Prof. Dr. J. Giesl
F. Frohn, J. Hensel, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Montag, den 12.12.2016 um 15:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor oder unmittelbar vor Beginn in der Globalübung abgeben.
- In einigen Aufgaben müssen Sie in Java programmieren und `.java`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Montag, dem 12.12.2016 um 15:00 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Einige Hausaufgaben müssen im Spiel Codescape gelöst werden:
  https://codescape.medien.rwth-aachen.de/progra/
  Diese Aufgaben werden getrennt von den anderen Hausaufgaben gewertet.
- Halten Sie sich beim Lösen von Programmieraufgaben an die auf der Website zur Vorlesung verfügbaren Codekonventionen. Verstöße, die zu unleserlichem Code führen, können zu Punktabzug führen.

Tutoraufgabe 1 (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, die mit dem Wert `v` beginnt, 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.

Aufgabe 2 (Bäume): (4 + 4 + 3 + 8 = 19 Punkte)

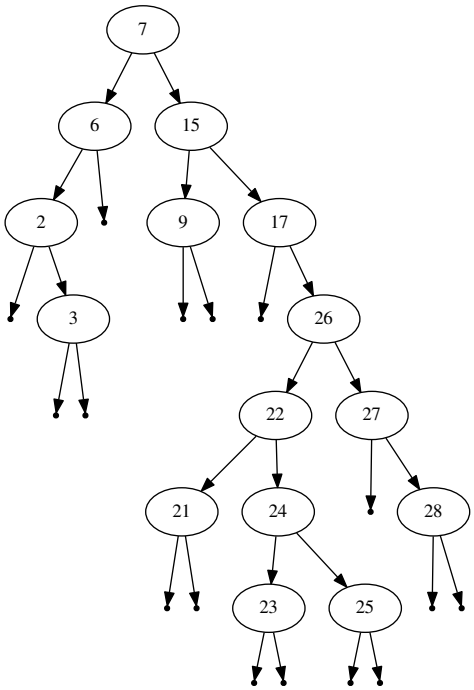
In dieser Aufgabe sollen einige rekursive Algorithmen auf sortierten Binärbäumen implementiert werden.
Von der Webseite können Sie die Klassen `Tree` und `TreeNode` herunterladen. Die Klasse `Tree` repräsentiert einen Binärbaum, entsprechend der Klasse Baum auf den Vorlesungsfolien. Einzelne Knoten des Baumes werden mit der Klasse `TreeNode` 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 `Tree` enthält außerdem eine `main` Methode, die einige Teile der Implementierung testet.

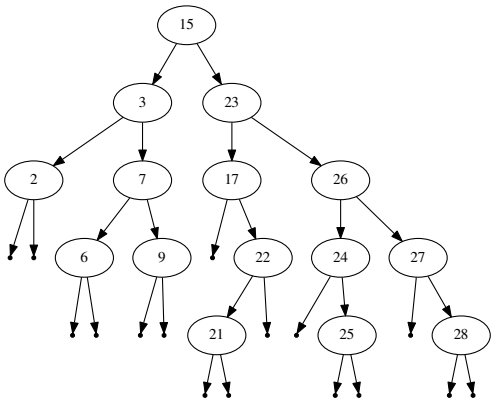
Am Schluss dieser Aufgabe sollte der Aufruf `java Tree t1.dot t2.dot` eine Ausgabe der folgenden Form erzeugen. Die Zahlen sind teilweise Zufallszahlen.

```
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
```

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.





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.

Die Methode `simpleSearch` in der Klasse `Tree` prüft, ob eine Wurzel existiert. Falls nicht, wird sofort `false` zurück gegeben. Existiert die Wurzel, wird ihre Methode `simpleSearch` aufgerufen.

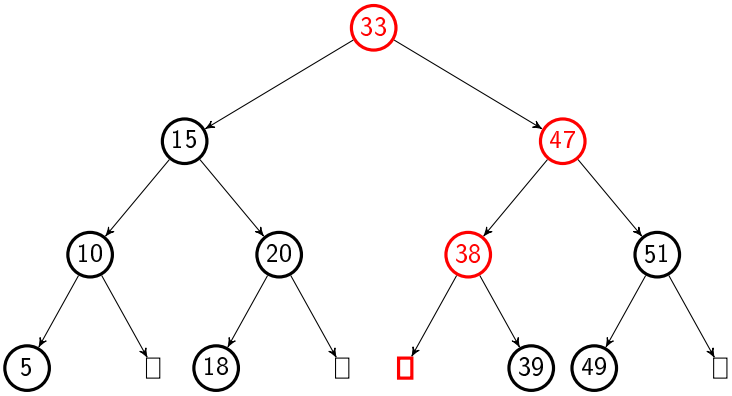
[1] Sie benötigen hierfür Graphviz

Die Methode `simpleSearch` der Klasse `TreeNode` durchsucht nun den Baum nach der übergebenen Zahl. Hat der aktuelle Knoten den gesuchten Wert gespeichert, kann `true` zurückgegeben werden. Andernfalls wird eine Fallunterscheidung gemacht. 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 `TreeNode` und `Tree`.

In der Klasse `Tree` 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 `TreeNode` 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 `Tree` und `TreeNode`.

Die `toString` Methode der Klasse `TreeNode` 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`.

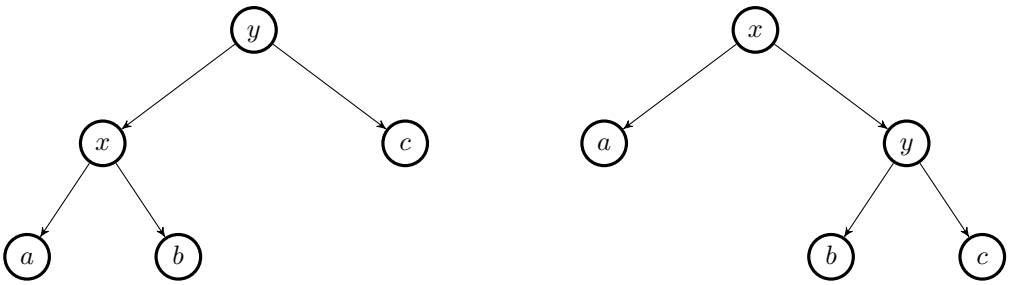
Die `toString` Methode der Klasse `Tree` soll die Ausgabe `tree[5, 10, 15, 18, 20, 33, 38, 39, 47, 49, 51]` für das obige Beispiel erzeugen.

d) Implementieren Sie in dieser Teilaufgabe die Methoden `search` und `rotationSearch` in der Klasse `Tree` beziehungsweise `TreeNode`. 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.

Aufgabe 3 (Deck 6): (Codescape)

Schließen Sie das Spiel Codescape ab, indem Sie den letzten Raum auf Deck 6 auf eine der drei möglichen Arten lösen. Genießen Sie anschließend das Outro.

Hinweise:

- Es gibt verschiedene Möglichkeiten wie die Story endet, abhängig von Ihrer Entscheidung im finalen Raum.
- Verraten Sie Ihren Kommilitonen nicht, welche Auswirkungen Ihre Entscheidung hatte, bevor diese selbst das Spiel abgeschlossen haben.