ADS 1: Algorithmen und Datenstrukturen

Teil X

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

19. Dezember 2017

[Letzte Aktualisierung: 18/12/2017, 10:05]

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: Faktor 100 000 langsamer als für Hauptspeicher (5 ms verglichen mit 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-sequenzielle Speicherung)

Sequenzieller Zugriff

- Lesen aller Seiten / Sätze vom Beginn der Datei an
- sehr hohe Zugriffskosten, wenn nur ein Satz benötigt wird

Optimierungsmöglichkeiten

- "dichtes Packen" der Sätze innerhalb der Seiten
- Clusterung zwischen Seiten, d.h. "dichtes Packen" der Seiten einer Datei auf physisch benachbarte Plattenbereiche, um geringe Zugriffszeiten zu ermöglichen

Schneller Zugriff auf einzelne Datensätze erfordert Einsatz von zusätzlichen Indexstrukturen, z.B. Mehrwegbäume

Alternative: gestreute Speicherung der Sätze ($\rightarrow$ Hashing)

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

Alternativen:

- m-Wege-Suchbäume
- B-Bäume
- B*-Bäume (heute eher als B+-Bäume bezeichnet)

Grundoperationen: Suchen, Einfügen, Löschen

m-Wege-Suchbäume I

Definition : Ein *m-Wege-Suchbaum oder ein m-ärer Suchbaum* B ist ein Baum, in dem jeder Knoten höchstens m Kinder besitzt. Entweder ist B leer oder B hat folgende Eigenschaften:

1. Jeder Knoten des Baums mit b Einträgen, $b \leq m-1$, hat folgende Struktur: $\boxed{P_0|K_1|D_1|P_1|K_2|D_2|P_2|\ldots|K_b|D_b|P_b}$
   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 in jedem Knoten sind aufsteigend geordnet: $K_i \leq K_{i+1}$, $1 \leq i < b$.
3. Alle Schlüssel im Unterbaum von P_i sind kleiner als K_{i+1} $(0 \leq i < b)$
4. Alle Schlüssel im Unterbaum von P_i sind größer als K_i $(0 < i \leq b)$
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.

B-Bäume

Definition: Sei $t \geq 1$ eine ganze Zahl. Ein *B-Baum B der Klasse t* ist ein $2t$-Wege-Suchbaum mit folgenden zusätzlichen Eigenschaften:

1. Alle Blätter sind von der Wurzel gleich weit entfernt.
2. Jeder Knoten außer der Wurzel hat mindestens $t - 1$ und höchstens $2t - 1$ Einträge.
3. Jeder Knoten, außer der Wurzel und den Blättern, hat mindestens t und höchstens $2t$ Töchter.
4. Falls B nicht leer ist, hat die Wurzel mindestens 1 und höchstens $2t - 1$ Einträge.
5. Falls B nicht leer und die Wurzel kein Blatt ist, hat sie mindestens 2 und höchstens $2t$ Töchter.

Aufbau der Knoten im B-Baum

Innerer Knoten

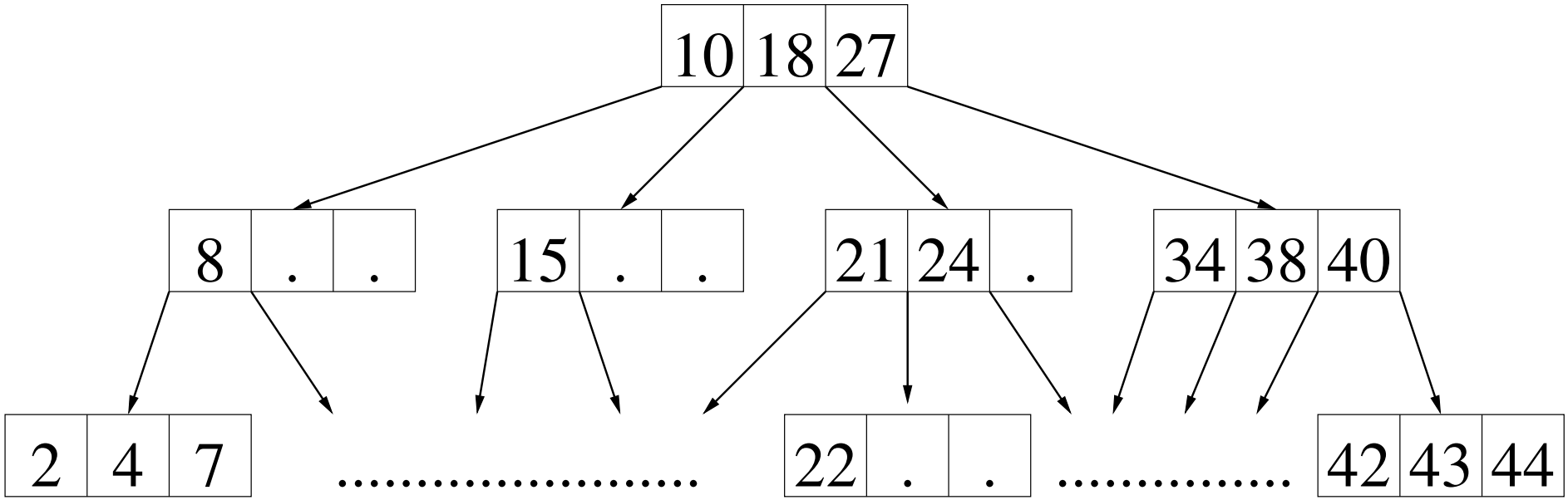
P_0	K_1	D_1	P_1	K_2	D_2	P_2	$\ldots$	K_b	D_b	P_b	freier Platz

Blattknoten

K_1	D_1	K_2	D_2	$\ldots$	K_b	D_b	freier Platz

- $P_0 \ldots P_b$ Zeiger auf Unterbäume
- $K_1 \ldots K_b$ Schlüssel
- $D_1 \ldots D_b$ (Zeiger auf) Daten
- $t - 1 \leq b \leq 2t - 1$

Beispiel: B-Baum mit $t = 2$



Suchen im B-Baum

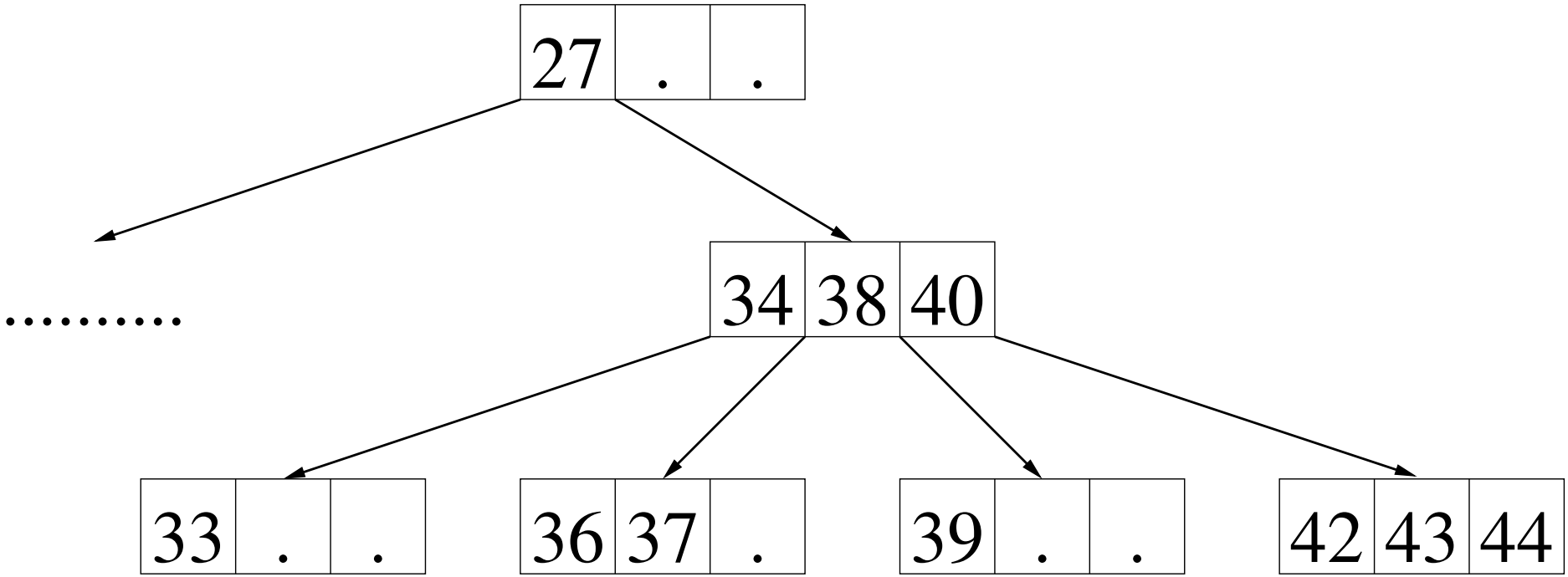
Verallgemeinerung der Suche im binären Suchbaum

1. Suche nach Schlüssel k beginnt mit der Wurzel als aktuellem Knoten x.
2. Finde größten Index $i \in \{1, \ldots, b\}$, so dass $k \geq K_i$ ist. Falls $k = K_i$, so wurde k gefunden (Ende, Suche erfolgreich).
3. Existiert so ein Index nicht, so setze $i = 0$.
4. Falls x bereits ein Blatt ist, Ende (Suche erfolglos).
5. Sonst: verzweige $x \leftarrow P_i$ und gehe zurück zu (2).

Einfügen im B-Baum

- Einfügen eines Schlüssels k in einen B-Baum geschieht immer in einem Blattknoten x.
- Wie bei der Suche wird ein Pfad von der Wurzel bis zu x durchlaufen.
- Trifft der Durchlauf auf einen vollen Knoten y, so wird y geteilt, bevor der Durchlauf fortgesetzt wird.
- Ein Knoten ist voll, wenn er die maximale Anzahl $2t - 1$ an Schlüsseln enthält.

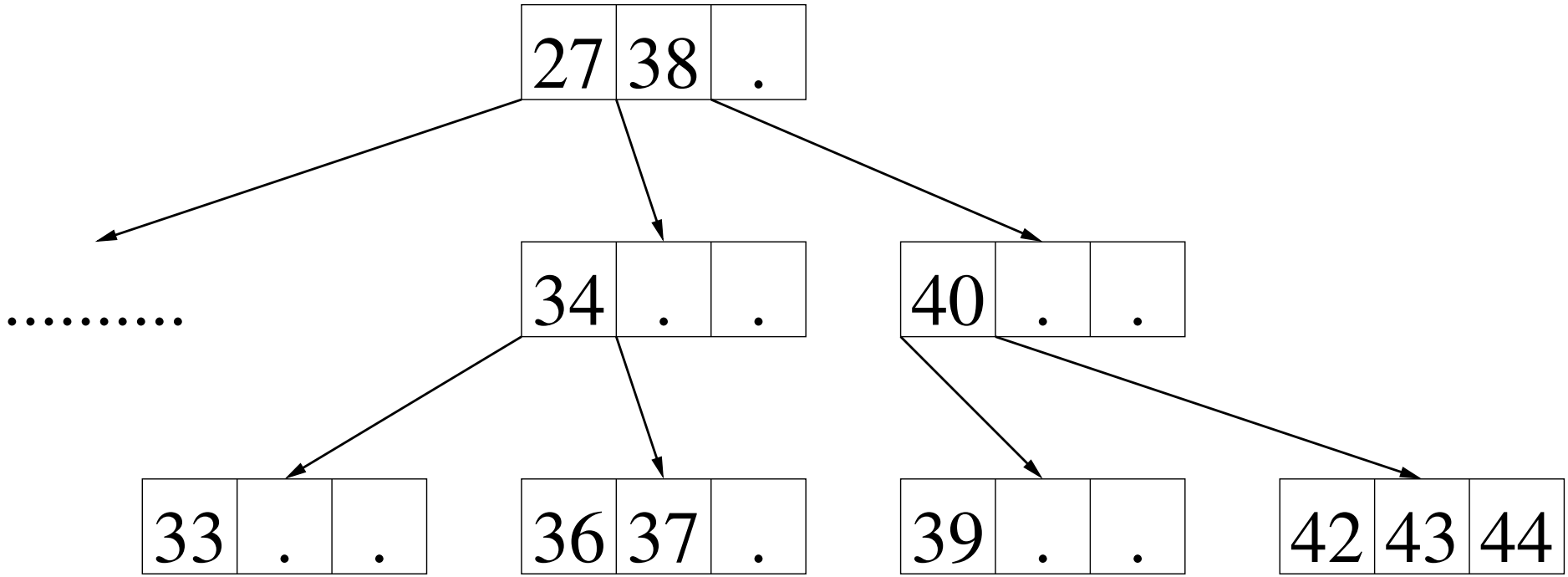
Teilen eines Knotens



Voraussetzungen für das Teilen eines Knotens x:

1. x ist voll (hat $2t - 1$ Schlüssel)
2. Elternknoten von x ist nicht voll

Teilen eines Knotens

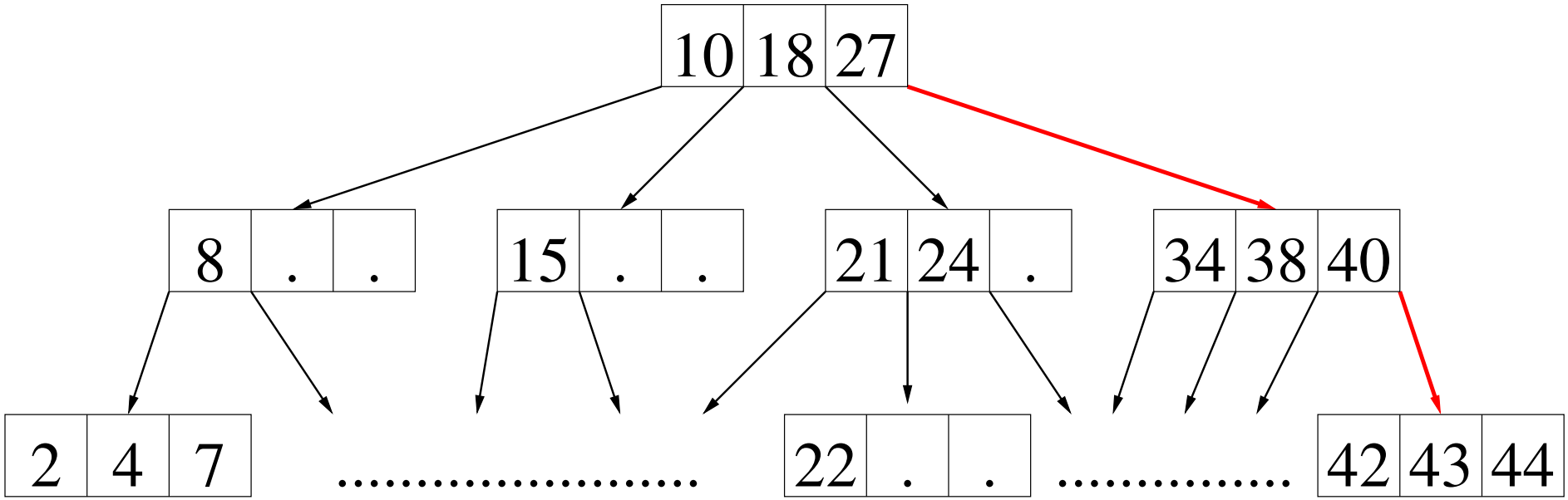


Voraussetzungen für das Teilen eines Knotens x:

1. x ist voll (hat $2t - 1$ Schlüssel)
2. Elternknoten von x ist nicht voll

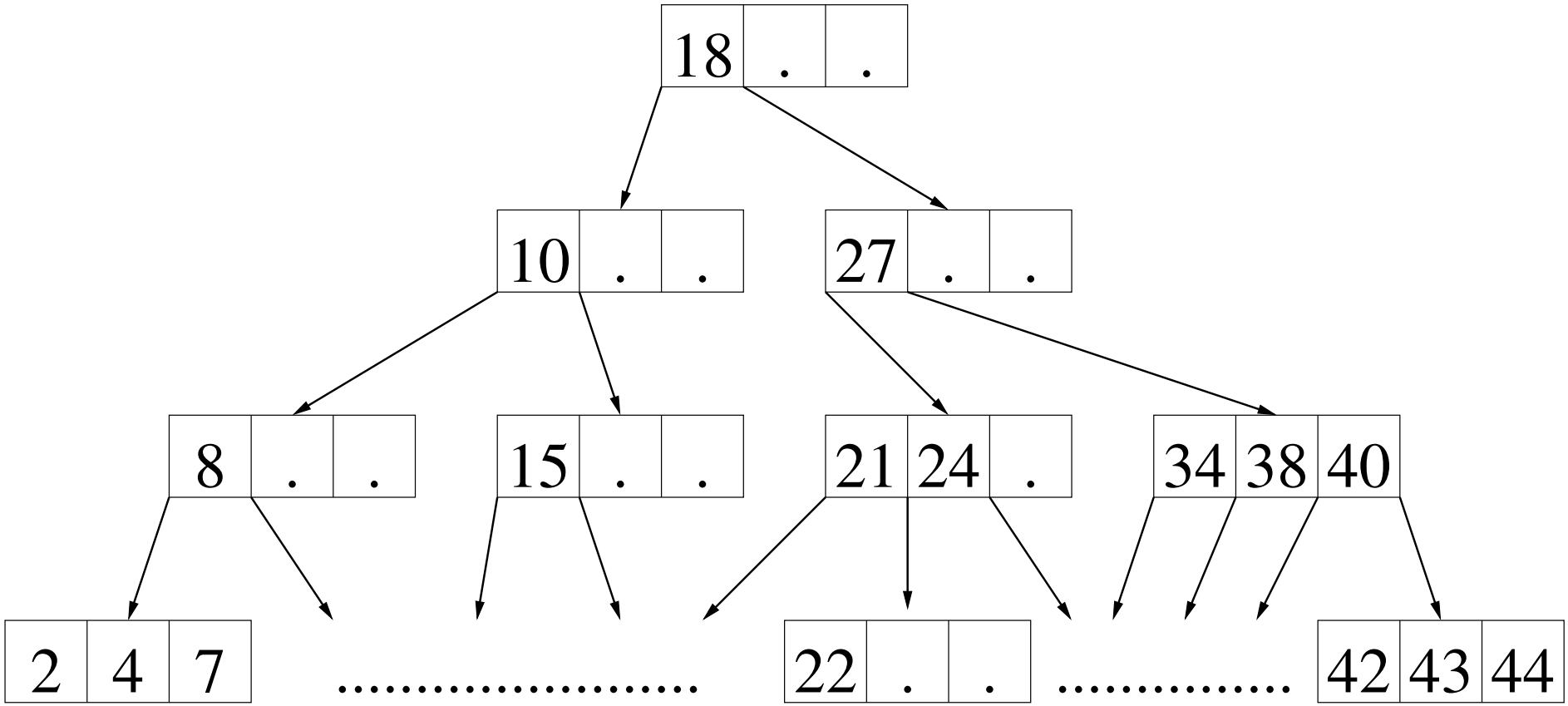
Beispiel: Einfügen im B-Baum

Einfügen von $k = 45$



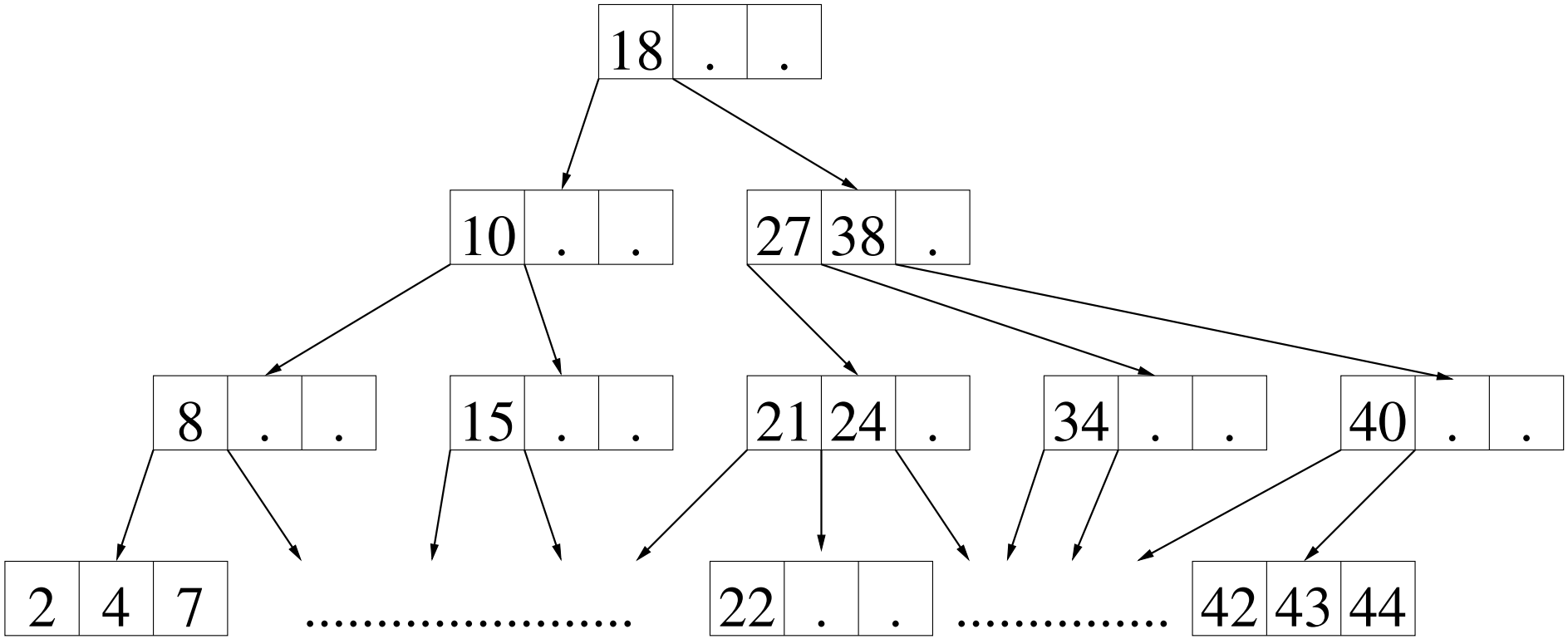
Beispiel: Einfügen im B-Baum

Einfügen von $k = 45$



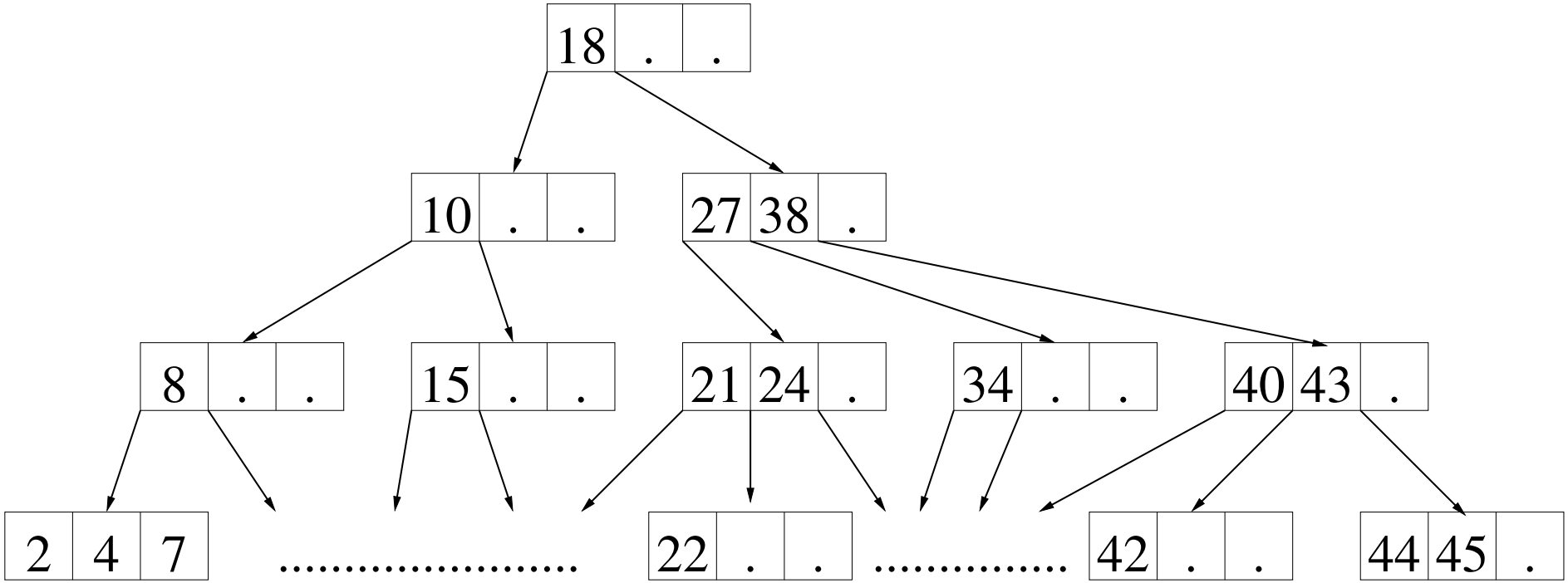
Beispiel: Einfügen im B-Baum

Einfügen von $k = 45$



Beispiel: Einfügen im B-Baum

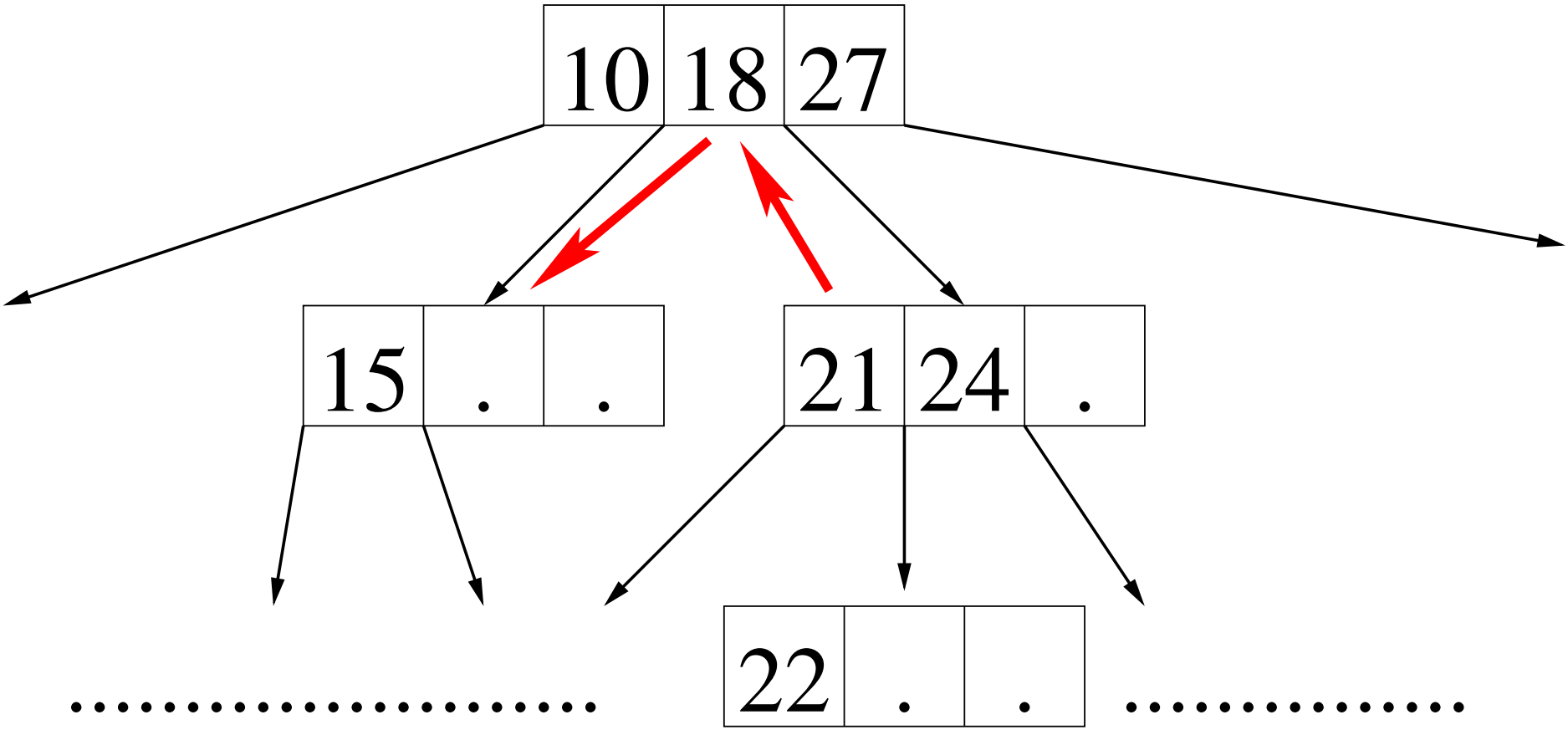
Einfügen von $k = 45$



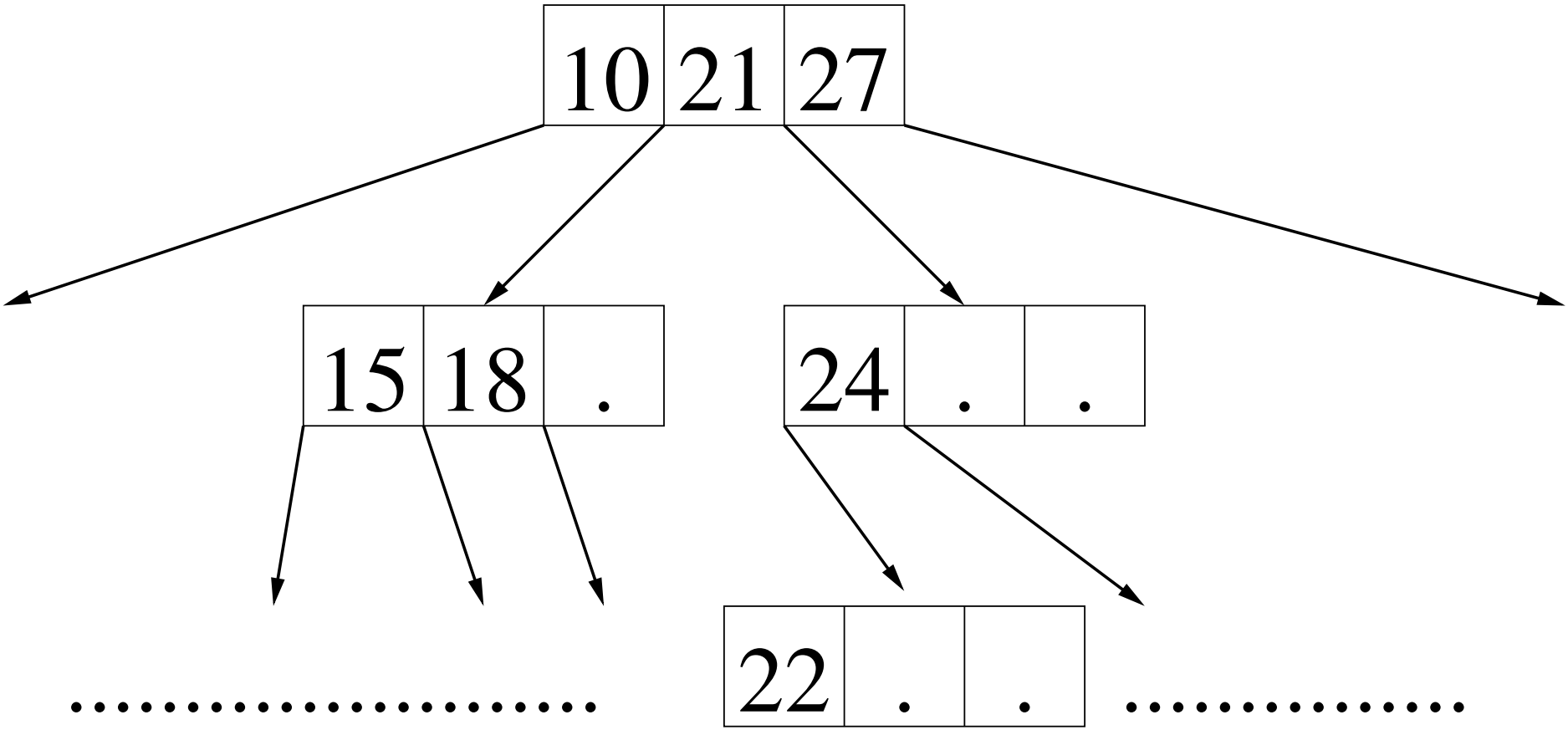
Löschen im B-Baum

- Löschen eines Schlüssels k beginnt mit einer Suche nach k, Durchlauf beginnend an der Wurzel.
- Trifft der Durchlauf auf einen Knoten y mit $t-1$ Schlüsseln (Minimalzahl), so wird y durch *Verschieben* eines Schlüssels auf t Schlüssel gefüllt, falls möglich. Ansonsten wird y mit einem Nachbarn *verschmolzen*.
- Durch das Verschieben/Verschmelzen hat der Knoten, aus dem k gelöscht wird, mindestens t Schlüssel, nach dem Löschen also mindestens $t-1$.
- Liegt k in einem Blatt, wird der Schlüssel k direkt entfernt. Andernfalls, suche den grössten Schlüssel k' im linken (oder kleinsten, im rechten) zum Schlüssel gehörigen Teilbaum. Dann ersetze k durch k' und lösche k'. Anmerkung: k' muss in einem Blatt liegen.

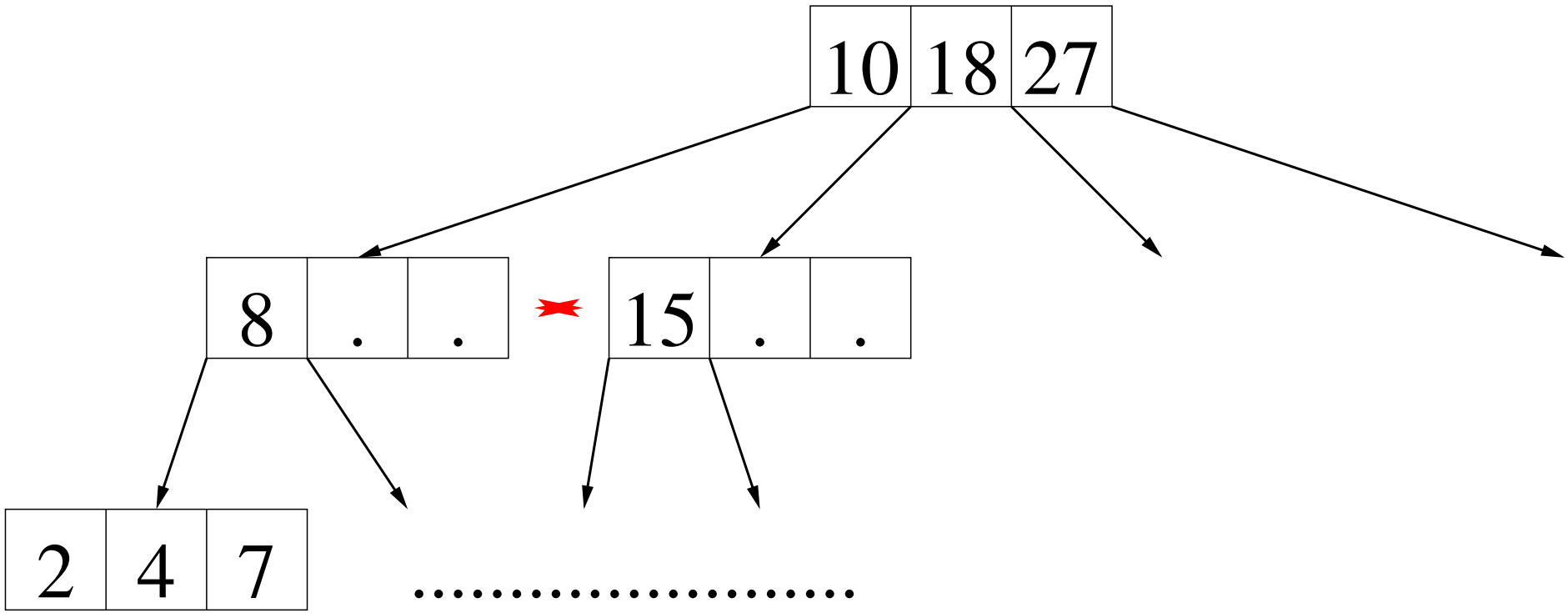
Verschieben eines Schlüssels



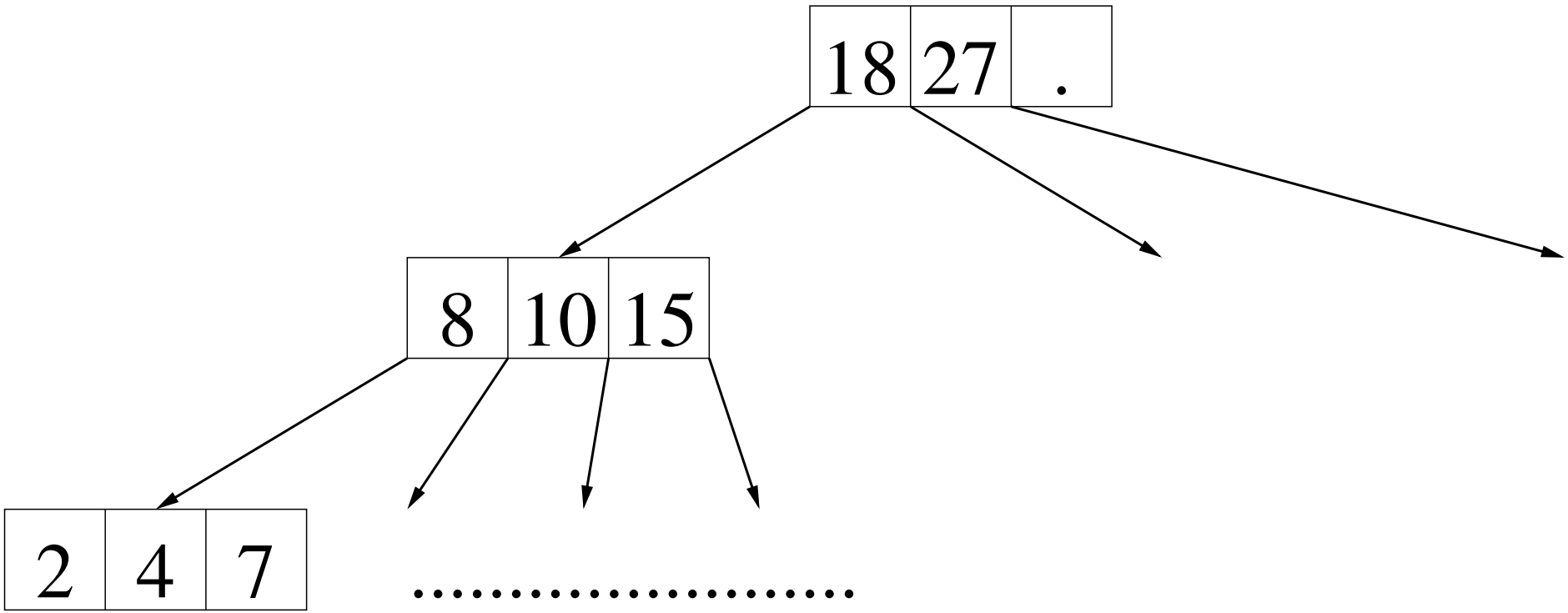
Verschieben eines Schlüssels



Verschmelzen zweier Knoten



Verschmelzen zweier Knoten



Höhe von B-Bäumen

Betrachte B-Baum der Höhe h mit minimalem Verzweigungsgrad t.

- 1 Knoten auf Stufe 0, mind. 2 Knoten auf Stufe 1
- mindestens $2t^{i-1}$ Knoten auf Stufe $i \geq 2$
- Gesamtzahl Knoten ist mindestens

$$n \geq 1 + 2 + 2 \sum_{i=2}^{h-1} t^{i-1}$$

- führt zu oberer Schranke für Höhe:

$$h \leq 1 + \log_t \frac{n+1}{2}$$

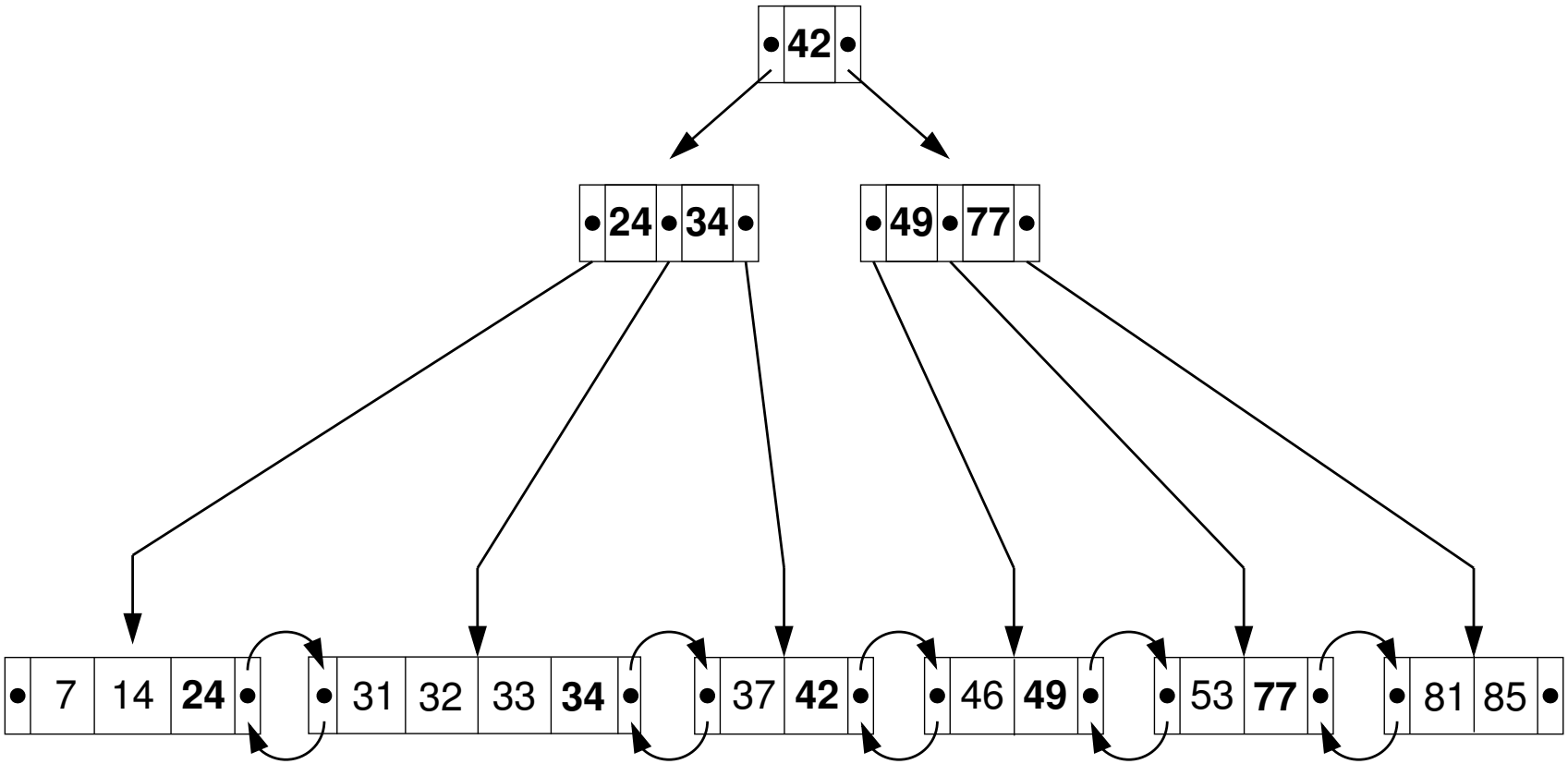
  $\Rightarrow$ Höhe wächst logarithmisch mit Zahl der Schlüssel
- Im Vergleich zum Binärbaum kann Höhe deutlich reduziert werden durch große Werte von t (Seitengröße).
- Beispiel: $t = 1024$ reduziert Höhe um Faktor 10 im Vergleich mit Binärbaum.

Variante: 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.
- 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*-Bäume: Beispiel



B- und B*-Bäume: Quantitativer Vergleich

Realistisches Beispiel:

- Seitengröße 2048 Byte,
- Hilfsinformation und Schlüssel 4 Byte lang
- Separate Speicherung der Daten, im Baum also nur Zeiger (4 Byte)
- Im B-Baum: Verzweigungsgrad zwischen 86 und 171
- Im B*-Baum: Verzweigungsgrad zwischen 127 und 255

Speicherkapazität für Bäume der Höhe 4:

- B-Baum: Zwischen 1.272.122 und 855.063.083 Datensätze
- B*-Baum: Zwischen 4.161.536 und 4.211.669.268 Datensätze

Fazit: Auch für sehr große Datenmengen sind ca. sieben Speicherzugriffe (d.h. wir haben einen Absolutwert als Maximun, keine Wachstumsfunktion).
Wenn der obere Teil des Baumes im Hauptspeicher gehalten wird, verringert sich die Zahl der Zugriffe auf den Externspeicher weiter!

Digitale Suchbäume I

Prinzip digitaler Suchbäume (kurz: Digitalbäume):

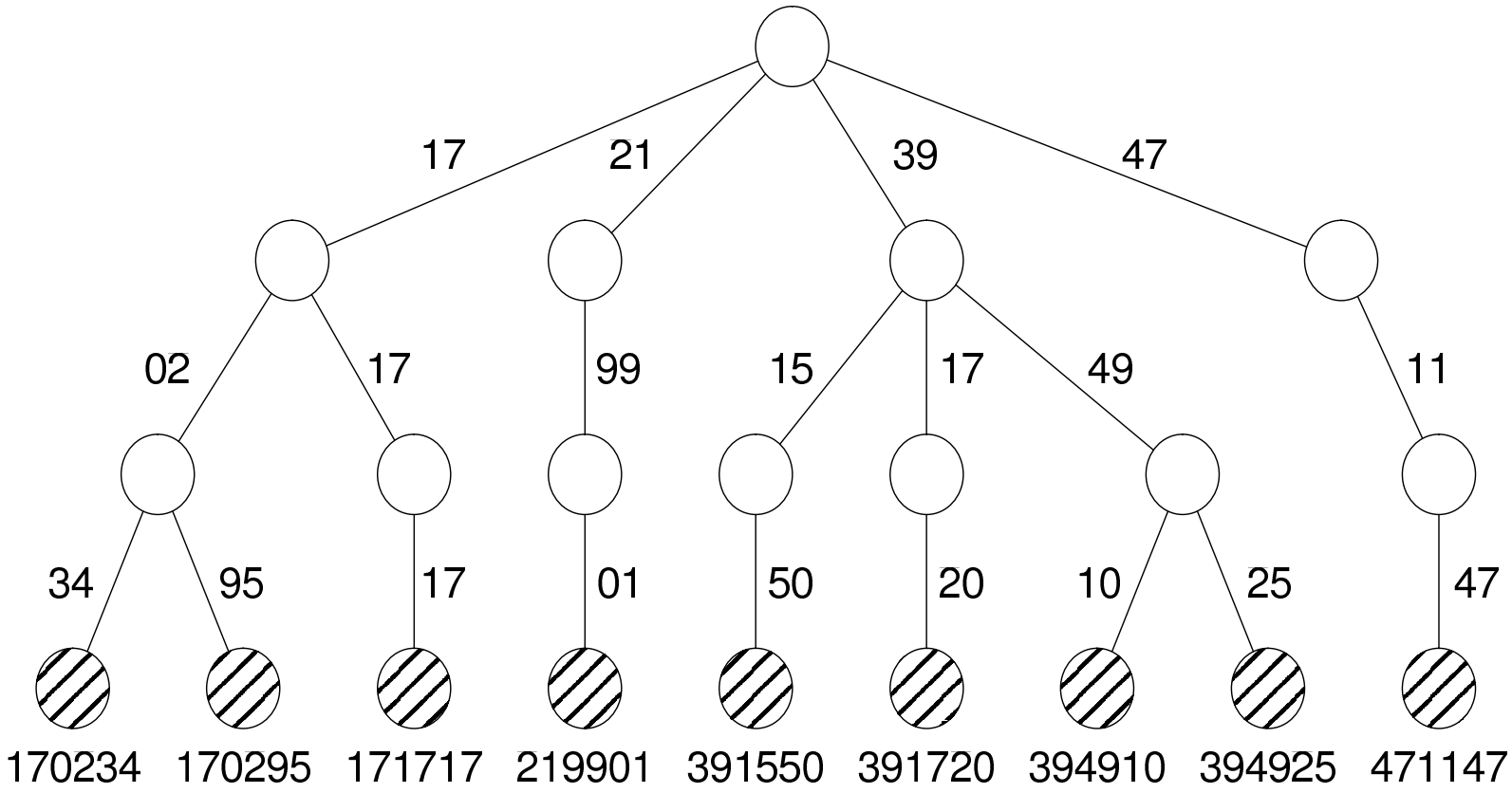
- Zerlegung des Schlüssels - bestehend aus Zeichen eines Alphabets - in Teile
- Aufbau des Baumes nach Schlüsselteilen
- Suche im Baum durch Vergleich von Schlüsselteilen
- jede unterschiedliche Folge von Teilschlüsseln ergibt eigenen Suchweg im Baum
- alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg

Digitale Suchbäume II

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 $= \frac{l}{k} + 1$, wenn l die max. Schlüssellänge und k die Schlüsselteillänge ist

Digitale Suchbäume: Beispiel



m-ärer Trie

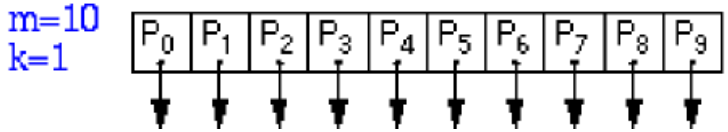
Spezielle Implementierung des Digitalbaumes: Trie
Trie leitet sich ab von "Information Re*trie*val" (E.Fredkin, 1960)

Tries sind spezielle m-Wege-Bäume, bei denen Zeiger für alle möglichen Schlüsselteile (=Kantenlabel) abgespeichert werden.

- bei Ziffern: m $= |\Sigma| = 10$
- bei Buchstaben: m $= |\Sigma| = 26$;
  bei alphanumerischen Zeichen: $|\Sigma| = 36$
- bei Schlüsselteilen der Länge k potenziert sich Grad entsprechend, d.h. als Grad ergibt sich $m = |\Sigma|^k$

Trie-Darstellung

- Jeder Knoten eines Tries vom Grad m ist ein Vektor mit m Zeigern
- Jede Vektorposition entspricht einem der $m = |\Sigma|^k$ möglichen Schlüsselteile. Die Schlüsselteile werden nicht explizit gespeichert.
- Beispiel: Knoten eines 10-ären Tries mit Ziffern als Schlüsselteilen

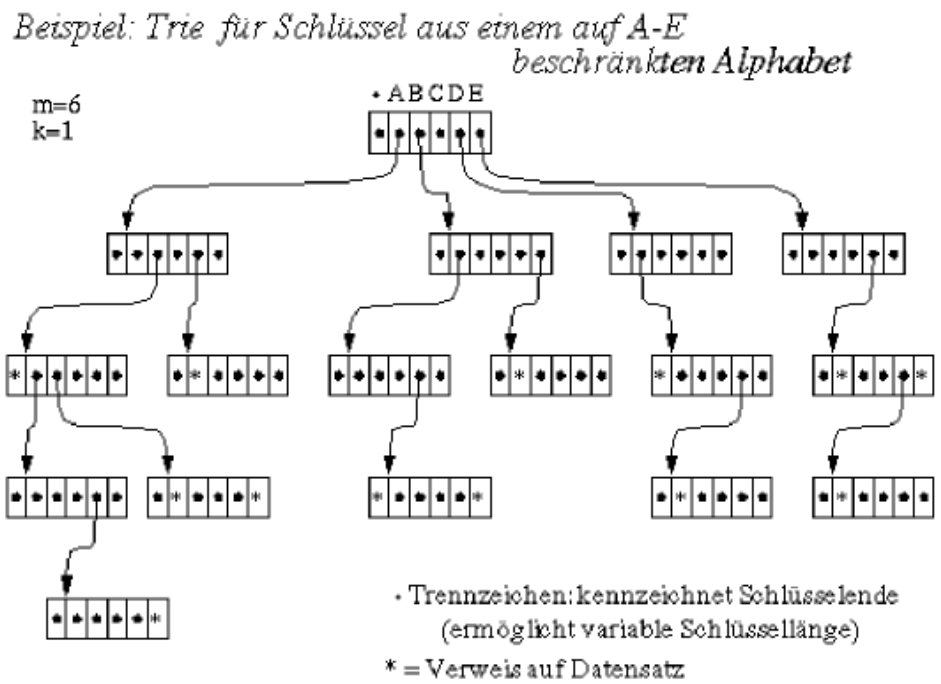


  im Beispiel: P_i ist der Zeiger für Ziffer i.
- In einem Knoten ist P_i entweder NULL oder zeigt auf den Unterbaum, der im dargestellten Digitalbaum mit i-tem Schlüsselteil (d.h. im Beispiel Ziffer i) gelabelt an diesem Knoten hängt.

Grundoperationen auf Tries I

Direkte Suche: In der Wurzel wird der Zeiger zum 1-ten Schlüsselteil bestimmt. Falls ungleich NULL, wird er verfolgt. Im nächsten Knoten wird mit dem 2-ten Schlüsselteil fortgefahren usw.

- Aufwand bei erfolgreicher Suche nach Schlüssel i: $\lceil \ell_i / k \rceil$
- effiziente Bestimmung der Abwesenheit eines Schlüssels



Grundoperationen auf Tries II

- **Einfügen:** Wenn Suchpfad schon vorhanden, wird NULL-Zeiger in Zeiger auf Datensatz umgewandelt, sonst Einfügen von neuen Knoten
- **Löschen:** Nach Aufsuchen des richtigen Knotens wird ein Datensatz-Zeiger auf NULL gesetzt. Besitzt daraufhin der Knoten nur NULL-Zeiger, wird er aus dem Baum entfernt (rekursive Überprüfung der Vorgängerknoten)

Eigenschaften von Tries

Anmerkungen:

- Triehöhe ist bestimmt durch längsten abgespeicherten Schlüssel
- Gestalt des Baumes hängt von der Gesamtheit der Schlüssel, d.h. von ihrer Verteilung, 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)

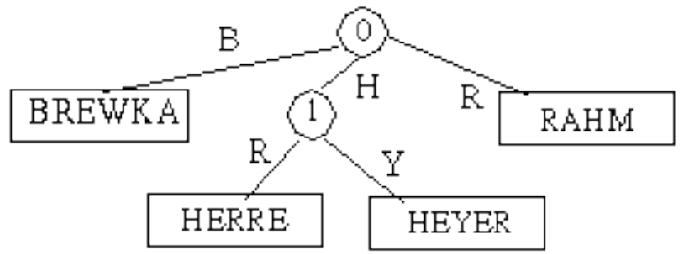
Möglichkeiten der Kompression

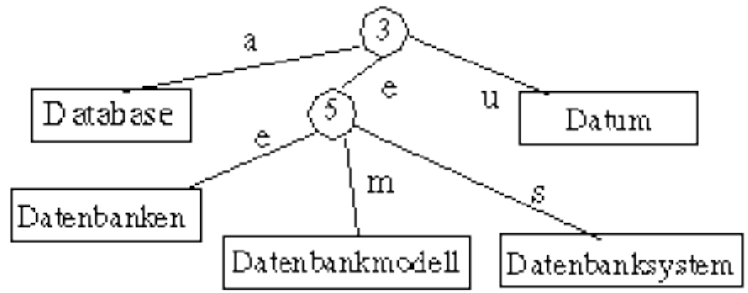
- Sobald ein Zeiger auf einen Unterbaum mit nur einem Schlüssel verweist, wird der (Rest-) Schlüssel in einem speziellen Knotenformat aufgenommen und Unterbaum eingespart $\rightarrow$ vermeidet Einweg-Verzweigungen
- Speichere nur besetzte Verweise (erfordert explizite Speicherung des zugehörigen Schlüsselteils, d.h. ,,trade-off")

PATRICIA-Tree (Practical Algorithm To Retrieve Information Coded In Alphanumeric)

Merkmale

- Speicherung der Schlüssel in den Blättern
- innere Knoten speichern, wie viele Zeichen beim Test zur Wegeauswahl zu überspringen sind
- Vermeidung von Einweg-verzweigungen, in dem bei nur noch einem verbleibenden Schlüssel direkt auf entsprechendes Blatt verwiesen wird
- üblicherweise Binärdarstellung für Schlüsselwerte, d.h. Alphabet $\{0, 1\}$ $\rightarrow$ binärer Digitalbaum

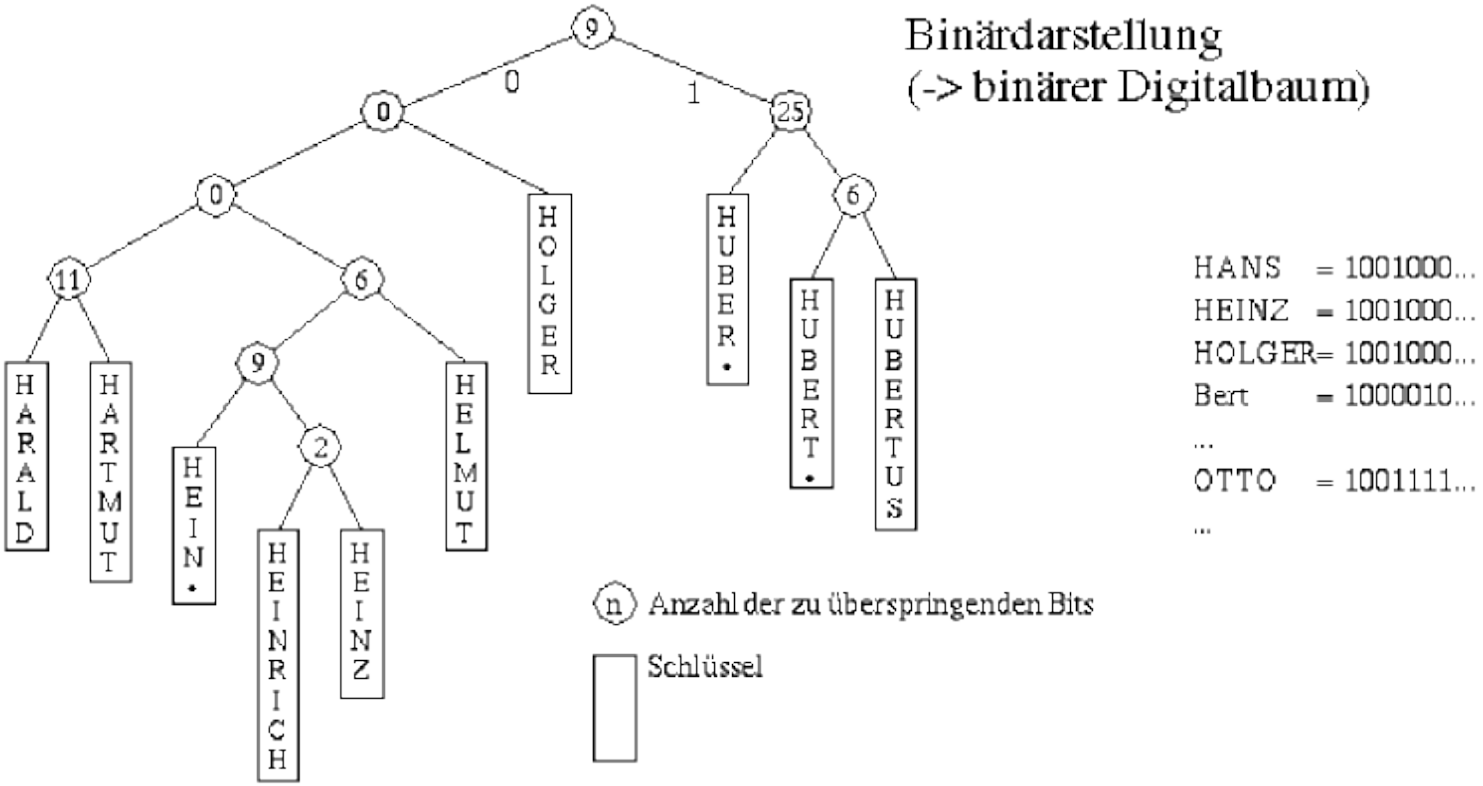




Eigenschaften von PATRICIA-Trees

- 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

Beispiel: PATRICIA-Tree



Suche nach Schlüssel HEINZ = `X'10010001000101100100110011101011010'`

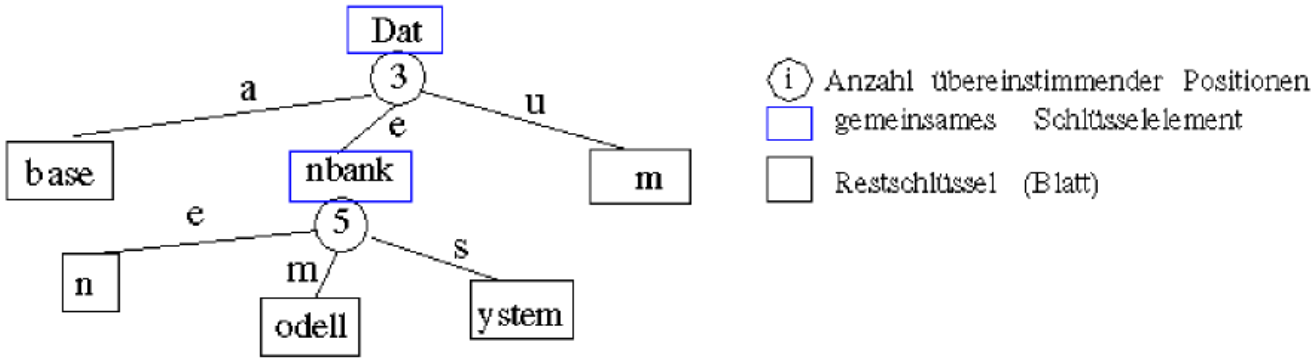
Suche nach ABEL = `X'10000011000010100010110011100'` **?**

Beobachtung: Erfolgreiche und erfolglose Suche endet in Blatt

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 “Mehrwegebäume” I

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

Zusammenfassung “Mehrwegebäume” II

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 wenig ($< 1\%$)
- Löschalgorithmus ist einfacher
- Verkettung der Blattseiten ergibt schnellere sequentielle Verarbeitung

Zusammenfassung “Mehrwegebäume” III

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