Prof. Dr. Marc Stamminger
Prof. Dr. Michael Philippsen
Department Informatik, FAU

28. Juli 2016

Klausur: Algorithmen und Datenstrukturen

Angaben zur Person (Bitte Etikett aufkleben bzw. in Druckbuchstaben ausfüllen!):

Name, Vorname: ....

Laufende Nr.: ....

Bitte kleben Sie hier das Etikett auf.

Matrikelnummer:

....

Folgende Hinweise bitte lesen und Kenntnisnahme durch Unterschrift bestätigen!

- **Hilfsmittel außer Schreibmaterialien sind *nicht* zugelassen**.
- Lösungen müssen in den dafür vorgesehenen freien Raum geschrieben werden. Sollte der Platz nicht ausreichen, verwenden Sie zunächst (mit deutlichem Hinweis) die gegebenenfalls am Ende schon vorhandene oder bei Bedarf weitere Zusatzseiten, die Ihnen von der Aufsicht ausgegeben und in die Klausur *eingeheftet* werden müssen.
- Sie können Schmierpapier von der Aufsicht anfordern. ***Das Schmierpapier darf nicht mit abgegeben werden.***
- Können Sie die Prüfung aus gesundheitlichen Gründen nicht fortsetzen, dann müssen Sie Ihre Prüfungsunfähigkeit durch Vorlage eines *erweiterten ärztlichen Attestes* beim Prüfungsamt nachweisen. Melden Sie sich in jedem Fall bei der Aufsicht und lassen Sie sich das *entsprechende Formular* aushändigen.
- ***Überprüfen Sie diese Klausur auf Vollständigkeit (14 Seiten inkl. Deckblatt) und einwandfreies Druckbild!***

Durch meine Unterschrift bestätige ich den Empfang der vollständigen Klausurunterlagen und die Kenntnisnahme der obigen Informationen.

Erlangen, den 28.07.2016 ....
(Unterschrift)

Nicht von der Kandidatin bzw. vom Kandidaten auszufüllen!

Bewertung (Punkteverteilung unter Vorbehalt):

Aufgabe	1	2	3	4	5	6	7	$\sum$
Maximal	16	21	15	21	22	10	15	120
Erreicht	2	15	12	9				38

Aufgabe 1 (Wissensfragen) (16 Punkte)

Bei den folgenden Teilaufgaben werden richtige Kreuze positiv (+) und falsche **oder fehlende** Kreuze entsprechend negativ (–) gewertet. Jede Teilaufgabe wird mit mindestens 0 Punkten bewertet. Pro Teilaufgabe ist mind. eine Aussage wahr. Kreuzen Sie alle richtigen Aussagen an:

a) Was trifft auf rekursive Java-Programme zu?

- [x] Endrekursive Methoden lassen sich immer entrekursivieren. 2P
- [x] Die komplette Ausführung einer rumpfrekursiven Methode kann stets mit einem zusätzlichen Speicherbedarf im Programmstapel von $\mathcal{O}(1)$ erfolgen.
- [x] Verschränkte Rekursion erfordert keine rekursiven Methodendefinitionen.
- [ ] Hat eine kaskadenartige Rekursion eine maximale Aufruftiefe h und -breite b, dann hat sie einen Speicherbedarf von $\mathcal{O}(b)$ auf dem Programmstapel.

b) Welche Behauptungen stimmen?

- [ ] *Überprüfte* Ausnahmen darf man nicht mit `throws` in der Signatur deklarieren. 0P
- [x] Ein zugehöriges `finally` wird trotz Ausnahme im `catch`-Block ausgeführt.
- [x] Die Annotation `@org.junit.Test` erklärt eine Methode zum Testfall.
- [ ] Mit jUnit kann man systembedingt keinen Code testen, der Ausnahmen wirft.

c) Welche Aussagen stimmen für Behälterdatentypen?

- [x] Das Vergrößern eines Feldes von n auf $n+1$ Plätze, bei dem die Einträge aus dem alten Feld in das neue kopiert werden, hat in Java eine Laufzeit von $\mathcal{O}(n)$. 0P
- [x] Das Verlängern einer aus Objekten bestehenden, doppelt verketteten Liste um ein Element kostet $\mathcal{O}(1)$ Zeit.
- [ ] Beim *geschlossenen* Hashing gibt es prinzipbedingt *keine Sekundär*kollisionen.
- [ ] Implementiert man Mengen mit sortierten verketteten Listen, dann kann man die Schnittmenge zweier Mengen mit jeweils n Elementen in $\mathcal{O}(log(n))$ bestimmen.

d) Was trifft auf Bäume zu?

- [x] Jeder *allgemeine Baum* ist auch ein *gerichteter azyklischer Graph* (DAG), wenn man die Kanten vom Elternknoten zu den Kindknoten als gerichtet betrachtet. 0P
- [ ] In einem *Binärbaum* kann ein *Blatt* niemals zugleich *Wurzel* sein.
- [ ] Der *Grad eines Knotens* ist die Anzahl seiner Elternknoten.
- [x] Die *Haldeneigenschaft* kann in einem beliebigen unsortierten Feld der Länge n mit einer Laufzeit in $\mathcal{O}(n \cdot log(n))$ hergestellt werden.

e) Welche Aussagen stimmen für folgende Sortierverfahren?

OP

- [x] *RadixSort* für n Werte mit bis zu k Segmenten hat eine Laufzeit von $\mathcal{O}(n \cdot k)$.
- [ ] Sogenannte *in-situ*-Verfahren können nicht schneller als in $\mathcal{O}(n^2)$ sortieren.
- [x] *Stabile* Sortierverfahren ändern niemals die relative Reihenfolge gleicher Werte.
- [ ] *Interne* Verfahren sortieren stets nach der *natürlichen Ordnung* (Java: `compareTo`), *externe* Verfahren hingegen immer mit Hilfe eines zusätzlichen `Comparators`.

f) *Ein Pfad (Weg, Kantenzug) im gerichteten Graphen $G = (V, E)$ von v nach w ist eine endliche Folge von Knoten $v = v_1, v_2, \ldots, v_n = w$, so dass für $1 \leq i \leq n-1$ gilt: $(v_i, v_{i+1}) \in E$.*

OP

- [ ] Die Länge l des Pfades ist die Anzahl der Knoten, also $l = n$.
- [ ] Gibt es keinen Pfad von v nach w, dann kann v kein *Nachfolger* von w sein.
- [x] Ein Pfad ist *einfach*, genau dann wenn alle Knoten im Pfad paarweise verschieden sind.
- [x] In einem Zyklus können alle Kanten paarweise verschieden sein.

g) Für einen *gerichteten* Graph $G = (V, E)$ gilt:

OP

- [x] Wenn $\forall v, w \in V$ ein einfacher Pfad von v nach w existiert, ist G *stark verbunden.*
- [ ] Ist G ein gerichteter azyklischer Graph (DAG), dann hat G keine Wurzel.
- [ ] DFS kann man nur kaskadenartig rekursiv, aber nicht iterativ implementieren.
- [x] Ein Graph $G' = (V', E')$, mit $V' \subseteq V \wedge E' = \{(v, w) \in E | v, w \in V'\}$, ist ein *induzierter Teilgraph* von G.

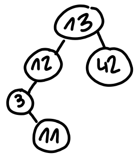
h) Welche Aussagen stimmen für die Programmiersprache Java?

OP

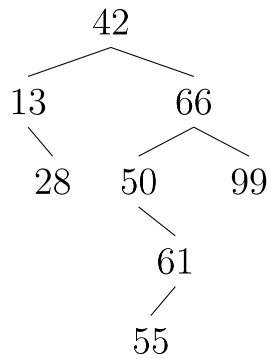
- [ ] In einer statischen Methode bezeichnet `this` die umschließende Klasse.
- [x] Der Konstruktor darf eine `return`-Anweisung enthalten.
- [x] `public void main(String[] args)` ist ohne `static`-Modifizierer keine erlaubte Methodensignatur.
- [ ] Attribute ohne explizite Initialisierung durch den Programmierer werden automatisch mit `null` (bei Objektreferenzen) bzw. `0` (bei primitiven Typen) initialisiert.

Aufgabe 2 (Bäume) (21 Punkte)

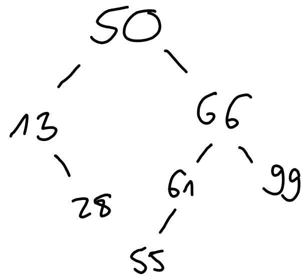
a) Fügen Sie die Zahlen **13, 12, 42, 3, 11** in der gegebenen Reihenfolge in einen *binären Suchbaum* mit aufsteigender Sortierung ein. Stellen Sie nur das Endergebnis dar:



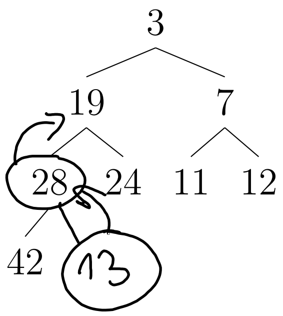
b) Löschen Sie den Wurzelknoten mit Wert **42** aus dem folgenden *binären Suchbaum* mit aufsteigender Sortierung und ersetzen Sie ihn dabei durch einen geeigneten Wert aus dem **rechten** Teilbaum. Lassen Sie möglichst viele Teilbäume unverändert und erhalten Sie die Suchbaumeigenschaft.



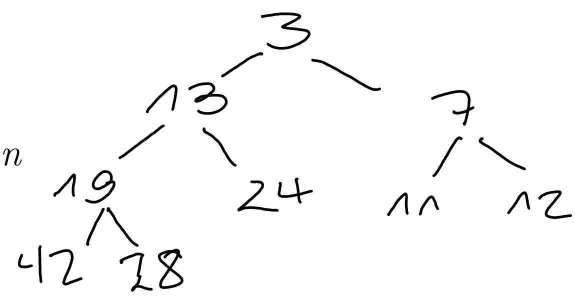
„42“ *löschen*
$\Longrightarrow$
Ersatz aus
rechtem Teilbaum



c) Fügen Sie einen neuen Knoten mit dem Wert **13** in die folgende *Min-Halde* ein und stellen Sie anschließend die Halden-Eigenschaft vom neuen Blatt aus beginnend wieder her, wobei möglichst viele Knoten der Halde unverändert bleiben. Geben Sie nur das Endergebnis an:



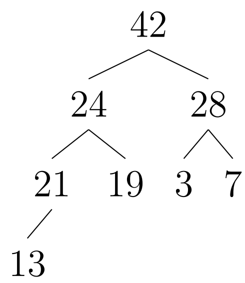
„13“ *einfügen*
$\Longrightarrow$



d) Geben Sie für die *ursprüngliche Min-Halde* aus Teilaufgabe c) (linker Baum, d.h. **ohne** den neu eingefügten Knoten 13) die Feld-Einbettung an:

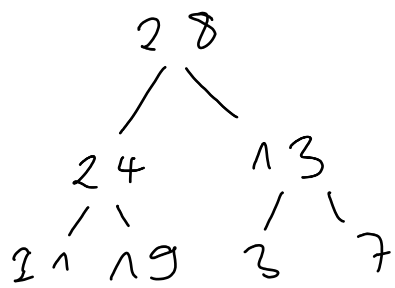
3	19	7	28	24	11	12	42

e) Löschen Sie den Wurzelknoten mit Wert **42** aus der folgenden *Max-Halde* und stellen Sie anschließend die Halden-Eigenschaft ausgehend von einer neuen Wurzel wieder her, wobei möglichst viele Knoten der Halde unverändert bleiben. Geben Sie nur das Endergebnis an:



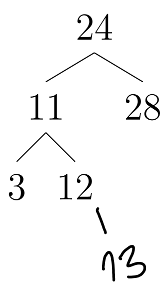
„42“ *löschen*

$\Longrightarrow$



f) Fügen Sie in jeden der folgenden *AVL-Bäume* mit aufsteigender Sortierung jeweils einen neuen Knoten mit dem Wert **13** ein (samt Kante direkt in den **gegebenen** Baum einzeichnen!) und führen Sie anschließend *bei Bedarf* die erforderliche(n) Rotation(en) durch:

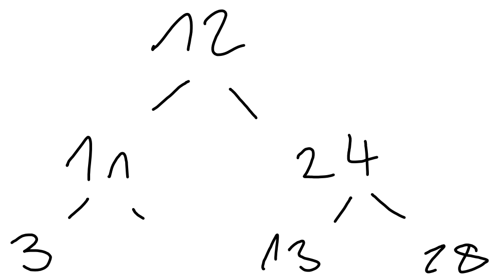
i) AVL-Baum $\mathcal{A}$:



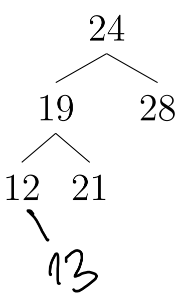
rotieren

$\Longrightarrow$

?



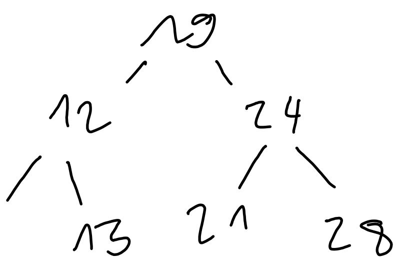
ii) AVL-Baum $\mathcal{B}$:



rotieren

$\Longrightarrow$

?

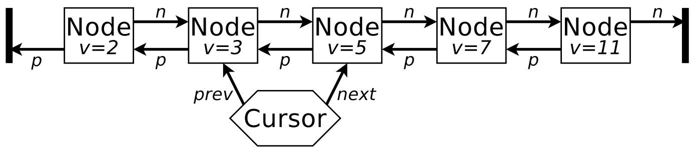


Aufgabe 3 (Listen) (15 Punkte)

Gemäß der Java-API ist ein `ListIterator<E>`

> *an iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, [...]. A* `ListIterator` *has no current element; its* cursor position *always lies between the element that would be returned by a call to* `previous()` *and the element that would be returned by a call to* `next()`*.*

Im Folgenden sollen Sie einen vereinfachten `ListIterator<E>` namens `Cursor<E>` ergänzen, mit dem man eine doppelt-verkettete Liste von `Node`-Objekten verarbeiten kann. Eine Instanz der Klasse `Cursor` verwaltet die aktuelle Schreib-/Lese-Position. Anfangs ist die Liste leer, d.h. beide `Cursor`-Referenzen `prev` und `next` sind `null`.




```java
public class Cursor<E> implements ListIterator<E> {
    private class Node {
        private E v; // value (payload)
        private Node p, n; // previous, next
    }

    private Node prev, next;
```

a) Ergänzen Sie die Methode `add(e)`, die den Wert `e` *zwischen* den beiden Knoten einfügt, auf die der `Cursor` aktuell verweist – `newN` wird dabei zum `next`-Knoten des `Cursors`.

```java
public void add(E e) {
    Node newN = new Node(); // new node becomes next
    newN.v = e;
```

```java
newN.n = next;      // Hier wäre es egal wenn man
newN.p = prev;      // null zuweisen würde

if (prev != null)
    prev.n = newN;
if (next != null)
    next.p = newN;

//Cursor vor neues Element stellen
next = newN;
```

```java
}
```

b) Wenn der `Cursor` einen `prev`-Knoten hat, dann soll `previous()` dessen Wert zurückgeben und den `Cursor` um einen Knoten *nach links bewegen* – andernfalls gibt die Methode `null` zurück. Die Listenstruktur selbst bleibt unverändert.

```
public E previous() {
if(prev != null) {
  E value = prev.v;
  next = prev;
  prev = prev.p;
  return value;
}
return null;
}
```

c) Ergänzen Sie nun die Methode `remove()`, die den `next`-Knoten des `Cursors` aus der Liste entfernt, sofern dieser existiert – die `prev`-Referenz bleibt dabei unverändert:

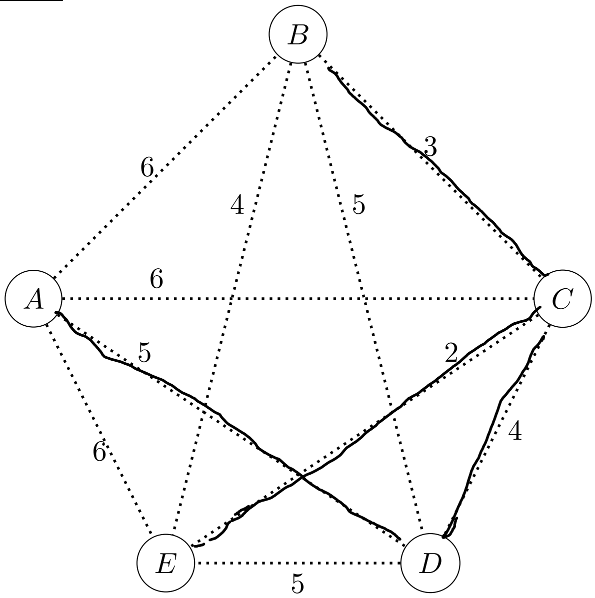
Lsg der FSI ist schöner

```
public void remove() {
  if(next != null) {
    if(next.n != null) {
      if(prev != null) {
        prev.n = next.n;
        if(next.n != null)
          next.n.p = prev;
      }
    }
  next = next.n;
}
```

Aufgabe 4 (Graphen) (21 Punkte)

a) Wenden Sie den Algorithmus von *Kruskal* auf den Graphen $\mathcal{F}$ an und geben Sie die Kanten in der Reihenfolge an, in welcher das Verfahren sie *in den Spannbaum* *aufnimmt*:

Graph $\mathcal{F}$:



Was Sie in den Graphen zeichnen wird ***nicht*** *bewertet!*

(E, C)
(C, B)
(C, D)
(A, D)
(___ , ___)
(___ , ___)
(___ , ___)
(___ , ___)
(___ , ___)
(___ , ___)

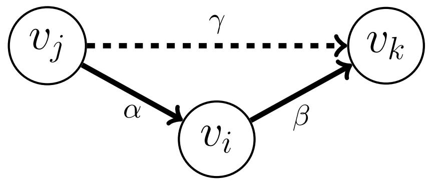
b) Für einen beliebigen Graphen $\mathcal{G} = (V, E)$ mit Kantenanzahl $e := |E|$ und Knotenanzahl $v := |V|$, welche Laufzeitkomplexitäten haben effiziente Implementierungen der Algorithmen von *Prim* und *Kruskal* im allgemeinen Fall?

Prim: $\mathcal{O}(e + v \cdot \log(v))$

Kruskal: $\mathcal{O}(e \cdot \log(e))$

c) Ergänzen Sie den Algorithmus von *Floyd* für Graphen g. Die Adjazenzmatrix `g[j][k]` enthält das positive Gewicht der Kante (j, k) bzw. 0, falls es keine solche Kante gibt oder falls $j = k$. Die Implementierung arbeitet hier *in-situ* und ergänzt auch indirekte Kanten direkt in g.

	A	B	C	D
A	0	X		
B		0		
C			0	
D	0			0




```
// betrachte alle Kanten-Tripel vj => vi => vk,
// vergleiche diese Weglaenge mit vj => vk (sofern vorhanden)
// und aktualisiere ggf. die Weglaenge vj => vk
public void floydify(double[][] g) {
    int n = g.length;
    // durchlaufe alle Knoten vi und vj:
    for (int i = 0; i < n; i++) {
```

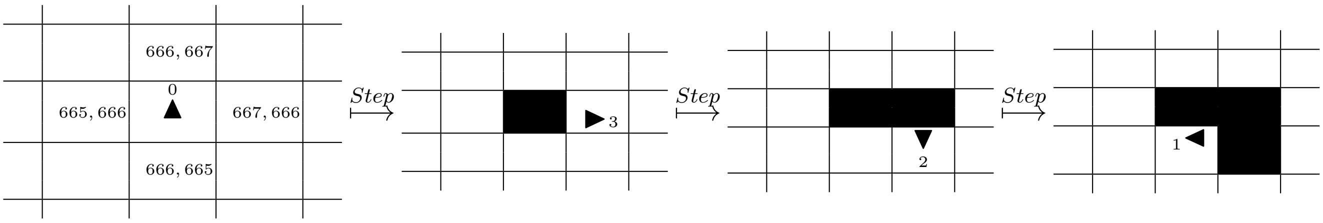
```
for(int j=0; j<n; j++)
  for(int k=0; k<n; k++)
    if(i!=j && i!=k && k!=j)
      if(g[j][i]>0 && g[i][k]>0)
        if(g[j][k]==0 || g[j][i]+g[i][k]<g[j][k])
          g[j][k]=g[j][i]+g[i][k];
```

```
    }
}
```

Aufgabe 5 (ADT) (22 Punkte)

Langtons Ameise (in der Abbildung anfangs mit Blickrichtung Norden als ▲ dargestellt) krabbelt in einem unendlichen Raster aus weißen (*false*) oder schwarzen (*true*) Feldern. Anfangs sind alle Felder weiß und die Ameise startet vom Feld $(666, 666)$ in Richtung d $(0/1/2/3 \widehat{=}$ Norden/Westen/Süden/Osten). In jedem Schritt (*Step*) macht die Ameise folgendes:

1) Auf einem weißen Feld drehe 90 Grad nach rechts, d.h. $d \mapsto (d+3)\%4$ bzw. auf einem schwarzen Feld drehe 90 Grad nach links, d.h. $d \mapsto (d+1)\%4$.
2) Wechsle die Farbe des Feldes (weiß nach schwarz oder schwarz nach weiß).
3) Schreite ein Feld in der aktuellen Blickrichtung fort.



Modellieren Sie nun schrittweise *Langtons Ameise* als abstrakten Datentyp *LA*. Dazu stehen Ihnen zusätzlich zum ADT nur die Datentypen *int* und *boolean* wie in Java zur Verfügung:

---

adt *LA*
sorts *LA, boolean, int*
ops

New:	*int*	$\mapsto$ *LA*	// *startet Ameise auf neuem Raster in übergebener Richtung d*
Step:	*LA*	$\mapsto$ *LA*	// *Ameise macht einen Schritt nach obigen Regeln*
getCol:	*LA* $\times$ *int* $\times$ *int*	$\mapsto$ *boolean*	// *liefert die aktuelle Farbe im übergebenen Feld* (x, y)
getDir:	*LA*	$\mapsto$ *int*	// *liefert die aktuelle Ausrichtung d der Ameise*
getX:	*LA*	$\mapsto$ *int*	// *liefert die aktuelle x-Koordinate der Ameise*
getY:	*LA*	$\mapsto$ *int*	// *liefert die aktuelle y-Koordinate der Ameise*

axs
... // *im Folgenden zu ergänzen*
end *LA*

---

a) Ergänzen Sie den ADT *LA* um Axiome für die Operation `getCol`:

getCol(New(d), x, y) = false

getCol(Step(l), x, y) = ________________________________

falls ________________________________

getCol(Step(l), x, y) = ________________________________ *sonst*

b) Ergänzen Sie den ADT LA um Axiome für die Operation `getDir`:

$getDir(New(d)) = d$

$getDir(Step(l)) =$ ____

falls ____

$getDir(Step(l)) =$ ____________________________________ *sonst*

c) Ergänzen Sie den ADT LA um Axiome für die Operation `getX`:

$getX(New(d)) = 666$

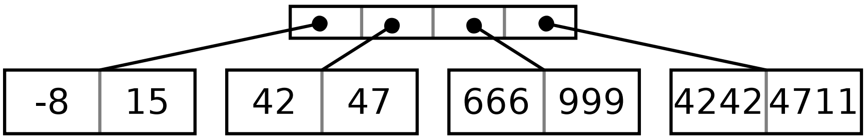
$$getX(Step(l)) = \begin{cases} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ & falls \quad \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ & falls \quad \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ & sonst \end{cases}$$

d) Folgende Klasse stellt eine zustandsbehaftete Implementierung des ADTs *LA* in Java dar. Anstatt eines unendlichen Rasters wird hier ein zweidimensionales Feld `raster` aus `boolean` verwendet, bei dem `raster[x][y] = false` einem weißen Feld entspricht. Sie können davon ausgehen, dass `raster` groß genug dimensioniert ist, so dass beliebig häufige Anwendungen von `step` keine `ArrayIndexOutOfBoundsException`-Probleme beim Rasterzugriff verursachen. Ergänzen Sie die Methode `step` entsprechend:

```
public class LangtonAntJava {
    boolean[][] raster = new boolean[4711][4242];
    int x = 666, y = 666, d = 0; // Position und Richtung

    void step() {





    }
}
```

Aufgabe 6 (Binäre Suche) (10 Punkte)

Ergänzen Sie die Methode `sucheIntervall`, die mittels *iterativer Binärer Suche* im Feld `is` ein `Intervall` sucht, in dem der übergebene Wert `v` liegt. Wenn es kein solches Intervall gibt, dann muss die Methode `null` zurückgeben. Die Intervalle im Feld sind garantiert überschneidungsfrei und aufsteigend sortiert.
Beispiel: Aufgerufen mit dem folgenden Feld gibt die Methode für $v = 747$ das Intervall $(666, 999)$ sowie für $v = 69$ lediglich `null` zurück.




```
class Intervall {
    int min, max;
}

class BinaereIntervallSuche {
    Intervall sucheIntervall(Intervall[] is, int v) {
        int a = 0, m, e = is.length - 1; // Anfang, Mitte, Ende
```

```
        return null;
    }
}
```

Aufgabe 7 (Backtracking) (15 Punkte)

Die Methode `tauziehen` soll die n Werte aus der Liste `in` in beliebiger Reihenfolge so auf die beiden Ergebnislisten `y` und `z` aufteilen, dass `y` und `z` gleich viele Werte haben (bei ungeradem n darf eine der beiden Listen um einen Wert länger sein) und die Summe der Zahlen in `y` so wenig wie möglich von der Summe der Zahlen in `z` abweicht. Ergänzen Sie die *kaskadenartig rekursive* Hilfsfunktion `helfer` geeignet. *Beispiel:*

in = {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4} $(n = 11)$
⇝ y = {45, -34, 12, 98, -1} $(n = 5, \Sigma_y = 120)$
⇝ z = {23, 0, -99, 4, 189, 4} $(n = 6, \Sigma_z = 121)$

```
public class Tauziehen {
    private static List<Long> a, b, y, z;
    private static long abDiff, yzDiff;

    public static List<List<Long>> tauziehen(List<Long> in) {
        a = new LinkedList<>(in); // erste Arbeitsliste (Kopie von in)
        b = new LinkedList<>(); // zweite Arbeitsliste (anfangs leer)
        y = null; // erste Ergebnisliste
        z = null; // zweite Ergebnisliste
        abDiff = 0; // Differenz der Summe der Werte in a bzw. b
        yzDiff = 0; // Differenz der Summe der Werte in y bzw. z
        for (Long v : a) {
            abDiff += v;
        }
        helfer(0);
        return Arrays.asList(y, z);
    }

    private static void helfer(int p) { // p durchlaeuft alle Positionen in a
        // Basisfall a und b (fast) gleich gross erreicht?
        if (Math.abs(a.size() - b.size()) <= 1) {
            if (y == null && z == null // erste Loesung benoetigt
```

```
                // oder bessere Loesung gefunden

                ||

            // merke beste bisher gefundene Loesung:
            y = new LinkedList<>(a);

            z =

            yzDiff =
          }
     } else if(                                         ) { //kein Abbruch
         // Rekursion 1: p-ter Wert bleibt in a:


         // Rekursion 2: p-ten Wert von a nach b verschieben:




         // Backtracking zur 2. Rekursion:
```