Informatik II
Hashing

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Das Datenbank-Problem "revisited"

- Lösung bisher:
  - Preprocessing: Elemente 1x sortieren, kostet $O(n \log n)$
  - Laufzeit: Suchen kostet dann nur noch Zeit $O(\log n)$
  - Nachteil: Einfügen ist teuer
- Ziel: Datenstruktur mit möglichst folgenden Eigenschaften
  - Kein Preprocessing nötig
  - Suchen und Einfügen soll in Zeit O(1) möglich sein!
  - Beliebige Keys
- Namen: *Hash-Table*, *Dictionary, Symbol-Table, assoziatives Array*
- In Python: Datenstruktur **`Dictionary`**
- Hash-Tabellen / Dictionaries sind — in gewissem Sinn — Verallgemeinerungen von Arrays

```
d = {}
d["hallo"] = 12
d["Paul"] = 18
```

Beispiele für Hash-Tabellen

- Memory Management: Tabelle im Betriebssystem
- Symbol-Tabellen im Compiler: Variablen/Identifier in einem Programm
  i int 0x87C50FA4
  j int 0x87C50FA8
  x double 0x87C50FAC
  name String 0x87C50FB2
  ...
- Environment-Variablen in Unix (Variablenname, Attribut)
  EDITOR=emacs
  GROUP=mitarbeiter
  HOST=vulcano
  HOSTTYPE=sun4
  PRINTER=hp5
  MACHTYPE=sparc
  ...
- Der Ort (= Pfad) ausführbarer Programme auf der Festplatte
  PATH=~/bin:/usr/local/gnu/bin:/usr/local/bin:/usr/bin:/bin:

Verschiedene Ansätze zur Lösung des DBP

- Strukturierung der Key-Menge: Listen, Arrays, Bäume, Graphen, …
- Hier: *Hashing* = Aufteilung des gesamten Key-Universums
- Die Position des Daten-Elements im Speicher ergibt sich (zunächst) durch Berechnung direkt aus dem Key
  - keine Vergleiche & konstante Zeit
- Datenstruktur = lineares Array der Größe m → Hash-Tabelle
- Hashing (engl.: to hash = zerhacken) beschreibt eine spezielle Art der Speicherung einer Menge von Keys durch Zerlegung des Key-Universums

Typische Implementierung einer Hash-Klasse

```
class TableEntry( object ):
    def __init__( self, key, value ):
        self.key = key
        self.value = value

class HashTable( object ):
    def __init__( self, capacity ):
        self.capacity = capacity
        self.table = capacity * [ None ]
        # wir nehmen hier an, daß table ein statisches Array sei,
        # mit fester Größe, wie das auch in C++/Java der Fall wäre
    # Hash-Funktion
    def h( self, key ): …
    # Füge value mit Schlüssel key ein, falls noch nicht vorhanden
    def insert( self, key, value ): …
    # Lösche Element mit key aus Tabelle, falls vorhanden
    def delete( self, key ): …
    # Suche Element mit key und liefere dessen Wert
    def search( self, key ): …
```


Prinzipielle Idee des Hashings

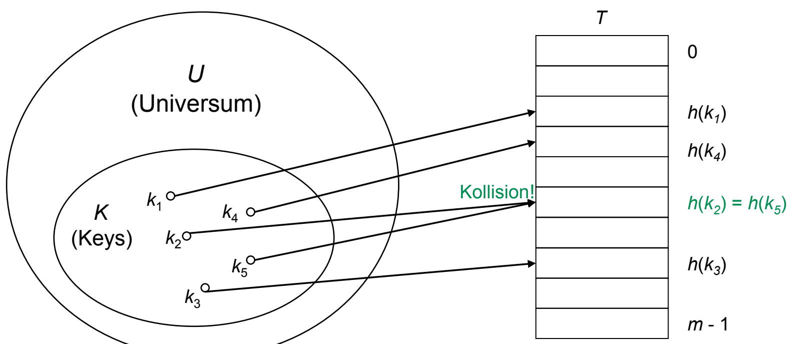
- Notation:
  - U = Universum aller möglichen Keys
  - K = Menge von Keys, die aktuell in der Hash-Tabelle gespeichert sind
  - $|K| = n$
- Beobachtung: wenn U sehr groß ist, ist ein Array für ganz U nicht praktikabel
  - Außerdem gilt im Allgemeinen: $|K| << |U|$
- Idee der Hash-Tabelle: benutze eine Tabelle, deren Größe in der selben Größenordnung wie $|K|$ ist
- Einträge der Hash-Tabelle nennt man Slots
- Definiere Funktion, die die Keys auf die Slots der Hash-Tabelle abbildet

- Hash-Funktion: surjektive Abbildung von U in alle Slots einer Hash-Tabelle T[0..m-1]

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

- Vergleiche Arrays: Key *k* wird abgebildet in den Slot *A*[*k*]
- Bei Hash-Tabellen: Key *k* wird abgebildet in den Slot *T*[*h*(*k*)] ("*k hashes to …*")
- *h*(*k*) heißt der Hash-Wert oder die Hash-Adresse des Keys *k*

Kollisionen und Belegungsfaktor



$h(k_2) = h(k_5)$ heißen auch Synonyme

- Normalerweise gilt |U| >> m ⇒ h kann nicht injektiv sein ⇒ (Adress-)Kollisionen sind unvermeindlich
- Belegungsfaktor (*load factor*):

$$\alpha = \frac{\#\text{ gespeicherter Keys}}{\text{Größe der Hash-Tabelle}} = \frac{|K|}{m}$$

Die beiden Bestandteile eines Hash-Verfahrens

- Hash-Verfahren :=
  1. möglichst "gute" Hash-Funktion +
  2. Strategie zur Auflösung von Adresskollisionen

- Annahme im Folgenden: Tabellengröße m ist fest

Anforderungen an gute Hash-Funktionen

- Eine Kollision tritt dann auf, wenn bei Einfügen eines Elementes mit Schlüssel k der Slot $T[\,h(k)\,]$ schon belegt ist
- Eine Hash-Funktion h heißt perfekt für eine Menge von Keys K, falls keine Kollisionen für K auftreten
- Die Hash-Funktion h kann nur dann perfekt sein, wenn $|K| = \leq m$; m.a.W.: der Belegungsfaktor der Hash-Tabelle muß $\frac{n}{m} \leq 1$ sein
- Eine Hash-Funktion ist gut, wenn:
  - für viele Schlüssel-Mengen auch bei hohem Belegungsfaktor die Anzahl der Kollisionen möglichst klein ist; und
  - diese effizient zu berechnen ist.

Beispiel einer Hash-Funktion

- *U* = {alle möglichen Identifiers in einer Programmiersprache}
  - M.a.W.: U = alle möglichen Funktions-, Variablen-, ..., Klassennamen
- Eine mögliche, einfache Hash-Funktion für Strings:

```
# m = size of table = const
def h( k, m ):
    s = 0
    for i in range( 0, len(k) ):
        s += ord( k[i] )    # ord = ASCII value of char
    return s % m
```

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

Schlüssel k	h(k)
Test	0
Hallo	2
bar	10
Algo	10

- *h* wird umso besser, je größer *m* gewählt wird

Zur Wahrscheinlichkeit einer Kollision

- Die Anforderungen "hoher Belegungsfaktor" und "Kollisionsfreiheit" stehen offensichtlich in Konflikt zueinander
- Für eine Menge *K*, mit $|K|=n$, und Tabelle *T*, mit $|T|$=m, gilt:
  - für $n > m$ sind Konflikte unausweichlich
  - für $n \leq m$ gibt es eine (Rest-) Wahrscheinlichkeit $P(n, m)$ für das Auftreten mindestens einer Kollision
- Wie findet man eine Abschätzung für $P(n, m)$?
  - Für beliebigen Key *k* ist die W'keit dafür, daß $h(k) = j$, $j \in \{0,...,m-1\}$: $P[\ h(k) = j\ ] = \frac{1}{m}$, falls Gleichverteilung gilt
  - Es ist $P(n, m) = 1 - \overline{P}(n, m)$, wenn $\overline{P}(n, m)$ die W'keit dafür ist, daß es beim Speichern von *n* Elementen in *m* Slots zu keinen Kollisionen kommt

- Werden n Schlüssel nacheinander auf die Slots $T_0, \ldots, T_{m-1}$ verteilt (bei Gleichverteilung), gilt jedes mal

$$P[h(s) = j] = \frac{1}{m}$$

- Die W'keit für keine Kollision im Schritt i ist $p_i = \frac{m-(i-1)}{m}$
- Damit ist

$$P(n, m) = 1 - p_1 \cdot p_2 \cdots p_n = 1 - \frac{m(m-1)\cdots(m-n+1)}{m^n}$$

- Beispiel ("Geburtstagsparadoxon"):

$$P(23{,}365) > 50\%$$

und

$$P(50{,}365) \approx 97\%$$

Gebräuchliche Hash-Funktionen

- Zunächst die Divisions-Methode
- Für U = Integer wird die Division-mit-Rest-Methode verwandt:

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

- Für Strings der Form $s = s_1 s_2 \ldots s_k$ nimmt man oft:

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

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

- (Einfache) Divisions-Methode:

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

- Wahl von *m* ?
- Schlechte Beispiele:
  - m gerade → $h(k)$ gerade ⇔ k gerade
    - Ist problematisch, wenn letztes Bit eine spezielle Bedeutung hat! (z.B. 0 = weiblich, 1 = männlich)
  - $m = 2^p$ liefert die p niedrigsten Binärziffern von k, d.h., höhere Ziffern gehen gar nicht in die Hash-Adresse ein!
- Regel: Wähle *m* prim, wobei *m* keine Zahl $2^i \pm j$ teilt, wobei *i* und *j* kleine, nicht-negative Zahlen sind, d.h., wähle *m* prim, aber nicht "zu nahe" an einer 2-er-Potenz

Multiplikative Methode

- Satz von Vera Turán Sós [1957]:

  Sei Θ eine irrationale Zahl. Platziert man die Punkte $\Theta - \lfloor\Theta\rfloor, 2\Theta - \lfloor 2\Theta\rfloor, \ldots, n\Theta - \lfloor n\Theta\rfloor$ in das Intervall $[0,1]$, dann haben die $n+1$ Intervallteile höchstens 3 verschiedene Längen. Außerdem fällt der nächste Punkt $(n+1)\Theta - \lfloor(n+1)\Theta\rfloor$ in eines der größten (schon existierenden) Intervallteile.

- Fazit: die so gebildeten "Punkte" liegen ziemlich gleichmäßig gestreut im Intervall [0,1]
- Es gilt: von allen Zahlen θ, 0 ≤ θ ≤ 1, führt der goldene Schnitt

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

zur gleichmäßigsten Verteilung.

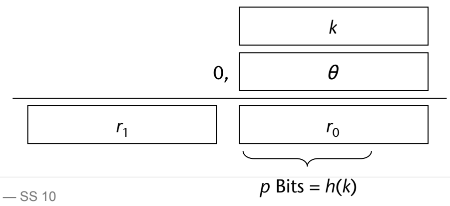
- Wähle eine Konstante θ, $0 < \theta < 1$
  1. Berechne $k\theta \bmod 1 := k\theta - \lfloor k\theta \rfloor$
  2. $h(k) = \lfloor m(k\theta \bmod 1) \rfloor$

- Beispiel:

$$\theta^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.6180339$$
$$k = 123456$$
$$m = 10000$$
$$\begin{aligned} h(k) &= \lfloor 10000(123456 \cdot 0.61803\ldots \bmod 1 \rfloor \\ &= \lfloor 10000(76300.0041151\ldots \bmod 1 \rfloor \\ &= \lfloor 41.151\ldots \rfloor = 41 \end{aligned}$$

Praktische Berechnung von *h* in der multiplikativen Methode

- Wahl von *m* (= Tabellengröße) ist hier unkritisch → wähle $m = 2^p$
- Ann.: *k* passe in ein einzelnes Wort, d.h., *k* hat *w* Bits
- Wähle $\theta \in [0,1)$ und setze $s = \theta \cdot 2^w$
- Dann ist $k \cdot s = k \cdot \theta \cdot 2^w = r_1 2^w + r_0$
- r_1 ist der ganzzahlige Teil von $k\theta$ (= $\lfloor k\theta \rfloor$) und r_0 ist der gebrochene Rest (= $k\theta \bmod 1 = k\theta - \lfloor k\theta \rfloor$)
- Damit kann man $h(k)$ mit Integer-Arithmetik berechnen:



Universelles Hashing

- Problem: h fest gewählt → es gibt ein $S \subseteq U$ mit vielen Kollisionen
  - Wir können nicht annehmen, daß die Keys gleichverteilt im Universum liegen (z.B. Identifier im Programm)
  - Könnte also bspw. passieren, daß die Compile-Zeit bei einigen best. Programmen sehr lange dauert, weil es sehr viele Kollisionen gibt
- Idee des universellen Hashing:
  - wähle Hash-Funktion h zufällig (→ randomisierte Datenstruktur)
- Definition: Sei H endliche Menge von Hash-Funktionen, $h \in H : U \rightarrow \{0, \ldots, m-1\}$, dann heißt H universell, wenn gilt:

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

- Äquivalent: für $x, y \in U$ beliebig und $h \in H$ zufällig gilt

$$P\left[h(x) = h(y)\right] \leq \frac{1}{m}$$

- Definition: "Kollisionsindikator"

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

- Erweiterung von δ auf Mengen

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

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

- Definition für "universell" nochmal mit δ formuliert:

H ist universell $\Leftrightarrow \forall x, y \in U : \delta(x, y, H) \leq \frac{|H|}{m}$

Nutzen des Universellen Hashing

- Sei K eine Menge von Schlüsseln, die in Tabelle T gespeichert werden sollen
  - Wähle zufällig Hash-Funktion $h \in H$, diese bleibt fest für die restliche Lebensdauer der Tabelle
  - Bilde alle Schlüssel $k \in K$ mit h auf Tabelle ab und füge diese ein
- Nun soll weiterer Key x gespeichert werden
  - Vernünftig ist: Maß für Aufwand = #Kollisionen zw. x und allen $k \in K$
  - Berechne den Erwartungswert für diese Anzahl, also $E[\,\delta(x, K, h)\,]$

G. Zachmann Informatik 2 — SS 10 Hashing 23

$$
\begin{aligned}
E\left[\delta(x,K,h)\right] &= \frac{1}{|H|}\sum_{h\in H}\delta(x,K,h) \\
&= \frac{1}{|H|}\sum_{h\in H}\sum_{y\in K}\delta(x,y,h) \\
&= \frac{1}{|H|}\sum_{y\in K}\sum_{h\in H}\delta(x,y,h) \\
&= \frac{1}{|H|}\sum_{y\in K}\delta(x,y,H) \\
&\leq \frac{1}{|H|}\sum_{y\in K}\frac{|H|}{m} \\
&= \frac{|K|}{m}
\end{aligned}
$$

G. Zachmann Informatik 2 — SS 10 Hashing 24

- Schlußfolgerung:

$$E\left[\delta(x, K, h)\right] \leq \frac{|K|}{m}$$

Man kann also erwarten, daß eine aus einer universellen Klasse *H* von Hash-Funktionen zufällig gewählte Funktion *h* eine beliebige, noch so "bösartig" gewählte Folge von Schlüsseln (also bei einem *"malicious adversary"*) so gleichmäßig wie nur möglich in der Hash-Tabelle verteilt.