IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(11 Hashverfahren: Allgemeiner Rahmen)

Prof. Dr. Susanne Albers

Das Wörterbuch-Problem (1)

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 (Methoden) effizient ausführbar sind:

- Suchen (Wiederfinden, Zugreifen)
- Einfügen
- Entfernen

Das Wörterbuch-Problem (2)

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 Suchen (statisches Verhalten)
  – überwiegend Einfügen & Löschen (dynamisches Verhalten)
  – annähernd Gleichverteilung
  – nichts bekannt
- Weitere zu implementierende Operationen:
  – Durchlaufen der Menge in bestimmter Reihenfolge (etwa nach Schlüsselwert aufsteigend)
  – Mengen-Operationen: Vereinigung, Durchschnitt, Differenz, . . .
  – Aufspalten
  – Konstruieren
- Kostenmaße zur Beurteilung der Lösung: average, worst, amortisierter worst case
- Ausführungsreihenfolge der Operationen:
  – sequentiell
  – nebenläufig

Das Wörterbuch-Problem (3)

Verschiedene Ansätze zur Lösung des WBP:

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

Hashing (engl.: to hash=zerhacken) beschreibt eine spezielle Art der Speicherung der Elemente einer Menge durch Zerlegung des Schlüssel-Universums.

Die Position des Datenelements im Speicher ergibt sich (zunächst) durch Berechnung direkt aus dem Schlüssel.

Hashverfahren

Wörterbuchproblem:

Suchen, Einfügen, Entfernen von Datensätzen (Schlüsseln)

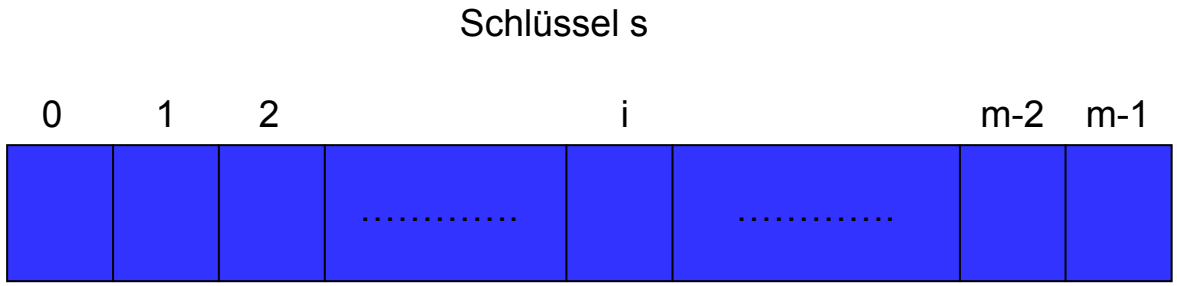
Ort des Datensatzes d: Berechnung aus dem Schlüssel *s* von *d*

→ keine Vergleiche

→ konstante Zeit

Datenstruktur: lineares Feld (Array) der Größe *m*

Hashtabelle



Der Speicher wird zerlegt in m gleich große Behälter (Buckets).

Hashtabellen - Beispiele

Beispiele:

- Compiler
  i int 0x87C50FA4
  j int 0x87C50FA8
  x double 0x87C50FAC
  name String 0x87C50FB2
  ...
- Umgebungsvariablen (Schlüssel,Attribut) Liste
  EDITOR=emacs
  GROUP=mitarbeiter
  HOST=vulcano
  HOSTTYPE=sun4
  LPDEST=hp5
  MACHTYPE=sparc
  ...
- Datenbanken (Index für Tabellen)
- Caches

Implementierung in Java

```
class TableEntry {
    private Object key,value;
}

abstract class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;

    //Konstruktor
    HashTable (int capacity) {
        this.capacity = capacity;
        tableEntry = new TableEntry [capacity];
        for (int i = 0; i <= capacity-1; i++)
            tableEntry[i] = null;
    }
    // die Hashfunktion
    protected abstract int h (Object key);

    // fuege Element mit Schluessel key und Wert value ein (falls nicht vorhanden)
    public abstract void insert (Object key Object value);

    // entferne Element mit Schluessel key (falls vorhanden)
    public abstract void delete (Object key);

    // suche Element mit Schluessel key
    public abstract Object search (Object key);
} // class hashTable
```

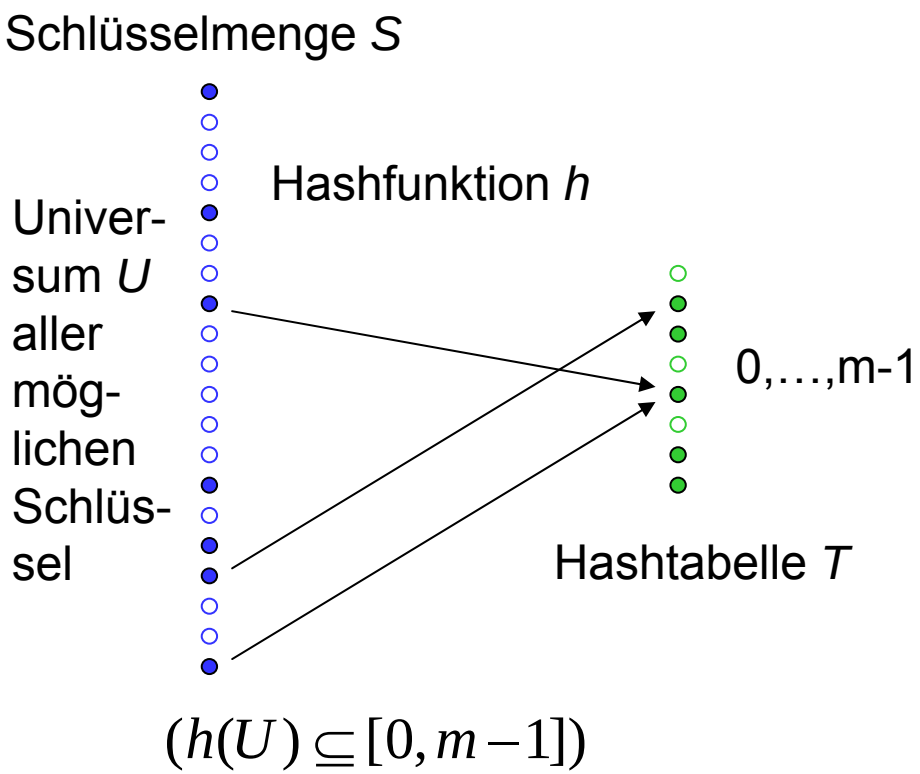
Hashverfahren - Probleme

1. Größe der Hashtabelle
   Nur eine kleine Teilmenge *S* aller möglichen Schlüssel (des Universums) *U* kommt vor
2. Berechnung der Adresse eines Datensatzes
   - Schlüssel sind keine ganzen Zahlen
     (können aber als Binärzahl aufgefasst werden)
3. - Index hängt von der Größe der Hashtabelle ab

In Java:

```
public class Object {
    ...
    public int hashCode()  {…}
    ...
}
```

Hashfunktion (1)



$h(s)$ = Hashadresse

$h(s) = h(s') \Leftrightarrow s$ und s' sind Synonyme bzgl. h

Adresskollision

Hashfunktion (2)

Definition: Sei *U* ein Universum möglicher Schlüssel und $\{B_0, \ldots, B_{m-1}\}$ eine Menge von *m* Behältern zum Speichern von Elementen aus *U:*
Dann ist eine Hash-Funktion eine totale Abbildung

$$h : U \rightarrow \{0, \ldots, m - 1\},$$

die jedem Schlüssel *s* aus *U* eine Nummer *h(s)* (und dem entsprechenden Element den Behälter $B_{h(s)}$) zuordnet.

- Die Behälternummern nennt man auch Hash-Adressen, die Gesamtmenge der Behälter Hash-Tabelle.

B_0
B_1
...
...
B_{m-1}

Adresskollisionen

- Eine Hashfunktion *h* berechnet für jeden Schlüssel *s* die Nummer des Buckets.
- Ideal wäre eine eindeutige Speicherzuordnung eines Datums mit Schlüssel *s* zum Bucket mit Nummer *h*(*s*): Einfügen und Suchen könnten in konstanter Zeit (O(1)) erfolgen.
- Tatsächlich treten natürlich Kollisionen auf: Mehrere Elemente können auf die gleiche Hash-Adresse abgebildet werden. Kollisionen müssen (auf eine von verschiedenen Arten) behandelt werden.

Hashverfahren

Beispiel für U: alle Namen in Java mit Länge ≤ 40 → $|U| = 62^{40}$

Falls $|U| > m$: Adresskollisionen unvermeidlich

Hashverfahren:

1. Wahl einer möglichst „guten“ Hash-Funktion
2. Strategie zur Auflösung von Adresskollisionen

Belegungsfaktor α :

$$\alpha = \frac{\#\,\text{gespeicherte Schlüssel}}{\text{Größe der Hash - Tabelle}} = \frac{|S|}{m} = \frac{n}{m}$$

Annahme: Tabellengröße m ist fest

Anforderungen an gute Hashfunktionen

Anforderungen an Hash-Funktionen

- Eine Kollision tritt dann auf, wenn bei Einfügen eines Elementes mit Schlüssel *s* der Bucket $B_{h(s)}$ schon belegt ist.
- Eine Hash-Funktion *h* heißt perfekt für eine Menge von Schlüsseln *S*, falls keine Kollisionen für *S* auftreten.
- Ist *h* perfekt und $|S| = n$, dann gilt: $n \leq m$. Der Belegungsfaktor (BF) der Hash-Tabelle ist $n/m \leq 1$.
- Eine Hash-Funktion ist gut gewählt, wenn
  - der Belegungsfaktor möglichst hoch ist,
  - für viele Schlüsselmengen die # der Kollisionen möglichst klein ist,
  - sie effizient zu berechnen ist.

Beispiel einer Hashfunktion

Beispiel: Hash-Funktion für Strings

```
public static int h (String s){
    int k = 0, m = 13;
    for (int i=0; i < s.length(); i++)
        k += (int)s.charAt (i);
    return ( k%m );
}
```

Folgende Hash-Adressen werden generiert für $m = 13$.

Schlüssel s	$h(s)$
Test	0
Hallo	2
SE	9
Algo	10

h wird perfekter, je größer m gewählt wird.

Kollisionswahrscheinlichkeit (1)

Zur Wahl der Hash-Funktion

- Die Anforderungen hoher Belegungsfaktor und Kollisionsfreiheit stehen in Konflikt zueinander. Es ist ein geeigneter Kompromiss zu finden.
- Für die Schlüssel-Menge S mit $|S| = n$ und Behälter $B_0, \ldots, B_{m-1}$ gilt:
  – für $n > m$ sind Konflikte unausweichlich
  – für $n < m$ gibt es eine (Rest-) Wahrscheinlichkeit $P_K(n,m)$ für das Auftreten mindestens einer Kollision.

Wie findet man Abschätzung für $P_K(n,m)$?

- Für beliebigen Schlüssel s ist die W'keit dafür, dass $h(s) = j$ mit $j \in \{0, \ldots, m-1\}$: $P_K[h(s) = j] = 1/m$, falls Gleichverteilung gilt.
- Es ist $P_K(n,m) = 1 - P_{\neg K}(n,m)$,
  wenn $P_{\neg K}(n,m)$ die W'keit dafür ist, dass es beim Speichern von n Elementen in m Behälter zu keinen Kollisionen kommt.

Kollisionswahrscheinlichkeit (2)

Zur Wahrscheinlichkeit von Kollisionen

- Werden n Schlüssel nacheinander auf die Behälter $B_0, \ldots, B_{m-1}$ verteilt (bei Gleichverteilung), gilt jedes mal $P[h(s) = j] = 1/m$.
- Die W'keit $P(i)$ für keine Kollision im Schritt i ist $P(i) = (m - (i - 1))/m$
- Damit ist

$$P_k(n, m) = 1 - P(1) * P(2) * \ldots * P(n) = 1 - \frac{m(m-1)\ldots(m-n+1)}{m^n}$$

Für $m = 365$ etwa ist $P(23) > 50\%$ und $P(50) \approx 97\%$ (Geburtstagsparadoxon)

Gebräuchliche Hashfunktionen

In der Praxis verwendete Hash-Funktionen:

- Siehe: D.E. Knuth: The Art of Computer Programming
- Für U = integer wird die Divisions-Rest-Methode verwandt:

$$h(s) = (a \times s) \bmod m \qquad (a \neq 0,\ a \neq m, \quad m \text{ Primzahl})$$

- Für Zeichenreihen der Form $s = s_0 s_1 \ldots s_{k-1}$ nimmt man etwa:

$$h(s) = \left(\left(\sum_{i=0}^{k-1} B^i s_i \right) \bmod 2^w \right) \bmod m$$

etwa mit $B = 131$ und w = Wortbreite des Rechners ($w = 32$ oder $w = 64$ ist üblich).

Einfache Hashfunktion

Wahl der Hash-Funktion

- leichte und schnelle Berechenbarkeit
- gleichmäßige Verteilung der Daten (Beispiel: Compiler)

(Einfache) Divisions-Rest-Methode

$$h(k) = k \bmod m$$

Wahl von m?

Beispiele:

a) m gerade → $h(k)$ gerade $\Leftrightarrow$ k gerade

Problematisch, wenn letztes Bit Sachverhalt ausdrückt (z.B. 0 = weiblich, 1 = männlich)

b) $m = 2^p$ liefert p niedrigsten Dualziffern von k

Regel: Wähle m prim, wobei m keine Zahl $r^i +- j$ teilt,
wobei i und j kleine, nicht negative Zahlen und r Radix der Darstellung sind.

Multiplikative Methode (1)

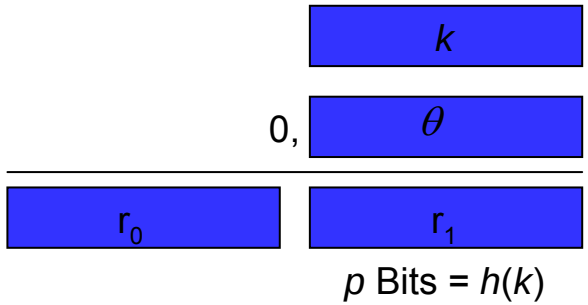
Wähle Konstante θ, $0 < \theta < 1$

1. Berechne $k\theta \bmod 1 = k\theta - \lfloor k\theta \rfloor$

2. $h(k) = \lfloor m(k\theta \bmod 1) \rfloor$

Wahl von m unkritisch, wähle $m = 2^p$:

Berechnung von $h(k)$:



Multiplikative Methode (2)

Beispiel:

$$\theta = \frac{\sqrt{5} - 1}{2} \approx 0.6180339$$

$$k = 123456$$

$$m = 10000$$

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

Von allen Zahlen $0 \leq \theta \leq 1$ führt zur $\frac{\sqrt{5} - 1}{2}$ gleichmäßigsten Verteilung.

Universelles Hashing

Problem: *h* fest gewählt → es gibt $S \subseteq U$ mit vielen Kollisionen

Idee des universellen Hashing:

Wähle Hashfunktion *h* zufällig

H endliche Menge von Hashfunktionen

$$h \in H : U \to \{0,\ldots,m-1\}$$

Definition: *H* heißt universell, wenn für beliebige $x,y \in U$ gilt:

$$\frac{|\{h \in H : h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

Folgerung: $x,y \in U$ beliebig, *H* universell, $h \in H$ zufällig

$$\Pr_H(h(x) = h(y)) \leq \frac{1}{m}$$

Universelles Hashing

Definition:

$$\delta(x,y,h) = \begin{cases} 1 \text{ falls } h(x) = h(y) \text{ und } x \neq y \\ 0 \text{ sonst} \end{cases}$$

Erweiterung von Elementen auf Mengen

$$\delta(x,S,h) = \sum_{s \in S} \delta(x, s, h)$$

$$\delta(x,y,G) = \sum_{h \in G} \delta(x, y, h)$$

Folgerung: H ist universell, wenn für alle $x,y \in U$

$$\delta(x, y, H) \leq \frac{|H|}{m}$$

Eine universelle Klasse von Hashfunktionen

Annahmen:

- $|U| = p$ mit Primzahl p und $|U| = \{0,\ldots,p\text{-}1\}$
- Seien $a \in \{1,\ldots,p\text{-}1\}$ und $b \in \{0,\ldots,p\text{-}1\}$ und $h_{a,b} : U \rightarrow \{0,\ldots,m\text{-}1\}$ wie folgt definiert

$$h_{a,b} = ((ax+b) \bmod p) \bmod m$$

Folgerung:

Die Menge

$$H = \{h_{a,b} \mid 1 \leq a \leq p\text{-}1,\ 0 \leq b \leq p\text{-}1\}$$

ist eine universelle Klasse von Hashfunktionen.

Universelles Hashing - Beispiel

Hashtabelle T der Größe 3, $|U| = 5$

Betrachte die 20 Funktionen (Menge H):

x+0	2x+0	3x+0	4x+0
x+1	2x+1	3x+1	4x+1
x+2	2x+2	3x+2	4x+2
x+3	2x+3	3x+3	4x+3
x+4	2x+4	3x+4	4x+4

jeweils (mod 5) (mod 3)

und die Schlüssel 1 und 4

Es gilt:

(1*1+0) mod 5 mod 3 = 1 = (1*4+0) mod 5 mod 3
(1*1+4) mod 5 mod 3 = 0 = (1*4+4) mod 5 mod 3
(4*1+0) mod 5 mod 3 = 1 = (4*4+0) mod 5 mod 3
(4*1+4) mod 5 mod 3 = 0 = (4*4+4) mod 5 mod 3

Möglichkeiten der Kollisionsbehandlung

Kollisionsbehandlung:

- Die Behandlung von Kollisionen erfolgt bei verschiedenen Verfahren unterschiedlich.
- Ein Datensatz mit Schlüssel *s* ist ein Überläufer, wenn der Behälter *h*(*s*) schon durch einen anderen Satz belegt ist.
- Wie kann mit Überläufern verfahren werden?
  1. Behälter werden durch verkettete Listen realisiert. Überläufer werden in diesen Listen abgespeichert.
  Chaining (Hashing mit Verkettung der Überläufer)
  2. Überläufer werden in noch freien anderen Behältern abgespeichert. Diese werden beim Speichern und Suchen durch sogenanntes Sondieren gefunden.
  Open Addressing (Offene Hashverfahren)