Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Tobias Lieber

Sommersemester 2011
Übungsblatt 3
16. September 2011

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: 16. September 2011

Hausaufgabe 1

Implementieren Sie die Methoden `insert`, `remove` und `find` für Hashing mit Linear Probing. Verwenden Sie die Hashfunktion

$$h(x) = ax \bmod m.$$

Stellen Sie sicher, dass nach dem Löschen eines Elements die Invariante gilt, dass für jedes Element e in der Hashtabelle mit idealer Position $i = h(\texttt{key}(e))$ und aktueller Position j gilt: $T[i], T[i+1], ..., T[j]$ sind besetzt.
Verwenden sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `LinHash`.

Lösungsvorschlag
Siehe Übungswebseite.

Hausaufgabe 2

Implementieren Sie die Methoden `insert`, `remove` und `find` für Cuckoo-Hashing.
Als Hashfunktionen werden Funktionen vom Typ

$$\left(\left(\sum_{j=0}^{k-1} a_j x^j\right) \bmod p\right) \bmod n$$

mit einem Vektor $a = (a_0, ..., a_{k-1})$ und ganzen Zahlen k, p und n verwendet, die bei der Initialisierung übergeben werden. Beachten Sie, dass x hier nur eine ganze Zahl und *kein* Vektor ist. Die Zahl n gibt hier die Größe der Hashtabelle an. Außerdem soll bei der Initialisierung der Hashtabelle ein Wert `max` übergeben werden, der einen Höchstwert für die Anzahl der Verschiebungen angibt (in der Vorlesung sind dies $2 \log n$). Wenn dieser Wert der Verschiebungen erreicht wird, so dürfen Sie das Programm sofort beenden.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `DynHash`.

Lösungsvorschlag
Siehe Übungswebseite.

Hausaufgabe 3

Implementieren Sie in der Klasse `UIsmsArray` den MergeSort-Algorithmus, in der Funktion `sort`, der die Elemente in dem Feld `A` sortiert.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `UIsmsArray`.

Lösungsvorschlag

Siehe Übungswebseite.

Aufgabe 1

Konstruieren Sie eine statische, perfekte Hashtabelle für die Elemente:

$$(16,10,11) \quad (8,2,15) \quad (7,12,8) \quad (1,10,3) \quad (13,11,14) \quad (6,11,14)$$
$$(7,3,16) \quad (2,2,8) \quad (10,5,15) \quad (7,3,14) \quad (2,10,1) \quad (14,11,6)$$

Jedes Element x besteht aus den Stellen (x_0, x_1, x_2). Verwenden Sie jeweils passend eine der Hashfunktionen:

$$\left(\textstyle\sum_{i=0}^{2} 2^i x_i\right) \bmod 17$$
$$\left(\textstyle\sum_{i=0}^{2} a_i x_i\right) \bmod 7 \text{ mit } \mathbf{a} = (0,0,1) \text{ oder } \mathbf{a} = (6,6,2)$$
$$\left(\textstyle\sum_{i=0}^{2} a_i x_i\right) \bmod 3 \text{ mit } \mathbf{a} = (1,0,0) \text{ oder } \mathbf{a} = (0,2,2).$$

Lösungsvorschlag

Zuerst bestimmen wir die Größe der Hash-Tabelle um die 12 Elemente in Buckets zu partitionieren. Da die Hashtabelle optimal sein soll, kommt hierfür nur die erste Funktion in Frage. Somit ergeben sich folgende Hash-Werte:

Element	Bucket
$(7,3,14)$	1
$(2,2,8)$	4
$(8,2,15)$	4
$(13,11,14)$	6
$(2,10,1)$	9
$(14,11,6)$	9
$(7,3,16)$	9
$(16,10,11)$	12
$(7,12,8)$	12
$(10,5,15)$	12
$(1,10,3)$	16
$(6,11,14)$	16

Wir überprüfen noch, ob die Anzahl der Kolisionen ($C(h)$) passt. Da diese gerade $6+6+2+2 = 16 < 17$ ist die gewählte Hashfunktion definitiv geeinget.
Bemerkung: Die erste Hashfunktion entstammt einer 3-universellen Hashfamilie und daher im Allgemeinen nicht für $c = 1$ geeignet (größeres c bedeutet auch mehr Platzverbrauch, da mehr Buckets benötigt werden). Allerdings muss die Hashfunktion zum Partitionieren der Elemente nur das Kriterium $C(h) \leq \frac{2cn}{\alpha}$ erfüllen (Es reicht die gröbere Abschätzung, da nur diese später im Beweis verwendet wurde).
Nun müssen die Elemente der Buckets mittels der anderen Hashfunktionen injektiv abgebildet werden. Die beiden Buckets mit nur einem Element werden hier jeweils in dem einen freien Feld des Buckets abgespeichert.
Die injektiven Hashfunktionen sollen bei einer Bucketgröße von b_ℓ nun auf $1 + b_\ell(b_\ell - 1)$ Felder streuen. Daher ist klar, dass wir uns jeweils nur zwischen zwei Funktionen entscheiden müßen. Die Buckets der Größe zwei werden von den Hashfunktionen mit $\bmod 3$ und

die Elemente der Buckets 9 und 12 mit den anderen beiden verbleibenden Hashfunktionen gehandhabt.
Da die Hashfunktionen injektiv sein müßen, folgt dass das Bucket 4 von der Hashfunktion mit dem Vektor $(0, 2, 2)$ abgebildet wird. Die Injektivität erzwingt auch, dass das Bucket 9 von der Hashfunktion mit dem Vektor $(0, 0, 1)$ behandelt wird, sonst ergibt sich eine Kollision auf der Position 1 der jeweils letzten Elemente in der Tabelle.
Insgesamt ergeben sich also folgende Platzierungen der Elemente:

Element	Bucket	Position in Bucket
(7 ,3 ,14)	1	0
(2 ,2 ,8)	4	2
(8 ,2 ,15)	4	1
(13 ,11 ,14)	6	0
(2 ,10 ,1)	9	1
(14 ,11 ,6)	9	6
(7 ,3 ,16)	9	2
(16 ,10 ,11)	12	3
(7 ,12 ,8)	12	4
(10 ,5 ,15)	12	1
(1 ,10 ,3)	16	1
(6 ,11 ,14)	16	0

Die gesamte Tabelle setzt sich nun aus den einelementigen (leeren) Bucktes und den Buckets aus der Tabelle zusammen.

Aufgabe 2

Wir betrachten ein Negativbeispiel für eine Hashfunktion, die auf einem String s der Länge n aus Charactern $(s_1, \ldots, s_n)$ arbeitet:

$$h(s) := \sum_{i=1}^{n} \text{Anzahl der Einsen in der Binärdarstellung von } s_i.$$

Nehmen Sie an, dass jedes der 256 Zeichen in dem Text gleichwahrscheinlich vorkommt. Begründen Sie, warum Sie die Hashfunktion nicht für geeignet halten.

Lösungsvorschlag

Ein ASCII-Zeichen besteht bekanntlich aus 8 Bit. Der größte Wert unserer Hashfunktion ist also $8n$. Die Hashfunktion bildet auf die Schlüssel $0, \ldots, 8n$ ab.
Betrachten wir nun die Anzahl der Elemente, die auf den Schlüssel k abbilden, so sind dies gerade $\binom{8n}{k}$ Elemente (Anzahl der Möglichkeiten aus $8n$ Stellen k Einsen zu ziehen). Dies heißt also dass die Elemente sehr ungleich auf die Positionen des Arrays verteilt werden. In der Mitte des Arrays sind sehr viele Kollisionen zu erwarten, während am Rand fast keine Kollisionen auftreten werden.
Tatsächlich ist eine gute Hashfunktion eine Funktion, die die Elemente sehr gleichmäßig auf die Array-Positionen verteilt.

Aufgabe 3

Ein Hotelmanager hat n Buchungen für die nächste Saison. Sein Hotel hat k identische Räume. Die Buchungen enthalten ein Ankunfts- und ein Abreisedatum. Er will herausfinden, ob er zu allen Zeiten genügend Räume für die Buchungen zur Verfügung hat.

Entwickeln Sie einen Algorithmus, der dieses Problem in Zeit $\mathcal{O}(n + \text{sort}(n))$ löst.

Lösungsvorschlag

Für jede Buchung i bezeichne A_i das Anreisedatum und D_i das Abreisedatum.
Sei $S = \{A_i, (D_i - 1) \mid 1 \leq i \leq n\}$ die Menge aller Daten. (Durch A_i und $D_i - 1$ werden die „Nächte" beschrieben, zu denen die Zimmer benötigt werden.)

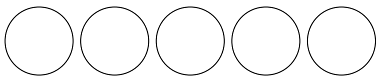
> $\langle s_1, \ldots, s_{2n}\rangle$:=*sort(S)*
>
> **int** $m = 0$
>
> **int** $i = 1$
>
> **while** $m \leq k$ **and** $i \leq 2n$ **do**
>
> > **if** s_i Anreisedatum **then** $m++$
> > **else** $m--$
> > $i++$
>
> **if** $i < 2n$ **then return** „zu wenig Zimmer zum Zeitpunkt s_{i-1}"
>
> **else return** „genug Zimmer"

Aufgabe 4

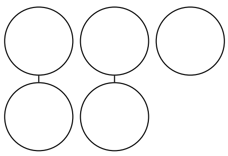
Geben Sie ein Verfahren an, um 5 Elemente mit 7 Vergleichen zu sortieren.

Lösungsvorschlag

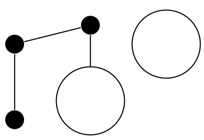
Wir beginnen mit 5 ungeordneten Elementen:



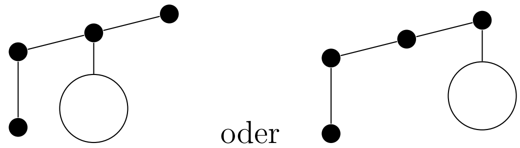
Mittels zwei Vergleichen stellen wir eine Ordnung unter jeweils zwei Elementen her:



Vergleichen wir die beiden größten Elemente so erhalten wir eine Ordnung unter drei Elementen:



Mit zwei weiteren Vergleichen können wir das bisher noch unverglichene Element sicher in die drei korrekt geordneten Elemente einfügen (Binäre Suche). Wir erhalten Eine der beiden Situationen:



In beiden Fällen können wir mit zwei Vergleichen das „weiße“ Element durch die noch zwei verbleibenden Vergleiche korrekt einsortieren.

Aufgabe 5

Wir haben uns in der Vorlesung beim Beweis, dass die Laufzeit von MergeSort in $\mathcal{O}(n \log n)$ liegt, auf das Mastertheorem berufen.
Zeigen Sie ohne Benutzung des Mastertheorems, dass die geschlossende Darstellung der Rekursionsformel

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) \\ T(1) &= \Theta(1) \end{aligned}$$

in $\mathcal{O}(n \log n)$ liegt. Sie dürfen vereinfachend annehemen, dass n eine Zweierpotenz ist.

Lösungsvorschlag

Wie in der Aufgabe angegeben, gehen wir davon aus, dass n von der Form 2^m ist. Wir berechnen die Laufzeit von $T(n)$ durch widerholtes anwenden der Definition.

$$T(n) = cn + 2T(\frac{n}{2}) = cn + 2\left(c\frac{n}{2} + 2T(\frac{n}{4})\right) = cn + cn + 4T(\frac{n}{4}) =$$
$$cn + cn + \cdots + cn + 2^m T(\frac{n}{2^m}) = cn + cn + \cdots + cn + 2^m T(1) =$$
$$cn + cn + \cdots + cn + nc = cn(\log n + 1) \in \mathcal{O}(n \log n)$$

Die letzte Zeile folgt, da $m = \log n$.

Aufgabe 6

Wir betrachten die Anzahl der Blocktransfers die beim externen Sortieren auftreten. Diese ist in der Vorlesung mit $\mathcal{O}(\frac{n}{B} \log_{M/B} \frac{n}{M})$ hergeleitet worden.

- Zeigen Sie: Der Algorithmus für das externe Sortieren muss unter normalen Umständen (realistische Werte für M und B) für noch nicht einmal jedes Element eine Lese- oder Schreiboperation ausführen.

  Gehen Sie dabei wie folgt vor: Gehen Sie von der asymptotischen Anzahl der Blocktransfers aus und geben Sie eine Bedingung für n an, so dass mindestens n Lese- oder Schreiboperation auf dem Hauptspeicher ausgeführt werden müssen. Argumentieren Sie dann warum diese Bedingung meist nicht zu erfüllen ist, oder wie M und B zu wählen ist, damit diese Bedingung realistisch erfüllbar ist.

- Man kann für externes Sortieren eine untere Schranke von $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ benötigten Blocktransfers zeigen. Ist damit der in der in der Vorlesung gezeigte Algorithmus asymptotisch optimal?

Lösungsvorschlag

- Wir berechnen ein Kriterium, so dass $\frac{n}{B}\log_{M/B}\frac{n}{M} \geq n$:

$$\begin{aligned} & \frac{n}{B}\log_{M/B}\frac{n}{M} \geq n \\ \Leftrightarrow\ & \log_{M/B}\frac{n}{M} \geq B \\ \Leftrightarrow\ & \frac{n}{M} \geq \left(\frac{M}{B}\right)^B \\ \Leftrightarrow\ & n \geq M\left(\frac{M}{B}\right)^B . \end{aligned}$$

  Gehen wir nun davon aus, dass die Blockgröße 100 Zahlen beeinhalten kann und $\frac{M}{B} = 2$ sei, was sehr klein wäre. So müsste man schon 2^{100} Elemente sortieren (geschätze Anzahl der Atome im Universum).

- Die obere Schranke ist asymptotisch optimal, da für ein $0 < c < 1$

$$\begin{aligned} \frac{n}{B}\log_{M/B}\frac{n}{M} &= \frac{n}{B}(\log_{M/B}\frac{n}{M} + 0) = \frac{n}{B}(\log_{M/B}\frac{n}{M} + \log_{M/B}\frac{B}{B}) = \\ &\frac{n}{B}(\log_{M/B}\frac{nB}{BM}) = \frac{n}{B}(\log_{M/B}\frac{n}{B} + \log_{M/B}\frac{B}{M}) = \frac{n}{B}(\log_{M/B}\frac{n}{B} - 1) \leq \\ &\frac{n}{B}(c\log_{M/B}\frac{n}{B}) \end{aligned}$$

  gilt. Somit ist also $\frac{n}{B}\log_{M/B}\frac{n}{M} \in \mathcal{O}(\frac{n}{B}\log_{M/B}\frac{n}{M})$. Dies bedeutet, dass die obere Schranke höchstens so schnell wächst wie die untere Schranke. Langsamer wachsen kann sie aber auch nicht, da $\frac{n}{B}\log_{M/B}\frac{n}{B}$ sonst keine unteree Schranke wäre. Deshalb müßen Sie bis auf einen konstanten Faktor gleich sein.