TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- 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.

Datenstruktur Baum

Nach Ottmann, Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Springer 2012

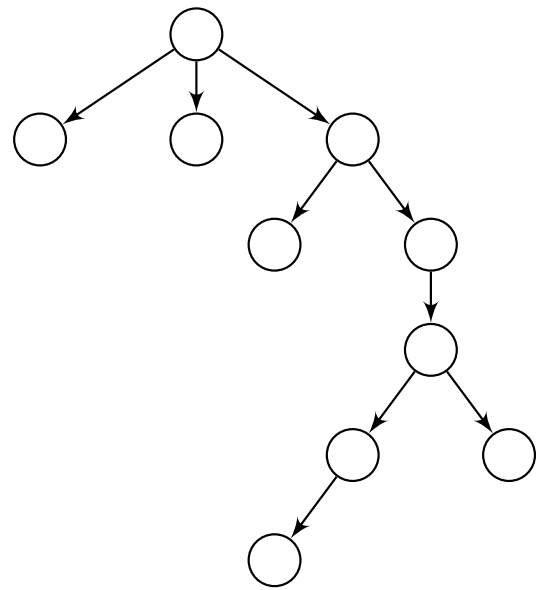
> *Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen.*

Anwendungen:

- Syntaxbäume
- Darstellung von Termen
- Dateisysteme
- Darstellung von Hierarchien (z.B. Typhierarchie)
- Implementierung von Datenbanken
- etc.

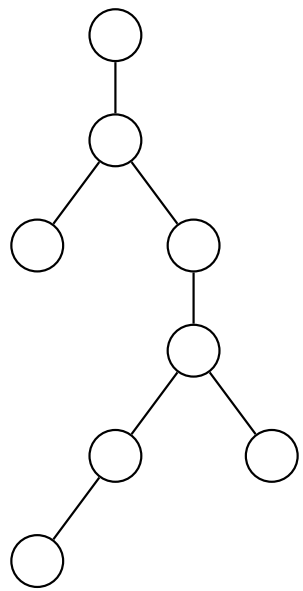
Grundkonzepte

- In einem endlich verzweigten **Baum** hat jeder **Knoten** endlich viele **Kinder**.
- Üblicherweise sagt man, die Kinder sind von *links nach rechts geordnet*.
- 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.



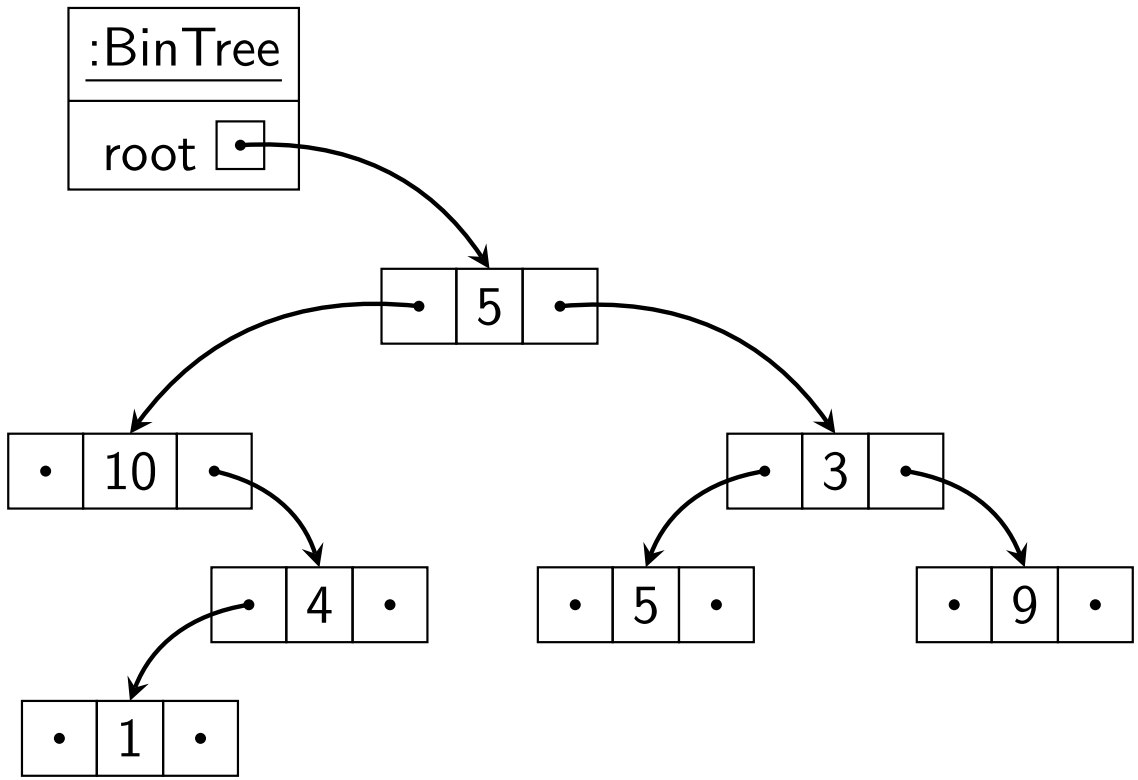
Frage

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



Markierte Bäume

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



Implementierung: Markierte Binärbäume

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

// Repraesentiert einen markierten Binaerbaum
public class BinTree {
  private TreeNode root; // mit null initialisiert
  ...
}
```

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.

Rekursive Methoden

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

Frage

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!

Lösungsvorschlag

```
class BinTree {
    ...
    public boolean contains(int x) {
      return contains(x, this.root);
    }
    private boolean contains(int x, TreeNode n){
      if (n == null) {
        // Basisfall
        return false;
      } else {
      // Rekursionsfall
      return n.getMark() == x
          || contains(x, n.getLeft())
          || contains(x, n.getRight());
      }
    }
}
```

Warum "funktionieren" die rekursiven Methoden?

- Problem: Zirkuläre Objektgeflechte!
- Wie bei der Implementierung der einfachverketteten Liste müssen wir auch hier sicherstellen, dass die Implementierung des Baumstruktur eine entsprechende Invariante garantiert.
- *Invariante:* Innerhalb der hier beschriebenen Bäume verweisen unterschiedliche Referenzen nicht auf das gleiche Objekt.
- Operationen, die die Struktur des Baumes verändern, müssen diese Invariante garantieren.

Terminierungsbeweise auf rekursiven Datenstrukturen I

Beispiel: Terminierung der Methode `sum(TreeNode node)`

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.

Terminierungsbeweise auf rekursiven Datenstrukturen II

3. Idee für Abstiegsfunktion:
   - Linker und rechter Teilbaum sind immer kleiner sind als der Baum selbst
   - Wähle Höhe des Knotens als Abstiegsfunktion!

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

Wir definieren die Höhe eines Blattes als 0 und die Höhe eines Zweiges als das Maximum der Höhe der Teilbäume um 1 erhöht.

Terminierungsbeweise auf rekursiven Datenstrukturen III

Berechnung der Baumhöhe für einen Knoten:

```
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:

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

Terminierungsbeweise auf rekursiven Datenstrukturen IV

Hinweis:

- 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.

Terminierungsbeweise auf rekursiven Datenstrukturen V

4. Zu zeigen bleibt, dass die Parameter für rekursive Aufrufe echt kleiner werden.
   - Für den Aufruf `sum(node.getLeft())` ist zu zeigen:

     $$h(node) > h(node.getLeft())$$

     Dies gilt, da es sich bei `node.getLeft()` um eine kleinere Instanz eines Baumes handelt, da `node` kein Blatt ist (wegen der `if`-Bedingung).

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

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

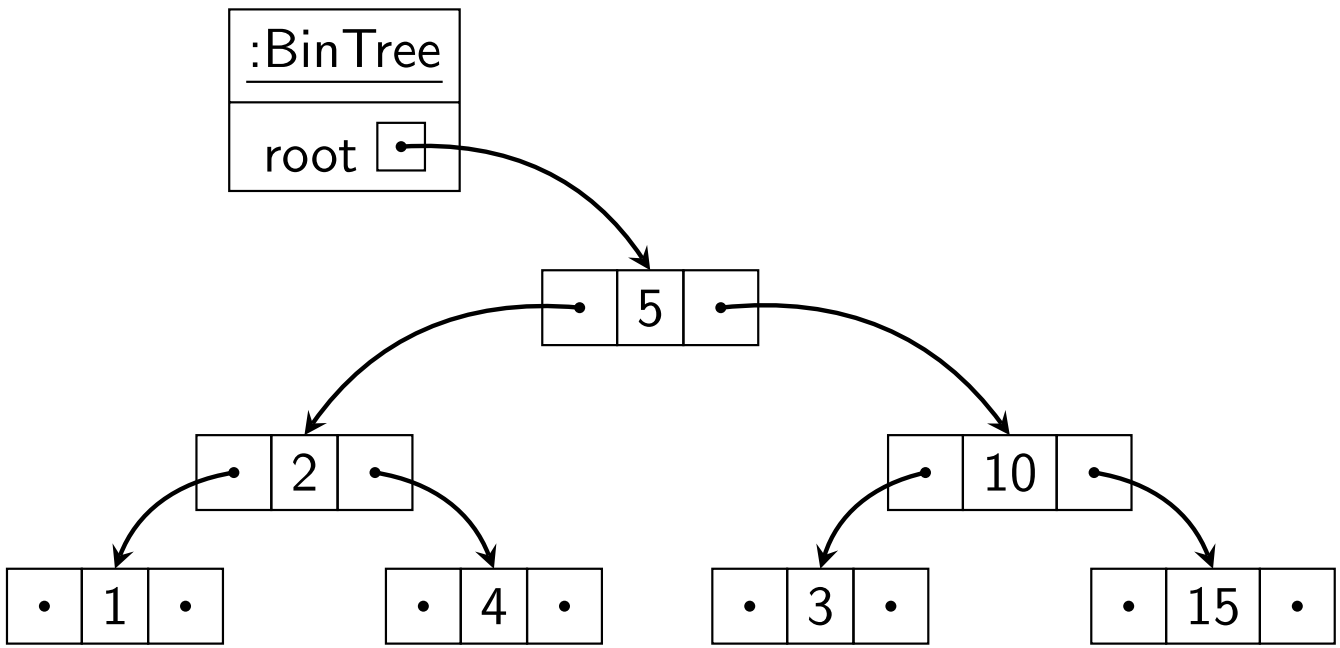
Definition: Sortiertheit markierter Binärbäume

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

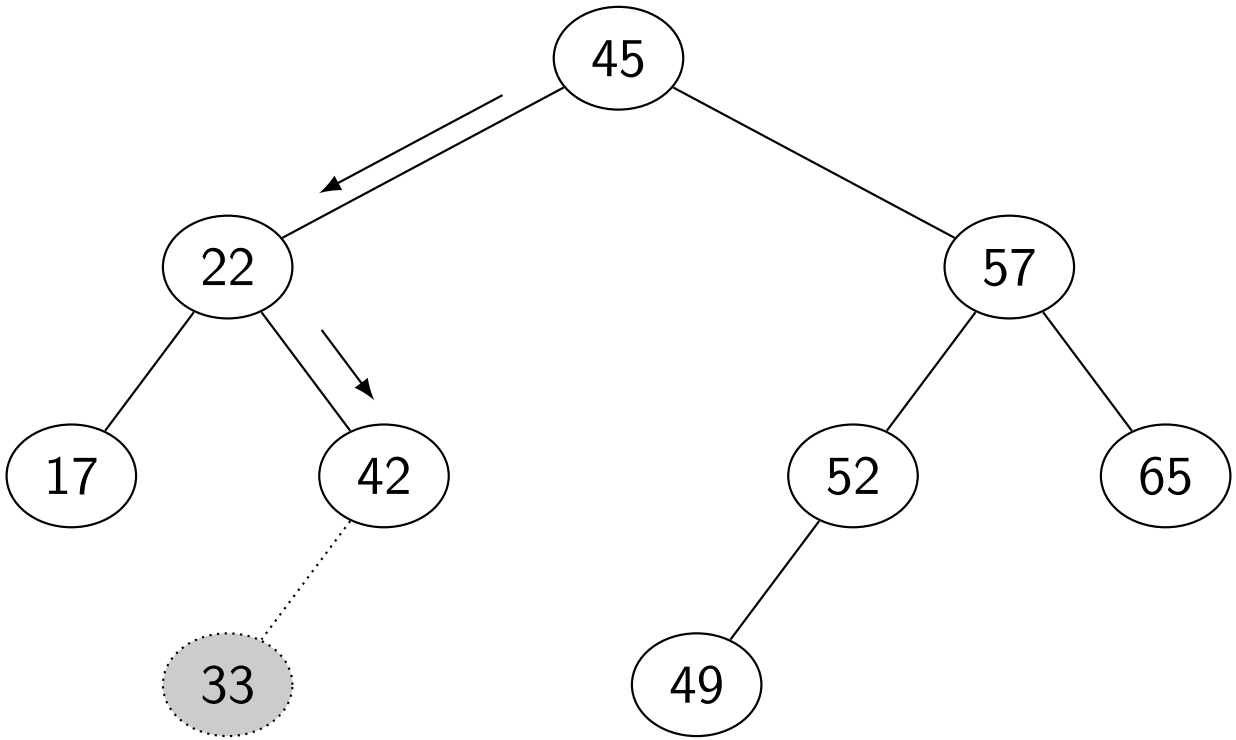
Ist dieser markierte Binärbaum sortiert?



SortedBinTree: Suchen eines Eintrags

```
// Repraesentiert einen sortierten markierten Binaerbaum
public class SortedBinTree {
  private TreeNode root;
  public SortedBinTree() {
    this.root = null;
  }
  // prueft, ob ein Element im Baum enthalten ist
  public boolean contains(int element) {
    return contains(this.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;
    }
  }
```

Beispiel: Einfügen von 33



SortedBinTree: Einfügen eines Eintrags I

Algorithmisches Vorgehen:

- Neue Knoten werden immer als Blätter eingefügt.
- Die Position des Blattes wird durch den Wert des neuen Eintrags festgelegt.
- Beim Aufbau eines Baumes ergibt der erste Eintrag die Wurzel.
- 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.

SortedBinTree: Einfügen eines Eintrags II

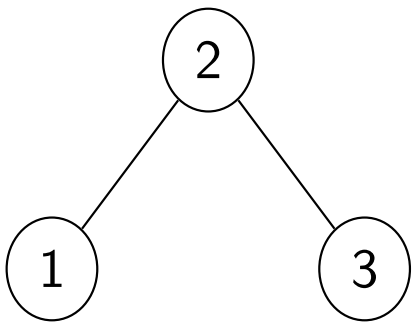
```
public void add(int element) {
    this.root = add(this.root, element);
}

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;
}
```

Bemerkungen

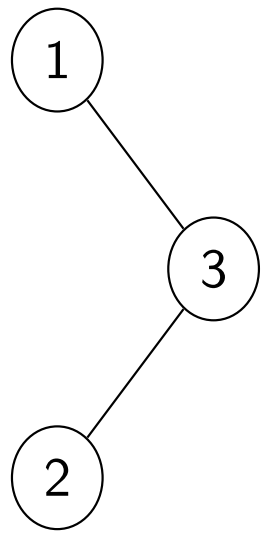
- Die Reihenfolge des Einfügens bestimmt das Aussehen des sortierten markierten Binärbaums:

  Reihenfolgen:

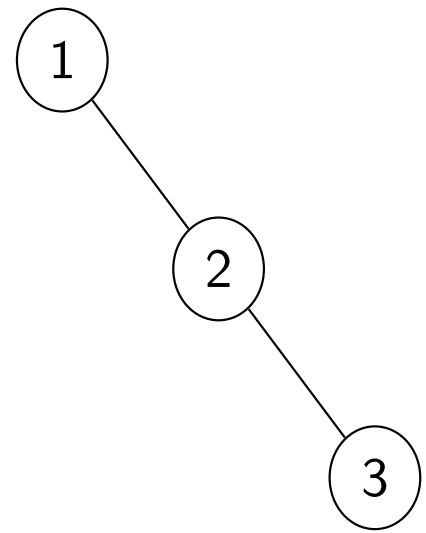
  2 ; 3 ; 1



  1 ; 3 ; 2

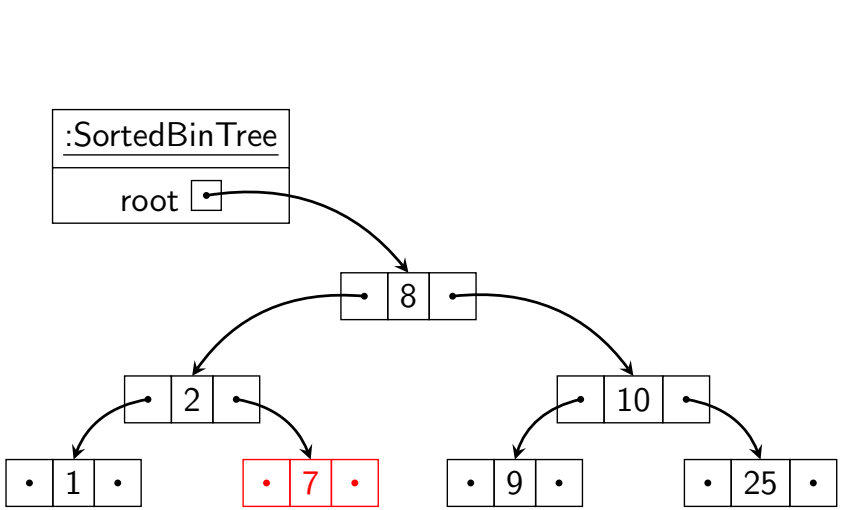


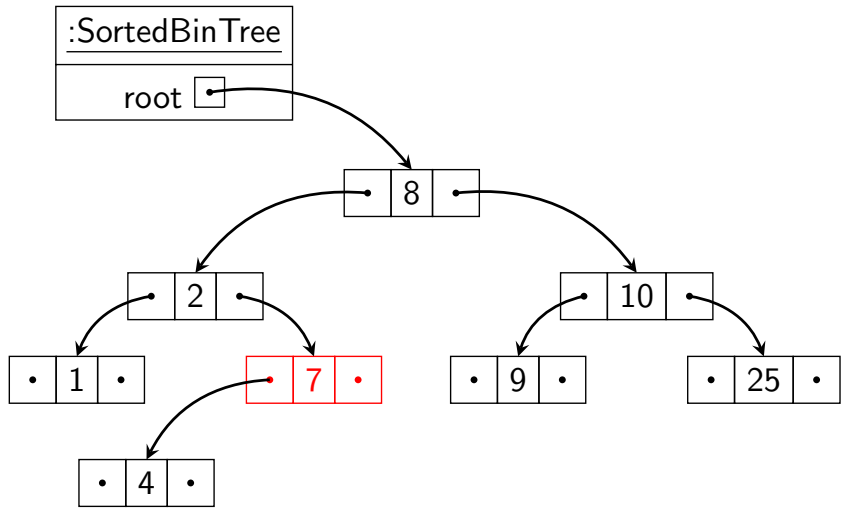
  1 ; 2 ; 3

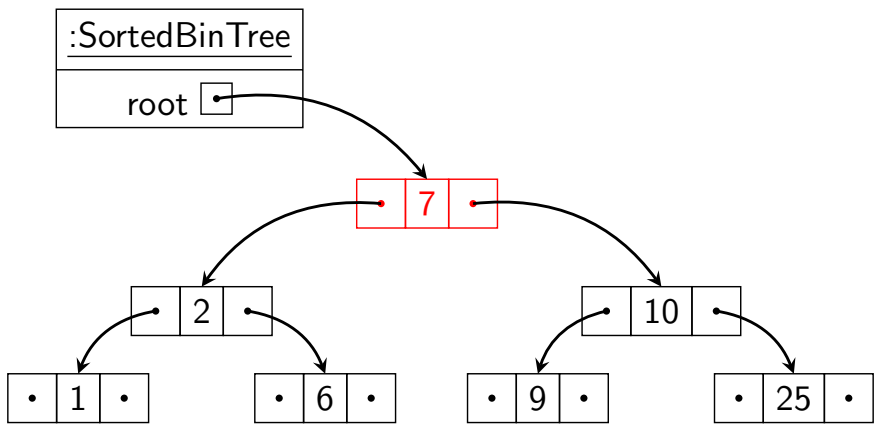


- Bei sortierter Einfügereihenfolge entartet der Baum zur linearen Liste.
- **Balancierte Bäume** stellen sicher, dass sich für jeden Knoten die Tiefe des linken Teilbaums und die Tiefe des rechten Teilbaums nur um einen bestimmte Wert unterscheidet.

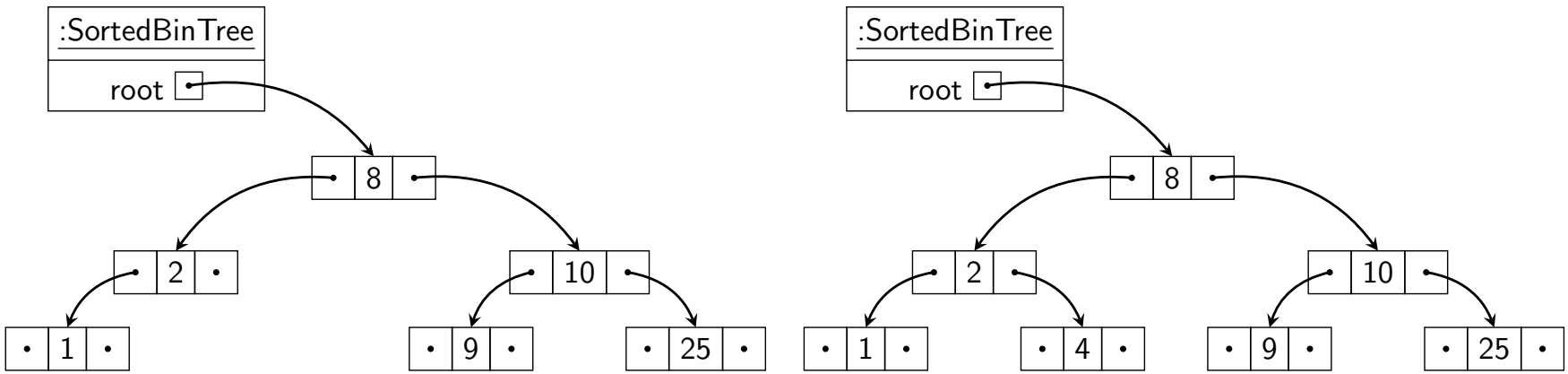
Wie sieht der Suchbaum aus, wenn wir 7 löschen?





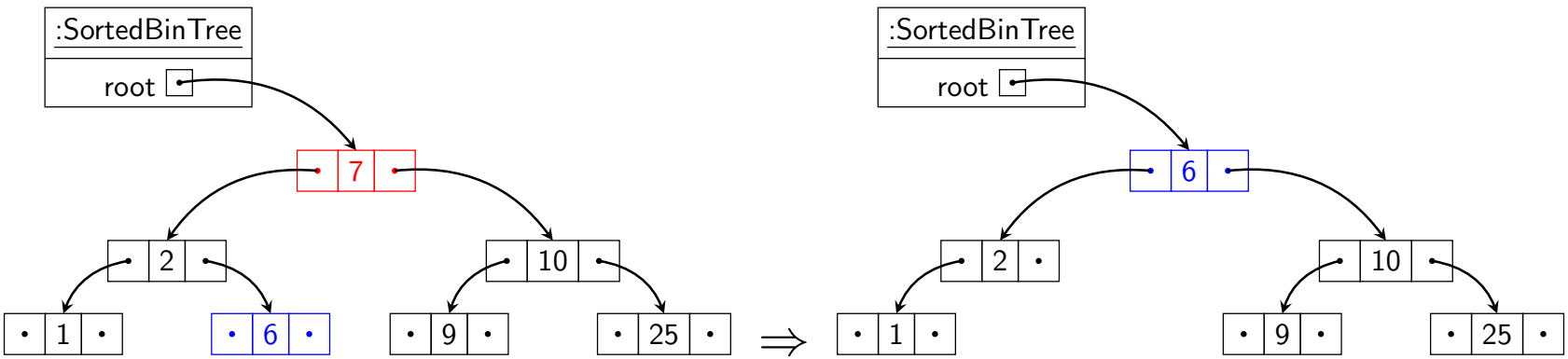


Nach dem Entfernen I



- **Knoten ohne Kinder**: Setze entsprechende Referenz des Elternknotens auf `null`
- **Knoten mit einem Kind**: Setze das eine Kind des zu löschenden Knoten an die Stelle des zu löschenden Knoten

Nach dem Entfernen II



- **Knoten mit zwei Kindern**: Ersetze durch Knoten mit der nächstgrößeren oder nächstkleineren Markierung aus Teilbaum , der durch den zu löschenden Knoten aufgespannt wird (**in-order** Nachbar)

Implementierung; Löschen I

```
/** 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 modifizierten Teilbaums
     */
private TreeNode remove(TreeNode node, int element) {
    if (node == null) {
        return null;
    }
    if (element < node.getMark()) {
        // Suche im linken Teilbaum rekursiv
        TreeNode newLeft = remove(node.getLeft(), element);
        node.setLeft(newLeft);
        return node;
    } else if (element > node.getMark()) {
        // Suche im rechten Teilbaum rekursiv
        TreeNode newRight = remove(node.getRight(), element);
        node.setRight(newRight);
        return node;
```

Implementierung; Löschen II

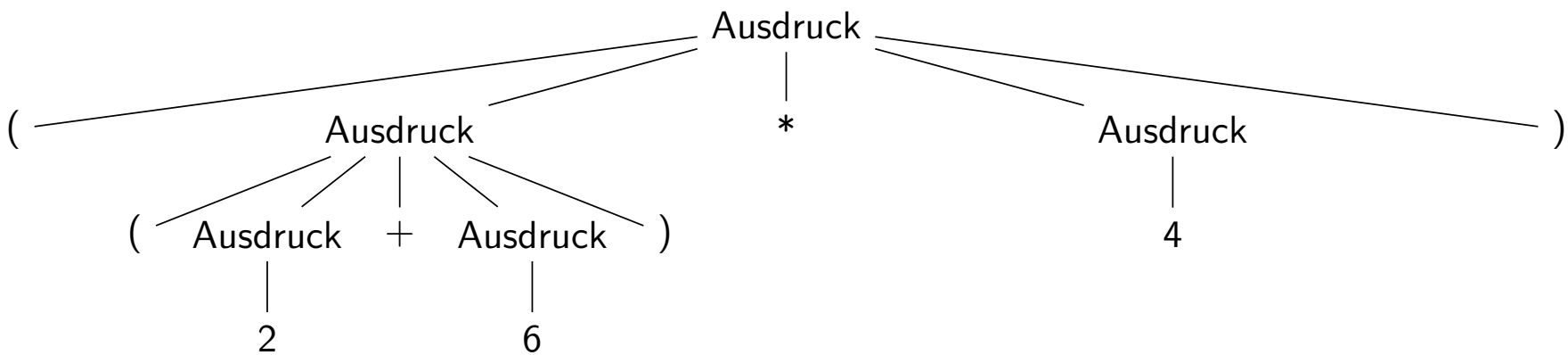
```
        } else {
            // Zu loeschendes Element gefunden
            if (node.getLeft() == null) {
                // wenn der linke Teilbaum leer ist, dann nur den rechten
                return node.getRight();
            } else if (node.getRight() == null) {
                // analog
                return node.getLeft();
            } else {
                // Knoten hat zwei Kinder;
                // suche den Knoten mit dem groessten Element
                // aus dem linken Teilbaum
                TreeNode maxNodeLeft = maxNode(node.getLeft());
                // Markierung von diesem Knoten nehmen wir fuer aktuellen:
                node.setMark(maxNodeLeft.getMark());
                // Dann loeschen wir das Element aus dem linken Teilbaum:
                node.setLeft(remove(node.getLeft(), node.getMark()));
                return node;
            }
        }
    }
```

Anwendungsbeispiel: Syntaxbäume

Arithmetischen Ausdrücke der Femto-Sprache:

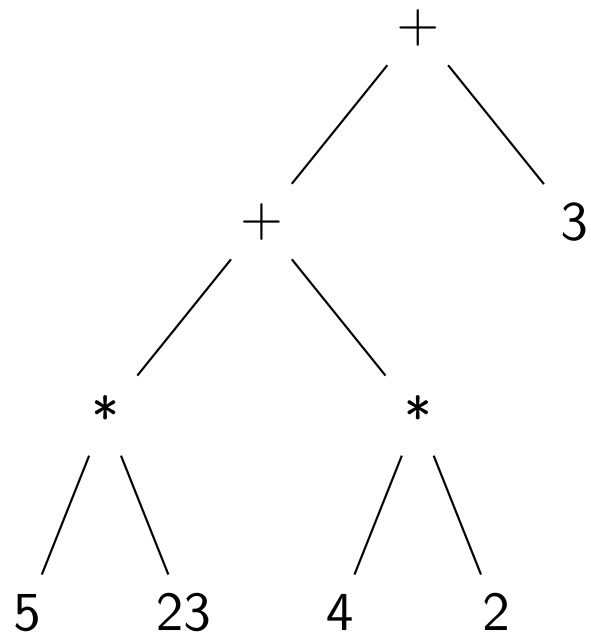
$$
\begin{aligned}
N &= \{\texttt{Ausdruck}\} \\
T &= \{\texttt{zahl}, \texttt{(}, \texttt{)}, \texttt{+}, \texttt{*}\} \\
\Pi &= \left\{ \begin{array}{lll}
\texttt{Ausdruck} & \rightarrow & \texttt{zahl} \\
& | & \texttt{(Ausdruck + Ausdruck)} \\
& | & \texttt{(Ausdruck * Ausdruck)}
\end{array} \right\}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

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



Abstrakter Syntaxbaum

- Ohne die Knoten, die für die weitere Verarbeitung überflüssig sind
- Struktur bestimmt Semantik!

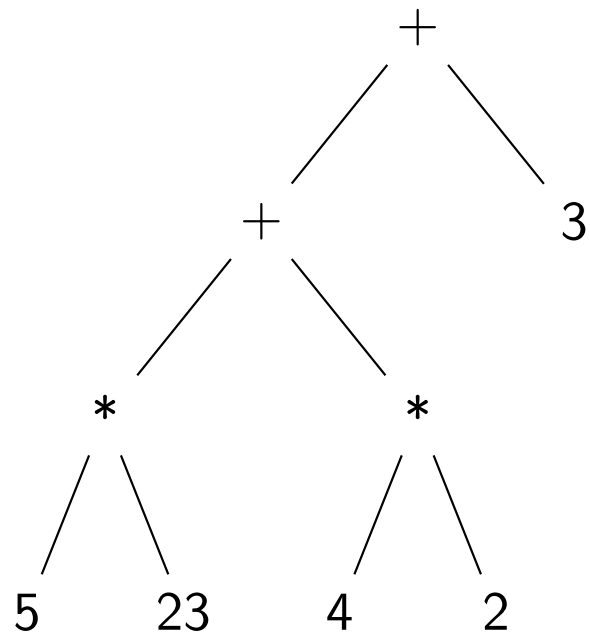


Traversieren von Bäumen

- Verschiedene Möglichkeiten die Knoten eines Baums zu durchlaufen
- Depth-first (in die Tiefe) vs. breadth-first (in die Breite)
- Hier: Drei Varianten für Tiefensuche

Vorordnung (preorder)

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

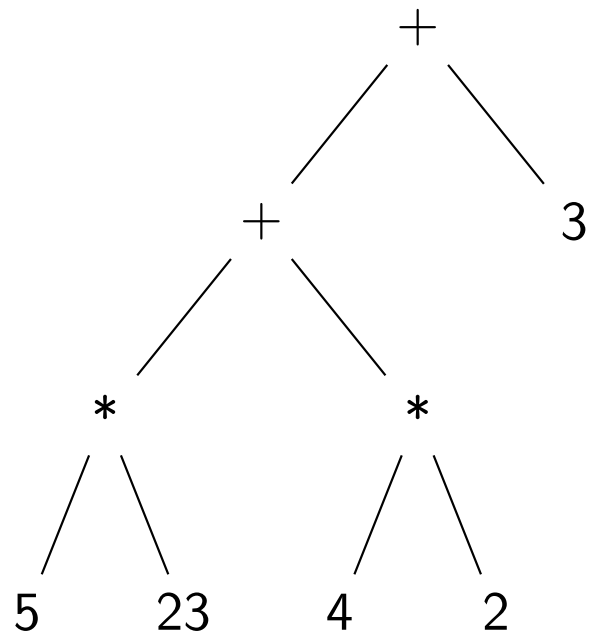


Ergebnis der Traversierung: `+ + * 5 23 * 4 2 3`

Diese Schreibweise eines arithmetischen Ausdrucks wird auch **Präfix-Notation** oder **polnische Notation** genannt.

Nachordnung (postorder)

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

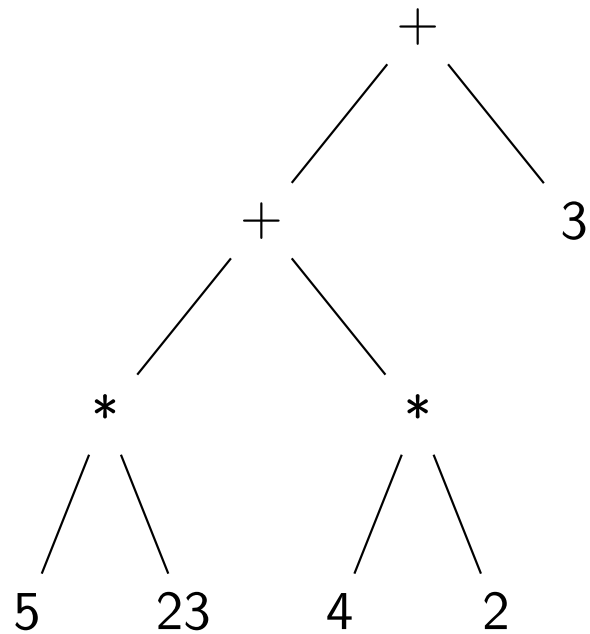


Ergebnis der Traversierung: `5 23 * 4 2 * + 3 +`

Diese Schreibweise eines arithmetischen Ausdrucks wird auch **Postfix-Notation** oder **umgekehrte polnische Notation** genannt.

Inordnung (inorder)

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



Ergebnis der Traversierung: `5 * 23 + 4 * 2 + 3`

OO-Modellierung von arithmetischen Ausdrücken

```
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 }
```

OO-Modellierung : Summe

```
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  }
```

OO-Modellierung: Konstantanten

```
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  }
```