Informationssysteme

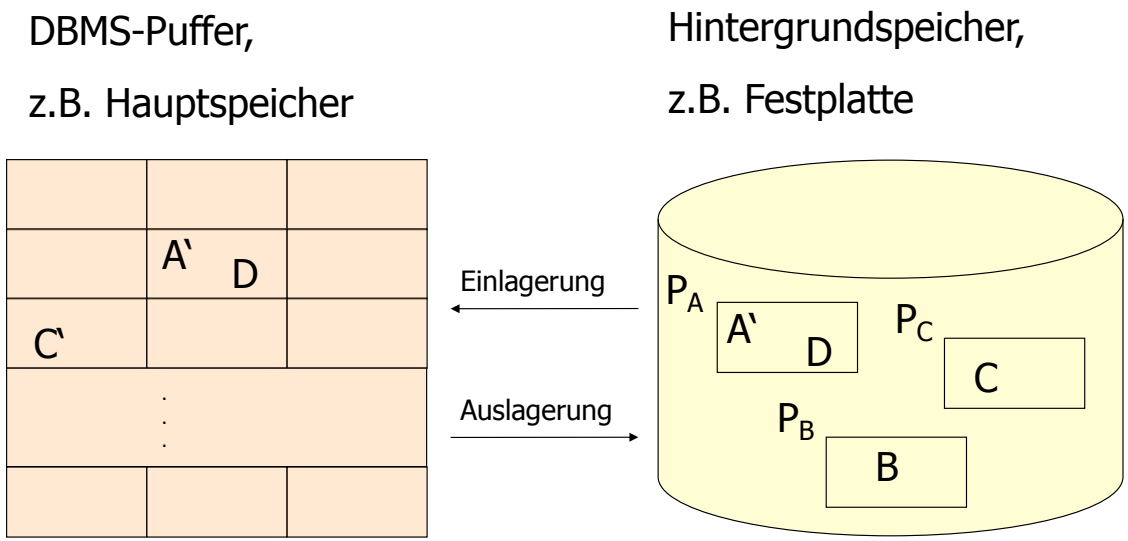
Sommersemester 2016

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Datenbank-Pufferverwaltung

- **Motivation**: Bereits erwähnte Zugriffslücke zwischen Hauptspeicher und Festplatte (Externspeicher)
- **Idee**: Halte Seiten, auf die zugegriffen wurde, in einem Puffer im Hauptspeicher
- Dieser Puffer kann durchaus sehr groß sein (hunderte Megabyte oder viele Gigabytes). Trotzdem viel kleiner als DB selbst.



Datenbank-Pufferverwaltung (2)

5 Minuten Regel

“Pages referenced every five minutes should be memory resident.”

Siehe Papier von Jim Gray & Franco Putzolu:

`http://www.hpl.hp.com/techreports/tandem/TR-86.1.pdf`

Empfehlung zum Thema Datenbank-Pufferverwaltung: Das Buch von Härder und Rahm: “Datenbanksysteme - Konzept und Techniken der Implementierung” ist hier sehr ausführlich.

Ersetzungsstrategien: Konzept und Klassifizierung

Wenn eine Seite nicht im Puffer auffindbar ist wird sie in Puffer eingetragen. Falls dieser bereits voll ist, muss eine andere Seite weichen, aber welche?

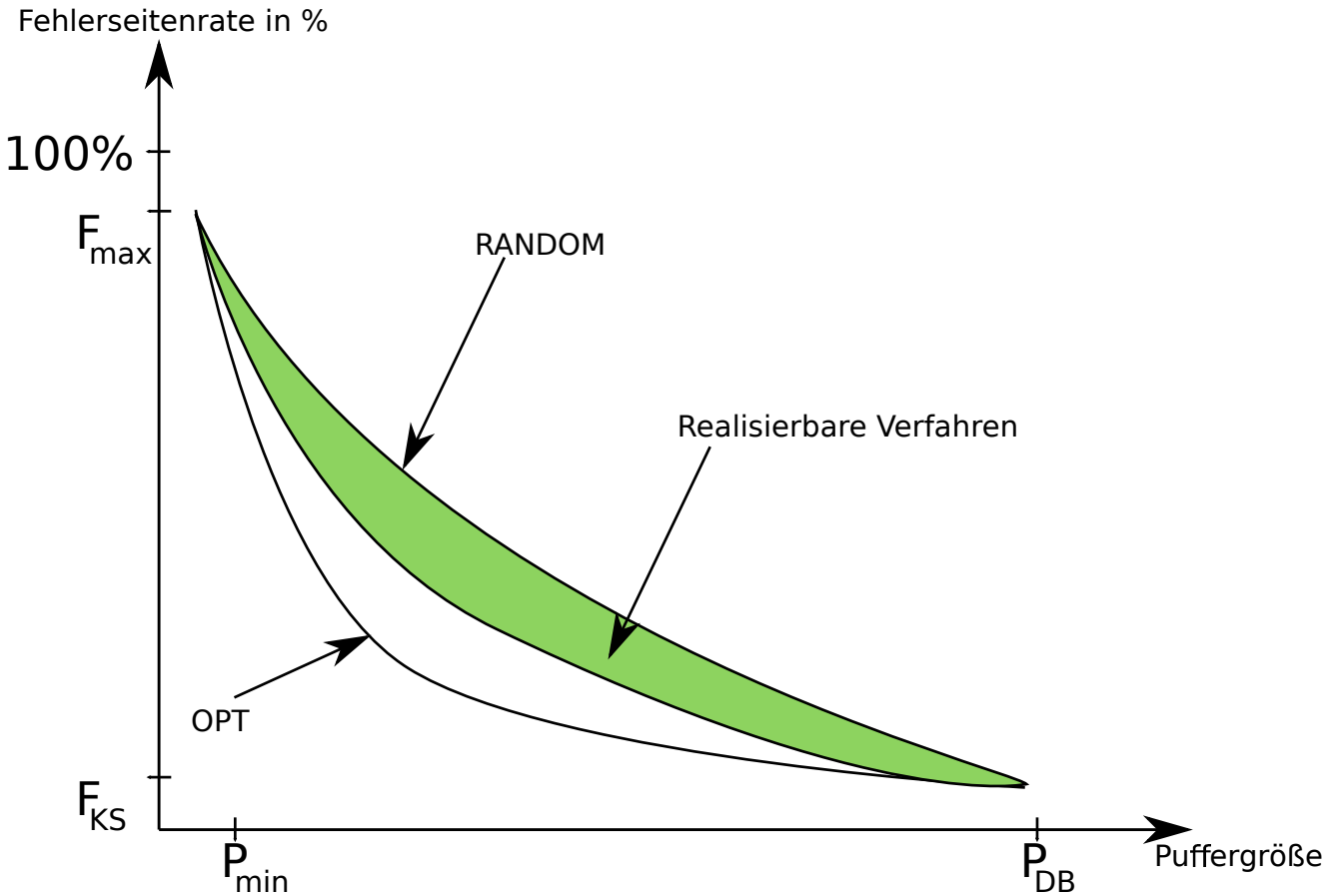
Klassifizierung von **Strategien** anhand ...

- ob das **Alter seit Einlagerung**, seit letztem Zugriff oder überhaupt nicht, und
- ob alle **Referenzen**, die letzte Referenz oder keine

bei der Auswahlentscheidung (welche Seite ersetzt werden soll) zum Tragen kommt.

Erste (triviale) Idee: Zufälliges Ersetzen von Seiten (RANDOM Strategie).

Optimales vs. Zufälliges vs. Realisierbare Verfahren



P_{min}= minimale Größe des DB-Puffers
P_{DB} = Datenbankgröße
F_{KS} = Fehlerseitenrate bei Kaltstart

FIFO

First-In, First-Out

- Ersetzt diejenige Seite, die am längsten im Puffer ist

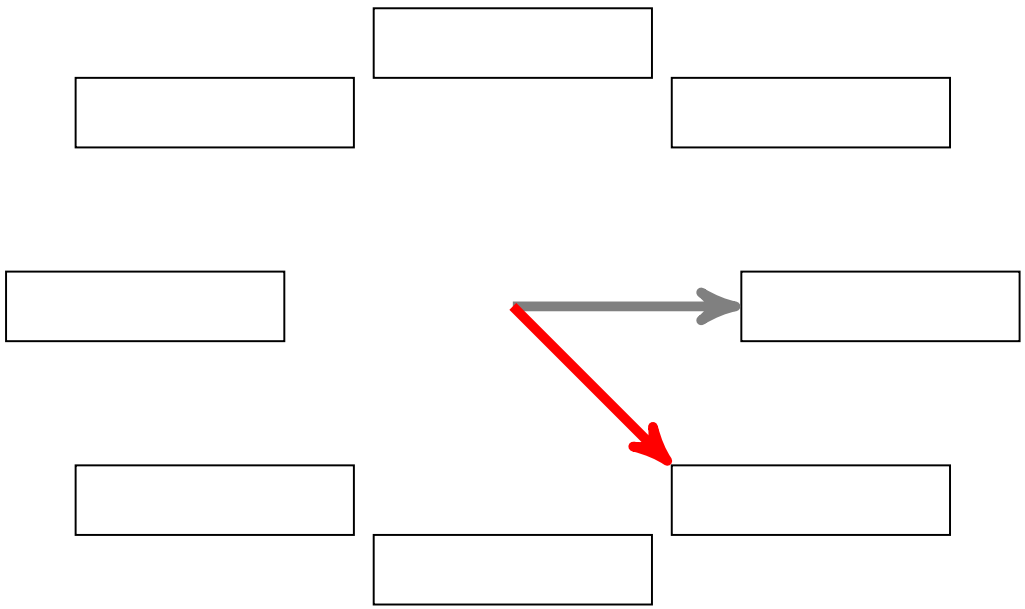


Illustration: Kreisförmige Anordnung. Zeiger verweist auf den älteste Eintrag. Grauer Zeiger= alt. Roter Zeiger=neu.

Least-Frequently-Used (LFU)

- **Ersetzt Seite mit der geringsten Referenz- (Zugriffs-) Häufigkeit**
- Für jede Seite wird ein **Zugriffszähler** (aka. Referenzzähler RZ) geführt.
- **Die Seite mit dem kleinsten Zählerstand wird ersetzt.**

Seiten, auf die kurzzeitig sehr sehr oft zugegriffen wurde bleiben sehr lange im Puffer, auch wenn nicht mehr wirklich benutzt.

Alter des Zugriffs (bzw. der letzten Zugriffe) wird nicht berücksichtigt.

- "Problem" kann durch periodisches Herabsetzen der Referenzzähler adressiert werden.

Least-Recently-Used (LRU)

- **Ersetzung basierend auf Zeit seit dem letzten Zugriff auf Seite.**
- Halte Seiten in Form eines Stacks:
  - Eine Seite kommt bei jeder Referenz auf oberste Position
  - Seite auf der untersten Position des Stacks wird bei Bedarf ersetzt

Beispiel:

Zugriff (in dieser Reihenfolge) auf Seiten:
A, B, C, D, A, C, A, B, B, B, C, D, A. Puffer hat Platz für 3 Seiten. Seite E soll eingelagert werden. Welche Seite muss weichen?

Beobachtung:

Was passiert, wenn in einer Leseoperation von der Festplatte viele neue Seiten gelesen werden?
Wieso ist das problematisch?

Theoretische Sicht

Wir betrachten einen **String von Referenzen auf Seiten**

$$r_1, r_2, \ldots, r_t, \ldots$$

wobei $r_t = p$ bedeutet, dass auf Seite p zugegriffen wurde.
Zu einem Zeitpunkt t nehmen wir an, dass jede **Seite eine gewisse Wahrscheinlichkeit b_p besitzt als nächstes Aufgerufen zu werden**, d.h. $Pr(r_{t+1} = p) = b_p$.

Interarrival Time

- **Wie viele Zugriffe liegen zwischen den Zugriffen auf eine Seite p?** Genau b_p^{-1}.

Interarrival Time Annäherung durch LRU

- Interarrival Time: Zwischen zwei Zugriffen auf Seite p liegen b_p^{-1} Zugriffe.

- **Diejenigen Seiten mit kleinster Interarrival Time bzw. größter Wahrscheinlichkeit b_p sollten im Puffer gehalten werden.**
- Dies wird **bei LRU approximiert** durch die Seiten mit dem Zeitpunkt des letzten Zugriffs auf Seite.

Weitere Ersetzungsstrategien werden in der VL Datenbanksysteme im Wintersemester vorgestellt.

Indexe

Überlegungen zu Performance

Beobachtung

- Je schneller die Anfrage berechnet wird, desto besser.
- Daten passen nicht in den Hauptspeicher.
- Es liegen Größenordnungen zwischen Performanz des Hauptspeichers und der Festplatte.
- Zugriffe sind unglaublich teuer!
- Insbesondere die wahlfreien (random access).

Was muss getan werden?

- Effiziente Speicherung auf Festplatte
- Wie findet man die gesuchten Daten möglichst schnell?
- Welche Garantien können bzgl. "Laufzeit" gegeben werden?

Grundidee eines Index

Abbildung: Schlüssel $\rightarrow$ Menge von Einträgen

Beispiele:

- MatrikelNummer $\rightarrow$ persönliche Daten des Studenten
- PLZ $\rightarrow$ Name und andere Informationen einer Stadt
- Term $\rightarrow$ alle Dokumente in denen dieser Term enthalten ist

Natürlich möchte man bei diesen häufig auftretenden Anfragen die Antwortzeit gering halten.

Dafür wird ein Index angelegt: Ein Index materialisiert diese Abbildung!

Was ist mit der binären Suche?

- Idee: Sortiertes Array, dann binär suchen
  - $O(n\ log\ n)$ Kosten für das Sortieren
  - was passiert, wenn ein neuer Datensatz eingefügt werden muss?
- Binärer Suchbaum, am besten balanciert (AVL oder rot-schwarz Baum)

Aber....

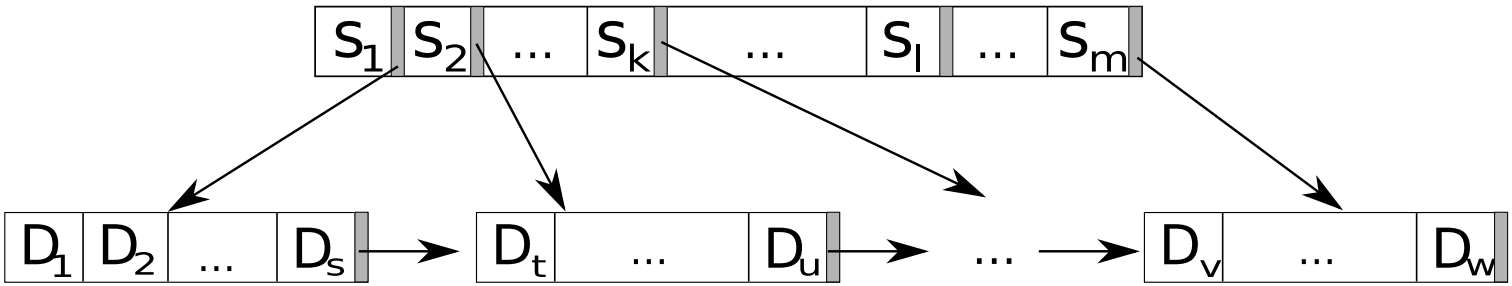
- binäre Suchbäume sind für DBMS ungeeignet
  - zu hoher Speicherverbrauch
  - schwer abzubilden auf Externspeicher
  - zu viele Cache-Misses (warum?)

$\Rightarrow$ viel zu langsam!

ISAM

Index-Sequential Access Method (ISAM)

Besteht aus Schlüsseln S_j und Datensätzen D_i.



- Sowohl Schlüssel als auch Daten werden geordnet abgespeichert.
- **Suche:**
  - Binäre Suche in Schlüsseln zur gewünschten Position
  - Sequentielles Lesen in Datenseiten
- **Einfügen:**
  - Auffinden der Einfügeposition (wie bei Suche)
  - Was passiert wenn die Seite, in die eingefügt werden soll, voll ist? Nicht gut ...
- **Löschen:**
  - Auffinden der Löschposition (wie bei Suche und Einfügen)
  - Löschen des Datensatzes. Was passiert wenn die Seite leer wird?

Bäume

Beobachtung

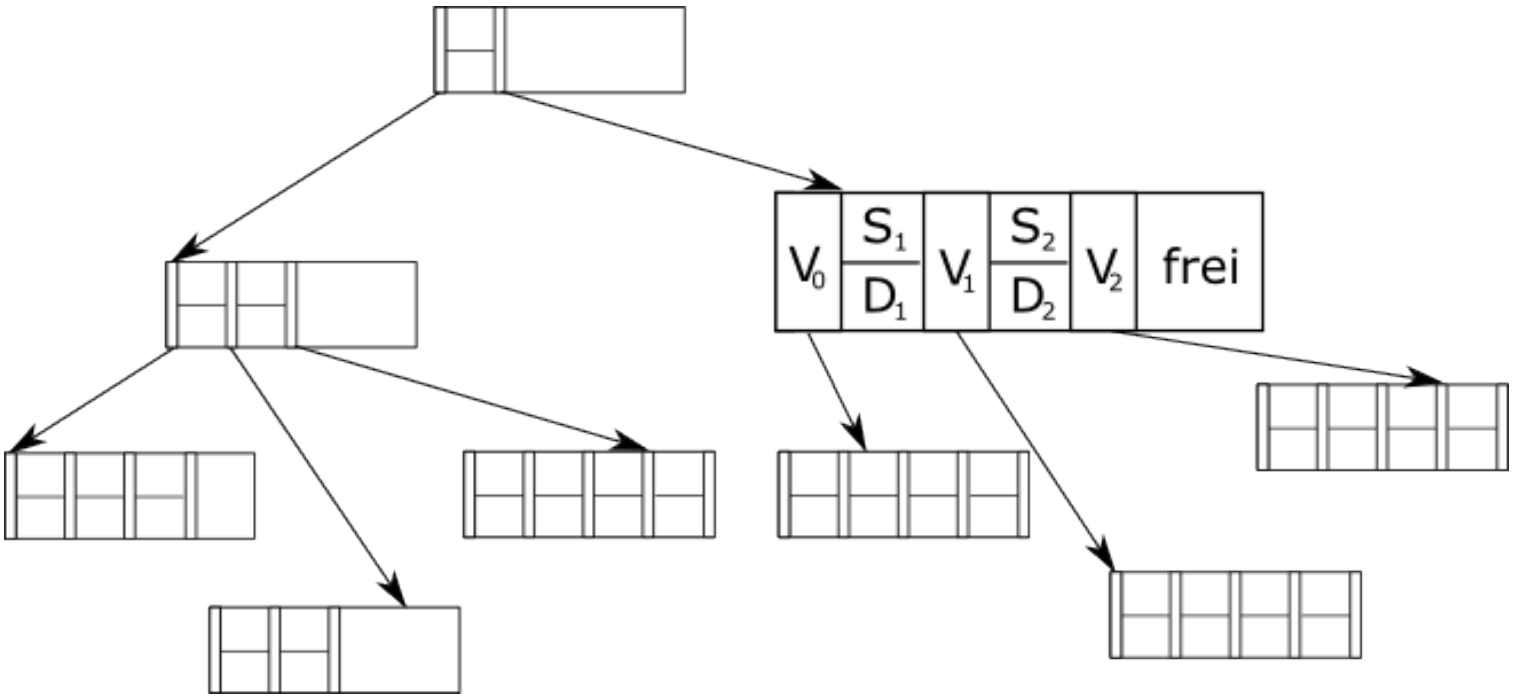
- Binäre Bäume nicht optimal für Festplatten
- Wichtig: Anpassung der Kapazität der Knoten an Größe der Seiten.
- Warum?

Fanout

- Je breiter der Baum desto weniger Tief.
- Je flacher desto weniger "Sprünge" zwischen Knoten

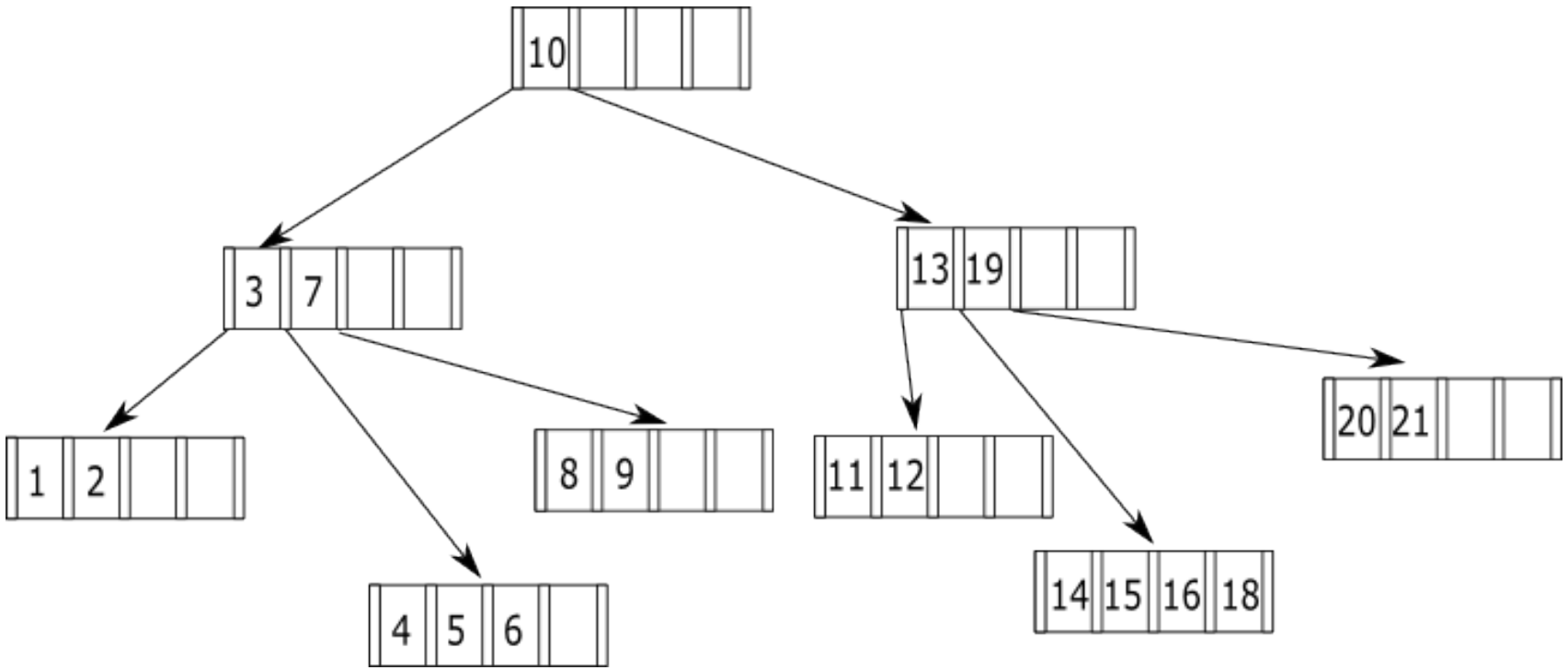
$\Rightarrow$ Weniger Zugriffe auf Seiten auf der Festplatte.

B Baum



- Verweise V_j
- Schlüssel S_i
- Datensätze D_k

B Baum Beispiel



Schlüssel in den inneren Knoten zwei Funktionen haben. Sie identifizieren Daten(sätze), und sie dienen als Wegweiser in der Baumstruktur.

Einfügen von Schlüsseln

1. **Führe eine Suche nach dem Schlüssel durch**; diese endet (scheitert) an der Einfügestelle
2. **Füge den Schlüssel dort ein.**
3. **Ist der Knoten überfüllt, teile ihn:**
   - Erzeuge einen neuen Knoten und belege ihn mit den Einträgen des überfüllten Knotens, deren Schlüssel größer ist als der des mittleren Eintrags.
   - Füge den mittleren Eintrag im Vaterknoten des überfüllten Knotens ein.
   - Verbinden den Verweis rechts des neuen Eintrags im Vaterknoten mit dem neuen Knoten
4. **Ist der Vaterknoten jetzt überfüllt?**
   - Handelt es sich um die Wurzel, so lege eine neue Wurzel an
   - Wiederhole Schritt 3 mit dem Vaterknoten

B Baum: Eigenschaften

- Balanciert

Verbesserung?

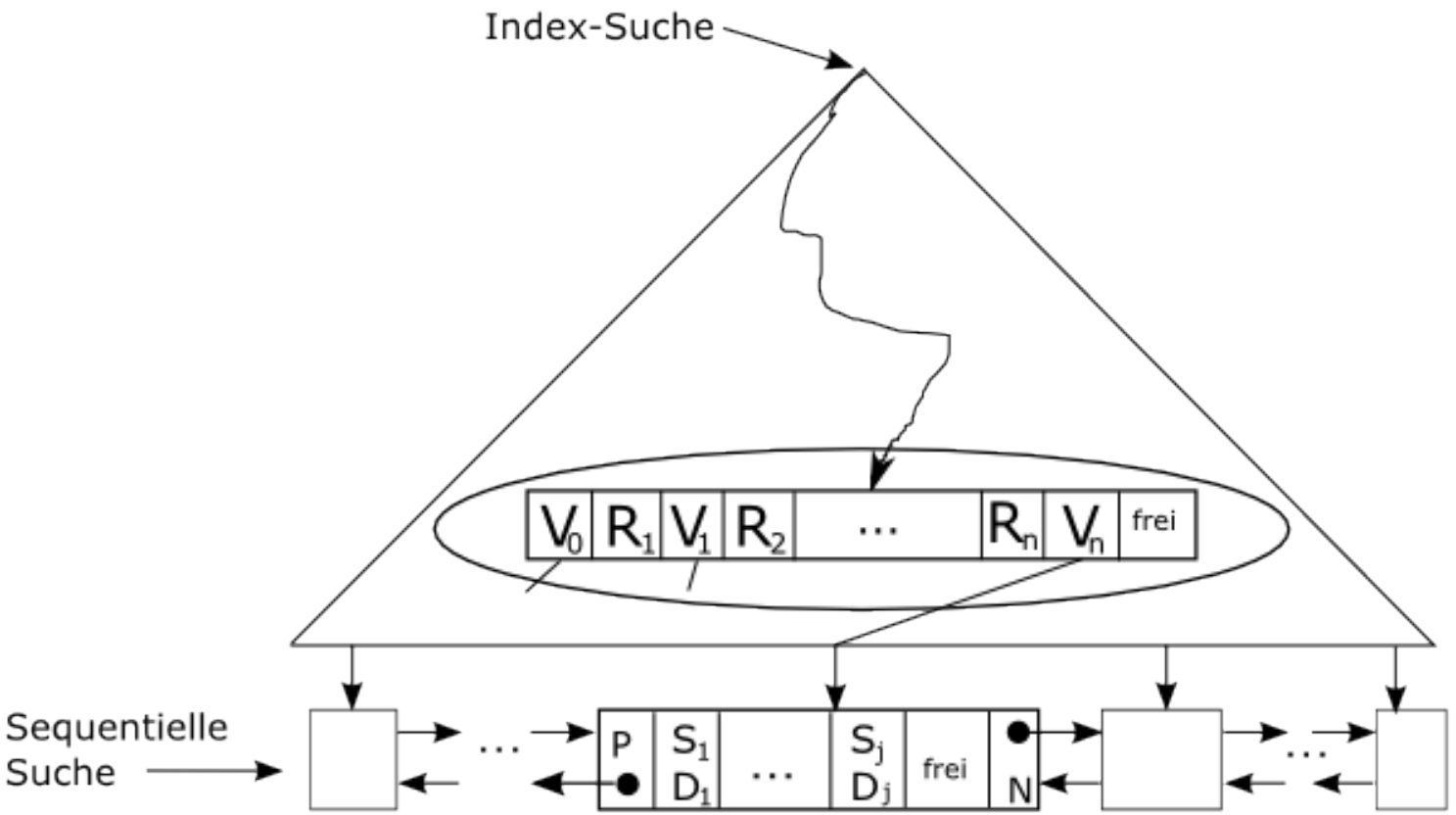
- Wie kann der Fanout erhöht werden?
- Platz in den Knoten einsparen.
- Also: Keine Daten in den Knoten (nur in der Wurzel), nur Verweise

$\Rightarrow$ B+ Baum

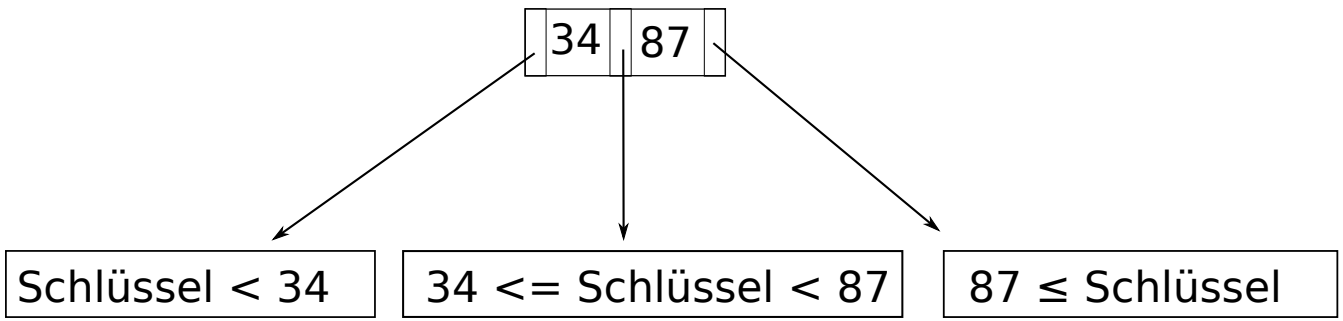
Schöne Beschreibung von Prof. Härder zu B und B+ Bäumen:
`http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/courses/ss2007/Informationssysteme/addons/mehrwegbaeume.full.pdf`

B+ Baum

- "Hohler" Baum: **Daten nur in den Blättern**
- Suche muss also immer bis zu den Blättern laufen
- Aufbau: Referenzschlüssel R_j, Schlüssel S_k, Daten D_i, Zeiger V_m
- **Blattknoten sequentiell verbunden!**

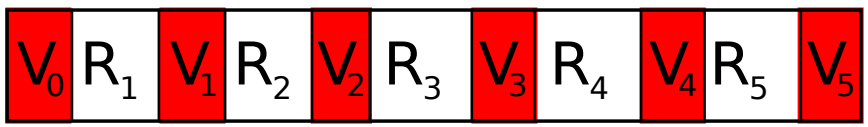


B+ Baum: Verweise



B+ Baum: Knotenstruktur

Die Stuktur eines inneren Knotens:



In diesem Beispiel hat der Knoten 6 Verweise auf Kindknoten, d.h. der sogenannte **Fanout** ist 6.

B+ Baum: Eigenschaften

Ein B+ Baum vom Typ (k,k^*) hat folgende Eigenschaften:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge
2. Jeder Knoten - außer Wurzeln und Blättern- hat mindestens k und höchstens $2k$ Einträge. Blätter haben mindestens k^* und höchstens $2k^*$ Einträge. Wurzel hat entweder maximal $2k$ oder sie ist ein Blatt mit maximal $2k^*$ Einträgen.
3. Jeder Knoten mit n Einträgen, außer den Blättern, hat $n+1$ Kinder.

B+ Baum: Eigenschaften

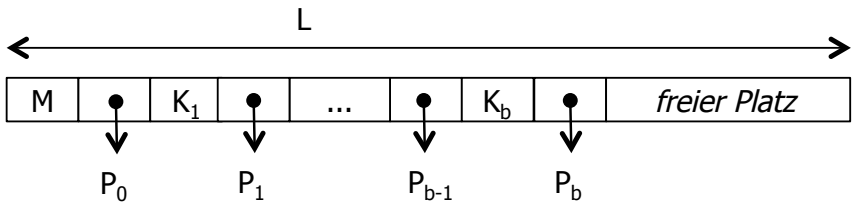
4. Seien $R_1, ..., R_n$ die Referenzschlüssel eines inneren Knotens (d.h. auch der Wurzel) mit $n+1$ Kindern. Seien $V_0, V_1, ..., V_n$ die Verweise auf diese Kinder.
   (a) V_0 verweist auf den Teilbaum der Schlüssel kleiner R_1
   (b) V_i $(i = 1,..., n-1)$ verweist auf einen Teilbaum, dessen Schlüssel zwischen R_i und R_{i+1} liegen (einschließlich R_i).
   (c) V_n verweist auf den Teilbaum mit Schlüsseln größer gleich R_n.

B+ Baum: Knotenformate

M: Kennung des Seitentyps + Anzahl Einträge; feste Länge L

Innere Knoten

- $k \leq b \leq 2k$



Blattknoten

- $k^* \leq m \leq 2k^*$
- Zeiger P_{prior} und P_{next} dienen dazu die Blattknoten linear durchsuchbar zu machen.

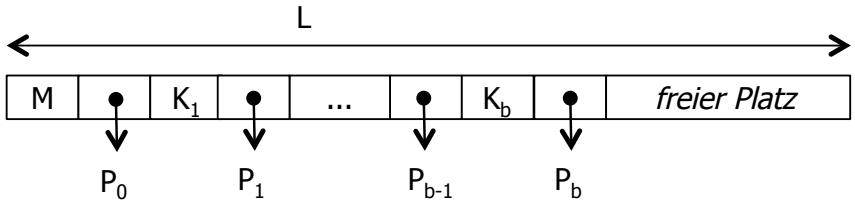
B+ Baum: Knotenformate (2)

M: Kennung des Seitentyps + Anzahl Einträge; feste Länge L, Einträge für Schlüssel (k), Daten (D) und Verweise (p)

Innere Knoten

- $k \leq b \leq 2k$
- $L = l_M + l_p + 2k(l_k + l_p)$

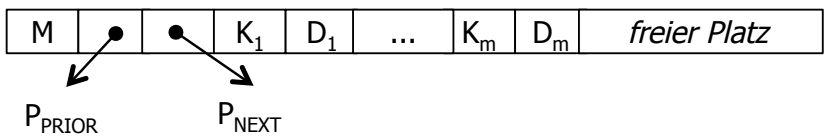
$$k = \lfloor \frac{L - l_M - l - p}{2 \times (l_k + l_p} \rfloor$$



Blattknoten

- $k^* \leq m \leq 2k^*$
- $L = l_M + 2l_p + 2k^*(l_k + l_D)$

$$k^* = \lfloor \frac{L - l_M - 2l_p}{2 \times (l_K + l_D} \rfloor$$



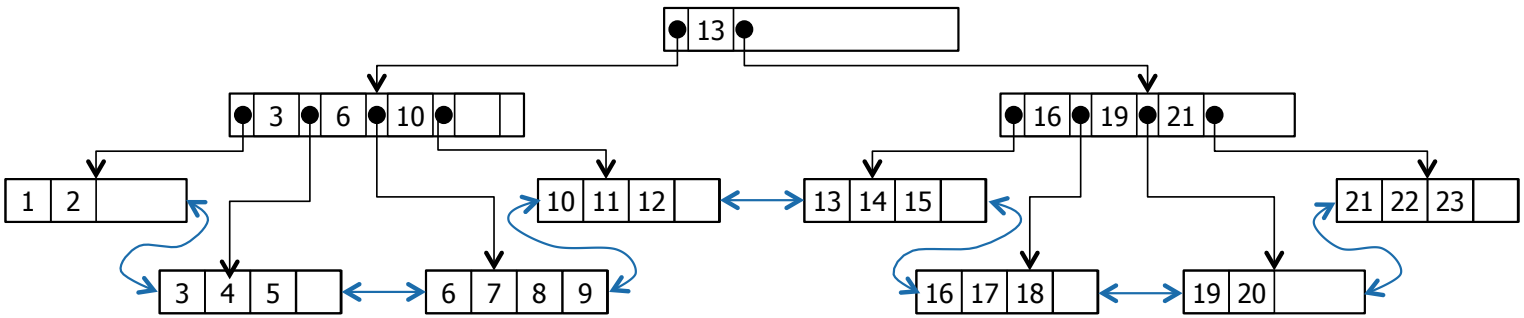
Typische Größen (in Byte):

$L = 8192$, $l_M = 4$, $l_p = 4$, $l_K = 8$, $l_D = 88$

Innere Knoten: $k = 341$;

$Blatt : k^* = 42$

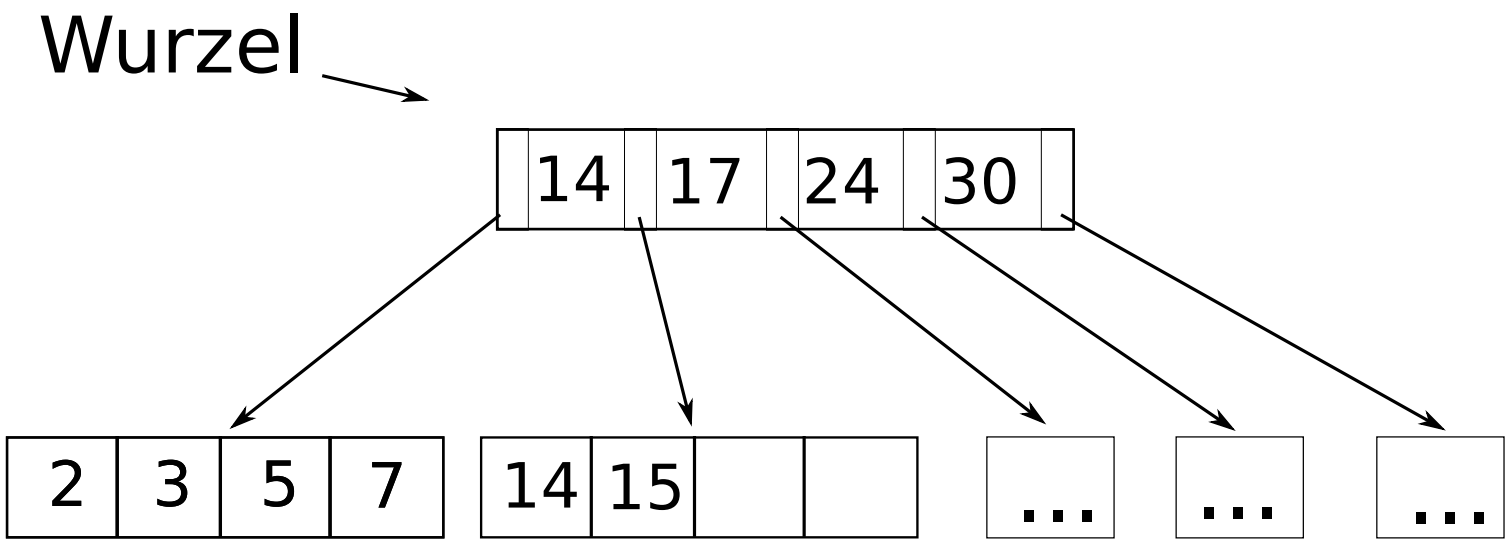
B+ Baum



- Jeder Knoten hat die Größe eines I/O Blocks.
- Ein Knoten ist zu mindestens 50% gefüllt.
- Das Lesen eines Knotens verursacht typischerweise einen wahlfreien Zugriff auf Festplatte
- Ein B+ Baum ist i.d.R. sehr flach, d.h. Suche verursacht nur wenige wahlfreie Zugriffe.
- Zudem, die ersten 1-2 Ebenen des Baums sind i.d.R. im Hauptspeicher "gecached".

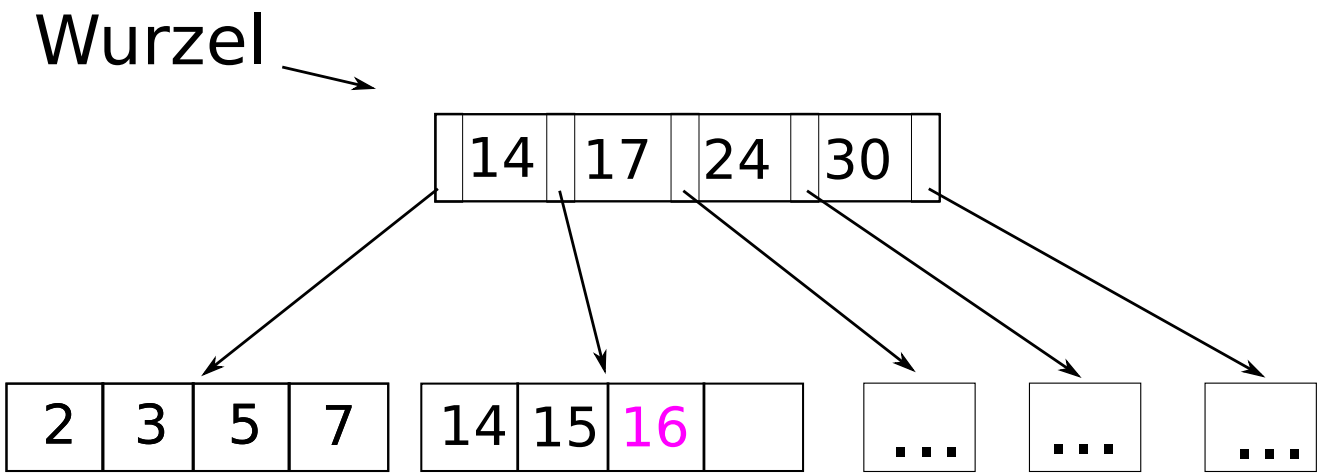
B+ Baum: Komplexeres Einfügebeispiel

Einfügen von 16 und 8



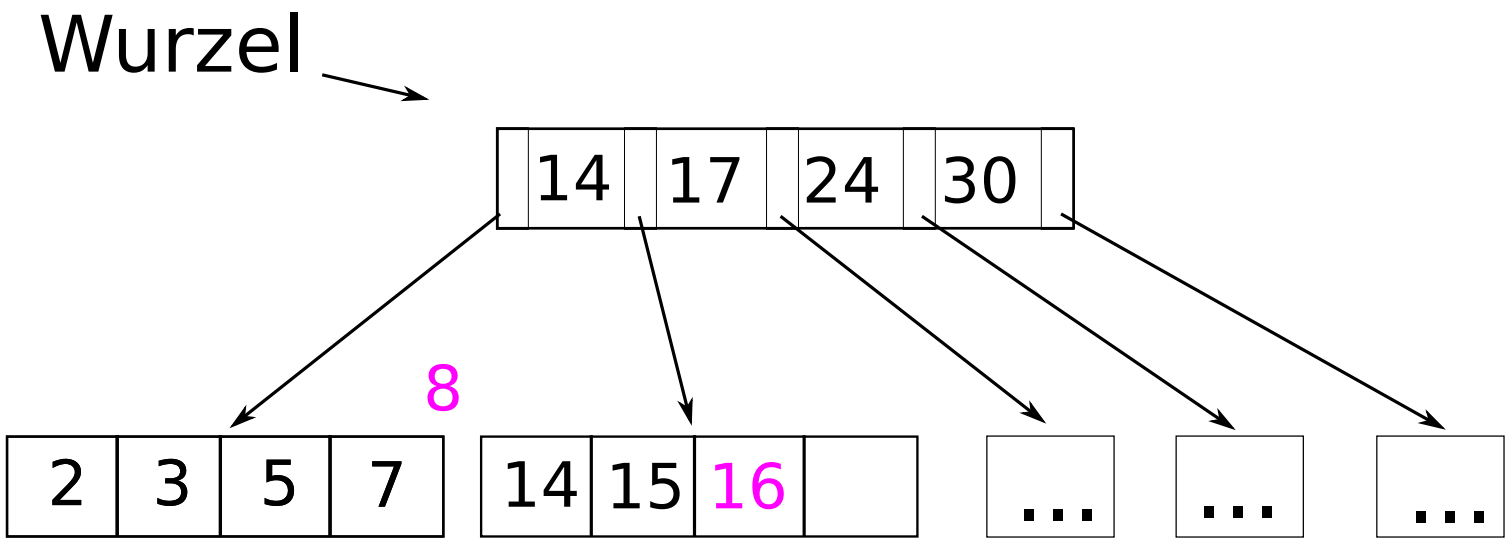
B+ Baum: Komplexeres Einfügebeispiel

Einfügen von 16 und 8



B+ Baum: Komplexeres Einfügebeispiel

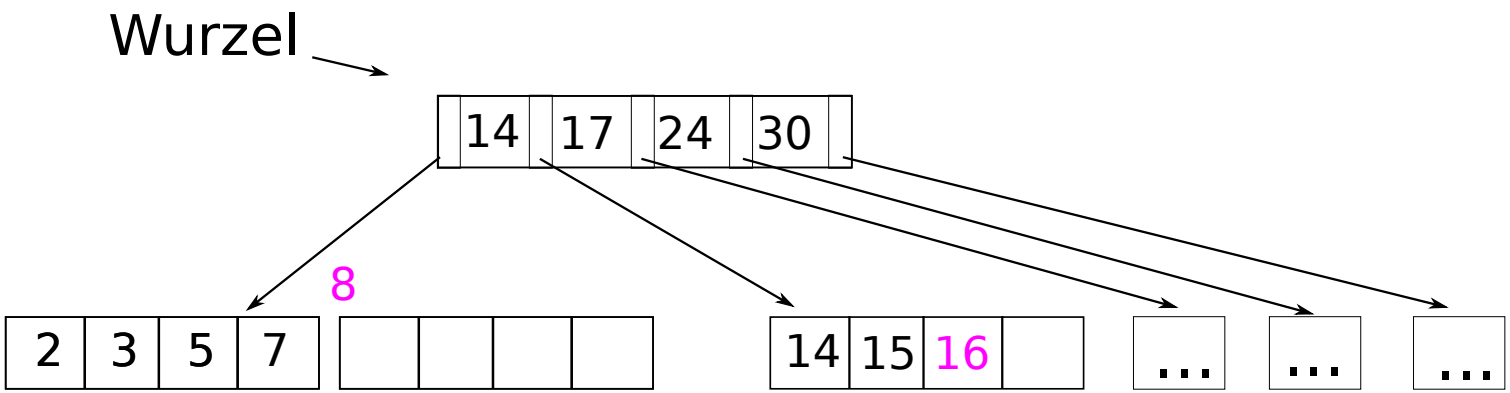
Einfügen von 16 und 8



Zum Einfügen der 8 in den Blattknoten ist kein Platz vorhanden.

B+ Baum: Komplexeres Einfügebeispiel

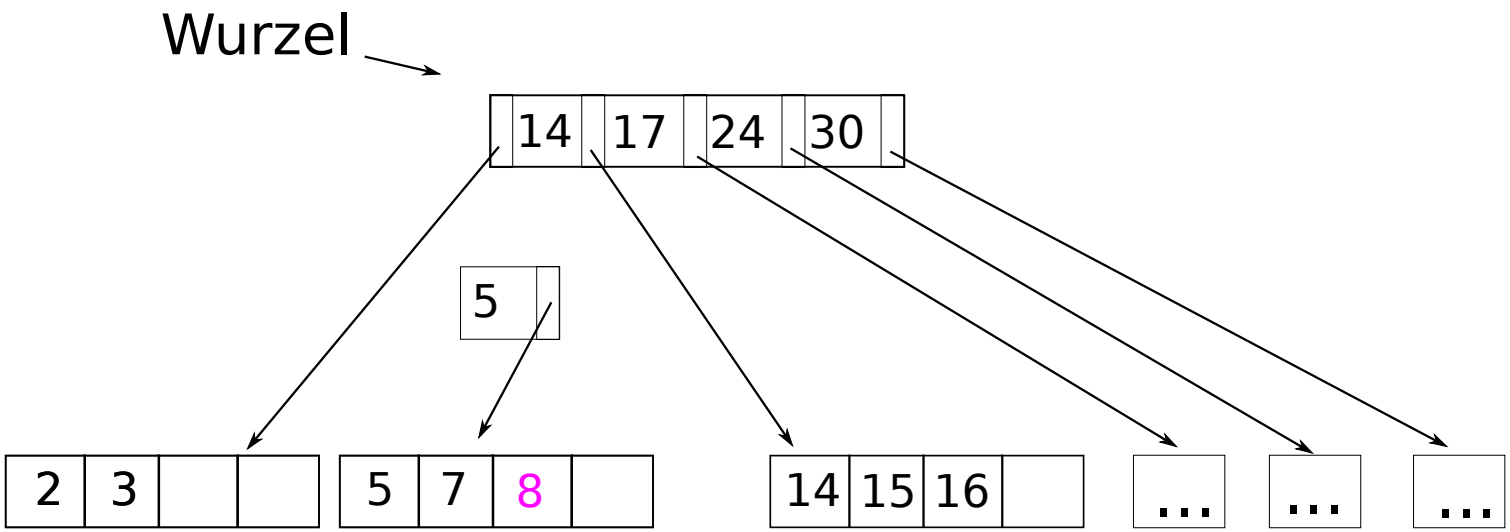
Einfügen von 16 und 8



Es wird also ein neuer Blattknoten erzeugt.

B+ Baum: Komplexeres Einfügebeispiel

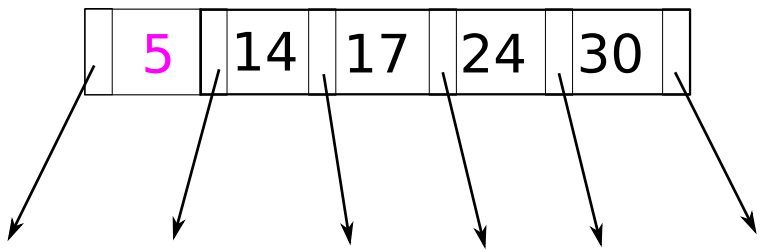
Einfügen von 16 und 8



... und der mittlere Wert nach oben kopiert.

B+ Baum - Komplexeres Einfügebeispiel

Einfügen von 16 und 8



Dieser innere Knoten ist nun voll und muss ebenfalls gesplittet werden. Dabei wird der mittlere Wert (17) in einen neuen inneren Knoten eingefügt.

B+ Baum: Komplexeres Einfügebeispiel

Einfügen von 16 und 8