Kapitel ADS:V

V. Suchen

- ❑ Binary Search Tree
- ❑ AVL Tree
- ❑ Red-Black Tree
- ❑ Maschinenmodell (Erweiterung)
- ❑ B-Tree

Maschinenmodell (Erweiterung)

Sekundärspeicher

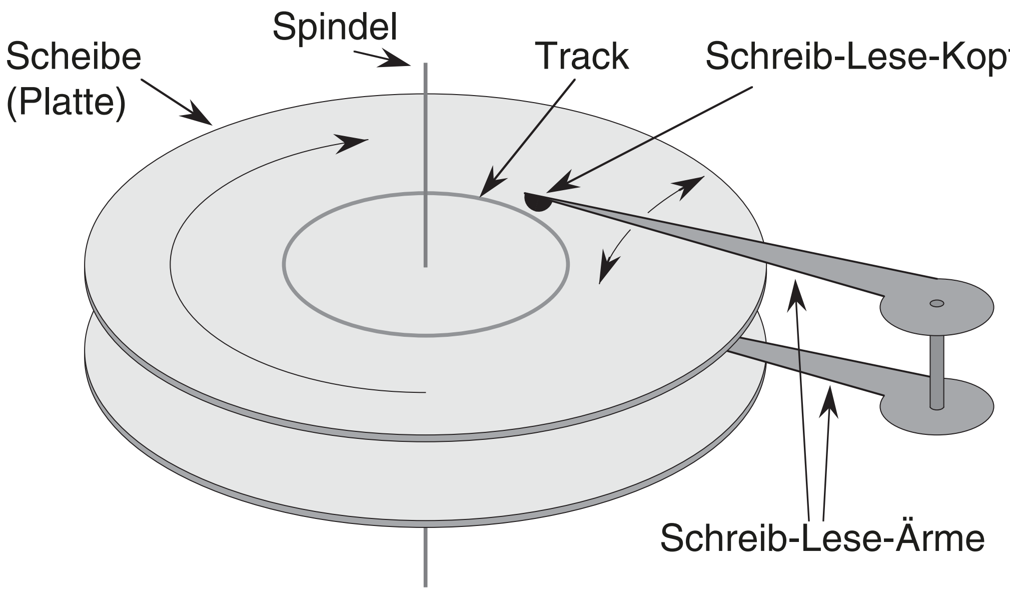
Die Verarbeitung großer Datenmengen erfordert Zugriffe auf Sekundärspeicher.
Abhängig vom Speichermedium ist jeder Zugriff mit Latenz behaftet.

Maschinenmodell (Erweiterung)

Sekundärspeicher

Die Verarbeitung großer Datenmengen erfordert Zugriffe auf Sekundärspeicher. Abhängig vom Speichermedium ist jeder Zugriff mit Latenz behaftet.

Beispiel:



Beim wahlfreien Zugriff auf eine rotierende Festplatte vergehen zwischen Anforderung und Antwort 1-10ms Latenzzeit.

Maschinenmodell (Erweiterung)

Sekundärspeicher

Die Verarbeitung großer Datenmengen erfordert Zugriffe auf Sekundärspeicher. Abhängig vom Speichermedium ist jeder Zugriff mit Latenz behaftet.

Sekundärspeicher	**Latenz**
1 CPU cycle	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	2.8 ns
Level 3 cache access	12.9 ns
Main memory access (DDR4)	60 ns
Non-Volatile Memory (express)	10-20 μs
Solid-state disk I/O	50-150 μs
Rotational disk I/O	1-10 ms
Internet: SF to NYC	40 ms
Internet: SF to UK	81 ms
Internet: SF to Australia	183 ms
OS virtualization reboot	4 s
Hardware virtualization reboot	40 s
Physical system reboot	5 m

Maschinenmodell (Erweiterung)

Sekundärspeicher

Die Verarbeitung großer Datenmengen erfordert Zugriffe auf Sekundärspeicher. Abhängig vom Speichermedium ist jeder Zugriff mit Latenz behaftet.

Sekundärspeicher	**Latenz**	**Analogie**
1 CPU cycle	0.3 ns	1 second
Level 1 cache access	0.9 ns	3 seconds
Level 2 cache access	2.8 ns	9 seconds
Level 3 cache access	12.9 ns	43 seconds
Main memory access (DDR4)	60 ns	3 minutes
Non-Volatile Memory (express)	10-20 μs	10-20 hours
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
Hardware virtualization reboot	40 s	4 millenia
Physical system reboot	5 m	32 millenia

Bemerkungen:

- Aufstellungen dieser Art werden Peter Norvig [Norvig 1998] und Jim Gray [Gray 1999] zugeschrieben.
- Vorträge von Jeff Dean haben dafür gesorgt, dass sie unter dem inzwischen zu einem Informatiker-Meme avancierten Titel „Numbers Everyone Should Know“ neuerlich populär geworden sind [Dean 2009a] [Dean 2009b].
- Die gezeigte Aufstellung ist „Systems Performance: Enterprise and the Cloud“ von Brendan Gregg, bzw. einem Blogpost von Jeff Attwood zum Thema entlehnt [codinghorror.com].
- Wichtig ist bei diesen Latenzen nicht, ob jede individuelle Zahl exakt stimmt, die fortwährende Hardwareentwicklung sorgt ständig für Beschleunigungen. Vielmehr das Verhältnis der Latenzen verschiedener Technologieparadigmen zueinander ist entscheidend.
- In der Spalte „Analogie“ werden die Latenzen in leichter zu erfassende Größenordnungen umgerechnet, wobei ein CPU-Zyklus = 1 Sekunde als Referenz dient.

Maschinenmodell (Erweiterung)

Sekundärspeicher

Ablauf eines Zugriffs:

1. Anforderung eines Datums von der Kontrolleinheit des Speichermediums.

2. Auslesen der Page, die das Datum enthält.
   Speichermedien organisieren Daten in Blöcken gleicher Größe, genannt Pages. Die Größe einer Page umfasst üblicherweise zwischen 2KB und 16KB.

3. Einfügen der Page in einen Cache.
   Zukünftige Zugriffe auf Daten in derselben Page werden drastisch beschleunigt und so die Gesamtlaufzeit amortisiert.

4. Extraktion und Rückgabe des angeforderten Datums aus der Page.

Annahmen:

- Der Hauptspeicher kann nur begrenzt viele Pages aufnehmen.
- Ein Hintergrundprozess entfernt nicht mehr benötigte Pages.
- Diese Logistik wird bei der Algorithmenanalyse nicht berücksichtigt.

Maschinenmodell (Erweiterung)

Laufzeitanalyse

Pseudocode:

1. $x =$ `pointer to some object`
2. *read*(x)
3. `operations that access / modify` x
4. *write*(x)
5. `operations that access but do not modify` x

Semantik:

1. Eine Kopie von x ist im Hauptspeicher und / oder im Sekundärspeicher.
2. Hilfsfunktion, die x vom Sekundärspeicher anfordert, wenn nötig; andernfalls benötigt sie keine Latenzzeit.
3. Normale CPU-Rechenzeit.
4. Hilfsfunktion, die x im Sekundärspeicher speichert, wenn nötig; andernfalls benötigt sie keine Latenzzeit.
5. Normale CPU-Rechenzeit.

Maschinenmodell (Erweiterung)

Laufzeitanalyse

Bestandteile der Laufzeit eines Algorithmus mit Sekundärspeicherzugriffen:

- Rechenzeit: Summe der CPU-Rechenzeit gemäß RAM-Modell
  Jede primitive Anweisung wird mit konstanten Kosten c angesetzt.

- Latenzzeit: Summe der Wartezeit für Zugriffe auf Sekundärspeicher
  Jeder Zugriff wird mit konstanten Kosten c' angesetzt.

Oft gilt, dass die CPU ein vorliegendes Datum schneller verarbeitet als der Sekundärspeicher neue Daten nachliefern kann (Rechenzeit $\ll$ Latenzzeit).

→ Rechenzeit und Latenzzeit werden getrennt voneinander betrachtet.
Beides wird durch Bachmann-Landau-Symbole als Funktion der Problemgröße n bemessen.

B-Tree

Definition

Ein B-Tree ist ein gewurzelter Baum mit folgenden vier Eigenschaften:

1. Jeder Knoten x hat folgende Attribute:
   (a) $x.n$: Zahl der Sortierschlüssel, die x aktuell speichert.
   (b) $x.n$ Sortierschlüssel $x.\mathit{key}_1, \ldots, x.\mathit{key}_{x.n}$, so dass $x.\mathit{key}_1 \leq \ldots \leq x.\mathit{key}_{x.n}$.
   (c) $x.\mathit{leaf}$: Boolean-Flag, das *TRUE* ist, wenn x ein Blatt ist, sonst *FALSE*.
   (d) $x.n + 1$ Pointer $x.c_1, \ldots, x.c_{x.n+1}$ zu Kindern (ausgenommen Blätter).

B-Tree

Definition

Ein B-Tree ist ein gewurzelter Baum mit folgenden vier Eigenschaften:

1. Jeder Knoten x hat folgende Attribute:
   (a) $x.n$: Zahl der Sortierschlüssel, die x aktuell speichert.
   (b) $x.n$ Sortierschlüssel $x.\mathit{key}_1, \ldots, x.\mathit{key}_{x.n}$, so dass $x.\mathit{key}_1 \leq \ldots \leq x.\mathit{key}_{x.n}$.
   (c) $x.\mathit{leaf}$: Boolean-Flag, das *TRUE* ist, wenn x ein Blatt ist, sonst *FALSE*.
   (d) $x.n + 1$ Pointer $x.c_1, \ldots, x.c_{x.n+1}$ zu Kindern (ausgenommen Blätter).

2. Die Schlüssel von x definieren Intervalle von Schlüsseln, die im jeweiligen Teilbaum gespeichert werden: Sei k_i ein in Teilbaum $x.c_i$ gespeicherter Schlüssel, dann gilt $k_1 \leq x.\mathit{key}_1 \leq k_2 \leq x.\mathit{key}_2 \leq \ldots \leq x.\mathit{key}_{x.n+1} \leq k_{x.n+1}$.

B-Tree

Definition

Ein B-Tree ist ein gewurzelter Baum mit folgenden vier Eigenschaften:

1. Jeder Knoten x hat folgende Attribute:
   (a) $x.n$: Zahl der Sortierschlüssel, die x aktuell speichert.
   (b) $x.n$ Sortierschlüssel $x.\mathit{key}_1, \ldots, x.\mathit{key}_{x.n}$, so dass $x.\mathit{key}_1 \leq \ldots \leq x.\mathit{key}_{x.n}$.
   (c) $x.\mathit{leaf}$: Boolean-Flag, das *TRUE* ist, wenn x ein Blatt ist, sonst *FALSE*.
   (d) $x.n + 1$ Pointer $x.c_1, \ldots, x.c_{x.n+1}$ zu Kindern (ausgenommen Blätter).

2. Die Schlüssel von x definieren Intervalle von Schlüsseln, die im jeweiligen Teilbaum gespeichert werden: Sei k_i ein in Teilbaum $x.c_i$ gespeicherter Schlüssel, dann gilt $k_1 \leq x.\mathit{key}_1 \leq k_2 \leq x.\mathit{key}_2 \leq \ldots \leq x.\mathit{key}_{x.n+1} \leq k_{x.n+1}$.

3. Alle Blätter haben dieselbe Tiefe, entsprechend der Höhe h des Baums.

B-Tree

Definition

Ein B-Tree ist ein gewurzelter Baum mit folgenden vier Eigenschaften:

1. Jeder Knoten x hat folgende Attribute:
   (a) $x.n$: Zahl der Sortierschlüssel, die x aktuell speichert.
   (b) $x.n$ Sortierschlüssel $x.\textit{key}_1, \ldots, x.\textit{key}_{x.n}$, so dass $x.\textit{key}_1 \leq \ldots \leq x.\textit{key}_{x.n}$.
   (c) $x.\textit{leaf}$: Boolean-Flag, das *TRUE* ist, wenn x ein Blatt ist, sonst *FALSE*.
   (d) $x.n + 1$ Pointer $x.c_1, \ldots, x.c_{x.n+1}$ zu Kindern (ausgenommen Blätter).

2. Die Schlüssel von x definieren Intervalle von Schlüsseln, die im jeweiligen Teilbaum gespeichert werden: Sei k_i ein in Teilbaum $x.c_i$ gespeicherter Schlüssel, dann gilt $k_1 \leq x.\textit{key}_1 \leq k_2 \leq x.\textit{key}_2 \leq \ldots \leq x.\textit{key}_{x.n+1} \leq k_{x.n+1}$.

3. Alle Blätter haben dieselbe Tiefe, entsprechend der Höhe h des Baums.

4. Die Zahl der Schlüssel $x.n$ eines Knotens ist beidseitig beschränkt. Sei $t \geq 2$:
   (a) Die Wurzel eines nicht-leeren Baums hat mindestens einen Schlüssel.
   (b) Jeder Knoten (außer der Wurzel) hat mindestens $t - 1$ Schlüssel.
   (c) Jeder Knoten hat höchstens $2t - 1$ Schlüssel.

Bemerkungen:

- Der B-Baum wurde 1972 von Rudolf Bayer und Edward M. McCreight entwickelt. Er erwies sich als ideale Datenstruktur zur Verwaltung von Indizes für das relationale Datenbankmodell, das 1970 von Edgar F. Codd entwickelt wurde. Diese Kombination führte zur Entwicklung des ersten SQL-Datenbanksystems System R bei IBM.

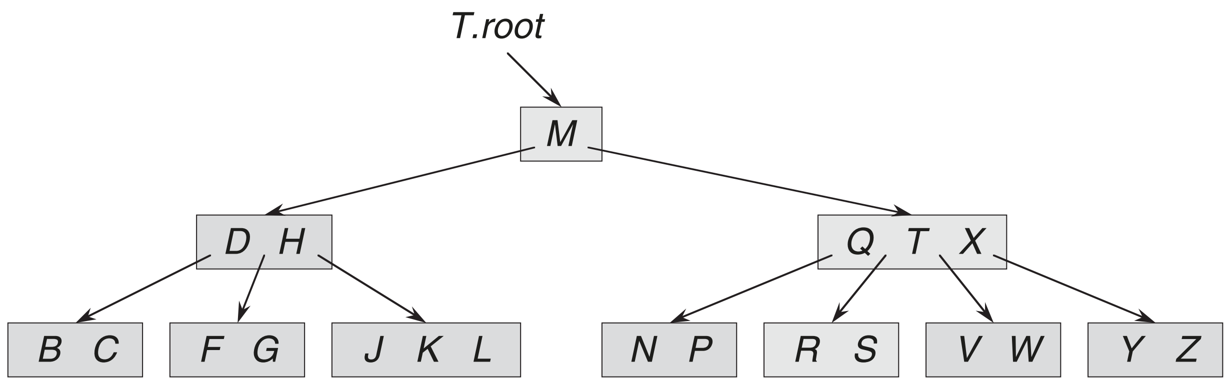
  Die Erfinder lieferten keine Erklärung über die Herkunft des Namens B-Baum. Die häufigste Interpretation ist, dass B für balanciert steht. Weitere Interpretationen sind B für Bayer, Barbara (nach seiner Frau), Broad, "Busch", Bushy, Boeing, da Rudolf Bayer für Boeing Scientific Research Labs gearbeitet hat, Banyanbaum, ein Baum bei dem Äste und Wurzeln ein Netz erstellen oder binär aufgrund der ausgeführten binären Suche innerhalb eines Knotens. [Wikipedia]

- Beachten Sie, dass B-Trees hier als Out-Trees modelliert sind; Knoten haben kein Elter-Attribut.

B-Tree

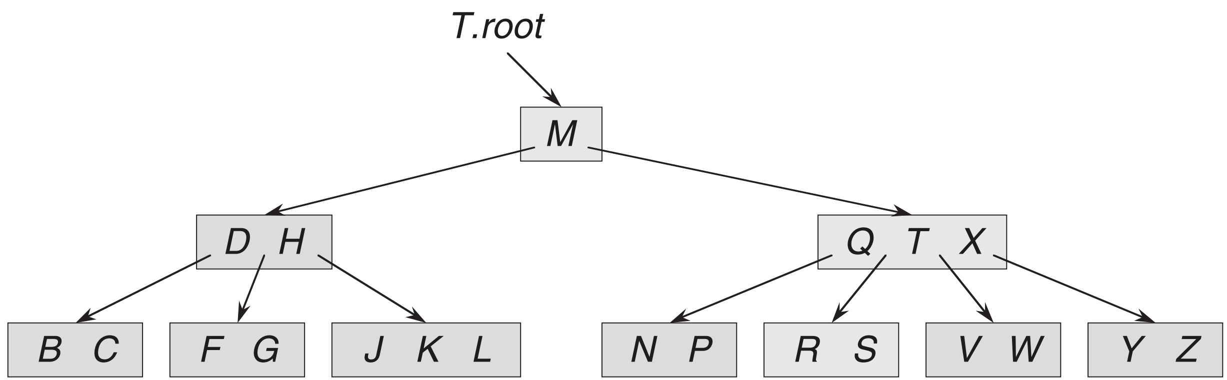
Definition

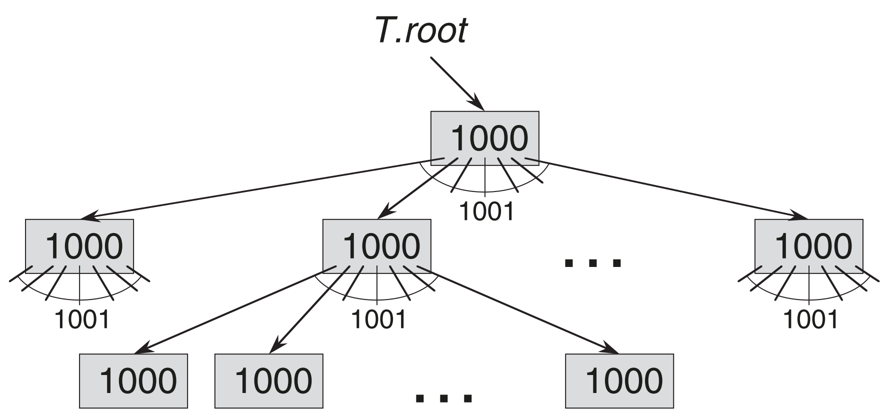
Beispiele:



B-Tree
Definition

Beispiele:





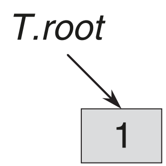
B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:



Tiefe	Knoten
0	1

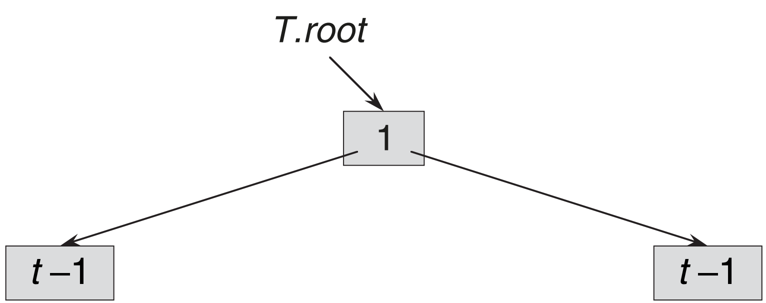
B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:



Tiefe	Knoten
0	1
1	2

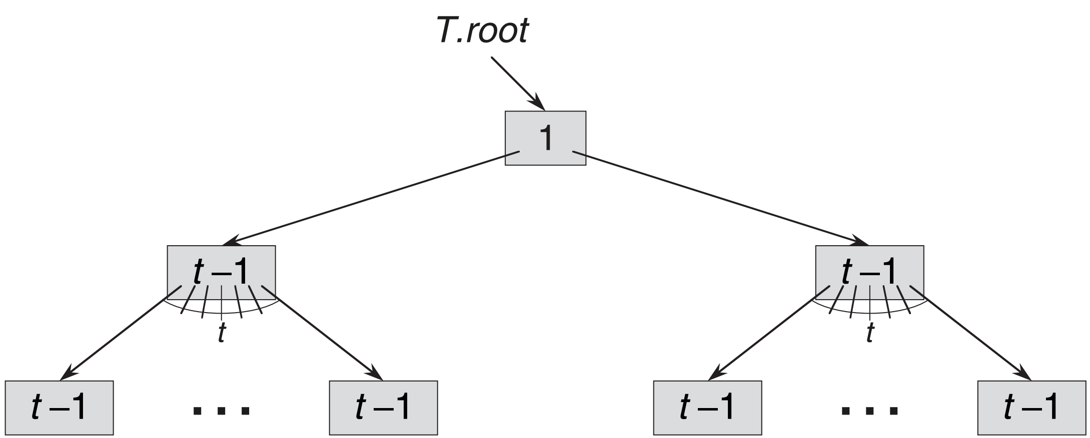
B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:



Tiefe	Knoten
0	1
1	2
2	$2t$

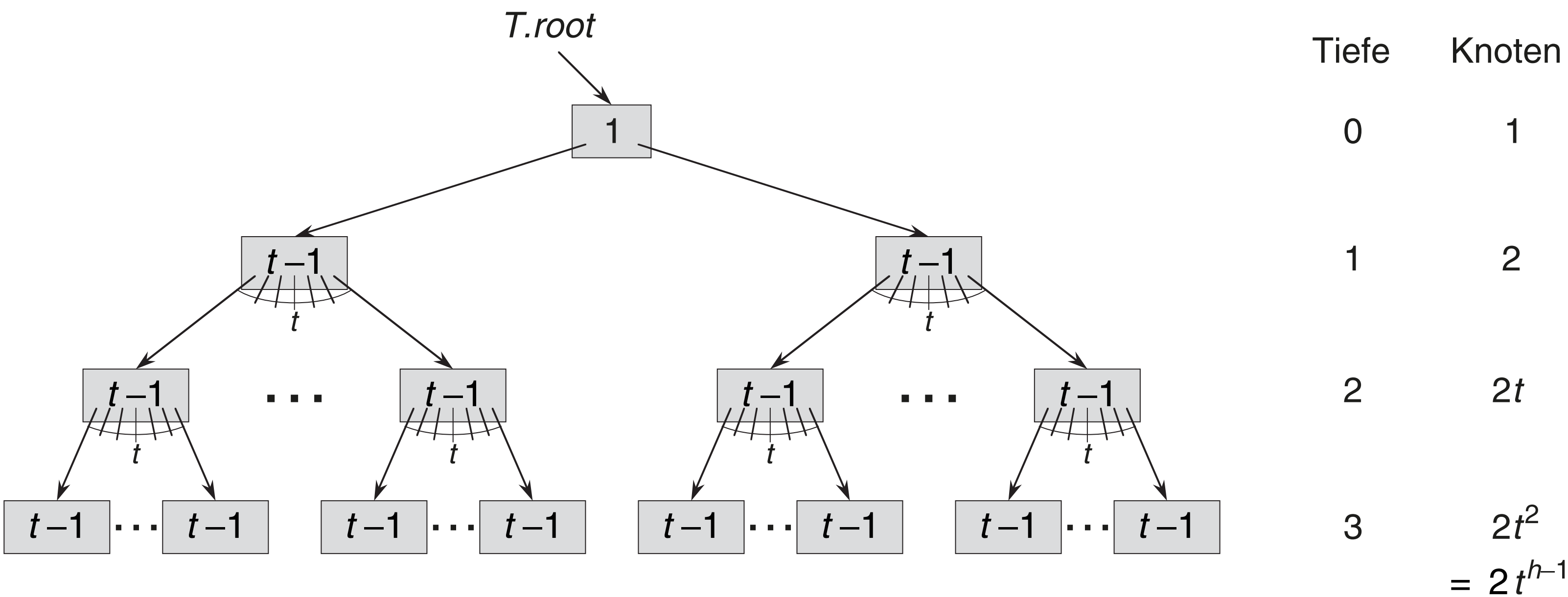
B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:



B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:

Sei $n(h)$ die Zahl der Schlüssel in einem B-Tree der Höhe h:

$$n(h) \;\geq\; 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:

Sei $n(h)$ die Zahl der Schlüssel in einem B-Tree der Höhe h:

$$\begin{aligned} n(h) &\geq 1 + (t-1) \sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \end{aligned}$$

B-Tree

Satz 1

Für einen B-Tree T der Höhe h mit n Schlüsseln und minimalem Knotengrad $t \geq 2$ gilt

$$h \leq \log_t \frac{n+1}{2}$$

Beweis:

Sei $n(h)$ die Zahl der Schlüssel in einem B-Tree der Höhe h:

$$\begin{aligned}
n(h) &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\
&= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\
&= 2t^h - 1 \\
\Leftrightarrow \quad t^h &\geq \frac{n(h)+1}{2} \\
\Leftrightarrow \quad h &\geq \log_t \frac{n(h)+1}{2} \quad \square
\end{aligned}$$

B-Tree

Konstruktion

Algorithmus: B-Tree Create.

Eingabe: keine

Ausgabe: Initialisierter B-Tree T ohne Schlüssel.

BTreeCreate$()$

$T =$ *tree*$()$
$x =$ *node*$()$
$x.$*leaf* $=$ *TRUE*
$x.n = 0$
write(x)
$T.$*root* $= x$
return(T)

Hilfsfunktionen:

- *tree* alloziert ein Objekt im Hauptspeicher, das den Baum T repräsentiert.
- *node* alloziert ein Objekt im Hauptspeicher, das einen Knoten repräsentiert.

Laufzeit:

- Latenzzeit: $O(1)$
- Rechenzeit: $O(1)$

B-Tree
Manipulation

- Schlüssel in Sortierreihenfolge besuchen
  Traversierung des Baumes mit DFS-Traverse (in-order; angepasst für k-näre Bäume).

- Schlüssel suchen (*Search*)
  Einen Knoten mit vorgegebenem Schlüssel suchen.

- Schlüssel einfügen (*Insert*)
  Einen Knoten an der richtigen Stelle im Baum einfügen.

- Schlüssel löschen (*Delete*)
  Einen bestimmten Knoten aus dem Baum löschen.

Konventionen:

- Die Wurzel eines B-Tree befindet sich immer im Hauptspeicher.

- Als Parameter übergebene Knoten müssen vorher in den Hauptspeicher geladen worden sein.

B-Tree

Manipulation: Suche

Algorithmus: B-Tree Search.

Eingabe: x. Wurzel eines B-Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Tupel (y, i), wobei y ein Knoten ist, so dass $y.\mathit{key}_i = k$, oder *NIL*.

B-Tree

Manipulation: Suche

Algorithmus: B-Tree Search.

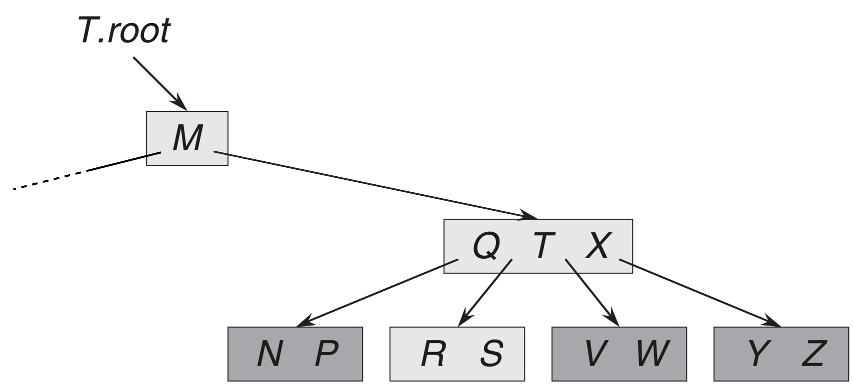
Eingabe: x. Wurzel eines B-Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Tupel (y, i), wobei y ein Knoten ist, so dass $y.\textit{key}_i = k$, oder *NIL*.

BTreeSearch(x, k)

WHILE $i \leq x.n$ **AND** $k > x.\textit{key}_i$ **DO**
  $i = i + 1$
ENDDO
IF $i < x.n$ **AND** $k == x.\textit{key}_i$ **THEN**
  return(x, i)
ELSE IF $x.\textit{leaf}$ **THEN**
  return(*NIL*)
ELSE
  read$(x.c_i)$
  return(*BTreeSearch*$(x.c_i, k)$)
ENDIF

Beispiel: *BTreeSearch*$(T.\textit{root}, R)$



B-Tree

Manipulation: Suche

Algorithmus: B-Tree Search.

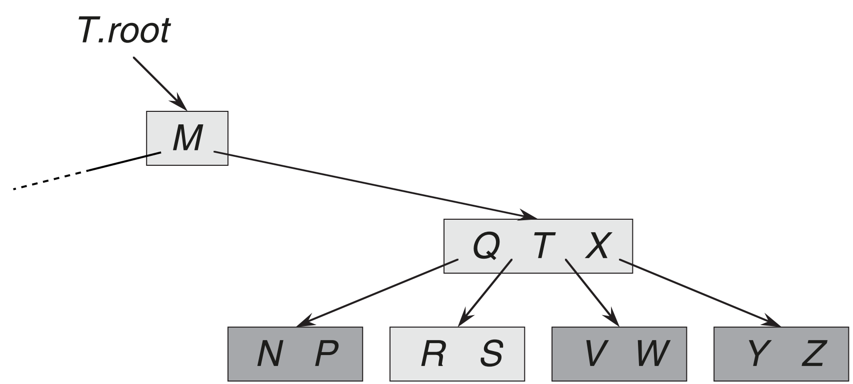
Eingabe: x. Wurzel eines B-Tree T.
k. Gesuchter Schlüssel.

Ausgabe: Tupel (y, i), wobei y ein Knoten ist, so dass $y.\mathit{key}_i = k$, oder *NIL*.

BTreeSearch(x, k)

```
i = 1
WHILE i ≤ x.n AND k > x.key_i DO
   i = i + 1
ENDDO
IF i < x.n AND k == x.key_i THEN
   return(x, i)
ELSE IF x.leaf THEN
   return(NIL)
ELSE
   read(x.c_i)
   return(BTreeSearch(x.c_i, k))
ENDIF
```

Beispiel: *BTreeSearch*$(T.\mathit{root}, R)$



Laufzeit

- Latenzzeit: $O(h) = O(\log_t n)$
- Rechenzeit: $O(th) = O(t \log_t n)$

B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Insert.

Eingabe: T. B-Tree.
k. Einzufügender Schlüssel.

Ausgabe: Um k erweiterter B-Tree.

B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Insert.

Eingabe: T. B-Tree.
k. Einzufügender Schlüssel.

Ausgabe: Um k erweiterter B-Tree.

Vorüberlegungen:

- Es gibt keine 1:1-Korrespondenz zwischen Schlüsseln und Knoten.
  Ein Knoten hat zwischen $t-1$ und $2t-1$ Schlüssel.
- k wird im passenden Blattknoten hinzugefügt, solange Platz ist.
  Beim Hinzufügen muss die Sortierreihenfolge beachtet werden.
- Wenn ein Blatt voll ist, darf kein neues Kind erzeugt werden.
  Gemäß Bedingung 3 müssen alle Blätter müssen auf derselben Ebene sein.
- Volle Blattknoten werden geteilt; zwei neue Blattknoten entstehen.
  Der Median-Schlüssel wird dem Elter hinzugefügt, um die neuen Blätter zu unterscheiden.
- Die Teilung eines Blattes kann die Teilung des (vollen) Elters erfordern.
  Statt darauf zu warten, wird ein Knoten „im Vorbeigehen“ geteilt, falls er voll ist.

B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Insert.

Eingabe: T. B-Tree.
k. Einzufügender Schlüssel.

Ausgabe: Um k erweiterter B-Tree.

BTreeInsert(T, k)

```
r = T.root
IF r.n == 2t − 1 THEN
    s = node()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    BTreeSplitChild(s, 1)
    BTreeInsertNonfull(s, k)
ELSE
    BTreeInsertNonfull(r, k)
ENDIF
```

Vorgehen:

- Sonderfall: Wurzel voll
  - Neue Wurzel erzeugen
  - Alte Wurzel teilen
  - Rekursiv fortfahren
- Sonst: Rekursiv fortfahren

B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Split Child.

Eingabe: x. Teilbaum eines B-Tree mit Wurzel x.
i. Index des zu teilenden Kindes von x, das $2t-1$ Schlüssel enthält.

Ausgabe: B-Tree mit Wurzel x, dessen i-tes Kind geteilt ist.

Vorgehen:

- Sei y das zu teilende i-te Kind von x.
- Sei z ein neuer Knoten, der die Hälfte von ys Schlüssen aufnehmen soll.
- Kopiere die hinteren $t-1$ Schlüssel und t Knoten von y nach z.
- Setze ys Schlüsselzähler zurück auf $t-1$.
  Die vorigen Einträge $> t-1$ werden nicht überschrieben.
- Schaffe Platz für z als neues Kind in x rechts neben y.
- Schaffe Platz für ys t-ten Schlüssel als neuen i-ten Schlüssel von x.
- Schreibe x, y und z in den Sekundärspeicher.

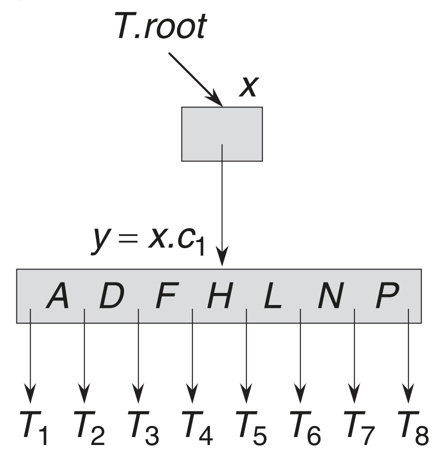
B-Tree

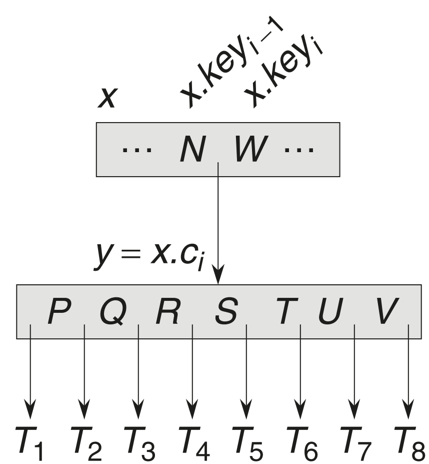
Manipulation: Einfügen

BTreeSplitChild(x, i)

$y = x.c_i$
$z = node()$
$z.leaf = y.leaf$
$z.n = t - 1$
FOR $j = 1$ **TO** $t - 1$ **DO**
    $z.key_j = y.key_{j+t}$
ENDDO
IF NOT $y.leaf$ **THEN**
    FOR $j = 1$ **TO** t **DO**
        $z.c_j = y.c_{j+t}$
    ENDDO
ENDIF
$y.n = t - 1$
FOR $j = x.n + 1$ **DOWNTO** $i + 1$ **DO**
    $x.c_{j+1} = x.c_j$
ENDDO
$x.c_{i+1} = z$
FOR $j = x.n$ **DOWNTO** i **DO**
    $x.key_{j+1} = x.key_j$
ENDDO
$x.key_i = y.key_t$
$x.n = x.n + 1$
$write(x)$; $write(y)$; $write(z)$

Fälle: (t=4)





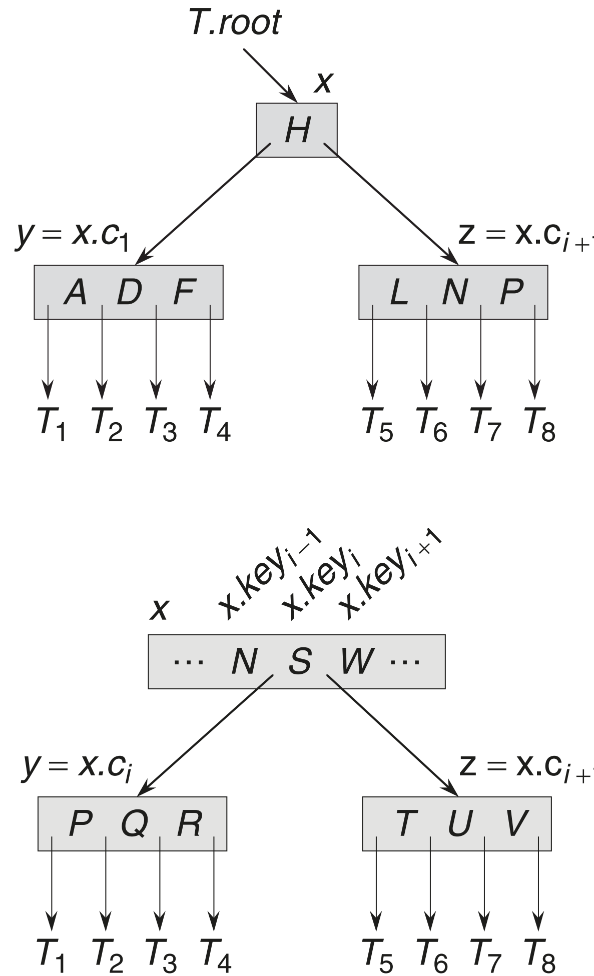
B-Tree

Manipulation: Einfügen

BTreeSplitChild(x, i)

```
y = x.c_i
z = node()
z.leaf = y.leaf
z.n = t - 1
FOR j = 1 TO t - 1 DO
   z.key_j = y.key_{j+t}
ENDDO
IF NOT y.leaf THEN
   FOR j = 1 TO t DO
      z.c_j = y.c_{j+t}
   ENDDO
ENDIF
y.n = t - 1
FOR j = x.n + 1 DOWNTO i + 1 DO
   x.c_{j+1} = x.c_j
ENDDO
x.c_{i+1} = z
FOR j = x.n DOWNTO i DO
   x.key_{j+1} = x.key_j
ENDDO
x.key_i = y.key_t
x.n = x.n + 1
write(x);  write(y);  write(z)
```

Fälle: (t=4)



B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Insert Nonfull.

Eingabe: x. Teilbaum eines B-Tree mit Wurzel x.
i. Index des zu teilenden Kindes von x, das $2t - 1$ Schlüssel enthält.

Ausgabe: B-Tree mit Wurzel x, dessen i-tes Kind geteilt ist.

Fallunterscheidung:

1. x ist ein Blatt: (x ist nicht voll; siehe Fall 2)
   - Verschiebe alle Schlüssel $> k$ um eine Indexposition nach oben.
   - Füge k an die freie Stelle ein und aktualisiere die Zahl der Schlüssel.
   - Schreibe x in den Sekundärspeicher.
2. x ist ein innerer Knoten:
   - Suche den Index i des Kindes von x, in dessen Teilbaum k einzufügen ist.
   - Lese $x.c_i$ aus dem Sekundärspeicher.
   - Falls $x.c_i$ voll ist, teile den Knoten; fahre rekursiv bei $x.c_i$ fort.

B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Insert Nonfull.

Eingabe: x. Teilbaum eines B-Tree mit Wurzel x.
i. Index des zu teilenden Kindes von x, das $2t-1$ Schlüssel enthält.

Ausgabe: B-Tree mit Wurzel x, dessen i-tes Kind geteilt ist.

Fallunterscheidung:

1. x ist ein Blatt: (x ist nicht voll; siehe Fall 2)
   - Verschiebe alle Schlüssel $> k$ um eine Indexposition nach oben.
   - Füge k an die freie Stelle ein und aktualisiere die Zahl der Schlüssel.
   - Schreibe x in den Sekundärspeicher.
2. x ist ein innerer Knoten:
   - Suche den Index i des Kindes von x, in dessen Teilbaum k einzufügen ist.
   - Lese $x.c_i$ aus dem Sekundärspeicher.
   - Falls $x.c_i$ voll ist, teile den Knoten; fahre rekursiv bei $x.c_i$ fort.

B-Tree

Manipulation: Einfügen

Algorithmus: B-Tree Insert Nonfull.

Eingabe: x. Teilbaum eines B-Tree mit Wurzel x.
i. Index des zu teilenden Kindes von x, das $2t-1$ Schlüssel enthält.

Ausgabe: B-Tree mit Wurzel x, dessen i-tes Kind geteilt ist.

Fallunterscheidung:

1. x ist ein Blatt: (x ist nicht voll; siehe Fall 2)
   - Verschiebe alle Schlüssel $> k$ um eine Indexposition nach oben.
   - Füge k an die freie Stelle ein und aktualisiere die Zahl der Schlüssel.
   - Schreibe x in den Sekundärspeicher.
2. x ist ein innerer Knoten:
   - Suche den Index i des Kindes von x, in dessen Teilbaum k einzufügen ist.
   - Lese $x.c_i$ aus dem Sekundärspeicher.
   - Falls $x.c_i$ voll ist, teile den Knoten; fahre rekursiv bei $x.c_i$ fort.

B-Tree

Manipulation: Einfügen

BTreeInsertNonfull(x, k)

$i = x.n$
IF $x.\textit{leaf}$ **THEN**
  WHILE $i \geq 1$ **AND** $k < x.\textit{key}_i$ **DO**
    $x.\textit{key}_{i+1} = x.\textit{key}_i$
    $i = i - 1$
  ENDDO
  $x.\textit{key}_{i+1} = k$
  $x.n = x.n + 1$
  write(x)
ELSE
  WHILE $i \geq 1$ **AND** $k < x.\textit{key}_i$ **DO**
    $i = i - 1$
  ENDDO
  $i = i + 1$
  read$(x.c_i)$
  IF $x.c_i.n == 2t - 1$ **THEN**
    BTreeSplitChild(x, i)
    IF $k > x.\textit{key}_i$ **THEN**
      $i = i + 1$
    ENDIF
  ENDIF
  BTreeInsertNonfull$(x.c_i, k)$
ENDIF

B-Tree

Manipulation: Einfügen

BTreeInsertNonfull(x, k)

$i = x.n$
IF $x.\mathit{leaf}$ **THEN**
  WHILE $i \geq 1$ **AND** $k < x.\mathit{key}_i$ **DO**
    $x.\mathit{key}_{i+1} = x.\mathit{key}_i$
    $i = i - 1$
  ENDDO
  $x.\mathit{key}_{i+1} = k$
  $x.n = x.n + 1$
  write(x)
ELSE
  WHILE $i \geq 1$ **AND** $k < x.\mathit{key}_i$ **DO**
    $i = i - 1$
  ENDDO
  $i = i + 1$
  read$(x.c_i)$
  IF $x.c_i.n == 2t - 1$ **THEN**
    BTreeSplitChild(x, i)
    IF $k > x.\mathit{key}_i$ **THEN**
      $i = i + 1$
    ENDIF
  ENDIF
  BTreeInsertNonfull$(x.c_i, k)$
ENDIF

B-Tree

Manipulation: Einfügen

BTreeInsertNonfull(x, k)

```
i = x.n
IF x.leaf THEN
  WHILE i ≥ 1 AND k < x.key_i DO
    x.key_{i+1} = x.key_i
    i = i − 1
  ENDDO
  x.key_{i+1} = k
  x.n = x.n + 1
  write(x)
ELSE
  WHILE i ≥ 1 AND k < x.key_i DO
    i = i − 1
  ENDDO
  i = i + 1
  read(x.c_i)
  IF x.c_i.n == 2t − 1 THEN
    BTreeSplitChild(x, i)
    IF k > x.key_i THEN
      i = i + 1
    ENDIF
  ENDIF
  BTreeInsertNonfull(x.c_i, k)
ENDIF
```

B-Tree

Manipulation: Einfügen

BTreeInsertNonfull(x, k)

```
i = x.n
IF x.leaf THEN
  WHILE i ≥ 1 AND k < x.key_i DO
    x.key_{i+1} = x.key_i
    i = i − 1
  ENDDO
  x.key_{i+1} = k
  x.n = x.n + 1
  write(x)
ELSE
  WHILE i ≥ 1 AND k < x.key_i DO
    i = i − 1
  ENDDO
  i = i + 1
  read(x.c_i)
  IF x.c_i.n == 2t − 1 THEN
    BTreeSplitChild(x, i)
    IF k > x.key_i THEN
      i = i + 1
    ENDIF
  ENDIF
  BTreeInsertNonfull(x.c_i, k)
ENDIF
```

Beobachtungen:

- Knotenteilung ist die einzige Ursache für Wachstum.
- Wachstum geschieht nur an der Wurzel.

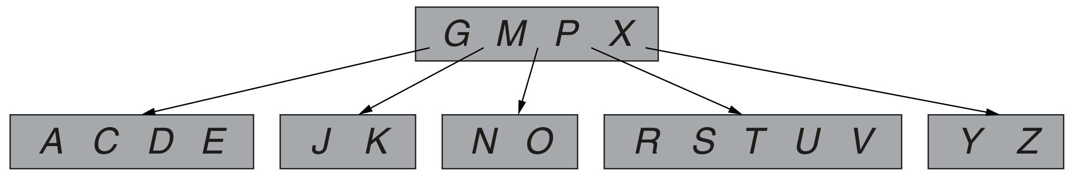
Laufzeit:

- Latenzzeit: $O(h) = O(\log_t n)$
- Rechenzeit: $O(th) = O(t \log_t n)$

B-Tree

Manipulation: Einfügen

Beispiel: (t=3)



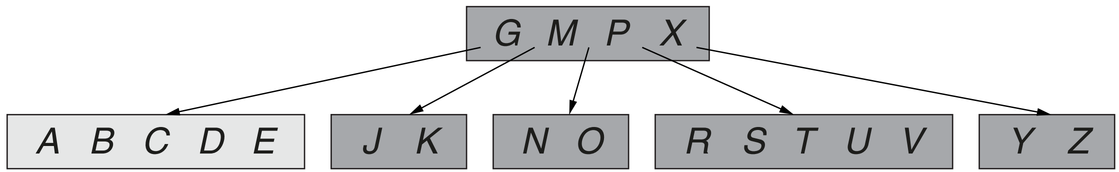
Nächste Operation:

- B einfügen

B-Tree

Manipulation: Einfügen

Beispiel: (t=3)



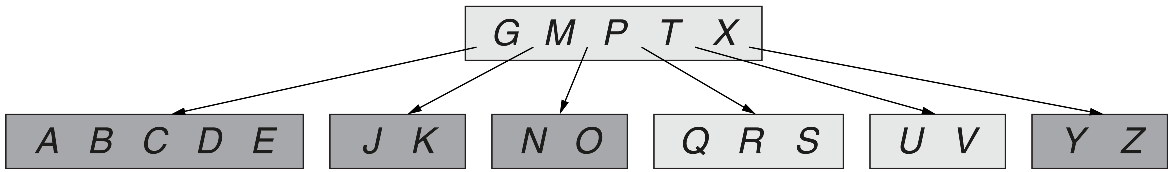
Nächste Operation:

- B einfügen
  B wird in Blatt $ACDE$ eingefügt.
- Q einfügen

B-Tree

Manipulation: Einfügen

Beispiel: (t=3)



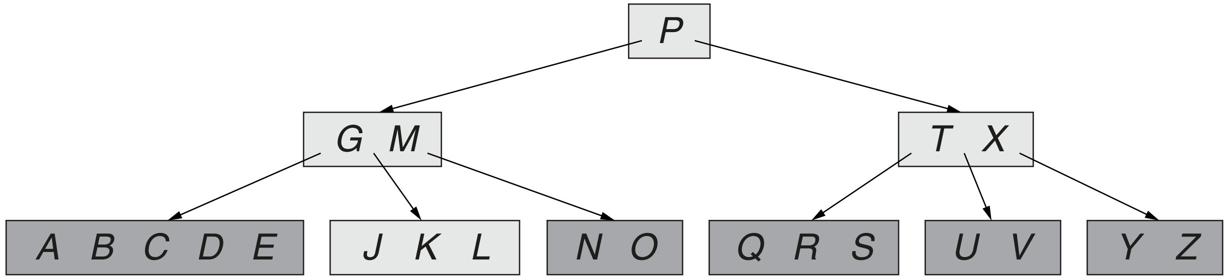
Nächste Operation:

- B einfügen
  B wird in Blatt $ACDE$ eingefügt.
- Q einfügen
  Blatt $RSTUV$ ist voll und wird geteilt in RS und UV, wobei T zur Wurzel hinzugefügt wird. Q wird daraufhin in Blatt RS hinzugefügt.
- L einfügen

B-Tree

Manipulation: Einfügen

Beispiel: (t=3)



Nächste Operation:

- B einfügen

  B wird in Blatt $ACDE$ eingefügt.

- Q einfügen

  Blatt $RSTUV$ ist voll und wird geteilt in RS und UV, wobei T zur Wurzel hinzugefügt wird. Q wird daraufhin in Blatt RS hinzugefügt.

- L einfügen

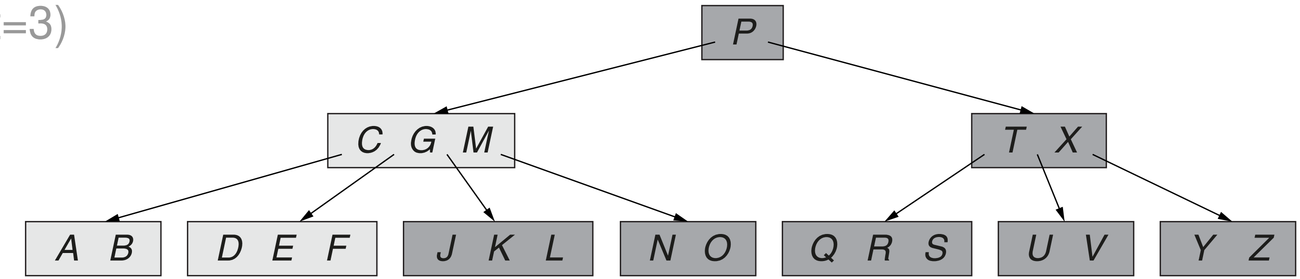
  Die Wurzel ist voll und wird geteilt in GM und TX, wobei P die neue Wurzel wird. L wird daraufhin in Blatt JK eingefügt.

- F einfügen

B-Tree

Manipulation: Einfügen

Beispiel: (t=3)



Nächste Operation:

- B einfügen

  B wird in Blatt $ACDE$ eingefügt.

- Q einfügen

  Blatt $RSTUV$ ist voll und wird geteilt in RS und UV, wobei T zur Wurzel hinzugefügt wird. Q wird daraufhin in Blatt RS hinzugefügt.

- L einfügen

  Die Wurzel ist voll und wird geteilt in GM und TX, wobei P die neue Wurzel wird. L wird daraufhin in Blatt JK eingefügt.

- F einfügen

  Blatt $ABCDE$ ist voll und wird geteilt in AB und DE, wobei C dem Elter hinzugefügt wird. F wird daraufhin in Blatt DE eingefügt.

B-Tree

Manipulation: Löschen

Algorithmus: B-Tree Delete.

Eingabe: x. Wurzel des Teilbaums, in dem k zu finden ist.
k. Zu löschender Schlüssel.

Ausgabe: Um k reduzierter B-Tree.

B-Tree

Manipulation: Löschen

Algorithmus: B-Tree Delete.

Eingabe: x. Wurzel des Teilbaums, in dem k zu finden ist.
k. Zu löschender Schlüssel.

Ausgabe: Um k reduzierter B-Tree.

Vorüberlegungen:

- k wird aus dem Knoten gelöscht, in dem er sich befindet.
  Befindet sich k in einem inneren Knoten, muss der Baum zunächst rekonfiguriert werden, damit das Entfernen von k möglich wird.
- Rekonfigurationsoperation: Schlüssel verschieben.
  Ein Schlüssel k kann durch seinen Vorgängern oder Nachfolger ersetzt werden. Eine Art Rotation von Schlüsseln aus Geschwisterknoten kann durchgeführt werden.
- Rekonfigurationsoperation: Knoten verschmelzen.
  Wenn Geschwisterknoten die Minimalzahland Schlüsseln enthalten, erlaubt nur ihre Verschmelzung die für das Löschen notwendigen Verschiebungen.
- Die Mindestzahl an Schlüssel wird auf t festgelegt (nicht $t-1$).
  Beim rekursiven Abstieg wird sichergestellt, dass alle Knoten mindestens t Schlüssel haben bzw. der Baum entsprechend rekonfiguriert.

B-Tree

Manipulation: Löschen

BTreeDelete(x, k)

1. k ist in x und x ist ein Blatt: Lösche k.

BTreeDelete(x, k)

1. k ist in x und x ist ein Blatt: Lösche k.
2. k ist in x und x ist innerer Knoten:
   (a) Kind y, das Schlüssel k vorangeht, hat mindestens t Schlüssel
   Finde und lösche k's Vorgänger k' und ersetze k durch k'.
   (b) Kind z, das Schlüssel k folgt, hat mindestens t Schlüssel
   Finde und lösche k's Nachfolger k' und ersetze k durch k'.
   (c) y und z haben $t-1$ Schlüssel
   Verschmelze z mit y mit k als Median; lösche z und fahre rekursiv bei y fort.

B-Tree

Manipulation: Löschen

BTreeDelete(x, k)

1. k ist in x und x ist ein Blatt: Lösche k.
2. k ist in x und x ist innerer Knoten:
   (a) Kind y, das Schlüssel k vorangeht, hat mindestens t Schlüssel
   Finde und lösche k's Vorgänger k' und ersetze k durch k'.
   (b) Kind z, das Schlüssel k folgt, hat mindestens t Schlüssel
   Finde und lösche k's Nachfolger k' und ersetze k durch k'.
   (c) y und z haben $t - 1$ Schlüssel
   Verschmelze z mit y mit k als Median; lösche z und fahre rekursiv bei y fort.
3. k ist nicht in x und x ist innerer Knoten:
   (a) Kind y, Wurzel des Teilbaums, der k enthalten müsste, hat $\geq$ Schlüssel
   Fahre rekursiv bei y fort.
   (b) y hat einen linken oder rechten Geschwister z mit mindestens t Schlüsseln
   Ersetze k durch Verschieben von Schlüssel k', der x von z trennt aus x nach y und ersetze k' durch seinen Vorgängern/Nachfolger k'' aus z. Fahre rekursiv bei y fort.
   (c) Beide direkten Geschwister von y haben $t - 1$ Schlüssel
   Verschmelze y mit einem seiner Geschwister. Fahre rekursive bei y fort.

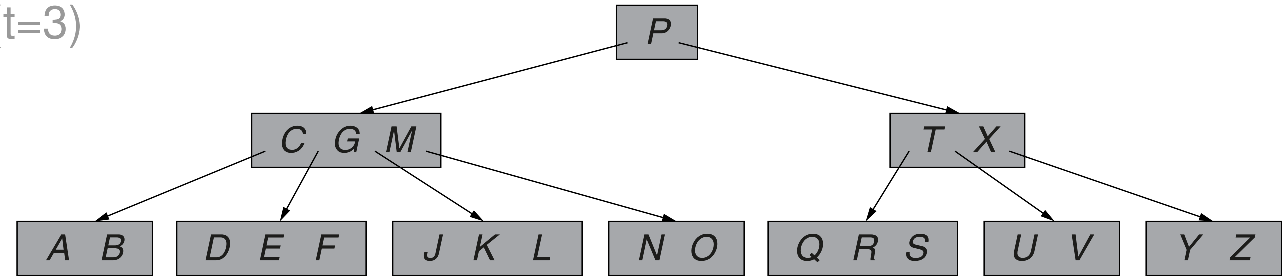
Bemerkungen:

- Laufzeit:
  - Latenzzeit: $O(h) = O(\log_t n)$
  - Rechenzeit: $O(th) = O(t \log_t n)$

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



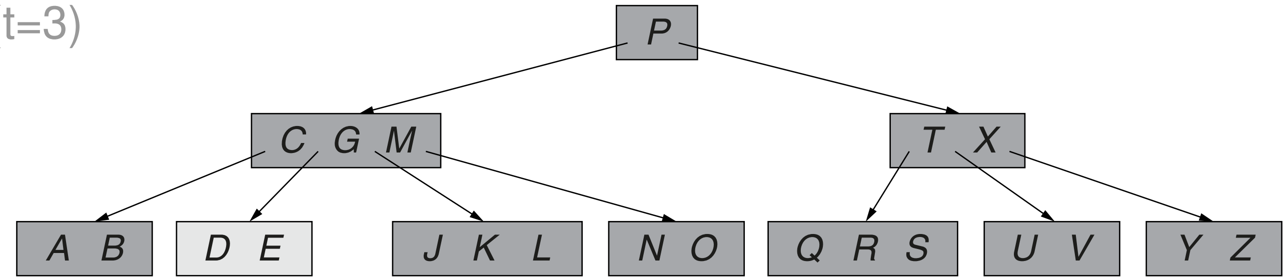
Nächste Operation:

- F löschen

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



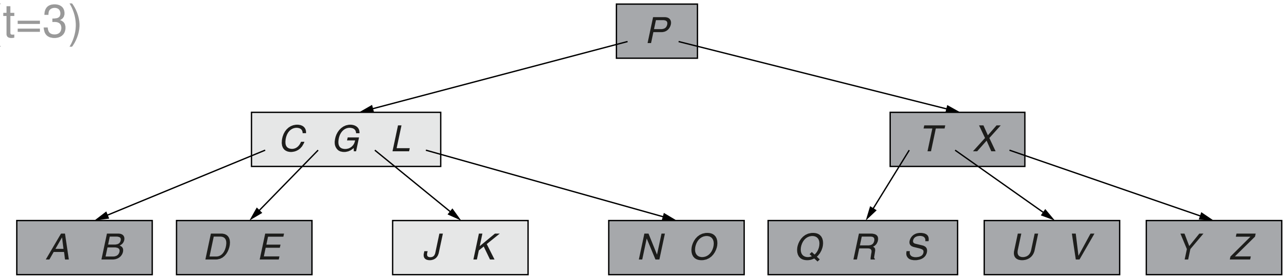
Nächste Operation:

- F löschen
  Fall 1: F wird gelöscht.
- M löschen

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



Nächste Operation:

- F löschen

  Fall 1: F wird gelöscht.

- M löschen

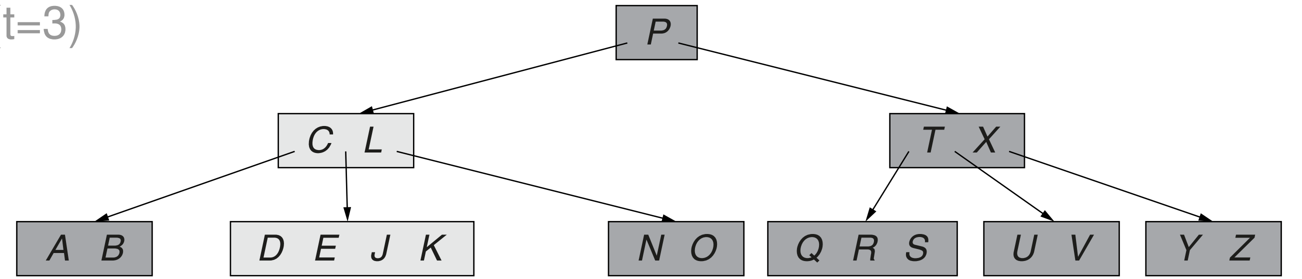
  Fall 2a: M wird durch Vorgänger L ersetzt.

- G löschen

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



Nächste Operation:

- F löschen

  Fall 1: F wird gelöscht.

- M löschen

  Fall 2a: M wird durch Vorgänger L ersetzt.

- G löschen

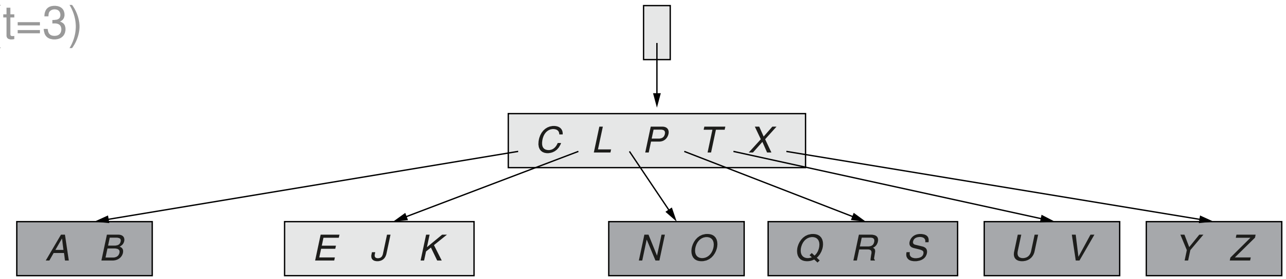
  Fall 2c: DE und JK werden vereint; daraufhin G gelöscht.

- D löschen

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



Nächste Operation:

- F löschen

  Fall 1: F wird gelöscht.

- M löschen

  Fall 2a: M wird durch Vorgänger L ersetzt.

- G löschen

  Fall 2c: DE und JK werden vereint; daraufhin G gelöscht.

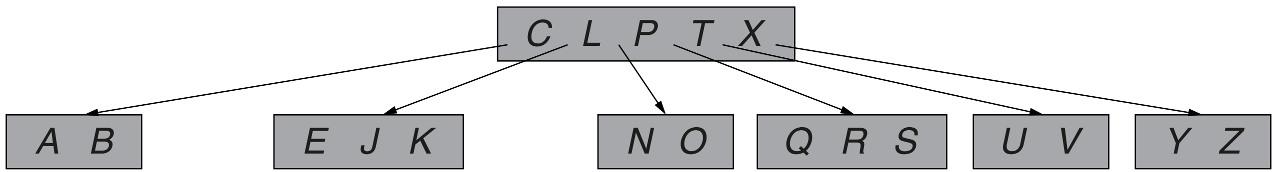
- D löschen

  Fall 3c: CL, P und TX werden vereint und zur neuen Wurzel; daraufhin D gelöscht.

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



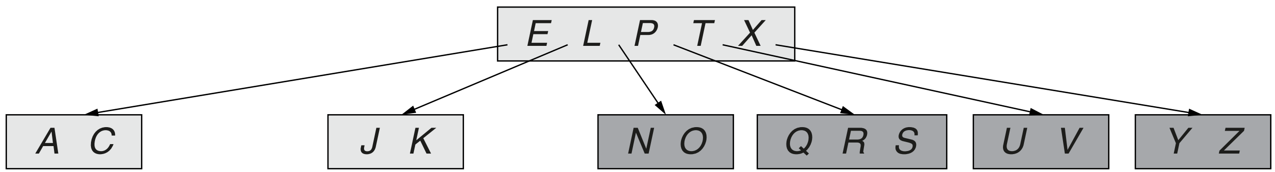
Nächste Operation:

- F löschen
  Fall 1: F wird gelöscht.
- M löschen
  Fall 2a: M wird durch Vorgänger L ersetzt.
- G löschen
  Fall 2c: DE und JK werden vereint; daraufhin G gelöscht.
- D löschen
  Fall 3c: CL, P und TX werden vereint und zur neuen Wurzel; daraufhin D gelöscht.
- B löschen

B-Tree

Manipulation: Löschen

Beispiel: (t=3)



Nächste Operation:

- F löschen
  Fall 1: F wird gelöscht.
- M löschen
  Fall 2a: M wird durch Vorgänger L ersetzt.
- G löschen
  Fall 2c: DE und JK werden vereint; daraufhin G gelöscht.
- D löschen
  Fall 3c: CL, P und TX werden vereint und zur neuen Wurzel; daraufhin D gelöscht.
- B löschen
  Fall 3b: B wird durch C und C durch E ersetzt.