Algorithmen und Datenstrukturen

Bäume

Suchbäume, AVL-Bäume, Bruder-Bäume, B-Bäume

Wolfram Burgard und Bernhard Nebel

Bäume

Idee: Bäume sind verallgemeinerte Listenstrukturen.

Nachfolger: Endlich viele Nachfolger eines direkten Vorgängers,
i.Allg. (an)geordnet (erster, zweiter, dritter Nachfolger).

Ordnung: max. Anzahl von Nachfolgern.

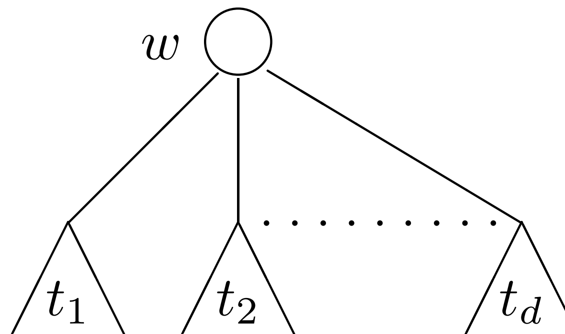
Darstellung: (ungerichteter) Verbund von Knoten mit ausgezeichneter
Wurzel.

Rekursive Definition

1. Der leere Baum $\square$ ist ein Baum der Ordnung d.

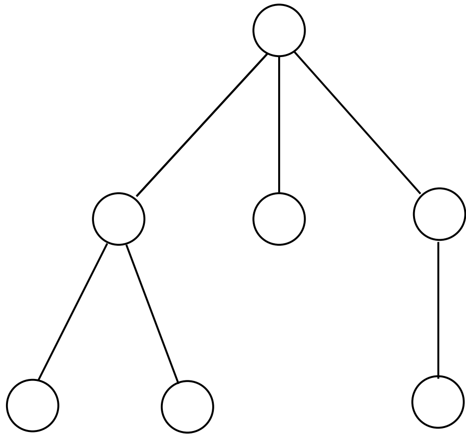
2. Der aus einem einzigen Knoten bestehende Baum ist ein Baum der Ordnung d. Die Höhe h ist 0.

3. Sind $t_1, \ldots, t_d$ beliebige, disjunkte Bäume der Ordnung d, so erhält man einen (weiteren) Baum der Ordnung d, indem man die Wurzeln von $t_1, \ldots, t_d$ zu Nachfolgern einer neu geschaffenen Wurzel w macht. Die Höhe h des neuen Baums ist dann $\max\{h(t_1), \ldots, h(t_d)\} + 1$.

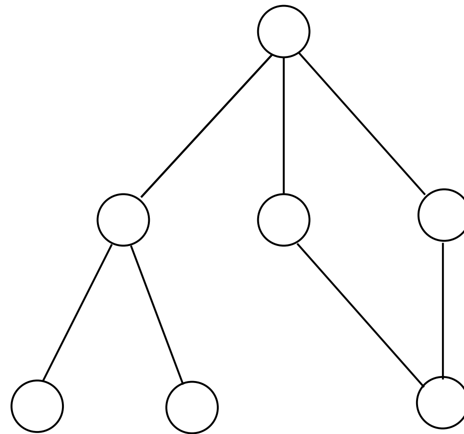


Festlegung: $d = 2$ Binärbäume, $d > 2$ Vielwegbäume.

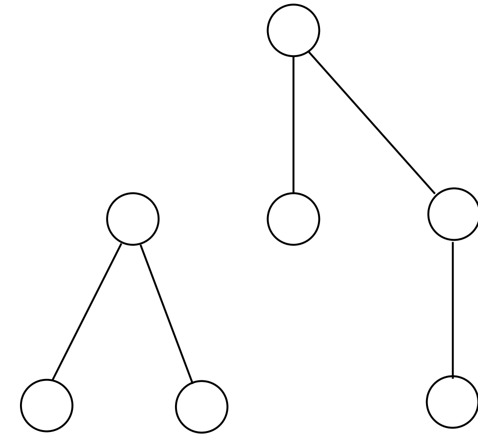
Beispiele



Baum



kein Baum



kein Baum
(aber zwei Bäume)

Knotentypen

Wurzel: Einziger Knoten ohne Vorgänger

Blatt: Knoten ohne Nachfolger

Innerer Knoten: Jeder Knoten, der nicht Blatt ist

(Binäre) Suchbäume

Annahme: Total geordnete Menge von Schlüsseln.

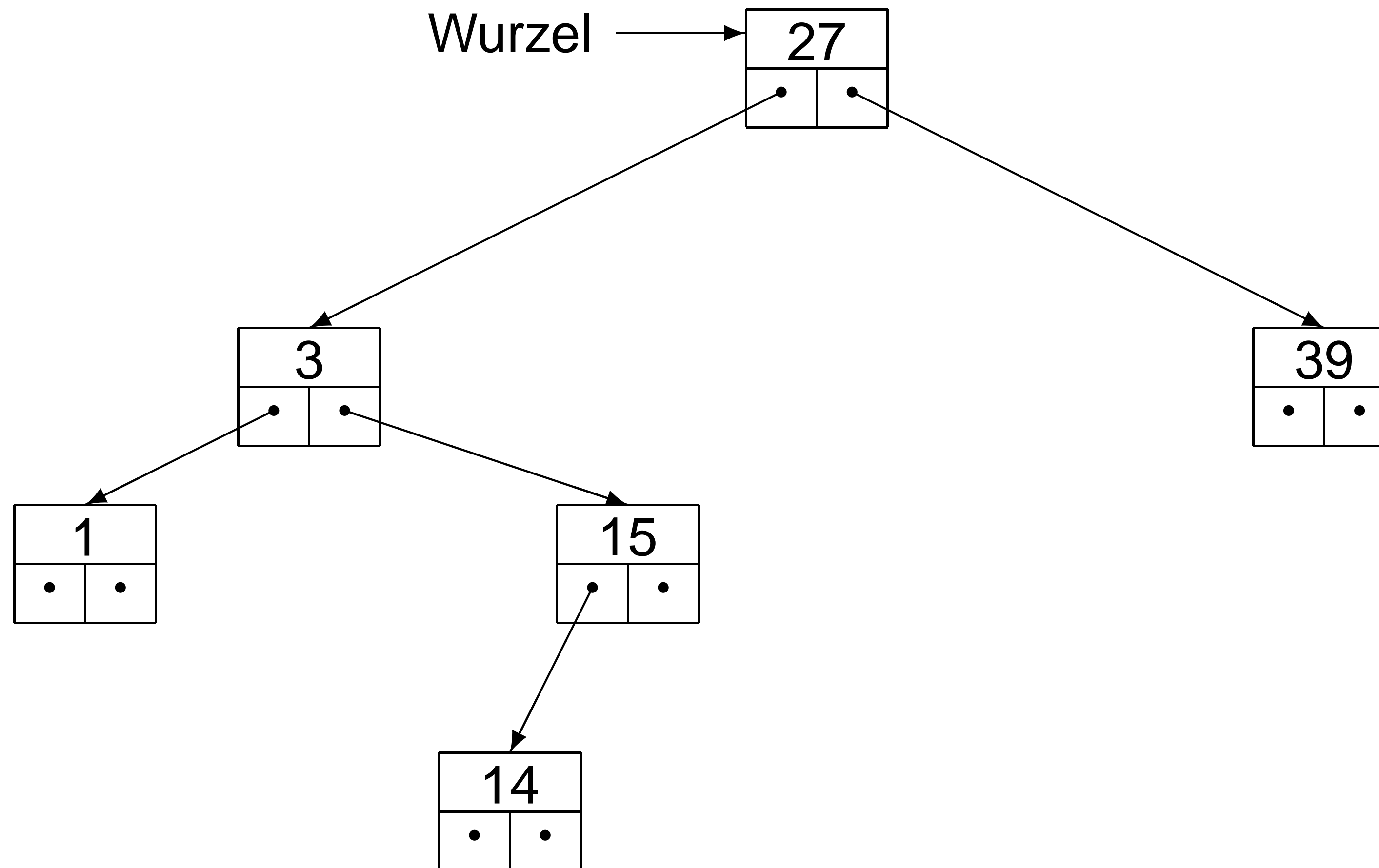
Charakterisierung: Die **Schlüssel im linken Teilbaum eines Knotens** p **sind alle kleiner als der Schlüssel von** p, **und dieser** ist wiederum **kleiner als sämtliche Schlüssel im rechten Teilbaum von** p.

Implementierung: (Schlüssel ist Ganzzahl und ohne Stopper)

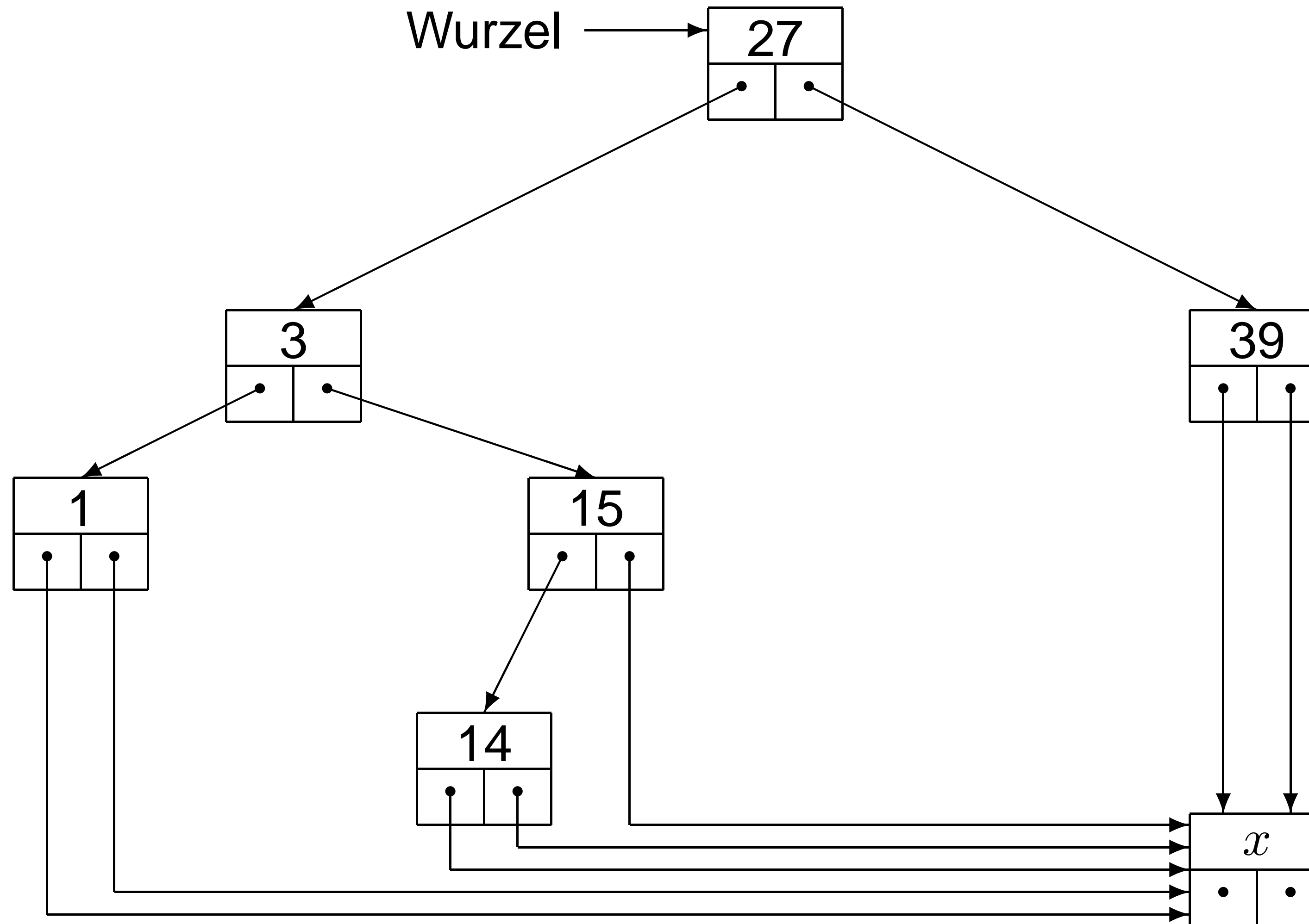
```
public class node{
    node(int key){
        this.key = key;
        this.left = this.right = null;
    }

    int key;
    node left, right;
}
```

Beispiel (ohne Stopper)



Beispiel (mit Stopper)



Die Klasse `searchTree`

Gewünschte Operationen: Suchen, Einfügen und Entfernen

```
public class searchTree {
    searchTree() {
        root = null;
    }

    public node find(int k) {...}
    public void insert(int k) {...}
    public void remove(int k) {...}
    public void print(){...}

    private node root;
}
```

Suchen im Suchbaum

ohne Stopper:

```
public node find(int k) {
  return find(root, k);
}

node find(node p, int x) {
  if (p == null)
     //Abfrage bei jedem Aufruf
     return null;
  else if (x < p.key)
     return find(p.left, x);
  else if (x > p.key)
     return find(p.right, x);
  else
     return p;
  }
```

mit Stopper:

```
public node find(int k) {
  this.stopNode.key = k;
  node tmp = find(root, k);
  if (tmp != this.stopNode)
  //Abfrage am Ende
     return tmp;
  else
     return null;
}

node find(node p, int x) {
  if (x < p.key)
     return find(p.left, x);
  else if (x > p.key)
     return find(p.right, x);
  else
     return p;
}
```

Einfügen im Suchbaum

1. Wir suchen den Schlüssel im Suchbaum.
2. Ist die Suche erfolglos, so endet sie bei einer `null`-Referenz, an dem wir den neuen Schlüssel dann anhängen.

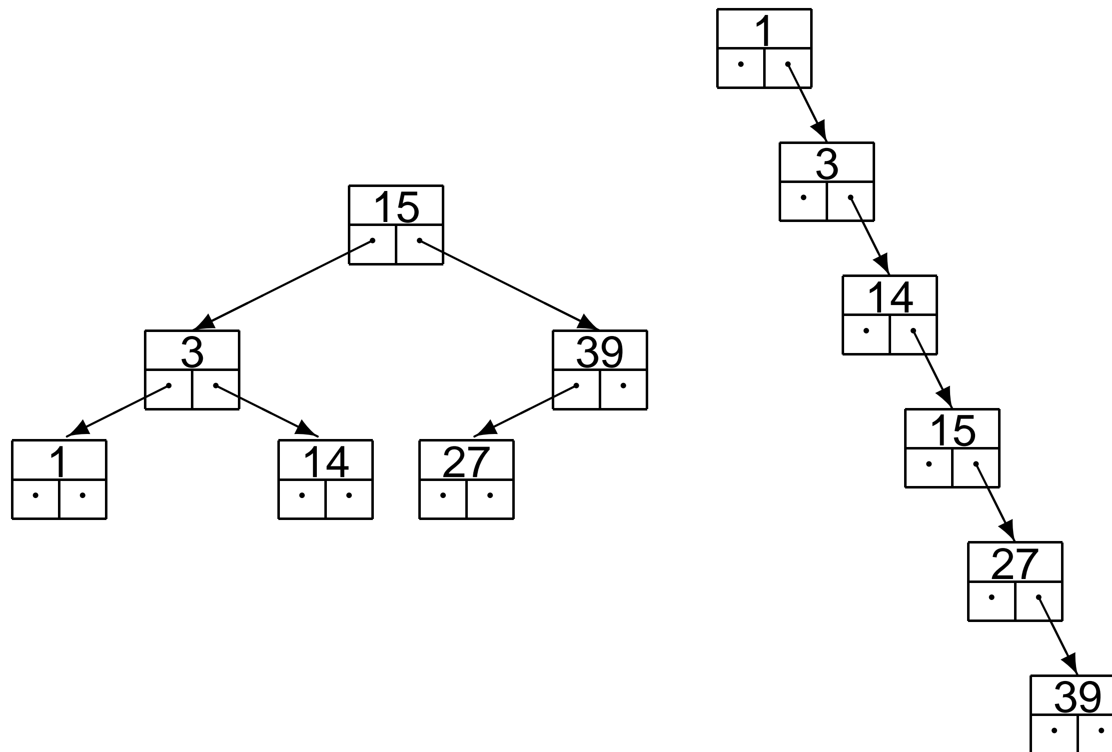
```
public void insert(int k) {
   root = insert(root, k);
}

private node insert(node p, int k) {
   if (p == null) // Suche erfolglos
      return new node(k);
   else if (k < p.key)
      p.left = insert(p.left,k);
   else if (k > p.key)
      p.right = insert(p.right,k);
                  // Schluessel schon vorhanden
   return p;
}
```

Sonderfälle

Die Struktur des resultierenden Baums hängt stark von der Reihenfolge ab, in der die Schlüssel eingefügt werden. Die minimale Höhe ist $\lceil \log_2 n \rceil$ und die maximale Höhe ist $n - 1$.

Resultierende Suchbäume für die Reihenfolgen 15, 39, 3, 27, 1, 14 und 1, 3, 14, 15, 27, 39:



Löschen eines Knotens

Wie kann ein Schlüssel gelöscht werden, sodass anschließend die Suchbaumeigenschaft noch erhalten ist?

- Wenn der Schlüssel nicht enthalten ist, ist nichts zu tun.
- Wenn der Schlüssel in einem Blatt gefunden wird, muss lediglich das Blatt entfernt werden.
- Hat der Knoten nur einen Nachfolger, so können wir ihn einfach durch seinen einzigen Nachfolger ersetzen.
- Schwieriger ist das Problem, wenn der zu entfernende Knoten zwei Nachfolger hat.

Symmetrische Nachfolger

Definition: Ein Knoten q heißt der symmetrische Nachfolger eines Knotens p, wenn q den kleinsten Schlüssel enthält, der größer oder gleich dem Schlüssel von p ist.

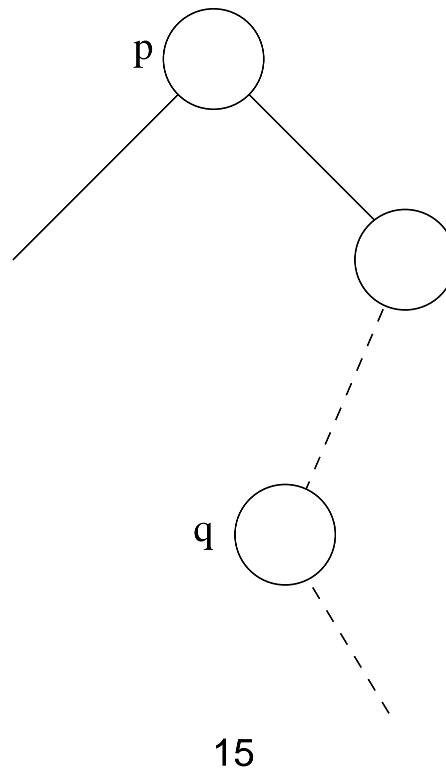
Beobachtungen:

1. Der symmetrische Nachfolger q von p ist der am weitesten links stehende Knoten im rechten Teilbaum von p.
2. Der symmetrische Nachfolger hat höchstens einen Nachfolger, welcher der rechte ist.

Auffinden des symmetrischen Nachfolgers

Beobachtung: Wenn p einen rechten Nachfolger hat, gibt es immer einen symmetrischen Nachfolger.

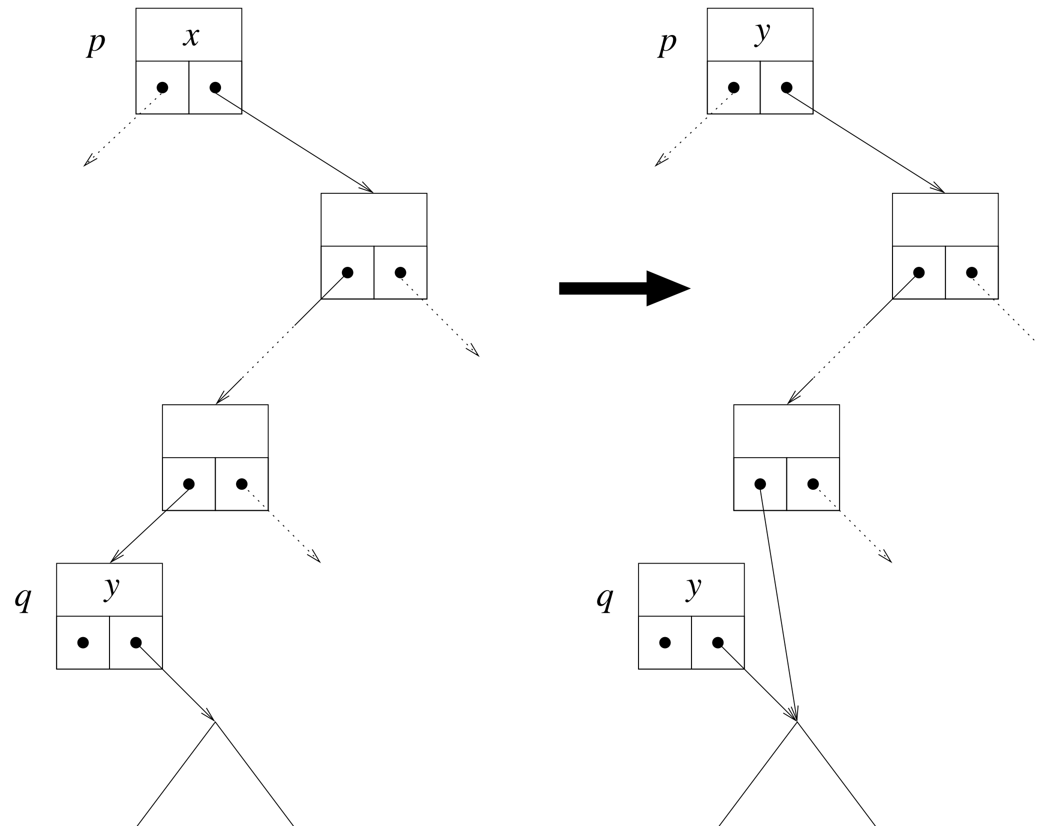
- Zunächst gehen wir zum rechten Nachfolger von p.
- Von dort aus gehen wir solange jeweils zum linken Nachfolger, bis wir einen Knoten ohne linken Nachfolger finden.



Idee der Löschoperation

Wir löschen p, indem wir den Inhalt von p durch den seines symmetrischen Nachfolgers q ersetzen. Danach löschen wir q.

Löschen von q ist einfach, weil q höchstens einen Nachfolger hat.



Die Methode für das Finden des symmetrischen Nachfolgers

Da wir den symmetrischen Nachfolger q anstelle von p löschen wollen, benötigen wir den Vorgängerknoten von q.

Dies leistet die Methode `predOfSymSucc`, die nur dann aufgerufen wird, wenn p genau zwei Nachfolger hat.

```
private node predOfSymSucc (node p) {
    if (p.right.left != null) {
        p = p.right;
        while (p.left.left != null)
            p = p.left;
    }
    return p;
}
```

Das Entfernen eines Knotens

1. Zunächst suchen wir den zu entfernenden Knoten im Baum.
2. Wenn der Knoten weniger als zwei Nachfolger hat, ist das Löschen einfach.
3. Wenn der Knoten genau 2 Nachfolger hat, ersetzen wir seinen Inhalt durch den des symmetrischen Nachfolgers und löschen den symmetrischen Nachfolger.
4. Die Top-Level-Routine ist:

```
public void remove(int k) {
    root = remove(root,k);
}
```

Die Methode für das Löschen

```
private node remove(node p, int k) {
    if (p == null) return p;
    if (k < p.key)
        p.left = remove(p.left, k);
    else if (k > p.key)
        p.right = remove(p.right, k);
    else {
        if (p.left == null)
            return p.right;
        if (p.right == null)
            return p.left;
        node q = predOfSymSucc(p);
        if (q == p) {  // rechter Nachfolger ist symmetrischer Nachfolger
            p.key = p.right.key;
            p.right = p.right.right;
        }
        else {
            p.key = q.left.key;
            q.left = q.left.right;
        }
    }
    return p;
}
```

Durchlaufreihenfolgen

- Eine weitere wichtige Operation auf Bäumen ist das Durchlaufen aller Knoten.
- Das Traversieren eines Baums ist beispielsweise notwendig, wenn die Elemente eines Baums ausgedruckt, kopiert oder modifiziert werden sollen.
- Die drei wichtigsten Durchlaufreihenfolgen für Binärbäume sind:
  - **inorder**
  - **preorder**
  - **postorder**

Charakterisierung der Durchlaufreihenfolgen

Ausgehend von der Wurzel p eines Baums sind die Durchlaufreihenfolgen wie folgt rekursiv definiert.

Inorder: Erst wird der linke Teilbaum von r traversiert, dann wird r bearbeitet, danach wird der rechte Teilbaum von r bearbeitet.

Preorder: Erst wird die Wurzel bearbeitet, dann wird der linke und anschließend der rechte Teilbaum traversiert.

Postorder: Erst wird der linke und dann der rechte Teilbaum traversiert. Anschließend wird die Wurzel bearbeitet.

Der Inorder-Durchlauf

Die Durchlaufreihenfolge ist: erst linker Teilbaum, dann Wurzel, dann rechter Teilbaum:

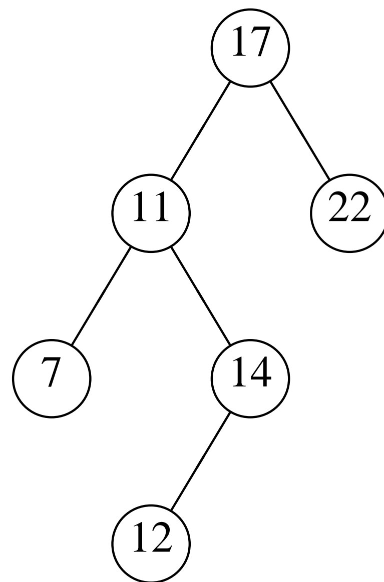
```
public void inOrder(){
    this.inOrder(this.root);

}
private void inOrder(node p){
    if (p != null){
        inOrder(p.left);
        System.out.println(p.key);
        inOrder(p.right);
    }

}
```

Die anderen beiden Durchlaufreihenfolgen werden analog implementiert.

Beispiel



Preorder:

17, 11, 7, 14, 12, 22

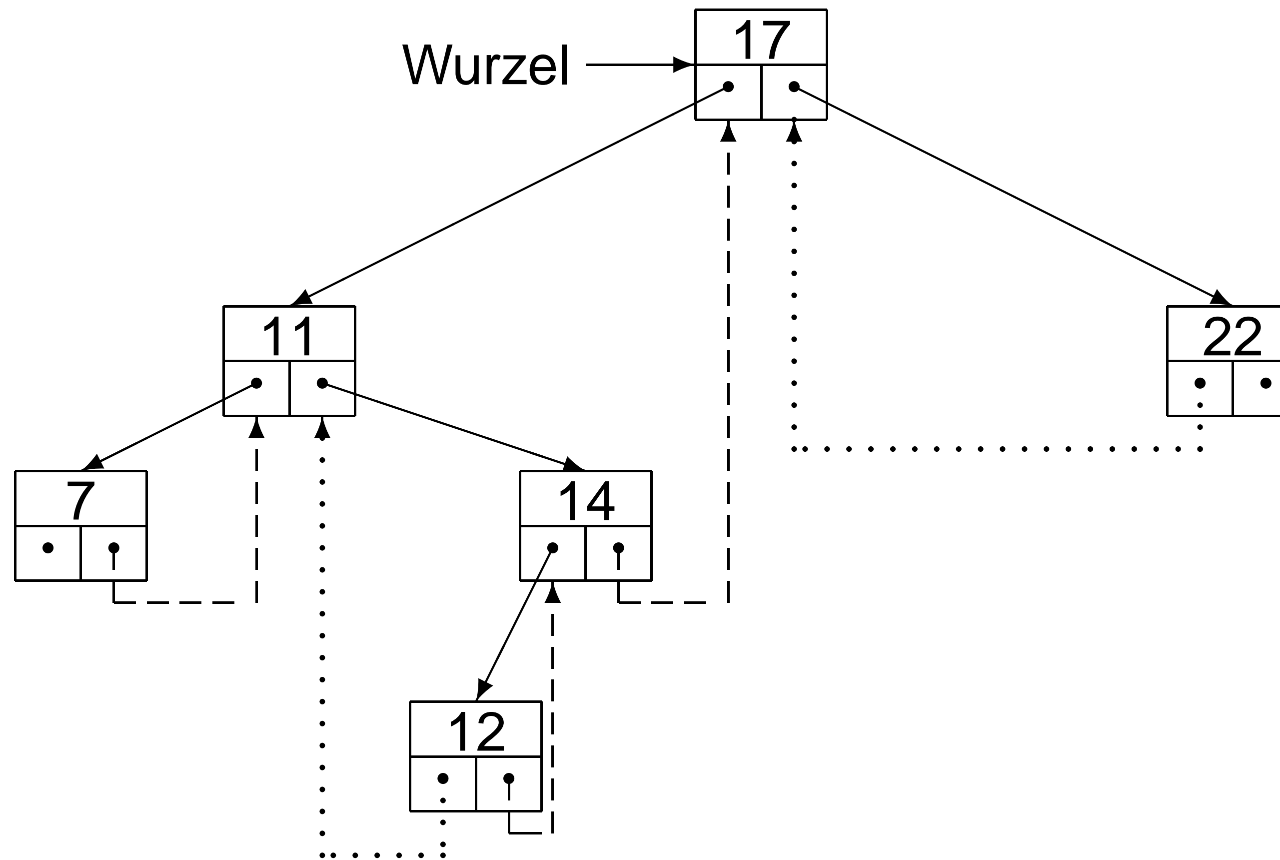
Postorder:

7, 12, 14, 11, 22, 17

Inorder:

7, 11, 12, 14, 17, 22

Nichtrekursive Varianten mit gefädelten Bäumen



Rekursion kann **vermieden** werden, wenn man anstelle der **`null`-Referenzen** sogenannte **Fädelungszeiger** auf die **Vorgänger bzw. Nachfolger** verwendet.

Sortieren mit natürlichen Suchbäumen

Idee: Bau für die Eingabefolge einen natürlichen Suchbaum auf und gib die Schlüssel in symmetrischer Reihenfolge (Inorder) aus.

Bemerkung: Abhängig von der Eingabereihenfolge kann der Suchbaum degenerieren.

Komplexität: Abhängig von der internen Pfadlänge

Schlechtester Fall: Vorsortierung $\Rightarrow \Omega(n^2)$ Schritte.

Bester Fall: Ausgeglichener Suchbaum (Analyse ähnlich wie die des mittleren Falles von binärer Suche) $\Rightarrow O(n \log n)$ — mit Faktor 1 vor $n \log n$

Mittlerer Fall: ?

Aufwand für den Average Case

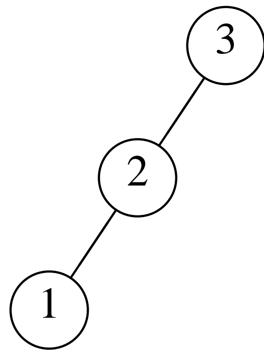
Zwei alternative Vorgehensweisen:

1. Random-Tree-Analyse, d.h. Mittelwert über alle möglichen Permutationen der einzufügenden Schlüssel.
2. Gestaltanalyse, d.h. Mittelwert über alle möglichen Bäume mit n Schlüsseln.

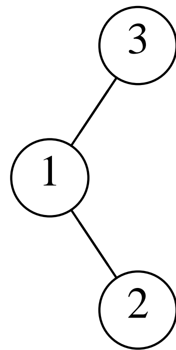
Unterschied:

1. $\approx 1.386 n \log_2 n - 0.846 n + O(\log n)$.
2. $\approx n \cdot \sqrt{\pi n} + O(n)$

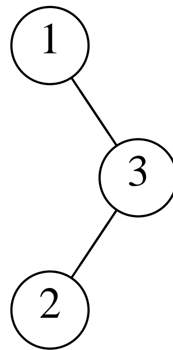
Ursache für den Unterschied



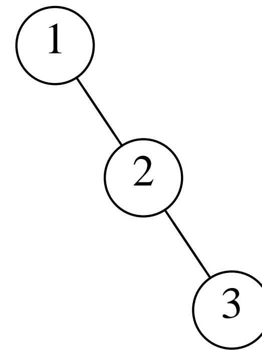
3,2,1



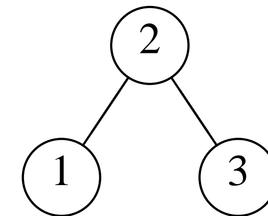
3,1,2



1,3,2



1,2,3



2,1,3 und 2,3,1

→ Bei der **Random-Tree-Analyse** werden **ausgeglichene Bäume häufiger gezählt**.

Random-Tree-Analyse (1): Interne Pfadlänge

Rekursive Definition:

1. Ist t der leere Baum, so ist
$$I(t) = 0.$$

2. Für einen Baum mit mit Wurzel t, linkem Teilbaum t_l, und rechtem Teilbaum t_r gilt:
$$I(t) := I(t_l) + I(t_r) + \text{ Zahl der Knoten von } t.$$

Offensichtlich gilt:

$$I(t) = \sum_{\substack{p \\ \text{mit } p \text{ Knoten von } t}} (\text{Tiefe}(p) + 1)$$

Zufällige Bäume

- Seien oBdA die **Schlüssel** $\{1, \ldots, n\}$ einzfügen.
- Sei ferner $s_1, \ldots, s_n$ eine **zufällige Permutation dieser Schlüssel**.
- Somit ist die **Wahrscheinlichkeit** $P(s_1 = k)$**, dass** s_1 **gerade den Wert** k **hat, genau** $1/n$.
- Wenn k **der erste Schlüssel** ist, wird k zur **Wurzel**.
- Dann enthalten der **linke Teilbaum** $k - 1$ **Elemente** (nämlich die Schlüssel $1, \ldots k - 1$) und der **rechte Teilbaum** $n - k$ **Elemente** (d.h. die Schlüssel $k + 1, \ldots n$).

Erwartete interne Pfadlänge

$EI(n)$: **Erwartungswert für die interne Pfadlänge eines zufällig erzeugten binären Suchbaums mit n Knoten**

Offensichtlich gilt:

$$\begin{aligned}
EI(0) &= 0 \\
EI(1) &= 1 \\
EI(n) &= \frac{1}{n}\sum_{k=1}^{n}(EI(k-1)+EI(n-k)+n) \\
&= n+\frac{1}{n}(\sum_{k=1}^{n}EI(k-1)+\sum_{k=1}^{n}EI(n-k))
\end{aligned}$$

Behauptung: $EI(n) \approx 1.386n\log_2 n - 0.846n + O(\log n)$.

Beweis (1)

$$EI(n+1) = (n+1) + \frac{2}{n+1} \cdot \sum_{k=0}^{n} EI(k)$$

und daher

$$\begin{aligned}(n+1) \cdot EI(n+1) &= (n+1)^2 + 2 \cdot \sum_{k=0}^{n} EI(k) \\ n \cdot EI(n) &= n^2 + 2 \cdot \sum_{k=0}^{n-1} EI(k)\end{aligned}$$

Aus den beiden letzten Gleichungen folgt

$$\begin{aligned}(n+1)EI(n+1) - n \cdot EI(n) &= 2n + 1 + 2 \cdot EI(n) \\ (n+1)EI(n+1) &= (n+2)EI(n) + 2n + 1 \\ EI(n+1) &= \frac{2n+1}{n+1} + \frac{n+2}{n+1} EI(n).\end{aligned}$$

Beweis (2)

Durch vollständige Induktion über n kann man zeigen, dass für alle $n \geq 1$ gilt:

$$EI(n) = 2(n+1)H_n - 3n$$

$H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$ ist die n-te harmonische Zahl, die wie folgt abgeschätzt werden kann:

$$H_n = \ln n + \gamma + \frac{1}{2n} + O(\frac{1}{n^2})$$

Dabei ist $\gamma = 0.5772\ldots$ die sogenannte Eulersche Konstante.

Beweis (3)

Damit ist

$$EI(n) = 2n \ln n - (3 - 2\gamma) \cdot n + 2 \ln n + 1 + 2\gamma + O(\frac{1}{n})$$

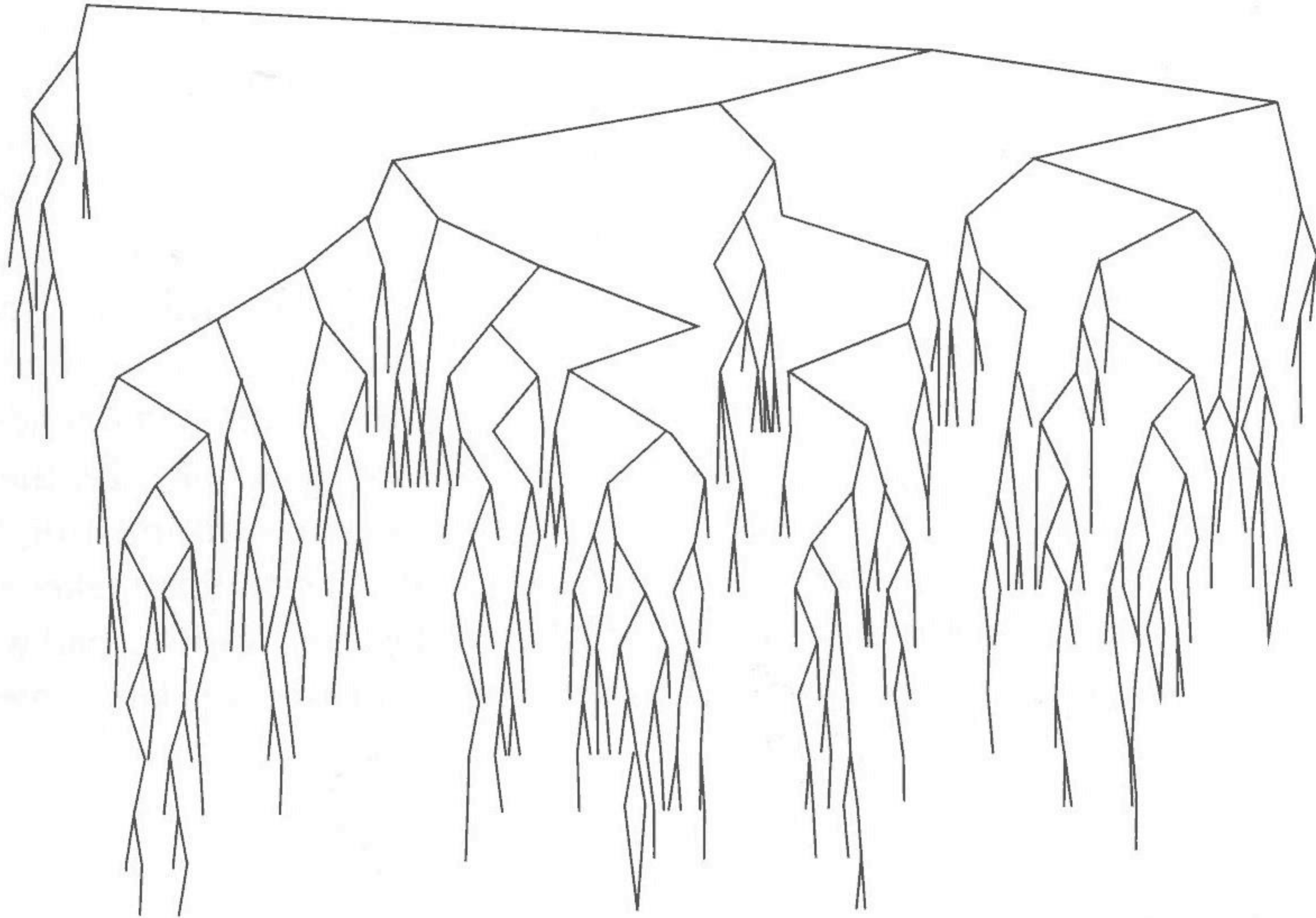
und daher

$$\begin{aligned}
\frac{EI(n)}{n} &= 2 \ln n - (3 - 2\gamma) + \frac{2 \ln n}{n} + \ldots \\
&= \frac{2}{\log_2 e} \cdot \log_2 n - (3 - 2\gamma) + \frac{2\ lnn}{n} + \ldots \\
&= \frac{2 \log_{10} 2}{\log_{10} e} \cdot \log_2 n - (3 - 2\gamma) + \frac{2 \ln n}{n} + \ldots \\
&\approx 1.386 \log_2 n - (3 - 2\gamma) + \frac{2 \ln n}{n} + \ldots
\end{aligned}$$

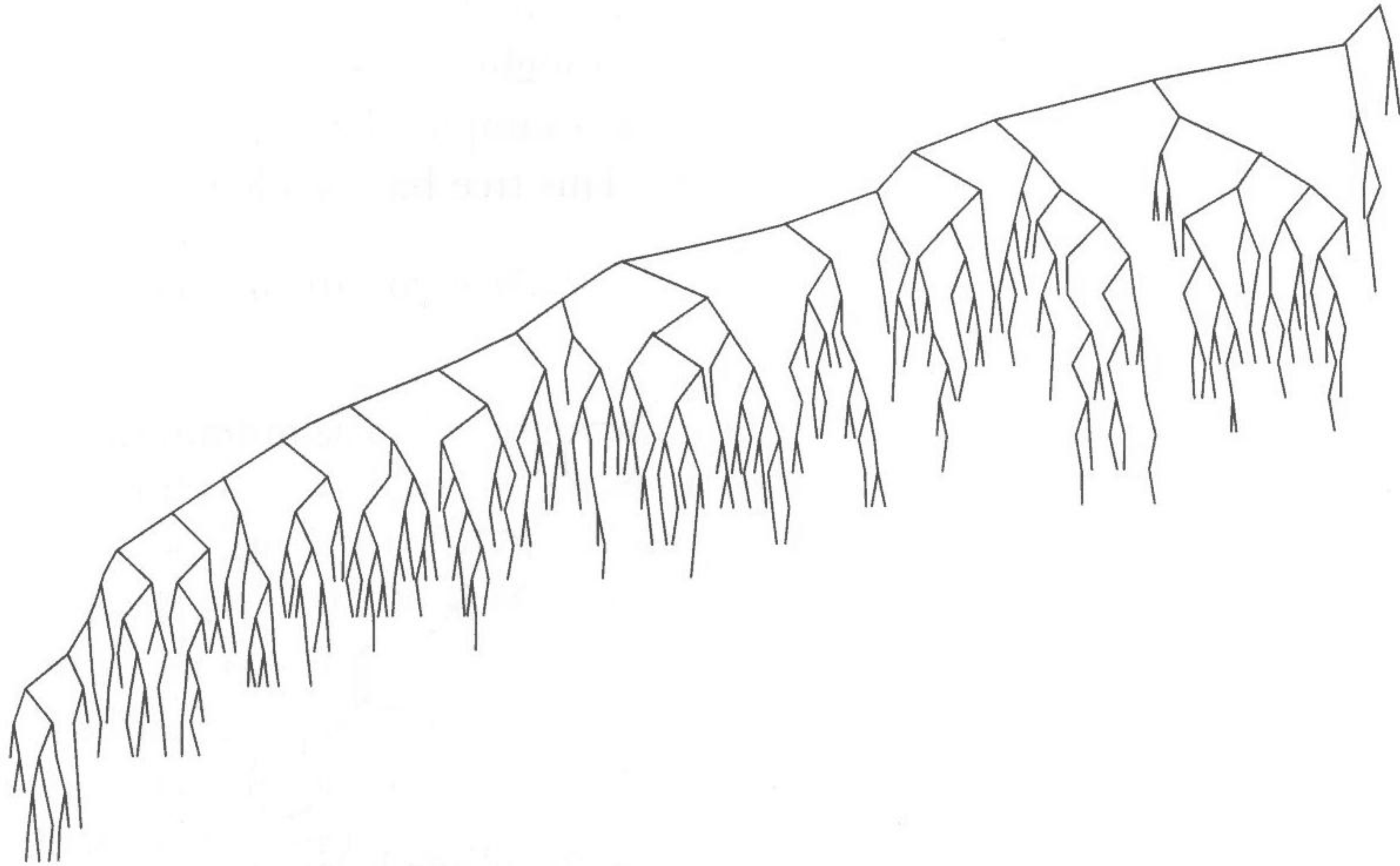
Beobachtungen

- **Suchen, Einfügen und Entfernen** eines Schlüssels ist bei einem **zufällig erzeugten binären Suchbaum** mit n Schlüsseln im Mittel in $O(\log_2 n)$ Schritten möglich.
- Im **schlechtesten Fall** kann der Aufwand jedoch $\Omega(n)$ betragen.
- Man kann nachweisen, dass der **mittlere Abstand eines Knotens von der Wurzel in einem zufällig erzeugten Baum nur etwa 40% über dem Optimum** liegt.
- Die **Einschränkung auf den symmetrischen Nachfolger verschlechtert** jedoch das **Verhalten**.
- Führt man in einem **zufällig erzeugten Suchbaum** mit n Schlüsseln n^2 **Update-Operationen** durch, so ist der **Erwartungswert für die durschnittliche Suchpfadlänge lediglich** $\Theta(\sqrt{n})$.

Typischer Binärbaum für eine zufällige Schlüsselsequenz



Resultierenter Binärbaum nach n^2 Updates



AVL-Bäume

Schöpfer: Adelson-Velskii und Landis (1962)

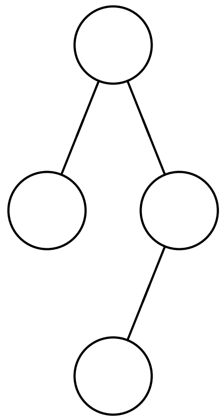
- **Suchen, Einfügen und Entfernen** eines Schlüssels in einem zufällig erzeugten Suchbaum mit n Schlüsseln ist im Mittel in $O(\log_2 n)$ Schritten ausführbar.
- Der **Worst Case** liegt jedoch bei $\Omega(n)$.
- **Idee von AVL-Bäumen**: Modifizierte Prozeduren zum Einfügen und Löschen, die ein **Degenerieren des Suchbaums verhindern**.
- Ziel von AVL-Bäumen: **Höhe** sollte $O(\log_2 n)$ und das **Suchen, Einfügen und Löschen** sollte **in logarithmischer Zeit** möglich sein.

Definition von AVL-Bäumen

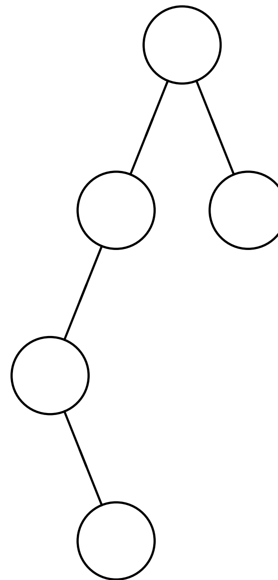
Definition: Ein binärer Suchbaum heißt **AVL-Baum**, wenn für jeden Knoten v gilt, dass sich die **Höhe des rechten Teilbaumes** $h(T_r)$ von v und die **Höhe des linken Teilbaumes** $h(T_l)$ von v um **maximal 1 unterscheiden**.

Balancegrad: $\mathrm{bal}(v) = h(T_r) - h(T_l) \in \{-1, 0, 1\}$

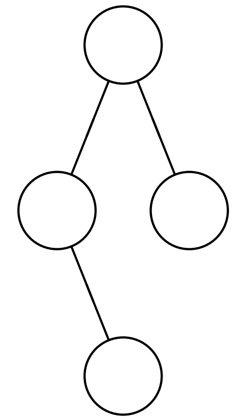
Beispiele



AVL-Baum



kein AVL-Baum



AVL-Baum

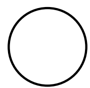
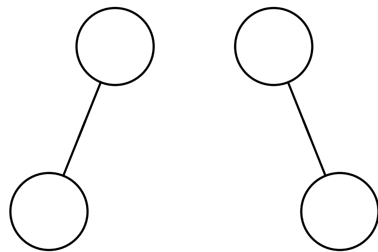
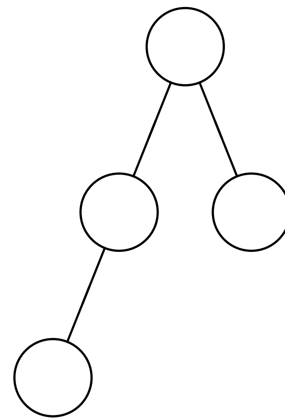
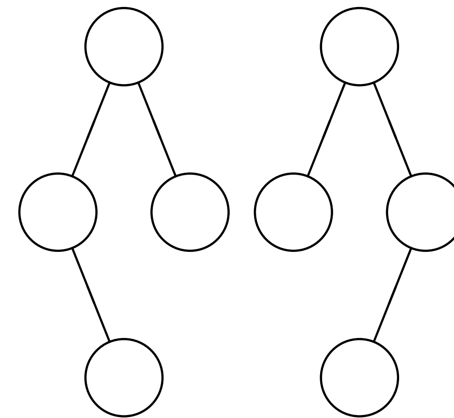
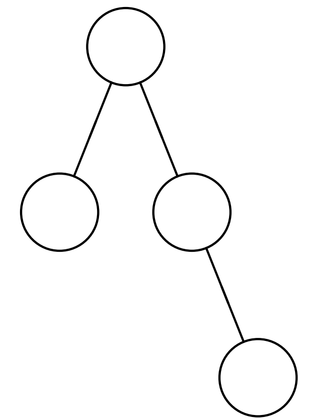
Eigenschaften von AVL-Bäumen

- AVL-Bäume können nicht zu linearen Listen degenerieren.
- AVL-Bäume mit n Knoten haben eine Höhe von $O(\log n)$.

Offenbar gilt:

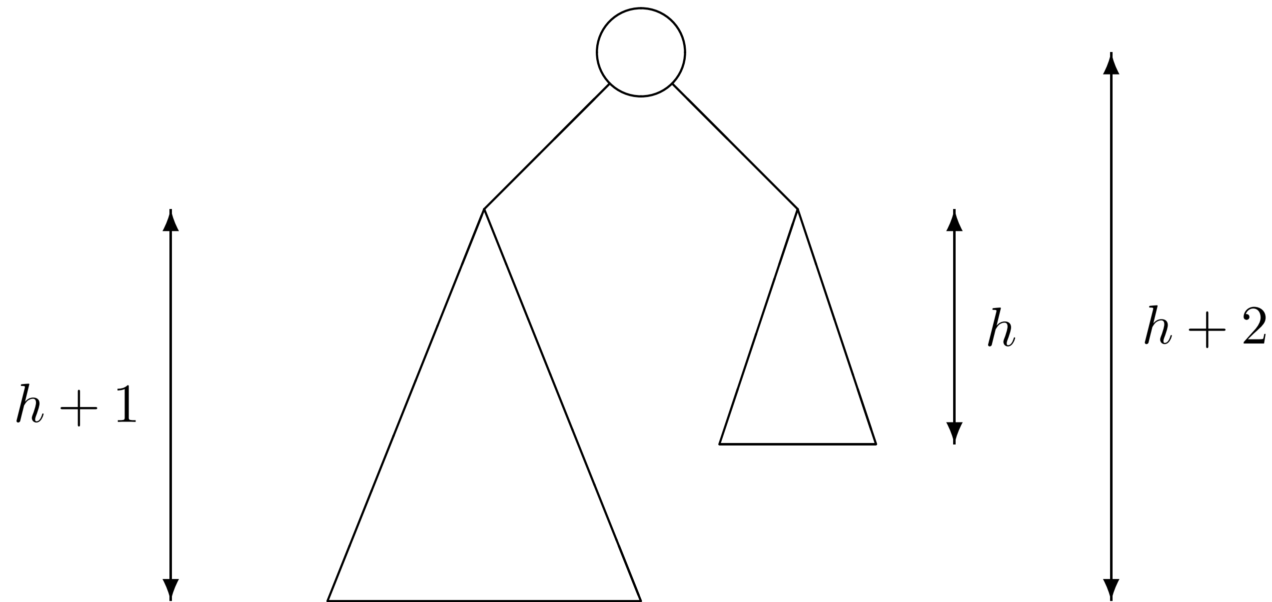
- Ein AVL-Baum der Höhe 0 hat 0 Blätter
- Ein AVL-Baum der Höhe 1 hat 1 Blatt
- ein AVL-Baum der Höhe 2 mit minimaler Blattzahl hat 1 Blatt
- ...
- Wieviele Blätter hat ein AVL-Baum der Höhe h mit minimaler Blattzahl?

AVL-Bäume mit minimaler Blattanzahl

Höhe 1 Höhe 2 Höhe 3

Minimale Blattanzahl von AVL-Bäumen mit Höhe h

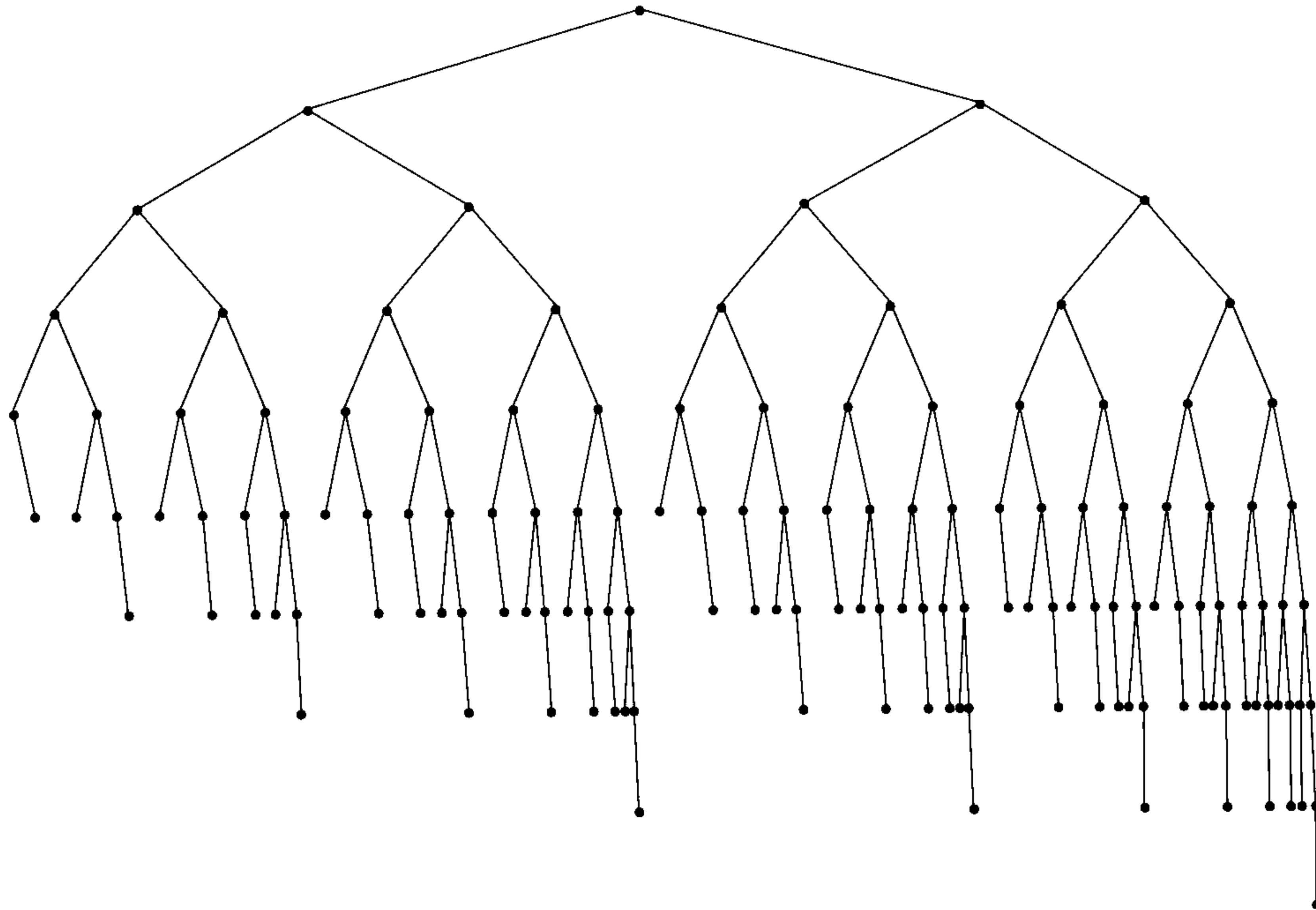


Folgerung: Ein AVL-Baum der Höhe h hat mindestens F_h Blätter mit

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_{i+2} &= F_{i+1} + F_i
\end{aligned}
$$

$\leadsto$ F_i ist die i-te **Fibonacci-Zahl**.

Minimaler AVL-Baum der Höhe 9



Höhe eines AVL-Baumes

Satz: Die Höhe h eines AVL-Baumes mit n Blättern (und $n-1$ inneren Knoten) beträgt höchstens, $1.44\ldots \cdot \log_2 n + 1$, d.h.

$$h \leq 1.44\ldots \log_s n + 1.$$

Beweis: Für die Fibonacci-Zahlen gilt:

$$F_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+1} \right) \approx 0.7236\ldots \cdot 1.618\ldots^h$$

Wegen

$$n \geq F_h \approx 0.7236\ldots \cdot 1.618\ldots^h$$

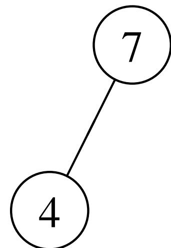
folgt somit

$$h \leq \frac{1}{\log_2 1.618\ldots} \cdot \log_2 n - \frac{\log_2 0.7236\ldots}{\log_2 1.618\ldots} \leq 1.44\ldots \cdot \log_2 n + 1.$$

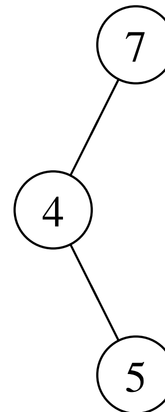
Einfügen in einen AVL-Baum

- Bei jeder Modifikation des Baums müssen wir garantieren, dass die AVL-Baum-Eigenschaft erhalten bleibt.

Ausgangssituation:



Nach Einfügen von 5:



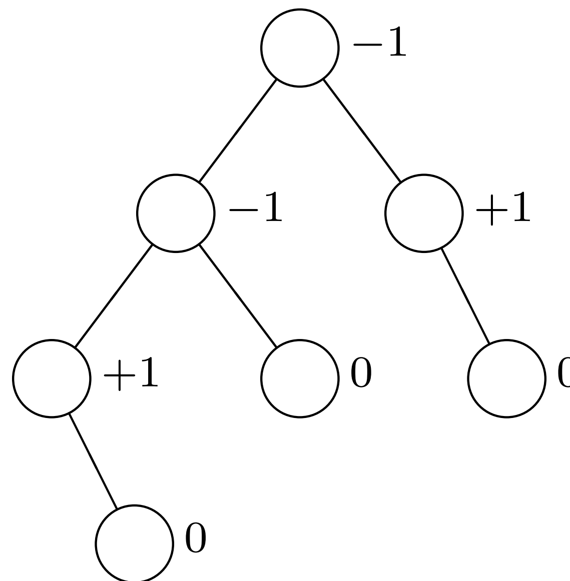
Problem: Wie können wir den neuen Baum so modifizieren, dass ein AVL-Baum daraus entsteht?

Speichern des Balancegrads in den Knoten

- Um die AVL-Baum-Eigenschaft wiederherzustellen, genügt es, in jedem Knoten den Balancegrad mitzuführen.
- Laut Definition gilt

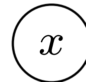
$$\mathrm{bal}(p) = h(p.right) - h(p.left) \in \{-1, 0, 1\}.$$

Beispiel:



Die verschiedenen Situationen beim Einfügen in den AVL-Baum

1. Der **Baum ist leer**:

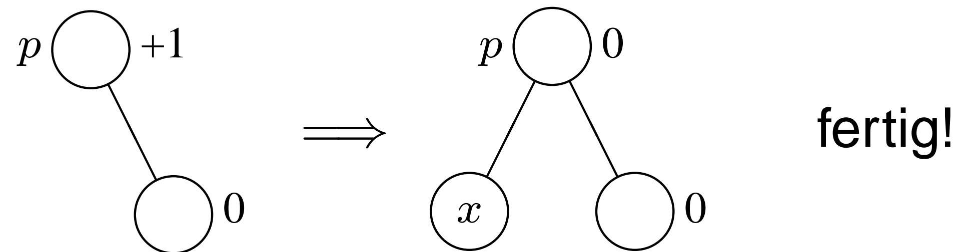
   Offensichtlich sind wir fertig.

2. Der **Baum ist nicht leer** und die **Suche endet bei einem Knoten** p.
   Wegen $\mathrm{bal}(p) \in \{-1, 0, 1\}$ muss gelten, dass entweder

   - der linke Nachfolger von p leer und der rechte Nachfolger ein Blatt ist (**Fall 1**) oder
   - der rechte Nachfolger von p leer und der linke Nachfolger ein Blatt ist (**Fall 2**) oder
   - p ein Blatt ist (**Fall 3**).

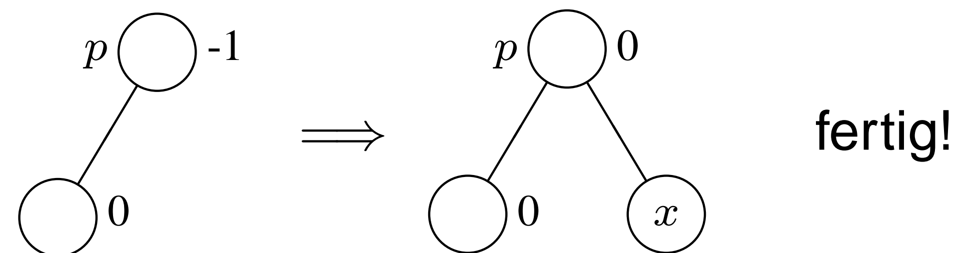
Die Fälle 1 und 2

Fall 1: $[bal(p) = +1]$ und $x < p.key$, da Suche bei p endet.



fertig!

Fall 2: $[bal(p) = -1]$ und $x > p.key$, da Suche bei p endet.



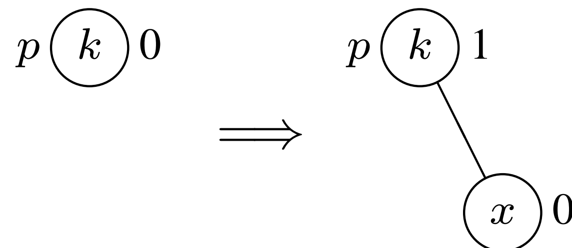
fertig!

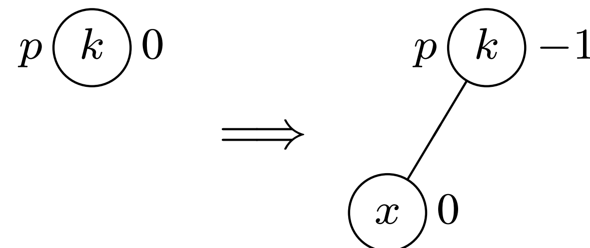
Beide Fälle sind unkritisch. Die **Höhe des Teilbaums**, in dem p sich befindet, **ändert sich nicht**.

Der kritische Fall 3

- Wenn p ein **Blatt** ist, **ändert sich die Höhe des Teilbaums**, in dem p sich befindet, **um 1**.
- Deswegen **ändern sich** auch die **Balancegrade oberhalb von p**.
- Wiederum unterscheiden wir, ob wir den neuen Schlüssel als rechten oder linken Nachfolger einfügen müssen:

$[bal(p) = 0 \wedge x > p.key]$ $\quad\quad$ $[\mathrm{bal}(p) = 0 \wedge x < p.key]$





- In beiden Fällen benötigen wir eine Prozedur $upin(p)$, die den **Suchpfad zurückläuft**, die **Balancegrade prüft** und Umstrukturierungen (so genannte **Rotationen** oder **Doppelrotationen**) durchführt.

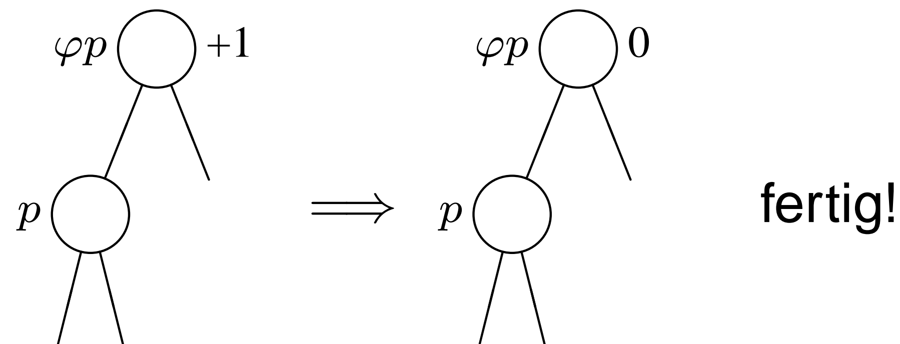
Fall 3: Die Prozedur $upin(p)$

- Wenn $upin(p)$ aufgerufen wird, ist **stets** $\mathrm{bal}(p) \in \{1, -1\}$ und die **Höhe des Teilbaums mit Wurzel p ist um 1 gewachsen**.
- $upin(p)$ **startet bei p und geht schrittweise nach oben** (ggf. bis zur Wurzel).
- In jedem Schritt wird dabei versucht, die AVL-Baum-Eigenschaft wiederherzustellen.
- Wir konzentrieren uns im folgenden auf die Situation, dass p linker Nachfolger seines Vorgängers φp ist.
- Die Situation, dass p rechter Nachfolger seines Vorgängers φp ist, kann analog behandelt werden.

Fall 3.1: $\mathrm{bal}(\varphi p) = 1$

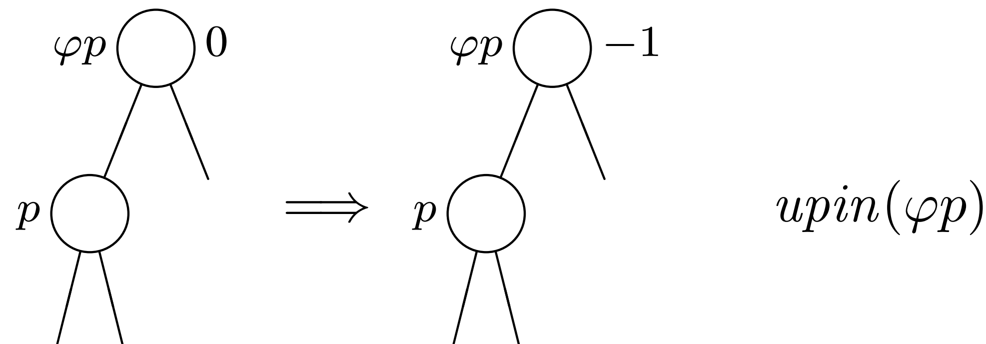
Wiederum unterscheiden wir drei Fälle:

1. Der Vorgänger φp hat den Balancegrad $+1$. Da sich die Höhe des Teilbaums mit Wurzel p als linker Nachfolger von φp um 1 erhöht hat, genügt es, den Balancegrad von φp auf 0 zu setzen:



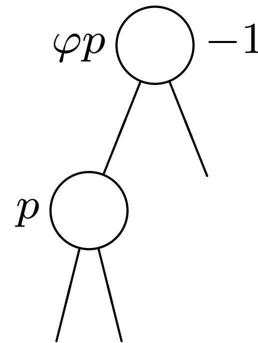
Fall 3.2: $\mathrm{bal}(\varphi p) = 0$

2. Der Vorgänger φp hat den Balancegrad 0. Da sich die Höhe des Teilbaums mit Wurzel p als linker Nachfolger von φp sich um 1 erhöht hat, ändert sich der Balancegrad von φp auf -1. Da sich gleichzeitig die Höhe des Teibaums mit Wurzel φp verändert hat, müssen wir $upin$ rekursiv mit φp als Argument aufrufen.



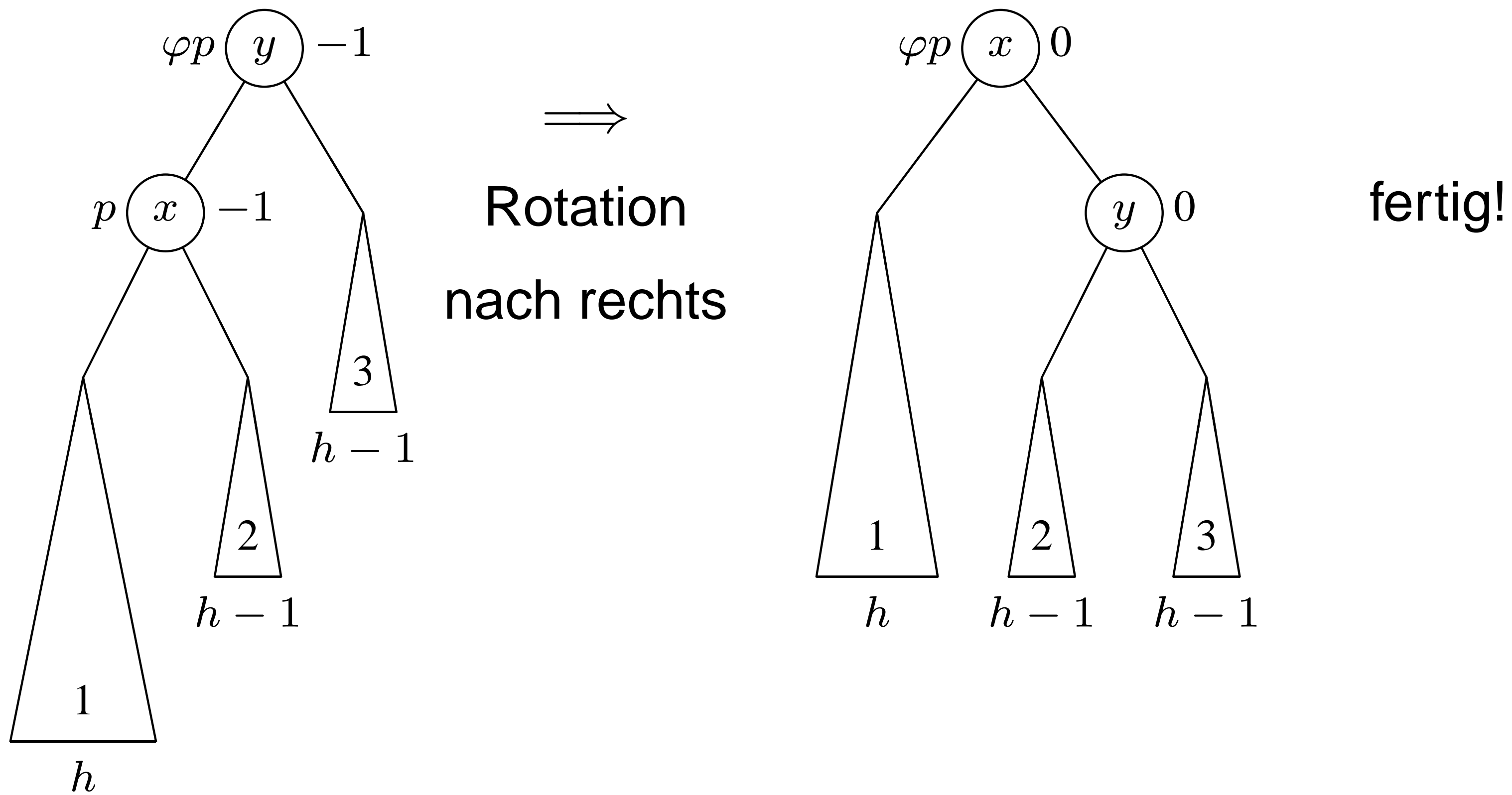
Der kritische Fall 3.3: $\mathrm{bal}(\varphi p) = -1$

3.



- Wenn $\mathrm{bal}(\varphi p) = -1$ und die Höhe des linken Teilbaums p von φp um 1 gewachsen ist, muss die **AVL-Baum-Eigenschaft in φp verletzt** sein.
- In diesem Fall müssen wir den **Baum umstrukturieren**.
- Erneut **unterscheiden wir zwei Fälle**, nämlich $\mathrm{bal}(p) = -1$ **(Fall 3.3.1)** und $\mathrm{bal}(p) = +1$ **(Fall 3.3.2)**.
- Da wir $upin(\varphi p)$ **rekursiv** ausführen, muss gelten, dass $\mathrm{bal}(p) \neq 0$, da wir andernfalls bei p bereits abgebrochen hätten (Fall 3.1).

Fall 3.3.1: $\mathrm{bal}(\varphi p) = -1 \land \mathrm{bal}(p) = -1$



Ist der resultierende Baum noch ein Suchbaum?

Es muss garantiert sein, dass der resultierende Baum die

1. **Suchbaumeigenschaft** und die
2. **AVL-Baum-Eigenschaft** erfüllt.

Suchbaumeigenschaft: Da der ursprüngliche Baum die Suchbaumeigenschaft erfüllt, muss gelten:

- Alle **Schlüssel in Baum 1 sind kleiner als x**.
- Alle **Schlüssel in Baum 2 sind größer als x und kleiner als y**.
- Alle **Schlüssel in Baum 3 sind größer als y (und x)**.

Daher erfüllt auch der resultierende Baum die Suchbaumeigenschaft.

Ist der resultierende Baum balanciert?

AVL-Baum-Eigenschaft: Da der ursprüngliche Baum ein AVL-Baum war, muss gelten:

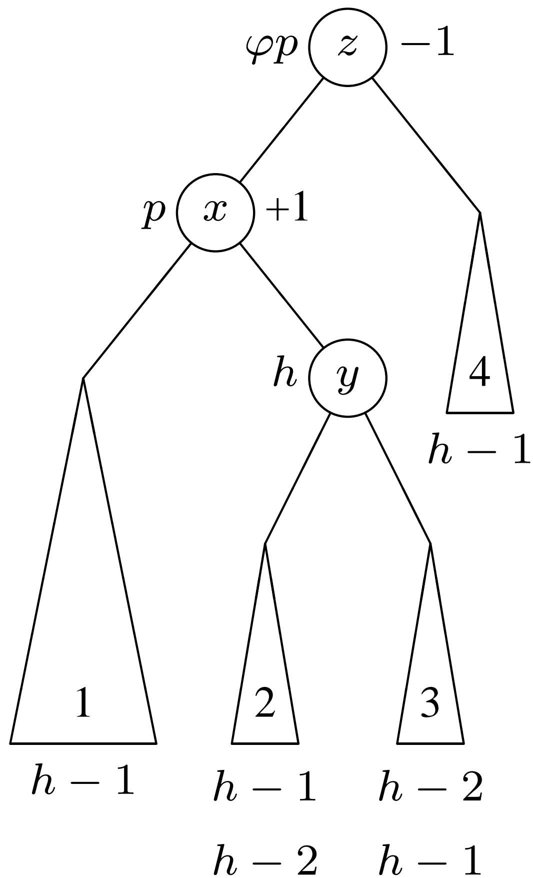
- Wegen $\mathrm{bal}(\varphi p) = -1$ haben **Baum 2 und Baum 3 die gleiche Höhe $h - 1$**.
- Wegen $\mathrm{bal}(p) = -1$ nach dem Einfügen, hat **Baum 1 die Höhe h, während Baum 2 die Höhe $h - 1$ hat**.

Damit gilt nach der Rotation:

- Der **Knoten, der y enthält, hat Balancegrad 0**.
- Der **Knoten φp hat Balancegrad 0**.

Somit ist der **AVL-Baum-Eigenschaft wieder hergestellt**.

Fall 3.3.2: $\mathrm{bal}(\varphi p) = -1 \wedge \mathrm{bal}(p) = 1$

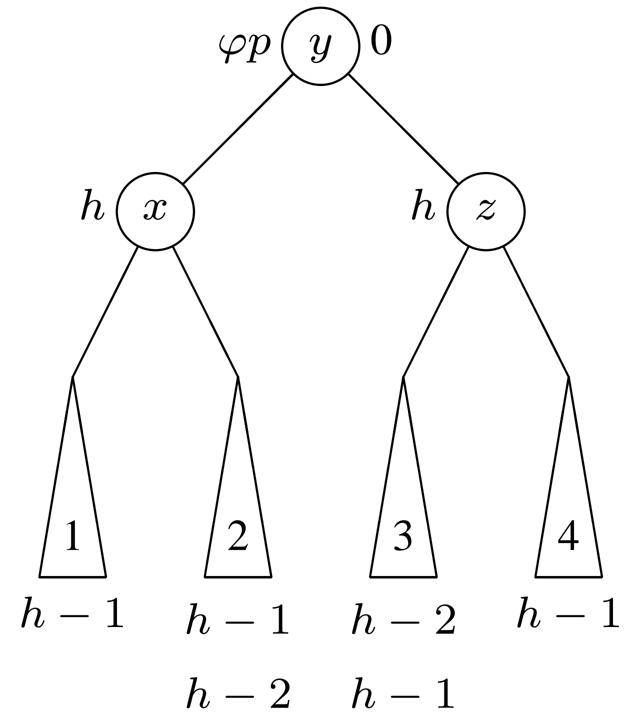


$\Longrightarrow$

Doppel-

rotation

links-rechts



fertig!

Eigenschaften der Teilbäume

1. Der neue Schlüssel muss in den rechten Teilbaum von p eingefügt worden sein.

2. Die Bäume 2 und 3 müssen unterschiedliche Höhe haben, weil sonst die Methode $upin$ nicht aufgerufen worden wäre.

3. Die einzig mögliche Kombination der Höhen in den Bäumen 2 und 3 ist somit $(h-1, h-2)$ und $(h-2, h-1)$, sofern sie nicht leer sind.

4. Wegen $\mathrm{bal}(p) = 1$ muss Baum 1 die Höhe $h-1$ haben.

5. Schließlich muss auch Baum 4 die Höhe $h-1$ haben (wegen $\mathrm{bal}(\varphi p) = -1$).

Somit erfüllt der resultierende Baum ebenfalls die AVL-Baum-Eigenschaft.

Suchbaumeigenschaft

Es gilt:

1. Die Schlüssel in Baum 1 sind sämtlich kleiner als x.
2. Die Schlüssel in Baum 2 sind sämtlich kleiner als y aber größer als x.
3. Die Schlüssel in Baum 3 sind alle größer als y und x aber kleiner als z.
4. Die Schlüssel in Baum 4 sind alle größer als x, y und z.

Daher hat auch der durch die Doppelrotation entstandene Baum die Suchbaumeigenschaft.

Hinweise

- Wir haben lediglich den Fall betrachtet, dass p linker Nachfolger seines Vorgängers φp ist.
- Der Fall, dass p rechter Nachfolger seines Vorgängers φp ist, kann analog behandelt werden.
- Um die Methode $upin(p)$ effizient zu implementieren, müssen wir bei der Suche nach der Einfügestelle des neuen Schlüssels eine Liste aller besuchten Knoten anlegen.
- Dann können wir diese Liste bei den rekursiven Aufrufen nutzen, um jeweils zum Vorgänger überzugehen und ggf. die erforderlichen Rotationen oder Doppelrotationen auszuführen.

Das Einfügen in einen nicht leeren AVL-Baum

1. $\mathrm{bal}(p) = +1 \wedge x < p.key \leadsto$ Anhängen links von p, fertig.
2. $\mathrm{bal}(p) = -1 \wedge x > p.key \leadsto$ Anhängen rechts von p, fertig.
3. p ist Blatt, d.h. jetzt gilt $\mathrm{bal}(p) \in \{-1, +1\} \leadsto upin(p)$

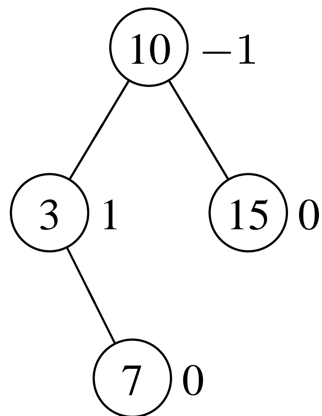
Die Methode $upin(p)$:

1. p ist **linker Nachfolger von** φp
   (a) $\mathrm{bal}(\varphi p) = +1 \leadsto bal(\varphi p) = 0$, fertig.
   (b) $\mathrm{bal}(\varphi p) = 0 \leadsto bal(\varphi p) = -1, upin(\varphi p)$
   (c) i. $\mathrm{bal}(\varphi p) = -1 \wedge \mathrm{bal}(p) = -1 \leadsto$ **Rotation nach rechts**, fertig.
       ii. $\mathrm{bal}(\varphi p) = -1 \wedge \mathrm{bal}(p) = 1 \leadsto$ **Doppelrotation links-rechts**, fertig.
2. p ist **rechter Nachfolger von** φp.

   ...

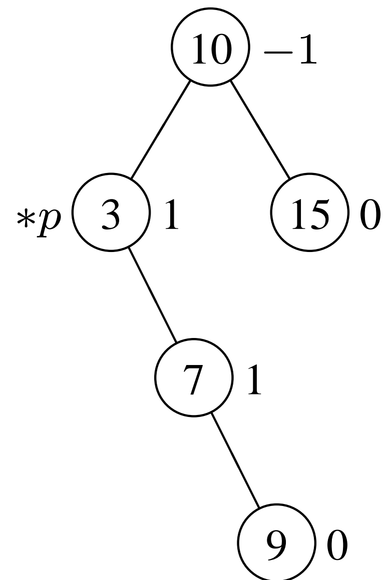
Ein Beispiel (1)

Ausgangssituation:



Ein Beispiel (2)

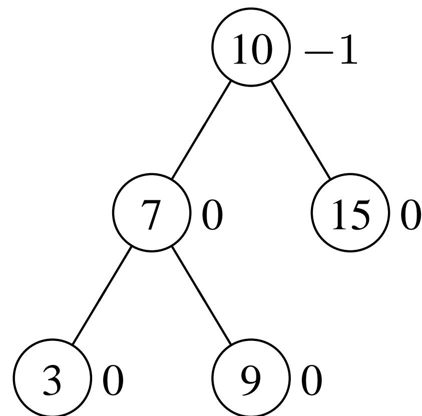
Einfügen von Schlüssel 9:



AVL-Baum-Eigenschaft ist verletzt!

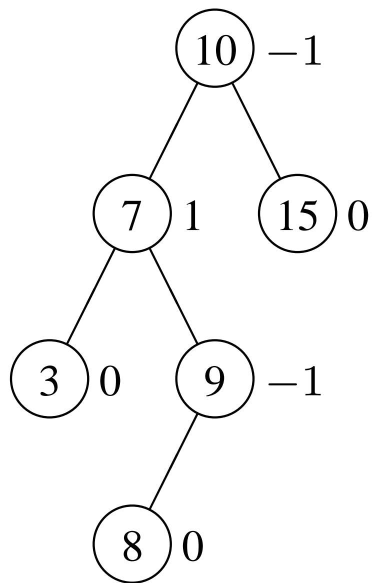
Ein Beispiel (3)

Linksrotation bei $*p$ liefert:

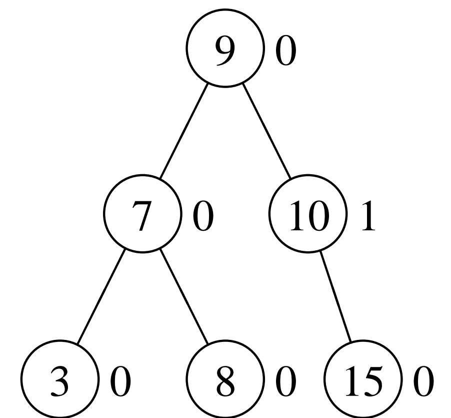


Ein Beispiel (4)

Einfügen von 8 mit anschließender Doppelrotation liefert:



$\Longrightarrow$
links-rechts



Entfernen aus einem AVL-Baum

- Wie gehen ähnlich vor wie bei Suchbäumen:
  1. Suche nach dem zu entfernenden Schlüssel.
  2. Falls der Schlüssel nicht enthalten ist, sind wir fertig.
  3. Andernfalls unterscheiden wir drei Fälle:
     (a) Der zu löschende Knoten hat keine Nachfolger.
     (b) Der zu löschende Knoten hat genau einen Nachfolger.
     (c) Der zu löschende Knoten hat zwei Nachfolger.
- Nach dem Löschen eines Knotens kann ggf. die AVL-Baum-Eigenschaft verletzt sein (wie beim Einfügen).
- Dies muss entsprechend behandelt werden.

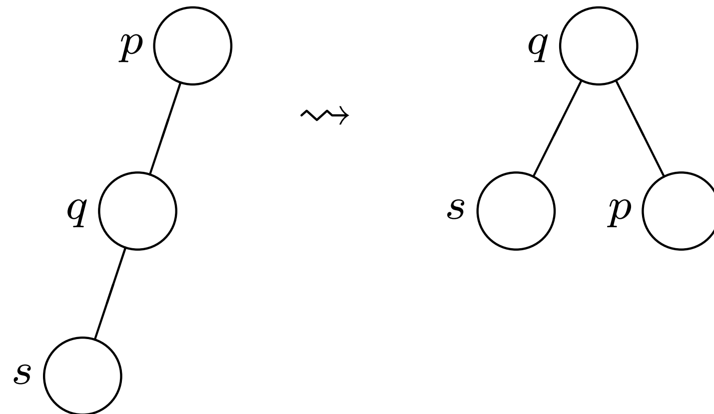
Fall 1: Der zu löschende Knoten ist ein Blatt

- Falls der zu löschende Knoten nicht der einzige Knoten im Baum war, muss er, da er ein Blatt war, wenigstens einen Vorgänger p haben.
- Weil der gelöschte Knoten ein Blatt war, hat der Teilbaum mit Wurzel p die Höhe 2 oder 3 haben.
- Entsprechend muss der andere Teilbaum von p mit Wurzel q die Höhe $0, 1$ oder 2 haben.

Löschen eines Blattes

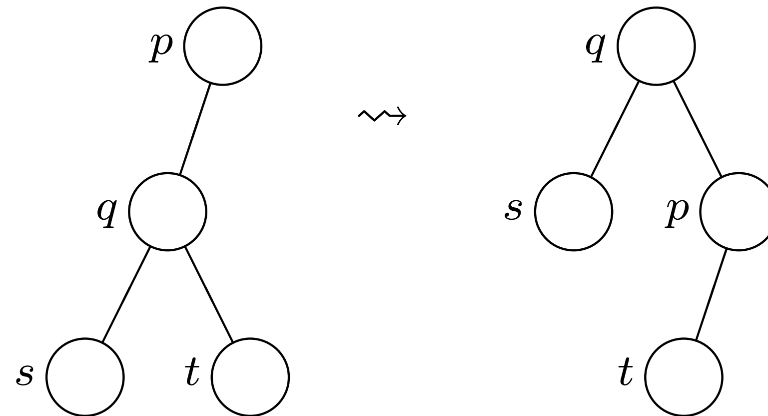
- Hat der Teilbaum mit Wurzel q die **Höhe 1**, so verändert sich der Balancegrad von p auf $+1$ oder -1, je nachdem auf welcher Seite von p das gelöschte Blatt war.
- Gleichzeitig hat sich die Höhe von p nicht verändert und wir sind **fertig**.
- Hat der Teilbaum mit Wurzel q die **Höhe 0**, so ändert man die Balance von von p von $+1$ oder -1 auf 0.
- Daraus folgt, dass die Höhe des Teilbaums mit Wurzel p gesunken sein muss.
- Wir rufen daher eine Methode $upout(p)$ auf, welche die AVL-Ausgeglichenheit wiederherstellt.
- Hat der Teilbaum mit Wurzel q die **Höhe 2**, d.h. war q kein Blatt, dann führen wir eine Rotation oder Doppelrotation aus.
- Dabei kann es passieren, dass die Höhe der neuen Wurzel r dieses Teilbaums um 1 gesunken ist, so dass wir $upout(r)$ aufrufen müssen, um die AVL-Baum-Eigenschaft wiederherzustellen.

Situation 1, falls Höhe von $q = 2$



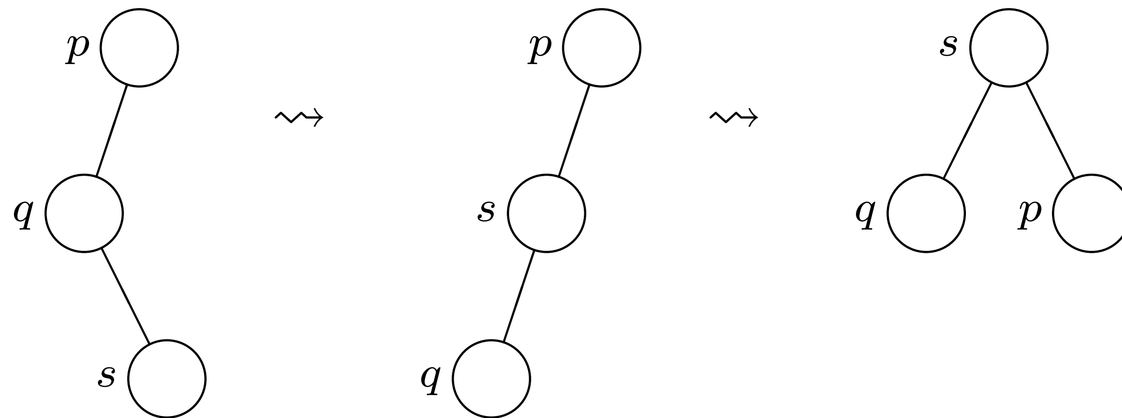
Rotation nach rechts und $upout(q)$.

Situation 2, falls Höhe von $q = 2$



Rotation nach rechts

Situation 3, falls Höhe von $q = 2$

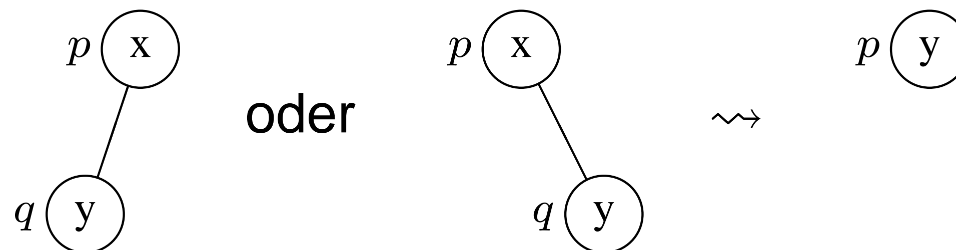


Doppelrotation links-rechts und $upout(s)$

Hinweis: Die weiteren Situationen sind symmetrisch zu den bisher geschilderten.

Fall 2: Der zu löschende Knoten hat genau einen Nachfolger

- Sei q **der einzige Nachfolger des zu löschenden Knotens** p.
- Wegen der AVL-Baum-Eigenschaft muss q **ein Blatt** sein.
- Wir **ersetzen den Inhalt von** p **durch den Inhalt von** q und **löschen** q.
- Damit ist jetzt p ein Blatt.
- Da die **Höhe des Teilbaums mit Wurzel** p **um** 1 **gesunken** ist, rufen wir die Methode $upout(p)$ auf, um ggf. die AVL-Baum-Eigenschaft wiederherzustellen.



Aufruf von $upout(p)$

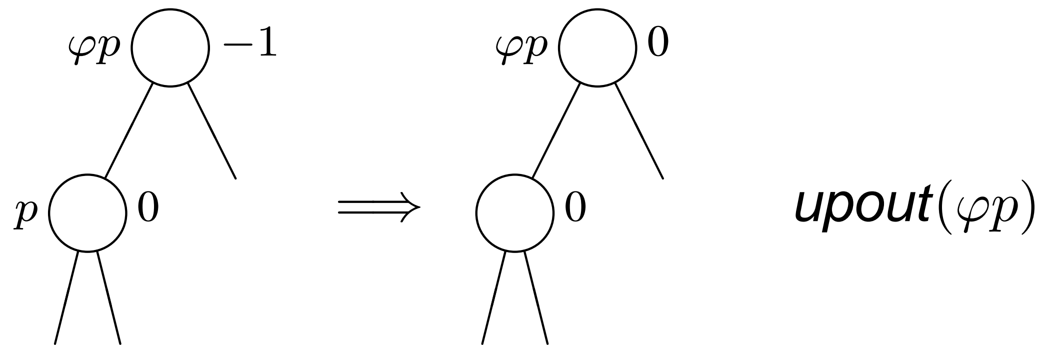
Der Fall 3: Der zu löschende Knoten hat genau 2 Nachfolger

- Wir gehen zunächst so vor, wie bei Suchbäumen:
  1. Wir **ersetzen den Inhalt des zu löschenden Knotens p durch den seines symmetrischen Nachfolgers q**.
  2. Danach **löschen** wir den Knoten q.
- Da q **höchstens einen Nachfolger** (den rechten) haben kann, **treffen für q die Fälle 1 und 2 zu**.

Die Methode $upout$

- Die Methode $upout$ funktioniert ähnlich wie die Methode $upin$.
- Sie wird entlang des Suchpfads rekursiv aufgerufen und adjustiert die Balancegrade durch Rotationen und Doppelrotationen.
- Wenn $upout$ für einen Knoten p aufgerufen wird, gilt (s.o.):
  1. $\mathrm{bal}(p) = 0$
  2. Die Höhe des Teilbaums mit Wurzel p ist um 1 gefallen.
- $upout$ wird nun so lange rekursiv aufgerufen, wie diese beiden Bedingungen gelten (Invariante).
- Wiederum unterscheiden wir 2 Fälle, abhängig davon, ob p linker oder rechter Nachfolger seines Vorgängers φp ist.
- Da beide Fälle symmetrisch sind, behandeln wir im Folgenden nur den Fall, dass p linker Nachfolger von φp ist.

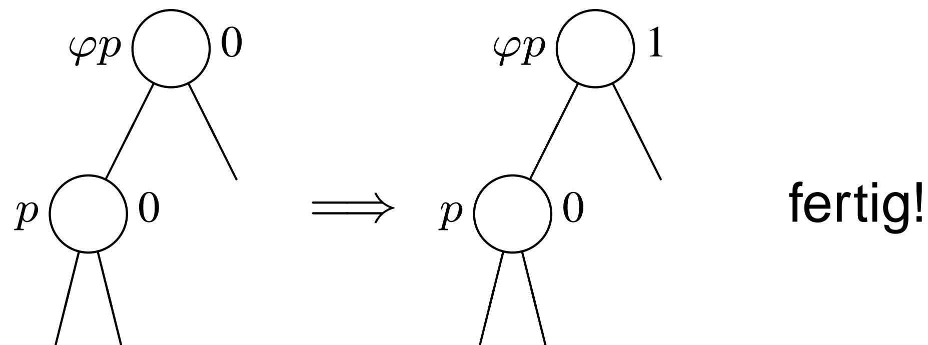
Fall 1.1: p ist linker Nachfolger von φp und $\mathrm{bal}(\varphi p) = -1$



upout(φp)

- Da die Höhe des Teilbaums mit Wurzel p um 1 gesunken ist, ändert sich die Balance von φp zu 0.
- Damit ist aber die Höhe des Teilbaums mit Wurzel φp um 1 gesunken und wir müssen $upout(\varphi p)$ aufrufen (die Invariante gilt jetzt für φp!).

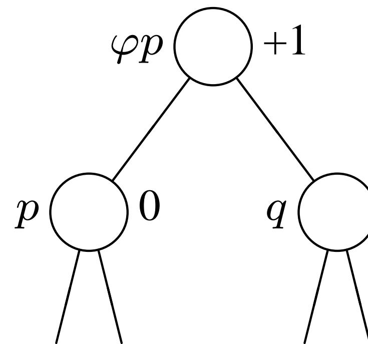
Fall 1.2: p ist linker Nachfolger von φp und $\mathrm{bal}(\varphi p) = 0$



fertig!

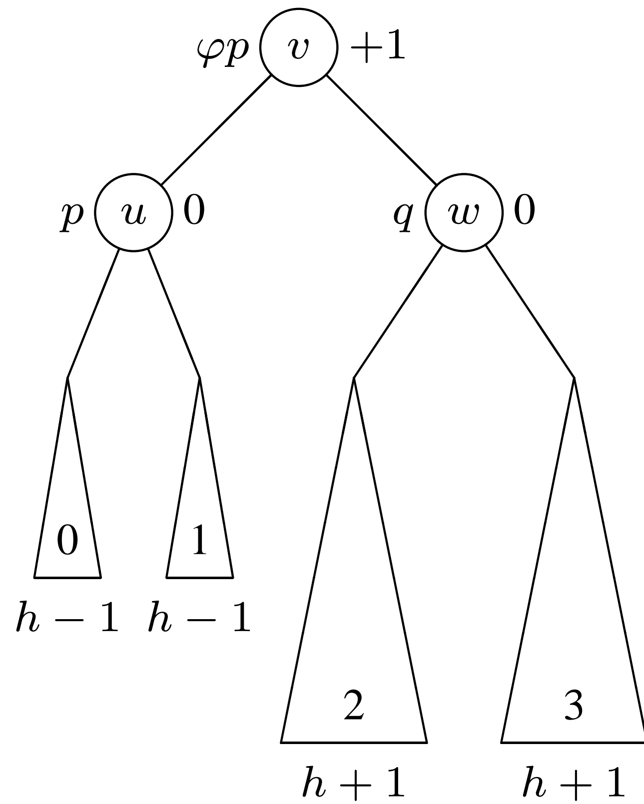
- Da sich die Höhe des Teilbaums mit Wurzel p um 1 verringert hat, ändert sich die Balance von φp zu 1.
- Anschließend sind wir fertig, weil sich die Höhe des Teilbaums mit Wurzel φp nicht geändert hat.

Fall 1.3: p ist linker Nachfolger von φp und $\mathrm{bal}(\varphi p) = +1$



- Der rechte Teilbaum von φp war also vor der Löschung bereits um 1 größer als der linke.
- Somit ist jetzt in dem Teilbaum mit Wurzel φp die AVL-Baum-Eigenschaft verletzt.
- Wir unterscheiden drei Fälle entsprechend dem Balancegrad von q.

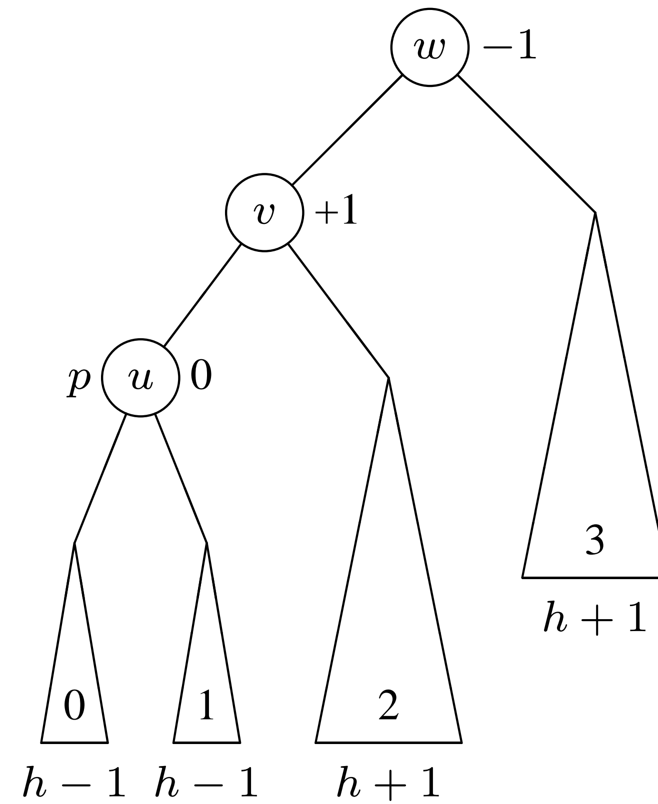
Fall 1.3.1: $\mathrm{bal}(q) = 0$



$\Longrightarrow$

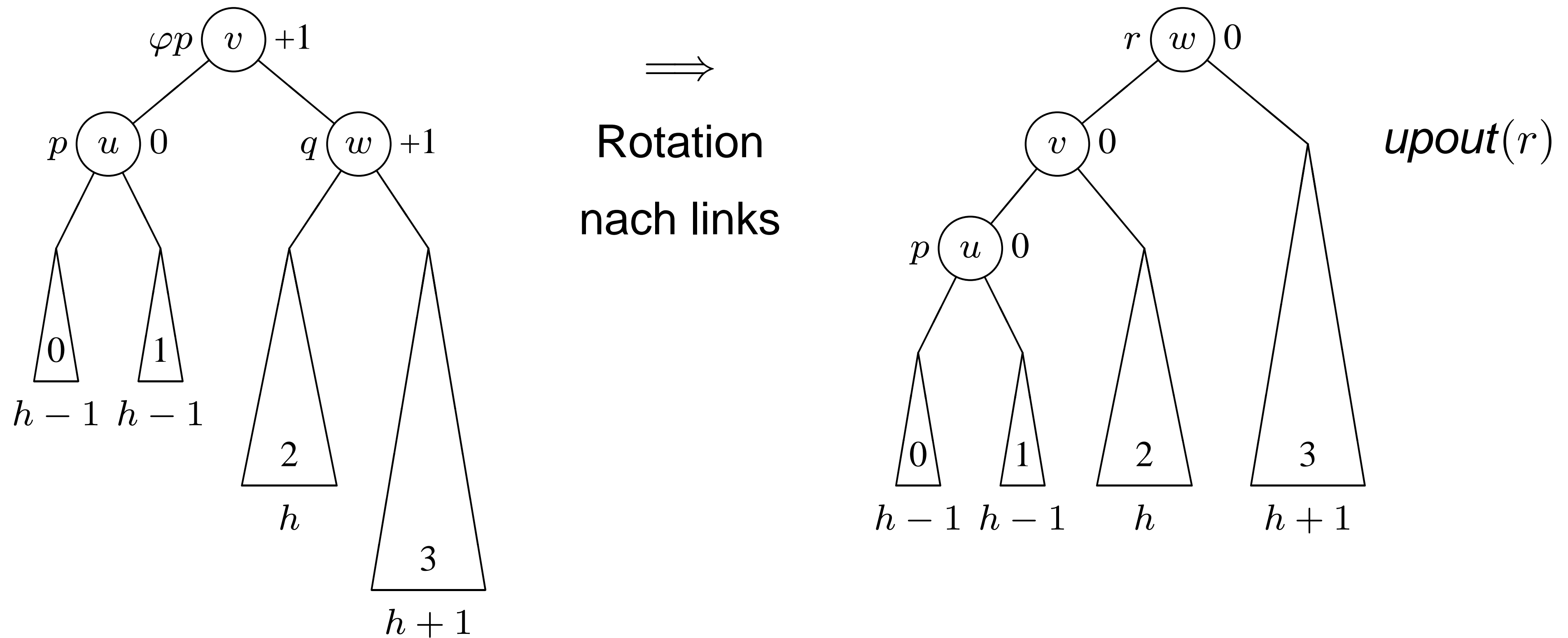
Rotation

nach links



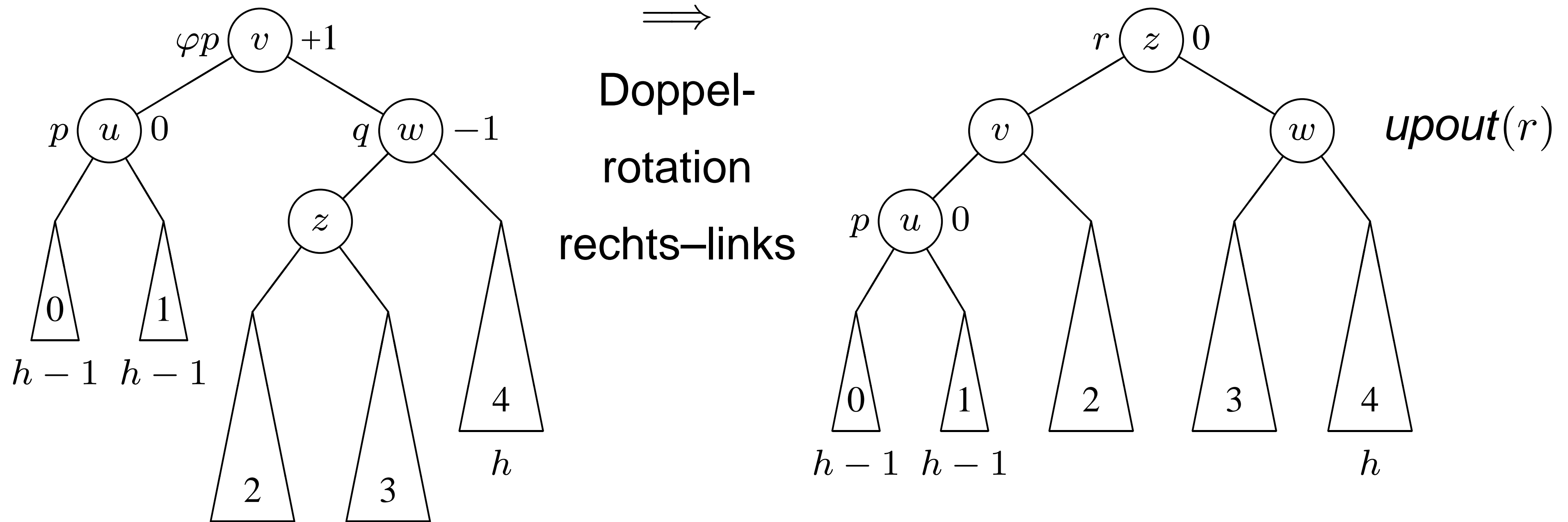
fertig!

Fall 1.3.2: $\mathrm{bal}(q) = +1$



- Erneut hat sich die Höhe des Teilbaums um 1 verringert, wobei $\mathrm{bal}(r) = 0$ (Invariante).
- Wir rufen also $upout(r)$ auf.

Fall 1.3.3: $\text{bal}(q) = -1$



- Wegen $\text{bal}(q) = -1$ muss einer der Bäume 2 oder 3 die Höhe h besitzen.
- Deswegen ist auch in diesem Fall die Höhe des gesamten Teilbaums um 1 gefallen, wobei gleichzeitig $\text{bal}(r) = 0$ gilt (Invariante).
- Wir rufen also wieder $upout(r)$ auf.

Beobachtungen

- Anders als beim Einfügen, kann es beim Löschen vorkommen, dass auch **nach einer Doppelrotation** die **Methode** $upout$ **rekursiv aufgerufen** werden muss.
- Daher **reicht i.Allg. eine einzelne Rotation oder Doppelrotation nicht aus**, um den Baum wieder auszugleichen.
- Man kann **Beispiele** konstruieren, in denen **an allen Knoten entlang des Suchpfads Rotationen oder Doppelrotationen** ausgeführt werden müssen.
- Wegen $h \leq 1.44 \ldots \log_2(n) + 1$ gilt aber, dass das **Entfernen eines Schlüssels aus einem AVL-Baum mit** n **Schlüsseln in höchstens** $O(\log n)$ **Schritten ausführbar** ist.
- AVL-Bäume sind eine ***worst-case-effiziente*** **Datenstruktur für das Suchen, Einfügen und Löschen von Schlüsseln**.

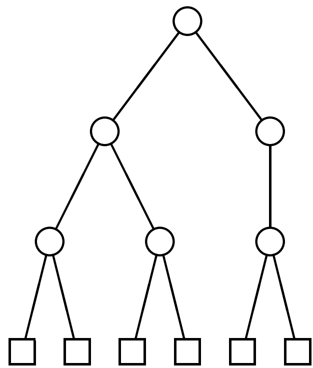
Bruder-Bäume

Idee:

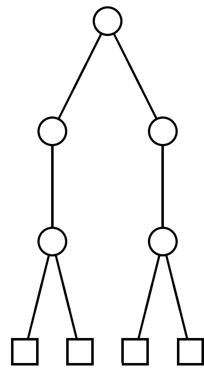
- **Innere Knoten** dürfen **auch nur einen Nachfolger haben**.
- Solche Knoten heißen **unäre Knoten**.
- Durch **Einfügen der unären Knoten** erreicht man, dass **alle Blätter dieselbe Tiefe** haben.
- **Zu viele unäre Knoten** führen jedoch zu **entarteten Bäumen mit großer Höhe und wenigen Blättern**.
- Man **verhindert das Entarten**, indem man eine Bedingung an so genannte Bruderknoten stellt.
- Zwei **Knoten heißen Brüder, wenn sie denselben Vorgänger haben**.

Definition von Bruder-Bäumen

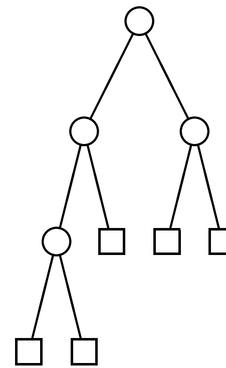
Definition: Ein binärer Baum heißt ein **Bruder-Baum**, wenn **jeder innere Knoten einen oder zwei Nachfolger** hat, **jeder unäre Knoten einen binären Bruder** hat und **alle Blätter dieselbe Tiefe** haben.



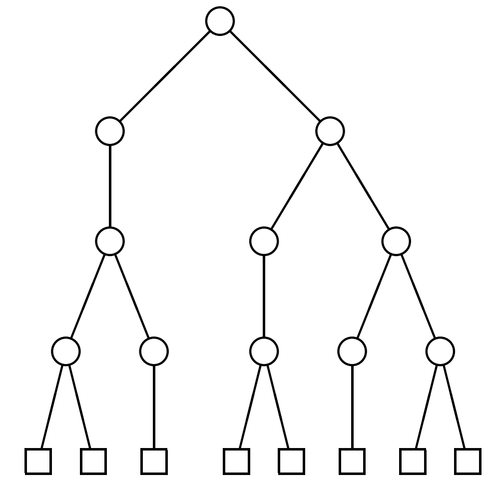
Bruder–Baum



kein Bruder-Baum



kein Bruder-Baum



Bruder-Baum

Beobachtungen

- Ist ein Knoten p der einzige Nachfolger seines Vorgängers, so ist p ein Blatt oder binär. Von zwei Nachfolgern eines binären Knotens kann höchstens einer unär sein.
- Offensichtlich ist die Anzahl der Blätter eines Bruder-Baumes stets um 1 größer als die Anzahl der binären (inneren) Knoten.
- Ebenso wie für AVL-Bäume gilt auch für Bruder-Bäume: Ein Bruder-Baum mit Höhe h hat wenigstens F_h Blätter.
- Entsprechend hat ein Bruder-Baum mit n Blättern und $(n-1)$ inneren Knoten eine Höhe $h \leq 1.44 \ldots \log_2 n$.

Bruder-Bäume als Suchbäume

Eine Möglichkeit Bruder-Bäume als Suchbäume zu verwenden sind die so genannten 1-2-Bruder-Bäume:

- Nur binäre Knoten enthalten Schlüssel.
- Die Schlüssel im linken Teilbaum eines Knotens p sind alle kleiner als der Schlüssel in p. Umgekehrt sind alle Schlüssel im rechten Teilbaum von p größer als der von p.
- Unäre Knoten enthalten keine Schlüssel.

Einschub: a-b-Bäume

Definition: Ein **a-b-Baum** ist ein Baum mit folgenden Eigenschaften:

1. **Jeder innere Knoten hat mindestens a und höchstens b Nachfolger**.
2. **Alle Blätter haben die gleiche Tiefe**.
3. **Jeder Knoten mit i Nachfolgern enthält genau $i-1$ Schlüssel**.

Bemerkungen:

1. Bruder-Bäume sind spezielle 1-2-Bäume.
2. Die später behandelten B-Bäume sind $\lceil m/2 \rceil$-m-Bäume ($m \geq 2$).

Operationen auf Bruder-Bäumen (1)

Suchen: Im Prinzip analog zu binären Suchbäumen. Stößt man auf einen unären Knoten, führt man die Suche bei dessen Nachfolger fort.

Einfügen: Beim Einfügen geht man anders vor als bei Suchbäumen:

- Da alle Blätter die gleiche Höhe haben, kann man einen neuen Knoten nicht einfach anhängen.
- Stattdessen versucht man, unäre in binäre Knoten umzuwandeln.
- Gelingt dies nicht, geht man stufenweise nach oben und versucht dort, durch geeignete Transformationen den Knoten einzufügen.
- Im schlimmsten Fall kommt man bis zur Wurzel und muss dann einen neuen Knoten zur Wurzel machen.

Operationen auf Bruder-Bäumen (2)

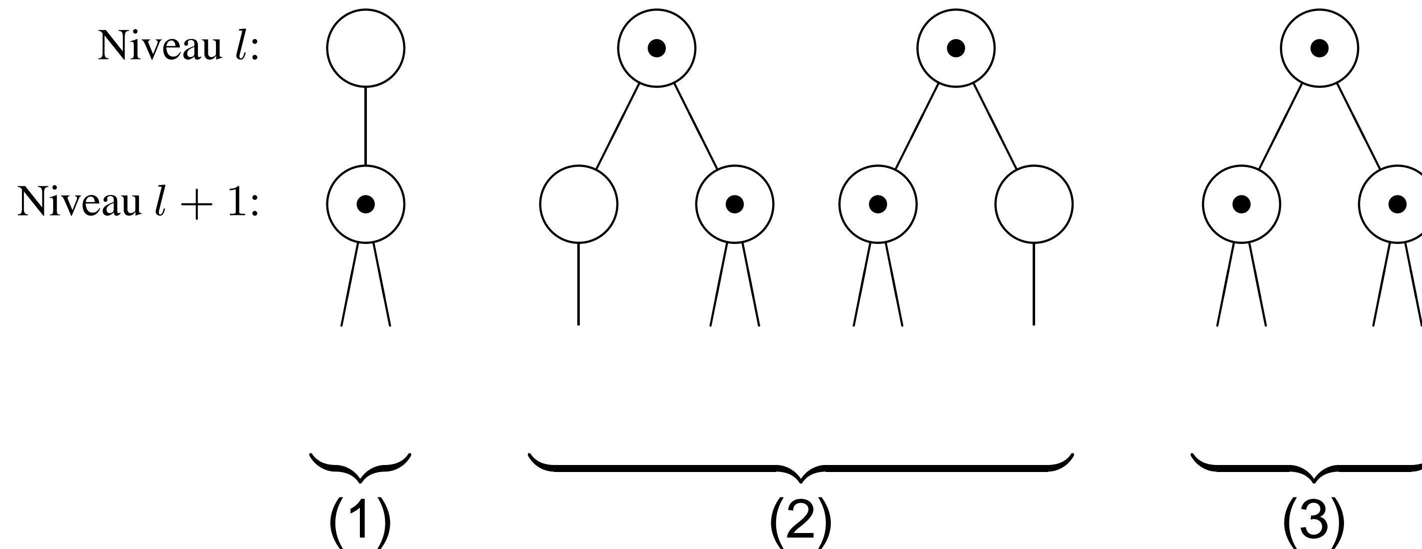
- Bruder-Bäume wachsen demnach an der Wurzel und nicht an den Blättern.

Löschen: Beim Löschen geht man ähnlich vor wie bei binären Suchbäumen:

- Gegebenenfalls muss man das Löschen auf das Löschen des symmetrischen Nachfolgers reduzieren.
- Ausgehend von dem zu löschenden Knoten geht man dann rekursiv entlang des Suchpfads (ähnlich wie bei AVL-Bäumen) im Baum nach oben, um die Bruder-Baum-Eigenschaften wiederherzustellen.
- Dabei muss man schlimmstenfalls bis zur Wurzel laufen.
- Damit kann man auch bei Bruderbäumen die Operationen Suchen, Einfügen und Löschen in $O(\log n)$ durchführen.

Analytische Betrachtungen

- 1-2-Bruder-Bäume enthalten i.Allg. unäre Knoten, die keine Schlüssel speichern.
- Wieviele können das sein?
- Für aufeinanderfolgende Levels sind lediglich folgende Konfigurationen möglich:



Wieviele Schlüssel sind in einem 1-2-Bruder-Baum?

- **Für jeden unären Knoten auf Niveau l muss es einen binären Bruder auf demselben Niveau geben.**
- **Zulässige Kombinationen der Konfigurationen** sind demnach:

Konfiguration	U
(2)	$\frac{2}{3}$
(3)	$\frac{3}{3}$
(1) und eine Konfig. aus (2)	$\frac{3}{5}$
(1) und (3)	$\frac{4}{5}$

mit

$$U = \frac{\text{Anzahl von binäre Knoten auf Niveau } l \text{ und } l+1}{\text{Anzahl Knoten insgesamt auf Niveau } l \text{ und } l+1}$$

- Folglich sind **wenigstens $\frac{3}{5}$ aller Knoten in einem 1-2-Bruder-Baum binär** und enthalten Schlüssel.

Konsequenzen

- Offensichtlich werden beim Einfügen von n Schlüsseln in einen anfangs leeren Teilbaum höchstens $\frac{5}{3} \cdot n$ Knoten erzeugt.
- Man kann nachweisen, dass bei der Einfügung eines neuen Knoten beim rekursiven Entlanglaufen des Suchpfads stets gestoppt wird, wenn kein neuer Knoten erzeugt wird.
- Also ist die **durchschnittliche Anzahl der notwendigen Transformationen bei einer Einfügung in einen 1-2-Bruder-Baum konstant**.
- Für AVL-Bäume ist die Herleitung einer entsprechenden Aussage wesentlich schwieriger.
- Zwar stoppt die Methode $upin$ nach einer Rotation oder Doppelrotation, allerdings ist nicht klar, wie weit man nach oben laufen muss.

B-Bäume

Motivation:

- Bisher waren wir davon ausgegangen, dass die durch Bäume strukturierten Daten im Hauptspeicher Platz finden.
- In der Praxis (z.B. in Datenbanken) ist man jedoch häufig gezwungen, die **Daten und auch die Schlüssel auf externen Medien (heute in der Regel Festplatten) abzulegen**.
- **Ziel von B-Bäumen ist, die Anzahl der Zugriffe auf das externe Medium zu reduzieren**.

Definition der B-Bäume

Definition: Ein Baum heißt B-Baum der Ordnung m, wenn die folgenden Eigenschaften erfüllt sind.

1. **Jeder Knoten mit Ausnahme der Wurzel enthält mindestens $\lceil m/2 \rceil - 1$ Daten. Jeder Knoten enthält höchstens $m - 1$ Daten. Die Daten sind sortiert.**

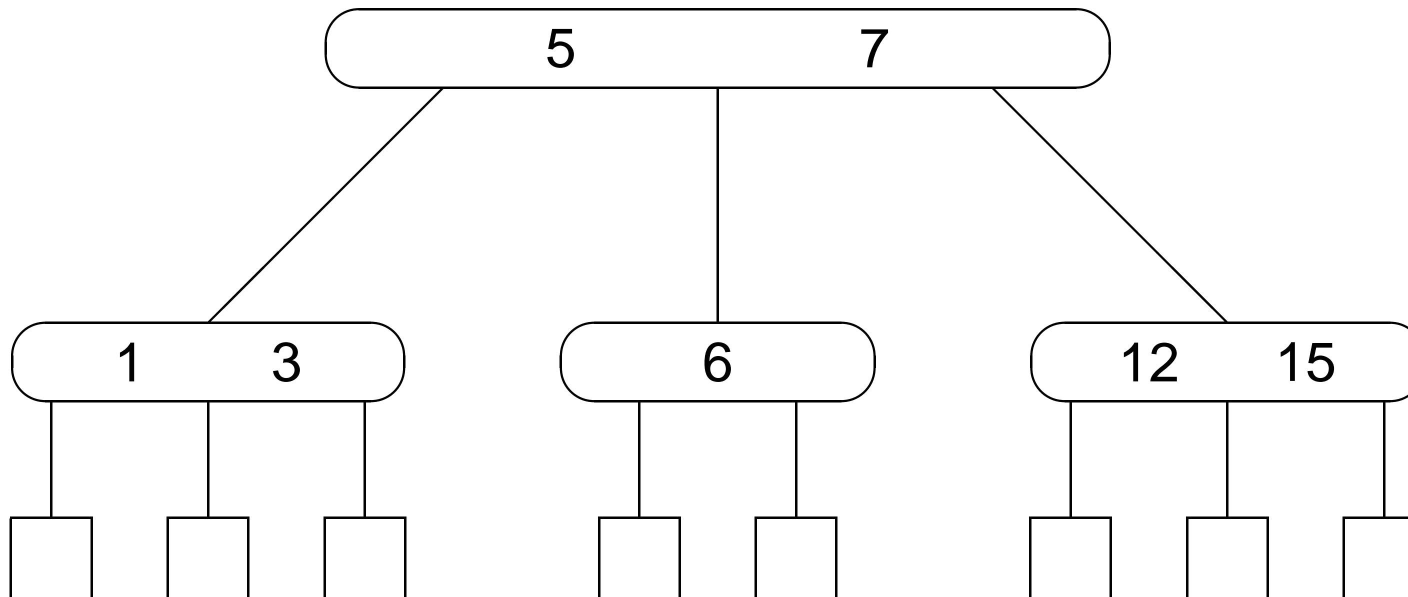
2. **Knoten mit k Daten $x_1, \ldots, x_k$ haben $k+1$ Referenzen auf Teilbäume** mit Schlüsseln aus den Mengen

$$\{-\infty \ldots, x_1{-}1\}, \{x_1{+}1, \ldots, x_2{-}1\}, \ldots, \{x_{k-1}{+}1, \ldots x_k{-}1\}, \{x_k{+}1, \ldots \infty\}.$$

3. Die Referenzen, die einen Knoten verlassen, sind entweder **alle `null`-Referenzen** oder **alle Referenzen auf Knoten**.

4. **Alle Blätter haben gleiche Tiefe**.

Ein Beispiel



Hinweis: **Im Kontext von B-Bäumen sind `null`-Referenzen durch □ dargestellt.**

Beobachtung: Dieser Baum hat 7 Schlüssel und 8 `null`-Referenzen.

Anzahl der `null`-Referenzen

Behauptung: Die Anzahl der `null-Referenzen` ist stets um eins größer als die Anzahl der Schlüssel in einem B-Baum.

Beweis:

Induktionsverankerung: Der leere B-Baum enthält eine `null`-Referenz aber keinen Knoten.

Induktionsvoraussetzung: Die Aussage gelte für alle B-Bäume mit höchstens $n-1$ Schlüsseln.

Induktionsschluss: Sei p die Wurzel eines B-Baums mit n Schlüsseln. Angenommen p hat $k-1$ Schlüssel und k Teilbäume mit $l_1, \ldots, l_k$ Schlüsseln. Nach Induktionsvoraussetzung ist die Summe der `null`-Referenzen in den Teilbäumen genau

$$\sum_{i=1}^{k} (l_i + 1) = 1 + (k-1) + \sum_{i=1}^{k} l_i = 1 + n$$

Höhe eines B-Baums

- Um die **Anzahl der in einem B-Baum mit Höhe h gespeicherten Schlüssel** abzuschätzen, genügt es also, die **Anzahl der `null`-Referenzen zu bestimmen**.
- Offensichtlich ist die **Höhe maximal**, wenn **in der Wurzel lediglich ein und in jedem weiteren Knoten genau $\lceil \frac{m}{2} \rceil - 1$ Schlüssel** enthalten sind.
- Die **minimale Anzahl von `null`-Referenzen** ist somit

$$N_{\min} = 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1}.$$

- Ist ein **B-Baum mit N Schlüsseln und Höhe h** gegeben, so **hat** er $(N+1)$ **`null`-Referenzen** und es muss gelten:

$$N_{\min} = 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1} \leq (N+1) \qquad \Longrightarrow \qquad h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right)$$

Konsequenzen

- **B-Bäume** haben demnach auch die für balancierte Bäume typische Eigenschaft, dass die **Höhe logarithmisch beschränkt** ist **in der Anzahl der gespeicherten Schlüssel**.
- Für $m = 199$ haben B-Bäume mit $1.999.999$ Schlüsseln höchstens die Höhe 4.

Suchen in einem B-Baum

Um einen Schlüssel in einem B-Baum zu suchen, verfahren wir wie folgt:

1. Ausgehend von der Wurzel prüfen wir, ob der gesuchte Schlüssel sich in dem gerade betrachteten Knoten befindet.
2. Ist das nicht der Fall, bestimmen wir den kleinsten Schlüssel der größer als der gesuchte ist.
   - Existiert dieser Schlüssel, gehen wir zum Nachfolger-Knoten links von diesem Schlüssel über.
   - Existiert der Schlüssel nicht, gehen wir zum letzten Nachfolger-Knoten über.
3. Wenn wir bei einer `null`-Referenz landen, ist die Suche erfolglos.

Suche innerhalb eines Knotens

Hier sind prinzipiell verschiedene Verfahren denkbar:

- lineare Suche
- binäre Suche
- ...

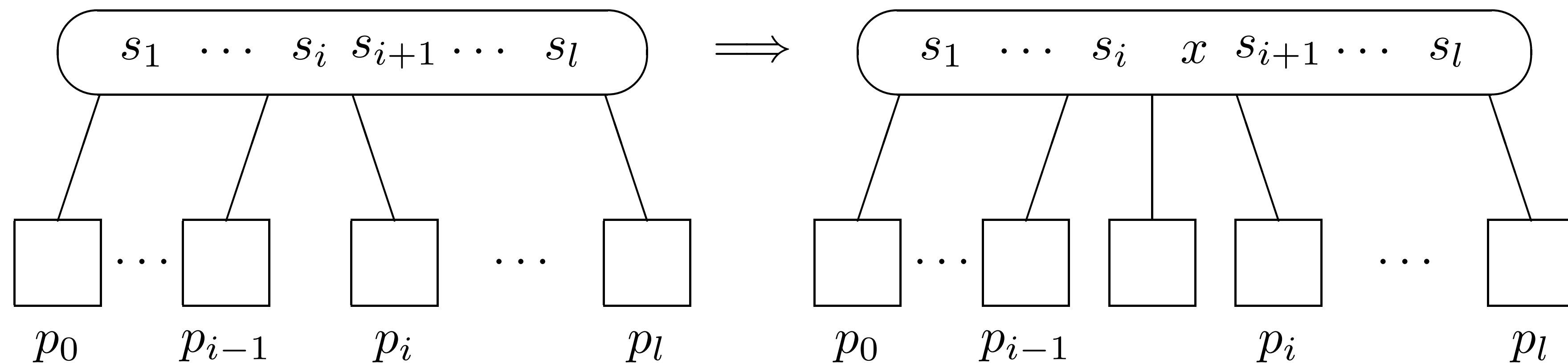
Allerdings beeinträchtigt die Suche innerhalb eines Knotens die Rechenzeit eher wenig, weil diese hauptsächlich durch die Anzahl der Zugriffe auf den Hintergrundspeicher (Festplatte) beeinflusst wird.

Einfügen eines Schlüssels in den B-Baum

- Zunächst suchen wir die Stelle, an der der Schlüssel x im Baum vorkommen sollte.
- Ist die Suche erfolglos, endet sie in einem Blatt p an der erwarteten Position von x.
- Seien $s_1, \ldots, s_l$ die in p gespeicherten Schlüssel.
- Wir unterscheiden 2 Fälle:
  1. Das Blatt ist noch nicht voll, d.h. $l < m - 1$.
  2. Das Blatt ist voll, d.h. $l = m - 1$.

Fall 1: Das Blatt ist noch nicht voll

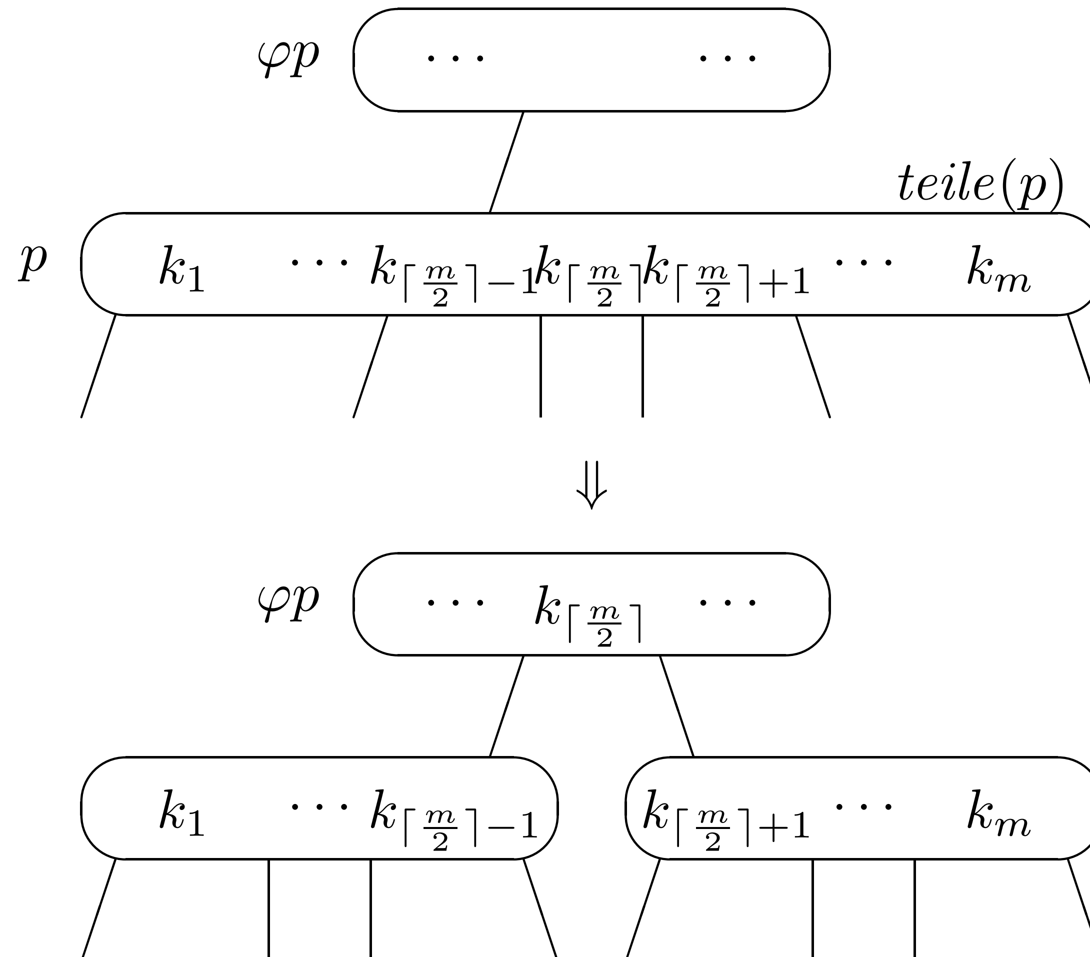
- Wir fügen x in dem Blatt p zwischen dem Schlüsselpaar (s_i, s_{i+1}) ein, für das $s_i < x < s_{i+1}$. Ist x kleiner als s_0 oder größer als s_l, so fügen wir x am Anfang oder am Ende ein.
- Dann setzen wir den Wert der ebenfalls einzufügenden Nachfolgereferenz auf `null`.



Fall 2: Das Blatt ist voll und enthält genau $m-1$ Schlüssel

- Wir ordnen x in die Folge der Schlüssel entsprechend der Ordnung ein.
- Seien $k_1, \ldots, k_m$ die Schlüssel in p in aufsteigender Reihenfolge in p.
- Wir erzeugen nun ein neues Blatt q und verteilen die Schlüssel so auf p und q, dass p die Schlüssel $k_1, \ldots, k_{\lceil m/2 \rceil - 1}$ und q die Schlüssel $k_{\lceil m/2 \rceil + 1}, \ldots, k_m$ enthält.
- Der Schlüssel $k_{\lceil m/2 \rceil}$ wandert in den Vorgänger φp von p.
- Wenn j die Position von $k_{\lceil m/2 \rceil}$ in φp ist, wird p_{j-i} eine Referenz auf p und p_j eine Referenz auf q.
- Dieses Verfahren wird rekursiv fortgesetzt, bis wir an der Wurzel angelangt sind und ggf. eine neue Wurzel mit einem Schlüssel erzeugt haben.

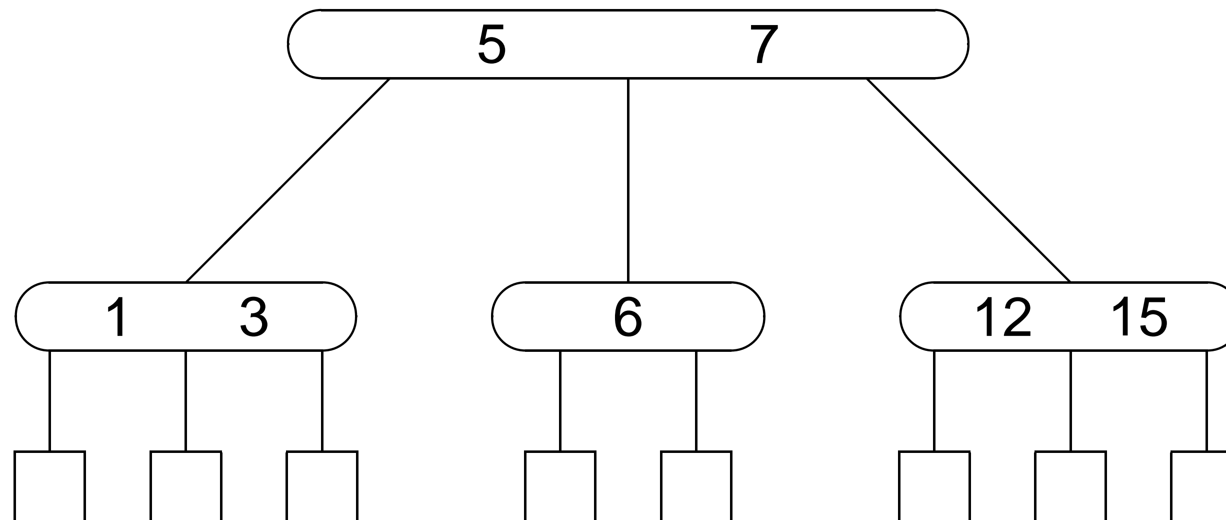
Die Einfügeoperation



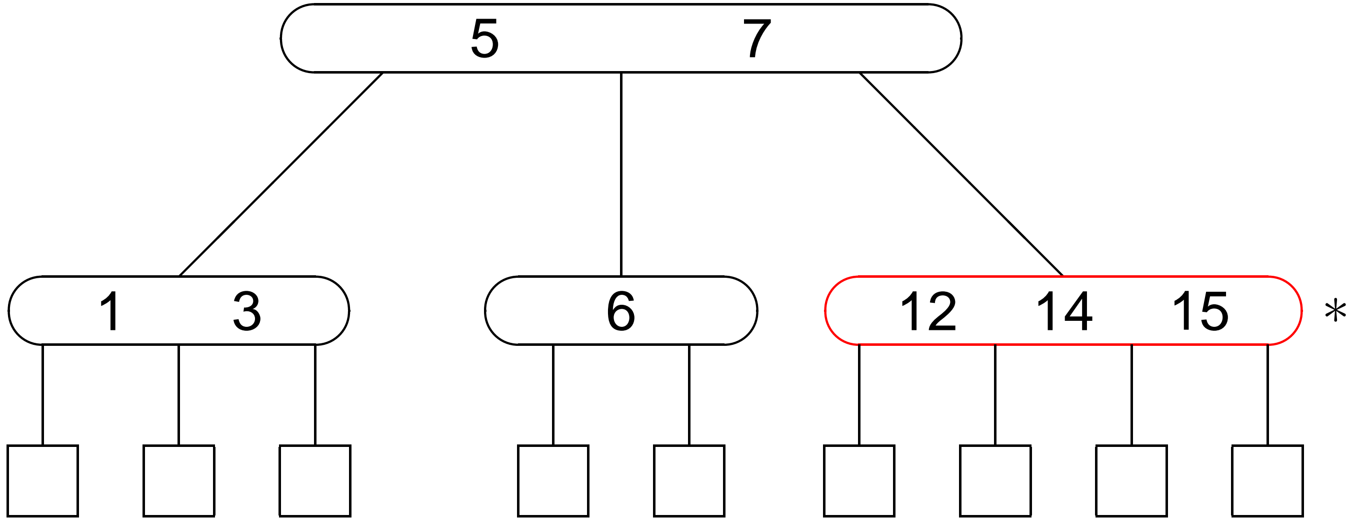
und $teile(\varphi p)$, falls φp (nach Einfügen von $k_{\lceil \frac{m}{2} \rceil}$) m Schlüssel hat

Ein Beispiel (2-3-Baum)

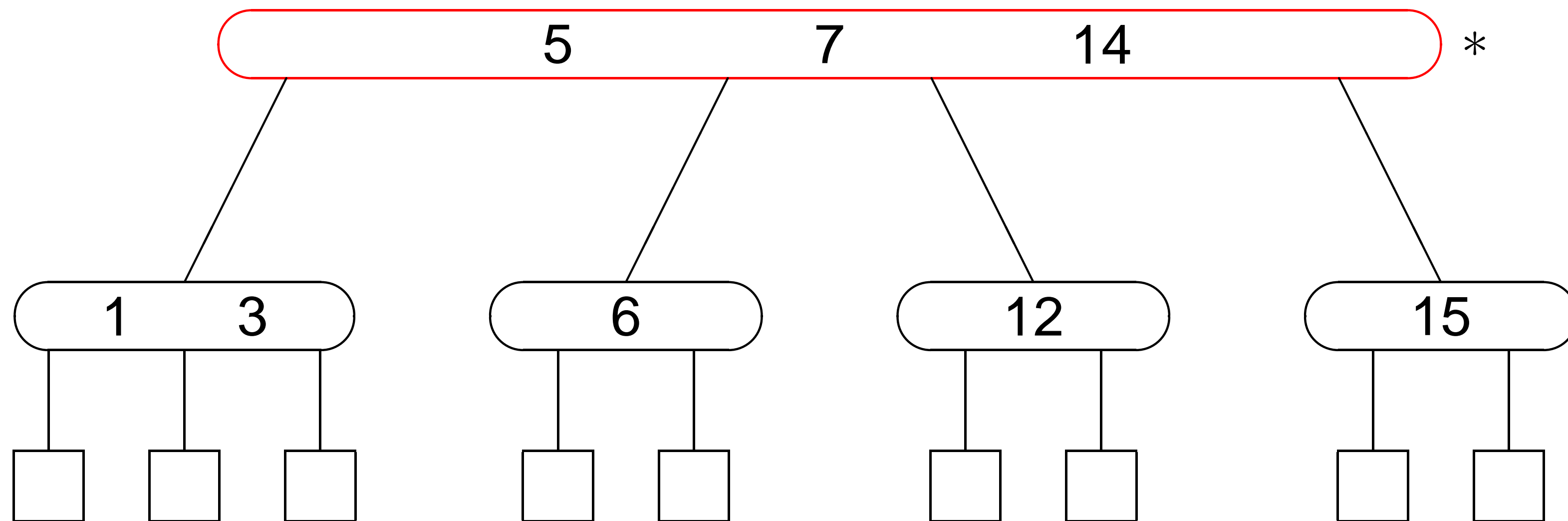
Ausgangssituation:



Einfügen von 14

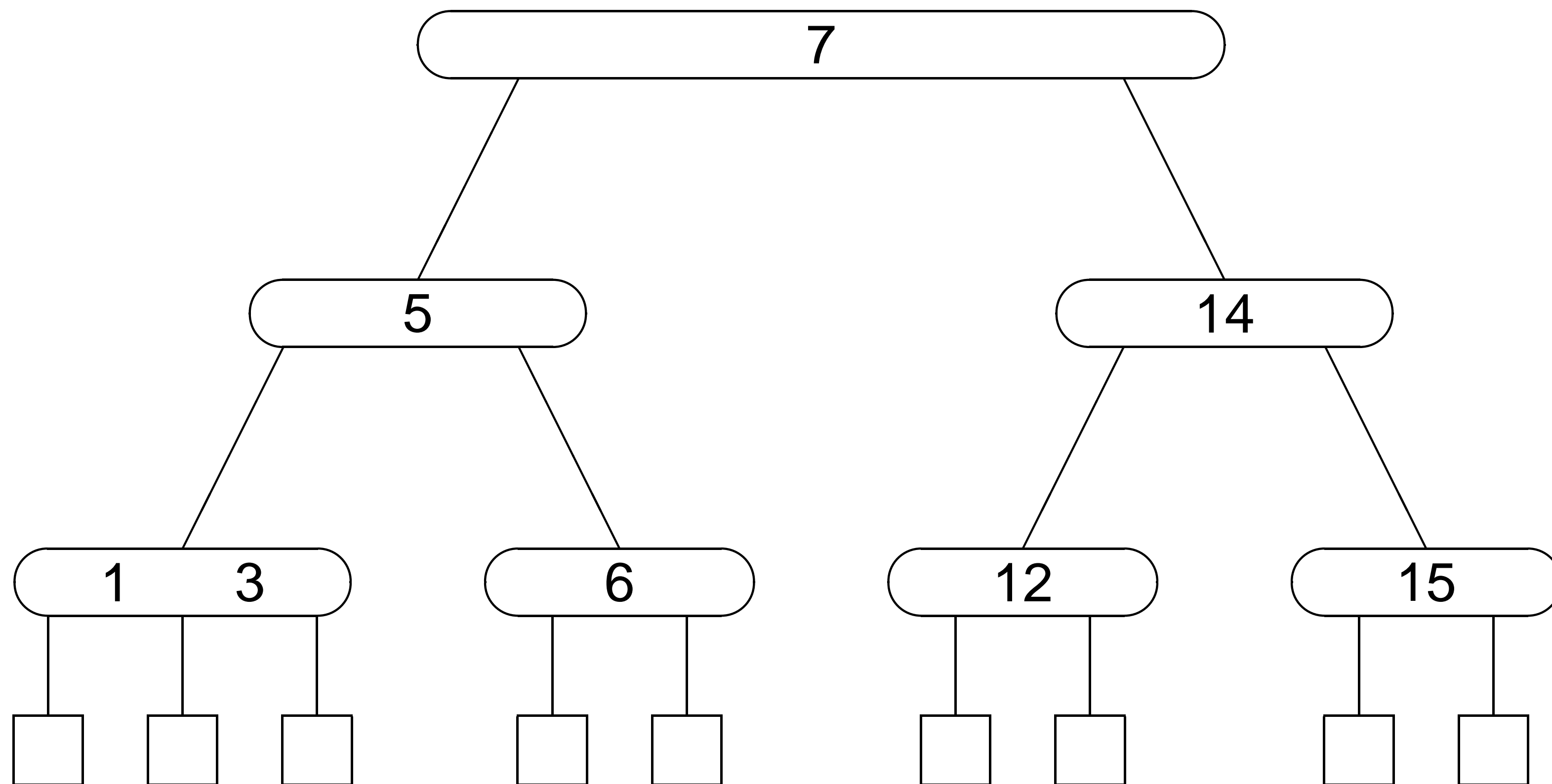


Aufteilen des Blattes



Aufteilen der Wurzel

Ergebnis:



Löschen von Schlüsseln aus einem B-Baum

- Das Löschen von Schlüsseln ist leider wieder komplizierter als das Einfügen.
- Beim Entfernen **unterscheiden** wir, **ob** der zu löschende Schlüssel x **aus einem Blatt oder aus einem inneren Knoten gelöscht werden soll**.
- Offensichtlich besteht das Problem, dass beim Löschen eines Schlüssels aus einem Knoten ein **Underflow** auftreten kann, d.h. dass **in dem Knoten zu wenig Schlüssel vorhanden** sind.

Löschen aus einem Blatt

- Ein Blatt hat die Struktur

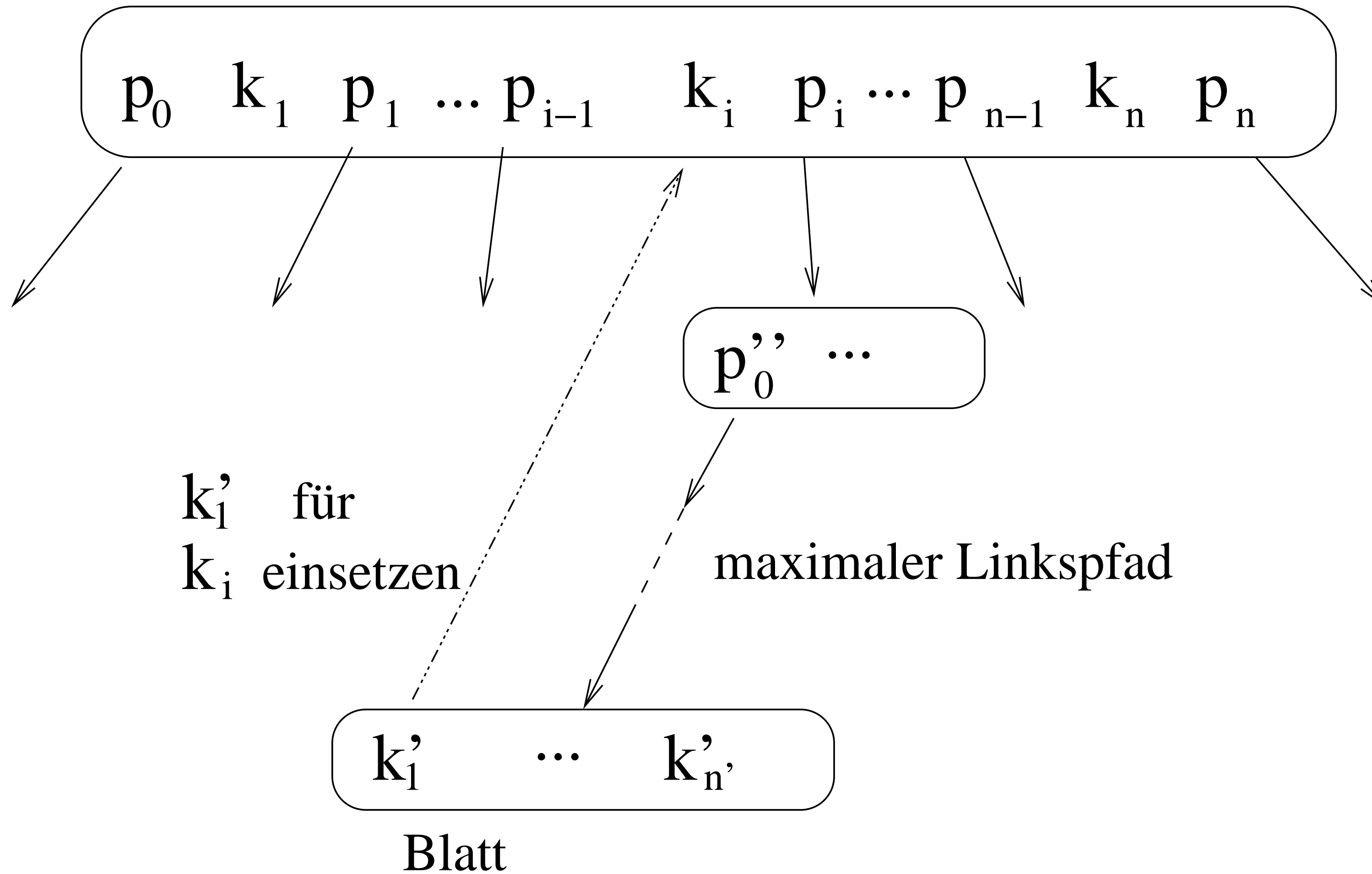
$$\texttt{null}, k_1, \ldots, \texttt{null}, k_i, \texttt{null}, \ldots, k_l, \texttt{null}$$

- Wenn nun das Element mit Schlüssel k_i entfernt werden soll, so streicht man einfach k_i und die darauf folgende `null`-Referenz.
- Ein **Underflow** tritt auf, falls $l = \lceil m/2 \rceil - 1$ war.

Entfernen aus einem inneren Knoten

- Im Unterschied zum Löschen aus einem Blatt haben **alle Referenzen einen Wert ungleich `null`**.
- Insbesondere **existiert ein Nachfolger p_i rechts von dem zu löschenden Schlüssel k_i**.
- Demnach ist der **symmetrische** (oder inorder) **Nachfolger** von k_i im durch p_i referenzierten Teilbaum zu finden **(analog zu binären Suchbäumen)**.
- **Wegen der Ausgeglichenheit** muss der **symmetrische Nachfolger** von k_i **in einem Blatt** sein.
- Wir **ersetzen nun k_i durch seinen symmetrischen Nachfolger und löschen k_i aus dem Blatt (erneut Fall 1)**.

Auffinden des symmetrischen Nachfolgers



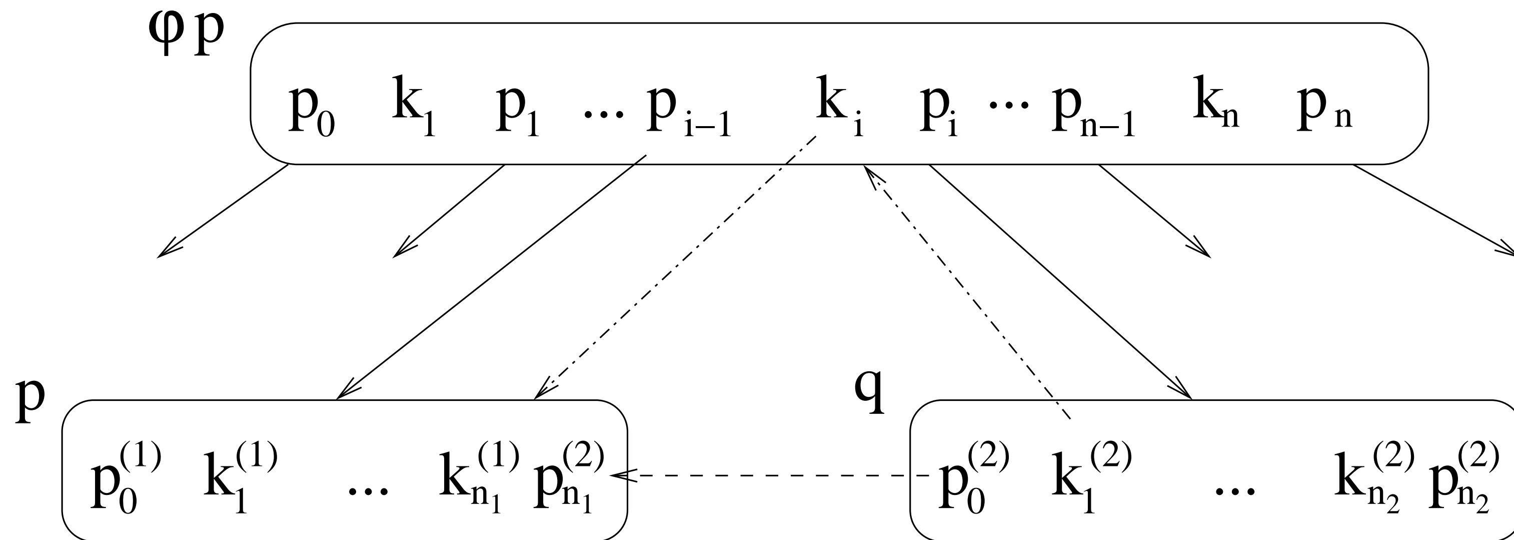
Behandlung des Underflows

- Bei der Behandlung des Underflows sind wieder verschiedene Fälle möglich.
- Wir benötigen zwei Operationen:
  1. das Ausgleichen (zwischen zwei Bruder-Knoten)
  2. das Verschmelzen (von zwei Bruder-Knoten)

Ausgleich zwischen Bruder-Knoten

- Zwischen zwei Brüdern p und q wird ausgeglichen, wenn bei p ein Underflow eintritt und q mehr als $\lceil m/2 \rceil - 1$ Schlüssel enthält.
- Idee der Austauschoperation ist, aus q einen Schlüssel zu entfernen und diesen in den Vorgänger φp (von p und q) zu einzufügen. Aus φp entnehmen wir dann einen Schlüssel, den wir in p einfügen.
- Dies erfolgt so, dass die B-Baum-Eigenschaft erhalten bleibt.
- Da das Problem symmetrisch ist, nehmen wir an, dass p linker Bruder von q ist.

Die Ausgleichsoperation



1. k_i wird größter Schlüssel in Knoten p.

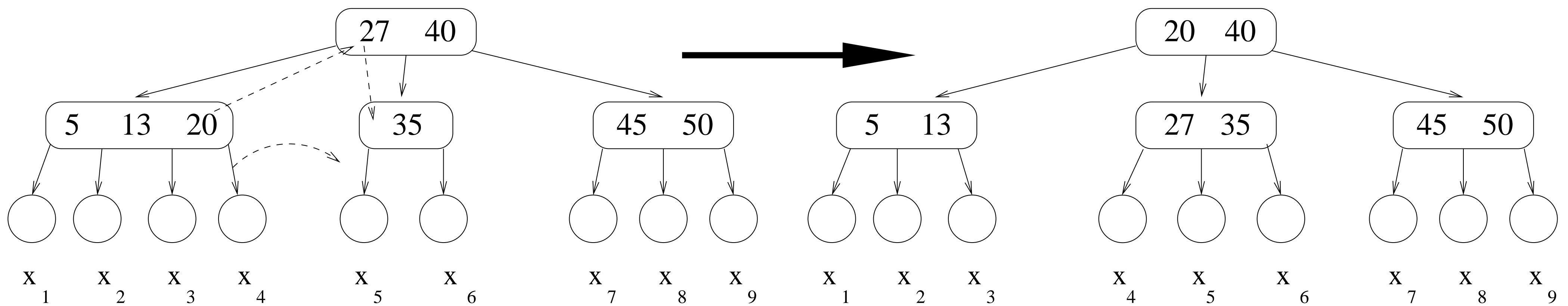
2. $p_0^{(2)}$ wird letzte Referenz in Knoten p.

3. k_i wird in φp durch $k_1^{(2)}$ aus q ersetzt.

Hinweis: Der Baum hat anschließend wieder die B-Baum-Eigenschaft.

Ein Beispiel

3-6-Baum mit Underflow im zweiten Nachfolger der Wurzel:



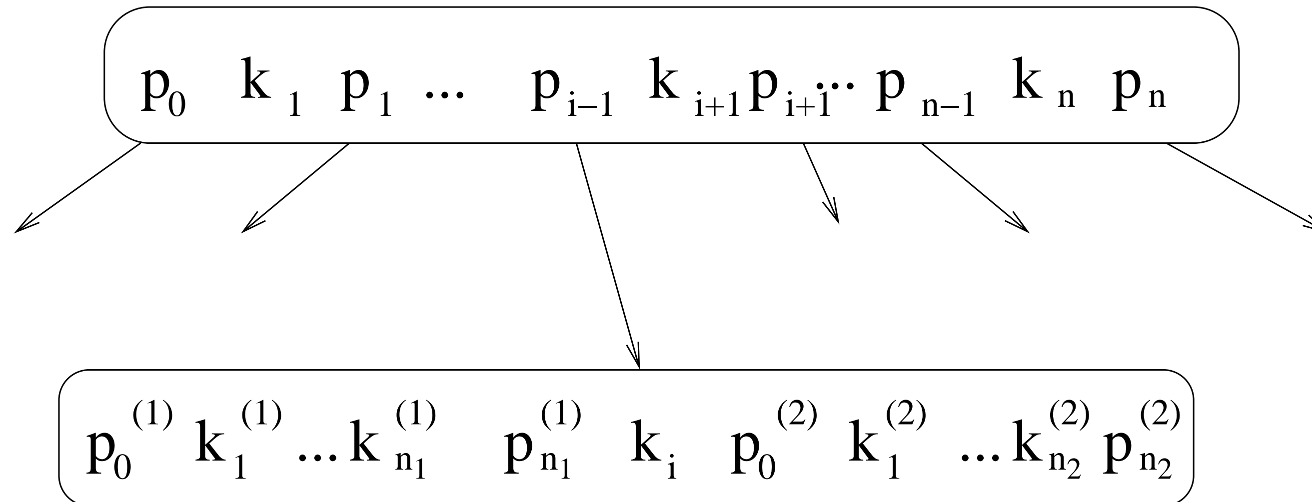
Das Verschmelzen von Bruder-Knoten

- Ein Knoten p wird mit einem seiner Brüder verschmolzen, wenn bei p ein Underflow eintritt und kein direkter Bruder von p mehr als $\lceil m/2 \rceil - 1$ Schlüssel enthält.
- Idee der Verschmelzungsoperation ist, aus p und seinem Bruder q einen neuen Knoten erzeugen. Zwischen die Schlüssel von p und q kommt jedoch der entsprechende Schlüssel aus dem Vorgängerknoten φp.
- Danach hat der neue Knoten

  $$\lceil m/2 \rceil - 2 + \lceil m/2 \rceil - 1 + 1 = 2 \cdot \lceil m/2 \rceil - 2 \leq m - 1$$

  Knoten.
- Allerdings haben wir aus φp einen Knoten entnommen, so dass dort ggf. ein **Underflow** eintreten kann.
- Da das Problem symmetrisch ist, nehmen wir erneut an, dass p linker Bruder von q ist.

Die Verschmelzungsoperation



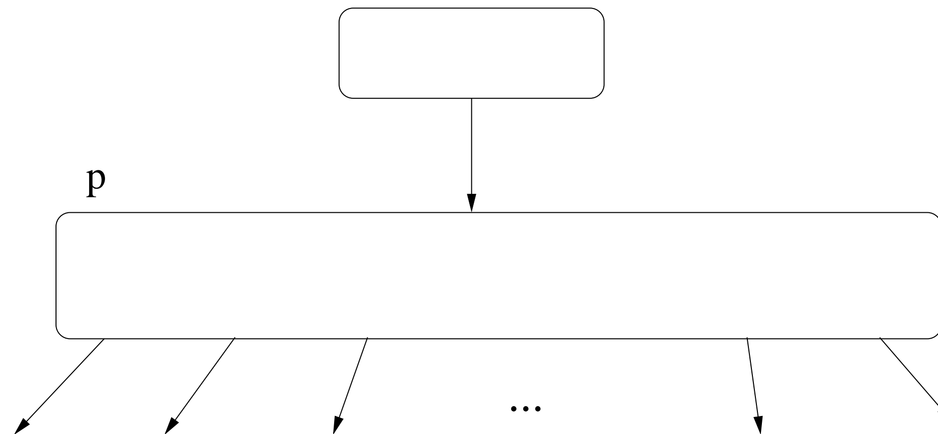
mit $n_1 = \lceil m/2 \rceil - 2$ und $n_2 = \lceil m/2 \rceil - 1$

Das Vorgehen:

- k_i aus φp wandert in den neuen Knoten.
- Der neue Knoten wird Nachfolger p_{i-1} in φp.
- p_i wird aus φp gelöscht.

Spezialfall Wurzel

- Die Verschmelzung wird rekursiv nach oben fortgesetzt, bis wir ggf. bei der Wurzel ankommen.
- Wenn wir dann aus der Wurzel den letzten Schlüssel entfernen, haben wir folgende Situation:



- Da die Wurzel keinen Schlüssel mehr enthält, können wir sie einfach löschen und ihren einzigen Nachfolger als Wurzel verwenden.
- Die Höhe des B-Baums ist dann um 1 gesunken.

Ein Beispiel für eine Verschmelzung

3-6-Baum mit Underflow im zweiten Nachfolger der Wurzel: