# Algorithmen und Datenstrukturen
# (Th. Ottmann und P. Widmayer)

## Folien: Balancierte Suchbäume

**Autor: Stefan Edelkamp**

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg

# 1 AVL-Bäume

Namensgebung: Schöpfer Adelson-Velskii und Landis (1962)

Definition: Ein binärer Suchbaum heißt AVL-Baum, wenn für jeden Knoten gilt, daß sich die Tiefe des rechten Teilbaumes $d(T_r)$ und die Tiefe des linken Teilbaumes $d(T_l)$ um maximal 1 unterscheiden.

Balancegrad: $b(v) = d(T_l) - d(T_r) \in \{-1, 0, 1\}$

Satz: Die Tiefe eines AVL-Baumes mit $n$ Daten beträgt mindestens $\lceil \log(n+1) \rceil - 1$ und höchstens $\log(\sqrt{5}(n+2))/\log((\sqrt{5}+1)/2) \approx 1.44 \log n$.

Beweis: Die untere Schranke gilt sogar für alle binären Suchbäume.
z.Z. minimale Anzahl der Knoten in AVL-Baum $A(d)$ der Tiefe $d$ ist $F(d+2)-1$, denn

$$F(d+2) - 1 = A(d) \leq n \Rightarrow d + 3 \leq \log(\sqrt{5}(n+2))/\log((\sqrt{5}+1)/2) \approx 1.44 \log n$$

Nun ist gemäß der Balancebedingung

$$A(d) = 1 + A(d-1) + A(d-2) = 1 + F(d+1) - 1 + F(d) - 1 = F(d+2) - 1$$

# Einfügen

O.b.d.A. Suchpfad rückwärts vom rechten Sohn. Die Tiefe des rechten Teilbaumes sei gewachsen.

Fall 1. Vorher $b(v) = 1$. Setze $b(v) = 0$.
Fall 2. Vorher $b(v) = 0$. Setze $b(v) = -1$.
(Rebalancierung am Vater fortsetzen.)
Fall 3. Vorher $b(v) = -1$ (aktuell gilt $b(v) = -2$). Sei $x$ rechter Sohn von $v$ und $w$ der linke Sohn von $x$.
Fall 3a. Rechter Teilbaum von $x$ gewachsen:
Linksrotation an $v$. Setze $b(v) = 0, b(x) = 0$
Fall 3b. Rechter Teilbaum von $w$ gewachsen:
Rechtsrotation an $x$, dann Linksrotation an $v$. Setze
$b(w) = 0, b(v) = 1, b(x) = 0$
Fall 3c. Linke Teilbaum von $w$ gewachsen:
Rechtsrotation an $x$, dann Linksrotation an $v$. Setze
$b(w) = 0, b(v) = 0, b(x) = -1$
Fall 3d. $w$ hat keinen Sohn. Rechtsrotation an $x$, dann Linksrotation an $v$. Setze
$b(w) = 0, b(v) = 0, b(x) = -1$

Rebalancierung nach einer Rotation oder Doppelrotation abgeschlossen.

# Entfernen

O.b.d.A. Datum in linken Teilbaum von $v$ gelöscht.

Fall 1. Vorher $b(v) = 1$. Setze $b(v) = 0$.
(Rebalancierung am Vater fortsetzen.)
Fall 2. Vorher $b(v) = 0$. Setze $b(v) = -1$.
Fall 3. Vorher $b(v) = -1$ (aktuell gilt $b(v) = -2$). Sei $x$ rechter Sohn von $v$ und $w$ der linke Sohn von $x$.
Fall 3a. Beide Teilbäum von $x$ haben gleiche Tiefe:
Linksrotation an $v$. Setze $b(v) = -1, b(x) = 1$
Fall 3b. Rechte Teilbaum von $x$ um 1 tiefer als der linke Teilbaum: Linksrotation an $v$. Setze $b(v) = 0, b(x) = 0$.
(Rebalancierung am Vater fortsetzen.)
Fall 3c. $b(x) = 1, b(w) = 1$: Rechtsrotation an $x$, dann Linksrotation an $v$. Setze
$b(w) = 0, b(v) = 0, b(x) = -1$. (Rebalancierung am Vater fortsetzen.)
Fall 3d. $b(x) = 1, b(w) = 1$: Rechtsrotation an $x$, dann Linksrotation an $v$. Setze
$b(w) = 0, b(v) = 0, b(x) = 0$. (Rebalancierung am Vater fortsetzen.)
Fall 3e. $b(x) = 1, b(w) = -1$. Rechtsrotation.
(Rebalancierung am Vater fortsetzen.) an $x$, dann Linksrotation an $v$. Setze
$b(w) = 0, b(v) = 1, b(x) = 0$. (Rebalancierung am Vater fortsetzen.)

# Implementation

## Knotenklasse:

```
class AVLNode {
   int content;           // Inhalt, hier integer
   byte balance;          // fuer Werte -2,-1,0,1,+2
   AVLNode left;          // linker  Nachfolger
   AVLNode right;         // rechter Nachfolger
   AVLNode (int c){       // Konstruktor fuer Knoten
      content = c;        //    uebergebener Inhalt
      balance = 0;        //    Balance ausgeglichen
      left = right = null; }}// erst keine Nachfolger
```

## Baumklasse:

```
class AVLTree {
   AVLNode root;                  // Die Wurzel des Baumes
   boolean grown, shrunk, found; // Hilfsvariablen
   AVLTree () {
      root = null;
      grown = shrunk = found = false;
   }
   boolean search (int c) ...
   boolean insert (int c) {
      found = false; grown = true;
      root = insert (root, c);
      return !found;
   }
   AVLNode insert (AVLNode n, int c) ...
   void rotateRight (AVLNode n) ...
   void rotateLeft (AVLNode n) ...
   boolean delete (int c) {
      found = shrunk = true; root = delete(root, c);
      return found;
   }
   AVLNode delete (AVLNode n, int c) ... }
```

# Rotationen

## Einfache Rotation nach rechts:

```
void rotateRight (AVLNode n) {
   AVLNode m = n.left;
   int cc = n.content;
   n.content = m.content;
   m.content = cc;
   n.left = m.left;
   m.left = m.right;
   m.right = n.right;
   n.right = m;
   int bm = 1+Math.max(-m.balance,0)+n.balance;
   int bn = 1+m.balance + Math.max(0,bm);
   n.balance = (byte)bn;
   m.balance = (byte)bm;
}
```

## Einfache Rotation nach links:

```
void rotateLeft (AVLNode n) {
   AVLNode m = n.right;
   int cc = n.content;
   n.content = m.content;
   m.content = cc;
   n.right = m.right;
   m.right = m.left;
   m.left = n.left;
   n.left = m;
   int bm = -(1+Math.max(+m.balance,0)-n.balance);
   int bn = -(1-m.balance + Math.max(0,-bm));
   n.balance = (byte)bn;
   m.balance = (byte)bm;
}
```

# Einfügen

## Füge c in Teilbaum ein:

```
AVLNode insert (AVLNode n, int c){
     if (n == null) return new AVLNode (c);
     if (c == n.content) { found = true;
                           grown = false;
     } else {
        if (c < n.content) {
           n.left = insert (n.left, c);
           if (grown) n.balance--;
        } else {
           n.right = insert (n.right, c);
           if (grown) n.balance++;
        }
        switch (n.balance) {
           case -2: if (n.left.balance == +1)
                    rotateLeft (n.left); // D-Rot
                    rotateRight (n);
                    grown = false; break;
           case -1: break;
           case  0: grown = false; break;
           case +1: break;
           case +2: if (n.right.balance == -1)
                    rotateRight (n.right); // D-Rot
                    rotateLeft (n);
                    grown = false; break;
           default: balanceError (n); break;
        }
     }
     return n;
  }
```

# Löschen

```
AVLNode delete (AVLNode n, int c){
   if (n == null) { found = shrunk = false;
                    return n; }
   if (c == n.content) {
      if (n.left == null) return n.right;
      if (n.right == null) return n.left;
      n.content = c = minValue (n.right);
   }
   if (c < n.content) {
      n.left = delete (n.left, c);
      if (shrunk) n.balance++;
   } else {
      n.right = delete (n.right, c);
      if (shrunk) n.balance--;    }
   switch (n.balance) {
      case -2: switch (n.left.balance) {
           case +1: rotateLeft(n.left); break;
           case  0: shrunk = false; break;
           case -1: break;
           default: balanceError (n.left); break;
         }
         rotateRight (n); break;
      case -1: shrunk = false; break;
      case  0: break;
      case +1: shrunk = false; break;
      case +2: switch (n.right.balance) {
           case -1: rotateRight(n.right); break;
           case  0: shrunk = false; break;
           case +1: break;
           default: balanceError (n.right); break;
         }
         rotateLeft (n); break;
      default: balanceError (n); break; }
   return n;
}
```

# 2 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 $k + 1$ Zeiger, die auf die Bereiche $(\cdot, x_1), (x_1, x_2), \ldots, (x_{k-1}, x_k), (x_k, \cdot)$ zeigen.

3. Die Zeiger, die einen Knoten verlassen, sind entweder alle `null`-Zeiger (Blätter) oder alle echte Zeiger.

4. Alle Blätter haben gleiche Tiefe.

Nutzung: Externe Speicherung ($m$ Seitengröße)

Bemerkung: 2-3 Bäume sind $B$-Bäume der Ordnung 3.

Speicherauslatung: Für alle Knoten mindestens 50 Prozent.

# Tiefe

Satz: Tiefe eines $B$-Baumes der Ordnung $m$ mit $n$ Daten liegt im Intervall
$[\lceil \log_m(n+1) - 1\rceil, \lceil \log_{\lceil m/2\rceil}((n+1)/2)\rceil]$

Beweis: Analog zu

Satz: Die Tiefe von 2-3-Bäumen mit $n$ Daten beträgt mindestens $\lceil \log_3(n+1)\rceil - 1$ und höchstens $\lceil \log(n+1)\rceil - 1$

Beweis: Maximale Datenzahl in Tiefe $d$ is
$2(1+3+\ldots+3^d) = 3^{d+1} - 1$ (vollständiger ternärer Baum)

Minimale Datenzahl in Tiefe $d$ is
$1+2+\ldots+2^d = 2^{d+1} - 1$ (vollständiger binärer Baum)

Für die Tiefe $d$ eines 2-3 Baumes mit $n$ Daten gilt:

$3^{d+1} - 1 \geq n$ und $d \varepsilon \lceil \log_3(n+1)\rceil - 1$

und

$2^{d+1} - 1 \geq n$ und $d \leq \lceil \log(n+1)\rceil - 1$

# Einfügen

Suche von $x$ In Knoten $v$ wird $x$ mit binärer Suche in $\log_m$ Vergleichen eingefügt mit $\log m + \log_m n \approx \log n$

(Nach erfolgloser Suche)

Einfügen von $x$ Gefundener `null`-Zeiger wird um $x$ erweitert.

Allgemein ist Zeiger auf Knoten mit Datum $y$ gegeben, so daß die Blätter des zugehörigen Teilbaumes eine Ebene zu tief liegen. Enthält der Knoten (inclusive $m$)

- weniger als $m - 1$ Daten, dann wird $y$ eingefügt.
- Genau $m$ Daten $z_1 < \ldots < z_m$, d.h. Knoten ist voll.

Bilde 3 Knoten mit den folgenden Daten:

$z_1, \ldots, z_{\lceil m/2 \rceil - 1}$ (links, $\lceil m/2 \rceil$ -1 Daten)
$z_{\lceil m/2 \rceil + 1}, \ldots, z_m$ (rechts, $\geq \lceil m/2 \rceil$ -1 Daten)
$y$ (Vater, 1 Datum)

und verteile die Nachfolgezeiger des ursprünglichen Knotens

Rekursion: Gehe Suchpfad einen Schritt zurück und fahre analog fort

# Löschen

Nach erfolgreicher Suche:

1. Wenn $x$ nicht in einem Blatt abgelegt ist, wird $x$ mit dem größten Datum $y < x$ vertauscht. Dieses Datum ist mit Sicherheit in einem Blatt

2. Entferne $x$ und `null`-Zeiger. Enthält der Knoten (exclusive $m$)

- mindestens $\lceil m/2 \rceil$ Daten, dann wird $x$ gelöscht
- ein Bruder mindestens $\lceil m/2 \rceil$ Daten, dann genügt eine einfache Datenrotation
- alle Brüder $\lceil m/2 \rceil$ Daten, dann wird kontraktiert.

Betrachte o.b.d.A. linken Bruder und $x'$ als trennendes Datum im Vater. Dieses $x'$ wird als Knoten extrahiert.

Nun wird aus den 3 Knoten ein Knoten mit
$2\lceil m/2 \rceil - 2 \leq m - 1$ Daten und
$\lceil m/2 \rceil + \lceil m/2 \rceil - 1$ Nachfolgezeiger

Rekursion: Gehe Suchpfad einen Schritt zurück und fahre analog fort

# Anhängen (2-3 Baum)

Ziel: Konstruiere aus den 2-3 Bäumen $T_1$ und $T_2$, wobei alle Daten in $T_1$ kleiner als alle Daten in $T_2$ sind, den 2-3-Baum $T$, der die Daten aus $T_1$ und $T_2$ enthält.

Annahme: Schlüssel in Blättern, innere Knoten entalten daß größte Datum des linken Teilbaumes und (falls vorhanden) des mittleren Teilbaumes.

1. Fall: $d(T_1) = d(T_2)$. $T$ besteht nur aus einer Wurzel, die als neuer Knoten daß größere Datum $z$ aus $T_1$ und als Teilbäume $T_1$ (ohne dieses Datum) und $T_2$ enthält.

2. Fall: O.B.d.A. $d(T_1) > d(T_2)$. Bilde $T'$ mit Zeiger auf die Wurzel von $T_2$. Gehe von der Wurzel in $T_1$ $d(T_1) - d(T_2) - 1$ mal zum rechten Kind und ende an $v$.
2a. $v$ hat enthält ein Datum $x$: Schreibe $x$ mit den 2 Teilbäumen in die Wurzel $T'$, zusätzlich kommt daß größere Datum $z$ aus $T_1$ in die Wurzel.
2b. $v$ hat enthält zwei Daten $x$ und $y$: Bilde weiteren Wurzelknoten mit $y$ an der Wurzel, $x$ zur linken und $z$ als Wurzel von $T'$ zur rechten. Reihe die 4 Teilbäume entsprechend ein.

Satz: Anhängen (engl. Concatenate) kann für 2-3 Bäume in Zeit $O(|d(T_1) - d(T_2)| + 1)$ durchgeführt werden.

# Aufteilen (2-3 Baum)

Ziel: Konstruiere aus den 2-3 Bäumen $T$, der $a$ enthält, zwei 2-3 Bäume, wobei $T_1$ alle Daten $x \leq a$ und $T_2$ alle Daten $x > a$ enthält.

Suche: Auf dem Weg zum Blatt mit Schlüssel $a$ werden die folgenden Informationen gespeichert: Der Suchpfad und alle Kinder von Knoten auf dem Suchpfad.

Aufbau von $T_1$: (analog $T_2$) Seien $T_1^1, \ldots, T_1^m$ die Teilbäume, für $T_1$. Es gilt: $d(T_1^i) \geq d(T_1^{i+1})$.

Algorithmus: Für alle $i = m-1, \ldots, 1$ entsteht $T_1^i$ aus dem Anhängen von $T_1^{i+1}$ an $T_1^i$. Schließlich ist $T_1 = T_1^1$

Beobachtung: Das Anhängen aller $T_1^i$ mit $d(T_1^i) \leq d'$ ergibt einen 2-3 Baum der Tiefe $\leq d' + 1$.
(Beweis durch Induktion über $d'$)

Analyse: Insgesamt hängen $m - 1 \leq 2d(T) - 1$ mal an. Es seien $d_1 < \ldots < d_k$ so gewählt, daß mindestens einer anzuhängenden Bäume $T_1^j$ Tiefe $d_i$ hat, $1 \leq i \leq k$. Die Konkatenateion mit den ein oder zwei Bäumen der Tiefe $d_{i+1}$ verursacht kosten von $O(d_{i+1} - d_i + 1)$ also insgesamt $O(d_k - d_1 + d(T)) = O(d(T))$

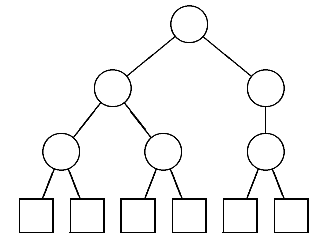
Satz: Aufteilen (engl. Split) kann für 2-3 Bäume in Zeit $O(d(T))$ durchgeführt werden.

# 3 Bruder-Bäume

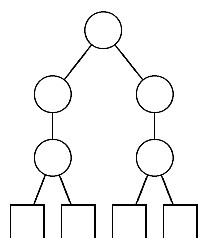
Idee: Bruder-Bäume kann man in einem präzisierbaren Sinn als expandierte AVL-Bäume auffassen.

Unärer Knoten Durch Einfügen unärer Knoten an den richtigen Stellen erhält man einen Baum, dessen sämtliche Blätter dieselbe Tiefe haben; und umgekehrt entsteht aus einem Bruder-Baum ein höhenbalancierter Baum, wenn man die unären Knoten mit ihren einzigen Söhnen verschmilzt.

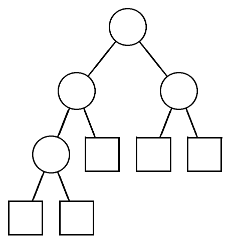
Definition: Ein binärer Baum heißt ein Bruder-Baum, wenn jeder innere Knoten einen oder zwei Söhne hat, jeder unäre Knoten einen binären Bruder hat und alle Blätter dieselbe Tiefe haben.



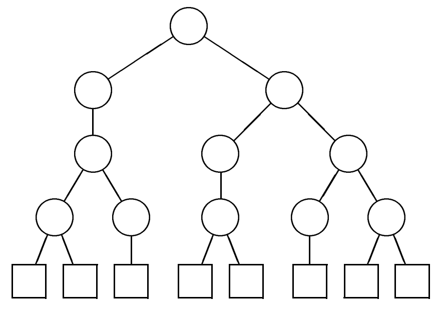
Bruder–Baum



kein Bruder-Baum



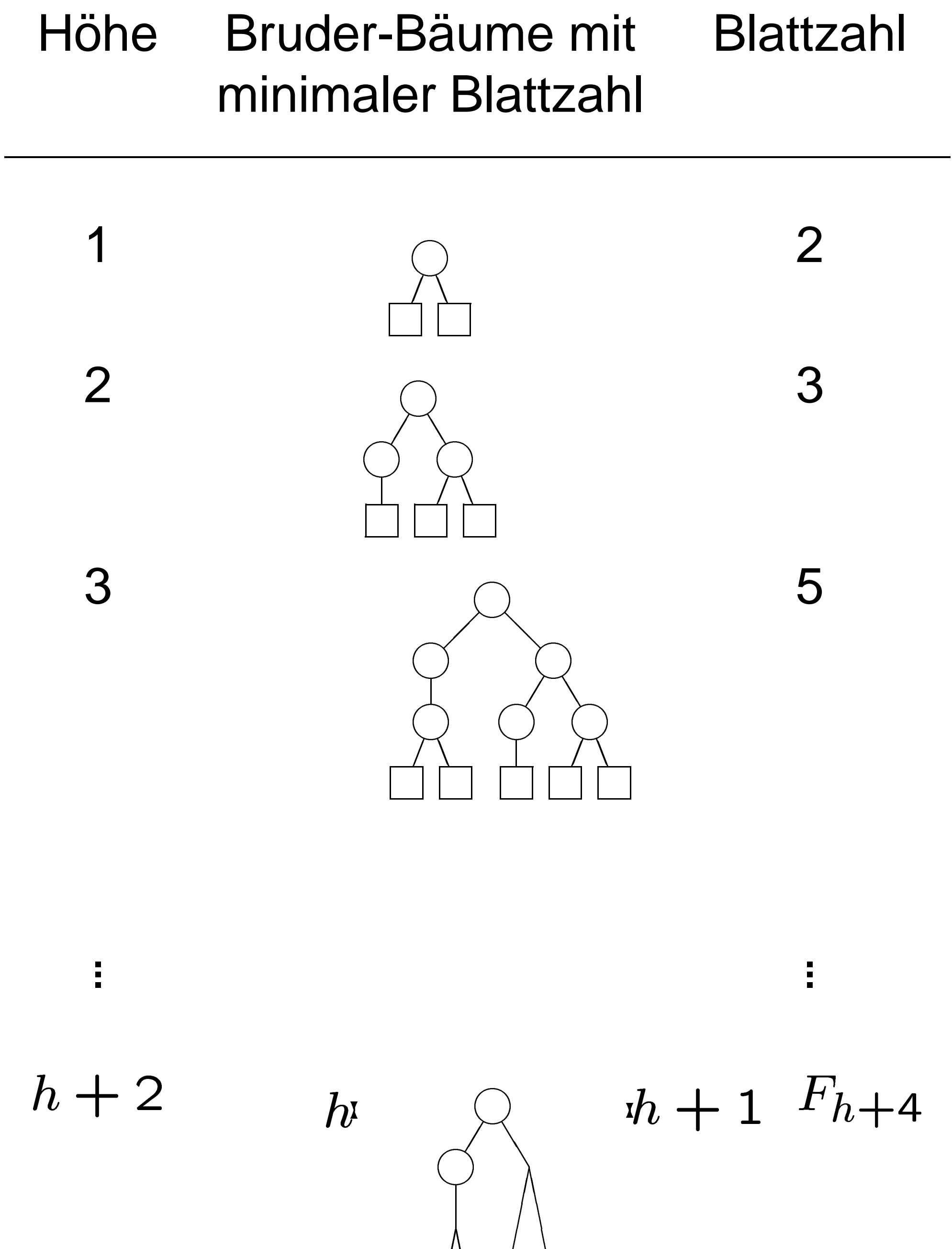
kein Bruder-Baum



Bruder-Baum

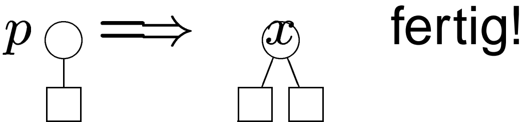
# Höhe

Ein Bruder-Baum mit Höhe $h$ hat wenigstens $F_{h+2}$ Blätter. ($F_i$ ist die $i$-te Fibonacci-Zahl.) Also umgekehrt: Ein Bruder-Baum mit $N$ Blättern und $(N-1)$ inneren Knoten hat eine Höhe $h \leq 1.44 \ldots \log_2 N$.

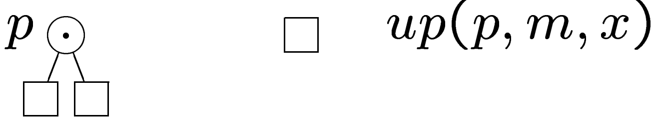


# Einfügen

Fall 1: $p$ hat nur einen Sohn




Fall 2: $p$ hat bereits zwei Söhne




Invariante: Wenn $\text{up}(p, m, x)$ aufgerufen wird, gilt:
(1) $p$ hat zwei Söhne $p_l$ und $p_r$, die beide Wurzeln von 1-2-Bruder-Bäumen sind.
(2) Der Knoten $m$ ist entweder ein Blatt oder hat einen einzigen Sohn, der Wurzel eines 1-2-Bruder-Baumes ist.
(3) Schlüssel im linken Teilbaum von $p < x$
$<$ Schlüssel im Teilbaum von $m$
$<$ Schlüssel von $p$
$<$ Schlüssel im rechten Teilbaum von $p$

# Eigenschaften

Fälle: (1) binär auf unär, (3) 2 binär auf binär, (2) sonst.

Statisch: Für jeden unären Knoten auf Niveau $l$ muß es einen binären Bruder auf demselben Niveau geben. Daher gilt für das Verhältnis

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

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

Somit $\frac{3}{5} \leq U \leq 1$ und $\geq 3/5$ der inneren Knoten binär

Da die amortisierte Anzahl von Aufrufen von up pro Einfügung konstant ist, ist der mittlere Aufwand zum Einfügen ist konstant

Bemerkung: Entsprechende Aussage für AVL-Bäume wesentlich schwieriger zu zeigen.

# Implementation Bruder-Bäume

## Grundstruktur:

```
class Btree{
    Bnode root;
    private final boolean UNARY = true;
    private final boolean BINARY = false;
    private String s;

    Btree(){
        root=new Bnode(0,null,null,UNARY);
    }
    Bnode search(int key) ...
    Bnode getFather(int key) ...
    void up(Bnode p, Bnode m ,int x) ...
    void insert(int key) ...
    public String toString() ..
}
```

**GetFather:** Liefert den Vater des Knotens $p$ mit `p.key==key` bzw. `null`, falls `p==root`. Existiert kein Knoten $p$ mit `p.key==key`, so wird der Knoten mit nächst größerem Schlussel zurück geliefert.

```
Bnode getFather(int key) {
  Bnode dummy = root;
  Bnode d = null;
  if (dummy.isunary) return null;
  while (dummy!=null && dummy.key!=key){
    if (dummy.isunary) dummy = dummy.leftSon;
      d = dummy;
      if(dummy.key < key) dummy = dummy.rightSon;
      else dummy = dummy.leftSon;
    }
  return d; }
```

# Suche und Einfügen in Bruder-Bäumen

```
Bnode search(int key) {
  Bnode dummy = getFather(key);
  if (dummy==null) return root;
    if (dummy.leftSon==null) return null;
    dummy = (dummy.key<key) ?
             dummy.rightSon :
             dummy.leftSon;
    return (dummy.key==key) ?
            dummy :
            null;
}
```

Insert fügt den Schlüssel `key` in den Baum ein

```
void insert(int key) {
  Bnode p=getFather(key);
  if (p==null) {
    root.key=key;
    root.isunary=false;
  }
  else up(p,null,key);
}
```

Up

```
void up(Bnode p, Bnode m ,int x) {
  Bnode father=(p!=null) ? getFather(p.key) : null;
    if (p.key < x){
      int temp = p.key; p.key = x; x = temp;
    }
  }
  ...
}
```

# Fälle in Up

## 1. Fall $p$ hat keinen Bruder

```
if (p==root || father.isunary) {
  root = new Bnode(x,
    new Bnode(0,(p.leftSon==null) ? null : p.leftSon,
              null,UNARY),
    (p.rightSon==null) ?
      new Bnode(p.key,null,null,BINARY) :
      new Bnode(p.key,m,p.rightSon,BINARY),
    BINARY);
}
```

## 2. Fall $p$ hat einen linken Bruder mit nur einem Sohn

```
if (father.leftSon!=null &&
    (father.leftSon).isunary) {
  father.leftSon =
    new Bnode(father.key,
              (father.leftSon).leftSon,
              p.leftSon,BINARY);
     father.key=x;
     (father.rightSon).leftSon=m;
}
```

## 3. Fall $p$ hat einen rechten Bruder mit nur einem Sohn

```
if (father.rightSon!=null &&
    (father.rightSon).isunary) {
  father.rightSon =
    new Bnode(father.key,
              (father.rightSon).rightSon,
              p.rightSon,BINARY);
    father.key=x;
    (father.leftSon).rightSon=m;
}
```

# Weitere Fälle in Up

## 4. Fall $p$ hat einen linken Bruder mit zwei Söhnen

```
if(father.leftSon!=null &&
   !(father.leftSon).isunary) {

  (father.rightSon).leftSon=m;
  int temp=father.key; father.key=x;
  up (father,
      new Bnode(0,
                (father.leftSon).leftSon,
                null,UNARY),
     temp);
}
```

## 5. Fall $p$ hat einen rechten Bruder mit zwei Söhnen

```
if (father.rightSon!=null &&
  !(father.rightSon).isunary) {
  (father.leftSon).rightSon=m;
  int temp=father.key; father.key=x;
  up (father,
      new Bnode(0,
                (father.rightSon).rightSon,
                null,UNARY),
      temp);
}
```

# 4 Rot-Schwarz-Bäume

(Knotengefärbte) Rotschwarz-Bäume sind binäre Suchbäume mit den folgenden Eigenschaften: $i)$ Jeder Knoten ist entweder rot oder schwarz. $ii)$ Jedes Blatt ist schwarz. $iii)$ Falls ein Knoten rot ist, dann sind beide Kinder schwarz. $iv)$ Jeder Pfad von einem Knoten zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten.

Satz: Ein (knotengefärbter) Rotschwarzbaum mit $n$ inneren Knoten besitzt eine Höhe von höchstens $2\log(n+1)$.

Rotation: Entsprechend den AVL-Bäumen gibt es Operationen `Left-Rotate` und `Right-Rotate`.

Einfügen: Wir starten mit dem eingefügten Knoten und der Färbung “rot” und hangeln uns je nach Lage und Farbe der Ahnen mittels Rotationen und Umfärbungen über die Vorgängerverweise an den Knoten zu den Eltern und Ureltern hinauf.