B*-BÄUME

Beobachtung:

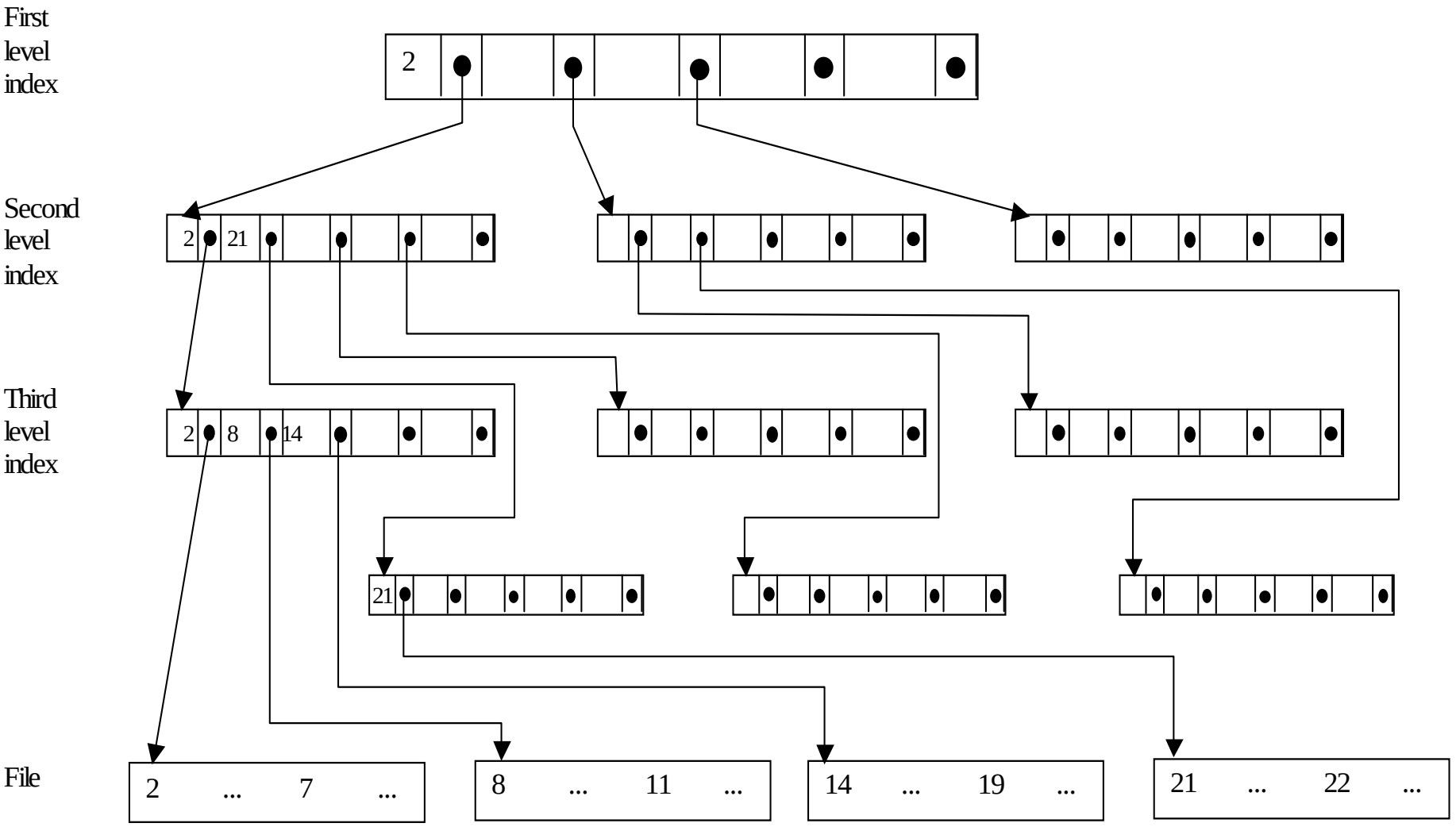
- Ein Index ist seinerseits wieder nichts anderes als eine Datei mit unpinned Records.

- Es gibt keinen Grund, warum man nicht einen Index über einem Index haben sollte, und so weiter, bis der letzte Index in einen einzigen Block paßt.

- Eine solche Hierarchie von Indizes (Multi-Level-Index) kann weitaus effektiver sein als ein einziger Index.

- Eine Multi-Level-Index Hierarchie kann als ein Baum betrachtet werden.

- Beispiel: Jeder Index-Block kann fünf Einträge enthalten.


First level index
2
Second level index
2
21
Third level index
2
8
14
21
File
2 ... 7 ...
8 ... 11 ...
14 ... 19 ...
21 ... 22 ...

Anforderungen an einen Multi-Level-Index:

- Baum von Indizes mit einer unspezifizierten Anzahl von Ebenen.

- Balancierter Baum:
  Jeder Pfad von der Wurzel (erste Index-Ebene) zu einem Blatt hat die gleiche Länge.

- Die Sätze der Hauptdatei sind unpinned.

→ B*-Baum

Definition:

Ein B*-Baum ist ein Baum, dessen Knoten aus Blöcken bestehen, mit den folgenden Eigenschaften:

- Jeder Blattknoten (außer der Wurzel) enthält zwischen k* und 2k*-1 sortierte Datensätze.
  Jeder Block ist also mindestens zur Hälfte gefüllt.

- Jeder Nichtblattknoten mit Ausnahme der Wurzel enthält zwischen k und 2k-1 Indexeinträge.
  Der erste Eintrag jedes Knotens hat keinen zugeordneten Schlüssel (dies spart Platz).

- Die Wurzel enthält maximal 2k-1 Indexeinträge oder maximal 2k*-1 Datensätze.

- Alle Blätter liegen auf gleicher Höhe.

Eine Variante des B*-Baum ist es, die Blöcke der Hauptdatei zu den Blättern des Baumes zu machen.

Dieser sehr einfache Ansatz ist allerdings nicht der Effizienteste (insbesondere im Platzverbrauch).

→ Ein besserer Ansatz wird beim Betrachten von B*-Bäumen mit pinned Records aufgezeigt.

Lookup

Gesucht wird ein Satz mit dem Schlüssel *v*.

Die Suche beginnt an der Wurzel des B*-Baumes.

Angenommen die Suche ist am Knoten B angekommen, dann ist B entweder

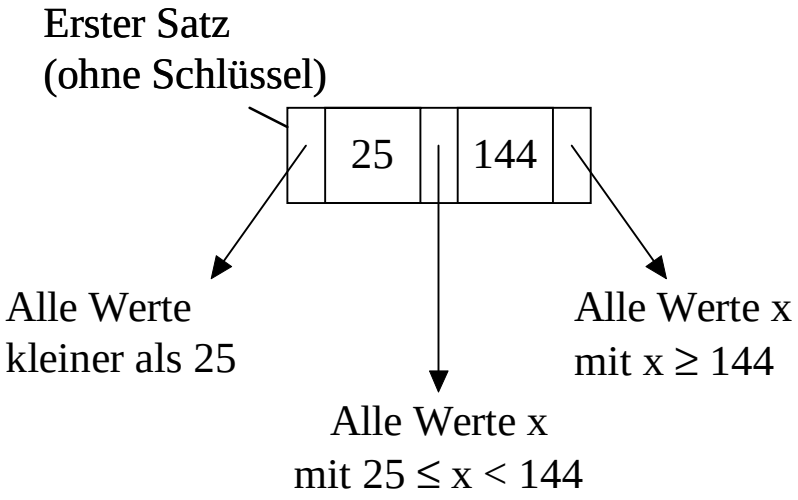
- Ein Blatt (dies kann durch Zählen der durchlaufenen Ebenen feststellen), oder
- Ein Knoten.

Wenn B ein Blatt ist, muß lediglich der Block nach dem Satz durchsucht werden.

Ist B hingegen ein Knoten, dann ist B ein Index-Block.

Ist B ein Index-Block, wird festgestellt, welcher Indexeintrag den Wert *v* überdeckt.

Anmerkung: Der erste Satz von B hat keinen Schlüssel, er überdeckt alle Werte die kleiner sind als der Schlüsselwert des zweiten Satzes.



Der Satz, der *v* überdeckt, enthält einen Zeiger zu einem weiteren Block B' dieser Block folgt dem Block B auf dem Pfad zu dem gesuchten Satz.

Diese Schritte werde wiederholt bis ein Blatt erreicht wird.

Modifikation

- Wenn der Schlüsselwert geändert wird,

  → Löschen und Einfügen.

- Wenn der Schlüsselwert unverändert bleibt,

  → Lookup und Rewrite.

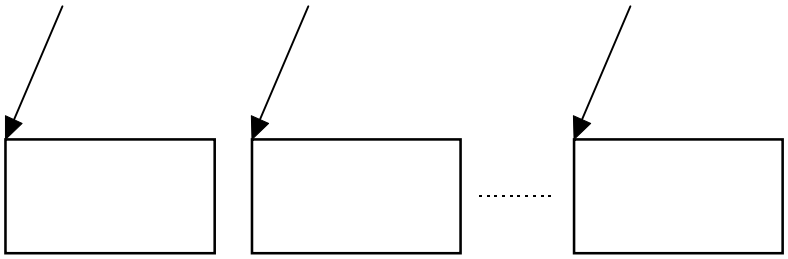
Einfügen

Es wird ein Satz mit dem Schlüssel *v* eingefügt.

- Lookup (v)
  → Block B.

- Wenn in B weniger als 2k*-1 Sätze sind, wird der neue Satz an der richtigen Stelle der Sortierreihenfolge eingefügt (unpinned Records).

- Der neue Satz kann niemals der erste in Block B sein, außer wenn B der äußerst linke Block ist.



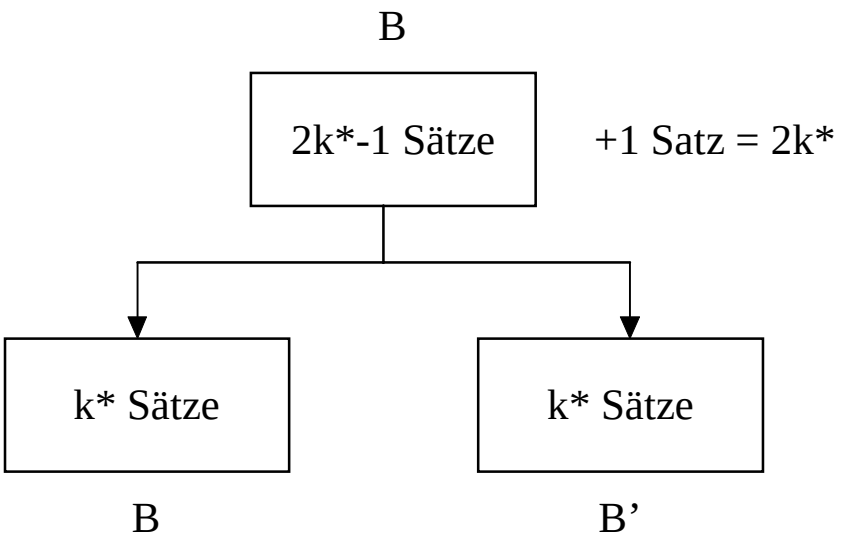
Hauptdatei

Daraus folgt, daß es unter keinen Umständen nötig wird, den Schlüsselwert eines Vorfahren von B zu ändern, denn der erste Satz jedes Indexblocks hat keinen Indexwert.

- Wenn aber in bereits 2k*-1 Sätze in Block B vorhanden sind (der Block ist also voll), dann

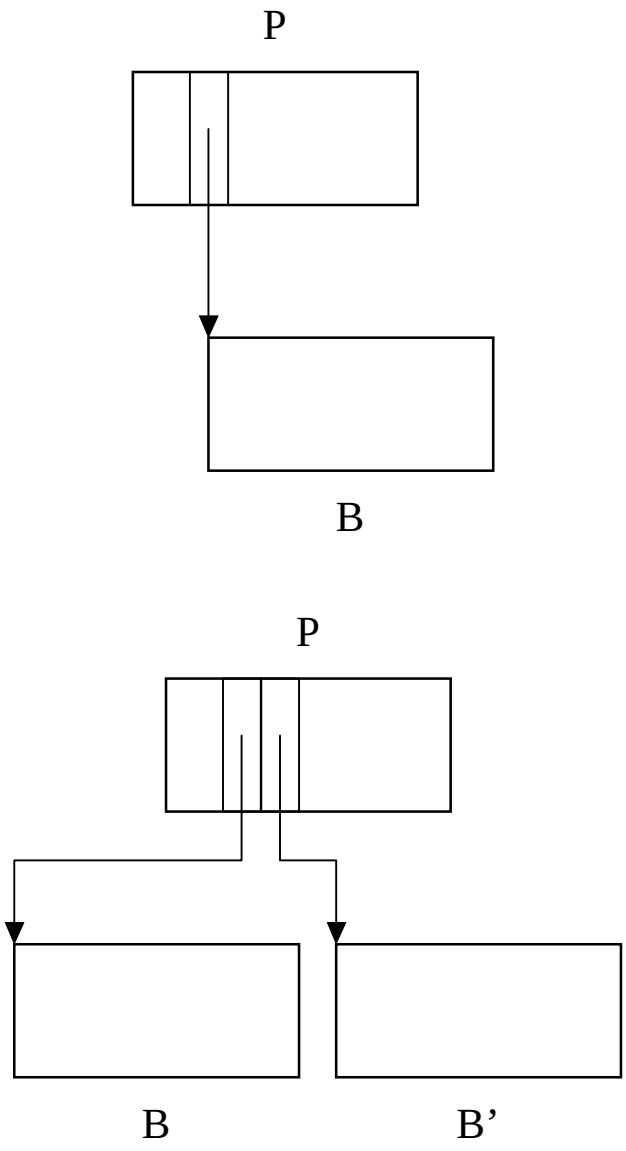
    - erzeuge einen neuen Block B’,

    - teile die Sätze von B und den eingefügten Satz auf die zwei Blöcke auf. Jeder Block ist danach mit k* Sätzen gefüllt.

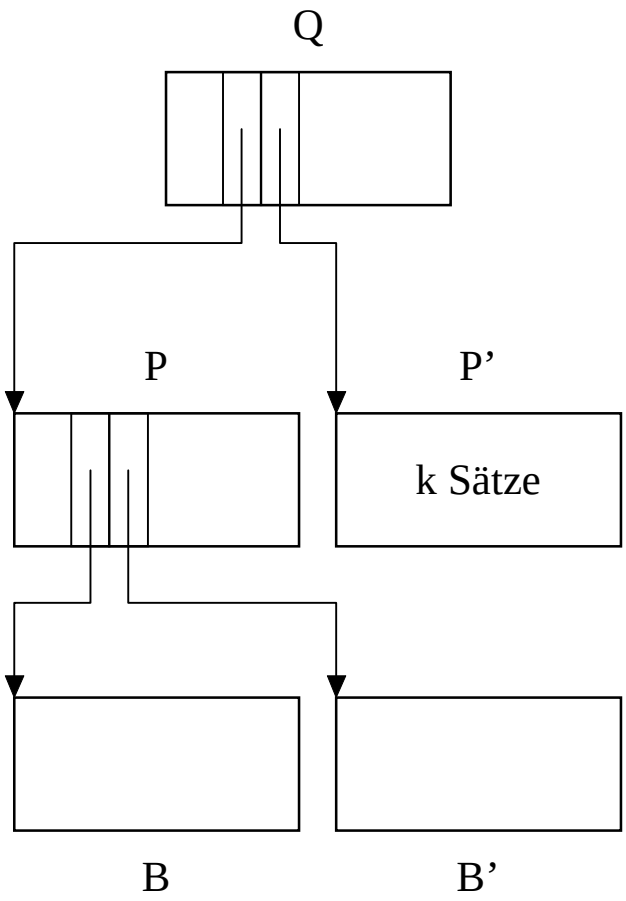


Jetzt muß noch der Index auf den neuen Stand gebracht werden:

P sei der Vaterknoten von B. Dann wird in den Block P ein Satz für den neuen Block B’ eingefügt, dies geschieht mit dem eben beschriebenen Verfahren nur mit dem Wert k statt k*.



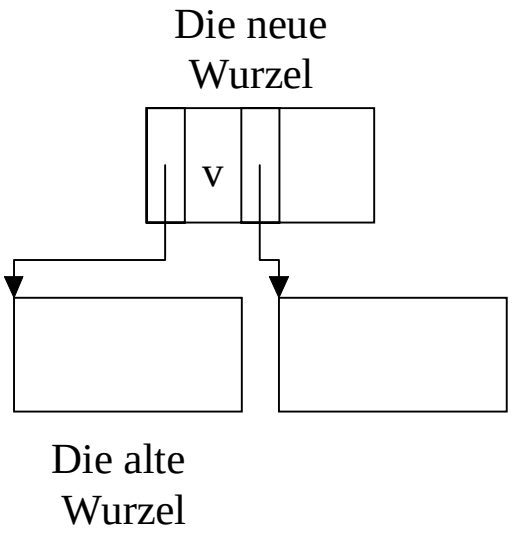
Falls P schon mit 2k-1 Sätzen gefüllt war:



Diese Prozeß kann bis in die Wurzel propagiert werden, wobei aber nur Vorfahren von B betroffen sind.

Falls der Prozeß die Wurzel erreicht, dann

- teile die Wurzel,
- Erzeuge eine neue Wurzel mit zwei Kindern.



Anmerkung:

Dies ist die einzige Situation in der ein Index-Block weniger als k Sätze haben kann.

Löschen

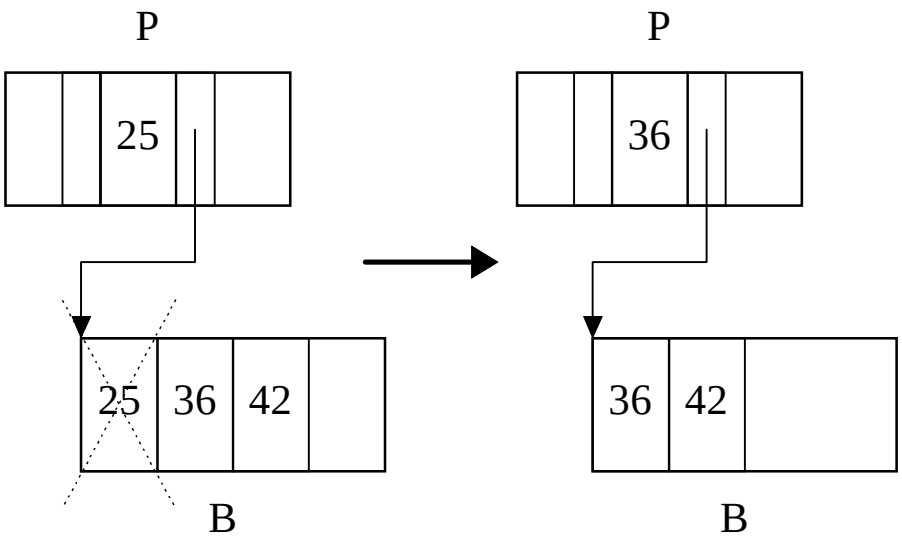
Der Satz mit dem Schlüssel *v* soll gelöscht werden.

- Lookup v
  → Block B.

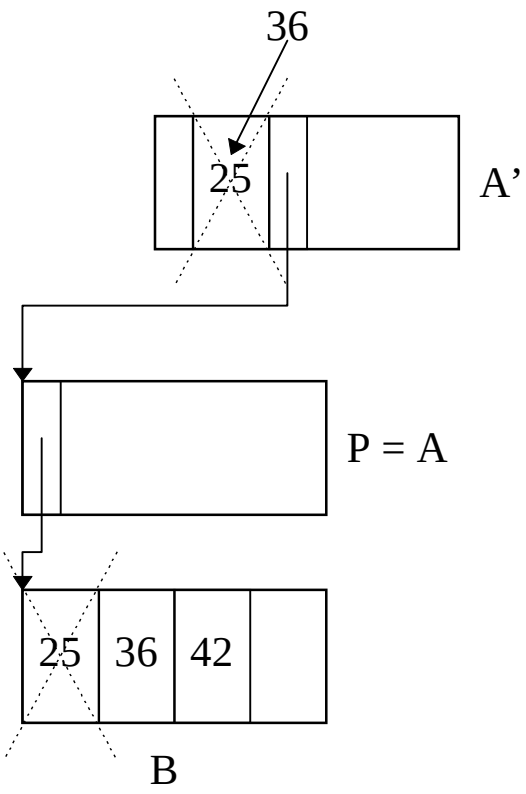
- Lösche den Satz in B.

1. Falls nach dem Löschen in B k* oder mehr Sätze übrig sind, ist der Vorgang beendet, es sei denn:

   - Der gelöschte Satz war der erste Satz in B, dann muß im Vater P von B der Schlüsselwert für B geändert werden.

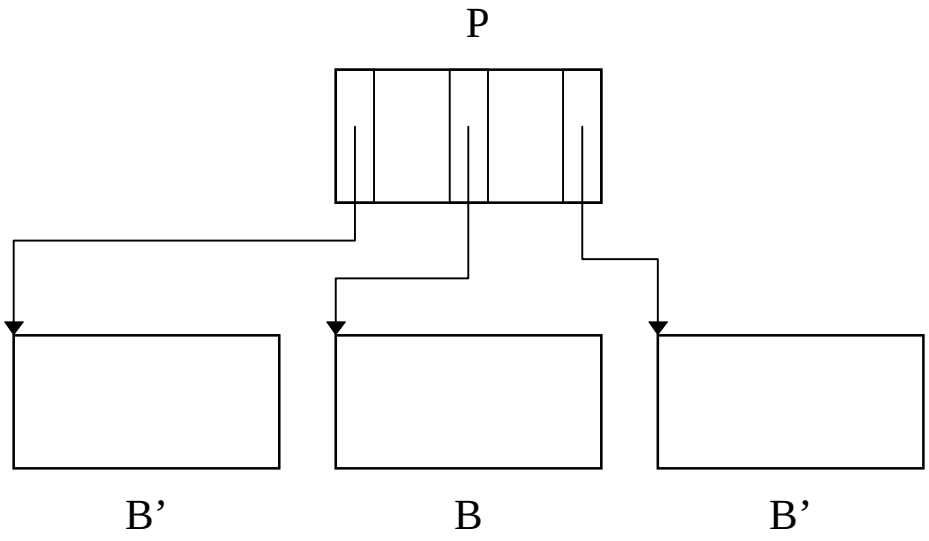


- Falls B das erste Kind von P ist, hat P keinen Schlüssel für B. Dann muß ein Vorfahre A von B gefunden werden für den gilt, daß er nicht das erste Kind seines Vaterknotens A’ ist.
  Dann wird der neue (kleinste) Schlüssel von B in den Satz von A’ eingetragen, der auf A verweist.



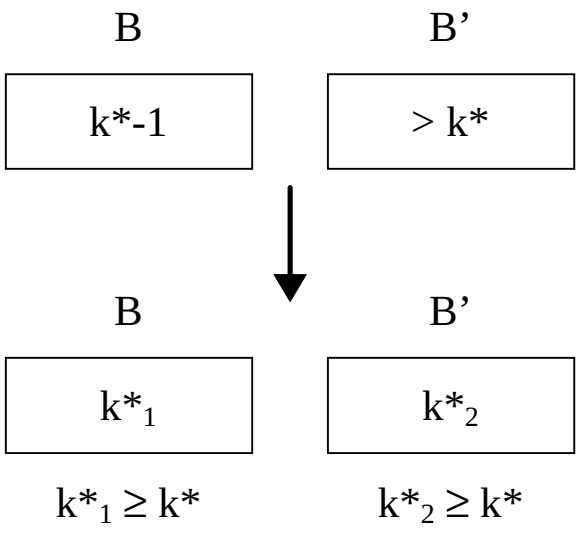
2. Falls nach dem Löschen in B nur noch k*-1 Sätze übrig sind, dann:

- Betrachte einen Block B’ der den selben Vaterknoten P hat und der unmittelbar links oder rechts von B liegt.



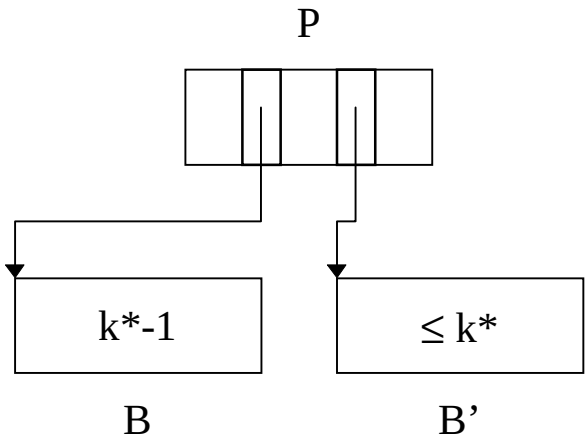
- Falls B’ mehr als k* Sätze hat, verteile die Sätze von B und B’ gleichmäßig auf beide Blöcke.

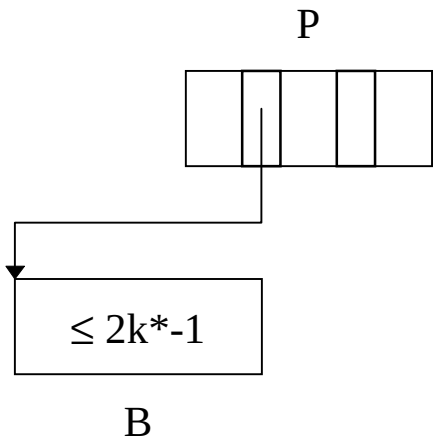
  Modifiziere den Schlüsselwert von B und/oder B’ und, falls nötig, propagiere die Änderungen zu den Vorfahren von B.



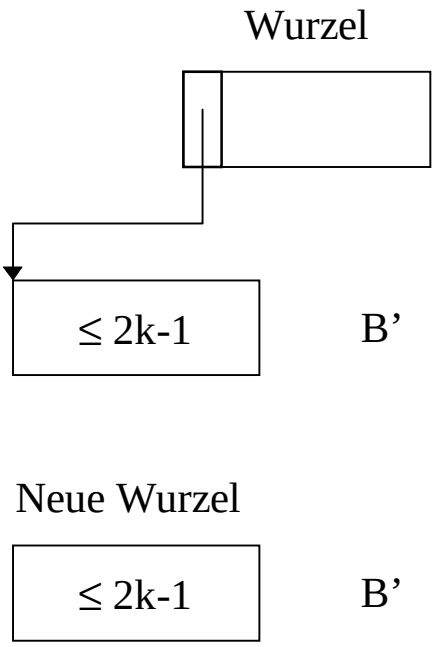
- Falls B’ nur k* Sätze hat, dann vereinige B mit B’ zu einem Block mit k*-1 + k* = 2k*-1 Sätzen.

  Lösche den Eintrag des rechten der beiden Blöcke (rekursiver Aufruf der Löschprozedur).





- Falls nach Anwendung dieses Verfahrens die Wurzel nur noch einen Zeiger enthält, kann die Wurzel wegfallen und das einzige Kind der Wurzel wird zur neuen Wurzel.



Anmerkung:

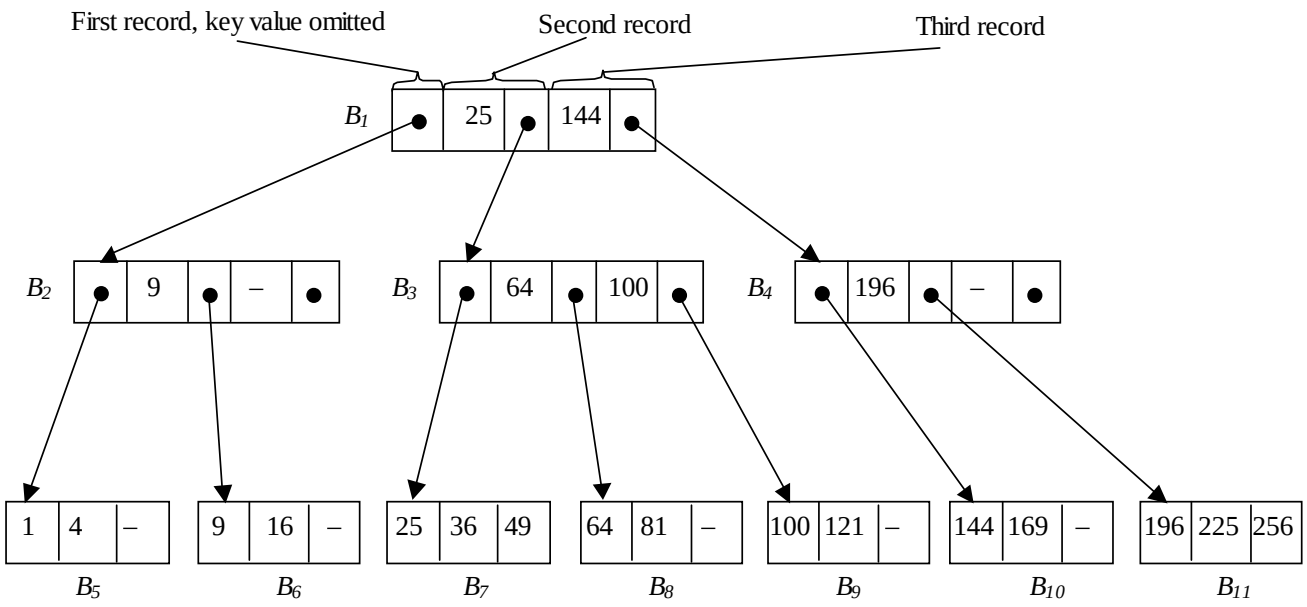
Dies ist der einzige Fall in dem die Anzahl der Ebenen des Baumes kleiner wird.

Beispiel:

Auf dem folgenden, bereits vorhandenen B*-Baum sollen zwei Operationen durchgeführt werden:

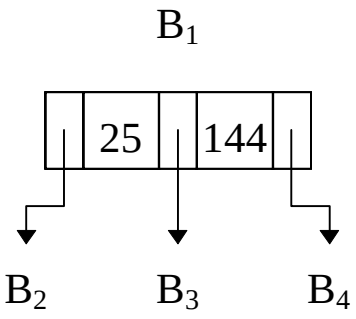
1. Einfügen eines Satzes mit Schlüsselwert 32

2. Löschen des Satzes mit dem Schlüsselwert 64

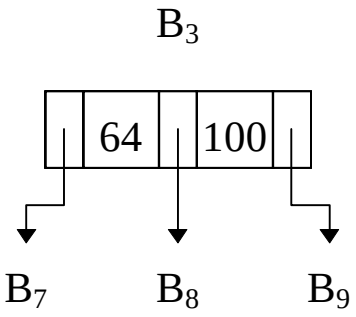


1. Einfügen von 32

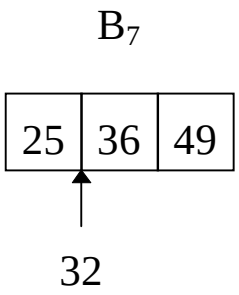
- Zuerst wird ein Pfad von der Wurzel zu dem Block, in den der Wert 32 gehört, gesucht.
- B_1 : Der Wert 25 überdeckt 32. Wir gehen also weiter zu Block B_3.



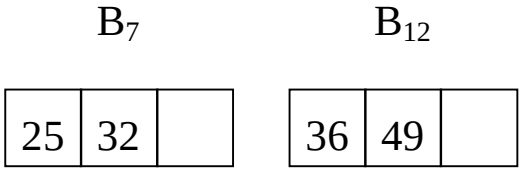
- B_3 : 32 ist kleiner als der Schlüsselwert 64 der zweiten Satzes von B_3, daher wird em Zeiger des ersten Satzes zu Block B7 gefolgt.



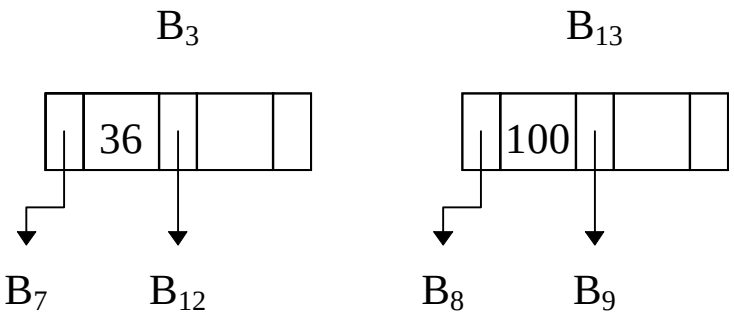
- B_7 : Der Block B_7 ist ein Blatt und daher ein Block der Hauptdatei. Der Wert 32 gehört hier zwischen die Werte 25 und 36.



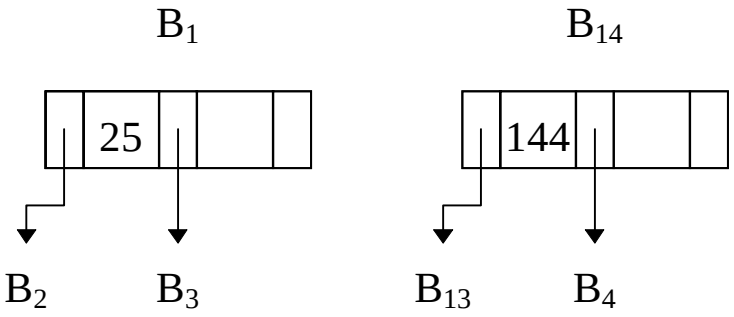
- Der Block B7 ist allerdings bereits voll, daher wird ein neuer Block B_{12} angelegt. Die Werte 25 und 32 kommen dann in Block B_7 und die Werte 36 und 49 in Block B_{12}.



- Nun muß ein Satz mit dem ersten Schlüssel von Block B_{12} in B_3 (der Vorfahre von B_7) eingefügt werden.
  Der Block B_3 ist aber auch schon voll, daher wird ein weiterer Block (B_{13}) angelegt. Die Sätze mit den Zeigern auf B_7 und B_{12} kommen in Block B_3 und die Sätze mit Zeigern auf B_8 und B_9 kommen in Block B_{13}.

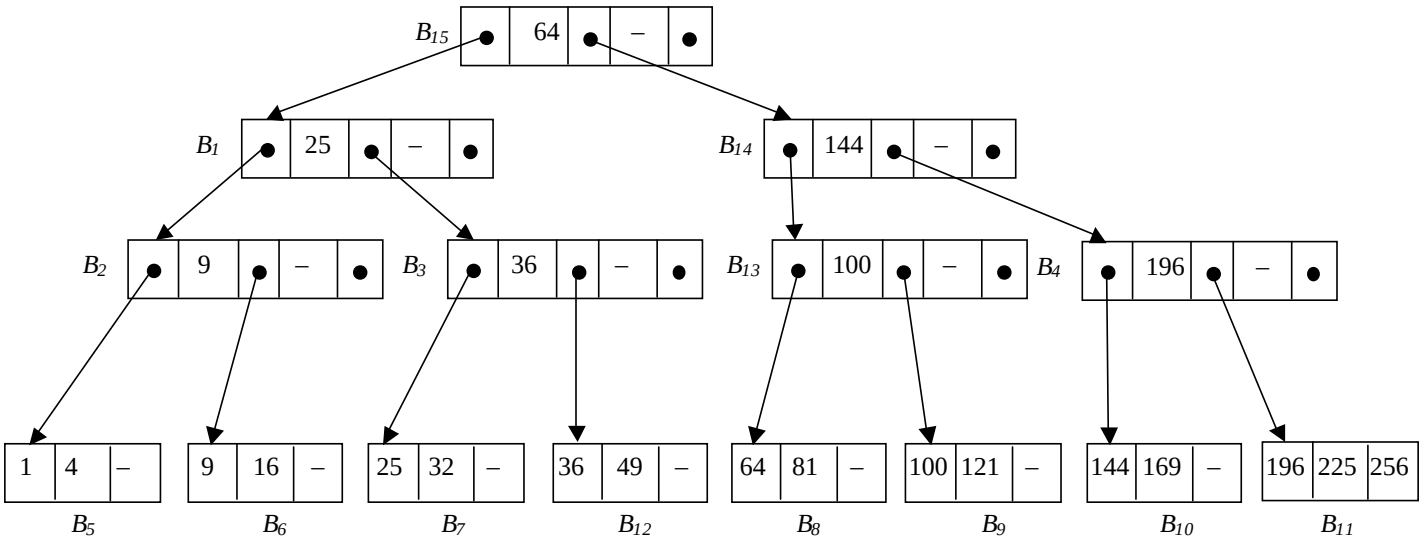


- Jetzt muß ein Satz mit dem Schlüsselwert 64 und einem Zeiger auf B_{13} in B_1 eingefügt werden. Leider bekommt B_1 dadurch 4 Sätze. Deshalb wird ein neuer Block B_{14} angelegt. Die Sätze mit Zeigern auf B_2 und B_3 kommen in Block B_1 und die Sätze mit Zeigern auf B_{13} und B_4 kommen in Block B_{14}.



- Da B_1 die Wurzel war und gesplittet wurde, wird jetzt ein neuer Block B_{15} erzeugt, der zur Wurzel wird und Zeiger auf B_1 und B_{14} hat.

  Der endgültige Baum sieht dann wie folgt aus:

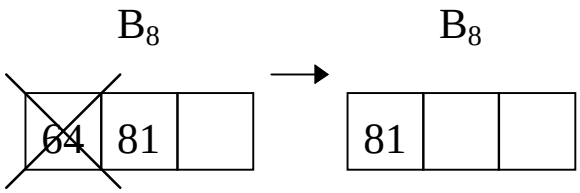


2. Löschen von 64.

- Durch suchen (lookup) findet man heraus, daß der Pfad zu dem Block der den Wert 64 enthält wie folgt ist:

$$B_{15}, B_{14}, B_{13}, B_8$$

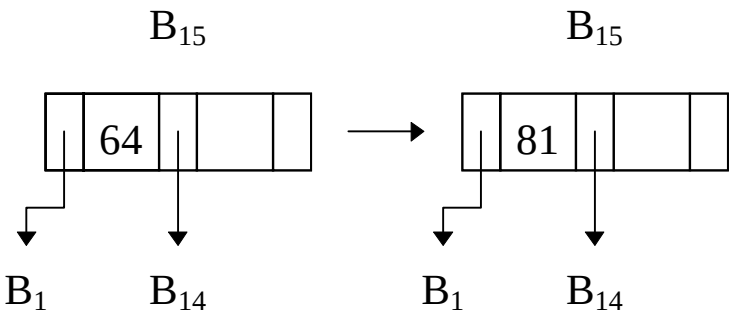
- Der Wert 64 wird aus dem Block B_8 gelöscht.



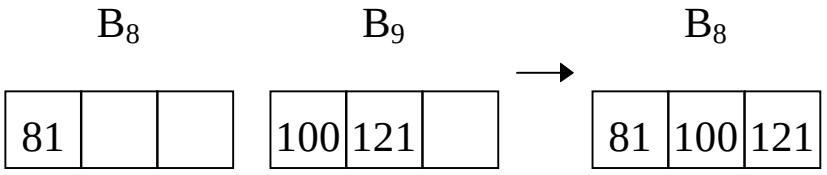
- Da es der erste Satz in dem Block war, muß auch der neue Schlüsselwert (81) in der Hierarchie nach oben propagiert werden.

- Da B_8 das links-außen liegende Kind von B_{13} ist, wird B_{13} nicht geändert, das gleiche gilt für B_{14} da für B_{14} der Block B_{13} das links-außen liegende Kind ist.

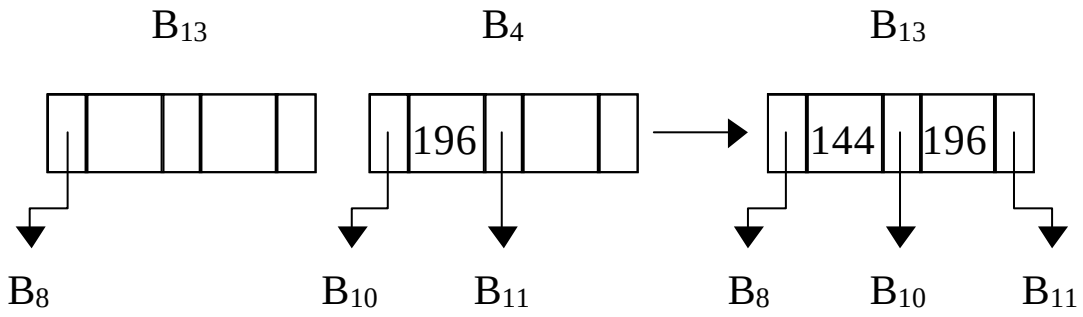
- B_{14} ist allerdings nicht links-außen in B_{15} verankert, daher muß ein Schlüsselwert von B_{15} geändert werden.



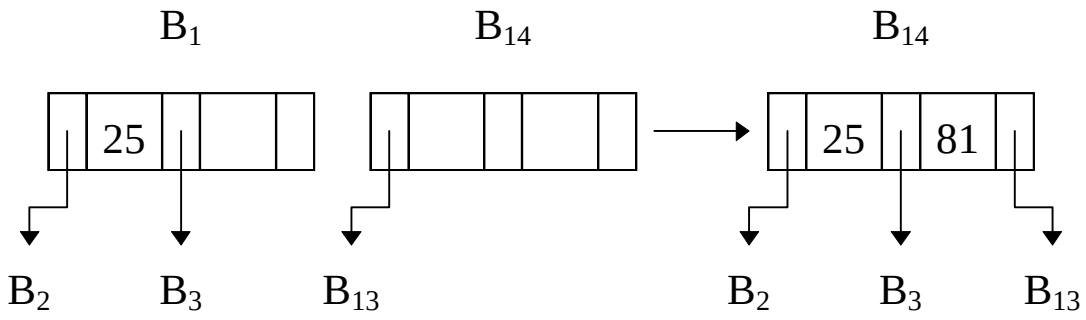
- Durch das löschen von 64 in Block B_8 hat dieser nur noch einen einzigen Satz. Dies widerspricht der Vorschrift, daß jeder Block mindestens k, also in diesem Fall 2, Sätze haben muß.
  Da B_8 keinen linken Geschwister hat, wird sein rechter Geschwister B_9 überprüft. B_9 hat zwei Sätze, B_8 und B_9 können also zusammengefaßt werden.



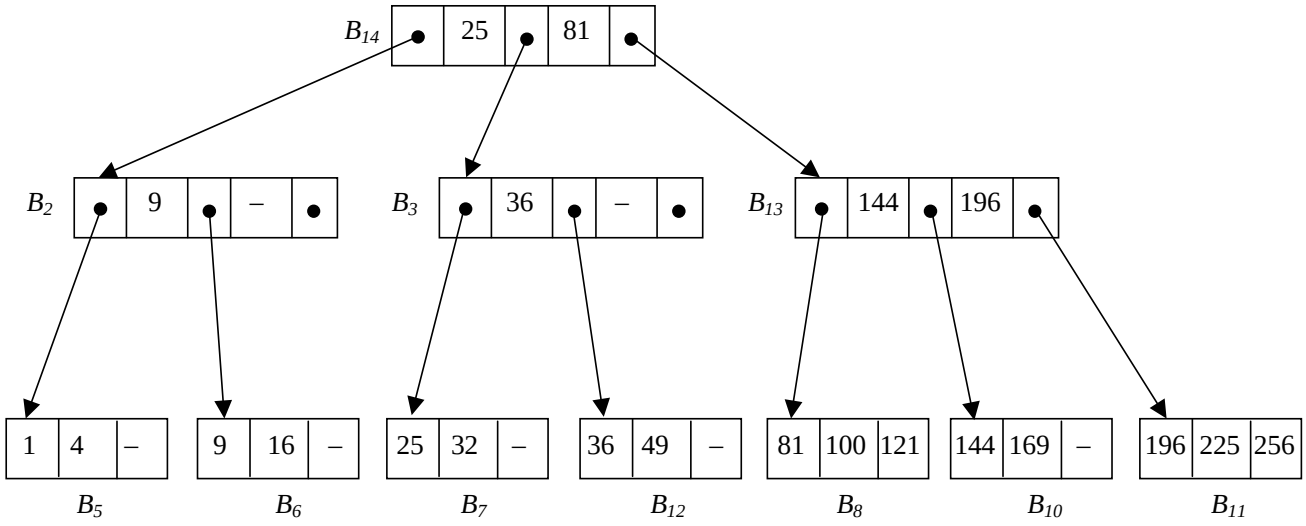
- B_{13} hat jetzt nur noch das eine Kind B_8. B_{13} wird deshalb mit B_4 zusammengefaßt:



- Jetzt hat auch B_{14} nur noch ein Kind und wird mit B_1 zusammengefaßt:



- Block B_{15} hat jetzt nur noch ein Kind und, da er die Wurzel ist, wird er gelöscht. B14 wird zur neuen Wurzel:



Leerer B*-Baum:

$k = k^* = 2$

Einfügen der Werte

2, 5, 8, 9, 3, 11, 50

⇓

Aufbauen des B*-Baumes.

Ändern des Schlüsselwertes 50 nach 7.

50 → 7

LAUFZEITANALYSE FÜR OPERATIONEN AUF B*-BÄUMEN

Annahme:

gegeben ist eine Datei mit n Sätzen, die in einem B*-Baum mit den Parametern k und $k*$ organisiert ist.

- Der Baum wird nicht mehr als

  $n/k*$ Blätter haben.

- Der Baum wird nicht mehr als

  $n/k \cdot k*$ Eltern von Blättern haben.

- Des weiteren kann er nicht mehr als

  $n/k^2 \cdot k*$ Eltern von Eltern von Blättern haben.

- und so weiter ...

Wenn ein Pfad von der Wurzel zu den Blätter i Knoten hat, dann gilt:

$$n \geq k^{i-1} \cdot k^*$$

Es folgt hieraus:

$$i \leq 1 + \log_k (n/k^*)$$

Für eine Datei mit n Sätzen in einem B*-Baum mit den Parametern k und k* folgt daher:

Für einen lookup benötigt man

$\underline{i \leq 1 + \log_k (n/k^*)}$ Zugriffe,

für alle anderen Operationen

$\underline{2 + \log_k (n/k^*)}$ Zugriffe.

Beispiel:

$$n = 1.000.000$$
$$k^* = 5$$
$$k = 50$$

$$\Downarrow$$

$$2 + \log_{50}(200.000) \leq 6$$

Für eine hashed Datei wären es ≅ 3 Zugriffe gewesen.

Der B*-Baum ist also besser als eine Ein-Level Index Struktur.

Der Vorteil gegenüber Hashing ist, daß die **Datei immer sortiert** vorliegt.

Dateien mit einem dense Index

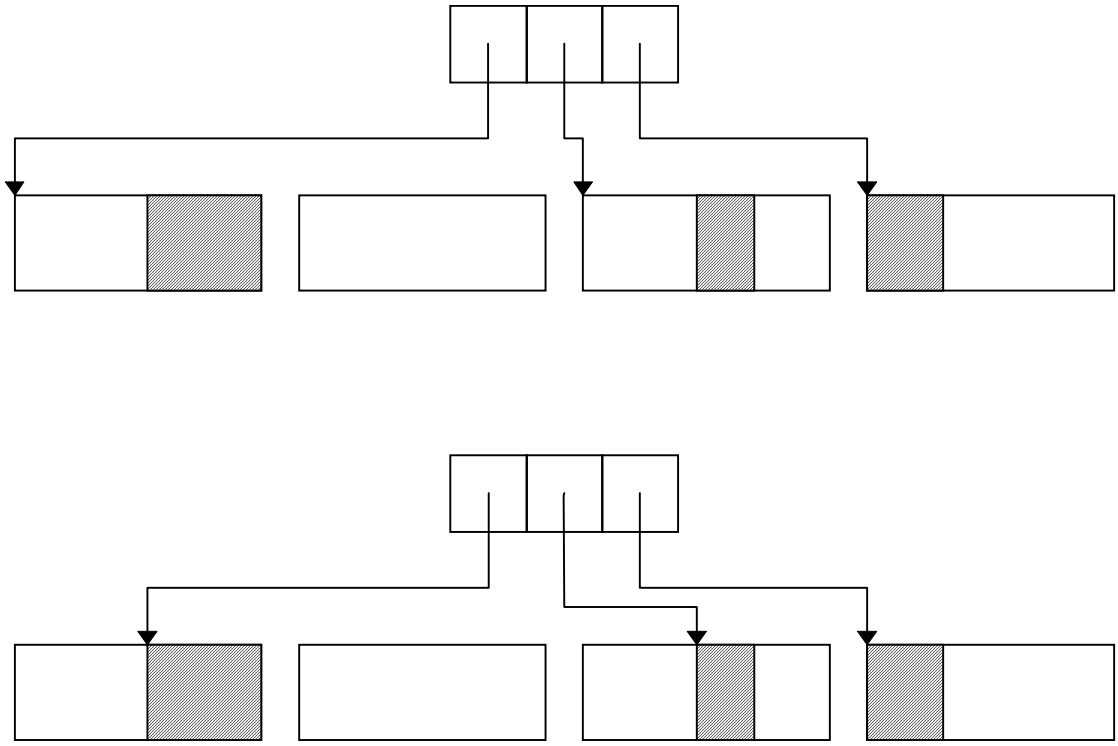
Wenn die Hauptdatei nicht sortiert vorliegen muß, dann

- kann man teilgefüllte Blöcke in der Hauptdatei vermeiden.
- kann man eine einfache Einfüge-Strategie anwenden: immer am Ende einfügen.

Für die dann beim Löschen auftretenden „Löcher“ in der Hauptdatei kann man zwei Strategien wählen:

- Man ignoriert die Tatsache und lebt mit den Löchern, oder

- Man hält eine separate Datei mit Zeigern auf die Blöcke mit leeren Subblöcken, oder sogar direkt auf die leeren Subblöcke:



Dadurch werden aber keine Blockzugriffe eingespart, es wird nur der freie Platz besser verwaltet.

Wenn die Hauptdatei unsortiert vorliegt,

- wie findet man einen Satz?

→ Dense Index

Ein Dense Index ist eine Datei mit einem Satz der Form (v,p) für **jeden** Schlüsselwert v in der Hauptdatei.

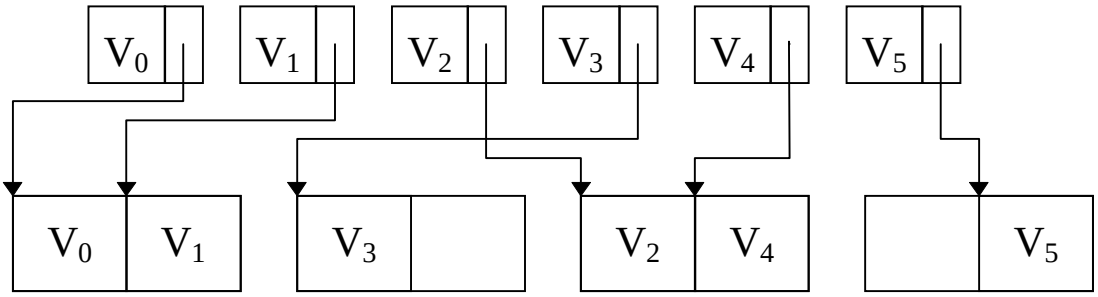
Ein Dense Index kann bei den bisher besprochenen Verfahren anstelle der Hauptdatei verwendet werden.

Der Dense Index kann also als

- Hash-Datei
- Index
- B*-Baum

organisiert sein.

Suchen (Lookup)



- Bestimmen des Blockes der Hauptdatei.
- Lesen des Blocks.
- evtl. Ändern/Zurückschreiben des Blocks.
  Modifikation

Löschen

- Löschen des Blockeintrages.
- Zurückschreiben des Blocks.
- Löschen des Indexeintrags.

Einfügen

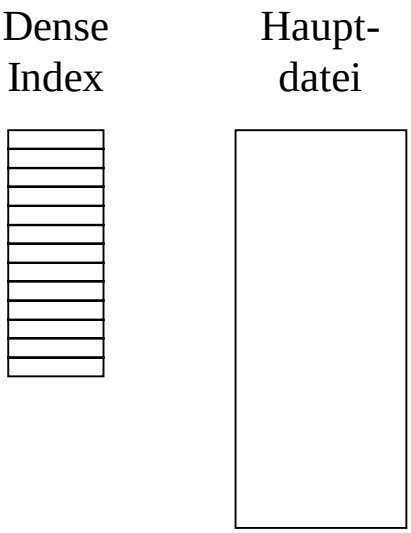
- Einfügen eines Satzes am Ende der Hauptdatei (evtl. In einem neuen Block).

- Einfügen eines entsprechenden Eintrags im Index.

Anmerkung: Durch die zusätzlichen Zugriffe auf die Hauptdatei werden immer 2 Zugriffe mehr benötigt als wenn die Organisation des Dense Index direkt auf die Hauptdatei angewendet würde.

Wozu Dense Index?

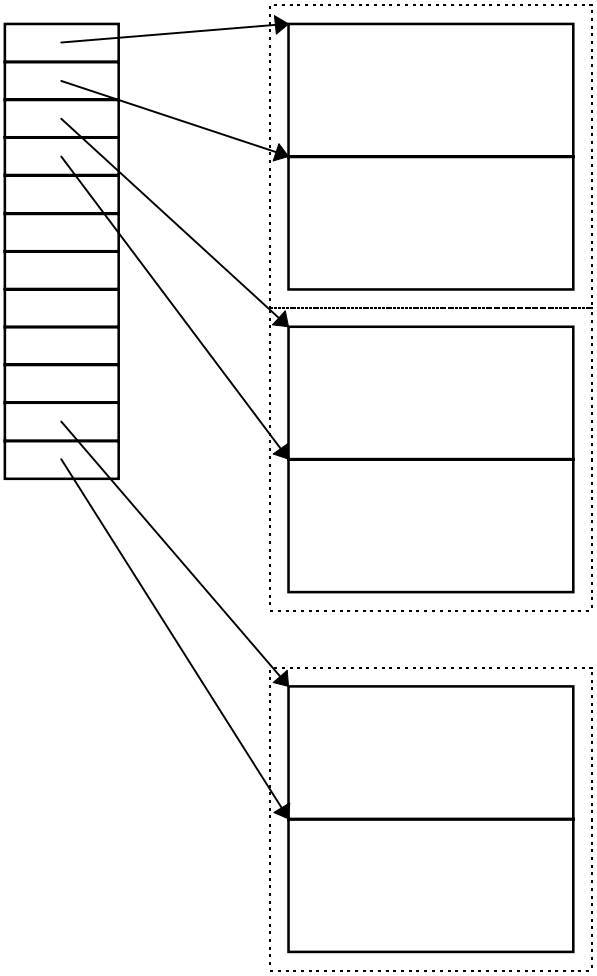
Wenn jede Operation über den Dense Index grundsätzlich 2 Zugriffe mehr benötigt als ohne Dense Index, muß der Einsatz eines Dense Index begründet werden.



Gründe für einen Dense Index:

1. Die Sätze in der Hauptdatei sind evtl. pinned, Die Sätze im Dense Index hingegen nicht.

   → Es kann eine einfachere oder effizientere Organisationsform für den Dense Index gewählt werden.

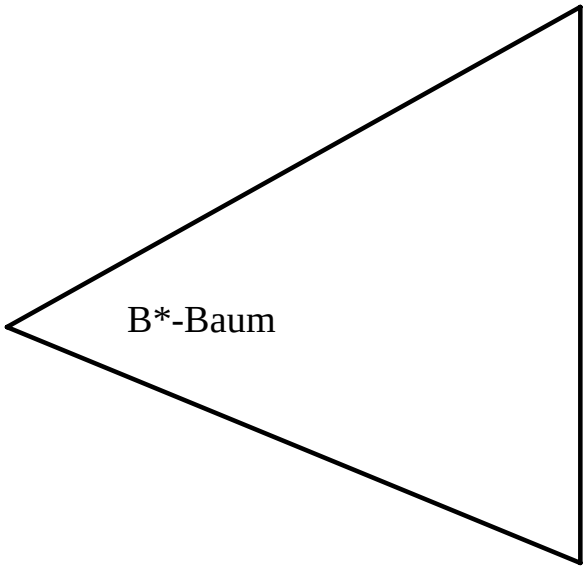


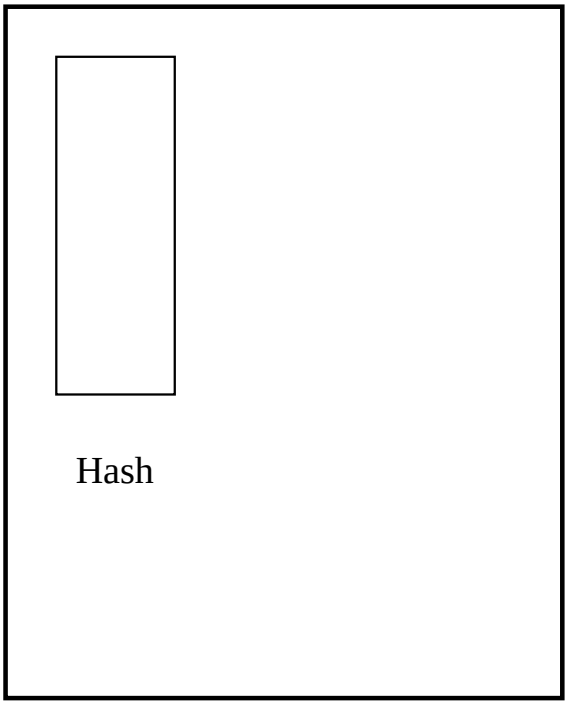
Dense Index

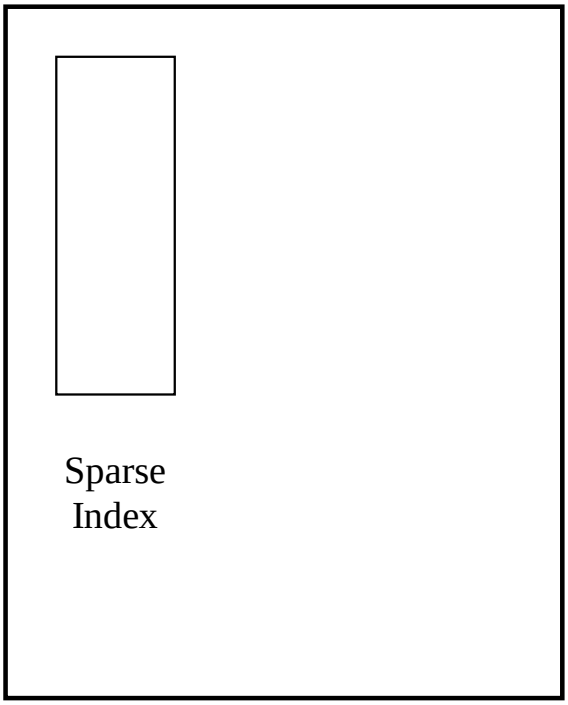
unpinned Records
sortiert

Hauptdatei

pinned Records
unsortiert

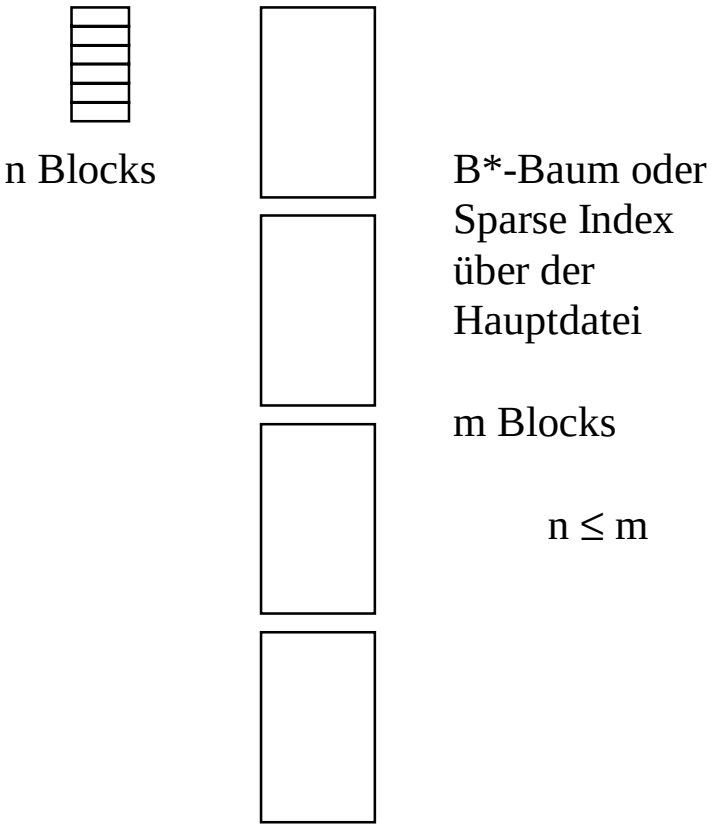

B*-Baum


Hash

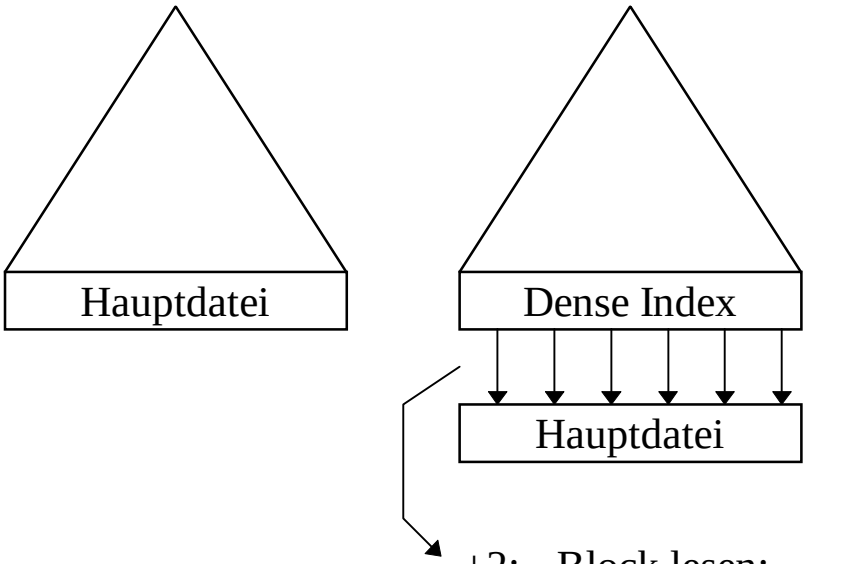

Sparse
Index

2. Falls die Sätze der Hauptdatei sehr groß sind, wird die Anzahl der Blocks, die für einen Dense Index benötigt werden, viel kleiner sein, als wenn ein Sparse Index oder ein B*-Baum auf der Hauptdatei anwendet würde.

   Gleiches gilt für einen Zugriff per Hashing, auch hier kann die durchschnittliche Anzahl der Blocks pro Bucket geringer ausfallen, wenn über den Dense Index statt über der Hauptdatei gehashed wird.



B*-Baum vs. Dense Index mit B*-Baum



Beispiel:

Hauptdatei mit

n = 1.000.000 Sätzen

B*-Baum mit

k* = 5

k = 50

B*-Baum über der Hauptdatei:

$$2 + \log_k (n / k^*) =$$
$$2 + \log_{50}(200.000) \leq 6$$

B*-Baum über Dense Index:

- Größe der Dense Index Record = Größe der Knoten des B*-Baumes. → k* = 50

$$2 + \log_k (n / k^*) =$$
$$2 + \log_{50}\left(\frac{1.000.000}{50}\right) \leq 5$$

Es müssen hierzu noch die 2 Zusätzlichen Zugriffe auf die Hauptdatei hinzugezählt werden:

2+5 = 7

→ Es werden mehr Zugriffe benötigt als für einen B*-Baum über der Hauptdatei.

Aber ...

Kompensations Faktoren

Es gibt zwei Faktoren die den Nachteil der zusätzlichen Zugriffe kompensieren:

1. Platzersparnis

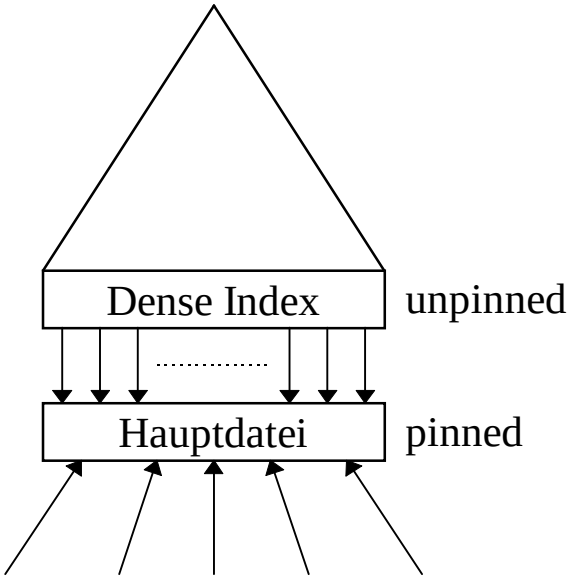
   Die Blöcke der Hauptdatei können jetzt immer dicht gepackt werden. In einer B*-Baum Organisation wären sie dagegen zwischen halb und ganz gefüllt. Platzersparnis 25% bei der Hauptdatei.

   Der Platz der für die Blätter des B*-Baumes beim Dense Index benötigt wird ist ca. 10% des Platzes der Hauptdatei.

   Die reale Ersparnis beträgt also ca. 25% - 10% = **15%**

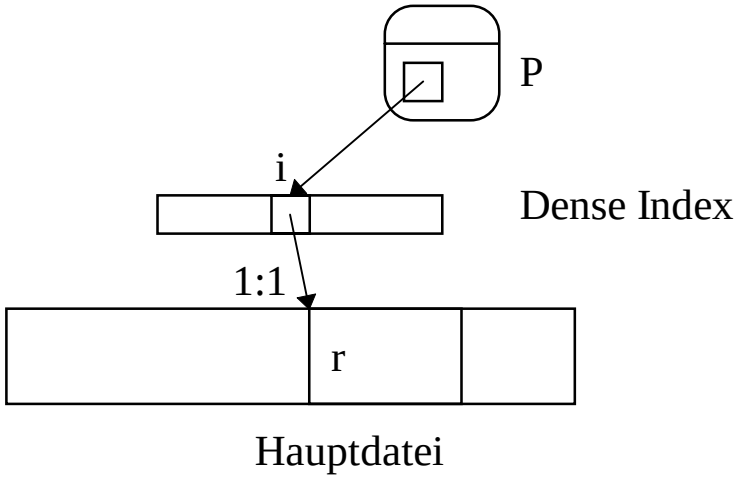
2. Falls die Sätze der Hauptdatei **pinned down** sind, kann die Organisationsform B*-Baum **nicht** benutzt werden.

   Dies kann durch benutzen eines Dense Index gelöst werden.



Methoden zum Unpinning von Sätzen

Eine andere Verwendung des Dense Index ist es, die Sätze der Hauptdatei unpinned zu machen.



(-) Es muß 2 Zeigern zum Satz r gefolgt werden.

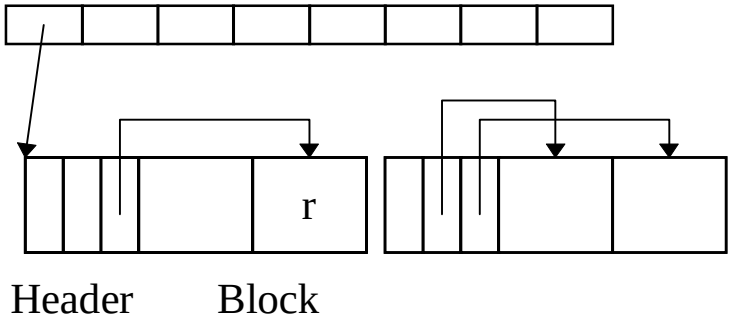
(+) Sätze der Hauptdatei sind nicht pinned.

(-) Sätze des Dense Index sind pinned.

(+) Beim Verschieben eines Satzes in der Hauptdatei muß nur ein Zeiger verändert werden.

Alternative Methoden zum Unpinnen von Sätzen

- Kein Dense Index für die Hauptdatei, sondern Zeiger in jedem Blockheader auf die Sätze in dem Block.



Alle Zeiger auf den Satz r zeigen nun auf den Block, der r enthält.

- Sätze können nicht zwischen Blocks ausgetauscht werden.

- Sätze sind innerhalb eines Blocks unpinned und daher frei beweglich.

Kosten des Verfahrens:

- Der Platz, der für die Zeiger im Blockheader verbraucht wird.

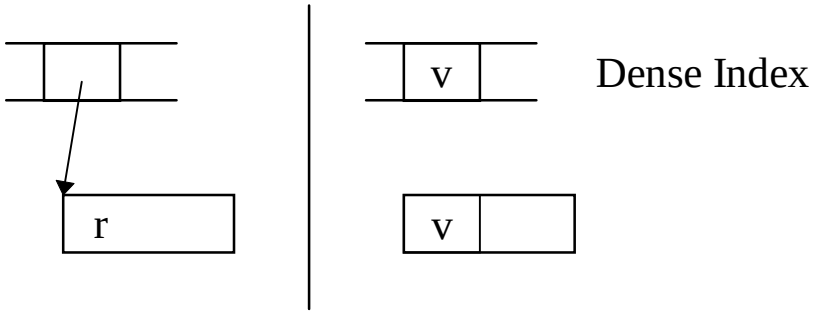
- Die Zeit, die benötigt wird dem zusätzlichen Zeiger innerhalb des Blocks zu folgen ist nicht relevant, da kein weiterer Blockzugriff erforderlich wird.

- **System R** benutzt dieses Verfahren für Sätze mit variabler Länge.

- Eine Generalisierung des Verfahrens ist es auf das Bucket eines Satzes zu zeigen statt auf den Satz direkt. (Hashing)

Eine weitere Möglichkeit ist es,

- die Schlüsselwerte anstelle von Zeigern als Referenz zu benutzen.



IBM nutzt dieses Verfahren bei der **IMS**-Datenbank.

(+) Dense Index und Haupdatei sind unpinned.

(-) Um einer Referenz zu folgen muß nach dem Schlüsselwert gesucht werden.