

# Übersicht

## 2 Suchen und Sortieren

- Einfache Suche
- Binäre Suchbäume
- **Hashing**
- Skip-Lists
- Mengen
- Sortieren
- Order-Statistics

# Hashing

Wie können wir eine partielle Funktion  $\{1, \dots, n\} \rightarrow \mathbf{N}$  effizient speichern?

Wie können wir ein assoziatives Array für die Schlüssel  $\{1, \dots, n\}$  effizient implementieren?

Durch ein Array mit  $n$  Elementen.

Aber: Gewöhnlich speichern wir  $n$  Schlüssel aus einem riesigen Universum  $U$ .

$S \subseteq U$ ,  $|S|$  klein,  $|U|$  sehr groß.

Ein Array der Größe  $|U|$  ist zu groß.

Ein Array der Größe  $O(|S|)$  wäre optimal.

Finde eine Funktion  $h: U \rightarrow \{1, \dots, |S|\}$ , die auf  $S$  injektiv ist.

# Hashing

Wie können wir eine partielle Funktion  $\{1, \dots, n\} \rightarrow \mathbf{N}$  effizient speichern?

Wie können wir ein assoziatives Array für die Schlüssel  $\{1, \dots, n\}$  effizient implementieren?

Durch ein Array mit  $n$  Elementen.

Aber: Gewöhnlich speichern wir  $n$  Schlüssel aus einem riesigen Universum  $U$ .

$S \subseteq U$ ,  $|S|$  klein,  $|U|$  sehr groß.

Ein Array der Größe  $|U|$  ist zu groß.

Ein Array der Größe  $O(|S|)$  wäre optimal.

Finde eine Funktion  $h: U \rightarrow \{1, \dots, |S|\}$ , die auf  $S$  injektiv ist.

# Hashing

Es sei  $S \subseteq U$ .

Wir haben eine Funktion  $h: U \rightarrow \{1, \dots, n\}$ , so daß

- $n \geq |S|$ ,
- $h(x) \neq h(y)$  für alle  $x \neq y, x, y \in S$ .

Wir können  $x \in S$  in der  $n$ ten Position eines Arrays speichern.

Probleme:

- Wie finden wir  $h$ ?
- Können wir  $h(x)$  effizient berechnen?
- Ist  $S$  überhaupt bekannt? Ändert es sich?

# Universelles Hashing

Wir nehmen an,  $S$  ist bekannt und ändert sich nicht.

Beispiel: Tabelle der Schlüsselwörter in einem Compiler.

Es sei  $n \geq |S|$  und  $\mathcal{H}$  die Menge aller Funktion  $U \rightarrow \{1, \dots, n\}$ .

Offensichtlich enthält  $\mathcal{H}$  Funktionen, die injektiv auf  $S$  sind.

Idee:

Ersetze  $\mathcal{H}$  durch eine kleinere Klasse von Funktionen, die für jedes  $S' \subseteq U$ ,  $|S| = |S'|$  eine auf  $S'$  injektive Funktion  $h$  enthält.

# Universelles Hashing

## Definition

Es sei  $\mathcal{H}$  eine nicht-leere Menge von Funktionen  $U \rightarrow \{1, \dots, m\}$ .

Wir sagen, daß  $\mathcal{H}$  eine **universelle Familie von Hashfunktionen** ist, wenn für jedes  $x, y \in U$ ,  $x \neq y$  folgendes gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

Wir können auch Funktionen  $U \rightarrow \{0, \dots, m-1\}$  erlauben, was oft praktischer ist.

## Theorem

Es sei  $\mathcal{H}$  eine universelle Familie von Hashfunktionen  $U \rightarrow \{1, \dots, m\}$  für das Universum  $U$  und  $S \subseteq U$  eine beliebige Untermenge.

Wenn  $x \in U$ ,  $x \notin S$  und  $h \in \mathcal{H}$  eine zufällig gewählte Hashfunktion ist, dann gilt

$$E\left(|\{y \in S \mid h(x) = h(y)\}|\right) \leq \frac{|S|}{m}.$$

Beweis.

$$E\left(|\{y \in S \mid h(x) = h(y)\}|\right) =$$

$$\sum_{y \in S} \Pr[h(x) = h(y)] = \sum_{y \in S} \frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{|S|}{m}$$



## Theorem

Es sei  $\mathcal{H}$  eine universelle Familie von Hashfunktionen  $U \rightarrow \{1, \dots, m\}$  für das Universum  $U$  und  $S \subseteq U$  eine beliebige Untermenge.

Wenn  $x \in U$ ,  $x \notin S$  und  $h \in \mathcal{H}$  eine zufällig gewählte Hashfunktion ist, dann gilt

$$E\left(|\{y \in S \mid h(x) = h(y)\}|\right) \leq \frac{|S|}{m}.$$

## Beweis.

$$E\left(|\{y \in S \mid h(x) = h(y)\}|\right) =$$

$$\sum_{y \in S} \Pr[h(x) = h(y)] = \sum_{y \in S} \frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{|S|}{m}$$



## Weitere Konsequenz:

## Lemma

Sei  $x \in U$  beliebig,  $\mathcal{H}$  eine universelle Familie von Hashfunktionen  $U \rightarrow \{1, \dots, m\}$  und  $k$  eine beliebige Zahl aus  $\{1, \dots, m\}$ .

Dann gilt  $\Pr[h(x) = k] = 1/m$ , falls  $h$  zufällig aus  $\mathcal{H}$ .

## Beweis.

Nehmen wir an, es gibt ein  $y$  mit  $h(y) = k$ . Dann setze  $S = \{y\}$  und wende das letzte Theorem an:

$$E\left(|\{y \in S \mid h(x) = h(y)\}|\right) \leq \frac{|S|}{m}.$$

Hier folgt daraus  $\Pr[h(x) = k] \leq 1/m$ .

Wenn es kein  $y$  mit  $h(y) = k$  gäbe, dann wäre  $\Pr[h(x) = k] = 0$ .

Das geht aber nicht, denn  $\sum_{k=1}^m \Pr[h(x) = k] = 1$ .



Weitere Konsequenz:

### Lemma

Sei  $x \in U$  beliebig,  $\mathcal{H}$  eine universelle Familie von Hashfunktionen  $U \rightarrow \{1, \dots, m\}$  und  $k$  eine beliebige Zahl aus  $\{1, \dots, m\}$ .

Dann gilt  $\Pr[h(x) = k] = 1/m$ , falls  $h$  zufällig aus  $\mathcal{H}$ .

### Beweis.

Nehmen wir an, es gibt ein  $y$  mit  $h(y) = k$ . Dann setze  $S = \{y\}$  und wende das letzte Theorem an:

$$E\left(|\{y \in S \mid h(x) = h(y)\}|\right) \leq \frac{|S|}{m}.$$

Hier folgt daraus  $\Pr[h(x) = k] \leq 1/m$ .

Wenn es kein  $y$  mit  $h(y) = k$  gäbe, dann wäre  $\Pr[h(x) = k] = 0$ .

Das geht aber nicht, denn  $\sum_{k=1}^m \Pr[h(x) = k] = 1$ .



# Universelles Hashing

Im folgenden nehmen wir an, daß wir eine zufällige Hashfunktion aus einer geeigneten universellen Familie verwenden.

In der Praxis verwendet man oft einfachere Hashfunktionen, aber Experimente deuten an, daß sie sich praktisch so gut wie universelle Hashfunktionen verhalten.

Sie funktionieren im Worst-Case nicht, aber haben sich doch praktisch bewährt. Wir müssen darauf achten, daß die Schlüssel möglichst gleichmäßig und unabhängig voneinander verteilt werden.

# Hashing mit Verkettung

Auf Hashing basiertes assoziatives Array:

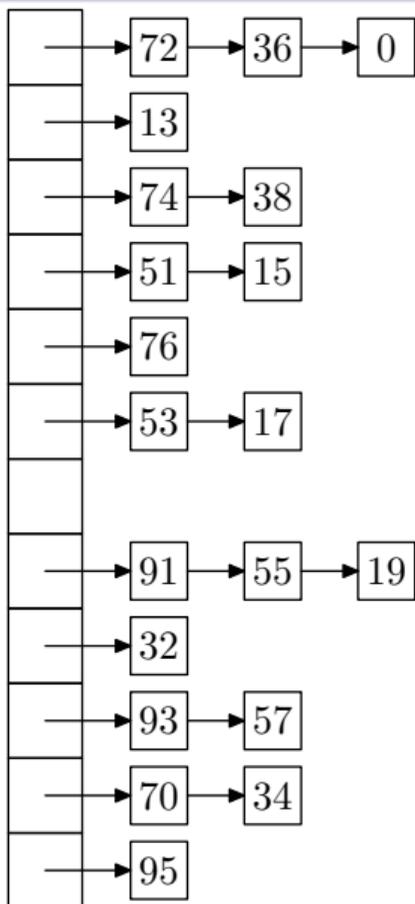
Wie behandeln wir Kollisionen?

Eine einfache Möglichkeit:

- Die Hashtabelle hat  $m$  Positionen
- Jede Position besteht aus einer verketteten Liste
- Die Liste auf Position  $k$  enthält alle  $x$  mit  $h(x) = k$

Vorteile: Einfügen, Suchen und Löschen sehr einfach umzusetzen.

Nachteile: Speicherverschwendung?



Beispiel einer Hashtabelle mit Verkettung

- 20 Elemente
- Tabellengröße 12
- Lastfaktor  $\alpha = 5/3$

# Hashing mit Verkettung – Analyse

Hashtabelle der Größe  $m$  mit  $n$  Elementen gefüllt.

Erfolglose Suche:

Sei  $x$  ein Element, das nicht in der Tabelle ist.

$$\Pr[h(x) = k] = 1/m$$

Sei  $L_i$  die Größe der Liste auf Position  $i$ .

Erwartete Anzahl besuchter Listenelemente:

$$\frac{1}{m} \sum_{k=1}^m L_i = \frac{n}{m} = \alpha$$

Laufzeit  $O(\alpha)$ .

## Hashing mit Verkettung – Analyse

Erfolgreiche Suche:

Wir wählen ein Element  $x$  aus der Tabelle **zufällig** aus.

Es bestehe  $S$  aus den anderen Elementen in der Tabelle.

Erwartete Anzahl von anderen Elementen in der Liste, die  $x$  enthält:

$$E(\{y \in S \mid h(x) = h(y)\}) \leq \frac{n-1}{m}$$

Da  $x$  zufällig gewählt wurde, ist  $x$  ein zufälliges Element in dieser Liste und daher an zufälliger Position.

→ Im Durchschnitt sind die Hälfte der anderen Elemente vor  $x$ .

Anzahl besuchter Listenelemente im Durchschnitt:

$\leq 1 + \alpha/2$  ( $x$  und Hälfte der Fremden)

## Hashing mit Verkettung – Analyse

Erfolgreiche Suche:

Wir wählen ein Element  $x$  aus der Tabelle **zufällig** aus.

Es bestehe  $S$  aus den anderen Elementen in der Tabelle.

Erwartete Anzahl von anderen Elementen in der Liste, die  $x$  enthält:

$$E(\{y \in S \mid h(x) = h(y)\}) \leq \frac{n-1}{m}$$

Da  $x$  zufällig gewählt wurde, ist  $x$  ein zufälliges Element in dieser Liste und daher an zufälliger Position.

→ Im Durchschnitt sind die Hälfte der anderen Elemente vor  $x$ .

Anzahl besuchter Listenelemente im Durchschnitt:

$\leq 1 + \alpha/2$  ( $x$  und Hälfte der Fremden)

## Hashing mit Verkettung – Analyse

Erfolgreiche Suche:

Wir wählen ein Element  $x$  aus der Tabelle **zufällig** aus.

Es bestehe  $S$  aus den anderen Elementen in der Tabelle.

Erwartete Anzahl von anderen Elementen in der Liste, die  $x$  enthält:

$$E(\{y \in S \mid h(x) = h(y)\}) \leq \frac{n-1}{m}$$

Da  $x$  zufällig gewählt wurde, ist  $x$  ein zufälliges Element in dieser Liste und daher an zufälliger Position.

→ Im Durchschnitt sind die Hälfte der anderen Elemente vor  $x$ .

Anzahl besuchter Listenelemente im Durchschnitt:

$\leq 1 + \alpha/2$  ( $x$  und Hälfte der Fremden)

## Hashing mit Verkettung – Analyse

Erfolgreiche Suche:

Wir wählen ein Element  $x$  aus der Tabelle **zufällig** aus.

Es bestehe  $S$  aus den anderen Elementen in der Tabelle.

Erwartete Anzahl von anderen Elementen in der Liste, die  $x$  enthält:

$$E(\{y \in S \mid h(x) = h(y)\}) \leq \frac{n-1}{m}$$

Da  $x$  zufällig gewählt wurde, ist  $x$  ein zufälliges Element in dieser Liste und daher an zufälliger Position.

→ Im Durchschnitt sind die Hälfte der anderen Elemente vor  $x$ .

Anzahl besuchter Listenelemente im Durchschnitt:

$\leq 1 + \alpha/2$  ( $x$  und Hälfte der Fremden)

# Hashing mit Verkettung – Analyse

Erfolgreiche Suche:

Wir wählen ein Element  $x$  aus der Tabelle **zufällig** aus.

Es bestehe  $S$  aus den anderen Elementen in der Tabelle.

Erwartete Anzahl von anderen Elementen in der Liste, die  $x$  enthält:

$$E(\{y \in S \mid h(x) = h(y)\}) \leq \frac{n-1}{m}$$

Da  $x$  zufällig gewählt wurde, ist  $x$  ein zufälliges Element in dieser Liste und daher an zufälliger Position.

→ Im Durchschnitt sind die Hälfte der anderen Elemente vor  $x$ .

Anzahl besuchter Listenelemente im Durchschnitt:

$\leq 1 + \alpha/2$  ( $x$  und Hälfte der Fremden)

# Hashing mit Verkettung – Analyse

Insgesamt erhalten wir folgende Abschätzung:

## Theorem

*Sei eine Hashtabelle mit Verkettung der Größe  $m$  gegeben, die mit  $n$  Schlüsseln gefüllt ist.*

*Bei universellem Hashing benötigt*

- 1 eine erfolglose Suche nach einem beliebigen Element  $x \in U$  im Durchschnitt höchstens  $\alpha = n/m$  viele Vergleiche,*
- 2 eine erfolgreiche Suche nach einem zufällig gewählten Element  $x$  aus der Tabelle im Durchschnitt höchstens  $1 + \alpha/2$  viele Vergleiche.*

Hashing ist sehr effizient, wenn wir die Hashfunktion schnell auswerten können und  $\alpha$  nicht zu groß ist.

# Hashing mit Verkettung – Analyse

Insgesamt erhalten wir folgende Abschätzung:

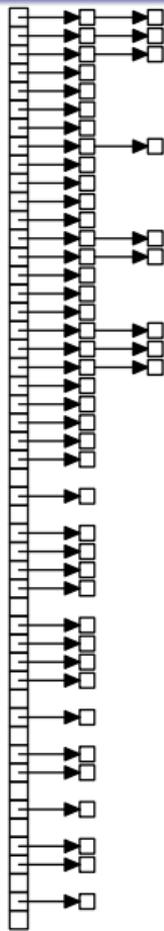
## Theorem

*Sei eine Hashtabelle mit Verkettung der Größe  $m$  gegeben, die mit  $n$  Schlüsseln gefüllt ist.*

*Bei universellem Hashing benötigt*

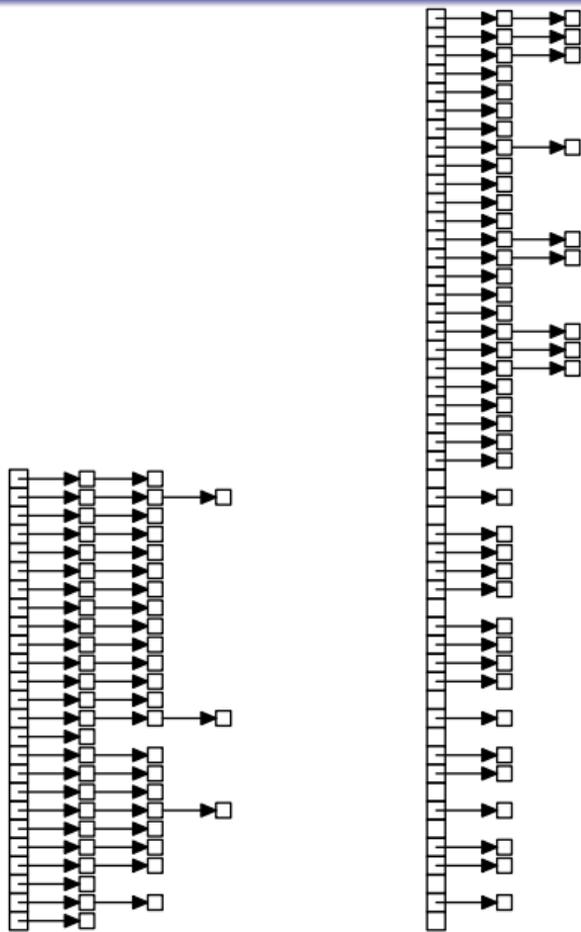
- 1 eine erfolglose Suche nach einem beliebigen Element  $x \in U$  im Durchschnitt höchstens  $\alpha = n/m$  viele Vergleiche,*
- 2 eine erfolgreiche Suche nach einem zufällig gewählten Element  $x$  aus der Tabelle im Durchschnitt höchstens  $1 + \alpha/2$  viele Vergleiche.*

Hashing ist sehr effizient, wenn wir die Hashfunktion schnell auswerten können und  $\alpha$  nicht zu groß ist.



Ein größeres Beispiel einer Hashtabelle mit Verkettung

- 50 Elemente
- Tabellengröße 50
- Lastfaktor  $\alpha = 1$



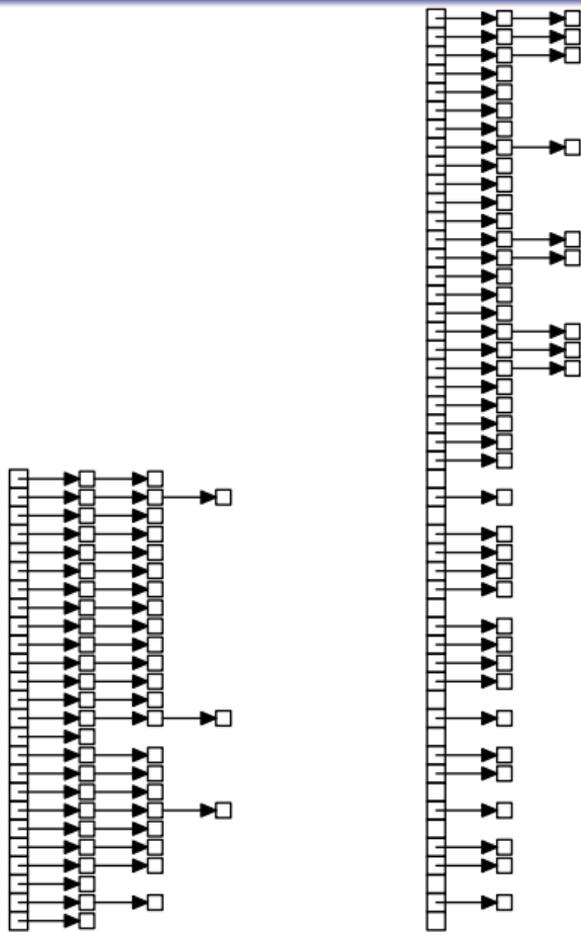
## Rehashing

Nach Einfügen ist Tabelle zu voll

- Lastfaktor  $\alpha = 2$
- Rehashing auf doppelte Größe
- $\rightarrow$  Lastfaktor 1

Wie teuer?

Amortisierte Kosten:  $O(1)$



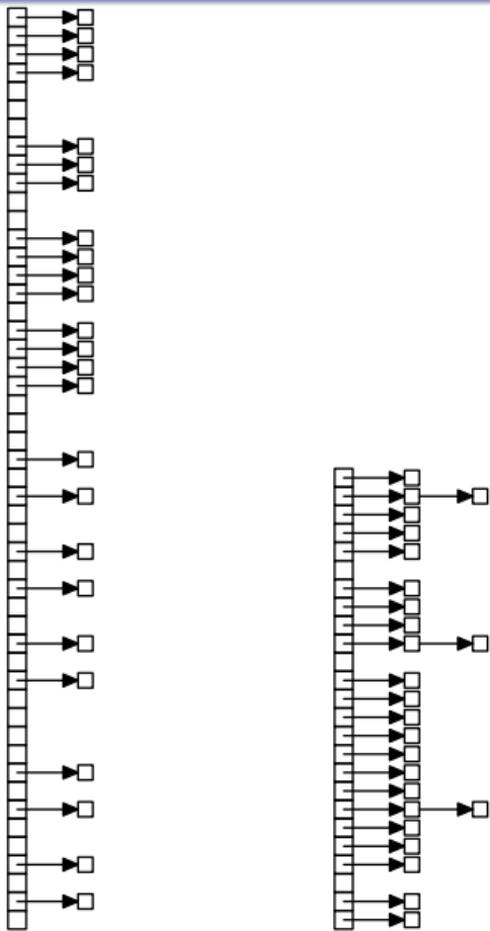
## Rehashing

Nach Einfügen ist Tabelle zu voll

- Lastfaktor  $\alpha = 2$
- Rehashing auf doppelte Größe
- $\rightarrow$  Lastfaktor 1

Wie teuer?

Amortisierte Kosten:  $O(1)$



## Rehashing

Nach Löschen ist Tabelle zu leer  
(Speicherplatz!)

- Lastfaktor  $\alpha = 1/2$
- Rehashing auf halbe Größe
- $\rightarrow$  Lastfaktor 1

Wie teuer?

Amortisierte Kosten:  $O(1)$

## Java

```
int slot(K k) {  
    int n = k.hashCode()%table.size();  
    return n > 0 ? n : -n;  
}
```

## Java

```
public D find(K k) {  
    int l = slot(k);  
    if(table.get(l) == null) return null;  
    return table.get(l).find(k);  
}
```

## Java

```
public void insert(K k, D d) {  
    int l = slot(k);  
    if(table.get(l) == null) table.set(l, new ADList<K, D>());  
    if(!table.get(l).containsKey(k)) size++;  
    table.get(l).put(k, d);  
    rehash();  
}
```

## Java

```
public void delete(K k) {  
    int l = slot(k);  
    if(table.get(l) == null || !table.get(l).containsKey(k)) return;  
    table.get(l).delete(k);  
    if(table.get(l).isEmpty()) table.set(l, null);  
    size--;  
    rehash();  
}
```

## Java

```
void rehash() {  
    if(size ≤ table.size() && 4 * size + 10 ≥ table.size()) return;  
    int newtablesize = 2 * size + 10;  
    List<ADList<K, D>> newtable;  
    newtable = new ArrayList<ADList<K, D>>(newtablesize);  
    for(int i = 0; i < newtablesize; i++) {  
        newtable.add(null);  
    }  
    Iterator<K, D> it;  
    for(it = iterator(); it.more(); it.step()) {  
        int l = it.key().hashCode() % newtablesize;  
        if(l < 0) l = -l;  
        if(newtable.get(l) == null) newtable.set(l, new ADList<K, D>());  
        newtable.get(l).put(it.key(), it.data());  
    }  
}
```

## Andere Methoden zur Kollisionauflösung

Neben Verkettung gibt es viele andere Methoden, Kollision zu behandeln:

- Linear Probing
- Quadratic Probing
- Double Hashing
- Sekundäre Hashtabelle

und viele andere...