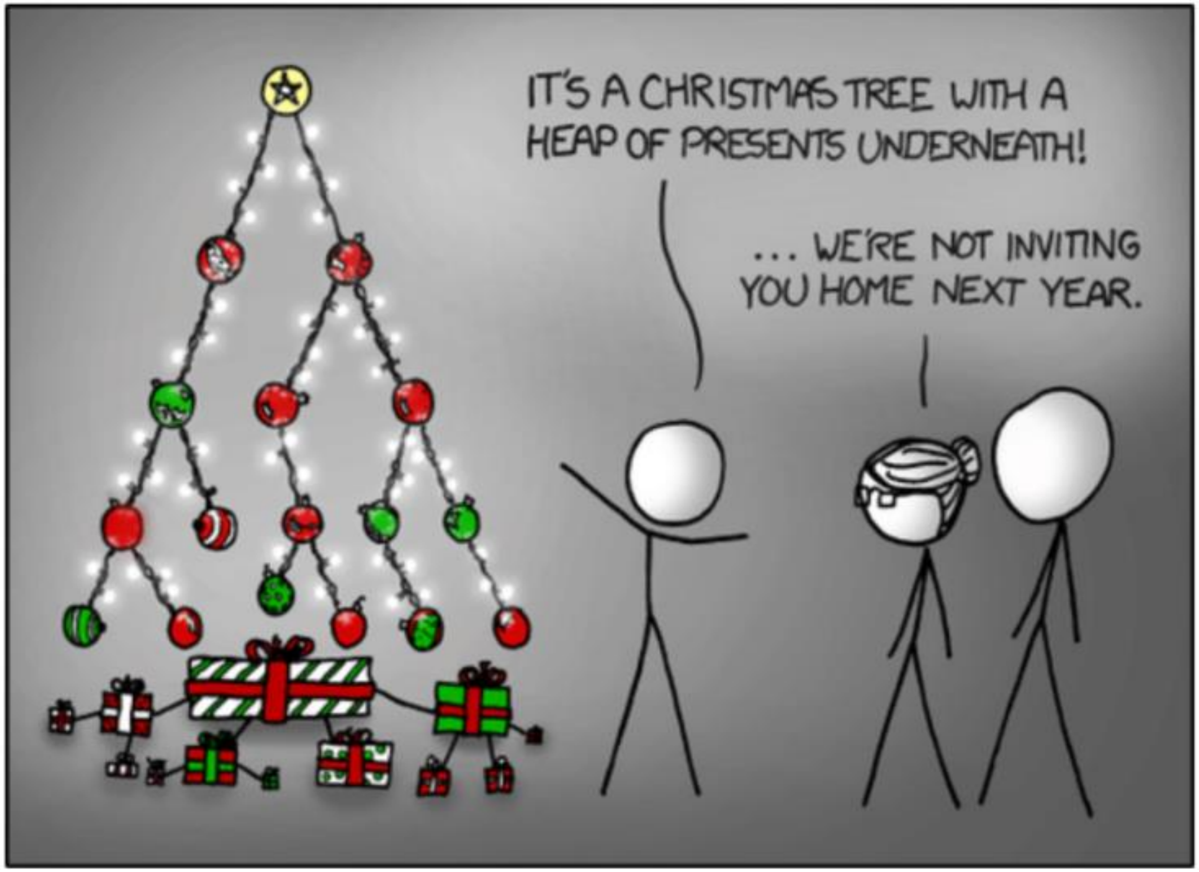
Algorithmen und Datenstrukturen
Kapitel 6C: Vielwegbäume

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Zur Auflockerung …



Quelle: [3]

- Hoffentlich passiert Ihnen das nicht dieses Jahr! ☺

Übersicht

- **Binäre Suchbäume**
  - Suchen, Einfügen und Entfernen von Schlüsseln
  - Traversieren von Bäumen
  - Laufzeitanalyse
- Balancierte Binärbäume
  - AVL-Bäume
  - AVL-Bäume in Java
  - Ausblick: Rot-Schwarz-Baum, Bruder-Baum
- **B-Bäume**
  - **Motivation**
  - Definition und Anwendung
  - Operationen: Suchen, Einfügen
  - Zusammenfassung

ADT Map und Binäre Suchbäume

- **Map, Dictionary, Symboltabelle (dt. "assoziatives Datenfeld")**
  - Speichert Key-Value Pairs (dt. "Schlüssel-Werte-Paare")
  - Bsp.: Alter von Personen → { (Trump, 73), (Merkel, 65), (Kurz, 33) }

- **Typische Operationen**
  - `void put(Key key, Value value)` (oft auch "`insert`")
  - `Value get(Key key)`
  - `void delete(Key key)` (oft auch "`remove`")
  - ...

- Bei Red-Black Trees gilt für die Basisoperationen: $\boldsymbol{O(\log n)}$
  - Voraussetzung: Daten passen in Hauptspeicher!

- **Problem:** Was tun, falls Datensatz nicht in Hauptspeicher passt?
  - Beispiel: Datenbanken

B-Baum: Überblick

- Ein **B-Baum** ist
  - ein ***balancierter*** Suchbaum
  - mit hohem Verzweigungsfaktor: Knoten darf ***viele Kinder*** haben!
  - Höhe: $h \approx O(\log_d n) = O(\log_2 n) = O(\log n)$
    - Sehr langsames Höhenwachstum bei vielen Einträgen

- ***Verwendung***
  - Suche in ***sehr großen*** Datenmengen.
  - Daten liegen auf Festplatte oder müssen durch Webanfrage geholt werden. Jeder Zugriff auf Datenblock (***Page***) ist teuer.

- ***Ziele***
  - Minimiere die Anzahl der Zugriffe auf Festplatte/Web ***(Disk I/O)***
  - Minimiere die Rechenzeit ***(CPU)***

Motivation: Datenbanken

employees

employee_id	last_name	job_id	manager_id	hire_date	salary	department_id
203	marvis	hr_rep	101	07–Jun–94	6500	40
204	baer	pr_rep	101	07–Jun–94	10000	70
205	higgins	ac_rep	101	07–Jun–94	12000	110
206	gietz	ac_account	205	07–Jun–94	8300	110
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Employees: Beispiel für eine Tabelle einer Datenbank [4]

- ***SQL statement***
  - `CREATE INDEX emp_deptid_ix ON employees(department_id);`

- Beim Anlegen eines Index wird im Hintergrund meist ein ***B-Baum*** (oder B+-Baum) erzeugt.
  - B-Baum so groß, dass er nicht komplett in den Hauptspeicher passt.

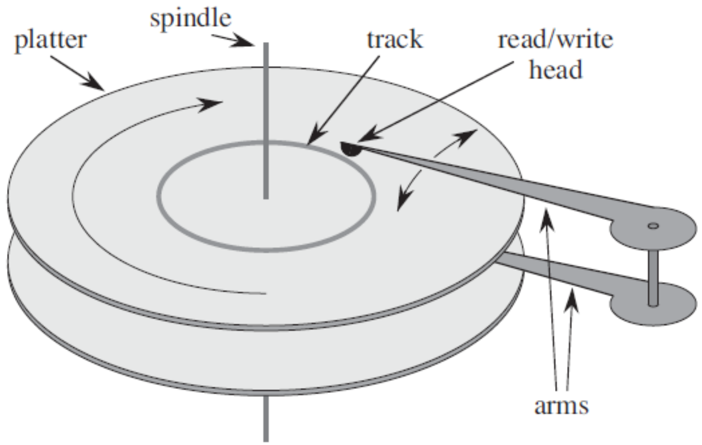
Motivation: Festplatte, Webzugriff

- Festplattenzugriff
  - Notwendig, da riesige Datenmenge
  - Extrem langsam im Vergleich zur CPU
  - Zugriffseinheit: Festplatten-Datenblock

- Verteilte Daten im Web / Cloud
  - Webanfragen (z.B. REST) um Datensätze herunterzuladen
  - Extrem langsam im Vergleich zur CPU
  - Zugriffseinheit: Web Request

- **Page**
  - Kleinste Einheit, auf die einzeln zugegriffen werden kann
  - Beispiel: Webrequest, Festplatte-Datenblock

- Minimiere
  - **CPU:** Rechenzeit
  - **Page:** Anzahl der Zugriff



Aufbau einer Festplatte: [1]

Übersicht

- Binäre Suchbäume
  - Suchen, Einfügen und Entfernen von Schlüsseln
  - Traversieren von Bäumen
  - Laufzeitanalyse
- Balancierte Binärbäume
  - Rot-Schwarz-Bäume
  - Suche, Einfügen, Löschen
- **B-Bäume**
  - Motivation
  - **Definition**
  - Operationen: Suchen, Einfügen
  - Zusammenfassung

Zugriff auf eine "Page"

- Verwende Baum, so dass: **1 Knoten == 1 Page**
  - Vor Lesen/Schreiben: Bringe Page in Hauptspeicher
  - Erinnerung: Page entweder Festplattenblock oder Datensatz aus Web.

- Repräsentation von Pagezugriffen im Pseudocode
  - **`DISK-READ(x)`**
    - Page schon im Hauptspeicher → Funktion lädt zunächst Page
    - Page nicht im Hauptspeicher → Leere Operation / "`no-op`"
  - **`DISK-WRITE(x)`**
    - Rückschreiben von Änderungen auf Festplatte bzw. ins Web.

- Typischer Ablauf

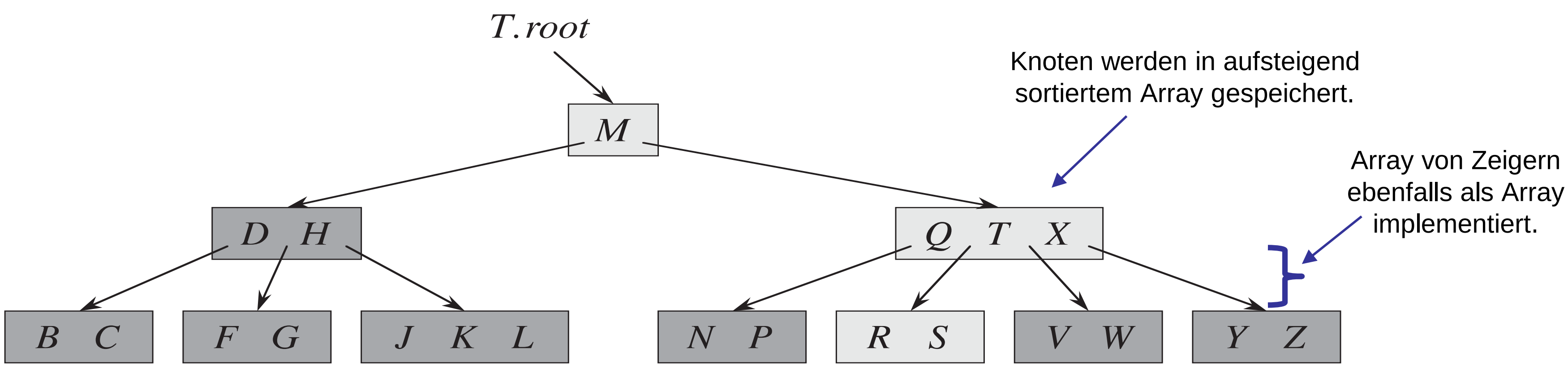
```
x = pointer to an object
DISK-READ(x)
verändere x
DISK-WRITE(x)
. . .
```

Annahme im Folgenden:
Wurzel des Baumes ist immer im Hauptspeicher

B-Baum: Erstes Beispiel

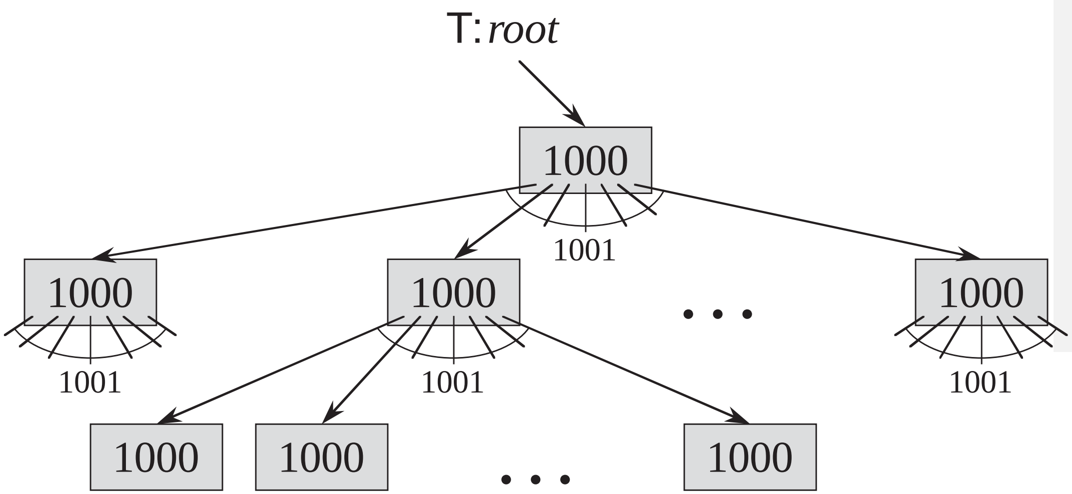
- Verallgemeinerung eines binären Suchbaums.
  - Hier: Jeder Knoten ***maximal 4 Kinder*** und ***maximal 3 Schlüssel***.
  - Schlüssel sind ***sortiert*** gespeichert.
  - Schlüssel trennen Wertebereiche in den darunterliegenden Teilbäumen.

- Wie sucht man in so einem Baum?
  - In jedem Knoten muss man mit *maximal* 3 Schlüssel vergleichen
  - Beim Weitergehen zum Kind hat man 4 Möglichkeiten.



Quelle: [1]

Publikums-Joker: Ein realistischer B-Baum



Annahmen:
Obere und untere Schranke für Anzahl der Schlüssel pro Knoten

- Jeder Knoten speichert 1000 Schlüssel.
- Jeder Knoten sei voll belegt.
- Höhe 2!

Welche Aussage ist ***wahr***?

A. Der Baum speichert maximal 1 Milliarde Schüssel.
B. Man benötigt bei der Suche maximal 2 Knotenzugriffe.
C. Jeder innere Knoten hat 1000 Kinder.
D. Der Baum enthält mehr als 1 Milliarde Knoten.

B-Baum: Definition

Quellcode: BTree.java
Private KlasseBNode

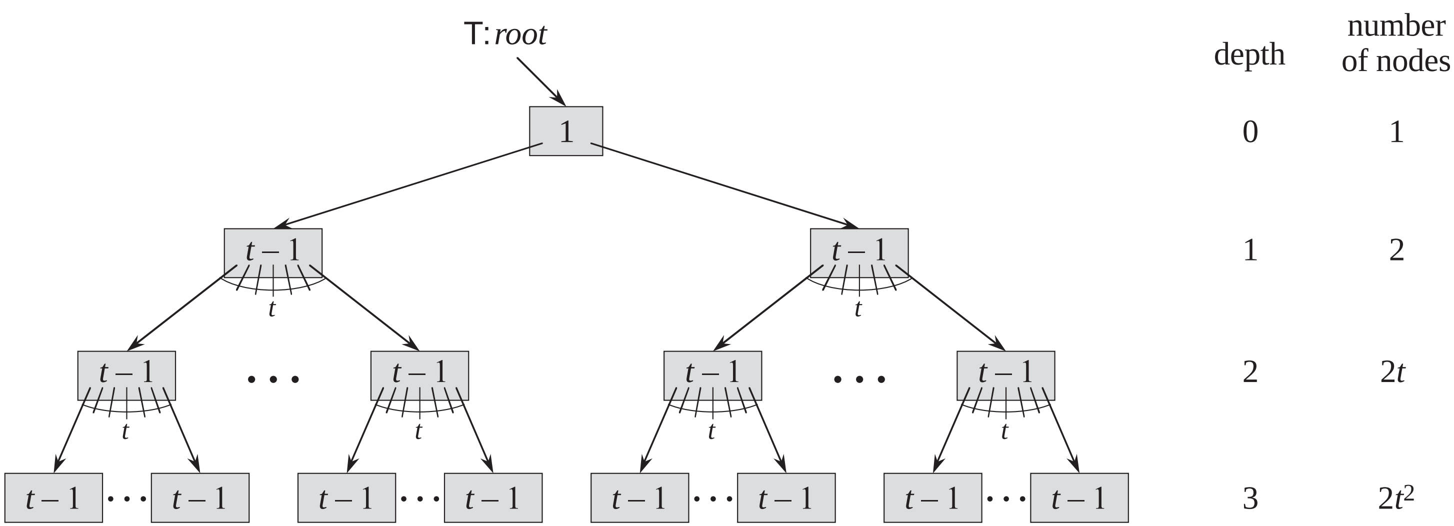
- ***Attribute <u>jedes</u>*** Knoten x
  - ***n:*** Anzahl der Schlüssel im Knoten
  - ***keys:*** Array von aufsteigend sortieren Schlüsseln
  - ***vals:*** Array von Werten, gleiche Reihenfolge wie Schlüssel.
  - ***leaf*:** `TRUE`, falls *x* ein Blatt ist.
  - ***children (kurz "c")*** $\boldsymbol{x.c}$**:** Array von *x.n*+1 - Zeiger auf Kinder
    - Falls $\mathrm{x.key}[i] < k < x.key[i+1]$, dann ist der Schlüssel *k* in dem Teilbaum gespeichert, der durch $x.c[i+1]$ referenziert wird.

- ***T*** **:** Minimaler Grad eines B-Baumes
  - ***Minimaler Grad***: $\boldsymbol{T}$ Kinder pro Knoten (entspricht *T - 1* Schlüssel)
  - ***Maximaler Grad:*** $\mathbf{2}\boldsymbol{T}$ Kinder pro Knoten (entspricht 2*T - 1* Schlüssel)
  - Ausnahme: Wurzel, Blätter
  - Spezialfall eines (a,b)-Baumes: https://de.wikipedia.org/wiki/(a,_b)-Baum
  - <u>Hinweis:</u> Ein voll belegter Knoten speichert *ungerade viele* Schlüssel.

- Alle Blätter haben die gleiche Tiefe.

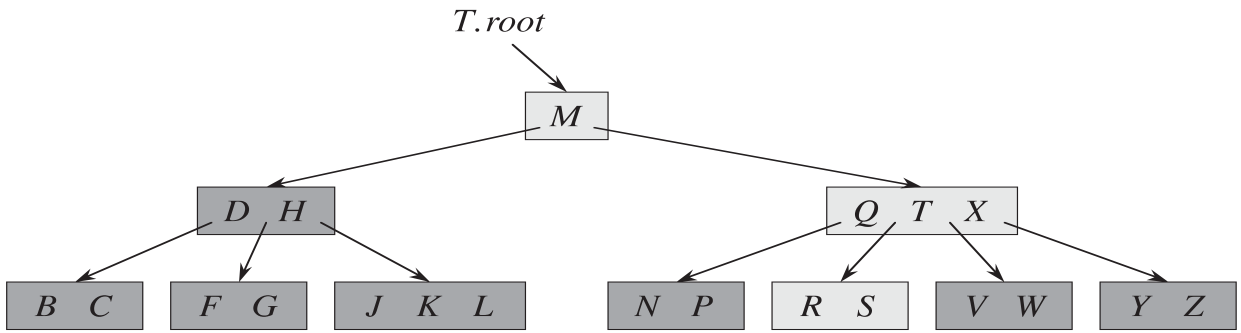
B-Baum: Höhe

- Die Anzahl der Festplattenzugriffe ist proportional zur Höhe: $O(h)$
- ***Theorem:*** $h \leq \log_t \frac{n+1}{2}$ (falls $\mathrm{n} \geq 1, \mathrm{t} \geq 2$)
  - Ohne Beweis
  - Das bedeutet: $h = O(\log n)$
  - Abbildungen unten illustriert den Fall, dass jeder Knoten die minimale Anzahl an Kindern hat (obere Schranke)



Quelle: [1]

Publikums-Joker



Für welche Werte von *t* ist der obige Baum ein legaler B-Baum?

A. Nur t=3

B. t=2 und t=3

C. t=3 und t=4

D. t=5

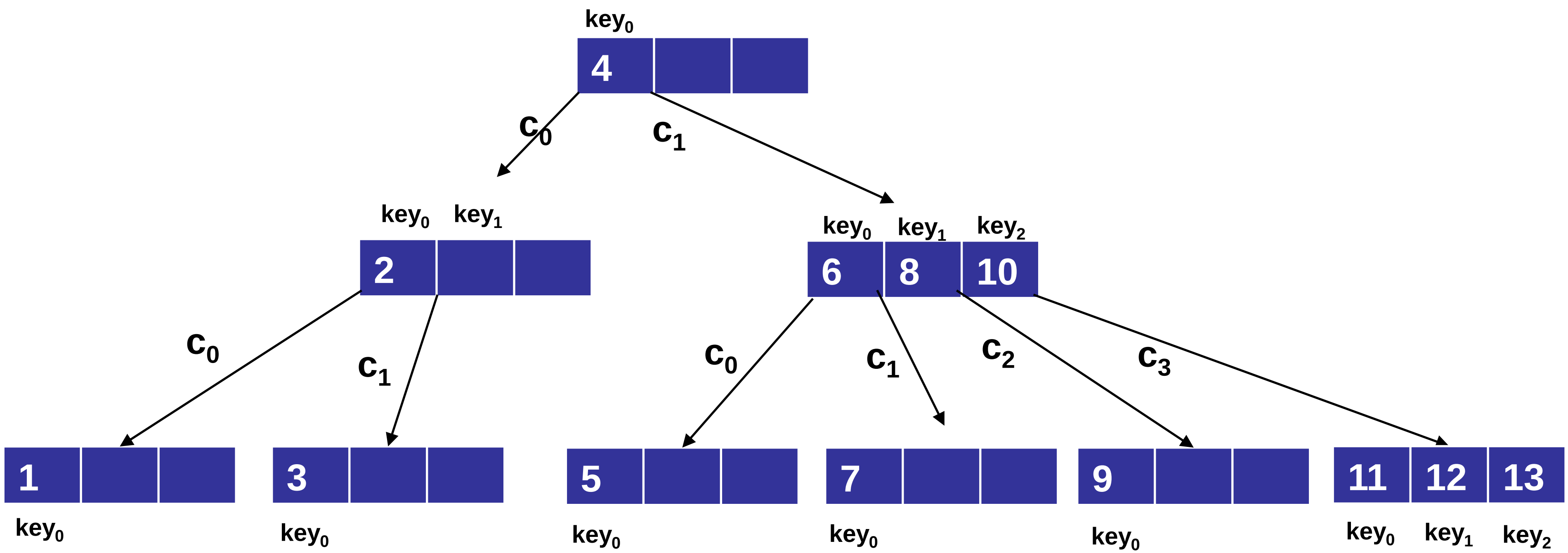
Erinnerung:
Obere und untere Schranke für Anzahl der Schlüssel pro Knoten

- Untere Schranke: Jeder Knoten (außer Wurzel) hat mindestens $t - 1$ Schlüssel.
- Obere Schranke: Jeder Knoten (außer Wurzel) hat höchstens $2t - 1$ Schlüssel.

Übersicht

- Binäre Suchbäume
  - Suchen, Einfügen und Entfernen von Schlüsseln
  - Traversieren von Bäumen
  - Laufzeitanalyse
- Balancierte Binärbäume
  - Rot-Schwarz-Bäume
  - Suche, Einfügen, Löschen
- **B-Bäume**
  - Motivation
  - Definition
  - **Operationen**
  - Zusammenfassung

B-Baum mit t=2: Suche die 13?



- Verweise sind "zwischen" den Schlüsseln gespeichert.
- **Animation**
  - https://www.cs.usfca.edu/~galles/visualization/BTree.html
  - "Max. Degree"
    - Maximale Anzahl der Kinder
    - Entspricht in unserer Definition: 2**T*

Suche

- ***Aufrufparameter***
  - ***x***: Zeiger auf beliebigen Knoten, initialer Aufruf mit Wurzel
  - ***k:*** Gesuchter Schüsselwert
- ***Rückgabe:*** Werte / Value
- **Laufzeit** bei Suche beginnend an der Wurzel: `SEARCH(x, root)`
  - CPU: $O(t \log_t n)$
  - Disk: $O(\log_t n)$

```
GET(Node x, Key k)
    i = 0
    while i < x.n and k > x.key[i]
        i = i + 1
    if i < x.n and k == x.key[i]
        return x.vals[i]
    elseif x.leaf is true
         return "null"
    else
        DISK-READ(x.c[i])
        return GET(x.c[i], k)
```

Quellcode: BTree.java / get

Suche Index in x.key[] an der k stehen müsste.
Höre auf, sobald $k > x.key_i$

Fall 1 "Erfolgreich": k ist im Knoten x gespeichert. .

Fall 2 "Erfolglos": Da Knoten ein Blatt ist,
kann k gar nicht im B-Baum gespeichert sein.

Fall 3 "Unklar": Man muss einen Kindknoten laden,
evtl. ist dort der Schlüssel gespeichert. Rekursion!

Erzeugen eines leeren B-Baumes

- Erzeuge mit **`CREATE`** einen leeren B-Baum des Grades ***T***.
- Anschließend: Füge mit **`PUT`** die gewünschten Schlüssel hinzu.

- **`CREATE`**
  - Erzeugt ersten leeren Knoten *x*.
  - Initialisiert Attribute, z.B.
    - `Leaf = true`: Wurzel ist Blatt!
    - `x.n = 0:` Noch keine Schlüssel im Baum.

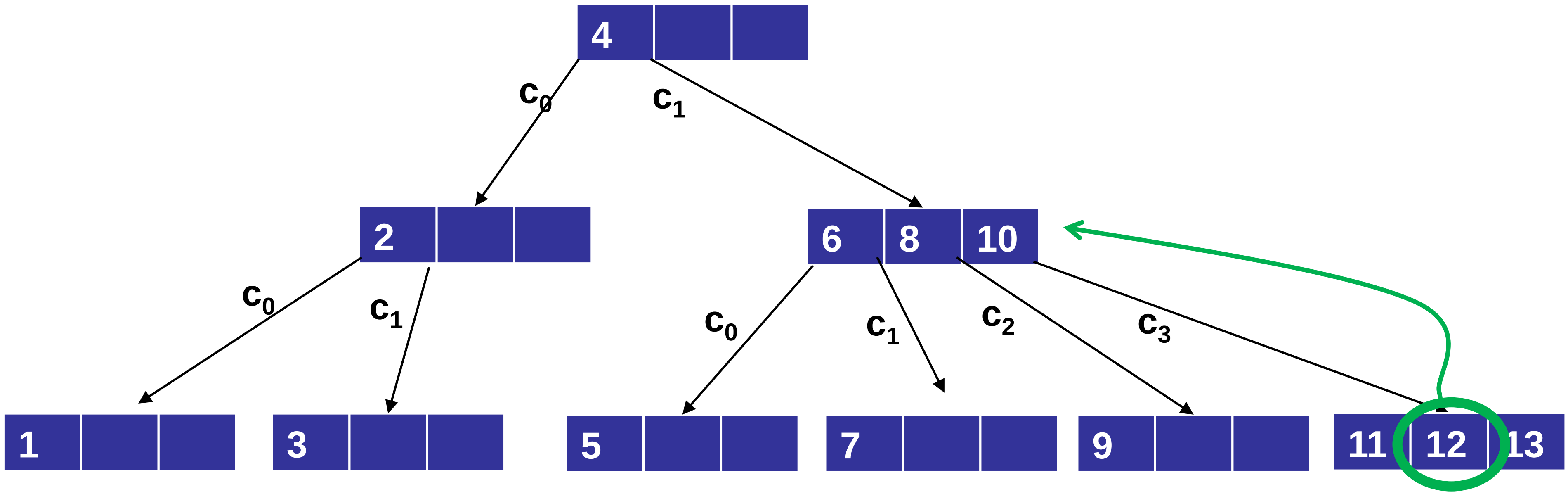
- **`ALLOCATE-NODE()`**
  - Alloziiert Speicher für neuen Knoten
  - In Java passiert das durch Konstruktor.
  - Laufzeit: O(1)

- **Laufzeit von `CREATE(t)`**
  - CPU: $O(1)$
  - Disk: $O(1)$

```
CREATE(t)
x = ALLOCATE-NODE()
x.leaf = true
x.n = 0
root = x
DISK-WRITE(x)
```

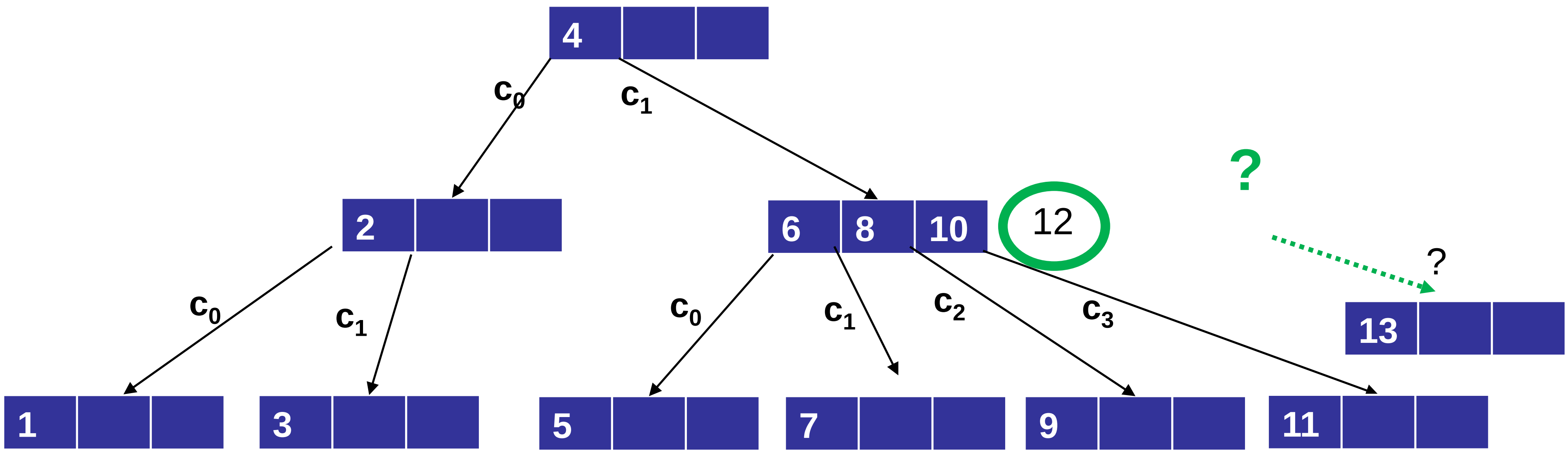
Quellcode: BTree.java
Siehe vor allem Konstruktor

Beispiel: Einfügen von 14



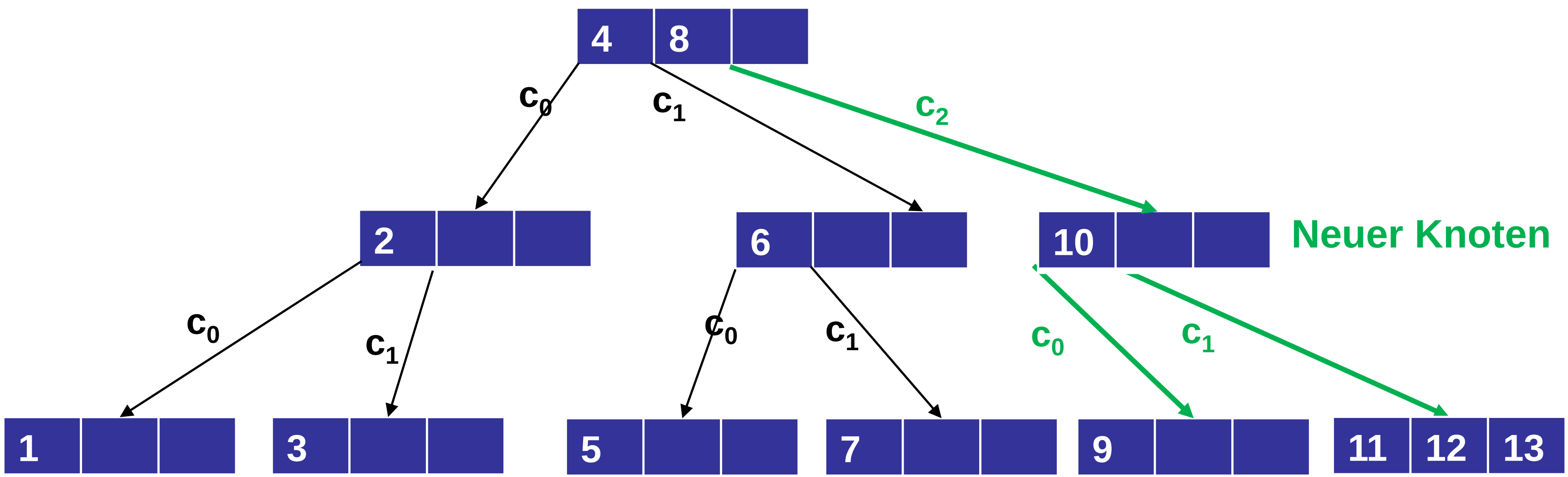
- **Idee**
  - Suche zunächst korrekte Einfügeposition (in einem Blatt).
- **Problem**
  - Knoten, der 14 speichern müsste, bereits voll belegt.
- **Idee**
  - Vor Einfügen: Spalte Knoten und bringe mittleres Element 12 (=Median von 11, 12, 13) um 1 Ebene nach oben.

Beispiel: Einfügen von 14



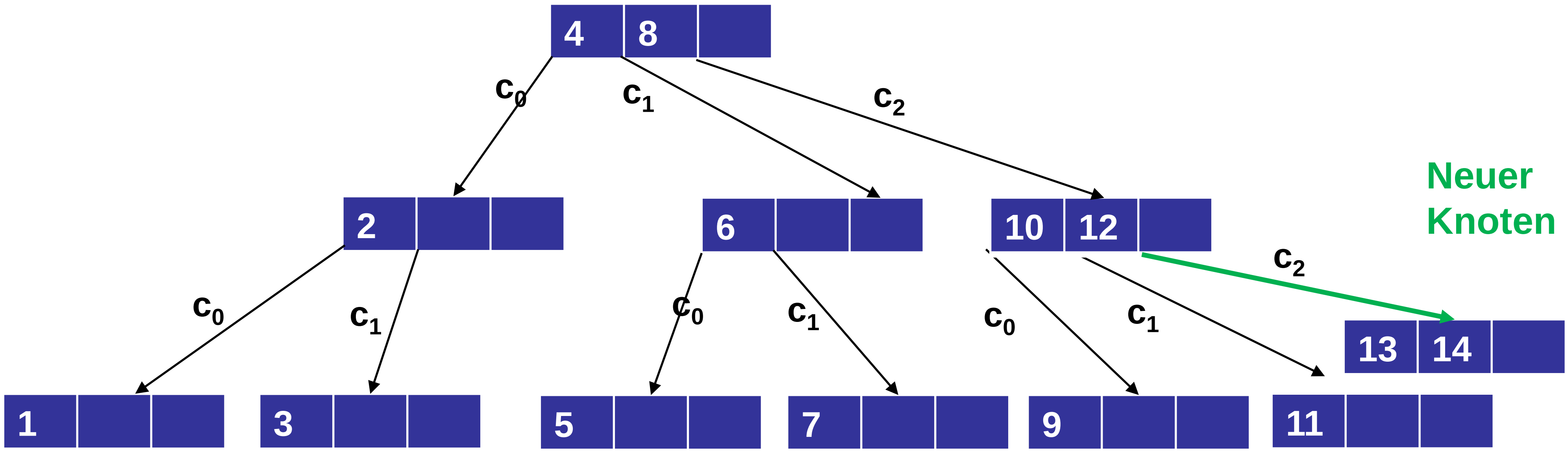
- **Problem:**
  - Elternknoten ist bereits voll und müsste ebenfalls aufgeteilt werden.
- **Lösung:** ***Preemptive Split***
  - Viele Implementierungen teilen bereits beim Suchen der Einfügeposition auf dem Weg von der Wurzel bis zum Blatt ***proaktiv(!)*** alle Knoten auf, die bereits voll sind.
  - Und nicht erst, wenn es zu spät ist.

Preemptive Split: Einfügen von 14



- <u>Bereits beim Suchen</u> der Einfügeposition für den Schlüssel 14 passiert man den vollen Knoten mit 6, 8 und 10 (siehe Vorvorgängerfolie)
- **Preemptive:**
  - Teile diesen proaktiv auf und bringe mittleren Knoten (8) nach oben.
  - Ergebnis, siehe Skizze.
- **Vorteil:**
  - Einfügen ist dann später sicher erfolgreich. Man muss nicht mehr zurück Richtung Wurzel laufen.

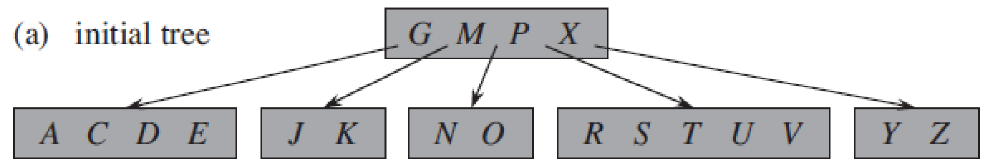
Preemptive Split: Einfügen von 14



- Beim Suchen der Einfügeposition für den Schlüssel 14 passiert nun als nächstes den Knoten 11, 12, 13 (siehe Vorgängerfolie)
  - Dieser Knoten ist ein Blatt. Dort gehört der Schlüssel 14 hin.
  - Da bereits voll: Aufteilen und mittleren Knoten nach oben schieben.
- Warum ist dort nun Platz?

Weiteres Beispiel: Es ist noch Platz im Knoten

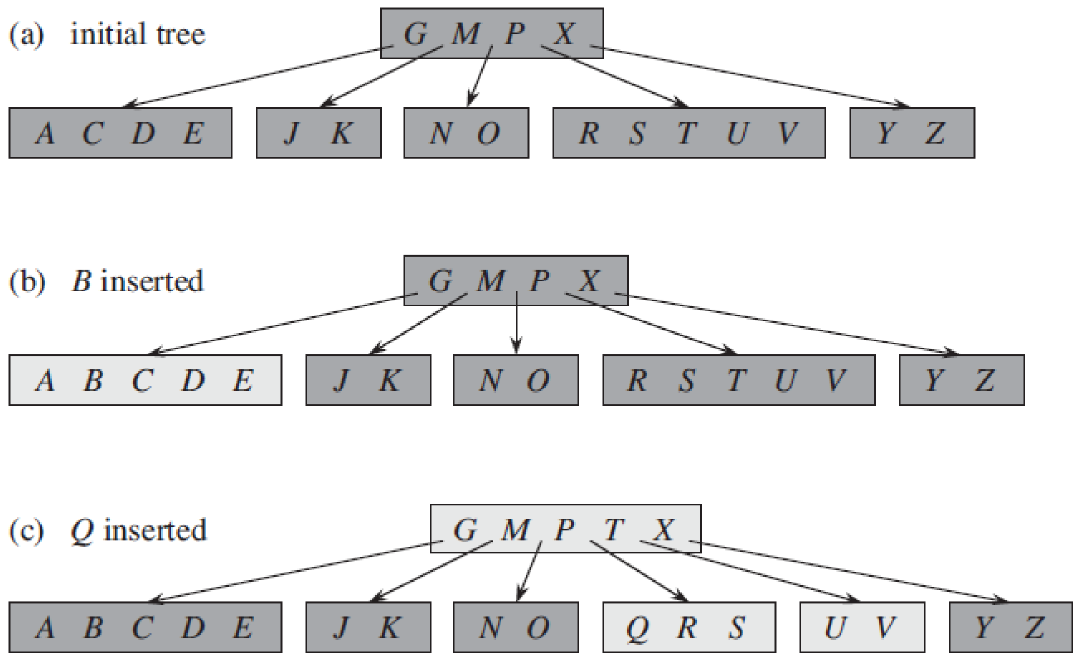
- t = 3: Jeder Knoten hat
  - höchstens 6 Kinder
    - 5 Schlüssel
  - mindestens 3 Kinder
    - 2 Schlüssel



- Einfügen von *B und Q*
  - Für beide Schlüssel ist noch Platz in den Knoten.

Weiteres Beispiel: Es ist noch Platz im Knoten

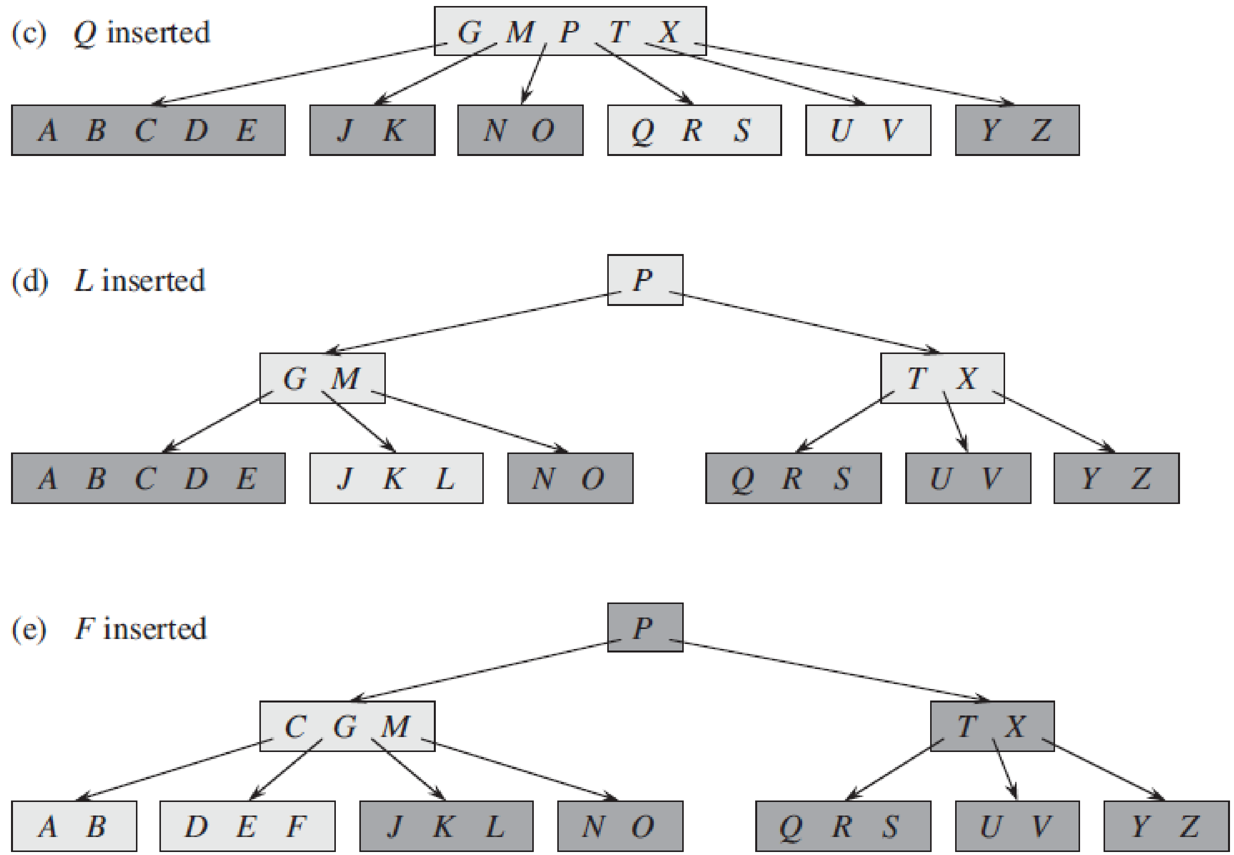
- $t = 3$: Jeder Knoten hat
  - höchstens 6 Kinder
  - mindestens 3 Kinder
- Einfügen von *B und Q*
  - Für beide Schlüssel ist noch Platz in den Knoten.



Quelle: [1]

Preemptive Split bei Einfügen von "L"

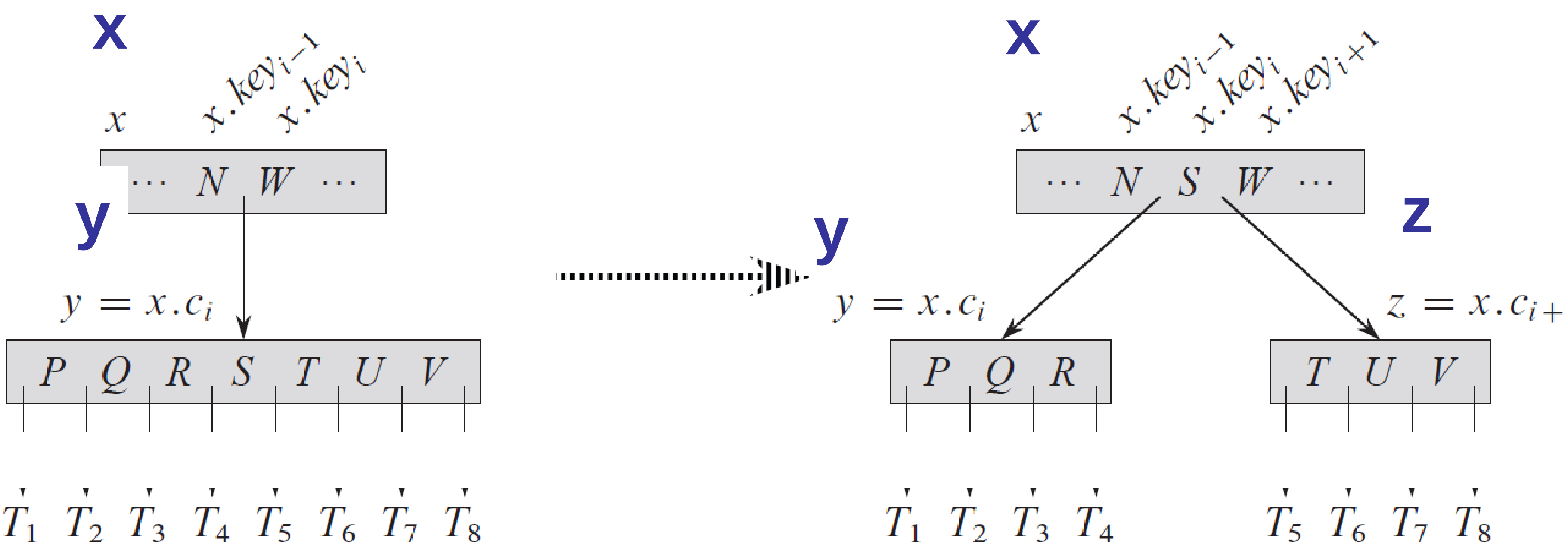
Quelle: [1]



- Was passiert, wenn man L einfügt?
- Wende Strategie des ***Preemptive Split*** an: Teile bereits bei der Suche der Einfügeposition auf.

Wie teilt man einen Knoten?

- Hilfsmethode **SPLIT-CHILD(x,i)**
  - Teile das Kind ***y=c[i]*** des ***Knoten x*** auf.
  - Bringt den Median (hier *S*) nach oben in den Knoten *x*.
  - Methode wird nur aufgerufen, wenn ***y=c[i]*** auch tatsächlich ***voll*** ist.
- Code nicht schwer, aber "unangenehm"



Quelle: [1]

Diskussion

- **Löschen**
  - Noch komplizierter, wird nicht behandelt!
  - http://www.geeksforgeeks.org/b-tree-set-3delete/
  - Evtl. zu löschenden Schlüssel nur als "gelöscht markieren".

- **Animationen**
  - https://www.cs.usfca.edu/~galles/visualization/BTree.html
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
    - **B+-Baum:** Variante des B-Baumes in dem die eigentlichen Datenelemente / Schlüssel nur in den Blättern gespeichert werden. Die Blätter sind miteinander verkettet.

- Ein Beispiel-Quellcode wird mitgeliefert.
  - Code jedoch kein Klausurstoff!

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 5.1, 5. Auflage, Spektrum Akademischer Verlag, 2012.

[3] Quelle: https://cs124.quora.com/xkcd-comics

[4] https://docs.oracle.com/cloud/latest/db112/CNCPT/indexiot.htm#CNCPT1170, abgerufen am 24.11.2017