
Datenbanken

Prof. Dr. Ralf Möller

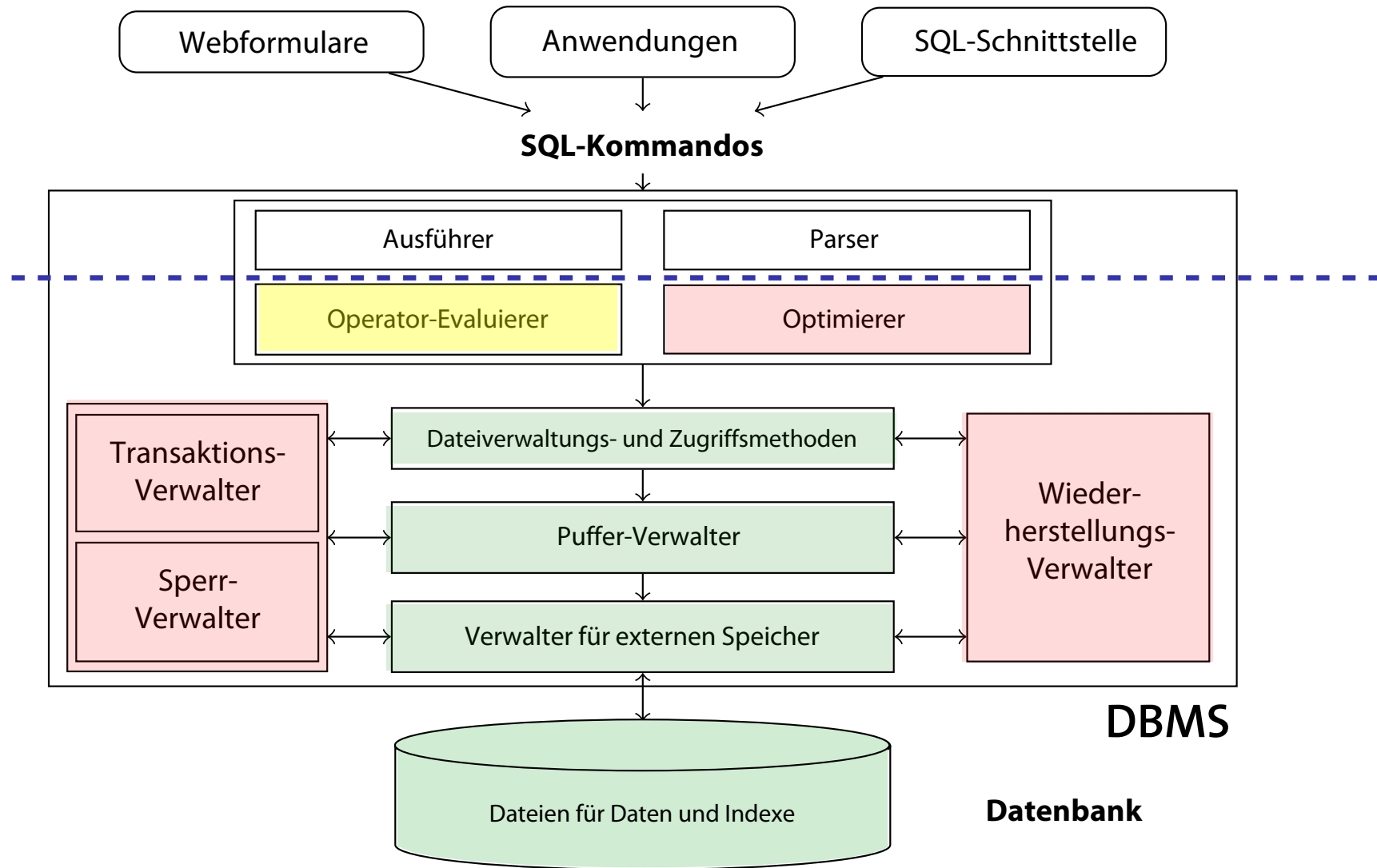
Universität zu Lübeck

Institut für Informationssysteme

Dennis Heinrich (Übungen)



Architektur eines DBMS



Danksagung

- Diese Vorlesung ist inspiriert von den Präsentationen zu dem Kurs:

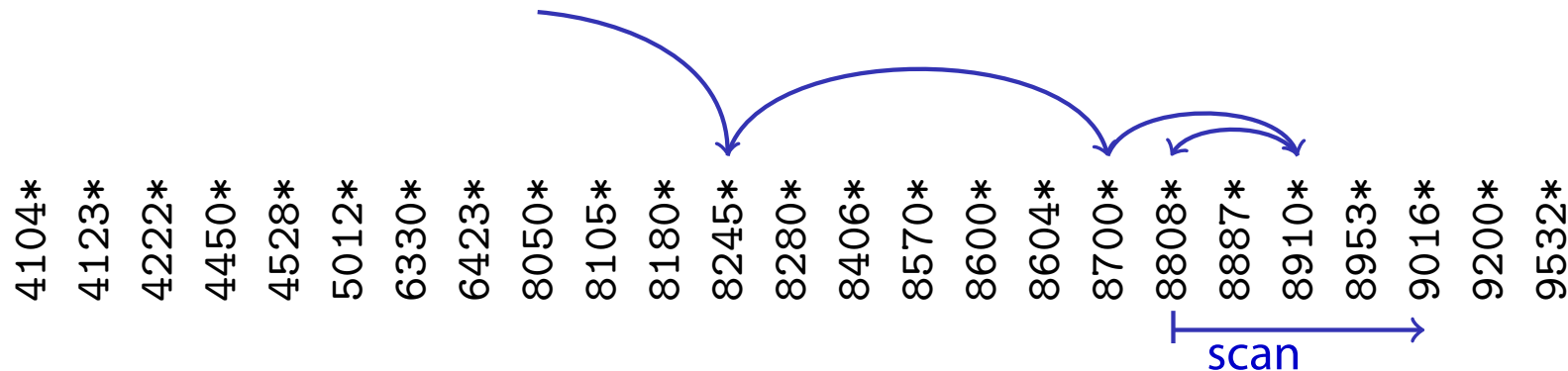
„Architecture and Implementation of Database Systems“
von Jens Teubner an der ETH Zürich

- Graphiken wurden mit Zustimmung des Autors aus diesem Kurs übernommen

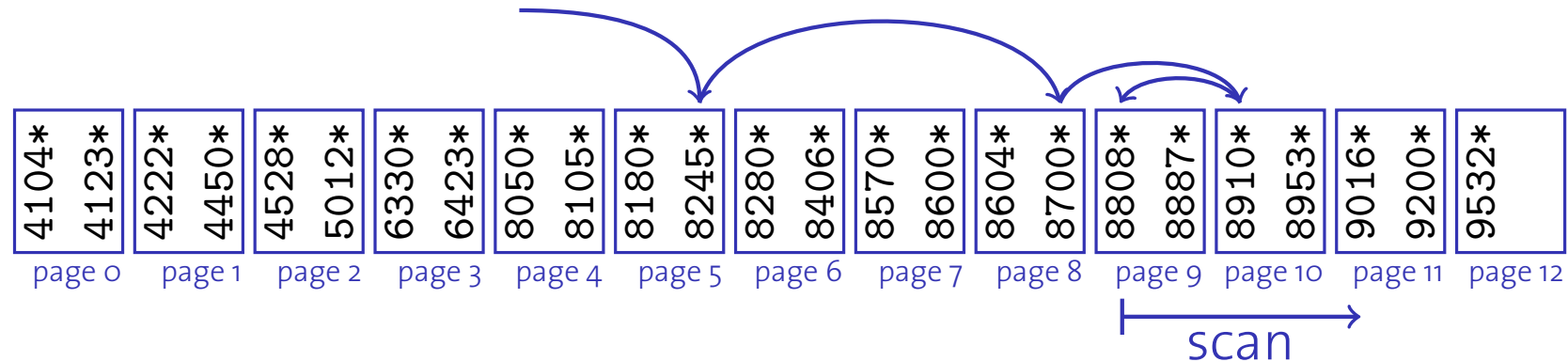
Effiziente Evaluierung einer Anfrage

```
SELECT *  
FROM CUSTOMERS  
WHERE ZIPCODE BETWEEN 8800 AND 8999
```

- Sortierung der Tabelle CUSTOMERS auf der Platte (nach ZIPCODE)
- Zur Evaluierung von Anfragen Verwendung von binärer Suche, um erstes Tupel zu finden, dann Scan solange $\text{ZIPCODE} < 8999$



Geordnete Dateien und binäre Suche



✓ Sequentieller Zugriff während der Scan-Phase

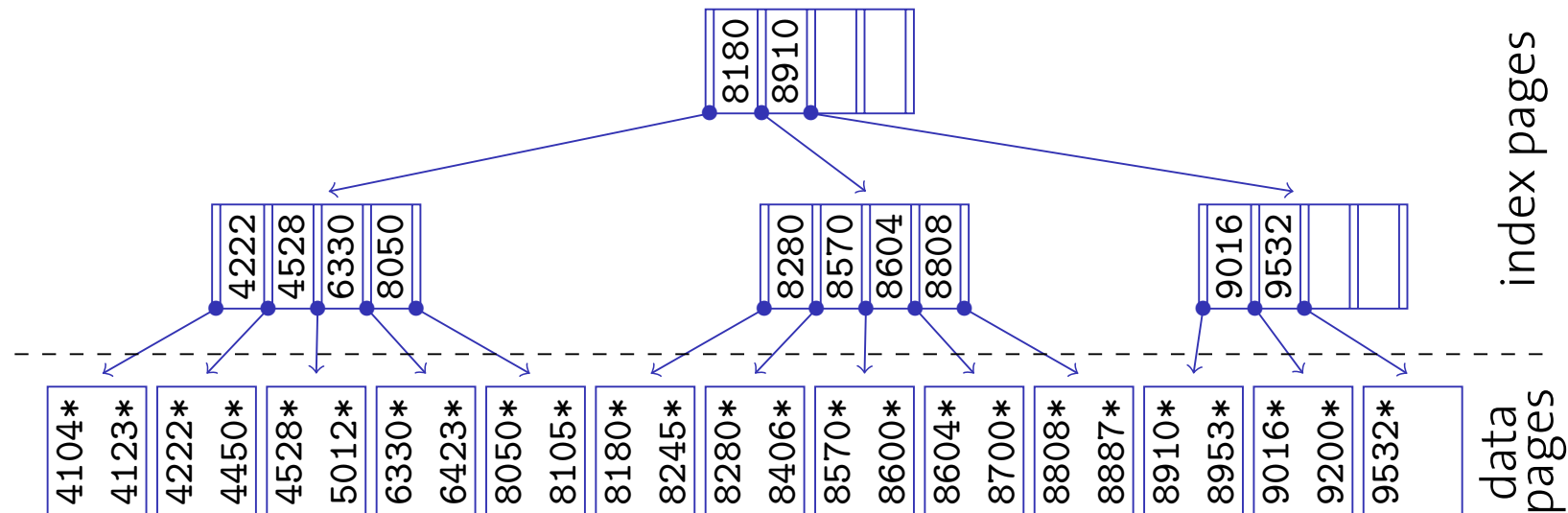
Es müssen $\log_2(\#Tupel)$ während der Such-Phase gelesen werden

✗ Für jeden Zugriff eine Seite!

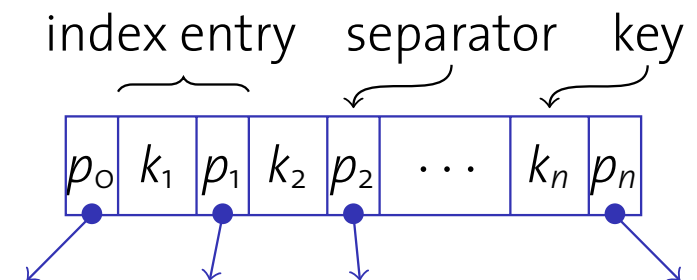
- Weite Sprünge sind die Idee der binären Suche
- Kein Prefetching möglich

ISAM: Indexed Sequential Access Method

Idee: Beschleunige die Suchphase durch sog. Index



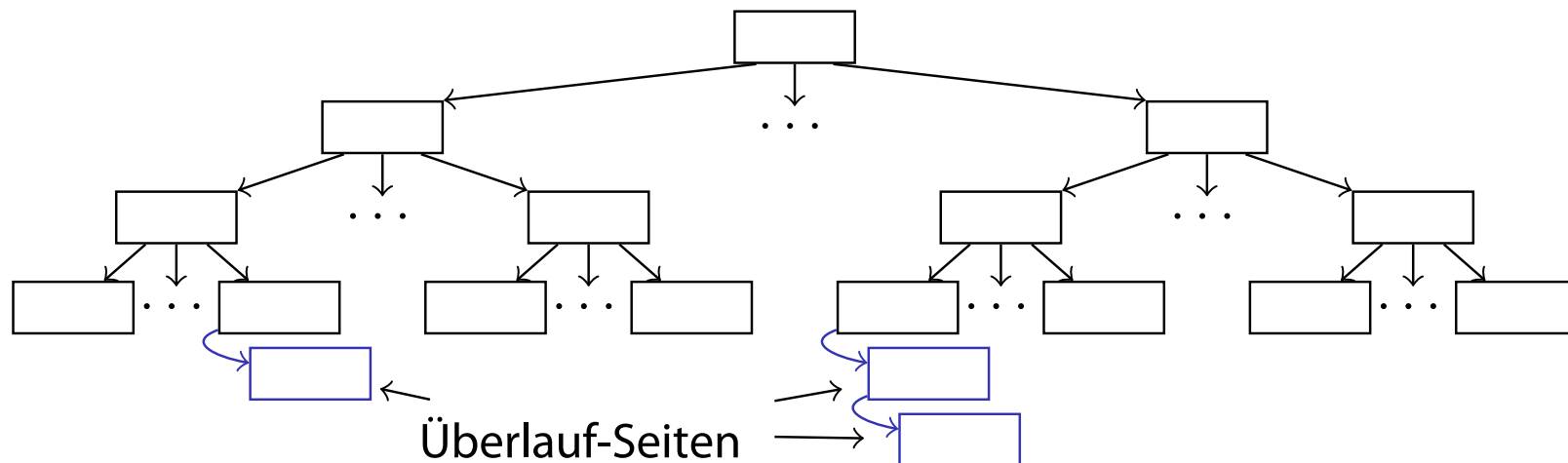
- Knoten von der Größe einer Seite
 - Hunderte Einträge pro Seite
 - Hohe Verzweigung, kleine Tiefe
- Suchaufwand: $\log_{\text{Verzweigung}}(\#Tupel)$



ISAM-Index: Aktualisierungsoperationen

ISAM-Indexte sind statisch

- **Löschen** einfach: Lösche Datensatz von Datenseite
- **Einfügen** von Daten aufwendig
 - Falls noch Platz auf Blattseite, füge Datensatz ein (z.B. nach einer vorherigen Löschung)
 - Sonst füge **Überlauf-Seite** ein (zerstört sequentielle Ordnung)
 - ISAM-Index **degeneriert**



Anmerkungen

- Das Vorsehen von Freiraum bei der Indexerzeugung reduziert das Einfügeproblem (typisch sind 20% Freiraum)
- Da Seiten statisch, keine Zugriffskoordination nötig
 - Zugriffskoordination (Sperrern) vermindert gleichzeitigen Zugriff (besonders nahe der Wurzel) für andere Anfragen
- ISAM ist nützlich für (relativ) statische Daten

B⁺-Bäume: Eine dynamische Indexstruktur

B⁺-Bäume von ISAM-Index abgeleitet, sind aber dynamisch

- Keine Überlauf-Ketten
- Balancierung wird aufrechterhalten
- Behandelt insert und delete angemessen

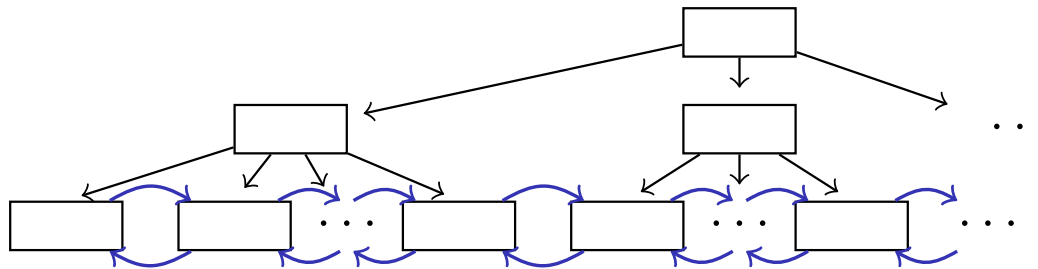
Minimale Besetzungsregel für B⁺-Baum-Knoten (außer der Wurzel): 50% (typisch sind 67%)

- Verzweigung nicht zu klein (Zugriff $O(\log n)$)
- Indexknotensuche nicht zu linear

B⁺-Bäume: Grundlagen

B⁺-Bäume ähnlich zu ISAM-Index, wobei

- Blattknoten üblicherweise nicht in seq. Ordnung
- Blätter zu doppelt verketteter Liste verbunden



- Blätter enthalten tatsächliche Daten (wie ISAM-Index) oder Referenzen (Rids) auf Datenseiten
 - Wir nehmen im Folgenden Letzteres an
- Jeder Knoten enthält zwischen d und $2d$ Einträge (d heißt Ordnung des Baumes, Wurzel ist Ausnahme)

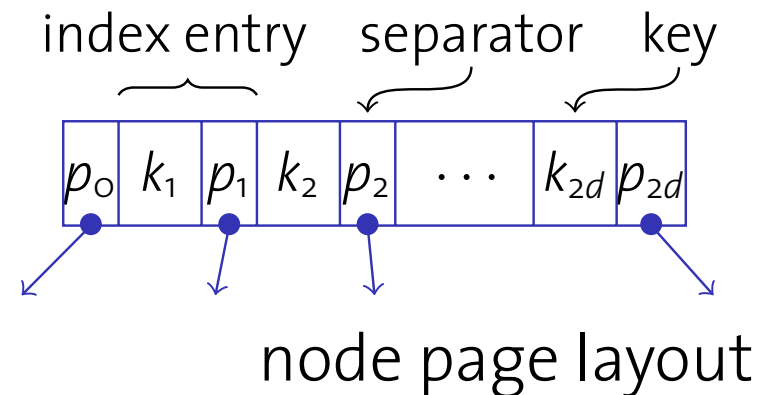
Suche im B⁺-Baum

```
1 Function: search (k)  
2 return tree_search (k, root);
```

```
1 Function: tree_search (k, node)  
2 if node is a leaf then  
3   | return node;  
4 switch k do  
5   | case  $k < k_1$   
6   |   | return tree_search (k,  $p_0$ );  
7   | case  $k_i \leq k < k_{i+1}$   
8   |   | return tree_search (k,  $p_i$ );  
9   | case  $k_{last} < k$   
10  |   | return tree_search (k,  $p_{last}$ );
```

$$i < last \leq 2d$$

- Funktionsaufruf **search(*k*)** bestimmt Blatt, das potentielle Treffer für eine Suche nach Elementen mit Schlüssel **k** enthält

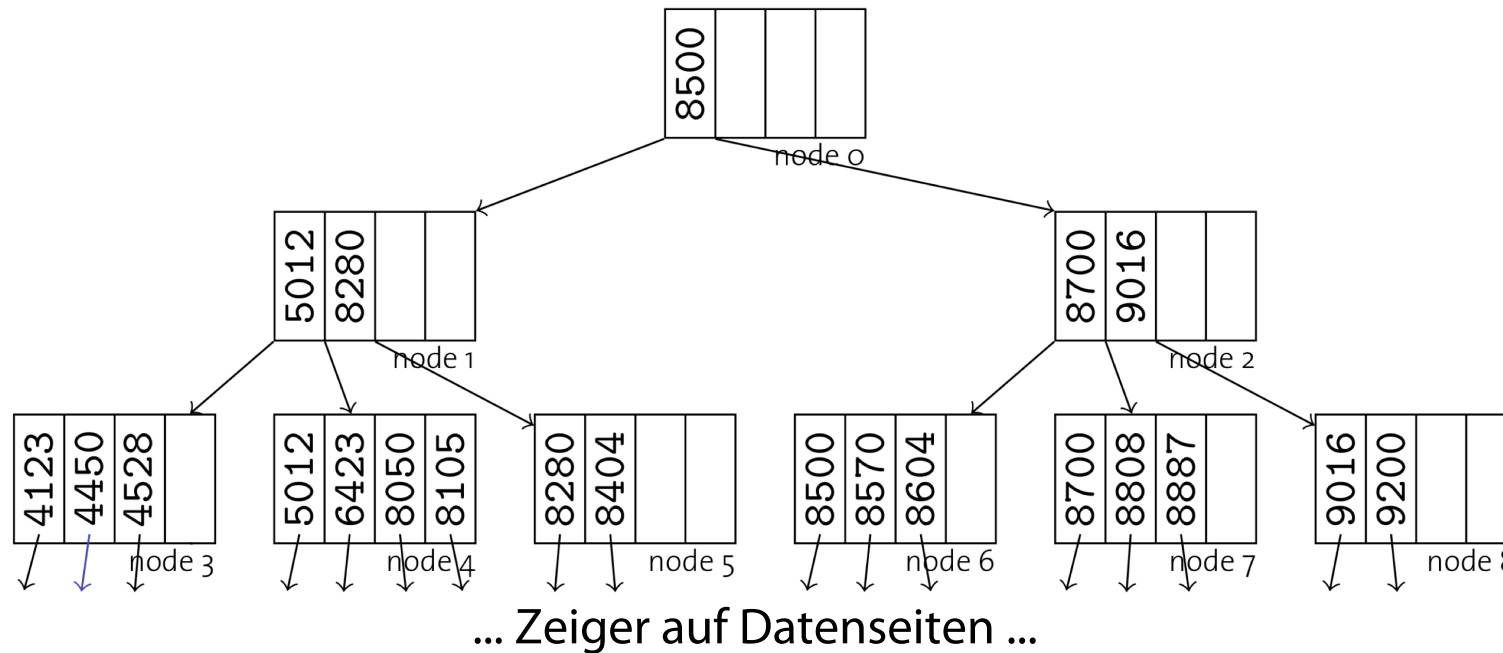


Insert: Überblick

- B⁺-Baum soll nach Einfügung **balanciert bleiben**
 - keine Überlauf-Seiten
- Algorithmus für `insert(k, p)` für Schlüsselwert `k` und Datenseite `p`
 1. Finde Blattseite `n`, in der Eintrag für `k` sein kann
 2. Falls `n` genug Platz hat (höchstens `2d-1` Einträge), füge Eintrag `<k, p>` in `n` ein
 3. Sonst muss Knoten `n` aufgeteilt werden in `n` und `n'` – weiterhin muss ein Separator in den Vater von `n` eingefügt werden

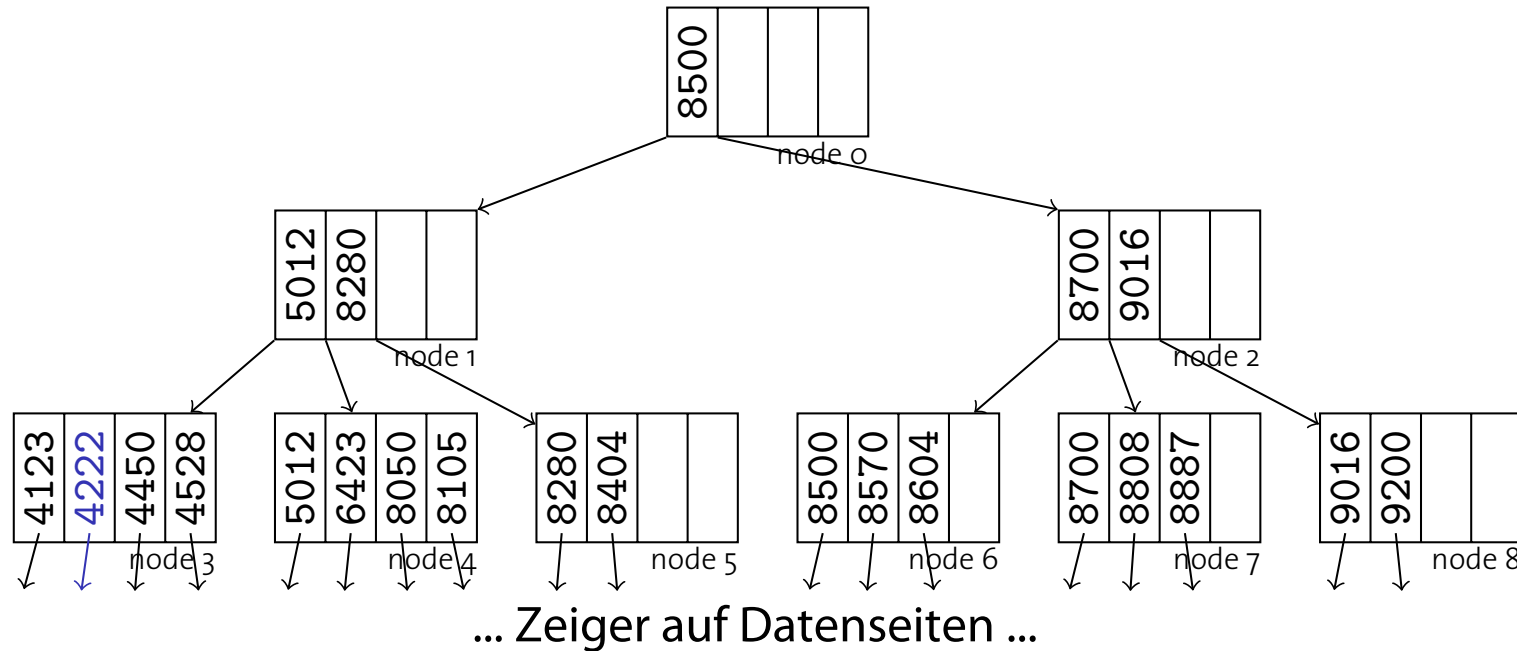
Die mögliche Aufspaltung erfolgt rekursiv nach oben, eventuell bis zur Wurzel (wodurch sich der Baum erhöht)

Insert: Beispiel ohne Aufspaltung



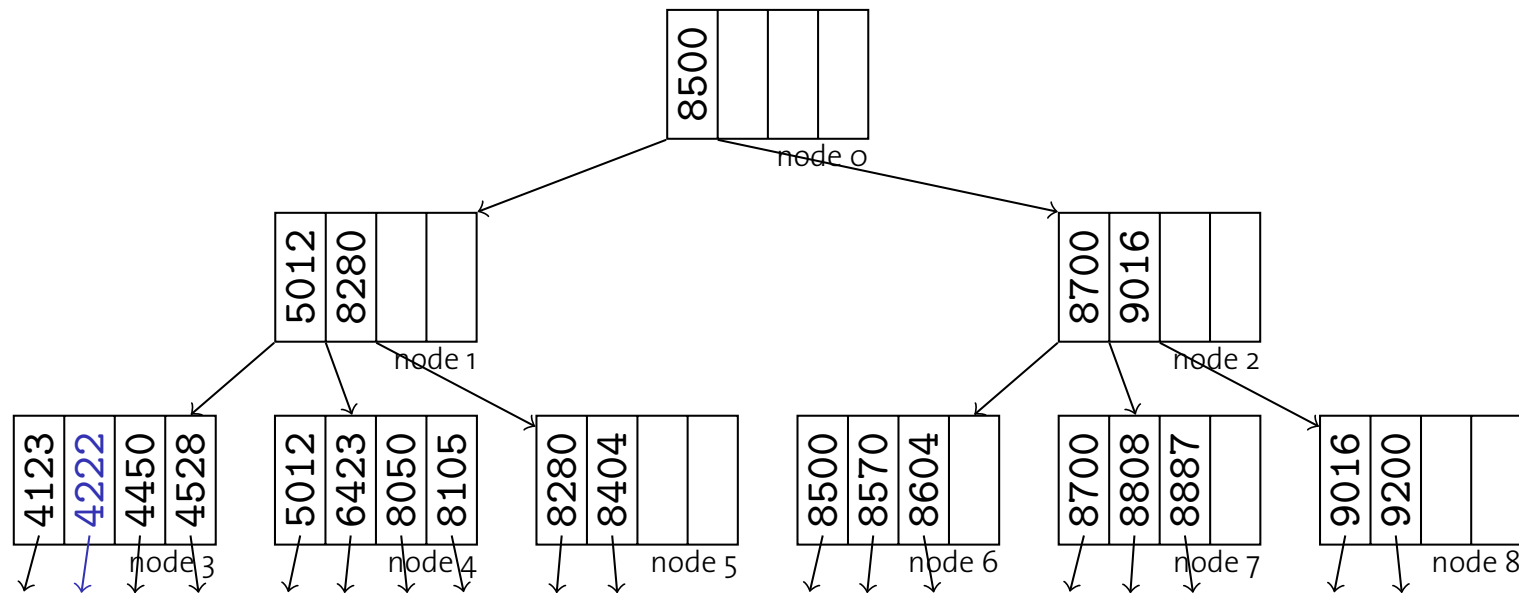
- Einfügung eines Eintrags mit Schlüssel **4222**
 - Es ist genug Platz in Knoten 3, einfach einfügen
 - Erhalte **Sortierung innerhalb der Knoten**

Insert: Beispiel ohne Aufspaltung



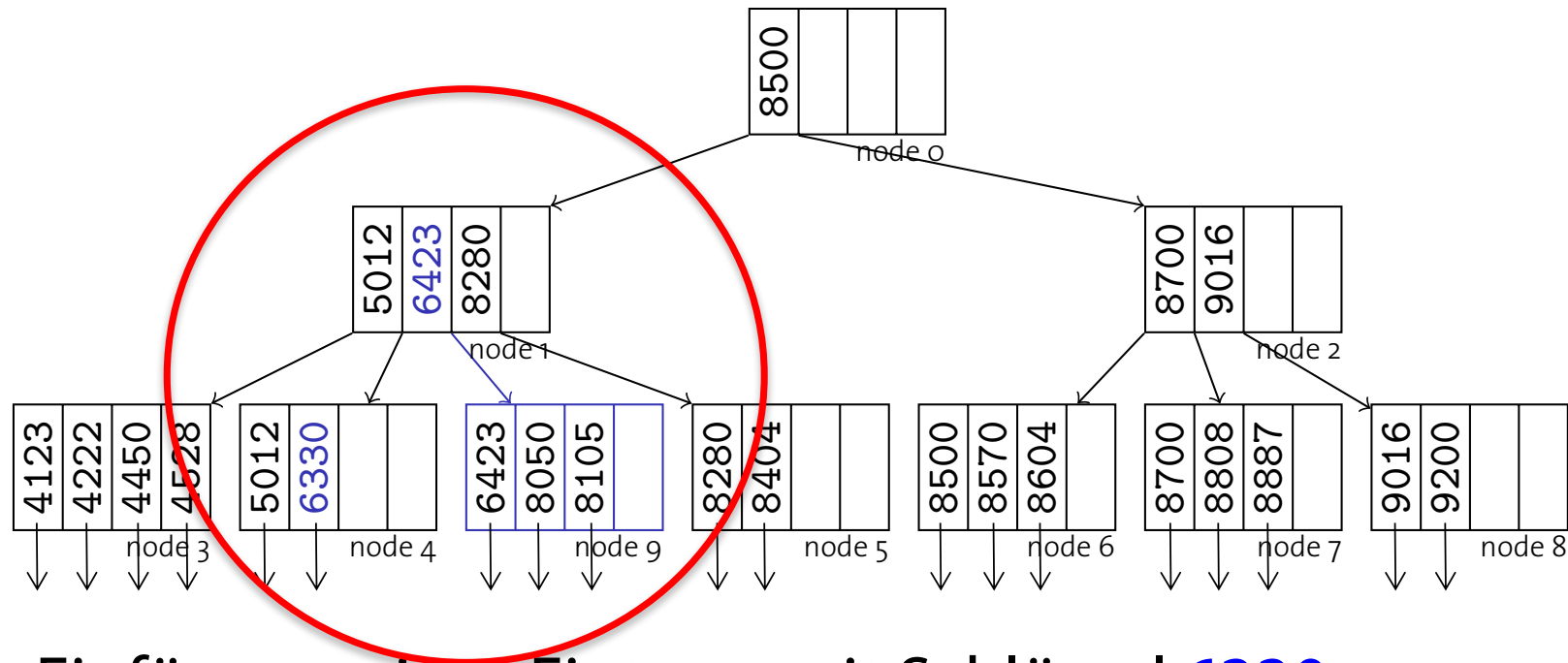
- Einfügung eines Eintrags mit Schlüssel **4222**
 - Es ist genug Platz in Knoten 3, einfach einfügen
 - Erhalte **Sortierung innerhalb der Knoten**

Insert: Beispiel mit Aufspaltung

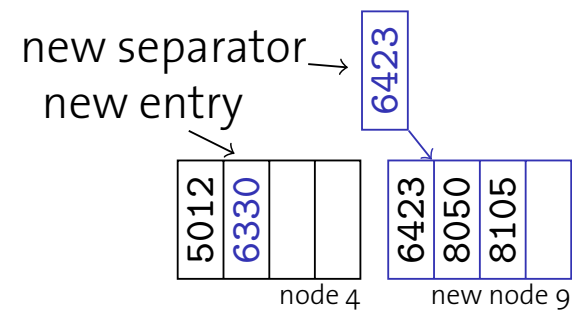


- Einfügung eines Eintrags mit Schlüssel **6330**
 - Knoten 4 aufgespalten
 - Neuer Separator in Knoten 1

Insert: Beispiel mit Aufspaltung

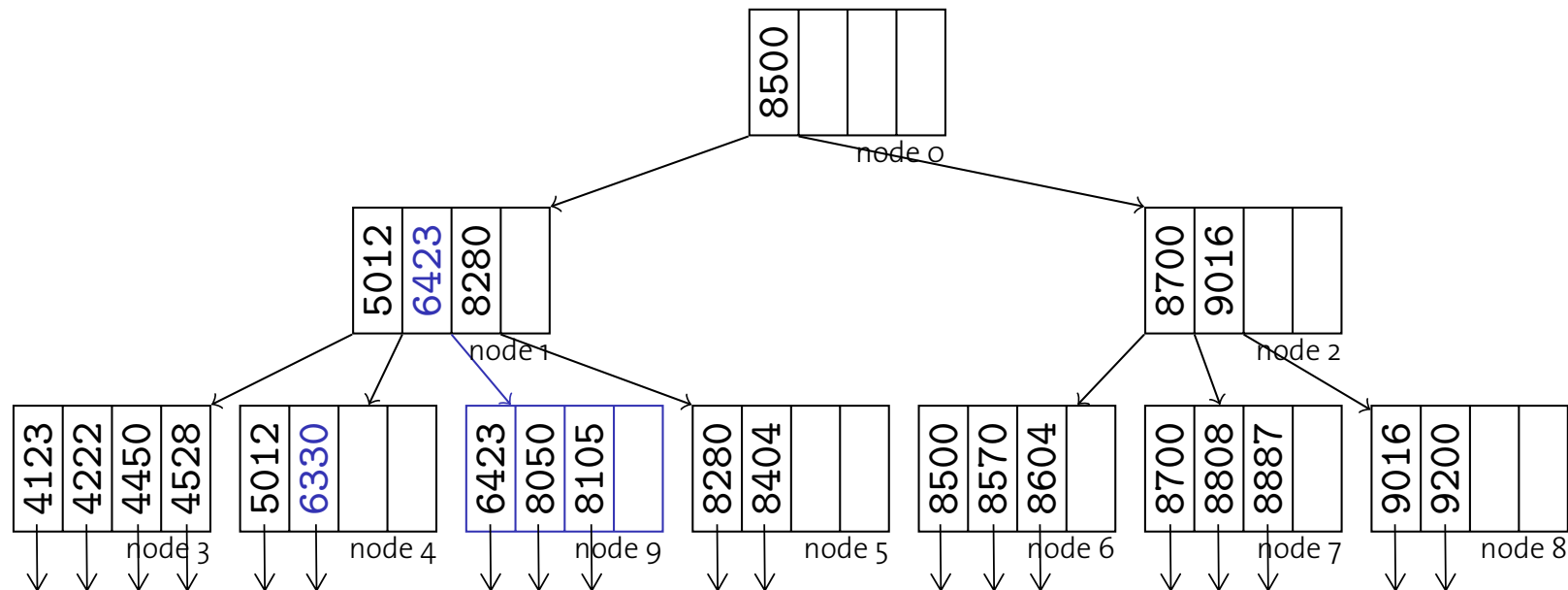


- Einfügung eines Eintrags mit Schlüssel **6330**
 - Knoten 4 aufgespalten
 - Neuer Separator in Knoten 1

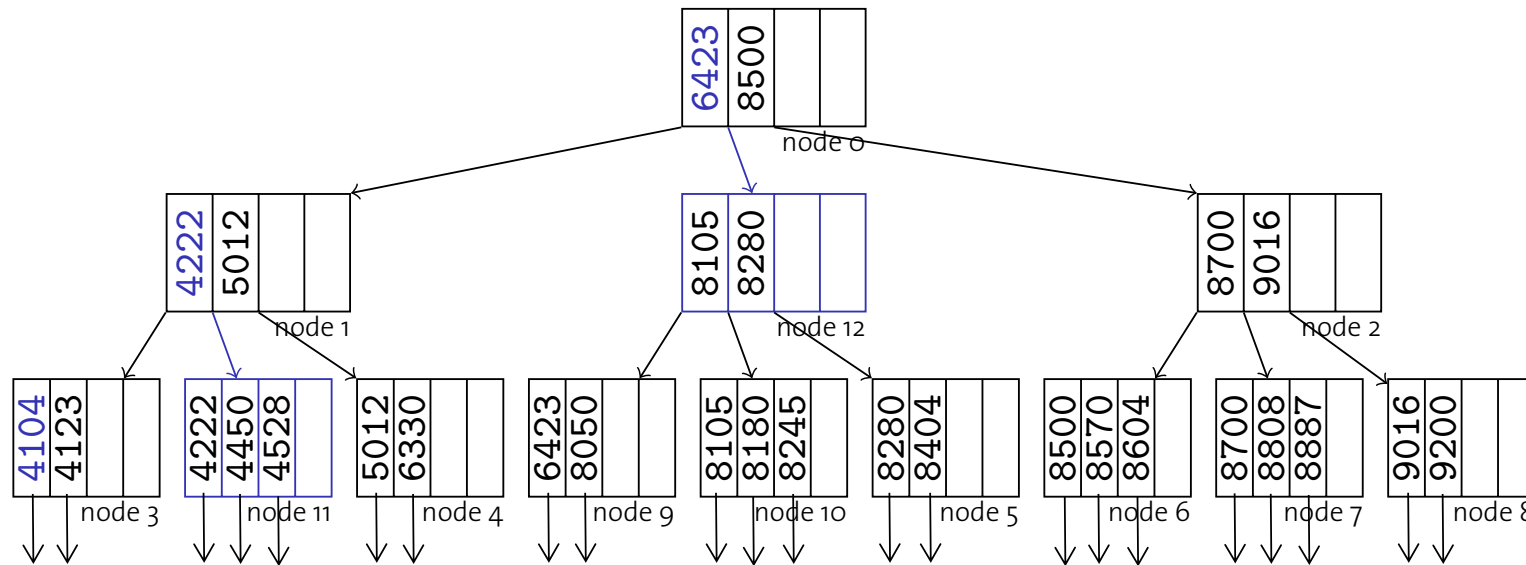


Insert: Beispiel mit Aufspaltung

- Einfügung von 8180, 8245...



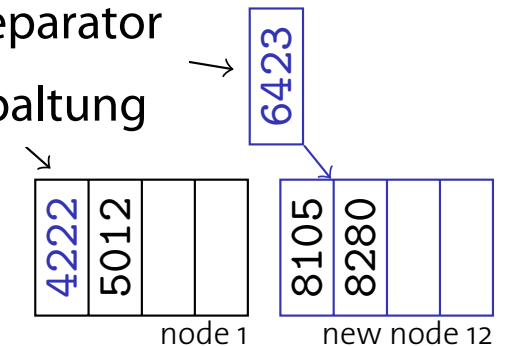
Insert: Beispiele mit Aufspaltung innerer Knoten



Nach 8180, 8245, füge 4104 ein

- Aufspaltung von Knoten 3 und 9
- Knoten 1 läuft über → Aufspaltung
- Neuer Separator für Wurzel

Neuer Separator
aus Knotenaufspaltung

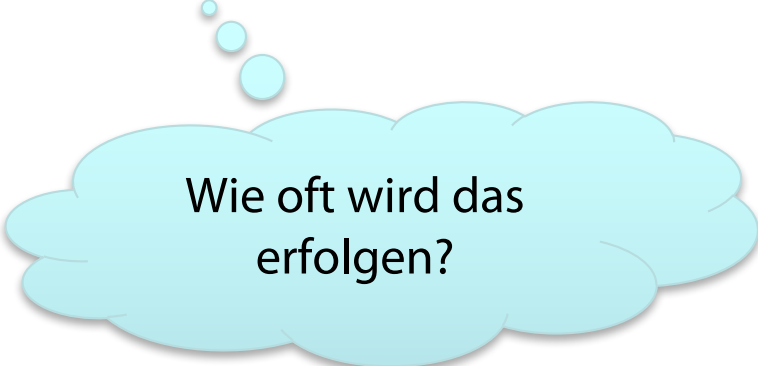


Separatorschlüssel aus inneren Knoten
können sich verschieben

Warum?

Insert: Aufspaltung eines inneren Knotens

- Aufspaltung beginnt auf Blattebene und verläuft nach oben solange Indexknoten vollständig belegt
- Schließlich kann die Wurzel aufgespalten werden
 - Aufspaltung wie bei inneren Knoten
 - Separator für einen neuen Wurzelknoten verwenden
- Neuer Wurzelknoten mit Füllgrad $< 50\%$ möglich
- Erhöhung nur bei Einfügung einer neuen Wurzel



Wie oft wird das
erfolgen?

Zusammenfassung: Algorithmus tree_insert

```

1 Function: tree_insert ( $k, rid, node$ )
2 if  $node$  is a leaf then
3   | return leaf_insert ( $k, rid, node$ );
4 else
5   | switch  $k$  do
6     | case  $k < k_1$ 
7       | |  $\langle sep, ptr \rangle \leftarrow \text{tree\_insert}(k, rid, p_o);$ 
8     | case  $k_i \leq k < k_{i+1}$ 
9       | |  $\langle sep, ptr \rangle \leftarrow \text{tree\_insert}(k, rid, p_i);$ 
10    | case  $k_{last} < k$ 
11      | |  $\langle sep, ptr \rangle \leftarrow \text{tree\_insert}(k, rid, p_{last});$ 
12    | if  $sep$  is null then
13      | | return  $\langle \text{null}, \text{null} \rangle;$ 
14    | else
15      | | return split ( $sep, ptr, node$ );

```

} see tree_search ()

$i < last \leq 2d$

Algorithmus leaf_insert

```
1 Function: leaf_insert ( $k, rid, node$ )
2 if another entry fits into  $node$  then
3   |   insert  $\langle k, rid \rangle$  into  $node$  ;
4   |   return  $\langle \text{null}, \text{null} \rangle$ ;
5 else
6   |   allocate new leaf page  $p$  ;
7   |   take  $\{ \langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} :=$  entries from  $node \cup \{ \langle k, ptr \rangle \}$ 
8   |   |   leave entries  $\langle k_1^+, p_1^+ \rangle, \dots, \langle k_d^+, p_d^+ \rangle$  in  $node$  ;
9   |   |   move entries  $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$  to  $p$  ;
10  |   return  $\langle k_{d+1}^+, p \rangle$ ;
```

Algorithmus split

```
1 Function: split ( $k, ptr, node$ )
2 if another entry fits into  $node$  then
3   insert  $\langle k, ptr \rangle$  into  $node$  ;
4   return  $\langle \text{null}, \text{null} \rangle$ ;
5 else
6   allocate new node  $p$ ;
7   take  $\{ \langle k_1^+, p_1^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle \} :=$  entries from  $node \cup \{ \langle k, ptr \rangle \}$ 
8   leave entries  $\langle k_1^+, p_1^+ \rangle, \dots, \langle k_d^+, p_d^+ \rangle$  in  $node$  ;
9   move entries  $\langle k_{d+2}^+, p_{d+2}^+ \rangle, \dots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$  to  $p$  ;
10  set  $p_0 \leftarrow p_{d+1}^+$  in  $node$ ;
11  return  $\langle k_{d+1}^+, p \rangle$ ;
```

Der erste Zeiger p_0 in der neuen Seite p wird auf p_{d+1} gesetzt. Dieser Zeiger bildet die Trennstelle, kommt demnach nicht mehr auf der alten Seite $node$ vor.

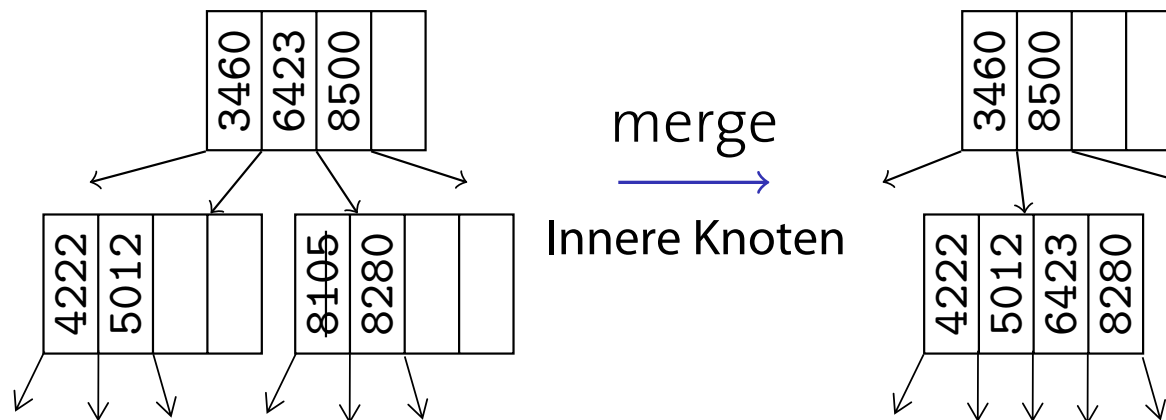
Algorithmus insert

```
1 Function: insert ( $k, rid$ )
2  $\langle key, ptr \rangle \leftarrow \text{tree\_insert}(k, rid, root);$  // root contains the root of the index tree
3 if  $key$  is not null then
4   allocate new root page  $r$ ;
5   populate  $r$  with
6      $p_0 \leftarrow root$ ;
7      $k_1 \leftarrow key$ ;
8      $p_1 \leftarrow ptr$ ;
9    $root \leftarrow r$ ;
```

- insert(k, rid) wird von außen aufgerufen
- Blattknoten enthalten Rids, innere Knoten enthalten Zeiger auf andere B⁺-Baum-Knoten

Löschung

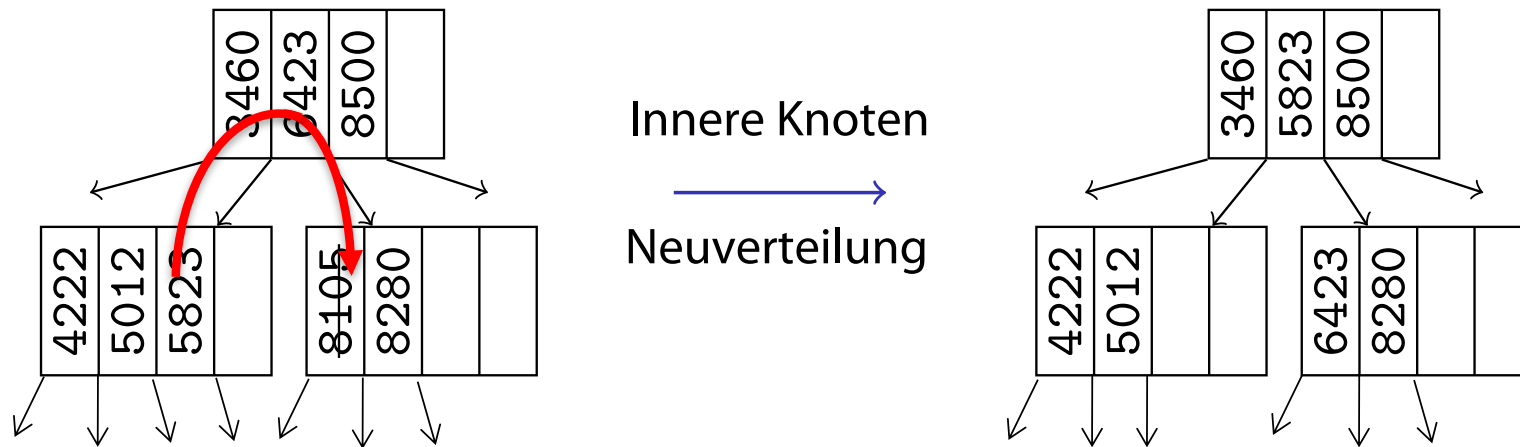
- Falls Knoten genügend gefüllt (mindestens $d+1$ Einträge), Eintrag einfach löschen
 - Hinterher können innere Knoten Schlüssel enthalten, die zu Einträgen gehören, die nicht mehr existieren.
 - Das ist OK
- Sonst verschmelze Knoten wegen Unterfüllung



- Ziehe Separator in den verschmolzenen Knoten

Löschung

- Leider ist die Situation nicht immer so einfach



- Verschmelzung nur, wenn Nachbarknoten zu 50% voll
- Sonst muss Neuverteilung erfolgen
 - Rotiere Eintrag über den Elternknoten

B⁺-Bäume in realen Systemen

- Implementierungen verzichten auf die Kosten der Verschmelzung und der Neuverteilung und weichen die Regel der Minimumbelegung auf
- Beispiel: IBM DB2 UDB
 - MINPCTUSED als Parameter zur Steuerung der Blattknotenverschmelzung (Online-Indexreorganisation)
 - Innere Knoten werden niemals verschmolzen (nur bei Reorganisation der gesamten Tabelle)
- Zur Verbesserung der Nebenläufigkeit evtl. nur Markierung von Knoten als gelöscht (keine aufwendige Neuverzeigerung)

Was wird in den Blättern gespeichert?

Drei Alternativen

1. Vollständiger Datensatz k^*
(ein solcher Index heißt geclustert, siehe unten)
2. Ein Paar $\langle k, \text{rid} \rangle$, wobei rid (record ID) ein Zeiger auf einen Datensatz darstellt
3. Ein Paar $\langle k, \{\text{rid}_1, \text{rid}_2, \dots\} \rangle$, wobei alle Rids den Suchschlüssel k haben

Varianten 2. und 3. bedingen, dass Rids stabil sein müssen, also nicht (einfach) verschoben werden können

Alternative 2 scheint am meisten verwendet zu werden

Erzeugung von Indexstrukturen in SQL

Implizite Indexe

Indexe automatisch erzeugt für Primärschlüssel und Unique-Integritätsbedingungen

Explizite Indexe

- Einfache Indexe:

```
CREATE INDEX name  
ON table_name(attr) ;
```

Beispiel

```
CREATE INDEX ZipcodeIndex  
ON CUSTOMERS (ZIPCODE) ;
```

```
SELECT *  
FROM CUSTOMERS  
WHERE ZIPCODE BETWEEN 8800 AND 8999 ;
```

- Zusammengesetzte Indexe (Verbundindexe):

```
CREATE INDEX name  
ON table_name(attr1, attr2, ..., attrn) ;
```

Indexe mit zusammengesetzten Schlüsseln

B⁺-Bäume können verwendet werden, um Dinge mit einer definierten **totalen Ordnung** zu indizieren (im Prinzip¹)

- Integer, Zeichenketten, Datumsangaben, ...,
- und auch eine Hintereinandersetzung davon (basierend auf einer lexikographischen Ordnung)

Beispiel:

```
CREATE INDEX idx_lastname_firstname  
ON CUSTOMERS (LASTNAME, FIRSTNAME);
```

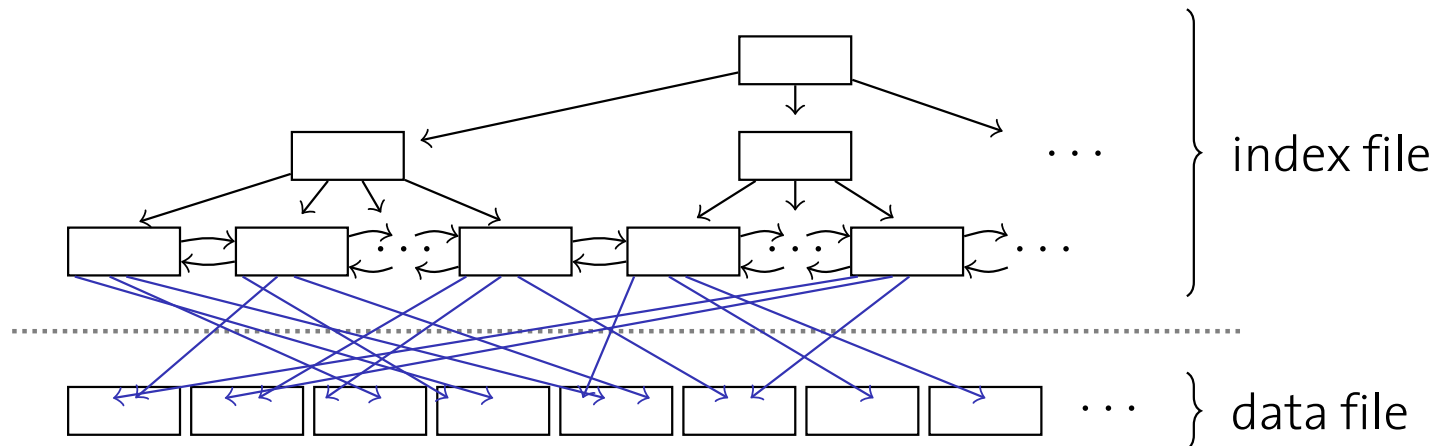
Eine weitere Anwendung sind **partitionierte B⁺-Bäume**

- Führende artifizielle Indexattribute partitionieren den B⁺-Baum horizontal (z.B. zur verteilten Verarbeitung)

¹ In einigen Implementierungen können lange Zeichenketten nicht als Index verwendet werden

B⁺-Bäume und Sortierung

Eine typische Situation nach Alternative 2 sieht so aus:

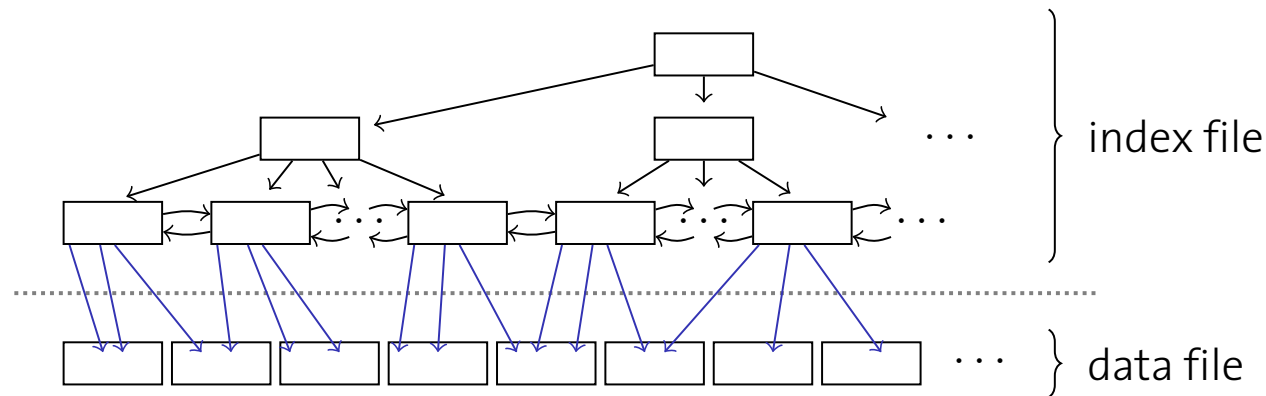


Was passiert, wenn man Folgendes ausführt?

```
SELECT * FROM CUSOTMERS ORDER BY ZIPCODE;
```

Geclusterte B⁺-Bäume

Wenn die Datei mit den Datensätzen sortiert und sequentiell gespeichert ist, erfolgt der Zugriff schneller



Ein so organisierter Index heißt geclusterter Index

- Sequentieller Zugriff während der Scan-Phase
- Besonders für Bereichsanfragen geeignet

Warum macht man
Indexe nicht immer
geclustert?

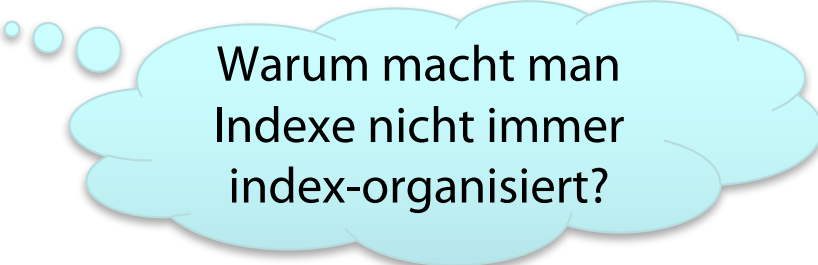
Index-organisierte Tabellen

Alternative 1 von oben ist ein Spezialfall eines geclusterten Index

- Indexdatei = Datensatz-Datei
- Eine solche Datei nennt man index-organisiert

Oracle:

```
CREATE TABLE (...  
                ...,  
                PRIMARY KEY (...))  
ORGANIZATION INDEX;
```

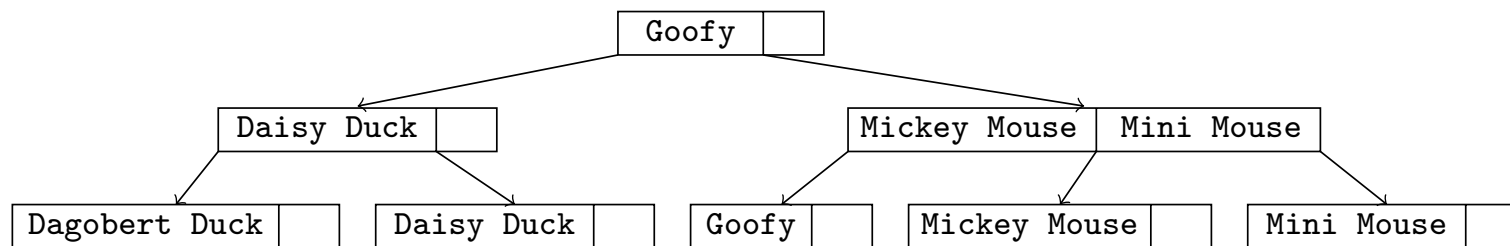


Warum macht man
Indexe nicht immer
index-organisiert?

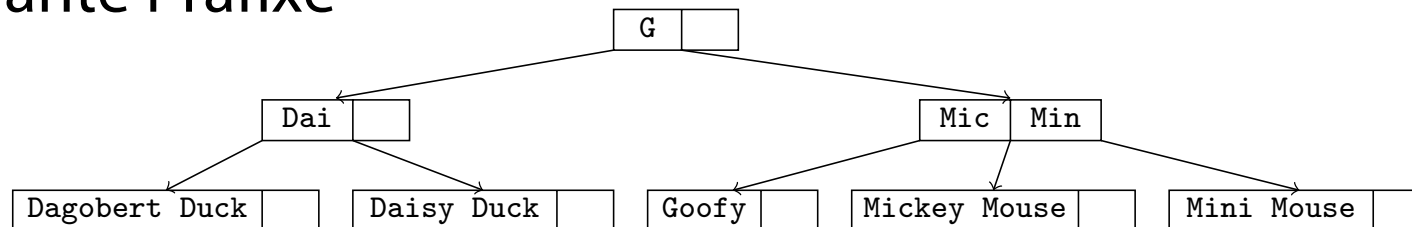
Suffix-Abschneidung

B⁺-Baum-Verzweigung proportional zur Anzahl der Einträge pro Seite, also umgekehrt proportional zur Schlüsselgröße

- Ziel: Schlüsselgröße verringern
(insb. relevant bei Zeichenketten variabler Länge)



Suffix-Abschneidung: Beschränkung der Separatoren auf relevante Präfixe



Separatoren benötigen Datenwerte nicht

Präfixabschneidung

Häufig treten Zeichenketten mit gleichem Präfix auf

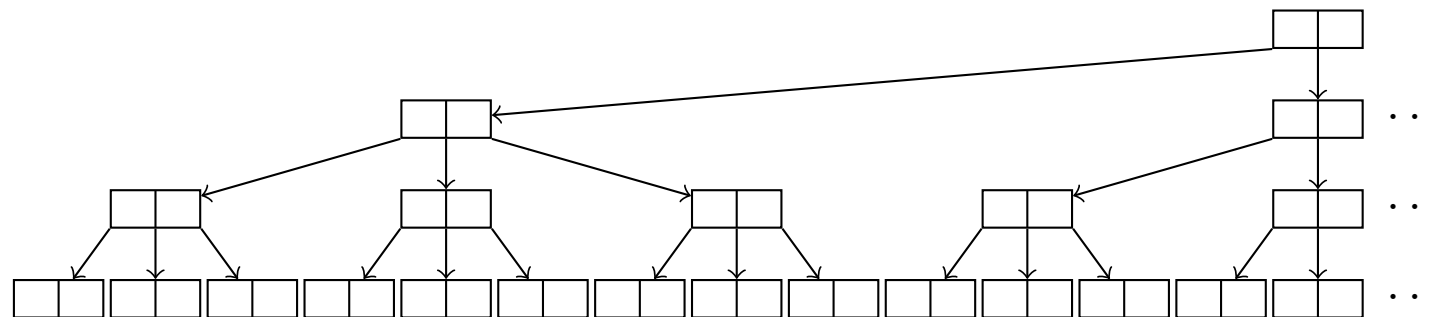


- Speichere gemeinsamen Präfix nur einmal (z.B. als k_0)
- Schlüssel sind nun stark diskriminierend

Außerkräftsetzen der 50%-Füllungsregel kann Effektivität der Präfixabschneidung verbessern

Bulk-Loading von B⁺-Bäumen

Aufbau eines B⁺-Baums ist einfach bei sortierter Eingabe



- Aufbau des B⁺-Baumes von links nach rechts möglich
- Eventuell mit Freiraum für Updates

B, B⁺, B^{*}, ...

Bisher B⁺-Bäume diskutiert

Ursprünglicher Vorschlag von Bayer und McCreight
enthielt sog. B-Bäume

- Innere Knoten enthalten auch Datensätze

Es gibt auch B^{*}-Bäume

- Fülle innere Knoten zu 2/3 statt nur zur 1/2
- Umverteilung beim Einfügen
(bei zwei vollen Knoten auf drei Knoten umverteilen)

B-Baum meint irgendeine dieser Formen, meist werden
in realen DBs die B⁺-Bäume implementiert

B⁺-Bäume auch außerhalb von DBs verwendet

Indexe: Zusammenfassung

- Zugriff auf Daten von $O(n)$ ungefähr auf $O(\log n)$
- Kosten der Indexierung aber nicht zu vernachlässigen
 - Nicht bei „kleinen“ Tabellen
 - Nicht bei häufigen Update- oder Insert-Anweisungen
 - Nicht bei Spalten mit vielen Null-Werten
- Standardisierung nicht gegeben
- Beispiel: MYSQL (Ausschnitt)

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name [index_type]  
ON tbl_name (index_col_name,...) [index_type]
```

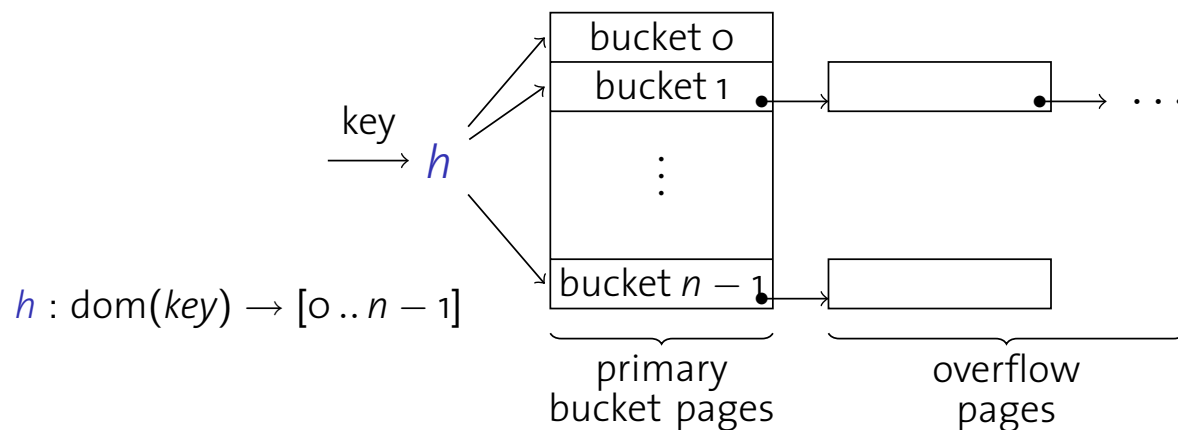
```
index_col_name:  
col_name [(length)] [ASC | DESC]
```

```
index_type:  
USING {BTREE | HASH}
```

Hash-basierte Indexierung

B⁺-Bäume dominieren in Datenbanken

Eine Alternative ist die hash-basierte Indexierung



- Hash-Indexe eignen sich nur für Gleichheitsprädikate
- Insbesondere für (lange) Zeichenketten und Verbundindexe
- Statt Kollisionslisten: **Lineares** Sondieren, o.a. Techniken

Statisches Hashing

- Hashfunktion $h(x) = x \bmod 3$

0	
1	(27550, 'Schopenhauer', 6)
2	(24002, 'Xenokrates', 18)
	(25403, 'Jonas', 12)

- Kollisionsbehandlung

0		
1	(27550, 'Schopenhauer', 6)	
	(26830, 'Aristoxenos', 8)	
2	(24002, 'Xenokrates', 18)	• →
	(25403, 'Jonas', 12)	

(26120, 'Fichte', 10)	• → ...
(28106, 'Carnap', 3)	

⇒ ineffizient bei nicht vorhersehbarer Datenmenge

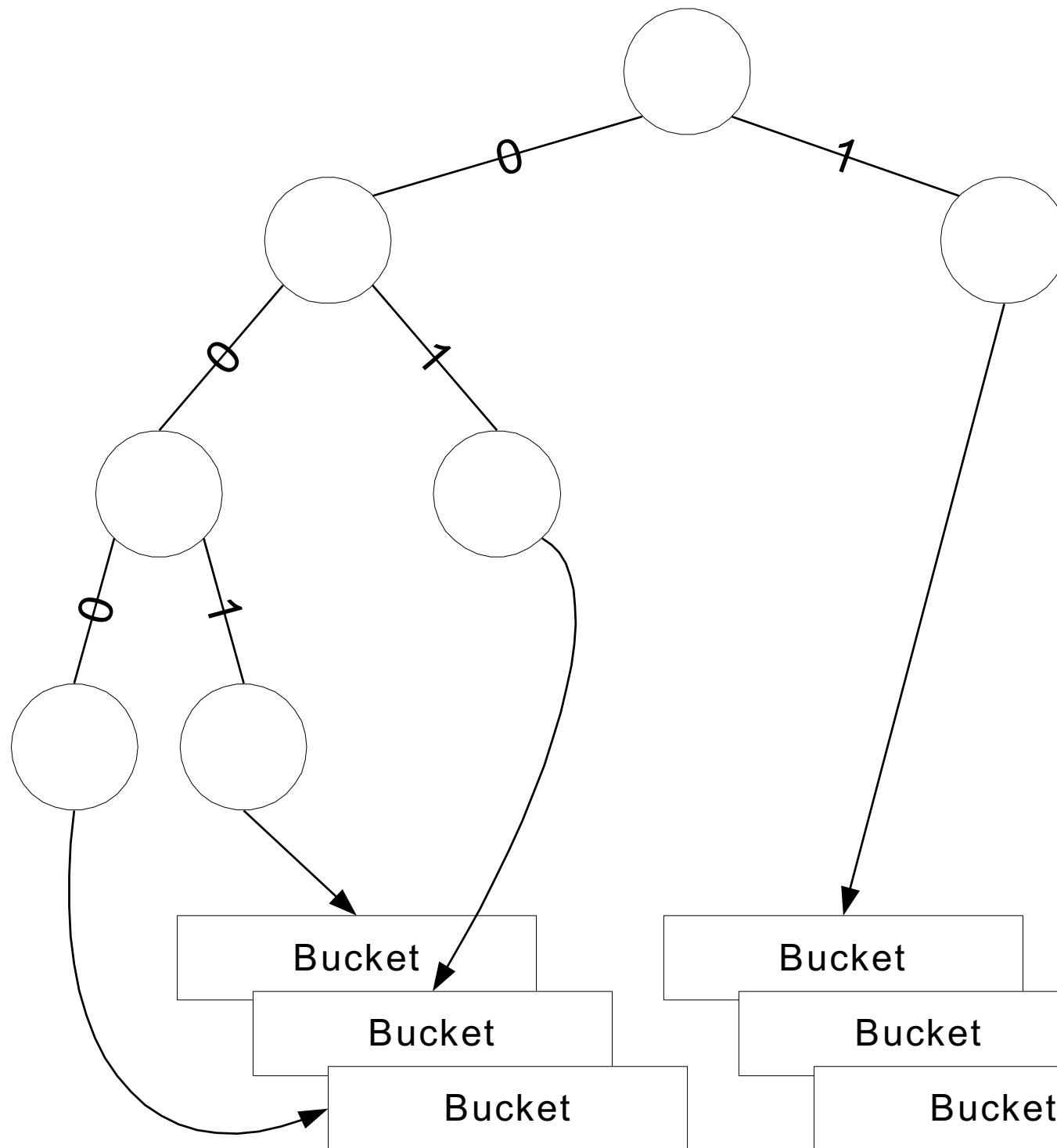
Dynamisches Hashen

Problem: Wie groß soll die Anzahl n der Hash-Felder sein?

- n zu groß $\rightarrow$ schlechte Platznutzung und –Lokalität
- n zu klein $\rightarrow$ Viele Überlaufseiten, lange Listen

Datenbanken verwenden daher **dynamisches Hashen**
(dynamisch wachsende und schrumpfende Bildbereiche)

- **Erweiterbares** Hashen
(Vermeidung des Umkopierens)



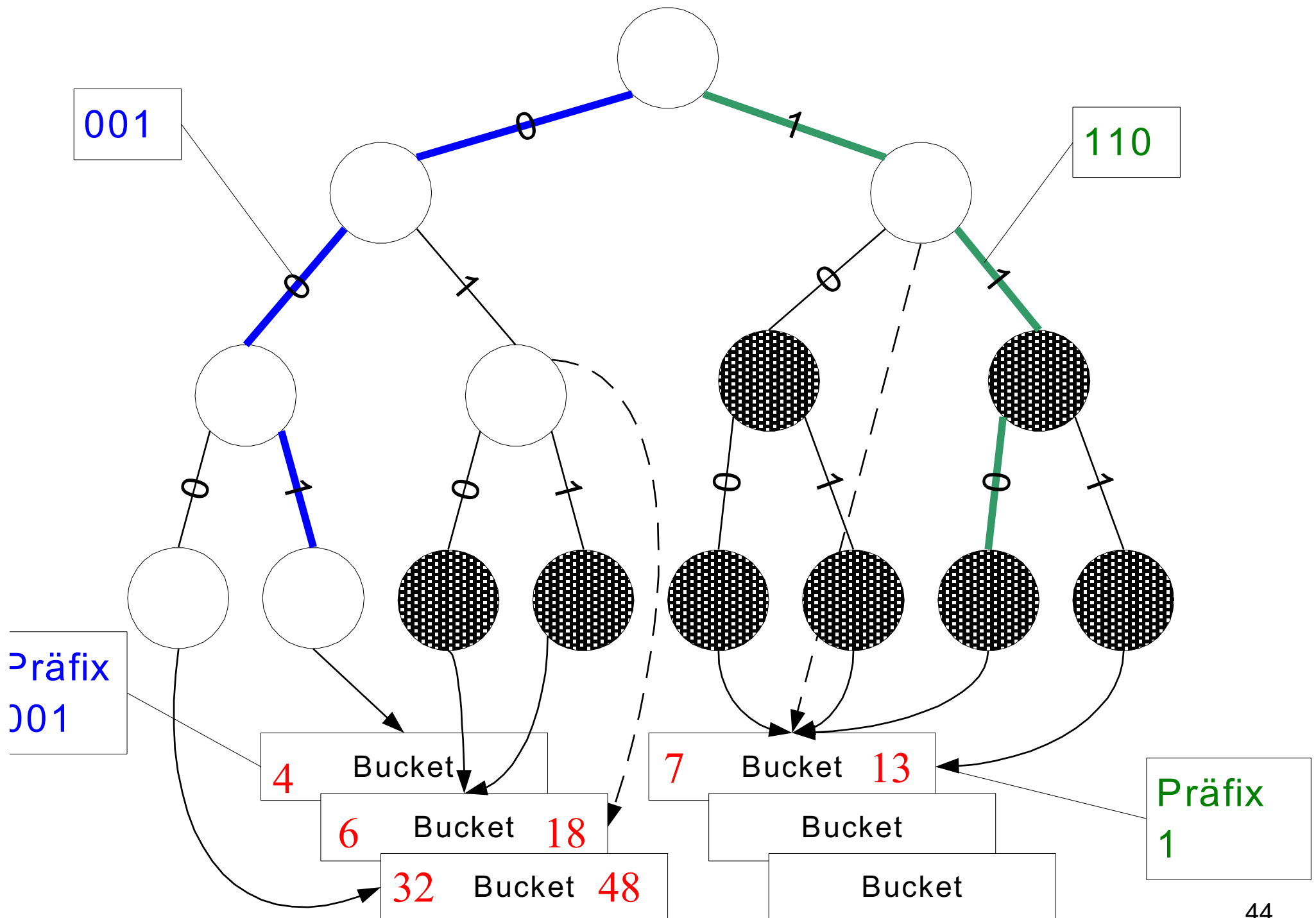
binärer
Trie,
Entschei-
dungs-
baum
Directory

Hashfunktion für erweiterbares Hashing

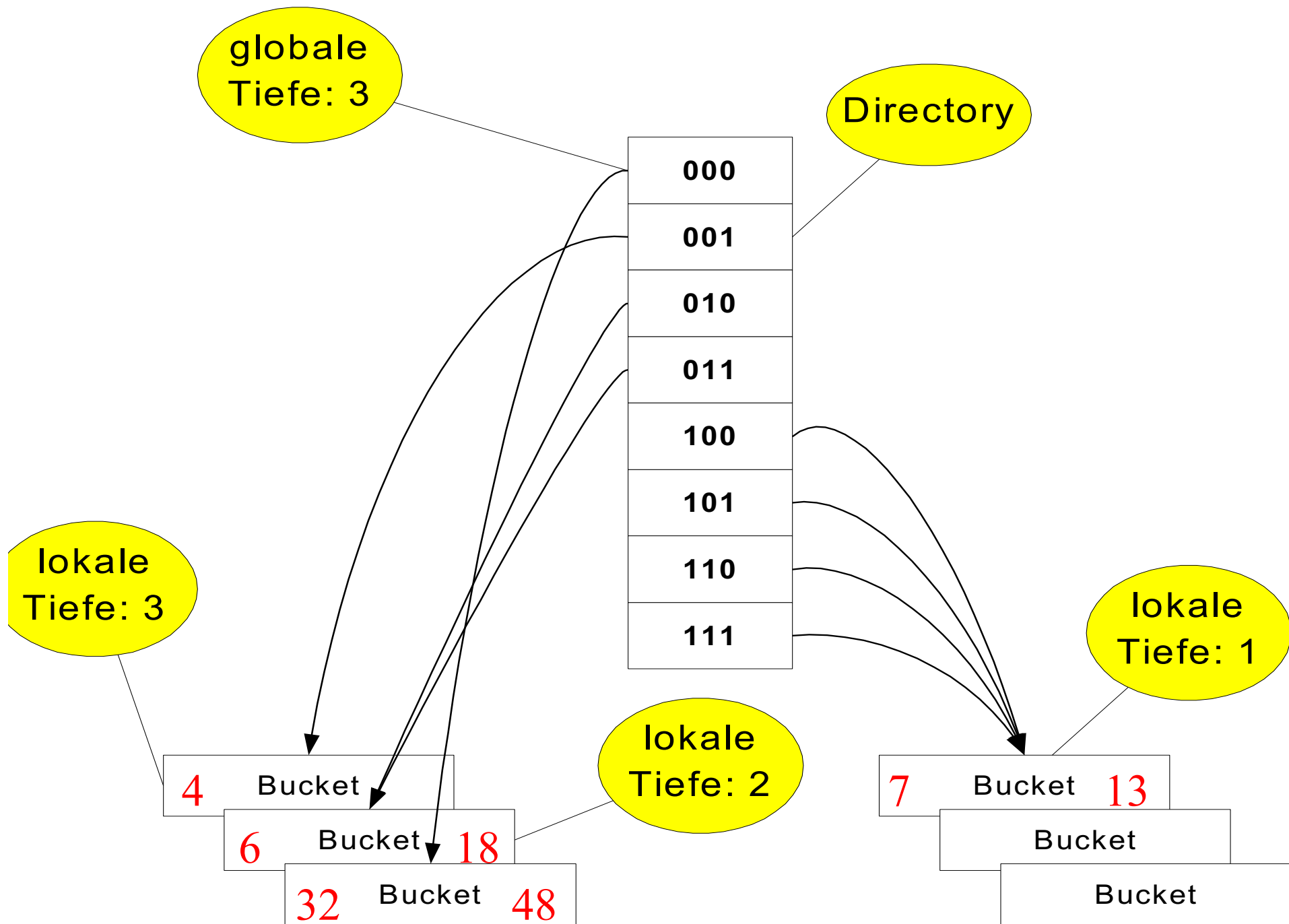
- Funktion h : Schlüsselmenge $\rightarrow \{0,1\}^*$
- Anfangs nur ein (kurzer) Präfix des Hashwertes (Bitstrings)
- Wenn Hashtabelle wächst, sukzessive längerer Präfix
- Beispiel-Hashfunktion: gespiegelte binäre PersNr

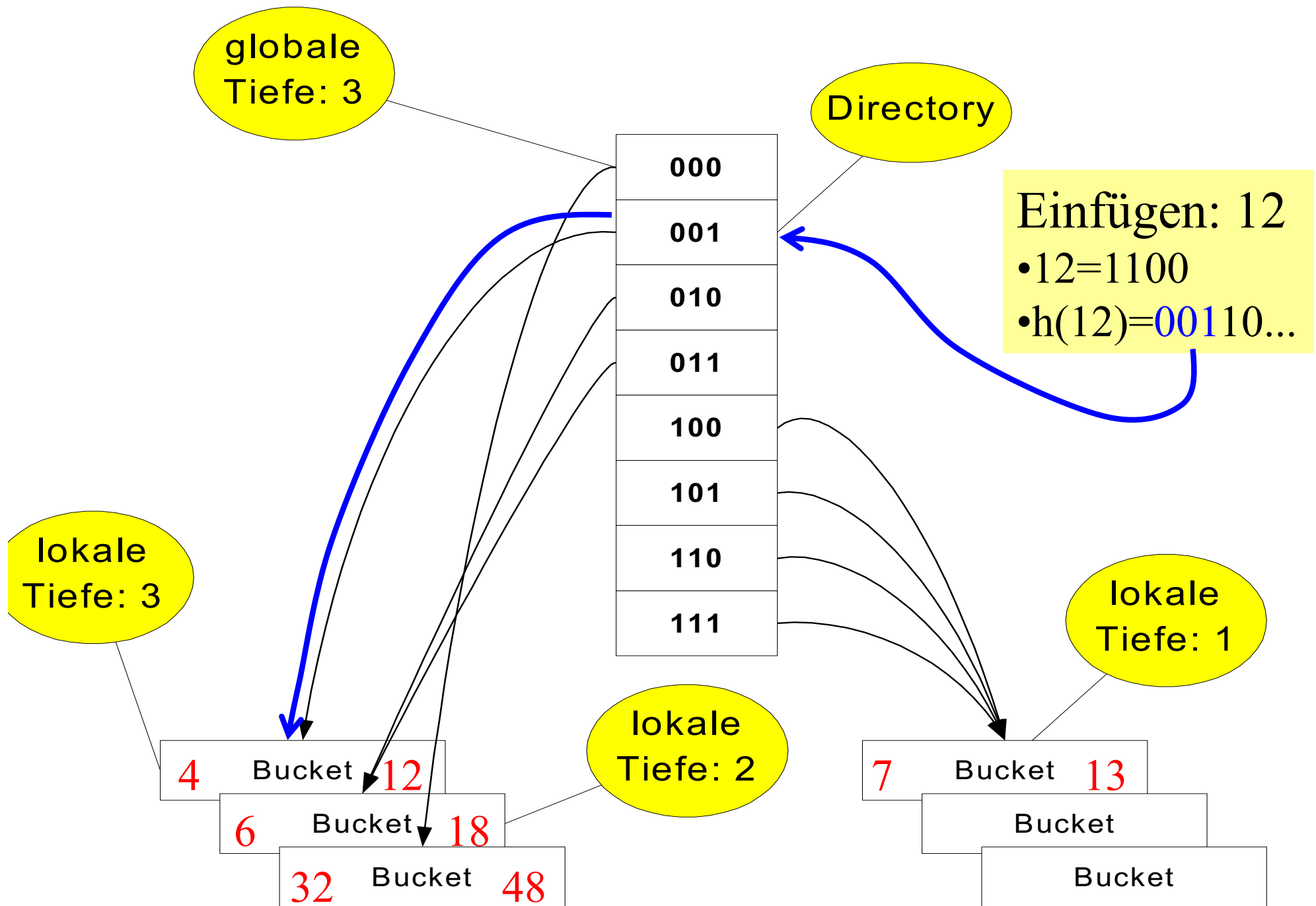
- $h(004) = 001000000\dots$ ($4 = 0..0100$)
- $h(006) = 011000000\dots$ ($6 = 0..0110$)
- $h(007) = 111000000\dots$ ($7 = 0..0111$)
- $h(013) = 101100000\dots$ ($13 = 0..01101$)
- $h(018) = 010010000\dots$ ($18 = 0..010010$)
- $h(032) = 000001000\dots$ ($32 = 0..0100000$)
- $H(048) = 000011000\dots$ ($48 = 0..0110000$)

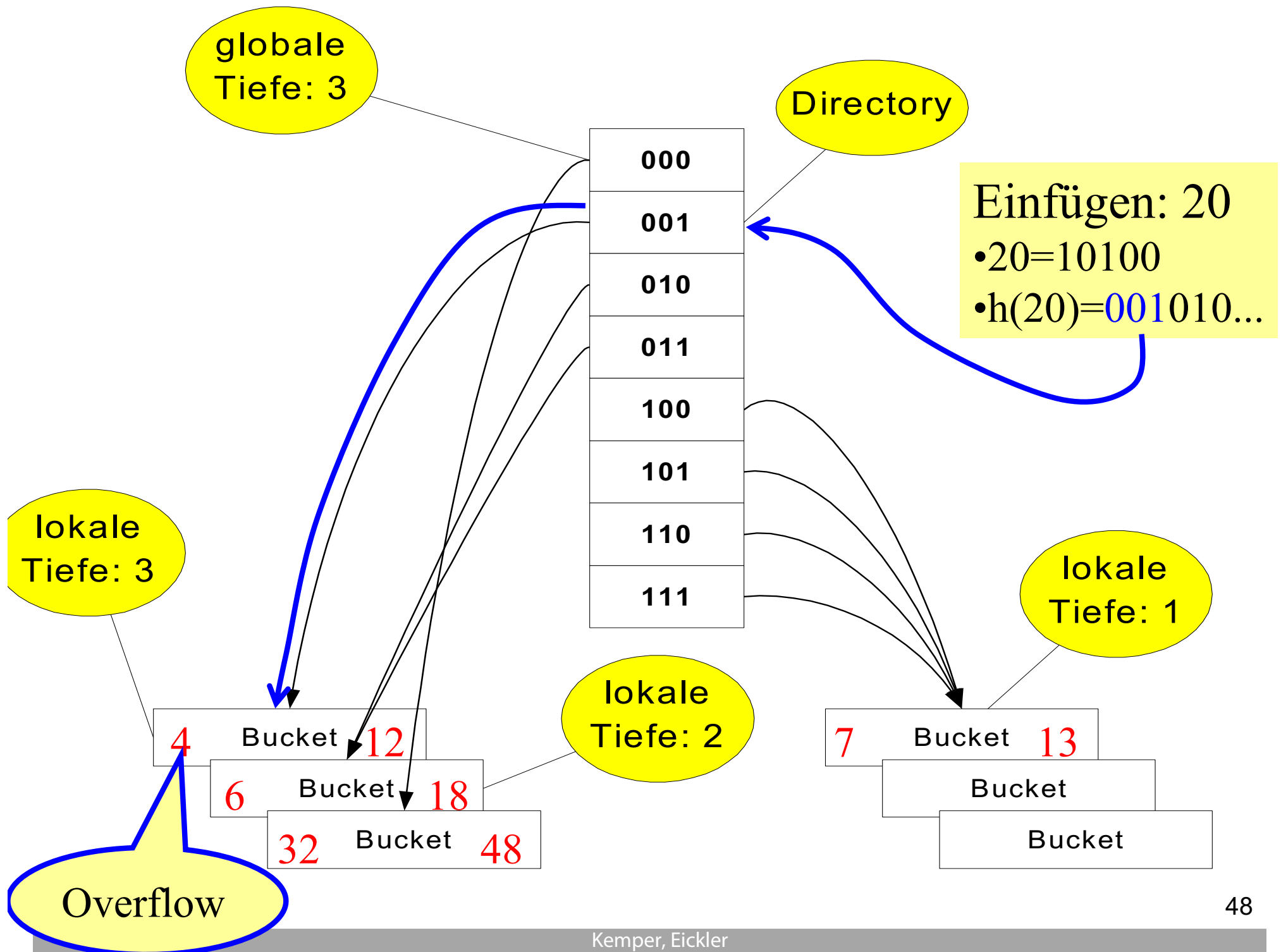


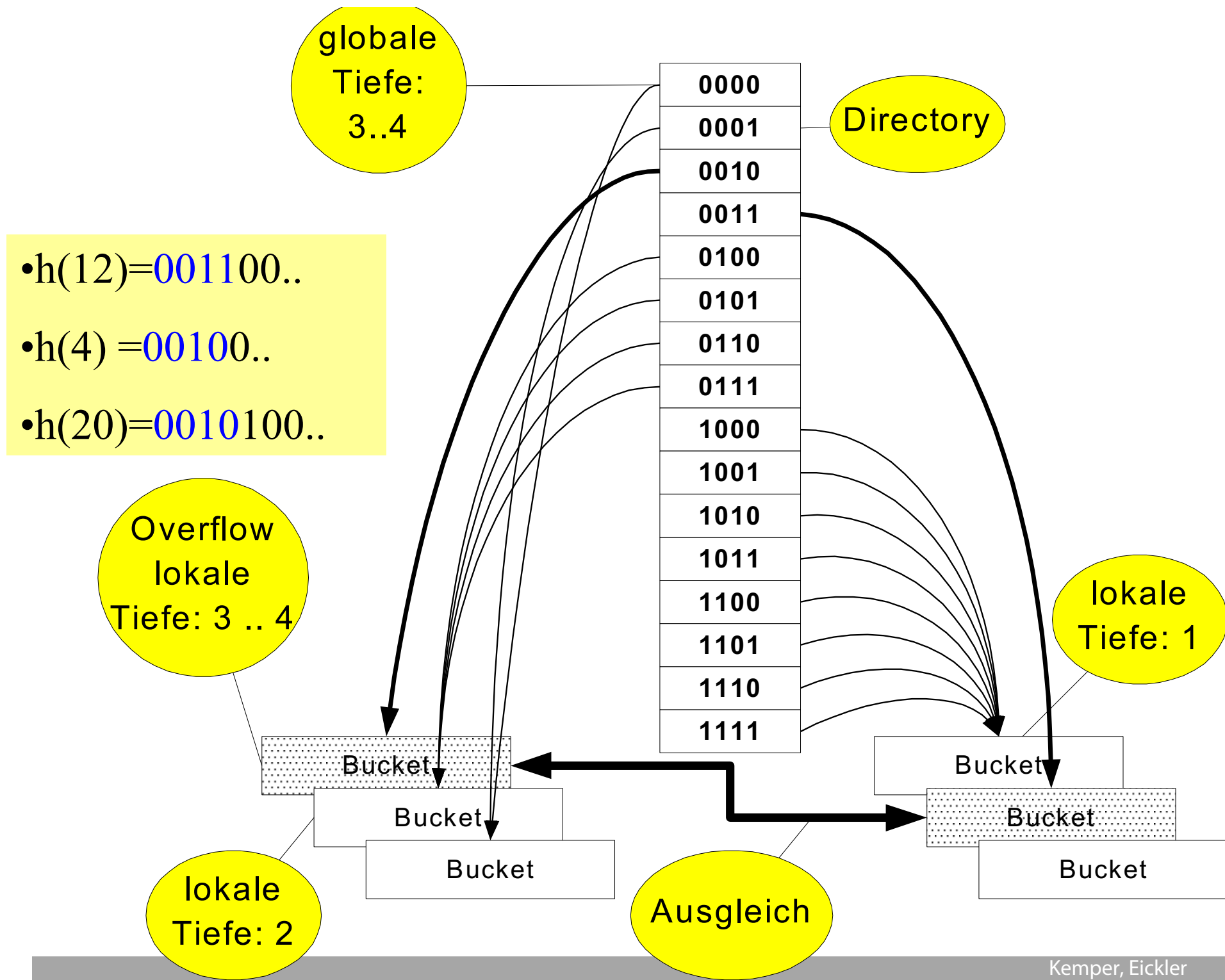


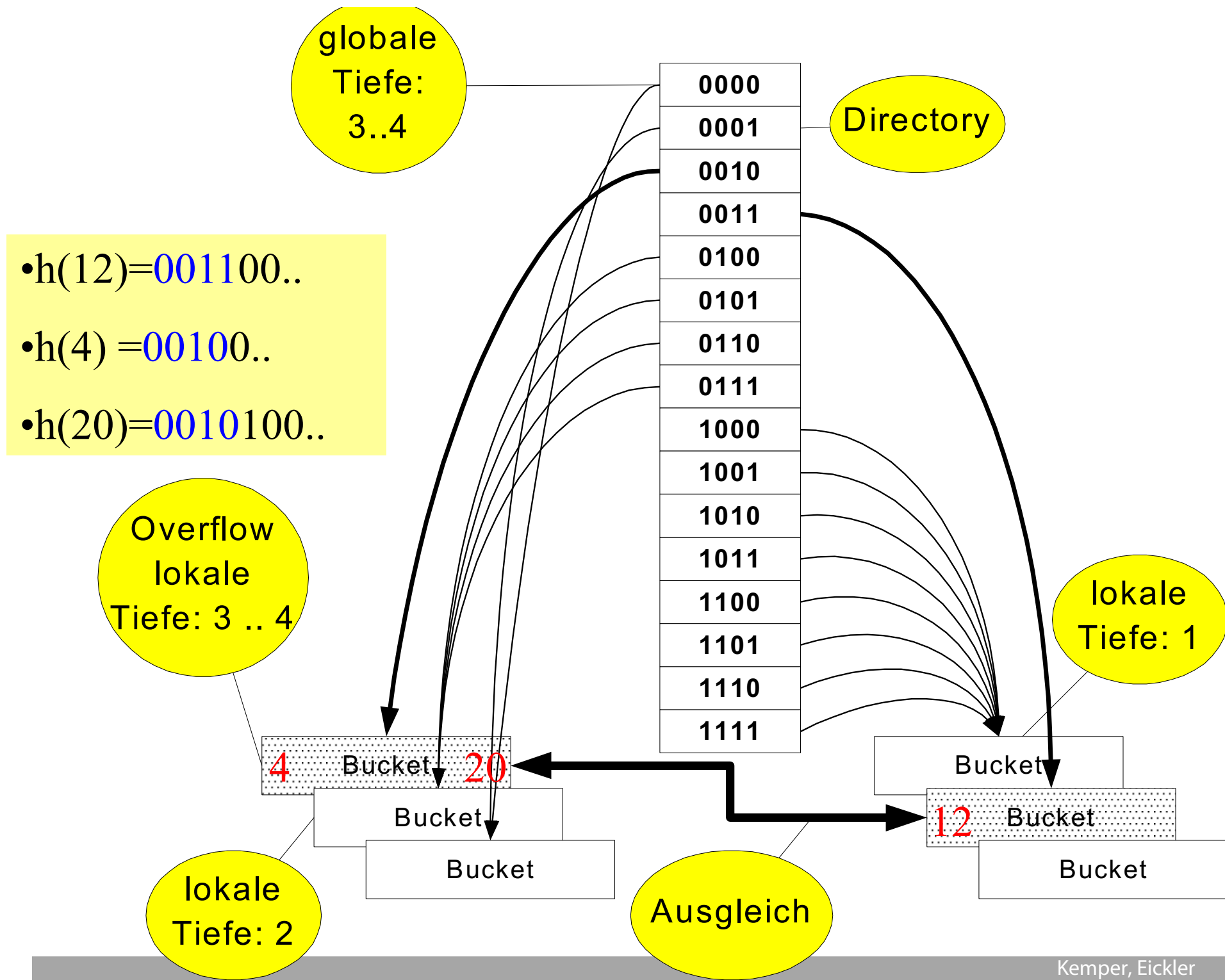


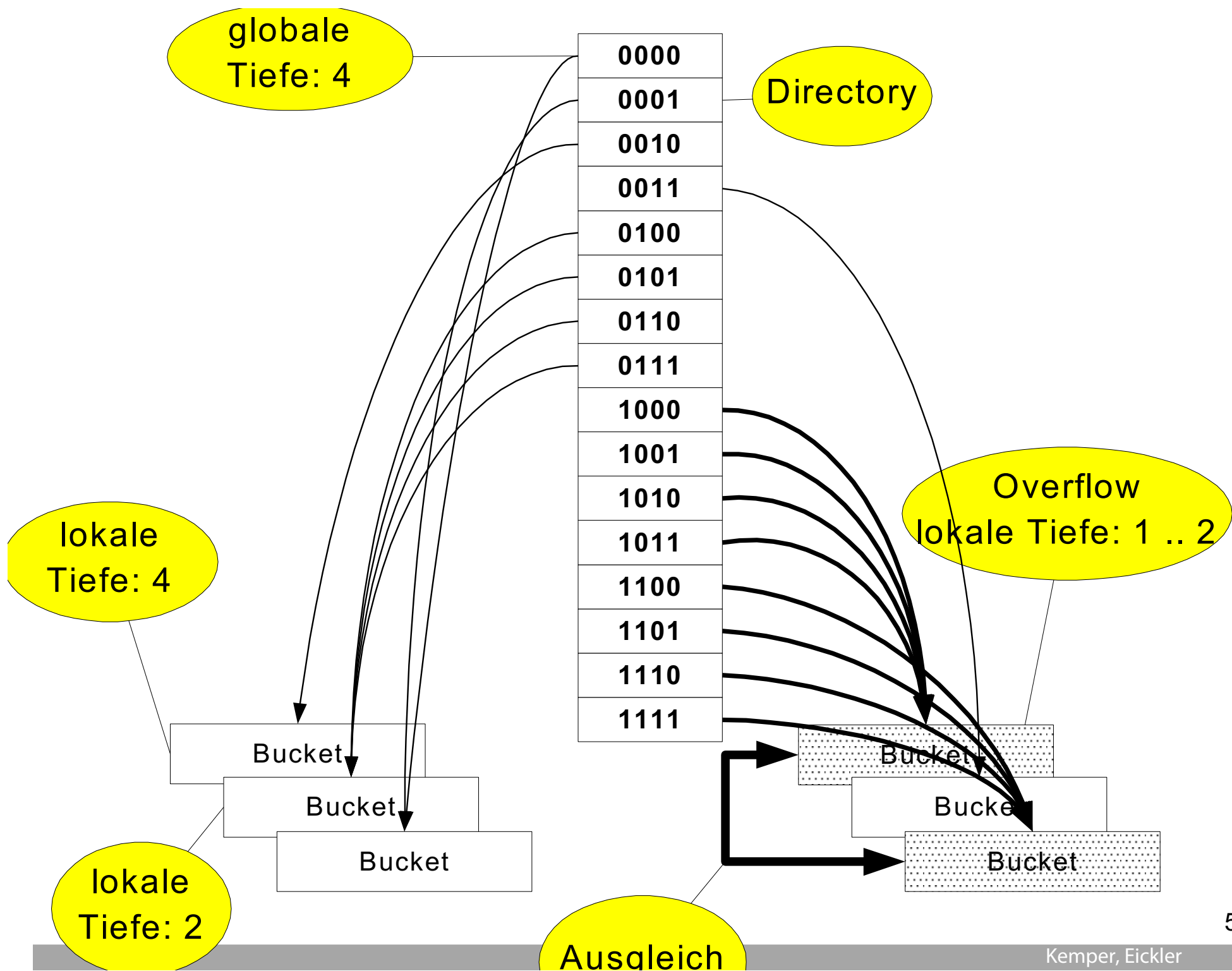






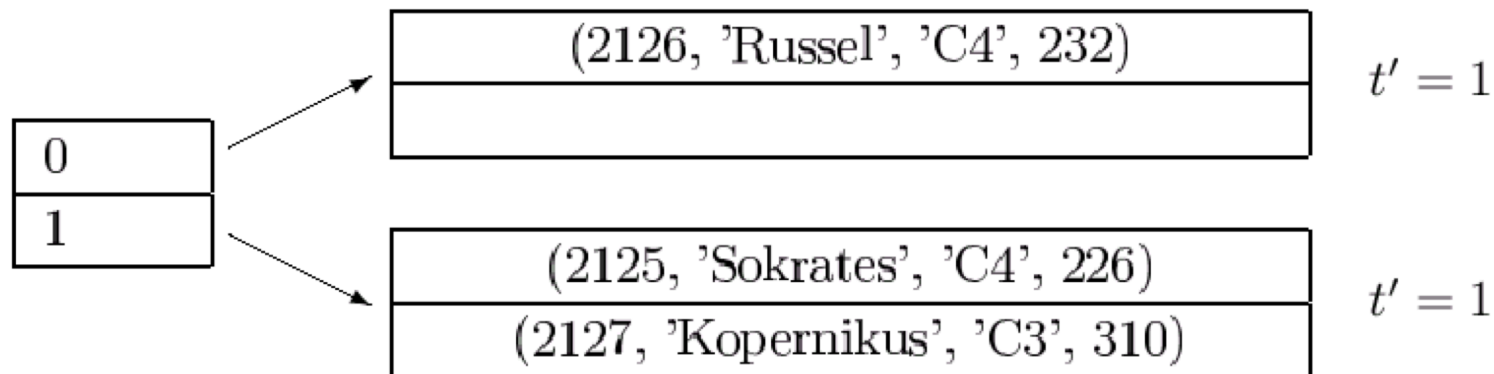






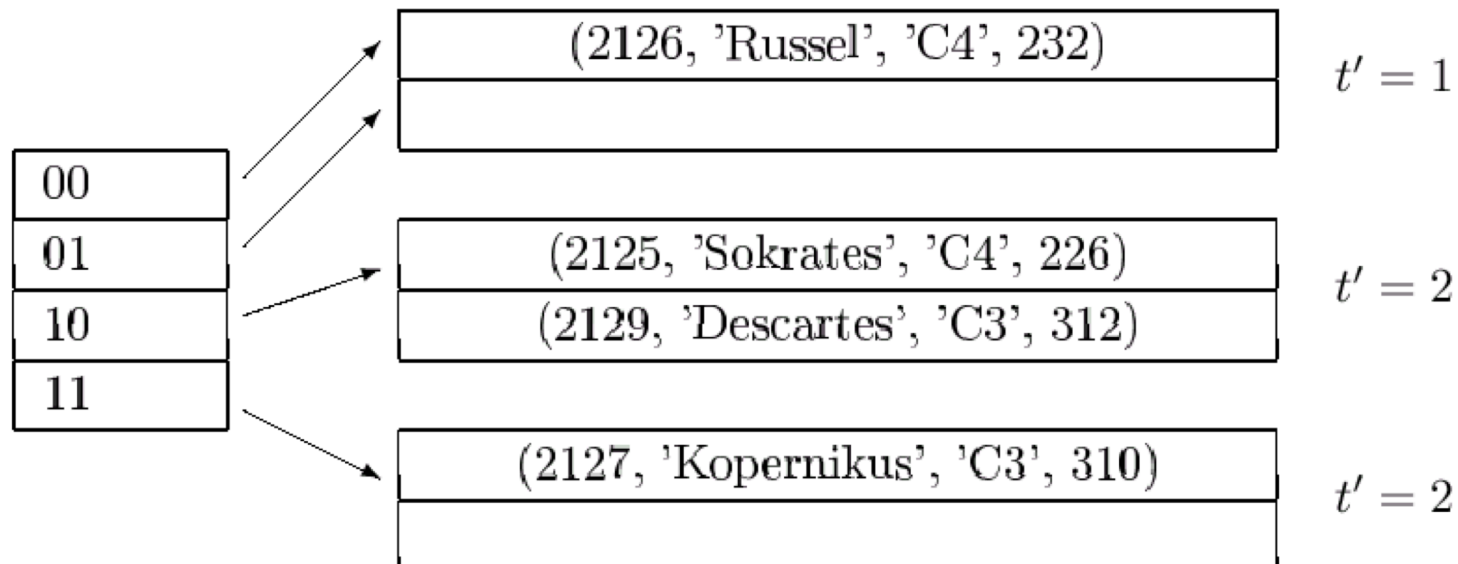
Erweiterbares Hashing: Beispiel

x	$h(x)$	
	d	p
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001



Erweiterbares Hashing: Beispiel

x	$h(x)$	
	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001



Zusammenfassung

- Index-Sequentielle Zugriffsmethode (ISAM-Index)
 - Statisch, baum-basierte Indexstruktur
- B⁺-Bäume
 - Die Datenbank-Indexstruktur, auf linearer Ordnung basierend, dynamisch, kleine Baumhöhe für fokussierten Zugriff auf Bereiche
- Geclusterte vs. ungeclusterte Indexe
 - Sequentieller Zugriff vs. Verwaltungsaufwand
- Hash-basierte Indexe
 - Dynamische Anpassung des Index auf die Daten

