Minimum und Maximum

```
TREE-MINIMUM(x)
    while left[x] ≠ NIL
        do x ← left[x]
    return x
```

```
TREE-MAXIMUM(x)
    while right[x] ≠ NIL
        do x ← right[x]
    return x
```

Laufzeit: $O(h)$

Successor

TREE-SUCCESSOR(x)
if right$[x] \neq$ NIL
  then return TREE-MINIMUM (**right**$[x]$)
$y \leftarrow p\,[x]$
while $y \neq$ NIL and $x =$ **right**$[y]$
    do $x \leftarrow y$
      $y \leftarrow p[y]$
return y

Predecessor analog.

Änderungsoperationen

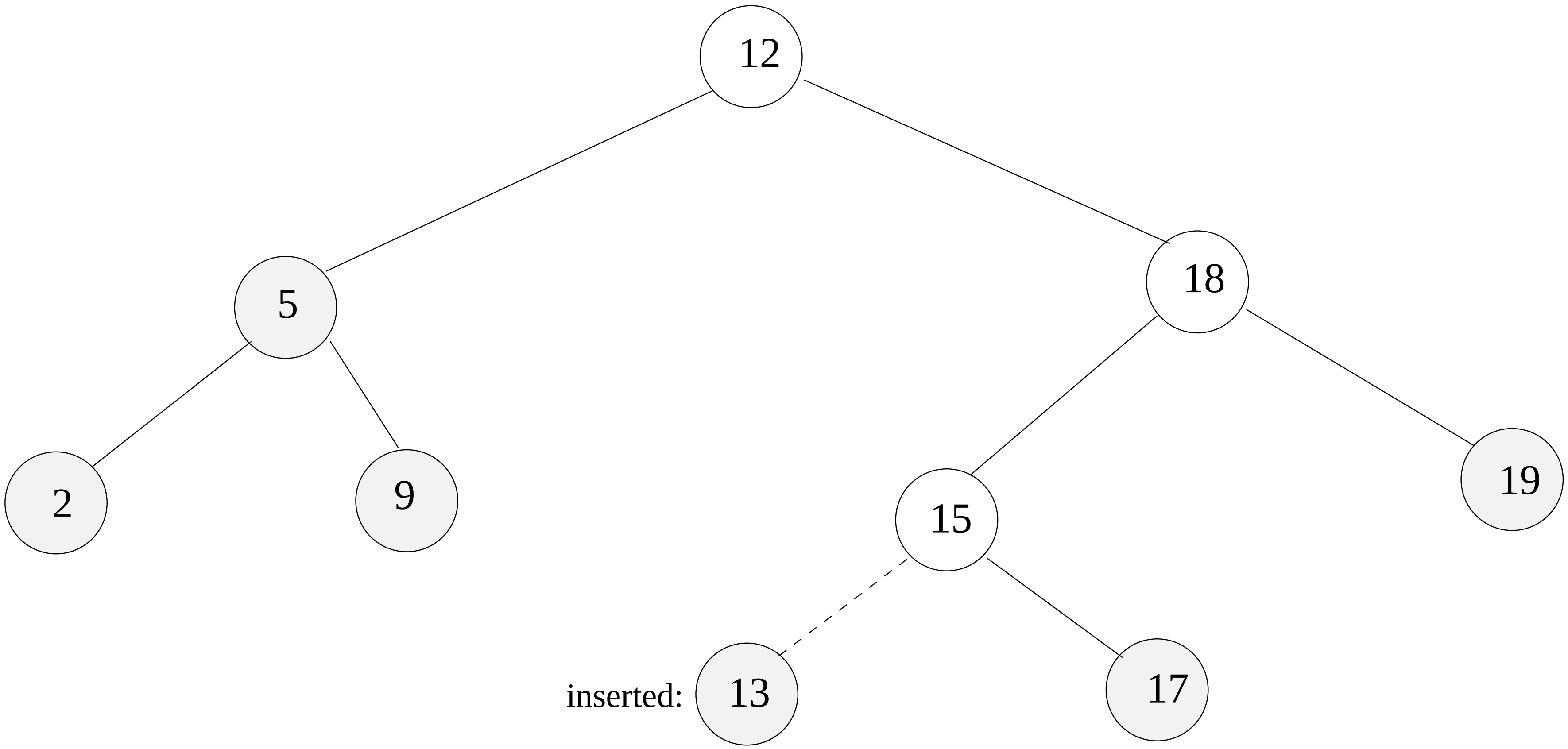
Änderungen sind

- insert
- delete

Diese müssen die binäre Suchbaumeigenschaft bewahren.
Beide werden Laufzeit $O(h)$ haben.

Insert

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
    then root[T] ← z
    else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
```


12
5
18
2
9
15
19
inserted:
13
17

Delete

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then root[T] ← x
    else if y = left[p[y]]
        then left[p[y]] ← x
        else right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
        ▷ If y has other fields, copy them, too.
return y
```

Die drei Fälle:

Rot-Schwarz-Baum

Motivation:

- einfacher binärer Suchbaum kann unbalanciert sein
- Jeder Knoten ist entweder Schwarz oder Rot.
- Nur bestimmte Farbmuster erlaubt. Dies garantiert:
- längster Pfad von Wurzel zu einem Blatt kann höchstens doppelt so lang sein wie kürzester Pfad von der Wurzel zu einem Blatt.
- Daher Höhe $O(\lg n)$ für n Elemente.

Rot-Schwarz-Baum

- Jeder Knoten hält die Attribute:
  color, key, left, right, p
- Falls kein Sohn oder Vater existiert enthält das entsprechende Feld NIL. Diese NILs sind für uns Zeiger auf externe Knoten (Blätter).
- Ein Binärer Suchbaum ist ein Rot-Schwarz-Baum, falls
  1. Jeder Knoten ist entweder Schwarz oder Rot.
  2. Jedes Blatt (NIL) ist schwarz.
  3. Falls Knoten rot, so sind beide Söhne schwarz.
  4. Jeder einfache Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten.

Rot-Schwarz-Baum

- Die Anzahl von schwarzen Knoten in einem Pfad von x, ohne x mitzuzählen, zu einem Blatt heißt schwarze Höhe (black-height) von x. ($\mathbf{bh}(x)$).
  Nach Eigenschaft 4 wohldefiniert.

Lemma Ein rot-schwarz-Baum mit n innere Knoten hat eine Höhe von maximal $2\lg(n+1)$.

Beweis Wir zeigen zunächst durch Induktion, daß ein Teilbaum mit Wurzel x mindestens $2^{bh(x)}-1$ innere Knoten hat.

height(x)=0: $\Longrightarrow$ x ist Blatt und Teilbaum von x enthält $2^0-1=0$ innere Knoten.

height(x)>0**:** Jeder Sohn von x hat entweder die schwarze Höhe $bh(x)$ oder $bh(x)-1$, abhängig von seiner Farbe. Also hat der Teilbaum von x mindestens

$$1+2*(2^{bk(x)-1}-1)=2^{bh(x)}-1$$

Knoten.

Sei h die Höhe des R-S-Baums. Eigenschaft 3 impliziert, daß mindestens die Hälfte der Knoten auf einem einfachen Pfad von der Wurzel zu einem Knoten schwarz sind. Also ist die schwarze Höhe mindestens $h/2$. Also

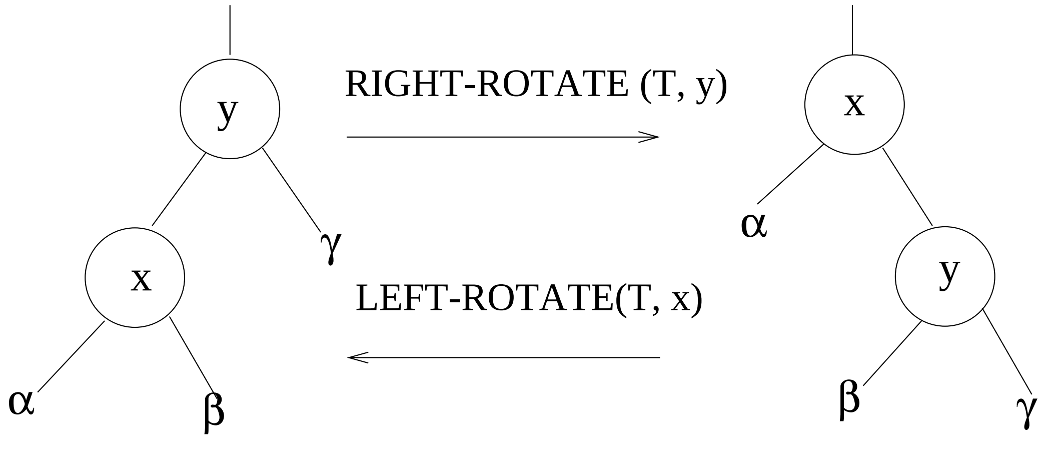
$$n \geq 2^{h/2}-1$$

und damit

$$h \leq 2\lg(n+1)$$

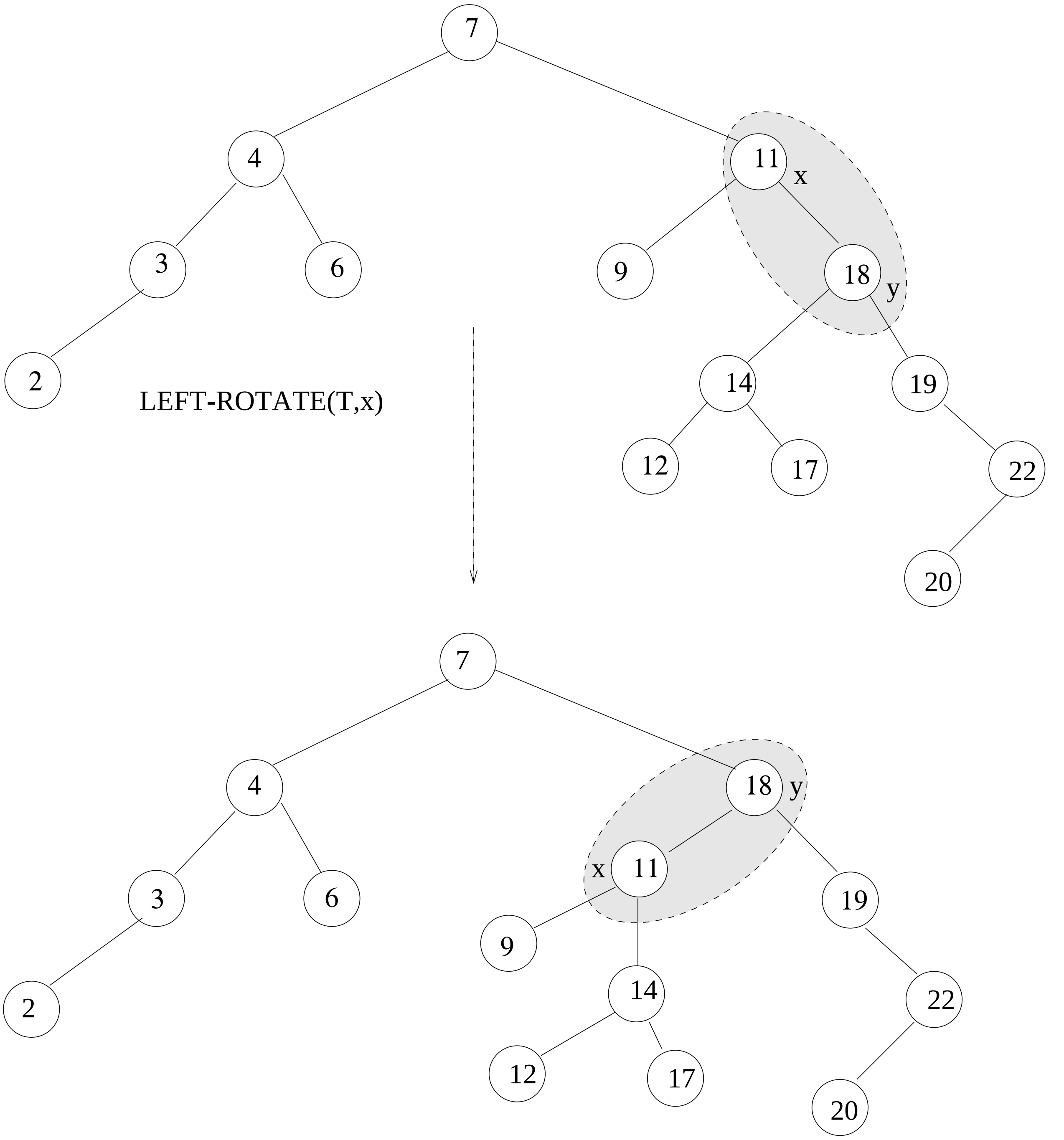
Rot-Schwarz-Baum

- Die Operationen SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR laufen in $O(\lg n)$, da $O(h)$ bereits nachgewiesen und ein rot-schwarz Baum mit n Knoten eine Höhe von $O(\lg n)$ hat.
- Die bisherigen INSERT und DELETE Operationen können aber die Eigenschaften zerstören.
- Wiederherstellen mit LEFT-ROTATE und RIGHT-ROTATE.
- Modifiziertes INSERT und DELETE nutzen dann diese Prozeduren.



Rotationen

```
LEFT-ROTATE(T, x)
y ← right[x]                    ▷ Set y
right[x] ← left[y]              ▷ y's left becomes x's right subtree.
if left[y] ≠ NIL
    then p[left[y]] ← x
p[y] ← p[x]                     ▷ Link x's parent to y.
if p[x] = NIL
    then root[T] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
left[y] ← x                     ▷ Put x on y's left.
p[x] ← y
```


7
4
11
x
3
6
9
18
y
2
LEFT-ROTATE(T,x)
14
19
12
17
22
20
7
4
18
y
x
11
3
6
19
9
2
14
22
12
17
20

RB-INSERT

1. Einfügen von x mit normalen TREE-INSERT
2. Setzen der Farbe von x auf rot
3. Korrigieren der Farbe beim Durchlaufen des Pfads von x zur Wurzel.
4. Dabei werden 2*drei Fälle unterschieden (x immer rot, p[x] rot):
   1. onkel[x] rot $\wedge$
      p[x] ::= onkel[x] ::= schwarz; grossvater[x] ::= rot;
      sprung zu grossvater[x]
   - onkel[x] schwarz
     2. x=right[p[x]]
        x:=p[x];LEFT-ROTATE von x
     3. x=left[p[x]]
        p[x] ::= schwarz; p[p[x]] ::= rot;
        RIGHT-ROTATE von p[p[x]]

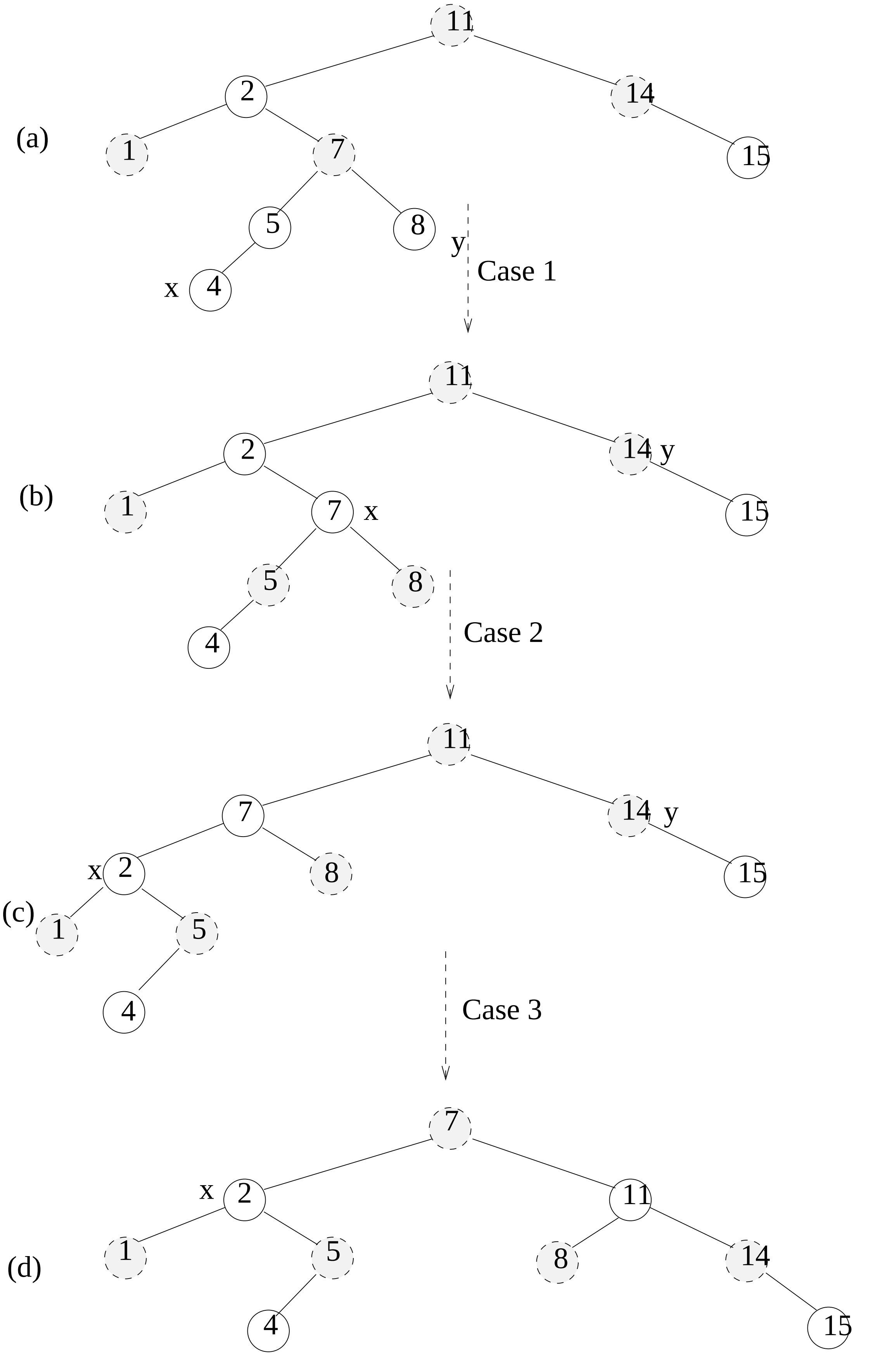
```
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED
   do if p[x] = left[p[p[x]]]
      then y ← right[p[p[x]]]
         if color[y] = RED
            then color[p[x]] ← BLACK          ▷Case1
               color[y] ← BLACK               ▷Case1
               color[p[p[x]]] ← RED           ▷Case1
               x ← p[p[x]]                    ▷Case1
            else if x = right[p[x]]
                  then x ← p[x]               ▷Case2
                     LEFT-ROTATE(T, x)        ▷Case2
               color[p[x]] ← BLACK            ▷Case3
               color[p[p[x]] ← RED            ▷Case3
               RIGHT-ROTATE(T, p[p[x]])       ▷Case3
      else (same as then clause
            with "right" and "left" exchanged)
color[ root[T]] ← BLACK
```

Rot-schwarz Baum

Diskussion in 3 Schritten:

1. Verletzungen durch 1 und 2
2. globales Ziel der while-Schleife
3. Unterfälle

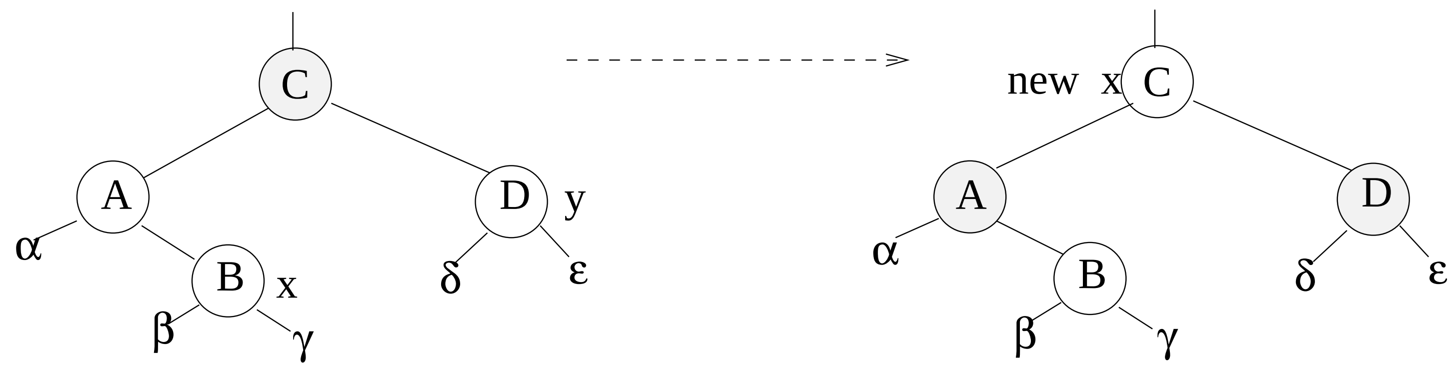
nächste Abbildung: Arbeitsweise von RB-INSERT.


(a)
11
2
14
1
7
15
5
8
y
x
4
Case 1
(b)
11
2
14
y
1
7
x
15
5
8
4
Case 2
(c)
11
7
14
y
x
2
8
15
1
5
4
Case 3
(d)
7
x
2
11
1
5
8
14
4
15

1. Jeder Knoten ist entweder Schwarz oder Rot:
   Kann nicht verletzt werden.
2. Jedes Blatt (NIL) ist schwarz:
   Kann nicht verletzt werden.
3. Falls Knoten rot, so sind beide Söhne schwarz:
   verletzt, falls p[x] rot.
4. Jeder einfache Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten:
   Kann nicht verletzt werden, da ein schwarzer Knoten (NIL) durch einen roten (x) mit schwarzem Sohn (NIL) ersetzt wird.

- Die while-Schleife schiebt die mögliche Verletzung von 3 in Richtung Wurzel.

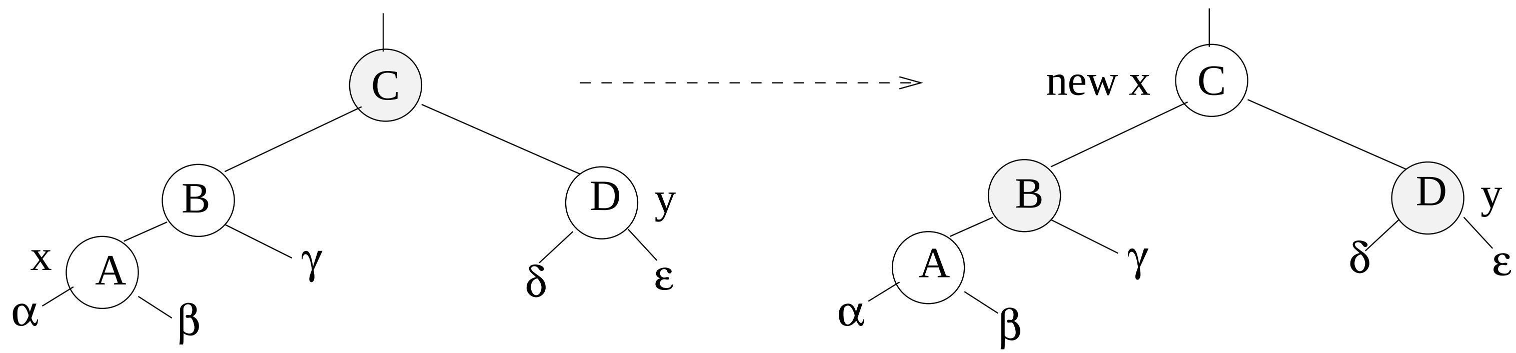
Die Fälle im einzelnen:



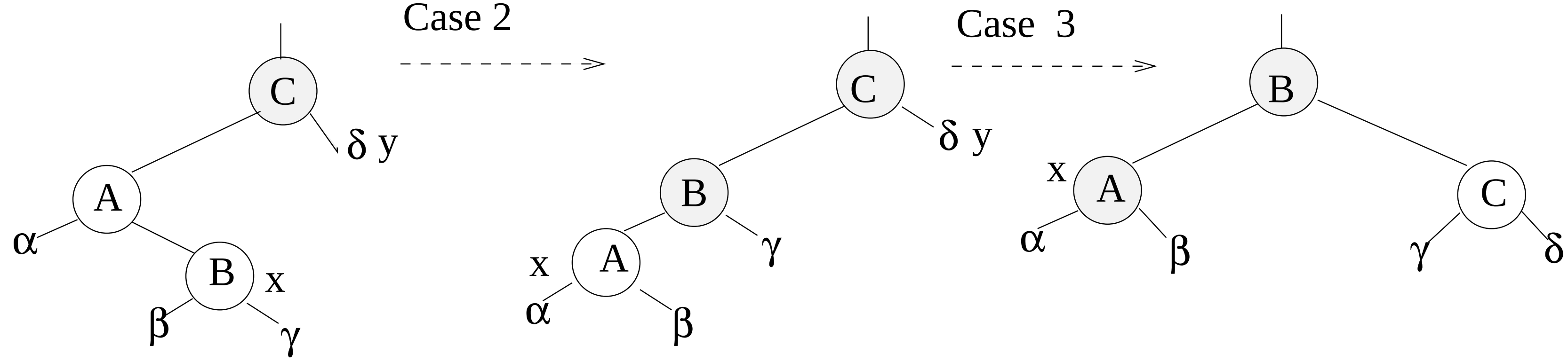
case 1

(a)

p[x] koennte rot sein!



case 1 (b)



c)

RB-INSERT

- Laufzeit: $O(\lg n)$
- never more than 2 rotations/insert

Delete

- Delete wird $O(\lg n)$ beanspruchen
- um Code zu vereinfachen: Dummies für NIL
- RB-DELETE ist fast identisch zum allgemeinen TREE-DELETE, es wird jedoch zusätzlich eine Prozedur RB-DELETE-FIXUP aufgerufen, die sich durch umfärben und rotieren um die red-black-Eigenschaften kümmert.

Dummy

Ein Dummy-Knoten **nil**$[T]$ für einen Rot-Schwarz-Baum hat die gleichen Felder wie die anderen Knoten.

- **color** hat den Wert black
- **p, left, right, und key** sind beliebig
- alle Zeiger auf NIL werden durch Zeiger auf **nil**$[T]$ gesetzt
- Es gibt nur einen Dummy-Knoten **nil**$[T]$
- Falls aber ein Sohn von x ein Dummy ist und wir mit p[nil[t]] operieren müssen, so müssen wir dies erst auf x setzen.

```
RB-DELETE(T, z)
if left[z] = nil[T] or right[z] = nil[T]
   then y ← z
   else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
   then root[T] ← x
   else if y = left[p[y]]
      then left[p[y]] ← x
      else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
      ▷ If y has other fields, copy them, too.
if color[y] = Black
   then RB-DELETE-FIXUP(T, x)
return y
```

Es gibt 3 Unterschiede zu TREE-DELETE:

1. Alle Auftauchen von NIL wurden durch **nil**[T] ersetzt
2. Der Test $\mathbf{x} \neq \mathbf{NIL}$ in Zeile 7 wurde eliminiert
   - falls **x=nil[T]**, dann gilt zeigt **p[nil[T]]** auf **p[y]** (**y** ist der ausgefügte Knoten).
3. Der Aufruf von RB-DELETE-FIXUP falls y schwarz (Z. 16/17)
   - y rot: schwarze Höhe unverändert
   - y schwarz: fixup von x aufwärts.
     - NOTE: p[x]=p[y] durch Z.7

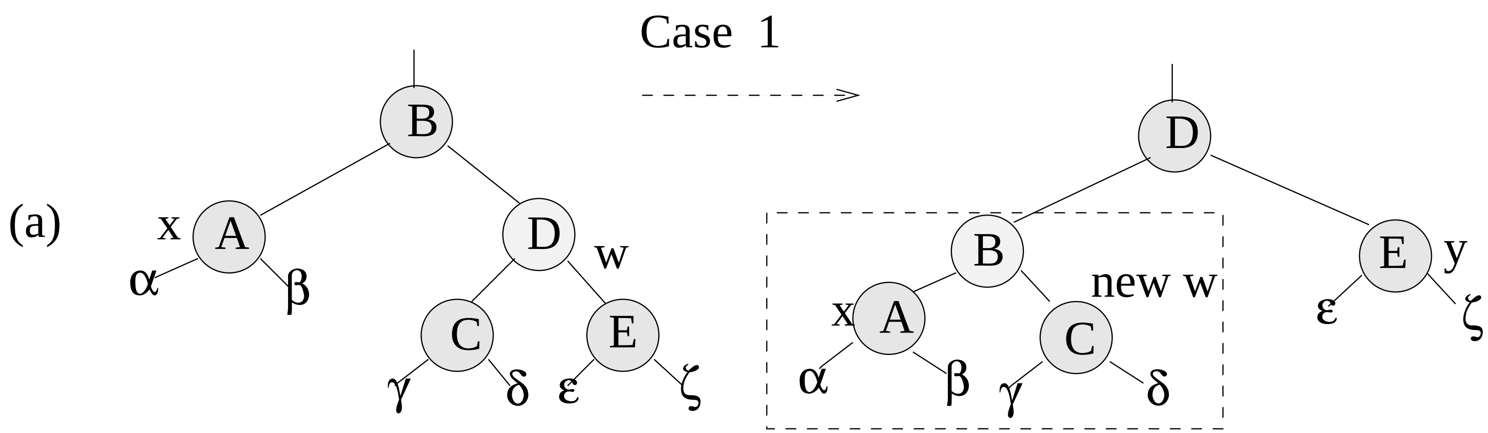
     schwarze Höhe eins weniger, Eigenschaft 4 somit verletzt. Wiederherstellen: Sohn schwarz machen gibt Probleme falls dieser schon schwarz ist (er ist jetzt doppelt schwarz). Daher mit Verletzung zur Wurzel laufen und dabei “irgendwo” beheben.

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
  do if x = left[p[x]]
    then w ← right[p[x]]
      if color[w] = RED
        then color[w] ← BLACK                    ▷ C 1
          color[p[x]] ← = RED                    ▷ C 1
          LEFT-ROTATE(T, p[x])                   ▷ C 1
          w ← right[p[x]]                        ▷ C 1
      if color[ left[w]] = BLACK
        and color[ right[w]] = BLACK
        then color[w] ← RED                      ▷ C 2
          x ← p[x]                               ▷ C 2
        else if color[ right[w]] = BLACK
          then color[ left[w]] ← BLACK           ▷ C 3
            color[w] ← RED                       ▷ C 3
            RIGHT-ROTATE(T, w)                   ▷ C 3
            w ← right[p[x]]                      ▷ C 3
          color[w] ← color[p[x]]                 ▷ C 4
          color[p[x]] ← BLACK                    ▷ C 4
          color[ right[w]] ← BLACK               ▷ C 4
          LEFT-ROTATE(T, p[x])                   ▷ Case 4
          x ← root[T]                            ▷ Case 4
```

else (same as **then** clause
with "right" and "left" exchanged)
color[x] $\leftarrow$ BLACK

RB-DELETE-FIXUP

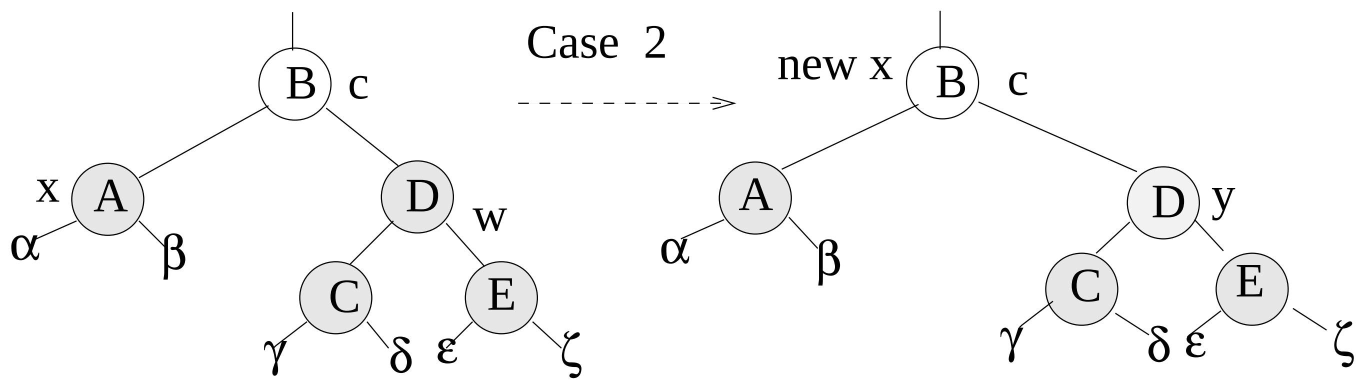
- Ziel der Schleife: das doppelte Schwarz nach oben schieben bis man
  1. auf einen roten Knoten trifft. Dieser wird dann schwarz gefärbt.
     ODER
  2. bei der Wurzel ankommt (dann schwarz einfach entfernbar)
     ODER
  3. Rotationen und Umfärbungen mit Bruder vorgenommen werden können.
- Innerhalb der Schleife:
  color[x]==black und x ist doppelt schwarz
- Sei w Onkel von x. Da x doppelt schwarz folgt:
  w $\neq$ nil[T] (Eigenschaft 4)



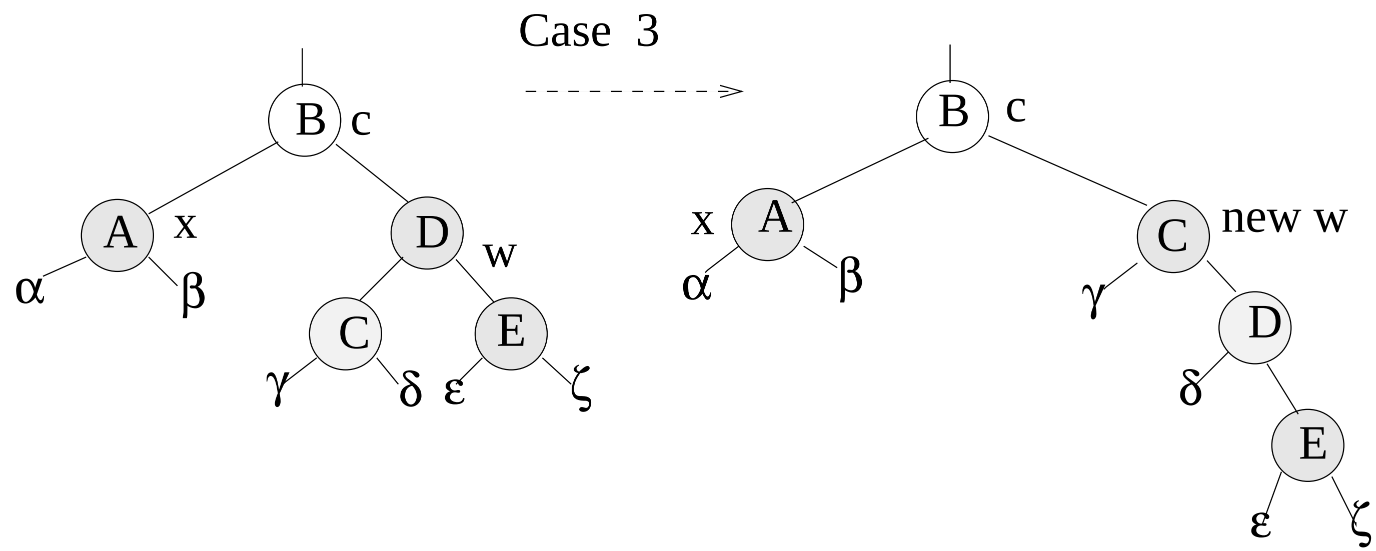
Dies entspricht Case 2, 3, oder 4.

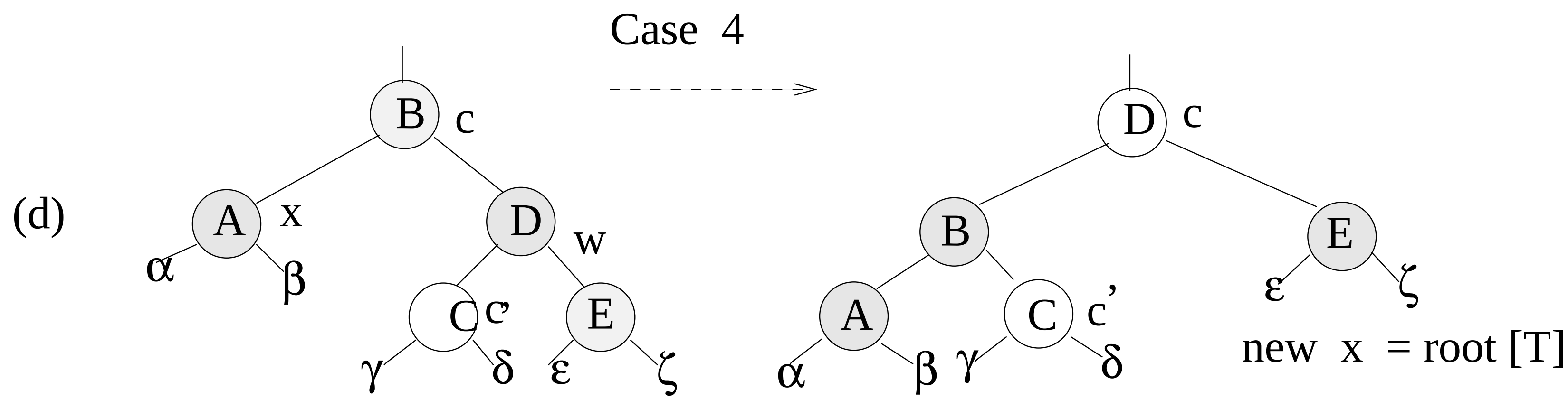
dunkel: schwarz
mittel: rot
hell: rot oder schwarz (kommt auf dieser Folie nicht vor)

(b)



(c)





(d)
Case 4
B c
A x
D w
C c'
E
α β γ δ ε ζ
D c
B
E
A
C c'
new x = root [T]

RB-DELETE-FIXUP

Laufzeit:

- Fälle 1,3,4: Schleife endet, also constant time.
- Fall 2: x $\leftarrow$ p[x], also maximal Höhe.

Laufzeit also: $O(\lg n)$.

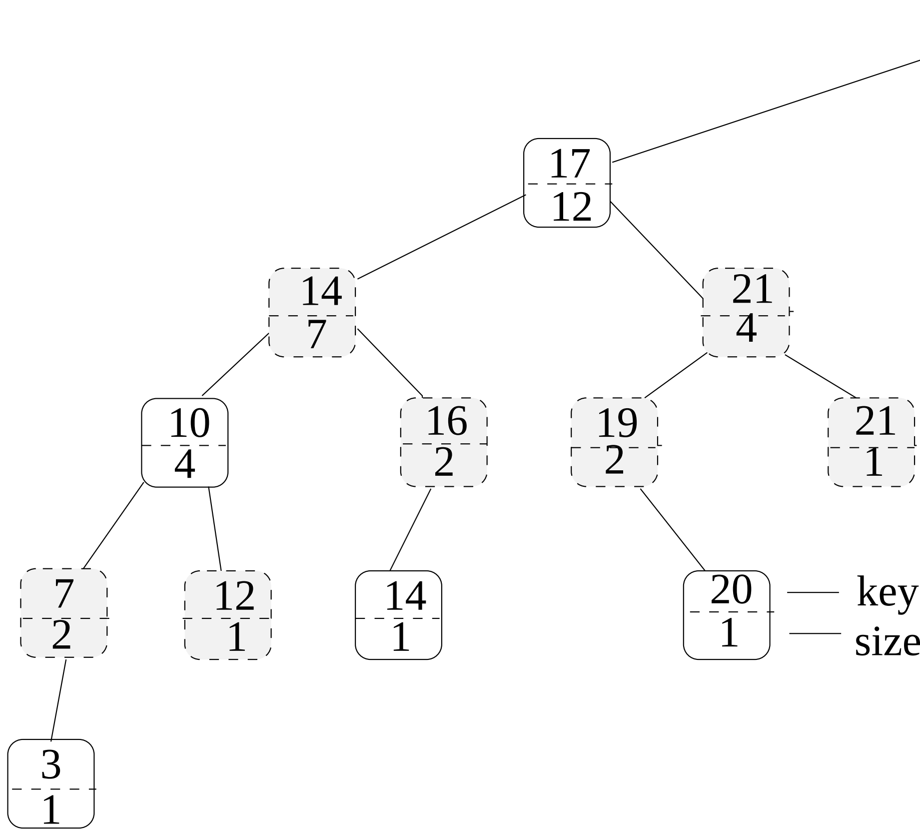
Anreichern von Datenstrukturen

- oft werden mehr Operationen als die Grundoperationen benötigt
- Diese können oft auf existierenden Datenstrukturen implementiert werden
- Beispiele:
  - suche x für gegebene Ordnung (rank)
  - suche Position von x bei inorder Durchlauf
- Dazu nützlich:
  - size[x]: # innere Knoten des Teilbaums von x (incl. x)
  - Es gilt (offensichtlich):

    $$size[x] = size[left[x]] + size[right[x]] + 1$$

  - Dummy bekommt Size 0.

Im Bild:


26
20
17
12
41
7
14
7
21
4
30
5
47
1
10
4
16
2
19
2
21
1
28
1
38
3
7
2
12
1
14
1
20
1
key
size
35
1
39
1
3
1

Bestimme i-t kleinstes Element

```
OS-SELECT(x, i)
r ← size[ left[x]]+1
if i = r
   then return x
elseif i < r
   then return OS-SELECT(left[x],i)
else return OS-SELECT(right[x],i − r)
```

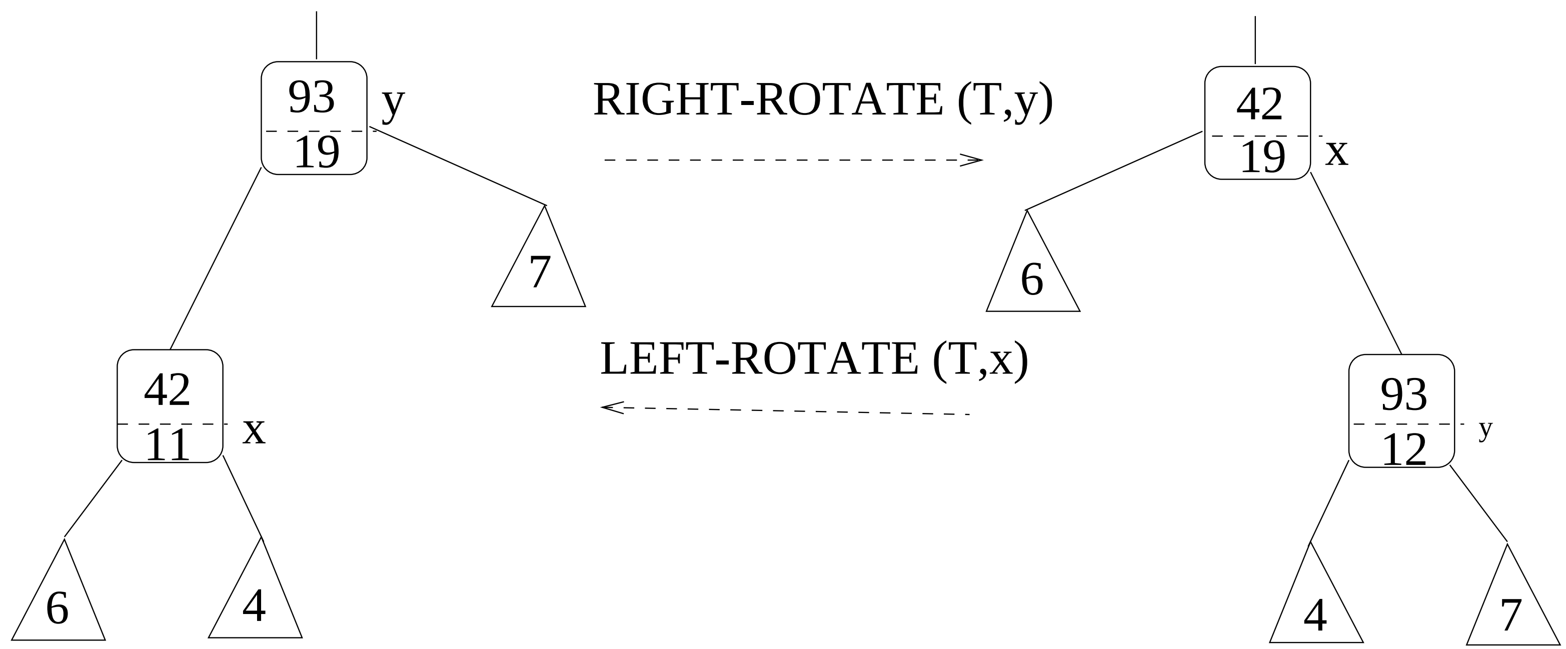
Bestimme Position von x bei inorder Durchlauf

OS-RANK(T, x)

$r \leftarrow$ **size**[**left**$[x]] + 1$
$y \leftarrow x$
while $y \neq$ **root**$[T]$
  do if $y =$ **right**$[p[y]]$
    then $r \leftarrow$ r + **size**[**left**$[p[y]]] + 1$
   $y \leftarrow p[y]$
return r

Wartung der Teilbaumgrößen

- Falls nicht effizient möglich, OS-SELECT und OS-RANK wertlos.
- Einfügen:
  - Beim Suchen der Einfügestelle, jedem besuchten Knoten von der Wurzel zur Einfügestelle eins aufaddieren.
    Aufwand: $O(\lg n)$
  - Rotationen (s. Bild):
    $$\begin{aligned} size[y] &\leftarrow size[x] \\ size[x] &\leftarrow size[left[x]] + size[right[x]] + 1 \end{aligned}$$
    Aufwand: $O(1)$
  - Gesamtaufwand: $O(\lg n)$.
- Delete: Analog


93
19
y
7
42
11
x
6
4
RIGHT-ROTATE (T,y)
LEFT-ROTATE (T,x)
42
19
x
6
93
12
y
4
7

Anreicherung von Datenstrukturen

Schritte:

1. wähle eine zugrundeliegende Datenstruktur
2. bestimme zusätzlich benötigte Information
3. prüfe daß zusätzliche Information für Mutatoren berechnet werden kann
4. entwickle die neuen Operationen

Anreicherung von R-S-Bäumen

Theorem: Sei f ein neues Feld in einem angereicherten Rot-Schwarz-Baum. Für jeden Knoten x sei der Inhalt von f aus den Feldinhalten von x, left[x], und right[x] in $O(1)$ berechenbar. Dann können wir die Information in f verwalten, ohne die asymptotische Laufzeit von $O(\lg n)$ der Mutatoren RB-INSERT und RB-DELETE zu verändern.

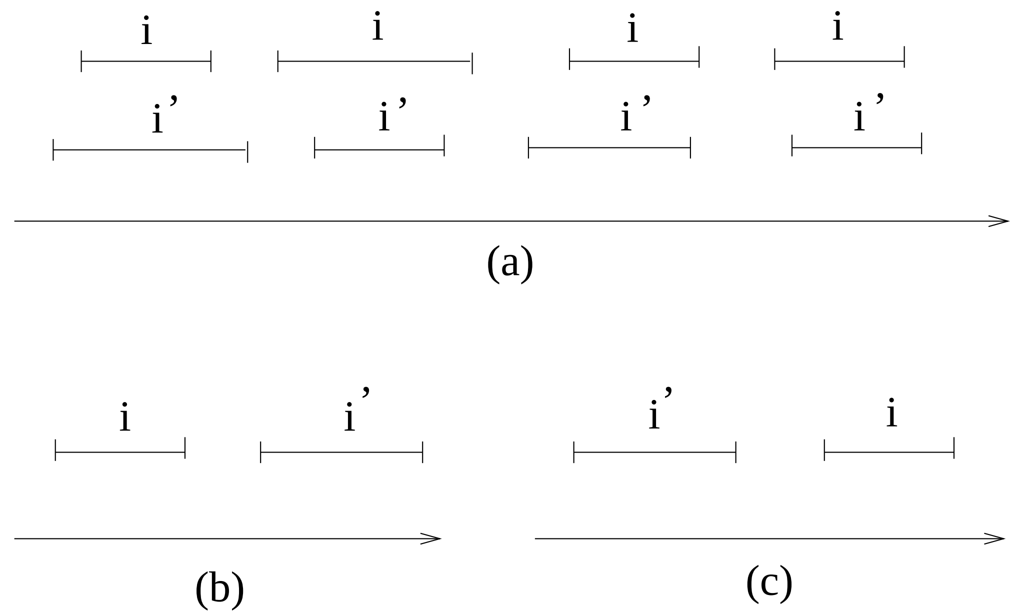
Intervallbäume

Bäume für Intervalle:

- Wir betrachten nur geschlossene Intervalle $i = [t_1, t_2]$ von reellen Zahlen.
- $low[i] = t_1$, $high[i] = t_2$
- Verallgemeinerung auf (halb-) offene Intervalle einfach.

Intervalltrichotomy: Je zwei Intervalle i und i' erfüllen genau eine der folgenden Eigenschaften:

1. i und i' überschneiden sich
2. $\mathbf{high}[i] < \mathbf{low}[i']$
3. $\mathbf{high}[i'] < \mathbf{low}[i]$


i
i
i
i
i'
i'
i'
i'
(a)
i
i'
i'
i
(b)
(c)

Intervallbäume

Ein Intervallbaum enthält eine dynamische Menge von Intervallen **int**$[x]$ und unterstützt die folgenden Operationen:

- INTERVAL-INSERT(T, x) fügt ein Element x mit assoziiertem Intervall **int**$[x]$ zu T hinzu.
- INTERVAL-DELETE(T, x) entfernt x aus T.
- INTERVAL-SEARCH(T, i) gibt einen Zeiger auf ein Element x aus T hinzu, das mit dem Intervall i überlappt. Falls kein solches existiert, so wird NIL zurückgegeben.

Wir reichern Rot-Schwarz-Bäume zu Intervallbäumen an.

Schritt 1: Anreichern der Datenstruktur

- Jeder Knoten x wird um ein Intervall $\mathbf{int}[x]$ angereichert.
- Der Schlüssel (key) von x ist $\mathbf{low}[\,\mathbf{int}[x]]$.
- inorder-Durchlauf ergibt also enthaltene Intervalle nach Anfangspunkt sortiert.

Schritt 2: Zusätzliche Information

- Jeder Knoten x enthält in $\mathbf{max}[x]$ das Maximum aller Endpunkte aller Intervalle im Teilbaum von x.

Schritt 3: Wartung der Information

Es gilt:

$$
\begin{aligned}
\max[x] \;=\; & \max(\\
& \quad \textbf{high}[\,\textbf{int}[x]], \\
& \quad \max[\textbf{left}[x]], \\
& \quad \max[\textbf{right}[x]] \\
)
\end{aligned}
$$

Unser Theorem schlägt also zu.

Die Details: Übung.

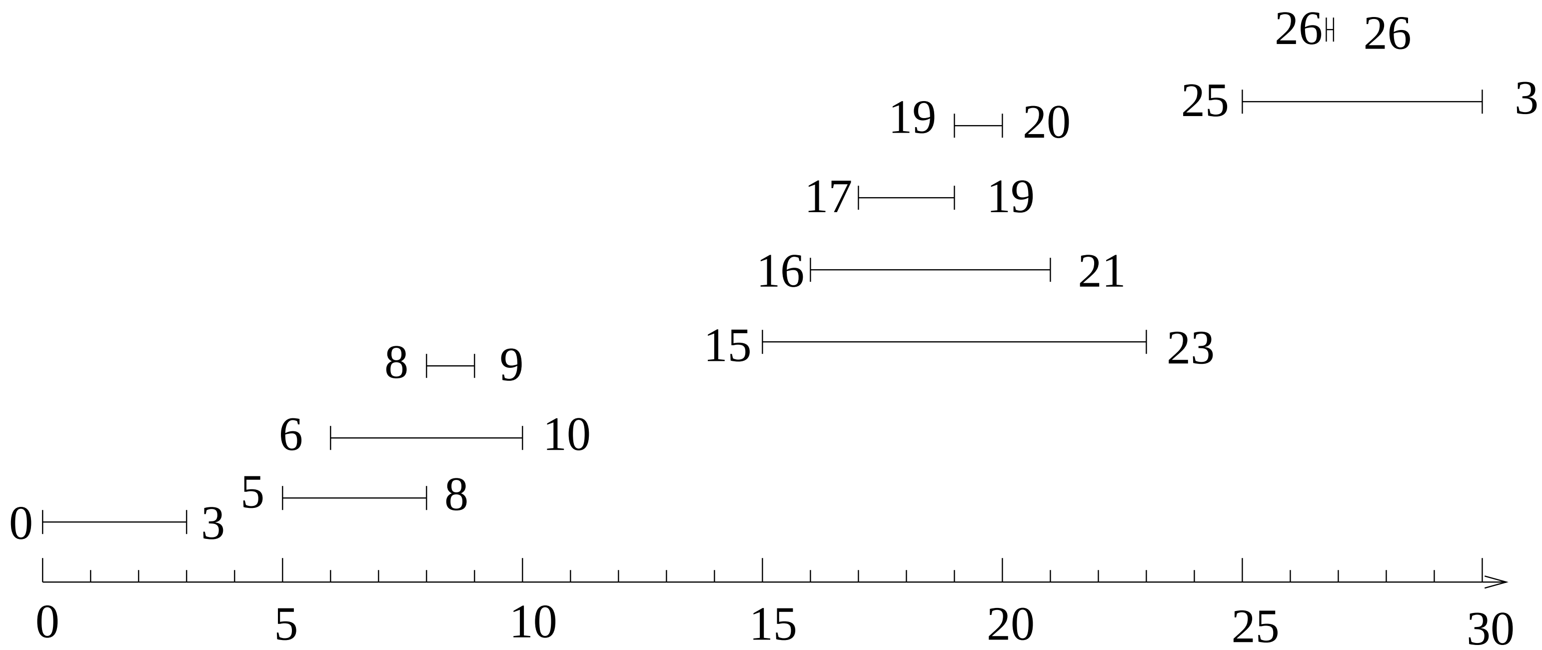
Schritt 4: Neue Operationen

INTERVAL-SEARCH(T, i)

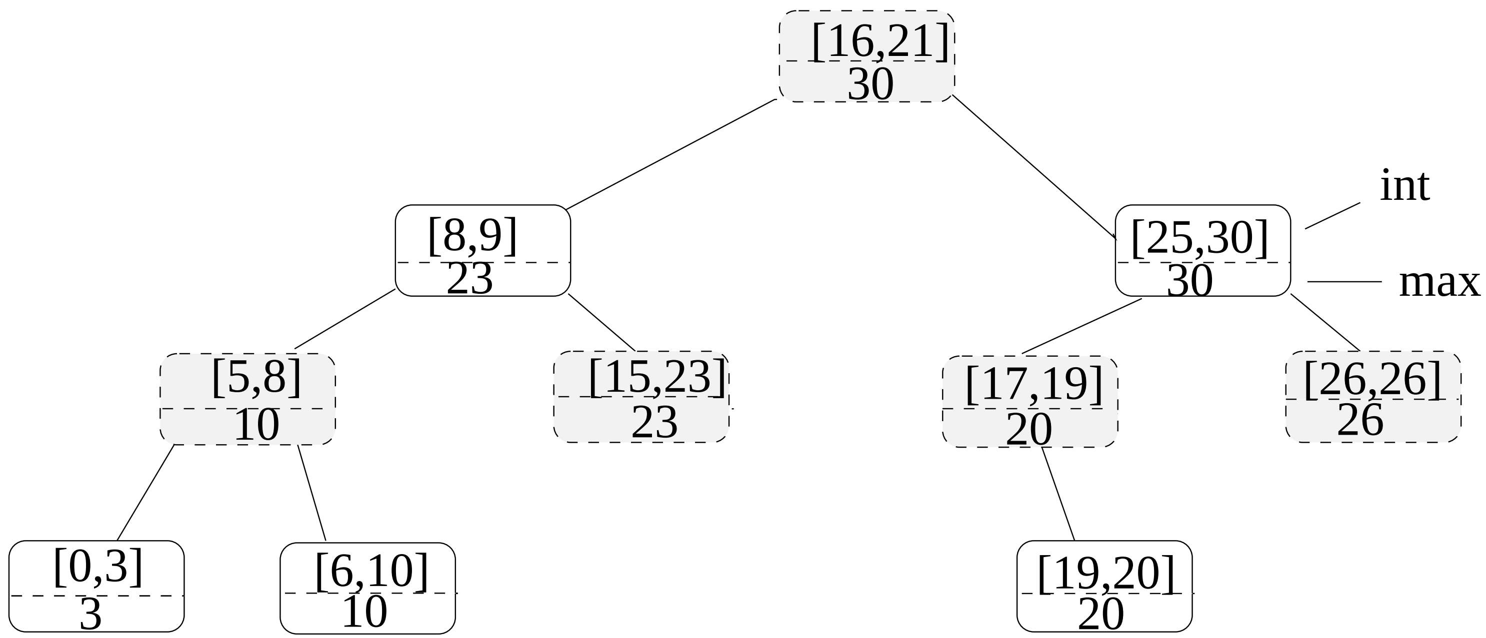
$x \leftarrow$ **root**$[T]$
while $x \neq$ NIL and i does not overlap $int[x]$
   do if left$[x] \neq$ NIL and **max**[**left**$[x]] \geq$ **low**$[i]$
      then $x \leftarrow$ **left**$[x]$
      else $x \leftarrow$ **right**$[x]$
return x

Da jeder Schritt $O(1)$ kostet und in jedem Schritt x auf einen Sohn gesetzt wird, ist die Laufzeit $O(\lg n)$.

Wie es funktioniert: Suche nach $[22, 25]$ in:



(a)


[16,21]
30
[8,9]
23
[25,30]
30
int
max
[5,8]
10
[15,23]
23
[17,19]
20
[26,26]
26
[0,3]
3
[6,10]
10
[19,20]
20

(b)

Korrektheit

Um zu sehen, daß INTERVAL-SEARCH korrekt ist, müssen wir einsehen, daß es genügt, einen einzigen Pfad herunterzulaufen:

- z.z.: falls **int**$[x]$ nicht mit i überlappt, so findet INTERVAL-SEARCH eines, falls eines existiert.

Genauer:

Theorem Man betrachte einen beliebigen Durchlauf der **while**-Schleife von INTERVAL-SEARCH(T, i). Dann gilt:

1. Falls Zeile 4 ausgeführt wird, so enthält **left**$[x]$ ein mit i überlappendes Intervall, oder es existiert keines in **right**$[x]$.
2. Falls Zeile 5 ausgeführt wird, so enthält **left**$[x]$ kein mit i überlappendes Intervall.

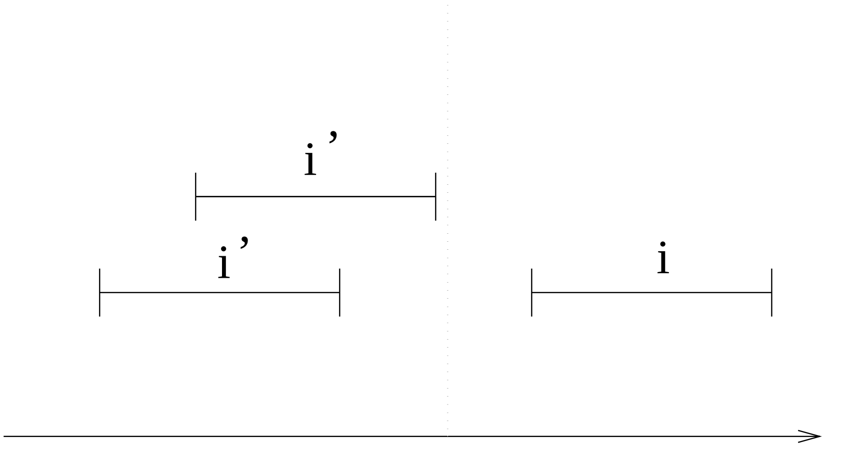
Beweis

Fall 2:

- Z. 5 ausgeführt $\Longrightarrow$
  left[x] == NIL $\vee$ max[left[x]] < low[i]
- Falls left[x] == NIL: $\surd$
- Falls max[left[x]] < low[i]. Sei i' ein Intervall in left[x].
  Dann gilt:

$$\begin{aligned} high[i'] &\leq max[left[x]] \\ &< low[i] \end{aligned}$$

  i und i' können also nicht überlappen.



(a)

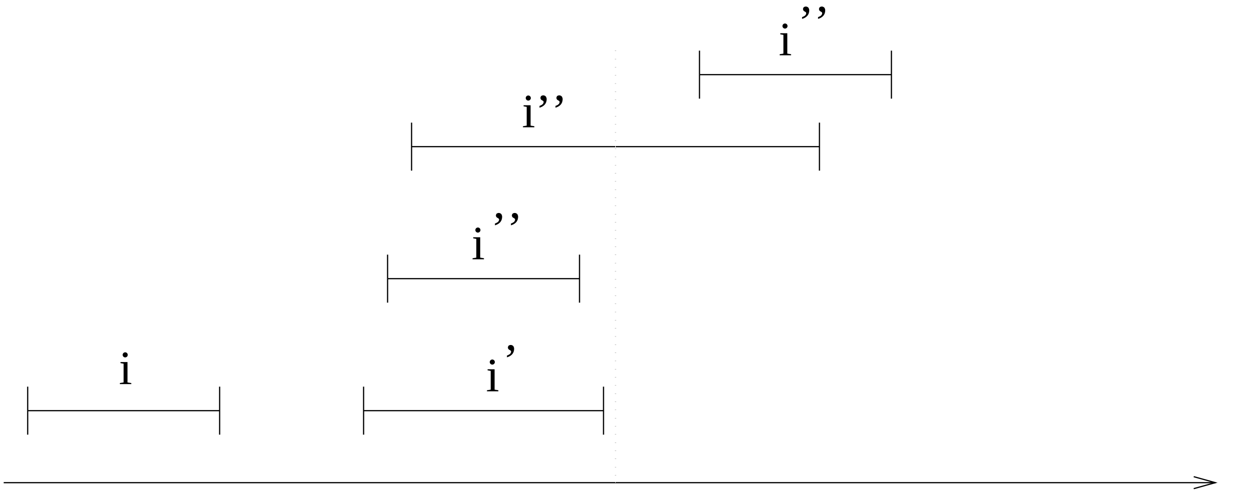
Fall 1:
Annahme: kein Intervall in left[x] überlappt mit i.

- z.z.: kein Intervall in right[x] überlappt mit i.
- Z. 4 ausgeführt $\Longrightarrow$ max[left[x]] $\geq$ low[i]
- nach Def. von **max** gibt es ein i' in left[x] mit
$$high[i'] = max[left[x]] \geq low[i]$$

Da i und i' nicht überlappen, und auch nicht
$$high[i'] < low[i]$$
gilt muß
$$high[i] < low[i']$$
gelten.

Da der Schlüssel im R-S-Baum low[x] ist, impliziert die Suchbaumeigenschaft für jedes i'' in right[x]:
$$\begin{aligned} high[i] &< low[i'] \\ &\leq low[i''] \end{aligned}$$
i und i'' überlappen also nicht.

$\square$



(b)

Datenstrukturen für Hintergrundspeicher

- Buch: Gio Wiederhold
- wir besprechen:
  - extensible hashing
  - B-Baum

B-Baum

- Ein B-Baum-Knoten hat viele Nachfolger (>> 2).
- Jeder Knoten paßt auf eine Seite.
- Jeder Teilbaum repräsentiert einen Schlüsselbereich.
- Wir machen keine Behandlung der Satelliteninformation.
- Normalerweise Satelliteninformation nur in den Blättern gespeichert (B^+-Baum) oder nur Zeiger auf Satelliteninformation in Blättern gespeichert (B^*-Baum).

  Dies maximiert Verzweigungsgrad.

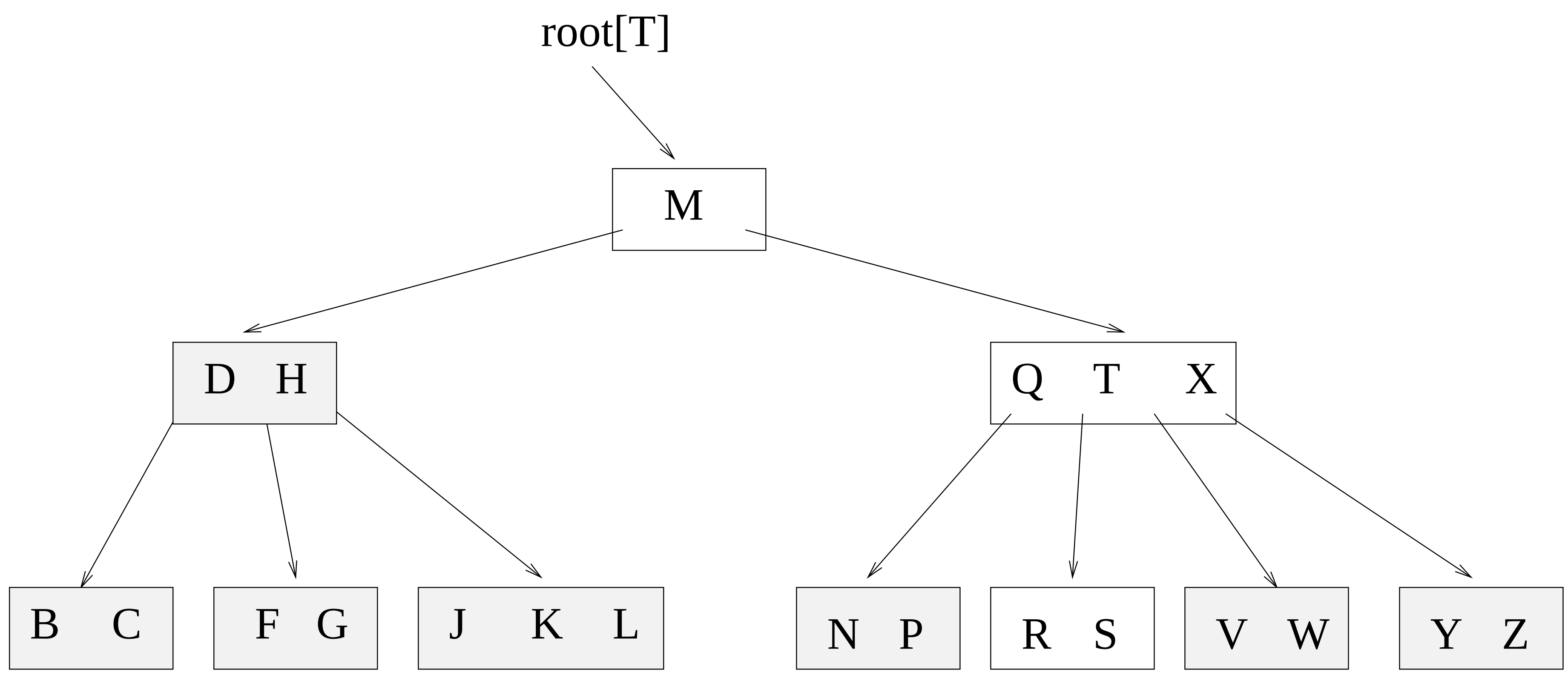
B-Baum

Ein B-Baum ist ein Wurzelbaum mit den folgenden Eigenschaften:

1. Jeder Knoten x hat die folgenden Felder:
   (a) $n[x]$ Anzahl der in x gespeicherten Schlüssel
   (b) die $n[x]$ Schlüssel in aufsteigender Reihenfolge:
   $$key_1[x] \leq \ldots \leq key_{n[x]}[x]$$
   (c) $leaf[x]$ ist TRUE, falls x leaf, FALSE sonst
2. Falls x ein innerer Knoten ist, so enthält x $n[x]+1$ Zeiger:
   $$c_1[x] \leq \ldots \leq c_{n[x]}[x]$$
3. Falls Schlüssel k_i in Teilbaum c_i gespeichert ist, so gilt:
   $$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]}$$
4. Jedes Blatt hat dieselbe Höhe.
5. Es gibt obere und untere Schranken für $n[x]$. Diese werden in $t \geq 2$ ausgedrückt:

(a) $\forall\ \ x \neq root$: $n[x] \geq t-1$ und Anz. Söhne $\geq t$.

(b) $n[x] \leq 2t-1$, also Anz. Söhne $\leq 2t$.
x ist voll, falls er genau $2t-1$ Schlüssel enthält.


root[T]
M
D H
Q T X
B C
F G
J K L
N P
R S
V W
Y Z

B-Baum

Theorem Sei $n \geq 1$. Dann gilt für jeden B-Baum der Höhe h mit n Schlüsseln und minimalem Verzweigungsgrad $t \geq 2$:

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

Beweis Schlechtester Fall:

- Wurzel enthält einen Schüssel.
- Andere Knoten enthalten $t-1$ Schlüssel

Dann gibt es

- 2 Knoten der Höhe 1
- $2t$ Knoten der Höhe 2
- $2t^2$ Knoten der Höhe 3, etc.

Also:

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

Hieraus folgt die Behauptung.

B-Baum

- Vorteil: hoher Verzweigungsgrad garantiert weniger Seitenzugriffe.
- Typische Höhe: 3.

Operationen:

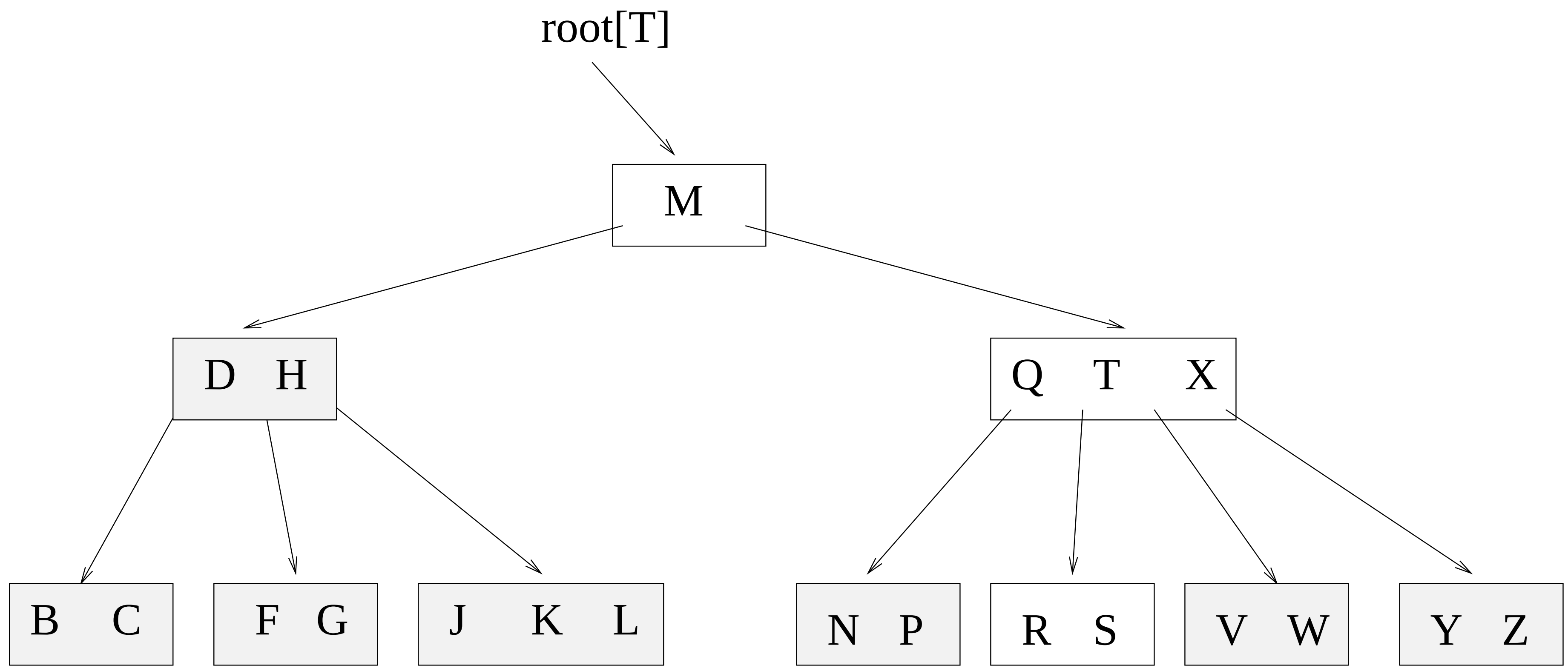
- B-TREE-SEARCH
- B-TREE-CREATE
- B-TREE-INSERT
- B-TREE-DELETE

Annahmen:

- Wurzel immer im Hauptspeicher
- Parameterknoten wurden immer schon in Hauptspeicher gelesen.

```
B-TREE-SEARCH(x, k)
1 i ← 1
2 while i ≤ n[x] and k > key_i[x]
3    do i ← i + 1
4 if i ≤ n[x] and k = key_i[x]
5    then return (x, i)
6 if leaf[x]
7    then return NIL
8    else DISK-READ (c_i[x])
9            return B-TREE-SEARCH (c_i[x], k)
```

- Z.2-3: Besser: binäre Suche.
- Komplexität: $O(\log_t n)$


root[T]
M
D H
Q T X
B C
F G
J K L
N P
R S
V W
Y Z

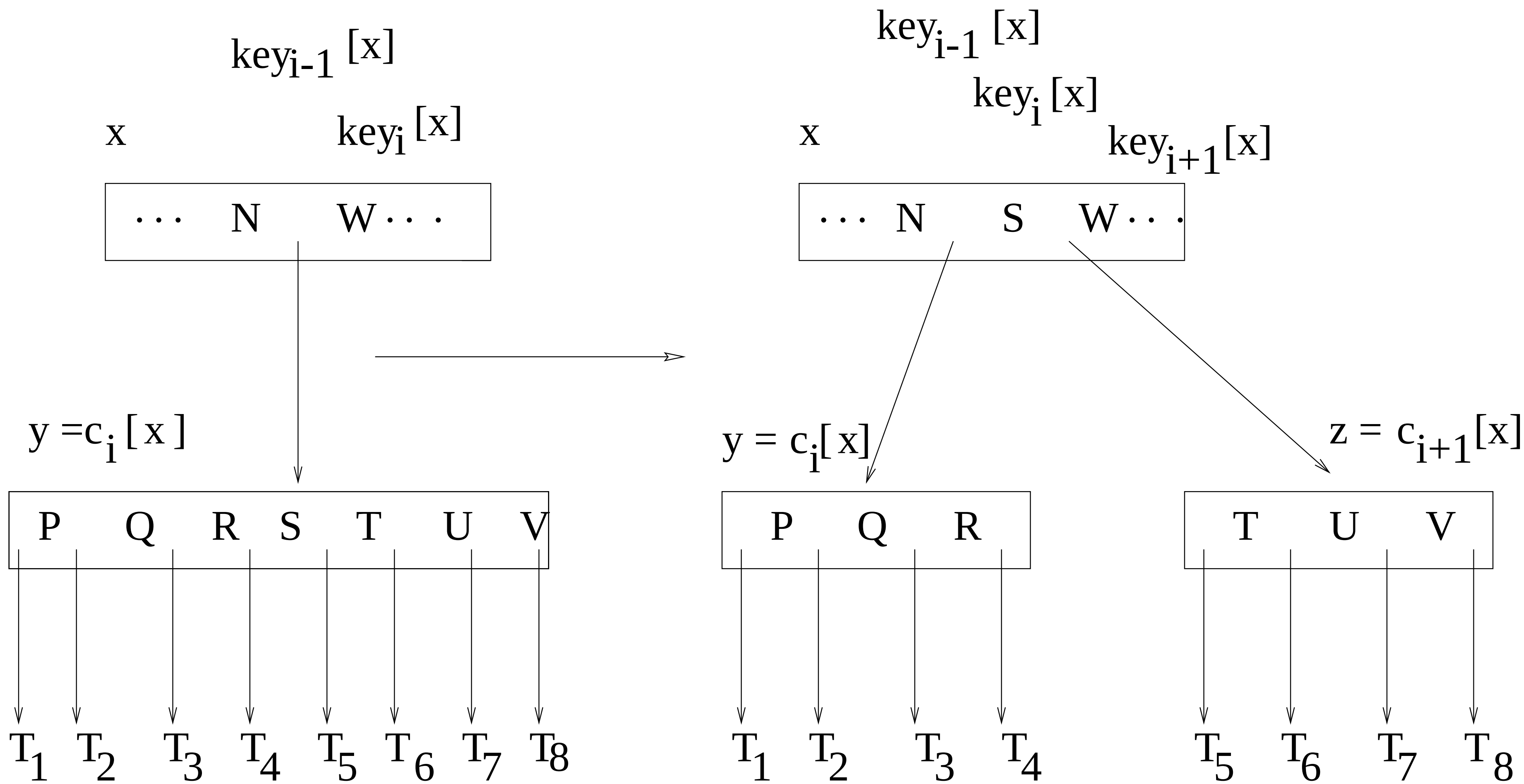
- create und insert benötigen ALLOCATE-NODE: erzeugt Knoten auf einer neuen Seite in $O(1)$, schreibt keine Seite, da noch nichts zu schreiben ist.

B-TREE-CREATE(T)

$x \leftarrow$ ALLOCATE-NODE()
$leaf[x] \leftarrow$ TRUE
$n[x] \leftarrow 0$
DISK-WRITE(x)
$root[T] \leftarrow x$

Splitten

- Falls eine Seite überläuft, so wird sie gesplittet.
- Dazu: Teilen entlang Median-Schlüssel $key_t[y]$.
- Dieser wird dann dem Vater hinzugefügt.
- Falls kein Vater vorhanden: Tiefe wächst um eins.
- B-TREE-SPLIT-CHILD hat als Argument einen nicht vollen Knoten x, bei dem ein Sohn $y = c_i[x]$ voll ist. Dieser wird dann gesplittet.



key i-1 [x]
x
key i [x]
· · · N W · · ·
y = c i [x]
P Q R S T U V
T1 T2 T3 T4 T5 T6 T7 T8
key i-1 [x]
key i [x]
key i+1 [x]
x
· · · N S W · · ·
y = c i [x]
z = c i+1 [x]
P Q R
T U V
T1 T2 T3 T4
T5 T6 T7 T8

```
B-TREE-SPLIT-CHILD(x, i, y)
    z ← ALLOCATE-NODE()
    leaf[z] ← leaf[y]
    n[z] ← t− 1
    for j ← 1 to t - 1
         do key_j[z] ← key_{j+t}[y]
    if not leaf[y]
       then for j ← 1 to t
                do c_j[z] ← c_{j+t}[y]
    n[y] ← t - 1
    forj ← n[x] + 1 downto i + 1
         do c_{j+1}[x] ← c_j[x]
    c_{i+1}[x] ← z
    for j ← n[x] downto i
         do key_{j+1}[x] ← key_i[x]
    key_i[x] ← key_t[y]
    n[x] ← n[x] + 1
    DISK-WRITE(y)
    DISK-WRITE(z)
    DISK-WRITE(x)
```

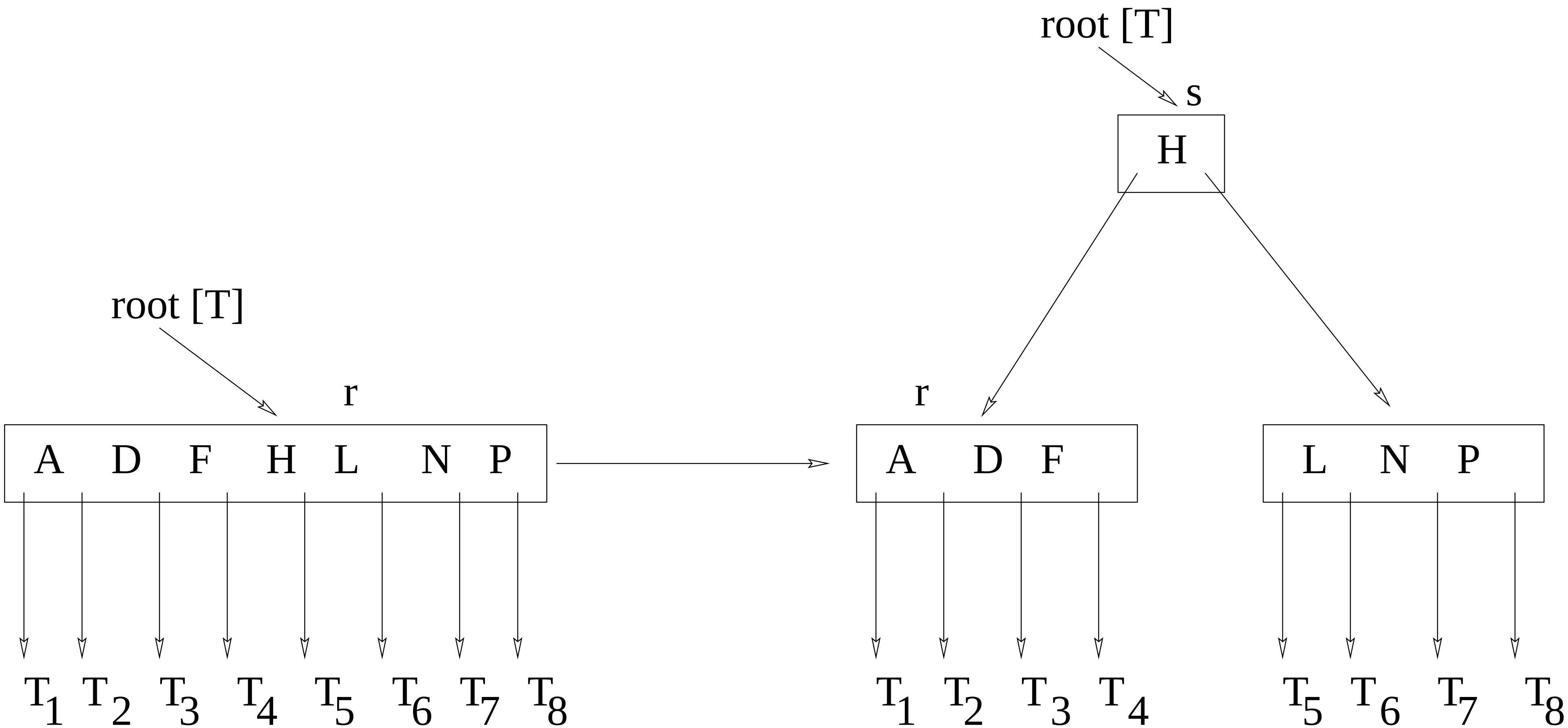
- xs voller Sohn y wird gesplittet.
- Dazu wird die Hälfte der Einträge auf einen neuen Knoten z kopiert und aus y entfernt.
- z wird dann in x der Eintrag direkt nach y.

- Z 1-8: erzeuge z, fülle z, setzen der Zeiger in z, falls z kein Blatt ist
- Z 9: Schlüsselzähler anpassen.
- 10-16: weiterrücken von Schlüsseln und Zeigern in x und einfügen von z
- Z 17-19: Speichern der Änderungen auf Platte

B-TREE-INSERT(T, k)

```
1  r ← root[T]
2  if n[r] = 2t − 1
3      then s ← ALLOCATE-NODE()
4           root[T] ← s
5           leaf[s] ← FALSE
6           n[s] ← 0
7           c_1[s] ← r
8           B-TREE-SPLIT-CHILD(s, 1, r)
9           B-TREE-INSERT-NONFULL(s, k)
10 B-TREE-INSERT-NONFULL(r, k)
```

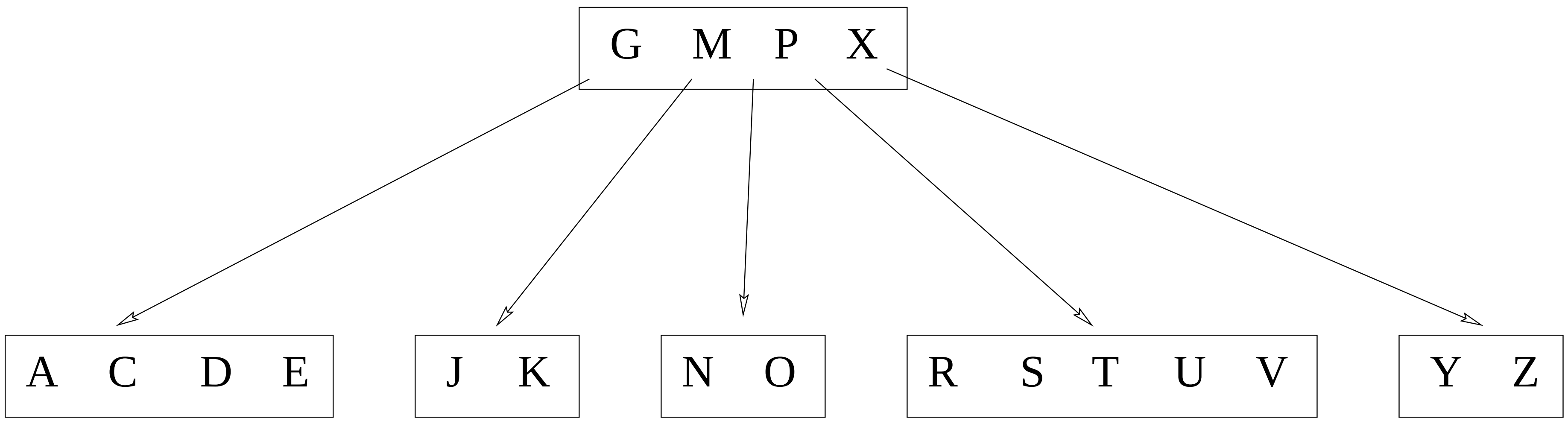
- Z 3-9: Wurzel voll: neue Wurzel erzeugen und splitten
- Z 10: sonst.
- B-TREE-INSERT-NONFULL: einfügen von k in x, wobei x nicht voll ist.


root [T]
r
A D F H L N P
T1 T2 T3 T4 T5 T6 T7 T8
root [T]
s
H
r
A D F
L N P
T1 T2 T3 T4
T5 T6 T7 T8

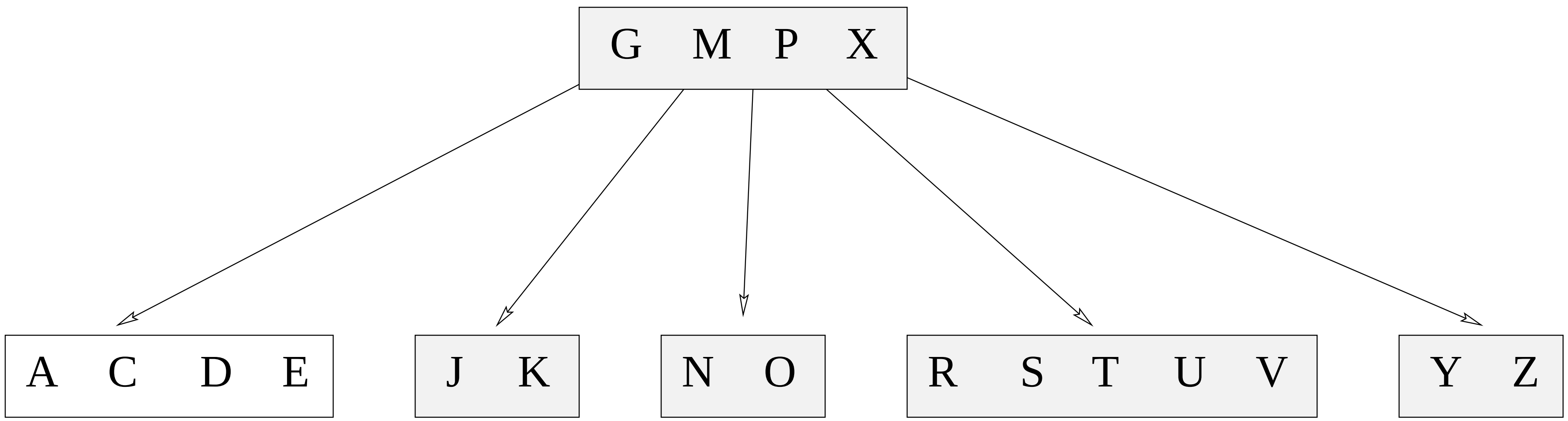
B-TREE-INSERT-NONFULL(x, k)

$i \leftarrow n[x]$
if $leaf[x]$
    then while $i \geq 1$ and $k < key_i[x]$
            do $key_{i+1}[x] \leftarrow key_i[x]$
                $i \leftarrow i$ - 1
        $key_{i+1}[x] \leftarrow k$
        $n[x] \leftarrow n[x] + 1$
        DISK-WRITE(x)
    else while $i \geq 1$ and $k < key_i[x]$
            do $i \leftarrow i$ - 1
        $i \leftarrow i + 1$
        DISK-READ$(c_i[x])$
        if $n[c_i[x]] = 2t$ - 1
            then B-TREE-SPLIT-CHILD$(x, i, c_i[x])$
                if $k > key_i[x]$
                    then $i \leftarrow i + 1$
        B-TREE-INSERT-NONFULL$(c_i[x], k)$

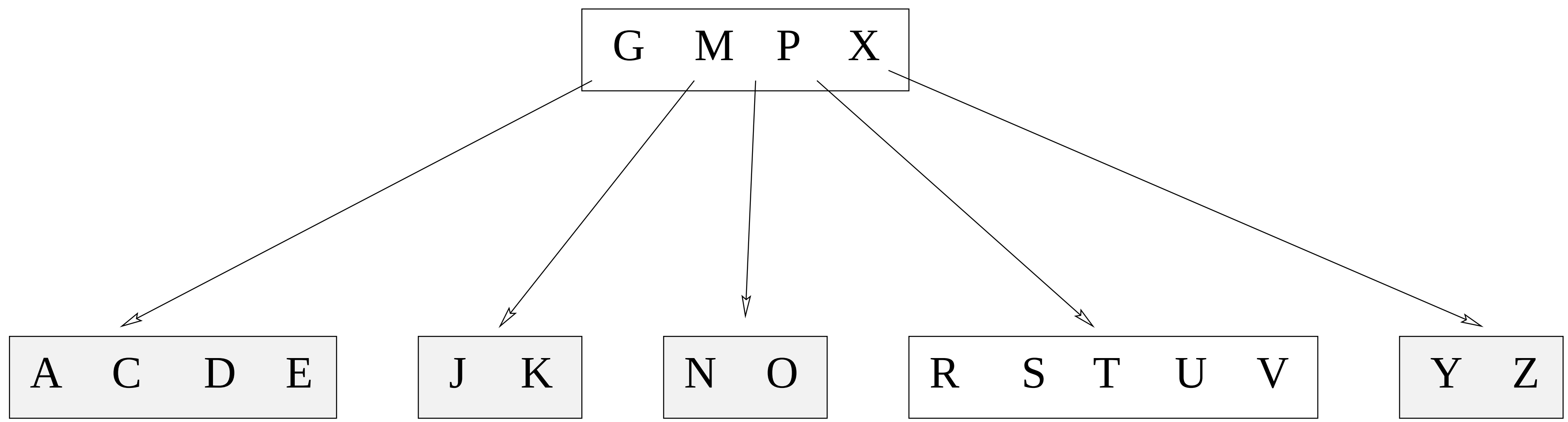
(a) initial tree



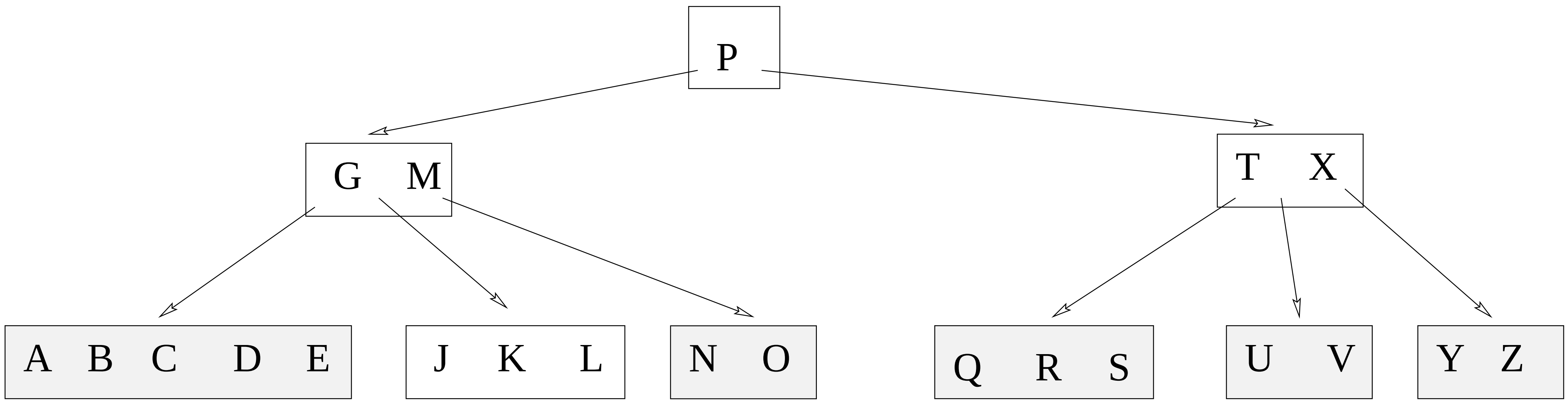
(b) B inserted



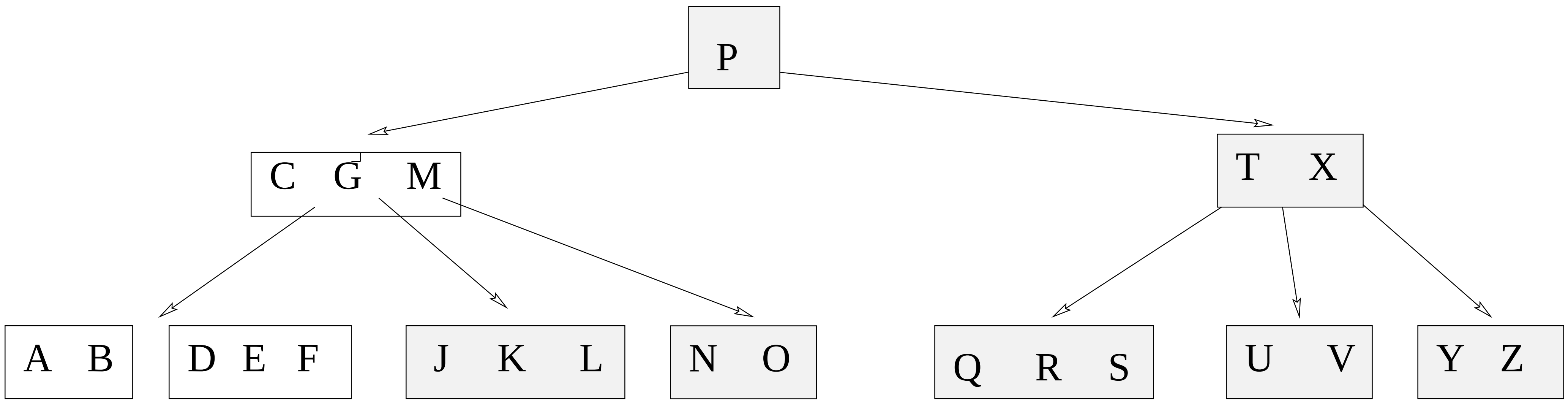
(c) Q inserted



(d) L inserted



(e) F inserted



- Z 3-8: x Blatt:
  - Z 3-5: weiterrücken der Schlüssel,
  - Z 6: einfügen k.
- Z 9-17: stelle Sohn fest, in den einzufügen ist
  - Z 9-11: Sohn suchen
  - Z 12: Sohn lesen
  - Z 13-16: falls Sohn voll, splitten.
  - Z 17: rek. Aufruf

Delete

- delete k aus x
- falls $n[x] > t$ und x Blatt: einfach entfernen
- Ansonsten:
  - mit Nachbarn balancieren
  - mit Nachbarn verschmelzen
- es sind einige Fälle zu betrachten.

Delete

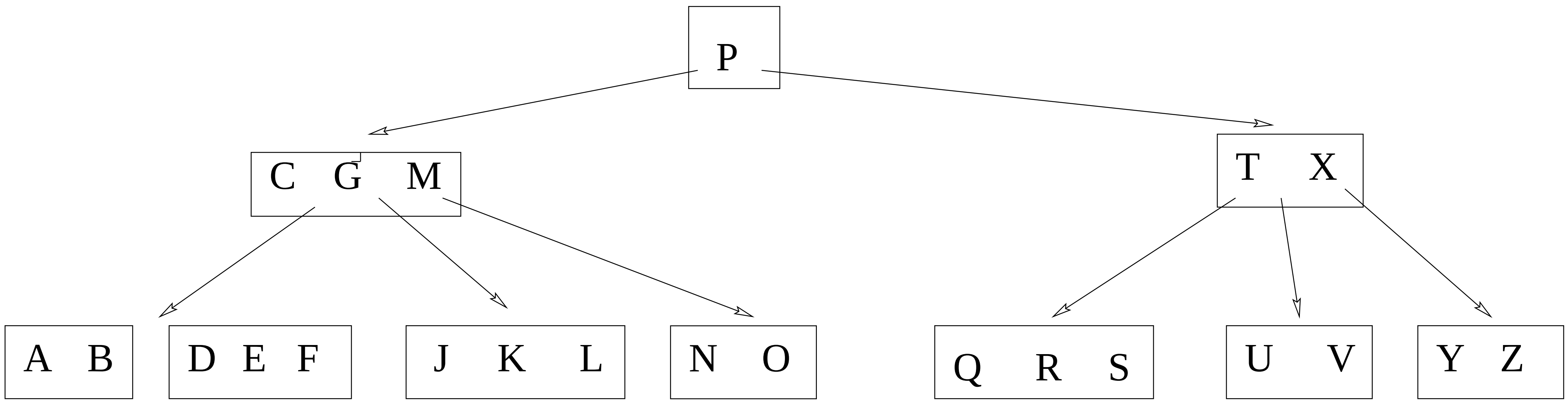
- Annahme: k aus x entfernen.
- B-TREE-DELETE garantiert, daß wenn immer sie rekursiv gerufen wird, in x mindestens t Schlüssel verbleiben.
- Damit muß vor dem Aufruf mindestens ein Schlüssel mehr da sein, als die B-Baum-Bedingung erfordert.
- Daher muß manchmal ein extra Schlüssel von woanders her eingefügt werden.

1. x Blatt, $k \in x$: lösche k aus x.
2. x kein Blatt, $k \in x$:
   (a) falls Vorgängersohn y von k mindestens t Schlüssel hat, ersetze k durch Vorgängerschlüssel k'
   (b) analog für Nachfolgersohn z
   (c) merge y und z
3. k nicht in einem internen Knoten x: bestimme $c_i[x]$ in dem k vorkommen kann. Falls $c_i[x]$ weniger als t Schlüssel hat:
   (a) falls Nachbar y genug Knoten hat: ausgleichen: Schlüssel aus y nach x, einer aus x nach $c_i[x]$
   (b) merge $c_i[x]$ mit einem Nachbarn.

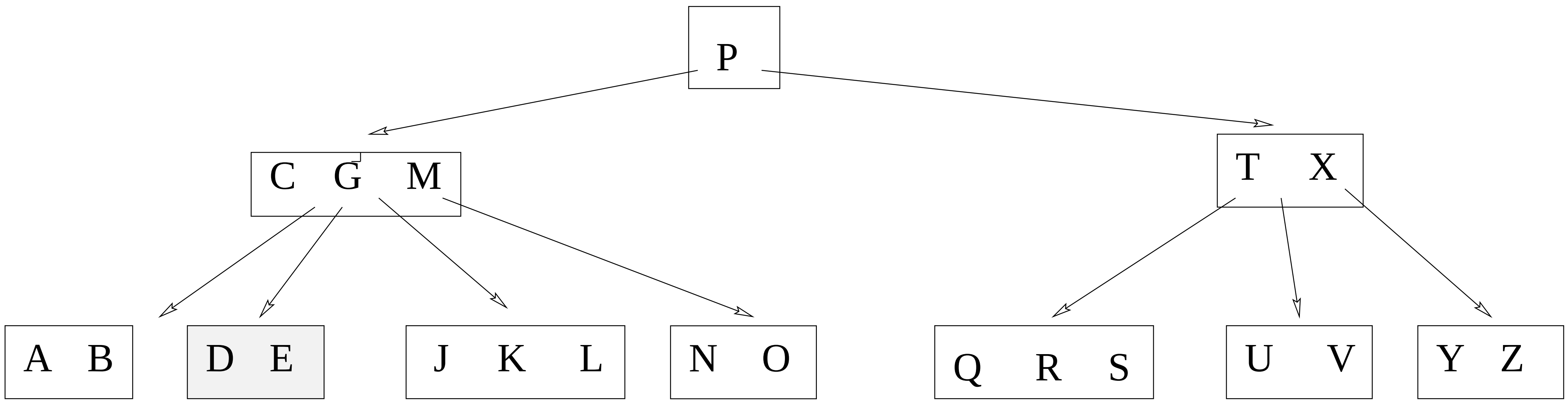
   rek. Aufruf auf $c_i[x]$

Aufwand: $O(h)$, also $O(\log_t n)$

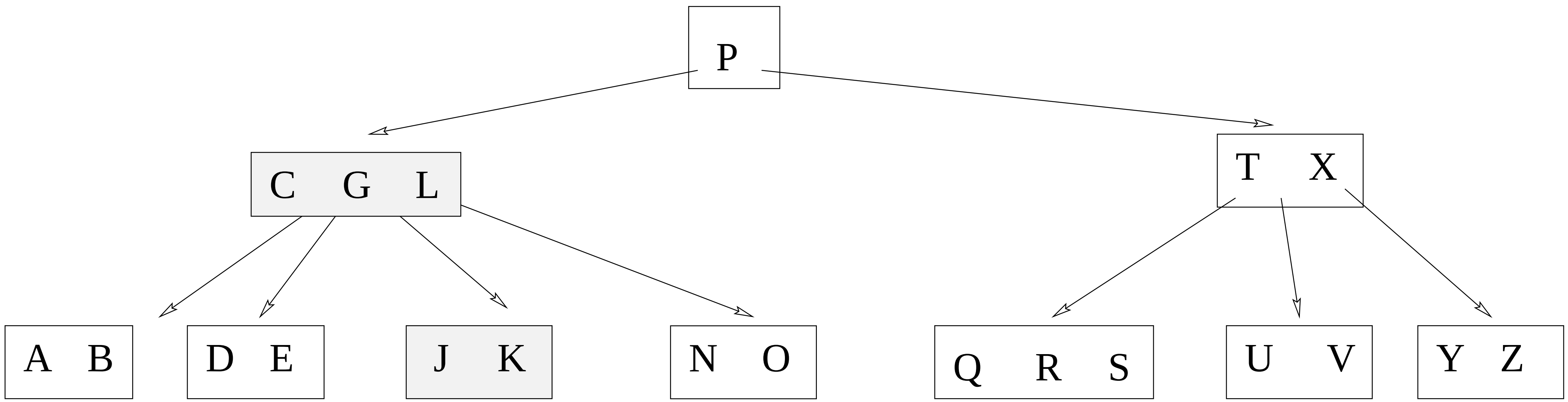
(a) initial tree



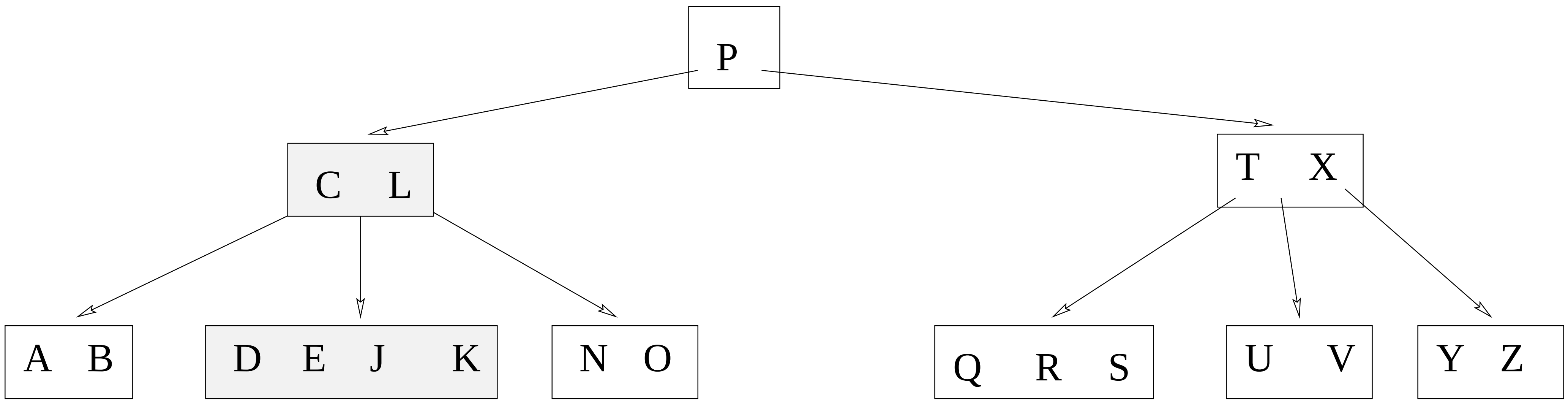
(b) F deleted: case 1



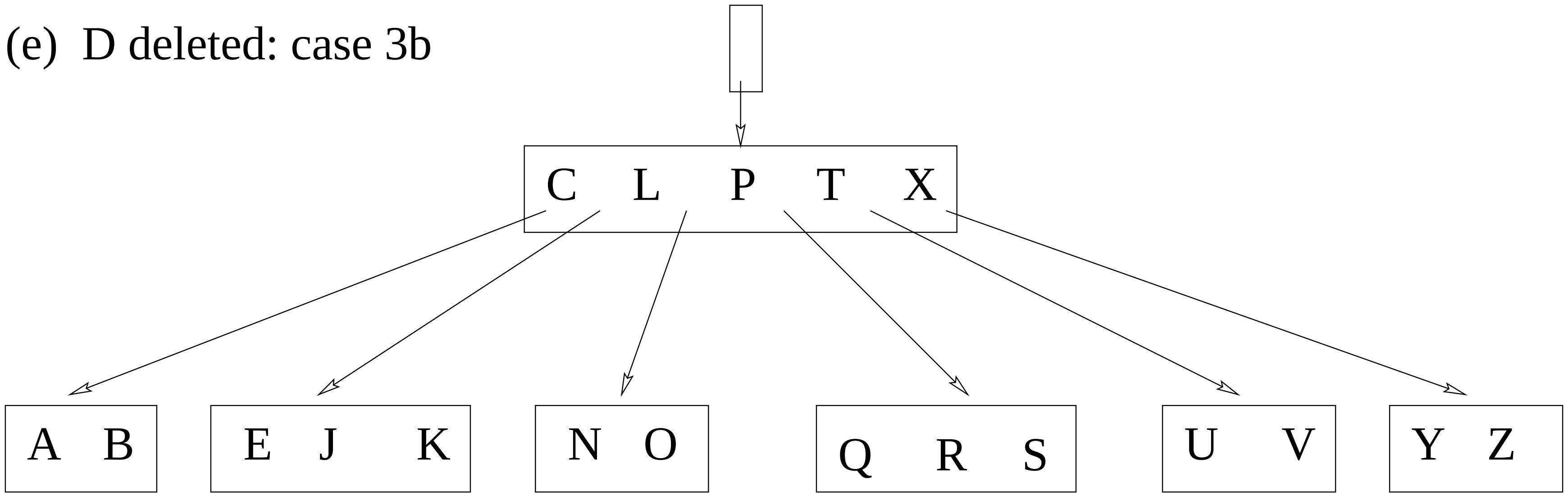
(c) M deleted: case 2a



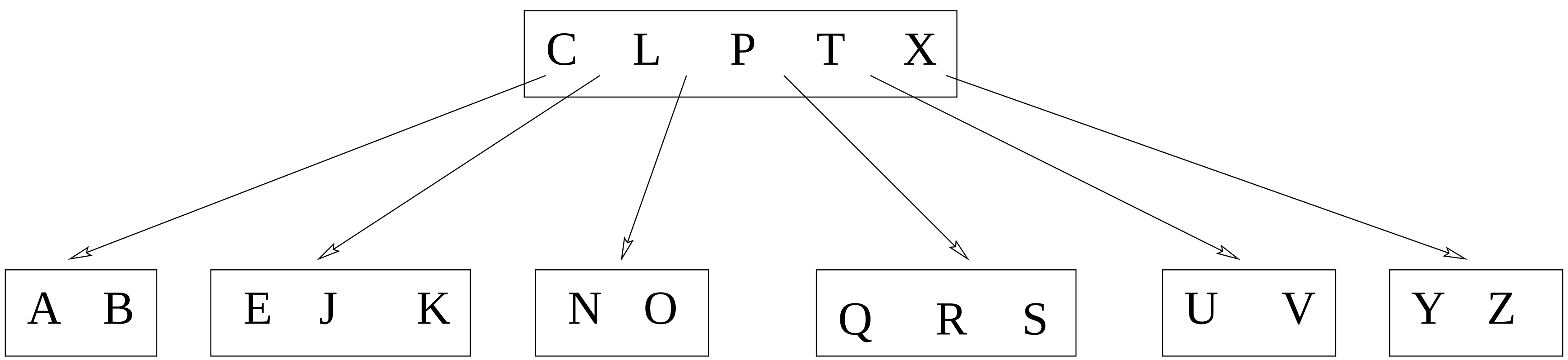
(d) G deleted: case 2c



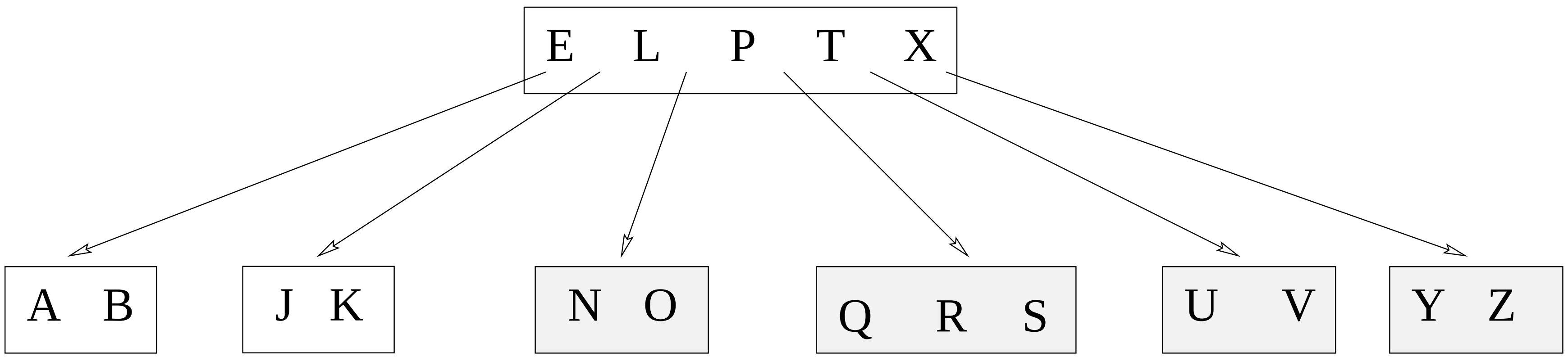
(e) D deleted: case 3b



(e’) tree shrinks in height



(f) B deleted: case 3a



Extensible Hashing

Ziel

Finde

1. das Beste, Billigste, etc.,
2. mit wenig Aufwand.

Methoden:

1. dynamisches Programmieren
2. Greedy-Algorithmen
3. + ...

Aber: diese Methoden nicht nur auf Optimierungsprobleme anwendbar.

Dynamisches Programmieren

Anwendbar, falls bei

- Problemlösung Teilprobleme häufig gleich sind
- und diese unabhängig voneinander lösbar sind.

Daher prädestiniert für so manches Optimierungsproblem.

Beispielklasse:

- gegeben ein Ausdruck
  (mit Operanden und Operationen)
- gesucht billigste Auswertung

Dynamisches Programmieren

Schritte:

1. Charakterisiere Struktur des Problems
2. Def. rekursiv die Kosten des Optimums
3. Berechne Optimum bottom-up
4. Konstruiere Optimum aus bereits berechneter Information

Bsp: Multiplikation mehrerer Matrizen

Gegeben

- Sequenz $\langle A_1, \ldots, A_n \rangle$ von Matrizen

Gesucht

- Produkt der Matrizen $A_1, \ldots, A_n$

Bem.

- Wir werden Produkt ausrechnen, indem wir fortlaufend zwei Matrizen (Original oder Zwischenergebnis) multiplizieren.
- Es gilt das Assoziativgesetz. Daher gibt es viele verschiedene Auswertungsreihenfolgen. Diese sind unterschiedlich teuer, wie wir sehen werden.