Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Aufgabe 3 (Rekursive Datenstrukturen): (8 + 4 + 3 + 14 + 7 = 36 Punkte)

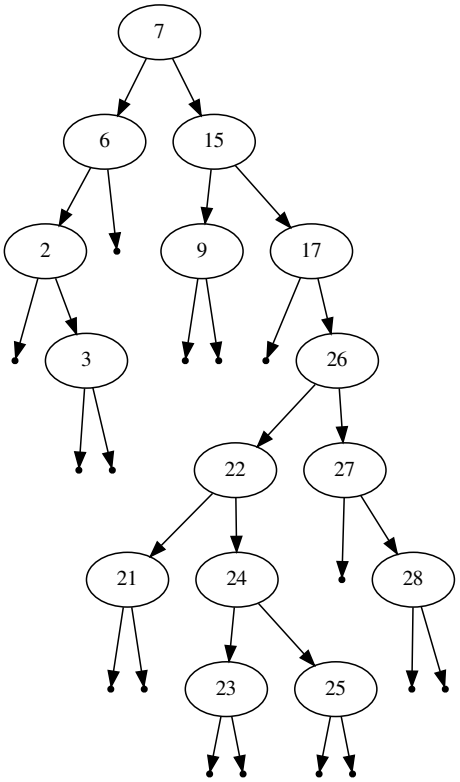
In dieser Aufgabe sollen einige rekursive Algorithmen auf sortierten Binärbäumen implementiert werden.
Aus dem Moodle-Lernraum können Sie die Klassen `BinTree` und `BinTreeNode` herunterladen. Die Klasse `BinTree` repräsentiert einen Binärbaum, entsprechend der Klasse Baum aus den Vorlesungsfolien. Einzelne Knoten des Baums werden mit der Klasse `BinTreeNode` dargestellt. Alle Methoden, die Sie implementieren, sorgen dafür, dass in dem Teilbaum `left` nur Knoten mit kleineren Werten als in der Wurzel liegen und in dem Teilbaum `right` nur Knoten mit größeren Werten.
Um den Baum zu visualisieren, ist eine Ausgabe als dot Datei bereits implementiert. In dieser einfachen Beschreibungssprache für Graphen steht eine Zeile `x -> y;` dafür, dass der Knoten `y` ein Nachfolger des Knotens `x` ist. In Dateien, die von dem vorgegebenen Code generiert wurden, steht der linke Nachfolger eines Knotens immer vor dem rechten Nachfolger in der Datei. Optional können Sie mit Hilfe der Software Graphviz, wie unten beschrieben, automatisch Bilder aus dot Dateien generieren.
Die Klasse `BinTree` enthält außerdem eine `main` Methode, die einige Teile der Implementierung testet.

Am Schluss dieser Aufgabe sollte der Aufruf `java BinTree t1.dot t2.dot` eine Ausgabe der folgenden Form erzeugen. Die Zahlen sind teilweise Zufallszahlen.

```
Aufgabe b): Zufaelliges Einfuegen
Baum als DOT File ausgegeben in Datei t1.dot
Aufgabe a): Suchen nach zufaelligen Elementen
17 ist enthalten
19 ist nicht enthalten
12 ist nicht enthalten
15 ist enthalten
12 ist nicht enthalten
13 ist nicht enthalten
3 ist enthalten
17 ist enthalten
2 ist enthalten
15 ist enthalten
26 ist enthalten
9 ist enthalten
18 ist nicht enthalten
29 ist nicht enthalten
Aufgabe c): geordnete String-Ausgabe
tree(2, 3, 6, 7, 9, 15, 17, 21, 22, 23, 24, 25, 26, 27, 28)
Aufgabe d): Suchen nach vorhandenen Elementen mit Rotation.
Baum nach Suchen von 15, 3 und 23 als DOT File ausgegeben in Datei t2.dot
Aufgabe e): merge
tree(2, 3, 6, 7, 9, 15, 17, 21, 22, 23, 24, 25, 26, 27, 28)
tree(2, 4, 5, 7, 9)
tree(2, 3, 4, 5, 6, 7, 9, 15, 17, 21, 22, 23, 24, 25, 26, 27, 28)
```

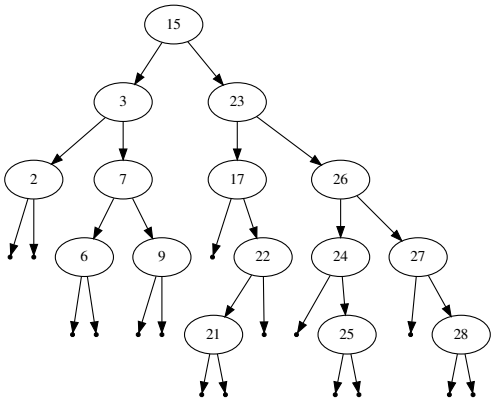
Falls Sie anschließend mit `dot -Tpdf t1.dot > t1.pdf` und `dot -Tpdf t2.dot > t2.pdf` die dot Dateien in PDF umwandeln[1], sollten Sie Bilder ähnlich zu den Folgenden erhalten.

[1] Sie benötigen hierfür das Programm Graphviz.



Listing 1: t1.dot

```
digraph {
graph [ordering="out"];
7 -> 6;
6 -> 2;
null0[shape=point]
2 -> null0;
2 -> 3;
null1[shape=point]
3 -> null1;
null2[shape=point]
3 -> null2;
null3[shape=point]
6 -> null3;
7 -> 15;
15 -> 9;
null4[shape=point]
9 -> null4;
null5[shape=point]
9 -> null5;
15 -> 17;
null6[shape=point]
17 -> null6;
17 -> 26;
26 -> 22;
22 -> 21;
null7[shape=point]
21 -> null7;
null8[shape=point]
21 -> null8;
22 -> 24;
24 -> 23;
null9[shape=point]
23 -> null9;
null10[shape=point]
23 -> null10;
24 -> 25;
null11[shape=point]
25 -> null11;
null12[shape=point]
25 -> null12;
26 -> 27;
null13[shape=point]
27 -> null13;
27 -> 28;
null14[shape=point]
28 -> null14;
null15[shape=point]
28 -> null15;
}
```



Listing 2: t2.dot

```
digraph {
graph [ordering="out"];
15 -> 3;
3 -> 2;
null0[shape=point]
2 -> null0;
null1[shape=point]
2 -> null1;
3 -> 7;
7 -> 6;
null2[shape=point]
6 -> null2;
null3[shape=point]
6 -> null3;
7 -> 9;
null4[shape=point]
9 -> null4;
null5[shape=point]
9 -> null5;
15 -> 23;
23 -> 17;
null6[shape=point]
17 -> null6;
17 -> 22;
22 -> 21;
null7[shape=point]
21 -> null7;
null8[shape=point]
21 -> null8;
null9[shape=point]
22 -> null9;
23 -> 26;
26 -> 24;
null10[shape=point]
24 -> null10;
24 -> 25;
null11[shape=point]
25 -> null11;
null12[shape=point]
25 -> null12;
26 -> 27;
null13[shape=point]
27 -> null13;
27 -> 28;
null14[shape=point]
28 -> null14;
null15[shape=point]
28 -> null15;
}
```

Wie oben erwähnt, sind die meisten Zahlen zufällig bei jedem Aufruf neu gewählt. In jedem Fall aber sollten die obersten Knoten in der zweiten Grafik die Zahlen 3, 15 und 23 sein.

In dieser Aufgabe dürfen Sie *keine* Schleifen verwenden. Die Verwendung von Rekursion ist hingegen erlaubt.

a) Implementieren Sie Methoden zum Suchen nach einer Zahl im Baum. Vervollständigen sie hierzu die Methoden `simpleSearch` in den Klassen `BinTree` und `BinTreeNode`.

Die Methode `simpleSearch` in der Klasse `BinTree` prüft, ob eine Wurzel existiert (d.h., ob der Baum nicht leer ist). Falls er leer ist, wird sofort `false` zurückgegeben. Existiert hingegen die Wurzel, wird die Methode `simpleSearch` auf der Wurzel aufgerufen.

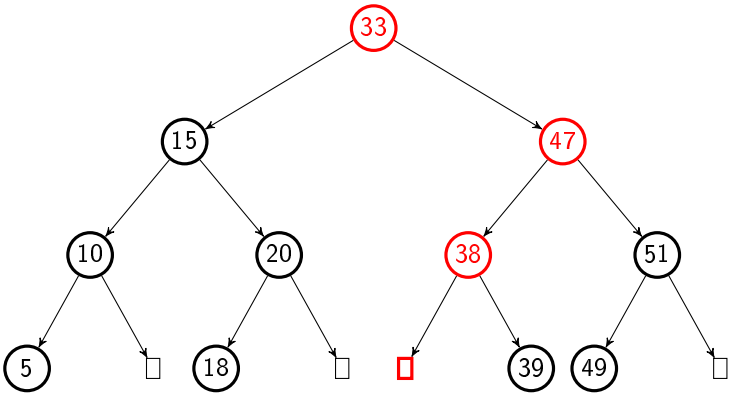
Die Methode `simpleSearch` in der Klasse `BinTreeNode` durchsucht nun den Baum nach der übergebenen Zahl. Hat der aktuelle Knoten den gesuchten Wert gespeichert, soll `true` zurückgegeben werden. Andernfalls wird eine Fallunterscheidung durchgeführt. Da der Baum sortiert ist, wird nach Zahlen, die

kleiner sind als der im aktuellen Knoten gespeicherte Wert, nur im linken Teilbaum weiter gesucht. Für Zahlen, die größer sind, muss nur im rechten Teilbaum gesucht werden. Trifft diese Suche irgendwann auf `null`, kann die Suche abgebrochen werden und es wird `false` zurückgegeben.

b) Implementieren Sie Methoden zum Einfügen einer Zahl in den Baum. Vervollständigen Sie dazu die Methoden `insert` in den Klassen `BinTreeNode` und `BinTree`.

In der Klasse `BinTree` muss zunächst überprüft werden, ob eine Wurzel existiert. Falls nein, so sollte das neue Element als Wurzel eingefügt werden. Existiert eine Wurzel, dann wird `insert` auf der Wurzel aufgerufen. In der Klasse `BinTreeNode` wird zunächst nach der einzufügenden Zahl gesucht. Wird sie gefunden, braucht nichts weiter getan zu werden (die Zahl wird also kein zweites Mal eingefügt). Existiert die Zahl noch nicht im Baum, muss ein neuer Knoten an der Stelle eingefügt werden, wo die Suche abgebrochen wurde.

Wird zum Beispiel im folgenden Baum die Zahl 36 eingefügt, beginnt die Suche beim Knoten 33, läuft dann über den Knoten 47 und wird nach Knoten 38 abgebrochen, weil der linke Nachfolger fehlt. An dieser Stelle, als linker Nachfolger von 38, wird nun die 36 eingefügt.



Hinweise:

Obwohl dem eigentlichen Einfügen eine Suche vorausgeht, ist es nicht sinnvoll, die Methode `simpleSearch` in dieser Teilaufgabe zu verwenden.

c) Schreiben Sie `toString` Methoden für die Klassen `BinTree` und `BinTreeNode`.

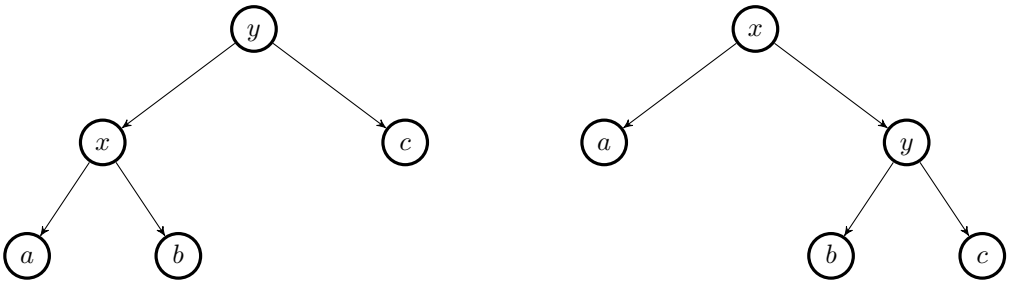
Die `toString` Methode der Klasse `BinTreeNode` soll alle Zahlen, die im aktuellen Knoten und seinen Nachfolgern gespeichert sind, aufsteigend sortiert und mit Kommas getrennt ausgeben. Ruft man beispielsweise `toString` auf dem Knoten aus dem Baum oben auf, der die Zahl 15 gespeichert hat, wäre die Ausgabe `5, 10, 15, 18, 20`.

Die `toString` Methode der Klasse `BinTree` soll die Ausgabe `tree(5, 10, 15, 18, 20, 33, 38, 39, 47, 49, 51)` für das obige Beispiel erzeugen.

d) Implementieren Sie in dieser Teilaufgabe die Methoden `search` und `rotationSearch` in der Klasse `BinTree` beziehungsweise `BinTreeNode`. Diese sollen einen alternativen Algorithmus zur Suche nach einem Wert im Baum implementieren.

Es ist sinnvoll, Elemente, nach denen häufig gesucht wird, möglichst weit oben im Baum zu speichern. Das kann realisiert werden, indem der Baum beim Aufruf der Suche so umstrukturiert wird, dass das gesuchte Element, falls es existiert, in der Wurzel steht und die übrige Struktur weitgehend erhalten wird. Da außerdem unbedingt die Sortierung erhalten bleiben muss, sollte ein spezieller Algorithmus verwendet werden.

Um einen Knoten eine Ebene im Baum nach oben zu befördern, kann die sogenannte *Rotation* verwendet werden. Soll im folgenden Beispiel x nach oben rotiert werden, wird die `left` Referenz des Vorgängerknotens y auf die `right` Referenz von x gesetzt. Anschließend wird die `right` Referenz von x auf y gesetzt. Das Ergebnis ist der rechts daneben gezeichnete Baum. Um im rechten Baum y nach oben zu rotieren, wird die Operation spiegelbildlich ausgeführt.



Diese Rotation kann nun so lange wiederholt werden, bis der Knoten mit der gesuchten Zahl in der Wurzel ist. Ist die gesuchte Zahl nicht enthalten, wird der Knoten, bei dem die Suche erfolglos abgebrochen wird, in die Wurzel rotiert.

Hinweise:

Die Signatur und Dokumentation der vorgegebenen Methoden geben Ihnen weitere Hinweise, wie die Rotation eines Knotens in die Wurzel rekursiv implementiert werden kann.

e) Implementieren Sie in der Klasse `BinTree` die statische Methode `merge`, welche als Parameter eine beliebige Anzahl von `BinTree`-Objekten erhält und ein neues `BinTree`-Objekt erstellt, welches genau die Zahlen enthält, die auch in mindestens einem der übergebenen `BinTree`-Objekte enthalten waren. Das neu erstellte `BinTree`-Objekt darf keine Zahl mehrfach enthalten.

Lösung:

Listing 3: BinTree.java

```
/**
* Import von Paketen, in denen benoetigte Library-Hilfsfunktionen zu finden sind
*/
import java.util.Random ;
import java.nio.file.*;
import java.io.*;
/**
 * Implementiert einen sortierten Binaerbaum mit Rotation-zur-Wurzel Optimierung.
 */
public class BinTree {
  /**
   * Wurzel des Baums
   */
  private BinTreeNode root;
  /**
   * Erstellt einen leeren Baum
   */
  public BinTree() {
    this.root = null ;
  }
  /**
   * Erstellt einen Baum mit den vorgegebenen Zahlen
   * @param xs die einzupflegenden Zahlen
   */
  public BinTree(int... xs) {
    for ( int x : xs ) {
      this.insert(x);
    }
  }
  /**
   * Test ob der Baum leer ist
   * @return true, falls der Baum leer ist, sonst false
   */
  public boolean isEmpty() {
    return this.root == null ;
  }

  /**
   * Fuegt alle Zahlen aus den Baeumen in einen neuen Baum ein und gibt diesen zurueck.
   * @param trees Die Baeume mit den einzufuegenden Zahlen.
   * @return Der neue Baum mit allen Zahlen.
   */
  public static BinTree merge(BinTree... trees) {
    BinTree result = new BinTree();
    doMerge(result, trees, 0);
```

```
  return result;
}

/**
 * Fuegt alle Zahlen aus den Baeumen mit einem gegebenen Mindestindex in den Ergebnisbaum ein.
 * @param result Der Ergebnisbaum.
 * @param trees Die Baeume mit den einzufuegenden Zahlen.
 * @param index Der Mindestindex.
 */
private static void doMerge(BinTree result, BinTree[] trees, int index) {
  if (index < trees.length) {
    insertAll(result, trees[index].root);
    doMerge(result, trees, index + 1);
  }
}

/**
 * Fuegt alle Zahlen in den Ergebnisbaum ein.
 * @param result Der Ergebnisbaum.
 * @param toInsert Der Baumknoten mit den einzufuegenden Zahlen.
 */
private static void insertAll(BinTree result, BinTreeNode toInsert) {
  if (toInsert != null) {
    insertAll(result, toInsert.getLeft());
    result.insert(toInsert.getValue());
    insertAll(result, toInsert.getRight());
  }
}

/**
 * Fuegt eine Zahl ein. Keine Aenderung, wenn das Element
 * schon enthalten ist.
 * @param x einzufuegende Zahl
 */
public void insert(int x) {
  if (this.isEmpty()) {
    this.root = new BinTreeNode(x);
  } else {
    this.root.insert(x);
  }
}
/**
 * Sucht x, ohne den Baum zu veraendern.
 * @return true, falls x im Baum enthalten ist, sonst false
 * @param x der gesuchte Wert
 */
public boolean simpleSearch(int x) {
  if (this.isEmpty ()) {
    return false;
  } else {
    return this.root.simpleSearch(x);
  }
}
/**
 * Sucht x und rotiert den Knoten, bei dem die Suche nach x endet, in die Wurzel.
 * @param x der gesuchte Wert
 * @return true, falls x im Baum enthalten ist, sonst false
 */
  public boolean search(int x) {
    if (this.isEmpty()) {
      return false;
    } else {
      this.root = this.root.rotationSearch(x);
      return this.root.getValue () == x ;
    }
  }
/**
 * @return Sortierte Ausgabe aller Elemente.
 */
public String toString() {
  if ( this.isEmpty ()) {
    return "tree()";
  }
  return "tree(" + this.root.toString () + ")";
}
/**
 * Wandelt den Baum in einen Graphen im dot Format um.
 * @return der umgewandelte Baum
 */
public String toDot() {
  if ( this.isEmpty ()) {
    return "digraph { null[shape=point]; }";
  }
  StringBuilder str = new StringBuilder ();
```

```
    this.root.toDot (str, 0);
    return "digraph { " + System.lineSeparator ()
      + "graph[ordering=\"out\"]; " + System.lineSeparator ()
      + str.toString ()
      + "}" + System.lineSeparator ();
  }
  /**
   * Speichert die dot Repraesentation in einer Datei.
   *
   * @param path Pfad unter dem gespeichert werden soll (Dateiname)
   * @return true, falls erfolgreich gespeichert wurde, sonst false
   * @see toDot
   */
  public boolean writeToFile(String path) {
    boolean retval = true;
    try {
      Files.write(FileSystems.getDefault().getPath(path), this.toDot().getBytes());
    } catch (IOException x) {
      System.err.println("Es ist ein Fehler aufgetreten.");
      System.err.format("IOException: %s%n" , x);
      retval = false;
    }
    return retval;
  }
  /**
   * Main-Methode, die einige Teile der Aufgabe testet.
   *
   * @param args Liste von Dateinamen, unter denen Baeume als dot
   * gespeichert werden sollen. Es werden nur die ersten beiden verwendet.
   */
  public static void main(String[] args) {
    Random prng = new Random();
    int nodeCount = prng.nextInt(10) + 5;
    BinTree myTree = new BinTree();
    System.out.println("Aufgabe b): Zufaelliges Einfuegen");
    for(int i = 0; i < nodeCount; ++i) {
      myTree.insert(prng.nextInt(30));
    }
    myTree.insert(15);
    myTree.insert(3);
    myTree.insert(23);
    if (args.length > 0) {
      if (myTree.writeToFile(args[0])) {
        System.out.println("Baum als DOT File ausgegeben in Datei " + args [0]);
      }
    } else {
      System.out.println("Keine Ausgabe des Baums in Datei, zu wenige Aufrufparameter.");
    }
    System.out.println("Aufgabe a): Suchen nach zufaelligen Elementen");
    for(int i = 0; i < nodeCount; ++i) {
      int x = prng.nextInt (30);
      if(myTree.simpleSearch(x)) {
        System.out.println(x + " ist enthalten");
      } else {
        System.out.println(x + " ist nicht enthalten");
      }
    }
    System.out.println("Aufgabe c): geordnete String-Ausgabe");
    System.out.println(myTree.toString());
    System.out.println("Aufgabe d): Suchen nach vorhandenen Elementen mit Rotation.");
    myTree.search(3);
    myTree.search(23);
    myTree.search(15);
    if (args.length > 1) {
      if (myTree.writeToFile(args[1])) {
        System.out.println("Baum nach Suchen von 15, 3 und 23 als DOT File ausgegeben in Datei "
            + args [1]);
      }
    } else {
      System.out.println("Keine Ausgabe des Baums in Datei, zu wenige Aufrufparameter.");
    }

    System.out.println("Aufgabe e): merge");
    BinTree tree2 = new BinTree(4, 7, 2,9 ,5);
    System.out.println(myTree.toString());
    System.out.println(tree2.toString());
    System.out.println(BinTree.merge(myTree, tree2).toString());
  }
}
```

Listing 4: BinTreeNode.java

```
/**
```

```
 * Ein Knoten in einem binaeren Baum.
 *
 * Der gespeicherte Wert ist unveraenderlich,
 * die Referenzen auf die Nachfolger koennen aber
 * geaendert werden.
 *
 * Die Klasse bietet Methoden, um Werte aus einem Baum
 * zu suchen und einzufuegen. Die Methode zur Suche gibt
 * es noch in einer optimierten Variante, um
 * rotate-to-root Baeume zu verwalten.
 */
public class BinTreeNode {
  /**
   * Linker Nachfolger
   */
  private BinTreeNode left;
  /**
   * Rechter Nachfolger
   */
  private BinTreeNode right;
  /**
   * Wert, der in diesem Knoten gespeichert ist
   */
  private final int value;

  /**
   * Erzeugt einen neuen Knoten ohne Nachfolger
   * @param val Wert des neuen Knotens
   */
  public BinTreeNode(int val) {
    this.value = val;
    this.left = null;
    this.right = null;
  }

  /**
   * Erzeugt einen neuen Knoten mit den gegebenen Nachfolgern
   * @param val Wert des neuen Knotens
   * @param left linker Nachfolger des Knotens
   * @param right rechter Nachfolger des Knotens
   */
  public BinTreeNode(int val, BinTreeNode left, BinTreeNode right) {
    this.value = val;
    this.left = left;
    this.right = right;
  }
  /**
  * @return Wert des aktuellen Knotens
  *
  */
  public int getValue() {
    return this.value;
  }

  /**
   * @return Der gespeicherte Wert, umgewandelt in einen String
   */
  public String getValueString() {
    return Integer.toString(this.value);
  }

  /**
   * @return true, falls der Knoten einen linken Nachfolger hat, sonst false
   */
  public boolean hasLeft() {
    return this.left != null;
  }

  /**
   * @return true, falls der Knoten einen rechten Nachfolger hat, sonst false
   */
  public boolean hasRight() {
    return this.right != null;
  }
  /**
   * @return linker Nachfolger des aktuellen Knotens
   */
  public BinTreeNode getLeft() {
    return this.left;
  }
  /**
   * @return rechter Nachfolger des aktuellen Knotens
   */
  public BinTreeNode getRight() {
```

```
  return this.right;
}

/**
 * Sucht in diesem Teilbaum nach x, ohne den Baum zu veraendern.
 * @param x der gesuchte Wert
 * @return true, falls x enthalten ist, sonst false
 */
public boolean simpleSearch(int x) {
  if(this.value == x) {
    return true;
  } else if(this.value > x && this.hasLeft()) {
    return this.left.simpleSearch(x);
  } else if(this.value < x && this.hasRight()) {
    return this.right.simpleSearch(x);
  } else {
    return false;
  }
}

/**
 * Fuegt x in diesen Teilbaum ein.
 * @param x der einzufuegende Wert
 */
public void insert(int x) {
  if(this.value == x) {
    return;
  }else if(this.value > x) {
    if(this.hasLeft()) {
      this.left.insert(x);
    } else {
      this.left = new BinTreeNode(x);
    }
  } else {
    if(this.hasRight()) {
      this.right.insert(x);
    } else {
      this.right = new BinTreeNode(x);
    }
  }
}

/**
 * Sucht in diesem Teilbaum nach x und rotiert den Endpunkt der Suche in die
 * Wurzel.
 * @param x der gesuchte Wert
 * @return die neue Wurzel des Teilbaums
 */
public BinTreeNode rotationSearch(int x) {
  if(this.value > x && this.hasLeft()) {
    BinTreeNode root = this.left.rotationSearch(x);
    this.left = root.right;
    root.right = this;
    return root;
  } else if(this.value < x && this.hasRight()) {
    BinTreeNode root = this.right.rotationSearch(x);
    this.right = root.left;
    root.left = this;
    return root;
  } else {
    return this;
  }
}

/**
 * @return Geordnete Liste aller Zahlen, die in diesem Teilbaum gespeichert sind.
 */
public String toString() {
  String str = this.getValueString();
  if(this.hasLeft()) {
    str = this.left.toString() + ", " + str;
  }
  if(this.hasRight()) {
    str = str + ", " + this.right.toString();
  }
  return str;
}

/**
 * Erzeugt eine dot Repraesentation in str
 * @param str Stringbuilder Objekt zur Konstruktion der Ausgabe
 * @param nullNodes Hilfsvariable, um Nullknoten zu indizieren. Anfangswert sollte 0 sein.
 * @return Den nullNodes Wert fuer den behandelten Baum
 */
```

```
  public int toDot(StringBuilder str, int nullNodes) {
    if(this.hasLeft()) {
      str.append(this.getValueString() + " -> " + this.left.getValueString() + ";"
        + System.lineSeparator());
      nullNodes = this.left.toDot(str, nullNodes);
    } else {
      str.append("null" + nullNodes + "[shape=point]" + System.lineSeparator()
        + this.getValueString() + " -> null" + nullNodes + ";" + System.lineSeparator());
      nullNodes += 1;
    }
    if(this.hasRight()) {
      str.append(this.getValueString() + " -> " + this.right.getValueString() + ";"
        + System.lineSeparator());
      nullNodes = this.right.toDot(str, nullNodes);
    } else {
      str.append("null" + nullNodes + "[shape=point]" + System.lineSeparator()
        + this.getValueString() + " -> null" + nullNodes + ";" + System.lineSeparator());
      nullNodes += 1;
    }
    return nullNodes;
  }

}
```

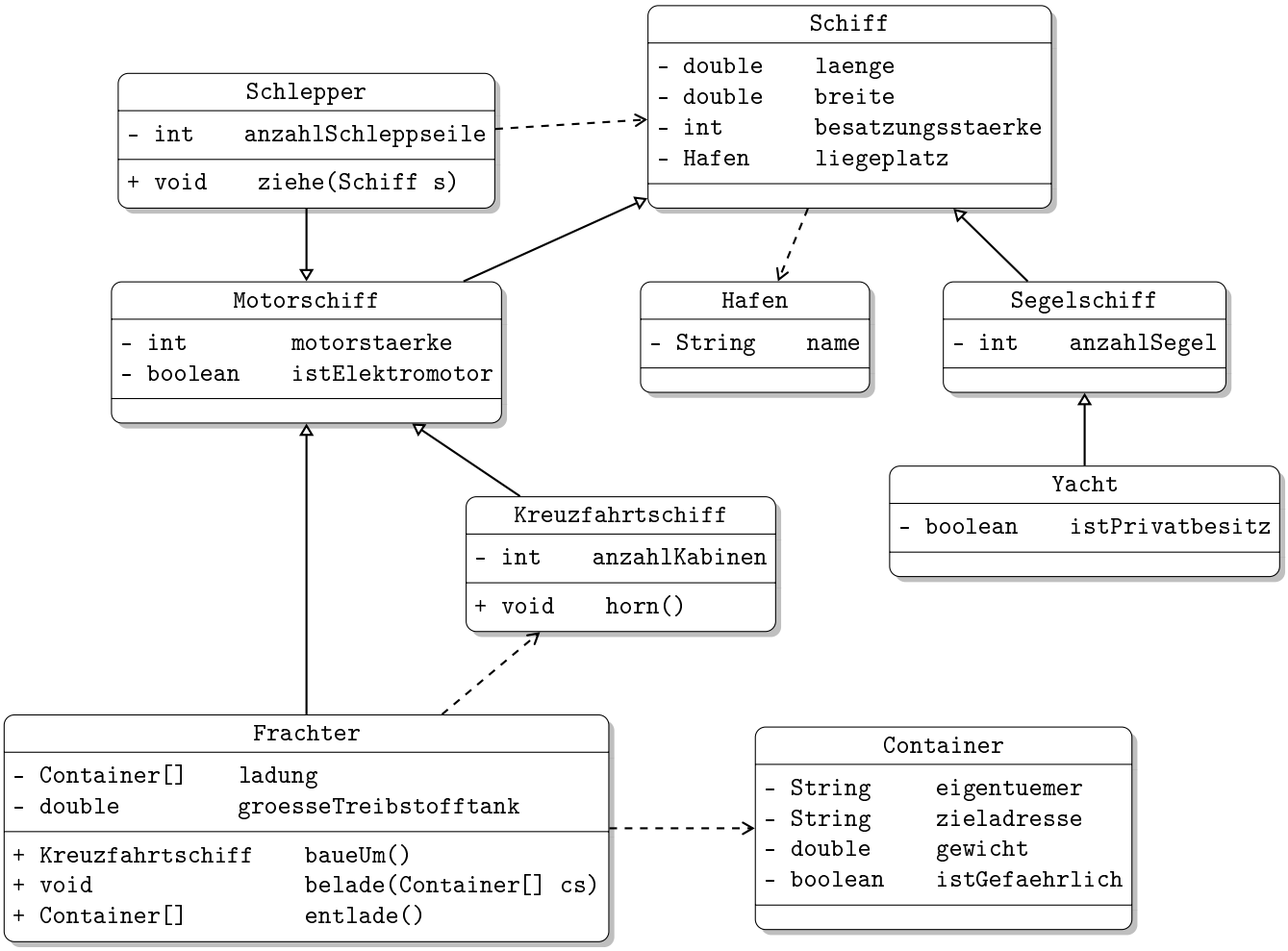
Aufgabe 5 (Entwurf einer Klassenhierarchie): (14 Punkte)

In dieser Aufgabe sollen Sie einen Teil der Schifffahrt modellieren.

- Die wichtigste Eigenschaft eines Hafens ist sein Name.
- Ein Schiff kann ein Segelschiff oder ein Motorschiff sein. Jedes Schiff hat eine Länge und eine Breite in Metern und eine Anzahl an Besatzungsmitgliedern. Ein Schiff hat seinen Liegeplatz in einem Hafen.
- Ein Segelschiff zeichnet sich durch die Anzahl seiner Segel aus.
- Die wichtigste Kennzahl eines Motorschiffs ist die PS-Stärke des Motors. Der Motor kann außerdem ein Dieselmotor oder ein Elektromotor sein.
- Ein Kreuzfahrtschiff ist ein Motorschiff. Es hat eine Anzahl an Kabinen und kann das Schiffshorn ertönen lassen.
- Eine Yacht ist ein Segelschiff. Yachten können in Privatbesitz sein oder nicht.
- Schlepper sind Motorschiffe mit einer Anzahl von Schleppseilen. Ein Schlepper kann ein beliebiges Schiff ziehen.
- Frachter sind Motorschiffe. Sie enthalten eine Ladung, die aus einer Sammlung von Containern besteht. Außerdem sind sie durch die Größe ihres Treibstofftanks in Litern gekennzeichnet. Ein Frachter kann mit Containern be- und entladen werden, wobei beim Entladen alle Container vom Frachter entfernt werden.
- Ein Container zeichnet sich durch seinen Eigentümer, seine Zieladresse und das Gewicht seines Inhalts aus. Zudem kann der Inhalt gefährlich sein oder nicht.
- In einem aufwendigen Verfahren kann ein Frachter zu einem Kreuzfahrtschiff umgebaut werden.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Schiffen. Notieren Sie keine Konstruktoren. Um Schreibarbeit zu sparen, brauchen Sie keine Selektoren anzugeben. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden, falls dies sinnvoll ist. Ergänzen Sie außerdem geeignete Methoden, um das Beladen, Entladen, das Umbauen, das Ziehen und das Erklingen lassen des Horns abzubilden.
Tragen Sie keine vordefinierten Klassen (`String`, etc.) oder Pfeile dorthin in Ihr Diagramm ein.
Verwenden Sie hierbei die Notation aus der entsprechenden Tutoriumsaufgabe.

Lösung:



Aufgabe 6 (Deck 6): (Codescape)

Lösen Sie die Missionen von Deck 6 des Codescape Spiels. Ihre Lösung für die Codescape Missionen wird nur dann für die Zulassung gezählt, wenn Sie Ihre Lösung vor der einheitlichen Codescape Deadline am Samstag, den 22.01.2022, um 23:59 Uhr abschicken.

Lösung: