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 anfragerelevante Attribute werden Sekundärindizes ***(invertierte Listen)*** angelegt.

$\Rightarrow$ Damit steht für jedes relevante Attribut eine eindimensionale Indexstruktur zur Verfügung.

Multiattributssuche für invertierte Listen:

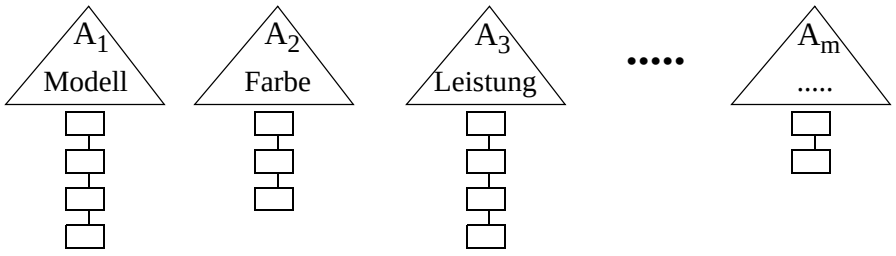
Eine Anfrage spezifiziere die Attribute A_1, ... ,A_m:

- *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:

Auto =



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.

  $\Rightarrow$ Ordnungserhaltung über den Sekundärschlüssel nicht möglich.

  $\Rightarrow$ schlechtes Leistungsverhalten von invertierten Listen.

3.2 Hierarchie von B-Bäumen: MDB-Bäume

Ziel: Speicherung multidimensionaler Schlüssel $(a_k, \ldots, a_1)$ 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 $(a_k, \ldots, a_1)$.

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 $(1 \le i \le k)$ 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)***

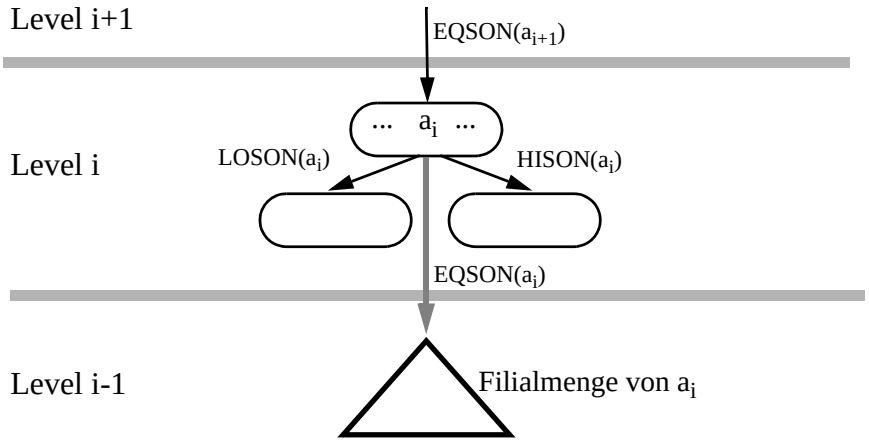
Verknüpfung der Level:

- 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 Schlüsseln mit Attributswert a_i gehören.

  Filialmenge

  Für einen *Präfix* $(a_k, \ldots, a_i)$ eines Schlüssels $(a_k, \ldots, a_1)$, 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 $(a_k, \ldots, a_i)$ (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.

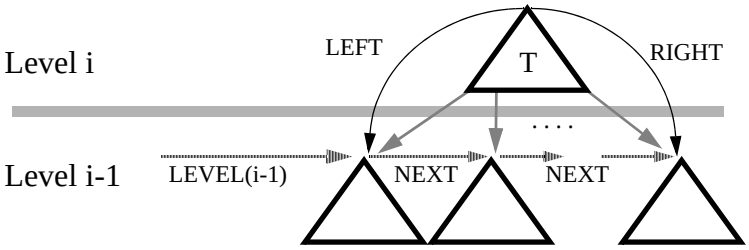
Range, Partial Match und Partial Range Queries:
erfordern zusätzlich sequentielle Verarbeitung auf jedem Level.

Unterstützung von Range Queries:

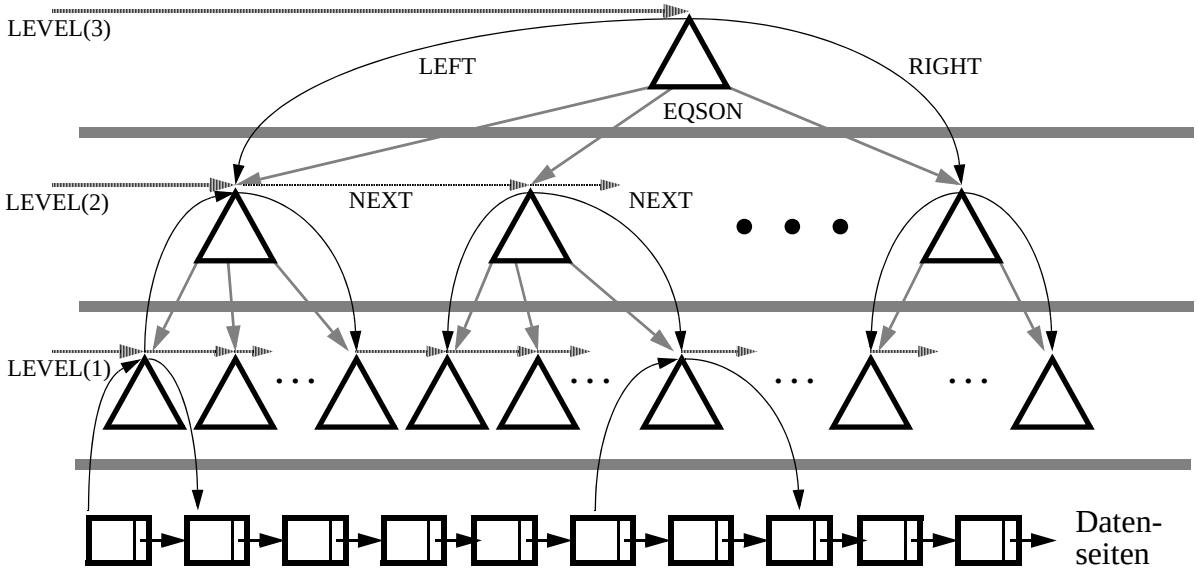
- *Verkettung der Wurzeln* der B-Bäume eines Levels: ***NEXT-Zeiger***.

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)*

$\Rightarrow$ Alle Bäume auf demselben Level haben dieselbe Höhe.

$\Rightarrow$ 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*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]).

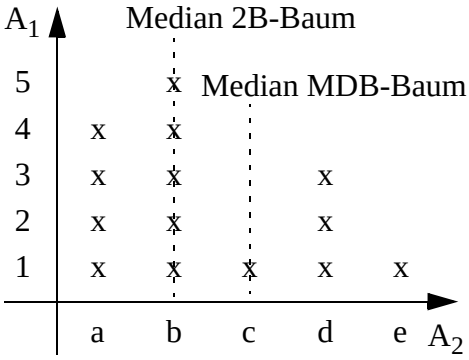
Konzept:

Balancierung über alle Attribute hinweg

⇒ auf höheren Leveln: *verzerrte B-Bäume*

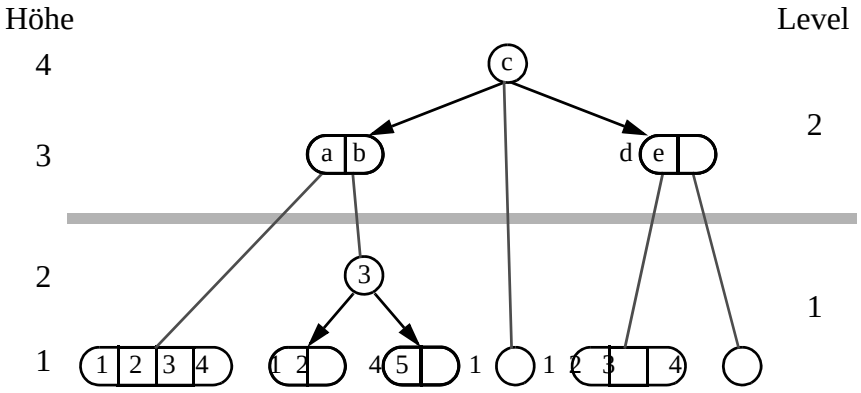
Beispiel:

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

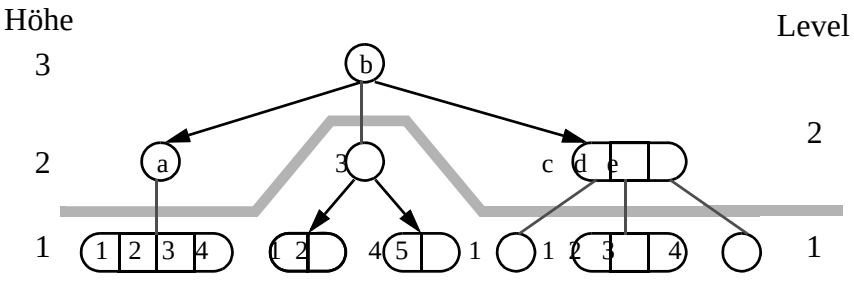


Speicherung dieser Daten:

- 2-Level MDB-Baum der Ordnung 2:



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

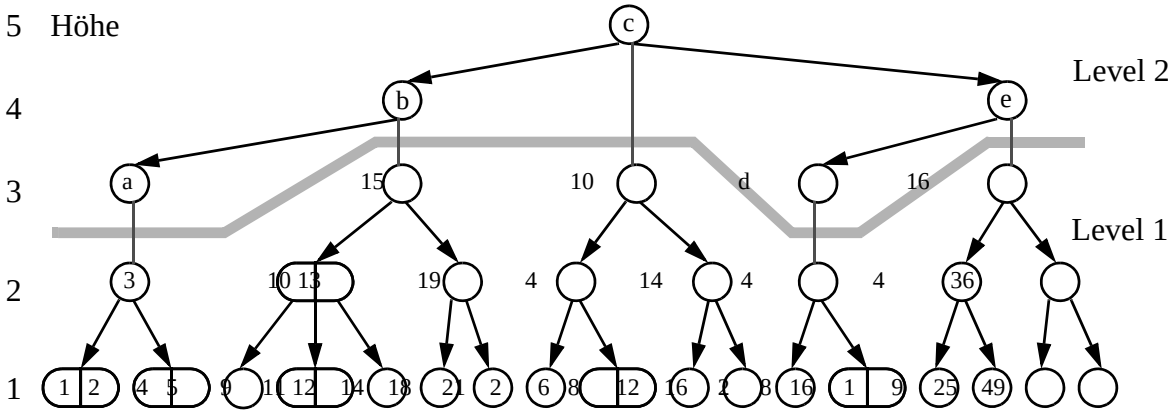


Realisierung des Balancierens

- 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.

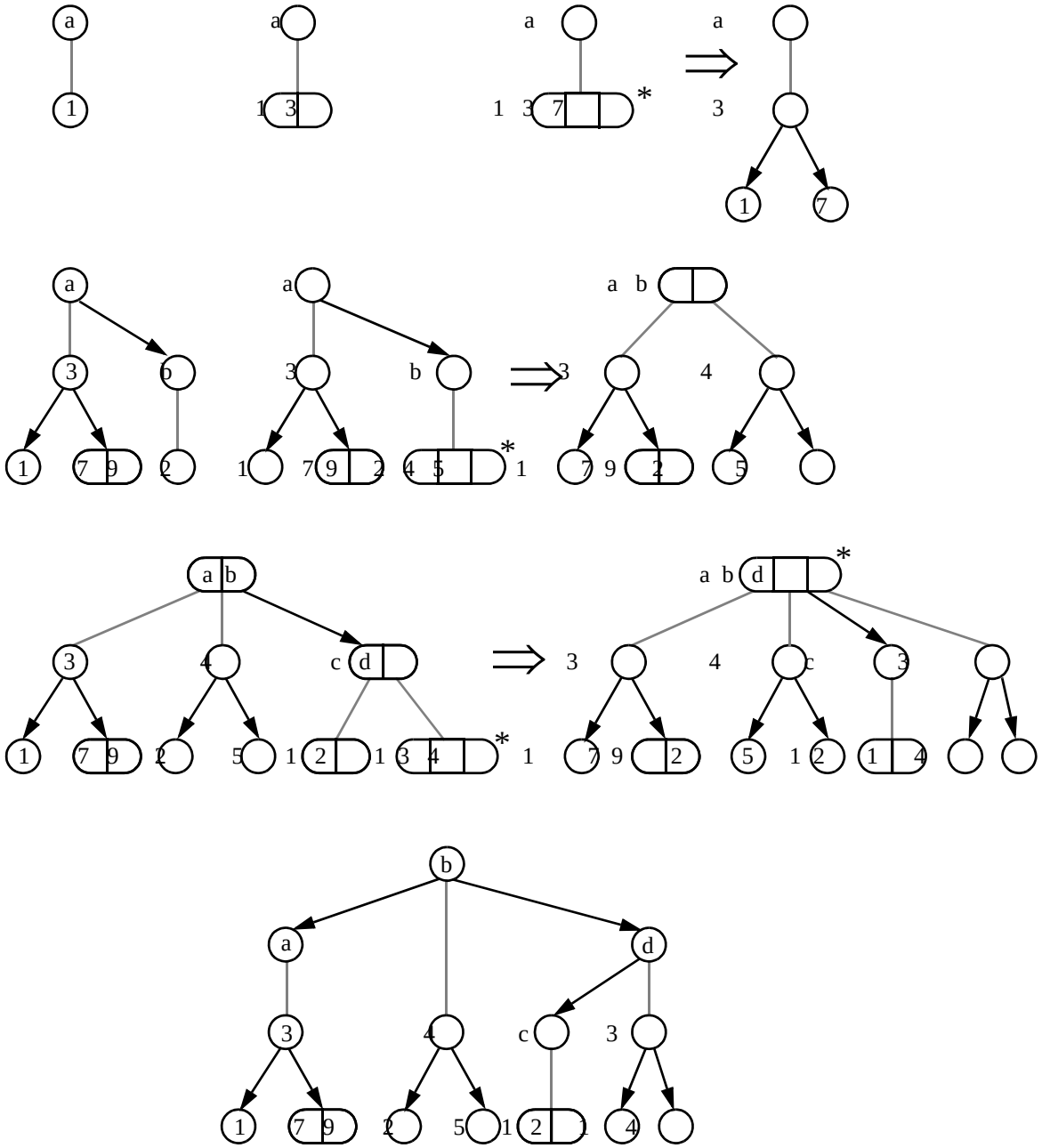
(ii) $\Rightarrow$ jeder Schlüssel ist so niedrig wie möglich.

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*

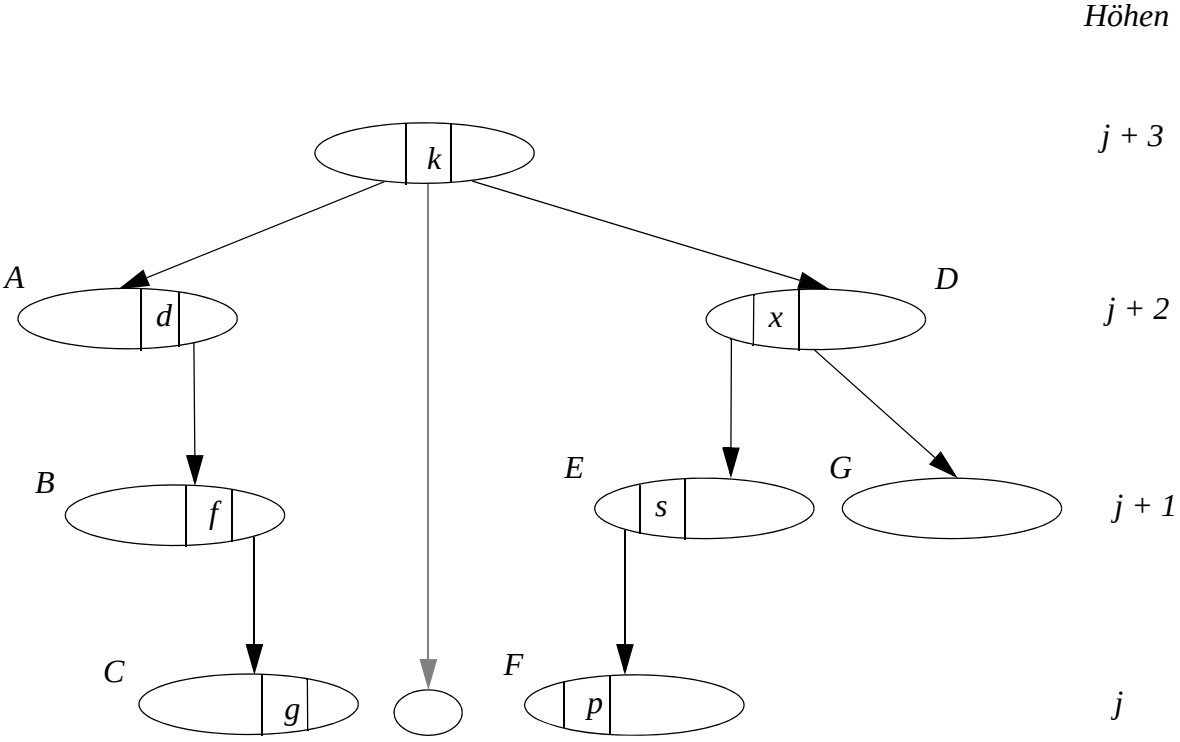
  Die Höhe eines kB-Baumes der Ordnung m mit N Datensätzen ist begrenzt durch:

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

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

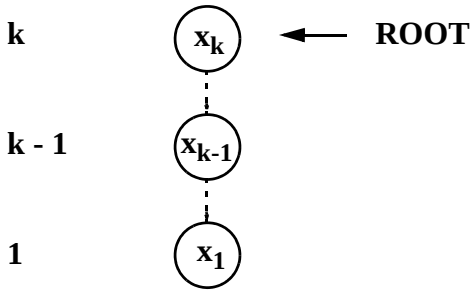
Remark:

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

Let $x = (x_k, x_{k-1}, \ldots, x_1)$ 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.

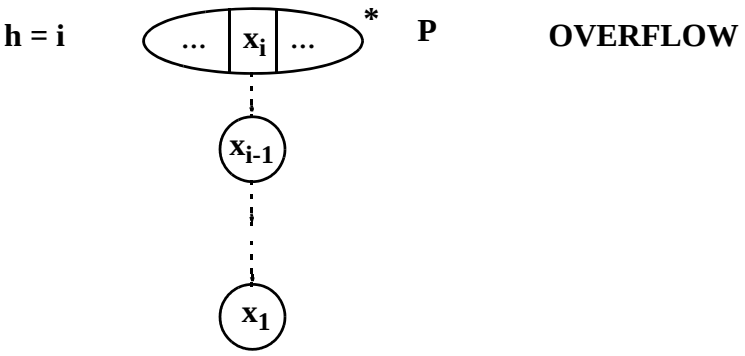
height:



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

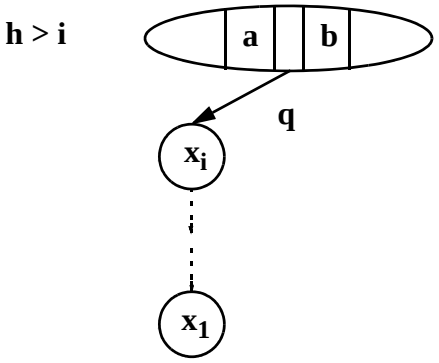
Case 2: The tree is not empty.
Searching for the keys x_k, x_{k-1}, … we traverse the levels k, k - 1, … . Since record x is not stored in the tree, we will reach some level i which does not contain key x_i. Thus, the search ends in an (h, i)-node P in which x_i falls between two keys.

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 $x_{i-1}, \ldots, x_1$. 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, …, 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

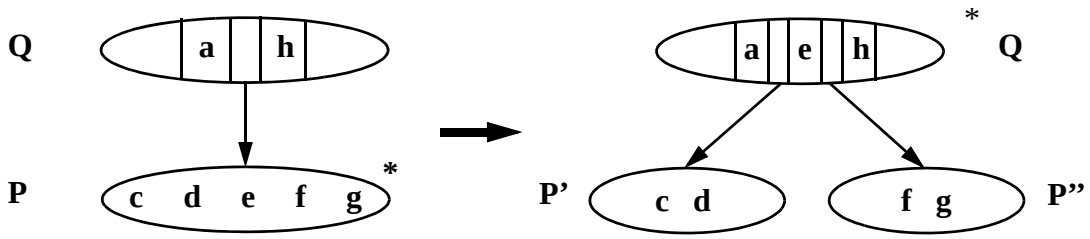
We will generalize the three restructuring operations which we know from B-trees: splitting a node, balancing of two brother nodes and collapsing two brother nodes. Additionally, we need two new operations: lift a key in a certain node up to its father and eliminate the root of a level.

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.

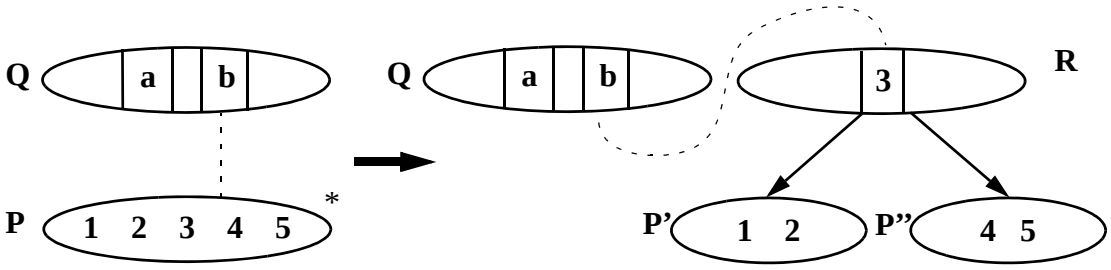
Case 1: The father of P has height h + 1.

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



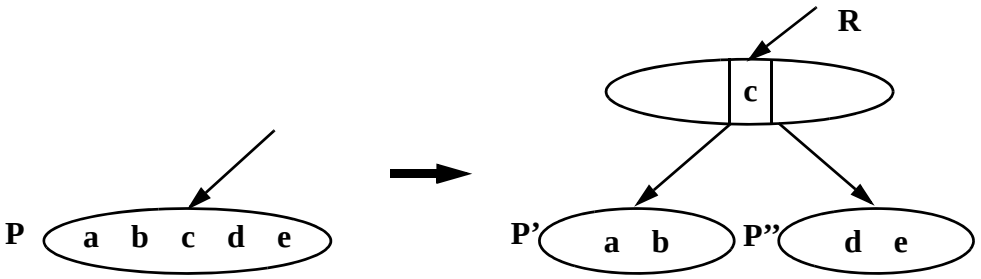
If Q now contains 2d + 1 keys, then SPLIT (Q).

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.

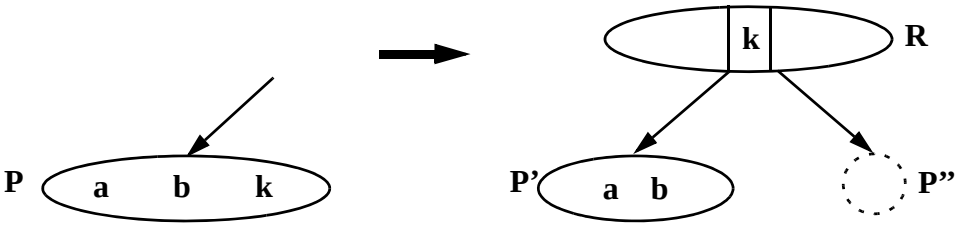
Case 1.2.: Since node R has no father key on the same level, it may be underfilled. Key 3 is separator. All other keys remain in their former separator or representative function. The violation of the representative property of b will be treated by LIFTKEY.

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.

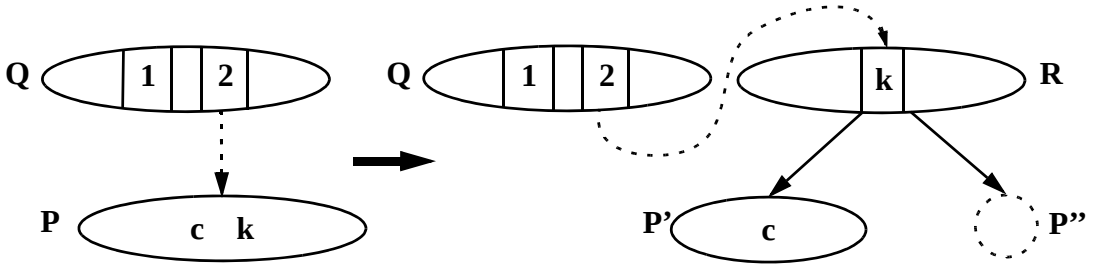


Correctness:

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.

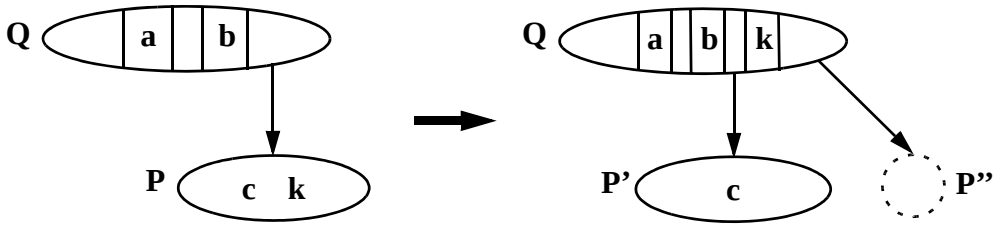


and LIFTKEY (2, Q)

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.



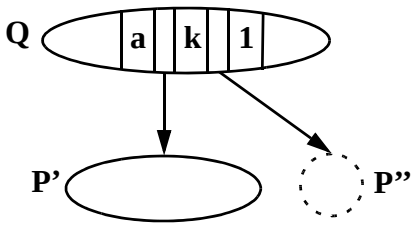
The following violations of the kB-tree structure may have been created by these transformations:

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.

h + 1



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.

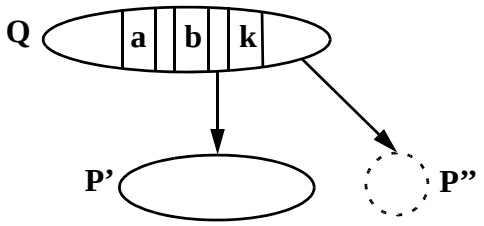
a.3 Both P’ and P’’ are underfilled and the left father key of P’ as well as the right father key of P’’ are sinkable.

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.

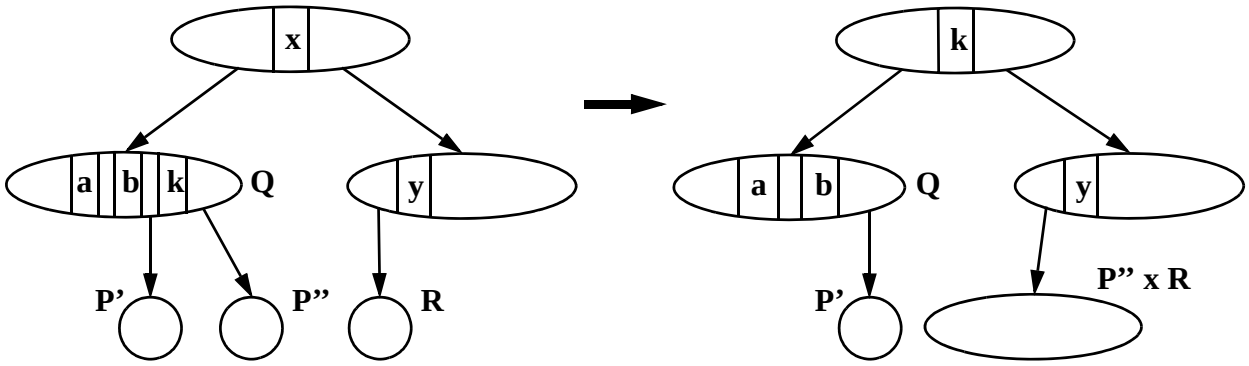
Case b: Exactly one father key except for key k is in Q. Let this be key b.



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.

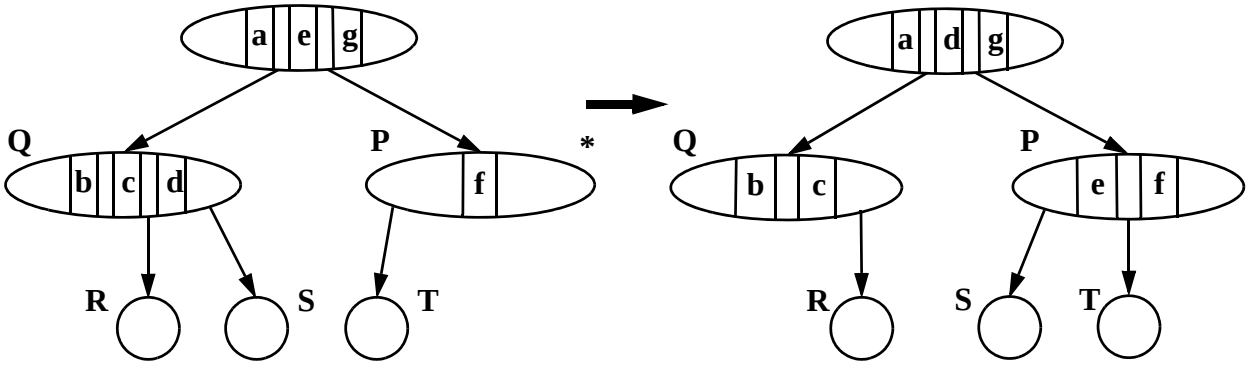
UNDERFLOW treatment

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.

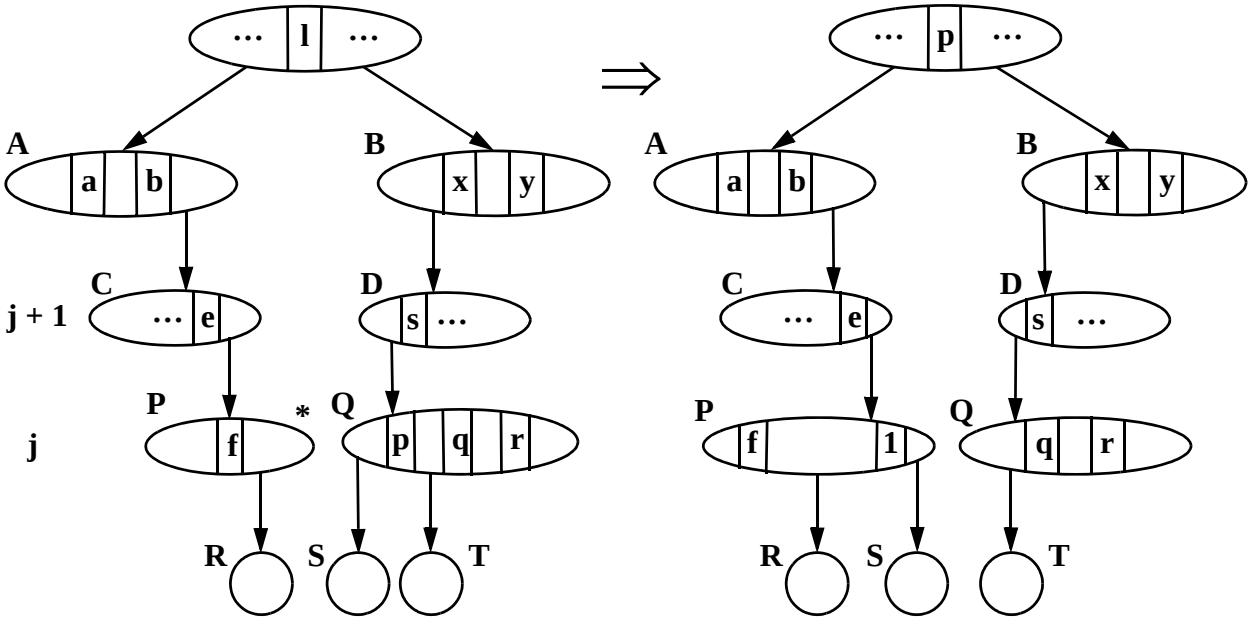


Balancing with a direct brother can be performed if the common father key of the two brothers, in our case key e, is sinkable. If none of the two father keys of P is sinkable, the situation is already correct.

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.

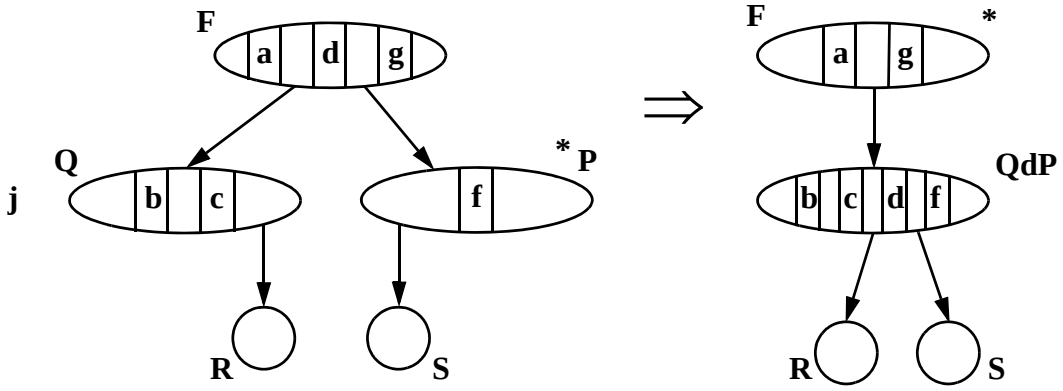
Correctness:

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

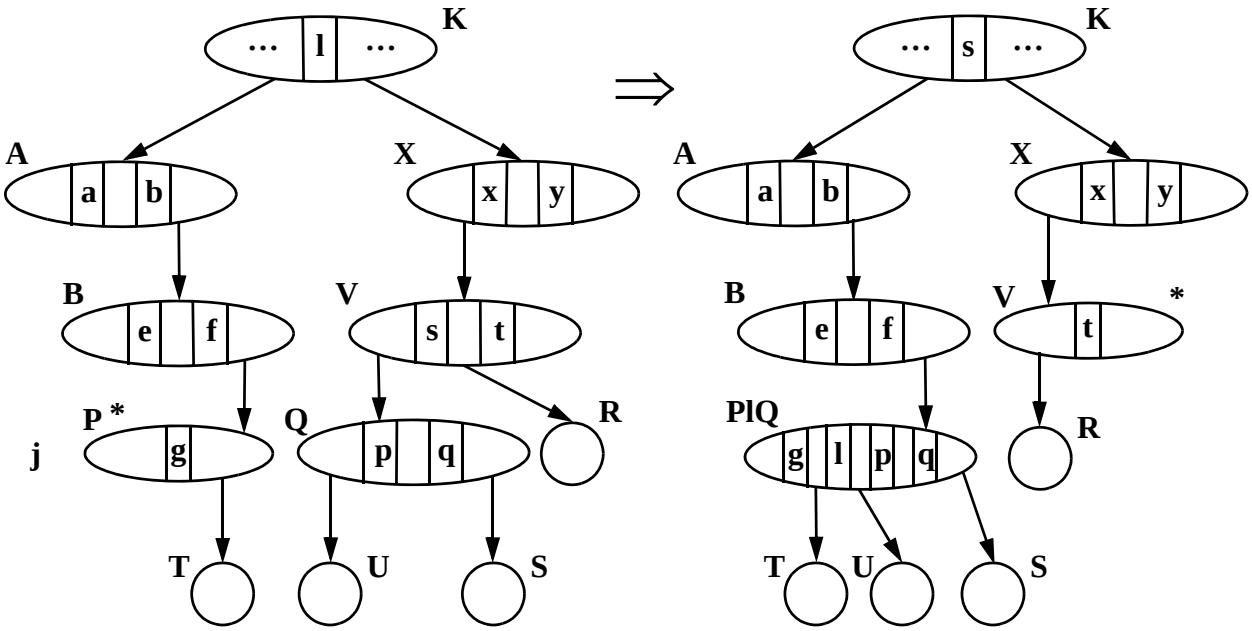


Collapsing with a direct brother is applicable if the common father key d of the nodes Q and P is sinkable to height j. If none of the two father keys of P is sinkable, the situation is already correct. An UNDERFLOW may occur in node F which will be treated.

Correctness:

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

b. Collapsing with an indirect brother



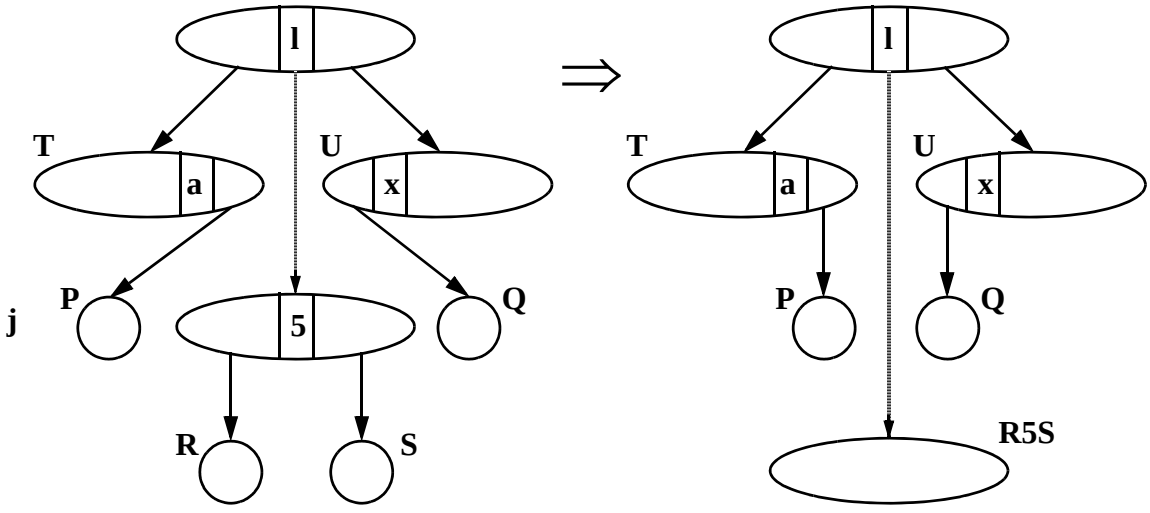
Collapsing with an indirect brother is possible if the indirect father key of node P, in our case key l, is sinkable to height j. An UNDERFLOW may occur in node V which will be treated.

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 PlQ.

Correctness:

In case of collapsing P and Q, the collapsed node PlQ 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 PlQ may still be underfilled. Then keys a and x, the two father keys of PlQ, are not sinkable to height j. Thus the underfilling of node PlQ does not violate the kB-tree structure. If exactly one of P and Q is underfilled, PlQ 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.

3.5 Balancierung gemäß Zugriffshäufigkeit

Bisherige (implizite) Annahme:

- die Zugriffshäufigkeit aller Datensätze ist gleichverteilt.

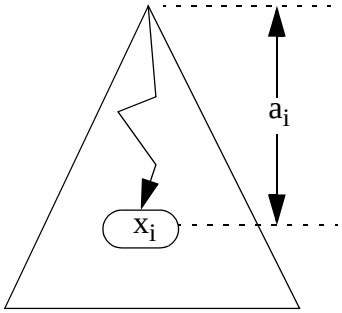
aber:

- Diese Annahme ist oft nicht gegeben,
- d.h. es wird mit unterschiedlicher Häufigkeit auf Datensätze zugegriffen.

Gewichte und Weglängen

Gegeben seien N Datensätze, die durch ihre Primärschlüssel x_1 bis x_N charakterisiert sind.

- ***Gewicht von x_i***
  die Anzahl der Zugriffe zu einem Schlüssel x_i.
  Bezeichung: ω_i.
- ***Gesamtgewicht aller Datensätze***
  die Gesamtanzahl aller Zugriffe.
  Bezeichnung: ω. Es gilt: $\omega = \sum_{i=1}^{N} \omega_i$
- ***relatives Gewicht von x_i***
  die relative Häufigkeit eines Zugriffs zu x_i.
  Bezeichnung: p_i Es gilt: $p_i = \frac{\omega_i}{\omega}$
- ***Weglänge a_i zu einem Schlüssel x_i***
  die Differenz zwischen der Höhe eines Schlüssels x_i und der Höhe der Baumwurzel.
  Die Weglänge entspricht der Anzahl der Seitenzugriffe, die bei einer Suche nach x_i von der Wurzel bis zu dem x_i beherbergenden Knoten anfallen.



- ***gewichtete Weglänge***
  die Summe aller gewichteten Weglängen p_i x a_i
  Bezeichnung: P
  Es gilt: $P = \sum_{i=1}^{N} p_i \times a_i$

Gewichtetes dynamisches Dictionary

Gesucht ist eine Datenstruktur,

- die dynamisch ist,
- die die Schlüssel gemäß ihrem Gewicht abspeichert,
- die die gewichtete Weglänge P möglichst gut minimiert.

Anmerkung:

In den bisherigen Definitionen wurde nur auf die Wahrscheinlichkeit eingegangen, daß man auf einen vorhandenen Schlüssel zugreift. Wichtig wäre es, zusätzlich die Wahrscheinlichkeit zu berücksichtigen, daß man einen Schlüssel nicht findet. Technisch ist dies in den im folgenden vorgestellten Strukturen problemlos möglich. Aus didaktischen Gründen wird jedoch darauf verzichtet.

Optimale Suchbäume

Suchbäume mit minimaler gewichteter Weglänge P heißen *optimale Suchbäume*.

Vorschlag:

Für Primärschlüssel mit Gewichten verwende man optimale binäre Suchbäume oder optimale B-Bäume.

Probleme:

- Die Zugriffshäufigkeiten sind oft nicht von vornherein bekannt oder ändern sich im Laufe der Zeit.
- Der Aufwand zur Bestimmung eines optimalen Suchbaumes ist bereits bei binären Bäumen sehr hoch: $O(n^2)$.

  Einfügungen oder Entfernungen zerstören die Optimalität, die dann mit gleichem Zeitaufwand wieder hergestellt werden muß.

Nahezu-optimale Suchbäume (gewichtete 2B-Bäume)

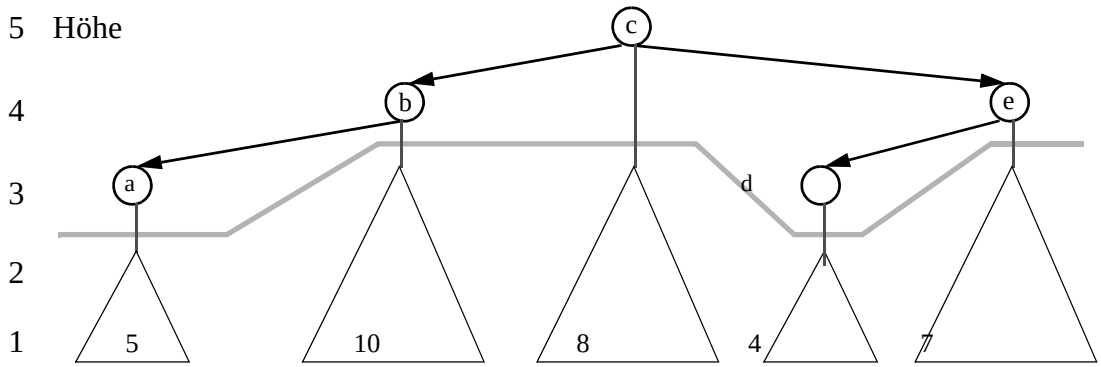
Idee:

- Anstatt optimaler Suchbäume verwenden wir *nahezu-optimale Bäume*, die wesentlich bessere dynamische Eigenschaften besitzen.
- Die Höhe eines Schlüssels im Baum wird durch eine *Heuristik* bestimmt:

  Je höher das Gewicht eines Schlüssels, desto höher ist er im Baum plaziert.

Beispiel:

- Betrachten wir das Beispiel aus Abschnitt 3.3 auf Seite 21.
- Wir können die Anzahl verschiedener Datensätze mit gleichem Wert des ersten Attributs auch als dessen Gewicht interpretieren.
- Wir erhalten dann folgenden Baum:



- Dieser 2B-Baum speichert folgende 5 Datensätze der Form (x_i,ω_i):

  (a,5), (b,10), (c,8), (d,4), (e,7).

Gewichteter 2B-Baum

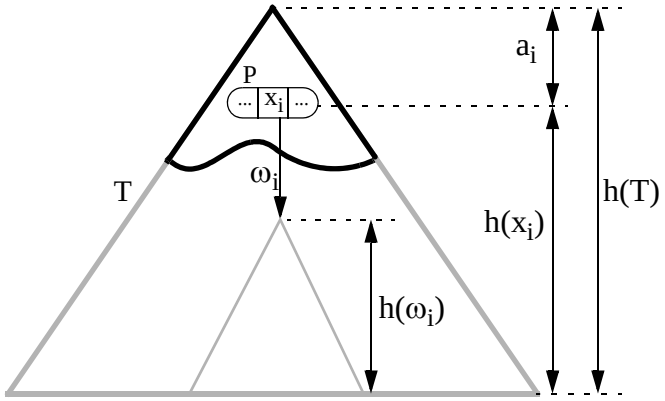
Ein *gewichteter 2B-Baum der Ordnung m* ist ein 2B-Baum mit folgenden Eigenschaften:

- Die Schlüssel werden auf dem oberen Level abgespeichert.
- Auf dem unteren Level werden physisch keine EQSON-Teilbäume mehr abgespeichert:
  - Die EQSON-Teilbäume existieren nur noch *virtuell*.
  - Diese virtuellen Teilbäume werden genutzt, um durch ihre von ω_i abhängige Höhe, den Schlüssel x_i auf der richtigen Höhe zu positionieren.
  - Damit wird jedem Schlüssel x_i statt seines EQSON-Zeigers sein Gewicht ω_i zugeordnet.

Suchzeit

Frage:

Wie groß kann die Weglänge a_i zu einem Schlüssel x_i in einem gewichteten 2B-Baum der Ordnung m maximal werden ?



$a_i \rightarrow MIN \Leftrightarrow h(\omega_i) \rightarrow MAX$

$a_i = h(T) - h(x_i)$

- $h(T) \leq \log_{m+1} \omega + 2$
- $h(x_i) \geq h_{max}(\omega_i) + 1 = (\log_{m+1} \omega_i + 1) + 1 = \log_{m+1} \omega_i + 2$

$\Rightarrow a_i \leq (\log_{m+1} \omega + 2) - (\log_{m+1} \omega_i + 2)$

$$= \log_{m+1}\left(\frac{\omega}{\omega_i}\right) = \log_{m+1}\left(\frac{1}{p_i}\right)$$

Folgerungen:

- In einem gewichteten 2B-Baum der Ordnung m mit einem Gesamtgewicht ω kann die Suche nach Schlüssel x_i mit Gewicht ω_i in

  $$O\left(\log_{m+1}\left(\frac{\omega}{\omega_i}\right)\right) \text{ Zeit}$$

  durchgeführt werden.
- Damit ist der Baum *"nahezu-optimal"*.

Zeit für eine Update-Operation

Update-Operation

Nach einem Zugriff auf einen Schlüssel x_i muß sein Gewicht ω_i um 1 erhöht werden.
Dadurch kann sich seine Position im Baum erhöhen.
Die Operation, die diese Erhöhung vornimmt, heißt UPDATE.

Eigenschaft

Die Zeit für ein UPDATE nach einer Suche ist höchstens proportional zur Suchzeit.

Begründung:

Zum Zeitpunkt t wird nach x_i im Knoten P gesucht.
Das Gewicht von x_i zu diesem Zeitpunkt ist ω_i^t.
Die Suche bewirkt:

$$\omega_i^{t+1} = \omega_i^t + 1$$

Die Höhe des virtuellen Teilbaumes von x_i verhält sich wie die eines normalen B-Baumes, d.h. falls

$$h(x_i) = h_{max}(\omega_i^{t+1}) = \left\lceil \log_{m+1}\left(\frac{\omega_i^{t+1}+1}{2}\right)\right\rceil + 1$$

muß x_i im oberen Level um 1 angehoben werden.

Dieses Anheben erfolgt über eine normale Operation des kB-Baumes:

LIFTKEY (x_i , P)

Im schlimmsten Fall entsteht:

$\Rightarrow$ ein OVERFLOW oder ein UNDERFLOW.

Deren Behandlung läuft im schlimmsten Fall den Suchpfad wieder hoch.

Einfügezeit

Eigenschaft:

In einen gewichteten 2B-Baum der Ordnung m kann ein neuer Schlüssel mit beliebigem Gewicht in

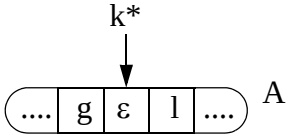
$O(\log_{m+1} \omega)$ Zeit

eingefügt werden.

Begründung:

Das Einfügen eines Schlüssels k* mit einem Gewicht ω läuft wie folgt ab:

- Suche nach der Einfügestelle von k*.
- Wir erreichen einen Knoten A, in dem k* zwischen zwei Schlüssel g und l fällt.
  Es gibt keinen Verweis (gekennzeichnet durch ε) zwischen g und l auf einen Knoten des gleichen Levels.
- $h_{max}(\omega)$ ist die Höhe des virtuellen B-Baumes für das Gewicht ω.



2 Fälle können eintreten:

1. A ist ein Blatt
2. A ist kein Blatt

Fall 1: A ist ein Blatt, d.h. die Höhe h von A ist 2.

Fall 1.1: $h_{max}(\omega) = \left\lceil \log_{m+1}\left(\frac{\omega+1}{2}\right) \right\rceil + 1 = 1$

k* hat die richtige Höhe.

Möglicherweise entsteht durch Einfügen von k* in A ein *Überlauf* (OVERFLOW).

⇒ Behandlung über normale kB-Baum-Algorithmen (bleibt auf Suchpfad beschränkt).

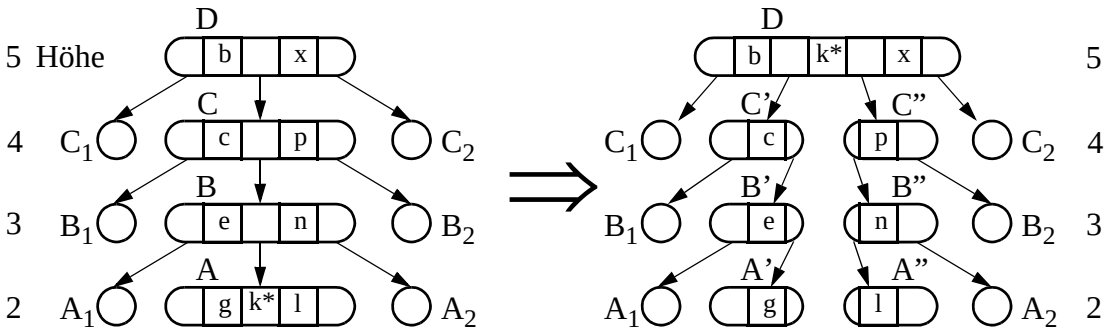
Fall 1.2: $h_{max}(\omega) = \left\lceil \log_{m+1}\left(\frac{\omega+1}{2}\right) \right\rceil + 1 > 1$

k* muß angehoben werden auf einen Knoten der Höhe $h_{max}(\omega)+1$.

Dieses erfolgt durch: $h_{max}(\omega)$-1 LIFTKEY-Operationen.

Beispiel: $h_{max}(\omega) = 4$

⇒ k* muß auf Höhe 5 angehoben werden.



k* befindet sich nun auf seiner korrekten Höhe 5 (wo er kB-Repräsentant ist).

⇒ Die Fragmente A', B', C' und A", B", C" sind entstanden.

⇒ Diese Knoten sind möglicherweise unterfüllt (UNDERFLOW).

⇒ Wenn der Vaterschlüssel, der nicht k* ist (k* ist kB-Repräsentant), dadurch seine Eigenschaft als kB-Separator verliert, wird die Struktur eines gewichteten 2B-Baumes verletzt

UNDERFLOW-Behandlung:

- durch Balancieren mit dem gesunden Bruder oder
- durch Kollabieren mit dem gesunden Bruder.

  Beispiel: Wenn A' unterfüllt ist, wird er mit seinem linken Bruder A_1 balanciert oder kollabiert.

Dieses Restrukturieren wandert einerseits den Pfad A'B'C'D und andererseits den Pfad A"B"C"D hoch, denn:

- Wird ein Knoten mit seinem direkten Bruder kollabiert, kann ein UNDERFLOW im gemeinsamen Vaterknoten auftreten.

- Wird ein Knoten mit einem indirekten Bruder kollabiert, kann ein UNDERFLOW im Bruder des leeren Vaterknotens des Fragmentknotens auftreten.

Beide Fälle werden korrekt behandelt, da die weitere UNDERFLOW-Behandlung genau diese beiden Knoten betrachtet.

Das Restrukturieren wandert die Pfade A'B'C' und A"B"C" hoch bis D erreicht wird.

Mögliche Fälle, die auftreten können:

1. Entweder sind C' und C" beide korrekt oder beide werden balanciert.
   ⇒ möglicher OVERFLOW in D (die Anzahl der Schlüssel in D wird um 1 erhöht).
2. Nur einer, C' oder C", wird kollabiert, der andere ist korrekt oder wird balanciert.
   ⇒ FINISH (die Anzahl der Schlüssel in D bleibt unverändert).
3. C' und C" werden beide kollabiert.
   ⇒ möglicher UNDERFLOW in D (die Anzahl der Schlüssel in D wird um 1 verringert).

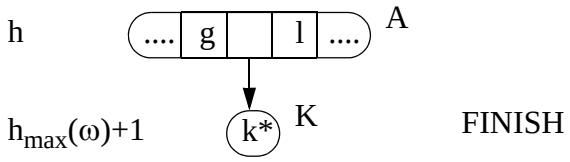
⇒ Das Restrukturieren bleibt auf den Suchpfad beschränkt.

⇒ Das Einfügen in ein Blatt kann in O ($\log_{m+1} \omega$) Zeit erfolgen.

Fall 2: A ist kein Blatt, d.h. h ist größer als 2.

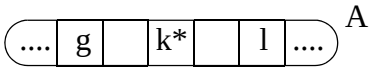
Fall 2.1: $h > h_{max}(\omega) + 1 = \left\lceil \log_{m+1}\left(\frac{\omega + 1}{2}\right) \right\rceil + 2$

Es wird ein neuer Knoten K auf Höhe $h_{max}(\omega)+1$ mit Schlüssel k* angelegt.
Der Zeiger zwischen g und l wird auf diesen neuen Knoten K gerichtet.



Fall 2.2: $h \leq h_{max}(\omega) + 1$

Wir fügen k* in Knoten A an der entsprechenden Position ein:



Weiter geht es wie in Fall 1.

⇒ Der Zeitaufwand für das Einfügen in einen inneren Knoten ist proportional zur Zugriffszeit zu diesem Knoten.

Es gilt:

Wenn ein Schlüssel mit hohem Gewicht zwischen zwei Schlüsseln mit niedrigem Gewicht eingefügt wird, bestimmt die Suchzeit nach den Schlüsseln mit niedrigem Gewicht die Einfügezeit.

Weitere Operationen für gewichtete dynamische Dictionaries

- **PROMOTE**
  erhöht das Gewicht ω_i eines Schlüssels x_i um eine positive Zahl δ auf den Wert $\omega_i+\delta$.

- **DEMOTE**
  erniedrigt das Gewicht ω_i eines Schlüssels x_i um eine positive Zahl δ auf den Wert $\omega_i-\delta$.
  Voraussetzung: $\omega_i > \delta$.

- **SPLIT**
  teilt einen gewichteten 2B-Baum T gemäß eines Schlüssels x_i in 2 gewichtete 2B-Bäume auf:
  - T_1 enthält alle Schlüssel, die kleiner als x_i sind und
  - T_2 enthält alle Schlüssel, die größer als x_i sind.

- **CONCATENATE**
  konstruiert aus zwei gewichteten 2B-Bäumen T_1 und T_2 einen gewichteten 2B-Baum T, der alle Schlüssel aus T_1 und T_2 enthält.
  Voraussetzung: alle Schlüssel aus T_1 sind kleiner als alle Schlüssel aus T_2.

Zeitaufwand

Die folgende Tabelle gibt den Zeitaufwand im schlechtesten Fall in einem gewichteten 2B-Baum für die jeweilige Operation an:

Operation	Zeit proportional zu	Bemerkung
Suchen	$\log\frac{\omega}{\omega_i}$	
Einfügen	$\log\omega$	
Entfernen	$\log\omega$	Einzige Operation in nicht-idealer Zeit (ideal wäre die Suchzeit zum Schlüssel)
PROMOTE	$\log\frac{\omega}{\omega_i}$	Zeit proportional zur alten Suchzeit
DEMOTE	$\log\frac{\omega}{\omega_i-\delta}$	Zeit proportional zur neuen Suchzeit
SPLIT	$\log\frac{\omega}{\omega_i}$	
CONCATENATE	$\log\max(\omega_1, \omega_2)$	

Nahezu-optimale mehrdimensionale Suchbäume (gewichtete (k+1)B-Bäume)

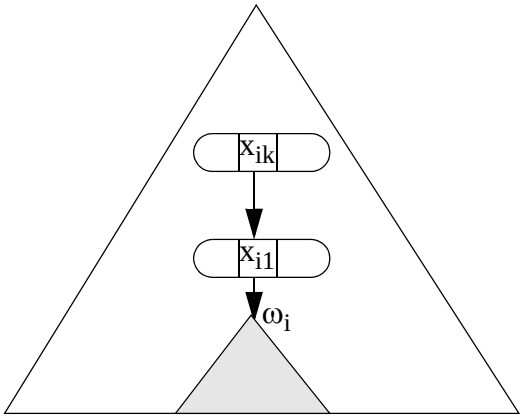
Idee:

Jedem k-dimensionalen Schlüssel (x_{i1}, ... , x_{ik}) wird ein Gewicht ω_i zugeordnet.

Gewichteter (k+1)B-Baum

Ein *gewichteter (k+1)B-Baum der Ordnung m* ist ein (k+1)B-Baum mit folgenden Eigenschaften:

- Die Schlüssel werden auf den oberen k Leveln abgespeichert.
- Auf dem unteren Level werden nur noch virtuelle Teilbäume abgespeichert, deren Höhe vom Gewicht ω_i des Schlüssels (x_{i1}, ... , x_{ik}) abhängt:



Eigenschaften:

- *Exact Match Queries* nach (x_{i1},...,x_{ik}) können in $O(\log_{m+1} \omega/\omega_i + k)$ Zeit durchgeführt werden
- Der Baum ist immer *nahezu optimal*.
- In einem gewichteten (k+1)B-Baum können alle Operationen eines gewichteten dynamischen Dictionaries ausgeführt werden.
- Der *Zeitaufwand* für die Operationen erhöht sich gegenüber gewichteten 2B-Bäumen um den additiven Faktor k.

Leistungsverhalten

Einwand:

Gewichtete 2B- und (k+1)B-Bäume sind viel zu komplexe Strukturen und lohnen sich erst bei sehr großen Dateien.

Leistungsvergleich für die Suchzeit:

B-Baum und gewichteter 2B-Baum:

- Es wurden B- und gewichtete 2B-Bäume implementiert und gegeneinander verglichen.
- Bei Primärschlüsseln mit normal verteilten Zugriffshäufigkeiten sind gewichtete 2B-Bäume bereits ab 750 Datensätzen den einfachen B-Bäumen überlegen.
- Diese Überlegenheit vergrößert sich ständig von diesem niedrigen Amortisationspunkt an.

3.6 Originalliteratur

MDB-Bäume:

[SO 82] Scheuermann P., Ouksel M.: *'Multidimensional B-trees for associative searching in database systems'*, Information Systems, Vol. 7, No. 2, 1982, pp. 123-137.

kB- und kB^+-Bäume:

[GK 80] Güting H., Kriegel H.-P.: *'Multidimensional B-trees: An efficient dynamic file structure for exact match queries'*, GI-Jahrestagung, Saarbrücken, 1980, in: Informatik-Fachberichte, Vol. 33, Springer, 1980, pp. 375-388.

[Kri 82] Kriegel H.-P.: *'Variants of Multidimensional B-trees as Dynamic Index Structures for Associative Retrieval in Database Systems'*, Proc. 8th Conf. on Graphtheoretic Concepts in Computer Science, 1982, pp. 109-128.

gewichtete kB-Bäume:

[GK 81] Güting H., Kriegel H.-P.: *'Dynamic k-dimensional multiway search under time-varying frequencies'*, Proc. 5th GI Conf. on Theoretical Computer Science, Karlsruhe, in: Lecture Notes in Computer Science, Vol. 104, Springer, 1981, pp. 135-145.

4 Raumorganisierende Strukturen zur Primärschlüsselsuche

Die bisher vorgestellten ***Suchbaumverfahren*** sind

- ***datenorganisierende Strukturen***

  d.h. sie organisieren die Menge der tatsächlich auftretenden Daten.

In diesem und dem nächsten Kapitel werden ***dynamische Hashverfahren*** zur Primär- und Sekundärschlüsselsuche vorgestellt. Die dynamischen Hashverfahren sind

- ***raumorganisierende Strukturen***

  d.h. sie organisieren den Raum, in den die Daten eingebettet sind.

4.1 Hashverfahren

Hashverfahren sind eine weit verbreitete Methode zur Organisation von Daten sowohl im Haupt- als auch im Sekundärspeicher.

Prinzip

Gegeben:

- *Wertemenge der Schlüssel*: Domain(K)
- Menge der möglichen Adressen: *Adreßraum* A (im weiteren sei A = [0 ... N-1])

Hashfunktion

Man nehme eine “geeignete” Funktion, die die Schlüssel auf den Adreßraum abbildet:

$h : Domain(K) \rightarrow A,$

Diese Funktion h heißt ***Hashfunktion***.

Verteilung der Schlüssel über den Adreßraum

Ziel:

gleichmäßige Verteilung der Schlüssel über den Adreßraum

Probleme:

- Die Schlüssel sind nicht gleichmäßig über den Datenraum verteilt.
- $|Domain(K)| >> |A|$ ($|x|$ = Kardinalität von x)

Beispiel: Personaldatei mit Nachnamen bis zur Länge 10 auf 1000 Datenseiten

- Häufungspunkte bei “Schmidt”, “Müller”, “Meyer” usw.
- $|Domain(K)| = 26^{10} >> |A| = 1000$

Verbreiteter Ansatz:

- h(K) sei möglichst unabhängig von K:

  h(K) erzeugt möglichst zufällige Adressen von den (nichtzufälligen) Schlüsseln K (daher auch der Begriff “Hash”-Funktion).

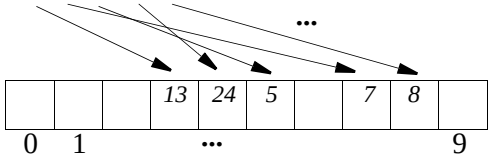
Beispiel: Divisionsmethode

h(K) = K MOD N	für numerische Schlüssel
h(K) = ORD(K) MOD N	für nicht-numerische Schlüssel

Konkrete Beipiele:
Für *ganzzahlige Schlüssel*: $h : K \rightarrow [0, 1, \ldots, N-1]$ mit $h(K) = K \text{ MOD } N$.

sei: $N = 10$

Schlüssel: 13, 7, 5, 24, 8, 18, 17, 31, 3, 11, 9, 30, 24, 27, 21, 19



Zeichenketten: Benutze die 'ORD-Funktion' zur Abbildung auf ganzzahlige Werte, z.B.:

$$h : STRING \rightarrow \left(\sum_{i=1}^{len(STRING)} (ORD(STRING[i]) - ORD(\text{'a'}) + 1) \right) \text{ MOD } N$$

sei: $N = 15$

JAN	→ 25 MOD 15 = 10	JUL	→ 43 MOD 15 = 13
FEB	→ 13 MOD 15 = 13	AUG	→ 29 MOD 15 = 14
MAR	→ 32 MOD 15 = 2	SEP	→ 40 MOD 15 = 10
APR	→ 35 MOD 15 = 5	OKT	→ 46 MOD 15 = 1
MAI	→ 23 MOD 15 = 8	NOV	→ 51 MOD 15 = 6
JUN	→ 45 MOD 15 = 0	DEZ	→ 35 MOD 15 = 5

Wie sollte N aussehen ?

- $N = 2^k$
  - einfach zu berechnen
  - K MOD 2^k liefert die letzten k Bits der Binärzahl K
    ⇒ Widerspruch zur Forderung nach Unabhängigkeit von K
- N gerade
  - h(K) gerade ⇔ K gerade
    ⇒ Widerspruch zur Forderung nach Unabhängigkeit von K
- N Primzahl
  - hat sich erfahrungsgemäß bewährt

Anforderungen von Datenbanksystemen (DBS):

- Gleichbleibend effiziente Suche
- Dynamisches Einfügen und Löschen von Datensätzen
- Hohe Speicherplatzausnutzung
  ⇒ der Adreßraum verändert sich dynamisch über die Laufzeit des DBS.

Dynamische Hashverfahren

Ein Hashverfahren ist *dynamisch*, wenn

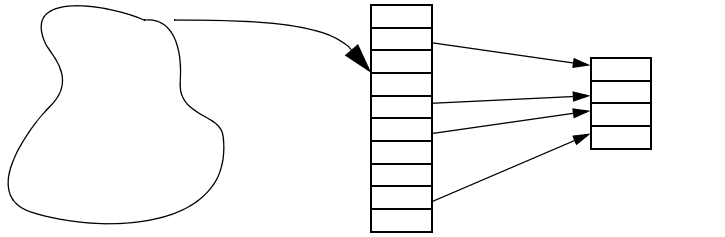
- Datensätze effizient eingefügt und gelöscht werden können und
- infolgedessen sich der Adreßraum dynamisch anpaßt.

Hashverfahren für DBS müssen dynamisch sein.

4.2 Klassifizierung

Bei den dynamischen Hashverfahren lassen sich zwei prinzipielle Vorgehensweisen unterscheiden:

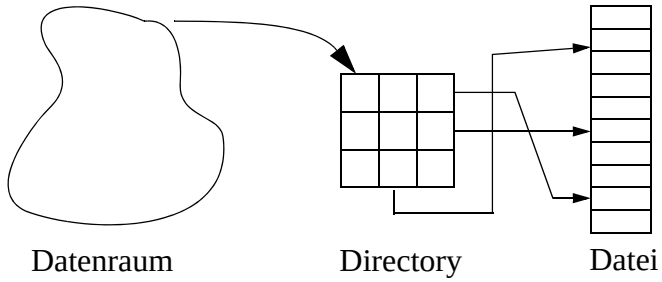
- ***Verfahren ohne Directory:***



Falls Seiten in der *Primärdatei* überlaufen, kann es notwendig sein, Datensätze in sogenannte *Überlaufseiten* einer *Sekundärdatei* auszulagern.

- Fallen viele Datensätze in Überlaufseiten, können die Suchzeiten stark zunehmen.

- ***Verfahren mit Directory:***



Die Größe des Directory ist eine monoton wachsende Funktion in der Anzahl der Datensätze.

Folgen:

- Auch das Directory muß für hinreichend große Dateien auf dem Sekundärspeicher abgelegt werden.
- Jeder Zugriff auf einen Datensatz verlangt mindestens 2 Sekundärspeicherzugriffe.

Besonderes Problem vieler Verfahren:

- Das Directory wächst superlinear in der Anzahl der Datensätze.

4.3 Verfahren mit Directory

Erweiterbares Hashing (extendible hashing) [FNPS 79]

Die Hashfunktion

Gegeben sei eine *Hashfunktion h*:

h : Domain (K) → {Indizes der Directoryeinträge}

Beim *erweiterbaren Hashing* liefert h eine Bitfolge:

$h(K) = (b_1, b_2, b_3, ...)$

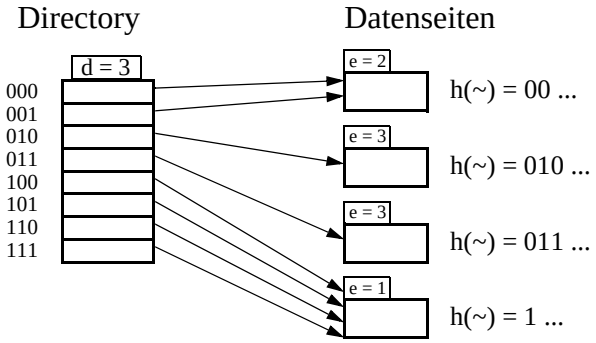
- h(K) heißt *Pseudoschlüssel.*
- Jede Hashfunktion ist einsetzbar, wenn die von ihr gelieferte Zahl als eine Folge von Bits interpretiert werden kann.

Das Directory

Das ***Directory*** D ist ein eindimensionales Feld:

D = ARRAY [0 ... 2^d-1] OF Seitennummer

- Die ***Größe des Directory*** ist eine 2er Potenz 2^d, d ≥ 0.
- d heißt ***Tiefe des Directory***.
- Jeder Directoryeintrag enthält einen Zeiger zu einer Datenseite (d.h. die Seitennummer).
- Verschiedene Directoryeinträge können zu derselben Datenseite verweisen.



Das Einfügen eines Schlüssels

Gegeben:

Datensatz mit Schlüssel K

1. Schritt:

Bestimme die ersten d Bits des Pseudoschlüssels $h(K) = (\underbrace{b_1, b_2, \ldots, b_d}, \ldots)$

2. Schritt:

Der Directoryeintrag $D[b_1 b_2 ... b_d]$ liefert die gewünschte Seitennummer.

Der Datensatz wird in der Seite mit der entsprechenden Seitennummer eingefügt.

Lokale Tiefe:

Um eine Seitenadresse zu bestimmen, benötigt man e Bits (e ≤ d).
e heißt *lokale Tiefe* (siehe Beispiel).

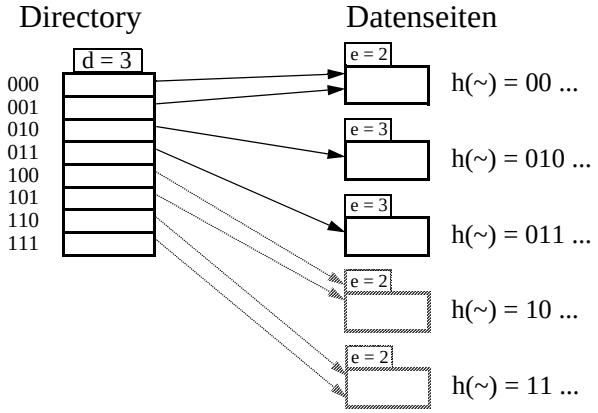
Fall: Datenseite muß aufgespalten werden

Grund: Sie läuft über oder ist z.B. mehr als 90 % gefüllt.

Beispiel:

Die Datenseite mit den Pseudoschlüsseln h(K) = 1... muß aufgespalten werden.

⇒ sie wird in 2 Datenseiten aufgespalten, jeweils mit lokaler Tiefe 2.



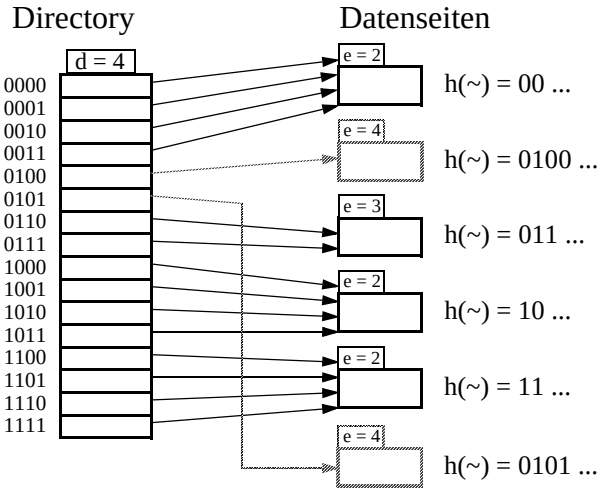
Fall: Directory muß verdoppelt werden.

Grund: Eine Datenseite läuft über mit lokaler Tiefe e = Tiefe des Directory d.

Beispiel:

Die Datenseite mit h(K) = 010... wird aufgespalten.

⇒ das Directory verdoppelt sich in seiner Größe, seine Tiefe d erhöht sich um 1.



Eigenschaften

- Das Directory muß i.a. aufgrund seiner Größe im Sekundärspeicher abgelegt werden.
  ⇒ jeder Datensatz kann garantiert mit 2 Zugriffen gefunden werden.

- *Wichtigster Nachteil:*
  Das *Wachstum des Directory* ist superlinear:
  bei Gleichverteilung: Directorygröße ist $O(n^{1+1/b})$
  (n = # Datensätze, b = # Datensätze pro Datenseite)
  bei Nichtgleichverteilung: Exponentielles Wachstum des Directory

- *Speicherplatzausnutzung* = ln 2 ≈ 0.693 für Gleichverteilung der Daten im Datenraum.

4.4 Verfahren ohne Directory

Bei Hashverfahren ohne Directory liefert die Hashfunktion direkt die Seitenadressen:

h: domain (K) $\rightarrow$ {Seitenadressen}

Beispiel:

h(K) = K mod M, M = 3, 5, ... (M Primzahl)

Kollisionsbehandlung

Kollision

- Ein Datensatz mit Schlüssel K wird in die Seite h(K) eingefügt.
- Die Seite h(K) ist bereits voll.
  d.h.: # Datensätze = b (Kapazität der Datenseite)

$\Rightarrow$ Jetzt liegt ein *Überlauf (eine Kollision)* vor.

Kollisionsbehandlung

- *Erweiterbares Hashing:* sofortiges Aufspalten
- *Hashverfahren ohne Directory:*
  - die Datenseiten werden nicht immer sofort aufgespalten, stattdessen:
  - besondere *Kollisionsbehandlung* der eingefügten Datensätze gemäß einer *Überlaufstrategie*.
  - Verbreitet ist die *Überlaufstrategie getrennte Verkettung*.

Überlaufstrategie getrennte Verkettung

Nutzung zweier verschiedener Seitentypen:

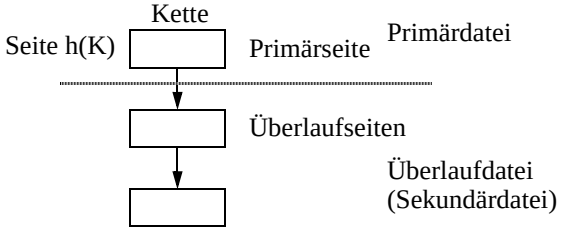
- ***Primärseiten***
  Die eigentlichen Datenseiten, deren Adressen von der Hashfunktion berechnet werden. Sie bilden die *Primärdatei*.
- ***Überlaufseiten***
  Sie speichern Datensätze, die nicht mehr in Primärseiten passen *(Überlaufsätze)*.
  Die Überlaufseiten bilden eine zweite Datei, die *Überlaufdatei (Sekundärdatei)*.

Es gilt:

(i) Überlaufseiten und Primärseiten sind verschieden.

(ii) Jede Überlaufseite ist genau einer Hashadresse h(K) zugeordnet.

(iii) Neue Überlaufseiten werden mit existierenden (Überlauf- oder Primär-) Seiten verkettet.



Es gilt:

- Die Suche nach einem Überlaufsatz benötigt mindestens 2 Seitenzugriffe.
- Entstehen lange Überlaufketten, können die Suchzeiten degenerieren.

Lineares Hashing [Lit 80]

Bis ca. 1978 ging man davon aus, daß

- die Größe der Primärdatei statisch ist und
- Überläufe nur durch Einführung von Überlaufsätzen gelöst werden können.

$\Rightarrow$ schnelle Degeneration der Suchzeiten, wenn die Primärseiten voll werden.

Folge von Hashfunktionen

- das Einfügen von K führt zu einem Überlauf
- keine weiteren Datensätze der Seite h(K) sollen in Überlaufseiten gespeichert werden

  $\Rightarrow$ Um K in einer Primärseite abzuspeichern, muß eine *neue Hashfunktion* gewählt werden.

  $\Rightarrow$ Wir benötigen *Folgen von Hashfunktionen*.

Grundideen des linearen Hashing:

- *Folge von Hashfunktionen*:
  $h_0, h_1, h_2, ...$
- *Expansion der Datei*:
  Die Datei wird um jeweils eine neue Primärseite erweitert.
- *Feste Splitreihenfolge:*
  Die Seiten werden in einer festen Reihenfolge gesplittet.
- *Expansionszeiger (Splitzeiger)* p
  bestimmt, welche Seite als nächstes gesplittet werden muß.
- *Kontrollfunktion*
  bestimmt, wann die Datei expandiert wird.

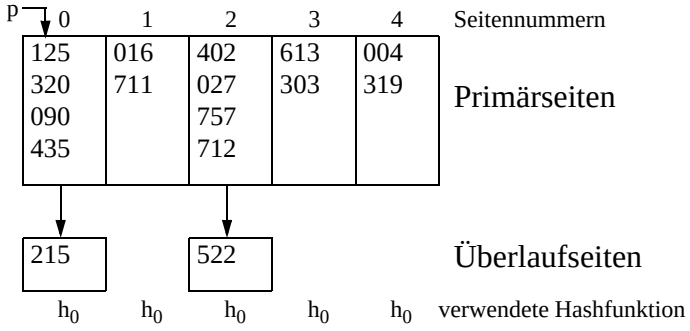
  Beim linearen Hashing wird folgende Kontrollfunktion für die Expansion benutzt:

  - Wenn der *Belegungsfaktor* bf einen vorgegebenen *Schwellenwert* (z.B. 0.8) überschreitet, wird die Datei expandiert.
  - Belegungsfaktor $bf = \frac{\#\text{ abgespeicherter Datensätze}}{\#\text{ Datensätze, die in Primärseiten passen}}$

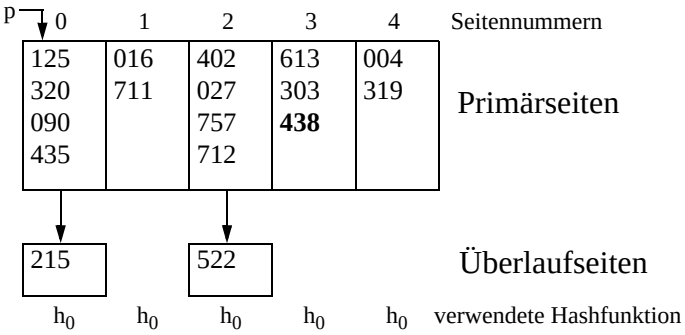
Beispiel:

Ausgangssituation:

- Datei mit 16 Datensätzen auf 5 Primärseiten der Seitengröße 4.
- Folge von Hashfunktionen: $h_0(K) = K \bmod 5$, $h_1(K) = K \bmod 10$, ...
- Aktuelle Hashfunktion: h_0.
- Belegungsfaktor bf = $\frac{16}{20} = 0.8$
- Schwellenwert für den Belegungsfaktor = 0.8
- Expansionszeiger p zeigt auf Seite 0.



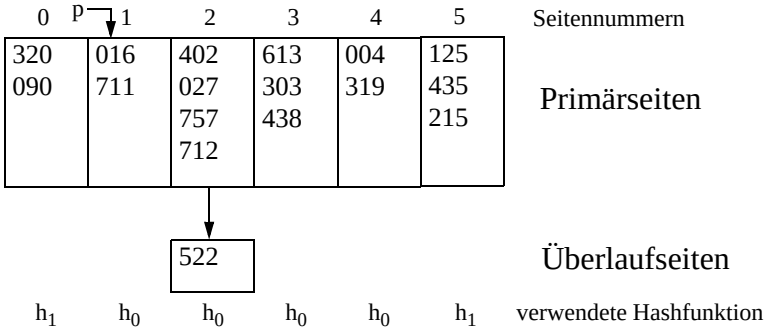
Einfügen eines Datensatzes mit Schlüssel 438:



- Der Belegungsfaktor übersteigt den Schwellenwert:

  bf = $\frac{17}{20} = 0.85 > 0.8$

⇒ *Expansion durch Split der Seite 0 auf die Seiten 0 und 5.*

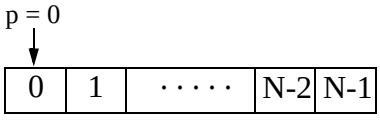


- Welche Datensätze von Seite 0 nach Seite 5 umgespeichert werden, bestimmt die Hashfunktion h_1:
  - alle Sätze mit $h_1(K) = 5$ werden umgespeichert
  - alle Sätze mit $h_1(K) = 0$ bleiben

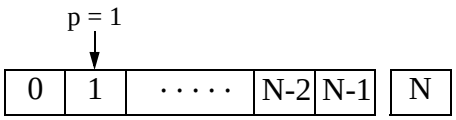
  Anschließend wird der Expansionszeiger p, der jeweils auf die nächste zu splittende Seite verweist, um 1 heraufgesetzt.

Prinzip der Expansion durch Splits

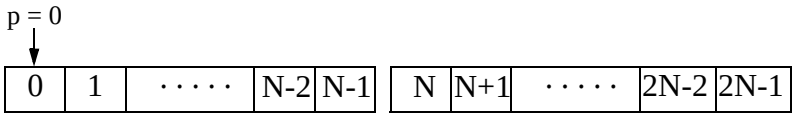
Ausgangssituation:



nach dem ersten Split:



nach der Verdoppelung der Datei:



Anforderungen an die Hashfunktionen

Wir benötigen eine Folge von Hashfunktionen $\{h_i\}$, $i \geq 0$, die folgende Bedingungen erfüllen:

1.) Bereichsbedingung:

h_L: domain (K) $\rightarrow \{0, 1, \ldots, (2^L N)-1\}$, $L \geq 0$

2.) Splitbedingung:

$h_{L+1}(K) = h_L(K)$ oder

$h_{L+1}(K) = h_L(K) + 2^L * N$, $L \geq 0$

Der *Level* L gibt an, wie oft sich die Datei schon vollständig verdoppelt hat.

Beispiel für eine mögliche Hashfunktion:

$h_L(K) = K \bmod (2^L N)$ erfüllt die Bedingungen 1.) und 2.)

Zusätzliche Eigenschaft der Hashfunktionen:

Für p = 0 haben alle Ketten die gleiche Wahrscheinlichkeit, einen neu eingefügten Datensatz aufzunehmen.

Wichtige Eigenschaften von linearem Hashing

- Die Seiten werden in einer fest vorgegebenen Ordnung gesplittet, anstatt die Seite zu splitten die überläuft.

  $\Rightarrow$ ein Datensatz, der einen Überlauf produziert, kommt nicht durch einen Split sofort in eine Primärseite, sondern erst nach einer Verzögerung (wenn der Splitzeiger auf die betreffende Kette zeigt).
- Der Prozentsatz der Überlaufsätze ist gering.
- Gutes Leistungsverhalten für gleichverteilte Datensätze.
- Der Adreßraum wächst linear an und ist gerade so groß wie nötig.

4.5 Partielle Erweiterungen

Lineares Hashing mit partiellen Erweiterungen [Lar 80]

Beobachtung:

- Hashverfahren sind am effizientesten, wenn die Datensätze möglichst gleichmäßig auf die Seiten in der Datei verteilt sind.

Einwand gegen das lineare Hashing:

- Die Verteilung der Datensätze auf die Seiten weicht stark von diesem Ideal ab.

Ursachen:

- Der Erwartungswert für den Belegungsfaktor einer bereits gesplitteten Seite ist nur halb so groß wie der Erwartungswert für den Belegungsfaktor einer noch nicht gesplitteten Seite.
- Die Primärdatei wird in nur einem Durchlauf verdoppelt.

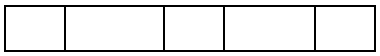
Partielle Expansionen

- Das Verdoppeln der Primärseitenzahl erfolgt nicht in einem, sondern in mehreren Durchläufen: Serie von $n_0 \geq 2$ *partiellen Expansionen.*

Vorgehen: (für $n_0 = 2$)

Ausgangssituation:

Datei mit 2N Seiten



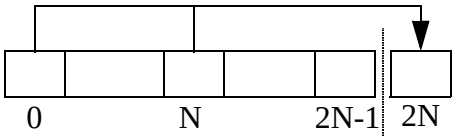
logisch unterteilt in N Paare (j, j+N) für j = 0, 1, ... , N-1

1. partielle Expansion

- Nach Einfügen von Datensätzen wird aufgrund der Kontrollfunktion mehr Speicherplatz verlangt.
- Expansion der Datei um die Seite 2N:

  Etwa $\frac{1}{3}$ der Sätze aus den Seiten 0 und N werden nach Seite 2N umgespeichert.

Datei mit 2N+1 Seiten

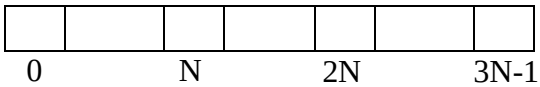


logisch unterteilt in N Paare (j, j+N) für j = 0, 1, ... , N-1

- Für j = 0, 1, ... , N-1 werden die Paare (j, j+N) um die Seite j+2N expandiert.

$\Rightarrow$ Die Datei hat sich von 2N Seiten auf 3N Seiten (auf das 1,5fache) vergrößert:

Datei mit 3N Seiten



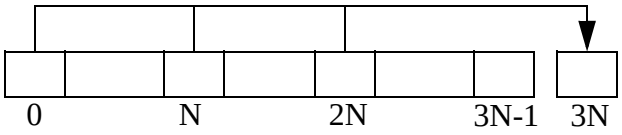
logisch unterteilt in N Tripel (j, j+N, j+2N) für j = 0, 1, ... , N-1

2. Partielle Expansion:

- Zunächst Expansion der Datei um die Seite 3N:

  Etwa je $\frac{1}{4}$ der Sätze aus den Seiten 0, N, 2N werden umgespeichert auf die Seite 3N.

  Datei mit 3N+1 Seiten



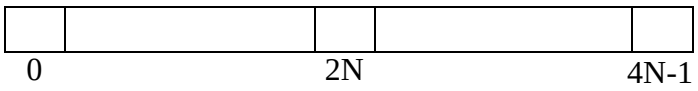
  logisch unterteilt in N Tripel (j, j+N, j+2N) für j = 0, 1, ... , N-1

- Nachfolgend werden für j = 1, 2, ... , N-1 die Tripel (j, j+N, j+2N) um die Seite j+3N expandiert, wobei ca. $\frac{1}{4}$ der Sätze aus den Seiten j, j+N, j+2N umgespeichert werden.

Resultat:

- Die Datei hat sich auf 4N Seiten verdoppelt:

  Datei mit 4N Seiten



- Weiter geht es mit dieser neuen Ausgangssituation.

Analyse der partiellen Expansionen

Für das lineare Hashing gilt während der 1. bzw. 2. partiellen Expansion:

- Der Belegungsfaktor einer gesplitteten Seite ist $\frac{2}{3}$ bzw. $\frac{3}{4}$ des Belegungsfaktors einer ungesplitteten Seite.

Kontrollfunktion:

Larson schlägt vor, als Kontrollfunktion die *Speicherplatzausnutzung* zu wählen.

$$\text{Speicherplatzausnutzung} = \frac{\text{\# tatsächlich abgespeicherter Sätze}}{\text{\# möglicher Sätze (in Primär- und Überlaufseiten)}}$$

Expansionsregel:

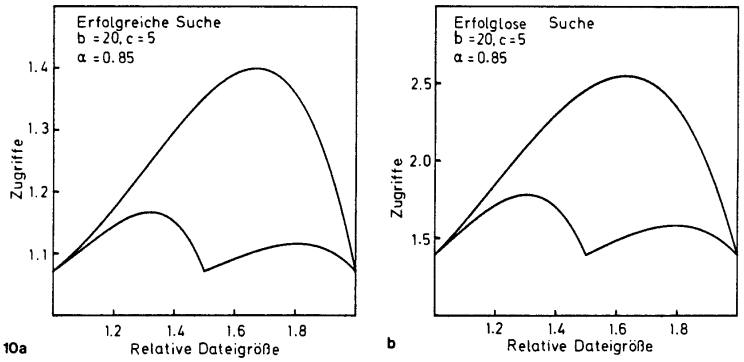
- Jedesmal, wenn ein Datensatz eingefügt wird, wird die Speicherplatzausnutzung überprüft.
- Falls die Speicherplatzausnutzung größer als ein Schwellenwert α ist, $0 < \alpha < 1$:

  wird die Datei um eine weitere Seite expandiert.

Somit ist die Kontrollfunktion in gewissem Sinne optimal und garantiert, daß die Speicherplatzausnutzung annähernd konstant gleich α ist.

Leistungsverhalten

Vergleich: # Zugriffe bei erfolgreicher bzw. erfolgloser Suche für $n_0 = 1$ und $n_0 = 2$.
b = Kapazität einer Primärseite, c = Kapazität einer Überlaufseite



Beobachtungen:

- Die Effizienz des Suchens ist dann am größten, wenn die Belegung aller Ketten möglichst gleich ist.
- zyklisches Verhalten:

  Länge eines Zyklus = 2 * Länge des vorangegangenen Zyklus

Vergleich: Durchschnittliche # Zugriffe für $n_0 = 1$, $n_0 = 2$ und $n_0 = 3$
für eine Datei mit b = 20, c = 5, $\alpha = 0.85$.

	$n_0 = 1$	$n_0 = 2$	$n_0 = 3$
Erfolgreiche Suche	1.27	1.12	1.09
Erfolglose Suche	2.12	1.58	1.48
Einfügen	3.57	3.21	3.31
Entfernen	4.04	3.53	3.56

Folgerung:

$n_0 = 2$ partielle Expansionen stellen den besten Kompromiß dar zwischen

- Effizienz des Suchens,
- Effizienz der Update-Operationen (Einfügen und Entfernen) und
- Komplexität des Programmcodes.

4.6 Ordnungserhaltung

Wichtige Eigenschaft von Indexstrukturen (vgl. 1. Kapitel):

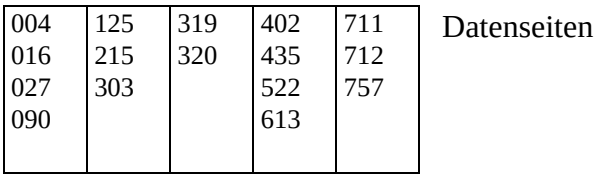
lokale Ordnungserhaltung

Datensätze mit Schlüsseln, die in der Ordnung aufeinander folgen, sollten in der gleichen Seite gespeichert werden.

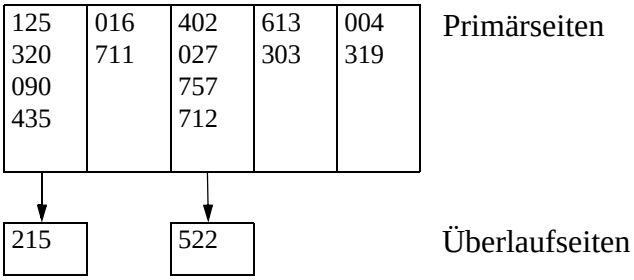
- Suchbäume wie z.B. B-Bäume sind (lokal) ordnungserhaltend.
- Lineares Hashing ist mit den bisherigen Hashfunktionen nicht ordnungserhaltend.

Beispiel:

B^+-Baum:



Lineares Hashing (Beispiel von Seite 50):



Für eine Bereichsanfrage (z.B. [0..125]) müßten in diesem Beispiel beim linearen Hashing alle Seiten durchsucht werden.

⇒ Wir benötigen ein ordnungserhaltendes lineares Hashing.

Ordnungserhaltendes Lineares Hashing (eindimensionales Interpolationsverfahren)

Datenraum: Domain (K) = [0,1)

[0,1) ist durch Transformation herstellbar.

Bitstring-Darstellung

- Jeder Schlüssel K ∈ [0,1) kann durch einen Bitstring $(b_1, \ldots, b_w)$, w ∈ *N*, dargestellt werden mit

$$K = \sum_{i=1}^{w} b_i 2^{-i}$$

- Diese Bitstring-Darstellung bewahrt die Ordnung der Schlüssel:

  Für zwei Schlüssel K_i und K_j mit $K_i < K_j$ gilt:

  $(b_1^i, \ldots, b_m^i) \leq (b_1^j, \ldots, b_m^j)$

  Dabei ist ≤ die lexikographische Ordnung auf Bitstrings

Ordnungserhaltende Hashfunktion

- Sei N die aktuelle Anzahl von Primärseiten der Datei mit

  $$2^L \leq N < 2^{L+1},$$

  wobei $L \in N_{,0}$ der ***Level der Datei*** ist, der angibt, wie oft sich die Datei komplett verdoppelt hat.
- Die Primärseiten seien mit {0, 1, . . ., N-1} adressiert.

Die Hashfunktion h:

$$h(K,N) := \begin{cases} \sum_{i=1}^{L+1} b_i \cdot 2^{i-1} & \text{, falls } \sum_{i=1}^{L+1} b_i \cdot 2^{i-1} < N \\ \sum_{i=1}^{L} b_i \cdot 2^{i-1} & \text{, sonst} \end{cases}$$

- h(K,N) nimmt die L bzw. L+1 Präfix-Bits des Bitstrings von K, dreht die Reihenfolge um und interpretiert das Resultat als Integer-Zahl.
- h(K,N) erfüllt die Bereichs- und Splitbedingung und ist ordnungserhaltend.

Beispiel: Domain (K) = [0, 1) , N = 9 , L = 3 , p = 1

Länge	Datenintervall	Seitenadresse der	Bitumkehrung	Bitdarstellung
1/16	[0, 1/16)	0	0000	0000
1/16	[1/16, 1/8)	8	1000	0001
1/8	[1/8, 1/4)	4	100	001
1/8	[1/4, 3/8)	2	010	010
1/8	[3/8, 1/2)	6	110	011
1/8	[1/2, 5/8)	1	001	100
1/8	[5/8, 3/4)	5	101	101
1/8	[3/4, 7/8)	3	011	110
1/8	[7/8, 1)	7	111	111

Auch Varianten vom linearen Hashing, wie das lineare Hashing mit partiellen Erweiterungen, lassen sich einfach auf das eindimensionale Interpolationsverfahren übertragen.

4.7 Literatur

Eindimensionale dynamische Hashverfahren werden in einer Reihe von Lehrbüchern über Datenbanksysteme eingeführt (vgl. 1. Kapitel). Eine Übersichtsdarstellung findet sich auch in:

[Lar 83] Larson P.-Å.: *'Dynamische Hashverfahren'*, Informatik-Spektrum, Springer, Vol. 6, No. 1, 1983, pp. 7-19.

Originalliteratur

[FNPS 79] Fagin R., Nievergelt J., Pippenger N., Strong H. R.: *'Extendible Hashing - A fast Access Method for Dynamic Files'*, ACM Trans. on Database Systems, Vol. 4, No. 3, 1979, pp. 315-344.

[Lar 80] Larson P. A.: *'Linear Hashing with Partial Expansions'*, Proc. 6th Int. Conf. on Very Large Databases, Montreal, Canada, 1980, pp. 224-232.

[Lit 80] Litwin W.: *'Linear Hashing: A New Tool for File and Table Addressing'*, Proc. 6th Int. Conf. on Very Large Databases, Montreal, Canada, 1980, pp. 212-223.