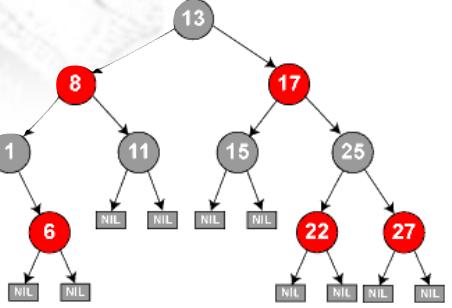
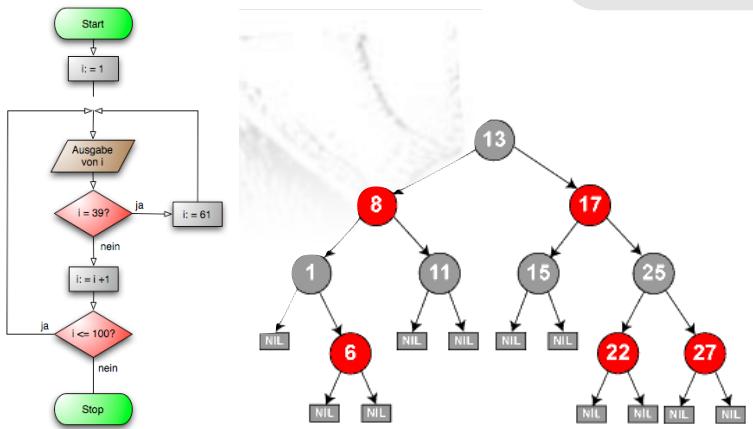




# Algorithmen und Datenstrukturen



Version 1.4 vom 17.April 2025

# **II. Einfache Datentypen und -strukturen**

---

## **2. Einfache Datentypen und -strukturen**

- 2.1. Grundtypen**
- 2.2. Verbund- und Zeigertypen**
- 2.3. Sequenzen**
- 2.4. Stacks und Queues**
- 2.5. Bäume**

# II. Einfache Datentypen und -strukturen

---

## 2. Einfache Datentypen und -strukturen

### 2.1. Grundtypen

- 2.2. Verbund- und Zeigertypen
- 2.3. Sequenzen
- 2.4. Stacks und Queues
- 2.5. Bäume

# Begriffsbestimmungen - Datentyp

**Die Begriffe Datenstruktur und Datentyp wollen wir gegeneinander abgrenzen:**

# Begriffsbestimmungen - Datentyp

**Die Begriffe Datenstruktur und Datentyp wollen wir gegeneinander abgrenzen:**

## D **Datentyp**

Ein Datentyp  $T = \langle V; O \rangle$  besteht aus einer Menge von Werten  $V$ , so wie einer Menge von Operationen  $O$ .

# Begriffsbestimmungen - Datentyp

**Die Begriffe Datenstruktur und Datentyp wollen wir gegeneinander abgrenzen:**

## D Datentyp

Ein Datentyp  $T = \langle V; O \rangle$  besteht aus einer Menge von Werten  $V$ , so wie einer Menge von Operationen  $O$ .

Im Zusammenhang mit Programmiersprachen (Compilerbau) sind Werte und die Operationen meist **syntaktische Objekte**, die interpretiert werden müssen. Darüber hinaus muss festgelegt werden, wie die Werte im Speicher eines Computers dargestellt werden.

Zwei Funktionen sind dann zusätzlich von Bedeutung:

- **Interpretation (Semantik)** - weist jedem Objekt aus  $V$  einen konkreten Wert (z.B. eine ganze Zahl) und jeder Operation aus  $O$  eine konkrete Funktion (z.B. die Addition ganzer Zahlen) zu.
- **Speicherfunktion** - sie bestimmt, wie die Objekte aus  $V$  im Speicher eines Computers abgelegt werden; d.h. sie bildet jedes Objekt aus  $V$  auf eine binäre Zahl mit fester Länge ab.

Umgangssprachlich benutzt man das Wort Datentyp häufig als Synonym für die Wertemenge  $V$ .

# Ganze Zahlen: Der Datentyp Integer

# Ganze Zahlen: Der Datentyp Integer

## B Datentyp: integer

$$\text{integer} = \langle \mathbb{I} ; \{+, -, *, /, \text{mod}\} \rangle$$

Die Menge  $\mathbb{I}$  enthält die syntaktischen Objekte des Datentyps `integer` - also die Konstanten, so wie man sie in ein Computerprogramm "eintippt". Das könnten Ganze Zahlen in gewöhnlicher, hexadezimaler, oktaler und/oder binärer Darstellung sein.

Die **Interpretation**  $\mathcal{I}$  ordnet jeder Konstanten eine Ganze Zahl-, und jedem Operationssymbol eine Operation zu; z.B.  $\mathcal{I}(0x16) = 22$  oder  $\mathcal{I}(+) = + : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ .

Die **Speicherfunktion**  $\mathcal{S}$  weist jeder Konstanten aus  $\mathbb{I}$  eine Ganze Zahl in Binärdarstellung zu - am gebräuchlichsten ist die Darstellung im Zweierkomplement. Die Länge der Binärzahl orientiert sich meist an der Wortgröße des jeweiligen Computers (z.B. 16 Bit) und beschränkt zugleich den Wertebereich der darstellbaren Zahlen (z.B. -32768 - 32767).

# Ganze Zahlen: Der Datentyp Integer

## B Datentyp: integer

$$\text{integer} = \langle \mathbb{I} ; \{+, -, *, /, \text{mod}\} \rangle$$

Die Menge  $\mathbb{I}$  enthält die syntaktischen Objekte des Datentyps integer - also die Konstanten, so wie man sie in ein Computerprogramm "eintippt". Das könnten Ganze Zahlen in gewöhnlicher, hexadezimaler, oktaler und/oder binärer Darstellung sein.

Die **Interpretation**  $\mathcal{I}$  ordnet jeder Konstanten eine Ganze Zahl-, und jedem Operationssymbol eine Operation zu; z.B.  $\mathcal{I}(0x16) = 22$  oder  $\mathcal{I}(+) = + : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ .

Die **Speicherfunktion**  $\mathcal{S}$  weist jeder Konstanten aus  $\mathbb{I}$  eine Ganze Zahl in Binärdarstellung zu - am gebräuchlichsten ist die Darstellung im Zweierkomplement. Die Länge der Binärzahl orientiert sich meist an der Wortgröße des jeweiligen Computers (z.B. 16 Bit) und beschränkt zugleich den Wertebereich der darstellbaren Zahlen (z.B. -32768 - 32767).

**Elementare Datentypen (Grundtypen)  
hängen eng mit der Rechnerarchitektur  
zusammen!**

# Ganze Zahlen: Der Datentyp Integer

## B Datentyp: integer

$$\text{integer} = \langle \mathbb{I} ; \{+, -, *, /, \text{mod}\} \rangle$$

Die Menge  $\mathbb{I}$  enthält die syntaktischen Objekte des Datentyps `integer` - also die Konstanten, so wie man sie in ein Computerprogramm "eintippt". Das könnten Ganze Zahlen in gewöhnlicher, hexadezimaler, oktaler und/oder binärer Darstellung sein.

Die **Interpretation**  $\mathcal{I}$  ordnet jeder Konstanten eine Ganze Zahl-, und jedem Operationssymbol eine Operation zu; z.B.  $\mathcal{I}(0x16) = 22$  oder  $\mathcal{I}(+) = + : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ .

Die **Speicherfunktion**  $\mathcal{S}$  weist jeder Konstanten aus  $\mathbb{I}$  eine Ganze Zahl in Binärdarstellung zu - am gebräuchlichsten ist die Darstellung im Zweierkomplement. Die Länge der Binärzahl orientiert sich meist an der Wortgröße des jeweiligen Computers (z.B. 16 Bit) und beschränkt zugleich den Wertebereich der darstellbaren Zahlen (z.B. -32768 - 32767).

**Elementare Datentypen (Grundtypen)  
hängen eng mit der Rechnerarchitektur  
zusammen!**

**Anmerkung:** Manchmal will man nicht zwischen syntaktischen Objekten und ihrer Interpretation unterscheiden, dann kann man den Datentyp `integer` für ein 16-Bit System z.B. auch so beschreiben:

$$\text{integer} = \langle \mathbb{Z}_{2^{16}} ; \{+, -, *, /, \text{mod}\} \rangle$$

# Darstellung von Datentypen I: Integer

# Darstellung von Datentypen I: Integer

## B 3-Bit Integer

Das höchstwertige Bit entspricht dem Vorzeichen. Ist es gesetzt, handelt es sich um eine negative Zahl.

Negative Zahlen entstehen durch Bildung des Komplements und anschließende Addition von 1; z.B. für -3:

Binärdarstellung für 3	011b
Komplement	100b
+1	101b

**Vorteil:** Addition analog zu natürlichen Zahlen!

# Darstellung von Datentypen I: Integer

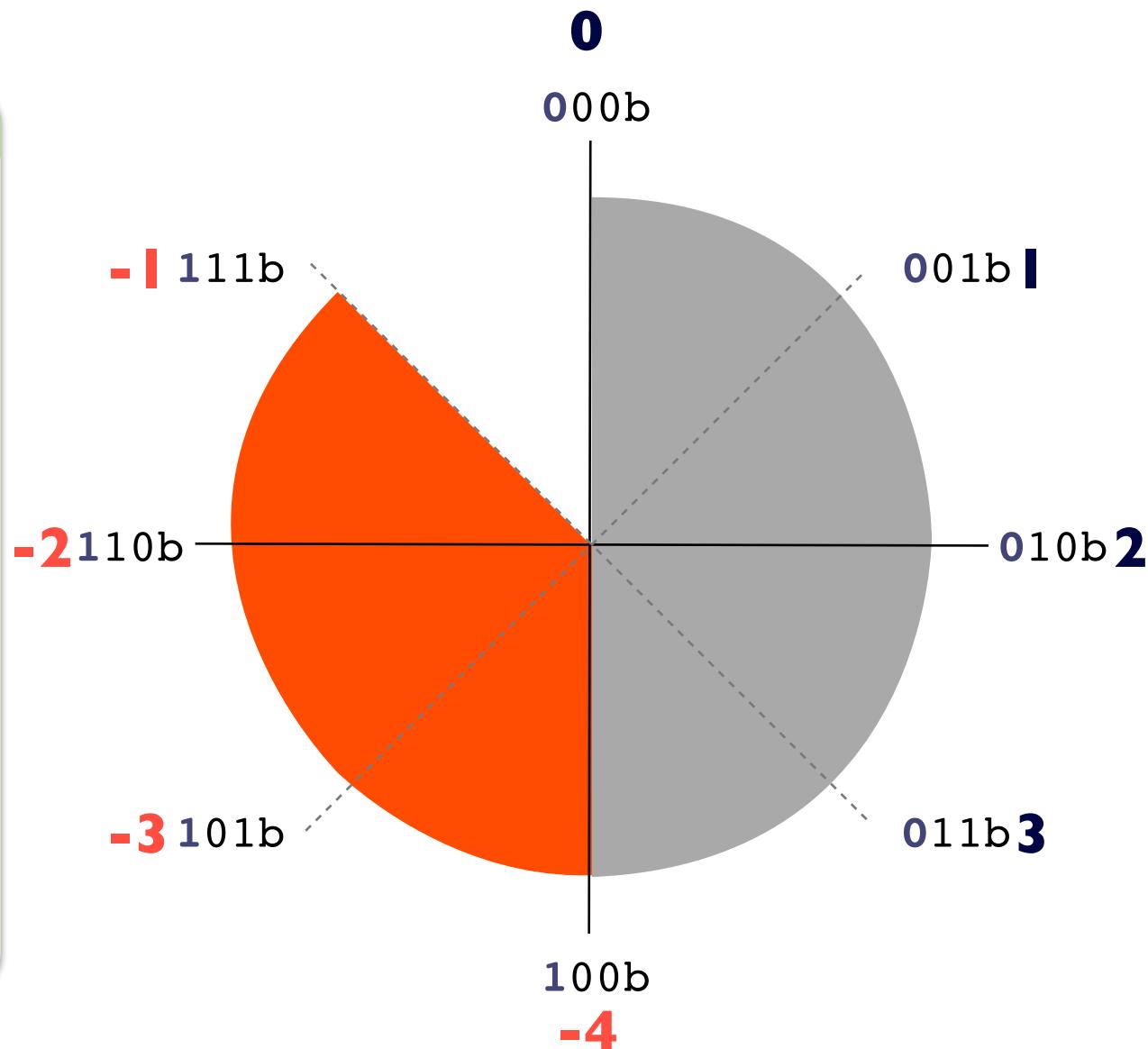
## B 3-Bit Integer

Das höchstwertige Bit entspricht dem Vorzeichen. Ist es gesetzt, handelt es sich um eine negative Zahl.

Negative Zahlen entstehen durch Bildung des Komplements und anschließende Addition von 1; z.B. für -3:

Binärdarstellung für 3	011b
Komplement	100b
+1	101b

**Vorteil:** Addition analog zu natürlichen Zahlen!



# Reelle Zahlen: Fließkommazahlen

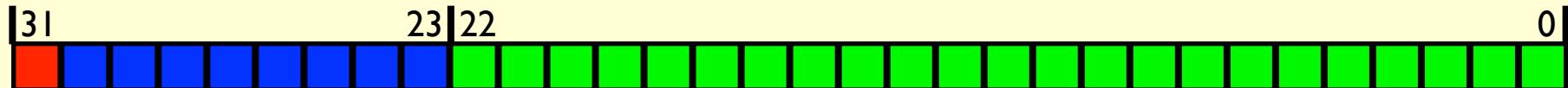
Die Reellen Zahlen  $\mathbb{R}$  sind nicht aufzählbar und können daher nicht unmittelbar auf Maschinenwerte abgebildet werden. Als Kompromiss greift man auf eine Fließkommadarstellung zurück.

## D Fließkommazahlen

Eine Fließkommazahl besteht aus drei Komponenten: dem **Vorzeichen**, der **Mantisse** und dem **Exponent**. Der Wert einer Fließkommazahl lässt sich wie folgt aus diesen Komponenten rekonstruieren:

$$x = (-1)^s * m * 2^e$$

Für Exponent und Mantisse steht nur eine begrenzte Zahl an Bits zur Verfügung; z.B. 8 Bits für den Exponenten und 23 Bits für die Mantisse (IEEE 754-single precision):



Die Menge aller Fließkommazahlen, die man mit  $m$ -Mantissen- und  $e$ -Exponentbits darstellen kann beschreiben wir durch  $\mathbb{F}_{(m,e)}$ .

Konsequenzen der Fließkommadarstellung:

- Reelle Zahlen können maximal auf  $\lceil \log 2^m \rceil$ -Stellen genau gespeichert werden, wobei  $m$  die Zahl der Bits für die Mantisse ist.
- Zahlen aus dem Dezimalsystem können nicht immer exakt im Dualsystem dargestellt werden. So ist 0.2 im Dualsystem periodisch: 0.0011.

# Weitere Datentypen am Beispiel JAVA

Typname	Wertebereich	Typische Operationen
<b>Ganzzahlige Typen</b>		
byte	$\mathbb{Z}_{2^8}$	+ , - , * , / , << , >> , & ,   , % , ...
short	$\mathbb{Z}_{2^{16}}$	+ , - , * , / , << , >> , & ,   , % , ...
int	$\mathbb{Z}_{2^{32}}$	+ , - , * , / , << , >> , & ,   , % , ...
long	$\mathbb{Z}_{2^{64}}$	+ , - , * , / , << , >> , & ,   , % , ...
<b>Fließkommatypen</b>		
float	$\mathbb{F}_{(23,8)}$	+ , - , * , / , % , ...
double	$\mathbb{F}_{(52,11)}$	+ , - , * , / , % , ...
<b>Sonstige Typen</b>		
char	{ „A“ , ... , „Z“ , ... }	!=, ==
bool	{ true , false }	&& ,    , ! , == , !=

# Weitere Datentypen am Beispiel JAVA

Typname	Wertebereich	Typische Operationen
<b>Ganzzahlige Typen</b>		
byte	$\mathbb{Z}_{2^8}$	+ , - , * , / , << , >> , & ,   , % , ...
short	$\mathbb{Z}_{2^{16}}$	+ , - , * , / , << , >> , & ,   , % , ...
int	$\mathbb{Z}_{2^{32}}$	+ , - , * , / , << , >> , & ,   , % , ...
long	$\mathbb{Z}_{2^{64}}$	+ , - , * , / , << , >> , & ,   , % , ...
<b>Fließkommatypen</b>		
float	$\mathbb{F}_{(23,8)}$	+ , - , * , / , % , ...
double	$\mathbb{F}_{(52,11)}$	+ , - , * , / , % , ...
<b>Sonstige Typen</b>		
char	{ „A“ , ... , „Z“ , ... }	!=, ==
bool	{ true , false }	&& ,    , ! , == , !=

- Die Operationen auf diesen **Grundtypen** sind meist durch elementare Hardware-Operationen realisiert (hier: JVM-Operationen)

# Weitere Datentypen am Beispiel JAVA

Typname	Wertebereich	Typische Operationen
<b>Ganzzahlige Typen</b>		
byte	$\mathbb{Z}_{2^8}$	+ , - , * , / , << , >> , & ,   , % , ...
short	$\mathbb{Z}_{2^{16}}$	+ , - , * , / , << , >> , & ,   , % , ...
int	$\mathbb{Z}_{2^{32}}$	+ , - , * , / , << , >> , & ,   , % , ...
long	$\mathbb{Z}_{2^{64}}$	+ , - , * , / , << , >> , & ,   , % , ...
<b>Fließkommatypen</b>		
float	$\mathbb{F}_{(23,8)}$	+ , - , * , / , % , ...
double	$\mathbb{F}_{(52,11)}$	+ , - , * , / , % , ...
<b>Sonstige Typen</b>		
char	{ „A“ , ... , „Z“ , ... }	!=, ==
bool	{ true , false }	&& ,    , ! , == , !=

- Die Operationen auf diesen **Grundtypen** sind meist durch elementare **Hardware-Operationen** realisiert (hier: **JVM-Operationen**)
- In Sprachen wie C++ gehören **Zeiger-** und **Referenztypen** ebenfalls zu den elementaren **Datentypen**

# Die $w$ -Word-RAM

- Modell auf dem wir (gedanklich) unsere Algorithmen und Datenstrukturen implementieren.
- Praxisnahe Abwandlung / Erweiterung der **RAM** aus dem Theorie-Teil:
  - Speicherzellen nehmen Worte auf
  - für ein Wort stehen  $w$  Bit zur Verfügung  
Speicherzellen sind exakt  $w$ -Bit breit
  - $w$ -Word-RAM arbeitet folglich nicht auf ganz  $\mathbb{Z}$  sondern auf  $\mathbb{Z}_{2^w}$  mit

$$\mathbb{Z}_{2^w} = \{ z \in \mathbb{Z} \mid -2^{w-1} \leq z \leq 2^{w-1} - 1 \}$$

Folge: Arbeit mit größeren Zahlen muss "simuliert" werden

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

## D Datenstruktur

Eine Datenstruktur ist ein Datentyp. Sie baut auf einem oder mehreren Datentypen auf und zeichnet sich dadurch aus, dass sie die Werte der ausgehenden Datentypen durch eine Menge von Regeln in Beziehung setzt.

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

## D Datenstruktur

Eine Datenstruktur ist ein Datentyp. Sie baut auf einem oder mehreren Datentypen auf und zeichnet sich dadurch aus, dass sie die Werte der ausgehenden Datentypen durch eine Menge von Regeln in Beziehung setzt.

- **Eine Datenstruktur ist ein zusammengesetzter Datentyp**

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

## D Datenstruktur

Eine Datenstruktur ist ein Datentyp. Sie baut auf einem oder mehreren Datentypen auf und zeichnet sich dadurch aus, dass sie die Werte der ausgehenden Datentypen durch eine Menge von Regeln in Beziehung setzt.

- **Eine Datenstruktur ist ein zusammengesetzter Datentyp**
- **Einfache Beispiele sind:**

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

## D Datenstruktur

Eine Datenstruktur ist ein Datentyp. Sie baut auf einem oder mehreren Datentypen auf und zeichnet sich dadurch aus, dass sie die Werte der ausgehenden Datentypen durch eine Menge von Regeln in Beziehung setzt.

- **Eine Datenstruktur ist ein zusammengesetzter Datentyp**
- **Einfache Beispiele sind:**
  - Felder (Arrays), Verbundtypen (Records), Klassen und Mengen (Sets)

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

## D Datenstruktur

Eine Datenstruktur ist ein Datentyp. Sie baut auf einem oder mehreren Datentypen auf und zeichnet sich dadurch aus, dass sie die Werte der ausgehenden Datentypen durch eine Menge von Regeln in Beziehung setzt.

- **Eine Datenstruktur ist ein zusammengesetzter Datentyp**
- **Einfache Beispiele sind:**
  - Felder (Arrays), Verbundtypen (Records), Klassen und Mengen (Sets)
- **Es besteht ein enger Zusammenhang zwischen einer Datenstruktur und den Algorithmen, die auf ihr operieren**

# Begriffsbestimmungen - Datenstruktur

**Den Begriff Datenstruktur wollen wir nur informell definieren:**

## D Datenstruktur

Eine Datenstruktur ist ein Datentyp. Sie baut auf einem oder mehreren Datentypen auf und zeichnet sich dadurch aus, dass sie die Werte der ausgehenden Datentypen durch eine Menge von Regeln in Beziehung setzt.

- **Eine Datenstruktur ist ein zusammengesetzter Datentyp**
- **Einfache Beispiele sind:**
  - Felder (Arrays), Verbundtypen (Records), Klassen und Mengen (Sets)
- **Es besteht ein enger Zusammenhang zwischen einer Datenstruktur und den Algorithmen, die auf ihr operieren**
  - Konsequenz: Sie können nicht getrennt voneinander betrachtet werden

# Abstrakte Datentypen (ADT)

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**
  - Menge von Objekten (Elemente / Typen)

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**
  - Menge von Objekten (Elemente / Typen)
  - Menge von Operationen auf diesen Objekten (Syntax und Typ -> Signatur)

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**
  - Menge von Objekten (Elemente / Typen)
  - Menge von Operationen auf diesen Objekten (Syntax und Typ -> Signatur)
  - Satz von Axiomen (Semantik)

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**
  - Menge von Objekten (Elemente / Typen)
  - Menge von Operationen auf diesen Objekten (Syntax und Typ -> Signatur)
  - Satz von Axiomen (Semantik)
- **Top-Down Software-Entwurf**
  - Erst festlegen „Was“ gebraucht wird

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**
  - Menge von Objekten (Elemente / Typen)
  - Menge von Operationen auf diesen Objekten (Syntax und Typ -> Signatur)
  - Satz von Axiomen (Semantik)
- **Top-Down Software-Entwurf**
  - Erst festlegen „Was“ gebraucht wird
- **Abstraktion erleichtert die Analyse von Algorithmen**

# Abstrakte Datentypen (ADT)

- **Spezielle Implementierung wird nicht betrachtet!**
- **Ein abstrakter Datentyp (ADT) wird definiert durch**
  - Menge von Objekten (Elemente / Typen)
  - Menge von Operationen auf diesen Objekten (Syntax und Typ -> Signatur)
  - Satz von Axiomen (Semantik)
- **Top-Down Software-Entwurf**
  - Erst festlegen „Was“ gebraucht wird
- **Abstraktion erleichtert die Analyse von Algorithmen**
- **Trennung von Interface (Signatur) und Implementierung (Semantik) in konkreten Programmiersprachen**
  - in JAVA z.B. durch interface-Deklarationen
  - in C++ durch abstrakte Basisklasse

# ADT: Bool

## B Abstrakter Datentyp: Bool

## B Abstrakter Datentyp: Bool

### Elemente

{true, false}

## B Abstrakter Datentyp: Bool

### Elemente

{true, false}

### Operationen

not : Bool → Bool  
and : Bool × Bool → Bool  
or : Bool × Bool → Bool

## B Abstrakter Datentyp: Bool

### Elemente

{true, false}

### Operationen

not : Bool → Bool  
and : Bool × Bool → Bool  
or : Bool × Bool → Bool

### Axiome

$x : \text{Bool}$

not true = false  
not false = true  
 $x \text{ and true} = x$   
 $x \text{ and false} = \text{false}$   
 $x \text{ or true} = \text{true}$   
 $x \text{ or false} = x$

## B Abstrakter Datentyp: Bool

### Elemente

{true, false}

### Operationen

not : Bool → Bool  
and : Bool × Bool → Bool  
or : Bool × Bool → Bool

### Axiome

$x : \text{Bool}$

not true	=	false
not false	=	true
$x$ and true	=	$x$
$x$ and false	=	false
$x$ or true	=	true
$x$ or false	=	$x$

- Einfaches Beweisen von Aussagen durch Induktion über Termaufbau

# ADT: Stack

## B Abstrakter Datentyp: Stack

## B Abstrakter Datentyp: Stack

### Typen

Ein beliebiger Elementtyp T

## B Abstrakter Datentyp: Stack

### Typen

Ein beliebiger Elementtyp T

### Operationen

StackInit : Stack (Konstante)

EmptyStack : Stack → Bool

push : T × Stack → Stack

pop : Stack → T × Stack

## B Abstrakter Datentyp: Stack

### Typen

Ein beliebiger Elementtyp T

### Operationen

StackInit : Stack (Konstante)  
EmptyStack : Stack → Bool  
push : T × Stack → Stack  
pop : Stack → T × Stack

### Axiome

$x : T$   
 $s : \text{Stack}$

$\text{pop}(\text{push}(x, s)) = (x, s)$   
 $\text{push}(\text{pop}(s)) = s$ , falls  $\text{EmptyStack}(s) = \text{false}$   
 $\text{EmptyStack}(\text{StackInit}) = \text{true}$   
 $\text{EmptyStack}(\text{push}(x, s)) = \text{false}$

# ADT: Stack

## B Abstrakter Datentyp: Stack

### Typen

Ein beliebiger Elementtyp T

### Operationen

StackInit : Stack (Konstante)

EmptyStack : Stack → Bool

push : T × Stack → Stack

pop : Stack → T × Stack

### Axiome

$x : T$

$s : \text{Stack}$

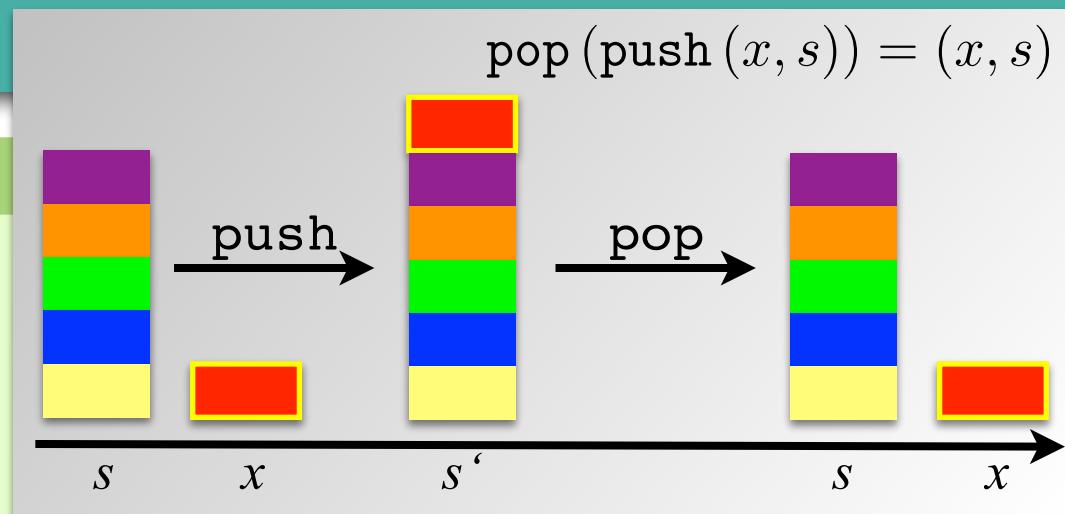
$$\text{pop}(\text{push}(x, s)) = (x, s)$$

$$\text{push}(\text{pop}(s)) = s \quad \text{falls } \text{EmptyStack}(s) = \text{false}$$

$$\text{EmptyStack}(\text{StackInit}) = \text{true}$$

$$\text{EmptyStack}(\text{push}(x, s)) = \text{false}$$

$$\text{pop}(\text{push}(x, s)) = (x, s)$$



# ADT: Stack

## B Abstrakter Datentyp: Stack

### Typen

Ein beliebiger Elementtyp T

### Operationen

StackInit : Stack (Konstante)

EmptyStack : Stack → Bool

push : T × Stack → Stack

pop : Stack → T × Stack

### Axiome

$x : T$

$s : \text{Stack}$

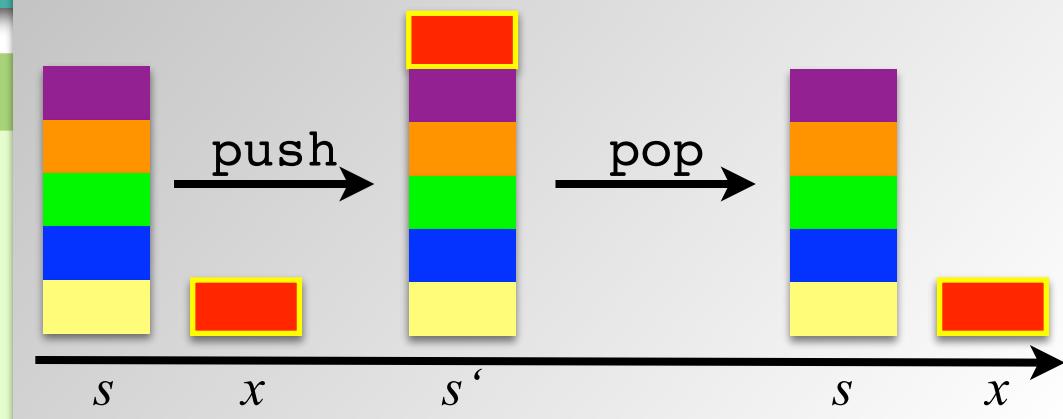
$$\text{pop}(\text{push}(x, s)) = (x, s)$$

$$\text{push}(\text{pop}(s)) = s \quad \text{falls } \text{EmptyStack}(s) = \text{false}$$

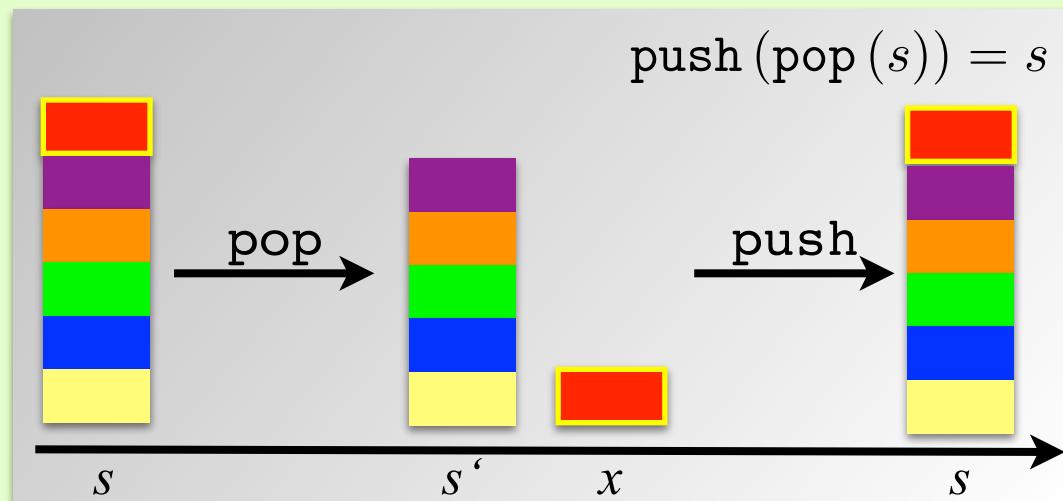
$$\text{EmptyStack}(\text{StackInit}) = \text{true}$$

$$\text{EmptyStack}(\text{push}(x, s)) = \text{false}$$

$$\text{pop}(\text{push}(x, s)) = (x, s)$$



$$\text{push}(\text{pop}(s)) = s$$



# ADT: Stack

## B Abstrakter Datentyp: Stack

### Typen

Ein beliebiger Elementtyp T

### Operationen

StackInit : Stack (Konstante)

EmptyStack : Stack → Bool

push : T × Stack → Stack

pop : Stack → T × Stack

### Axiome

$x : T$

$s : \text{Stack}$

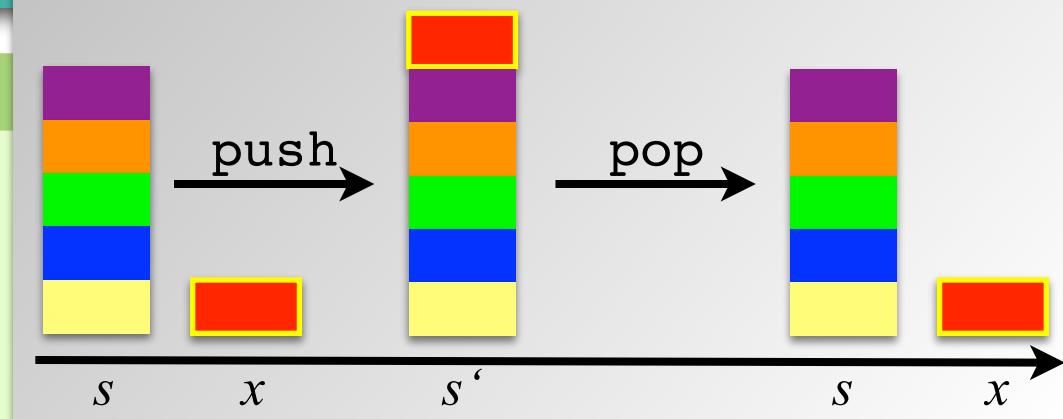
$$\text{pop}(\text{push}(x, s)) = (x, s)$$

$$\text{push}(\text{pop}(s)) = s \quad \text{falls } \text{EmptyStack}(s) = \text{false}$$

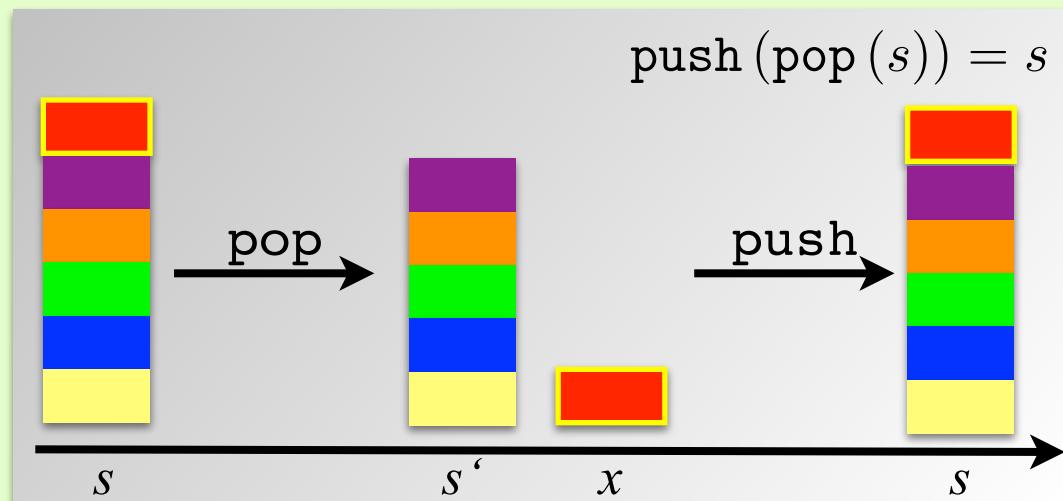
$$\text{EmptyStack}(\text{StackInit}) = \text{true}$$

$$\text{EmptyStack}(\text{push}(x, s)) = \text{false}$$

$$\text{pop}(\text{push}(x, s)) = (x, s)$$



$$\text{push}(\text{pop}(s)) = s$$



- Undefinierte Operationen müssen gesondert behandelt werden

# ADTs: Summary

## ● **Potentielle Probleme mit ADT**

- Anzahl der Axiome wird bei komplexen Fällen sehr groß
- Spezifikation nicht immer leicht zu verstehen
- Vollständigkeit und Widerspruchsfreiheit sind schwer zu prüfen
- Wünschenswert, aber nicht immer trivial: Axiome können in terminierendes und konfluentes Termersetzungssystem überführt werden

# Schnittstelle vs. Datenstruktur

Schnittstelle (API, ADT)	Datenstruktur
<b>Spezifikation:</b> <ul style="list-style-type: none"><li>• Was wird gespeichert?</li><li>• Welche Operationen werden unterstützt?</li></ul>	<b>Repräsentation:</b> <ul style="list-style-type: none"><li>• Wie werden Daten gespeichert?</li><li>• Algorithmen zur Umsetzung der Operationen</li></ul>

# Schnittstelle vs. Datenstruktur

Schnittstelle (API, ADT)	Datenstruktur
<b>Spezifikation:</b> <ul style="list-style-type: none"><li>Was wird gespeichert?</li><li>Welche Operationen werden unterstützt?</li></ul>	<b>Repräsentation:</b> <ul style="list-style-type: none"><li>Wie werden Daten gespeichert?</li><li>Algorithmen zur Umsetzung der Operationen</li></ul>

## ● **ADTs werden wir nicht weiter vertiefen**

- Wir werden nur kleine und kompakte Programme betrachten
- Uns interessieren verschiedene **Implementierungstechniken**, und
- wir wollen **konkrete Implementierungen untersuchen und vergleichen**.

# Schnittstelle vs. Datenstruktur

Schnittstelle (API, ADT)	Datenstruktur
<b>Spezifikation:</b> <ul style="list-style-type: none"><li>Was wird gespeichert?</li><li>Welche Operationen werden unterstützt?</li></ul>	<b>Repräsentation:</b> <ul style="list-style-type: none"><li>Wie werden Daten gespeichert?</li><li>Algorithmen zur Umsetzung der Operationen</li></ul>

- **ADTs werden wir nicht weiter vertiefen**
  - Wir werden nur kleine und kompakte Programme betrachten
  - Uns interessieren verschiedene **Implementierungstechniken**, und
  - wir wollen **konkrete Implementierungen untersuchen und vergleichen**.

Ein ADT abstrahiert von einer konkreten Implementierung!

# II. Einfache Datentypen und -strukturen

---

## 2. Einfache Datentypen und -strukturen

2.1. Grundtypen

**2.2. Verbund- und Zeigertypen**

2.3. Sequenzen

2.4. Stacks und Queues

2.5. Bäume

# Zusammengesetzte Typen

# Zusammengesetzte Typen

- **Wir hatten Datenstrukturen als zusammengesetzte Datentypen definiert**

# Zusammengesetzte Typen

- **Wir hatten Datenstrukturen als zusammengesetzte Datentypen definiert**
  - **In C++ lassen sich die folgenden zusammengesetzten Typen unterscheiden**
    - **Statische Felder / Arrays** z.B. `char Feld[10]`
    - **Strukturen** z.B. `struct Point { int x; int y; };`
    - **Varianten** z.B. `union value { char *s; int I; };`
    - **Klassen** z.B. `class Point { public: int x; int y; };`
- Anmerkung:** In C++ sind Klassen und Strukturen nahezu synonym. In einer Struktur sind alle Mitglieder per default „`public`“, in einer Klasse dagegen „`private`“.

# Zusammengesetzte Typen

- **Wir hatten Datenstrukturen als zusammengesetzte Datentypen definiert**
- **In C++ lassen sich die folgenden zusammengesetzten Typen unterscheiden**
  - **Statische Felder / Arrays** z.B. `char Feld[10]`
  - **Strukturen** z.B. `struct Point { int x; int y; };`
  - **Varianten** z.B. `union value { char *s; int I; };`
  - **Klassen** z.B. `class Point { public: int x; int y; };`
- **Anmerkung:** In C++ sind Klassen und Strukturen nahezu synonym. In einer Struktur sind alle Mitglieder per default „`public`“, in einer Klasse dagegen „`private`“.
- **Alle Typen erlauben direkten Zugriff auf ihre Komponenten - Aufwand des Zugriffs ist konstant**  
... sofern man die Wort-Größe der  $w$ -Word-RAM außer Acht lässt (warum?)

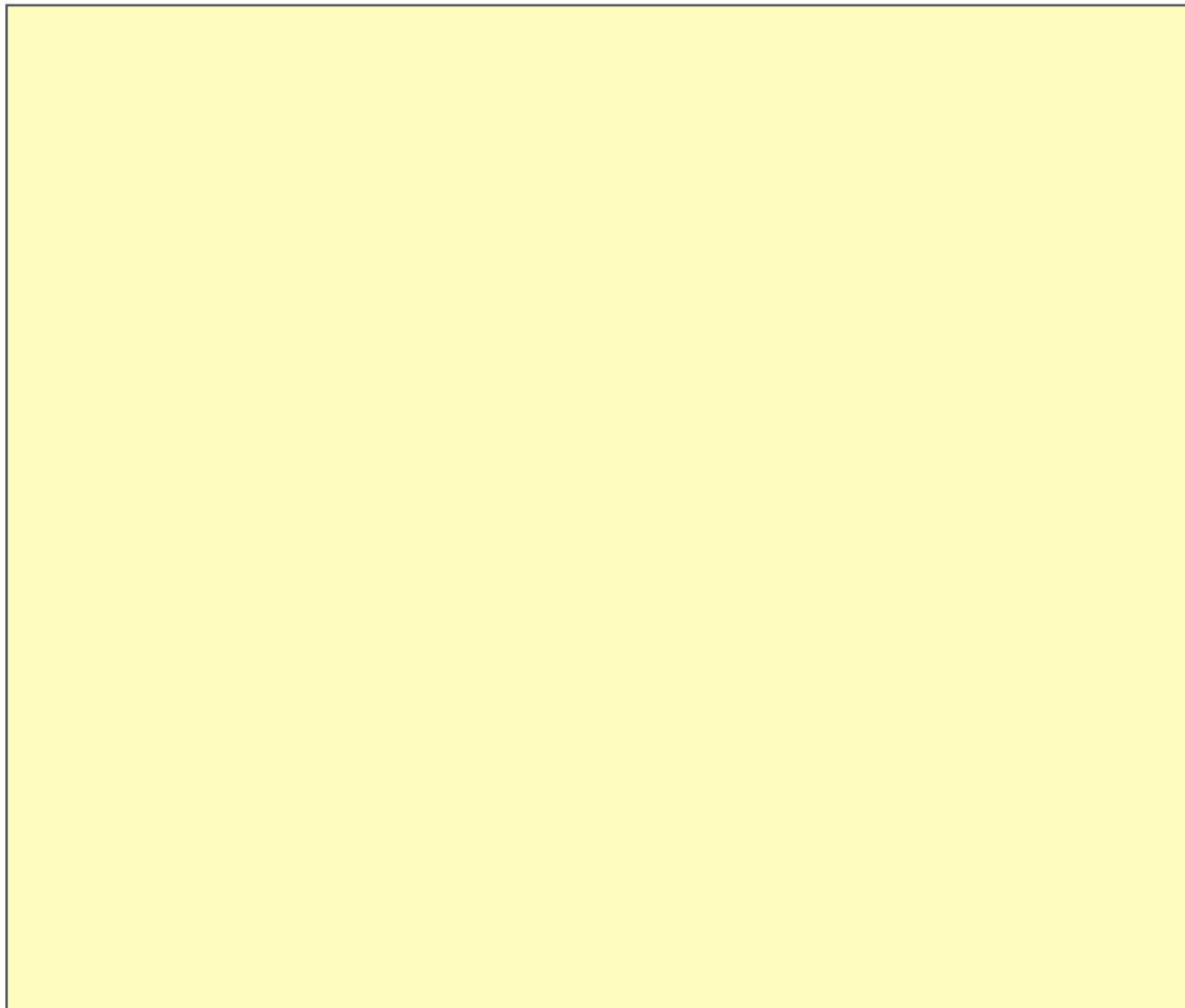
# Zeiger: Motivation mithilfe der RAM

$\sigma$
0
1
2
3
4
5
:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

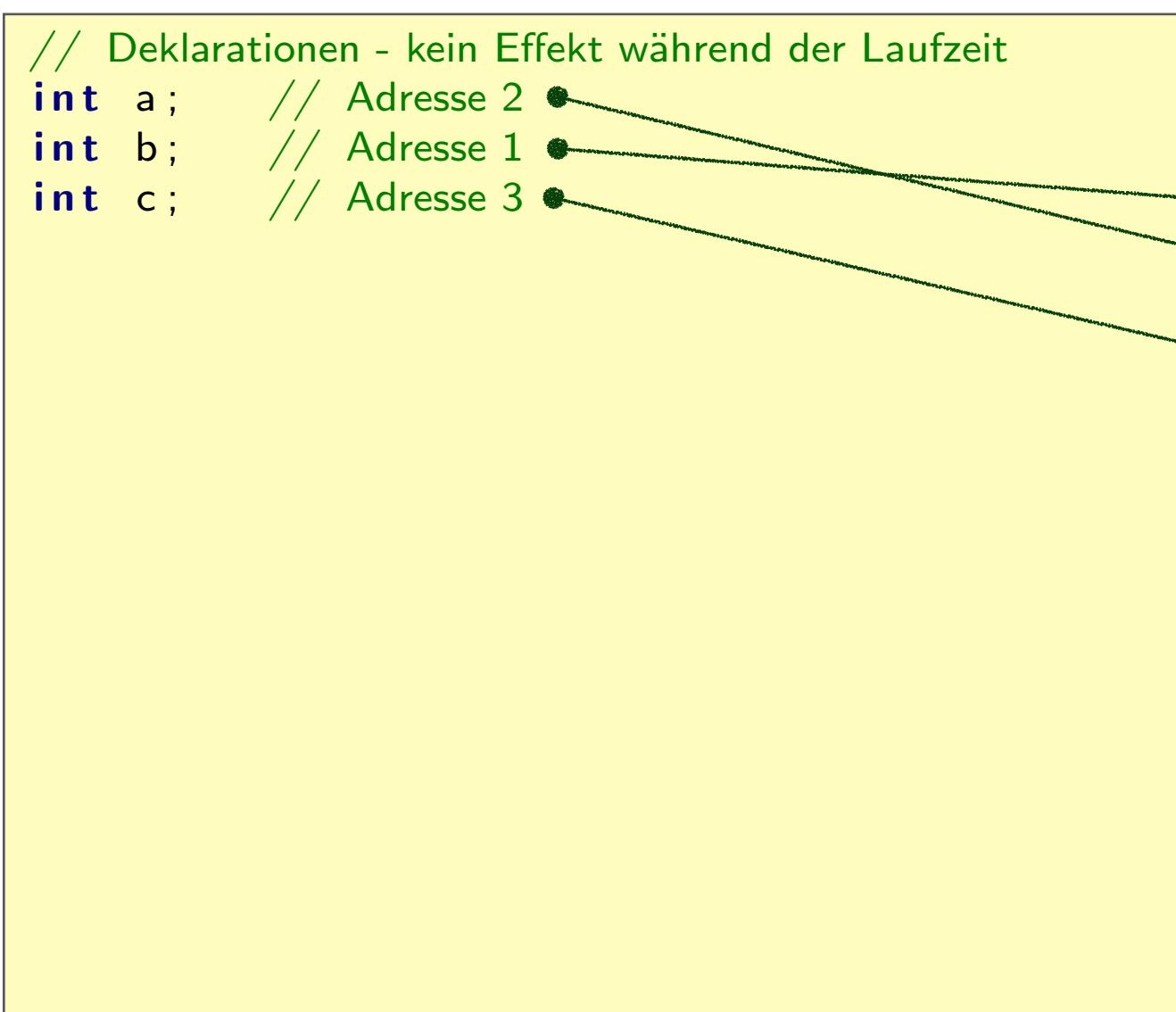


$\sigma$	
	0
	1
	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung



$\sigma$
0
1
2
3
4
5
:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen
```

$\sigma$	
	0
	1
	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen
```

$\sigma$	
	0
	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
```

```
int a; // Adresse 2
```

```
int b; // Adresse 1
```

```
int c; // Adresse 3
```

```
a = 42; // load 42 42 unmittelbar laden  
         // store 2 bei a ablegen
```

```
b = a; // load *2 Wert an Adresse 2 laden  
         // store 1 bei b ablegen
```

$\sigma$	
	0
	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
```

```
int a; // Adresse 2
```

```
int b; // Adresse 1
```

```
int c; // Adresse 3
```

```
a = 42; // load 42 42 unmittelbar laden  
         // store 2 bei a ablegen
```

```
b = a; // load *2 Wert an Adresse 2 laden  
         // store 1 bei b ablegen
```

$\sigma$	
	0
42	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen
```

$\sigma$	
	0
42	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen

p = &a;    // load 2 Adresse 2 laden
           // store 5 bei p ablegen
```

$\sigma$	
	0
42	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen

p = &a;    // load 2 Adresse 2 laden
           // store 5 bei p ablegen
```

$\sigma$	
	0
42	1
42	2
	3
	4
2	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen

p = &a;    // load 2 Adresse 2 laden
           // store 5 bei p ablegen

c = *p;    // load **5 indirekte Adressierung!
           // store 3 bei p ablegen
```

$\sigma$	
	0
42	1
42	2
	3
	4
2	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen

p = &a;    // load 2 Adresse 2 laden
           // store 5 bei p ablegen

c = *p;    // load **5 indirekte Adressierung!
           // store 3 bei p ablegen
```

$\sigma$	
	0
42	1
42	2
42	3
	4
2	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen

p = &a;    // load 2 Adresse 2 laden
           // store 5 bei p ablegen

c = *p;    // load **5 indirekte Adressierung!
           // store 3 bei p ablegen

*p = 4711; // load 4711 direkt laden
           // store *5 indirekte Adressierung!
```

$\sigma$	
	0
42	1
42	2
42	3
	4
2	5
	:

# Zeiger: Motivation mithilfe der RAM

- Variablen sind **Aliase für Speicheradressen**

Compiler/Laufzeitsystem übernimmt in der Regel die Zuordnung

```
// Deklarationen - kein Effekt während der Laufzeit
int a;      // Adresse 2
int b;      // Adresse 1
int c;      // Adresse 3

a = 42;    // load 42 42 unmittelbar laden
           // store 2 bei a ablegen

b = a;     // load *2 Wert an Adresse 2 laden
           // store 1 bei b ablegen

// Zeigervariablen speichern Adressen
int *p;    // p wird Adresse 5 zugewiesen

p = &a;    // load 2 Adresse 2 laden
           // store 5 bei p ablegen

c = *p;    // load **5 indirekte Adressierung!
           // store 3 bei p ablegen

*p = 4711; // load 4711 direkt laden
           // store *5 indirekte Adressierung!
```

$\sigma$	
	0
42	1
4711	2
42	3
	4
2	5
	:

# Zeiger: Motivation mithilfe der RAM - Aliasing

```
// Deklarationen - kein Effekt während der Laufzeit
double a;           // Adresse 2
double *p;          // Adresse 1
double **pp;         // Adresse 3
```

$\sigma$	
	0
	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM - Aliasing

- Eine Speicherzelle kann mehrere Bezeichnungen haben  
Evtl. erst zur Laufzeit entscheidbar!

```
// Deklarationen - kein Effekt während der Laufzeit
double a;           // Adresse 2
double *p;          // Adresse 1
double **pp;        // Adresse 3
```

$\sigma$	
	0
	1
42	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM - Aliasing

- Eine Speicherzelle kann mehrere Bezeichnungen haben  
Evtl. erst zur Laufzeit entscheidbar!

```
// Deklarationen - kein Effekt während der Laufzeit
double a;           // Adresse 2
double *p;          // Adresse 1
double **pp;         // Adresse 3

a = 47.11; // load 47.11 47.11 unmittelbar laden
            // store 2 bei a ablegen
```

$\sigma$	
	0
	1
47.11	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM - Aliasing

- Eine Speicherzelle kann mehrere Bezeichnungen haben  
Evtl. erst zur Laufzeit entscheidbar!

```
// Deklarationen - kein Effekt während der Laufzeit
double a;           // Adresse 2
double *p;          // Adresse 1
double **pp;        // Adresse 3

a = 47.11;           // load 47.11 47.11 unmittelbar laden
                     // store 2 bei a ablegen

p = &a;             // load 2 Adresse 2 laden
                     // store 1 bei p ablegen
```

$\sigma$	
	0
2	1
47.11	2
	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM - Aliasing

- Eine Speicherzelle kann mehrere Bezeichnungen haben  
Evtl. erst zur Laufzeit entscheidbar!

```
// Deklarationen - kein Effekt während der Laufzeit
double a;           // Adresse 2
double *p;          // Adresse 1
double **pp;         // Adresse 3

a = 47.11; // load 47.11 47.11 unmittelbar laden
            // store 2 bei a ablegen

p = &a; // load 2 Adresse 2 laden
        // store 1 bei p ablegen

pp = &p; // load 2 Adresse 1 laden
        // store 3 bei pp ablegen
```

$\sigma$	
	0
2	1
47.11	2
1	3
	4
	5
	:

# Zeiger: Motivation mithilfe der RAM - Aliasing

- Eine Speicherzelle kann mehrere Bezeichnungen haben  
Evtl. erst zur Laufzeit entscheidbar!

```
// Deklarationen - kein Effekt während der Laufzeit
double a;           // Adresse 2
double *p;          // Adresse 1
double **pp;        // Adresse 3

a = 47.11;          // load 47.11 47.11 unmittelbar laden
                    // store 2 bei a ablegen

p = &a;            // load 2 Adresse 2 laden
                    // store 1 bei p ablegen

pp = &p;           // load 2 Adresse 1 laden
                    // store 3 bei pp ablegen
```

$\sigma$	
	0
2	1
47.11	2
1	3
	4
	5
	:

Speicherzelle 2 kann über verschiedene Ausdrücke angesprochen werden:

- a,
- \*p und
- \*\*p.

Die Ausdrücke sind **Aliase** für Speicherzelle 2 - man spricht von einem **Aliasing**.

# Zeigertypen

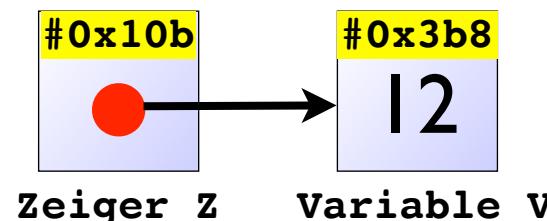
- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird

# Zeigertypen

- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird
- Ein **Zeiger** ist eine Variable, deren Wert eine **Speicheradresse** ist  
Eventuell ist dieser Speicheradresse eine andere Variablen zugeordnet - indirekte Adressierung!

# Zeigertypen

- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird
- Ein **Zeiger** ist eine Variable, deren Wert eine **Speicheradresse** ist  
Eventuell ist dieser Speicheradresse eine andere Variablen zugeordnet - indirekte Adressierung!

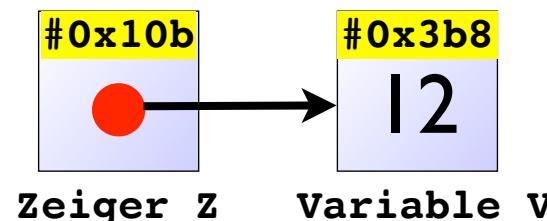


Die Variable „v“ wird durch den Zeiger „z“ **referenziert**

- Um über einen Zeiger auf den Wert einer Variablen zuzugreifen, muss dieser **derefenziert** werden

# Zeigertypen

- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird
- Ein **Zeiger** ist eine Variable, deren Wert eine **Speicheradresse** ist  
Eventuell ist dieser Speicheradresse eine andere Variablen zugeordnet - indirekte Adressierung!

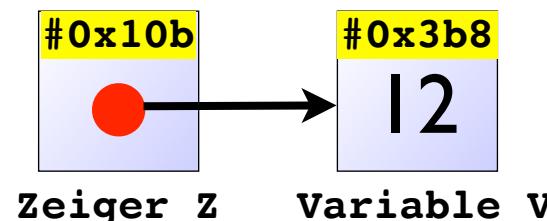


Die Variable „v“ wird durch den Zeiger „z“ **referenziert**

- Um über einen Zeiger auf den Wert einer Variablen zuzugreifen, muss dieser **derefenziert** werden
- Die referenzierte Variable ist vom **Bezugstyp**, der Zeiger ist eine Variable vom **Zeigertyp**

# Zeigertypen

- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird
- Ein **Zeiger** ist eine Variable, deren Wert eine **Speicheradresse** ist  
Eventuell ist dieser Speicheradresse eine andere Variablen zugeordnet - indirekte Adressierung!

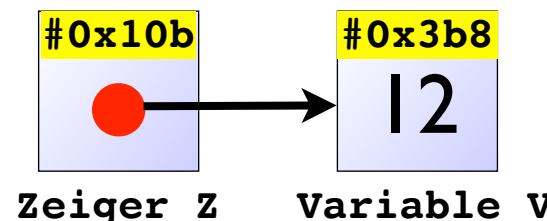


Die Variable „v“ wird durch den Zeiger „z“ **referenziert**

- Um über einen Zeiger auf den Wert einer Variablen zuzugreifen, muss dieser **derefenziert** werden
- Die referenzierte Variable ist vom **Bezugstyp**, der Zeiger ist eine Variable vom **Zeigertyp**
- **Vorsicht:**

# Zeigertypen

- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird
- Ein **Zeiger** ist eine Variable, deren Wert eine **Speicheradresse** ist  
Eventuell ist dieser Speicheradresse eine andere Variablen zugeordnet - indirekte Adressierung!

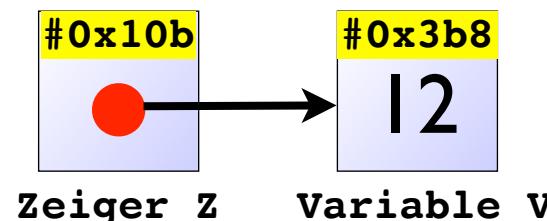


Die Variable „v“ wird durch den Zeiger „z“ **referenziert**

- Um über einen Zeiger auf den Wert einer Variablen zuzugreifen, muss dieser **derefenziert** werden
- Die referenzierte Variable ist vom **Bezugstyp**, der Zeiger ist eine Variable vom **Zeigertyp**
- **Vorsicht:**
  - In C++ wird zwischen Referenz- und Zeigertyp unterschieden!

# Zeigertypen

- Mit Hilfe von Zeigern: **dynamische Datenstrukturen** realisierbar  
Das sind Datenstrukturen, deren Speicherplatz erst zur **Laufzeit** angefordert wird
- Ein **Zeiger** ist eine Variable, deren Wert eine **Speicheradresse** ist  
Eventuell ist dieser Speicheradresse eine andere Variablen zugeordnet - indirekte Adressierung!



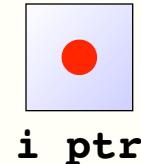
Die Variable „v“ wird durch den Zeiger „z“ **referenziert**

- Um über einen Zeiger auf den Wert einer Variablen zuzugreifen, muss dieser **derefenziert** werden
- Die referenzierte Variable ist vom **Bezugstyp**, der Zeiger ist eine Variable vom **Zeigertyp**
- **Vorsicht:**
  - In C++ wird zwischen Referenz- und Zeigertyp unterschieden!
  - In JAVA sind Zeiger oftmals implizit!

# Typische Zeigeroperationen in C/C++ I

## Definieren eines Zeigers

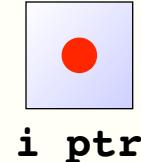
```
int *i_ptr; // Bezugstyp  
            // ist „integer“
```



# Typische Zeigeroperationen in C/C++ I

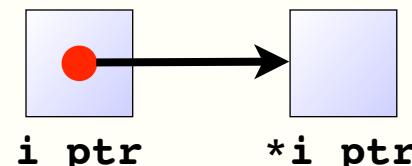
## Definieren eines Zeigers

```
int *i_ptr; // Bezugstyp  
            // ist „integer“
```



## Speicher kann dynamisch angefordert werden

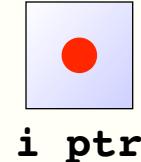
```
i_ptr = new int;
```



# Typische Zeigeroperationen in C/C++ I

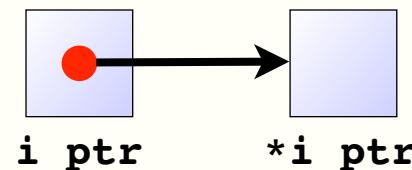
## Definieren eines Zeigers

```
int *i_ptr; // Bezugstyp  
            // ist „integer“
```



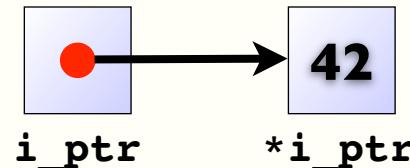
## Speicher kann dynamisch angefordert werden

```
i_ptr = new int;
```



## Dereferenzierung

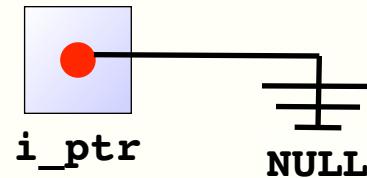
```
*i_ptr = 42;
```



# Typische Zeigeroperationen in C/C++ II

## Zuweisungen an einen Zeiger

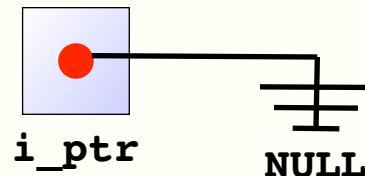
```
i_ptr = j_ptr;          // Bezugstypen müssen kompatibel sein!  
i_ptr = NULL;           // Die Konstante NULL (oder der Wert 0)  
                        // sagt, dass ein Zeiger auf keine  
                        // Variable zeigt
```



# Typische Zeigeroperationen in C/C++ II

## Zuweisungen an einen Zeiger

```
i_ptr = j_ptr;           // Bezugstypen müssen kompatibel sein!  
i_ptr = NULL;            // Die Konstante NULL (oder der Wert 0)  
                         // sagt, dass ein Zeiger auf keine  
                         // Variable zeigt
```



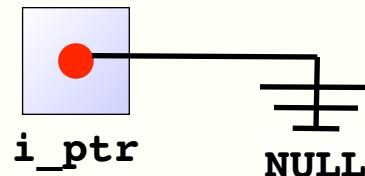
## Freigabe einer Variablen vom Bezugstyp

```
delete i_ptr; // Nach dieser Anweisung ist der  
               // Zeiger ungültig
```

# Typische Zeigeroperationen in C/C++ II

## Zuweisungen an einen Zeiger

```
i_ptr = j_ptr;           // Bezugstypen müssen kompatibel sein!  
i_ptr = NULL;           // Die Konstante NULL (oder der Wert 0)  
                        // sagt, dass ein Zeiger auf keine  
                        // Variable zeigt
```



## Freigabe einer Variablen vom Bezugstyp

```
delete i_ptr; // Nach dieser Anweisung ist der  
               // Zeiger ungültig
```

In Sprachen mit **garbage collection** (z.B. JAVA, Modula-3) wird eine mit **new** erzeugte Variable automatisch freigegeben, wenn kein Zeiger mehr auf sie verweist.

# II. Einfache Datentypen und -strukturen

---

## 2. Einfache Datentypen und -strukturen

- 2.1. Grundtypen
- 2.2. Verbund- und Zeigertypen
- 2.3. Sequenzen**
- 2.4. Stacks und Queues
- 2.5. Bäume

## D Statische Sequenz

Eine **statische Sequenz** ist eine Folge von Elementen  $(a_0, \dots, a_{n-1})$  gleichen Typs.

**Grundfunktionen** für statische Sequenzen:

- **create** $(a_0, \dots, a_{n-1})$ : erzeugt eine statische Sequenz aus den gegebenen Elementen.
- **len()**: liefert die Länge der Sequenz (also  $n$ )
- **get(i)**: liefert  $a_i$ , das  $i$ -te Element der Sequenz.
- **set(i, a)**: ändert  $a_i$  zu  $a$ .
- **iterate(f)**: iteriert über die Elemente; ruft nacheinander  $f(a_0)$  bis  $f(a_{n-1})$  auf.

## D Statische Sequenz

Eine **statische Sequenz** ist eine Folge von Elementen  $(a_0, \dots, a_{n-1})$  gleichen Typs.

**Grundfunktionen** für statische Sequenzen:

- **create** $(a_0, \dots, a_{n-1})$ : erzeugt eine statische Sequenz aus den gegebenen Elementen.
- **len()**: liefert die Länge der Sequenz (also  $n$ )
- **get(i)**: liefert  $a_i$ , das  $i$ -te Element der Sequenz.
- **set(i, a)**: ändert  $a_i$  zu  $a$ .
- **iterate(f)**: iteriert über die Elemente; ruft nacheinander  $f(a_0)$  bis  $f(a_{n-1})$  auf.

**Beachte:** Vergrößern / Verkleinern hier **nicht** möglich!

## D Statische Sequenz

Eine **statische Sequenz** ist eine Folge von Elementen  $(a_0, \dots, a_{n-1})$  gleichen Typs.

**Erweiterte Funktionen** für statische Sequenzen:

- Können durch Grundfunktionen ausgedrückt werden.
- Konkrete Implementierung u.U. effizienter möglich.
- **set\_first(a)**: entspricht `set(0, a)`.
- **set\_last(a)**: entspricht `set(n-1, a)`.
- **get\_first()**: entspricht `get(0)`.
- **get\_last()**: entspricht `get(n-1)`.

# Statische Felder als Implementierung von Sequenzen

- **Statische Felder** (Arrays) implementieren statische Sequenzen.

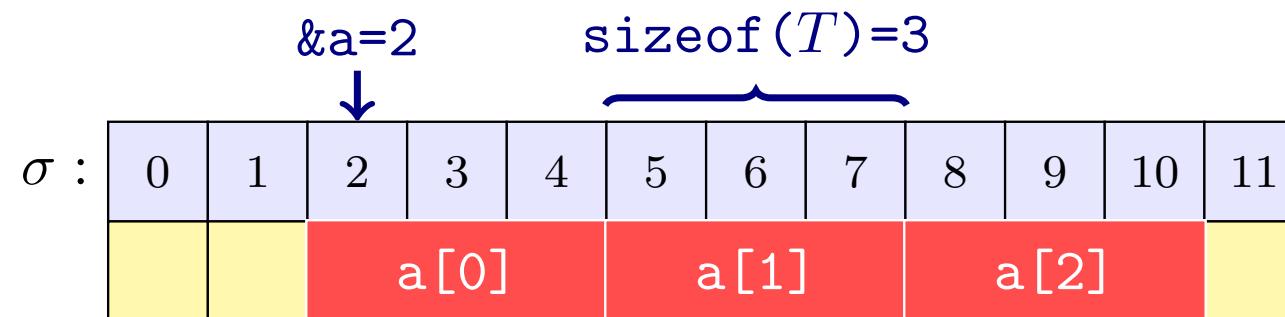
```
/* Beispiel: JAVA */
int a[] = {1,2,3,4,5,6}; // create(1,...,6)
int b = a[0];           // get(0) -> liefert 1
a[3]=42;               // set(3,42) -> überschreibt 4 mit 42
int l = a.length;       // len() -> liefert 6
for (int x : a) { // iterate() -> durchläuft das Feld a
    ...
}
```

# Statische Felder als Implementierung von Sequenzen

- **Statische Felder** (Arrays) implementieren statische Sequenzen.

```
/* Beispiel: JAVA */
int a[] = {1,2,3,4,5,6}; // create(1,...,6)
int b = a[0];           // get(0) -> liefert 1
a[3]=42;               // set(3,42) -> überschreibt 4 mit 42
int l = a.length;       // len() -> liefert 6
for (int x : a) {       // iterate() -> durchläuft das Feld a
    ...
}
```

- Die Elemente eines statischen Feldes liegen **konsekutiv** im Speicher:



- **$\&a$** : Speicheradresse, an der das statische Feld **a** im Speicher einer Word-RAM liege.
- **$\text{sizeof}(T)$** : Anzahl der Worte (Speicherzellen), die der Elementtyp **T** einnimmt.
- Das *i*-te Element **a[i]** von **a** liegt an Adresse  $\&a + i \cdot \text{sizeof}(T)$ .

# Statische Felder als Sequenzen: Analyse

Wir betrachten eine statische Sequenz aus  $n$  Elementen vom Typ  $T$ ; realisiert als statisches Feld auf einer  $w$ -Word-RAM:

- **Speicherplatz:**  $n \cdot \text{sizeof}(T)$
- **Laufzeit:**
  - **create:**  $O(n)$   
Reservieren von Speicher ggf. mit  $O(1)$ ; Initialisierung mit  $O(n)$
  - **set, get:**  $O(1)$   
... sofern  $w \geq \log_2(n)$
  - **len:**  $O(1)$   
da Länge a priori (vor Programmlauf) bekannt.
  - **iterate:**  $O(n)$

# Dynamische Sequenzen - Interface (ADT)

- Eine **dynamische Sequenz** ist eine Sequenz deren Länge variabel ist; es können Elemente hinzugefügt oder entfernt werden.
- Sie erweitert die **Grundfunktionen** der statischen Sequenz:
  - **insert\_at(i,a)**: fügt das Element  $a$  an der Indexposition  $i$  in die Sequenz ein:
$$(a_0, \dots, a_n).insert\_at(i, a) = (a_0, \dots, a_{i-1}, a, a_i, \dots, a_n)$$
  - **remove\_at(i)**: entfernt das Element an Indexposition  $i$ :
$$(a_0, \dots, a_{i-1}, a_i, a_{i+1}, a_n).delete\_at(i) = (a_0, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$$
- Sie stellt folgende, **erweiterte Funktionen** bereit:
  - **insert\_first(a)** =  $insert\_at(0, a)$  ; **insert\_last(a)** =  $insert\_at(n-1, a)$
  - **delete\_first(a)** =  $delete\_at(0)$  ; **delete\_last(a)** =  $delete\_at(n-1)$

# Dynamisch Sequenzen: Implementierung

- **Dynamische Sequenzen** werden z.B. durch folgende Datenstrukturen realisiert:
  - **Verkettete Liste**: einfach oder doppelt verkettete Listen.
  - **Dynamisches Feld**
  - **Verkettete Liste in Cursor-Darstellung**: eine Mischung aus verketterter Liste und dynamischem Feld.

- Sequenzen können wir nahezu analog zu Wörtern (siehe Theorieteil) induktiv definieren:

## D Sequenz

Sei  $T$  eine beliebige Menge. Mit  $[T]$  bezeichnen wir die Menge aller **Sequenzen von  $T$** ; sie ist induktiv definiert durch:

- $[] \in [T]$  - die **leere Sequenz**  $[]$  ist eine Sequenz vom Typ  $T$ .
- Seien  $t \in [T]$ ,  $h \in T$ , dann ist  $h : t \in [T]$ ; : nennen wir **Sequenzkonstruktor**.

Sei  $h : t$  eine Sequenz. Dann nennen wir  $h$  den **Kopf** von  $h : t$  und  $t$  den **Rest** von  $h : t$ .

# Dynamische Sequenzen; theoretischer Zugang 2

- Darauf aufbauend lassen sich die Grundfunktionen leicht als rekursive Funktionen definieren; z.B.:

len	$\text{[]}.len() = 0$ $h : t.len() = 1 + t.len()$
get	$(h : t).get(0) = h$ $(h : t).get(n) = t.get(n - 1)$ für $n > 0$
set	$(h : t).set(0, a) = a : t$ $(h : t).set(n, a) = h : t.set(n - 1, a)$ für $n > 0$
insert_at	$\text{[]}.insert\_at(0, a) = a : \text{[]}$ $(h : t).insert\_at(0, a) = a : h : t$ $(h : t).insert\_at(n, a) = h : t.insert\_at(n - 1, a)$ für $n > 0$

- **Beachte:** Aus naheliegenden Gründen verwenden wir hier eine *Methoden-Schreibweise*...

# Verkettete Liste in JAVA - Version 1 - 1

- Wir können "entlang der Theorie" entwickeln:

```
// Node<T> entspricht [T]
class Node<T> {
    T h;
    Node<T> t;

    // Konstruktor entspricht dem :
    Node(T h, Node<T> t) {
        this.h = h;
        this.t = t;
    }
}

// null entspricht dann der leeren Sequenz []
```

Sei  $T$  eine beliebige Menge. Mit  $[T]$  bezeichnen wir die Menge aller **Sequenzen von  $T$** ; sie ist induktiv definiert durch:

- $[] \in [T]$  - die **leere Sequenz**  $[]$  ist eine Sequenz vom Typ  $T$ .
- Seien  $t \in [T]$ ,  $h \in T$ , dann ist  $h : t \in [T]$ ;  $:$  nennen wir **Sequenzkonstruktor**.

- Eine Sequenz ist entweder eine **Instanz von**  $\text{Node}<\text{T}>$  oder null.
- Die Sequenz  $1:2:3:[]$  könnten wir dann so erzeugen:

```
new Node<Integer>(1, new Node<Integer>(2, new Node<Integer>(3,
null))))
```

# Verkettete Liste in JAVA - Version 1 - 2

- Umsetzung der **Grundfunktionen**; betrachten Sie `len`:

$$[].\text{len}() = 0$$

$$(h : t).\text{len} = 1 + t.\text{len}()$$

Notation impliziert *Methodenaufrufe*; auf `null`, der JAVA-Repräsentation der leeren Liste `[]`, nicht erlaubt!  $\Rightarrow$  nicht 1 : 1 in JAVA umsetzbar!

- Wie führen daher die **Proxy-Klasse LinkedList** ein, die einen Wert vom Typ `Node<T>` kapselt:

```
public class LinkedList<T> {
    class Node<T> {
        T h;
        Node<T> t;
        Node(T h, Node<T> t) {
            this.h = h; this.t = t;
        }
    }

    Node<T> theList;
}
```

# Verkettete Liste in JAVA - Version 1 - 3

- Aus der "Methode" `len()` der Theorie wird die Java-Methode `int len(Node<T>):`

```
public class LinkedList<T> {  
    ...  
    // Privat! Entspricht der "Implementierung" aus der Theorie  
    private int len(Node<T> list) {  
        if (list == null) { // Fallunterscheidung wird explizit!  
            return 0; // [].len() = 0  
        } else {  
            return 1 + len(list.t); // (h:t).len() = 1 + t.len()  
        }  
    }  
    // Public! Umsetzung Schnittstelle dyn. Sequenz  
    public int len() {  
        return len(theList);  
    }  
}
```

- **Beachte:** Dank **Proxy** kann `int len()` für beliebige Sequenz (leer oder nicht) aufgerufen werden!

# Verkettete Liste in JAVA - Version 1 - 4

- Analog verfahren wir für `get`:

```
public class LinkedList<T> {  
    ...  
    // Privat! Entspricht der "Implementierung" aus der Theorie  
    private T get(Node<T> list, int idx) {  
        if (idx == 0) {  
            return list.h;  
        } else {  
            return get(list.t, idx-1);  
        }  
    }  
    // Public! Umsetzung Schnittstelle dyn. Sequenz  
    public T get(int idx) {  
        return get(theList, idx);  
    }  
}
```

$$(h : t).get(0) = h$$

$$(h : t).get(n) = t.get(n - 1) \text{ für } n > 0$$

- Beachte:** Die Implementierung ist so noch **nicht sicher!** Warum?

# Verkettete Liste in JAVA - Version 1 - 5

- Auch für `insert_at` führen wir zwei Methoden ein:

```
public class LinkedList<T> {  
    ...  
    private Node<T> insert_at(Node<T> list, int idx, T val) {  
        if (idx == 0 && list == null) {  
            return new Node<T>(val, null);  
        } else if (idx == 0) {  
            return new Node<T>(val, list);  
        } else {  
            return new Node<T>(list.h, insert_at(list.t, idx-1, val));  
        }  
    }  
    public void insert_at(int idx, T val) {  
        theList = insert_at(theList, idx, val);  
    }  
}
```

`[].insert_at(0,a) = a : []`  
 `(h:t).insert_at(0,a) = a : h:t`  
 `(h:t).insert_at(n,a) = h:t.insert_at(n-1,a) für n > 0`

- Wie im theoretischen Modell wird komplett neue Sequenz erzeugt!

# Verkettete Liste in JAVA - Version 1 - 5

- Auch für `insert_at` führen wir zwei Methoden ein:

```
public class LinkedList<T> {  
    ...  
    private Node<T> insert_at(Node<T> list, int idx, T val) {  
        if (idx == 0 && list == null) {  
            return new Node<T>(val, null);  
        } else if (idx == 0) {  
            return new Node<T>(val, list);  
        } else {  
            return new Node<T>(list.h, insert_at(list.t, idx-1, val));  
        }  
    }  
    public void insert_at(int idx, T val) {  
        theList = insert_at(theList, idx, val);  
    }  
}
```

`[].insert_at(0,a) = a : []`  
`(h:t).insert_at(0,a) = a : h:t`  
`(h:t).insert_at(n,a) = h:t.insert_at(n-1,a) für n > 0`

- Wie im theoretischen Modell wird komplett neue Sequenz erzeugt!
- **Beachte:** Die Implementierung ist so **nicht effizient** und auch nicht sicher! Warum?

# Verkettete Liste in JAVA - Version 1 - 6

- Bei der Umsetzung von `set` weichen wir von der Theorie ab:

```
public class LinkedList<T> {  
    ...  
  
    private void set(Node<T> list, int idx, T val) {  
        if (list == null) return;  
        if (idx == 0) {  
            list.h = val; // <- Seiteneffekt!  
        } else {  
            set(list.t, idx-1, val);  
        }  
    }  
  
    public void set(int idx, T val) {  
        set(theList, idx, val);  
    }  
}
```

$$\begin{aligned}(h : t).\text{set}(0, a) &= a : t \\ (h : t).\text{set}(n, a) &= h : t.\text{set}(n - 1, a) \text{ für } n > 0\end{aligned}$$

- `set` erzeugt hier keine neue Sequenz! JAVA erlaubt, dass die Sequenz als **Seiteneffekt** der Funktion ändern!
- Kann man noch mehr "Eigenheiten" von JAVA ausnutzen, die zu schnellerem Code führen?

# Verkettete Liste in JAVA - Version 1 - Diskussion

- Implementierung bisher **nicht sicher** und **nicht effizient**  
Bevor wir Verbesserungen angehen, wollen den aktuellen Ansatz besser verstehen
- Datenstruktur **Node<T>** ist rekursiv definiert  
Passend zu unserer induktiven Definition einer Sequenz:

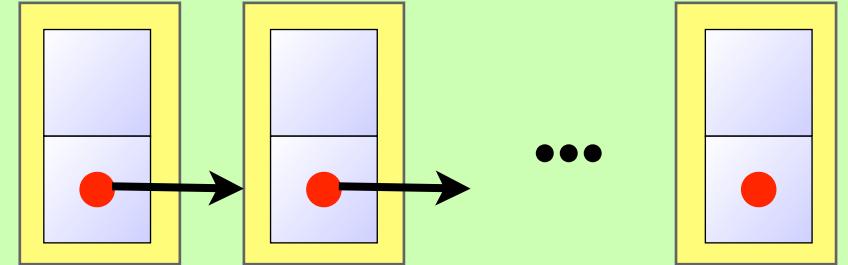
```
class Node<T> {  
    T      h ;  
    Node<T> t ;  
}
```

Wie bildet JAVA Instanzen von **Node<T>** im Speicher ab?  
Eine "unendliche Schachtelung" ist nicht möglich!

# Verkettete Liste in C++ - 1

- Ein Blick in Richtung C++ ist vielleicht aufschlussreich:

```
// Einführung von Typ-Parameter in C++
template <typename T>
// struct wie class, aber alles public
struct Node {
    T h;
    Node<T> *t; // ohne * Fehler: incomplete type
};
```



- **Node<T>\*** ist Zeiger auf eine Instanz der Struktur **Node<T>**  
Die verkettete Liste wird von Zeigern gebildet.
- Technisch gesehen löst JAVA das Problem der unendlichen Schachtelung ebenfalls durch Zeiger!  
Zeiger sind in JAVA aber implizit.

# Verkettete Liste in C++ - 2

- Die Umsetzung von `len` ist in C++ ähnlich:

```
template <typename T>
int len(Node<T>* list) {
    // "nil" ist in C++ "nullptr"
    if (list == nullptr) {
        return 0;
    } else {
        // Bei Zeigern wird mit "->" auf die Komponenten zugegriffen
        return 1 + len(list->t);
    }
}
```

- `get` lässt sich ebenfalls leicht portieren:

```
template <typename T>
T get(Node<T>* list, int idx) {
    if (idx == 0) {
        return list->h;
    } else {
        return get(list->t, idx - 1);
    }
}
```

```
private T get(Node<T> list, int idx) {
    if (idx == 0) {
        return list.h;
    } else {
        return get(list.l, idx-1);
    }
}
```

# Einschub: *Call by Reference* vs. *Call by Value* 1

- Beim Aufruf einer Funktion / Methode werden Argumente an Parameter gebunden; die Semantik dieser Bindung ist Teil der **Aufrufkonvention**.
- Im Wesentlichen werden zwei Arten der Bindung unterschieden: der Parameter agiert innerhalb der Funktion
  - als Kopie des Arguments (**Call by Value**) oder
  - als Alias für des Arguments (**Call by Reference**).
- Im Falle von *Call by Reference* werden Änderungen am Parameter innerhalb der Funktion als **Seiteneffekt** der Funktion im Argument sichtbar.

# Einschub: *Call by Reference* vs. *Call by Value* 2

Beispiele:

Call by	C++	C#
Value	<pre>int f( int para ) {     para++;     return para; }  int arg = 1; int res = f( arg ); // res = 2 ; arg = 1</pre>	<pre>int f( int para ) {     para++;     return para; }  int arg = 1; int res = f( arg ); // res = 2 ; arg = 1</pre>
Reference	<pre>int f( int &amp;para ) {     para++;     return para; }  int arg = 1; int res = f( arg ); // res = 2 ; arg = 2</pre>	<pre>int f( ref int para ) {     para++;     return para; }  int arg = 1; int res = f( ref arg ); // res = 2 ; arg = 1</pre>

**Beachte:** In C# kann man sowohl an der Deklaration des Parameters als auch an der *Call Site* erkennen, ob *Call by Reference* zum Einsatz kommt.

# Verkettete Liste in C++ - 3

- Die Umsetzung von `insert_at` ist etwas komplexer.  
Dank *Call by Reference* könnte man in C++ auf einen Proxy verzichten:

```
template <typename T>
// Beachte: l ist durch & ein "Alias" für das konkrete Argument
void insert_at(Node<T>*& l, int idx, T value) {
    if (idx == 0) {
        // Beachte: Hier wird das konkrete Argument geändert!
        l = new Node<T>{.h=value, .t=*l};
    } else {
        // Die Komponente t wird "by reference" übergeben.
        insert_at(l->t, idx-1, value);
    }
}
```

- Beachte:** Hier wird keine neue Sequenz erzeugt!  
Stattdessen: neuer Knoten wird eingehängt. Auch für JAVA möglich; mangels Zeigern aber nicht so "einfach", da JAVA *Call by Reference* nicht unterstützt.

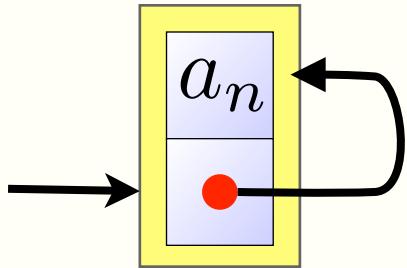
# Varianten in der Darstellung (I)

# Varianten in der Darstellung (I)

- **Varianten beim Listenende:**

# Varianten in der Darstellung (I)

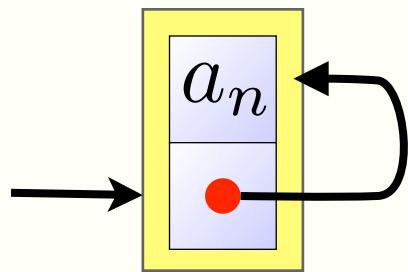
- **Varianten beim Listenende:**



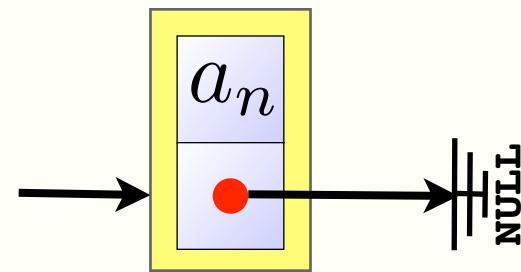
a) Letzter Knoten zeigt auf  
sich selbst

# Varianten in der Darstellung (I)

- **Varianten beim Listenende:**



a) Letzter Knoten zeigt auf sich selbst



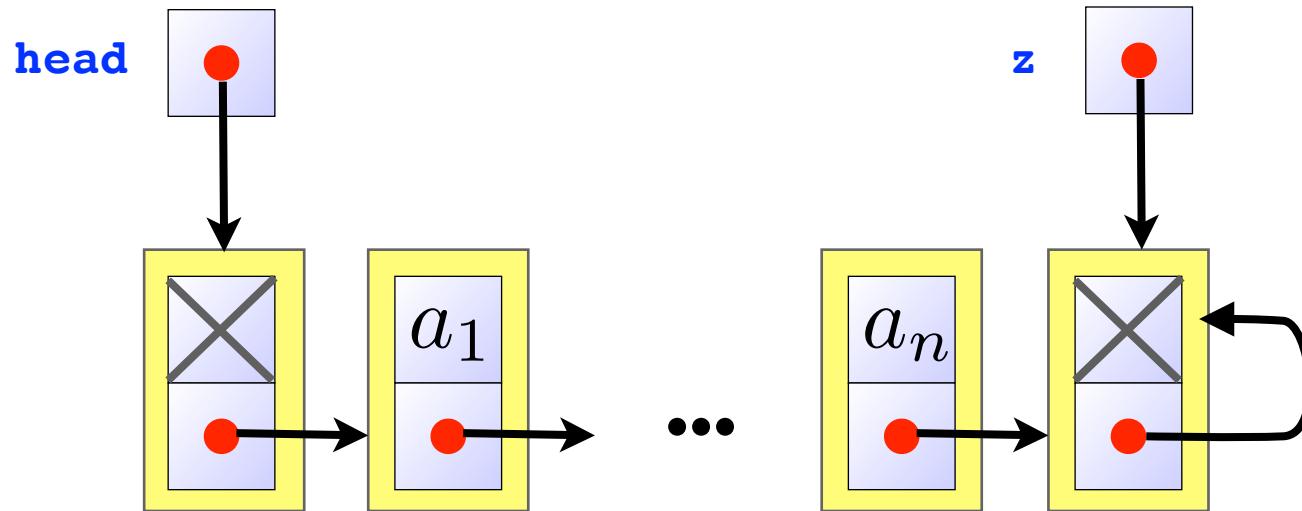
b) Letzter Knoten zeigt auf NULL

# Varianten in der Darstellung (2)

- **Verwendung von Dummy-Elementen:**

# Varianten in der Darstellung (2)

- **Verwendung von Dummy-Elementen:**



**Zwei zusätzliche Knoten `head` und `z` markieren Anfang und Ende einer Liste - sie speichern selbst **keine** Listenelemente**

Man nennt sie daher auch *Dummy-Elemente*; in Programmiersprachen ohne Zeiger/Referenzen führt diese Darstellung ggf. zu **einfacherem** Programmcode

# Doppelt verkettete Liste

# Doppelt verkettete Liste

- Ein Knoten speichert zusätzlich einen Zeiger auf den Vorgängerknoten:

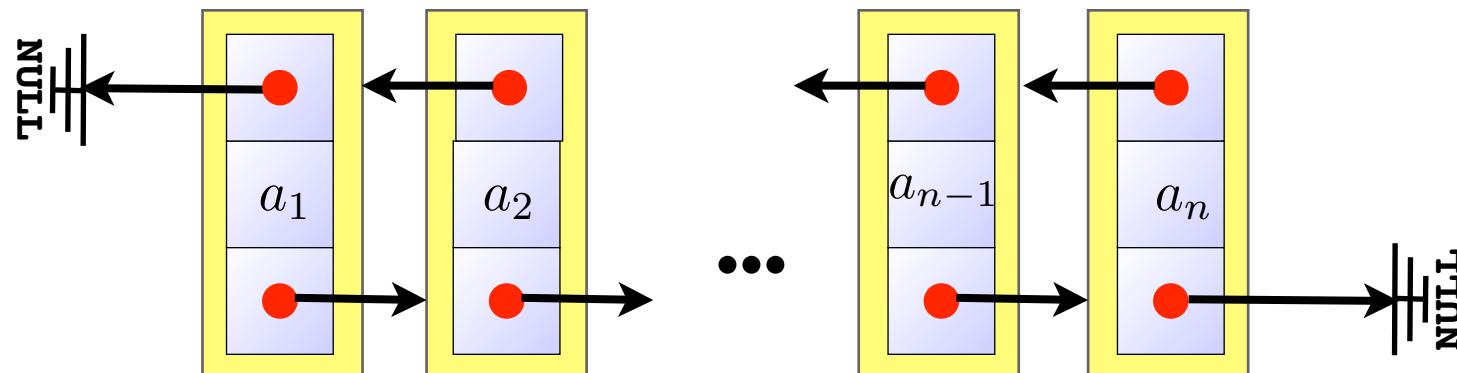
```
typedef int elemType; // Typ der Listenelmente
struct node {
    node* previous; // Zeiger auf Vorgängerknoten
    elemType value; // Wert des Elements
    node* next; // Zeiger auf Nachfolgeknoten
};
```

# Doppelt verkettete Liste

- Ein Knoten speichert zusätzlich einen Zeiger auf den Vorgängerknoten:

```
typedef int elemType; // Typ der Listenelmente
struct node {
    node* previous; // Zeiger auf Vorgängerknoten
    elemType value; // Wert des Elements
    node* next; // Zeiger auf Nachfolgeknoten
};
```

- Schematische Darstellung:

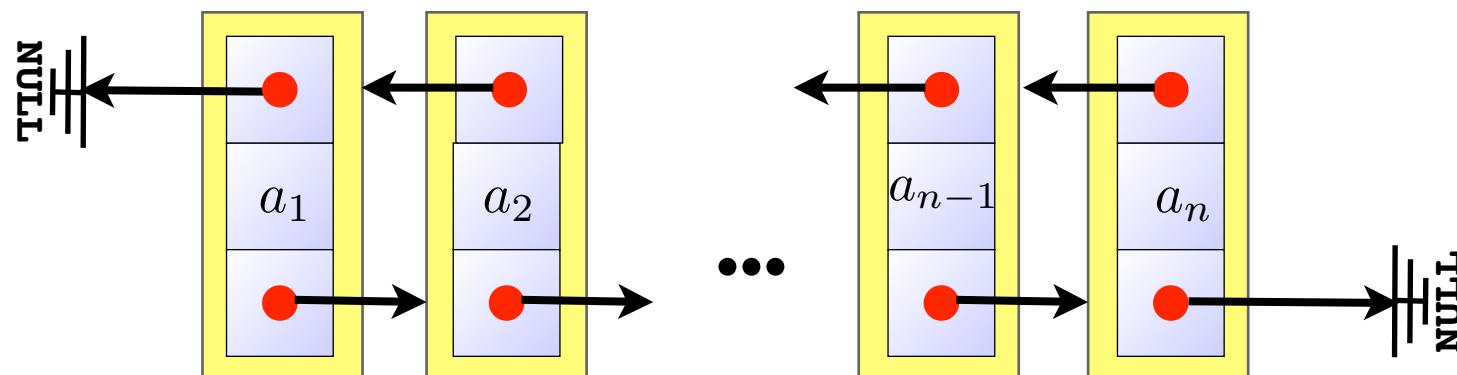


# Doppelt verkettete Liste

- Ein Knoten speichert zusätzlich einen Zeiger auf den Vorgängerknoten:

```
typedef int elemType; // Typ der Listenelmente
struct node {
    node* previous; // Zeiger auf Vorgängerknoten
    elemType value; // Wert des Elements
    node* next; // Zeiger auf Nachfolgeknoten
};
```

- Schematische Darstellung:



Übung ...

# Linked List: Listenoperationen I

# Linked List: Listenoperationen I

- **Leere Liste anlegen** (Implementierung mit Dummies)

```
node *head, *z;  
  
void initList() {  
    head = new node;  
    z    = new node;  
    head->next=z;  
    z->next=z;  
}
```

# Linked List: Listenoperationen I

- **Leere Liste anlegen** (Implementierung mit Dummies)

```
node *head, *z;  
  
void initList() {  
    head = new node;  
    z    = new node;  
    head->next=z;  
    z->next=z;  
}
```

- **Nachfolger von t löschen**

```
void deleteNext(node* t) {  
    node* toDel = t->next;  
    t->next = t->next->next;  
    delete toDel;  
}
```

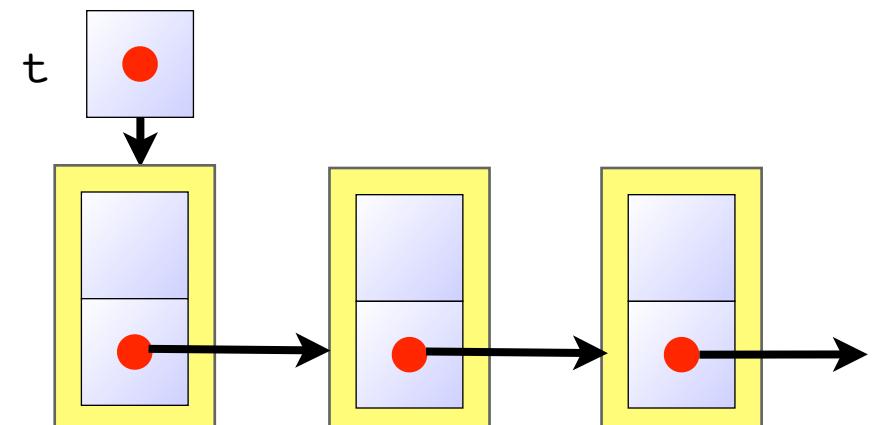
# Linked List: Listenoperationen I

- **Leere Liste anlegen** (Implementierung mit Dummies)

```
node *head, *z;  
  
void initList() {  
    head = new node;  
    z    = new node;  
    head->next=z;  
    z->next=z;  
}
```

- **Nachfolger von t löschen**

```
void deleteNext(node* t) {  
    node* toDel = t->next;  
    t->next = t->next->next;  
    delete toDel;  
}
```



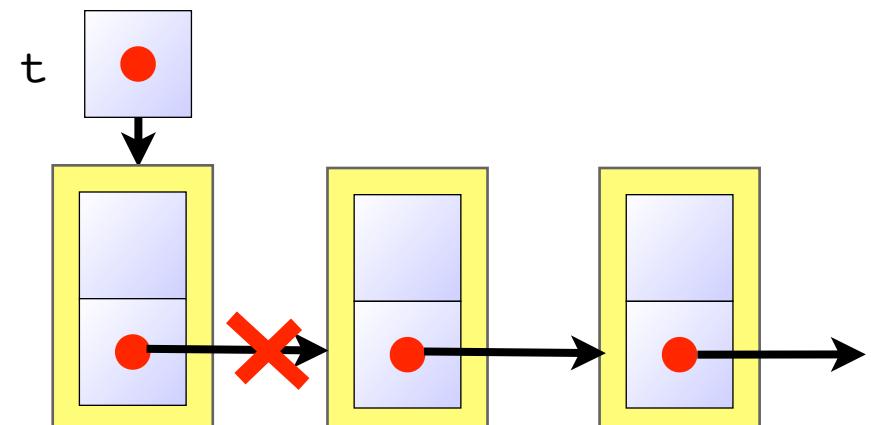
# Linked List: Listenoperationen I

- **Leere Liste anlegen** (Implementierung mit Dummies)

```
node *head, *z;  
  
void initList() {  
    head = new node;  
    z    = new node;  
    head->next=z;  
    z->next=z;  
}
```

- **Nachfolger von t löschen**

```
void deleteNext(node* t) {  
    node* toDel = t->next;  
    t->next = t->next->next;  
    delete toDel;  
}
```



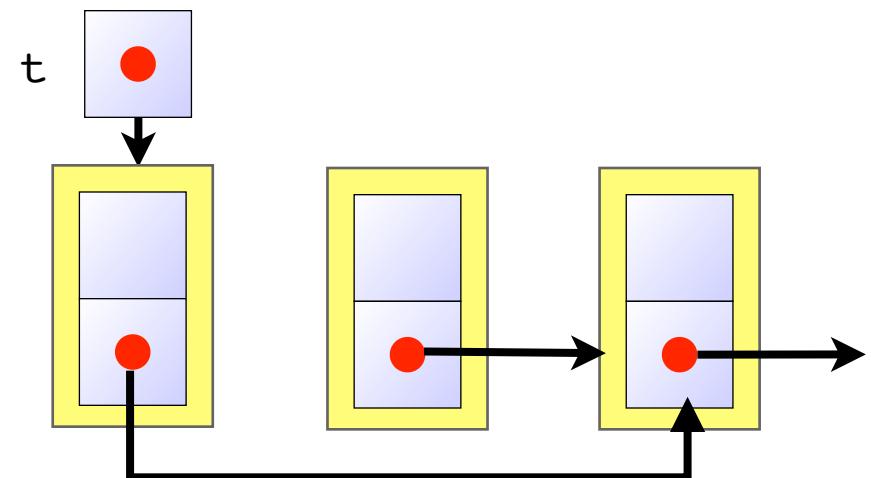
# Linked List: Listenoperationen I

- **Leere Liste anlegen** (Implementierung mit Dummies)

```
node *head, *z;  
  
void initList() {  
    head = new node;  
    z    = new node;  
    head->next=z;  
    z->next=z;  
}
```

- **Nachfolger von t löschen**

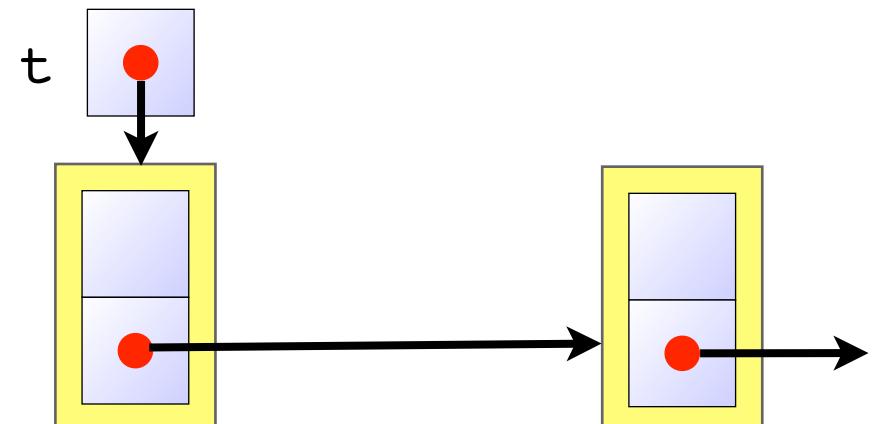
```
void deleteNext(node* t) {  
    node* toDel = t->next;  
    t->next = t->next->next;  
    delete toDel;  
}
```



# Linked List: Listenoperationen II

- **Nachfolger von t einfügen**

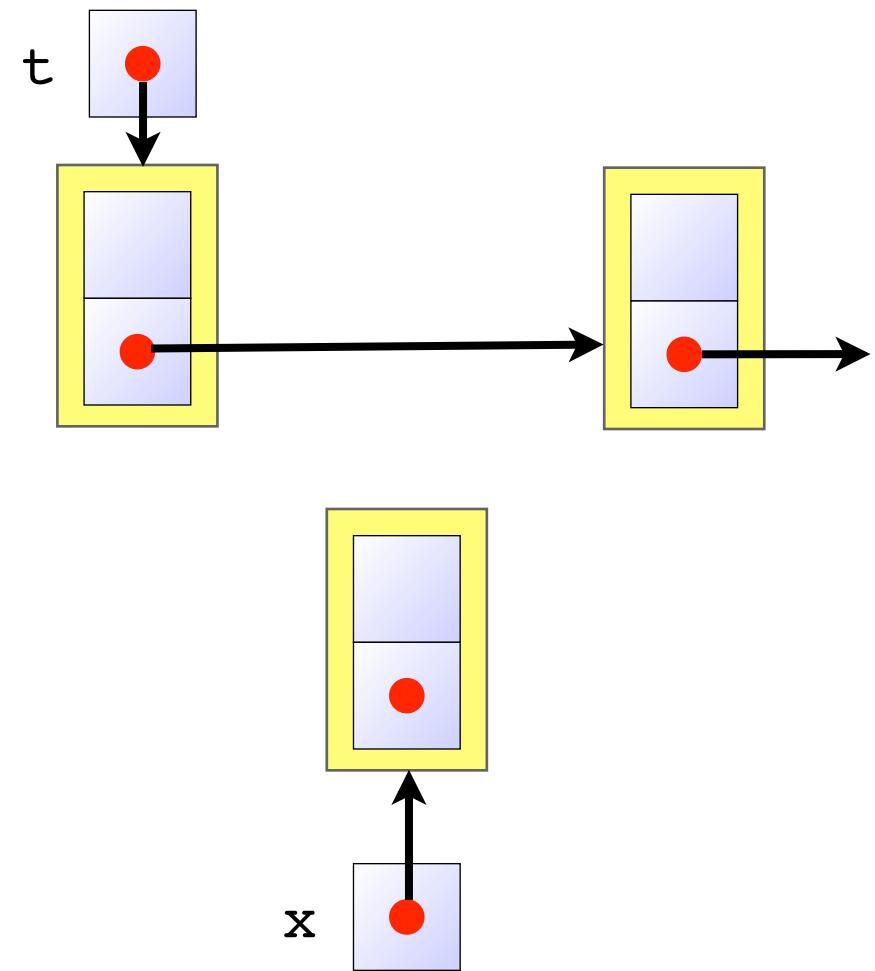
```
void insertAfter(elemType e,
                 node* t) {
    node* x;
    x = new node;
    x->value = e;
    x->next = t->next;
    t->next = x;
}
```



# Linked List: Listenoperationen II

- **Nachfolger von t einfügen**

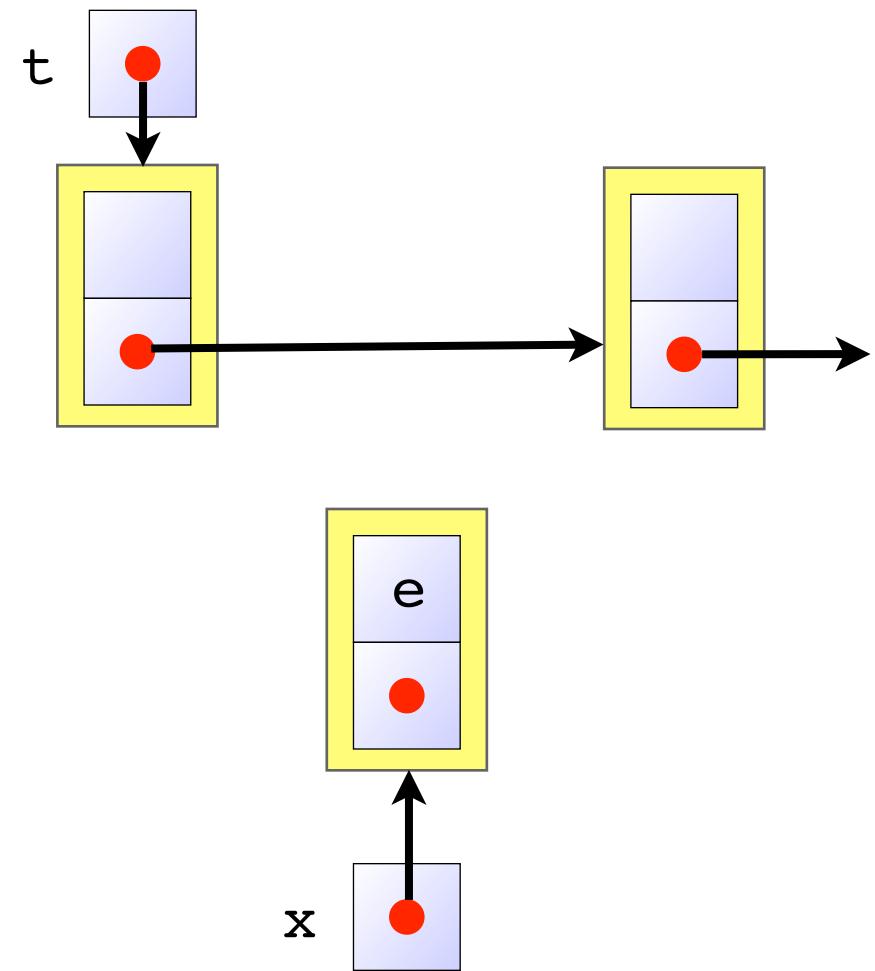
```
void insertAfter(elemType e,
                 node* t) {
    node* x;
    x = new node;
    x->value = e;
    x->next = t->next;
    t->next = x;
}
```



# Linked List: Listenoperationen II

- **Nachfolger von t einfügen**

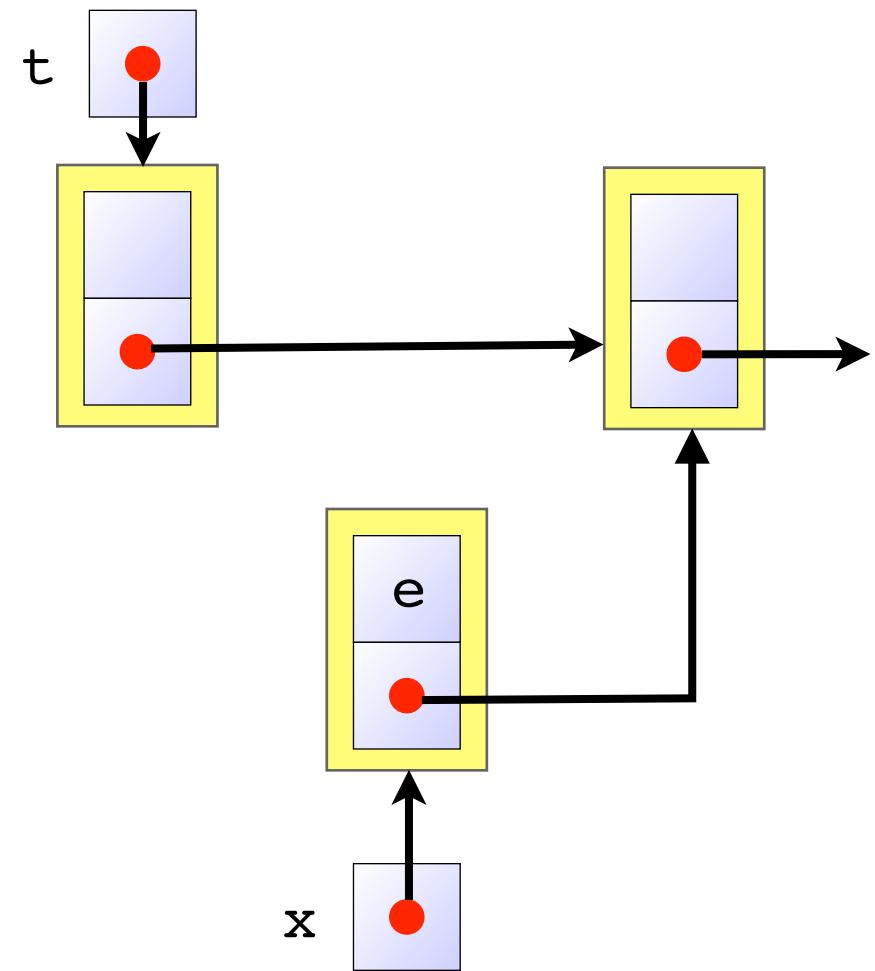
```
void insertAfter(elemType e,
                 node* t) {
    node* x;
    x = new node;
    x->value = e;
    x->next = t->next;
    t->next = x;
}
```



# Linked List: Listenoperationen II

- **Nachfolger von t einfügen**

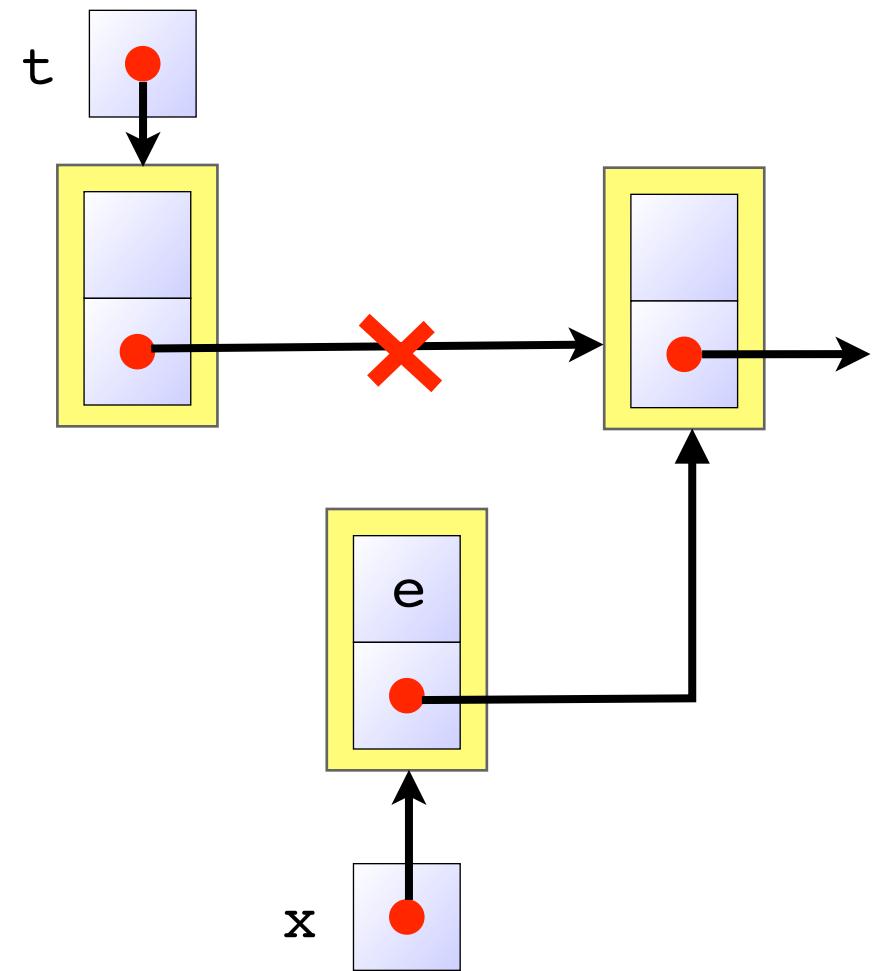
```
void insertAfter(elemType e,
                 node* t) {
    node* x;
    x = new node;
    x->value = e;
    x->next = t->next;
    t->next = x;
}
```



# Linked List: Listenoperationen II

- **Nachfolger von t einfügen**

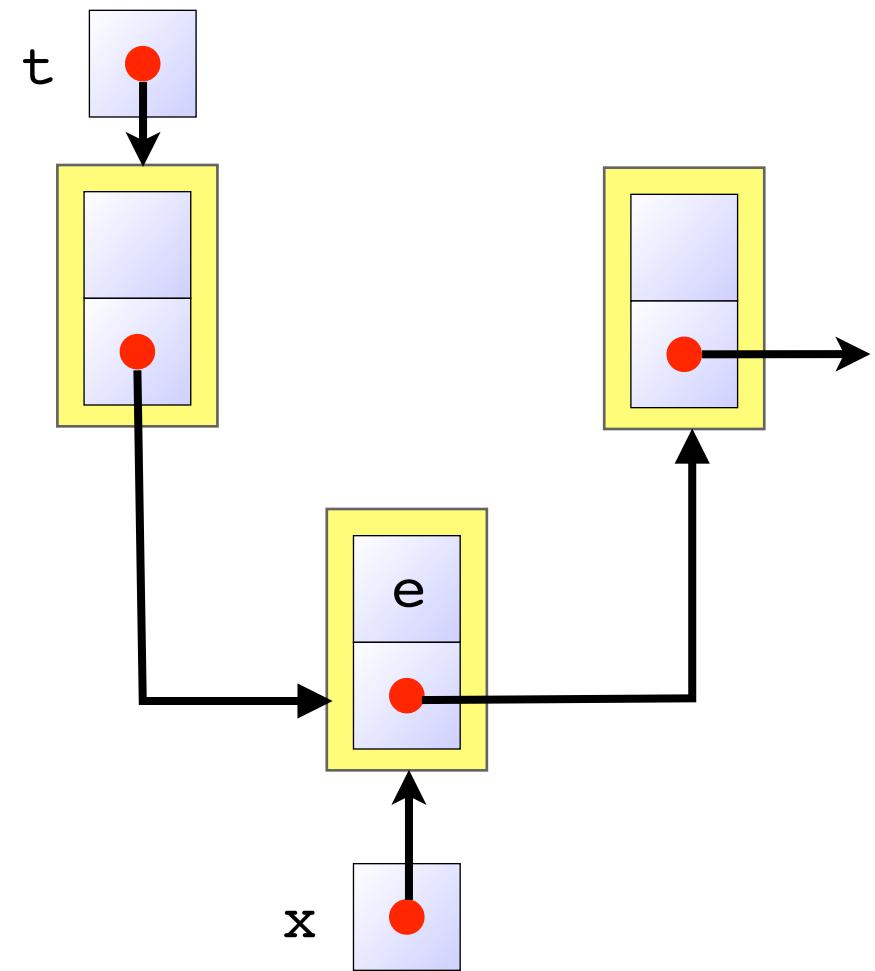
```
void insertAfter(elemType e,
                 node* t) {
    node* x;
    x = new node;
    x->value = e;
    x->next = t->next;
    t->next = x;
}
```



# Linked List: Listenoperationen II

- **Nachfolger von t einfügen**

```
void insertAfter(elemType e,
                 node* t) {
    node* x;
    x = new node;
    x->value = e;
    x->next = t->next;
    t->next = x;
}
```



# Verkettete Liste: Analyse

Wir betrachten eine dynamische Sequenz aus  $n$  Elementen vom Typ  $T$ ; realisiert als verkettete Liste auf einer  $w$ -Word-RAM:

- **Speicherplatz:**  $n \cdot (\text{sizeof}(T) + 1)$

Annahme: Zeiger belegt ein Wort

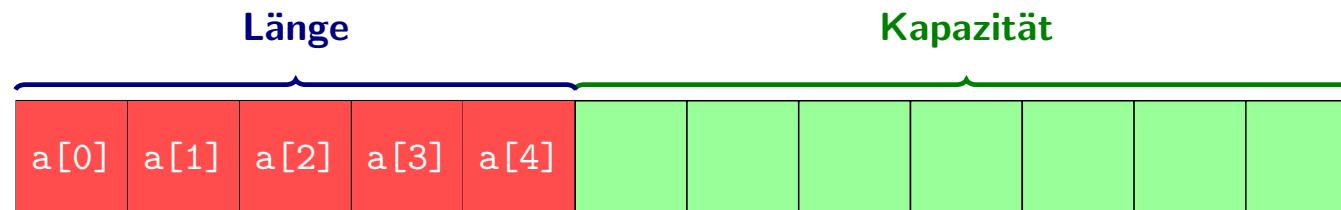
- **Laufzeit:**

- **create:**  $O(n)$
- **insert\_first, delete\_first:**  $O(1)$   
Startknoten wird "ausgehängt".
- **insert\_last, delete\_last:**  $O(n)$   
`insert_last` mit  $O(1)$  durch Zeiger auf letztes Element.
- **insert\_at, delete\_at:**  $O(n)$
- **set, get:**  $O(n)$
- **len:**  $O(n)$
- **iterate:**  $O(n)$

# Dynamische Felder

Ein **dynamisches Feld** ist ein Feld, bei dem nicht alle Einträge in Verwendung sein müssen.

- Es hat eine **Länge**  $n$   
Anzahl der Elemente in der Sequenz.
- und eine **Kapazität**  $c$ .  
Anzahl der Elemente für die noch Speicherplatz vorhanden ist.



Zur Datenstruktur gehören oft noch

- ein **Wachstumsfaktor**  $f$   
Gibt an, um welchen Faktor das Feld vergrößert wird, wenn Kapazität erschöpft (oft Verdoppelung)
- und ein **minimaler Belegungsgrad**  
Falls unterschritten, wird das Feld wieder verkleinert.

# Dynamische Felder: Analyse

Wir betrachten eine dynamisches Feld von Elementen vom Typ  $T$  mit Länge  $n$  und Kapazität von  $c$ ; realisiert als statisches Feld auf einer  $w$ -Word-RAM:

- **Speicherplatz**:  $(n + c) \cdot \text{sizeof}(T)$
- **Laufzeit**:
  - **create**:  $O(1)$
  - **set, get**:  $O(1)$   
... sofern  $w \geq \log_2(n)$
  - **len**:  $O(1)$   
Attribut des dynamischen Feldes.
  - **iterate**:  $O(n)$
  - **insert\_last**:  $O(1)$   
solange  $c \neq 0!$   $\rightarrow$  amortisierte Analyse s.n.F.

# Dynamische Felder: Amortisierte Laufzeit `insert_last`

Sofern bei `insert_last` die Kapazität des Feldes erschöpft ist:

1. reserviere Speicher für  $f \cdot n$  Elemente mit  $\theta(1)$  Operationen,  
 $f$  mal so viel Speicher wie vorher → Speicherverbrauch wächst exponentiell!
2. kopiere Feld in neuen Speicherbereich mit  $\theta(n)$  Kopiervorgängen,
3. schreibe neues Element an  $a[n]$  und passe  $n$  und  $c$  an (jeweils mit  $\theta(1)$  Operationen).

Kosten für einen Vergrößerungsvorgang:  $\theta(n)$

Nach  $n = f^N$  Einfügungen am Ende summieren sich die Kosten zu

$$\sum_{i=1}^N f^i \in \theta(f^N) = \theta(n)$$

Geometrische Reihe - letztes Glied dominiert Wachstum.

Pro Einfügung somit  $O(1)$  - man spricht von **amortisierter Laufzeit**.

# Gegenüberstellung der Alternativen

- **Laufzeiten:**

Typ	build	statisch	dynamisch		
		get_at set_at	insert_first delete_first	insert_last delete_last	insert_at delete_at
Statisches Feld	$n$	1	$n$	$n$	$n$
Verkettete Liste	$n$	$n$	1	$n$	$n$
Dynamisches Feld	$n$	1	$n$	1	$n$

# Gegenüberstellung der Alternativen

- **Laufzeiten:**

Typ	build	statisch	dynamisch		
		get_at set_at	insert_first delete_first	insert_last delete_last	insert_at delete_at
Statisches Feld	$n$	1	$n$	$n$	$n$
Verkettete Liste	$n$	$n$	1	$n$	$n$
Dynamisches Feld	$n$	1	$n$	1	$n$



Amortisierte Kosten!

# Array-Implementierung von Listen: Cursor I

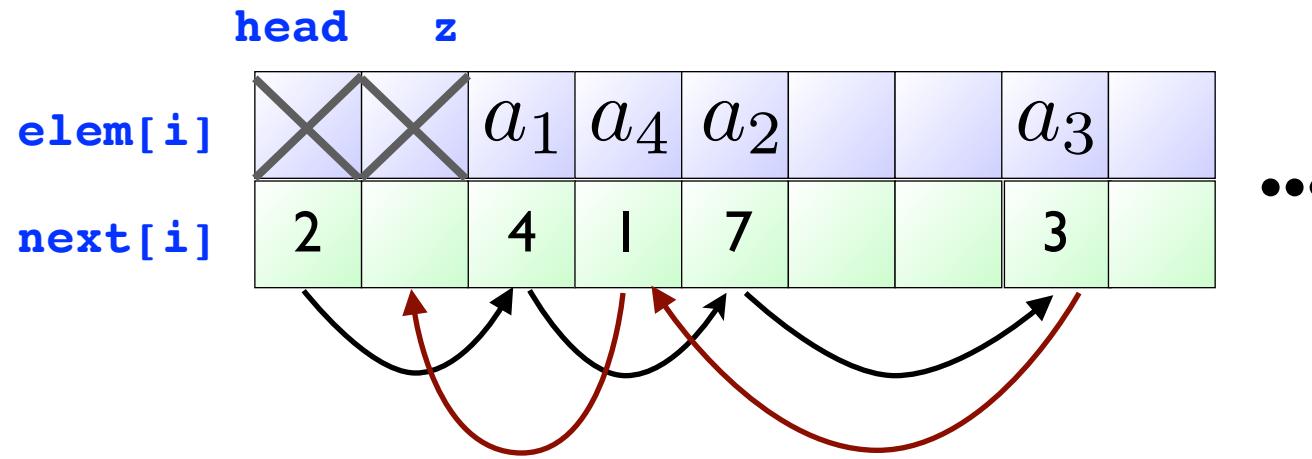
- **Cursor-Darstellung**

Stark angelehnt an Zeiger-Darstellung (*linked list*)

# Array-Implementierung von Listen: Cursor I

## ● Cursor-Darstellung

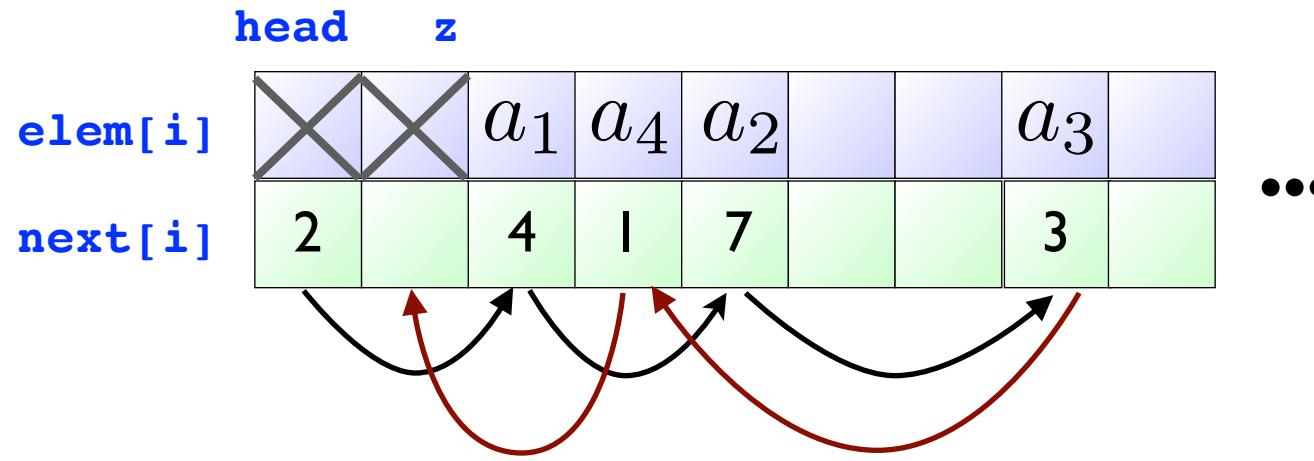
Stark angelehnt an Zeiger-Darstellung (*linked list*)



# Array-Implementierung von Listen: Cursor I

## ● Cursor-Darstellung

Stark angelehnt an Zeiger-Darstellung (*linked list*)



```
class cursorList {  
    elemType value[N]; // const int N=...  
    int next[N];  
    int head,z,x;  
public:  
    cursorList() : head(0),z(1),x(1) {  
        next[head]=z; next[z]=z;  
    }  
    ...  
};
```

# Array-Implementierung von Listen: Cursor II

- **insertAfter und deleteNext:**

```
class cursorList {  
    ...  
    // Verbesserungswürdig: freigegebene Positionen werden  
    // nicht wiederverwendet.  
    void deleteNext(int t) {  
        next[t]=next[next[t]];  
    }  
  
    // Vorsicht: testet nicht, ob das Feld bereits voll ist  
    void insertAfter(elemType e,int t) {  
        x := x + 1;  
        value[x] = e;  
        next[x] = next[t];  
        next[t] = x ;  
    }  
};
```

# Gegenüberstellung der Alternativen

# Gegenüberstellung der Alternativen

## ● **Statisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (Keine Speicher für Verwaltungsstrukturen)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- Einfügen / Löschen von Elementen immer teuer!

# Gegenüberstellung der Alternativen

## ● **Statisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (Keine Speicher für Verwaltungsstrukturen)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- Einfügen / Löschen von Elementen immer teuer!

## ● **Dynamisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (2 zusätzliche int-Werte)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- ⊕ Einfügen / Löschen meist günstig!

# Gegenüberstellung der Alternativen

## ● **Statisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (Keine Speicher für Verwaltungsstrukturen)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- Einfügen / Löschen von Elementen immer teuer!

## ● **Dynamisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (2 zusätzliche int-Werte)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- ⊕ Einfügen / Löschen meist günstig!

## ● **Einfach und doppelt verkettete Liste**

- ✓ maximale Anzahl der Elemente variabel (durch Speicherplatz beschränkt)
- ✓ Einfügen und löschen mit konstantem Aufwand
- ✓ Freispeicherverwaltung wird vom System übernommen
- Aber: Overhead durch Systemaufrufe
- Wahlfreier Zugriff auf Elemente aufwendig

# Gegenüberstellung der Alternativen

## ● **Statisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (Keine Speicher für Verwaltungsstrukturen)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- Einfügen / Löschen von Elementen immer teuer!

## ● **Dynamisches Feld**

- ✓ schneller Element-Zugriff (Elemente liegen konsekutiv im Speicher)
- ✓ Speichereffizient (2 zusätzliche int-Werte)
- maximale Sequenzlänge fix (bzw. sehr aufwendig zu ändern)
- ⊕ Einfügen / Löschen meist günstig!

## ● **Einfach und doppelt verkettete Liste**

- ✓ maximale Anzahl der Elemente variabel (durch Speicherplatz beschränkt)
- ✓ Einfügen und Löschen mit konstantem Aufwand
- ✓ Freispeicherverwaltung wird vom System übernommen
- Aber: Overhead durch Systemaufrufe
- Wahlfreier Zugriff auf Elemente aufwendig

## ● **Cursor-Darstellung**

- ✓ Einfügen und Löschen mit konstantem Aufwand
- maximale Größe vorgegeben
- spezielle Speicherverwaltung notwendig

# Listen in C++ und JAVA

- **C++ stellt mit der Standard-Template-Library (STL) folgende Sequenztypen als Templates Verfügung:**
  - `list<T,alloc>` Doppelt verkettete Liste
  - `forward_list<T,alloc>` Einfach verkettete Liste
  - `vector<T,alloc>` Dynamisches Feld
- **In JAVA:**
  - `LinkedList<T>` Doppelt verkettete Liste
  - `ArrayList<T>` Dynamisches Feld

# II. Einfache Datentypen und -strukturen

---

## 2. Einfache Datentypen und -strukturen

- 2.1. Grundtypen
- 2.2. Verbund- und Zeigertypen
- 2.3. Sequenzen
- 2.4. Stacks und Queues**
- 2.5. Bäume

- **Andere Namen: Stapel, Kellerspeicher, LIFO-Liste**  
(LIFO = Last-In First-Out)
- **Anwendungen**
  - Speicherung von Aufrufkontexten (Parameter + Rücksprungadresse)
  - Auswertung von Ausdrücken (Interpreter)
  - Übersetzung von Programmiersprachen (Compiler)
  - ...
- **Außerhalb des EDV-Bereichs:**
  - Tablettstapel in Mensa
  - Papierstapel auf einem Schreibtisch
- **Ähnlich wie Liste, aber Zugriff ist auf erstes Element beschränkt**
  - Kann aufbauend auf den vorgestellten Listentypen implementiert werden

# Stack II

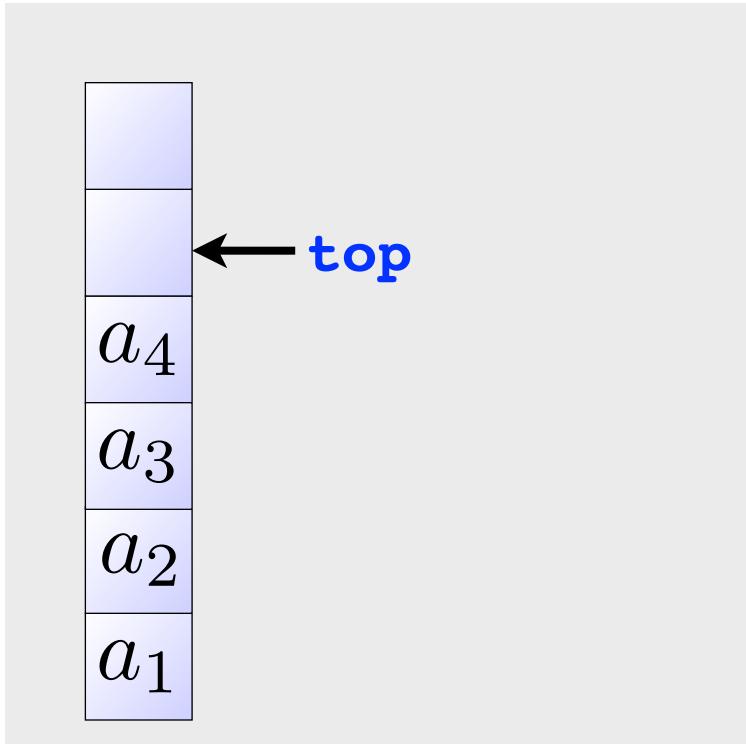
- **Zentrale Operationen**
  - **push**: neues Element auf Stackspitze legen
  - **pop**: oberstes Element vom Stack auslesen und entfernen
  - **isEmpty**: prüfen, ob Stack leer ist
- **top-Zeiger für oberstes Element**

# Stack II

- **Zentrale Operationen**

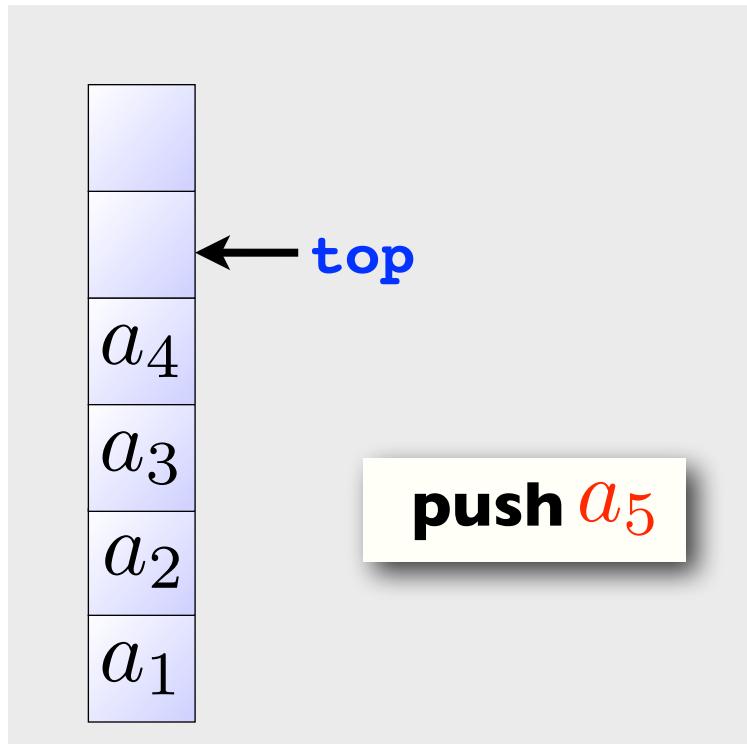
- **push**: neues Element auf Stackspitze legen
- **pop**: oberstes Element vom Stack auslesen und entfernen
- **isEmpty**: prüfen, ob Stack leer ist

- **top-Zeiger für oberstes Element**



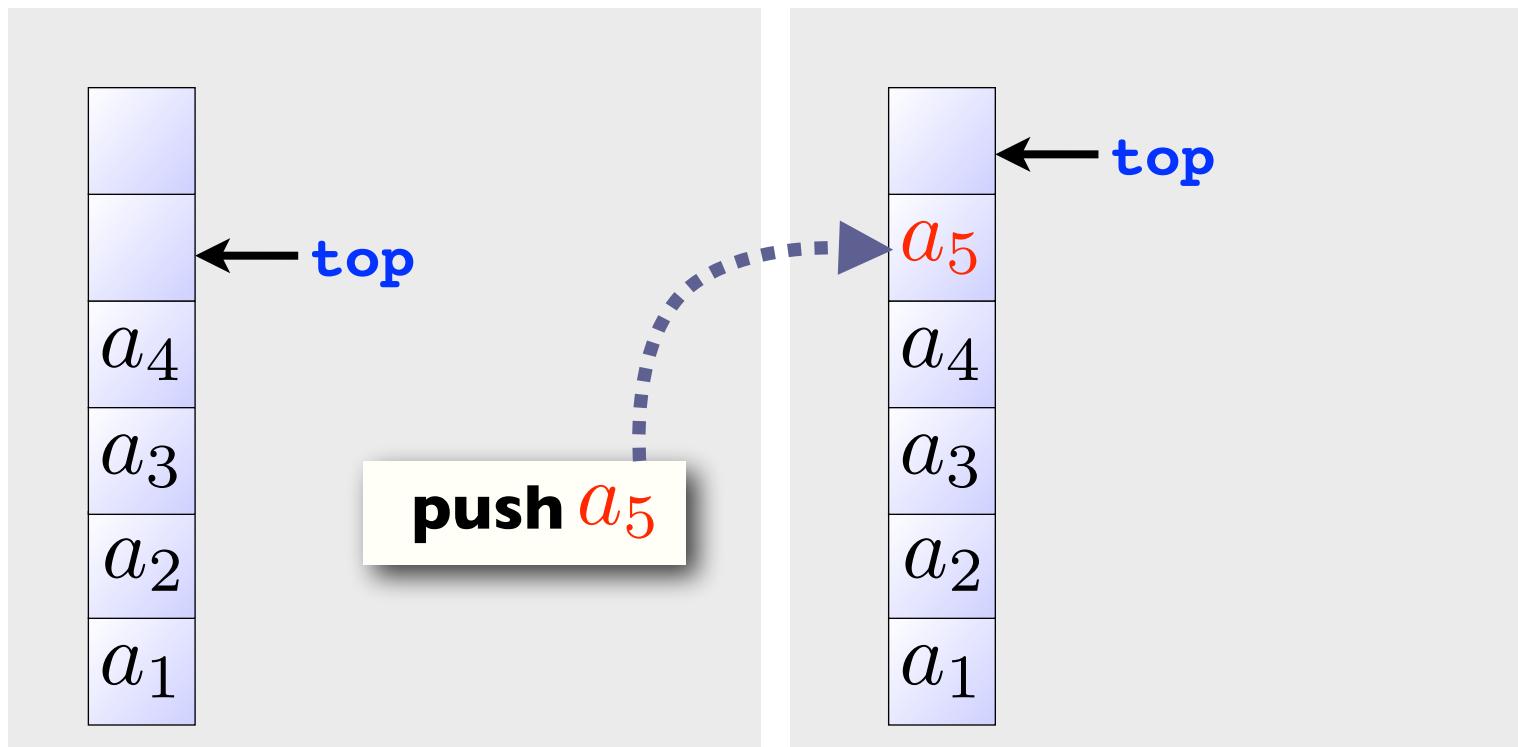
# Stack II

- **Zentrale Operationen**
  - **push**: neues Element auf Stackspitze legen
  - **pop**: oberstes Element vom Stack auslesen und entfernen
  - **isEmpty**: prüfen, ob Stack leer ist
- **top-Zeiger für oberstes Element**



# Stack II

- **Zentrale Operationen**
  - **push**: neues Element auf Stackspitze legen
  - **pop**: oberstes Element vom Stack auslesen und entfernen
  - **isEmpty**: prüfen, ob Stack leer ist
- **top-Zeiger für oberstes Element**

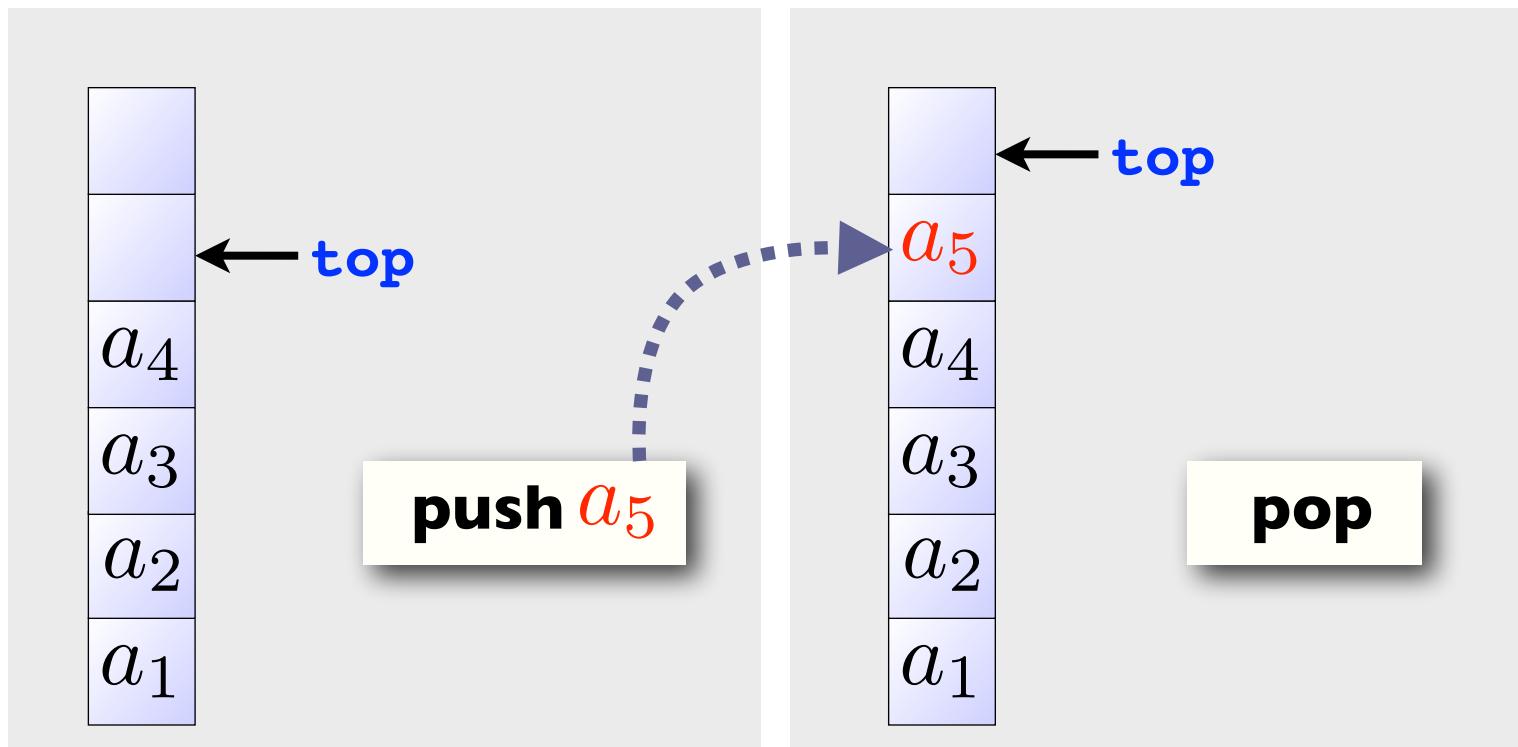


# Stack II

- **Zentrale Operationen**

- **push**: neues Element auf Stackspitze legen
- **pop**: oberstes Element vom Stack auslesen und entfernen
- **isEmpty**: prüfen, ob Stack leer ist

- **top-Zeiger für oberstes Element**

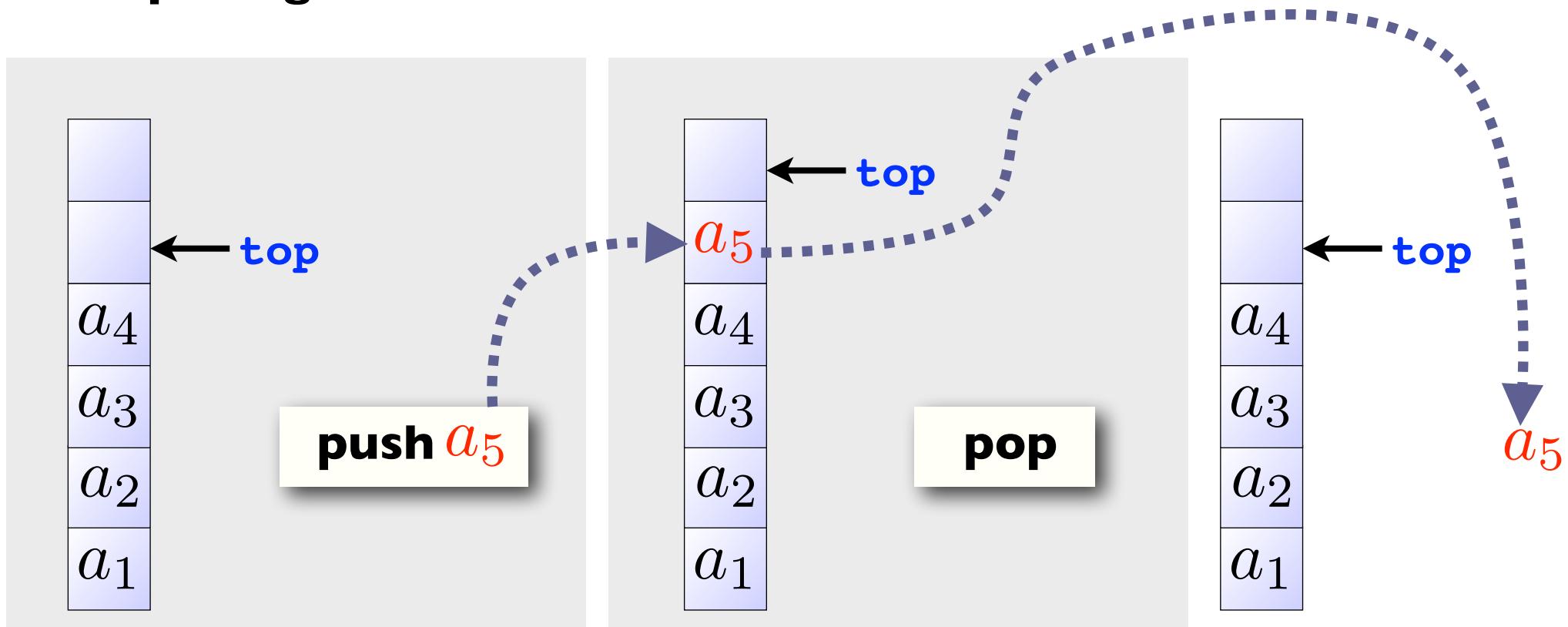


# Stack II

- **Zentrale Operationen**

- **push**: neues Element auf Stackspitze legen
- **pop**: oberstes Element vom Stack auslesen und entfernen
- **isEmpty**: prüfen, ob Stack leer ist

- **top-Zeiger für oberstes Element**



# Stack: Implementierung mit Arrays (JAVA)

# Stack: Implementierung mit Arrays (JAVA)

```
public class MyStackA {  
    int top;  
    Object[] theStack;  
    public MyStackA(int capacity) {  
        theStack =  
            new Object[capacity];  
        top=0;  
    }  
    public void push(Object t) {  
        if (isFull())  
            // ... Fehlerbehandlung  
        theStack[top++]=t;  
    }  
  
    public Object pop() {  
        if (isEmpty())  
            // ... Fehlerbehandlung  
        return theStack[--top];  
    }  
    public boolean isEmpty() {  
        return top==0;  
    }  
    public boolean isFull() {  
        return top >= theStack.length;  
    }  
}
```

# Stack: Implementierung mit Arrays (JAVA)

```
public class MyStackA {  
    int top;  
    Object[] theStack;  
    public MyStackA(int capacity) {  
        theStack =  
            new Object[capacity];  
        top=0;  
    }  
    public void push(Object t) {  
        if (isFull())  
            // ... Fehlerbehandlung  
        theStack[top++]=t;  
    }  
  
    public Object pop() {  
        if (isEmpty())  
            // ... Fehlerbehandlung  
        return theStack[--top];  
    }  
    public boolean isEmpty() {  
        return top==0;  
    }  
    public boolean isFull() {  
        return top >= theStack.length;  
    }  
}
```

## ● Beachte:

- top zeigt immer auf das erste freie Feld - *nicht* auf das oberste Element
- Alle von Object abgeleiteten Klassen können gespeichert werden - typecast nach pop nicht unproblematisch => **Generics**
- **Memory Leak** - sehen Sie wo?

# Stack: Implementierung mit Zeigern (JAVA)

# Stack: Implementierung mit Zeigern (JAVA)

- **Ausgangspunkt: Lineare Liste**

```
public class Node {  
    public Object value;  
    public Node next;  
}
```

# Stack: Implementierung mit Zeigern (JAVA)

- **Ausgangspunkt: Lineare Liste**

```
public class Node {  
    public Object value;  
    public Node next;  
}
```

- **Klasse MyStack:**

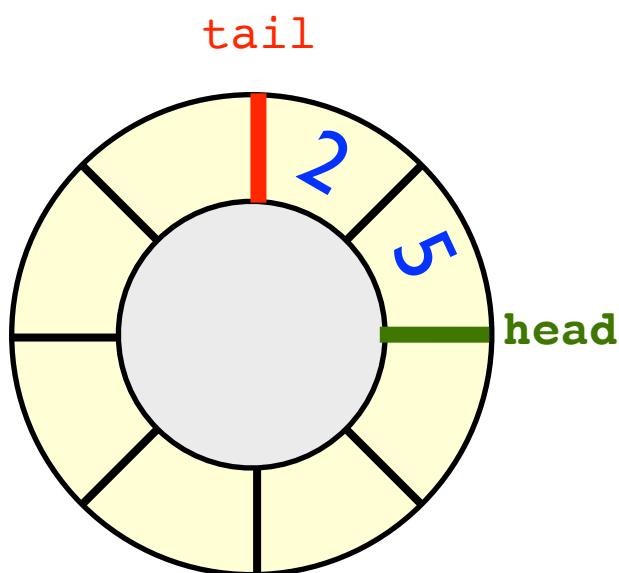
```
public class MyStack {  
    Node top;  
    public MyStack() {  
        top = null;  
    }  
    public boolean isEmpty() {  
        return top==null;  
    }  
    public void push(Object t) {  
        Node x = new Node();  
        x.value=t;  
        x.next=top;  
        top=x;  
    }  
}
```

```
public Object pop() {  
    if (isEmpty()) {  
        // Fehlerbehandlung  
        return null;  
    } else {  
        Node x = top;  
        top=top.next;  
        return x.value;  
    }  
}
```

- **Andere Namen:** **Warteschlange, Ringspeicher, FIFO-Liste**  
(FIFO=First-In First-Out)
- **Anwendungen**
  - Überall dort, wo die Reihenfolge des Eintreffens eine Rolle spielt
  - z.B.: Scheduling von Prozessen in einem Betriebssystem, Tastaturpuffer
- **Anwendungen außerhalb der EDV**
  - Warteschlange an der Essensausgabe
- **Zentrale Operationen**
  - **put / enqueue:** Element am Ende der Liste einfügen
  - **get / dequeue:** Element am Anfang der Liste entfernen

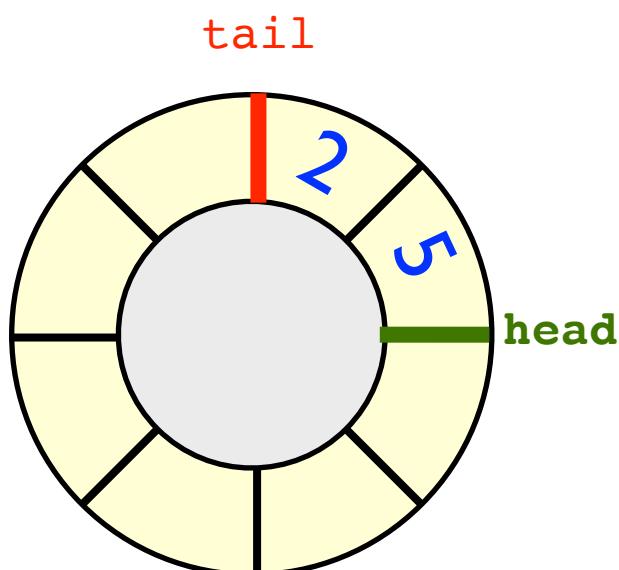
- **Liste mit zwei Enden**

- **tail** Ende der Warteschlange
- **head** Kopf der Warteschlange



- **Liste mit zwei Enden**

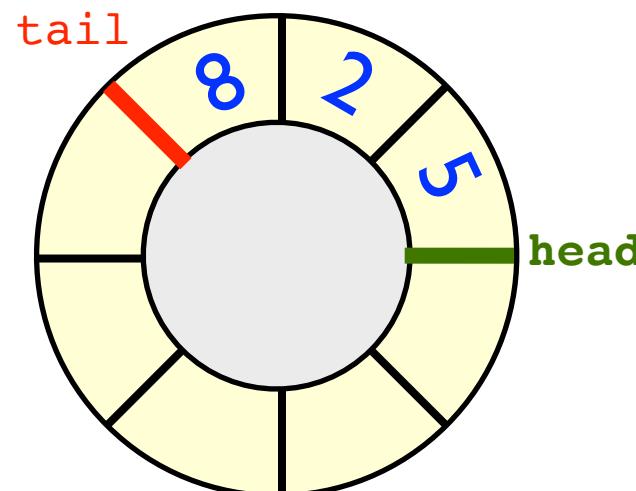
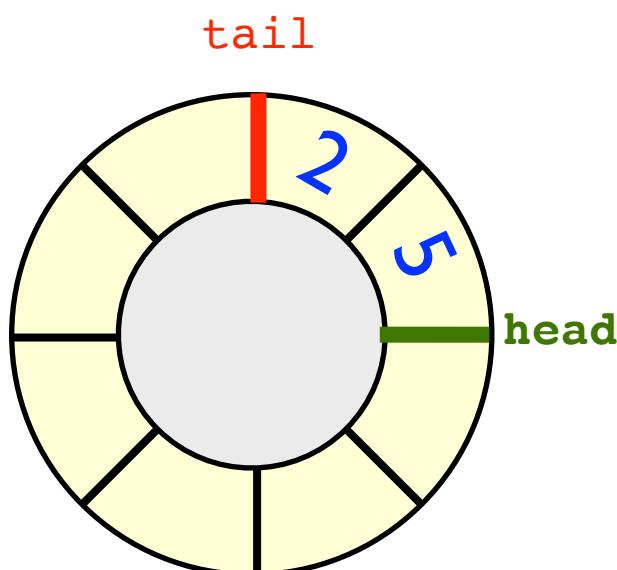
- **tail** Ende der Warteschlange
- **head** Kopf der Warteschlange



**enqueue 8**

- **Liste mit zwei Enden**

- **tail** Ende der Warteschlange
- **head** Kopf der Warteschlange

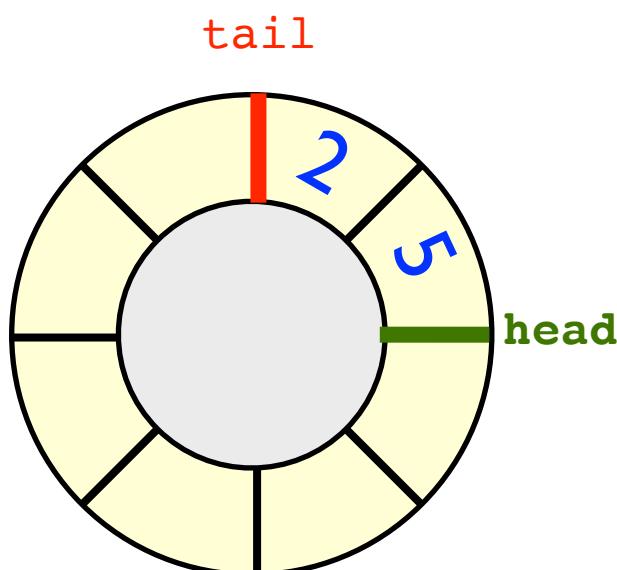


**enqueue 8**

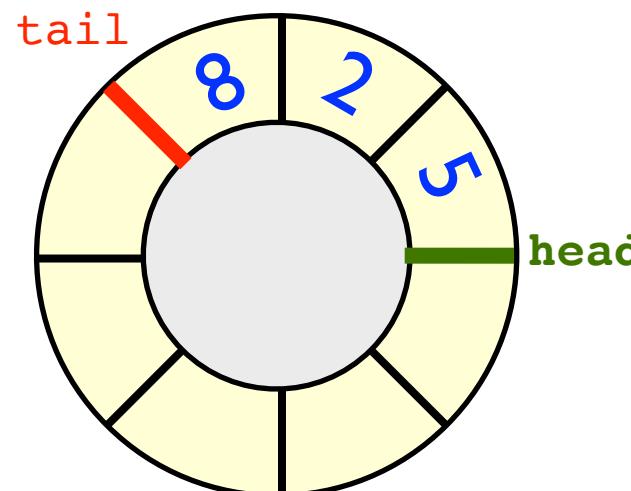
# Queue II

- **Liste mit zwei Enden**

- **tail** Ende der Warteschlange
- **head** Kopf der Warteschlange



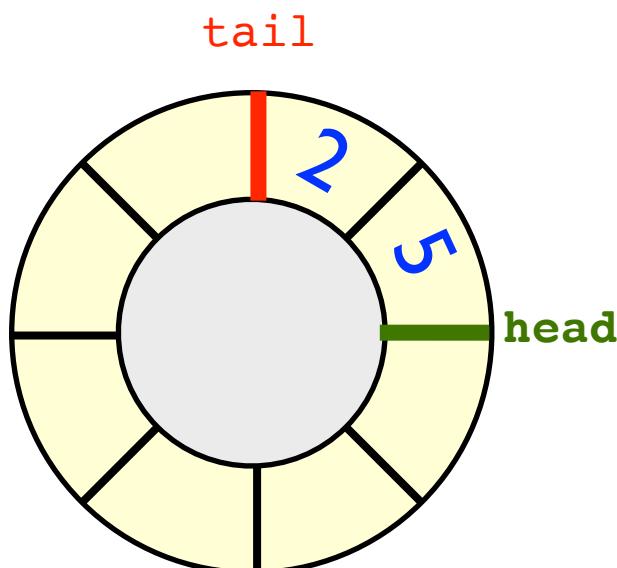
**enqueue 8**



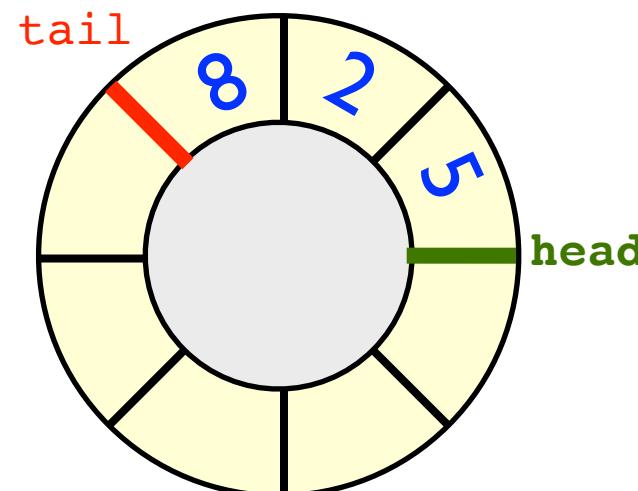
**dequeue**

- **Liste mit zwei Enden**

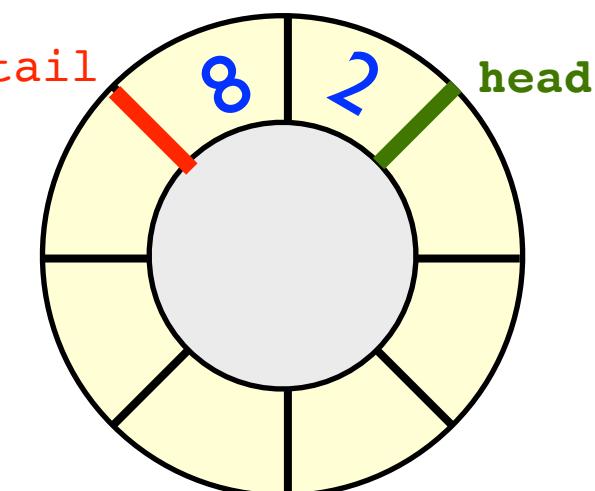
- **tail** Ende der Warteschlange
- **head** Kopf der Warteschlange



**enqueue 8**



**dequeue**



# Queue: Implementierung mit Array (JAVA)

# Queue: Implementierung mit Array (JAVA)

```
public class MyQueue {  
    Object theQueue[];  
    int head, tail;  
  
    public MyQueue(int capacity) {  
        theQueue =  
            new Object[capacity];  
        head=tail=0;  
    }  
    public void enqueue(Object t) {  
        if (isFull())  
            // ... Fehlerbehandlung  
        theQueue[tail]=t;  
        tail = (tail + 1) %  
              theQueue.length;  
    }  
}
```

```
    public Object dequeue() {  
        if (isEmpty())  
            // ... Fehlerbehandlung  
        Object ret = theQueue[head];  
        head=(head + 1) %  
              theQueue.length;  
        return ret;  
    }  
    public boolean isEmpty() {  
        return head==tail;  
    }  
    public boolean isFull() {  
        return head == (tail + 1) %  
              theQueue.length;  
    }  
}
```

# Queue: Implementierung mit Array (JAVA + Generics)

```
import java.util.ArrayList;

public class MyGenericQueue<T> {
    ArrayList<T> theQueue;
    int head, tail, capacity;

    public MyGenericQueue(int capacity) {
        this.capacity = capacity;
        theQueue = new
                    ArrayList<T>(capacity);
        // Alle Einträge mit null
        // initialisieren
        for (int i=0 ; i<capacity ; ++i)
            theQueue.add(null);
        head=tail=0;
    }

    public void enqueue(T t) {
        if (isFull())
            // ... Fehlerbehandlung
        theQueue.set(tail, t);
        tail = (tail + 1) %
              capacity;
    }

    public T dequeue() {
        if (isEmpty())
            // ... Fehlerbehandlung
        T ret = theQueue.get(head);
        head=(head + 1) % capacity;
        return ret;
    }

    public boolean isEmpty() {
        return head==tail;
    }

    public boolean isFull() {
        return head == (tail + 1) %
               capacity;
    }
}
```

Exkurs

# Queue: Implementierung mit Array (JAVA + Generics)

```
import java.util.ArrayList;

public class MyGenericQueue<T> {
    ArrayList<T> theQueue;
    int head, tail, capacity;

    public MyGenericQueue(int capacity) {
        this.capacity = capacity;
        theQueue = new
                    ArrayList<T>(capacity);
        // Alle Einträge mit null
        // initialisieren
        for (int i=0 ; i<capacity ; ++i)
            theQueue.add(null);
        head=tail=0;
    }

    public void enqueue(T t) {
        if (isFull())
            // ... Fehlerbehandlung
        theQueue.set(tail, t);
        tail = (tail + 1) %
              capacity;
    }

    public T dequeue() {
        if (isEmpty())
            // ... Fehlerbehandlung
        T ret = theQueue.get(head);
        head=(head + 1) % capacity;
        return ret;
    }

    public boolean isEmpty() {
        return head==tail;
    }

    public boolean isFull() {
        return head == (tail + 1) %
               capacity;
    }
}
```

Warum nicht **T theQueue[]**?

# Generische Felder in JAVA

- Betrachten wir folgende (**fehlerhafte !**) JAVA-Methode zur Erzeugung eines generischen Feldes

```
public <T> T[ ] makeArray(int size) {  
    // Angenommen, es wäre erlaubt ...  
    T[ ] genericArray = new T[size];  
    return genericArray;  
}
```

# Generische Felder in JAVA

- Betrachten wir folgende (**fehlerhafte !**) JAVA-Methode zur Erzeugung eines generischen Feldes

```
public <T> T[ ] makeArray(int size) {  
    // Angenommen, es wäre erlaubt ...  
    T[ ] genericArray = new T[size];  
    return genericArray;  
}
```

- In JAVA werden Generics durch **Typ-Elimination** übersetzt.

# Generische Felder in JAVA

- Betrachten wir folgende (**fehlerhafte !**) JAVA-Methode zur Erzeugung eines generischen Feldes

```
public <T> T[ ] makeArray(int size) {  
    // Angenommen, es wäre erlaubt ...  
    T[ ] genericArray = new T[size];  
    return genericArray;  
}
```

- In JAVA werden Generics durch **Typ-Elimination** übersetzt.
- Der Compiler erzeugt gewissermaßen folgenden Code für `makeArray`:

```
public <T> Object[ ] makeArray(int size) {  
    Object[ ] genericArray = new Object[size];  
    return genericArray;  
}
```

# Generische Felder in JAVA

- Betrachten wir folgende (**fehlerhafte !**) JAVA-Methode zur Erzeugung eines generischen Feldes

```
public <T> T[ ] makeArray(int size) {  
    // Angenommen, es wäre erlaubt ...  
    T[ ] genericArray = new T[size];  
    return genericArray;  
}
```

- In JAVA werden Generics durch **Typ-Elimination** übersetzt.
- Der Compiler erzeugt gewissermaßen folgenden Code für **makeArray**:

```
public <T> Object[ ] makeArray(int size) {  
    Object[ ] genericArray = new Object[size];  
    return genericArray;  
}
```

- Egal mit welchem Typ **makeArray** aufgerufen wird, es wird immer derselbe Code auf der JVM ausgeführt wird!

# Generische Felder in JAVA

- Setzen wir unsere Methode ein, um ein String-Feld zu erzeugen: Array:

```
String[ ] myArray = makeArray(5);
```

Exkurs

# Generische Felder in JAVA

- Setzen wir unsere Methode ein, um ein String-Feld zu erzeugen: Array:

```
String[ ] myArray = newArray(5);
```

- Dieser Code übersetzt einwandfrei, allerdings kommt es zu einem Laufzeitfehler (**ClassCastException**), weil wir ein **Object[ ]** einem **String[ ]**-Array zuweisen, was nicht erlaubt ist.

Exkurs

# Generische Felder in JAVA

- Setzen wir unsere Methode ein, um ein String-Feld zu erzeugen: Array:

```
String[ ] myArray = newArray(5);
```

- Dieser Code übersetzt einwandfrei, allerdings kommt es zu einem Laufzeitfehler (**ClassCastException**), weil wir ein **Object[ ]** einem **String[ ]**-Array zuweisen, was nicht erlaubt ist.
- Das Problem ließe sich mit **Reflection** lösen, allerdings ändert sich dann die Schnittstelle von **makeArray**:

```
public T[ ] makeStack(Class<E> clazz,  
                      int capacity) {  
    return (T[ ])Array.newInstance(clazz, capacity);  
}
```

Exkurs

# Listen / Queues in C++ und JAVA

- **C++ stellt mit der Standard-Template-Library (STL) eine Queue und einen Stack zur Verfügung**
  - `deque<T, alloc>` double-ended queue
  - `stack<T, sequence>` sequence ist der Typ eines Sequenzcontainers (z.B. `vector` oder `deque` - default)

# Listen / Queues in C++ und JAVA

- **C++ stellt mit der Standard-Template-Library (STL) eine Queue und einen Stack zur Verfügung**

- `deque<T, alloc>` double-ended queue
- `stack<T, sequence>` sequence ist der Typ eines Sequenzcontainers (z.B. `vector` oder `deque` - default)

```
template <class T, class Container = deque<T>> class stack;
```

# Listen / Queues in C++ und JAVA

- **C++ stellt mit der Standard-Template-Library (STL) eine Queue und einen Stack zur Verfügung**

- `deque<T, alloc>` double-ended queue
- `stack<T, sequence>` sequence ist der Typ eines Sequenzcontainers (z.B. `vector` oder `deque` - default)

```
template <class T, class Container = deque<T>> class stack;
```

- **Auch in JAVA gibt es vordefinierte Datentypen Queue und Stack:**

- `java.util.Queue<E>`
- `java.util.Stack<E>`

# **II. Einfache Datentypen und -strukturen**

---

## **2. Einfache Datentypen und -strukturen**

- 2.1. Grundtypen**
- 2.2. Verbund- und Zeigertypen**
- 2.3. Sequenzen**
- 2.4. Stacks und Queues**
- 2.5. Bäume**

# Bäume - Definition und Begriffe I

## D Baum

Ein **Baum**  $B = (V, E)$  besteht aus einer endlichen Menge von Knoten  $V$  und einer Menge  $E \subseteq V \times V$  von geordneten Paaren mit folgenden Eigenschaften:

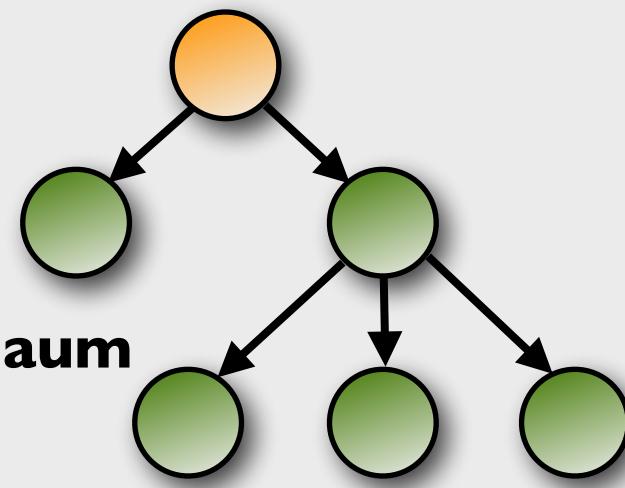
- $\exists v_r \in V, \forall v \in V : (v, v_r) \notin E$  - es gibt genau einen Knoten, der keinen Vorgänger hat - den **Wurzelknoten**  $v_r$ .
- $\forall v_c \in V \setminus \{v_r\}, \exists v_f \in V : (v_f, v_c) \in E$  und  $\nexists v \in V \setminus \{v_f\} : (v, v_c) \in E$  - abgesehen vom Wurzelknoten hat jeder Knoten  $v_c$  einen eindeutigen Vorgänger  $v_f$ .

# Bäume - Definition und Begriffe I

## D Baum

Ein **Baum**  $B = (V, E)$  besteht aus einer endlichen Menge von Knoten  $V$  und einer Menge  $E \subseteq V \times V$  von geordneten Paaren mit folgenden Eigenschaften:

- $\exists v_r \in V, \forall v \in V : (v, v_r) \notin E$  - es gibt genau einen Knoten, der keinen Vorgänger hat - den **Wurzelknoten**  $v_r$ .
- $\forall v_c \in V \setminus \{v_r\}, \exists v_f \in V : (v_f, v_c) \in E$  und  $\nexists v \in V \setminus \{v_f\} : (v, v_c) \in E$  - abgesehen vom Wurzelknoten hat jeder Knoten  $v_c$  einen eindeutigen Vorgänger  $v_f$ .

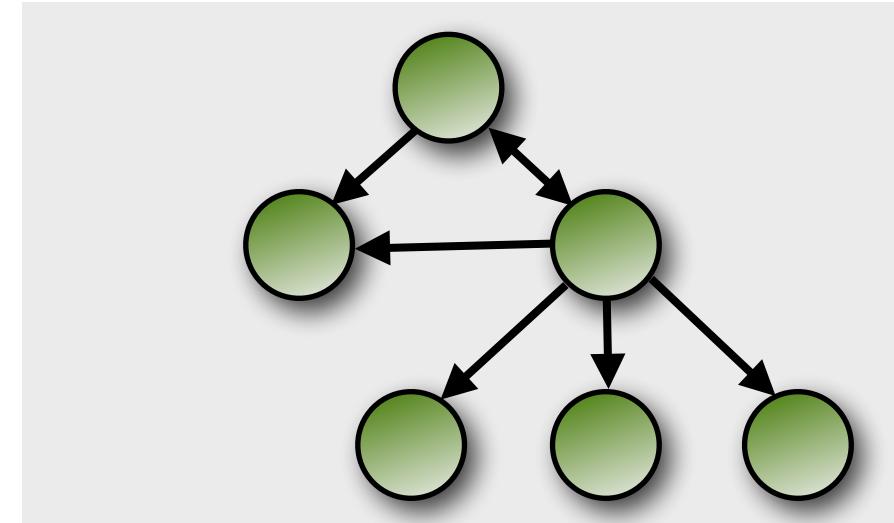
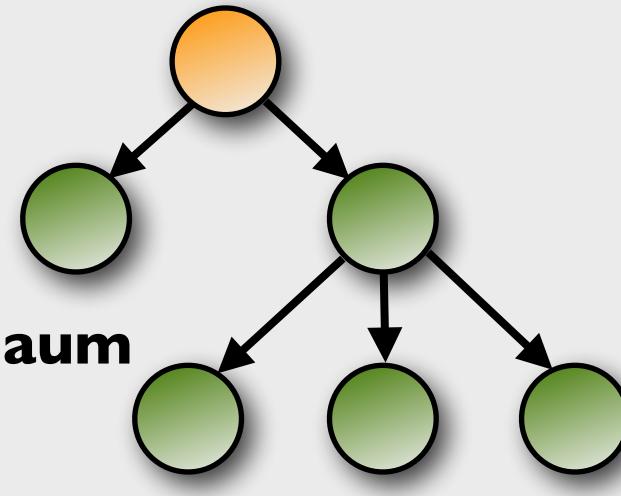


# Bäume - Definition und Begriffe I

## D Baum

Ein **Baum**  $B = (V, E)$  besteht aus einer endlichen Menge von Knoten  $V$  und einer Menge  $E \subseteq V \times V$  von geordneten Paaren mit folgenden Eigenschaften:

- $\exists v_r \in V, \forall v \in V : (v, v_r) \notin E$  - es gibt genau einen Knoten, der keinen Vorgänger hat - den **Wurzelknoten**  $v_r$ .
- $\forall v_c \in V \setminus \{v_r\}, \exists v_f \in V : (v_f, v_c) \in E$  und  $\nexists v \in V \setminus \{v_f\} : (v, v_c) \in E$  - abgesehen vom Wurzelknoten hat jeder Knoten  $v_c$  einen eindeutigen Vorgänger  $v_f$ .

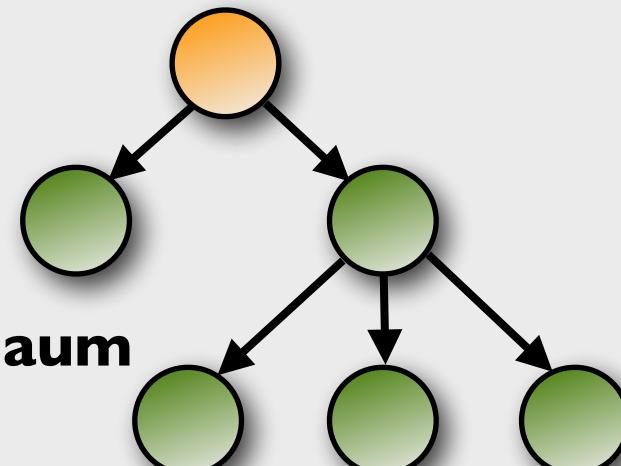


# Bäume - Definition und Begriffe I

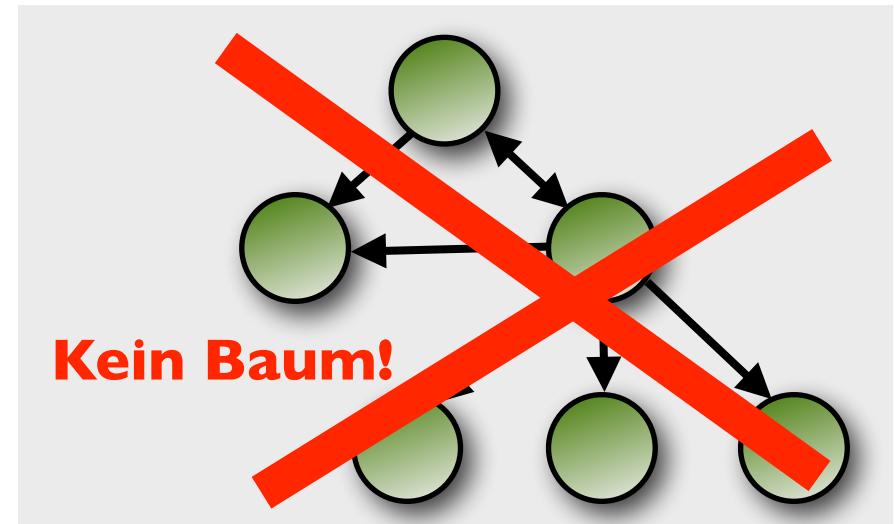
## D Baum

Ein **Baum**  $B = (V, E)$  besteht aus einer endlichen Menge von Knoten  $V$  und einer Menge  $E \subseteq V \times V$  von geordneten Paaren mit folgenden Eigenschaften:

- $\exists v_r \in V, \forall v \in V : (v, v_r) \notin E$  - es gibt genau einen Knoten, der keinen Vorgänger hat - den **Wurzelknoten**  $v_r$ .
- $\forall v_c \in V \setminus \{v_r\}, \exists v_f \in V : (v_f, v_c) \in E$  und  $\nexists v \in V \setminus \{v_f\} : (v, v_c) \in E$  - abgesehen vom Wurzelknoten hat jeder Knoten  $v_c$  einen eindeutigen Vorgänger  $v_f$ .



**Ein Baum**



**Kein Baum!**

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger
- Ein **Blatt** (Blattknoten, *leaf*) ist ein Knoten ohne Nachfolger (also mit Grad 0)

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger
- Ein **Blatt** (Blattknoten, *leaf*) ist ein Knoten ohne Nachfolger (also mit Grad 0)
- **Geschwister** (Brüder, *siblings*) sind alle unmittelbare Nachfolger desselben Vaterknotens

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger
- Ein **Blatt** (Blattknoten, *leaf*) ist ein Knoten ohne Nachfolger (also mit Grad 0)
- **Geschwister** (Brüder, *siblings*) sind alle unmittelbare Nachfolger desselben Vaterknotens
- Ein **Pfad** (Weg) ist eine Folge von Knoten  $v_1, \dots, v_k$ , wobei  $v_i$  unmittelbarer Nachfolger von  $v_{i-1}$  ist. Die **Länge eines Pfades** entspricht der Zahl seiner Kanten (bzw. Zahl der Knoten - 1)

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger
- Ein **Blatt** (Blattknoten, *leaf*) ist ein Knoten ohne Nachfolger (also mit Grad 0)
- **Geschwister** (Brüder, *siblings*) sind alle unmittelbare Nachfolger desselben Vaterknotens
- Ein **Pfad** (Weg) ist eine Folge von Knoten  $v_1, \dots, v_k$ , wobei  $v_i$  unmittelbarer Nachfolger von  $v_{i-1}$  ist. Die **Länge eines Pfades** entspricht der Zahl seiner Kanten (bzw. Zahl der Knoten - 1)
- Die **Tiefe** (oder Höhe) eines Knotens  $v$  entspricht der Länge des Pfades vom Wurzelknoten  $v_r$  nach  $v$ .

# Bäume - Definitionen und Begriffe II

- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger
- Ein **Blatt** (Blattknoten, *leaf*) ist ein Knoten ohne Nachfolger (also mit Grad 0)
- **Geschwister** (Brüder, *siblings*) sind alle unmittelbare Nachfolger desselben Vaterknotens
- Ein **Pfad** (Weg) ist eine Folge von Knoten  $v_1, \dots, v_k$ , wobei  $v_i$  unmittelbarer Nachfolger von  $v_{i-1}$  ist. Die **Länge eines Pfades** entspricht der Zahl seiner Kanten (bzw. Zahl der Knoten - 1)
- Die **Tiefe** (oder Höhe) eines Knotens  $v$  entspricht der Länge des Pfades vom Wurzelknoten  $v_r$  nach  $v$ .
- Die **Höhe eines Baumes** entspricht der Länge des Pfades von der Wurzel zum tiefsten Blatt

# Bäume - Definitionen und Begriffe II

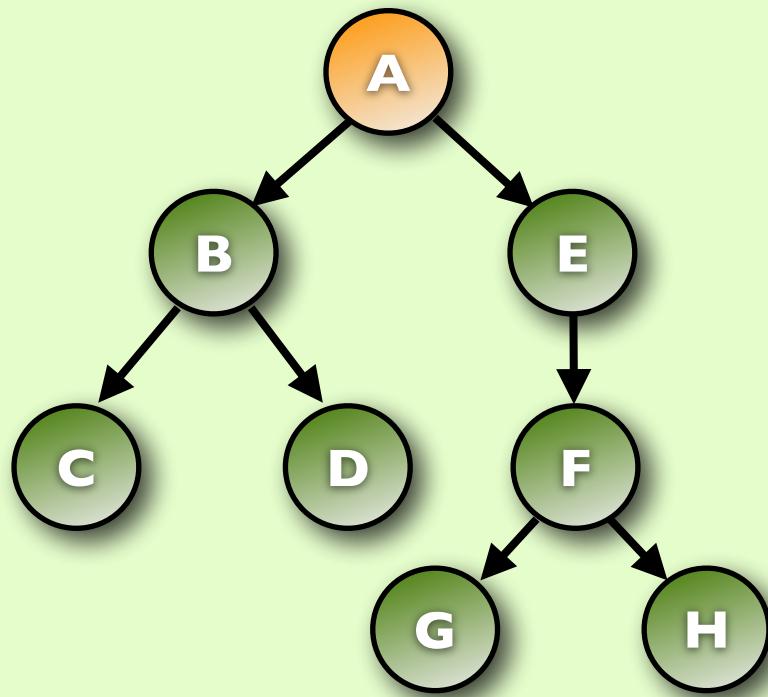
- Jeder Knoten (ausgenommen dem Wurzelknoten) hat einen **Vorgänger** (Vater, *father, parent*)
- Eine **Kante** (Zweig, Ast, *branch*) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus
- Der **Grad** eines Knotens entspricht der Zahl seiner unmittelbaren Nachfolger
- Ein **Blatt** (Blattknoten, *leaf*) ist ein Knoten ohne Nachfolger (also mit Grad 0)
- **Geschwister** (Brüder, *siblings*) sind alle unmittelbare Nachfolger desselben Vaterknotens
- Ein **Pfad** (Weg) ist eine Folge von Knoten  $v_1, \dots, v_k$ , wobei  $v_i$  unmittelbarer Nachfolger von  $v_{i-1}$  ist. Die **Länge eines Pfades** entspricht der Zahl seiner Kanten (bzw. Zahl der Knoten - 1)
- Die **Tiefe** (oder Höhe) eines Knotens  $v$  entspricht der Länge des Pfades vom Wurzelknoten  $v_r$  nach  $v$ .
- Die **Höhe eines Baumes** entspricht der Länge des Pfades von der Wurzel zum tiefsten Blatt
- Der **Grad eines Baumes** ergibt sich aus dem Maximum der Grade seiner Knoten (Ein Baum mit Grad=2 heißt **Binärbaum**)

# Bedeutung von Bäumen

- **Das wichtigste Merkmal von Baumstrukturen ist ihre hierarchische Struktur.**
- **Bäume spielen daher überall dort eine Rolle, wo hierarchische Ordnungsprinzipien vorkommen, z.B.**
  - als Suchbäume zum Suchen in geordneten Mengen;
  - als Entscheidungsbäume zur systematischen Auswahl von Alternativen;
  - zur Repräsentation der Syntax von Programmen oder der Struktur von Dokumenten;
  - als Heap-Struktur in Sortierprozessen, Warteschlangen, ...
  - als Datenstruktur für die effiziente Organisation hierarchischer Dateisysteme.

# Beispiel

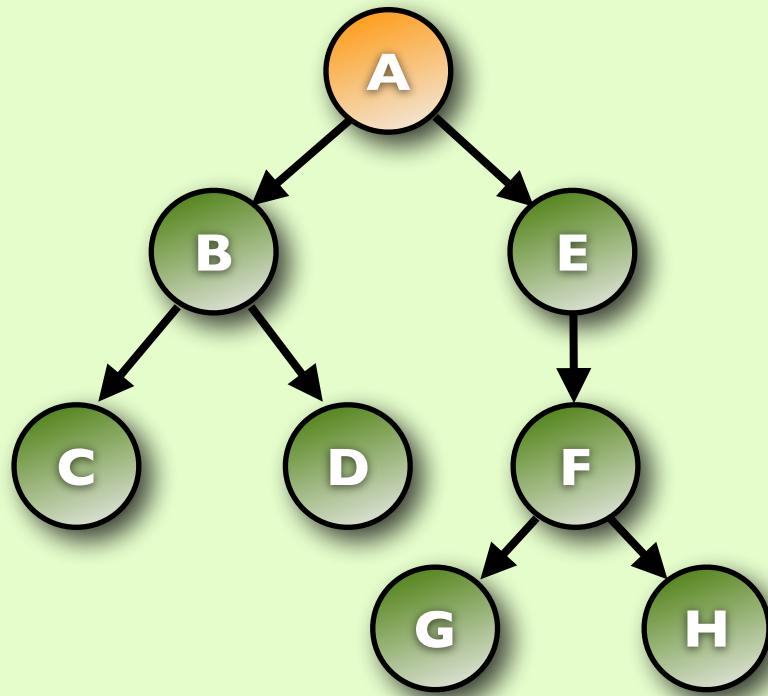
## B Ein Beispielbaum



- A ist **Wurzelknoten**

# Beispiel

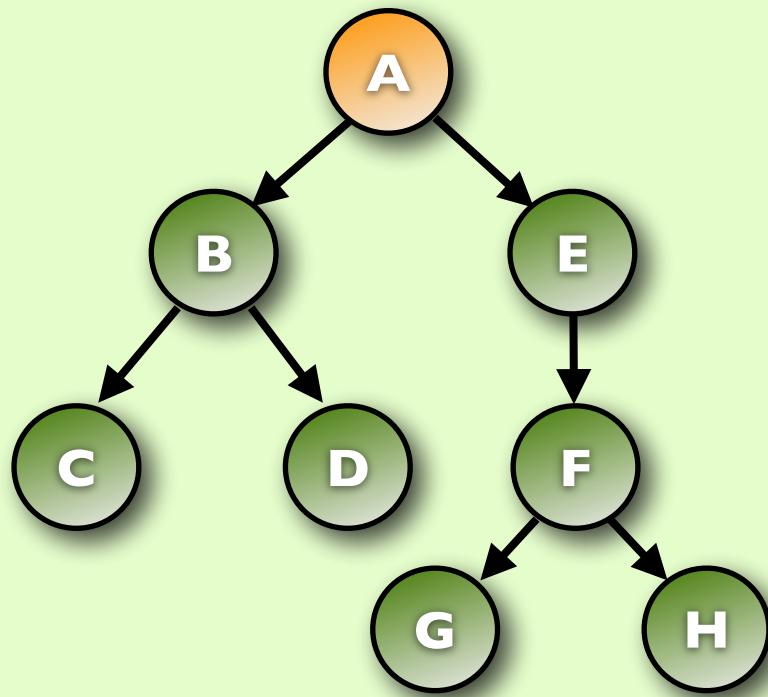
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E

# Beispiel

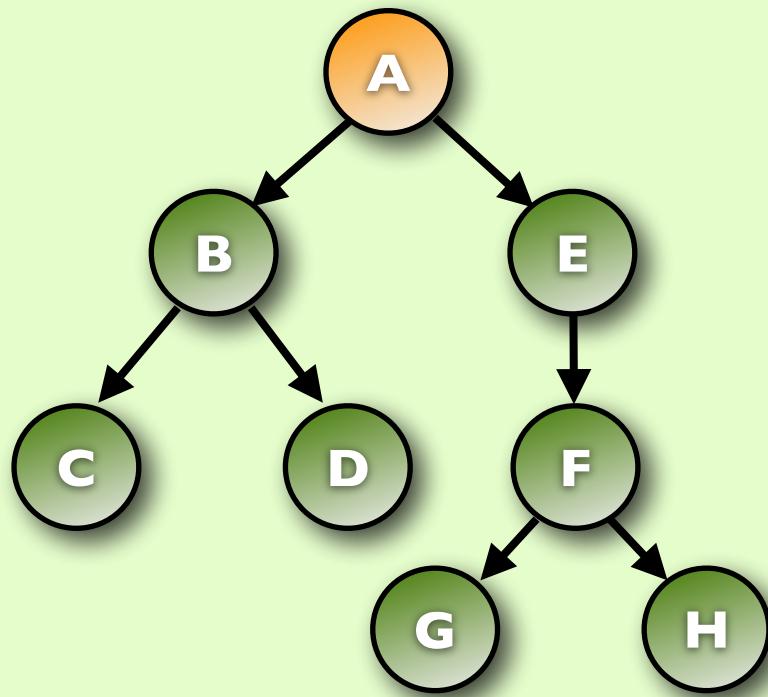
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E
- B und E sind **Brüder**

# Beispiel

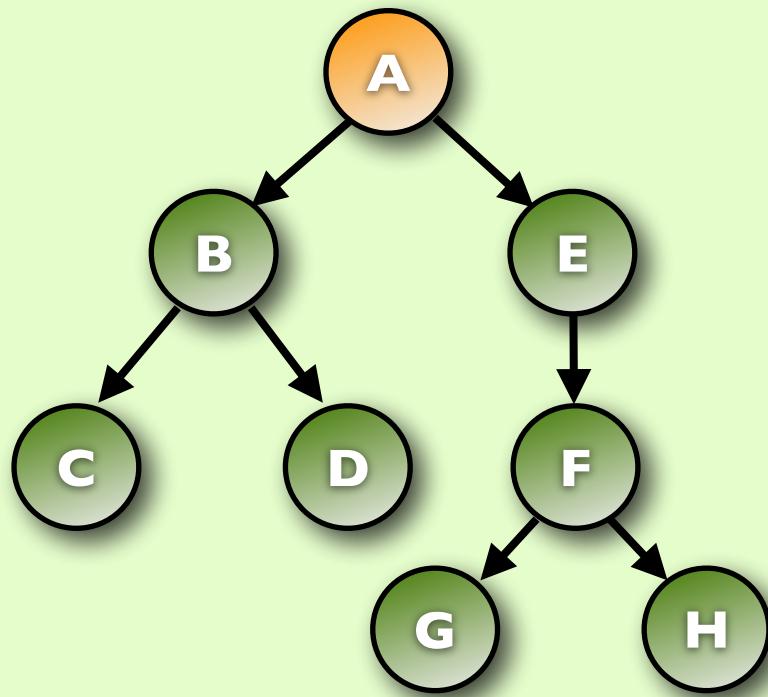
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E
- B und E sind **Brüder**
- C,D,G und H sind **Blätter**

# Beispiel

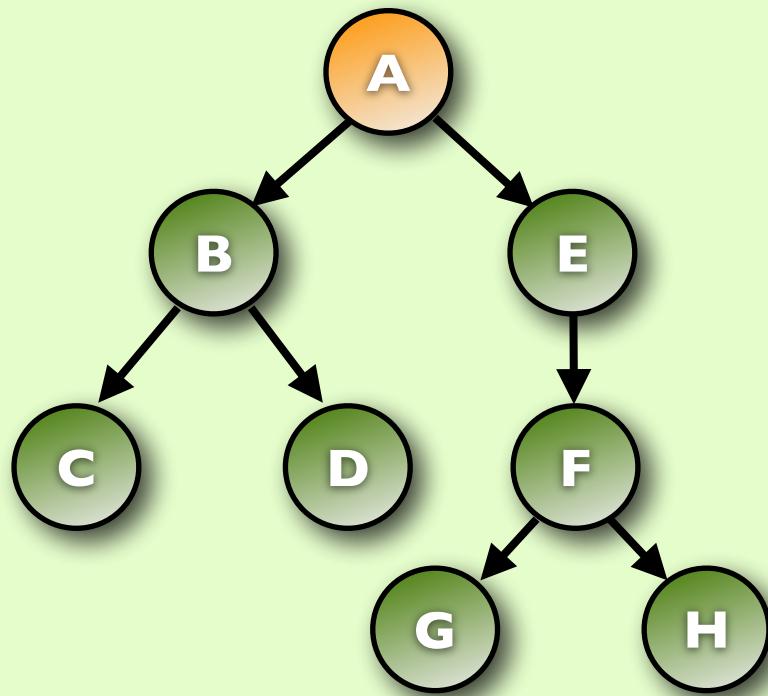
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E
- B und E sind **Brüder**
- C,D,G und H sind **Blätter**
- F hat die **Tiefe 2**

# Beispiel

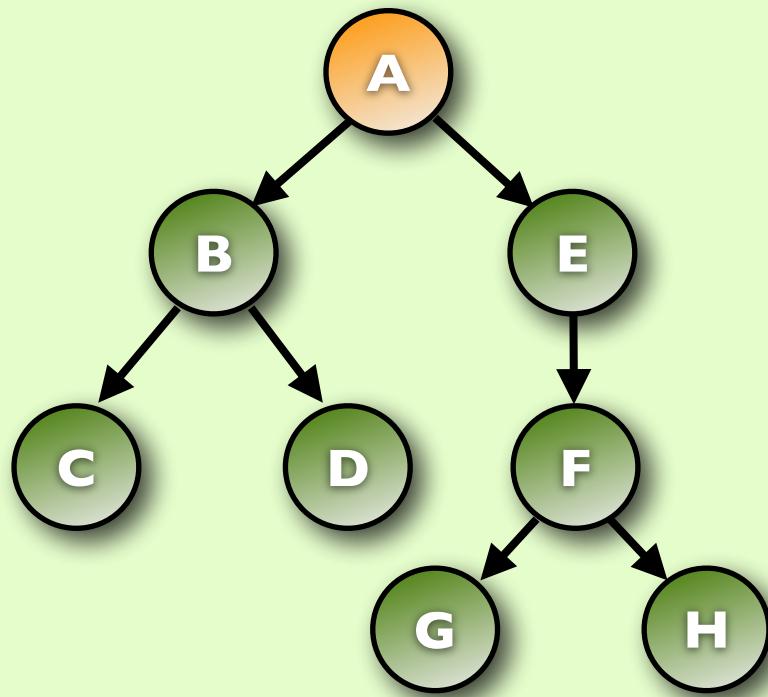
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E
- B und E sind **Brüder**
- C,D,G und H sind **Blätter**
- F hat die **Tiefe** 2
- Die **Höhe des Baumes** ist 3

# Beispiel

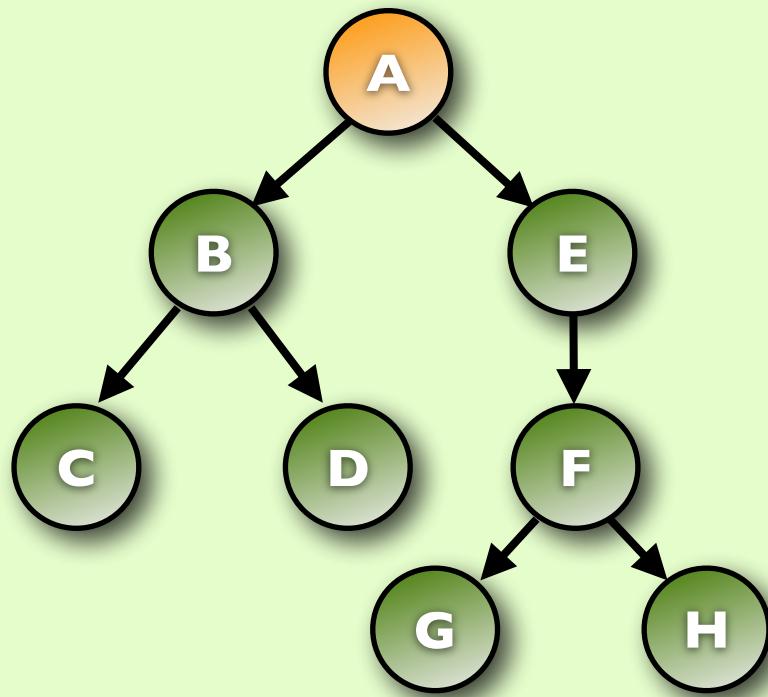
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E
- B und E sind **Brüder**
- C,D,G und H sind **Blätter**
- F hat die **Tiefe 2**
- Die **Höhe des Baumes** ist 3
- Der **Grad des Baumes** ist 2

# Beispiel

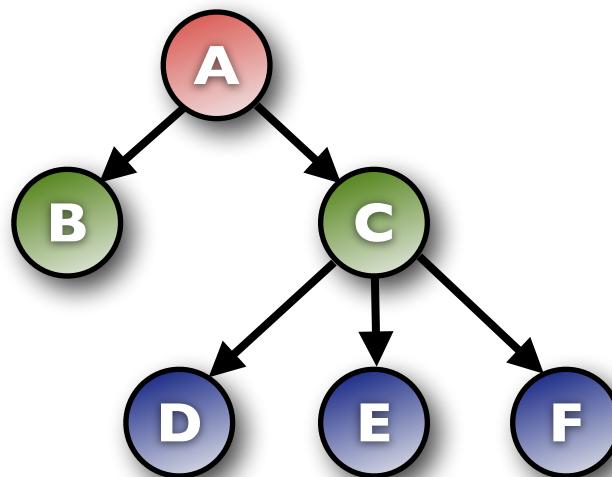
## B Ein Beispielbaum



- A ist **Wurzelknoten**
- A ist **Vater** von B und E
- B und E sind **Brüder**
- C,D,G und H sind **Blätter**
- F hat die **Tiefe 2**
- Die **Höhe des Baumes** ist 3
- Der **Grad des Baumes** ist 2
- (A,E,F,G) ist ein **Pfad**

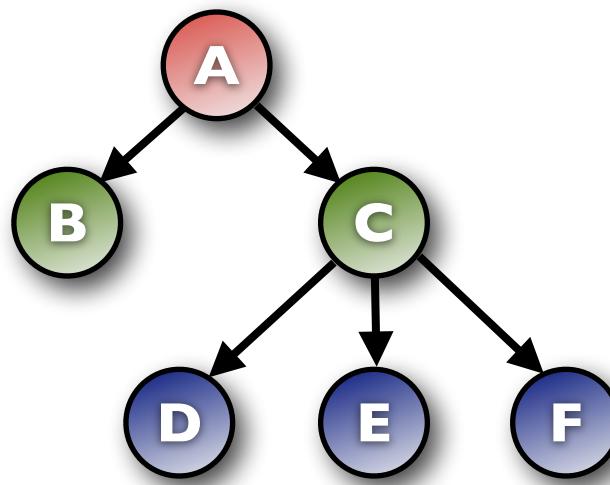
# Darstellung von Bäumen I (Grafische Darstellungen)

- **Graph**

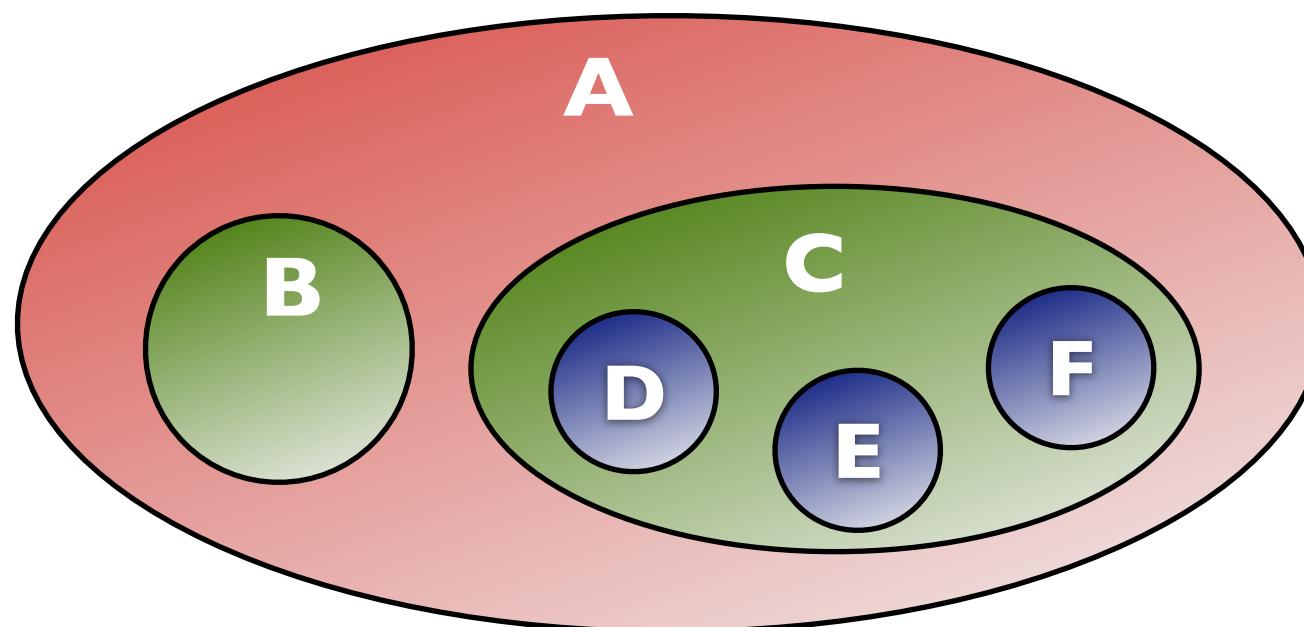


# Darstellung von Bäumen I (Grafische Darstellungen)

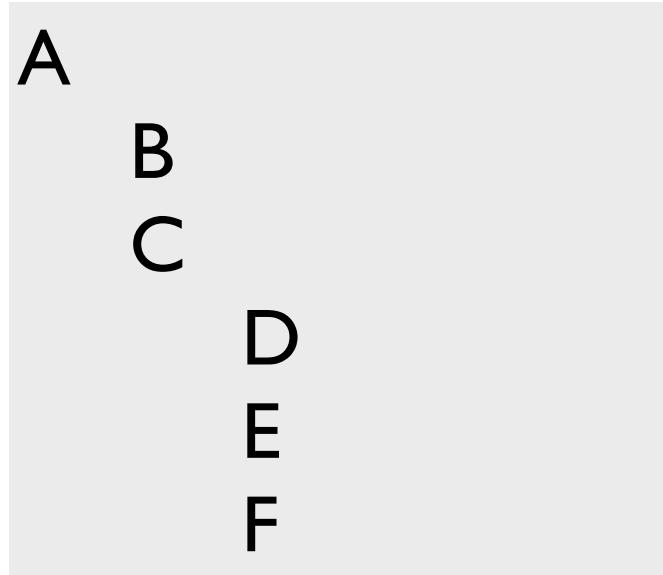
- **Graph**



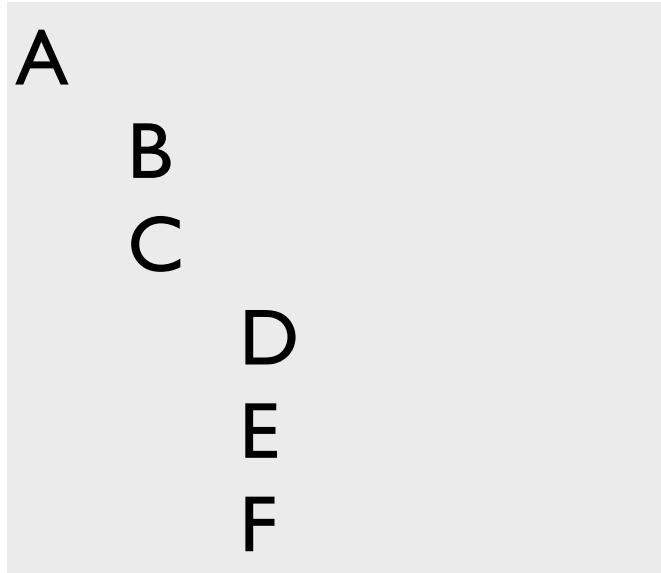
- **Geschachtelte Menge**



- **Einrückung**



- **Einrückung**



- **Geschachtelte Klammerung**

A(B, C(D,E,F) )

# Induktive Definition von Bäumen

- **Mit einer induktiven Definition lassen sich leicht Beweise mittels vollständiger Induktion konstruieren:**

**D Baum (induktiv, schematisch)**

# Induktive Definition von Bäumen

- **Mit einer induktiven Definition lassen sich leicht Beweise mittels vollständiger Induktion konstruieren:**

## D Baum (induktiv, schematisch)

1. Die leere Menge ist ein **Baum**.
2. Ein einzelner Knoten ist ein **Baum**. Dieser Knoten ist gleichzeitig der Wurzelknoten.
3. Sei  $v$  ein Knoten und seien  $B_1, B_2, \dots, B_k$  Bäume mit den zugehörigen Wurzelknoten  $v_1, \dots, v_k$ . Dann wird ein neuer **Baum** konstruiert, indem  $v_1, \dots, v_k$  zu unmittelbaren Nachfolgern von  $v$  gemacht werden.  $B_k$  heisst dann  $k$ -ter **Unterbaum** des Knotens  $v$ .

# Höhe eines Baumes I

- **Folgende Eigenschaft hilft später bei der Abschätzung von Laufzeiten:**

## S Maximale und minimale Höhe eines Baumes

Sei  $B$  ein Baum von maximalem Grad  $d$  und habe  $n$  Knoten dann gilt:

- die maximale Baumhöhe ist  $n - 1$
- die minimale Baumhöhe ist  $\lceil \log_d(n \cdot (d - 1) + 1) \rceil - 1 \leq \lceil \log_d n \rceil$

### Beachte:

- Normalerweise ist Grad eines Baumes fix
- Grad kann sich durch Einfügen / Löschen von Knoten ändern
- Hier geht es um den Effekt dieser Operationen
- Mathematisch korrekt: maximale Baumhöhe ist (falls  $n \geq d$ )  $|n-d|$

### Beweis (Tafel - nicht mitschreiben)

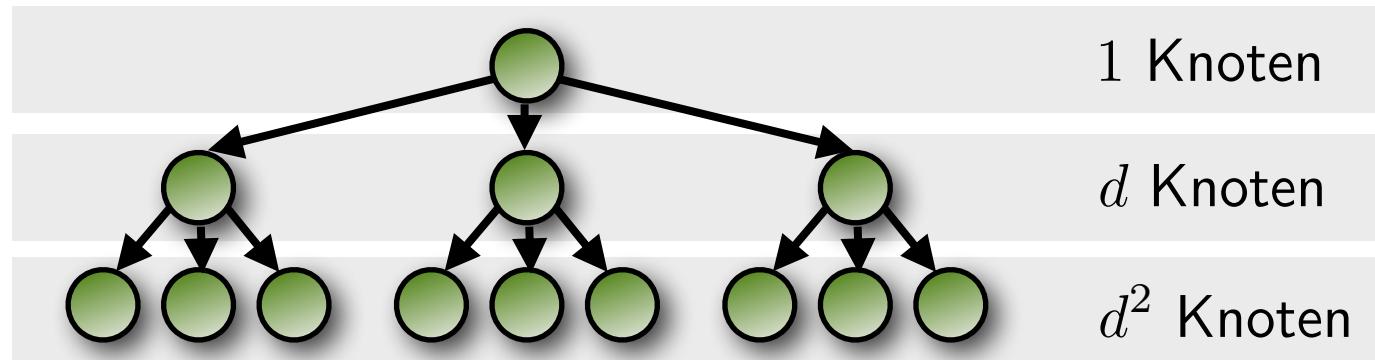
**Maximale Höhe:** Klar,  $B$  besteht aus  $n$  Knoten und somit kann es maximal  $n - 1$  Kanten geben. Man sagt, der Baum ist zu einer linearen Liste entartet.

... Fortsetzung auf nächster Folie

# Höhe eines Baumes II

## Beweis (Fortsetzung ....)

**Minimale Höhe:** Ein maximaler Baum ist ein Baum, der bei gegebener Höhe  $h$  und gegebenem Grad  $d$  die maximale Knotenzahl  $N(h)$  erreicht:



Also gilt für die maximale Knotenzahl  $N(h)$ :

$$\begin{aligned} N(h) &= 1 + d + d^2 + d^3 + \cdots + d^h \\ &= \frac{d^{h+1} - 1}{d - 1} \end{aligned}$$

$$= \sum_{i=0}^h d^i \text{ (Geometrische Reihe)}$$

# Höhe eines Baumes III

## Beweis (Fortsetzung ....)

Bei fester Knotenzahl  $n$  hat ein maximaler gefüllter Baum minimale Höhe.

Ist ein Baum bis auf die letzte (die unterste) Ebene komplett gefüllt, hat er offenbar minimale Höhe.

⇒ Die Anzahl  $n$  der Knoten, die man in einem Baum der Höhe  $h$  unterbringen kann, liegt also zwischen  $N(h)$  und  $N(h - 1)$ :

$$\begin{aligned} N(h-1) &< n &\leq N(h) \\ \frac{d^h-1}{d-1} &< n &\leq \frac{d^{h+1}-1}{d-1} \\ d^h &< n \cdot (d-1) + 1 &\leq d^{h+1} \\ h &< \log_d(n \cdot (d-1) + 1) &\leq h+1 \end{aligned}$$

Offenbar gilt:  $\forall n > 1 : n(d-1) + 1 < nd$ ; somit folgt:

$$\begin{aligned} h &= \lceil \log_d(n \cdot (d-1) + 1) \rceil - 1 \\ &\leq \lceil \log_d(n \cdot d) \rceil - 1 &| \quad \log_d(n \cdot d) = \log_d(n) + \log_d(d) \\ &\leq \lceil \log_d(n) + \log_d(d) \rceil - 1 &| \quad \log_d(d) = 1 \\ &\leq \lceil \log_d n \rceil \end{aligned}$$

# Implementierung von Binärbäumen

- **Wir betrachten zwei geläufige Implementierungen**
  - Die Implementierung mit Zeigern
  - Die Einbettung eines Baumes in ein Array

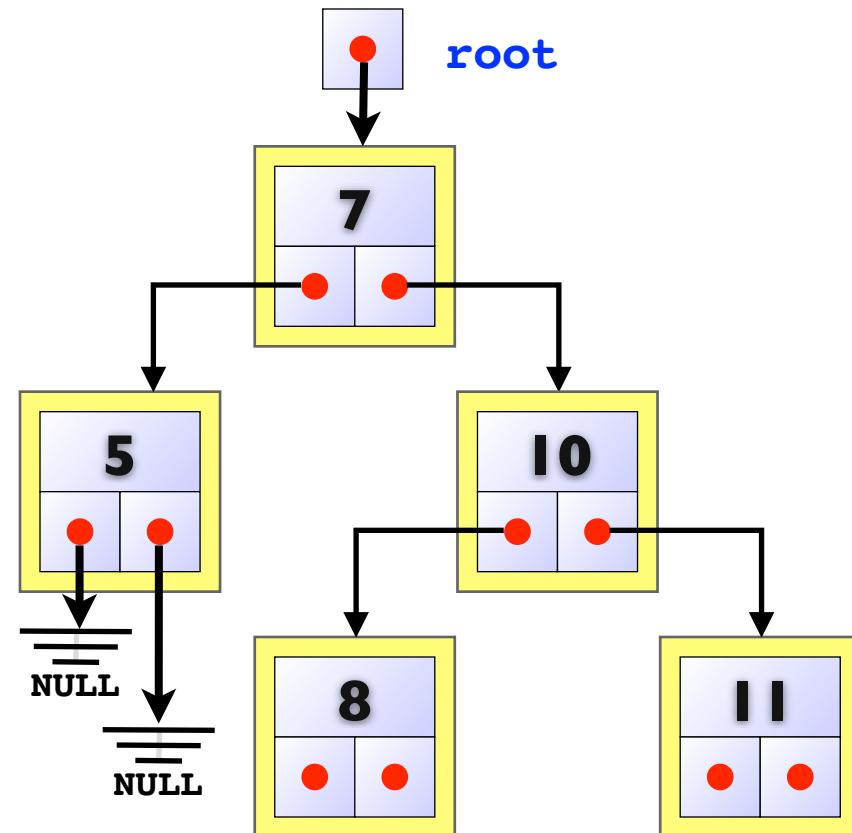
# Binärbaum: Implementierung mit Zeigern (C++)

```
struct treeNode {  
    elemType key;          // Wert des Knotens  
    treeNode *left;        // Zeiger auf linken Sohn  
    treeNode *right;       // Zeiger auf rechten Sohn  
}
```

# Binärbaum: Implementierung mit Zeigern (C++)

```
struct treeNode {  
    elemType key;          // Wert des Knotens  
    treeNode *left;        // Zeiger auf linken Sohn  
    treeNode *right;       // Zeiger auf rechten Sohn  
}
```

## Schematische Darstellung:



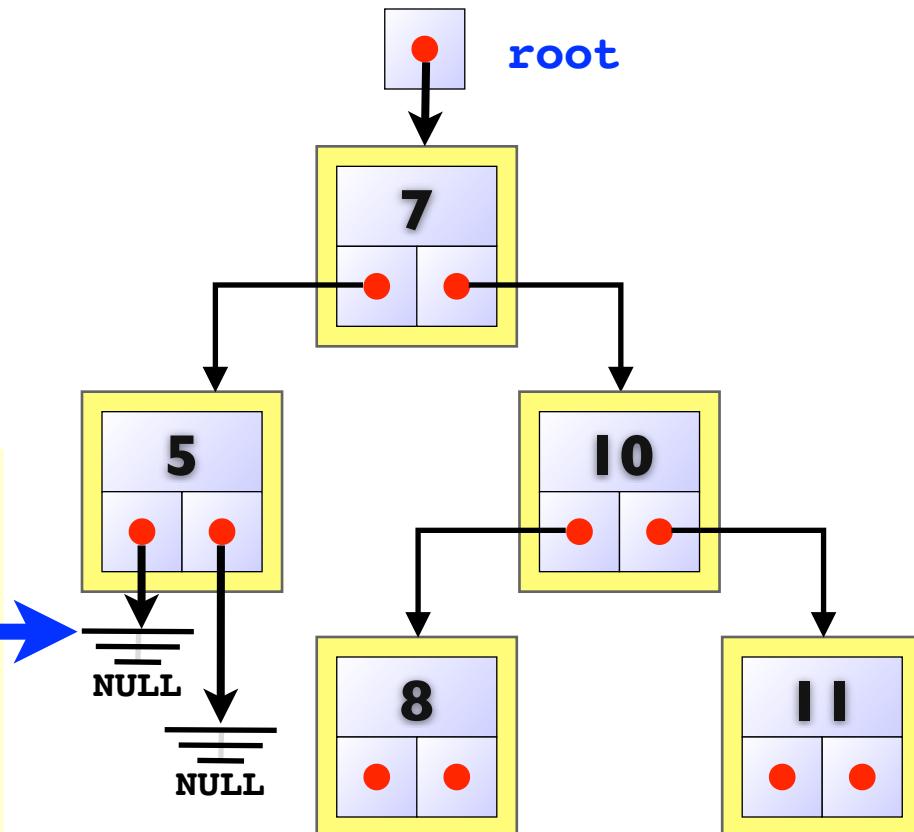
# Binärbaum: Implementierung mit Zeigern (C++)

```
struct treeNode {  
    elemType key;          // Wert des Knotens  
    treeNode *left;        // Zeiger auf linken Sohn  
    treeNode *right;       // Zeiger auf rechten Sohn  
}
```

## Schematische Darstellung:

Blätter werden wie Endknoten bei linearen Listen realisiert; also entweder

- null-Zeiger,
- Zeiger auf das Blatt selbst, oder
- Zeiger auf speziellen Dummy-Knoten



# Binärbaum: Array-Einbettung

**Eignet besonders zur Darstellung von vollständigen Bäumen.**

# Binärbaum: Array-Einbettung

**Eignet besonders zur Darstellung von vollständigen Bäumen.**

## D Vollständiger Binärbaum

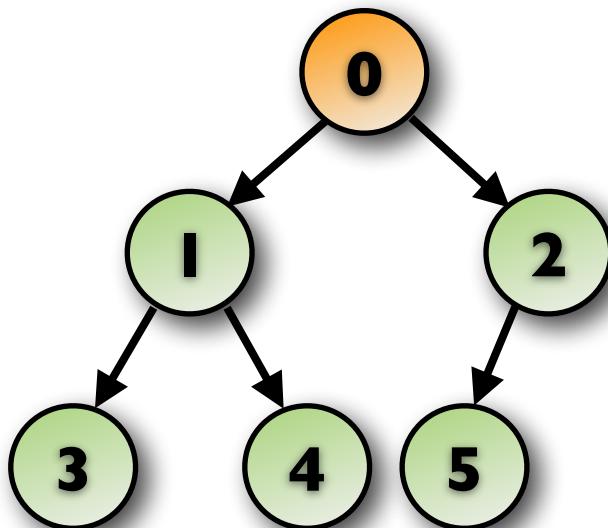
Ein Binärbaum heisst **vollständig**, wenn alle Ebenen vollständig besetzt sind.  
Er heißt **links-vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist.

# Binärbaum: Array-Einbettung

Eignet besonders zur Darstellung von vollständigen Bäumen.

## D Vollständiger Binärbaum

Ein Binärbaum heisst **vollständig**, wenn alle Ebenen vollständig besetzt sind. Er heißt **links-vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist.



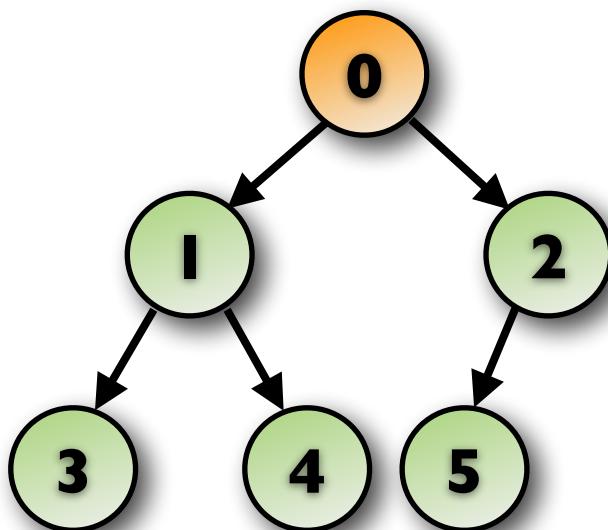
Mit der angegebenen Nummerierung (von links nach rechts - von oben nach unten) lassen sich Vorgänger und Nachfolger (falls existent) folgendermaßen berechnen:

# Binärbaum: Array-Einbettung

Eignet besonders zur Darstellung von vollständigen Bäumen.

## D Vollständiger Binärbaum

Ein Binärbaum heisst **vollständig**, wenn alle Ebenen vollständig besetzt sind. Er heißt **links-vollständig**, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts ausgefüllt ist.



Mit der angegebenen Nummerierung (von links nach rechts - von oben nach unten) lassen sich Vorgänger und Nachfolger (falls existent) folgendermaßen berechnen:

**Vorgänger:**  $\text{pred}(n) = \left\lfloor \frac{n-1}{2} \right\rfloor$

**Nachfolger:**  $\text{succ}(n) = \begin{cases} 2n + 1 & \text{linker Sohn} \\ 2n + 2 & \text{rechter Sohn} \end{cases}$

# Implementierung von Bäumen mit Grad $d > 2$

**Zeiger auf die Nachfolgeknoten werden in einem Feld gespeichert:**

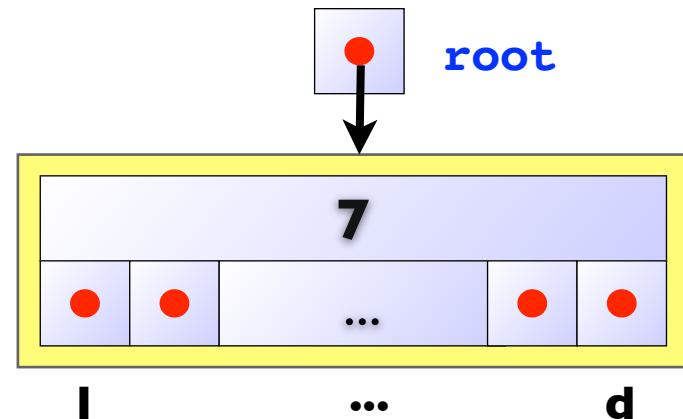
```
template <int d=2, typename ElemType=int>
struct treeNode {
    ElemType key;
    treeNode *children[d];
}
```

# Implementierung von Bäumen mit Grad $d > 2$

**Zeiger auf die Nachfolgeknoten werden in einem Feld gespeichert:**

```
template <int d=2, typename ElemType=int>
struct treeNode {
    ElemType key;
    treeNode *children[d];
}
```

**Schematische Darstellung:**

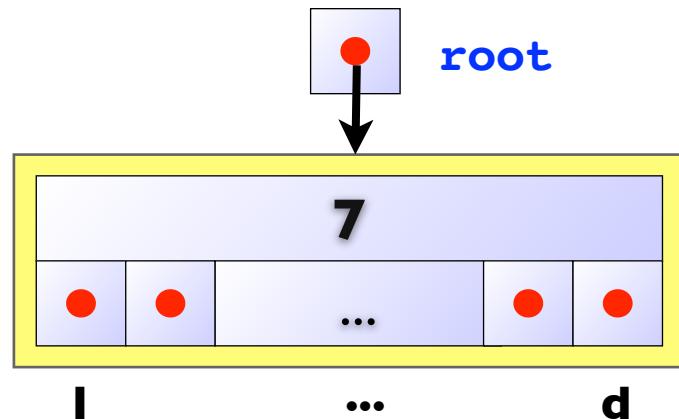


# Implementierung von Bäumen mit Grad $d > 2$

**Zeiger auf die Nachfolgeknoten werden in einem Feld gespeichert:**

```
template <int d=2, typename ElemType=int>
struct treeNode {
    ElemType key;
    treeNode *children[d];
}
```

**Schematische Darstellung:**



**Nachteile dieser Darstellung:**

- der maximale Grad ist fest vorgegeben
- Speicherplatz wird verschenkt, falls der tatsächliche Grad eines Knotens kleiner  $d$  ist

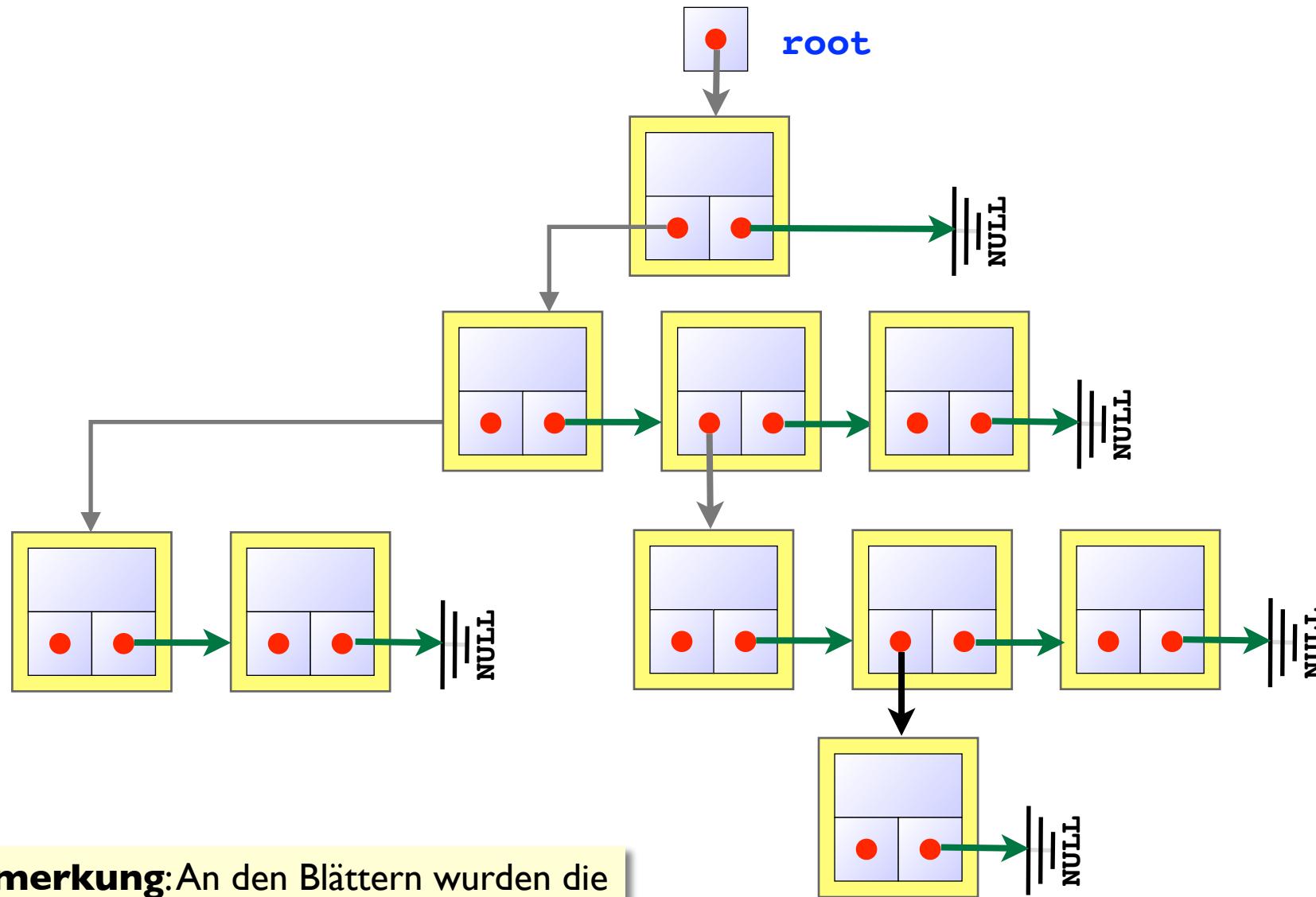
# Darstellung durch Pseudo-Binärbäume I

- **Idee:** Geschwister sind untereinander zu einer linearen Liste verkettet.
- Jeder Knoten verweise auf sein erstes Kind und seinen rechten Bruder (*leftmost child, right sibling*)

```
struct treeNode {  
    elemType key;                      // Wert des Knotens  
    treeNode *leftMostChild;            // Zeiger auf am weitesten  
                                         // links stehenden Kindknoten  
    treeNode *rightSibling;             // Zeiger auf rechten Bruder  
}
```

# Darstellung durch Pseudo-Binärbäume II

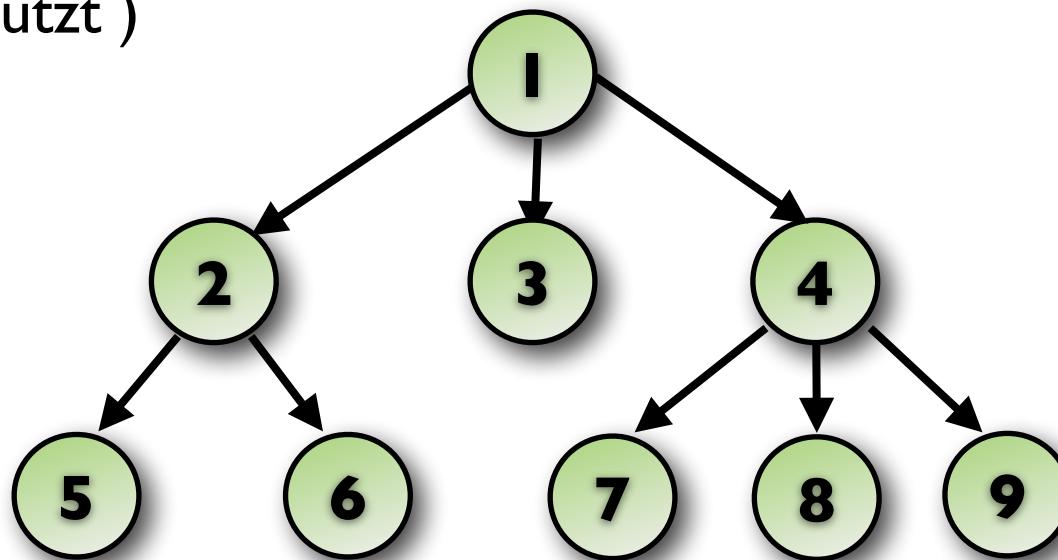
## Schematische Darstellung:



**Anmerkung:** An den Blättern wurden die null-Zeiger z.T. weggelassen

# Einbettung in Array I

- **Knoten werden ebenenweise von links nach rechts durchnummeriert**  
(ähnlich wie beim Binärbaum, jedoch wird die 0 für nicht verwendete Knoten genutzt )

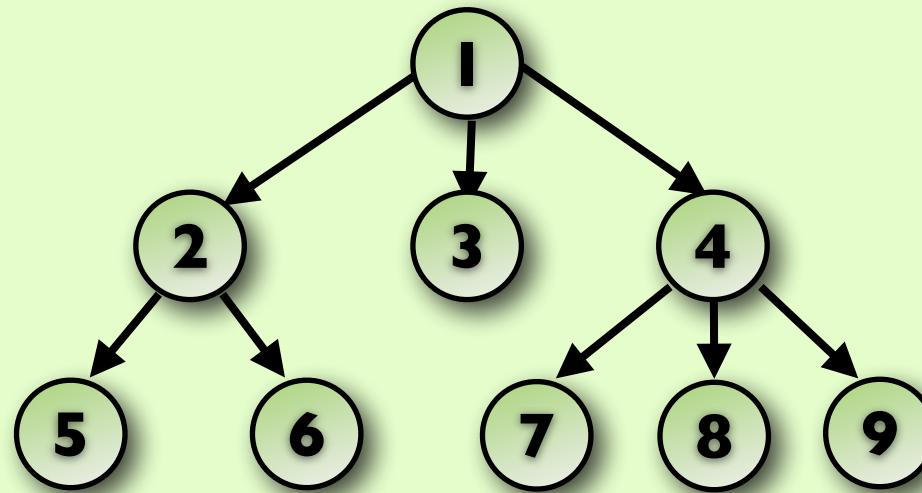


- **Geschwisterknoten stehen dann an benachbarten Array-Positionen**
- **Drei Felder zum Auffinden der Vorgänger und Nachfolger**

```
int parent[n], leftMostChild[n], rightMostChild[n];
```

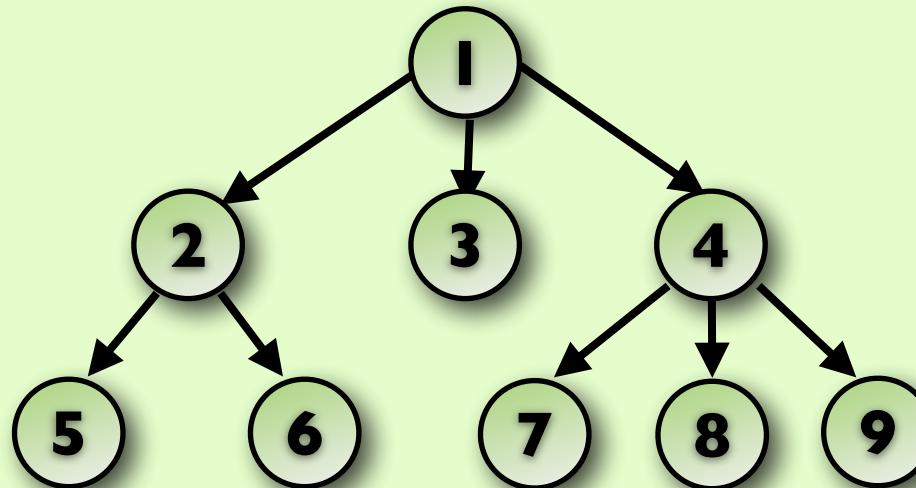
# Einbettung in Array II - Beispiel

## B Einbettung Baum mit Rang $d$ in Array



# Einbettung in Array II - Beispiel

## B Einbettung Baum mit Rang $d$ in Array



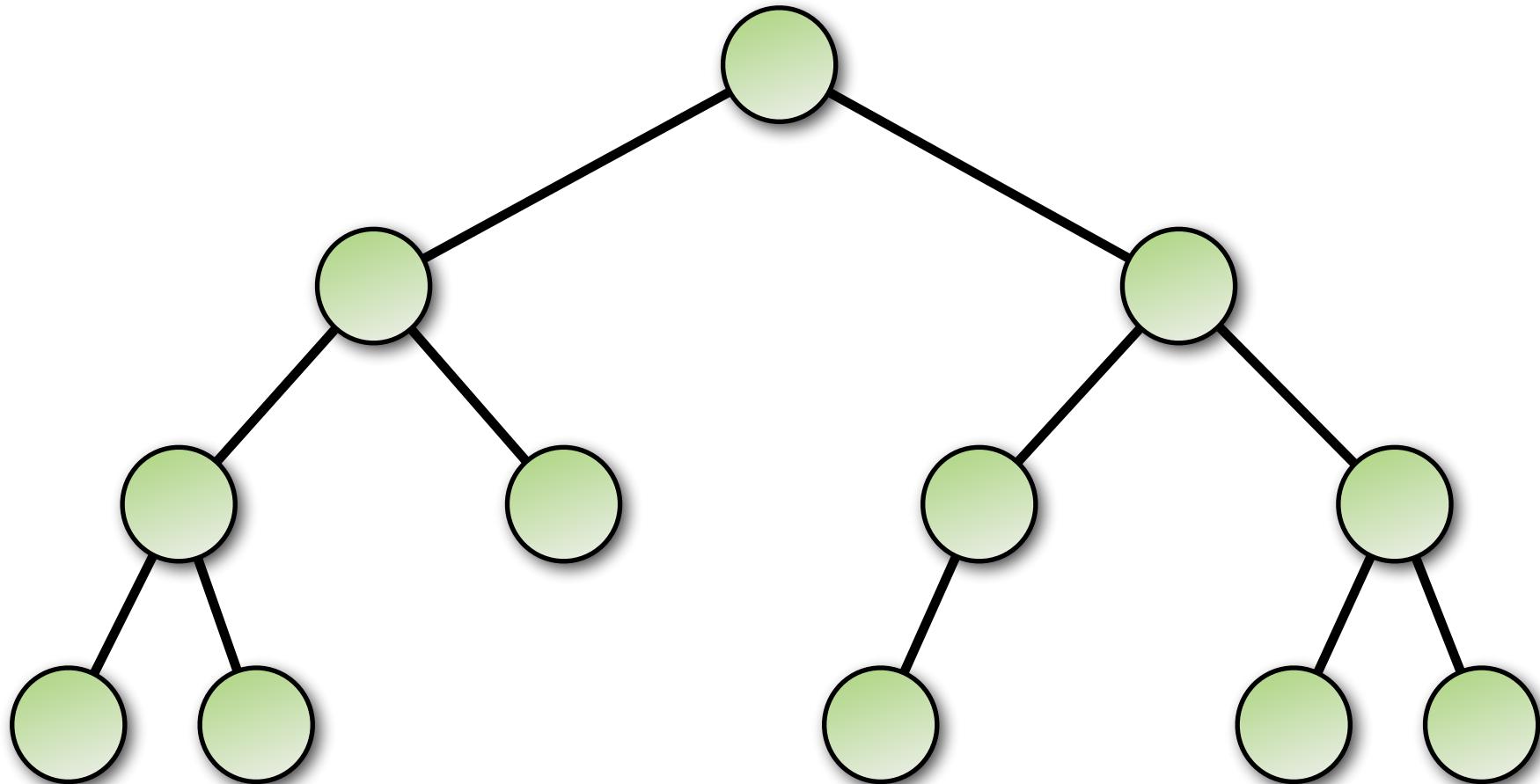
$i$	parent	leftMostChild	rightMostChild
1	0	2	4
2	1	5	6
3	1	0	0
4	1	7	9
5	2	0	0
6	2	0	0
7	4	0	0
8	4	0	0
9	4	0	0

# Durchlaufen eines Baumes

- Aufzählen aller Knoten in einem Baum  
(auch: Durchmusterung, Traversierung, *tree traversal*)
- Man unterscheidet:
  - **Tiefendurchlauf** (*depth first*):  
Zuerst werden alle Teilbäume eines Knotens besucht. Je nach Reihenfolge unterscheidet man weiter
    - **Preorder** (Prefixordnung, Hauptreihenfolge)  
betrachte  $n$ , dann durchlaufe  $n_1, \dots, n_k$
    - **Postorder** (Postfixordnung, Nebenreihenfolge)  
durchlaufe  $n_1, \dots, n_k$ , dann betrachte  $n$
    - **Inorder** (Infixordnung, symmetrische Reihenfolge)  
durchlaufe  $n_1$ , betrachte  $n$ , durchlaufe  $n_2, \dots, n_k$
  - **Breitendurchlauf** (*breadth first, level-order*)  
Knoten werden ihrer Tiefe nach aufgelistet und zwar von links nach rechts (wie Nummerierung bei Speicherung in Array)

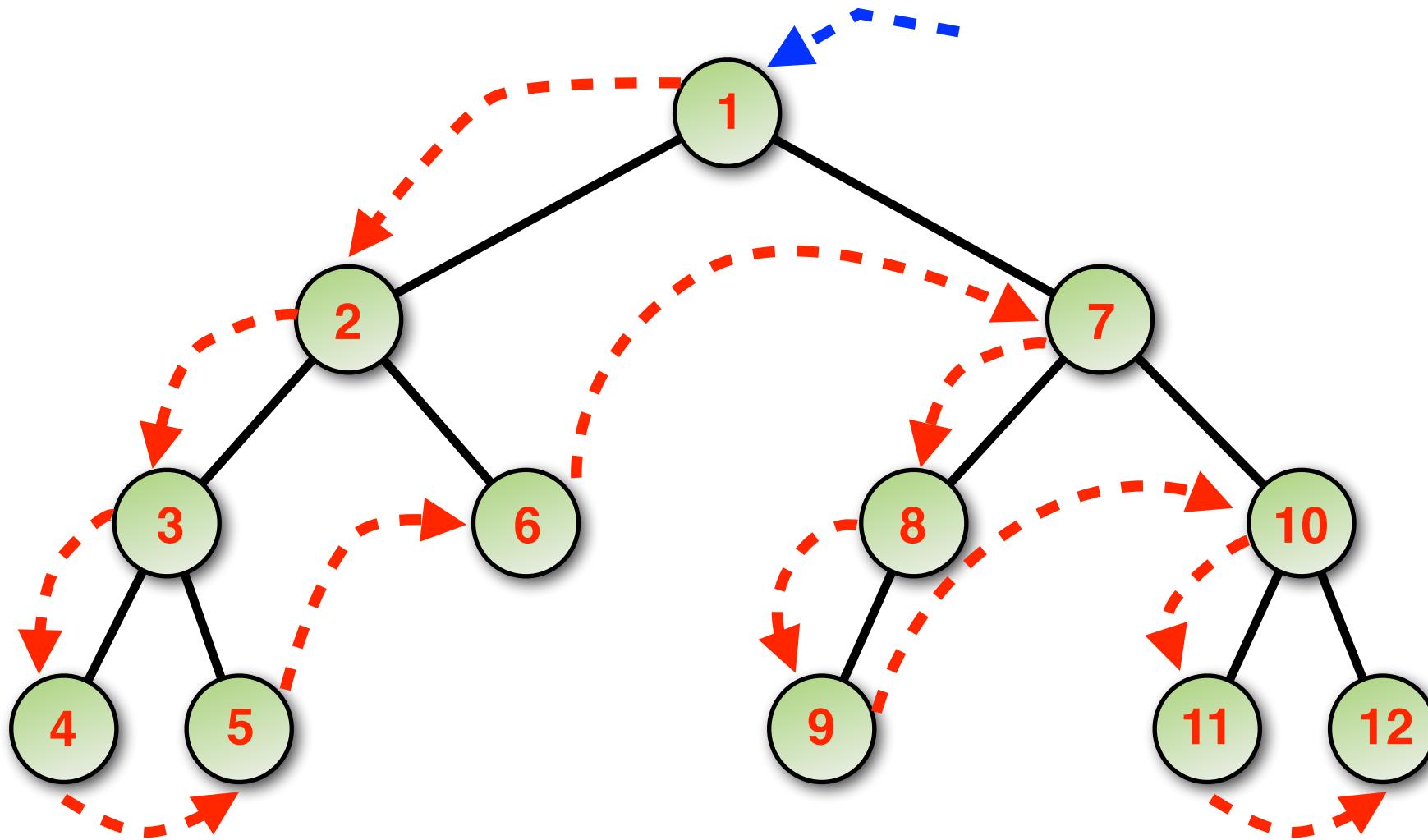
# *depth first: Preorder*

betrachte  $n$ , dann durchlaufe  $n_1, \dots, n_k$



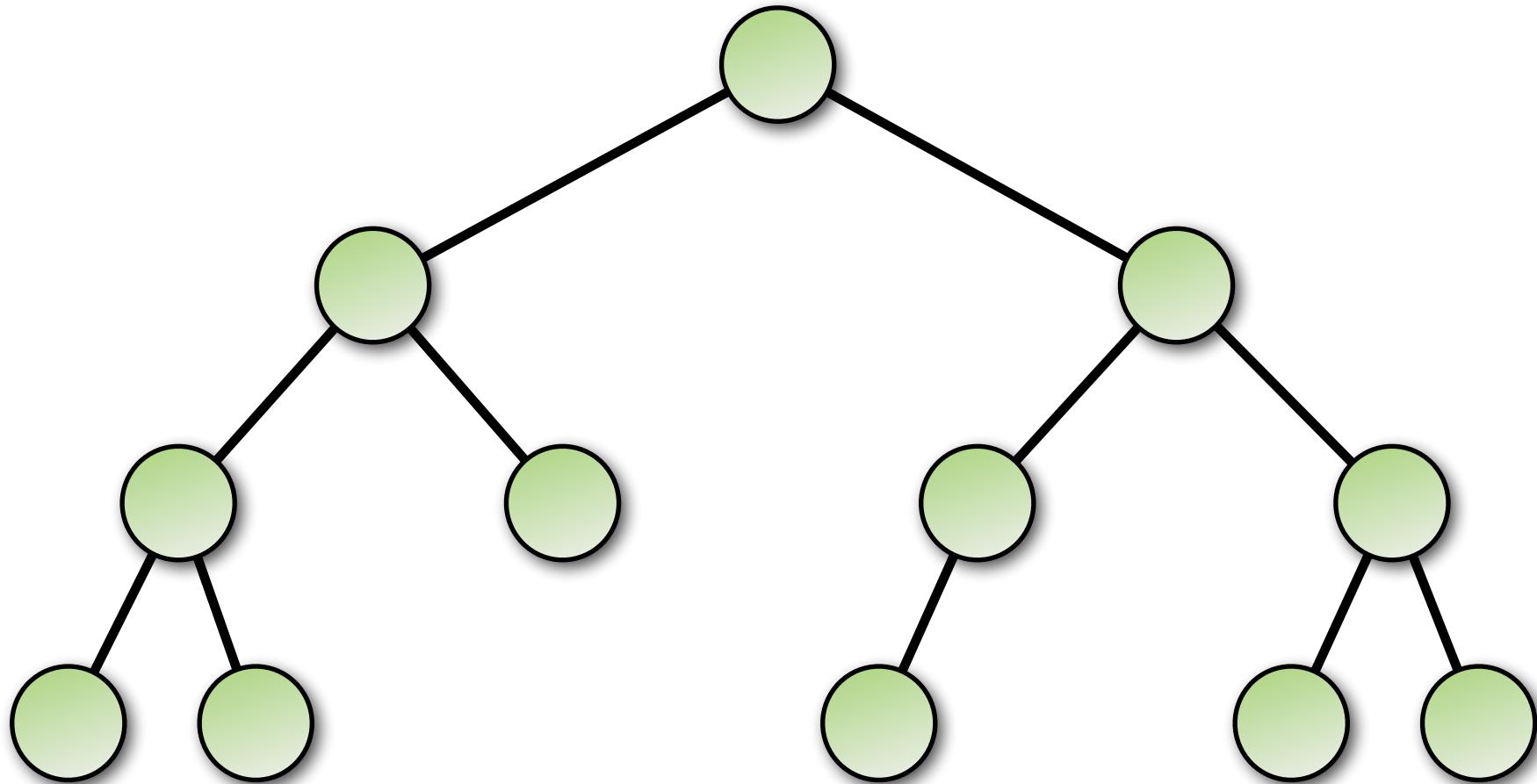
# depth first: Preorder

betrachte  $n$ , dann durchlaufe  $n_1, \dots, n_k$



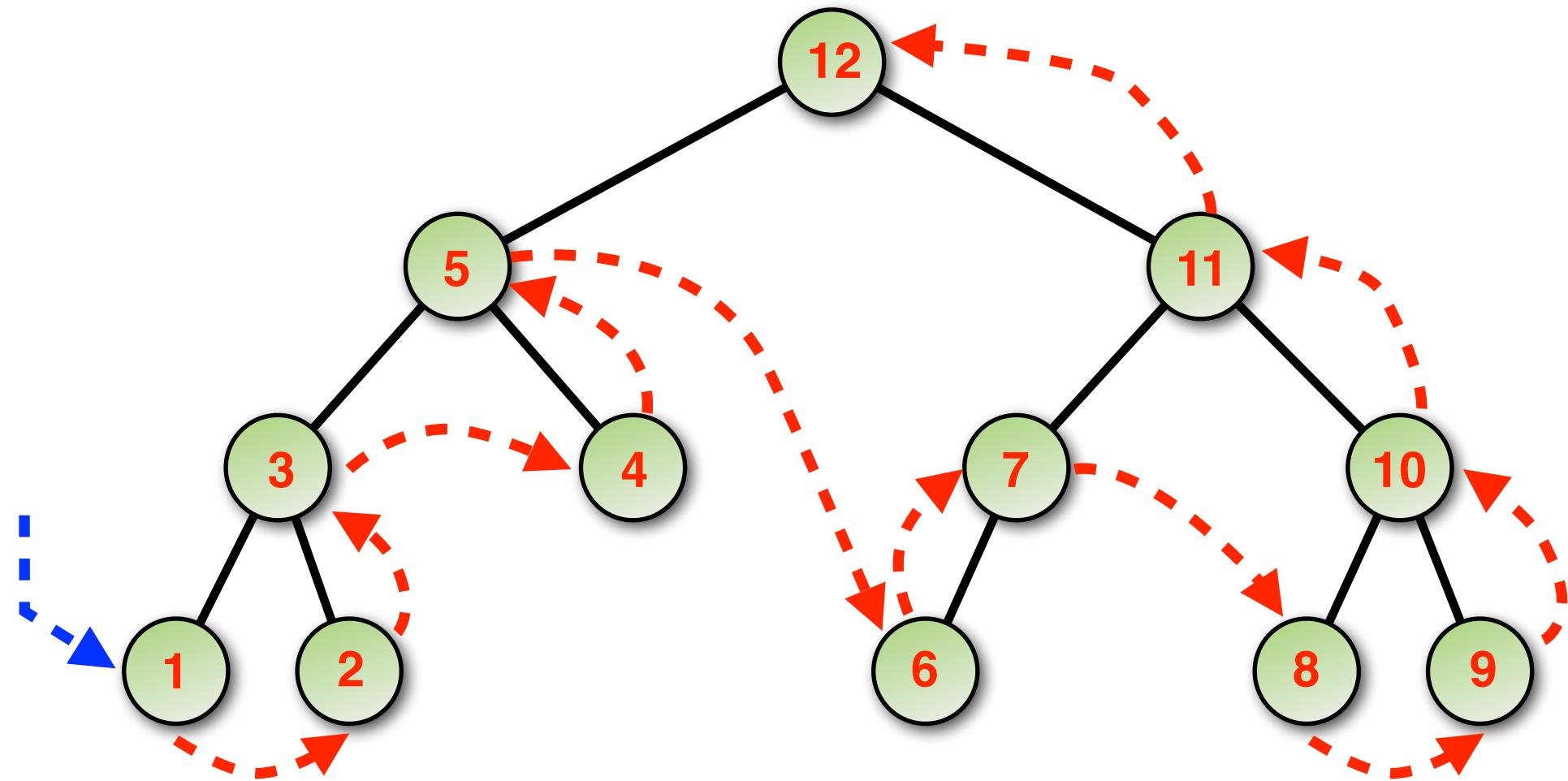
# *depth first: Postorder*

durchlaufe  $n_1, \dots, n_k$ , dann betrachte  $n$



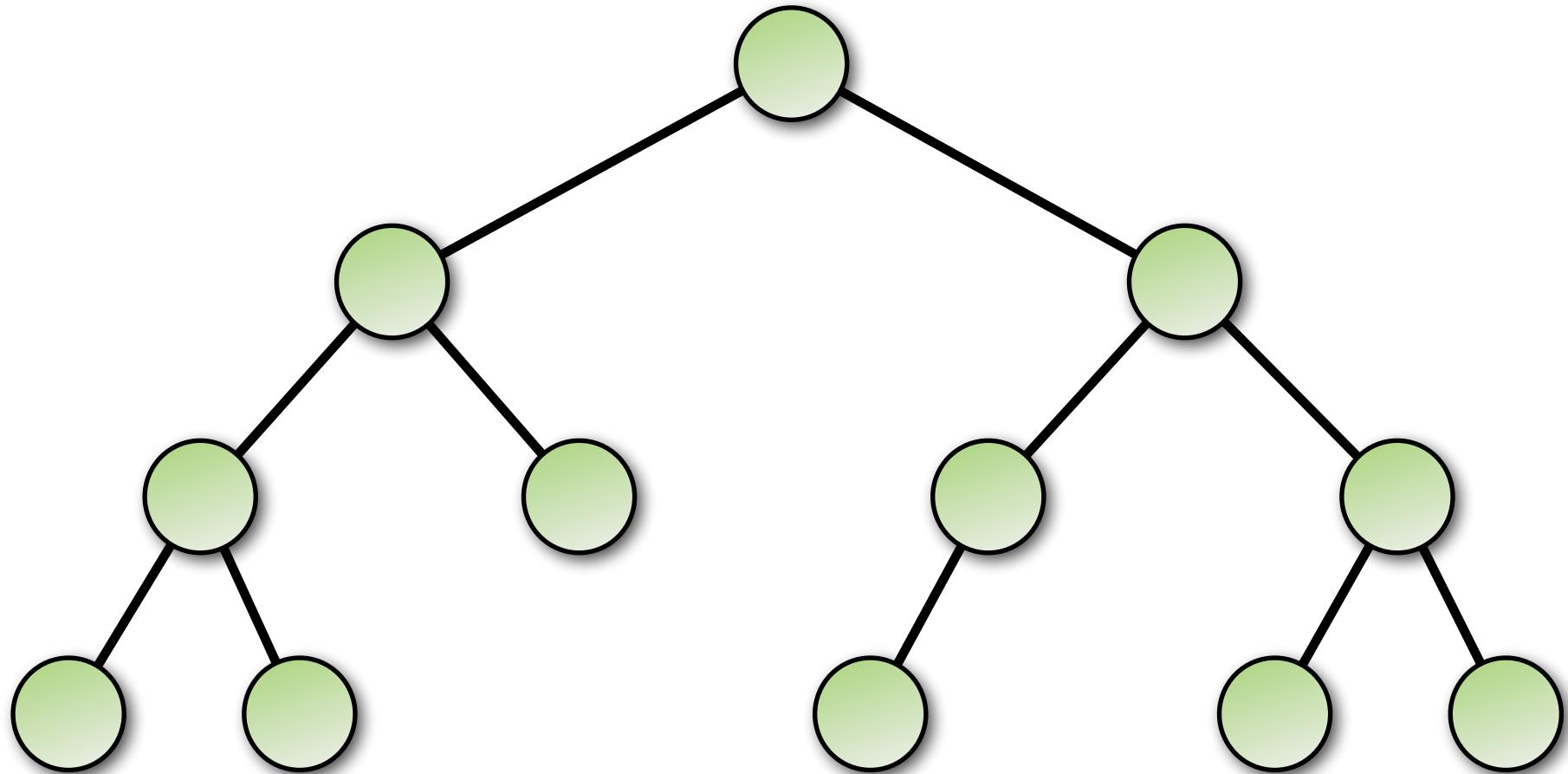
# **depth first: Postorder**

**durchlaufe  $n_1, \dots, n_k$ , dann betrachte  $n$**



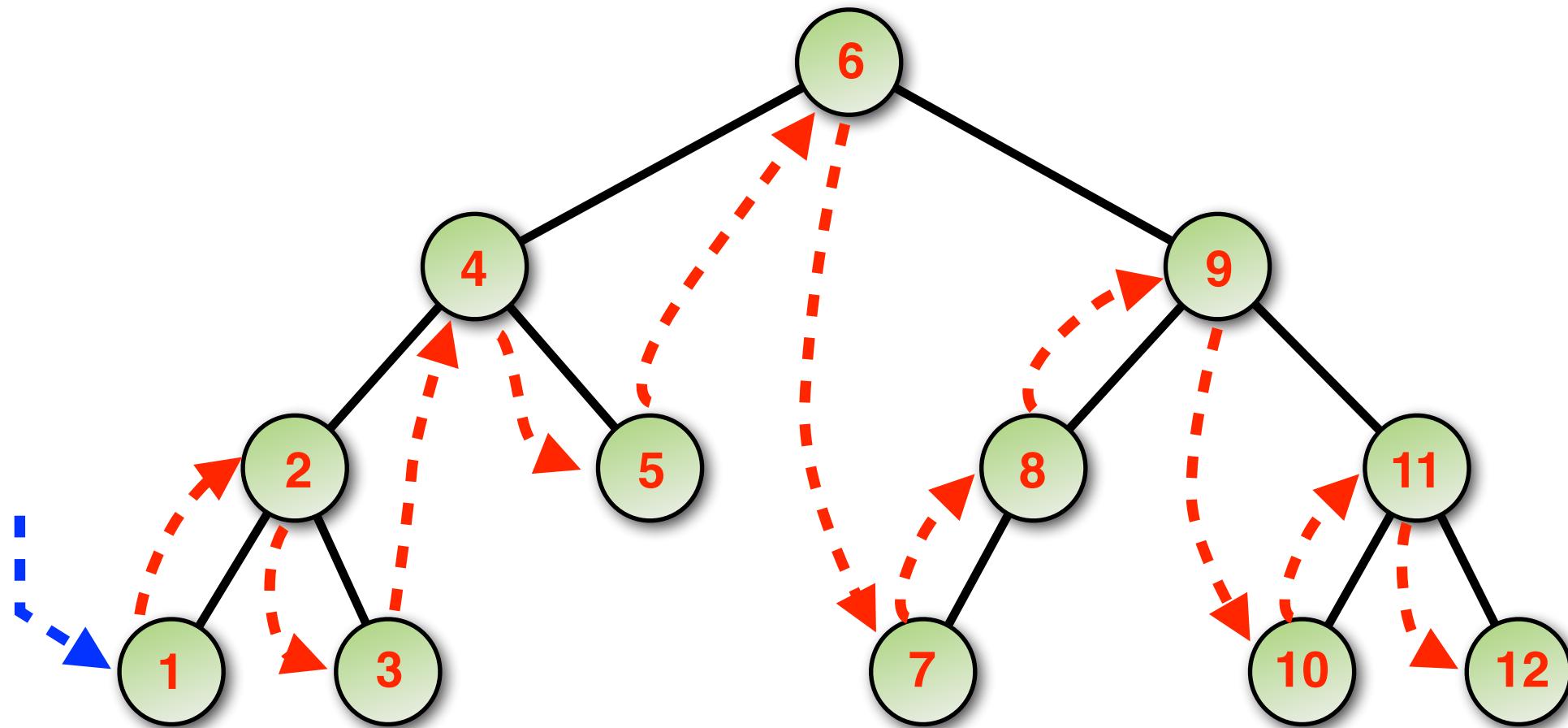
# depth first: Inorder

durchlaufe  $n_1$ , betrachte  $n$ , durchlaufe  $n_2, \dots, n_k$



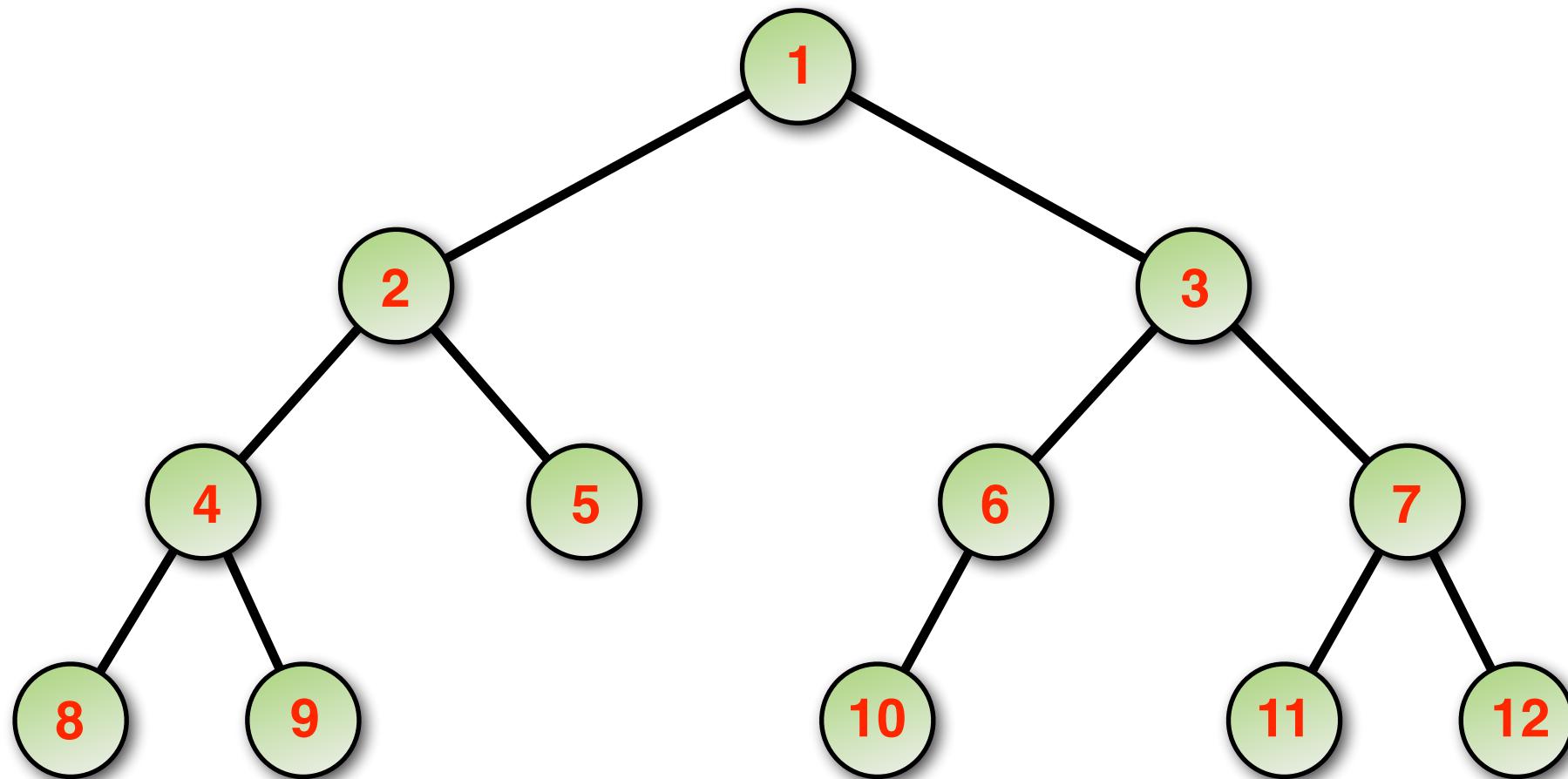
# depth first: Inorder

durchlaufe  $n_1$ , betrachte  $n$ , durchlaufe  $n_2, \dots, n_k$

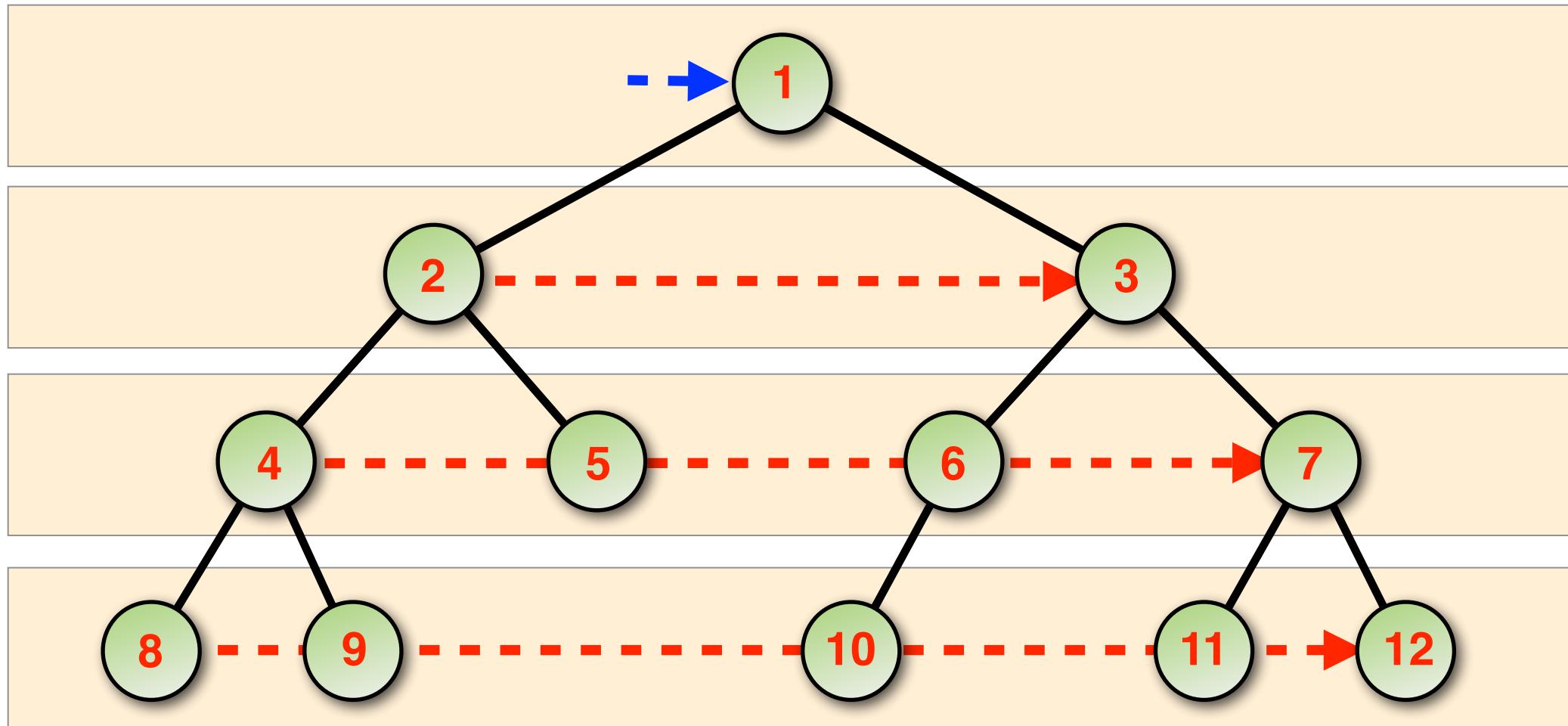


# *breadth first*

**Knoten werden ebenenweise durchlaufen (level order)**

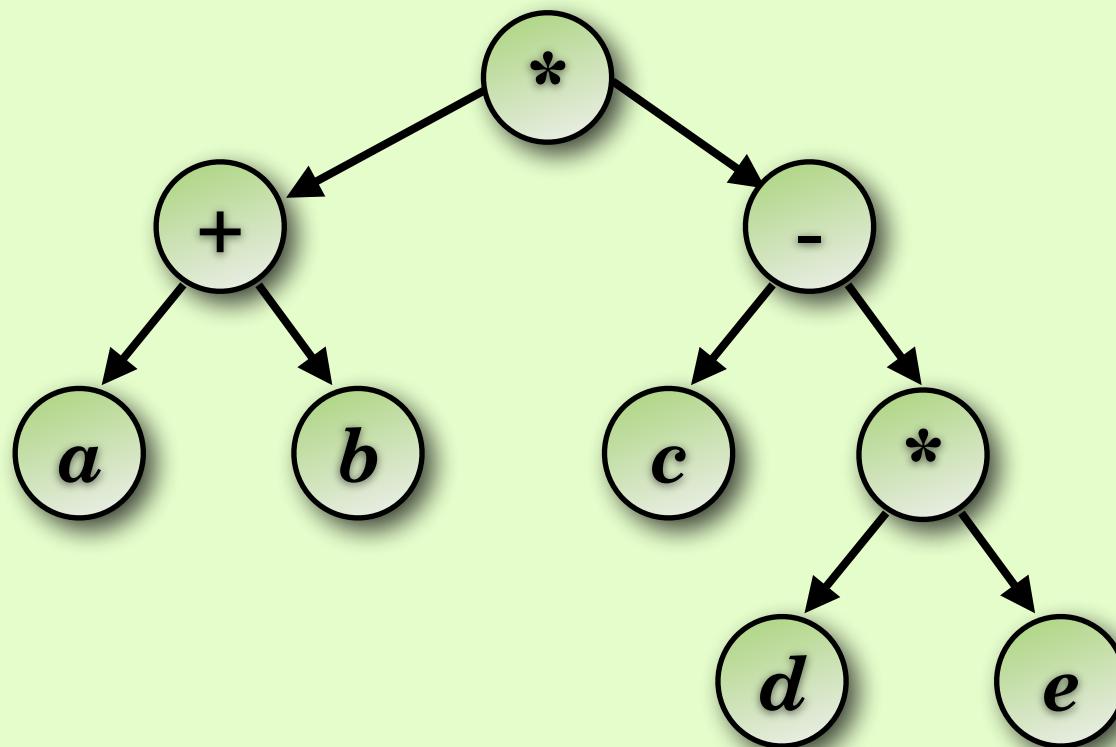


Knoten werden ebenenweise durchlaufen (level order)



# Traversieren: Beispiel

## B Traversieren eines Baumes



**Preorder:**  $*+ab-c*de$  (polnische Notation)

**Postorder:**  $ab+cde-*-$  (umgekehrt polnische Notation)

**Inorder:**  $a+b*c-d*e$  (Infixnotation)

# Traversieren: rekursive Implementierung in C++ I

## ● Preorder

```
void preOrderTraverse(treeNode* tree) {  
    if (tree != 0) {  
        visit(tree);  
        preOrderTraverse(tree->left);  
        preOrderTraverse(tree->right);  
    }  
}
```

## ● PostOrder

```
void postOrderTraverse(treeNode *tree) {  
    if (tree != 0) {  
        postOrderTraverse(tree->left);  
        postOrderTraverse(tree->right);  
        visit(tree);  
    }  
}
```

# Traversieren: rekursive Implementierung in C++ II

- **Inorder:**

```
void inOrderTraverse(treeNode *tree) const {  
    if (tree != 0) {  
        inOrderTraverse(tree->left);  
        visit(tree);  
        inOrderTraverse(tree->right);  
    }  
}
```

**visit** führt eine beliebige Operation auf dem Knoten aus; z.B.:

```
void visit(treeNode *n) {  
    cout << n->key << " ";  
}
```

# Traversieren: level-order in C++

- **level-order traversal mit queue:**

```
void levelOrderTraverse(const treeNode* tree) const {  
    deque<treeNode*> q;  
    q.push_back( tree ); // Wurzelknoten in queue  
    do {  
        treeNode* n = q.pop_front();  
        if (n!=0) {  
            // Kindknoten in queue einreihen  
            visit( n );  
            q.push_back( q->left );  
            q.push_back( q->right );  
        }  
    } while ( ! q.empty() );  
}
```

# Homogene / Heterogene Bäume

- Bezogen auf den Knotentyp unterscheidet man zwei Arten von Bäumen

# Homogene / Heterogene Bäume

- Bezogen auf den Knotentyp unterscheidet man zwei Arten von Bäumen
- **Homogene Bäume**
  - Alle Knoten haben eine gemeinsame Struktur (z.B. TreeNode)

# Homogene / Heterogene Bäume

- Bezogen auf den Knotentyp unterscheidet man zwei Arten von Bäumen
- **Homogene Bäume**
  - Alle Knoten haben eine gemeinsame Struktur (z.B. TreeNode)
- **Heterogene Bäume**
  - Knoten unterscheiden sich in ihrer Struktur
  - Die Zahl der „gültigen Knotentypen“ ist beschränkt
    - Objektorientierte Sprachen (OO): Klassenhierarchie
    - Funktionale Sprachen (FP): Algebraischer Datentyp (Summentyp)  
Aber: Parallelen zu homogener Implementierung (Tags)

# Homogene Bäume - Beispiel (C++ - I)

```
struct HomExpr {  
    // Knotenart: Entweder eine Operation oder ein Wert  
    typedef enum { tOpSymbol, tValue } t_nodeKind;  
    // Art von binären Operation  
    typedef enum { opPlus,opMinus,opMult,opDiv} t_opKind;  
    // Den Inhalt eines Knotens kodieren wir als Union; t_node ist also  
    // entweder eine (V)alue- oder ein (O)perator-Struktur  
    typedef union { struct V {  
        int value; // Integer-Wert  
    } value;  
    struct O {  
        t_opKind opKind; // Operatorart  
        HomExpr *left,*right; // Linker und Rechter Kindknoten  
    } op;  
    } t_node;  
  
    t_nodeKind nodeKind; // Kontenart  
    t_node node; // Knotenwert  
  
    HomExpr(int val) :  
        nodeKind(tValue) { node.value.value=val; }  
    HomExpr(t_opKind op, HomExpr* l, HomExpr* r) :  
        nodeKind(tOpSymbol) { node.op.opKind=op; node.op.left=l; node.op.right=r; }  
};
```

# Homogene Bäume - Beispiel (C++ - I)

```
struct HomExpr {  
    // Knotenart: Entweder eine Operation oder ein Wert  
    typedef enum { tOpSymbol, tValue } t_nodeKind;  
    // Art von binären Operation  
    typedef enum { opPlus,opMinus,opMult,opDiv} t_opKind;  
    // Den Inhalt eines Knotens kodieren wir als Union; t_node ist also  
    // entweder eine (V)alue- oder ein (O)perator-Struktur  
    typedef union { struct V {  
        int value; // Integer-Wert  
    } value;  
    struct O {  
        t_opKind opKind; // Operatorart  
        HomExpr *left,*right; // Linker und Rechter Kindknoten  
    } op;  
    } t_node;  
  
    t_nodeKind nodeKind; // Kontenart  
    t_node node; // Knotenwert  
  
    HomExpr(int val) :  
        nodeKind(tValue) { node.value.value=val; }  
    HomExpr(t_opKind op, HomExpr* l, HomExpr* r) :  
        nodeKind(tOpSymbol) { node.op.opKind=op; node.op.left=l; node.op.right=r; }  
};
```

Werden  
platzsparend  
auf **identische**  
Speicherbereiche  
abgebildet!

# Homogene Bäume - Beispiel (C++ - 2)

```
int eval(HomExpr* e) {
    switch (e->nodeKind) { // Wert oder Operator?
        case HomExpr::tValue:
            return e->node.value.value; break;
        case HomExpr::tOpSymbol:
            // linken und rechten Teilbaum ausrechnen
            int l0p = eval(e->node.op.left);
            int r0p = eval(e->node.op.right);
            switch (e->node.op.opKind) { // Ausrechnen ...
                case HomExpr::opPlus: return l0p + r0p; break;
                case HomExpr::opMinus: return l0p - r0p; break;
                case HomExpr::opMult: return l0p * r0p; break;
                case HomExpr::opDiv: return l0p / r0p; break;
            }
    }
}
```

# Homogene Bäume - Beispiel (JAVA - I)

Da JAVA keine **unions** kennt, arbeiten wir mit interfaces ...

```
public class HomExpr {  
    // Knotenarten  
    public enum NodeKind { VALUE, BINOP }  
  
    // Jeder Knoten muss Auskunft über seine Art geben können  
    public interface Node {  
        NodeKind Kind();  
    }  
  
    // Knoten für ganze Zahlen sind immer Blätter  
    public final class Value implements Node {  
        public int value;  
        public NodeKind Kind() { return NodeKind.VALUE; }  
        public Value(int value) { this.value = value; }  
    }  
    ...
```

# Homogene Bäume - Beispiel (JAVA - 2)

```
public class HomExpr {  
    ...  
    // Binäre Operatoren, die wir unterstützen  
    public enum BinOp { ADD, SUB, MUL, DIV }  
    // Knoten für binäre Operatoren haben immer zwei Kinder  
    // und einen Operationstyp  
    public final class BinOpNode implements Node {  
        public BinOp op;  
        public Node left, right;  
        public NodeKind Kind() { return NodeKind.BINOP; }  
        public BinOpNode(BinOp op, Node left, Node right) {  
            this.op = op; this.left = left; this.right = right;  
        }  
    }  
    ...  
}
```

# Homogene Bäume - Beispiel (JAVA - 3)

```
public class HomExpr {  
    ...  
    // Auswertung eines Baums  
    public static int eval(Node node) {  
        switch (node.Kind()) {  
            case VALUE:  
                return ((Value)node).value;  
            case BINOP:  
                BinOpNode binOp = (BinOpNode)node;  
                int l = eval(binOp.left);  int r = eval(binOp.right);  
                switch (binOp.op) {  
                    case ADD: return l + r;  
                    case SUB: return l - r;  
                    case MUL: return l * r;  
                    case DIV: return l / r;  
                }  
            }  
        return 0;  
    }  
    ...
```

# Homogene Bäume - Beispiel (JAVA - 4)

```
public class HomExpr {  
    ...  
    Node root; // Wurzel des Baumes  
  
    int eval() { // Convenience-Methode zur Auswertung  
        return eval(root);  
    }  
  
    // Konstruktoren für Bäume  
    HomExpr(int value) {  
        this.root = new Value(value);  
    }  
    HomExpr(BinOp op, HomExpr left, HomExpr right) {  
        this.root = new BinOpNode(op, left.root, right.root);  
    }  
}
```

# Homogene Bäume - Beispiel (JAVA - 4)

```
public class HomExpr {  
    ...  
    Node root; // Wurzel des Baumes  
  
    int eval() { // Convenience-Methode zur Auswertung  
        return eval(root);  
    }  
  
    // Konstruktoren für Bäume  
    HomExpr(int value) {  
        this.root = new Value(value);  
    }  
    HomExpr(BinOp op, HomExpr left, HomExpr right) {  
        this.root = new BinOpNode(op, left.root, right.root);  
    }  
}
```

**Kompromiss**, da so eigentlich **nicht wirklich homogen!**

Dazu hätten wir Node, Value und Op in **einer** Klasse vereinen müssen, würden dann aber Platz verschenken.

# Heterogene Bäume - Beispiel (**JAVA** - I)

Hier nutzen wir die Inklusionspolymorphie (**Inheritance**)

```
// interface für einen Knoten des Expression-Baums
public interface Expr {
    public int eval();
}

// Abstrakte Basisklasse für Knoten, die binäre Operationen
// repräsentieren.
public abstract class BinOp implements Expr {
    // Die zu verknüpfenden Teilbäume (Teil-Ausdrücke)
    protected Expr left;
    protected Expr right;

    public BinOp(Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }
}
```

# Heterogene Bäume - Beispiel (JAVA - 2)

```
// Knoten, der einen ganzzahligen Wert repräsentiert.  
// Diese Knoten sind immer Blätter im Expression-Baum.  
public class Value implements Expr {  
    private int value;  
  
    public Value(int value) { this.value = value; }  
    public int eval() { return value; }  
}  
  
// Konkrete binäre Operationen stellen eine geeignete eval()-Methode  
bereit.  
public class OpAdd extends BinOp {  
    public OpAdd(Expr left, Expr right) {  
        super(left, right);  
    }  
  
    public int eval() {  
        return left.eval() + right.eval();  
    }  
}
```

# Heterogene Bäume - Beispiel (JAVA - 2)

```
// Knoten, der einen ganzzahligen Wert repräsentiert.  
// Diese Knoten sind immer Blätter im Expression-Baum.  
public class Value implements Expr {  
    private int value;  
  
    public Value(int value) { this.value = value; }  
    public int eval() { return value; }  
}  
  
// Konkrete binäre Operationen stellen eine geeignete eval()-Methode  
bereit.  
public class OpAdd extends BinOp {  
    public OpAdd(Expr left, Expr right) {  
        super(left, right);  
    }  
  
    public int eval() {  
        return left.eval() + right.eval();  
    }  
}
```



Implizite Berücksichtigung des  
Knotentyps (*dynamic dispatch*)

# Heterogene Bäume - Beispiel (C++ - I)

## B Heterogener Baum für Ausdrücke - I

Alle Knotentypen müssen ein „interface“ implementieren:

```
// Abstrakte Basisklasse. Alle Knotentypen müssen
// dieses "interface" implementieren
struct hetExpr {
    hetExpr() {}
    virtual ~hetExpr() = 0;
    virtual int eval() const = 0;
};

// Operatorsymbole können entweder innere Knoten
// oder der Wurzelknoten sein - sie haben daher
// immer Kindknoten
struct binOp : public hetExpr {
    hetExpr *left, *right;
    binOp(hetExpr* l,hetExpr* r) : left(l),right(r) {}
    virtual ~binOp() = 0;
    virtual int eval() const = 0;
};
```

# Heterogene Bäume - Beispiel (C++ - 3)

## B Heterogener Baum für Ausdrücke - II

```
// Ein int-Wert ist immer ein Blatt!
struct intval : public hetExpr {
    int iValue;
    intval(int i) : iValue(i) {}
    ~intval() {}
    virtual int eval() const { return iValue; }
};

// Operatoren stellen eine entsprechende eval-Methode
// zur Verfügung:
struct opPlus : public binOp {
    ~opPlus() {}
    opPlus(hetExpr* l,hetExpr* r) : binOp(l,r) {}
    virtual int eval() const { // inorder-Traversal ...
        return left->eval() + right->eval();
    }
};

int eval(hetExpr* e) { return e->eval(); }
```

# Heterogene Bäume - Beispiel (C++ - 3)

## B Heterogener Baum für Ausdrücke - II

```
// Ein int-Wert ist immer ein Blatt!
struct intval : public hetExpr {
    int iValue;
    intval(int i) : iValue(i) {}
    ~intval() {}
    virtual int eval() const { return iValue; }
};

// Operatoren stellen eine entsprechende eval-Methode
// zur Verfügung:
struct opPlus : public binOp {
    ~opPlus() {}
    opPlus(hetExpr* l,hetExpr* r) : binOp(l,r) {}
    virtual int eval() const { // inorder-Traversal ...
        return left->eval() + right->eval();
    }
};

int eval(hetExpr* e) { return e->eval(); }
```

Implizite Berücksichtigung des Knotentyps (*dynamic dispatch*)

## ● Homogene Bäume

- Knoten haben identische Struktur (i.d.R. **ein** Knotentyp)
- Funktionen lassen sich leicht ergänzen
  - Man schreibt einfach eine neue
- Neue Knotentypen lassen sich nur schwer hinzufügen
  - Alle bereits implementierten Funktionen müssen angepasst werden

# Das Erweiterbarkeitsproblem

## ● Homogene Bäume

- Knoten haben identische Struktur (i.d.R. **ein** Knotentyp)
- Funktionen lassen sich leicht ergänzen
  - Man schreibt einfach eine neue
- Neue Knotentypen lassen sich nur schwer hinzufügen
  - Alle bereits implementierten Funktionen müssen angepasst werden

## ● Heterogene Bäume

- Knoten dürfen sich in ihrer Struktur (ihrem Typ) unterscheiden
- Knotentypen lassen sich sehr leicht ergänzen
  - z.B.: Einfach eine neue Klasse ableiten (OO)
  - Neuen Summanden zu Summentyp hinzufügen (FP)
- Funktionen lassen sich nur sehr schwer hinzufügen
  - Alle Knotentypen müssen angepasst werden

# Das Erweiterbarkeitsproblem

## ● Homogene Bäume

- Knoten haben identische Struktur (i.d.R. **ein** Knotentyp)
- Funktionen lassen sich leicht ergänzen
  - Man schreibt einfach eine neue
- Neue Knotentypen lassen sich nur schwer hinzufügen
  - Alle bereits implementierten Funktionen müssen angepasst werden

## ● Heterogene Bäume

- Knoten dürfen sich in ihrer Struktur (ihrem Typ) unterscheiden
- Knotentypen lassen sich sehr leicht ergänzen
  - z.B.: Einfach eine neue Klasse ableiten (OO)
  - Neuen Summanden zu Summentyp hinzufügen (FP)
- Funktionen lassen sich nur sehr schwer hinzufügen
  - Alle Knotentypen müssen angepasst werden

## ● Fazit: **Erweiterbarkeitsproblem!**

- Entweder um Knoten oder Funktionen erweiterbar!