

- *Java-Klassen* – Die Wurzel ist die Klasse *Object*, ein Pfeil von Klasse *A* nach Klasse *B* bedeutet *B extends A*, d.h. *B* ist Unterklasse von *A*.
- *Listen* sind Bäume bei denen jeder Knoten höchstens einen Nachfolger hat.

Bäume stellt man gewöhnlich grafisch dar, indem jeder Knoten durch einen Punkt und jede Kante durch eine Strecke dargestellt wird. Dabei wird ein Vater immer über seinen Söhnen platziert, so dass die Wurzel der höchste Punkt ist.

In Anwendungsprogrammen hat sich auch eine Darstellung eingebürgert, in der die Sohnknoten jeweils in den Zeilen unter dem Vaterknoten und um einen festen Betrag eingerückt dargestellt werden. Die Knoten werden durch ein kleines Quadrat und ihren Namen dargestellt. Klickt man auf ein solches Quadrat, dann verschwindet der entsprechende Unterbaum – er wird *weggefaltet* – und in dem Quadrat erscheint ein „+“. Klickt man es erneut an, so wird der Unterbaum wieder *entfaltet* und das „+“ durch ein „-“ ersetzt. Diese Darstellung findet man in vielen *Datei-Browsern*, siehe Abbildung 1.31.

## 4.8.2 Binärbäume

Im Allgemeinen können Baumknoten mehrere Söhne haben. Ein *Binärbaum* ist dadurch charakterisiert, dass jeder Knoten genau zwei Söhne besitzt. Eine rekursive Definition ist:

Ein *Binärbaum* ist

- *leer*, oder besteht aus
- einem Knoten (*Wurzel*) mit linkem und rechtem Sohn, die jeweils Binärbäume sind.

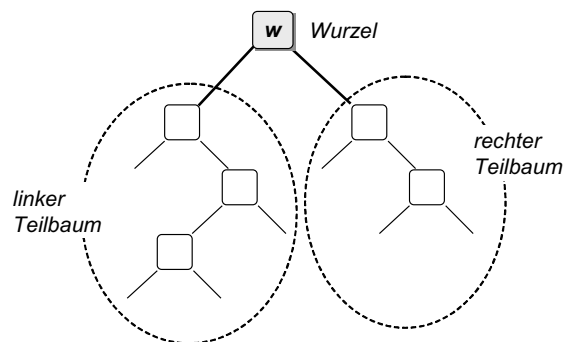


Abb. 4.41: Binärbaum

Binärbäume kann man als 2-dimensionale Verallgemeinerung von Listen ansehen, denn jede Liste kann als spezieller Binärbaum repräsentiert werden, in dem z.B. jeweils der linke Sohn eines Knotens leer ist. Ähnlich wie in den Zellen einer Liste kann man in den Knoten eines Binärbaumes beliebige Informationen speichern. Im Unterschied zu den Listenzellen enthält jeder Knoten zwei Verweise, einen zum linken und einen zum rechten Unterbaum.

Die Definition von Binärbäumen kann man durch Einführen spezieller Blattknoten variieren:

Ein **Binärbaum mit Blättern**

- ist leer oder besteht aus
- einem *Blatt*, oder
- einem *Knoten (Wurzel)* mit linkem und rechtem Sohn, die jeweils Binärbäume sind.

Eine wichtige Anwendung von Bäumen, insbesondere auch von Binärbäumen, ist die Repräsentation arithmetischer Ausdrücke. Innere Knoten enthalten Operatoren, Blätter enthalten Werte oder Variablenamen. Einstellige Operatoren werden als Knoten mit nur einem nicht-leeren Teilbaum repräsentiert. In der Baumdarstellung sind Klammern und Präzedenzregeln überflüssig. Erst wenn wir einen „zweidimensionalen“ Baum eindimensional (als String) darstellen wollen, sind Klammern notwendig.

Beachte, dass das Vorzeichen „-“ und das Quadrieren einstellige Operatoren sind. Im Baum stellen wir sie durch „+/-“ bzw. durch  $()^2$  dar.

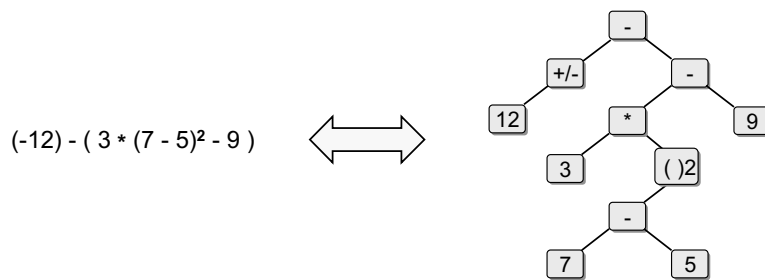


Abb. 4.42: Binärbaum zur Repräsentation eines arithmetischen Ausdrucks

### 4.8.3 Implementierung von Binärbäumen

Da Binärbäume zweidimensionale Verallgemeinerungen von Listen sind, lassen wir uns bei der Implementierung von Binärbäumen auch von der Implementierung verketteter Listen leiten. Statt „Zelle“ sagt man hier „Knoten“ und statt einem Zeiger *next* auf den Rest der Liste hat man hier zwei Zeiger *links* und *rechts* auf die entsprechenden Teilbäume. Wir entscheiden uns für eine generische Implementierung.

```
class Knoten<E>{
    E inhalt;
    Knoten<E> links,rechts;
    Knoten(E el, Knoten<E> li, Knoten<E> re){
        inhalt = el;
        links = li;
        rechts = re;
    }
}
```

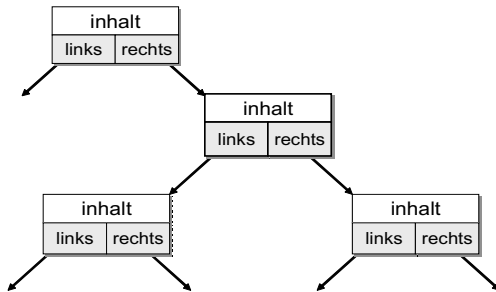


Abb. 4.43: Jeder Knoten ist Wurzel eines Baumes

Ein Baum mit  $n$  Knoten hat  $(n - 1)$  Kanten – denn jede Kante verbindet einen Sohn mit seinem Vater. Bei einer Pointer-Darstellung eines Baumes mit  $n$  Knoten gibt es daher  $2 \times n - (n - 1) = n + 1$  Pointer, die den Wert *null* haben.

Ein Binärbaum „ist“ nichts anderes als ein ausgewählter Knoten, *Wurzel* genannt, ggf. zusammen mit weiteren Feldern und Methoden. Wir sehen die Implementierung von Iteratoren vor:

```
class BinärBaum <E> implements Iterable<E>{
    private Knoten<E> wurzel;
    boolean istLeer() { return wurzel == null; }
    ...
}
```

Es ist recht einfach, aus zwei Bäumen  $b_1$  und  $b_2$  und einem Knoten  $k$  einen neuen Baum mit  $k$  als Wurzel und  $b_1$  als linkem,  $b_2$  als rechtem Teilbaum zu bauen, dies erledigt der Konstruktor

```
wurzel          = k;
wurzel.links    = b1;
wurzel.rechts   = b2;
```

Bisher haben wir aber noch keine Vorstellung davon, was es heißen könnte, ein Element an einer inneren Position in einem Binärbaum einzufügen. Was soll mit den bisherigen Elementen geschehen? Meist sind sie nach irgendeinem Ordnungsprinzip in den Baum eingeordnet, und dies wollen wir erhalten. Das analoge Problem haben wir auch bei dem Löschen eines Knotens. Wie sollen die anderen Knoten aufrücken?

In einer Liste hat jedes Element eine Position. Bei Bäumen entspricht dem sinnvollerweise eine so genannte *Baumadresse*. Sie gibt an, wie der betreffende Knoten von der Wurzel ausgehend zu erreichen ist. Die Adresse *rechts.links.rechts.links.links* beschreibt z.B. in der obigen Abbildung 4.42 den Weg zu dem Blatt mit Inhalt „7“. Natürlich kann man solche Adressen auch binär codieren, hier z.B. durch 10100.

### 4.8.4 Traversierungen

Listen konnten wir auf natürliche Weise von vorne nach hinten durchlaufen – bei Bäumen können wir ähnlich einfach von der Wurzel zu jedem beliebigen Blatt gelangen, sofern wir seine Baumadresse kennen. Um den *Vorgänger* eines Elementes  $e$  in einer Liste zu finden, mussten wir am *Anfang* einsteigen und nach hinten laufen, bis wir zu einer Zelle  $z$  gelangten mit  $z.next=e$ . Um in einem Baum den Vater eines Knotens zu finden, haben wir es schwerer – wir müssen an der *Wurzel* einsteigen und uns in jedem Schritt entscheiden, ob wir nach rechts oder nach links gehen sollen. Spätestens hier erhebt sich die Frage, wie wir systematisch alle Knoten eines Baumes durchlaufen – vornehmer: *traversieren* – können.

Für Listen gab es nur eine sinnvolle Traversierung – vom Anfang zum Ende – und wir haben einen entsprechenden Iterator in der Liste definiert (S. 358). Die Baumklassen wurde oben ebenfalls generisch definiert und wir werden auch die Traversierung von Bäumen mit Hilfe von Iteratoren implementieren. Für die Reihenfolge eines Baumdurchlaufs gibt es mehrere naheliegende Möglichkeiten, von denen wir die wichtigsten, *Preorder*, *Postorder*, *Inorder* und *Levelorder*, besprechen wollen. Analog zu dem Listeniterator werden wir entsprechende Iteratoren *PreOrder*, *PostOrder*, *Inorder* und *LevelOrder* definieren. Wir zeigen jedes Mal das Ergebnis am Beispiel des Baumes aus Abbildung 4.42.

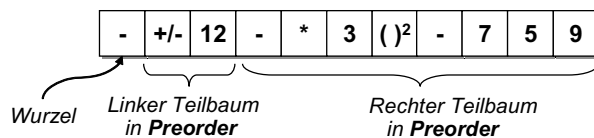
Beim Besuch eines Knotens wollen wir diesen ausgeben. Wir testen also unsere Iteratoren mit einer passenden *toString* Methode, die mit Hilfe des Iterators alle Knoten des Baumes besucht und ausgibt:

```
public String toString() {
    String s = "";
    for (E e : this) s += e+" ";
    return s;
}
```

Die ersten drei Traversierungen sind rekursiv beschrieben – dies liegt aufgrund der induktiven Definition nahe – und lassen sich daher besonders einfach implementieren.

#### **Preorder:**

- Besuche die Wurzel.
- Traversiere den linken Teilbaum in *Preorder*.
- Traversiere den rechten Teilbaum in *Preorder*.



**Abb. 4.44:** *Preorder* Traversierung des Baumes aus Abbildung 4.42

*Inorder* und *Postorder* vertauschen gegenüber *Preorder* die Reihenfolge der Anweisungen.

**Inorder:**

Traversiere den linken Teilbaum in *Inorder*.  
 Besuche die Wurzel.  
 Traversiere den rechten Teilbaum in *Inorder*.

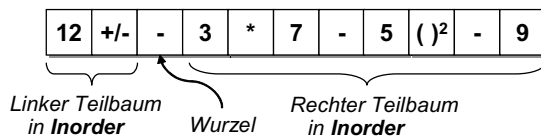


Abb. 4.45: Inorder-Traversierung des Baumes aus Abbildung 4.42

**Postorder:**

Traversiere den linken Teilbaum in *Postorder*.  
 Traversiere den rechten Teilbaum in *Postorder*.  
 Besuche die Wurzel.

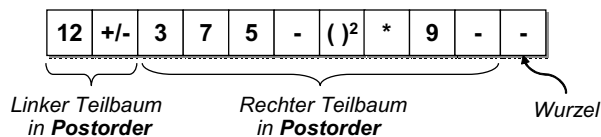


Abb. 4.46: Postorder Traversierung des Baumes aus Abbildung 4.42

Postorder ist die wichtigste der diskutierten Traversierungen. Die Postorder Traversierung liefert nämlich genau die *Postfix-Notation* (siehe S. 347) des arithmetischen Ausdrucks, welcher durch einen Binärbaum repräsentiert wird. Die Umwandlung eines als Zeichenkette repräsentierten Ausdrucks wie z.B.

$$( -12 ) - ( 3 * ( 7 - 5 )^2 - 9 )$$

in einen Baum, und die anschließende Postorder Traversierung zur Generierung von Code für eine Stackmaschine, ist eine der Kernaufgaben jedes Compilers.

Für die oben definierte Klasse *Baum* können wir sofort statische Methoden angeben, die die bisher diskutierten Traversierungen rekursiv durchführen. Für *Preorder* erhalten wir:

```
void preorder() { rekPreorder(wurzel); }

void rekPreorder(Knoten<E> k) {
    if (k != null) {
        tuWas(k.inhalt);
        rekPreorder(k.links);
        rekPreorder(k.rechts);
    }
}
```