Algorithmen und Datenstrukturen

Hashverfahren

Das Wörterbuch-Problem

Das Wörterbuch-Problem (WBP) kann wie folgt beschrieben werden:

Gegeben: Menge von Objekten (Daten) die über einen eindeutigen Schlüssel (ganze Zahl, String, …) identifizierbar sind.

Gesucht: Struktur zur Speicherung der Objektmenge, so dass mindestens die folgenden Operationen effizient ausführbar sind:

- Suchen (Wiederfinden, Zugreifen)
- Einfügen
- Entfernen

Das Wörterbuch-Problem

Folgende Bedingungen können die Wahl einer Lösung des WBP beeinflussen:

- **Ort** an dem die Daten gespeichert werden: Hauptspeicher, Platte, Band, WORM (Write Once Read Multiple)
- **Häufigkeit** der Operationen:
  - überwiegend Einfügen & Löschen (dynamisches Verhalten)
  - überwiegend Suchen (statisches Verhalten)
  - annähernd Gleichverteilung
  - unbekannt
- **Weitere** zu implementierende **Operationen**:
  - Durchlaufen der Menge in bestimmter Reihenfolge
  - Mengen-Operationen: Vereinigung, Durchschnitt, Differenz, …
  - Aufspalten
  - Konstruieren
- **Ausführungsreihenfolge** der Operationen:
  - sequentiell
  - nebenläufig

Das Wörterbuch-Problem

Verschiedene Ansätze zur Lösung des WBP:

- Strukturierung der aktuellen Schlüsselmenge: Listen, Bäume, Graphen, …
- Aufteilung des gesamten Schlüssel-Universums: **Hashing**

Hashing (engl.: hackend/zerlegend) beschreibt eine spezielle Art der Speicherung der Elemente einer Menge durch **Zerlegung des Schlüssel-Universums**.

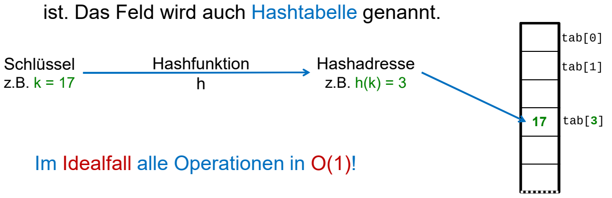
Idee von Hashing: Ermittle die Position eines Elements durch eine arithmetische Berechnung direkt aus dem Schlüssel statt durch Schlüsselvergleiche.

Hashing

- **Anwendung** von Hashing:
  - Verfahren für dynamisch veränderliche Menge von Objekten mit effizienten Grundoperationen: Suchen, Einfügen und Löschen
  - Suchen für statische Mengen.
  - Sicherheit: Verschleierung von Passwortdateien, Digitale Signaturen und Integritätsprüfung von Dateien.
- **Probleme** bei Implementierung durch schon bekannte Datenstrukturen:
  - Listen Suche ist aufwändig (O(n)) und vor jedem Löschen und Einfügen (an sich effizient) ist eine Suche notwendig.
  - Bäume: Suche ist bei balancierten Such-Bäumen effizient (O(log(n)), aber nach jedem Löschen und Einfügen ist Ausbalancieren notwendig.
  - Felder als Reihung: Suche schnell aber dynamisches Wachstum teuer, viel Speicher notwendig bei dünn besetzten Reihungen.

Hashing

- Mit einer Hashfunktion h wird aus dem Schlüssel k eine Hashadresse h(k) (positive ganze Zahl) berechnet.
- Die Hashadresse gibt den Index in einem Feld an, wo der Datensatz abgespeichert werden kann bzw. abgespeichert ist. Das Feld wird auch Hashtabelle genannt.



Im Idealfall alle Operationen in O(1)!

Hashtabelle `tab`

Informatik — Hochschule für angewandte Wissenschaften Hamburg

Hashing: Bewertungskriterien

- **Gleichverteilung/Zufälligkeit** (geringe Kollisionsgefahr): Ideal ist injektive Hashfunktion (verschiedene Objekte haben verschiedene Schlüssel). Dies benötigt jedoch meist zu viel Platz. Daher ist es nicht ausgeschlossen, dass zwei verschiedene Objekte den selben Schlüssel haben: Kollision (spezielle Überlaufverfahren erforderlich)
- **Geringer Speicherbedarf**: Ideal ist surjektive Hashfunktion (das ganze Feld abdecken), d.h. im Schnitt hoher Belegungsfaktor der Hashtabelle: (Anzahl Elemente)/(Anzahl der verfügbaren Speicherplätze) ist dicht an 1.
- Ähnliche Eingabewerte liefern völlig verschiedene Hashwerte (**gute Streuwirkung**), d.h. die Anzahl der Kollisionen ist gering, da die Hashfunktion dadurch Häufungen fast gleicher Schlüssel möglichst gleichmäßig auf den Adressbereich streut.
- **Effizienz** – Die Funktion muss schnell (und damit meist einfach) berechenbar sein

Informatik — Hochschule für angewandte Wissenschaften Hamburg

Hashing: Kollisionswahrscheinlichkeit

Zur Wahl der Hash-Funktion:

- Anforderungen hoher Belegungsfaktor und Kollisionsfreiheit stehen im Konflikt zueinander. Gesucht: geeigneter Kompromiss.
- Für die Schlüssel-Menge S mit $|S|=n$ und Feldplätzen $B_0, \ldots, B_{m-1}$ gilt:
  - für $n > m$ sind Kollisionen unausweichlich
  - für $n \leq m$ gibt es eine Wahrscheinlichkeit $P_K(n,m)$ für das Auftreten mindestens einer Kollision.

Abschätzung für $P_K(n,m)$:

- Bei Gleichverteilung (Laplace-Versuch): $P_K(n,m) = 1/m$, für beliebigen Schlüssel s, $j \in \{0, \ldots, m-1\}$ und (h ideal) $h(s)=j$.
- Es ist $P_K(n,m) = 1 - P_{\neg K}(n,m)$, wenn $P_{\neg K}(n,m)$ die Wahrscheinlichkeit dafür ist, dass es beim Speichern von n Elementen in m Feldplätzen zu keinen Kollisionen kommt.

Informatik — Hochschule für angewandte Wissenschaften Hamburg

Hashing: Kollisionswahrscheinlichkeit

Zur Wahrscheinlichkeit von Kollisionen

- Werden n Schlüssel nacheinander auf die Feldplätze $B_0, \ldots, B_{m-1}$ verteilt (bei Gleichverteilung), gilt jedes mal $P_K(n,m) = 1/m$.
- Die Wahrscheinlichkeit P(i) für keine Kollision im Schritt i ist $P(i) = (m - (i - 1))/m$
- Damit ist

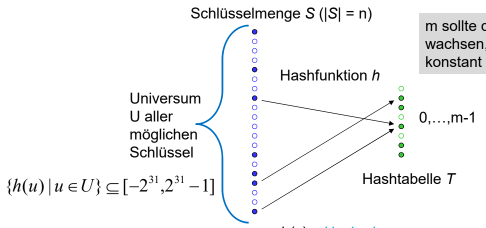
$$P_K(n,m) = 1 - \prod_{i=1}^{n} P(i) = 1 - \frac{\prod_{i=1}^{n}(m-(i-1))}{m^n}$$

Beispiel: Für m = 365 etwa ist $P(2) \approx 0{,}27\%$, $P(23) > 50\%$ und $P(50) \approx 97\%$ (Geburtstagsparadoxon)

$$P_K(n,365) = 1 - \prod_{i=1}^{n} P(i) = 1 - \frac{\prod_{i=1}^{n}(365-(i-1))}{365^n}$$

Hashfunktion

m sollte quadratisch mit n wachsen, um $P_K(n,m)$ konstant zu halten



h(s) = Hashadresse

h(s) = h(s') ⇔ s und s' sind Synonyme bzgl. h

Adresskollision

Hashfunktion: Divisions-(Rest-)Methode

(Einfache) Divisions-Rest-Methode

h(k) = k mod m

Beispiele zur Wahl von m:

a) m gerade: h(k) gerade ↔ k gerade
Ist problematisch, wenn das letzte Bit den Sachverhalt ausdrückt (z.B. 0 = weiblich, 1 = männlich)

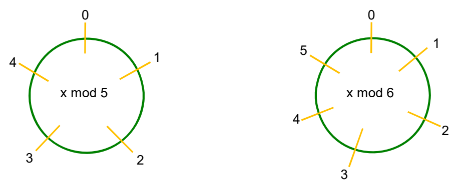
b) $m = 2^p$: liefert die p niedrigsten Bits von k; restlichen Bits gehen nicht in die Berechnung ein. Schnelle Berechnung durch Bit-shift-Operationen möglich.

c) m teilt $b^i \pm j$, i,j kleine nicht negative Zahlen, b Basis des Zahlensystems. Ähnlich problematisch, wie $m = 2^p$

d) Regel daher: Wähle m als Primzahl (die kein solches $b^i \pm j$ teilt und weit entfernt von einer Zweierpotenz ist).

Beispiel: Hashtabelle für ca. 700 Einträge für character strings (interpretiert als 28-adische Zahlen, ähnlich 28^i). Eine gute Wahl wäre z.B. m=701, da 2^9=512 und 2^{10}=1024 sowie 28^2 = 784.

Hashfunktion: Divisions-(Rest-)Methode



x+4: 0 4 8 12 16
x mod 5: 0 4 3 2 1

x+4: 0 4 8 12 16 20
x mod 6: 0 4 2 0 4 2

Hashfunktion: Divisions-(Rest-)Methode

Mit der Divisions-(Rest-)Methode können **Zeichenketten** als Hashkey verwendet werden, indem sie in ganze Zahlen zur Basis b umgewandelt werden, z.B. binär mit b=2.

Um Integer-Überläufe zu vermeiden, kann für die Berechnung des Hashwertes das **Horner-Schema** (Berechnung von Polynomen durch Ausklammern von X) eingesetzt werden.

Beispiel: Berechnung eines Hashwertes für eine 7-Bit ASCII-Zeichenkette s (ASCII- Zeichenkodierung definiert 128 Zeichen).

k mod m = (… (s_1*128 + s_2) mod m)*128 + s_3) mod m)*128 + … + s_l) mod m

Als Zwischenergebnis kann maximal (m – 1)*128 + 127 auftreten.

Pseudocode: i natürliche Zahl, h=0; s Zeichenkette / Feld

1. for i = 0 to i < länge_von(s)
2. h = (h * 128 + s[i]) mod m;

Ergebnis: h.

Die Multiplikation mit 128 = 2^7 entspricht einer Links-Bit-Shift-Operation << 7.

Hashfunktion: Multiplikative Methode

- Wähle (irrationale) Konstante θ, $0 < \theta < 1$, um eine willkürliche Zahl bzw. Zufallszahl zu simulieren.
- Berechne $(k*\theta) \bmod 1 = k*\theta - \lfloor k*\theta \rfloor$ (gebrochener Teil von $k*\theta$, d.h. der ganzzahlige Anteil wird abgeschnitten)
- Berechne $\boxed{h(k) = \lfloor m*((k*\theta) \bmod 1) \rfloor}$
- Wahl von m ist hier unkritisch (z.B. m = 2^p, dann kann h(k) effizient berechnet werden: eine einfache Multiplikation und ein bis zwei Bit-shift-Operationen), da der gebrochene Teil für eine gute Verteilung sorgt.
- Empirische Untersuchungen zeigen: Divisions-Rest-Methode ist besser als viele andere Hashfunktionen.

Hashfunktion: Multiplikative Methode

Irrationale Zahlen sind eine gute Wahl für θ, denn:

- Sei ξ eine irrationale Zahl. Platziert man die Punkte ξ - ⌊ξ⌋, 2ξ - ⌊2ξ⌋, …,nξ - ⌊nξ⌋ in das Intervall [0,1), dann haben die n+1 Intervallteile höchstens drei verschiedene Längen. Außerdem fällt der nächste Punkt (n+1)ξ -⌊(n+1)ξ⌋ in eines der größeren Intervallteile. [Vera Turan Sós 1957]
- Beste Wahl für θ nach Knuth: Der Goldene Schnitt

$$\boxed{\theta = \frac{\sqrt{5}-1}{2} \approx 0.6180339887498948482...}$$

$$h(123456) = \lfloor 10000*(123456*0.61803...\bmod 1) \rfloor = \lfloor 10000*(76300{,}0041151...\bmod 1) \rfloor = \lfloor 41.151... \rfloor = 41$$

Für m=10 bilden z.B. die Werte h(1), h(2), …, h(10) eine Permutation der Zahlen 0,1,…,9, nämlich 6, 2, 8, 4, 0, 7, 3, 9, 5, 1

Hashfunktion: Mittel-Quadrat-Methode

Die Zifferndarstellung der Schlüsselwerte werden als „simulierten Zufall“ verwendet. Das Berechnungsverfahren verwendet nur die Schlüsselwerte.

- Das Berechnungsverfahren: Quadrat, d.h. k^2.
- Die Zifferndarstellung sei

$$k^2 = s_{2r}s_{2r-1}\ldots s_1$$

$$h(k) = s_i s_{i-1}\ldots s_j \text{ für } 2r \geq i > j \geq 1$$

- Empfehlung für i,j: Ein mittlerer Block von Ziffern hängt von allen Ziffern in k ab. Dadurch werden aufeinanderfolgende Werte von k besser gestreut.

Hashfunktion: Vergleich

k	k mod m	k^2	$h_{Mittel-Quadrat}(k)$	$k \cdot \theta$	$h_{mult}(k)$
722	15	521284	28	446,2205398414	22
723	16	522729	72	446,8385738301	83
724	17	524176	17	447,4566078188	45

In diesem Beispiel sind

•Divisions-Rest-Methode: m = 101

•Mittel-Quadrat-Methode: zweite und dritte Ziffer von rechts

•multiplikative Methode: θ = 0.6180339887 und die ersten beiden Nachkommastellen

Kollisionsbehandlung

- Situation: zwei Schlüssel s und s´ werden durch die Hash-Funktion h auf die gleiche Feldadresse abgebildet (Adresskollision), d.h. h(s) = h(s‘), s und s‘ sind also Synonyme bzgl. h.
- Es sei s bereits eingetragen, dann muss der Überläufer s‘ anderswo so gespeichert werden, das er später leicht wiedergefunden werden kann.
- Grundsätzlich zwei Methoden:
- Verkettung (Geschlossenes Hashing, (direct/separate) Chaining, Hashing mit Verkettung): Hashtabelle als Feld von Zeigern auf eine einfach verkettete lineare Liste: die mehrfach Einträge (Überläufer) zu einem Index werden unter diesem Index als verkettete Liste abgespeichert
- offene Adressierung (Offenes Hashing): Jedes Element wird in der Hashtabelle gespeichert. Überläufer werden in noch freien anderen Feldplätzen abgespeichert. Diese werden beim Speichern und Suchen durch sogenanntes Sondieren (Re-Hashing, Probing) gefunden.

Verkettung

- Die Hashtabelle ist ein Feld (Länge m) von Listen. Jeder Index wird durch eine Liste realisiert.

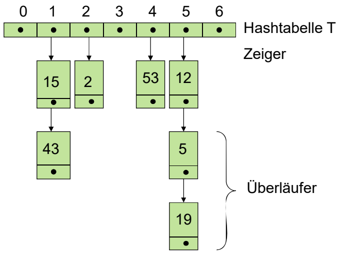
```
class hashTable {
Liste [] ht; // ein Listen-Array
hashTable (int m){ // Konstruktor
    ht = new Liste[m];
    for (int i = 0; i < m; i++)
        ht[i] = new Liste(); // Listen-Erzeugung
}    ...     }
```

- Zwei verschiedene Möglichkeiten der Listen-Anlage:
  1. Hashtabelle enthält nur Listen-Köpfe, Datensätze sind in Listen: Direkte Verkettung
  2. Hashtabelle enthält pro Index maximal einen Datensatz sowie einen Listen-Kopf. Überläufer kommen in die Liste: Separate Verkettung

Direkte Verkettung

Schlüssel werden in Überlauflisten gespeichert

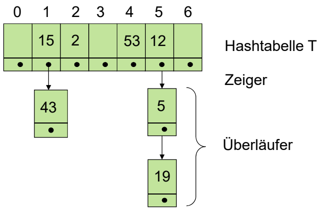
$h(k) = k \bmod 7$



Separate Verkettung

Schlüssel werden in Feld und Überlauflisten gespeichert

$h(k) = k \bmod 7$



Verkettung

Suchen nach Schlüssel k
- Berechne h(k) und Überlaufliste T[h(k)]
- Suche nach k in der Überlaufliste

Einfügen eines Schlüssels k
- (Erfolglose) Suche nach k
- Einfügen in die Überlaufliste

Entfernen eines Schlüssels k
- (Erfolgreiche) Suche nach k
- Entfernen aus Überlaufliste

→ Reine Listenoperationen

Wenn die Schlüssel eine Sortierung ermöglichen, kann In Abhängigkeit des Verhältnisses von *Anzahl Schlüssel n* zu der *Größe der Hashtabelle m* die Liste durch effizientere Strukturen ersetzt werden, wie etwa sortierte Listen oder balancierte Suchbäume.

Verkettung

Allgemeine Annahme beim Hashing:
- alle Hashadressen werden mit gleicher Wahrscheinlichkeit (Gleichverteilung, Laplace-Versuch) gewählt, d.h. $P_K(n,m) = 1/m$
- Mittlere Kettenlänge bei *n* Einträgen (und *m* Feldplätzen): $n/m = \alpha$
- Festlegung:
  C'_n = Erwartungswert für die Anzahl betrachteter Einträge bei erfolgloser Suche = α.
  C_n = Erwartungswert für die Anzahl betrachteter Einträge bei erfolgreicher Suche ≈ 1 + (α/2).

Effizienz der Suche:

Anzahl bei der Suche betrachteter Einträge	direkte Verkettung		separate Verkettung	
	erfolgreich	erfolglos	erfolgreich	erfolglos
α= 0.50	1.25	0.50	1.25	1.11
0.90	1.45	0.90	1.45	1.31
0.95	1.48	0.95	1.48	1.34
1.00	1.50	1.00	1.50	1.37

Verkettung

Vorteile:
+ C_n und C'_n (Erwartungswerte) und Varianz niedrig
+ Belegungsfaktor α > 1 möglich
+ echtes Löschen möglich: keine Belastung durch als gelöscht markierte Elemente
+ für Sekundärspeicher geeignet (Hashtabelle intern, externe Seiten verketten)

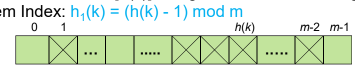
Nachteile:
- Zusätzlicher Speicherplatz für Zeiger
- Überläufer außerhalb der Hashtabelle (obwohl evtl. intern noch Platz wäre)

Analyse:
- worst case: h(s) liefert immer den gleichen Wert, alle Datensätze sind in einer Liste. Verhalten wie bei Linearer Liste.
- average case:Erfolgreiche Suche & Entfernen: Aufwand in Datenzugriffen ≈ 1 + (α/2).Erfolglose Suche & Einfügen: Aufwand ≈ α. Das gilt für direkte Verkettung, bei separater Verkettung ist der Aufwand jeweils etwas höher.
- best case: Die Suche hat sofort Erfolg. Aufwand konstant.

Offene Adressierung

Idee:
Unterbringung der Überläufer an freien („offenen“) Plätzen in der Hashtabelle. Falls T[h(k)] belegt ist, suche anderen Platz für k nach fester Regel.

Beispiel:
h(k) = k mod m. Falls T[h(k)] belegt ist, betrachte Eintrag mit nächst kleinerem Index: $h_1(k)$ = (h(k) - 1) mod m



Allgemeiner:
Betrachte die Folge $h_j(k)$ = (h(k) - j) mod m mit j = 0,…,m-1

Noch allgemeiner:
Betrachte Sondierungsfolge $h_j(k)$=(h(k) – s(j,k)) mod m mit j = 0,...,m-1, für eine gegebene (Sondierungs-)Funktion s(j,k)

Sondierungsfolgen

Beispiele für die Sondierungsfunktionen

$s(j,k) = j$ (lineares Sondieren)

$s(j,k) = \left\lceil \frac{j}{2} \right\rceil^2 * (-1)^j$ (quadratisches Sondieren)

$s(j,k) = j * h'(k)$ (Double Hashing)

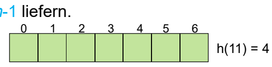
Eigenschaften von $s(j,k)$: Die Folge

$h_0(k)=(h(k) - s(0,k)) \bmod m,$
$h_1(k)=(h(k) - s(1,k)) \bmod m,$
…
$h_{m-2}(k)=(h(k) - s(m-2,k)) \bmod m,$
$h_{m-1}(k)=(h(k) - s(m-1,k)) \bmod m$

sollte eine Permutation von 0,...,m-1 liefern.

Kritisch:
Entfernen von Sätzen → als entfernt markieren

Beispiel: Quadratisches Sondieren
resultierende Folge: 4,5,3,1,0,5,2,4



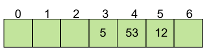
$s(j,k)$ = 0,-1,1,-4,4,-9,9,-16

Lineares Sondieren

$h_j(k)$=(h(k) – s(j,k)) mod m mit s(j,k) = j allg. s(j,k)=c*j

Sondierungsfolge für k: h(k), h(k)-1, h(k)-2, ..., 0, m-1,..., h(k)+1

Problem:
primäre Häufung („primary clustering“)
(lange Sequenzen gefüllter Feldplätze)



Lange Ketten werden mit größerer Wahrscheinlichkeit verlängert als kurze:
P(nächstes Objekt landet an Position 2) = 4/7
P(nächstes Objekt landet an Position 1, 0 oder 6) = 1/7

erfolgreiche Suche: $C_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$ erfolglose Suche: $C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

Effizienz des linearen Sondierens verschlechtert sich drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert.

α	C_n (erfolgreich)	C'_n(erfolglos)
0.50	1.5	2.5
0.90	5.5	50.5
0.95	10.5	200.5
1.00	-	-

Quadratisches Sondieren

$h_j(k)=(h(k) - s(j,k)) \bmod m$ mit $s(j,k) = (-1)^j * \left\lceil \frac{j}{2} \right\rceil^2$

Sondierungsfolge für k (mod m!): h(k)-0, h(k)-(-1), h(k)-1, h(k)-(-4), h(k)-4, ...
Permutation über 0, …,m-1, falls m eine Primzahl der Form 4i + 3 ist.

Problem: sekundäre Häufung („secondary clustering“), d.h. zwei Synonyme k und k´ durchlaufen stets dieselbe Sondierungsfolge: Bei linearem und quadratischen Sondieren ist s(j,k) unabhängig von k!

erfolgreiche Suche: $C_n \approx 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{(1-\alpha)}\right)$

erfolglose Suche: $C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{(1-\alpha)}\right)$

α	C_n (erfolgreich)	C'_n(erfolglos)
0.50	1.44	2.19
0.90	2.85	11.40
0.95	3.52	22.05
1.00	-	-

Uniformes Sondieren

$h_j(k)=(h(k) - s(j,k)) \bmod m$ mit $s(j,k) = \pi_k(j)$

wobei π_k eine („die k-te“) der m! Permutationen von {0,...,*m*-1} ist und $\pi_k(j)$ die j-te Zahl daraus beschreibt.

- hängt (nur) von k ab
- gleichwahrscheinlich für jede Permutation

erfolgreiche Suche: $C_n \approx \frac{1}{\alpha} * \ln\left(\frac{1}{(1-\alpha)}\right)$ erfolglose Suche: $C'_n \approx \frac{1}{1-\alpha}$

α	C_n (erfolgreich)	C'_n(erfolglos)
0.50	1.39	2
0.90	2.56	10
0.95	3.15	20
1.00	-	-

Realisierung von uniformem Sondieren ist praktisch sehr aufwendig.
Alternative: Zufälliges Sondieren mit s(j,k) = von *k* abhängige Zufallszahl.
s(j,k) = s(j´,k) ist hier möglich, aber unwahrscheinlich.

Double Hashing

$h_j(k)=(h(k) - s(j,k)) \bmod m$ mit $s(j,k) = j*h'(k)$

wobei h‘(k) eine weitere Hashfunktion ist.
Sondierungsfolge für k (mod m!): h(k), h(k)-h‘(k), h(k)-2h‘(k), ...

Forderung: Sondierungsfolge muss Permutation der Hashadressen entsprechen.
Folgerung: h‘(k) ≠ 0 und h‘(k) kein Teiler von m.

Beispiel: Sei h(k) = k mod m. Dann ist h‘(k) = 1 + (k mod (m-2)) eine gute Wahl, da
- h‘(k) von h(k) unabhängig ist
- h‘(k) ≠ 0 wegen 1+
- 1+(k mod (m-1)) wegen (m-1) gerade ungünstig ist

Double Hashing

Hashfunktionen: h(k) = k mod 7 und h´(k) = 1 + (k mod 5)

Schlüsselfolge: 15, 22, 1, 29, 26

0	1	2	3	4	5	6
	15					

$h(15) = 1$; $h'(15) = 1$; $h_0(15) = 1$

$h(22) = 1$; $h'(22) = 3$; $h_1(22) = -2 \bmod 7 = 5$

0	1	2	3	4	5	6
	15				22	

$h(1) = 1$; $h'(1) = 2$; $h_1(1) = -1 \bmod 7 = 6$

0	1	2	3	4	5	6
	15				22	1

$h(29) = 1$; $h'(29) = 5$; $h_1(29) = -4 \bmod 7 = 3$

0	1	2	3	4	5	6
	15		29		22	1

$h(26) = 5$; $h'(26) = 2$; $h_1(26) = 3$; $h_2(26) = 1$;
$h_3(26) = -1 \bmod 7 = 6$; $h_4(26) = -3 \bmod 7 = 4$

In diesem Beispiel genügt fast immer einfaches Sondieren ($h_1(k)$).

- Double Hashing ist genauso effizient wie uniformes Sondieren.
- Double Hashing ist leichter zu implementieren.

Lineares Sondieren

Hashfunktionen: h(k) = k mod 7

Schlüsselfolge: 15, 22, 1, 29, 26

0	1	2	3	4	5	6
	15					

$h(15) = 1$; $h_0(15) = 1$

$h(22) = 1$; $h_1(22) = 1\text{-}1 \bmod 7 = 0$

0	1	2	3	4	5	6
22	15					

$h(1) = 1$; $h_1(1) = 1\text{-}1 \bmod 7 = 0$
$h_2(1) = 1\text{-}2 \bmod 7 = 6$

0	1	2	3	4	5	6
22	15					1

$h(29) = 1$; $h_1(29) = 1\text{-}1 \bmod 7 = 0$
$h_2(29) = 1\text{-}2 \bmod 7 = 6$
$h_3(29) = 1\text{-}3 \bmod 7 = 5$

0	1	2	3	4	5	6
22	15				29	1

$h(26) = 5$; $h_1(26) = 5\text{-}1 \bmod 7 = 4$

0	1	2	3	4	5	6
22	15			26	29	1

Quadratisches Sondieren

Hashfunktionen: h(k) = k mod 7

Schlüsselfolge: 15, 22, 1, 29, 26

0	1	2	3	4	5	6
	15					

$h(15) = 1$; $h_0(15) = 1$

$h(22) = 1$; $h_1(22) = 1+1 \bmod 7 = 2$

0	1	2	3	4	5	6
	15	22				

$h(1) = 1$; $h_1(1) = 1+1 \bmod 7 = 2$
$h_2(1) = 1\text{-}1 \bmod 7 = 0$

0	1	2	3	4	5	6
1	15	22				

$h(29) = 1$; $h_1(29) = 1+1 \bmod 7 = 2$
$h_2(29) = 1\text{-}1 \bmod 7 = 0$
$h_3(29) = 1+4 \bmod 7 = 5$

0	1	2	3	4	5	6
1	15	22			29	

$h(26) = 5$; $h_1(26) = 5+1 \bmod 7 = 6$

0	1	2	3	4	5	6
1	15	22			29	26

Verbesserung der erfolgreichen Suche

Einfügen:

- k trifft in T[i] auf k_{alt}, d.h. i = h(k) - s(j,k) = h(k_{alt}) - s(j‘,k_{alt})
- k_{alt} bereits in T[i] gespeichert

Idee: Suche freien Platz für k oder k_{alt}

Zwei Möglichkeiten (ggf. nebenläufig berechnen):

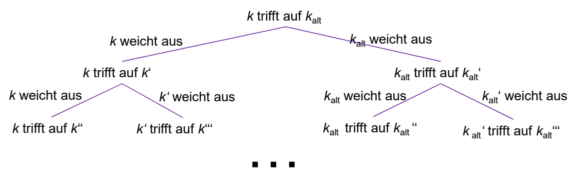
(M1) k_{alt} bleibt in T[i]; betrachte neue Position
h(k) - s(j+1,k) für k

(M2) k verdrängt k_{alt}; betrachte neue Position
h(k_{alt}) - s(j‘+1,k_{alt}) für k_{alt}

if (M1) or (M2) trifft auf einen freien Platz
then trage entsprechenden Schlüssel ein; fertig
else verfolge (M1) [und (M2)] weiter

Verbesserung der erfolgreichen Suche

Binärbaum Sondieren: verfolge (M1) und (M2)



$$C_n < 2{,}2$$

$$C'_n \approx \frac{1}{1-\alpha}$$

Verbesserung der erfolgreichen Suche

Hashtabelle mit m = 11, Double Hashing mit $h_j(k)=(h(k) – j*h‘(k))$ mod m wobei

h(k) = k mod 11 und h‘(k) = 1+(k mod 9)

bereits eingefügt: 22, 10, 37, 47, 17; noch einzufügen: 6 und 30:

0	1	2	3	4	5	6	7	8	9	10
22_0			47_0	37_0		17_0				10_0

h_0(6)=h(6)=6; h_1(6)=h(6)-h‘(6) =6-7 =-1 mod 11 = 10;
h_2(6)=h(6)-2h‘(6) = -8 mod 3 = 3; h_3(6)=h(6)-3h‘(6) = -15 mod 11 = 7;
h_0(17)=6; h_1(17)=h(17)-h‘(17)=6-9=-3 mod 11 = 8;

0	1	2	3	4	5	6	7	8	9	10
22_0			47_0	37_0		6_0		17_1		10_0

Verbesserung der erfolgreichen Suche

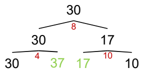
0	1	2	3	4	5	6	7	8	9	10
22_0			47_0	37_0		6_0		17_1		10_0

a) $h_0(30)=h(30)=8$; $h_1(30)=h(30)-h'(30)=8-4=4$; $h_2(30)=h(30)-2h'(30)=0$;
$h_3(30)=h(30)-3h'(30)=-4 \bmod 11=7$;

$h_0(17)=6$; $h_1(17)=8$; **b)** $h_2(17)=h(17)-2h'(17)=-12 \bmod 11=10$;
$h_3(17)=h(17)-3h'(17)=-21 \bmod 11=1$;

$h_0(37)=4$; **c)** $h_1(37)=h(37)-h'(37)=2 \bmod 11=2$;

$h_0(10)=10$; **d)** $h_1(10)=h(10)-h'(10)=8 \bmod 11=8$; $h_2(10)=h(10)-2h'(10)=6 \bmod 11=6$;
$h_3(10)=h(10)-3h'(10)=4 \bmod 11=4$; $h_4(10)=h(10)-4h'(10)=2 \bmod 11=2$;



$h_0(30)=8$; $h_3(17)=1$;

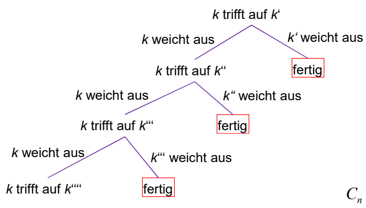
0	1	2	3	4	5	6	7	8	9	10
22_0	17_3		47_0	37_0		6_0		30_0		10_0

$h_1(30)=4$; $h_1(37)=2$;

0	1	2	3	4	5	6	7	8	9	10
22_0		37_1	47_0	30_1		6_0		17_1		10_0

Verbesserung der erfolgreichen Suche

Brent's Verfahren: verfolge nur (M1)



$$C_n < 2{,}5$$

$$C'_n \approx \frac{1}{1-\alpha}$$

Verbesserung der erfolgreichen Suche

Wurde eben die Sondierungsfolge von dem ersten k_{alt} verfolgt, wird nun stets mit dem neuen k_{alt} die Sondierungsreihenfolge betrachtet.

0	1	2	3	4	5	6	7	8	9	10
22_0			47_0	37_0		17_0				10_0

$h_0(6)=h(6)=6$; $h_1(6)=h(6)-h'(6)=6-7=-1 \bmod 11=10$; $h_2(6)=h(6)-2h'(6)=-8 \bmod 3=3$;
$h_3(6)=h(6)-3h'(6)=-15 \bmod 11=7$;

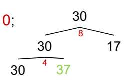
$h_0(17)=6$; $h_1(17)=h(17)-h'(17)=6-9=-3 \bmod 11=8$;

0	1	2	3	4	5	6	7	8	9	10
22_0			47_0	37_0		6_0		17_1		10_0

$h_0(30)=h(30)=8$; $h_1(30)=h(30)-h'(30)=8-4=4$; $h_2(30)=h(30)-2h'(30)=0$;
$h_3(30)=h(30)-3h'(30)=-4 \bmod 11=7$;

$h_0(17)=6$; $h_1(17)=8$; $h_2(17)=h(17)-2h'(17)=-12 \bmod 11=10$;

$h_0(37)=4$; $h_1(37)=h(37)-h'(37)=4-2 \bmod 11=2$;



$h_1(30)=4$; $h_1(37)=2$;

0	1	2	3	4	5	6	7	8	9	10
22_0		37_1	47_0	30_1		6_0		17_1		10_0

Verbesserung der erfolglosen Suche

Suche nach *k*:
falls $k' > k$ in Sondierungsfolge: → Suche erfolglos

Einfügen:
kleinere Schlüssel verdrängen größere Schlüssel

Invariante:
Alle Schlüssel in der Sondierungsfolge vor *k* sind kleiner als *k*
(aber nicht notwendigerweise aufsteigend sortiert im Feld!)

Probleme (bei allen Verdrängungsprozessen!):

- Verdrängungsprozess kann „Kettenreaktion" auslösen
- k' von *k* verdrängt: Position von k' in Sondierungsfolge?

→ Es muss für alle Schlüssel k gelten
$s(j,k) - s(j-1,k) = s(1,k)$, $1 \le j \le m$
d.h. die Schrittweite muss konstant sein. Oder
→ Die Sondierungsfolge wird am Schlüssel gespeichert

Lineares Sondieren mit Sortierung

Hashfunktionen: h(k) = k mod 7
Schlüsselfolge: 15, 22, 1, 29, 26

$h(15) = 1;\ h_0(15) = 1$

0	1	2	3	4	5	6
	15_0					

$h(22) = 1;\ h_1(22) = 1-1 \bmod 7 = 0$

0	1	2	3	4	5	6
22_1	15_0					

$h(1) = 1;\ h_1(15) = 1-1 \bmod 7 = 0$
$h_2(22) = 1-2 \bmod 7 = 6$

0	1	2	3	4	5	6
15_1	1_0					22_2

$h(29) = 1;\ h_1(29) = 1-1 \bmod 7 = 0$
$h_2(29) = 1-2 \bmod 7 = 6$
$h_3(29) = 1-3 \bmod 7 = 5$

0	1	2	3	4	5	6
15_1	1_0				29_3	22_2

$h(26) = 5;\ h_1(29) = 5-1 \bmod 7 = 4$
$h_4(29) = 1-4 \bmod 7 = 4$

0	1	2	3	4	5	6
22_1	15_0			29_4	26_0	1_2

Quadratisches Sondieren mit Sortierung

Hashfunktionen: h(k) = k mod 7
Schlüsselfolge: 15, 22, 1, 29, 26

$h(15) = 1;\ h_0(15) = 1$

0	1	2	3	4	5	6
	15_0					

$h(22) = 1;\ h_1(22) = 1+1 \bmod 7 = 2$

0	1	2	3	4	5	6
	15_0	22_1				

$h(1) = 1;\ h_1(15) = 1+1 \bmod 7 = 2$
$h_2(22) = 1-1 \bmod 7 = 0$

0	1	2	3	4	5	6
22_2	1_0	15_1				

$h(29) = 1;\ h_1(29) = 1+1 \bmod 7 = 2$
$h_2(29) = 1-1 \bmod 7 = 0$
$h_3(29) = 1+4 \bmod 7 = 5$

0	1	2	3	4	5	6
1_2	15_0	22_1			29_3	

$h(26) = 5;\ h_4(29) = 1+9 \bmod 7 = 3$

0	1	2	3	4	5	6
1_2	15_0	22_1	29_4		26_0	