Datenstruktur Baum

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

> Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen.
> [Ottmann, Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Springer 2012]

Baumstrukturen finden sich in verschiedenen Kontexten in der Informatik wieder:

- Syntaxbäume, Darstellung von Termen, Visualierung von Dateisystemen, Darstellung von Hierarchien (z.B. Typhierarchie) etc.

Die Datenstruktur Baum spielt außerdem eine wichtige Rolle in der Implementierung von abstrakten Datentypen wie Maps und auch von Datenbanken.

In diesem Kapitel geben wir eine Einführung in die Datenstruktur Baum und zeigen Varianten, wie Baumstrukturen implementiert werden können. Wir diskutieren außerdem, wie sortierte markierte Binärbäume die Suche von Elementen im Baum erleichtern.

Lernziele dieses Kapitels:

- Die Definition wichtiger Begriffe im Zusammenhand mit Bäumen zu kennen.
- Markierte Bäumen, insbesondere Suchbäume, in Java zu implementieren.
- Terminierungsbeweise für Methoden auf Bäumen durchzuführen.
- Baumstrukturen zur Darstellung der Syntax von arithmetischen Ausdrücken zu verwenden.

1 Grundkonzepte

Führen wir zunächst einige Begriffe ein:

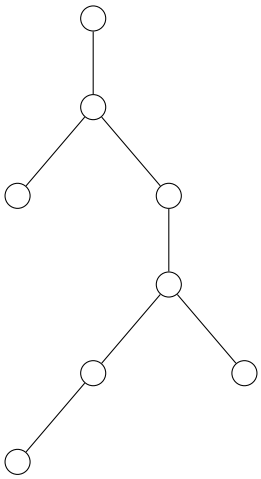
- In einem endlich verzweigten ***Baum*** hat jeder ***Knoten*** endlich viele ***Kinder***.
- Einen Knoten ohne Kinder nennt man ein ***Blatt***, einen Knoten mit Kindern einen ***inneren*** Knoten oder ***Zweig***.

- Jeder Knoten (bis auf die Wurzel) hat genau einen ***Elternknoten***. Den Knoten ohne Elternknoten nennt man ***Wurzel***.
- Zu jedem Knoten k gehört ein ***Unterbaum***, nämlich der Baum, der k als Wurzel hat.
- In einem ***Binärbaum*** hat jeder Knoten maximal zwei Kinder.

In unserer Vorlesung behandeln wir eine bestimmte Art von Bäumen, sogenannte *gerichtete gewurzelte* Bäume, bei denen wir stets einen Knoten als Wurzelknoten auszeichnen und die Verbindung von Eltern- zu Kindknoten (d.h. weg von der Wurzel) ausgerichtet ist. In der Graphentheorie werden allgemeinere Arten von Bäumen definiert. Dazu erfahren Sie mehr in den Vorlesungen FGdP oder Graphentheorie.

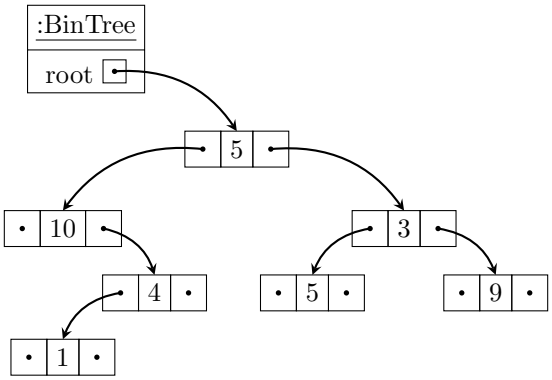
Frage 1:

- Markieren Sie im folgenden Baum einen Wurzelknoten!
- Welche Knoten sind Blätter? Welche sind innere Knoten?
- Handelt es sich um einen Binärbaum?



1.1 Markierte Bäume

Ein Baum heißt ***markiert***, wenn jedem Knoten k ein Wert/eine Markierung $m(k)$ zugeordnet ist.



Um einen markierten Binärbaum mit ganzen Zahlen als Markierung zu implementieren, benötigen wir zunächst eine Klasse für die Baum-Knoten. Diese Knoten-Klasse erhält Attribute für die Markierung und die Referenzen auf die beiden Kindknoten `left` und `right`.

```
// Repraesentiert die Knoten eines Baums mit Markierungen
class TreeNode {
  private int mark;
  private TreeNode left;
  private TreeNode right;

  TreeNode(int mark, TreeNode left, TreeNode right) {
    this.mark = mark;
    this.left = left;
    this.right = right;
  }
  // Liefert die Markierung eines Knotens
  int getMark() {
    return this.mark;
  }
  // Liefert die Referenz auf den linken Kindknoten
  TreeNode getLeft() {
    return this.left;
  }
  // Liefert die Referenz auf den rechten Kindknoten
  TreeNode getRight() {
    return this.right;
  }
  ...
}
```

Die Implementierung des Baums hält dann die Referenz auf den Wurzelknoten `root`. Dieses Attribut ist anfangs mit `null` initialisiert. Von diesem Wurzelknoten sind alle anderen Knoten des Baumes erreichbar. Jeder Knoten des Baumes ist selbst wiederum Wurzelknoten des von ihm aufgespannten Unterbaums.

```
// Repraesentiert einen markierten Binaerbaum
public class BinTree {
  private TreeNode root;
  ...
```

```
}
```

Bäume sind - wie auch verlinkte Listen - ***rekursive Datentypen***. Bei rekursiven Datentypen wird der Datentyp selbst zu seiner eigenen Definition herangezogen.

- Eine Liste ist entweder leer oder besteht aus einem Element und einer (Rest-) Liste.
- Eine Baum ist entweder leer oder besteht aus einer Markierung und einem linken und rechten (Unter-) Baum.

Bei einer Definition einer rekursiven Datenstruktur gibt es einen oder mehrere Basisfälle (z.B. leere Liste / leerer Baum) und Rekursionsfälle, die beschreiben, wie man aus kleineren Instanzen der Datenstruktur größere aufbaut.
In der Implementierung in Java, die wir hier präsentieren, zeigt sich der rekursive Charakter von Bäumen dadurch, dass die Baumknoten als Attribute Referenzen auf Baumknoten haben.[1]
Berechnungen auf rekursiven Datenstrukturen wie Bäumen und Listen lassen sich in der Regel auf eine Kombination der Berechnung für den aktuellen Knoten und des Ergebnisses der Berechnung für den rekursiven Teil der Datenstruktur zurückführen. Essentiell ist es dabei, die Berechnung für die Basisfälle nicht zu vergessen.

Beispiel Um alle Markierungen in einem mit int-markierten Binärbaum aufzusummieren, addieren wir zu der Markierung des Knotens, der diesen Baum aufspannt, die Summe der Markierungen für den linken und für den rechten Unterbaum. Dazu berechnen wir die Summe zum Unterbaum des linken Kindknoten, dann zum rechten Kindknoten und addieren die jeweiligen Ergebnisse dann zur Markierung des aktuellen Knotens. Im Basisfall für den leeren Baum (Knoten ist gleich null) ist die Summe gleich 0.

```
public class BinTree {
  private TreeNode root;
  ....
  // Berechnet die Summe der Markierungen des Baums
  public int sum() {
    return sum(this.root);
  }
  // Hilfsmethode
  private int sum(TreeNode node) {
    if (node == null) {
        return 0;
    }
    return node.getMark() + sum(node.getLeft())
                          + sum(node.getRight());
  }
```

[1] In Java können Referenzen auf beliebige Objekt des zugehörigen Referenztyps verweisen. Dies kann zu unerwünschten zirkulären Objektgeflechten führen, die der obigen Definition von rekursiven Datentypen widersprechen. Wie bei der Implementierung der einfachverketteten Liste müssen wir auch hier sicherstellen, dass die Implementierung des Baumstruktur eine entsprechende Invariante garantiert. Wir gehen hier davon aus, dass innerhalb der hier beschriebenen rekursiven Datentypen unterschiedliche Referenzen nicht auf das gleiche Objekt verweisen.

Die öffentliche Methode `sum()` implementiert den beschriebenen Algorithmus. In ihr wird zunächst die Summation auf dem Wurzelknoten `root` aufgerufen in der privaten Hilfsmethode `sum(TreeNode node)` (Stichwort: Overloading). In der Implementierung verwenden wir die gleiche private Hilfsmethode `sum(TreeNode node)` für die Summation bei den Kindknoten. Die Methode `sum(TreeNode node)` ist rekursiv, sie ruft sich selbst erneut auf. Die rekursive Struktur des Baums spiegelt sich wider in diesen rekursiven Methodenaufrufen.

Frage 2: Wie können wir testen, ob der Baum eine bestimmte Markierung enthält?

- Was ist das Ergebnis im Basisfall (leerer Baum)?
- Was ist das Ergebnis im Rekursionsfall?

Schreiben Sie eine Methode `public boolean contains(int x)` für die Klasse `BinTree`, die `true` liefert, wenn einer der Knoten des Baumes mit `x` markiert ist!

1.2 Terminierungsbeweise auf rekursiven Datenstrukturen

Um die Terminierung der Methode `sum(TreeNode node)` zu zeigen, gehen wir wieder nach dem bekannten Schema vor (siehe Kapitel zur Terminierung).

1. Da wir die Terminierung für alle gültigen Bäume mit `node` als Wurzelknoten beweisen wollen, schränken wir den Parameterbereich nicht weiter ein.
2. Damit ist der Beweis, dass der gültige Parameterbereich nicht verlassen wird, auch relativ einfach, da ein gültiger Wurzelknoten `node` einen gültigen linken und rechten Teilbaum besitzt.
3. Um den Terminierungsbeweis zu führen, müssen wir eine gültige Abstiegsfunktion wählen. Die Idee dabei ist, dass die linken und rechten Teilbäume immer kleiner sind als der Baum selbst. Die informelle Beschreibung der Abstiegsfunktion ist somit wie folgt:

$$h : \text{TreeNode} \to \mathbb{N}_0$$
$$h(t) = \text{Höhe des Baumes t}$$

Wir definieren die Höhe eines Blattes als 0 und die Höhe eines Zweiges als das Maximum der Höhe der Teilbäume um 1 erhöht. Somit ergibt sich die folgende Java-Methode, die die Höhe eines Baumes beginnend mit der Wurzel berechnet:

```
public int height(TreeNode node) {
  if (node == null) {
    return 0;
  }
```

```
  return 1 +
    Math.max(height(node.getLeft()), height(node.getRight()));
}
```

Wir können nun die Abstiegsfunktion h definieren:

$$h(t) = height(t)$$

Die Abstiegsfunktion h bildet in die natürlichen Zahlen ab, da das Ergebnis für die kleinsten möglichen Bäume bestehend nur aus einem Blatt gleich 0 ist und für größere Bäume entsprechend größer.

4. Zu zeigen bleibt, dass die Parameter für rekursive Aufrufe echt kleiner werden.

   - Aufruf `sum(node.getLeft())`:
     $h(node) = height(node) > height(node.getLeft()) = h(node.getLeft())$
     Dies gilt, da es sich bei `node.getLeft()` um eine kleinere Instanz eines Baumes handelt, da `node` kein Blatt ist (wegen der `if`-Bedingung).
     Formal bedeutet das, dass

     $$\begin{aligned} height(node) &= 1 + Math.max(height(node.getLeft()), height(node.getRight())) \\ &\geq 1 + height(node.getLeft()) \\ &> height(node.getLeft()) \end{aligned}$$

   - Aufruf `sum(node.getRight())`:
     Gleiche Begründung wie oben.

Somit haben wir gezeigt, dass die `sum`-Methode für alle gültigen Bäume terminiert.

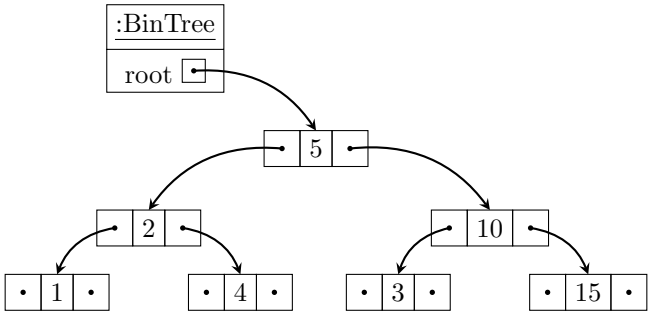
Da es sich bei `height` ebenfalls um eine rekursive Methode handelt, müsste man für diese Methode ebenfalls die Terminierung zeigen, um sie in der Abstiegsfunktion benutzen zu können. Der Beweis der Terminierung der `height`-Methode ist auf die Baum-Invariante begründet, dass es keine Zyklen in dem Objektgeflecht der Baumknoten gibt. Im Rahmen dieser Vorlesung können Sie davon ausgehen, dass die Terminierung der `height`-Methode gilt.

1.3 Sortierte markierte Bäume

Definition Ein mit ganzen Zahlen markierter Binärbaum heißt ***sortiert***, wenn für alle Knoten k gilt:

- Alle Markierungen der linken Nachkommen von k sind kleiner als oder gleich $m(k)$.
- Alle Markierungen der rechten Nachkommen von k sind größer als $m(k)$.

Frage 3: Ist dieser markierte Binärbaum sortiert?



Die Sortiertheitseigenschaft von sortierten markierten Binärbaum erleichtert es ungemein, Elemente in dem Baum zu suchen. Diese Bäume werden daher auch oft ***Suchbäume*** genannt.
Um ein Element in dem Baum zu suchen, beginnen wir zunächst wieder mit dem `root`-Knoten:

- In einem leeren Baum ist das Element nicht enthalten.
- Falls der aktuelle Knoten mit dem gesuchten Element markiert ist, ist es im Baum enthalten.
- Andernfalls suchen wir rekursiv beim linken Kindknoten, falls das gesuchte Element kleiner ist, als die Markierung des Knotens; ist das gesuchte Element größer, suchen wir rekursiv beim rechten Kindknoten.

```
// Repraesentiert einen sortierten markierten Binaerbaum
public class SortedBinTree {
  private TreeNode root;
  public SortedBinTree() {
    root = null;
  }
  // prueft, ob ein Element im Baum enthalten ist
  public boolean contains(int element) {
    return contains(root, element);
  }
  private boolean contains(TreeNode node, int element) {
    if (node == null) {
        return false;
    }
    if (element < node.getMark()) {
        // kleinere Elemente links suchen
        return contains(node.getLeft(), element);
    } else if (element > node.getMark()) {
        // groessere Elemente rechts suchen
        return contains(node.getRight(), element);
    } else {
        // gefunden!
```

```
        return true;
    }
}
```

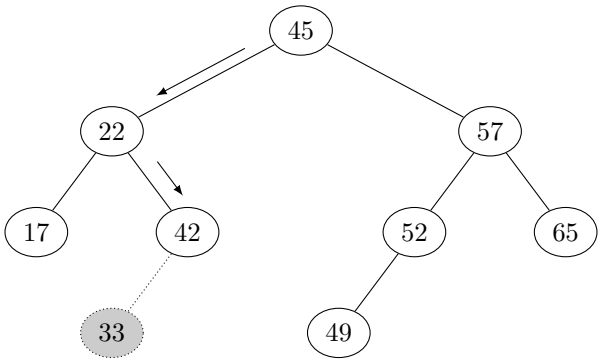
Einfügen in binären Suchbaum Damit die Suche korrekt funktioniert, muss auch hier wieder eine entsprechende Invariante, nämlich die Sortiertheit, sicher gestellt werden. Neue Knoten werden immer als Blätter eingefügt. Die Position des Blattes wird durch den Wert des neuen Eintrags festgelegt. Um Elemente in einen `SortedBinTree` einzufügen wählen wir das folgende algorithmische Vorgehen:

- Ist der Baum noch leer, wird der erste Eintrag als Wurzel des Baums hinzugefügt.
- Ein Knoten wird
  - in den linken Unterbaum der Wurzel eingefügt, wenn sein Wert kleiner gleich ist als der Wert der Wurzel;
  - in den rechten, wenn er größer ist.

  Dieses Verfahren wird rekursiv fortgesetzt, bis die Einfügeposition bestimmt ist.

Beispiel: Beim Einfügen von 33 in den skizzierten Baum wird - angefangen bei der Wurzel - die Markierung des Knotens mit 33 verglichen.

- 33 ist kleiner als 45; daher wird das Einfügen rekursiv auf dem linken Kindknoten fortgeführt;
- 33 ist größer als 22; daher wird das Einfügen rekursiv auf dem rechten Kindknoten fortgeführt;
- 33 ist kleiner als 42; da der Knoten mit Markierung 42 keinen linken Kindknoten hat, wird ein neuer Knoten mit Markierung 33 als linker Kindknoten hier eingefügt.



Die Implementierung folgt diesem rekursiven Schema:

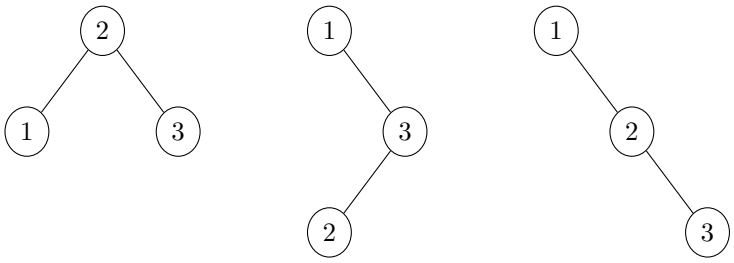
```
public void add(int element) {
    root = add(root, element);
}

private TreeNode add(TreeNode node, int element){
    if (node == null) {
        return new TreeNode(element);
    } else if (element <= node.getMark()) {
        node.setLeft(add(node.getLeft(), element));
    } else {
        node.setRight(add(node.getRight(), element));
    }
    return node;
}
```

Die Hilfsmethode `add` gibt jeweils die Referenz auf die neue Wurzel des Teilbaums nach Einfügen von `element` zurück. Damit ist es nicht nötig, zu unterscheiden, ob die Wurzel oder der linke oder rechte Teilbaum aktualisiert werden muss.

Bemerkungen

- Die Reihenfolge des Einfügens bestimmt das Aussehen des sortierten markierten Binärbaums.

  Die folgenden Bäume sind durch die Einfügereihenfolge $2 \to 3 \to 1$ bzw. $1 \to 3 \to 2$ bzw. $1 \to 2 \to 3$ enstanden.



- Wie man bei dem letzten der drei Beispiel sieht, entartet der Baum zur linearen Liste, wenn die Elemente in einer sortierten Reihenfolge einfügt werden. Eine solche Entartung kann man vermeiden, indem man den Suchbaum während des Einfügens umstrukturiert. ***Balancierte Bäume*** stellen sicher, dass sich für jeden Knoten die Tiefe des linken Teilbaums und die Tiefe des rechten Teilbaums nur um eine bestimmte Wert unterscheidet.

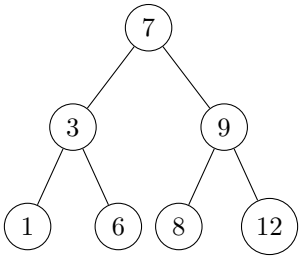
Löschen in binärem Suchbaum Um Elemente aus einem binären Suchbaum zu entfernen, suchen wir zunächst den Eintrag im Baum. Haben wir den entsprechenden Knoten gefunden, machen wir folgende Fallunterscheidung:

- *Knoten hat keine Kinder*
  Der zu löschende Knoten wird einfach entfernt (d.h. die Referenz im Elternknoten wird durch `null` ersetzt).

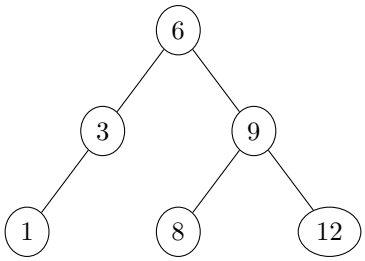
- *Knoten hat genau ein Kind*
  Der Kindknoten wird an die Stelle gesetzt, an der der zu löschende Knoten war.

- *Knoten hat zwei Kinder*
  Der zu löschende Knoten wird durch den am weitesten rechts liegenden Knoten seines linken Teilbaums ersetzt.[2] Dieser Knoten muss dann aus dem entsprechenden Teilbaum entfernt werden.

Betrachten wir einmal folgenden Baum:



Nach Löschen von Element 7:




```
/** entfernt ein Vorkommen von element aus dem Baum */
public void remove(int element) {
    root = remove(root, element);
}

/** Entfernt ein Vorkommen von element aus dem Teilbaum
    mit Wurzel node und liefert die Wurzel des veränderten Teilbaums */
private TreeNode remove(TreeNode node, int element) {
    if (node == null) {
        return null;
    }
    if (element < node.getMark()) {
        TreeNode newLeft = remove(node.getLeft(), element);
        node.setLeft(newLeft);
        return node;
    } else if (element > node.getMark()) {
        TreeNode newRight = remove(node.getRight(), element);
        node.setRight(newRight);
        return node;
    } else {
        // zu loeschendes Element gefunden
        if (node.getLeft() == null) {
            // wenn der linke Teilbaum leer ist, dann nur den rechten nehmen
            return node.getRight();
        } else if (node.getRight() == null) {
            // analog zu node.left == null
```

[2] Alternativ: den am weitesten links liegenden Knoten seines rechten Teilbaums

```
            return node.getLeft();
        } else {
            // wenn der Knoten zwei Kinder hat, den Knoten mit dem groessten
            // Element aus dem linken Teilbaum suchen:
            TreeNode maxNodeLeft = maxNode(node.getLeft());
            // Die Markierung von diesem Knoten nehmen wir fuer den aktuellen:
            node.setMark(maxNodeLeft.getMark());
            // Und dann loeschen wir das Element aus dem linken Teilbaum:
            node.setLeft(remove(node.getLeft(), node.getMark()));
            return node;
        }
    }
}
```

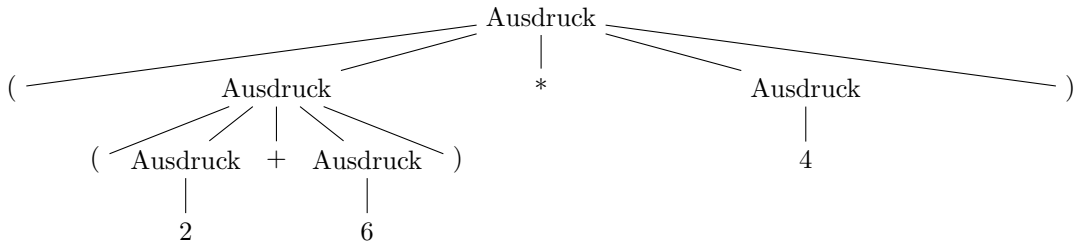
Frage 4: Ergänzen Sie die Implementierung der Methode `TreeNode maxNode(TreeNode node)`.

2 Anwendungsbeispiel: Syntaxbäume

Um die Struktur eines Wortes einer formalen Sprache darzustellen, werden ***Syntaxbäume*** verwendet. Die rekursive Art der Sprache, die oft durch eine kontextfreie Grammatik beschrieben ist, hat dabei ihre Entsprechung in der rekursiven Natur des Baums.
In Kapitel 2 haben wir Syntaxbäume bereits kennengelernt. Wir betrachten hier den Ausschnitt für die arithmetischen Ausdrücke der Femto-Sprache.

$$
\begin{aligned}
N &= \{\texttt{Ausdruck}\} \\
T &= \{\texttt{zahl}, \texttt{(}, \texttt{)}, \texttt{+}, \texttt{*}\} \\
\Pi &= \left\{ \begin{array}{lcl} \texttt{Ausdruck} & \rightarrow & \texttt{zahl} \\ & | & \texttt{(Ausdruck + Ausdruck)} \\ & | & \texttt{(Ausdruck * Ausdruck)} \end{array} \right\}
\end{aligned}
$$

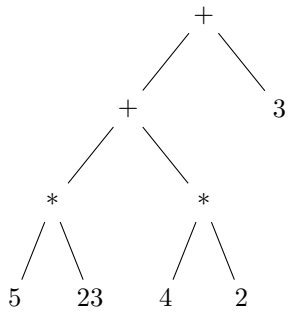
Für den Ausdruck $((2 + 6) * 4)$ ergibt sich folgender Syntaxbaum:



Der Syntaxbaum spiegelt direkt die Produktionen der Grammatik wider. Er beschreibt die Struktur des Ausdrucks in allen Details; so ist für jedes Terminalsymbol ein Blattknoten vorhanden. Für die Auswertung des Ausdrucks werden einige der Terminalsymbole, wie die Klammern, nicht benötigt.

Für die Analyse und Auswertung von Ausdrücken verwenden wir daher alternativ einen ***abstrakter Syntaxbaum***, in dem diese für die weitere Verarbeitung unnötigen Knoten entfernt sind.

Beispiel Für den Ausdruck $(((5 * 23) + (4 * 2)) + 3)$ ergibt sich folgender abstrakter Syntaxbaum:



2.1 Traversieren von Bäumen

Es gibt verschiedene Möglichkeiten die Knoten eines Baums zu durchlaufen (***traversieren***). Ein Baum kann zum Beispiel zuerst in die Tiefe (depth-first, Tiefensuche) durchlaufen werden oder zuerst in der Breite (breadth-first, Breitensuche).
Wir betrachten hier nur die Tiefensuche, für die es drei Varianten gibt:

Vorordnung (preorder) Das Vorgehen zum Durchlauf in Vorordnung ist:

1. Betrachte den aktuellen Knoten.
2. Durchlaufe (rekursiv) den linken Teilbaum.
3. Durchlaufe (rekursive) den rechten Teilbaum.

Das Ergebnis der Traversierung des Syntaxbaums oben ist: `+ + * 5 23 * 4 2 3` Diese Schreibweise eines arithmetischen Ausdrucks wird auch ***Präfix-Notation*** oder ***polnische Notation*** genannt.

Nachordnung (postorder) Das Vorgehen zum Durchlauf in Nachordnung ist:

1. Durchlaufe (rekursiv) den linken Teilbaum.
2. Durchlaufe (rekursive) den rechten Teilbaum.
3. Betrachte den aktuellen Knoten.

Das Ergebnis der Traversierung des Syntaxbaums oben ist: `5 23 * 4 2 * + 3 +` Diese Schreibweise eines arithmetischen Ausdrucks wird auch ***Postfix-Notation*** oder ***umgekehrte polnische Notation*** genannt.

Inordnung (inorder) Das Vorgehen zum Durchlauf in Inordnung ist:

1. Durchlaufe (rekursiv) den linken Teilbaum.
2. Betrachte den aktuellen Knoten.
3. Durchlaufe (rekursive) den rechten Teilbaum.

Das Ergebnis der Traversierung des Syntaxbaums oben ist: `5 * 23 + 4 * 2 + 3`

2.2 OO-Modellierung von Syntaxbäumen

Die abstrakten Syntaxbäume für arithmetischen Ausdrücke können wir nun folgendermaßen in einem OO-Modell umsetzen:

- Ein arithmetischer Ausdruck ist vom Typ `AExpr`. Er soll Methoden bereitstellen zur Auswertung sowie zur Darstellung als String in den verschiedenen Notationen (Präfix-/Postfix-Notation, Inorder-Notation).
- Eine Summe, ein Produkt und eine int-Konstante sind verschiedene Arten von arithmetischen Ausdrücken.
  - Die Auswertung einer int-Konstante soll den zugehörigen int-Wert liefern.
  - Bei Produkt und Summe wird zunächst der linke und rechte Teilausdruck ausgewertet und diese Teilergebnisse zusammen mit dem jeweiligen Operator ausgewertet.

```
1  // Schnittstelle fuer Arithmetische Ausdruecke
2  interface AExpr {
3    // Wert des Ausdrucks
4    int evaluate();
5    // String in Vorordnung
6    String printPreOrder();
7    // String in Nachordnung
8    String printPostOrder();
9    // String in Inordnung
10   String printInOrder();
11 }
```

```
1  class Sum implements AExpr {
2    private AExpr left;
3    private AExpr right;
4
5    Sum (AExpr left, AExpr right) {
6      this.left = left;
7      this.right = right;
8    }
9
10   public int evaluate() {
11     return left.evaluate() + right.evaluate();
12   }
13
14   public String printPreOrder() {
15     return "+ " + left.printPreOrder() + " " + right.printPreOrder();
16   }
17
18   public String printPostOrder() {
```

```
19      return left.printPostOrder() + " " + right.printPostOrder() + " +";
20    }
21
22    public String printInOrder() {
23      return left.printInOrder() + " + " + right.printInOrder() ;
24    }
25
26    public String toString() {
27      return "(" + left.toString() +  " + "
28        + right.toString() + ")";
29    }
30  }
```

```
1   class Const implements AExpr {
2     private int value;
3
4     Const (int value) {
5       this.value = value;
6     }
7
8     public int evaluate() {
9       return value;
10    }
11
12    public String printPreOrder() {
13      return Integer.toString(value);
14    }
15
16    public String printPostOrder() {
17      return Integer.toString(value);
18    }
19
20    public String printInOrder() {
21      return Integer.toString(value);
22    }
23
24    public String toString() {
25      return Integer.toString(value);
26    }
27  }
```

Im Gegensatz zu Prä- und Post-Notation, geht bei der Inorder-Notation für die Ausdrücke verloren, wie die ursprüngliche Baumstruktur und damit die ursprüngliche Klammerung des Ausdrucks war. Wir ergänzen die Implementierung daher noch um eine `toString()`-Methode, die den Ausdruck vollständig geklammert darstellt.