RUB

Vorlesung Datenstrukturen

Hashing

Maike Buchin

2. und 4.5.2017

Motivation

RUB

häufig werden Daten anhand eines numerischen *Schlüssel* abgespeichert

Beispiele:

- Studenten der RUB nach *Matrikelnummer*
- Kunden einer Bank nach *Kontonummer*
- Bürger einer Stadt nach *Ausweisnummer*

Problem:

- nur ein kleiner Teil der Schlüsselmenge wird genutzt
- Schlüsselmenge ist zu gross, um direkt für die Indizierung genutzt zu werden

Intuitiv: *Hashing* bildet Schlüsselmenge auf kleinere Menge ab, und versucht dabei *Kollisionen* zu vermeiden

Beispiel: Unibibliothek möchte Bücher aus dem Archiv in Regalfächern nach der Matrikelnummer bereitstellen.
Welchen Teil der Nummer verwendet sie?

Hashtabellen

Ziel: Hashtabellen *implementieren* den *Abstrakten Datentyp* Dictionary (dt. assoziatives Array)

Formal: Ein assoziatives Array enthält eine Menge von Einträgen S, und jedem Element $e \in S$ ist ein eindeutiger Schlüssel key$(e) \in Key$ zugeordnet.

- $S.build(\{e_1, \ldots, e_n\})$: $S := \{e_1, \ldots, e_n\}$.
- $S.insert(e : Element)$: Falls es ein $e' \in S$ mit $key(e') = key(e)$ gibt: $S := (S \setminus \{e'\}) \cup \{e\}$; andernfalls: $S := S \cup \{e\}$.
- $S.remove(x : Key)$: Falls es ein $e \in S$ mit $key(e) = x$ gibt: $S := S \setminus \{e\}$.
- $S.find(x : Key)$: Falls es ein $e \in S$ mit $key(e) = x$ gibt, dann gib e zurück, andernfalls gib $\perp$ zurück.

Sei stets $|S| := n$ und $|Key| = N$.

Wir interessieren uns für den Fall $n \ll N$.

Da sowohl n als auch N gross sind, suchen wir eine Lösung mit $O(n)$ Platzbedarf und idealerweise $O(1)$ Operationen.

Hashtabellen

RUB

Kurz: Menge S von Einträgen mit Schlüsseln $key(e) \in Key$
Wir interessieren uns für den Fall $|S| := n \ll N =: |Key|$,
und suchen eine Lösung mit $O(n)$ Platz und $O(1)$ Operationen

Dazu verwenden wir

- **Hashfunktion** $h : Key \to [0 \ldots m-1]$
- **Hashtabelle** array $t[0 \ldots m-1]$ zur Speicherung der Daten
- Element $e \in S$ wird der Wert $h(key(e))$, kurz $h(e)$, zugewiesen
- idealerweise wird $e \in S$ an Index $h(e)$ im Array a gespeichert $\to$ Zugriff in $O(1)$

Allerdings wird dies nicht immer möglich sein, sondern es werden Einträge *kollidieren*, d.h. den gleichen Hashwert haben.

Überblick

RUB

Um Kollisionen zu vermeiden, gibt es verschiedene (allgemeine) Lösungsansätze:

- Hashing mit Verkettung [engl. chaining]
- Hashing mit offener Adressierung [engl. open addressing]

Zusätzlich stellt sich die Frage, wie eine geeignete Hashfunktion zu wählen ist

- universelle Hashfunktionen

4.1 Hashing mit Verkettung

RUB

Einträge der Hashtabelle werden als *verkettete Liste* gespeichert

- jede Zelle $t[\cdot]$ speichert eine verkette Liste
- jedes Element e wird in der Liste in $t[h(e)]$ gespeichert

Operationen:

- find(x)
- remove(x)
- insert(e)

} durchsuche jeweils Folge $t[h(x)]$

- build$(e_1, \ldots, e_n)$ — n Einfügeoperationen

Analyse:

- Operationen im schlimmsten Fall $O(n)$
- Platzbedarf $O(n + m)$

4.1 Hashing mit Verkettung

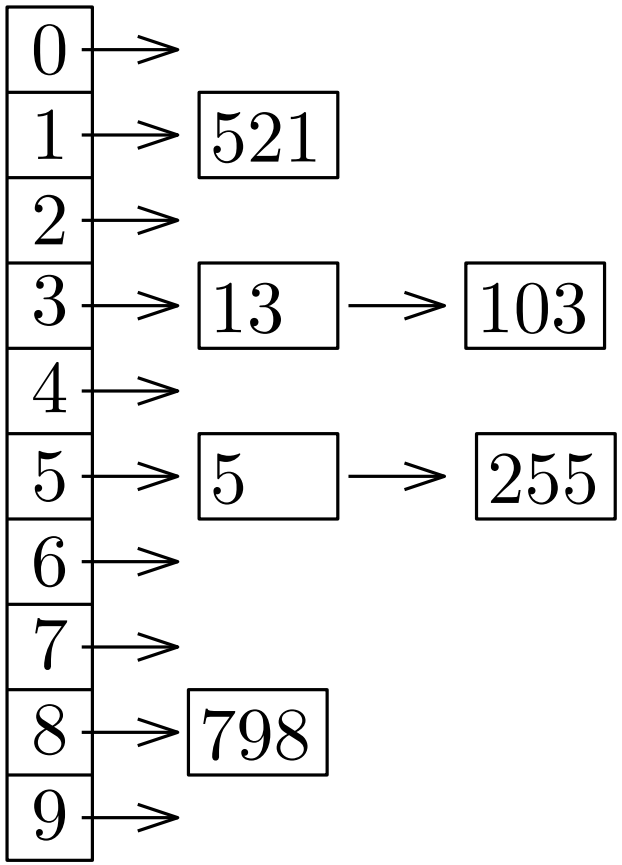
RUB

Einträge der Hashtabelle werden als *verkettete Liste* gespeichert

- jede Zelle $t[\cdot]$ speichert eine verkette Liste
- jedes Element e wird in der Liste in $t[h(e)]$ gespeichert

Bsp.:

- sei $m = 10$, $S = 0, \ldots, 999$ und $h(e) = e \mod 10$
- füge $13, 5, 255, 798, 521, 103$ in eine leere Hashtabelle ein



4.1 Hashing mit Verkettung

RUB

Frage: Gibt es Hashfunktionen, die kurze Listen garantieren?

Antwort: Nein, kann es nicht geben.

Angenommen N Elemente werden auf eine Tabelle der Größe m abgebildet. Dann muss es Plätze in der Tabelle geben, auf die $\leq N/m$ Elemente abgebildet werden. Häufig gilt $n < N/m$ und dann entartet Hashing mit Verkettung zu linearer Suche.

Ansätze:

- Durchschnittsanalyse
- Randomisierung
- *Perfektes Hashing* für statische Schlüsselmenge

Durchschnittsanalyse

unter Annahme von *Simple Uniform Hashing*

RUB

Sei H die Menge aller Funktionen von $Key \to \{0, \ldots, m-1\}$.
Wir nehmen an, dass Hashfkt. h zufällig aus H gewählt wird.

Achtung: Dies ist keine sinnvolle Annahme.

Da es m^N Funktionen in H gibt, würden bereits $N \log m$ bits benötigt, um eine Funktion in H zu benennen.
Damit ist ein Speicherbedarf von $n \ll N$ nicht zu erreichen.

Durchschnittsanalyse

unter Annahme von *Simple Uniform Hashing*

RUB

Satz 4.1 Wenn n Einträge in einer Hashtabelle mit m Tabellenplätzen gespeichert sind, und wir Hashing mit Verkettung benutzen, dann besitzen die Operationen insert, remove und find erwartete Ausführungszeit $O(1 + n/m)$.

Beweis: Für festen Schlüssel x ist die Ausführungszeit $O(1 + E[X])$, wobei die Zufallsvariable X die Länge der Folge $t[h(x)]$ angibt.
Für jedes $e \in S$ sei X_e die Indikator-Zufallsvariable $X_e = [h(e) = h(x)]$.
Zwei Fälle:
$i)$ Wenn es kein $e \in S$ gibt mit $key(e) = key(x)$, dann ist $X = \sum_{e \in S} X_e$.
$ii)$ Wenn es ein $e_0 \in S$ gibt mit $key(e_0) = key(x)$, dann ist $X = 1 + \sum_{e \in S \setminus \{e_0\}} X_e$.

Im ersten Fall:

$$E[X] = E[\sum_{e \in S} X_e] = \sum_{e \in S} E[X_e] = \sum_{e \in S} P(X_e = 1) = \sum_{e \in S} 1/m = n/m$$

Im zweiten Fall ergibt sich analog $E[X] = 1 + (n-1)/m < 1 + n/m$.

Durchschnittsanalyse

unter Annahme von *Simple Uniform Hashing*

RUB

Satz 4.1 Wenn n Einträge in einer Hashtabelle mit m Tabellenplätzen gespeichert sind, und wir Hashing mit Verkettung benutzen, dann besitzen die Operationen insert, remove und find erwartete Ausführungszeit $O(1 + n/m)$.

Folgerung: Wir können linearen Platzbedarf und konstante erwartete Ausführungszeit der Operationen durch $m = \Theta(n)$ erreichen. Dabei passen wir die Tabellengröße wie für unbeschränkte Arrays an.

4.2 Universelles Hashing

RUB

Nun betrachten wir Familien von Hashfunktionen, bei denen man eine Funktion daraus in konstanten Platz beschreiben kann.

Definition: Sei $c > 0$ eine Konstante. Eine Familie H von Funktionen von $Key \to [0 \ldots m-1]$ heißt $c-$*universell*, wenn $\forall x, y \in Key$ mit $x \neq y : |\{h \in H : h(x) = h(y)\}| \leq c/m|H|$.
Äquivalent: $P[h(x) = h(y)] \leq c/m$, wenn h aus H zufällig.

Satz: Wird bei Hashing mit Verkettung die Hashfunktion zufällig aus einer c-universellen Klasse gewählt, dann haben die Operationen insert, remove und find erwartete Ausführungszeit $O(1 + cn/m)$.

Beweis: Analog zum vorherigen Beweis. Nur jetzt wird h aus H gewählt, daher $P[X_e = 1] \leq c/m$ also $E[X] \leq cn/m$

1-universelle Klasse von Hashfunktionen

RUB

- Wir nehmen an, dass die Tabellengröße m prim ist. (Denn dann ist $\mathbb{Z}_m = \{0, \ldots, m-1\}$ ein Körper.)
- Sei $w := \lfloor \log m \rfloor$.
- Teile Schlüssel in Stücke von w Bits.
- Fasse jedes Stück als (Binärdarstellung einer) Zahl in $\{0, \ldots, 2^w - 1\} \subseteq \{0, \ldots, m-1\}$ auf.
- Ein Schlüssel ist ein $k-$Tupel solcher Zahlen: $x = (x_1, \ldots, x_k)$ mit $x_i \in \{0, \ldots, 2^w - 1\}$
- Für ein k-Tupel $a = (a_1, \ldots, a_k) \in (0, \ldots, m)^k$ definieren wir die Hashfunktion:

$$h_a(x) = a \cdot x \mod m.$$

Also das Skalarprodukt von a und x über dem Körper $\mathbb{Z}_m$.

1-universelle Klasse von Hashfunktionen

Beispiel:

Sei $m = 17$, also $w = \log m = 4$, und $k = 3$.
Hashfunktion wird durch ein $a \in \{0, \ldots, m-1\}^k$ bestimmt.
Sei z.B. $a = (2, 7, 16)$ und $x = (11, 6, 3)$. Dann ist:

$$h_a(x) = (2 \cdot 11 + 7 \cdot 6 + 16 \cdot 3) \mod 17$$

$$= (22 + 42 + 48) \mod 17 = 10$$

1-universelle Klasse von Hashfunktionen

RUB

Satz: $$H^{SP} = \{h_a : a \in (0 \ldots m-1)^k\}$$

ist eine 1-universelle Klasse von Hashfunktionen, falls m prim ist.

Beweis:

- seine $x = (x_1, ..., x_k)$ und $y = (y_1, ..., y_k)$ zwei verschiedene Schlüssel
- wähle ein $h_a \in H^{SP}$ zufällig
- wir zeigen, dass für genau m^{k-1} viele Vektoren a gilt $P[h_a(x) = h_a(y)]$
- also $P[h_a(x) = h_a(y)] = m^{k-1}/m^k = 1/m$
- wähle einen Index j mit $x_j \neq y_j$
- dann ist $x_j - y_j \not\equiv 0 \mod m$
- also kann jede Gleichung der Form $a_j(x_j - y_j) \equiv b \mod m$ mit $b \in \mathbb{Z}_m$ eindeutig aufgelöst werden
- folgende Rechnung zeigt, dass für jede Wahl der a_i's mit $i \neq j$ genau eine Wahl für a_j die Gleichheit $h_a(x) = h_a(y)$ erfüllt

$$h_a(x) = h_a(y) \Leftrightarrow \sum_i a_i x_i \equiv \sum_i a_i y_i \mod m$$
$$\Leftrightarrow a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j} a_i(y_i - x_i) \mod m$$

4.3 Hashing mit linearem Sondieren

RUB

Offene Adressierung: Speicherung nur in Plätzen der Hashtabelle selbst, dafür muss ein Element nicht mehr zwingenderweise an der Stelle $t[h(e)]$ gespeichert werden

Mehrere Methoden hierfür:

- lineares Sondieren
- quadratisches Sondieren
- doppeltes Hashing

4.3 Hashing mit linearem Sondieren

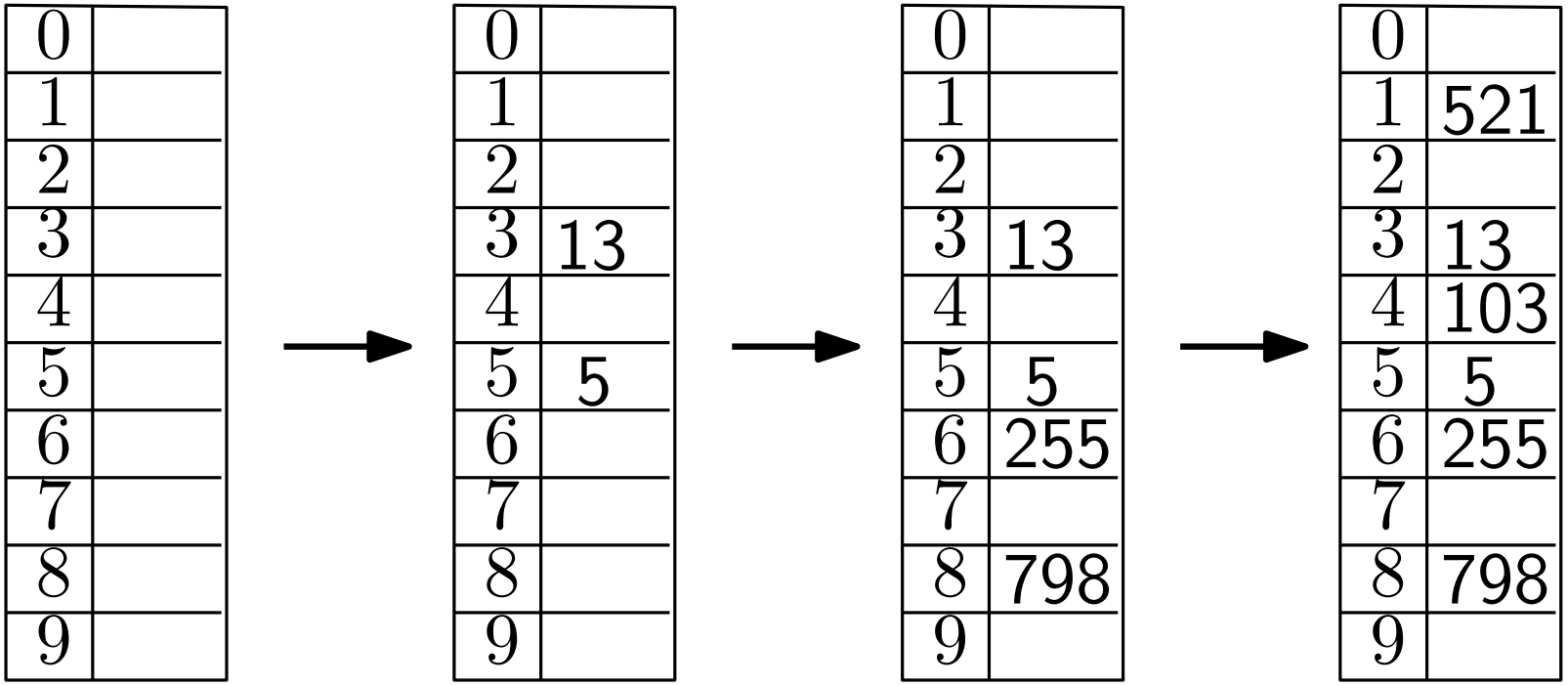
Lineares Sondieren: Wenn ein Tabellenplatz belegt ist, gehe weiter zum nächsten bis ein freier gefunden.

Operationen:

- insert und find: suche im Block startend bei $t[h(e)]$
- remove: markiere gelöschte Felder oder fülle Lücken

Bsp.:

- sei $m = 10$, $S = 0, \ldots, 999$ und $h(e) = e \mod 10$
- füge $13, 5, 255, 798, 521, 103$ in eine leere Hashtabelle ein



4.3 Hashing mit linearem Sondieren

RUB

Lineares Sondieren: Wenn ein Tabellenplatz belegt ist, gehe weiter zum nächsten bis ein freier gefunden.

Operationen:

- insert und find: suche im Block startend bei $t[h(e)]$
- remove: markiere gelöschte Felder oder fülle Lücken

Invariante: Ist e in $t[i]$ mit $i > h(e)$ gespeichert, dann sind alle Plätze von $h(e)$ bis $i-1$ belegt.

Lücken füllen: Nach Löschen an Stelle i:

$j := i+1$
while $j < m$ do
  if $t[j]$.isEmpty return
  else $f := t[j]$
    if $h(f) \leq i$ do
      $t[i] = f; t[j] =$ empty; $i := j;$
    $j++$

4.7 Analyse

Sei $\alpha = n/m$. Die erwartete Anzahl T_{fail} von Zugriffen bei einer erfolglosen Suche und die erwartete mittlere Anzahl T_{success} von Zugriffen bei einer erfolgreichen Suche ist:

$$T_{\text{fail}} \approx \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right) \quad \text{und} \quad T_{\text{success}} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

Für α nahe an 1 ergeben sich also hohe Zugriffszeiten.

Im Vergleich: Hashing mit Verkettung mit einer zufälligen Hashfkt aus einer 1-universellen Hashklasse hat erwartete Ausführungszeit $O(1+\alpha)$.
Allerdings: für $m = cn = O(n)$ ist $\alpha = 1/c$ constant.

4.4 Vergleich

RUB

Hashing mit Verkettung	Hashing mit linearem Sondieren
+ referentielle Integrität	− referentielle Integrität nicht bei Verschieben
− extra Zeiger	+ zush. Speichersegmente
	− hohe Suchzeiten bei hoher Auslastung

referentielle Integrität bedeutet, dass sich die Position eines Eintrages im Speicher nicht mehr ändert

4.5 Perfektes Hashing

RUB

Motivation: bisherige Verfahren garantieren "nur" erwartet $O(1)$ Operationen; dies reicht nicht immer aus.

Perfektes Hashing: keine Kollisionen, d.h. die Hashfunktion ist injektiv.

Frage: Wie findet man für eine beliebige Schlüsselmenge eine perfekte Hashfunktion?

Jetzt: Konstruktion für statische Schlüsselmenge.

4.5 Perfektes Hashing

RUB

Sei $S = \{x_1, \ldots, x_n\}$ die gegebene Menge von Schlüsseln.
Sei H_m eine c-universelle Hashfunktion auf $[0 \ldots m-1]$.
Bekannt: Für jedes m gibt es solche Klassen bereits für $c = 1$.

Für $h \in H_m$ bezeichne $C(h)$ die Anzahl der Kollisionen, d.h.
$C(h) = |\{(x, y) : x, y \in S, x \neq y, h(x) = h(y)\}|$.
Für $h \in H_m$ zufällig gewählt ist C eine Zufallsvariable.
Zunächst schätzen wir $E[C]$ nach oben ab.

4.5 Perfektes Hashing

RUB

Lemma 4.5 Es gilt $E[C] \leq cn(n-1)/m$. Für mindestens die Hälfte der Hashfunktionen $h \in H_m$ gilt $C(h) < 2cn(n-1)/m$.

Beweis:
Definiere $n(n-1)$ Indikator-Zufallsvariablen $X_{ij} = [h(x_i) = h(x_j)]$.
Dann ist $C(h) = \sum_{i \neq j} X_{ij}(h)$ und daher

$$E[C] = E[\sum_{i \neq j} X_{ij}] = \sum_{i \neq j} E[X_{ij}] = \sum_{i \neq j} P[X_{ij} = 1] \leq n(n-1)c/m$$

Die zweite Aussage folgt aus der Markovschen Ungleichung.

Lemma 4.6 Wenn $m \geq cn(n-1)$, dann sind mindestens die Hälfte der $h \in H_m$ injektiv auf S.

Beweis: Nach vorherigem Lemma gilt $C(h) < 2$ für mind. die Hälfte der $h \in H_m$. Da $C(h)$ eine gerade Zahl ist, folgt $C(h) = 0$.

4.5 Perfektes Hashing

RUB

Wahl einer Hashfkt mit quadratischem Platz

Verfahren:

- wähle $m \geq c \cdot n(n-1)$
- wähle $h \in H_m$ zufällig
- prüfe ob h injektiv ist
- falls nicht wiederhole

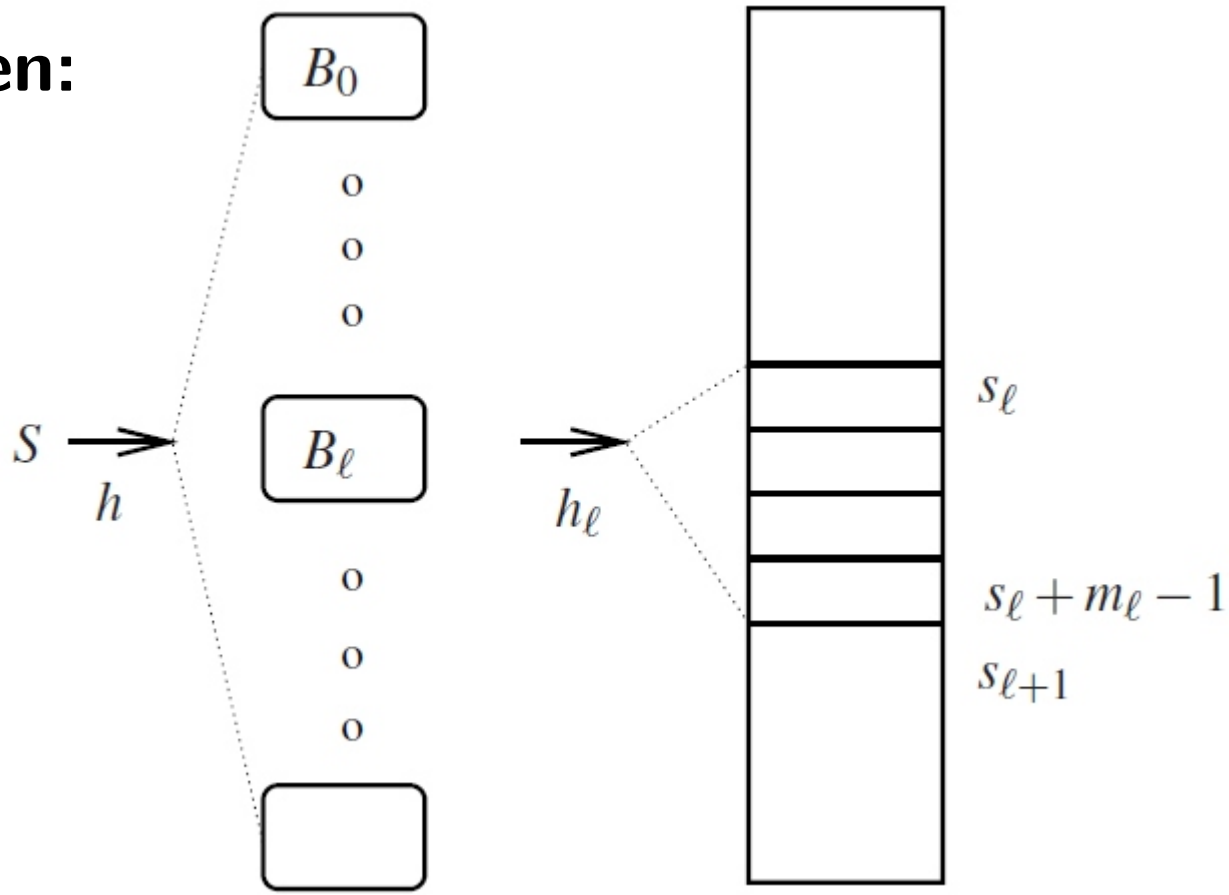
Im Mittel nicht mehr als zwei Versuche nötig (wg. Lm 4.6).

Resultat: Perfekte Hashfkt in Tabelle quadratischer Größe.

4.5 Perfektes Hashing

Reduktion des Platzbedarfs auf $O(n)$.

Zweistufiges Vorgehen:



1. Stufe: Zerlege Schlüsselmenge so in Behälter, dass der Mittelwert der Quadrate der Behältergrößen konstant ist

2. Stufe: Verwende für jeden Behälter quadratischen Speicherplatz

4.5 Perfektes Hashing

RUB

Reduktion des Platzbedarfs auf $O(n)$.

Sei weiterhin $h \in H_m$ aus einer c-universeller Hashklasse.
Für $\ell \in \{0, \ldots, m-1\}$ und $h \in H_m$ bezeichne B_ℓ^h die Schlüssel in S, die von h auf Behälter ℓ abgebildet werden; sei $|B_\ell^h| =: b_\ell^h$.

Lemma 4.7 Für jedes $h \in H_m$ gilt $C(h) = \sum_\ell b_\ell^h (b_\ell^h - 1)$.

Beweis:

Für jedes ℓ erzeugen die Schlüssel in B_ℓ^h genau $b_\ell^h(b_\ell^h - 1)$ Kollisionen.

4.5 Perfektes Hashing

RUB

Reduktion des Platzbedarfs auf $O(n)$.

1. sei $\alpha \geq 1$ eine (noch zu wählende) Konstante und $m = \lceil \alpha n \rceil$
   - wähle eine Hashfkt $h \in H_m$ mit $C(h) \leq 2cn(n-1)/m$
   - für $\ell = 0 \ldots m-1$ sei $B_\ell := \{e \in S | h(e) = \ell\}$ und $b_\ell = |B_\ell|$
2. - für $\ell = 0 \ldots m-1$ sei $m_\ell := \lceil c \cdot b_\ell(b_\ell - 1) \rceil \leq c \cdot b_\ell(b_\ell - 1) + 1$
   - wähle eine injektive Hashfkt $h_\ell \in H_{m_\ell}$
   - h_ℓ bildet B_ℓ auf eine Tabelle der Größe m_ℓ ab

$+$

- wir "stapeln" diese Tabelle aufeinander zu einer großen Tabelle der Größe $\sum_\ell m_\ell$
- in dieser beginnt die Teiltabelle für B_ℓ an der Position $s_\ell = m_0 + \ldots + m_{\ell-1}$
- unsere perfekte Hashfkt wird berechnet durch

$$\ell := h(x);\ \text{return}\ s_\ell + h_\ell(x)$$

4.5 Perfektes Hashing

Reduktion des Platzbedarfs auf $O(n)$.

Die Werte der Hashfkt sind beschränkt durch

$$\sum_{\ell} m_\ell - 1 \leq \sum_{\ell} (c b_\ell (b_\ell - 1) + 1) - 1 \leq \alpha n + c C(h) \leq n(\alpha + 2c^2/\alpha)$$

Dies wird minimiert für $\alpha = \sqrt{22}$ und $c = 1$ und ergibt $n\sqrt{22}$.

Satz 4.8 Für jede Menge S von n Schlüsseln kann eine perfekte Hashfkt in $[0 \ldots 2\sqrt{2}n]$ in erwartet linearer Zeit berechnet werden.