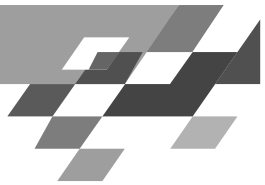
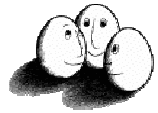
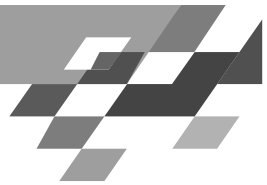


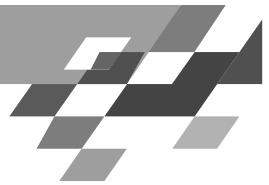
Speichern von Daten





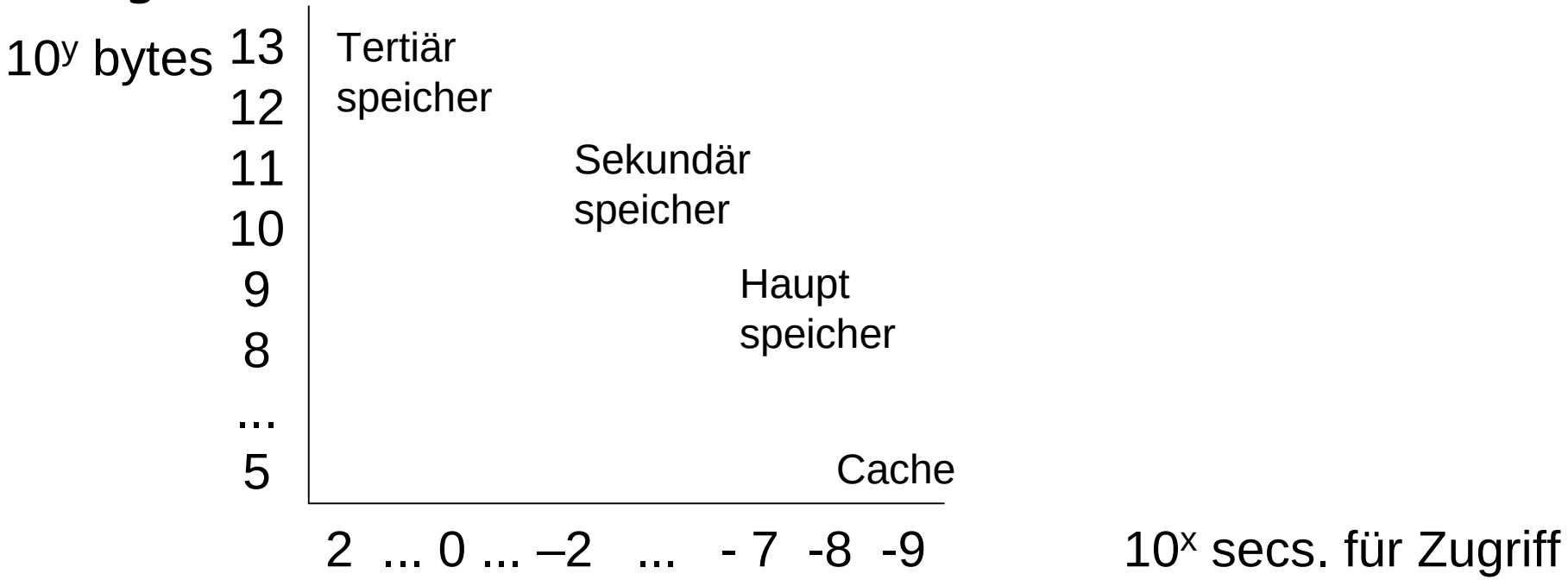
Unterschiede von DBMS und files

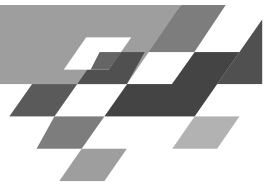
- DBMS unterstützt viele Benutzer, die gleichzeitig auf dieselben Daten zugreifen – concurrency control.
- DBMS speichert mehr Daten als in den Hauptspeicher passen.
  - Platten (Sekundärspeicher) oder sogar Bänder, CD, DVD (Tertiärspeicher) sind im Zugriff.
- DBMS organisiert die Daten so, dass minimal viele Plattenzugriffe nötig sind.



Zugriffszeit vs. Speicherplatz

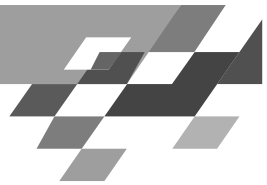
- Sekundärspeicher können mehr speichern, sind aber langsamer als der Hauptspeicher.
- Tertiärspeicher können mehr speichern, sind aber langsamer als Platten.





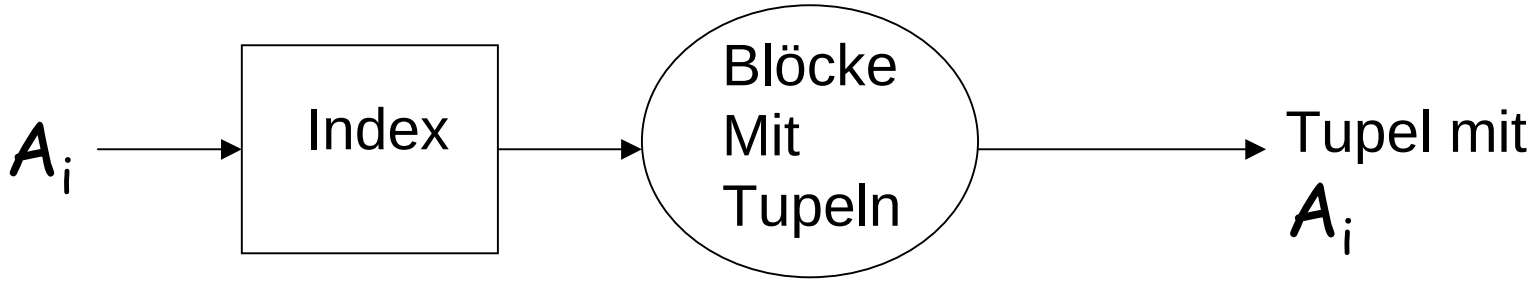
Konsequenz für Algorithmen

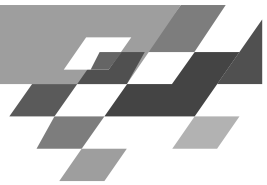
- Random Access Model -- Annahmen:
  - alle Daten sind im Hauptspeicher.
  - Zugriff dauert bei allen Daten gleich lange.
- DBMS haben nicht RAM Annahmen
  $\Rightarrow$ Algorithmen müssen anders geschrieben werden für dasselbe Problem.
  - Wenig Plattenzugriffe!
  - So viel wie möglich mit einem Tupel, das man mal hat, tun!
  - Daten, die häufig gebraucht werden, in den Cache ziehen!
  - Nur sehr wenige Daten aus dem Tertiärspeicher verwenden!



Index

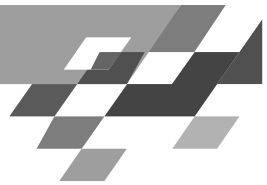
- Ein Index ist jede Datenstruktur, die als Eingabe eine Eigenschaft eines Tupels nimmt und das Tupel mit dieser Eigenschaft schnell findet.
- Typischerweise ist ein Index ein Verzeichnis, das zu einem Attribut A die möglichen Werte A_i angibt und auf die Speicherblöcke verweist, wo Tupel mit dem Wert A_i gespeichert sind.





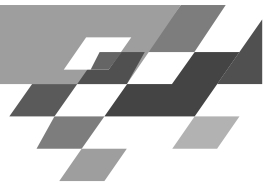
Wahl der Indizes -- Beispiel

- Q1: SELECT titel, jahr — Filme eines
  FROM stars — Schauspielers
  WHERE name=s;
- Q2: SELECT name — Schauspieler
  FROM stars — eines Films
  WHERE titel=t AND jahr=j;
- I: INSERT INTO stars VALUES (t, j, s);



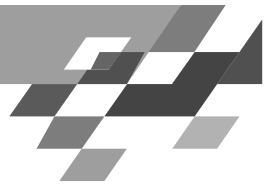
Kosten bei Anfragen

- stars sei auf Platte in 10 Blöcken gespeichert. Ohne Index machen wir also 10 Plattenzugriffe bei Q1 und Q2.
- Jeder Schauspieler macht durchschnittlich 3 Filme pro Jahr. Mit Index auf name brauchen wir 1 Zugriff auf den Index und 3 Plattenzugriffe für seine 3 Filme.
- Jeder Film hat durchschnittlich 3 Schauspieler genannt. Mit Index auf titel brauchen wir einen Zugriff auf den Index und 3 Plattenzugriffe für die 3 Schauspieler



Einfügekosten

- Ohne Index ist Einfügen (I) nur 1 Plattenzugriff, um in einem Block einen freien Platz zu finden und 1 Zugriff, um den Block mit dem neuen Tupel zurückzuschreiben.
- Mit Index müssen 2 Plattenzugriffe für die Veränderung des Index und 2 Zugriffe für das Platzfinden und Schreiben des Tupels, also 4 Zugriffe, vorgenommen werden.



Durchschnittskosten

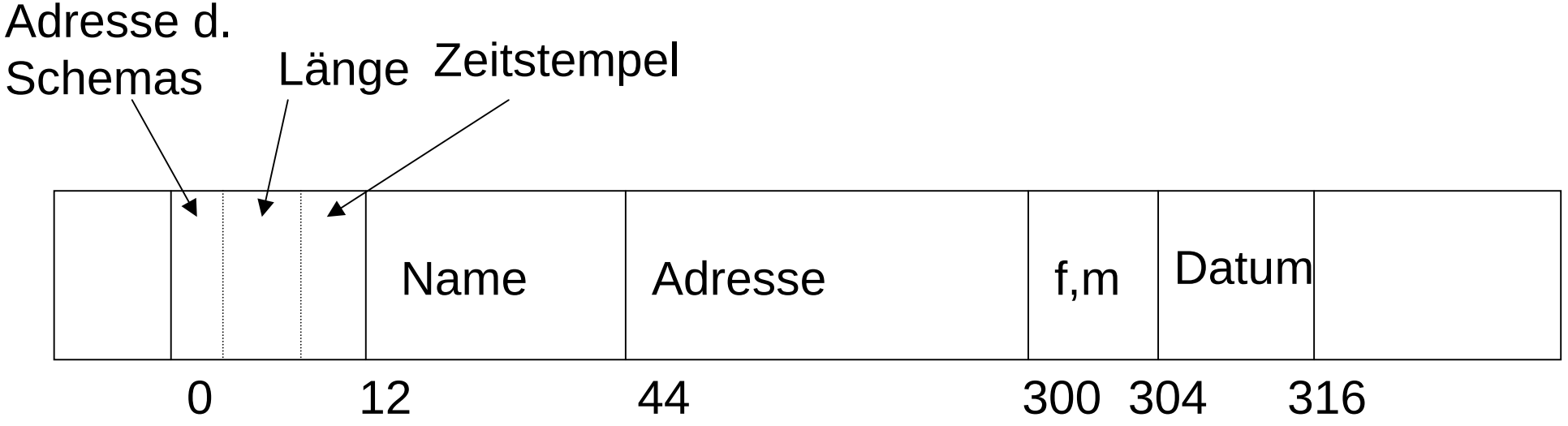
	Kein Index	Index auf name	Index auf titel	Beide Indizes
Q1	10	4	10	4
Q2	10	10	4	4
I	2	4	4	6
Durchschnitt	2+8p1+8p2	4+6p2	4+6p1	6-2p1-2p2

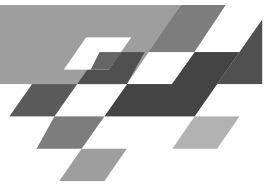
Sei p1 der Bruchteil der Zeit, in dem Q1 bearbeitet wird
(z.B. : bei 10% Q1-Anfragen ist p1=0,1)
p2 der für Q2, so ist der Bruchteil von I: 1-p1-p2.
Je nach tatsächlicher Häufigkeit sind unterschiedliche Indizes günstig.

Tupel eines Schemas

```
CREATE TABLE MovieStar(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE);
```

Tupel werden mit Metadaten gespeichert, die das Schema, die Tupellänge und zufüge- oder zuletzt-gelesen-Zeitpunkt angeben.

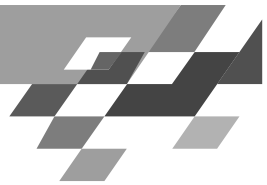




Blöcke

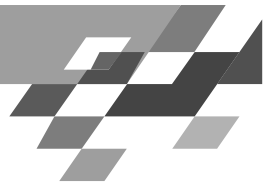
- Tupel werden in Blöcken gespeichert.
- Ganze Blöcke werden in den Hauptspeicher geladen.
- Um ein Tupel zu finden, muss man:
  - den Block in den Hauptspeicher laden, der das Tupel enthält.
  - innerhalb des Blocks das Tupel finden.

Kopf	$Tupel_2$	$Tupel_2$		$Tupel_n$



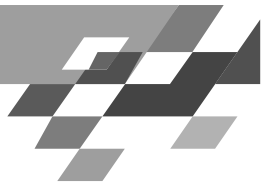
Indizes

- Ein Index dient dem schnellen Finden:
  - Eingabe: eine Eigenschaft eines Tupels (z.B. einen Attributwert) – Suchschlüssel.
  - Ausgabe: alle Tupel mit der Eigenschaft.
- Es kann mehrere Suchschlüssel für Tupel geben!
- Datenstrukturen für Indizes.
  - Schnelles Finden von Tupelmengen.
  - Leichtes Einfügen und Löschen von Tupeln.



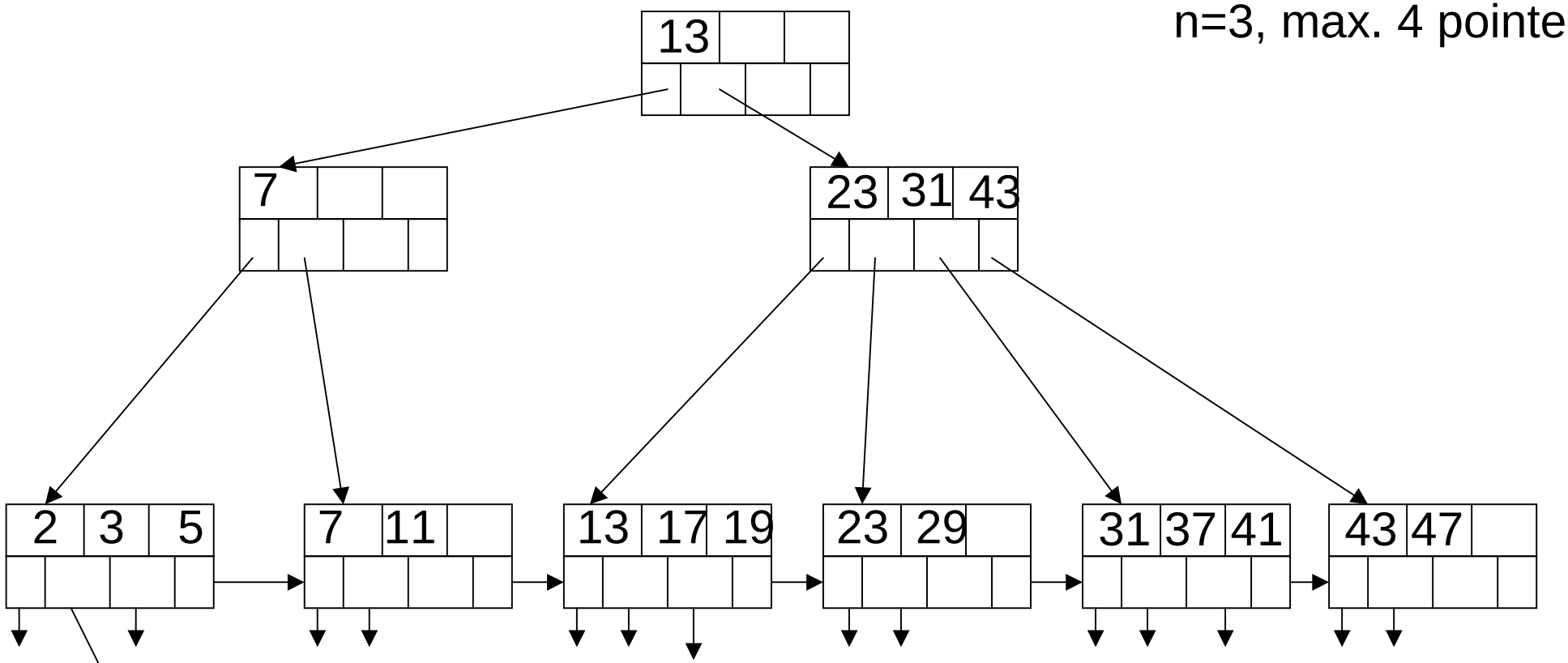
B-Trees

- B-trees verwalten so viele Suchschlüssel wie nötig.
- Ein Block in einem B-tree besteht aus n Suchschlüsseln und n+1 Zeigern. Die Zeiger der Blätter verweisen auf Tupel, der letzte auf den Nachbarblock.
- Jeder Block in einem B-tree ist halb oder vollständig gefüllt – nicht weniger. Er enthält mindestens 2 Zeiger.
- B-trees organisieren Blöcke in einem Baum fester Tiefe.

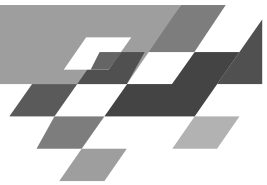


Beispiel für einen numerischen Suchschlüssel

n=3, max. 4 pointer



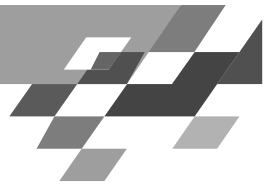
Block mit Tupeln mit Attributwert 3 (Suchschlüssel)



Suche im B-Tree

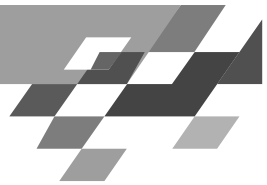
Suche nach Tupeln mit Suchschlüsselwert K:

- An Mittelknoten mit Werten K_1, ..., K_n:
  - $K < K_1 \Rightarrow$ zum 1.linken Unterknoten gehen,
  - $K_1 \leq K < K_2 \Rightarrow$ zum 2.linken Unterknoten gehen,
  - ...
- Am Blattknoten nachsehen, ob K da ist – der Zeiger zeigt auf Tupel mit K.



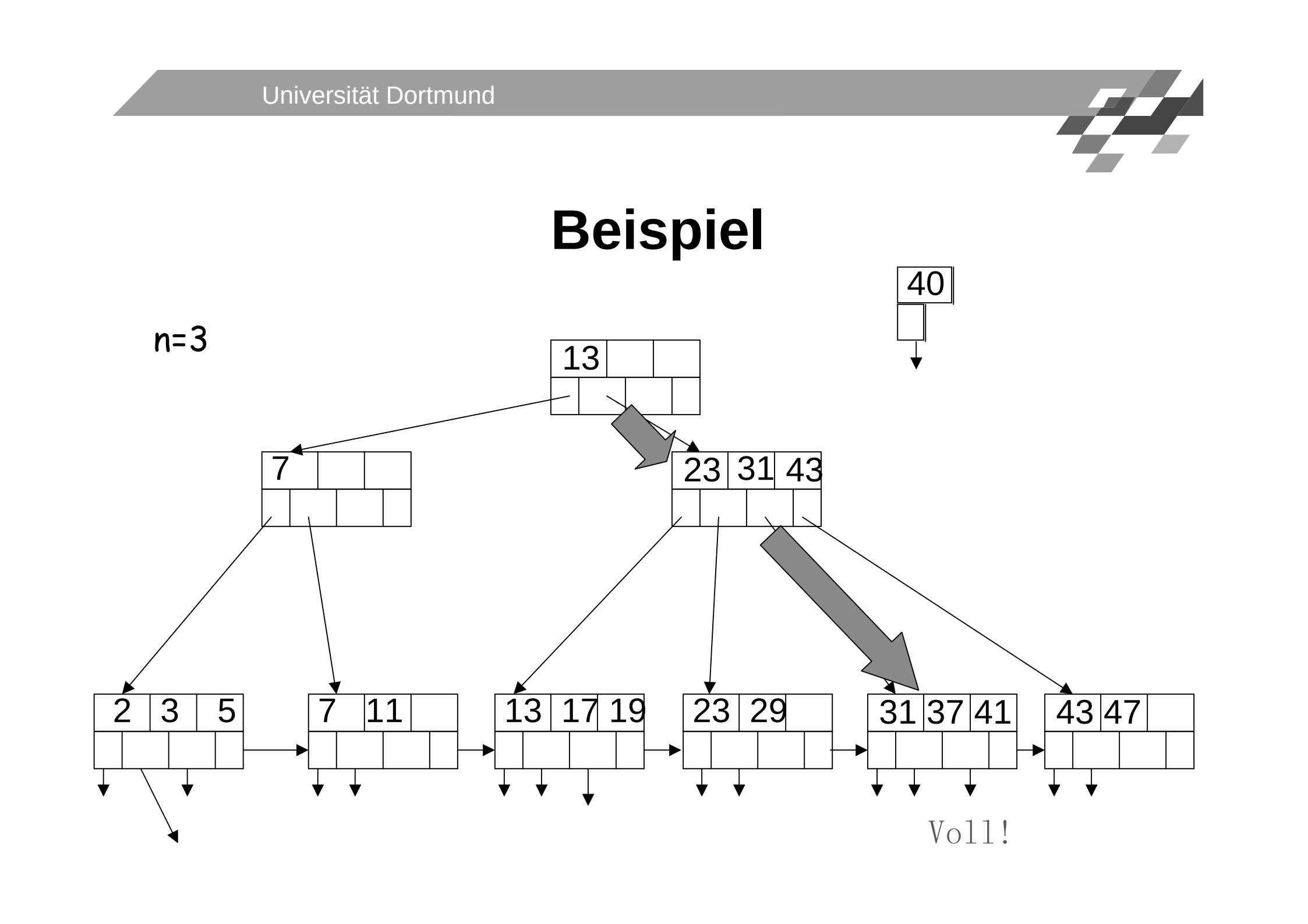
Einfügen von Tupeln

- Suche passenden Knoten für Schlüssel des Tupels.
  - Wenn dort noch Platz ist, füge Schlüssel ein.
  - Wenn kein Platz ist,
    1. teile den Knoten,
    2. füge im übergeordneten Knoten einen Zeiger hinzu, wenn dort noch Platz ist
    3. Sonst gehe zu 1.

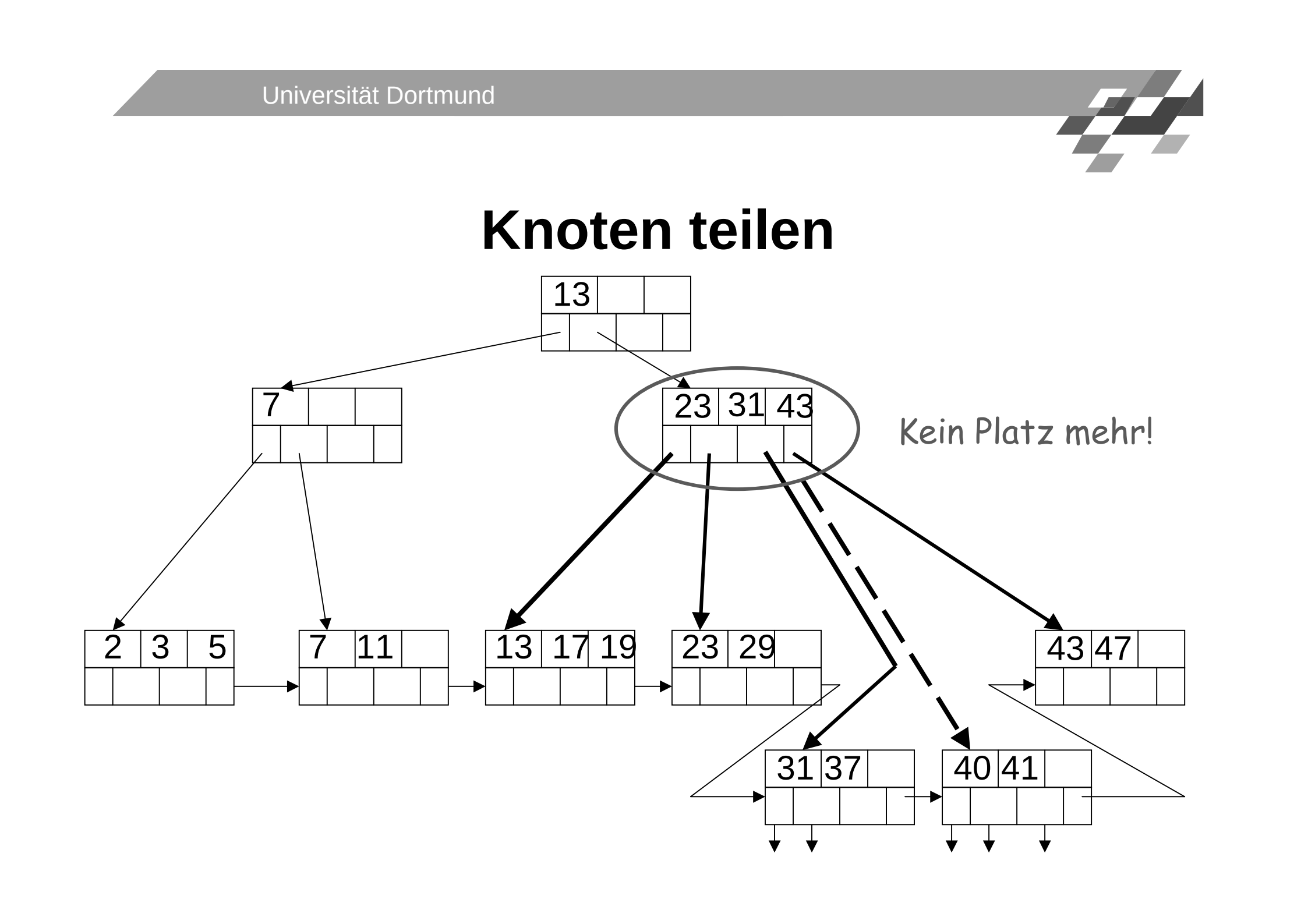


Teilen von Knoten

- Erzeuge neuen Knoten M neben dem ursprünglichen Knoten N.
- N behält die ersten $\lceil (n+2):2 \rceil$ Zeiger, M erhält die restlichen Zeiger.
- N behält die ersten $\lceil n:2 \rceil$ (aufgerundet-halben) Schlüssel, M erhält die letzten $\lfloor n:2 \rfloor$ (abgerundet-halben) Schlüssel, der verbleibende Schlüssel in der Mitte wandert eine Ebene hinauf.

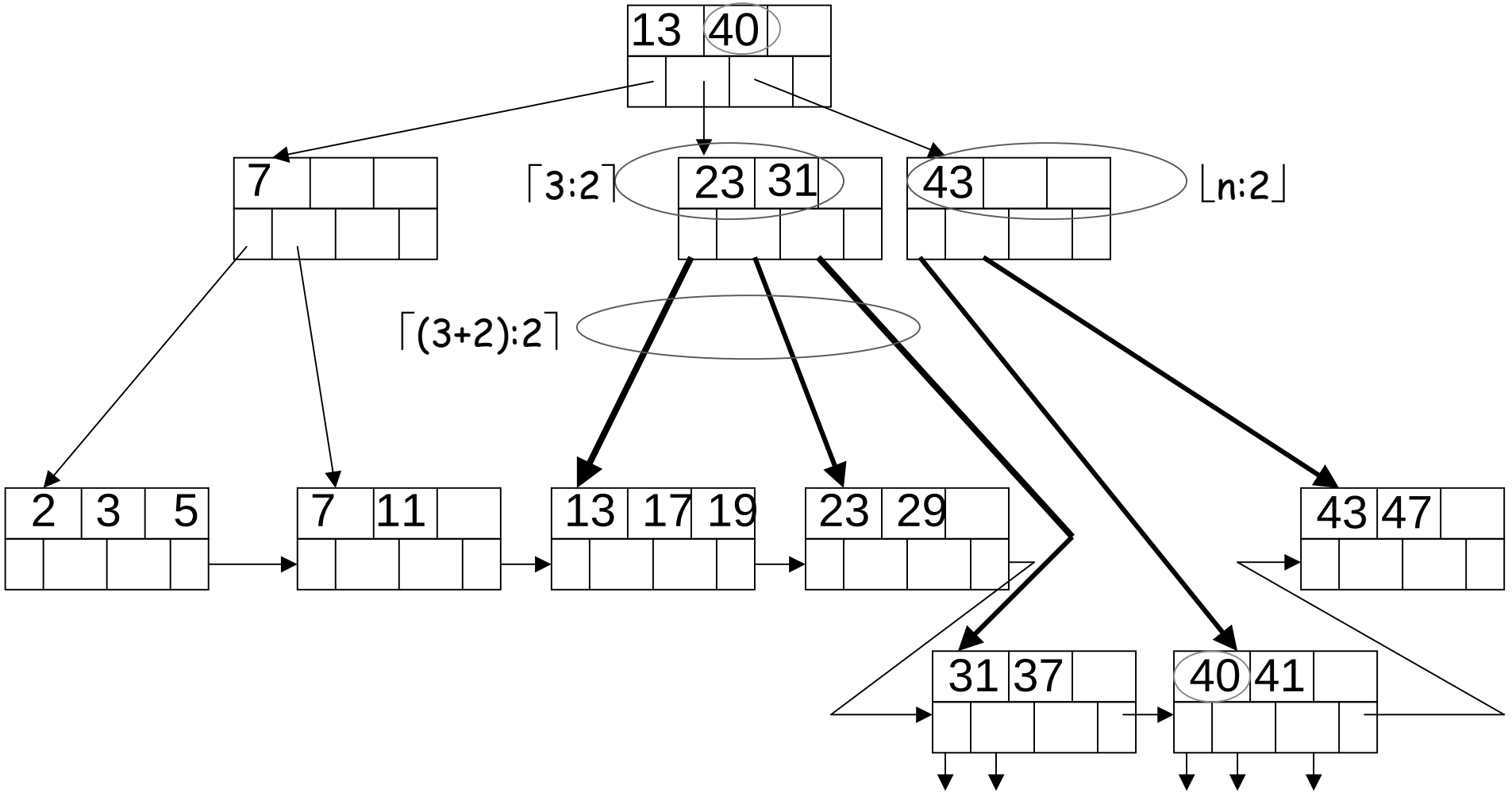


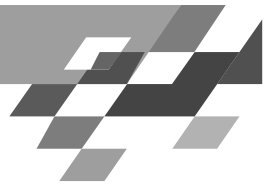
Beispiel



Knoten teilen

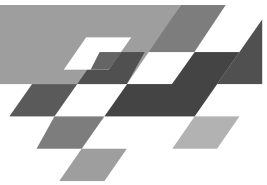
Nach oben propagieren





Eigenschaften der B-Trees

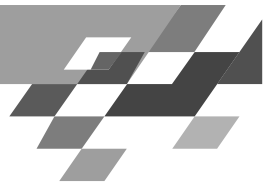
- Sehr wenige Plattenzugriffe:
  - Man liest von der Wurzel aus die passenden Knoten
    $\Rightarrow$ so viele Plattenzugriffe wie Ebenen des B-Trees + Lesen der Tupel (+ Reorganisation).
  - Meist genügen 3 Ebenen eines B-Trees und die Wurzel wird permanent im Hauptspeicher gehalten
    $\Rightarrow$ 2 Plattenzugriffe.
- Für geeignetes n (z.B. n=10) kommt Teilen oder Verbinden von Knoten selten vor, so dass die Reorganisation wenig kostet.



Darstellung von Mengen: Hashing

- Eine Hash-Funktion bildet M mögliche Elemente einer Menge auf eine feste Anzahl B von Beuteln ab.
- Im Idealfall befindet sich in einem Beutel genau eine Adresse.

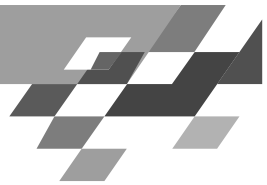
  h(x) → $\mathbb{N}$ liefert für einen Schlüssel x eine natürliche Zahl, die die Speicheradresse bezeichnet.
- Hash-Tabellen sind extrem effizient (meist nur 1 Plattenzugriff).



Beispiel Hash-Funktion

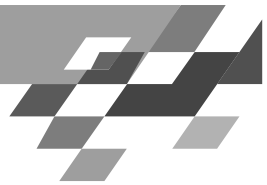
{braun, rot, blau, violett, türkis} M=5

- h(x)=Wortlänge – 3 ergibt 2, 0, 1, 4, 3
  perfekt, M=B, je Beutel 1 Adresse
- Buchstabenwerte c_i: A-Z,Ä,Ö,Ü durchzählen
- h(x)= ($\sum c_i$) modulo C
  - C=6 ergibt 2, 5, 0, 1, 4 perfekt, M=B (1 Element)
  - C=4 ergibt 0, 1, 0, 3, 2 B=4 (1,25 Elemente)
  - C=5 ergibt 1, 3, 1, 3, 1 B=2 (2,5 Elemente)
- B Adressen mit (1/B) M Elementen



Lastfaktor

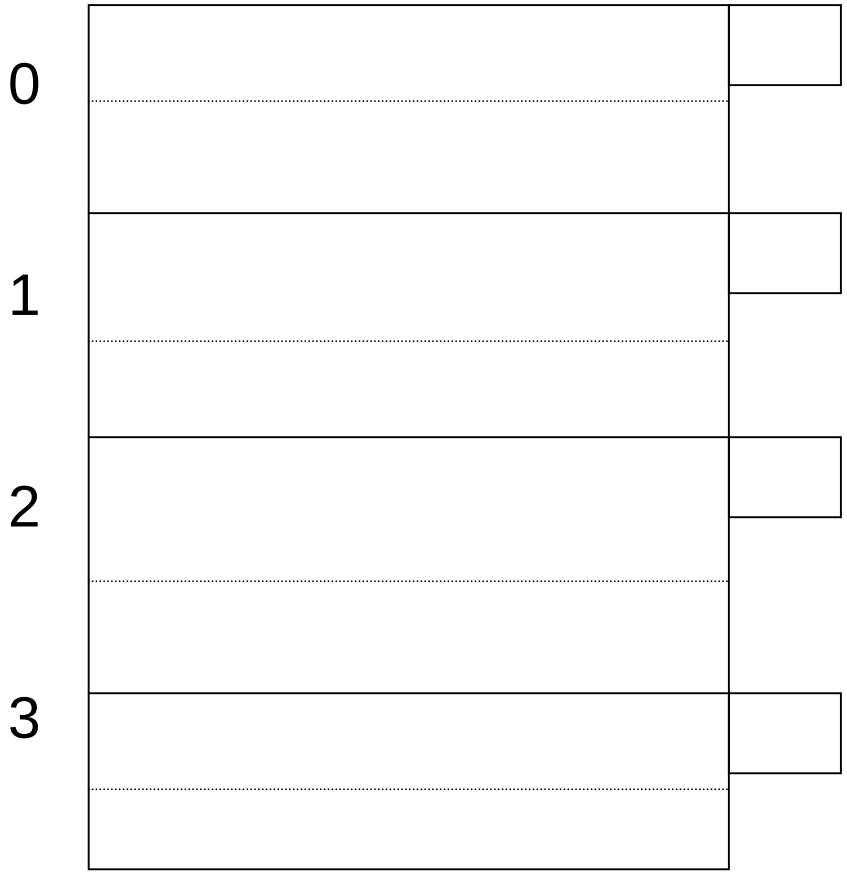
- Der Lastfaktor einer Hash-Tabelle ist bei der Anzahl B von Adressen und der Kardinalität M der darzustellenden Menge das Verhältnis M/B.
- Üblich: Lastfaktor 1,33.
- Abschätzung von M.
- Falls man sich verschätzt hat, muss neu angelegt werden. Aufwand in O(M)!



Hash-Tabellen

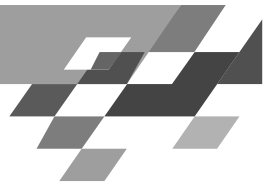
- Hash-Tabellen werden zur Verwaltung des Hauptspeichers eingesetzt.
- Sie können aber auch den Sekundärspeicher indexieren.
- Eine Hash-Tabelle enthält n Beutel, von denen jeder eine Menge von Objekten (hier: Blöcke) enthält.
- Eine Hash-Funktion h(K) errechnet für einen Schlüssel einen Wert v aus [0...n], v$\in \mathbb{N}$.

Beispiel einer Hash-Tabelle



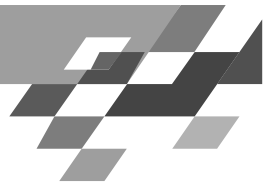
4 Beutel: 0,1,2,3
Jeder Beutel enthält 2 Tupel (+Überlauf).

Ein Tupel mit Suchschlüssel K gelangt in den Beutel h(K).



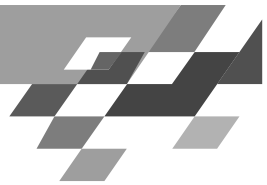
Suchen, Einfügen

- Suche nach Tupel mit Suchschlüssel K: Beutel mit der Nummer h(K) aufsuchen und darin nach Tupel suchen. Idealerweise hat der Beutel genau einen Block, dann ist Suche in O(1).
- Einfügen eines Tupels:
  Beutel mit h(K) finden und Tupel anhängen. Falls der Beutel voll ist, den Überlauf verwenden. Der Überlauf sollte nicht mehrere Blöcke umfassen.



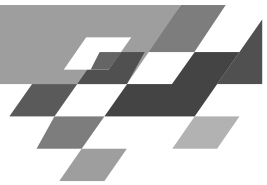
Linear hashing

- Indirekte Adressierung: die Beutel enthalten Adressen von Blöcken, nicht die Blöcke selbst.
- Dynamisch: Anzahl der Beutel wird immer so gewählt, dass die durchschnittliche Anzahl von Tupeln je Beutel 80% der in einen Block passenden Tupel beträgt.
  Da nur die Adressen verschoben werden und nicht die Daten, ist das möglich.



Genauer

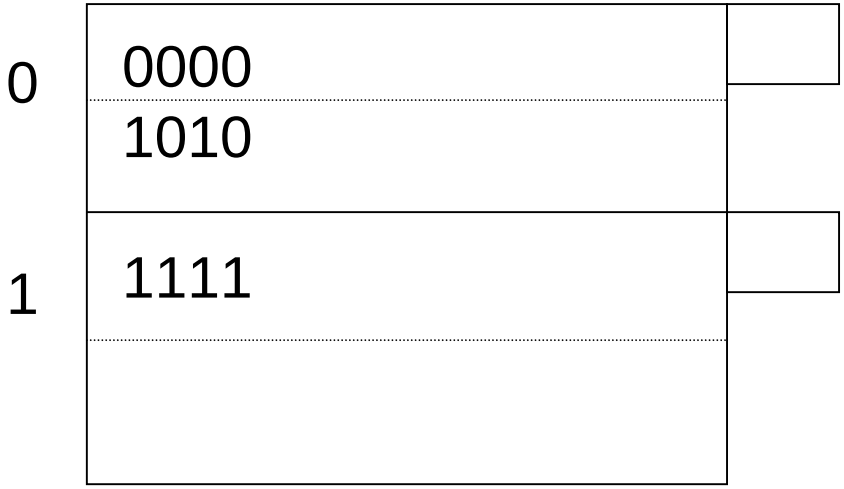
- Die hash-Funktion liefert eine k-bit Binärzahl.
- Eine i-bit Binärzahl, i<k, i= $\lceil \log_2 n \rceil$ nummeriert die Beutel durch, bei gegenwärtig n Beuteln.
- Es werden die i-hintersten Stellen von h(K) genommen!



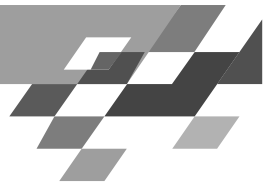
Beispiel

k=4, i=1, n=2

3 Tupel (r=3)

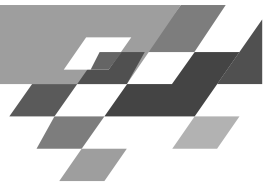


Im Beutel mit der Nummer 0 sind alle Tupel, deren Suchschlüssel mit 0 endet, in dem mit der Nummer 1 sind alle Tupel, deren Suchschlüssel mit 1 endet.
Soll r < 1,7 n gelten, dann muss ein neuer Beutel hinzugefügt werden.



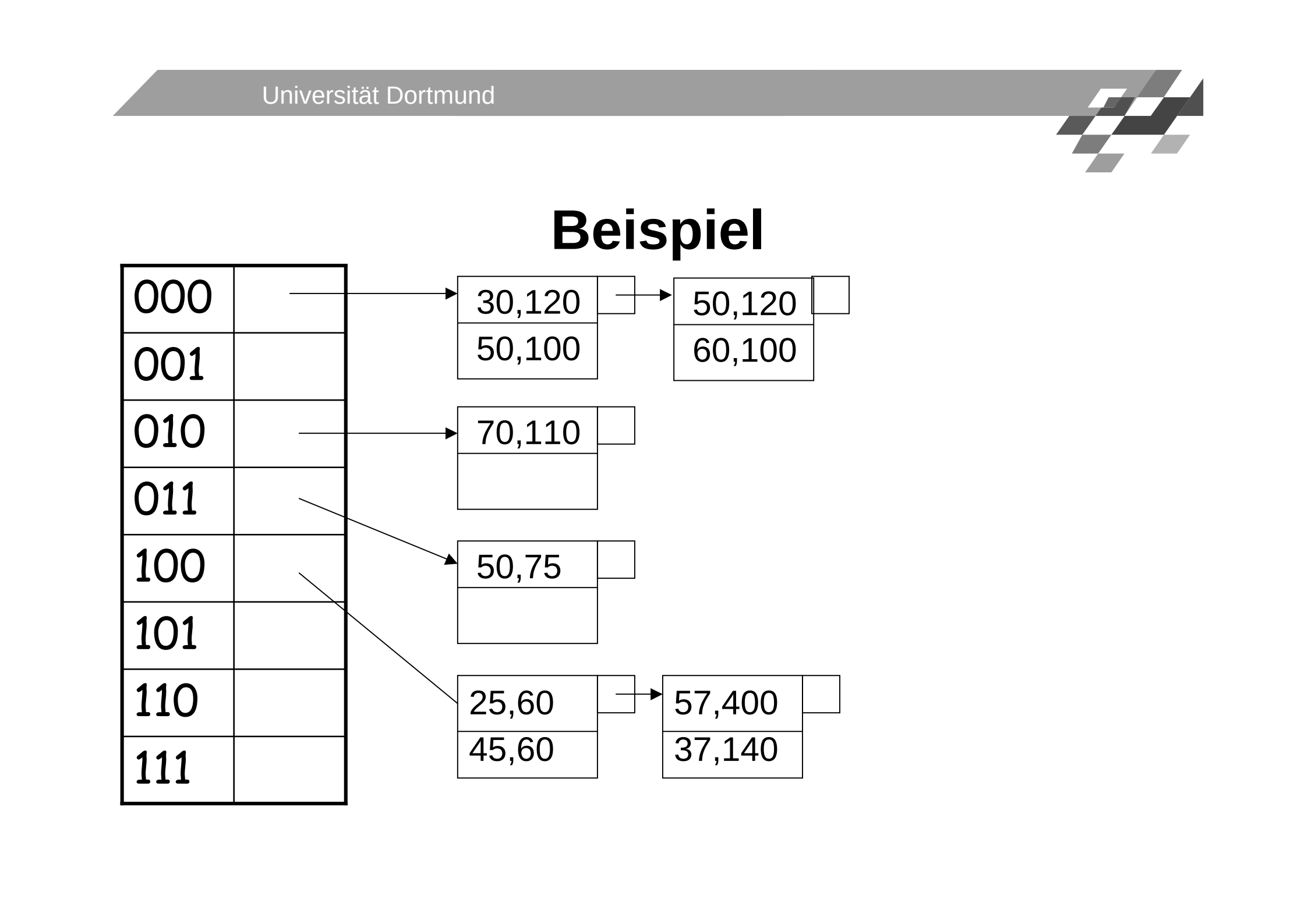
Partitioned hashing

- Multidimensionale Indexierung:
  - Mehrere Attribute $A_1,\ldots, A_n$ sollen als Suchschlüssel verwendet werden.
  - Es sollen aber bei einer Suche nicht alle Attribute A_i durch Werte v_i angegeben werden müssen.
- Die bits der hash-Funktionen $h_1(v_1),\ldots, h_n(v_n)$ werden konkateniert und ergeben so die k-bit der Gesamt-hash-Funktion.



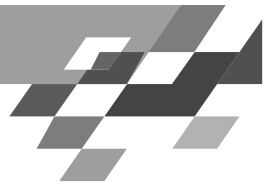
Beispiel

- Indexierung nach
  - Alter h(a)= a modulo 2
    Gerades Alter ergibt 0yz, ungerades Alter ergibt 1yz, (y,z sind Variablen für bits).
  - Gehalt h(g)=g:1000 modulo 4
    Rest 1 ergibt x01, Rest 3 ergibt 11.
  - Gesamt-hash-Wert ist xyz.
- h(50,75000) ergibt 011.



Beispiel

000
001
010
011
100
101
110
111
30,120
50,100
50,120
60,100
70,110
50,75
25,60
45,60
57,400
37,140



Was wissen Sie jetzt?

- Sie ahnen, dass Datenbanken etwas anderes sind als eine Sammlung von Dateien.
- Sie kennen einige Datenstrukturen zur Speicherverwaltung:
  - B-Trees,
  - Hash-Tabellen mit linear hashing und partitioned hashing
- Sie wissen dass jede Datenstruktur mit Operationen einhergeht, hier besprochen: Suchen und Einfügen.