Kap. 5 Hashing

Professor Dr. Petra Mutzel
Lehrstuhl für Algorithm Engineering, LS11
Fakultät für Informatik, TU Dortmund

nach Übungstest

15./16. VO DAP2 SS 2009 16./18. Juni 2009

Linux-Kurs 2. Teil

- **Beginn:** Di **16.6**. und Do **18.6.**
- **Zeiten:** 14:15 Uhr oder Do 16:15 (2 SWS)
- **Ort:** GB V R. 014/015
- **Themen:** Backup & Recovery, Partitionieren, Bootmanager, ...
- **Veranstalter:** Wilfried Rupflin und Sven Jörges
- **Informationen:** http://dokserver.cs.uni-dortmund.de/linux09s/

Motivation

„Warum soll ich heute hier bleiben?"
Hashing macht Spaß!

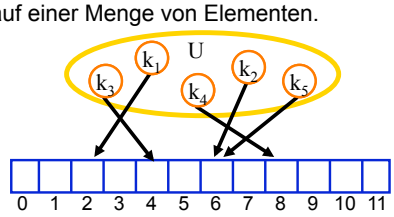
„Und wenn nicht?"
Beliebte Klausuraufgaben!

Überblick

- Wahl der Hashfunktion
- Hashing mit Verkettung
- Hashing mit offener Adressierung
  - Lineares Sondieren
  - Quadratisches Sondieren
  - Double Hashing inkl. Brent
- Vergleich der Hash-Verfahren

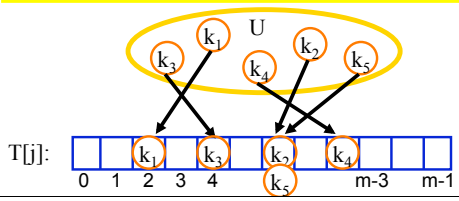
Idee von Hashing

- **Idee:** Ermittle die Position eines Elements durch eine arithmetische Berechnung statt durch Schlüsselvergleiche.
- Hashverfahren unterstützen die Dictionary-Operationen **Suchen, Einfügen** und **Entfernen** auf einer Menge von Elementen.



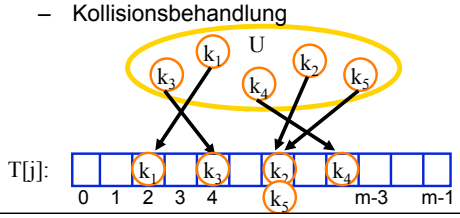
Idee von Hashing

- Sei U das Universum aus dem die Schlüssel kommen können, sei m die Tabellengröße, n die Anzahl der einzufügenden Schlüssel. Wir wählen eine Hashfunktion $h: U \to \{0,\ldots,m\text{-}1\}$.
- Die Schlüssel werden in Hashtabelle $T[h(i)]$ gespeichert.



Idee von Hashing

- Hashtabelle wird als ein Array T[0],…,T[m-1] der Größe m realisiert. Sie speichert die Einträge mit Hashadressen 0,…,m-1.
- Probleme:
  - Was sind gute Hashfunktionen?
  - Kollisionsbehandlung



5.1 Zur Wahl der Hashfunktion

- Ziel: Hashadressen sollten möglichst gleich verteilt in {0,…,m-1} sein.
- Hashfunktionen sollten Häufungen fast gleicher Schlüssel möglichst gleichmäßig auf den Adressbereich streuen.
- Generalannahme: Schlüssel sind nicht-negative ganze Zahlen.
- Lösung für character strings: deren ASCII-Code ist Nummer in [0,…,127],
- z.B. p≅112, t≅116 → pt ≅112·128+116=14452
- Allgemein: für String $(s_1,\ldots,s_l)$: $k=\sum_{1\le i\le l} 128^{l-i}\ \mathrm{ord}(s_i)$

5.1.1 Divisions-Rest Methode

- Die Hashfunktion der Divisions-Rest Methode ist gegeben durch: $h(k) = k \bmod m$

Eigenschaften:

- sehr schnelle Berechnung der Hashfunktion
- gute Wahl von m (Tabellengröße) ist hier sehr wichtig! Vermeide z.B.
  - $m=2^i$: ignoriert alle bis auf letzten Binärziffern
  - $m=10^i$: analog bei Dezimalzahlen
  - $m=r^i$: analog zu r-adischen Zahlen
  - $m=r^i \mp j$ für kleines j: Problem bei Vertauschung, denn Diff=a·128+b-(b·128+a) =(a-b)·(127∓j)-(a-b)

5.1.1 Divisions-Rest Methode

Z.B.: $m=2^8-1=127$:
pt = (112·128+116) mod 127 = 14452 mod 127 = 101
tp = (116·128+112) mod 127 = 14960 mod 127 = 101

Eigenschaften:

- sehr schnelle Berechnung der Hashfunktion
- gute Wahl von m (Tabellengröße) ist hier sehr wichtig! Vermeide z.B.
  - $m=2^i$: ignoriert alle bis auf letzten Binärziffern
  - $m=10^i$: analog bei Dezimalzahlen
  - $m=r^i$: analog zu r-adischen Zahlen
  - $m=r^i \mp j$ für kleines j: Problem bei Vertauschung, denn Diff=a·128+b-(b·128+a) =(a-b)·(127∓j)-(a-b)

Gute Wahl von m

- Primzahl, die
- kein $r^i \mp j$ teilt für kleines j und
- weit weg von einer Zweierpotenz ist.

Beispiel:

- Hashtabelle für ca. 700 Einträge für character strings (interpretiert als 28-adische Zahlen).
- Gute Wahl z.B. $m=701$, da $2^9=512$ und $2^{10}=1024$.

5.1.2 Multiplikationsmethode

- Die Hashfunktion der Multiplikationsmethode ist gegeben durch:
  $h(k) = \lfloor m\ (k \cdot A \bmod 1) \rfloor = \lfloor m(k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$
  mit $0<A<1$.

Term $(k \cdot A - \lfloor k \cdot A \rfloor)$ heißt der gebrochene Teil von kA.

Eigenschaften:

- Wahl von m ist hierbei unkritisch
- Gleichmäßige Verteilung für $U=\{1,2,\ldots,n\}$ bei guter Wahl von A.

Gute Wahl für A

Irrationale Zahlen sind eine gute Wahl, denn:

Sei ξ eine irrationale Zahl. Platziert man die Punkte $\xi - \lfloor \xi \rfloor$, $2\xi - \lfloor 2\xi \rfloor$, ..., $n\xi - \lfloor n\xi \rfloor$ in das Intervall $[0,1)$, dann haben die $n+1$ Intervallteile höchstens drei verschiedene Längen. Außerdem fällt der nächste Punkt $(n+1)\xi - \lfloor (n+1)\xi \rfloor$ in eines der größeren Intervallteile.

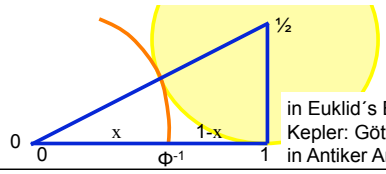
Beweis: Vera Turan Sós 1957

Beste Wahl für A nach Knuth: Der Goldene Schnitt

Der Goldene Schnitt

Sei $A=\Phi^{-1}=2/(1+\sqrt{5})=(\sqrt{5}-1)/2=0{,}6180339887\ldots$
Φ^{-1} ist bekannt als der goldene Schnitt.

Er ergibt sich durch die Zerlegung einer Strecke a in zwei positive Summanden x und $a-x$, so dass x geometrisches Mittel von a und $a-x$ ist, d.h. $x^2=a(a-x)$. Es gilt: $x/a=(a-x)/x$.



in Euklid´s Elementen & Kepler: Göttliche Teilung in Antiker Architektur

Multiplikationsmethode

Beispiel: Dezimalrechnung, $k=123456$, $m=10000$, $A=\Phi^{-1}$:

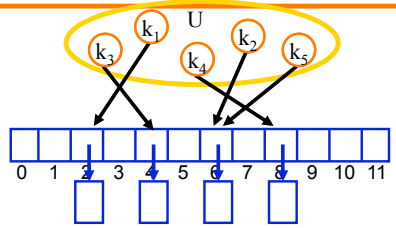
$$h(k) = \lfloor 10000\ (123456 \cdot 0{,}61803\ldots \bmod 1) \rfloor =$$
$$= \lfloor 10000\ (76300{,}0041151\ldots \bmod 1) \rfloor =$$
$$= \lfloor 10000 \cdot 0{,}0041151\ldots) \rfloor =$$
$$= \lfloor 41{,}151\ldots \rfloor = 41$$

Diskussion:

- Gute Wahl für m wäre hier $m=2^i$. Dann kann $h(k)$ effizient berechnet werden (eine einfache Multiplikation und ein bis zwei Shifts).
- Empirische Untersuchungen zeigen: Divisions-Rest-Methode ist besser.

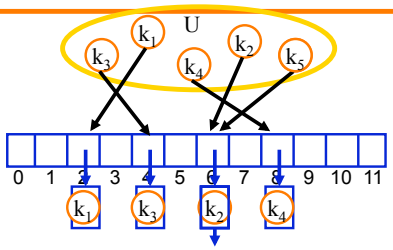
5.2 Hashing mit Verkettung

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



5.2 Hashing mit Verkettung

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



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

Realisierung Hashing mit Verkettung: INIT+SEARCH

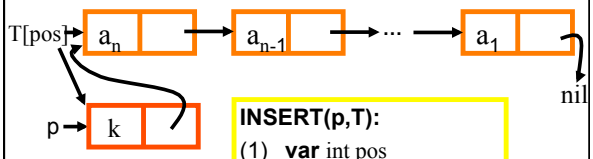
INIT(T)

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

SEARCH(k,T):

(1) p:=T[hash(k)]
(2) **while** p≠nil AND p.key ≠ k **do**
(3) p:=p.next
(4) return p

Realisierung Hashing mit Verkettung: INSERT



INSERT(p,T):

(1) **var** int pos
(2) pos:= hash(p.key)
(3) p.next:=T[pos]
(4) T[pos]:=p

Realisierung: DELETE

DELETE(k,T): // Schlüssel k ist in T enthalten

(1) **var** int pos
(2) **var** SListElement p,q
(3) pos:= hash(k)
(4) q:=nil, p:=T[pos]
(5) **while** p.key ≠ k **do** {
(6) q:=p
(7) p:=p.next
(8) }
(9) **if** q==nil **then** T[pos]:=T[pos].next // erstes El entfernt
(10) **else** q.next:=p.next

Analyse der Suchzeit

- **Def.:** Sei α:=n/m der **Auslastungsfaktor** (Belegungsfaktor): durchschnittliche Anzahl von Elementen in den verketteten Listen.

- **Worst Case:** Alle Schlüssel erhalten den gleichen Hashwert: $C_{worst}(n)=\Theta(n)$

Average-Case Analyse der Suchzeit

Annahmen:

(1) Ein Element wird auf jeden der m Plätze mit gleicher Wahrscheinlichkeit 1/m abgebildet, unabhängig von den anderen Elementen.
(2) Jeder der n gespeicherten Schlüssel ist mit gleicher Wahrscheinlichkeit der Gesuchte.
(3) INSERT fügt neue Elemente am Ende der Liste ein (in Wirklichkeit: am Anfang: aber egal für durchschnittliche Suchzeit)
(4) Berechnung der Hashfunktion h(k) benötigt konstante Zeit.

Average-Case Analyse der Suchzeit

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

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.

$$C_{\text{avg}}(n) = \frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m})$$
$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) = 1+\frac{1}{nm}\frac{n(n-1)}{2}$$
$$= 1+\frac{n}{2m}-\frac{1}{2m} = 1+\frac{\alpha}{2}-\frac{1}{2m}$$

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

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.

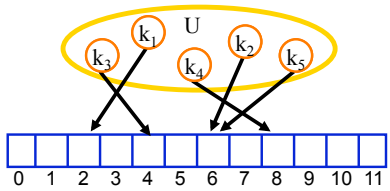
Diskussion

- 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.

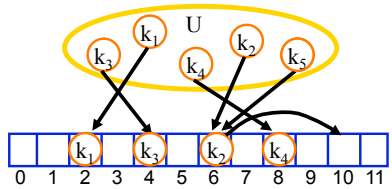
5.3 Hashing mit offener Adressierung

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



5.3 Hashing mit offener Adressierung

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



Hashing mit offener Adressierung

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

Problem: Entfernen von Elementen?

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.

Sondierungsreihenfolgen

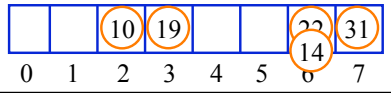
- Ideal wäre das „Uniform Hashing“: Jeder Schlüssel erhält mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $\{0,1,\ldots,m-1\}$ als Sondierungsreihenfolge zugeordnet.
- In der Praxis versucht man möglichst nahe an „Uniform Hashing“ zu kommen.

5.3.1 Lineares Sondieren

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

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

k	10	19	31	22	14	16
h´(k)	2	3	7	6	6	0

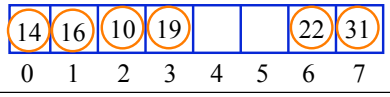


5.3.1 Lineares Sondieren

- $h(14,1)=(h'(k)+1) \bmod 8 = 6+1 \bmod 8 = 7$
- $h(14,2)=(h'(14)+2) \bmod 8 = 6+2 \bmod 8 = 0$
- 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



Analyseergebnisse

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

Diskussion

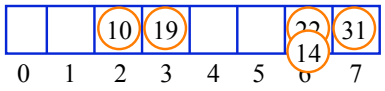
- Lange belegte Teilstücke tendieren dazu, schneller zu wachsen als kurze. Dieser unangenehme Effekt wird **Primäre Häufungen** genannt.
- Es gibt nur m verschiedene Sondierungsfolgen, da die erste Position die gesamte Sequenz festlegt: $h'(k), h'(k)+1,\ldots,m-1,0,1,\ldots,h'(k)-1$

5.3.2 Quadratisches Sondieren

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

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

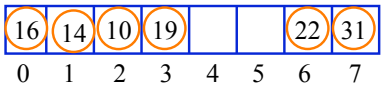


5.3.2 Quadratisches Sondieren

- $h(k,1)=6+1/2(1+1^2) \bmod 8 = 7$
- $h(k,2)=6+1/2(2+2^2) \bmod 8 = 9 \bmod 8 = 1$
- 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



Diskussion

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

Analyseergebnisse

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

Uniform Hashing:

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

5.3.3 Double Hashing

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

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

Bedingungen an h_2

- Für alle Schlüssel k muss $h_2(k)$ relativ prim zu m sein:
  $ggT(h_2(k),m) = 1$
- Denn sonst wird die Tabelle nicht vollständig durchsucht:
- Für $ggT(h_2(k),m) = d > 1$, so wird nur 1/d-tel durchsucht.
- Achtung: Natürlich muss $h_2(k) \neq 0$ sein

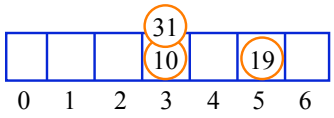
Zwei Vorschläge

- $m=2^P$, $P>1$, $h_2(k)$ 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 $m'=m-2$

Beispiel für Double Hashing

- $m=7$, $h_1(k)=k \bmod 7$, $h_2(k)=1+(k \bmod 5)$
- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

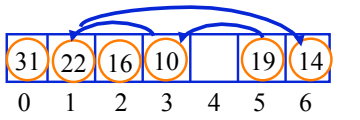
k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2



Beispiel für Double Hashing

- Durchschnittliche Suchzeit hier: (1+1+3+1+5+1)/6=12/6=2 (untypisch)
- $h_2(31)=1+(31 \bmod 5)=2$; $h(31,1)=3+2 \bmod 7=5$

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



Diskussion

- Bei Double Hashing ergeben sich $\Theta(m^2)$ verschiedene Sondierungsreihenfolgen
- Es ist eine gute Approximation an uniformes Hashing
- Analyse zeigt, dass erfolgreiche und erfolglose Suche ungefähr genauso hoch sind wie bei uniform Hashing
- Double Hashing ist sehr gut in der Praxis!
- und läßt sich sehr leicht implementieren

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:

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

(2) Falls Platz j_1 frei oder Platz j_2 belegt ist, **dann**

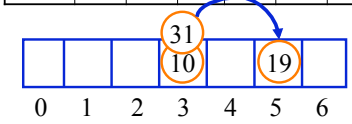
(3) fahre fort mit Double Hashing (mit k).

(4) Sonst: trage k' in $T[j_2]$ ein und k in $T[j]$

Beispiel mit Brent

- $m=7$, $h_1(k)=k \bmod 7$, $h_2(k)=1+(k \bmod 5)$
- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

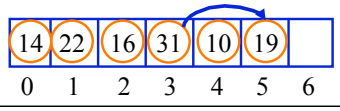
k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2



Beispiel mit Brent

- Durchschnittliche Suchzeit hier: (2+1+1+1+1+1)/6=7/6=1,17
- j=3: $j_2=j+h_2(10)=4$

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



Analyseergebnisse

- Für die Anzahl der Sondierungen im Durchschnitt gilt für $0<\alpha=n/m<1$ (ohne Beweis):
- Erfolglose 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!

5.4 Übersicht über die Güte der Kollisionsstrategien

Anzahl der Vergleiche

α	Verkettung		Hashing mit offener Adressierung					
	erfolgreich	erfolglos	lineares S.		quadr. Sond.		uniformes S.	
			e.reich	e.los	e.reich	e.los	e.reich	e.los
0.5	1.25	0.50	1.5	2.5	1.44	2.19	1.39	2.00
0.9	1.45	0.90	5.5	50.5	2.85	11.4	2.56	10
0.95	1.47	0.95	10.5	200	3.52	22.0	3.15	20
1.0	1.50	1.00	-----	-----	-----	-----	-----	-----

Double Hashing mit Brent: e.reich: <2.5 e.los: =uniformes S.

Bemerkungen

- Hashing mit Verkettung hat deutlich weniger Schritte; jedoch Nachteil: hoher Speicherbedarf und evtl. Suchzeit trotzdem langsamer, wegen Zeigerverfolgung
- quadratisches Hashing ist nahe an uniform Hashing
- lineare Sondierung fällt deutlich ab ➔ nicht empfehlenswert
- Belegungsfaktor α sollte nicht höher als 0.9 sein, besser:0.7 oder 0.8

ENDE Hashing

Beispiel zum Ausschneiden und Üben:

m=8, h(k)=k mod m

k	10	19	31	22	14	16
h´(k)	2	3	7	6	6	0