Algorithmen und Datenstrukturen
Musterlösung 7

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

10. Mai, zur Besprechung am 17. Mai

Aufgabe 1) Hashfunktionen

1. Voraussetzung für die Anwendbarkeit von universellem Hashing ist, dass es sich bei der Menge möglicher Schlüssel um eine endliche Menge von Zahlen $\mathcal{K} = \{0, \ldots, p-1\}$ handelt, wobei p eine Primzahl ist. Für die Listen-Klasse ist dies nicht gegeben, da die Listen beliebig lang werden können und daher keine Obergrenze für die Zahl der Schlüssel angegeben werden kann. Für die Tripel-Klasse ist universelles Hashing dagegen möglich, wenn für die Datentypen `X`, `Y` und `Z` universelles Hashing existiert. Die Menge $\mathcal{K}$ kann dann aus einer Kombination der Schlüsselmengen $\mathcal{K}_X$, $\mathcal{K}_Y$ und $\mathcal{K}_Z$ gebildet werden.

2. Für die Tripel-Klasse wird angenommen, dass die `hashCode`-Methode der Klassen `X`, `Y` und `Z` universelles Hashing durchführt und Schlüssel mit einer Länge von jeweils 32 Bit liefert, die zu einem Schlüssel der Länge $3 \cdot 32 = 96$ Bit konkateniert werden. Mit Hilfe einer 97 Bit langen Primzahl `p` und 96 Bit langen Zufallszahlen `a` und `b` wird universelles Hashing durchgeführt:

```
1  import java.math.*;
2  import java.util.*;
3
4  public class Triple<X,Y,Z> {
5
6     private final static BigInteger p = new BigInteger("
          95673531099682755594247122307");
7     private final static Random r = new Random();
8     private final static BigInteger a = new BigInteger(3*32, r).add(
          BigInteger.ONE);
9     private final static BigInteger b = new BigInteger(3*32, r);
10
11    X x;
12    Y y;
13    Z z;
14
15    public boolean equals(Object other) {
16       if (other instanceof Triple) {
```

```
17            Triple<X,Y,Z> t = (Triple<X,Y,Z>)other;
18            return this.x.equals(t.x) && this.y.equals(t.y) && this.z.
                  equals(t.z);
19         } else {
20            return false;
21         }
22      }
23
24      public int hashCode() {
25         int hx = x.hashCode();
26         int hy = y.hashCode();
27         int hz = z.hashCode();
28         BigInteger n = BigInteger.valueOf(hx);
29         n = n.shiftLeft(32);
30         n = n.or(BigInteger.valueOf(hy));
31         n = n.shiftLeft(32);
32         n = n.or(BigInteger.valueOf(hz));
33         n = a.multiply(n).add(b).mod(p);
34         return n.intValue();
35      }
36
37   }
```

Für eine Liste mit Elementen $a_0, \ldots, a_n$ wird unter Verwendung einer großen Primzahl p abwechselnd Addition und Multiplikation nach dem Schema $((a_0 \cdot p + a_1) \cdot p + a_2) \cdot p + \ldots$ durchgeführt. Dadurch beeinflussen sowohl Wert als auch Position der Elemente den Hash-Wert, und dank der großen Primzahl wird ein großer Wertebereich abgedeckt:

```
1  public class List {
2
3     private final static int p = 2082205841;
4
5     Element root;
6
7     public boolean equals(Object other) {
8        if (other instanceof List) {
9           Element one = this.root;
10          Element two = ((List)other).root;
11          while (one != null && two != null) {
12             if (one.x != two.x) {
13                return false;
14             }
15             one = one.next;
16             two = two.next;
17          }
18          return (one == null && two == null);
19       } else {
20          return false;
21       }
22    }
23
24
25    public int hashCode() {
26       Element e = root;
27       int hc = 0;
28       while (e != null) {
29          hc = hc * p + e.x; // mult. with prime
30          e = e.next;
31       }
32       return hc;
33    }
34 }
```

```
35
36
37 class Element {
38    int x;
39    Element next;
40
41    Element(int x, Element next) {
42       this.x = x;
43       this.next = next;
44    }
45 }
```

Aufgabe 2) Ketten-Multiplikation von Matrizen

1. Die Tabelle `m` enthält die Zwischenergebnisse `m[i][j]` mit den Kosten zur Berechnung des Produkts $A_i \times \cdots \times A_j$:

	j = 1	2	3	4
$i = 1$	0	150000	450000	950000
2		0	15000000	2250000
3			0	1500000
4				0

   Die Tabelle `s` enthält die Zwischenergebnisse `s[i][j] = k` zur Klammerung des Produkts $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$:

	j = 1	2	3	4
$i = 1$		1	2	3
2			2	2
3				3

2. Der Wert `s[1][4] = 3` gibt an, dass an Position 3 eine Klammerung eingefügt werden sollte, also $(A_1 \times A_2 \times A_3) \times A_4$. Anschließend kann rekursiv die Klammerung der Teilprodukte $(A_1 \times A_2 \times A_3)$ und A_4 bestimmt werden. A_4 kann nicht weiter geklammert werden, für das linke Teilprodukt wird die Klammerung aus `s[1][3] = 2` abgelesen, d.h. $((A_1 \times A_2) \times A_3)$. Insgesamt ergibt sich so die Klammerung $((A_1 \times A_2) \times A_3) \times A_4$.

3. Um die Klammerung für $A_i \times \cdots \times A_j$ ohne die Informationen aus `s` zu bestimmen, muss zunächst wieder ein `k` gesucht werden, so dass `m[i][k] + m[k+1][j] + d[i-1]*d[k]*d[j] = m[i][j]`, indem diese Bedingung mit einer Schleife für alle $k \in \{i, \ldots, j\}$ geprüft wird. Anschließend kann die Klammerung der Teilprodukte $A_i \times \cdots \times A_k$ und $A_{k+1} \times \cdots \times A_j$ rekursiv bestimmt werden. Die Komplexität dieses Verfahrens im Worst-Case ist $\mathcal{O}(n^2)$.