Hashing

Idee: Die Adresse eines zu speichernden Mengenelementes ergibt sich aus dem Wert des Elementes.

Speicherplätze:

Menge von Behältern (engl. buckets) $B_0,..., B_{m-1}$

Gespeichert werden soll eine Menge von n Objekten, die jeweils repräsentiert sind durch einen Schlüssel aus einer Schlüsselmenge D.

Gewöhnlich gilt, dass $|D| >> m$.

3.2.2.1 Hashfunktionen

Eine **Hash-Funktion** (oder auch: Schlüsseltransformation) ist eine totale Funktion

$h: D \rightarrow \{0, ..., m-1\}$

Anforderungen an eine Hash-Funktion h

- h sollte surjektiv sein, d.h. alle Behälter erfassen.
- h sollte die zu speichernden Schlüssel gleichmäßig auf alle Behälter verteilen.
- h sollte einfach zu berechnen sein.

Beispiel

Aufgabe: Zu speichern sind die 12 Monatsnamen in 17 verfügbare Buckets.

Einfache Hashfunktion:

für einen Buchstaben c sei N(c) der Zahlenwert der Binärdarstellung von c.

Hier vereinfacht: N(c) = Position von c im Alphabet

N(a) = 1, N(b) = 2, ..., N(z) = 26

Abgeleitete Hash-Funktion:

$$h_0 (c_1,..., c_k) = \sum_{i=1}^{k} N(c_i) \textbf{ mod } 17.$$

Weitere Vereinfachung:

Betrachte nur die ersten drei Buchstaben

$$h_1 (c_1,..., c_k) = (N(c_1) + N(c_2) + N(c_3)) \textbf{ mod } 17$$

Verteilung der Monatsnamen auf die Buckets:

0	9
1	10
2	11
3	12
4	13
5	14
6	15
7	16
8	

Kollisionen sind Schlüssel, die auf denselben Behälter abgebildet werden.

Zwei prinzipielle Ansätze von Hashverfahren:

Offenes Hashing:

Jeder Behälter kann beliebig viele Elemente aufnehmen.

Realisierung:

Geschlossenes Hashing:

Jeder Behälter kann nur eine feste Anzahl b von Elementen aufnehmen.

In der Regel: $b = 1$

Für $h(d) = h(d')$ mit $d \neq d'$ erhalten wir eine Kollision.

3.2.2.3 Wahrscheinlichkeit von Kollisionen

Annahme: Es liege eine „ideale“ Hashfunktion vor,
d.h. h verteilt n Schlüssel völlig gleichmäßig auf m Buckets.
Es gelte $n < m$.

P_X bezeichne die Wahrscheinlichkeit für das Eintreten des Ereignis X.

Es gilt:

$$P_{\text{Kollision}} = 1 - P_{\text{keine Kollision}} .$$

$$P_{\text{keine Kollision}} = P(1) * P(2) * ... * P(n)$$

mit

P(i) = Wahrscheinlichkeit, dass der i-te Schlüssel auf einen freien Speicherblock abgebildet wird, falls alle vorhergehenden Schlüssel ebenfalls auf einen freien Block abgebildet wurden.

Es gilt: $P(1) = 1, \; P(2) = \frac{m-1}{m}$

$$P(i) = \frac{m-i+1}{m}$$

Gesamtwahrscheinlichkeit $P_{Kollision}$ ist dann:

$$P_{Kollision} = 1 - \frac{m*(m-1)*(m-2)*\ldots*(m-n+1)}{m^n}$$

Zahlenbeispiel

Sei m=365:

n	$P_{Kollision}$
22	0.475
23	0.507
50	0.970

Dieses Beispiel wird auch das ***„Geburtstagsparadoxon“*** genannt:

> Wenn 23 Personen in einem Raum zusammen sind, dann ist die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, bereits größer als 0.5.
>
> Ab 50 Personen ist dies schon fast mit Sicherheit der Fall.

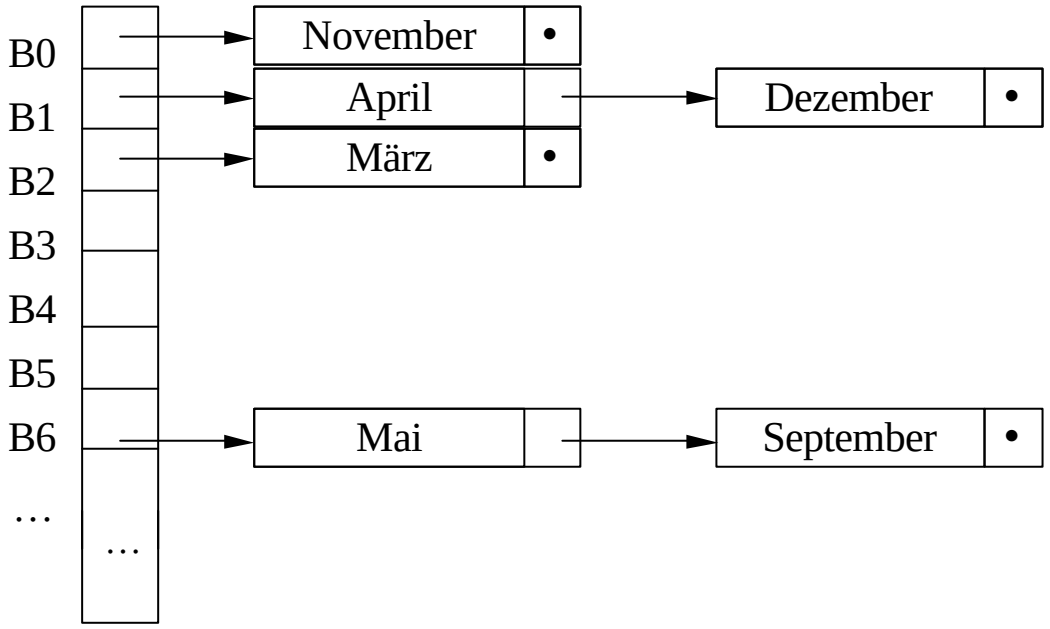
Konsequenz: **Kollisionen sind unvermeidlich !**

Offenes Hashing

Bucket = dynamische Liste von Schlüsseln

Menge der Buckets = Array von Zeigern auf die Listen

Grafisch:



Komplexitätsbetrachtungen

Annahme: gleichmäßige Verteilung der Schlüssel

- Die durchschnittliche Listenlänge ist $\frac{n}{m}$.

- insert, delete, member: $O(1 + \frac{n}{m}) = O(\frac{n}{m})$

  (d.h. falls $n \approx m$ ist $O(n/m) = O(1)$)

- worst case (alle Schlüssel sind in einer Liste, also in einem Bucket):

  delete, member: $O(n)$

- Platzbedarf: $O(n+m)$

Geschlossenes Hashing

Zahl der Einträge in die Hashtabelle ≤ m*b.

Im Folgenden: Spezialfall b=1

Jede Zelle der Hashtabelle kann also genau ein Element aufnehmen.

⇒ Darstellung der Hashtabelle als Array.

Bemerkung zu „weitere Einträge“:

Es muss darstellbar sein, dass

- Feldkomponente unbesetzt ist
- eine Feldkomponente besetzt war, aber inzwischen gelöscht wurde.

Gelingt dies nicht durch ausgezeichnete Elemente des Schlüsselbereiches, so muss die Element-Definition geeignet erweitert werden.

Methoden zur Kollisionsbehandlung

Idee: Eintragen eines Elementes x nach folgendem Schema:

platz = $hash_0(x)$; i = 1;
while (kollision (platz)) && frei(hashtable) **do**
{ i++; platz = $hash_i(x)$; }

Diese Vorgehensweise heißt **Rehashing**:

Erste Suche eines Speicherplatzes:

Hash-Funktion $h = h_0$

Bei Kollision:

weitere Hash-Funktionen $h_1, h_2, ..., h_{m-1}$

Die h_i sollen dabei (im Extremfall) alle m Speicherplätze durchlaufen.

Ein einfacher Ansatz für diese Menge von Hashfunktionen

Lineares Sondieren

Die Folgezellen von h(x) werden nacheinander untersucht, d.h.

$$h_i(x) = (h(x) + i) \textbf{ mod } m \qquad 1 \leq i \leq m-1$$

$$h_0(x) = h(x)$$

Beispiel Monatsnamen

(Einfügen in der „jährlichen“ Reihenfolge)

0	9
1	10
2	11
3	12
4	13
5	14
6	15
7	16
8	

Problem des Linearen Sondierens:

es tendiert zu **Kettenbildungen**:

bei häufigen Kollisionen werden im Array praktisch einfach verkettete Listen gebildet.

Begründung:

Durch neue Einträge können kurze Listen zu größeren zusammenwachsen.

Analyse des „idealen“ geschlossenen Hashing

Gesucht:

die erwarteten Kosten für das Einfügen eines Elementes, nachdem bereits n Elemente eingetragen wurden.

Diese Kosten hängen von der Anzahl der Zellen ab, die betrachtet werden müssen (d.h. die Anzahl der Proben), bis ein freies Feld gefunden wird.

Für ein Feld fester Größe m bezeichne

C_n = Erwartungswert für die Anzahl der betrachteten Einträge bei erfolgreicher Suche,

C'_n = Erwartungswert bei erfolgloser Suche.

Dem Einfügen geht immer eine erfolglose Suche voraus, dem Entfernen eine erfolgreiche Suche.

Kosten in Abhängigkeit vom Auslastungsfaktor (oder Belegungsgrad) $\alpha := \frac{n}{m}$

- Einfügen, erfolglose Suche: $\approx \frac{1}{1-a}$ $=: C_n'$

- Entfernen, erfolgreiche Suche: $\approx \frac{1}{a}\ln\frac{1}{1-a}$ $=: C_n$

Kosten für die Durchführung einer Operation:

Auslastung α	C_n'	C_n
20 %	1.25	1.28
50 %	2	1.38
80 %	5	2.01
90 %	10	2.55
95 %	20	3.15

Ohne Berücksichtigung der Kettenbildung!

Kollisionsstrategien

- **Verallgemeinertes Lineares Sondieren**

  $h_i(x) = (h(x) + c * i)$ **mod** m

  (c, m teilerfremd, $1 \leq i \leq m$)

  Keine Verbesserung gegenüber dem Linearen Sondieren; es entstehen Ketten mit Abstand c.

- **Quadratisches Sondieren**

  $h_i(x) = (h(x) + i^2)$ **mod** m

  Verfeinerung dieser Idee:

  $h_0(x) = h(x)$
  $h_1(x) = (h(x) + 1^2)$
  $h_2(x) = (h(x) - 1^2)$ **mod** m
  $h_3(x) = (h(x) + 2^2)$
  $h_4(x) = (h(x) - 2^2)$
  ....

  d.h

  $h_{2i-1}(x) = (h(x) + i^2)$ **mod** m
  $h_{2i}(x) = (h(x) - i^2)$ **mod** m

  $1 \leq i \leq \frac{m-1}{2}$

Beim quadratischen Sondieren wird die Wahrscheinlichkeit für die Bildung längerer Ketten herabgesetzt.

- **Doppel-Hashing**

  Beim Doppel-Hashing wählt man zwei **voneinander unabhängige** Hash-Funktionen.

  Annahme:

  h und h’ sind ideale Hashfunktionen, d.h. eine Kollision tritt nur mit der Wahrscheinlichkeit 1/m auf:

  $$P(h(x) = h(y)) = 1/m \quad \text{und}$$

  $$P(h'(x) = h'(y)) = 1/m$$

h und h’ sind voneinander unabhängig, falls gilt:

$$P(h(x) = h(y) \ \text{Ù} \ h'(x) = h'(y)) = \frac{1}{m^2} .$$

Definiere nun eine Folge von Hash-Funktionen durch:

$$\mathbf{h_i(x) = (h(x) + h'(x) * i^2) \bmod m}$$

Problem: Es ist nicht leicht, zwei beweisbar voneinander unabhängige Hash-Funktionen zu finden.

Zur Wahl von Hash-Funktionen

- Einfachste Methode: **Divisionsmethode**

  Seien die natürlichen Zahlen der Schlüsselbereich, dann wählt man

  $$h(x) = x \textbf{ mod } m.$$

Nachteil:

Aufeinander folgende Werte werden auf benachbarte Speicherplätze abgebildet.

Günstige Wahl für m: Wähle m als Primzahl p

(→ unterstützt Streuung der Hashfunktion).

Falls k Schlüssel einzutragen sind (k vorab bekannt)

Wähle $p > k$.

Eine gute Wahl für h ist dann:

$$h(x) = (a * x + b) \textbf{ mod } p,$$

mit $1 \leq a \leq p-1, 0 \leq b \leq p-1$

Zusammenfassung Hash-Funktionen

Komplexität:

- Hash-Verfahren haben schlechtes Worst-Case Verhalten.

  (O(n) für die drei Dictionary Operationen)

- Sie können aber ein gutes Durchschnittsverhalten zeigen.

  (O(1) für alle diese Operationen)

- Alle Hashverfahren sind relativ einfach zu implementieren.

Anwendungsbedingungen:

- Bei allgemeinen dynamischen Anwendungen sollte offenes Hashing verwendet werden.

- Geschlossenes Hashing nur, wenn
  - Gesamtzahl der Elemente von vornherein beschränkt ist
  - keine oder nur wenige Elemente entfernt werden müssen
  - ein Belegungsgrad von 80% nicht überschritten wird.

Nachteil:

- unmittelbare (d.h. effiziente) Ausgabe der eingetragenen Suchschlüssel ist nicht möglich