

# Datenstrukturen und Algorithmen

## Vorlesung 12: Hashing I

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<http://www-i2.informatik.rwth-aachen.de/i2/dsal12/>

25. Mai 2012

# Übersicht

## 1 Direkte Adressierung

- Counting Sort

## 2 Grundlagen des Hashings

## 3 Verkettung

## 4 Hashfunktionen

# Einführung (I)

## Dictionary (Wörterbuch)

Das **Dictionary** (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres **Schlüssels** abgerufen werden können.

# Einführung (I)

## Dictionary (Wörterbuch)

Das **Dictionary** (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres **Schlüssels** abgerufen werden können. Weiterhin:

- ▶ Die Daten sind dynamisch gespeichert.

# Einführung (I)

## Dictionary (Wörterbuch)

Das **Dictionary** (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres **Schlüssels** abgerufen werden können. Weiterhin:

- ▶ Die Daten sind dynamisch gespeichert.
- ▶ **Element** `dictSearch(Dict d, int k)` gibt die in `d` zum Schlüssel `k` gespeicherten Informationen zurück.

# Einführung (I)

## Dictionary (Wörterbuch)

Das **Dictionary** (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres **Schlüssels** abgerufen werden können. Weiterhin:

- ▶ Die Daten sind dynamisch gespeichert.
- ▶ **Element** `dictSearch(Dict d, int k)` gibt die in `d` zum Schlüssel `k` gespeicherten Informationen zurück.
- ▶ **void** `dictInsert(Dict d, Element e)` speichert Element `e` unter seinem Schlüssel `e.key` in `d`.

# Einführung (I)

## Dictionary (Wörterbuch)

Das **Dictionary** (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres **Schlüssels** abgerufen werden können. Weiterhin:

- ▶ Die Daten sind dynamisch gespeichert.
- ▶ **Element** `dictSearch(Dict d, int k)` gibt die in `d` zum Schlüssel `k` gespeicherten Informationen zurück.
- ▶ **void** `dictInsert(Dict d, Element e)` speichert Element `e` unter seinem Schlüssel `e.key` in `d`.
- ▶ **void** `dictDelete(Dict d, Element e)` löscht das Element `e` aus `d`, wobei `e` in `d` enthalten sein muss.

# Einführung (I)

## Dictionary (Wörterbuch)

Das **Dictionary** (auch: Map, assoziatives Array) speichert Informationen, die jederzeit anhand ihres **Schlüssels** abgerufen werden können. Weiterhin:

- ▶ Die Daten sind dynamisch gespeichert.
- ▶ **Element** `dictSearch(Dict d, int k)` gibt die in `d` zum Schlüssel `k` gespeicherten Informationen zurück.
- ▶ **void** `dictInsert(Dict d, Element e)` speichert Element `e` unter seinem Schlüssel `e.key` in `d`.
- ▶ **void** `dictDelete(Dict d, Element e)` löscht das Element `e` aus `d`, wobei `e` in `d` enthalten sein muss.

## Beispiel

Symboltabelle eines Compilers, wobei die Schlüssel Strings (etwa Bezeichner) sind.

# Einführung (II)

## Problem

*Welche Datenstrukturen sind geeignet, um ein Dictionary zu implementieren?*

# Einführung (II)

## Problem

*Welche Datenstrukturen sind geeignet, um ein Dictionary zu implementieren?*

- ▶ **Heap**: Einfügen und Löschen sind effizient. Aber was ist mit Suche?

# Einführung (II)

## Problem

*Welche Datenstrukturen sind geeignet, um ein Dictionary zu implementieren?*

- ▶ **Heap**: Einfügen und Löschen sind effizient. Aber was ist mit Suche?
- ▶ Sortiertes **Array/Liste**: Einfügen ist im Worst-Case linear.

# Einführung (II)

## Problem

*Welche Datenstrukturen sind geeignet, um ein Dictionary zu implementieren?*

- ▶ **Heap**: Einfügen und Löschen sind effizient. Aber was ist mit Suche?
- ▶ Sortiertes **Array/Liste**: Einfügen ist im Worst-Case linear.
- ▶ **Rot-Schwarz-Baum**: Alle Operationen sind im Worst-Case logarithmisch.

# Einführung (II)

## Problem

*Welche Datenstrukturen sind geeignet, um ein Dictionary zu implementieren?*

- ▶ **Heap**: Einfügen und Löschen sind effizient. Aber was ist mit Suche?
- ▶ Sortiertes **Array/Liste**: Einfügen ist im Worst-Case linear.
- ▶ **Rot-Schwarz-Baum**: Alle Operationen sind im Worst-Case logarithmisch.

## Lösung

*Unter realistischen Annahmen benötigt eine **Hash-Tabelle** im Durchschnitt  $O(1)$  für alle Operationen.*

# Übersicht

## 1 Direkte Adressierung

- Counting Sort

## 2 Grundlagen des Hashings

## 3 Verkettung

## 4 Hashfunktionen

# Direkte Adressierung (I)

## Direkte Adressierung

- ▶ Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.

# Direkte Adressierung (I)

## Direkte Adressierung

- ▶ Alloziere ein Array (die [Direkte-Adressierungs-Tabelle](#)), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- ▶ Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - ▶ Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.

# Direkte Adressierung (I)

## Direkte Adressierung

- ▶ Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- ▶ Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - ▶ Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.
- ▶ Mit Schlüsselmenge  $U = \{0, 1, \dots, n - 1\}$  ergibt sich:
  - ▶ Eine Direkte-Adressierungs-Tabelle  $T[0..n-1]$ , wobei  $T[k]$  zu Schlüssel  $k$  gehört.

# Direkte Adressierung (I)

## Direkte Adressierung

- ▶ Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- ▶ Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - ▶ Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.
- ▶ Mit Schlüsselmenge  $U = \{0, 1, \dots, n - 1\}$  ergibt sich:
  - ▶ Eine Direkte-Adressierungs-Tabelle  $T[0..n-1]$ , wobei  $T[k]$  zu Schlüssel  $k$  gehört.
  - ▶ `datSearch(T, int k): return T[k];`

# Direkte Adressierung (I)

## Direkte Adressierung

- ▶ Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- ▶ Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - ▶ Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.
- ▶ Mit Schlüsselmenge  $U = \{0, 1, \dots, n - 1\}$  ergibt sich:
  - ▶ Eine Direkte-Adressierungs-Tabelle  $T[0..n-1]$ , wobei  $T[k]$  zu Schlüssel  $k$  gehört.
  - ▶ `datSearch(T, int k): return T[k];`
  - ▶ `datInsert(T, Element e): T[e.key] = e;`

# Direkte Adressierung (I)

## Direkte Adressierung

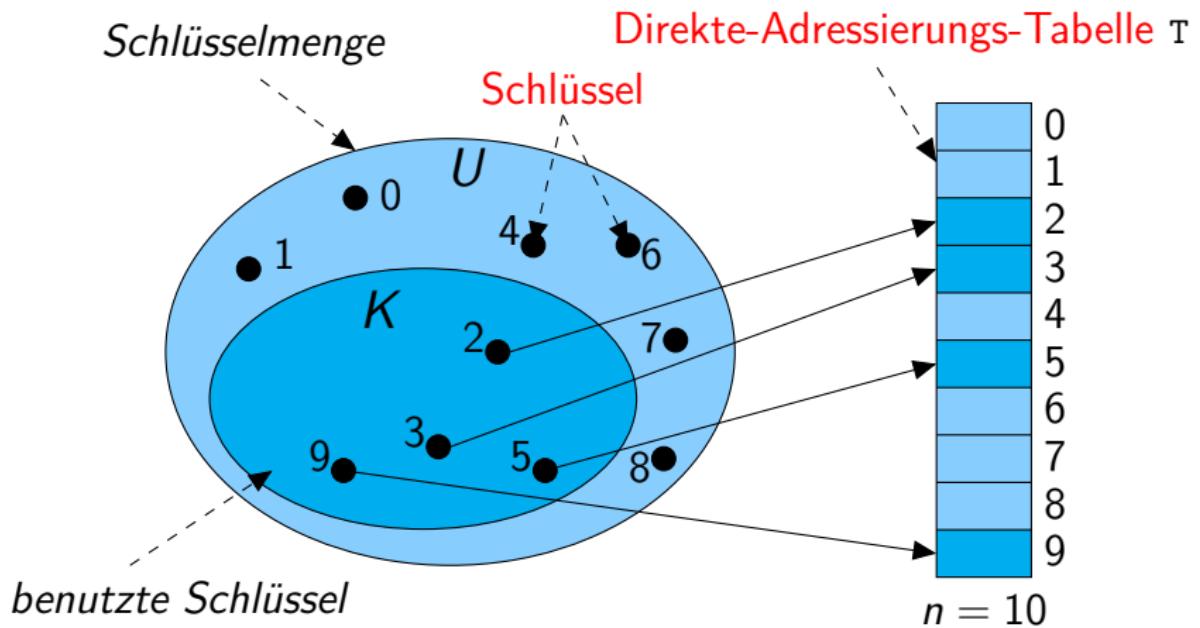
- ▶ Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- ▶ Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - ▶ Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.
- ▶ Mit Schlüsselmenge  $U = \{0, 1, \dots, n - 1\}$  ergibt sich:
  - ▶ Eine Direkte-Adressierungs-Tabelle  $T[0..n-1]$ , wobei  $T[k]$  zu Schlüssel  $k$  gehört.
  - ▶ `datSearch(T, int k): return T[k];`
  - ▶ `datInsert(T, Element e): T[e.key] = e;`
  - ▶ `datDelete(T, Element e): T[e.key] = null;`

# Direkte Adressierung (I)

## Direkte Adressierung

- ▶ Alloziere ein Array (die Direkte-Adressierungs-Tabelle), so dass es für jeden möglichen Schlüssel eine(1) Position gibt.
- ▶ Jedes Array-Element enthält einen Pointer auf die gespeicherte Information.
  - ▶ Der Einfachheit halber vernachlässigen wir in der Vorlesung die zu den Schlüsseln gehörenden Informationen.
- ▶ Mit Schlüsselmenge  $U = \{0, 1, \dots, n - 1\}$  ergibt sich:
  - ▶ Eine Direkte-Adressierungs-Tabelle  $T[0..n-1]$ , wobei  $T[k]$  zu Schlüssel  $k$  gehört.
  - ▶ `datSearch(T, int k): return T[k];`
  - ▶ `datInsert(T, Element e): T[e.key] = e;`
  - ▶ `datDelete(T, Element e): T[e.key] = null;`
- ▶ Die Laufzeit jeder Operation ist im Worst-Case  $\Theta(1)$ .

# Direkte Adressierung (II)



# Duplikate in Linearzeit erkennen

Alle Elemente seien ganze Zahlen zwischen 0 und  $k$ , wobei  $k \in \Theta(n)$ .

---

```
1 bool checkDuplicates(int E[n], int n, int k) {
2     int histogram[k] = 0; // "Direkte-Adressierungs-Tabelle"
3     for (int i = 0; i < n; i++) {
4         if (histogram[E[i]] > 0) {
5             return true; // Duplikat gefunden
6         } else {
7             histogram[E[i]]++; // Zähle Häufigkeit
8         }
9     }
10    return false; // keine Duplikate
11 }
```

---

# Counting Sort – Idee

1. Berechne Häufigkeit

# Counting Sort – Idee

1. Berechne Häufigkeit
2. Berechne „Position von  $x$ “ = „Anzahl der Elemente  $\leq x$ “

# Counting Sort – Idee

1. Berechne Häufigkeit
2. Berechne „Position von  $x$ “ = „Anzahl der Elemente  $\leq x$ “
3. Erzeuge Ausgabearray anhand dieser neuen Positionen

# Counting Sort

Alle Elemente seien ganze Zahlen zwischen 0 und  $k$ , wobei  $k \in \Theta(n)$ .

# Counting Sort

Alle Elemente seien ganze Zahlen zwischen 0 und  $k$ , wobei  $k \in \Theta(n)$ .

```
1 int[n] countSort(int E[n], int n, int k) {  
2     int histogram[k] = 0; // "Direkte-Adressierungs-Tabelle"  
3     for (int i = 0; i < n; i++) {  
4         histogram[E[i]]++; // Zähle Häufigkeit  
5     }  
6     for (int i = 1; i < k; i++) { // Berechne Position  
7         histogram[i] = histogram[i] + histogram[i - 1];  
8     }  
9     // Erzeuge Ausgabe  
10    int result[n];  
11    for (int i = n - 1; i >= 0; i--) { // stabil: rückwärts  
12        histogram[E[i]]--;  
13        result[histogram[E[i]]] = E[i];  
14    }  
15    return result;  
16 }
```

# Counting Sort

Alle Elemente seien ganze Zahlen zwischen 0 und  $k$ , wobei  $k \in \Theta(n)$ .

```
1 int[n] countSort(int E[n], int n, int k) {  
2     int histogram[k] = 0; // "Direkte-Adressierungs-Tabelle"  
3     for (int i = 0; i < n; i++) {  
4         histogram[E[i]]++; // Zähle Häufigkeit  
5     }  
6     for (int i = 1; i < k; i++) { // Berechne Position  
7         histogram[i] = histogram[i] + histogram[i - 1];  
8     }  
9     // Erzeuge Ausgabe  
10    int result[n];  
11    for (int i = n - 1; i >= 0; i--) { // stabil: rückwärts  
12        histogram[E[i]]--;  
13        result[histogram[E[i]]] = E[i];  
14    }  
15    return result;  
16 }
```

- ▶ Worst-Case Zeitkomplexität:  $\Theta(n)$

# Counting Sort: Beispiel

Eingabe

6	8	0	3	5	4	0	4
0	1	2	3	4	5	6	7

# Counting Sort: Beispiel

Eingabe

6	8	0	3	5	4	0	4
0	1	2	3	4	5	6	7

Histogramm

2	0	0	1	2	1	1	0	1
0	1	2	3	4	5	6	7	8

Ausgabe

0	0	3	4	4	5	6	8
0	1	2	3	4	5	6	7

# Counting Sort: Beispiel

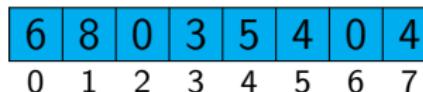
Eingabe	6	8	0	3	5	4	0	4
	0	1	2	3	4	5	6	7

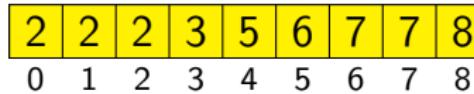
Histogramm	2	0	0	1	2	1	1	0	1
	0	1	2	3	4	5	6	7	8

Positionen	2	2	2	3	5	6	7	7	8
	0	1	2	3	4	5	6	7	8

Ausgabe	0	0	3	4	4	5	6	8
	0	1	2	3	4	5	6	7

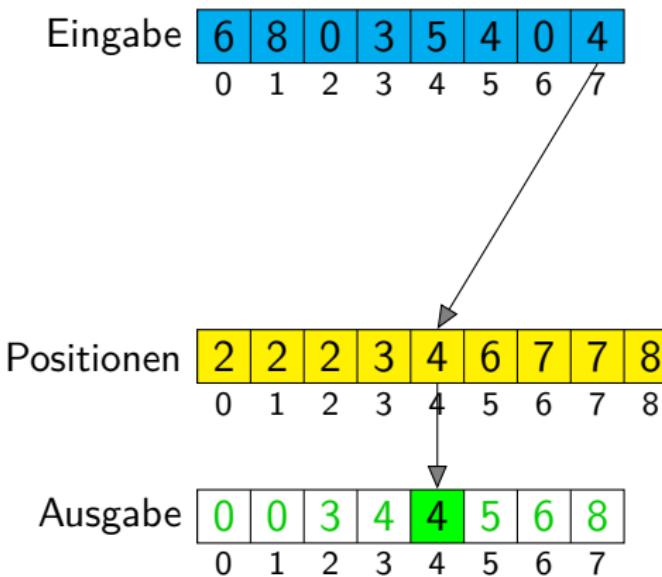
# Counting Sort: Beispiel

Eingabe   
0 1 2 3 4 5 6 7

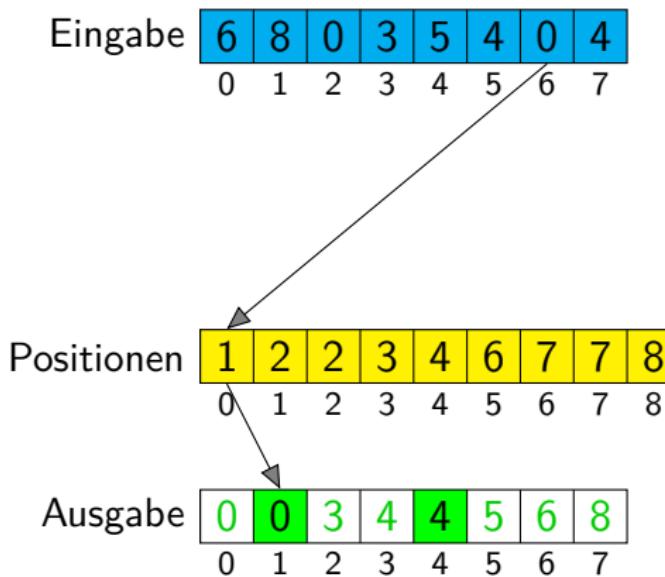
Positionen   
0 1 2 3 4 5 6 7 8

Ausgabe   
0 1 2 3 4 5 6 7

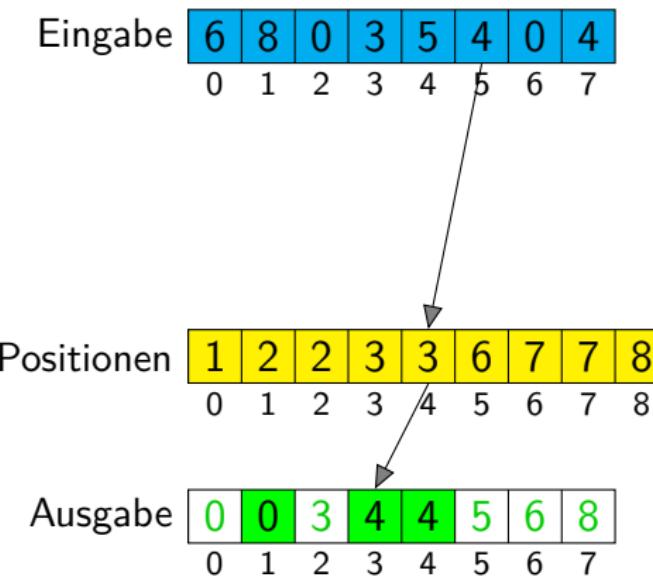
# Counting Sort: Beispiel



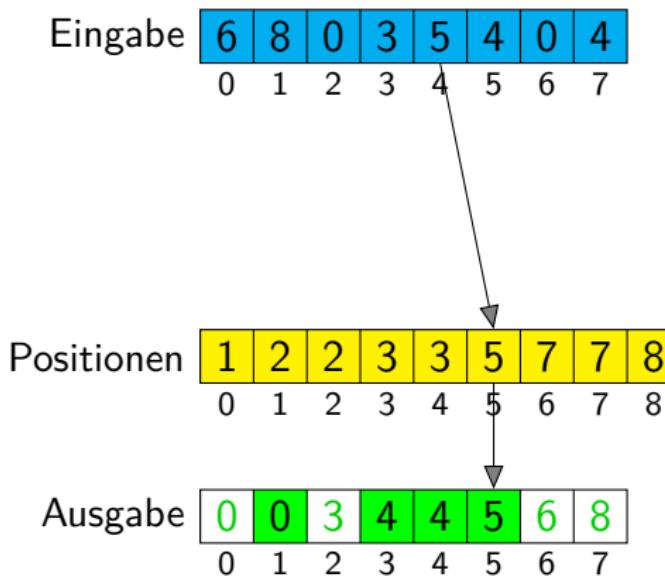
# Counting Sort: Beispiel



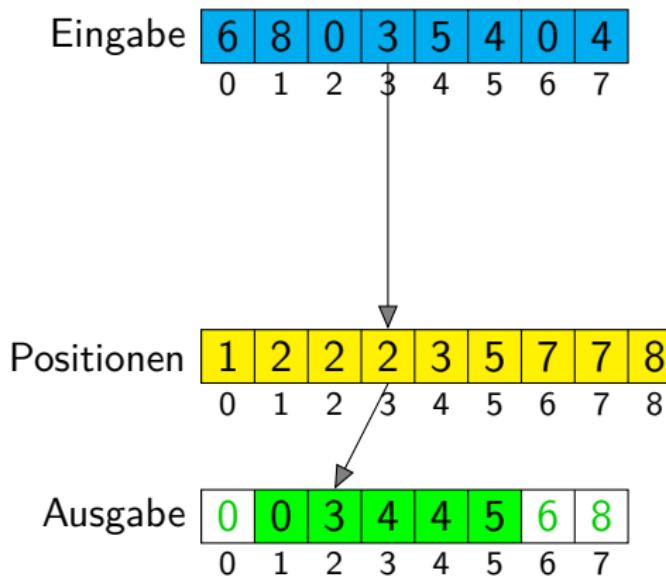
# Counting Sort: Beispiel



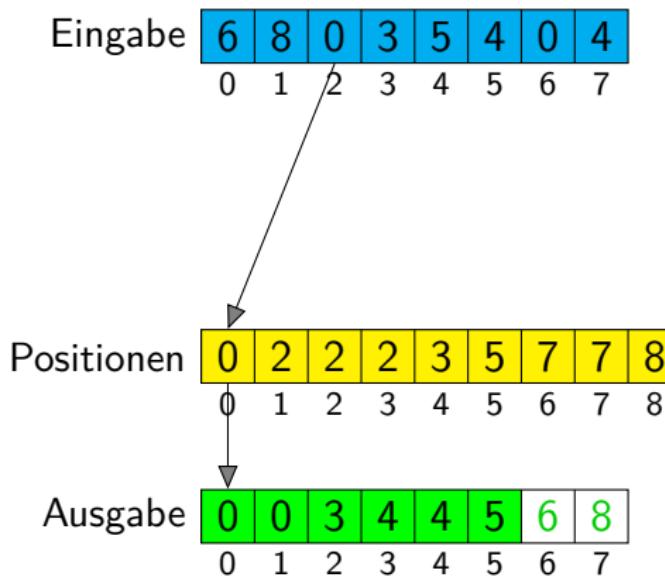
# Counting Sort: Beispiel



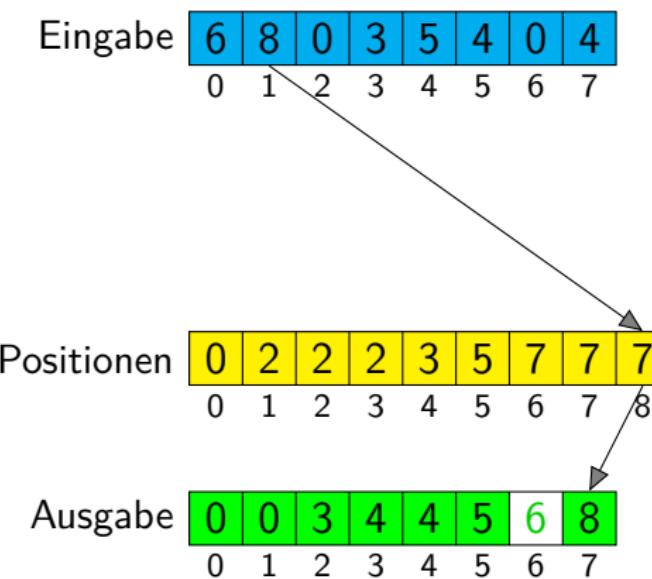
# Counting Sort: Beispiel



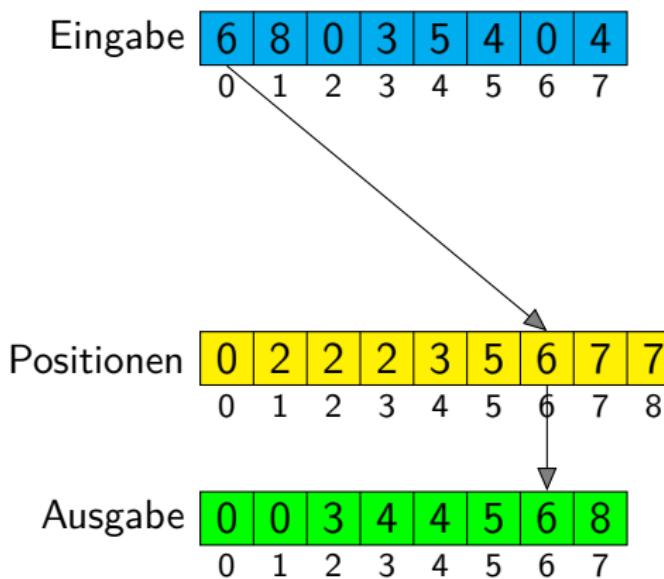
# Counting Sort: Beispiel



# Counting Sort: Beispiel



# Counting Sort: Beispiel



# Counting Sort: Beispiel

Eingabe	6	8	0	3	5	4	0	4
	0	1	2	3	4	5	6	7

Positionen	0	2	2	2	3	5	6	7	7
	0	1	2	3	4	5	6	7	8

Ausgabe	0	0	3	4	4	5	6	8
	0	1	2	3	4	5	6	7

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

## Lösung

*Dieser Algorithmus ist nicht mit Quicksort, Heapsort, usw. vergleichbar*

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

## Lösung

*Dieser Algorithmus ist nicht mit Quicksort, Heapsort, usw. vergleichbar*

- ▶ *denn er basiert nicht auf Vergleich von Elementen, sondern auf Häufigkeiten.*
- ▶ *Das funktioniert, indem wir Direkte-Adressierung (Einfügen, Suchen, Löschen in  $\Theta(1)$ ) ausnutzen.*

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

## Lösung

*Dieser Algorithmus ist nicht mit Quicksort, Heapsort, usw. vergleichbar*

- ▶ *denn er basiert nicht auf Vergleich von Elementen, sondern auf Häufigkeiten.*
- ▶ *Das funktioniert, indem wir Direkte-Adressierung (Einfügen, Suchen, Löschen in  $\Theta(1)$ ) ausnutzen.*

**Hauptproblem:** Übermäßiger Speicherbedarf für das Array.

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

## Lösung

*Dieser Algorithmus ist nicht mit Quicksort, Heapsort, usw. vergleichbar*

- ▶ *denn er basiert nicht auf Vergleich von Elementen, sondern auf Häufigkeiten.*
- ▶ *Das funktioniert, indem wir Direkte-Adressierung (Einfügen, Suchen, Löschen in  $\Theta(1)$ ) ausnutzen.*

**Hauptproblem:** Übermäßiger Speicherbedarf für das Array.

- ▶ Zum Beispiel bei Strings mit 20 Zeichen (5 bit/Zeichen) als Schlüssel benötigt man  $2^{5 \cdot 20} = 2^{100}$  Arrayeinträge.

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

## Lösung

*Dieser Algorithmus ist nicht mit Quicksort, Heapsort, usw. vergleichbar*

- ▶ *denn er basiert nicht auf Vergleich von Elementen, sondern auf Häufigkeiten.*
- ▶ *Das funktioniert, indem wir Direkte-Adressierung (Einfügen, Suchen, Löschen in  $\Theta(1)$ ) ausnutzen.*

**Hauptproblem:** Übermäßiger Speicherbedarf für das Array.

- ▶ Zum Beispiel bei Strings mit 20 Zeichen (5 bit/Zeichen) als Schlüssel benötigt man  $2^{5 \cdot 20} = 2^{100}$  Arrayeinträge.
- ▶ Können wir diesen riesigen Speicherbedarf vermeiden und effizient bleiben?

# Counting Sort (II)

## Problem

Wir sortieren also mit Worst-Case Komplexität  $\Theta(n)$ , obwohl wir als untere Schranke  $\Theta(n \cdot \log n)$  bewiesen hatten?

## Lösung

*Dieser Algorithmus ist nicht mit Quicksort, Heapsort, usw. vergleichbar*

- ▶ *denn er basiert nicht auf Vergleich von Elementen, sondern auf Häufigkeiten.*
- ▶ *Das funktioniert, indem wir Direkte-Adressierung (Einfügen, Suchen, Löschen in  $\Theta(1)$ ) ausnutzen.*

**Hauptproblem:** Übermäßiger Speicherbedarf für das Array.

- ▶ Zum Beispiel bei Strings mit 20 Zeichen (5 bit/Zeichen) als Schlüssel benötigt man  $2^{5 \cdot 20} = 2^{100}$  Arrayeinträge.
- ▶ Können wir diesen riesigen Speicherbedarf vermeiden und effizient bleiben? **Ja!** – mit **Hashing**.

# Übersicht

## 1 Direkte Adressierung

- Counting Sort

## 2 Grundlagen des Hashings

## 3 Verkettung

## 4 Hashfunktionen

# Hashing (I)

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h.  $|K| \ll |U|$ .

⇒ Bei Direkter-Adressierung ist der größte Teil von  $T$  **verschwendet**.

# Hashing (I)

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h.  $|K| \ll |U|$ .

⇒ Bei Direkter-Adressierung ist der größte Teil von  $T$  **verschwendet**.

Das Ziel von **Hashing** ist:

- ▶ Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.

# Hashing (I)

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h.  $|K| \ll |U|$ .

⇒ Bei Direkter-Adressierung ist der größte Teil von  $T$  **verschwendet**.

Das Ziel von **Hashing** ist:

- ▶ Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.
- ▶ Dass zwei Schlüssel auf die selbe Zahl abgebildet werden, soll möglichst unwahrscheinlich sein.

# Hashing (I)

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h.  $|K| \ll |U|$ .

⇒ Bei Direkter-Adressierung ist der größte Teil von  $T$  **verschwendet**.

Das Ziel von **Hashing** ist:

- ▶ Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.
- ▶ Dass zwei Schlüssel auf die selbe Zahl abgebildet werden, soll möglichst unwahrscheinlich sein.

## Hashfunktion, Hashtabelle, Hashkollision

Eine **Hashfunktion** bildet einen Schlüssel auf einen Index der **Hashtabelle**  $T$  ab:

$$h : U \longrightarrow \{0, 1, \dots, m-1\} \text{ für Tabellengröße } m \text{ und } |U| = n.$$

# Hashing (I)

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h.  $|K| \ll |U|$ .

⇒ Bei Direkter-Adressierung ist der größte Teil von  $T$  **verschwendet**.

Das Ziel von **Hashing** ist:

- ▶ Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.
- ▶ Dass zwei Schlüssel auf die selbe Zahl abgebildet werden, soll möglichst unwahrscheinlich sein.

## Hashfunktion, Hashtabelle, Hashkollision

Eine **Hashfunktion** bildet einen Schlüssel auf einen Index der **Hashtabelle**  $T$  ab:

$$h : U \longrightarrow \{0, 1, \dots, m-1\} \text{ für Tabellengröße } m \text{ und } |U| = n.$$

Wir sagen, dass  $h(k)$  der **Hashwert** des Schlüssels  $k$  ist.

# Hashing (I)

Praktisch wird nur ein kleiner Teil der Schlüssel verwendet, d. h.  $|K| \ll |U|$ .

⇒ Bei Direkter-Adressierung ist der größte Teil von  $T$  **verschwendet**.

Das Ziel von **Hashing** ist:

- ▶ Einen extrem großen Schlüsselraum auf einen vernünftig kleinen Bereich von ganzen Zahlen abzubilden.
- ▶ Dass zwei Schlüssel auf die selbe Zahl abgebildet werden, soll möglichst unwahrscheinlich sein.

## Hashfunktion, Hashtabelle, Hashkollision

Eine **Hashfunktion** bildet einen Schlüssel auf einen Index der **Hashtabelle**  $T$  ab:

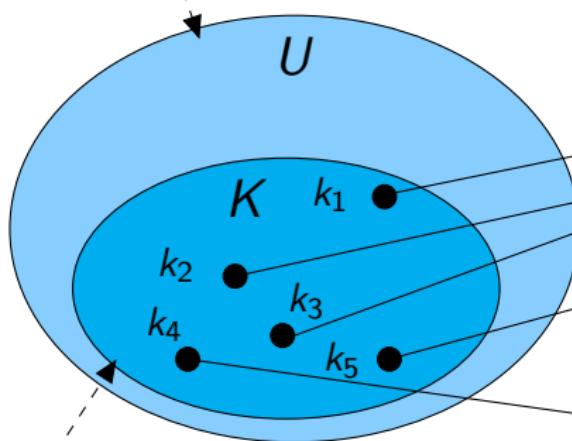
$$h : U \longrightarrow \{0, 1, \dots, m-1\} \text{ für Tabellengröße } m \text{ und } |U| = n.$$

Wir sagen, dass  $h(k)$  der **Hashwert** des Schlüssels  $k$  ist.

Das Auftreten von  $h(k) = h(k')$  für  $k \neq k'$  nennt man eine **Kollision**.

# Hashing (II)

*Schlüsselmenge*



*benutzte Schlüssel*

*Hashfunktion*

*Hashtabelle*

$h(k_1)$

$h(k_2) = h(k_3)$

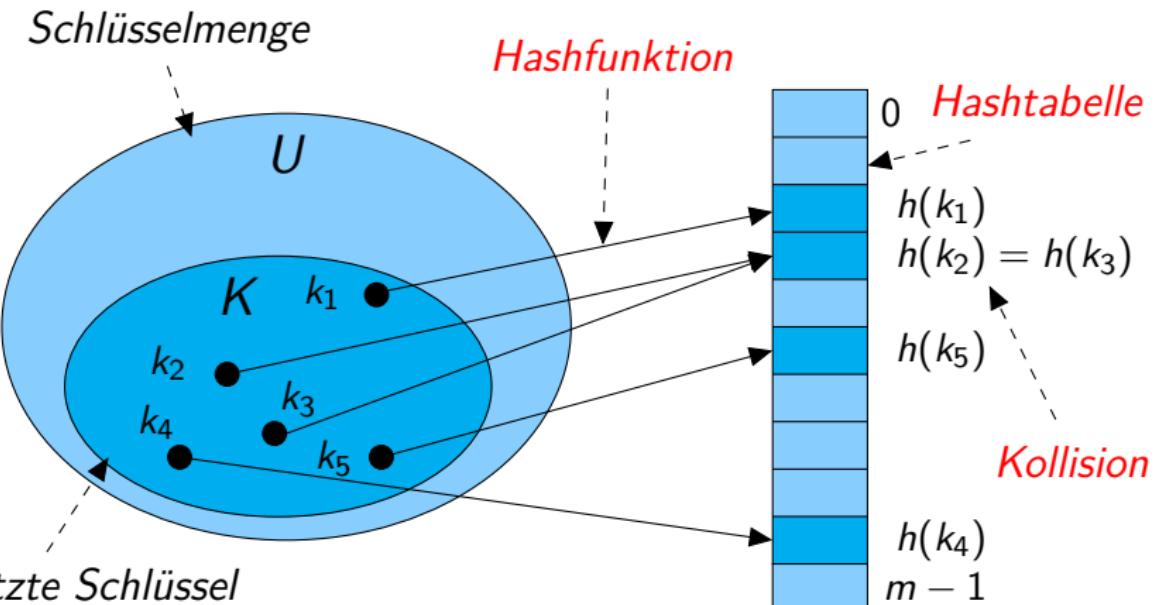
$h(k_5)$

$h(k_4)$

$m - 1$

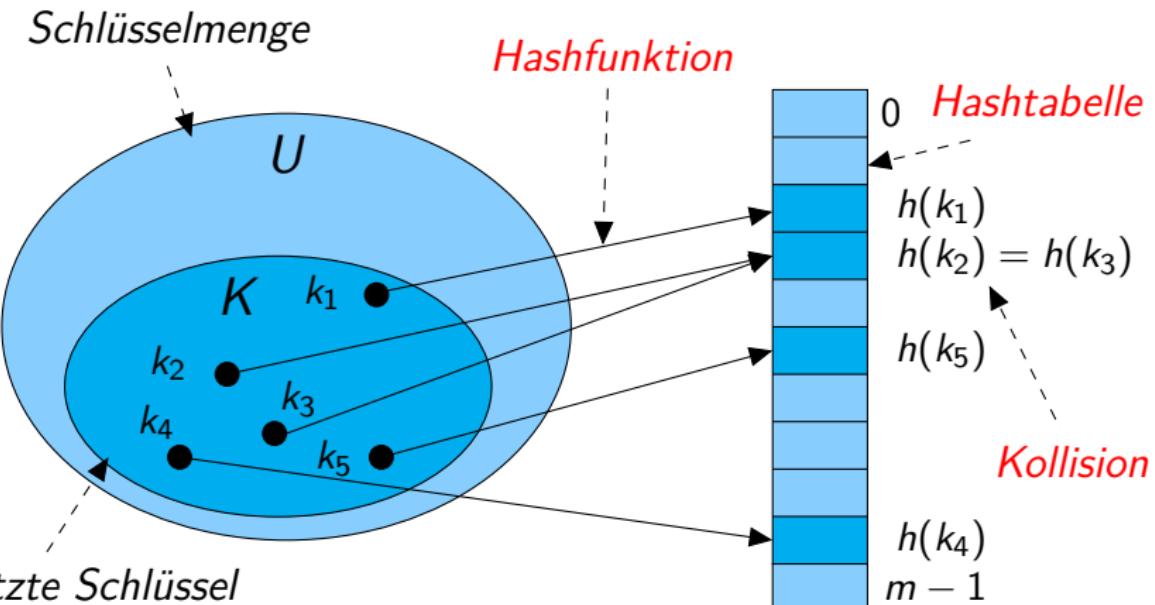
*Kollision*

# Hashing (II)



- Wie finden wir Hashfunktionen, die einfach auszurechnen sind und Kollisionen minimieren?

# Hashing (II)



- ▶ Wie finden wir Hashfunktionen, die einfach auszurechnen sind und Kollisionen minimieren?
- ▶ Wie behandeln wir dennoch auftretende Kollisionen?

# Kollisionen: Das Geburtstagsparadoxon (I)

Unsere Hashfunktion mag noch so gut sein,  
wir sollten auf Kollisionen vorbereitet sein!

# Kollisionen: Das Geburtstagsparadoxon (I)

Unsere Hashfunktion mag noch so gut sein,  
wir sollten auf Kollisionen vorbereitet sein!

Das liegt am

## Geburtstagsparadoxon

# Kollisionen: Das Geburtstagsparadoxon (I)

Unsere Hashfunktion mag noch so gut sein,  
wir sollten auf Kollisionen vorbereitet sein!

Das liegt am

## Geburtstagsparadoxon

- Die Wahrscheinlichkeit, dass dein Nachbar am selben Tag wie du Geburtstag hat ist  $\frac{1}{365} \approx 0,027$ .

# Kollisionen: Das Geburtstagsparadoxon (I)

Unsere Hashfunktion mag noch so gut sein,  
wir sollten auf Kollisionen vorbereitet sein!

Das liegt am

## Geburtstagsparadoxon

- ▶ Die Wahrscheinlichkeit, dass dein Nachbar am selben Tag wie du Geburtstag hat ist  $\frac{1}{365} \approx 0,027$ .
- ▶ Fragt man 23 Personen, wächst die Wahrscheinlichkeit auf  $\frac{23}{365} \approx 0,063$ .

# Kollisionen: Das Geburtstagsparadoxon (I)

Unsere Hashfunktion mag noch so gut sein,  
wir sollten auf Kollisionen vorbereitet sein!

Das liegt am

## Geburtstagsparadoxon

- Die Wahrscheinlichkeit, dass dein Nachbar am selben Tag wie du Geburtstag hat ist  $\frac{1}{365} \approx 0,027$ .
- Fragt man 23 Personen, wächst die Wahrscheinlichkeit auf  $\frac{23}{365} \approx 0,063$ .
- Sind aber 23 Personen in einem Raum, dann haben zwei von ihnen den selben Geburtstag mit Wahrscheinlichkeit

$$1 - \left( \frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \dots \cdot \frac{343}{365} \right) \approx 0,5$$

# Kollisionen: Das Geburtstagsparadoxon (II)

Auf Hashing angewendet bedeutet das:

- ▶ Die Wahrscheinlichkeit **keiner** Kollision nach  $k$  Einfügevorgängen in einer  $m$ -elementigen Tabelle ist:

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-k+1}{m} = \prod_{i=0}^{k-1} \frac{m-i}{m}$$

# Kollisionen: Das Geburtstagsparadoxon (II)

Auf Hashing angewendet bedeutet das:

- Die Wahrscheinlichkeit **keiner** Kollision nach  $k$  Einfügevorgängen in einer  $m$ -elementigen Tabelle ist:

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-k+1}{m} = \prod_{i=0}^{k-1} \frac{m-i}{m}$$

- Dieses Produkt geht gegen 0.

# Kollisionen: Das Geburtstagsparadoxon (II)

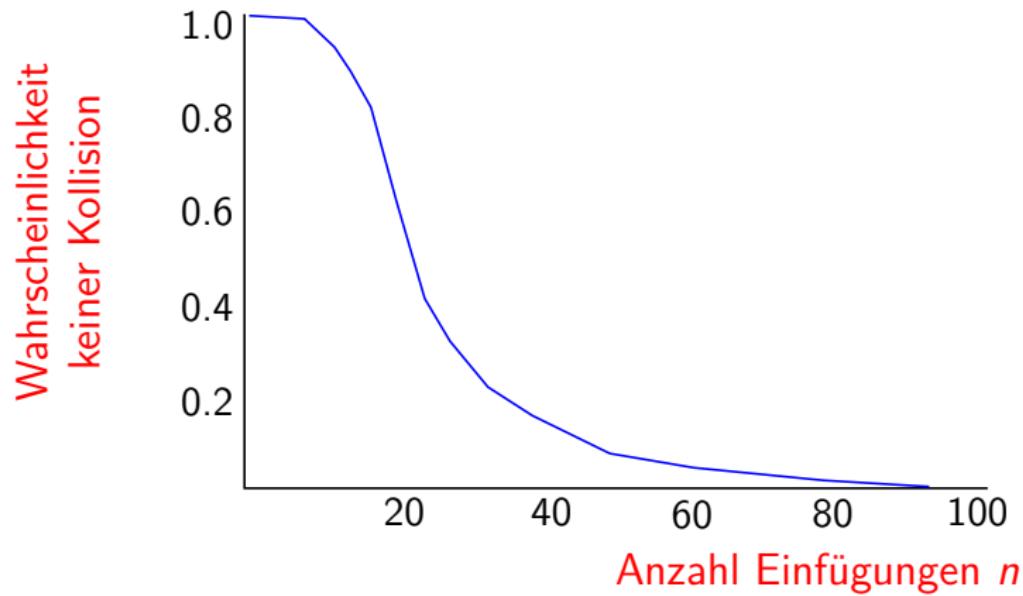
Auf Hashing angewendet bedeutet das:

- ▶ Die Wahrscheinlichkeit **keiner** Kollision nach  $k$  Einfügevorgängen in einer  $m$ -elementigen Tabelle ist:

$$\frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-k+1}{m} = \prod_{i=0}^{k-1} \frac{m-i}{m}$$

- ▶ Dieses Produkt geht gegen 0.
- ▶ Etwa bei  $m = 365$  ist die Wahrscheinlichkeit für  $k \geq 50$  praktisch 0.

# Kollisionen: Das Geburtstagsparadoxon (III)



# Übersicht

## 1 Direkte Adressierung

- Counting Sort

## 2 Grundlagen des Hashings

## 3 Verkettung

## 4 Hashfunktionen

# Kollisionsauflösung durch Verkettung (I)

## Idee

Alle Schlüssel, die zum gleichen Hash führen, werden in einer **verketteten Liste** gespeichert.

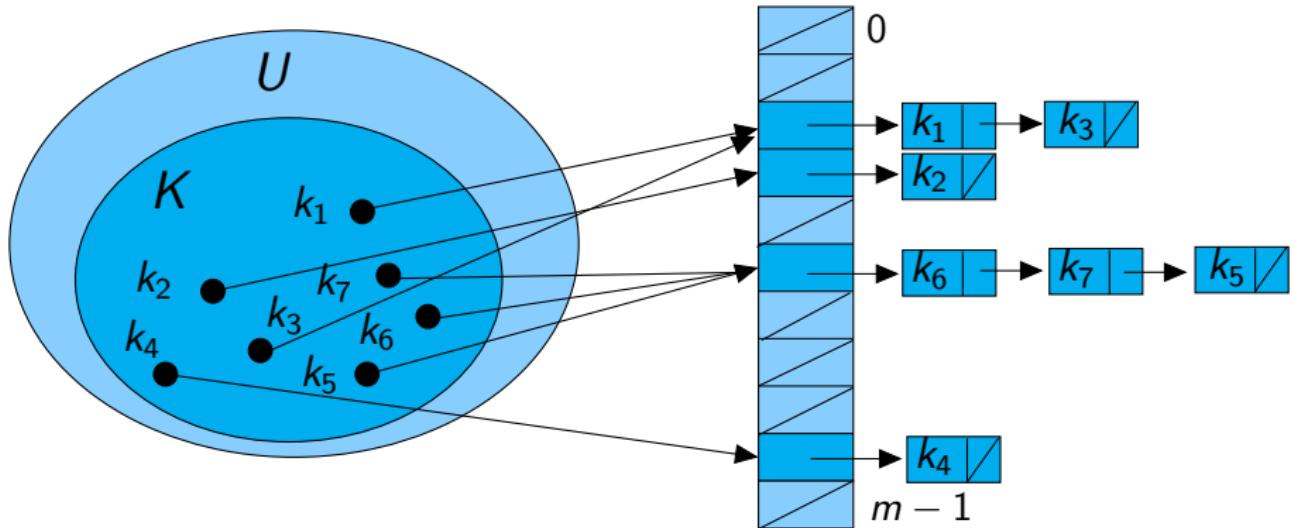
[Luhn 1953]

# Kollisionsauflösung durch Verkettung (I)

## Idee

Alle Schlüssel, die zum gleichen Hash führen, werden in einer **verketteten Liste** gespeichert.

[Luhn 1953]



# Kollisionsauflösung durch Verkettung (II)

## Dictionary-Operationen bei Verkettung (informell)

# Kollisionsauflösung durch Verkettung (II)

## Dictionary-Operationen bei Verkettung (informell)

- ▶ `hcSearch(int k)`: Suche nach einem Element mit Schlüssel  $k$  in der Liste  $T[h(k)]$ .

# Kollisionsauflösung durch Verkettung (II)

## Dictionary-Operationen bei Verkettung (informell)

- ▶ `hcSearch(int k)`: Suche nach einem Element mit Schlüssel  $k$  in der Liste  $T[h(k)]$ .
- ▶ `hcInsert(Element e)`: Setze Element  $e$  an den Anfang der Liste  $T[h(e.key)]$ .

# Kollisionsauflösung durch Verkettung (II)

## Dictionary-Operationen bei Verkettung (informell)

- ▶ `hcSearch(int k)`: Suche nach einem Element mit Schlüssel  $k$  in der Liste  $T[h(k)]$ .
- ▶ `hcInsert(Element e)`: Setze Element  $e$  an den Anfang der Liste  $T[h(e.key)]$ .
- ▶ `hcDelete(Element e)`: Lösche Element  $e$  aus der Liste  $T[h(e.key)]$ .

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ .

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ .  
Die Komplexität ist:

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ .  
Die Komplexität ist:

**Suche:** Proportional zur Länge der Liste  $T[h(k)]$ .

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ .  
Die Komplexität ist:

**Suche:** Proportional zur Länge der Liste  $T[h(k)]$ .

**Einfügen:** Konstant (ohne Überprüfung, ob das Element schon vorhanden ist).

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ .  
Die Komplexität ist:

**Suche:** Proportional zur Länge der Liste  $T[h(k)]$ .

**Einfügen:** Konstant (ohne Überprüfung, ob das Element schon vorhanden ist).

**Löschen:** Proportional zur Länge der Liste  $T[h(k)]$ .

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ . Die Komplexität ist:

**Suche:** Proportional zur Länge der Liste  $T[h(k)]$ .

**Einfügen:** Konstant (ohne Überprüfung, ob das Element schon vorhanden ist).

**Löschen:** Proportional zur Länge der Liste  $T[h(k)]$ .

- ▶ Im Worst-Case haben alle Schüssel den selben Hashwert.

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ . Die Komplexität ist:

**Suche:** Proportional zur Länge der Liste  $T[h(k)]$ .

**Einfügen:** Konstant (ohne Überprüfung, ob das Element schon vorhanden ist).

**Löschen:** Proportional zur Länge der Liste  $T[h(k)]$ .

- ▶ Im Worst-Case haben alle Schüssel den selben Hashwert.
- ▶ Suche und Löschen hat dann die selbe Worst-Case Komplexität wie Listen:  $\Theta(n)$ .

# Kollisionsauflösung durch Verkettung (III)

## Worst-Case Komplexität

Angenommen, die Berechnung von  $h(k)$  ist recht effizient, etwa  $\Theta(1)$ . Die Komplexität ist:

**Suche:** Proportional zur Länge der Liste  $T[h(k)]$ .

**Einfügen:** Konstant (ohne Überprüfung, ob das Element schon vorhanden ist).

**Löschen:** Proportional zur Länge der Liste  $T[h(k)]$ .

- ▶ Im Worst-Case haben alle Schüssel den selben Hashwert.
  - ▶ Suche und Löschen hat dann die selbe Worst-Case Komplexität wie Listen:  $\Theta(n)$ .
- 
- ▶ Im Average-Case ist Hashing mit Verkettung aber dennoch effizient!

# Average-Case-Analyse von Verkettung (I)

Annahmen:

- ▶ Es gebe  $n$  mögliche Schlüssel und  $m$  Hashtabellenpositionen,  $n \gg m$ .

# Average-Case-Analyse von Verkettung (I)

Annahmen:

- ▶ Es gebe  $n$  mögliche Schlüssel und  $m$  Hashtabellenpositionen,  $n \gg m$ .
- ▶ **Gleichverteiltes Hashing:** Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Schlüssel auf jedes der  $m$  Slots abgebildet.

# Average-Case-Analyse von Verkettung (I)

Annahmen:

- ▶ Es gebe  $n$  mögliche Schlüssel und  $m$  Hashtabellenpositionen,  $n \gg m$ .
- ▶ **Gleichverteiltes Hashing:** Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Schlüssel auf jedes der  $m$  Slots abgebildet.
- ▶ Der Hashwert  $h(k)$  kann in konstanter Zeit berechnet werden.

# Average-Case-Analyse von Verkettung (I)

Annahmen:

- ▶ Es gebe  $n$  mögliche Schlüssel und  $m$  Hashtabellenpositionen,  $n \gg m$ .
- ▶ **Gleichverteiltes Hashing:** Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Schlüssel auf jedes der  $m$  Slots abgebildet.
- ▶ Der Hashwert  $h(k)$  kann in konstanter Zeit berechnet werden.

## $O$ , $\Theta$ , $\Omega$ erweitert

Aus technischen Gründen erweitern wir die Definition von  $O$ ,  $\Theta$  und  $\Omega$  auf Funktionen mit zwei Parametern.

# Average-Case-Analyse von Verkettung (I)

Annahmen:

- ▶ Es gebe  $n$  mögliche Schlüssel und  $m$  Hashtabellenpositionen,  $n \gg m$ .
- ▶ **Gleichverteiltes Hashing:** Jeder Schlüssel wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Schlüssel auf jedes der  $m$  Slots abgebildet.
- ▶ Der Hashwert  $h(k)$  kann in konstanter Zeit berechnet werden.

## $O$ , $\Theta$ , $\Omega$ erweitert

Aus technischen Gründen erweitern wir die Definition von  $O$ ,  $\Theta$  und  $\Omega$  auf Funktionen mit zwei Parametern.

- ▶ Beispielsweise ist  $g \in O(f)$  gdw.  
$$\exists c > 0, n_0, m_0 \text{ mit } \forall n \geq n_0, m \geq m_0 : 0 \leq g(n, m) \leq c \cdot f(n, m)$$

# Average-Case-Analyse von Verkettung (II)

- ▶ Der Füllgrad der Hashtabelle  $\mathbb{T}$  ist  $\alpha(n, m) = \frac{n}{m}$ .

# Average-Case-Analyse von Verkettung (II)

- Der **Füllgrad** der Hashtabelle  $T$  ist  $\alpha(n, m) = \frac{n}{m}$ .
- ⇒ Auch die durchschnittliche Länge der Liste  $T[h(k)]$  ist  $\alpha$ !

# Average-Case-Analyse von Verkettung (II)

- ▶ Der **Füllgrad** der Hashtabelle  $T$  ist  $\alpha(n, m) = \frac{n}{m}$ .
- ⇒ Auch die durchschnittliche Länge der Liste  $T[h(k)]$  ist  $\alpha$ !
- ▶ Wieviele Elemente aus  $T[h(k)]$  müssen nun im Schnitt untersucht werden, um den Schlüssel  $k$  zu finden?

# Average-Case-Analyse von Verkettung (II)

- ▶ Der Füllgrad der Hashtabelle  $T$  ist  $\alpha(n, m) = \frac{n}{m}$ .
- ⇒ Auch die durchschnittliche Länge der Liste  $T[h(k)]$  ist  $\alpha$ !
- ▶ Wieviele Elemente aus  $T[h(k)]$  müssen nun im Schnitt untersucht werden, um den Schlüssel  $k$  zu finden?
- ⇒ Unterscheide **erfolgreiche** von **erfolgloser** Suche (wie in Vorlesung 1).

# Average-Case-Analyse von Verkettung (III)

## Erfolglose Suche

Die erfolglose Suche benötigt  $\Theta(1 + \alpha)$  Zeit im Average-Case.

# Average-Case-Analyse von Verkettung (III)

## Erfolglose Suche

Die erfolglose Suche benötigt  $\Theta(1 + \alpha)$  Zeit im Average-Case.

- ▶ Die erwartete Zeit, um Schlüssel  $k$  zu finden ist gerade die Zeit, um die Liste  $T[h(k)]$  zu durchsuchen.

# Average-Case-Analyse von Verkettung (III)

## Erfolglose Suche

Die erfolglose Suche benötigt  $\Theta(1 + \alpha)$  Zeit im Average-Case.

- ▶ Die erwartete Zeit, um Schlüssel  $k$  zu finden ist gerade die Zeit, um die Liste  $T[h(k)]$  zu durchsuchen.
- ▶ Die erwartete Länge dieser Liste ist  $\alpha$ .

# Average-Case-Analyse von Verkettung (III)

## Erfolglose Suche

Die erfolglose Suche benötigt  $\Theta(1 + \alpha)$  Zeit im Average-Case.

- ▶ Die erwartete Zeit, um Schlüssel  $k$  zu finden ist gerade die Zeit, um die Liste  $T[h(k)]$  zu durchsuchen.
- ▶ Die erwartete Länge dieser Liste ist  $\alpha$ .
- ▶ Das Berechnen von  $h(k)$  benötige nur eine Zeiteinheit.

# Average-Case-Analyse von Verkettung (III)

## Erfolglose Suche

Die erfolglose Suche benötigt  $\Theta(1 + \alpha)$  Zeit im Average-Case.

- ▶ Die erwartete Zeit, um Schlüssel  $k$  zu finden ist gerade die Zeit, um die Liste  $T[h(k)]$  zu durchsuchen.
  - ▶ Die erwartete Länge dieser Liste ist  $\alpha$ .
  - ▶ Das Berechnen von  $h(k)$  benötige nur eine Zeiteinheit.
- ⇒ Insgesamt erhält man  $1 + \alpha$  Zeiteinheiten im Durchschnitt.

# Average-Case-Analyse von Verkettung (IV)

## Erfolgreiche Suche

Die erfolgreiche Suche benötigt im Average-Case auch  $\Theta(1 + \alpha)$ .

# Average-Case-Analyse von Verkettung (IV)

## Erfolgreiche Suche

Die erfolgreiche Suche benötigt im Average-Case auch  $\Theta(1 + \alpha)$ .

- Sei  $k_i$  der  $i$ -te eingefügte Schlüssel und  $A(k_i)$  die erwartete Zeit, um  $k_i$  zu finden:

# Average-Case-Analyse von Verkettung (IV)

## Erfolgreiche Suche

Die erfolgreiche Suche benötigt im Average-Case auch  $\Theta(1 + \alpha)$ .

- Sei  $k_i$  der  $i$ -te eingefügte Schlüssel und  $A(k_i)$  die erwartete Zeit, um  $k_i$  zu finden:

$$A(k_i) = 1 + \begin{array}{l} \text{Durchschnittliche Anzahl Schlüssel,} \\ \text{die in } T[h(k_i)] \text{ erst nach } k_i \text{ eingefügt wurden} \end{array}$$

# Average-Case-Analyse von Verkettung (IV)

## Erfolgreiche Suche

Die erfolgreiche Suche benötigt im Average-Case auch  $\Theta(1 + \alpha)$ .

- Sei  $k_i$  der  $i$ -te eingefügte Schlüssel und  $A(k_i)$  die erwartete Zeit, um  $k_i$  zu finden:

$$A(k_i) = 1 + \begin{array}{l} \text{Durchschnittliche Anzahl Schlüssel,} \\ \text{die in } T[h(k_{-i})] \text{ erst nach } k_i \text{ eingefügt wurden} \end{array}$$

- Annahme von gleichverteiltem Hashing ergibt:  $A(k_i) = 1 + \sum_{j=i+1}^n \frac{1}{m}$

# Average-Case-Analyse von Verkettung (IV)

## Erfolgreiche Suche

Die erfolgreiche Suche benötigt im Average-Case auch  $\Theta(1 + \alpha)$ .

- Sei  $k_i$  der  $i$ -te eingefügte Schlüssel und  $A(k_i)$  die erwartete Zeit, um  $k_i$  zu finden:

$$A(k_i) = 1 + \begin{array}{l} \text{Durchschnittliche Anzahl Schlüssel,} \\ \text{die in } T[h(k_{-i})] \text{ erst nach } k_i \text{ eingefügt wurden} \end{array}$$

- Annahme von gleichverteiltem Hashing ergibt:  $A(k_i) = 1 + \sum_{j=i+1}^n \frac{1}{m}$
- Durchschnitt über alle  $n$  Einfügungen in die Hashtabelle:  $\frac{1}{n} \sum_{i=1}^n A(k_i)$

# Average-Case-Analyse von Verkettung (IV)

Die erwartete Anzahl an untersuchten Elementen bei einer erfolgreichen Suche ist:

# Average-Case-Analyse von Verkettung (IV)

Die erwartete Anzahl an untersuchten Elementen bei einer erfolgreichen Suche ist:

$$\begin{aligned}
 & \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \frac{1}{m} \right) && \text{Summe aufteilen} \\
 &= \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{nm} \sum_{i=1}^n \sum_{j=i+1}^n 1 && \text{Vereinfachen} \\
 &= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) && \text{Summe } 1 \dots n - 1 \\
 &= 1 + \frac{1}{nm} \cdot \frac{n(n - 1)}{2} && \text{Vereinfachen} \\
 &= 1 + \frac{n - 1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} && \text{und damit in } \Theta(1 + \alpha)
 \end{aligned}$$

# Komplexität der Dictionary-Operationen mit Verkettung

- ▶ Vorausgesetzt die Anzahl der Einträge  $m$  ist (wenigstens) proportional zu  $n$ ,

# Komplexität der Dictionary-Operationen mit Verkettung

- ▶ Vorausgesetzt die Anzahl der Einträge  $m$  ist (wenigstens) proportional zu  $n$ ,
- ▶ dann ist der Füllgrad  $\alpha(n, m) = \frac{n}{m} \in \frac{O(m)}{m} = O(1)$ .

# Komplexität der Dictionary-Operationen mit Verkettung

- ▶ Vorausgesetzt die Anzahl der Einträge  $m$  ist (wenigstens) proportional zu  $n$ ,
- ▶ dann ist der Füllgrad  $\alpha(n, m) = \frac{n}{m} \in \frac{O(m)}{m} = O(1)$ .
- ▶ Damit benötigen alle Operationen im Durchschnitt  $O(1)$ .

# Komplexität der Dictionary-Operationen mit Verkettung

- ▶ Vorausgesetzt die Anzahl der Einträge  $m$  ist (wenigstens) proportional zu  $n$ ,
- ▶ dann ist der Füllgrad  $\alpha(n, m) = \frac{n}{m} \in \frac{O(m)}{m} = O(1)$ .
- ▶ Damit benötigen alle Operationen im Durchschnitt  $O(1)$ .
- ▶ Weil das auch *Suche* mit einschließt, können wir im Average-Case mit  $O(n)$  sortieren.

# Übersicht

## 1 Direkte Adressierung

- Counting Sort

## 2 Grundlagen des Hashings

## 3 Verkettung

## 4 Hashfunktionen

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,
  - ▶ sie sollte alle Indizes mit möglichst **gleicher Häufigkeit** verwenden, und

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,
  - ▶ sie sollte alle Indizes mit möglichst **gleicher Häufigkeit** verwenden, und
  - ▶ **ähnliche Schlüssel** möglichst breit auf die Hashtabelle **verteilen**.

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,
  - ▶ sie sollte alle Indizes mit möglichst **gleicher Häufigkeit** verwenden, und
  - ▶ **ähnliche Schlüssel** möglichst breit auf die Hashtabelle **verteilen**.
- ▶ Drei oft verwendete Techniken, eine „gute“ Hashfunktion zu erhalten:

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,
  - ▶ sie sollte alle Indizes mit möglichst **gleicher Häufigkeit** verwenden, und
  - ▶ **ähnliche Schlüssel** möglichst breit auf die Hashtabelle **verteilen**.
- ▶ Drei oft verwendete Techniken, eine „gute“ Hashfunktion zu erhalten:
  - ▶ Die **Divisionsmethode**,

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,
  - ▶ sie sollte alle Indizes mit möglichst **gleicher Häufigkeit** verwenden, und
  - ▶ **ähnliche Schlüssel** möglichst breit auf die Hashtabelle **verteilen**.
- ▶ Drei oft verwendete Techniken, eine „gute“ Hashfunktion zu erhalten:
  - ▶ Die **Divisionsmethode**,
  - ▶ die **Multiplikationsmethode**, und

# Hashfunktionen

## Hashfunktion

- ▶ Eine **Hashfunktion** bildet einen Schlüssel auf eine ganze Zahl (d. h. einen Index) ab.
- ▶ Was macht eine „gute“ Hashfunktion aus?
  - ▶ Die Hashfunktion  $h(k)$  sollte **einfach zu berechnen** sein,
  - ▶ sie sollte **surjektiv** auf der Menge  $0 \dots m-1$  sein,
  - ▶ sie sollte alle Indizes mit möglichst **gleicher Häufigkeit** verwenden, und
  - ▶ **ähnliche Schlüssel** möglichst breit auf die Hashtabelle **verteilen**.
- ▶ Drei oft verwendete Techniken, eine „gute“ Hashfunktion zu erhalten:
  - ▶ Die **Divisionsmethode**,
  - ▶ die **Multiplikationsmethode**, und
  - ▶ **universelles Hashing**.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.
  - ▶ Für  $m = 2^p$  ist  $h(k)$  einfach die letzten  $p$  Bits.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.
  - ▶ Für  $m = 2^p$  ist  $h(k)$  einfach die letzten  $p$  Bits.
  - ▶ Besser ist es,  $h(k)$  abhängig von mehreren Bits zu machen.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.
  - ▶ Für  $m = 2^p$  ist  $h(k)$  einfach die letzten  $p$  Bits.
  - ▶ Besser ist es,  $h(k)$  abhängig von mehreren Bits zu machen.
- ▶ Gute Wahl ist  $m$  prim und nicht zu nah an einer Zweierpotenz.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.
  - ▶ Für  $m = 2^p$  ist  $h(k)$  einfach die letzte  $p$  Bits.
  - ▶ Besser ist es,  $h(k)$  abhängig von mehreren Bits zu machen.
- ▶ Gute Wahl ist  $m$  prim und nicht zu nah an einer Zweierpotenz.

## Beispiel

- ▶ Strings mit 2000 Zeichen als Schlüssel.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.
  - ▶ Für  $m = 2^p$  ist  $h(k)$  einfach die letzte  $p$  Bits.
  - ▶ Besser ist es,  $h(k)$  abhängig von mehreren Bits zu machen.
- ▶ Gute Wahl ist  $m$  prim und nicht zu nah an einer Zweierpotenz.

## Beispiel

- ▶ Strings mit 2000 Zeichen als Schlüssel.
- ▶ Wir erlauben durchschnittlich 3 Sondierungen für die erfolglose Suche.

# Divisionsmethode

## Divisionsmethode

Hashfunktion:  $h(k) = k \bmod m$

- ▶ Bei dieser Methode muss der Wert von  $m$  sorgfältig gewählt werden.
  - ▶ Für  $m = 2^p$  ist  $h(k)$  einfach die letzte  $p$  Bits.
  - ▶ Besser ist es,  $h(k)$  abhängig von mehreren Bits zu machen.
- ▶ Gute Wahl ist  $m$  prim und nicht zu nah an einer Zweierpotenz.

## Beispiel

- ▶ Strings mit 2000 Zeichen als Schlüssel.
- ▶ Wir erlauben durchschnittlich 3 Sondierungen für die erfolglose Suche.  
⇒ Wähle  $m \approx 2000/3 \rightarrow 701$ .

# Multiplikationsmethode (I)

## Multiplikationsmethode

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$  für  $0 < c < 1$

# Multiplikationsmethode (I)

## Multiplikationsmethode

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$  für  $0 < c < 1$

- ▶  $k \cdot c \bmod 1$  ist der Nachkommateil von  $k \cdot c$ , d. h.  $k \cdot c - \lfloor k \cdot c \rfloor$ .

# Multiplikationsmethode (I)

## Multiplikationsmethode

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$  für  $0 < c < 1$

- ▶  $k \cdot c \bmod 1$  ist der Nachkommateil von  $k \cdot c$ , d. h.  $k \cdot c - \lfloor k \cdot c \rfloor$ .
- ▶ Knuth empfiehlt  $c \approx (\sqrt{5} - 1)/2 \approx 0,62$ .

# Multiplikationsmethode (I)

## Multiplikationsmethode

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$  für  $0 < c < 1$

- ▶  $k \cdot c \bmod 1$  ist der Nachkommateil von  $k \cdot c$ , d. h.  $k \cdot c - \lfloor k \cdot c \rfloor$ .
  - ▶ Knuth empfiehlt  $c \approx (\sqrt{5} - 1)/2 \approx 0,62$ .
- ⇒ Der Wert von  $m$  ist hier nicht kritisch.

# Multiplikationsmethode (II)

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ .

# Multiplikationsmethode (II)

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ .

- ▶ Das übliche Vorgehen nimmt  $m = 2^p$  und  $c = \frac{s}{2^w}$ , wobei  $0 < s < 2^w$ .

# Multiplikationsmethode (II)

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ .

- ▶ Das übliche Vorgehen nimmt  $m = 2^p$  und  $c = \frac{s}{2^w}$ , wobei  $0 < s < 2^w$ .  
Dann:
  - ▶ Berechne zunächst  $k \cdot s$  ( $= k \cdot c \cdot 2^w$ ).

# Multiplikationsmethode (II)

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ .

- ▶ Das übliche Vorgehen nimmt  $m = 2^p$  und  $c = \frac{s}{2^w}$ , wobei  $0 < s < 2^w$ .  
Dann:
  - ▶ Berechne zunächst  $k \cdot s$  ( $= k \cdot c \cdot 2^w$ ).
  - ▶ Teile durch  $2^w$ , verwende nur die Nachkommastellen.

# Multiplikationsmethode (II)

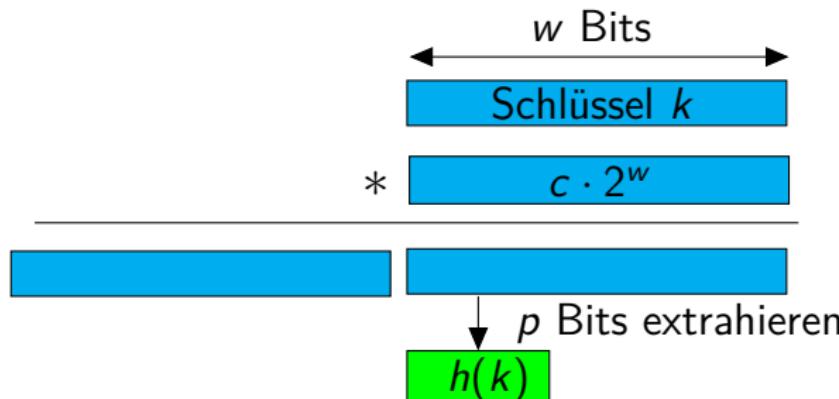
Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ .

- ▶ Das übliche Vorgehen nimmt  $m = 2^p$  und  $c = \frac{s}{2^w}$ , wobei  $0 < s < 2^w$ .  
Dann:
  - ▶ Berechne zunächst  $k \cdot s$  ( $= k \cdot c \cdot 2^w$ ).
  - ▶ Teile durch  $2^w$ , verwende nur die Nachkommastellen.
  - ▶ Multipliziere mit  $2^p$  und verwende nur den ganzzahligen Anteil.

# Multiplikationsmethode (II)

Hashfunktion:  $h(k) = \lfloor m \cdot (k \cdot c \bmod 1) \rfloor$ .

- Das übliche Vorgehen nimmt  $m = 2^p$  und  $c = \frac{s}{2^w}$ , wobei  $0 < s < 2^w$ . Dann:
  - Berechne zunächst  $k \cdot s$  ( $= k \cdot c \cdot 2^w$ ).
  - Teile durch  $2^w$ , verwende nur die Nachkommastellen.
  - Multipliziere mit  $2^p$  und verwende nur den ganzzahligen Anteil.



# Universelles Hashing (I)

Das größte Problem beim Hashing ist,

- ▶ dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

# Universelles Hashing (I)

Das größte Problem beim Hashing ist,

- dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

## Idee

Wähle **zufällig** eine Hashfunktion aus einer gegebenen kleinen Menge  $H$ , unabhängig von den verwendeten Schlüsseln.

# Universelles Hashing (I)

Das größte Problem beim Hashing ist,

- dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

## Idee

Wähle **zufällig** eine Hashfunktion aus einer gegebenen kleinen Menge  $H$ , unabhängig von den verwendeten Schlüsseln.

Eine Menge Hashfunktionen ist **universell**, wenn

- der Anteil der Funktionen aus  $H$ , so dass  $k$  und  $k'$  kollidieren ist  $\frac{|H|}{m}$ .

# Universelles Hashing (I)

Das größte Problem beim Hashing ist,

- dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

## Idee

Wähle **zufällig** eine Hashfunktion aus einer gegebenen kleinen Menge  $H$ , unabhängig von den verwendeten Schlüsseln.

Eine Menge Hashfunktionen ist **universell**, wenn

- der Anteil der Funktionen aus  $H$ , so dass  $k$  und  $k'$  kollidieren ist  $\frac{|H|}{m}$ .
- d. h., die W'lichkeit einer Kollision von  $k$  und  $k'$  ist  $\frac{1}{|H|} \cdot \frac{|H|}{m} = \frac{1}{m}$ .

Für universelles Hashing ist die erwartete Länge der Liste  $T[k]$

# Universelles Hashing (I)

Das größte Problem beim Hashing ist,

- dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

## Idee

Wähle **zufällig** eine Hashfunktion aus einer gegebenen kleinen Menge  $H$ , unabhängig von den verwendeten Schlüsseln.

Eine Menge Hashfunktionen ist **universell**, wenn

- der Anteil der Funktionen aus  $H$ , so dass  $k$  und  $k'$  kollidieren ist  $\frac{|H|}{m}$ .
- d. h., die W'lichkeit einer Kollision von  $k$  und  $k'$  ist  $\frac{1}{|H|} \cdot \frac{|H|}{m} = \frac{1}{m}$ .

Für universelles Hashing ist die erwartete Länge der Liste  $T[k]$

1. Gleich  $\alpha$ , wenn  $k$  nicht in  $T$  enthalten ist.

# Universelles Hashing (I)

Das größte Problem beim Hashing ist,

- dass es immer eine ungünstige Sequenz von Schlüsseln gibt, die auf den selben Slot abgebildet werden.

## Idee

Wähle **zufällig** eine Hashfunktion aus einer gegebenen kleinen Menge  $H$ , unabhängig von den verwendeten Schlüsseln.

Eine Menge Hashfunktionen ist **universell**, wenn

- der Anteil der Funktionen aus  $H$ , so dass  $k$  und  $k'$  kollidieren ist  $\frac{|H|}{m}$ .
- d. h., die W'lichkeit einer Kollision von  $k$  und  $k'$  ist  $\frac{1}{|H|} \cdot \frac{|H|}{m} = \frac{1}{m}$ .

Für universelles Hashing ist die erwartete Länge der Liste  $T[k]$

1. Gleich  $\alpha$ , wenn  $k$  nicht in  $T$  enthalten ist.
2. Gleich  $1+\alpha$ , wenn  $k$  in  $T$  enthalten ist.

# Universelles Hashing (II)

## Beispiel

Definiere die Elemente der Klasse  $H$  von Hashfunktionen durch:

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

- $p$  sei Primzahl mit  $p > m$  und  $p >$  größter Schlüssel.
- Die ganzen Zahlen  $a$  ( $1 \leq a < p$ ) und  $b$  ( $0 \leq b < p$ ) werden erst bei der Ausführung gewählt.

# Universelles Hashing (II)

## Beispiel

Definiere die Elemente der Klasse  $H$  von Hashfunktionen durch:

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

- $p$  sei Primzahl mit  $p > m$  und  $p >$  größter Schlüssel.
- Die ganzen Zahlen  $a$  ( $1 \leq a < p$ ) und  $b$  ( $0 \leq b < p$ ) werden erst bei der Ausführung gewählt.

Die Klasse der obigen Hashfunktionen  $h_{a,b}$  ist universell.