10. Kapitel
(Teil1)

BÄUME
GRUNDLAGEN

Algorithmen & Datenstrukturen

Prof. Dr. Wolfgang Schramm

Übersicht

1. Einführung
2. Algorithmen
3. Eigenschaften von Programmiersprachen
4. Algorithmenparadigmen
5. Suchen & Sortieren
6. Hashing
7. Komplexität von Algorithmen
8. Abstrakte Datentypen (ADT)
9. Listen
10. **Bäume**
11. Graphen

Lernziele des Kapitels

- Verstehen, was ein Baum (in der Informatik) ist?
- Kennenlernen von verschiedenen Baumarten.
- ADT Baum (Tree) kennenlernen.
- Den ADT Tree mit seinen verschiedenen Operationen in Java implementieren können.
- Spezielle Ausprägung von Bäumen kennenlernen.

Inhalt

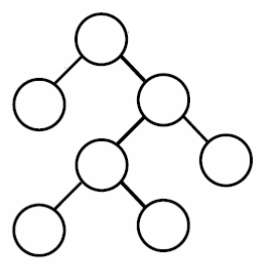
1. Einführung
2. Bäume – Begriffe, Definition
3. Binäre Bäume, binäre Suchbäume
4. Balancierte Bäume
   - AVL-Bäume
   - B-Bäume
5. Weitere Bäume
6. Sortieren mit Bäumen: Heapsort

Anwendungen von Bäumen

- Familienstammbuch
- Ergebnisse eines Sportturniers („KO-System“)
- Organigramm
- Gliederung eines Buches
- Datei-Struktur im Rechner
- Maximum-Bestimmung rekursiv
- Arithmetischer Ausdruck
- …

Baum – Begriffe 1/7

- Dynamische Datenstruktur
- Anzahl der Elemente beliebig: 0 .. ∞
- Funktionen
  - Erzeugen → leerer Baum
  - Einfügen → Baum mit einem Element mehr
  - Rausnehmen → Baum mit einem Element weniger
  - Nachschauen, ob der Baum leer ist
    → Baum unverändert
  - Linken (rechten) Teilbaum bilden
    → (neuer) Baum
- Visualisierung



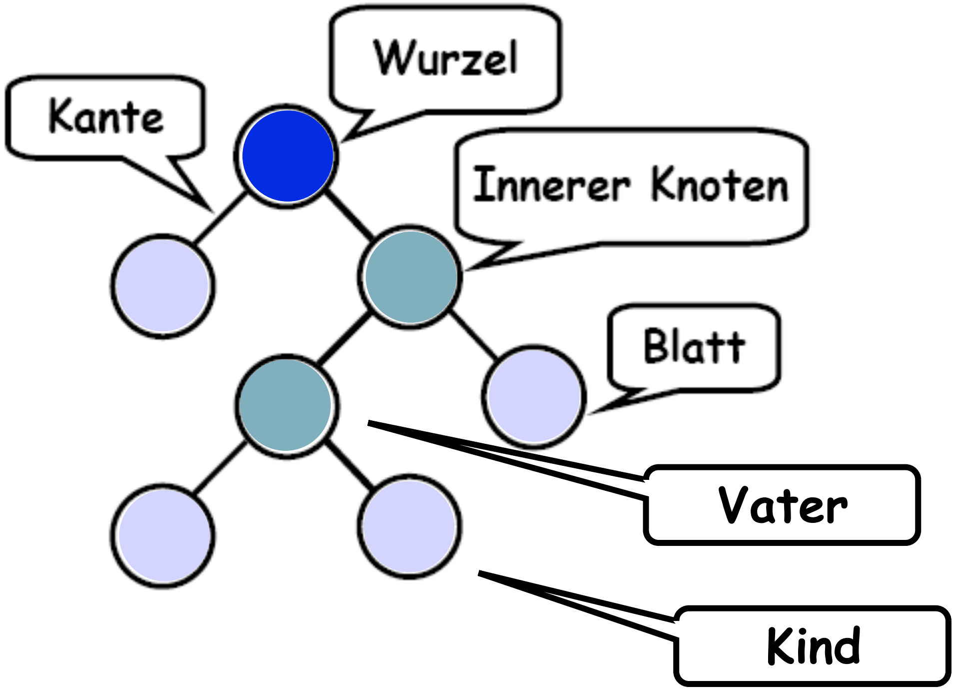
Baum – Begriffe 2/7

- Hierarchisches Strukturierungs- und Organisationsprinzip.
- Verallgemeinerte Liste
  - Mehr als ein Nachfolger erlaubt
- Spezieller Graph
  - Zusammenhängender, zyklenfreier Graph

Baum – Begriffe 3/7

- **Baum = Menge von Knoten und Kanten**, die besondere Eigenschaften aufweisen.
- Jeder Baum besitzt einen ausgezeichneten Knoten, die ***Wurzel*** (*root*); Ausnahme: leerer Baum.
- Jeder Knoten, außer der Wurzel ist durch genau eine Kante mit seinem ***Vater*knoten** (*parent*, Synonyme: Mutter, Elternknoten, Vorgänger) verbunden. Er wird ***Kind*** (*child* , Synonyme: Tochter, Sohn, Nachfolger) dieses Knotens genannt.
- Ein Knoten ohne Kinder heißt ***Blatt*** (*leaf*), alle anderen Knoten nennt man innere Knoten.

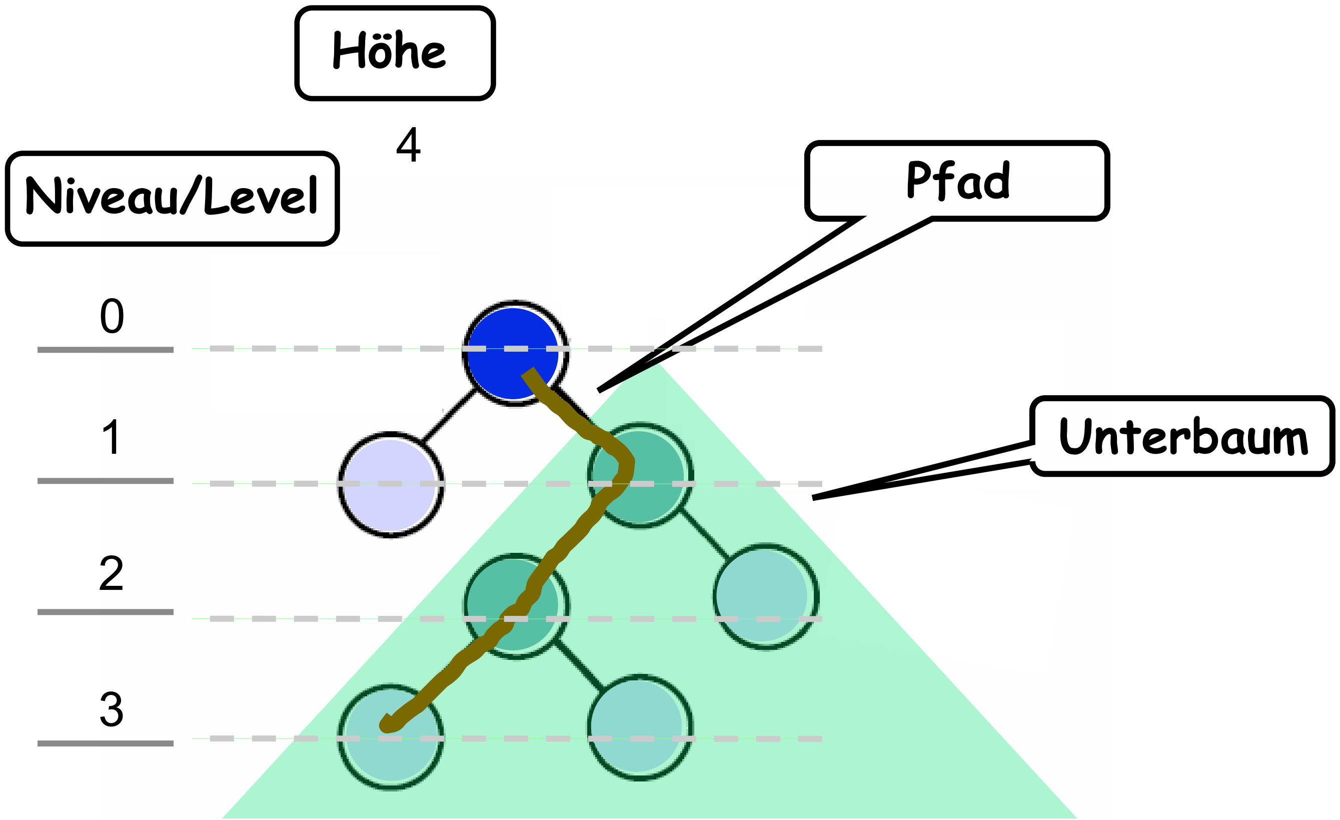
Baum – Begriffe 4/7



Baum – Begriffe 5/7

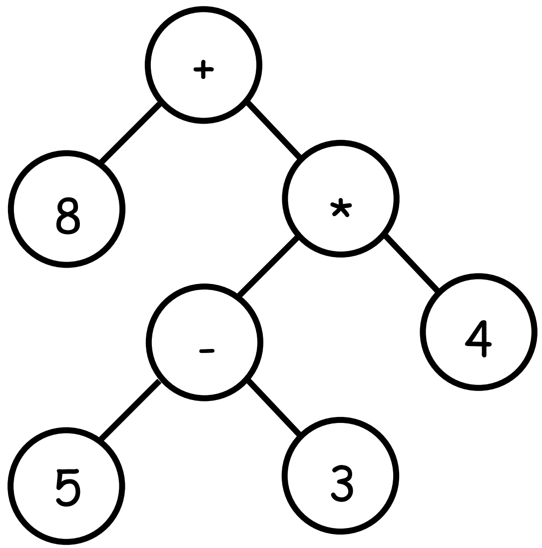
- Ein ***Pfad*** (*path*) in einem Baum ist eine Folge von unterschiedlichen Knoten, in der die aufeinander folgenden Knoten durch Kanten miteinander verbunden sind.
- Zwischen jedem Knoten und der Wurzel gibt es genau einen Pfad. Das bedeutet dass
  - → ein Baum zusammenhängend ist und
  - → es keine Zyklen gibt.
- Das ***Niveau*** (*level*) eines Knotens ist die Länge des Pfades von der Wurzel zu diesem Knoten.
- Die ***Höhe*** (*height*) eines Baumes entspricht dem größten level eines Blattes + 1.
- Es gibt verschiedene Arten von Bäumen. Sie können dadurch charakterisiert sein, dass jeder Knoten eine bestimmte Anzahl von direkten Kindern haben muss und wie die Kinder angeordnet sind.

Baum – Begriffe 6/7



Baum – Begriffe 7/7

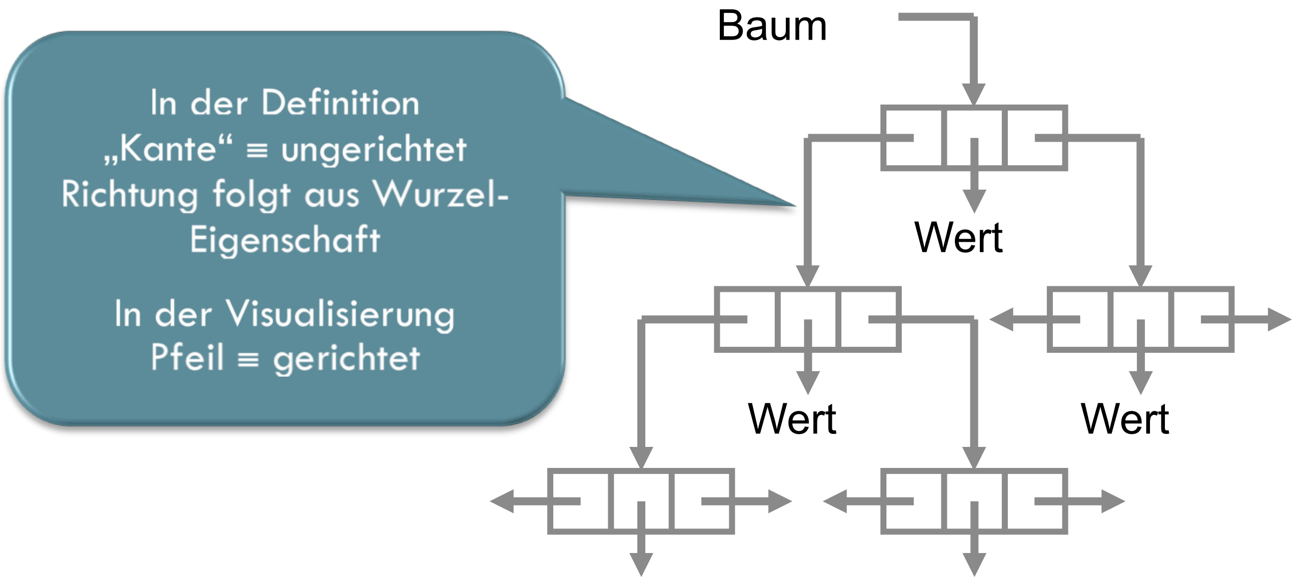
- Bei Vorgabe der Anzahl von Kindern: ***n-ärer Baum*** (*n-ary tree*).
- Sind die Kinder jedes Knotens in einer bestimmten Reihenfolge geordnet: ***geordneter Baum*** (*ordered tree*).
- Binärer Baum = geordneter Baum, bei welchem jeder Knoten maximal 2 Kinder hat.
- Beispiel: arithmetischer Ausdruck als Baum 8 + (5 – 3) * 4



Visualisierung 1/2

- Leerer Baum

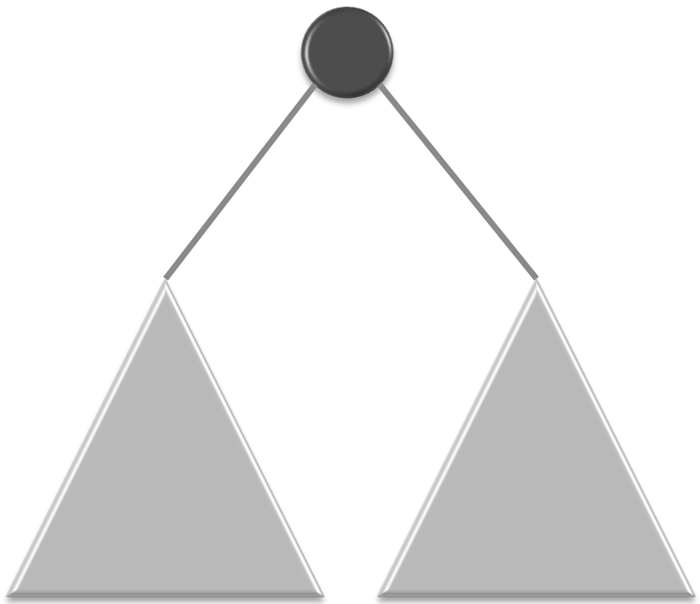
- Nicht-leerer Baum



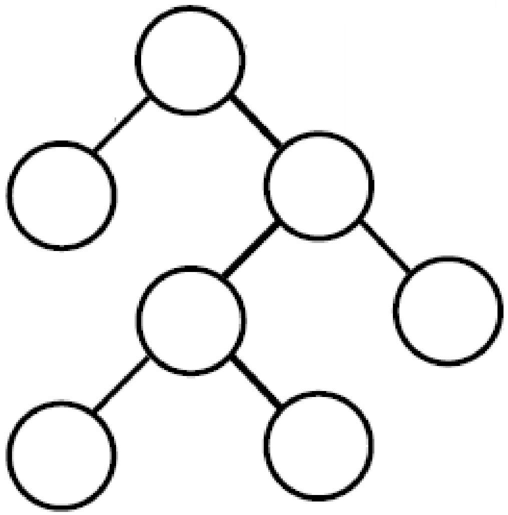
Visualisierung 2/2

- … oder einfacher

…so…



…oder so…



Tree – Operationen

- Binärer Baum (Tree) als ADT:
  - **Operationen / Functions (Auswahl):**
    - insert - fügt ein neues Element in den Baum ein

      insert: Element × Tree → Tree
    - remove – entfernt ein Element aus dem Baum

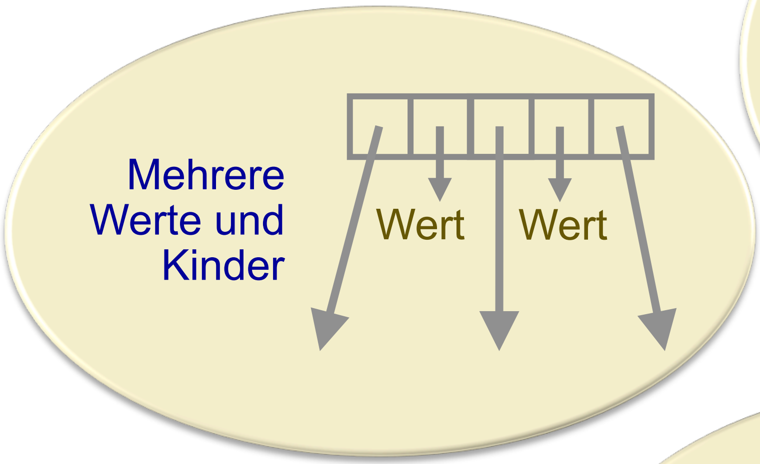
      remove: Element × Tree → Tree
    - empty – erzeugt einen leeren neuen Baum

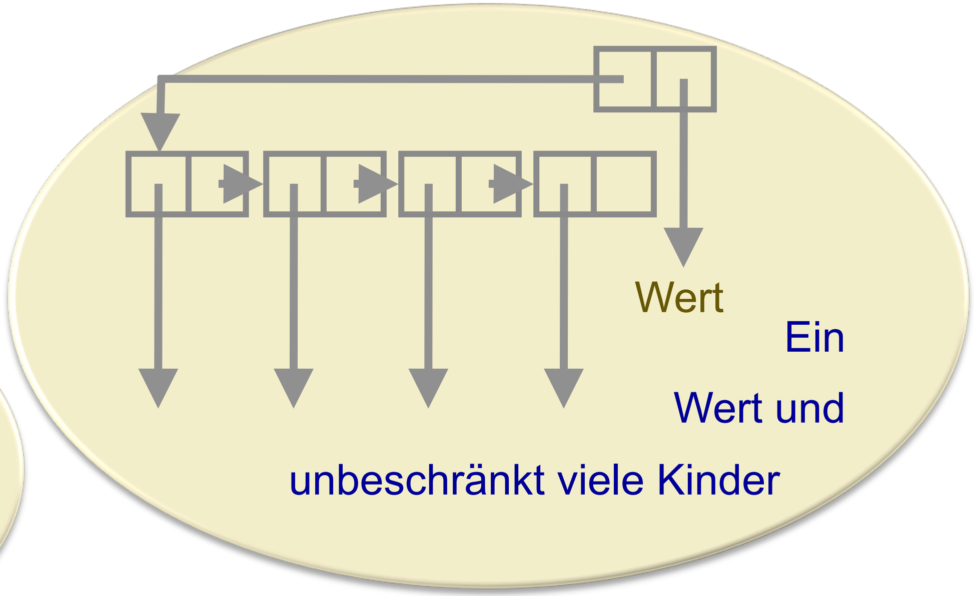
      empty: → Tree
    - isEmpty - liefert true genau dann, wenn der Baum leer ist

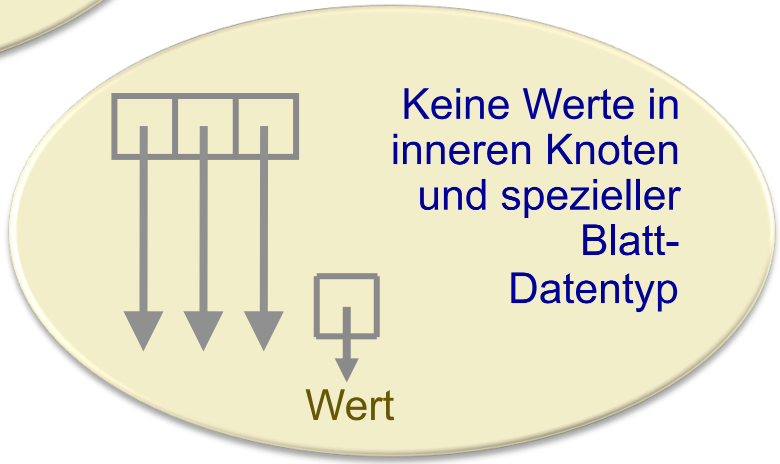
      isEmpty: Tree → boolean

Typische verwendete Datentypen für die Implementierung

- Baumknoten…
  etwas allgemeiner







Tree – Implementierung

Knotenklasse

```
class TreeNode {// Node of a binary tree
  int elem;
  TreeNode left;
  TreeNode right;

  public TreeNode (int i) {
    elem = i;
    left = right = null;
  }

  public TreeNode getLeft () {
    return left;
  }

  public TreeNode getRight () {
    return right;
  }
```

```
public int getElement () {
  return elem;
}

public void setLeft (TreeNode n) {
  left = n;
}

public void setRight (TreeNode n) {
  right = n;
}

public void setElement (int e) {
  elem = e;
}
```

Tree – Implementierung (allg.)

- Knotenklasse

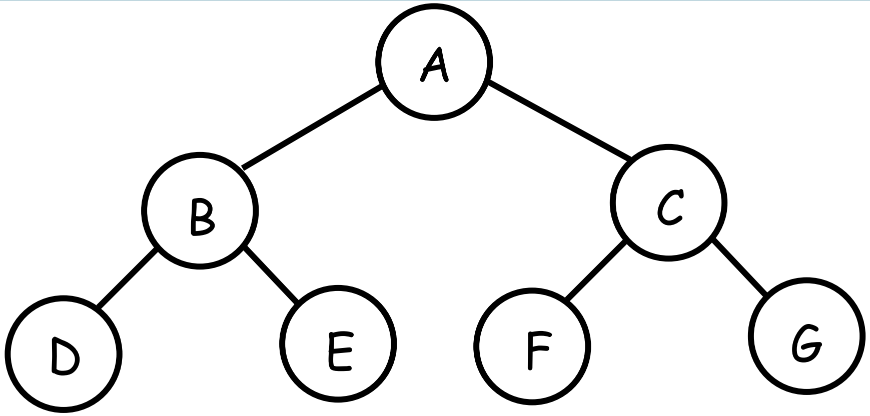
```
class TreeNode {// Node of a binary tree
  Element elem;
  TreeNode left;
  TreeNode right;

  public TreeNode (Element i) {
    elem = i;
    left = right = null;
  }

  public TreeNode getLeft () {
    return left;
  }

  public TreeNode getRight () {
    return right;
  }

public Element getElement () {
  return elem;
}

public void setLeft (TreeNode n) {
  left = n;
}

public void setRight (TreeNode n) {
  right = n;
}

public void setElement (Element e) {
  elem = e;
}
```

Algorithmen zur Traversierung 1/2

Typische Aktion: Ausdrucken

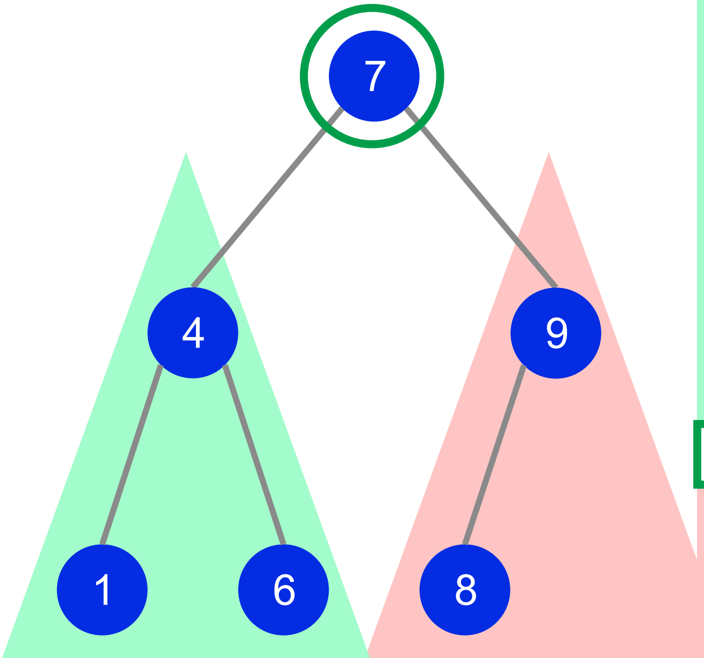
- **Inorder** (Zwischenordnung) Durchlauf
  - Zuerst wird der linke Teilbaum besucht, dann der Knoten selbst und anschließend der rechte Teilbaum.
- **Preorder** (Vorordnung) Durchlauf
  - Zuerst wird der der Knoten selbst besucht, dann linke Teilbaum und anschließend der rechte Teilbaum.
- **Postorder** (Nachordnung) Durchlauf
  - Zuerst wird der linke Teilbaum besucht, dann der rechte Teilbaum und anschließend der Knoten selbst.
- **Levelorder** Durchlauf (auch: breadth-first search)
  - Zuerst werden alle Knoten auf demselben Niveau besucht, dann wird auf das nächste Niveau gewechselt.

Algorithmen zur Traversierung 2/2



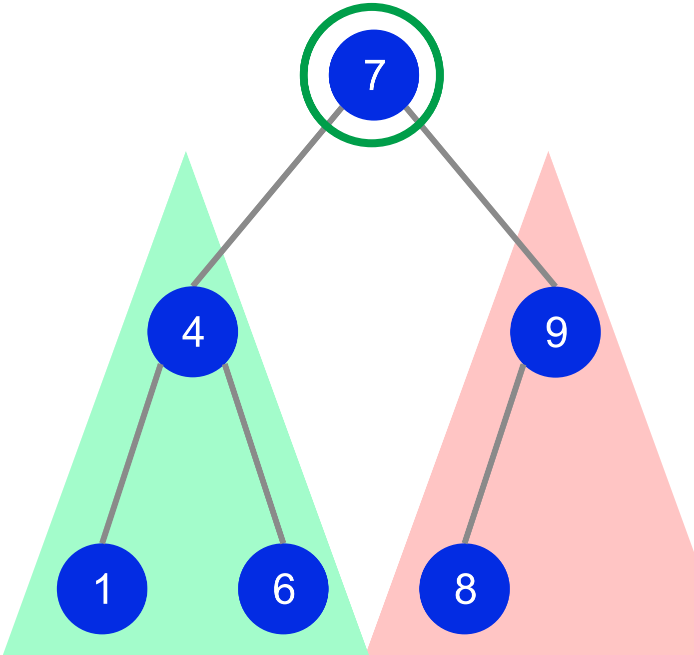
- Inorder Durchlauf
  - D → B → E → A → F → C → G
- Preorder Durchlauf
  - A → B → D → E → C → F → G
- Postorder Durchlauf
  - D → E → B → F → G → C → A
- Levelorder Durchlauf
  - A → B → C → D → E → F → G

Traversieren von Bäumen: Inorder



Start: wurzel 7	
linker Unterbaum (Wurzel 4)	
linker Unterbaum (Wurzel 1)	
linker Unterbaum (null)	
Wurzel 1	1
rechter Unterbaum (null)	
Wurzel 4	4
rechter Unterbaum (Wurzel 6)	
linker Unterbaum (null)	
Wurzel 6	6
rechter Unterbaum (null)	
Wurzel 7	7
rechter Unterbaum (Wurzel 9)	
linker Unterbaum (Wurzel 8)	
linker Unterbaum (null)	
Wurzel 8	8
rechter Unterbaum (null)	
…	

Traversieren von Bäumen: Preorder



Start: wurzel 7	
Wurzel 7	**7**
linker Unterbaum (Wurzel 4)	
Wurzel 4	**4**
linker Unterbaum (Wurzel 1)	
Wurzel 1	**1**
linker Unterbaum (null)	
rechter Unterbaum (null)	
rechter Unterbaum (Wurzel 6)	
Wurzel 6	**6**
linker Unterbaum (null)	
rechter Unterbaum (null)	
rechter Unterbaum (Wurzel 9)	
Wurzel 9	**9**
linker Unterbaum (Wurzel 8)	
Wurzel 8	**8**
linker Unterbaum (null)	
…	

Algorithmus - Inorder

Inorder (k)

Eingabe: Knoten k eines binären Baums mit Verweis auf linken (k.left) und rechten (k.right) Teilbaum sowie dem Element k.elem.

```
Inorder (k.left);    // besuche den linken Teilbaum
Verarbeite k.elem;
Inorder (k.right);   // besuche den rechten Teilbaum
```

Algorithmus - Preorder

Preorder (k)

Eingabe: Knoten k eines binären Baums mit Verweis auf linken (k.left) und rechten (k.right) Teilbaum sowie dem Element k.elem.

```
Verarbeite k.elem;
Preorder (k.left);     // besuche den linken Teilbaum
Preorder (k.right);   // besuche den rechten Teilbaum
```

Preorder – Programm


```
private void printPreorder (TreeNode n) {
   if (n != null) {// tree not empty
     println(n.getElement());
     printPreorder (n.getLeft());
     printPreorder (n.getRight());
   }
}
```

Algorithmus - Postorder

Postorder (k)

Eingabe: Knoten k eines binären Baums mit Verweis auf linken (k.left) und rechten (k.right) Teilbaum sowie dem Element k.elem.

```
Postorder (k.left);     // besuche den linken Teilbaum
Postorder (k.right);   // besuche den rechten Teilbaum
Verarbeite k.elem;
```

Levelorder (k)

Eingabe: Knoten k eines binären Baums mit Verweis auf linken (k.left) und rechten (k.right) Teilbaum sowie dem Element k.elem.

queue := leere Warteschlange; // vom Typ Queue

enter (k, q); // (aktuelle) Wurzel in queue aufnehmen

while not isEmpty(q) **do**

    Knoten n := leave (q);

    Verarbeite n.elem;

    enter (n.left, q); // linken Sohn in queue aufnehmen

    enter (n.right, q); // rechten Sohn in queue aufnehmen

od

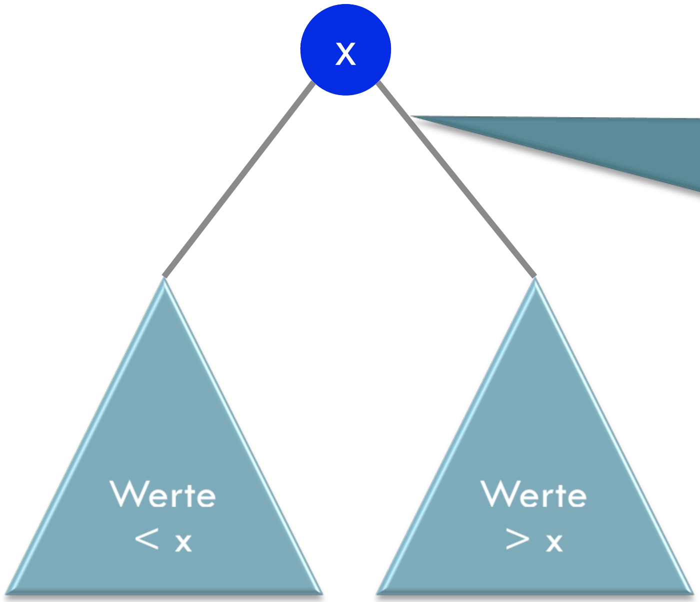
Suchbäume 1/2

- Bäume bisher: hierarchische Repräsentation und Organisation von Daten.
- Wichtigstes Einsatzgebiet von Bäumen: Unterstützung einer effektiven Suche.
- Voraussetzung für den Einsatz zum beim Suchen: Schlüsselwerte in den Knoten.

$\Rightarrow$ Dann ist es möglich Suchbäume aufzubauen.

Wir werden speziell binäre Suchbäume betrachten.

Geordneter Baum



Alle Werte im linken (*rechten*) Teilbaum sind kleiner (*größer*) als der Wert der Wurzel

Anforderungen an den Elementtyp:

- eine (totale) Ordnung ist definiert
- es ist eine Vergleichsoperation verfügbar (compareTo)

Eigenschaften binärer Suchbäume

Für jeden inneren Knoten k gilt:

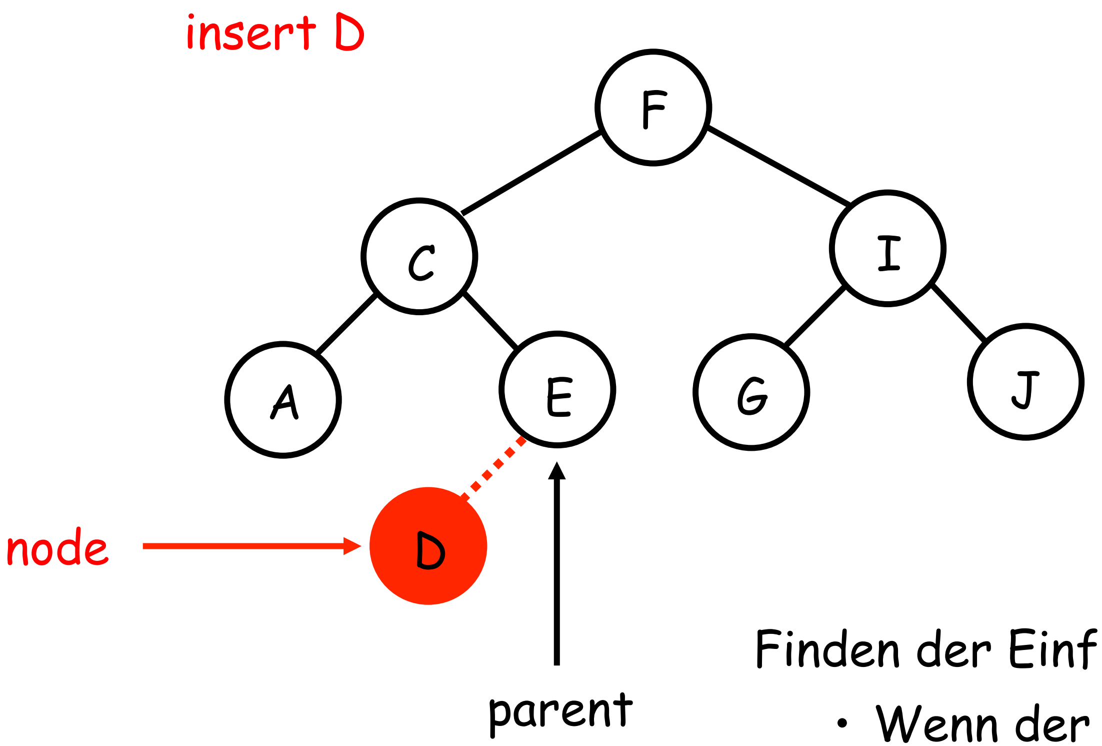
- Der Knoten k enthält einen Schlüsselwert k.key.
- Alle Schlüsselwerte in linken Teilbaum k.left sind kleiner als k.key.
- Alle Schlüsselwerte in rechten Teilbaum k.right sind größer als k.key.

⇒ Die Elemente in einem Suchbaum sind nach ihrem Schlüsselwert angeordnet.

⇒ Auf den Schlüsseln der Elemente ist eine totale Ordnung definiert.

⇒ Es wird eine Vergleichsoperation (compareTo) für die Schlüssel bereit gestellt.

Binärbaum - Einfügen



Finden der Einfügeposition:

- Wenn der Baum leer ist, wird der einzufügende Knoten die neue Wurzel.
- Wenn schon Knoten im Baum sind, suchen des Elternknotens des neuen Elements.

Binärbaum – insert 1/2

Die Klasse Element stellt eine Methode compareTo zur Verfügung. Diese liefert als Ergebnis:

- 0, wenn beide Elemente gleich sind
- < 0, wenn das 1. Element < 2. Element ist
- > 0, wenn das 1. Element > 2. Element ist

```
public boolean insert (Element i) {
   TreeNode parent = null;
   TreeNode child = root;

   while (child != null) { // at least 1 node in tree
     parent = child;
     if (i.compareTo(child.getElement()) == 0)
       return false;   // element already in tree, i is not inserted
     else if (i.compareTo(child.getElement()) < 0)
       child = child.getLeft(); // insert in left tree
     else
       child = child.getRight();  // insert in left tree
   }
```

Binärbaum – insert 2/2

```
// parent node found

    if (parent == null) // empty tree -> insert first node
      root = new TreeNode (i);
    else if (i.compareTo(parent.getElement()) < 0)
      parent.setLeft(new TreeNode(i)); // insert left from parent
    else
      parent.setRight(new TreeNode(i));  // insert left from parent

    return true;  // i successfully inserted
  }
}
```

Binärbaum – Löschen 1/2

- Löschen des Knotens k – Fallunterscheidung
  - Zuerst wird der der Elternknoten von k bestimmt (sofern es ihn gibt).

  a) Der Knoten k ist ein Blatt $\Rightarrow$ Es muss nur der Elternknoten (parent) bestimmt werden und dort die Referenz auf k entfernt werden.

  b) Der Knoten k besitzt nur ein Kind (child) $\Rightarrow$ Im Elternknoten wird die Referenz auf das Kind ersetzt durch die Referenz auf das Kind von k.

  c) Der Knoten ist ein innerer Knoten, d.h. er hat zwei Kinder. Dann gibt es 2 Möglichkeiten:

     i. Der Knoten k wird ersetzt durch den am weitesten rechts stehenden Knoten des linken Teilbaums, denn dieser ist in der Sortierreihenfolge der nächste Knoten.

     ii. Der Knoten k wird ersetzt durch den am weitesten links stehenden Knoten des rechten Teilbaums, denn dieser ist in der Sortierreihenfolge der nächste Knoten.

Binärbaum – Löschen 2/2

- Ersetzen des Knotens:
  - Austausch der Daten der Knoten
    - einfach, aber u.U. viel zu kopieren.
  - Aktualisieren der Referenzen der Knoten
    - Vermeidung aufwändigen Kopierens, das fehlerträchtig sein kann (bei flachen Kopien).
    - Bei balancierten Bäumen müssen auch immer wieder Referenzen aktualisiert werden. Dazu ist dies eine gute Vorbereitung.

Binärbaum – Löschalgorithmus 1/2

RemoveNode (T, x)

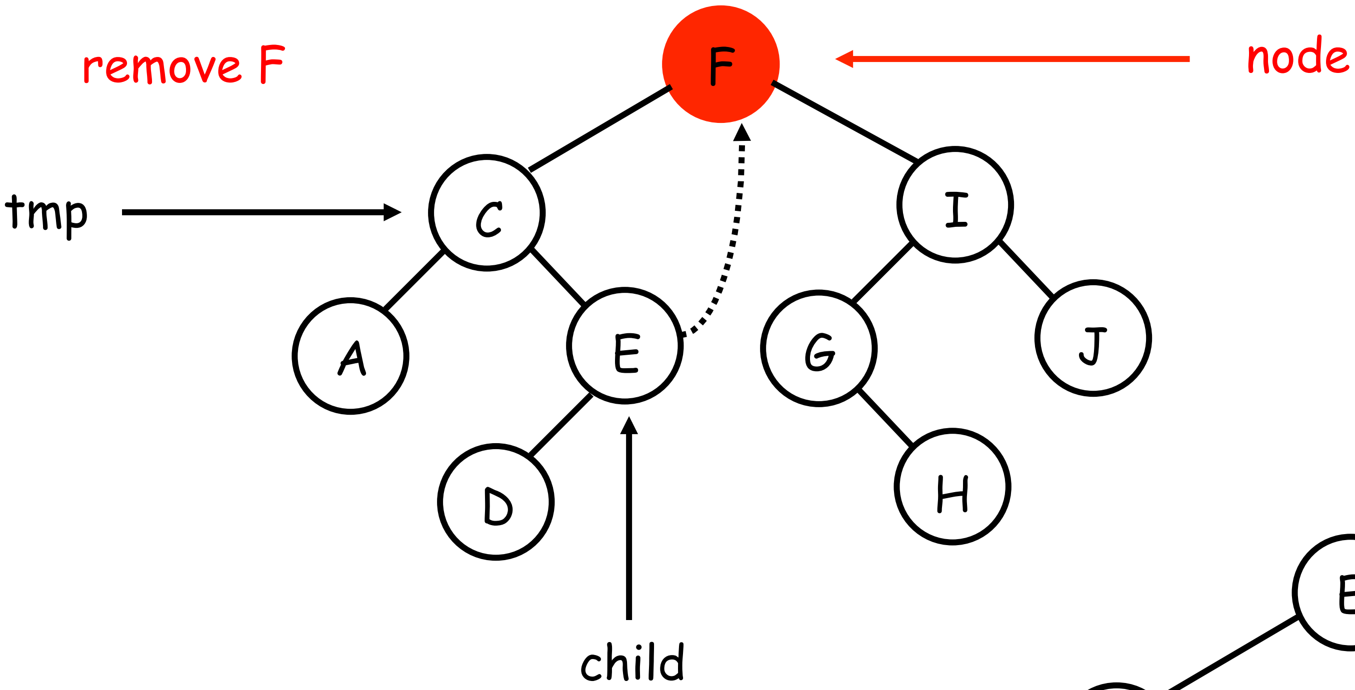
Eingabe: Baum T, Schlüssel x, des zu löschenden Elements.

```
k  := search (T, x);   // liefert Knoten k mit Schlüssel x im Baum T
if k == null then return fi;   // x nicht im Baum
if k == T.root  // Sonderfall: Wurzel soll gelöscht werden
if k.left == null then T.root := k.right;
else if k.right == null then T.root := k.left;
else
  child := größtes Element im linken Teilbaum von k (d.h. von k.left);
  ersetze k durch child;
fi
```

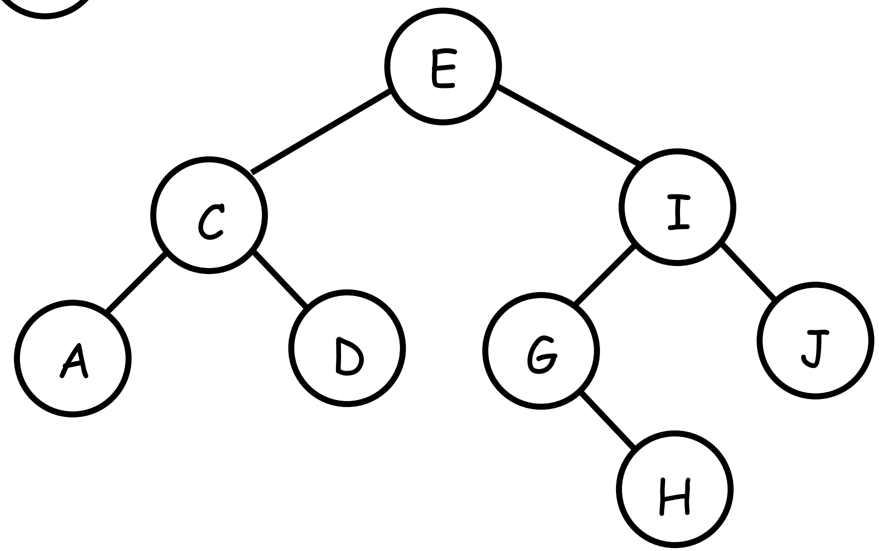
Binärbaum – Löschalgorithmus 2/2


```
else // Normaler Knoten soll gelöscht werden
if k.left == null then
  p := parent(k); // merke den Elternknoten
  if k ist linkes Kind von p then p.left := k.right;
                                  else  p.right := k.right; fi
else if k.right == null then
    if k ist linkes Kind von p then p.left := k.left;
                                    else  p.right := k.left; fi
else
    child := größtes Element im linken Teilbaum von k (d.h. von k.left);
    ersetze k durch child;
fi
fi
```

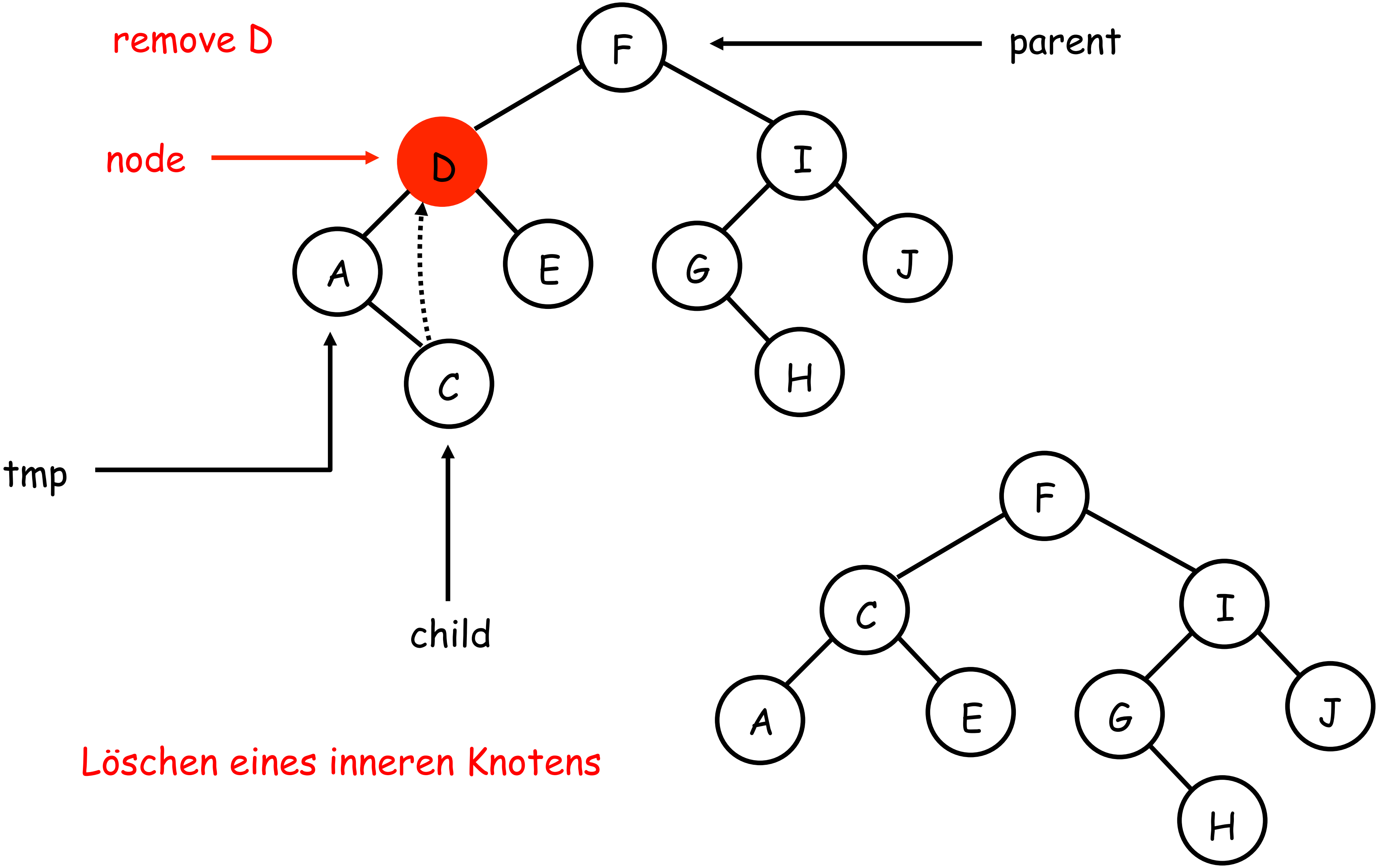
Binärbaum – Löschen/Fälle 1/2



Löschen des Wurzelknotens



Binärbaum – Löschen/Fälle 2/2



Löschen eines inneren Knotens

Komplexität der Operationen 1/2

Suchen, Einfügen, Löschen

Feststellung: es wird jeweils nur ein Pfad von der Wurzel bis zum entsprechenden Knoten bearbeitet.

- Der Aufwand wird bestimmt durch die Höhe des Baums $\Rightarrow$ die maximale Höhe h, die der Baum erreichen kann, bestimmt die Komplexität der Operationen, d.h. ist gleich O(h).
- Die Einfügereihenfolge der Elemente bestimmt das Aussehen des Baums, d.h. dieselbe Menge von Elementen führt bei unterschiedlicher Eingabereihenfolge zu unterschiedlichen Bäumen.

Beispiel: A, C, E, F, G, H, I und F, C, H, A, E, G, I

Komplexität der Operationen 2/2

- Welche Höhe kann ein Baum mit n Knoten erreichen?
- Im **schlechtesten** Fall: Baum entartet zu einer Liste ⇒ **h = n**.
- Im **besten** Fall: Jeder innere Knoten hat immer 2 Nachfolger ⇒ auf Level 0 gibt es einen Knoten, auf Level 1 gibt es 2 Knoten, auf Level 2 gibt es 4 Knoten etc. auf Level k gibt es 2^k Knoten. D.h. ein Baum der Höhe k+1 (wenn Level = k) kann $1 + 2 + 4 + \ldots + 2^{k-1} + 2^k$ Knoten fassen. Sind n Knoten in einem solchen Baum, dann ist die Höhe $\mathbf{h = \log_2 n}$.
- Suchbäume mit logarithmischer Höhe nennt man ausgeglichene oder balancierte Bäume.
- Ein Baum heißt ausgeglichen, wenn bei einer gegebenen Zahl n von Elementen die Höhe möglichst klein ist.