Fakultät Elektrotechnik/Informatik

Klausur Einführung in die Informatik II für Elektrotechniker 16. Juli 2003

Name: .....

Matr.-Nr. .....

Bearbeitungszeit: 120 Minuten

Bewertung
(bitte offenlassen :–)

Aufgabe	Punkte	Erreichte Punkte
1	6	
2	7	
3	5	
4	12	
5	14	
Summe	44	

Spielregeln (**Jetzt lesen!**):

- Benutzen Sie für die Lösung der Aufgaben **nur** das mit diesem Deckblatt ausgeteilte Papier. Lösungen, die auf anderem Papier geschrieben werden, können **nicht** bewertet werden. Schreiben Sie ihre Lösung auch auf die Rückseiten der Blätter; benötigen Sie für eine Lösung mehr als ein Blatt, finden Sie am Ende der Klausur Leerblätter. Zusätzliches Papier können Sie von den Tutoren bekommen.
- Tragen Sie jetzt (vor Beginn der eigentlichen Bearbeitungszeit !!!) auf *allen* Blättern ihren Namen und ihre Matrikelnummer ein.
- Schreiben Sie deutlich! Unleserliche oder zweideutige Lösungen können nicht gewertet werden.
- Schreiben Sie *nicht* mit Bleistift und *nicht* mit rotem oder grünem Stift (das sind die Farben für die Korrektur).
- Lesen Sie die Aufgaben jeweils bis zum Ende durch; oft gibt es hilfreiche Hinweise!
- Kommentare kosten Zeit; kommentieren Sie ihr Programm nur da, wo der Code alleine nicht verständlich wäre.
- Wir weisen noch einmal darauf hin, daß die Benutzung von Taschenrechnern und anderen elektronischen Hilfsmitteln nicht gestattet ist.

Viel Erfolg!

• **AUFGABE 1 (6 Punkte) Theorie.**

1. (3 Punkte) Welchen Aufwand hat das (intelligente) Suchen in den folgenden Datenstrukturen, jeweils im *sortierten* und im *unsortierten* Fall:

   (a) Arrays

   sortiert: unsortiert:

   (b) Listen

   sortiert: unsortiert:

   (c) 2-3-Bäume

   sortiert: unsortiert:

2. (2 Punkte) Zeichnen Sie einen balancierten Binärbaum, bei dem durch *postorder*-Traversierung die Zeichenkette `P O S T O R D E R B A U M` (ohne Leerzeichen) ausgegeben wird.

3. (1 Punkt) Was verstehen Sie unter einer *Queue*?

• **AUFGABE 2 (7 Punkte) JAVA.**

1. (1 Punkt) Welche Bedeutungen hat das Schlüsselwort `final` in Java?

2. (2 Punkte) Was kann in Java mit *Interfaces*, nicht aber mit *abstrakten Klassen* gelöst werden? Wie ist es umgekehrt?

3. (1 Punkt) Unter welchen Umständen kann man sicher sein, daß *Casting* einer Objekt-Referenz nicht zu einem Laufzeitfehler (Programmabsturz) führt?

4. (3 Punkte) Welche Fehler enthält folgendes Java-Code-Fragment? Geben Sie jeweils die Zeilennummer an und beschreiben Sie den Fehler. Folgefehler werden ignoriert.

```
 1  class CreditCard {
 2    private int pin ;
 3  }
 4
 5  class CashDispenser {
 6    private int amount ;
 7
 8    void load(int amount) {
 9      this.amount = amount ;
10    }
11
12    boolean check(int pin) {
13      Terminal.println("ok") ;
14    }
15
16    void withdraw(CreditCard card, int amount) {
17      if (check(card.pin)) { this.amount -= amount ; }
18    }
19  }
20
21  class Bank {
22    CashDispenser dispenser ;
23
24    Bank() {
25      this.dispenser = dispenser ;
26      dispenser.load(1000000) ;
27    }
28  }
```

Diese Seite wurde absichtlich freigelassen.

• **AUFGABE 3 (5 Punkte) Numerik.** Berechnen Sie die Maßzahl ϕ des *goldenen Schnitts*. Diese ist definiert als Grenzwert der Folge:

$$a_n = \frac{fib(n+1)}{fib(n)}$$

wobei die *Fibonacci-Funktion fib* wie folgt rekursiv definiert ist:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2) \qquad (n \geq 2)$$

1. (2 Punkte) Schreiben Sie eine Hilfsmethode `double fib(int n)` zur Berechnung der Fibonacci-Funktion.
2. (3 Punkte) Programmieren Sie die Methode `double phi()`, welche die Approximation durchführt.

   Beenden Sie den Approximationsprozeß, wenn eine Genauigkeit von 9 Nachkomma-Stellen erreicht ist. Formulieren Sie diese Bedingung mit Hilfe einer (ebenfalls zu programmierenden) Hilfsmethode `boolean close(double x, double y, double eps)`.

Diese Seite wurde absichtlich freigelassen.

• AUFGABE 4 (12 Punkte) Vererbung.

In dieser Aufgabe sollen spezielle Bäume modelliert werden. Diese Bäume besitzen drei Sorten von Knoten: `Trunk` (Stamm), `Branch` (Ast) und `Leaf` (Blatt).

- `Trunk`-Knoten besitzen genau einen Nachfolger `rest`.
- `Branch`-Knoten besitzen genau zwei Nachfolger `left` und `right`.
- `Leaf`-Knoten besitzen keinen Nachfolger.

Alle drei Knotentypen sind zu einer gemeinsamen Oberklasse `Tree` zusammengefaßt. Die Nachfolger eines Knotens sollen Objekte dieser Klasse `Tree` sein.

Nun wollen wir bestimmte Baumformen definieren:

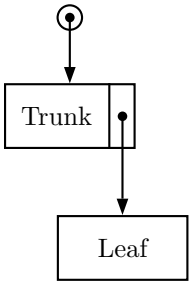
- *Baumkrone*:
  - Ein einzelnes `Leaf` ist eine Baumkrone.
  - Ein `Branch` ist eine Baumkrone, falls seine Nachfolger Baumkronen sind.
  - Ein `Trunk` ist *keine* Baumkrone.
- *wohlgeformter Baum*:
  - Ein `Leaf` ist *kein* wohlgeformter Baum.
  - Alle Nachfolger eines wohlgeformten Baumes müssen Baumkronen sein.

1. (3 Punkte) Schreiben Sie eine abstrakte Oberklasse `Tree`, welche beliebige Baumknoten repräsentiert. Sie soll folgende *abstrakte* Methoden zur Verfügung stellen:
   - `Tree[] children()`, welche ein Array mit den Nachfolgern des aktuellen Baumknotens liefert. Die Größe des Arrays (0, 1 oder 2) hängt von Knotentyp ab.
   - `boolean isCrown()`, welche prüft ob es sich um eine Baumkrone handelt.

   Definieren Sie in der Klasse `Tree` außerdem eine Methode
   - `boolean isProperTree()` welche prüft ob es sich um einen wohlgeformten Baum handelt. Verwenden Sie dazu die oben eingeführten abstrakten Methoden.
2. (6 Punkte) Schreiben Sie die Klassen `Trunk`, `Branch` und `Leaf`, welche die Klasse `Tree` spezialisieren. Jede dieser drei Klassen soll einen geeigneten Konstruktor sowie die abstrakten Methoden aus der Klasse `Tree` definieren.

   Hinweis: Sie müssen *keine* Attribute für Werte definieren, die an den Knoten gespeichert werden sollen.
3. (2 Punkte) Definieren Sie (*außerhalb* der bereits programmierten Klassen) eine Methode `boolean bonsai(Tree t)`, welche prüft ob der übergebene Baum `t` genau aus einem `Trunk` gefolgt von einem `Leaf` besteht.

   Hinweis: Sie können dabei annehmen, daß `t` nicht `null` ist.



4. (1 Punkt) Geben Sie ein Stück Java-Code an, welches einen solchen Bonsai-Baum erzeugt.

Diese Seite wurde absichtlich freigelassen.

• **AUFGABE 5 (14 Punkte) Umkehrbare Listen.**

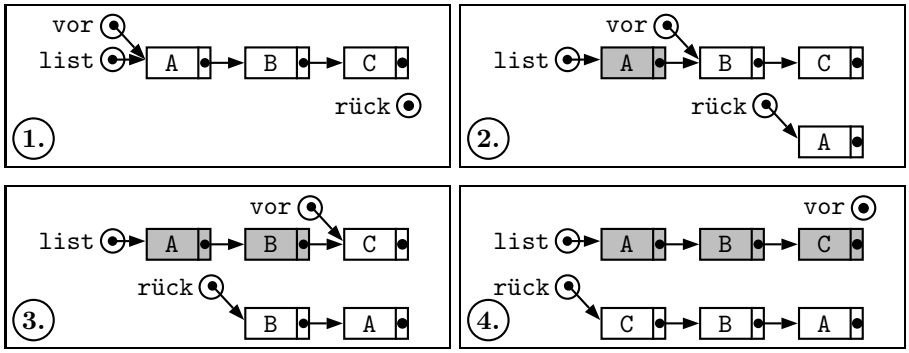
1. (3 Punkte) Gegeben sei eine Klasse für *einfach* verkettete Listenzellen:

```
class SingleCell {
  Object content;
  SingleCell next;
  SingleCell(Object content, SingleCell next) {
    this.content = content;
    this.next = next;
  }
}
```

Programmieren Sie eine Methode

– `SingleCell reverse(SingleCell list)`, welche die Reihenfolge der Listenelemente in der Liste `list` umkehrt. Die übergebene Liste soll dabei allerdings nicht zerstört werden.

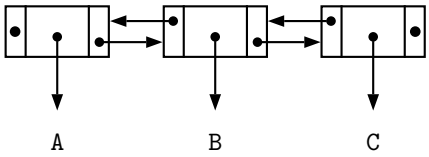
Hinweis: Das folgende Bild zeigt eine mögliche Lösungsidee mit zwei Hilfsvariablen `vor` und `rück`:



2. (2 Punkte) Um das Umkehren einer Liste zu vereinfachen, bietet sich die Verwendung von *doppelt* verketteten Listenzellen an. Schreiben Sie eine Klasse `DoubleCell`, die geeignete Attribute `content`, `next` und `previous` und einen Konstruktor

   – `DoubleCell(Object content, DoubleCell next, DoubleCell previous)`

   besitzt.



3. (9 Punkte) Eine doppelt verkettete Liste kann einfach umgekehrt werden, indem die Rollen der Vorwärts- und Rückwärts-Referenz vertauscht werden. Die erste Zelle übernimmt dann die Rolle der letzten und umgekehrt.

   Schreiben Sie hierzu eine Klasse `RevList`, die über folgende, von außen *nicht* zugreifbare Attribute verfügt:

   – `DoubleCell first, last` sind Referenzen auf erste und letzte Zelle.

   – `boolean reversed` ist `true`, falls die Liste im Moment umgekehrt zu lesen ist, also die Rollen von `first` und `last` vertauscht sind.

   Schreiben Sie in der Klasse `RevList` weiterhin folgende Methoden (*Hinweis:* Alle Zugriffs-Methoden sollen die *aktuelle Leserichtung* beachten!):

   – `RevList()` erzeugt eine leere Liste.

   – `DoubleCell currentFirst()` liefert die tatsächliche erste Zelle. (Ist die Liste gerade umgekehrt, liefert `currentFirst().content` das letzte, ansonsten das erste Element)

   – `void reverse()` kehrt die Liste um. Beachten Sie, daß bei zweimaligem Umkehren wieder die ursprüngliche Liste entsteht.

   – `Object select(int i)` liefert den Inhalt der augenblicklich `i`-ten Zelle. Im Fall eines undefinierten Zugriffs soll `null` zurückgeliefert werden.

   – `void append(Object x)` fügt das Objekt `x` am augenblicklichen Ende der Liste (also je nach Leserichtung hinten bzw. vorne) ein.

Diese Seite wurde absichtlich freigelassen.

Diese Seite wurde absichtlich freigelassen.

Name: .....

Matr.-Nr. .....