Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 7

Abgabe: Montag, **17.06.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Aufgabe 1 (*Hashing* [**10 Punkte**])

(a) Güte von Hash-Funktionen
Betrachten Sie die folgenden Funktionen, die ein Wort $s = a_1 \dots a_n$ auf einen Hash-Wert zwischen 0 und m abbilden. Dabei sind die a_i als ASCII-Werte gegeben, also $0 \leq a_i \leq 127$ für alle i.

- $h_1(s) = n \mod m$
- $h_2(s) = (\sum_{k=1}^{n} a_k) \mod m$
- $h_3(s) = (\sum_{k=1}^{n} k \cdot a_k^2) \mod m$
- $h_4(s) = ((h_{good}(s)^{p-1} \mod p) \mod m)$

wobei $h_{good}(s)$ eine Hash-Funktion ist, die alle wichtigen an eine Hash-Funktion gestellten Eigenschaften (einfache Berechenbarkeit, Surjektivität und Streuung) erfüllt und p eine Primzahl ist. Erläutern Sie, inwiefern die Funktionen h_i ($1 \leq i \leq 4$) die drei genannten Eigenschaften einer guten Hash-Funktion erfüllen. [3 Punkte]

Hinweis: Wenn Sie sich nicht sicher sind, wie sich die Hash-Funktionen verhalten, ist es ratsam, sie auf ein paar Beispiel-Wörter anzuwenden. Für die vierte Hash-Funktion können Sie dabei beispielsweise die drei anderen Hash-Funktionen anstelle von h_{good} verwenden. Dies dient jedoch lediglich Ihrer Intuition und ist für die Bearbeitung dieser Aufgabe nicht zwingend erforderlich.

(b) Einfaches Hashing
Gegeben sei eine Hash-Tabelle der Größe 19 und die folgenden beiden Hash-Funktionen über der Universalmenge $U = \{0, \dots, 99\}$:

- $h_1(x) =$ Quersumme von x
- $h_2(x) = x \mod 19$

1. Fügen Sie die Werte $12, 99, 21, 76, 23, 30$ sowohl mit h_1 als auch h_2 mittels
   (i) Hashing mit Verkettung
   (ii) Hashing mit linearem Sondieren

   in jeweils eine Tabelle ein (es sind also 4 Tabellen zu erstellen). Geben Sie die nicht-leeren Teile der Tabellen nach jedem Einfügeschritt an. [3 Punkte]
2. Löschen Sie anschließend nacheinander die Werte 21, 12 und 76 aus allen Tabellen aus dem vorigen Aufgabenteil. Geben Sie die nicht-leeren Teile der Tabellen nach jedem Löschschritt an. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist. [3 Punkte]
3. Suchen Sie in den Tabellen, die aus der letzten Teilaufgabe (nach dem Löschen) resultierten, nach dem Wert 12. Erläutern Sie, welches Vorgehen dafür jeweils nötig ist. [1 Punkt]

(c) Doppeltes Hashing
Diese Aufgabe ist aus der regulären Wertung genommen worden. Sie können jedoch durch die korrekte Lösung dieser Aufgabe Bonuspunkte erhalten. Die Bonuspunkte werden zu Ihren regulären Punkten addiert, als würden sie zu einer regulären Aufgabe gehören.

In der Vorlesung haben Sie unter anderem auch doppeltes Hashing kennen gelernt, bei dem eine zusätzliche Hash-Funktion verwendet wird, um Sondierungsschritte durchzuführen. In dieser Teilaufgabe sollen Sie dieses Verfahren und eine Variante davon implementieren. Dazu sollen Sie den von uns bereit gestellten Code vervollständigen. Letzterer enthält drei Klassen. Die Datei `MainClass.java` enthält die Hauptklasse, die zum Testen genutzt werden kann und nicht modifiziert werden soll. Ebenso soll die Datei `Hash.java` nicht modifiziert werden. Diese stellt eine einfache Datenstruktur für Hash-Werte mit einer zusätzlichen Markierung bereit, um angeben zu können, ob ein Hash-Eintrag frei (free), belegt (used) oder entfernt worden (deleted) ist. In der Datei `Hashing.java` soll von Ihnen die Methode `insert2` vervollständigt werden. Die Methode `insert1` implementiert bereits eine Einfüge-Operation nach doppeltem Hashing mit der Sondierungs-Funktion $h(k,i) = (h_1(k)+i\times h_2(k,m))$ mod m, wobei h_1 und h_2 Hash-Funktionen sind und h_2 sicherstellt, dass der resultierende Wert teilerfremd zu m ist. Die von Ihnen zu vervollständigende Methode soll im Gegensatz dazu bei einer Kollision die nächste Sondierungsposition sowohl für das neu einzufügende als auch für das bereits enthaltene Element an der Kollisionsposition berechnen. Sollte das alte Element an seiner neuen Position ohne weitere Sondierungen einzufügen sein, während das neue Element weitere Sondierungen benötigen würde, wird das alte Element an seine neue Position verschoben und das neue Element an der ursprünglichen Position des alten Elementes eingefügt. Ansonsten wird rekursiv versucht, das neue Element an der nächsten Sondierungsposition einzufügen.

Etwas formaler ausgedrückt funktioniert die von Ihnen zu implementierende Hashing-Variante für ein Element k_1 im i-ten Sondierungsschritt wie folgt:

- Ist $h(k_1, i)$ frei, so füge k_1 an dieser Position ein.
- Ist $h(k_1, i)$ belegt mit einem Element k_2, $h(k_1, i+1)$ ebenfalls belegt, k_2 wurde mit j Sondierungsschritten eingefügt (d. h. $h(k_1, i) = h(k_2, j)$) und $h(k_2, j+1)$ ist frei, so füge k_2 an der Position $h(k_2, j+1)$ ein und k_1 an der bisherigen Position von k_2.
- Ansonsten fahre mit der Sondierung für k_1 und $i+1$ fort.

Die von Ihnen zu ergänzenden Teile sind im Code durch Kommentare markiert. [2 Bonuspunkte]

(d) Experimenteller Vergleich
Diese Aufgabe ist aus der regulären Wertung genommen worden. Sie können jedoch durch die korrekte Lösung dieser Aufgabe Bonuspunkte erhalten. Die Bonuspunkte werden zu Ihren regulären Punkten addiert, als würden sie zu einer regulären Aufgabe gehören.

Die Testklasse gibt für das bereits vorgegebene und das von Ihnen ergänzte Hash-Verfahren aus, wie viele Kollisionen beim Einfügen von 500000 zufällig

gewählten Elementen in eine anfangs leere Hash-Tabelle der Größe 1000000 entstehen. Messen Sie einige Male die Kollisionszahlen und erklären Sie den gemessenen Unterschied. [1 Bonuspunkt]

Lösungsvorschlag

(a) Güte von Hash-Funktionen

h_1 ist zwar in konstanter Zeit pro Eingabewort (bei entsprechender Datenspeicherung) berechenbar, kann jedoch nur dann surjektiv sein, wenn die Eingabewörter sehr lang sind und m sehr klein ist. Außerdem gewährleistet sie eine Gleichverteilung der Indizes nur bei Gleichverteilung der Eingabelänge. Werden beispielsweise sehr unterschiedliche Wörter mit gleicher Länge eingefügt, so werden diese alle auf denselben Index abgebildet. Darüberhinaus werden ähnliche Schlüssel auf einen ziemlich engen Index-Bereich abgebildet.

h_2 ist in linearer Zeit aus der Eingabe berechenbar. Da der maximale Hash-Wert eines Wortes $n \cdot 127$ beträgt, darf auch hier m nicht viel größer als n sein, um Surjektivität zu erreichen. Sie ist auch nur dann gleichverteilend, wenn starke Bedingungen an die Eingabe geknüpft werden. Wie schon bei h_1 werden ähnliche Wörter auf einen engen Index-Bereich abgebildet. Beispielsweise werden alle Permutationen eines Wortes auf denselben Wert abgebildet.

h_3 behebt in gewisser Weise einige Schwachstellen von h_2 bei gleichbleibender asymptotischer Komplexität der Berechnung. Durch den zusätzlichen Vorfaktor und die Quadrierung sind größere Hash-Werte möglich als zuvor und auch Permutationen werden jetzt nicht mehr (per se) auf denselben Wert abgebildet. Ein Nachteil bleibt, dass Surjektivität nur dann erreicht wird, wenn m und n nicht zu weit auseinander liegen und tendenziell kürzere Wörter auf kleinere Hash-Werte abgebildet werden.

h_4 benutzt zwar eine eingebettete, gute Hash-Funktion, ist aber aus arithmetischen Gründen eine schlechte Wahl, da aus Fermats kleinem Satz folgt

$$h_4(x) = \begin{cases} 1 & h_{good}(x) \text{ ist nicht durch } p \text{ teilbar} \\ 0 & \text{sonst} \end{cases}$$

h_4 trifft also nur zwei mögliche Werte, unabhängig davon wie gut h_{good} oder p gewählt wurden.

(b) Einfaches Hashing

1. (i) 12 einfügen:

h_1		h_2	
3	12	12	12

99 einfügen:

3	12	4	99
18	99	12	12

21 einfügen:

		2	21
3	21 → 12	4	99
18	99	12	12

76 einfügen:

		0	76
3	21 → 12	2	21
13	76	4	99
18	99	12	12

23 einfügen:

3	21 → 12	0	76
5	23	2	21
13	76	4	23 → 99
18	99	12	12

30 einfügen:

		0	76
3	30 → 21 → 12	2	21
5	23	4	23 → 99
13	76	11	30
18	99	12	12

(ii) 12 einfügen:

h_1		h_2	
3	12	12	12

99 einfügen:

3	12	4	99
18	99	12	12

21 einfügen:

3	12	2	21
4	21	4	99
18	99	12	12

76 einfügen:

3	12	0	76
4	21	2	21
13	76	4	99
18	99	12	12

23 einfügen:

3	12	0	76
4	21	2	21
5	23	4	99
13	76	5	23
18	99	12	12

30 einfügen:

3	12	0	76
4	21	2	21
5	23	4	99
6	30	5	23
13	76	11	30
18	99	12	12

2. Beim Hashing mit Verkettung muss das Element in der jeweiligen Liste gesucht, dann gelöscht und anschließend noch gegebenenfalls Zeiger berichtigt werden.

   Beim Hashing mit linearem Sondieren wird jeweils zuerst in dem "Fach" gesucht, das mit dem Hashwert assoziiert ist. Enthält dieses das gesuchte Element, so kann es durch **del** ersetzt werden. Enthält es jedoch nicht das Element sondern **del** oder einen anderen Wert, so muss die Suche mit linearem Sondieren fortgesetzt werden.

   21 löschen:

(i)

	h_1	h_2
0		76
3	30 → 12	
4		23 → 99
5	23	
11		30
12		12
13	76	
18	99	

(ii)

	h_1	h_2
0		76
2		**del**
3	12	
4	**del**	
4		99
5	23	
5		23
6	30	
11		30
12		12
13	76	
18	99	

12 löschen:

(i)

	h_1	h_2
0		76
3	30	
4		23 → 99
5	23	
11		30
13	76	
18	99	

(ii)

	h_1	h_2
0		76
2		**del**
3	**del**	
4	**del**	
4		99
5	23	
5		23
6	30	
11		30
12		**del**
13	76	
18	99	

76 löschen:

(i)

	h_1	h_2
3	30	
4		23 → 99
5	23	
11		30
18	99	

(ii)

	h_1	h_2
0		**del**
2		**del**
3	**del**	
4	**del**	
4		99
5	23	
5		23
6	30	
11		30
12		**del**
13	**del**	
18	99	

3. Beim Hashing mit Verkettung muss die Liste der Werte, die mit dem Hashwert von 12 assoziiert ist, durchlaufen werden. In diesem Beispiel wird in

der ersten Tabelle die Liste mit dem Element 30 durchlaufen, bevor festgestellt wird, dass sich die 12 nicht in der Tabelle befindet. Bei der zweiten Tabelle wird dies sofort festgestellt, da die entsprechende Liste leer ist.

Beim Hashing mit linearem Sondieren wird jeweils zuerst in dem "Fach" gesucht, das mit dem Hashwert von 12 assoziiert ist. In diesem Beispiel enthalten jedoch beide nicht das Element 12 sondern **del**. Daher muss die Suche mit linearem Sondieren fortgesetzt werden. Dies gilt auch für den Fall, dass an einem Element nicht **del** sondern ein anderer Eintrag steht. Es wird dann nach 4 (für h_1) bzw. 1 (für h_2) Sondierungsschritt(en) festgestellt, dass sich die 12 nicht in der Tabelle befindet.

(c) Doppeltes Hashing
Siehe Quellcode.

(d) Experimenteller Vergleich
Das alternative Verfahren führt zu weniger Kollisionen. Dies liegt daran, dass lange Kollisionsketten manchmal dadurch vermieden werden können, dass bereits in der Tabelle vorhandene Elemente auf freie Positionen verschoben werden können. Kürzere Kollisionsketten verringern auch die Kollisionswahrscheinlichkeiten für spätere Einfüge-Operationen.

Aufgabe 2 (*Selektion* [**10 Punkte**])

Gegeben sei eine Folge von n unsortierten Integer-Zahlen.

(a) Entwerfen Sie einen Algorithmus, welcher das kleinste Element der Folge mit $n-1$ Vergleichen findet. Vergleiche, welche nicht die Folgenelemente selbst betreffen, z. B. Vergleiche zwischen den Folgenindizes, werden nicht dazugezählt. Außerdem soll jedes einzelne Element maximal $\lceil \log_2 n \rceil$ mal verglichen werden. Zeigen Sie, dass Ihr Algorithmus höchstens die geforderte Anzahl von Vergleichen benötigt. [6 Punkte]

(b) Erweitern Sie das Verfahren so, dass es das zweitkleinste Element von n Elementen im Worst-Case mit $n + \lceil \log_2 n \rceil - 2$ Vergleichen bestimmt. Zeigen Sie, dass Ihr Algorithmus höchstens die geforderte Anzahl von Vergleichen benötigt. Wenden Sie Ihr Verfahren auf die Elemente

$$10, 5, 3, 2, 8, 7, 1, 6, 9$$

an. Stellen Sie dabei die notwendigen Schritte in geeigneter Form dar. [2 Punkte]

(c) In der Vorlesung haben Sie den Algorithmus `PartitionSelect` kennengelernt mit welchem sich das k-kleinste Element einer Folge finden lässt. Während dieser Algorithmus im Average-case $O(n)$ Operationen benötigt, ist dessen Komplexität Im Worst-case $O(n^2)$. Beschreiben Sie eine Erweiterung des Algorithmus, welcher auch eine Worst-case Komplexität von $O(n)$ garantiert. Sie brauchen die Worst-case Komplexität von $O(n)$ nicht zu beweisen. [2 Punkte]

Lösungsvorschlag

(a) Finde kleinstes Element in Folge mit n Elementen:
Teile wie beim Merge Sort die Folge in zwei Partitionen uns bestimme rekursiv die Minima beider Partitionen. Das Minimum der Gesamtfolge ist dann das Minima der beiden Teilminima.

Algorithmus:

```
function PartitionMin(A[l..r])
    if r > l then
        m ← l + ⌊(r−l)/2⌋
        m_l ← PartitionMin(A[l..m])
        m_r ← PartitionMin(A[m + 1..r])
        if m_l ≤ m_r then
            return m_l
        else
            return m_r
        end if
    else
        return A[l]
    end if
end function
```

Um die maximale Anzahl von Vergleichen zu ermitteln stellen wir eine Rekursionsgleichung für die Anzahl der Vergleiche auf:

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= 2T(n/2) + 1 \\
&= 2\left[2T(n/4) + 1\right] + 1 \\
&= 4T(n/4) + 2 + 1 \\
&\quad\vdots \\
&= \sum_{i=0}^{\log_2 n - 1} 2^i \\
&= n - 1
\end{aligned}
$$

Für die Anzahl der Vergleiche eines einzelnen Elements lässt sich eine der beiden Partitionen vernachlässigen, da sich das Element nur in einer Partition befinden kann:

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= T(n/2) + 1 \\
&= \left[T(n/4) + 1\right] + 1 \\
&= 4T(n/4) + 1 + 1 \\
&\quad\vdots \\
&= \log_2 n
\end{aligned}
$$

Ist n keine Zweierpotenz, ist die Anzahl der Vergleiche eines einzelnen Elements also durch $\lceil \log_2 n \rceil$ nach oben beschränkt.

(b) Das zweitkleinste Element der Gesamtfolge muss sich unter den direkten Verlieren der Vergleiche mit dem kleinsten Element der Gesamtfolge befinden. Speiche also für das Minimum sämtliche direkten Verlierer. Dies sind nach Teilaufgabe (a) maximal $\lceil \log_2 n \rceil$ Elemente, unter welchen sich das Minimum mit maximal $\lceil \log_2 n \rceil - 1$ Vergleichen finden lässt. Damit ist also die Gesamtanzahl der benötigten Vergleiche $n - 1 + \lceil \log_2 n \rceil - 1 = n + \lceil \log_2 n \rceil - 2$.

TODO

(c) Der Worst-case tritt beim `PartitionSelect` Algorithmus dann auf, wenn als Pivotelement immer das kleinste oder größte Element der Folge gewählt wird. Um dies zu verhindern, muss also in Linearzeit sichergestellt werden, dass stets ein gewisser Prozentsatz des Array nach der Partitionierung abgespaltet werden kann. Wähle daher den Median der Mediane als Pivotelement.