Abgabetermin: SA in der KW 22

Gr. 1, E. Pitzer **Name** _____ **Aufwand in h** _____
Gr. 2, F. Gruber-Leitner

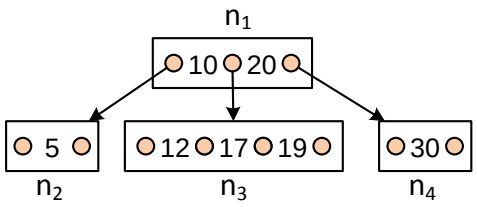
Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

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.

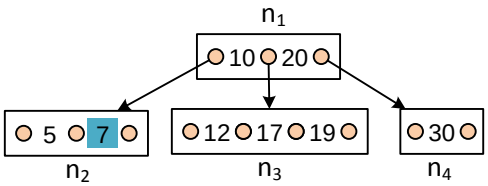
Mit so genannten 2-3-4-Bäumen lassen sich alle Baumoperationen so realisieren, dass der Baum immer ausbalanziert bleibt. Im Gegensatz zu Binärbäumen, bei denen jeder Knoten zwei Zeiger auf die Nachfolgerknoten aufweisen kann, können 2-3-4-Bäume Knoten mit zwei, drei oder vier Zeigern auf Nachfolgerknoten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



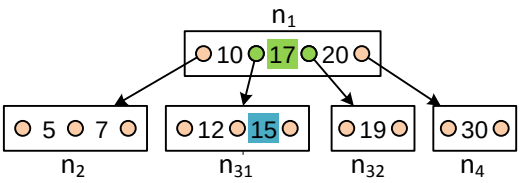
In jedem Knoten des Baums werden bis zu drei aufsteigend sortierte Schlüssel gespeichert. Damit in derartigen Bäumen effizient gesucht werden kann, sind die Elemente in Unterbäumen eines Knotens folgendermaßen angeordnet: Alle Schlüssel im ersten Unterbaum (jener, welcher am weitesten links liegt) sind 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.

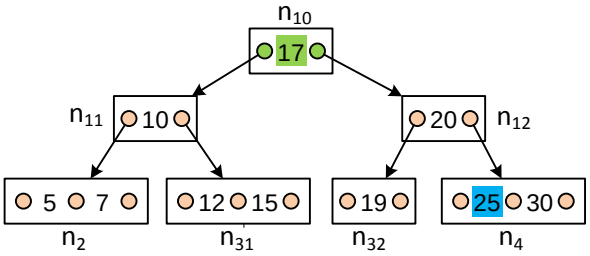
Das Einfügen eines neuen Elements gestaltet sich hingegen etwas komplizierter. Da neue Elemente nur in Blättern eingefügt werden, muss zunächst mit der oben beschriebenen Suchstrategie jenes Blatt bestimmt werden, in welches das Element gehört. In dieses Blatt wird das Element sortiert eingefügt, sofern 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 die Anzahl der Kanten von jenem Konten, 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 grunsätzliche Idee eines **2-3-4** Baumes ist bereits in der Angabe beschrieben und wird hier nicht mehr extra aufgeführt.
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 muss 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 weiterzugben.

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
   */
  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 height starts with 0, this means that a tree
6     * with only one item has height 0
7     *
8     * @return
9     */
10   int height();
11
12 }
```

AbstractSortedTreeSet.java

```
1  package at.lumetsnet.swe4.collections;
2
3  import java.util.Comparator;
4
5  public abstract class AbstractSortedTreeSet<T> implements SortedTreeSet<T> {
6
7    protected Comparator<T> comparator;
8    protected int size;
9    protected int level;
10
11   public AbstractSortedTreeSet(Comparator<T> comparator) {
```

```
12      this.comparator = comparator;
13      this.size = 0;
14      this.level = -1;
15    }
16
17    protected int compareElements(T left, T right) {
18      return Util.compareElements(left, right, comparator);
19    }
20
21    @Override
22    public Comparator<T> comparator() {
23      return comparator;
24    }
25
26    @Override
27    public int height() {
28      return level;
29    }
30
31    @Override
32    public int size() {
33      return size;
34    }
35
36    @Override
37    public boolean contains(T elem) {
38      return get(elem) != null;
39    }
40
41
42  }
```

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    * @author romanlum
13    *
14    * @param <T>
15    */
```

```
16  private static class Node<T> {
17    private T value;
18    private Node<T> left, right;
19
20    Node(T val, Node<T> left, Node<T> right) {
21      this.left = left;
22      this.right = right;
23      this.value = val;
24    }
25  }
26
27  /***
28   * Iterator class
29   * @author romanlum
30   *
31   * @param <T>
32   */
33  private static class BSTIterator<T> implements Iterator<T> {
34
35    private Stack<Node<T>> unvisitedParents = new Stack<>();
36
37    public BSTIterator(Node<T> root) {
38      Node<T> next = root;
39      while (next != null) {
40        unvisitedParents.push(next);
41        next = next.left;
42      }
43    }
44
45    @Override
46    public boolean hasNext() {
47      return !unvisitedParents.isEmpty();
48    }
49
50    @Override
51    public T next() {
52      if (!hasNext()) {
53        throw new NoSuchElementException("Stack is empty");
54      }
55
56      Node<T> cur = unvisitedParents.pop();
57      Node<T> next = cur.right;
58      while (next != null) {
59        unvisitedParents.add(next);
60        next = next.left;
61      }
62      return cur.value;
63    }
64
```

```
65  }
66
67  private Node<T> root;
68
69
70  public BSTSet() {
71    this(null);
72  }
73
74  public BSTSet(Comparator<T> comparator) {
75    super(comparator);
76    root = null;
77    level = -1;
78  }
79
80  @Override
81  public Iterator<T> iterator() {
82    return new BSTIterator<>(root);
83  }
84
85  /***
86   * Adds an element to the set
87   *
88   * @param elem
89   * @return true if the item was added, otherwise false
90   */
91  @Override
92  public boolean add(T elem) {
93    int curLevel = -1;
94
95    Node<T> newNode = new Node<>(elem, null, null);
96
97    if (root == null) {
98      root = newNode;
99    } else {
100     Node<T> current = root;
101     curLevel++;
102     while (current != null) {
103       int cmpResult = compareElements(current.value, elem);
104       if (cmpResult == 0) return false; //duplicate element
105
106       if( cmpResult > 0) {
107         if (current.left == null) {
108           current.left = newNode;
109           break;
110         } else {
111           current = current.left;
112         }
113       } else {
```

```
114           if (current.right == null) {
115
116             current.right = newNode;
117             break;
118           } else {
119             current = current.right;
120           }
121         }
122         curLevel++;
123       }
124     }
125     size++;
126     curLevel++;
127
128     if(curLevel > level)
129       level = curLevel; //update current level
130     return true;
131   }
132
133   /***
134    * Searches for an element in the set
135    *
136    * @param elem
137    * @return found element or null if the element was not found
138    */
139   @Override
140   public T get(T elem) {
141     Node<T> t = root;
142     while (t != null) {
143       int cmpRes = compareElements(t.value, elem);
144       if (cmpRes == 0) {
145         return t.value;
146       } else if (cmpRes > 0) {
147         t = t.left;
148       } else {
149         t = t.right;
150       }
151     }
152     return null;
153   }
154
155
156   /***
157    * Gets the smallest element of the set
158    *
159    * @return
160    */
161   @Override
162   public T first() {
```

```
163     if (root == null) {
164       throw new NoSuchElementException("Set is empty");
165     }
166     Node<T> tmp = root;
167     while (tmp.left != null) {
168       tmp = tmp.left;
169     }
170     return tmp.value;
171   }
172
173   /***
174    * Gets the biggest element of the set
175    *
176    * @return
177    */
178   @Override
179   public T last() {
180     if (root == null) {
181       throw new NoSuchElementException("Set is empty");
182     }
183     Node<T> tmp = root;
184     while (tmp.right != null) {
185       tmp = tmp.right;
186     }
187     return tmp.value;
188   }
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  public class TTFNode<T> {
9    private ArrayList<T> values;
10   private ArrayList<TTFNode<T>> children;
11   private Comparator<T> comparator;
12   private TTFNode<T> parent;
13
14
15   public TTFNode(Comparator<T> comparator, TTFNode<T> parent) {
16     this.values = new ArrayList<>(3);
17     this.children = new ArrayList<>(4);
```

```
18      this.comparator = comparator;
19      this.parent = parent;
20    }
21
22    public TTFNode(Comparator<T> comparator, TTFNode<T> parent, T value) {
23      this(comparator, parent);
24      this.values.add(0,value);
25    }
26
27    public TTFNode(Comparator<T> comparator, TTFNode<T> parent, T value, TTFNode<T> left,TTFNode<T>
28      this(comparator, parent);
29      this.values.add(0,value);
30      //update parent
31      if(left != null) {
32        left.parent = this;
33        children.add(0, left);
34      }
35      //update parent
36      if(right != null) {
37        right.parent = this;
38        children.add(1, right);
39      }
40    }
41
42    /***
43     * Returns all the values used for the iterator
44     * @param iteratorList
45     */
46    void getValues(List<T> iteratorList) {
47      //Check if we have children
48      if (children.size() != 0) {
49        for (int i = 0; i < values.size(); i++) {
50          //add child values
51          children.get(i).getValues(iteratorList);
52          //add node value
53          iteratorList.add(values.get(i));
54        }
55        children.get(children.size() - 1).getValues(iteratorList);
56      } else {
57        iteratorList.addAll(values);
58      }
59    }
60
61    /***
62     * Gets if the node is full (4-Node)
63     * @return
64     */
65    public boolean isFull() {
66      return values.size() == 3;
```

```
67    }
68
69    /**
70     * Gets if the node has children
71     * @return
72     */
73    public boolean hasChildren() {
74      return children.size() != 0;
75    }
76
77
78    /***
79     * Gets the child in which the element should be in
80     * @param elem
81     * @return
82     */
83    public TTFNode<T> getChild(T elem) {
84      if (children.size() == 0)
85        return null;
86      return (children.get(getChildIndex(elem)));
87    }
88
89    /***
90     * splits the node
91     * @return
92     */
93    public TTFNode<T> split() {
94      TTFNode<T> tmpParent = parent;
95      if (tmpParent == null) {
96        //create new parent (used for splitting root)
97        tmpParent = new TTFNode<T>(comparator,null);
98      }
99      //add the value to the node
100     tmpParent.addValue(values.get(1));
101
102     TTFNode<T> left = new TTFNode<T>(comparator, tmpParent, values.get(0),
103         getChildByPosition(0),
104         getChildByPosition(1));
105
106     TTFNode<T> right = new TTFNode<T>(comparator, tmpParent, values.get(2),
107         getChildByPosition(2),
108         getChildByPosition(3));
109
110     //get the correct child index
111     int childIndex = tmpParent.getChildIndex(this.values.get(0));
112     if(childIndex < tmpParent.children.size())
113       tmpParent.children.remove(childIndex); //remove old child
114     //insert childs
115     tmpParent.children.add(childIndex, right);
```

```
116     tmpParent.children.add(childIndex, left);
117
118     return tmpParent;
119   }
120
121   /***
122    * Gets the element
123    * @param elem
124    * @return
125    */
126   public T get(T elem) {
127     int idx = values.indexOf(elem);
128     if (idx == -1)
129       throw new NoSuchElementException("Element "+elem+" not found");
130
131     return values.get(idx);
132   }
133
134   /***
135    * Checks if the node contains the value
136    * @param elem
137    * @return
138    */
139   public boolean contains(T elem) {
140     return values.contains(elem);
141   }
142
143   /***
144    * Adds a value to the node
145    * @param value
146    */
147   public void addValue(T value) {
148     values.add(value);
149     //sort the values
150     values.sort(comparator);
151   }
152
153   /***
154    * Gets the parent
155    * @return
156    */
157   public TTFNode<T> getParent() {
158     return parent;
159   }
160
161   /***
162    * Gets the first value
163    * @return
164    */
```

```
165   public T getFirstValue() {
166     if(values.isEmpty()) {
167       throw new NoSuchElementException("Node is empty");
168     }
169     return values.get(0);
170   }
171 
172   /***
173    * Gets the last value according to node size
174    * @return
175    */
176   public T getLastValue() {
177     if(values.isEmpty()) {
178       throw new NoSuchElementException("Node is empty");
179     }
180     return values.get(values.size()-1);
181   }
182 
183   /***
184    * Gets the first child
185    * @return
186    */
187   public TTFNode<T> getFirstChild() {
188     if(children.isEmpty()) return null;
189     return children.get(0);
190   }
191 
192   /***
193    * Gets the last child
194    * @return
195    */
196   public TTFNode<T> getLastChild() {
197     if(children.isEmpty()) return null;
198     return children.get(children.size()-1);
199   }
200 
201   /***
202    * gets the child if available
203    * @param index
204    * @return
205    */
206   private TTFNode<T> getChildByPosition(int index) {
207     if(index < children.size())
208       return children.get(index);
209     return null;
210   }
211 
212   /***
213    * calculates the correct child index for the given element
```

```
214     * @param elem
215     * @return
216     */
217   private int getChildIndex(T elem) {
218     if (Util.compareElements(values.get(0), elem, comparator) > 0)
219       return 0;
220     else if (values.size() == 1
221         || Util.compareElements(values.get(1), elem, comparator) > 0)
222       return 1;
223     else if (values.size() == 2
224         || Util.compareElements(values.get(2), elem, comparator) > 0)
225       return 2;
226     else
227       return 3;
228   }
229 }
```

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    * @param comparator
23    */
24   public TwoThreeFourTreeSet(Comparator<T> comparator) {
25     super(comparator);
26     root = null;
27   }
28
29   /***
30    * Adds an element to the set
```

```
31    *
32    * @param elem
33    * @return true if the item was added, otherwise false
34    */
35   @Override
36   public boolean add(T elem) {
37     if(root == null) {
38       root = new TTFNode<>(comparator, null,elem);
39       size++;
40       level = 0;
41       return true;
42     }
43
44     int currentLevel = 0;
45     TTFNode<T> tmp = root;
46     while(tmp != null) {
47
48       //split if full
49       if(tmp.isFull()) {
50         TTFNode<T> result = tmp.split();
51         if(result.getParent() == null) {
52           root = result; //set new root
53           currentLevel = 0;
54         } else {
55           currentLevel--;
56         }
57         tmp = result;
58         //we start again one level above
59
60       } else {
61         //element already added
62         if(tmp.contains(elem))
63           return false;
64
65         if(tmp.hasChildren()) {
66           tmp = tmp.getChild(elem);
67           currentLevel++;
68         } else {
69           tmp.addValue(elem);
70           size++;
71           if(currentLevel > level)
72             level = currentLevel;
73           return true;
74         }
75       }
76     }
77     return false;
78   }
79
```

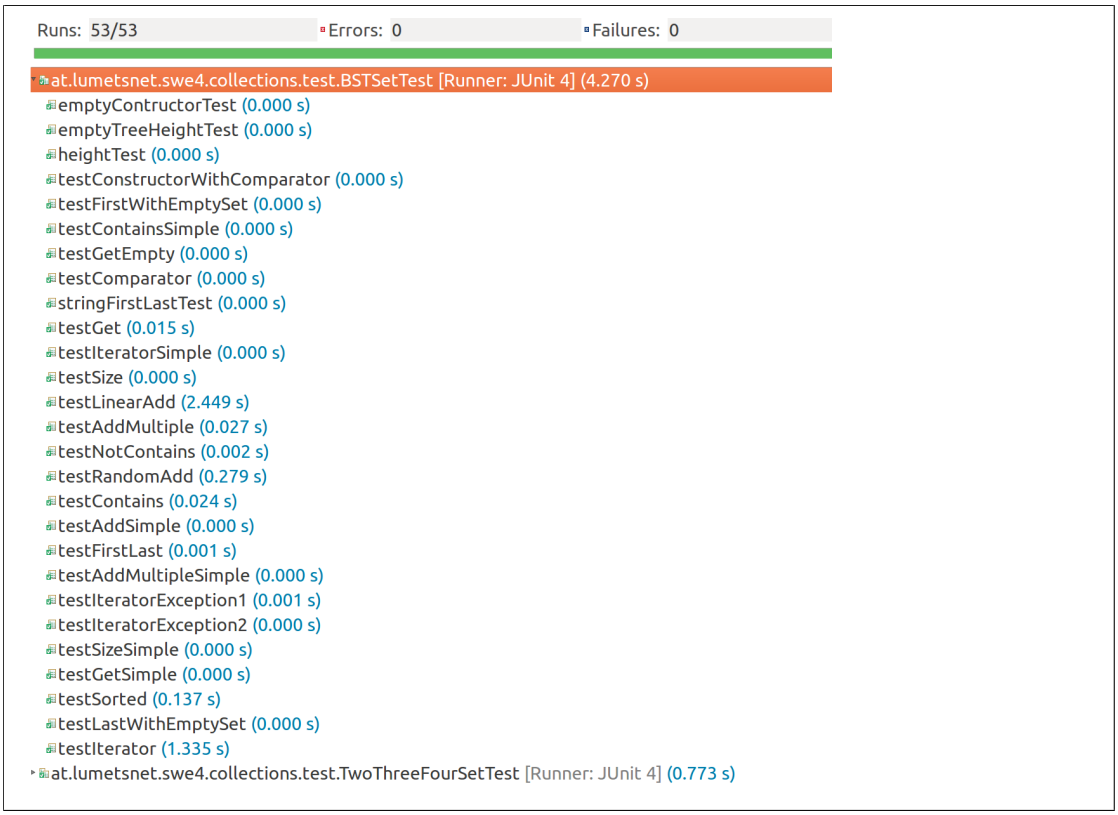
```
80  /***
81   * Searches for an element in the set
82   *
83   * @param elem
84   * @return found element or null if the element was not found
85   */
86  @Override
87  public T get(T elem) {
88    TTFNode<T> tmp = root;
89    while(tmp != null) {
90      if(tmp.contains(elem)) {
91        return tmp.get(elem);
92      }
93      tmp = tmp.getChild(elem);
94    }
95    return null;
96  }
97
98  /***
99   * Gets the smallest element of the set
100  *
101  * @return
102  */
103 @Override
104 public T first() {
105   if(root == null) {
106     throw new NoSuchElementException("Set is empty");
107   }
108   TTFNode<T> tmp = root;
109   while(tmp != null && tmp.getFirstChild() != null) {
110     tmp = tmp.getFirstChild();
111   }
112   return tmp.getFirstValue();
113 }
114
115 /***
116  * Gets the biggest element of the set
117  *
118  * @return
119  */
120 @Override
121 public T last() {
122   if(root == null) {
123     throw new NoSuchElementException("Set is empty");
124   }
125   TTFNode<T> tmp = root;
126   while(tmp != null && tmp.getLastChild() != null) {
127     tmp = tmp.getLastChild();
128   }
```

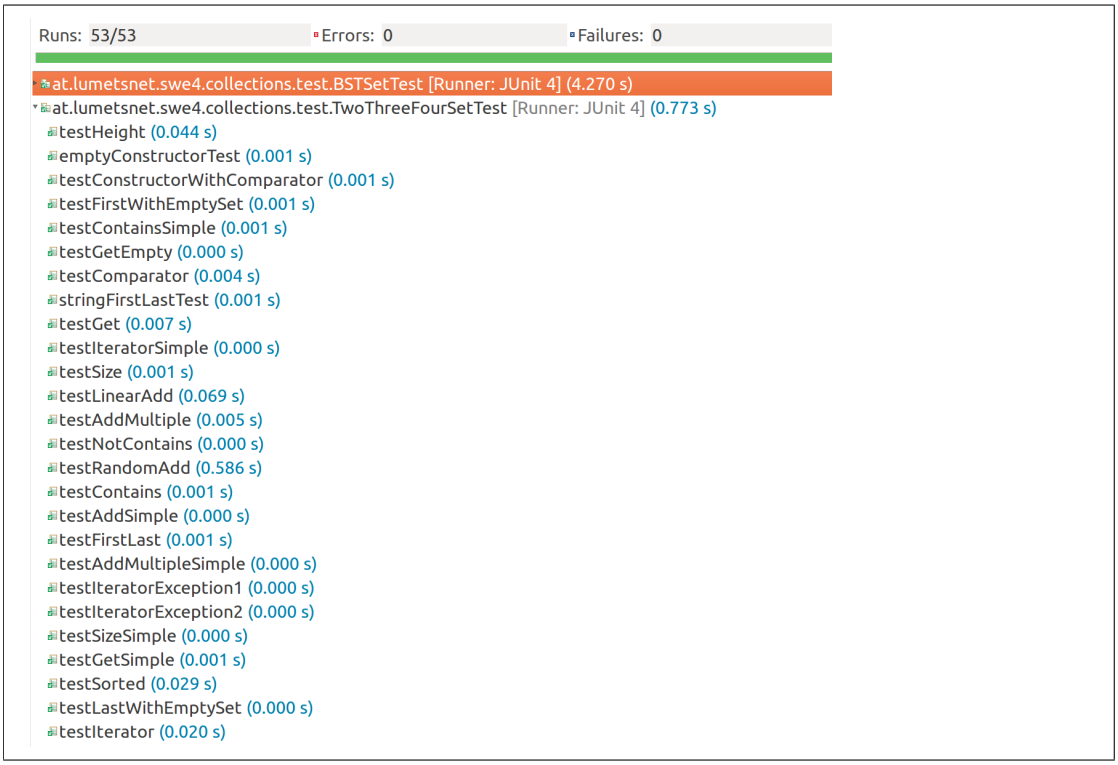
```
129     return tmp.getLastValue();
130   }
131
132   /***
133    * Gets the iterator
134    */
135   @Override
136   public Iterator<T> iterator() {
137     List<T> iteratorList = new ArrayList<T>();
138     if(root != null)
139       root.getValues(iteratorList);
140     return iteratorList.iterator();
141   }
142 }
```

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
8     * otherwise it is assumed that T implements comparable
9     * @param left
10    * @param right
11    * @param comparator
12    * @return
13    */
14   @SuppressWarnings("unchecked")
15   public static <T> int compareElements(T left, T right, Comparator<T> comparator) {
16
17     if(comparator != null) {
18       return comparator.compare(left, right);
19     }
20
21     if(!(left instanceof Comparable<?>)) {
22       throw new IllegalArgumentException("Elements are not comparable");
23     }
24     return ((Comparable<T>)left).compareTo(right);
25   }
26 }
```

2.3 Testfälle




Runs: 53/53
Errors: 0
Failures: 0
at.lumetsnet.swe4.collections.test.BSTSetTest [Runner: JUnit 4] (4.270 s)
at.lumetsnet.swe4.collections.test.TwoThreeFourSetTest [Runner: JUnit 4] (0.773 s)
testHeight (0.044 s)
emptyConstructorTest (0.001 s)
testConstructorWithComparator (0.001 s)
testFirstWithEmptySet (0.001 s)
testContainsSimple (0.001 s)
testGetEmpty (0.000 s)
testComparator (0.004 s)
stringFirstLastTest (0.001 s)
testGet (0.007 s)
testIteratorSimple (0.000 s)
testSize (0.001 s)
testLinearAdd (0.069 s)
testAddMultiple (0.005 s)
testNotContains (0.000 s)
testRandomAdd (0.586 s)
testContains (0.001 s)
testAddSimple (0.000 s)
testFirstLast (0.001 s)
testAddMultipleSimple (0.000 s)
testIteratorException1 (0.000 s)
testIteratorException2 (0.000 s)
testSizeSimple (0.000 s)
testGetSimple (0.001 s)
testSorted (0.029 s)
testLastWithEmptySet (0.000 s)
testIterator (0.020 s)