Algorithmen und Datenstrukturen

Baumstrukturen

Prof. Justus Piater, Ph.D.

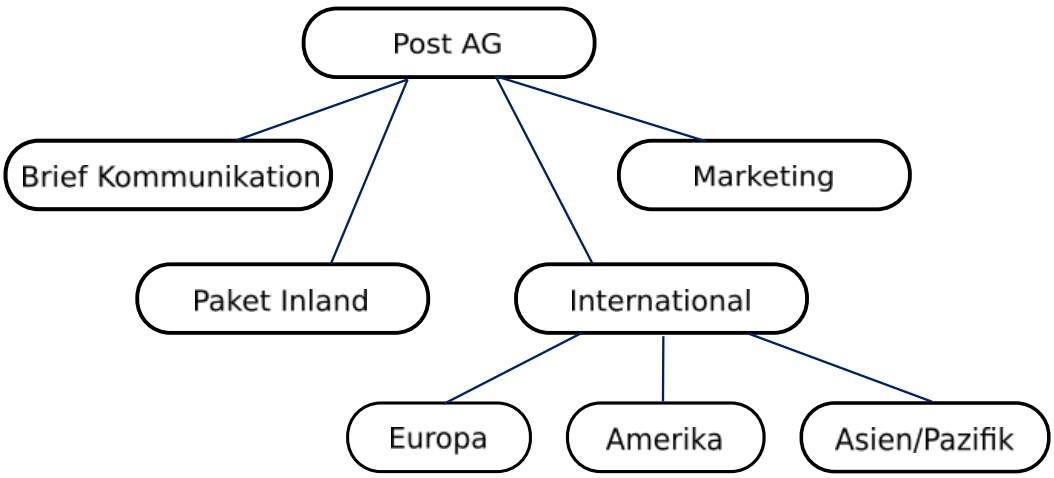
6. April 2022

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

1 Definitionen

Baum [Slide 1]



Baum: Definitionen [Slide 2]

- Ein ***Baum*** T ist eine Menge von ***Knoten*** mit den folgenden Eigenschaften:
  - Wenn T nicht leer ist, dann ist genau ein ausgezeichneter Knoten die ***Wurzel*** und hat keinen Elternknoten.
  - Jeder Knoten v von T außer der Wurzel hat genau einen ***Eltern***-Knoten w; jeder Knoten v von T mit Elternknoten w ist ein ***Kind*** von w.
- Ein Baum T ist entweder leer, oder er besteht aus seiner Wurzel r und einer (möglicherweise leeren) Menge von Unterbäumen, deren Wurzeln die Kinder von r sind.

Knoten [Slide 3]

- Die Kinder desselben Elternteils sind ***Geschwister***.
- Ein Knoten ohne Kinder ist ein ***externer*** Knoten, ein ***Blatt***.
- Ein Knoten mit mindestens einem Kind ist ein ***interner*** Knoten.
- Ein Knoten u ist ein ***Vorfahr*** eines Knotens v, falls $u = v$ oder falls u ein Vorfahr des Elternknotens von v ist.
- Ein Knoten v ist ein ***Nachkomme*** eines Knotens u, falls u ein Vorfahr von v ist.

Kanten und Pfade [Slide 4]

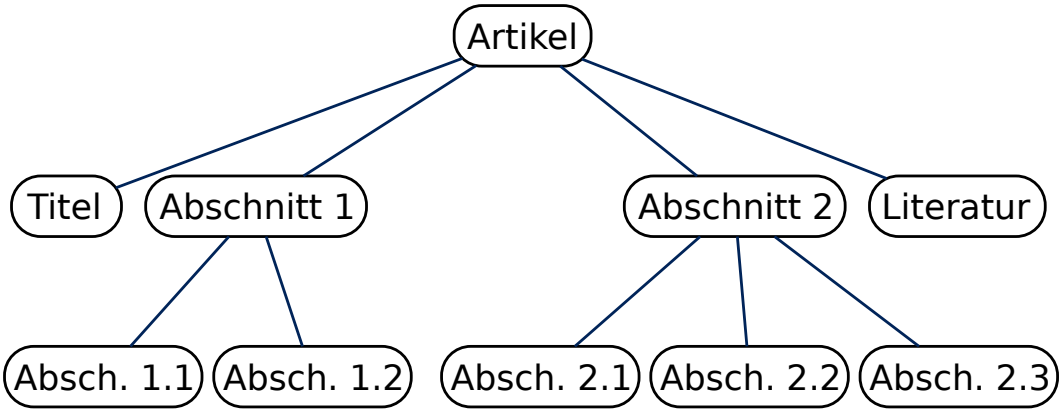
- Eine ***Kante*** ist ein Knotenpaar (u, v), so dass u der Elternknoten von v ist, oder umgekehrt.
- Ein ***Pfad*** ist eine Sequenz von Knoten, so dass alle aufeinander folgenden Knotenpaare jeweils eine Kante bilden.

Geordneter Baum [Slide 5]

Ein Baum ist *geordnet*, falls die Kinder jedes seiner Knoten jeweils eine Sequenz bilden.

… im Gegensatz zu einer Menge.

Wichtig
geordnet $\neq$ *sortiert*!



2 ADT und Methoden

ADT: Baum [Slide 6]

```
root()          // Returns the position of the root of the tree
                // (or null if empty).
parent(p)       // Returns the position of the parent of position p
                // (or null if p is the root).
children(p)     // Returns an iterable collection containing the
                // children of position p (if any).
numChildren(p)  // Returns the number of children of position p.

// Query Methods:
isInternal(p)  // Returns true if position p has at least one child.
isExternal(p)  // Returns true if position p does not have any children.
isRoot(p)      // Returns true if position p is the root of the tree.

// General Collection Methods:
size()
isEmpty()
iterator()     // Returns an iterator for all elements in the tree
positions()    // Returns an iterable collection of all positions of the tree.
```

Änderungsmethoden von Bäumen besprechen wir später.

> Ein Baum ist ein positionsbasierter abstrakter Datentyp, unser zweiter nach der positionsbasierten Liste. In der positionsbasierten Liste verweist eine Position (neben ihrem Datenelement) auf ihren Vorgänger und ihren Nachfolger. In einem Baum entspricht eine Position einem Knoten, und verweist auf seine Eltern- und Kindknoten, soweit vorhanden.

Der Baum ist ein extrem wichtiges Konzept. Er bildet die Grundlage verschiedener Datenstrukturen und Algorithmen wie zu Beispiel Heaps und Suchbäume. Den Heap haben wir bereits kurz erwähnt als geeignete Datenstruktur für Vorrangwarteschlangen. Suchbäume eignen sich als Datenstrukturen für den abstrakten Datentyp *Sortierte Zuordnungstabelle*. Diese alle werden wir im Laufe dieser Lehrveranstaltung noch genauer betrachten.

Je nach Anwendung unterscheiden sich Bäume insbesondere in ihren Änderungsmethoden. Daher schauen wir uns zunächst nur Abfragemethoden an.

Die Methode `root()` liefert die Position des Wurzelknotens zurück, oder Null, falls der Baum leer ist. `parent(p)` liefert die Position des Elternknotens von `p` zurück, oder Null, falls `p` die Wurzel ist.

`numChildren(p)` liefert die Anzahl der Kindknoten des Knotens `p`.

Iteratoren iterieren per Konvention immer über die *Elemente* eines Containers, nicht über eventuelle Positionen. Hierfür stellt der abstrakte Datentyp Baum seine Methode `iterator()` bereit, wie alle iterierbaren Container.

Wenn wir über Positionen iterieren möchten, dann holen wir uns einen Container, der diese Positionen enthält. Dies tut hier für uns die Methode `children(p)` für die Kinder der Position `p`, und die Methode `positions()` für alle Knoten des Baumes.

Neben den allgemeinen Methoden `size()` und `isEmpty()` haben wir ansonsten noch drei Methoden, die einen gegebenen Koten `p` auf bestimmte Eigenschaften testen, nämlich `isInternal()`, `isExternal(p)` und `isRoot()`. Ein Knoten ist immer entweder intern oder extern. Auch der Wurzelknoten kann intern oder extern sein.

Java-Interface [Slide 7]

```
import java.util.Iterator;

public interface Tree<E> extends Iterable<E> {
  Position<E> root();
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
    throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;

  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;

  int size();
  boolean isEmpty();
  Iterator<E> iterator();
  Iterable<Position<E>> positions();
}
```

Tiefe eines Knotens [Slide 8]

Die Anzahl der Vorfahren von v, ausgenommen v selbst.

```
Algorithm depth(T, v):
  if T.isRoot(v) then
    return 0
  else
    return 1 + depth(T, T.parent(v))
```

Laufzeit?

$O(d_v)$, schlimmstenfalls $O(n)$

> Wir schauen uns nun zwei wichtige Konzepte an, die ***Tiefe eines Knotens*** und die ***Höhe eines Baums***. Beide sind von großer Bedeutung für die Analyse von Bäumen und damit für die Entwicklung effizienter Algorithmen für Bäume.
>
> Die ***Tiefe eines Knotens*** v ist die Anzahl der Vorfahren von v, ausgenommen v selbst. Schlagen Sie bei Bedarf noch einmal unsere Definition des Vorfahrs eines Knotens nach: Jeder Knoten gehört zu seinen eigenen Vorfahren.
>
> Wir können die Tiefe eines Knotens v einfach bestimmen, indem wir sukzessive zum Elternknoten gehen. Hier sehen wir eine rekursive Formulierung dieser Idee: Die Tiefe von v ist um 1 größer als die Tiefe des Elternknotens von v. Der Rekursionsanfang ergibt sich aus der Definition: Da v selbst bei seiner Tiefe nicht mitgezählt wird, muss die Tiefe der Wurzel 0 sein.
>
> Was ist die asymptotische Laufzeit dieses Algorithmus? Da er sich selbst genau einmal rekursiv aufruft, bildet der Aufrufbaum eine lineare Liste; die Länge dieser Liste ist offensichtlich O(Tiefe von v). Damit ist die Gesamtlaufzeit von `depth(T, v)` ebenfalls O(Tiefe von v) unter der Annahme, dass die Laufzeiten von `isRoot()` und `parent()` beide O(1) sind. Um diese konstanten Laufzeiten sicherzustellen, müssen wir eine geeignete Datenstruktur für unseren Baum vorsehen.
>
> Schlimmstenfalls ist der Baum zu einer linearen Liste degeneriert. In diesem Fall ist die asymptotische Laufzeit von `depth()` $O(n)$, wobei n die Gesamtzahl der Knoten des Baums ist.

Höhe eines Baumes [Slide 9]

Die Höhe eines Baumes ist die maximale Tiefe eines Knotens.

```
Algorithm height(T):
  h ← 0
  foreach v ∈ T.positions() do
    if T.isExternal(v) then
      h ←max(h, depth(T, v))
  return h
```

Laufzeit?

$O\left(\sum_v d_v\right)$, schlimmstenfalls $\Omega(n^2)$ (siehe Aufgabe C-8.27).

Beispiel eines schlimmsten Falls?

> Die ***Höhe eines Baums*** ist die maximale Tiefe seiner Knoten. Diese Definition können wir unmittelbar in einen Algorithmus zu seiner Berechnung übersetzen: Wir iterieren über die Knoten des Baums, und für jeden externen Knoten bestimmen wir die Tiefe mittels unseres Algorithmus `depth()`. Die maximale auf diese Weise ermittelte Tiefe eines Knotens ist dann definitionsgemäß die Höhe des Baums.

> Dieser Algorithmus riecht jedoch ineffizient: Da sich Bäume nach unten hin verzweigen, führen umgekehrt ihre Wege zur Wurzel zusammen. Folglich werden die Berechnungen von `depth()` für viele verschiedene externe Knoten dieselben Pfade beschreiten.
>
> Tatsächlich ist die Laufzeit dieses Algorithmus im schlimmsten Fall $\Omega(n^2)$.
>
> Meine Herausforderung für Sie: Beschreiben Sie einen Baum, für den dieser Algorithmus diese maximale, quadratische Laufzeit erreicht!
>
> Uns stellt sich aber natürlich die Frage, ob es nicht auch schneller geht.

Schnellere Berechnung der Höhe [Slide 10]

Höhe eines Knotens:

- Ist v extern, dann ist seine Höhe 0.
- Andernfalls ist seine Höhe 1 plus die maximale Höhe seiner Kinder.

Dies ist nichts weiter als die Definition der Höhe eines *Baumes* angewandt auf den *Unterbaum* mit Wurzel v.

Die Höhe eines Baumes ist die Höhe seiner Wurzel.

```
Algorithm height(T, v):
  h ← 0
  foreach w ∈ T.children(v) do
    h ←max(h, 1 + height(T, w))
  return h
```

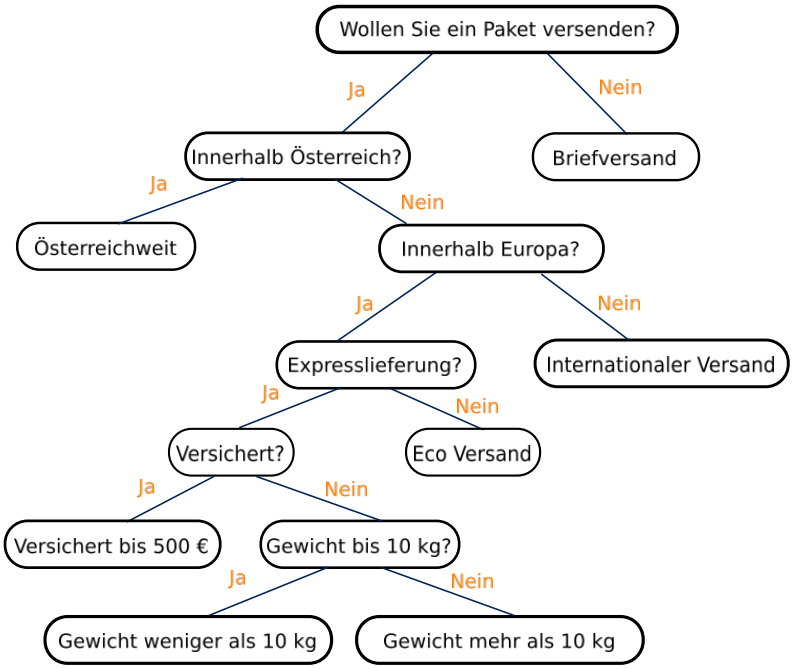
Laufzeit?

$O(n)$

> In der Tat geht es schneller. Hierzu müssen wir die Berechnung vom Kopf auf die Füße stellen. Statt die Tiefen der Blätter von unten her bis ganz zur Wurzel zu berechnen, berechnen wir die Höhen der einzelnen Unterbäume. Der Vorteil ist, dass wir für jeden Knoten die Höhe seines Unterbaums einfach aus den Höhen der Unterbäume seiner Kinder berechnen können, ohne auch nur einen Schritt Richtung Blätter zu tun.
>
> Um uns die Beschreibung etwas zu vereinfachen, definieren wir hier die ***Höhe eines Knotens*** als die Höhe des Unterbaums, dessen Wurzel dieser Knoten ist.
>
> Hier ist eine sehr einfache, elegante, rekursive Formulierung: Die Höhe eines externen Knotens ist immer 0. Die Höhe eines internen Knotens ist immer um 1 größer als die maximale Höhe seiner Kinder.
>
> Diese rekursive Formulierung lässt sich wieder trivial in einen rekursiven Algorithmus übersetzen. Er folgt exakt dieser Formulierung und bedarf kaum einer Erklärung. Um die Höhe des Unterbaums mit der Wurzel v zu berechnen, bestimmt er rekursiv die maximale Höhe seiner Kinder, und zählt 1 hinzu.
>
> Die asymptotische Laufzeit dieses rekursiven Algorithmus ergibt sich wieder aus dem Aufrufbaum. Wir sehen unmittelbar, dass alle elementaren Operationen in konstanter Zeit ablaufen. Nur die Schleife iteriert über die Kinder von v, und für jedes erfolgt ein rekursiver Aufruf.
>
> Wegen der Baumstruktur ist leicht einsichtig, dass `height(T, v)` auf diese Weise genau einmal für jeden Knoten v im Baum T aufgerufen wird, wenn wir am Anfang `height(T, T.root())` aufrufen. Der Aufrufbaum ist zum Baum `T` isomorph. Daraus folgt, dass die Gesamtlaufzeit $O(n)$ ist.

3 Binärbäume

Entscheidungsbaum [Slide 11]

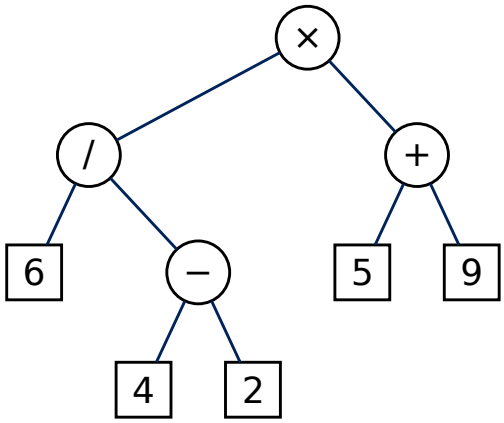


Binärbaum [Slide 12]

Ein ***Binärbaum*** ist ein *geordneter* Baum mit den folgenden Eigenschaften:

- Jeder Knoten hat höchstes zwei Kinder.
- Jedes Kind ist entweder ein ***linkes*** oder ein ***rechtes*** Kind.
- Das rechte Kind folgt dem linken Kind in der Sequenz der Kinder ihres gemeinsamen Elternknotens.

Ein Binärbaum ist ***voll***, falls jeder seiner internen Knoten genau zwei Kinder hat.



> Ein extrem wichtiger Spezialfall eines Baums ist der sogenannte ***Binärbaum***. Ein Binärbaum ist ein *geordneter* Baum, in dem jeder Knoten genau 0, 1 oder 2 Kinder hat.
>
> Ein Baum ist *geordnet*, wenn die Kinder eine *Sequenz* bilden, also eine Reihenfolge über ihnen defininiert ist, im Gegensatz zu einer Menge. Bei Binärbäumen bedeutet dies, dass wir zwischen dem ***linken*** und dem ***rechten*** Kind unterscheiden. Per Definition folgt dem linken das rechte Kind in der Sequenz. Dennoch ist es möglich, dass ein Knoten eines Binärbaums nur ein rechtes Kind hat, aber kein linkes.
>
> Hat jeder Knoten in einem Binärbaum entweder kein Kind oder 2 Kinder, dann handelt es um einen sogenannten ***vollen*** Binärbaum.
>
> Rechts sehen wir ein typisches Beispiel eines vollen Binärbaums. Er repräsentiert einen arithmetischen Ausdruck. Jeder interne Knoten repräsentiert einen binären Operator, das

heißt, einen Operator, der zwei Operanden miteinander verknüpft. In diesem Baum sind diese Operanden die Kinder des Operators. Die Blätter enthalten Zahlenwerte. Damit kann der Wert dieses arithmetischen Ausdrucks auf eindeutige Weise berechnet werden.

Wir könnten auch sogenannte *unäre* Operatoren einführen, z.B. die additive und die multiplikative Negation, also Vorzeichenwechsel und Kehrwert. Unäre Operatoren haben nur einen Operanden. Verwenden wir mindestens einen unären Operator in einem arithmetischen Ausdruck, dann ist der zugehörige Binärbaum nicht mehr *voll*.

ADT: Binärbaum [Slide 13]

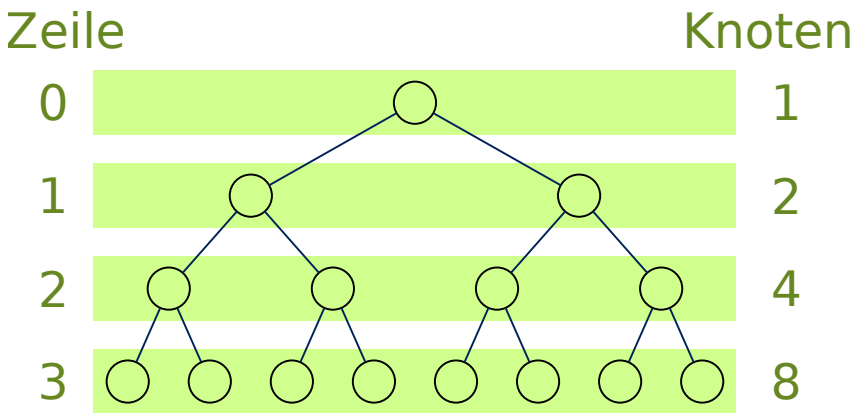
Dieser ADT spezialisiert den Baum-ADT durch die folgenden, zusätzlichen Methoden:

```
left(p)     // Returns the position of the left child of p
            // (or null if p has no left child).

right(p)    // Returns the position of the right child of p
            // (or null if p has no right child).

sibling(p)  // Returns the position of the sibling of p
            // (or null if p has no sibling).
```

Die Beschränkung eines Binärbaums auf eine begrenzte Anzahl Kinder pro Knoten, nämlich 2, legt es nahe, spezialisierte Zugriffsmethoden für die Kinder zu definieren. Anstatt etwas umständlich über die Sequenz der Kinder zu iterieren, definiert der abstrakte Datentyp Binärbaum die Methoden `left(p)` und `right(p)`, die unmittelbar die Position des linken bzw. rechten Kinds des Knotens `p` liefern.

Da ein Knoten maximal zwei Kinder hat, ist es oft praktisch, sich auf das *andere* Kind zu beziehen. Hierzu stellt der abstrakte Datentyp Binärbaum die Methode `sibling(p)` zur Verfügung, die den Geschwisterknoten von `p` zurückgibt, sofern er existiert.

Anzahl der Knoten eines Binärbaums [Slide 14]



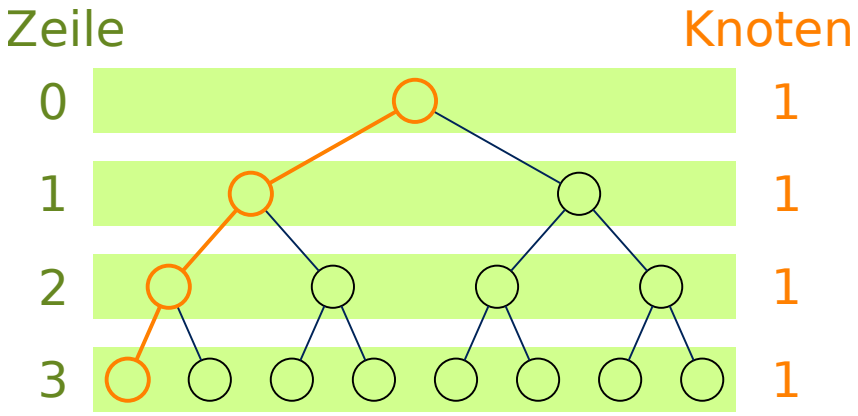
Wenden wir uns nun einigen Eigenschaften von Binärbäumen zu, die für spätere Analysen von Datenstrukturen und Algorithmen wichtig sein werden.

Wir sehen hier die allgemeine Struktur eines Binärbaums unterteilt in ihre sogenannten *Zeilen*. Zeile 0 enthält die Wurzel, Zeile 1 die Kinder der Wurzel, Zeile 2 die Kinder der Kinder der Wurzel, und so weiter. Die Nummer der untersten Zeile des Baums ist gleich der Höhe des Baums.

Da jeder Knoten maximal zwei Kinder hat, sehen wir sofort, dass die Anzahl der Knoten in Zeile l, a für englisch *level*, maximal 2^l betragen kann. Die maximale Anzahl Knoten, die ein Binärbaum der Höhe h enthalten kann, ist also gleich $2^0 + 2^1 + 2^2 + \cdots + 2^h$. Diese Summe ist bekanntermaßen gleich $2^{h+1} - 1$.

Wir sehen auch sofort, dass die maximale Anzahl *interner* Knoten eines Binärbaums der Höhe h gleich der maximalen Gesamtzahl der internen und externen Knoten eines Binärbaums der Höhe $h - 1$ sein muss.

Anzahl der Knoten eines Binärbaums [Slide 15]

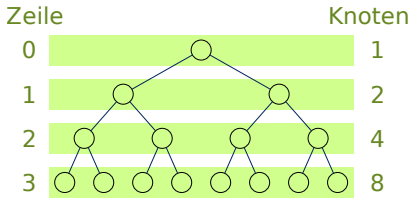


Umgekehrt muss jede Zeile mindestens einen Knoten enthalten, damit diese Zeile überhaupt existiert. Die minimale Anzahl Knoten, die ein Baum der Höhe h enthalten muss, ist also gleich $h + 1$.

Eigenschaften von Binärbäumen [Slide 16]

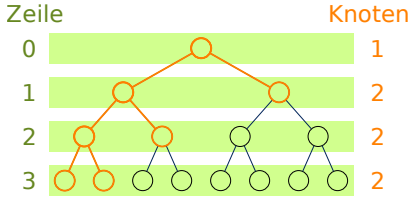
Proposition: Sei T ein nichtleerer Binärbaum, seien n, n_{E} und n_{I} die Anzahl der Knoten bzw. der externen und internen Knoten, und sei h die Höhe von T. Dann hat T die folgenden Eigenschaften:

$$\begin{array}{rcccl} h+1 & \leq & n & \leq & 2^{h+1}-1 \\ 1 & \leq & n_{\mathrm{E}} & \leq & 2^h \\ h & \leq & n_{\mathrm{I}} & \leq & 2^h-1 \\ \log_2(n+1)-1 & \leq & h & \leq & n-1 \end{array}$$



Ist T voll, dann hat er insbesondere folgende Eigenschaften:

$$\begin{array}{rcccl} 2h+1 & \leq & n & \leq & 2^{h+1}-1 \\ h+1 & \leq & n_{\mathrm{E}} & \leq & 2^h \\ \log_2(n+1)-1 & \leq & h & \leq & (n-1)/2 \\ & & n_{\mathrm{E}} & = & n_{\mathrm{I}}+1 \end{array}$$



Beweis der 1. Ungleichung: Man benötigt mindestens 1 Knoten, um einen Baum der Höhe 0 zu erzeugen, 2 Knoten für Höhe 1, usw., allgemein $h+1$ für Höhe h.
Beweis der 2. Ungleichung (ein Binärbaum der Höhe h fasst maximal $n = 2^{h+1}-1$ Knoten): $n = \sum_{i=0}^{h} 2^i = 2^{h+1}-1$; siehe die *Endliche geometrische Reihe* in im Kapitel *Rekursion*.
Weitere Beweise: Aufgabe R-8.7 [Goodrich u. a. 2014]

Fassen wir hier eine ganze Sammlung wichtiger Eigenschaften von Binärbäumen zusammen, die die Anzahl seiner Knoten mit seiner Höhe in Beziehung setzen.

Die ersten 6 Ungleichungen haben wir bereits argumentiert, nämlich: 1. Ein Baum der Höhe h hat mindestens $h+1$ Knoten. 2. Ein Baum der Höhe h hat höchstens $2^{h+1}-1$ Knoten.

3. und 4. Ein Baum der Höhe h hat mindestens einen und höchstens 2^h *externe* Knoten.

5. und 6. Ein Baum der Höhe h hat mindestens h und höchstens 2^h-1 *interne* Knoten.

Die 7. Ungleichung besagt, dass die Höhe h eines Baums mindestens logarithmisch in der Anzahl n seiner Knoten ist. Sie ergibt sich durch Logarithmieren der 2. Ungleichung. Die 8. Ungleichung ergibt sich unmittelbar aus der 1.

Betrachten wir nun insbesondere volle Binärbäume. Sie erinnern sich: In einem vollen Binärbaum hat jeder Knoten genau 0 oder 2 Kinder. Dadurch erhöhen sich die Mindestzahlen der Knoten, aber nicht deren Maximalzahlen.

Die Mindestanzahl der Knoten eines Baums ergibt sich wieder aus einem zu einer linearen Liste degenerierten Baum. Nur müssen wir hier den Baum auffüllen, indem wir jedem internen Knoten in der Liste jeweils ein zweites Kind anhängen. Dadurch ergibt sich die Situation wie rechts illustriert: Jede Zeile enthält genau zwei Knoten, bis auf Zeile 0, die lediglich die Wurzel enthält. Folglich ist die Mindestanzahl der Knoten eines vollen Binärbaums gleich $2h+1$.

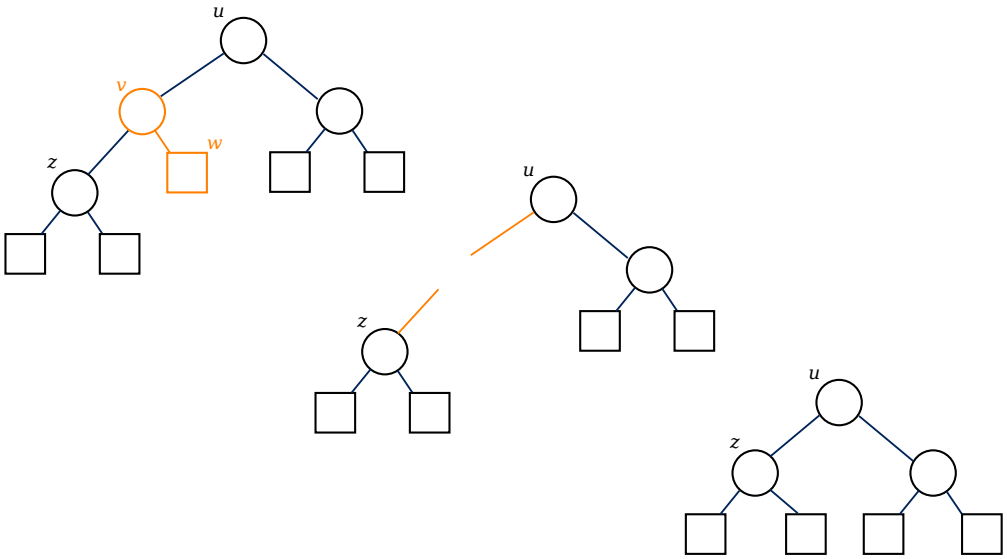
Die Mindestanzahl *externer* Knoten können wir ebenfalls leicht zählen. In Zeile 0 befindet sich kein externer Knoten; dafür hat die letzte Zeile 2. In jeder Zeile dazwischen befindet sich genau ein externer Knoten. Da die Nummer der letzten Zeile gleich der Höhe des Baums ist, ist die minimale Anzahl externer Knoten eines vollen Binärbaums gleich $h+1$.

Übrigens gilt für jeden vollen Binärbaum, dass die Anzahl seiner *externen* Knoten genau

um 1 größer ist die Anzahl seiner *internen* Knoten.

Beweis der Gleichung $n_{\mathbf{E}} = n_{\mathbf{I}} + 1$ [Slide 17]

Wir entfernen iterativ einen externen Knoten und seinen (internen) Elternknoten:



Am Schluss bleibt genau ein (externer) Knoten übrig.

Dies können wir wie folgt belegen: Wählen wir irgendeinen externen Knoten w aus. Entfernen wir diesen gemeinsam mit seinem Elternknoten v, der kraft dieser Eigenschaft ein interner Knoten Sein muss.

Damit ist der Elternknoten u von v eines seiner Kinder beraubt, und der Geschwisterknoten z von w seines Elternknotens. Diese für beide Knoten unerfreuliche Situation entschärfen wir dadurch, dass Knoten u seinen vormaligen Enkel z als Kind adoptiert.

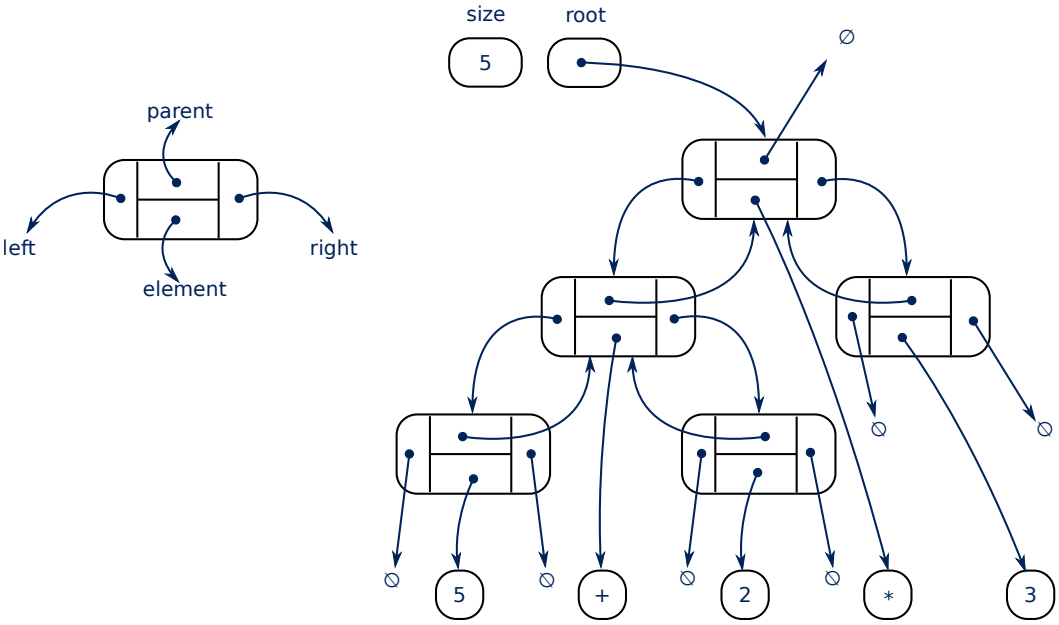
Auf diese Weise haben wir unseren vollen Binärbaum um einen externen Knoten und einen internen Knoten erleichtert. Es handelt sich jedoch immer noch um einen vollen Binärbaum, denn bei keinem Knoten haben wir die Anzahl der Kinder geändert.

Diese Prozedur wiederholen wir, bis es nicht mehr geht. Da unser Baum immer ein voller Binärbaum bleibt, kann dies nur der Fall sein, wenn nur noch ein einziger Knoten übrig ist, ein Blatt.

Damit ist erwiesen, dass in einem vollen Binärbaum die Anzahl Blätter um eins größer ist als die Anzahl der internen Knoten.

4 Datenstrukturen

Verkettete Struktur für Binärbäume [Slide 18]



> Die abstrakten Datentypen Baum und Binärbaum können wir mittels verschiedener Datenstrukturen implementieren, genau wie die anderen abstrakten Datentypen, die wir bisher betrachtet haben. Eine naheliegende und in der Praxis häufig verwendete Datenstruktur ist die verkette Struktur, hier gezeigt für einen Binärbaum.
>
> Eine verkettete Baumstruktur ist eine Generalisierung der doppelt verketteten Liste. Jeder Position entspricht ein Knoten. Jeder Knoten enthält einen Zeiger auf sein Element, sowie Zeiger auf seinen Elternknoten und auf seinen linken und rechten Kindknoten.

Modifikation eines verketteten Binärbaums [Slide 19]

Ausgewählte Methoden, die hier konstante Zeit benötigen:

```
addRoot(e)      // Creates a root for an empty tree,
                // storing e as the element;
                // returns the position of that root;
                // an error occurs if the tree is not empty.

addLeft(p, e)
addRight(p, e)

attach(p, T, U) // Attaches the internal structure of trees T and U
                // as the respective left and right subtrees of leaf p,
                // and resets T and U to empty trees;
                // error if position p is not a leaf.

remove(p)       // Removes the node at position p, replacing it
                // with its child (if any), and returns the element
                // formerly stored at p; error if p has two children.
```

Hier sehen wir beispielhaft typische Änderungsmethoden für Binärbäume. Die genauen Methoden hängen in der Praxis vom Verwendungszweck des Baums ab. Wir werden später verschiedene Anwendungen und zugehörige Methoden kennenlernen.

Die Methode `addRoot()` initialisiert einen leeren Binärbaum mit einem neuen Wurzelknoten, der das übergebene Datenelement hält, und liefert die Position zurück, die den Wurzelknoten repräsentiert.

`addLeft(p, e)` und `addRight(p, e)` hängen einen neuen linken bzw. rechten Kindknoten an den Knoten `p` an. Falls ein solcher Kindknoten bereits existiert, wird ein Fehler signalisiert.

`attach(p, T, U)` hängt die kompletten Bäume `T` und `U` als linken bzw. rechten Unterbaum an `p` an. Ist `p` intern, wird ein Fehler signalisiert.

Wie können wir einen Knoten entfernen? Ein Blatt ist trivial entfernbar. Ein Knoten mit genau einem Kind kann entfernt werden, indem man seinen Elternknoten dieses Kind an eigener statt adoptieren lässt.

Was ist, wenn ein Knoten zwei Kinder hat? Falls er das einzige Kind seines Elternknotens ist, funktioniert auch hier die Adoptionsmethode. Andernfalls würde die Entfernungsaktion dem Elternknoten ein drittes Kind unterjubeln, was in Binärbäumen nicht gern gesehen ist.

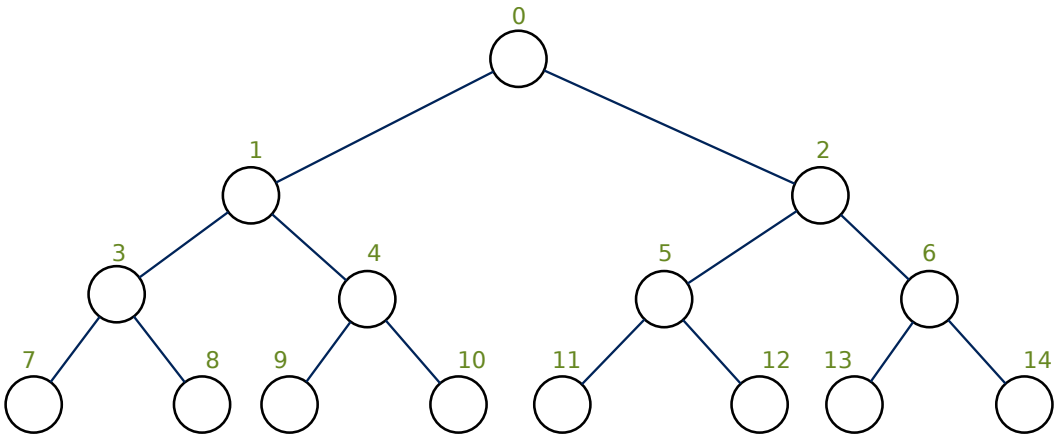
Daher wird konventionellerweise nur die Entfernung von Knoten erlaubt, die selber maximal ein Kind haben. Unsere Methode `remove()` folgt dieser Konvention und signalisiert einen Fehler, falls der zu entfernende Knoten zwei Kinder hat.

Bei einer verketteten Datenstruktur lassen sich alle diese Methoden in konstanter Laufzeit implementieren. Bei keiner von ihnen ist die Größe des Baums von Bedeutung. Alle beziehen sich direkt auf eine Position `p`, und die Operationen beschränken sich auf die unmittelbaren Nachbarn von `p`, die dank der doppelt verketteten Struktur in konstanter Zeit erreichbar sind.

Zeilenweise Nummerierung [Slide 20]

Sei $f(p)$ für jede Position p eines Binärbaums T folgendermaßen definiert:

- Ist p die Wurzel von T, dann ist $f(p) = 0$.
- Ist p das linke Kind von Position q, dann ist $f(p) = 2f(q) + 1$.
- Ist p das rechte Kind von Position q, dann ist $f(p) = 2f(q) + 2$.

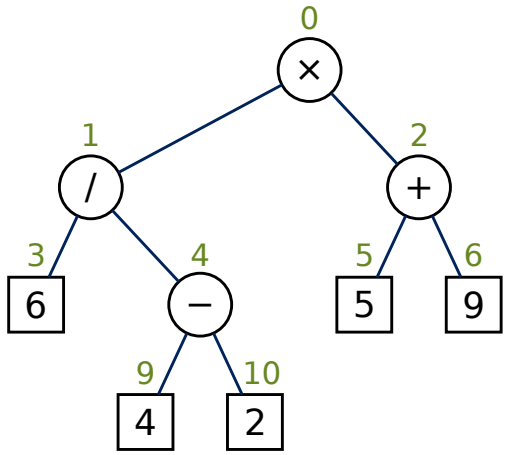


Auch bei Binärbäumen kann eine Array-basierte Datenstruktur interessant sein. Diese beruht auf der sogenannten ***zeilenweisen Nummerierung***, englisch *level numbering*. Die zeilenweise Nummerierung nummeriert – große Überraschung – die Knoten eines

Binärbaums zeilenweise durch, beginnend bei 0, wie hier gezeigt. Hier benennen wir die Nummer eines Knotens p mit $f(p)$.

Damit lassen sich, genau wie bei der verketteten Struktur, Nachbarschaftsverhältnisse in konstanter Zeit berechnen. Da jede Zeile genau doppelt so viele Knoten enthält wie ihre Vorgängerzeile, sind die Nummern der Kinder einer gegebenen Position q $2f(q)+1$ und $2f(q)+2$. Umgekehrt hat der Elternknoten eines Knotens p die Nummer $(f(p)-1)/2$, abgerundet auf eine ganze Zahl.

Array-Repräsentation für Binärbäume [Slide 21]



×	/	+	6	−	5	9			4	2				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Anmerkung
Der hier gezeigte Baum ist voll, aber dies ist für die Array-Repräsentation nicht notwendig.

Ein Binärbaum lässt sich also auf natürliche Weise durch ein Array repräsentieren, indem man jeden Knoten am Index seiner zeilenweisen Nummerierung ablegt.

Als erstes fällt nun auf, dass das Array dort Lücken aufweist, wo im Baum Zeilen unvollständig besetzt sind. Aus unseren Betrachtungen allgemeiner Eigenschaften von Binärbäumen folgt nun, dass die Array-Repräsentation bis zu $2^n - 1$ Zellen bereitstellen muss, um einen Baum mit n Knoten zu repräsentieren. Eine solche exponentielle Speicherverschwendung ist natürlich extrem ineffizient.

Ein weiterer Nachteil ist, dass eine strukturelle Änderung im Allgemeinen das Verschieben vieler Elemente erfordet, im Gegensatz zur verketteten Struktur, mit der wir ganze Unterbäume in konstanter Zeit umhängen können.

Umgekehrt ist die Array-Repräsentation extrem platzeffizient, wenn der Baum gemessen an seiner Höhe sehr viele Knoten besitzt, da die Zeiger auf Eltern und Kinder entfallen. Darüber hinaus vereinfacht ein Array auf natürliche Weise die Speicherverwaltung, was sich durch weniger Fragmentierung und verbesserte Lokalität der Speicherzugriffe auszahlt.

Die Array-Repräsentation bietet sich also an, wenn der Baum im Verhältnis zu seiner Höhe sehr viele Knoten enthält und keine strukturellen Änderungen erforderlich sind.

Quiz [Slide 22]

Eine solche Array-Repräsentation lässt sich für welche Klassen von Bäumen definieren:

A: nur für Binärbäume

B: für Bäume mit bis zu K Kindern pro Knoten, für eine beliebige Konstante K

C: für beliebige Bäume

D: weiß nicht

Eigenschaften der Array-Repräsentation [Slide 23]

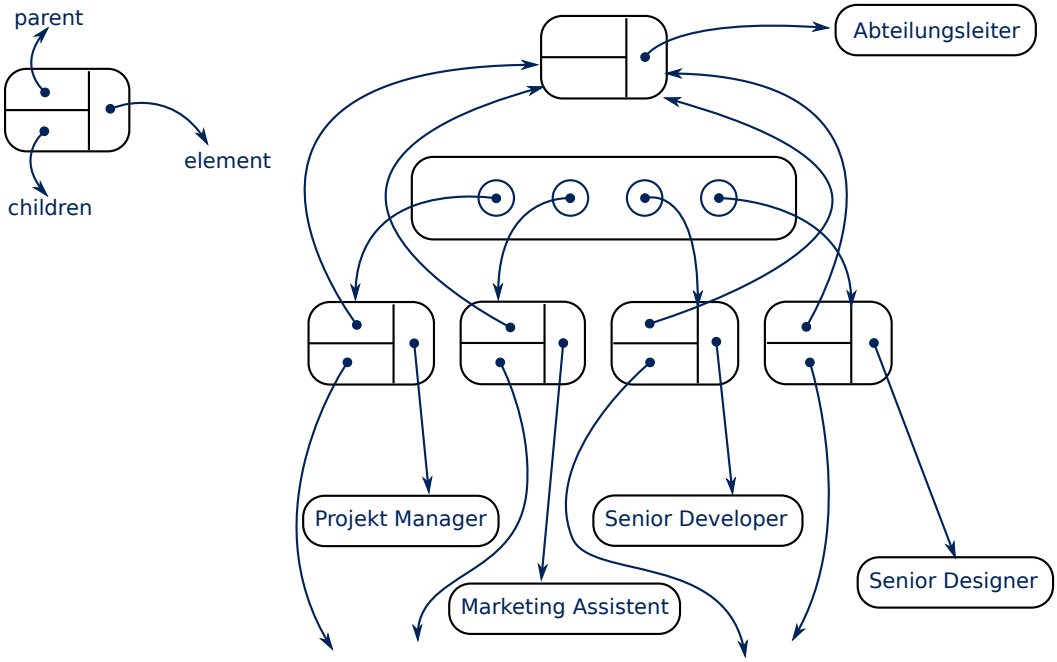
Vorteile:

- Keine Zeiger notwendig:
  - Einfache Implementierung;
    Berechnung von linkem Kind, rechtem Kind, Elternknoten?
  - platzeffizient, falls der Baum (gemessen an seiner Höhe) relativ viele Knoten besizt.

Nachteile:

- Platzbedarf bis zu $N = 2^n - 1$ für n Knoten.
  Aufgabe R-8.12
- Viele strukturelle Modifikationen sind $O(n)$.

Verkettete Struktur für allgemeine Bäume [Slide 24]



Die verkettete Datenstruktur für Binärbäume lässt sich auf einfache Weise für nichtbinäre Bäume generalisieren. Da die Anzahl der Kinder eines Knotens nicht begrenzt ist, sehen wir statt der beiden Zeiger auf das linke und rechte Kind einen Container vor, der seinerseits Zeiger auf die Kinder enthält. Für diesen Container können wir z.B. eine ArrayList verwenden, oder eine andere geeignete Datenstruktur.

Quiz [Slide 25]

Behauptung: Die verkettete Struktur für allgemeine Bäume ist nicht unbedingt notwendig; man kann jeden Baum durch einen Binärbaum repräsentieren.

A: wahr

B: falsch

D: weiß nicht

5 Traversierung

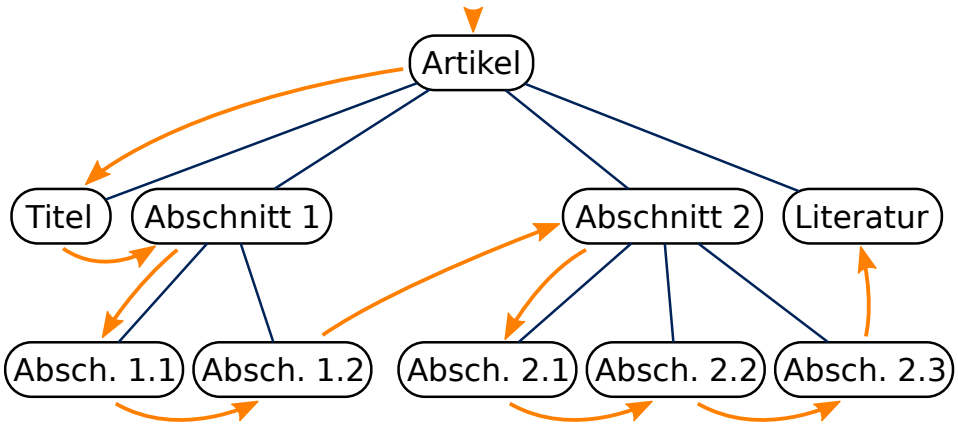
Ein Baum ist ein Container, und über die Elemente eines Containers möchte man oft iterieren. Wir haben kürzlich den abstrakten Datentyp Iterator kennengelernt. Wie könnte man einen Iterator für Bäume definieren? Die erste Frage, die sich stellt, ist die Reihenfolge, in der man die Elemente innerhalb der Baumstruktur besucht. Ein systematisches Abschreiten der Elemente einer Datenstruktur bezeichnet man als *Traversierung*. Bei Bäumen bieten sich verschiedene Traversierungen an.

Präorder-Traversierung [Slide 26]

```
Algorithm preorder(T, v):
  visit v
  foreach w ∈ T.children(v) do
    preorder(T, w)
```

Laufzeit?

$O(n)$



Die Pfeile zeigen die Sequenz der `visit`-Aktionen, nicht den Verlauf der Rekursion.

In der *Präorder*-Traversierung wird jeder Knoten *vor* seinen Kindern besucht. Dieses einfache Prinzip führt unmittelbar zum hier gezeigten Algorithmus. Er wird zu Beginn mit der Wurzel des Baums T aufgerufen. Der als Argument übergebene Knoten v wird

besucht, und *anschließend* ruft sich der Algorithmus für jeden der Kinder des aktuellen Knotens v rekursiv auf.

Ein sogenannter Besuch ist hier ein Platzhalter für das, was mit diesem Knoten bei der Traversierung geschehen soll. In dieser Illustration ist jeder orange Pfeilkopf ein Besuch.

Bei einem Besuch kann z.B. der Name des Knotens auf der Konsole ausgegeben werden, vielleicht ein eingerückter Form abhängig von der Rekursionstiefe, genau wie bei unserem früheren Beispiel mit dem Verzeichnisbaum. In der Tat handelte es sich bei jenem Beispiel um eine Präorder-Traversierung der Verzeichnis-Baumstruktur.

Bei der Präorder-Traversierung ruft jeder Knoten v seine Kinder auf, bevor die Geschwister von v an die Reihe kommen. Die Traversierung geht also zuerst in die Tiefe und erst anschließend in die Breite. Daher gehört die Präorder-Traversierung zu den sogenannten ***Tiefen-Traversierungen***, auf englisch *depth-first traversals*.

Wenn wir die Abschnittsorganisation eines Buchs oder Artikels als Baumstruktur betrachten, dann stellt das sequenzielle Lesen des Schriftstück eine Präorder-Traversierung dieser Baumstruktur dar. Wir lesen den Text unter der Überschrift von Abschnitt 1 *bevor* wir die Unterabschnitte 1.1 und 1.2 lesen.

Auf Abschnitt 1.2 folgt in diesem Beispiel der Abschnitt 2, da bei der Rückkehr der rekursiven Aufrufe die Knoten namens Abschnitt 1 und Artikel nicht noch einmal besucht werden. Der Knoten Abschnitt 2 ist dann das Ziel des dritten rekursiven Aufrufs von `preorder()` vom Wurzelknoten aus.

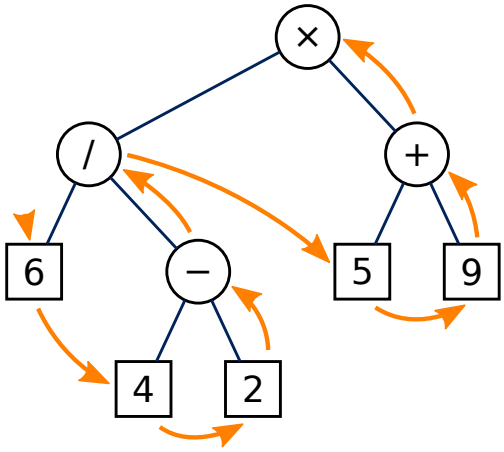
Was ist die Laufzeit dieses `preorder`-Algorithmus? Wir sehen, dass der Code nur `visit` und die rekursiven Aufrufe enthält. Die Kinder eines Knotens können mit gängigen Datenstruktren in einer Zeit proportional zu ihrer Anzahl erreicht werden. Der Aufrufbaum folgt der Baumstruktur. Damit ist die Gesamtlaufzeit O(n $\times$ der Laufzeit von `visit`).

Postorder-Traversierung [Slide 27]

```
Algorithm postorder(T, v):
  foreach w ∈ T.children(v) do
    postorder(T, w)
  visit v
```

Laufzeit?

$O(n)$



Besuche: `6 4 2 - / 5 9 + ×`

Arithmetischer Ausdruck in *Postfix-* (*umgekehrter polnischer*) Notation

- Berechnung mittels eines Stapels: Zahlen werden gepusht, Operatoren poppen ihre Operanden und pushen anschließend das Ergebnis.
- Keine Klammerung notwendig.

Die Postorder-Traversierung ist das Gegenstück zur Präorder-Traversierung. In der Postorder-Traversierung wird ein Knoten *nach* seinen Kindern besucht. Wir sehen dies anschaulich im Algorithmus: Die `visit`-Aktion findet nun nach den rekursiven Aufrufen statt.

Auf diese Weise ist in unserem Beispiel der Knoten mit der 6 der erste besuchte Knoten, denn die Knoten mit der Multiplikation und mit der Division rufen jeweils rekursiv `postorder()` auf ihren Kindern auf, bevor der Besuch `visit` stattfindet.

Der Knoten mit der 6 ist der erste kinderlose Knoten, der durch die rekursiven Aufrufe erreicht wird. Daher ist er der erste *besuchte* Knoten bei der Postorder-Traversierung.

Wie bei der Präorder-Traversierung ruft auch bei der Postorder-Traversierung jeder Knoten seine Kinder auf, bevor seine Geschwister an die Reihe kommen. Die Traversierung geht also ebenfalls zuerst in die Tiefe und erst anschließend in die Breite. Daher ist auch die Postorder-Traversierung eine Tiefen-Traversierung.

Ein natürliches Anwendungsbeispiel ist die Berechnung des Werts eines arithmetischen Ausdrucks. Ein Operator (an der Wurzel eines Unterbaums) kann erst angewendet werden, *nachdem* die Werte beider Operanden (d.h. seiner Unterbäume) berechnet worden sind. Der Wert eines als Baum dargestellten arithmetischen Ausdrucks kann also per Postorder-Traversierung berechnet werden.

Die Ausgabe eines arithmetischen Ausdrucks per Postorder-Traversierung liefert diesen Ausdruck in der sogenannten Postfix-Notation, bei der der Operator auf seine Operanden folgt. Diese Notation hat gegenüber der üblichen Infix-Notation den Vorteil, dass sie viel einfacher syntaktisch zu analysieren ist, und damit der Wert des Ausdrucks auf einfachste Weise berechnet werden kann.

Operator-Präzedenzen wie Punktrechnung-vor-Strichrechnung spielen keine Rolle, und Klammerung existiert nicht. Man muss die Zeichenkette einfach nur von links nach rechts

lesen. Immer, wenn ein Operator angetroffen wird, hat man seine Operanden bereits gelesen.

Damit bietet es sich an, den Wert eines mathematischen Ausrucks in Postfix-Notation mit Hilfe eines Stacks zu berechnen. Man liest die Zeichenkette von links nach rechts. Ist das aktuelle Element eine Zahl, pusht man sie auf den Stack. Ist es ein Operator, poppt man seinen zweiten und dann seinen ersten Operanden vom Stack, verknüpft sie durch den Operator, und pusht das Ergebnis wieder auf den Stack.

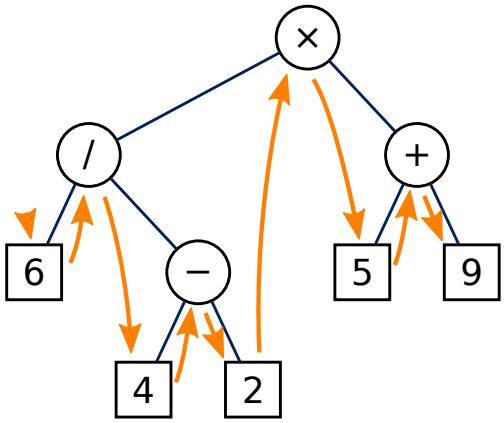
Auf diese Weise findet jeder Operator seine beiden Operanden fertig berechnet auf dem Stack vor.

Die Postfix-Notation ist übrigens auch als *umgekehrte polnische Notation* bekannt, benannt nach dem polnischen Logiker Jan Łukasiewicz. Dieser erfand 1924 die Präfix-Notation, die der Präorder-Traversierung entspricht.

Die Laufzeit der Postorder-Traversierung lässt sich analog zur Präorder-Traversierung argumentieren, und ist ebenfalls $O(n)$ bei einem Baum mit n Knoten.

Inorder-Traversierung von Binärbäumen [Slide 28]

```
Algorithm binaryInorder(T, v):
  if T.hasLeft(v) then
    binaryInorder(T, T.left())
  visit v
  if T.hasRight(v) then
    binaryInorder(T, T.right())
```



Besuche: `6 / 4 - 2 × 5 + 9`

Arithmetischer Ausdruck in *Infix*-Notation (allerdings ohne Klammern)

Bei der Präorder-Traversierung wird jeder Knoten *Vor* seinen Kindern besucht, und bei der Postorder-Traversierung wird jeder Knoten *nach* seinen Kindern besucht. Insbesondere bei Binärbäumen bietet sich eine dritte Variante an, bei der jeder Knoten *zwischen* seinen Kindern besucht wird. Dies nennt sich eine ***Inorder-Traversierung***.

Geben wir die Knoten dieses Baumes während einer Inorder-Traversierung aus, dann erhalten wir den arithmetischen Ausdruck in der üblichen Infix-Notation, allerdings ungeklammert.

Im Gegensatz zur Postfix-Notation müssen bei der Infix-Notation die Präzedenzen der Operatoren berücksichtigt werden. Dies leistet unsere Inorder-Traversierung jedoch nicht.

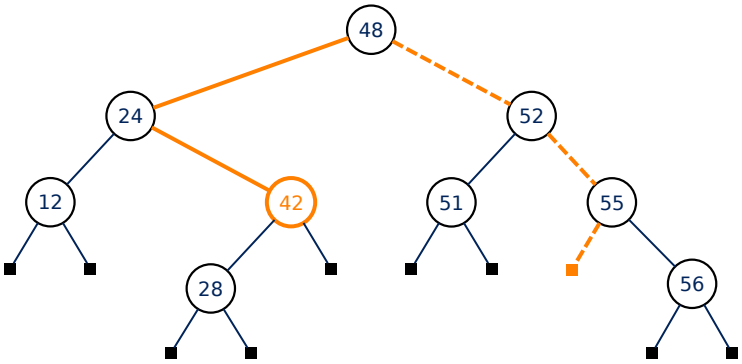
Auch die Inorder-Traversierung ist eine Tiefen-Traversierung und hat ebenfalls eine Laufzeit von $O(n)$.

Binärer Suchbaum [Slide 29]

Definition: Sei S eine Menge mit einer Ordnungsrelation. Ein ***binärer Suchbaum*** ist ein voller Binärbaum, für dessen sämtliche internen Knoten p gilt:

- p speichert ein Element $e(p) \in S$.
- Alle Elemente im linken Unterbaum von p sind kleiner als $e(p)$.
- Alle Elemente im rechten Unterbaum von p sind größer als $e(p)$.

Anmerkung
Eine Inorder-Traversierung besucht die Knoten in aufsteigender Reihenfolge.



Suche in einem binären Suchbaum [Slide 30]

```
Algorithm treeSearch(p, e):
  if p empty then
    return False
  if e < e(p) then
    return treeSearch(left(p), e)
  if e(p) < e then
    return treeSearch(right(p), e)
  return True
```

Laufzeit?

Proportional zur Höhe des Baumes. Es ist also wichtig, die Höhe von Suchbäumen klein zu halten (gemessen an der Anzahl der Knoten)!

Anwendung: Inhaltsverzeichnis [Slide 31]

```
Artikel              Artikel
Titel                  Titel
Abschnitt 1            Abschnitt 1
Abschnitt 1.1            Abschnitt 1.1
Abschnitt 1.2            Abschnitt 1.2
Abschnitt 2            Abschnitt 2
Abschnitt 2.1            Abschnitt 2.1
Abschnitt 2.2            Abschnitt 2.2
Abschnitt 2.3            Abschnitt 2.3
Literatur              Literatur
```

Artikelstruktur

Inhaltsverzeichnis in Java [Slide 32]

```
public static <E> void printPreorder(AbstractTree<E> T) {
  for (Position<E> p : T.preorder())
    System.out.println(p.getElement());
}

public static <E> void
printPreorderIndent(Tree<E> T, Position<E> p, int d) {
  // indent based on d:
  System.out.println(spaces(2*d) + p.getElement());
  for (Position<E> c : T.children(p))
    printPreorderIndent(T, c, d+1); // child depth is d+1
}
```

Anwendung: Automatische Nummerierung [Slide 33]

```
Post AG
1 Brief Kommunikation
2 Paket Inland
3 International
  3.1 Europa
  3.2 Amerika
  3.3 Asien/Pazifik
4 Marketing
```

Firmenstruktur

Automatische Nummerierung in Java [Slide 34]

```
public static <E> void
printPreorderLabeled(Tree<E> T, Position<E> p,
                     ArrayList<Integer> path) {
  int d = path.size();          // depth equals the length of the path

  // print indentation, then label:
  System.out.print(spaces(2*d));
  for (int j=0; j < d; j++)
    System.out.print(path.get(j) + (j == d-1 ? " " : "."));
  System.out.println(p.getElement());

  path.add(1);                  // add path entry for first child
  for (Position<E> c : T.children(p)) {
    printPreorderLabeled(T, c, path);
    path.set(d, 1 + path.get(d)); // increment last entry of path
  }
  path.remove(d);               // restore path to its incoming state
}
```

Anwendung: Dateibaum-Größe [Slide 35]

```
public static int
diskSpace(Tree<Integer> T, Position<Integer> p) {
  // we assume element represents space usage:
  int subtotal = p.getElement();
  for (Position<Integer> c : T.children(p))
    subtotal += diskSpace(T, c);
  return subtotal;
}
```

Anwendung: Geklammerte Serialisierung [Slide 36]

$P(T) =$ `p.getElement() +`
`"(" +` $P(T_1)$ `+ ", " + + ", " +` $P(T_k)$ `+ ")"`

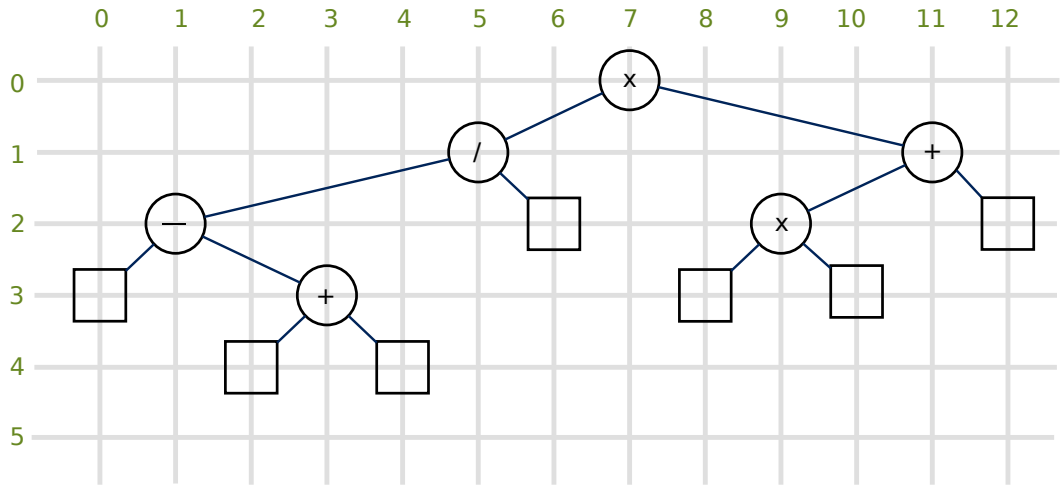
Firmenstruktur:

`Post AG (Brief Kommunikation, Paket Inland, International (Europa, Amerika, Asien/Pazifik), Marketing)`

Geklammerte Serialisierung in Java [Slide 37]

```
public static <E> void
parenthesize(Tree<E> T, Position<E> p) {
  System.out.print(p.getElement());
  if (T.isInternal(p)) {
    boolean firstTime = true;
    for (Position<E> c : T.children(p)) {
      System.out.print(firstTime ? " (" : ", ");
      firstTime = false; // any future passes will get comma
      parenthesize(T, c); // recur on child
    }
    System.out.print(")");
  }
}
```

Anwendung: Graphische Koordinaten [Slide 38]



Graphische Koordinaten in Java [Slide 39]

```
// fake interface for geometric layout problem
public interface Geometric {
  public void setX(int x);
  public void setY(int y);
}

/** Defines geometry for an inorder layout of subtree of a binary tree. */
// x: number of nodes visited before p in an inorder traversal
// y: depth of p
public static <E extends Geometric> int
layout(BinaryTree<E> T, Position<E> p, int d, int x) {
  if (T.left(p) != null)
    x = layout(T, T.left(p), d+1, x); // resulting x will be increased
  p.getElement().setX(x++);         // post—increment x
  p.getElement().setY(d);
  if (T.right(p) != null)
    x = layout(T, T.right(p), d+1, x); // resulting x will be increased
  return x;
}
```

Euler-Tour-Traversierung [Slide 40]

Generalisierung von Prä- und Postorder-Traversierung mit Prä- und Post-Besuchen:

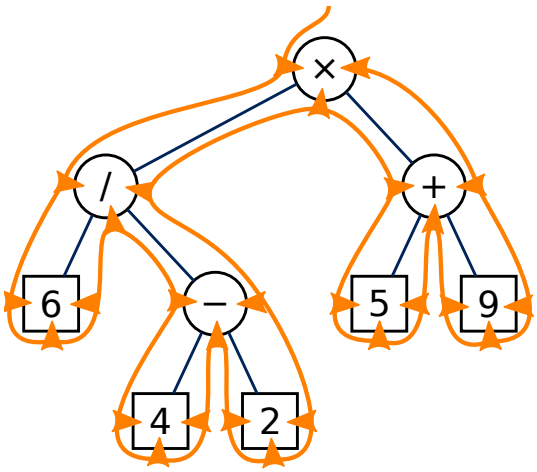
```
Algorithm eulerTour(T, v):
  pre-visit v
  foreach child c in T.children(v) do
    eulerTour(T, c)
  post-visit v
```

> Die sogenannte Euler-Tour-Traversierung kombiniert die Prä- und die Postorder-Traversierungen, indem bei der rekursiven Traversierung jeder Knoten sowohl vor als auch nach den rekursiven Aufrufen auf seinen Kindern besucht wird.
>
> Wir sehen das hier im Pseudocode des `eulerTour()`-Algorithmus: Der aktuelle Knoten v wird zweimal besucht. Die beiden Besuche, `pre-visit` und `post-visit`, können sich in ihrer Funktion durchaus unterscheiden. Dazwischen finden die rekursiven Aufrufe auf den Kindern statt.

Binäre Euler-Tour-Traversierung [Slide 41]

Zusätzlich ein In-Besuch:

```
Algorithm eulerTourBinary(T, v):
  pre-visit v
  if T.hasLeft(v) then
    eulerTourBinary(T, T.left())
  in-visit v
  if T.hasRight(v) then
    eulerTourBinary(T, T.right())
  post-visit v
```



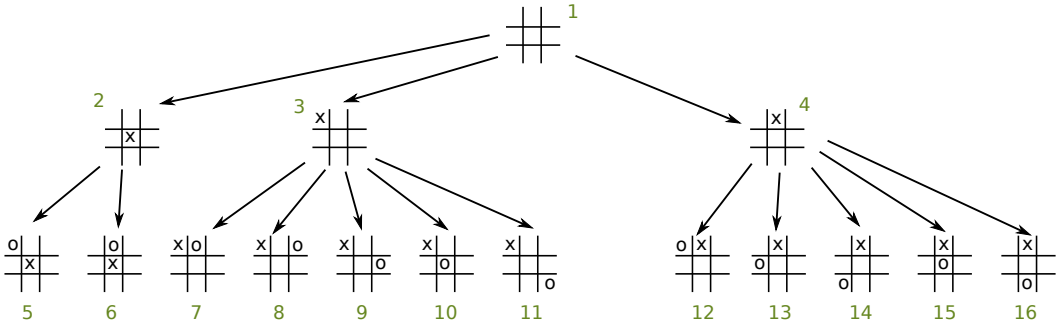
Besuche: `((6/(4-2))×(5+9))`

- Prä-Besuch (interne Knoten): `"("`
- In-Besuch: Wert bzw. Operator
- Post-Besuch (interne Knoten): `")"`

> Bei Binärbäumen fügt man oft einen weiteren Besuch *zwischen* den rekursiven Aufrufen auf den Kindern hinzu, den `in-visit`. Jeder Knoten wird nun also dreimal besucht, wie hier durch die drei Pfeilköpfe an jedem Knoten angedeutet: Der Prä-Besuch von links, der In-Besuch von unten, und der Post-Besuch von rechts.
>
> Diese binäre Euler-Tour-Traversierung können wir nutzen, um unseren arithmetischen Ausdruck vollständig geklammert auszugeben. Dafür geben wir bei jedem Prä-Besuch eines internen Knotens eine öffnende Klammer aus und bei jedem Post-Besuch eine schließende Klammer. Bei den Blättern können wir diese weglassen, weil sie ja nur ein einziges Element umschließen würden.
>
> Der In-Besuch dient wie vor dazu, das Element des jeweiligen Knotens auszugeben.
>
> Das Ergebnis dieser Besuche sehen wir unten rechts: ein vollständig geklammerter arithmetischer Ausdruck in Infix-Notation, dessen Berechnung den erwarteten Wert liefert.

Breiten-Traversierung: Beispiel [Slide 42]

Spiel-KI, wobei der *game tree* nicht komplett absuchbar ist:



> Wir haben nun die Prä- und Postorder-Traversierung und die Euler-Tour-Traversierung kennengelernt, die beide kombiniert. Für Binärbäume haben wir darüber hinaus die Inorder-Traversierung definiert, sowie eine spezielle Erweiterung der Euler-Tour-Traversierung mit einem zusätzlichen In-Besuch. Alle diese Traversierungen sind Tiefentraversierungen; die Kinder werden vor den Geschwistern besucht. Alle Tiefentraversierungen lassen sich durch einfache, rekursive Algorithmen implementieren.

Bei einer ***Breitentraversierung*** werden die *Geschwister vor den Kindern* besucht. Sie besucht die Knoten eines Baums also zeilenweise.

Ein typisches Anwendungsbeispiel ist eine Spiel-KI. Die KI muss einen möglichst vorteilhaften Spielzug ermitteln. Hierzu repräsentiert man die möglichen weiteren Verläufe des Spiels als einen Baum, den sogenannten *game tree*.

Die Wurzel des *game trees* ist der aktuelle Zustand des Spiels. Jede von der Wurzel ausgehende Kante entspricht einem möglichen Spielzug der KI, und führt jeweils zu einem Knoten, der den aus diesem Spielzug resultierenden Folgezustand des Spiels repräsentiert.

Als nächstes ist die Gegnerin am Zug. Da die KI nicht weiß, welchen Zug die Gegnerin ausführen wird, bedenkt sie alle möglichen Züge. Damit wird die nächste Zeile des *game trees* aufgespannt.

Da meist eine Vielzahl von Spielzügen zur Auswahl steht und stochastische Elemente mehrere Kinder pro Spielzug generieren können, wächst dieser Baum sehr schnell in die Breite. Dies bedeutet umgekehrt, dass die Anzahl der Knoten exponentiell mit der Höhe des Baums wächst, wobei der Exponent die Anzahl der Kinder der Knoten ist, der sogenannte *branching factor*.

Es liegt auf der Hand, dass die Spiel-KI den *game tree* meist nicht bis zum Ende des Spiels berechnen kann. Sie muss also nach Verbrauch seines Zeitbudgets abbrechen, und den Wert der bis dahin generierten, möglichen zukünftigen Spielzustände abschätzen.

Damit diese Schätzungen tatsächlich zu einem guten Spielzug führen, sollte man möglichst alle aktuell möglichen Spielzüge evaluieren.

Bei einer Tiefentraversierung hat man aber möglicherweise sein gesamtes Zeitbudget verbraucht, bevor man überhaupt den ersten aktuell möglichen Spielzug fertig evaluiert hat. Also wählt man die Breitentraversierung. Auf diese Weise hat man immer eine Schätzung der Werte aller aktuell möglichen Züge. Diese Schätzwerte werden mit jeder weiteren Zeile des *game trees* verbessert, bis das Zeitbudget aufgebraucht ist. Dann wird der Spielzug mit dem besten geschätzten Wert ausgeführt.

Breiten-Traversierung [Slide 43]

```
Algorithm breadthFirst():
  initialize queue Q to contain root()
  while Q not empty do
    p = Q.dequeue()
    visit p
    foreach child c ∈children(p) do
      Q.enqueue(c)
```

Anmerkung
Ersetzt man die Warteschlange durch einen Stapel (und fügt die Kinder in umgekehrter Reihenfolge ein), dann erhalten wir eine nicht-rekursive Implementation einer Präorder-Traversierung.

Wie kann man eine Breitentraversierung implementieren? Sie muss ja die Geschwister besuchen, bevor die Kinder besucht werden. Eine rekursive Lösung bietet sich also nicht an.

Eine Möglichkeit besteht darin, die Datenstruktur um Zeiger zwischen den Knoten innerhalb einer Zeile zu erweitern. Dies kann in der Praxis sinnvoll sein, verkompliziert jedoch die Datenstruktur und kann Folgekosten haben. Wenn beispielsweise Unterbäume umgehängt werden, dann müssen diese Zeiger an verschiedenen Stellen im Baum angepasst werden.

> Wir können allerdings auf elegante Weise eine Breitentraversierung implementieren, ohne unsere Datenstruktur zu modifizieren. Dazu benötigen wir eine Warteschlange. Damit ist der Algorithmus ganz einfach:
>
> Wenn ein Knoten besucht wird, schiebt er danach seine Kinder in die Warteschlange. Danach wird der nächste Knoten aus der Warteschlange geholt.
>
> Damit liegen immer die Geschwister nebeneinander in der Warteschlange und werden in dieser Reihenfolge besucht. Da jeder dieser Geschwisterknoten seinerseits seine Kinder in die Warteschlange einreiht, ein Geschwister nach dem anderen, landen auf diese Weise die Enkel ebenfalls alle nebeneinander in der Warteschlange, und so weiter, Zeile für Zeile.
>
> Von diesen Zeilen ist im Algorithmus übrigens nichts zu sehen. Es werden einfach nur jeweils die Kinder hinten in die Warteschlange eingereiht. Die Fortsetzung in der nächsten Zeile ergibt sich ganz von selbst, wenn in der Warteschlange auf den letzten Knoten der aktuellen Zeile der erste Knoten der nächsten Zeile folgt.
>
> Es lohnt sich, diesen Algorithmus anhand eines Beispielbaums Schritt für Schritt durchzuspielen und dabei zu beobachten, wie sich der Inhalt der Warteschlange entwickelt.

6 Zusammenfassung

Zusammenfassung [Slide 44]

- ADT: Generische Bäume und Binärbäume
- DS: verkettete Struktur; Array
- Traversierung:
  - ***Tiefen***-Traversierung (*rekursiv*): Präorder, Postorder, Inorder, Euler-Tour
  - ***Breiten***-Traversierung (iterativ mittels *Warteschlange*)

Bibliographie [Slide 45]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.