

# Algorithmen und Datenstrukturen 2

**Prof. Dr. E. Rahm**

Sommersemester 2002

Universität Leipzig

Institut für Informatik

<http://dbs.uni-leipzig.de>



## Zur Vorlesung allgemein

- Vorlesungsumfang: 2 + 1 SWS
- Vorlesungsskript
  - im WWW abrufbar (PDF, PS und HTML)
  - Adresse <http://dbs.uni-leipzig.de>
  - ersetzt nicht die Vorlesungsteilnahme oder zusätzliche Benutzung von Lehrbüchern
- Übungen
  - Durchführung in zweiwöchentlichem Abstand
  - selbständige Lösung der Übungsaufgaben wesentlich für Lernerfolg
  - Übungsblätter im WWW
  - praktische Übungen auf Basis von Java
- Vordiplomklausur ADS1+ADS2 im Juli
  - Zulassungsvoraussetzungen: Übungsschein ADS1 + erfolgreiche Übungsbearbeitung ADS2
  - erfordert fristgerechte Abgabe und korrekte Lösung der meisten Aufgaben sowie Bearbeitung aller Übungsblätter (bis auf höchstens eines)

# Termine Übungsbetrieb

- Ausgabe 1. Übungsblatt: Montag, 8. 4. 2002; danach 2-wöchentlich
- Abgabe gelöster Übungsaufgaben bis spätestens Montag der übernächsten Woche, 11:15 Uhr
  - vor Hörsaal 13 (Abgabemöglichkeit 11:00 - 11:15 Uhr)
  - oder früher im Fach des Postschranks HG 2. Stock, neben Raum 2-22
  - Programmieraufgaben: dokumentierte Listings der Quellprogramme sowie Ausführung
- 6 Übungsgruppen

Nr.	Termin	Woche	Hörsaal	Beginn	Übungsleiter	#Stud.
1	Mo, 17:15	A	HS 16	22.4.	Richter	60
2	Mo, 9:15	B	SG 3-09	29.4.	Richter	30
3	Di, 11:15	A	SG 3-07	23.4.	Böhme	30
4	Di, 11:15	B	SG 3-07	30.4.	Böhme	30
5	Do, 11.15	A	SG 3-05	25.4.	Böhme	30
6	Do, 11.15	B	SG 3-05	2.5.	Böhme	30

- Einschreibung über Online-Formular
- Aktuelle Infos siehe WWW



# Ansprechpartner ADS2

- Prof. Dr. E. Rahm
  - während/nach der Vorlesung bzw. Sprechstunde (Donn. 14-15 Uhr), HG 3-56
  - rahm@informatik.uni-leipzig.de
- Wissenschaftliche Mitarbeiter
  - Timo Böhme, boehme@informatik.uni-leipzig.de, HG 3-01
  - Dr. Peter Richter, prichter@informatik.uni-leipzig.de, HG 2-20
- Studentische Hilfskräfte
  - Tilo Dietrich, TiloDietrich@gmx.de
  - Katrin Starke, katrin.starke@gmx.de
  - Thomas Tym, mai96iwe@studserv.uni-leipzig.de
- Web-Angelegenheiten:
  - S. Jusek, juseks@informatik.uni-leipzig.de, HG 3-02



# Vorläufiges Inhaltsverzeichnis

## 1. Mehrwegbäume

- m-Wege-Suchbaum
- B-Baum
- B\*-Baum
- Schlüsselkomprimierung , Präfix-B-Baum
- 2-3-Baum, binärer B-Baum
- Digitalbäume

## 2. Hash-Verfahren

- Grundlagen
- Kollisionsverfahren
- Erweiterbares und dynamisches Hashing

## 3. Graphenalgorithmien

- Arten von Graphen
- Realisierung von Graphen
- Ausgewählte Graphenalgorithmien

## 4. Textsuche



# Literatur

Das intensive Literaturstudium zur Vertiefung der Vorlesung wird dringend empfohlen. Auch Literatur in englischer Sprache sollte verwendet werden.

■ *T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, Reihe Informatik, Band 70, BI-Wissenschaftsverlag, 4. Auflage, Spektrum-Verlag, 2002*

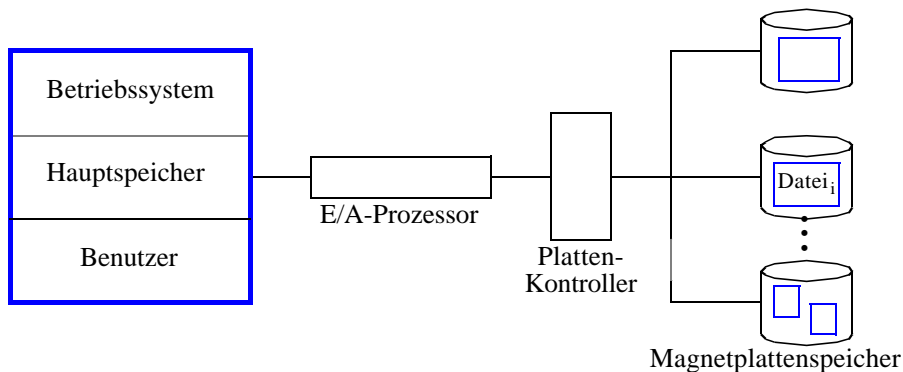
### ■ Weitere Bücher

- *V. Claus, A. Schwill: Duden Informatik, BI-Dudenverlag, 3. Auflage 2001*
- *D.A. Knuth: The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973*
- *R. Sedgewick: Algorithmen. Addison-Wesley 1992*
- *G. Saake, K. Sattler: Algorithmen und Datenstrukturen - Eine Einführung mit Java. dpunkt-Verlag, 2002*
- *M.A. Weiss: Data Structures & Problem Solving using Java. Addison-Wesley, 2. Auflage 2002*



# Suchverfahren für große Datenmengen

- bisher betrachtete Datenstrukturen (Arrays, Listen, Binärbäume) und Algorithmen waren auf im Hauptspeicher vorliegende Daten ausgerichtet
- effiziente Suchverfahren für große Datenmengen auf Externspeicher erforderlich (persistente Speicherung)
  - große Datenmengen können nicht vollständig in Hauptspeicher-Datenstrukturen abgebildet werden
  - Zugriffsgranulat sind Seiten bzw. Blöcke von Magnetplatten : z.B. 4-16 KB
  - Zugriffskosten 5 Größenordnungen langsamer als für Hauptspeicher (5 ms vs. 50 ns)



## Sequentieller Dateizugriff

- Sequentielle Dateiorganisation
  - Datei besteht aus Folge gleichartiger Datensätze
  - Datensätze sind auf Seiten/Blöcken gespeichert
  - ggf. bestimmte Sortierreihenfolge (bzgl. eines Schlüssels) bei der Speicherung der Sätze (sortiert-sequentielle Speicherung)
- Sequentieller Zugriff
  - Lesen aller Seiten / Sätze vom Beginn der Datei an
  - sehr hohe Zugriffskosten, v.a. wenn nur ein Satz benötigt wird
- Optimierungsmöglichkeiten
  - „dichtes Packen“ der Sätze innerhalb der Seiten (hohe Belegungsichte)
  - Clustering zwischen Seiten, d.h. „dichtes Packen“ der Seiten einer Datei auf physisch benachbarte Plattenbereiche, um geringe Zeiten für Plattenzugriff zu ermöglichen
- Schneller Zugriff auf einzelne Datensätze erfordert Einsatz von zusätzlichen *Indexstrukturen*, z.B. Mehrwegbäume
- Alternative: gestreute Speicherung der Sätze (-> Hashing)

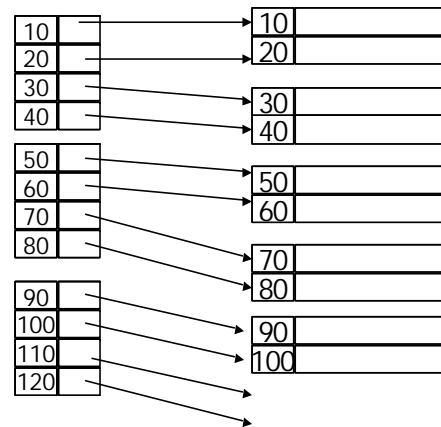
# Dichtbesetzer vs. dünnbesetzter Index

## ■ Dichtbesetzter Index (dense index)

- für jeden Datensatz existiert ein Eintrag in Indexdatei
- höherer Indexaufwand als bei dünnbesetztem Index
- breiter anwendbar, u.a auch bei unsortierter Speicherung der Sätze
- einige Auswertungen auf Index möglich, ohne Zugriff auf Datensätze (Existenztest, Häufigkeitsanfragen, Min/Max-Bestimmung)

Dense Index

Sequential File



## ■ Anwendungsbeispiel

- 1 Million Sätze, B=20, 200 Indexeinträge pro Seite
- Dateigröße:
- Indexgröße:
- mittlere Zugriffskosten:



# Dichtbesetzer vs. dünnbesetzter Index (2)

## ■ Dünnbesetzter Index (sparse index)

- nicht für jeden Schlüsselwert existiert Eintrag in Indexdatei
- sinnvoll v.a. bei Clusterung gemäß Sortierreihenfolge des Indexattributes: ein Indexeintrag pro Datenseite

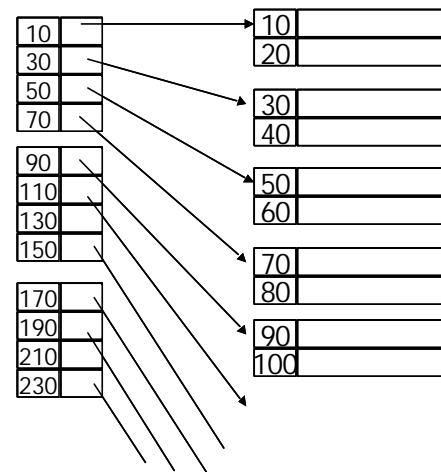
## ■ indexsequentielle Datei (ISAM): sortierte sequentielle Datei mit dünnbesetztem Index für Sortierschlüssel

## ■ Anwendungsbeispiel

- 1 Million Sätze, B=20, 200 Indexeinträge pro Seite
- Dateigröße:
- Indexgröße:
- mittlere Zugriffskosten:

Sparse Index

Sequential File



## ■ Mehrstufiger Index

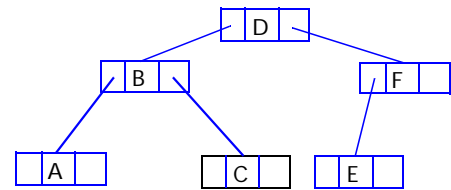
- Indexdatei entspricht sortierter sequentieller Datei -> kann selbst wieder indexiert werden
- auf höheren Stufen kommt nur dünnbesetzte Indexierung in Betracht
- beste Umsetzung im Rahmen von Mehrwegbäumen (B-/B\*-Bäume)



# Mehrwegbäume

## ■ Ausgangspunkt: Binäre Suchbäume (balanciert)

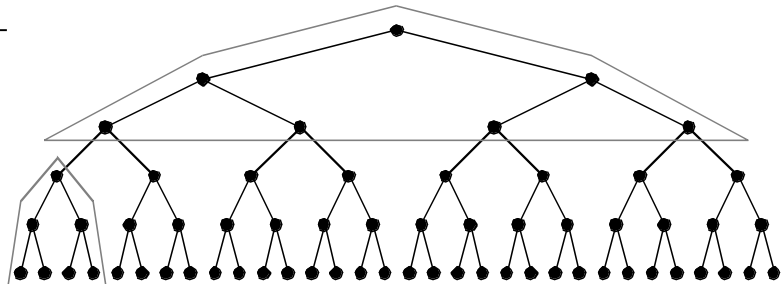
- entwickelt für Hauptspeicher
- ungeeignet für große Datenmengen



## ■ Externspeicherzugriffe erfolgen auf Seiten

- Abbildung von Schlüsselwerten/Sätzen auf Seiten
- Index-Datenstruktur für schnelle Suche

Beispiel: Zuordnung von Binärbaum-Knoten zu Seiten



## ■ Alternativen:

- m-Wege-Suchbäume
- B-Bäume
- B\*-Bäume

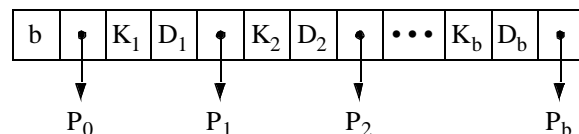
## ■ Grundoperationen: Suchen, Einfügen, Löschen

## ■ Kostenanalyse im Hinblick auf Externspeicherzugriffe

# m-Wege-Suchbäume

## ■ Def.: Ein m-Wege-Suchbaum oder ein m-ärer Suchbaum B ist ein Baum, in dem alle Knoten einen Grad $\leq m$ besitzen. Entweder ist B leer oder er hat folgende Eigenschaften:

- (1) Jeder Knoten des Baums mit  $b$  Einträgen,  $b \leq m - 1$ , hat folgende Struktur:



Die  $P_i$ ,  $0 \leq i \leq b$ , sind Zeiger auf die Unterbäume des Knotens und die  $K_i$  und  $D_i$ ,  $1 \leq i \leq b$  sind Schlüsselwerte und Daten.

- (2) Die Schlüsselwerte im Knoten sind aufsteigend geordnet:  $K_i \leq K_{i+1}$ ,  $1 \leq i < b$ .
- (3) Alle Schlüsselwerte im Unterbaum von  $P_i$  sind kleiner als der Schlüsselwert  $K_{i+1}$ ,  $0 \leq i < b$ .
- (4) Alle Schlüsselwerte im Unterbaum von  $P_b$  sind größer als der Schlüsselwert  $K_b$ .
- (5) Die Unterbäume von  $P_i$ ,  $0 \leq i \leq b$  sind auch m-Wege-Suchbäume.

## ■ Die $D_i$ können Daten oder Zeiger auf die Daten repräsentieren

- **direkter Index**: eingebettete Daten (weniger Einträge pro Knoten; kleineres  $m$ )
- **indirekter Index**: nur Verweise; erfordert separaten Zugriff auf Daten zu dem Schlüssel

## m-Wege-Suchbäume (2)

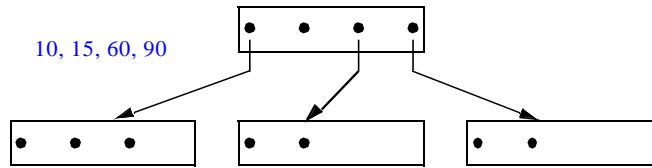
### ■ Beispiel: Aufbau eines m-Wege-Suchbaumes (m = 4)

#### Einfügereihenfolge:

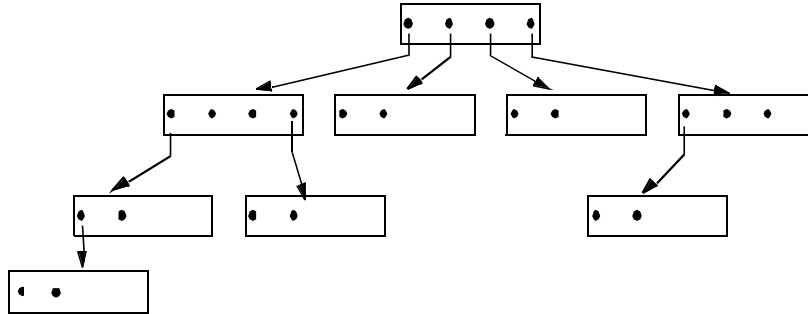
30, 50, 80



10, 15, 60, 90



20, 35, 5, 95, 1, 25



### ■ Beobachtungen

- Die Schlüssel in den inneren Knoten besitzen zwei Funktionen. Sie identifizieren Daten(sätze) und sie dienen als Wegweiser in der Baumstruktur
- Der m-Wege-Suchbaum ist im allgemeinen nicht ausgeglichen

## m-Wege-Suchbäume (3)

### ■ Wichtige Eigenschaften für alle Mehrwegbäume:

$S(P_i)$  sei die Seite, auf die  $P_i$  zeigt, und  $K(P_i)$  sei die Menge aller Schlüssel, die im Unterbaum mit Wurzel  $S(P_i)$  gespeichert werden können. Dann gelten folgende Ungleichungen:

- (1)  $x \in K(P_0): \quad x < K_1$
- (2)  $x \in K(P_i): \quad K_i < x < K_{i+1} \quad \text{für } i = 1, 2, \dots, b-1$
- (3)  $x \in K(P_b): \quad K_b < x$

### ■ Kostenanalyse

- Die Anzahl der Knoten  $N$  in einem vollständigen Baum der Höhe  $h$ ,  $h \geq 1$ , ist

$$N = \sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1}$$

- Im ungünstigsten Fall ist der Baum völlig entartet:  $n = N = h$
- Schranken für die Höhe eines m-Wege-Suchbaums:  $\log_m(n+1) \leq h \leq n$

# m-Wege-Suchbäume (4)

## ■ Definition des Knotenformats:

```
class MNode {
    int m;           // max. Grad des Knotens (m)
    int b;           // Anzahl der Schluessel im Knoten (b <= m-1)
    Orderable[] keys; // Liste der Schluessel
    Object[] data;   // Liste der zu den Schluesseln gehoerigen Datenobjekte
    MNode[] ptr;    // Liste der Zeiger auf Unterbaeume

    /** Konstruktor */
    public MNode(int m, Orderable key, Object obj) {
        this.m = m; b = 1;
        keys = new Orderable[m-1];
        data = new Object[m-1];
        ptr = new MNode[m];
        keys[0] = key; // Achtung: keys[0] entspricht K1, keys[1] K2, ...
        data[0] = obj; // Achtung: data[0] entspricht D1, data[1] D2, ...
    }
}
```

## ■ Rekursive Prozedur zum Aufsuchen eines Schlüssels

```
public Object search(Orderable key, MNode node) {
    if ((node == null) || (node.b < 1)) {
        System.err.println("Schluessel nicht im Baum.");
        return null;
    }
}
```



```
    if (key.less(node.keys[0]))
        return search(key, node.ptr[0]); // key < K1

    if (key.greater(node.keys[node.b-1]))
        return search(key, node.ptr[node.b]); // key > Kb

    int i=0;
    while ((i<node.b-1) && (key.greater(node.keys[i])))
        i++; // gehe weiter, solange key > Ki+1

    if (key.equals(node.keys[i]))
        return node.data[i]; // gefunden

    return search(key, node.ptr[i]); // Ki < key < Ki+1
}
```

## ■ Durchlauf eines m-Wege-Suchbaums in symmetrischer Ordnung

```
public void print(MNode node) {
    if ((node == null) || (node.b < 1)) return;
    print(node.ptr[0]);
    for (int i=0; i<node.b; i++) {
        System.out.println("Schluessel: " + node.keys[i].getKey() +
            " \tDaten: " + node.data[i].toString());
        print(node.ptr[i+1]);
    }
}
```





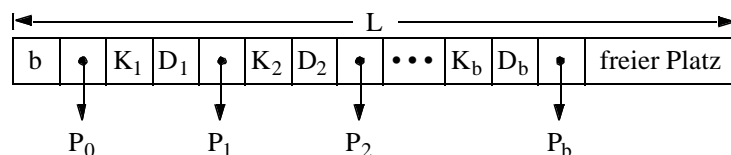
# Mehrwegbäume

- **Ziel: Aufbau sehr breiter Bäume von geringer Höhe**
  - in Bezug auf Knotenstruktur vollständig ausgeglichen
  - effiziente Grundoperationen auf Seiten (= Transporteinheit zum Externspeicher)
  - Zugriffsverhalten weitgehend unabhängig von Anzahl der Sätze
    - ⇒ Einsatz als Zugriffs-/Indexstruktur für 10 als auch für  $10^{10}$  Sätze
- **Grundoperationen:**
  - direkter Schlüsselzugriff auf einen Satz
  - sortiert sequentieller Zugriff auf alle Sätze
  - Einfügen eines Satzes; Löschen eines Satzes
- **Varianten**
  - ISAM-Dateistruktur (1965; statisch, periodische Reorganisation)
  - Weiterentwicklungen: B- und B\*-Baum
  - B-Baum: 1970 von R. Bayer und E. McCreight entwickelt
    - ⇒ dynamische Reorganisation durch Splitten und Mischen von Seiten
- **Breites Spektrum von Anwendungen ("The Ubiquitous B-Tree")**
  - Dateioorganisation ("logische Zugriffsmethode", VSAM)
  - Datenbanksysteme (Varianten des B\*-Baumes sind in allen DBS zu finden!)
  - Text- und Dokumentenorganisation . . .



# B-Bäume

- **Def.:** Seien  $k, h$  ganze Zahlen,  $h \geq 0, k > 0$ .  
Ein B-Baum  $B$  der Klasse  $\tau(k, h)$  ist entweder ein leerer Baum oder ein geordneter Baum mit folgenden Eigenschaften:
  1. Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge  $h-1$ .
  2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens  $k+1$  Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne
  3. Jeder Knoten hat höchstens  $2k+1$  Söhne
  4. Jedes Blatt mit der Ausnahme der Wurzel als Blatt hat mindestens  $k$  und höchstens  $2k$  Einträge.
- Für einen B-Baum ergibt sich folgendes Knotenformat:



## B-Bäume (2)

### ■ Einträge

- Die Einträge für Schlüssel, Daten und Zeiger haben die festen Längen  $l_b$ ,  $l_K$ ,  $l_D$  und  $l_p$ .
- Die Knoten- oder Seitengröße sei  $L$ .
- Maximale Anzahl von Einträgen pro Knoten:  $b_{\max} = \left\lfloor \frac{L - l_b - l_p}{l_K + l_D + l_p} \right\rfloor = 2k$

### ■ Reformulierung der Definition

- (4) und (3). Eine Seite darf höchstens voll sein.
- (4) und (2). Jede Seite (außer der Wurzel) muß mindestens halb voll sein.  
Die Wurzel enthält mindestens einen Schlüssel.
- (1) Der Baum ist, was die Knotenstruktur angeht, vollständig ausgeglichen

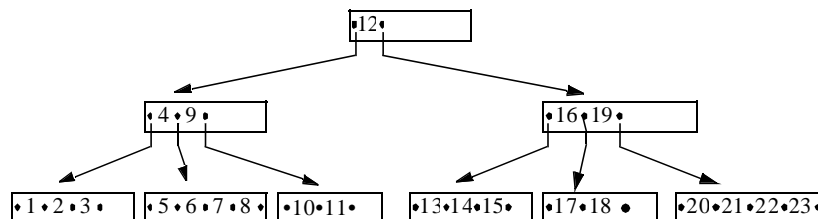
### ■ Balancierte Struktur:

- unabhängig von Schlüsselmenge
- unabhängig ihrer Einfügereihenfolge



## B-Bäume (3)

### ■ Beispiel: B-Baum der Klasse $\tau(2,3)$



- In jedem Knoten stehen die Schlüssel in aufsteigender Ordnung mit  $K_1 < K_2 < \dots < K_b$
- Jeder Schlüssel hat eine Doppelrolle als Identifikator eines Datensatzes und als Wegweiser im Baum
- Die Klassen  $\tau(k,h)$  sind nicht alle disjunkt. Beispielsweise ist ein maximaler Baum aus  $\tau(2,3)$  ebenso in  $\tau(3,3)$  und  $\tau(4,3)$  ist

### ■ Höhe $h$ : Bei einem Baum der Klasse $\tau(k,h)$ mit $n$ Schlüsseln gilt für seine Höhe:

$$\log_{2k+1}(n+1) \leq h \leq \log_{k+1}((n+1)/2) + 1 \quad \text{für } n \geq 1$$

und  $h = 0$

für  $n = 0$



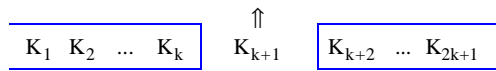
# Einfügen in B-Bäumen

## ■ Was passiert, wenn Wurzel überläuft ?

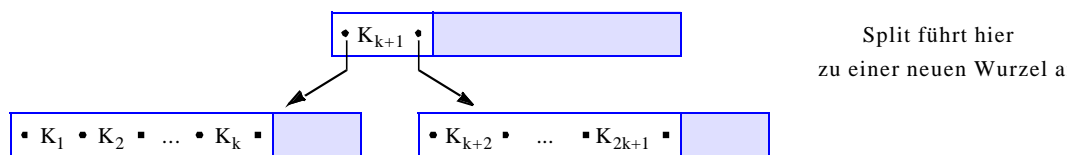
•  $K_1$  •  $K_2$  • ... •  $K_{2k}$  •  $K_{2k+1}$

## ■ Fundamentale Operation: Split-Vorgang

1. Anforderung einer neuen Seite und
2. Aufteilung der Schlüsselmenge nach folgendem Prinzip



- mittlere Schlüssel (Median) wird zum Vaterknoten gereicht
- Ggf. muß Vaterknoten angelegt werden (Anforderung einer neuen Seite).



- Blattüberlauf erzwingt Split-Vorgang, was Einfügung in den Vaterknoten impliziert
- Wenn dieser überläuft, folgt erneuter Split-Vorgang
- Split-Vorgang der Wurzel führt zu neuer Wurzel: Höhe des Baumes erhöht sich um 1

## ■ Bei B-Bäumen ist Wachstum von den Blättern zur Wurzel hin gerichtet

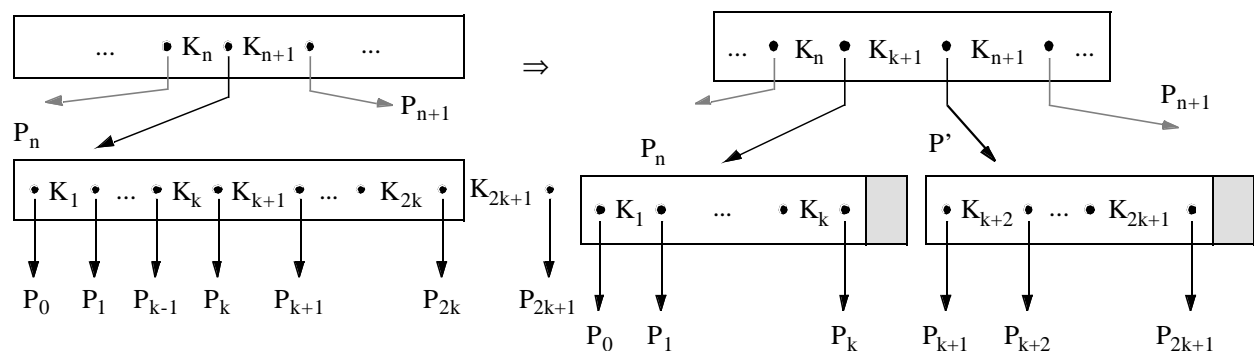


# Einfügen in B-Bäumen (2)

## ■ Einfügealgorithmus (ggf. rekursiv)

- Suche Einfügeposition
- Wenn Platz vorhanden ist, speichere Element, sonst schaffe Platz durch Split-Vorgang und füge ein

## ■ Split-Vorgang als allgemeines Wartungsprinzip

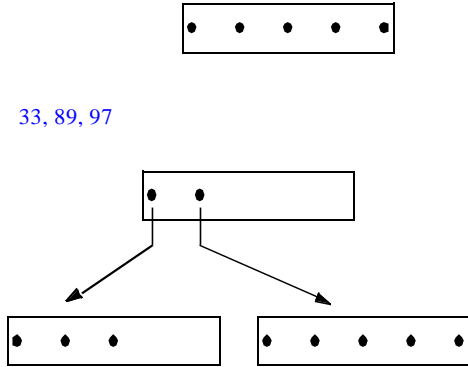


# Einfügen in B-Bäumen (3)

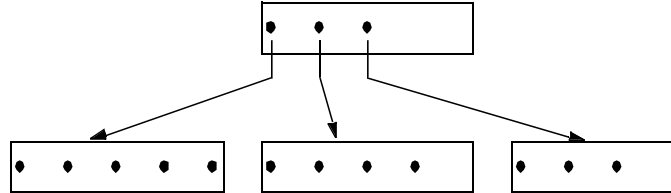
## ■ Aufbau eines B-Baumes der Klasse $\tau(2, h)$

Einfügereihenfolge:

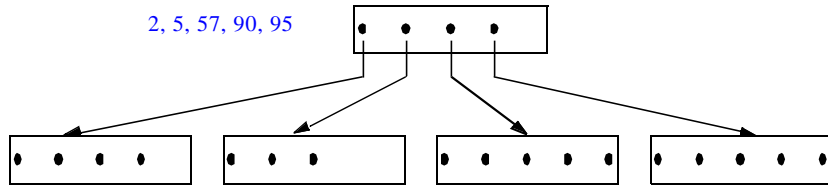
77, 12, 48, 69



91, 37, 45, 83

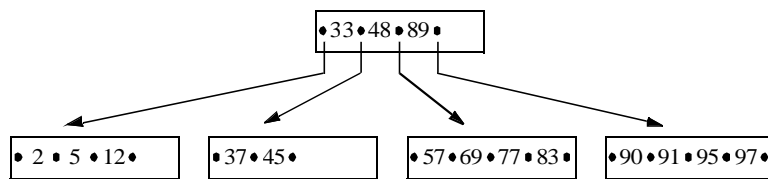


2, 5, 57, 90, 95

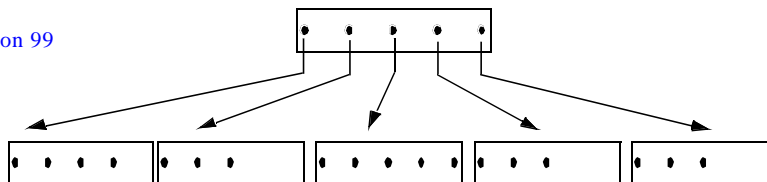


# Einfügen in B-Bäumen (4)

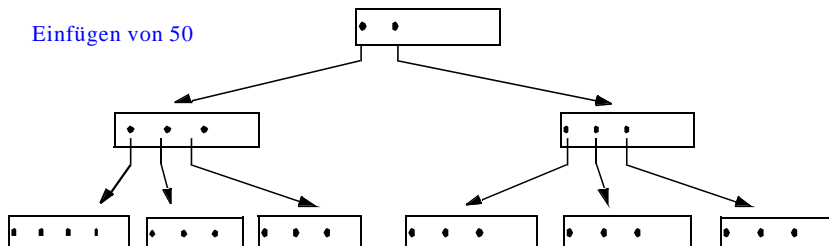
## ■ Aufbau eines B-Baumes der Klasse $\tau(2, h)$



Einfügen von 99



Einfügen von 50



# Kostenanalyse für Suche und Einfügen

## ■ Kostenmaße

- Anzahl der zu holenden Seiten:  $f$  (fetch)
- Anzahl der zu schreibenden Seiten (#geänderter Seiten):  $w$  (write)

## ■ Direkte Suche

- $f_{\min} = 1$  : der Schlüssel befindet sich in der Wurzel
- $f_{\max} = h$  : der Schlüssel ist in einem Blatt
- **mittlere Zugriffskosten**  $h - \frac{1}{k} \leq f_{\text{avg}} \leq h - \frac{1}{2k}$  (für  $h > 1$ )

## ■ Beim B-Baum sind die maximalen Zugriffskosten $h$ eine gute Abschätzung der mittleren Zugriffskosten.

⇒ Bei  $h = 3$  und einem  $k = 100$  ergibt sich  $2.99 \leq f_{\text{avg}} \leq 2.995$

## ■ Sequentielle Suche

- Durchlauf in symmetrischer Ordnung :  $f_{\text{seq}} = N$
- Pufferung der Zwischenknoten im Hauptspeicher wichtig!

## ■ Einfügen

- günstigster Fall - kein Split-Vorgang:  $f_{\min} = h$ ;  $w_{\min} = 1$
- durchschnittlicher Fall:  $f_{\text{avg}} = h$ ;  $w_{\text{avg}} < 1 + \frac{2}{k}$

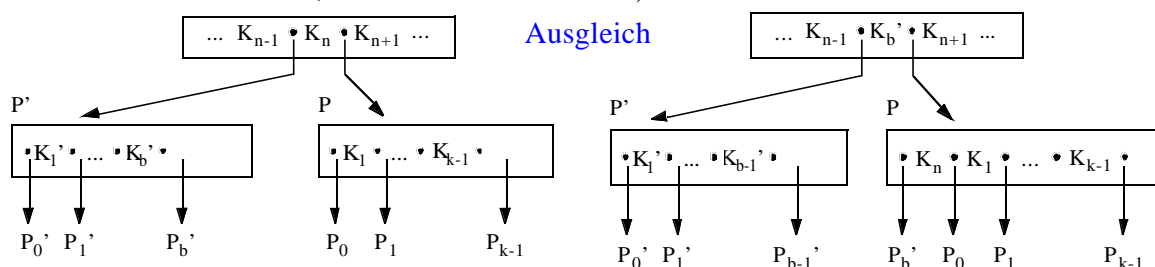


# Löschen in B-Bäumen

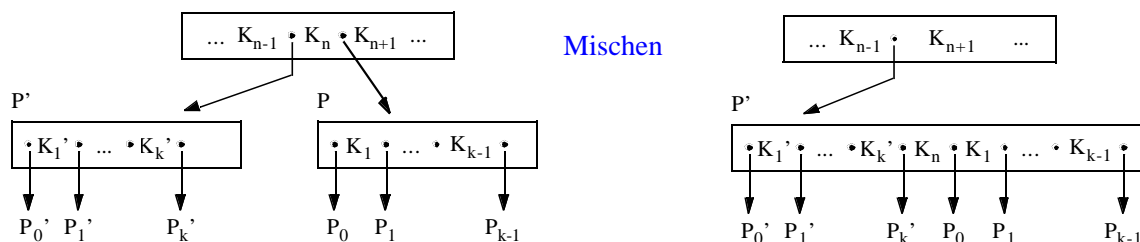
## ■ Die B-Baum-Eigenschaft muß wiederhergestellt werden, wenn die Anzahl der Elemente in einem Knoten kleiner als $k$ wird.

## ■ Durch **Ausgleich** mit Elementen aus einer Nachbarseite oder durch **Mischen** (Konkatenation) mit einer Nachbarseite wird dieses Problem gelöst.

- Maßnahme 1: Ausgleich durch Verschieben von Schlüssel (Voraussetzung: Nachbarseite  $P'$  hat mehr als  $k$  Elemente; Seite  $P$  hat  $k-1$  Elemente)



- Maßnahme 2: Mischen von Seiten



# Löschen in B-Bäumen (2)

## ■ Löschalgorithmus

### (1) Löschen in Blattseite

- Suche x in Seite P
- Entferne x in P und wenn
  - a)  $\#E \geq k$  in P: tue nichts
  - b)  $\#E = k-1$  in P und  $\#E > k$  in P': gleiche Unterlauf über P' aus
  - c)  $\#E = k-1$  in P und  $\#E = k$  in P': mische P und P'.

### (2) Löschen in innerer Seite

- Suche x
- Ersetze  $x = K_i$  durch kleinsten Schlüssel y in  $B(P_i)$  oder größten Schlüssel y in  $B(P_{i-1})$  (nächstgrößerer oder nächstkleinerer Schlüssel im Baum)
- Entferne y im Blatt P
- Behandle P wie unter 1

## ■ Kostenanalyse für das Löschen

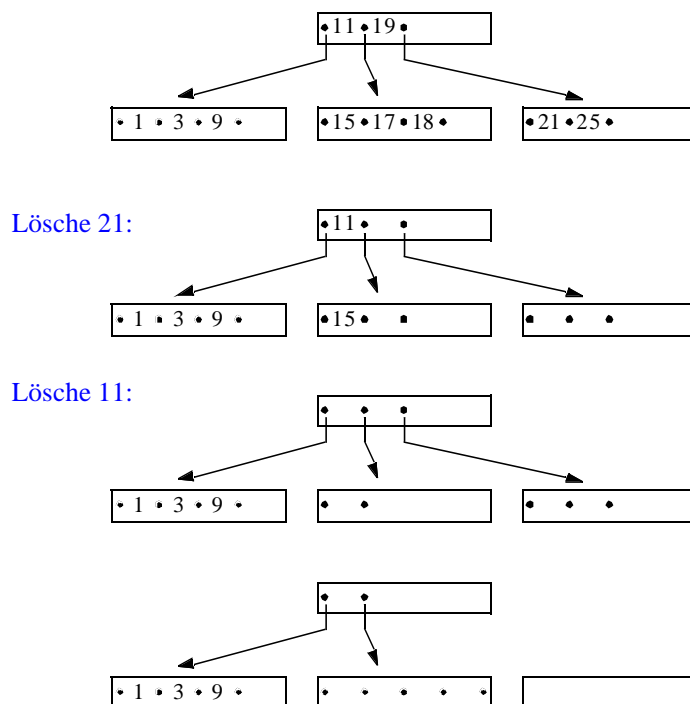
- günstigster Fall:  $f_{\min} = h$  ;  $w_{\min} = 1$
- obere Schranke für durchschnittliche Löschkosten (drei Anteile: 1. Löschen, 2. Ausgleich, 3. anteilige Mischkosten):

$$f_{\text{avg}} \leq f_1 + f_2 + f_3 < h + 1 + \frac{1}{k}$$

$$w_{\text{avg}} \leq w_1 + w_2 + w_3 < 2 + 2 + \frac{1}{k} = 4 + \frac{1}{k}$$



# Löschen in B-Bäumen: Beispiel

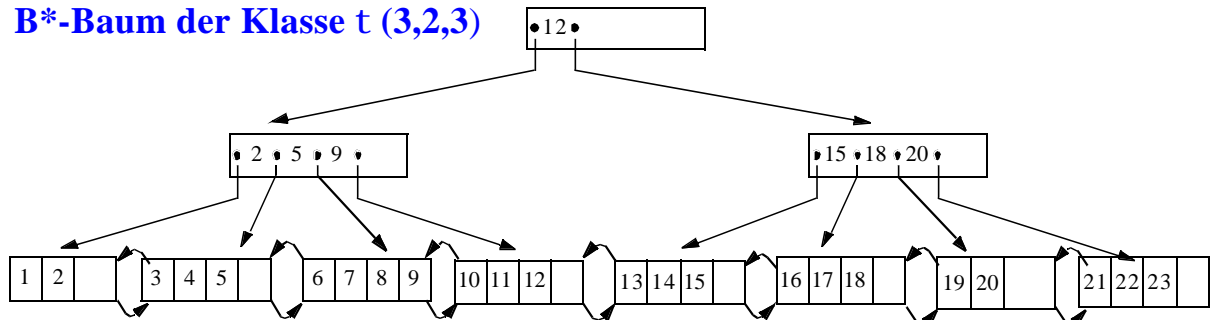


# B\*-Bäume

## ■ Hauptunterschied zu B-Baum: in inneren Knoten wird nur die Wegweiser-Funktion ausgenutzt

- innere Knoten führen nur  $(K_i, P_i)$  als Einträge
  - Information  $(K_i, D_i)$  wird in den Blattknoten abgelegt. Dabei werden alle Schlüssel mit ihren zugehörigen Daten in Sortierreihenfolge in den Blättern abgelegt werden.
  - Für einige  $K_i$  ergibt sich eine redundante Speicherung. Die inneren Knoten bilden also einen Index, der einen schnellen direkten Zugriff zu den Schlüsseln gestattet.
  - Der Verzweigungsgrad erhöht sich beträchtlich, was wiederum die Höhe des Baumes reduziert
  - Durch Verkettung aller Blattknoten läßt sich eine effiziente sequentielle Verarbeitung erreichen, die beim B-Baum einen umständlichen Durchlauf in symmetrischer Ordnung erforderte
- ⇒ B\*-Baum ist die für den praktischen Einsatz wichtigste Variante des B-Baums

### B\*-Baum der Klasse $t(3,2,3)$

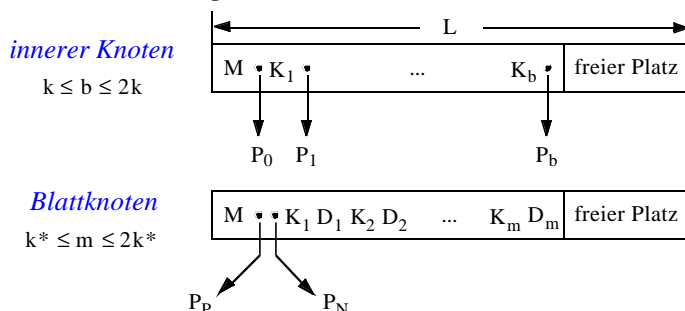


# B\*-Bäume (2)

## ■ Def.: Seien $k, k^*$ und $h^*$ ganze Zahlen, $h^* \geq 0, k, k^* > 0$ . Ein **B\*-Baum** $B$ der Klasse $\tau(k, k^*, h^*)$ ist entweder ein leerer Baum oder ein geordneter Baum, für den gilt:

1. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge  $h^*-1$ .
2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens  $k+1$  Söhne, die Wurzel mindestens 2 Söhne, außer wenn sie ein Blatt ist.
3. Jeder innere Knoten hat höchstens  $2k+1$  Söhne.
4. Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens  $k^*$  und höchstens  $2k^*$  Einträge.

## ■ Unterscheidung von zwei Knotenformaten:



Feld M enthalte Kennung des Seitentyps sowie Zahl der aktuellen Einträge

# B\*-Bäume(3)

- Da die Seiten eine feste Länge L besitzen, läßt sich aufgrund der obigen Formate k und k\* bestimmen:

$$L = l_M + l_P + 2 \cdot k \cdot (l_K + l_P) ; \quad k = \left\lfloor \frac{L - l_M - l_P}{2 \cdot (l_K + l_P)} \right\rfloor$$

$$L = l_M + 2 \cdot l_P + 2 \cdot k^* \cdot (l_K + l_D) ; \quad k^* = \left\lfloor \frac{L - l_M - 2l_P}{2 \cdot (l_K + l_D)} \right\rfloor$$

- Höhe des B\*-Baumes

$$1 + \log_{2k+1} \left( \frac{n}{2k^*} \right) \leq h^* \leq 2 + \log_{k+1} \left( \frac{n}{2k^*} \right) \quad \text{für } h^* \geq 2 .$$



# B- und B\*-Bäume

- Quantitativer Vergleich

- Seitengröße sei L = 2048 B. Zeiger P<sub>i</sub>, Hilfsinformation und Schlüssel K<sub>i</sub> seien 4 B lang. Fallunterscheidung:
- eingebettete Speicherung: l<sub>D</sub> = 76 Bytes
- separate Speicherung: l<sub>D</sub> = 4 Bytes, d.h., es wird nur ein Zeiger gespeichert.

- Allgemeine Zusammenhänge:

	B-Baum	B*-Baum
n <sub>min</sub>	2 · (k + 1) <sup>h-1</sup> - 1	2k* · (k + 1) <sup>h* - 2</sup>
n <sub>max</sub>	(2k + 1) <sup>h</sup> - 1	2k* · (2k + 1) <sup>h* - 1</sup>

- Vergleich für Beispielwerte:

## B-Baum

h	Datensätze separat (k=85)		Datensätze eingebettet (k=12)	
	n <sub>min</sub>	n <sub>max</sub>	n <sub>min</sub>	n <sub>max</sub>
1	1	170	1	24
2	171	29.240	25	624
3	14.791	5.000.210	337	15.624
4	1.272.112	855.036.083	4.393	390.624

## B\*-Baum

h	Datensätze separat (k=127, k* = 127)		Datensätze eingebettet (k=12, k* = 127)	
	n <sub>min</sub>	n <sub>max</sub>	n <sub>min</sub>	n <sub>max</sub>
1	1	254	1	24
2	254	64.770	24	6.120
3	32.512	16.516.350	3.072	1.560.600
4	4.161.536	4.211.669.268	393.216	397.953.001





# Historie und Terminologie

## ■ Originalpublikation B-Baum:

- R. Bayer, E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1:4. 1972. 290-306.

## ■ Überblick:

- D. Comer: The Ubiquitous B-Tree. ACM Computing Surveys, 11:2, Juni 1979, pp. 121-137.

## ■ B\*-Baum Originalpublikation:

D. E. Knuth: The Art of Programming, Vol. 3, Addison-Wesley, 1973.

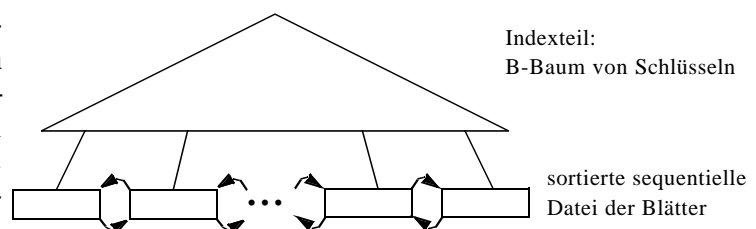
## ■ Terminologie:

- Bei Knuth: B\*-Baum ist ein B-Baum mit garantierter 2 / 3-Auslastung der Knoten
- B+-Baum ist ein Baum wie hier dargestellt
- Heutige Literatur: B\*-Baum = B+-Baum.



# B\*-Bäume: Operationen

- B\*-Baum entspricht einer geketteten sequentiellen Datei von Blättern, die einen Indexteil besitzt, der selbst ein B-Baum ist. Im Indexteil werden insbesondere beim Split-Vorgang die Operationen des B-Baums eingesetzt.

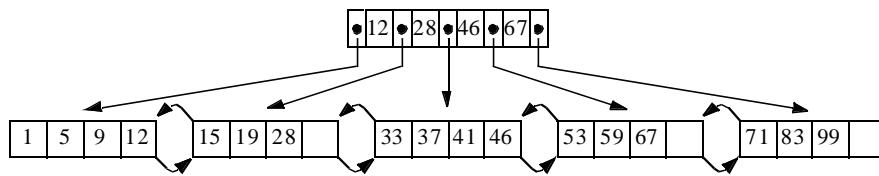


## ■ Grundoperationen beim B\*-Baum

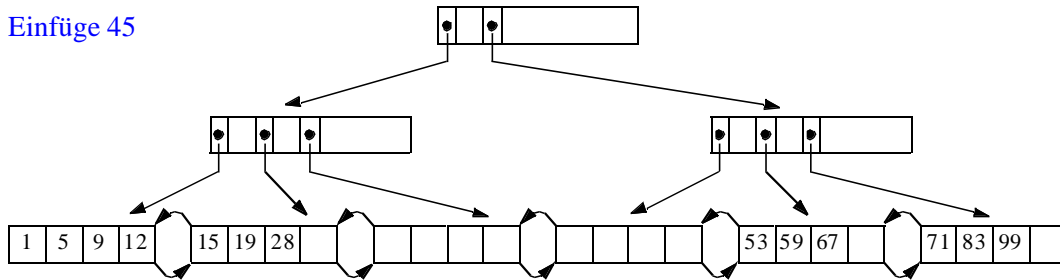
- (1) **Direkte Suche:** Da alle Schlüssel in den Blättern, kostet jede direkte Suche  $h^*$  Zugriffe.  $h^*$  ist jedoch im Mittel kleiner als  $h$  in B-Bäumen (günstigeres  $f_{avg}$  als beim B-Baum)
- (2) **Sequentielle Suche:** Sie erfolgt nach Aufsuchen des Linksaußen der Struktur unter Ausnutzung der Verkettung der Blattseiten. Es sind zwar ggf. mehr Blätter als beim B-Baum zu verarbeiten, doch da nur  $h^*-1$  innere Knoten aufzusuchen sind, wird die sequentielle Suche ebenfalls effizienter ablaufen.
- (3) **Einfügen:** Von Durchführung und Leistungsverhalten dem Einfügen in einen B-Baum sehr ähnlich. Bei inneren Knoten wird die Spaltung analog zum B-Baum durchgeführt. Beim Split-Vorgang einer Blattseite muß gewährleistet sein, daß jeweils die höchsten Schlüssel einer Seite als Wegweiser in den Vaterknoten kopiert werden.
- (4) **Löschen:** Datenelemente werden immer von einem Blatt entfernt (keine komplexe Fallunterscheidung wie beim B-Baum). Weiterhin muß beim Löschen eines Schlüssels aus einem Blatt dieser Schlüssel nicht aus dem Indexteil entfernt werden; er behält seine Funktion als Wegweiser.



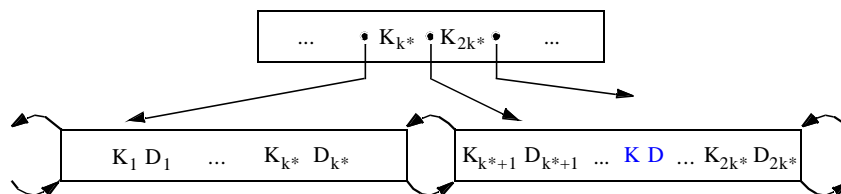
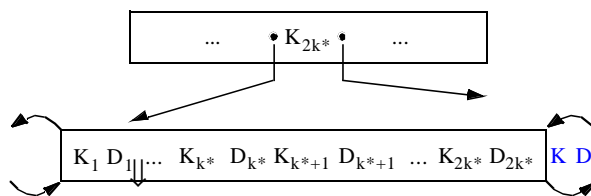
# Einfügen im B\*-Baum



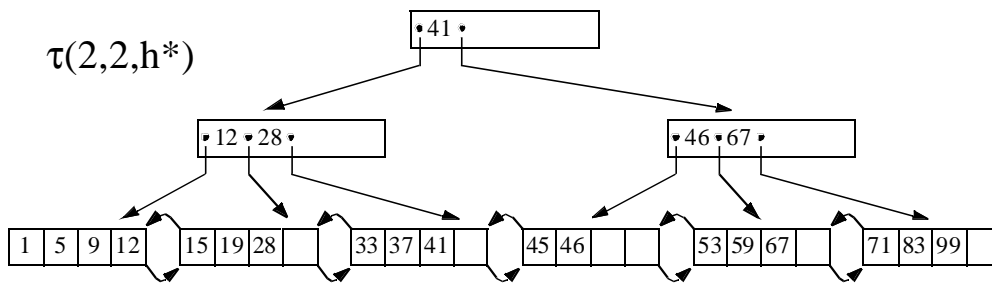
Einfüge 45



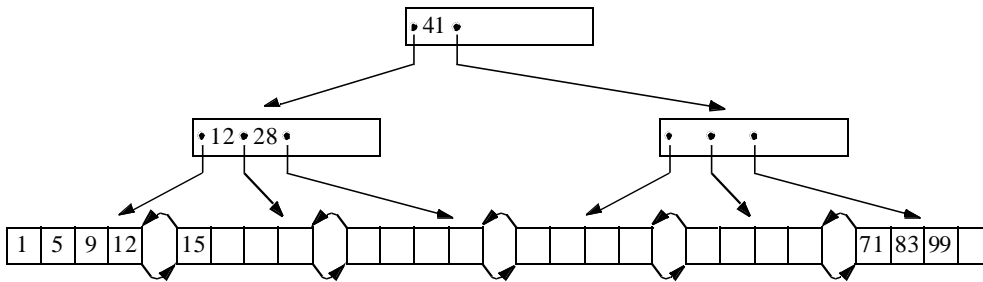
# B\*-Bäume: Schema für Split-Vorgang



# Löschen im B\*-Baum: Beispiel

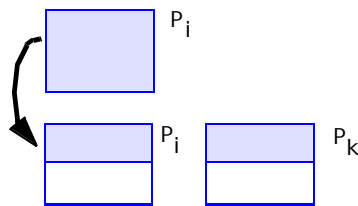


Lösche 28, 41, 46

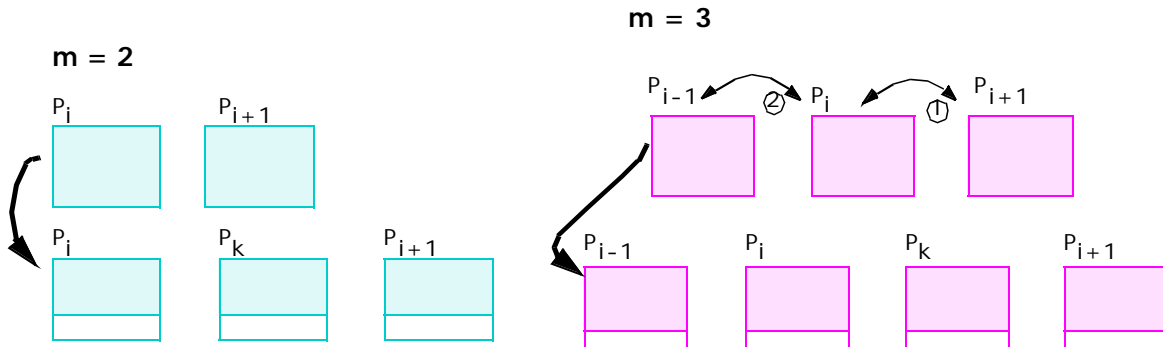


# Verallgemeinerte Überlaufbehandlung

Standard ( $m=1$ ): Überlauf führt zu zwei halb vollen Seiten



$m > 1$ : Verbesserung der Belegung



# Verallgemeinerte Überlaufbehandlung (2)

## ■ Speicherplatzbelegung als Funktion des Split-Faktors

Split-Faktor	Belegung		
	$\beta_{\min}$	$\beta_{\text{avg}}$	$\beta_{\max}$
1	$1/2 = 50\%$	$\ln 2 \approx 69\%$	1
2	$2/3 = 66\%$	$2 \cdot \ln(3/2) \approx 81\%$	1
3	$3/4 = 75\%$	$3 \cdot \ln(4/3) \approx 86\%$	1
m	$\frac{m}{m+1}$	$m \cdot \ln\left(\frac{m+1}{m}\right)$	1

## ■ Vorteile der höheren Speicherbelegung

- geringere Anzahl von Seiten reduziert Speicherbedarf
- geringere Baumhöhe
- geringerer Aufwand für direkte Suche
- geringerer Aufwand für sequentielle Suche

## ■ erhöhter Split-Aufwand ( $m > 3$ i.a. zu teuer)



# Schlüsselkomprimierung

## ■ Zeichenkomprimierung ermöglicht weit höhere Anzahl von Einträgen pro Seite (v.a. bei B\*-Baum)

- Verbesserung der Baumbreite (höherer Fan-Out)
- wirkungsvoll v.a. für lange, alphanumerische Schlüssel (z.B. Namen)

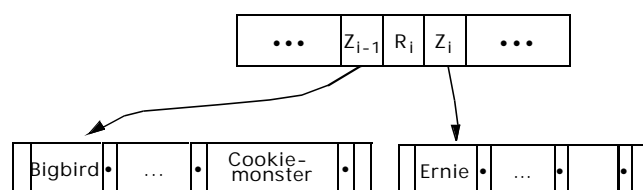
## ■ Präfix-Komprimierung

- mit Vorgängerschlüssel übereinstimmender Schlüsselanfang (Präfix) wird nicht wiederholt
- v.a. wirkungsvoll für Blattseiten
- höherer Aufwand zur Schlüsselrekonstruktion

Schlüssel	F	kompr. Schlüssel
HARALD		
HARTMUT		
HEIN		
HEINRICH		
HEINZ		
HELMUT		
HOLGER		

## ■ Suffix-Komprimierung

- für innere Knoten ist vollständige Wiederholung von Schlüsselwerten meist nicht erforderlich, um Wegweiserfunktion zu erhalten
- Weglassen des zur eindeutigen Lokalisierung nicht benötigten Schlüsselendes (Suffix)
- *Präfix-B-Bäume*: Verwendung minimale Separatoren (Präfixe) in inneren Knoten



## Schlüsselkomprimierung (2)

- für Zwischenknoten kann Präfix- und Suffix-Komprimierung kombiniert werden: *Präfix-Suffix-Komprimierung* (Front and Rear Compression)

- gespeichert werden nur solche Zeichen eines Schlüssels, die sich vom Vorgänger und Nachfolger unterscheiden
- u.a. in VSAM eingesetzt

- **Verfahrensparameter:**

V = Position im Schlüssel, in der sich der zu komprimierende Schlüssel vom *Vorgänger* unterscheidet

N = Position im Schlüssel, in der sich der zu komprimierende Schlüssel vom *Nachfolger* unterscheidet

F = V - 1 (Anzahl der Zeichen des komprimierten Schlüssels, die mit dem Vorgänger übereinstimmen)

L = MAX (N-F, 0) Länge des komprimierten Schlüssels

Schlüssel	V	N	F	L	kompr. Schlüssel
HARALD					
HARTMUT					
HEIN					
HEINRICH					
HEINZ					
HELMUT					
HOLGER					

- Durchschnittl. komprimierte Schlüssellänge ca. 1.3 - 1.8



## Präfix-Suffix-Komprimierung: weiteres Anwendungsbeispiel

*Schlüssel (unkomprimiert)*

Schlüssel (unkomprimiert)	V	N	F	L	Wert
CITY_OF_NEW_ORLEANS ... GUTHERIE, ARLO	1	6	0	6	CITY_O
CITY_TO_CITY ... RAFFERTTY, GERRY	6	2	5	0	
CLOSET_CHRONICLES ... KANSAS	2	2	1	1	L
COCAINE ... CALE, J.J	2	3	1	2	OC
COLD_AS_ICE ... FOREIGNER	3	6	2	4	LD_A
COLD_WIND_TO_WALHALLA ... JETHRO_TULL	6	4	5	0	
COLORADO ... STILLS, STEPHEN	4	5	3	2	OR
COLOURS ... DONOVAN	5	3	4	0	
COME_INSIDE ... COMMODORES	3	13	2	11	ME_INSIDE__
COME_INSIDE_OF_MY_GUITAR ... BELLAMY_BROTHERS	13	6	12	0	
COME_ON_OVER ... BEE_GEES	6	6	5	1	O
COME_TOGETHER ... BEATLES	6	4	5	0	
COMING_INTO_LOS_ANGELES ... GUTHERIE, ARLO	4	4	3	1	I
COMMOTION ... CCR	4	4	3	1	M
COMPARED_TO_WHAT? ... FLACK, ROBERTA	4	3	3	0	
CONCLUSION ... ELP	3	4	2	2	NC
CONFUSION ... PROCOL_HARUM	4	1	3	0	



# 2-3-Bäume

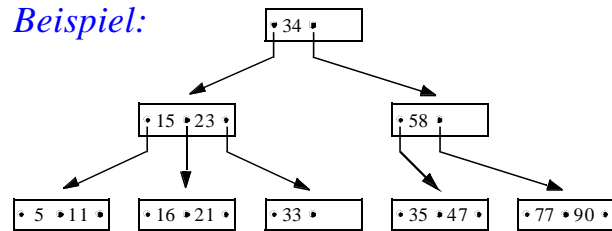
■ B-Bäume können auch als Hauptspeicher-Datenstruktur verwendet werden

- möglichst kleine Knoten wichtiger als hohes Fan-Out
- 2-3 Bäume: B-Bäume der Klasse  $\tau(1,h)$ , d.h. mit minimalen Knoten
- ⇒ Es gelten alle für den B-Baum entwickelten Such- und Modifikationsalgorithmen

■ Ein 2-3-Baum ist ein m-Wege-Suchbaum ( $m=3$ ), der entweder leer ist oder die Höhe  $h \geq 1$  hat und folgende Eigenschaften besitzt:

- Alle Knoten haben einen oder zwei Einträge (Schlüssel).
- Alle Knoten außer den Blattknoten besitzen 2 oder 3 Söhne.
- Alle Blattknoten sind auf derselben Stufe.

Beispiel:



■ Beobachtungen

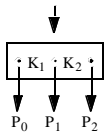
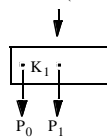
- 2-3-Baum ist balancierter Baum
- ähnliche Laufzeitkomplexität wie AVL-Baum
- schlechte Speicherplatznutzung (besonders nach Höhenänderung)



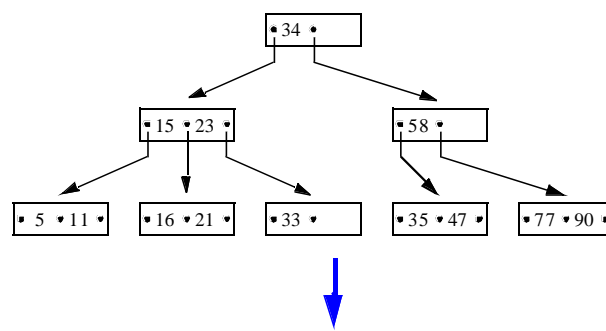
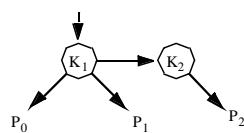
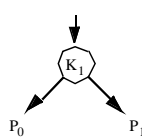
# Binäre B-Bäume

■ Verbesserte Speicherplatznutzung gegenüber 2-3-Bäumen durch Speicherung der Knoten als gekettete Listen mit einem oder zwei Elementen:

B-Baum ( $k = 1$ )



binärer B-Baum



■ Variante: symmetrischer binärer B-Baum



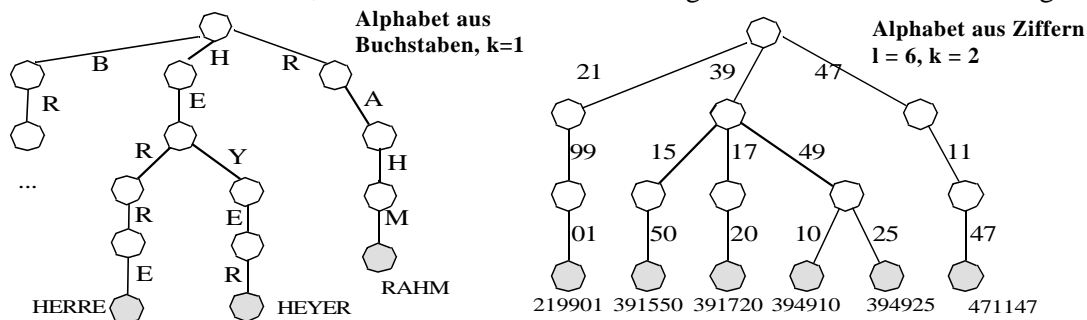
# Digitale Suchbäume

## Prinzip des digitaler Suchbäume (kurz: Digitalbäume)

- Zerlegung des Schlüssels - bestehend aus Zeichen eines Alphabets - in Teile
- Aufbau des Baumes nach Schlüsselteilen
- Suche im Baum durch Vergleich von Schlüsselteilen
- jede unterschiedliche Folge von Teilschlüsseln ergibt eigenen Suchweg im Baum
- alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg
- vorteilhaft u.a. bei variabel langen Schlüsseln, z.B. Strings

## Was sind Schlüsselteile ?

- Schlüsselteile können gebildet werden durch Elemente (Bits, Ziffern, Zeichen) eines Alphabets oder durch Zusammenfassungen dieser Grundelemente (z. B. Silben der Länge k)
- Höhe des Baumes =  $l/k + 1$ , wenn l die max. Schlüssellänge und k die Schlüsselteillänge ist



# m-ärer Trie

## Spezielle Implementierung des Digitalbaumes: Trie

- Trie leitet sich von Information Retrieval ab (E.Fredkin, 1960)
- spezielle m-Wege-Bäume, wobei Kardinalität des Alphabets und Länge k der Schlüsselteile den Grad m festlegen
  - bei Ziffern:  $m = 10$
  - bei Alpha-Zeichen:  $m = 26$ ; bei alphanumerischen Zeichen:  $m = 36$
  - bei Schlüsselteilen der Länge k potenziert sich Grad entsprechend, d. h. als Grad ergibt sich  $m^k$

## Trie-Darstellung

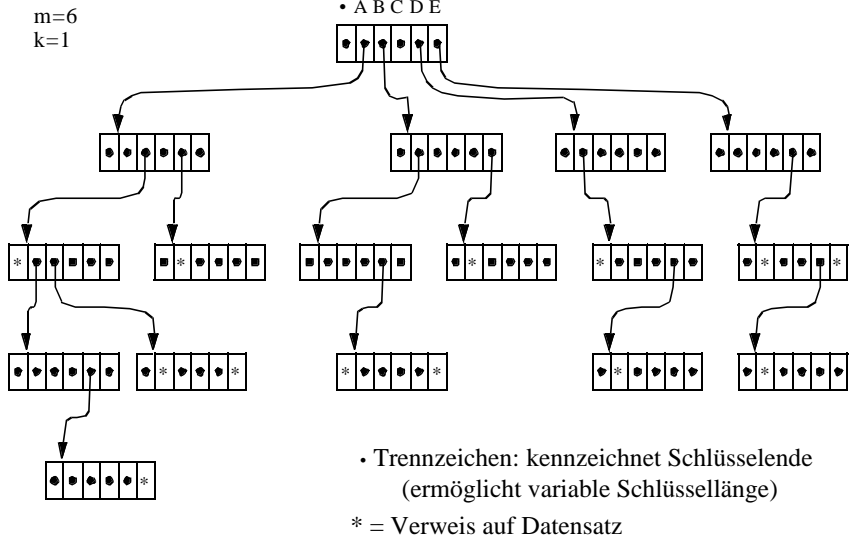
- Jeder Knoten eines Tries vom Grad m ist im Prinzip ein eindimensionaler Vektor mit m Zeigern
- Jedes Element im Vektor ist einem Zeichen (bzw. Zeichenkombination) zugeordnet. Auf diese Weise wird ein Schlüsselteil (Kante) implizit durch die Vektorposition ausgedrückt.
- Beispiel: Knoten eines 10-ären Trie mit Ziffern als Schlüsselteilen
 

$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$	$P_8$	$P_9$
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
- implizite Zuordnung von Ziffer/Zeichen zu Zeiger.  $P_i$  gehört also zur Ziffer i. Tritt Ziffer i in der betreffenden Position auf, so verweist  $P_i$  auf den Nachfolgerknoten. Kommt i in der betreffenden Position nicht vor, so ist  $P_i$  mit NULL belegt
- Wenn der Knoten auf der j-ten Stufe eines 10-ären Trie liegt, dann zeigt  $P_i$  auf einen Unterbaum, der nur Schlüssel enthält, die in der j-ten Position die Ziffer i besitzen

## ■ Grundoperationen

- **Direkte Suche:** In der Wurzel wird nach dem 1. Zeichen des Suchschlüssels verglichen. Bei Gleichheit wird der zugehörige Zeiger verfolgt. Im gefundenen Knoten wird nach dem 2. Zeichen verglichen usw.
  - Aufwand bei erfolgreicher Suche:  $\lfloor l/k \rfloor + 1$  bei Präfix)
  - effiziente Bestimmung der Abwesenheit eines Schlüssels (z. B. CAD)
- **Einfügen:** Wenn Suchpfad schon vorhanden, wird NULL-Zeiger in \*-Zeiger umgewandelt, sonst Einfügen von neuen Knoten (z. B. CAD)
- **Löschen:** Nach Aufsuchen des richtigen Knotens wird ein \*-Zeiger auf NULL gesetzt. Besitzt daraufhin der Knoten nur NULL-Zeiger, wird er aus dem Baum entfernt (rekursive Überprüfung der Vorgängerknoten)
- Sequentielle Suche ?

Beispiel: Trie für Schlüssel aus einem auf A-E beschränkten Alphabet



## m-ärer Trie (3)

### ■ Beobachtungen:

- Höhe des Trie wird durch den längsten abgespeicherten Schlüssel bestimmt
- Gestalt des Baumes hängt von der Schlüsselmenge, also von der Verteilung der Schlüssel, nicht aber von der Reihenfolge ihrer Abspeicherung ab
- Knoten, die nur NULL-Zeiger besitzen, werden nicht angelegt

### ■ dennoch schlechte Speicherplatzausnutzung

- dünn besetzte Knoten
- viele Einweg-Verzweigungen (v.a. in der Nähe der Blätter)

### ■ Möglichkeiten der Kompression

- Sobald ein Zeiger auf einen Unterbaum mit nur einem Schlüssel verweist, wird der (Rest-)Schlüssel in einem speziellen Knotenformat aufgenommen und Unterbaum eingespart  
-> vermeidet Einweg-Verzweigungen
- nur besetzte Verweise werden gespeichert (erfordert Angabe des zugehörigen Schlüsselteils)

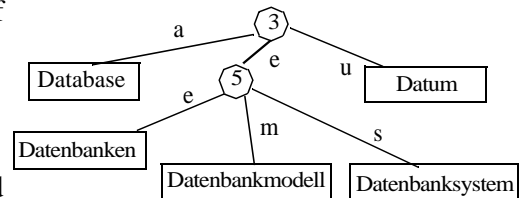
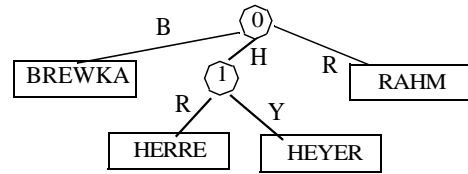


# PATRICIA-Baum

(Practical Algorithm To Retrieve Information Coded In Alphanumeric)

## Merkmale

- Binärdarstellung für Schlüsselwerte -> binärer Digitalbaum
- Speicherung der Schlüssel in den Blättern
- **innere Knoten** speichern, wieviele Zeichen (Bits) beim Test zur Wegeauswahl zu überspringen sind
- Vermeidung von Einwegverzweigungen, in dem bei nur noch einem verbleibenden Schlüssel direkt auf entsprechendes Blatt verwiesen wird



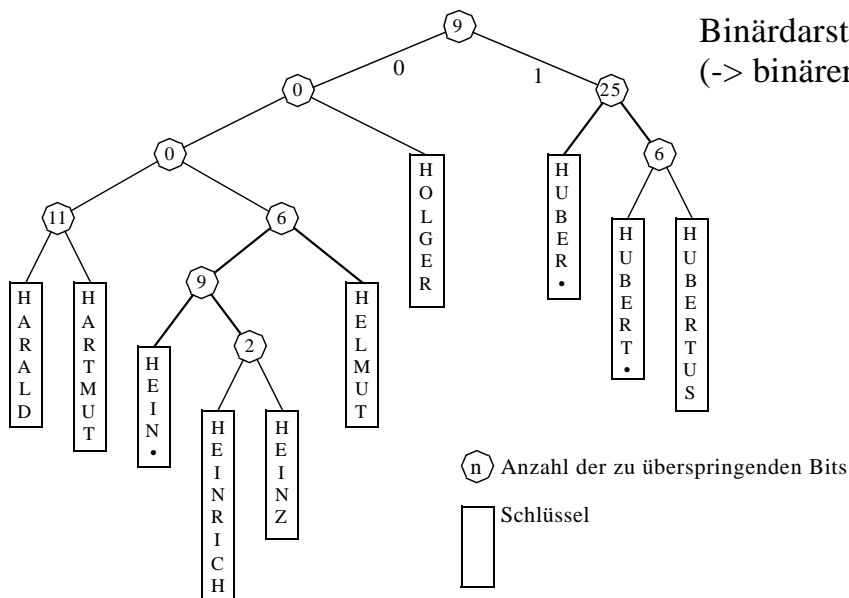
## Bewertung

- speichereffizient
- sehr gut geeignet für variabel lange Schlüssel und (sehr lange) Binärdarstellungen von Schlüsselwerten
- bei jedem Suchschlüssel muß die Testfolge von der Wurzel beginnend ganz ausgeführt werden, bevor über Erfolg oder Mißerfolg der Suche entschieden werden kann



# PATRICIA-Baum (2)

Binärdarstellung  
(-> binärer Digitalbaum)



HANS = 1001000...  
 HEINZ = 1001000...  
 HOLLGER = 1001000...  
 Bert = 1000010...  
 ...  
 OTTO = 1001111...  
 ...

$\odot n$  Anzahl der zu überspringenden Bits

$\square$  Schlüssel

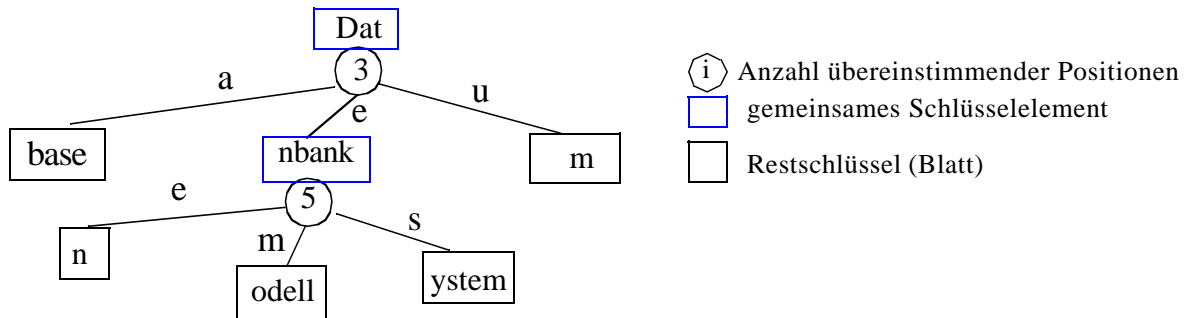
- Suche nach dem Schlüssel HEINZ = X'10010001000101100100110011101011010' ?
  - Suche nach ABEL = X'1000001100001010001011001100' ?
- ⇒ erfolgreiche und erfolglose Suche endet in einem Blattknoten



# Präfix- bzw. Radix-Baum

## ■ (Binärer) Digitalbaum als Variante des PATRICIA-Baumes

- Speicherung variabel langer Schlüsselteile in den inneren Knoten, sobald sie sich als Präfixe für die Schlüssel des zugehörigen Unterbaums abspalten lassen
- komplexere Knotenformate und aufwendigere Such- und Aktualisierungsoperationen
- erfolglose Suche läßt sich oft schon in einem inneren Knoten abbrechen



# Zusammenfassung

## ■ Konzept des Mehrwegbaumes:

- Aufbau sehr breiter Bäume von geringer Höhe
- Bezugsgröße: Seite als Transporteinheit zum Externspeicher
- Seiten werden immer größer, d. h., das Fan-out wächst weiter

## ■ B- und B\*-Baum gewährleisten eine balancierte Struktur

- unabhängig von Schlüsselmenge
- unabhängig ihrer Einfügereihenfolge

## ■ Wichtigste Unterschiede des B\*-Baums zum B-Baum:

- strikte Trennung zwischen Datenteil und Indexteil. Datenelemente stehen nur in den Blättern des B\*-Baumes
- Schlüssel innerer Knoten haben nur Wegweiserfunktion. Sie können auch durch beliebige Trenner ersetzt oder durch Komprimierungsalgorithmen verkürzt werden
- kürzere Schlüssel oder Trenner in den inneren Knoten erhöhen Verzweigungsgrad des Baumes und verringern damit seine Höhe
- die redundant gespeicherten Schlüssel erhöhen den Speicherplatzbedarf nur geringfügig (< 1%)
- Löschalgorithmus ist einfacher
- Verkettung der Blattseiten ergibt schnellere sequentielle Verarbeitung

# Zusammenfassung (2)

- Standard-Zugriffspfadstruktur in DBS: B\*-Baum
- verallgemeinerte Überlaufbehandlung verbessert Seitenbelegung
- Schlüsselkomprimierung
  - Verbesserung der Baumbreite
  - Präfix-Suffix-Komprimierung sehr effektiv
  - Schlüssellängen von 20-40 Bytes werden im Mittel auf 1.3-1.8 Bytes reduziert
- Binäre B-Bäume: Alternative zu AVL-Bäumen als Hauptspeicher-Datenstruktur
- Digitale Suchbäume: Verwendung von Schlüsselteilen
  - Unterstützung von Suchvorgängen u.a. bei langen Schlüsseln variabler Länge
  - wesentliche Realisierungen: PATRICIA-Baum / Radix-Baum

