Prof. Dr. Ir. Joost-Pieter Katoen

Sebastian Junges, Benjamin Kaminski, David Korzeniewski, Tim Quatmann

Übung 8

– Musterlösung –

Hinweise:

- Die Lösungen müssen bis **Donnerstag, den 21. Juni um 16:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Die Übungsblätter **müssen** in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung abgegeben werden.
- Drucken Sie ggf. digital angefertigte Lösungen aus. Abgaben z.B. per Email sind nicht zulässig.
- Namen und Matrikelnummer sowie die **Nummer der Übungsgruppe** sind auf jedes Blatt der Abgabe zu schreiben. Abgaben, die aus mehreren Blättern bestehen **müssen geheftet bzw. getackert** werden! Die **Gruppennummer muss sich auf der ersten Seite oben links** befinden.
- **Bei Nichtbeachten der obigen Hinweise müssen Sie mit erheblichen Punktabzügen rechnen!**

Aufgabe 1 (Countingsort): (15 Punkte)

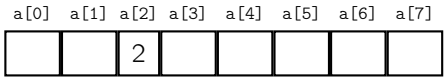
Sortieren Sie das Array $3, 6, 2, 4, 5, 5, 0, 2$ mit Countingsort ($k = 6$):
Geben Sie dazu das Histogramm, das Positionsarray an, sowie das Ergebnisarray nach jedem Einfügen.

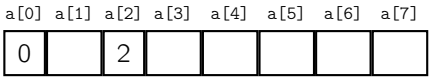
Lösung:

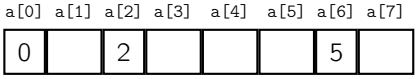
Histogramm: 1, 0, 2, 1, 1, 2, 1

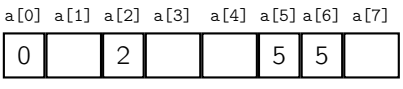
Positionsarray: 1, 1, 3, 4, 5, 7, 8

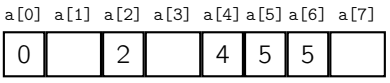
Array:

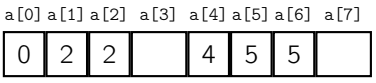


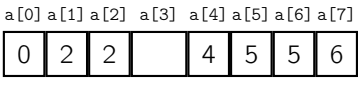


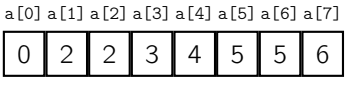












Aufgabe 2 (Hashing II): (25 Punkte)

Wir betrachten vollständig randomisiertes "Hashing" mit offener Adressierung, welches wie folgt funktionieren soll: Sei m die Größe der Hashtabelle.

Einfügen eines Schlüssels k:

1) Ermittle eine *zufällige* Slotnummer s zwischen 0 und $m-1$, jeweils mit Wahrscheinlichkeit $\frac{1}{m}$.

2a) Falls der Slot s belegt ist, gehe zu 1)

2b) Falls der Slot s frei ist, speichere Schlüssel k in Slot s.

Suchen eines Schlüssels k:

1) Ermittle eine *zufällige* Slotnummer s zwischen 0 und $m-1$, jeweils mit Wahrscheinlichkeit $\frac{1}{m}$.

2a) Falls der Slot s den Schlüssel k nicht enthält, gehe zu 1)

2b) Falls der Slot s den Schlüssel k enthält, gebe s zurück.

a) Leiten Sie in Abhängigkeit vom Füllgrad α die Average-Case-Komplexität für das Einfügen eines Schlüssels k in die Hashtabelle her.

Hinweise:
Für $0 \leq q < 1$ gilt $\sum_{k=0}^{\infty} q^k \cdot k = \frac{q}{(q-1)^2}$ und $\sum_{k=0}^{\infty} q^k = \frac{1}{1-q}$.
Außerdem gilt $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \in \mathcal{O}\big(\ln(n)\big)$.

b) Leiten Sie in Abhängigkeit von der Anzahl $\#(k)$ an Vorkommnissen eines Schlüssels k in der Hashtabelle die Average-Case-Komplexität für die erfolgreiche Suche nach einem Schlüssel k her.

c) Leiten Sie die Average-Case-Komplexität für die erfolglose Suche nach einem Schlüssel k her.

d) Beurteilen sie die Güte von vollständig randomisiertem Hashing auf Grundlage der in der Vorlesung aufgeführten Gütekriterien von Hashfunktionen. Fällt ihnen ein zusätzliches, in der Vorlesung nicht aufgeführtes, Gütekriterium ein, dass im Zusammenhang mit randomisiertem Hashing relevant ist?

Lösung:

a) Seien n Schlüssel bereits in die Hashtabelle der Größe m eingefügt. Wir nehmen an, dass $n < m$. Der Füllgrad der Hashtabelle ist dann

$$\alpha = \frac{n}{m}$$

Für die asymptotische Average-Case-Analyse zählen wir nun, wie oft der Schritt 1 durchgeführt wird. Die Wahrscheinlichkeit, dass beim Ermitteln einer zufälligen Slotnummer ein *belegter* Slot getroffen wird, ist

$$\frac{n}{m} = \alpha \ ,$$

denn es gibt n belegte Slots aus insgesamt m möglichen Slots, die jeweils alle mit gleicher Wahrscheinlichkeit getroffen werden.

Dual dazu beträgt die Wahrscheinlichkeit, dass beim Ermitteln einer zufälligen Slotnummer ein *freier* Slot getroffen wird,

$$\frac{m-n}{m} = 1-\alpha \ ,$$

denn es gibt $m-n$ freie Slots aus insgesamt m möglichen Slots, die jeweils alle mit gleicher Wahrscheinlichkeit getroffen werden.

Die erwartete Anzahl an Schritten, die gebraucht werden, bis ein freier Slot gefunden wird, ist die Summe über alle Möglichkeiten, zunächst i belegte Slotnummern zu generieren und anschließend eine unbelegte zu generieren, multipliziert mit der Wahrscheinlichkeit für das Auftreten dieses Ereignisses. Aufgeschrieben als Summe ergibt sich also

$$\begin{aligned}
&\sum_{i=0}^{\infty} \alpha^i \cdot (1-\alpha) \cdot (i+1) \\
&= (1-\alpha) \cdot \sum_{i=0}^{\infty} \alpha^i \cdot (i+1) \\
&= (1-\alpha) \cdot \left(\sum_{i=0}^{\infty} \alpha^i \cdot i + \sum_{i=0}^{\infty} \alpha^i \right) \\
&= (1-\alpha) \cdot \left(\frac{\alpha}{(\alpha-1)^2} + \frac{1}{1-\alpha} \right) && \text{(siehe Hinweise)} \\
&= (1-\alpha) \cdot \left(\frac{\alpha}{(1-\alpha)^2} + \frac{1}{1-\alpha} \right) && ((a-b)^2 = (b-a)^2) \\
&= (1-\alpha) \cdot \left(\frac{\alpha}{(1-\alpha)^2} + \frac{1-\alpha}{(1-\alpha)^2} \right) \\
&= \frac{\alpha}{1-\alpha} + \frac{1-\alpha}{1-\alpha} \\
&= \frac{1}{1-\alpha}
\end{aligned}$$

Damit ist die asymptotische Average-Case-Komplexität für das Einfügen in die Hashtabelle $\mathcal{O}(1/1-\alpha)$.

b) Da es sich um eine erfolgreiche Suche handeln soll, nehmen wir an, dass $\#(k) > 0$. Für die asymptotische Average-Case-Analyse zählen wir nun, wie oft der Schritt 1 durchgeführt wird. Die Wahrscheinlichkeit, dass beim Ermitteln einer zufälligen Slotnummer *nicht* der gesuchte Schlüssel k getroffen wird, ist

$$\frac{m-\#(k)}{m} =: \beta \ ,$$

denn es gibt $m-\#(k)$ Slots aus insgesamt m möglichen Slots, die nicht den gesuchten Schlüssel k enthalten und jeweils alle mit gleicher Wahrscheinlichkeit getroffen werden.

Dual dazu beträgt die Wahrscheinlichkeit, dass beim Ermitteln einer zufälligen Slotnummer der gesuchte Schlüssel k getroffen wird,

$$\frac{\#(k)}{m} = 1-\beta \ ,$$

denn es gibt $\#(k)$ Slots aus insgesamt m möglichen Slots, die den Schlüssel k enthalten und jeweils alle mit gleicher Wahrscheinlichkeit getroffen werden.

Die erwartete Anzahl an Schritten, die gebraucht werden, bis der Schlüssel k gefunden wird, ist die Summe über alle Möglichkeiten, zunächst i Slotnummern, die den Schlüssel k nicht enthalten, zu generieren und anschließend eine zu generieren, die den Schlüssel k enthält, multipliziert mit der Wahrscheinlichkeit für das Auftreten dieses Ereignisses. Aufgeschrieben als Summe ergibt sich also

$$\sum_{i=0}^{\infty} \beta^i \cdot (1-\beta) \cdot (i+1)$$

Diese Summe ist die gleich wie in Aufgabe a), wobei α durch β ersetzt werden muss, und als Auflösung ergibt sich somit

$$\frac{1}{1-\beta} = \frac{m}{\#(k)}$$

Damit ist die asymptotische Average-Case-Komplexität für die erfolgreiche Suche nach einem Schlüssel k gegeben durch $\mathcal{O}(m/\#(k))$.

c) Bei der erfolglosen Suche nach dem Schlüssel k kommt k überhaupt nicht in der Hashtabelle vor. Die vorliegende Hashingmethode hat jedoch keinerlei Möglichkeit, herauszufinden, dass der Schlüssel tatsächlich nicht vorkommt. Insbesondere gibt es in diesem Fall kein Abbruchkriterium und die Suche nach k läuft unendlich lang. Die Komplexität ist in diesem Falle also $\Theta(\infty)$.

d) Die vorgeschlagene Hashfunktion ist einfach zu berechnen, surjektiv, gleichverteilt Schlüssel optimal auf alle Indizes und streut ähnliche Schlüssel optimal breit über die Hashtabelle. Insoweit handelt es sich um eine optimale Hashfunktion.

Ein weiteres Kriterium, dass in der Vorlesung nicht angesprochen wurde und für diese Hashfunktion ein Problem darstellt, ist die Wiederauffindbarkeit von Schlüsseln. Im Falle, dass der Schlüssel nicht in der Hashtabelle enthalten ist, dauert die Suche nach dem Schlüssel unendlich lange. Insofern ist diese Hashfunktion von unterster Güte.

Aufgabe 3 (Güte von Hashingfunktionen): (10 Punkte)

Beurteilen Sie anhand der in der Vorlesung aufgeführten Gütekriterien von Hashfunktionen, d.h.

- einfache Berechenbarkeit
- Surjektivität
- Gleichverteilung auf alle Indizes
- Möglichst breite Verteilung ähnlicher Schlüssel auf die Hashtabelle,

jeweils die folgenden Hashfunktionen:

- $f\colon \{1, 2, 3, \ldots, 100\} \to \{0, 1, 2, \ldots, 10\},\ x \mapsto \left\lfloor \frac{10}{x} \right\rfloor$
- $g\colon \mathbb{N} \to \mathbb{Z}/101\mathbb{Z},\ x \mapsto 2^x \bmod 101$
- $h\colon \{0, 1, 2, \ldots, 100\} \to \{0, 1, 2, \ldots, 10\},\ x \mapsto x \bmod 11$
- $i\colon \mathbb{N} \to \{0, 1, 2, \ldots, 50\},\ x \mapsto \left\lfloor \frac{x}{2} \right\rfloor \bmod 51$

Begründen Sie ihre Antworten.

Lösung:

Die Funktion f ist *einfach zu berechnen* aber nicht *surjektiv*: Die Werte 9, 8, 7 und 6 werden nicht getroffen. Des Weiteren verteilt f die Werte ihres Definitionsbereiches *nicht mit gleicher Häufigkeit* auf ihren Bildbereich: Die meisten Werte werden auf 0 abgebildet. Auch werden *ähnliche Schlüssel nicht* möglichst breit auf den Bildbereich *verteilt*.

Die Funktion g ist nicht surjektiv, da die 0 nicht getroffen wird. Entfernt man jedoch die 0 aus dem Bildbereich, so erfüllt g alle Kriterien einer guten Hashfunktion.

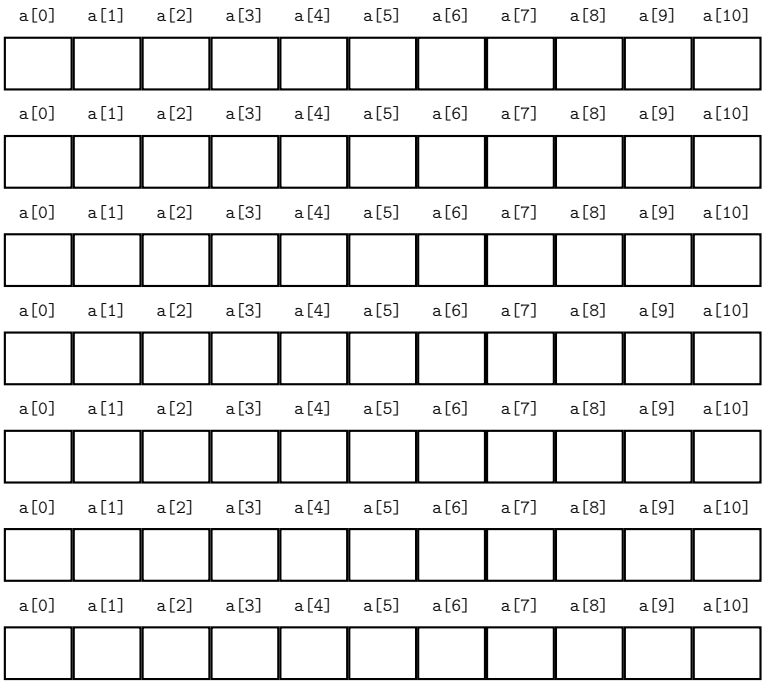
Die Funktion h erfüllt alle Kriterien einer guten Hashfunktion. Die Streuung ist jedoch nicht so gut wie die von g.

Die Funktion i ist *einfach zu berechnen* und *surjektiv*. i verteilt die Werte ihres Definitionsbereiches *mit gleicher Häufigkeit* auf ihren Bildbereich. *Ähnliche Schlüssel* werden aber *nicht* möglichst breit auf den Bildbereich *verteilt*. Zum Beispiel werden die benachbarten Zahlen 0 und 1 beide auf 0 abgebildet.

Aufgabe 4 (Hashing mit linearer Sondierung): (15 Punkte)

Fügen Sie die folgenden Werte in der angegebenen Reihenfolge in das Array a der Länge 11 unter Verwendung der *Multiplikationsmethode* ($c = 0.01$) mit *linear Sondierung* ein:

5, 21, 23, 17, 11, 7, 1.



Lösung:

m = 11, c = 0.01:

5										
5		21								
5		21	23							

5	17	21	23							
5	17	21	23	11						
5	17	21	23	11	7					
5	17	21	23	11	7	1				

______________________.

Aufgabe 5 (Hashing mit quadratischer Sondierung): (20 Punkte)

Fügen Sie die folgenden Wert in der angegebenen Reihenfolge in das Array a der Länge 11 unter Verwendung der *Multiplikationsmethode* ($c = 0.01$) mit *quadratischer Sondierung* ($c_1 = 2.0$, $c_2 = 1.0$) ein:

5, 21, 23, 17, 11, 7, 1.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Lösung: ______________________

______________________.

m = 11, c = 0.01, c_1 = 2.0, c_2 = 1.0:

5										
5		21								
5		21			23					

5	17	21			23					
5	17	21		11	23					
5	17	21	7	11	23					
5	17	21	7	11	23			1		

Aufgabe 6 (Laufzeit): (15 Punkte)

Für die nächste Präsenzübung überlegen wir uns, nochmal Laufzeiten abzufragen. Wir haben uns sogar schon die Antworten überlegt, aber uns fehlen noch die Algorithmen. Bitte, helfen Sie uns!
Schreiben Sie, in Pseudocode, **kurze** Algorithmen die einen `print()`-Aufruf so oft, wie angegeben, ausführen. Beachten Sie folgende Regeln:

- Der Algorithmus soll **keine** Rekursion benutzen. Schleifen und If-Abfragen sind natürlich erlaubt.
- Der Pseudo-code soll ausschliesslich folgende Operationen ausführen: $+$, $\cdot$, $/$, mod
- Integer können mit den üblichen Relationen $=, <, \leq$ verglichen werden.

a) A(n): Integer -> void
mit $\frac{2n}{3} + 1$ `print()`-Aufrufe.

b) B(n): Integer -> void
mit $n^3 + n^2$ `print()`-Aufrufe.

c) C(n): Integer -> void
mit $3^n + n^n$ `print()`-Aufrufe.

d) E(n): Integer -> void
mit $\Theta(\log(\log(n)))$ `print()`-Aufrufe.

e) F(n): Integer -> void
mit $\Theta(\sqrt[3]{\log n})$ `print()`-Aufrufe.

Lösung:

a)

```
A(n): Integer -> void
  int m = 2*n/3 + 1
  for i in 1..m:
    print(..)
```

b)

```
B(n): Integer -> void

for i in 1..n:
    for j in 1..n:
        print(..)
        for k in 1..n:
            print(..)
```

c)

```
C(n): Integer -> void

x = 1
y = 1
for i in 1..n:
    x = x * n
    y = y * 3
for j in 1.. x+y:
    print(..)
```

d)

```
D(n): Integer -> void

c = 0
while n > 1:
    c = c + 1
    n = n / 2
while c > 1:
    c = c / 2
    print(..)
```

e)

```
E(n): Integer -> void

c = 0
while n > 1:
    c = c + 1
    n = n / 2
for i in 1 .. c:
    if i*i*i < c:
        print(..)
```