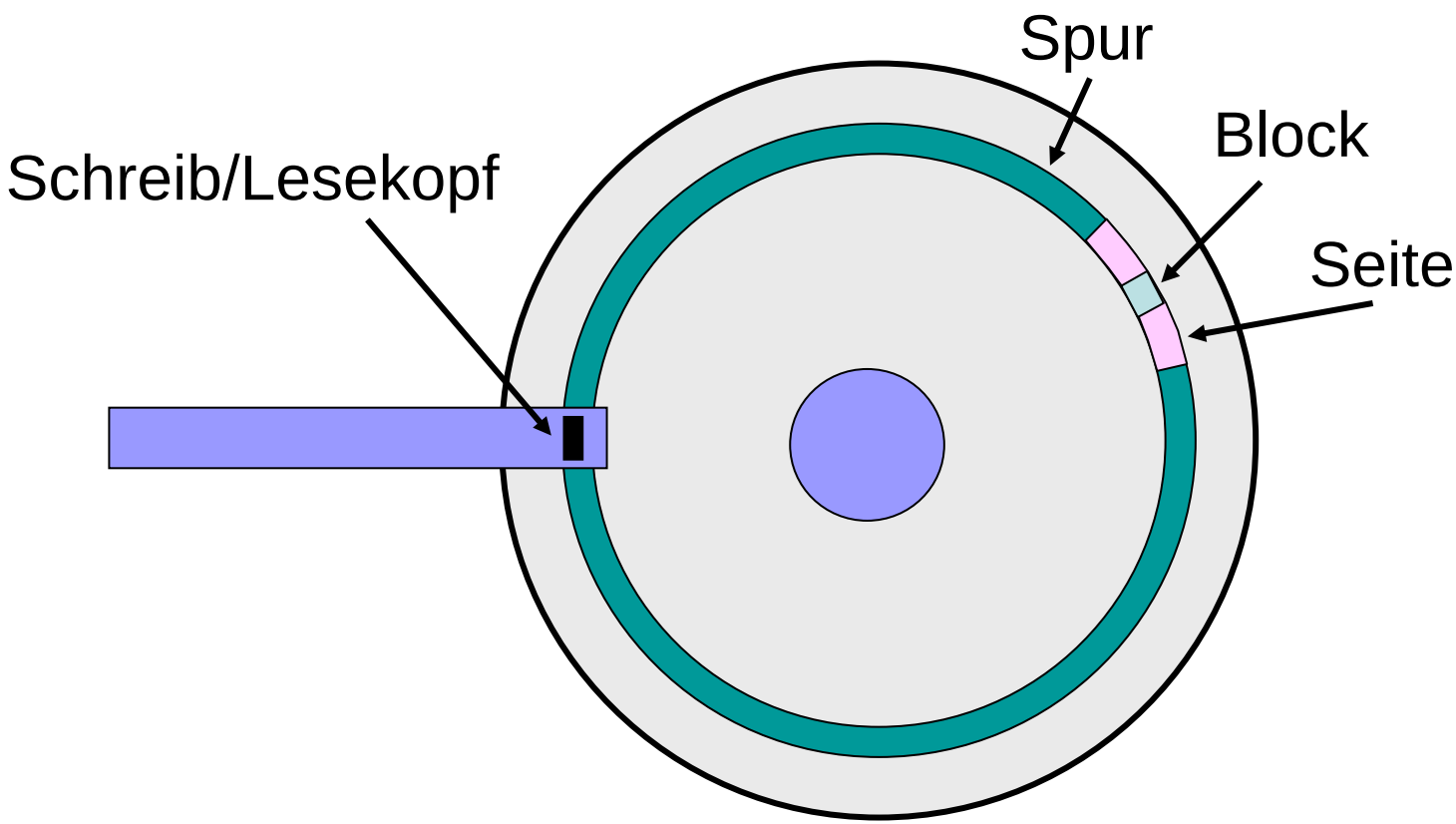
Kapitel 4: Physikalische Datenorganisation

Speicherhierarchie

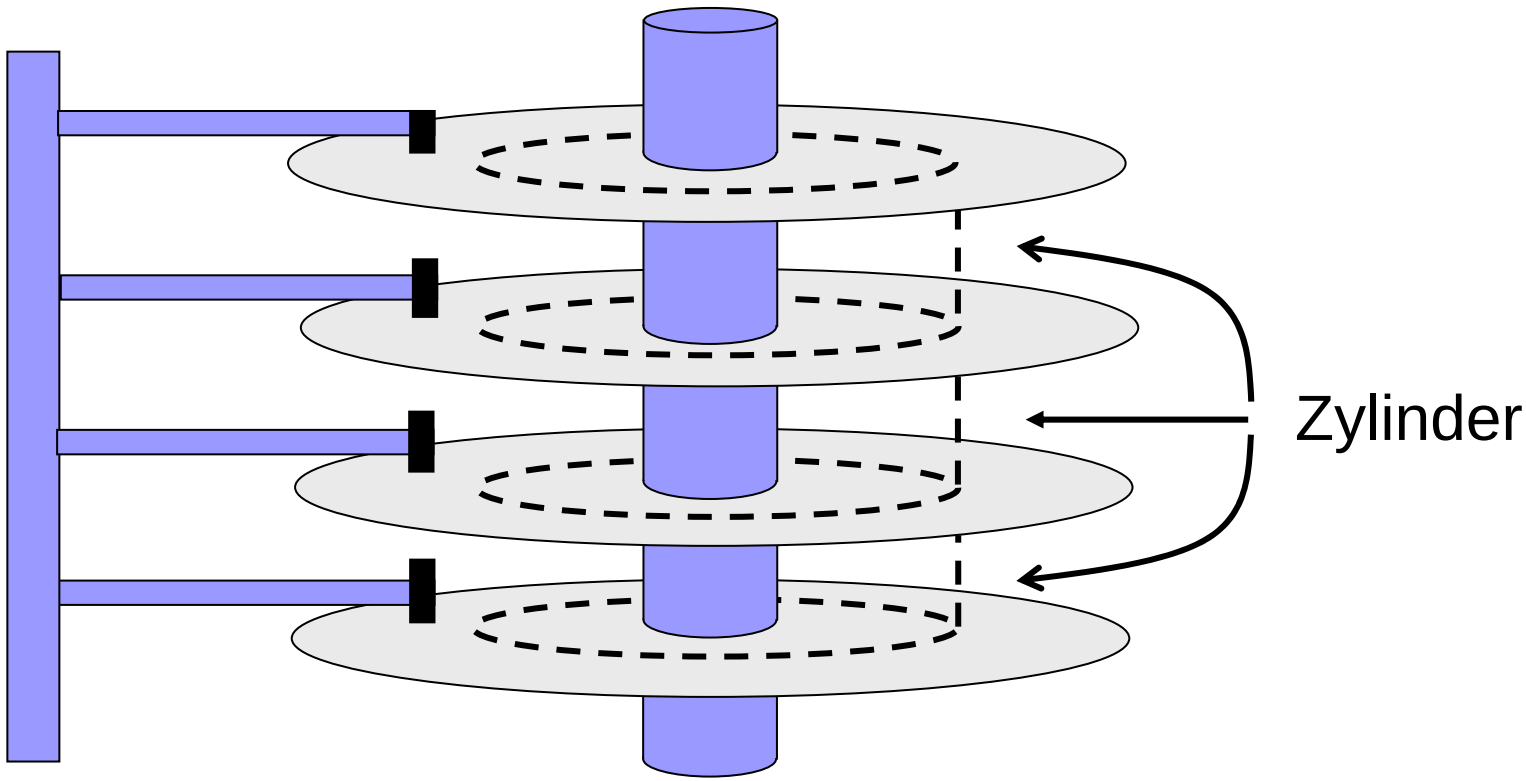
	Primär	Sekundär	Tertiär
Größe	klein	groß [10^3]	sehr groß
Tempo	schnell	langsam [10^{-5}]	sehr langsam
Preis	teuer	billig [10^{-2}]	billig
Granularität	fein	grob	grob
Stabilität	flüchtig	stabil	stabil

Festplatte: von oben



Zugriff = Positionieren + Warten + Lesen

Festplatte: seitlich



Physikalische Datenorganisation

Record: Datensatz fester oder variabler Länge
mit Feldern bestimmten Typs

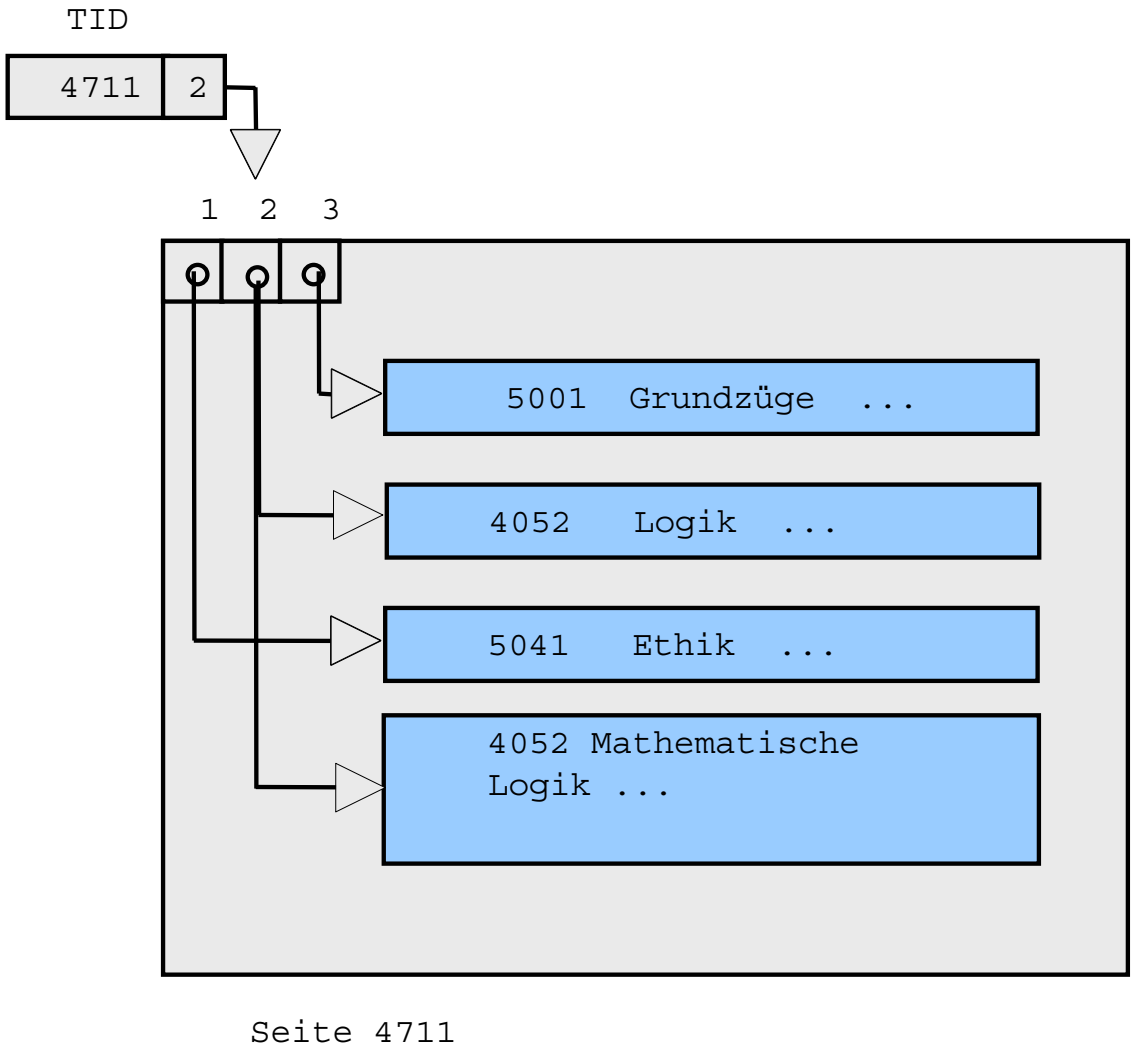
Block: Speichereinheit imHintergrundspeicher
(2^9 - 2^{12} Bytes)

File: Menge von Blöcken

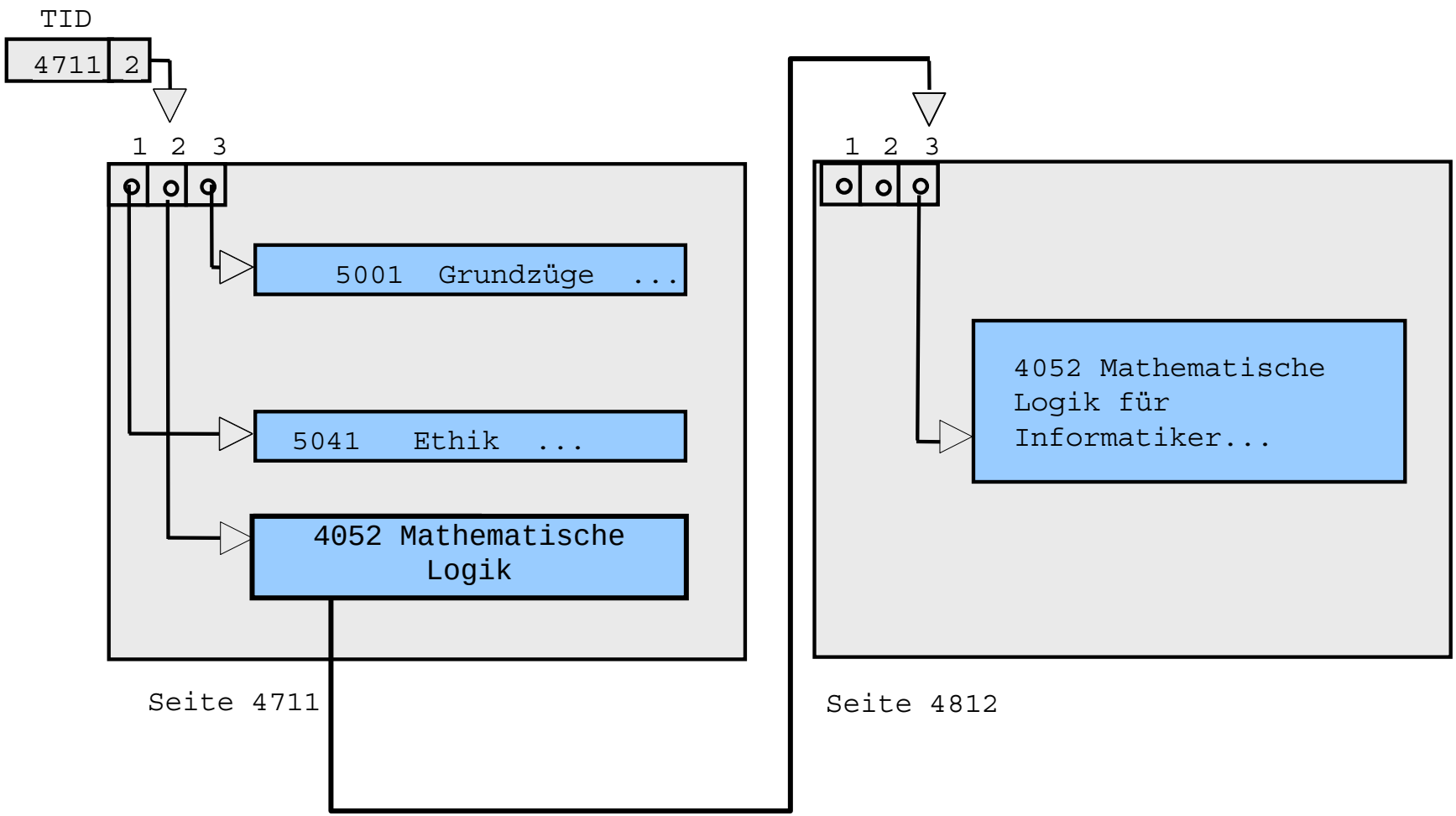
Pinned record: Blockadresse + Offset

Unpinned record: Blockadresse + Recordschlüssel
Blockadresse + Tupelidentifikator

Tupelidentifikator: Verschieben innerhalb der Seite



Tupelidentifikator: Verdrängen auf andere Seite



Implementierung des E-R-Modells

- pro Entity ein Record mit den Attributen als Datenfelder
- pro Relationship ein Record mit den TIDs der beteiligten Entities

Speicher-Operationen

- INSERT: Einfügen eines Records
- LOOKUP: Suchen eines Records
- MODIFY: Modifizieren eines Records
- DELETE: Löschen eines Records

Heap-File

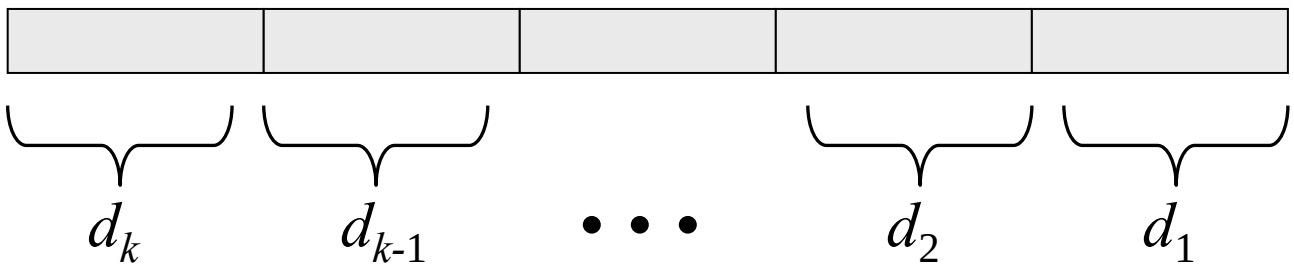
- INSERT: Record am Ende einfügen
- LOOKUP: Gesamtes File durchsuchen
- MODIFY: Record überschreiben
- DELETE: Lösch-Bit setzen

Hashing

- alle Records sind auf Buckets verteilt
- ein Bucket = verzeigerte Liste von Blöcken
- Bucketdirectory enthält Einstiegsadressen
- Hashfunktion liefert zuständiges Bucket
- Wertebereich: **`[0 .. N-1]`**
- Pro Datenrecord ein Frei/Belegt-Bit

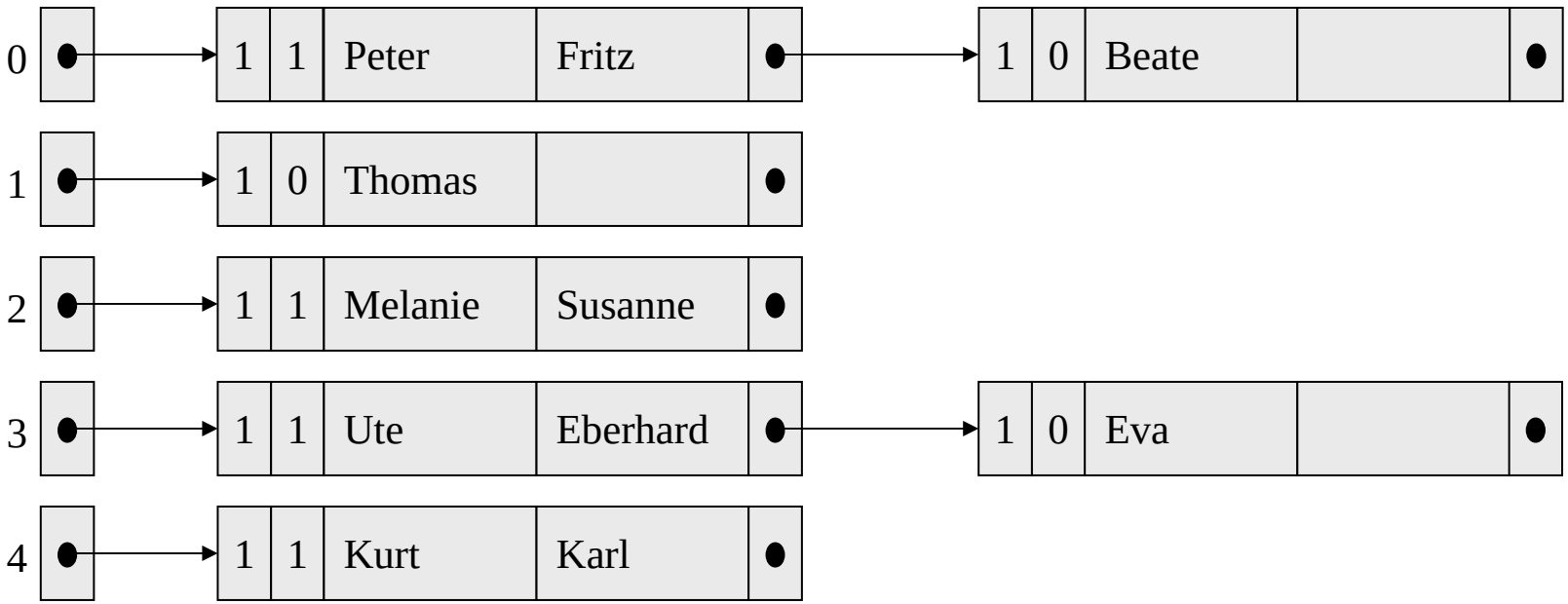
Beispiel für Hash-Funktion

Zerlege Schlüssel *v* in *k* Gruppen zu je *n* Bits.
Fasse jede Gruppe als Zahl auf.



$$h(v) = (d_k + d_{k\text{-}1} + \ldots + d_2 + d_1) \bmod N$$

Beispiel für Hashorganisation (|v| mod 5)



Hash-Operationen für Schlüssel *v*

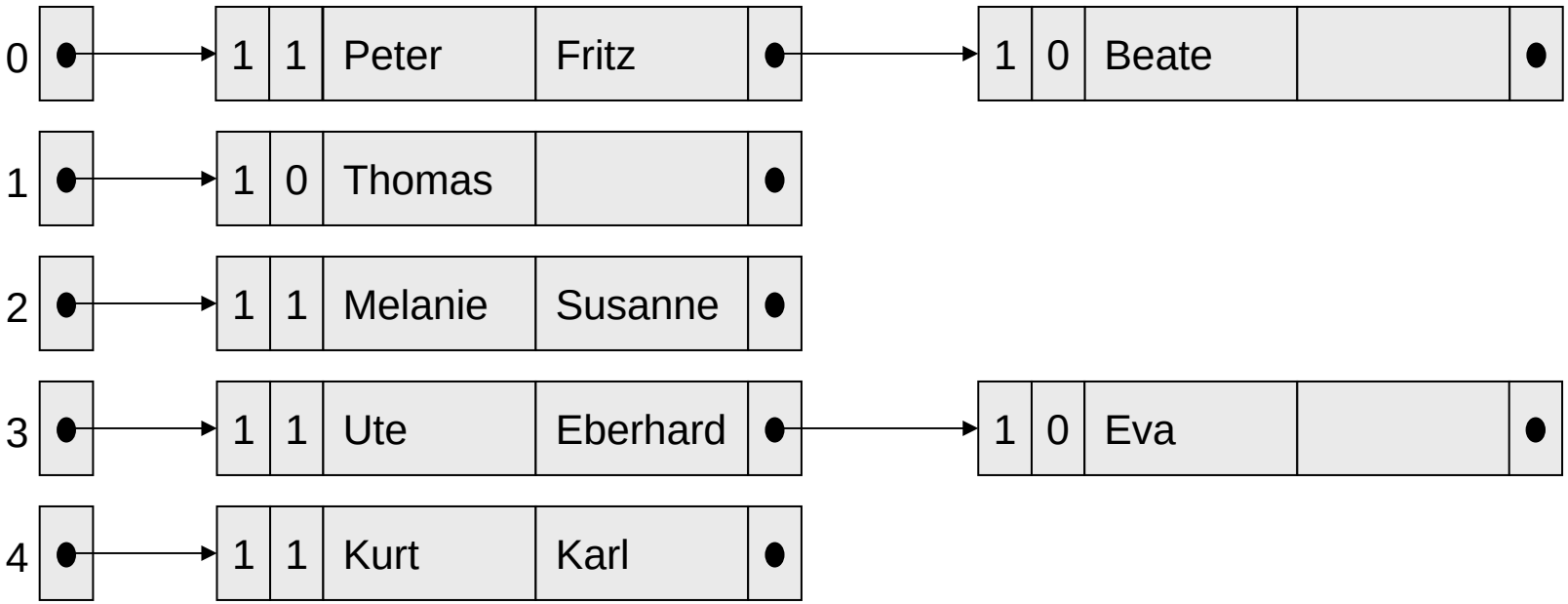
- **LOOKUP:**
  Berechne $h(v) = i$. Lies zuständigen Directory-Block, durchsuche alle Blöcke

- **DELETE:**
  LOOKUP, dann Löschbit setzen.

- **INSERT:**
  Zunächst LOOKUP. Falls Satz mit *v* vorhanden: Fehler. Sonst: Freien Platz im Block überschreiben oder neuen Block anfordern.

- **MODIFY:**
  Falls Schlüssel beteiligt: DELETE und INSERT
  Andernfalls: LOOKUP und überschreiben.

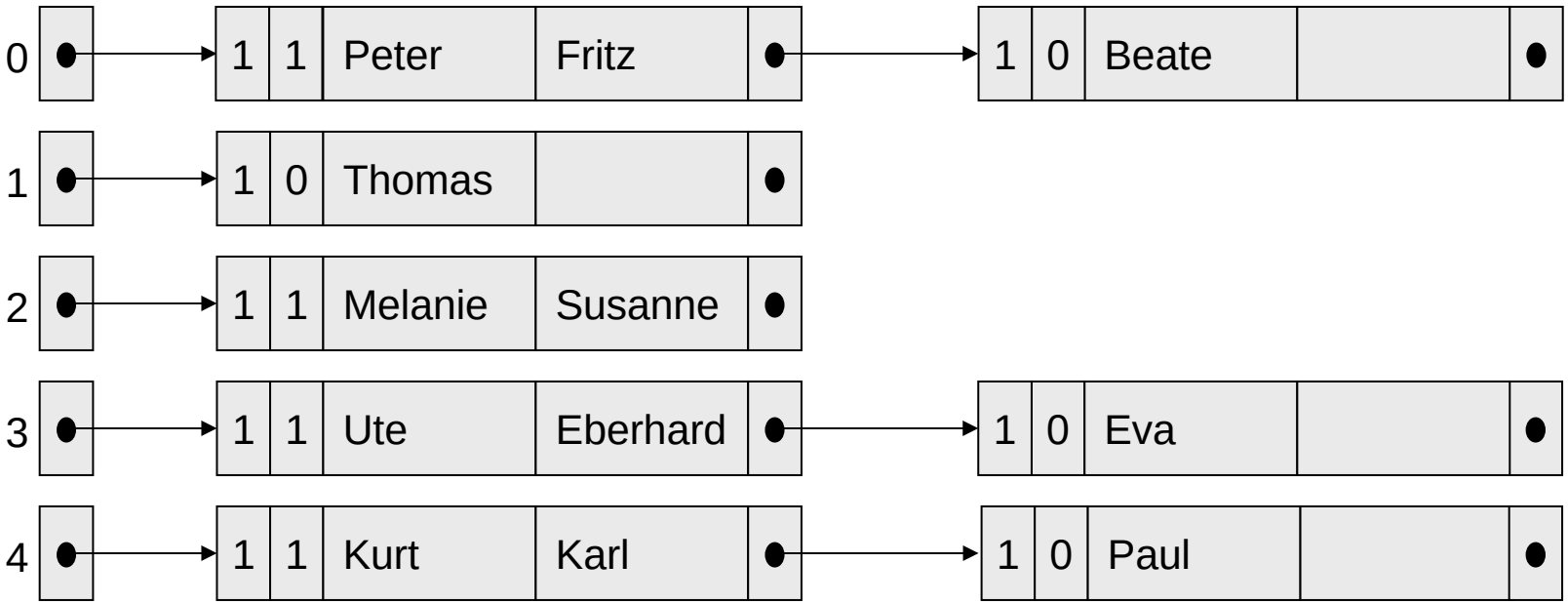
Beispiel für Hashorganisation



Hashorganisation: Ausgangslage $h(s) = |s| \bmod 5$

Paul einfügen

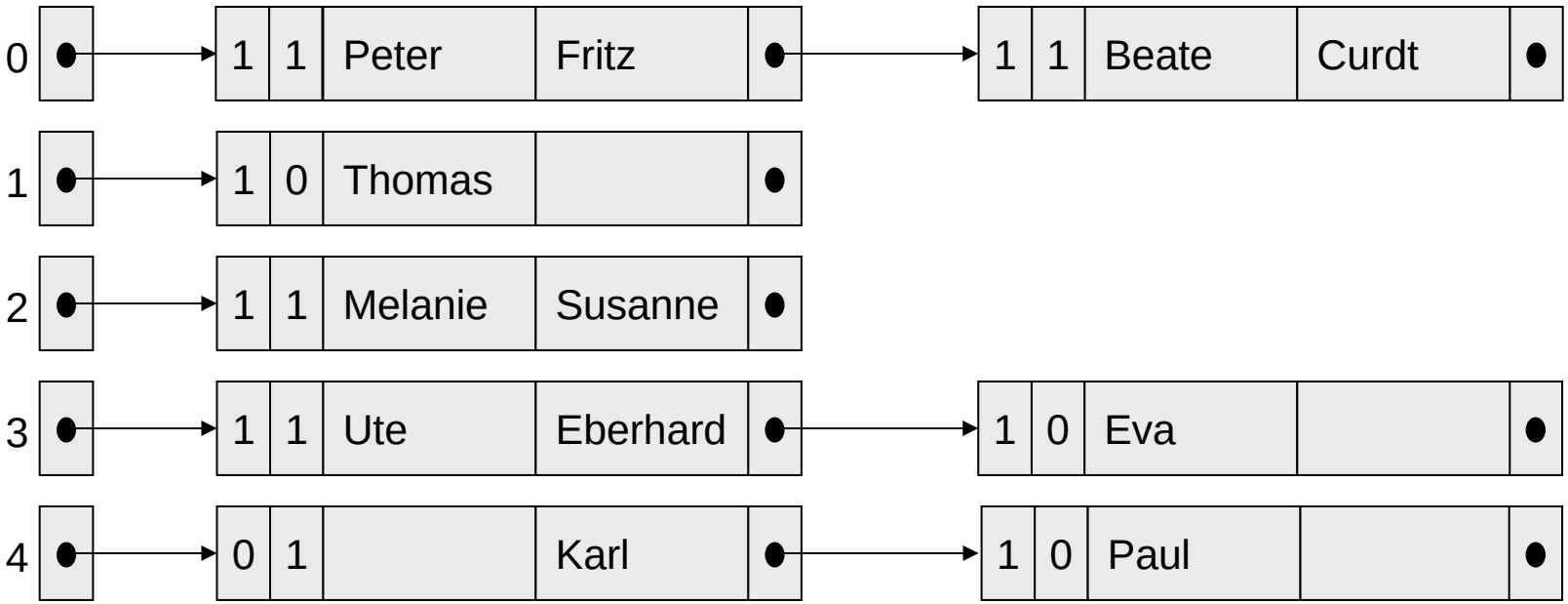
Beispiel für Hashorganisation



Hashorganisation: nach Einfügen von Paul

Kurt umbenennen nach Curdt

Beispiel für Hashorganisation



Hashorganisation: nach Umbenennen von Kurt in Curdt

Probleme beim Hashing

- Keine Sortierung
- Keine Bereichsabfragen
- Blocklisten werden immer länger
- Reorganisation erforderlich

ISAM (Index sequential access method)

- Index-Datei mit Verweisen in die Hauptdatei.
- Index-Datei enthält Tupel
  < Schlüssel, Blockadresse>
  sortiert nach Schlüsseln.
- Liegt **<*v*,*a*>** in der Index-Datei, so sind alle Record-Schlüssel im Block, auf den *a* zeigt, größer oder gleich *v*.

ISAM-Operationen für Record mit Schlüssel *v*

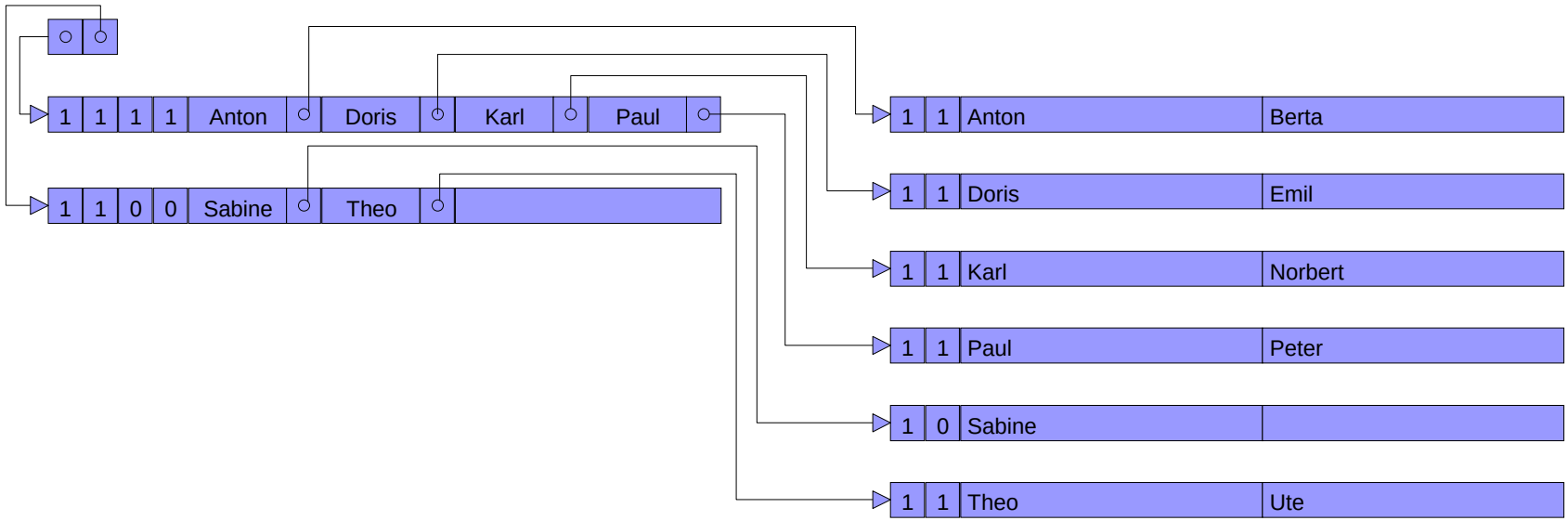
- **LOOKUP** (für Schlüssel v):
  Suche in Index-Datei den letzten Block mit erstem Eintrag $v_2 \leq v$. Suche in diesem Block das letzte Paar (v_3, a) mit $v_3 \leq v$. Lies Block mit Adresse *a* und durchsuche ihn nach Schlüssel *v*.

- **INSERT:**
  Zunächst LOOKUP. Falls Block noch Platz für Record hat: einfügen. Falls Block voll ist: Nachfolgerblock oder neuen Block wählen und Index anpassen.

- **DELETE:**
  Analog zu INSERT

- **MODIFY:**
  Zunächst LOOKUP. Falls Schlüssel an Änderung beteiligt: DELETE + INSERT. Sonst: Record ändern, Block zurückschreiben.

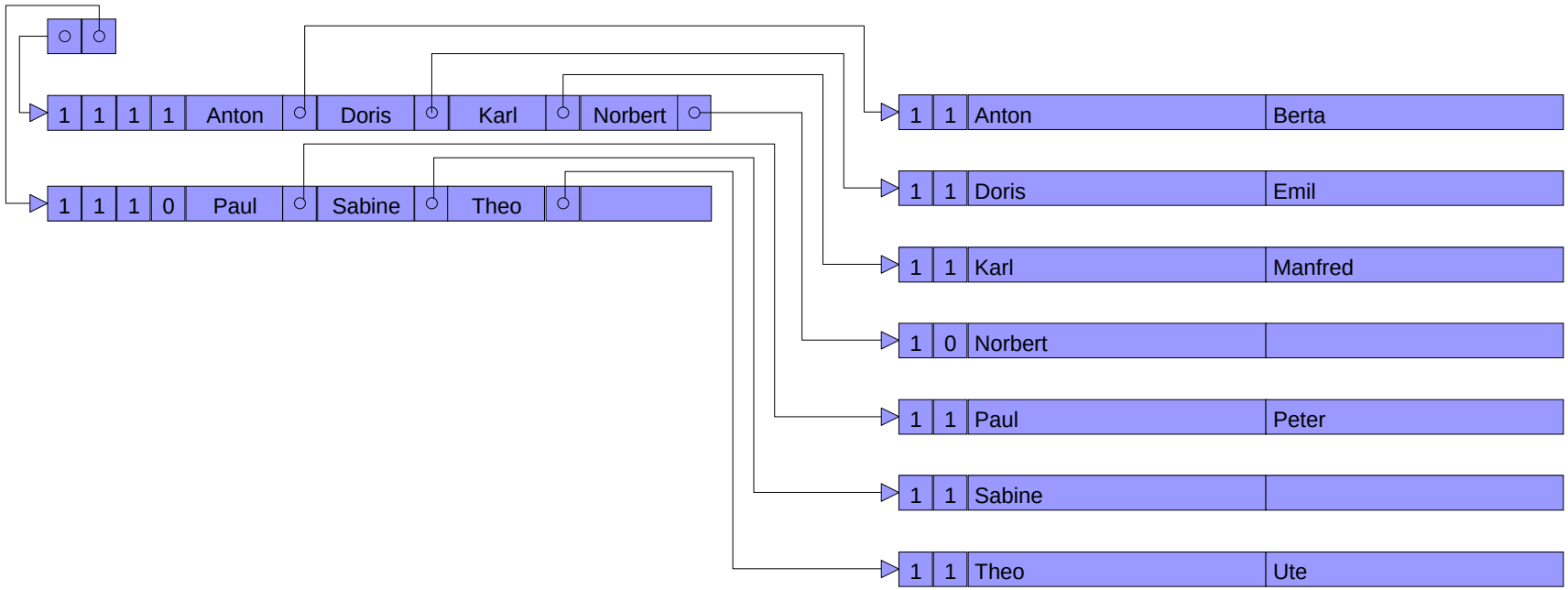
Beispiel für Indexorganisation



Index-Organisation: Ausgangslage

Manfred einfügen

Beispiel für Indexorganisation



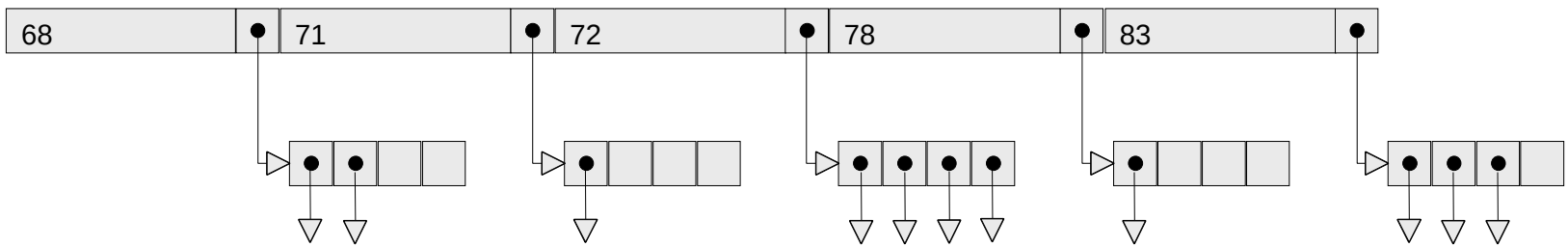
Index-Organisation: nach Einfügen von Manfred

Sekundär-Index

Sekundärindex besteht aus Index-File mit Einträgen der Form **<Attributwert, Adresse>**.

Liegt **<v,a>** im Sekundärindex, so verweist v auf Block mit Verweisen auf Records in Hauptdatei mit Attribut ≥ **v**

Sekundär-Index für Gewicht



Beispiel zur physikalischen Speicherung

Gegeben: 300.000 Records

Attribut	Bytes
Pers-Nr.	15
Vorname	15
Nachname	15
Straße	25
PLZ	5
Ort	25

Platzbedarf pro Record: 100 Bytes.

Die Blockgröße betrage 1024 Bytes.

Fragen zur Zahl der Records

Wieviel Daten-Records passen in einen zu 100% gefüllten Datenblock?

1024 / 100 = 10

Wieviel Daten-Records passen in einen zu 75% gefüllten Datenblock?

10 * 0,75 = 7-8

Wieviel Schlüssel / Adresspaare passen in einen zu 100% gefüllten Indexblock?

1.024 / (15+4) = 53

Wieviel Schlüssel / Adresspaare passen in einen zu 75% gefüllten Indexblock?

1.024 / (15+4)*0,75 ≈ 40

Heapfile versus ISAM

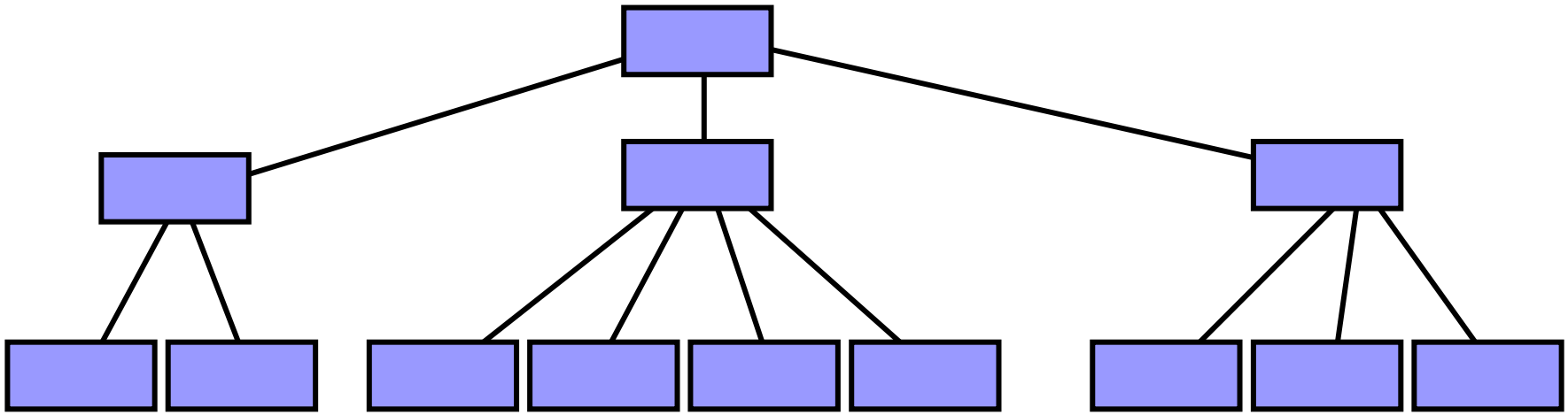
Welcher Platzbedarf entsteht beim Heapfile?
300.000 / 10 = 30.000 Blöcke

Wieviel Blockzugriffe entstehen im Mittel beim Heapfile?
30.000 / 2 = 15.000

Welcher Platzbedarf entsteht im Mittel bei ISAM?
300.000 / 7,5 $\approx$ 40.000 zu 75% gefüllte Datenblöcke +
40.000 / 40 $\approx$ 1.000 zu 75% gefüllte Indexblöcke

Wieviel Blockzugriffe entstehen im Mittel bei ISAM?
$\log_2(1.000) + 1 \approx 11$ Blockzugriffe

B*-Baum



- Jeder Weg von der Wurzel zu einem Blatt hat dieselbe Länge.
- Jeder Knoten außer der Wurzel und den Blättern hat mindestens k Nachfolger.
- Jeder Knoten hat höchstens $2 \bullet k$ Nachfolger.
- Die Wurzel hat keinen oder mindestens 2 Nachfolger.

B*-Baum-Adressierung

Ein Knoten wird auf einem Block gespeichert

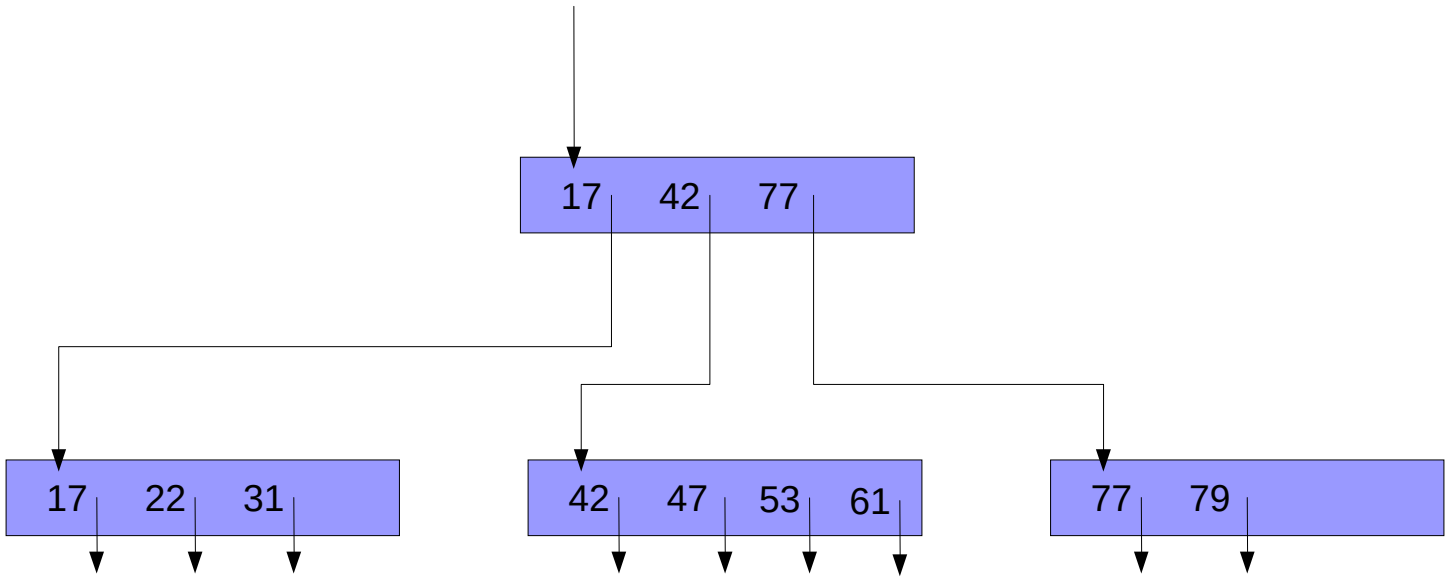
Ein Knoten mit *j* Nachfolgern ($j \leq 2 \bullet k$)
speichert *j* Paare von Schlüsseln und Adressen (s_1, a_1), . . . , (s_j, a_j).

Es gilt $s_1 \leq s_2 \leq \ldots \leq s_j$.

Eine Adresse in einem Blattknoten führt zum Datenblock mit den restlichen Informationen zum zugehörigen Schlüssel

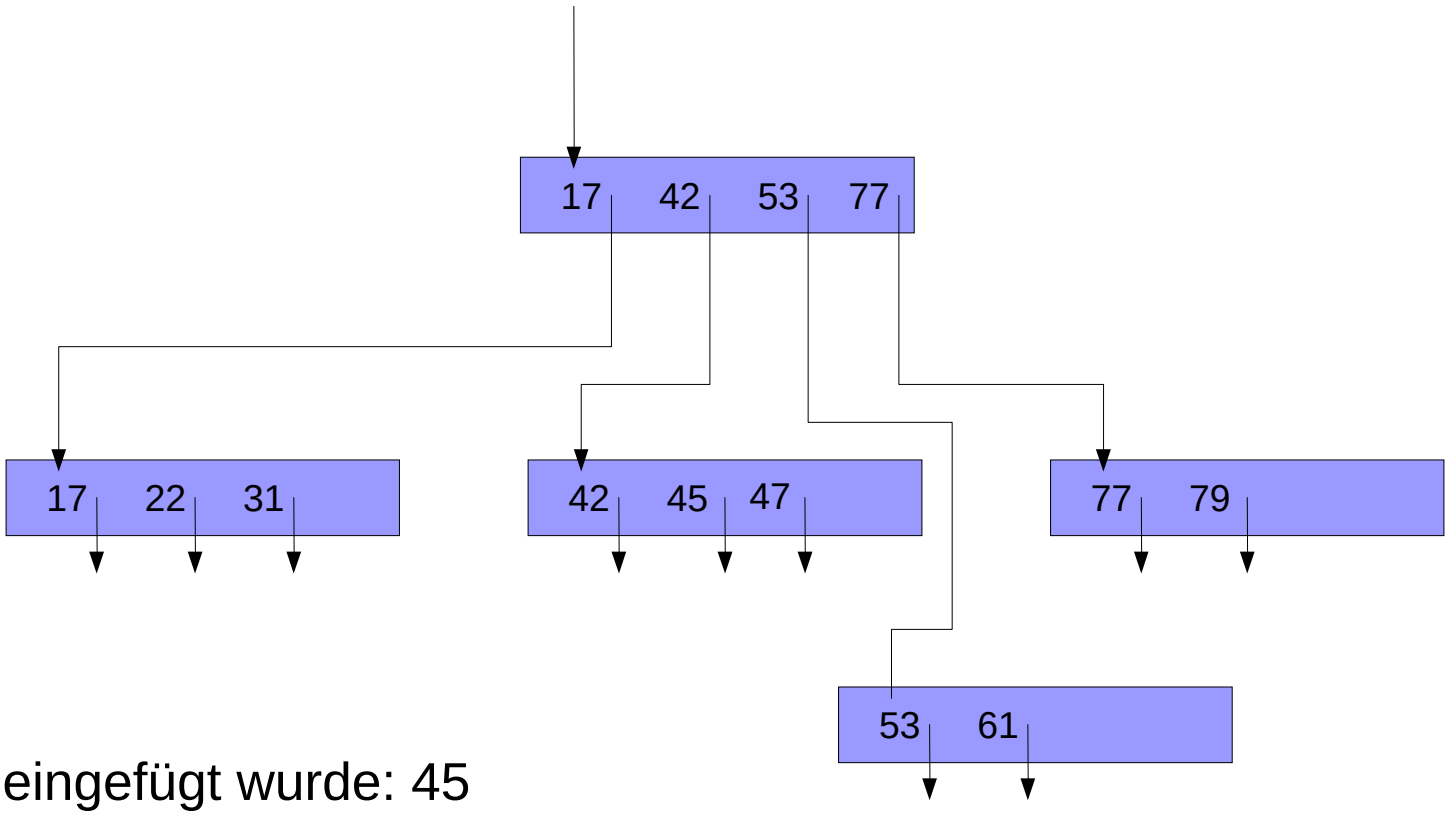
Eine Adresse in einem anderen Knoten führt zu einem Baumknoten

Einfügen in B*Baum



eingefügt werden soll: 45

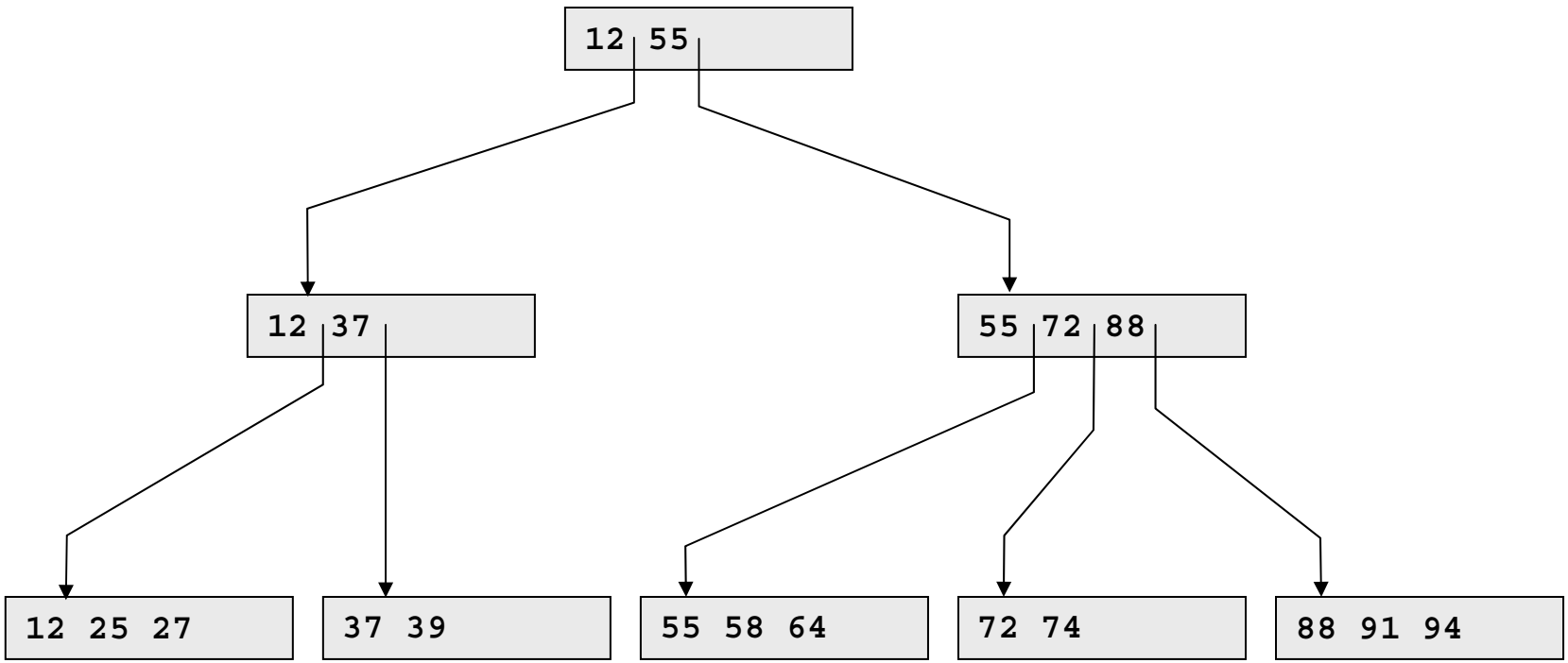
Einfügen in B*Baum



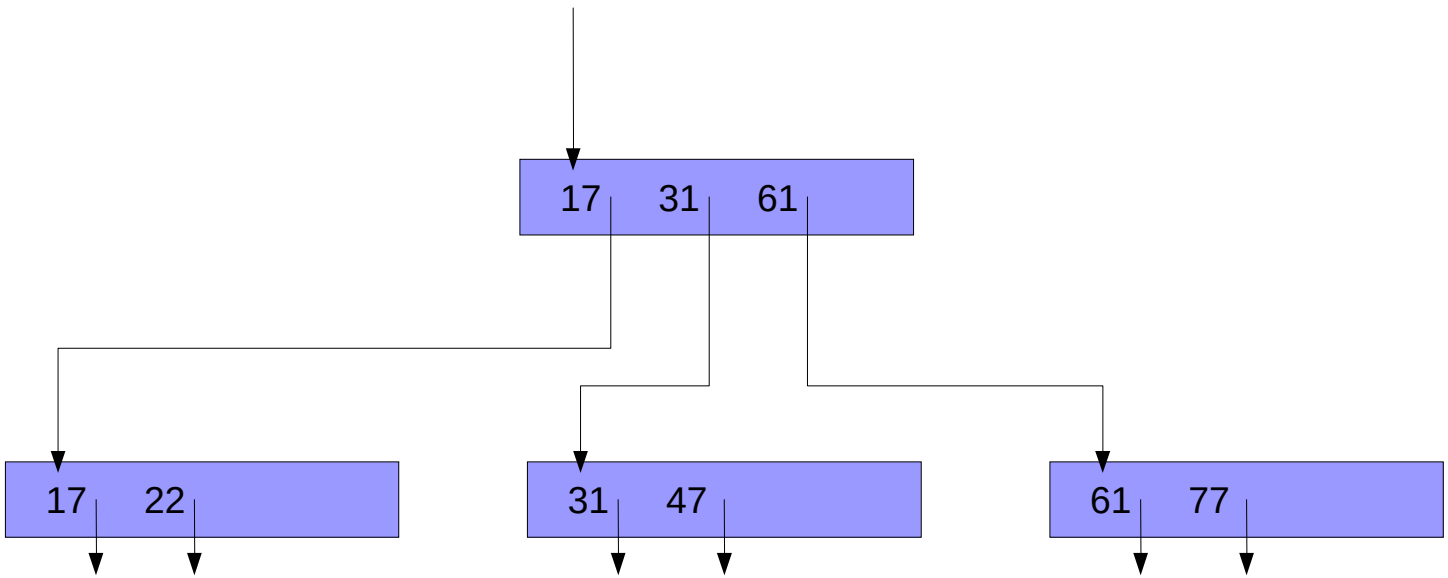
eingefügt wurde: 45

Sequenz für B*-Baum mit k=2

27 55 12 94 37 88 72 39 25 88 74 58 64

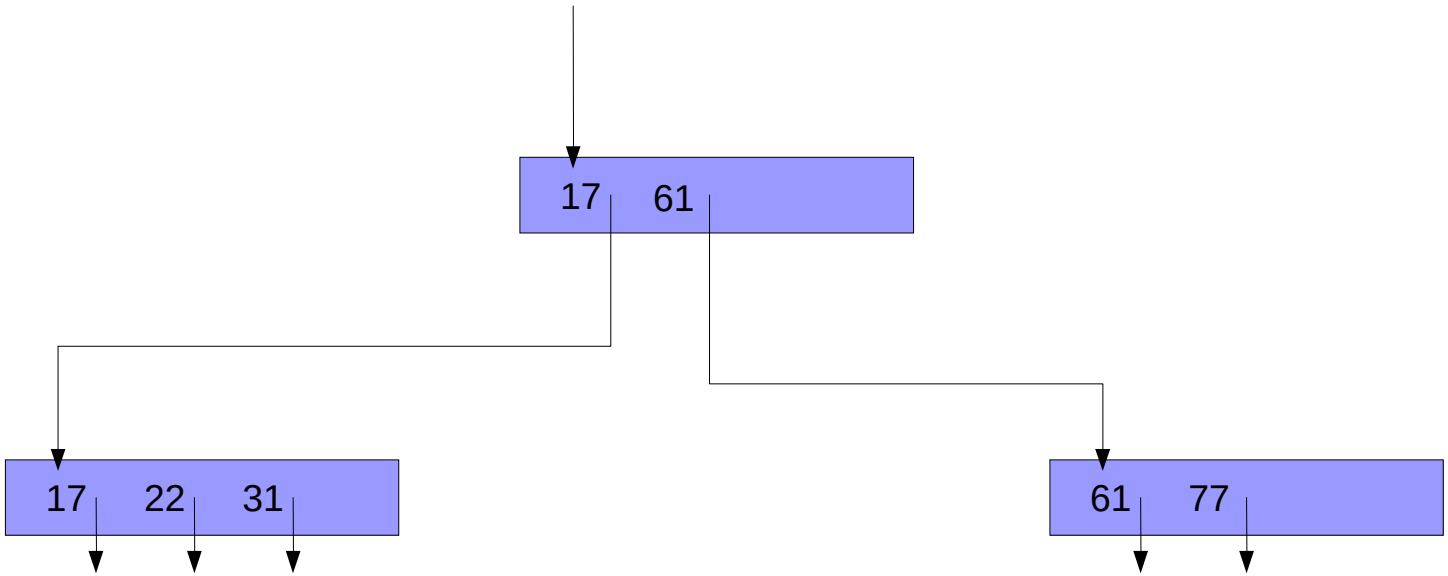


Löschen in B*Baum



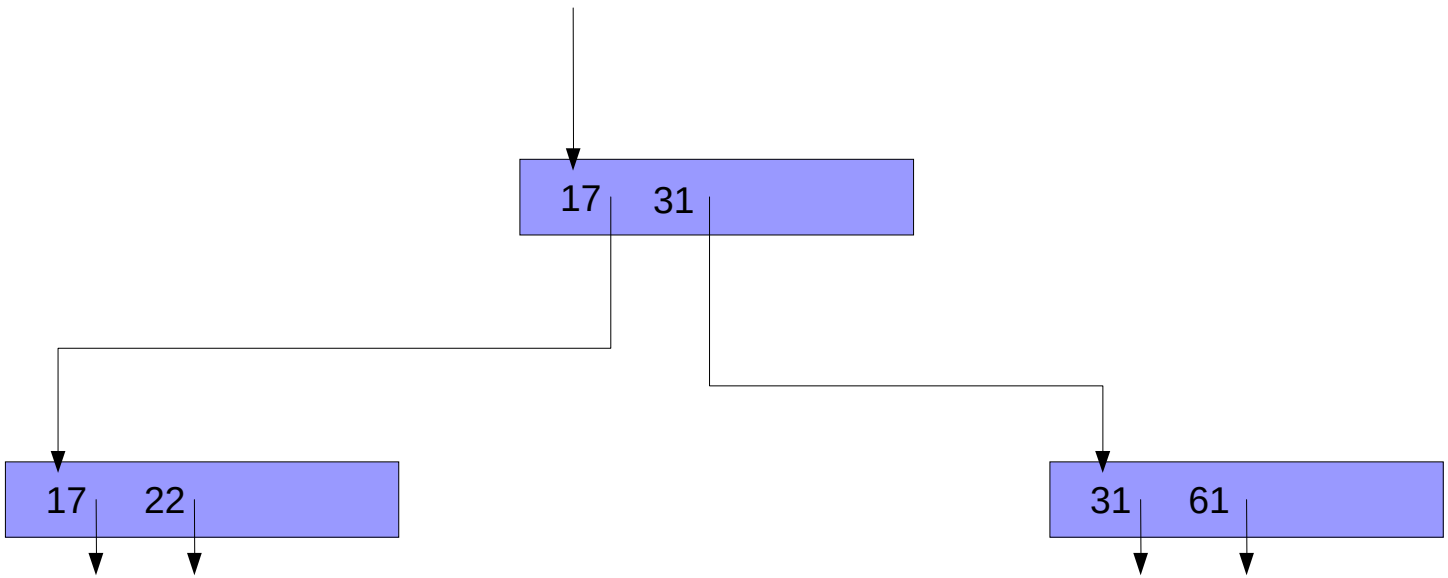
Entferne 47

Löschen in B*Baum



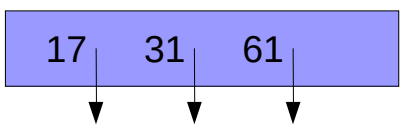
Entferne 77

Löschen in B*Baum



Entferne 22

Löschen in B*Baum



Fragen zum B*Baum

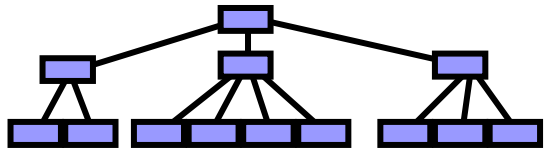
Wie groß ist k ?
Blockgröße / (Schlüssel / Adresspaar-Größe) =
1024 / (15+4) / 2 = 26

Wieviel Söhne hat eine zu 50 % gefüllte Wurzel ?
26

Wieviel Söhne hat ein zu 75 % gefüllter Knoten ?
39

Wieviel zu 75 % gefüllte Datenblöcke sind erforderlich ?
300.000 / 7,5 $\approx$ 40.000

Platzbedarf B*Baum



Wieviel Blöcke belegt der B*Baum ?

Höhe	Knoten	Zeiger aus Knoten
0	1	26
1	26	26 * 39 = 1.014
2	26*39	26*39*39 = 39.546

$\Rightarrow$ drei Ebenen reichen aus

$\Rightarrow$ Platzbedarf = 1 + 26 + 26*39 + 39.546 $\approx$ 40.000 Blöcke

Wieviel Blockzugriffe sind erforderlich ?

4

Hashing versus B*Baum

Welcher Platzbedarf entsteht beim Hashing, wenn dieselbe Zugriffszeit erreicht werden soll wie beim B*Baum?

4 Blockzugriffe = 1 Directory + 3 Datenblöcke

$\Rightarrow$ Buckets bestehen im Mittel aus 5 Blöcken.

$\Rightarrow$ von 5 Blöcken sind 4 voll und der letzte halb voll.

$\Rightarrow$ 4,5 * 10 = 45 Records pro Bucket

$\Rightarrow$ 300.000 / 45 = 6666 Buckets erforderlich

$\Rightarrow$ 6666 / (1024 / 4) = 26 Directory-Blöcke

$\Rightarrow$ Platzbedarf = 26 + 5 * 6.666 = 33.356

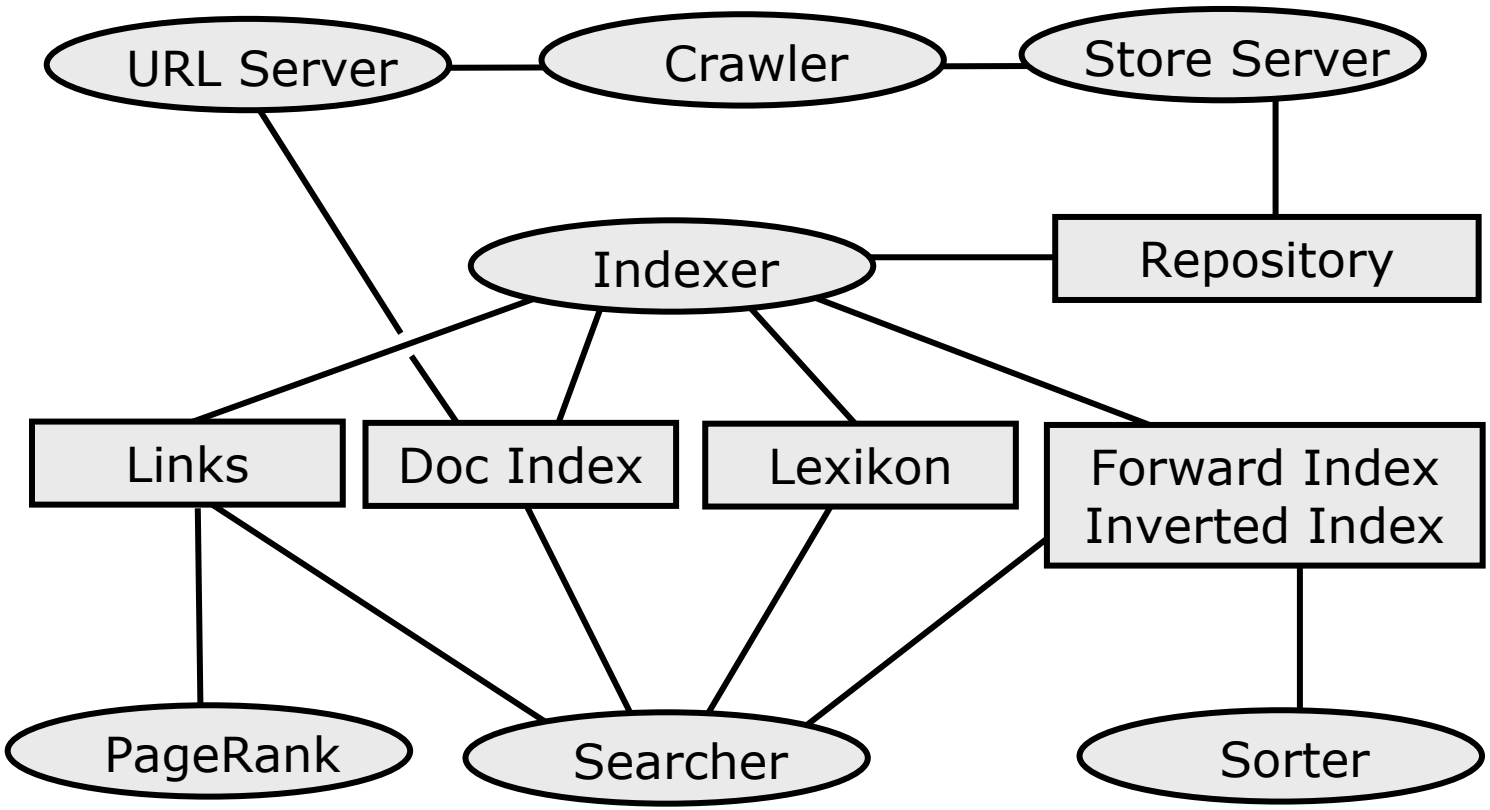
B*Baum versus Hashing

	B*Baum	Hashing
Vorteile	Dynamisch Sortierung möglich	Schnell platzsparend
Nachteile	Speicheroverhead kompliziert	keine Sortierung Neuorganisation

Google

- 1998: Sergey Brin + Larry Page
- 2004: 2.500 Mitarbeiter
- Oberfläche in 100 Sprachen
- 8.000.000.000 Webseiten im Cache
- 40.000.000.000.000 Bytes Plattenplatz
- 100.000.000 Queries pro Tag
- 1.000 Queries pro Sekunde
- 14.000.000 Wörter im Lexikon

Systemarchitektur



Repository

- komplettes HTML
- komprimiert mit zlib (1:3)

```
docID, docLength, docURL, docContent
```

Lexikon

- 14 Millionen Einträge
- jedes Wort gehasht auf **wordID**

wordID	**#docs**	**pointer**

zeigt auf erste Seite
im Inverted Index
mit **docIDs**,
relevant für **wordID**

HIT

- jeweils kodiert in 2 Bytes:
  Bit 00: capitalization
  Bit 01-03: font size
  Bit 04-15: position

- Plain Hit: innerhalb von HTML
- Fancy Hit: innerhalb von
  - URL
  - title
  - anchor text
  - meta tag

Forward Index

erzeugt vom Indexer aus Repository

```
docID wordID  #hits  hit hit hit hit hit
      wordID  #hits  hit hit hit
      wordID  #hits  hit hit hit hit

docID wordID  #hits  hit hit hit
      wordID  #hits  hit hit
      wordID  #hits  hit hit hit hit
      wordID  #hits  hit hit hit
```

Inverted Index

erzeugt vom Sorter aus Forward Index

```
wordID docID  #hits  hit hit hit hit hit
       docID  #hits  hit hit hit
       docID  #hits  hit hit hit hit
       docID  #hits  hit hit hit hit


wordID docID  #hits  hit hit hit hit
       docID  #hits  hit hit hit hit hit
       docID  #hits  hit hit hit hit
```

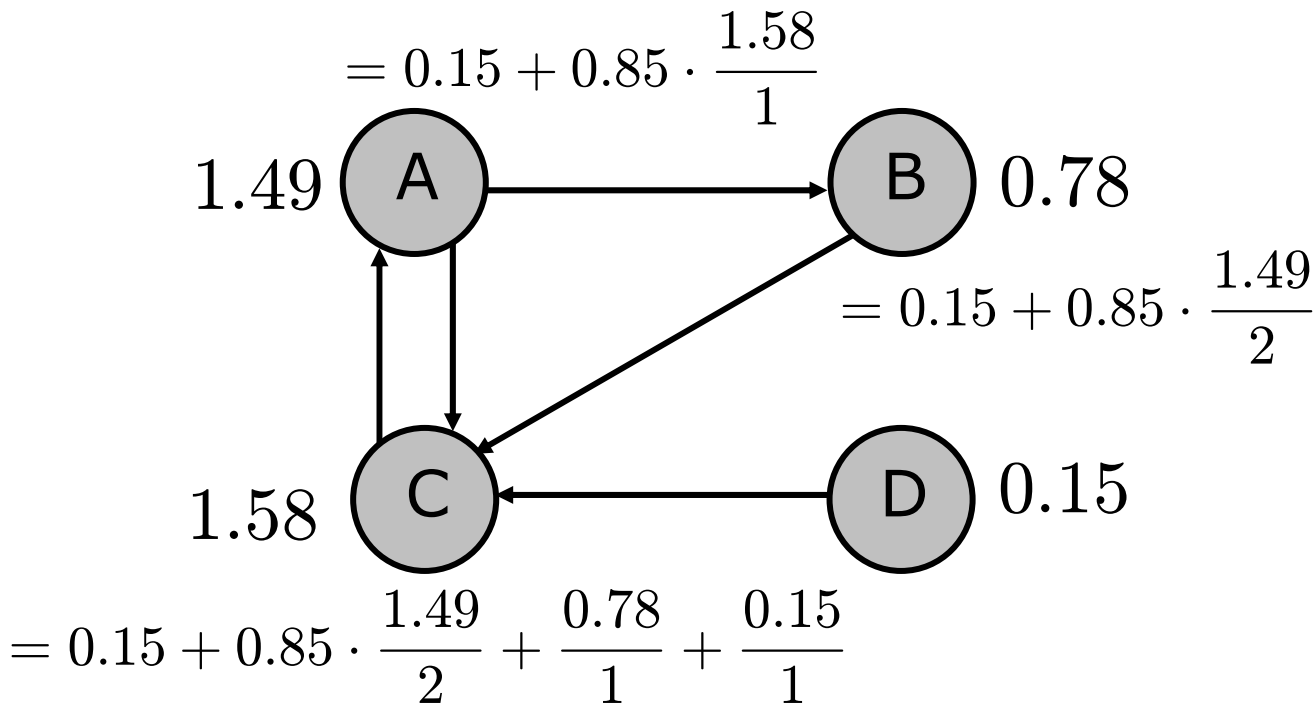
PageRank: Definition

Seite T habe $C(T)$ ausgehende Links
Seiten $T_1, T_2, \ldots, T_n$ zeigen auf Seite A
Gegeben sei Dämpfungsfaktor $0 \leq d \leq 1$

$$PR(A) := (1-d) + d \cdot \sum_{i=1}^{n} \frac{PR(T_i)}{C(T_i)}$$

$\Rightarrow$ Gleichungssystem, iterativ lösbar

PageRank: Beispiel



Dämpfungsfaktor $d = 0.85$

Single Word Query

Betrachte Hit-Liste für Dokument d bzgl. Word w

Hit-Type[title ,anchor ,URL, plain large, plain small, ...]
Hit-Weight [, , , , , …]
Hit-Count [, , , , , …]

Weight-Score (d,w) := Hit-Weight * Hit-Count
Final-Score (d,w) := Weight-Score (d,w) $\oplus$ PageRank (d)

Multi Word Query

- 10 Proximity Klassen
  (von "benachbart" bis "weit entfernt")

- Proximity-Weight belohnt nah beieinanderliegende Suchwörter

- Final-Score $(d, w_1, w_2, \ldots, w_n)$:=

  Weight-Score $(d, w_1, w_2, \ldots, w_n) \oplus$
  Proximity-Score $(d, w_1, w_2, \ldots, w_n) \oplus$
  PageRank (d)

http://www.google.de