FHV

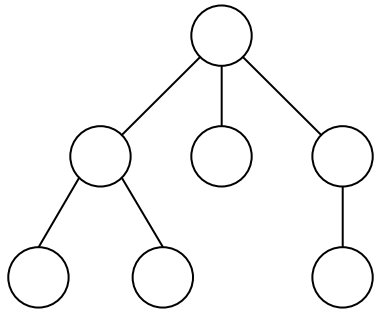
Algorithmen & Datenstrukturen

Bäume und Binäre Suchbäume

Wolfgang Auer

Bäume

- Ein *Baum* (tree) ist eine zwei- oder mehrdimensionale, *hierarchische* Datenstruktur, bei der jedes Element mit genau einem *Vorgänger* und mehreren *Nachfolgern* verknüpft sein kann.



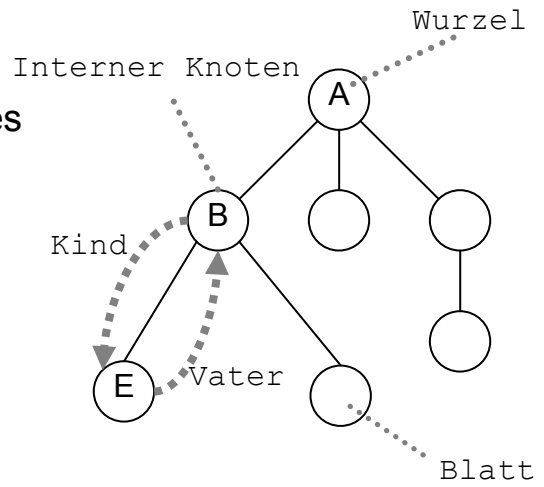
- Beispiele
  - Organisationsstrukturen von Betrieben
  - Dateisystem
  - Syntaxbaum
  - Entscheidungsbäume
  - Organisation von Wissen
  - ...

Begriffe und Definitionen

FHV

Ein Baum ist eine Menge von *Knoten* und *Kanten*

- *Knoten* sind beliebige Objekte, die durch Kanten verbunden werden
- Spezielle Knoten:
  - *Wurzel (root)* ist der Ursprung eines Baumes und hat keinen Vaterknoten
  - *Blätter (leaves)* haben keine Kind-Knoten
  - *Interne Knoten* haben einen Vater- und mindestens einen Kinderknoten
- Folgt man den Kanten, die von einem Knoten wegführen, erreicht man dessen *Kinder (children).*
- Die Anzahl der Kinder eines Knotens wird als sein *Grad* bezeichnet
- Die maximale Anzahl der Kinder eines Knotens als die *Ordnung des Baumes.*

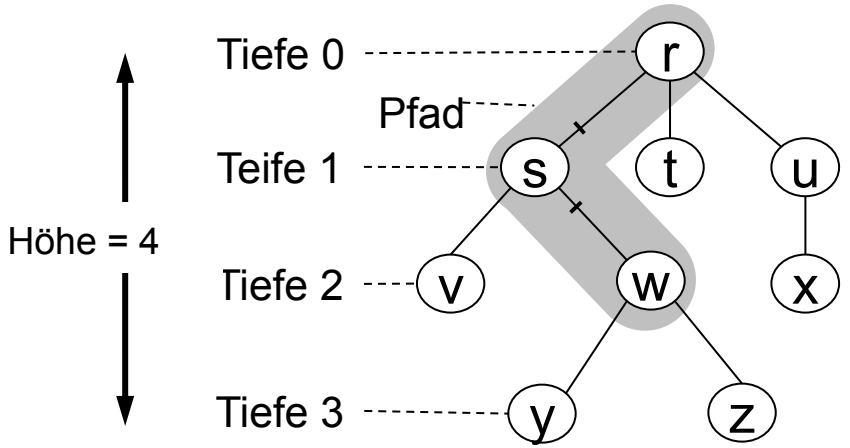


- Grad von *B* = *2*
- *Ordnung des Baumes = 3*
- *E* ist ein Kind von *B*
- *B* ist der Vater von *E*

Pfad

- Ein *Pfad (path)* bezeichnet eine geordnete Liste von Knoten, für die gilt p_{i+1} ist Kind von p_i für $0 \leq i \leq k$ und p_0 mit p_k verbindet
  - In einem Baum besteht **genau ein Pfad** von einem Knoten zu jedem seiner Nachkömmlinge (*descendant).* Die Knoten, die sich auf diesem Pfad befinden, werden als Vorfahren (*ancestors*) bezeichnet

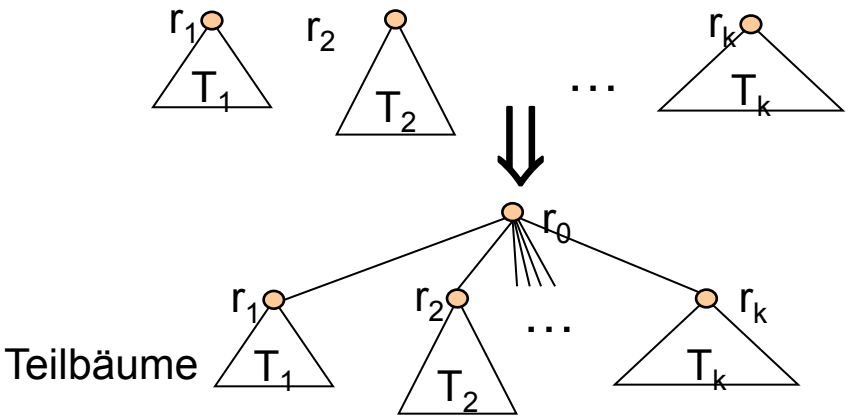
- Die *Tiefe (depth)* eines Knotens oder dessen *Level* wird durch die Länge des Pfads von der Wurzel bestimmt.



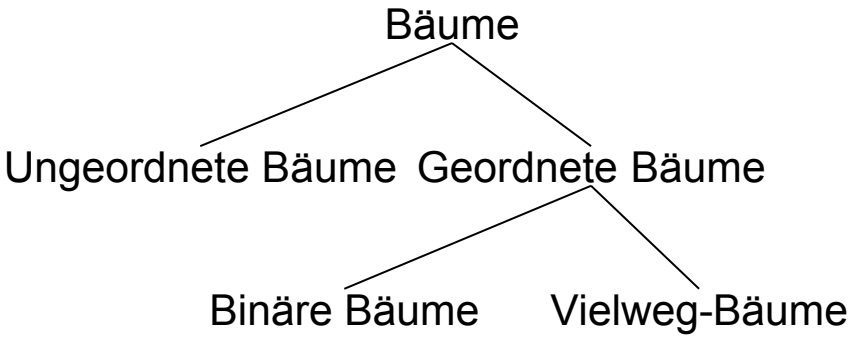
- Die größte Tiefe aller Knoten + 1 legt die *Höhe* des Baumes fest.

Rekursive Definition

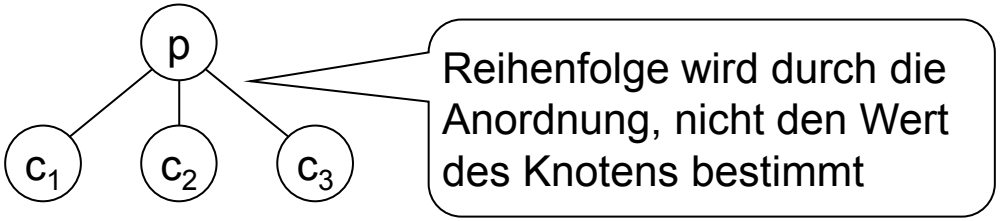
- Ein einzelner Knoten ohne Kanten ist ein Baum.
- Seien T_1, ..., T_k ($k > 0$) Bäume ohne gemeinsame Knoten .
  - Die Wurzeln dieser Bäume seien r_1, ..., r_k.
  - Ein Baum T_0 mit der Wurzel r_0 besteht aus den Knoten und Kanten der Bäume T_1, ..., T_k und neuen Kanten von r_0 zu den Knoten r_1, ..., r_k.
- Der Knoten r_0 ist dann die neue Wurzel und T_1, ..., T_k sind Unterbäume von T_0.



Einteilung der Bäume



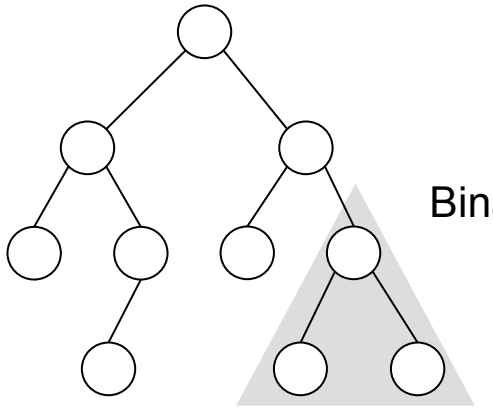
- In *geordneten Bäumen* besteht für die Knoten eine festgelegte Reihenfolge der Kinder.



- Binäre Bäume sind Bäume, bei denen jeder Knoten einen Grad ≤ 2 haben kann.
- Vielweg-Bäume besitzen eine Ordnung > 2

Binäre Bäume

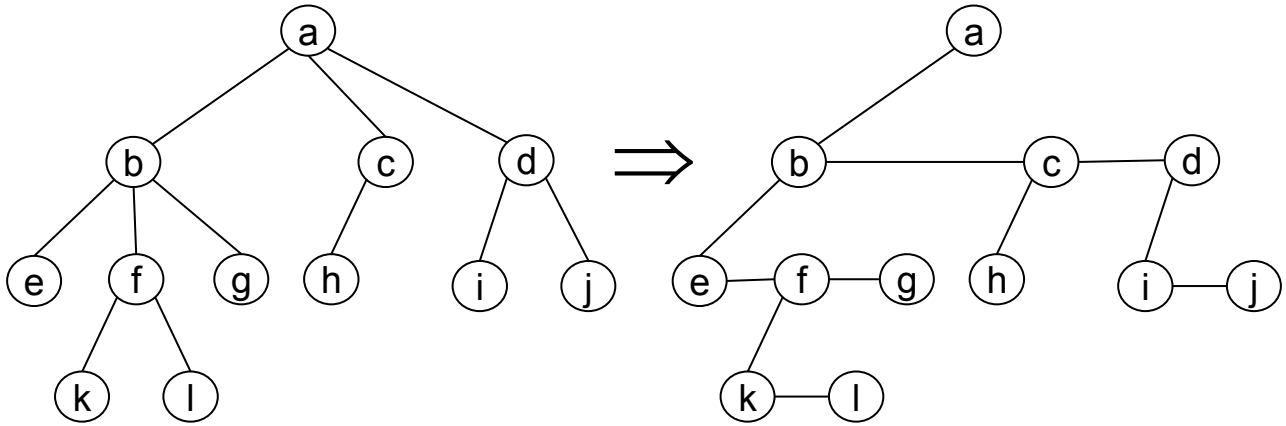
- Ein *binärer Baum* ist entweder ein leerer Baum oder ein Knoten, der einen *linken* und *rechten binären Teilbaum* hat.



- Binäre Bäume sind eine Sonderform der Vielweg-Bäume. Durch die Beschränkung der Ordnung des Baums können effiziente Datenstrukturen und Algorithmen verwendet werden
- Vielweg-Bäume können in binäre Bäume transformiert werden

Transformation eines Vielwegbaumes

- Algorithmus:
  - Verknüpfe alle Kinder eines Knotens von links nach rechts
  - Lösche, mit Ausnahme der linksten, alle Kanten, die zu dem Kind führen
  - Wiederhole diese Aktionen für alle Knoten des Baumes



Formen binärer Bäume

- *Leerer* binärer Baum

  Binärer Baum ohne Knoten

- *Streng* binärer Baum

  Jeder innere Knoten hat sowohl einen linken als auch rechten Nachfolgeknoten

- *Vollständiger* binärer Baum

  Alle Blätter befinden sich auf der gleichen Ebene oder auf adjazenten (benachbarten) Ebenen, wobei die Blätter der unteren Stufe soweit links wie möglich stehen

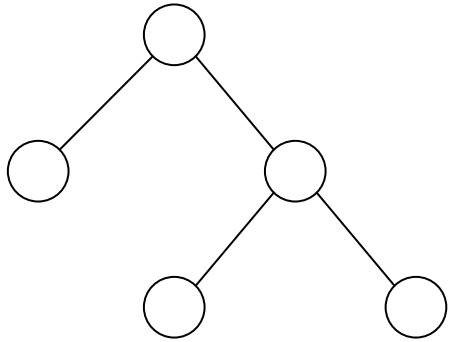
- *Ausgeglichener* binärer Baum

  Für jeden Knoten ist der Betrag der Differenz der Höhen des linken und rechten Teilbaums maximal 1.

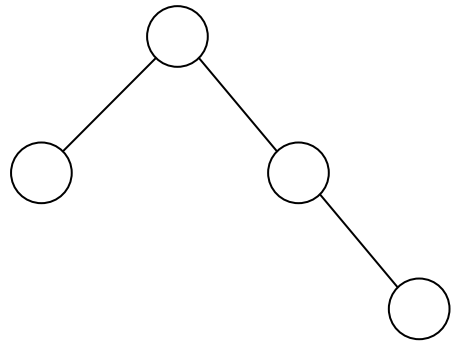
Die Eigenschaften der Bäume sind maßgeblich für die Effizienz der verwendeten Algorithmen!

Streng Binärer Baum

- Jeder innere Knoten hat sowohl einen linken als auch rechten Nachfolgeknoten d.h. den Grad 2



Streng binärer Baum

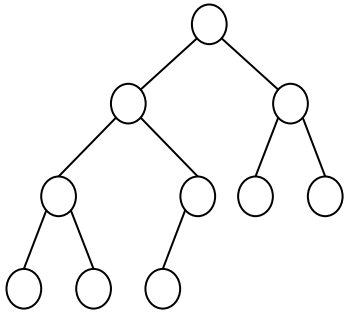


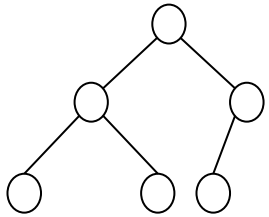
Kein streng binärer Baum

Vollständiger Binärer Baum

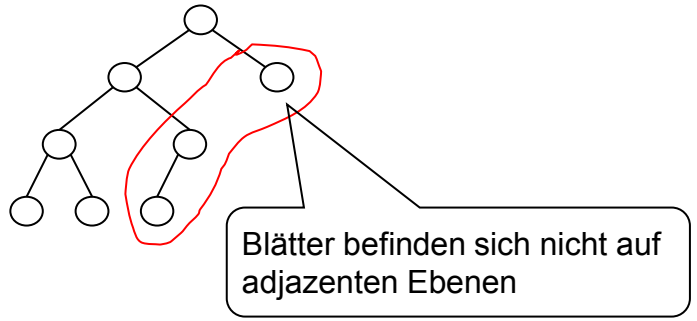
- Alle Blätter befinden sich auf der gleichen Ebene oder auf adjazenten (benachbarten) Ebenen, wobei die Blätter der unteren Stufe soweit links wie möglich stehen.

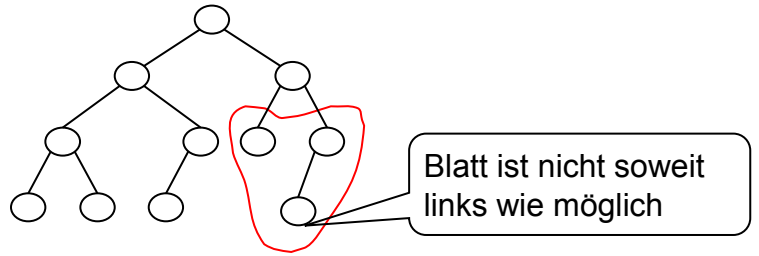
Vollständige binäre Bäume





Keine vollständigen binären Bäume

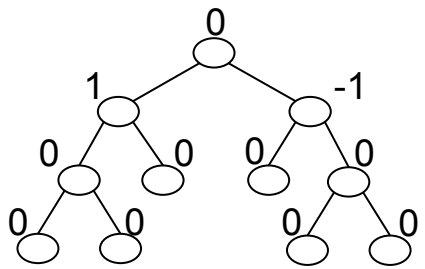




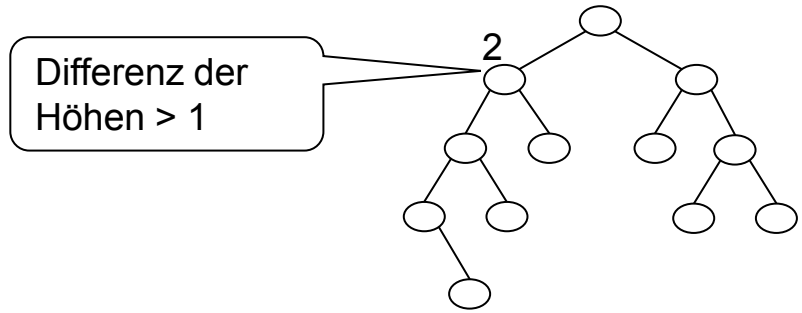
Ausgeglichener Binärer Baum

- Ein Baum ist ausgeglichen (balanciert) wenn für jeden Knoten gilt, dass sich die Höhe des linken Teilbaumes und die Höhe des rechten Teilbaumes um maximal 1 unterscheiden.

Balancierter binärer Baum



Unbalancierter binärer Baum



Eigenschaften Binärer Bäume

- Ein Binärer Baum hat auf dem Level *l* maximal 2^l, für $l \geq 0$ Knoten
- Ein streng binärer Baum, der alle Blätter auf der Höhe *h* hat, hat $2^h - 1$ Knoten, davon sind 2^{h-1} Blätter.
  - Beweis: Vollständige Induktion anhand der Tiefe

$$\textit{Induktionsanfang}:$$
$$n_o = 1$$
$$\textit{Induktionsannahme}:$$
$$n_i = \sum_{j=0}^{i} 2^j = 2^{i+1} - 1$$

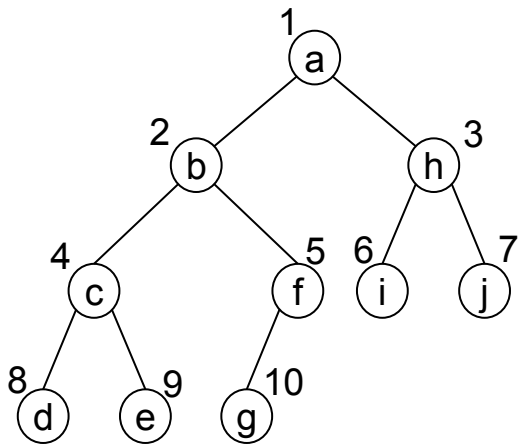
$$\textit{Induktionsschluss}:$$
$$n_{h-1} = \sum_{j=0}^{h-1} 2^j = 2^h - 1$$
$$n_h = \sum_{j=0}^{h-1} 2^j + 2^h = 2^{h+1} - 1$$
$$q.e.d.$$

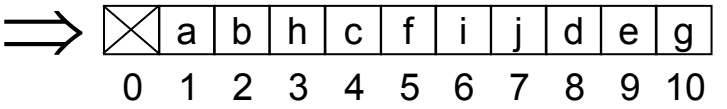
- Ein Baum der 2^h Blätter besitzt, hat eine minimale Höhe von *ld(Anzahl der Blätter)*

Darstellungsformen (1)

- Sequentielle Repräsentation
  - Knoten eines vollständigen binären Baumes werden Level-by-Level nummeriert. Die Ordnung der Knoten wird als *level-order* bezeichnet.
  - Verwende die Ordnungsnummer, um den Wert des Knoten in einem Array einzufügen.

Größe beträgt maximal 2^h-1





Wurzel	A[1]	
Linkes Kind von A[i]	A[2i]	für 2i ≤ n
Rechtes Kind von A[i]	A[2i + 1]	für 2i +1≤ n
Vater von A[i]	A[i/2]	für i > 1
A[i] ist Blatt	wahr	für 2i > n

Darstellungsformen (2)

- Verkettete Darstellung (Verwendung von Referenzen)

```
class TreeNode<T> {
     T item;
     TreeNode left;
     TreeNode right;
}
```

Traversieren eines Baumes

FHV

- Unter dem Begriff *Traversieren* wird das Besuchen (Verarbeiten) aller Knoten eines Baumes verstanden. Jeder Knoten wird dabei nur einmal besucht (verarbeitet).

- Für das Durchwandern des Baumes bestehen vier Möglichkeiten
  - Preorder

    Wurzel – linker Unterbaum – rechter Unterbaum
  - Inorder

    linker Unterbaum – Wurzel – rechter Unterbaum
  - Postorder

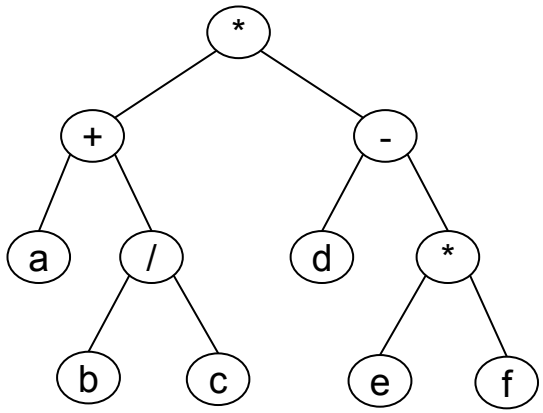
    linker Unterbaum – rechter Unterbaum – Wurzel
  - Level-order

    Von links nach rechts, Level für Level

Expression Tree

- *Expression trees* sind binäre Bäume, die zur Darstellung von arithmetischen Ausdrücken verwendet werden.

- Beispiel: (a + b / c) * (d – e * f)



Expression tree

- Annahme: Verarbeiten besteht in der Ausgabe des Wertes
  - Preorder: * + a / b c – d * e f
  - Inorder: a + b / c * d – e * f
  - Postorder: a b c / + d e f * - *

Polnische Notation

$Operator\ Operand_1\ Operand_2$

Rekursive Implementierung

- Preorder (**W**urzel-**L**inks-**R**echts)

```
preorder(TreeNode t) {
   if (t != null) {
     process(t);
     preorder (t.left);
     preorder (t.right);

   }
}
```

- Inorder (**L**inks-**W**urzel-**R**echts)

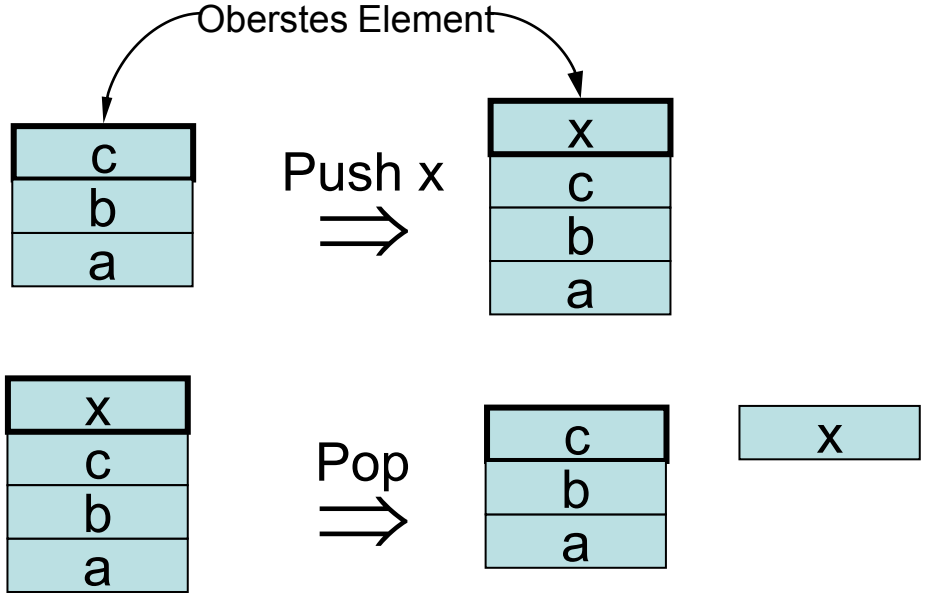
```
inorder(TreeNode t) {
   if (t != null) {
     inorder (t.left);
     process(t);
     inorder (t.right);

   }
}
```

- Postorder (**L**inks-**R**echts-**W**urzel)

```
postorder(TreeNode t) {
   if (t != null) {
     postorder(t.left);
     postorder(t.right);
     process(t);
   }
}
```

Iterative Implementierung (1)

- Vorgriff auf Datenstruktur *Stapel (Stack)*
- Ein *Stack* ist eine Liste, für die das Einfügen (Push) und Entfernen (Pop) von Elementen am gleichen Listenende stattfinden.
- Der Zugriff ist nur auf das oberste Element des Stapels möglich.



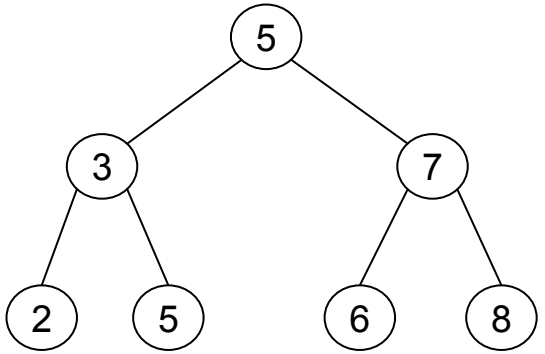
Iterative Implementierung (2)

- Verwendung eines Stacks zur Zustandssicherung
- Preorder (**W**urzel-**L**inks-**R**echts)

```
preorder(TreeNode t) {
      Stack s;
      TreeNode n;

      s.push(t); //Lege t auf den Stack
      while (!s.empty()) {
              n = s.pop()//Hole oberstes Element
              process(n);
              if (n.right != NULL){
                      s.push(n.right);
              }
              if (n.left != NULL) {
                      s.push(n.left);
              }
      }
}
```

Binäre Suchbäume

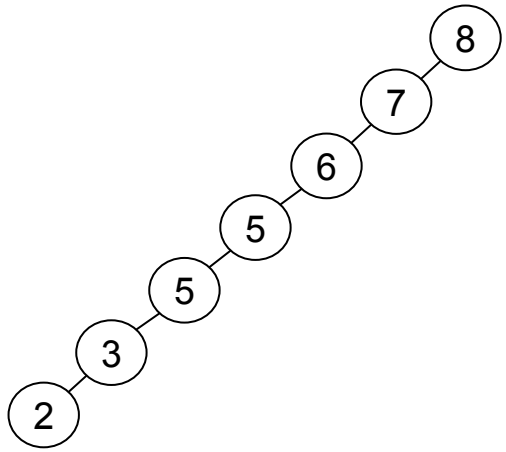
- Motivation: Ordnung der Elemente verbessert Effizienz des Zugriffs.
- Für Binäre Suchbäume gilt
  - Jedem Knoten ist ein *Schlüsselwert* zugeordnet.
  - Alle Nachfolger im *linken Unterbaum* haben *kleinere* oder *gleiche* Schlüsselwerte als der Knoten, alle Nachfolger im *rechten Unterbaum* haben *größere* Schlüsselwerte.



  - In vielen Fällen wird der Wert des Knotens als Schlüssel verwendet. Sollte dies nicht möglich sein, wird die *TreeNode* Struktur um ein *Key*-Attribut erweitert.

Eigenschaften

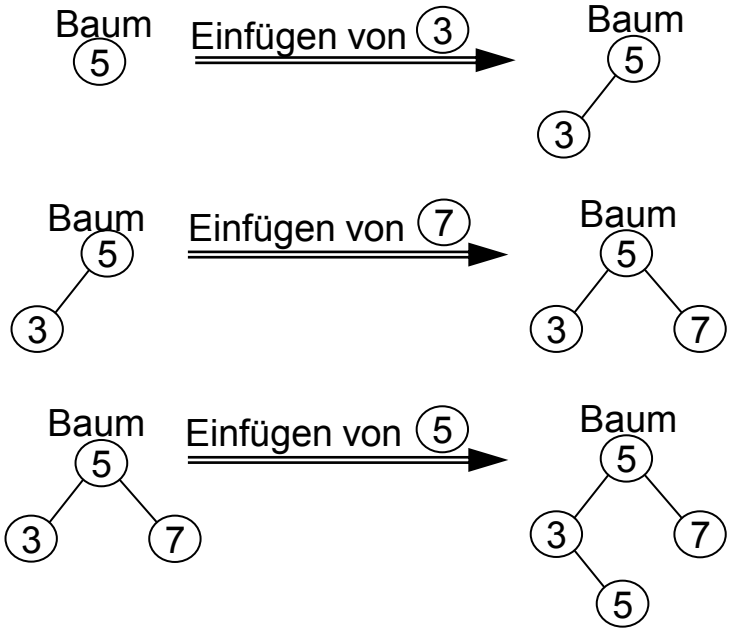
- Verwaltung von beliebig großen Datenmengen
- Laufzeit der Basis-Operationen ist proportional zu der *Höhe* des Baumes, O(h)
- Worst case ist O(log n), wobei n die Anzahl der Knoten ist.
- Worst case Bei Entartung (Liste) O(n).



Entartung, da Knoten nach den Schlüsselwerten sortiert eingefügt wurden.

Einfügen eines Knotens (1)

- Knoten werden sortiert eingefügt:
  - Alle Nachfolger eines Knotens K im linken Unterbaum haben einen Schlüssel der kleiner oder gleich als der Schlüssel von K ist.
  - Alle Nachfolger eines Knotens K im rechten Unterbaum haben einen Schlüssel der größer als der Schlüssel von K ist.



Einfügen eins Knotens (2)

- Iterative Implementierung

```
void insert(TreeNode t,
       TreeNode n) {
  TreeNode parent =  t;

  while (t != null) {
     parent = t;
     if (t.key >= n.key) {
        t = t.left;
     } else {
        t = t.right;
     }
  } // end while

  if(parent.key >=n.key) {
     parent.left = n;
  } else {
     parent.right = n;
  }
} // end insert
```

Suchen eines Knotens

- Gesucht ein Knoten mit einem bestimmten Schlüsselwert
- Rekursive Lösung

```
TreeNode search(TreeNode t, Key k) {
    if (t == null || k == t.key) {
        return t;
    }
    if (t.key >= k)
        return search(t.left, k);
    else
        return search(t.right, k);
}
```

- Iterative Lösung

```
TreeNode search(TreeNode t, Key k) {
    while (t != null && k != t.key) {
        if (t.key >= k)
           t = t.left;
        else
           t = t.right;
    }
    return t;
}
```

- Bei der Suche wird genau ein Pfad bewandert ⇒ Laufzeit O(h), für h = Höhe des Baumes

Suchen des Min / Max

- Folgt man in einem Binären Suchbaum den Zeigern auf die linken Kinder bis man auf NULL trifft, findet man das Minimum

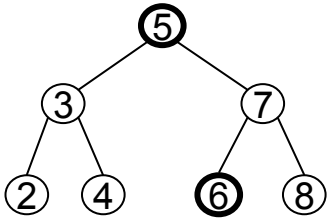
```
TreeNode min(TreeNode t) {
      while (t.left != null) {
         t = t.left;
      }
      return t;
 }
```

- Folgt man den Zeigern auf die rechten Kinder erhält man das Maximum

```
TreeNode max(TreeNode t) {
      while (t.right != null) {
         t = t.right;
      }
      return t;
 }
```

Nachfolger (Successor) (1)

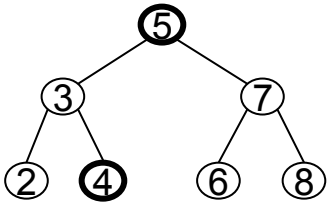
- Successor eines Knoten *K* ist jener Knoten mit dem nächst größeren Schlüssel als *K*.
- In einem binären Baum, in dem **keine Knoten mit gleichen Schlüsseln** vorkommen, kann der Successor eines Knotens ohne Schlüsselvergleich festgestellt werden.
- Bei der Suche müssen zwei Fälle unterschieden werden
  - Der Knoten K hat einen nicht leeren, rechten Unterbaum. Successor ist das Minimum des rechten Unterbaums



Successor (⑤) = ⑥

⑥ist das *Minimum* des rechten Unterbaums von ⑤

  - Der rechte Unterbaum ist leer. Successor ist jener „kleinste“ Vorfahre von K, dessen linkes Kind Vorfahre von K ist.



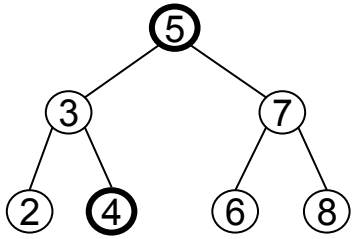
Successor (④) = ⑤

Nachfolger (Successor) (2)

```
TreeNode successor(TreeNode t, int key) {
   TreeNode succ = null;
   while (t != null) {
      if (t.key == key) {
         if (t.right == null) {
            return succ;
         }
         return min(t.right);
      }

      if (t.key < key) {
         t = t.right;
      }
      else {
         succ = t;
         t = t.left;
      }
   }

   return succ;
}
```

Vorgänger (Predecessor)

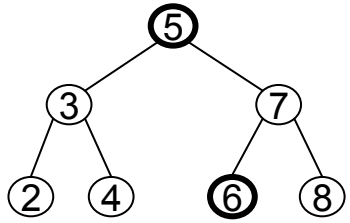
- Predecessor eines Knoten *K* ist jener Knoten mit dem nächst kleineren Schlüssel als *K*.
- In einem binären Baum, in dem **keine Knoten mit gleichen Schlüsseln** vorkommen, kann der Predecessor eines Knotens ohne Schlüsselvergleich festgestellt werden.
- Bei der Suche müssen zwei Fälle unterschieden werden
  - Der Knoten K hat einen nicht leeren, linken Unterbaum. Predecessor ist das Maximum dieses Unterbaums



Predecessor (⑤) = ④

④ ist das *Maxmum* des linken Unterbaums von ⑤

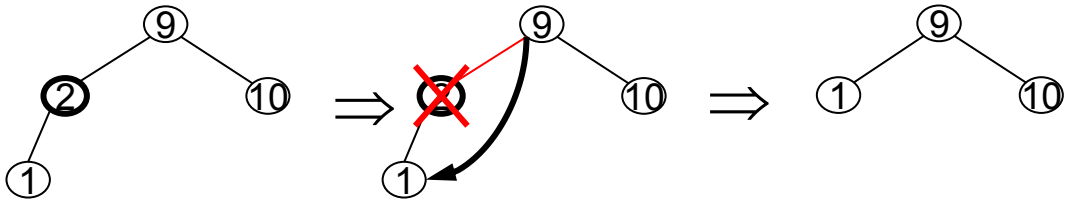
  - Der linke Unterbaum ist leer. Predecessor ist jener „größte“ Vorfahre von K, dessen rechtes Kind Vorfahre von K ist.



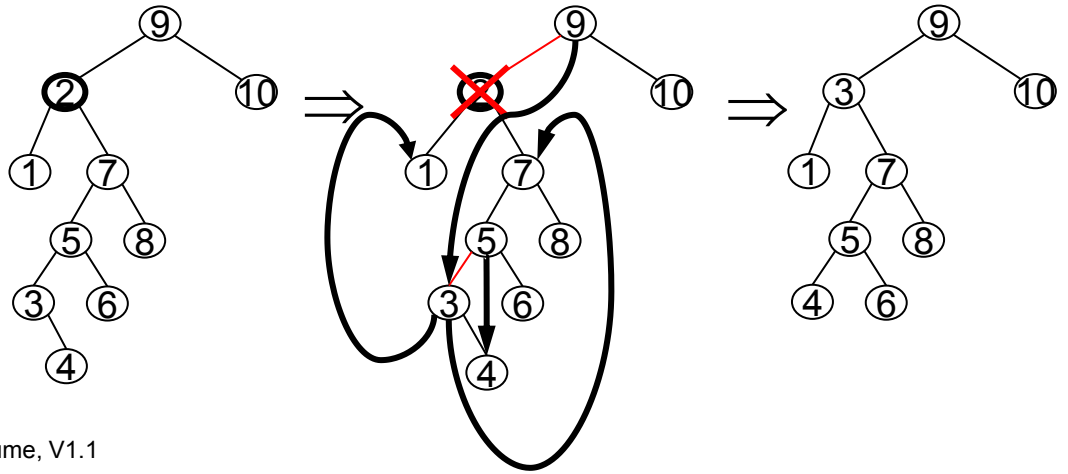
Predecessor (⑥) = ⑤

Löschen eines Knotens

- Beim Löschen eines Knotens K müssen folgende Fälle unterschieden werden, um wieder einen Binären Suchbaum zu erhalten
  - K hat nur ein Kind ⇒ Ersetzte K durch das Kind



  - K hat zwei Kinder: Ersetze K durch den Successor



Bewertung Binärer Suchbäume

FHV

Vorteile

- Gute Modellierung von hierarchischen Strukturen
- Laufzeit der Operationen ist von der Höhe abhängig, nicht von der Anzahl der Knoten
- Einfache Suche
- Einfaches Ermitteln von Minimum und Maximum
- Durchlauf in *Pre-, In-* oder *Post-Order* sehr einfach (Sortierung)

Nachteile

- Bei sortiertem Einfügen von Knoten entartet der Baum => Liste
- Kein direkter Zugriff auf die Knoten