

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

***Anfragebearbeitung und
Indexstrukturen in
Datenbanksystemen***

im Wintersemester 2013/2014

Prof. Dr. Hans-Peter Kriegel

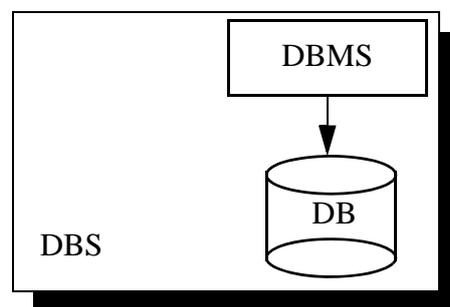
1 Einleitung

Um die Anfragen und Operationen in einem Datenbanksystem zu unterstützen, müssen die Daten durch entsprechende *Datenstrukturen* und *Speicherungsverfahren* geeignet organisiert werden. Datenstrukturen und Speicherungsverfahren, die den besonderen Anforderungen von Datenbanksystemen gerecht werden, werden als *Index- und Speicherungsstrukturen* bezeichnet.

1.1 Datenbanksysteme

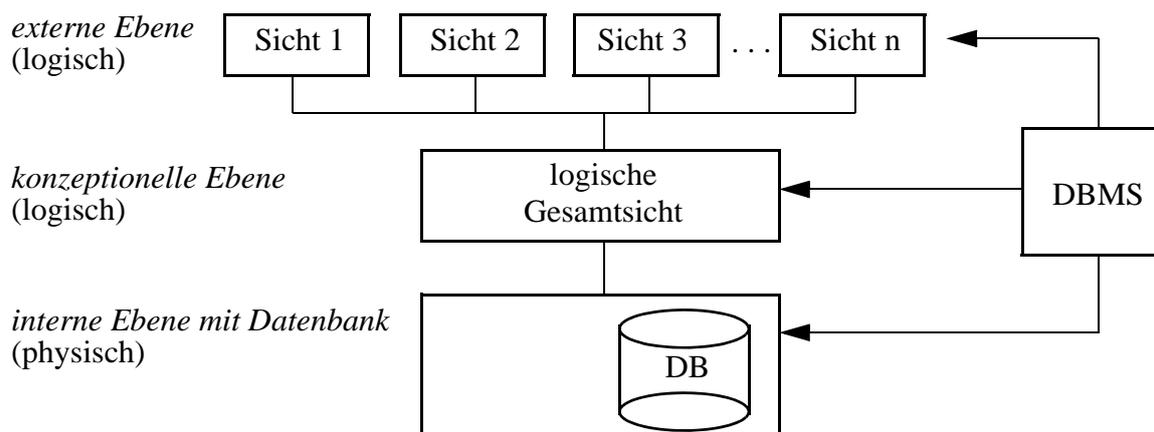
Ein *Datenbanksystem (DBS)* besteht aus:

- **Datenbank (DB)**
Sammlung aller gespeicherten Daten samt ihrer Beschreibung.
- **Datenbank-Managementsystem (DBMS)**
Programmsystem, das die DB verwaltet, fortschreibt und alle Zugriffe auf die DB regelt.



Architekturebenen eines Datenbanksystems:

- **Externe Ebene**
umfaßt alle *individuelle Sichten von Benutzern* oder Benutzergruppen auf die Datenbank.
- **Konzeptionelle Ebene**
repräsentiert die *logische Gesamtsicht* aller Daten in der Datenbank.
- **Interne Ebene**
legt *internen Aufbau und Zugriff auf die Daten* fest (*physische Datenorganisation*).



Beschreibung der Ebenen:

- durch *externes, konzeptionelles und internes Schema*.

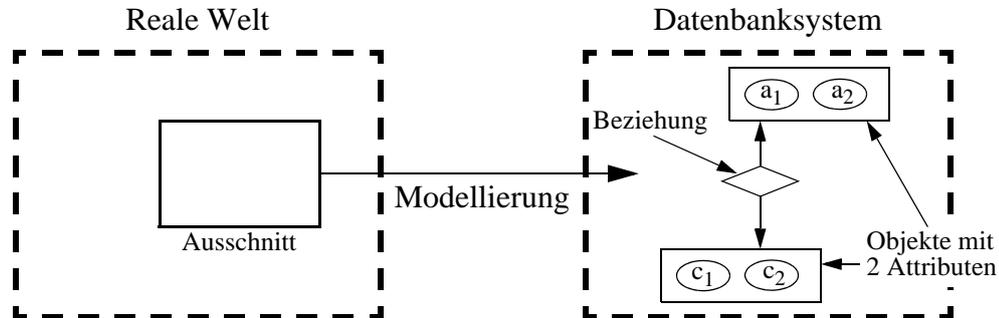
Verbindung der Ebenen:

- durch *Transformationen*
(Schnittstellen, die die einzelnen Anfragen, Operationen und Resultate transformieren).

1.2 Objekte

Modellbildung

Für die Abbildung eines Ausschnittes der realen Welt in ein DBS ist eine Abstraktion, eine **Modellierung** dieser Welt notwendig.



Beschreibung der Welt im Modell durch

- **Objekte (Entities)** mit
- **Eigenschaften (Attributen)** und
- **Beziehungen** untereinander.

Datensätze

- Objekte werden durch eine Reihe von Eigenschaften (**Attribute**) beschrieben.
Anzahl der Sitzplätze
- Jedem Attribut a ist ein **Wertebereich (Domain)** A zugeordnet.
CARDINAL
- Damit kann die *Ausprägung eines bestimmten Objektes* als *geordnetes k -Tupel* (a_1, \dots, a_k) beschrieben werden.

a_i ist der **Attributswert** des i -ten Attributs; es gilt $a_i \in A_i$. k ist die **Dimension**.

(VW, Passat, silbergrau, 5)

Dieses Tupel wird **logischer Datensatz (Record)** genannt.

- In der Datenbank physisch gespeicherte Datensätze werden oft um Verwaltungsinformationen angereichert und komprimiert (**physische Datensätze**).
(VW, Passat, silbergrau, 5) \rightarrow (4890, 67, 203, 5)

Dateien

- Gleichartige Objekte (d.h. mit gleichen Attributen) werden auf *logischer Ebene* in **Entity-Sets** zusammengefaßt.
{ (VW, Passat, silbergrau, 5), (VW, Golf, schwarz, 4), (VW, Jetta, schwarz, 5), ... }
- Datensätze eines Entity-Sets werden i.allg. auf *physischer Ebene* in einer **Datei (File)** zusammengefaßt und gespeichert.

1.3 Anfragen

Schlüssel

- **Schlüssel (Key)**
Ein oder mehrere Attribute, die einen Datensatz in einer Datei eindeutig identifizieren.
(Name, Vorname, Geburtsdatum, Geburtsort)
- **Primärschlüssel**
Ein Attribut, das einen Datensatz in einer Datei eindeutig identifiziert.
(Fahrgestellnummer)
- **Sekundärschlüssel / Suchschlüssel**
Ein oder mehrere Attribute, nach denen ein Datensatz gesucht werden kann.
Sie brauchen einen Datensatz aber nicht eindeutig zu identifizieren.
(Fahrzeugmodell, Farbe)

Anfragen

- **Primärschlüsselsuche**
Eine Anfrage über ein Suchattribut x_i , wobei x_i Primärschlüssel ist.
- **Sekundärschlüsselsuche / Multiattributssuche**
Eine Anfrage über einen Sekundärschlüssel.

Typen von Anfragen:

- **Exact Match Query**
(x_1, \dots, x_k)
spezifiziert k Attribute exakt.
- **Partial Match Query**
($*, x_{i1}, *, \dots, x_{is}, \dots, x_{is}$)
spezifiziert Werte für $s < k$ Attribute; $k-s$ Attribute bleiben unspezifiziert (*).
- **Range Query**
($[u_1, o_1], \dots, [u_k, o_k]$)
spezifiziert k Bereiche mit $u_i \leq o_i, 1 \leq i \leq k$.
- **Partial Range Query**
($*, [u_{i1}, o_{i1}], *, \dots, [u_{is}, o_{is}]$)

Nichtstandard-Datenbanksysteme müssen weitere Anfragen unterstützen (siehe später).

Kernfrage:

Wie finden wir effizient (schnell) die gewünschten Daten in einer Datenbank ?

Lösungsprinzip:

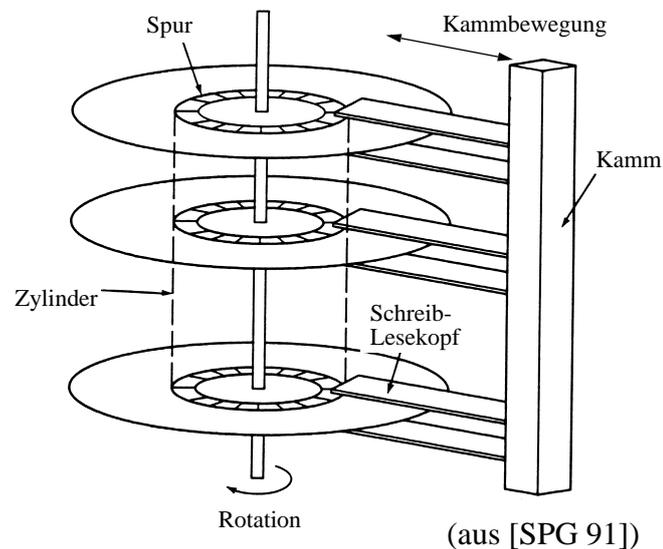
Durch Einsatz geeigneter Datenstrukturen und Speicherungsverfahren,
d.h. *Index- und Speicherungsstrukturen*.

Physischer Aufbau eines Magnetplattenspeichers

- Eine Reihe übereinanderliegender, rotierender Magnetplatten.
- Strukturierung:

Zylinder, Spur und Sektor

- Zugriff erfolgt über einen Kamm mit Schreib-/Leseköpfen, der quer zur Rotation bewegt wird.

**Zugriff auf Seiten****Phasen:**

- *Positionierung des Schreib-/Lesekopfes*
Zeit für die Kambewegung [3,9 ms]
- *Warten auf den Sektor / Seite*
Ø halbe Rotationszeit der Platte [2 ms]
- *Übertragung der Seite*
Zeit für Schreiben bzw. Lesen [0,02 - 0,04 ms/ 4KByte Seite]
- *Kontrolle der Übertragung*
Zeit des Platten-Controllers [0,1 ms]

In eckigen Klammern sind die Zeiten einer typischen Server-Platte angegeben (Seagate Cheetah 15K.7, 600 GBytes).

Zeit für Zugriff auf eine Seite >> Zeit für Operation im Hauptspeicher !

Entwicklung:

- Die Kapazität von Plattenspeichern hat sich drastisch erhöht (Faktor 100 in den letzten 10 Jahren)
- Die Zugriffszeit hat sich relativ dazu kaum verändert (Faktor 2 in den letzten 10 Jahren).

Leistungsmaß:

- Annahme: der Zugriff auf Seiten erfolgt unabhängig voneinander.

⇒ Die Anzahl der Seiten, auf die zugegriffen wird, ist ein geeignetes Leistungsmaß für spätere Untersuchungen.

1.5 Indexstrukturen

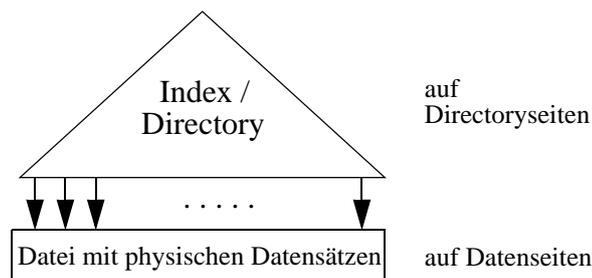
Um Anfragen und Operationen effizient durchführen zu können, setzt die interne Ebene des Datenbanksystemes geeignete Datenstrukturen und Speicherungsverfahren (*Indexstrukturen*) ein.

Aufgaben

- *Zuordnung eines Suchschlüssels* zu denjenigen physischen Datensätzen, die diese Wertekombination besitzen,
d.h. Zuordnung zu der oder den Seiten der Datei, in denen diese Datensätze gespeichert sind.
(VW, Golf, schwarz, M-ÜN 40) → (logische) Seite 37
- *Organisation der Seiten* einer Datei unter dynamischen Bedingungen.
Überlauf einer Seite ⇒ Aufteilen der Seite auf zwei Seiten

Index / Directory

- *Strukturinformation* zur Zuordnung von Suchschlüsseln und zur Organisation der Datei.
 - **Directoryseiten**
Seiten in denen das Directory gespeichert wird.
 - **Datenseiten**
Seiten mit den eigentlichen physischen Datensätzen.



Klassen

- **Datenorganisierende Strukturen**
Organisiere die Menge der tatsächlich auftretenden Daten (Suchbaumverfahren).
- **Raumorganisierende Strukturen**
Organisiere den Raum, in den die Daten eingebettet sind (dynamische Hash-Verfahren).
- **Hybride Strukturen**
Kombination beider Vorgehensweisen (*Hash-Bäume*).

1.6 Allgemeine Anforderungen an Indexstrukturen

- **Effizientes Suchen**

Häufigste Operation in einem DBS: *Suchanfragen*.

⇒ Insbesondere Suchoperationen müssen mit wenig Seitenzugriffen auskommen.

Beispiel: unsortierte sequentielle Datei

- Einfügen und Löschen von Datensätzen werden effizient durchgeführt.
- Suchanfragen müssen ggf. die gesamte Datei durchsuchen.

⇒ Eine Anfrage sollte daher mit Hilfe der Indexstruktur möglichst schnell zu der Seite oder den Seiten hingeführt werden, wo sich die gesuchten Datensätze befinden.

- **Dynamisches Einfügen, Löschen und Verändern von Datensätzen**

Der Datenbestand einer Datenbank verändert sich im Laufe der Zeit.

⇒ Verfahren, die zum Einfügen oder Löschen von Datensätzen eine Reorganisation der gesamten Datei erfordern, sind nicht akzeptabel.

Beispiel: sortierte sequentielle Datei

- Das Einfügen eines Datensatzes erfordert im schlechtesten Fall, daß alle Datensätze um eine Position verschoben werden müssen.
- Folge: auf alle Seiten der Datei muß zugegriffen werden.

⇒ Das Einfügen, Löschen und Verändern von Datensätzen darf daher nur *lokale Änderungen* bewirken.

- **Ordnungserhaltung**

Datensätze, die in ihrer Sortierordnung direkt aufeinander folgen, werden oft gemeinsam angefragt.

⇒ In der Ordnung aufeinanderfolgende Datensätze sollten in der gleichen Seite oder in benachbarten Seiten gespeichert werden.

- **Hohe Speicherplatzausnutzung**

Dateien können sehr groß werden.

⇒ Eine möglichst *hohe Speicherplatzausnutzung* ist wichtig:

- Möglichst geringer Speicherplatzverbrauch.
- Im Durchschnitt befinden sich mehr Datensätze in einer Seite, wodurch auch die Effizienz des Suchens steigt und die Ordnungserhaltung an Bedeutung gewinnt.

- **Implementierbarkeit, Nutzen des Implementationsaufwandes**

- *Implementierbarkeit*

- *Robustheit*

- Der Mehraufwand der Implementierung einer komplexen Struktur A gegenüber einer einfachen Struktur B sollte in einem vertretbaren Verhältnis zum Nutzen (Effizienzgewinn) stehen.

1.7 Literatur

Derzeit gibt es kein Lehrbuch, welches sich ausschließlich mit Index- und Speicherungsstrukturen für Datenbanksysteme befaßt. Somit ist man in der Regel auf die Originalliteratur angewiesen, in der die Verfahren vorgestellt wurden. Die entsprechenden Literaturstellen werden in Rahmen dieses Skriptes genannt.

In den meisten *Lehrbüchern über Datenbanksysteme oder Datenstrukturen* gibt es Abschnitte oder Kapitel, die sich nur sehr oberflächlich mit Indexstrukturen für Standard-Datenbanksysteme befassen.

Über die Speicherung von Daten und Dateien auf dem Sekundärspeicher kann man in *Literatur über Betriebssysteme oder Rechnerarchitektur* nachlesen, beispielsweise in:

[HPG 02] Hennessy J.L., Patterson D.A., Goldberg D.: *Computer Architecture: A Quantitative Approach*, ISBN 1558605967, 2002.

[Sie 90] Sierra, H.M.: *An Introduction to Direct Access Storage Devices*, Academic Press, 1990.

[SGG 02] Silberschatz A., Galvin P., Gagne G.: *Operating System Concepts, Sixth Edition*, John Wiley & Sons, 2002.

[Tan 02] Tanenbaum A.S.: *Moderne Betriebssysteme, 2. Auflage*, Pearson Studium, 2002.

2 Baumstrukturen zur Primärschlüsselsuche

An dieser Stelle werden B-Bäume und B*-Bäume als bekannt vorausgesetzt.

Häufig tritt in Datenbankanwendungen neben der Primärschlüsselsuche auch sequentielle Verarbeitung auf.

Beispiele für sequentielle Verarbeitung:

- *Sortiertes Auslesen aller Datensätze*, die von einer Indexstruktur organisiert werden.
- *Unterstützung von Bereichsanfragen* der Form:
“Nenne mir alle Studenten, deren Nachname im Bereich [Be ... Brz] liegt.”

⇒ Die Indexstruktur sollte die *sequentielle Verarbeitung* unterstützen,

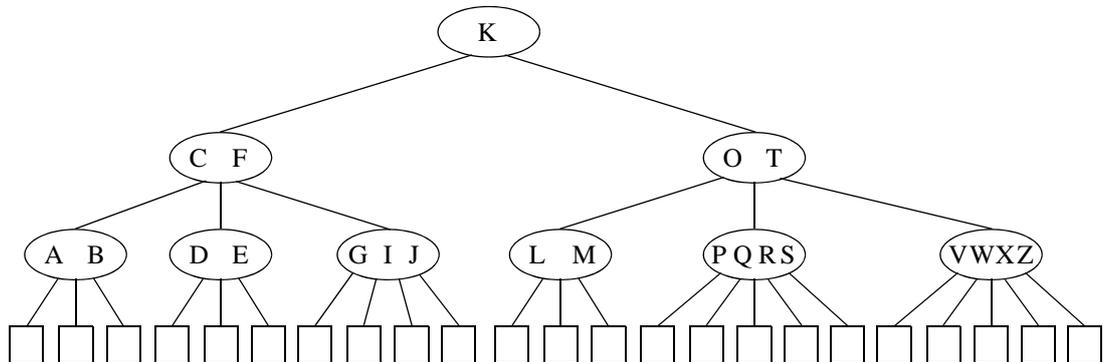
d.h. die Verarbeitung der Datensätze in aufsteigender Reihenfolge ihrer Primärschlüssel.

Betrachten wir das Beispiel eines B-Baumes:

Definition: B-Baum der Ordnung m (Bayer und McCreight (1972))

- (1) Jeder Knoten enthält höchstens $2m$ Schlüssel.
- (2) Jeder Knoten außer der Wurzel enthält mindestens m Schlüssel.
- (3) Die Wurzel enthält mindestens einen Schlüssel.
- (4) Ein Knoten mit k Schlüsseln hat genau $k+1$ Söhne.
- (5) Alle Blätter befinden sich auf demselben Level.

Beispiel: für einen B-Baum der Ordnung 2



Warum unterstützt dieser B-Baum die folgende Bereichsanfrage nicht effizient?

“Lies die Informationen aller Datensätze im Bereich [B ... V] aus”

Grundidee:

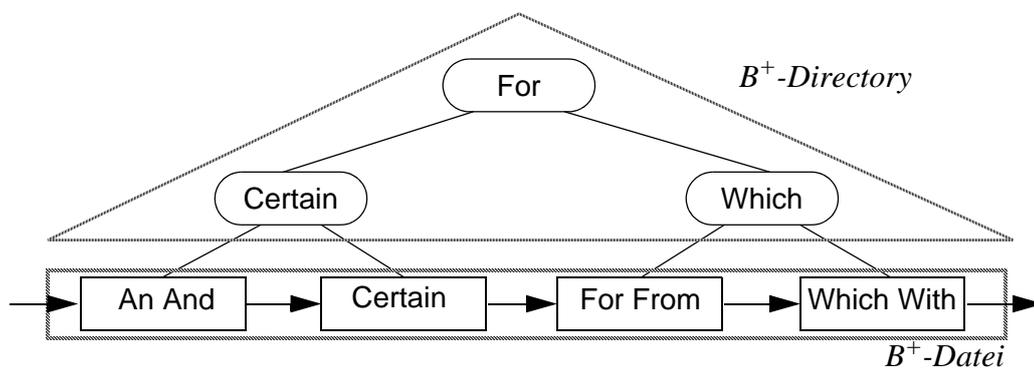
- Trennung der Indexstruktur in *Directory* und *Datei*.
- *Sequentielle Verkettung* der Daten in der Datei.

B⁺-Baum

- **B⁺-Datei:**
 - Die Blätter des B⁺-Baumes heißen *Datenknoten* oder *Datenseiten*.
 - Die Datenknoten enthalten alle Datensätze.
 - Alle Datenknoten sind entsprechend der Ordnung auf den Primärschlüsseln *verkettet*.

- **B^+ -Directory:**
 - Die inneren Knoten des B^+ -Baumes heißen *Directoryknoten* oder *Directoryseiten*.
 - Directoryknoten enthalten nur noch *Separatoren* s .
 - Für jeden Separator $s(u)$ eines Knotens u gelten folgende *Separatoreigenschaften*:
 - $s(u) > s(v)$ für alle Directoryknoten v im linken Teilbaum von $s(u)$.
 - $s(u) < s(w)$ für alle Directoryknoten w im rechten Teilbaum von $s(u)$.
 - $s(u) > k(v')$ für alle Primärschlüssel $k(v')$ und alle Datenknoten v' im linken Teilbaum von $s(u)$.
 - $s(u) \leq k(w')$ für alle Primärschlüssel $k(w')$ und alle Datenknoten w' im rechten Teilbaum von $s(u)$.

Beispiel: B^+ -Baum für die Zeichenketten: An, And, Certain, For, From, Which, With.



2.1 Erhöhung des Verzweigungsgrades: Präfix-Bäume

Wichtige Anforderungen an Suchbaumverfahren:

- *Geringe Höhe*

Die Anzahl der Seitenzugriffe für Anfragen und andere Operationen korrespondiert direkt mit der Höhe des Baumes.

⇒ Eine niedrige Baumhöhe ist anzustreben.

- *Hoher Verzweigungsgrad*

Ein wichtiges Kriterium, um die Höhe des Baumes zu beeinflussen, ist der *Verzweigungsgrad* der Knoten. Eine Erhöhung des Verzweigungsgrades schlägt sich in einer Reduktion der Höhe des Suchbaumes nieder.

⇒ Ein hoher Verzweigungsgrad ist anzustreben.

Separatoren mit Präfixeigenschaft: einfache Präfix-B⁺-Bäume

Beobachtung:

- Im B⁺-Baum werden *vollständige Schlüssel* als Separatoren verwendet.
- Kürzere Separatoren können die Separatoreigenschaften genauso gut erfüllen.
- Kürzere Separatoren erhöhen den Verzweigungsgrad.

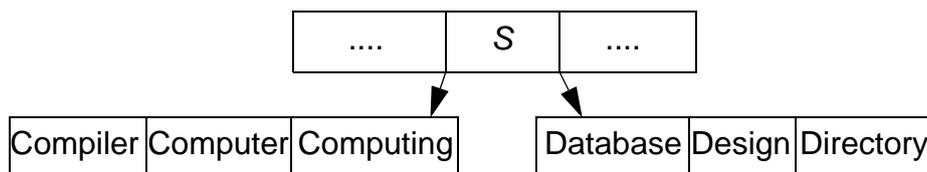
Beispiel:

Ein Datenknoten enthalte folgende Schlüssel und sei schon voll:

Compiler	Computer	Computing	Design	Directory
----------	----------	-----------	--------	-----------

Einfügen des Schlüssels "Database".

⇒ Der Knoten muß in zwei aufgespalten werden (*Split*):



Wir können jedes *S* als *Separator* wählen, das die Eigenschaft erfüllt:

$$\text{Computing} < S \leq \text{Database}$$

Präfix-Eigenschaft

Die Schlüssel seien Worte über einem Alphabet und die Ordnung der Schlüssel die lexikographische Ordnung. Dann gilt folgende *Präfix-Eigenschaft*:

Für zwei Schlüssel *x* und *y* mit $x < y$ gibt es einen *Präfix* \bar{y} von *y* mit:

- \bar{y} ist Separator zwischen *x* und *y*,
- kein anderer Separator zwischen *x* und *y* ist kürzer als \bar{y} .

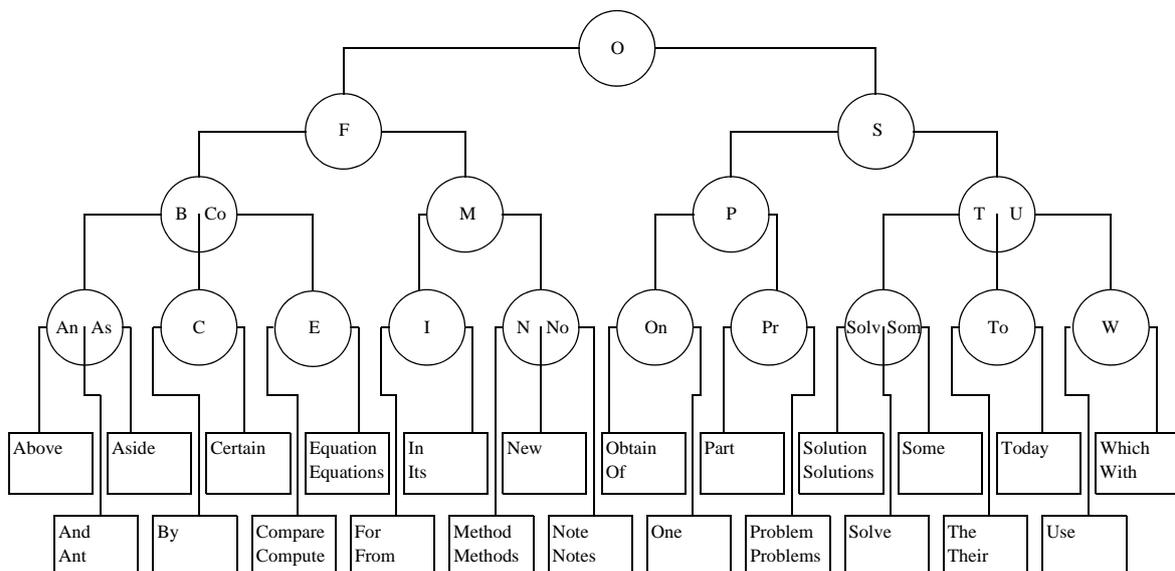
Beispiel (oben): $\bar{y} = D$.

Einfacher Präfix-B⁺-Baum [BU 77]

Ein *einfacher Präfix-B⁺-Baum* ist ein B⁺-Baum, dessen Directory aus einem B-Baum von Separatoren variabler Länge besteht, die die Präfix-Eigenschaft erfüllen.

Prinzip:

- Je kürzer die Separatoren in den Directoryknoten
- ⇒ desto höher der Verzweigungsgrad,
- ⇒ desto niedriger der Baum,
- ⇒ desto besser das Leistungsverhalten.

Beispiel:**Vorgehen beim Split von Knoten:**

- **Split eines Datenknotens**
Ein Separator mit Präfix-Eigenschaft wird gebildet und in den Vaterknoten eingefügt.
- **Split eines Directoryknotens**
Der Separator des überlaufenden Knotens, der die beiden neuen Knoten trennt, wird in den Vater eingefügt.

Split-Intervall

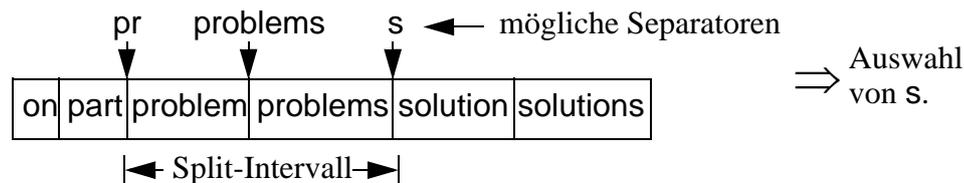
Ziel:

Erhöhung des Verzweigungsgrades.

Idee:

Man teilt eine Seite nicht genau in der Mitte auf, sondern wählt den Splitpunkt aus einem Bereich. Die Wahl des tatsächlichen Splitpunktes erfolgt gemäß der Länge der möglichen Separatoren.

Beispiel:



Split-Intervall σ

Gibt das Intervall an, in dem der Splitpunkt liegt.

Vorgehen:

- Auswahl des kürzesten Separators aus dem Split-Intervall.
- Aufteilen der Knoten gemäß dieses Separators.

σ_{dat} : Split-Intervall für Datenknoten

- je größer σ_{dat} , desto weniger Directoryknoten.

σ_{dir} : Split-Intervall für Directoryknoten

- je größer σ_{dir} , desto höher der Verzweigungsgrad nahe der Wurzel.

Nachteil:

Speicherplatzausnutzung kann unter 50 % sinken.

Frontkomprimierung: Präfix-B⁺-Bäume**Ziel:**

Weiteres Verkürzen der Separatoren.

Idee:

Zusammensetzen der Separatoren beim Durchlaufen des Baumes von der Wurzel zum gesuchten Blatt.

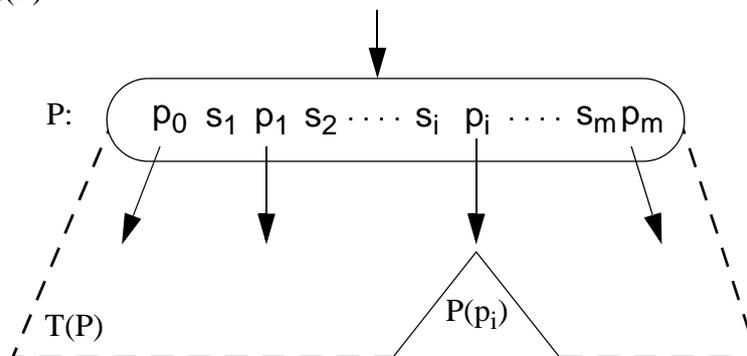
Beobachtung:

Für jeden Directoryknoten P in einem einfachen Präfix-B⁺-Baum gibt es *Schranken*:

- eine *größte untere Schranke* $\lambda(P)$ und
- eine *kleinste obere Schranke* $\mu(P)$,

s.d. für alle Schlüssel k und alle Separatoren s in T(P) (= Teilbaum mit Wurzel P) gilt:

- $\lambda(P) \leq k < \mu(P)$
- $\lambda(P) \leq s < \mu(P)$

**Berechnung von $\lambda(P)$ und $\mu(P)$:**

Wurzel:

- $\lambda(R) = b_0$ (b_0 ist der kleinste Buchstabe im Alphabet)
- $\mu(R) = \infty$ (∞ ist größer als jeder Buchstabe im Alphabet)

Sonst:

- $\lambda(P(p_i)) = \lambda(p_i) = \begin{cases} s_i & \text{für } i > 0 \\ \lambda(P) & \text{für } i = 0 \end{cases}$
- $\mu(P(p_i)) = \mu(p_i) = \begin{cases} s_{i+1} & \text{für } i < m \\ \mu(P) & \text{für } i = m \end{cases}$

Beobachtung:

Alle Separatoren und Schlüssel in $T(P(p_i))$ haben einen gemeinsamen (möglicherweise leeren) Präfix.

$g(p_i)$: der *längste gemeinsame Präfix aller Separatoren und Schlüssel in $T(P(p_i))$*

Berechnung des gemeinsamen Präfix $g(p_i)$:

\bar{k}_i : der längste gemeinsame Präfix von $\lambda(p_i)$ und $\mu(p_i)$
 (\bar{k}_i kann möglicherweise auch eine leere Zeichenkette sein).

Für $g(p_i)$ gilt:

$$g(p_i) = \begin{cases} \bar{k}_i b_j & \text{wenn } \lambda(p_i) = \bar{k}_i b_j z \text{ und } \mu(p_i) = \bar{k}_i b_{j+1}, \text{ wobei } z \text{ eine beliebige} \\ & \text{Zeichenkette ist und der Buchstabe } b_{j+1} \text{ dem Buchstaben } b_j \text{ direkt} \\ & \text{in der alphabetischen Reihenfolge folgt.} \\ \bar{k}_i & \text{sonst} \end{cases}$$

Beispiele

1. $T(P(p_i))$ enthalte die Schlüssel aab, aac, aaf, aag, aaz.

Seien $\lambda(p_i) = aab$, $\mu(p_i) = ab$, dann ist $g(p_i) = aa$ nach dem ersten obigen Fall.

2. $T(P(p_i))$ enthalte nun die Schlüssel aab, aac, aaf, aag, aah.

Seien $\lambda(p_i) = aab$, $\mu(p_i) = aai$, dann ist $g(p_i) = aa$ nach dem zweiten obigen Fall.

Bestimmung des Separators aus dem Präfix:

vollständiger Separator $s = g(p_i) \bar{s}$

- $\lambda(p_i)$, $\mu(p_i)$ und $g(p_i)$ können beim Durchlauf des Baumes von der Wurzel zum Knoten $P(p_i)$ abgeleitet werden.

⇒ Nur noch frontkomprimierte Separatoren sind abzuspeichern.

Präfix- B^+ -Baum [BU 77]

- in den Directoryknoten werden nur noch frontkomprimierte Separatoren abgespeichert.
- die Separatoren werden beim Traversieren abgeleitet.

Leistungsvergleich**Ersparnis von Seitenzugriffen bei Primärschlüsselsuche:**

einfacher Präfix- B^+ -Baum gegenüber B^+ -Baum: 25 %

Präfix- B^+ -Baum gegenüber einfachem Präfix- B^+ -Baum: 2 %

Zusätzlicher Aufwand für Implementierung und CPU-Zeit:

einfacher Präfix- B^+ -Baum gegenüber B^+ -Baum:

Speicherung und Suche variabel langer Schlüssel

Präfix- B^+ -Baum gegenüber einfachem Präfix- B^+ -Baum:

Rekonstruktion der Separatoren aus den komprimierten Separatoren

⇒ Der einfache Präfix- B^+ -Baum ist der beste Kompromiß zwischen Effizienz und Komplexität der Implementierung.

2.2 Literatur

Bäume (einschließlich B-Bäume) werden in der Regel in *Lehrbüchern über Datenstrukturen* dargestellt, z.B. in:

- [GD 03] Güting R.H., Dieker S.: *'Datenstrukturen und Algorithmen, 2. Auflage'*, Teubner Verlag, 2003.
- [OW 02] Ottmann T., Widmayer P.: *'Algorithmen und Datenstrukturen, 4. Auflage'*, Spektrum Akademischer Verlag, 2002.
- [Wir 96] Wirth N.: *'Algorithmen und Datenstrukturen mit Modula-2'*, Teubner Verlag, 1996.

B- und B^{*}-Bäume wurden auch bereits vorgestellt in:

- [Knu 99] Knuth D.E.: *'The Art of Computer Programming, Vol. 3'*, Addison Wesley Longman, 1999.

Originalartikel:

- [AVL 62] Adel'son-Vel'skii G.M., Landis E.M., Soviet Math, Vol. 3, pp. 1259-63.
- [BM 72] Bayer R., McCreight E.M.: *'Organization and Maintenance of Large Ordered Indexes'*, Acta Informatica, Vol. 1, No. 3, 1972, pp. 173-189.
- [BU 77] Bayer R., Unterauer K.: *'Prefix B-Trees'*, ACM Transactions on Database Systems, Vol. 2, No. 1, 1977, pp. 11-28.

3 Baumstrukturen zur Sekundärschlüsselsuche

Ziel: Unterstützung von Anfragen über mehrere Attribute
(*Sekundärschlüsselsuche = Multiattributssuche*).

3.1 Invertierte Listen

In fast allen kommerziell vertriebenen Datenbanksystemen:

Multiattributssuche mit Hilfe *invertierter Listen*.

- **Primärindex**
Index über den Primärschlüssel.
- **Sekundärindex**
Index über ein Attribut, das kein Primärschlüssel ist.

Im Gegensatz zu einem Primärindex beeinflusst der Sekundärindex den Ort der Speicherung eines Datensatzes nicht. Es werden nur Verweise gespeichert.

Konzept der invertierten Listen:

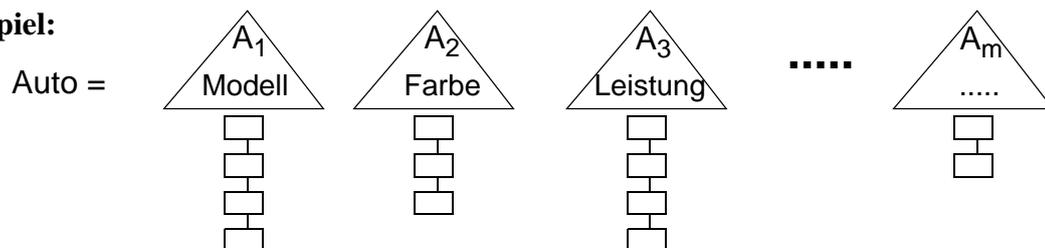
- Für anfragerrelevante Attribute werden Sekundärindizes (*invertierte Listen*) angelegt.
⇒ Damit steht für jedes relevante Attribut eine eindimensionale Indexstruktur zur Verfügung.

Multiattributssuche für invertierte Listen:

Eine Anfrage spezifiziere die Attribute A_1, \dots, A_m :

- m Anfragen über m Indexstrukturen
Ergebnis:
m Listen mit Verweisen auf die entsprechenden Antwortkandidaten in der Datei.
- *Mengentheoretische Verknüpfung* (z.B. Durchschnitt) der m Listen gemäß der Anfrage.

Beispiel:



Anfrage: Gesucht sind alle Autos mit Modell = GLD, Farbe = rot und Leistung = 75 PS

Eigenschaften:

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten.
- Die Suche dauert umso länger, je mehr Attribute spezifiziert sind.
Ursache für beide Beobachtungen:
Die Attributswerte eines Datensatzes sind nicht in einer Struktur miteinander verbunden.
- Invertierte Listen sind einigermaßen effizient, wenn die Antwortlisten sehr klein sind.
- Invertierte Listen haben hohe Kosten für Update-Operationen.
- Sekundärindizes beeinflussen die physische Speicherung der Datensätze nicht.
⇒ Ordnungserhaltung über den Sekundärschlüssel nicht möglich.
⇒ schlechtes Leistungsverhalten von invertierten Listen.

3.2 Hierarchie von B-Bäumen: MDB-Bäume

Ziel: Speicherung multidimensionaler Schlüssel (a_k, \dots, a_1) in einer Indexstruktur.

Idee: Hierarchie von Bäumen, wobei jede Hierarchiestufe jeweils einem Attribut entspricht.

Notation:

Im folgenden werden für eine einfachere Notation die Attributswerte multidimensionaler Schlüssel absteigend numeriert (a_k, \dots, a_1) .

Außerdem ändert sich die Numerierung der Höhen: Blattknoten eines Baumes haben die Höhe 1 und die Höhe der Wurzel entspricht der Höhe des Baumes.

Multidimensionale B-Bäume (MDB-Bäume) [SO 82]

- k-stufige Hierarchie von B-Bäumen.
- Jede Hierarchiestufe (*Level*) entspricht einem Attribut. Werte des Attributs a_i werden in Level i ($1 \leq i \leq k$) gespeichert.
- Die B-Bäume des Levels i haben die *Ordnung* m_i ; m_i hängt von der Länge der Werte des i -ten Attributs ab.

Verzeigerung in einem B-Baum:

- linker Teilbaum bzgl. a_i : $LOSON(a_i)$
- rechter Teilbaum bzgl. a_i : $HISON(a_i)$

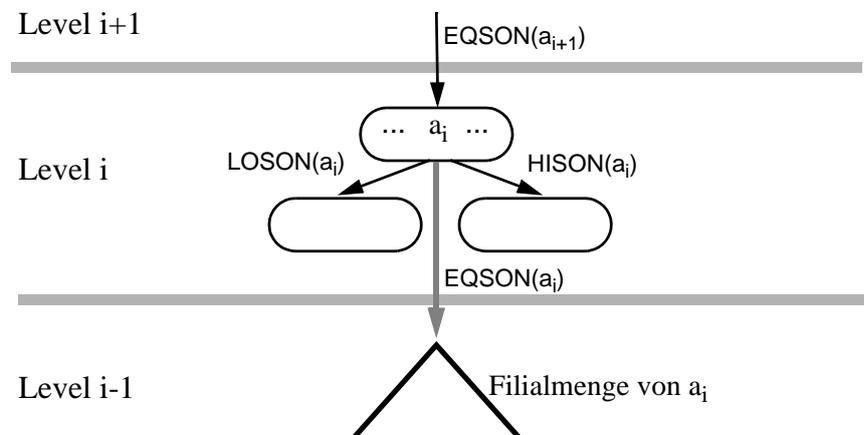
Verknüpfung der Level:

- Jeder Attributswert a_i hat zusätzlich einen *EQSON-Zeiger*: $EQSON(a_i)$ zeigt auf den B-Baum des Levels $i-1$, der die verschiedenen Werte abspeichert, die zu Schlüssel mit Attributswert a_i gehören.

Filialmenge

Für einen *Präfix* (a_k, \dots, a_i) eines Schlüssels (a_k, \dots, a_1) , $i < k$, betrachte man die Menge M aller Schlüssel in der Datei mit diesem Präfix. Die Menge der $(i-1)$ -dimensionalen Schlüssel, die man aus M durch Weglassen des gemeinsamen Präfixes erhält, heißt *Filialmenge* von (a_k, \dots, a_i) (oder kurz: Filialmenge von a_i).

$\Rightarrow EQSON(a_i)$ zeigt auf einen B-Baum, der die Filialmenge von a_i abspeichert.



Exact Match Queries:

können effizient beantwortet werden.

Range, Partial Match und Partial Range Queries:

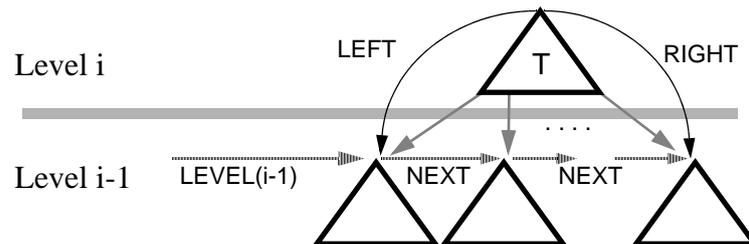
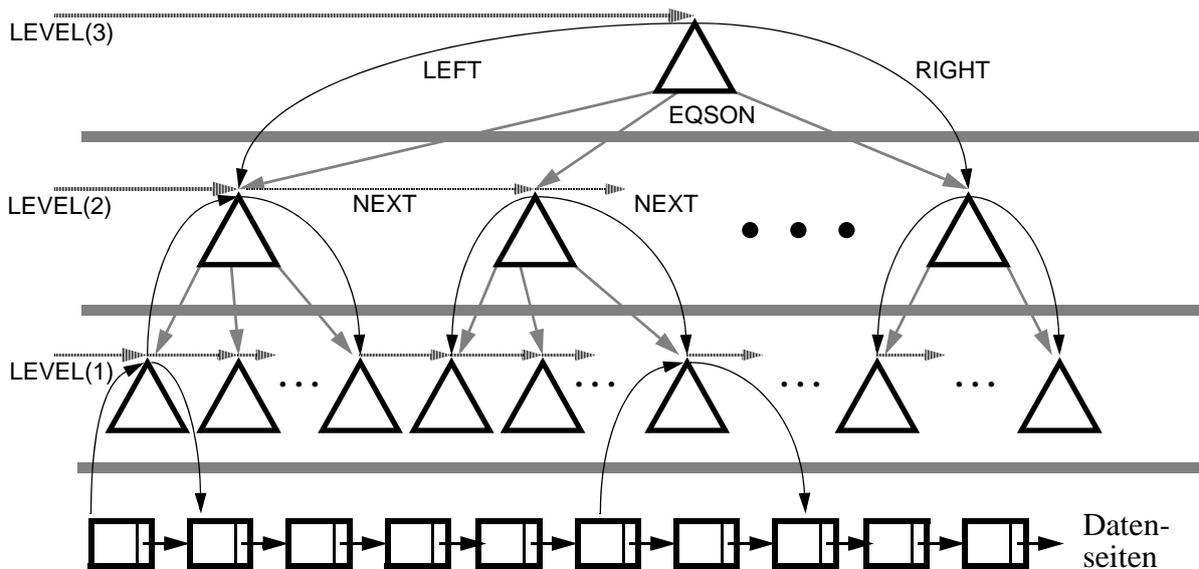
erfordern zusätzlich sequentielle Verarbeitung auf jedem Level.

Unterstützung von Range Queries:

- Verkettung der Wurzeln der B-Bäume eines Levels: *NEXT*-Zeiger.

Unterstützung von Partial Match Queries und Partial Range Queries:

- *Einstiegszeiger für jedes Level:*
 - *LEVEL(i)* zeigt für Level i auf den Beginn der verketteten Liste aus *NEXT*-Zeigern
- *Überspringzeiger* von jeder Level-Wurzel eines Level-Baumes T zum folgenden Level:
 - *LEFT(T)* zeigt zur Filialmenge des kleinsten Schlüssels von T
 - *RIGHT(T)* zeigt zur Filialmenge des größten Schlüssels von T.

**Beispiel: MDB-Baum****Höhenabschätzung**

Annahme: Für jedes Attribut gilt:

- Die Werte sind gleichverteilt im zugehörigen Wertebereich.
- Die Werte eines Attributs sind unabhängig von den Werten anderer Attribute (*stochastische Unabhängigkeit der Attribute*)

⇒ Alle Bäume auf demselben Level haben dieselbe Höhe.

⇒ Die maximale Höhe ist $O(\log N + k)$.

Einwand: Die Annahmen sind in realen Dateien äußerst selten erfüllt.

Die maximale Höhe eines MDB-Baumes, der die obige Annahme nicht erfüllt, beträgt:

$$O(k \cdot \log N)$$

3.3 Balancierung über die Attribute hinweg: kB-Bäume

Ziel:

MDB-Baum, der eine maximale Höhe von $\log N + k$ unabhängig von der Verteilung der Daten garantiert

→ *kB-Bäume* ([GK 80], [Kri 82]).

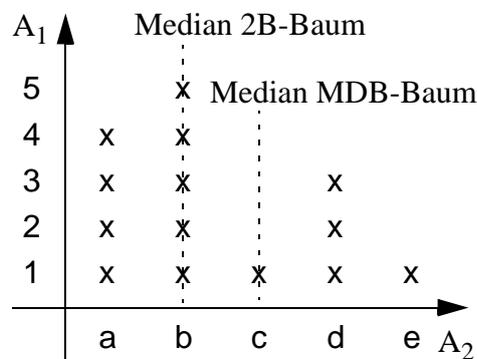
Konzept:

Balancierung über alle Attribute hinweg

⇒ auf höheren Leveln: *verzerrte B-Bäume*

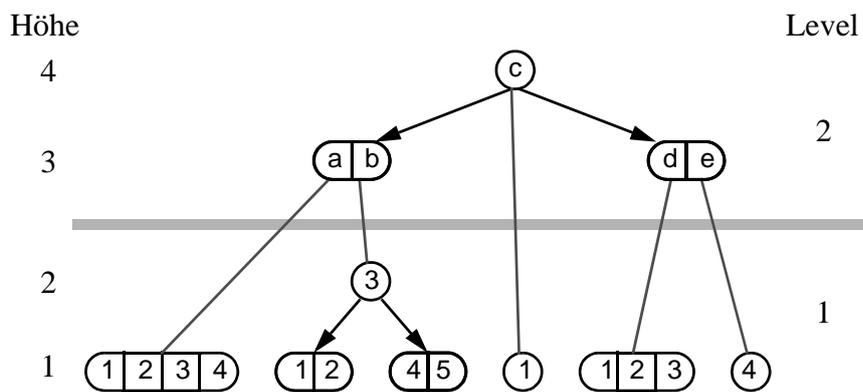
Beispiel:

Gegeben seien folgende zweidimensionalen Datensätze (a_2, a_1) (als Punkte in der Ebene im folgenden Diagramm):

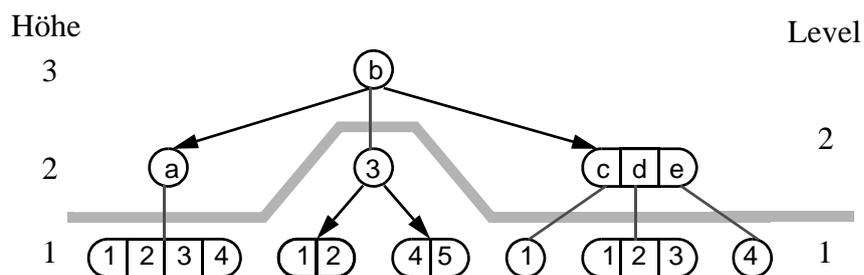


Speicherung dieser Daten:

- 2-Level MDB-Baum der Ordnung 2:



- kB-Baum der Ordnung 2 (k=2):

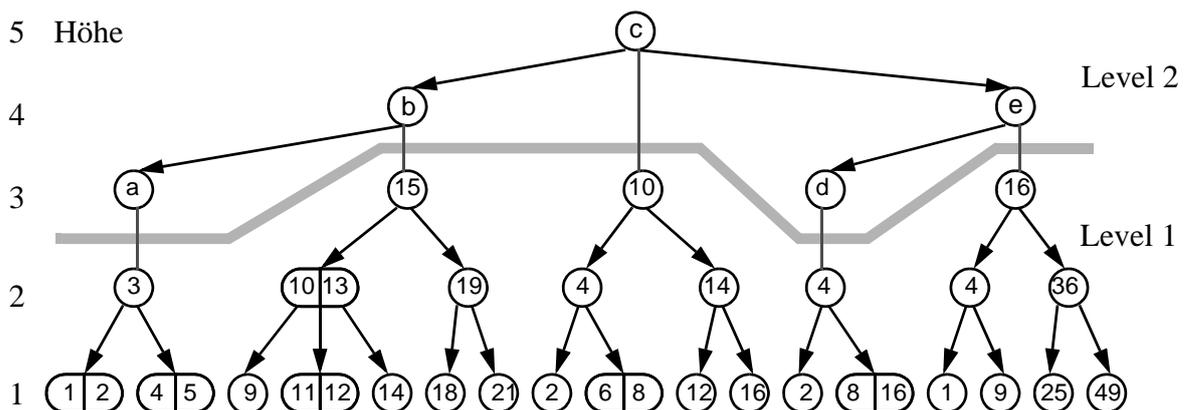


Realisierung des Balancierens

- über Eigenschaften der Attributswerte (im folgenden auch als Schlüssel bezeichnet):
 - Auf dem Level des 1. Attributs haben wir normale B-Bäume.
 - Jeder Attributswert (Schlüssel) eines anderen Levels wird aufgrund der Höhe seines EQSON-Teilbaumes positioniert.
 - Jeder Schlüssel ist entweder *kB-Repräsentant* oder *kB-Separator* (auch kurz *Repräsentant* oder *Separator* genannt):
 - ***kB-Repräsentant***
 - Die Höhe ist gleich der Höhe des EQSON-Teilbaumes plus 1.
 - Ein *kB-Repräsentant* ist *nicht absenkbar*.
 - Repräsentanten haben die niedrigst mögliche Höhe.
 - ***kB-Separator***
 - Die Höhe ist größer als die Höhe des EQSON-Teilbaumes plus 1.
 - Ein *kB-Separator* ist *absenkbar*.
 - Alle Söhne des *kB-Separators* sind auf allen Höhen, auf die er absenkbar ist, mindestens minimal gefüllt, d.h. zu mindestens 50 %.
 - Durch die Eigenschaft, daß jeder Schlüssel entweder Separator oder Repräsentant ist, rechtfertigen die Schlüssel ihre Höhe.

Beispiel: 2B-Baum der Ordnung 1 für die Datensätze (a_2, a_1) :

- $(a,1), (a,2), (a,3), (a,4), (a,5),$
- $(b,9), (b,10), (b,11), (b,12), (b,13), (b,14), (b,15), (b,18), (b,19), (b,21),$
- $(c,2), (c,4), (c,6), (c,8), (c,10), (c,12), (c,14), (c,16),$
- $(d,2), (d,4), (d,8), (d,16),$
- $(e,1), (e,4), (e,9), (e,16), (e,25), (e,36), (e,49)$



- Auf Level 1: *normale B-Bäume.*
- Auf Level 2: *verzerrter B-Baum.*
- Schlüssel a, b, d und e: *kB-Repräsentanten.*
- Schlüssel c: *kB-Separator*
 c hat Höhe 5, ist aber auf Höhe 4 absenkbar. Da seine Söhne auf Höhe 4 (LOSON und HISON) minimal gefüllt sind (mit jeweils 1 Schlüssel), kann c seine Höhe rechtfertigen, obwohl der Schlüssel nicht seine niedrigst mögliche Höhe hat.

Struktureigenschaften eines kB-Baumes der Ordnung m:

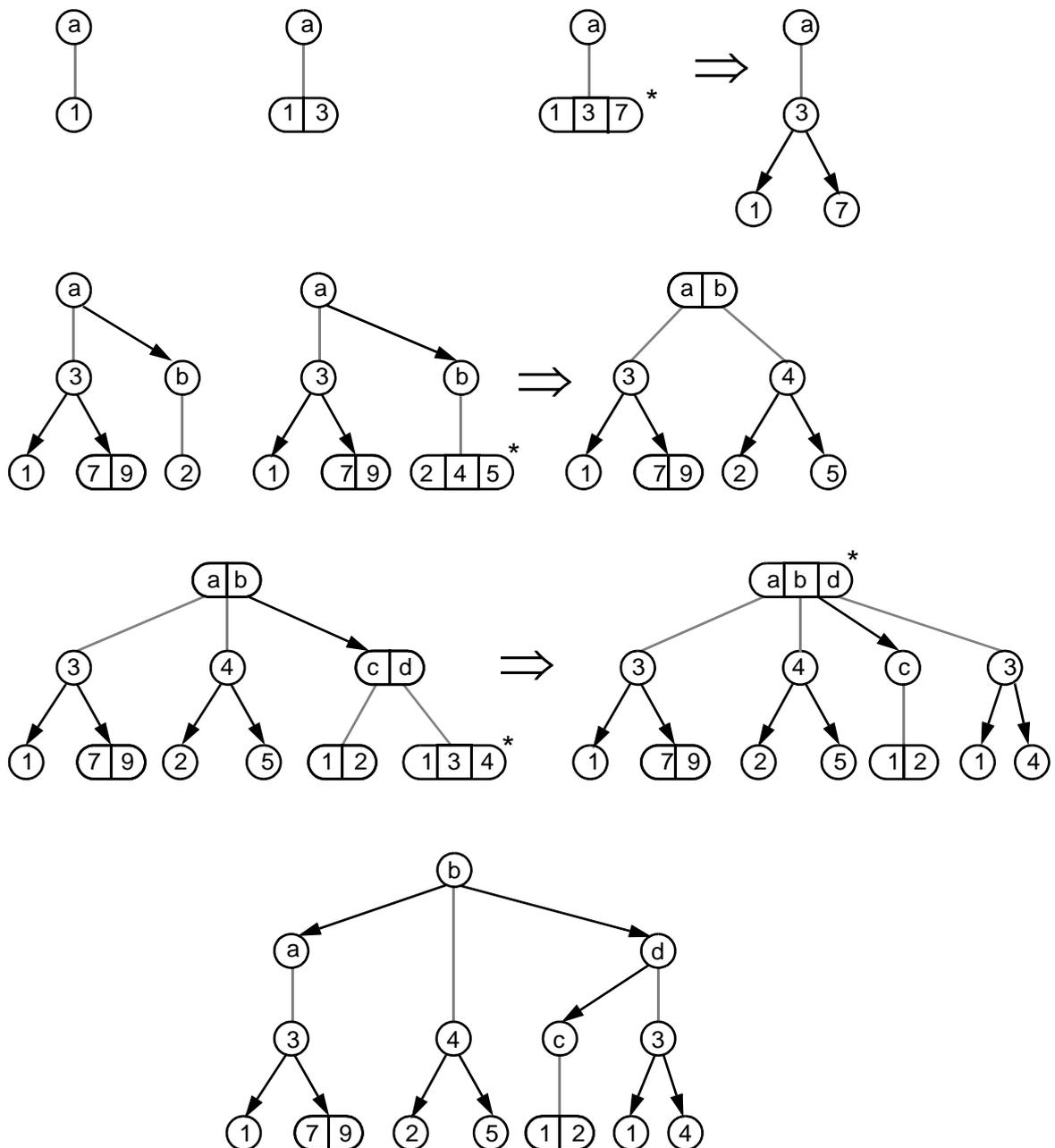
- (i) Jeder Knoten enthält mindestens 1 Schlüssel und höchstens 2m Schlüssel desselben Attributs.
 - (ii) Jeder Schlüssel ist entweder kB-Separator oder kB-Repräsentant.
- (ii) \Rightarrow jeder Schlüssel ist so niedrig wie möglich.

Beispiel für einen 2B-Baum der Ordnung 1

Datensätze (in dieser Reihenfolge eingefügt):

(a,1), (a,3), (a,7), (a,9), (b,2), (b,4), (b,5), (c,1), (c,2), (d,1), (d,3), (d,4).

Einfügeprozeß:



Eigenschaften

- *kB-Bäume verallgemeinern B-Bäume auf den multidimensionalen Fall*

Für jeden kB-Separator ist die Höhe seines LOSON-Teilbaumes gleich der Höhe seines HISON-Teilbaumes. Diese Eigenschaft des kB-Baumes ist eine Verallgemeinerung des Balancierens in B-Bäumen auf den multidimensionalen Fall, so daß alle Blätter den gleichen Abstand von der Wurzel haben.

- *kB-Bäume mit $k = 1$ sind normale B-Bäume.*

Die Struktureigenschaften (i) und (ii) werden bei $k=1$ zu den üblichen B-Baum-Bedingungen:

- Alle Schlüssel in den Blättern sind nicht absenkbar und daher kB-Repräsentanten.
- Nach (ii) sind alle anderen Schlüssel kB-Separatoren. Da jeder Schlüssel auf Höhe 1 absenkbar ist, enthält jeder Knoten mit Ausnahme der Wurzel mindestens m Schlüssel. Die Wurzel enthält mindestens einen Schlüssel.
- Nach (i) enthält jeder Knoten höchstens $2m$ Schlüssel.
- Für jeden kB-Separator ist die Höhe seines LOSON-Teilbaumes gleich der Höhe seines HISON-Teilbaumes.

Mit diesen Eigenschaften ist der 1B-Baum ein B-Baum.

- *Höhe von kB-Bäumen*

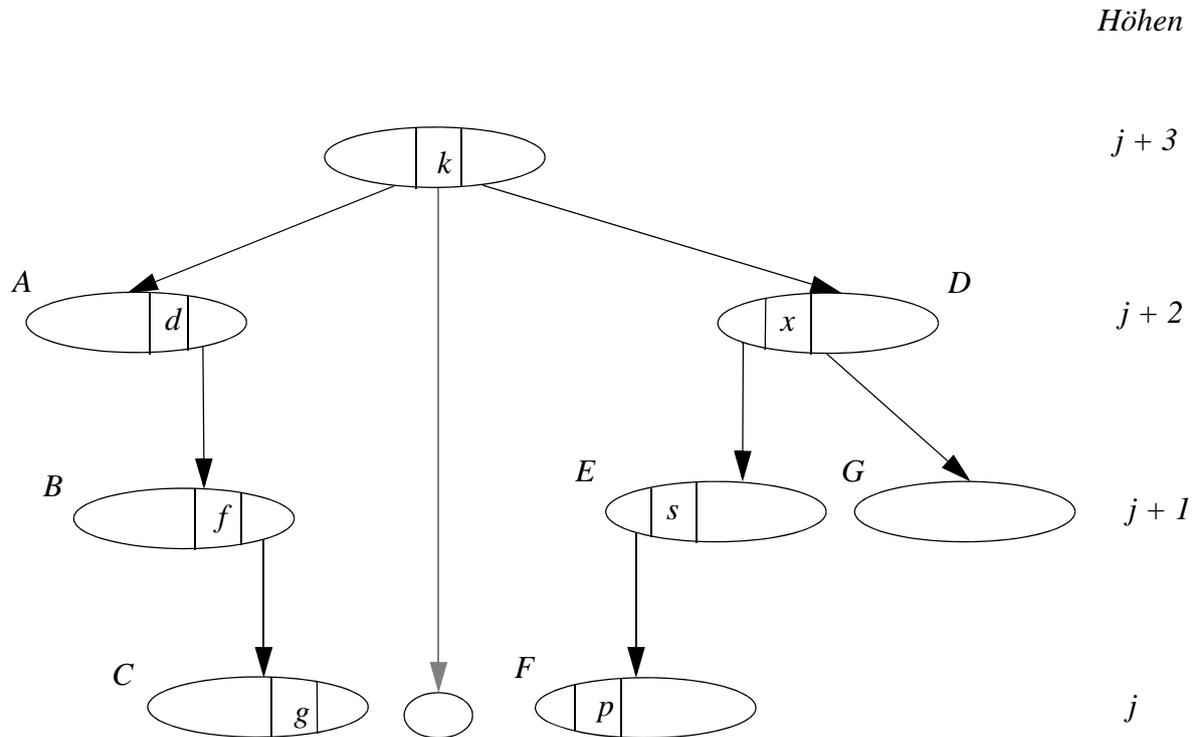
Die Höhe eines kB-Baumes der Ordnung m mit N Datensätzen ist begrenzt durch:

$$h \leq \log_{m+1}(N) + k.$$

Weitere Bezeichnungen

- Ein Knoten der Höhe h und des Levels i heißt (h,i) -Knoten.
- Ein Knoten heißt *Wurzel eines Levels*, wenn er keinen Vaterknoten auf demselben Level hat.
- Ein (h,i) -Knoten P heißt *Blatt des Levels i* , wenn $h = i$ gilt.
Diese Eigenschaft impliziert, daß P keine Söhne auf Level i hat.

Vaterschlüssel, Söhne und Brüder



- k ist *direkter rechter Vaterschlüssel* von Knoten A und *direkter linker Vaterschlüssel* von Knoten D .
- k ist *indirekter Vaterschlüssel* der Knoten B , C , E und F .
- Der Knoten B hat den *direkten linken Vaterschlüssel* d und den *indirekten rechten Vaterschlüssel* k .
- Die Knoten A , B , C , D , E und F sind *Söhne* von Schlüssel k ;
 A und D sind *direkte Söhne* von k .
- A und D sind *direkte Brüder*,
 B und E sowie C und F sind *indirekte Brüder*.

3.4 Einfügealgorithmus des kB-Baumes

Remark:

The following insertion algorithm assumes a kB-tree of order d .

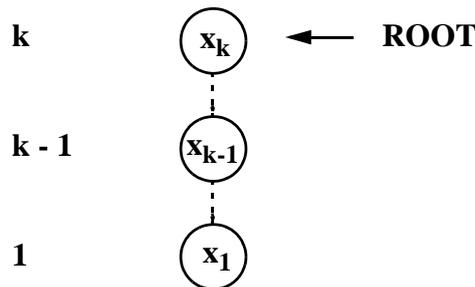
Let $x = (x_k, x_{k-1}, \dots, x_1)$ be the record which should be inserted. We assume that x is not stored in the tree.

Case 1: The tree is empty.

Then we create a node of height $h = k$ containing key x_k , pointing with its EQSON pointer to a node of height $k - 1$ containing key x_{k-1} , etc. The leaf contains key x_1 .

As a result we have the following chain of nodes.

height:



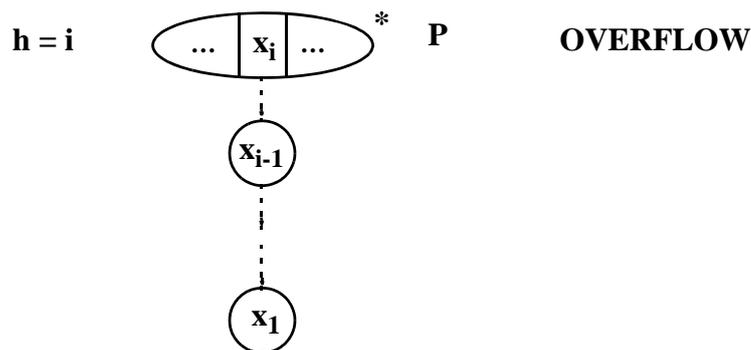
This structure is a correct kB-tree since all keys are representatives.

Case 2: The tree is not empty.

Searching for the keys x_k, x_{k-1}, \dots we traverse the levels $k, k - 1, \dots$. Since record x is not stored in the tree, we will reach some level i which does not contain key x_i . Thus, the search ends in an (h, i) -node P in which x_i falls between two keys.

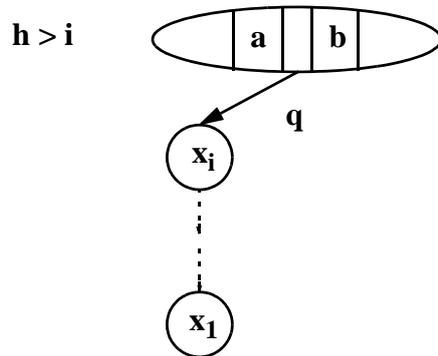
2.1.: P is a leaf of level i , i.e. $h = i$.

We insert key x_i in node P and create a chain of EQSON-nodes below key x_i storing the keys x_{i-1}, \dots, x_1 . As a result we have



Key x_i is a representative. The insertion of key x_i may have created an OVERFLOW in node P , denoted by the star. The restructuring will be described in the next section.

- 2.2.: P is not a leaf of level i, i.e. $h > i$, and the value of the pointer q which we follow from node P, is **nil**.
 We create a chain of nodes storing x_i, \dots, x_1 and let q point to the root of this chain.



Since pointer q pointed to an empty and thus underfilled node, keys a and b are representatives. Thus the modified structure is correct. Since no OVERFLOW can occur, the insertion is finished.

Restructuring Operations

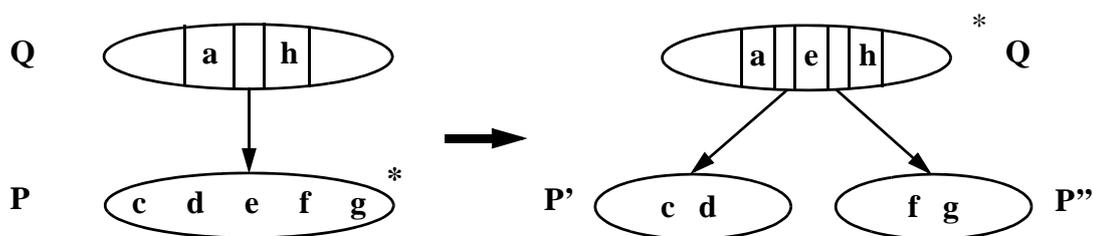
We will generalize the three restructuring operations which we know from B-trees: splitting a node, balancing of two brother nodes and collapsing two brother nodes. Additionally, we need two new operations: lift a key in a certain node up to its father and eliminate the root of a level.

1 SPLIT (P)

This operation is performed if an OVERFLOW occurs in node P. The middle key of the $2d + 1$ keys in node P is pushed up into the father node and splits the overfilled nodes into two fragments of each d keys. Let P be an (h, i) -node.

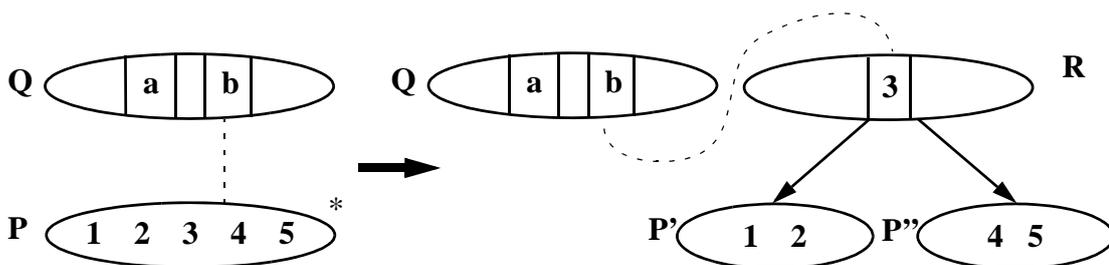
Case 1: The father of P has height $h + 1$.

1.1.: The father Q of P belongs to level i.



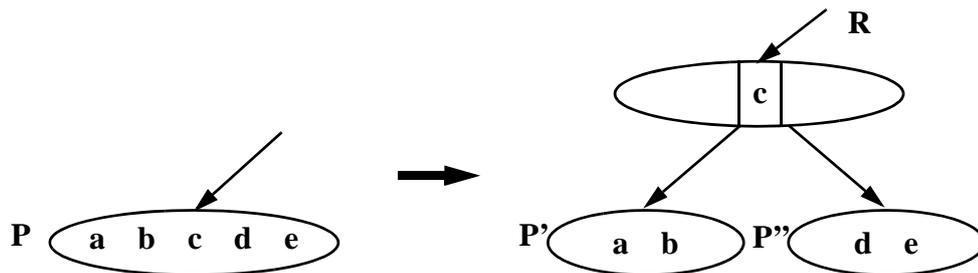
If Q now contains $2d + 1$ keys, then SPLIT (Q).

1.2.: The father Q of P belongs to level $i + 1$.



and lift key b in Q: LIFTKEY (b, Q)

Case 2: The father of P has height $> h + 1$ or does not exist.



An operation is called correct if the result of the operation violates the structure properties (i) - (ii) of a kB-tree only at positions where further operations will be performed. For proving correctness of an operation, the following facts are helpful.

Fact 1: Condition (ii) can only be violated if

- a) an underfilled node is created (separator) or
- b) the root of a level is eliminated (separator or representative) or
- c) the height of the EQSON-subtree increases (representative)

Fact 2: A node may be underfilled without violating the kB-tree structure if its left and right father key both either do not exist or are not sinkable to its height.

We will now verify that SPLIT is correct.

Correctness:

Case 1.1.: Since P is split in the middle, no underfilled node is created. Thus key e is separator on height h. All other keys keep their former separator or representative function.

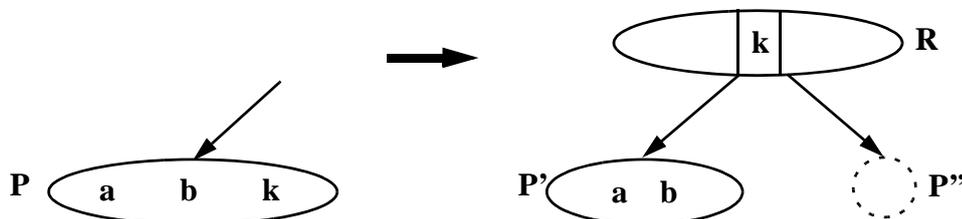
Case 1.2.: Since node R has no father key on the same level, it may be underfilled. Key 3 is separator. All other keys remain in their former separator or representative function. The violation of the representative property of b will be treated by LIFTKEY.

Case 2.: An underfilled node R containing key c is created. Since we had a correct situation, both father keys of R are representatives. Therefore, R may be underfilled. Key c is separator.

2 Liftkey (k, P)

This operation is initiated by case 1.2. in SPLIT. In node P some key k has to be lifted to the father of P. Thus node P is split into two parts of arbitrary size. Let P be an (h, i)-node.

Case 1.: The father of P has height $> h + 1$ or does not exist.

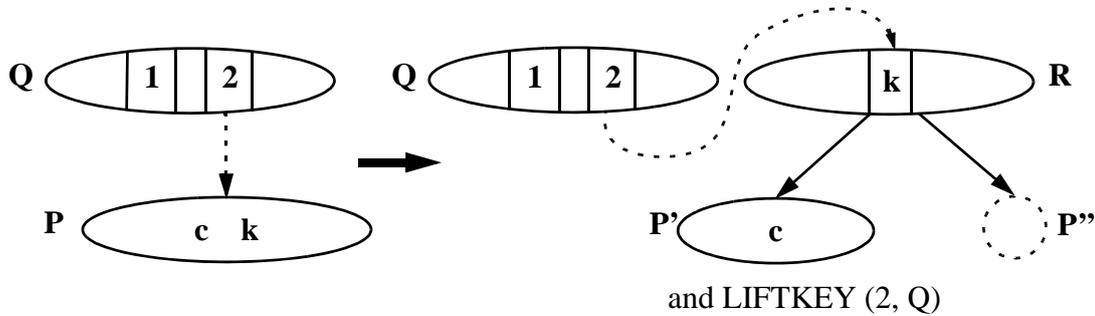


Correctness:

The father key of node R does not exist or is not sinkable. Key k is not sinkable. Thus nodes P', P'' and R may be underfilled.

Case 2.: The father of P has height $h + 1$.

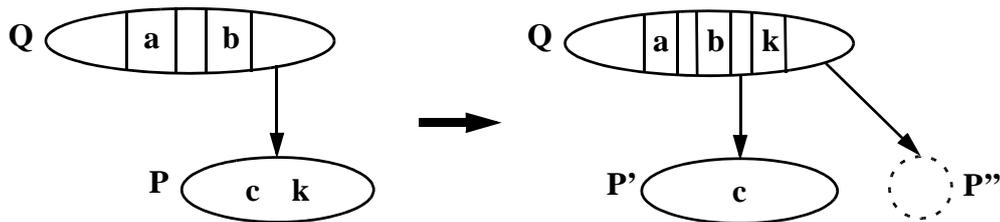
2.1. The father Q of P belongs to level $i + 1$.



Correctness:

Since for node R there is no sinkable father key of the same level and key k is not sinkable, nodes P', P'' and R may be underfilled.

2.2.: The father Q of P belongs to level i .

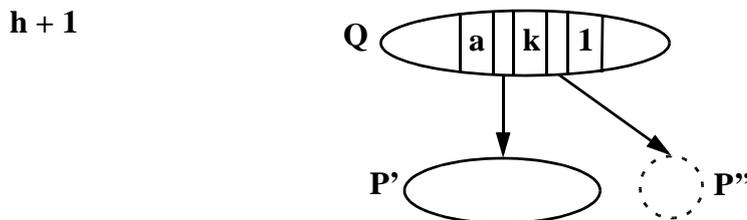


The following violations of the kB-tree structure may have been created by these transformations:

1. Q has too many keys, **OVERFLOW**.
2. P' is underfilled, the left father key of P' may have lost its separator property.
3. P'' is underfilled, the right father key of P'' may have lost its separator property.

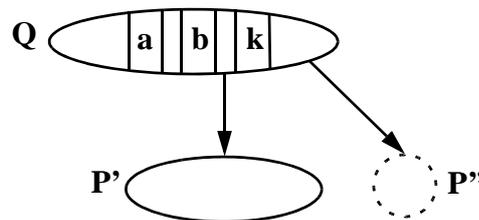
In each case the situation can be corrected such that only one **OVERFLOW** or **UNDERFLOW** remains.

Case a: All father keys of P' and P'' are in Q.

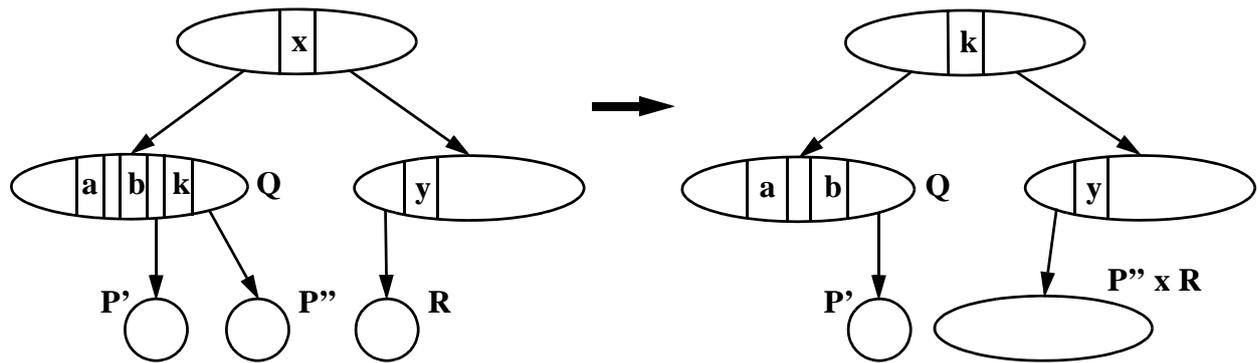


- a.1 Both father keys a and 1 are representatives. UNDERFLOW in P' and P'' does not violate the kB-tree structure. Possible OVERFLOW in Q has to be treated.
- a.2 Exactly one of P' and P'' is underfilled and the corresponding father key is sinkable.
- a.2.1. Balancing with the direct (left in case of P' , right in case of P'') brother is possible. As a result of balancing, the kB-tree conditions are satisfied below height $h + 1$. Possible OVERFLOW in Q has to be treated.
- a.2.2. Collapsing with the direct brother is possible.
As a result of collapsing the kB-tree conditions are satisfied below height $h + 1$. Since the number of keys in Q is the same as before performing LIFTKEY (k, P), the restructuring is finished.
- a.3 Both P' and P'' are underfilled and the left father key of P' as well as the right father key of P'' are sinkable.
- a.3.1. For both P' and P'' balancing is possible.
Possible OVERFLOW in Q has to be treated.
- a.3.2. For one of P' and P'' balancing is possible, for the other one collapsing is possible.
Since the number of keys in Q is the same as before performing LIFTKEY (k, P), the restructuring is finished.
- a.3.3. For both P' and P'' collapsing is possible.
Since the number of keys in Q is one less than before performing LIFTKEY (k, P), possible UNDERFLOW in Q has to be treated.

Case b: Exactly one father key except for key k is in Q . Let this be key b .



- b.1 b and the indirect father key of P'' are representatives. UNDERFLOW in P' and P'' does not violate the kB-tree structure. Possible OVERFLOW in Q has to be treated.
- b.2 Collapsing of P' with the direct left brother is possible.
As a result of collapsing, the number of keys in Q is the same as before performing LIFTKEY (k, P). If the right father key of P'' lost its separator property, balancing or collapsing is performed. In case of collapsing an UNDERFLOW may occur.
- b.3 b is a representative or P' is balanced with the direct brother.
In both cases the number of keys in Q is one more than before performing LIFTKEY (k, P). If balancing with the indirect brother is possible for P'' , an OVERFLOW in Q remains. If balancing is not possible, P'' is collapsed with its indirect brother in the following way:



Since this transformation leaves a correct kB-tree, restructuring is finished.

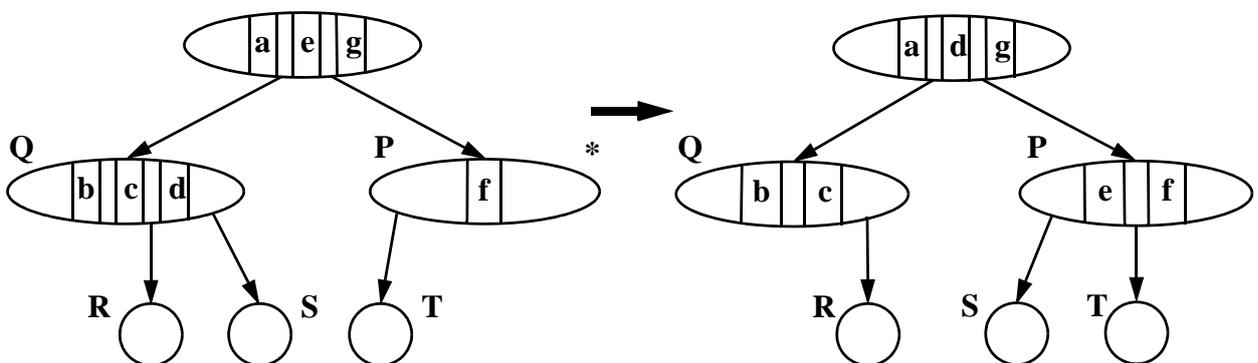
UNDERFLOW treatment

Balancing with a brother

Node P is balanced with its brother Q if P and Q together contain at least $2d$ keys. Otherwise P and Q are collapsed.

a. Balancing with a direct brother

Let P be the underfilled node which causes balancing.

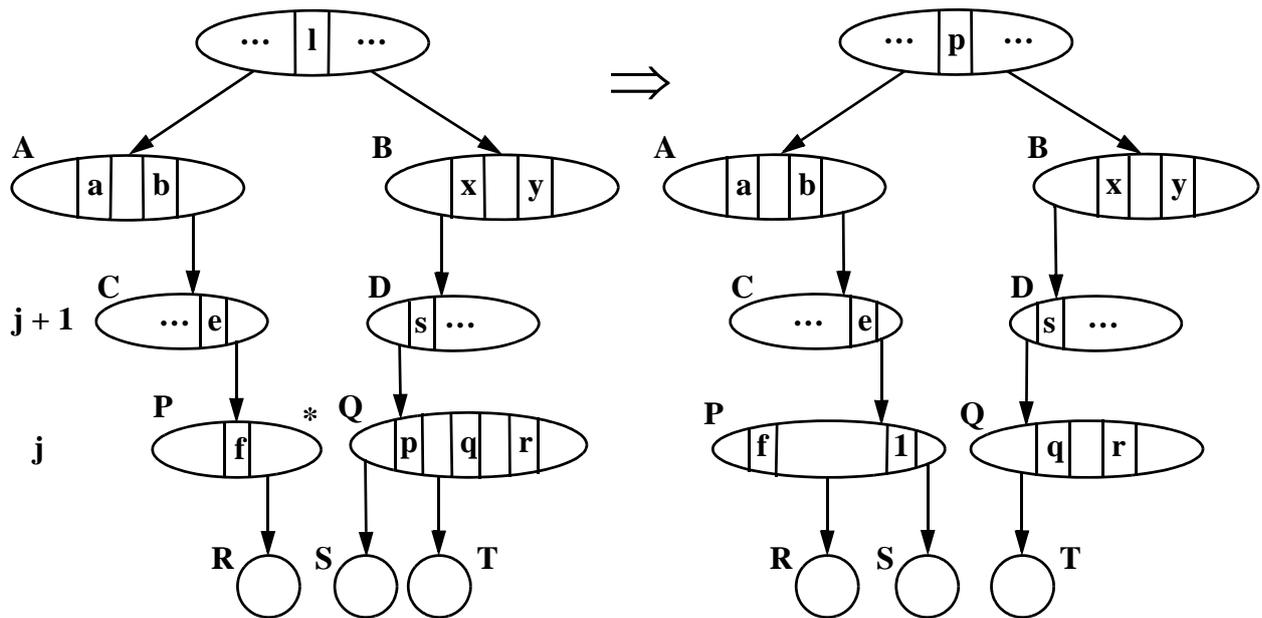


Balancing with a direct brother can be performed if the common father key of the two brothers, in our case key e, is sinkable. If none of the two father keys of P is sinkable, the situation is already correct.

Correctness:

The separator property of key e was violated only on height j. Thus nodes S and T and further indirect sons of key e have at least d keys or e is not sinkable to their height. Therefore, e fullfills condition (ii) in its new position. If the separator property of key g was violated, it is now repaired by filling up node P. In case key d was separator in its old position, d is sinkable to height j - 1. In either case, key d is separator on height j, since Q and P are not underfilled. Thus d is separator.

b. Balancing with an indirect brother:



Balancing with an indirect brother is applicable only if the indirect father key of node P, in our case key l, is sinkable to height j. If the direct father key of P, in our case key e, would be sinkable to height j, it would be better to balance with the direct left brother.

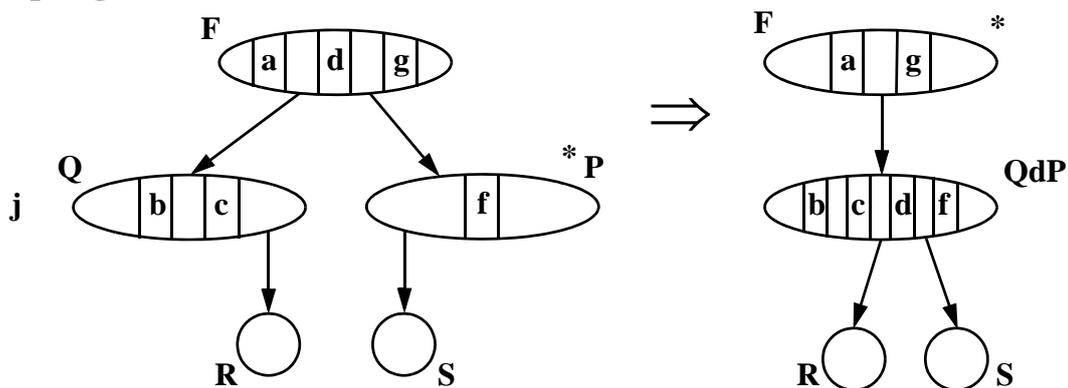
Correctness:

Node A - D are not underfilled, since key l was sinkable to their height. Thus key p is separator on heights > j. Nodes P and Q on height j are not underfilled any more and below height j, key p has the same sons as before. Therefore key p is separator. Key l fulfills condition (ii) since the tree fulfilled the kB-tree properties below height j.

Collapsing with a brother

The underfilled node P is collapsed with its brother Q if P and Q together contain less than 2d keys.

a. Collapsing with a direct brother

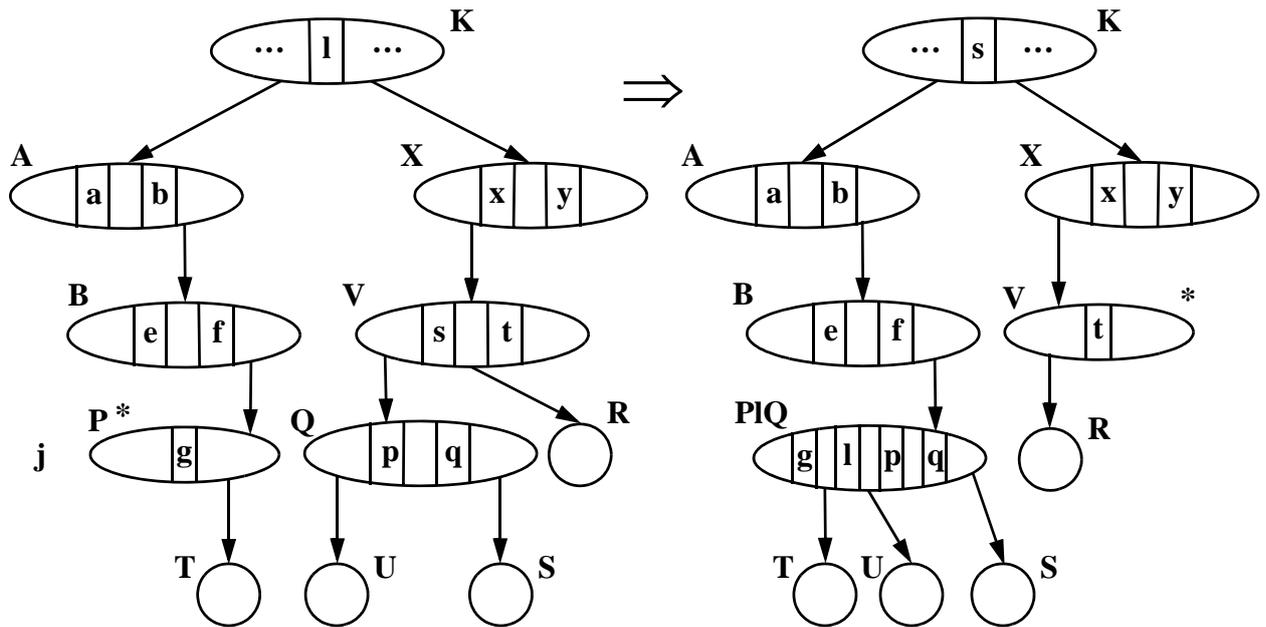


Collapsing with a direct brother is applicable if the common father key d of the nodes Q and P is sinkable to height j. If none of the two father keys of P is sinkable, the situation is already correct. An UNDERFLOW may occur in node F which will be treated.

Correctness:

Key d is separator below height j if it is sinkable, representative otherwise.

b. Collapsing with an indirect brother



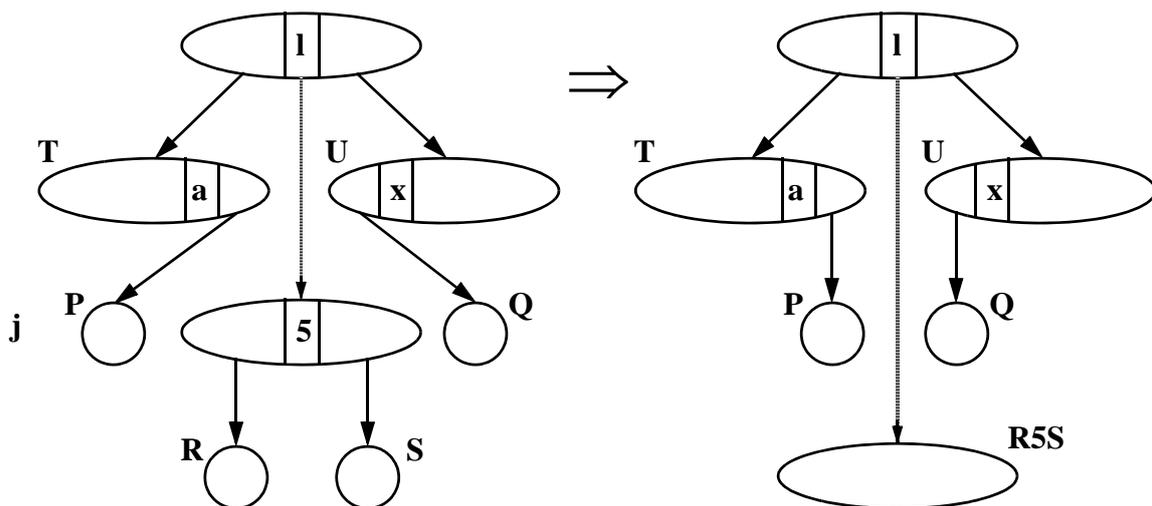
Collapsing with an indirect brother is possible if the indirect father key of node P, in our case key l, is sinkable to height j. An UNDERFLOW may occur in node V which will be treated.

Correctness:

In case key l is sinkable below height j, l is a separator, otherwise a representative. For key s the situation remains unchanged on height j and below. The sons of key s on heights j + 1 and j + 2 are not underfilled with the possible exception of node V which will be treated.

Eliminating the root of a level

The root of a level can be underfilled. If by collapsing its two sons, the last key is deleted from the root, the root is eliminated.



Thus, the father key l of this level is now sinkable to height j. Therefore, we have to check whether its two sons on height j, the nodes P and Q are underfilled. If this is the case, nodes P and Q are balanced or collapsed to a new node PIQ.

Correctness:

In case of collapsing P and Q, the collapsed node PIQ has key a as its left father key and key x as its indirect right father key. Thus a possible UNDERFLOW in U has to be treated. There is one crucial case. If both P and Q are underfilled, the collapsed node PIQ may still be underfilled. Then keys a and x, the two father keys of PIQ, are not sinkable to height j. Thus the underfilling of node PIQ does not violate the kB-tree structure. If exactly one of P and Q is underfilled, PIQ is not underfilled.

Now let us consider once more the insertion algorithm together with the restructuring operations. The restructuring operations are initiated only in case 2.1 of the insertion algorithm where a key is inserted in a leaf P of a level which may create an OVERFLOW in P. This is treated by calling SPLIT (P) which may create again an OVERFLOW (case 1.1 of SPLIT (P)). Thus SPLIT (P) may be called for the root of a level. If the result violates condition (ii), LIFTKEY has to be performed in the next level. As a result, OVERFLOW, UNDERFLOW or FINISH can occur. OVERFLOW is treated as already mentioned. In case of UNDERFLOW, we try to balance or collapse with direct or indirect brothers. If balancing is possible, the restructuring is finished. Collapsing may result in a further UNDERFLOW. If collapsing eliminates the root of a level, UNDERFLOW may occur in the next higher level. OVERFLOW and UNDERFLOW can walk up a path in the tree until finally reaching the root of the tree which may be split or eliminated.

What concerns the insertion time, the only problematic restructuring operation is collapsing with an indirect brother. Collapsing nodes P and Q (see diagram) may result in an UNDERFLOW of V and later A. In order to guarantee that not for each collapsing operation the indirect father key l and the indirect brother is determined again, walking up the search path, we bring the nodes P, B and A on a stack. Walking down from the indirect father key l, we stack the nodes pairwise, i.e. AX, BV and PQ. Thus the nodes of the search path and their brothers are visited at most once which is necessary for an external storage implementation. With this implementation of collapsing, insertion time is $O(\log_{(d+1)} N + k)$ in the worst case.