



Speichern von Daten





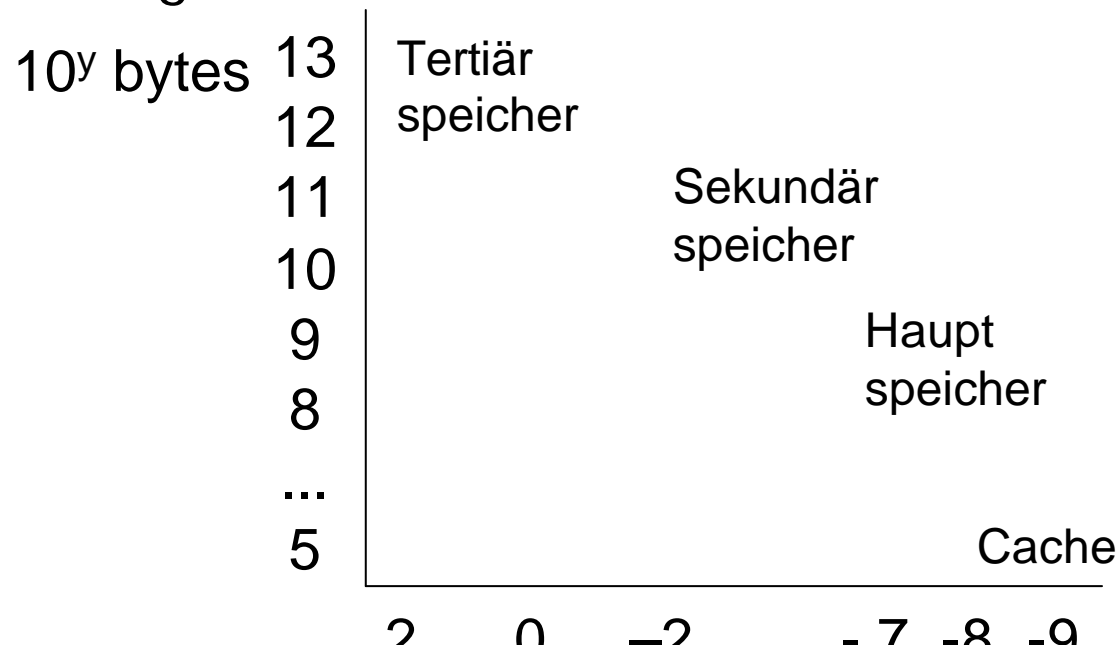
Unterschiede von DBMS und files

- 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.



10^x secs für Zugriff



Konsequenz für Algorithmen

- Random Access Model -- Annahmen:
 - alle Daten sind im Hauptspeicher
 - Zugriff dauert bei allen Daten gleich lange
- DBMS haben nicht RAM Annahmen
⇒ Algorithmen müssen anders geschrieben werden für dasselbe Problem.

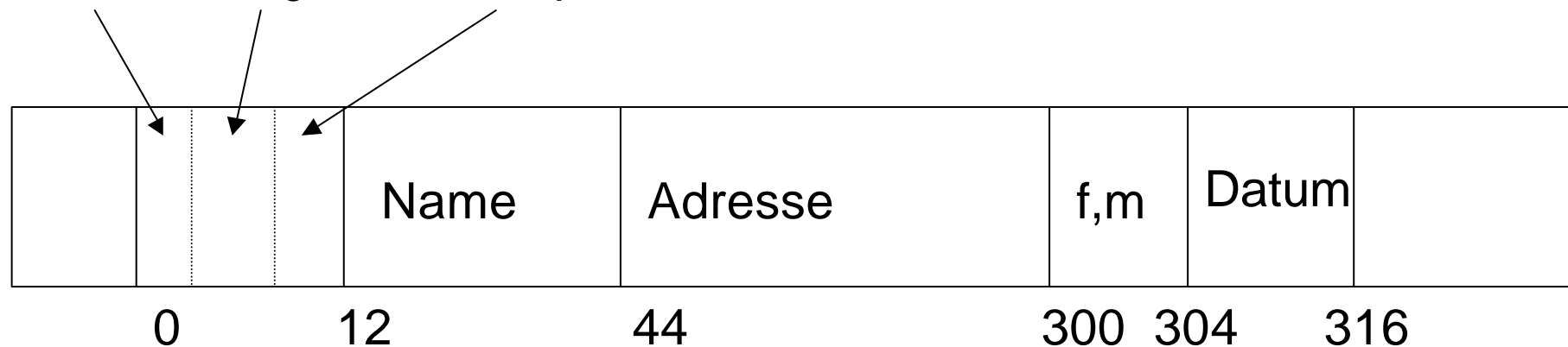


Tupel eines Schemas

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

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

Adresse d.
Schemas Länge Zeitstempel





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

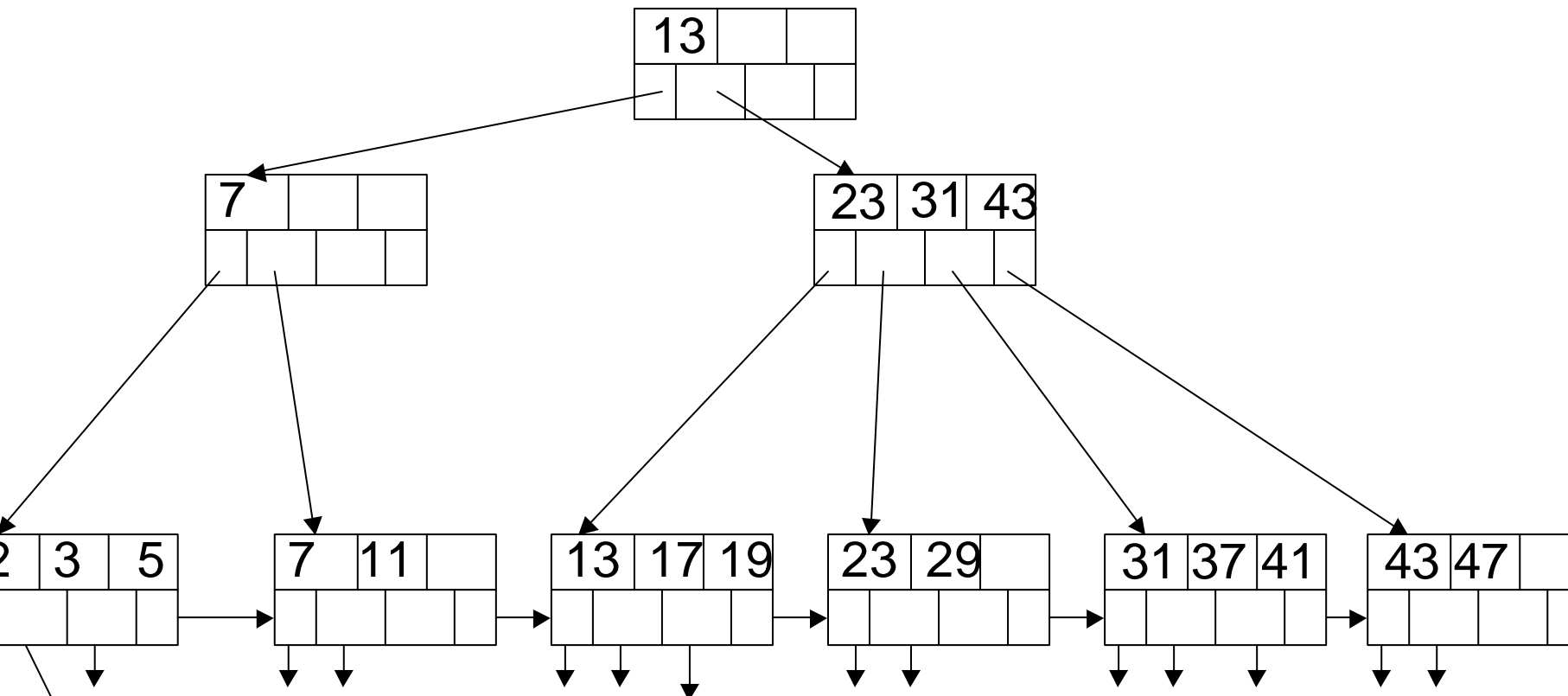


B-Trees

- 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.
- 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



Block mit Tupeln mit Attributwert 3 (Suchschlüssel)



Suche im B-Tree

Such nach Tupeln mit Suchschlüsselwert K :

- An Mittelknoten mit Werten K_1, \dots, 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.



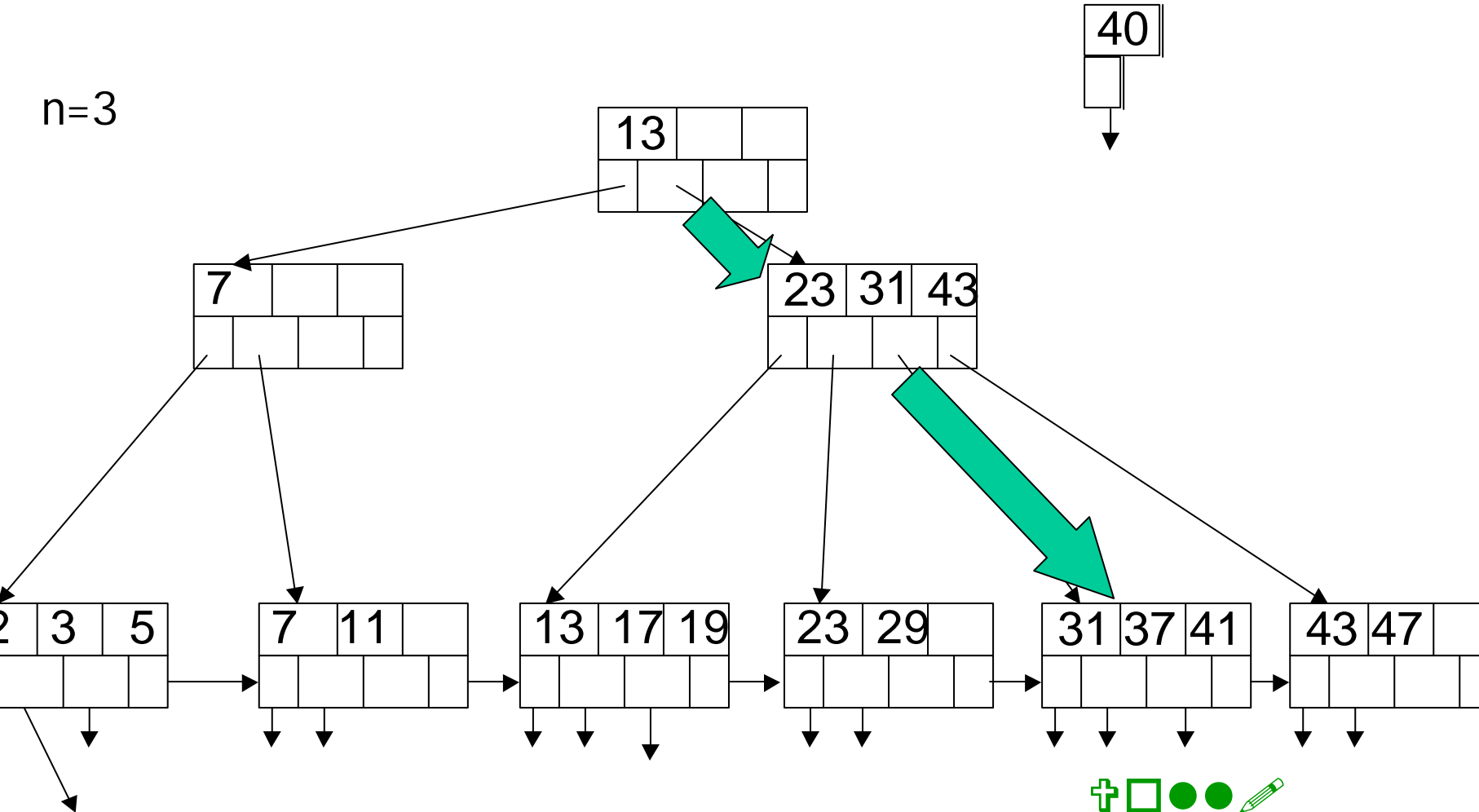
Teilen von Knoten

- 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.



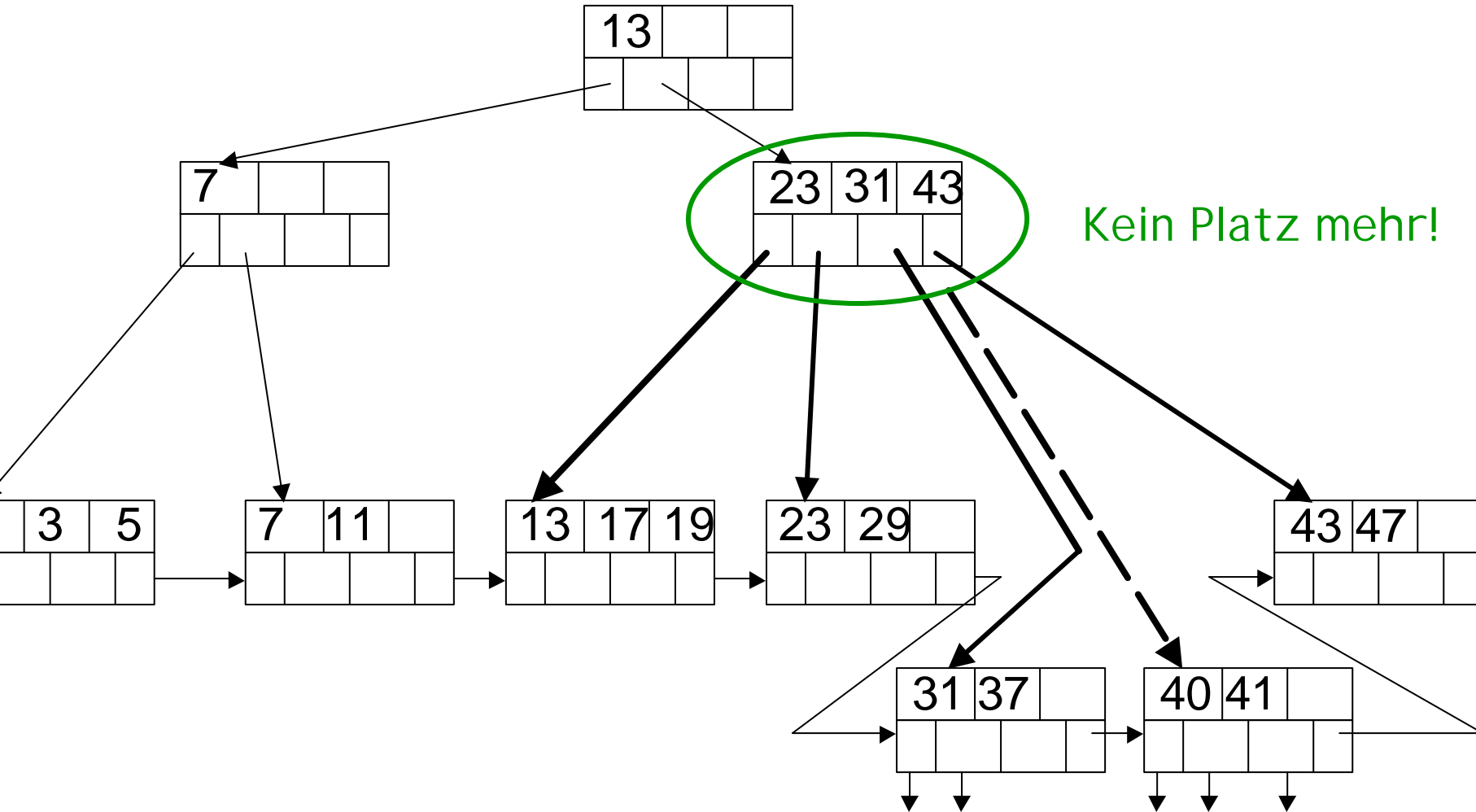
Beispiel

n=3



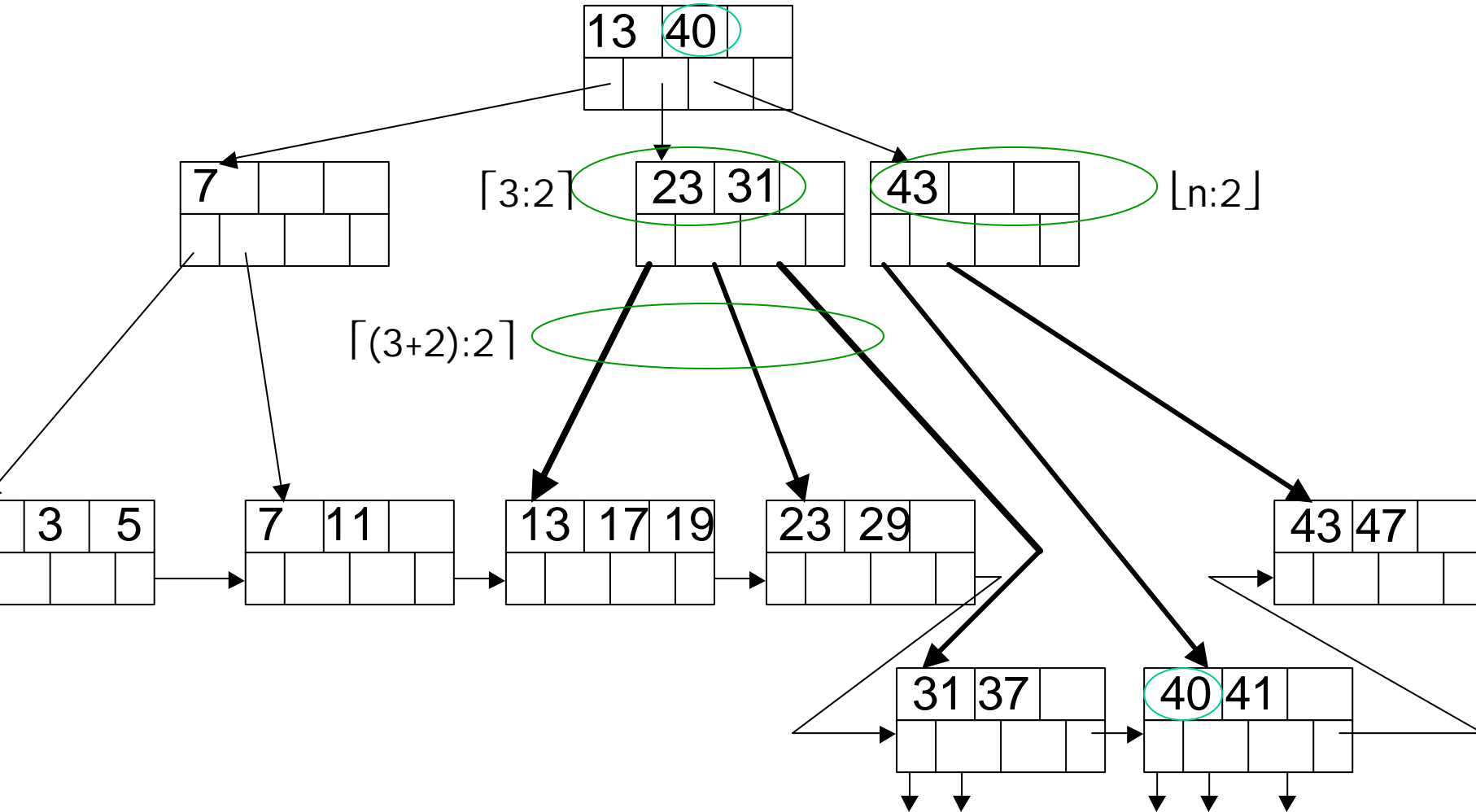


Knoten teilen





Nach oben propagieren





Eigenschaften der B-Trees

- Sehr wenige Plattenzugriffe:
 - Man liest von der Wurzel aus die passenden Knoten
⇒ 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
⇒ 2 Plattenzugriffe.
- Für geeignetes n (z.B. $n=10$) kommt Teilen oder Verbinden von Knoten selten vor, so dass die Reorganisation wenig kostet.

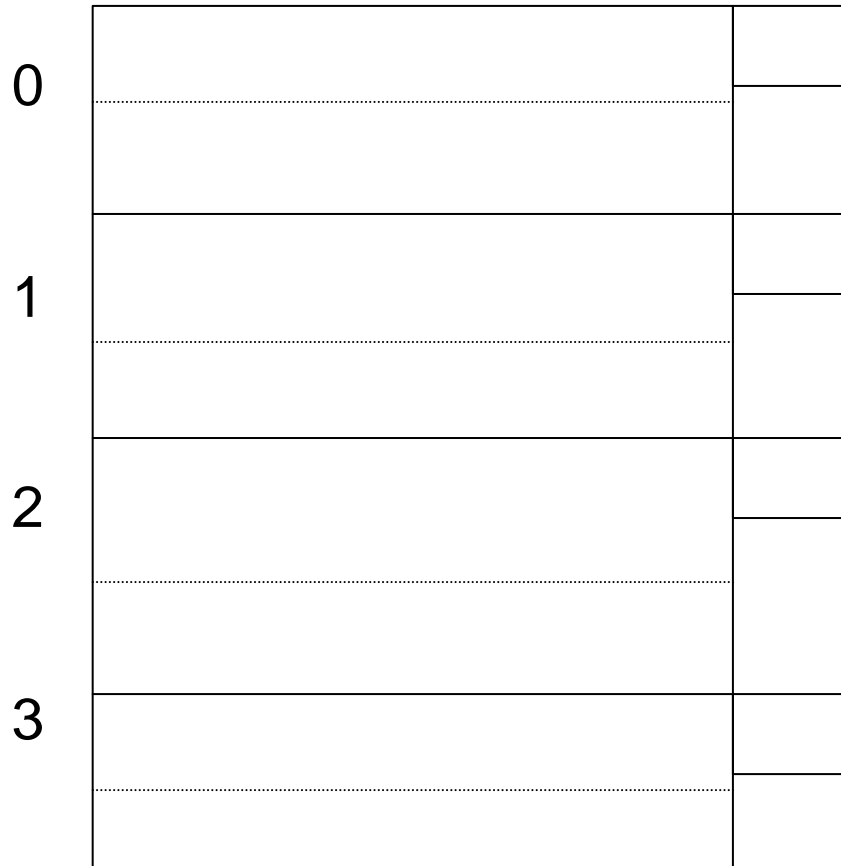


Hash-Tabellen

- 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.
- 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.
- Hash-Tabellen sind extrem effizient (meist nur 1 Plattenzugriff).



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 ($r=3$)

0	0000	
	1010	
1	1111	

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, \dots, 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 $h_1(v_1), \dots, h_n(v_n)$ werden konkateniert und ergeben so die k -bit der Gesamt-hash-Funktion.

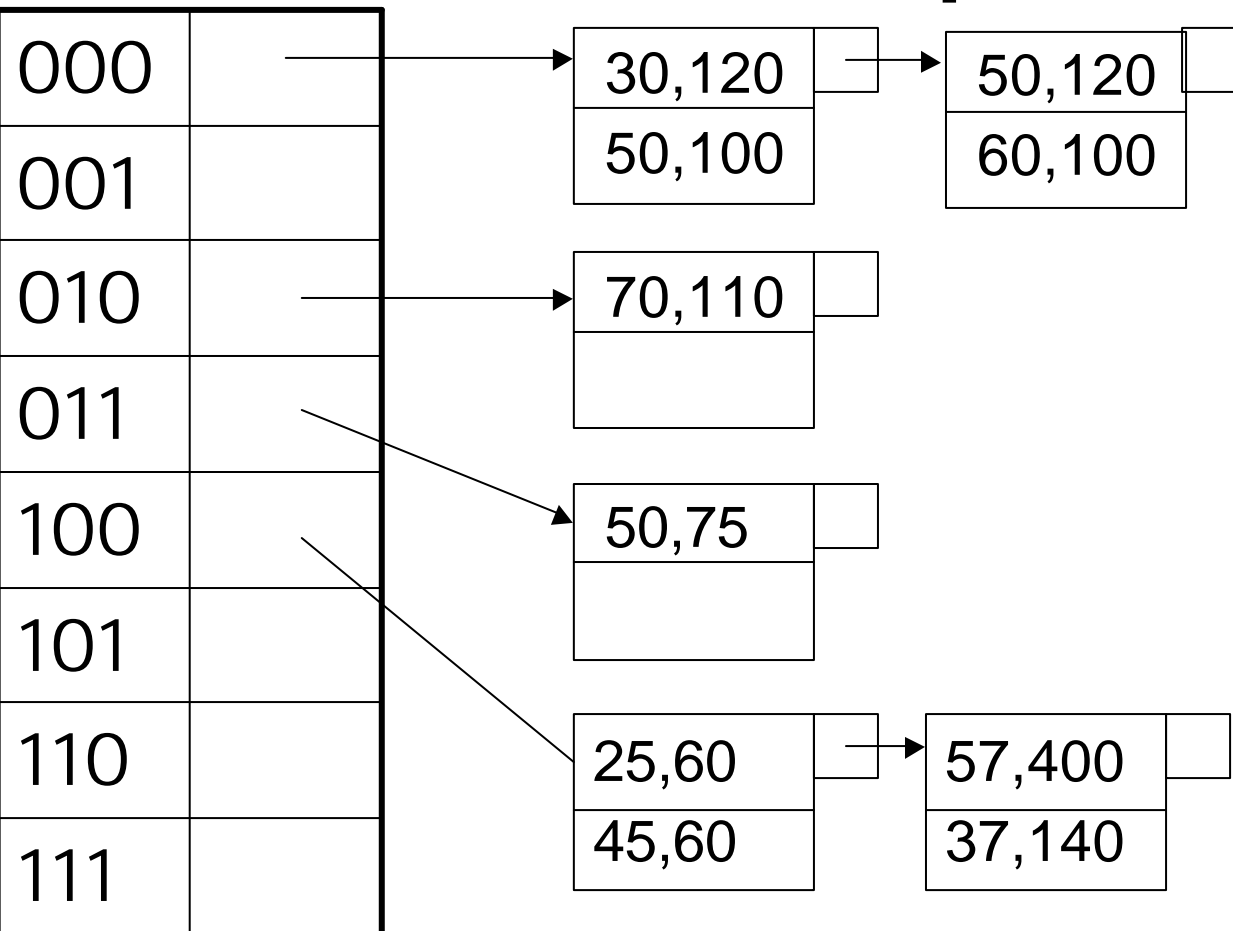


Beispiel

- Indexierung nach
 - Alter $h(a) = a \text{ modulo } 2$
Gerades Alter ergibt 0xy, ungerades Alter ergibt 1yz, (y,z sind bits).
 - Gehalt $h(g) = g:1000 \text{ modulo } 4$
Rest 1 ergibt x01.
 - Gesamt-hash-Wert ist xyz.
- $h(50,75000)$ ergibt 011.



Beispiel





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.