Algorithmen und Datenstrukturen

– Wintersemester 2019 –

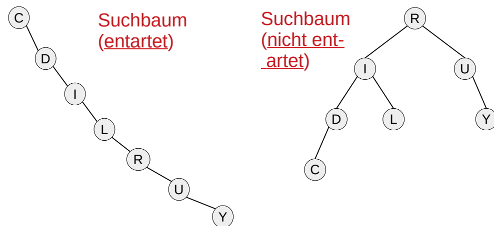
Kapitel 09: Balancierte Suchbäume

Fachbereich Informatik
TU Kaiserslautern

Dozent: Dr. Patrick Michel

Folien ursprünglich von Prof. Dr. Adrian Ulges (Hochschule RheinMain)

Effizienz von Suchbäumen



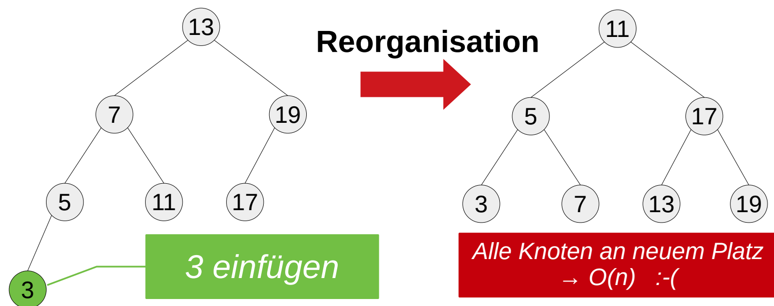
Problem: Entartung von Suchbäumen

- Bäume können quasi zu **verketteten Listen** “entarten”.
- Suchen, Einfügen, Löschen kosten im Worst Case $\Theta(n)$ ☹
- **Idee**: Baum sollte **Höhe** $O(log\ n)$ garantieren.

Balancierte/ausgeglichene Suchbäume

- 234-Bäume
- B-Bäume
- Red-Black-Trees

Reorganisation von Suchbäumen



Vollständiges Ausgleichen

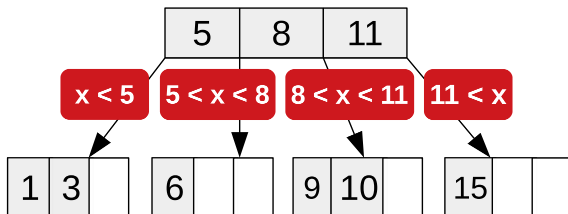
... müsste den Baum bei **jedem Einfügen** in einen **fast vollständigen** Baum transformieren.

- Im schlimmsten Fall Änderung **aller Knoten** $\rightarrow O(n)$ ☹
- Wir brauchen ein **unvollständiges Balancieren**.
- **Ziel**: Suchen $=$ Einfügen $=$ Löschen $= O(log\ n)$.

Outline

1. 2-3-4-Bäume
2. 2-3-4-Bäume: Effizienz
3. B-Bäume
4. B-Bäume: Effizienz

2-3-4-Bäume: Ansatz

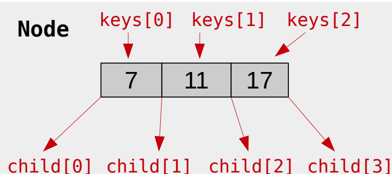


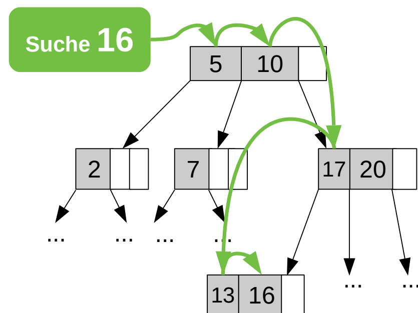
Idee: Ein Knoten enthält mehrere Schlüssel

- Ein Knoten besitzt **bis zu 3** *(aufsteigend sortierte)* Schlüssel und hat bis zu 4 Kinder.
- Schlüssel werden **von links** "aufgefüllt".
- **Allgemeinerer Suchbaum**: Subbaum zwischen A und B enthält nur Schlüssel zwischen A und B.

2-3-4-Bäume: Suche

```
function find(Node n, int key):

  // reached leaf -> not found
  if n == null:
    return false

  // check keys (from left to right)
  for i = 0,1,2:
    if (keys[i] == null or
        keys[i] > key):
      // go down -> recurse
      return find(n.child[i], key)

    else if key == keys[i]:
      return true

  // last chance to go down
  return find(n.child[3], key)

// start at root
find(root, 16)
```

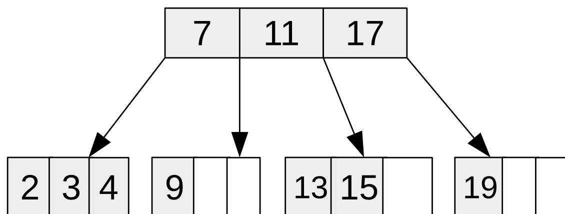




2-3-4-Bäume: Einfügen

Was ist das Problem beim Einfügen in den 2-3-4-Baum?

- Füge 16 ein ☺
- Füge 6 ein ☹



2-3-4-Bäume: Einfügen

Einfüge-Prozedur

Ansatz: An den **Blättern** einfügen, überschüssige Schlüssel **nach oben** durchreichen.

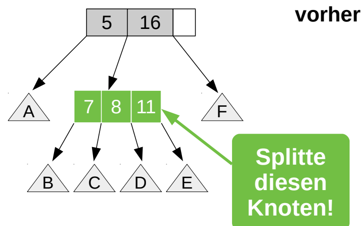
- Beginne an der **Wurzel**, wandere nach unten *(siehe "Suche")*.
- Schlüssel **gefunden**? → **Überschreiben**.
- **Blatt** erreicht? → Schlüssel **einfügen**.

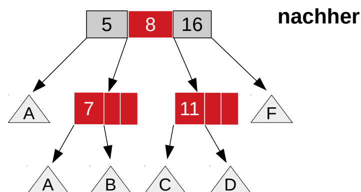
Reorganisation

- Wir müssen sicher sein, dass obere Knoten potenzielle **Überläufe** aus dem Blatt aufnehmen können.
- Deshalb **splitten** wir jeden vollen Knoten den wir treffen.

Definition "Split"

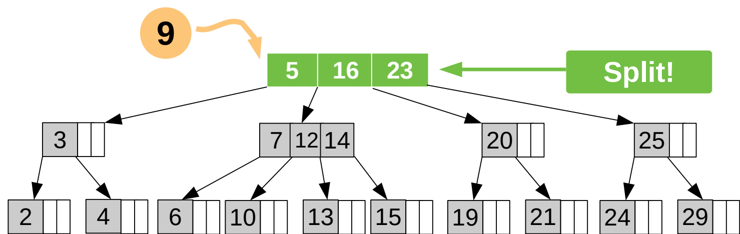
- Aus dem vollen Knoten werden **zwei neue**.
- Der mittlere Schlüssel wandert **nach oben** *(dort ist Platz, weil wir ja alle vollen Knoten oberhalb bereits gesplittet haben)*.

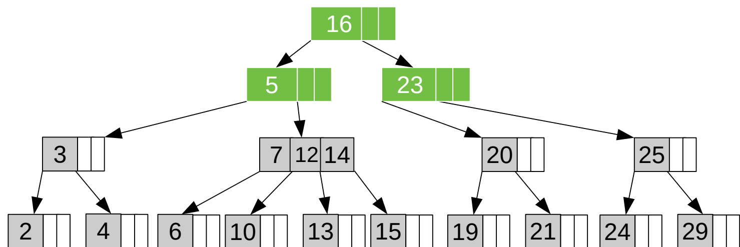




Beispiel: Füge 9 ein

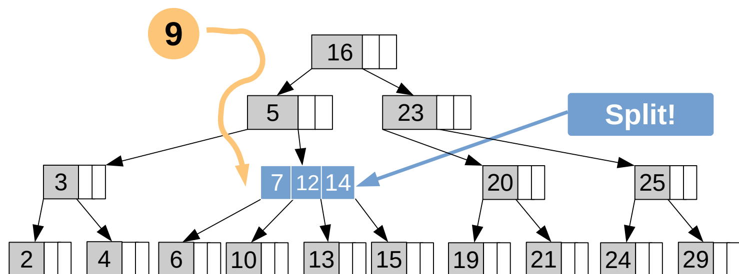
1. Wurzelknoten voll → split.

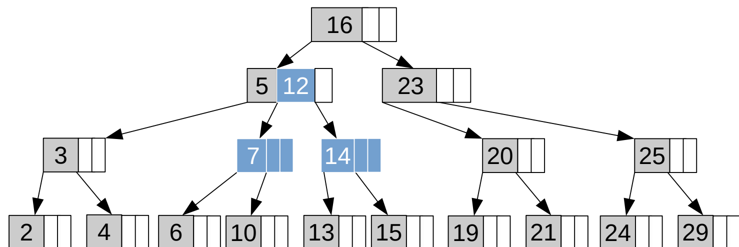




Beispiel: Füge 9 ein

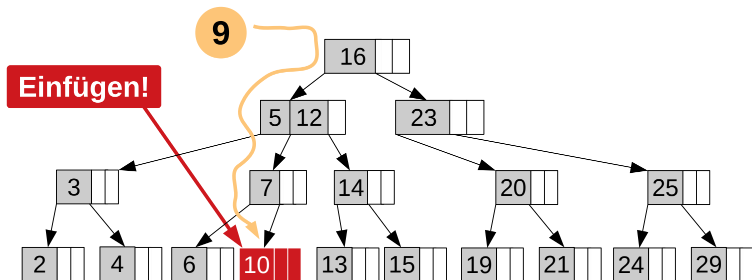
2. Knoten (7-12-14) voll → split.

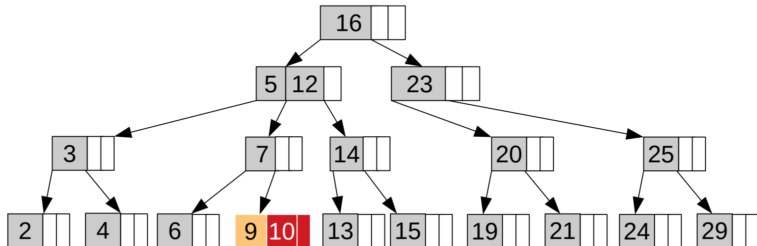




Beispiel: Füge 9 ein

3. Blatt erreicht: 9 einfügen.





Do-234-Yourself

Füge nacheinander die Schlüssel 1,2,...,10
in einen (anfangs leeren) 2-3-4-Baum ein!

Outline

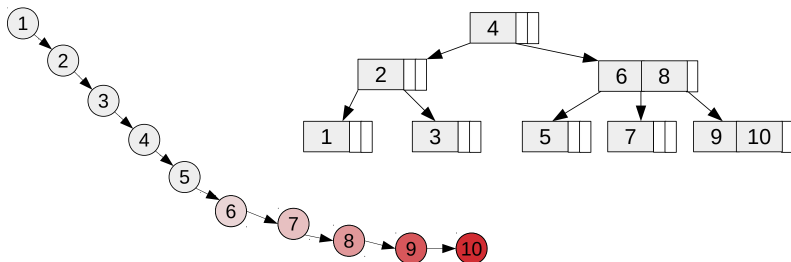
1. 2-3-4-Bäume

2. 2-3-4-Bäume: Effizienz

3. B-Bäume

4. B-Bäume: Effizienz

2-3-4-Bäume: Diskussion



2-3-4-Bäume sind ausgeglichen!

- Die **Höhe** eines 2-3-4-Baums *(siehe Beispiel eben)* steigt nur, wenn eine **neue Wurzel** angelegt wird.
- **Hieraus folgt**: Alle Blätter befinden sich auf **derselben Ebene**!
- **Schlechtester Fall**: Eine Seite nur **"leere"**, andere Seite nur **volle** Knoten.

2-3-4-Bäume: Effizienz

- Was ist die **maximale Höhe** eines 2-3-4-Baums mit **n Elementen**?
- **Wieviele Knoten** werden maximal beim **Suchen** in einem 2-3-4-Baum besucht? **Wieviele Vergleiche** werden dabei maximal durchgeführt, und was ist die **Komplexität**?
- **Welche Komplexität** besitzt das **Einfügen** in einen 2-3-4-Baum?

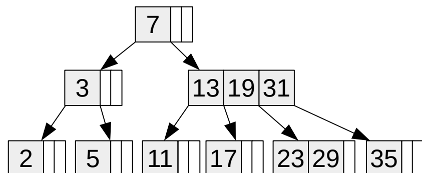
2-3-4-Bäume: Diskussion

2-3-4-Bäume sind **ausgeglichene Bäume**

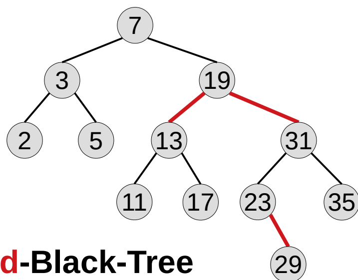
- Effizientes Suchen + Einfügen
- Löschen: nicht besprochen.

Ausblick: Red-Black-Trees

- **Binäre Version** von 2-3-4-Bäumen
- Einfügen / Suchen / Löschen: **Garantiert $O(\log(n))$** ☺
- Wird im **JCF** verwendet (`TreeMap`).



2-3-4-Baum



Red-Black-Tree

Outline

1. 2-3-4-Bäume

2. 2-3-4-Bäume: Effizienz

3. B-Bäume

4. B-Bäume: Effizienz

ADS meets Datenbanken

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

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

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;
```

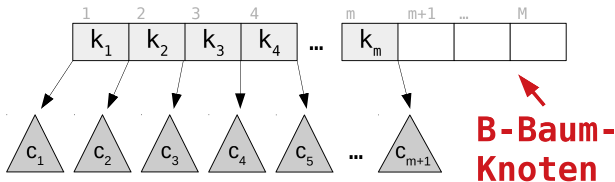
- **Index** anlegen → **500-facher Speed-up** ☺

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

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

B-Bäume = "Generelle" 2-3-4-Bäume

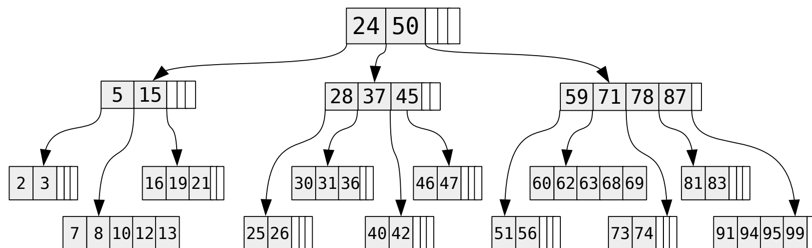
Im **2-3-4-Baum** gibt es **1 bis 3 Schlüssel** je Knoten.
Im **B-Baum** sind **M** *(z.B.* 1000*)* Schlüssel je Knoten erlaubt.



Knoten in B-Bäumen

- Ein Knoten enthält m **aufsteigende Schlüssel** $k_1, ..., k_m$.
- m beträgt **maximal** M und **mindestens** $\lfloor M/2 \rfloor$.
- M ist der maximale **Branch-Faktor** des Baums.
- **Zwischen** den Schlüsseln sind Kinder/**Subbäume** $c_1, ..., c_{m+1}$:
  - **Subbaum $\mathbf{c_1}$** enthält nur Schlüssel $< k_1$.
  - **Subbaum $\mathbf{c_{m+1}}$** enthält nur Schlüssel $> k_m$.
  - **Alle anderen** c_i enthalten nur Schlüssel zwischen k_{i-1} und k_i.

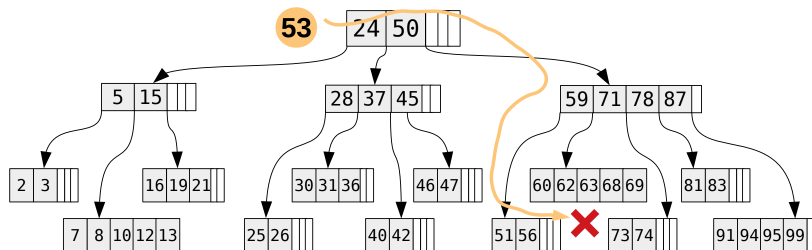
B-Bäume: Beispiel



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

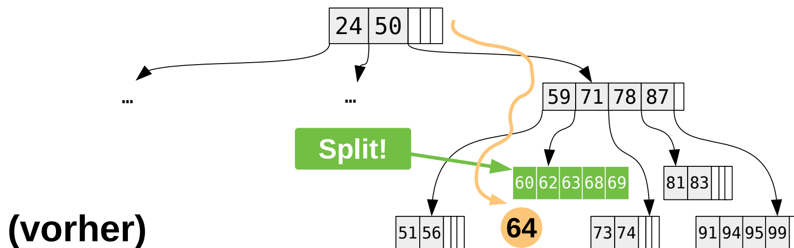


Beispiel: Suche 53

Wie im **2-3-4-Baum**

- Beginne bei der **Wurzel**.
- **Durchsuche** den aktuellen Knoten *(z.B. linear oder* ***binär****)*.
- Schlüssel **gefunden** → zurückgeben.
- Schlüssel **nicht gefunden** → gehe zu Kind.

B-Bäume: Einfügen

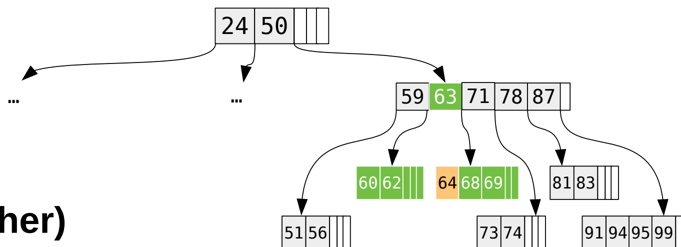


Beispiel: Füge 64 ein

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 Schlüssel (68,69) kommen in neuen Knoten.

Outline

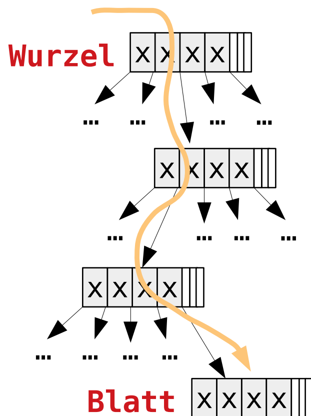
1. 2-3-4-Bäume

2. 2-3-4-Bäume: Effizienz

3. B-Bäume

4. B-Bäume: Effizienz

B-Bäume: Effizienz



Suche in B-Bäumen: Aufwand

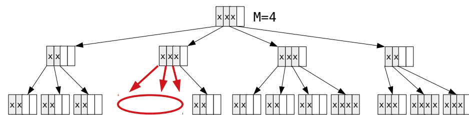
- Gegeben: B-Baum mit **n Schlüsseln**.
- Wie teuer ist die Suche im **Worst Case**?

Aufwand je Knoten $\times$
Höhe des Baums

Faktor 1: Aufwand je Knoten

- **Durchsuchen** eines Knotens nach passendem Schlüssel.
- Im Knoten befinden sich maximal M Schlüssel.
- **Binäre Suche** $\rightarrow O(log\ M)$.
- **Hinweis**: In der Praxis ist M zwar **groß** *(z.B. 1000)*, aber **konstant**! Denn M wächst nicht mit der Datenmenge.
- Also beträgt der **Aufwand je Knoten O(1)**.

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:

$$n = M \cdot \left(1 + \underbrace{(M+1)}_{\#\text{Ebene 2}} + \underbrace{(M+1)\cdot(M+1)}_{\#\text{Ebene 3}} + \ldots + \underbrace{(M+1)^{h-1}}_{\#\text{Ebene h}}\right)$$

$$\rightarrow n = M \cdot \left(\frac{(M+1)^h - 1}{(M+1) - 1}\right) \qquad \text{// siehe Analysis}$$

$$\rightarrow n \approx (M+1)^h$$

$$\rightarrow \mathbf{h} \approx \mathbf{log}_{\mathbf{M+1}}(\mathbf{n}).$$

Bei **minimalem Füllgrad** gilt analog: $\mathbf{h} \approx \mathbf{log}_{\lfloor \mathbf{M/2} \rfloor + \mathbf{1}}(\mathbf{n})$.

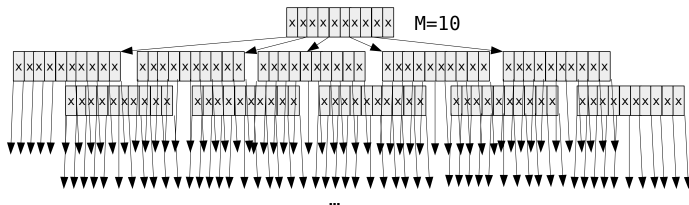
Suche im B-Baum

Gesamtkomplexität (Worst Case)

$$\textit{Aufwand je Knoten} \times \textit{Höhe des Baums}$$
$$O(1) \quad \times \quad O(\log_{\lfloor M/2 \rfloor} n)$$

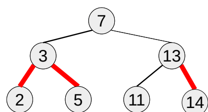
Anmerkungen

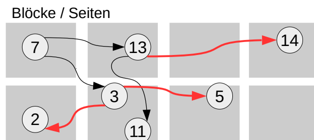
- Bei **großem M** ist die Baumhöhe sogar **quasi konstant**.
- **Beispiel**: $M = 100$, Höhe 5
  → Platz für bis zu $>$ **10 Mrd. (!!!) Schlüssel!**



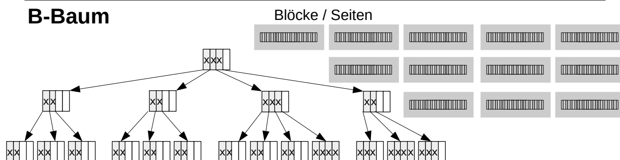
B-Bäume: Seitenoptimierung

- In **realen Rechnerarchitekturen** werden Daten per Block *(engl. “page)* aus dem **Hauptspeicher** geladen.
- Das Laden eines Blocks ist **extrem zeitaufwändig**!
- **Binärbäume** *(mit verstreuten Knoten)* sind hier ungünstig *(Laden vieler Blöcke)*.
- **B-Bäume** laden mit einem Knoten viele Daten auf einmal. Sie sind **seiten-optimiert**.





Binärbaum



References I