11. Hash-Tabellen

Hash Tabellen, Geburtstagsparadoxon, Hashfunktionen,
Kollisionsauflösung durch Verketten, offenes Hashing, Sondieren

Motivation

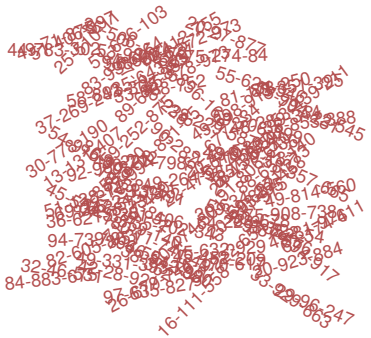
Ziel: Tabelle aller n Studenten dieser Vorlesung

Anforderung: Schneller Zugriff per Name oder Legi-Nr

Motivation : Legi $\leftrightarrow$ Name ?

Tabelle: Legi $\rightarrow$ Name

keys



table

26-398-736	Corinne
72-666-564	Odysseas
77-279-538	Nicolas
20-70-371	Ricco
97-333-579	Josianne
17-503-670	Cem
22-114-797	Lucas
56-27-960	Sandrine
50-846-549	Jonas
53-400-549	Daniel
37-263-200	Julian
49-558-111	Lenard
16-304-218	Malou
26-379-523	Dominic
56-39-968	Victor
27-817-212	Stefano

Schlüsseluniversum: 100.000.000 mögliche Schlüsselwerte

Tabelle: Name $\rightarrow$ Legi

keys

table

Jessica	70-534-949
Antoine	71-875-287
Tamara	44-366-474
Jérôme	13-645-632
Joël	29-242-212
Kerstin	69-826-960
Enrico	25-506-11
Saskia	18-441-662
Joshua	96-77-509
Gianluca	78-899-25
Davide	10-239-900
Kim	54-277-635
Raffael	17-605-302
Corinne	61-836-416
Odysseas	26-679-744
Nicolas	62-365-358

Schlüsseluniversum: konzeptuell unendlich viele Schlüsselwerte

Plan

- Überlegen uns zuerst eine Strategie zur Abbildung Schlüsselmenge (Legi-Nr oder Name) $\rightarrow$ Indizes (ganze Zahlen) Wir nennen diese Abbildung eine *Hashfunktion*.
- Dann schränken wir den Wertebereich der Hashfunktionen ein: Schlüsselmenge $\rightarrow \{0, \ldots, m-1\}$.
  Damit können wir auf der Basis eines Arrays eine *Hashtabelle* implementieren.
- Es entstehen *Kollisionen*. Wir überlegen uns, wie wahrscheinlich das ist und wie viele Kollisionen wir erwarten.
- Zuletzt klären wir, wie wir Kollisionen auflösen.

Naive Ideen

Zuordnung Name $s = s_1 s_2 \ldots s_{l_s}$ zu Schlüssel

$$k(s) = \sum_{i=1}^{l_s} s_i \cdot b^i$$

b gross genug, so dass verschiedene Namen verschiedene Schlüssel erhalten.

Speichere jeden Datensatz an seinem Index in einem grossen Array.

Beispiel, mit $b = 100$. Ascii-Werte s_i.

Anna $\mapsto 71111065$
Jacqueline $\mapsto 102110609021813999774$

Unrealistisch: erfordert zu grosse Arrays.

Bessere Idee?

Allokation eines Arrays der Länge m ($m > n$).

Zuordnung Name s zu

$$k_m(s) = \left(\sum_{i=1}^{l_s} s_i \cdot b^i \right) \bmod m.$$

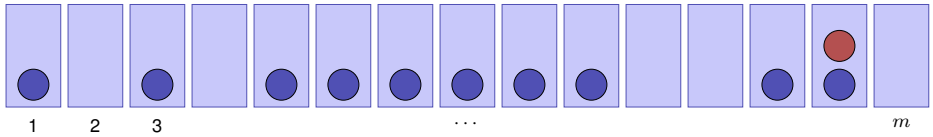
Verschiedene Namen können nun denselben Schlüssel erhalten (“Kollision”). Und dann?

Abschätzung

Vielleicht passieren Kollisionen ja fast nie. Wir schätzen ab ...

Abschätzung

Annahme: m Urnen, n Kugeln (oBdA $n \leq m$).
n Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die Kollisionswahrscheinlichkeit?

Sehr verwandte Frage: Bei wie vielen Personen (n) ist die Wahrscheinlichkeit, dass zwei am selben Tag ($m = 365$) Geburtstag haben grösser als 50%?

[Abschätzung]

$\mathbb{P}(\text{keine Kollision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Sei $a \ll m$. Mit $e^x = 1 + x + \frac{x^2}{2!} + \ldots$ approximiere $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$.
Damit:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\cdots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Es ergibt sich

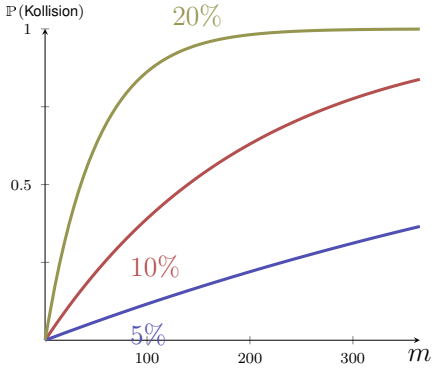
$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Auflösung zum Geburtstagsparadoxon: Bei 23 Leuten ist die Wahrscheinlichkeit für Geburstagskollision 50.7%. Zahl stammt von der leicht besseren Approximation via Stirling Formel.

Mit Füllgrad

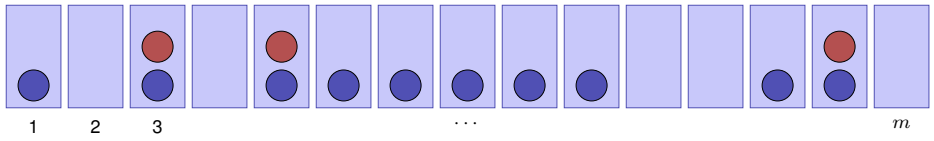
Mit Füllgrad $\alpha := n/m$ ergibt sich (weiter vereinfacht)

$$\mathbb{P}(\text{Kollision}) \approx 1 - e^{-\alpha^2 \cdot \frac{m}{2}}.$$



Andere Frage

Annahme: m Urnen, n Kugeln (oBdA $n \leq m$).
n Kugeln werden gleichverteilt in Urnen gelegt.



Wie gross ist die erwartete Anzahl von Kollisionen?

Erwartete Anzahl Kollisionen

- $\mathbb{P}(\text{Kugel } B \text{ trifft Kugel } A_i) = 1/m$.
- $\mathbb{P}(\text{Kugel } B \text{ trifft Kugel } A_i \text{ nicht}) = 1 - 1/m$.
- $\mathbb{P}(n-1 \text{ Kugeln treffen } A_i \text{ nicht}) = (1-1/m)^{n-1}$.
- $\mathbb{P}(A_i \text{ getroffen}) = 1 - (1-1/m)^{n-1}$.
- Sei X_i Zufallsvariable mit $X_i = \mathbb{1}_{A_i \text{getroffen}}$
- $\mathbb{E}(\sum X_i) = \sum \mathbb{E}(X_i)$
- $\mathbb{E}(\text{Anzahl getroffene Kugeln}) = n(1-(1-1/m)^n) \approx \frac{n^2}{2m}$.

Nomenklatur

Hashfunktion h: Abbildung aus der Menge der Schlüssel $\mathcal{K}$ auf die Indexmenge $\{0, 1, \ldots, m-1\}$ eines Arrays (*Hashtabelle*).

$$h : \mathcal{K} \to \{0, 1, \ldots, m-1\}.$$

Meist $|\mathcal{K}| \gg m$. Es gibt also $k_1, k_2 \in \mathcal{K}$ mit $h(k_1) = h(k_2)$ (*Kollision*).

Eine Hashfunktion sollte die Menge der Schlüssel **möglichst gleichmässig** auf die Positionen der Hashtabelle verteilen.

Implementation Hashfunktion (String) in Java

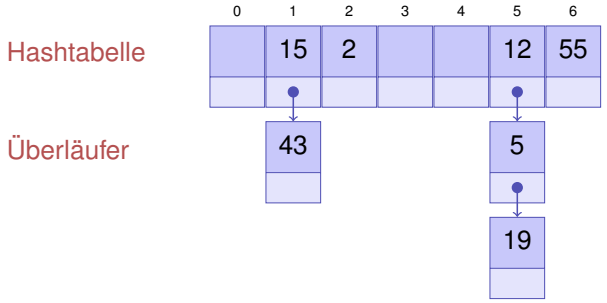
$$h_{b,m}(s) = \left(\sum_{i=0}^{l-1} s_i \cdot b^i \right) \bmod m$$

```
int ComputeHash(int m, String s) {
  int sum = 0;
  int b = 1;
  for (int k = 0; k<s.length(); ++k){
    sum = (sum + s.charAt(k) * b) % m;
    b = (b * 31) % m;
  }
  return sum;
}
```

Behandlung von Kollisionen

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

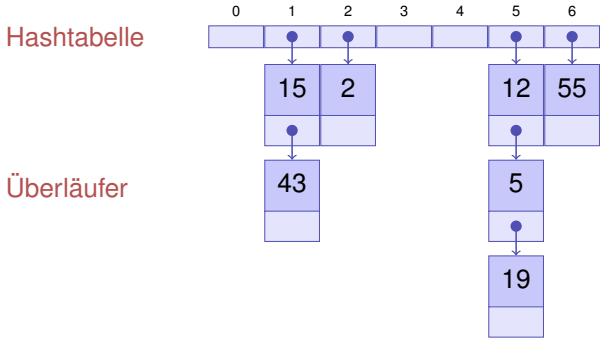
Verkettung der Überläufer



Behandlung von Kollisionen

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 55 , 5 , 15 , 2 , 19 , 43

Direkte Verkettung der Überläufer



Algorithmen zum Hashing mit Verkettung

- `contains`(k) Durchsuche Liste an Position $h(k)$ nach k. Gib wahr zurück, wenn gefunden, sonst falsch.
- `put`(k) Prüfe ob k in Liste an Position $h(k)$. Falls nein, füge k am Ende der Liste ein. Andernfalls Fehlermeldung.
- `get`(k) Prüfe ob k in Liste an Position $h(k)$. Falls ja, gib die Daten zum Schlüssel k zurück. Andernfalls Fehlermeldung
- `remove`(k) Durchsuche die Liste an Position $h(k)$ nach k. Wenn Suche erfolgreich, entferne das entsprechende Listenelement.

Vor und Nachteile

Vorteile der Strategie:

- Belegungsfaktoren $\alpha > 1$ möglich
- Entfernen von Schlüsseln einfach

Nachteile

- Speicherverbrauch der Verkettung

Offene Hashverfahren

Speichere die Überläufer direkt in der Hashtabelle mit einer *Sondierungsfunktion* $s(j,k)$ $(0 \leq j < m,\ k \in \mathcal{K})$

Tabellenposition des Schlüssels entlang der *Sondierungsfolge*

$$S(k) := ((h(k) + s(0,k)) \bmod m, \ldots, (h(k) + s(m-1,k)) \bmod m)$$

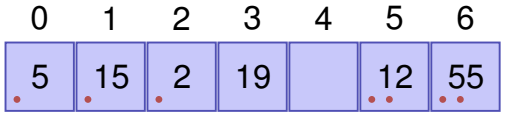
Algorithmen zum Open Addressing

- `contains`(k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib `true` zurück. Ist die Sondierungsfolge zu Ende oder eine leere Position erreicht, gib `false` zurück.
- `put`(k) Suche k in der Tabelle gemäss $S(k)$. Ist k nicht vorhanden, füge k an die erste freie Position in der Sondierungsfolge ein. Andernfalls Fehlermeldung.
- `get`(k) Durchlaufe Tabelleneinträge gemäss $S(k)$. Wird k gefunden, gib die zu k gehörenden Daten zurück. Andernfalls Fehlermeldung.
- `remove`(k) Suche k in der Tabelle gemäss $S(k)$. Wenn k gefunden, ersetze k durch den speziellen `removed` key.

Lineares Sondieren

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

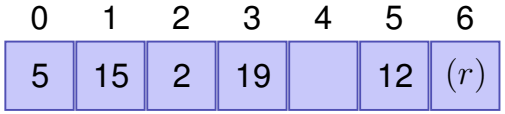
Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 55 , 5 , 15 , 2 , 19



Löschen beim Open Addressing

- Ein leeres Feld zeigt das Ende einer Sondierungsfolge an.
- Wird ein Element entfernt, indem das Feld geleert wird, könnte eine Sondierungsfolge unterbrochen werden.
- Daher wird ein Spezialeintrag “removed” verwendet, um anzuzeigen, dass die Sondierungsfolge nicht notwendigerweise zu Ende ist, das Feld jedoch als frei gilt.

Beispiel: Lösche Eintrag 55 am Index 6:



Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 200 Tabelleneinträge!

? Was ist ein mögliches Problem?

! *Primäre Häufung:* Ähnliche Hashaddressen haben ähnliche Sondierungsfolgen $\Rightarrow$ lange zusammenhängende belegte Bereiche.

Quadratisches Sondieren

$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$
$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
19	15	2		5	12	55

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 22 Tabelleneinträge

? Was ist ein mögliches Problem?

! *Sekundäre Häufung:* Synonyme k und k' (mit $h(k) = h(k')$) durchlaufen dieselbe Sondierungsfolge.

Double Hashing

Zwei Hashfunktionen $h(k)$ und $h'(k)$. $s(j,k) = j \cdot h'(k)$.
$S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), \dots, h(k)+(m-1)h'(k)) \mod m$

Beispiel:
$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.
Schlüssel 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
5	15	2	19		12	55

Double Hashing

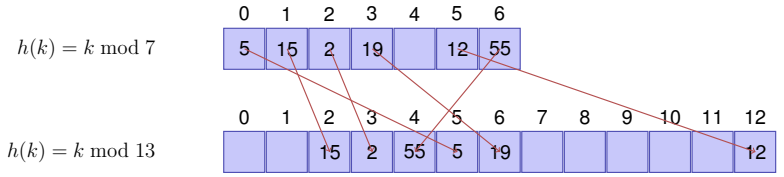
- Sondierungsreihenfolge muss Permutation aller Hashadressen bilden. Also $h'(k) \neq 0$ und $h'(k)$ darf m nicht teilen, z.B. garantiert mit m prim.
- h' sollte unabhängig von h sein (Vermeidung sekundärer Häufung).

Unabhängigkeit gilt zum Beispiel, wenn $h(k) = k \bmod m$ und $h'(k) = 1 + k \bmod (m-2)$, m Primzahl.

Dynamische Hashtabellen

- Wenn eine Hashtabelle zur Laufzeit wachsen soll, muss der Inhalt in eine grössere Tabelle kopiert werden.
- Hashfunktion ändert sich; jeder Eintrag der alten Hashtabelle muss neu zugeteilt werden (“re-hashing”).

Beispiel (mit linearem Sondieren):



Generische Hashtabellen in Java `java.util.HashMap`

```
import java.util.HashMap;
...

// Map String --> Integer
HashMap<String, Integer> map = new HashMap<String,Integer>();

map.put("abc",3);
map.put("xyz",100);
int i = map.get("abc"); // i = 3
int j = map.get("xyz"); // j = 100
```