Universität Bremen

# Praktische Informatik 2
# Bäume


Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

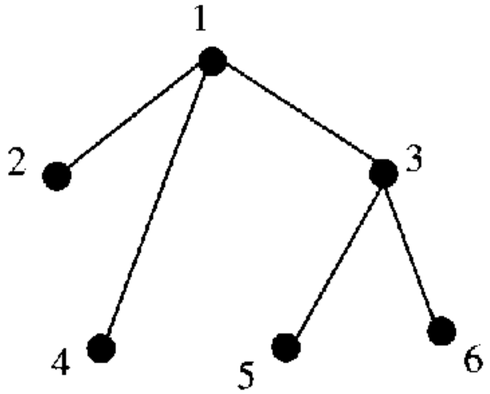
Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

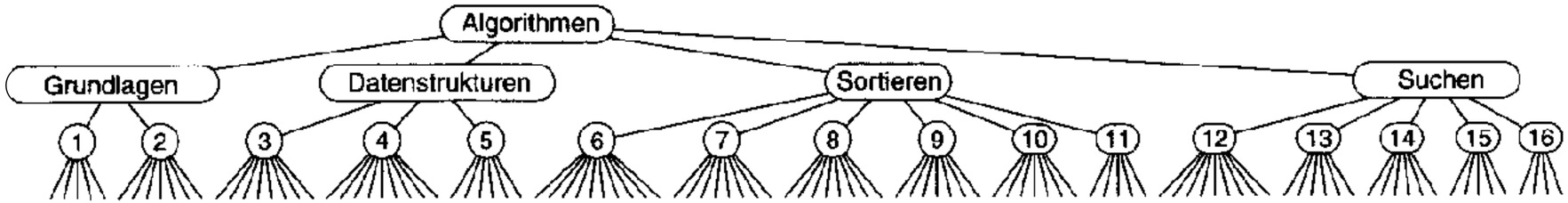


Bild: ©Michael Ihle

# Motivation

- In Arrays sind Einfügen und Löschen teure Operationen
- In verketteten Listen kann nur linear gesucht werden
- Man kann **Bäume** als verallgemeinerte Listenstruktur ansehen
  - Jeder Knoten hat nicht einen, sondern mehrere Nachfolger
- Beispiel: Teile und Kapitel eines Buchs

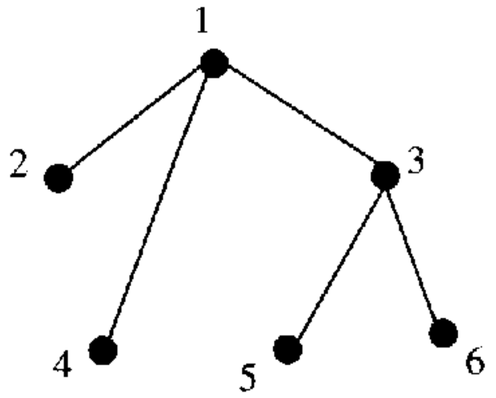


Universität Bremen

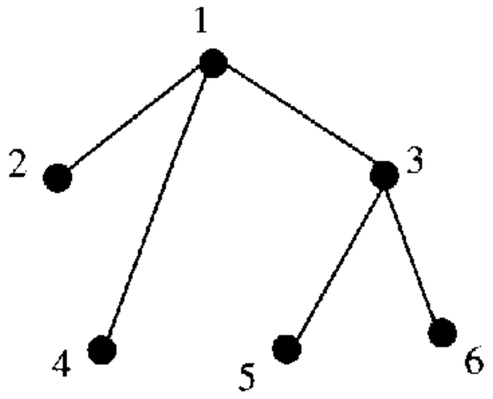
# Begriffe: Knoten

- Kann einen Namen haben (**Etikett**, **Label**) und/oder andere Informationen tragen
- Kann mehrere **untergeordnete Knoten** haben (**Kindknoten**, **Nachfolger**)
- Kann maximal einen **übergeordneten Knoten** haben (**Elternknoten**, **Vorgänger**)
- Hat ein Knoten keinen übergeordneten Knoten, ist er die **Wurzel** des Baums
- Der **Verzweigungsgrad** eines Knotens ist die Anzahl seiner Kindknoten
- Hat ein Knoten keine Nachfolger, ist er ein **Blatt** (**Terminalknoten**)



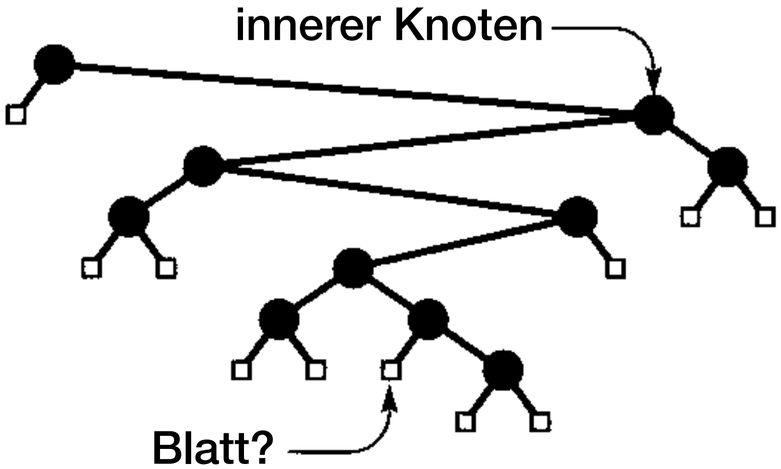
# Begriffe

- **Kante**: Verbindet jeweils zwei Knoten (Eltern- mit Kindknoten)
- **Pfad**: Folge von unterschiedlichen Knoten, wobei aufeinander folgende Knoten durch Kanten verbunden sind
- **Baum**
  - Nichtleere Sammlung von Knoten und Kanten
  - Jeweils zwei Knoten sind genau durch einen Pfad verbunden
- **Wald**: Mehrere nicht miteinander verbundene Bäume

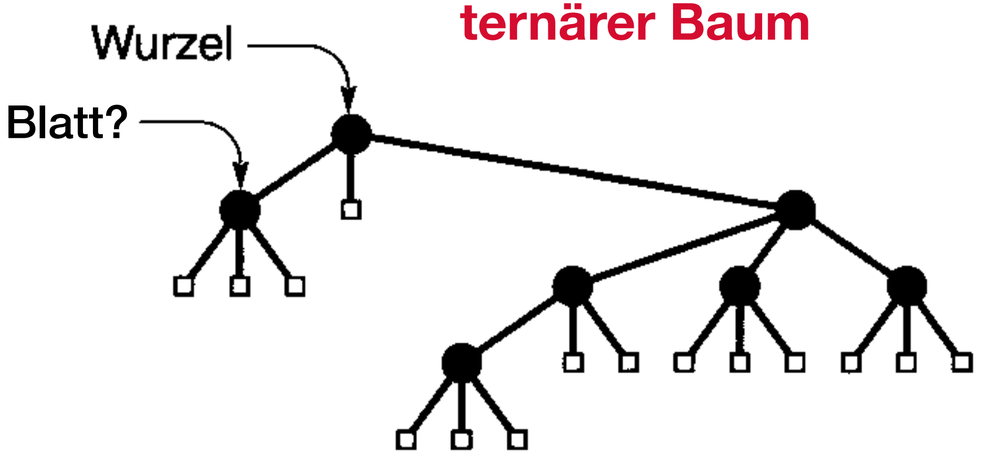


# Begriffe

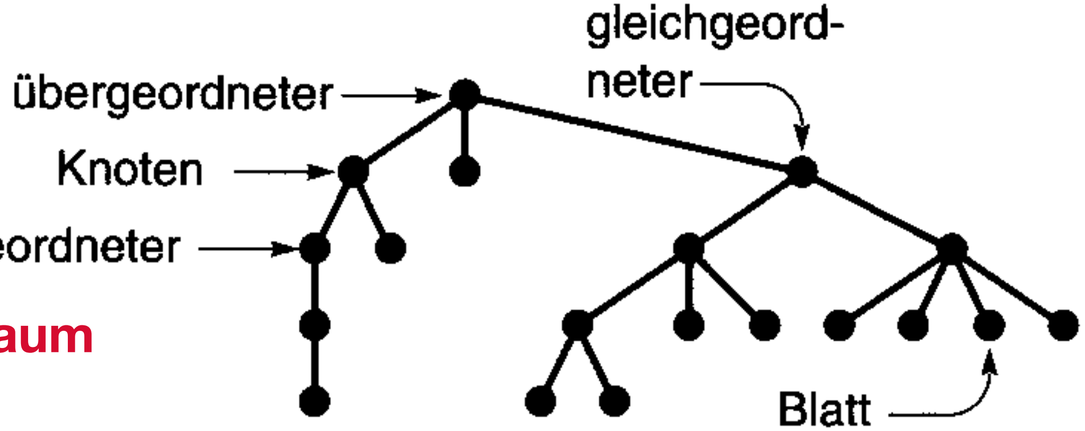
**binärer Baum**



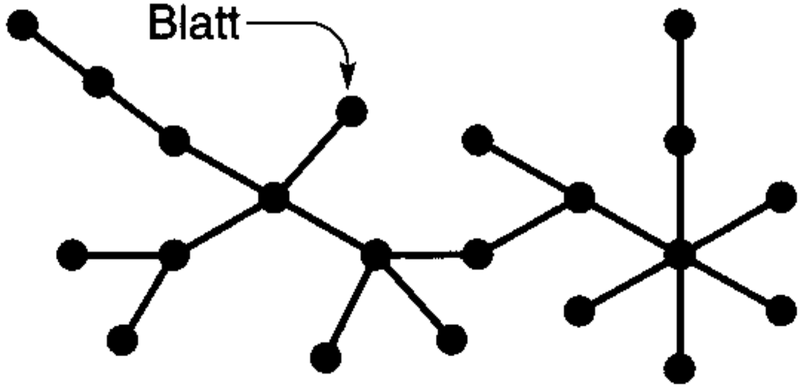

**ternärer Baum**




**Wurzelbaum**

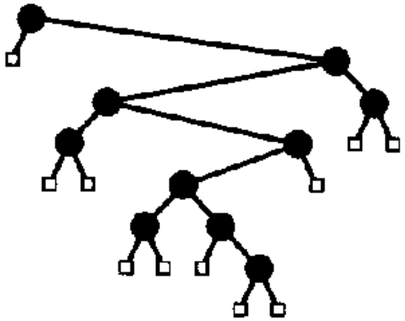



**freier Baum**

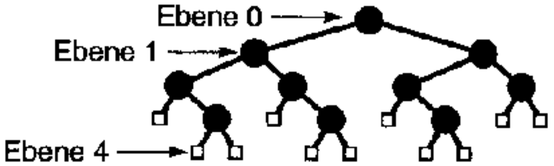


Universität Bremen

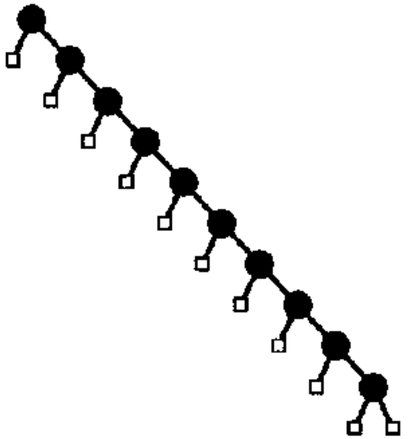
# Begriffe



- **Ordnung eines Baums**: Der maximale Verzweigungsgrad seiner Knoten
- **Binärbaum**: Die Ordnung des Baums ist 2
- **Geordneter Baum**: Zwischen den Kindern der Knoten ist eine **Ordnung** definiert, z.B. erster, zweiter, dritter … Kindknoten



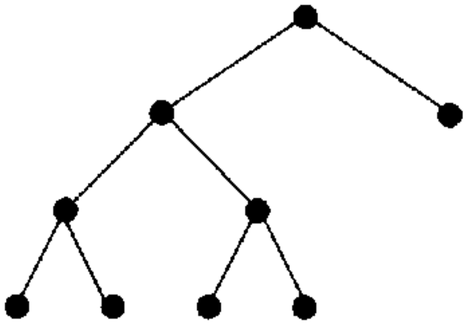

- **Tiefe eines Knotens**: Abstand zwischen Wurzel und Knoten (Anzahl der Kanten)
  - Die Wurzel hat die Tiefe 0
  - Knoten mit derselben Tiefe gehören zur selben **Ebene** (**Niveau**)



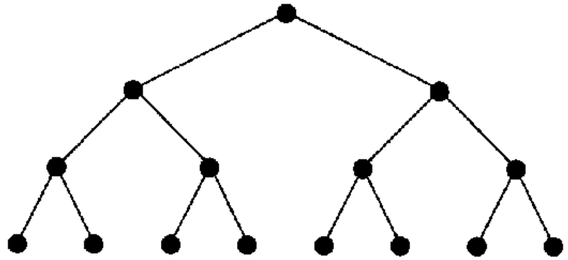
- **Höhe eines Baums**: Die maximale Tiefe eines Knotens im Baum

Universität Bremen

# Definitionen und Eigenschaften von Bäumen



- **Voller Baum**: Verzweigungsgrad aller inneren Knoten entspricht der Ordnung des Baums

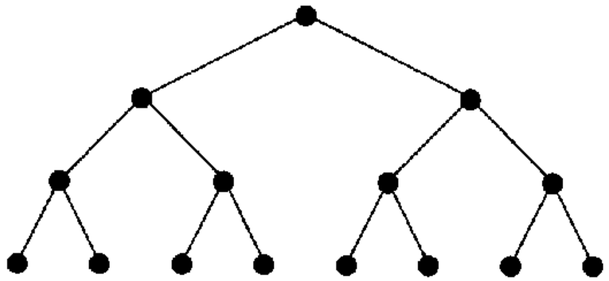


- **Vollständiger Baum**: Tiefe aller Blätter entspricht der Höhe des Baums
- Ein Baum mit **n** Knoten hat **n-1** Kanten
  - Zu jedem Knoten führt eine Kante, nur zur Wurzel nicht

Universität Bremen

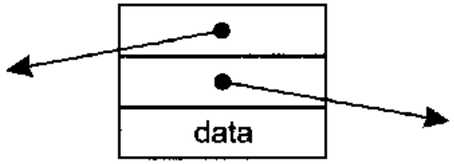
# Eigenschaften von vollständigen Binärbäumen

- Ein vollständiger Binärbaum der Höhe **n** hat **$2^n$** Blätter
  - Ein Binärbaum, der nur aus der Wurzel besteht, hat ein Blatt
  - Ein vollständiger Binärbaum der Höhe **n+1** hat doppelt so viele Blätter wie einer der Höhe **n**

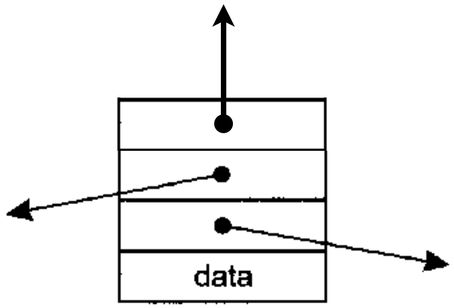


- Ein vollständiger Binärbaum der Höhe **n** hat **$2^{n+1} - 1$** Knoten
  - Ein Binärbaum, der nur aus der Wurzel besteht, hat einen Knoten
  - Mit jeder Ebene kommt ein Knoten mehr hinzu, als schon im Baum vorhanden sind: **knoten(n+1) = 2 × knoten(n) + 1**

# Implementierung von Bäumen

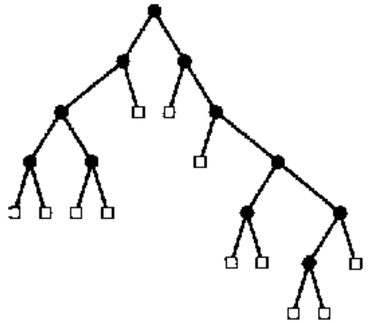


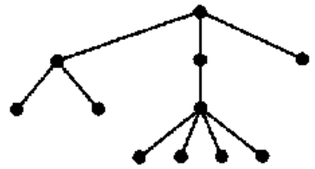

**Knoten**

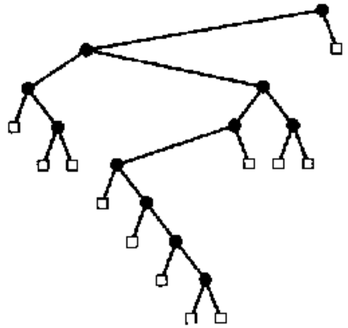


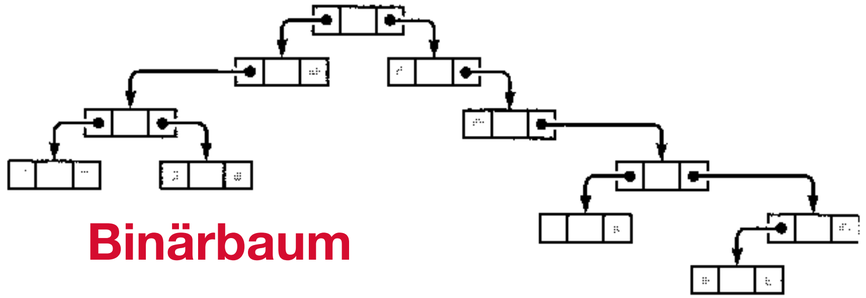

**Knoten mit Elternreferenz**

**Baum**

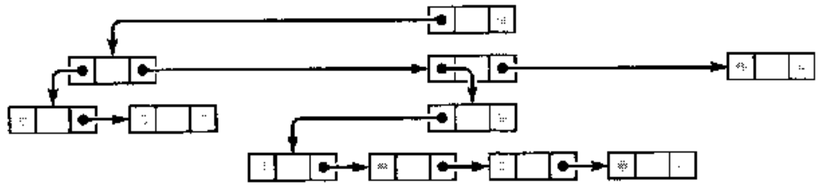


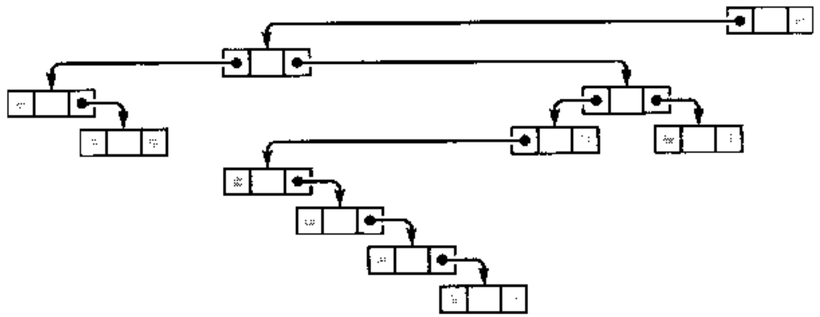






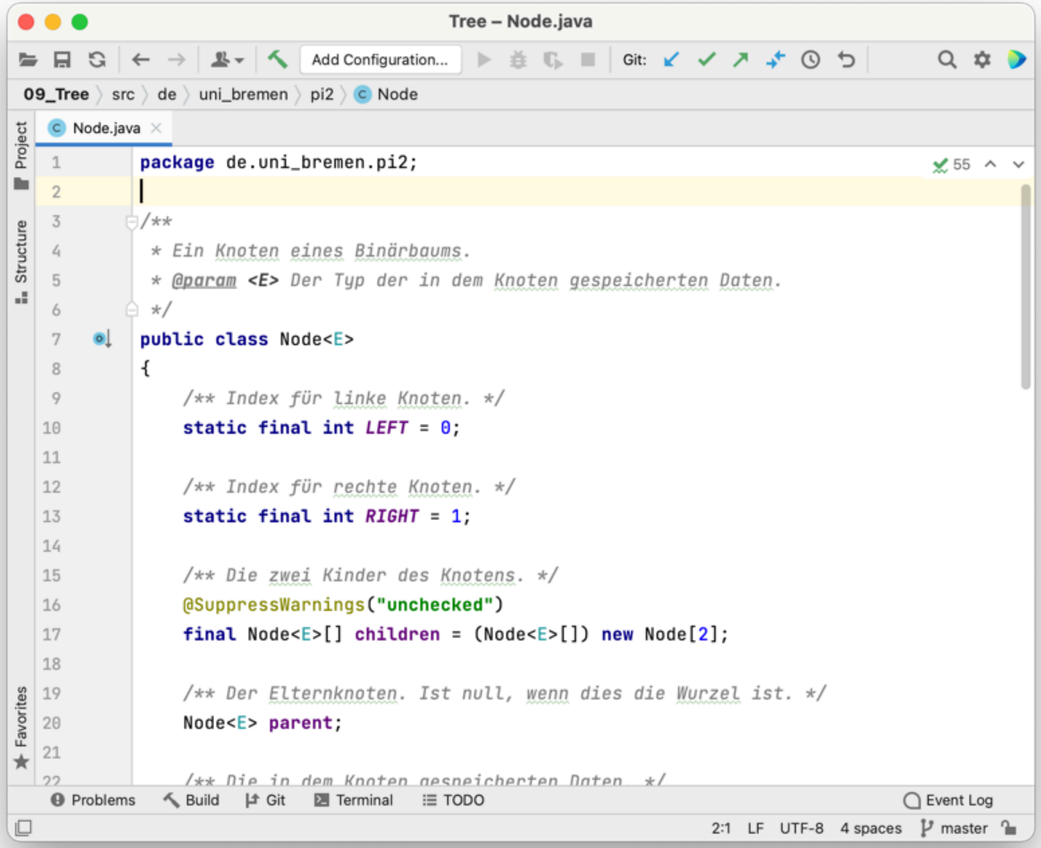
**Binärbaum**





Universität Bremen

# Binärbaum: Demo



Universität Bremen

# Durchlaufen eines Baums

- Knoten können in verschiedenen Reihenfolgen durchlaufen werden
- Beim Durchlaufen können Operationen auf den in den Knoten enthaltenen Daten durchgeführt werden
- Mögliche Reihenfolgen
  - Präorder-Sequenz
  - Postorder-Sequenz
  - Inorder-Sequenz
  - Level-Order-Sequenz

```
public void someOrder(final Consumer<E> action)
{
    someOrder(root, action);
}
```

```
data -> System.out.print(data + " ")
```

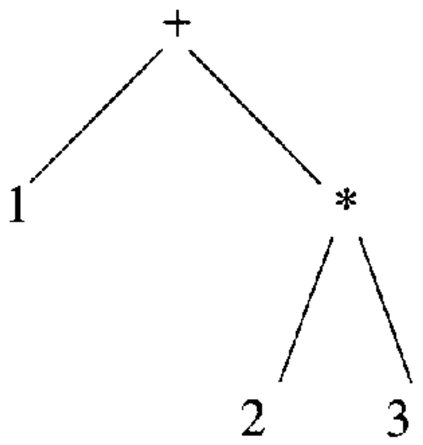
- Kindknoten werden dabei immer von links nach rechts durchlaufen

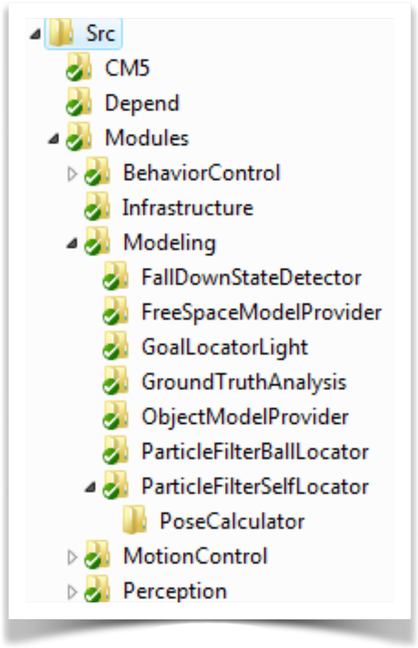
# Präorder-Sequenz

- In jedem Knoten wird zuerst der Knoten selbst verarbeitet, dann die Kindknoten

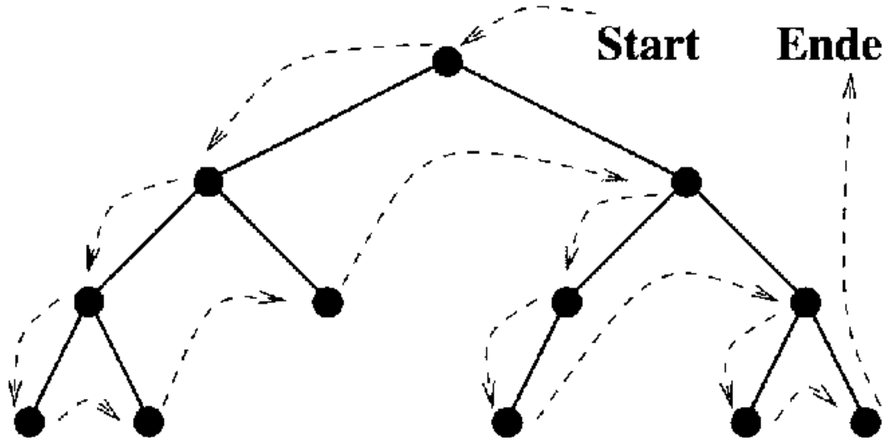
- Beispiel

  - + 1 * 2 3




```
private void preorder(final Node<E> node,
        final Consumer<E> action)
{
    if (node != null) {
        action.accept(node.data);
        preorder(node.children[LEFT], action);
        preorder(node.children[RIGHT], action);
    }
}
```



# Postorder-Sequenz

- In jedem Knoten werden zuerst die Kindknoten verarbeitet, dann der Knoten selbst

- Beispiel
  - 1 2 3 * +

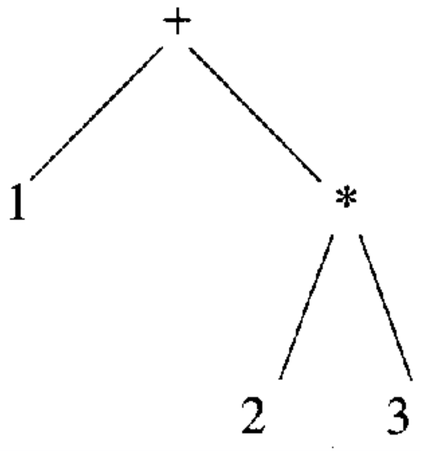
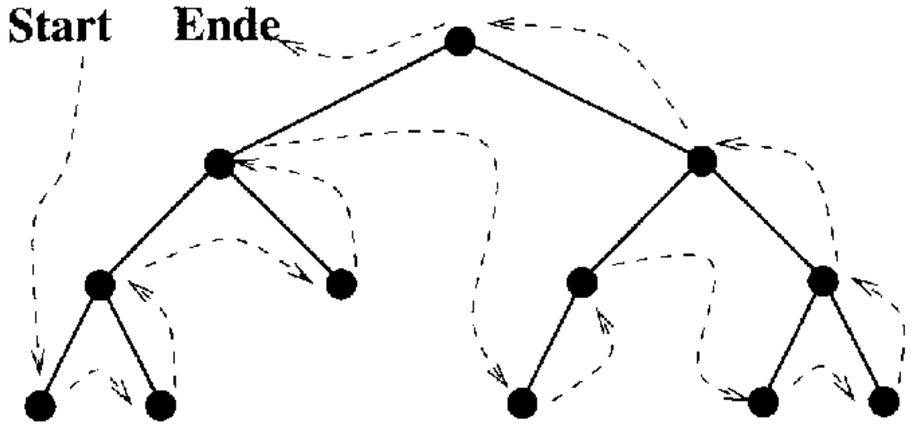




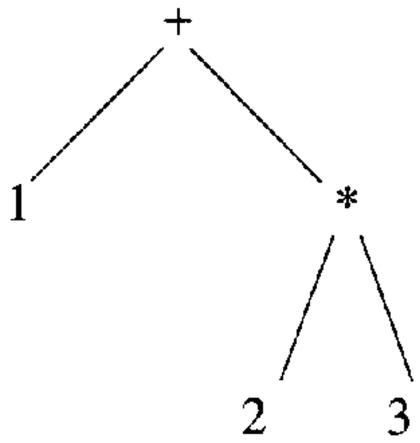
Foto: Holger Weihe

```
private void postorder(final Node<E> node,
        final Consumer<E> action)
{
    if (node != null) {
        postorder(node.children[LEFT], action);
        postorder(node.children[RIGHT], action);
        action.accept(node.data);
    }
}
```

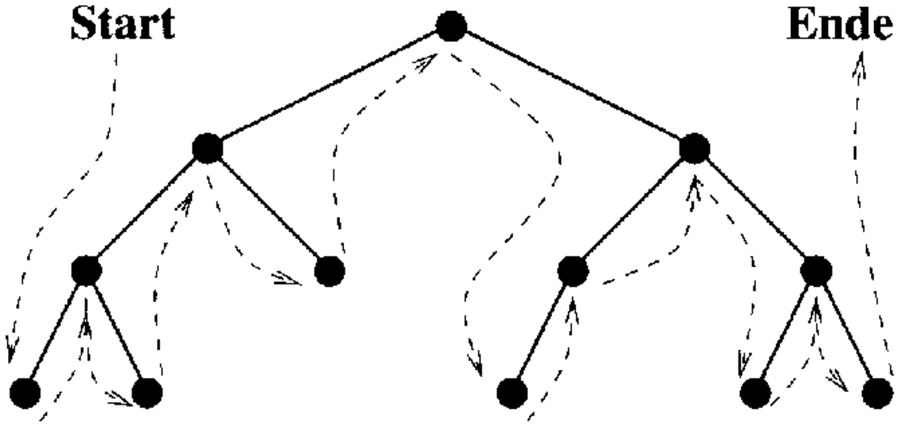


# Inorder-Sequenz

- In jedem Knoten wird zuerst der linke Kindknoten verarbeitet, dann der Knoten selbst, dann der rechte Kindknoten
- Nur bei Binärbäumen
- Beispiel
  - 1 + 2 * 3

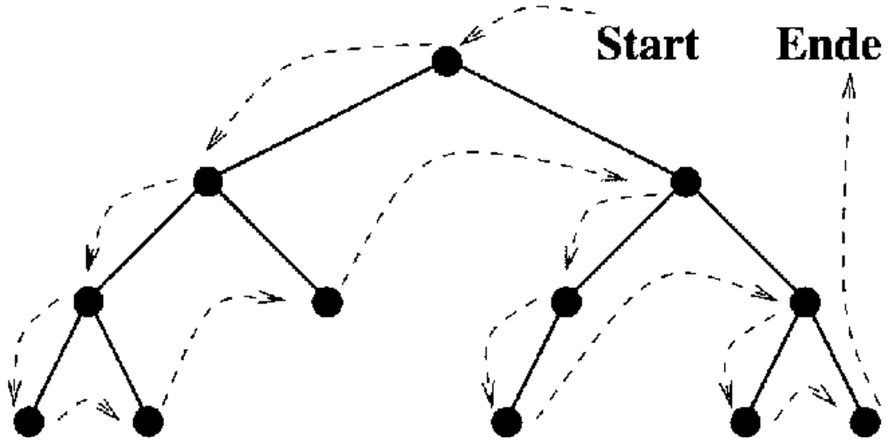



```
private void inorder(final Node<E> node,
        final Consumer<E> action)
{
    if (node != null) {
        inorder(node.children[LEFT], action);
        action.accept(node.data);
        inorder(node.children[RIGHT], action);
    }
}
```



# Präorder-Sequenz ohne Rekursion

- Statt Rekursion wird ein **Stapel** verwendet, um die offenen Knoten zu speichern
- Der Stapel ist vom Typ **last-in-first-out**




```
public void preorder2(
        final Consumer<E> action)
{
    final Stack<Node<E>> stack
            = new Stack<>();
    stack.push(root);
    while (!stack.isEmpty()) {
        final Node<E> node = stack.pop();
        if (node != null) {
            action.accept(node.data);
            stack.push(node.children[RIGHT]);
            stack.push(node.children[LEFT]);
        }
    }
}
```

Universität Bremen

# Präorder-Sequenz ohne Rekursion: Iterator

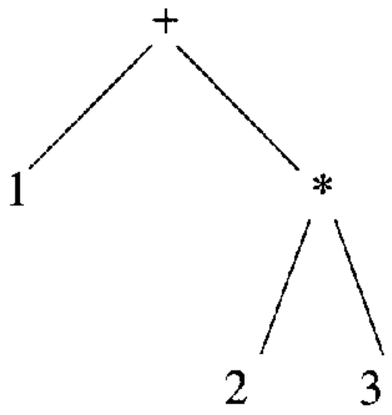
```
public Iterator<E> iterator()
{
    final Stack<Node<E>> stack = new Stack<>();
    stack.push(root);

    return new Iterator<E>() {
       public boolean hasNext() {
            while (!stack.isEmpty()
                    && stack.peek() == null) {
                stack.pop();
            }
            return !stack.isEmpty();
        }
```

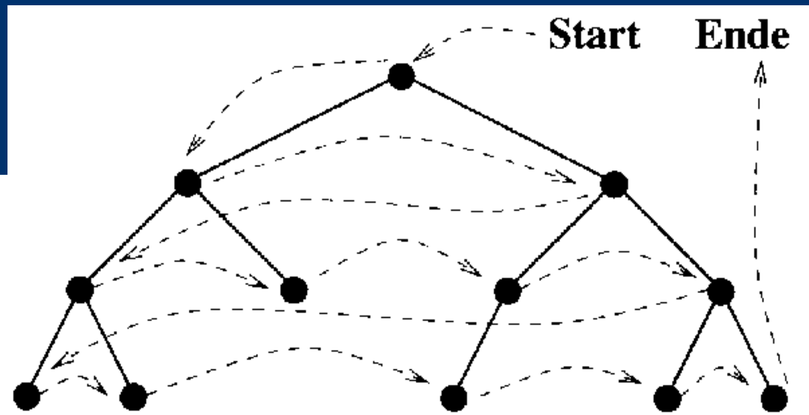
```
        public E next() {
            if (hasNext()) {
                final Node<E> node = stack.pop();
                stack.push(node.children[RIGHT]);
                stack.push(node.children[LEFT]);
                return node.data;
            }
            else {
                throw new NoSuchElementException();
            }
        }
    };
}
```

# Level-Order-Sequenz

- Die Knoten werden ihrer Ebene nach durchlaufen und innerhalb jeder Ebene von links nach rechts
- Nutzung eines **first-in-first-out** Stapels (Warteschlange)
- Beispiel
  - + 1 * 2 3

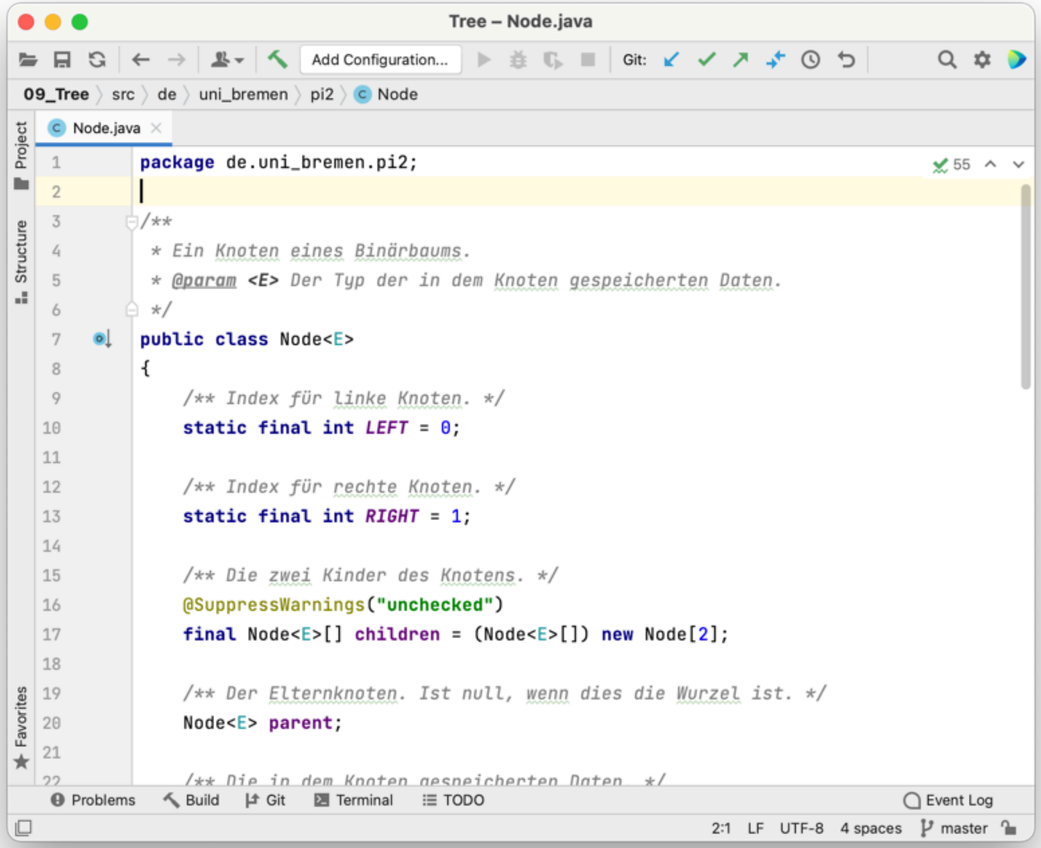



```
public void levelOrder(final Consumer<E> action)
{
    final Queue<Node<E>> stack
        = new LinkedList<>();
    stack.add(root);
    while (!stack.isEmpty()) {
        final Node<E> node = stack.poll();
        if (node != null) {
            action.accept(node.data);
            stack.add(node.children[LEFT]);
            stack.add(node.children[RIGHT]);
        }
    }
}
```



Universität Bremen

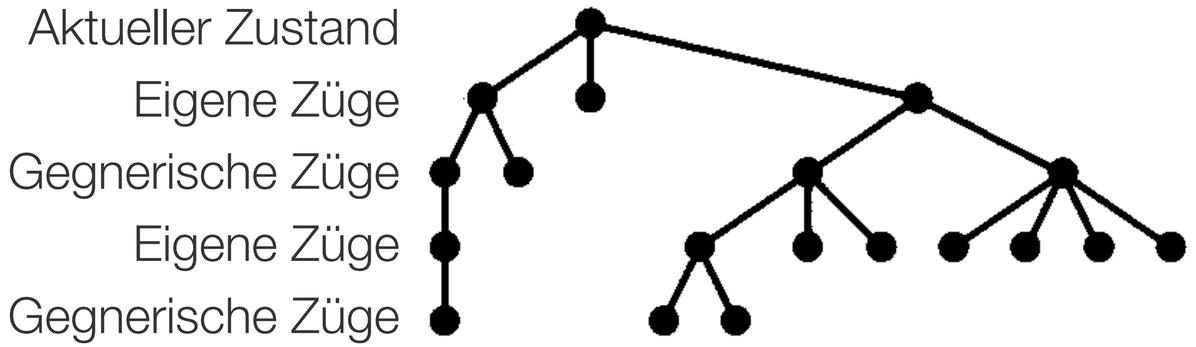
# Bäume durchlaufen: Demo



Universität Bremen

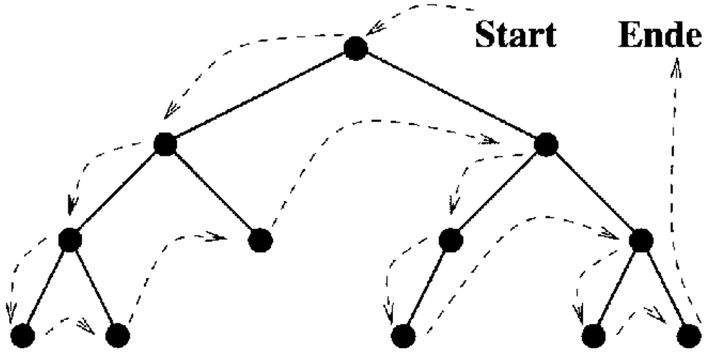
# Spielprobleme

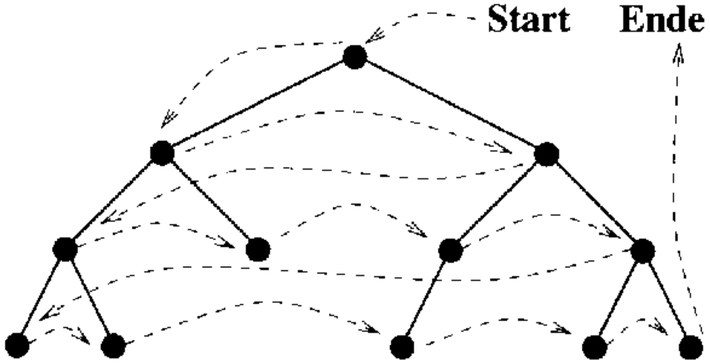
- Bei Spielproblemen (z.B. Schach) werden mögliche Zugfolgen durchprobiert und das Ergebnis bewertet
- Dabei wird unter allen eigenen Zügen der mit der besten Bewertung gesucht (Bewertung **maximieren**)
- Allerdings muss davon ausgegangen werden, dass die Gegner:in den für einen selbst ungünstigsten Zug auswählen wird (Bewertung **minimieren**)
- Daher spricht man dabei von einem **MiniMax**-Problem



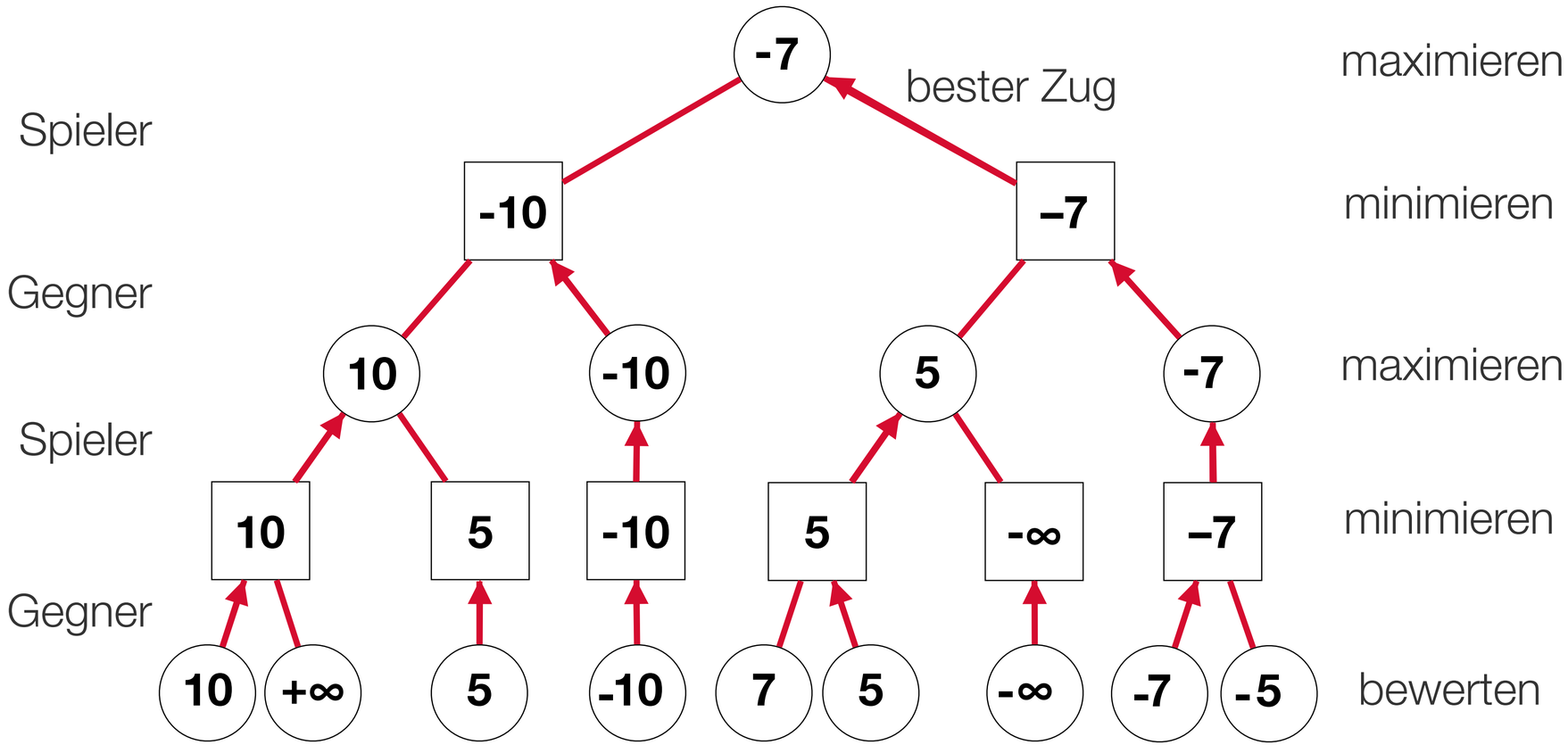
# Spielprobleme: Tiefen-/Breitensuche

- Suche und verwende den besten gefundenen Zug
- **Tiefensuche**: Durchsuche den Baum der möglichen Zugfolgen bis zu einer bestimmten Tiefe
- **Breitensuche**: Durchsuche den Baum der möglichen Zugfolgen in Level-Order-Reihenfolge so lange, bis die Zeit um ist
- **Iterative Tiefensuche**: Wie Tiefensuche, aber wenn noch Zeit übrig ist, erhöhe die Tiefe um 1 und beginne von vorne



# MiniMax-Algorithmus



Universität Bremen

# MiniMax-Algorithmus: Beispiel

```
public Zug maximum(final int höhe)
{
    if (höhe == 0) {
        return new Zug(bewerten());
    }
    Zug bester = new Zug(Integer.MIN_VALUE);
    for (final var zug : möglicheZüge()) {
        macheEigenenZug(zug);
        final int bewertung = gewonnen()
                ? guteBewertung(höhe)
                : minimum(höhe - 1).bewertung;
        if (bewertung > bester.bewertung) {
            bester = new Zug(zug, bewertung);
        }
        rückgängig(zug);
    }
    return bester;
}
```

```
public Zug minimum(final int höhe)
{
    if (höhe == 0) {
        return new Zug(bewerten());
    }
    Zug schlechtester = new Zug(Integer.MAX_VALUE);
    for (final var zug : möglicheZüge()) {
        macheGegnerZug(zug);
        final int bewertung = verloren()
                ? schlechteBewertung(höhe)
                : maximum(höhe - 1).bewertung;
        if (zug.bewertung < schlechtester.bewertung) {
            schlechtester = new Zug(zug, bewertung);
        }
        rückgängig(zug);
    }
    return schlechtester;
}
```

Universität Bremen

# NegaMax-Algorithmus: Beispiel

```
public Zug negaMax(final Spieler spieler, int höhe)
{
    if (höhe == 0) {
        return new Zug(bewerten(spieler));
    }
    Zug bester = new Zug(-Integer.MAX_VALUE);
    for (final var zug : möglicheZüge()) {
        macheZug(zug, spieler);
        final int bewertung = gewonnen(spieler)
                ? guteBewertung(höhe)
                : -negaMax(andere(spieler), höhe - 1).bewertung;
        if (bewertung > bester.bewertung) {
            bester = new Zug(zug, bewertung);
        }
        rückgängig(zug);
    }
    return bester;
}
```

Universität Bremen

# Zusammenfassung der Konzepte

- **Baum**, **Knoten** (**Wurzel**, **innerer Knoten**, **Blatt**)
- **Baum-Höhe** und **Knoten-Tiefe**
- **Baum-Ordnung** und **Knoten-Verzweigungsgrad**
- **Voller Baum** und **vollständiger Baum**
- **Präorder**, **Postorder**, **Inorder** und **Level-Order**
- **MiniMax**-Problem