Algorithmen und Programmierung III WS 04/05
Nachklausur am 4.4.05, 10:15 – 11:45
Zu erreichende Mindestpunktzahl: 20 (von insgesamt 40)

Aufgabe 1 (8 Punkte, 18 min)

Die folgenden Klassen seien vorgegeben:

a) (4 P)

```
class A {
  static int x;
  static void op(int a) {
    System.out.println(x++);
  }
}
class B extends A {
  static int x;
  void op(char y) {
    System.out.println(x-1);
  }
  static void test() {
    B b;
    A a = (b = new B());
    a.x = 6;
    a.op('a');
    b.op('a');
  }
}
```

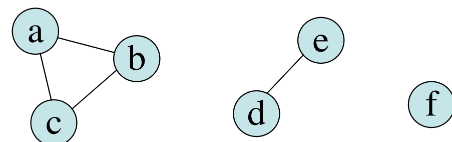
b) (4 P)

```
class C<E> {
  E[] a = new E[8];
  boolean[] b = new boolean[8];
  static class D {
    boolean m(int i) {
        return !b[i];
    }
  }
  void m(byte b) {
    System.out.println(
       (new D()).m(b));
  }
  static void test() {
    C<Float> c = new C<Float>();
    int x = 8;
    c.m(x);
  }
}
```

Entscheiden Sie jeweils, ob es sich hierbei um gültigen Java-(1.5-)Code handelt. Geben Sie an, welche Ausgabe bei Ausführung der Methode `test` erzeugt wird, bzw. erklären Sie alle statischen Fehler.

Aufgabe 2 (14 Punkte, 32 min)

Eine zusammenhängende Teilmenge der Knotenmenge eines Graphen bezeichnet man als *Zusammenhangskomponente*. Der Graph rechts hat z.B. die drei Zusammenhangskomponenten {a,b,c}, {d,e} und {f}:



Ein solcher Graph sei wie folgt als Geflecht in Java repräsentiert (vgl. VL 9.1, S. 17):

```
public class Graph {
    static class Node {
        Set<Node> adj = new TreeSet<Node>(); // Nachbarn
        boolean mark  = false;               // zwecks Markierung
        int size() { ... }
    }
    private Set<Node> nodes;  // Knotenmenge des Graphen
    public Vector<Integer> componentSizes() { ... }
}
```

a) (8 P) Entwickeln Sie die Methode `size` der Klasse `Node`, die ermittelt, wie viele Knoten die Zusammenhangskomponente umfasst, zu der der aktuelle Knoten gehört. Aufgerufen auf dem Knoten e im Beispiel oben, müsste `size` den Wert 2 liefern.

b) (6 P) Entwickeln Sie die Methode `componentSizes`, die als Ergebnis einen `Vector` liefert, der die Größen der einzelnen Zusammenhangskomponenten des Graphen enthält. Im obigen Beispiel könnte sich z.B. [2,1,3] ergeben. (*Hinweis:* Es könnte hilfreich sein, evtl. von `size` gesetzte Markierungen stehen zu lassen.)

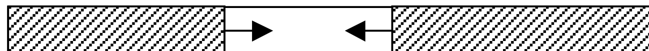
Aufgabe 3 (6 Punkte, 13 min)

Implementieren Sie in der folgenden Klasse `BinTree` die Methode `duplicate`, die eine *tiefe Kopie* des Baums herstellt. Die Klasse kann bei Bedarf erweitert werden.

```
interface Duplicatable<E> {
    E duplicate();
}
public class BinTree<E extends Duplicatable<E>>
                       implements Duplicatable<BinTree<E>>  {
    private E val;
    private BinTree<E> left, right;
    ...
    public BinTree<E> duplicate()  {
        ...
    }
}
```

Aufgabe 4 (12 Punkte, 27 min)

Wenn man einen Keller als Feld repräsentiert, wird mit den jeweils nicht belegten Zellen Speicherplatz verschwendet. Benötigt man zwei Keller, so bietet sich eine gemeinsame Repräsentation mit *einem* Feld an, in dem die beiden Keller gegeneinander laufen und ein Kellerüberlauf nur dann eintritt, wenn *beide* Keller voll sind:



Wir modellieren ein Keller-Paar für Zeichenketten wie folgt:

```
x, y :: [String]
max  :: Int

-- abstrakte Invariante
ainv(x,y,max) = length x + length y <= max
```

Die Implementierung in Java sehe so aus:

```
class StackPair { .......
    public void pushX(String s) throws Overflow ...
    public String popX() throws Underflow ...
    public void pushY(String s) throws Overflow ...
    public String popY() throws Underflow ...
}
```

Gesucht ist:

a) (2 P.) Repräsentation und Konstruktor,
b) (1 P.) die konkrete Invariante, formuliert in Java,
c) (4 P.) die Abstraktionsfunktion in Pseudo-Haskell,
d) (5 P.) die konkrete Spezifikation von `pushX` und `popY`, prädikativ formal.