11. Hash Tables

Hash Tables, Birthday Paradoxon, Hash functions, Resolving Collisions with Chaining, Open Addressing, Probing

Motivation

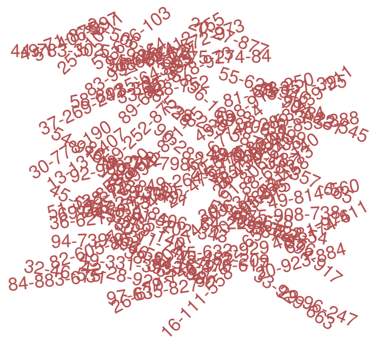
Gloal: Table of all n students of this course

Requirement: fast access by name or legi-nr

Motivation : Legi $\leftrightarrow$ Name ?

Table: 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

key

universe: 100.000.000 possible key values

Table: Name $\longrightarrow$ 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

key

universe: conceptually infinitely many possible key values

Plan

- First, we consider a startegy for a mapping key set (legi nr or name) $\rightarrow$ indices (integers) Such a function we call a *hash function*.
- Then we limit the range of values of the hash-function key set (legi nr or name) $\rightarrow \{0, \ldots, m-1\}$. Using this,we can implement a *hash table* based on an array.
- as a consequence, there will be *collisions*. We think about the probabilities of a collision and how many collisions we expect.
- Finally, we clarify how collisions can be handled.

Naive Ideas

Mapping Name $s = s_1 s_2 \ldots s_{l_s}$ to key

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

b large enough such that different names map to different keys.

Store each data set at its index in a huge array.

Example with $b = 100$. Ascii-Values s_i.

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

Unrealistic: requires too large arrays.

Better idea?

Allocation of an array of size m ($m > n$).

Mapping Name s to

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

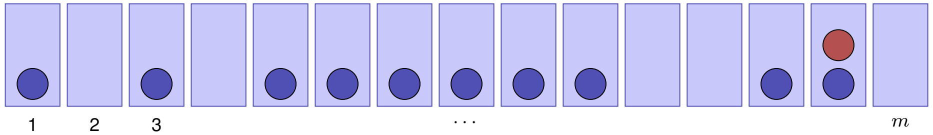
Different names can map to the same key (“Collision”). And then?

Estimation

Maybe collision do not really exist? We make an estimation ...

Estimation

Assumption: m urns, n balls (wlog $n \leq m$).
n balls are put uniformly distributed into the urns



What is the collision probability?

Very similar question: with how many people (n) the probability that two of them share the same birthday ($m = 365$) is larger than 50%?

[Estimation]

$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Let $a \ll m$. With $e^x = 1 + x + \frac{x^2}{2!} + \dots$ approximate $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$.
This yields:

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

Thus

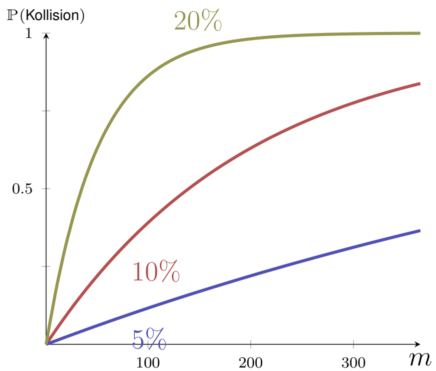
$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Puzzle answer: with 23 people the probability for a birthday collision is 50.7%. Derived from the slightly more accurate Stirling formula.

With filling degree:

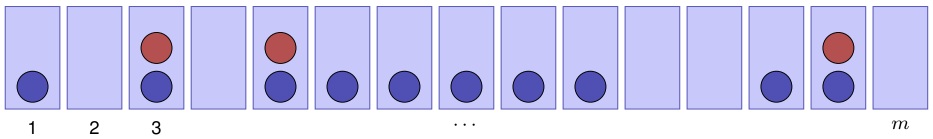
With filling degree $\alpha := n/m$ it holds that (simplified further)

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



Different Question

Assumption: m urns, n balls (wlog $n \leq m$).
n balls are put uniformly distributed into the urns



What is the expected number of collisions?

Expected Number Collisions

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

Nomenclature

Hash funtion h: Mapping from the set of keys $\mathcal{K}$ to the index set $\{0, 1, \ldots, m-1\}$ of an array (*hash table*).

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

Normally $|\mathcal{K}| \gg m$. There are $k_1, k_2 \in \mathcal{K}$ with $h(k_1) = h(k_2)$ (*collision*).

A hash function should map the set of keys **as uniformly as possible** to the hash table.

Implementation Hashfunktion (String) in Java

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

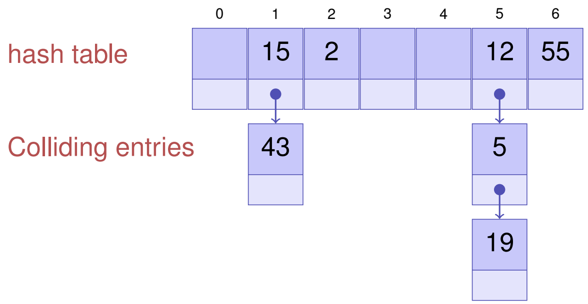
```
int ComputeHash(int m, String s) {
  int sum = 0;
  int b = 1;
  for (int k = 0; k<s.length(); ++k){
    sum = (sum + s.charAt(k) * b) % m;
    b = (b * 31) % m;
  }
  return sum;
}
```

Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Chaining the Collisions

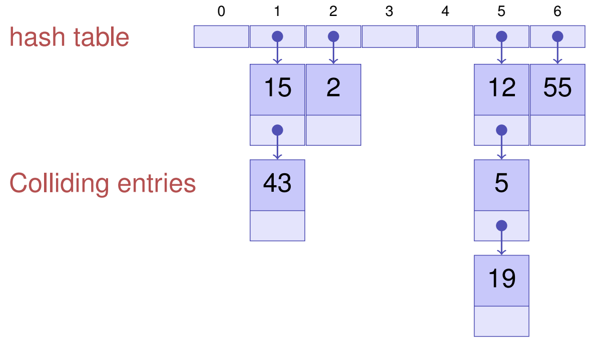


Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Direct Chaining of the Colliding entries



Algorithm for Hashing with Chaining

- `contains`(k) Search in list from position $h(k)$ for k. Return true if found, otherwise false.
- `put`(k) Check if k is in list at position $h(k)$. If no, then append k to the end of the list. Otherwise error message.
- `get`(k) Check if k is in list at position $h(k)$. If yes, return the data associated to key k, otherwise error message.
- `remove`(k) Search the list at position $h(k)$ for k. If successful, remove the list element.

Advantages and Disadvantages

Advantages

- Possible to overcommit: $\alpha > 1$
- Easy to remove keys.

Disadvantages

- Memory consumption of the chains-

Open Addressing

Store the colliding entries directly in the hash table using a *probing function* $s(j, k)$ $(0 \leq j < m, k \in \mathcal{K})$

Key table position along a *probing sequence*

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

Algorithms for open addressing

- `contains`(k) Traverse table entries according to $S(k)$. If k is found, return true. If the probing sequence is finished or an empty position is reached, return false.
- `put`(k) Search for k in the table according to $S(k)$. If k is not present, insert k at the first free position in the probing sequence. Otherwise error message.
- `get`(k) Traverse table entries according to $S(k)$. If k is found, return data associated to k. Otherwise error message.
- `remove`(k) Search k in the table according to $S(k)$. If k is found, replace it with a special `removed` key.

Linear Probing

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

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

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

Deletion with Open Addressing

- An empty slot determines the end of a probing sequence.
- If an element is removed by emptying the respective slot, a probing sequence could be interrupted.
- Therefore a special entry “removed” is used in order to specify that the probing sequence does not necessarily end but the slot is still considered empty.

Example: remove entry 55 at index 6:

0	1	2	3	4	5	6
5	15	2	19		12	(r)

Discussion

Example $\alpha = 0.95$

The unsuccessful search consideres 200 table entries on average!

? Disadvantage of the method?

! *Primary clustering:* similar hash addresses have similar probing sequences $\Rightarrow$ long contiguous areas of used entries.

Quadratic Probing

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

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

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

Discussion

Example $\alpha = 0.95$

Unsuccessfuly search considers 22 entries on average

? Problems of this method?

! *Secondary clustering:* Synonyms k and k' (with $h(k) = h(k')$) travers the same probing sequence.

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.
$S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), \dots, h(k)+(m-1)h'(k)) \mod m$

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

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

Double Hashing

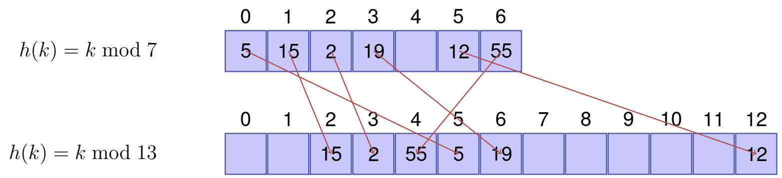
- Probing sequence must permute all hash addresses. Thus $h'(k) \neq 0$ and $h'(k)$ may not divide m, for example guaranteed with m prime.
- h' should be independent of h (avoiding secondary clustering)

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

Dynamic Hash Tables

- When a hashtable has to grow during runtime, its content has to be copied into a new table.
- The hash function changes; each entry of the old hash table needs to be re-allocated (“re-hashing”).

example (with linear probing):



Generic Hashtables 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
```