# Algorithmen und Datenstrukturen

Zsolt István , SS 2025

## 05
## Probabilistische Datenstrukturen

bisher:

**deterministische Datenstrukturen**

Verhalten für identische Eingaben immer gleich

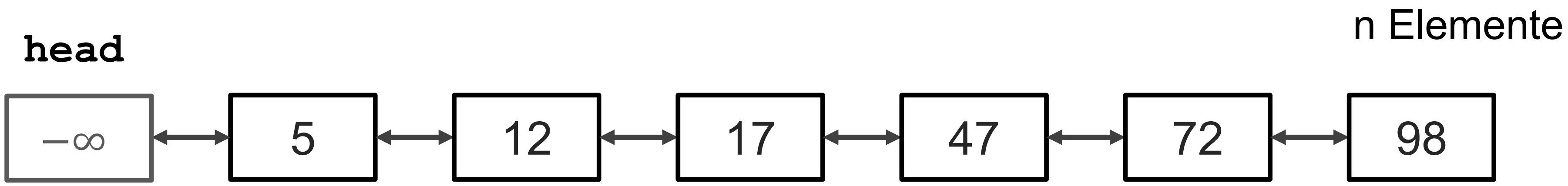
in diesem Abschnitt:

**randomisierte Datenstrukturen**

Verhalten hängt auch von zufälligen Entscheidungen der Datenstruktur ab

# Skip Lists

# Suchen/Löschen/Einfügen in Sortierter Liste



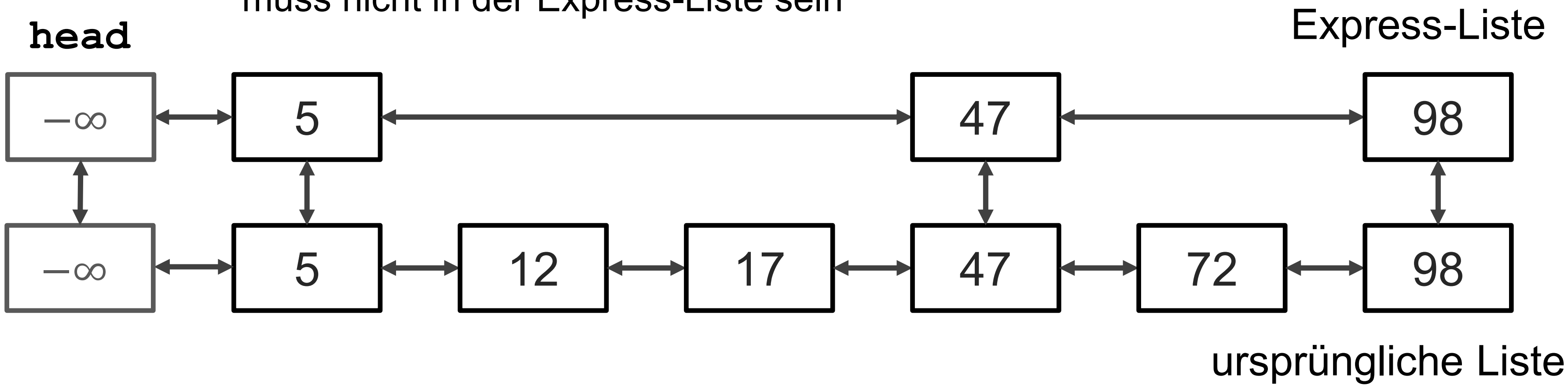

| Operation | Laufzeit* |
|---|---|
| Suchen | $\Omega(n)$ |
| Löschen (Wert) | $\Omega(n)$ |
| Einfügen | $\Omega(n)$ |

*Worst Case

# Idee von Skip-Listen

Füge „Express-Liste" mit einigen Elemente ein:

Bemerkung: Das erste „richtige" Element
muss nicht in der Express-Liste sein



# Suche mittels Express-Listen

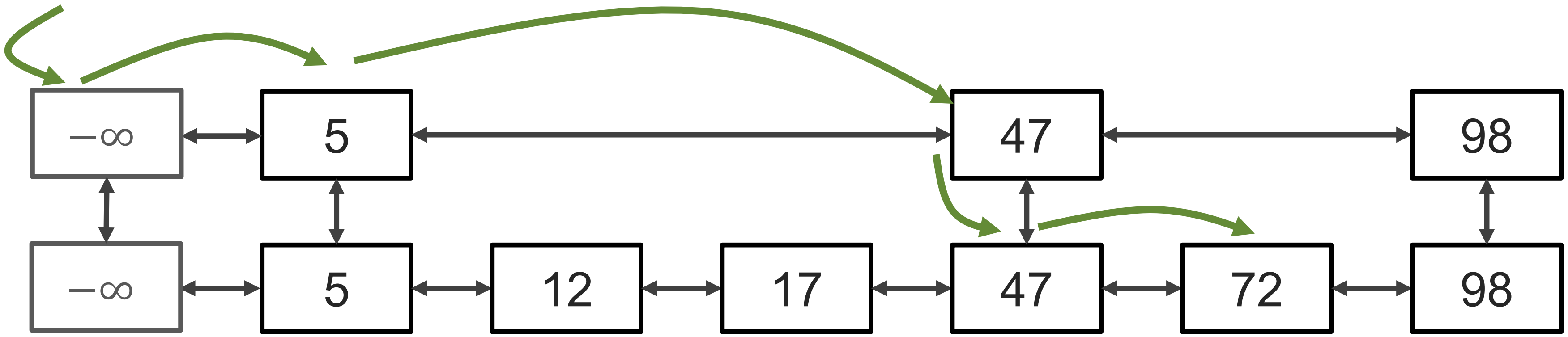
Beginne in Express-Liste:

Element gefunden? → Element ausgeben

Nächstes Element in Express-Liste kleiner-gleich gesuchtes Element?
→ Weiter nach rechts

Nächstes Element in Express-Liste größer als gesuchtes Element?
→ Nach unten in ursprüngliche Liste und dort weitersuchen

`search(L.head,72)`

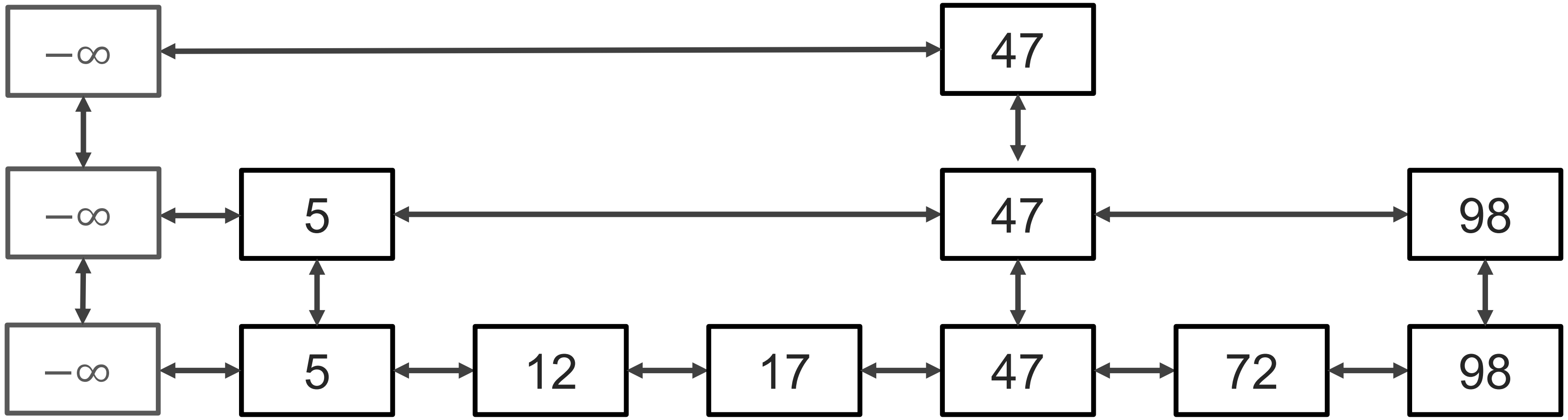


# Verbesserung

Express-Liste ist wieder Liste → Verfahre rekursiv

Beispiel:
jede Express-Liste Hälfte der Elemente in voriger Liste
ergibt $\frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1 \leq n$ zusätzliche Elemente in Express-Listen

(Betrachtung ohne Runden)



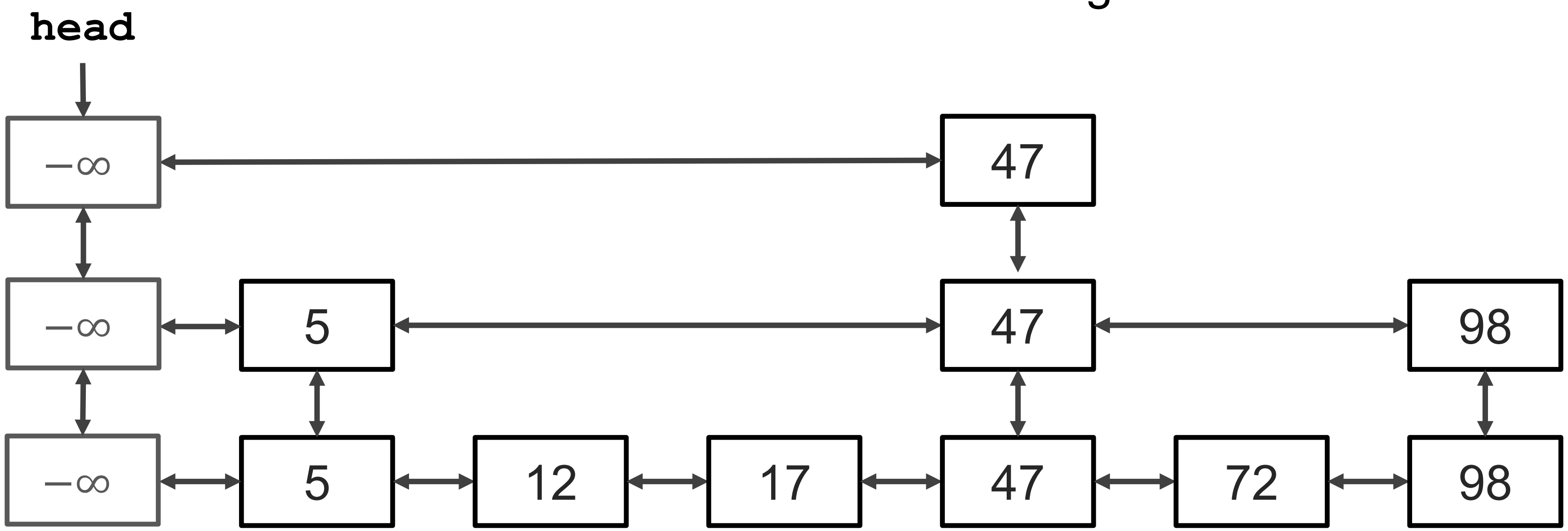



TECHNISCHE UNIVERSITÄT DARMSTADT

SYSTEMS

## Implementierung

**`L.head`** – erstes/oberstes Element der Liste
**`L.height`** – Höhe der Skiplist (beginnt mit 1)
**`x.key`** – Wert
**`x.next`** – Nachfolger
**`x.prev`** – Vorgänger
**`x.down`** – Nachfolger Liste unten
**`x.up`** – Nachfolger Liste oben
**`nil`** – kein Nachfolger / leeres Element



# Skip-Liste: Suchalgorithmus

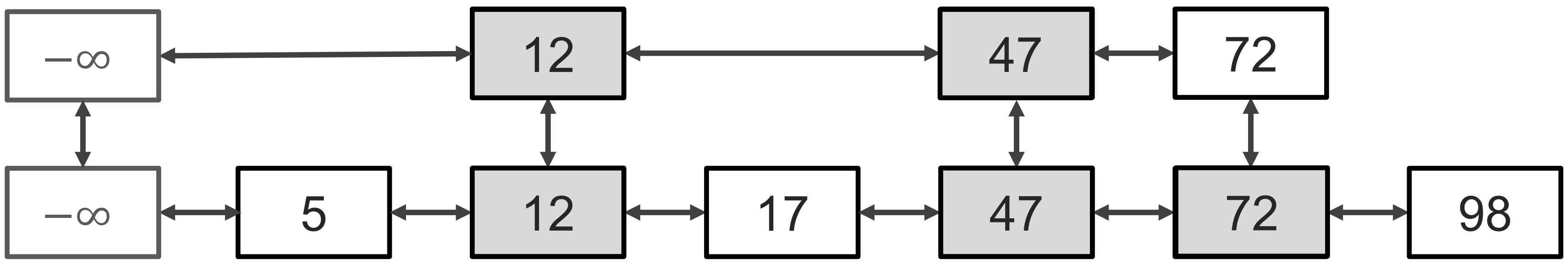
```
search(L,k)

current=L.head;
WHILE current != nil DO
   IF current.key == k THEN return current;
   IF current.next != nil AND current.next.key =< k
   THEN current=current.next
   ELSE current=current.down;
return nil;
```

Laufzeit hängt von Expresslisten ab

# Auswahl der Elemente für Express-Listen

Idee: Wähle jedes Element aus Liste
mit Wahrscheinlichkeit $p$ (z.B. $p = \frac{1}{2}$) für übergeordnete Liste

Wie viele Elemente gibt es auf den jeweiligen Ebenen?




$n$ Elemente

# Linearität des Erwartungswerts

$\mathrm{E}\,[X] = \sum_{i=1}^{n} x_i \cdot Prob[X = x_i]$

**Linearität des Erwartungswerts**

Gegeben Zufallsvariablen $X_1, \dots, X_n$ mit Erwartungswerten $E[X_i]$

Dann gilt: $\mathrm{E}\,[\sum_{i=1}^{n} a_i \cdot X_i] = \sum_{i=1}^{n} a_i \cdot E[X_i]$

Bei unseren Skip-Listen:

$X_i$ =Zufallsvariable, ob $i$-tes Element ausgewählt, $Prob[X_i = 1] = E[X_i] = p$

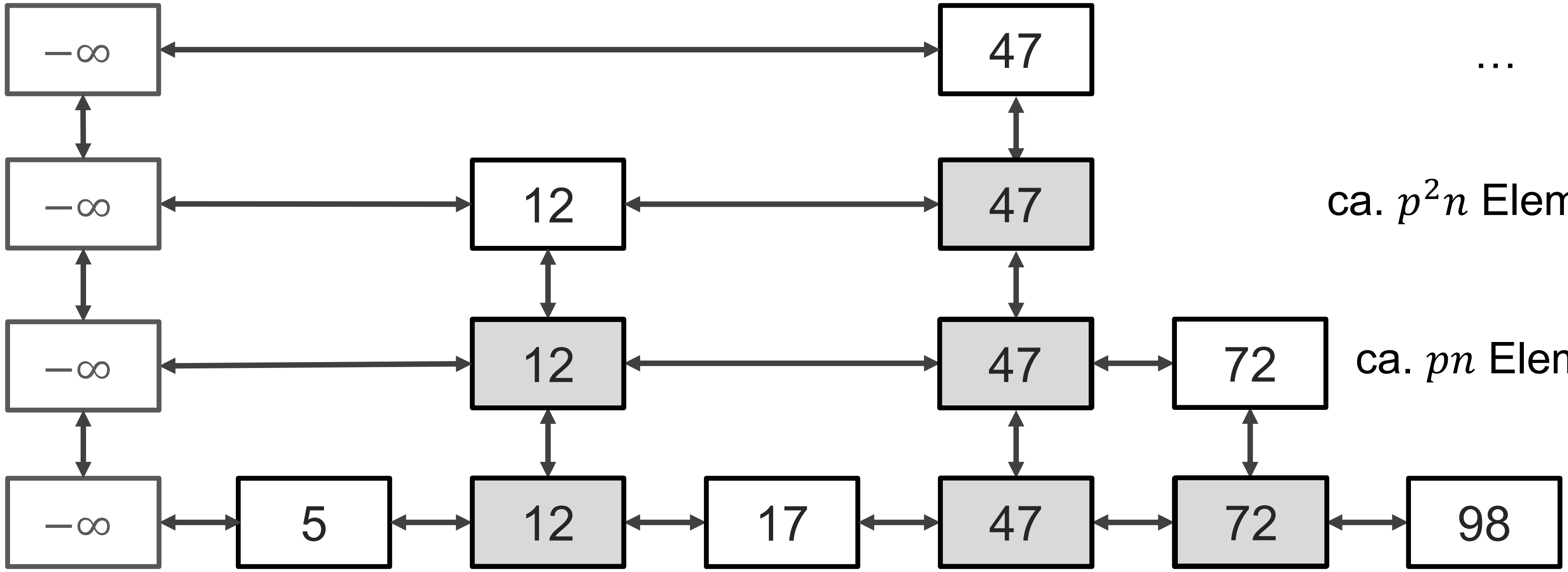
Also $\mathrm{E}\,[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = pn$ Elemente im Durchschnitt

Für nächste Ebene $Prob[X_i = 1] = E[X_i] = p^2$, also $p^2 n$ Elemente usw.

# Auswahl der Elemente für Express-Listen

Idee: Wähle jedes Element aus Liste
mit Wahrscheinlichkeit $p$ (z.B. $p = \frac{1}{2}$) für übergeordnete Liste

**Ohne Beweis: durchschnittliche Höhe $\mathbf{h = O(\log_{1/p} n)}$**




…

ca. $p^2 n$ Elemente

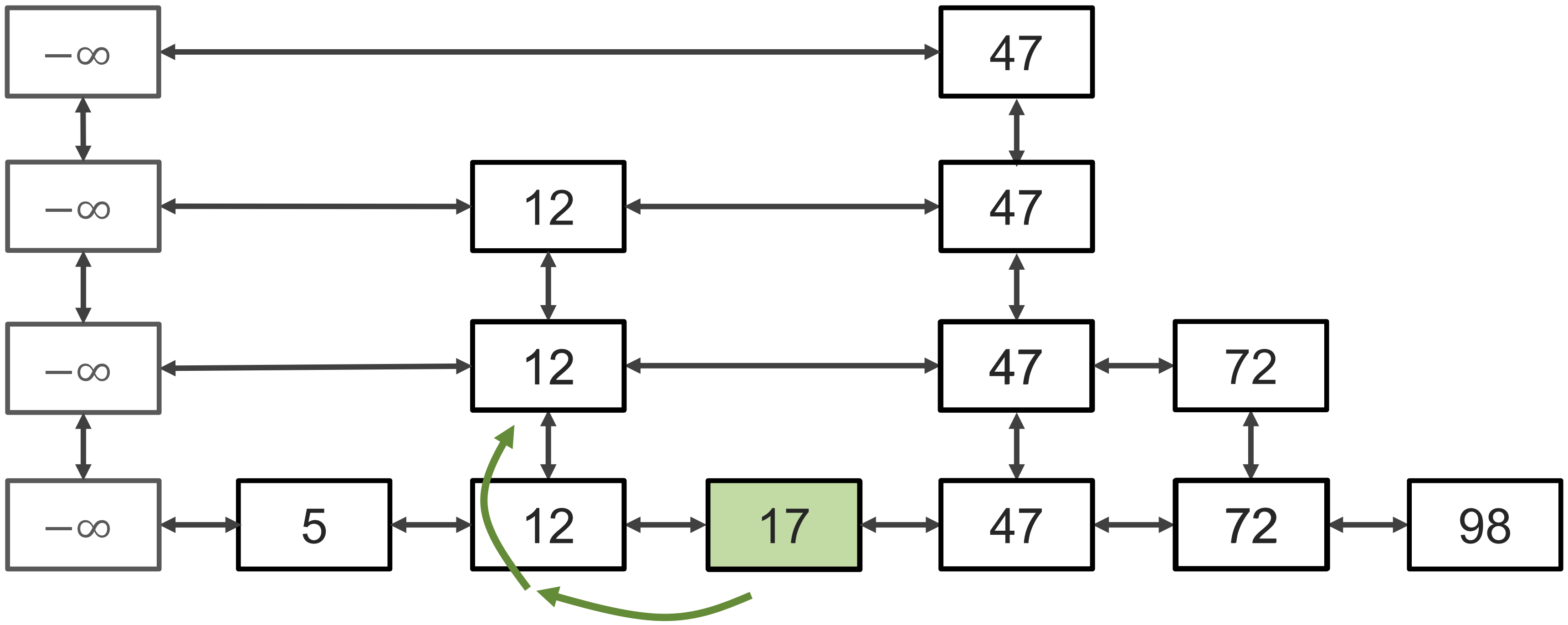
ca. $pn$ Elemente

$n$ Elemente

# Durchschnittliche Laufzeit für Suchen (I)

Im schlimmsten Fall wird Suche erst in unterster Liste beendet

Betrachte Weg gedanklich rückwärts bis zu oberster Express-Liste



# Erwartungswert geometrisch verteilter Zufallsvariablen

Wie oft muss man im Durchschnitt würfeln, bis eine 6 kommt?

**Erwartungswert geometrisch verteilter Zufallsvariablen**

Gegeben 0-1-Zufallsvariable $X$ mit $\Pr[X = 1] = p > 0$

Zufallvariable $\mathrm{Y}$ zählt, wie oft man $\mathrm{X}$ unabhängig wiederholt, bis $X = 1$

Dann gilt: $\mathrm{E}\,[Y] = \frac{1}{p}$

Also *im Durchschnitt* muss man 6-mal würfeln.

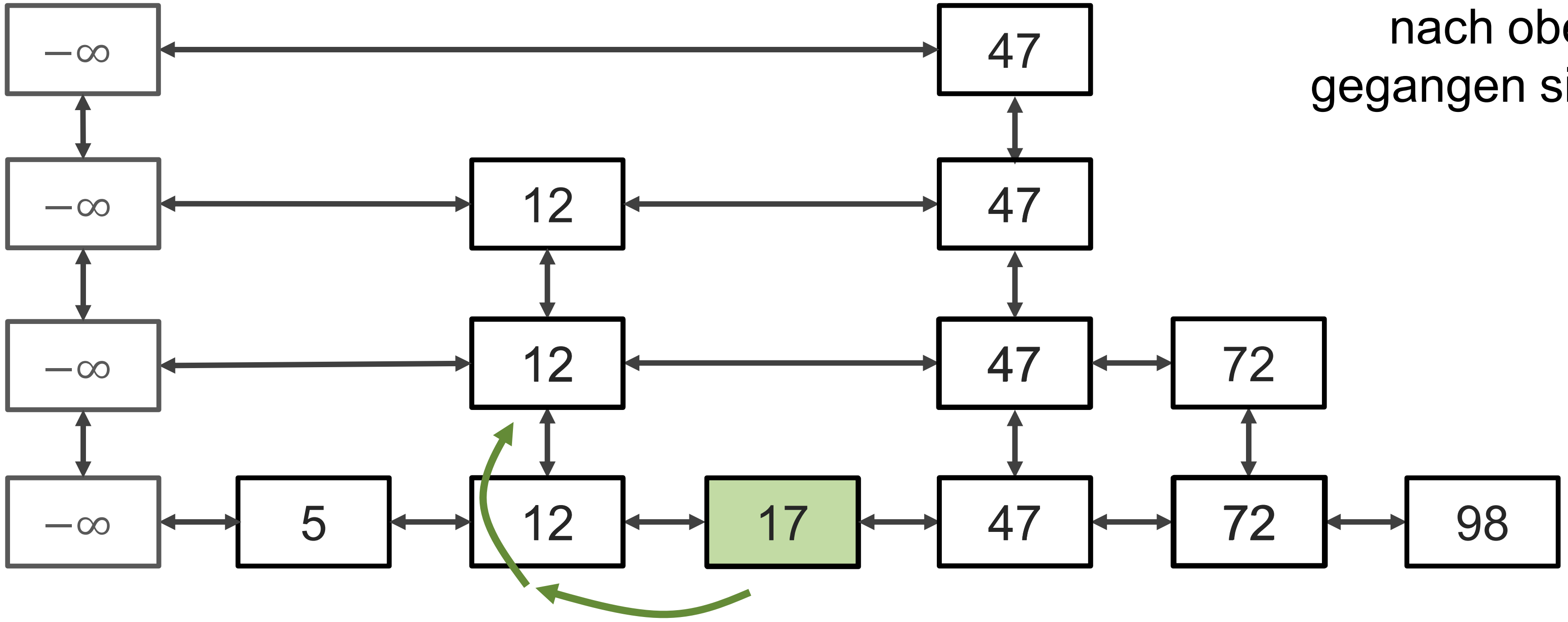
Aber: *im Worst-Case* würfelt man öfter!

# Durchschnittliche Laufzeit für Suchen (I)

Im schlimmsten Fall wird Suche erst in unterster Liste beendet

Betrachte Weg gedanklich rückwärts bis zu oberster Express-Liste

Im Durchschnitt machen wir nur $\frac{1}{p}$ Schritte auf jeder Ebene, bevor wir eine Ebene nach oben gegangen sind



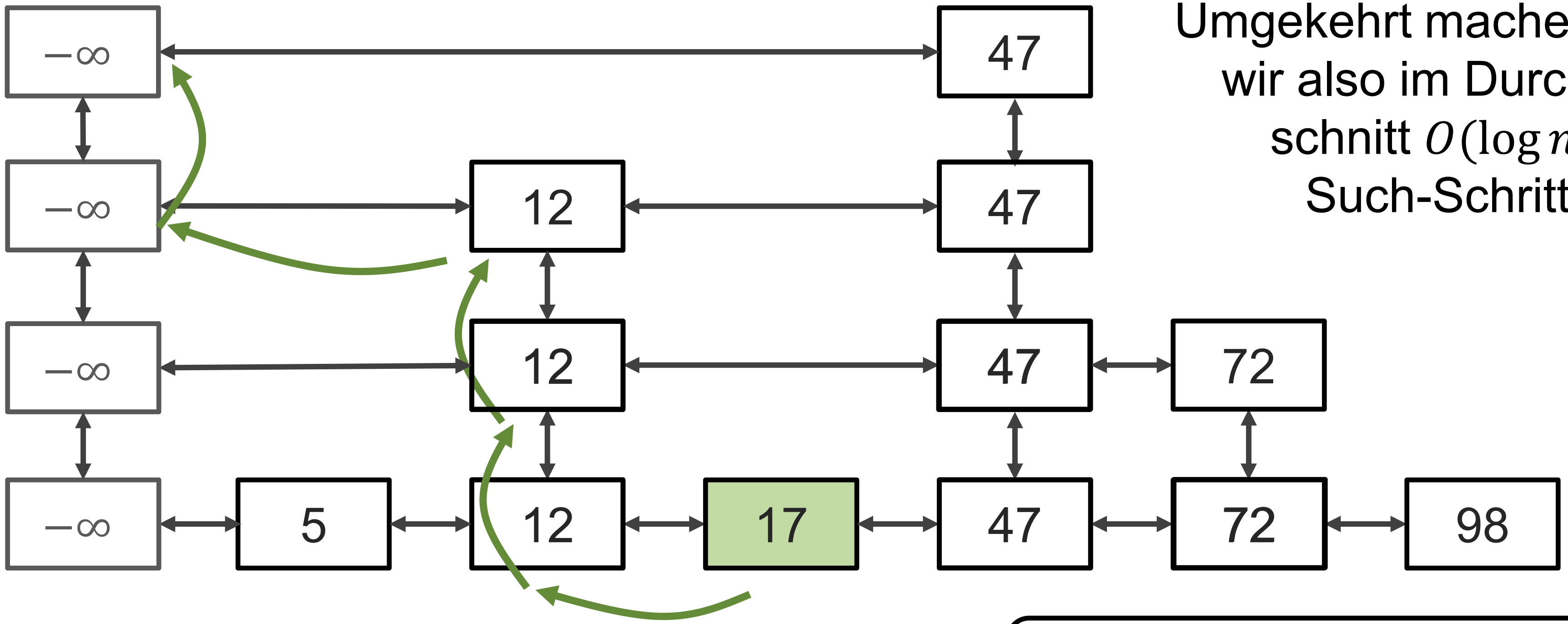
# Durchschnittliche Laufzeit für Suchen (II)

Wenn die Skip-Liste die Höhe $h$ hat, brauchen wir also im Durchschnitt

$$\underbrace{\frac{1}{p}+\frac{1}{p}+\frac{1}{p}+\cdots+\frac{1}{p}}_{h\ mal}=\frac{h}{p}=O(h)=\ O(\log n)$$

Linearität des Erwartungswerts!

viele Schritte, bis wir am Anfang sind.




Umgekehrt machen wir also im Durchschnitt $O(\log n)$ Such-Schritte.

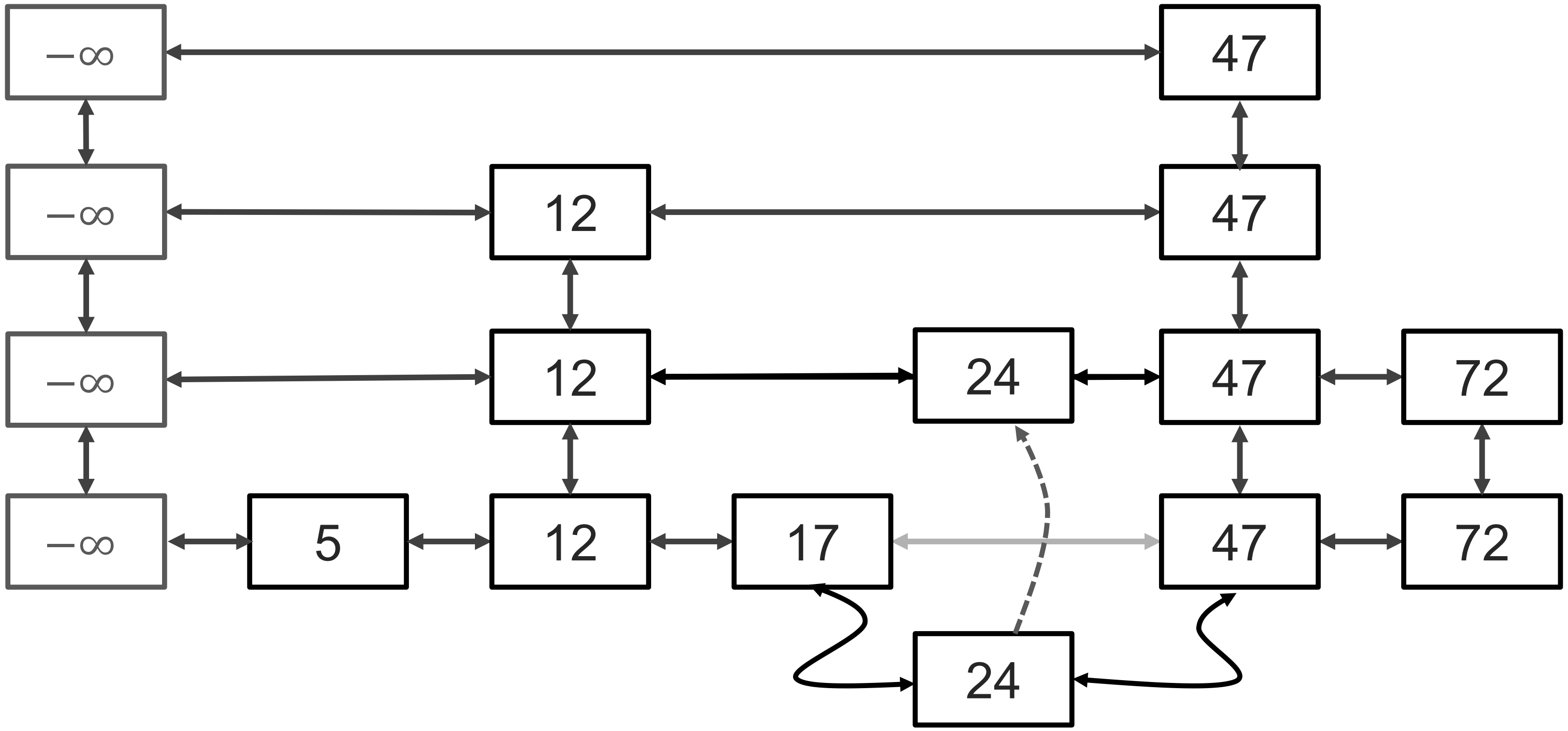
Durchschn. Laufzeit $= O(h)$



SYSTEMS

# Einfügen

Prinzip: Füge auf unterster Ebene ein und dann evtl. auf Ebenen darüber
(zufällige Wahl mit Wahrscheinlichkeit $p$ auf jeder Ebene)



Durchschn. Laufzeit $= O(h)$

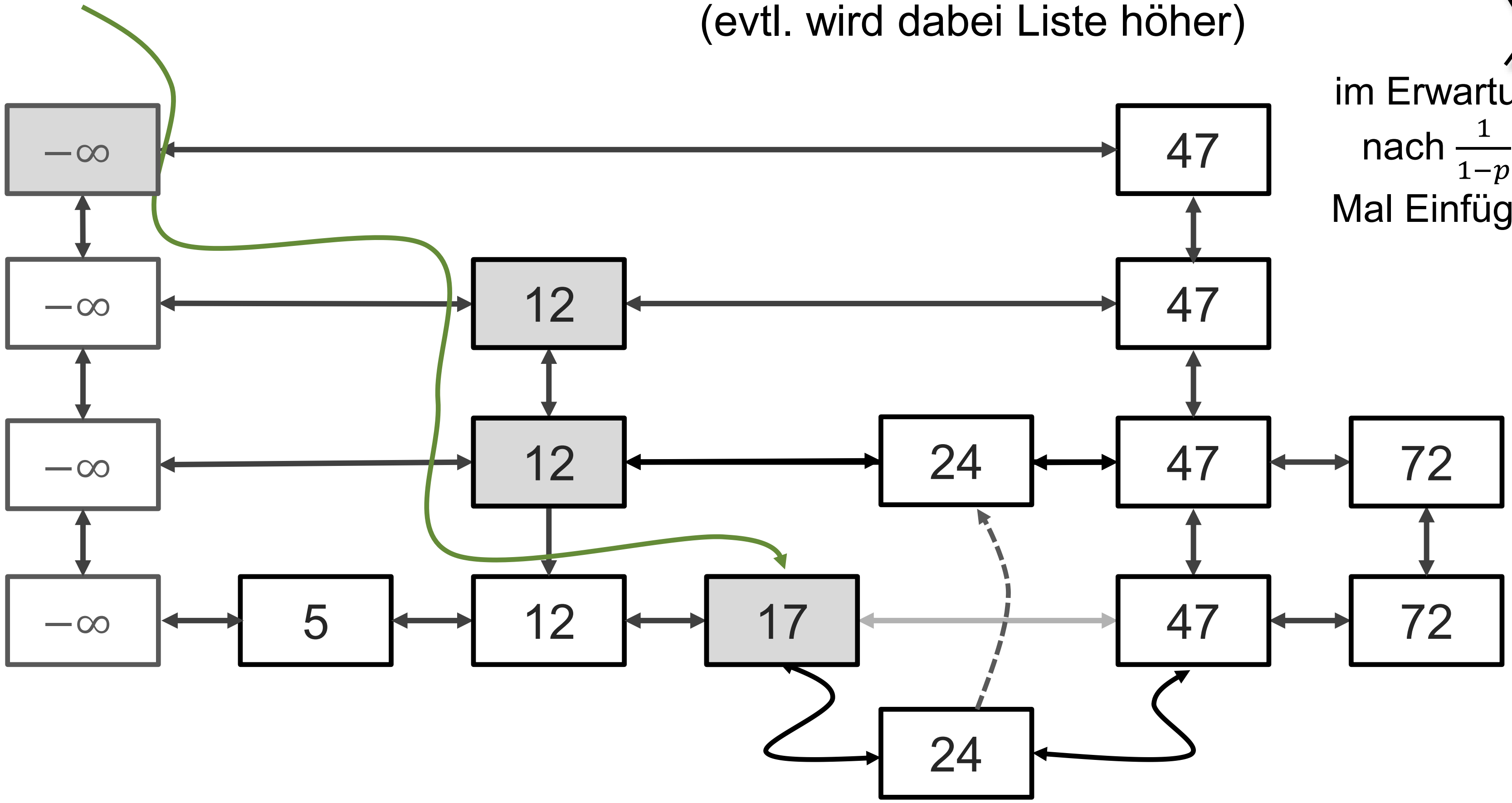
# Einfügen: Laufzeit

Speichere beim Suchen der Position jeweils Vorgängerknoten $O(h)$

Füge neues Element auf nächster Ebene mit Wskeit $p$ ein $O(1)$

(evtl. wird dabei Liste höher)

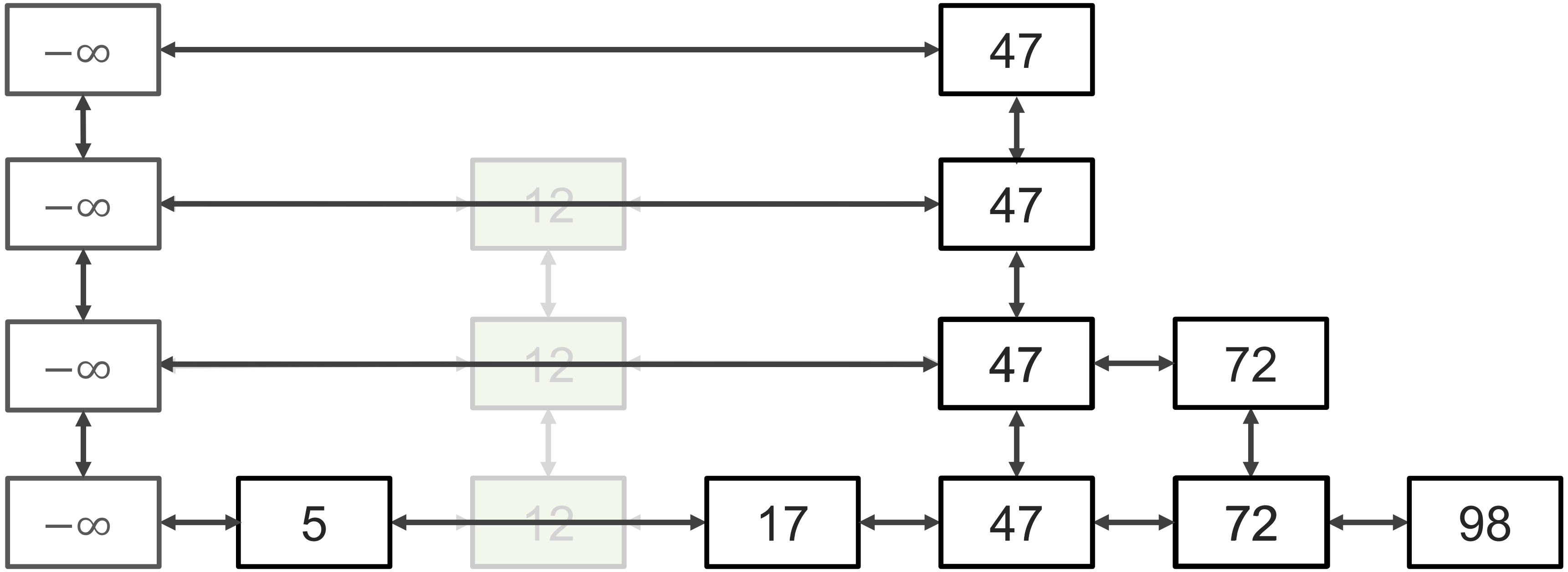
im Erwartungswert nach $\frac{1}{1-p} = O(1)$ Mal Einfügen fertig



# Löschen

Laufzeit = $O(h)$

Entferne Vorkommen des Elements auf allen Ebenen $\quad O(h)$



# Skip-Listen: Laufzeiten

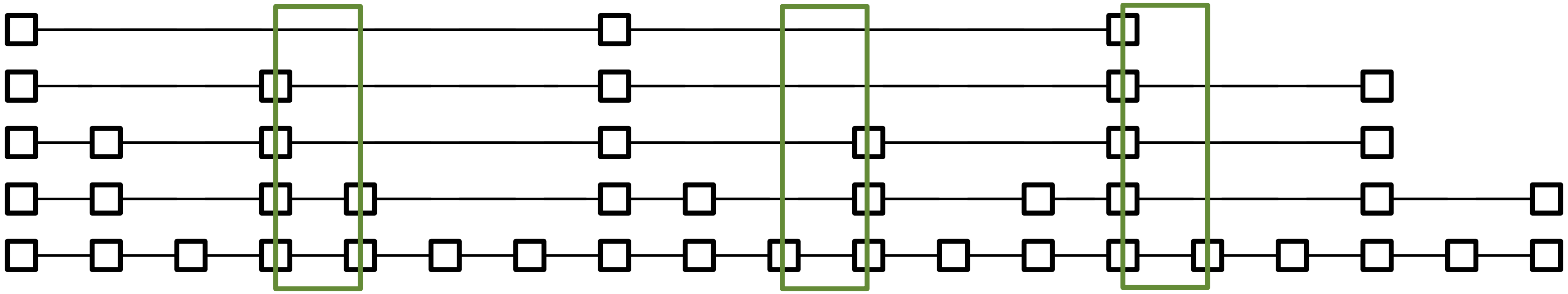
*im Durchschnitt

| Operation | Laufzeit* |
| --- | --- |
| Einfügen | $\Theta(\log_{1/p} n)$ |
| Löschen | $\Theta(\log_{1/p} n)$ |
| Suchen | $\Theta(\log_{1/p} n)$ |

$O$-Notation versteckt (konstanten) Faktor $1/p$

Speicherbedarf im Durchschnitt $= n + pn + p^2 n + \cdots = n \cdot \sum_{i \geq 0} p^i = \frac{n}{1-p}$

# Skip Lists: Anwendung



Einfügen/Löschen unterstützen
parallele Verarbeitung (z.B. Multi-Core-Systeme),
da nur sehr lokale Änderungen

vgl. zu Re-Balancierung bei Bäumen

Dafür logarithmische Laufzeit nur im Durchschnitt

# Beispiel: Discord

Quelle: **de.wikipedia.org**

Verwaltung von Member Lists per Skip Lists

Having exhausted all the obvious candidates that come with the language, a cursory search of packages was done to see if someone else had already solved and open sourced the solution to this problem. A few packages were checked, but none of them provided the properties and performance required. Thankfully, the field of Computer Science has been optimizing algorithms and data structures for storing and sorting data for the last 60 years, so there were plenty of ideas about how to proceed.

## SkipList

The ordsets perform extremely well at small sizes. Maybe there was some way that we could chain a bunch of very small ordsets together and quickly access the correct one when accessing a particular position. If you turn your head sideways and squint real hard, this starts to look like a Skip List, which is exactly what was implemented.

Discord Blog: Using RUST to Scale ELIXIR for 11 Million Concurrent Users, Mai 2019

Modifizieren Sie den Such-Algorithmus so ab, dass Sie beim Einfüge-Algorithmus ein Array mit den Vorgängerknoten auf jeder Ebene bereits haben.

Perfekte Skip-Listen wählen deterministisch bei einer Liste mit $n$ Elementen jeweils genau jedes zweite Element für die übergeordnete Expressliste. Sie haben die *Worst-Case-Laufzeit* $O(\log n)$.
Warum arbeiten wir hier mit randomisierten Skip-Listen?

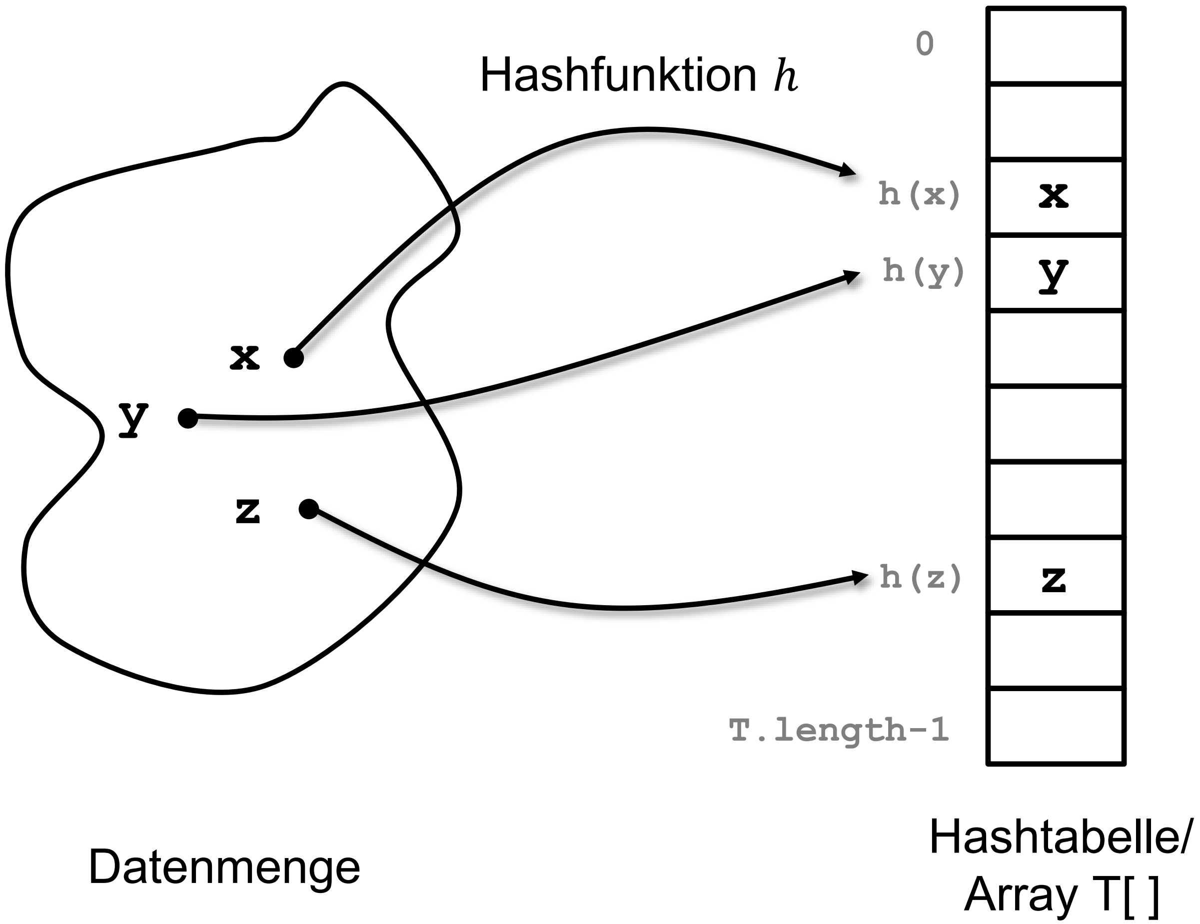
# Hash Tables

Bäume, Skip-Listen,…

| Operation | Laufzeit |
|---|---|
| Einfügen | $\Theta(\log n)$ |
| Löschen | $\Theta(\log n)$ |
| Suchen | $\Theta(\log n)$ |

**(Durchschnittliche) Laufzeit $\Theta(1)$ möglich?**

# Hash Tables: Idee

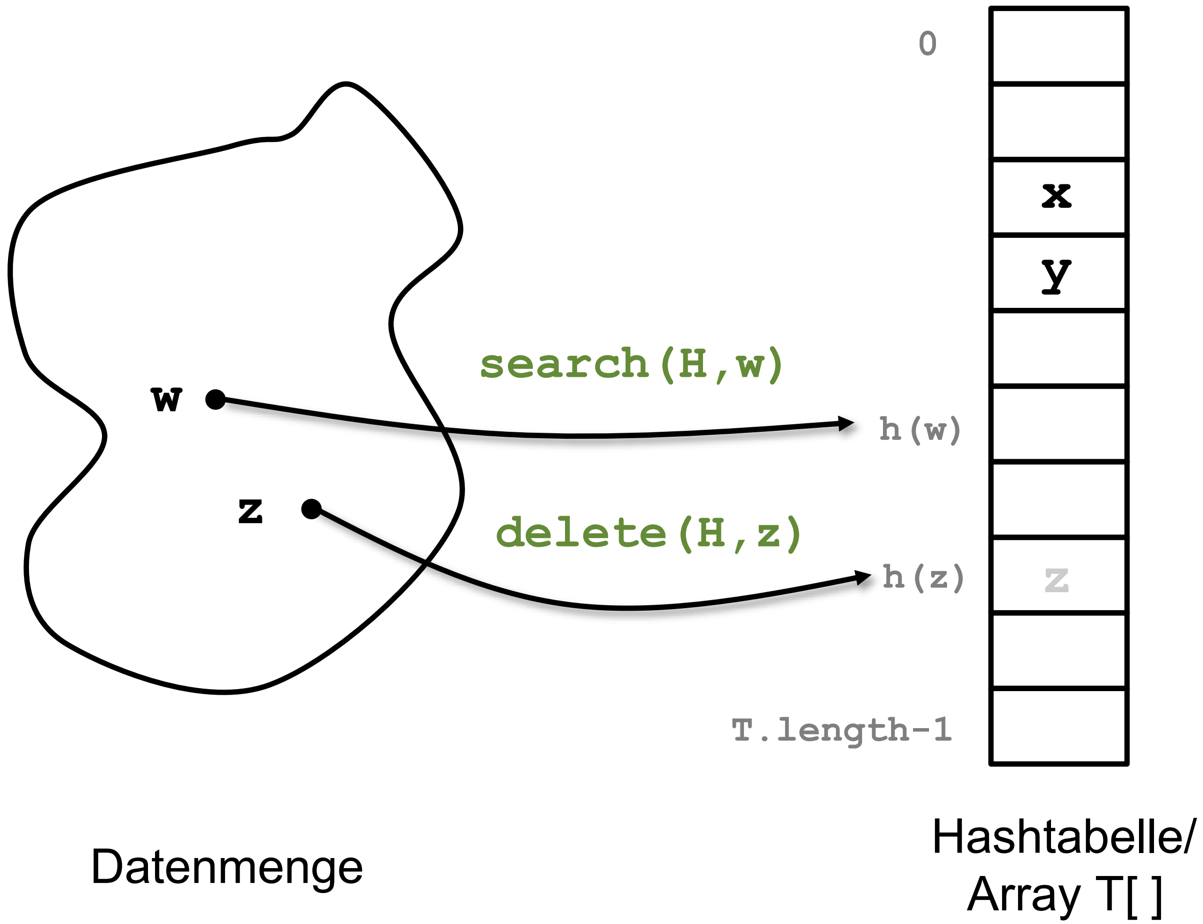



Hashfunktion sollte „gut verteilen"

Mathematisch:
**`h(x)`** ist uniform und unabhängig im Intervall $[0, \mathtt{T.length}-1]$ verteilt

Einfügen mit konstant vielen Array-Operationen

# Hash Tables: Suchen und Löschen

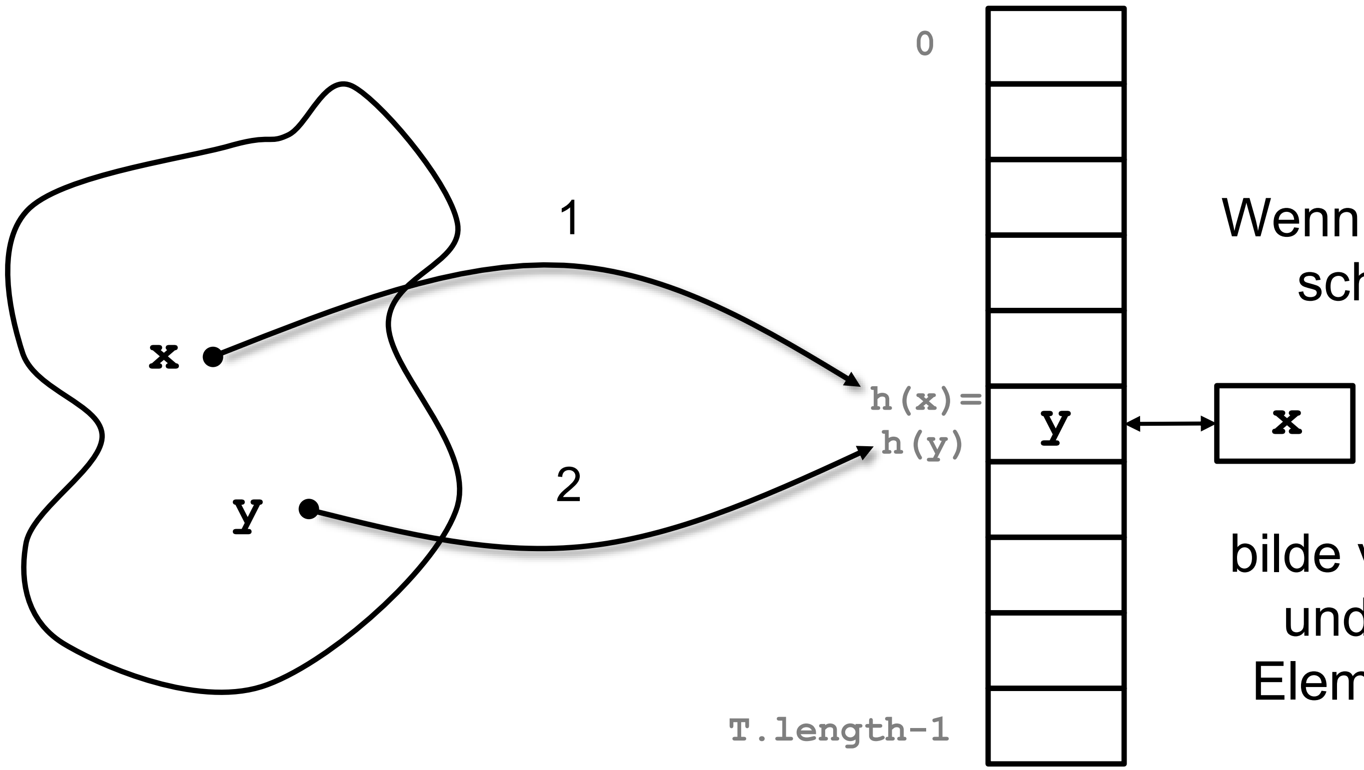



`w` in `T[h(w)]` vorhanden?

Lösche `z` aus `T[h(z)]`

Suchen/Löschen mit konstant vielen Array-Operationen

# Hash Tables: Kollisionsauflösung




Wenn Array-Eintrag schon belegt,

bilde verkettete Liste und füge neues Element vorne ein.

Datenmenge

Hashtabelle/ Array T[ ]

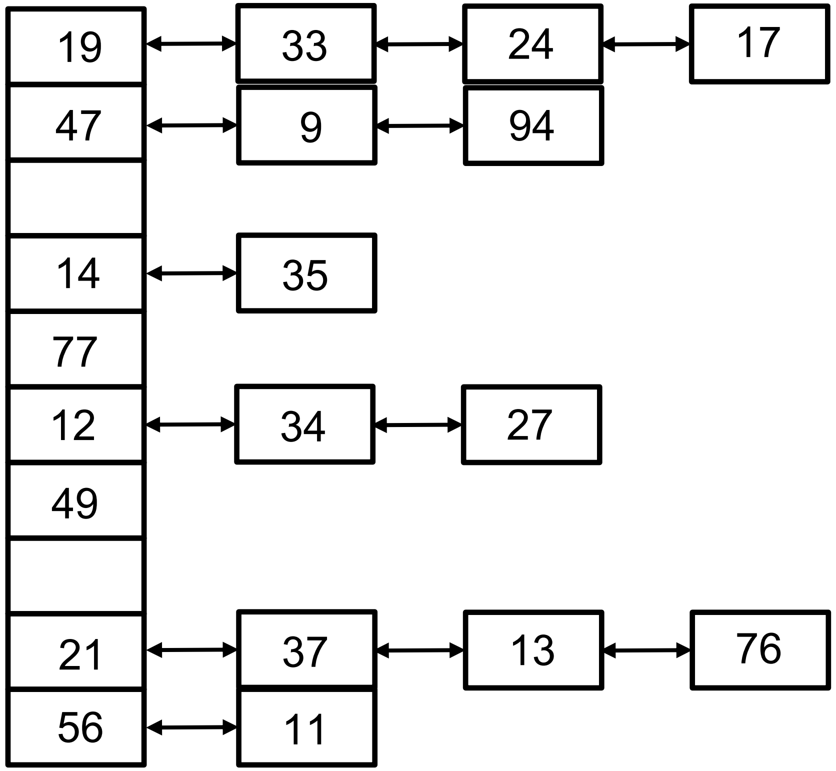
Es gibt weitere Arten der Kollisionsauflösung

# Hash Tables mit verketteten Listen

Einfügen immer noch konstante Anzahl Array-/Listen-Operation

Suchen/Löschen benötigen so viele Schritte, wie jeweilige Liste lang ist

Wenn Hashfunktion uniform verteilt, dann hat jede Liste im Erwartungswert $n/_{T.length}$ viele Einträge

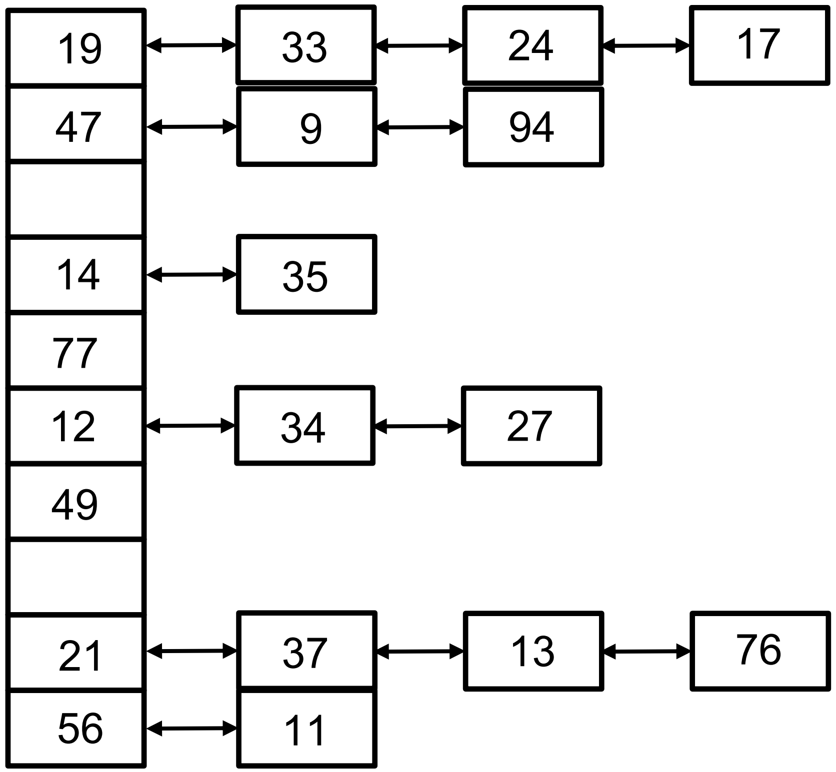



Hashtabelle/
Array T[ ]

# Hash Tables mit verketteten Listen: Laufzeit

Bei uniform und unabhängig verteilten Hashwerten benötigen Suchen und Löschen im Durchschnitt $\Theta(^n/_{T.length})$ viele Schritte.
Einfügen benötigt im Worst-Case $\Theta(1)$ viele Schritte.

ohne Analyse



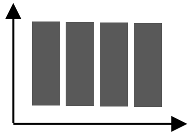

Hashtabelle/
Array T[ ]

Wählt man $T.length \approx n$ ergibt sich also konstante Laufzeit (im Durchschnitt).

# Gute Hash-Funktionen? (I)

interpretiere (Binär-)Daten als Zahlen zwischen $0$ und $p-1$,
$p\ prim, \quad p \gg T.length$

„Universelle“ Hash-Funktion:
wähle zufällige $a, b \in [0, p-1]$, $p\ prim, a \quad 0$,
setze $h_{a,b}(x) = ((a \cdot x + b)\ mod\ p)\ mod\ T.length$

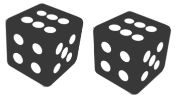


„Verteilung“:

Für alle $x, t \in [0, p-1]$ gilt:

$$Pr_h[(a \cdot x + b)\ mod\ p = t\ ] = \frac{1}{p}$$

Zu gegebenen $t, x, a$
gibt es genau ein
$b = (ax - t)\ mod\ p$ mit
$h_{a,b}(x) = (ax + b)\ mod\ p = t$



„Unabhängigkeit“/
„Kollisionsresistenz“:

Für alle $x \quad y, t_x, t_y \in [0, p-1]$ gilt:

$$Pr_h\begin{bmatrix}(ax+b)\ mod\ p = t_x, \\ (ay+b)\ mod\ p = t_y\end{bmatrix} = \frac{1}{p^2}$$

(ohne Beweisidee)

# Gute Hash-Funktionen? (II)

Kryptographische Hash-Funktionen wie MD5, SHA-1: $\{0,1\}^* \rightarrow \{0,1\}^{160}$

Achtung: Diese Funktionen werden
aus Sicherheitsgründen
in der Kryptographie
bei „böswillig“ gewählten Daten
nicht mehr eingesetzt

besser: SHA-2 oder SHA-3

Setze $h(x) = MD5(x)\ mod\ T.length$

# Hash Tables: Anwendungen

Quelle: **de.wikipedia.org**

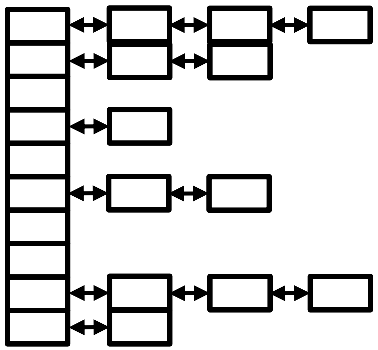
## 10.3.1 How MySQL Uses Indexes

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

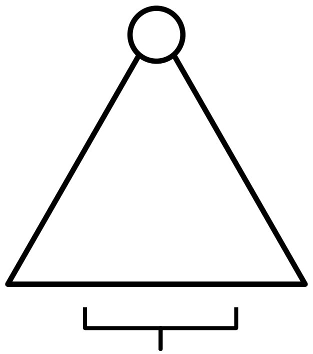
Most MySQL indexes (`PRIMARY KEY`, `UNIQUE`, `INDEX`, and `FULLTEXT`) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; `MEMORY` tables also support hash indexes; `InnoDB` uses inverted lists for `FULLTEXT` indexes.

Reference Manual

# Hash Tables vs. Bäume



nur Suche nach
bestimmten Wert möglich



schnelles Traversieren zu „Nachbarn“
möglich (z.B. nächstkleinerer Wert),
auch Bereichssuche

In der Regel Hashtable größer als zu erwartende Anzahl Einträge:

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs.
cost (reflected in most of the operations of the HashMap class, including get and put). The expected
account when setting its initial capacity, so as to minimize the number of rehash operations. If the
by the load factor, no rehash operations will ever occur.

Java Class HashMap

# Hash Tables: Laufzeiten

*im Durchschnitt

| Operation | Laufzeit* |
|---|---|
| Einfügen | $\boldsymbol{\Theta}(1)$** |
| Löschen | $\boldsymbol{\Theta}(1)$ |
| Suchen | $\boldsymbol{\Theta}(1)$ |

**sogar Worst-Case

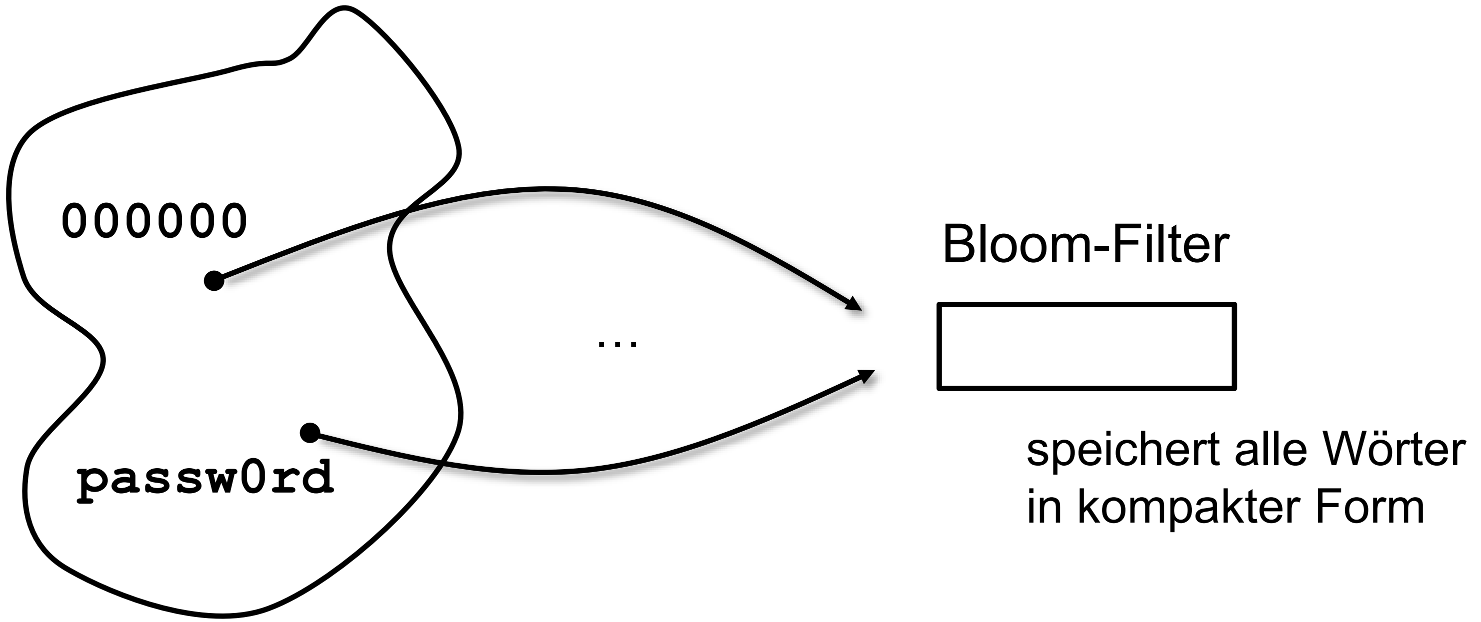
Speicherbedarf in der Regel größer als $n$, üblicherweise ca. $1{,}33 \cdot n$

# Bloom-Filter

„Speicherschonende Wörterbücher mit kleinem Fehler“

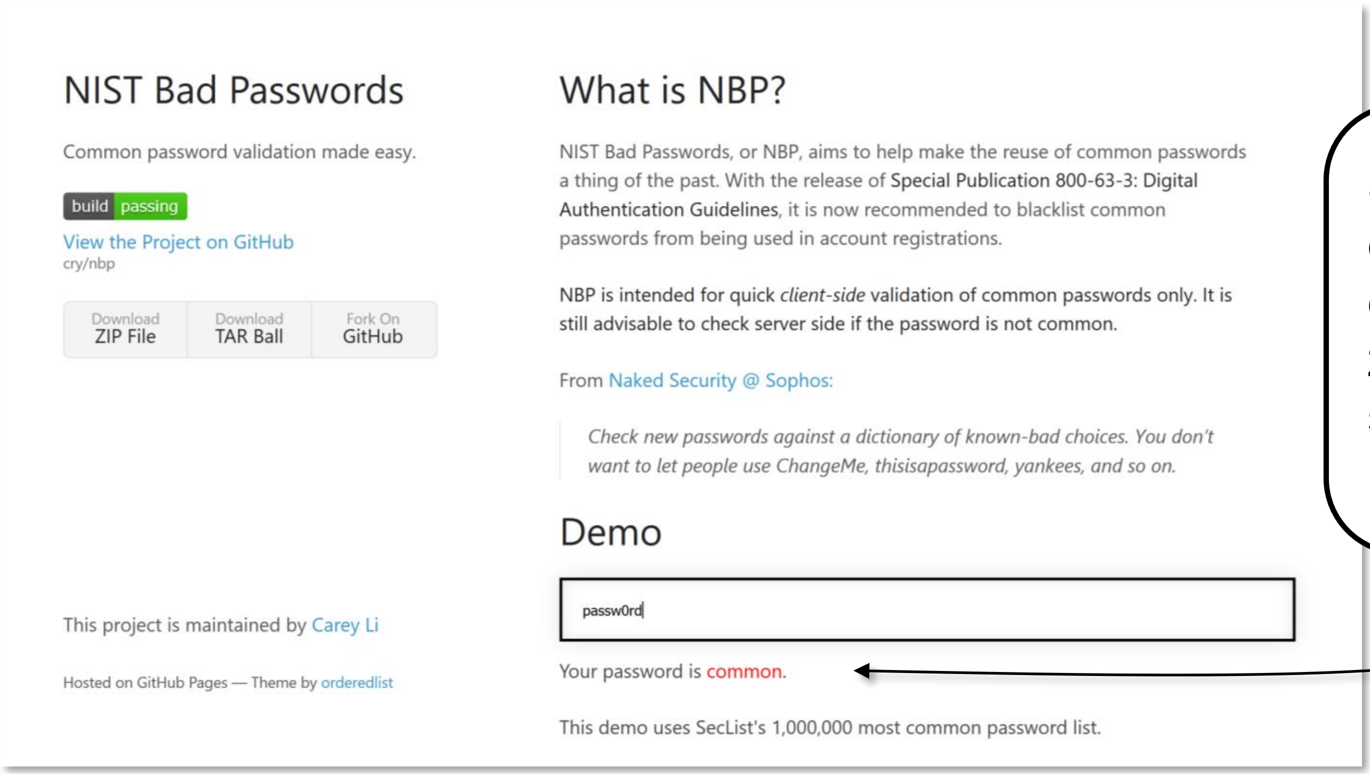
# Beispiel: Schlechte Passwörter vermeiden (I)

1. Speichere („offline“) alle schlechten Passwörter im Bloom-Filter



# Beispiel: Schlechte Passwörter vermeiden (II)

2. Prüfe („online“), ob eingegebenes Passwort im Bloom-Filter



## NIST Bad Passwords

Common password validation made easy.

build passing

View the Project on GitHub
cry/nbp

Download ZIP File | Download TAR Ball | Fork On GitHub

This project is maintained by Carey Li

Hosted on GitHub Pages — Theme by orderedlist

## What is NBP?

NIST Bad Passwords, or NBP, aims to help make the reuse of common passwords a thing of the past. With the release of Special Publication 800-63-3: Digital Authentication Guidelines, it is now recommended to blacklist common passwords from being used in account registrations.

NBP is intended for quick *client-side* validation of common passwords only. It is still advisable to check server side if the password is not common.

From Naked Security @ Sophos:

> *Check new passwords against a dictionary of known-bad choices. You don't want to let people use ChangeMe, thisisapassword, yankees, and so on.*

## Demo

passw0rd

Your password is common.

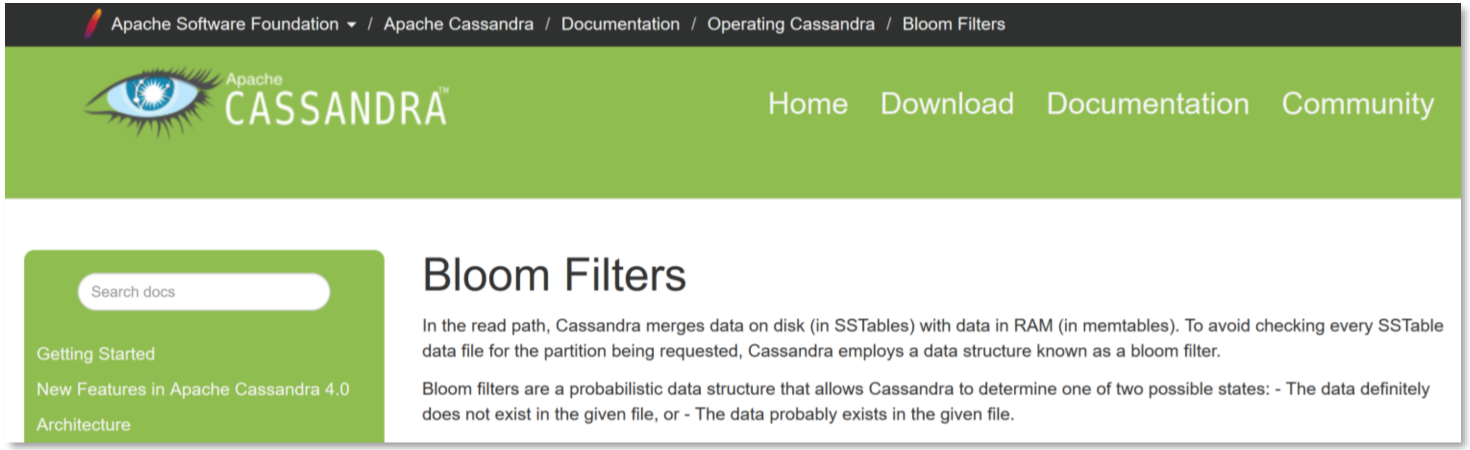
This demo uses SecList's 1,000,000 most common password list.

Starke Passwörter, die fälschlicherweise dem Wörterbuch zugeordnet werden, sind ärgerlich, aber nicht sehr schlimm

Quelle: https://cry.github.io/nbp/

# Anwendungen Bloom-Filter

Quelle: `http://cassandra.apache.org`




NoSQL-Datenbanken: Abfragen für nicht-vorhandene Elemente verhindern

Bitcoin: Prüfen von Transaktionen ohne gesamte Daten zu laden

Früher auch Chrome-Browser: Erkennen schädlicher Webseiten

# Bloom-Filter: Erstellen (I)

Gegeben: $n$ Elemente $x_0, \dots, x_{n-1}$ beliebiger Komplexität

$m$ Bits Speicher, üblicherweise in einem Bit-Array

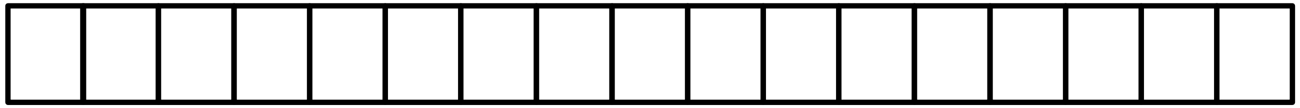
$k$ „gute“ Hash-Funktionen $H_0, \dots, H_{k-1}$ mit Bildbereich $0,1,\dots,m-1$

empfohlene Wahl: $k = \frac{m}{n} \cdot \ln 2$

ergibt Fehlerrate von ca. $2^{-k}$

üblicherweise $k = 5,6,\dots,20$

BF[ ]=

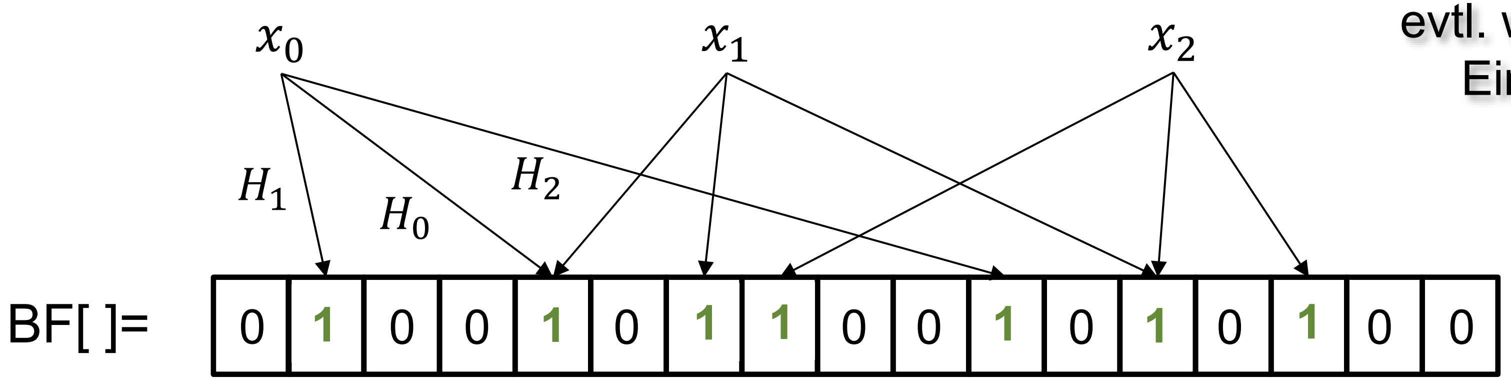


# Bloom-Filter: Erstellen (II)

```
initBloom(X,BF,H) //H array of functions H[j]
    FOR i=0 TO BF.length-1 DO BF[i]=0;
    FOR i=0 TO X.length-1 DO
        FOR j=0 TO H.length-1 DO
            BF[H[j](X[i])]=1;
```

1. Initialisiere Array mit 0-Einträgen

2. Schreibe für jedes Element in jede Bit-Position $H_0(x_i)$, ...,$H_{k-1}(x_i)$ eine 1




evtl. werden dabei Einträge erneut auf 1 gesetzt

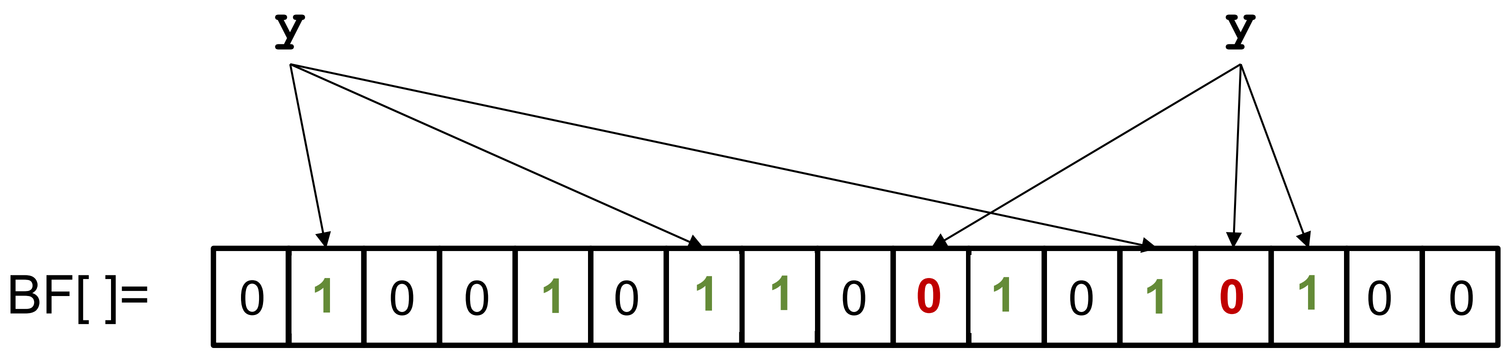
# Bloom-Filter: Suchen

```
searchBloom(BF,H,y) //H array of functions H[j]
    result=1;
    FOR j=0 TO H.length-1 DO
        result=result AND BF[H[j](y)];
    return result;
```

Gib an, dass **`y`** im Wörterbuch, wenn genau alle $k$ Einträge für **`y`** in **`BF=1`** sind

in Wörterbuch:

nicht in Wörterbuch:



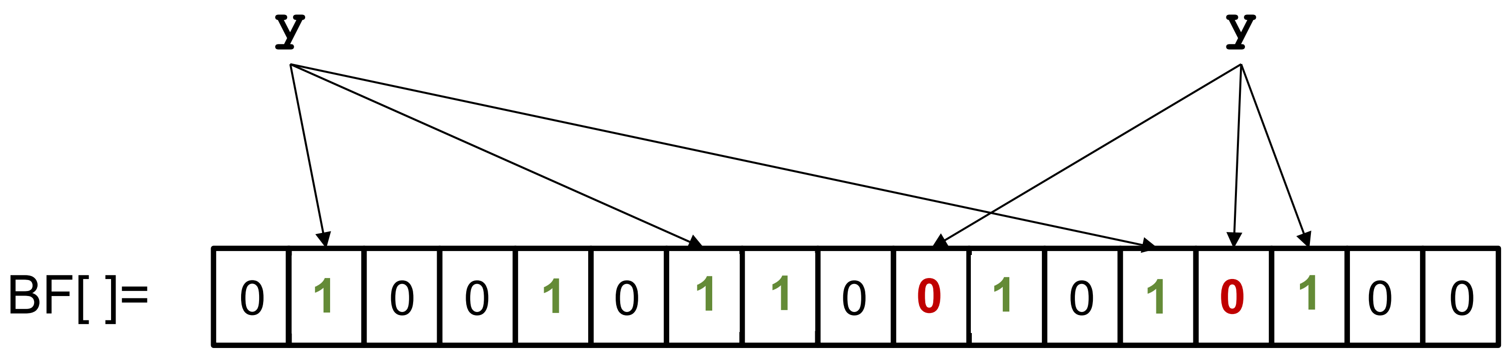
# Bloom-Filter: Korrektheit (I)

keine „false negatives“

Wenn `y` im Wörterbuch, also `y=X[i]` für ein `i`, dann wurden alle Einträge `H[j](X[i])` in `BF` zuvor gesetzt, also gibt Algorithmus auch `1` zurück.

in Wörterbuch:

nicht in Wörterbuch:



# Bloom-Filter: Korrektheit (II)

„false positives“

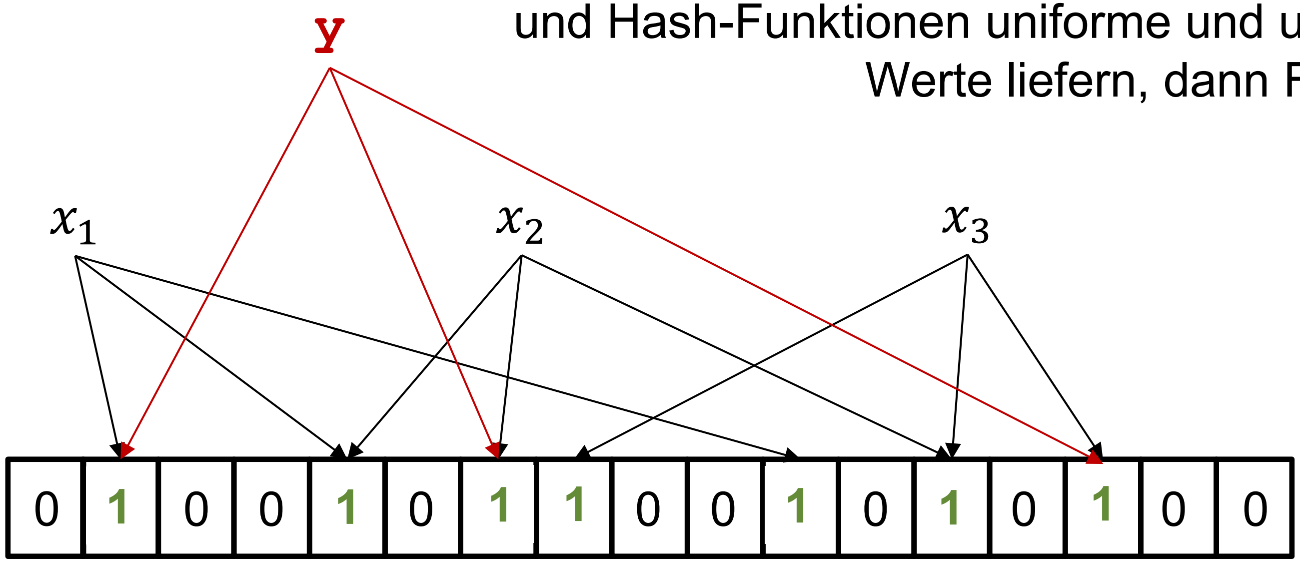
Wenn **y** **nicht** im Wörterbuch, dann gibt Algorithmus evtl. trotzdem **1** zurück.

Passiert, wenn Einträge für von anderen Werten getroffen wurden.

Daher „gute“ Hash-Funktionen und Größe des Filters nicht zu klein.

Wenn BF nur bis zur Hälfte mit 1en gefüllt und Hash-Funktionen uniforme und unabhängige Werte liefern, dann Fehler $\leq 2^{-k}$

BF[ ]=



# Bloom-Filter: Beispielrechnung

$n = 100.000$ Passwörter, je 10 ASCII-Zeichen

## Baumstruktur

Speicherbedarf:
8.000.000 Bits
(+Baumstruktur)

Suchen:
ca. $\log_2 100.000 \approx 17$
Elemente betrachten

## Bloom-Filter

$k = 7,\ m = k \cdot n \cdot \ln 2$

Speicherbedarf:
ca. 1.000.000 Bits

Suchen:
$k = 7$ Mal Hashen und
$k = 7$ Array-Zugriffe

Gegeben je einen Bloom-Filter gleicher Größe $m$ für Datenmengen D1 und D2, wie berechnen Sie einen Bloom-Filter für die Vereinigung D1∪D2?

Können Sie in einen Bloom-Filter problemlos ein Element $x$ wieder löschen, indem Sie alle Einträge $H_j(x)$ im Filter auf 0 setzen?