SS 2021

Algorithmen und Datenstrukturen

3. Kapitel

Binärbäume

Martin Dietzfelbinger

Mai 2021

3.1 Binärbäume: Grundlagen

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum
- als Repräsentation von binären Ausdrücken (logisch, arithmetisch)

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum
- als Repräsentation von binären Ausdrücken (logisch, arithmetisch)
- als Entscheidungsbaum

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum
- als Repräsentation von binären Ausdrücken (logisch, arithmetisch)
- als Entscheidungsbaum
- als Codierungs-/Decodierungsbaum

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum
- als Repräsentation von binären Ausdrücken (logisch, arithmetisch)
- als Entscheidungsbaum
- als Codierungs-/Decodierungsbaum
- als binärer **Heap** (später)

3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum
- als Repräsentation von binären Ausdrücken (logisch, arithmetisch)
- als Entscheidungsbaum
- als Codierungs-/Decodierungsbaum
- als binärer **Heap** (später)
- ...

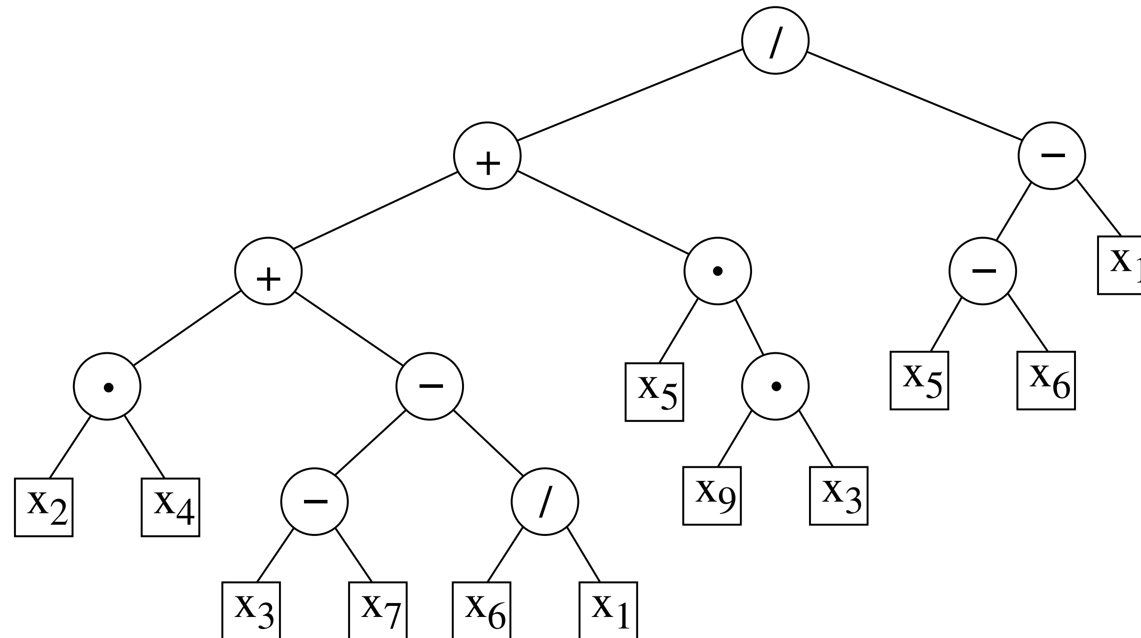
Beispiel 1: **Arithmetischer Ausdruck** mit zweistelligen Operatoren:

Beispiel 1: **Arithmetischer Ausdruck** mit zweistelligen Operatoren:

$((((x_2 \cdot x_4) + ((x_3 - x_7) - (x_6/x_1))) + (x_5 \cdot (x_9 \cdot x_3)))/((x_5 - x_6) - x_1))$

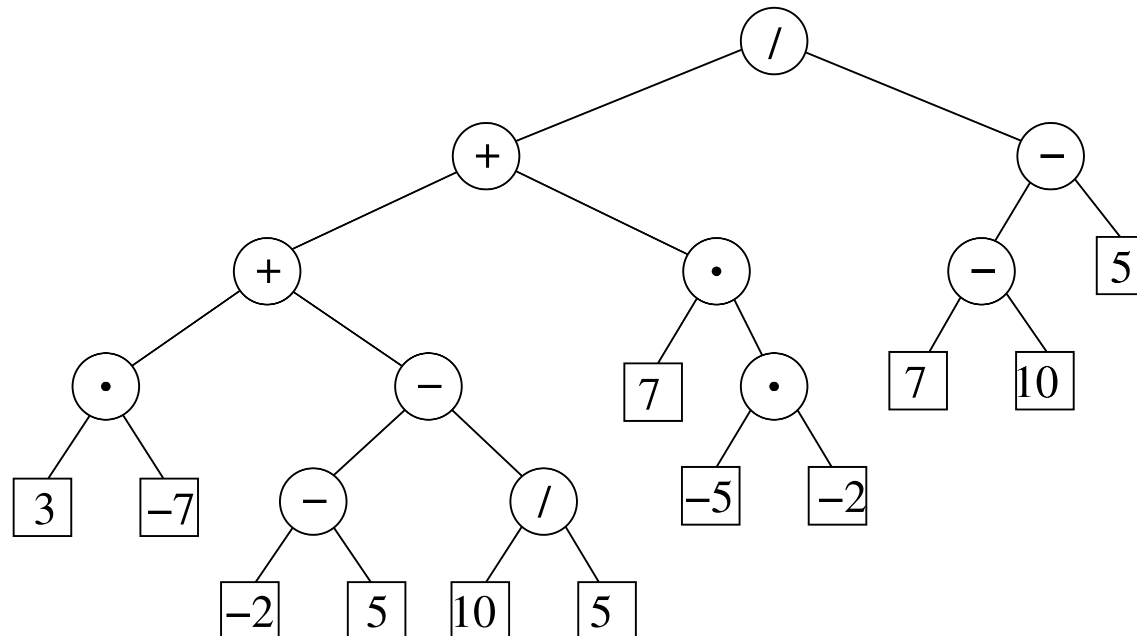
Beispiel 1: **Arithmetischer Ausdruck** mit zweistelligen Operatoren:

$((((x_2 \cdot x_4) + ((x_3 - x_7) - (x_6/x_1))) + (x_5 \cdot (x_9 \cdot x_3)))/((x_5 - x_6) - x_1))$



Auswertung:

1) Ordne den Blättern konkrete Werte zu,
z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

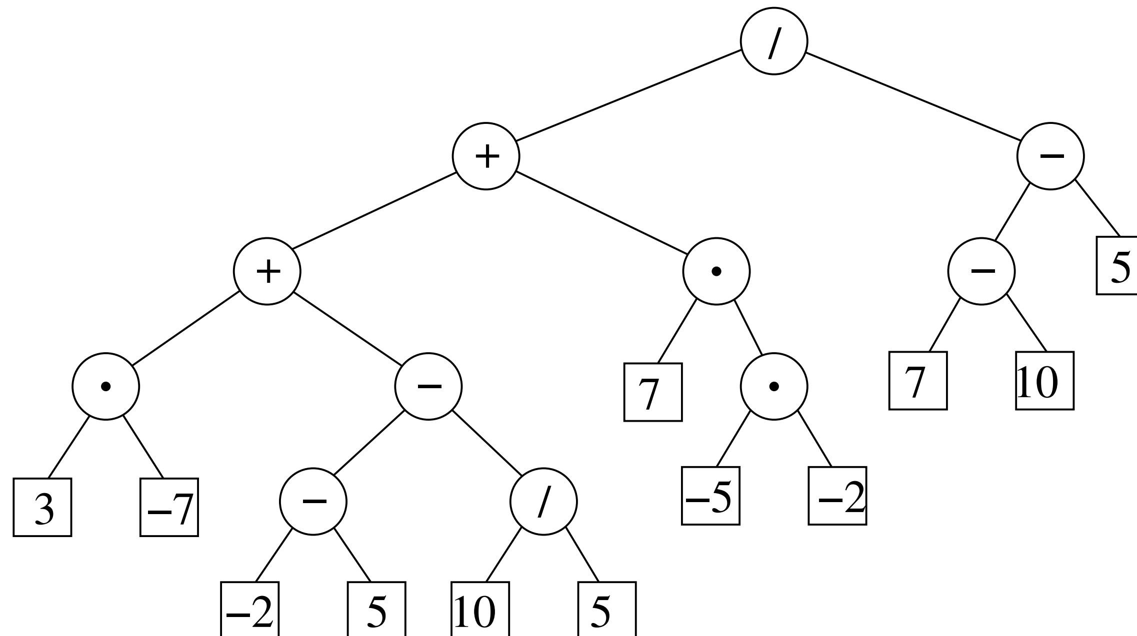


Auswertung:

1) Ordne den Blättern konkrete Werte zu,

z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

2) Werte aus, von den Blättern startend („bottom-up“).

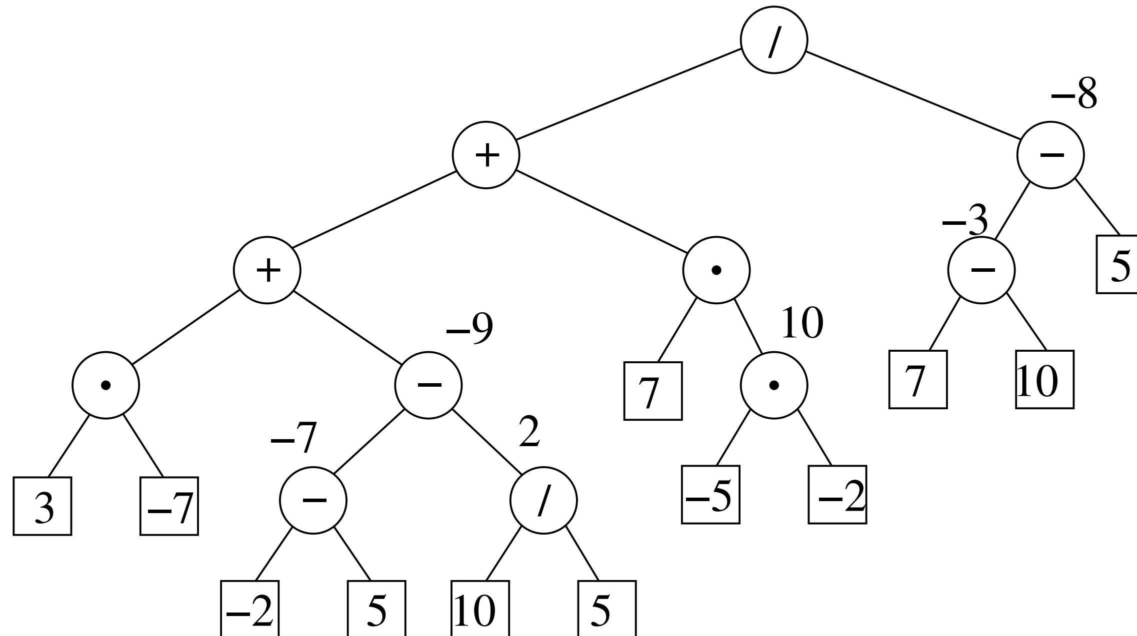


Auswertung:

1) Ordne den Blättern konkrete Werte zu,

z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

2) Werte aus, von den Blättern startend („bottom-up“).

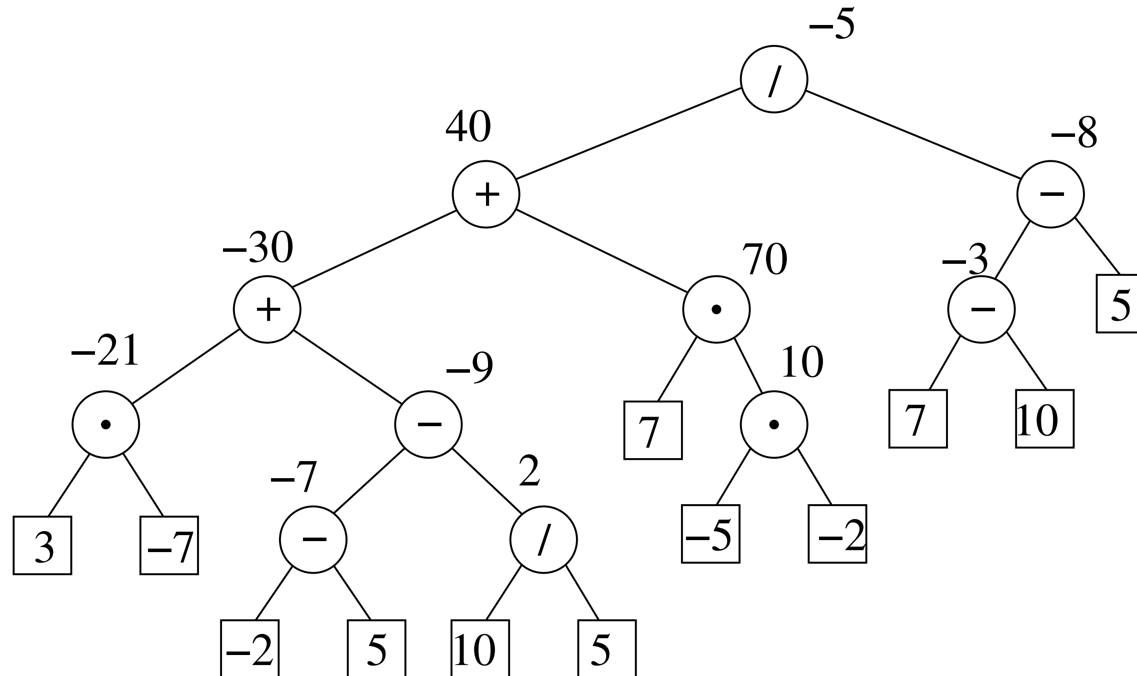


Nach sechs Schritten.

Auswertung:

1) Ordne den Blättern konkrete Werte zu,
z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

2) Werte aus, von den Blättern startend („bottom-up“).



Fertig. Resultat: -5, an der Wurzel abzulesen.

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:
FEIGE hat Codierung

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:
FEIGE hat Codierung 0011

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:

`FEIGE` hat Codierung $0011\,10$

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:
`FEIGE` hat Codierung $0011\,10\,0111$

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:

FEIGE hat Codierung $0011\,10\,0111\,010$

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:
`FEIGE` hat Codierung $0011\,10\,0111\,010\,10$.

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:
FEIGE hat Codierung 001110011101010.

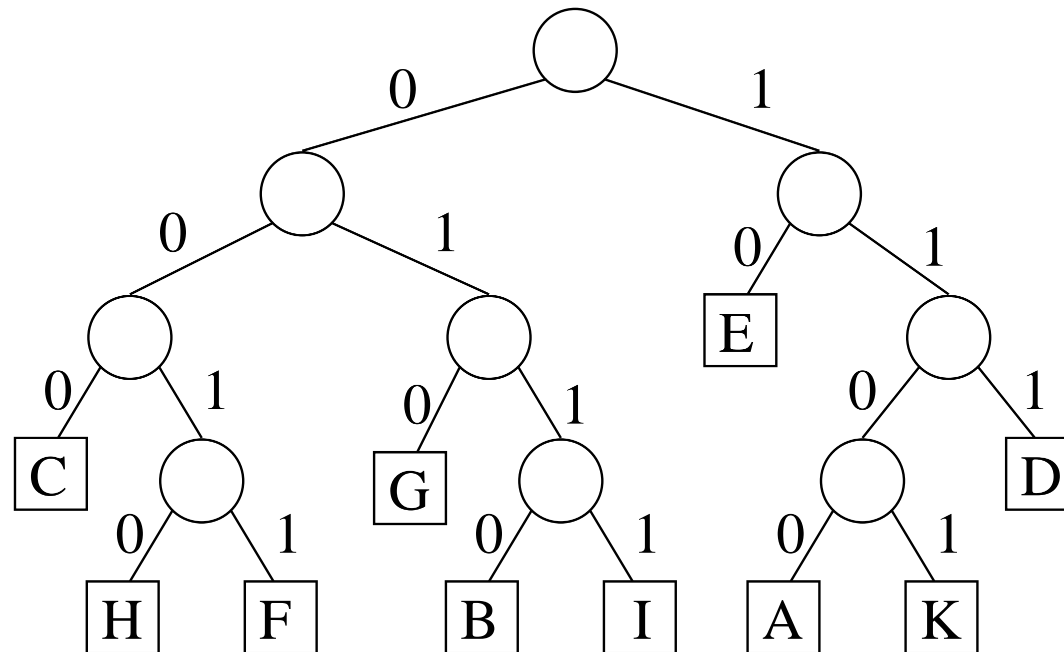
„Präfixfreier“ Code: Kein Codewort ist Anfangsstück eines anderen.
Damit: „eindeutig decodierbar“; Fuge zwischen Codes für Buchstaben muss nicht markiert werden: Schreibe $00111001110101 0$ statt $0011\,10\,0111\,010\,10$.

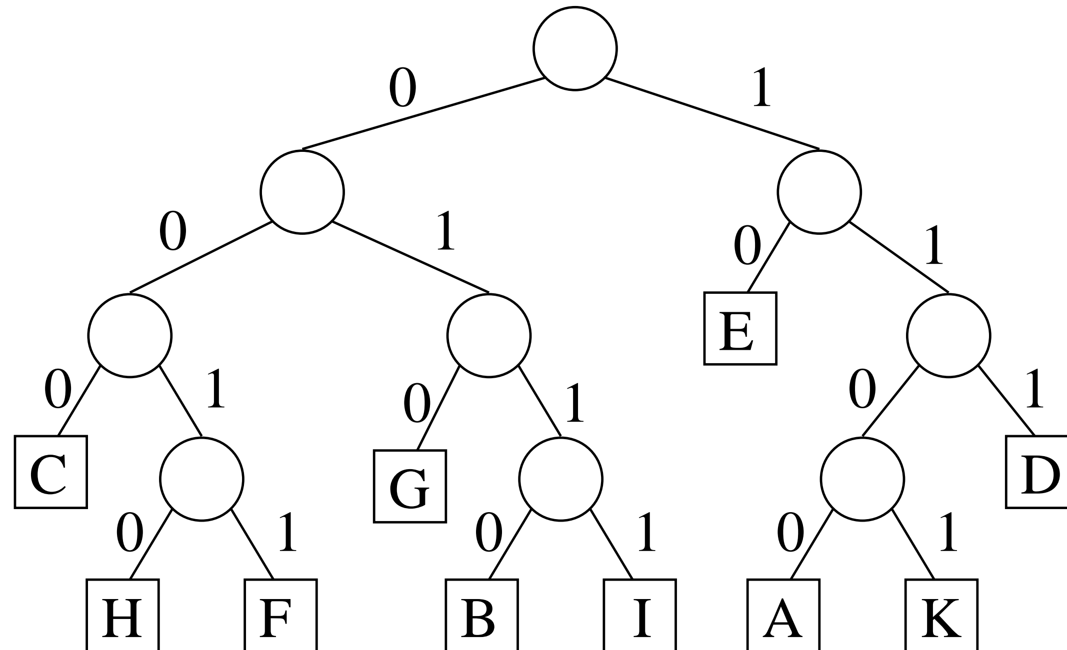
„Präfixfreier“ Code: Kein Codewort ist Anfangsstück eines anderen.
Damit: „eindeutig decodierbar“; Fuge zwischen Codes für Buchstaben muss nicht markiert werden: Schreibe $00111001110101 0$ statt $0011\,10\,0111\,010\,10$.

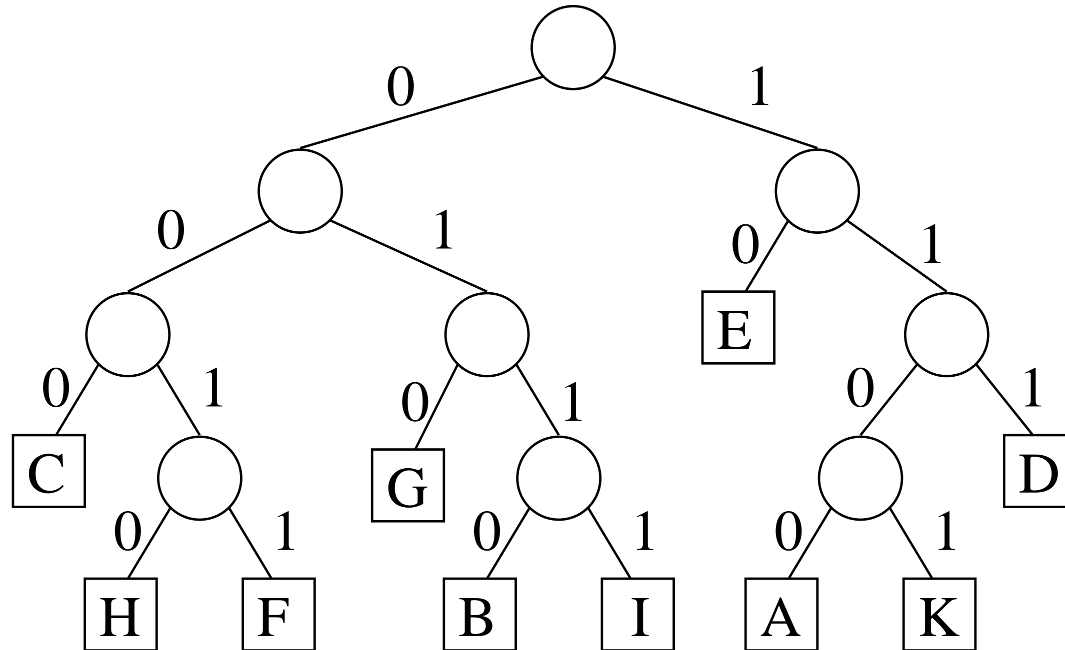
Kompakte Repräsentation des Codes als Binärbaum:

„Präfixfreier“ Code: Kein Codewort ist Anfangsstück eines anderen.
Damit: „eindeutig decodierbar“; Fuge zwischen Codes für Buchstaben muss nicht markiert werden: Schreibe 0011100111101010 statt $0011\,10\,0111\,010\,10$.

Kompakte Repräsentation des Codes als Binärbaum:



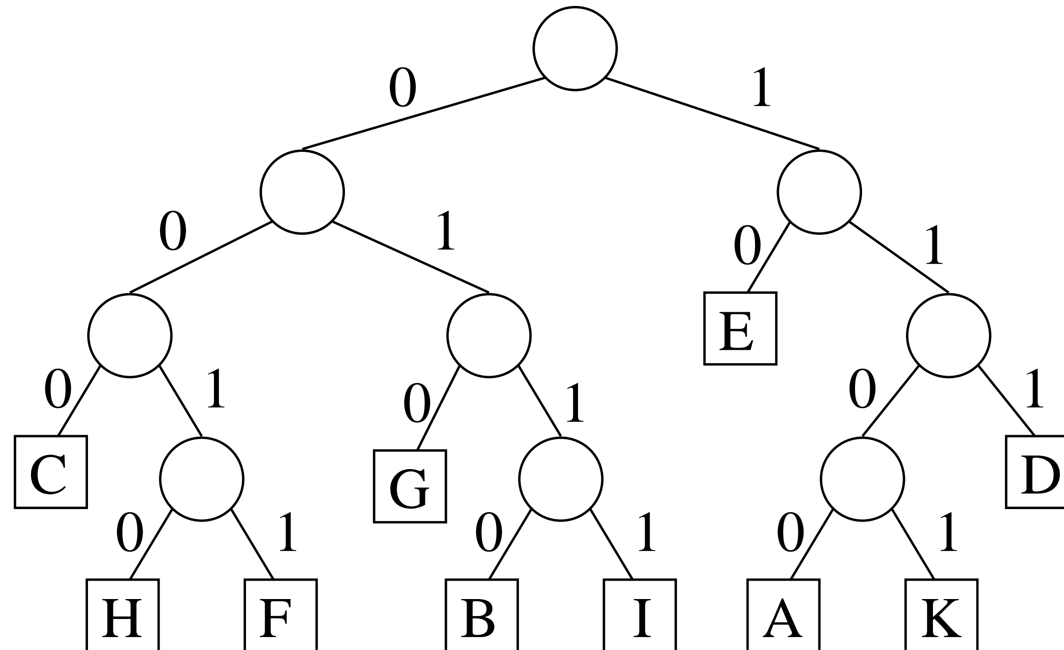

0
1
0
1
0
1
E
0
1
0
1
0
1
C
G
D
0
1
0
1
0
1
H
F
B
I
A
K



Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

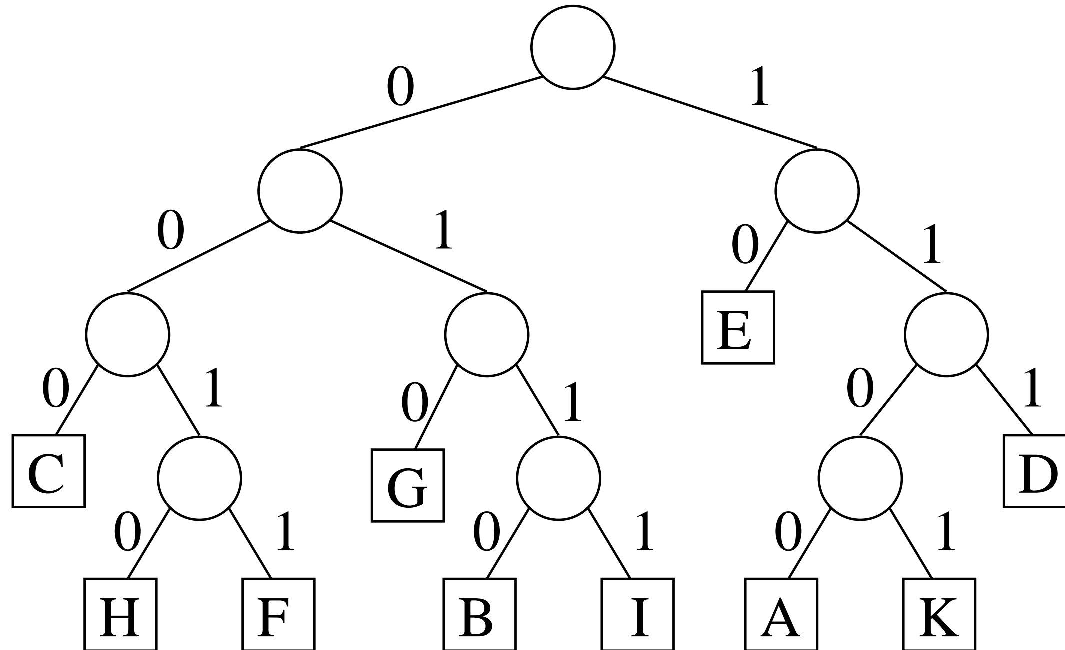
Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.



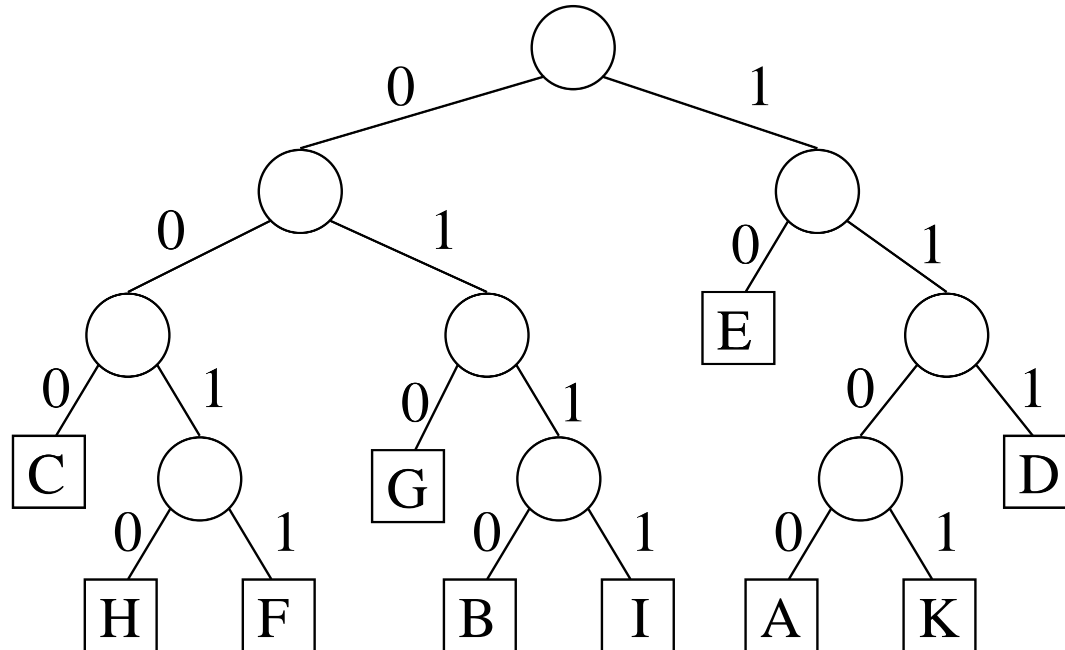
Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.



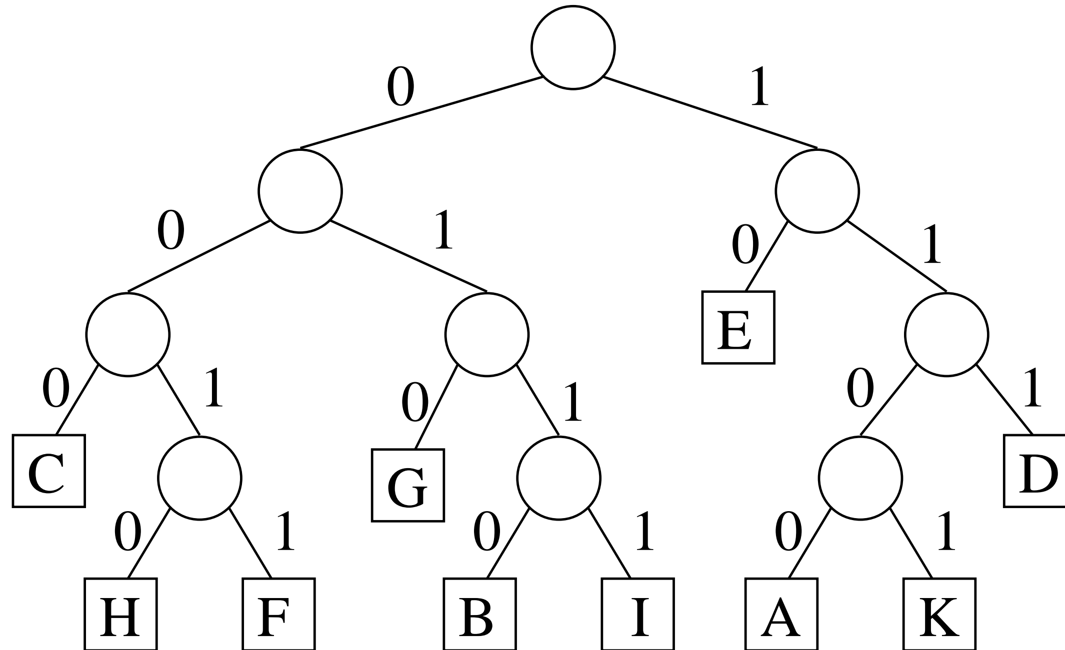
Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: 000001010011 liefert



Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: 000010100011 liefert „C

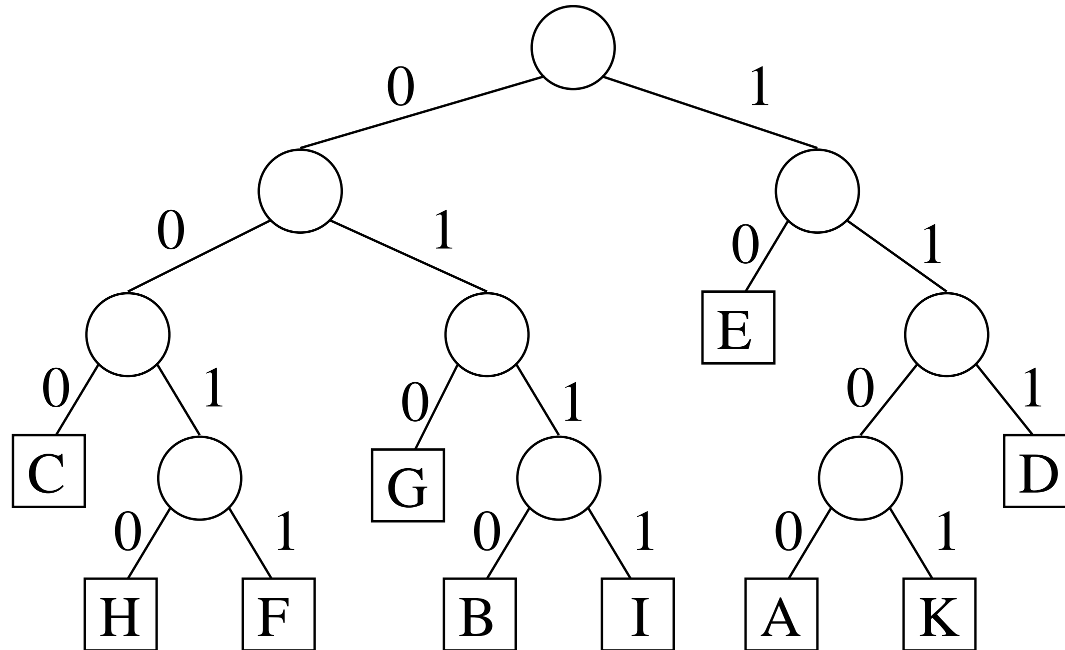


Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: 0000010100011 liefert „CH

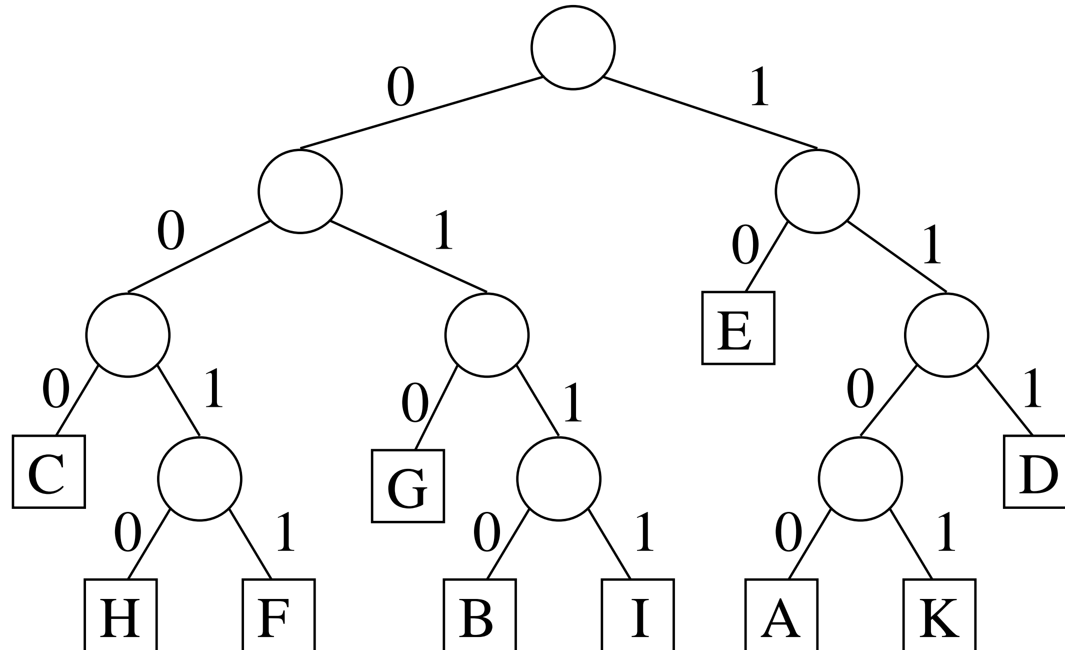
Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: $000001010 0011$ liefert „CHE



Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: 0000010100011 liefert „`CHEF`

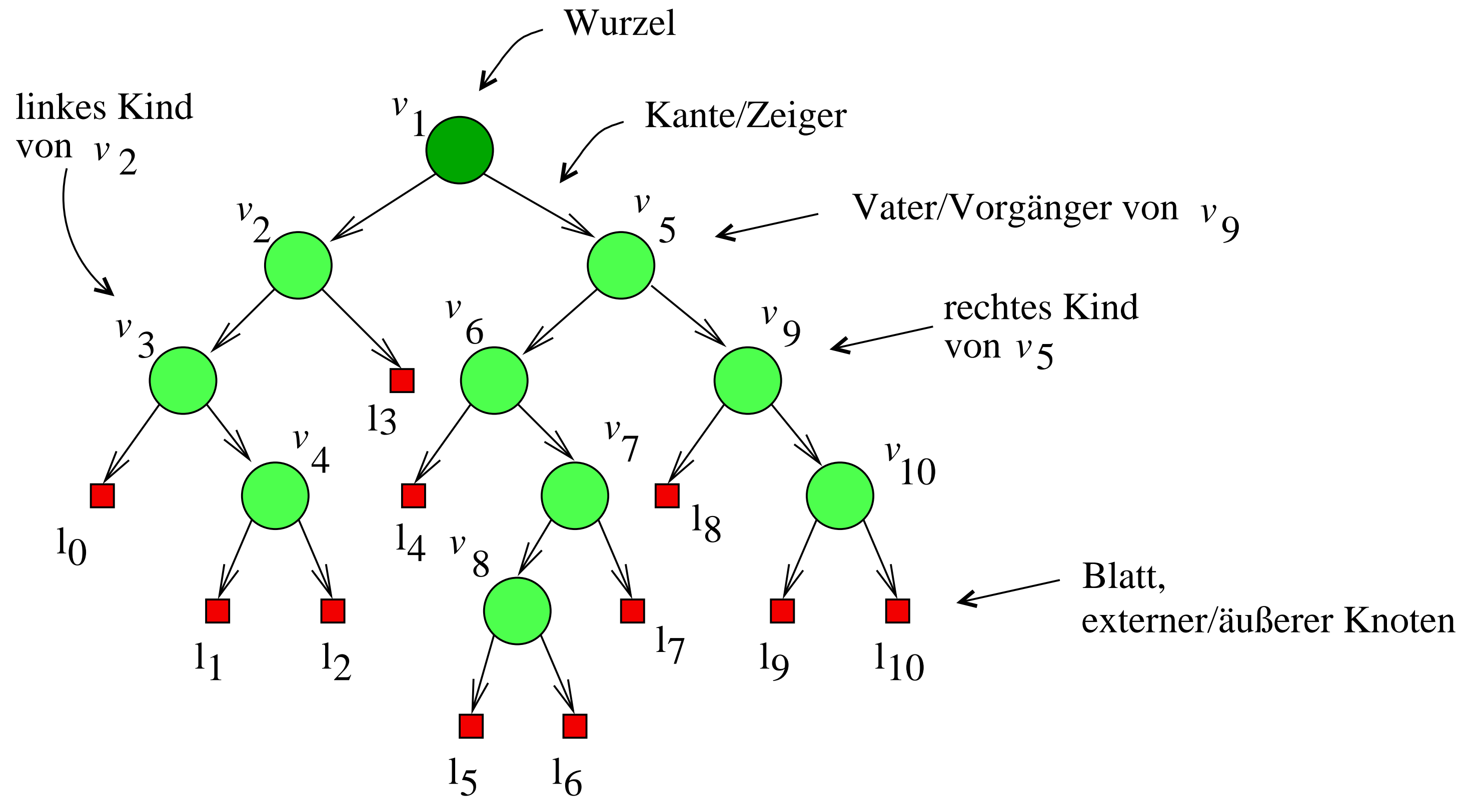


Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: 000001010011 liefert „CHEF“.

Teil 2: Definitionen

Binärbaum-Terminologie: Überblick



Definition von Binärbäumen: „flach“

Definition 3.1.1

Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht

Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht aus einer endlichen Menge V von „inneren“ Knoten,

Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht aus einer endlichen Menge V von „inneren“ Knoten, einer dazu disjunkten endlichen Menge L von „äußeren“ Knoten

Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht aus einer endlichen Menge V von „inneren“ Knoten, einer dazu disjunkten endlichen Menge L von „äußeren“ Knoten sowie einer **injektiven** Funktion

$$\textit{child}\colon V \times \{\mathit{left}, \mathit{right}\} \to V \cup L,$$

wobei Folgendes gilt:

Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht aus einer endlichen Menge V von „inneren“ Knoten, einer dazu disjunkten endlichen Menge L von „äußeren“ Knoten sowie einer **injektiven** Funktion

$$\mathit{child}\colon V \times \{\mathit{left}, \mathit{right}\} \to V \cup L,$$

wobei Folgendes gilt:

(i) In $V \cup L$ gibt es genau einen Knoten r, der nicht als Wert $\mathit{child}(v, \mathit{left})$ oder $\mathit{child}(v, \mathit{right})$ vorkommt.

Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht aus einer endlichen Menge V von „inneren“ Knoten, einer dazu disjunkten endlichen Menge L von „äußeren“ Knoten sowie einer **injektiven** Funktion

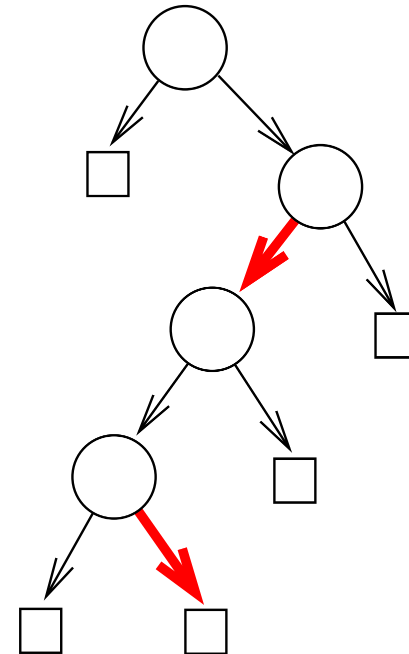
$$\mathit{child}\colon V \times \{\mathit{left}, \mathit{right}\} \to V \cup L,$$

wobei Folgendes gilt:

(i) In $V \cup L$ gibt es genau einen Knoten r, der nicht als Wert $\mathit{child}(v, \mathit{left})$ oder $\mathit{child}(v, \mathit{right})$ vorkommt. Dieser Knoten r heißt die **Wurzel**.

Definition von Binärbäumen: „flach“

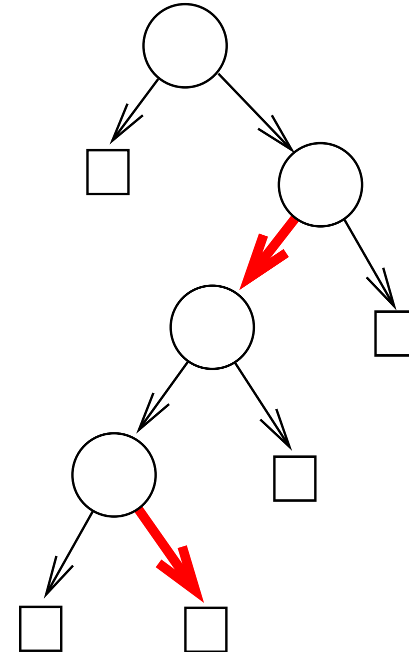
child(v, left) [*child*(v, right)] heißt
das **linke** [**rechte**] **Kind** von $v \in V$.



Definition von Binärbäumen: „flach“

$\mathit{child}(v, \mathit{left})$ $[\mathit{child}(v, \mathit{right})]$ heißt
das **linke** [**rechte**] **Kind** von $v \in V$.

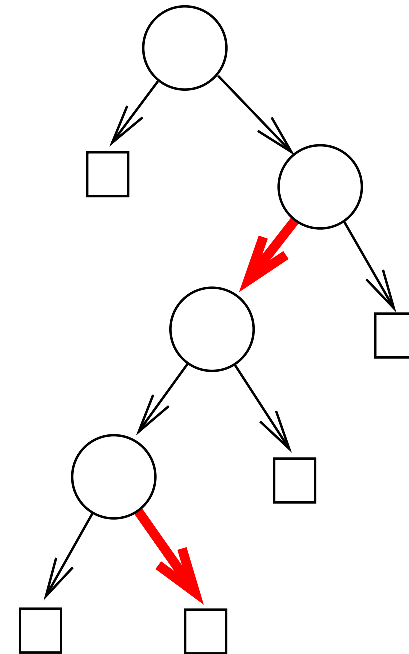
“*child* injektiv” heißt:



Definition von Binärbäumen: „flach“

child(v, left) [*child*(v, right)] heißt
das **linke** [**rechte**] **Kind** von $v \in V$.

“*child* injektiv” heißt:
Für jeden Knoten $w \in V \cup L$ mit $w \neq r$ gibt es
genau einen Knoten $u \in V$ mit
$w =$ *child*(u, left) oder $w =$ *child*(u, right)
(aber nicht beides).

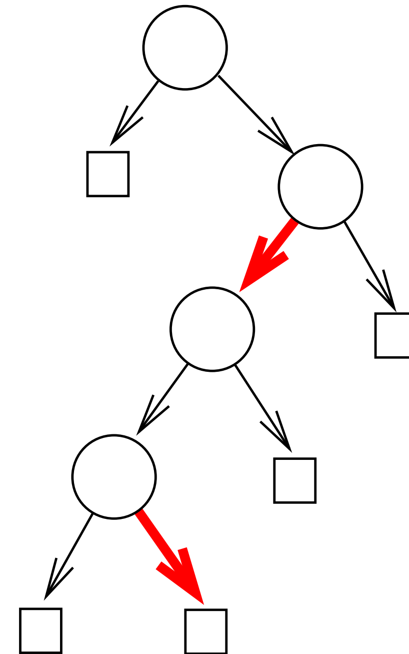


Definition von Binärbäumen: „flach“

child$(v, left)$ [*child*$(v, right)$] heißt das **linke** [**rechte**] **Kind** von $v \in V$.

“*child* injektiv” heißt:
Für jeden Knoten $w \in V \cup L$ mit $w \neq r$ gibt es **genau einen** Knoten $u \in V$ mit
$w =$ *child*$(u, left)$ oder $w =$ *child*$(u, right)$
(aber nicht beides).

Dieses u heißt $\boldsymbol{p(w)}$, der **Vorgänger**knoten oder **Vater**knoten von w.



Definition von Binärbäumen: „flach“

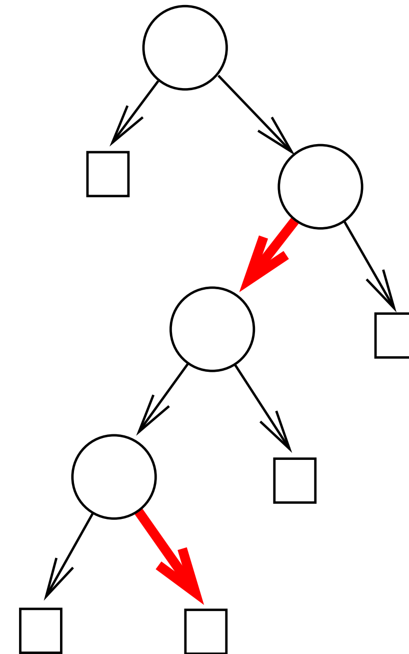
child(v, left) [*child*(v, right)] heißt das **linke** [**rechte**] **Kind** von $v \in V$.

“*child* injektiv” heißt:
Für jeden Knoten $w \in V \cup L$ mit $w \neq r$ gibt es **genau einen** Knoten $u \in V$ mit
$w =$ *child*(u, left) oder $w =$ *child*(u, right)
(aber nicht beides).

Dieses u heißt $\boldsymbol{p(w)}$, der **Vorgänger**knoten oder **Vater**knoten von w.

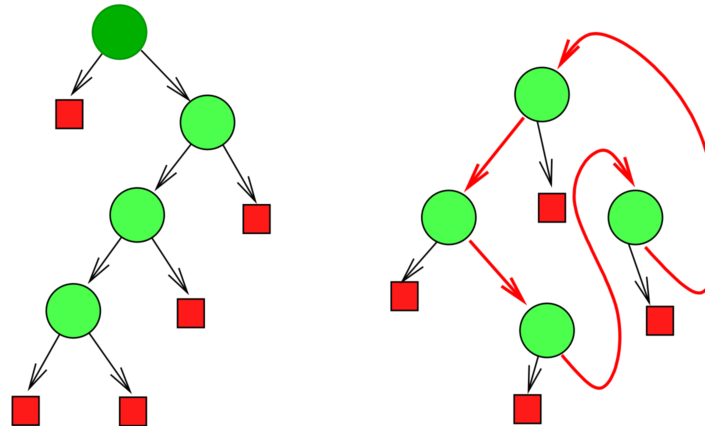
Wir sagen:

Von $p(w)$ nach w verläuft eine **Kante**.



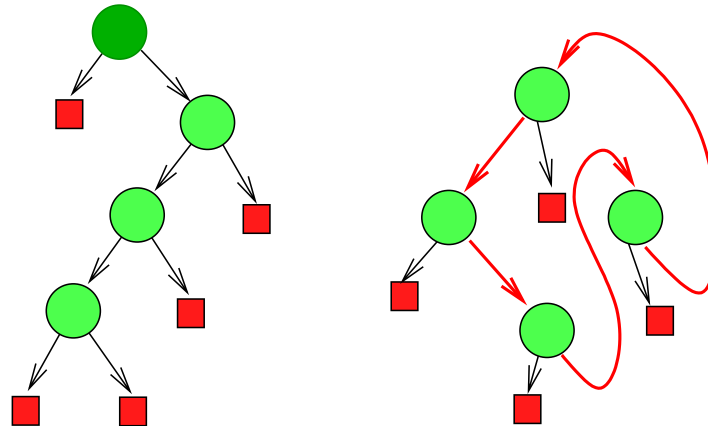
Definition von Binärbäumen: „flach“

Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



Definition von Binärbäumen: „flach“

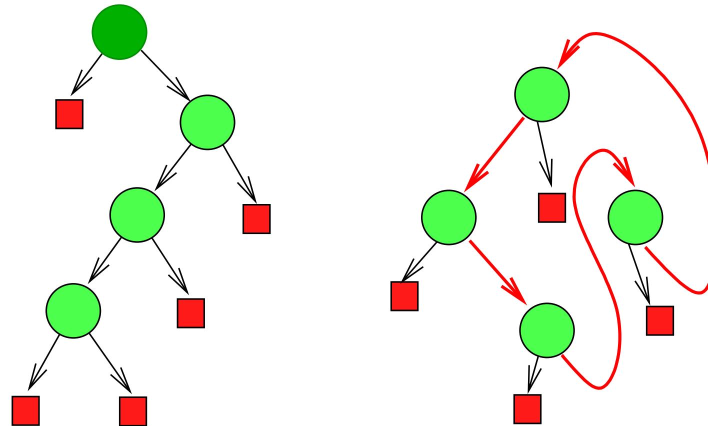
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:

Definition von Binärbäumen: „flach“

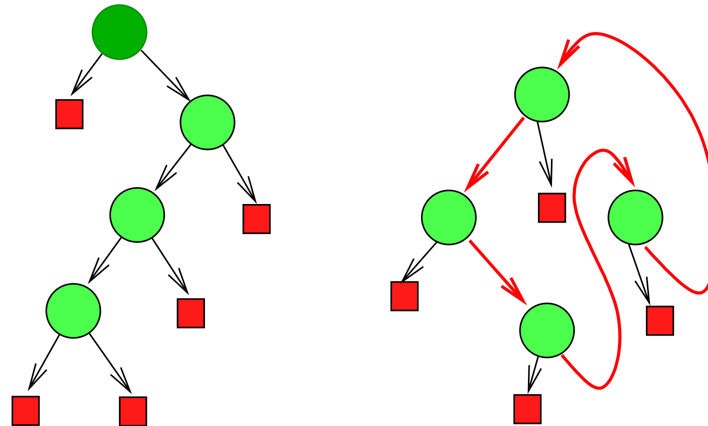
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w$,

Definition von Binärbäumen: „flach“

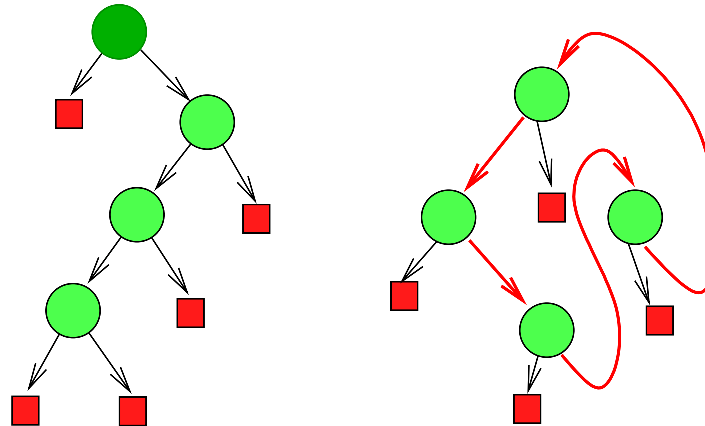
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w),$

Definition von Binärbäumen: „flach“

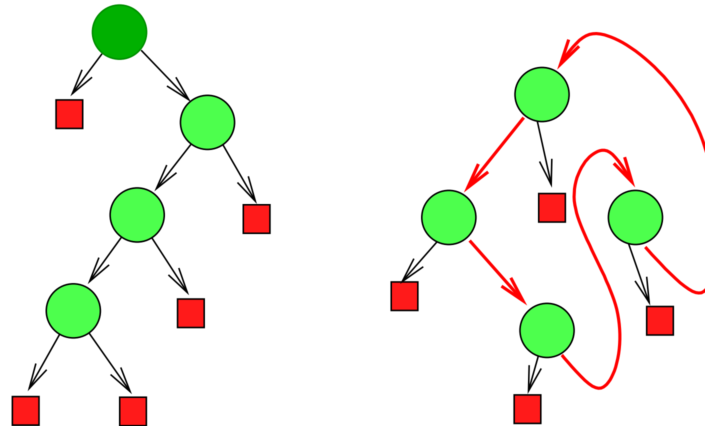
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w), v_2 = p(v_1),$

Definition von Binärbäumen: „flach“

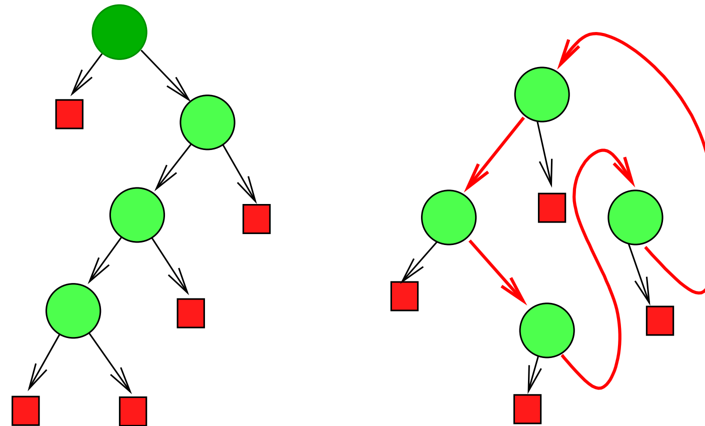
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) „**Kreisfreiheit**“: Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w), v_2 = p(v_1), v_3 = p(v_2),$

Definition von Binärbäumen: „flach“

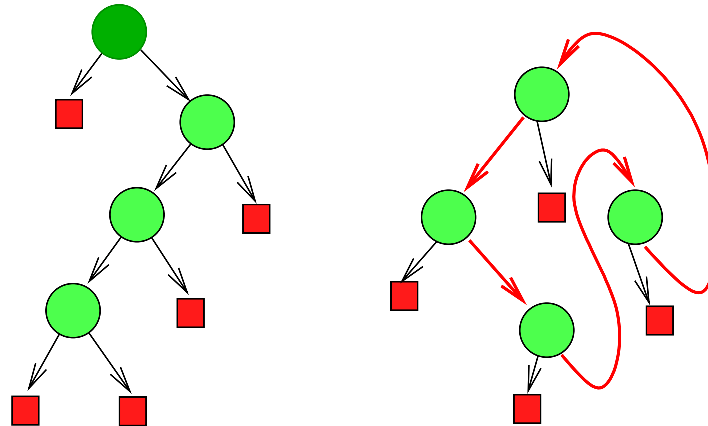
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w), v_2 = p(v_1), v_3 = p(v_2), \ldots$

Definition von Binärbäumen: „flach“

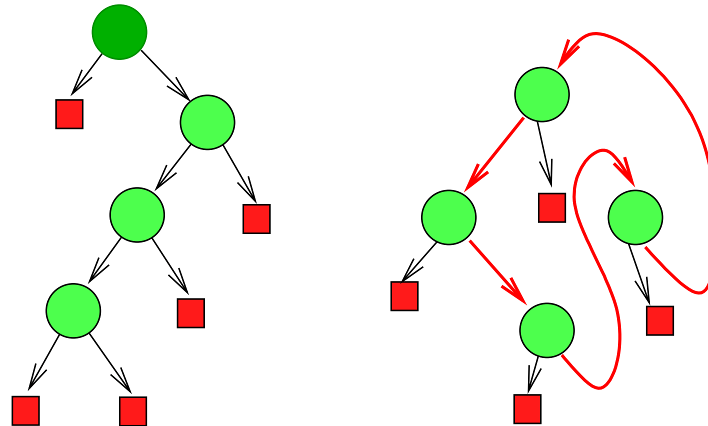
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w), v_2 = p(v_1), v_3 = p(v_2), \ldots$ bricht nach endlich vielen Schritten mit einem $v_d = r$ ab.

Definition von Binärbäumen: „flach“

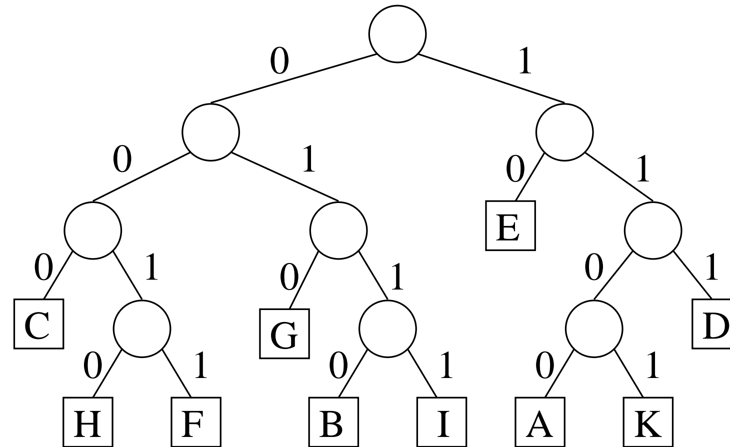
Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



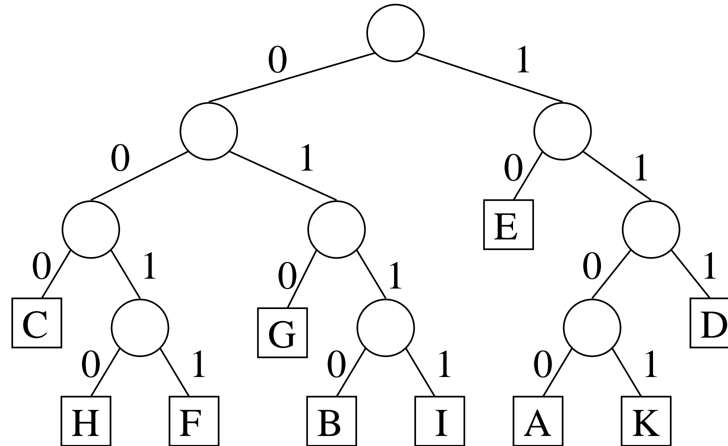
(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w), v_2 = p(v_1), v_3 = p(v_2), \ldots$ bricht nach endlich vielen Schritten mit einem $v_d = r$ ab.

(Dadurch ist ein **eindeutiger Weg** von w zur Wurzel festgelegt. Also ist jeder Knoten des Binärbaums von der Wurzel aus entlang einer Kantenfolge („Weg“) erreichbar.)

Definition von Binärbäumen: „flach“

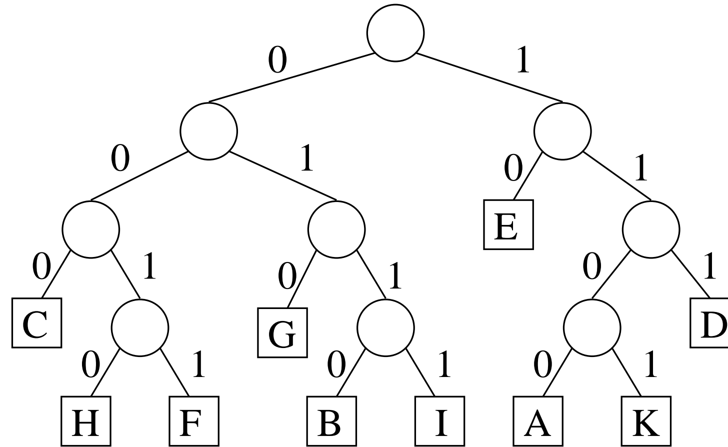


Definition von Binärbäumen: „flach“



Markierungen (oder **Beschriftungen**) von Knoten und Kanten werden als Funktionen dargestellt:

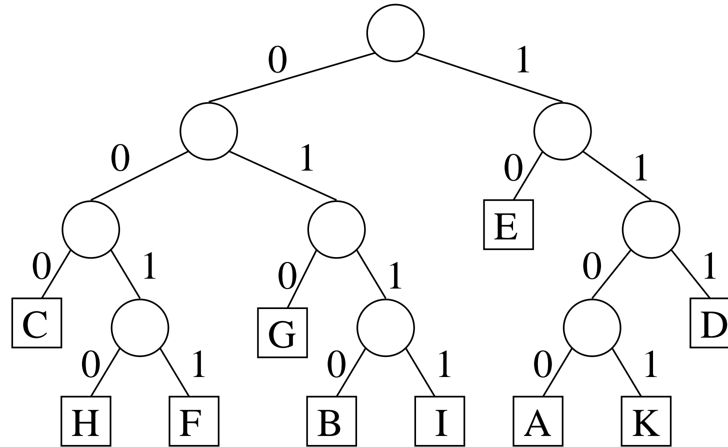
Definition von Binärbäumen: „flach“



Markierungen (oder **Beschriftungen**) von Knoten und Kanten werden als Funktionen dargestellt: (D, Z, X sind beliebige Mengen.)

Knotenmarkierungen: Funktionen $m_V : V \to D$ und $m_L : L \to Z$.

Definition von Binärbäumen: „flach“

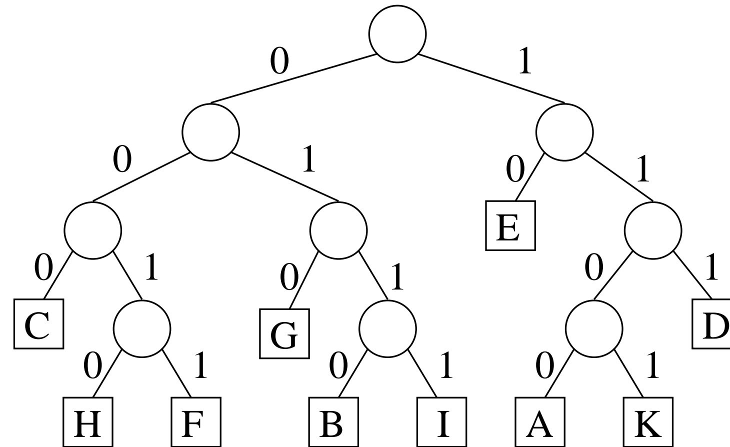


Markierungen (oder **Beschriftungen**) von Knoten und Kanten werden als Funktionen dargestellt: (D, Z, X sind beliebige Mengen.)

Knotenmarkierungen: Funktionen $m_V : V \to D$ und $m_L : L \to Z$.

Kantenmarkierungen: Funktion $m_K : V \times \{\mathit{left}, \mathit{right}\} \to X$.

Definition von Binärbäumen: „flach“



Markierungen (oder **Beschriftungen**) von Knoten und Kanten werden als Funktionen dargestellt: (D, Z, X sind beliebige Mengen.)

Knotenmarkierungen: Funktionen $m_V : V \to D$ und $m_L : L \to Z$.

Kantenmarkierungen: Funktion $m_K : V \times \{\mathit{left}, \mathit{right}\} \to X$.

Beispiel: In einem Codierungsbaum sind die Blätter mit Buchstaben markiert, die beiden Kanten, die aus einem inneren Knoten herausführen, mit 0 und 1.

Definition von Binärbäumen: rekursiv

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist, dann ist auch (T_1, x, T_2) ein (D, Z)-Binärbaum.

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist,
dann ist auch (T_1, x, T_2) ein (D, Z)-Binärbaum.

// Wurzel mit Markierung x, linker Unterbaum T_1, rechter Unterbaum T_2.

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist,
dann ist auch (T_1, x, T_2) ein (D, Z)-Binärbaum.

// Wurzel mit Markierung x, linker Unterbaum T_1, rechter Unterbaum T_2.

(iii)* Nichts sonst ist (D, Z)-Binärbaum.

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist,
dann ist auch (T_1, x, T_2) ein (D, Z)-Binärbaum.

// Wurzel mit Markierung x, linker Unterbaum T_1, rechter Unterbaum T_2.

(iii)* Nichts sonst ist (D, Z)-Binärbaum.

Definition ist äquivalent zur “flachen” Auffassung.

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist,
dann ist auch (T_1, x, T_2) ein (D, Z)-Binärbaum.

// Wurzel mit Markierung x, linker Unterbaum T_1, rechter Unterbaum T_2.

(iii)* Nichts sonst ist (D, Z)-Binärbaum.

Definition ist äquivalent zur "flachen" Auffassung.

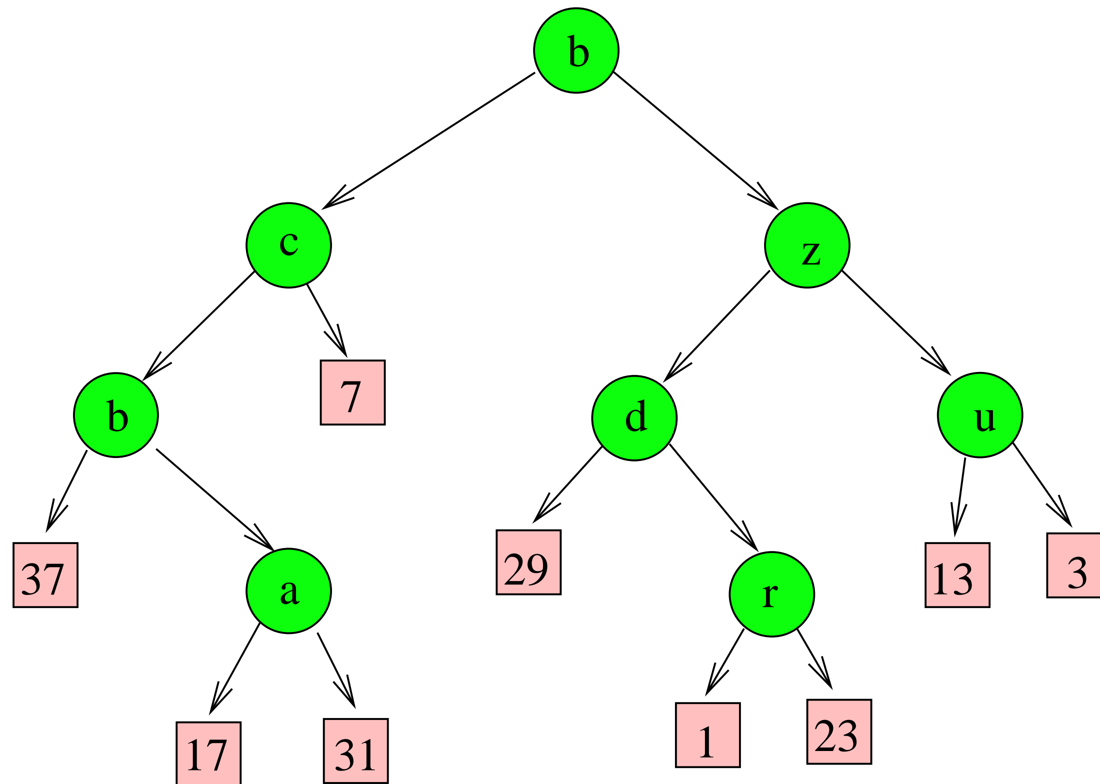
---

*(iii) wird bei induktiven Definitionen meist weggelassen.

Beispiel: $D = \{a, \ldots, z\}$ und $Z = \mathbb{N}$. – Tupeldarstellung (rekursiv):

$(((37, b, (17, a, 31)), c, 7), b, ((29, d, (1, r, 23)), z, (13, u, 3)))$

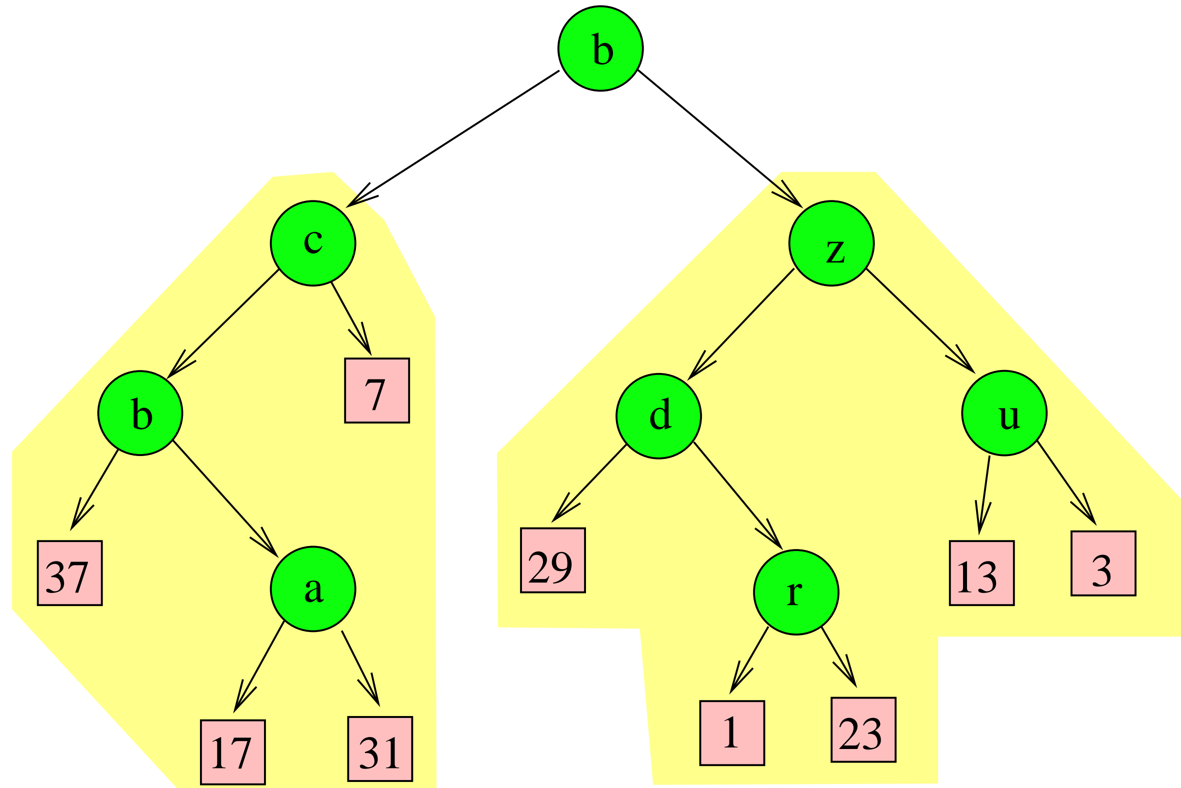
Graphische Darstellung („flach“):



Beispiel: $D = \{\mathrm{a}, \ldots, \mathrm{z}\}$ und $Z = \mathbb{N}$. – Tupeldarstellung (rekursiv):

$(((37, \mathrm{b}, (17, \mathrm{a}, 31)), \mathrm{c}, 7), \mathrm{b}, ((29, \mathrm{d}, (1, \mathrm{r}, 23)), \mathrm{z}, (13, \mathrm{u}, 3)))$

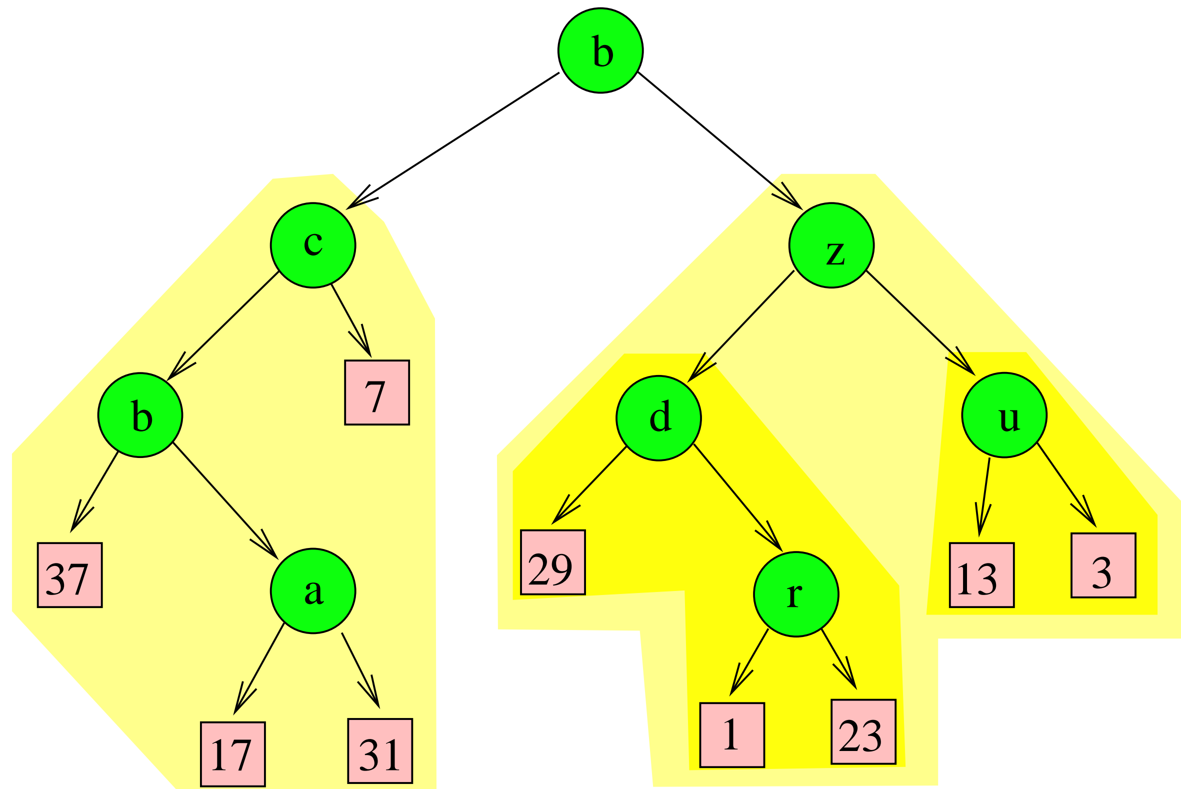
Graphische Darstellung („flach“):



Beispiel: $D = \{\mathrm{a}, \ldots, \mathrm{z}\}$ und $Z = \mathbb{N}$. – Tupeldarstellung (rekursiv):

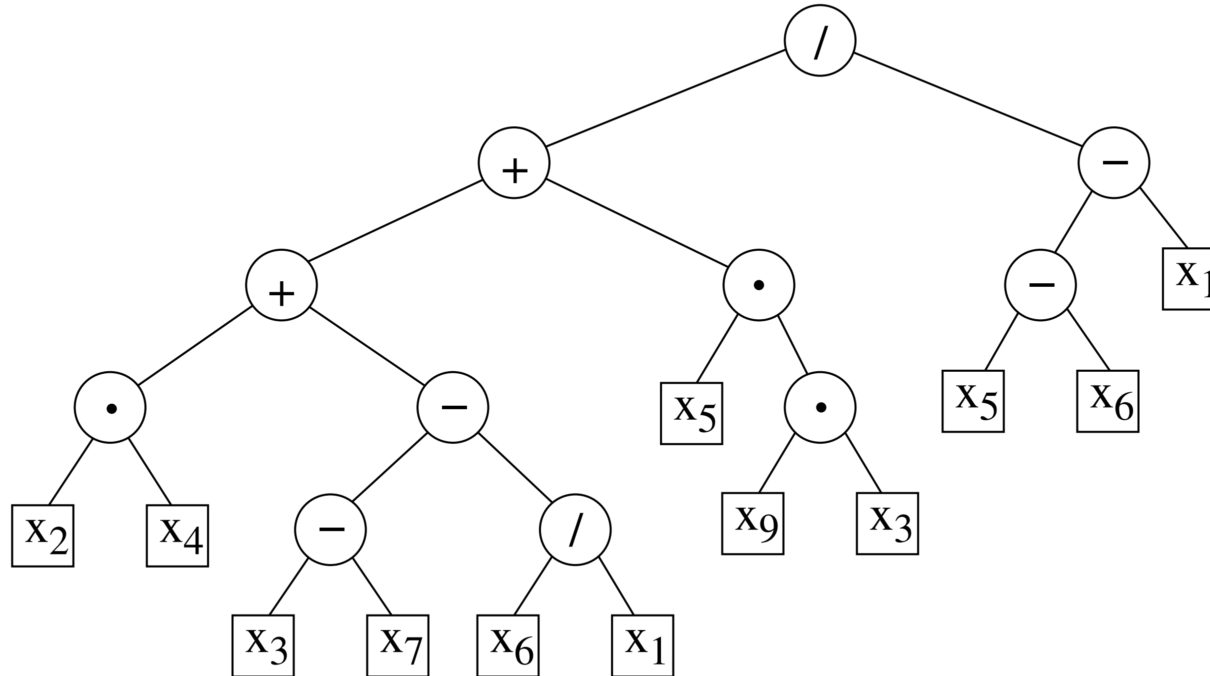
$(((37, \mathrm{b}, (17, \mathrm{a}, 31)), \mathrm{c}, 7), \mathrm{b}, ((29, \mathrm{d}, (1, \mathrm{r}, 23)), \mathrm{z}, (13, \mathrm{u}, 3)))$

Graphische Darstellung („flach“):



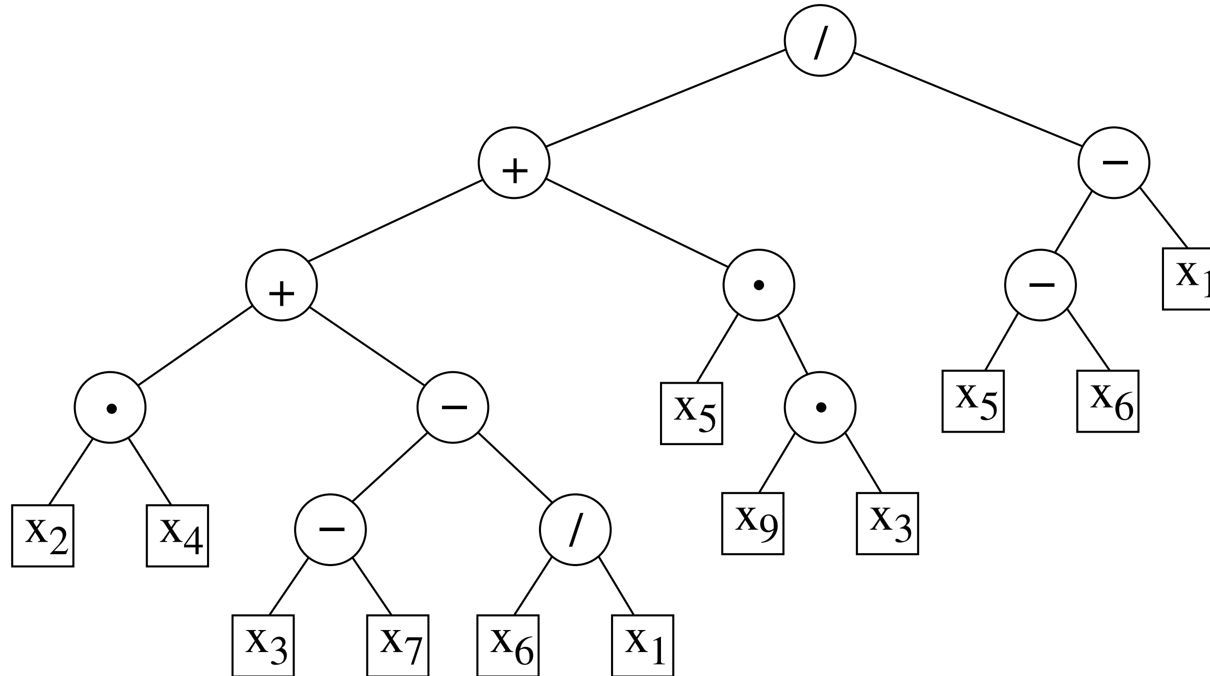
Beispiel: **Arithmetischer Ausdruck.** $D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.

Beispiel: **Arithmetischer Ausdruck.** $D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



$$((((x_2 \cdot x_4) + ((x_3 - x_7) - (x_6/x_1))) + (x_5 \cdot (x_9 \cdot x_3)))/((x_5 - x_6) - x_1))$$

Beispiel: **Arithmetischer Ausdruck.** $D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



$$((((x_2 \cdot x_4) + ((x_3 - x_7) - (x_6/x_1))) + (x_5 \cdot (x_9 \cdot x_3)))/((x_5 - x_6) - x_1))$$

Der Ausdruck ist genau die (rekursive) Tupeldarstellung, nur ohne Kommas!

Teil 3: Operationen

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten:

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*

Bintrees

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen:

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*

Bintrees

Boolean

Operationen: $\textit{empty}\colon \to \textit{Bintrees}$

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen: $\textit{empty}\colon \to \textit{Bintrees}$
$\textit{buildTree}\colon \textit{Elements} \times \textit{Bintrees} \times \textit{Bintrees} \to \textit{Bintrees}$

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen: $empty\colon \to Bintrees$
$buildTree\colon Elements \times Bintrees \times Bintrees \to Bintrees$
$leftSubtree\colon Bintrees \to Bintrees$

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen:
$\mathit{empty}\colon \to \mathit{Bintrees}$
$\mathit{buildTree}\colon \mathit{Elements} \times \mathit{Bintrees} \times \mathit{Bintrees} \to \mathit{Bintrees}$
$\mathit{leftSubtree}\colon \mathit{Bintrees} \to \mathit{Bintrees}$
$\mathit{rightSubtree}\colon \mathit{Bintrees} \to \mathit{Bintrees}$

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen:
$\textit{empty}\colon \to \textit{Bintrees}$
$\textit{buildTree}\colon \textit{Elements} \times \textit{Bintrees} \times \textit{Bintrees} \to \textit{Bintrees}$
$\textit{leftSubtree}\colon \textit{Bintrees} \to \textit{Bintrees}$
$\textit{rightSubtree}\colon \textit{Bintrees} \to \textit{Bintrees}$
$\textit{data}\colon \textit{Bintrees} \to \textit{Elements}$

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen:
$\mathit{empty}\colon \to \mathit{Bintrees}$
$\mathit{buildTree}\colon \mathit{Elements} \times \mathit{Bintrees} \times \mathit{Bintrees} \to \mathit{Bintrees}$
$\mathit{leftSubtree}\colon \mathit{Bintrees} \to \mathit{Bintrees}$
$\mathit{rightSubtree}\colon \mathit{Bintrees} \to \mathit{Bintrees}$
$\mathit{data}\colon \mathit{Bintrees} \to \mathit{Elements}$
$\mathit{isempty}\colon \mathit{Bintrees} \to \mathit{Boolean}$

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen:
$empty\colon \rightarrow Bintrees$
$buildTree\colon Elements \times Bintrees \times Bintrees \rightarrow Bintrees$
$leftSubtree\colon Bintrees \rightarrow Bintrees$
$rightSubtree\colon Bintrees \rightarrow Bintrees$
$data\colon Bintrees \rightarrow Elements$
$isempty\colon Bintrees \rightarrow Boolean$

2. Mathematisches Modell: (nächste Folie)

Sorten: *Elements*: (nichtleere) Menge D

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{true, false\}$

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{true, false\}$

Op'en: *empty*() $:= \Box := (-)$ // „leerer Baum“

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{true, false\}$

Op'en: *empty*() $:= \square := (-)$ // „leerer Baum“

buildTree$(x, T_1, T_2) := (T_1, x, T_2)$

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T \text{ ist } (D, \{-\})\text{-Binärbaum}\}$

Boolean: $\{true, false\}$

Op'en: *empty*() $:= \Box := (-)$ // „leerer Baum“

buildTree$(x, T_1, T_2) := (T_1, x, T_2)$

$$\textit{leftSubtree}(T) := \begin{cases} T_1, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \Box \end{cases}$$

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{true, false\}$

Op'en: *empty*() $:= \square := (-)$ // „leerer Baum“

buildTree$(x, T_1, T_2) := (T_1, x, T_2)$

$$\textit{leftSubtree}(T) := \begin{cases} T_1, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\textit{rightSubtree}(T) := \begin{cases} T_2, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{\mathit{true}, \mathit{false}\}$

Op'en: *empty*() $:= \square := (-)$ // „leerer Baum"

buildTree$(x, T_1, T_2) := (T_1, x, T_2)$

$$\mathit{leftSubtree}(T) := \begin{cases} T_1, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\mathit{rightSubtree}(T) := \begin{cases} T_2, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\mathit{data}(T) := \begin{cases} x, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{\mathit{true}, \mathit{false}\}$

Op'en: *empty*() $:= \square := (-)$ // „leerer Baum“

buildTree$(x, T_1, T_2) := (T_1, x, T_2)$

$$\textit{leftSubtree}(T) := \begin{cases} T_1, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\textit{rightSubtree}(T) := \begin{cases} T_2, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

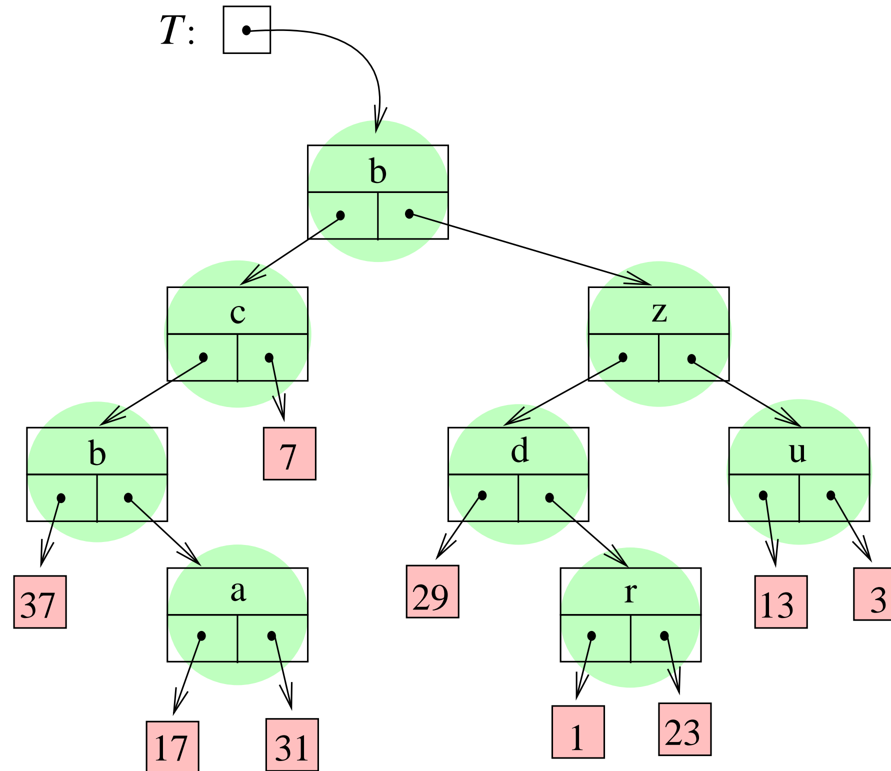
$$\textit{data}(T) := \begin{cases} x, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\textit{isempty}(T) := \begin{cases} \mathit{false}, & \text{falls } T = (T_1, x, T_2) \\ \mathit{true}, & \text{falls } T = \square \end{cases}$$

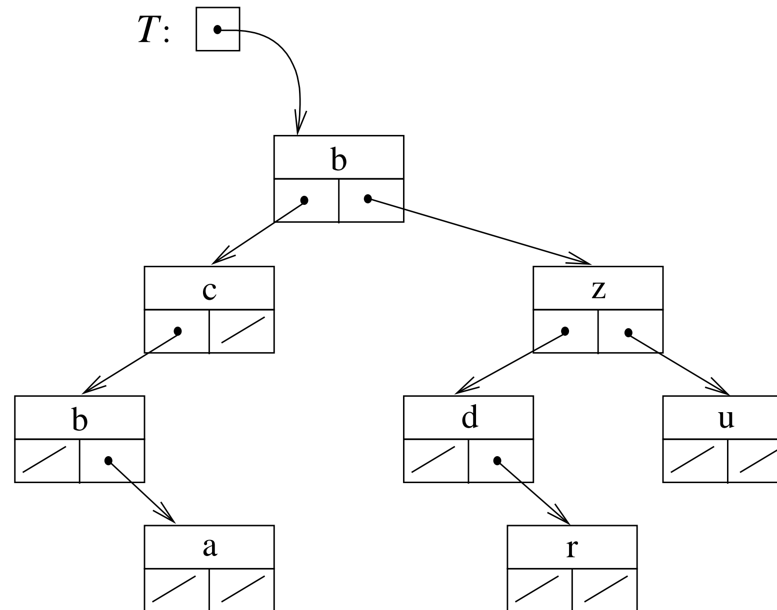
Binärbäume, Implementierung I:

Zeigerstruktur bzw. Referenzstruktur, hier mit externen Knoten, die Inhalt haben.

Für Implementierung in Java vgl. [AuP], Kap. 10 und [Saake/Sattler], Kap. 14.2.

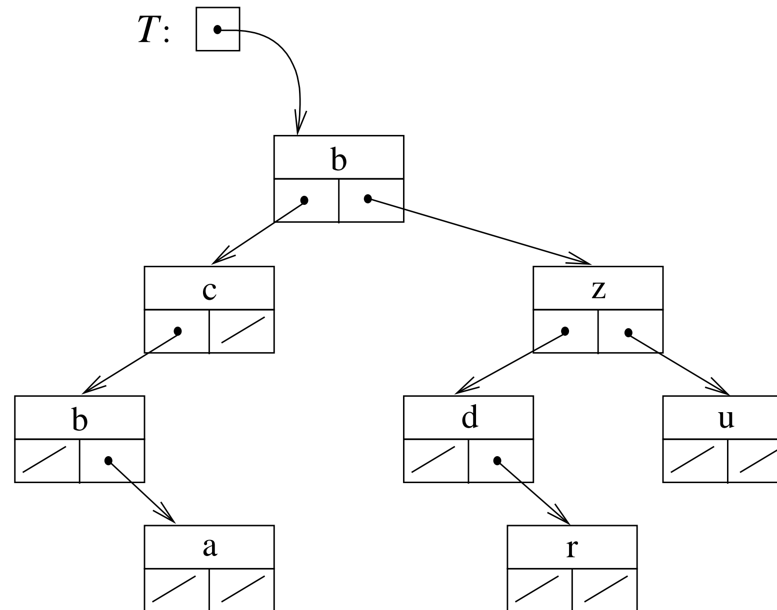


Binärbäume, Implementierung II: Leere externe Knoten



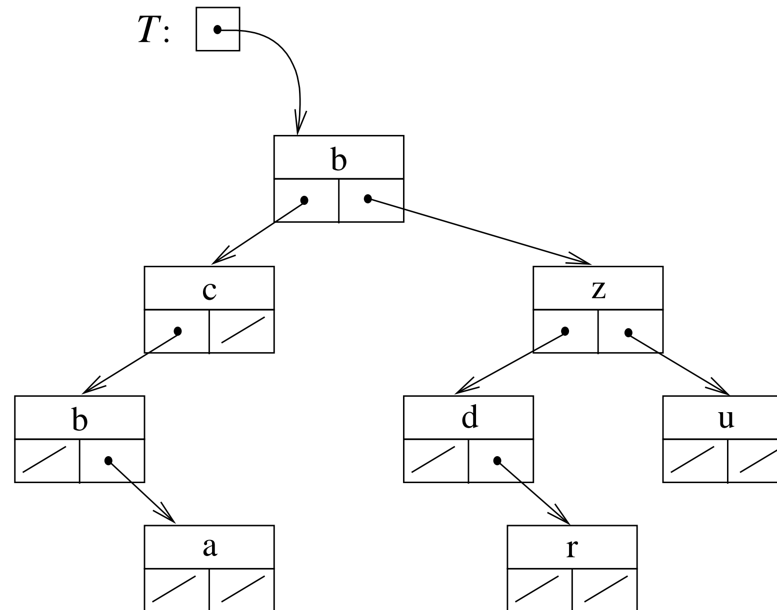
Externe Knoten werden nicht repräsentiert (`null`-Zeiger/-Referenz).

Binärbäume, Implementierung II: Leere externe Knoten



Externe Knoten werden nicht repräsentiert (`null`-Zeiger/-Referenz). – Oder: Ein externer Knoten ℓ, Referenz auf ℓ statt `null` (`nullNode` in [Saake/Sattler], Kap. 14.2.)

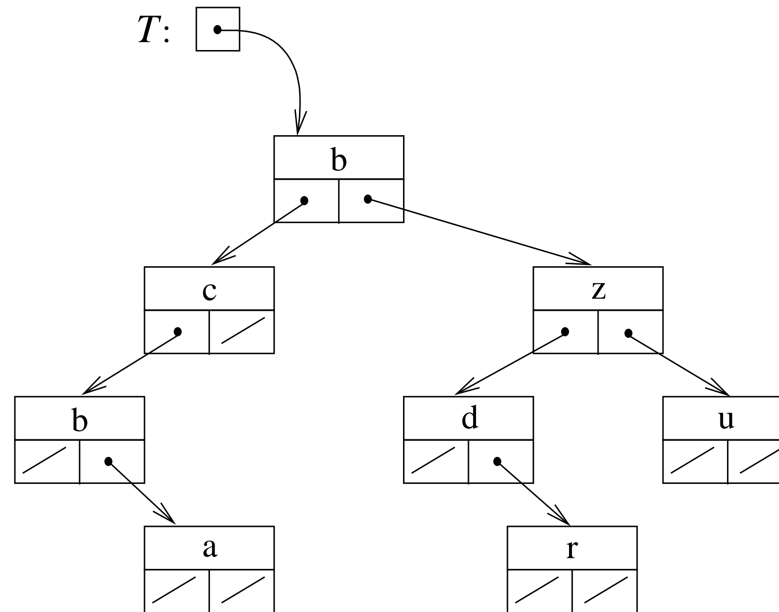
Binärbäume, Implementierung II: Leere externe Knoten



Externe Knoten werden nicht repräsentiert (`null`-Zeiger/-Referenz). – Oder: Ein externer Knoten ℓ, Referenz auf ℓ statt `null` (`nullNode` in [Saake/Sattler], Kap. 14.2.)

Speicherplatzbedarf für n Knoten:

Binärbäume, Implementierung II: Leere externe Knoten

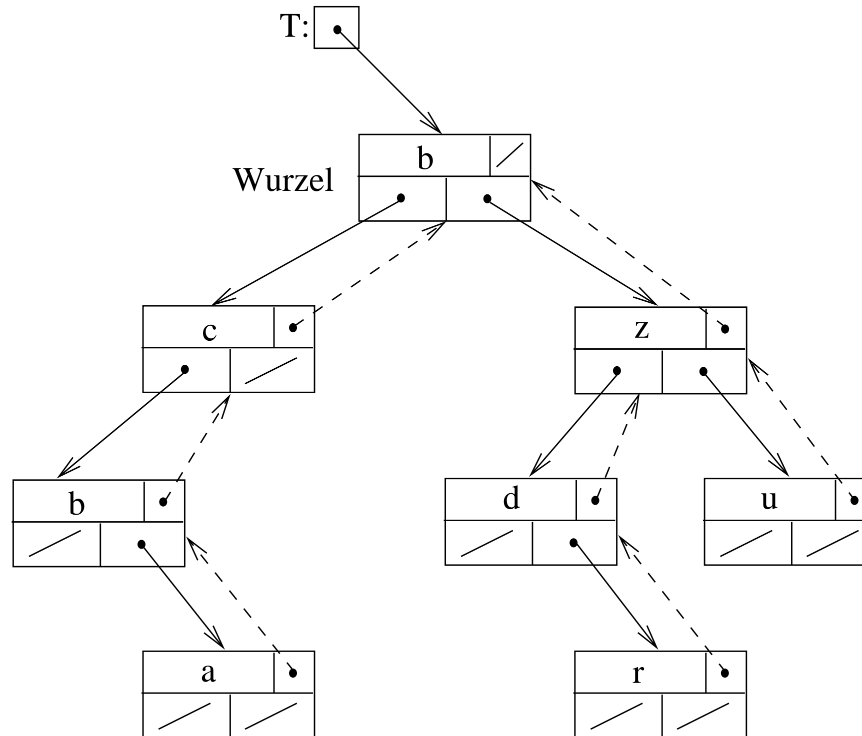


Externe Knoten werden nicht repräsentiert (`null`-Zeiger/-Referenz). – Oder: Ein externer Knoten ℓ, Referenz auf ℓ statt `null` (`nullNode` in [Saake/Sattler], Kap. 14.2.)

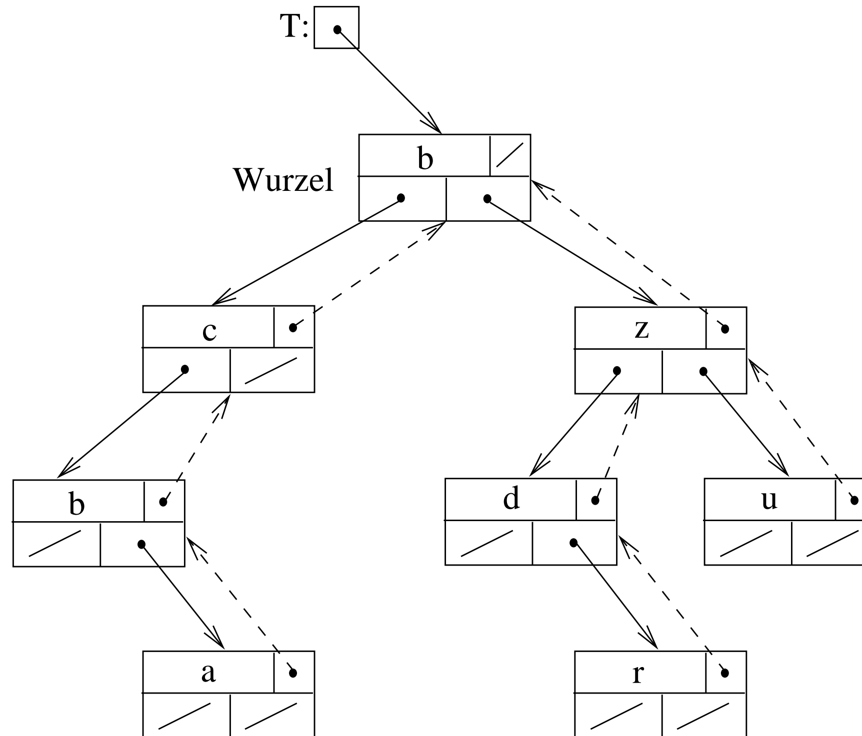
Speicherplatzbedarf für n Knoten: n Dateneinträge, $2n$ Zeiger/Referenzen, davon $n+1$ mal `null`.

Mögliche Erweiterung, konzeptuell nur in „flacher“ Auffassung: Vorgängerzeiger.

Mögliche Erweiterung, konzeptuell nur in „flacher“ Auffassung: Vorgängerzeiger.

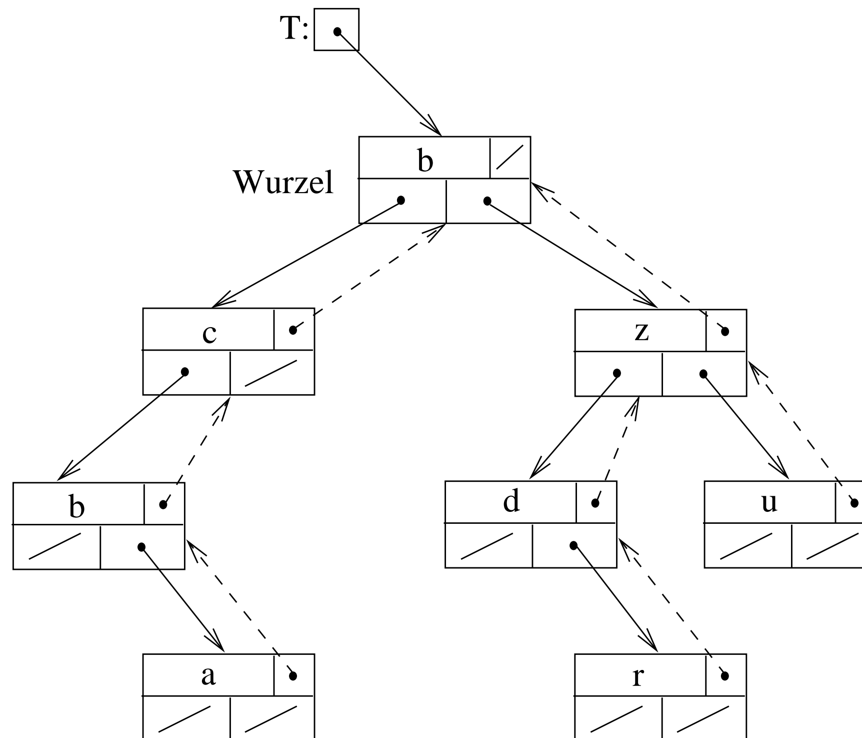


Mögliche Erweiterung, konzeptuell nur in „flacher“ Auffassung: Vorgängerzeiger.



Erleichtert Navigieren im Baum,

Mögliche Erweiterung, konzeptuell nur in „flacher“ Auffassung: Vorgängerzeiger.



Erleichtert Navigieren im Baum, besonders bei Verwenden von iterativen Verfahren.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

data(`T`): auch klar; nur definiert, wenn `T` nicht `null` ist.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

data(`T`): auch klar; nur definiert, wenn `T` nicht `null` ist.

leftSubtree(`T`) liefert Zeiger/Referenz `T.left`; nur definiert, wenn `T` nicht `null` ist.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

data(`T`): auch klar; nur definiert, wenn `T` nicht `null` ist.

leftSubtree(`T`) liefert Zeiger/Referenz `T.left`; nur definiert, wenn `T` nicht `null` ist.

rightSubtree(`T`): analog.

buildTree(x, T_1, T_2) erzeugt einen neuen Knoten (Wurzel) r mit Eintrag x, in die Kindplätze werden die Zeiger/Referenzen auf T_1 und T_2 eingetragen.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

data(`T`): auch klar; nur definiert, wenn `T` nicht `null` ist.

leftSubtree(`T`) liefert Zeiger/Referenz `T.left`; nur definiert, wenn `T` nicht `null` ist.

rightSubtree(`T`): analog.

buildTree(x, T_1, T_2) erzeugt einen neuen Knoten (Wurzel) r mit Eintrag x, in die Kindplätze werden die Zeiger/Referenzen auf T_1 und T_2 eingetragen.

Achtung: Der **Benutzer** der *buildTree*-Operation muss sicherstellen, dass die Knotenmengen von T_1 und T_2 disjunkt sind!

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

data(`T`): auch klar; nur definiert, wenn `T` nicht `null` ist.

leftSubtree(`T`) liefert Zeiger/Referenz `T.left`; nur definiert, wenn `T` nicht `null` ist.

rightSubtree(`T`): analog.

buildTree(x, T_1, T_2) erzeugt einen neuen Knoten (Wurzel) r mit Eintrag x, in die Kindplätze werden die Zeiger/Referenzen auf T_1 und T_2 eingetragen.

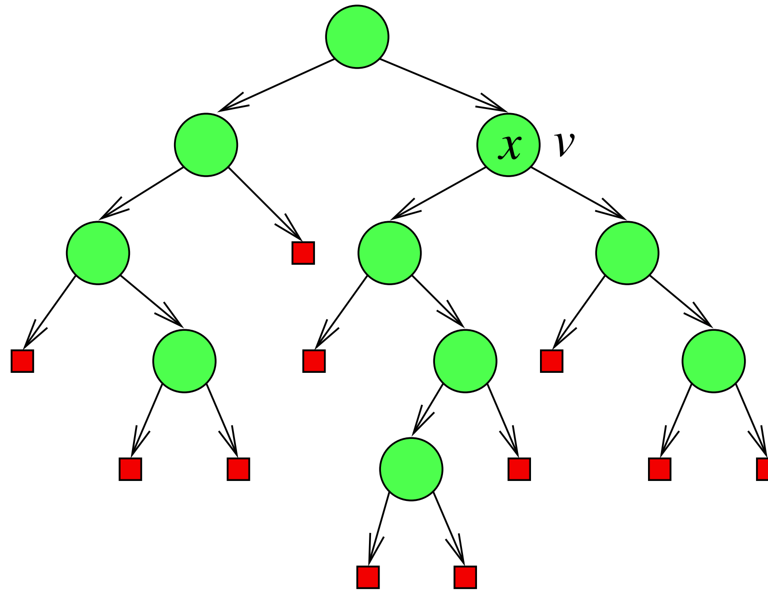
Achtung: Der **Benutzer** der *buildTree*-Operation muss sicherstellen, dass die Knotenmengen von T_1 und T_2 disjunkt sind!

Details: [Saake/Sattler], Kap. 14.2., wenn auch mit leicht anderen Bezeichnungen.

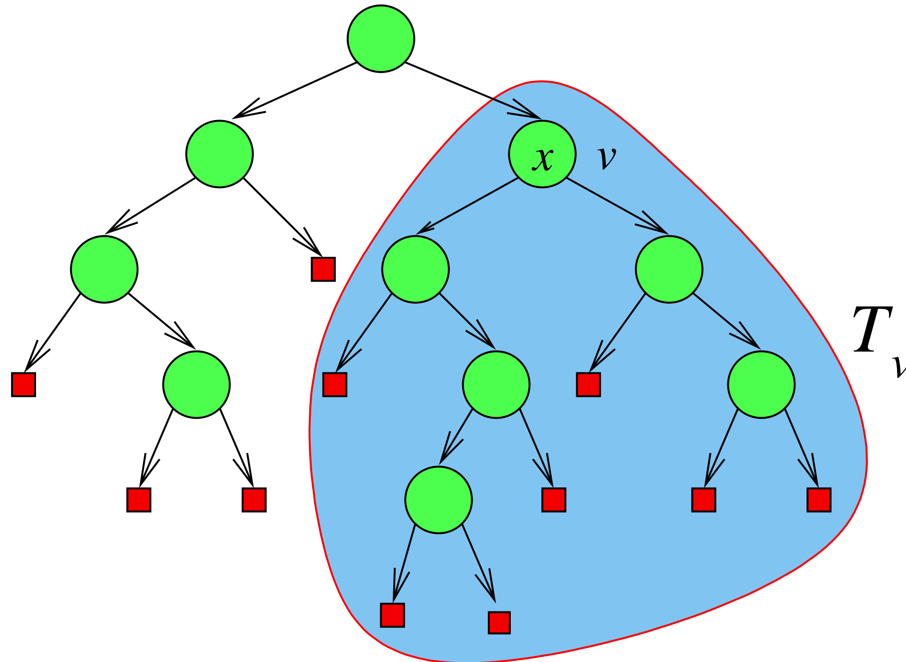
Teil 4: Terminologie

3.2 Binärbäume: Terminologie

- **Binärbaum** T, engl: „*binary tree*“
- (innerer) **Knoten** v, engl: „*node*“
- Knoteneintrag: $x = \mathit{data}(v) = \mathit{data}(T')$ $\qquad (T' = T_v)$



- T_v: Unterbaum mit Wurzel v



T_v ist der Teil von T, der aus den Knoten besteht, deren Weg zur Wurzel durch v verläuft. Die Wurzel von T_v ist v.
Dabei kann v ein beliebiger innerer oder äußerer Knoten sein.

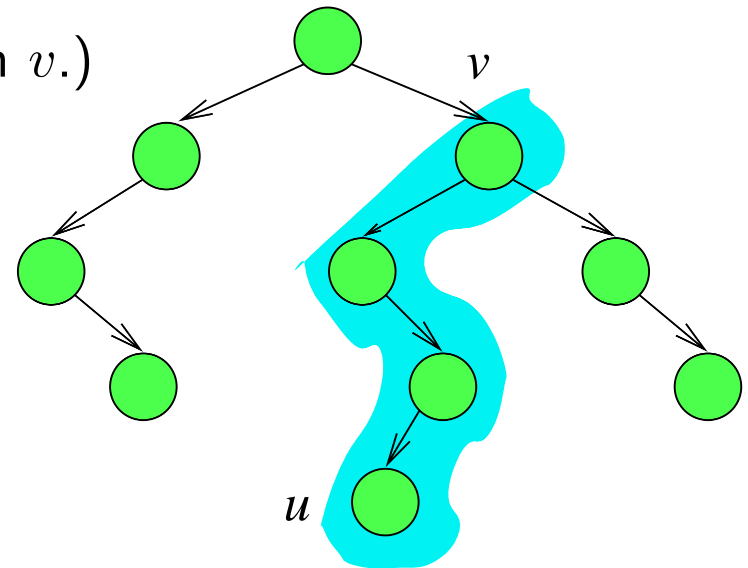
- **(äußerer) Knoten** $\equiv$ **Blatt** l, engl: „*leaf*“, Plural „*leaves*“
- Wurzel, engl: „*root*“

z l

- Blatteintrag: $z = data(l)$

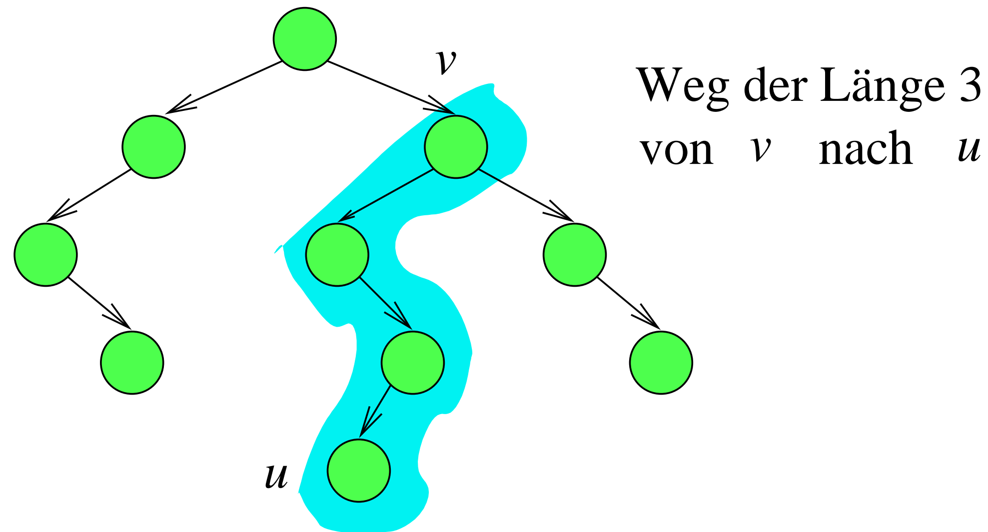
- **Vater/Vorgänger**, engl: „*parent*“, von v: $\boldsymbol{p(v)}$
- linkes **Kind**, engl: „*left child*“, von v: von v: *child*$(v, left)$
- rechtes **Kind**, engl: „*right child*“, von v: von v: *child*$(v, right)$

- **Vater/Vorgänger**, engl: „*parent*“, von v: $\boldsymbol{p(v)}$
- linkes **Kind**, engl: „*left child*“, von v: von v: *child*(v, left)
- rechtes **Kind**, engl: „*right child*“, von v: von v: *child*(v, right)
- **Vorfahr** (Vater von Vater von Vater . . . von u),
  engl: „*ancestor*“ (Im Bild: v ist Vorfahr von u.)
- **Nachfahr** (Kind von Kind von . . . von v),
  engl: „*descendant*“ (Im Bild: u ist Nachfahr von v.)



Achtung: Oft gilt v als (uneigentlicher) Vorfahr/Nachfahr von v.

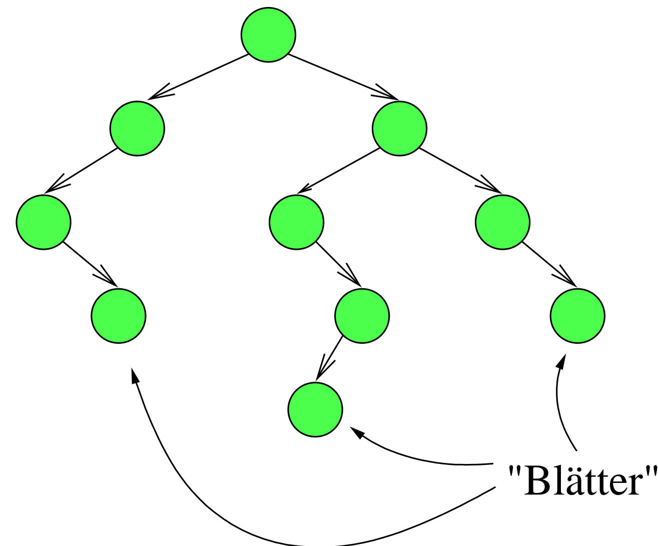
- **Weg** in T: Knotenfolge oder Kantenfolge, die von v zu einem Nachfahren u führt
- **Länge** eines Wegs $(v_0, v_1, \ldots, v_s)$: Anzahl s der **Kanten** auf dem Weg



Häufiger Spezialfall: Die externen Knoten $\Box$ haben keinen Eintrag.

(Bzw. jeder hat denselben nichtssagenden Eintrag, z.B. „$-$“ .)

Ein solcher externer Knoten wird auch als „**leerer Baum**“ bezeichnet.

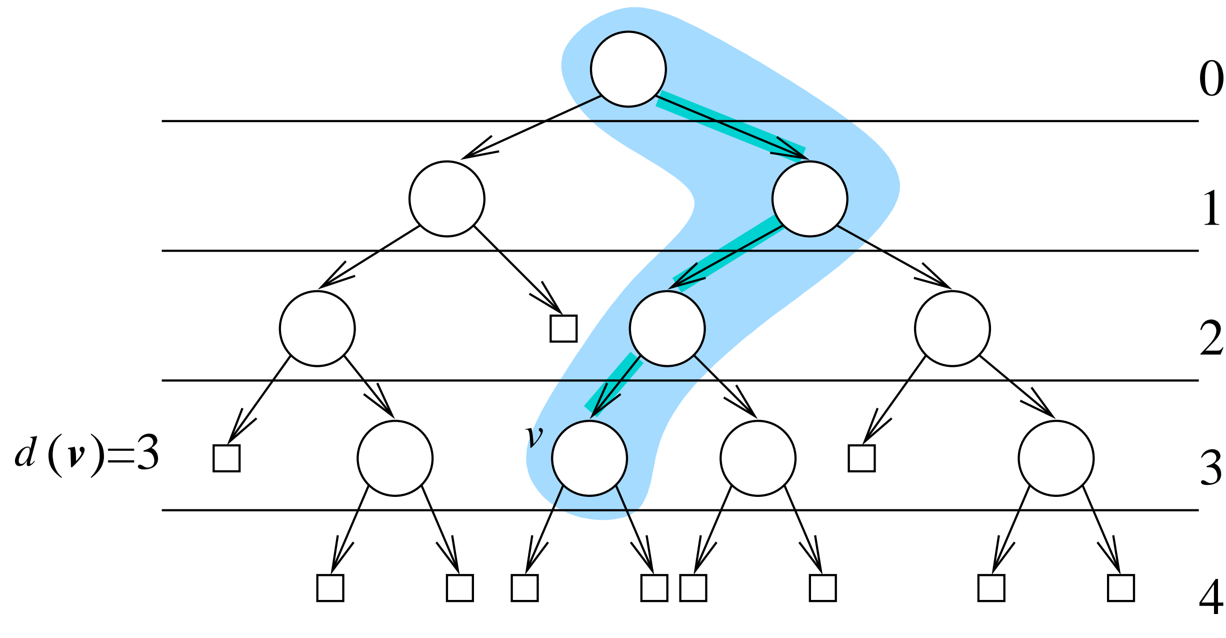


Oft bezeichnet man dann die inneren Knoten, die keinen inneren Knoten als Nachfolger haben, als „Blätter“.

Tiefe/Level eines Knotens

Die Länge (das ist die Kantenzahl) des Wegs $(v_0, v_1, \ldots, v_d)$ von der Wurzel $r = v_0$ zum Knoten $v = v_d$ heißt die **Tiefe** oder das **Level** $d(v)$ von Knoten v.
Die Knoten mit Tiefe l bilden das **Level l**.

Auch externen Knoten wird so eine Tiefe zugeordnet.



Tiefe (oder synonym: Höhe) $d(T)$ (oder: $h(T)$) eines Binärbaums:

1. Fall: Die externen Knoten sind leer (□):

$d(T) := \max(\{-1\} \cup \{d(v) \mid v$ **innerer** Knoten in $T\})$.

Tiefe (oder synonym: Höhe) $d(T)$ (oder: $h(T)$) eines Binärbaums:

1. Fall: Die externen Knoten sind leer ($\square$):

$d(T) := \max(\{-1\} \cup \{d(v) \mid v$ **innerer** Knoten in $T\})$.

Rekursive Formel: $d(T) = -1$ für $T = \square$ und $d((T_1, x, T_2)) = 1 + \max\{d(T_1), d(T_2)\}$.

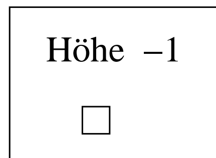
Tiefe (oder synonym: **Höhe**) $d(T)$ (oder: $h(T)$) **eines Binärbaums:**

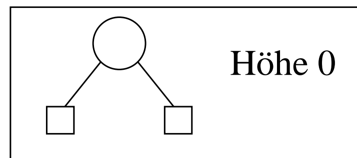
1. Fall: Die externen Knoten sind leer ($\Box$):

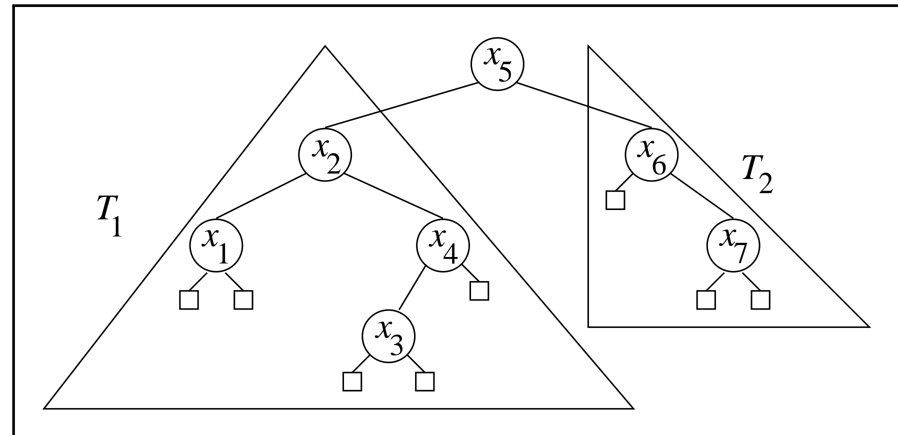
$d(T) := \max(\{-1\} \cup \{d(v) \mid v$ **innerer** Knoten in $T\})$.

Rekursive Formel: $d(T) = -1$ für $T = \Box$ und $d((T_1, x, T_2)) = 1 + \max\{d(T_1), d(T_2)\}$.

Beispiele: Höhe/Tiefe -1, 0 und 3.





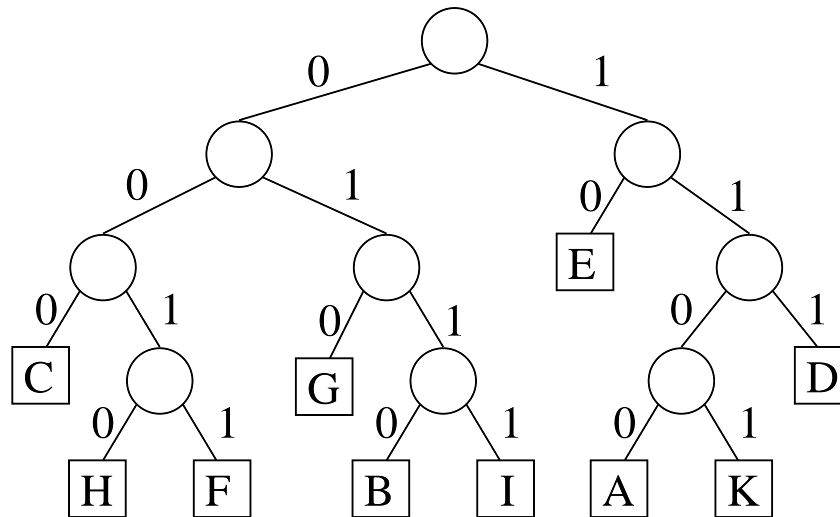


Tiefe (oder synonym: Höhe) $d(T)$ (oder: $h(T)$) eines Binärbaums:

2. Fall: Die externen Knoten sind nicht leer, also von der Form $\boxed{z}$, $z \in Z$:

$d(T) := \max\{d(v) \mid v$ **äußerer** Knoten in $T\}$.

Rekursive Formel: $d(\square) = 0$; $d((T_1, x, T_2) = 1 + \max\{d(T_1), d(T_2)\}$.

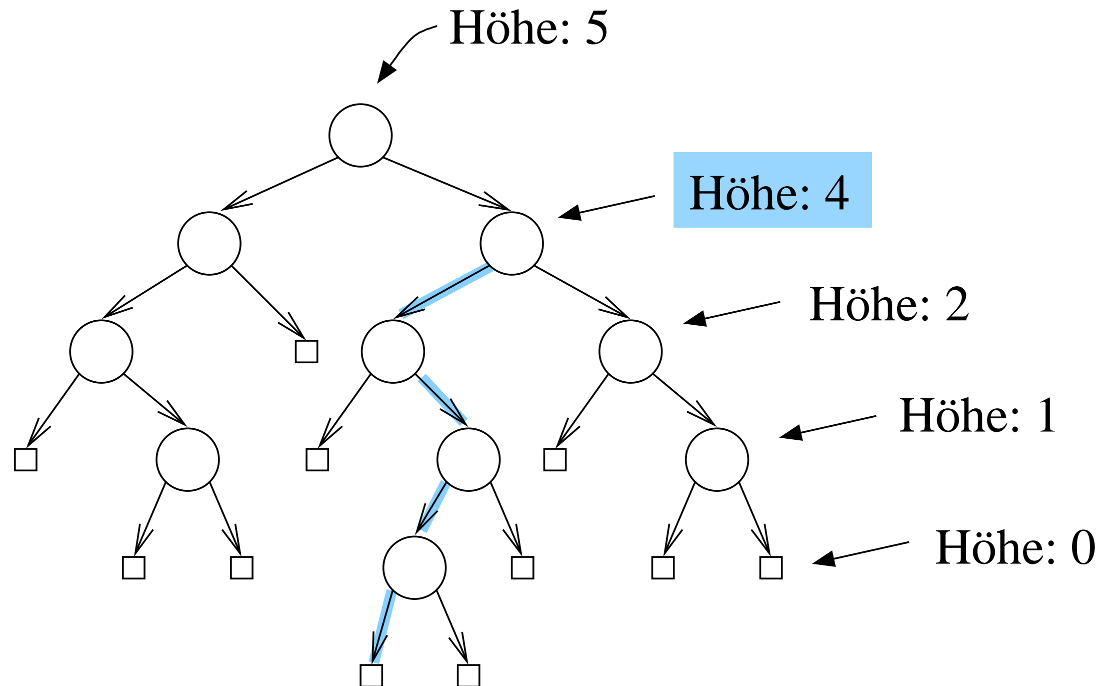


Ein Baum mit Tiefe 4, gleich der maximalen Länge eines Weges zu einem Blatt.

Höhe eines Knotens:

Die **Höhe** $h(v)$ eines Knotens in T ist die **Tiefe** $d(T_v)$ des Teilbaums mit Wurzel v.

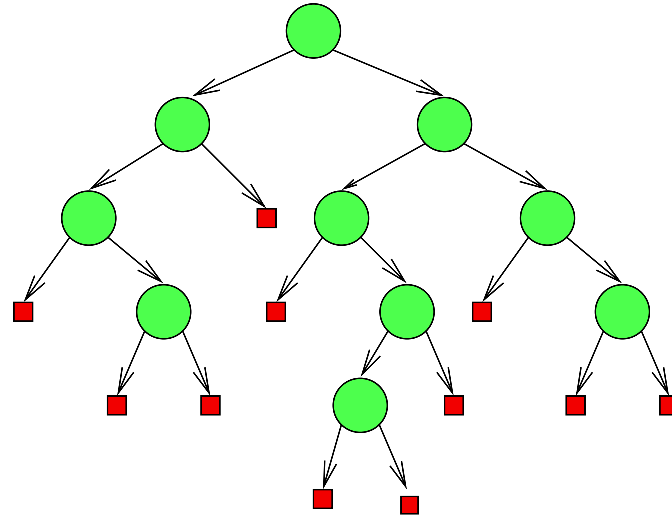
Beispiel:



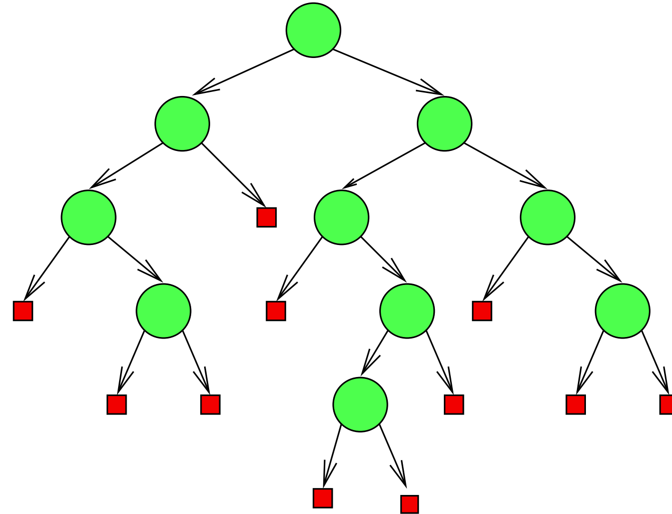
Hier angegeben: Fall „Blätter nicht leer“.

Teil 5: TIPL und TEPL

3.3 Binärbäume: Eigenschaften



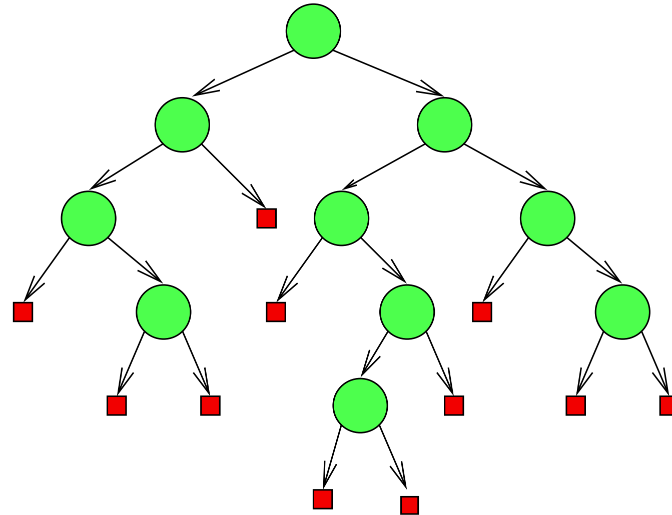
3.3 Binärbäume: Eigenschaften



Proposition 3.3.1

Für einen Binärbaum T mit $n \geq 0$ inneren Knoten und **leeren** äußeren Knoten gilt:

3.3 Binärbäume: Eigenschaften

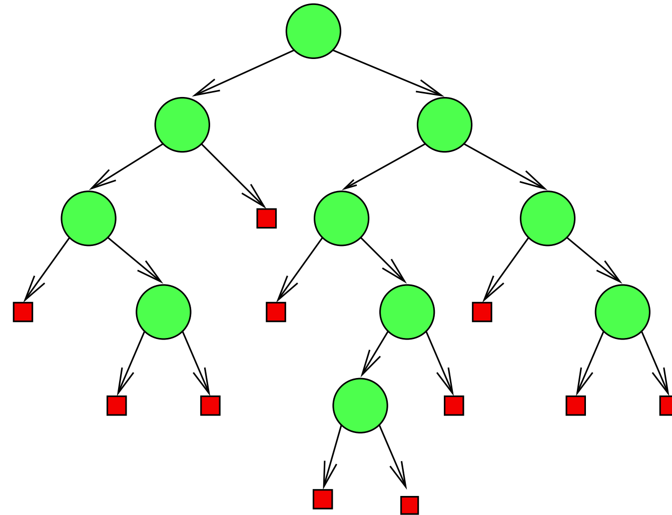


Proposition 3.3.1

Für einen Binärbaum T mit $n \geq 0$ inneren Knoten und **leeren** äußeren Knoten gilt:

(a) T hat $2n$ Zeiger/Kanten. (Im Beispiel: $n = 10$, $2n = 20$.)

3.3 Binärbäume: Eigenschaften



Proposition 3.3.1

Für einen Binärbaum T mit $n \geq 0$ inneren Knoten und **leeren** äußeren Knoten gilt:

(a) T hat $2n$ Zeiger/Kanten. (Im Beispiel: $n = 10$, $2n = 20$.)

(b) T hat $n + 1$ äußere Knoten. (Im Beispiel: $n + 1 = 11$.)

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l$$

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l = 2^{l+1} - 1$$

innere Knoten.

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l = 2^{l+1} - 1$$

innere Knoten.
Daraus: $\log(n+1) \leq l + 1$, für das Level l mit dem tiefsten inneren Knoten.

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l = 2^{l+1} - 1$$

innere Knoten.
Daraus: $\log(n+1) \leq l + 1$, für das Level l mit dem tiefsten inneren Knoten.

(e)* $\lceil \log(n+1) \rceil - 1 \leq d(T) \leq n - 1$.

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l = 2^{l+1} - 1$$

innere Knoten.
Daraus: $\log(n+1) \le l + 1$, für das Level l mit dem tiefsten inneren Knoten.

(e)* $\lceil \log(n+1) \rceil - 1 \le d(T) \le n - 1$.

Merke: Die Tiefe eines Binärbaums mit n inneren Knoten ist

mindestens $\log n - 1$ und höchstens $n - 1$.

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l = 2^{l+1} - 1$$

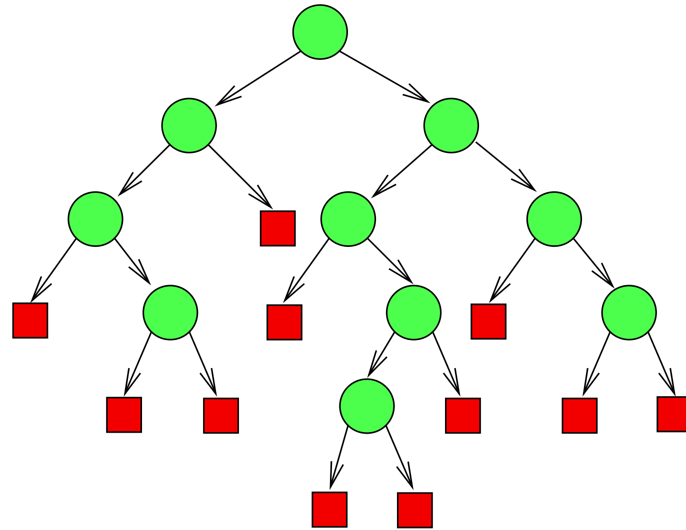
innere Knoten.
Daraus: $\log(n + 1) \leq l + 1$, für das Level l mit dem tiefsten inneren Knoten.

(e)* $\lceil \log(n + 1) \rceil - 1 \leq d(T) \leq n - 1$.

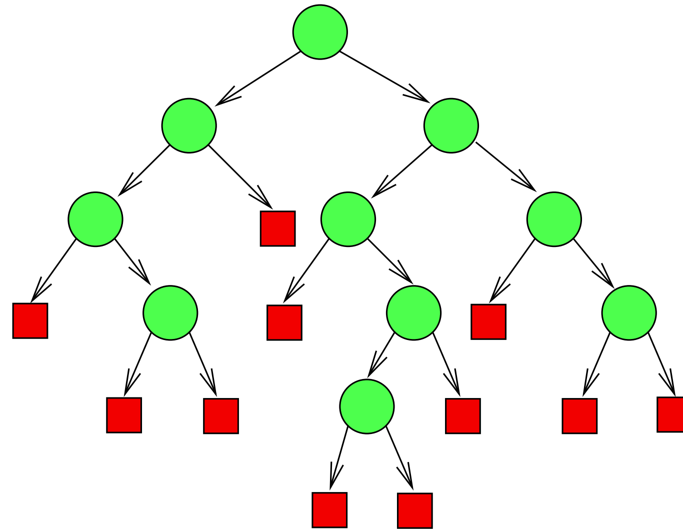
Merke: Die Tiefe eines Binärbaums mit n inneren Knoten ist

mindestens $\log n - 1$ und höchstens $n - 1$.

* $\log n = \log_2 n$.

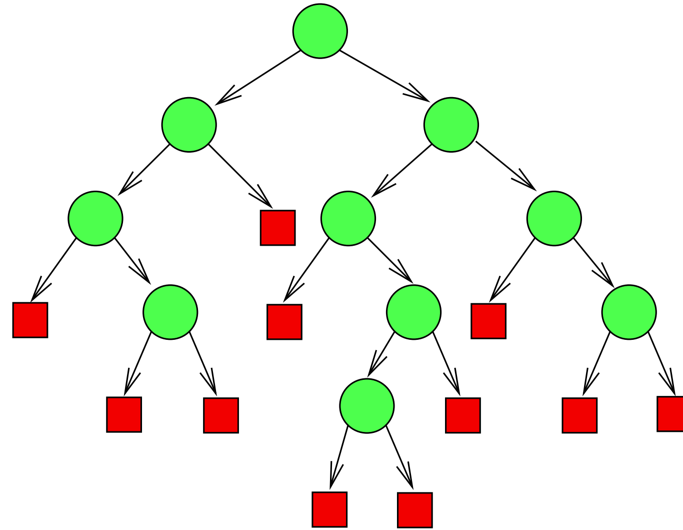


Proposition 3.3.2



Proposition 3.3.2

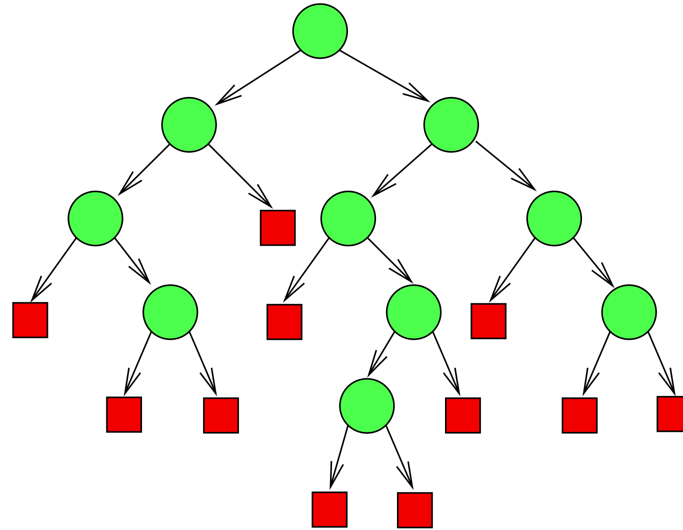
Für einen Binärbaum T mit $N \geq 1$ **nichtleeren** äußeren Knoten gilt:



Proposition 3.3.2

Für einen Binärbaum T mit $N \geq 1$ **nichtleeren** äußeren Knoten gilt:

(a) T hat $n = N - 1$ innere Knoten. (Im Beispiel: $N = 11$, $n = 10$.)

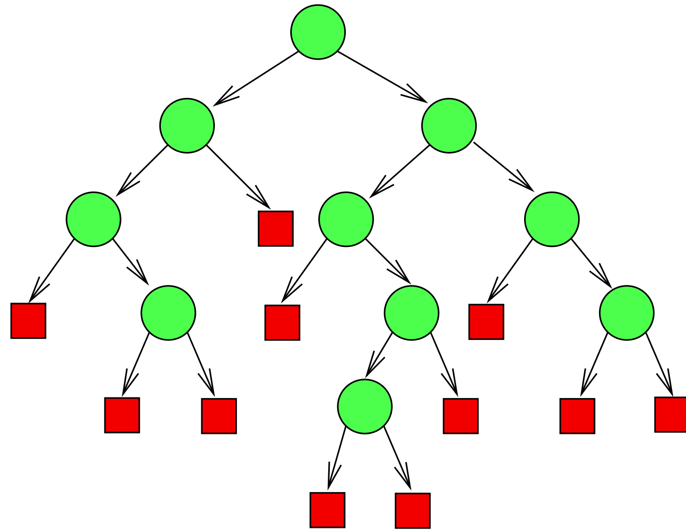


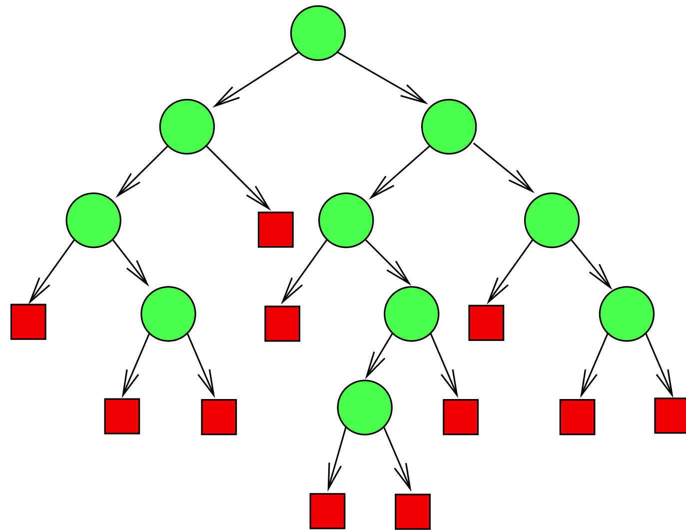
Proposition 3.3.2

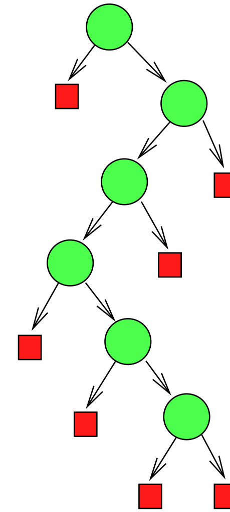
Für einen Binärbaum T mit $N \geq 1$ **nichtleeren** äußeren Knoten gilt:

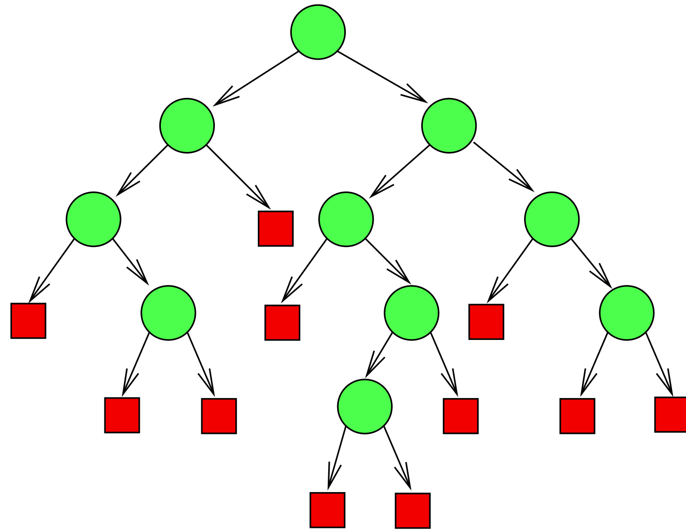
(a) T hat $n = N - 1$ innere Knoten. (Im Beispiel: $N = 11$, $n = 10$.)

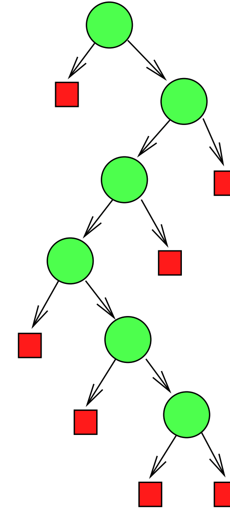
(b) T hat $2N - 2$ Zeiger/Kanten.



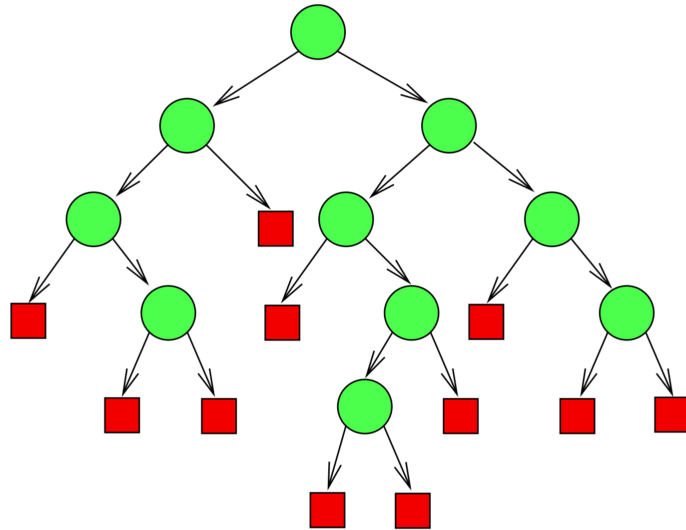


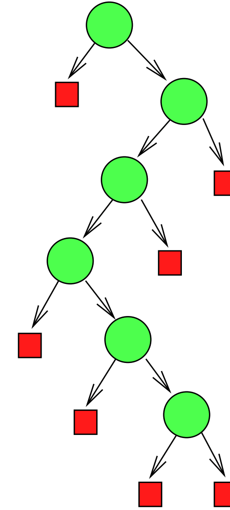




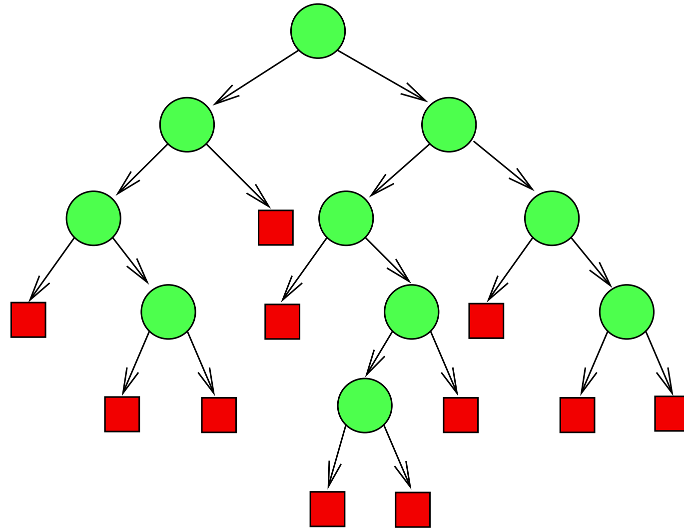


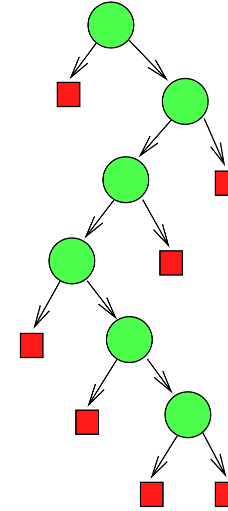
(c) $N \leq 2^{d(T)}$.





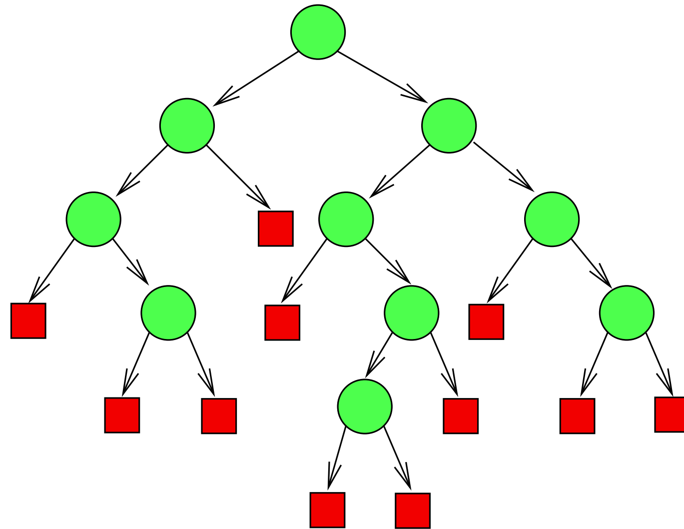
(c) $N \leq 2^{d(T)}$. (*Beweis*: Induktion über Binärbaume, rekursive Struktur.)

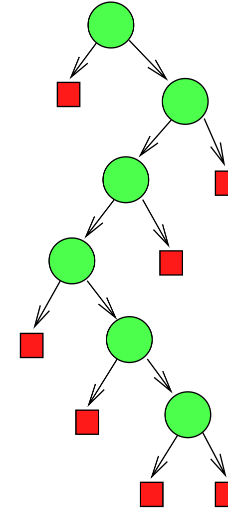




(c) $N \leq 2^{d(T)}$. (*Beweis*: Induktion über Binärbaume, rekursive Struktur.)

(d) $N - 1 \geq d(T) \geq \lceil \log N \rceil$.





(c) $N \le 2^{d(T)}$. (*Beweis*: Induktion über Binärbaume, rekursive Struktur.)

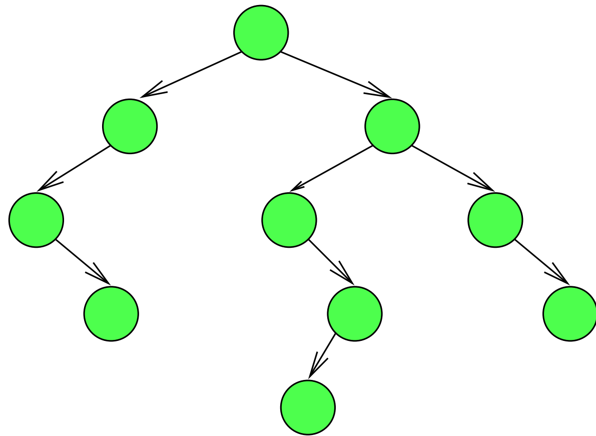
(d) $N - 1 \ge d(T) \ge \lceil \log N \rceil$.

Merke:

Die Tiefe eines Binärbaums mit N **nichtleeren** äußeren Knoten ist $\ge \log N$.

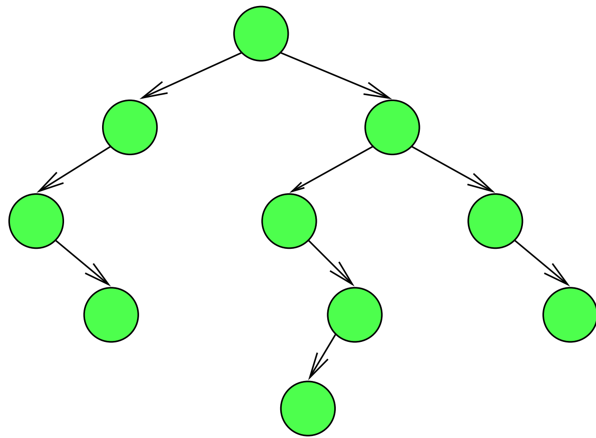
Totale innere Weglänge

Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



Totale innere Weglänge

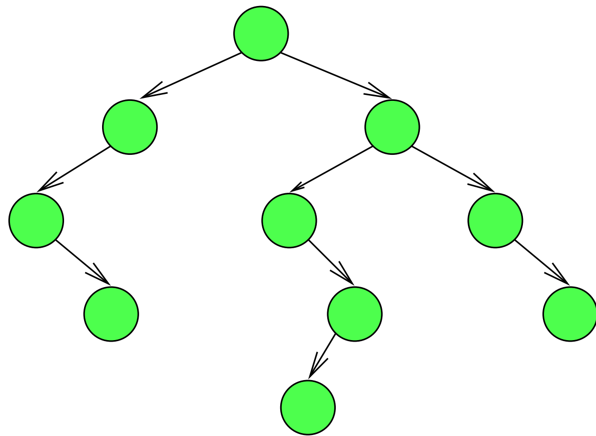
Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



0

Totale innere Weglänge

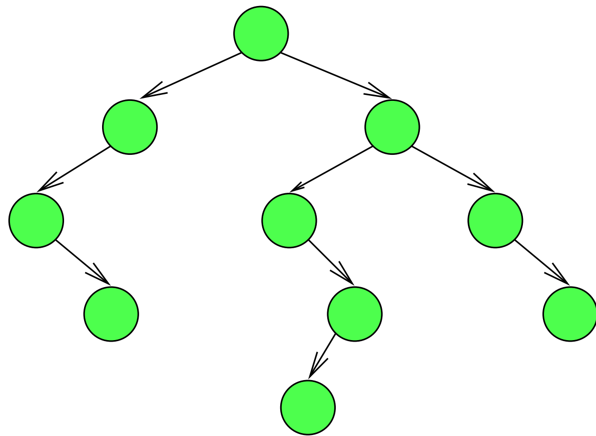
Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$0 + 2 \cdot 1$

Totale innere Weglänge

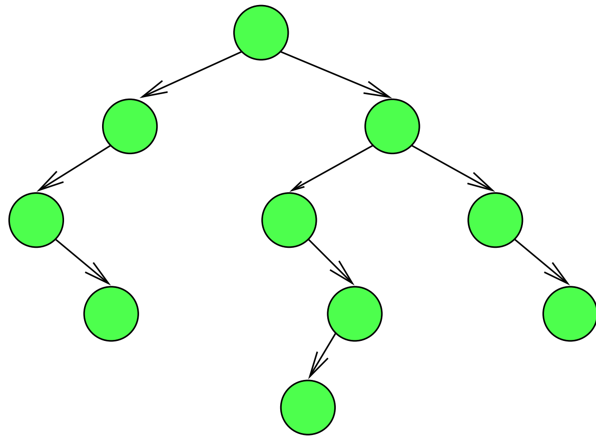
Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 +$$

Totale innere Weglänge

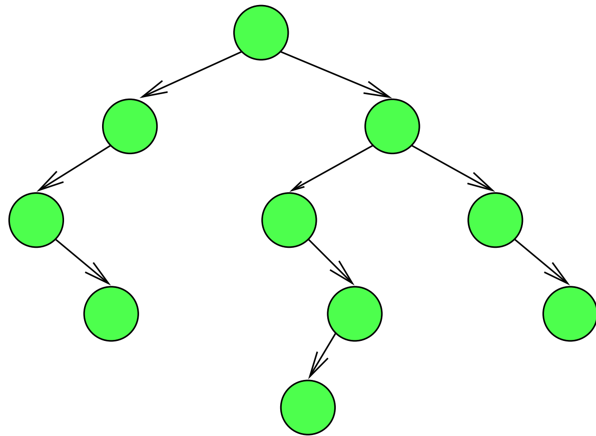
Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 +$$

Totale innere Weglänge

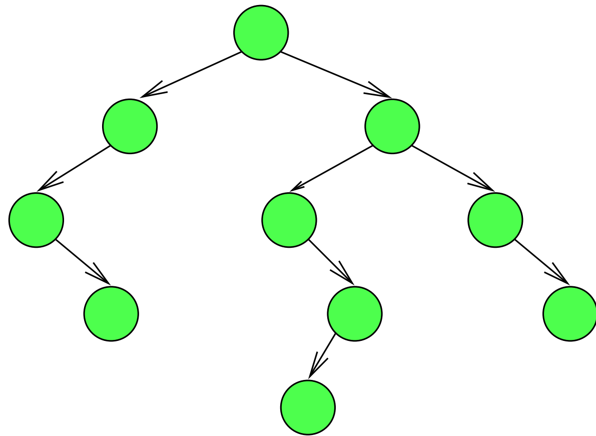
Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4$$

Totale innere Weglänge

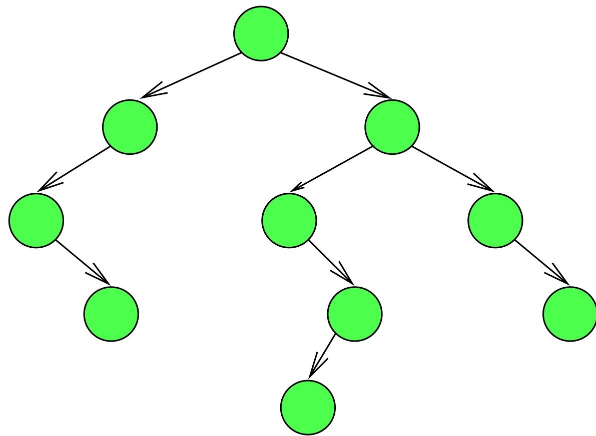
Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 = 21$$

Totale innere Weglänge

Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



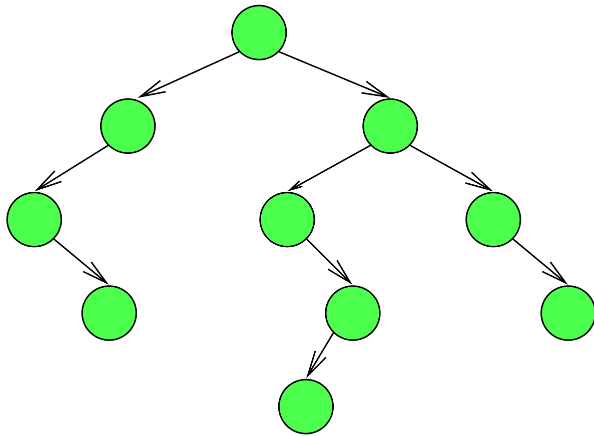
$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 = 21$$

Totale innere Weglänge („**t**otal **i**nternal **p**ath **l**ength“):

TIPL$(T) := \sum_{v \in V} d(v)$.

Totale innere Weglänge

Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



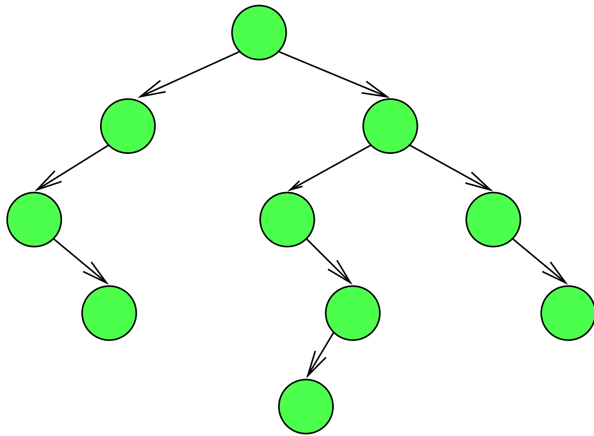
$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 = 21$$

Totale innere Weglänge („**t**otal **i**nternal **p**ath **l**ength“):

TIPL$(T) := \sum_{v \in V} d(v)$. (Im Beispiel: 21.)

Totale innere Weglänge

Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 = 21$$

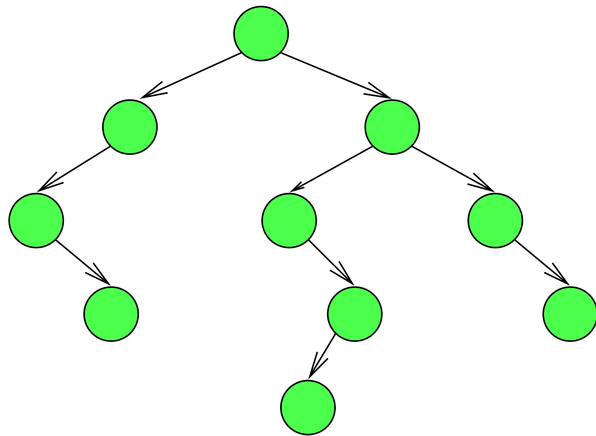
Totale innere Weglänge („**t**otal **i**nternal **p**ath **l**ength"):

TIPL$(T) := \sum_{v \in V} d(v)$.   (Im Beispiel: 21.)

Mittlere innere Weglänge: $\frac{1}{n} \cdot \text{TIPL}(T)$.

Totale innere Weglänge

Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 = 21$$

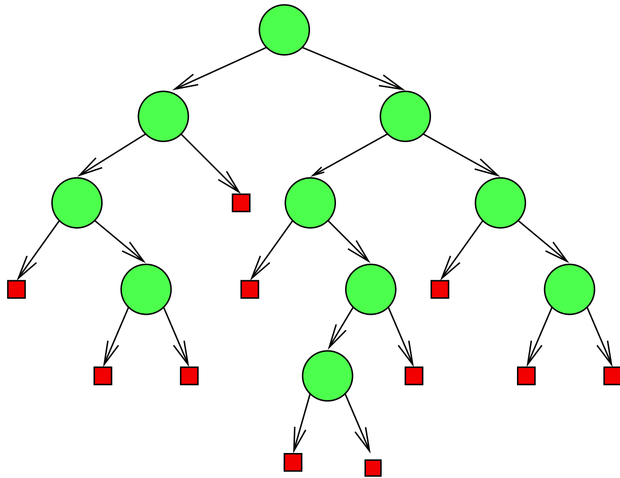
Totale innere Weglänge („**t**otal **i**nternal **p**ath **l**ength“):

TIPL$(T) := \sum_{v \in V} d(v)$. (Im Beispiel: 21.)

Mittlere innere Weglänge: $\frac{1}{n} \cdot \text{TIPL}(T)$. (Im Beispiel: $\frac{21}{10}$.)

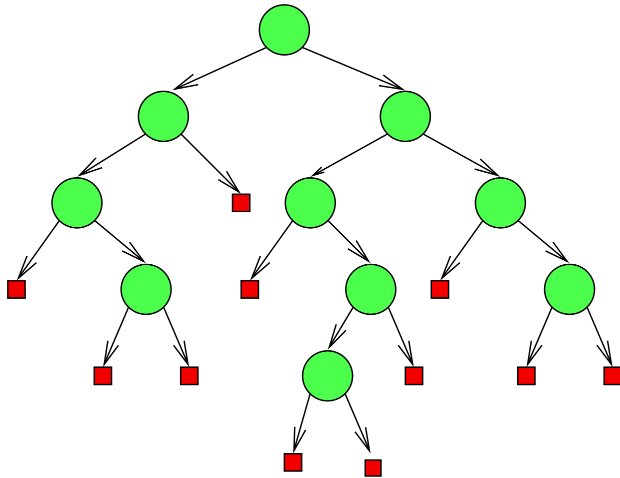
Totale äußere Weglänge

Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



Totale äußere Weglänge

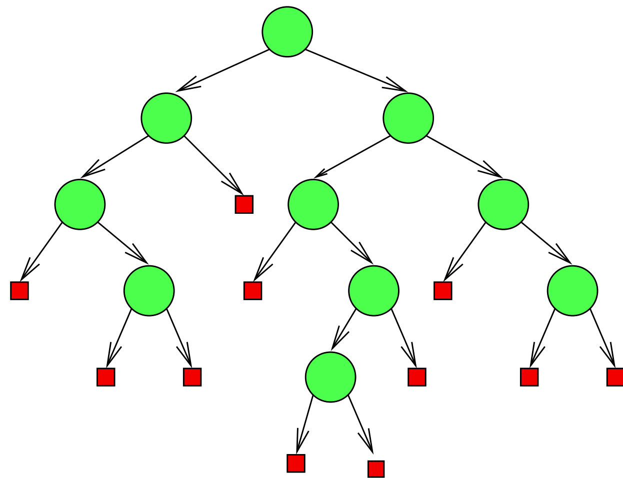
Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2$$

Totale äußere Weglänge

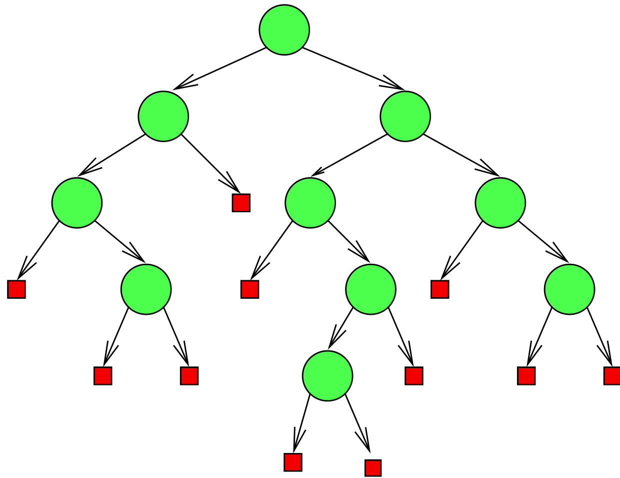
Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2 + 3 \cdot 3$$

Totale äußere Weglänge

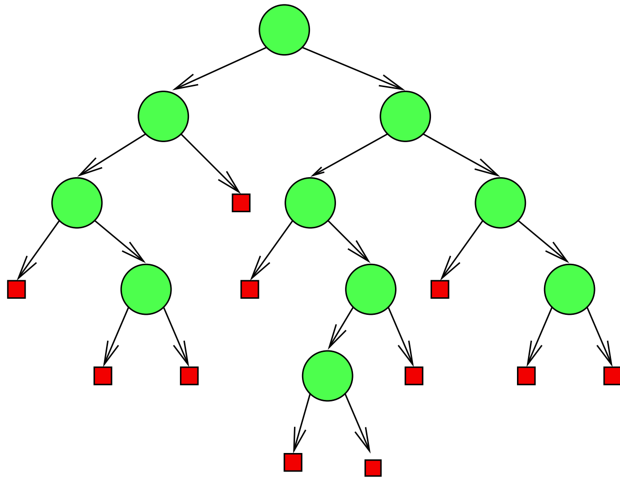
Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2 + 3 \cdot 3 + 5 \cdot 4$$

Totale äußere Weglänge

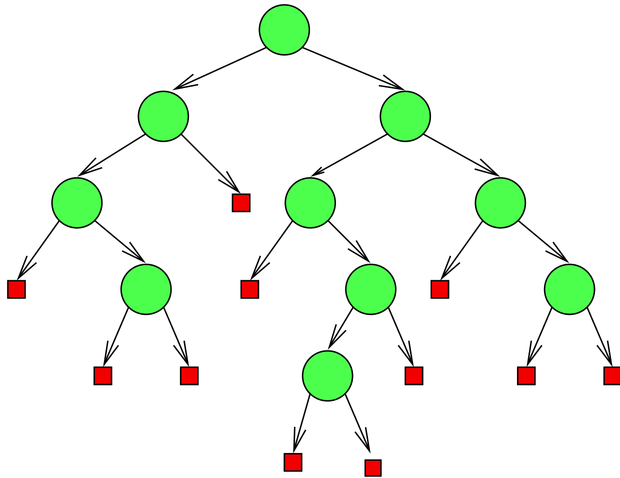
Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 5 = 41.$$

Totale äußere Weglänge

Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



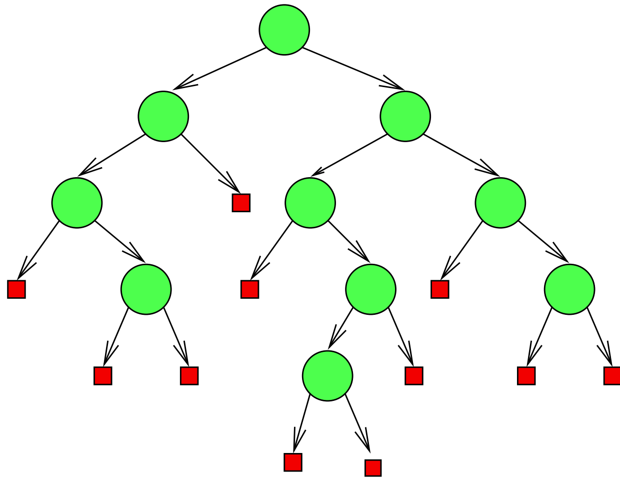
$$2 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 5 = 41.$$

Totale äußere Weglänge („**t**otal **e**xternal **p**ath **l**ength“):

TEPL$(T) := \sum_{l \in L} d(l)$.

Totale äußere Weglänge

Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



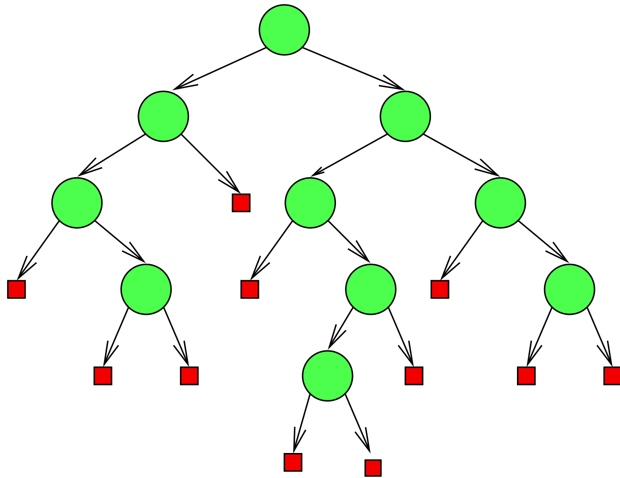
$$2 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 5 = 41.$$

Totale äußere Weglänge („**t**otal **e**xternal **p**ath **l**ength“):

TEPL$(T) := \sum_{l \in L} d(l)$. (Im Beispiel: 41.)

Totale äußere Weglänge

Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 5 = 41.$$

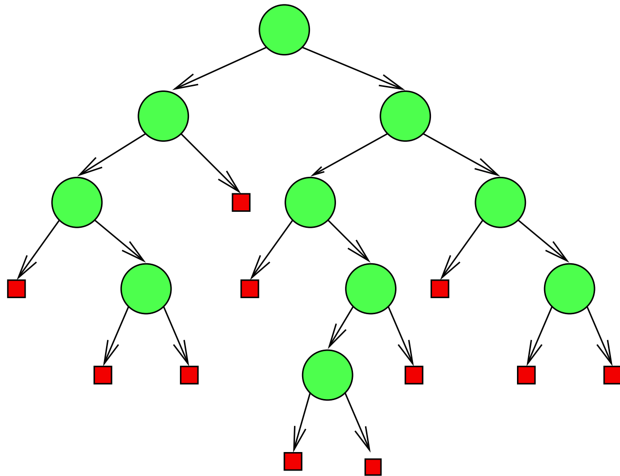
Totale äußere Weglänge („**t**otal **e**xternal **p**ath **l**ength“):

TEPL$(T) := \sum_{l \in L} d(l)$. (Im Beispiel: 41.)

Mittlere äußere Weglänge: TEPL$(T)/(n+1)$.

Totale äußere Weglänge

Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 5 = 41.$$

Totale äußere Weglänge („**t**otal **e**xternal **p**ath **l**ength“):

TEPL$(T) := \sum_{l \in L} d(l)$. (Im Beispiel: 41.)

Mittlere äußere Weglänge: TEPL$(T)/(n+1)$. (Im Beispiel $\frac{41}{11}$.)

Proposition 3.3.3

Für jeden Binärbaum mit n inneren Knoten gilt:

Proposition 3.3.3

Für jeden Binärbaum mit n inneren Knoten gilt: $\text{TEPL}(T) = \text{TIPL}(T) + 2n$.

Proposition 3.3.3

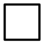
Für jeden Binärbaum mit n inneren Knoten gilt: $\text{TEPL}(T) = \text{TIPL}(T) + 2n$.

(Im Beispiel: $41 = 21 + 2 \cdot 10$.)

Proposition 3.3.3

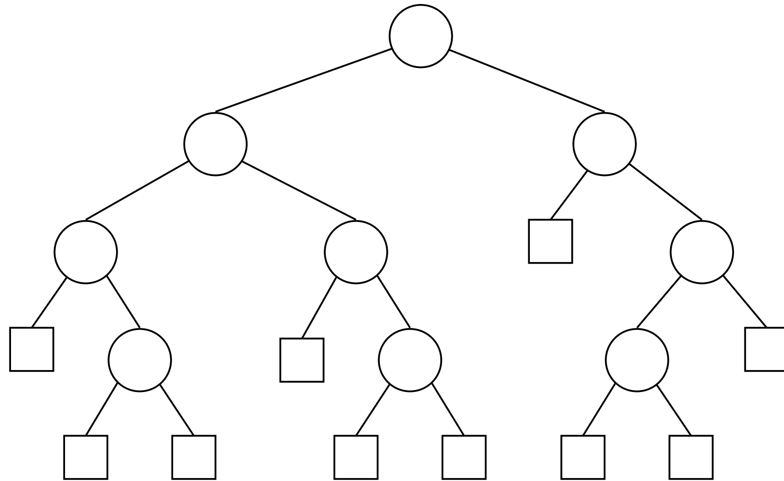
Für jeden Binärbaum mit n inneren Knoten gilt: $\text{TEPL}(T) = \text{TIPL}(T) + 2n$.

(Im Beispiel: $41 = 21 + 2 \cdot 10$.)

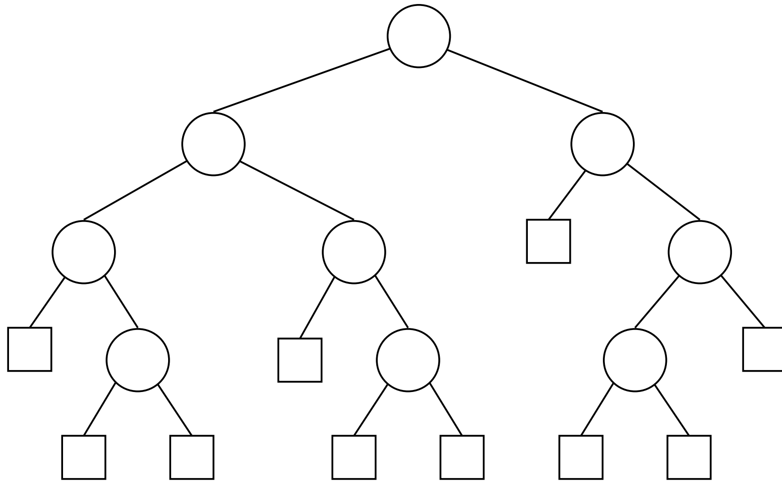
Beweis: Induktion über n, s. Tafel/Druckfolien. 

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.



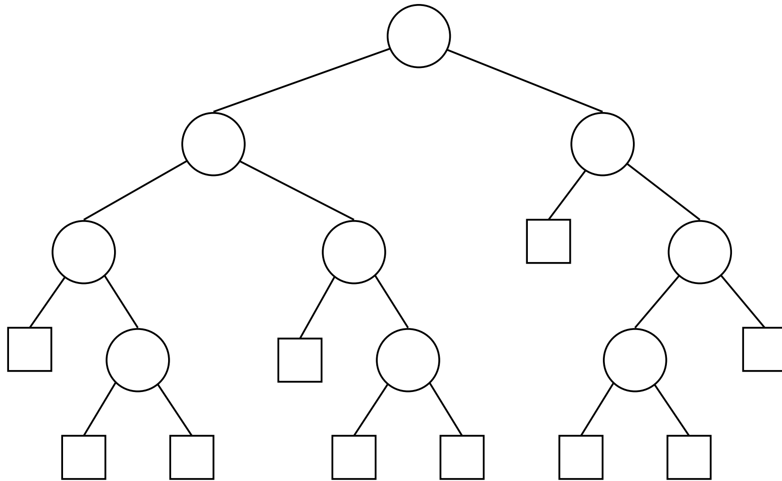
Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.



Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge.

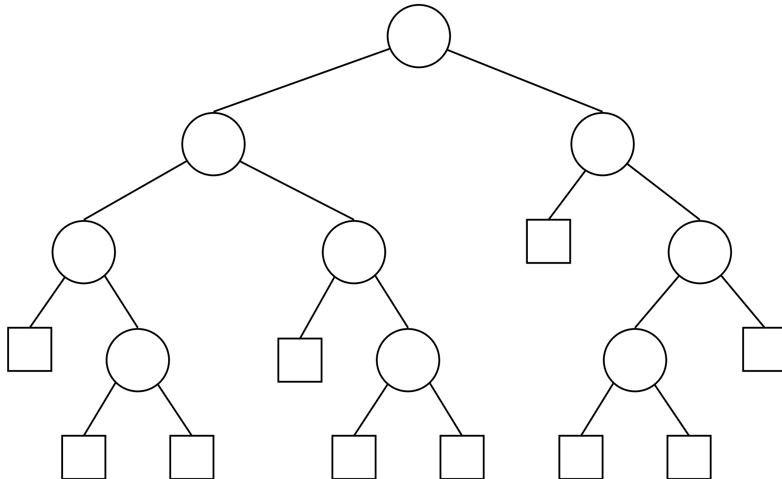
Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.



Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.

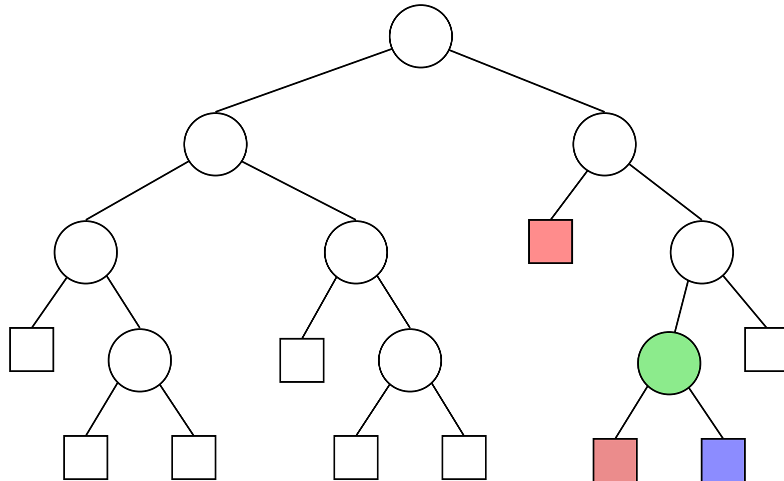


Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Beweis hiervon: Sonst könnte man TEPL(T) durch Umbauen verkleinern, wie im Bild.

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.

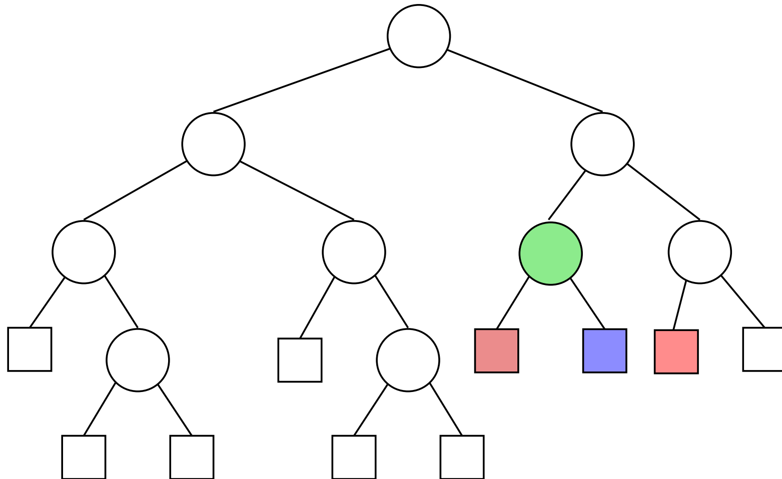


Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Beweis hiervon: Sonst könnte man TEPL(T) durch Umbauen verkleinern, wie im Bild.

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.



Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Beweis hiervon: Sonst könnte man TEPL(T) durch Umbauen verkleinern, wie im Bild.

Proposition 3.3.5 Binärbaum T hat N äußere Knoten $\Rightarrow$ TEPL$(T) \geq N \log N$.

Beweis: Siehe Tafel/Druckfolien.

Korollar 3.3.6

Korollar 3.3.6

Für die **mittlere äußere Weglänge** eines Binärbaums T mit N Blättern gilt:

$$\boxed{\frac{1}{N}\mathsf{TEPL}(T) \geq \log N.}$$

Korollar 3.3.6

Für die **mittlere äußere Weglänge** eines Binärbaums T mit N Blättern gilt:

$$\boxed{\frac{1}{N}\mathsf{TEPL}(T) \geq \log N.}$$

Nach Proposition 3.3.3: $\mathsf{TEPL}(T) = \mathsf{TIPL}(T) + 2n$.

Korollar 3.3.6

Für die **mittlere äußere Weglänge** eines Binärbaums T mit N Blättern gilt:

$$\boxed{\frac{1}{N}\mathsf{TEPL}(T) \geq \log N.}$$

Nach Proposition 3.3.3: $\mathsf{TEPL}(T) = \mathsf{TIPL}(T) + 2n$.
Also: $\mathsf{TIPL}(T) \geq \mathsf{TEPL}(T) - 2n \geq (n+1)\log(n+1) - 2n > n(\log n - 2)$.

Korollar 3.3.7

Korollar 3.3.6

Für die **mittlere äußere Weglänge** eines Binärbaums T mit N Blättern gilt:

$$\boxed{\frac{1}{N}\mathsf{TEPL}(T) \geq \log N.}$$

Nach Proposition 3.3.3: $\mathsf{TEPL}(T) = \mathsf{TIPL}(T) + 2n$.
Also: $\mathsf{TIPL}(T) \geq \mathsf{TEPL}(T) - 2n \geq (n+1)\log(n+1) - 2n > n(\log n - 2)$.

Korollar 3.3.7

Für die **mittlere innere Weglänge** eines Binärbaums T mit n Knoten gilt:

$$\boxed{\frac{1}{n}\mathsf{TIPL}(T) > \log n - 2.}$$

Teil 6: Baumdurchläufe

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen,

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit`(T) löst Abarbeitung einer Methode von *data*(T) aus.

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit`(T) löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit(`T`)` löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

– Daten *data*(v) ausgeben;

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit`(T) löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

– Daten *data*(v) ausgeben;

– laufende Nummer für v vergeben;

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit`(T) löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

– Daten *data*(v) ausgeben;

– laufende Nummer für v vergeben;

– Zeiger auf v an Liste anhängen;

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit`(T) löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

– Daten *data*(v) ausgeben;

– laufende Nummer für v vergeben;

– Zeiger auf v an Liste anhängen;

– zu v gehörende Aktion ausführen

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit(`T`)` löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

– Daten *data*(v) ausgeben;

– laufende Nummer für v vergeben;

– Zeiger auf v an Liste anhängen;

– zu v gehörende Aktion ausführen (z. B. arithmetische Operation)

usw.

Inorder-Durchlauf durch T

Inorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ; // besuche äußeren Knoten

Inorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ; // besuche äußeren Knoten

falls $T = (T_1, x, T_2)$:

Inorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ; // besuche äußeren Knoten

falls $T = (T_1, x, T_2)$:

Inorder-Durchlauf durch T_1;

Inorder-Durchlauf durch T

falls $T = \boxed{z}$:

    e-visit(z) ;   // besuche äußeren Knoten

falls $T = (T_1, x, T_2)$:

    Inorder-Durchlauf durch T_1;

    i-visit(x) ;   // besuche inneren Knoten

Inorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ; // besuche äußeren Knoten

falls $T = (T_1, x, T_2)$:

Inorder-Durchlauf durch T_1;

`i-visit`(x) ; // besuche inneren Knoten

Inorder-Durchlauf durch T_2.

Inorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ; // besuche äußeren Knoten

falls $T = (T_1, x, T_2)$:

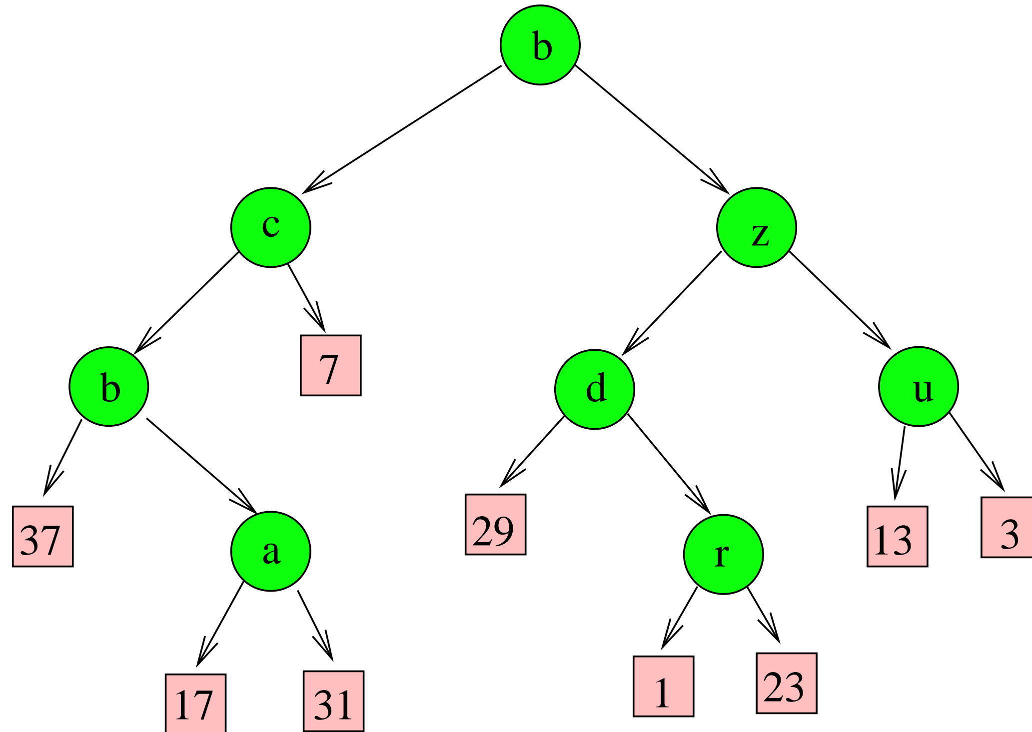
Inorder-Durchlauf durch T_1;

`i-visit`(x) ; // besuche inneren Knoten

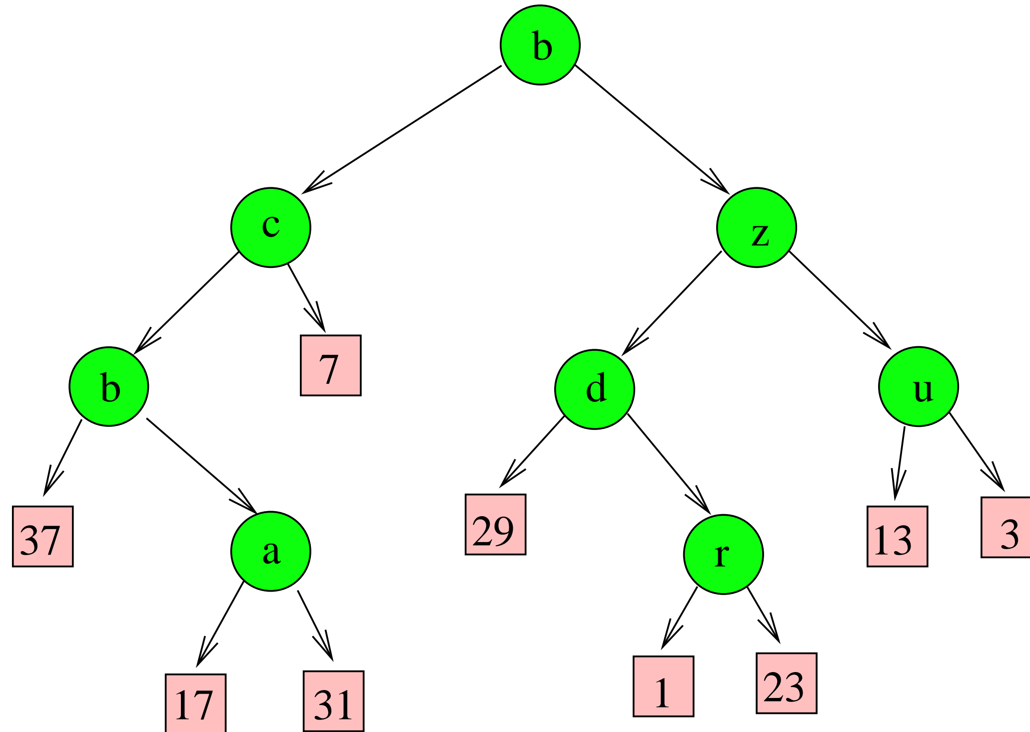
Inorder-Durchlauf durch T_2.

Inorder:	„Erst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum.“

Inorder-Durchlauf durch T

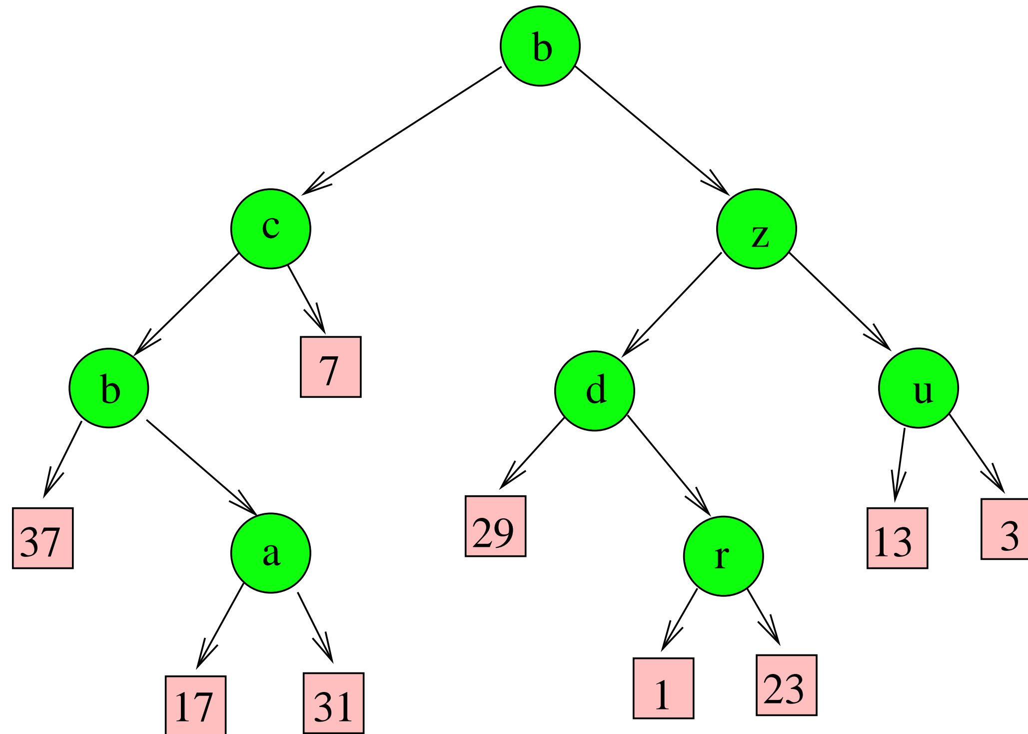


Inorder-Durchlauf durch T



Ausgabe: $37, \mathrm{b}, 17, \mathrm{a}, 31, \mathrm{c}, 7, \mathrm{b}, 29, \mathrm{d}, 1, \mathrm{r}, 23, \mathrm{z}, 13, \mathrm{u}, 3.$

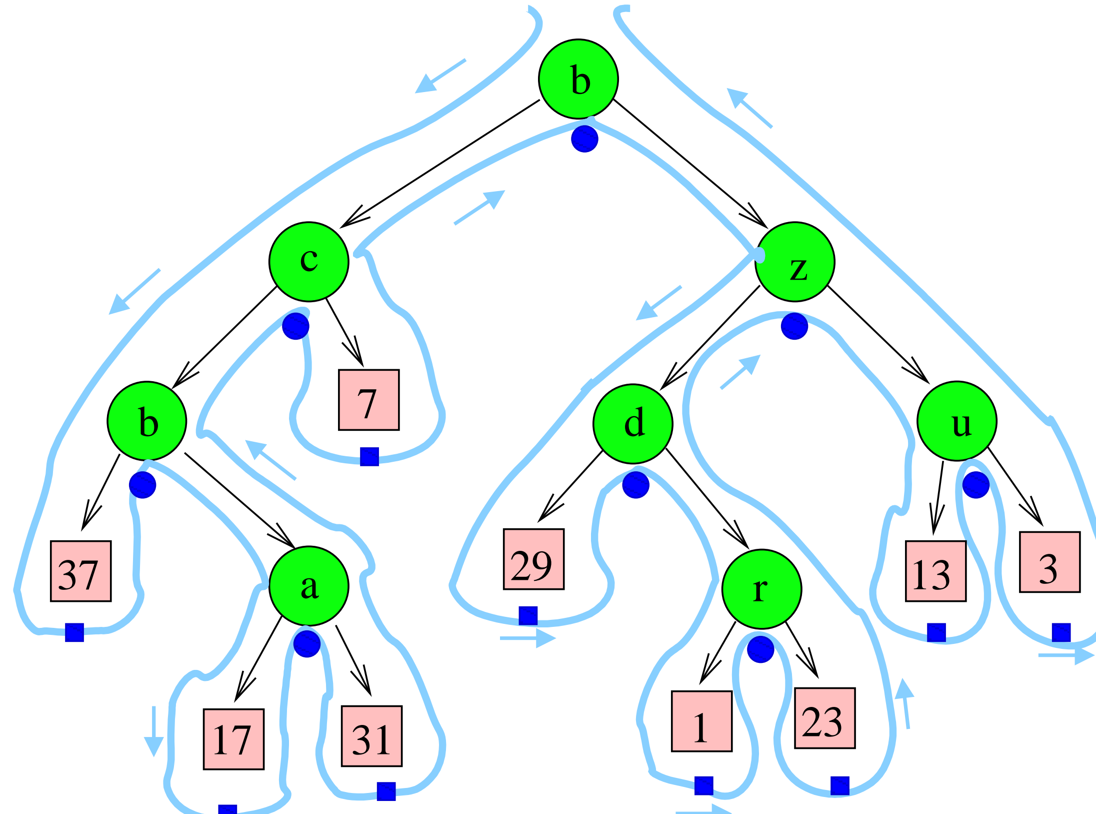
Inorder-Durchlauf durch T



Ausgabe: $37, \mathrm{b}, 17, \mathrm{a}, 31, \mathrm{c}, 7, \mathrm{b}, 29, \mathrm{d}, 1, \mathrm{r}, 23, \mathrm{z}, 13, \mathrm{u}, 3.$

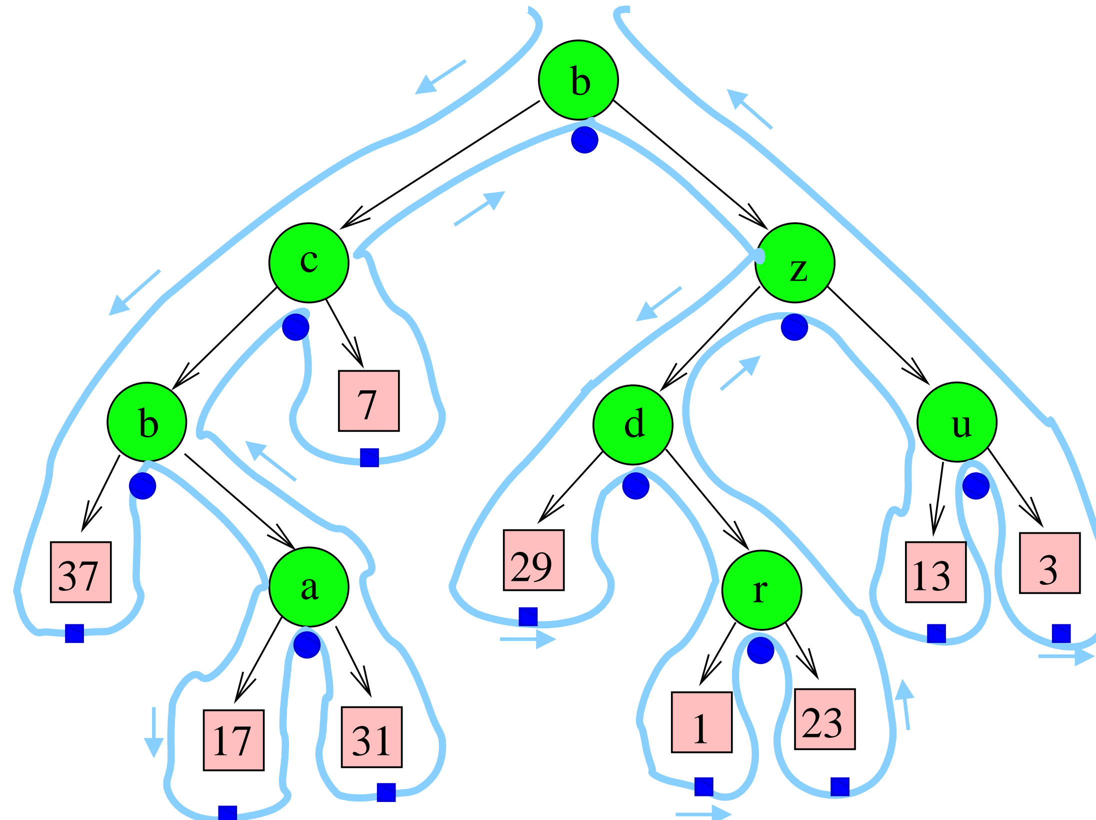
Beobachte: Man besucht abwechselnd externe und interne Knoten.

Inorder-Durchlauf durch T



Ausgabe: 37, b, 17, a, 31, c, 7, b, 29, d, 1, r, 23, z, 13, u, 3.

Inorder-Durchlauf durch T



Ausgabe: $37, \text{b}, 17, \text{a}, 31, \text{c}, 7, \text{b}, 29, \text{d}, 1, \text{r}, 23, \text{z}, 13, \text{u}, 3.$

Beobachte: Man besucht abwechselnd externe und interne Knoten.

Präorder-Durchlauf durch T

Präorder-Durchlauf durch T

falls $T = \boxed{z}$:

```
    e-visit(z) ;
```

Präorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ;

falls $T = (T_1, x, T_2)$:

Präorder-Durchlauf durch T

falls $T = \boxed{z}$:

```
e-visit(z) ;
```

falls $T = (T_1, x, T_2)$:

```
i-visit(x) ;
```

Präorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ;

falls $T = (T_1, x, T_2)$:

`i-visit`(x) ;

Präorder-Durchlauf durch T_1;

Präorder-Durchlauf durch T

falls $T = \boxed{z}$:

  `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:

  `i-visit`(x) ;

  Präorder-Durchlauf durch T_1;

  Präorder-Durchlauf durch T_2;

Präorder-Durchlauf durch T

falls $T = \boxed{z}$:

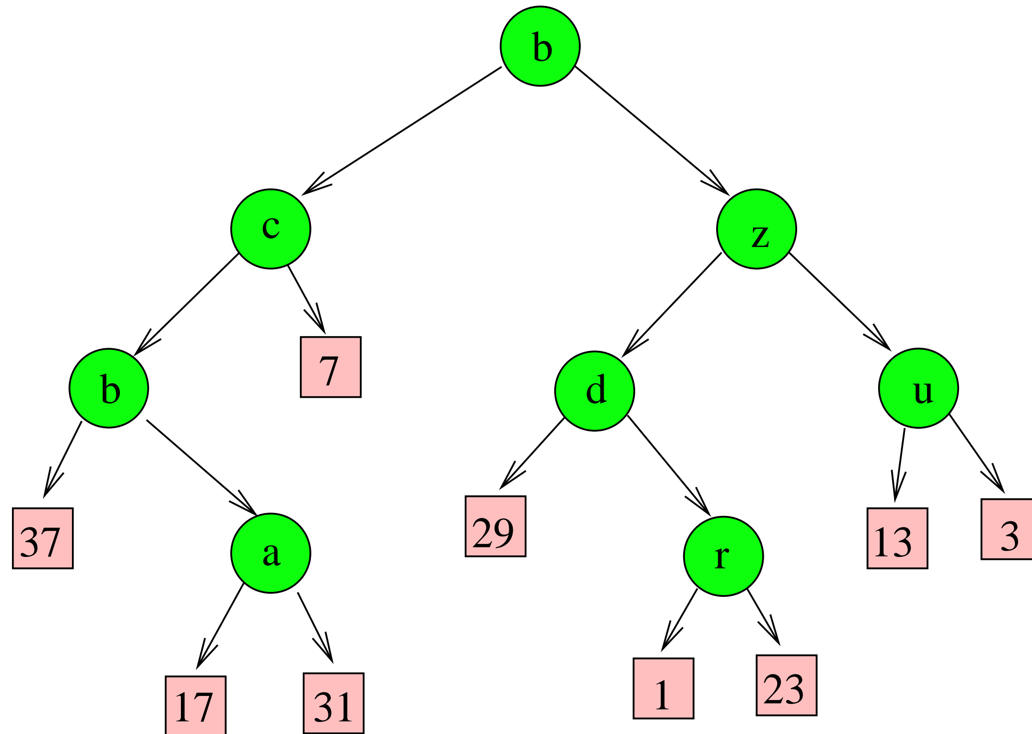
`e-visit`(z) ;

falls $T = (T_1, x, T_2)$:

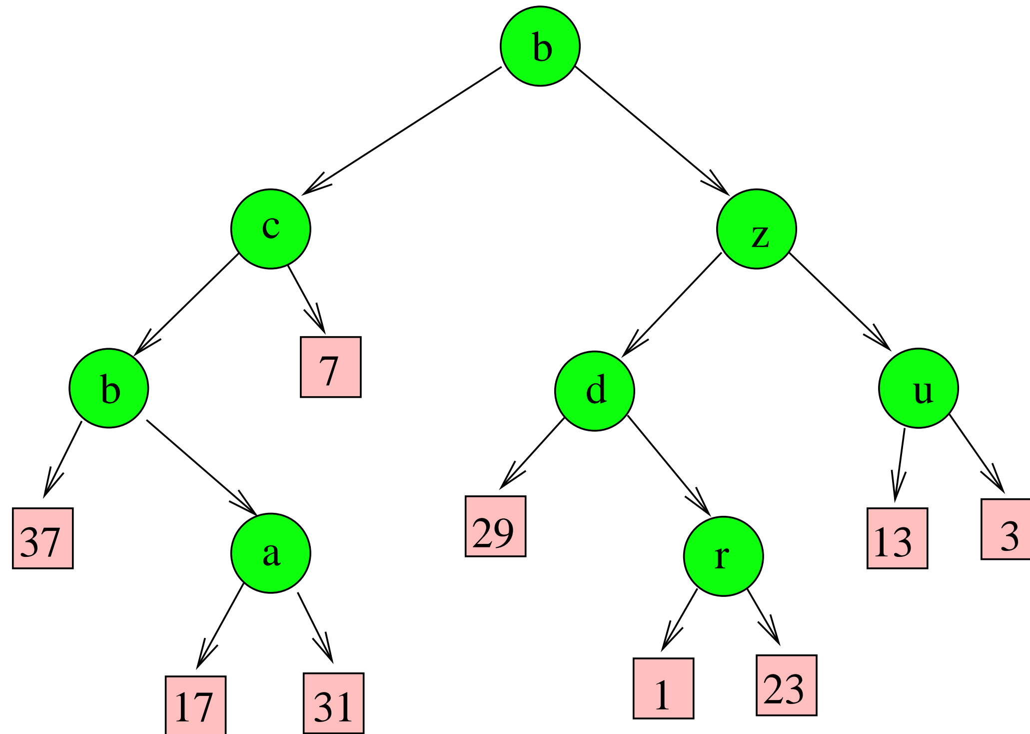
`i-visit`(x) ;
Präorder-Durchlauf durch T_1;
Präorder-Durchlauf durch T_2;

Präorder: „Erst die Wurzel, dann den linken Unterbaum, dann den rechten Unterbaum.“

Präorder-Durchlauf durch T

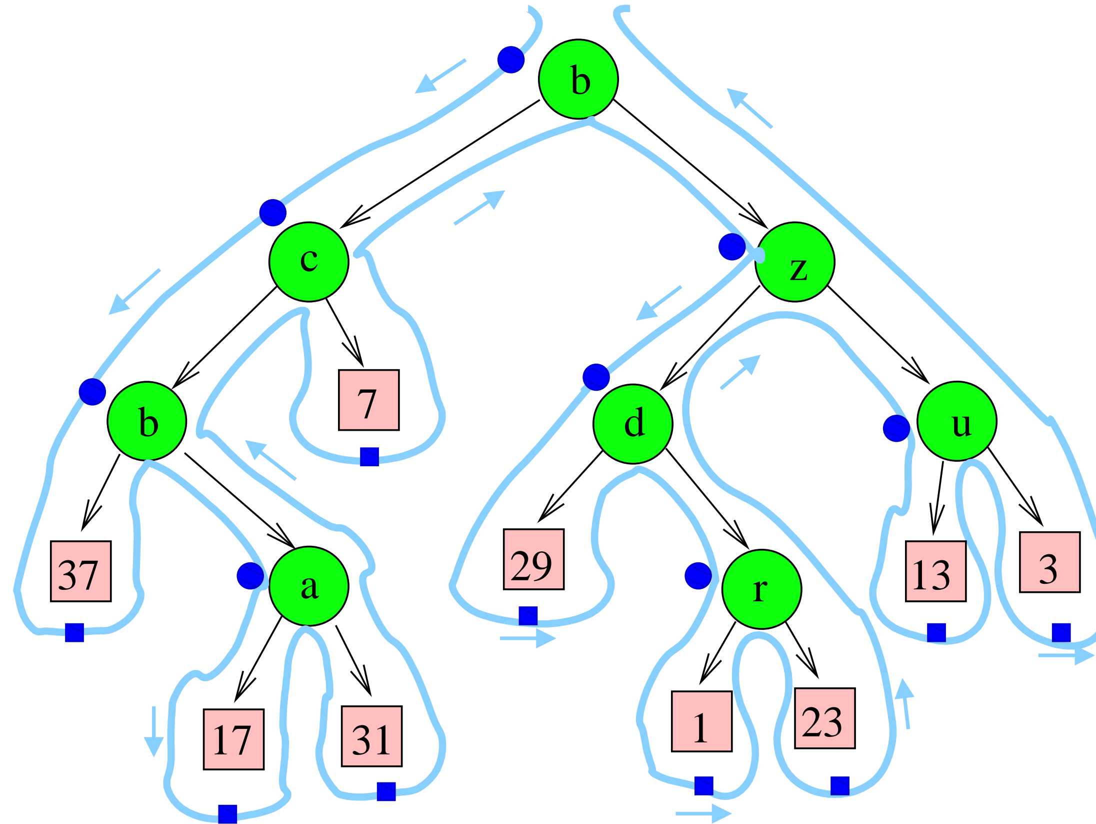


Präorder-Durchlauf durch T



Ausgabe: $\mathrm{b, c, b, 37, a, 17, 31, 7, z, d, 29, r, 1, 23, u, 13, 3}$

Präorder-Durchlauf durch T



Ausgabe: $\mathrm{b, c, b, 37, a, 17, 31, 7, z, d, 29, r, 1, 23, u, 13, 3}$

Postorder-Durchlauf durch T

Postorder-Durchlauf durch T

falls $T = \boxed{z}$:

```
    e-visit(z) ;
```

Postorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ;

falls $T = (T_1, x, T_2)$:

Postorder-Durchlauf durch T

falls $T = \boxed{z}$:

`e-visit`(z) ;

falls $T = (T_1, x, T_2)$:

Postorder-Durchlauf durch T_1;

Postorder-Durchlauf durch T

falls $T = \boxed{z}$:

```
e-visit(z) ;
```

falls $T = (T_1, x, T_2)$:

Postorder-Durchlauf durch T_1;
Postorder-Durchlauf durch T_2;

Postorder-Durchlauf durch T

falls $T = \boxed{z}$:
  `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
  Postorder-Durchlauf durch T_1;
  Postorder-Durchlauf durch T_2;
  `i-visit`(x) ;

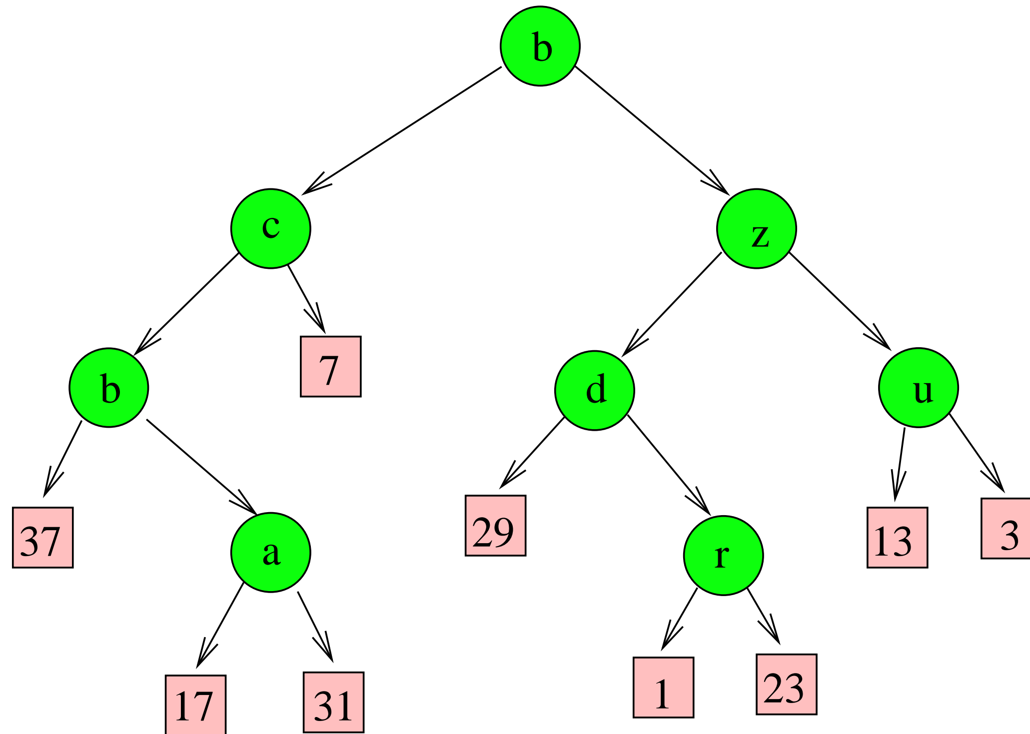
Postorder-Durchlauf durch T

falls $T = \boxed{z}$:
  `e-visit`(z) ;

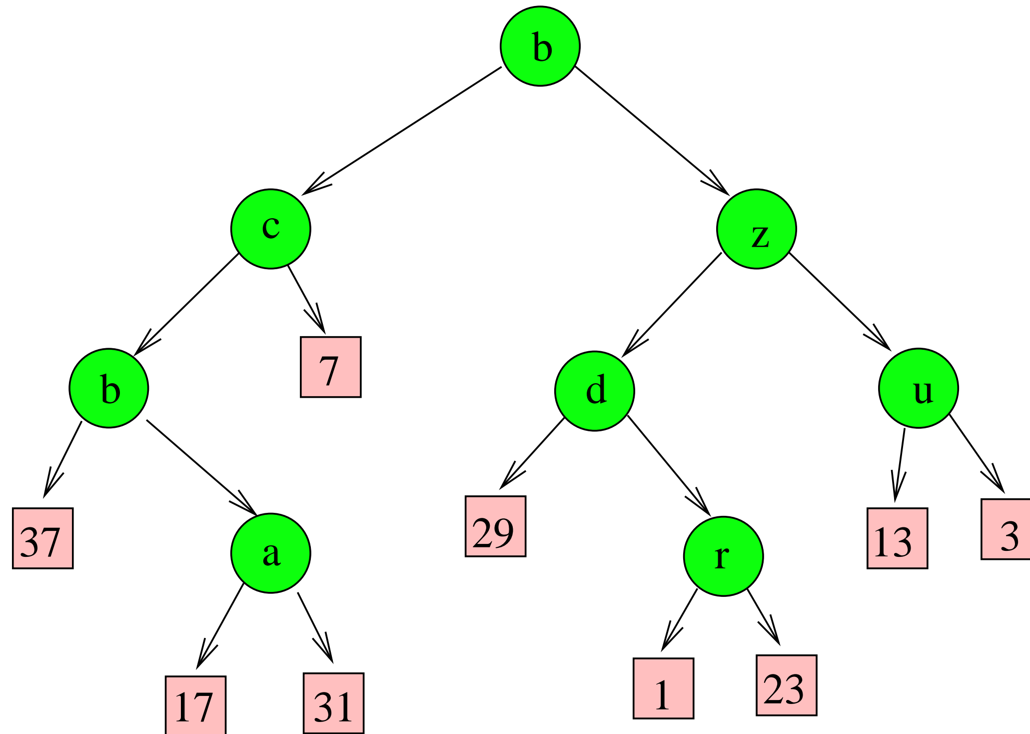
falls $T = (T_1, x, T_2)$:
  Postorder-Durchlauf durch T_1;
  Postorder-Durchlauf durch T_2;
  `i-visit`(x) ;

Postorder: „Erst den linken Unterbaum, dann den rechten Unterbaum, zuletzt die Wurzel.“

Postorder-Durchlauf durch T

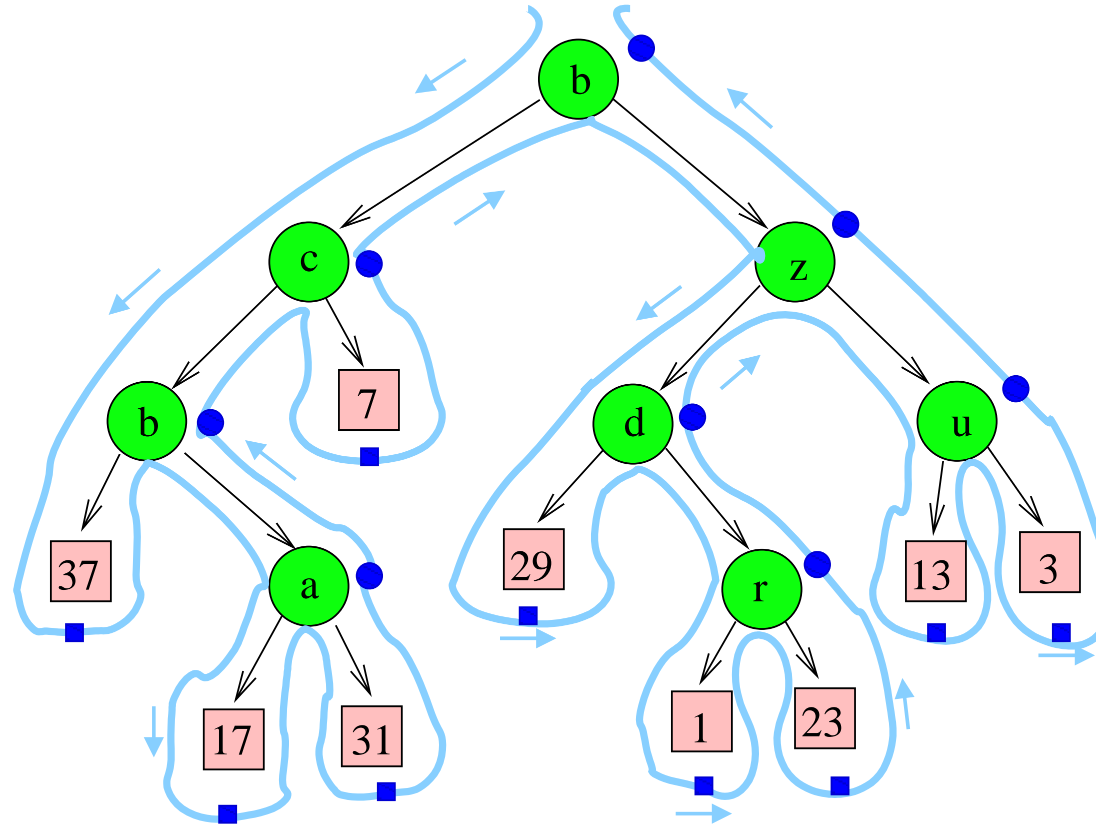


Postorder-Durchlauf durch T



Ausgabe: $37, 17, 31, \mathrm{a}, \mathrm{b}, 7, \mathrm{c}, 29, 1, 23, \mathrm{r}, \mathrm{d}, 13, 3, \mathrm{u}, \mathrm{z}, \mathrm{b}.$

Postorder-Durchlauf durch T



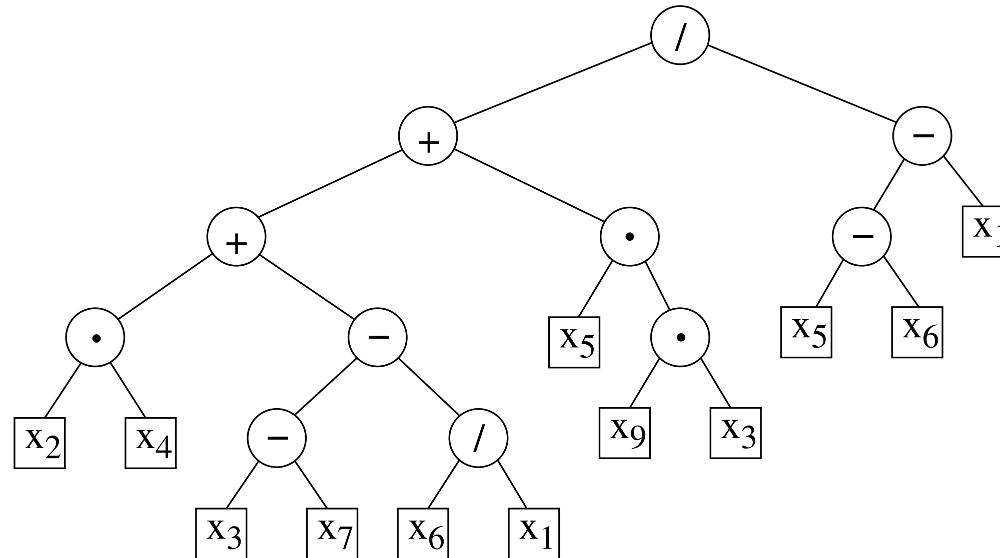
Ausgabe: 37, 17, 31, a, b, 7, c, 29, 1, 23, r, d, 13, 3, u, z, b.

Beispiel: **Arithmetischer Ausdruck als Baum**

$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.

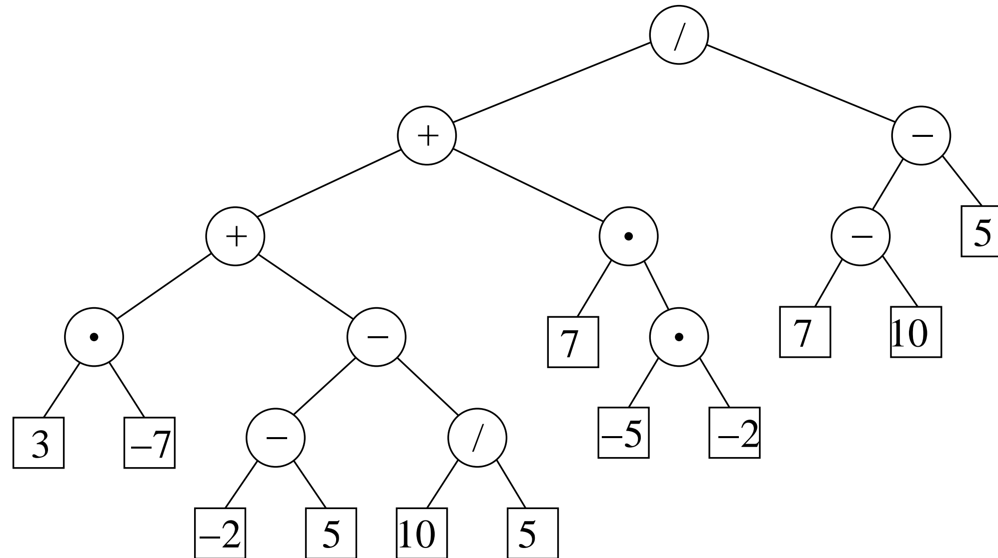
Beispiel: **Arithmetischer Ausdruck als Baum**

$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Beispiel: **Arithmetischer Ausdruck als Baum**

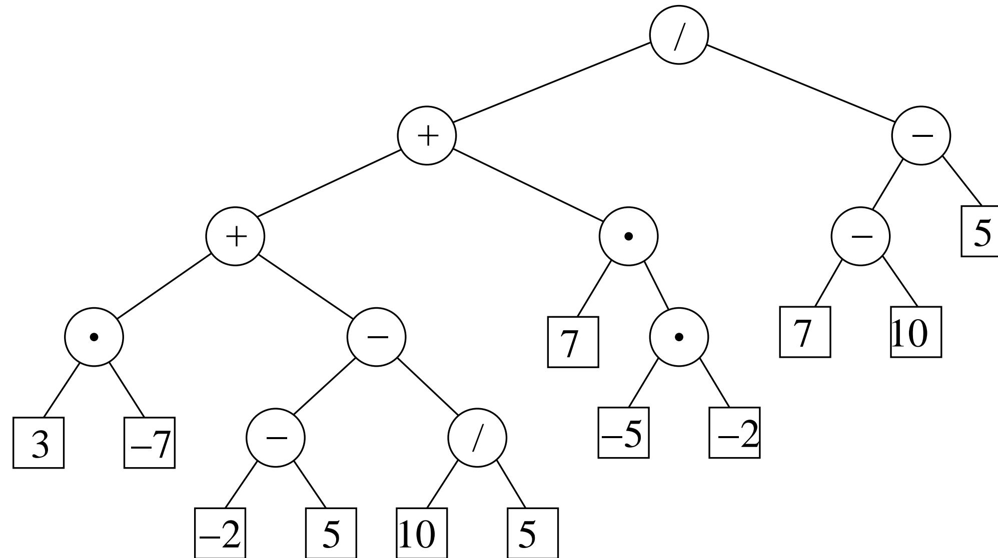
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

Beispiel: **Arithmetischer Ausdruck als Baum**

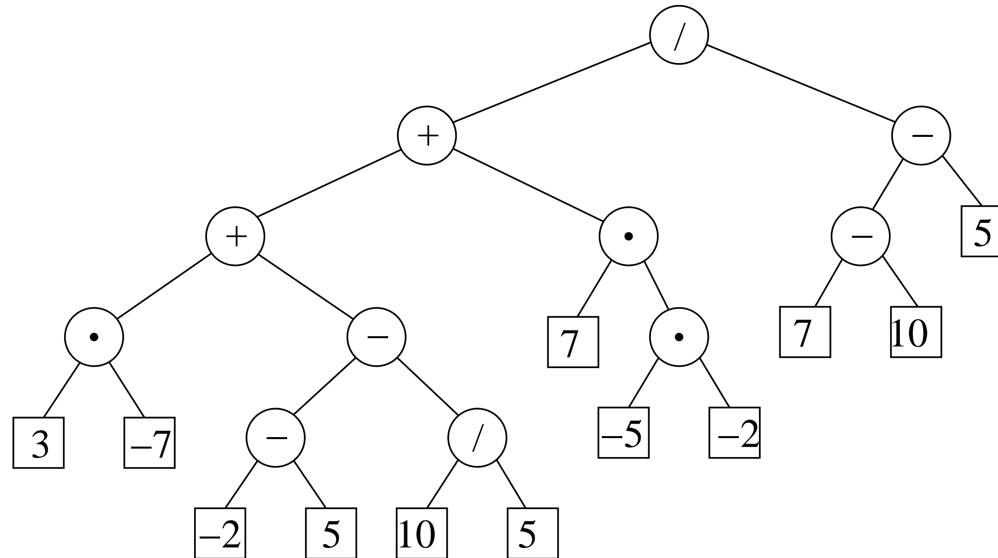
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.)

Beispiel: **Arithmetischer Ausdruck als Baum**

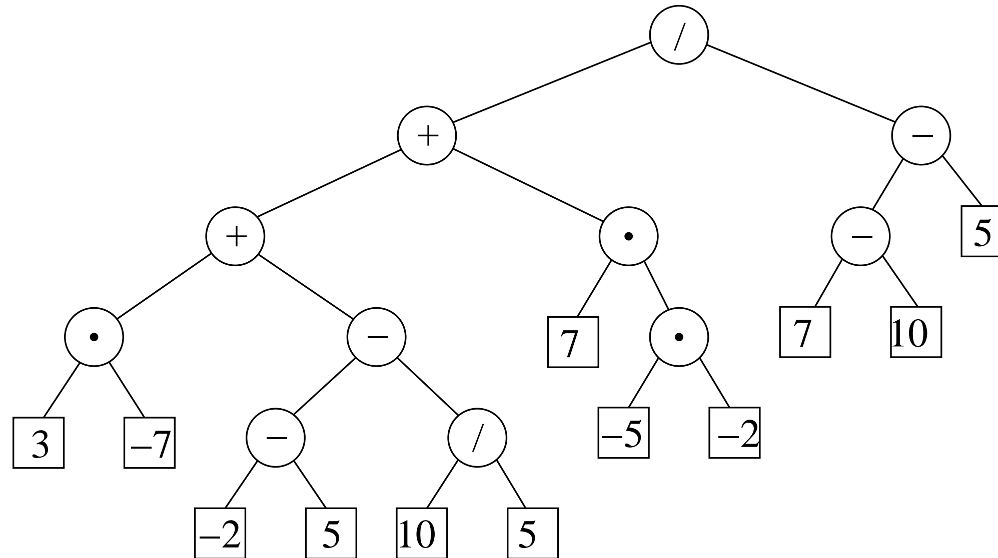
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.)
Geeignete Reihenfolge:

Beispiel: **Arithmetischer Ausdruck als Baum**

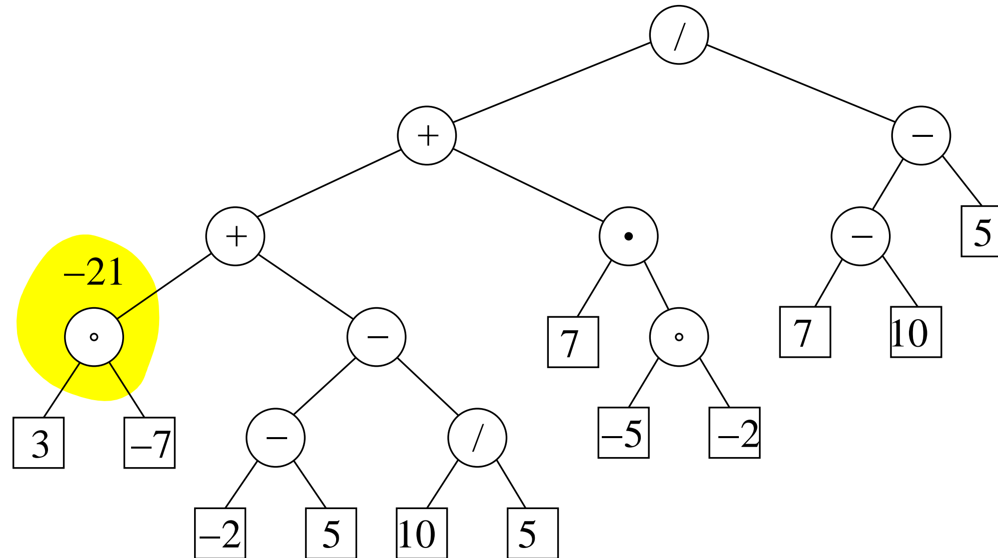
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

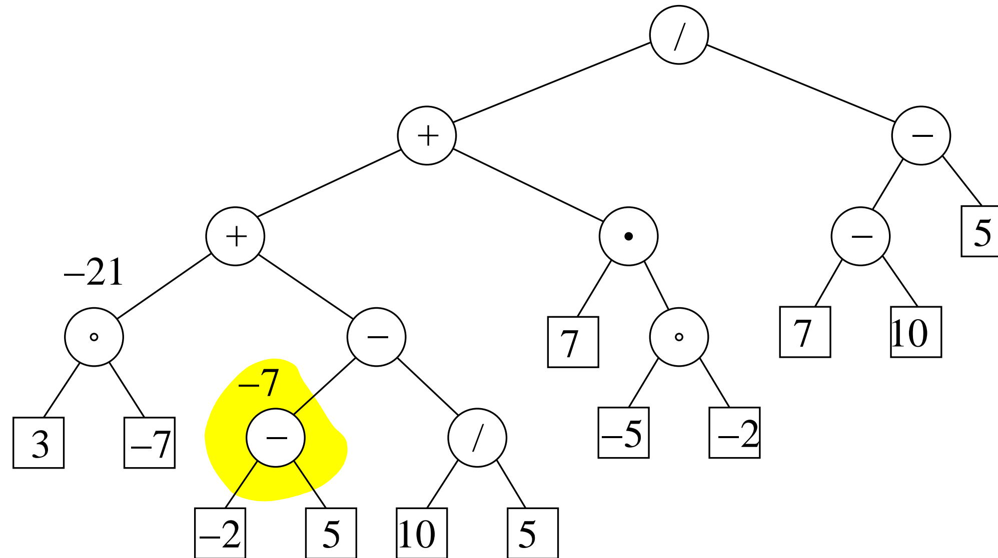
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

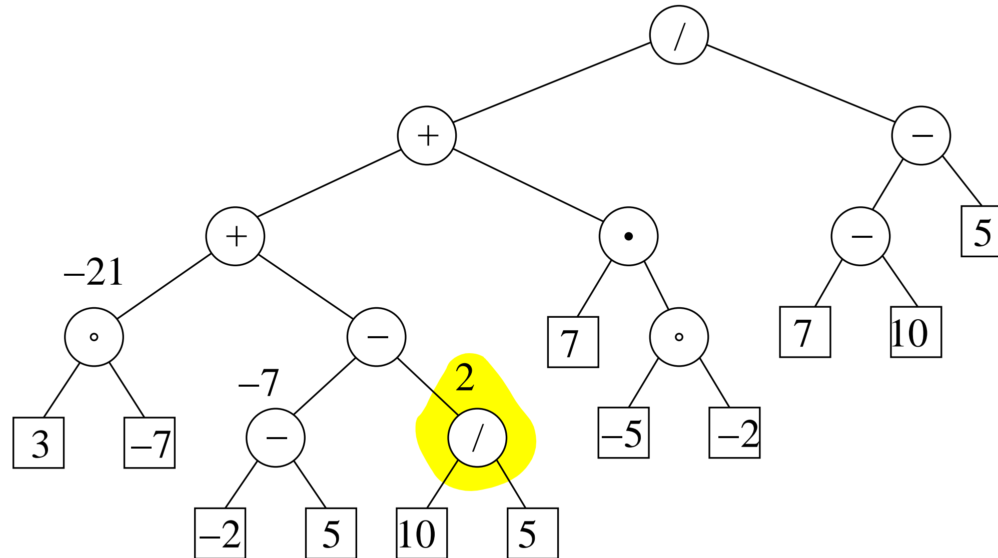
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

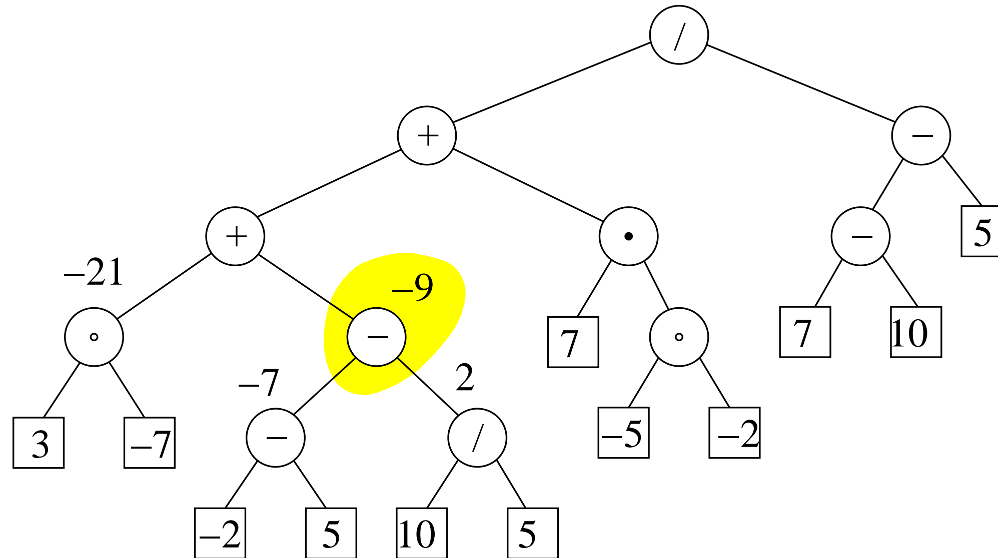
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

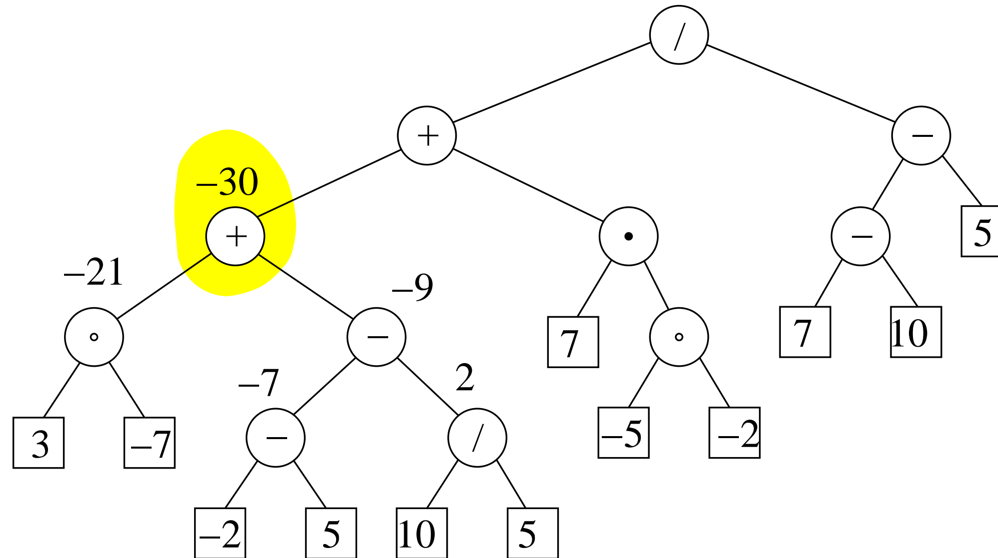
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

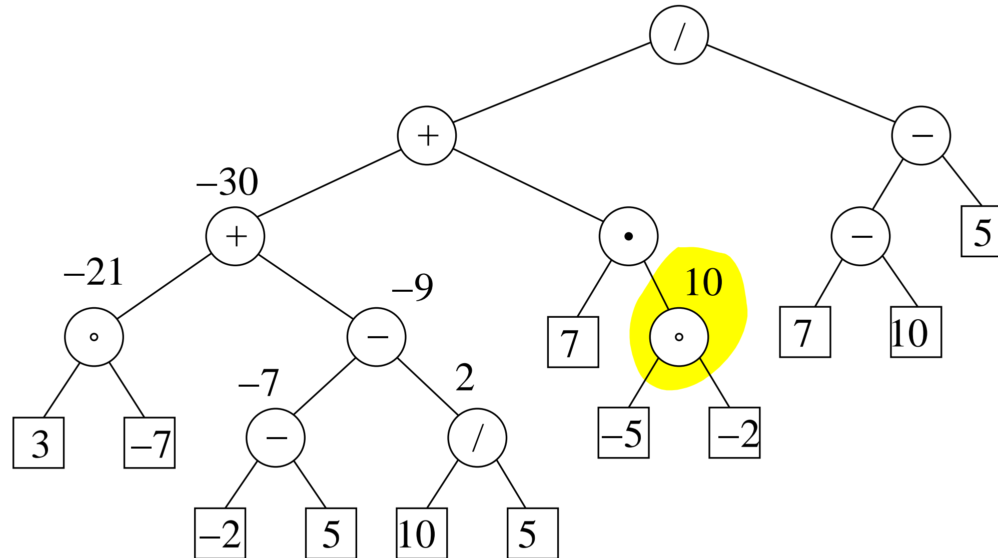
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

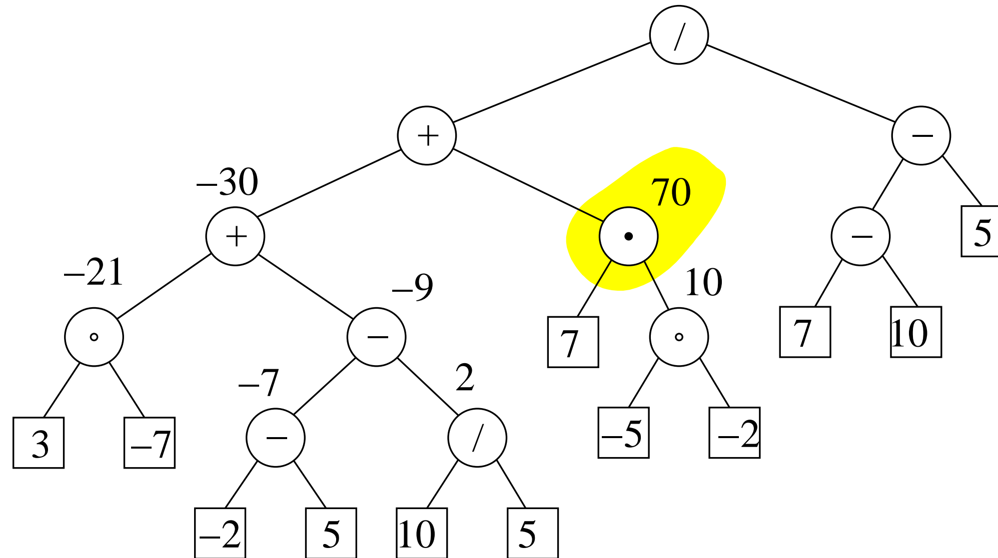
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

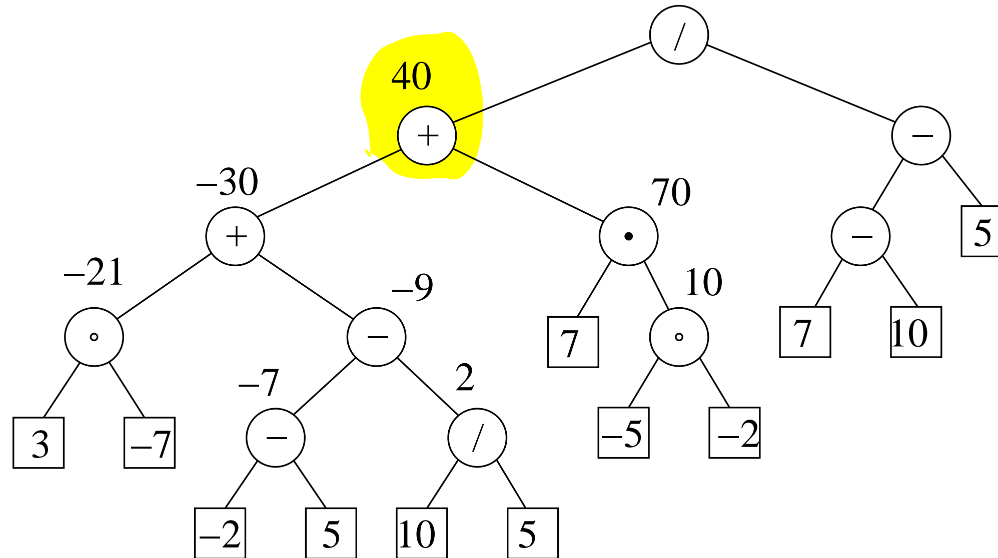
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.)
Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

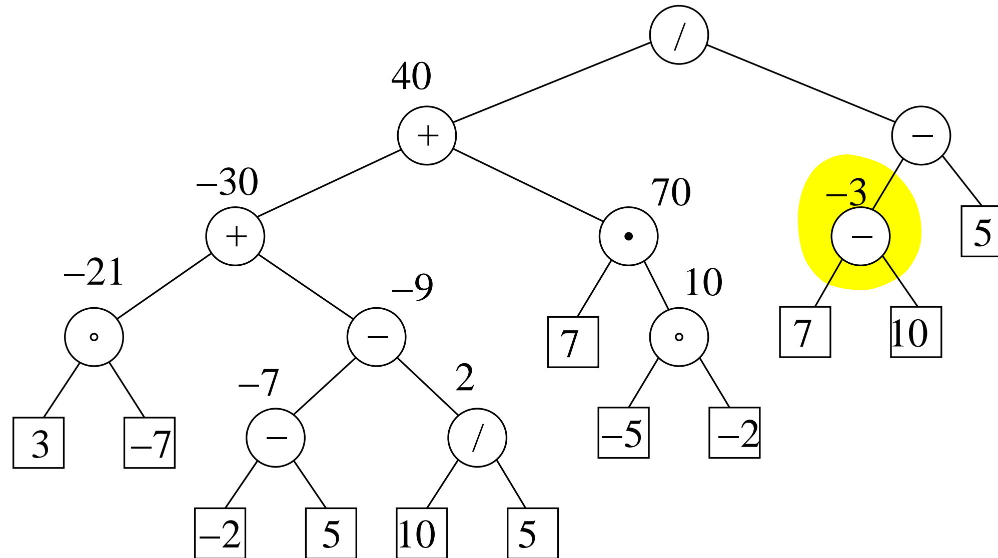
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

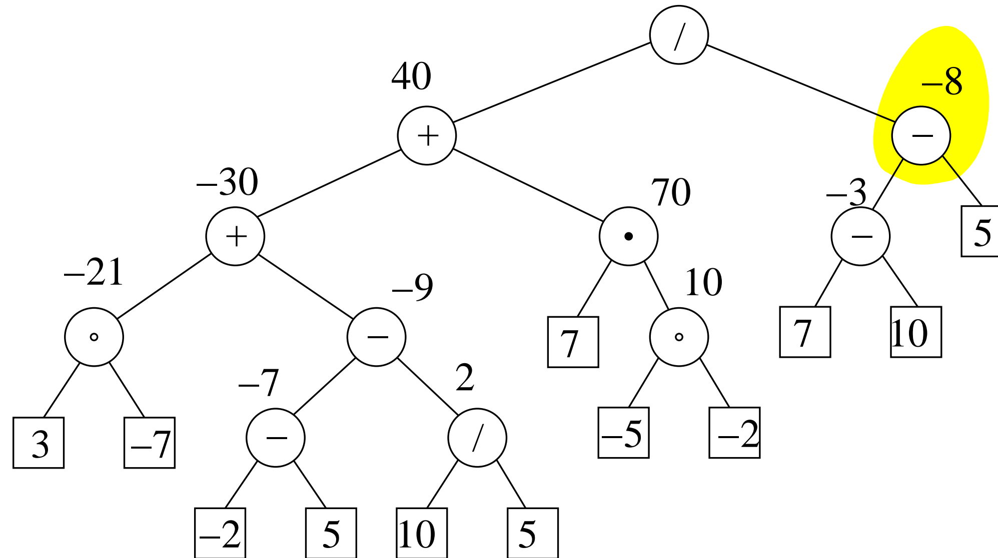
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

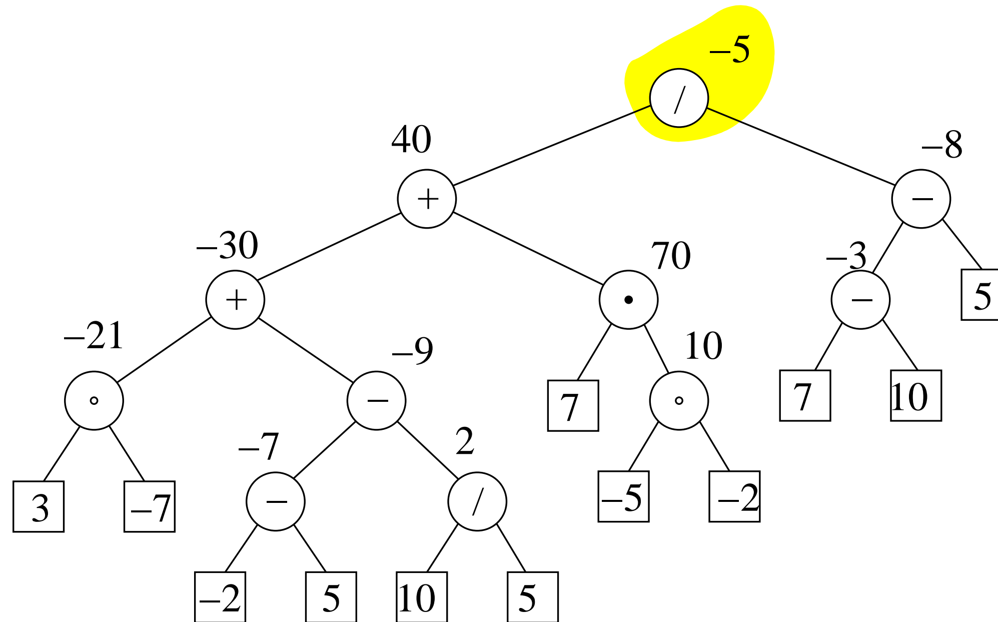
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

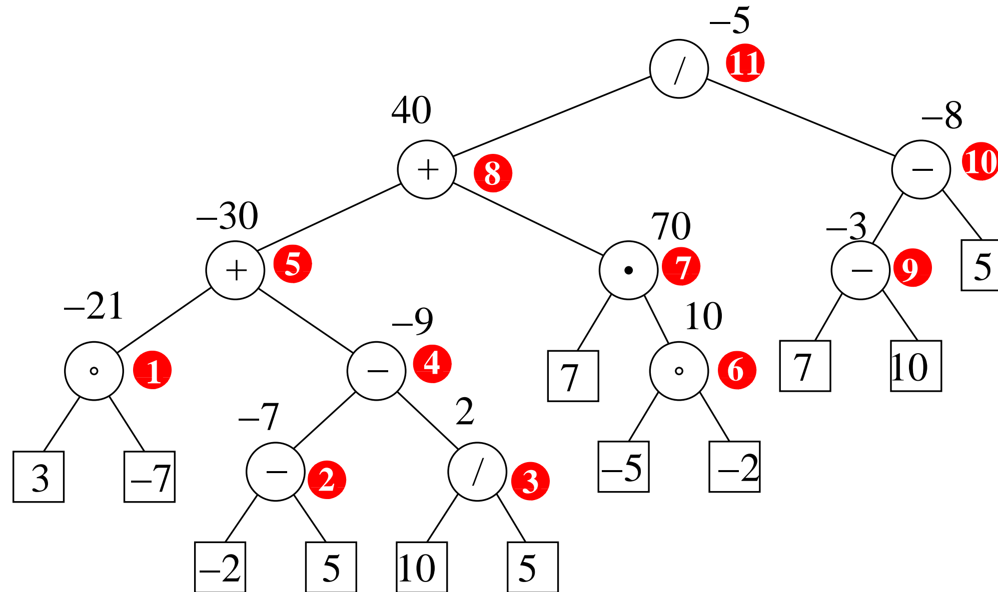
$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf.

Beispiel: **Arithmetischer Ausdruck als Baum**

$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.) Geeignete Reihenfolge: Postorder-Durchlauf. **(Rote Kreise.)**

Auch möglich: Kombi-Durchlauf:

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
  $\texttt{e-visit}(z)$;

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:

$\texttt{e-visit}(z)$;

falls $T = (T_1, x, T_2)$:

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
    $\texttt{e-visit}(z)$;

falls $T = (T_1, x, T_2)$:
    $\texttt{preord-i-visit}(x)$;

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
   `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
   `preord-i-visit`(x) ;
   Kombi-Durchlauf durch T_1;

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
    `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
    `preord-i-visit`(x) ;
    Kombi-Durchlauf durch T_1;
    `inord-i-visit`(x) ;

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
  `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
  `preord-i-visit`(x) ;
  Kombi-Durchlauf durch T_1;
  `inord-i-visit`(x) ;
  Kombi-Durchlauf durch T_2;

Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
  `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
  `preord-i-visit`(x) ;
  Kombi-Durchlauf durch T_1;
  `inord-i-visit`(x) ;
  Kombi-Durchlauf durch T_2;
  `postord-i-visit`(x) ;

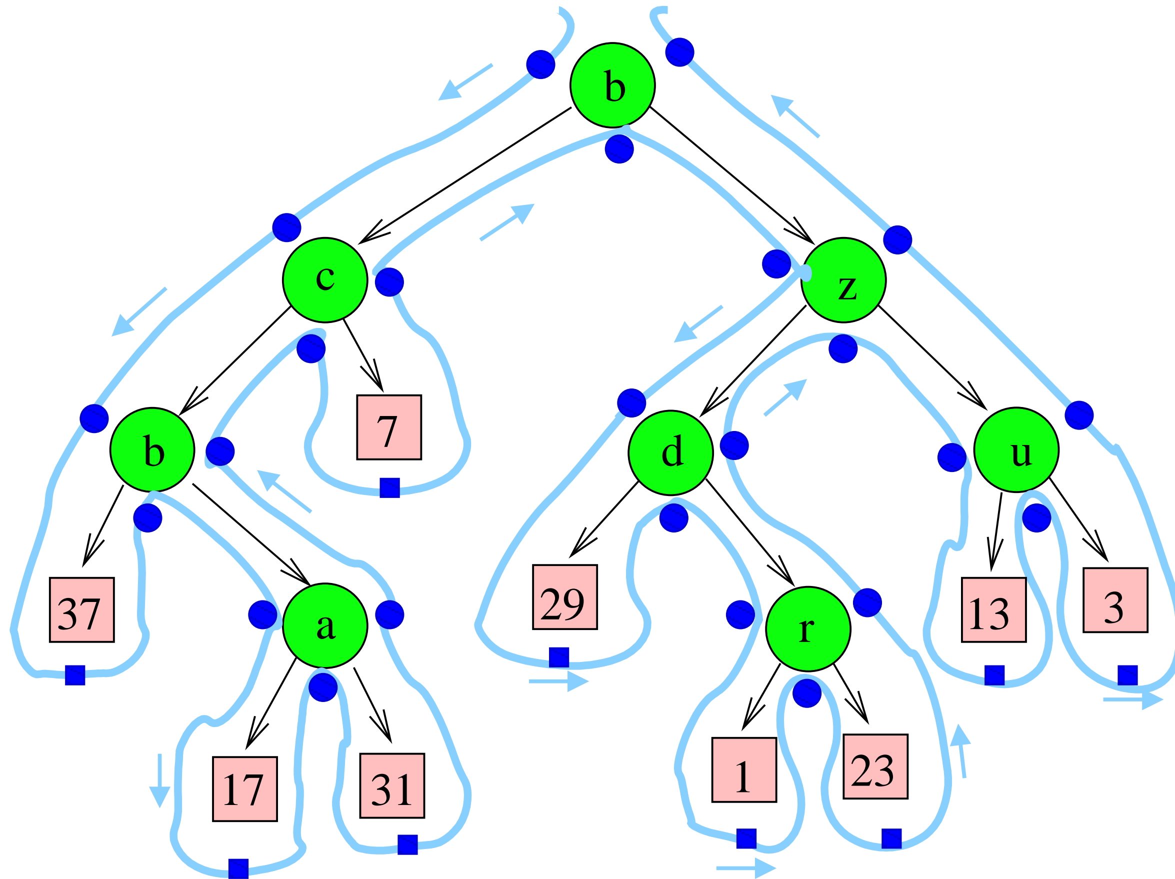
Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
  `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
  `preord-i-visit`(x) ;
  Kombi-Durchlauf durch T_1;
  `inord-i-visit`(x) ;
  Kombi-Durchlauf durch T_2;
  `postord-i-visit`(x) ;

Ermöglicht Datentransport von der Wurzel nach unten in den Baum hinein und wieder zurück.

Kombi-Durchlauf durch T



Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

$\texttt{e-visit}(z)$:

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z): **print**("(", z, ")").

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z): **print**("(",z,")").

`preord-i-visit`(x):

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z): **print**("(",z,")").

`preord-i-visit`(x): **print**("(").

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**").

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z):	**print**("(",z,")").
`preord-i-visit`(x):	**print**("(").
`inord-i-visit`(x):	

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z):	**print**("(", z, ")").
`preord-i-visit`(x):	**print**("(").
`inord-i-visit`(x):	**print**(",", x, ",").

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z):	**print**("(",z,")").
`preord-i-visit`(x):	**print**("(").
`inord-i-visit`(x):	**print**(",",x,",").
`postord-i-visit`(x):	

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z):	**print**("(",z,")").
`preord-i-visit`(x):	**print**("(").
`inord-i-visit`(x):	**print**(",",x,",").
`postord-i-visit`(x):	**print**(")").

Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z):	**print**("(",z,")").
`preord-i-visit`(x):	**print**("(").
`inord-i-visit`(x):	**print**(",",x,",").
`postord-i-visit`(x):	**print**(")").

Man sollte dies an einem Beispiel testen.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Für jeden inneren und äußeren Knoten wird die Prozedur `inorder(.)` einmal aufgerufen,

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Für jeden inneren und äußeren Knoten wird die Prozedur `inorder(.)` einmal aufgerufen, insgesamt $(2n+1)$-mal.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Für jeden inneren und äußeren Knoten wird die Prozedur `inorder(.)` einmal aufgerufen, insgesamt $(2n+1)$-mal.

Kosten pro Aufruf: $O(1)$ plus Kosten für `i-visit`/`e-visit`-Operation.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Für jeden inneren und äußeren Knoten wird die Prozedur `inorder(.)` einmal aufgerufen, insgesamt $(2n+1)$-mal.

Kosten pro Aufruf: $O(1)$ plus Kosten für `i-visit`/`e-visit`-Operation.

Insgesamt: $(2n+1) \cdot (O(1) + C_{\texttt{visit}})$,
wobei $C_{\texttt{visit}}$ eine Schranke für die Kosten der `visit`-Operationen ist.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Für jeden inneren und äußeren Knoten wird die Prozedur `inorder(.)` einmal aufgerufen, insgesamt $(2n+1)$-mal.

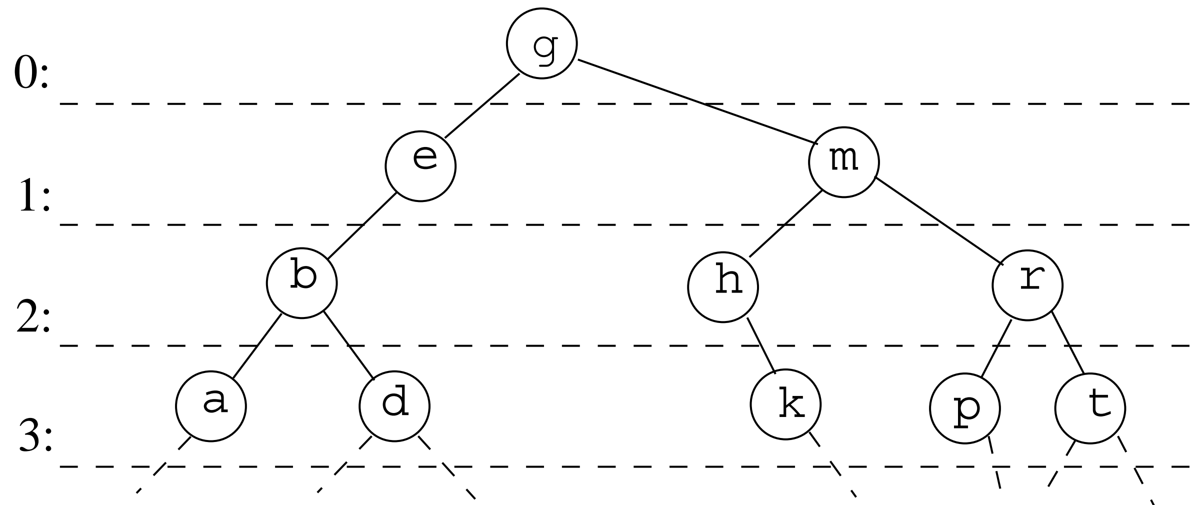
Kosten pro Aufruf: $O(1)$ plus Kosten für `i-visit`/`e-visit`-Operation.

Insgesamt: $(2n+1) \cdot (O(1) + C_{\texttt{visit}})$,
wobei $C_{\texttt{visit}}$ eine Schranke für die Kosten der `visit`-Operationen ist.

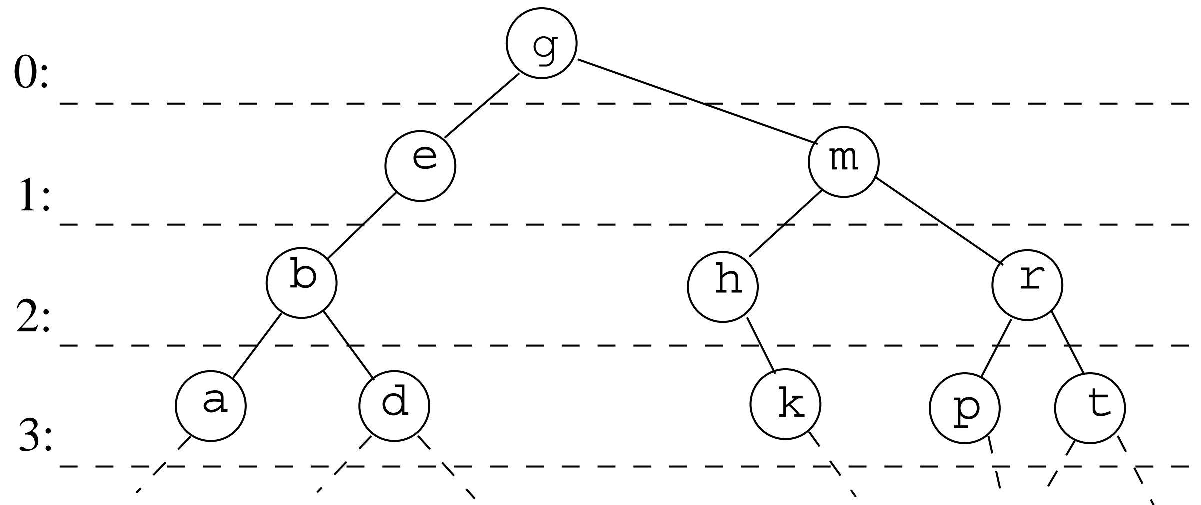
Behauptung 3.4.1

Baumdurchläufe haben lineare Laufzeit.

Levelorder-Durchlauf

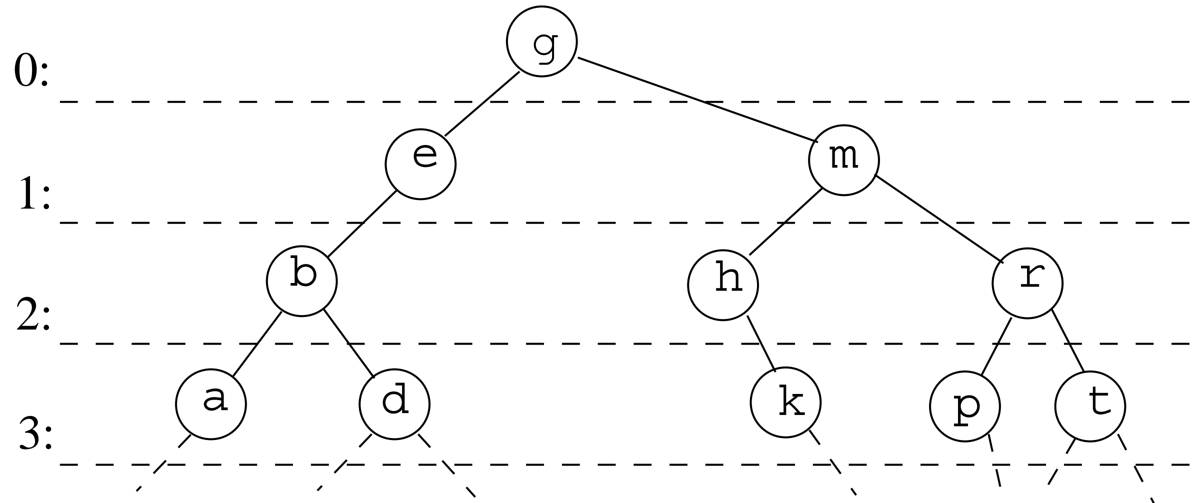


Levelorder-Durchlauf



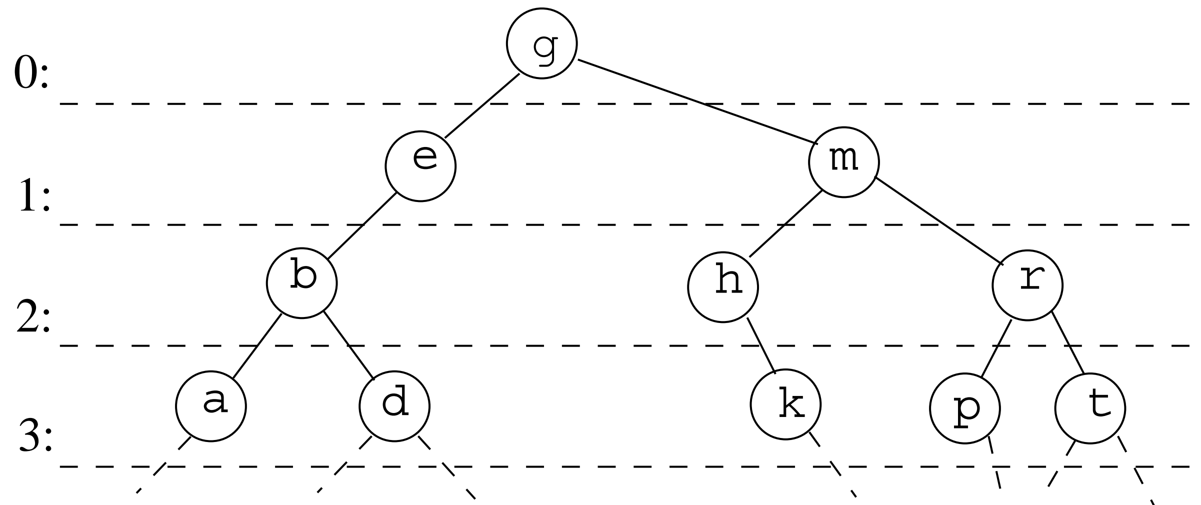
Wunschausgabe:

Levelorder-Durchlauf



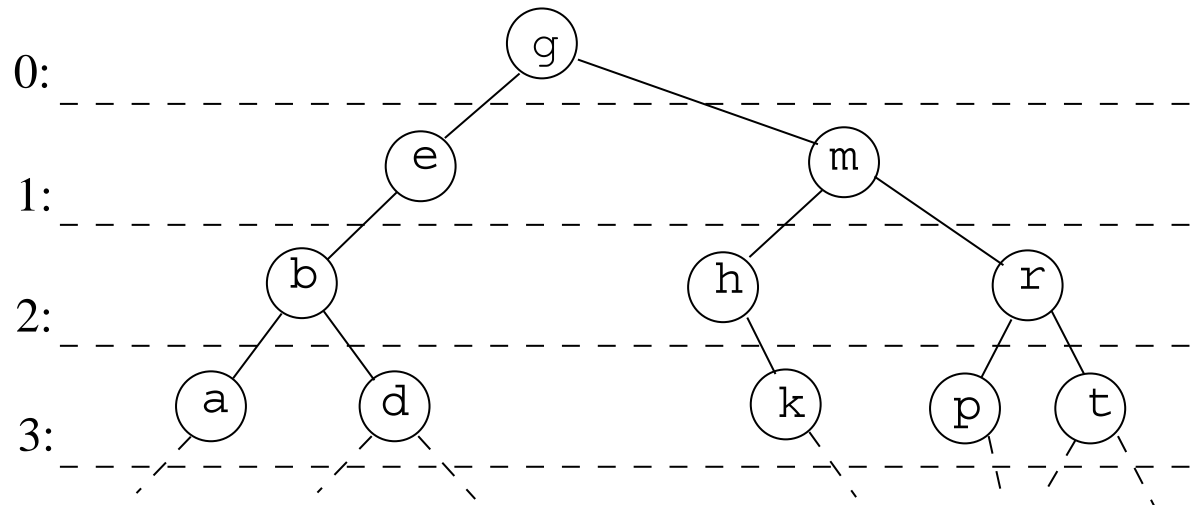
Wunschausgabe: `g em bhr adkpt`

Levelorder-Durchlauf



Wunschausgabe: `g em bhr adkpt` , „levelweise von links nach rechts“.

Levelorder-Durchlauf



Wunschausgabe: `g em bhr adkpt` , „levelweise von links nach rechts“.

Datenstruktur:
Queue `Q`, die Zeiger/Referenzen auf Binärbaumknoten speichern kann.

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)

Eingabe: v: node; // Wurzel von T

```
(1)   Q: Queue für node;
(2)   count ← 0;
(3)   if v ≠ NULL then // entdecke Wurzel
(4)      count++; v.lnum ← count;
(5)      v.level ← 0; enqueue(v);
(6)   while not Q.isempty do
(7)      w ← Q.first; Q.dequeue;
(8)      visit(w);   // bearbeite w
(9)      p ← w.leftchild;
(10)     if p ≠ NULL then
(11)        count++; p.lnum ← count; // entdecke p
(12)        p.level ← w.level + 1; enqueue(p);
(13)     p ← w.rightchild;
(14)     if p ≠ NULL then
(15)        count++; p.lnum ← count; // entdecke p
(16)        p.level ← w.level + 1; enqueue(p);
```

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)

Eingabe: v: node; // Wurzel von T

Initialisierung

```
(1)   Q: Queue für node;
(2)   count ← 0;
(3)   if v ≠ NULL then // entdecke Wurzel
(4)     count++; v.lnum ← count;
(5)     v.level ← 0; enqueue(v);
(6)   while not Q.isempty do
(7)     w ← Q.first;  Q.dequeue;
(8)     visit(w);   // bearbeite w
(9)     p ← w.leftchild;
(10)    if p ≠ NULL then
(11)      count++; p.lnum ← count; // entdecke p
(12)      p.level ← w.level + 1; enqueue(p);
(13)    p ← w.rightchild;
(14)    if p ≠ NULL then
(15)      count++; p.lnum ← count; // entdecke p
(16)      p.level ← w.level + 1; enqueue(p);
```

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)

Eingabe: v: node; // Wurzel von T

```
(1)  Q: Queue für node;
(2)  count ← 0;
(3)  if v ≠ NULL then // entdecke Wurzel
(4)    count++; v.lnum ← count;
(5)    v.level ← 0; enqueue(v);
(6)  while not Q.isempty do
(7)    w ← Q.first; Q.dequeue;
(8)    visit(w);   // bearbeite w
(9)    p ← w.leftchild;
(10)   if p ≠ NULL then
(11)     count++; p.lnum ← count; // entdecke p
(12)     p.level ← w.level + 1; enqueue(p);
(13)   p ← w.rightchild;
(14)   if p ≠ NULL then
(15)     count++; p.lnum ← count; // entdecke p
(16)     p.level ← w.level + 1; enqueue(p);
```

entdecke Wurzel

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)
Eingabe: v: node; // Wurzel von T

```
(1)   Q: Queue für node;
(2)   count ← 0;
(3)   if v ≠ NULL then // entdecke Wurzel
(4)      count++; v.lnum ← count;
(5)      v.level ← 0; enqueue(v);
(6)   while not Q.isempty do                    Bearbeitungsschleife
(7)      w ← Q.first;  Q.dequeue;
(8)      visit(w);    // bearbeite w
(9)      p ← w.leftchild;
(10)     if p ≠ NULL then
(11)        count++; p.lnum ← count; // entdecke p
(12)        p.level ← w.level + 1; enqueue(p);
(13)     p ← w.rightchild;
(14)     if p ≠ NULL then
(15)        count++; p.lnum ← count; // entdecke p
(16)        p.level ← w.level + 1; enqueue(p);
```

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)

Eingabe: v: node; // Wurzel von T

```
(1)   Q: Queue für node;
(2)   count ← 0;
(3)   if v ≠ NULL then // entdecke Wurzel
(4)      count++; v.lnum ← count;
(5)      v.level ← 0; enqueue(v);
(6)   while not Q.isempty do
(7)      w ← Q.first; Q.dequeue;
(8)      visit(w);   // bearbeite w
(9)      p ← w.leftchild;
(10)     if p ≠ NULL then
(11)        count++; p.lnum ← count; // entdecke p
(12)        p.level ← w.level + 1; enqueue(p);
(13)     p ← w.rightchild;
(14)     if p ≠ NULL then
(15)        count++; p.lnum ← count; // entdecke p
(16)        p.level ← w.level + 1; enqueue(p);
```

Bearbeitung: prüfe linkes Kind

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)

Eingabe: v: node; // Wurzel von T

```
(1)   Q: Queue für node;
(2)   count ← 0;
(3)   if v ≠ NULL then // entdecke Wurzel
(4)     count++; v.lnum ← count;
(5)     v.level ← 0; enqueue(v);
(6)   while not Q.isempty do
(7)     w ← Q.first; Q.dequeue;
(8)     visit(w);   // bearbeite w
(9)     p ← w.leftchild;
(10)    if p ≠ NULL then
(11)      count++; p.lnum ← count; // entdecke p
(12)      p.level ← w.level + 1; enqueue(p);
(13)    p ← w.rightchild;
(14)    if p ≠ NULL then
(15)      count++; p.lnum ← count; // entdecke p
(16)      p.level ← w.level + 1; enqueue(p);
```

Bearbeitung: entdecke linkes Kind

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)
Eingabe: v: node; // Wurzel von T

```
(1)  Q: Queue für node;
(2)  count ← 0;
(3)  if v ≠ NULL then // entdecke Wurzel
(4)     count++; v.lnum ← count;
(5)     v.level ← 0; enqueue(v);
(6)  while not Q.isempty do
(7)     w ← Q.first; Q.dequeue;
(8)     visit(w);   // bearbeite w
(9)     p ← w.leftchild;
(10)    if p ≠ NULL then
(11)       count++; p.lnum ← count; // entdecke p
(12)       p.level ← w.level + 1; enqueue(p);
(13)    p ← w.rightchild;
(14)    if p ≠ NULL then
(15)       count++; p.lnum ← count; // entdecke p
(16)       p.level ← w.level + 1; enqueue(p);
```

Bearbeitung: prüfe rechtes Kind
entdecke es, falls vorhanden

Behauptung 3.4.2

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\mathtt{visit}} + O(n)$,
wobei $t_{\mathtt{visit}}$ die Zeit für `visit` angibt.

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\texttt{visit}} + O(n)$,
wobei $t_{\texttt{visit}}$ die Zeit für `visit` angibt.

(b) Der Levelorder-Durchlauf ordnet jedem Binärbaumknoten v (in v`.level`)
korrekt seine Tiefe zu.

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\texttt{visit}} + O(n)$, wobei $t_{\texttt{visit}}$ die Zeit für `visit` angibt.

(b) Der Levelorder-Durchlauf ordnet jedem Binärbaumknoten v (in v`.level`) korrekt seine Tiefe zu.

Beweis: Man zeigt durch Induktion über das Level von v:

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\texttt{visit}} + O(n)$, wobei $t_{\texttt{visit}}$ die Zeit für `visit` angibt.

(b) Der Levelorder-Durchlauf ordnet jedem Binärbaumknoten v (in v`.level`) korrekt seine Tiefe zu.

Beweis: Man zeigt durch Induktion über das Level von v:

Jeder innere Knoten v des Binärbaums wird genau einmal in die Queue **eingefügt**, genau einmal **entnommen** und im Schleifenrumpf `(7)-(16)` **bearbeitet**.

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\texttt{visit}} + O(n)$,
wobei $t_{\texttt{visit}}$ die Zeit für `visit` angibt.

(b) Der Levelorder-Durchlauf ordnet jedem Binärbaumknoten v (in v`.level`) korrekt seine Tiefe zu.

Beweis: Man zeigt durch Induktion über das Level von v:

Jeder innere Knoten v des Binärbaums wird genau einmal in die Queue **eingefügt**, genau einmal **entnommen** und im Schleifenrumpf `(7)-(16)` **bearbeitet**.
Dabei wird die korrekte Levelnummer vergeben.

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\texttt{visit}} + O(n)$,
wobei $t_{\texttt{visit}}$ die Zeit für `visit` angibt.

(b) Der Levelorder-Durchlauf ordnet jedem Binärbaumknoten v (in v`.level`)
korrekt seine Tiefe zu.

Beweis: Man zeigt durch Induktion über das Level von v:

Jeder innere Knoten v des Binärbaums wird genau einmal in die Queue **eingefügt**,
genau einmal **entnommen** und im Schleifenrumpf `(7)-(16)` **bearbeitet**.
Dabei wird die korrekte Levelnummer vergeben.

Weil Queue-Operationen Zeit $O(1)$ benötigen, ist der restliche Zeitbedarf $O(n)$. □

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann.

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt.

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes" und ein „rechtes" Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren" Kindplatzes).

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:

(i) Für $x \in D$ ist (x) ein Mw-D-Baum.

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:

(i) Für $x \in D$ ist (x) ein Mw-D-Baum.

(ii) Wenn $d \geq 1$ ist und $T_1, \ldots, T_d$ Mw-D-Bäume sind und $x \in D$ ist,
dann ist auch $(x, T_1, \ldots, T_d)$ ein Mw-D-Baum.

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:

(i) Für $x \in D$ ist (x) ein Mw-D-Baum.

(ii) Wenn $d \geq 1$ ist und $T_1, \ldots, T_d$ Mw-D-Bäume sind und $x \in D$ ist,
dann ist auch $(x, T_1, \ldots, T_d)$ ein Mw-D-Baum.

Implementierung solcher Bäume: Ein Knoten enthält einen Dateneintrag aus D und eine (lineare) Liste oder ein Array von Zeigern auf Kindknoten.

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:

(i) Für $x \in D$ ist (x) ein Mw-D-Baum.

(ii) Wenn $d \geq 1$ ist und $T_1, \ldots, T_d$ Mw-D-Bäume sind und $x \in D$ ist, dann ist auch $(x, T_1, \ldots, T_d)$ ein Mw-D-Baum.

Implementierung solcher Bäume: Ein Knoten enthält einen Dateneintrag aus D und eine (lineare) Liste oder ein Array von Zeigern auf Kindknoten. In [AuP], Kap. 10, wird skizziert, wie sich solche Bäume mit Hilfe von Binärbaumknoten darstellen lassen.

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:

(i) Für $x \in D$ ist (x) ein Mw-D-Baum.

(ii) Wenn $d \geq 1$ ist und $T_1, \ldots, T_d$ Mw-D-Bäume sind und $x \in D$ ist,
dann ist auch $(x, T_1, \ldots, T_d)$ ein Mw-D-Baum.

Implementierung solcher Bäume: Ein Knoten enthält einen Dateneintrag aus D und eine (lineare) Liste oder ein Array von Zeigern auf Kindknoten. In [AuP], Kap. 10, wird skizziert, wie sich solche Bäume mit Hilfe von Binärbaumknoten darstellen lassen.

Für solche Mehrwegbäume sind Präorder- und Postorder-Durchlauf sowie Levelorder-Durchlauf sinnvoll (und benötigen lineare Zeit).

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes" und ein „rechtes" Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren" Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D":

(i) Für $x \in D$ ist (x) ein Mw-D-Baum.

(ii) Wenn $d \geq 1$ ist und $T_1, \ldots, T_d$ Mw-D-Bäume sind und $x \in D$ ist, dann ist auch $(x, T_1, \ldots, T_d)$ ein Mw-D-Baum.

Implementierung solcher Bäume: Ein Knoten enthält einen Dateneintrag aus D und eine (lineare) Liste oder ein Array von Zeigern auf Kindknoten. In [AuP], Kap. 10, wird skizziert, wie sich solche Bäume mit Hilfe von Binärbaumknoten darstellen lassen.

Für solche Mehrwegbäume sind Präorder- und Postorder-Durchlauf sowie Levelorder-Durchlauf sinnvoll (und benötigen lineare Zeit).

Uns werden solche Bäume später als **Mehrweg-Suchbäume** und im Kontext von **Graphdurchläufen** („Breitensuchbaum", „Tiefensuchbaum") begegnen.

ENDE

3. Kapitel