6. Baumbasierte Zugriffspfade

- **Ziele**
  - Entwurfsprinzipien für Zugriffspfade auf die Sätze einer Tabelle, bei denen ein Suchkriterium unterstützt wird
  - Abbildungsmöglichkeiten für hierarchische Zugriffsanforderungen
- **Anforderungen und Klassifikation**
- **Zugriffspfade für Primärschlüssel**
  - Binäre Suchbäume?
  - Mehrwegbäume und Digitalbäume
  - Hash-Verfahren (Kapitel 7)
- **B- und B*-Bäume**
  (Wiederholung)
- **Digitalbäume**
  - m-ärer Trie
  - Binärer Digitalbaum
- **Adressierung in Bäumen**
  - wichtig für die fein-granulare Abbildung von XML-Dokumenten
  - Nummerierungsschema für Knoten soll die Dokumentenstruktur und -ordnung berücksichtigen und bei beliebigen Einfügungen Renummerierung vermeiden
  - Unterstützung von Navigation, deklarativer Anfrageauswertung und Sperren
- **Wichtige Kenngrößen:**
  - n = #Sätze eines Satztyps
  - b = mittlere #Sätze/Seite (Blockungsfaktor)
  - q = #Treffer der Anfrage
  - N_S = #Seitenzugriffe
  - N_B = #Blattseiten des B*-Baums
  - h_B = Höhe des B*-Baums

Anforderungen an Zugriffspfade

- **Folgende Arten von Zugriffen müssen unterstützt werden:**
  - Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)

    Select * From Pers
  - Sequentieller Zugriff in Sortierreihenfolge eines Attributes

    ... Order by Name
  - Direkter Zugriff über den Primärschlüssel

    ... Where Pnr = 0815
  - Direkter Zugriff über einen Sekundärschlüssel

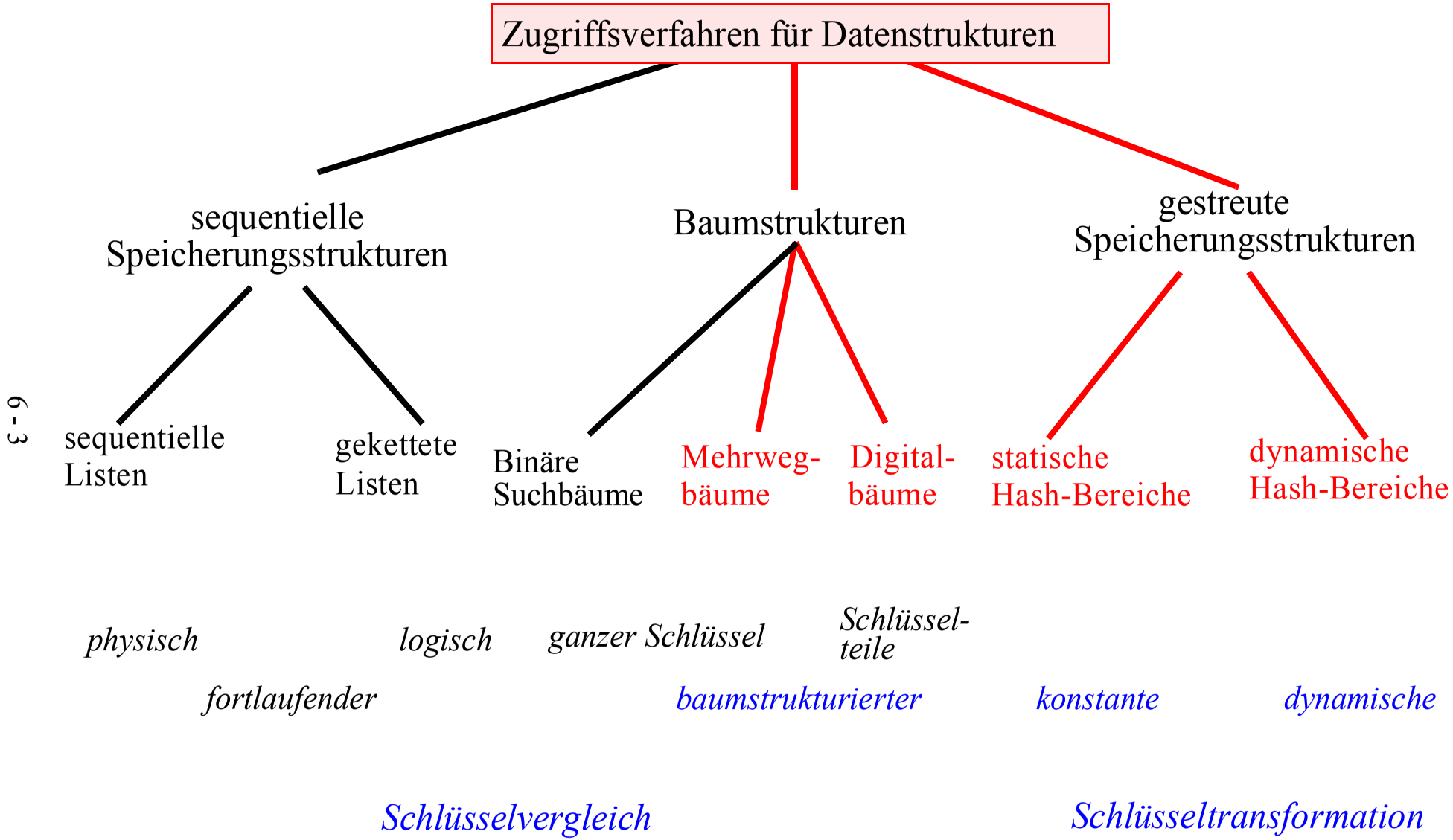
    ... Where Beruf = 'Programmierer'
  - Direkter Zugriff über zusammengesetzte Schlüssel und komplexe Suchausdrücke (Wertintervalle, ...)

    ... Where Gehalt Between 50K And 100K
  - Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge desselben oder eines anderen Satztyps

    ... Where P.Pnr = A.Pnr

➥ **Wenn kein geeigneter Zugriffspfad vorhanden ist, sind alle Zugriffsarten durch fortlaufende Suche (Scan) abzuwickeln**

- **Scan**
  - muß von allen DBMS unterstützt werden!
  - ist ausreichend / effizient bei:
    - kleinen Satztypen (z. B. ≤ 5 Seiten)
    - Anfragen mit großen Treffermengen (z. B. > 3%)
  - DBMS kann Prefetching zur Scan-Optimierung nutzen


Zugriffsverfahren für Datenstrukturen
sequentielle Speicherungsstrukturen
Baumstrukturen
gestreute Speicherungsstrukturen
sequentielle Listen
gekettete Listen
Binäre Suchbäume
Mehrweg-bäume
Digital-bäume
statische Hash-Bereiche
dynamische Hash-Bereiche
physisch
logisch
ganzer Schlüssel
Schlüssel-teile
fortlaufender
baumstrukturierter
konstante
dynamische
Schlüsselvergleich
Schlüsseltransformation

Binäre Suchbäume

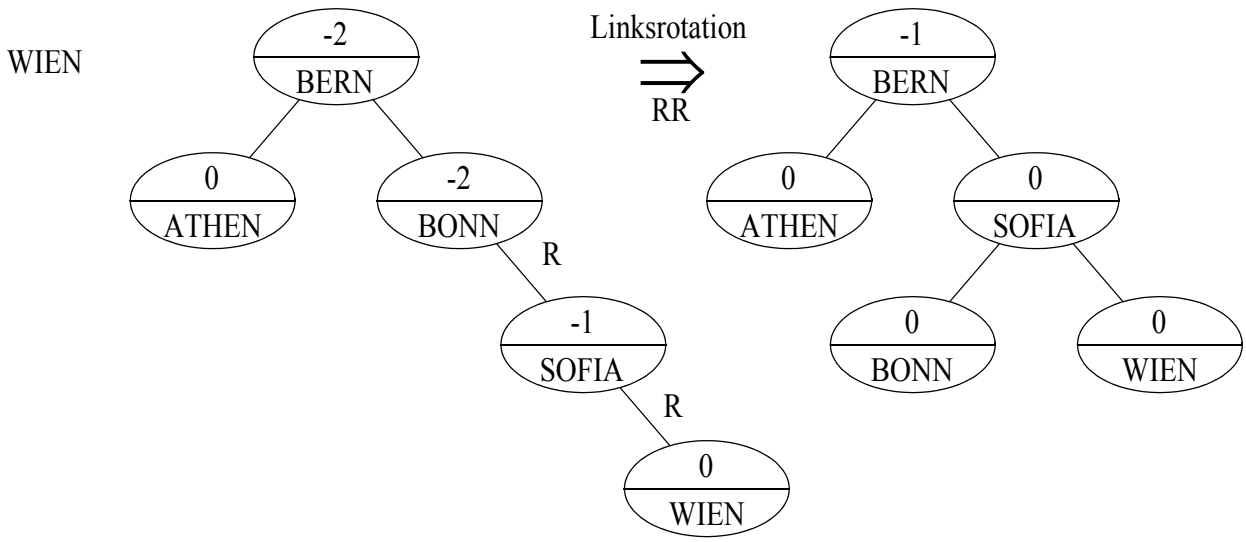
- **Balancierte binäre Suchbäume**

Definition: Seien $B_l(x)$ und $B_r(x)$ die linken und rechten Unterbäume eines Knotens x. Weiterhin sei h(B) die Höhe eines Baumes B. Ein **k-balancierter binärer Suchbaum** ist entweder leer oder es ist ein Suchbaum, bei dem für jeden Knoten x gilt:

$$\left| h(B_l(x)) - h(B_r(x)) \right| \leq k$$

Definition: Ein 1-balancierter binärer Suchbaum heißt **AVL-Baum.**

- **Beispiel: AVL-Baum**



- **Satz**

  Für die Höhe h_b eines AVL-Baumes mit n Knoten gilt:

$$\lfloor \log_2(n) \rfloor + 1 \leq h_b \leq 1{,}44 \cdot \log_2(n+1)$$

Mehrwegbäume

- **Bezugsgröße: Seite = Transporteinheit zum Externspeicher**

  im Gegensatz zu binären Suchbäumen

- **Vorfahr: ISAM (statisch, periodische Reorganisation)**

- **Weiterentwicklung: B- und B*-Baum**
  - referenzierte und materialisierte Speicherung der Datensätze
  - dynamische Reorganisation durch Splitten und Mischen von Seiten

- **Funktion**
  - direkter Schlüsselzugriff
  - sortiert sequentieller Zugriff

- **Balancierte Struktur**
  - unabhängig von Schlüsselmenge
  - unabhängig von Einfügereihenfolge

- **Realisierung von Index-organisierten Tabellen**
  - oft nach Primärschlüssel geordnet
  - Cluster-Bildung durch eingebettete Datensätze

- **Verbesserung der Baumbreite (fan-out)**
  - Schlüsselkomprimierung
  - Nutzung von „Wegweisern" in B*-Bäumen
  - Präfix-B-Bäume

- **Verbesserung des Belegungsgrades**

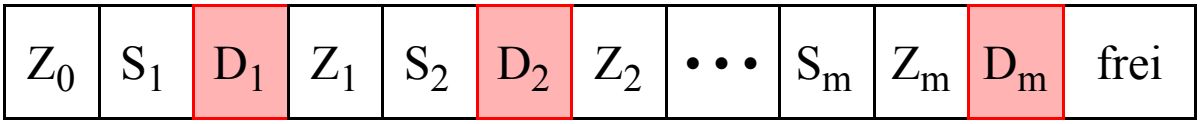
  ➥ verallgemeinertes Splittingverfahren

B-Baum

- **Def.: Ein B-Baum vom Typ (k, h) ist ein Baum mit folgenden Eigenschaften**

  1. Jeder Weg von der Wurzel zum Blatt hat die Länge h
  2. Jeder Zwischenknoten hat mindestens k+1 Söhne.
     Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne
  3. Jeder Knoten hat höchstens 2k+1 Söhne

- **Seitenformat:**

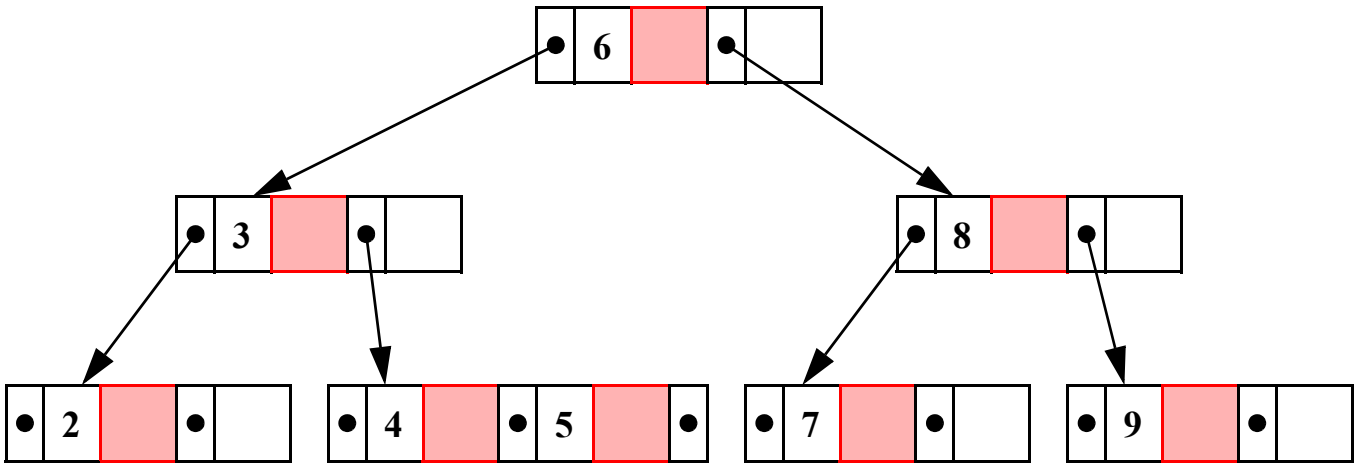


Z_i = Zeiger Sohnseite

S_i = Schlüssel

D_i = Daten des Satzes oder Verweis auf den Satz
(materialisiert oder referenziert)

- **Beispiel:**



bei 4 KB Seiten:

Z=4 B, S=4 B, D=92 B ⇒ 100 B pro Eintrag ⇒ ca. 40 Söhne

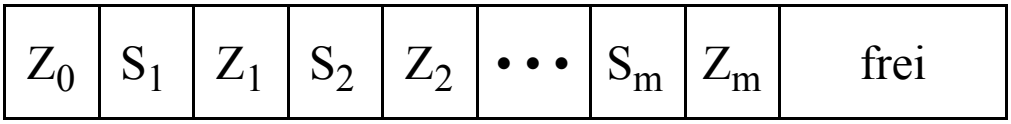
Z=4 B, S=4 B, D=4 B ⇒ 12 B pro Eintrag ⇒ ca. 330 Söhne

B*-Baum

- **Def.: Ein B*-Baum vom Typ (k, k*, h) ist ein Baum mit folg. Eigenschaften**

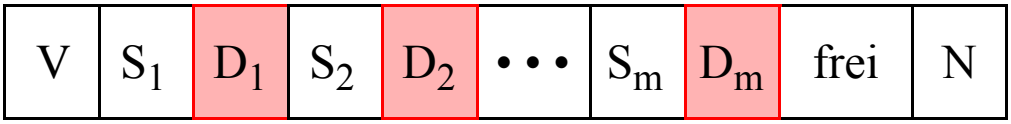
  1. Jeder Weg von der Wurzel zum Blatt hat die Länge h
  2. Jeder Zwischenknoten hat mindestens k+1 Söhne.
     Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne.
     Jedes Blatt hat mindestens k* Einträge.
  3. Jeder Zwischenknoten hat höchstens 2k+1 Söhne.
     Jedes Blatt hat höchstens 2k* Einträge.

- **Zwischenknoten:**



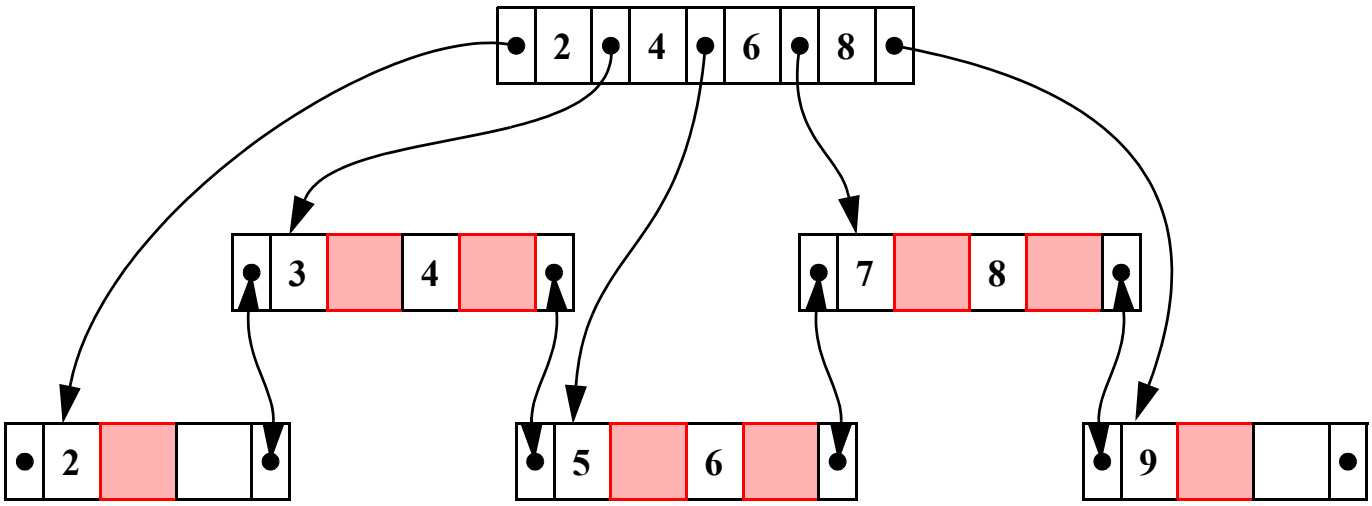
Z_i = Zeiger Sohnseite, S_i = Schlüssel

- **Blattknoten:**



D_i = Verweis auf Satz (materialisiert oder referenziert)

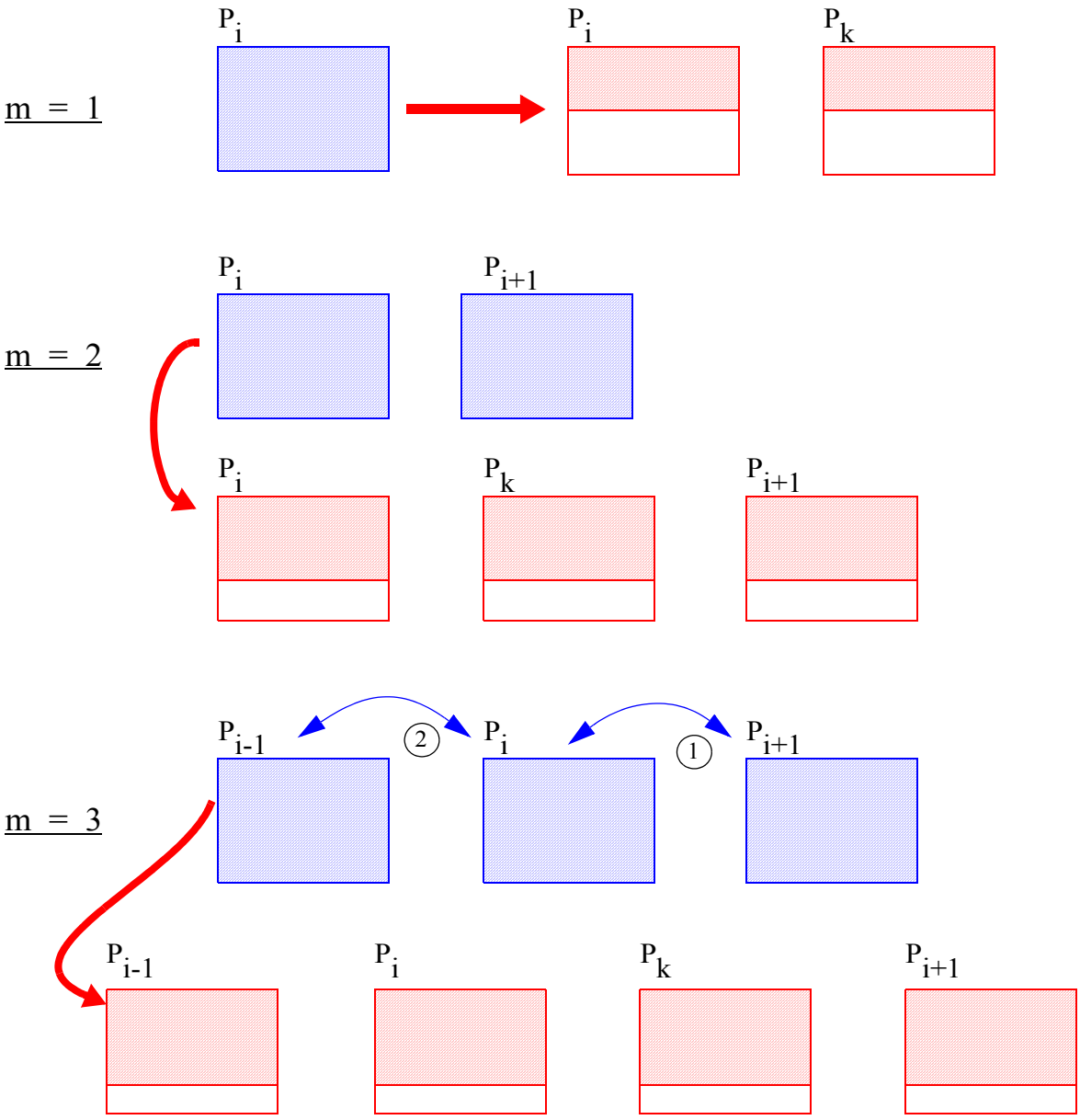
N = Nachfolger-Zeiger, V = Vorgänger-Zeiger



Z=4 B, S=4 B ⇒ 8 B pro Eintrag ⇒ ca. 500 Söhne bei 4 KB Seite

Splitting bei B*-Bäumen

- **Split-Faktor m**



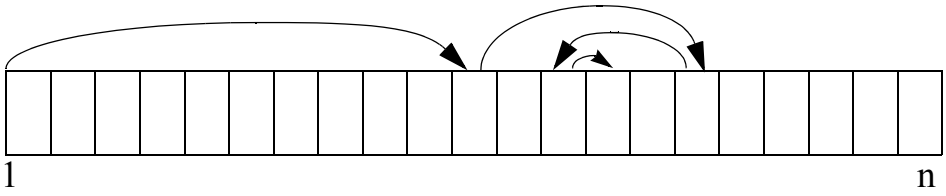
- **Belegung**

Belegung	m = 1	m = 2	m
worst case:	$\frac{1}{1+1}$	$\frac{2}{2+1}$	$\frac{m}{m+1}$
avg. case:	ln 2 (69 %)		$m \cdot \ln\left(\frac{m+1}{m}\right)$

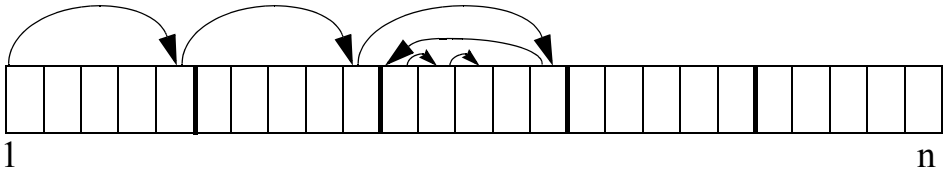
➥ **m ≤ 3: sonst zu aufwendig!**

Suche in der Seite

- **Interne Struktur sei eine Liste mit n Einträgen**
  - **sequentielle Suche**
    - sortierte oder ungeordnete Schlüsselmenge
    - $C_{avg}\ (n) \approx n/2$
    - nur geringe Verbesserungen auf sortierten Listen (bei erfolgloser Suche)
  - **Binärsuche** wesentlich effizienter (Divide-and-Conquer-Strategie)



    - Voraussetzung: Sortierung und Einträge fester Länge
    - $C_{avg}\ (n) \approx\ log_2\ (n+1)\ -\ 1$ für große n
  - **Sprungsuche**
    - Voraussetzung: Sortierung und Einträge fester Länge
    - Prinzip



    - zunächst wird Liste in Sprüngen von m Einträgen überquert, um Abschnitt zu lokalisieren, der ggf. den gesuchten Schlüssel enthält
    - danach wird der Schlüssel im gefundenen Abschnitt nach irgendeinem Verfahren gesucht
    - $C_{avg}(n) = \frac{1}{2}a \cdot \frac{n}{m} + \frac{1}{2}b(m-1)$

      wenn ein Sprung a und ein sequentieller Vergleich b Einheiten kostet
    - Was ist die optimale Sprungweite m?

Digitalbäume

- **Bisher: stets Vergleich des ganzen Schlüssels**

  Bei den digitalen Suchbäumen oder kurz Digitalbäumen erfolgen die Vergleiche in den Baumknoten zur Bestimmung des weiteren Suchweges nicht nach dem ganzen Schlüssel, sondern jeweils nach aufeinanderfolgenden Teilen des Schlüssels. Jede unterschiedliche Folge von Teilschlüsseln ergibt einen eigenen Suchweg im Baum; alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg.

  ➥ Organisation des Digitalbaumes und Suche im Baum erfolgt nach Schlüsselteilen

- **Digitale Suchbäume - Prinzip**

- **m-ärer Trie**

  allgemeines Alphabet

  - Trie-Darstellung
  - Grundoperationen
  - Verbesserung der Platzausnutzung
  - Digitalbaum mit variablem Knotenformat

- **Binärer Digitalbaum**

  binäres Alphabet

  - Binärer digitaler Suchbaum
  - PATRICIA-Baum
    Vermeidung von Einwegverzweigungen
  - Binärer Radix-Baum
    Verbesserung der Suchmöglichkeiten

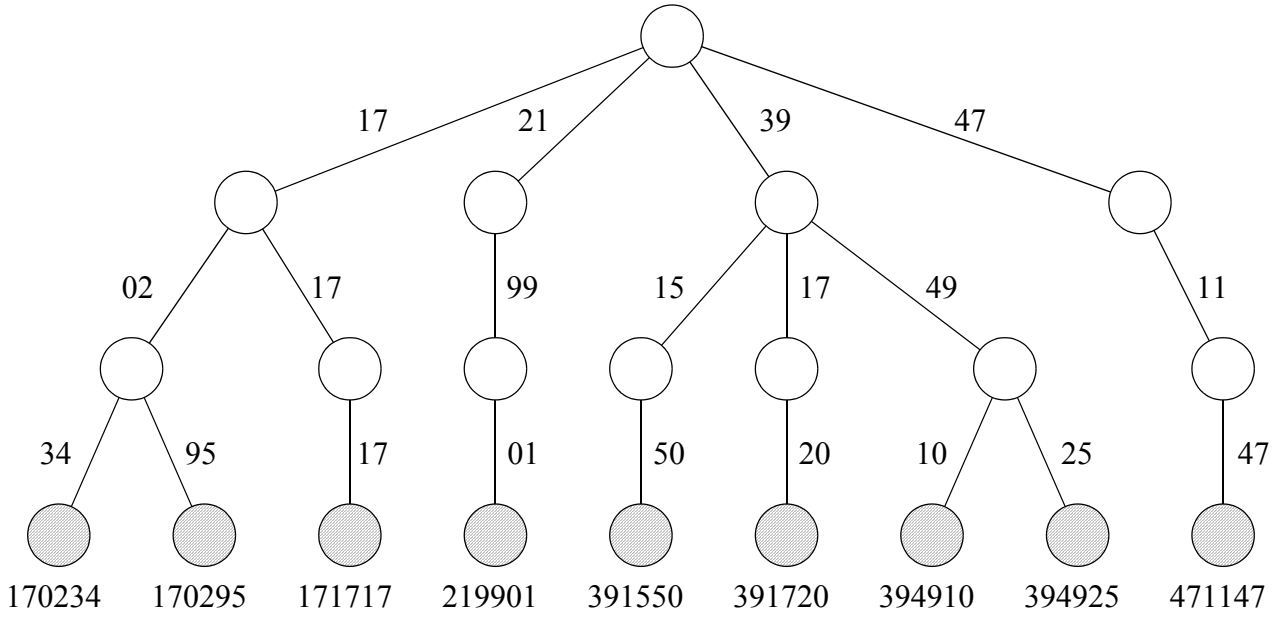
Digitalbaum

- **Prinzip**
  - Zerlegung des Schlüssels in Teile
  - Aufbau des Baumes nach Schlüsselteilen
  - Suche im Baum durch Vergleich von Schlüsselteilen

- **Was sind Schlüsselteile ?**
  - Schlüssel bestehe aus L Zeichen eines Alphabets
  - Schlüsselteile können gebildet werden durch Bits, Ziffern, Zeichen als Elemente eines Alphabets
  - aber auch Zusammenfassungen dieser Grundelemente können herangezogen werden (z. B. Silben der Länge k)
  - längster Pfad im Baum + 1 = Höhe des Baumes = L/k + 1,
  - wenn L die Schlüssellänge und k die Schlüsselteillänge ist.

- **Konzeptionelle Darstellung eines Digitalbaumes**

Alphabet aus Ziffern
L = 6, k = 2



➥ max. Grad des Digitalbaumes m = 100

m-ärer Trie

- **Spezielle Implementierung des Digitalbaumes: Trie**

  Trie leitet sich von Information Retrieval ab (E.Fredkin, 1960)

  Verwendetes Alphabet der Schlüssel legt den Grad eines Trie fest:

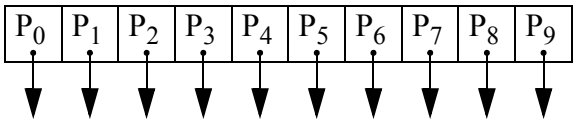
  - bei Ziffern: m = 10
  - bei Alpha-Zeichen: m = 26
  - bei alphanumerischen Zeichen: m = 36

  ➥ bei Schlüsselteilen der Länge k potenziert sich der Grad entsprechend, d. h., als Grad ergibt sich m^k

- **Trie-Darstellung**

  - Der Grad m eines Trie wird durch Kardinalität des Alphabets und Länge k des Schlüsselteils festgelegt.
  - Jeder Knoten eines Tries vom Grad m ist im Prinzip ein eindimensionaler Vektor mit m Zeigern.
  - Jedes Element im Vektor ist einem Zeichen des verwendeten Alphabets zugeordnet. Auf diese Weise wird ein Schlüsselteil (Kante) implizit durch die Vektorposition ausgedrückt.
  - Beispiel: Knoten eines 10-ären Trie mit Ziffern als Schlüsselteilen

    m=10
    k=1

  - Es existiert eine implizite Zuordnung von Ziffer (oder Zeichen) zu Zeiger. P_i gehört also zur Ziffer i. Tritt Ziffer i in der betreffenden Position auf (ist also Kante i in der konzeptionellen Darstellung vorhanden), so verweist P_i auf den Nachfolgerknoten. Kommt i in der betreffenden Position nicht vor, so ist P_i mit NIL 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.

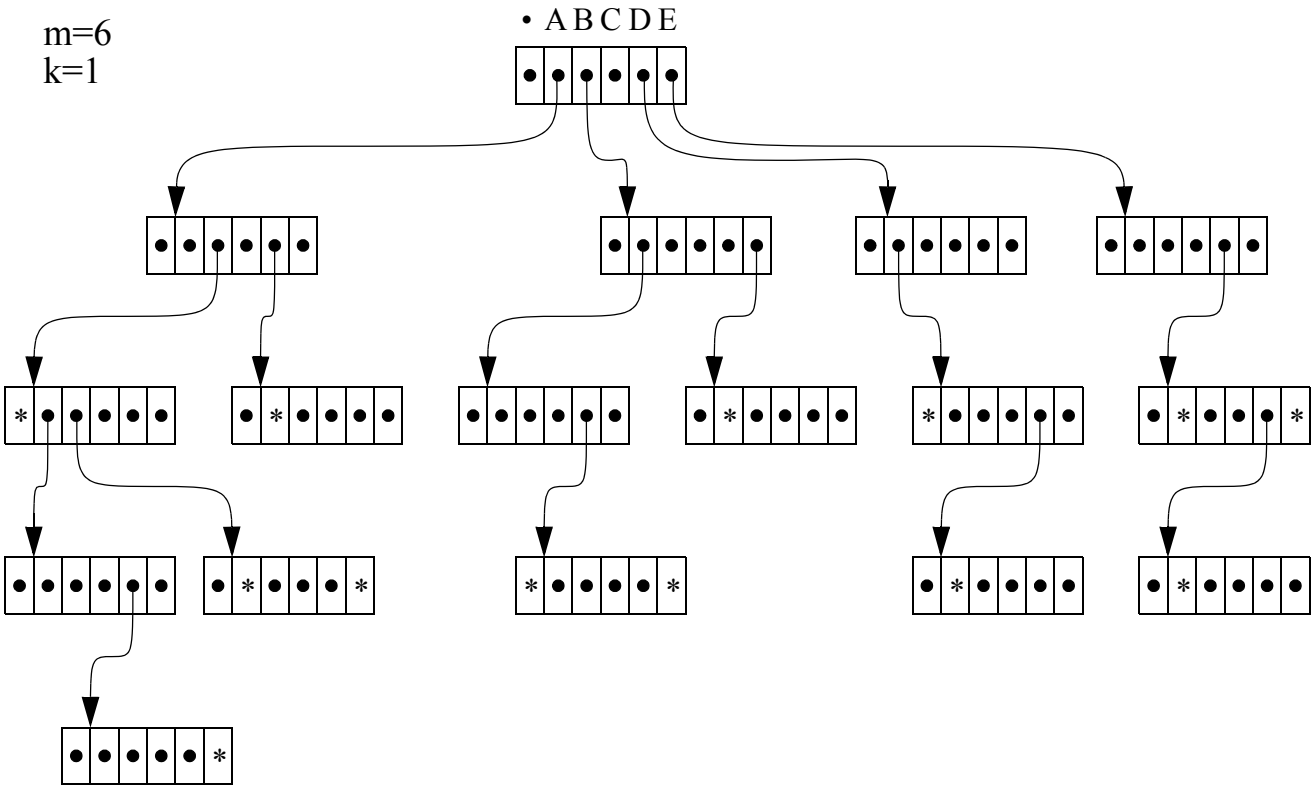
m-ärer Trie (2)

- **Grundversion des Trie**
  - Alle Schlüssel besitzen gleiche Länge. Dann hat der Trie eine ähnliche Struktur wie der B*-Baum: die inneren Knoten dienen als Index und von den Blattknoten aus wird auf die Datensätze verwiesen.

- **Flexiblere Struktur des Trie**
  - Spezielles Trennzeichen (Leerzeichen oder Punkt) im verwendeten Alphabet ermöglicht es, Schlüssel, die Präfix eines anderen Schlüssels sind, im Trie zu speichern. Beispielsweise wird der Schlüssel AB im Trie durch AB**.** dargestellt, um seinen Suchweg von dem des Schlüssels ABBA zu unterscheiden.

    ➥ uneingeschränkte Speicherung von variabel langen Schlüsseln

- **Trie** für Schlüssel aus einem auf A-E beschränkten Alphabet



➥ Welche Schlüssel sind im Trie dargestellt ?

m-ärer Trie (3)

- **Verweis auf Datensätze:**

  - Die mit * gekennzeichneten Zeiger - jeweils am Ende eines Suchpfades - können entweder Zeiger auf den zugehörigen Datensatz sein oder Platzhalter, die anzeigen, daß der zugehörige Schlüssel gültig ist und existiert.

- **Beobachtungen:**

  - Die Höhe des Trie wird durch den längsten abgespeicherten Schlüssel bestimmt.
  - Die Gestalt des Baumes hängt von der Schlüsselmenge, also von der Verteilung der Schlüssel, nicht aber von der Reihenfolge ihrer Abspeicherung ab.
  - Knoten, die nur NIL-Zeiger besitzen, werden nicht angelegt.
  - Wegen der impliziten Zeigerzuordnung muß für jedes Zeichen in jedem Knoten Platz reserviert werden.
  - Zu den Blättern hin treten sehr viele Einweg-Verzweigungen auf.

- **Grundoperationen**

  (1) Direkte Suche:

  In der Wurzel wird nach dem ersten Zeichen des Suchschlüssels verglichen. Bei Gleichheit wird der zugehörige Zeiger verfolgt. Im gefundenen Knoten wird nach dem zweiten Zeichen verglichen usw.
  Aufwand bei erfolgreicher Suche: L_i/k (+ 1 bei Präfix)

  ➥ effiziente Bestimmung der Abwesenheit eines Schlüssels (z. B. CAD)

  (2) Einfügen:

  Wenn Suchpfad schon vorhanden, wird ein NIL-Zeiger in einen *-Zeiger umgewandelt, sonst Einfügen von neuen Knoten (z. B. CAD)

  (3) Löschen:

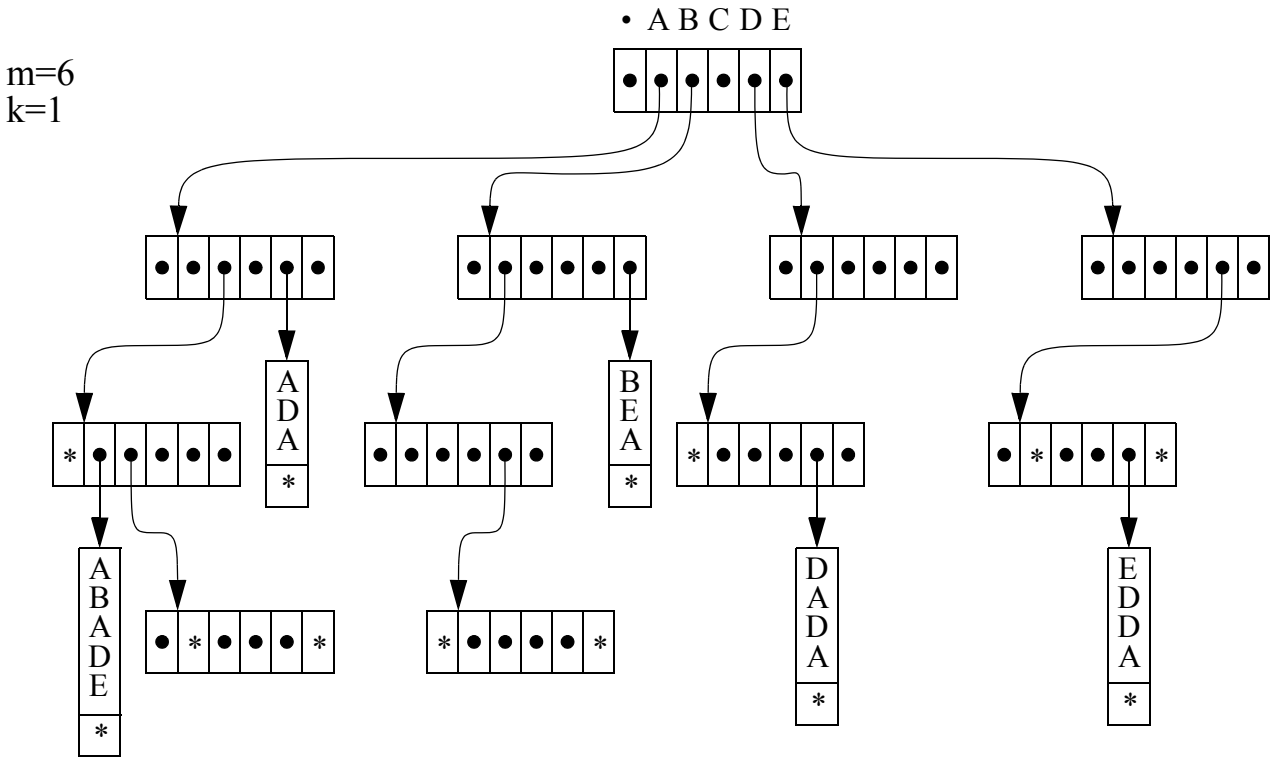
  Nach Aufsuchen des richtigen Knotens wird ein *-Zeiger auf NIL gesetzt. Besitzt daraufhin der Knoten nur NIL-Zeiger, wird er aus dem Baum entfernt (rekursive Überprüfung der Vorgängerknoten).

  (4) Sequentielle Suche ?

m-ärer Trie (4)

- **Keine dynamische Reorganisation**
  - Es findet also weder ein Split- noch ein Mischvorgang statt.
  - Die Zuordnung der Knoten ohne Ausgleichsvorgänge ist auch ein Grund für die schlechte Platzausnutzung

- **Verbesserung der Platzausnutzung**
  - Vermeidung von Einweg-Verzweigungen im Trie
  - **Variante:** Sobald ein Zeiger auf einen Unterbaum mit nur einem Schlüssel verweist, wird anstelle des Unterbaums der darin enthaltene Rest-Schlüssel (oder der gesamte Schlüssel) in einem speziellen Knotenformat dargestellt.
  - **Modifizierte Trie-Darstellung**

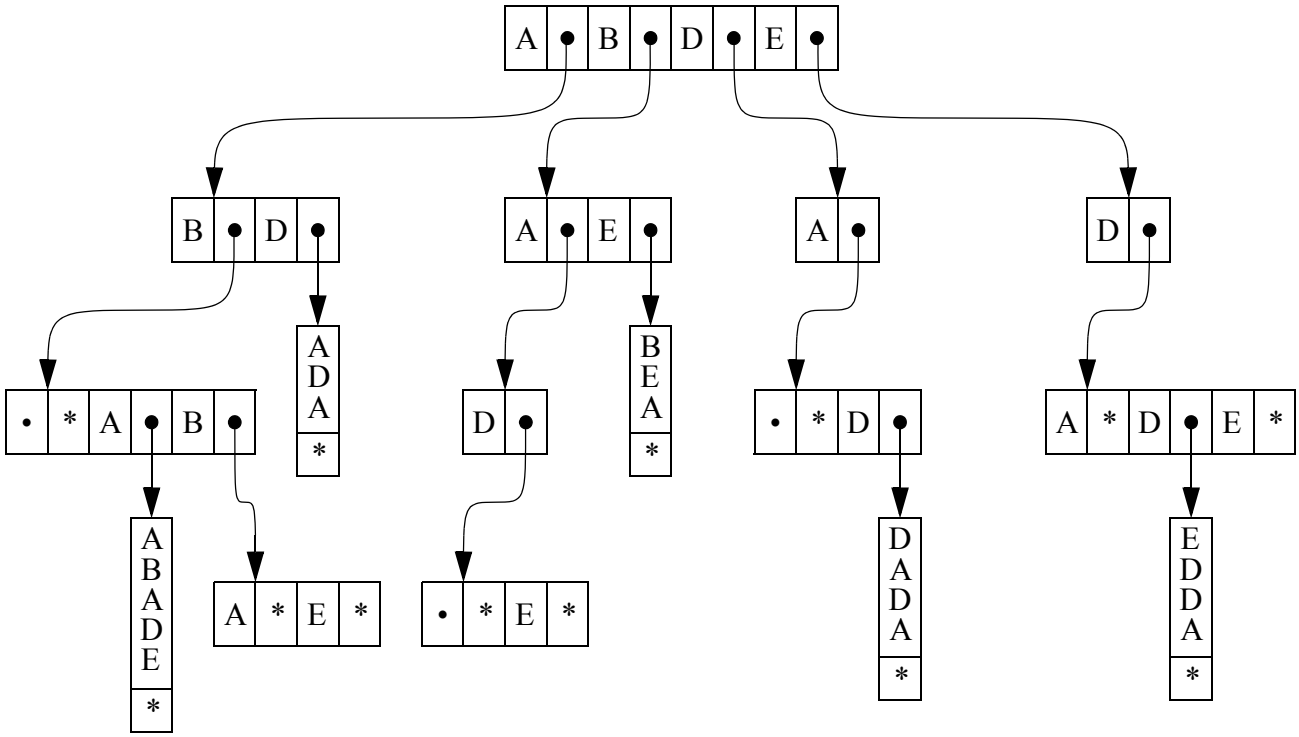


➥ Für große n kann man nach Knuth mit einem durchschnittlichen Suchaufwand von $\log_m n$ Suchschritten rechnen, wenn die Schlüssel zufällig erzeugt sind.

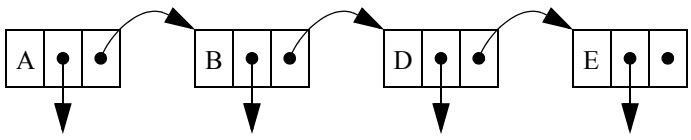
m-ärer Trie (5)

- **Digitalbaum mit variablem Knotenformat**
  - Selbst wenn im Trie Einweg-Verzweigungen vermieden werden, sind viele Knoten nur dünn besetzt.
  - Deutliche Verbesserung der Speicherausnutzung, wenn durch Übergang auf eine variable Knotengröße nur noch die besetzten Zeiger gespeichert werden.
  - Da implizite Zuordnung von Schlüsselzeichen zu Zeigerposition aufgegeben wird, ist zu jedem Zeiger zugehöriges Schlüsselzeichen explizit zu speichern.

    ➥ Die Trie-Charakteristik geht verloren!



  - Variables Knotenformat verursacht bei der Speicherverwaltung oft Probleme.
  - Vorschlag: doppelte Verkettung



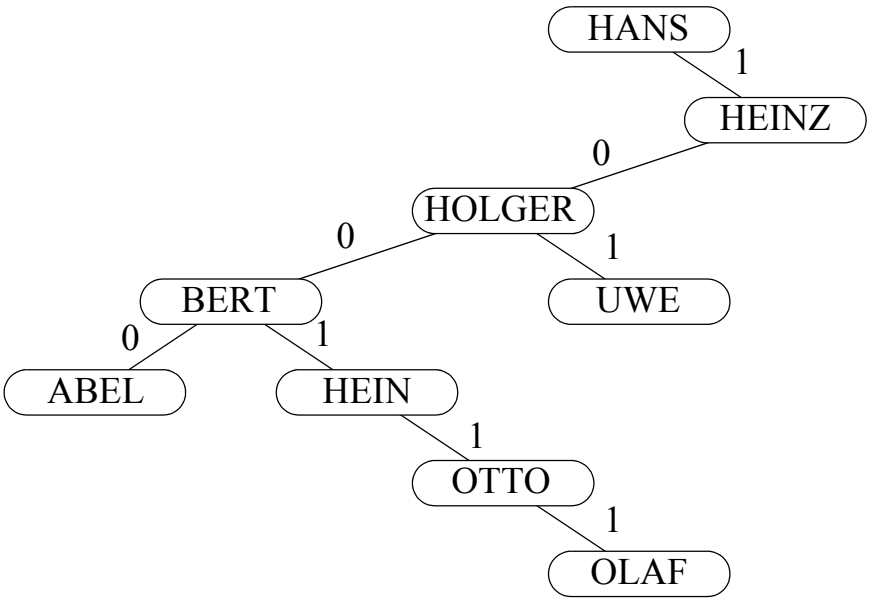
    ➥ Binärbaum-Darstellung

Binärer Digitalbaum

- **Spezialfall: binäres Alphabet**

- **1. Variante: binärer digitaler Suchbaum**
  - In jedem Knoten ist jeweils ein vollständiger Schlüssel - genauso wie im binären Suchbaum - gespeichert
  - Beim Einfügen erhält der Schlüssel den ersten freien Blattknoten, der über seine Bitfolge gefunden wird
  - Zur Entscheidung, ob in einem Knoten links oder rechts verzweigt werden soll, wenn der gespeicherte Schlüssel nicht mit dem Suchschlüssel übereinstimmt, dienen der Reihe nach die einzelnen Bits des Suchschlüssels

HANS = 1001000...
HEINZ = 1001000...
HOLGER= 1001000...
BERT = 1000010...
...
OTTO = 1001111...
...



- **Bewertung**
  - keine Darstellung einer geordneten Menge (LWR ?)
  - abhängig von Schlüsselmenge und Einfügereihenfolge
  - lange Einwegverzweigungen, keine dynamische Balancierung

    ➥ **besser ausgewogene Bäume:** anstelle der Bitfolge von K_i Zufallszahl mit K_i als Saat

- **Anwendung:** statische Schlüsselmengen mit stark gewichteten Zugriffshäufigkeiten

Binärer Digitalbaum (2)

- **2. Variante: PATRICIA-Baum**

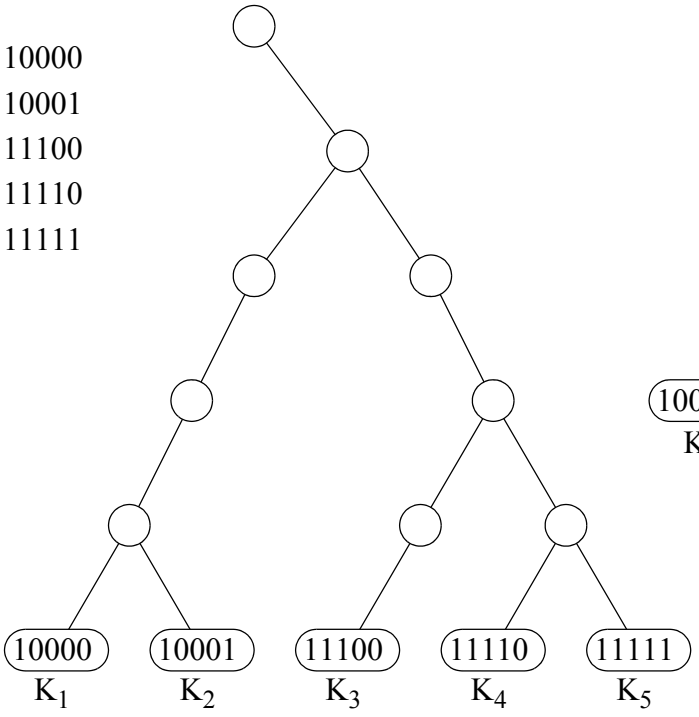
  (Practical Algorithm To Retrieve Information Coded In Alphanumeric)

  - **Grundidee:** Vermeidung von Einwegverzweigungen
  - Speicherung der Schlüssel in den Blättern
  - **innere Knoten:** es wird gespeichert, wieviele Bits beim Test zur Wegeauswahl zu überspringen sind
  - Konstruktionsprinzip

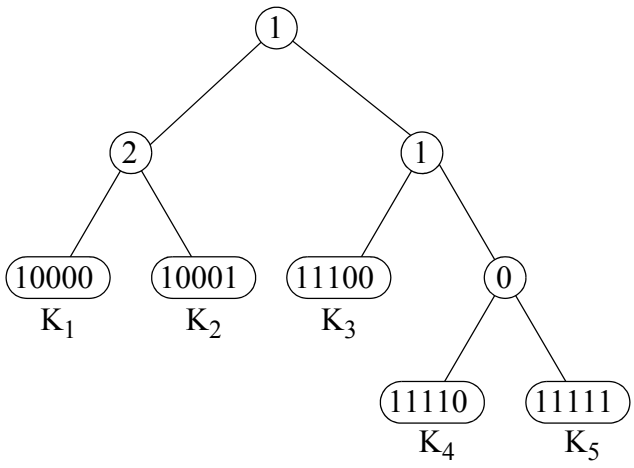
Schlüsselmenge

K_1= 10000
K_2= 10001
K_3= 11100
K_4= 11110
K_5= 11111

binärer Digitalbaum mit Einwegverzweigungen



PATRICIA -Baum

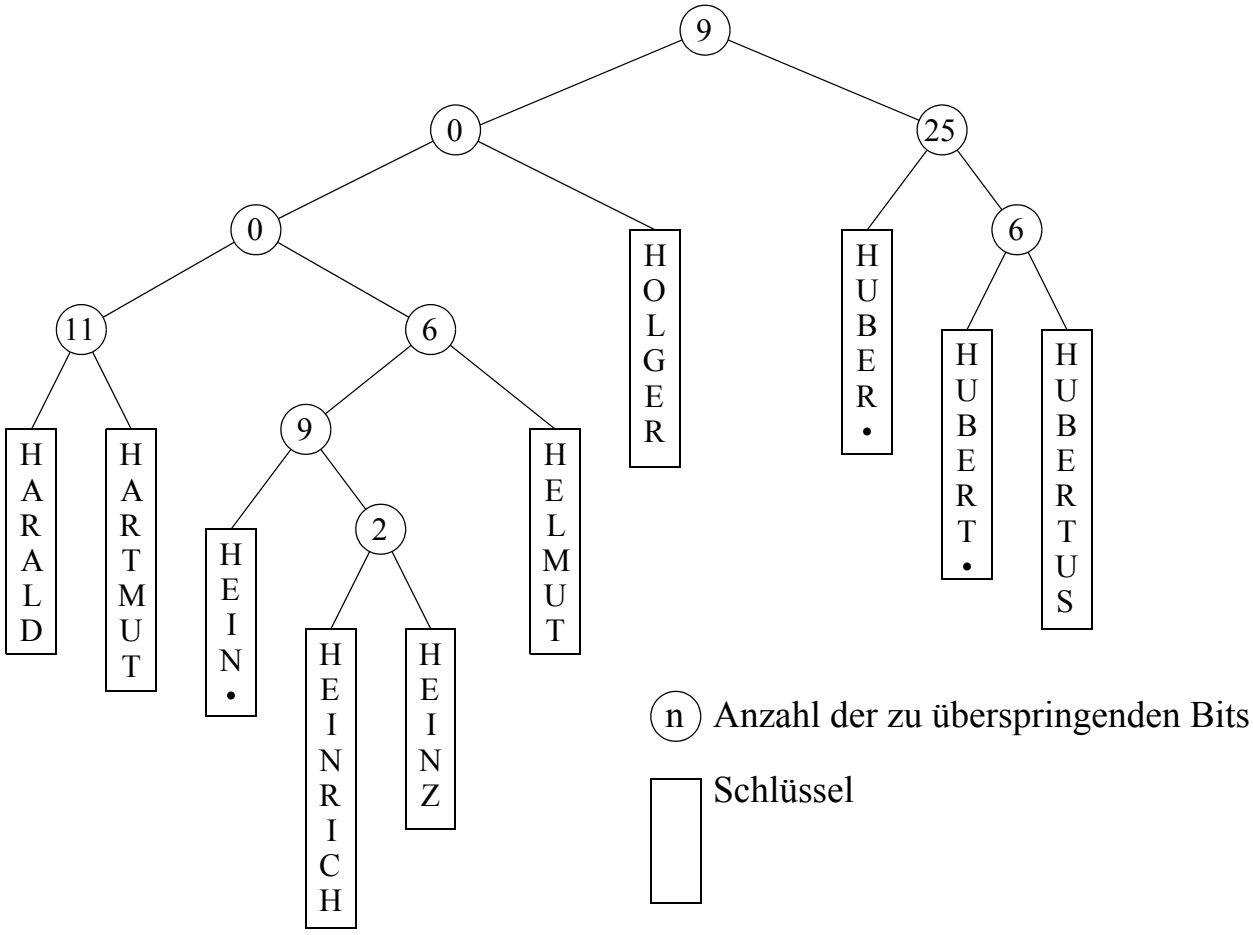


- **Bewertung**

  - Es treten keine Einwegverzweigungen auf
  - sonst jedoch ähnlich wie beim binären digitalen Suchbaum
  - Baumstruktur läßt sich als Testanleitung für Suchschlüssel auffassen.
    Bei jedem beliebigen Schlüssel muß die Testfolge ganz ausgeführt werden, bevor über Erfolg oder Mißerfolg der Suche entschieden werden kann.

Binärer Digitalbaum (3)

- **PATRICIA-Baum für ein Anwendungsbeispiel**



- einfache Struktur der inneren Knoten

- Wie verläuft die Suche nach dem Schlüssel
  HEINZ = X'10010001000101100100110011101011010' ?

- Wie wird getestet, wenn
  ABEL = X'10000011000010100010110011100'
  gesucht wird ?

  ➥ erfolgreiche und erfolglose Suche endet in einem Blattknoten

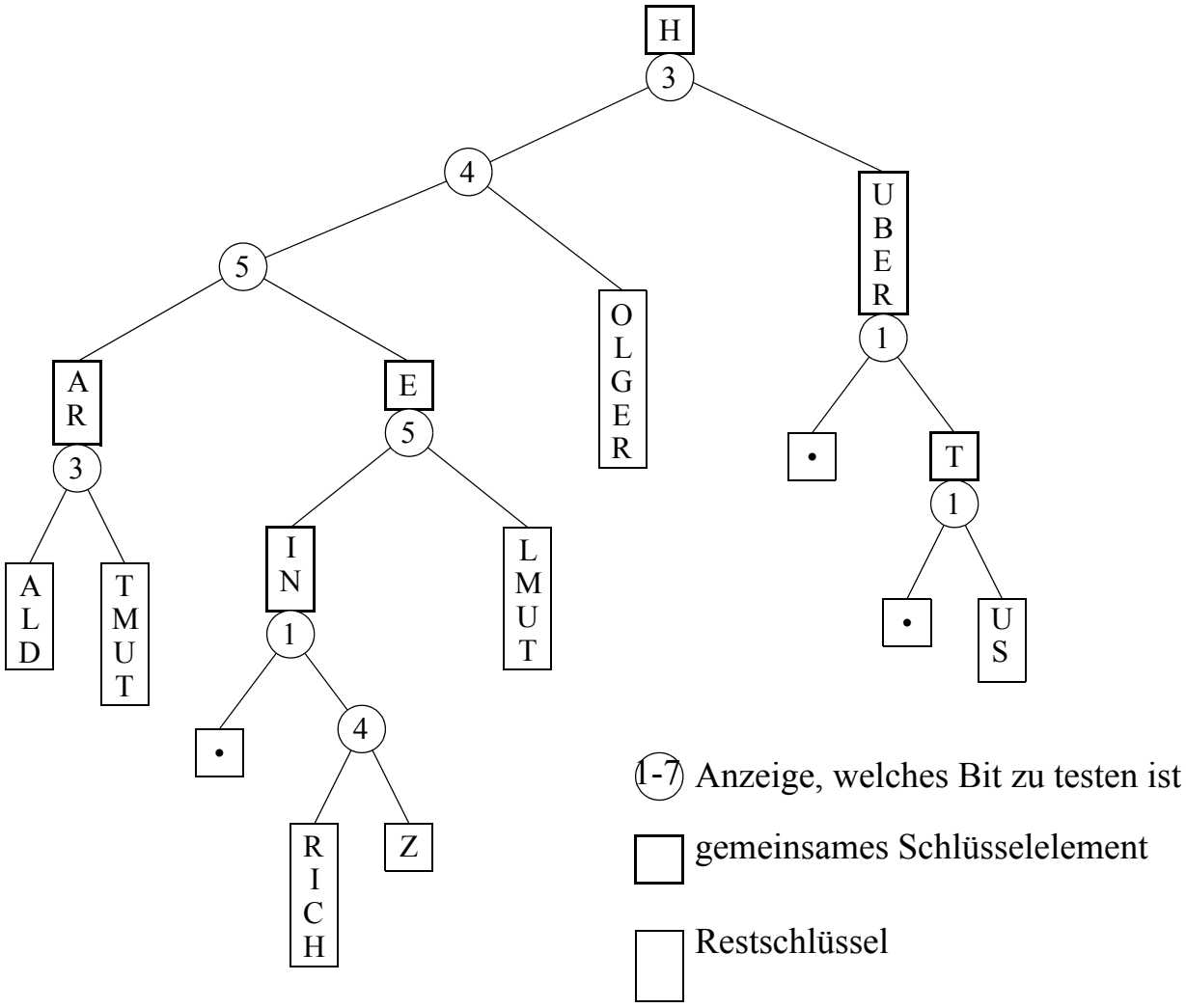
Binärer Digitalbaum (4)

- **3. Variante: Binärer Radix-Baum**

  als Modifikation des PATRICIA-Baumes

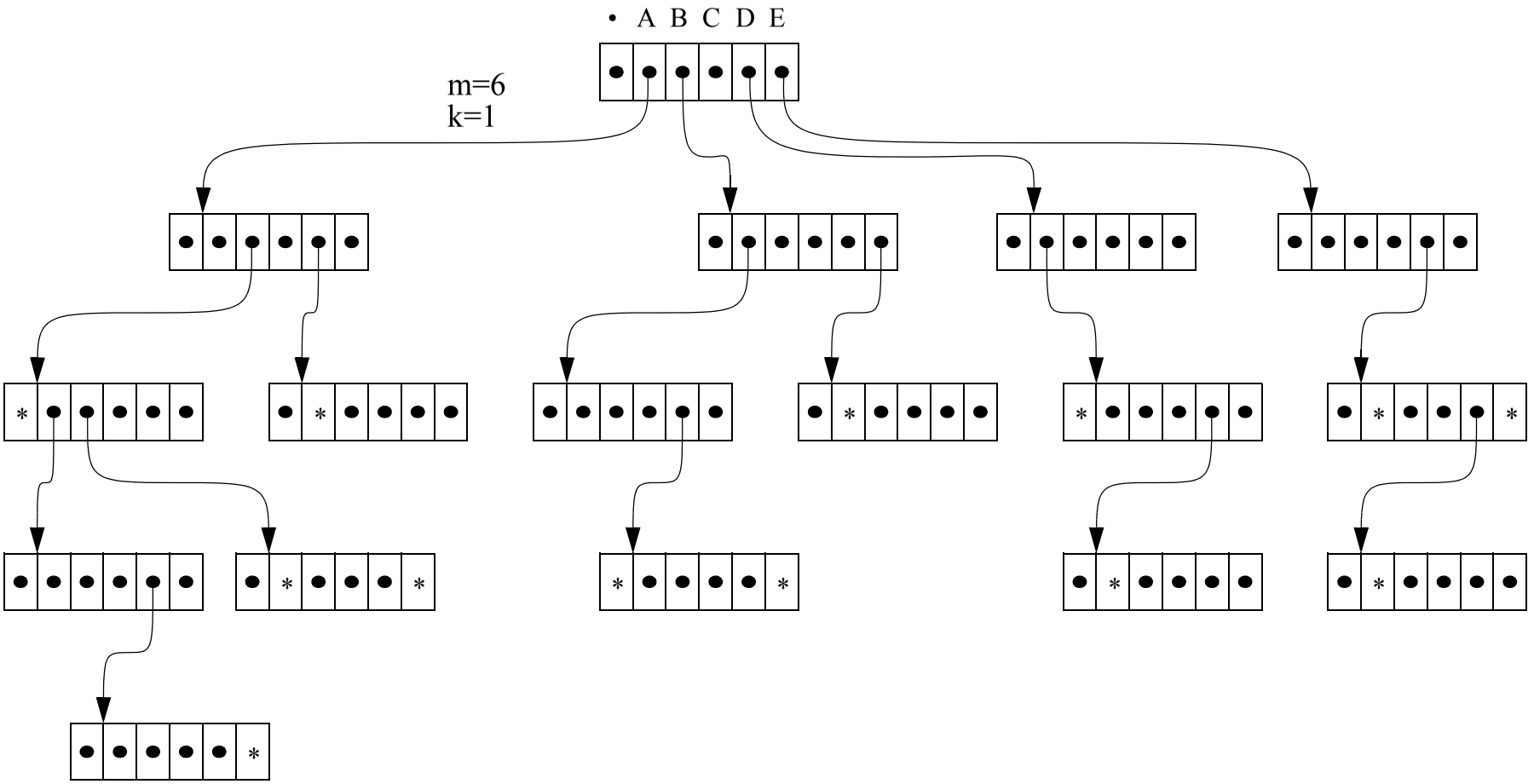
  - Speicherung der Testinformation
  - zusätzlich Speicherung variabel langer Schlüsselteile in den inneren Knoten, sobald sie sich als Präfixe für die Schlüssel des zugehörigen Unterbaums abspalten lassen

- **Gewähltes Anwendungsbeispiel**



  - komplexere Knotenformate und aufwendigere Such- und Aktualisierungs-operationen
  - erfolglose Suche läßt sich oft schon in einem inneren Knoten abbrechen

Trie-Darstellung für Schlüssel aus einem auf A-E beschränkten Alphabet



Welche Schlüssel sind im Trie dargestellt ?

Addressing in Trees

- **XML documents are to be stored in databases**
  - conceptual representation: trees with nodes and edges
  - document order must be preserved / recoverable:
    node order matters!
  - **LOBs** don't enable fine-granular management,
    no content-based search and no multi-user operation
  - **mapping onto relational tables?**
    - many solutions: „shredding"
    - XML query language (e.g., XQuery, XPath, DOM, SAX)
      must be mapped to SQL
    - use of the SQL optimizer!
    - but: concurrency control (locking) very cumbersome,
      because a document is distributed over n tables
  - improved solution needed: **native XML DBMS!**
    - fine-granular management and storage of the XML documents
      as native tree-like storage structures
    - navigating and direct access to all document nodes
    - indexing of all nodes should accelerate declarative queries
    - modification of documents also required under multi-user operations
      (cooperative processing)
    - fine-granular locking: nodes, edges and subtrees

- **How to address tree nodes,**
  which can be arbitrarily displaced by later insertions?
  - how do XML documents appear at the user level?
  - which storage structures are adequate?
  - which labeling scheme
    should be used for the nodes?

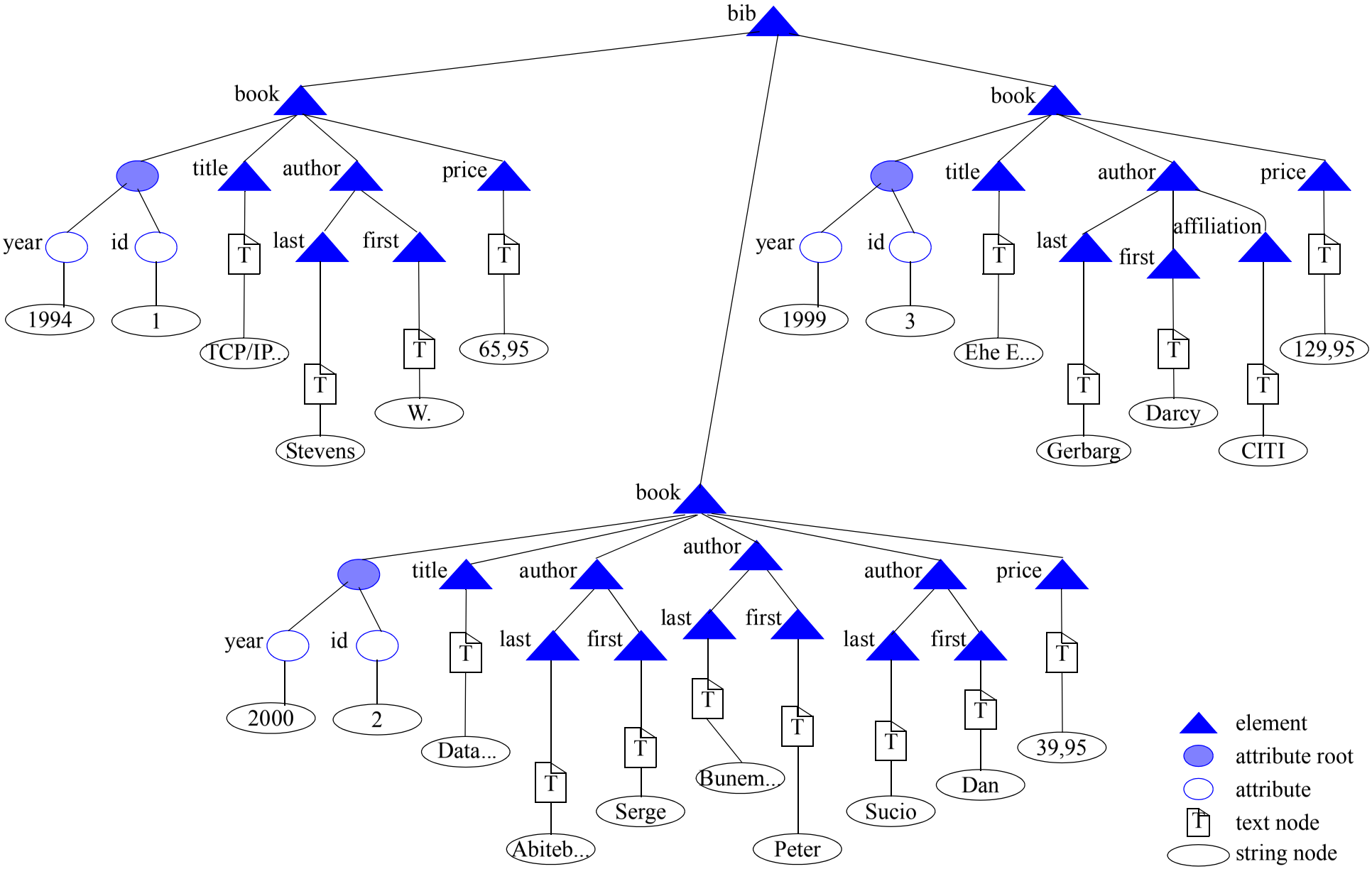
Addressing in Trees (2)

- **Example of an XML document**

```
<bib>
   <book year="1994" id="1">
        <title>TCP/IP Illustrated</title>
        <author>
             <last>Stevens</last>
             <first>W.</first>
        </author>
        <price>65.95</price>
   </book>
   <book year="2000" id="2">
        <title>Data on the Web</title>
        <author>
             <last>Abiteboul</last>
             <first>Serge</first>
        </author>
        <author>
             <last>Buneman</last>
             <first>Peter</first>
        </author>
        <author>
             <last>Suciu</last>
             <first>Dan</first>
        </author>
        <price>39.95</price>
   </book>
   <book year="1999" id="3">
        <title>The Economics of . . . </title>
        <editor>
             <last>Gerbarg</last>
             <first>Darcy</first>
             <affiliation>CITI</affiliation>
        </editor>
        <price>129.95</price>
   </book>
</bib>
```

- **Modified storage structure called taDOM tree**
  - DOM structure was extended to optimize lock protocols
  - taDOM storage structure has additional nodes:
    attribute root and string node
  - these nodes are not visible at the XML-DBMS interface

Example of a taDOM Tree

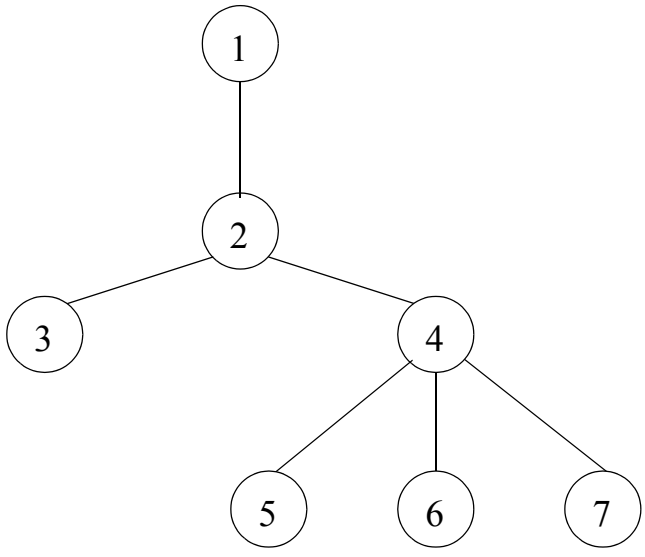


Addressing in Trees (3)

- **Principal approaches to a solution**
  - representation of an XML document: ordered, labeled tree with nodes of type element, attribute, text (and attribute root, string)
  - labeling scheme should be insensitive to insertions
  - 13 different axes defined in XPath (sequence semantics):
    child, parent, descendant, ancestor, following-sibling, preceding-sibling, following, preceding, descendant-or-self, ancestor-or-self, self, attribute, namespace
  - support of the most important axes required:
    parent/child, ancestor/descendant, preceding-sibling/following-sibling
  - two classes: range-based and prefix-based schemes

- **Range-based solutions**
  - positions of nodes in trees are marked by (DocNo, LeftPos:RightPos, LevelNo)
  - LeftPos (LP) and RightPos (RP) describe the labeling range in each node with its sub-tree and can be generated by a depth-first traversal of the tree
  - ancestor-descendant containment (DocNo is omitted): a node n_1 (LP_1:RP_1, lv_1) contains a node n_2 (LP_2:RP_2, lv_2), iff $LP_1 < LP_2$ and $RP_1 > RP_2$.
  - additional condition for parent-child containment: $lv_1 = lv_2 - 1$
  - supporting preceding-sibling/following-sibling relationship?

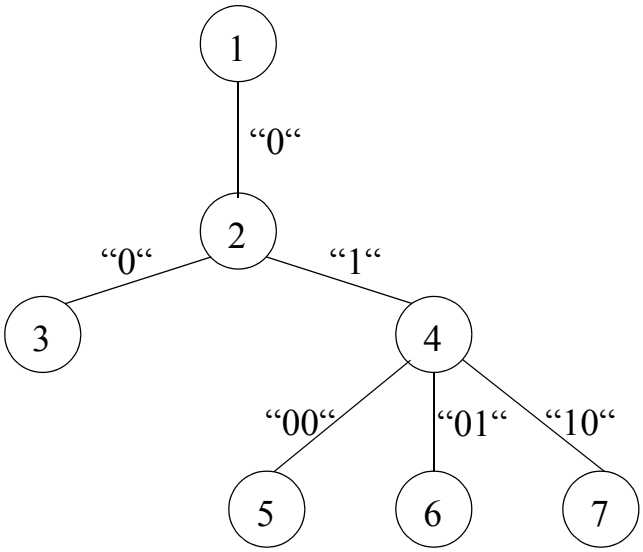
- **Example of a simple range scheme**



label template (LP:RP, lv)

Addressing in Trees (4)

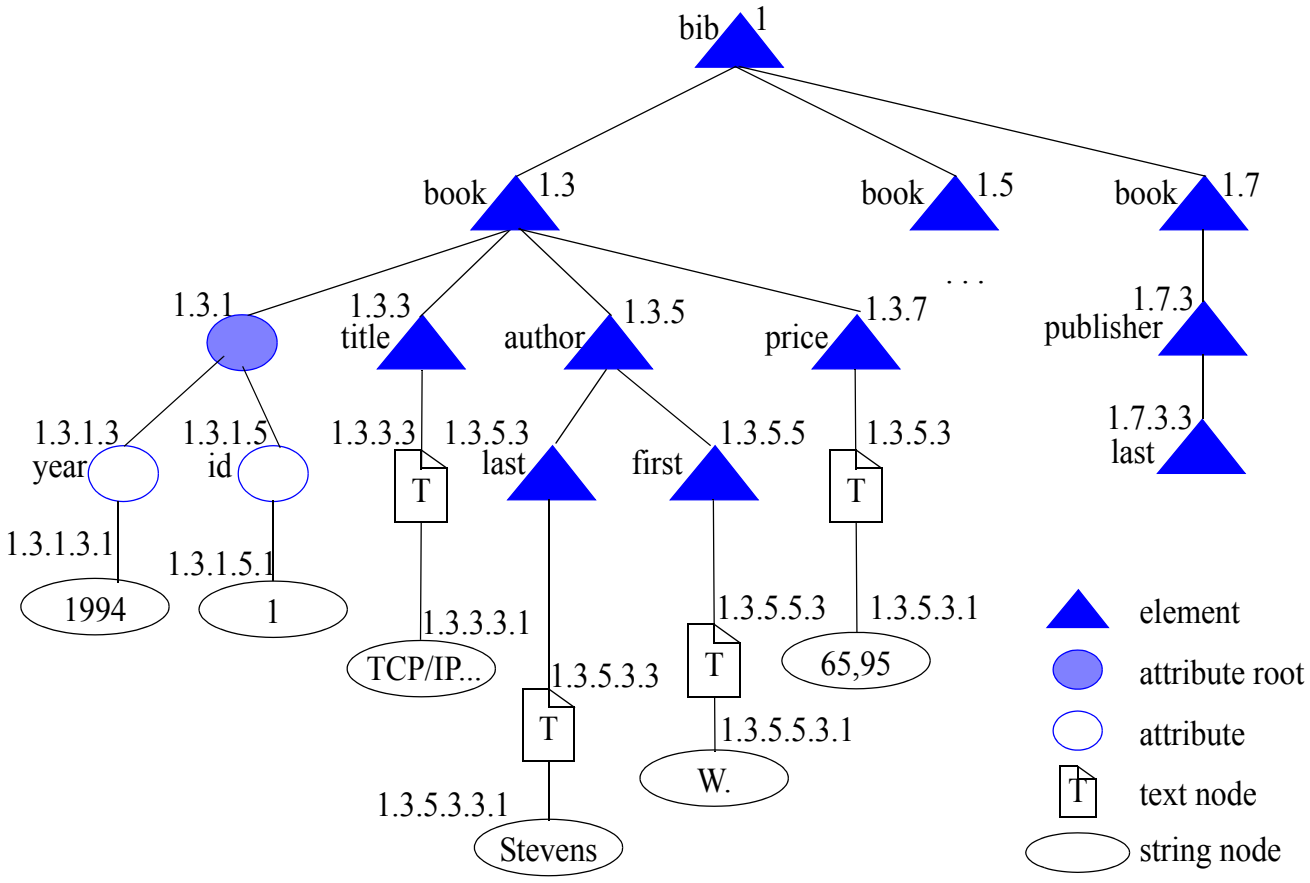
- **Prefix-based solutions**
  - in a prefix-based labeling scheme, each node is encoded with a unique string S such that
  1. S(v) is before S(u) in lexicographic order iff node v is before node u in the document order
  2. S(v) is a prefix of S(u) iff node v is the ancestor of node u
  - simple example prefix scheme:
    - assign to the outgoing edges of each node a set of prefix-free binary strings in lexicographical order from left to right
    - starting from the root and going down, the label of each node is the concatenation of the parent's label and the string assigned to its incoming edge
    - the level of a node is recorded to distinguish the parent-child and the ancestor-descendant relationship
    - add the edge string length esl to each node descriptor to derive the parent (ancestor) label
  - support of preceding-sibling/following-sibling relationship needs indexes
- **Example**



label template (S, lv, esl)

Addressing in Trees (5)

- **Concept of DeweyIDs**[1]
  - Dewey order was introduced in libraries to enable the insertion of new books at arbitrary places thereby preserving the given order
  - suitable adaptation to XML trees; satisfies both conditions for prefix schemes
- **Assignment of DeweyIDs**
  - distinction: initial loading of the document (bulk-loading) and later insertion of nodes
  - DeweyID consists of several divisions separated by dots
  - assignment of division values is affected by parameter *distance*
  - on initial loading, only odd division values are assigned
  - *distance* (in the example *distance=2*) enables later insertions; even division values are used for a kind of overflow mechanism



1. Decimal classification system of Dewey: http://www.mtsu.edu/~vvesper/dewey.html

Addressing in Trees (6)

- **Initial document loading**[2]

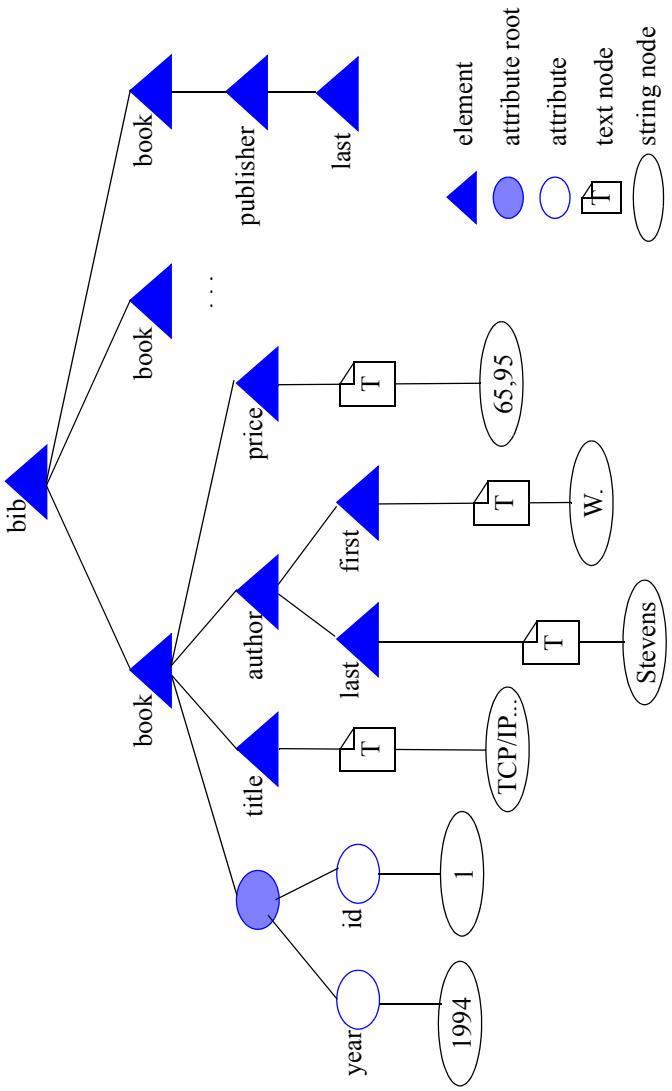
While a new document is loaded—typically bulk-loaded in left-most depth-first order—, the DeweyIDs for its nodes are dynamically assigned which is guided by the following rules:

1. Element root node: It always obtains DeweyID 1.
2. Element nodes: The first node at a level receives the DeweyID of its parent node extended by a division of *distance + 1*. If a node N is inserted after the last node L at a level, DeweyID of L is assigned to N where the value of the last division is increased by *distance*.
3. Attribute nodes: A node N having at least one attribute, obtains (in taDOM) an attribute root R for which the DeweyID of N extended by a division with value 1 is assigned. The attribute node yields the DeweyID of R extended by a division. If it is the first attribute node of R, this division has the value 3. Otherwise, the division receives the value of the last division of the last attribute node increased by 2. In this case, the distance value does not matter, because the attribute sequence does not affect the semantics of the document. Therefore, new attributes can always be inserted at the end of the attribute list.
4. Text nodes: A node containing text is represented in taDOM by a text node and a string node. For text nodes, the same rules apply as for element nodes. The value of an attribute or a text node is stored in a string node. This string node obtains the DeweyID of the text node resp. attribute node, extended by a division with value 1.

2. M. Haustein, T. Härder, C. Mathis, M. Wagner: DeweyIDs – The Key to Fine-Grained Management of XML Documents, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html

Addressing in Trees (7)

- **Initial assignment of DeweyIDs**
  - Example using distance = 8



	node type	rule	DeweyID
bib	element	1	1
book	element	2	1.9
	attr. root	3	1.9.1
year	attribute	3	1.9.1.3
1994	string	4	1.9.1.3.1
id	attribute	3	1.9.1.5
1	string	4	1.9.1.5.1
title	element	2	1.9.9
	text	4	1.9.9.9
TCP/IP...	string	4	1.9.9.9.1
author	element	2	1.9.17
last	element	2	1.9.17.9
	text	4	1.9.17.9.9
Stevens	string	4	1.9.17.9.9.1
first	element	2	1.9.17.17
	text	4	1.9.17.17.9
W.	string	4	1.9.17.17.9.1
price	element	2	1.9.25
	text	4	1.9.25.9
65.95	string	4	1.9.25.9.1
book	element	2	1.17
book	element	2	1.25
publisher	element	2	1.25.9
last	last	2	1.25.9.9

Addressing in Trees (8)

- **DeweyID assignment when new nodes are Inserted**

When new nodes are inserted at arbitrary logical positions, their DeweyIDs must reflect the intended Document order as well as position, level, and type of node without enforcing modifications of DeweyIDs already present. For element nodes and text nodes, the same rules apply. In contrast to them, attribute roots, attribute nodes, and string nodes do not need special consideration by applying rule 3, because order and level properties do not matter.

Assignment of a DeweyID for a new last sibling is similar to the initial loading. Here, the last level only consists of one division. Hence, when inserting element node *year* after price, addition of the distance value yields 1.9.33. In case, the last level consists of more than one division (indicated by even values), the first division of this level is increased by *distance - 1*. For example, the successor of 1.3.14.6.5 is 1.3.21.

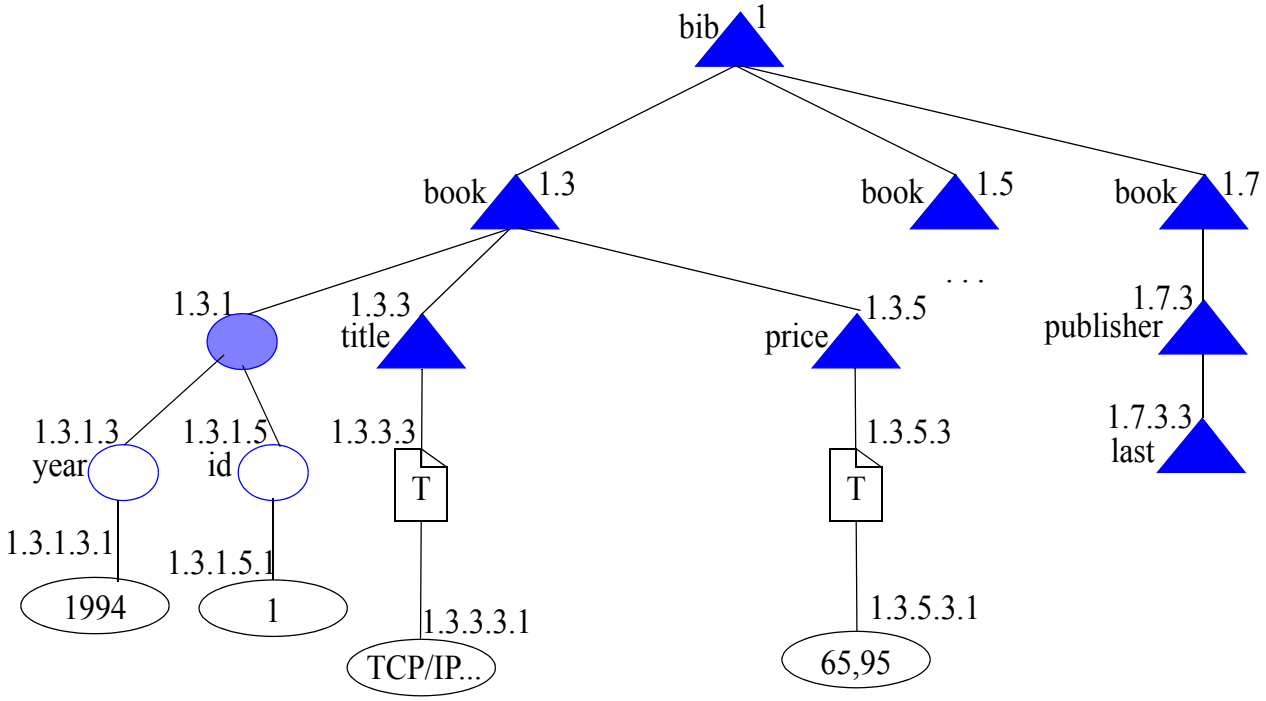
If a sibling is inserted before the first existing sibling, the first division of the last level is halved and, if necessary, ceiled to the next integer or increased by 1 to get an odd division. This measure secures that the "before-and-after gaps" for new nodes remain equal. Hence, inserting a *type* node before *title* would result in DeweyID 1.9.5. If the first divisions of the last level are already 2, they have to be adopted unchanged, because smaller division values than 2 are not possible, e.g., the predecessor of 1.9.2.2.8.9 is 1.9.2.2.5. In case the first division of the last level is 3, it will be replaced by *2.distance+1*. For example, the predecessor of 1.9.3 receives 1.9.2.9.

The remaining case is the insertion of node d_2 between two existing nodes d_1 and d_3. Hence, for d_2 we must find a new DeweyID which is between the DeweyIDs of d_1 and d_3. Because they are allocated at the same level and have the same parent node, they only differ at the last level (which may consist of arbitrary many even divisions and one odd division, in case a weird insertion history took place at that position in the tree). All common divisions before the first differing division are also equal for the new DeweyID. The first differing division determines the division becoming part of DeweyID for d_2. If possible, we prefer a median division to keep the before-and-after gaps equal. Assume for example, d_1 = 1.9.5.7.5 and d_3 = 1.9.5.7.16.5, for which the first differing divisions are 5 and 16. Hence, choosing the median odd division result in d_2 = 1.9.5.7.11.

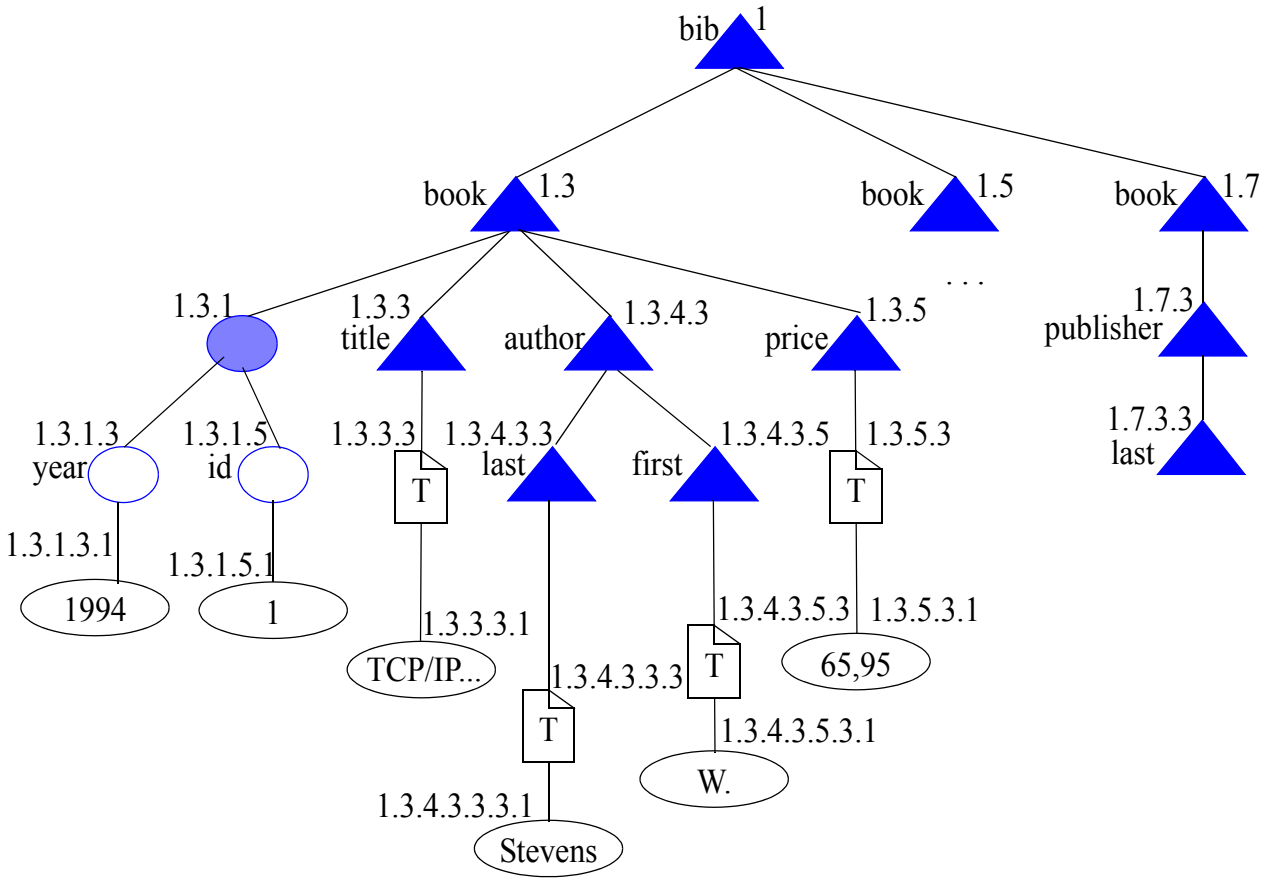
If d_4 = 1.5.6.7.5 and d_6 = 1.5.6.7.7, only even division 6 would fit. Remember, we have to recognize the correct level. Hence, having distance value 8, d_5 = 1.5.6.7.6.9.

Addressing in Trees (9)

- **Initial assignment of DeweyIDs (*distance=2*)**



- **Insertion of a subtree**



Addressing in Trees (10)

- **Worst-case Considerations**

Assume a test case with root 1 and a child with DeweyID *1.distance+1*. The insertion of the sibling always takes place before the last inserted one. Hence, using distance value 16 and "halving the gap" as an example, the sequence of assigned DeweyIDs is

- **Benefits of DeweyID Use**

  - Existing DeweyIDs allow the assignment of new IDs without the need to reorganize the IDs of nodes present. A relabeling after weird insertion histories (think of point insertions[3] of thousands of nodes between two existing nodes) is only required, if the length of a DeweyID exceeds a given limit and violates existing implementation restrictions, e.g., the maximum lengths of keys in B*-trees.
  - The DeweyID of the parent node can be determined in a very simple way, e.g., frequently needed for locking the ancestor path.
  - Comparison of two DeweyIDs delivers the order of the respective nodes in the left-most depth-first stored document.
  - Checking whether node d_1 is an ancestor of d_2 only requires to check whether DeweyID of d_1 is a prefix of DeweyID of d_2.
  - High distance values reduce the probability of reorganization. They have to be balanced against increased storage space for the representation of DeweyIDs.

3. New insertions before the currently inserted node may enforce increased use of even division values thereby extending the total length of a DeweyID)

Zusammenfassung

- **Cluster-Bildung optimiert (sortiert) sequentielle Zugriffe**
- **Zugriffsverhalten des AVL-Baumes mit $O(\log_2 n)$ ist nicht gut genug**
- **Standard-Zugriffspfadstruktur: B*-Baum** (the ubiquitous B*-tree)
  - fehlt in keinem DBMS
  - materialisierte und referenzierte Speicherung der Datensätze
  - Index-organisierte Tabelle mit Cluster-Bildung
- **Indexstrukturen als B*-Bäume**
  - mit und ohne Clusterbildung spezifizierbar
  - Balancierte Struktur unabhängig von Schlüsselmenge und Einfügereihenfolge

  ➥ dynamische Reorganisation durch Aufteilen (Split) und Mischen von Seiten
  - direkter Schlüsselzugriff auf einen indexierten Satz
  - sortiert sequentieller Zugriff auf alle Sätze (unterstützt Bereichsanfragen, Verbundoperation usw.)

  ➥ Wie viele Indexstrukturen/Tabellen?
- **Digitalbäume**
  - Kein „eingebautes“ Balancierungskriterium
  - als Pfadindexe für XML-Dokumente vorgeschlagen
  - Abbildung auf Externspeicher ist schwierig bei dynamischen Dokumenten
- **DeweyIDs als bevorzugtes Nummerierungsschema für Bäume**
  - ordnungserhaltend und stabil bei Einfügungen, aber variabel lange Einträge
  - ausdrucksstark mit effektiver Unterstützung für DB-Operationen

Zugriffspfade in kommerziellen Datenbanksystemen

DB2 (IBM)	B*-Baum (clustered, non-clustered), partitionierte Tabellen, ...
Informix	B-Baum, statisches Hashing, ISAM, HEAP, ...
Oracle	B*-Baum (mit Präfix-/Suffix-Komprimierung), (Join-) Cluster-Bildung, ...
Sybase	B*-Baum (clustered, non-clustered), ...
RDB (DEC)	B*-Baum (clustered, non-clustered), Hashing, Join-Cluster-Bildung, ...
NonStop SQL (Tandem)	B*-Baum (clustered, non-clustered) mit Präfix-Komprimierung, ...
UDS (Siemens)	B*-Baum, statisches Hashing, Cluster-Bildung (LIST), Invertierung (Pointer-Array), Kettung (CHAIN)