

# Algorithmen und Datenstrukturen

## B10. Hashtabellen

Marcel Lüthi and Gabriele Röger

Universität Basel

# Algorithmen und Datenstrukturen

## — B10. Hashtabellen

### B10.1 Einführung

### B10.2 Hashfunktionen

### B10.3 Hashtabellen

## B10.1 Einführung

## Symboltabellen: Übersicht

Implementation	Worst-case			Average-case		
	suchen	einfügen	löschen	suchen (hit)	einfügen	löschen
Verkettete Liste	N	N	N	N/2	N	N/2
Binäre suche	$\log_2(N)$	N	N	$\log_2(N)$	N/2	N
BST	N	N	N	$\log_2(N)$	$\log_2(N)$	$\sqrt{N}$
Rot-Schwarz Bäume	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$

### Frage

Geht es noch besser?

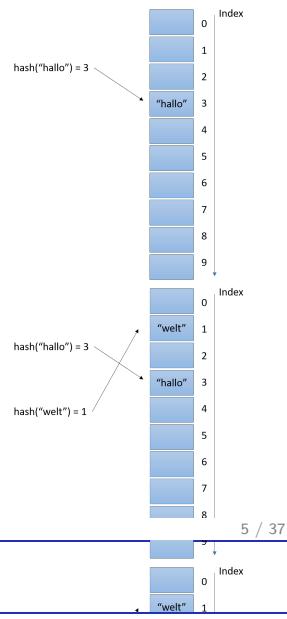
## Hashtabellen: Idee

Elemente werden in Array gespeichert, wobei Position durch Schlüssel bestimmt ist.

- Wichtigstes Werkzeug: Hashfunktion
  - Berechnet Index aus Schlüssel

Herausforderungen:

- Hashfunktion berechnen
- Kollisionen (2 unterschiedliche Schlüssel haben gleichen Hashwert)



## Hashfunktion: Ziele

- Konsistenz: Gleicher Schlüssel sollte immer gleichen Hashwert ergeben.
- Hashfunktion sollte effizient berechnet werden können.
- Schlüssel sollten gleichverteilt sein.
  - gleiche Wahrscheinlichkeit für jedes Feld

## B10.2 Hashfunktionen

## Quiz: Hashfunktion

Was sind mögliche Hashfunktionen für

- Integer (32 Bit Ganzzahl)
- Datum
- Strings
- Bilder

Wie aufwändig ist jeweils die Berechnung der Hashfunktion?

## Hashfunktionen in Java

Alle Java Klassen erben Methode `hashCode`

Anforderung:

- Falls `x.equals(y)` dann `x.hashCode() == y.hashCode()`

Gewünscht:

- Falls `!x.equals(y)` dann `x.hashCode() != y.hashCode()`

Wenn immer `equals` überschrieben wird, muss auch `hashCode` überschrieben werden.

## Beispiele von Hashfunktionen in Java

Integer:

```
public int hashCode() {
    return this.value;
}
```

## Beispiele von Hashfunktionen in Java

String:

```
public int hashCode() {
    int h = 0;
    if (value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
    }
    return h;
}
```

## Beispiele von Hashfunktionen in Java

LinkedList:

```
public int hashCode() {
    int hashCode = 1;
    for (E e : this)
        hashCode = 31 * hashCode + (e==null ? 0 : e.hashCode());
    return hashCode;
}
```

## Praktisches Rezept für benutzerdefinierte Typen

```
public int hashCode()
{
    int hash = 17;
    hash = 31*hash + field1.hashCode();
    hash = 31*hash + field2.hashCode();
    hash = 31*hash + field3.hashCode();
    ...
    return hash;
}
```

Funktioniert gut in Praxis - aber theoretisch nicht optimal.

## Hashfunktionen in Python

- ▶ Hashfunktionen werden via die Methode `__hash__` angegeben.

### `__hash__()`

Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. `__hash__()` should return an integer. The only required property is that objects which `compare equal` have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple.

Python Language Reference - Section 3: Data Model

## Praktische Tips

Gute Hashfunktionen zu entwerfen ist schwierig!

Einige Tips:

- ▶ Alle Bits im Schlüssel sollten bei Berechnung gleich mitberücksichtigt werden.
  - ▶ Verbessert Verteilung!
  - ▶ Experimentell überprüfen (plot?)
- ▶ Hashing ist klassischer Performancebug. (Alles läuft korrekt aber Programm ist langsam.)
  - ▶ Hashfunktion auf Effizienz prüfen.
  - ▶ Was ist schneller, Vergleich oder Hash?

## Modulares Hashing

Werte der Hashfunktion können negativ sein. Wir wollen aber Werte zwischen 0 und  $M$ .

- ▶ Positiven Hash-wert nehmen und Modulo  $M$  rechnen.  
In Java:

```
private int modularHash(Key x) {
    return (x.hashCode() & 0x7fffffff) % M;
}
```

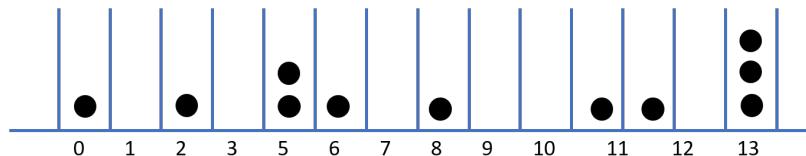
In Python:

```
def modularHash(x):
    return (hash(x) % ((sys.maxsize + 1) * 2) % M)
```

## Theoretische Analyse von Hashtabellen

### Typische Annahme

Die von uns verwendeten Hashfunktionen verteilen die Schlüssel gleichmäßig und unabhängig voneinander auf die Integer-Werte zwischen 0 und  $M - 1$ .



Bälle werden zufällig in  $M$  verschiedene Gefäße verteilt.

## Kollisionen

Wir können Kollisionen nicht verhindern.

Beispiele relevanter mathematischer Resultate:

**Geburtstagsparadox** In einer Gruppe von 23 Kindern ist die Wahrscheinlichkeit 0.5, dass zwei am selben Tag Geburtstag haben.

- ▶ Angewandt auf hashing: Anzahl Plätze:  $M = 365$ , Nach  $N = 23$  Elementen bereits grosse Chance, dass Kollision auftritt.
- ▶ Allgemein: Wir erwarten Kollision nach ungefähr  $\sqrt{\pi M/2}$  Elementen.

## Kollisionen

Wir können Kollisionen nicht verhindern.

Beispiele relevanter mathematischer Resultate:

**Sammelbilderproblem** Gegeben  $M$  Sammelbilder, wieviele Bilder muss man ziehen (mit zurücklegen), bevor man jedes einmal gezogen hat?

- ▶ Angewandt auf hashing: Wie lange dauert es bis alle Felder besetzt sind?
- ▶ Der Erwartungswert wächst mit  $\Theta(M \log(M))$

Um  $M = 50$  unterschiedliche Sammelbilder zu haben benötigen wir ungefähr  $50 \log(50) \approx 200$  Bilder

## Experimente

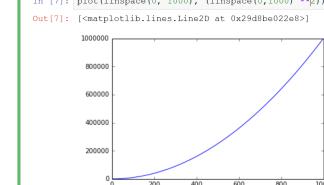


### Algorithmen und Datenstrukturen

#### Interaktive Experimente

In [3]: %pylab inline  
Populating the interactive namespace from numpy and matplotlib

In [7]: plot(linspace(0, 1000), (linspace(0, 1000) \*\* 2))  
Out[7]: [matplotlib.lines.Line2D at 0x29d0be022a8]



IPython Notebooks: Hashtables.ipynb

## B10.3 Hashtabellen

## Hashtabelle: 2 Implementationen

Grundlage ist immer ein Array der Grösse  $M$  um  $N$  Einträge zu speichern.

Wichtigste Frage: Wie behandle ich Kollisionen?

2 Strategien

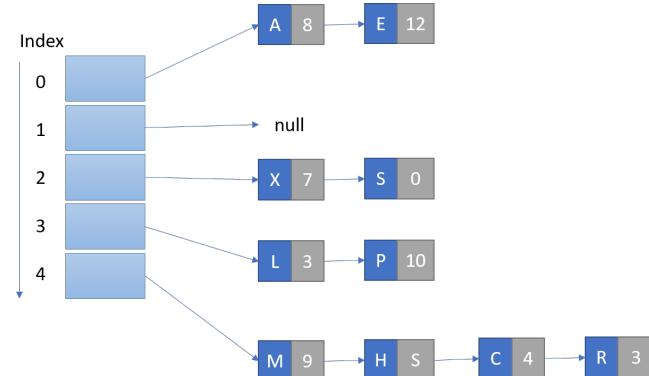
- ▶ Verkettung (separate chaining)
  - ▶ Jedes Element enthält Verkettete Liste mit allen Schlüssel / Werte Paaren
  - ▶  $M$  kann kleiner sein als  $N$
- ▶ Lineare Sondierung (linear probing)
  - ▶  $M$  wird grösser gewählt als  $N$ .
  - ▶ Suche nach nächstem freien Platz.

## Verkettung

**Hash:** Schlüssel wird auf Zahl zwischen 0 und  $M - 1$  gemappt.

**Einfügen:** Falls nicht gefunden, am Anfang in Liste einfügen

**Suche:** Relevante Liste durchsuchen



## Komplexität

### Theorem

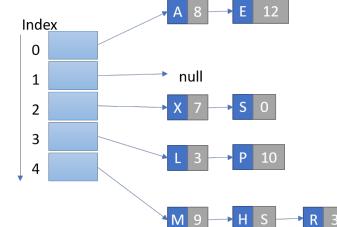
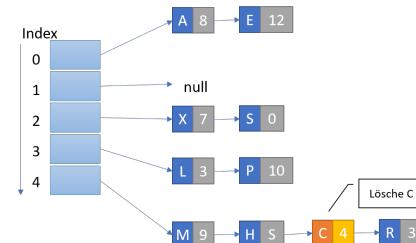
In einer auf Verkettung basierenden Hashtabelle mit  $M$  Listen und  $N$  Schlüsseln ist die Wahrscheinlichkeit (unter der Gleichverteilungsannahme), dass die Anzahl der Schlüssel in einer Liste bis auf einen kleinen konstanten Faktor bei  $N/M$  liegt, extrem nahe an 1.

### Theorem

In einer auf Verkettung basierenden Hashtabelle mit  $M$  Listen und  $N$  Schlüsseln ist die Anzahl der Vergleiche (Gleichheitstests) für Einfügungen und erfolglose Suchen  $\sim N/M$ .

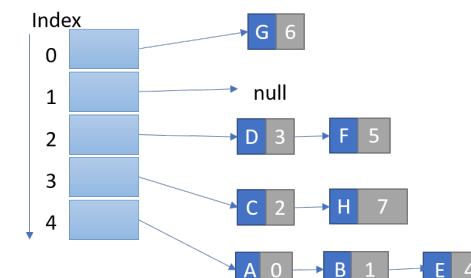
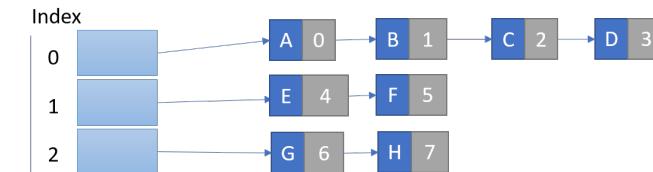
## Verkettung: Elemente Löschen

- Einfache Operation: Element aus relevanter Liste löschen.

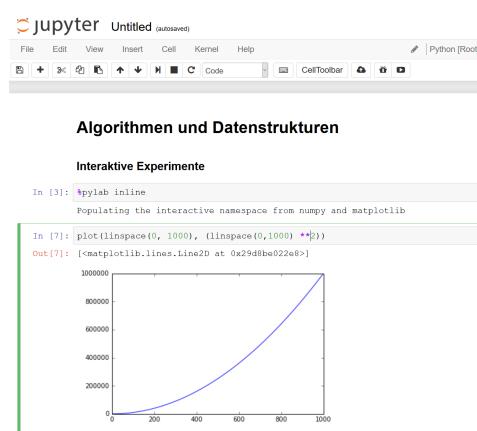


## Verkettung: Größenanpassung

- Ziel: Länge N/M bleibt etwa konstant
- Alle Elemente müssen neu gehashed werden.



## Implementation und Beispielanwendung



IPython Notebooks: Hashtables.ipynb

## Informatiker des Tages : Arthur Lee Samuel



Arthur Lee Samuel

- Professor in Stanford
- Mitentwickler von  $\text{\TeX}$
- Pionier in Künstlicher Intelligenz / Maschinellem lernen
  - Entwickelte erstes erfolgreiches Dame-Programm.
- Erste Implementation der linearen Sondierungsstrategie in Hashtabellen (1953)

## Lineares sondieren

Voraussetzung:  $M > N$

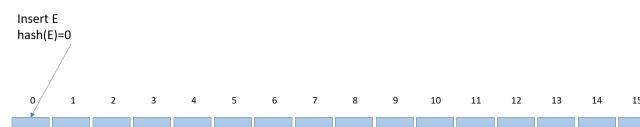
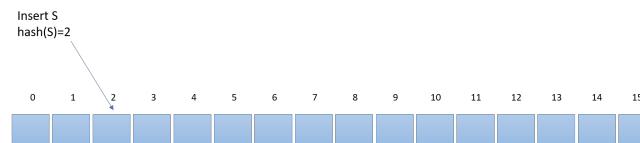
Hash: Schlüssel wird auf Zahl  $i$  zwischen 0 und  $M - 1$  gemappt.

Einfügen: An Position  $i$  einfügen.

► Falls belegt, probiere Position  $i + 1, i + 2, \dots$

Suche: Suche an Index  $i$

► Falls nicht leer, aber Eintrag  $\neq$  gesuchter Schlüssel, suche an Position  $i + 1, i + 2, \dots$

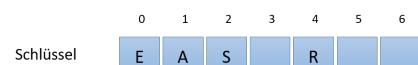


Insert A  
hash(A)=0

## Lineare Sondierung: Größenanpassung

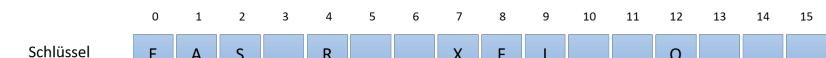
► Ziel: Länge  $N/M \leq 1/2$

► Alle Elemente müssen neu gehashed werden.

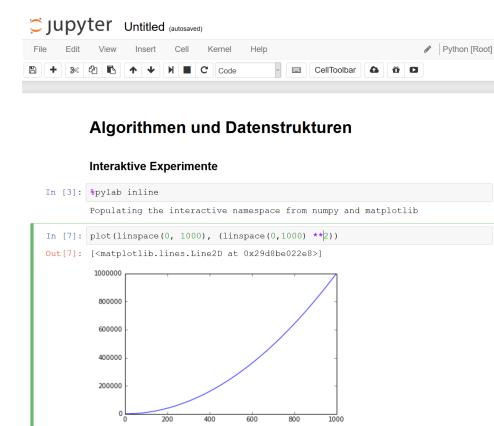


## Lineare Sondierung: Elemente Löschen

► Wenn erstes Element in Cluster gelöscht wird, müssen Nachfolger gelöscht werden.



Was ist wenn hash(l)=7?



IPython Notebooks: Hashtables.ipynb

## Clustering

### Beobachtung

Lineares Sondieren führt zu Clusterbildung.

- Bei Kollision wächst ein Cluster, da das Element am Ende eingefügt wird.

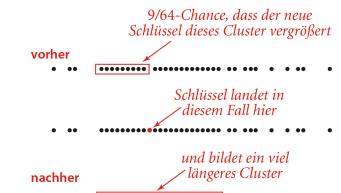


## Clustering

### Beobachtung

Lange Cluster wachsen schneller als kurze.

- Wahrscheinlichkeit in einem grossen Cluster zu landen ist grösser.



## Clustering

### Beobachtung

Laufzeit der Suche hängt von Länge der Cluster ab.

### Theorem

In einer auf linearer Sondierung basierenden Hashtabelle mit einer Liste der Grösse  $M$  und  $N = \alpha M$  Schlüsseln ist die erforderliche durchschnittliche Anzahl von Sondierungen für erfolgreiches beziehungsweise erfolgloses Suchen

$$\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{und} \quad \sim \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$

## Komplexität

Implementation	suchen	Worst-case		Average-case		
		einfügen	löschen	suchen (hit)	einfügen	löschen
Verkettete Liste	N	N	N	N/2	N	N/2
Binäre suche	$\log_2(N)$	N	N	$\log_2(N)$	$N/2$	N
BST	N	N	N	$\log_2(N)$	$\log_2(N)$	$\sqrt{N}$
Rot-Schwarz Bäume	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$	$\log_2(N)$
Hashtabellen	N	N	N	$O(1)$	$O(1)$	$O(1)$

## Diskussion

Wann sollen wir welche Art von Datenstruktur verwenden?

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	$N$	$N$	$N$	$\frac{1}{2}N$	$N$	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>
separate chaining	$N$	$N$	$N$	3.5 *	3.5 *	3.5 *		<code>equals()</code> <code>hashCode()</code>
linear probing	$N$	$N$	$N$	3.5 *	3.5 *	3.5 *		<code>equals()</code> <code>hashCode()</code>

Abbildung: Sedgewick & Wayne, Tabelle 3.15