```

```
187
188   @Test
189   public void testIteratorSimple() {
190     SortedSet<Integer> set = createSet();
191     set.add(10);
192     set.add(5);
193     set.add(15);
194
195     Iterator<Integer> it = set.iterator();
196
197     assertTrue(it.hasNext());
198     assertEquals(new Integer(5), it.next());
199
200     assertTrue(it.hasNext());
201     assertEquals(new Integer(10), it.next());
202
203     assertTrue(it.hasNext());
204     assertEquals(new Integer(15), it.next());
205
206     assertFalse(it.hasNext());
207   }
208
209   @Test
210   public void testIterator() {
211     final int NELEMS = 10000;
212     SortedSet<Integer> set = createSet();
213
214     for (int i = 1; i <= NELEMS; i++)
215       set.add(i);
216
217     Iterator<Integer> it = set.iterator();
218     int prev = it.next();
219     assertEquals(1, prev);
220
221     while (it.hasNext()) {
222       int curr = it.next();
223       assertTrue(prev + 1 == curr);
224       prev = curr;
225     }
226   }
227
228   @Test(expected = NoSuchElementException.class)
229   public void testIteratorException1() {
230     SortedSet<Integer> set = createSet();
231     Iterator<Integer> it = set.iterator();
232     it.next();
233   }
234
235   @Test(expected = NoSuchElementException.class)
```

```
236   public void testIteratorException2() {
237     SortedSet<Integer> set = createSet();
238     set.add(1);
239
240     Iterator<Integer> it = set.iterator();
241     assertNotNull(it.next());
242     it.next();
243   }
244
245   @Test(expected = NoSuchElementException.class)
246   public void testFirstWithEmptySet() {
247     SortedSet<Integer> set = createSet();
248     set.first();
249   }
250
251   @Test(expected = NoSuchElementException.class)
252   public void testLastWithEmptySet() {
253     SortedSet<Integer> set = createSet();
254     set.last();
255   }
256
257   @Test
258   public void testFirstLast() {
259     final int NELEMS = 100;
260     SortedSet<Integer> set = createSet();
261     Integer min = Integer.MAX_VALUE;
262     Integer max = Integer.MIN_VALUE;
263     Random rand = new Random();
264
265     for (int i = 1; i <= NELEMS; i++) {
266       int r = rand.nextInt();
267       set.add(r);
268       min = Math.min(min, r);
269       max = Math.max(max, r);
270     }
271
272     assertEquals(min, set.first());
273     assertEquals(max, set.last());
274   }
275
276   @Test
277   public void testConstructorWithComparator() {
278     SortedSet<Integer> set1 = createSet();
279     assertNull(set1.comparator());
280     SortedSet<Integer> set2 = createSet((i1, i2) -> i1.compareTo(i2));
281     assertNotNull(set2.comparator());
282   }
283
284   @Test
```

```
285   public void testComparator() {
286     final int NELEMS = 100;
287     SortedSet<Integer> set = createSet((i1, i2) -> i2.compareTo(i1));
288     Random rand = new Random();
289
290     for (int i = 1; i <= NELEMS; i++)
291       set.add(rand.nextInt());
292
293     assertTrue(isSortedInComparatorOrder(set));
294   }
295
296   @Test
297   public void testStringFirstLast() {
298     SortedSet<String> set = createSet();
299     set.add("a");
300     set.add("b");
301     set.add("c");
302     set.add("d");
303     set.add("e");
304     assertEquals("a", set.first());
305     assertEquals("e", set.last());
306
307   }
308
309   @Test
310   public void testGetEmpty() {
311     SortedSet<Integer> set = createSet();
312     assertNull(set.get(1));
313   }
314
315   @Test
316   public void testNotContains() {
317     SortedSet<Integer> set = createSet();
318     set.add(1);
319     assertFalse(set.contains(2));
320   }
321
322   @Test(expected = NoSuchElementException.class)
323   public void TestEmptyIterator() {
324     SortedSet<Integer> set = createSet();
325     set.iterator().next();
326   }
327
328   @Test(expected = NoSuchElementException.class)
329   public void testIteratorOverflow() {
330     SortedSet<Integer> set = createSet();
331     set.add(1);
332     set.add(2);
333     set.add(3);
```

```
334     Iterator it = set.iterator();
335     assertEquals((Integer) 1, it.next());
336     assertEquals((Integer) 2, it.next());
337     assertEquals((Integer) 3, it.next());
338     it.next();
339   }
340
341   protected boolean isSorted(SortedSet<Integer> set) {
342     Iterator<Integer> it = set.iterator();
343     if (!it.hasNext())
344       return true;
345
346     int prev = it.next();
347     while (it.hasNext()) {
348       int curr = it.next();
349       if (!(prev < curr))
350         return false;
351       prev = curr;
352     }
353
354     return true;
355   }
356
357   protected boolean isSortedInComparatorOrder(SortedSet<Integer> set) {
358     Iterator<Integer> it = set.iterator();
359     if (!it.hasNext())
360       return true;
361
362     int prev = it.next();
363     while (it.hasNext()) {
364       int curr = it.next();
365       if (set.comparator().compare(prev, curr) >= 0)
366         return false;
367       prev = curr;
368     }
369
370     return true;
371   }
372
373 }
```

BSTSetTest.java

```
1 package at.lumetsnet.swe4.collections.test;
2
3 import static org.junit.Assert.assertEquals;
4
5 import java.util.Comparator;
6
```

```java
import org.junit.Test;

import at.lumetsnet.swe4.collections.BSTSet;
import at.lumetsnet.swe4.collections.SortedTreeSet;

public class BSTSetTest extends SortedTreeSetTestBase {

  @Override
  protected <T> SortedTreeSet<T> createSet(Comparator<T> comparator) {
    return new BSTSet<T>(comparator);
  }

  @Test
  public void testEmptyContructor() {
    BSTSet<Integer> set = new BSTSet<Integer>();
    assertEquals(0, set.size());
    assertEquals(-1, set.height());
  }

  @Test
  public void testHeight() {
    SortedTreeSet<Integer> set = createSet();
    set.add(2);
    assertEquals(0, set.height());
    set.add(1);
    assertEquals(1, set.height());
    set.add(0);
    assertEquals(2, set.height());
  }

  @Test
  public void testEmptyTreeHeight() {
    SortedTreeSet<Integer> set = createSet();
    assertEquals(-1, set.height());
  }

}
```

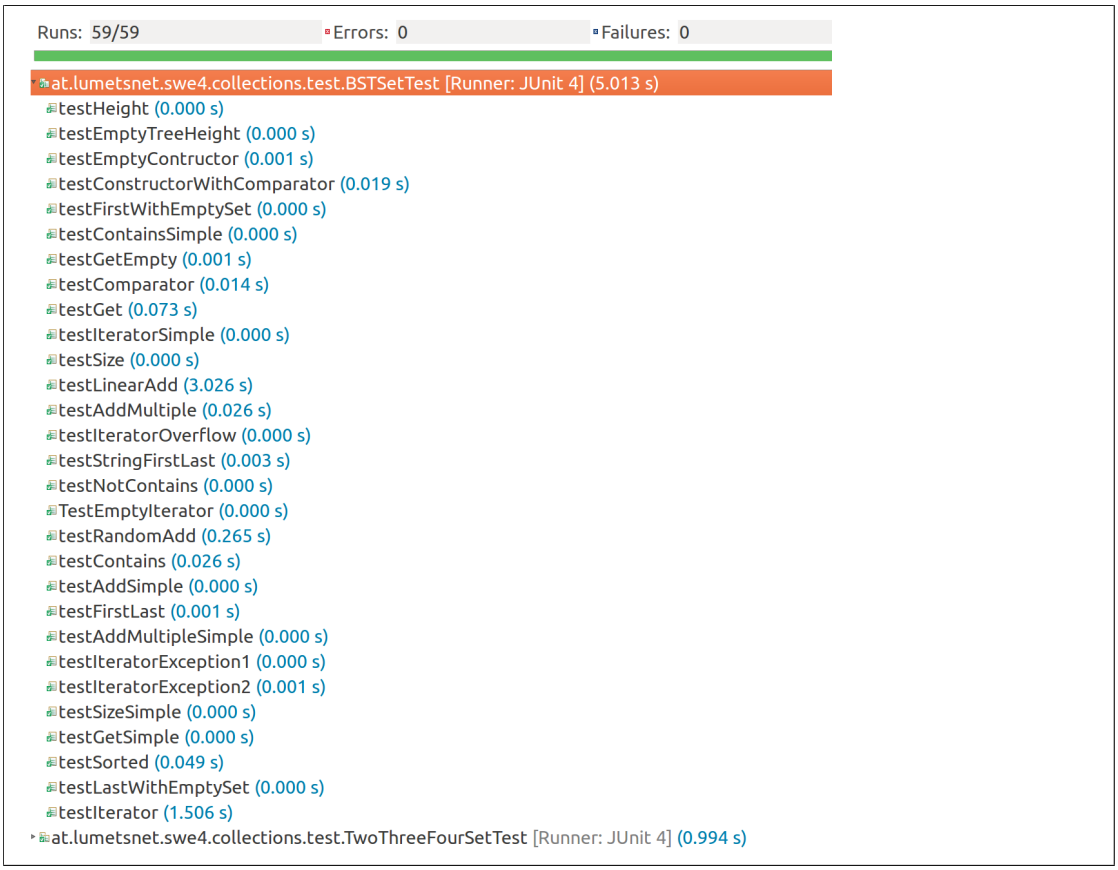
TwoThreeFourSetTest.java

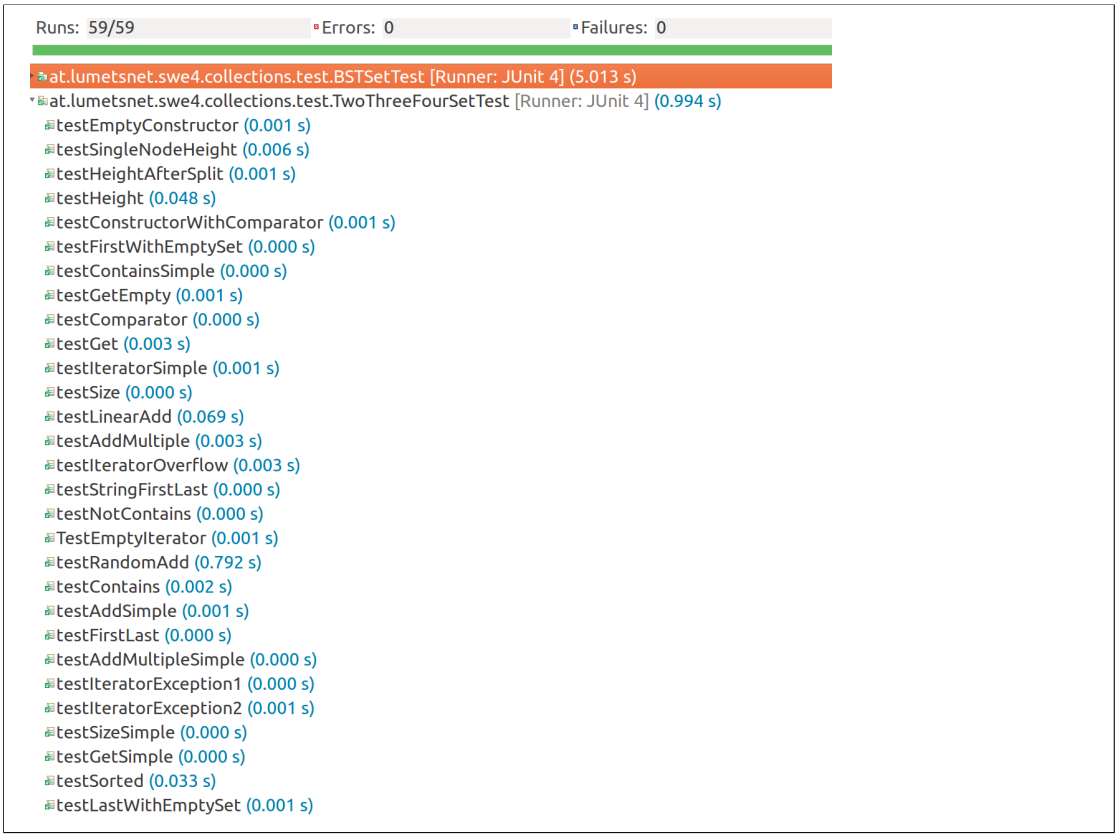
```java
package at.lumetsnet.swe4.collections.test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.util.Comparator;

import org.junit.Test;

```

```java
import at.lumetsnet.swe4.collections.SortedTreeSet;
import at.lumetsnet.swe4.collections.TwoThreeFourTreeSet;

public class TwoThreeFourSetTest extends SortedTreeSetTestBase {

  @Override
  protected <T> TwoThreeFourTreeSet<T> createSet(Comparator<T> comparator) {
    return new TwoThreeFourTreeSet<T>(comparator);
  }

  @Test
  public void testEmptyConstructor() {
    TwoThreeFourTreeSet<Integer> set = new TwoThreeFourTreeSet<>();
    assertEquals(null, set.comparator());
  }

  @Test
  public void testHeight() {
    final int NELEMS = 10000;
    SortedTreeSet<Integer> set = createSet();

    for (int i = 1; i <= NELEMS; i++) {
      set.add(i);
      int h = set.height();
      int n = set.size();
      assertTrue("height(set) <= ld(size(set))+1",
          h <= Math.log((double) n) / Math.log(2.0) + 1);
    }
  }

  @Test
  public void testSingleNodeHeight() {
    SortedTreeSet<Integer> set = createSet();
    set.add(1);
    assertEquals(0, set.height());
    assertEquals(1, set.size());
  }

  @Test
  public void testHeightAfterSplit() {
    SortedTreeSet<Integer> set = createSet();
    set.add(1);
    set.add(2);
    set.add(3);
    set.add(4); // split
    assertEquals(1, set.height());
    assertEquals(4, set.size());

  }
```

```
59
60 }
```

2.4 Testfälle




Runs: 59/59
Errors: 0
Failures: 0
at.lumetsnet.swe4.collections.test.BSTSetTest [Runner: JUnit 4] (5.013 s)
at.lumetsnet.swe4.collections.test.TwoThreeFourSetTest [Runner: JUnit 4] (0.994 s)
testEmptyConstructor (0.001 s)
testSingleNodeHeight (0.006 s)
testHeightAfterSplit (0.001 s)
testHeight (0.048 s)
testConstructorWithComparator (0.001 s)
testFirstWithEmptySet (0.000 s)
testContainsSimple (0.000 s)
testGetEmpty (0.001 s)
testComparator (0.000 s)
testGet (0.003 s)
testIteratorSimple (0.001 s)
testSize (0.000 s)
testLinearAdd (0.069 s)
testAddMultiple (0.003 s)
testIteratorOverflow (0.003 s)
testStringFirstLast (0.000 s)
testNotContains (0.000 s)
TestEmptyIterator (0.001 s)
testRandomAdd (0.792 s)
testContains (0.002 s)
testAddSimple (0.001 s)
testFirstLast (0.000 s)
testAddMultipleSimple (0.000 s)
testIteratorException1 (0.000 s)
testIteratorException2 (0.001 s)
testSizeSimple (0.000 s)
testGetSimple (0.000 s)
testSorted (0.033 s)
testLastWithEmptySet (0.001 s)