Prof. Dr. M. Müller C. Aschermann, J. Hensel, J. Protze, P. Reble

Tutoraufgabe 1 (Klassenhierarchie):

Ziel dieser Aufgabe ist die Erstellung einer Hierarchie zur Verwaltung von verschiedenen Möglichkeiten, seinen Urlaub zu verbringen.

- Ein Urlaub ist in unserem Modell ein Aktivurlaub, eine Kreuzfahrt, ein Strandurlaub oder ein Kultururlaub. Von jedem Urlaub wissen wir die Dauer (in Tagen) und den Preis des Urlaubs.
- Ein Aktivurlaub zeichnet sich dadurch aus, dass man viele Kalorien verbraucht. Deswegen ist zu jedem Aktivurlaub bekannt, wie viele Gramm Körpergewicht man durch die Anstrengung pro Tag abnimmt.
- Ein Skiurlaub ist ein spezieller Aktivurlaub. Für einen solchen Urlaub ist bekannt, wie viele Ski-Pisten in unmittelbarer Nähe zur Verfügung stehen.
- Ein Wanderurlaub ist ebenfalls ein spezieller Aktivurlaub. Die Kilometer-Anzahl der verfügbaren Wanderwege ist für jeden Wanderurlaub bekannt.
- Zusätzlich gibt es die Möglichkeit, seinen Urlaub mit einer Kreuzfahrt zu verbringen. Hierfür ist der Name des Schiffes und die Anzahl der Sterne des Bord-Restaurants bekannt.
- Alternativ zu den Aktivurlauben und Kreuzfahrten gibt es auch die Möglichkeit eines Strandurlaubs. Die Länge des Strandes zeichnet diesen Urlaub aus.
- Ein besonderer Strandurlaub ist der Party-Strandurlaub, der besonders bei jungen Leuten beliebt ist. Für einen solchen Urlaub ist besonders relevant, wie viele verschiedene Clubs direkt am Strand liegen.
- Zu guter Letzt kann auch ein Kultururlaub gebucht werden. Bei einem solchen Urlaub ist angegeben, wie viele Museen besichtigt werden können.
- Jeder Urlaub soll eine Methode `ausgabe()` besitzen, die eine Beschreibung des Urlaubs als `String` zurückgibt. Begründen Sie Ihre Entscheidung bezüglich der Frage, in welchen Klassen diese Methode implementiert werden sollte.
- Was wäre ein Urlaub ohne Urlauber: Sie sind durch ihren Namen gekennzeichnet und haben die Möglichkeit, Urlaube zu buchen.

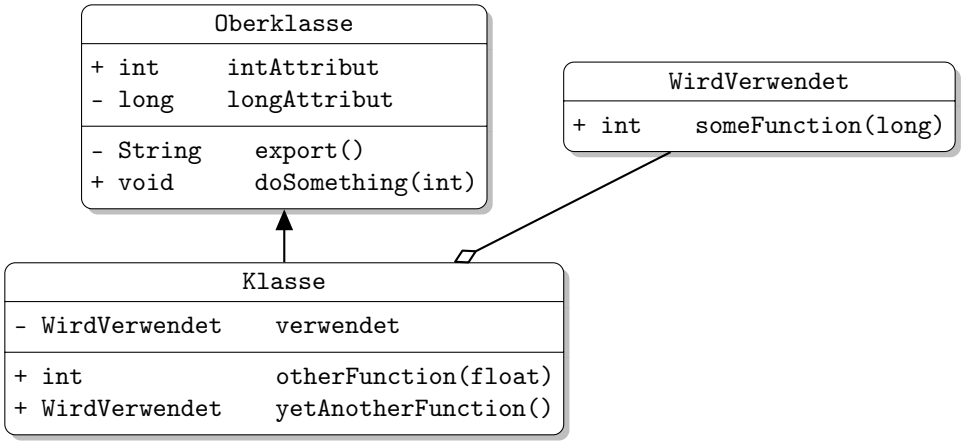
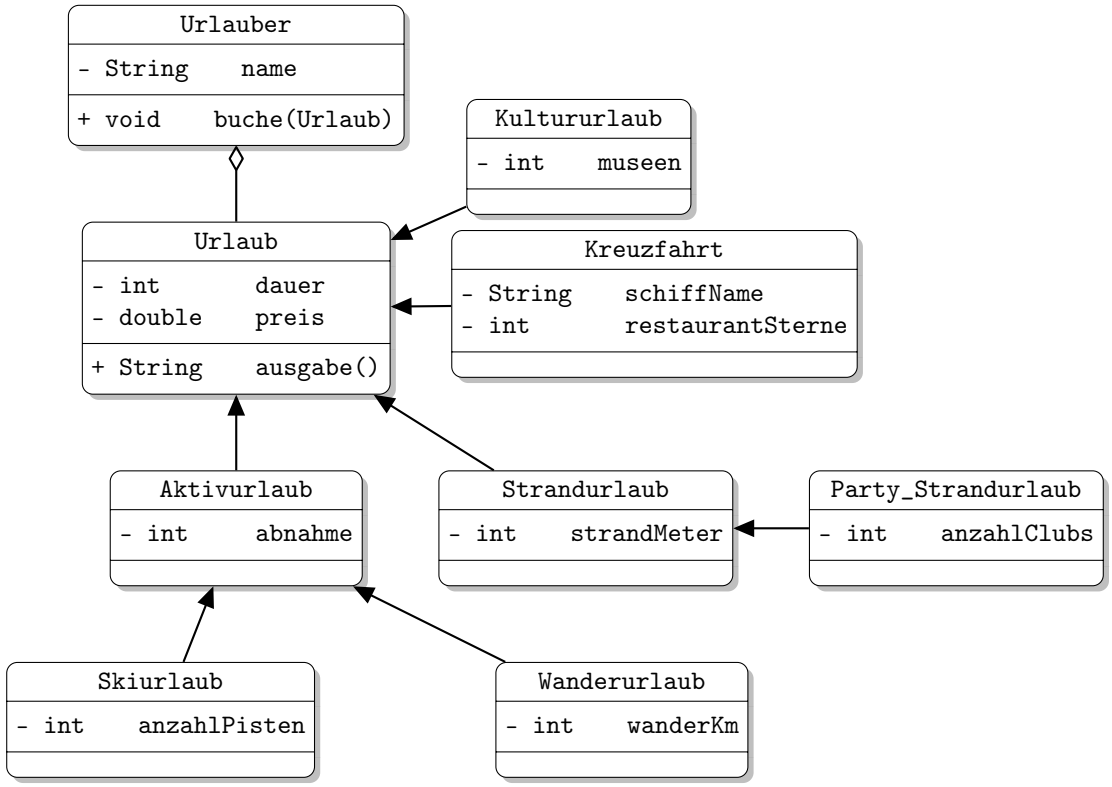


Abbildung 1: Graphische Notation zur Darstellung von Klassen.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Urlauben. Notieren Sie keine Konstruktoren. Um Schreibarbeit zu sparen, brauchen Sie keine Selektoren anzugeben. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden.

Verwenden Sie hierbei die Notation aus Abb. 1. Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \longrightarrow A$, dass A die Oberklasse von B ist (also `class B extends A`) und $A \diamond\!\!- B$, dass A den Typen B in den Typen seiner Attribute oder in den Ein- oder Ausgabeparametern seiner Methoden verwendet. Benutzen Sie ein `-` um `private` und ein `+` um `public` abzukürzen.
Tragen Sie keine vordefinierten Klassen (String, etc.) oder Pfeile dorthin in Ihr Diagramm ein.

Lösung:



Aufgabe 2 (Klassenhierarchie): (6 Punkte)

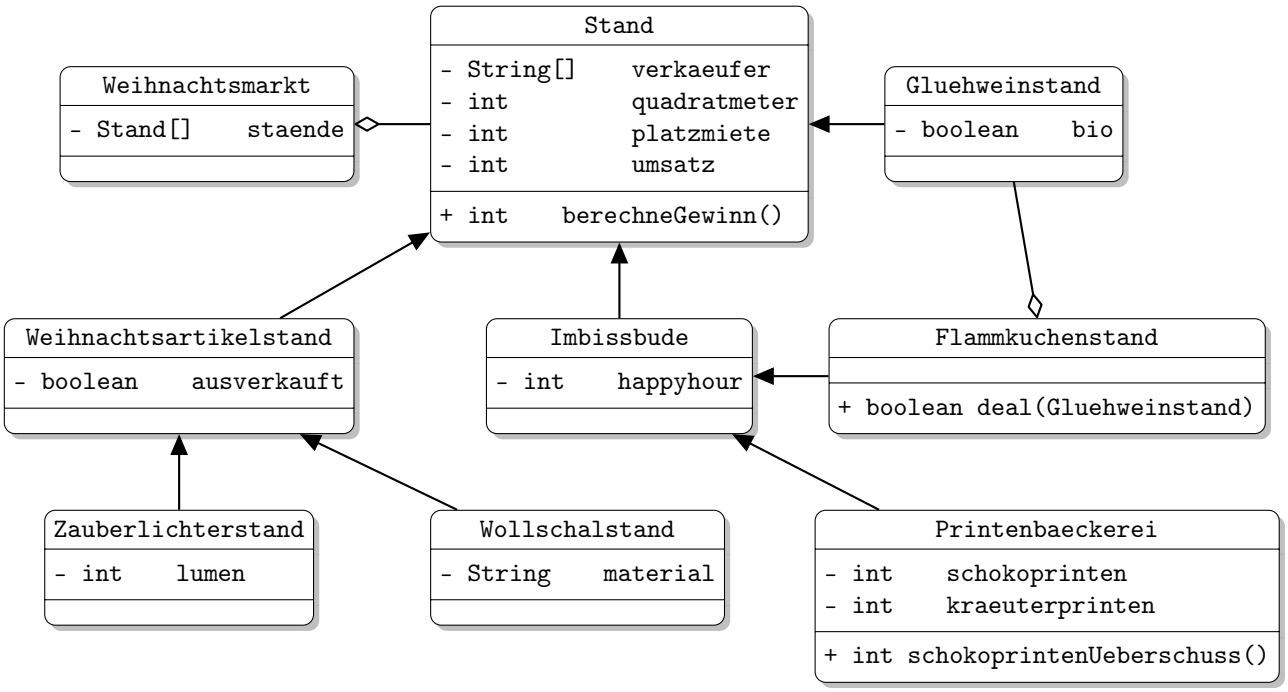
In dieser Aufgabe soll ein Teil des Aachener Weihnachtsmarktes modelliert werden.

- Ein Weihnachtsmarkt besteht aus verschiedenen Ständen.
- Ein Stand kann ein Weihnachtsartikelstand, ein Glühweinstand oder eine Imbissbude sein. Alle Stände haben einen oder mehrere Verkäufer, eine Anzahl an Quadratmetern, zahlen eine bestimmte Platzmiete und wissen ihren aktuellen Umsatz. Für jeden Stand kann aus den zur Verfügung stehenden Attributen der aktuelle Gewinn berechnet werden.
- Manche Glühweinstände haben Bio-Glühwein in ihrer Auswahl.
- Ein Weihnachtsartikelstand kann ein Zauberlichterstand oder ein Wollschalstand sein. Jeder Weihnachtsartikelstand kann irgendwann ausverkauft sein.
- Ein Zauberlichterstand wirbt mit der Mindestzahl an Lumen, die für jedes seiner Lichter garantiert wird.
- Ein Wollschalstand wirbt mit dem Material, aus dem die Schals hauptsächlich bestehen.
- Imbissbuden können Flammkuchenstände oder Printenbäckereien sein. Jede Imbissbude hat eine tägliche Uhrzeit, zu der die Happy Hour beginnt.

- Flammkuchenstände geben während der Happy Hour ihren Kunden zu jeder Portion gratis einen Glühwein mit. Dazu haben sie die Möglichkeit, mit einem beliebigen Glühweinstand einen Deal zu vereinbaren.
- Printenbäckereien wissen ihren aktuellen Bestand an Schokoladen- und an Kräuterprinten. Da Schokoladenprinten nicht so lange haltbar sind, wird zur Zeit der Happy Hour der geschätzte Tagesüberschuss berechnet, damit dieser zum Probieren an Kunden verteilt werden kann.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Ständen. Notieren Sie keine Konstruktoren. Um Schreibarbeit zu sparen, brauchen Sie keine Selektoren anzugeben. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden. Ergänzen Sie außerdem geeignete Methoden, um die Berechnung des Gewinns, die Berechnung den Schokoladenprintenüberschusses und die Vereinbarung eines Glühwein-Deals abzubilden. Notieren Sie Ihren Entwurf graphisch wie in Tutoraufgabe 1.

Lösung:



Tutoraufgabe 3 (Listen):

In dieser Aufgabe geht es um einfach verkettete Listen als Beispiel für eine dynamische Datenstruktur. Wir legen hier besonderen Wert darauf, dass eine einmal erzeugte Liste nicht mehr verändert werden kann. Achten Sie also in der Implementierung darauf, dass die Attribute der einzelnen Listen-Elemente `nur` im Konstruktor geschrieben werden.
Für diese Aufgabe benötigen Sie die Klasse `ListExercise.java`, welche Sie von unserer Webseite herunter laden können.
In der gesamten Aufgabe dürfen Sie **keine Schleifen** verwenden (die Verwendung von Rekursion ist hingegen erlaubt). Ergänzen Sie in Ihrer Lösung für alle öffentlichen Methoden außer Konstruktoren und Selektoren geeignete `javadoc`-Kommentare.

a) Erstellen Sie eine Klasse `List`, die eine einfach verkettete Liste als rekursive Datenstruktur realisiert. Die Klasse `List` muss dabei mindestens die folgenden öffentlichen Methoden und Attribute enthalten:

- `static final List EMPTY` ist die einzige `List`-Instanz, die die *leere* Liste repräsentiert

- `List(List n, int v)` erzeugt eine neue Liste, bestehend aus `v`, gefolgt von allen Elementen der Liste `n`
- `List getNext()` liefert die von `this` referenzierte Liste ohne ihr erstes Element zurück
- `int getValue()` liefert das erste Element der Liste zurück

b) Implementieren Sie in der Klasse `List` die öffentlichen Methoden `int length()` und `String toString()`. Die Methode `length` soll die Länge der Liste zurück liefern. Die Methode `toString` soll eine textuelle Repräsentation der Liste zurück liefern, wobei die Elemente der Liste durch Kommata separiert hintereinander stehen. Beispielsweise ist die textuelle Repräsentation der Liste mit den Elementen 2, 3 und 1 der `String "2, 3, 1"`.

c) Ergänzen Sie diese Klasse darüber hinaus noch um eine Methode `getSublist`, welche ein Argument i vom Typ `int` erhält und eine unveränderliche Liste zurück liefert, welche die ersten i Elemente der aktuellen Liste enthält. Sollte die aktuelle Liste nicht genügend Elemente besitzen, wird einfach eine Liste mit allen Elementen der aktuellen Liste zurück gegeben.

d) Vervollständigen Sie die Methode `merge` in der Klasse `ListExercise.java`. Diese Methode erhält zwei Listen als Eingabe, von denen wir annehmen, dass diese bereits aufsteigend sortiert sind. Sie soll eine Liste zurück liefern, die alle Elemente der beiden übergebenen Listen in aufsteigender Reihenfolge enthält.

Hinweise:

- Verwenden Sie zwei Zeiger, die jeweils auf das kleinste noch nicht in die Ergebnisliste eingefügte Element in den Argumentlisten zeigen. Vergleichen Sie die beiden Elemente und fügen Sie das kleinere ein, wobei Sie den entsprechenden Zeiger ein Element weiter rücken. Sobald eine der Argumentlisten vollständig eingefügt ist, können die Elemente der anderen Liste ohne weitere Vergleiche hintereinander eingefügt werden.

e) Vervollständigen Sie die Methode `mergesort` in der Klasse `ListExercise.java`. Diese Methode erhält eine unveränderliche Liste als Eingabe und soll eine Liste mit den gleichen Elementen in aufsteigender Reihenfolge zurückliefern. Falls die übergebene Liste weniger als zwei Elemente enthält, soll sie unverändert zurück geliefert werden. Ansonsten soll die übergebene Liste mit der vorgegebenen Methode `divide` in zwei kleinere Listen aufgespalten werden, welche dann mit `mergesort` sortiert und mit `merge` danach wieder zusammengefügt werden.

Hinweise:

- Sie können die ausführbare `main`-Methode verwenden, um das Verhalten Ihrer Implementierung zu überprüfen. Um beispielsweise die unveränderliche Liste `[2,4,3]` sortieren zu lassen, rufen Sie die `main`-Methode durch `java ListExercise 2 4 3` auf.

Lösung:

Listing 1: List.java

```
public class List {

  public static final List EMPTY = new List(null, 0);

  private final List next;
  private final int value;

  public List(List n, int v) {
    this.next = n;
    this.value = v;
  }

  public List getNext() {
    return this.next;
  }

  public int getValue() {
    return this.value;
  }

  /**
   * @return true iff this list is empty
   */
  public boolean isEmpty() {
```

```
      return this == EMPTY;
    }

    /**
     * @return a String representation of this list
     */
    public String toString() {
      if (this.isEmpty()) {
        return "";
      } else if (this.next.isEmpty()) {
        return String.valueOf(this.value);
      } else {
        return this.value + ", " + this.next.toString();
      }
    }

    /**
     * @return the length of the list
    */
    public int length() {
      if (this.isEmpty()) {
        return 0;
      } else {
        return 1 + this.next.length();
      }
    }

    /**
     * Computes a list containing the first <code>length</code> elements
     * of the current list. If this list does not contain enough
     * elements, the whole list is returned instead.
     * @param length the length of the sublist to compute
     * @return the computed sublist
     */
    public List getSublist(int length) {
      if (length <= 0 || this.isEmpty()) {
        return EMPTY;
      } else {
        List newNext = this.getNext().getSublist(length - 1);
        return new List(newNext, this.value);
      }
    }

}
```

Listing 2: ListExercise.java

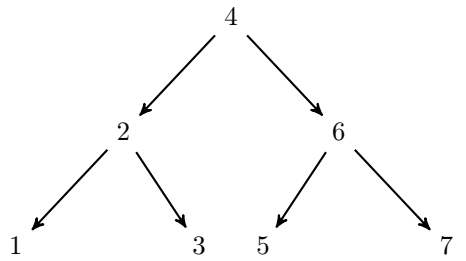
```
public class ListExercise {

    /**
     * Sorts the given list.
     * @param list the list that will be sorted
     * @return the sorted list
     */
    public static List mergesort(List list) {
        if (list.isEmpty() || list.getNext().isEmpty()) {
            return list;
        } else {
            List[] twoLists = divide(list);
            List newListA = mergesort(twoLists[0]);
            List newListB = mergesort(twoLists[1]);
            return merge(newListA, newListB);
        }
    }

    /*
     * Merges two sorted non-empty lists to one sorted list.
     */
    private static List merge(List first, List second) {
        if (first.isEmpty()) {
            return second;
        }
        if (second.isEmpty()) {
            return first;
        }
        if (first.getValue() > second.getValue()) {
            return new List(merge(first, second.getNext()), second.getValue());
        } else {
            return new List(merge(first.getNext(), second), first.getValue());
        }
    }

    /*
```

```
     * Divides a list of at least two elements into two lists of the same
     * length (up to rounding).
     */
    private static List[] divide(List list) {
        List[] res = new List[2];
        int length = list.length() / 2;
        res[0] = list.getSublist(length);
        for (int i = 0; i < length; i++) {
            list = list.getNext();
        }
        res[1] = list;
        return res;
    }

    /*
     * Creates a list from the given inputs and outputs the sorted list and
     * the original list thereafter.
     */
    public static void main(String[] args) {
        if (args != null && args.length > 0) {
            List list = buildList(0,args);
            System.out.println(mergesort(list));
            System.out.println(list);
        }
    }

    /*
     * Builds a list from the given input array.
     */
    private static List buildList(int i, String[] args) {
        if (i < args.length) {
            return new List(buildList(i + 1,args), Integer.parseInt(args[i]));
        } else {
            return List.EMPTY;
        }
    }

}
```

Aufgabe 4 (Bäume): (2,5 + 1,5 + 3 + 2 + 4 = 13 Punkte)

In dieser Aufgabe geht es um einfach verkettete Bäume als weiteres Beispiel für eine dynamische Datenstruktur. In der gesamten Aufgabe dürfen Sie **keine Schleifen** verwenden (die Verwendung von Rekursion ist hingegen erlaubt). Sie können Ihre Implementierung mit der auf der Webseite zur Verfügung gestellten Klasse `TreeExercise.java` testen.

a) Schreiben Sie eine Klasse `Tree` mit drei Attributen `value` vom Typ `int`, `left` vom Typ `Tree` und `right` vom Typ `Tree`. Alle Attribute sollen `private` sein. Diese Klasse soll genau einen Konstruktor enthalten, welcher nur ein `int`-Argument zur Belegung des `value`-Attributs erhält und die Attribute `left` und `right` jeweils auf den Wert `null` setzt. Versehen Sie diese Klasse mit Getter- und Setter-Methoden (d.h. mit Selektoren) für alle Attribute. Fügen Sie darüber hinaus noch die Methoden `toString`, `copy`, `getMin` und `getMax` jeweils ohne Argumente hinzu. Die Methode `copy` soll eine Kopie des aktuellen Baumes liefern, sodass Modifikationen an der Kopie keine Modifikationen am ursprünglichen Baum verursachen können. Die Methode `toString` soll eine textuelle Repräsentation des Baums als `String` zurück liefern, wobei die Elemente des Baums durch Kommata separiert hintereinander stehen. Die Reihenfolge der Elemente ist dabei die sogenannte *depth-first left-to-right* (auch: *in-order*) Reihenfolge, d.h. linke Nachfolger kommen vor dem aktuellen Wert und rechte Nachfolger erst danach. Als Beispiel ist hier eine grafische und die zugehörige textuelle Repräsentation eines Baums gegeben.



`"1,2,3,4,5,6,7"`

Die Methode `getMin` soll das linkeste und die Methode `getMax` das rechteste Element des Baumes liefern (in obigem Beispiel sind dies 1 bzw. 7).

b) Ergänzen Sie die Klasse `Tree` um eine Methode `isSorted` ohne Argumente, welche überprüft, ob der aktuelle Baum bzgl. der depth-first left-to-right Reihenfolge aufsteigend sortiert ist (d.h., für jeden Knoten gilt, dass sein gespeicherter Wert größer oder gleich allen Werten der Knoten im linken Unterbaum und kleiner oder gleich allen Werten der Knoten im rechten Unterbaum ist). Die Methode soll den entsprechenden Wahrheitswert zurückgeben.

Hinweise:

- Verwenden Sie die Methoden `getMin` und `getMax` aus der ersten Teilaufgabe.

c) Ergänzen Sie die Klasse `Tree` um die Methoden `getElement`, `getMean` und `getMedian`.

- Die Methode `getElement` bekommt ein Argument `i` vom Typ `int` übergeben und liefert den Wert des Elementes, das sich bzgl. der depth-first left-to-right Reihenfolge an Position `i` befindet, zurück. Hierbei befindet sich bei einem sortierten Baum das kleinste Element an Position 0 und das größte an Position $n-1$, wobei n die Anzahl der gespeicherten Werte ist.
- Die Methode `getMean` bekommt kein Argument übergeben und liefert das arithmetische Mittel aller im aktuellen Baum gespeicherten Werte als `double` zurück. Für den obigen Beispielbaum beträgt das arithmetische Mittel $\frac{1+2+3+4+5+6+7}{7} = 4$.
- Die Methode `getMedian` bekommt kein Argument übergeben und liefert für einen sortierten Baum den Median der gespeicherten Werte zurück. Ist die Anzahl der gespeicherten Werte gerade, ist es egal, welcher der beiden mittleren Werte zurückgegeben wird. Für den obigen Beispielbaum beträgt der Median `4`. Beachten Sie, dass der Median im Allgemeinen nicht immer im Wurzelknoten zu finden ist.

Hinweise:

- Sie dürfen beliebige private Hilfsmethoden zum Beispiel zur Berechnung der Größe des Baumes definieren.

d) Ergänzen Sie die Klasse `Tree` um eine Methode `sortedInsert`, welche ein Argument vom Typ `int` als Eingabe erhält und diesen Wert so in den aktuellen Baum einfügt, dass dieser in seiner textuellen Repräsentation aufsteigend sortiert ist, falls er dies vorher schon war. Da diese Methode nur den Baum modifiziert, ohne etwas zurück zu liefern, hat sie den Rückgabetyp `void`. Nehmen Sie in dabei keine Umstrukturierung des Baums vor (d.h. der einzige Unterschied zwischen dem Zustand des Baums vor und nach dem Aufruf von `sortedInsert` ist das neu eingefügte Element).

e) Nun wollen wir auch eine Methode zum Löschen von Werten schreiben. Dabei stellt sich uns allerdings das Problem, dass ein Objekt sich nicht selbst auf `null` setzen kann, sodass das Löschen des letzten Elements aus einem Baum nicht vom Baum-Objekt selbst erledigt werden kann. In der Vorlesung haben Sie zur Lösung dieses Problems bei Listen die Verwendung zweier Klassen kennen gelernt. Hier verwenden wir stattdessen eine statische Methode. Ergänzen Sie die Klasse `Tree` um eine statische Methode `sortedDelete`, welche zwei Argumente vom Typ `Tree` und `int` als Eingabe erhält und einen neuen Baum zurück liefert, in welchem ein Element mit dem übergebenen Wert aus dem übergebenen Baum gelöscht wurde (bei mehreren Vorkommen dieses Wertes wird nur einer gelöscht und falls kein solches Element existiert, wird einfach kein Element gelöscht). Dabei soll die textuelle Repräsentation des resultierenden Baums aufsteigend sortiert sein, falls sie es vorher auch war. Achten Sie darauf, dass der übergebene Baum nicht modifiziert wird und dass Modifikationen am Ergebnisbaum keine Modifikationen am übergebenen Baum verursachen können.

Hinweise:

- Verwenden Sie die Methoden `getMin` und `copy` aus der ersten Teilaufgabe.

Lösung:

Listing 3: Tree.java

```
public class Tree {

    private Tree left;
    private Tree right;
    private int value;

    public Tree(int v) {
        this.left = null;
        this.right = null;
        this.value = v;
    }

    /**
     * Creates a copy of this tree.
     */
    public Tree copy() {
        Tree res = new Tree(this.getValue());
        res.setLeft(this.getLeft() == null ? null : this.getLeft().copy());
        res.setRight(this.getRight() == null ? null : this.getRight().copy());
        return res;
    }

    public Tree getLeft() {
        return this.left;
    }

    /**
     * Returns the value of the element at the given position in the tree
     * w.r.t. the depth-first left-to-right ordering of the tree's elements.
     */
    public int getElement(int i) {
        int pos;
        if (this.getLeft() == null) {
            pos = 0;
        } else {
            pos = this.getLeft().getSize();
        }
        if (i < pos) {
            return this.getLeft().getElement(i);
        } else if (i > pos) {
            return this.getRight().getElement(i - pos - 1);
        } else {
            return this.getValue();
        }
    }

    /**
     * Returns the value of the greatest element in the tree w.r.t. the
     * depth-first left-to-right ordering of the tree's elements.
     */
    public int getMax() {
        if (this.getRight() == null) {
            return this.getValue();
        } else {
            return this.getRight().getMax();
        }
    }

    /**
     * Returns the arithmetic mean of all values in the tree.
     */
    public double getMean() {
        return ((double) this.getSum()) / this.getSize();
    }

    /**
     * Returns the median of all values in the tree.
     */
    public int getMedian() {
        int size = this.getSize();
        return this.getElement(size/2);

    }

    /**
     * Returns the value of the smallest element in the tree w.r.t. the
     * depth-first left-to-right ordering of the tree's elements.
     */
    public int getMin() {
        if (this.getLeft() == null) {
            return this.getValue();
        } else {
```

```
            return this.getLeft().getMin();
        }
    }

    public Tree getRight() {
        return this.right;
    }

    private int getSize() {
        int size = 1;
        if (this.getLeft() != null) {
            size += this.getLeft().getSize();
        }
        if (this.getRight() != null) {
            size += this.getRight().getSize();
        }
        return size;
    }

    private int getSum() {
        int sum = this.getValue();
        if (this.getLeft() != null) {
            sum += this.getLeft().getSum();
        }
        if (this.getRight() != null) {
            sum += this.getRight().getSum();
        }
        return sum;
    }

    public int getValue() {
        return this.value;
    }

    /**
     * Checks if all values in the tree are sorted w.r.t. the depth-first
     * left-to-right ordering of the tree's elements.
     */
    public boolean isSorted() {
        if (this.getLeft() != null) {
            if (this.getLeft().getMax() > this.getValue())
                return false;
            if (!this.getLeft().isSorted())
                return false;
        }
        if (this.getRight() != null) {
            if (this.getRight().getMin() < this.getValue())
                return false;
            if (!this.getRight().isSorted())
                return false;
        }
        return true;
    }

    public void setLeft(Tree left) {
        this.left = left;
    }

    public void setRight(Tree right) {
        this.right = right;
    }

    public void setValue(int value) {
        this.value = value;
    }

    /**
     * Creates a tree where one occurrence of the specified value has been
     * deleted from the specified tree. If the specified value does not occur
     * in the specified tree, a copy of the specified tree is returned.
     * The deletion works in a way that the resulting tree is sorted from the
     * smallest to the biggest element in an depth-first left-to-right way
     * (provided it was sorted that way before).
     */
    public static Tree sortedDelete(Tree tree, int v) {
        if (tree == null) {
            return null;
        } else if (tree.getValue() == v) {
            if (tree.getLeft() == null) {
                return tree.getRight() == null ? null : tree.getRight().copy();
            } else if (tree.getRight() == null) {
                return tree.getLeft() == null ? null : tree.getLeft().copy();
            } else {
                int min = tree.getRight().getMin();
```

```
                Tree res = new Tree(min);
                res.setLeft(tree.getLeft().copy());
                res.setRight(sortedDelete(tree.getRight(), min));
                return res;
            }
        } else {
            if (tree.getValue() < v) {
                Tree res = new Tree(tree.getValue());
                res.setLeft(
                    tree.getLeft() == null ? null : tree.getLeft().copy()
                );
                res.setRight(sortedDelete(tree.getRight(),v));
                return res;
            } else {
                Tree res = new Tree(tree.getValue());
                res.setLeft(sortedDelete(tree.getLeft(),v));
                res.setRight(
                    tree.getRight() == null ? null : tree.getRight().copy()
                );
                return res;
            }
        }
    }

    /**
     * Inserts the specified value into this tree such that the tree is
     * sorted from the smallest to the biggest element in an depth-first
     * left-to-right way (provided it was sorted that way before).
     */
    public void sortedInsert(int v) {
        if (this.getValue() < v) {
            if (this.getRight() == null) {
                this.setRight(new Tree(v));
            } else {
                this.getRight().sortedInsert(v);
            }
        } else {
            if (this.getLeft() == null) {
                this.setLeft(new Tree(v));
            } else {
                this.getLeft().sortedInsert(v);
            }
        }
    }

    /**
     * Returns a String representation of this tree.
     */
    public String toString() {
        return (this.getLeft() == null ? "" : this.getLeft().toString() + ", ")
            + this.getValue() +
            (this.getRight() == null ? "" : ", " + this.getRight().toString());
    }

}
```