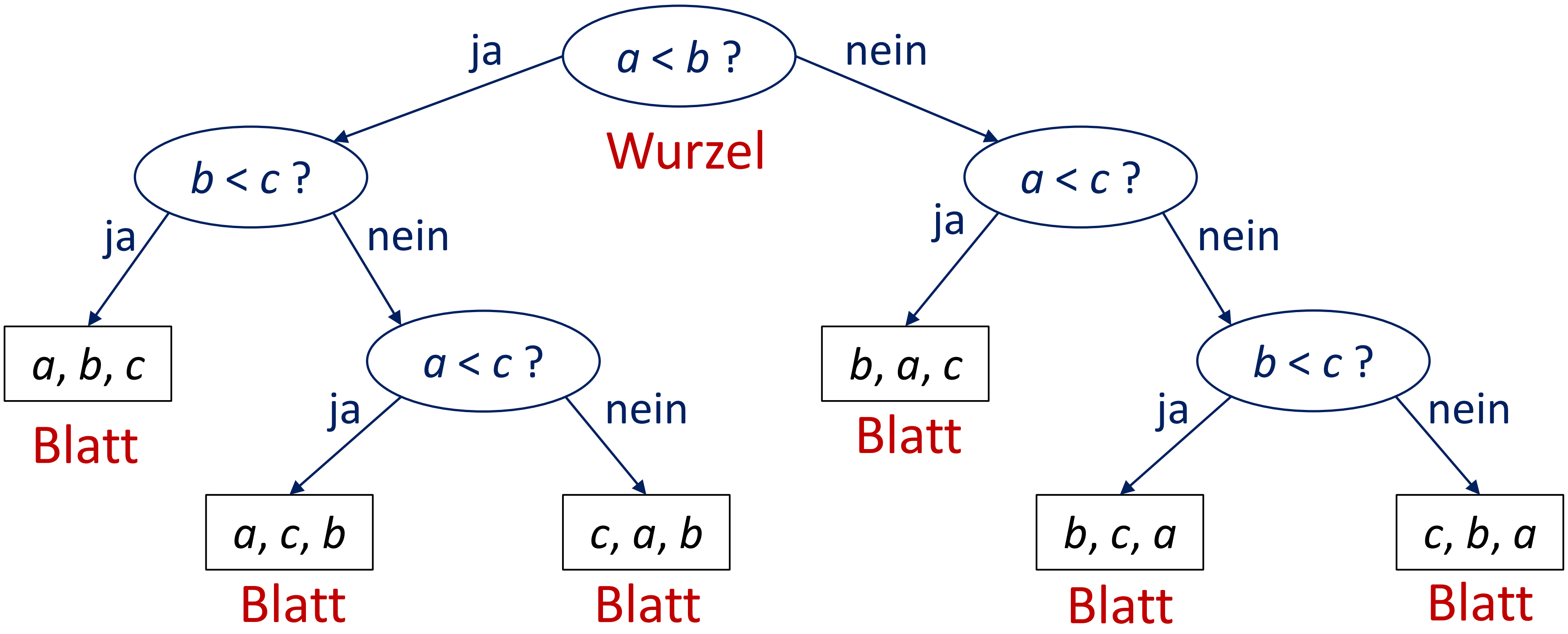
Algorithmen und Datenstrukturen

Bäume

Entscheidungsbäume und Sortieralgorithmen

Entscheidungsbaum (*intuitiver Algorithmus*)

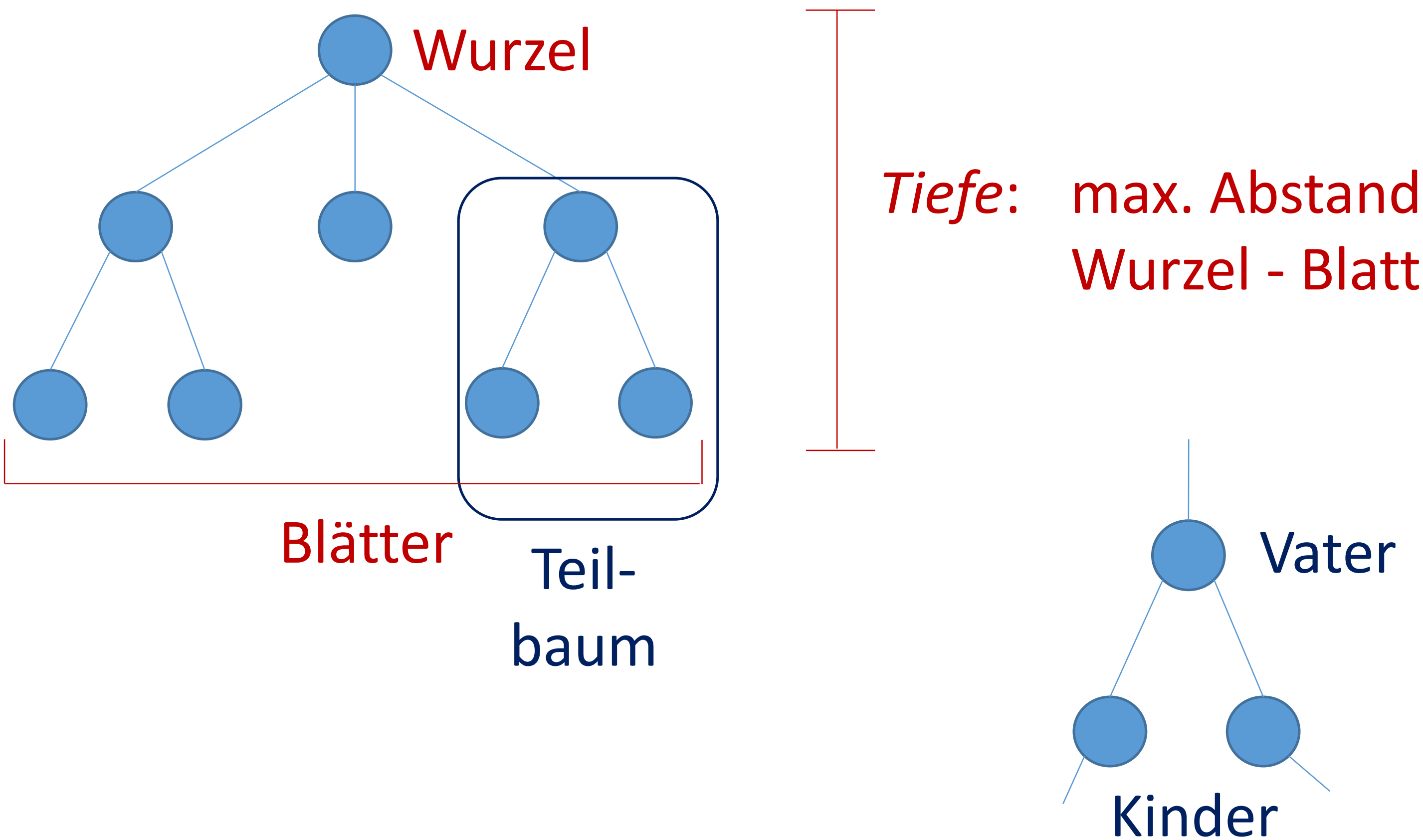
Folge [*a*, *b*, *c*] – Vergleiche:



Entscheidungsbaum und Komplexität

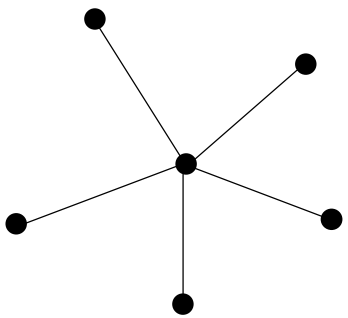
- Für eine Folge der Länge 3:
  - Es gibt sechs verschiedene Anordnungen (3! = 6)
  - Für jede Anordnung sind zwei bis maximal drei Vergleiche erforderlich.
- Für eine Folge der Länge *n:*
  - Es gibt *n*! verschiedene Anordnungen.
    → *n*! viele Blätter
  - Die Anzahl der Entscheidungen ist für jede Anordnung der Abstand des Blattes von der Wurzel.
    → **maximal:** die **Tiefe** des Baums
    (*die Anzahl der „Ebenen“* - 1)

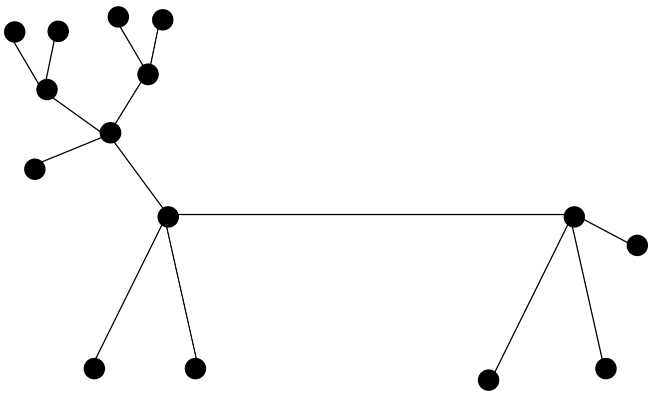
Baum anschaulich



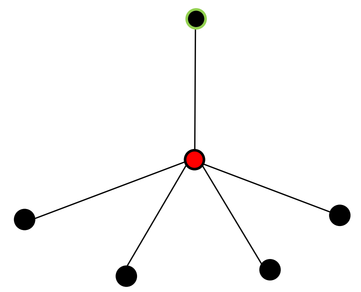
Definition Baum

- Ein ungerichteter zusammenhängender, kreisfreier Graph heißt (ungerichteter) **Baum**.

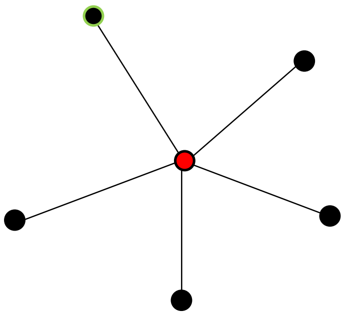




Gewurzelter Baum



- Konstruieren aus einem Baum *T* einen gerichteten Graphen:
  1. Wähle einen beliebigen Knoten *r* von *T*. Markiere *r*.
  2. Kanten {*u*,*v*}, wobei *u* markiert und *v* nicht markiert ist, werden zur gerichteten Kante (*u*,*v*). Dann markiere *v*. Dann heißt *u* **Vater** von *v* und *v* **Kind** von *u*.
  3. Wiederhole 2. bis alle Knoten markiert sind.
- Der so entstandene gerichtete Graph ist der zu *T* gehörige **Baum mit Wurzel *r*.**
- Er wird aber oft ungerichtet gezeichnet.
- Ein Knoten ohne Kinder heißt **Blatt**.
  Alle anderen Knoten heißen **innere Knoten**.



Tiefe

- Sei T ein Baum mit Wurzel r.

- Die Tiefe eines Knotens u in T, **tiefe(u,T)**, ist der Abstand von der Wurzel r zu u .

- Die Tiefe des Baums, **tiefe(T)**, ist der größte Abstand von r zu einem Blatt:

$$\mathbf{tiefe}(T) = \mathbf{max}\ \{\ \mathbf{tiefe}(v,T) : v \text{ ist Blatt in } T\ \}$$

Teilbäume, Wälder

- Jeder isolierte Knoten ist ein Baum der Tiefe 0.
- In jedem gewurzelten Baum gilt:
  - Die Wurzel hat keinen Vater, Blätter haben keine Kinder.
  - Jeder Knoten ist Wurzel eines **Teilbaums**,
    nämlich des gerichteten Teilgraphen mit allen
    von diesem Knoten erreichbaren Knoten.
- Ein kreisfreier Graph heißt **Wald**.
  Seine Zusammenhangskomponenten sind Bäume.

Elementare Eigenschaften

Satz: Sei G = (V,E) ein ungerichteter Graph.
Folgende Eigenschaften sind paarweise äquivalent.

1. G ist ein Baum.
2. Für jedes Paar (u,v) von Knoten von G gibt es genau einen *elementaren Pfad* von u nach v.
3. G ist zusammenhängend und $|E| = |V| - 1$.

Ein Pfad in G heißt **elementarer Pfad**, falls alle in ihm auftretenden Knoten paarweise verschieden sind.

Als Ausnahme ist $u = v$ erlaubt (elementarer **Kreis**).

Beweis

1. *G* ist ein Baum.
2. Für jedes Paar (*u*,*v*) von Knoten von *G* gibt es genau einen elementaren Pfad von *u* nach *v*.

- (1) ⇔ (2).
  - *G* ist *nicht zusammenhängend* gdw. Knoten *u*,*v* existieren, so dass es keinen (elementaren) Pfad von *u* nach *v* gibt.
  - *G* ist *nicht kreisfrei* gdw. Knoten *u,v* existieren, so dass es mehr als einen elementaren Pfad von *u* nach *v* gibt.
  - Also ist *G* zusammenhängend *und* kreisfrei gdw. für jedes Paar (*u*,*v*) *genau ein* elementarer Pfad von *u* nach *v* existiert.

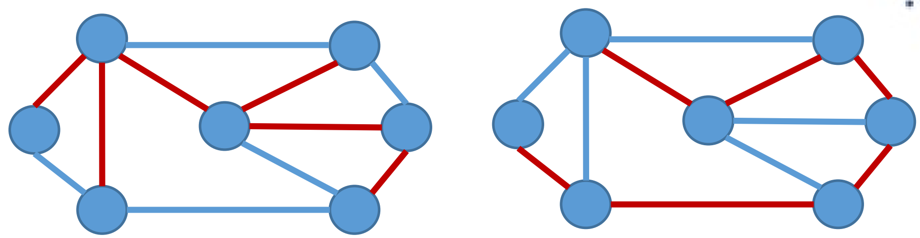
Beweis (*Forts.*)

1. *G* ist ein Baum.

3. *G* ist zusammenhängend und $|E| = |V| - 1$.

- (1) $\Rightarrow$ (3).
  - In jedem gewurzelten Baum zu *G* hat jeder Knoten außer der Wurzel genau einen Vater, also Eingangsgrad 1.
  - Also gehört zu jedem Knoten außer der Wurzel von *G* eine eindeutig bestimmte („hineinführende“) Kante.

Beweis (*Forts.*)



- (3) $\Rightarrow$ (1).
  - Sei $G = (V,E)$ ein beliebiger zusammenhängender Graph.
  - Ein **aufspannender Baum** eines Graphen G ist ein Teilgraph von G, der ein Baum ist und alle Knoten von G enthält
    (existiert für *jeden* zusammenhängenden Graphen).
  - Er ist ein Graph $T_G = (V, E')$ mit $E' \subseteq E$.
    Da T_G ein Baum ist, gilt $|E'| = |V| - 1$.
  - Da laut (3) $|E| = |V| - 1$, gilt $|E'| = |E|$,
    mit $E' \subseteq E$ also $E' = E$ und $T_G = G$. Somit ist G ein Baum.

q.e.d.

Binäre Bäume

- Sei T ein Baum mit Wurzel r.
  T ist ein **binärer Baum**, falls der Ausgangsgrad jedes Knotens höchstens 2 ist.
  D.h. jeder Knoten hat höchstens zwei Kinder.

- **Satz über binäre Bäume.**
  1. Ein binärer Baum der Tiefe d hat höchstens $2^{d+1}-1$ Knoten.
  2. Ein binärer Baum der Tiefe d hat höchstens 2^d Blätter.

  I.A.: $d = 0$ $\rightarrow$ genau $2^0 = 1$ Blatt in Bäumen der Tiefe 0
  I.S.: Vergrößern die Anzahl der Blätter, wenn an ein bisheriges Blatt zwei neue angefügt werden. Das ist nach I.V. höchstens 2^d mal möglich. $\rightarrow$ höchstens $2 \cdot 2^d = 2^{d+1}$ Blätter in Bäumen der Tiefe d+1

Binäre Bäume

- Sei T ein Baum mit Wurzel r.
  T ist ein **binärer Baum**, falls der Ausgangsgrad jedes Knotens höchstens 2 ist.
  D.h. jeder Knoten hat höchstens zwei Kinder.

- **Satz über binäre Bäume.**
  1. Ein binärer Baum der Tiefe d hat höchstens $2^{d+1}-1$ Knoten.
  2. Ein binärer Baum der Tiefe d hat höchstens 2^d Blätter.

  ➢ Ein binärer Baum mit k Blättern hat eine Tiefe von mindestens $\lceil \log_2 k \rceil$.

Folgerung für Entscheidungsbäume

- Entscheidungsbäume für Sortieralgorithmen sind binäre Bäume.
- Für eine Sequenz der Länge *n* entstehen *n*! Blätter.
- Die Tiefe des Entscheidungsbaums ist also mindestens $\lceil \log_2 (n!) \rceil$.
- Für gerade *n* (für ungerade *n* analog):

$$n! = 1 \cdot 2 \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2} + 1\right) \cdot \ldots \cdot (n-1) \cdot n \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$\log_2 (n!) \geq \frac{n}{2} \cdot \log_2 \frac{n}{2} = \frac{n}{2} \cdot (\log_2 n - \log_2 2) \in \boldsymbol{\Theta(n \log n)}$$

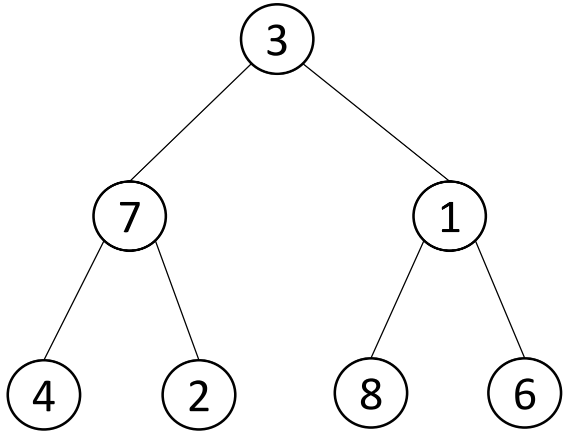
Zeitkomplexität von Sortieralgorithmen

- Die Tiefe des Entscheidungsbaums ist also mindestens $\lceil \log_2 (n!) \rceil$.
- Also sind für mindestens eine Anordnung mindestens $\lceil \log_2 (n!) \rceil$ Vergleiche durchzuführen.
- **Jeder Sortieralgorithmus, der auf Vergleichen von je zwei Elementen basiert, benötigt mindestens $\Theta(n \log n)$ Vergleiche.**
- **Die Zeitkomplexität jedes Sortieralgorithmus, der auf Vergleichen von je zwei Elementen basiert, ist im worst case $\Omega(n \log n)$.**
- Mergesort ist bzgl. der worst case Komplexität optimal.

Bäume als Datenstrukturen

Bäume als strukturierte Datentypen

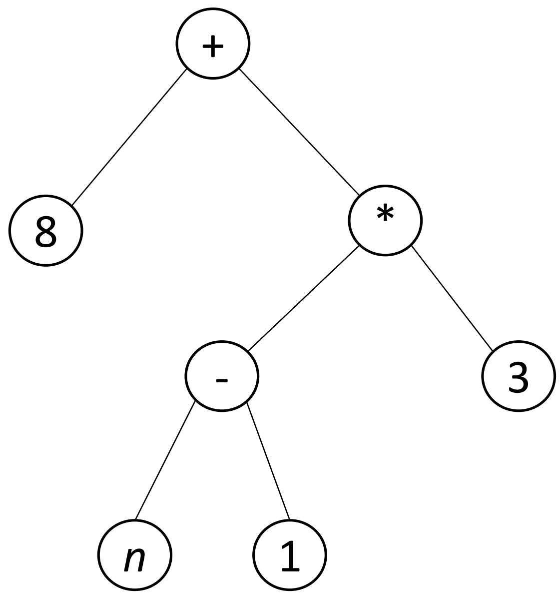
- Verwenden binäre Bäume zum Speichern von Werten, *z.B.* für die Sequenz [3, 7, 1, 4, 2, 8, 6] so:



Zugehörige Folge ist abhängig von der Art, in der der Baum durchlaufen wird.

Arithmetische Ausdrücke als Bäume

- 8 + (n – 1) * 3



- Struktur wird offensichtlich:
  - Vorrangregeln
  - Assoziativität (5-4-1 = ?)
- Klammern werden überflüssig
- Compiler benutzen diese Darstellung

ADT Baum (Auszug)

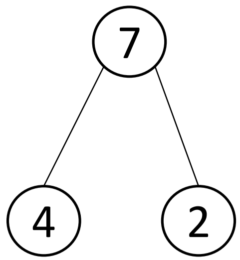
```
type  Tree =                                  /* mit informaler Interpretation */
   sorts boolean, int, tree
   functions
         empty:  → tree                       /* leerer Baum */
         isEmpty: tree → boolean              /* Test, ob Baum leer */
         depth: tree → int                    /* Tiefe des Baums */
         ...                                  /* kann erweitert werden */

end.
```

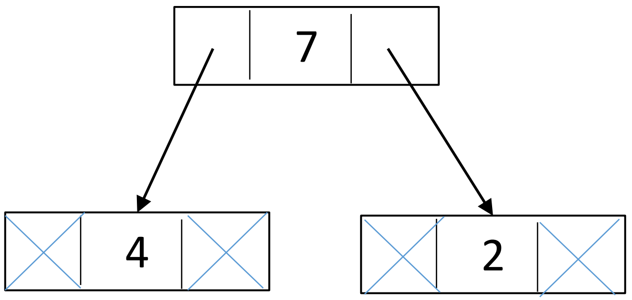
Implementierung (1)

- **Baum** durch Zeiger auf Wurzel**knoten**
- `null` für leeren Baum

- **Knoten** eines binären Baums als Tripel
  - Zeiger auf linkes Kind (ggf. `null`)
  - Wert
  - Zeiger auf rechtes Kind (ggf. `null`)



repräsentiert durch



ADT Binärer Baumknoten

<u>type</u> BinNode =

    <u>sorts</u> T, tree, node

    <u>functions</u>

        new: $T \to node$

        value: $node \to T$

        left: $node \to tree$

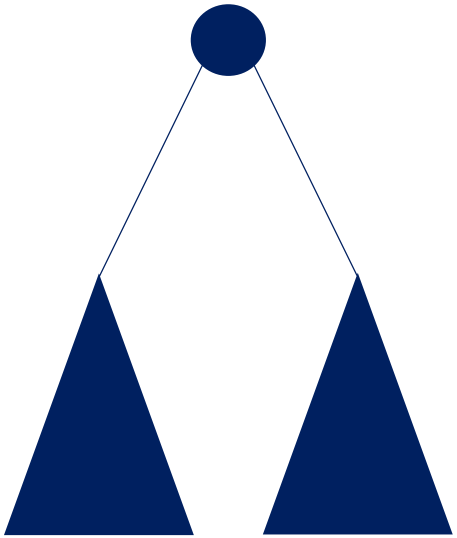
        right: $node \to tree$

        setValue: $node \times T \to node$

        setLeft: $node \times tree \to node$

        setRight: $node \times tree \to node$

<u>end</u>.



Interpretation

- **I(T) =** ADT des Grundtyps
- **I(tree)** = {**`null`**} ∪ Adr /* Zeigertyp */
- **I(node)** = I(tree) × I(T) × I(tree) mit:

- new(*x*) = (**`null`**, *x*, **`null`**) /* isolierter Knoten */
- value((*lt*, *v*, *rt*)) = *v*
- left((*lt*, *v*, *rt*)) = *lt* /* linker Teilbaum */
- right((*lt*, *v*, *rt*)) = *rt* /* rechter Teilbaum */
- setValue((*lt*, *v*, *rt*), ***x***) = (*lt*, ***x***, *rt*)
- setLeft((*lt*, *v*, *rt*), ***nt***) = (***nt***, *v*, *rt*)
- setRight((*lt*, *v*, *rt*), ***nt***) = (*lt*, *v*, ***nt***)

Implementierung (2)

- Effizient möglich sind z.B.
  - empty(), isEmpty(*tree*), value(*node*), left(*node*), right(*node*) in ***O*(1)**
  - Tiefe bestimmen in ***O*(*n*)** (wobei *n* die Anzahl der Knoten ist):

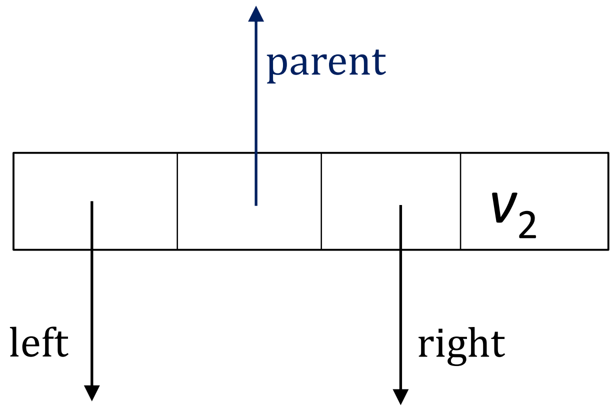
```
# pre: tree t ist nicht leer
def depth(t):
    if (isEmpty(left(content(t))): t_left = -1
    else: t_left = depth(left(content(t)))
    if (isEmpty(right(content(t))): t_right = -1
    else: t_right = depth(right(content(t)))
    return 1 + max(t_left,t_right)
```

- Schwierig:

  Vater bestimmen, Backtracking Tiefensuche, Bestimmung der Wurzel, ...

Andere Implementierungen (1)

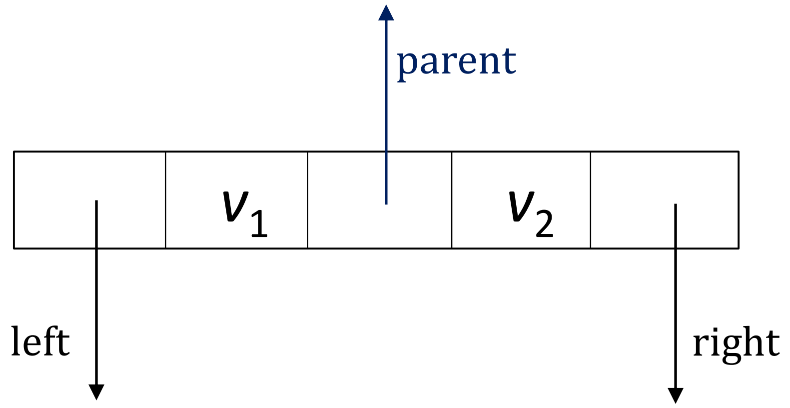
- Strategie der doppelt verketteten Liste:
  Zeiger auf den Vater als zusätzliche Komponente



- zusätzliche Funktionen im ADT Knoten:
  - getParent: node $\rightarrow$ tree
  - setParent: : node $\times$ tree $\rightarrow$ node

Andere Implementierungen (2)

- Verallgemeinerung auf mehrere Werte und/oder Kinder, *z.B.* zwei Werte bei Implementierung im Stil der doppelt verketteten Liste:

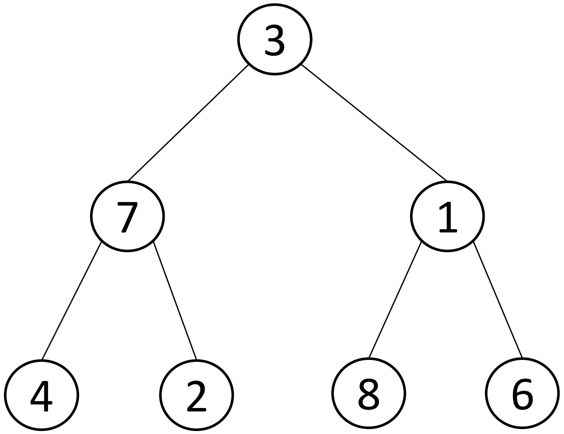


- Spezieller Knotentyp für Blätter (Verringern der Platzkomplexität durch Vermeidung von übermäßig vielen Null-Zeigern)

Traversieren von Bäumen

- **Inorder:** Zuerst linken Teilbaum, dann den Wert der Wurzel, dann den rechten Teilbaum.
- **Preorder:** Zuerst den Wert der Wurzel, dann den linken Teilbaum, dann den rechten Teilbaum. (DFS)
- **Postorder:** Zuerst linken Teilbaum, dann den rechten Teilbaum, dann den Wert der Wurzel.
- **Levelorder:** BFS

Traversieren am Beispiel



- **Inorder:** 4 – 7 – 2 – 3 – 8 – 1 – 6
- **Preorder:** 3 – 7 – 4 – 2 – 1 – 8 – 6 (*DFS*)
- **Postorder:** 4 – 2 – 7 – 8 – 6 – 1 – 3
- **Levelorder:** 3 – 7 – 1 – 4 – 2 – 8 – 6 (*BFS*)

Algorithmen

- Annahme: Das Interface ist implementiert.
- Beispiel **Inorder:**

```
def inorder(t):
     if isEmpty(t): return
     inorder(left(content(t)))
     print(value(content(t))) # oder andere Aktionen
     inorder(right(content(t)))
```