Bäume

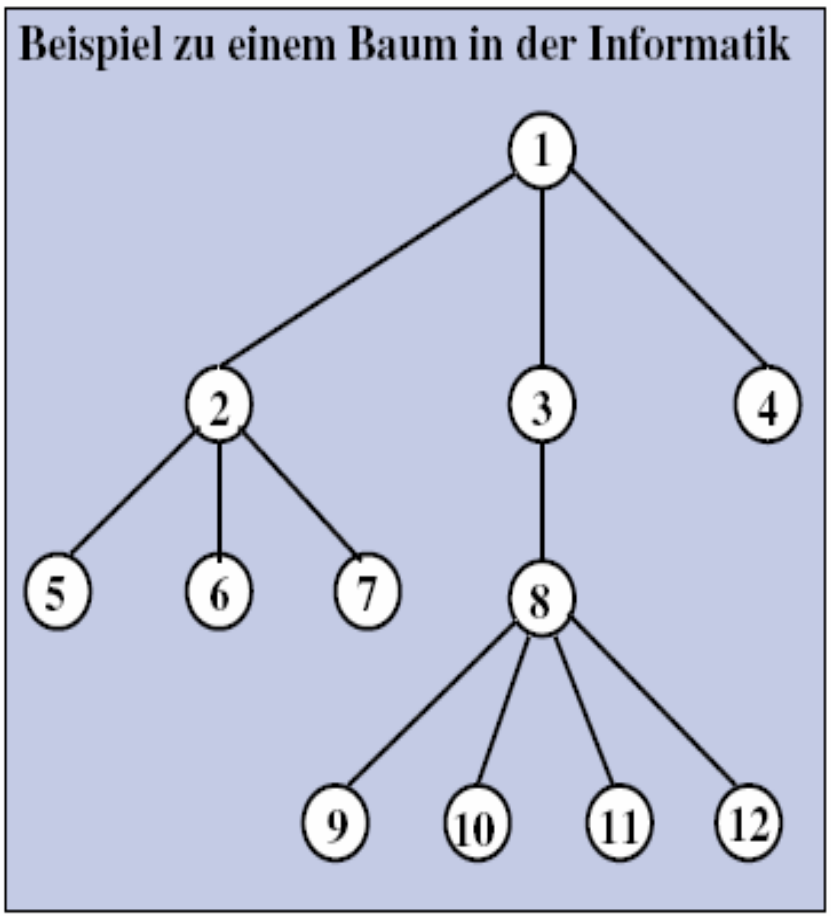
Grundlegendes zu Bäumen

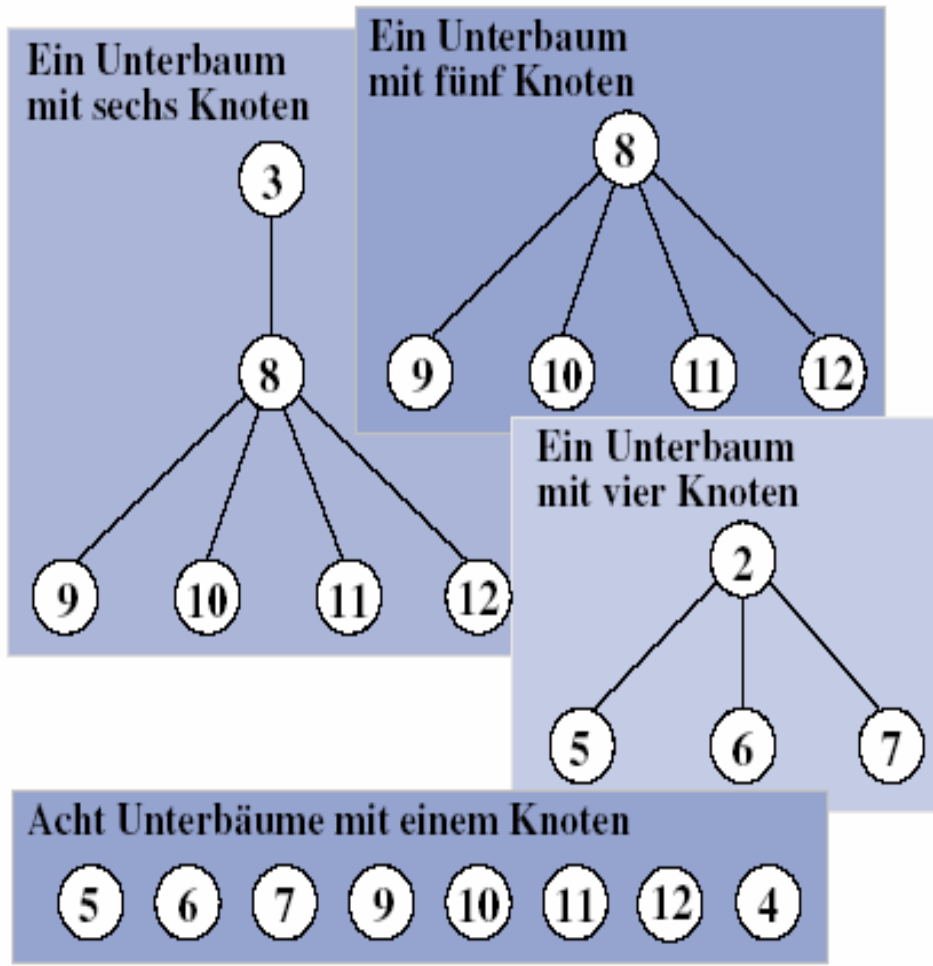
- **Baum** = Menge von Knoten u. Kanten, die Beding. erfüllen
- **Knoten** = Objekt, das Namen und weitere Infos haben kann
- **Kante** = Verbindung zwischen zwei Knoten
- **Pfad** = Folge von unterschiedlichen Knoten, die durch Kanten im Baum miteinander verbunden sind
- **Wurzel** = besonderer Knoten (Ursprung des Baums)

Es gilt immer, dass es zwischen Wurzel und jedem beliebigen anderen Knoten eines Baums genau einen Pfad gibt.

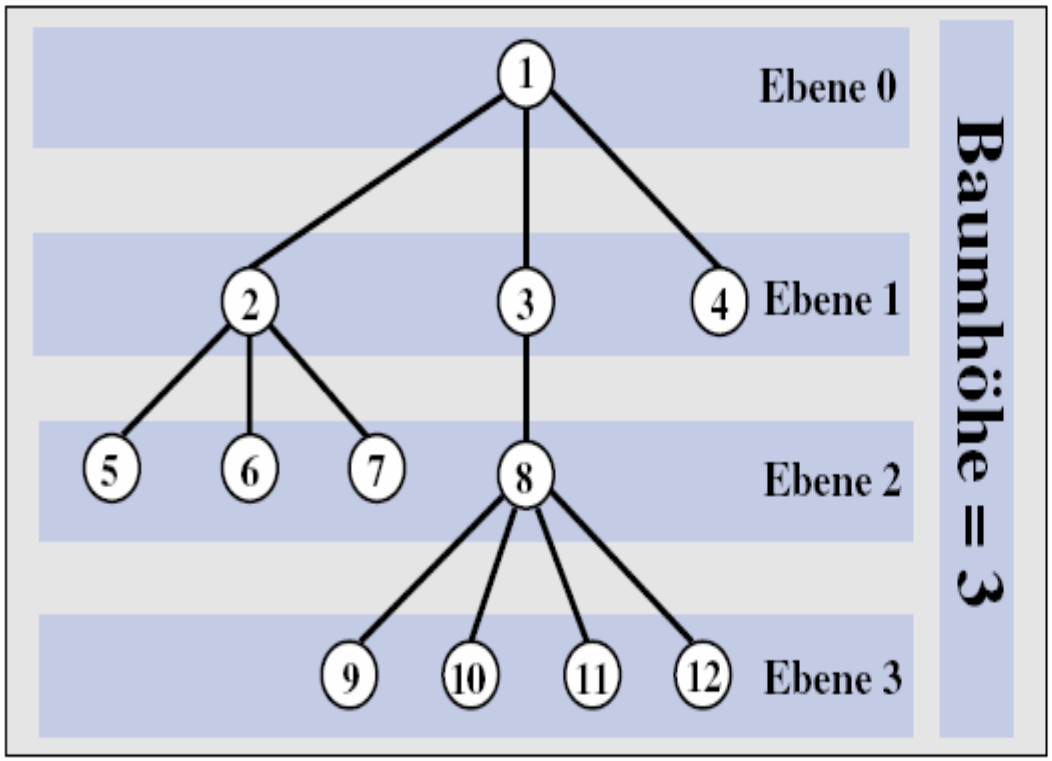
Gibt es zw. Wurzel und einem der Knoten mehr als einen oder auch keinen Pfad → kein Baum, sondern Graph.

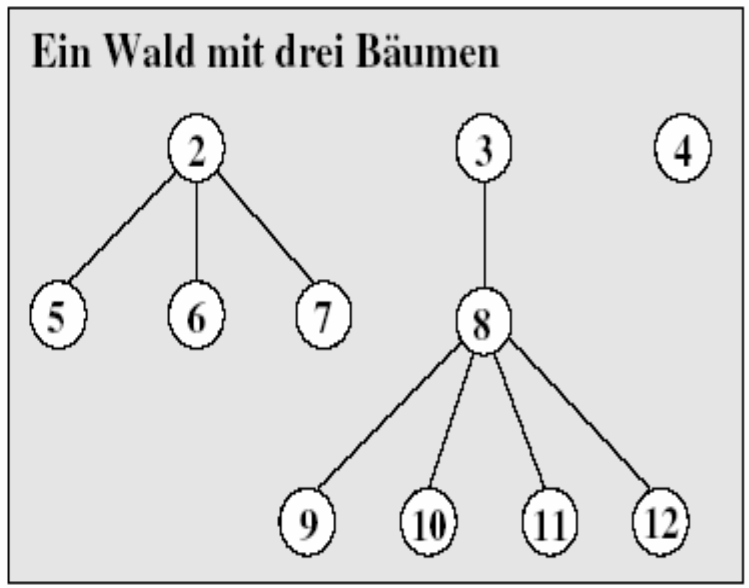
Bäume


Beispiel zu einem Baum in der Informatik
1
2
3
4
5
6
7
8
9
10
11
12


Ein Unterbaum mit sechs Knoten
3
8
9
10
11
12
Ein Unterbaum mit fünf Knoten
8
9
10
11
12
Ein Unterbaum mit vier Knoten
2
5
6
7
Acht Unterbäume mit einem Knoten
5
6
7
9
10
11
12
4

Bäume





(2) (3) (4) : 1 + 1 + 1 = 3 +

(5) (6) (7) (8) : 2 + 2 + 2 + 2 = 8 +

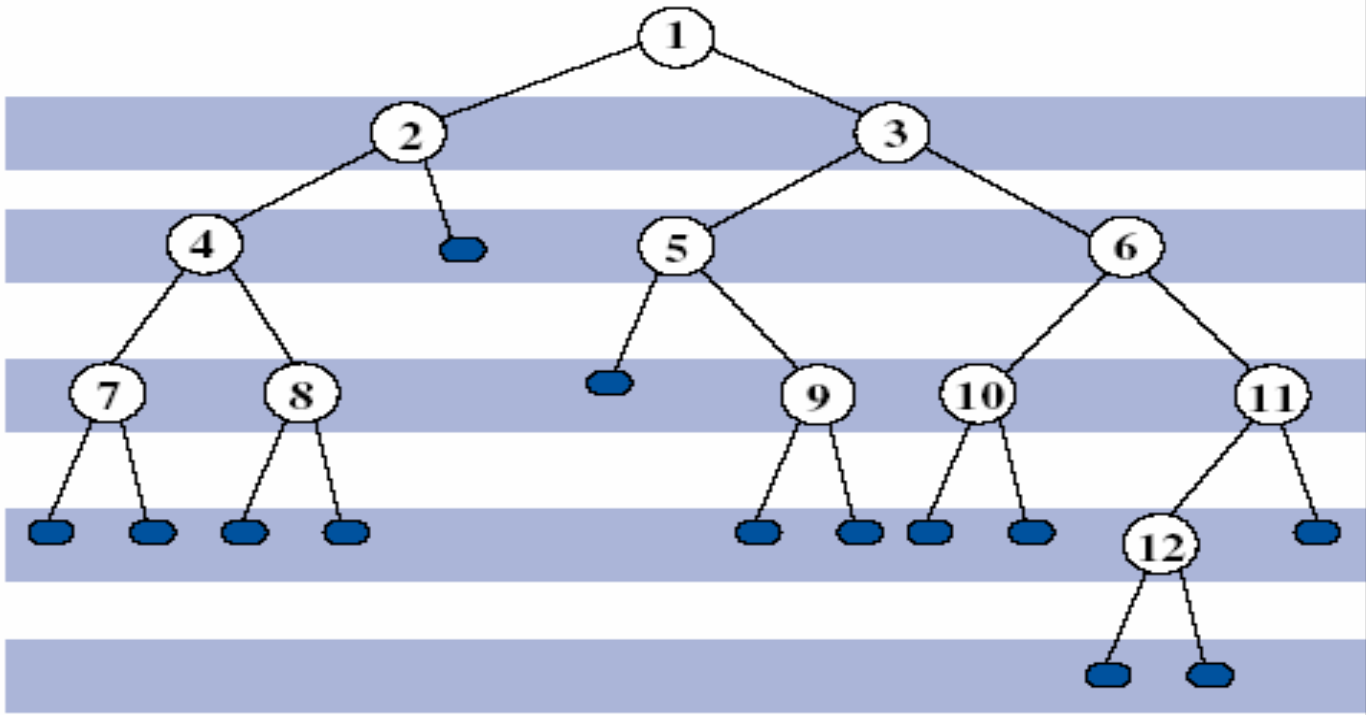
(9) (10) (11) (12) : 3 + 3 + 3 + 3 = 12 = 23 > Pfadlänge des Baums = 23

Binäre Bäume (BB)

- Binärer Baum = geordneter Baum mit 2 Typen von Knoten:
  - Innere Knoten haben immer max. zwei direkte, geordnete Nachfolger (linker und rechter Nachfolger).
  - Äußere Knoten sind Knoten ohne Nachfolger.

- **BB ist leer**, wenn er nur aus einem äußeren Knoten besteht und keinen inneren Knoten besitzt.
- **Voller binärer Baum** ist ein binärer Baum, in dem sich in keiner Ebene, außer in vorletzten, äußere Knoten befinden.
- **Vollständiger Baum** ist voller binärer Baum, bei dem sich in der letzten Ebene nur äußere Knoten befinden.

Binäre Bäume (BB)



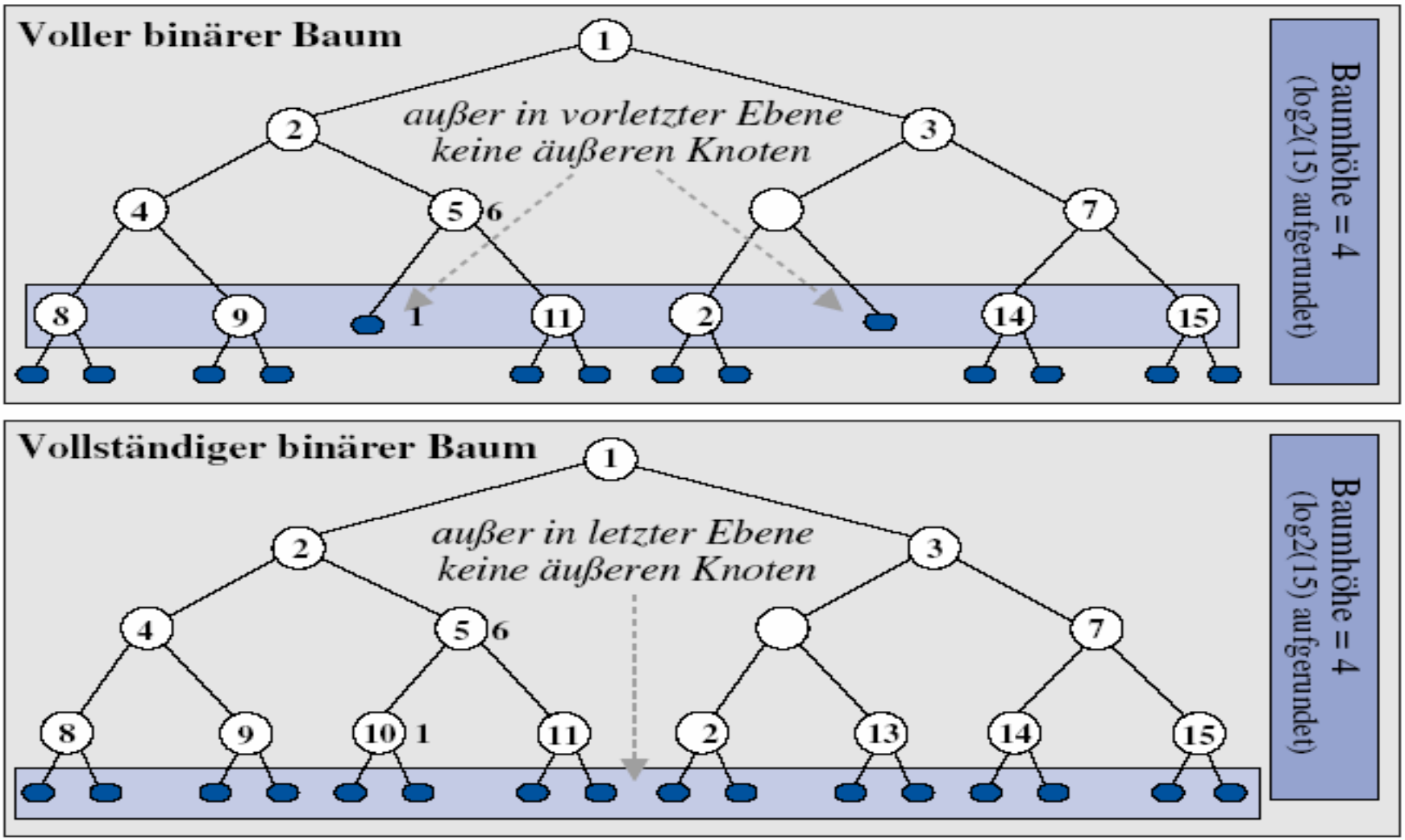
	Innere Pfadlänge	Äußere Pfadlänge
	2	
	6	2
	15	3
	4	36
	0	10
Summe	27	51

Innere Knoten: 12
Äußere Knoten: 13

Innere Pfadlänge: 27
Äußere Pfadlänge: 51 (= 27+24 = 27 + 2*12 Knoten)

Baum ohne äußere Knoten: 12 Knoten, 11 Kanten
Baum mit äußeren Knoten: 25 Knoten, 24 Kanten

Binäre Bäume (BB)

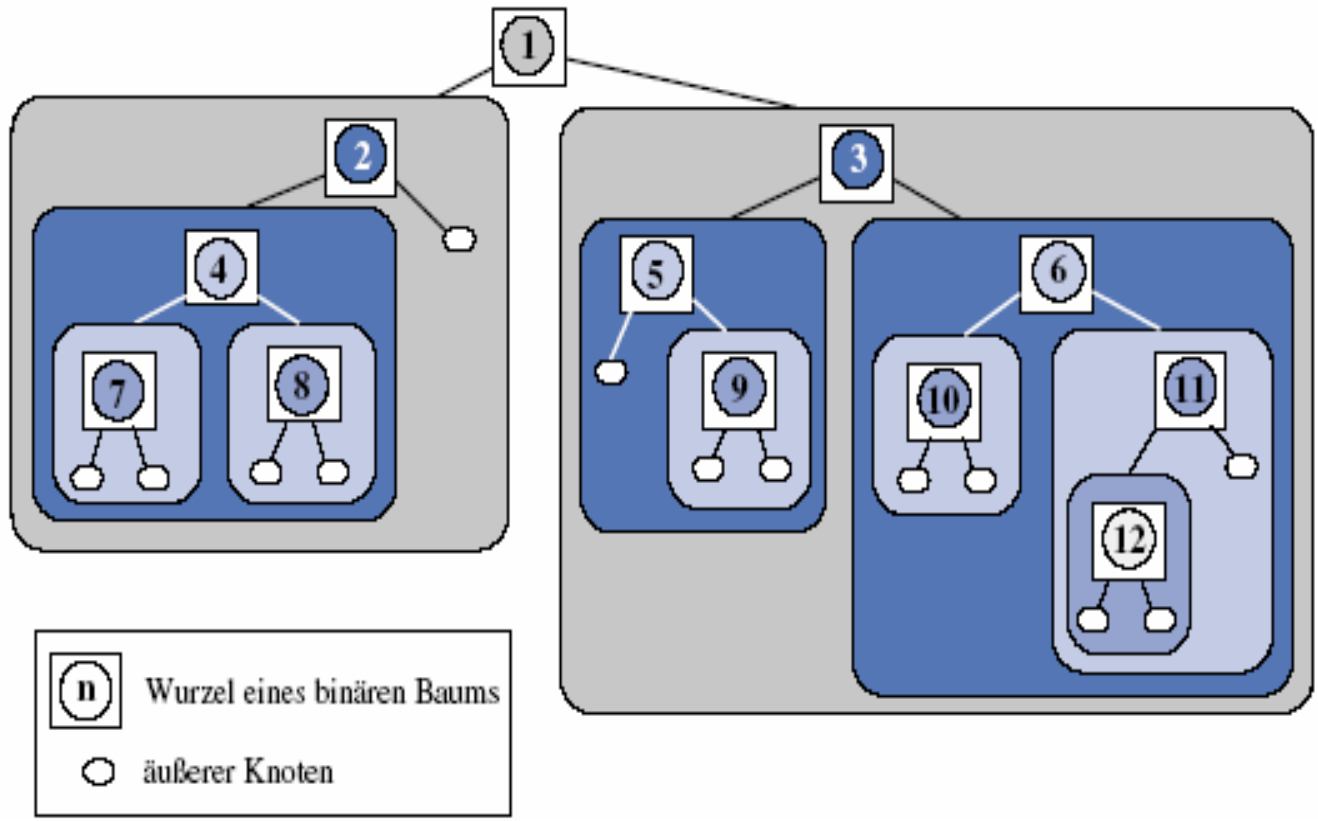


Binäre Bäume (BB)

Binäre Bäume haben Vorteile von Arrays (schneller Zugriff auf bestimmte Elemente) und auch Vorteile von Listen (leichtes Einfügen bzw. Entfernen eines Elements).

Binäre Bäume lassen sich rekursiv definieren:

Binärer Baum ist entweder Wurzel eines anderen BB oder ein äußerer Knoten.



Realisierung von binären Bäumen in C/C++ und Java

```
struct node { /* In C */
    int          zahl;
    struct node *links;
    struct node *rechts;
};
```

```
class Node { // In Java
    int    zahl;
    Node links;
    Node rechts;
    Node(int z) { zahl = z; links = rechts = null; }
}
```

Für hier verwendeten binären Baum soll Folgendes gelten:

1. **Knoten eines BB kann nicht mehr als 2 Nachkommen besitzen, wobei ein oder gar kein Nachkomme möglich ist.**
2. **Jeder linke Nachkomme eines Knotens ist kleiner als der Knoten selbst → alle Zahlen in Knoten des gesamten linken Unterbaums sind kleiner als Zahl im Knoten selbst.**
3. **Jeder rechte Nachkomme eines Knotens ist größer als der Knoten selbst → alle Zahlen in Knoten des gesamten rechten Unterbaums sind größer als Zahl im Knoten selbst.**

Realisierung binärer Bäume in C

```
struct node { /* binbaumsort.c */
    int           zahl;
    struct node *links;
    struct node *rechts;
};
struct node *wurzel = NULL;
struct node *neuerKnoten(int zahl) {
  struct node *knot = (struct node *)
              malloc(sizeof(struct node));
  knot->zahl = zahl;
  knot->links = knot->rechts = NULL;
  return knot;
}
void einordnen(int zahl) {
  if (wurzel == NULL)
     wurzel = neuerKnoten(zahl);
  else
     insert(zahl, wurzel);
}
```

```
void insert(int zahl, struct node *k) {
  if (zahl < k->zahl) {
     if (k->links == NULL)
        k->links = neuerKnoten(zahl);
     else
        insert(zahl, k->links);
  } else {
     if (k->rechts == NULL)
        k->rechts = neuerKnoten(zahl);
     else
        insert(zahl, k->rechts);
  }
}
void drucke_baum(struct node *k) {
  if (k != NULL) {
     drucke_baum(k->links);
     printf("%d, ", k->zahl);
     drucke_baum(k->rechts);
  }
}
```

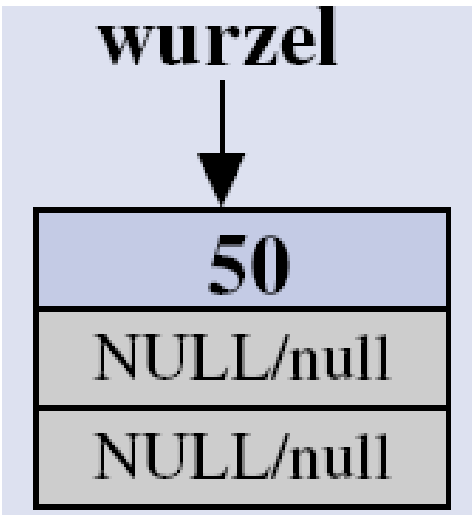
Realisierung binärer Bäume in Java

```
class Node {
    int     zahl;
    Node links;
    Node rechts;
    Node(int z) {
        zahl = z;
        links = rechts = null;
    }
}
private static Node wurzel = null;
static void insert(int zahl, Node k) {
    if (zahl < k.zahl) {
        if (k.links == null)
            k.links = new Node(zahl);
        else
            insert(zahl, k.links);
    } else {
        if (k.rechts == null)
            k.rechts = new Node(zahl);
        else
            insert(zahl, k.rechts);
    }
}
```

```
public static void einordnen(int zahl) {
    if (wurzel == null)
        wurzel = new Node(zahl);
    else
        insert(zahl, wurzel);
}
public static void drucke_baum(Node k) {
    if (k != null) {
        drucke_baum(k.links);
        System.out.print(k.zahl + ", ");
        drucke_baum(k.rechts);
    }
}
```

Realisierung binärer Bäume in C/Java

1. Nach Eing. 1. Zahl 50 wird **einordnen(zahl)** aufgerufen. Da **wurzel** beim 1. Aufruf von einordnen() NULL / null ist, wird in einordnen() mit **wurzel = neuerKnoten(zahl)** bzw. **wurzel = new Node(zahl)** ein neuer Knoten angelegt.

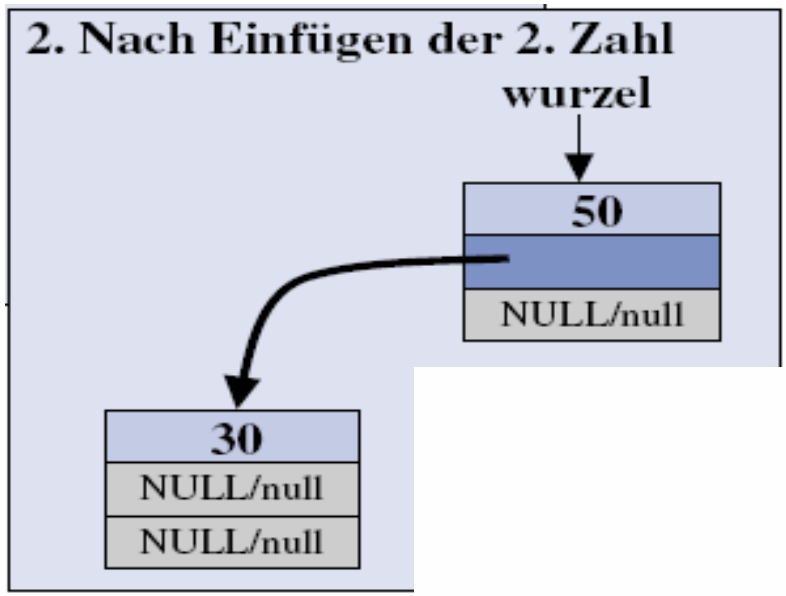


Realisierung binärer Bäume in C/Java

2. Nach Eing. 2. Zahl 30 wird **einordnen(zahl)** aufgerufen. Da nun **wurzel** nicht mehr NULL/null, wird in einordnen() **insert(zahl, wurzel)** aufgerufen. In insert() wird fett gedruckte Zeile ausgeführt, da 30 < 50 im wurzel-Knoten und linke Nachfolger des wurzel-Knotens NULL/null:

```
if (zahl < k->zahl) { /* In C */
    if (k->links == NULL)
        k->links = neuerKnoten(zahl);
```

```
if (zahl < k.zahl) { // in Java
    if (k.links == null)
        k.links = new Node(zahl);
```



Realisierung binärer Bäume in C/Java

3. Nach 3. Zahl 40 wird **einordnen(zahl)** aufgerufen.
   Da wurzel nicht mehr NULL/null → **insert(zahl, wurzel).**
   In insert() wird fette Zeile ausgeführt, da 40 < 50 im wurzel-Knoten u. linke Nachfolger des wurzel-Kn. nicht NULL/null:

```
if (zahl < k->zahl) { /* In C */
    if (k->links == NULL)
        k->links = neuerKnoten(zahl);
    else
        insert(zahl, k->links);
```

```
if (zahl < k.zahl) {
    if (k.links == null)
        k.links = new Node(zahl);
    else
        insert(zahl, k.links);
```

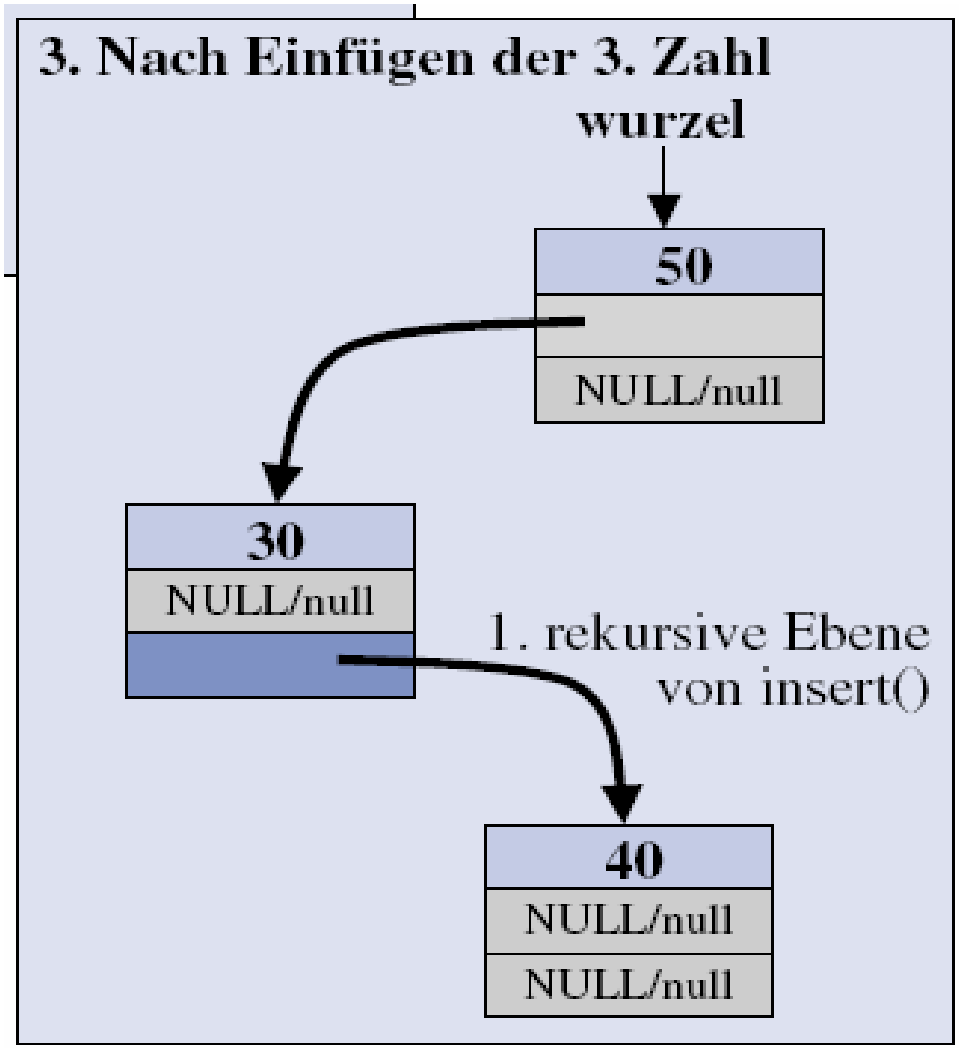
Realisierung binärer Bäume in C/Java

Es wird also insert() rekursiv aufgerufen. Da nun Zahl 40 größer als Zahl 30 im linken Nachfolger-Knoten des wurzel-Knotens ist und der rechte Nachfolger dieses Nachfolger-Knotens NULL/null ist, wird fette Codezeile ausgeführt:

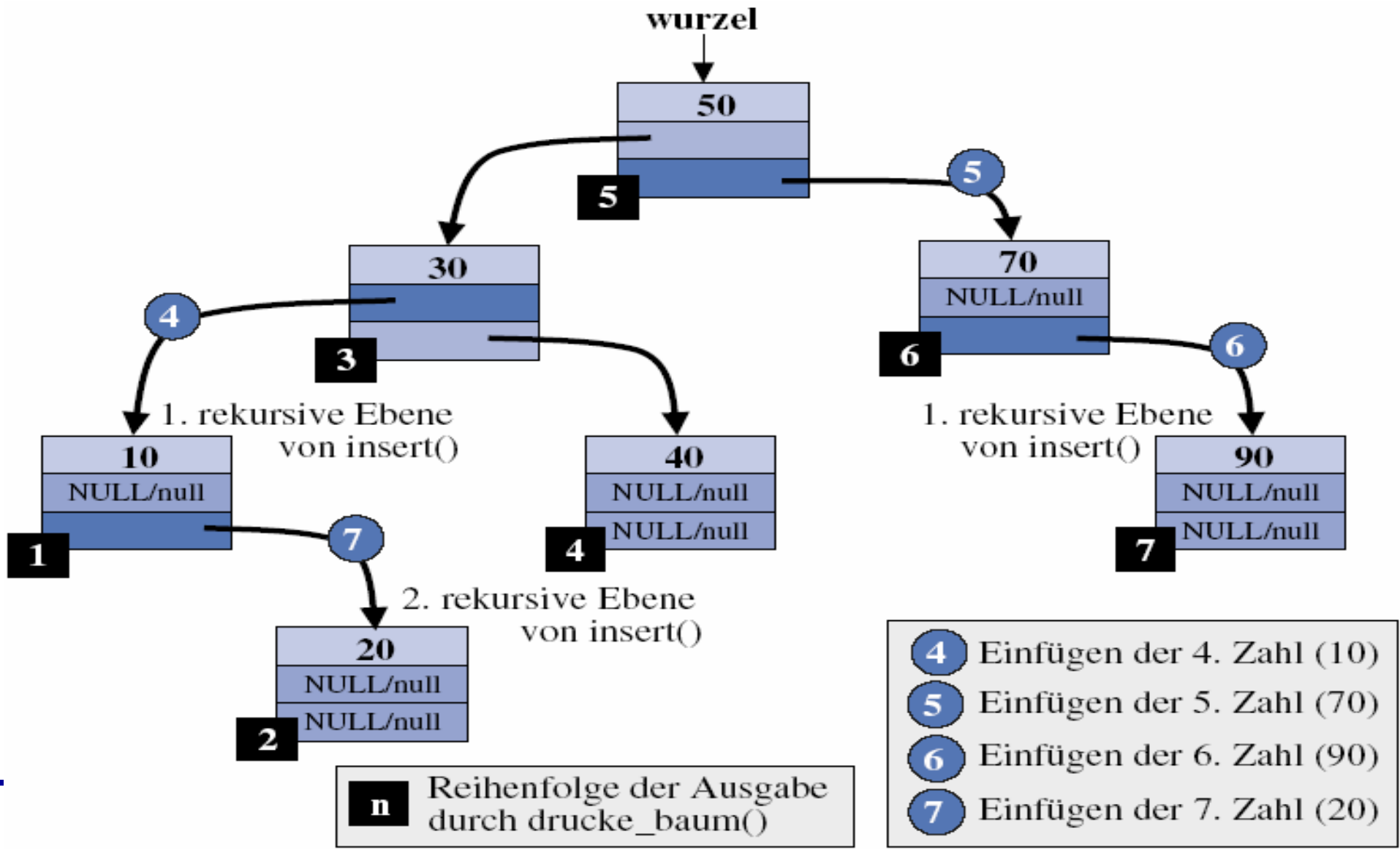
```
if (zahl < k->zahl) { /* in C */
    if (k->links == NULL)
        k->links = neuerKnoten(zahl);
    else
        insert(zahl, k->links);
} else {
    if (k->rechts == NULL)
        k->rechts = neuerKnoten(zahl);
```

```
if (zahl < k.zahl) {
    if (k.links == null)
        k.links = new Node(zahl);
    else
        insert(zahl, k.links);
} else {
    if (k.rechts == null)
        k.rechts = new Node(zahl);
```

Realisierung binärer Bäume in C/Java



Realisierung binärer Bäume in C/Java

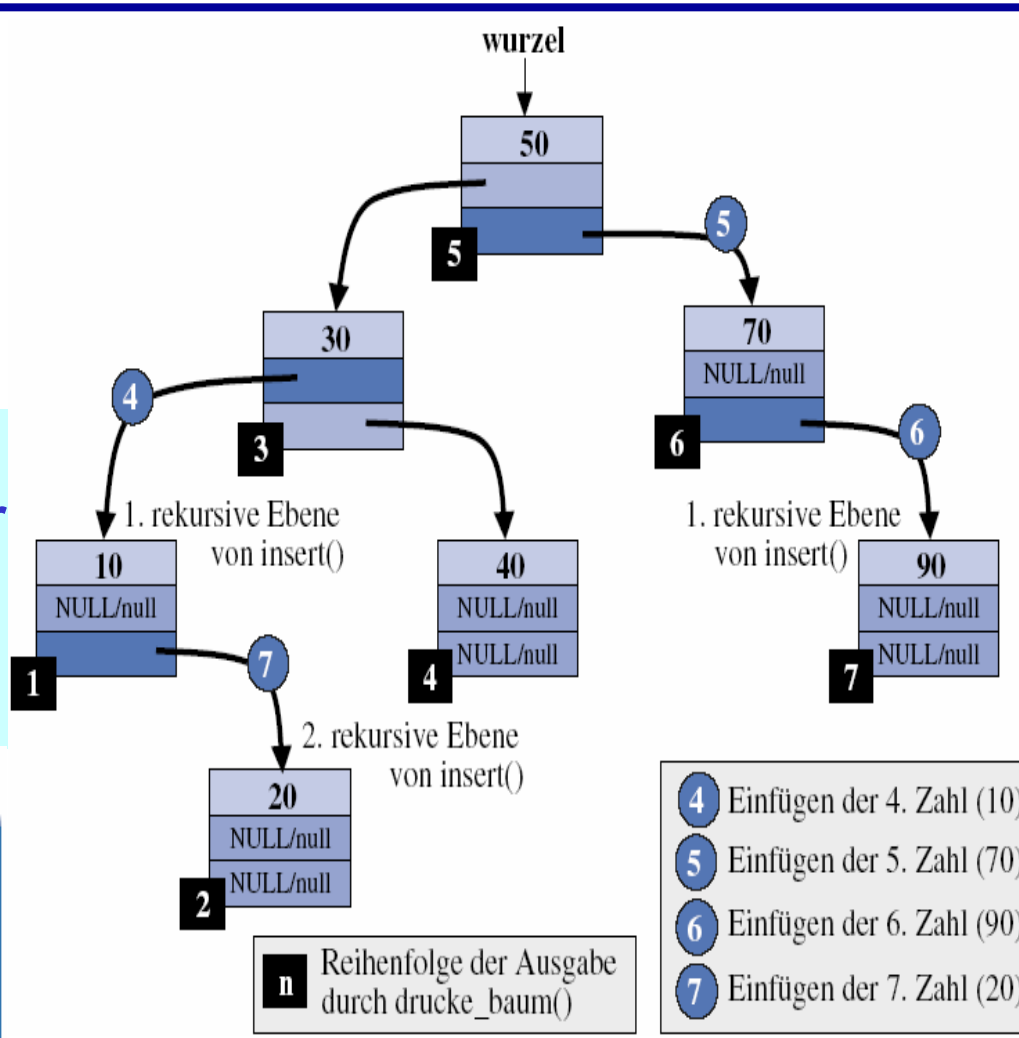


Realisierung binärer Bäume in C/Java

```
void drucke_baum(struct node *k) {
  if (k != NULL) {
     drucke_baum(k->links);
     printf("%d, ", k->zahl);
     drucke_baum(k->rechts);
  }
}
```

Inorder-Traversierung:
Zu jedem Knoten wird zuerst linker Unterbaum, dann Zahl des Knoten selbst, und schließlich der rechte Unterbaum ausgegeben.

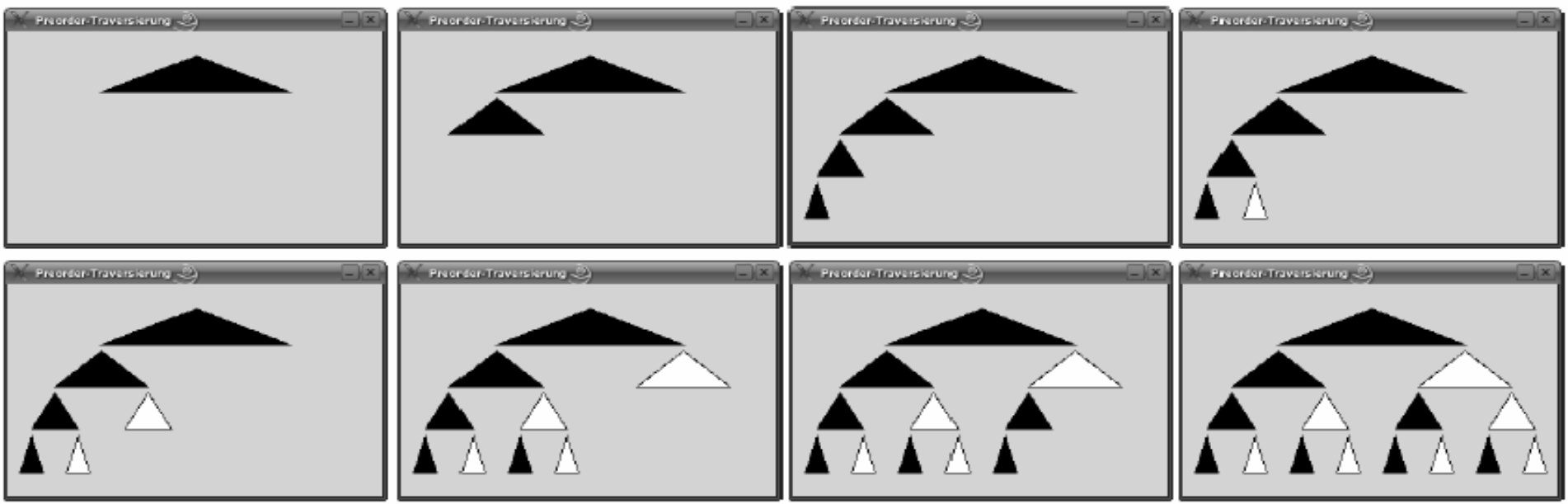
```
public static void drucke_baum(Node k) {
    if (k != null) {
       drucke_baum(k.links);
       System.out.print(k.zahl + ", ");
       drucke_baum(k.rechts);
    }
}
```



Preorder-Traversierung eines Binärbaums

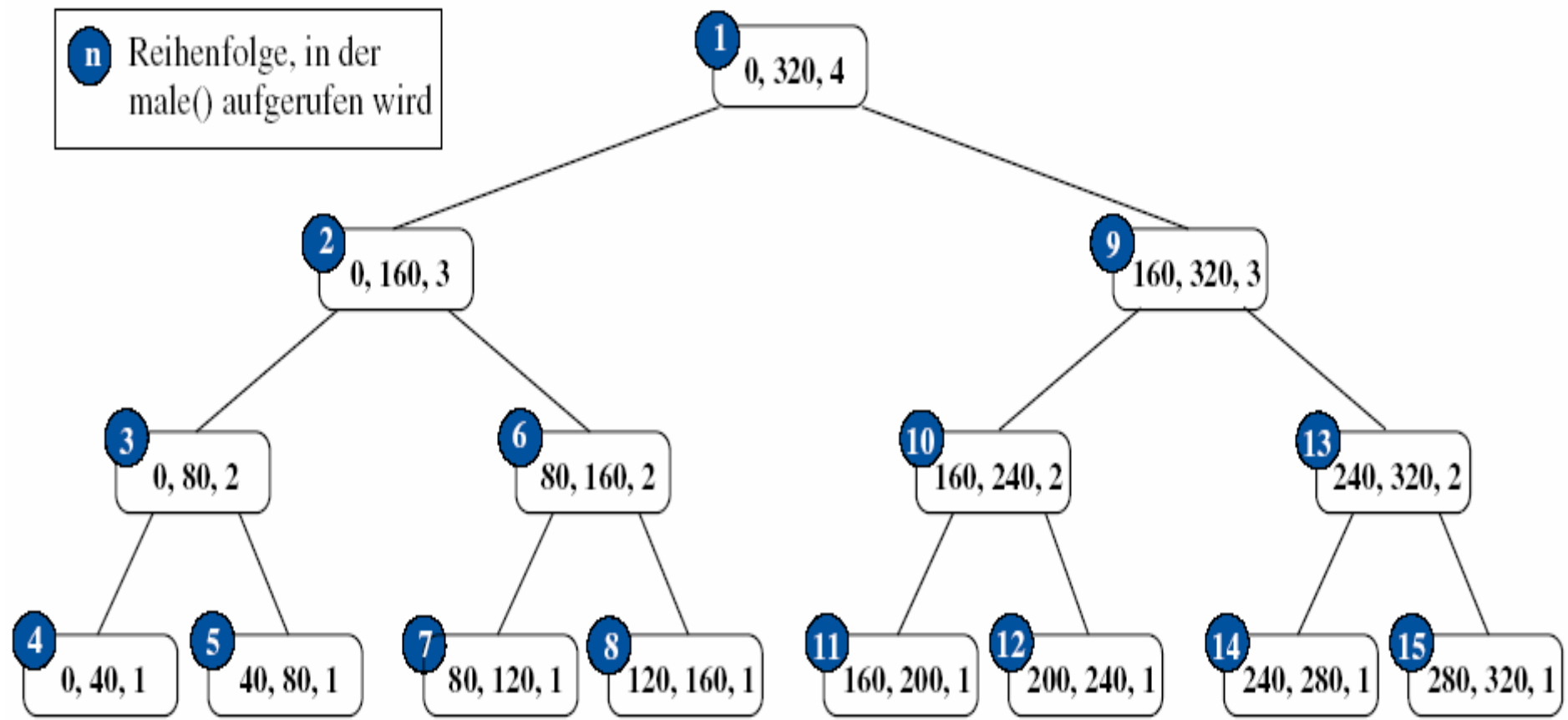
Bearbeite zuerst Wurzel, dann bearbeite ganzen linken Unterbaum und anschließend ganzen rechten Unterbaum zu dieser Wurzel.

```
void dreieck(int l, int r, int h, Color farbe) {
    int m = (l+r)/2;
    if (h > 0) {
        male(l, r, h, farbe); // Preorder
        dreieck(l, m, h−1, Color.black);
        dreieck(m, r, h−1, Color.white);
    }
}
```



Preorder-Traversierung eines Binärbaums

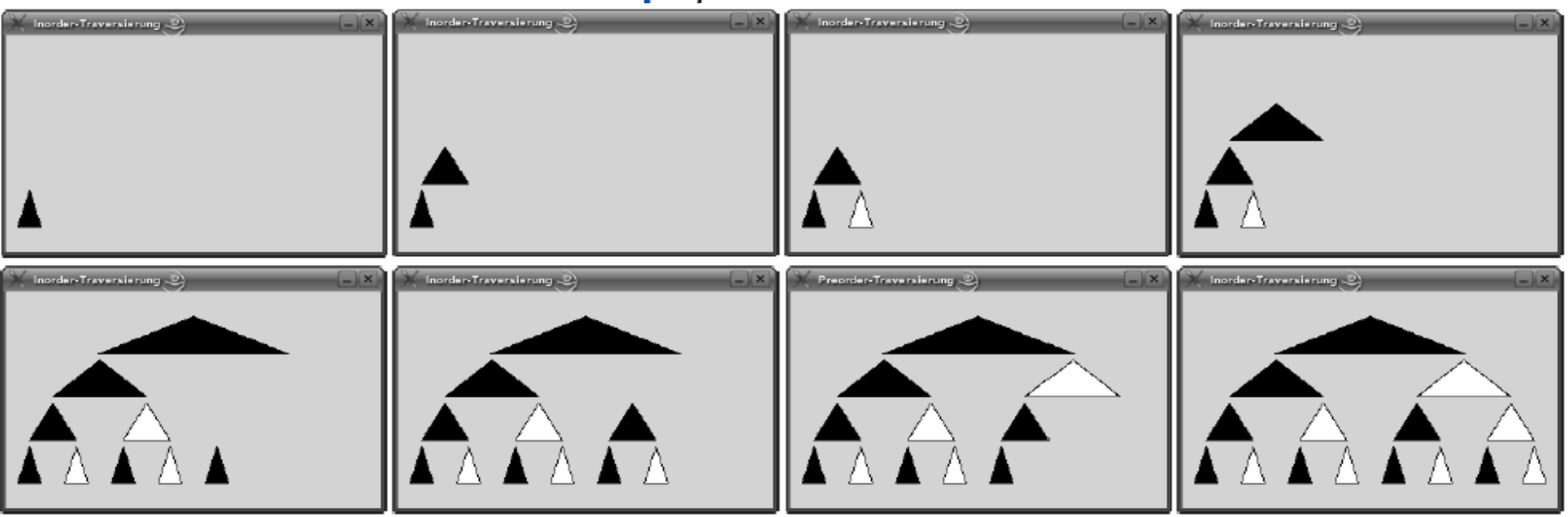
Die zuvor vorgestellte rekursive Funktion wird dann wie folgt aufgerufen: **dreieck(0, 320, 4, Color.black);**



Inorder-Traversierung eines Binärbaums

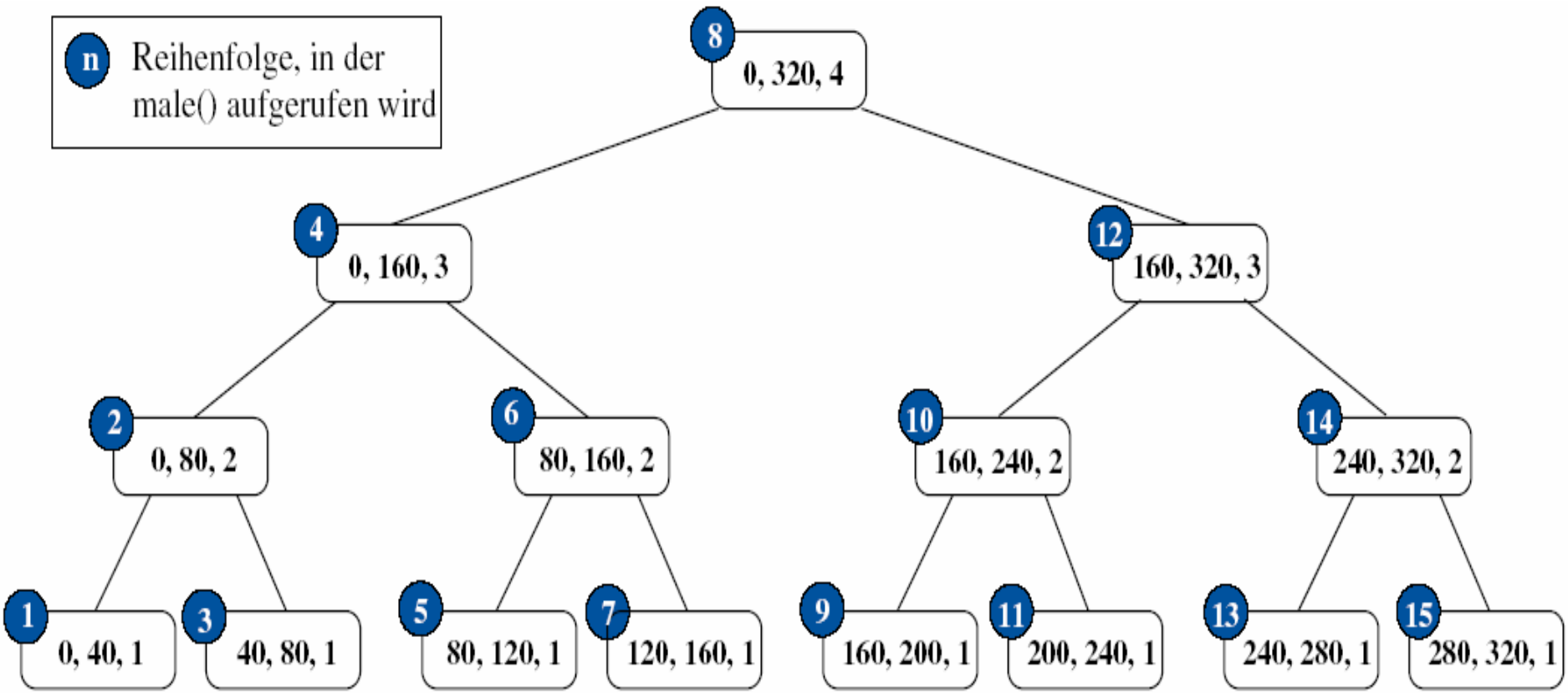
Bearbeite zuerst ganzen linken Unterbaum, dann Wurzel und dann ganzen rechten Unterbaum zu dieser Wurzel.

```
void dreieck(int l, int r, int h, Color farbe) {
    int m = (l+r)/2;
    if (h > 0) {
        dreieck(l, m, h−1, Color.black);
        male(l, r, h, farbe); // Inorder
        dreieck(m, r, h−1, Color.white);
    }
}
```



Inorder-Traversierung eines Binärbaums

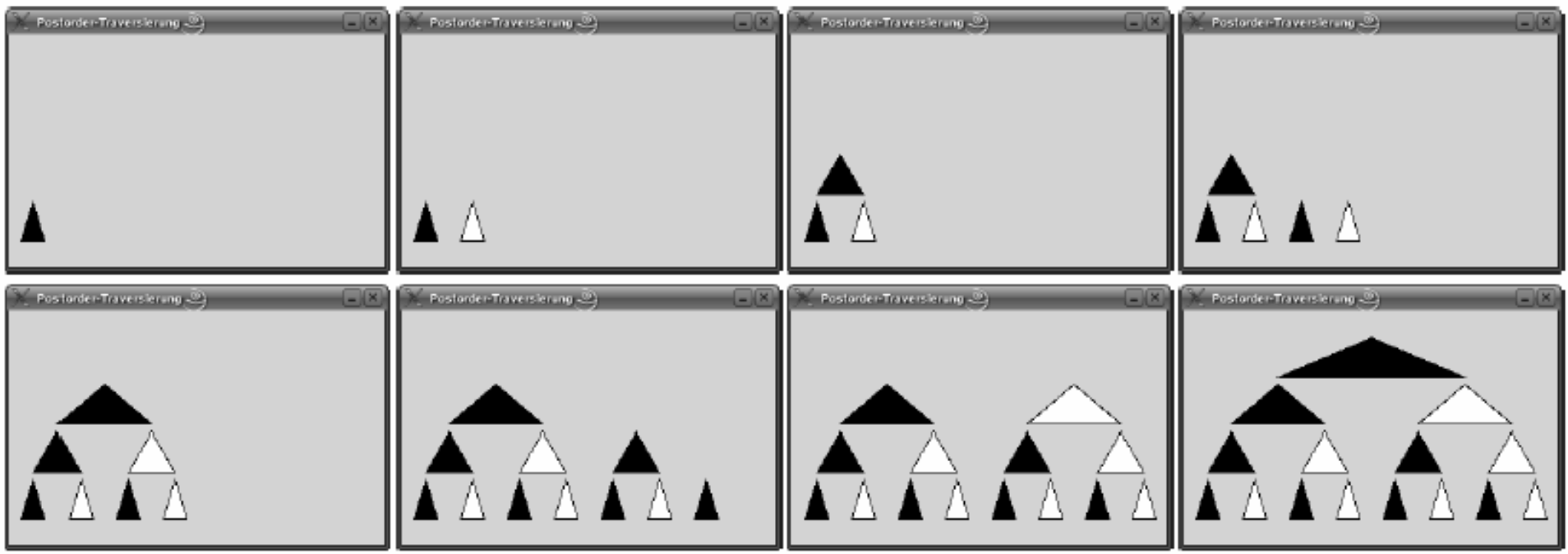
Die zuvor vorgestellte rekursive Funktion wird dann wie folgt aufgerufen: **dreieck(0, 320, 4, Color.black);**



Postorder-Traversierung eines Binärbaums

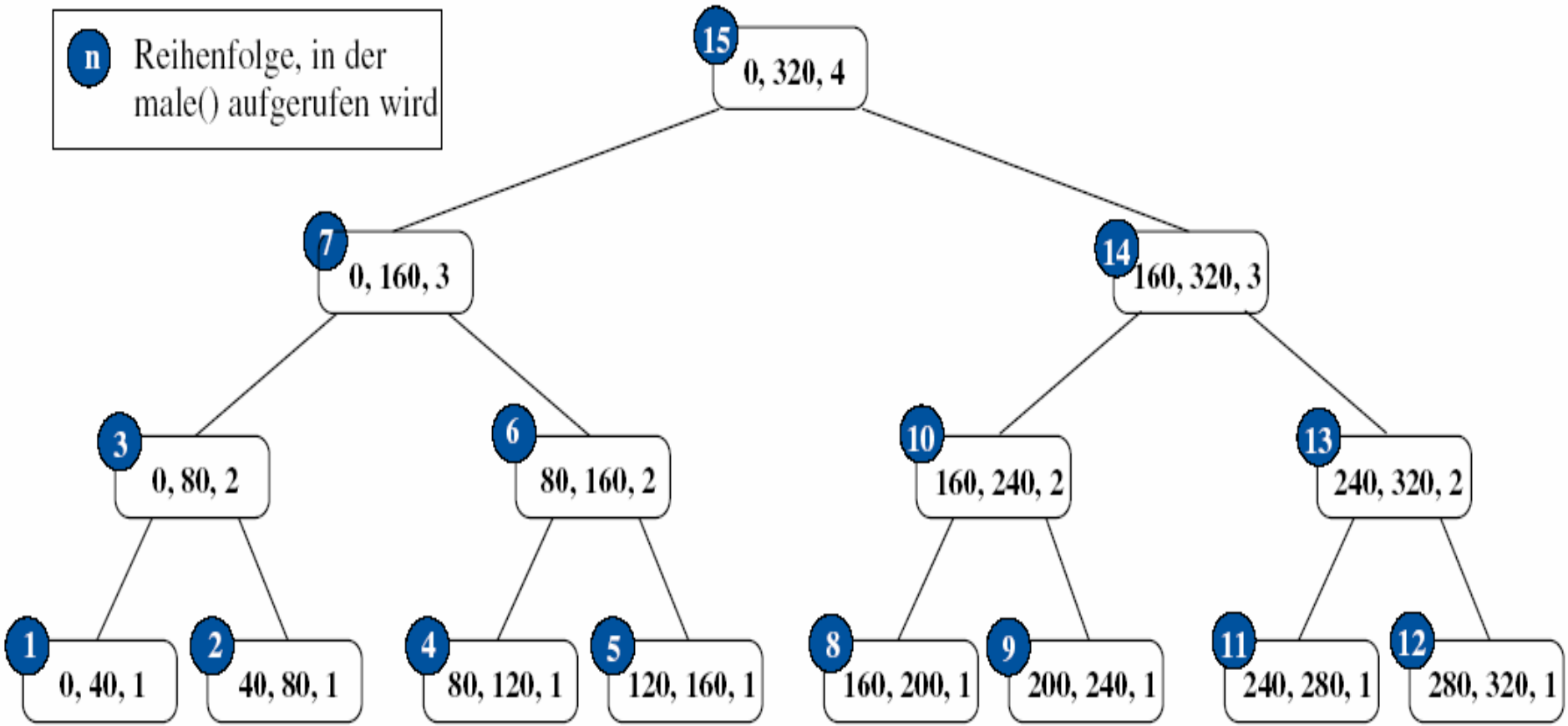
Bearbeite zuerst ganzen linken Unterbaum, dann dann ganzen rechten Unterbaum und zuletzt Wurzel zu diesen Unterbäumen.

```
void dreieck(int l, int r, int h, Color farbe) {
    int m = (l+r)/2;
    if (h > 0) {
        dreieck(l, m, h−1, Color.black);
        dreieck(m, r, h−1, Color.white);
        male(l, r, h, farbe); // Postorder
    }
}
```



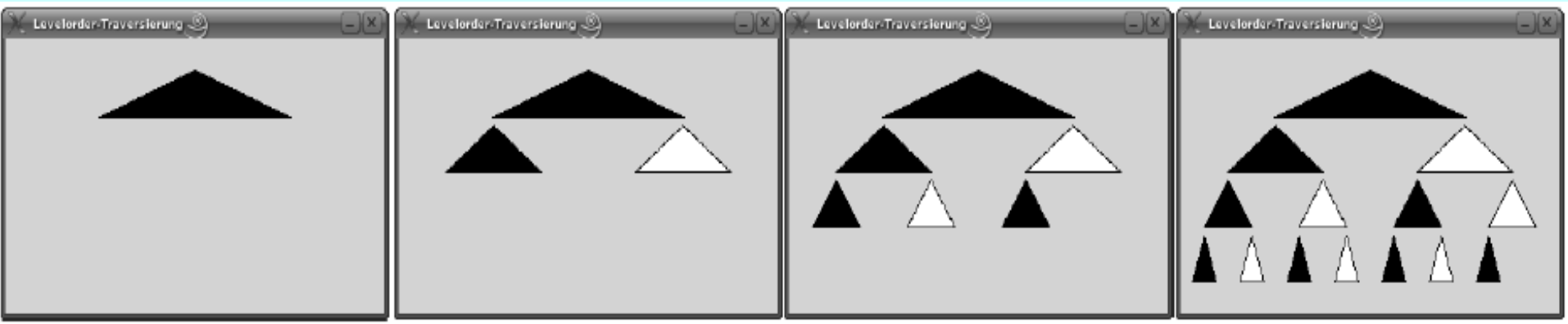
Postorder-Traversierung eines Binärbaums

Die zuvor vorgestellte rekursive Funktion wird dann wie folgt aufgerufen: **dreieck(0, 320, 4, Color.black);**



Levelorder-Traversierung eines Binärbaums

- Bei Levelorder-Traversierung immer zuerst Wurzel und dann Knoten der nächsten Ebene ganz durchlaufen, bevor auf nächste Ebene abgestiegen wird.
- Hier eine Queue als Zwischenspeicher für einzelne Knoten einer Ebene, wobei Wurzeln der einzelnen Unterbäume jeweils am Ende dieser Queue eingereiht werden, so dass Knoten der nächsten Ebene in richtiger Reihenfolge in die Queue eingefügt werden.



Grundlegende Operationen auf einem binären Baum

```c
struct node { /* In C */
    int          zahl;
    struct node *links;
    struct node *rechts;
};
```

```java
class Node { // In Java
    int  zahl;
    Node links;
    Node rechts;
    Node(int z) { zahl = z; links = rechts = null; }
}
```

Höhe eines Binärbaums

```c
int hoehe(struct node *k) { /* in C */
  if (k == NULL)
    return 0;
  else {
    int hl = hoehe(k->links),
    int hr = hoehe(k->rechts);
    return (hl > hr) ? hl+1 : hr+1;
  }
}
```

```java
int hoehe(Node k) { // in Java
  if (k == null)
    return 0;
  else {
    int hl = hoehe(k.links),
    int hr = hoehe(k.rechts);
    return (hl > hr) ? hl+1 : hr+1;
  }
}
```

Grundlegende Operationen auf einem binären Baum

Anzahl der Knoten in einem Binärbaum

```
int   anzahl(struct node *k) {   /* in C */
    return (k==NULL) ? 0 : anzahl(k->links) + anzahl(k->rechts) + 1;
}
```

```
int   anzahl(Node k) {  // in Java
    return (k==null) ? 0 :  anzahl(k.links) + anzahl(k.rechts) + 1;
}
```

Vollständiger Binärbaum

Binärbaum ist vollständiger Binärbaum der Höhe h, wenn er $2^h - 1$ Knoten besitzt. In diesem Fall haben alle Blätter die Höhe h und alle inneren Knoten besitzen zwei Nachfolger.

Grundlegende Operationen auf einem binären Baum

Anzahl der Blätter in einem Binärbaum

```
int  blattzahl(struct node *k) { /* in C */
    if (k == NULL)
        return 0;
    else if (k->links == NULL && k->rechts == NULL)
        return 1;
    return blattzahl(k->links) + blattzahl(k->rechts);
}
```

```
int  blattzahl(Node k) { // in Java
    if (k == null)
        return 0;
    else if (k.links == null && k.rechts == null)
        return 1;
    return blattzahl(k.links) + blattzahl(k.rechts);
}
```

Löschen eines Knotens aus dem Binärbaum

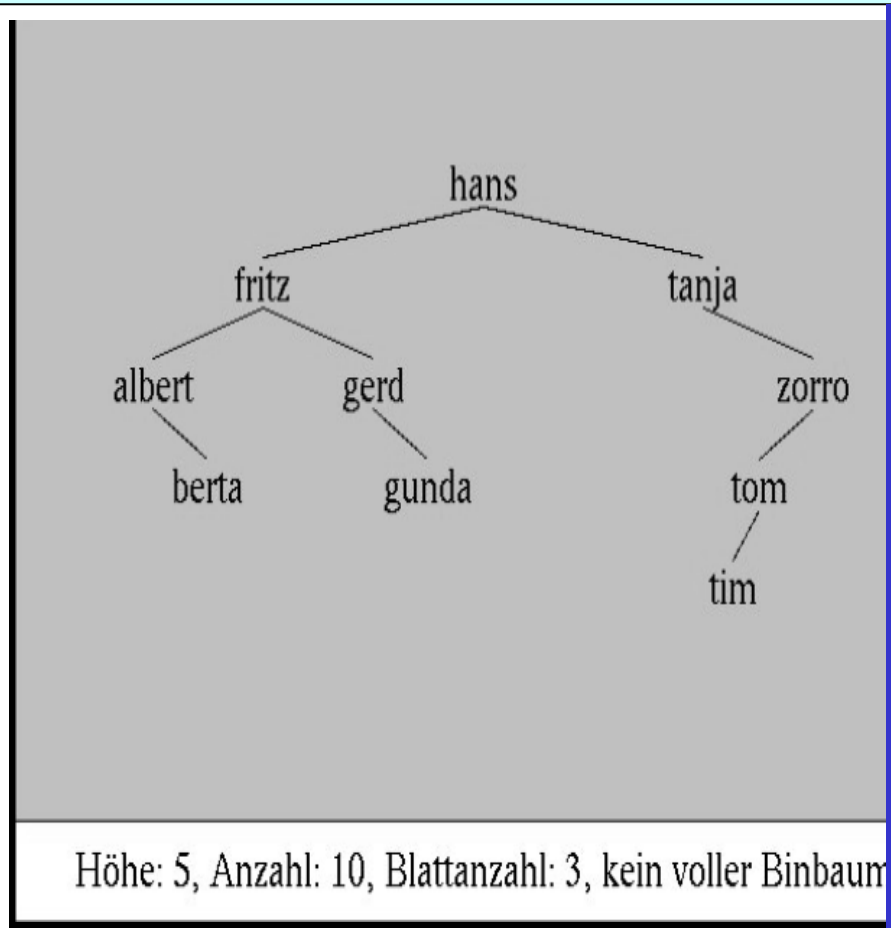
1. **Entfernen eines Blatts:**
   Knoten ohne Nachfolger können einfach entfernt werden.
2. **Entfernen eines Knotens mit nur einem Nachfolger:**
   Zeiger auf zu entfernenden Knoten wird eine Kopie des nicht leeren Nachfolger-Knotens zugewiesen:

```
k = k->links; bzw. k = k->rechts; /* in C */
k = k.links;   bzw.  k = k.rechts;   // in Java
```

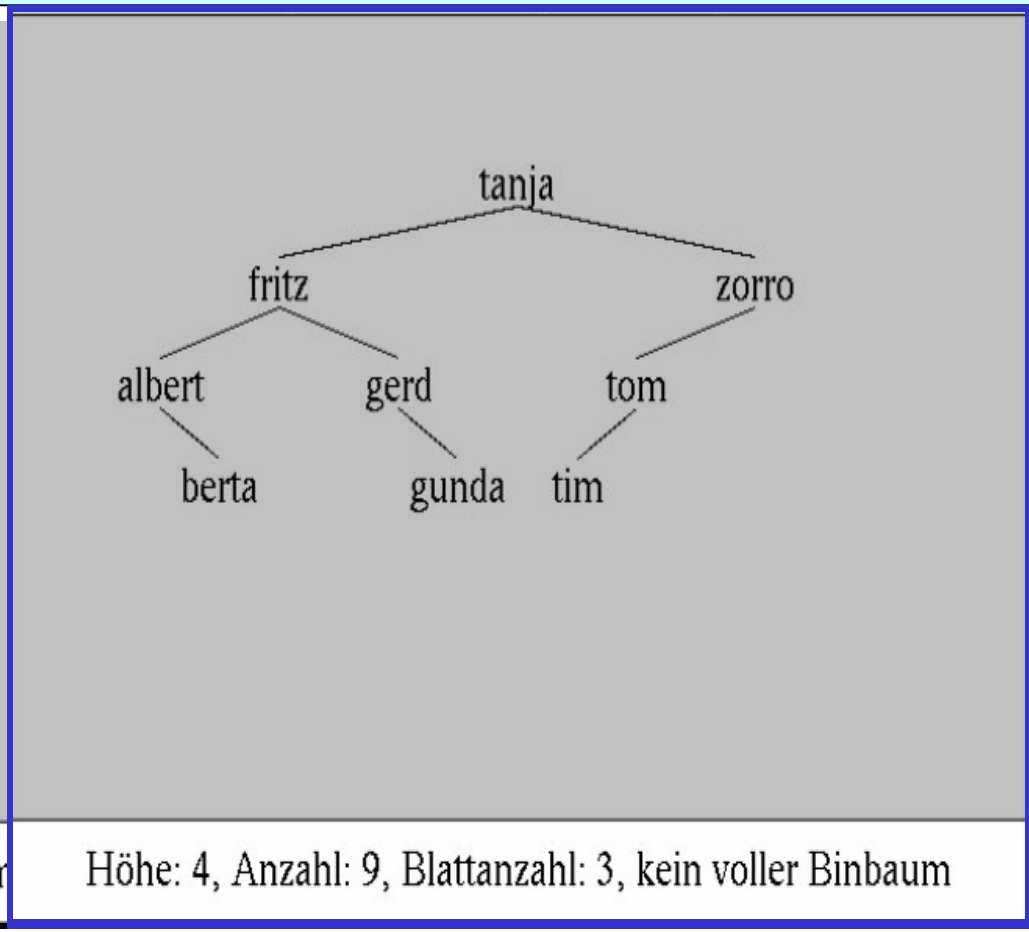
3. **Entfernen eines Knotens mit zwei Nachfolgern:**
   Um Binärbaum mit seinen Eigenschaften zu erhalten, muss an Stelle des zu löschenden Knotens der größte Knoten in seinem linken Teilbaum oder der kleinste Knoten in seinem rechten Teilbaum treten.

Löschen eines Knotens aus dem Binärbaum

Löschen von „hans“



Höhe: 5, Anzahl: 10, Blattanzahl: 3, kein voller Binbaum



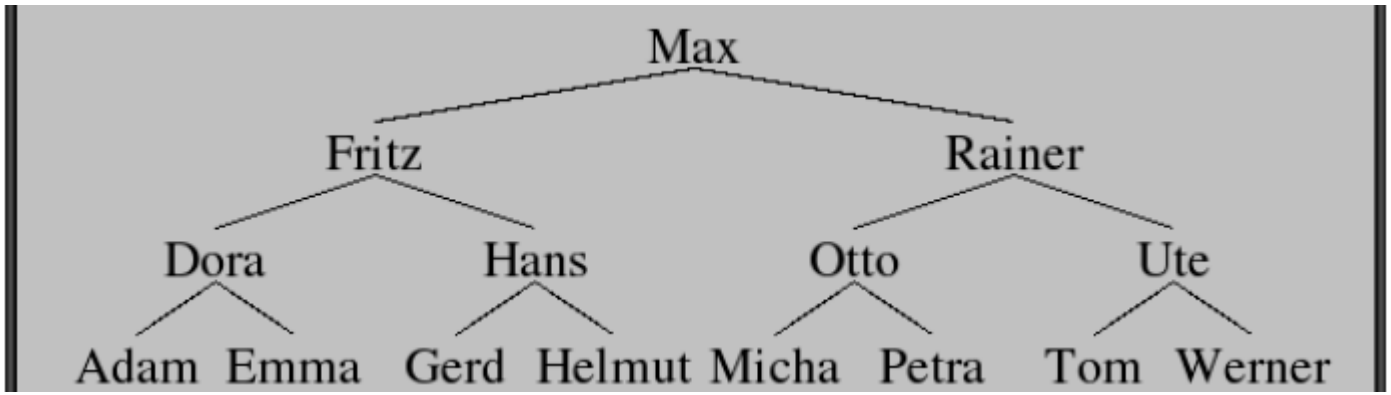
Höhe: 4, Anzahl: 9, Blattanzahl: 3, kein voller Binbaum

Balancierte Binärbäume

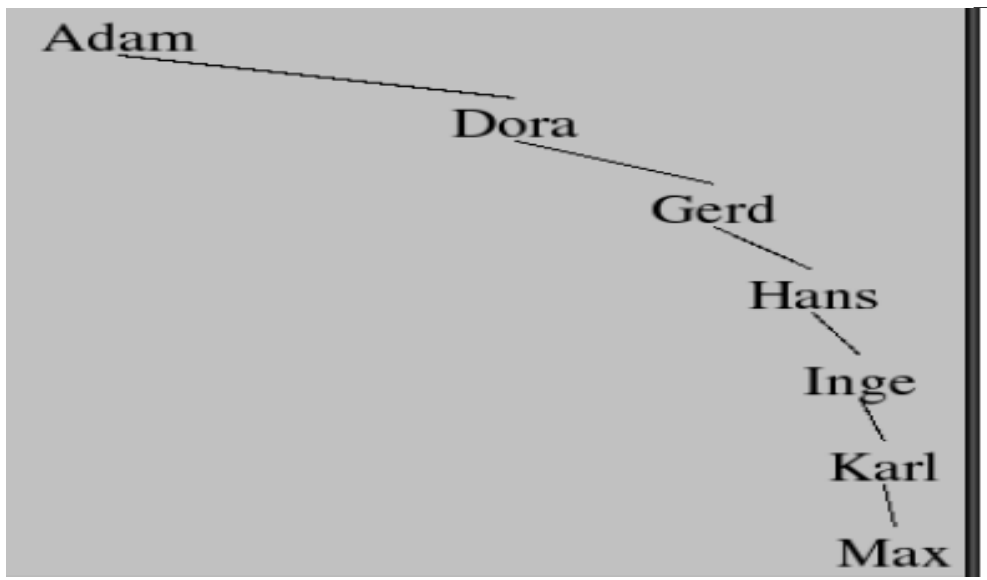
Bei einem balancierten Baum besitzen alle Blattknoten in etwa die gleiche Tiefe, was bedeutet, dass der Binärbaum bei n Knoten eine Höhe von $\log_2(n + 1)$ hat.

→ Bei einem balancierten Binärbaum sind somit maximal $\log_2(n+1)$ Vergleiche (= Höhe des Binärbaums) notwendig, um ein Element zu finden.

Balancierte Binärbäume



Optimal balancierter Binärbaum



Zur Liste degenerierter Binärbaum

Baumrekursion bei Bäumen mit mehr als zwei Zweigen

Rekursion bei L-Systemen

Hat man z. B. das Alphabet {a,b} und die Produktionsregeln:

(1) a → ab

(2) b → a

lassen sich ausgehend vom Grundwort (Axiom) a durch Anwenden der Produktionsregel folg. Zeichenketten ableiten:

a → ab → aba → abaab → abaababa → abaababaabaab

Baumrekursion bei Bäumen mit mehr als zwei Zweigen

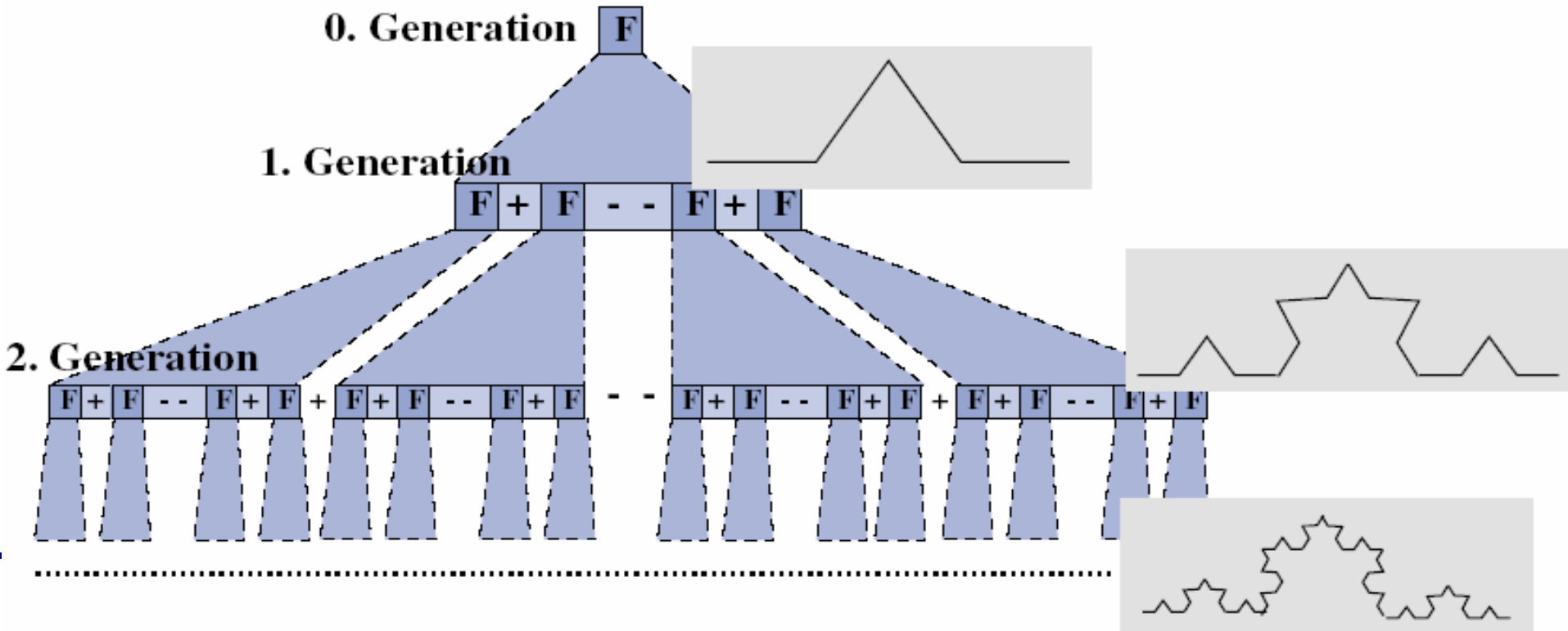
0L-System = Angabe eines Axioms und Produktionsregel

Für eine Seite (_/_) der Koch-Kurve z.B.:

F → F + F − − F + F

+ → +

− → −



Baumrekursion bei Bäumen mit mehr als zwei Zweigen

Notwendige Symbole für 0L-Systeme

F	um bestimmte Länge in aktueller Richtung eine Linie zeichnen
f	um bestimmte Länge in aktueller Richtung vorwärtsbewegen ohne zu zeichnen
+	aktuelle Richtung um vorgegebenen Winkel nach links drehen
−	aktuelle Richtung um vorgegebenen Winkel nach rechts drehen
[	Speichern von aktueller Position und Richtung auf dem Stack
]	Setzen von aktueller Position und Richtung auf zuvor im Stack abgelegte Werte

(x,y) = momentane Koord.
w = aktueller Winkel
→ momentaner Zustand durch Tripel (x,y,w) beschreibbar

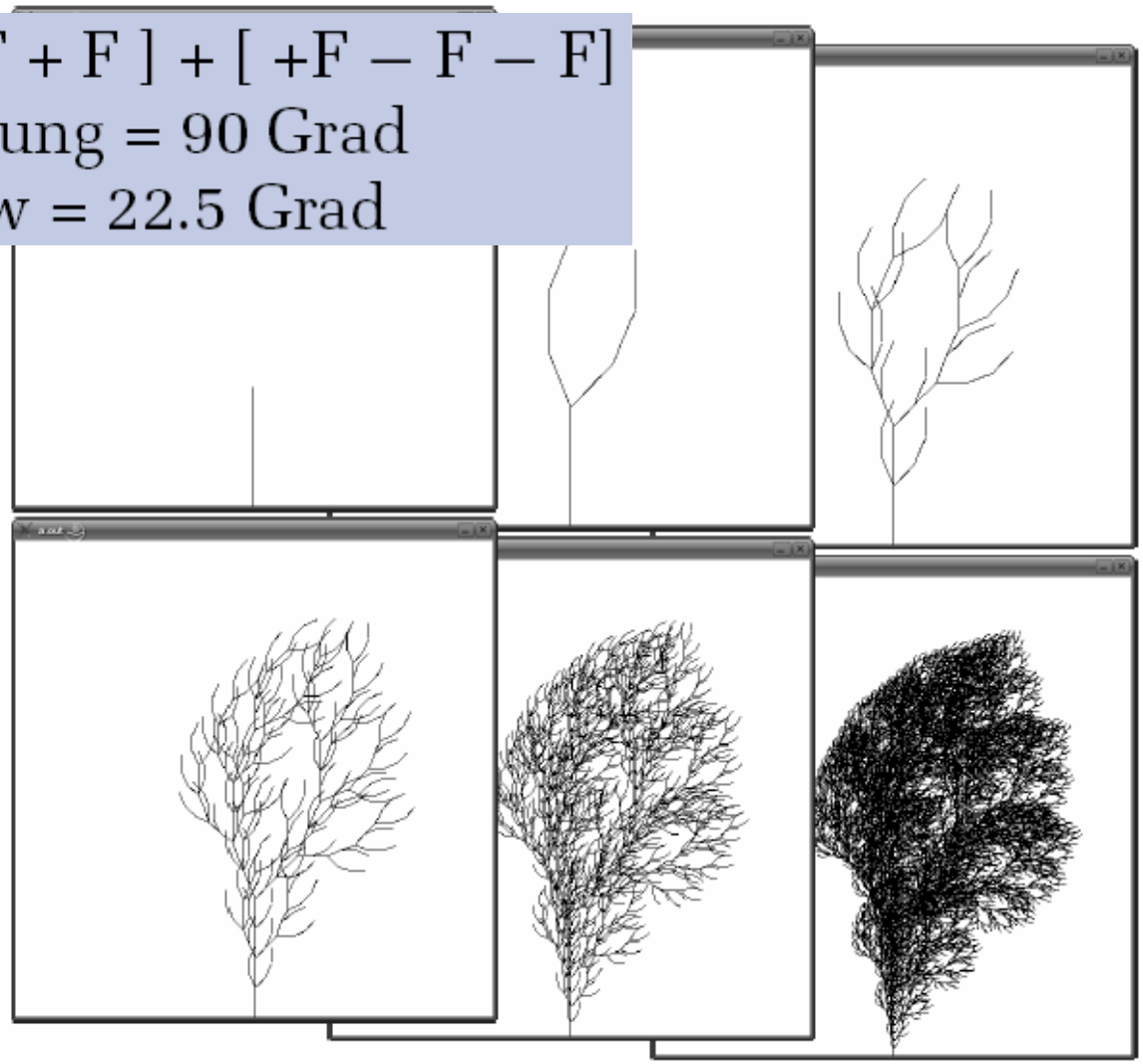
Symbol	Grafik-Aktion
F	$(x + l \cdot \cos(-w), y + l \cdot \sin(-w), w)$
+	$(x, y, w + d)$
−	$(x, y, w - d)$

Baumrekursion bei Bäumen mit mehr als zwei Zweigen

$$F \longrightarrow F\,F - [-F + F + F] + [+F - F - F]$$

Anfangsrichtung = 90 Grad

Drehwinkel w = 22.5 Grad



Baumrekursion bei Bäumen mit mehr als zwei Zweigen

```
/*---------- Makros fuer die Symbole der Produktionsregeln ----
#define   F   f(t-1);                                          /* F */
#define   M   phi=phi-delta;                                   /* - */
#define   P   phi=phi+delta;                                   /* + */
#define   A_  altx=x; alty=y; altphi=phi;                      /* [ */
#define   _Z  x=altx,y=alty,moveto((int)x,(int)y); phi=altphi; /* ] */
```

```
int   main( int argc, char *argv[] )
{
   int      anfWinkel = 90;    /* W
   double drehWinkel = 22.5; /*
   double faktor     =  0.5; /* V
   int      rekTiefe  =  7;    /* R

   delta = (drehWinkel*PI)/180;
   laenge = maxY/4;
```

```
   for (t=1; t <= rekTiefe; t++) {
      cleardevice(WHITE);
      phi = (anfWinkel*PI)/180;
      x = maxX/2;
      y = maxY;
      moveto( (int)x, (int)y );
      f(t);
      laenge *= faktor;
      getch();
   }
```

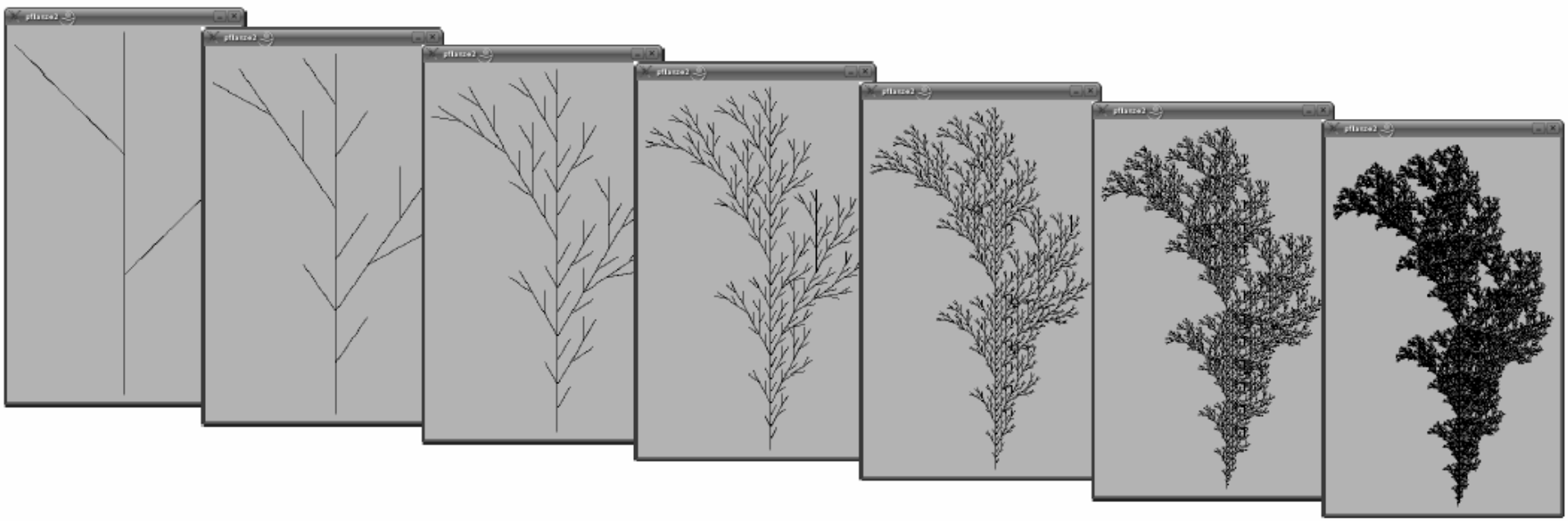
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

F ---> F F – [–F + F + F] + [+F – F – F]
Anfangsrichtung = 90 Grad
Drehwinkel w = 22.5 Grad

```
void f(int t)
{
  double altx, alty, altphi;
   if (t > 1) {
      F F   M  A_ M F P F P F _Z P  A_ P F M F M F _Z
   } else {
      x += laenge*cos(–phi);
      y += laenge*sin(–phi);
      lineto( (int)x, (int)y );
   }
}
```

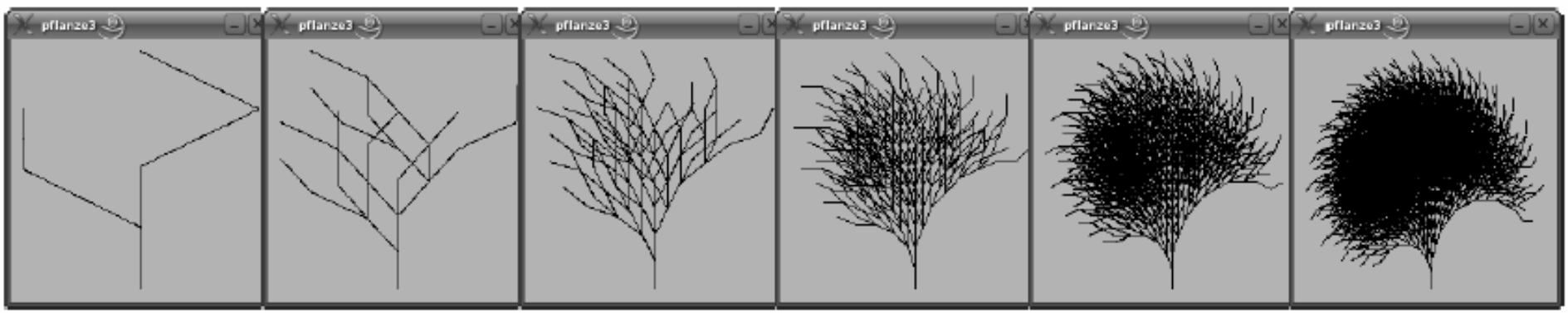
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

F[+F]F[−F][F]	$\Longleftarrow$ *Regel des 0L-Systems*
90	$\Longleftarrow$ *Startwinkel*
25.7	$\Longleftarrow$ *Drehwinkel*
7	$\Longleftarrow$ *Rekursionstiefe*
300	$\Longleftarrow$ *Fensterbreite*
480	$\Longleftarrow$ *Fensterhöhe*



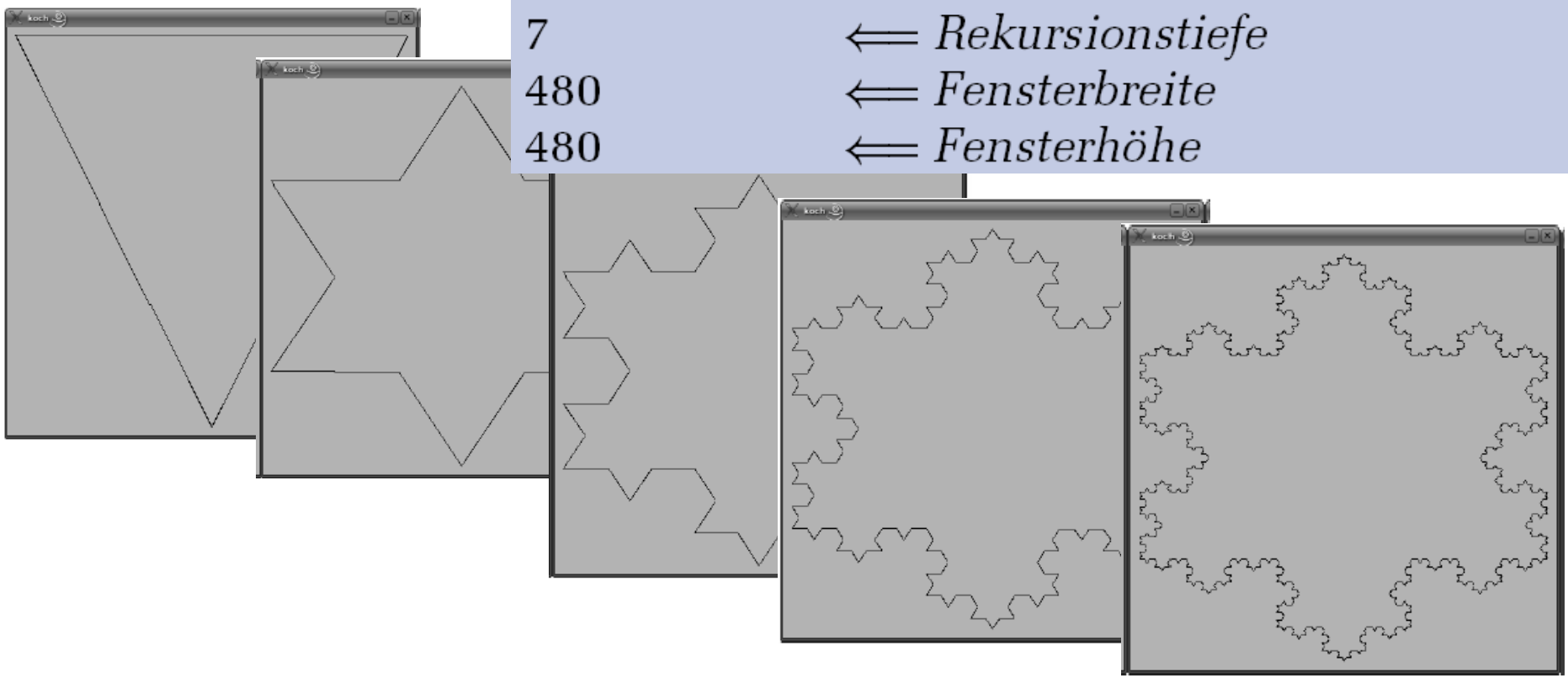
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

```
F[F+F--F][-F+F]
90
22.5
7
200
200
```

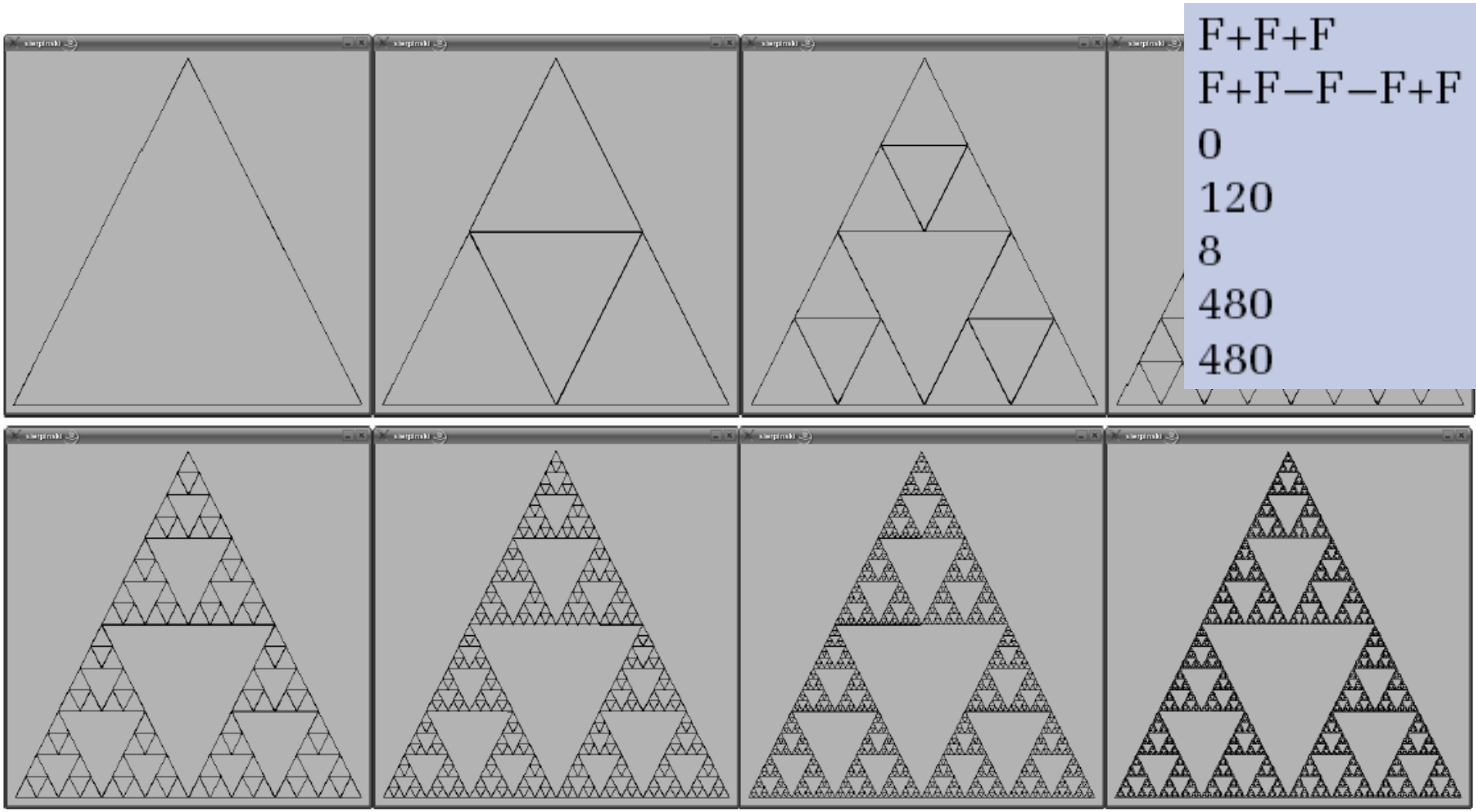


Baumrekursion bei Bäumen mit mehr als zwei Zweigen (Koch-Kurve)

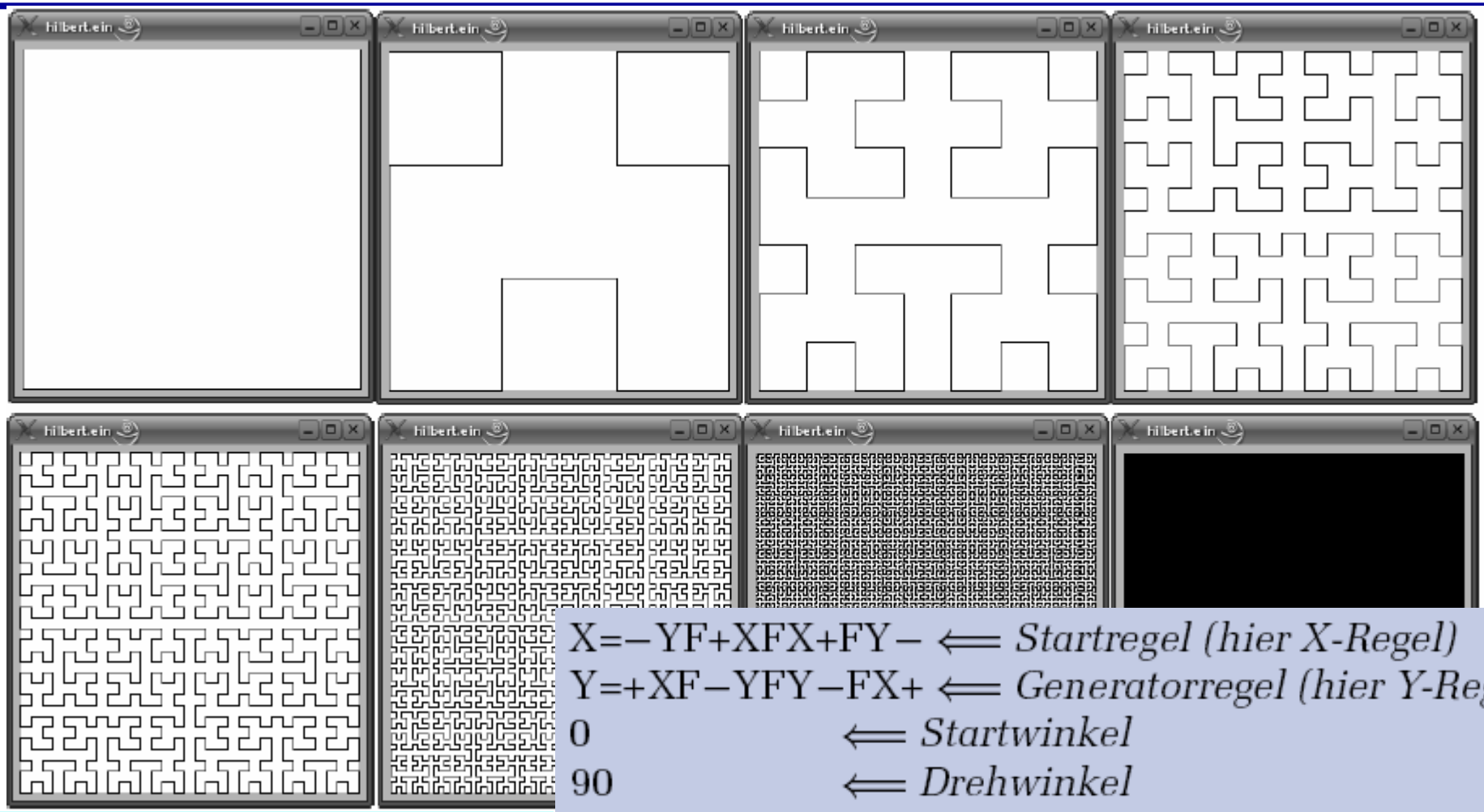
F−−F−−F	$\Longleftarrow$	*Startregel des 0L-Systems*
F+F−−F+F	$\Longleftarrow$	*Generatorregel des 0L-Systems*
0	$\Longleftarrow$	*Startwinkel*
60	$\Longleftarrow$	*Drehwinkel*
7	$\Longleftarrow$	*Rekursionstiefe*
480	$\Longleftarrow$	*Fensterbreite*
480	$\Longleftarrow$	*Fensterhöhe*



Baumrekursion bei Bäumen mit mehr als zwei Zweigen (Sierpinski-Sieb)



Baumrekursion bei Bäumen mit mehr als zwei Zweigen

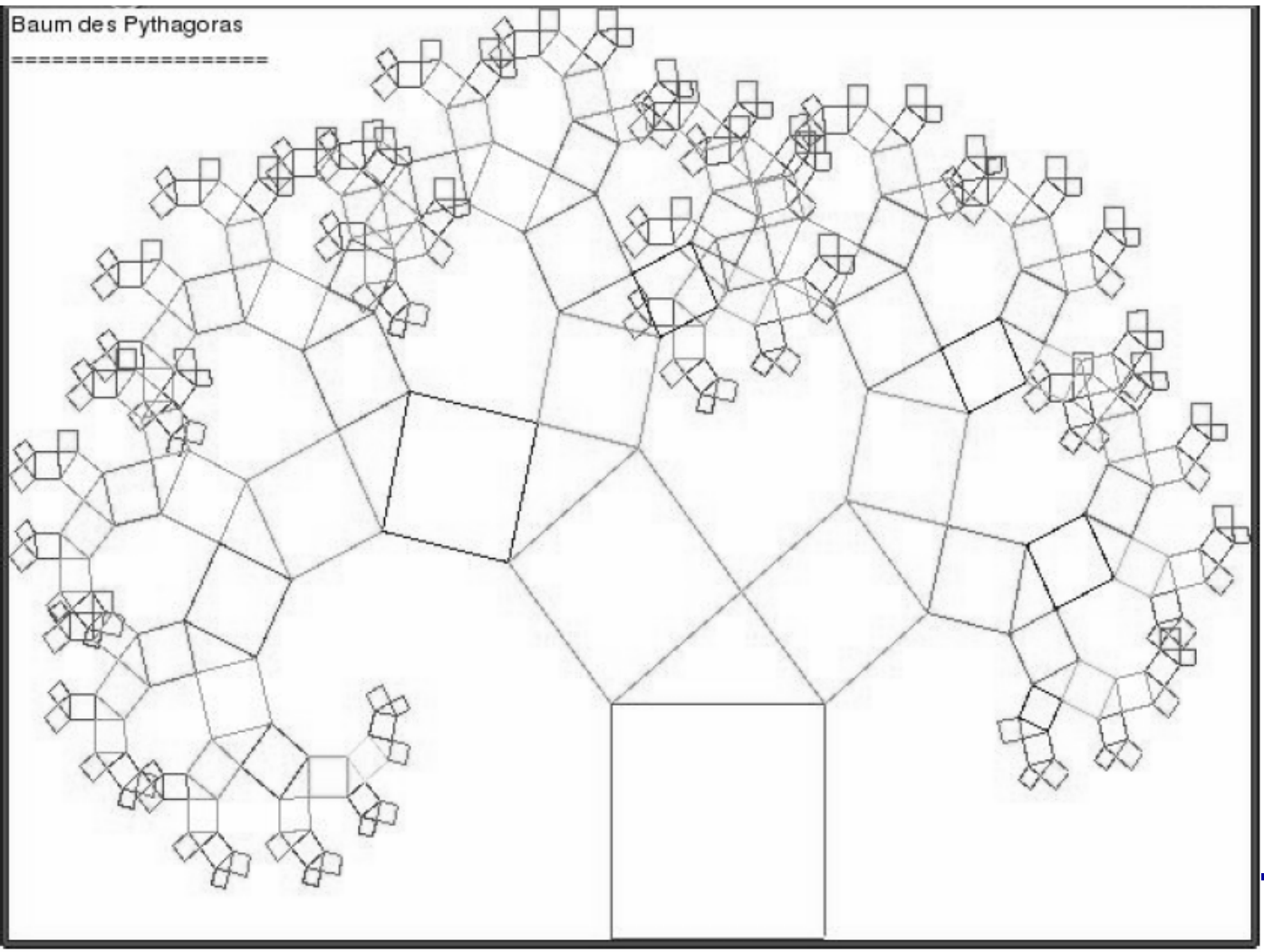


X=−YF+XFX+FY− ⟸ *Startregel (hier X-Regel)*
Y=+XF−YFY−FX+ ⟸ *Generatorregel (hier Y-Regel)*
0 ⟸ *Startwinkel*
90 ⟸ *Drehwinkel*
9 ⟸ *Rekursionstiefe*
275 ⟸ *Fensterbreite*
275 ⟸ *Fensterhöhe*

Flächenfüllende Kurve von Hilbert

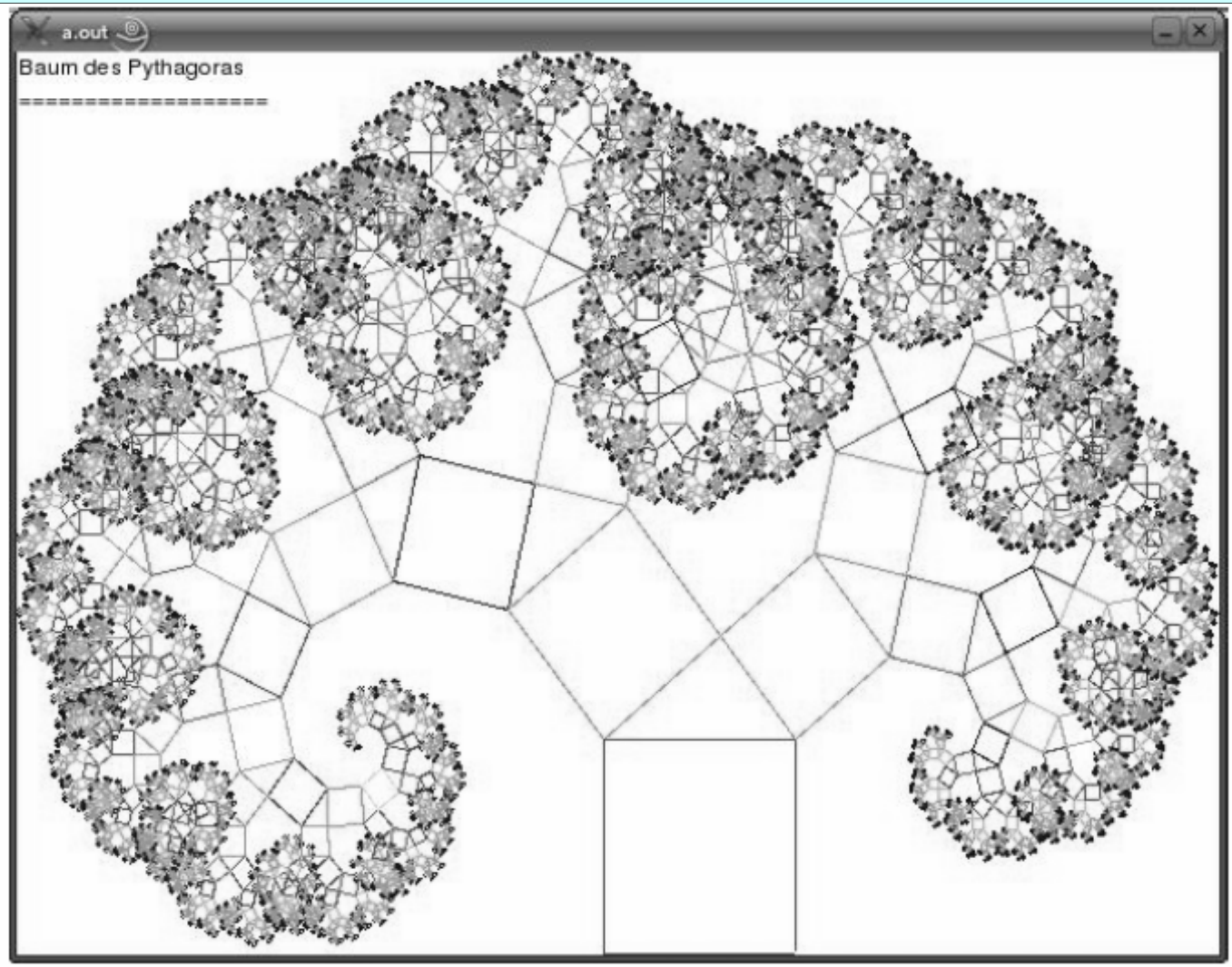
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

Der Baum des Pythagoras (Seitenverhältnis 4:5)



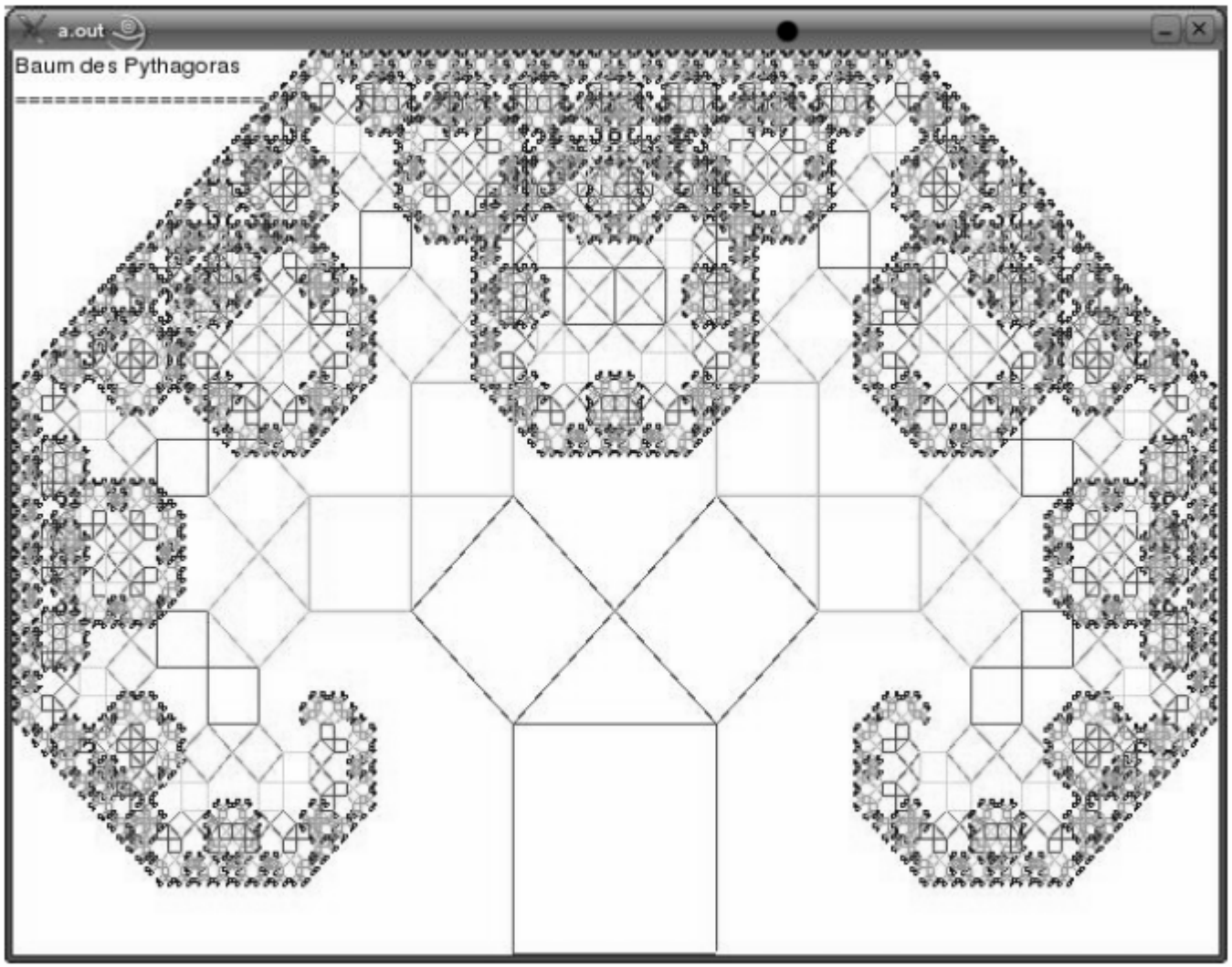
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

Der Baum des Pythagoras (Seitenverhältnis 0.8)



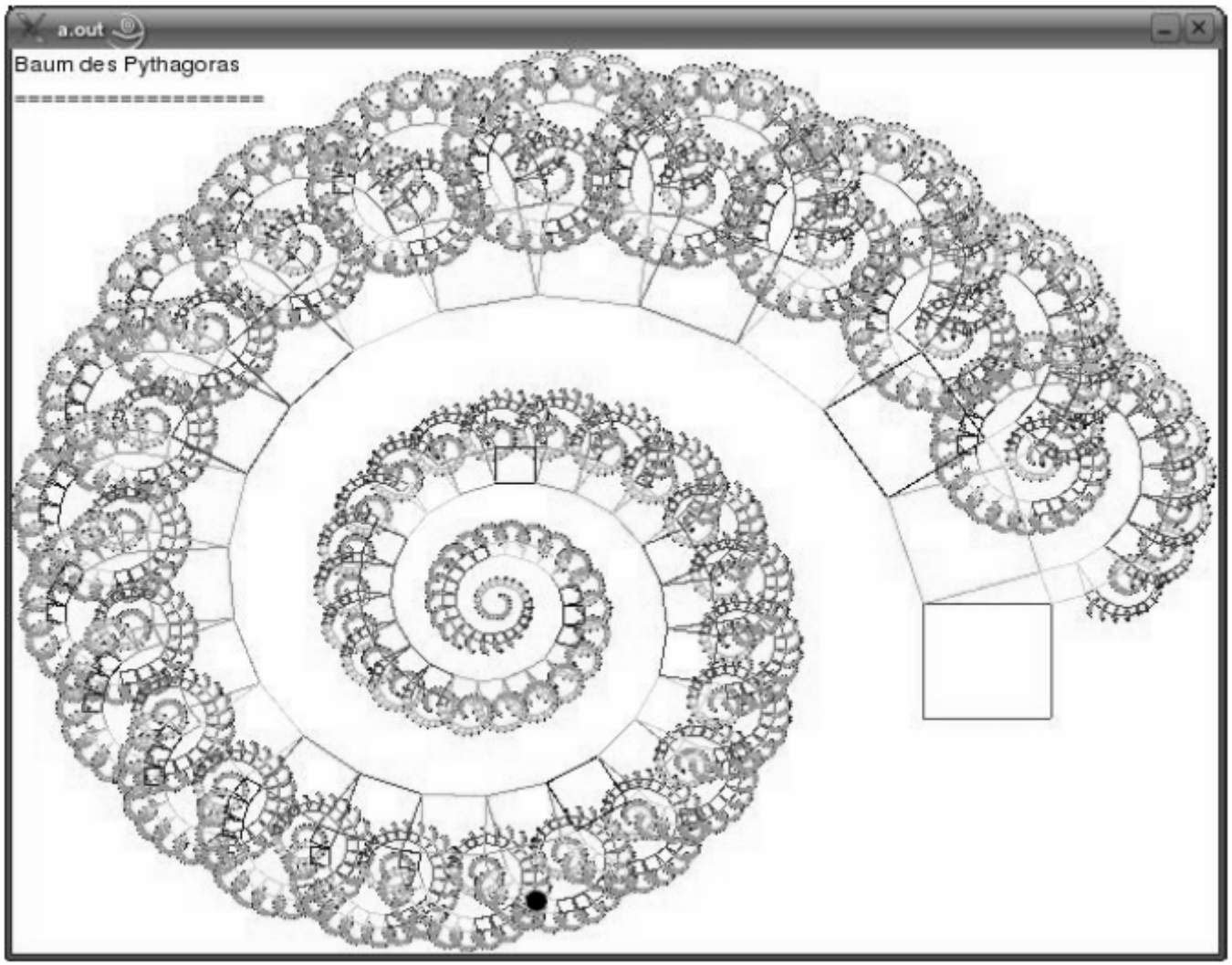
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

Der Baum des Pythagoras (Seitenverhältnis 1)



Baumrekursion bei Bäumen mit mehr als zwei Zweigen

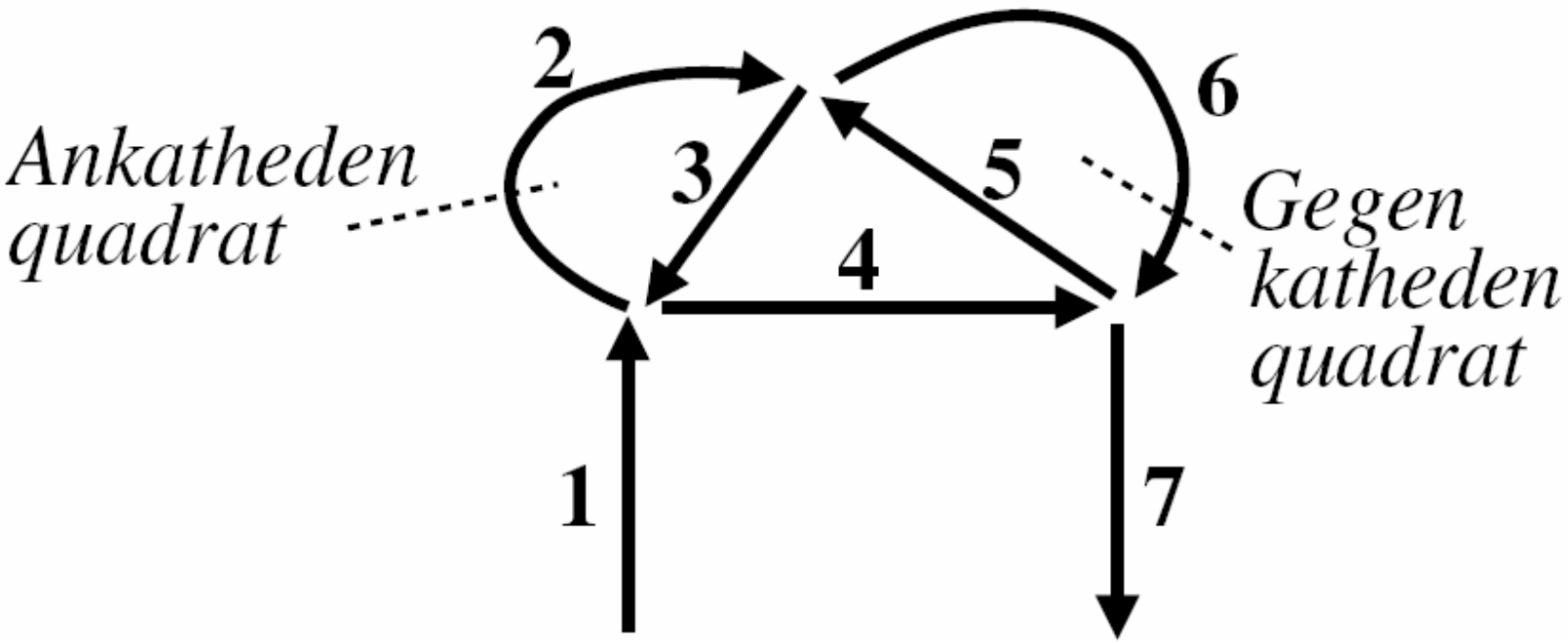
Der Baum des Pythagoras (Seitenverhältnis 0.3)



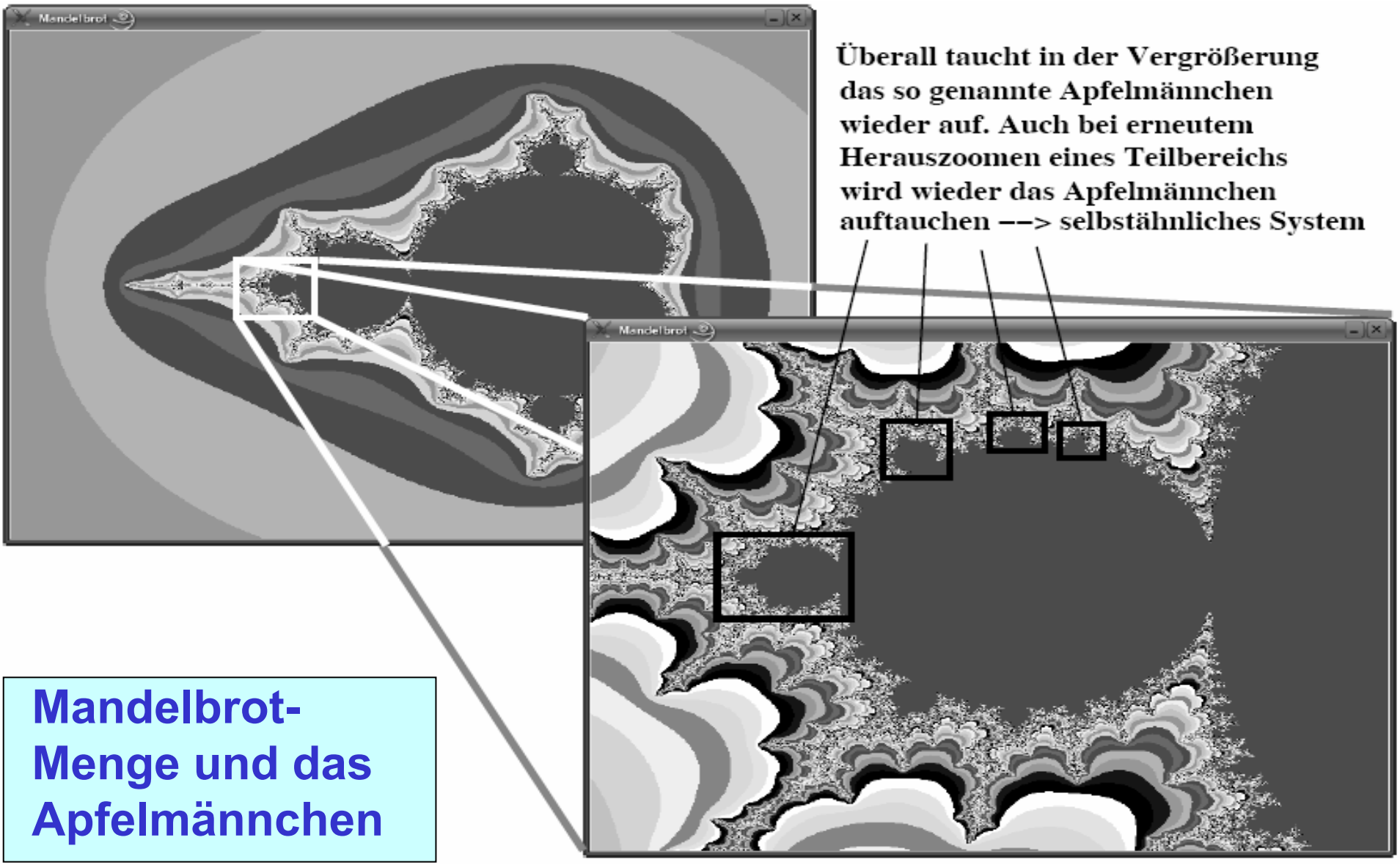
Baumrekursion bei Bäumen mit mehr als zwei Zweigen

Der Baum des Pythagoras

Man kann sich hier überlegen, dass ganzer pythagoräischer Baum in einem Zug ohne Absetzen des Zeichenstifts als so genannter Euler'scher Zyklus erzeugt werden kann.



Baumrekursion bei Bäumen mit mehr als zwei Zweigen



Überall taucht in der Vergrößerung das so genannte Apfelmännchen wieder auf. Auch bei erneutem Herauszoomen eines Teilbereichs wird wieder das Apfelmännchen auftauchen --> selbstähnliches System

Mandelbrot-Menge und das Apfelmännchen