DHBW Lörrach
Duale Hochschule Baden-Württemberg

Algorithmen und Komplexität
TIF 21A/B
Dr. Bruno Becker

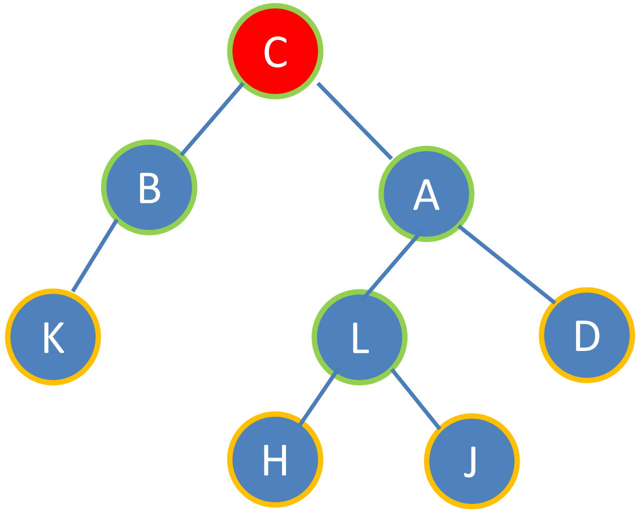
5. Bäume

Bäume

- **Binärbäume**
- Binäre Suchbäume
- Balancierte Suchbäume

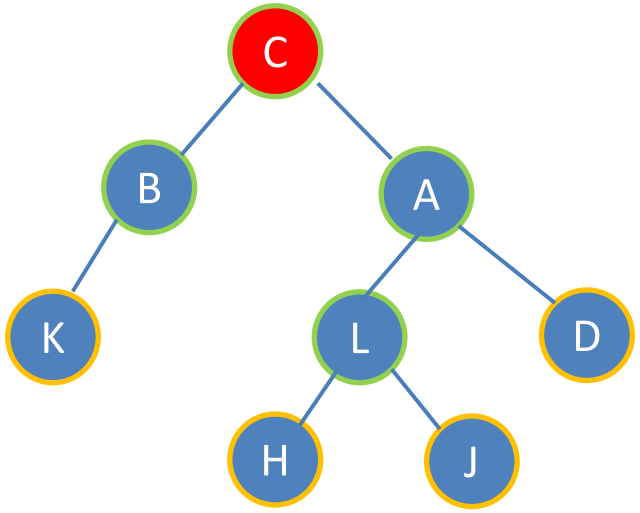
Bäume – Definitionen und Begriffe

- **Baum –** Verallgemeinerte Listenstruktur. Jedes Element (***Knoten***) hat maximal ***n*** (*n >1)* Nachfolger (***Söhne, Kinder***)
- **Ordnung** des Baumes: Max. Anzahl Nachfolger; ***n = 2 : Binärbaum***
- **Wurzel** – Knoten an der Spitze des Baumes hat **keinen** Vorgänger
- **Blatt** – Knoten ohne Nachfolger
- **Innerer Knoten** – Knoten mit Nachfolger



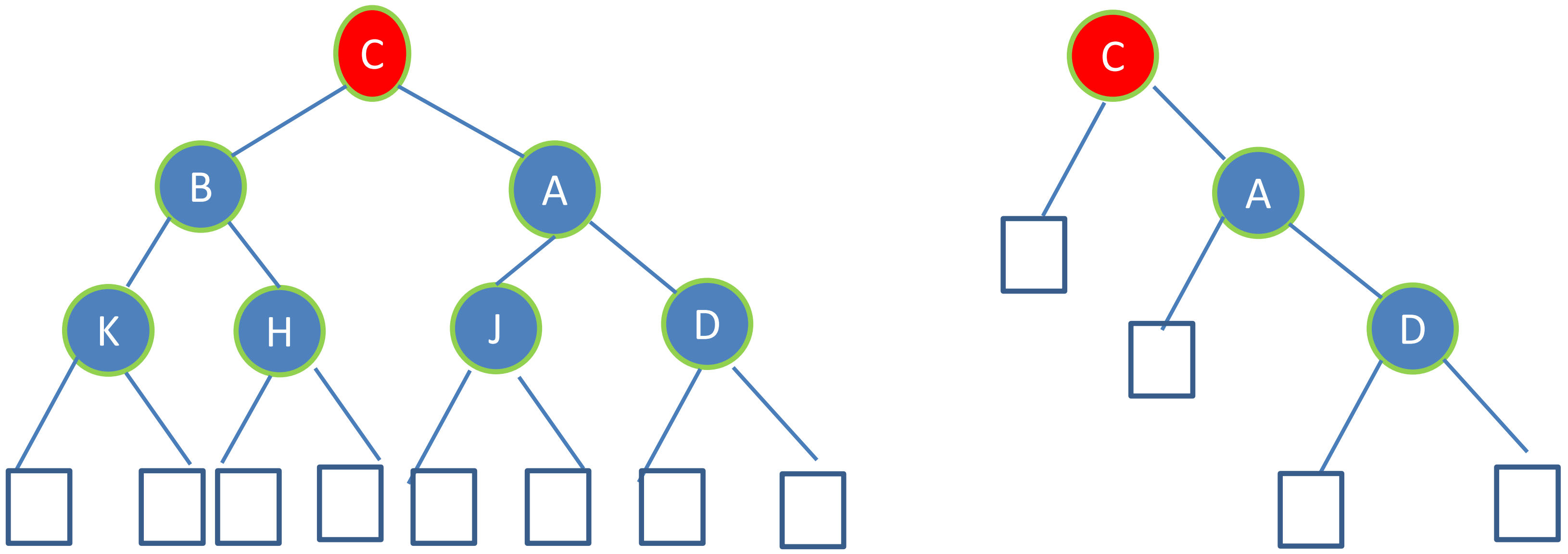
Bäume – Definitionen und Begriffe (2)

- **Pfad –** Folge von Knoten $P_0, \ldots, P_k$ mit der Eigenschaft, dass P_{i+1} ist Sohn von P_i (für alle $i < k$). Pfad hat ***Länge*** k
  Beispiel C,A,L,H – Länge 3
- Jeder Knoten hat eindeutigen Pfad zur Wurzel. Die Pfadlänge heißt **Tiefe** des Knotens
- **Tiefe** eines Baumes ist die maximale Tiefe eines Knotens
- Die Knoten mit der gleichen Tiefe haben das gleiche **Level(Niveau, Ebene)**
- Im **vollständigen Baum** haben alle Blätter gleiche Tiefe – Alle Level voll besetzt



Binärbäume – Eigenschaften

- Innere Knoten mit Werten, Blätter ohne Werte -> ***Suchbaum***
- Binärbaum mit *n* inneren Knoten hat n + 1 Blätter ; **Anzahl Knoten** auf Level $i \leq 2^i$
- Baum der Tiefe *t* hat höchstens $2^{t+1}-1$ Knoten und mindestens *t+1* Knoten
- Umgekehrt hat Baum mit *n* inneren Knoten maximal Tiefe *n;* minimal $log_2 n + 1$

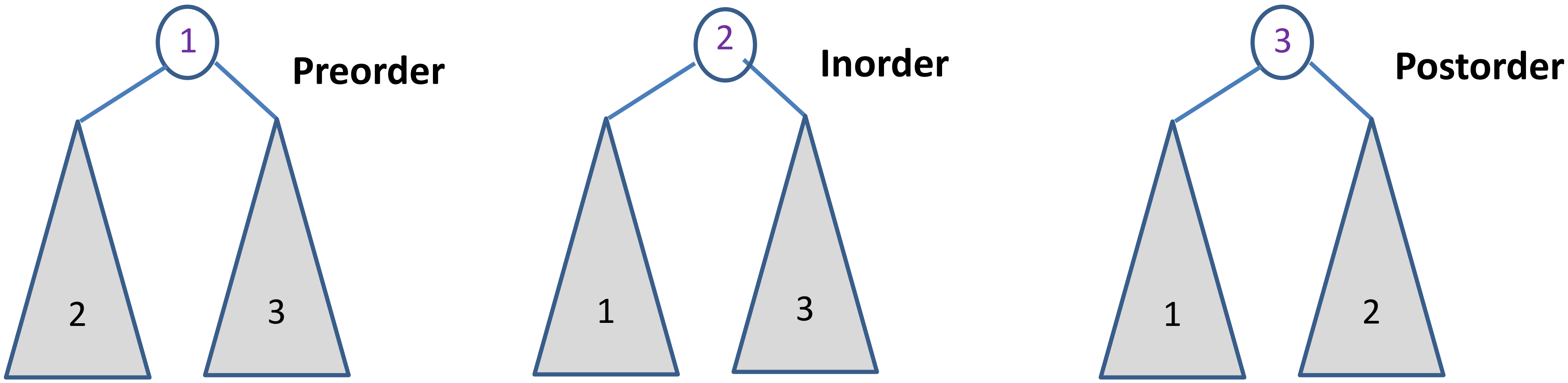


Binärbäume – Implementierung als verkettete Struktur

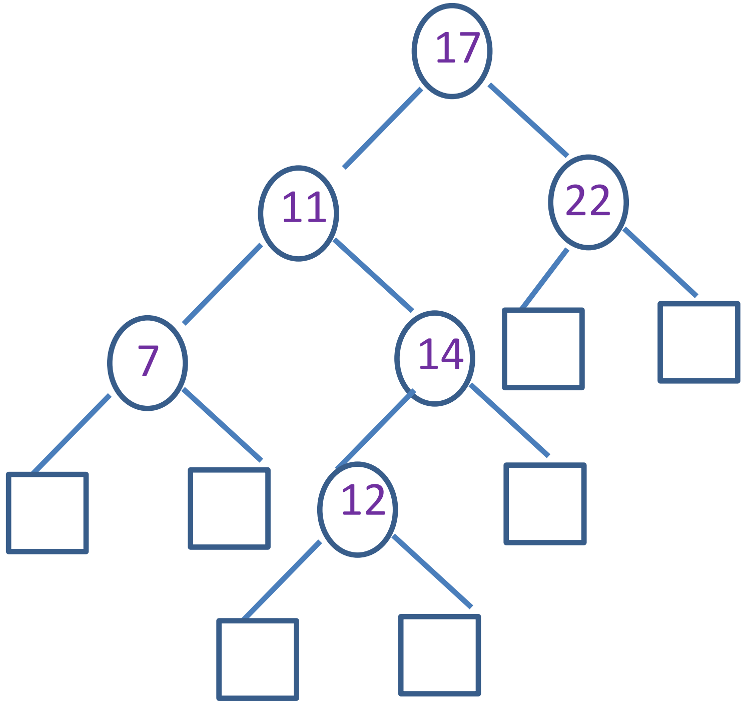
```
Public class  Tree {
      private Node root = null ;  // Zeiger auf Wurzel

      private class Node;  // Knoten
            private Key key // Schlüssel
            private Value value // Datensatz
            private Node left,  right// linker, rechter Sohn
      }
}
```

Binärbäume – Durchlaufanordnungen

- **Hauptreihenfolge *(Preorder):*** Wurzel; Linker Teilbaum; Rechter Teilbaum
- **Symmetrische Reihenfolge *(Inorder):*** Linker Teilbaum; Wurzel ; Rechter Tb.
- **Nebenreihenfolge *(Postorder):*** Linker Teilbaum; Rechter Teilbaum; Wurzel



Binärbäume – Beispiel Traversierung



Preorder: 17-11-7-14-12-22

Inorder: 7-11-12-14-17-22

Postorder: 7-12-14-11-22--17

Übung: Gegeben Binärbaum – Wie ist Preorder, Inorder- und Postorder-Ausgabe?

Bäume mit variabler Anzahl von Teilbäumen

- Bäume mit variabler Anzahl von Söhnen
- **Anwendungen**
  - Hierarchische Menüstrukturen in Benutzeroberfläche (z.B. Website)
  - Dateisystem in Windows und Unix
  - Entscheidungsbäume mit unterschiedlicher Zahl von Möglichkeiten
  - Verarbeitung von XML-Dateien

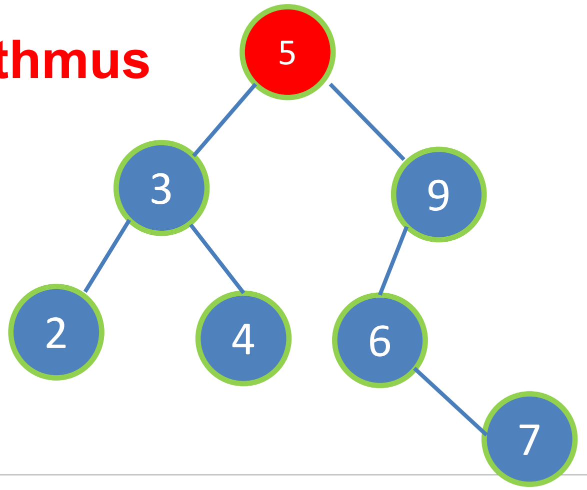
Bäume

- Binärbäume
- **Binäre Suchbäume**
- Balancierte Suchbäume

Binäre Suchbäume - Suche

- Schlüssel sind in inneren Knoten, Blätter sind leer
- **Suche:** Nach Schlüssel *k.* Starte mit Wurzel.
  Ist der Knoten ein Blatt → Schlüssel nicht gefunden;
  Ist der Schlüssel gleich *k* → Element gefunden. Ansonsten,
  falls Schlüssel < k → Suche im linken Teilbaum,
  sonst → Suche im rechten Teilbaum

Übung: Such-Algorithmus




```
Node search (Node x, int key)
 {
  if (x==null)
     return null; // Schlüssel nicht gefunden
  if (key < x.key)
     return search(x.left, key);
 else if (key > x.key)
    return search (x.right, key);
  else
     return x
 }
```

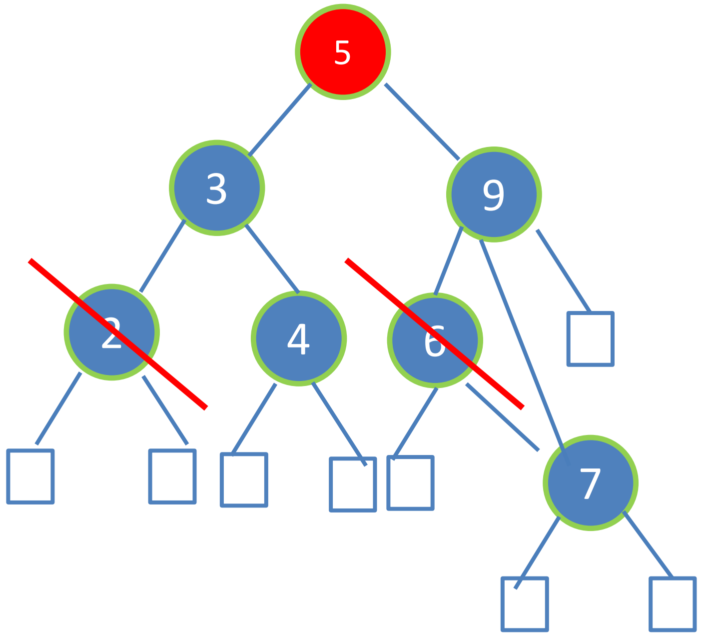
Binäre Suchbäume –Einfügen

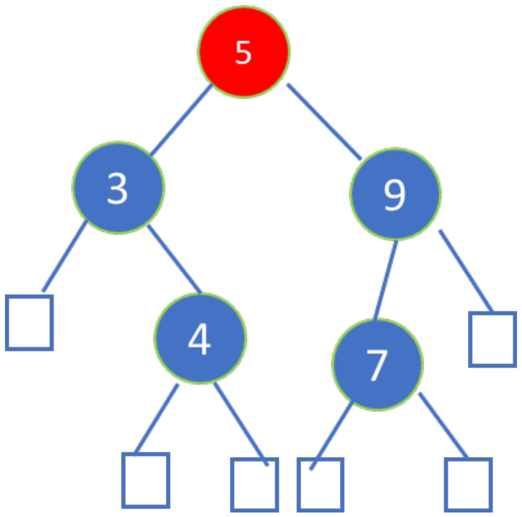
Einfügen: Wie Suchen. Statt erfolgloser Suche an der Stelle des Blattes das Element einfügen.

```
void insert (Node root, Node ins)
 { if (root==null)
   { root = ins;
     return;
   }
   Node parent, x = root;
   while (x!=null)
    {   parent = x;
        if (ins.key < x.key)
          { x= x.left;
          }
        else
          { x= x.left;
          }
      }  // Suche endet in Blatt parent
        if (ins.key < parent.key)
            { parent.left = ins;
            }
          else
            { parent.right = ins;
            }
 }
```

Binäre Suchbäume – Löschen

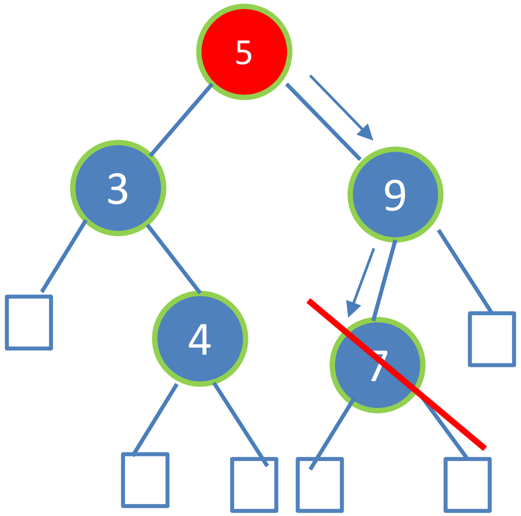
1. Suchen
2. Nach erfolgreicher Suche dieses Element löschen
   - Wenn Element keinen Nachfolger hat => Element kann einfach gelöscht werden (Beispiel 2)
   - Wenn Element **einen** Nachfolger hat => Ersetze Element mit seinem Nachfolger (Beispiel 6)
   - **Was passiert, wenn Element zwei Nachfolger hat? (Beispiel 5)**

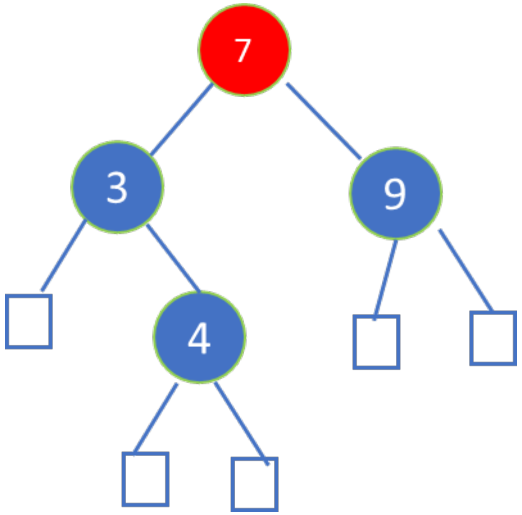




Binäre Suchbäume – Löschen

- Element x hat 2 Nachfolger (Beispiel 5)
- Suche nächstgrößeres Element *symmetrischer Nachfolger(x)*
  - *Gehe einmal nach rechts*
  - *Danach nach links bis Blattlevel*
- Diese Suche endet in einem Blatt. (Beispiel 7)
- Lösche das Blatt und ersetze x durch *symmetrischer Nachfolger(x)*





Übung: Algorithmen für
- Symmetrischer Nachfolger
- Löschen eines Elements

Binäre Suchbäume – Form binärer Suchbäume

- Zu einer Menge von Schlüsseln gibt es viele verschiedene binäre Suchbäume
- Die Form hängt von der *Reihenfolge des Einfügens* ab!

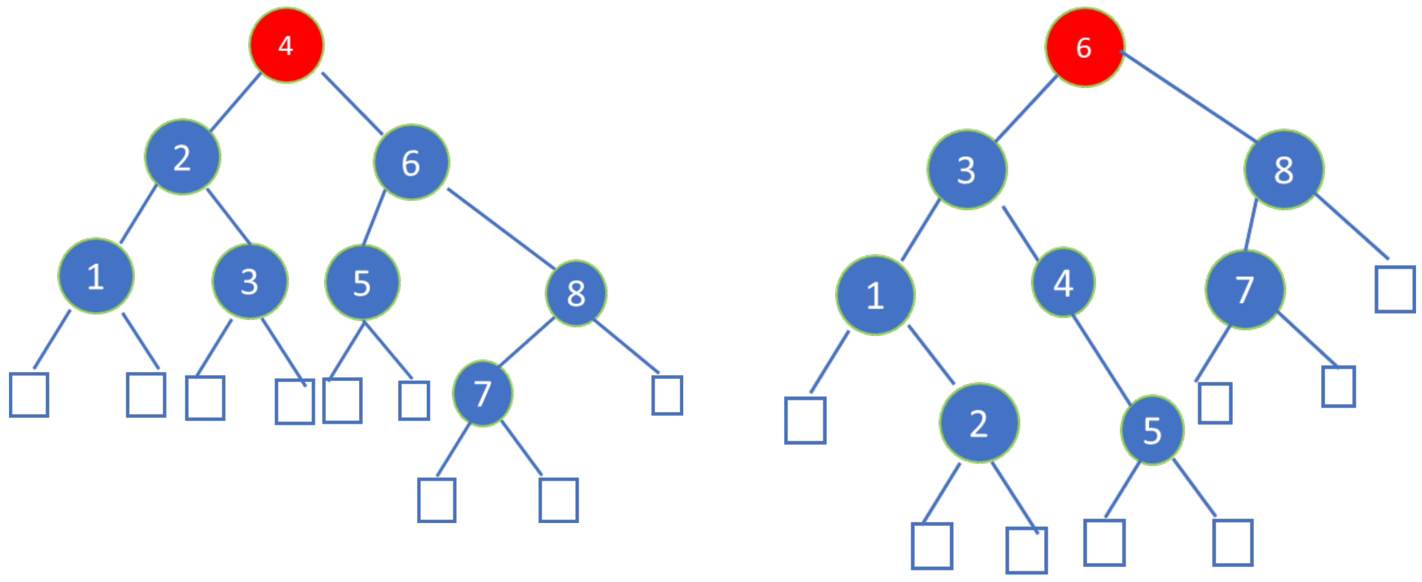
Beispiel: {1,2,3,4,5,6,7,8}

4 2 6 8 1 3 5 7

6 3 1 8 2 7 4 5

1 2 3 4 5 6 7 8

➔ Lineare Liste nach rechts



Binäre Suchbäume – Analyse der Operationen

- **Aufwand** für Suche dominiert; Einfügen und Löschen nur konstanter Zusatzaufwand (beim Löschen plus Suche symmetrischer Nachfolger)
- **Wort Case Analyse:**
  - **Erfolgreiche Suche:** ***n***
  - **Erfolglose Suche:** ***n***

Der Worst Case ist selten. Man kann zeigen:

Wenn man *n* verschieden Schlüssel *in zufälliger Reihenfolge* in binären Suchbaum einfügt, dann ist:

- **Die erwartete Tiefe des Baumes ca.** $2.99 \log_2 n$
- **Die erwartete Anzahl der Vergleiche für Suche/Einfügen ca.** $1.39 \log_2 n$

Datenstrukturen für Suche: Vergleich (Tilde-Notation)

Datenstruktur	Aufwand Suchen (worst case)	Aufwand Einfügen (worst case)	Aufwand Suchen (avg. case)	Aufwand Einfügen (avg. case)
Lineare Liste sortiert (unsortiert)	n (n)	n (1)	$n/2$ ($n/2$)	$n/2$ (1)
Binäre Suche Sortiertes Array	$\log_2 n$	n	$\log_2 n$	$n/2$
Binärer Suchbaum	n	n	$1{,}39 \log_2 n$	$1{,}39 \log_2 n$

Bäume

- Binärbäume
- Binäre Suchbäume
- **Balancierte Suchbäume**

Binäre Suchbäume – Balancierte Suchbäume

- **Idee für Verbesserung Aufwand:** Bäume möglichst vollständig (*balanclert*) halten.
- Dafür muss man nach Einfügen oder Löschen eines Elements wieder balancierte Struktur herstellen
- 1. Vorschlag hierzu aus dem Jahr 1962: **AVL-Bäume**
- Ein binärer Baum ist ein **AVL-Baum (*höhenbalanciert*),** wenn für jeden Knoten *p* gilt: Die Höhe des linken Teilbaums von *p* unterscheidet sich von der Höhe des rechten Teilbaums höchstens um 1.

→ Damit ist logarithmische Höhe des Baumes garantiert, d.h. sehr gutes WorstCase-Verhalten für Suche.

Balancierte Suchbäume – AVL-Bäume

- Damit ein AVL-Baum bei Einfügen und Löschen von Elementen höhenbalanciert bleibt, muss man nach jeder Operation prüfen, ob Baum noch höhenbalanciert ist oder eine Korrektur vornehmen
- **Idee:**
  - Speichere an jedem Knoten *p* die Höhendifferenz *bal* (*p*) = *Höhe rechter Teilbaum – Höhe linker Teilbaum*
  - Für AVL-Bäume gilt: *bal* (*p*) ε $\{-1,0,+1\}$
  - Einfügen: Suche endet in Blatt. Füge dort Element ein. Überprüfe den Weg zurück zur Wurzel, ob innerer Knoten noch höhenbalanciert sind

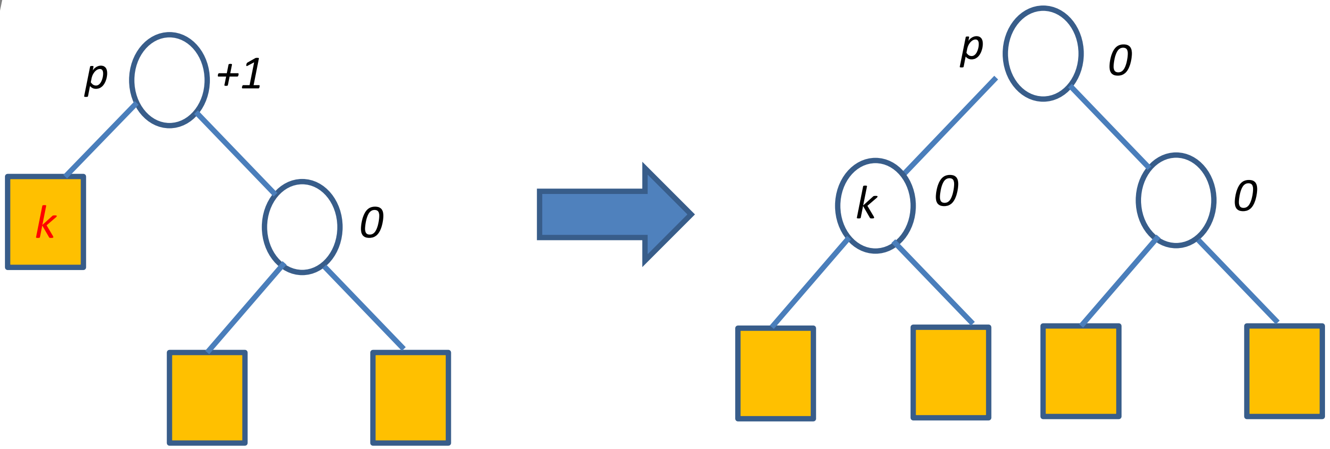
    → Falls ja, ist Baum weiterhin AVL-Baum

    → Sonst „Korrektur-Operation“

Balancierte Suchbäume – Einfügen

- Schlüssel *k* soll in Baum eingefügt werden;
- Wenn Baum leer ist, dann füge *k* ein -> fertig.
- Ansonsten sei *p* sei der Vater des Blattes, an dem Suche nach *k* endet.

1. *bal(p) = +1*

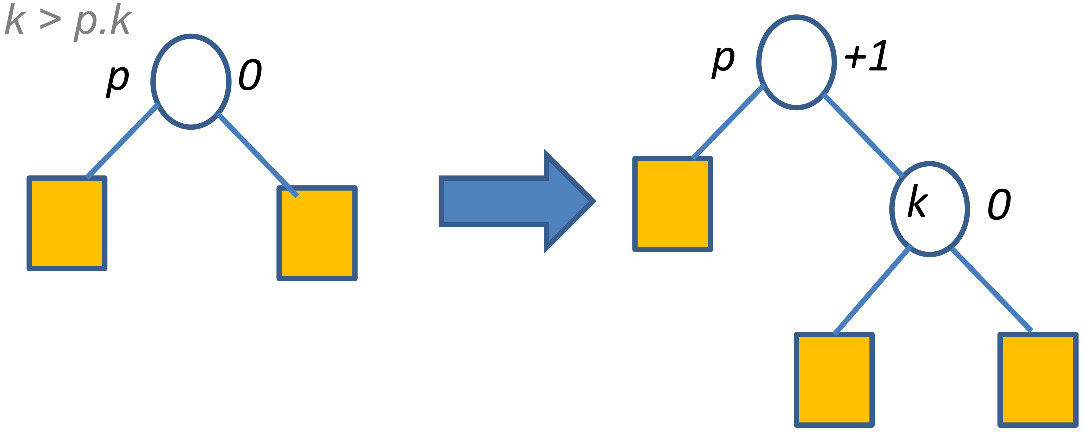


2. *bal(p) = -1* → genauso

Balancierte Suchbäume – Einfügen (2)

3. bal(p) = 0

3.1 k > p.k



- Höhe im rechten Teilbaum von *p* ist um +1 gewachsen
- *bal (p) = +1*

3.2 k < p.k

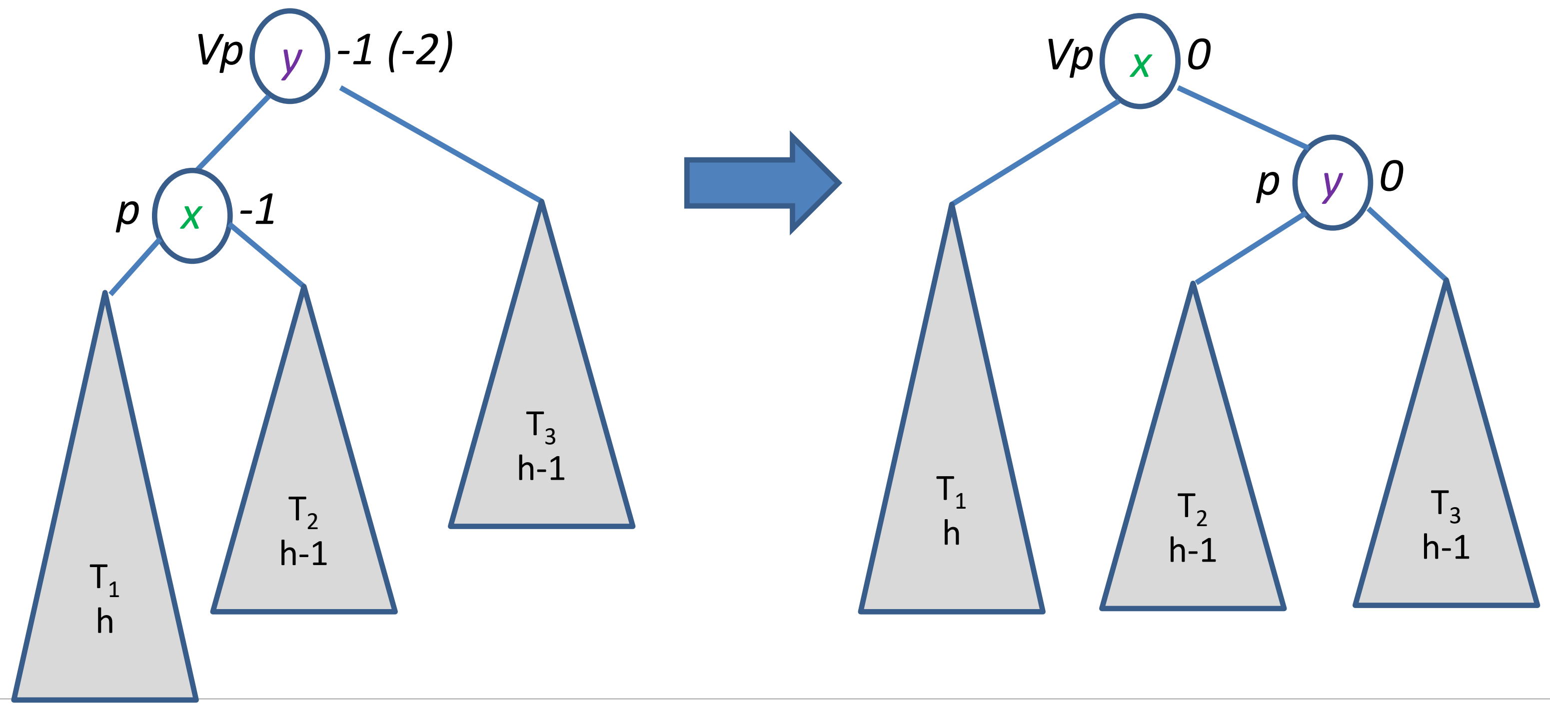
- Höhe im linken Teilbaum von *p* ist um +1 gewachsen
- *bal (p) = -1*

→ Prüfe nun Balance-Bedingung für Vater Vp von p

Balancierte Suchbäume – Einfügen (3)

Sei *p* der linke Sohn seines Vaters *Vp* (*p* rechter Sohn von *Vp* als Übung)

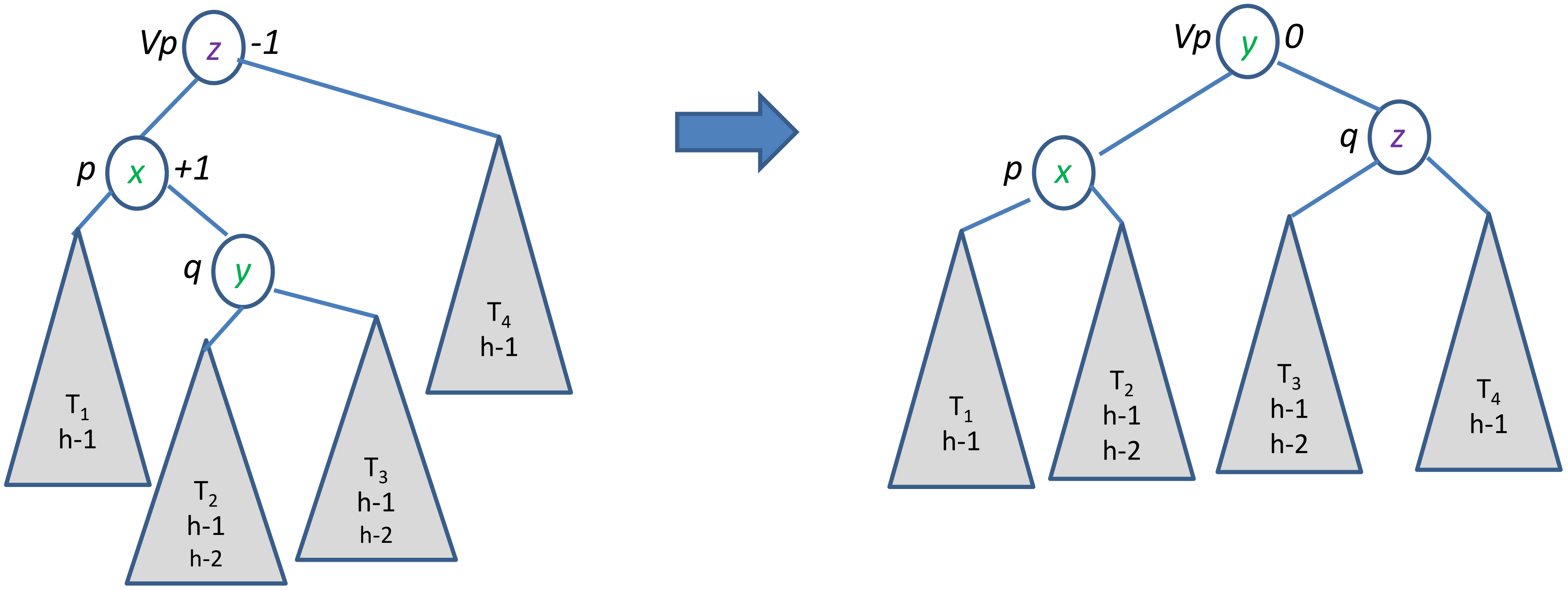
1. *bal (Vp) = +1 → bal (Vp) = 0* **fertig.**
2. *bal (Vp) = 0 → bal (Vp) = -1; Überprüfe nun Balance-Bedingung für Vater von Vp*
3. *bal (Vp) = -1 und bal (p) = -1 → bal (Vp) = -2* **Korrektur erforderlich! Rotation**



Balancierte Suchbäume – Einfügen (4)

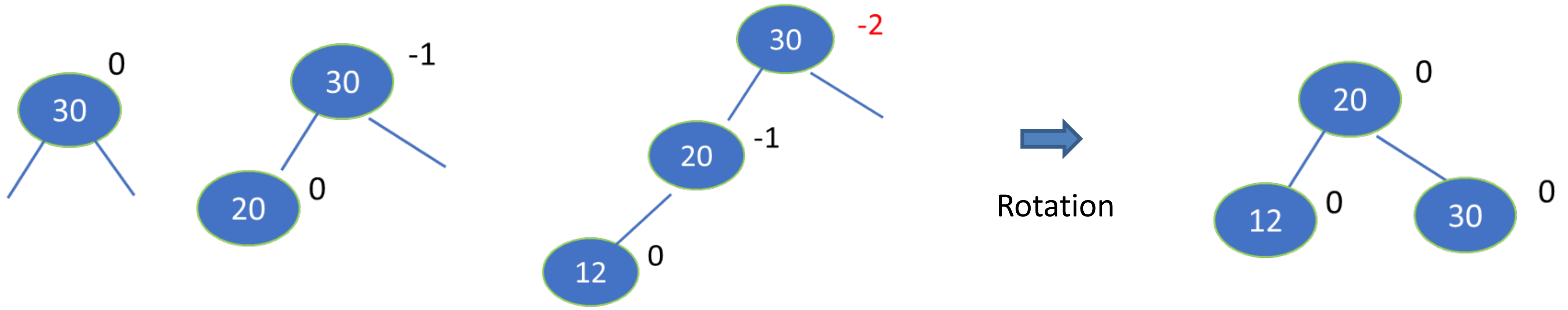
Sei *p* der linke Sohn seines Vaters *Vp*

4. bal (Vp) = -1 und bal (p) = +1 => Doppelrotation



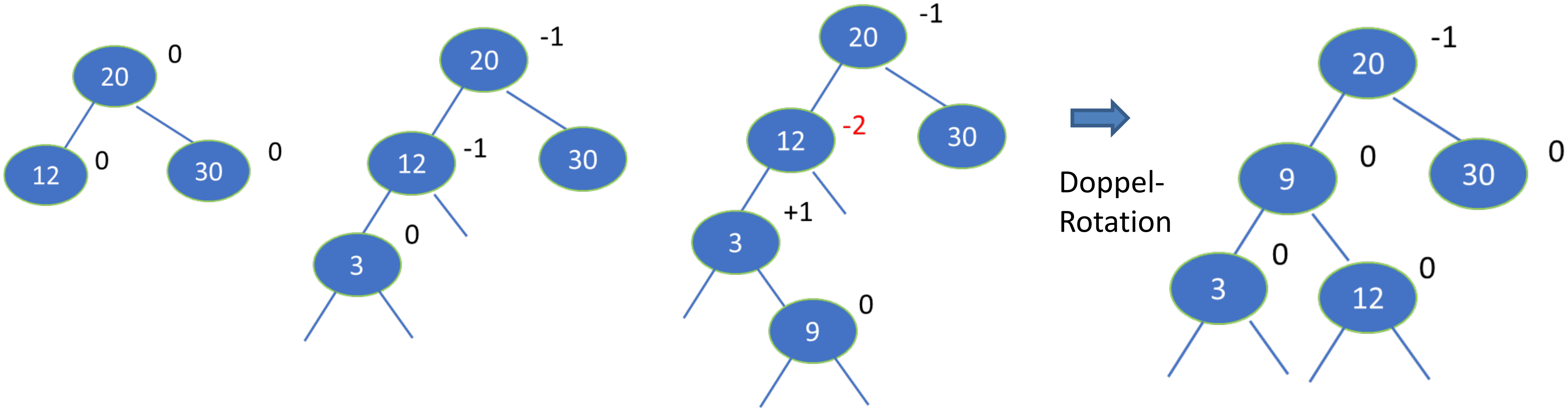
Übung

- Einfügen der Folge 30-20-12-3-9 in leeren AVL-Baum



Übung

- Einfügen der Folge 30-20-12-3-9 in leeren AVL-Baum



Balancierte Suchbäume – Löschen im AVL-Baum

- **Geht ähnlich zum Einfügen**
  - Unterschied: Nach einer Korrektur-Operation (Rotation oder Doppelrotation) ist man nicht zwingend fertig, sondern Korrektur-Operation kann im schlimmsten Fall für jeden Knoten auf dem Suchpfad zurück zur Wurzel erforderlich sein

- *Siehe Ottmann/Widmayer Kap 5.2.1 S. 292ff*

Balancierte Suchbäume – Zusammenfassung

- Balancierte Suchbäume können nicht zur Liste degenerieren
  - Damit Suchen immer in $O(\log_2 N)$
  - Das gilt auch für Einfügen und Löschen
- Damit sind Balancierte Suchbäume Worst case-Optimal für Suchen, Einfügen und Löschen
- Neben AVL-Bäumen gibt es weitere balancierte Suchbäume
  - Bruder-Bäume
    - modifizierter AVL-Baum, unäre Knoten statt Balancewert 1 oder -1
  - 2-3 Bäume
    - Knoten enthält entweder 1 Schlüssel und 2 Kinder, oder
    - 2 Schlüssel und 3 Kinder