

Gr. 1, E. Pitzer

Name \_\_\_\_\_ Aufwand in h \_\_\_\_\_

Gr. 2, F. Gruber-Leitner

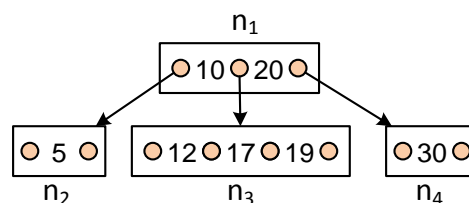
Punkte \_\_\_\_\_ Kurzzeichen Tutor / Übungsleiter \_\_\_\_\_ / \_\_\_\_\_

**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 ausbalanciert. Ein Baum ist balanciert, wenn der linke und der rechte Unterbaum im Wesentlichen dieselbe Höhe aufweisen und diese Eigenschaft auch für die Unterbäume der Unterbäume gilt.

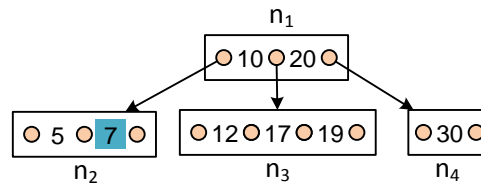
Mit so genannten 2-3-4-Bäumen lassen sich alle Baumoperationen so realisieren, dass der Baum immer ausbalanciert bleibt. Im Gegensatz zu Binärbäumen, bei denen jeder Knoten zwei Zeiger auf die Nachfolgerknoten aufweisen kann, können 2-3-4-Bäume Knoten mit zwei, drei oder vier Zeigern auf Nachfolgerknoten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



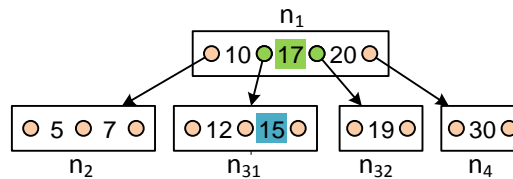
In jedem Knoten des Baums werden bis zu drei aufsteigend sortierte Schlüssel gespeichert. Damit in derartigen Bäumen effizient gesucht werden kann, sind die Elemente in Unterbäumen eines Knotens folgendermaßen angeordnet: Alle Schlüssel im ersten Unterbaum (jener, welcher am weitesten links liegt) sind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

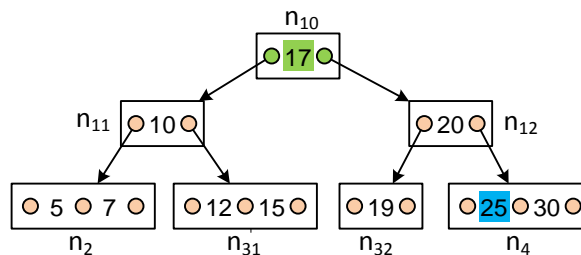
Das Einfügen eines neuen Elements gestaltet sich hingegen etwas komplizierter. Da neue Elemente nur in Blättern eingefügt werden, muss zunächst mit der oben beschriebenen Suchstrategie jenes Blatt bestimmt werden, in welches das Element gehört. In dieses Blatt wird das Element sortiert eingefügt, sofern 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 kleinste 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`).

- 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.
- 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

Folgend ist die Dokumentation für die Aufgabenstellung binäre Suchbäume angeführt.

Wie verlangt soll der binäre Suchbaum aus der Übung erweitert werden, wobei jedoch die bestehende Implementierung verbessert werden soll, z.B.: über eine abstrakte Superklasse, die Methoden wie `boolean contains(Object); Comparator<T> comparator();` implementiert.

Des Weiteren soll eine Methode `compareElemets(T o1, T o2);` implementiert werden, die entweder, bei zu Verfügung stehender `Comparator<T>` Implementierung, an diesen delegiert oder die Elemente auf `Comparable<T>` casted und so den Vergleich durchführt. Ansonsten ist die bestehende Implementierung wie verlangt zu erweitern.

Bezüglich des 2-3-4 Baums sei angemerkt, dass sich hier das Iterieren über den Baum als Herausforderung herausstellen könnte, da der gewöhnliche Ansatz (left, right) nicht anwendbar ist.

Hier ist es erforderlich das man immer zum Parent zurückkehrt und seine nächste Child Node besucht und hier wieder solange nach links geht bis man zur letzten Node kommt.

Diese kann man wieder ausgeben und man muss denselben Ansatz bei dessen Parent anwenden, solange bis man alle Nodes des Baums besucht hat.

Daher scheint es mir sinnvoll hier über ein Model zu arbeiten, und eine doppelte Verkettung der Nodes (`parent < - > child`) anzuwenden. Die tatsächlichen Nodes dürfen nicht geändert werden, daher muss hier über die Iteratoren deren Children und Keys zu arbeiten, die vom Model verwaltet werden sollen.

Beim Durchwandern des Baums sollten folgende Zustände auftreten:

1. Node hat keine Kinder, daher können alle seine Keys ausgegeben werden
2. Node hat Children daher soll der nächste Key ausgegeben werden und zur nächsten Child Node gewechselt werden, sofern die Node (Parent) aus dem Stack kommt. (Achtung: Parent gehört wieder am Stack)
3. Hat diese Node wieder Kinder, so muss wieder zu seiner kleinsten Node gewandert werden.
4. Die Parents müssen beim Besuch des Child darüber Informiert werden, das dieser vollständig besucht wurde. (Über Model und doppelte Verkettung)

Für Implementierungsdetails sei auf den Source verwiesen.

Für die Tests soll wieder die Testsuite aus der letzten Übung verwendet werden, wobei in dieser Dokumentation auf dessen Source und Implementierung nicht mehr genauer eingegangen werden soll. Sie soll als Third-Party-Library angesehen werden.

Da sehr viel Dokumentation in Form von Java-Doc vorhanden ist, sei generell auf diese Dokumentation verwiesen.

## Übung 3

### 1.2 Source-Code(Implementation)

Folgend ist der Source der Implementierungen der Übung angeführt.

#### 1.2.1 SortedSet.java

Folgend ist das Interfaces `SortedSet<T>` angeführt, welches aus der Aufgabenstellung hervorging.

Listing 1: SortedSet.java

```

1  package at.fh.ooe.swe4.collections.api;
2
3  import java.util.Comparator;
4  import java.util.NoSuchElementException;
5
6  /**
7   * This interface is the specification for a sorted set.
8   *
9   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
10  * @date May 17, 2015
11  * @param <T>
12  *         the generic type of the managed elements in this sorted set
13  */
14  public interface SortedSet<T> extends Iterable<T> {
15
16      /**
17       * Adds the element to the tree.
18       *
19       * @param el
20       *         the element to be added.
21       * @return true if the elements was successfully added to the tree, false
22       *         otherwise
23       */
24      public boolean add(T el);
25
26      /**
27       * Gets the element from the tree.
28       *
29       * @param el
30       *         the element to be get form the tree
31       * @return the found element null otherwise.
32       */
33      public T get(T el);
34
35      /**
36       * Answers the question of the given element is managed by this tree.
37       *
38       * @param el
39       *         the element to be searched in the tree
40       * @return true if the element is managed by this tree, false otherwise
41       */
42      public boolean contains(T el);
43
44      /**
45       * The current size of the tree. The size is equal to the managed elements
46       * in this tree
47       *
48       * @return the tree size
49       */
50      public int size();
51
52      /**

```

# Übung 3

```

53     * The first element in this tree which will be the element with the lowest
54     * value.
55     *
56     * @return the first element in this tree
57     * @throws NoSuchElementException
58     *         if the tree is empty.
59     */
60     public T first();
61
62     /**
63     * The last element in this tree which will be the element with the highest
64     * value.
65     *
66     * @return the last element in this tree
67     * @throws NoSuchElementException
68     *         if the tree is empty.
69     */
70     public T last();
71
72     /**
73     * Gets the backed {@link Comparator} instance of null if natural order is
74     * used for the tree
75     *
76     * @return the backed {@link Comparator} instance
77     */
78     public Comparator<T> comparator();
79
80     /**
81     * Returns the elements managed by this tree represented by an array.
82     *
83     * @param array
84     *         the array to be filled
85     * @return the filled array
86     */
87     public T[] toArray(T[] array);
88 }

```

## Übung 3

### 1.2.2 SortedTreeSet.java

Folgend ist das Interfaces `SortedTreeSet<T>` angeführt, welches aus der Aufgabenstellung hervorging und eine Erweiterung des Interfaces `SortedSet<T>` darstellt.

Listing 2: SortedTreeSet.java

```

1 package at.fh.ooe.swe4.collections.api;
2
3 /**
4  * This interface specifies a sorted tree set.
5  *
6  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
7  * @date May 17, 2015
8  * @param <T>
9  *         the generic type of the managed elements
10  */
11 public interface SortedTreeSet<T> extends SortedSet<T> {
12
13     /**
14      * Returns the maximum height the backed sorted tree has. The height is
15      * equal to the number of levels
16      *
17      * @return the height of the {@link SortedTreeSet} implementation backed
18      *         tree
19      */
20     public int height();
21 }

```

### 1.2.3 Node.java

Folgend ist das Interfaces `Node<T>` angeführt, welches als Marker Interface für die Tree Nodes fungiert, damit die abstrakte Klasse `AbstractSortedSet<T, M extends Model<T>` alle Node Typen aufnehmen kann. Dies ist möglich, da e

Listing 3: Node.java

```

1 package at.fh.ooe.swe4.collections.api;
2
3 /**
4  * Marker interface for an TreeNode
5  *
6  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
7  * @date May 17, 2015
8  */
9 public interface Node<T> {
10 }

```

## Übung 3

### 1.2.4 AbstractSortedSet.java

Folgend ist das Interfaces `AbstractSortedSet<T>` angeführt, welches als abstrakte Klasse implementiert wurde. Sie implementiert ebenfalls eine `compareElements(T o1, T o2)` Methode, die Elemente entweder mit einem zur Verfügung gestellten `Comparator<T>` oder mittels natürlicher Ordnung ordnet.

Listing 4: AbstractSortedSet.java

```

1 package at.fh.ooe.swe4.collections.api;
2
3 import java.util.Comparator;
4 import java.util.Iterator;
5
6 import at.fh.ooe.swe4.collections.comparator.NullSafeComparableComparator;
7
8 /**
9  * The base class for the {@link SortedSet} implementing classes.
10  *
11  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
12  * @date May 17, 2015
13  * @param <T>
14  *         the type of the managed node values
15  */
16 public abstract class AbstractSortedSet<T, M extends Node<T>> implements
17     SortedSet<T> {
18
19     protected final Comparator<T> comparator;
20     protected M root = null;
21     protected int size = 0;
22
23     /**
24      * Default constructor where the tree uses natural ordering and assumes that
25      * Tree nodes are Comparable instances.
26      */
27     public AbstractSortedSet() {
28         this(null);
29     }
30
31     /**
32      * Sets an comparator which is used for sorting the tree managed elements.
33      * The managed instance does not need to be comparable instances.
34      *
35      * @param comparator
36      *         the comparator to use for the sorting of this tree
37      */
38     public AbstractSortedSet(Comparator<T> comparator) {
39         super();
40
41         this.comparator = comparator;
42     }
43
44     @Override
45     public int size() {
46         return size;
47     }
48
49     @Override
50     public boolean contains(T el) {
51         return (get(el) != null) ? Boolean.TRUE : Boolean.FALSE;
52     }
53
54     /**

```



# Übung 3

```

55  * ATTENTION:<br>
56  * Uses {@link Iterator} for searching for the node. This can heavily
57  * decrease performance.
58  */
59  @Override
60  public T get(T el) {
61      // null or empty tree
62      if ((el == null) || (root == null)) {
63          return null;
64      }
65      // Iterate over tree
66      final Iterator<T> it = iterator();
67      T value = null;
68      while ((it.hasNext()) && (value == null)) {
69          final T itValue = it.next();
70          if (el.equals(itValue)) {
71              value = itValue;
72          }
73      }
74      return value;
75  }
76
77  @Override
78  public Comparator<T> comparator() {
79      return comparator;
80  }
81
82  @Override
83  public T[] toArray(T[] array) {
84      if ((array == null) || (array.length == 0)) {
85          return array;
86      }
87      final Iterator<T> it = iterator();
88      int idx = 0;
89      while ((it.hasNext()) && (idx < array.length)) {
90          array[idx] = it.next();
91          idx++;
92      }
93      return array;
94  }
95
96  /**
97   * Compares the two given elements either with the set comparator or with
98   * natural ordering.<br>
99   * Assumes that the managed keys are of type {@link Comparable}
100   *
101   * @param o1
102   *         the first instance
103   * @param o2
104   *         the second instance
105   * @return -1 if o1 is lower than o2<br>
106   *         0 if o1 and o2 are equal <br>
107   *         1 if o1 is greater than o2
108   * @throws ClassCastException
109   *         if the comparator is null and the managed keys are not of
110   *         type {@link Comparable}
111   */
112  protected int compareElements(final T o1, final T o2) {
113      if (comparator != null) {
114          return comparator.compare(o1, o2);
115      } else {
116          return new NullSafeComparableComparator<Comparable<T>>().compare(
117              ((Comparable<T>) o1), ((Comparable<T>) o2));

```

## Übung 3

---

118 }  
119 }  
120 }

## Übung 3

### 1.2.5 NullSafeComparableComparator.java

Folgend ist Implementierung eines Null-Safe Comparator<T> angeführt, welcher als Default für die compareElements{T o1, T o2} verwendet wird.

Listing 5: NullSafeComparableComparator.java

```

1 package at.fh.ooe.swe4.collections.comparator;
2
3 import java.util.Comparator;
4
5 /**
6  * This is a null safe ascending {@link Comparator} implementation for
7  * {@link Comparable} types
8  *
9  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
10 * @date May 27, 2015
11 * @param <T>
12 *         the {@link Comparable} type of the to compare objects
13 */
14 @SuppressWarnings({ "unchecked", "rawtypes" })
15 public class NullSafeComparableComparator<T extends Comparable> implements
16     Comparator<T> {
17
18     @Override
19     public int compare(T o1, T o2) {
20         // both are null
21         if ((o1 == null) && (o2 == null)) {
22             return 0;
23         } else if (o1 == null) {
24             return -1;
25         } else if (o2 == null) {
26             return 1;
27         }
28         return o1.compareTo(o2);
29     }
30 }

```

## Übung 3

### 1.2.6 EnumerationConstants.java

Folgend ist Implementierung einer Konstanten Klasse angeführt, welche die Enumeration Klassen Spezifikationen enthält. Es sollten für diese einfachen Enumerations keine eigenen Klassendateien erstellt werden.

Listing 6: EnumerationConstants.java

```

1 package at.fh.ooe.swe4.collections.constants;
2
3 /**
4  * This class holds the enumerations needed by the implementations.
5  *
6  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
7  * @date May 27, 2015
8  */
9 public final class EnumerationConstants {
10
11     /**
12      * Not meant to be instantiated
13      */
14     private EnumerationConstants() {
15         super();
16     }
17
18     /**
19      * Defines the boundary for a split set
20      *
21      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
22      * @date May 27, 2015
23      */
24     public static enum Boundary {
25         LOWER, UPPER;
26     }
27
28     /**
29      * Defines the direction to go.
30      *
31      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
32      * @date May 27, 2015
33      */
34     public static enum Direction {
35         LEFT, RIGHT;
36     }
37 }

```

## Übung 3

### 1.2.7 BinaryTreeNode.java

Folgend ist Implementierung des Tree Node Models angeführt, welches als Wrapper für die verwalteten Keys fungiert. Dieses Model ist für einen binären Suchbaum (2 Nachfolgeknoten) implementiert worden.

Listing 7: BinaryTreeNode.java

```

1 package at.fh.ooe.swe4.collections.model;
2
3 import at.fh.ooe.swe4.collections.api.Node;
4
5 /**
6  * The tree node for a binary tree implementation with two descendants.
7  *
8  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
9  * @date May 17, 2015
10 * @param <T>
11 *         the managed key tpye
12 */
13 public class BinaryTreeNode<T> implements Node<T> {
14
15     public T element;
16     public BinaryTreeNode<T> left;
17     public BinaryTreeNode<T> right;
18
19     /**
20      * @param element
21      */
22     public BinaryTreeNode(T element) {
23         this(element, null, null);
24     }
25
26     /**
27      * @param element
28      * @param left
29      * @param right
30      */
31     public BinaryTreeNode(T element, BinaryTreeNode<T> left,
32         BinaryTreeNode<T> right) {
33         super();
34         this.element = element;
35         this.left = left;
36         this.right = right;
37     }
38
39     @Override
40     public int hashCode() {
41         final int prime = 31;
42         int result = 1;
43         result = prime * result + ((element == null) ? 0 : element.hashCode());
44         return result;
45     }
46
47     @Override
48     public boolean equals(Object obj) {
49         if (this == obj) {
50             return true;
51         }
52         if (obj == null) {
53             return false;
54         }
55         if (getClass() != obj.getClass()) {

```

## Übung 3

```
56     return false;
57 }
58 BinaryTreeNode<T> other = (BinaryTreeNode<T>) obj;
59 if (element == null) {
60     if (other.element != null) {
61         return false;
62     }
63 } else if (!element.equals(other.element))
64     return false;
65 return true;
66 }
67
68 }
```

## Übung 3

### 1.2.8 NMKTreeNode.java

Folgend ist Implementierung des Tree Node Models angeführt, welches als Wrapper für die verwalteten Keys fungiert. Dieses Model ist für einen n-m-k Suchbaum, in unserem Fall einen 2-3-4 Baum, implementiert worden.

Listing 8: NMKTreeNode.java

```

1 package at.fh.ooe.swe4.collections.model;
2
3 import java.util.Comparator;
4 import java.util.HashMap;
5 import java.util.Iterator;
6 import java.util.Map;
7 import java.util.Objects;
8 import java.util.SortedSet;
9 import java.util.TreeSet;
10
11 import at.fh.ooe.swe4.collections.api.Node;
12 import at.fh.ooe.swe4.collections.comparator.NullSafeComparableComparator;
13
14 /**
15  * This is the node used for a tree which is allowed to hold multiple keys and
16  * children references.
17  *
18  * @author Thomas Herzog <thomas.hertzog@students.fh-hagenberg.at>
19  * @date May 22, 2015
20  * @param <T>
21  *         the comparable type managed by this node
22  */
23 public class NMKTreeNode<T> implements Node<T>,
24     Comparable<NMKTreeNode<T>>, Iterable<NMKTreeNode<T>> {
25
26     /**
27      * Enumeration for representing the split type.
28      *
29      * @author Thomas Herzog <thomas.hertzog@students.fh-hagenberg.at>
30      * @date May 23, 2015
31      */
32     public static enum Split {
33         HEAD, TAIL;
34     }
35
36     private NMKTreeNode<T> parent;
37     private final SortedSet<NMKTreeNode<T>> children;
38     private final SortedSet<T> keys;
39
40     private final Comparator<T> comparator;
41
42     /**
43      * Creates an empty node
44      */
45     public NMKTreeNode() {
46         this(null, new TreeSet<NMKTreeNode<T>>(), null);
47     }
48
49     /**
50      * Creates an node with one element set
51      *
52      * @param element
53      *         the element to be set
54      */
55     public NMKTreeNode(final T element) {

```

## Übung 3

```

56     this(element, new TreeSet<NMKTreeNode<T>>(
57         new NullSafeComparableComparator<NMKTreeNode<T>>()), null);
58 }
59
60 /**
61  * Creates an node with an custom {@link Comparator} instance
62  *
63  * @param element
64  *         the element managed by this node
65  * @param comparator
66  *         the comparator used for sorting the managed keys and
67  *         referenced children
68  */
69 public NMKTreeNode(final T element, final Comparator<T> comparator) {
70     this(element, new TreeSet<NMKTreeNode<T>>(
71         new NullSafeComparableComparator<NMKTreeNode<T>>()),
72         comparator);
73 }
74
75 /**
76  * Creates an fully customized node
77  *
78  * @param key
79  *         the key for this node
80  * @param children
81  *         the children to be set on this node
82  * @param comparator
83  *         the comparator used for key sorting
84  * @throws NullPointerException
85  *         if the children set is null
86  */
87 public NMKTreeNode(final T element,
88     final SortedSet<NMKTreeNode<T>> children,
89     final Comparator<T> comparator) {
90     super();
91     Objects.requireNonNull(children, "Children must be given");
92
93     this.comparator = comparator;
94     this.children = new TreeSet<>(
95         new NullSafeComparableComparator<NMKTreeNode<T>>());
96     this.children.addAll(children);
97     this.keys = new TreeSet<>(comparator);
98
99     if (element != null) {
100         addKey(element);
101     }
102 }
103
104 /**
105  * Adds a child to this node
106  *
107  * @param child
108  *         the child to be added
109  */
110 public void addChild(final NMKTreeNode<T> child) {
111     child.setParent(this);
112     children.add(child);
113 }
114
115 /**
116  * Removes the given child from the referenced children
117  *
118  * @param node

```



## Übung 3

```

119     *           the node to be removed
120     * @return true if the node could be removed, false otherwise
121     */
122     public boolean removeChild(final NMKTTreeNode<T> node) {
123         return children.remove(node);
124     }
125
126     /**
127      * Removes all references to all children.
128      */
129     public void clearChildren() {
130         children.clear();
131     }
132
133     /**
134      * Adds a key to this node
135      *
136      * @param key
137      *           the key to be added
138      */
139     public void addKey(final T key) {
140         keys.add(key);
141     }
142
143     /**
144      * Removes an key from this node
145      *
146      * @param key
147      *           the key to removed from this node
148      * @return true if the node has been removed, false otherwise
149      */
150     public boolean removeKey(final T key) {
151         return keys.remove(key);
152     }
153
154     /**
155      * Gets the lowest key hold by this tree.
156      *
157      * @return the lowest key or null if no key is set.
158      */
159     public T lowestKey() {
160         return (getKeySize() == 0) ? null : keys.first();
161     }
162
163     /**
164      * Gets the highest key hold by this tree.
165      *
166      * @return the highest key or null if no key is set.
167      */
168     public T highestKey() {
169         return (getKeySize() == 0) ? null : keys.last();
170     }
171
172     /**
173      * Gets the middle key and only if there are all possible keys set.
174      *
175      * @return the middle key if all possible keys are set, null otherwise
176      */
177     public T middleKey() {
178         if (getKeySize() != 3) {
179             return null;
180         } else {
181             final Iterator<T> it = keyIterator();

```

## Übung 3

```

182     it.next();
183     return it.next();
184 }
185 }
186
187 /**
188  * Gets and key at the given index.
189  *
190  * @param _idx
191  *         the index where the value resides
192  * @return the found value
193  * @throws IndexOutOfBoundsException
194  *         if the index is invalid
195  */
196 public T getKeyByIdx(final int _idx) {
197     int idx = 0;
198     if ((_idx >= 0) && (_idx < getKeySize())) {
199         for (T value : keys) {
200             if (idx == _idx) {
201                 return value;
202             }
203             idx++;
204         }
205     }
206     throw new IndexOutOfBoundsException("Key with idx=" + idx
207         + " does not exist");
208 }
209
210 /**
211  * Gets the key hold by this node.
212  *
213  * @param _key
214  *         the key to search in the key set
215  * @return the found key from the collection, null if given key is null or
216  *         node does not hold the given key.
217  */
218 public T getKey(final T _key) {
219     if (_key == null) {
220         return _key;
221     }
222     final Iterator<T> it = keyIterator();
223     T key = null;
224     while ((it.hasNext()) && (!_key.equals(key))) {
225         key = it.next();
226     }
227     return (_key.equals(key)) ? key : null;
228 }
229
230 /**
231  * Gets the highest child referenced by this tree.
232  *
233  * @return the highest child or null if there is no next higher.
234  */
235 public NMKTreeNode<T> highestChild() {
236     return (getChildrenSize() == 0) ? null : children.last();
237 }
238
239 /**
240  * Gets the lowest child referenced by this tree.
241  *
242  * @return the lowest child or null if there is no next higher.
243  */
244 public NMKTreeNode<T> lowestChild() {

```

## Übung 3

```

245     return (getChildrenSize() == 0) ? null : children.first();
246 }
247
248 /**
249  * Splits the hold children set into two sets where the set gets split in
250  * half (integer division).
251  *
252  * @return the map containing the head and tail set of the hold children set
253  */
254 public Map<Split, SortedSet<NMKTreeNode<T>>> splitChildren() {
255     int counter = 1;
256     final Map<Split, SortedSet<NMKTreeNode<T>>> split = new HashMap<>();
257     split.put(Split.HEAD, new TreeSet<NMKTreeNode<T>>(
258         new NullSafeComparableComparator<NMKTreeNode<T>>());
259     split.put(Split.TAIL, new TreeSet<NMKTreeNode<T>>(
260         new NullSafeComparableComparator<NMKTreeNode<T>>());
261
262     for (NMKTreeNode<T> child : children) {
263         if (counter <= ((int) (getChildrenSize() / 2))) {
264             split.get(Split.HEAD).add(child);
265         } else {
266             split.get(Split.TAIL).add(child);
267         }
268         counter++;
269     }
270
271     return split;
272 }
273
274 /**
275  * The current size of the hold keys
276  *
277  * @return the hole key size
278  */
279 public int getKeySize() {
280     return keys.size();
281 }
282
283 /**
284  * Gets the current referenced children size
285  *
286  * @return the hold children size
287  */
288 public int getChildrenSize() {
289     return children.size();
290 }
291
292 /**
293  * Gets the referenced parent node
294  *
295  * @return the parent node
296  */
297 public NMKTreeNode<T> getParent() {
298     return parent;
299 }
300
301 /**
302  * Sets the parent which references this node
303  *
304  * @param parent
305  *         the parent to be set on this node
306  */
307 public void setParent(NMKTreeNode<T> parent) {

```

## Übung 3

```

308     this.parent = parent;
309 }
310
311 /**
312  * Gets the iterator for the children referenced by this node.
313  *
314  * @return the iterator of the node's children
315  */
316 @Override
317 public Iterator<NMKTreeNode<T>> iterator() {
318     return children.iterator();
319 }
320
321 /**
322  * Gets the iterator for the hold keys.
323  *
324  * @return the iterator for the old keys.
325  */
326 public Iterator<T> keyIterator() {
327     return keys.iterator();
328 }
329
330 /**
331  * Gets the iterator for the hold children
332  *
333  * @return the children iterator
334  */
335 public Iterator<NMKTreeNode<T>> childrenIterator() {
336     return children.iterator();
337 }
338
339 /**
340  * TreeNodes are sorted by their first hold element.
341  */
342 @Override
343 public int compareTo(NMKTreeNode<T> o) {
344     // in case of provided comparator
345     if (comparator != null) {
346         return comparator.compare(this.lowestKey(), o.lowestKey());
347     }
348     // Otherwise null safe comparator implementation
349     else {
350         if ((lowestKey() == null) && (o.lowestKey() == null)) {
351             return 0;
352         }
353         if (lowestKey() == null) {
354             return -1;
355         }
356         if (o.lowestKey() == null) {
357             return 1;
358         }
359         if (!(lowestKey() instanceof Comparable)) {
360             throw new IllegalStateException(
361                 "Managed Elements need to implement Comparable<T> interface if no Comparator<T>
                 ↳ instance is provided");
362         }
363
364         return ((Comparable<T>) lowestKey()).compareTo(o.lowestKey());
365     }
366 }
367
368 @Override
369 public int hashCode() {

```

## Übung 3

```

370     final int prime = 31;
371     int result = 1;
372     result = prime * result + ((keys == null) ? 0 : keys.hashCode());
373     return result;
374 }
375
376 @Override
377 public boolean equals(Object obj) {
378     if (this == obj) {
379         return true;
380     }
381     if (obj == null) {
382         return false;
383     }
384     if (getClass() != obj.getClass()) {
385         return false;
386     }
387     NMKTreeNode<T> other = (NMKTreeNode<T>) obj;
388     if (keys == null) {
389         if (other.keys != null) {
390             return false;
391         }
392     } else if (!keys.equals(other.keys)) {
393         return false;
394     }
395     return true;
396 }
397 }

```

## Übung 3

### 1.2.9 BinarySearchTreeSet.java

Folgend ist Implementierung des binären Suchbaums angeführt. Der `Iterator<T>` wurde als innere Klasse implementiert.

Listing 9: BinarySearchTreeSet.java

```

1 package at.fh.ooe.swe4.collections.impl;
2
3 import java.util.Comparator;
4 import java.util.Iterator;
5 import java.util.NoSuchElementException;
6 import java.util.Stack;
7
8 import at.fh.ooe.swe4.collections.api.AbstractSortedSet;
9 import at.fh.ooe.swe4.collections.api.SortedTreeSet;
10 import at.fh.ooe.swe4.collections.model.BinaryTreeNode;
11
12 /**
13  *
14  * @author Thomas Herzog <thomas.hertzog@students.fh-hagenberg.at>
15  * @date May 17, 2015
16  * @param <T>
17  */
18 public class BinarySearchTreeSet<T> extends
19     AbstractSortedSet<T, BinaryTreeNode<T>> implements SortedTreeSet<T> {
20
21     /**
22      * This is the iterator for the {@link BinarySearchTreeSet} class.
23      *
24      * @author Thomas Herzog <thomas.hertzog@students.fh-hagenberg.at>
25      * @date May 17, 2015
26      * @param <T>
27      */
28     public static class BinarySearchTreeIterator<T> implements Iterator<T> {
29
30         private final Stack<BinaryTreeNode<T>> unvisitedNodes = new Stack<>();
31
32         /**
33          * Initializes this iterator with the root node.
34          *
35          * @param node
36          *         the root node of the backed tree
37          */
38         public BinarySearchTreeIterator(final BinaryTreeNode<T> root) {
39             super();
40             BinaryTreeNode<T> node = root;
41             while (node != null) {
42                 unvisitedNodes.push(node);
43                 node = node.left;
44             }
45         }
46
47         @Override
48         public boolean hasNext() {
49             return !unvisitedNodes.isEmpty();
50         }
51
52         @Override
53         public T next() {
54             if (!hasNext()) {
55                 throw new NoSuchElementException(
56                     "No further elements are available");

```

## Übung 3

```

57     }
58     final BinaryTreeNode<T> current = unvisitedNodes.pop();
59     BinaryTreeNode<T> node = current.right;
60     while (node != null) {
61         unvisitedNodes.push(node);
62         node = node.left;
63     }
64     return current.element;
65 }
66
67 }
68
69 /**
70  * The level count of the backed tree
71  */
72 private int levels = 0;
73
74 /**
75  * Initialized with natural ordering which requires Comparable<T> interface
76  * implemented by the managed elements
77  */
78 public BinarySearchTreeSet() {
79     super();
80 }
81
82 /**
83  * Initializes this Instance with a custom Comparator<T> instance
84  *
85  * @param comparator
86  *         the Comparator<T> instance to be used
87  */
88 public BinarySearchTreeSet(Comparator<T> comparator) {
89     super(comparator);
90 }
91
92 @Override
93 public T get(T el) {
94     BinaryTreeNode<T> node = root;
95     while (node != null) {
96         int res = compareElements(node.element, el);
97         if (res == 0) {
98             return node.element;
99         }
100         if (res > 0) {
101             node = node.left;
102         } else {
103             node = node.right;
104         }
105     }
106     return null;
107 }
108
109 @Override
110 public boolean add(T el) {
111     boolean modified = Boolean.FALSE;
112     int currentLevels = 0;
113     /*-- Null not allowed --*/
114     if (el != null) {
115         // root already present
116         if (root != null) {
117             BinaryTreeNode<T> previous = null;
118             BinaryTreeNode<T> node = root;
119             while (node != null) {

```

## Übung 3

```

120         int compareResult = compareElements(el, node.element);
121         // No duplicates allowed
122         if (compareResult == 0) {
123             previous = null;
124             break;
125         }
126         previous = node;
127         currentLevels++;
128         // go to left
129         if (compareResult < 0) {
130             node = node.left;
131         }
132         // go to right
133         else {
134             node = node.right;
135         }
136     }
137     // add new child if previous found
138     if (previous != null) {
139         currentLevels++;
140         modified = Boolean.TRUE;
141         size++;
142         final BinaryTreeNode<T> newNode = new BinaryTreeNode<T>(el,
143             null, null);
144         if (compareElements(el, previous.element) < 0) {
145             previous.left = newNode;
146         } else {
147             previous.right = newNode;
148         }
149     }
150 }
151 // add root element
152 else {
153     root = new BinaryTreeNode<T>(el, null, null);
154     modified = Boolean.TRUE;
155     currentLevels = 1;
156     size++;
157 }
158 // set if new level reached
159 if (levels < currentLevels) {
160     levels = currentLevels;
161 }
162 }
163 return modified;
164 }
165
166 @Override
167 public T first() {
168     if (root == null) {
169         throw new NoSuchElementException(
170             "Tree is empty therefore no first element available");
171     }
172     BinaryTreeNode<T> node = root;
173     while (node.left != null) {
174         node = node.left;
175     }
176     return node.element;
177 }
178
179 @Override
180 public T last() {
181     if (root == null) {
182         throw new NoSuchElementException(

```



## Übung 3

```
183         "Tree is empty therefore no last element available");
184     }
185     BinaryTreeNode<T> node = root;
186     while (node.right != null) {
187         node = node.right;
188     }
189     return node.element;
190 }
191
192 @Override
193 public Iterator<T> iterator() {
194     return new BinarySearchTreeIterator<T>(root);
195 }
196
197 @Override
198 public int height() {
199     return levels;
200 }
201
202 }
```

## Übung 3

### 1.2.10 TwoThreeFourTreeSet.java

Folgend ist Implementierung des 2-3-4 Baums angeführt. Der `Iterator<T>` wurde als externe Klasse implementiert.

Listing 10: TwoThreeFourTreeSet.java

```

1 package at.fh.ooe.swe4.collections.impl;
2
3 import java.util.Comparator;
4 import java.util.Iterator;
5 import java.util.Map;
6 import java.util.Map.Entry;
7 import java.util.NoSuchElementException;
8 import java.util.Objects;
9 import java.util.SortedSet;
10
11 import at.fh.ooe.swe4.collections.api.AbstractSortedSet;
12 import at.fh.ooe.swe4.collections.api.SortedTreeSet;
13 import at.fh.ooe.swe4.collections.iterator.NMKTreeIterator;
14 import at.fh.ooe.swe4.collections.model.NMKTreeNode;
15 import at.fh.ooe.swe4.collections.model.NMKTreeNode.Split;
16
17 /**
18  * This is a tree implementation which uses a self balancing 2-3-4 tree for
19  * managing the hold elements.
20  *
21  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
22  * @date May 16, 2015
23  * @param <T>
24  *         the managed key type
25  */
26 public class TwoThreeFourTreeSet<T> extends
27     AbstractSortedSet<T, NMKTreeNode<T>> implements SortedTreeSet<T> {
28
29     /**
30      * Enumeration which specifies which process shall be applied on the current
31      * metatdata instance.
32      *
33      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
34      * @date May 25, 2015
35      */
36     private static enum ProcessState {
37         NEXT_STEP, ADD_TO_CURRENT, ADD_TO_SPLIT, IS_CURRENT;
38     }
39
40     /**
41      * This class is used for holding the balance result and allows to pass
42      * through the actual height.
43      *
44      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
45      * @date May 25, 2015
46      * @param <T>
47      *         the managed key type
48      */
49     private static class BalanceResult<T> {
50         private int height;
51         private NMKTreeNode<T> node;
52
53         /**
54          * Creates a result instance and sets current height
55          *
56          * @param height

```

## Übung 3

```

57      *           the height where the node resides
58      */
59      public BalanceResult(final int height) {
60          super();
61          this.height = height;
62      }
63
64      /**
65       * Decrease height by one.
66       */
67      public void dec() {
68          height--;
69      }
70
71      /**
72       * Gets the resulting node
73       *
74       * @return the resulting node
75       */
76      public NMKTreeNode<T> getNode() {
77          return node;
78      }
79
80      /**
81       * Sets the current node. Should correspond tot he set height.
82       *
83       * @param node
84       *           the node to be set
85       */
86      public void setNode(NMKTreeNode<T> node) {
87          this.node = node;
88      }
89
90      /**
91       * Gets the height of the current node
92       *
93       * @return the height of the hold node
94       */
95      public int getHeight() {
96          return height;
97      }
98  }
99
100  /**
101   * This class is used for determining if the visited node is processable or
102   * not.<br>
103   * If the current node is not visible then the toVisit node should be set.
104   * It should never be possible that current node is never processable.
105   *
106   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
107   * @date May 23, 2015
108   * @param <T>
109   *           the managed key tpye
110   */
111  public static class TreeNodeMetadata<T> {
112
113      public final ProcessState process;
114      public final NMKTreeNode<T> current;
115      public final NMKTreeNode<T> toVisit;
116
117      /**
118       * Creates a metadata instance with the current node set
119       *

```

## Übung 3

```

120     * @param current
121     *         the current visited node
122     * @throws NullPointerException
123     *         if the current node is null
124     */
125     public TreeNodeMetadata(final NMKTreeNode<T> current) {
126         this(current, null, null);
127     }
128
129     /**
130     * Creates an instance with the current node and toVisit node.<br>
131     * A null toVisit indicates that the current node is processable.
132     *
133     * @param current
134     *         the current visited node
135     * @param toVisit
136     *         the next to visit node
137     * @param process
138     *         TODO
139     * @throws NullPointerException
140     *         if the current node is null
141     */
142     public TreeNodeMetadata(final NMKTreeNode<T> current,
143         final NMKTreeNode<T> toVisit, ProcessState process) {
144         super();
145         Objects.requireNonNull(current,
146             "At least current node must be given");
147
148         this.current = current;
149         this.toVisit = toVisit;
150         this.process = process;
151     }
152 }
153
154 private int height = 0;
155
156 public TwoThreeFourTreeSet() {
157     super();
158 }
159
160 public TwoThreeFourTreeSet(Comparator<T> comparator) {
161     super(comparator);
162 }
163
164 @Override
165 public Iterator<T> iterator() {
166     return new NMKTreeIterator<T>(root);
167 }
168
169 @Override
170 public boolean add(T e) {
171     boolean modified = Boolean.FALSE;
172     Integer currentHeight = Integer.valueOf(1);
173     /*-- null not allowed --*/
174     if (e != null) {
175         /*-- no root present --*/
176         if (root == null) {
177             root = new NMKTreeNode<T>(e, comparator());
178             currentHeight = 1;
179             modified = Boolean.TRUE;
180         }
181         /*-- search node and add key --*/
182         else {

```

## Übung 3

```

183     NMKTreeNode<T> node = root;
184     while (node != null) {
185         TreeNodeMetadata<T> metadata = calculateTreeNodeMetadata(
186             node, e);
187         switch (metadata.process) {
188             // do nothing on duplicate
189             case IS_CURRENT:
190                 node = null;
191                 break;
192             // add to current node
193             case ADD_TO_CURRENT:
194                 node.addKey(e);
195                 modified = Boolean.TRUE;
196                 node = null;
197                 break;
198             // split node and walk to proper child
199             case ADD_TO_SPLIT:
200                 // balance tree and revisit balanced tree
201                 final BalanceResult<T> result = new BalanceResult<T>(
202                     currentHeight);
203                 balanceTree(node, result);
204                 // get new current node
205                 node = result.getNode();
206                 // set the height of the new current node
207                 currentHeight = result.getHeight();
208                 break;
209             // go to next node
210             case NEXT_STEP:
211                 node = metadata.toVisit;
212                 currentHeight++;
213                 break;
214             default:
215                 break;
216         }
217     }
218 }
219 }
220
221 // Set new height if higher than actual set one
222 height = (height < currentHeight) ? currentHeight : height;
223
224 if (modified) {
225     size++;
226 }
227
228 return modified;
229 }
230
231 @Override
232 public T get(T el) {
233     NMKTreeNode<T> node = root;
234     if (node == null) {
235         return null;
236     }
237     while (node != null) {
238         final TreeNodeMetadata<T> metadata = calculateTreeNodeMetadata(
239             node, el);
240         switch (metadata.process) {
241             // found on this node
242             case IS_CURRENT:
243                 return metadata.current.getKey(el);
244             // need to visit next node
245             case NEXT_STEP:

```

# Übung 3

```

246     node = metadata.toVisit;
247     break;
248     // could be placed here therefore no further node to visit
249     case ADD_TO_CURRENT:
250         return null;
251         // would cause new node therefore node further node to visit
252     case ADD_TO_SPLIT:
253         return null;
254     default:
255         break;
256     }
257 }
258
259 // Should never get here
260 throw new IllegalStateException(
261     "Should never get to end of get method ");
262 }
263
264 @Override
265 public T first() {
266     if (size() == 0) {
267         throw new NoSuchElementException("Tree is empty");
268     }
269     NMKTreeNode<T> node = root;
270     while (node.lowestChild() != null) {
271         node = node.lowestChild();
272     }
273     return node.lowestKey();
274 }
275
276 @Override
277 public T last() {
278     if (size() == 0) {
279         throw new NoSuchElementException("Tree is empty");
280     }
281     NMKTreeNode<T> node = root;
282     while (node.highestChild() != null) {
283         node = node.highestChild();
284     }
285     return node.highestKey();
286 }
287
288 @Override
289 public int height() {
290     return height;
291 }
292
293 // Private section
294 /**
295  * Calculates the metadata for the current node, where the metadata instance
296  * contains the current node and toVisit node, which indicates that there is
297  * another node needed to be visited.<br>
298  * The calculation is performed by searching where the given key needs to be
299  * placed.<br>
300  * Be aware that the overflows are not considered here.
301  *
302  * @param _node
303  *         the node to calculate metadata for
304  * @param _key
305  *         the key which gets included in the calculation
306  * @return
307  */
308 private TreeNodeMetadata<T> calculateTreeNodeMetadata(

```

# Übung 3

```

309     final NMKTreeNode<T> _node, final T _key) {
310     TreeNodeMetadata<T> model = null;
311
312     /*-- 1. one of last nodes --*/
313     if (_node.getChildrenSize() == 0) {
314         final ProcessState process;
315         // node holds key
316         if (_node.getKey(_key) != null) {
317             process = ProcessState.IS_CURRENT;
318         }
319         // node is full
320         else if (_node.getKeySize() == 3) {
321             process = ProcessState.ADD_TO_SPLIT;
322         }
323         // able to add to current node
324         else {
325             process = ProcessState.ADD_TO_CURRENT;
326         }
327         model = new TreeNodeMetadata<T>(_node, null, process);
328     }
329     /* 2. find node to visit */
330     else {
331         final Iterator<NMKTreeNode<T>> childrenIt = _node
332             .childrenIterator();
333         final Iterator<T> keyIt = _node.keyIterator();
334         // visit all children along with keys
335         while ((childrenIt.hasNext()) && (model == null)) {
336             final NMKTreeNode<T> node = childrenIt.next();
337             // as long as keys are left we can go left depending current key
338             if (keyIt.hasNext()) {
339                 final T key = keyIt.next();
340                 final int compResult = compareElements(_key, key);
341                 // we found a duplicate
342                 if (compResult == 0) {
343                     model = new TreeNodeMetadata<T>(_node, null,
344                         ProcessState.IS_CURRENT);
345                 }
346                 // is lower than current key, therefore visit child
347                 else if (compResult < 0) {
348                     model = new TreeNodeMetadata<T>(_node, node,
349                         ProcessState.NEXT_STEP);
350                 }
351             }
352             // here we have the highest node reached means go right
353             else {
354                 model = new TreeNodeMetadata<T>(node, node,
355                     ProcessState.NEXT_STEP);
356             }
357         }
358     }
359
360     Objects.requireNonNull(model, "Model should be set here");
361
362     return model;
363 }
364
365 /**
366  * Re-balances the tree if the current node has a keySize = 3.<br>
367  * It recalls itself recursively in case the parent has been modified and
368  * will perform a split as long as the current node overflows boundaries.
369  *
370  * @param node
371  *         the node to be split

```

## Übung 3

```

372  * @param currentLevels
373  *      TODO
374  * @throws NullPointerException
375  *      if the node is null
376  * @throws IllegalStateException
377  *      if the node has not exactly 0 or 4 children set
378  */
379  private void balanceTree(final NMKTreeNode<T> node,
380      final BalanceResult<T> result) {
381      Objects.requireNonNull(node, "The node to split must not be null");
382      Objects.requireNonNull(result, "result instance must be given");
383
384      // skip when less keys or null
385      if (node.getKeySize() != 3) {
386          result.setNode(node);
387          return;
388      }
389
390      // valid nodes have zero or four children
391      if ((node.getChildrenSize() != 0) && (node.getChildrenSize() != 4)) {
392          throw new IllegalStateException(
393              " with 3 keys 4 nodes must be present. size: "
394              + node.getChildrenSize());
395      }
396
397      NMKTreeNode<T> parent, leftNode, rightNode;
398
399      // 1. keys for split
400      final T leftKey = node.lowestKey();
401      final T middleKey = node.middleKey();
402      final T rightKey = node.highestKey();
403      // 2. involved nodes
404      parent = node.getParent();
405      leftNode = new NMKTreeNode<T>(leftKey, comparator());
406      rightNode = new NMKTreeNode<T>(rightKey, comparator());
407      // 3. split children if 4 children present
408      if (node.getChildrenSize() == 4) {
409          // split the children
410          Map<Split, SortedSet<NMKTreeNode<T>>> splitChildren = node
411              .splitChildren();
412          // iterate over head an tail subset
413          for (Entry<Split, SortedSet<NMKTreeNode<T>>> entry : splitChildren
414              .entrySet()) {
415              // Add each node to proper new node
416              for (NMKTreeNode<T> child : entry.getValue()) {
417                  switch (entry.getKey()) {
418                      case HEAD:
419                          leftNode.addChild(child);
420                          break;
421                      case TAIL:
422                          rightNode.addChild(child);
423                      default:
424                          break;
425                  }
426              }
427          }
428          node.clearChildren();
429      }
430      // 4.1 add to parent if present
431      if (parent != null) {
432          parent.addKey(middleKey);
433          parent.removeChild(node);
434          parent.addChild(leftNode);

```



## Übung 3

```
435     parent.addChild(rightNode);
436     // re-balance because overflow could occur
437     result.dec();
438     balanceTree(parent, result);
439     return;
440 }
441 // 3.2 else modify current node
442 else {
443     node.removeKey(leftKey);
444     node.removeKey(rightKey);
445     node.addChild(leftNode);
446     node.addChild(rightNode);
447     result.setNode(node);
448     return;
449 }
450 }
451 }
```

## Übung 3

### 1.2.11 NMKTreeIterator.java

Folgend ist Implementierung des n-m-k `Iterator<T>` angeführt. Dieser Iterator sollte mit 1-2-3 genauso wie mit 2-3-4 oder irgendeiner anderen Art funktionieren, solange sich die Baumvariationen wie ein 2-3-4 Baum verhalten nur mit andere Node/Key Anzahl.

Listing 11: NMKTreeIterator.java

```

1 package at.fh.ooe.swe4.collections.iterator;
2
3 import java.util.Collections;
4 import java.util.Iterator;
5 import java.util.NoSuchElementException;
6 import java.util.Objects;
7 import java.util.Stack;
8
9 import at.fh.ooe.swe4.collections.model.NMKTreeNode;
10
11 /**
12  * The iterator for the n-m-k Tree.
13  *
14  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
15  * @date May 22, 2015
16  * @param <T>
17  *         the {@link Comparable} type managed by the backed tree
18  */
19 public class NMKTreeIterator<T> implements Iterator<T> {
20
21     private IterateModel<T> currentModel = null;
22     private final Stack<IterateModel<T>> unvisitedNodes = new Stack<>();
23
24     /**
25      * The helper model for iterating over an n-m-k tree.
26      *
27      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
28      * @date May 24, 2015
29      * @param <T>
30      *         the type of the managed nodes
31      */
32     private static class IterateModel<T> {
33
34         public final NMKTreeNode<T> current;
35         public final IterateModel<T> parent;
36         public final Stack<NMKTreeNode<T>> children = new Stack<>();
37         public final Iterator<T> keyIt;
38
39         /**
40          * Creates a iterate model instance
41          *
42          * @param parent
43          *         the parent of this model
44          * @param current
45          *         the current node which is backed by this model
46          */
47         public IterateModel(final IterateModel<T> parent,
48                             final NMKTreeNode<T> current) {
49             super();
50             Objects.requireNonNull(current, "Current node must be given");
51
52             this.parent = parent;
53             this.current = current;
54             this.keyIt = current.keyIterator();
55             // set all children on stack in proper order

```

## Übung 3

```

56     final Iterator<NMKTreeNode<T>> it = current.childrenIterator();
57     while (it.hasNext()) {
58         children.add(it.next());
59     }
60     Collections.reverse(children);
61 }
62 }
63
64 /**
65  * Creates an iterator instance and moves to first to visit node.
66  *
67  * @param root
68  *      the root of the tree
69  */
70 public NMKTreeIterator(final NMKTreeNode<T> root) {
71     super();
72     pushUntillLowest(root, null);
73     if (hasNext()) {
74         currentModel = unvisitedNodes.pop();
75     }
76 }
77
78 @Override
79 public boolean hasNext() {
80     return (!unvisitedNodes.isEmpty())
81         || (currentModel != null && (currentModel.keyIt.hasNext() || (!currentModel.children
82             .isEmpty())));
83 }
84
85 @Override
86 public T next() {
87     T value = null;
88     /*-- no more nodes available --*/
89     if (!hasNext()) {
90         throw new NoSuchElementException(
91             "No further elements are available");
92     }
93     /*-- current model has no more children --*/
94     if (currentModel.children.isEmpty()) {
95         // keys still left
96         if (currentModel.keyIt.hasNext()) {
97             value = currentModel.keyIt.next();
98         }
99         // no children and no keys means next node ins tack
100        else {
101            // Remove this node from parent, because was completely visited
102            // before
103            currentModel.parent.children.remove(currentModel.current);
104
105            // get next node
106            currentModel = unvisitedNodes.pop();
107
108            // get current value from new node
109            value = currentModel.keyIt.next();
110
111            // if there are children
112            if (!currentModel.children.isEmpty()) {
113                // need to revisit if keys still left
114                if (currentModel.keyIt.hasNext()) {
115                    unvisitedNodes.push(currentModel);
116                }
117                // inform parent that this node does not need to be visited
118                // again if no further keys are available

```

## Übung 3

```

119         else if (currentModel.parent != null) {
120             currentModel.parent.children
121                 .remove(currentModel.current);
122         }
123
124         // go to lowest node
125         pushUntillLowest(currentModel.children.pop(), currentModel);
126
127         // get lowest node from stack
128         currentModel = unvisitedNodes.pop();
129     }
130 }
131 }
132 /*-- get on further children --*/
133 else {
134     // we need to revisit this node if it has children set
135     if (!currentModel.children.isEmpty()) {
136         unvisitedNodes.push(currentModel);
137     } else if (currentModel.parent != null) {
138         currentModel.parent.children.remove(currentModel.current);
139     }
140     // get lowest node of next child to visit
141     pushUntillLowest(currentModel.children.pop(), currentModel);
142     // get lowest node
143     currentModel = unvisitedNodes.pop();
144 }
145 // next key to provide
146 return value;
147 }
148
149 // #####
150 // Private section
151 // #####
152 /**
153  * Pushes the nodes to the unvisited {@link Stack} including the next
154  * neighbor of the lowest element if present.
155  *
156  * @param root
157  *         the node to walk all way left to the lowest node
158  */
159 private void pushUntillLowest(final NMKTreeNode<T> root,
160     final IterateModel<T> parent) {
161     NMKTreeNode<T> node = root;
162     IterateModel<T> prev, model;
163     prev = model = null;
164     while (node != null) {
165         // if there is a parent get its children
166         if (node.getParent() != null) {
167             // set provided parent. For first visited node
168             prev = (prev == null) ? parent : prev;
169         }
170         // create current model
171         model = new IterateModel<T>(prev, node);
172         // remember parent model of next visited node
173         prev = model;
174         unvisitedNodes.push(model);
175         // walk left
176         node = node.lowestChild();
177     }
178 }
179 }

```

## Übung 3

### 1.3 Source-Code(Tests)

Folgend ist der Source der Tests der Übung angeführt.

#### 1.3.1 BinarySearchTreeTestDataProducer.java

Folgend ist die Implementierung eines Tests Data Producers angeführt, welche die Testdaten für binäre Test Klassen zur Verfügung stellt, wobei es sich hierbei um 2 selbst aufgebaute Bäume handelt.

Listing 12: BinarySearchTreeTestDataProducer.java

```

1 package at.fh.ooe.swe4.test.collections.binarySearchTreeSet.api;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.function.IntConsumer;
6 import java.util.stream.IntStream;
7
8 import at.fh.ooe.swe4.collections.impl.BinarySearchTreeSet;
9 import at.fh.ooe.swe4.collections.model.BinaryTreeNode;
10
11 /**
12  * This class provides the test data for the {@link BinarySearchTreeSet} tests.
13  *
14  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
15  * @date May 27, 2015
16  */
17 public class BinarySearchTreeTestDataProducer {
18
19     /**
20      * The provided test cases.
21      *
22      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
23      * @date May 27, 2015
24      */
25     public static enum TestCase {
26         TREE_1, TREE_2;
27     }
28
29     /**
30      * The model for holding the test data for a specific test case.
31      *
32      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
33      * @date May 27, 2015
34      */
35     public static class TestData {
36
37         public final List<Integer> orderedValues;
38         public final List<Integer> insertOrderedValues;
39         public final int levels;
40         public final BinaryTreeNode<Integer> root;
41
42         /**
43          * @param orderedValues
44          * @param insertOrderedValues
45          * @param root
46          * @param levels
47          */
48         public TestData(final List<Integer> orderedValues,
49             final List<Integer> insertOrderedValues,
50             final BinaryTreeNode<Integer> root, final int levels) {

```

# Übung 3

```

51     super();
52     this.orderedValues = orderedValues;
53     this.insertOrderedValues = insertOrderedValues;
54     this.root = root;
55     this.levels = levels;
56 }
57 }
58
59 /**
60  * Creates the test model for the given test case.
61  *
62  * @param testCase
63  *         the test case to create test data for
64  * @return the model holding the test data for this test case
65  */
66 public static TestData createTestData(final TestCase testCase) {
67     final int size = 7;
68     final List<Integer> orderedValues = new ArrayList<Integer>(size);
69     final List<Integer> insertOrderedValues = new ArrayList<Integer>(size);
70     IntStream.range(1, 8).forEachOrdered(new IntConsumer() {
71
72         @Override
73         public void accept(int value) {
74             orderedValues.add(value);
75
76         }
77     });
78     switch (testCase) {
79     case TREE_1:
80         // -- Test should be able to walk this tree
81         // ----- 5 -----
82         // ----- 4 ----- 6 -----
83         // ----- 2 ----- 7 -----
84         // -- 1 --- 3 -----
85         // -----
86         final BinaryTreeNode<Integer> l1 = new BinaryTreeNode<Integer>(1);
87         final BinaryTreeNode<Integer> l2 = new BinaryTreeNode<Integer>(3);
88         final BinaryTreeNode<Integer> l1l2 = new BinaryTreeNode<Integer>(2,
89             l1, l2);
90         final BinaryTreeNode<Integer> l0 = new BinaryTreeNode<Integer>(4,
91             l1l2, null);
92         final BinaryTreeNode<Integer> r1 = new BinaryTreeNode<Integer>(7);
93         final BinaryTreeNode<Integer> r0 = new BinaryTreeNode<Integer>(6,
94             null, r1);
95         final BinaryTreeNode<Integer> root = new BinaryTreeNode<Integer>(5,
96             l0, r0);
97
98         insertOrderedValues.add(5);
99         insertOrderedValues.add(4);
100        insertOrderedValues.add(6);
101        insertOrderedValues.add(7);
102        insertOrderedValues.add(2);
103        insertOrderedValues.add(1);
104        insertOrderedValues.add(3);
105        return new TestData(orderedValues, insertOrderedValues, root, 4);
106     case TREE_2:
107         // -- Example in order
108         // -- Test should be able to walk this tree
109         // ----- 4 -----
110         // ----- 2 ----- 6 -----
111         // ----- 1 -----3-----5----- 7 -----
112         // -----
113         final BinaryTreeNode<Integer> _l1 = new BinaryTreeNode<Integer>(1);

```

## Übung 3

```

114     final BinaryTreeNode<Integer> _l2 = new BinaryTreeNode<Integer>(3);
115     final BinaryTreeNode<Integer> _l1l2 = new BinaryTreeNode<Integer>(
116         2, _l1, _l2);
117     final BinaryTreeNode<Integer> _r1 = new BinaryTreeNode<Integer>(5);
118     final BinaryTreeNode<Integer> _r2 = new BinaryTreeNode<Integer>(7);
119     final BinaryTreeNode<Integer> _r1r2 = new BinaryTreeNode<Integer>(
120         6, _r1, _r2);
121     final BinaryTreeNode<Integer> _root = new BinaryTreeNode<Integer>(
122         4, _l1l2, _r1r2);
123
124     insertOrderedValues.add(4);
125     insertOrderedValues.add(6);
126     insertOrderedValues.add(5);
127     insertOrderedValues.add(7);
128     insertOrderedValues.add(2);
129     insertOrderedValues.add(3);
130     insertOrderedValues.add(1);
131
132     return new TestData(orderedValues, insertOrderedValues, _root, 3);
133     default:
134         return null;
135     }
136 }
137 }

```

## Übung 3

### 1.3.2 IteratorValidTest.java

Folgende Testklasse testet den Iterator des binären Suchbaums und wendet dazu parametrisierte Tests an, die mit den produzierten Testdaten das iterieren testen.

Listing 13: IteratorValidTest.java

```

1 package at.fh.ooe.swe4.test.collections.binarySearchTreeSet.impl.parameterized;
2
3 import static org.junit.Assert.assertEquals;
4
5 import java.util.Arrays;
6 import java.util.Iterator;
7 import java.util.List;
8
9 import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.junit.runners.Parameterized;
12 import org.junit.runners.Parameterized.Parameter;
13 import org.junit.runners.Parameterized.Parameters;
14
15 import at.fh.ooe.swe4.collections.impl.BinarySearchTreeSet.BinarySearchTreeIterator;
16 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
17 import at.fh.ooe.swe4.test.collections.binarySearchTreeSet.api.BinarySearchTreeTestDataProducer;
18 import
19     ↳ at.fh.ooe.swe4.test.collections.binarySearchTreeSet.api.BinarySearchTreeTestDataProducer.TestCase;
20 import
21     ↳ at.fh.ooe.swe4.test.collections.binarySearchTreeSet.api.BinarySearchTreeTestDataProducer.TestData;
22
23 /**
24  * This test class is a parameterized test for the
25  * {@link BinarySearchTreeIterator} implementation.
26  *
27  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
28  * @date May 27, 2015
29  */
30 @RunWith(Parameterized.class)
31 public class IteratorValidTest extends AbstractConsoleLoggingTest {
32
33     @Parameter
34     public TestData testData;
35
36     @Parameters
37     public static final List<Object[]> getTestParameters() {
38         final Object[][] parameters = new Object[TestCase.values().length][1];
39         int idx = 0;
40         for (TestCase testCase : TestCase.values()) {
41             parameters[idx][0] = BinarySearchTreeTestDataProducer
42                 .createTestData(testCase);
43             idx++;
44         }
45         return Arrays.asList(parameters);
46     }
47
48     @Test
49     public void valid() {
50         // -- Given --
51         final Iterator<Integer> it = new BinarySearchTreeIterator<Integer>(
52             testData.root);
53
54         // -- When --
55         for (int i = 0; i < testData.orderedValues.size(); i++) {
56             // -- Then --

```



## Übung 3

---

```
55     assertEquals(testData.orderedValues.get(i), it.next());
56   }
57 }
58 }
```

## Übung 3

### 1.3.3 AddTest.java

Folgende Testklasse testet das hinzufügen von Elementen in den Baum.

Listing 14: AddTest.java

```

1 package at.fh.ooe.swe4.test.collections.binarySearchTreeSet.impl;
2
3 import static org.junit.Assert.assertEquals;
4 import static org.junit.Assert.assertFalse;
5
6 import org.junit.Test;
7 import org.junit.runner.RunWith;
8 import org.junit.runners.JUnit4;
9
10 import at.fh.ooe.swe4.collections.api.SortedTreeSet;
11 import at.fh.ooe.swe4.collections.impl.BinarySearchTreeSet;
12 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
13
14 /**
15  * This test class tests the {@link BinarySearchTreeSet#add(Object)} method.
16  *
17  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
18  * @date May 27, 2015
19  */
20 @RunWith(JUnit4.class)
21 public class AddTest extends AbstractConsoleLoggingTest {
22
23     private final SortedTreeSet<Integer> set = new BinarySearchTreeSet<Integer>();
24
25     @Test
26     public void addNull() {
27         // -- Given --
28         final Integer i = null;
29
30         // -- When / Then --
31         assertFalse(set.add(i));
32     }
33
34     @Test
35     public void testLevel4() {
36         // -- Given --
37         final int height = 4;
38
39         // -- When --
40         set.add(1);
41         set.add(2);
42         set.add(3);
43         set.add(4);
44
45         // -- Then --
46         assertEquals(height, set.height());
47     }
48
49     @Test
50     public void testLevel5() {
51         // -- Given --
52         final int height = 5;
53
54         // -- When --
55         // 1. First level
56         set.add(5);
57         set.add(10);
58         set.add(15);

```

## Übung 3

```

59
60 // 2 levels after split of root
61 set.add(1);
62 // 3. Add elements to 2 level
63 set.add(2);
64 set.add(11);
65 set.add(12);
66
67 // -- Then --
68 assertEquals(height, set.height());
69 }
70
71 @Test
72 public void testLevel6() {
73 // -- Given --
74 final int height = 6;
75
76 // -- When --
77 // 1. first level
78 set.add(5);
79 set.add(10);
80 set.add(15);
81
82 // 2. 2 level after split
83 set.add(1);
84 // 3. New keys for child '5'
85 set.add(2);
86 set.add(3);
87 // 4. new keys for child '15'
88 set.add(11);
89 set.add(12);
90 // 5. 3 level after split
91 set.add(13);
92
93 // -- Then --
94 assertEquals(height, set.height());
95 }
96 }

```

## Übung 3

### 1.3.4 IteratorTest.java

Folgende Testklasse testet den Iterator auf Fehlerverhalten.

Listing 15: IteratorTest.java

```

1 package at.fh.ooe.swe4.test.collections.binarySearchTreeSet.impl;
2
3 import static org.junit.Assert.assertEquals;
4
5 import java.util.Iterator;
6 import java.util.NoSuchElementException;
7
8 import org.junit.Test;
9 import org.junit.runner.RunWith;
10 import org.junit.runners.JUnit4;
11
12 import at.fh.ooe.swe4.collections.impl.BinarySearchTreeSet;
13 import at.fh.ooe.swe4.collections.impl.BinarySearchTreeSet.BinarySearchTreeIterator;
14 import at.fh.ooe.swe4.collections.model.BinaryTreeNode;
15 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
16
17 /**
18  * This class tests the {@link BinarySearchTreeIterator} used by the
19  * {@link BinarySearchTreeSet} implementation.
20  *
21  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
22  * @date May 27, 2015
23  */
24 @RunWith(JUnit4.class)
25 public class IteratorTest extends AbstractConsoleLoggingTest {
26
27     // -- Then --
28     @Test(expected = NoSuchElementException.class)
29     public void noNodes() {
30         // -- Given --
31         final Iterator<Integer> it = new BinarySearchTreeIterator<Integer>(null);
32
33         // -- When --
34         it.next();
35     }
36
37     // -- Then --
38     @Test(expected = NoSuchElementException.class)
39     public void noMoreNodes() {
40         // -- Given --
41         final Integer expected = 1;
42         final BinaryTreeNode<Integer> node = new BinaryTreeNode<Integer>(
43             expected);
44         final Iterator<Integer> it = new BinarySearchTreeIterator<Integer>(node);
45
46         // -- When --
47         int i = 0;
48         while (i < 5) {
49             assertEquals(expected, it.next());
50             i++;
51         }
52     }
53 }

```

## Übung 3

### 1.3.5 SortedTreeSetTest.java

Folgende Testklasse testet das Interface `SortedTreeSet<T>` über einen parametrisierten Test, der beide Implementierungen mit und ohne `Comparator<T>` testet.

Listing 16: SortedTreeSetTest.java

```

1 package at.fh.ooe.swe4.test.collections.common.parametrized;
2
3 import static org.junit.Assert.assertEquals;
4 import static org.junit.Assert.assertTrue;
5
6 import java.util.ArrayList;
7 import java.util.Arrays;
8 import java.util.Collection;
9 import java.util.Collections;
10 import java.util.Comparator;
11 import java.util.HashSet;
12 import java.util.List;
13 import java.util.Random;
14 import java.util.Set;
15
16 import org.junit.Before;
17 import org.junit.Test;
18 import org.junit.runner.RunWith;
19 import org.junit.runners.Parameterized;
20 import org.junit.runners.Parameterized.Parameter;
21 import org.junit.runners.Parameterized.Parameters;
22
23 import at.fh.ooe.swe4.collections.api.SortedTreeSet;
24 import at.fh.ooe.swe4.collections.impl.BinarySearchTreeSet;
25 import at.fh.ooe.swe4.collections.impl.TwoThreeFourTreeSet;
26 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
27
28 /**
29  * This test class tests the Sorted
30  *
31  * @author Thomas Herzog <thomas.hertzog@students.fh-hagenberg.at>
32  * @date May 27, 2015
33  */
34 @RunWith(Parameterized.class)
35 public class SortedTreeSetTest extends AbstractConsoleLoggingTest {
36
37     private SortedTreeSet<Integer> instance;
38
39     @Parameter(0)
40     public Class<? extends SortedTreeSet<Integer>> clazz;
41     @Parameter(1)
42     public Comparator<Integer> comparator;
43
44     @Parameters
45     public static Collection<Object[]> prepareParameters() {
46         return Arrays.asList(new Object[] [] {
47             { BinarySearchTreeSet.class, null },
48             { TwoThreeFourTreeSet.class, null },
49             { BinarySearchTreeSet.class, new Comparator<Integer>() {
50                 @Override
51                 public int compare(Integer o1, Integer o2) {
52                     return o1.compareTo(o2);
53                 }
54             } }, { TwoThreeFourTreeSet.class, new Comparator<Integer>() {
55                 @Override
56                 public int compare(Integer o1, Integer o2) {

```

## Übung 3

```

57         return o1.compareTo(o2);
58     }
59     } } });
60 }
61
62 /**
63  * Initializes an instance for the injected class.
64  */
65 @Before
66 public void before() {
67     try {
68         if (comparator == null) {
69             instance = clazz.newInstance();
70         } else {
71             instance = clazz.getConstructor(Comparator.class).newInstance(
72                 comparator);
73         }
74     } catch (Throwable e) {
75         throw new IllegalStateException("Could not prepare test");
76     }
77 }
78
79 @Test
80 public void addDuplicates() {
81     // -- Given --
82     final Integer value = 1;
83
84     // -- When --
85     instance.add(value);
86     instance.add(value);
87     instance.add(value);
88
89     // -- Then --
90     assertEquals(1, instance.size());
91     assertTrue(Arrays.equals(new Integer[] { 1 },
92         instance.toArray(new Integer[instance.size()]));
93 }
94
95 @Test
96 public void addNull() {
97     // -- Given --
98     final Integer value = null;
99
100    // -- When --
101    instance.add(value);
102
103    // -- Then --
104    assertEquals(0, instance.size());
105    assertTrue(Arrays.equals(new Integer[] {},
106        instance.toArray(new Integer[instance.size()]));
107 }
108
109 @Test
110 public void addAscendingSorted() {
111     // -- Given --
112     final int expectedSize = 10;
113     final Integer[] expectedElements = new Integer[expectedSize];
114     for (int i = 0; i < expectedSize; i++) {
115         expectedElements[i] = i + 1;
116
117         // -- When --
118         instance.add(expectedElements[i]);
119     }

```

# Übung 3

```

120
121 // -- Then --
122 assertEquals(expectedSize, instance.size());
123 assertTrue(Arrays.equals(expectedElements,
124     instance.toArray(new Integer[instance.size()]));
125 }
126
127 @Test
128 public void addDescendingSorted() {
129     // -- Given --
130     final int expectedSize = 10;
131     final List<Integer> expectedElements = new ArrayList<>(expectedSize);
132     for (int i = expectedSize; i > 0; i--) {
133         final int value = i + 1;
134         expectedElements.add(value);
135
136         // -- When --
137         instance.add(value);
138     }
139     Collections.sort(expectedElements);
140
141     // -- Then --
142     assertEquals(expectedSize, instance.size());
143     assertTrue(Arrays.equals(
144         expectedElements.toArray(new Integer[expectedSize]),
145         instance.toArray(new Integer[instance.size()]));
146 }
147
148 @Test
149 public void addRandomly() {
150     // -- Given --
151     final int expectedSize = 10;
152     final Random rand = new Random();
153     final Set<Integer> tmpSet = new HashSet<>();
154     while (tmpSet.size() < 10) {
155         final int value = rand.nextInt(1000) + 1;
156         tmpSet.add(value);
157     }
158     final List<Integer> expectedElements = new ArrayList<Integer>(tmpSet);
159     Collections.sort(expectedElements);
160
161     // -- When --
162     for (Integer value : tmpSet) {
163         instance.add(value);
164     }
165
166     // -- Then --
167     assertEquals(expectedSize, instance.size());
168     assertTrue(Arrays.equals(
169         expectedElements.toArray(new Integer[expectedSize]),
170         instance.toArray(new Integer[instance.size()]));
171 }
172 }

```

## Übung 3

### 1.3.6 TwoThreeFourTreeSetDataProducer.java

Folgend ist die Implementierung eines Testdaten Producers angeführt, welcher Testdaten produziert, also fertig und manuell aufgebaute Bäume.

Listing 17: TwoThreeFourTreeSetDataProducer.java

```

1  package at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.api;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.List;
6
7  import at.fh.ooe.swe4.collections.impl.TwoThreeFourTreeSet;
8  import at.fh.ooe.swe4.collections.model.NMKTTreeNode;
9
10 /**
11  * This class provides the test data for the {@link TwoThreeFourTreeSet} tests,
12  * where dedicated trees are build via the {@link NMKTTreeNode} model.
13  *
14  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
15  * @date May 27, 2015
16  */
17 public class TwoThreeFourTreeSetDataProducer {
18
19     /**
20      * Used for the paramter build for the parameterized test.
21      *
22      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
23      * @date May 27, 2015
24      */
25     public static enum TestCase {
26         TREE_1, TREE_2;
27     }
28
29     /**
30      * Model class for holding the test case specific test data.
31      *
32      * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
33      * @date May 27, 2015
34      */
35     public static class TestData {
36
37         public final List<Integer> orderedValues;
38         public final List<Integer> insertOrderedValues;
39         public final int levels;
40         public final NMKTTreeNode<Integer> root;
41
42         /**
43          * @param orderedValues
44          * @param insertOrderedValues
45          * @param root
46          * @param levels
47          */
48         public TestData(final List<Integer> orderedValues,
49             final List<Integer> insertOrderedValues,
50             final NMKTTreeNode<Integer> root, final int levels) {
51             super();
52             this.orderedValues = orderedValues;
53             this.insertOrderedValues = insertOrderedValues;
54             this.root = root;
55             this.levels = levels;
56         }
57     }
58 }

```



# Übung 3

```

57     }
58
59     /**
60      * Creates the test data for the given test case
61      *
62      * @param testCase
63      *      the test case to produce test data for
64      * @return the model instance holding the test case test data
65      */
66     public static TestData createTestData(final TestCase testCase) {
67         final int size = 10;
68         List<Integer> orderedValues = new ArrayList<Integer>(size);
69         switch (testCase) {
70             case TREE_1:
71                 // -- Test should be able to walk this tree
72                 // ----- 6 -----
73                 // ----- 3 ----- 8 -----
74                 // ----- / -- \ ----- / -- | ---- \ -----
75                 // -- 1-2 ---- 4-5 --- 7 ---- 9-10 --- 12-13 -----
76                 // -----
77                 final NMKTreeNode<Integer> l1 = new NMKTreeNode<Integer>();
78                 l1.addKey(1);
79                 l1.addKey(2);
80                 orderedValues.add(1);
81                 orderedValues.add(2);
82
83                 final NMKTreeNode<Integer> l2 = new NMKTreeNode<Integer>();
84                 l2.addKey(4);
85                 l2.addKey(5);
86                 orderedValues.add(4);
87                 orderedValues.add(5);
88
89                 final NMKTreeNode<Integer> r1 = new NMKTreeNode<Integer>(7);
90                 orderedValues.add(7);
91
92                 final NMKTreeNode<Integer> r2 = new NMKTreeNode<Integer>();
93                 r2.addKey(9);
94                 r2.addKey(10);
95                 orderedValues.add(9);
96                 orderedValues.add(10);
97
98                 final NMKTreeNode<Integer> r3 = new NMKTreeNode<Integer>();
99                 r2.addKey(12);
100                r2.addKey(13);
101                orderedValues.add(12);
102                orderedValues.add(13);
103
104                final NMKTreeNode<Integer> l1l2 = new NMKTreeNode<Integer>(
105                    3);
106                l1l2.addChild(l1);
107                l1l2.addChild(l2);
108                orderedValues.add(3);
109
110                final NMKTreeNode<Integer> r1r2 = new NMKTreeNode<Integer>(
111                    8);
112                r1r2.addChild(r1);
113                r1r2.addChild(r2);
114                orderedValues.add(8);
115
116                final NMKTreeNode<Integer> root = new NMKTreeNode<Integer>(
117                    6);
118                root.addChild(l1l2);
119                root.addChild(r1r2);

```

# Übung 3

```

120     orderedValues.add(6);
121
122     Collections.sort(orderedValues);
123
124     return new TestData(orderedValues, orderedValues, root, 3);
125 case TREE_2:
126     // -- Test should be able to walk this tree
127     // ----- 12 -----
128     // ----- 4-8 ----- 16-20 -----
129     // --- / - | --- \ ----- / - | ----- \ -----
130     // 1-2-3 - 5-6-7 - 9-10-11 -- 13-14-15 - 17-18-19 - 21-22-23 -----
131     // -----
132     final NMKTreeNode<Integer> _l1 = new NMKTreeNode<Integer>();
133     _l1.addKey(1);
134     _l1.addKey(2);
135     _l1.addKey(3);
136     orderedValues.add(1);
137     orderedValues.add(2);
138     orderedValues.add(3);
139
140     final NMKTreeNode<Integer> _l2 = new NMKTreeNode<Integer>();
141     _l2.addKey(5);
142     _l2.addKey(6);
143     _l2.addKey(7);
144     orderedValues.add(5);
145     orderedValues.add(6);
146     orderedValues.add(7);
147
148     final NMKTreeNode<Integer> _l3 = new NMKTreeNode<Integer>();
149     _l3.addKey(9);
150     _l3.addKey(10);
151     _l3.addKey(11);
152     orderedValues.add(9);
153     orderedValues.add(10);
154     orderedValues.add(11);
155
156     final NMKTreeNode<Integer> _r1 = new NMKTreeNode<Integer>();
157     _r1.addKey(13);
158     _r1.addKey(14);
159     _r1.addKey(15);
160     orderedValues.add(13);
161     orderedValues.add(14);
162     orderedValues.add(15);
163
164     final NMKTreeNode<Integer> _r2 = new NMKTreeNode<Integer>();
165     _r2.addKey(17);
166     _r2.addKey(18);
167     _r2.addKey(19);
168     orderedValues.add(17);
169     orderedValues.add(18);
170     orderedValues.add(19);
171
172     final NMKTreeNode<Integer> _r3 = new NMKTreeNode<Integer>();
173     _r3.addKey(21);
174     _r3.addKey(22);
175     _r3.addKey(23);
176     orderedValues.add(21);
177     orderedValues.add(22);
178     orderedValues.add(23);
179
180     final NMKTreeNode<Integer> _l1l2l3 = new NMKTreeNode<Integer>();
181     _l1l2l3.addChild(_l1);
182     _l1l2l3.addChild(_l2);

```

# Übung 3

```

183     _l1l2l3.addChild(_l3);
184     _l1l2l3.addKey(4);
185     _l1l2l3.addKey(8);
186     orderedValues.add(4);
187     orderedValues.add(8);
188
189     final NMKTreeNode<Integer> _r1r2r3 = new NMKTreeNode<Integer>();
190     _r1r2r3.addChild(_r1);
191     _r1r2r3.addChild(_r2);
192     _r1r2r3.addChild(_r3);
193     _r1r2r3.addKey(16);
194     _r1r2r3.addKey(20);
195     orderedValues.add(16);
196     orderedValues.add(20);
197
198     final NMKTreeNode<Integer> _root = new NMKTreeNode<Integer>();
199     _root.addChild(_l1l2l3);
200     _root.addChild(_r1r2r3);
201     _root.addKey(12);
202     orderedValues.add(12);
203
204     Collections.sort(orderedValues);
205
206     return new TestData(orderedValues, orderedValues, _root, 3);
207 default:
208     return null;
209 }
210 }
211 }

```

## Übung 3

### 1.3.7 IteratorValidTest.java

Folgende Testklasse testen den Iterator über einen parametrisierten Test, der die Tesdaten zur Verfügung stellt.

Listing 18: IteratorValidTest.java

```

1 package at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.impl.parameterized;
2
3 import static org.junit.Assert.assertEquals;
4
5 import java.util.Arrays;
6 import java.util.Iterator;
7 import java.util.List;
8
9 import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.junit.runners.Parameterized;
12 import org.junit.runners.Parameterized.Parameter;
13 import org.junit.runners.Parameterized.Parameters;
14
15 import at.fh.ooe.swe4.collections.iterator.NMKTreeIterator;
16 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
17 import at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.api.TwoThreeFourTreeSetDataProducer;
18 import
19     ↳ at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.api.TwoThreeFourTreeSetDataProducer.TestCase;
20 import
21     ↳ at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.api.TwoThreeFourTreeSetDataProducer.TestData;
22
23 /**
24  * This is parameterized test which runs the test cases provided via
25  * {@link TwoThreeFourTreeSetDataProducer}.
26  *
27  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
28  * @date May 27, 2015
29  */
30 @RunWith(Parameterized.class)
31 public class IteratorValidTest extends AbstractConsoleLoggingTest {
32
33     @Parameter
34     public TestData testData;
35
36     @Parameters
37     public static final List<Object[]> getTestParameters() {
38         final Object[] [] parameters = new Object[TestCase.values().length][1];
39         int idx = 0;
40         for (TestCase testCase : TestCase.values()) {
41             parameters[idx][0] = TwoThreeFourTreeSetDataProducer
42                 .createTestData(testCase);
43             idx++;
44         }
45         return Arrays.asList(parameters);
46     }
47
48     @Test
49     public void valid() {
50         // -- Given --
51         final Iterator<Integer> it = new NMKTreeIterator<Integer>(testData.root);
52
53         // -- When --
54         for (int i = 0; i < testData.orderedValues.size(); i++) {
55             // -- Then --
56             assertEquals(testData.orderedValues.get(i), it.next());
57         }
58     }
59 }

```

## Übung 3

---

```
55 |      }  
56 |    }  
57 | }
```

## Übung 3

### 1.3.8 AddTest.java

Folgende Testklasse testen die Methode `boolean add(T e1)`; der 2-3-4 Baum Implementierung.

Listing 19: AddTest.java

```

1 package at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.impl;
2
3 import static org.junit.Assert.assertEquals;
4 import static org.junit.Assert.assertFalse;
5
6 import org.junit.Test;
7 import org.junit.runner.RunWith;
8 import org.junit.runners.JUnit4;
9
10 import at.fh.ooe.swe4.collections.api.SortedTreeSet;
11 import at.fh.ooe.swe4.collections.impl.TwoThreeFourTreeSet;
12 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
13
14 /**
15  * This test class tests the {@link TwoThreeFourTreeSet#add(Object)} method.
16  *
17  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
18  * @date May 27, 2015
19  */
20 @RunWith(JUnit4.class)
21 public class AddTest extends AbstractConsoleLoggingTest {
22
23     private final SortedTreeSet<Integer> set = new TwoThreeFourTreeSet<Integer>();
24
25     @Test
26     public void addNull() {
27         // -- Given --
28         final Integer i = null;
29
30         // -- When / Then --
31         assertFalse(set.add(i));
32     }
33
34     @Test
35     public void addSplitOnThreeKeys() {
36         // -- Given --
37         final int height = 2;
38
39         // -- When --
40         set.add(1);
41         set.add(2);
42         set.add(3);
43         set.add(4);
44
45         // -- Then --
46         assertEquals(height, set.height());
47     }
48
49     @Test
50     public void addTwoLevels() {
51         // -- Given --
52         final int height = 2;
53
54         // -- When --
55         // 1. First level
56         set.add(5);
57         set.add(10);
58         set.add(15);

```

## Übung 3

```

59
60 // 2 levels after split of root
61 set.add(1);
62 // 3. Add elements to 2 level
63 set.add(2);
64 set.add(11);
65 set.add(12);
66
67 // -- Then --
68 assertEquals(height, set.height());
69 }
70
71 @Test
72 public void addThreeLevels() {
73 // -- Given --
74 final int height = 3;
75
76 // -- When --
77 // 1. first level
78 set.add(5);
79 set.add(10);
80 set.add(15);
81
82 // 2. 2 level after split
83 set.add(1);
84 // 3. New keys for child '5'
85 set.add(2);
86 set.add(3);
87 // 4. new keys for child '15'
88 set.add(11);
89 set.add(12);
90 // 5. 3 level after split
91 set.add(13);
92
93 // -- Then --
94 assertEquals(height, set.height());
95 }
96 }

```

## Übung 3

### 1.3.9 IteratorTest.java

Folgende Testklasse testet das Fehlerverhalten der Iterator Implementierung.

Listing 20: IteratorTest.java

```

1 package at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.impl;
2
3 import static org.junit.Assert.assertEquals;
4
5 import java.util.Iterator;
6 import java.util.NoSuchElementException;
7
8 import org.junit.Test;
9 import org.junit.runner.RunWith;
10 import org.junit.runners.JUnit4;
11
12 import at.fh.ooe.swe4.collections.impl.TwoThreeFourTreeSet;
13 import at.fh.ooe.swe4.collections.iterator.NMKTTreeIterator;
14 import at.fh.ooe.swe4.collections.model.NMKTTreeNode;
15 import at.fh.ooe.swe4.junit.test.suite.watcher.AbstractConsoleLoggingTest;
16
17 /**
18  * This class tests the {@link NMKTTreeIterator} used by the
19  * {@link TwoThreeFourTreeSet} implementation.
20  *
21  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
22  * @date May 27, 2015
23  */
24 @RunWith(JUnit4.class)
25 public class IteratorTest extends AbstractConsoleLoggingTest {
26
27     // -- Then --
28     @Test(expected = NoSuchElementException.class)
29     public void noNodes() {
30         // -- Given --
31         final Iterator<Integer> it = new NMKTTreeIterator<Integer>(null);
32
33         // -- When --
34         it.next();
35     }
36
37     // -- Then --
38     @Test(expected = NoSuchElementException.class)
39     public void noMoreNodes() {
40         // -- Given --
41         final Integer expected = 1;
42         final NMKTTreeNode<Integer> node = new NMKTTreeNode<Integer>(
43             expected);
44         final Iterator<Integer> it = new NMKTTreeIterator<Integer>(node);
45
46         // -- When --
47         int i = 0;
48         while (i < 5) {
49             assertEquals(expected, it.next());
50             i++;
51         }
52     }
53 }

```



## Übung 3

## 1.4 Tests

Folgend sind die Testergebnisse angeführt.

Da es wenig sinnvolle Information zu loggen gab, sei hier nur ein Screenshot der JUnit Tests angeführt. Für Informationen bezüglich der Tests sei auf den Test Source verwiesen.

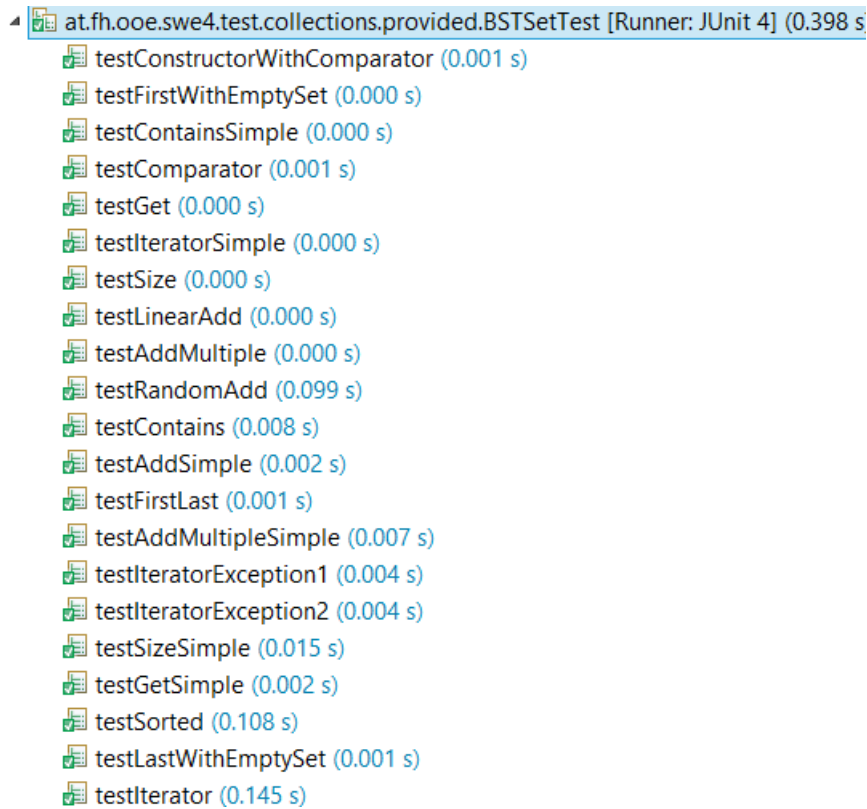
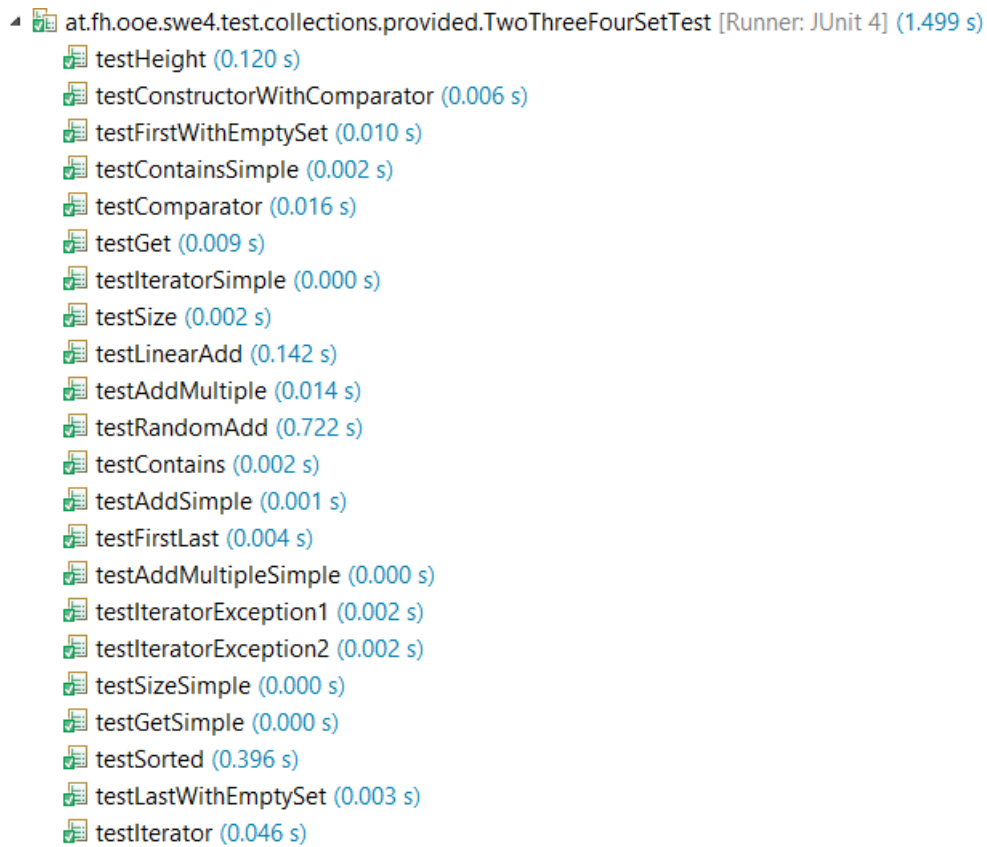


Abbildung 1: Diese Abbildung zeigt die Testergebnisse des `BinarySearchTreeSet<T>` der zur Verfügung gestellten Tests

## Übung 3



The screenshot displays the test results for the class `at.fh.ooe.swe4.test.collections.provided.TwoThreeFourSetTest` using JUnit 4. The total execution time for all tests is 1.499 seconds. Each test case is listed with a green checkmark icon, indicating a successful pass, followed by the test name and its duration in seconds.

- testHeight (0.120 s)
- testConstructorWithComparator (0.006 s)
- testFirstWithEmptySet (0.010 s)
- testContainsSimple (0.002 s)
- testComparator (0.016 s)
- testGet (0.009 s)
- testIteratorSimple (0.000 s)
- testSize (0.002 s)
- testLinearAdd (0.142 s)
- testAddMultiple (0.014 s)
- testRandomAdd (0.722 s)
- testContains (0.002 s)
- testAddSimple (0.001 s)
- testFirstLast (0.004 s)
- testAddMultipleSimple (0.000 s)
- testIteratorException1 (0.002 s)
- testIteratorException2 (0.002 s)
- testSizeSimple (0.000 s)
- testGetSimple (0.000 s)
- testSorted (0.396 s)
- testLastWithEmptySet (0.003 s)
- testIterator (0.046 s)

Abbildung 2: Diese Abbildung zeigt die Testergebnisse des `TwoThreeFourSortedTreeSet<T>` der zur Verfügung gestellten Tests

# Übung 3

```

▶ at.fh.ooe.swe4.test.collections.provided.BSTSetTest [Runner: JUnit 4] (0.398 s)
└─ at.fh.ooe.swe4.test.collections.binarySearchTreeSet.impl.IteratorTest [Runner: JUnit 4] (0.009 s)
    ├── noMoreNodes (0.007 s)
    └── noNodes (0.002 s)
└─ at.fh.ooe.swe4.test.collections.binarySearchTreeSet.impl.parametrized.IteratorValidTest [Runner: JUnit 4] (0.003 s)
    ├── [0] (0.001 s)
    │   └── valid[0] (0.001 s)
    └── [1] (0.002 s)
        └── valid[1] (0.002 s)
└─ at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.impl.AddTest [Runner: JUnit 4] (0.014 s)
    ├── addNull (0.000 s)
    ├── addTwoLevels (0.010 s)
    ├── addThreeLevels (0.002 s)
    └── addSplitOnThreeKeys (0.002 s)
▶ at.fh.ooe.swe4.test.collections.provided.TwoThreeFourSetTest [Runner: JUnit 4] (1.499 s)
└─ at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.impl.parametrized.IteratorValidTest [Runner: JUnit 4] (0.020 s)
    ├── [0] (0.003 s)
    │   └── valid[0] (0.003 s)
    └── [1] (0.017 s)
        └── valid[1] (0.017 s)
└─ at.fh.ooe.swe4.test.collections.binarySearchTreeSet.impl.AddTest [Runner: JUnit 4] (0.007 s)
    ├── addNull (0.001 s)
    ├── testLevel4 (0.002 s)
    ├── testLevel5 (0.000 s)
    └── testLevel6 (0.004 s)
└─ at.fh.ooe.swe4.test.collections.common.parametrized.SortedTreeSetTest [Runner: JUnit 4] (0.088 s)
    ├── [0] (0.006 s)
    │   ├── addDuplicates[0] (0.000 s)
    │   ├── addNull[0] (0.001 s)
    │   ├── addRandomly[0] (0.001 s)
    │   ├── addDescendingSorted[0] (0.002 s)
    │   └── addAscendingSorted[0] (0.002 s)
    ├── [1] (0.012 s)
    │   ├── addDuplicates[1] (0.002 s)
    │   ├── addNull[1] (0.004 s)
    │   ├── addRandomly[1] (0.000 s)
    │   ├── addDescendingSorted[1] (0.004 s)
    │   └── addAscendingSorted[1] (0.002 s)
    ├── [2] (0.017 s)
    │   ├── addDuplicates[2] (0.007 s)
    │   ├── addNull[2] (0.003 s)
    │   ├── addRandomly[2] (0.001 s)
    │   ├── addDescendingSorted[2] (0.004 s)
    │   └── addAscendingSorted[2] (0.002 s)
    └── [3] (0.053 s)
        ├── addDuplicates[3] (0.002 s)
        ├── addNull[3] (0.006 s)
        ├── addRandomly[3] (0.004 s)
        ├── addDescendingSorted[3] (0.003 s)
        └── addAscendingSorted[3] (0.038 s)

```

Abbildung 3: Diese Abbildung zeigt die Testergebnisse der eigenen Tests




▲  at.fh.ooe.swe4.test.collections.twoThreeFourTreeSet.impl.IteratorTest [Runner: JUnit 4] (0.005 s)  
     noMoreNodes (0.003 s)  
     noNodes (0.002 s)

Abbildung 4: Der zweite Teil der eigenen Tests