17. Hashing

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

Motivation

Ziel: Tabelle aller n Studenten dieser Vorlesung

Anforderung: Schneller Zugriff per Name

Naive Ideen

Zuordnung Name $s = s_1 s_2 \dots s_{l_s}$ 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 $\mapsto 1021106090218139997 74$

Unrealistisch: erfordert zu grosse Arrays.

Bessere Idee?

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

Zuordnung Name s zu

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

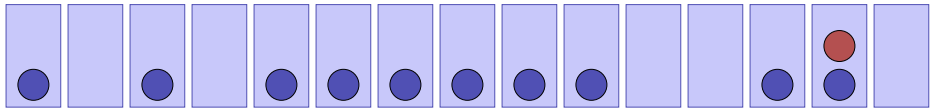
Verschiedene Namen können nun denselben Schlüssel erhalten (“Kollision”). Und dann?

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.

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

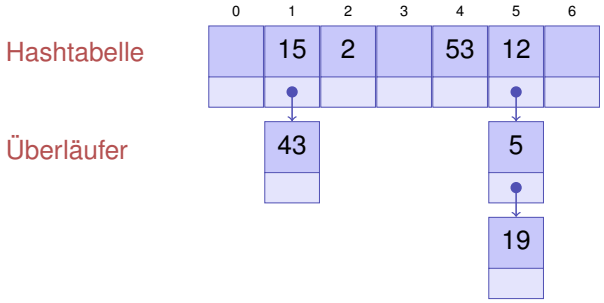
$$h_{b,m}(s) = \left(\sum_{i=0}^{l-1} s_i \cdot b^i \right) \bmod m$$

```
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 , 53 , 5 , 15 , 2 , 19 , 43

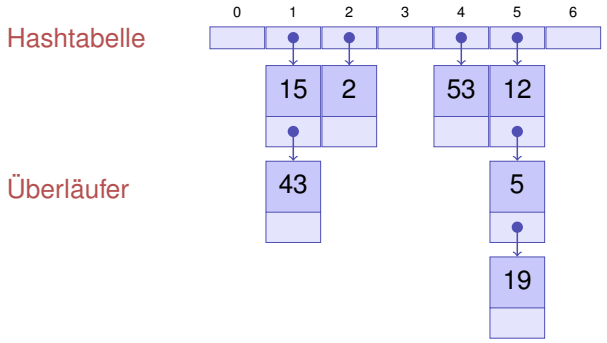
Verkettung der Überläufer



Behandlung von Kollisionen

Beispiel $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 53 , 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*

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

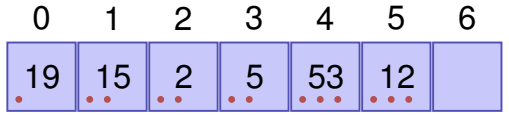
Algorithmen zum Open Addressing

- `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 , 53 , 5 , 15 , 2 , 19



Java Implementation: Tabelleneinträge

```
public class Entry{
    String key; // the key
    int value;  // data

    Entry(String n, int v){
        key = n;
        value = v;
    }
}
```


Java Implementation: Tabelle

```
class Hashtable{
    Entry[] data; // invariant: data.length = always a power of two
    int m;       // data.length−1
    int used;    // used entries in data

    Entry empty; // special entry for deletion

    // constructor
    Hashtable(int size){
        data = new Entry[size];
        m = size−1;
        empty = new Entry("",0);
    }
    ...
```


Java Implementation: Hashfunktion

```
    ...
    int ComputeHash(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;
    }
    ...
```


Java Implementation: Lineares Sondieren

```
    ...
    // linear probing, returns the index where the data set
    // containing key is stored or the index of an empty slot
    int probe(String key){
        int h = ComputeHash(key);
        Entry e = data[h];
        while (e != null && !key.equals(e.key)){
            h = (m+h−1) % m;
            e = data[h];
        }
        return h;
    }
    ...
```

Java Implementation: Suchen

```
...
// returns if the table contains a data set with key
boolean contains(String key){
    int h = probe(key);
    return data[h] != null;
}

// returns the value stored with the key
int get(String key){
    assert(contains(key));
    int h = probe(key);
    return data[h].value;
}
...
```


Java Implementation: Einfügen

```
...
void put(String name, int value){
    assert(!contains(name));
    if (++used >= m/2) // max 50% load
        Resize();
    int h = probe(name);
    data[h] = new Entry(name,value);
}
...
```


Java Implementation: Resize

```
...
void Resize(){
    Entry[] old = data;
    data = new Entry[data.length*2];
    m = data.length-1; // 2^k-1, reasonably "close" to prime
    for (int k = 0; k<old.length; ++k){
        if (old[k] != null && old[k] != empty){
            put(old[k].key, old[k].value);
        }
    }
}
...
```


Java Implementation: Löschen

```
    ...
    void remove(String key){
        assert(contains(key));
        int h = probe(key);
        data[h] = empty;
   }
}
```

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 200 Tabelleneinträge!

? Grund für die schlechte Performance?

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

Quadratisches Sondieren

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

Beispiel $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Schlüssel 12 , 53 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
19	15	2		53	12	5

Diskussion

Beispiel $\alpha = 0.95$

Erfolglose Suche betrachtet im Durchschnitt 22 Tabelleneinträge

? Grund für die schlechte Performance?

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

Double Hashing

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

Beispiel:
$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.
Schlüssel 12 , 53 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
19	15	2	5	53	12	

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.

Übersicht

	$\alpha = 0.50$		$\alpha = 0.90$		$\alpha = 0.95$	
	C_n	C'_n	C_n	C'_n	C_n	C'_n
Separate Verkettung	1.250	1.110	1.450	1.307	1.475	1.337
Direkte Verkettung	1.250	0.500	1.450	0.900	1.475	0.950
Lineares Sondieren	1.500	2.500	5.500	50.500	10.500	200.500
Quadratisches Sondieren	1.440	2.190	2.850	11.400	3.520	22.050
Double Hashing	1.39	2.000	2.560	10.000	3.150	20.000

: C_n: Anzahl Schritte erfolgreiche Suche, C'_n: Anzahl Schritte erfolglose Suche, Belegungsgrad α.

Generische Hashtabellen in Java `java.util.HashMap`

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

// Abbildung 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
```


Generische Liste in Java `java.util.List`

```
import java.util.LinkedList; // es gibt auch ArrayList
...

// Liste von Strings
LinkedList<String> list = new LinkedList<String>();

list.add("abc");
list.add("xyz");
list.add(1,"123"); // Fuege 123 an Position 1 ein
System.out.println(list.get(0)); // abc

for (String s: list) // For auf Iterator der Liste
       System.out.println(s); // abc 123 xyz
```