

Kap. 5: Hashing

Professor Dr. Petra Mutzel
Lehrstuhl für Algorithm Engineering, LS11

16. VO

1. Juni 2006

Motivation

„Warum soll ich heute hier bleiben?“

Hashing macht Spaß!

„Und wenn nicht?“

Beliebte Klausuraufgaben!

2

Überblick

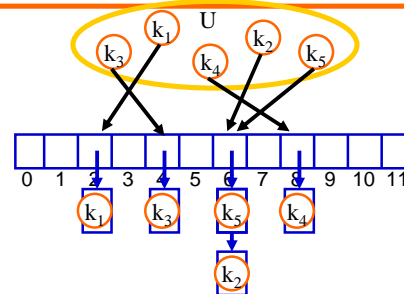
- Hashing mit Verkettung

- Hashing mit offener Adressierung
 - Lineares Sondieren
 - Quadratisches Sondieren
 - Double Hashing inkl. Brent

3

5.2 Hashing mit Verkettung

- **Idee:** Jedes Element der Hashtabelle ist ein Zeiger auf eine einfach verkettete lineare Liste:
Hashing mit Verkettung



4

Datenstruktur

Hashtabelle als Array $T[]$ mit Zeigern auf eine einfach verkettete Liste ohne Dummy-Elemente:

$T[i].key$: Schlüssel

$T[i].info$: Datenfeld

$T[i].next$: Zeiger auf das nachfolgende Listenelement

$hash(k)$: berechnet Hashfunktion für Schlüssel k

5

Realisierung Hashing mit Verkettung: INIT+SEARCH

INIT(T)

(1) **for** $i:=0, \dots, m-1$ **do** $T[i]:=nil$

SEARCH(k,T):

- $p:=T[hash(k)]$

(1) **while** $p \neq nil$ AND $p.key \neq k$ **do**

(2) $p:=p.next$

(3) **return** p

6

Realisierung Hashing mit Verkettung: INSERT

```

INSERT(p,T):
• var int pos
• pos:= hash(p.key)
• p.next:=T[pos]
• T[pos]:=p
  
```

7

Realisierung: DELETE

```

DELETE(k,T): // Schlüssel k ist in T enthalten
• var int pos
• var SListElement p,q
• pos:= hash(k)
• q:=nil, p:=T[pos]
(1) while p.key ≠ k do {
(2)   q:=p;
(3)   p:=p.next
(4) }
(5) if q==nil then T[pos]:=T[pos].next
(6) else q.next:=p.next
  
```

8

Analyse der Suchzeit

- Def.:** Sei $\alpha:=n/m$ der **Auslastungsfaktor** (Belegungsfaktor): durchschnittliche Anzahl von Elementen in den verketteten Listen.
- Worst Case:** Alle Schlüssel erhalten den gleichen Hashwert: $C_{\text{worst}}(n)=\Theta(n)$

9

Average-Case Analyse der Suchzeit

Annahmen:

- Ein Element wird auf jeden der m Plätze mit gleicher Wahrscheinlichkeit $1/m$ abgebildet, unabhängig von den anderen Elementen.
- Jeder der n gespeicherten Schlüssel ist mit gleicher Wahrscheinlichkeit der Gesuchte.
- INSERT fügt neue Elemente am Ende der Liste ein (in Wirklichkeit: am Anfang: aber egal für durchschnittliche Suchzeit)

(1) Berechnung der Hashfunktion $h(k)$ benötigt konstante Zeit.

10

Average-Case Analyse der Suchzeit

- Erfolgreiche Suche:** offensichtlich $C_{\text{avg}}(n)=\alpha=n/m$
- Wenn $n=O(m) \rightarrow$ konstante Suchzeit!

11

Average-Case Analyse der Suchzeit

Erfolgreiche Suche:

- Gemäß Ann. (3) wird 1 Schritt mehr gemacht als beim Einfügen des gesuchten Elements.
- Durchschnittliche Listenlänge beim Einfügen des i -ten Elements ist $(i-1) / m$.

$$\begin{aligned}
C_{\text{avg}}(n) &= \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m}\right) \\
&= 1 + \frac{1}{nm} \sum_{i=1}^n (i-1) = 1 + \frac{1}{nm} \frac{n(n-1)}{2} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m}
\end{aligned}$$

- $C_{\text{avg}}(n)=\Theta(1+\alpha)$
- Wenn also $n=O(m) \rightarrow$ konstante Suchzeit

12

Average-Case Analyse

- **Operation DELETE:** identisch mit Suchzeit!
- **Operation INSERT:** geht immer in $O(1)$, da wir annehmen, dass die Hashfunktion in konstanter Zeit berechnet werden kann.

13

Diskussion



- Belegungsfaktor von mehr als 1 ist möglich
- Echte Entfernungen von Einträgen sind möglich
- eignet sich für Externspeichereinsatz (Hashtabelle im Internspeicher, Listen extern)

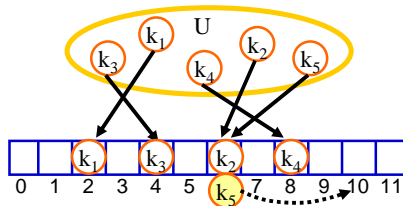


- Zu den Nutzdaten kommt der Speicherplatzbedarf für die Zeiger
- Der Speicherplatz der Tabelle wird nicht genutzt; dies ist umso schlimmer, wenn in der Tabelle viele Plätze leer bleiben.

14

5.3 Hashing mit offener Adressierung

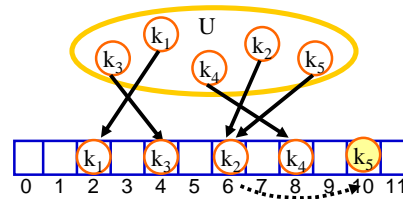
- **Idee:** Jedes Element wird direkt in der Hashtabelle gespeichert. Wenn ein Platz belegt ist, werden gemäß einer Sondierungsreihenfolge weitere Plätze ausprobiert.



15

Hashing mit offener Adressierung

- **Idee:** Jedes Element wird direkt in der Hashtabelle gespeichert. Wenn ein Platz belegt ist, werden gemäß einer Sondierungsreihenfolge weitere Plätze ausprobiert.



16

Hashing mit offener Adressierung

- Erweiterte Hashfunktion:
 $h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$
- Die Sondierungsreihenfolge ergibt sich dann durch: $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$
- Diese sollte idealerweise eine Permutation von $\langle 0, 1, \dots, m-1 \rangle$ sein.
- Annahme: Die Hashtabelle enthält immer wenigstens einen unbelegten Platz.

Problem: Entfernen von Elementen?

17

Eigenschaften

- (1) Stößt man bei der Suche auf einen unbelegten Platz, so kann die Suche erfolglos abgebrochen werden.
- (2) Wegen (1) wird nicht wirklich entfernt, sondern nur als entfernt markiert. Beim Einfügen wird ein solcher Platz als „frei“, beim Suchen als „wieder frei“ betrachtet.
- (3) Nachteil: wegen (2) ist die Suchzeit nicht mehr proportional zu $\Theta(1+\alpha)$, deshalb sollte man bei vielen Entfernungen die Methode der Verkettung der Überläufer vorziehen.

18

Sondierungsreihenfolgen

- Ideal wäre das „Uniform Hashing“: Jeder Schlüssel erhält mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $\{0,1,\dots,m-1\}$ als Sondierungsreihenfolge zugeordnet.
- In der Praxis versucht man möglichst nahe an „Uniform Hashing“ zu kommen.

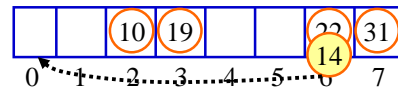
19

5.3.1 Lineares Sondieren

- Gegeben ist eine normale Hashfunktion $h': U \rightarrow \{0,1,\dots,m-1\}$
- Wir definieren für $i=0,1,\dots,m-1$: $h(k,i)=(h'(k)+i) \bmod m$

Beispiel: $m=8, h'(k)=k \bmod m$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0



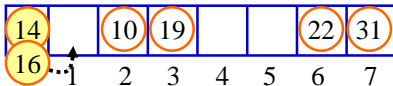
20

6.3.1 Lineares Sondieren

- $h(14,1)=(h'(k)+1) \bmod 8 = 6+1 \bmod 8 = 7$
- $h(14,2)=(h'(14)+2) \bmod 8 = 6+2 \bmod 8 = 0$
- Wir definieren für $i=0,1,\dots,m-1$: $h(k,i)=(h'(k)+i) \bmod m$

Beispiel: $m=8, h'(k)=k \bmod m$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0



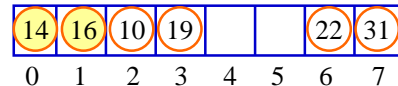
21

6.3.1 Lineares Sondieren

- $h(16,1)=(h'(k)+1) \bmod 8 = 0+1 \bmod 8 = 1$
- Durchschnittliche Zeit für erfolgreiche Suche: $(1+1+1+1+3+2) / 6 = 9/6 = 1,5$

Beispiel: $m=8, h'(k)=k \bmod m$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0



22

Analyseergebnisse

- Für die Anzahl der Sondierungen im Durchschnitt gilt für $0 < \alpha = n/m < 1$ (ohne Beweis):
- Erfolgreiche Suche $C_{\text{avg}}(\alpha) \approx \frac{1}{2}(1+1/(1-\alpha)^2)$
- Erfolgreiche Suche $C_{\text{avg}}(\alpha) \approx \frac{1}{2}(1+1/(1-\alpha))$

23

Diskussion

- Lange belegte Teilstücke tendieren dazu, schneller zu wachsen als kurze. Dieser unangenehme Effekt wird **Primäre Häufungen** genannt.
- Es gibt nur m verschiedene Sondierungsfolgen, da die erste Position die gesamte Sequenz festlegt: $h'(k), h'(k)+1, \dots, m-1, 0, 1, \dots, h'(k)-1$

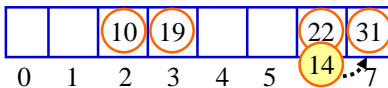
24

5.3.2 Quadratisches Sondieren

- Gegeben ist eine normale Hashfunktion $h': U \rightarrow \{0,1,\dots,m-1\}$
- Wir definieren für $i=0,1,\dots,m-1$: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

Beispiel: $m=8, h'(k)=k \bmod m, c_1=c_2=1/2$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0



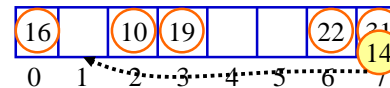
25

6.3.2 Quadratisches Sondieren

- $h(14,1)=6+1/2(1+1^2) \bmod 8 = 7$
- $h(14,2)=6+1/2(2+2^2) \bmod 8 = 9 \bmod 8 = 1$
- Wir definieren für $i=0,1,\dots,m-1$: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

Beispiel: $m=8, h'(k)=k \bmod m, c_1=c_2=1/2$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0



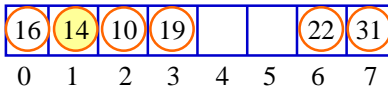
26

6.3.2 Quadratisches Sondieren

- $h(14,1)=6+1/2(1+1^2) \bmod 8 = 7$
- $h(14,2)=6+1/2(2+2^2) \bmod 8 = 9 \bmod 8 = 1$
- Durchschnittliche erfolgreiche Suchzeit $(1+1+1+1+3+1) / 6 = 8/6 = 1,33$

Beispiel: $m=8, h'(k)=k \bmod m, c_1=c_2=1/2$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0



27

Diskussion

- Es gibt nur m verschiedene Sondierungsfolgen, da die erste Position die gesamte Sequenz festlegt. Dieser Effekt wird **Sekundäre Häufungen** genannt.

28

Analyseergebnisse

- Für die Anzahl der Sondierungen im Durchschnitt gilt für $0 < \alpha = n/m < 1$ (ohne Beweis):
- Erfolgreiche Suche $C_{\text{avg}}(\alpha) \approx 1/(1-\alpha) - \alpha + \ln(1/(1-\alpha))$
- Erfolgreiche Suche $C_{\text{avg}}(\alpha) \approx 1 + \ln(1/(1-\alpha)) - \alpha/2$

Uniform Hashing:

- Erfolgreiche Suche $C_{\text{avg}}(\alpha) \approx 1/(1-\alpha)$
- Erfolgreiche Suche $C_{\text{avg}}(\alpha) \approx 1/\alpha \ln(1/(1-\alpha))$

30

Übersicht über die Güte der Kollisionsstrategien

- s. Folie

5.3.3 Double Hashing

- Die Sondierungsreihenfolge hängt von einer zweiten Hashfunktion ab, die unabhängig von der ersten Hashfunktion ist.

- Gegeben sind zwei Hashfunktionen $h_1, h_2: U \rightarrow \{0, 1, \dots, m-1\}$.
- Wir definieren für $i=0, 1, \dots, m-1$: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

31

Bedingungen an h_2

- Für alle Schlüssel k muss $h_2(k)$ relativ prim zu m sein:
 $\text{ggT}(h_2(k), m) = 1$
- Denn sonst wird die Tabelle nicht vollständig durchsucht:
- Für $\text{ggT}(h_2(k), m) = d > 1$, so wird nur $1/d$ -tel durchsucht.

32

Bedingungen an h_2

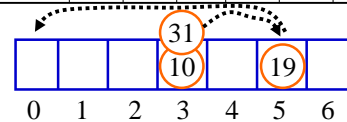
- Zwei Vorschläge:
- $m=2^p$, $p>1$, $h_2(k)$ immer ungerade
- m Primzahl, $0 < h_2(k) < m$, z.B.
 $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$
mit $m' = m-1$ oder $m' = m-2$

33

Beispiel für Double Hashing

- $m=7$, $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 5)$
- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
- $h_2(31) = 1 + (31 \bmod 5) = 2$; $h(31, 1) = 3 + 2 \bmod 7 = 5$

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2

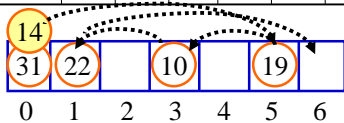


34

Beispiel für Double Hashing

- $m=7$, $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 5)$
- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
- $h(14, 1) = 0 + 5 \bmod 7 = 5$; $h(14, 2) = 10 \bmod 7 = 3$;
- $h(14, 3) = 15 \bmod 7 = 1$; $h(14, 4) = 20 \bmod 7 = 6$

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2

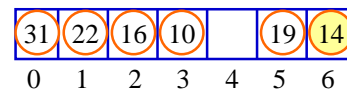


35

Beispiel für Double Hashing

- Durchschnittliche Suchzeit hier:
 $(1+1+3+1+5+1)/6 = 12/6 = 2$ (untypisch)
- $h(14, 1) = 0 + 5 \bmod 7 = 5$; $h(14, 2) = 10 \bmod 7 = 3$;
- $h(14, 3) = 15 \bmod 7 = 1$; $h(14, 4) = 20 \bmod 7 = 6$

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2



36

Diskussion



- Bei Double Hashing ergeben sich $\Theta(m^2)$ verschiedene Sondierungsreihenfolgen
- Es ist eine gute Approximation an uniformes Hashing.
- Double Hashing ist sehr gut in der Praxis!
- und lässt sich sehr leicht implementieren.

37

Verbesserung nach Brent [1973]

Wenn häufiger gesucht als eingefügt wird, ist es von Vorteil, die Schlüssel beim Einfügen so zu reorganisieren, dass die Suchzeit verkürzt wird.

- **Idee:** Wenn beim Einfügen eines Schlüssels ein Sondierter Platz j belegt ist mit $k' = T[j].key$, dann setze:

$$j_1 = j + h_2(k) \bmod m, j_2 = j + h_2(k') \bmod m$$

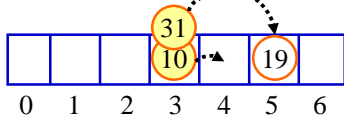
Ist Platz j_1 frei oder Platz j_2 belegt, fahre fort mit Double Hashing. Sonst: trage k' in $T[j_2]$ ein und k in $T[j]$

38

Beispiel mit Brent

- $m=7, h_1(k)=k \bmod 7, h_2(k)=1+(k \bmod 5)$
- $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$
- $j=3: j_1=5; j_2=j+h_2(10)=4 \rightarrow j_1$ ist besetzt, j_2 ist frei

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2

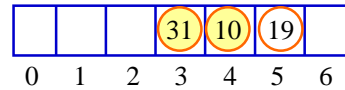


39

Beispiel mit Brent

- $m=7, h_1(k)=k \bmod 7, h_2(k)=1+(k \bmod 5)$
- $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$
- $j=3: j_1=5; j_2=j+h_2(10)=4 \rightarrow j_1$ ist besetzt, j_2 ist frei

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2

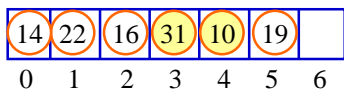


40

Beispiel mit Brent

- Durchschnittliche Suchzeit hier:
 $(2+1+1+1+1)/6=7/6=1,17$
- $j=3: j_1=5; j_2=j+h_2(10)=4 \rightarrow j_1$ ist besetzt, j_2 ist frei

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2



41

Analyseergebnisse

- Für die Anzahl der Sondierungen im Durchschnitt gilt für $0 < \alpha = n/m < 1$ (ohne Beweis):
- Erfolgreiche Suche $C_{avg}(\alpha) \approx 1/(1-\alpha)$ (wie uniform)
- Erfolgreiche Suche $C_{avg}(\alpha) < 2.5$ (unabhängig von α für $\alpha \leq 1$)

Double Hashing mit Brent ist also sehr gut!

42

Beispiel zum Ausschneiden und selbst nachspielen: $m=8$, $h(k)=k \bmod m$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

--	--	--	--	--	--	--	--

0 1 2 3 4 5 6 7

10 19 14 22 31 16