OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG
INF FAKULTÄT FÜR INFORMATIK

Institut für Simulation und Graphik
Dr. Christian Rössl

Magdeburg, 31. Juli 2012

Klausur zur Vorlesung „Algorithmen und Datenstrukturen“

Name, Vorname		Studiengang	Matrikelnummer
Zusatzbätter	Unterschriften unbenoteter Schein ☐	Student/in	Aufsicht

Tabelle bitte *nicht ausfüllen*!

	Aufgabe	max. Punkte	erreicht
1	Listen	4	
2	Binärbäume	6	
3	Binäre Suchbäume	4	
4	AVL-Bäume	5	
5	Heaps	5	
6	Hashverfahren	5	
7	Graphen: Datenstrukturen	3	
8	Graphen: Adjazenzmatrizen	5	
9	Topologisches Sortieren	4	
10	Dijkstra-Algorithmus: Beispiel	4	
11	Dijkstra-Algorithmus: Implementierung	7	
12	Dynamische Programmierung	3	
13	Wahr oder falsch?	5	
A	*Bonus*: Dynamische Programmierung	(+2)	
		60 (+2)	

Bearbeitungszeit: 120 Minuten

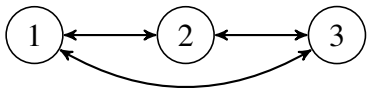
Hinweise zur Klausurbearbeitung

- Überprüfen Sie die Klausur auf Vollständigkeit (15 nummerierte Blätter).
- Füllen Sie das Deckblatt aus!
- Beschriften Sie **alle** Blätter (Klausur, verteiltes Papier) mit Ihrem Namen.
- Legen Sie bitte alle für die Klausur benötigten Dinge, Stifte, Verpflegung und insbesondere Lichtbildausweis, auf Ihren Tisch. Schalten Sie Ihr **Mobiltelefon aus**!
- Hilfsmittel (Taschenrechner, Bücher, Skripten, Mobiltelefone, . . .) sind **nicht** zugelassen!
- Täuschungsversuche führen zum Nichtbestehen der Klausur!
- Verwenden Sie für Ihre Antworten den freien Platz nach den Aufgaben und ggf. die Rückseiten der Blätter. – Melden Sie sich, wenn Sie zusätzliche leere Blätter benötigen.
- **Schreiben Sie deutlich!** Unleserliche Passagen können nicht korrigiert werden.
- Benutzung von Bleistiften sowie roter und grüner Stiftfarbe ist untersagt.
- Beschränken Sie sich auf die geforderten Angaben und halten Sie Ihre Antworten kurz und präzise. Nicht geforderte Angaben ergeben keine zusätzlichen Punkte.
- Geben Sie bei Rechenaufgaben vor jedem Schritt an, was Sie gerade berechnen. Bewertet wird der Rechenvorgang, das Endergebnis allein genügt nicht.

Viel Erfolg!

Aufgabe 1: Listen (4 Punkte)

Die folgende Java-Klasse `RNode` implementiert einen Knoten in einer doppelt verketteten Liste, die zu einem Ring geschlossen ist. Das heißt, der „letzte“ Eintrag ist wieder mit dem „ersten“ verkettet und umgekehrt (siehe Skizze).



Beispiel für einen Ring mit drei Einträgen.

```
1 public class RNode {
2   int  data  = Integer.MAX_VALUE;
3   RNode prev = null;                  // previous node
4   RNode next = null;                  // next node
5   public RNode() {}
6 }
```

Implementieren Sie die folgenden Java-Funktionen!

(a) Einfügen eines neuen Knotens `n` *vor* dem bestehenden Knoten `position` durch

public static void `insert_before(RNode n,RNode position)`.

Gehen Sie dabei davon aus, dass sowohl `position` als auch `n` nicht **null** sind.

(b) Suche nach dem ersten Knoten mit Wert `data==x` im Ring `ring` durch

public static `RNode find(RNode ring,`**int** `x)`.

Falls kein solcher Knoten existiert, soll **null** zurückgegeben werden.

Aufgabe 2: Binärbäume (6 Punkte)

Die Klasse `BinaryTree` (siehe *Zusatzblatt*) beschreibt Knoten eines Binärbaums und damit auch Teilbäume von Binärbäumen. Jeder Knoten speichert eine ganze Zahl (`getData()`).

Erweitern Sie die Klasse um eine Methode

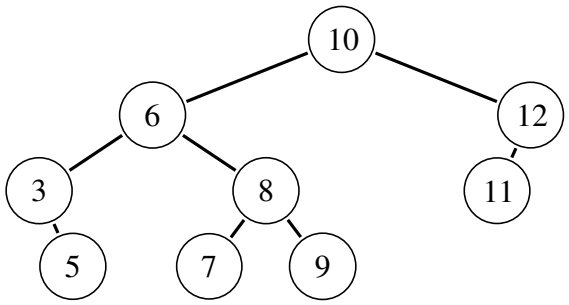
public int `maxLevelSum()` ,

die für jede Ebene des Baums die Summe der Zahlen `getData()` berechnet und das Maximum über alle Ebenen als Ergebnis liefert.

Sie dürfen, wenn nötig, die auf dem *Zusatzblatt* angegeben Datenstrukturen – und *ausschließlich diese* – verwenden.

Aufgabe 3: Binäre Suchbäume (4 Punkte)

Gegeben ist der folgende binäre Suchbaum.



(a) Löschen Sie danach die Knoten 5, 12 und 6 (in dieser Reihenfolge) und skizzieren Sie jeweils den verbleibenden Baum. Beschreiben Sie an diesem Beispiel *kurz*, welche Fälle beim Löschen auftreten und wie diese behandelt werden.

(b) Nehmen Sie an, Knoten in einem binären Suchbaum sind als Klasse `BinaryTree` (siehe *Zusatzblatt*) repräsentiert. Erweitern Sie diese Klasse um die Methode

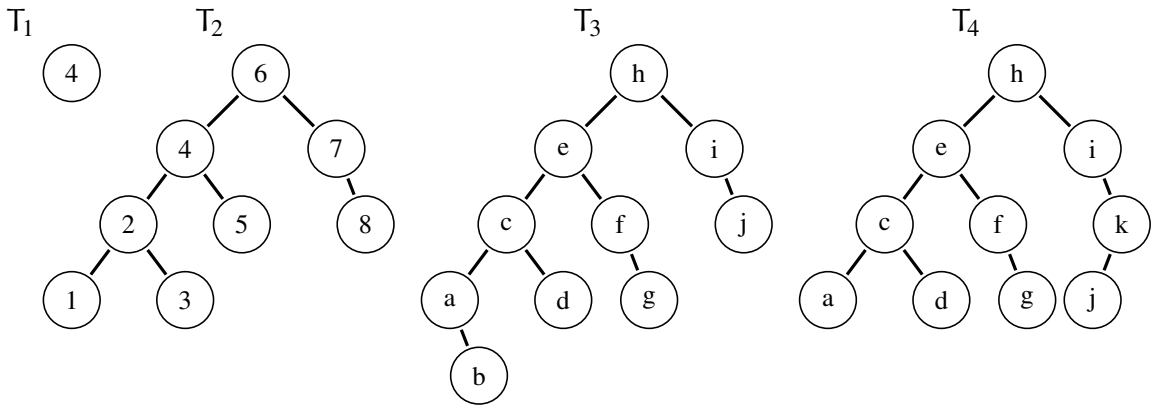
public `BinaryTree find(`**int** `x)` ,

die den Wert `x` im aktuellen Baum **this** sucht. Bei erfolgloser Suche soll **null** zurückgegeben werden.

Aufgabe 4: AVL-Bäume (5 Punkte)

(a) Welche Eigenschaften zeichnen einen AVL-Baum aus? Welchen Vorteil hat ein AVL-Baum gegenüber einem einfachen binären Suchbaum?

(b) Welche der vier Bäume T_1, T_2, T_3 und T_4 sind AVL-Bäume, welche nicht? Begründen Sie Ihre Antwort kurz.



(c) Fügen Sie die Zahlen $2, 6, 7, 4, 3, 5, 1$ (in dieser Reihenfolge) in einen zunächst leeren AVL-Baum ein. Skizzieren Sie die Bäume jeweils nach dem Einfügen eines Eintrags *und* nach dem Herstellen der AVL-Eigenschaft falls nötig. Kennzeichnen Sie im letzten Fall die Verletzung der AVL-Eigenschaft.

Aufgabe 5: Heaps (5 Punkte)

(a) Erstellen Sie einen binären Min-Heap für die Zahlen

6	5	2	3	1	4

(in dieser Reihenfolge) im *top-down* Verfahren!

Skizzieren Sie dazu jeweils binäre Bäume nach Einfügen eines Eintrags und Herstellen der Heap-Eigenschaft. (Falls Sie weitere Zwischenschritte angeben, kennzeichnen Sie diese bitte.)

(b) Erstellen Sie den gleichen Min-Heap wie in (a) aber im *bottom-up* Verfahren. Skizzieren Sie dazu die binären Bäume beginnend mit der angegebenen Tabelle nach *jedem* Vertauschen von zwei Einträgen.

(c) Wie erhält man das linke und das rechte Kind im Knoten zum i-ten Tabelleneintrag? Wie erhält man den Elternknoten zum i-ten Tabelleneintrag? Es gilt jeweils $1 \leqslant i \leqslant n$.

(d) Nennen Sie zwei Anwendungen von Heaps aus der Vorlesung.

Aufgabe 6: Hashverfahren (5 Punkte)

Es sollen ganze Zahlen x in eine Hashtabelle der Größe $m = 11$ eingefügt werden. Dabei wird die Hashfunktion

$$h(x) = \big(3x+7\big) \bmod 11$$

verwendet. Gegeben ist weiterhin die Funktion

$$h_2(x) = ((2x+1) \bmod 13)+1\ .$$

Die folgende Tabelle gibt eine Reihe von Werten x sowie von $h(x)$ und $h_2(x)$ an.

x	0	1	2	3	5	8	13	21
$h(x)$	7	10	2	5	0	9	2	4
$h_2(x)$	2	4	6	8	12	5	2	5

Für die beiden folgenden Teilaufgaben (a) und (b) fügen Sie bitte die Werte x in der angegeben, aufsteigenden Reihenfolge in die Hashtabelle `a[]` ein und verwenden Sie jeweils die angegebene Methode zur Kollisionsbehandlung. Geben Sie zusätzlich für den eingefügten Wert die Anzahl der erfolgten Kollisionen C – das heißt die maximale Anzahl Sondierungen i – an.

(a) Einfügen und Kollisonsbehandlung durch *lineares Sondieren* mit

$$\big(h(x)+i\big) \bmod 11$$

`k`	0	1	2	3	4	5	6	7	8	9	10
`a[k]`											
C											

(b) Einfügen und Kollisonsbehandlung durch *double hashing* mit

$$\big(h(x)+i\cdot h_2(x)\big) \bmod 11$$

`k`	0	1	2	3	4	5	6	7	8	9	10
`a[k]`											
C											

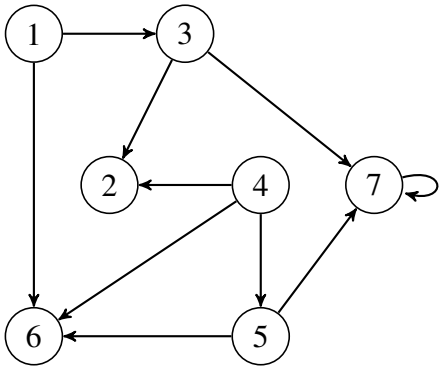
(c) Nennen Sie zwei weitere Methoden zur Kollisionsbehandlung.

(d) Erläutern Sie für *eine beliebige* der in (a) – (c) genannten Methoden zur Behandlung von Kollisionen *kurz*, wie Einträge gelöscht werden.

(e) Welchen asymptotischen Aufwand (O-Notation) erwartet man typischerweise beim Suchen in einer (gut implementierten) Hashtabelle?

(f) Die Suche in einer Menge, z.B. für den ADT `Set`, kann sowohl mit balancierten Suchbäumen als auch mit Hashtabellen implementiert werden. Nennen Sie jeweils einen Vor- und einen Nachteil der Hashtabelle.

Aufgabe 7: Datenstrukturen für Graphen (3 Punkte)



(a) Welche der folgenden Darstellungen zeigt den abgebildeten gerichteten Graphen?

(1) Kantenliste

```
int[] edgelist = {7,10, 1,3,1,6,3,2,3,7,4,2,4,5,4,6,5,6,5,7,7,7};
```

(2) Knotenliste in zwei Tabellen

```
int[] neighborhoods = {6,3,7,2,2,5,6,6,7,7};
int[] nodeNhd = {0,2,2,4,7,9,9,10}; // last entry = number of edges
```

(3) Adjazenzmatrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(4) Adjazenzliste

1	3	6	
2			
3	2	7	
4	2	5	6
5	6	7	
6			
7	7		

(b) Welche Darstellung eignet sich Ihrer Meinung nach am besten für eine Anwendung, in der ein Graph oft gezeichnet werden muss (z.B. Kartendarstellung im Navigationssystem)? Begründen Sie Ihre Antwort kurz.

Aufgabe 8: Datenstrukturen für Graphen: Adjazenzmatrizen (5 Punkte)

Gegeben ist die folgende Adjazenzmatrix eines gerichteten Graphen mit Knoten $1, 2, \ldots, 10$.

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

(a) Welche Knoten können von Knoten 3 aus über eine Kante erreicht werden?

(b) Von wie vielen Knoten aus kann der Knoten 7 über eine Kante erreicht werden?

(c) Gibt es in dem Graphen eine Schleife, das heißt eine Kante von einem Knoten zu sich selbst? Wenn ja, für welche(n) Knoten?

(d) Die Klasse `aud.graph.GraphAM` aus den Beispielen zur Vorlesung verwendet eine Darstellung von gerichteten Graphen, die grob wie folgt implementiert ist:
Kanten werden als Paare (i,j) in einem Rot-Schwarz-Baum gespeichert, dabei gilt eine lexikographische Ordnung so dass

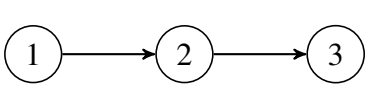
$$(i,j) < (k,l) \;\Leftrightarrow\; (i < k) \vee (i = k \wedge j < l)\,.$$

Sei n die Anzahl Knoten, und sei m die Anzahl Kanten des Graphen.

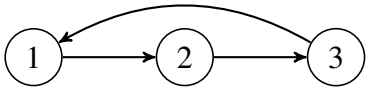
(1) Welcher asymptotische Aufwand wird benötigt, um eine neue Kante einzufügen?

(2) Erläutern Sie *kurz*, wie alle von einem Knoten i *ausgehenden* Kanten effizient aufgezählt werden können. Wie hoch ist der Aufwand dafür (O-Notation), wenn es für beliebige Knoten i maximal M von i ausgehende Kanten gibt?

(e) Gegeben sind die beiden folgenden Graphen G_1 und G_2, die zugehörigen Adjazenzmatrizen seien mit $\mathbf{A}_1$ und $\mathbf{A}_2$ bezeichnet.



G_1

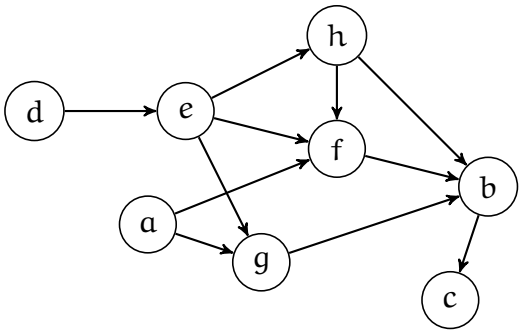


G_2

Geben Sie die Matrixpotenzen $\mathbf{A}_1^{10}$ und $\mathbf{A}_2^{10}$ an. (*Hinweis*: Sie müssen dazu *nicht* rechnen!)

Aufgabe 9: Topologisches Sortieren (4 Punkte)

(a) Geben Sie eine topologische Sortierung des folgenden Graphen an.



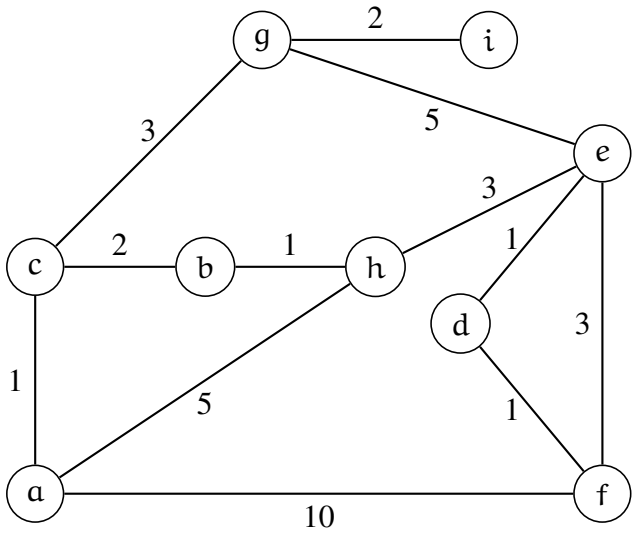
(b) Zeigen Sie an einem gerichteten Graphen mit *drei* Knoten, dass die topologische Sortierung nicht eindeutig ist.

(c) Geben Sie einen gerichteten Graphen mit *zwei* Knoten an, der *nicht* topologisch sortiert werden kann.

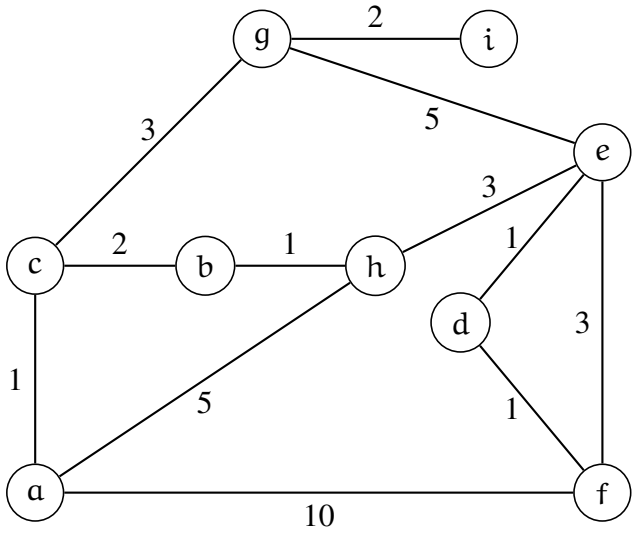
Aufgabe 10: Kürzeste Pfade und minimaler Spannbaum (4 Punkte)

(a) Wenden Sie den Algorithmus von Dijkstra für den folgenden ungerichteten Graphen an, ausgehend vom Knoten $s_0 = a$.

Notieren Sie dazu an jedem Knoten die Distanz zum Ausgangsknoten s_0 und markieren Sie die Kante, über die der Knoten im kürzesten Pfad erreicht wird (d.h. den *shortest path tree*). Geben Sie zusätzlich die Reihenfolge an, in der die Knoten vollständig abgearbeitet werden. (Es ist keine weitere Angabe von Zwischenergebnissen nötig.)



(b) Geben Sie zum selben Graphen einen minimalen Spannbaum (*minimum spanning tree*/MST) an! Markieren sie dazu die Kanten des MST im Graphen (Zeichnung unten) und geben Sie die Summe der Kantengewichte des MST an.



Aufgabe 11: Algorithmus von Dijkstra (7 Punkte)

(a) Der Algorithmus von Dijkstra bestimmt alle kürzesten Wege in einem Graphen `g` ausgehend von einem Knoten `s0`. Die folgende *unvollständige* Java-Implementierung des Algorithmus berechnet für alle Knoten eines gerichteten Graphen `g` die Attribute `Node.d` (Distanz zu `s0`) und `Node.parent` (Vorgänger im Pfad). Vervollständigen Sie die Implementierung! Sie dürfen dazu das vorgegebene Gerüst beliebig *erweitern*. Kennzeichnen Sie sorgfältig, an welcher Stelle Sie was einfügen.

Hinweise: (1) Die verwendeten Klassen sind auf dem *Zusatzblatt* angegeben.
(2) Die Klasse `PriorityQueue<Node>` soll Attribute `Node.d` vergleichen.
(3) Verwenden Sie wenn nötig `Double.isInfinite(x)`.
(4) Sie können `Node` um Attribute erweitern. (Das ist nicht unbedingt nötig.) Achten Sie in diesem Fall auf korrekte Initialisierung.

```
1  public static void dijkstra(Graph g,Node s0) {
2   for (Node node : g) {
3     node.d=Double.POSITIVE_INFINITY; // distance
4     node.p=null;                      // parent
5   }
6   s0.d=0.0;
7   PriorityQueue<Node> open=new PriorityQueue<Node>();
8   open.push(s0);                    //  only s0 in PQ
9
10  while (!open.is_empty()) {
11    Node s=open.pop();
12    for (Edge e : g.getOutEdges(s)) {
13      Node t=e.destination();
14
15      // HIER FEHLT ETWAS !!
16    }
17 }
```

(b) Implementieren Sie die Java-Funktion

public static void `print_path(Node t)`,

die *nach* Aufruf von `dijkstra(g,s0)` den Pfad *von* `s0` *nach* `t` ausgibt. Die Ausgabe der Knoten `node` im Pfad soll dabei als `System.out.println(node)` erfolgen.

(c) Wie müssen Sie die Funktion(`dijkstra()`) modifizieren, um einen minimalen Spannbaum (*minimum spanning tree*/MST) zu konstruieren? (Algorithmus von Prim)

Hinweis: Verwenden Sie zur Lösung der Programmieraufgaben die Datenstrukturen vom *Zusatzblatt* – und *ausschließlich diese* – je nach Bedarf.

Aufgabe 12: Dynamische Programmierung (3 Punkte)

(a) Für welche der folgenden rekursiv definierten Funktionen ist es möglich *und effizienter* (bezüglich des asymptotischen Aufwands), dynamische Programmierung einzusetzen? Es ist jeweils keine Begründung erforderlich.

(1)
```
static int a(int n) { return n==0 ? 1 : n*a(n-1); }
```

(2)
```
static int b(int n,int a) { return n==0 ? 1 : b(n-1,a*n); }
```

(3)
```
static int c(int n) { return n<2 ? 1 : c(n-1)+c(n-2); }
```

(4)
```
static double d(double t,int i,int j,double[] b) {
  if (i==0) return b[j];
  else return (1.0-t)*d(t,i-1,j,b)+t*d(t,i-1,j+1,b);
}
```

(5)
```
static Node e(int x,Node node)  {
  if (node==null || node.data==x) return node;
  else if (node.data<x) return e(x,node.getLeft()); }
  else return e(x,node.getRight());
}
```

(6)
```
static int f(int n,int k) {
  if (k==0 || k==n) return 1;
  else if (k>n) return 0;
  else return f(n-1,k-1)+f(n-1,k);
}
```

(b) Geben Sie für *eine beliebige* der von Ihnen in (a) benannten Funktionen eine iterative Implementierung (in Java) unter Verwendung von dynamischer Programmierung an.

Aufgabe 13: Wahr oder falsch? (5 Punkte)

Kreuzen Sie jeweils (*ohne Begründung*!) an, ob eine Aussage wahr oder falsch ist.

Für jede richtige Antwort gibt es einen halben Punkt, für jede falsche Antwort wird ein halber Punkt abgezogen. (*Es gibt wenigstens 0 Punkte auf die gesamte Aufgabe.*) Beantworten Sie also lieber nur Fragen, bei denen Sie sich sicher sind!

Aussage	**wahr**	**falsch**
Eine verkettete Liste mit n Einträgen benötigt mehr Speicher als ein Feld (*array*) der Länge n.		
In einem Min-Heap steht der größte Eintrag stets in der untersten Ebene (Baum) am weitesten rechts.		
Hashfunktionen sind stets injektiv.		
Double hashing halbiert die Anzahl der Kollisionen im Mittel.		
Die Größe einer Hashtabelle muss eine Primzahl sein.		
Der Bellmann-Ford Algorithmus zur Berechnung kürzester Wege funktioniert *nicht* für Graphen mit negativen Zyklen.		
Der Wert des maximalen Flusses in einem Graphen ist gleich den Kosten des maximalen Schnitts.		
Die Lösung eines Optimierungsproblems mit Hilfe eines *Greedy*-Algorithmus liefert *nicht* zwangsläufig ein (global) optimales Ergebnis.		
Mit Hilfe des A*-Algorithmus sollen kürzeste Wege bestimmt werden. Dazu wird jedem Knoten t eine Position $(x(t), y(t)) \in \mathbb{R}^2$ zugeordnet. Sei s_1 der Zielknoten. Dann ist $f(t) = \lvert x(t) - x(s_1)\rvert$ eine zulässige Heuristik.		
Das Rucksackproblem kann nur dann mit dynamischer Programmierung gelöst werden, wenn der Gewinn pro Gegenstand ganzzahlig ist.		

Bonusaufgabe A: Dynamische Programmierung (2 Zusatzpunkte)

Gegeben ist eine Folge von reellen Zahlen $a_1, \ldots, a_n$. Gesucht ist eine zusammenhängende Teilfolge $a_i, \ldots, a_j$, für die die Summe $\sum_{k=i}^{j} a_k$ maximiert wird. (*Hinweis*: Negative Einträge $a_i < 0$ sind erlaubt! Sonst wäre das Problem trivial.)

Konstruieren Sie einen Algorithmus, der den *Wert*

$$\max_{i,j} \sum_{k=i}^{j} a_k$$

bestimmt. Verwenden Sie dazu *dynamische Programmierung*:

(a) Geben Sie eine Rekursionsformel für das Ergebnis an.

(b) Implementieren Sie ein effizientes, iteratives Lösungsschema in Java.

(c) Geben Sie den asymptotischen Aufwand Ihres Algorithmus an.

Graph
Sie dürfen Knoten der Klasse `Node` und Kanten der Klasse `Edge` (unten) annehmen.

```
public class Graph implements Iterable<Node> {
  public Graph();
  public Edge getEdge(Node source,Node destination);
  public Vector<Edge> getInEdges(Node node);
  public Vector<Edge> getOutEdges(Node node);
  public Iterator<Edge> edges();
}
```

Node (Knoten in **Graph**)

```
public class Node {
  public String getLabel();
  public String toString();

  public Node   p = null;                          // parent
  public double d = Double.POSITIVE_INFINITY; // distance
}
```

Erweitern Sie die Klasse falls nötig um fehlende Attribute. (Für deren Initialisierung sind Sie selbst verantwortlich.)

Verwenden Sie `Double.isInfinite(x)`, um zu prüfen, ob $x \to \infty$.

Edge (Kante in **Graph**)

```
public class Edge {
  public Node source();
  public Node destination();
  public double getWeight();
}
```

Vector (Sie benötigen nur die Eigenschaft *iterable*.)

```
public class Vector<T> implements Iterable<T> {}
```

Vorgegebene Datenstrukturen

Sie dürfen zur Lösung der Aufgaben die folgenden Datenstrukturen – und *ausschließlich diese* – verwenden. Es handelt sich hierbei um vereinfachte Versionen der in der Vorlesung verwendeten Datenstrukturen.

BinaryTree

```
public class BinaryTree {
  public BinaryTree(int data);
  public BinaryTree getParent();
  public BinaryTree getLeft();
  public BinaryTree getRight();
  public int getData();              // store integer values
}
```

Stack

```
public class Stack<T> {
  public Stack();
  public boolean is_empty();
  public T top();
  public T pop();
  public void push(T x);
}
```

Queue

```
public class Queue<T> {
  public Queue();
  public boolean is_empty();
  public T front();
  public T dequeue();
  public void enqueue(T x);
}
```

PriorityQueue

Sie dürfen annehmen, dass ein Vergleich auf `T` definiert ist, so dass `front()` bzw. `pop()` den *kleinsten* Eintrag liefern.

```
public class PriorityQueue<T> {
  public PriorityQueue();
  public boolean is_empty();
  public boolean contains(T x); // Is x included in PQ?
  public T front();
  public T pop();
  public void push(T x);
  public void lower(T x); // adapt PQ to changed (lowered)
                          //  priority of x
}
```