

# ADS: Algorithmen und Datenstrukturen

## Teil XI

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for  
Bioinformatics, **University of Leipzig**

12. Januar 2011

# Hashing

- Hash-Funktionen
  - Divisionsrest-Verfahren
  - Faltung
  - Mid-Square-Methode, ...
- Behandlung von Kollisionen
  - Verkettung der Überläufer
  - Offene Hash-Verfahren: lineares Sondieren, quadratisches Sondieren, ...
- Analyse des Hashing
- Hashing auf Externspeichern
  - Bucket-Adressierung mit separaten Überlauf-Buckets
  - Analyse
- Dynamische Hash-Verfahren
  - Erweiterbares Hashing
  - Lineares Hashing

# Hashing

Gibt es bessere Strukturen für direkte Suche für Haupt- und Externspeicher ???

- AVL-Baum:  $O(\log_2 n)$  Vergleiche
- B\*-Baum: E/A-Kosten  $O(\log_k(n))$ , vielfach 3 Zugriffe

Bisher:

- Suche über Schlüsselvergleich
- Allokation des Satzes als physischer Nachbar des “Vorgängers” oder beliebige Allokation und Verknüpfung durch Zeiger

Gestreute Speicherungsstrukturen / Hashing (Schlüsseltransformation, Adreßberechnungsverfahren, scatter-storage technique usw.)

- Berechnung der Satzadresse  $SA(i)$  aus Satzschlüssel  $K_i \rightarrow$  Schlüsseltransformation
- Speicherung des Satzes bei  $SA(i)$
- Ziele: schnelle direkte Suche + Gleichverteilung der Sätze (möglichst wenig Synonyme)

# Hashing

## Definition.

- $S$  sei Menge aller möglichen Schlüsselwerte eines Satztyps (Schlüsselraum).  $A = 0, 1, \dots, m - 1$  sei Intervall der ganzen Zahlen von 0 bis  $m - 1$  zur Adressierung eines Arrays bzw. einer Hash-Tabelle mit  $m$  Einträgen.
- Eine Hash-Funktion  $h : S \rightarrow A$  ordnet jedem Schlüssel  $s \in S$  des Satztyps eine Zahl  $h(s)$  aus  $A$  als Adresse in der Hash-Tabelle zu.

Idealfall: 1 Zugriff zur direkten Suche

Problem: Kollisionen  $\dots h(s) = h(s')$

# Perfektes Hashing: Direkte Adressierung

Idealfall (perfektes Hashing): keine Kollisionen

- $h$  ist eine injektive Funktion.
- Für jeden Schlüssel aus  $S$  muss Speicherplatz bereitgehalten werden, d. h., die Menge aller möglichen Schlüssel ist bekannt.

Parameter

- $l$  Schlüssellänge,  $b =$  Basis,  $m = \#$  Speicherplätze
- $n_p = \#S = b^l$  mögliche Schlüssel
- $n_a = \#K = \#$  vorhandene Schlüssel
- Wenn  $K$  bekannt ist und  $K$  fest bleibt, kann leicht eine injektive Abbildung  $h : K \rightarrow \{0, \dots, m - 1\}$   
z. B. wie folgt berechnet werden:
- Die Schlüssel in  $K$  werden lexikographisch geordnet und auf ihre Ordnungsnummern abgebildet oder
- Der Wert eines Schlüssels  $K_i$  oder eine einfache ordnungserhaltende Transformation dieses Wertes (Division/Multiplikation mit einer Konstanten) ergibt die Adresse:

$$A_i = h(K_i) = K_i$$

## Beispiel: Direkte Adressierung

- Beispiel: Schlüsselmenge  $\{00, \dots, 99\}$
- Eigenschaften
  - Statische Zuordnung des Speicherplatzes
  - Kosten für direkte Suche und Wartung ?
  - Reihenfolge beim sequentiellen Durchlauf ?
- Bestes Verfahren bei geeigneter Schlüsselmenge  $K$ , aber aktuelle Schlüsselmenge  $K$  ist oft nicht "dicht":
  - eine 9-stellige Sozialversicherungsnummer bei 105 Beschäftigten
  - Namen / Bezeichner als Schlüssel (Schlüssellänge  $k$ ): Faktor  $10^4$

# Allgemeines Hashing

## Annahmen

- Die Menge der möglichen Schlüssel ist meist sehr viel größer als die Menge der verfügbaren Speicheradressen
- $h$  ist nicht injektiv

## Definitionen

- Zwei Schlüssel  $K_i$ ,  $K_j$  *kollidieren* (bzgl. einer Hash-Funktion  $h$ ) genau dann wenn  $h(K_i) = h(K_j)$ .
- Tritt für  $K_i$  und  $K_j$  eine Kollision auf, so heißen diese Schlüssel *Synonyme*.
- Die Menge der Synonyme bezüglich einer Speicheradresse  $A_i$  heißt Kollisionsklasse.

$k$  Personen auf einer Party haben gleichverteilte und stochastisch unabhängige Geburtstage. Mit welcher Wahrscheinlichkeit  $p(n, k)$  haben mindestens 2 von  $k$  Personen am gleichen Tag ( $n = 365$ ) Geburtstag?

Die Wahrscheinlichkeit, dass keine Kollision auftritt, ist

$$q(n, k) = \frac{n}{n} \frac{n-1}{n} \frac{n-2}{n} \frac{n-3}{n} \dots \frac{n-k+1}{n} = \frac{(n-1)(n-2)\dots(n-k+1)}{n^{k-1}}$$

Es ist  $p(365, k) = 1 - q(365, k) > 0.5$  für  $k > 22$

(z.B. <http://www.mathematik.ch/anwendungenmath/wkeit/geburtstag/>)

**ALSO:** Behandlung von Kollisionen erforderlich !



# Hash-Verfahren: Einflußfaktoren I

Leistungsfähigkeit eines Hash-Verfahrens: Einflußgrößen und Parameter

- Hash-Funktion
- Datentyp des Schlüsselraumes: Integer, String, ...
- Verteilung der aktuell benutzten Schlüssel
- Belegungsgrad der Hash-Tabelle HT
- Anzahl der Sätze, die sich auf einer Adresse speichern lassen, ohne Kollision auszulösen (Bucket-Kapazität)
- Technik zur Kollisionsauflösung
- ggf. Reihenfolge der Speicherung der Sätze

# Einflußfaktoren II

## Belegungsfaktor der Hash-Tabelle

- Verhältnis von aktuell belegten zur gesamten Zahl an Speicherplätzen  $\beta = n_a/m$
- für  $\beta > 0.85$  erzeugen alle Hash-Funktionen viele Kollisionen und damit hohen Zusatzaufwand
- Hash-Tabelle ausreichend groß zu dimensionieren ( $m > n_a$ )

Für die Hash-Funktion  $h$  gelten folgende Forderungen:

- Sie soll sich einfach und effizient berechnen lassen (konstante Kosten)
- Sie soll eine möglichst gleichmäßige Belegung der Hash-Tabelle HT erzeugen, auch bei ungleich verteilten Schlüsseln
- Sie soll möglichst wenige Kollisionen verursachen

# Hash-Funktionen I: Divisions-Verfahren

1. Divisionsrest-Verfahren:  $h(K_i) = K_i \bmod q$ , ( $q \sim m$ )

Der entstehende Rest ergibt die relative Adresse in HT

*Beispiel:*

Die Funktion *nat* wandle Namen in natürliche Zahlen um:

$nat(\text{Name}) = ord(\text{1. Buchstabe von Name}) - ord('A')$

$h(\text{Name}) = nat(\text{Name}) \bmod q$

- Wichtigste Forderung an Divisor  $q$ :  
 $q = \text{Primzahl (größte Primzahl } \leq m)$
- Hash-Funktion muß etwaige Regelmäßigkeiten in Schlüsselverteilung eliminieren, damit nicht ständig die gleichen Plätze in HT getroffen werden
- Bei äquidistantem Abstand der Schlüssel  $K_0 + jK$ ,  $j = 0, 1, 2, \dots$  maximiert eine Primzahl die Distanz, nach der eine Kollision auftritt. Eine Kollision ergibt sich, wenn

$$K_0 = (K_0 + jK) \bmod q$$

d.h.  $jK = kq$ ,  $k = 1, 2, 3, \dots$

- Eine Primzahl  $q$  kann keine gemeinsamen Teiler mit  $K$  besitzen, die den Kollisionsabstand verkürzen würden

# Hash-Funktionen II: Faltung

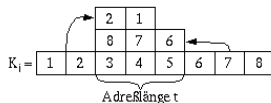
Schlüssel wird in Teile zerlegt, die bis auf das Letzte die Länge einer Adresse für HT besitzen

Schlüsselteile werden dann übereinandergefaltet und addiert.

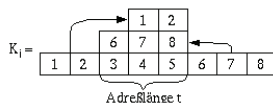
Variationen:

- Rand-Falten: wie beim Falten von Papier am Rand
- Shift-Falten: Teile des Schlüssels werden übereinandergeschoben
- Sonstige: z.B. XOR-Verknüpfung bei binärer Zeichendarstellung
- Beispiel:  $b = 10$ ,  $t = 3$ ,  $m = 10^3$

Rand-Falten



Shift Falten



## Faltung

- verkürzt lange Schlüssel auf "leicht berechenbare" Argumente, wobei alle Schlüsselteile Beitrag zur Adreßberechnung liefern
- diese Argumente können dann zur Verbesserung der Gleichverteilung mit einem weiteren Verfahren "gehasht" werden

# Hash-Funktionen III

## Mid-Square-Methode

- Schlüssel  $K_i$  wird quadriert.  $t$  aufeinanderfolgende Stellen werden aus der Mitte des Ergebnisses für die Adressierung ausgewählt.
- Es muss also  $b^t = m$  gelten.
- mittlere Stellen lassen beste Gleichverteilung der Werte erwarten
- Beispiel für  $b = 2$ ,  $t = 4$ ,  $m = 16$ :  $K_i = 1100100$

$$K_i^2 = 10011 \underbrace{1000}_{t} 1000 \quad \rightarrow \quad h(K_i) = 1000$$

# Hash-Funktionen IV

- **Zufallsmethode:**  
 $K_i$  dient als Saat für Zufallszahlengenerator
- **Ziffernanalyse:**  
setzt Kenntnis der Schlüsselmenge  $K$  voraus. Die  $t$  Stellen mit der besten Gleichverteilung der Ziffern oder Zeichen in  $K$  werden von  $K_i$  zur Adressierung ausgewählt
- **Problemangepaßte Methoden:**  
Unter Ausnutzung von speziellen Eigenschaften der Schlüssel wird versucht, eine möglichst günstige Hashfunktion zu konstruieren. Im günstigsten Fall erhält man so eine Hashfunktion, die injektiv und effektiv zu berechnen ist.

# Bewertung von Hash-Funktionen

Verhalten / Leistungsfähigkeit einer Hash-Funktion hängt von der gewählten Schlüsselmenge ab

- Deshalb lassen sie sich auch nur unzureichend theoretisch oder mit Hilfe von analytischen Modellen untersuchen
- Wenn eine Hash-Funktion gegeben ist, läßt sich immer eine Schlüsselmenge finden, bei der sie besonders viele Kollisionen erzeugt
- Keine Hash-Funktion ist immer besser als alle anderen

Über die Güte der verschiedenen Hash-Funktionen liegen jedoch eine Reihe von empirischen Untersuchungen vor

- Das Divisionsrest-Verfahren ist im Mittel am leistungsfähigsten; für bestimmte Schlüsselmenen können jedoch andere Techniken besser abschneiden
- Wenn die Schlüsselverteilung nicht bekannt ist, dann ist das Divisionsrest-Verfahren die bevorzugte Hash-Technik
- Wichtig dabei: ausreichend große Hash-Tabelle, Verwendung einer Primzahl als Divisor

# Behandlung von Kollisionen

Zwei Ansätze, wenn  $h(K_q) = h(K_p)$

- $K_p$  wird in einem separaten Überlaufbereich (außerhalb der Hash-Tabelle) zusammen mit allen anderen Überläufern gespeichert; Verkettung der Überläufer
- Es wird für  $K_p$  ein freier Platz innerhalb der Hash-Tabelle gesucht (“Sondieren”); alle Überläufer werden im Primärbereich untergebracht (“offene Hash-Verfahren”)

Methode der Kollisionsauflösung entscheidet darüber, welche Folge und wie viele relative Adressen zur Ermittlung eines freien Platzes aufgesucht werden

Adressfolge bei Speicherung und Suche für Schlüssel  $K_p$  sei  $h_0(K_p), h_1(K_p), h_2(K_p), \dots$

- Bei einer Folge der Länge  $n$  treten also  $n - 1$  Kollisionen auf
- Primärkollision:  $h(K_p) = h(K_q)$
- Sekundärkollision:  $h_i(K_p) = h_j(K_q), i \neq j$

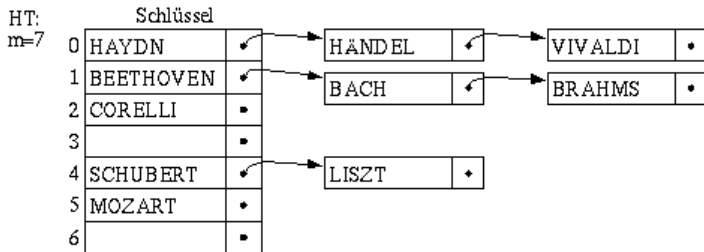


# Hash-Verfahren mit Verkettung der Überläufer (separater Überlaufbereich)

## Dynamische Speicherplatzbelegung für Synonyme

- Alle Sätze, die nicht auf ihrer Hausadresse unterkommen, werden in einem separaten Bereich gespeichert (Überlaufbereich)
- Verkettung der Synonyme (Überläufer) pro Hash-Klasse
- Suchen, Einfügen und Löschen sind auf Kollisionsklasse beschränkt
- Unterscheidung nach Primär- und Sekundärbereich:  $n > m$  ist möglich!

# Hash-Verfahren mit Verkettung der Überläufer (separater Überlaufbereich)



Entartung zur linearen Liste prinzipiell möglich

Nachteil: Anlegen von Überläufern, auch wenn Hash-Tabelle (Primärbereich) noch wenig belegt ist

# Offene Hash-Verfahren: Lineares Sondieren

Eigenschaften:

- Speicherung der Synonyme (Überläufer) im Primärbereich
- Hash-Verfahren muß in der Lage sein, eine Sondierungsfolge, d.h. eine Permutation aller Hash-Adressen, zu berechnen

## **Lineares Sondieren (linear probing)**

Von der Hausadresse (Hash-Funktion  $h$ ) aus wird sequentiell (modulo der Hash-Tabellen-Größe) gesucht. Offensichtlich werden dabei alle Plätze in HT erreicht:

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + i) \pmod{m}, \quad i = 1, 2, \dots$$

Häufung von Kollisionen durch "Klumpenbildung"  $\implies$  lange Sondierungsfolgen möglich

## Lineares Sondieren: Beispiel

Einfügereihenfolge 79, 28, 49, 88, 59

0	1	2	3	4	5	6	7	8	9
								28	79

$$h_0(K_p) = h(K_p) = K_p \bmod 10$$

$$h_i(K_p) = (K_p + i) \bmod 10, \quad i = 1, 2, \dots$$

## Lineares Sondieren: Beispiel

Einfügereihenfolge 79, 28, 49, 88, 59

0	1	2	3	4	5	6	7	8	9
49								28	79

$$h_0(K_p) = h(K_p) = K_p \bmod 10$$

$$h_i(K_p) = (K_p + i) \bmod 10, \quad i = 1, 2, \dots$$

## Lineares Sondieren: Beispiel

Einfügereihenfolge 79, 28, 49, 88, 59

0	1	2	3	4	5	6	7	8	9
49	88							28	79

$$h_0(K_p) = h(K_p) = K_p \bmod 10$$

$$h_i(K_p) = (K_p + i) \bmod 10, \quad i = 1, 2, \dots$$

## Lineares Sondieren: Beispiel

Einfügereihenfolge 79, 28, 49, 88, 59

0	1	2	3	4	5	6	7	8	9
49	88	59						28	79

$$h_0(K_p) = h(K_p) = K_p \bmod 10$$

$$h_i(K_p) = (K_p + i) \bmod 10, \quad i = 1, 2, \dots$$

## Lineares Sondieren (2)

### Aufwendiges Löschen

- impliziert oft Verschiebungen
- entstehende Lücken in Suchsequenzen sind aufzufüllen, da das Antreffen eines freien Platzes die Suche beendet.

0	1	2	3	4	5	6	7	8	9
49	88	59						28	79

↓ Lösche 28 ↓

0	1	2	3	4	5	6	7	8	9
49	88	59						♠	79

♠ = Platzhalter



## Lineares Sondieren (3)

Verbesserung: Modifikation der Überlauflfolge

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + f(i)) \bmod m \quad \text{oder}$$

$$h_i(K_p) = (h_0(K_p) + f(i, h(K_p))) \bmod m, \quad i = 1, 2, \dots$$

Beispiele:

- Weiterspringen um festes Inkrement  $c$  (statt nur 1):  $f(i) = ci$
- Sondierung in beiden Richtungen:  $f(i) = ci(-1)^i$

# Quadratisches Sondieren

Bestimmung der Speicheradresse

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + ai + bi^2) \pmod m, \quad i = 1, 2, \dots$$

$m$  sollte Primzahl sein.

Folgender Spezialfall ist wichtig:

$$h_i(K_p) = (h_0(K_p) - \lceil i/2 \rceil (-1)^i) \pmod m, \quad 1 \leq i \leq m-1$$

Beispiel: Einfügereihenfolge 79, 28, 49, 88, 59

## Weitere offene Hash-Verfahren

### Sondieren mit Zufallszahlen

- Mit Hilfe eines deterministischen Pseudozufallszahlen-Generators wird die Adressenfolge  $[1 \dots m - 1] \bmod m$  genau einmal erzeugt:

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = h_0(K_p) + z_i \bmod m, \quad i = 1, 2, \dots$$

### Double Hashing

- Einsatz einer zweiten Funktion für die Sondierungsfolge

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + ih'(K_p)) \bmod m, \quad i = 1, 2, \dots$$

Wähle  $h'(K)$  so, dass für alle Schlüssel  $K$  die Sondierungsfolge eine Permutation aller Hash-Adressen bildet

### Kettung von Synonymen

- explizite Kettung aller Sätze einer Kollisionsklasse
- verringert nicht die Anzahl der Kollisionen; sie verkürzt jedoch den Suchpfad beim Aufsuchen eines Synonyms.