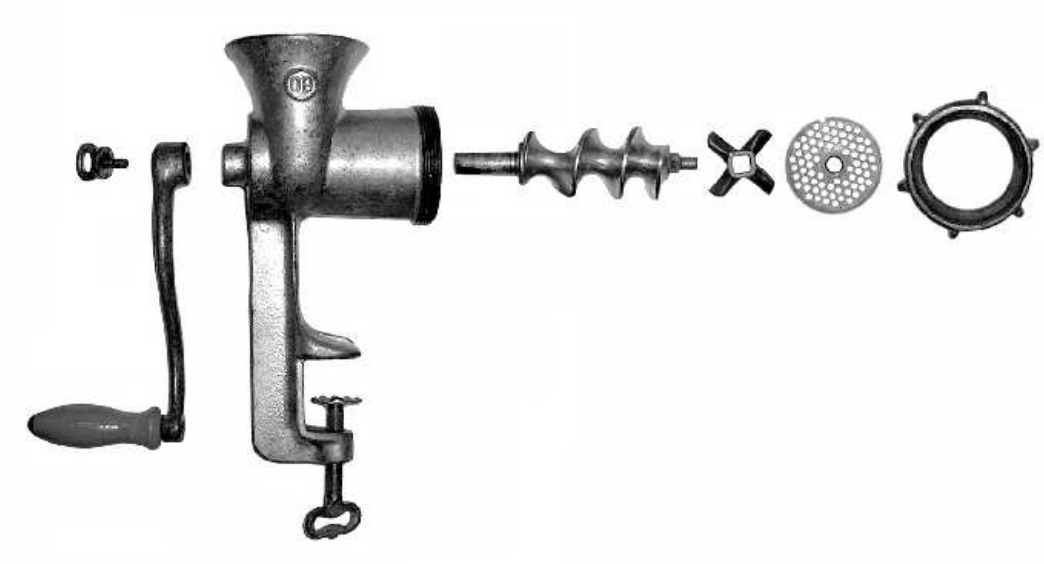
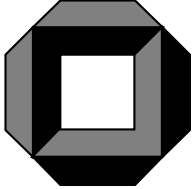
4 Hashing (Streuspeicherung)



“to hash” ≈ “völlig durcheinander bringen”.

Paradoxerweise hilft das, Dinge wiederzufinden



Hashtabellen

speichere Menge $M \subseteq$ Element.

key(e) ist eindeutig für $e \in M$.

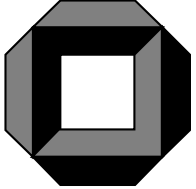
unterstütze Wörterbuch-Operationen in Zeit $O(1)$.

M.insert$(e :$ Element$)$: $M := M \cup \{e\}$

M.remove$(k :$ Key$)$: $M := M \setminus \{e\}, e = k$

M.find$(k :$ Key$)$: return $e \in M$ with $e = k$; $\perp$ falls nichts gefunden

(Konvention: key ist *implizit*), d.h. $e = k$ gdw key$(e) = k$)



Hashing: Anwendungen

- Auslieferungsregale der UB Karlsruhe
- Entfernen exakter Duplikate
- Schach (oder andere kombinatorische Suchprogramme):
  welche Stellungen wurden bereits durchsucht ?
- Symboltabelle bei Compilern
- Assoziative Felder bei Script-Sprachen wie perl oder awk
- Datenbank-Gleichheits-Join
  (wenn eine Tabelle in den Speicher passt)
- Unsere Routenplaner: Teilmengen von Knoten,
  z.B. Suchraum
- . . .

Ein (über)optimistischer Ansatz

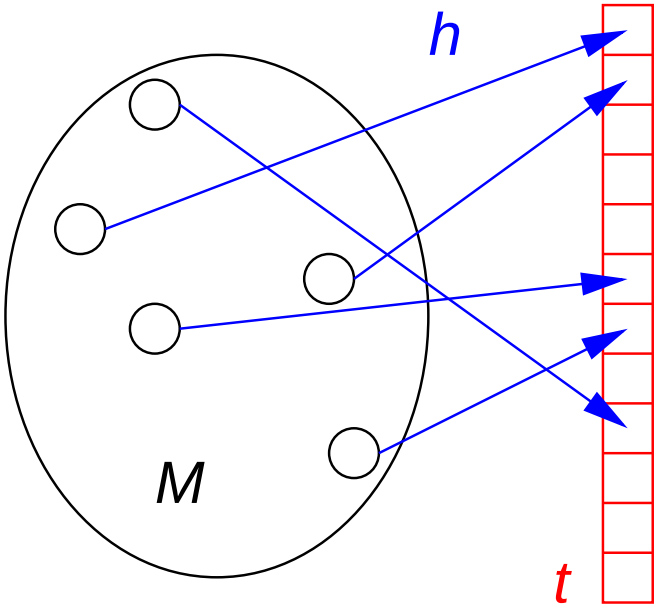
Eine perfekte Hash-Funktion h

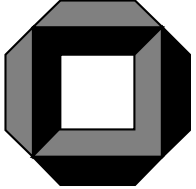
bildet Elemente von M injektiv

auf eindeutige Einträge

der Tabelle $t[0..m-1]$ ab, d.h.,

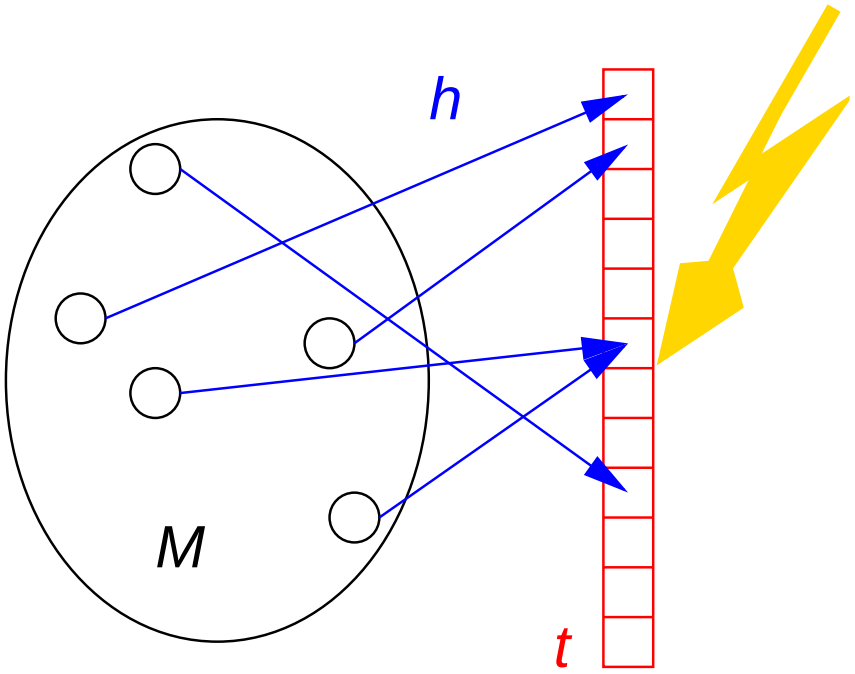
$t[h(\text{key}(e))] = e$





Kollisionen

Perfekte Hash-Funktionen sind schwer zu finden

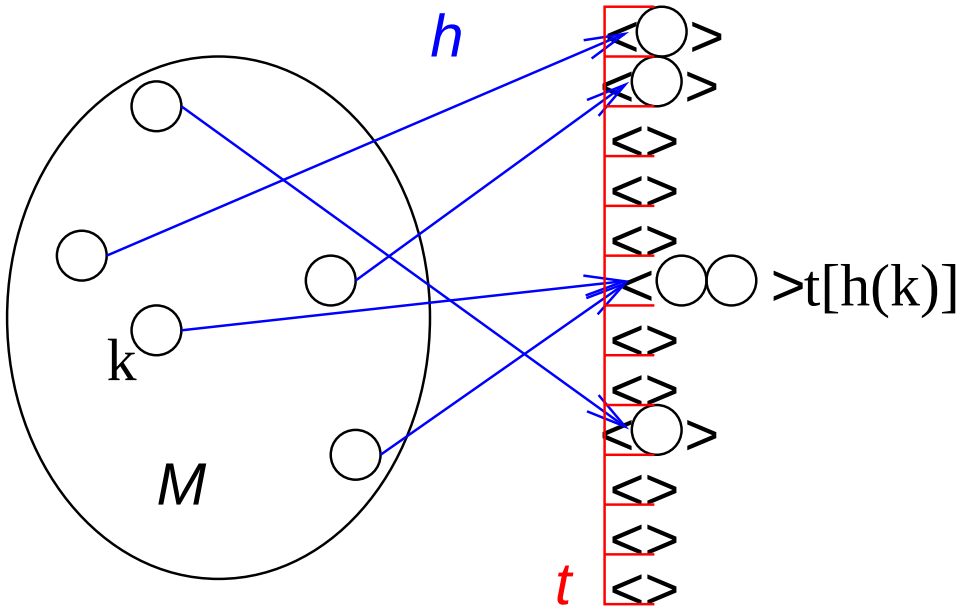


Beispiel: Geburtstagsparadox

Kollisionsauflösung

Beispiel geschlossenes Hashing

Tabelleneinträge: Elemente $\rightsquigarrow$ Folgen von Elementen



4.1 Hashing mit verketteten Listen

Implementiere die Folgen beim geschlossenen Hashing

durch einfach verkettete Listen

insert(e): Füge e am Anfang von $t[h(e)]$ ein. konstante Zeit

remove(k): Durchlaufe $t[h(k)]$. Element e mit $h(e) = k$ gefunden $\rightsquigarrow$

löschen und zurückliefern.

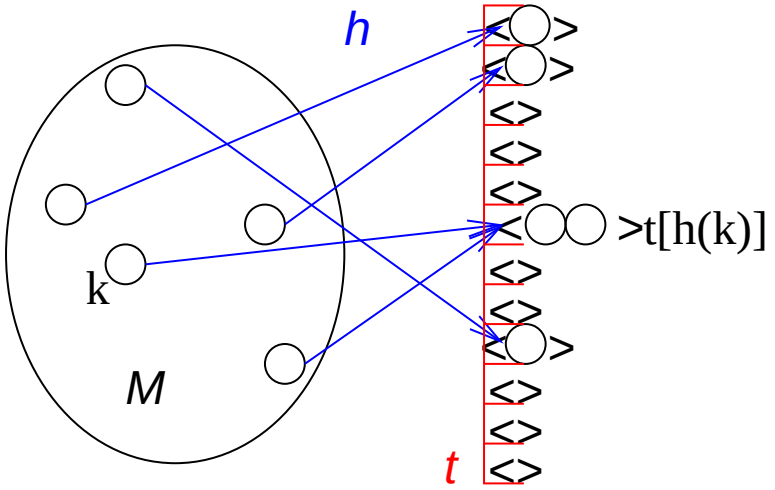
find(k) : Durchlaufe $t[h(k)]$. Element e mit $h(e) = k$ gefunden $\rightsquigarrow$

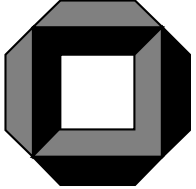
zurückliefern.

Sonst: $\perp$ zurückgeben.

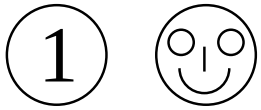
Zeit $O(|M|)$ im schlechtesten Fall für

remove und find





W.T. im Schnelldurchlauf



Hash-Beispiel

Elementarereignisse Ω — zuf. Hash-Funktionen $\{0..m-1\}^{\text{Key}}$

Ereignisse: Teilmengen von Ω — $\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$

p_x = Wahrscheinlichkeit von $x \in \Omega$ — Gleichverteilung $p_h = m^{-|\text{Key}|}$

$\mathbb{P}[\mathcal{E}] = \sum_{x \in E} p_x$ — $\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$

Zufallsvariable $X_0 : \Omega \to \mathbb{R}$ — $X = |\{e \in M : h(e) = 0\}|.$

Erwartungswert $\mathrm{E}[X_0] = \sum_{y \in \Omega} p_y X(y)$ — $\mathrm{E}[X] = \frac{|M|}{m}$ (*)

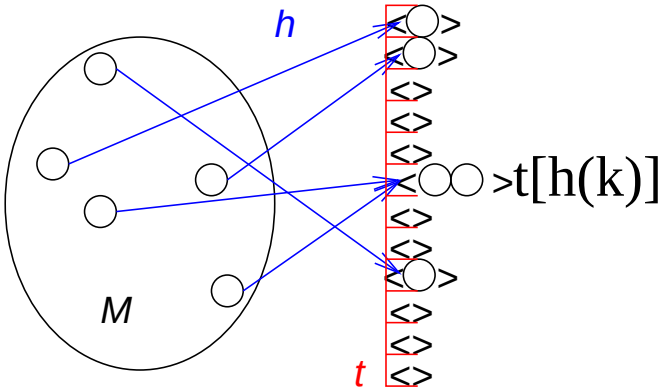
Linearität des Erwartungswerts: $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$

Beweis von (*):

Betrachte die 0-1 ZV $X_e = 1$ für $h(e) = 0$, $e \in M$ und $X_e = 0$ sonst.

$$\mathrm{E}[X_0] = \mathrm{E}[\sum_{e \in M} X_e] = \sum_{e \in M} \mathrm{E}[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = |M| \cdot \frac{1}{m}$$

Analyse für zufällige Hash-Funktionen



Satz 1. *Die erwartete Ausführungszeit von* remove(k) *und* find(k) *ist* $O(1)$ *falls* $|M| = O(m)$.

Beweis. Konstante Zeit plus Zeit für Durchlaufen von $t[h(k)]$.

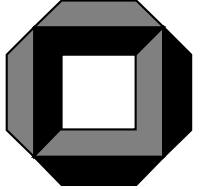
$X := |t[h(k)]| = |\{e \in M : h(e) = h(k)\}|$.

Betrachte die 0-1 ZV $X_e = 1$ für $h(e) = h(k)$, $e \in M$ und $X_e = 0$ sonst.

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}[\sum_{e \in M} X_e] = \sum_{e \in M} \mathrm{E}[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = \frac{|M|}{m} \\ &= O(1) \end{aligned}$$

□

Das gilt unabhängig von der Eingabe M.



4.2 Universelles Hashing

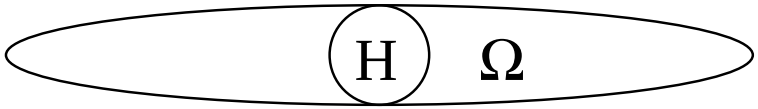
Idee: nutze nur bestimmte "einfache" Hash-Funktionen

Definition 1. $\mathcal{H} \subseteq \{0..m-1\}^{\text{Key}}$ *ist universell*
falls für alle x, y in Key *mit $x \neq y$ und zufälligem $h \in \mathcal{H}$,*

$$\mathbb{P}[h(x) = h(y)] = \frac{1}{m} \quad .$$

Satz 2. *Theorem 1 gilt auch für universelle Familien von Hash-Funktionen.*

Beweis. Für $\Omega = \mathcal{H}$ haben wir immer noch $\mathbb{P}[X_e = 1] = \frac{1}{m}$.
Der Rest geht wie vorher. □



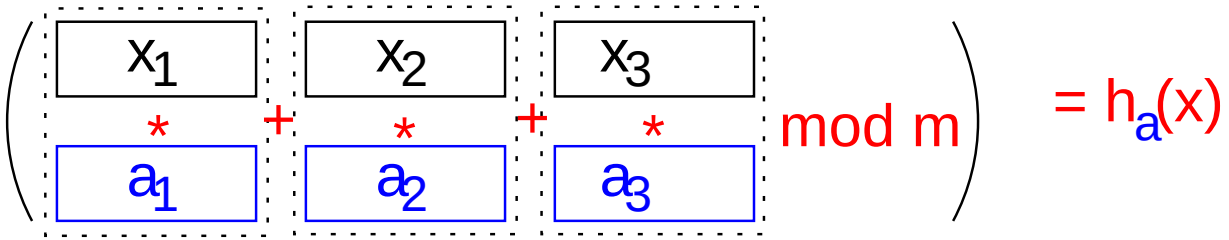
Eine einfache universelle Familie

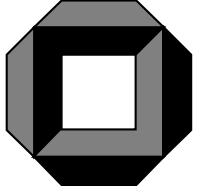
m sei eine Primzahl, Key $\subseteq \{0,\ldots,m-1\}^k$

Satz 3. *Für* $\mathbf{a} = (a_1,\ldots,a_k) \in \{0,\ldots,m-1\}^k$ *definiere*

$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a}\cdot\mathbf{x} \bmod m$, $H^{\cdot} = \left\{h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k\right\}$.

$H^{\cdot}$ *ist eine universelle Familie von Hash-Funktionen*





Beweis. Betrachte $\mathbf{x} = (x_1, \ldots, x_k)$, $\mathbf{y} = (y_1, \ldots, y_k)$ mit $x_j \neq y_j$
zähle **a**-s mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
Für jede Wahl von a_is, $i \neq j$, $\exists$ genau ein a_j mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\sum_{1 \leq i \leq k} a_i x_i \equiv \sum_{1 \leq i \leq k} a_i y_i \; (\mathbf{mod}\; m)$$

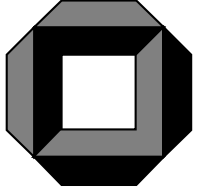
$$\Leftrightarrow a_j (x_j - y_j) \equiv \sum_{i \neq j, 1 \leq i \leq k} a_i (y_i - x_i) \; (\mathbf{mod}\; m)$$

$$\Leftrightarrow a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \leq i \leq k} a_i (y_i - x_i) \; (\mathbf{mod}\; m)$$

m^{k-1} Möglichkeiten a_i auszuwählen (mit $i \neq j$).
m^k ist die Gesamtzahl **a**s, d.h.,

$$\mathbb{P}[h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}.$$

□



Bit-basierte Universelle Familien

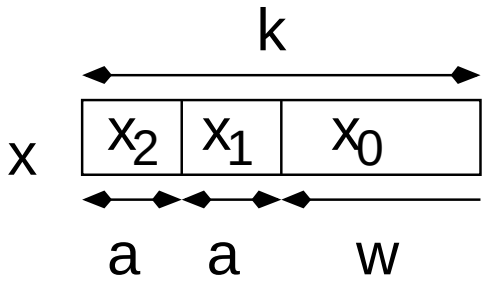
Sei $m = 2^w$, Key $= \{0,1\}^k$

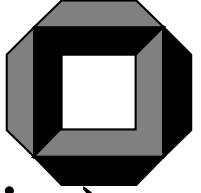
Bit-Matrix Multiplikation: $H^{\oplus} = \left\{ h_{\mathbf{M}} : \mathbf{M} \in \{0,1\}^{w \times k} \right\}$

wobei $h_{\mathbf{M}}(\mathbf{x}) = \mathbf{M}x$ (Arithmetik $\bmod 2$, d.h., xor, and)

Tabellenzugriff: $H^{\oplus[]} = \left\{ h^{\oplus[]}_{(t_1,\ldots,t_b)} : t_i \in \{0..m-1\}^{\{0..w-1\}} \right\}$

wobei $h^{\oplus[]}_{(t_1,\ldots,t_b)}((x_0,x_1,\ldots,x_b)) = x_0 \oplus \bigoplus_{i=1}^{b} t_i[x_i]$





4.3 Hashing mit Linearer Suche (Linear Probing)

Offenes Hashing: zurück zur Ursprungsidee.

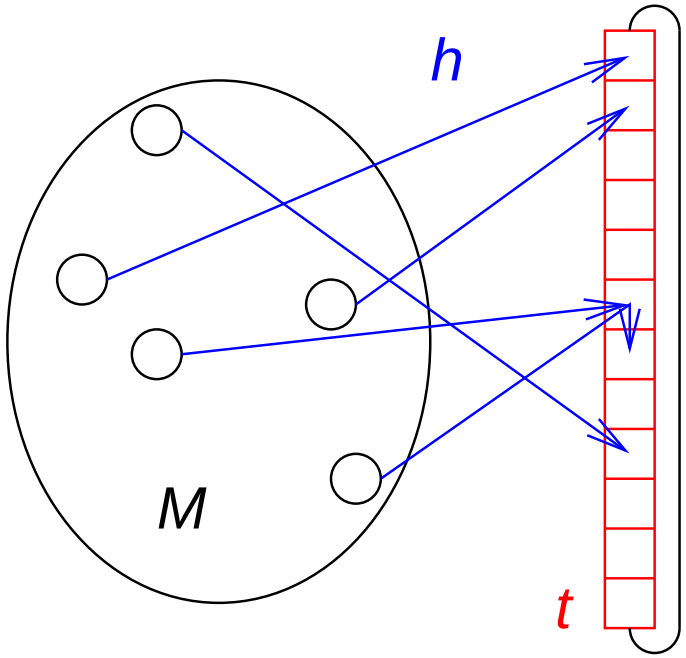
Elemente werden direkt in der Tabelle gespeichert.

Kollisionen werden durch Finden anderer Stellen aufgelöst.

linear probing: Suche nächsten freien Platz.

Am Ende fange von vorn an.

- einfach
- Platzeffizient
- Cache-effizient



Der einfache Teil

Class BoundedLinearProbing$(m, m' : \mathbb{N}; h : \text{Key} \to 0..m-1)$

$t = [\bot, \ldots, \bot]$: **Array** $[0..m+m'-1]$ **of** Element

invariant $\forall i : t[i] \neq \bot \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[i] \neq \bot$

Procedure insert$(e : \text{Element})$

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \bot$ **do** ;

assert $i < m + m' - 1$

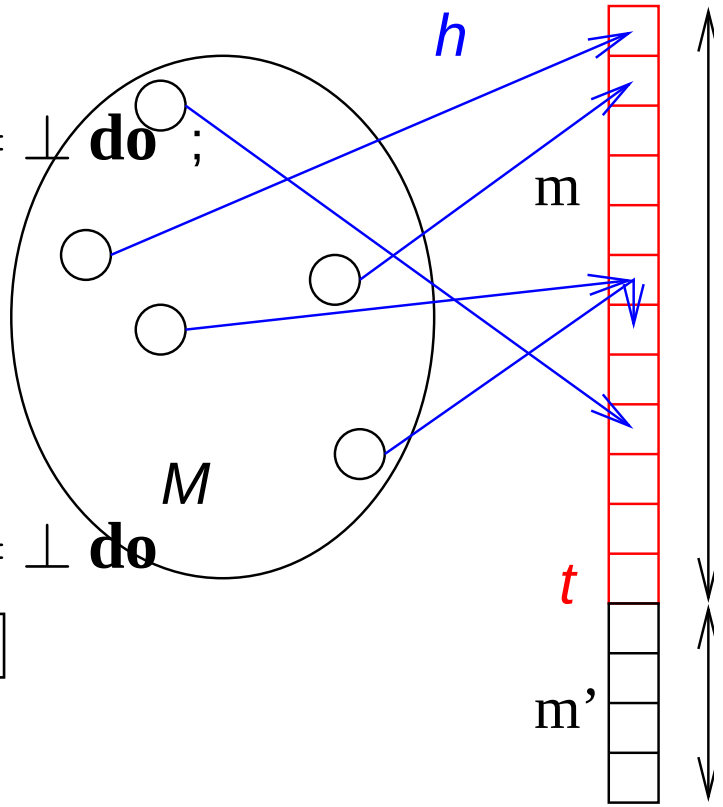
$t[i] := e$

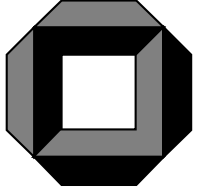
Function find$(k : \text{Key}) : \text{Element}$

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \bot$ **do**

if $t[i] = k$ **then return** $t[i]$

return $\bot$





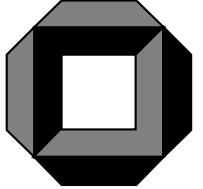
Remove

Beispiel: $t = [\ldots, \underset{h(z)}{x}, y, z, \ldots]$, remove$(x)$

invariant $\forall i : t[i] \neq \perp \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[i] \neq \perp$

Procedure remove$(k :$ Key$)$

```
for i := h(k) to ∞ while k ≠ t[i] do                    // search k
    if t[i] = ⊥ then return                              // nothing to do
//we plan for a hole at i.
for j := i+1 to ∞ while t[j] ≠ ⊥ do
    //Establish invariant for t[j].
    if h(t[j]) ≤ i then
        t[i] := t[j]                                     // Overwrite removed element
        i := j                                           // move planned hole
t[i] := ⊥                                                // erase freed entry
```

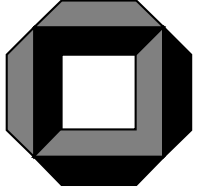


insert : axe, chop, clip, cube, dice, fell, hack, hash, lop, slash

	an	bo	cp	dq	er	fs	gt	hu	iv	jw	kx	ly	mz
tt	0	1	2	3	4	5	6	7	8	9	10	11	12
	⊥	⊥	⊥	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	slash	hack	fell	⊥

remove ⇩ clip

	⊥	⊥	chop	~~clip~~	axe	cube	dice	hash	lop	slash	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	~~lop~~	slash	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	slash	~~slash~~	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	slash	⊥	hack	fell	⊥



4.4 Verketten $\leftrightarrow$ Lineare Suche

Volllaufen: Verketten weniger empfindlich.

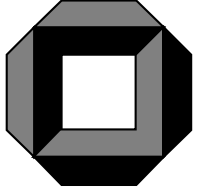
Unbeschränktes offenes Hashing hat nur amortisiert konst. Einfügezeit

Cache: Lineare Suche besser. Vor allem für doall

Platz/Zeit Abwägung: Kompliziert! Abhängig von n, Füllgrad, Elementgröße, Implementierungsdeteils bei Verketten (shared dummy!, t speichert Zeiger oder item), Speicherverwaltung bei Verketten, beschränkt oder nicht,...

Referentielle Integrität: Nur bei Verketten !

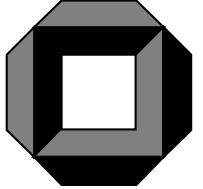
Leistungsgarantien: Universelles Hashing funktioniert so nur mit Verketten



4.5 Perfektes Hashing

Idee: mache h injektiv.

Braucht $\Omega(n)$ bits Platz !



Here: Fast Space Efficient Hashing

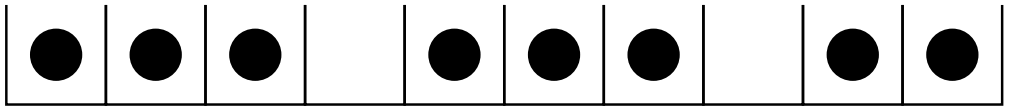
[Fotakis, Pagh, Sanders, Spirakis]

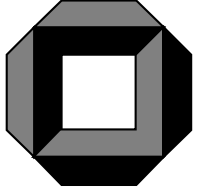
Represent a set of n elements (with associated information) using space $(1+\varepsilon)n$.

Support operations insert, delete, lookup, (doall) efficiently.

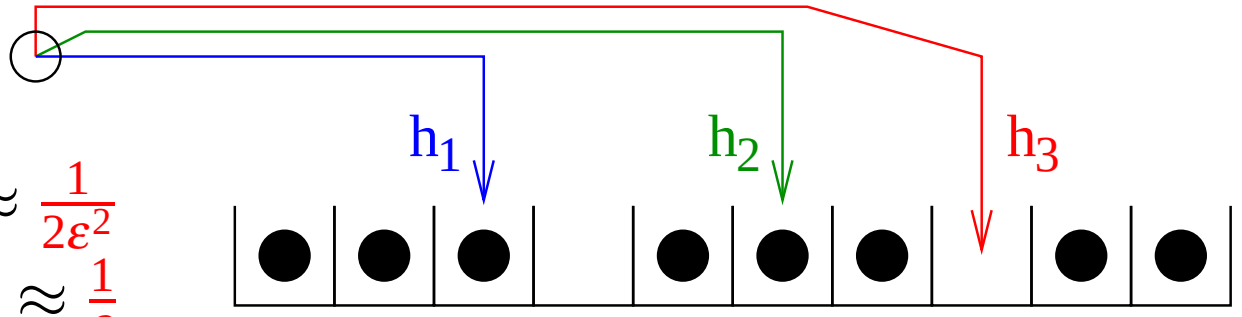
Assume a truly random hash function h

(Trick [Dietzfelbinger, Weidling 2005]: läßt sich rechtfertigen.)





Related Work



Linear probing: $E[T_{\text{find}}] \approx \frac{1}{2\varepsilon^2}$

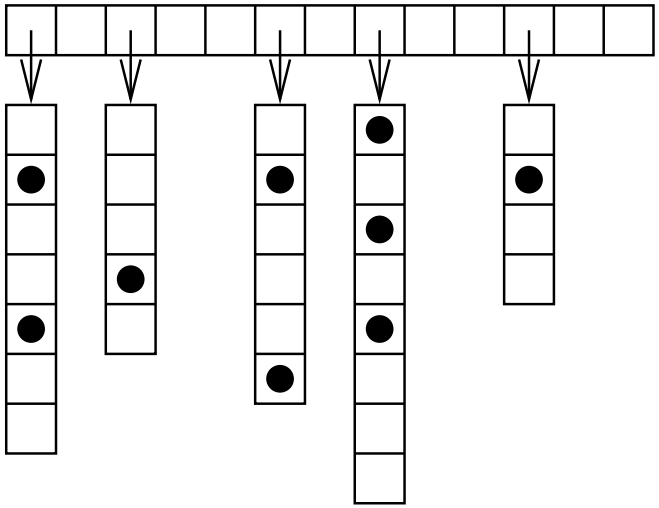
Uniform hashing: $E[T_{\text{find}}] \approx \frac{1}{\varepsilon}$

Dynamic Perfect Hashing,

[Dietzfelbinger et al. 94]

Worst case constant time

for lookup but ε is not small.



Approaching the Information Theoretic Lower Bound:

[Brodnik Munro 99,Raman Rao 02]

Space $(1+o(1))\times$lower bound without associated information

[Botelho Pagh Ziviani 2007] static case.

Cuckoo Hashing

[Pagh Rodler 01]

Table of size $(2+\varepsilon)n$.

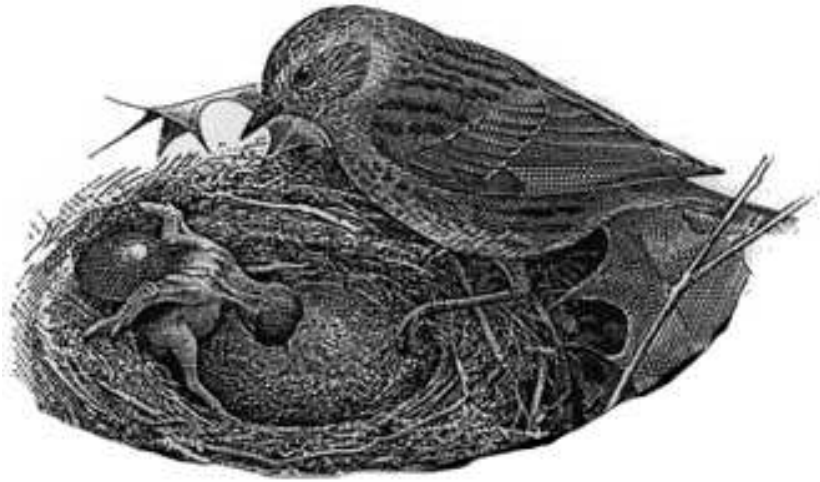
Two choices for each element.

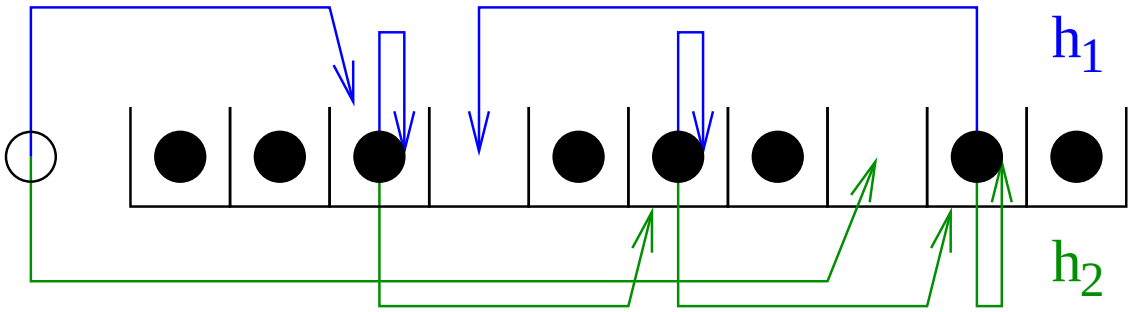
Insert moves elements;

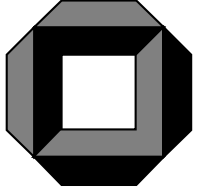
rebuild if necessary.

Very fast lookup and insert.

Expected constant insertion time.







d-ary Cuckoo Hashing

d choices for each element.

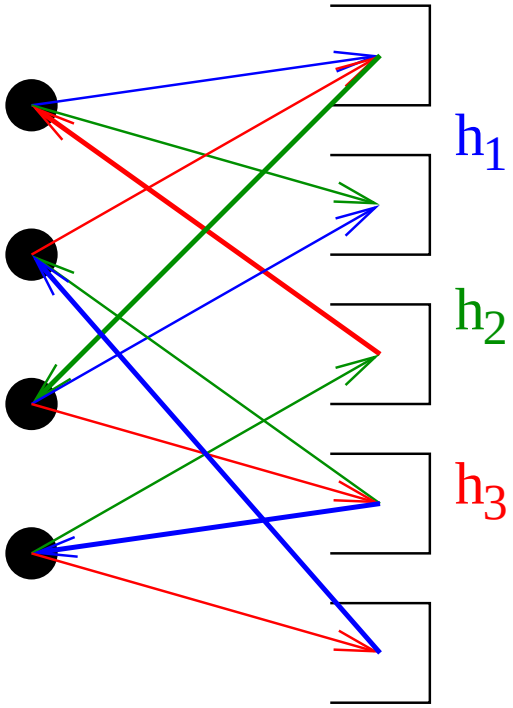
Worst case d probes for delete and lookup.

Task: maintain perfect matching

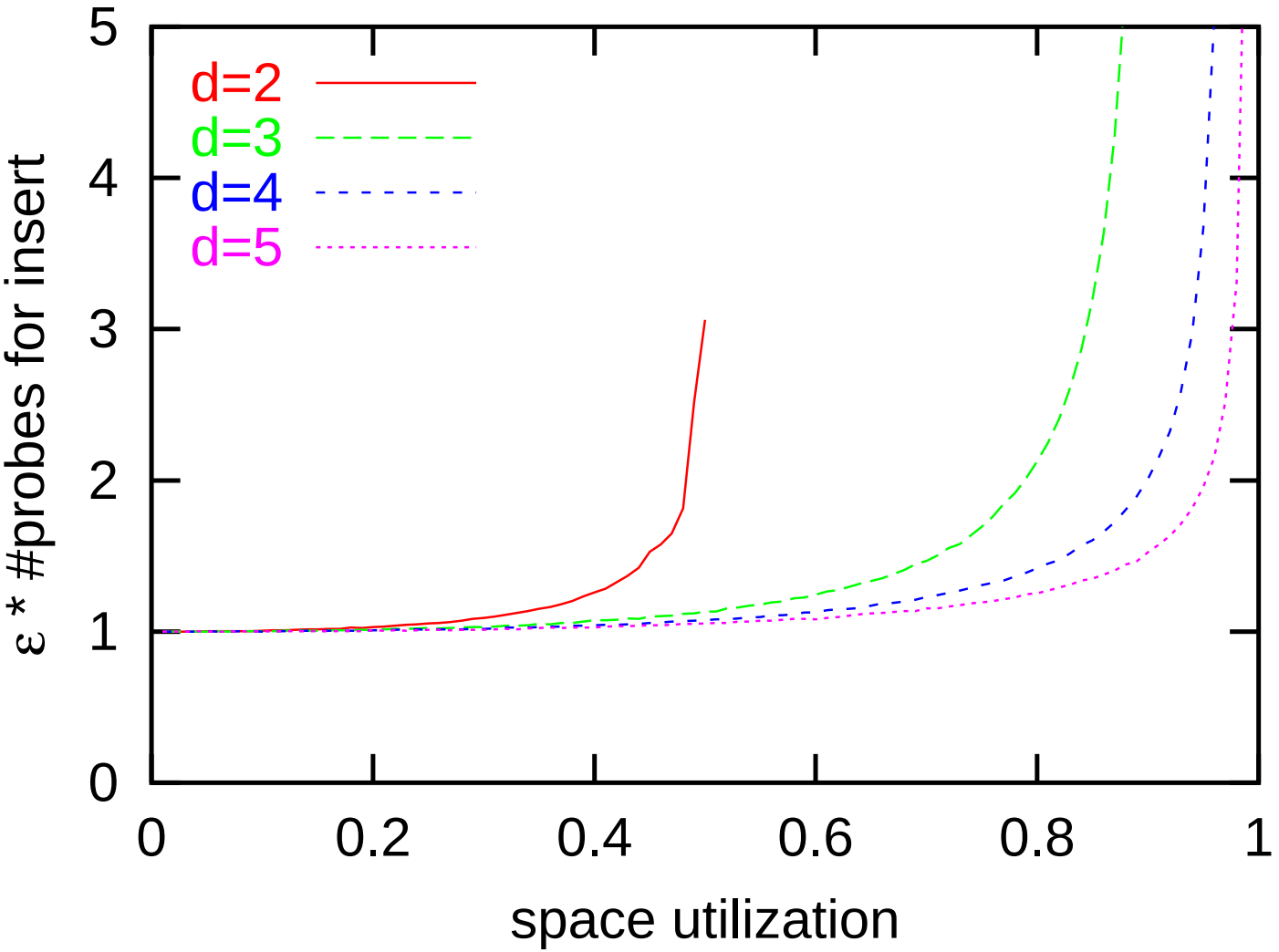
in the bipartite graph

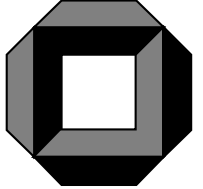
$(L = \text{Elements}, R = \text{Cells}, E = \text{Choices})$,

e.g., insert by BFS.



Experiments



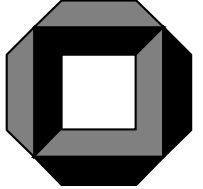


Analysis

Lookup and Delete: $d = \mathrm{O}\left(\log \frac{1}{\varepsilon}\right)$ probes

Insert: $\left(\frac{1}{\varepsilon}\right)^{\mathrm{O}(\log(1/\varepsilon))}$, (experiments) $\longrightarrow \mathrm{O}(1/\varepsilon)$?

Approach: Properties of random bipartite graphs / hypergraphs.



4.6 Mehr Hashing

- Hohe Wahrscheinlichkeit und Garantien für den schlechtesten Fall

  $\rightsquigarrow$ höhere Anforderungen and die Hash-Funktionen

- Hashing als Mittel zur Lastverteilung z.B., storage servers,

  (peer to peer Netze,. . .)

- Verschiedene Plattengrößen und Geschwindigkeiten

- Platten hinzufügen / ersetzen

- $O(1)$ find / perfektes Hashing