

## Die Musterlösung Probeklausur ALP3 WS 05/06

Lösungen stehen in eckigen Klammern [ ].

Achtung dies ist eine Probeklausur. Leiten Sie nicht zuviel über die echte Klausur aus der vorliegenden Aufgabensammlung ab.

80 Punkte == 80 Minuten Zeit (die echte Klausur wird 90 Minuten lang sein bei 90 Punkten).

### Verständnisfragen (15 Punkte)

Richtig oder Falsch? Richtige Antworten zählen 1,5 Punkte. Falsche Antworten 1 Punkt Abzug. Die Summe der Punkte dieser Aufgabe kann aber nicht unter 0 sinken.

	Richtig	Falsch
1.) Wird beim Hashing Kollisionsbehandlung mit offener Speicherung betrieben, dann braucht man keine Sondierung nach einem freien Speicherplatz zu betreiben.		
[JA. Bei offener Speicherung muss man nicht nach einem freien Platz suchen, sondern fügt das Element dem jeweiligen Bucket hinzu.]		
2.) Im Vergleich zwischen sortierten Feld und Hashfeld schneidet letzterer bei der Suche solange besser ab, bis die Belegung 60% nicht übersteigt.		
[NEIN: 80%]		
3.) Im Gegensatz zur Tiefensuche macht es bei der Breitensuche keinen Sinn von Präordnung, Inordnung, Postordnung zu sprechen.		
[JA. Da die Breitensuche Ebene nach Ebene im Baum heruntersteigt, macht es nur Sinn eine Reihenfolge innerhalb einer Ebene (links nach rechts) anzugeben]		
4.) Vielwegbäume lassen sich als Binärbäume darstellen, welche dann mindestens genauso hoch sind, aber eine kleiner-gleiche Anzahl von Blätter haben		
[JA.]		
5.) Ein vollständiger Baum mit Höhe h hat $2^h - 1$ Knoten.		
[Nein: Er hat k Knoten. $2^{(h-1)} - 1 < k \leq 2^h - 1$ ]		
6.) Ein interner Iterator überlässt dem Aufrufer die Kontrolle über die Traversierung der Datenstruktur.		
[Nein. Beim externen Iterator kann der Aufrufer über next() die Traversierung steuern.]		
7.) Die Abstraktionsfunktion ist eine surjektive Abbildung von Werten der konkreten Repräsentation auf Werte des abstrakten Modells.		

[Nein. In den meisten Fällen wird es für eine konkrete Repräsentation nicht möglich alle Werte des abstrakten Modells anzunehmen. Man denke nur z.B. an ein Bitfeld mit eingeschränktem Wertebereich $0 \leq n \leq \text{max.}$ ]		
8.) In einer korrekten Repräsentation muss für jedes Element aus der konkreten Wertemenge, welches der konkreten Invariante genügt, auch die Abbildung dieses Elements durch die Abstraktionsfunktion der abstrakten Invariante genügen.		
[Ja]		
9.) Durch eine Spezifikation ist genau eine konkrete Repräsentation vorgegeben.		
[Nein]		
10.) Durch eine konkrete Repräsentation ist genau eine Implementierung vorgegeben.		
[Nein]		

## Aufgabe Typverträglichkeit, Vererbung, Generizität (20 Punkte)

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.

(Diese Aufgabe stammt aus der Nachklausur 2005.)

[ a) Der Quelltext enthält keine Fehler. Die Ausgabe lautet:

6

-1

b)

<code>E[]a = new E[8];</code>	Hier wird versucht, ein Feld zu erzeugen, dessen Elementtyp durch die Typvariable E bestimmt ist. Dies ist in Java 5 nicht möglich (generic array creation).
-------------------------------	--

<pre>boolean[] b = new boolean[8]; static class D {     boolean m(int i) {         return !b[i];     } }</pre>	<p>Hier wird aus der statischen inneren Klasse D versucht, auf ein nicht-statisches Element der umschließenden Klasse C zuzugreifen. Dies ist nicht möglich.</p>
<pre>void m(byte b) {...} static void test() { ...     int x = 8;     c.m(x); ...}</pre>	<p>Der Typ des Parameters b der Methode m ist byte, aufgerufen wird m mit einem int, int ist jedoch nicht verträglich mit byte.</p>

### **Algebraische Spezifikation (10 Punkte)**

Sei folgende Signaturen von Operationen auf binären Bäumen mit Typparameter T gegeben:

EmptyTree: Tree

Construct: T X Tree X Tree -> Tree

Left: Tree -> Tree

Right: Tree -> Tree

Root: Tree -> T

Height: Tree -> Int

Size: Tree -> Int

Formulieren Sie 10 Axiome als algebraische Spezifikation für diesen ADT.

[

Root(Construct(t, a, b)) == t

Left(Construct(t, a, b)) == a

Right(Construct(t, a, b)) == b

Left(EmptyTree), Right(EmptyTree), Root(EmptyTree) sind undefiniert.

Height(EmptyTree) == -1

Height(Construct(t,a,b)) == 1 + max(Height(a),Height(b))

Size(EmptyTree) == 0

Size(Construct(t,a,b)) == 1 + Size(a) + Size(b)

]

### **Spezifikation (30 Punkte):**

a.) Eine Multimenge (Bag)  $\{|t|\}$ , sei durch eine Abbildung repräsentiert:

**Modell:** `type Bag t = t -> Int`

**Invariante:** `inv bag = all(\x -> bag x >= 0)t`

Geben Sie eine Abstraktionsfunktion an:

```
[
abs bag = bagOf [(x,bag x) | x <- t]
  where bagOf [] = { | |}
        bagOf (x,0):xs = bagOf xs
        bagOf (x,i):xs = {|x|} ++ bagOf (x,i-1):xs
]
```

b.) Ein Menge  $\{t\}$  sei durch ein Bitfeld repräsentiert:

```
class IntSet { // { t } (model)
  protected final boolean[] set;
```

Geben Sie eine Abstraktionsfunktion an:

```
[
// abs set = { i | i<-[0..set.length-1], set[i] ]
]
```

c.) Geben Sie die Invariante für einen AVL-Baum umgangssprachlich an:

[Der Baum muss ein 1-ausgeglichener binärer Suchbaum sein, d.h. für jeden Knoten unterscheidet sich die Höhe seiner Teilbäume um höchstens eins.]

## **Baum-Traversierung (15 Punkte)**

Gegeben sei folgender Rot-Schwarz-Baum in zusätzlich gefädelter Darstellung:

```
public class ThreadedRedBlackTree<T extends Comparable<T>> {
    class Node {
        private T val;
        private Node l, r;
        private boolean black;
        private boolean rthread = true; // true -> "Faden", false -
        > echtes Kind
        Node(T val) {
            this.val = val;
        }
    }
    private Node root;
    public int count(Predicate<T> valuePredicate,
        Predicate<Boolean> blackPredicate) {
```

Implementieren Sie die Methode count, welche die Anzahl aller Knoten zurückgeben soll, die beide Prädikate (eins bezogen auf den Wert des Knoten und das andere bezogen auf die Farbe) wahr werden lassen.

Um zum Beispiel alle roten Knoten zu zählen könnte man aufrufen:

```
        public static void main(String[] args) {
            ThreadedRedBlackTree<Integer> set = new
            ThreadedRedBlackTree<Integer>();

            set.count(
                new Predicate<Integer>() {
                    public boolean decide(Integer i) { return true;
                }
            },
            new Predicate<Boolean>() {
                public boolean decide(Boolean isBlack) { return
            false; }
            });
        }
```

[2 Lösungen:

1.) Ohne Fädung (war ja nicht gefordert):

```
public int count(Predicate<T> valuePredicate,
Predicate<Boolean> blackPredicate) {
    int count = 0;
    List<Node> todo = new LinkedList<Node>();
    todo.add(root);
    while (!todo.isEmpty()) {
        Node current = todo.remove(0);
        if (current == null) continue;
        if (valuePredicate.decide(current.val) &&
blackPredicate.decide(current.black)) {
            count++;
        }
        todo.add(current.l);
        todo.add(current.r);
    }
    return count;
}
```

2.) Man benutzt die Fädung

```
public int count2(Predicate<T> valuePredicate,
Predicate<Boolean> blackPredicate) {
    int count = 0;
    Node current = root;
    boolean comingBack = false;
    while (current != null) {
        if (comingBack) {
            if (current.r != null) {
                current = current.r;
                comingBack = current.rthread;
            } else
                current = null;
        } else {
            if (valuePredicate.decide(current.val) &&
blackPredicate.decide(current.black)) {
                count++;
            }
            if (current.l != null) {
                current = current.l;
            }
        }
    }
}
```

```
        comingBack = false;
    } else {
        comingBack = current.rthread;
        current = current.r;
    }
}
}
return count;
}
```

## O-Kalkül (10 Punkte)\*

Aus einem Feld lässt sich ein Heap „in-place“ erstellen, wenn man mit der rechten Hälfte des Feldes als Teilheaps beginnt und dann Elemente iterativ einfügt (vgl. 8.4.2. Version 2, rearrangieren A, Variante 2). Für jedes Element muss man hierbei seine Einordnungen in den Teilheap korrigieren, wie es auch beim Entfernen von Elementen für das nach oben gesetzte Element der Fall ist („runtertrickeln“). Beweisen Sie, dass diese Operation in  $O(n)$  und nicht in den erwarteten  $O(n \log n)$  durchgeführt werden kann.

\* Extra-Aufgabe für Theoretiker. Macht nix, wenn 10 Minuten etwas illusorisch waren ;-).

[

Erinnerung: Kosten für das Entfernen eines Knotens ist proportional zur Höhe des Baums  $O(\log n)$ .

Worst-Case Abschätzung für die Kosten für das Arrangieren des Heaps:

Stufe 0: Rechte  $n / 2$  Einträge: Kosten: 0 // Sind am richtigen Platz und damit fertig

Stufe 1: Nächste  $n / 4$  Einträge: Kosten: 1 // Im Worst-Case einmal tauschen.

Stufe 2: Nächste  $n / 8$  Einträge: Kosten: 2 // Im Worst-Case zweimal tauschen.

Stufe 3: Nächste  $n / 16$  Einträge: Kosten 3 // Im Worst-Case dreimal tauschen

...

Stufe  $\log(n)$ : Wurzel: Kosten  $\log(n)$  // Im Worst-Case ganz durchtauschen

$$\sum_{i=1}^{\log n} \frac{n}{2^i} i = n \sum_{i=1}^{\log(n)} \frac{i}{2^i} \quad (1)$$

$$\leq n \sum_{i=1}^{\log(n)} \frac{1}{2^{i/2}} + c \quad (2)$$

$$\leq n \sum_{i=1}^{2\log(n)} \frac{1}{2^i} + c \quad (3)$$

$$\leq n + c \quad (4)$$

(2) Abschätzung:  $i \leq 2^{i/2}$  für  $i \geq 4$

(3)  $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$

=> Kosten  $O(n)$

]