Kapitel ADS:IV

IV. Datenstrukturen

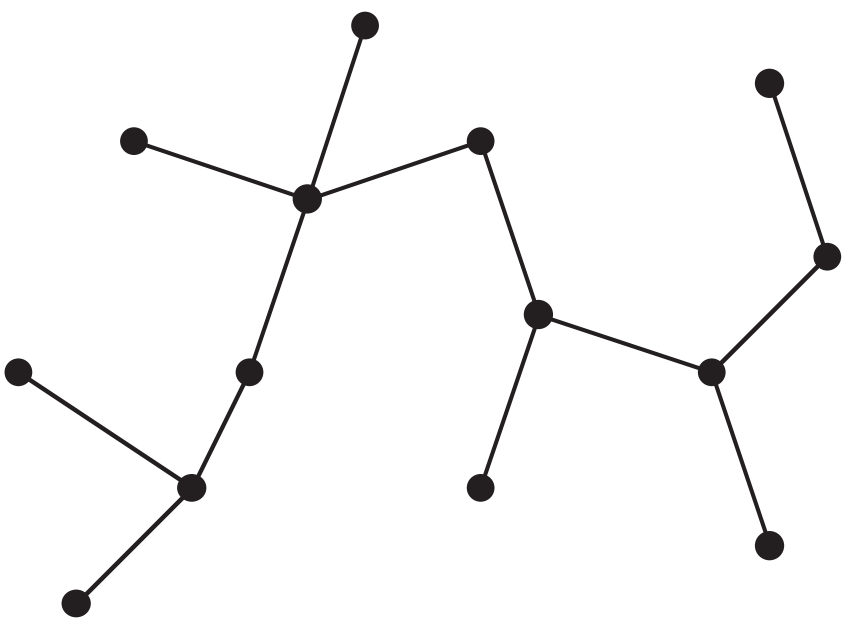
- Record
- Linear List
- Linked List
- Stack
- Queue
- Priority Queue
- Dictionary
- Direct-address Table
- Hash Table
- Hash Function
- Tree

Tree

Definition

Ein Graph heißt Tree (*Baum*), wenn er zusammenhängend und azyklisch ist.

Beispiel:



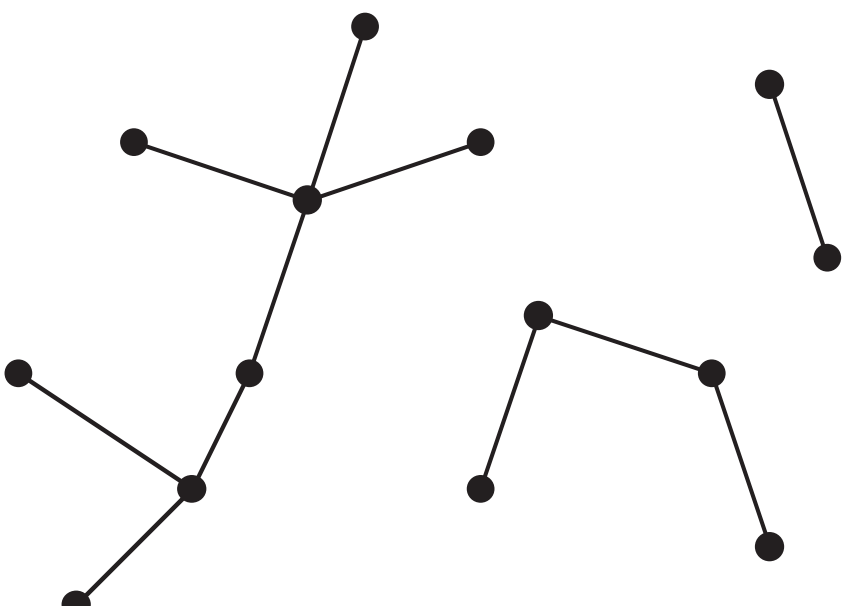
Tree

Definition

Ein Graph heißt Tree (*Baum*), wenn er zusammenhängend und azyklisch ist.

Ein Graph heißt Forest (*Wald*), wenn jede seiner Komponenten ein Baum ist.

Beispiel:



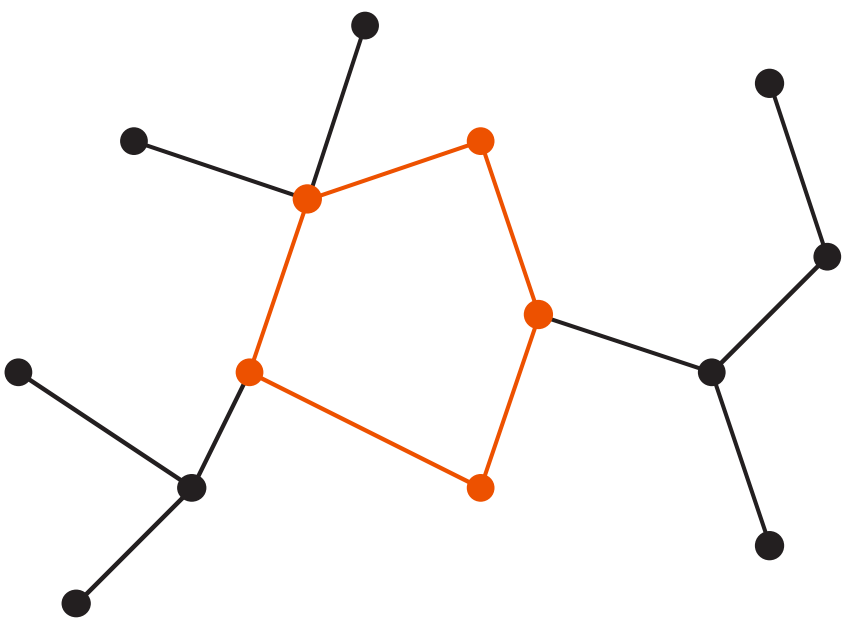
Tree

Definition

Ein Graph heißt Tree (*Baum*), wenn er zusammenhängend und azyklisch ist.

Ein Graph heißt Forest (*Wald*), wenn jede seiner Komponenten ein Baum ist.

Beispiel:



Bemerkungen:

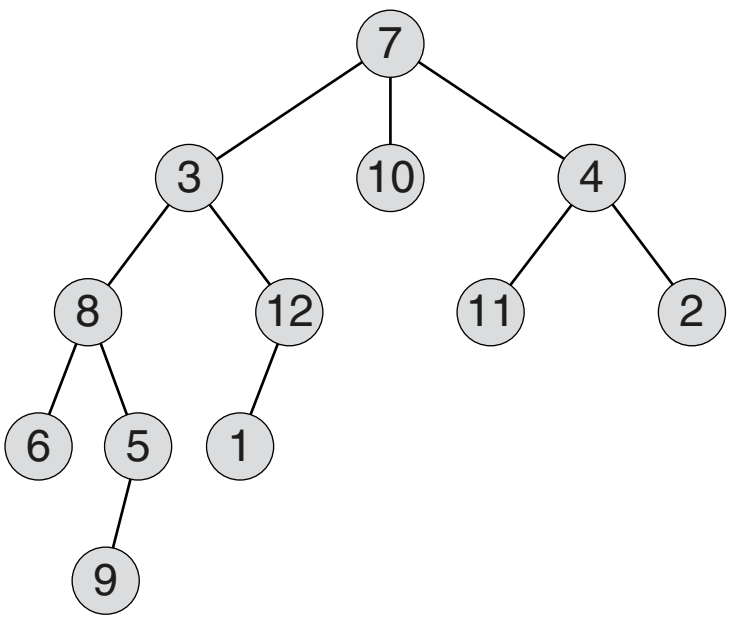
- Ein ungerichteter Graph heißt zusammenhängend, falls es zu je zwei beliebigen Knoten einen Weg von einem zum anderen gibt.
- Einen maximalen zusammenhängenden Teilgraphen eines ungerichteten Graphen bezeichnet man als Komponente oder Zusammenhangskomponente.
- Beachten Sie den Unterschied zum Begriff des (starken) Zusammenhangs und der (starken) Zusammenhangskomponente bei gerichteten Graphen:

Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:

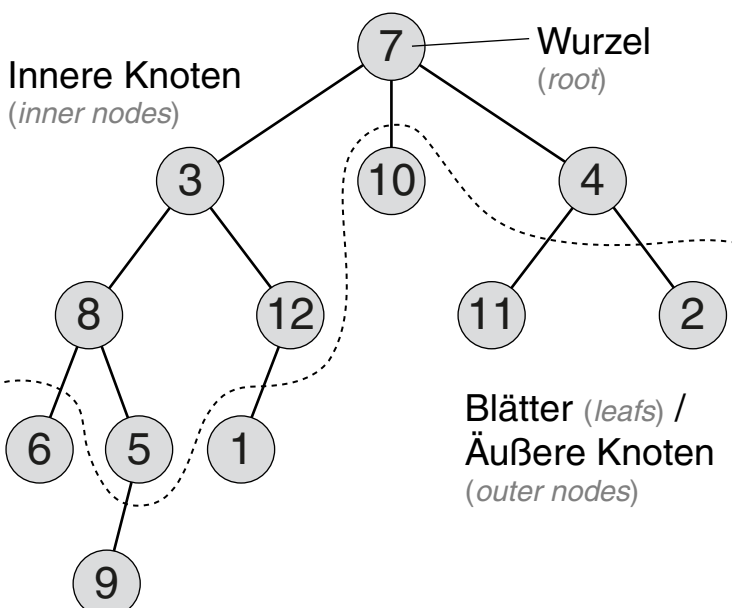


Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:



Knotenarten

Rekursivität

Verwandtschaft

Grad und Tiefe

Höhe

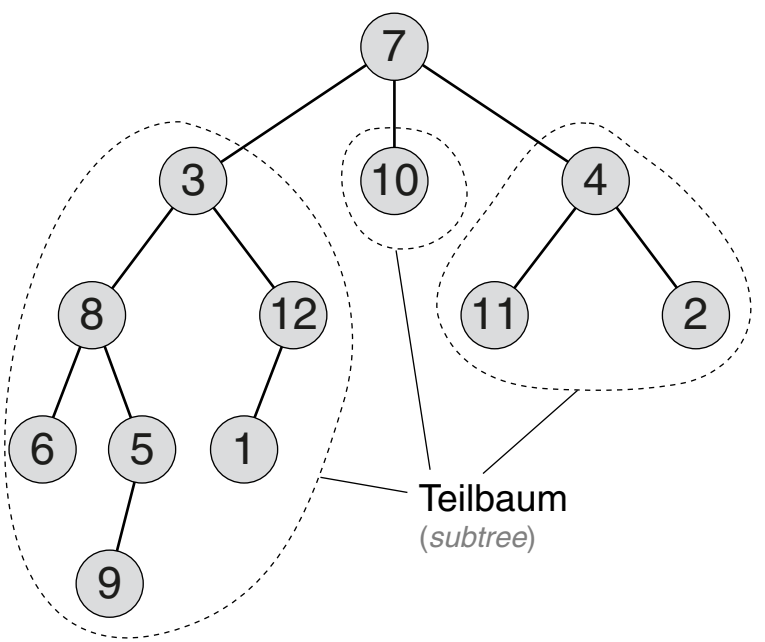
Äquivalenz

Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:

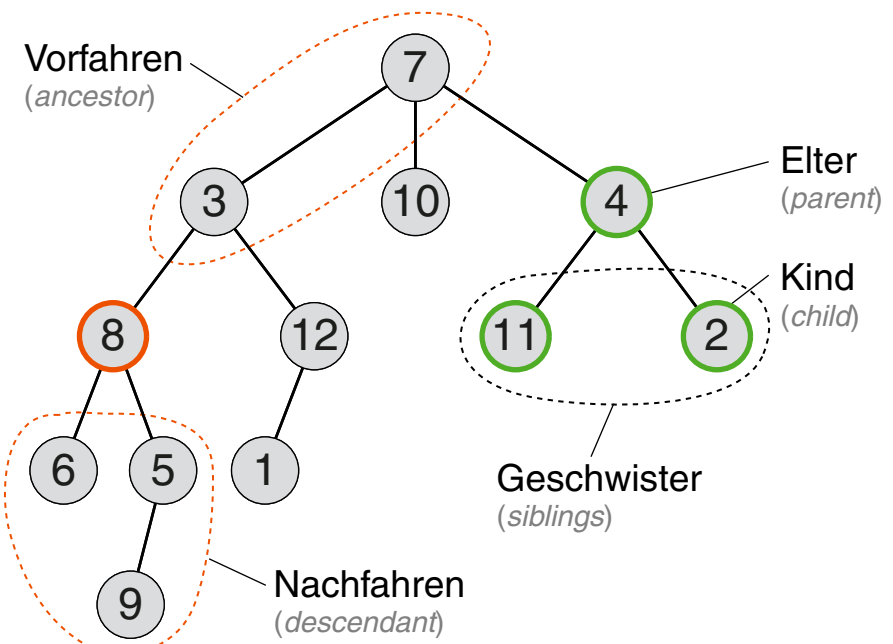


Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:

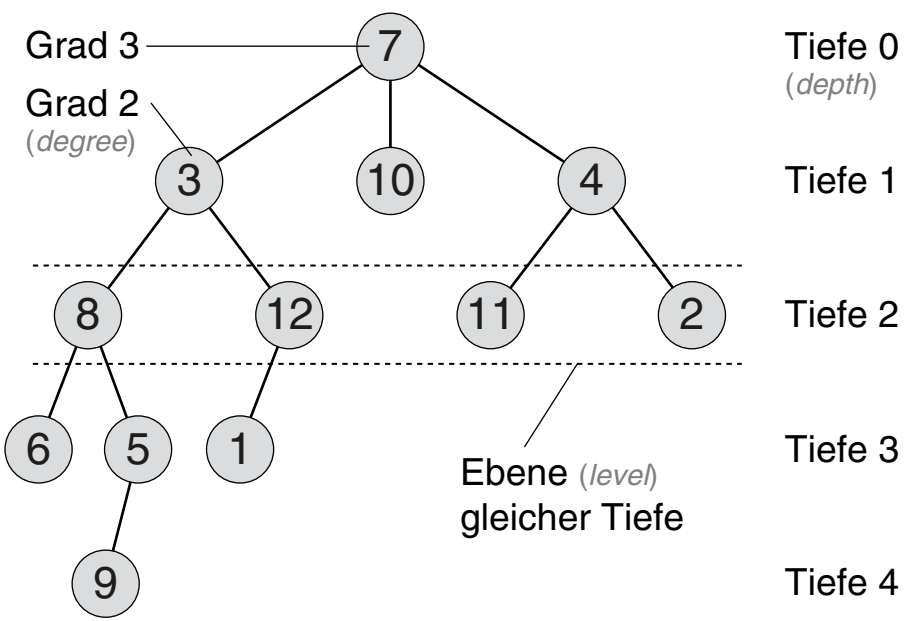


Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:

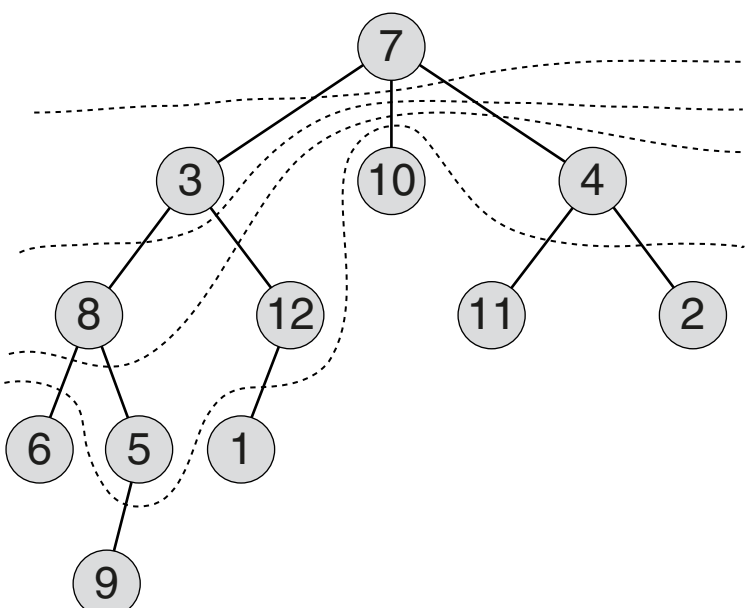


Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:



Höhe 4
Höhe 3
Höhe 2

Höhe 1

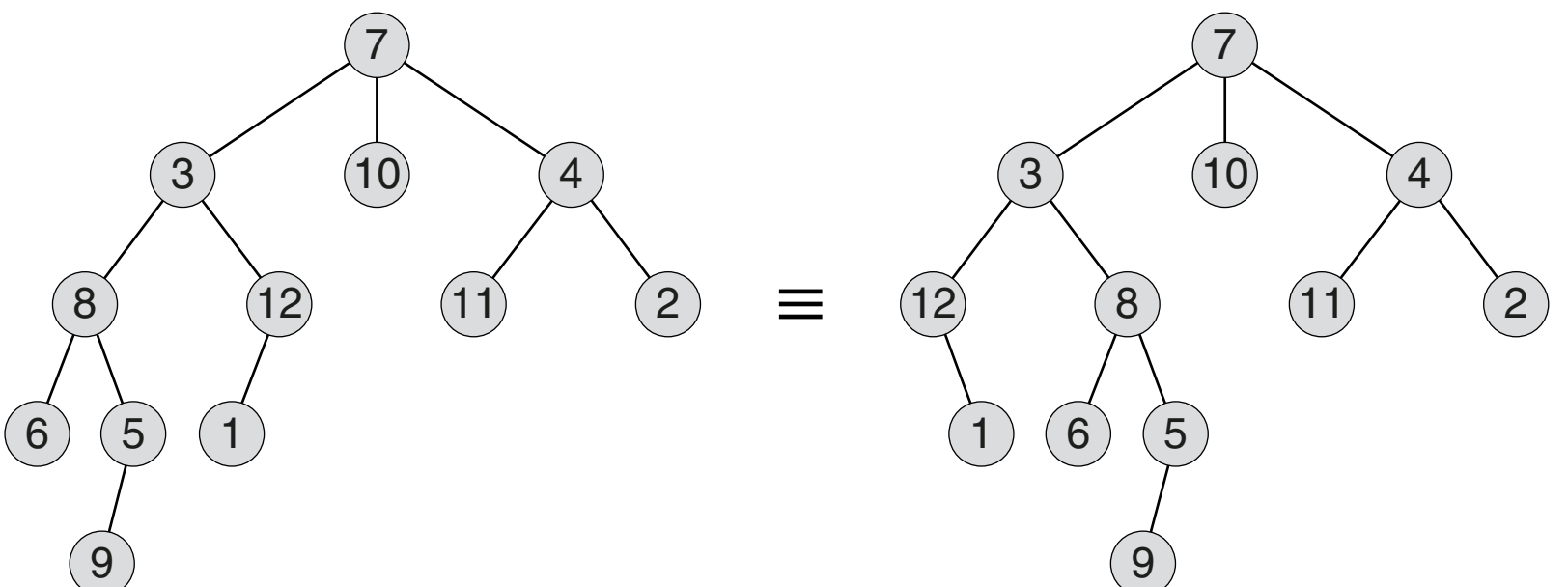
Höhe 0
(*height*)

Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Beispiel:



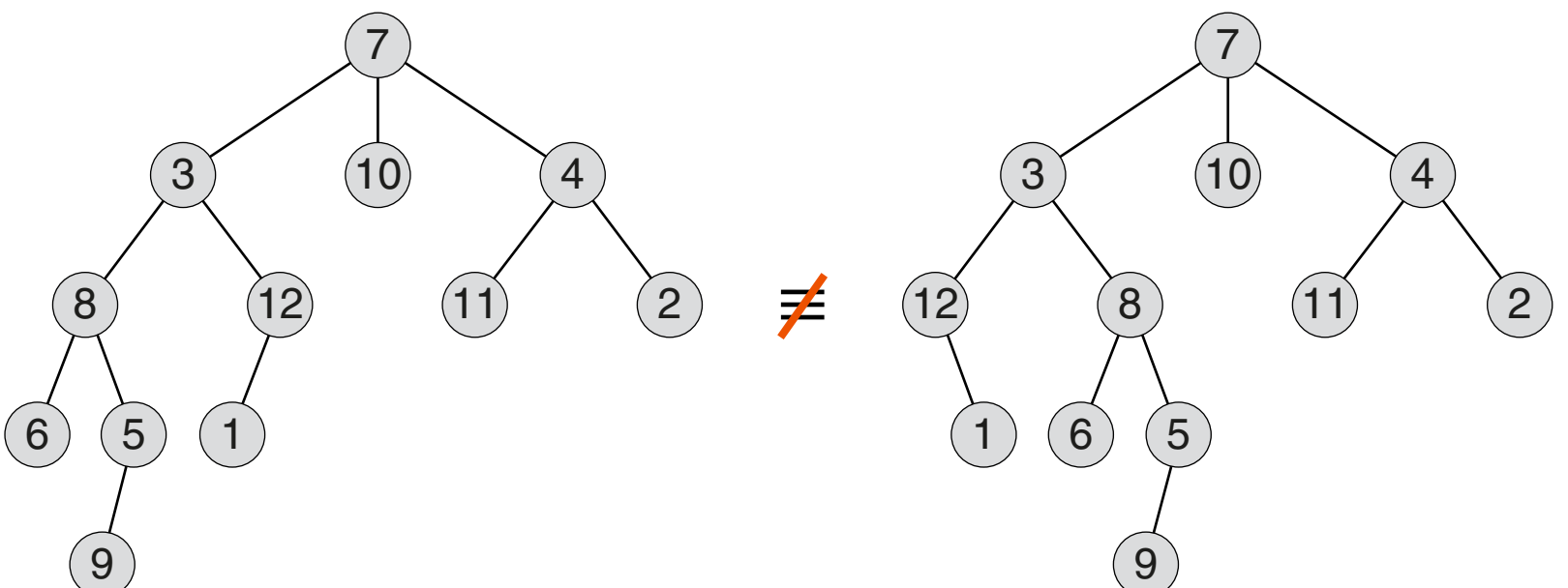
Tree

Definition

Ein Tree heißt Rooted Tree (*gewurzelter Baum*), wenn einer seiner Knoten als Wurzel bezeichnet wird.

Ein Rooted Tree heißt Ordered Tree (*geordneter Baum*), wenn die Kinder jedes seiner Knoten einer linearen Ordnung unterliegen.

Beispiel:

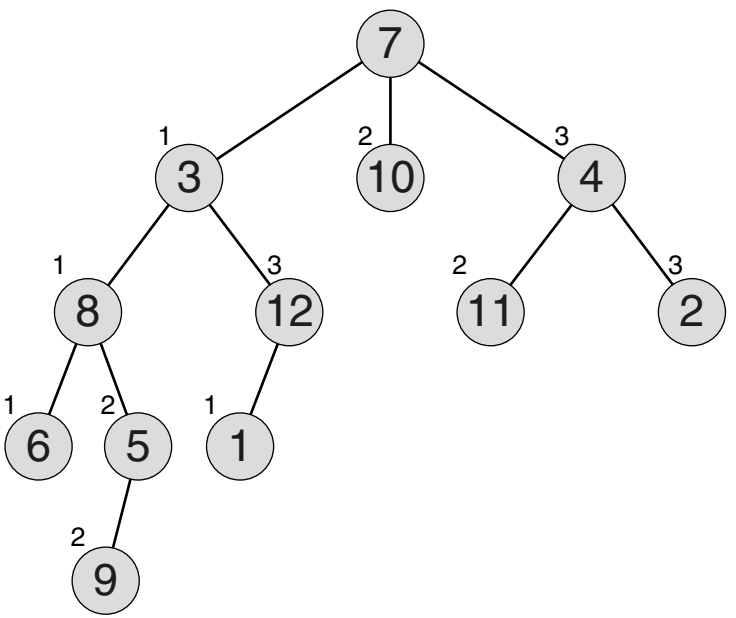


Tree

Definition

Ein Ordered Tree heißt Positional Tree (*Positionsbaum*), wenn jedes Elter feste Positionen für Kinder hat. Ein k-ary Tree (k*-närer Baum*) hat k Positionen pro Elter.

Beispiel:



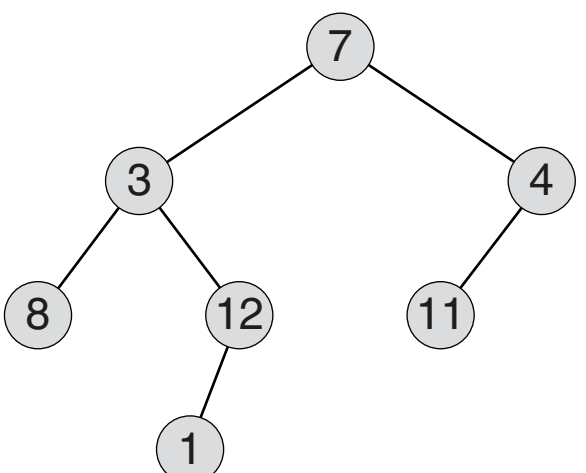
Tree

Definition

Ein Ordered Tree heißt Positional Tree (*Positionsbaum*), wenn jedes Elter feste Positionen für Kinder hat. Ein k-ary Tree (k*-närer Baum*) hat k Positionen pro Elter.

Ein 2-ary Tree heißt Binary Tree (*Binärbaum*). Rekursiv: Ein Binärbaum ist entweder leer oder seine Wurzel hat je einen Binärbaum als linken und rechten Teilbaum.

Beispiel:



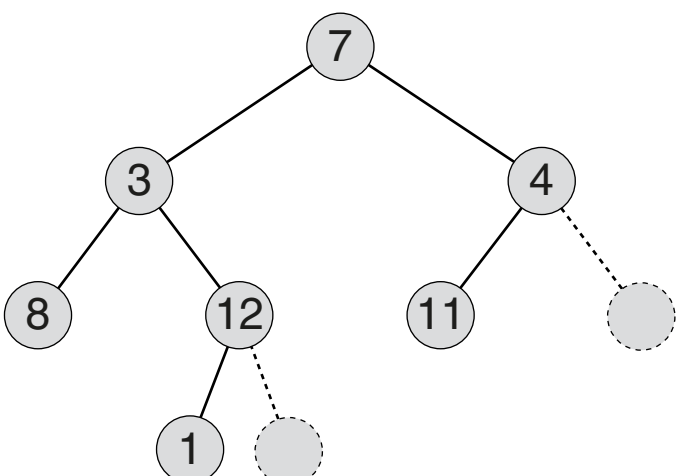
Tree

Definition

Ein Ordered Tree heißt Positional Tree (*Positionsbaum*), wenn jedes Elter feste Positionen für Kinder hat. Ein k-ary Tree (k*-närer Baum*) hat k Positionen pro Elter.

Ein 2-ary Tree heißt Binary Tree (*Binärbaum*). Rekursiv: Ein Binärbaum ist entweder leer oder seine Wurzel hat je einen Binärbaum als linken und rechten Teilbaum.

Beispiel:



Strikter *k*-närer Baum:
Jeder Knoten hat entweder keinen oder *k* Kinder.

Vollständiger *k*-närer Baum:
Strikter *k*-närer Baum, dessen Ebenen mit Ausnahme der letzten komplett gefüllt sind. Die letzte Ebene ist von links an komplett.

Perfekter *k*-närer Baum:
Strikter *k*-närer Baum, dessen Blätter alle auf derselben Ebene sind.

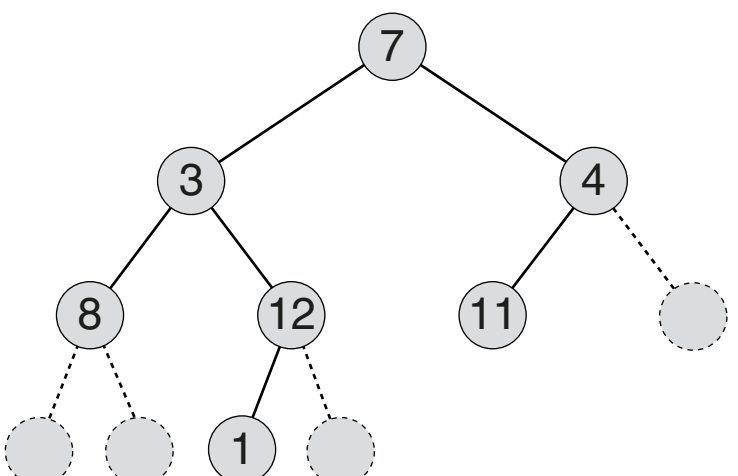
Tree

Definition

Ein Ordered Tree heißt Positional Tree (*Positionsbaum*), wenn jedes Elter feste Positionen für Kinder hat. Ein k-ary Tree (k*-närer Baum*) hat k Positionen pro Elter.

Ein 2-ary Tree heißt Binary Tree (*Binärbaum*). Rekursiv: Ein Binärbaum ist entweder leer oder seine Wurzel hat je einen Binärbaum als linken und rechten Teilbaum.

Beispiel:



Strikter *k*-närer Baum:
Jeder Knoten hat entweder keinen oder *k* Kinder.

Vollständiger *k*-närer Baum:
Strikter *k*-närer Baum, dessen Ebenen mit Ausnahme der letzten komplett gefüllt sind. Die letzte Ebene ist von links an komplett.

Perfekter *k*-närer Baum:
Strikter *k*-närer Baum, dessen Blätter alle auf derselben Ebene sind.

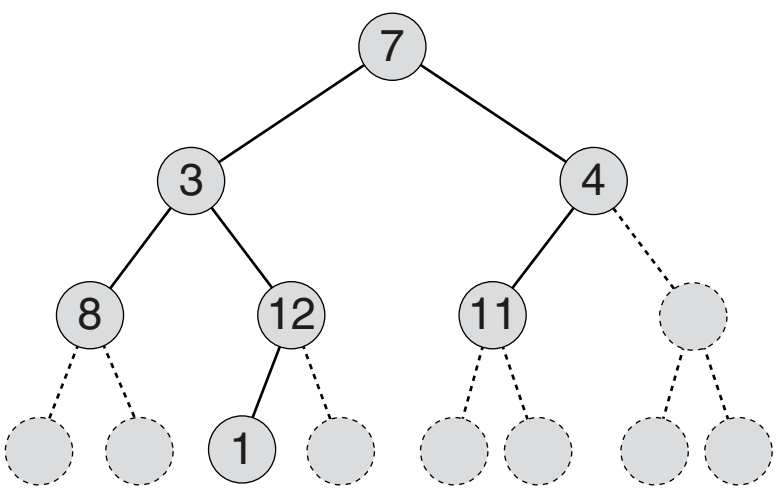
Tree

Definition

Ein Ordered Tree heißt Positional Tree (*Positionsbaum*), wenn jedes Elter feste Positionen für Kinder hat. Ein k-ary Tree (k*-närer Baum*) hat k Positionen pro Elter.

Ein 2-ary Tree heißt Binary Tree (*Binärbaum*). Rekursiv: Ein Binärbaum ist entweder leer oder seine Wurzel hat je einen Binärbaum als linken und rechten Teilbaum.

Beispiel:



Strikter *k*-närer Baum:
Jeder Knoten hat entweder keinen oder *k* Kinder.

Vollständiger *k*-närer Baum:
Strikter *k*-närer Baum, dessen Ebenen mit Ausnahme der letzten komplett gefüllt sind. Die letzte Ebene ist von links an komplett.

Perfekter *k*-närer Baum:
Strikter *k*-närer Baum, dessen Blätter alle auf derselben Ebene sind.

Bemerkungen:

- Alle obigen Definitionen setzen voraus, dass der zugrundeliegende Graph ungerichtet ist. Bei gerichteten Graphen wird bei Rooted Trees zusätzlich gefordert, dass entweder alle Kanten von der Wurzel weg zeigen (sogenannte Out-Trees; der Normalfall) oder dass alle Kanten zur Wurzel hin zeigen (In-Trees). Bei Out-Trees muss jeder Blattknoten eines Rooted Trees von der Wurzel aus auf genau einem Pfad erreichbar sein; bei In-Trees muss die Wurzel von allen Blattknoten und inneren Knoten außer der Wurzel auf genau einem Pfad erreichbar sein.
- Der Grad (*degree*) eines Knotens ist die Zahl seiner Kinder.
- Die Tiefe (*depth*) eines Knotens ist die Zahl der Kanten auf dem Pfad von der Wurzel zum Knoten. Die Tiefe eines Baums entspricht der Zahl der Kanten des längsten Pfads von der Wurzel zu einem Blatt.
- Die Ebene (*level*) eines Knotens ist $1+$ die Tiefe des Knotens.
- Die Höhe (*height*) eines Knotens ist die Zahl der Kanten im längsten Pfad vom Knoten zur einem Blatt. Die Höhe eines Baums entspricht der Höhe seines Wurzelknotens.

Bemerkungen: (Fortsetzung)

- Der leere Baum ist ein Graph $G = (V, E)$ mit leerer Knotenmenge $V = \{\}$ leerer Kantemenge $E = \{\}$. Als Konzept vereinfacht er die rekursive Definition von Binärbäumen.
- Die Begriffe strikt (*strict*), komplett (*complete*) und perfekt (*perfect*) sind in unterschiedlichen Quellen unterschiedlich definiert; beispielsweise werden oft komplett und perfekt vertauscht. Anders als [Knuth 1997] definiert [Cormen 2009] „complete“ wie perfekt und beschreibt komplett stattdessen als „nearly complete“.
- Ein perfekter k-närer Baum der Tiefe h hat k^h Blattknoten. Die Höhe h eines perfekten k-nären Baums mit n Blättern ist damit $h = \log_k n$.
- Ein perfekter k-närer Baum der Tiefe h hat

$$\begin{aligned} 1 + k + k^2 + \ldots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

  interne Knoten. Ein Binärbaum hat damit $2^h - 1$ interne Knoten.

Tree

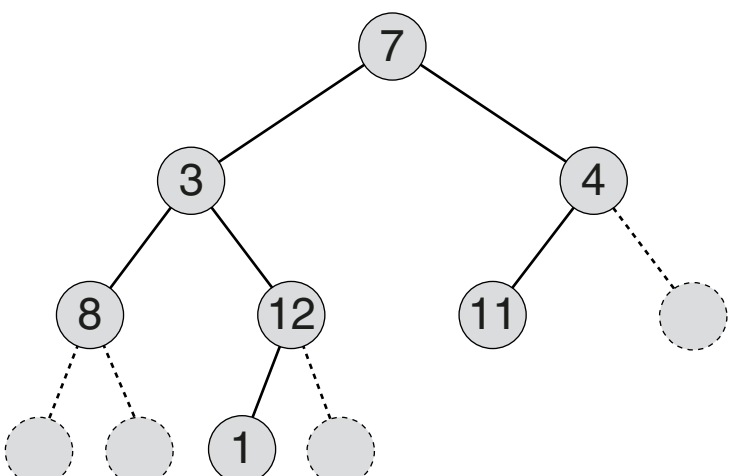
Implementierung

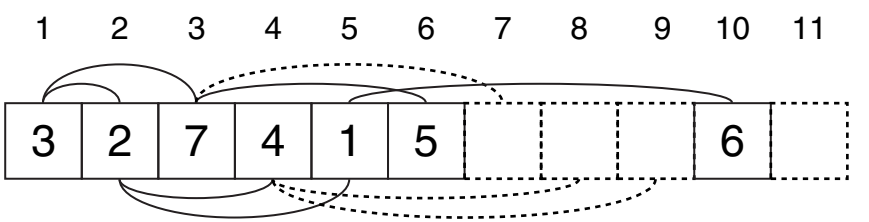
- Array-basiert

  Einzelnes Array oder parallele Arrays für Satellitendaten. Effektive Speichernutzung nur bei (nahezu) kompletten k-nären Bäumen.

- Link-basiert

Beispiel:

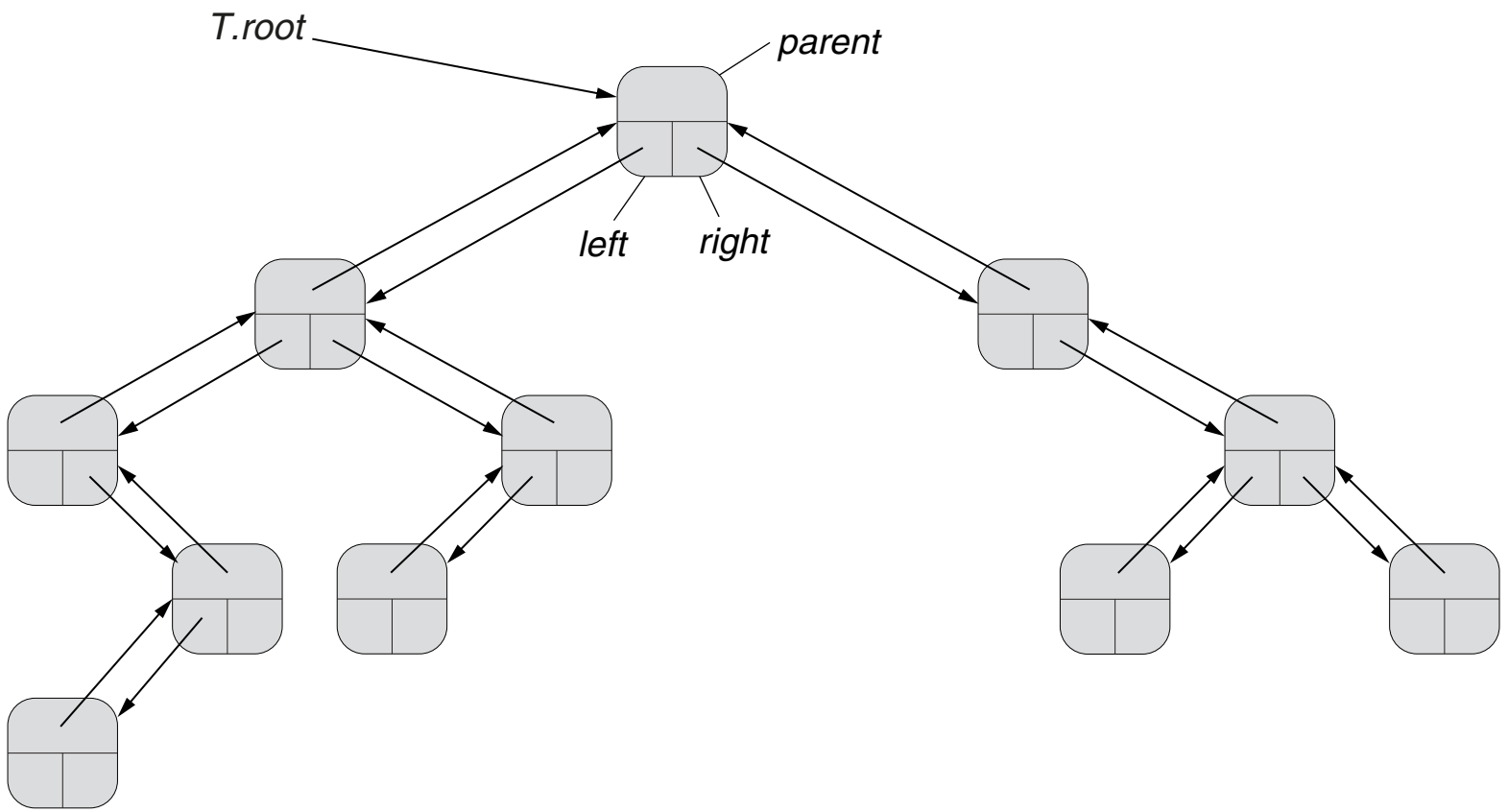




Tree

Implementierung: Beispiel

Link-basierter Binärbaum:

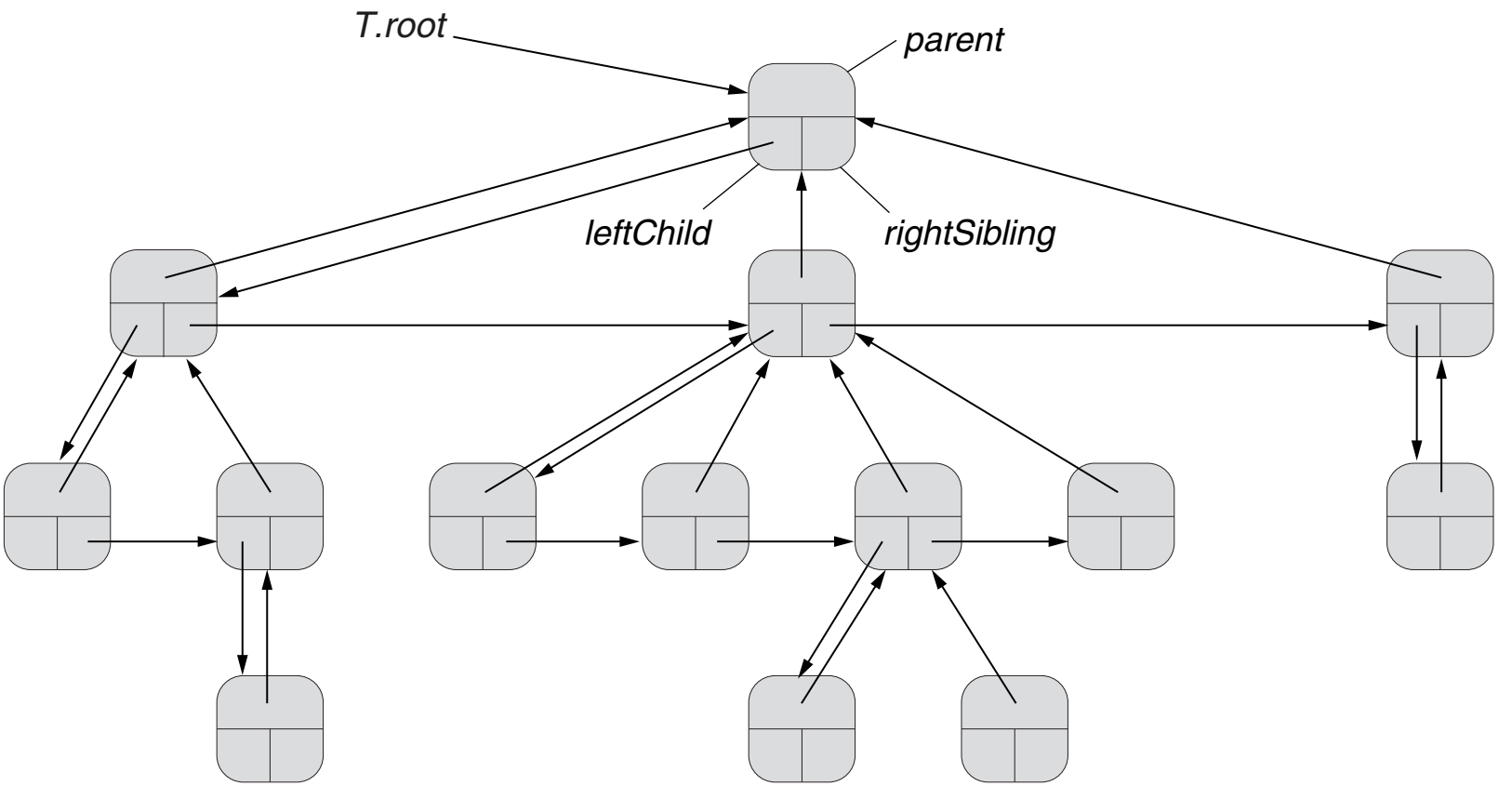


Jeder Knoten hat zusätzlich das Schlüsselattribut *key* (hier ausgeblendet).

Tree

Implementierung: Beispiel

Link-basierter Rooted Tree mit unbegrenztem Knotengrad:



Jeder Knoten hat zusätzlich das Schlüsselattribut *key* (hier ausgeblendet).

Tree

Manipulation

- Baum traversieren (*Traverse*)
  Alle Knoten im Baum in einer bestimmten Reihenfolge durchschreiten.
- Knoten suchen (*Search*)
  Den Baum traversieren, bis ein Knoten mit vorgegebenem Schlüssel gefunden oder der gesamte Baum traversiert wurde.
- Knoten einfügen (*Insert*)
  Einen Knoten an einer vorgegebenen Stelle im Baum einfügen.
- Knoten löschen (*Delete*)
  Einen bestimmten Knoten aus dem Baum löschen.
- Knoten verändern
  Den Schlüssel eines bestimmten Knotens ändern.

Bemerkungen:

- Die Algorithmen für das Einfügen, Löschen und Ändern von Knoten hängen von der Implementierung des Baums ab: Bei Array-basierten Bäumen sind das Einfügen und Löschen gegebenenfalls mit Mehraufwand verbunden, andere Knoten des Baumes an anderen Stellen im Array zu speichern. Link-basierte Bäume ermöglichen diese Operationen analog zu verlinkten Listen mit geringem Mehraufwand.
- Weiterhin hängen die Algorithmen für das Einfügen, Löschen und Ändern von Knoten davon ab, für welchen Zweck ein Baum verwendet wird und ob globale Eigenschaften aufrecht erhalten werden müssen.

Tree

Manipulation: Traversierung

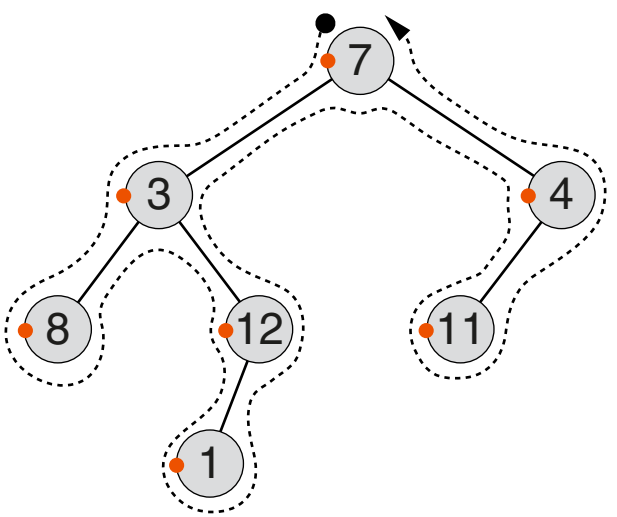
- Depth-first search (DFS, *Tiefensuche*)
  Exploration aller Kinder vor Geschwistern. Drei Varianten werden unterschieden.
- Breadth-first search (BFS, *Breitensuche*)
  Ebenenweise Exploration aller Geschwister vor Kindern.

Tree

Manipulation: Traversierung

- Depth-first search (DFS, *Tiefensuche*)
  Exploration aller Kinder vor Geschwistern. Drei Varianten werden unterschieden.
- Breadth-first search (BFS, *Breitensuche*)
  Ebenenweise Exploration aller Geschwister vor Kindern.

Beispiel:



Visits: 7, 3, 8, 12, 1, 4, 11

Algorithmus: DFS Traverse (pre-order).

Eingabe: x. Wurzel eines Binärbaums.

DFSTraverse(x)

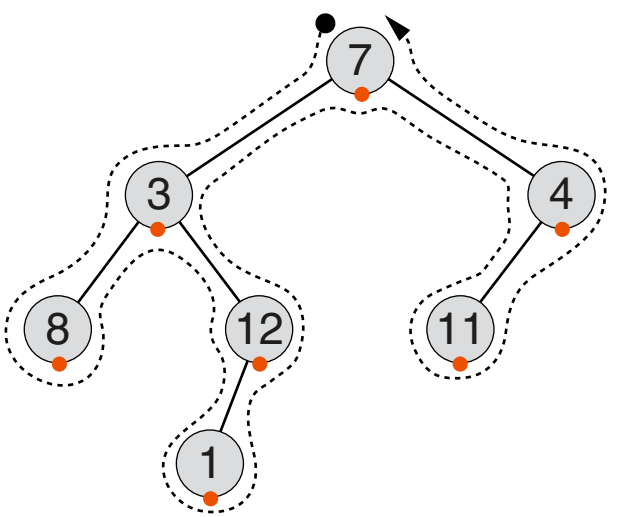
```
1. IF x == NIL THEN return() ENDIF
2. visit(x)
3. DFSTraverse(x.left)
4. visit(x)
5. DFSTraverse(x.right)
6. visit(x)
```

Tree

Manipulation: Traversierung

- Depth-first search (DFS, *Tiefensuche*)
  Exploration aller Kinder vor Geschwistern. Drei Varianten werden unterschieden.
- Breadth-first search (BFS, *Breitensuche*)
  Ebenenweise Exploration aller Geschwister vor Kindern.

Beispiel:



Visits: 8, 3, 1, 12, 7, 11, 4

Algorithmus: DFS Traverse (in-order).

Eingabe: x. Wurzel eines Binärbaums.

DFSTraverse(x)

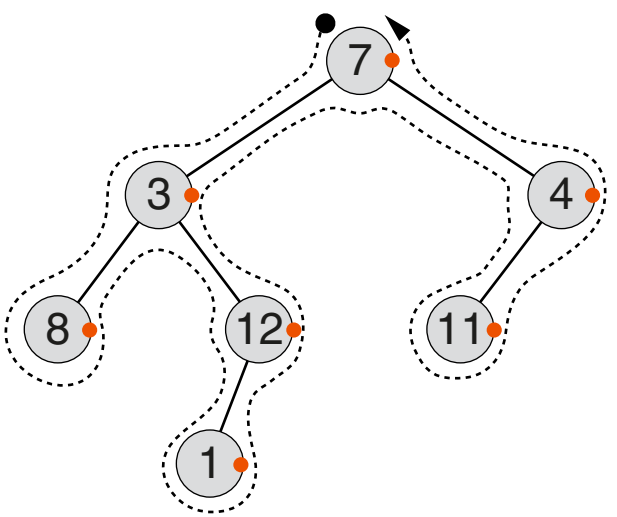
```
1. IF x == NIL THEN return() ENDIF
2. visit(x)
3. DFSTraverse(x.left)
4. visit(x)
5. DFSTraverse(x.right)
6. visit(x)
```

Tree

Manipulation: Traversierung

- Depth-first search (DFS, *Tiefensuche*)
  Exploration aller Kinder vor Geschwistern. Drei Varianten werden unterschieden.
- Breadth-first search (BFS, *Breitensuche*)
  Ebenenweise Exploration aller Geschwister vor Kindern.

Beispiel:



Visits: 8, 1, 12, 3, 11, 4, 7

Algorithmus: DFS Traverse (post-order).

Eingabe: x. Wurzel eines Binärbaums.

DFSTraverse(x)

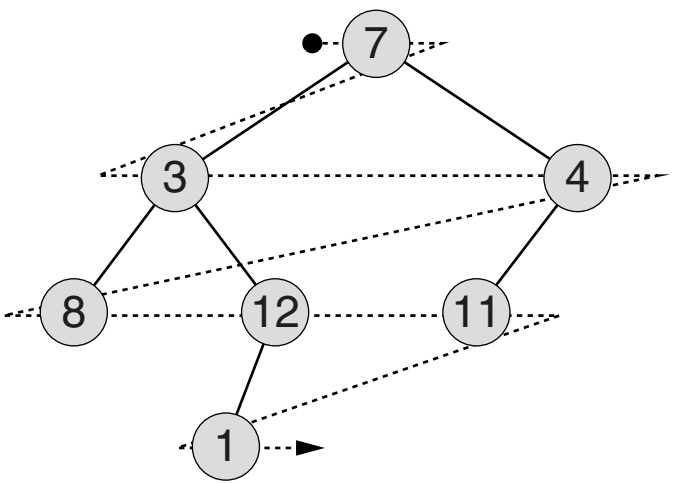
```
1. IF x == NIL THEN return() ENDIF
2. visit(x)
3. DFSTraverse(x.left)
4. visit(x)
5. DFSTraverse(x.right)
6. visit(x)
```

Tree

Manipulation: Traversierung

- Depth-first search (DFS, *Tiefensuche*)
  Exploration aller Kinder vor Geschwistern. Drei Varianten werden unterschieden.
- Breadth-first search (BFS, *Breitensuche*)
  Ebenenweise Exploration aller Geschwister vor Kindern.

Beispiel:



Visits: 7, 3, 4, 8, 12, 11, 1

Algorithmus: BFS Traverse.

Eingabe: T. Binärbaum.

BFSTraverse(T)

```
1.  Q = queue()
2.  Enqueue(Q, T.root)
3.  WHILE |Q| ≠ 0 DO
4.    x = Dequeue(Q)
5.    visit(x)
6.    IF x.left THEN Enqueue(x.left) ENDIF
7.    IF x.right THEN Enqueue(x.right) ENDIF
8.  ENDDO
```

Bemerkungen:

- Alle gezeigten Varianten von Traverse lassen sich auf Rooted Trees verallgemeinern.
- Der Begriff „in-order“ erschließt sich, wenn der Binärbaum ein Binary Search Tree ist: Dann nämlich gibt eine In-Order-Traversierung die in der Datenstruktur gespeicherten Elemente in aufsteigender Reihenfolge ihrer Sortierschlüssel aus. Die Begriffe „pre-order“ und „post-order“ sind gemäß ihrem Verhältnis zur In-Order-Traversierung davon abgeleitet.
- Die Hilfsfunktion *visit*(x) steht für Operationen, die unter Verwendung eines Knotens x ausgeführt werden können, zum Beispiel den Sortierschlüssel Ausdrucken oder Verändern.
- Manche Algorithmen erfordern eine DFS-Traversierung, bei der Knoten sowohl pre-order, in-order als auch post-order besucht werden.