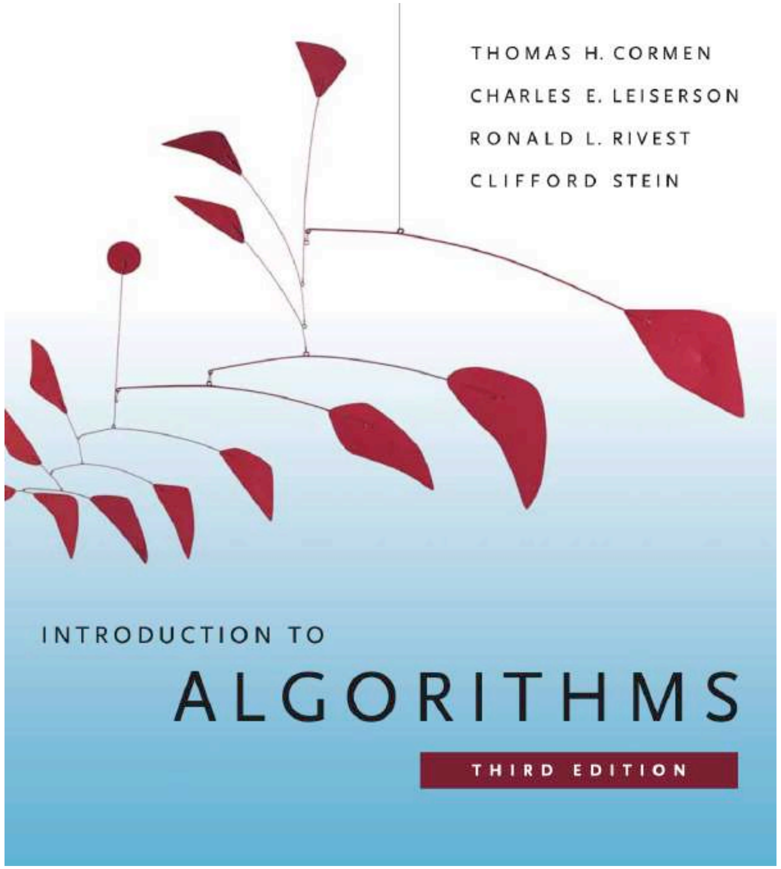
Algorithmen und Datenstrukturen

Prof. Dr. Thomas Wiemann - FB AI

Hochschule Fulda
University of Applied Sciences

Inhalt



Gliederung

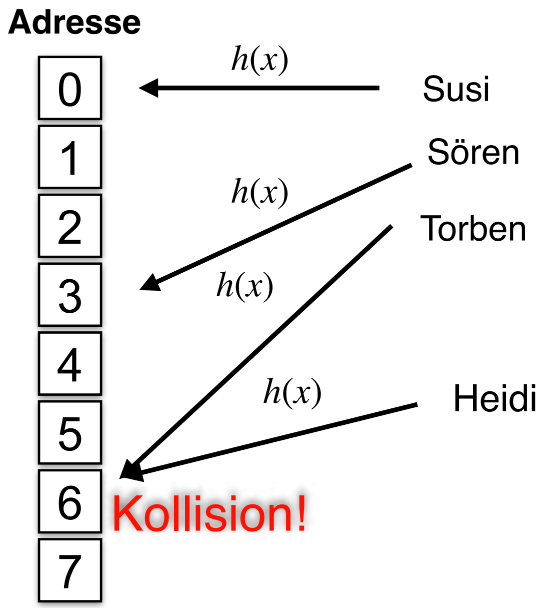
1. Laufzeit und Komplexität
2. Sortieren
3. Abstrakte Datentypen
4. **Hashing**
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

https://edutechlearners.com/download/Introduction_to_algorithms-3rd Edition.pdf

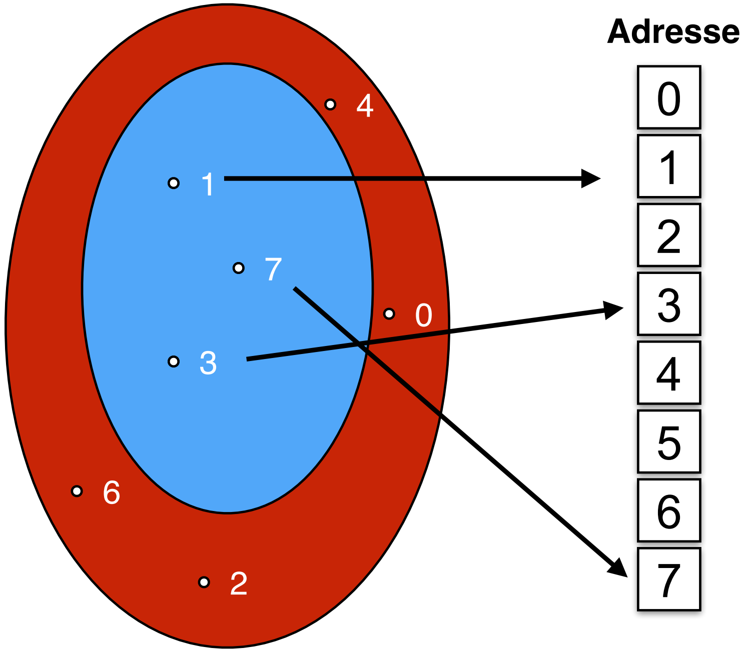
Hashing

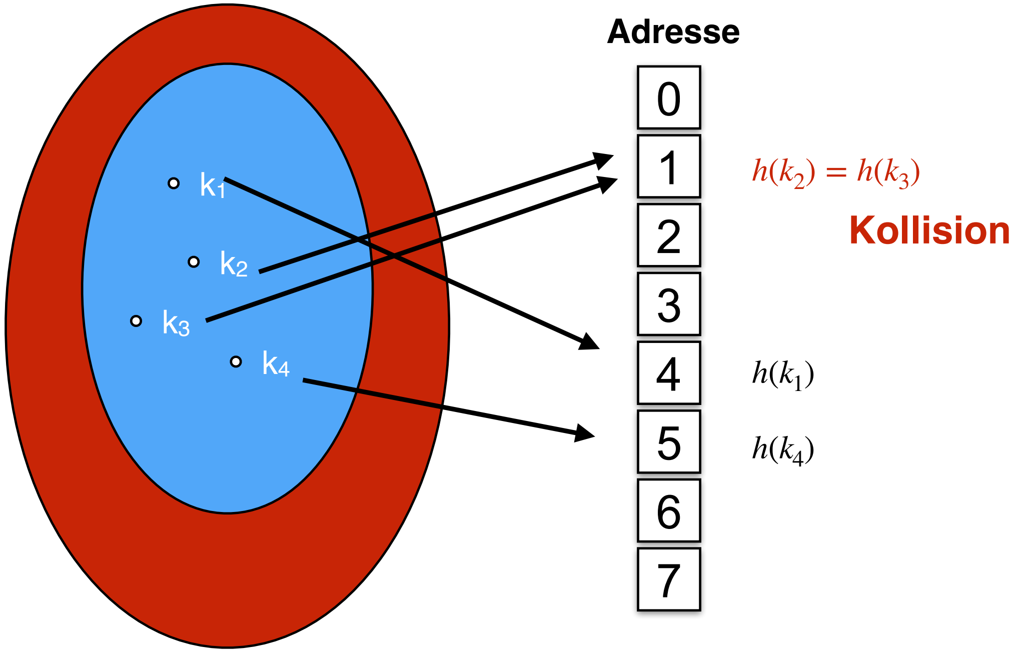
- Verfahren zum Suchen von Einträgen in großen Datenmengen
- Ansatz: Nutze eine mathematische Funktion (Hashfunktion), um eine Position in einer Tabelle (Hashtabelle) zu berechnen
- Direktes auffinden der Einträge durch direkten Lookup
- Kein “Suchen” über Schlüssel erforderlich

$$h : \text{Objekte} \rightarrow \mathbb{N}$$



Direkte Adressierung vs. Hashing





Hashing

- Häufige Anwendungen: Dictionaries / Key-Value-Stores
- Erforderliche Operationen:
  - Suche nach einem Datensatz *d* bei gegebenem Schlüssel *x*: `search(x)`
  - Suche nach einem Datensatz *d* ohne Schlüssel: `search(d)`
  - Einfügen eines neuen Datensatzes: `insert(x,d)` oder `insert(d)`
  - Entfernen eines Datensatzes: `delete(d)`
- Menge möglicher Schlüssel (**Universum**) kann **sehr** groß ein!

Hashing - Begriffe

- Menge U möglicher Schlüssel sehr groß, aktuelle Schlüsselmenge S nur eine kleine Teilmenge von U
- S ist möglicherweise nicht bekannt
- Ziel: Durch Berechnung feststellen, wo Datensatz mit Schlüssel x gespeichert wird
- Abspeicherung der Datensätze in einem Array T mit Indizes $\{0,1,\ldots,m-1\}$ (**Hashtabelle**)
- Hashfunktion h liefert für jeden Schlüssel $x \in U$ eine Adresse in der der Hashtabelle
- $h : U \rightarrow \{0,1,\ldots,m-1\}$
- Wie wähle ich h und m?
- $h(x) = h(y)$ für $x \neq y$ ➠ **Kollision**
- $h(x_1) \neq h(x_2) \neq h(x_3)\ldots \forall x_i \in U$ ➠ **Perfekte Hashfunktion**

Anforderungen an Hashfunktionen

- Möglichst Gleichverteilung der Hashwerte für den Eingabewertbereich
- Keine “Löcher” im Wertebereich
- Schnell berechenbar
- Je nach Anwendung:
  - Bei sortiertem Zugriff ist Ordnungserhalt wünschenswert
  - Gute Diffusion - ähnliche Eingaben sollen zu völlig unterschiedlichen Hashwerten führen
  - Vom Hashwert soll nicht auf den Eingabewert geschlossen werden können
- Beispiele / Diskussion

$$h(x) = x \mod 10$$

$$h(x) = \text{Quersumme}(x)$$

$$h(x) = x \mod 7$$

$$h(x) = \lfloor x/2 \rfloor$$

Beispiel einer einfachen Hashfunktion

- Idee einer Hashfunktion für Java-Objekte
- Sei `x` ein beliebiges Objekt
- Dann ist `x.toString()` seine Stringrepräsentation
- Wie baue ich daraus eine Hashfunktion $h(x) \rightarrow \{0 \ldots 16\}$?

$$x = x_{n-1}x_{n-2}\ldots x_1x_0$$

$$h(x) = \left(\sum_{i=0}^{N} \mathrm{int}(x) \right) \mod 17$$

- Zur Behandlung von Kollisionen gibt es die Verfahren **offenes** und **geschlossenes** Hashing

Perfekte Hashfunktion - Beispiel

- Erinnerung: $h(x_1) \neq h(x_2) \neq h(x_3) \ldots \forall x_i \in U$ ➠ **Perfekte Hashfunktion**
- Gesucht wird eine Hashfunktion, die auf den Elementen keine Kollision verursacht

Gesucht:	h : braun,	rot,	blau,	violett,	türkis $\rightarrow \{0 \ldots 4\}$
Länge (w) =	5	3	4	7	6
Länge $(w - 3)$ =	2	0	1	4	3

➠ $h(w) =$ Länge$(w) - 3 \in [0..4]$ ist hier eine perfekte Hashfunktion

Multiplikation und Division

- Hashing durch Division
  - $h(x) = x \mod \text{N}$
  - Sehr schnell
  - N sollte keine Potenz einer Zahl sein
  - N sollte eine Primzahl sein, die nicht zu nah an einer Zweipotenz liegt
- Hashing durch Multiplikation
  - $h(x) = \lfloor x \cdot c \mod 1 \rfloor$
  - Länge der Hashtabelle ist hier irrelevant
  - Schnelle Implementierung mit Zweierpotenz
  - Funktioniert mit jeder reellen Zahl c

Beispiel: Division

- N sollte eine Primzahl sein, die nicht zu nah an einer Zweipotenz liegt

$h_1(x) = x \mod 16$ $h_2(x) = x \mod 17$

$x_1 = 34 \quad \rightarrow h_1(x_1) = 2 \quad 0010\ 0010 \quad \rightarrow h_2(x_1) = 0$

$x_2 = 50 \quad \rightarrow h_1(x_2) = 2 \quad 0011\ 0010 \quad \rightarrow h_2(x_2) = 16$

$x_3 = 66 \quad \rightarrow h_1(x_3) = 2 \quad 0100\ 0010 \quad \rightarrow h_2(x_3) = 15$

$x_4 = 82 \quad \rightarrow h_1(x_4) = 2 \quad 0101\ 0010 \quad \rightarrow h_2(x_4) = 14$

Ergebnis hängt nur von den niedrigwertigsten Bits ab!

Bekannte Hashfunktionen

- MD2, MD4, MD5, SHA (Kryptograhie)
- CRC (Prüfsummen)
- …

Beispiele für Nicht-kryptografische Hashfunktionen (Wikipedia)

Hashfunktion	Geschwindigkeit	Entwickler	Jahr
xxHash	5,4 GB/s	Yann Collet	2012
MurmurHash 3a	2,7 GB/s	Austin Appleby	2008
SBox	1,4 GB/s	Bret Mulvey	2007
Lookup3	1,2 GB/s	Bob Jenkins	2006
CityHash64	1,05 GB/s	Geoff Pike & Jyrki Alakuijala	2011
FNV	0,55 GB/s	Fowler, Noll, Vo	1991
SipHash/HighwayHash[4]		Jan Wassenberg & Jyrki Alakuijala	2016 / 2012

Beispiel MD5 (Wikipedia)

MD5-Hashwert [Bearbeiten | Quelltext bearbeiten]

Die 128 Bit langen MD5-Hashwerte werden üblicherweise als 32-stellige Hexadezimalzahl notiert. Beispiel für eine 59 Byte lange ASCII-Eingabe mit zugehörigem MD5-Hashwert:

```
md5("Franz jagt im komplett verwahrlosten Taxi quer durch Bayern") =
a3cca2b2aa1e3b5b3b5aad99a8529074
```

Es ist praktisch unmöglich, eine weitere Nachricht, die genau diesen Hashwert ergibt, zu bestimmen. Eine beliebige Änderung des Textes (im Folgenden wird nur ein Buchstabe verändert) erzeugt aufgrund des Lawineneffekts einen komplett anderen Hashwert:

```
md5("Frank jagt im komplett verwahrlosten Taxi quer durch Bayern") =
7e716d0e702df0505fc72e2b89467910
```

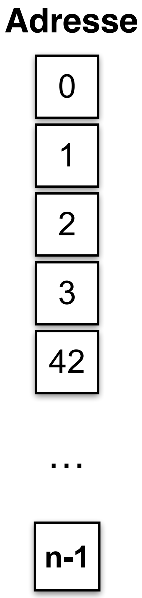
Der Hash einer Zeichenfolge der Länge null ist:

```
md5("") =
d41d8cd98f00b204e9800998ecf8427e
```

Offenes Hashing

- Idee: Array von Listen
- Implementierung des Set-Interfaces

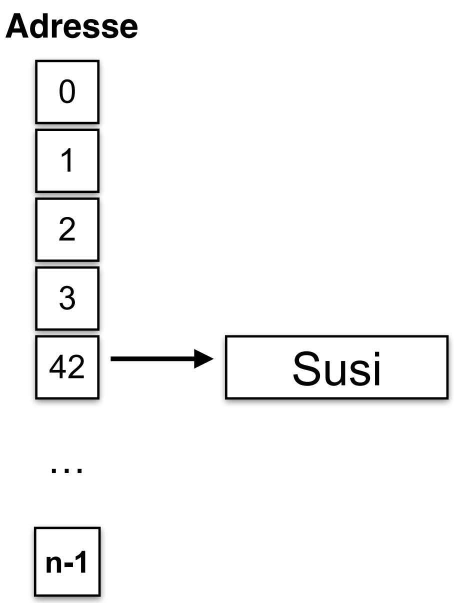
$$h(x) \rightarrow \{0 \ldots N-1\}$$



Offenes Hashing

- Idee: Array von Listen
- Implementierung des Set-Interfaces

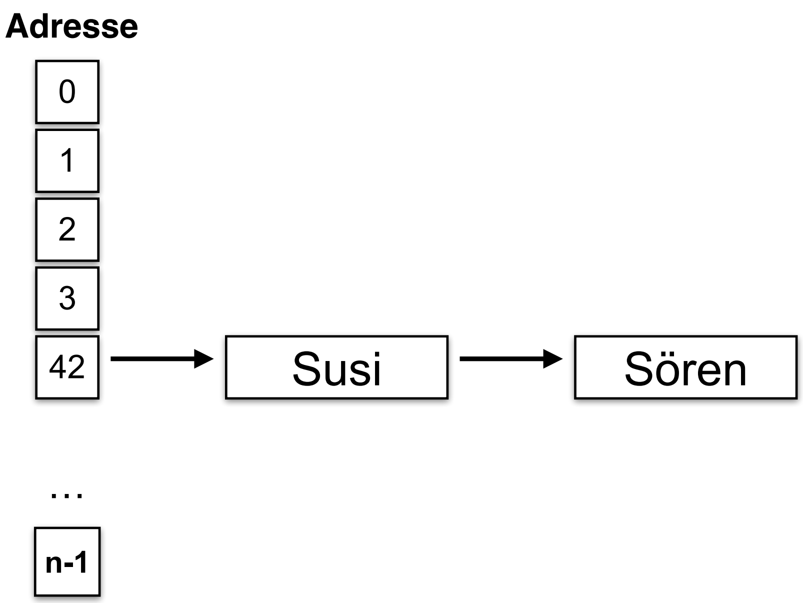
$$h(x) \rightarrow \{0 \ldots N-1\}$$



Offenes Hashing

- Idee: Array von Listen
- Implementierung des Set-Interfaces

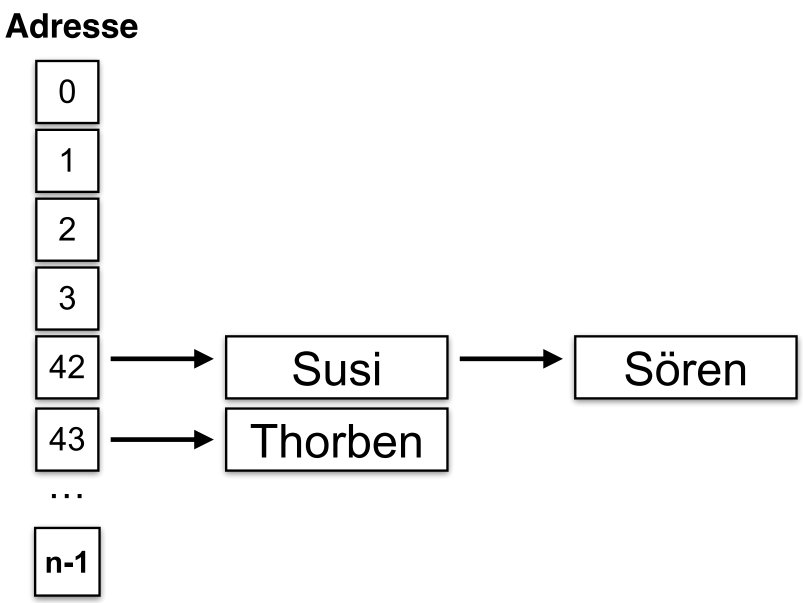
$$h(x) \rightarrow \{0 \ldots N-1\}$$



Offenes Hashing

- Idee: Array von Listen
- Implementierung des Set-Interfaces

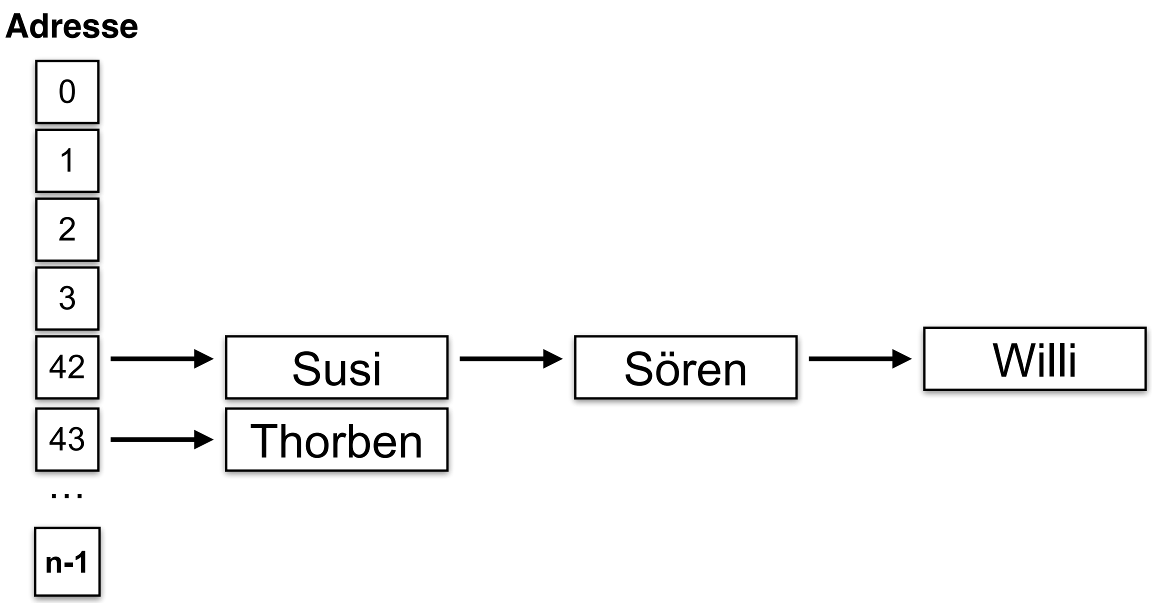
$$h(x) \rightarrow \{0 \ldots N-1\}$$



Offenes Hashing

- Idee: Array von Listen
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$



Beispielhafte Implementierung

```
public class OpenHash implements Set {
    private List[] b;

    public OpenHash(int N) {
        b = new List[N];
        for(int i = 0; i < N; i++) {
            b[i] = new LinkedList();
        }
    }

    public boolean insert(Comparable x) {
        int i = hash(x);
        b[i].reset();
        while(!b[i].end() && x.compareTo(b[i].current() != 0)
        {
            b[i].advance();
        }
        if(x.compareTo(b[i].current) == 0) return false;
        b[i].insert(x);
        return true;
    }
}
```

```
public Comparable find(Comparable x) {
    int i = hash(x);
    return b[i].find(x);
}

public boolean delete(Comparable x) {
    int i = hash(x);
    return b[i].delete(x);
}
```

Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

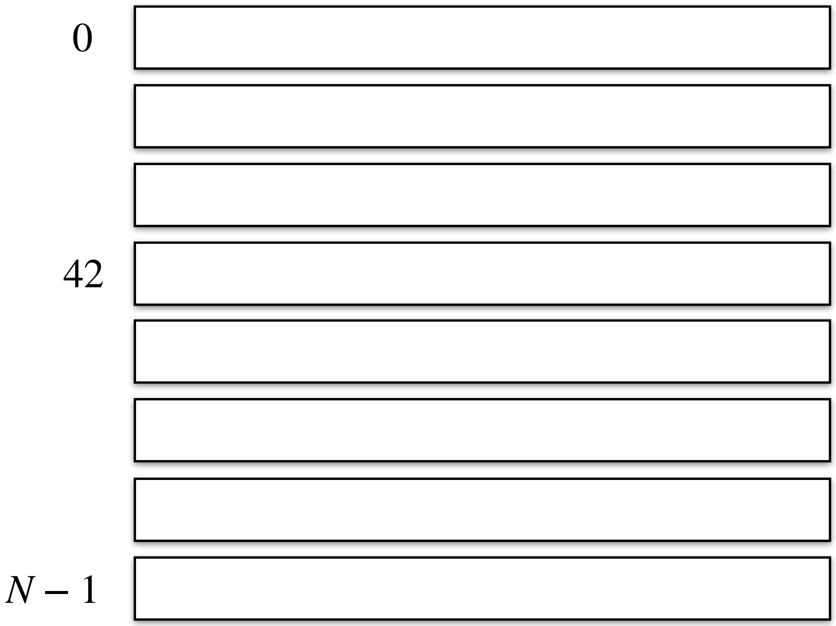
$h(x) \rightarrow \{0 \ldots N-1\}$

Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

Einfügen von Susi: h(Susi) = 42



Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

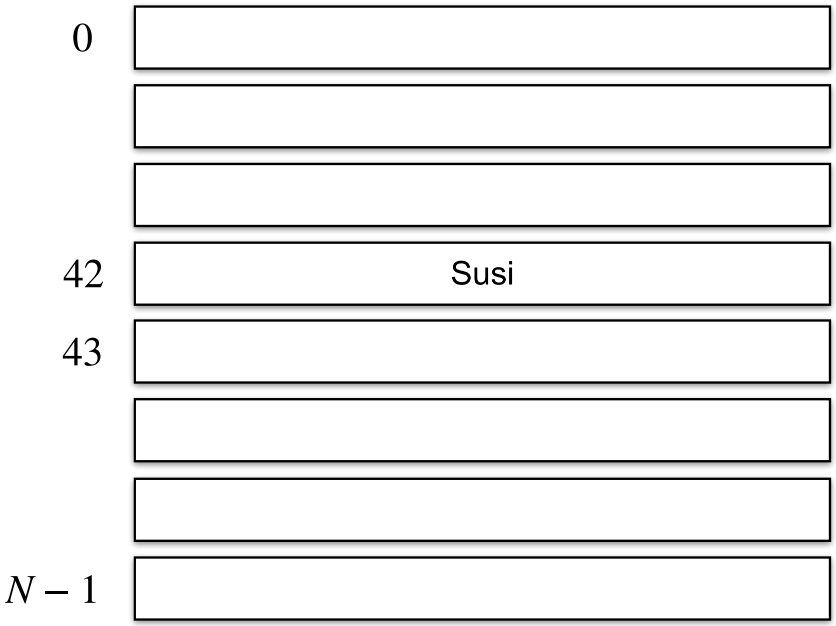
$$h(x) \rightarrow \{0 \ldots N-1\}$$

Geschlossenes Hashing

Einfügen von Susi: h(Sören) = 42

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

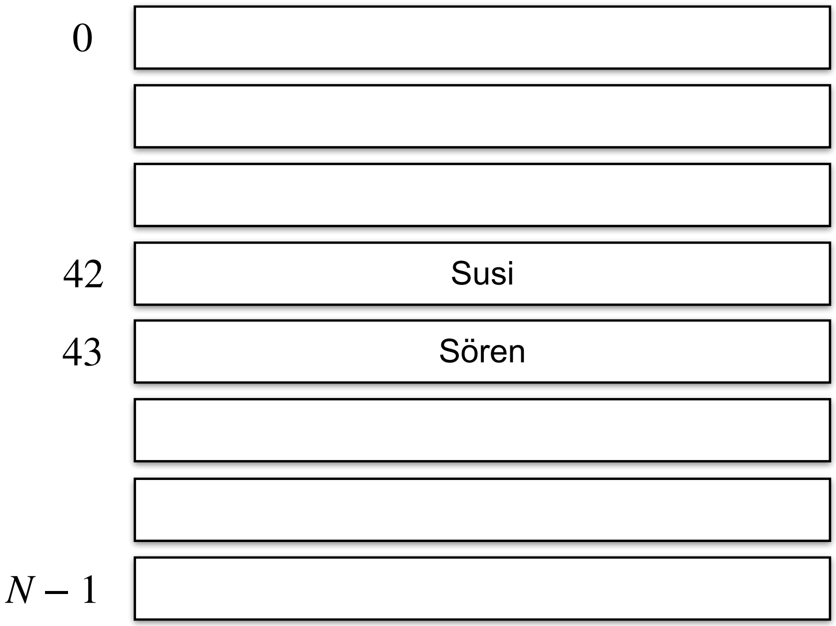


Geschlossenes Hashing

Einfügen von Susi: h(Sören) = 42

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \to \{0 \ldots N-1\}$$

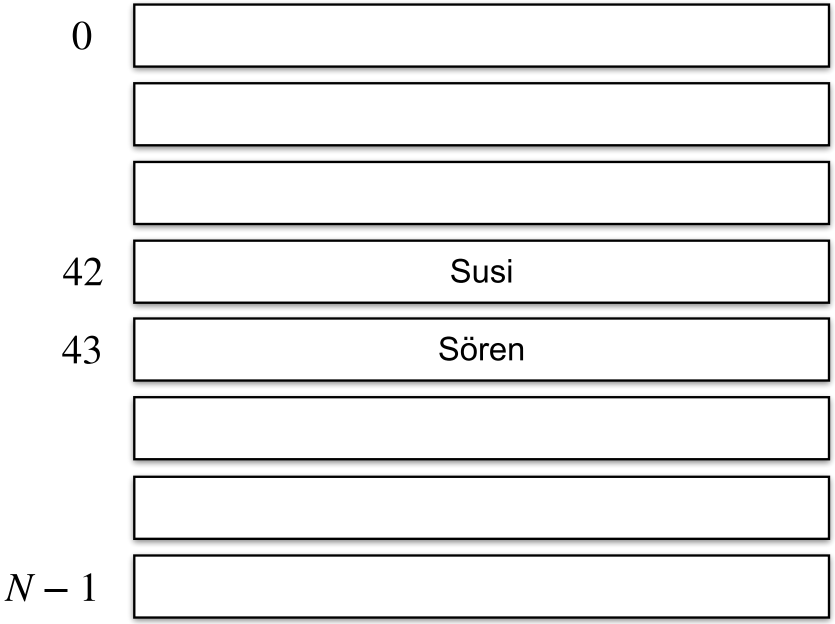


Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

Einfügen von Susi: $h(\text{Willi}) = 42$

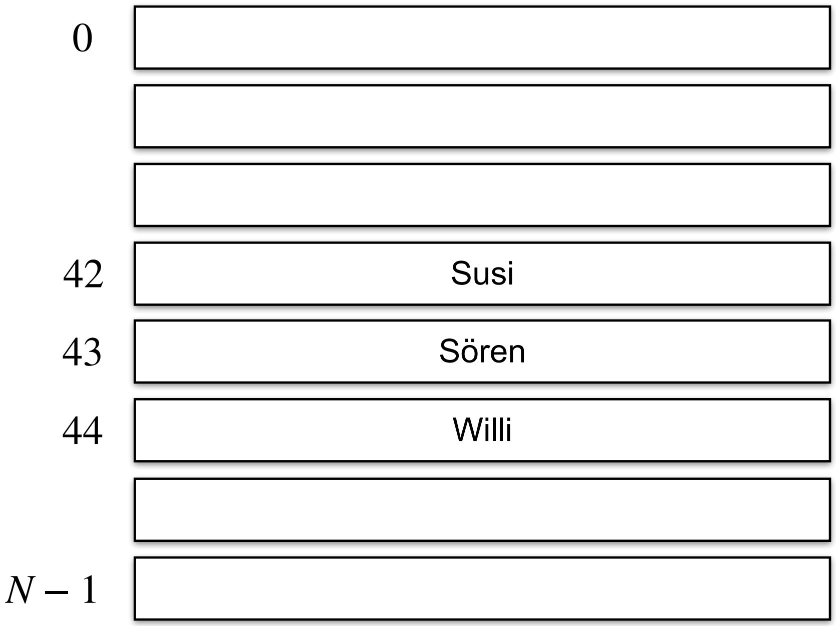
$$h(x) \rightarrow \{0 \ldots N-1\}$$



Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

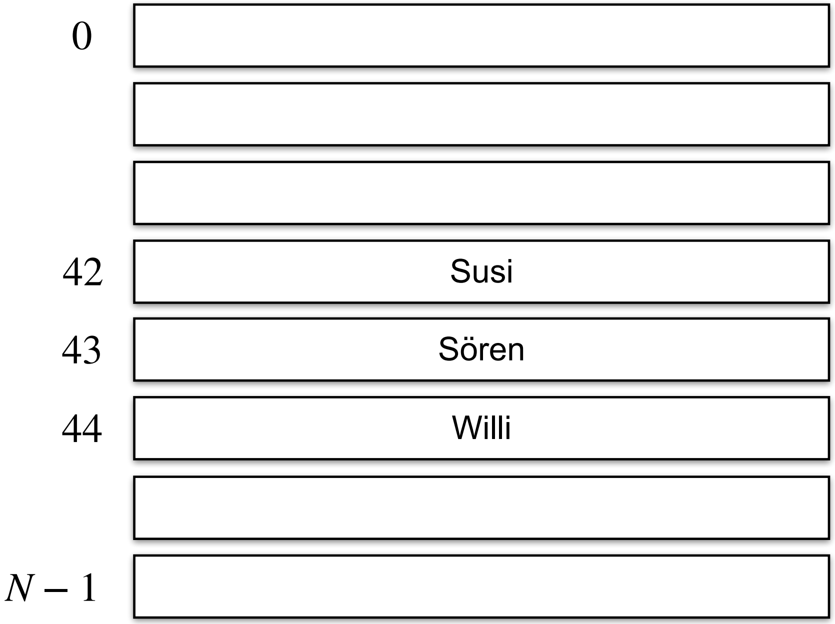


Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

Löschen von Sören: $h(\text{Sören}) = 42$

$$h(x) \rightarrow \{0 \ldots N-1\}$$



Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

Löschen von Sören: h(Willi) = 42

0	
42	Susi
43	
44	Willi
$N-1$	

Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

Löschen von Sören: h(Sören) = 42

0		
42	Susi	
43	Sören	
44	Willi	
$N-1$		

Mögliche Einträge:

- Belegt (B)
- Leer (L)
- Gelöscht (G)

Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

Löschen von Sören: h(Sören) = 42

0		L
		L
		L
42	Susi	B
43	Sören	G
44	Willi	B
		L
$N-1$		L

Mögliche Einträge:

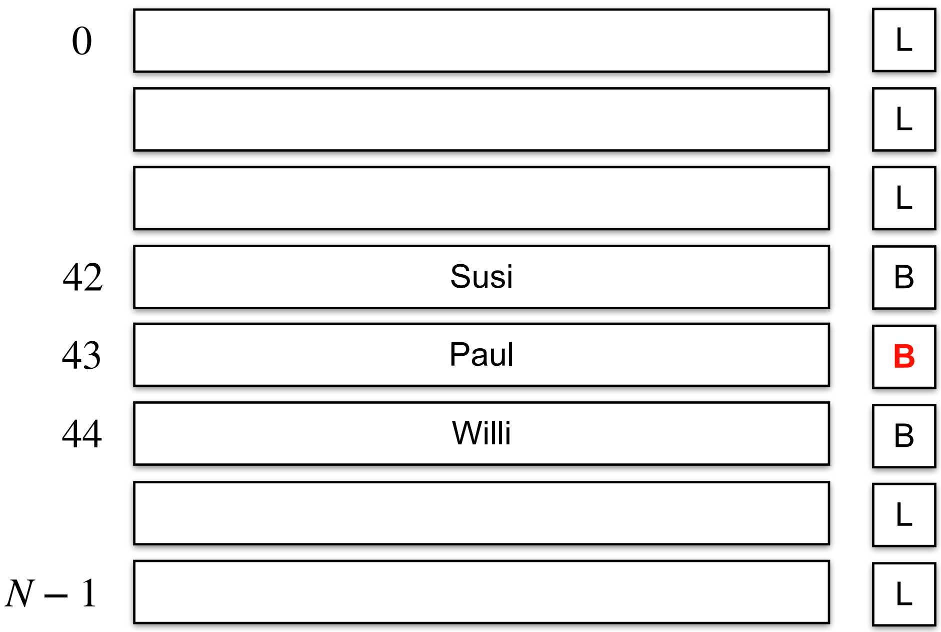
- Belegt (B)
- Leer (L)
- Gelöscht (G)

Geschlossenes Hashing

- Array von Objekten
- Implementierung des Set-Interfaces

$$h(x) \rightarrow \{0 \ldots N-1\}$$

Löschen von Paul: h(Paul) = 42



Mögliche Einträge:

- Belegt (B)
- Leer (L)
- Gelöscht (G)

Sondierungsstrategien

- Im Beispiel: “Lineares Sondieren”
  - $(y, y+1, y+2, y+3) \mod N$
  - Neigung zur Bildung von Clustern
- Quadratisches Sondieren
  - $(y, y+1, y+4, y+9) \mod N$
  - Nicht jede Stelle des Arrays kann erreicht werden
  - Es können noch freie Plätze da sein
  - Man kann Fälle konstruieren, indem die Hälfte der Tabelle noch frei ist
- Double Hashing
  - $y, y + h_2(x), y + 2h_2(x), y + 3h_2(x) \ldots$

Laufzeit bei geschlossenem Hashing

Sie fügen mittels geschlossenem Hashing unter Verwendung von linearem Sondieren 100 Objekte in eine zunächst leere Tabelle der Größe 1000 ein. Wieviele Operationen sind im Best-Case, wieviele im Worst-Case nötig?

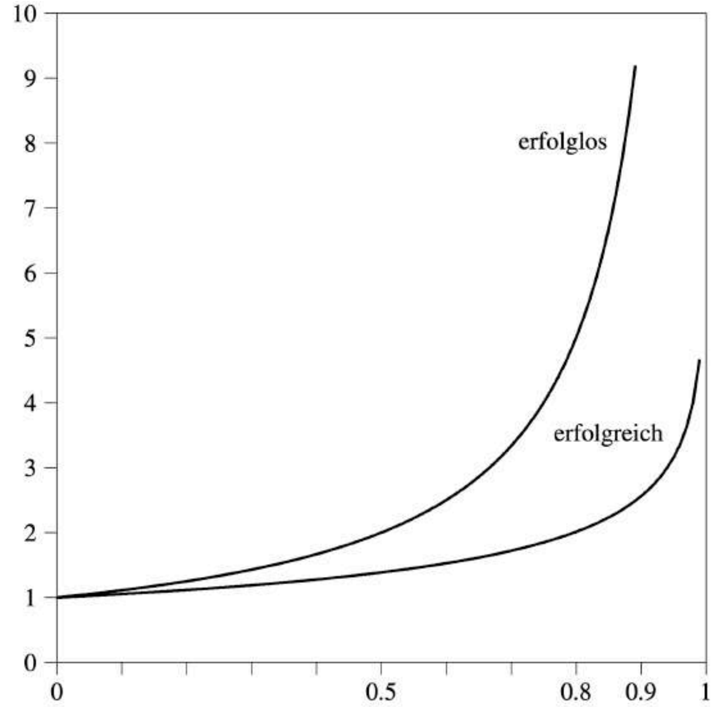
- Best Case: Perfekte Hash-Funktion ⇒ Keine Kollisionen ⇒ 100
- Worst Case: Hashfunktion liefert für jedem Wert das gleiche Ergebnis
  - Eine Operation für den ersten Wert
  - Zwei Operationen für den zweiten Wert
  - Drei Operationen für den dritten Wert
  - …
  - Insgesamt also $\sum_{i=0}^{100} = 5050$

Laufzeit bei geschlossenem Hashing

- Sei n die Anzahl der in der Hashtabelle gespeicherten Objekte
- N ist die Anzahl der möglichen Speicherpositionen

Auslastungsfaktor: $\alpha = \frac{n}{N} \leq 1$

- Anzahl der Schritte bei Double-Hashing als Strategie:
  - Erfolglose Suche: $\approx \frac{1}{1-\alpha} = 5.0$, für $\alpha = 0.8$
  - Erfolgreicher Suche: $\approx -\frac{\ln(1-\alpha)}{\alpha} = 2.01$,für $\alpha = 0.8$



Struktur einer Implementierung

```
public class HashTable implements Set {
    private Comparable[] array;
    private byte[] flag;

    public HashTable(int N){
        array = new Comparable[N];
        for(i = 0; i < N; i++) flag[i] = EMPTY;
    }

    ...
 }
```