Der folgende Mitschrieb wurde von Prof. Alexa am 16.07.2008 als Probeklausur in der MPGI2 Vorlesung gezeigt und wurde auf http://www.basicinside.de/2008/node/94 veröffentlicht.

1. Aufgabe (6 Punkte): Java-Programmierung (Arrays)

In einer Jugendherberge gibt es Zimmer für Übernachtungen. Auf jedem Zimmer steht eine bestimme Anzahl von Betten für die Gäste zur Verfügung.

```
public class Gaestezimmer {
    public int betten;
    public int gaeste;
}
```

Das Attribut *betten* gibt an, wie viele Betten in dem Zimmer vorhanden sind. Das Attribut *gaeste* gibt an, wie viele der Betten zur Zeit von Gästen belegt sind.
Die Jugendherberge hat mehrere Etagen, auf jeder Etage gibt es mehrere Zimmer. Die Zimmer sind nach Etage und Nummer in einem *2-dimensionalen Array* erfasst. (1. Dimension: Etage, 2. Dimension: Nummer). Gehen Sie davon aus, dass alle Felder im Array mit Zimmern initialisiert wurden (d.h. sie sind nicht mit *null* belegt).
Erweitern Sie in den folgenden Aufgaben die Klasse *Jugendherberge*.

```
public class Jugendherberge {
    public Gaestezimmer [][] zimmer;
    //... Ihre Methoden
}
```

1.1. Zimmer mit freien Betten (3 Punkte)

Implementieren Sie die Methode *sucheZimmer*, die ein Gästezimmer der Jugendherberge zurückliefert, in dem es noch mindestens eine bestimmte Anzahl von freien Betten gibt. Die Mindest-Anzahl freier Betten soll als Parameter übergeben werden. Wählen Sie einen geeigneten Rückgabewert für den Fall, dass kein Zimmer diese Bedingung erfüllt.

```
public Gaestezimmer sucheZimmer(int freieBetten) {
    for(int etage = 0; etage < zimmer.length; etage++) {
        for(int raum = 0; zimmer[etage].length; raum++) {
            Gaestezimmer g = zimmer[etage][raum];
            if ((g.betten - g.gaeste) >= freieBetten)
                return g;
        }
    }
    return null;
}
```

1.2 Freie Betten auf einer Etage (3 Punkte)

Implementieren Sie die Methode *freieBettenAufEtage*, welche die Anzahl nicht belegter Betten auf einer bestimmten Etage zurückgibt. Die Nummer der Etage wird als Parameter übergeben. Falls die Etage nicht existiert, soll die Methode 0 als Ergebnis liefern.

```
public int freieBettenAufEtage(int etage) {
    int frei = 0;
    if ((etage >= 0) &amp;amp;&amp;amp; (etage < zimmer.length))
        for(int raum = 0; raum < zimmer[etage].length; raum++)
            frei += (zimmer[etage][raum].betten - zimmer[etage][raum].gaeste);
    return frei;
}
```

2. Aufgabe (ohne weitere Angaben)

2.2 Sortieralgorithmen (3 Punkte)

Gegeben sind die Implementationen von den Sortieralgorithmen *A* und *B* durch die gleichgenannten Methoden.

```
public static void A (lang [] a) {
    for(int i = 1; i < a.length; i++) {
        helpA(a, i);
    }
}

public static void helpA (long [] a, int h) {
    for(int i = h; i>=1; i--) {
        if (a[i-1] <= a[i])
            break;
        swap(a, i-1, i);
    }
}

public static void B (long [] a) {
    for(int i = 0; i < a.length;i++) {
        int j = helpB(a,i);
        swap(a,i,j);
    }
}

public static void helpB(long [] a, int h) {
    int result = h;
    for(int i = h+i); i < a.length; i++)
        if (a[i] < [a.result])
            result = i;
    return result;
}
```

Beide Implementationen benutzen die Hilfsmethode *static void swap(long [] a, int i, int j)*, die die Zahlen an Position *i* und *j* im Array *a* vertauscht.

Algorithmen	in-situ (ja/nein)	stabil (ja/nein)	Name des Verfahrens
A	ja	ja	InsertionSort
B	ja	ja	SelectionSort

3. Aufgabe (ohne weitere Angaben)

4. Aufgabe (ohne weitere Angaben)

5. Aufgabe (2 Punkte)

5.1 Hashtabellen (2 Punkte)

Fügen Sie die drei Schlüssel *10*, *15*, *21* in der angegeben Reihenfolge in eine anfangs leere Hashtabelle ein. Die Tabelle hat 5 Speicherplätze. Die folgende Hashfunktion *h* und Sondierungsfunktion *g* sind zu verwenden.

```
  h(k) = k mod 5
g(k,j) = (j + 3) mod 5
```

Hierbei ist *k* der einzufügende Schlüssel und *j* der zuletzt getestete Speicherplatz. Geben Sie für jeden Schlüssel an, welche Positionen getestet werden, wo Kollisionen auftreten und wo der Schlüssel letztendlich gespeichert wird.

```
[-,-,-,-,-]
h(10)=0                    keine Kollision
[10,-,-,-,-]
h(15)=0                    Kollision
k(15,0)=(0+3) mod 5=3      keine Kollision
[10,-,-,15,-]
h(21)=1                    keine Kollision
[10,21,-,15,-]
```

6. Aufgabe (9 Punkte): Java-Programmierung (Listen)

Im Rahmen dieser Aufgabe soll mit einer einfachen Liste gearbeitet werden.
Gegeben sind die Klassen *Rechteck* und *ListElement*

```
public class Rechteck {

    private double a;
    private double b;

    public Rechteck(double a, double b) {
        this.a = a;
        this b = b;
    }
}

public class ListElement {

    private Rechteck rechteck;
    private ListElement naechstes;

    public ListElement(Rechteck rechteck, ListElement naechstes) {
        this.rechteck = rechteck;
        this.naechstes = naechstes;
    }

    public void setRechteck(Rechteck rechteck) {
        this.rechteck = rechteck;
    }

    public Rechteck getRechteck() {
        return this.Rechteck;
    }

    public ListElement getNaechstes() {
        return this.naechstes;
    }

   public void setNaechstes(ListElement naechstes) {
       this.naechstes = naechstes;
    }
}
```

6.1 Listenklasse (2 Punkte)

Implementieren Sie zunächst eine Klasse *RechteckListe*.
Die Klasse repräsentiert eine einfach verkettete Liste mit Objekten der Klasse *ListElement* und soll die Attribute *Listenanfang* und *Länge der Liste* beinhalten. Achten Sie darauf, Kapselung zu verwirklichen.

Ergänzen Sie eine *get-Methode*, für den Zugriff auf die Listenlänge und eine Methode *loeschen*, die alle Elemente aus der Liste entfernt.

```
public class RechteckListe {
    private ListElement first;
    private int length;

    public int getLength() {
        return length;
    }

    public void loeschen() {
       length = 0;
       first = null;
    }
}
```

6.2 Gefilterte Liste (3 Punkte)

Ergänzen Sie in der Klasse *RechteckListe* eine Methode *filter*, die aus der Rechteckliste alle Rechtecke entfernt, deren Fläche kleiner ist als der übergebene Wert *minFlaeche*.
Hinweis: Für die Lösung dieser Aufgabe stehen Ihnen *keine* weiteren als die Methoden aus der vorgegebenen Klasse *ListElement* zur Verfügung.

```
public void filter (double minFlaeche) {
    ListElement current = first;
    ListElement previous = current;

    while(current != null) {
        if ((current.getRechteck().a * current.getRechteck().b) >= minFlaeche) {
            previous = current;
        } else {
            if (current == first)
                first = current.getNaechstes();
            else
                previous.setNaechstes(current.getNaechstes());
            length--;
        }
        current = current.getNaechstes();
    }
}
```

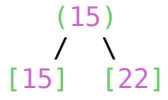
7.Aufgabe (8 Punkte): Suchbäume

7.1 Einfügen in Suchbäume

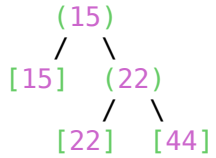
Fügen Sie in einen binären, geordneten Baum (Suchbaum) mit Werten nur in den Blättern die folgenden Werte in gegebener Reihenfolge ein:
15,22,44,43,42
Der Baum ist vor dem Einfügen des Wertes *15* leer. Beim Erzeugen innerer Knoten solll gelten: Der Schlüssel gibt den größten Wert des linken Teilbaums an. Geben Sie den Baum nach jedem eingefügten Wert an.

Einfügen der 15:

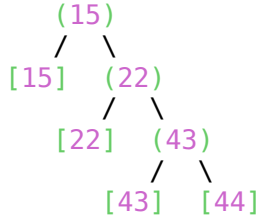
Einfügen der 22:



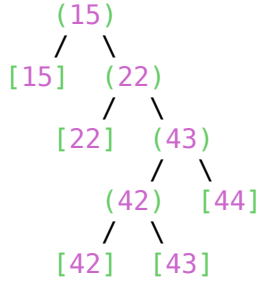
Einfügen der 44:



Einfügen der 43:



Einfügen der 42:



7.2 (ohne weitere Angaben)

7.3 Binäre Suchlisten - Aufbau (2 Punkte)

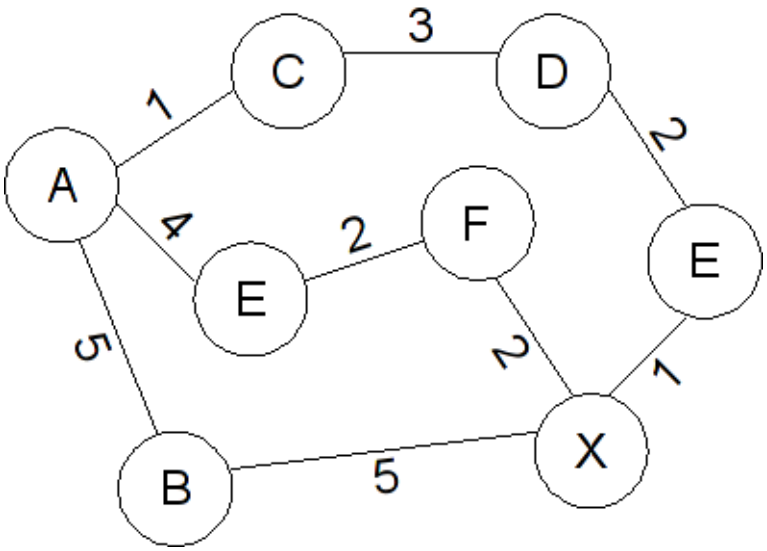
Gegeben ist die folgende Struktur eines binären, geordneten Baums (Suchbaum). Die Werte sind ganze Zahlen und befinden sich nur in den Blättern. In welcher Reihenfolge müssen die vier Elemente *1*, *2*, *3* und *4* in einen leeren Baum eingefügt werden, um diese Struktur zu ergeben? Geben Sie alle geeigneten Einfügereihenfolgen an. Es gilt dieselbe Anforderung beim einfügen wie in Teilaufgabe 1: Der Wert eines inneren Knotens gibt den größten Wert im linken Teilbaum an.

```
Mögliche Reihenfolgen (unvollständig!):
2,3,4,1
2,3,1,4
2,4,1,3
2,4,3,1
etc...
```

8. Aufgabe (7 Punkte): Graphen

8.1 Dijkstra Algorithmus (3 Punkte)

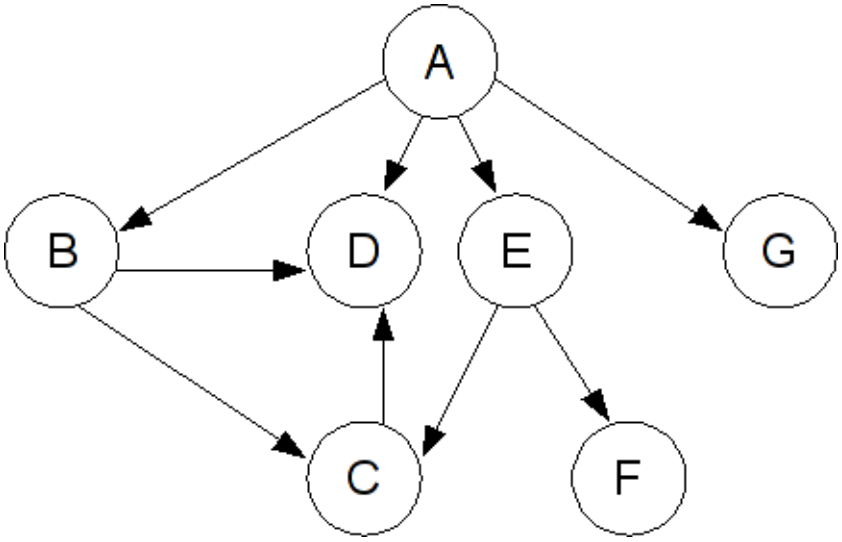
Bestimmen Sie mit Hilfe des Dijkstra-Algorithmus die kürzesten Wege vom Startknoten *A* zu allen anderen Knoten. Verwenden Sie dazu die folgende Tabelle, in der Sie Felder, in denen Relaxation stattfindet, umkreisen und Felder, die die endgültigen Knoten angeben, unterstreichen.
Hinweis: Die gleichen Kosten sei die Knotenordnung so, dass kleinere Buchstaben jeweils vor größeren kommen (alphabetische Ordnung), d.h. z.B. wenn *B* und *C* beide die Kosten *5* haben, dann wählen Sie *B*.



Iteration	A	B	C	D	E	F	G	X
0	0	inf	inf	inf	inf	inf	inf	inf
1	0	(5)	(1)	inf	(4)	inf	inf	inf
2	-	5	1	(4)	4	inf	inf	inf
3		5	-	4	4	inf	(6)	inf
4		5		-	4	(6)	6	inf
5		5			-	6	6	(10)
6		-				6	6	(8)
7						-	6	(7)
8							-	7
								-

8.2 Topologische Sortierung (2 Punkte)

Geben Sie eine topologische Sortierung der Knoten des folgenden Graphen an.



A, G, E, F, B, C, D

8.3 Eindeutigkeit (1 Punkt)

Ist die topologische Sortierung des Graphen aus der letzten Unteraufgabe (8.2) eindeutig? Begründen Sie ihre Antwort.

Nein, denn A, E, F, B, C, D, G ist auch eine Lösung

8.4 Topologische Sortierbarkeit (1 Punkt)

Unter welcher Bedingung ist eine topologische Sortierung nicht möglich?

Wenn der Graph nicht kreisfrei ist, ist die topologische Sortierung nicht möglich.