13. Bäume: effektives Suchen und Sortieren

Java-Beispiele:

Baum.java

Traverse.java

TraverseTest.java

Schwerpunkte

- Aufgabe und Vorteile von Bäumen
- Sortieren mit Bäumen
- Ausgabealgorithmen:
  - Preorder
  - Postorder
  - Inorder
- AVL-Bäume: ausgeglichene Bäume

Aufgabe von Bäumen

- **Schnelles Suchen und Sortieren**
- **Repräsentation zusammengesetzter Daten:**

  Datenmengen besitzen hierarchischen Aufbau

  → Repräsentation als Baum
  (z.B. Syntax eines Programms)

→ Ohne Bäume oft kein sinnvolles Programmieren

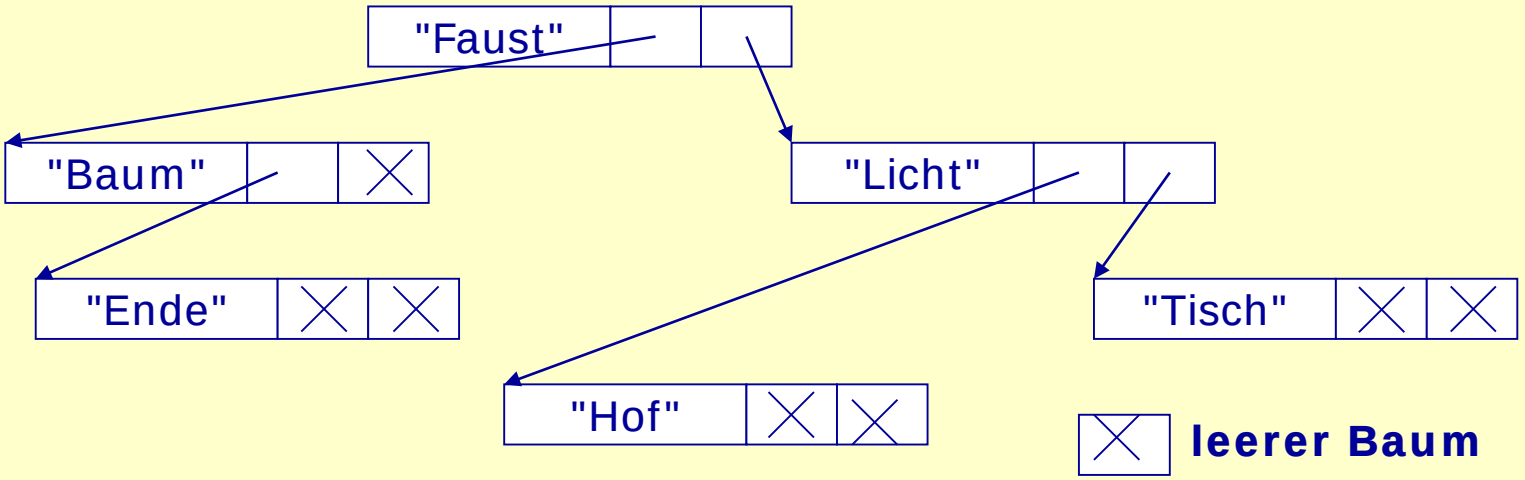
Definition: b i n ä r e r Baum

Ein binärer Baum ist e n t w e d e r

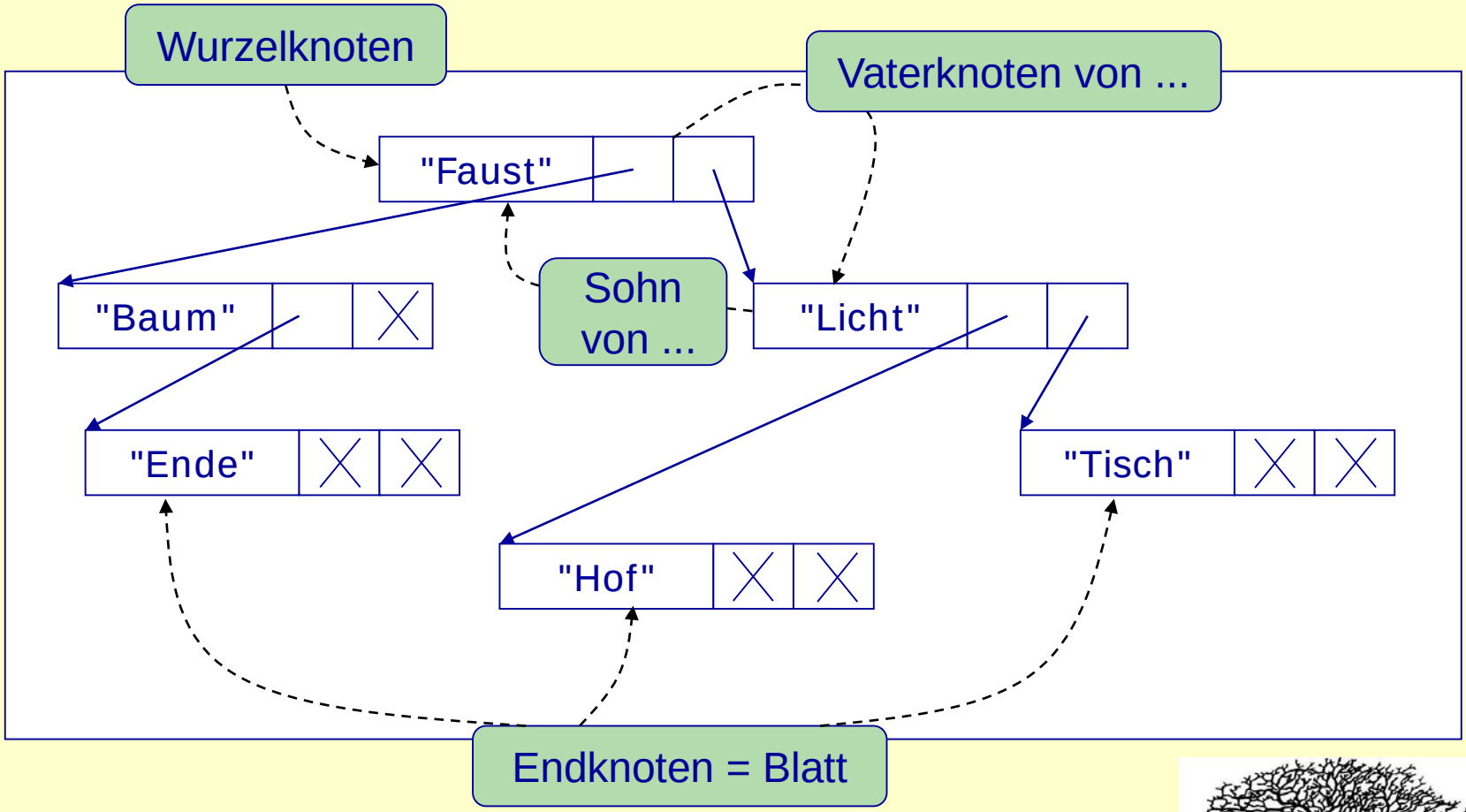
- leer (ohne Information) o d e r
- ein Knoten mit einem Inhalt (die Information) und zwei binären Bäumen (linker und rechter Teilbaum)

Rekursive Definition

Beispiel: nicht-leerer Baum



Begriffe



- Jeder Vater kann mehrere Söhne haben (0, 1, 2)
- Jeder Sohn hat genau einen Vater (bis auf Wurzel)

Wurzel unten

Bäume als abstrakte Datentypen

(z.B. Inhalt vom Typ 'String')

new1:		→	Baum
new2:	String	→	Baum
new3:	String x Baum x Baum	→	Baum
isEmpty:	Baum	→	boolean
left:	Baum	→	Baum
right:	Baum	→	Baum
value:	Baum	→	String **u.a.**

Konstruktoren:

new1: leerer Baum erzeugt
new2: Baum mit einem (Wurzel-)Knoten erzeugt
(linker und rechter Teilbaum leer)
new3: beliebiger Baum erzeugt

Implementation: Baum.java

(1. Teil der Klasse: new1/2/3 → Baum(...))

Aufgabe von Bäumen

- **Schnelles Suchen und Sortieren**
- **Repräsentation zusammengesetzter Daten:**

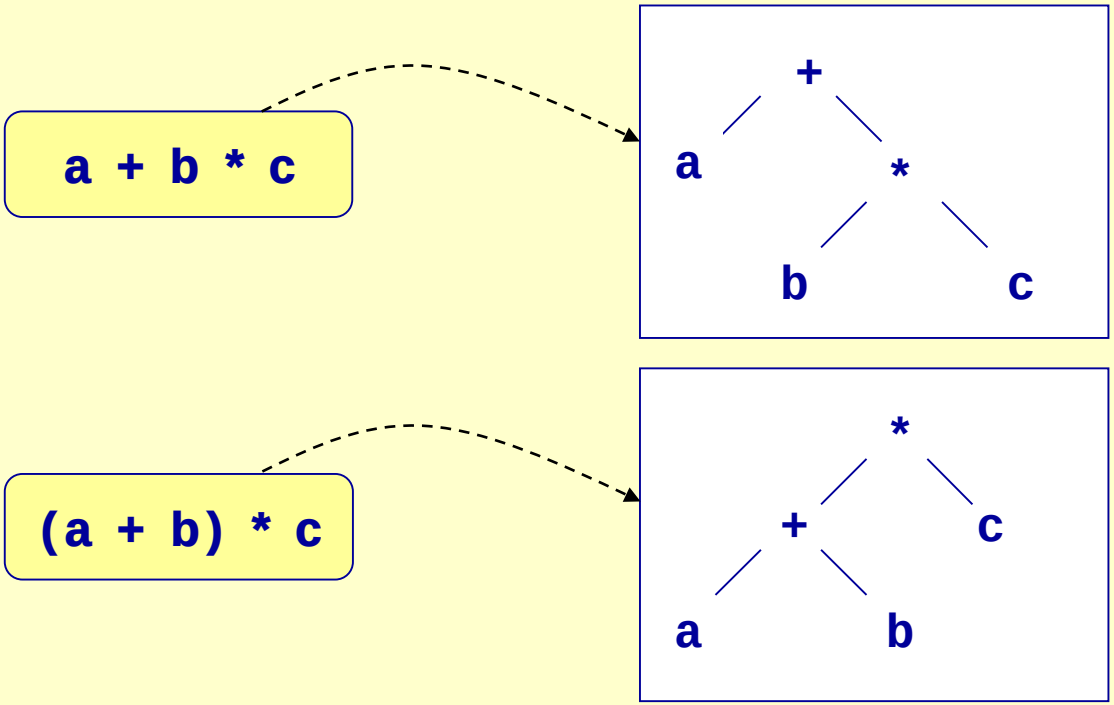
  Datenmengen besitzen hierarchischen Aufbau

  → Repräsentation als Baum

Repräsentation zusammengesetzter Daten

z. B. Programmiersprachen: Ausdrücke

→ **Bäume repräsentieren Vorrangregeln der Operatoren** (Syntaxbäume)



Wurzel: Operator
Blätter: Operanden

Aufgabe von Bäumen

- **Schnelles Suchen und Sortieren**
- **Repräsentation zusammengesetzter Daten:**

  Datenmengen besitzen hierarchischen Aufbau

  → Repräsentation als Baum

Schnelles Suchen und Sortieren

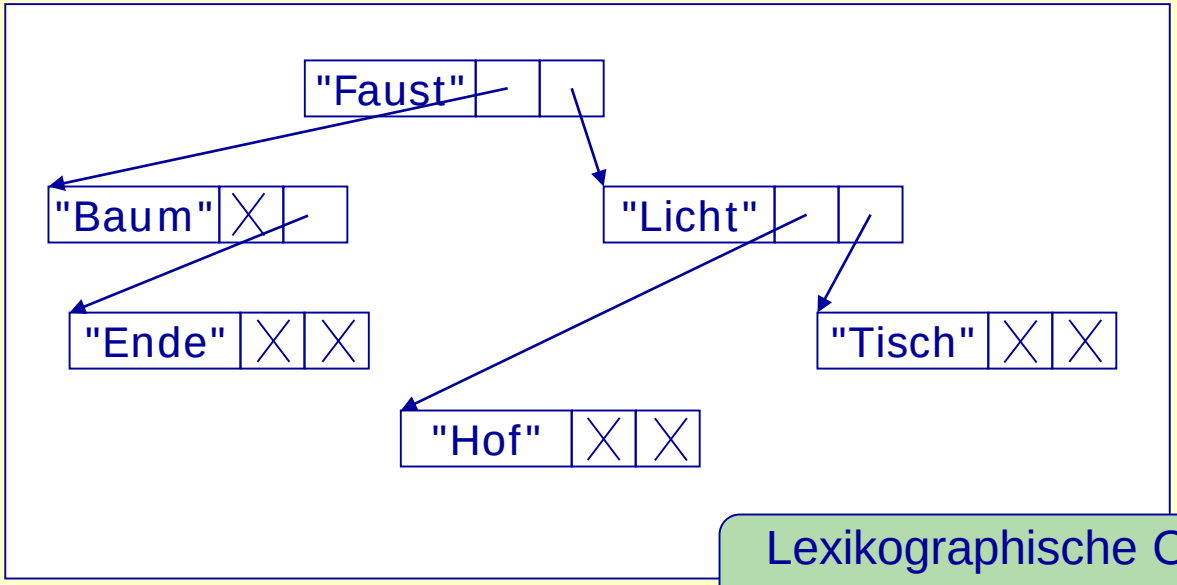
Grundaufgabe in vielen Programmen:

Information abspeichern und wiederfinden

	Unbeschränkte Größe	suchen	einfügen	streichen
Arrays	-	+	-	-
verkettete Listen	+	-	+	+
Bäume*)	+	+	+	+

*) der Preis: je Informationseinheit → 2 Verweise

Sortierte Bäume



Lexikographische Ordnung
(Strings nach Unicode)

Aufsteigend sortierter binärer Baum:

- der Baum ist leer *oder*
- der Baum besteht nur aus dem Wurzelknoten *oder*
- der Inhalt des Wurzelknotens ist größer als alle Knoteninhalte im linken Teilbaum und kleiner als alle Inhalte im rechten Teilbaum

u n d der rechte und linke Teilbaum sind selbst aufsteigend sortiert.

Rekursive Definition

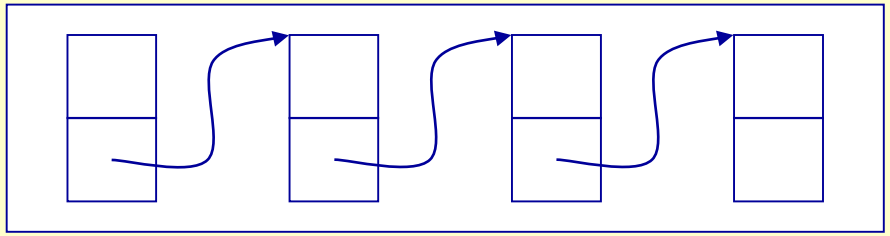
Binäre Bäume:

Operationen

Bäume sind rekursive Datenstrukturen: Verarbeitungsalgorithmen rekursiv

- Iterative Algorithmen i. allg. unnatürlich

- Andere Situation bei Listen:



Auffassen als

- rekursive Struktur:
  (Liste: Zelle mit Inhalt + (Rest-)Liste)
  → rekursiver Algorithmus

- iterative Struktur:
  (Liste: Folge verketteter Zellen)
  → iterativer Algorithmus

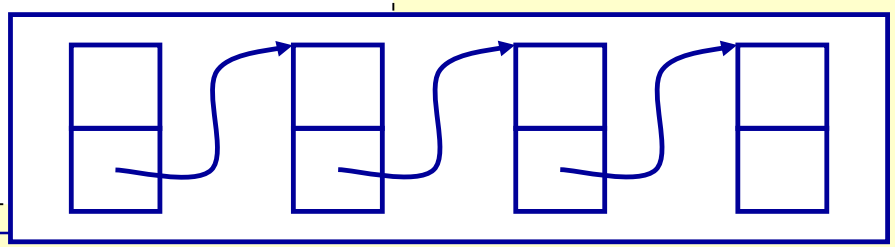
Länge einer Liste: rekursiv - iterativ

```
public int length () {
   if (rest == null)
      return 1;
   else
      return 1 + rest.length();
}
```

```
public int length1 () {
   int lgt = 1;
   IntList actList = this;

   while (actList.rest != null) {
      actList = actList.rest;
      lgt++;
   }
   return lgt;
}
```

Verweis auf Listenelement:
läuft von vorn nach hinten



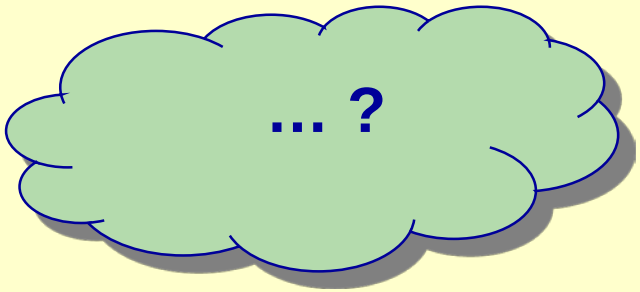
Länge eines Baumes: Knotenanzahl

Rekursive Lösung:

Baum.java

```
public int lengthTree () {
   if (isEmpty())
      return 0;
   else
      return 1
             + links.lengthTree()
             + rechts.lengthTree();
}
```

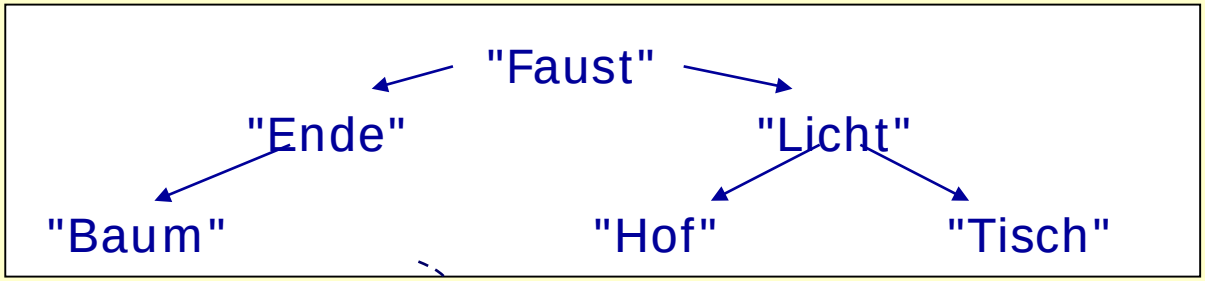
Iterative Lösung:



Sortierte Bäume: neues Element aufnehmen

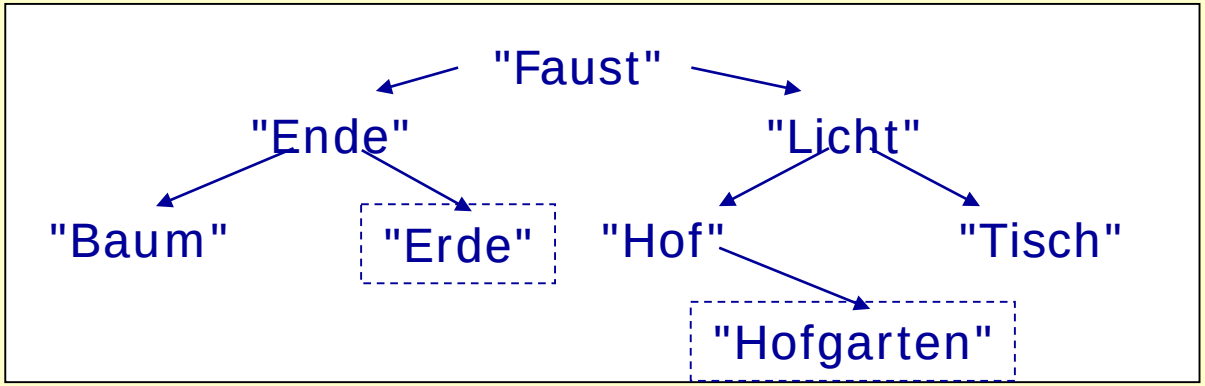
insertSorted: String x Baum → Baum

vorher:



insertSorted("Erde",); insertSorted("Hofgarten",);

nachher:



Prinzip: - neues Element unten
- bestehende Eintragungen unverändert

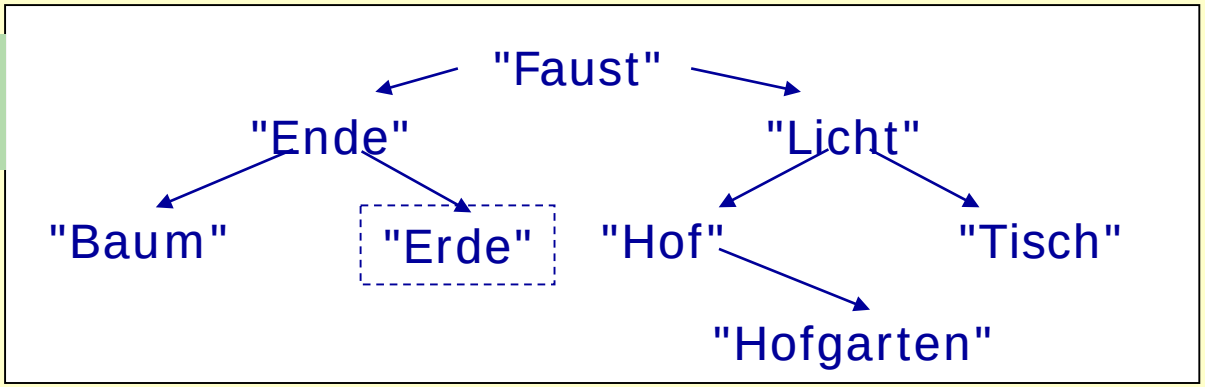
Sortierte Bäume: Element streichen

Fallunterscheidung?

deleteSorted: String x Baum → Baum

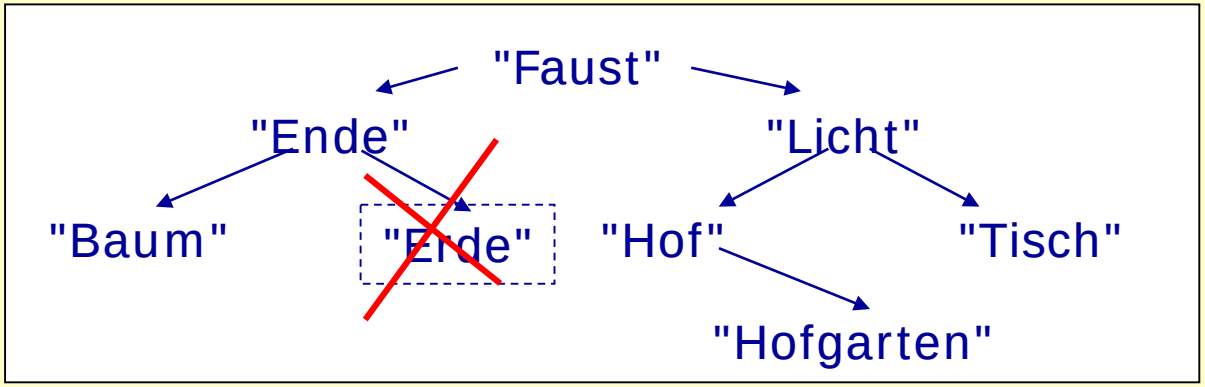
1. Fall: Endknoten

vorher:



delete („Erde“)

nachher:



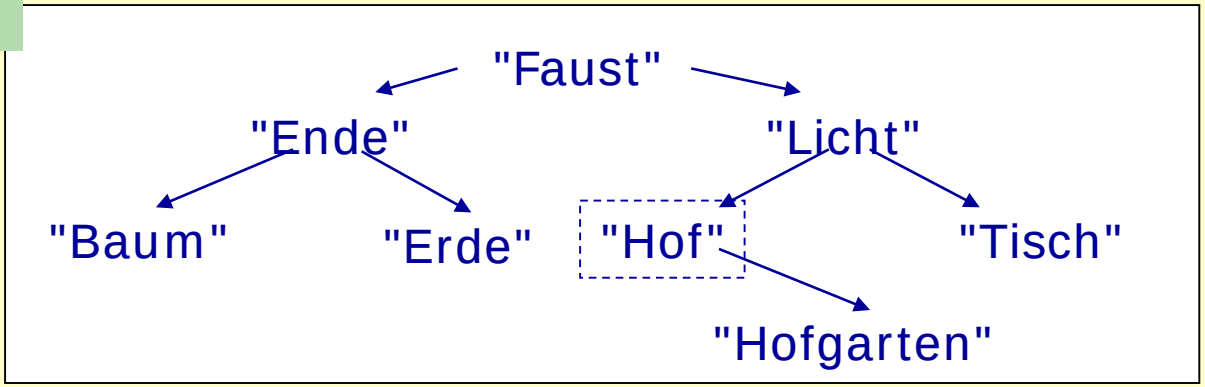
Prinzip: - streichen („Erde“) → leerer Baum

Sortierte Bäume: Element streichen

2. Fall: Nur ein Sohn

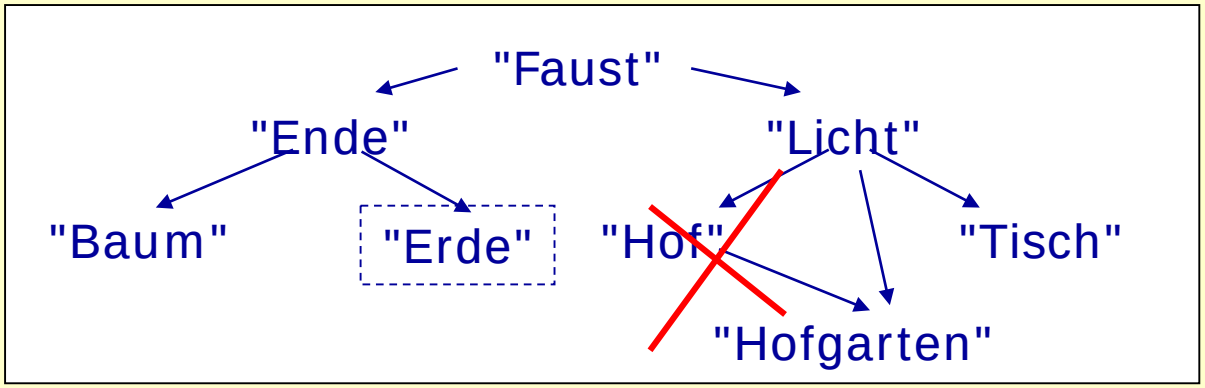
deleteSorted: String x Baum → Baum

vorher:



delete („Hof“)

nachher:



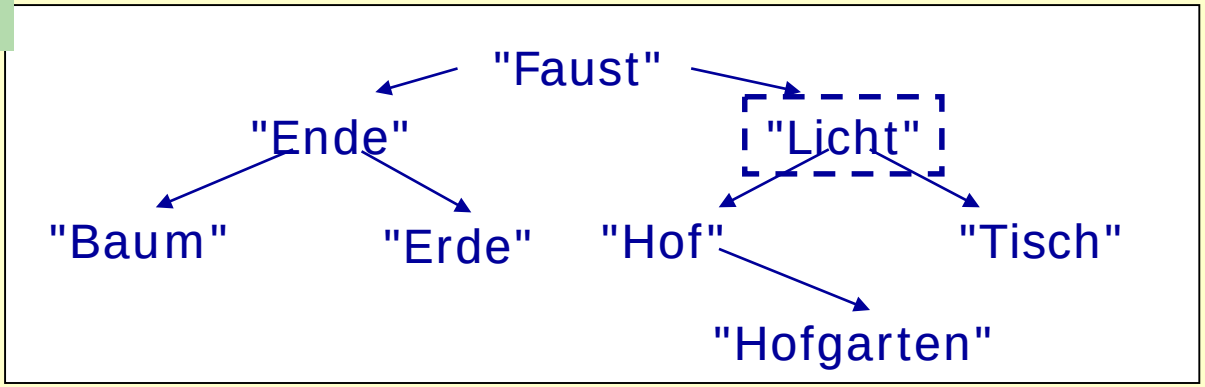
Prinzip: - direkter Verweis vom Vaterknoten auf Sohnkonten

Sortierte Bäume: Element streichen

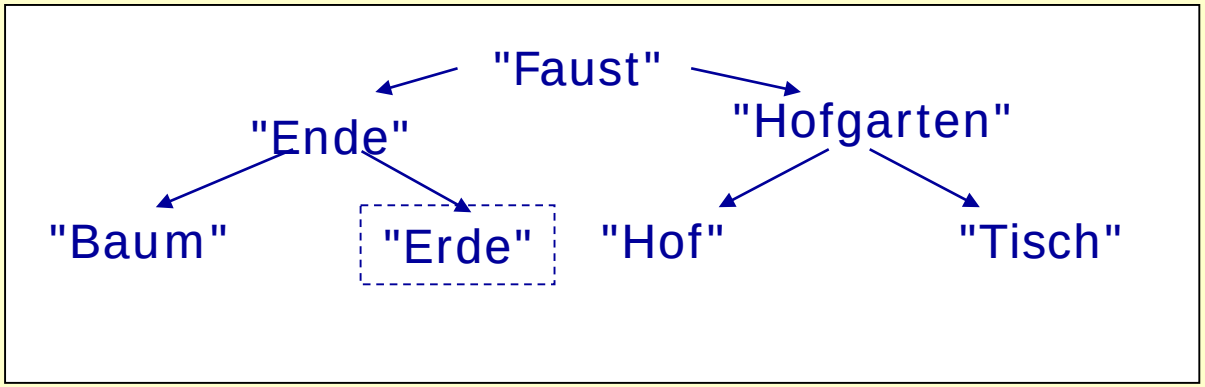
3. Fall: Mittelknoten

deleteSorted: String x Baum → Baum

vorher:



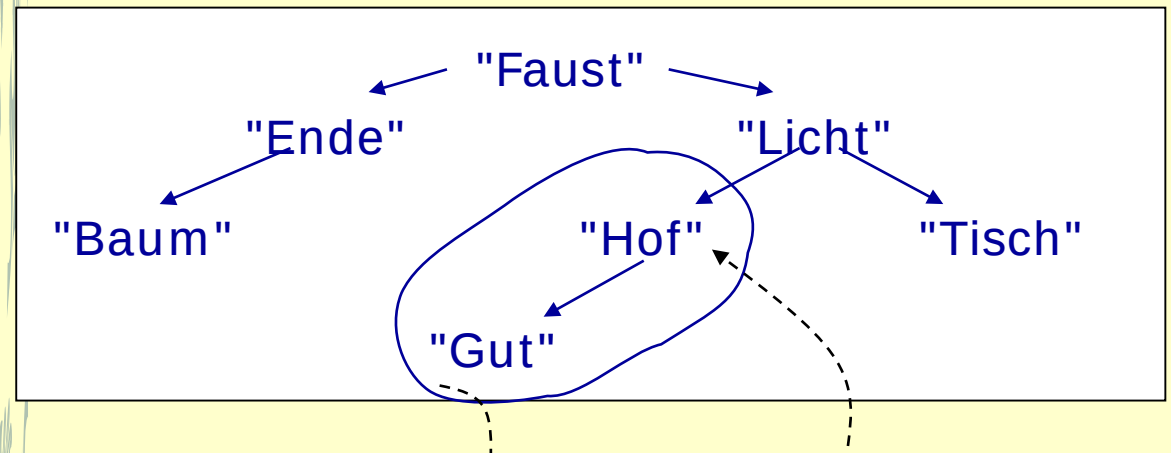
nachher:



Prinzip: - Ersetzen durch größten Knoten des linken Teilbaums
(hat nur einen / keinen Nachfolger, d.h. kann leicht gestrichen werden)
(alternativ: kleinsten Konten des rechten Teilbaums)

Sortierte Bäume: Element suchen

search: Baum x String → Baum



Vergleich s mit Wurzel:
1. gleich s: gefunden
2. s kleiner Wurzel: suche links
3. sonst: suche rechts

search(, "Hof");

→ Knoten (Teilbaum), der „Hof“ als Inhalt des Wurzelknotens hat

search(, "Stuhl"); → leerer Baum

Halbierungsverfahren (binäre Suche)
Komplexität: O(log n)

Binäre Bäume:

Implementation in Java

Bäume in Java: Datendarstellung und Konstruktormethode

```
public class Baum {
   String inhalt;
   Baum links, rechts;

   public Baum () {
     inhalt = null;
     links = null;
     rechts = null;
   }
  ...
```

Baum.java

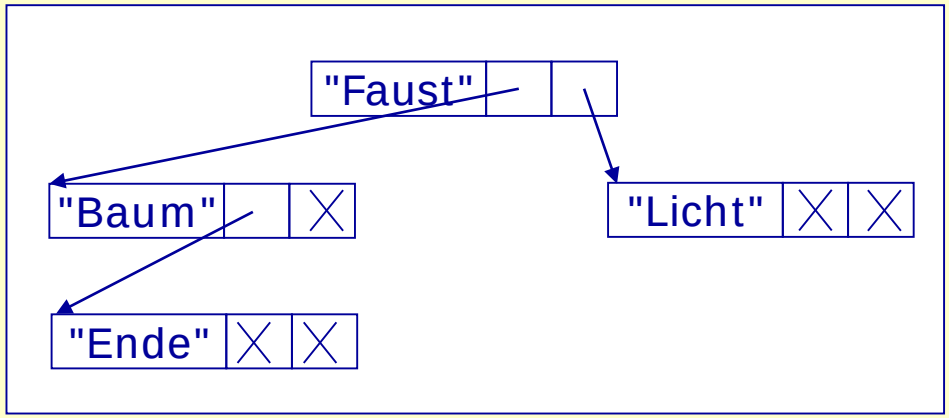
Implementationsvariante:

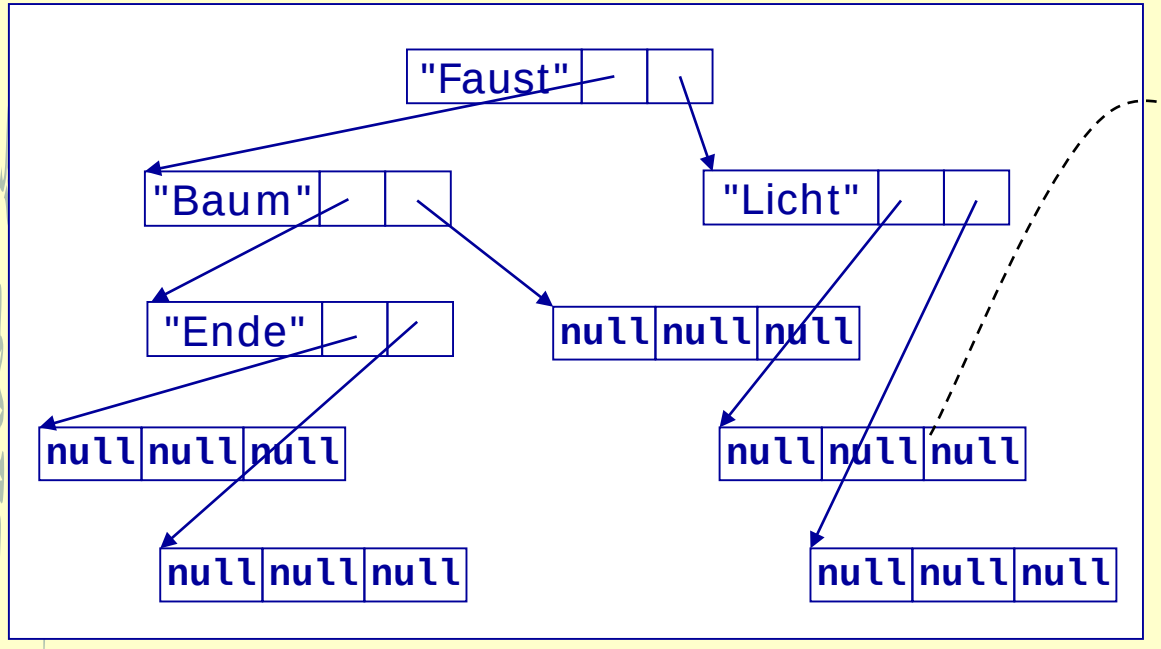
Leerer Baum als Knoten mit inhalt = null

null	null	null

(nicht zwingend
- aber bequem ... für Implementation der Methoden
- und ... Speicherplatz-aufwendig)

Leere Bäume: als null-Objekt (x) oder spezielle Datenzelle





Für jeden leeren Baum (x):

extra Leerknoten-Datenfelder
(im Beispiel: 9 statt 4 Knoten)

Neues Element aufnehmen: Implementation

Baum.java

```
public void insertSorted(String s) {

   if (isEmpty()) {
     inhalt = s
     links = new Baum();
     rechts = new Baum();
   }

   else if (s.compareTo(inhalt) == 0);


   else if (s.compareTo(inhalt) < 0)
      links.insertSorted(s);
   else
      rechts.insertSorted(s);
}
```

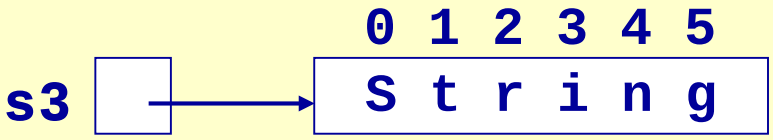
Endknoten

s bereits enthalten

lexikographisch kleiner
→ füge links ein

isEmpty() → | null | null | null | → bequeme Erweiterung des Baumes
→ aber: ein neuer Eintrag erfordert das Anfordern von zwei neuen Datensätzen

Klasse java.lang.*String*



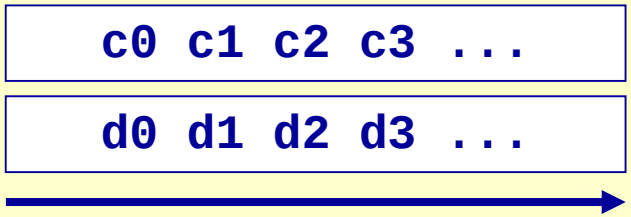
- Vergleich: **lexikographische Ordnung**

```
int compareTo(String s)
```

return-Wert

s3.compareTo("String")	0
s3.compareTo("Ttring")	< 0
s3.compareTo("Aaaaaa")	> 0
s3.compareTo("String1")	< 0
s3.compareTo("S")	> 0

"String" = "String"
"String" < "Ttring"
"String" > "Aaaaaa"
"String" < "String1"
"String" > "S"



Vergleichsrichtung: Unicode-Werte (erster verschiedener Wert)

Element suchen: Implementation

Baum.java

```
public Baum search (String s) {

   if (isEmpty())
      return null;

   else if (s.compareTo(inhalt) == 0)
      return this;


   else if (s.compareTo(inhalt) < 0)
      return links.search(s);
   else
      return rechts.search(s);
}
```

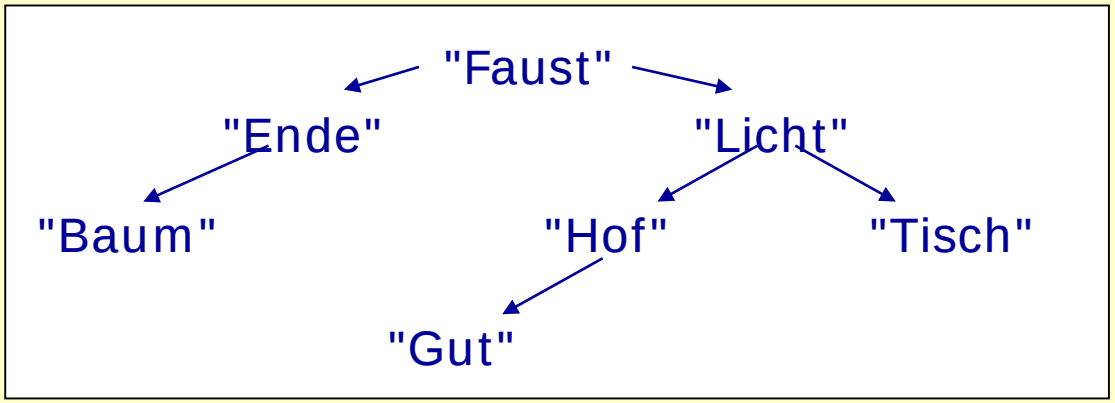
nicht gefunden

gefunden → Resultat: Teilbaum

lexikographisch kleiner → suche links

Bäume: lineare Ausgabestrategien

2-dimensionale Darstellung:



lineare Darstellung (Liste):

Ausgabereihenfolge: Wann die Wurzel?

- **inorder:**
  1. **linker Teilbaum**
  2. **Wurzelelement**
  3. **rechter Teilbaum**
- **preorder:**
  1. **Wurzelelement**
  2. **linker Teilbaum**
  3. **rechter Teilbaum**
- **postorder:**
  1. **linker Teilbaum**
  2. **rechter Teilbaum**
  3. **Wurzelelement**

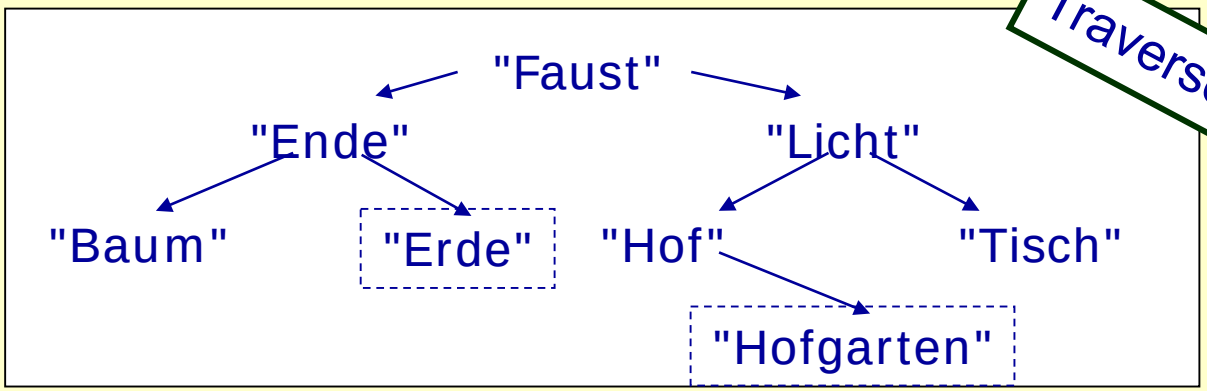
Rekursive Definition

Welche Strategie erhält Sortierung?

Behandlung syntaktischer Strukturen

Lineare Ausgabestrategien: Beispiel

TraverseTest.java



Inorder:

```
Baum Ende Erde Faust Hof Hofgarten Licht Tisch
```

Preorder:

```
Faust Ende Baum Erde Licht Hof Hofgarten Tisch
```

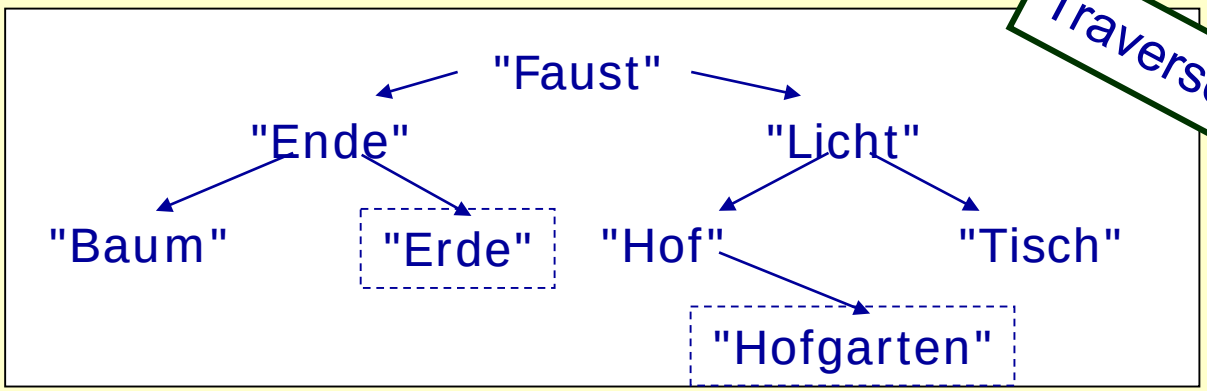
Postorder:

```
Baum Erde Ende Hofgarten Hof Tisch Licht Fa
```

- Inorder: 1. linker Teilbaum
  2. Wurzelelement
  3. rechter Teilbaum
- preorder: 1. Wurzelelement
  2. linker Teilbaum
  3. rechter Teilbaum
- postorder: 1. linker Teilbaum
  2. rechter Teilbaum
  3. Wurzelelement

Lineare Ausgabestrategien: Beispiel

TraverseTest.java



Inorder:

```
Baum Ende Erde Faust Hof Hofgarten Licht Tisch
```

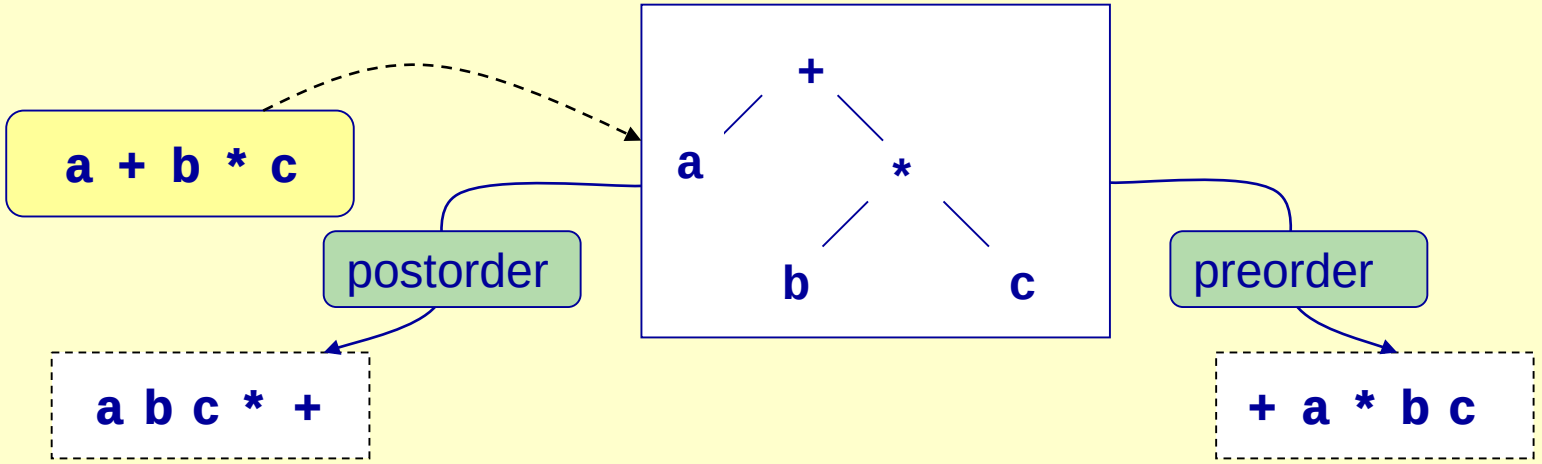
Preorder:

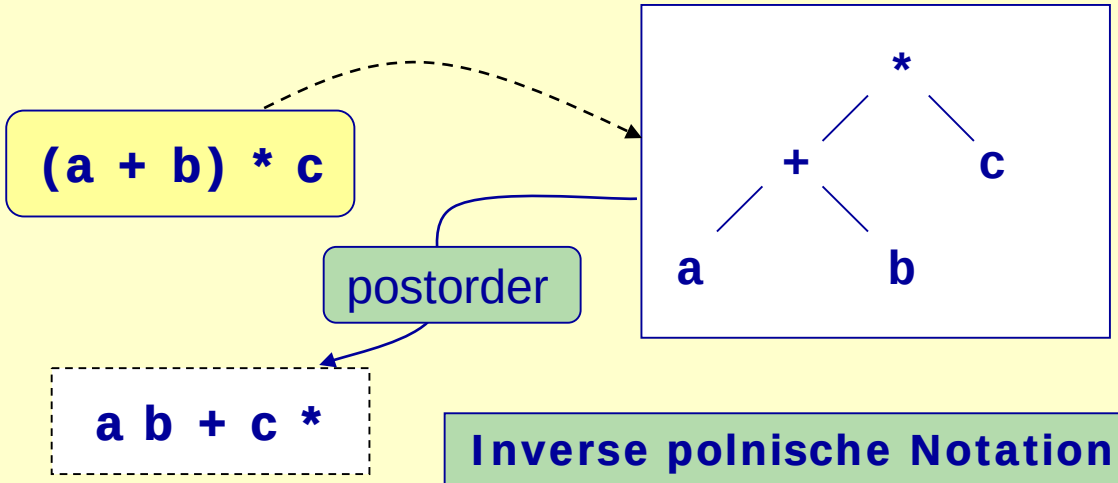
```
Faust Ende Baum Erde Licht Hof Hofgarten Tisch
```

Postorder:

```
Baum Erde Ende Hofgarten Hof Tisch Licht Faust
```

Sinn von 'postorder' und 'preorder'





Polnische Notation: Operanden folgen Operator

Inverse polnische Notation: Operator folgt Operanden

Anwendung:
- **Klammerfreie Ausdrücke**
- **Stack-Computer**

Ausgabestrategien: Implementation

Traverse.java

```
public static void inorder(Baum b) {
   if (!b.isEmpty()) {
      inorder (b.left());
      System.out.print(b.value() + " ");
      inorder (b.right());
   }
}
```

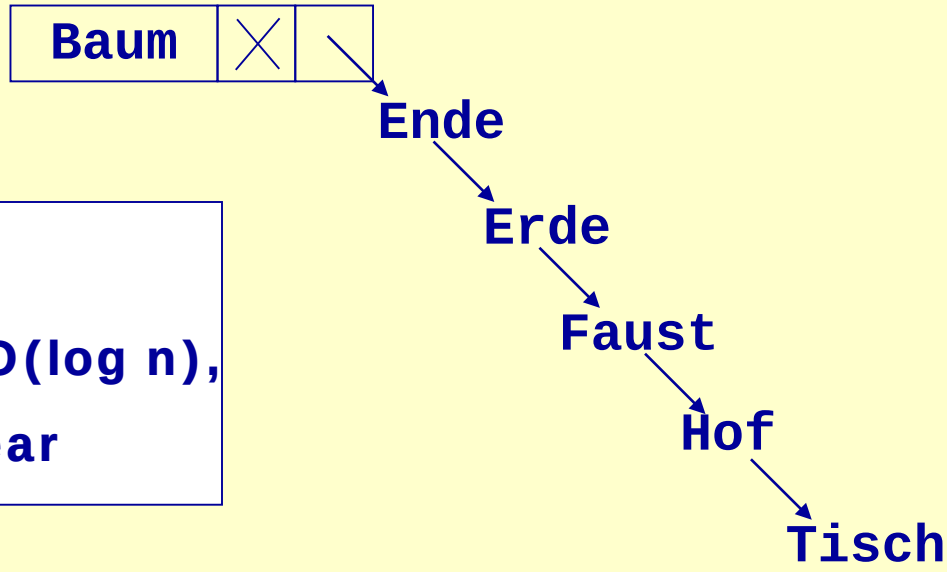
Inorder:

1. linken Teilbaum ausgeben
2. Wurzelelement ausgeben
3. rechten Teilbaum ausgeben

- 'static' günstig?
- Alternative?

Entartete Bäume: schlecht ausbalanciert

```
t.insertSorted("Baum");
t.insertSorted("Ende");
t.insertSorted("Erde");
t.insertSorted("Faust");
t.insertSorted("Hof");
t.insertSorted("Tisch");
```



Suchen:

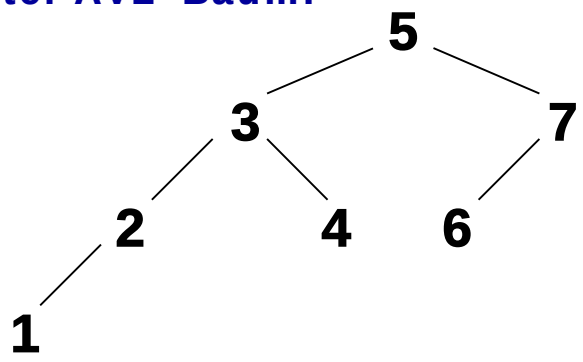
- nicht mehr O(log n),
- sondern linear

AVL–Bäume: ausgeglichene Bäume

AVL-Baum:

Für jeden Knoten unterscheiden sich die Höhen der beiden Teilbäume (rechts und links) um höchstens 1

sortierter AVL–Baum:



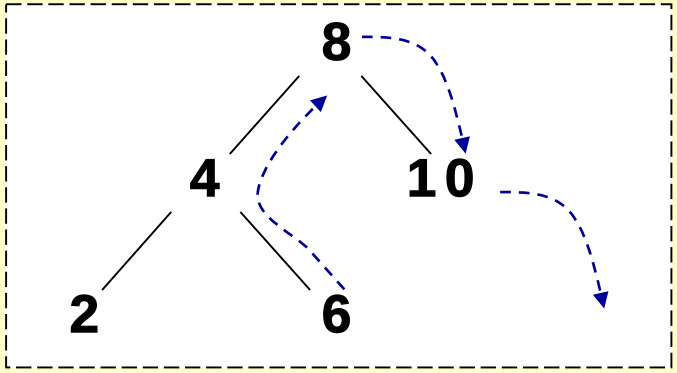
für Knoten 5:

Höhe(Teilbaum(3)) = 3
Höhe(Teilbaum(7)) = 2

für Knoten 7:

...

Einfügen neuer Knoten in AVL–Bäume



Einfügen auf rechter Seite kein Problem : 9 oder 11

Problematisch : 1, 5, 7

Baum muss reorganisiert werden:
hier: größtes Element des linken Teilbaums als neue Wurzel
(neue Wurzel und damit neue linke / rechte Teilbäume)