Algorithmen und Datenstrukturen

Kapitel 10: B-Bäume

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Outline

1. B-Bäume
2. B-Bäume: Effizienz

ADS meets Datenbanken

00000001	Bruce Wayne
00000002	Muhammad Lee
00000003	Yann LeCun
00000004	Bilbo Baggins
00000005	The Big Lebowski
00000006	Aegon Targaryen
00000007	Whalter White
00000008	Loddamaddäus
...	...
10000000	Rudolf Bayer

In **Datenbanken** speichern wir große Mengen von Daten und wollen *(z.B. per SQL)* nach **bestimmten Werten suchen**.

Kleines Experiment (MySQL)

- **Tabelle** anlegen (**10 Mio. Einträge**).

```
CREATE TABLE STUDENTS ( MATRNR INT, NAME VARCHAR(90) );
```

- **Anfrage** an die Datenbank dauert **4.08 Sekunden** ☹

```
SELECT * FROM STUDENTS WHERE MATRNR=7654321;
```

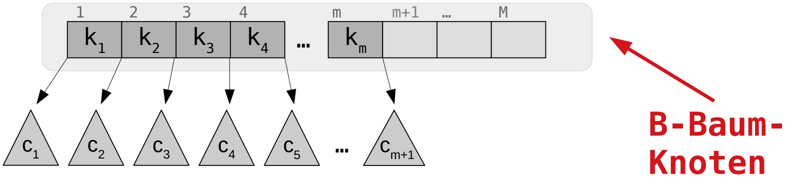
- **Index** anlegen → **500-facher Speed-up** ☺

```
CREATE INDEX MAGIC ON STUDENTS(MATRNR);
```

Wie realisieren Datenbanken solche Index-Strukturen? → B-Bäume

B-Bäume = "Generelle" 2-3-4-Bäume

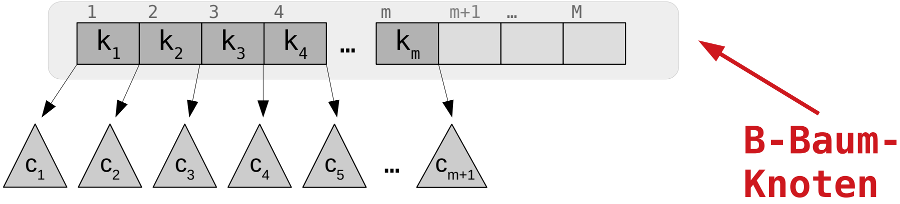
Im **2-3-4-Baum** gibt es **1 bis 3 Schlüssel** je Knoten.
Im **B-Baum** sind **M** *(z.B.* 1001*)* Schlüssel je Knoten erlaubt.



Knoten in B-Bäumen

- Ein Knoten enthält m **aufsteigende Schlüssel** $k_1, \ldots, k_m$.
- m beträgt **maximal** M und **mindestens** $\lfloor M/2 \rfloor$.
- Ausnahme: **Wurzel** (enthält mindestens **einen** Wert).
- M ist üblicher Weise **ungerade**.

B-Bäume = “Generelle” 2-3-4-Bäume

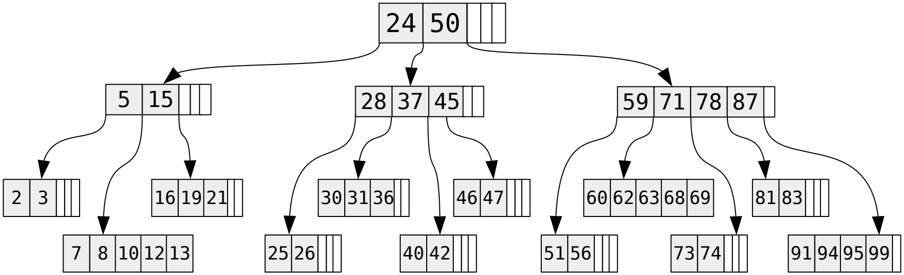


Knoten in B-Bäumen

Zwischen den Schlüsseln befinden sich **Kinder/Subbäume** $c_1, \ldots, c_{m+1}$:

- **Subbaum** c_1 enthält nur Schlüssel $< k_1$.
- **Subbaum** $\mathrm{c}_{\mathrm{m}+1}$ enthält nur Schlüssel $> k_m$.
- **Alle anderen** c_i enthalten nur Schlüssel <u>zwischen</u> k_{i-1} und k_i.

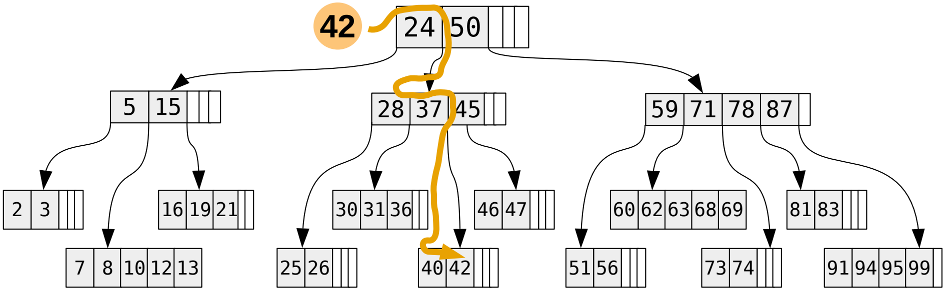
B-Bäume: Beispiel



Eigenschaften dieses B-Baums

- $M = 5$: Maximaler **Branch-Faktor** $M+1 = 6$.
- Jeder Knoten enthält mindestens $\lfloor M/2 \rfloor = 2$ **Schlüssel**.
- Der Baum ist **deutlich flacher** als ein Binärbaum.

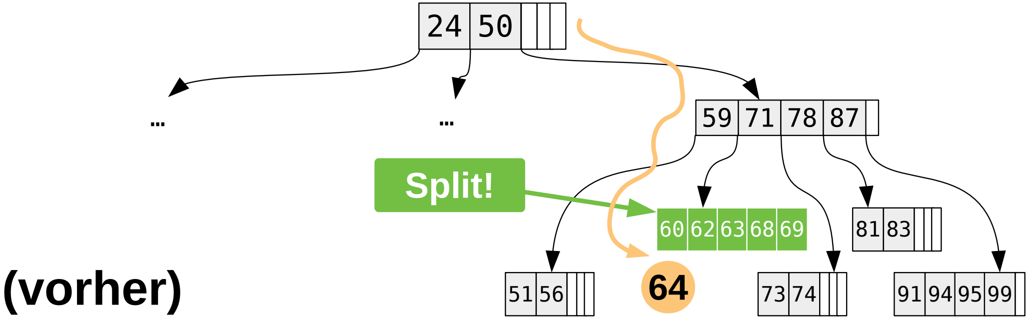
B-Bäume: Suche



Suche: Wie im 2-3-4-Baum

- Beginne bei der **Wurzel**, **durchsuche** aktuellen Knoten *(linear oder **binär**)*.
- Schlüssel **gefunden** → zurückgeben.
- Schlüssel **nicht gefunden** → gehe zu Kind.

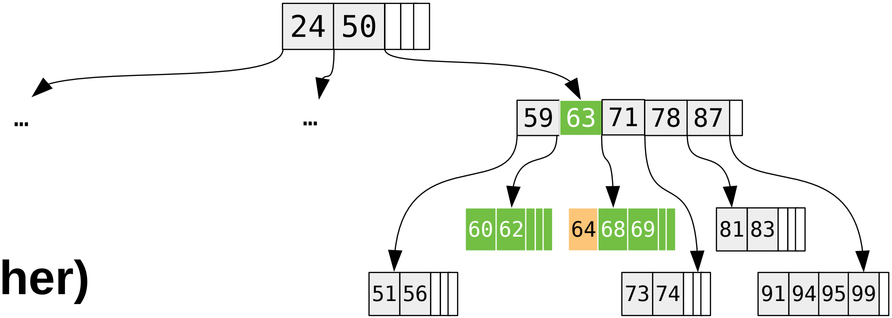
B-Bäume: Einfügen



Einfügen: Wie im 2-3-4-Baum

- Suche passendes **Blatt** zum Einfügen.
- Wenn Schlüssel bereits **vorhanden**: Überschreiben.
- **Verschiebe** die anderen Schlüssel und mache Platz für neuen.
- **Splitte volle Knoten**!

B-Bäume: Einfügen



(nachher)

Beispiel: Füge 64 ein

... **Splitte** volle Knoten!

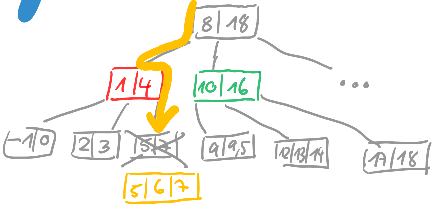
- Der mittlerer Schlüssel (63) wandert nach oben.
- Die ersten $\lfloor M/2 \rfloor$ Schlüssel (60,62) bleiben im gleichen Knoten.
- Die restlichen $\lfloor M/2 \rfloor$ Schlüssel (68,69) kommen in neuen Knoten.

Do-B-Baum-Einfügen-Yourself (M=5)

Füge 6 ein!

Knoten = Arrays!

Split



„proaktives" Splitting

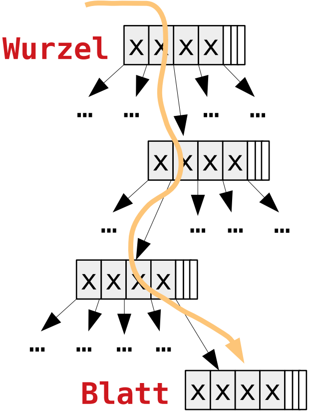
Do-B-Baum-Einfügen-Yourself

Outline

1. B-Bäume

2. B-Bäume: Effizienz

B-Bäume: Effizienz



Suche in B-Bäumen: Aufwand

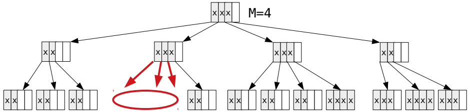
- Gegeben: B-Baum mit **n Schlüsseln**.
- Wie teuer ist die Suche im **Worst Case**?

Aufwand je Knoten × *Höhe des Baums*

Faktor 1: Aufwand je Knoten

- **Durchsuchen** eines Knotens nach passendem Schlüssel.
- Im Knoten befinden sich maximal M Schlüssel.
- **Binäre Suche** → $O(log\ M)$.
- In der Praxis ist M zwar **groß** *(z.B. 1001)*, aber **konstant**! Denn M wächst **nicht** mit der Schlüsselzahl n.

Faktor 2: Höhe des Baums



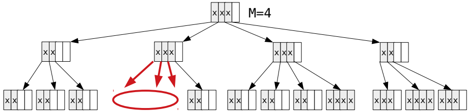
Wie im 2-3-4-Baum befinden sich im **B-Baum** **alle Blätter auf derselben Ebene**!

Höhe von B-Bäumen

Bei **maximaler Füllung** *(M Schlüssel je Knoten)* und Höhe h beträgt die **Anzahl der Schlüssel** im Baum:

Ebene		# Knoten	# Schlüssel
1		1	M
2	$\times(M+1)$	$M+1$	$M\cdot(M+1)$
3		$(M+1)^2$	$M\cdot(M+1)^2$
⋮			
h		$(M+1)^{h-1}$	$M\cdot(M+1)^{h-1}$

Faktor 2: Höhe des Baums (cont'd)



$$n = M \cdot \left(1 + (M+1) + (M+1)^2 + \dots + (M+1)^{h-1}\right) \quad // \text{geometr. Summe}$$

$$= \cancel{M} \cdot \left(\frac{(M+1)^h - 1}{\cancel{(M+1)} \cancel{-1}} \right)$$

$$n = (M+1)^h - 1$$

$$h = \log_{(M+1)}(n+1) \in O(\log_{M+1} n)$$

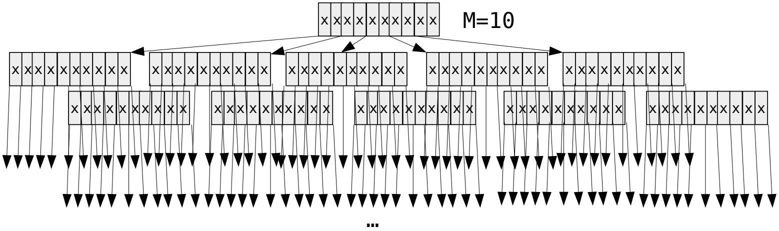
Suche im B-Baum

Gesamtkomplexität (voller Baum)

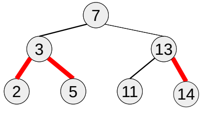
Aufwand je Knoten	$\times$	*Höhe des Baums*
$O(\log M)$	$\times$	$O(\log_{M+1} n)$

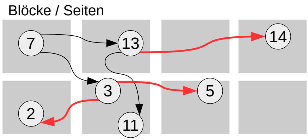
Anmerkungen

- Bei **großem M** ist die Baumhöhe sogar **quasi konstant**.
- **Beispiel**: $M = 101$, Höhe 5 → Platz für bis zu **> 10 Mrd. (!!!) Schlüssel!**



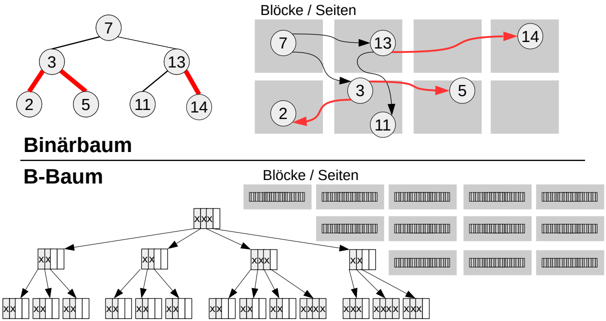
B-Bäume: Seitenoptimierung





- In **realen Rechnerarchitekturen** werden Daten per Block *(engl. "page)* aus dem **Hauptspeicher** geladen.
- Das Laden eines neuen Blocks ist **extrem zeitaufwändig** [1]!
- **Binärbäume** *(mit verstreuten Knoten)* sind hier ungünstig *(Laden vieler Blöcke)*.

B-Bäume: Seitenoptimierung



- **B-Bäume** laden mit einem Knoten viele Daten auf einmal *(da die Werte in einem Array nebeneinander stehen).*
- B-Bäume werden deshalb als **seiten-optimiert** bezeichnet.

References I

[1] Colin Scott.
Latency Numbers Every Programmer Should Know.
https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html (retrieved: Mar 2018).