Bäume

Bäume, Binärbäume, Traversierungen, abstrakte Klassen, Binäre Suchbäume, Balancierte Bäume, AVL-Bäume, Heaps, Heapsort, Priority queues

Behälter mit Standardreihenfolgen

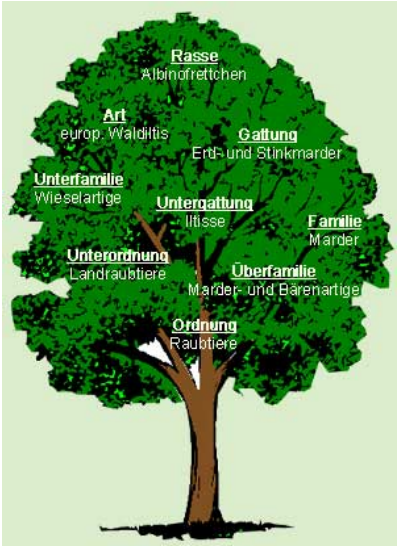
- Listen
  - Standardreihenfolge
  - veränderbare Größe
  - kein schneller Zugriff auf die Elemente

- Bäume
  - Hierarchie und meist Standard-Reihenfolge der Söhne

  - Elemente
    - in Knoten oder in Blättern oder in Knoten und in Blättern

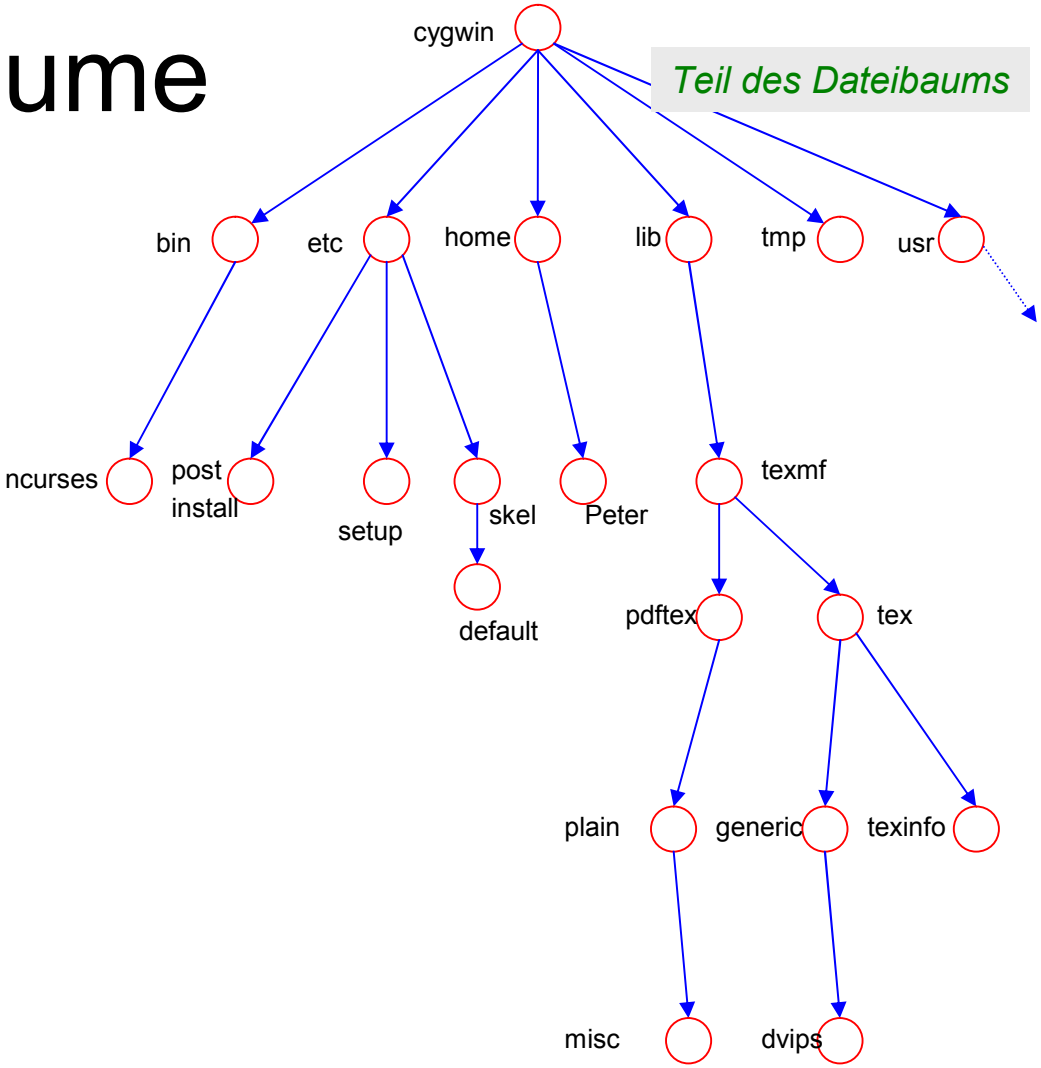
  - Reihenfolge der Söhne gibt Anlass zu Standardreihenfolgen
    - pre-order, in-order, postorder

  - Hierarchie gibt Anlass zu Standardreihenfolge
    - depth-first und breadth-first



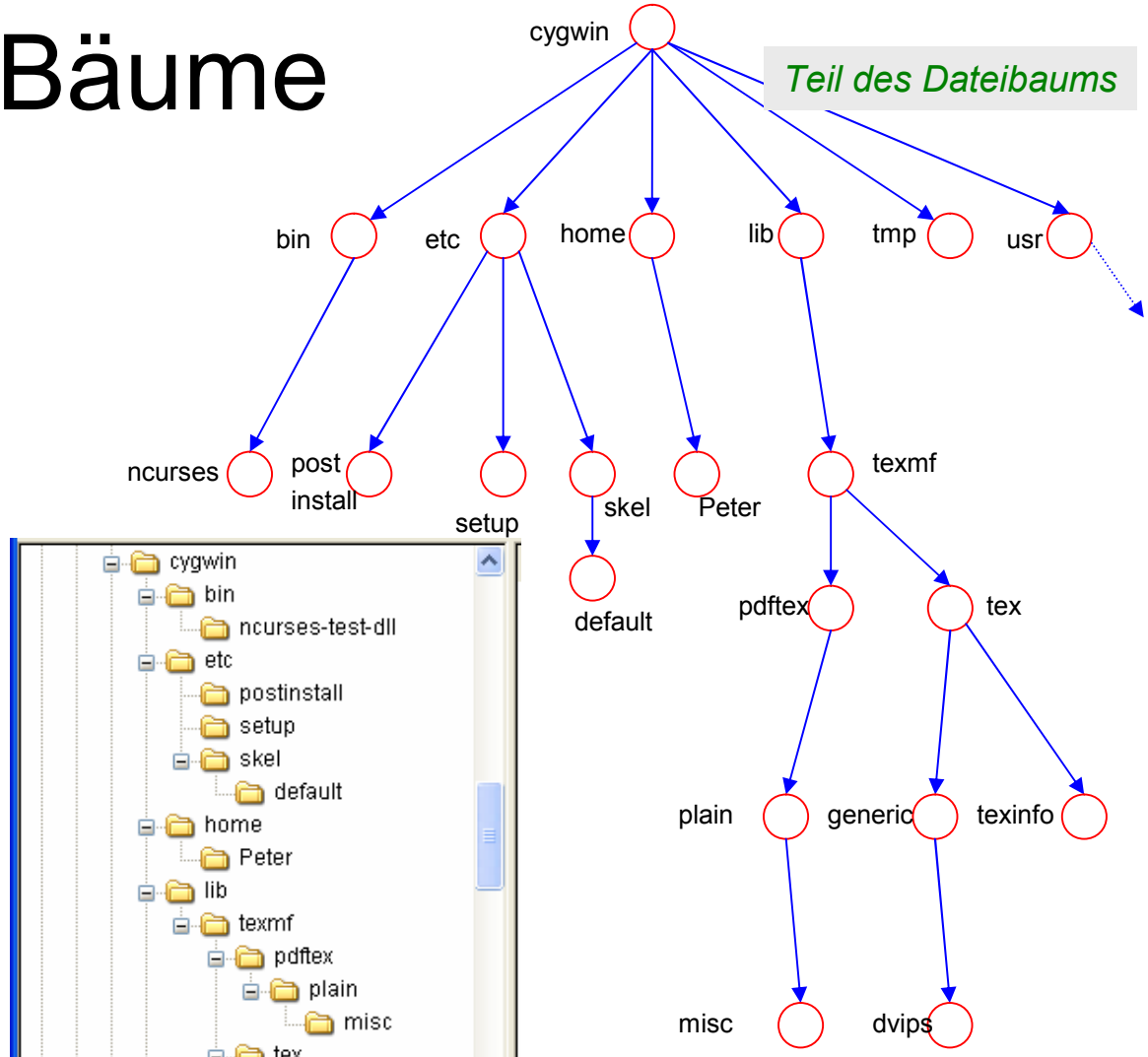
Hierarchien - Bäume

- Bäume : hierarchische Strukturen
  - Knoten
    - Objekte/Mitglieder der Hierarchie
  - Kanten :
    - verbinden Objekte mit übergeordnetem (Vater)
  - Wurzel:
    - oberster Knoten der Hierarchie
    - kein Vorgänger (Vater)
  - Blätter:
    - Knoten ohne Nachfolger (Sohn)



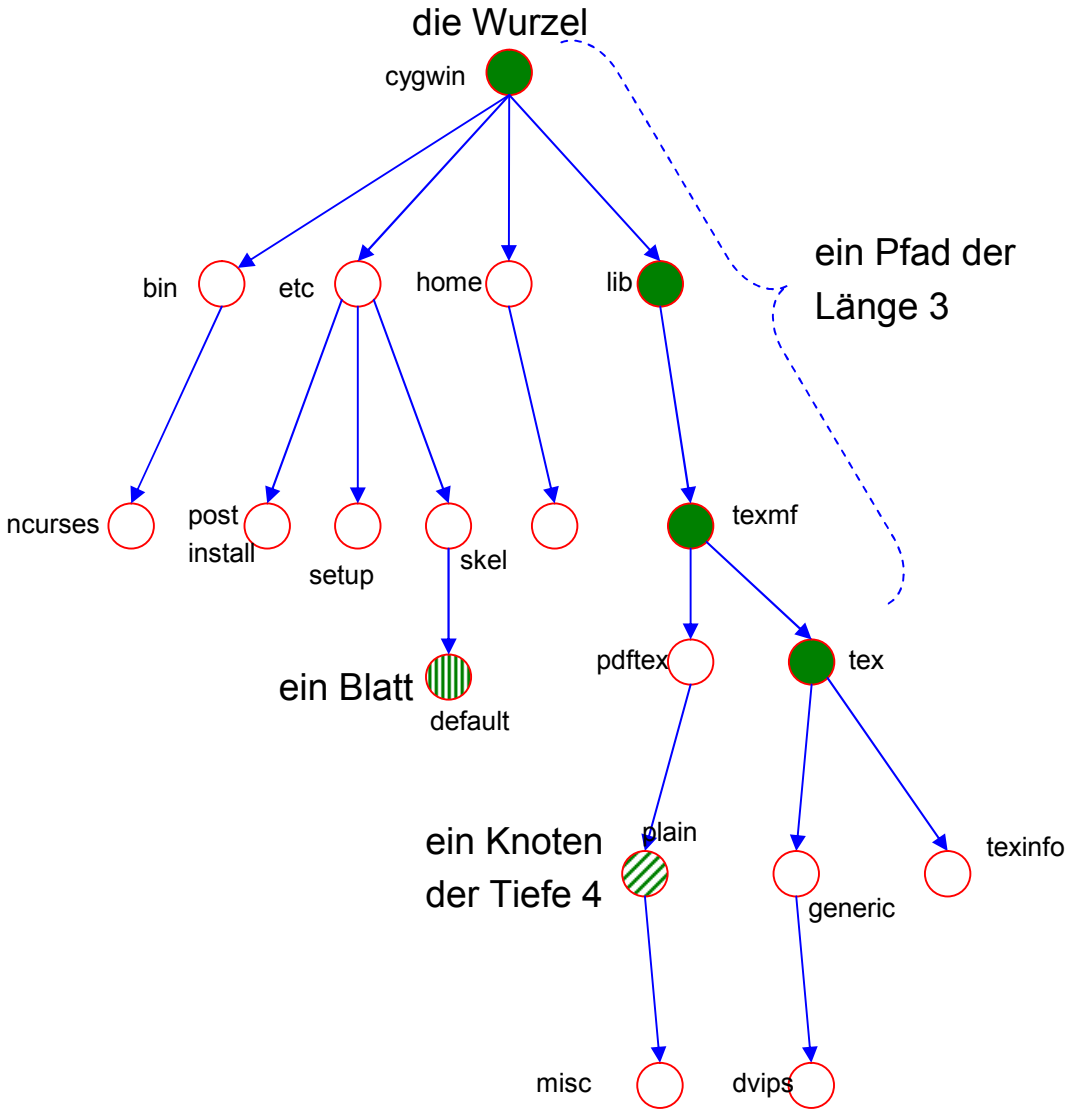
Hierarchien - Bäume

- Bäume : hierarchische Strukturen
  - Knoten
    - Objekte/Mitglieder der Hierarchie
  - Kanten :
    - verbinden Objekte mit übergeordnetem (Vater)
    -
  - Wurzel:
    - oberster Knoten der Hierarchie
    - kein Vorgänger (Vater)
  - Blätter:
    - Knoten ohne Nachfolger (Sohn)
- Beispiel: Dateisystem
  - Knoten :
    - Verzeichnisse
  - Kanten:
    - Unterverzeichnis
  - Blätter
    - Dateien
  - Wurzel:
    - Laufwerk (C:)



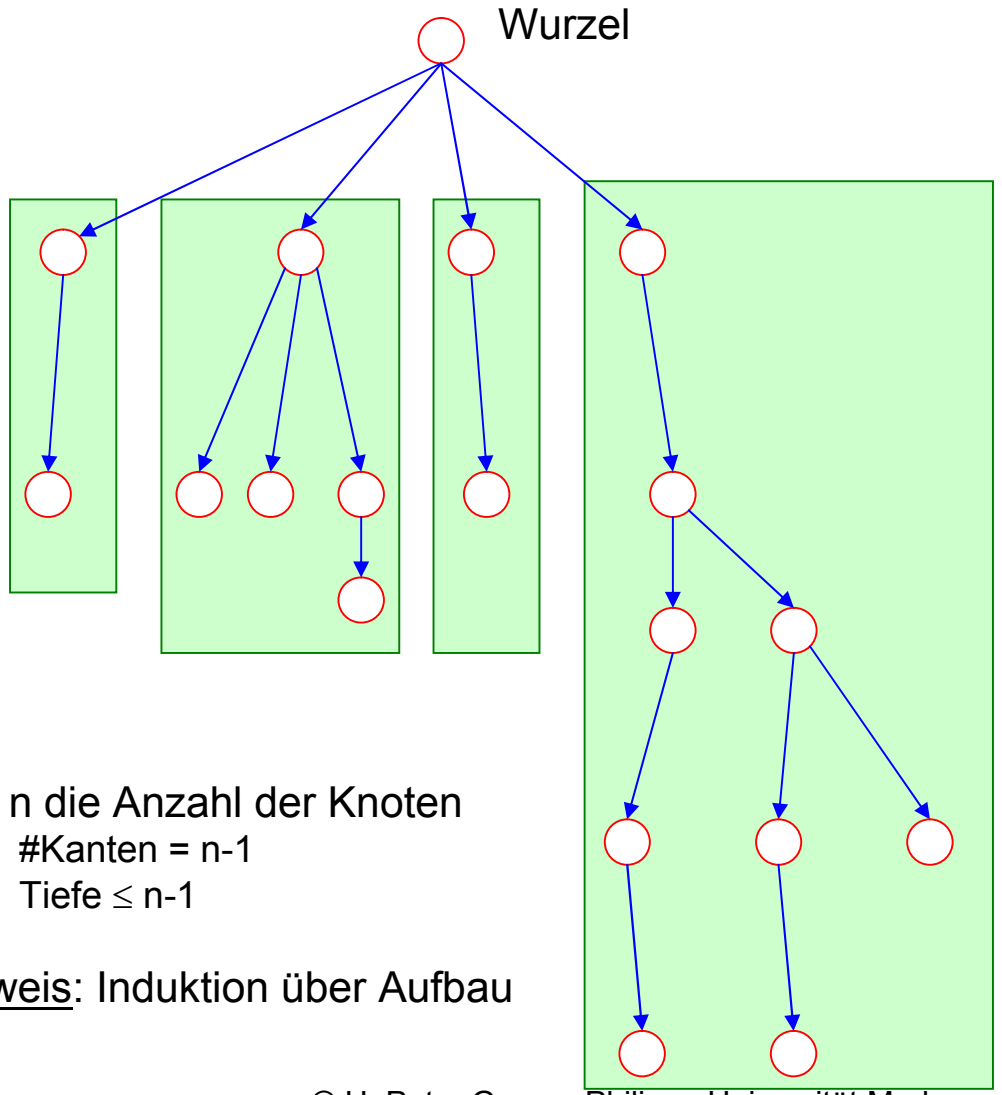
Definitionen

- *Pfad*:
  - Knotenfolge von der Wurzel zu einem Knoten
  - Beispiel: cygwin/lib/texmf/tex
- *Länge* eines Pfades:
  - #Knoten-1 = #Kanten
- *Tiefe eines Knoten:*
  - Länge des Pfades zur Wurzel
- *Tiefe des Baumes:*
  - leerer Baum hat Tiefe -1.
  - sonst :
    - max. Tiefe eines Knoten
    - = max. Länge eines Pfades



Bäume – induktiv definiert

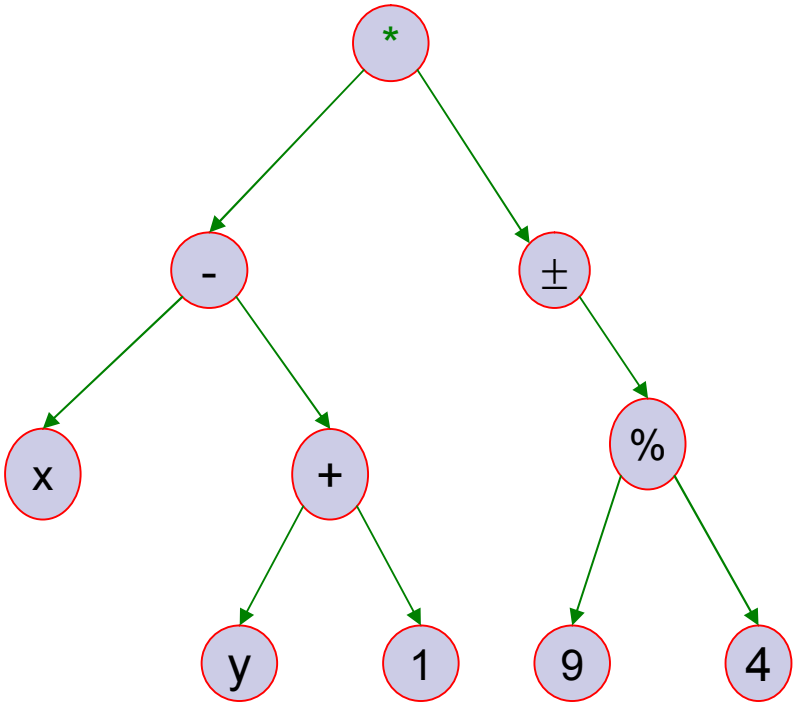
- Ein (nichtleerer) Baum besteht aus
  - der Wurzel
  - den Unterbäumen der Wurzel
- Von den Wurzeln dieser Unterbäume geht genau eine Kante zur Wurzel des Baumes



- Sei n die Anzahl der Knoten
  - #Kanten = n-1
  - Tiefe ≤ n-1
- Beweis: Induktion über Aufbau

Operatorbaum

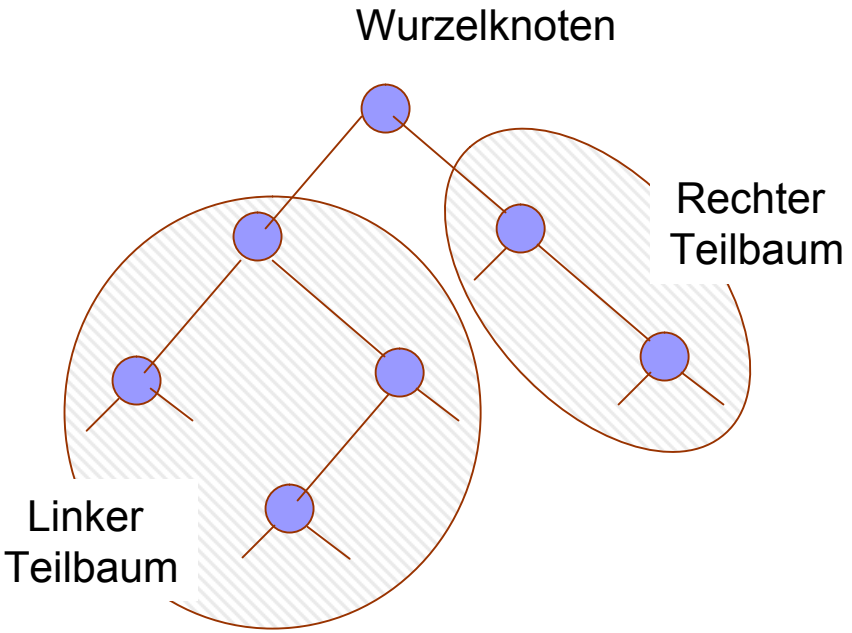
- repräsentiert Ausdruck (Expression)
- Knoten: Operator
  - n-stelliger Operator hat n Söhne
- Argumente: Söhne
- Blätter: Konstanten oder Variablen
- Reihenfolge der Söhne relevant
  - falls Operator nicht kommutativ
- Jeder Knoten repräsentiert einen Wert
  - Blatt:
    - Konstante, bzw. gespeicherter Wert
  - Operatorknoten
    - Nimm die Werte, die den Söhnen zugeordnet sind
    - Wende die Operation an



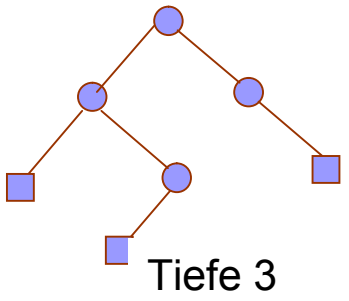
Operatorbaum für
(x - (y+1)) * (- (9 % 4))

BinärBäume

- Bei *Binärbäumen* hat jeder Knoten zwei Unterbäume,
  - den linken Teilbaum
  - den rechten Teilbaum
- ein Binärbaum darf leer sein
- In den Knoten kann Information gespeichert werden
- Ein Blatt in einem Binärbaum ist ein Knoten,dessen beide Söhne leer sind.



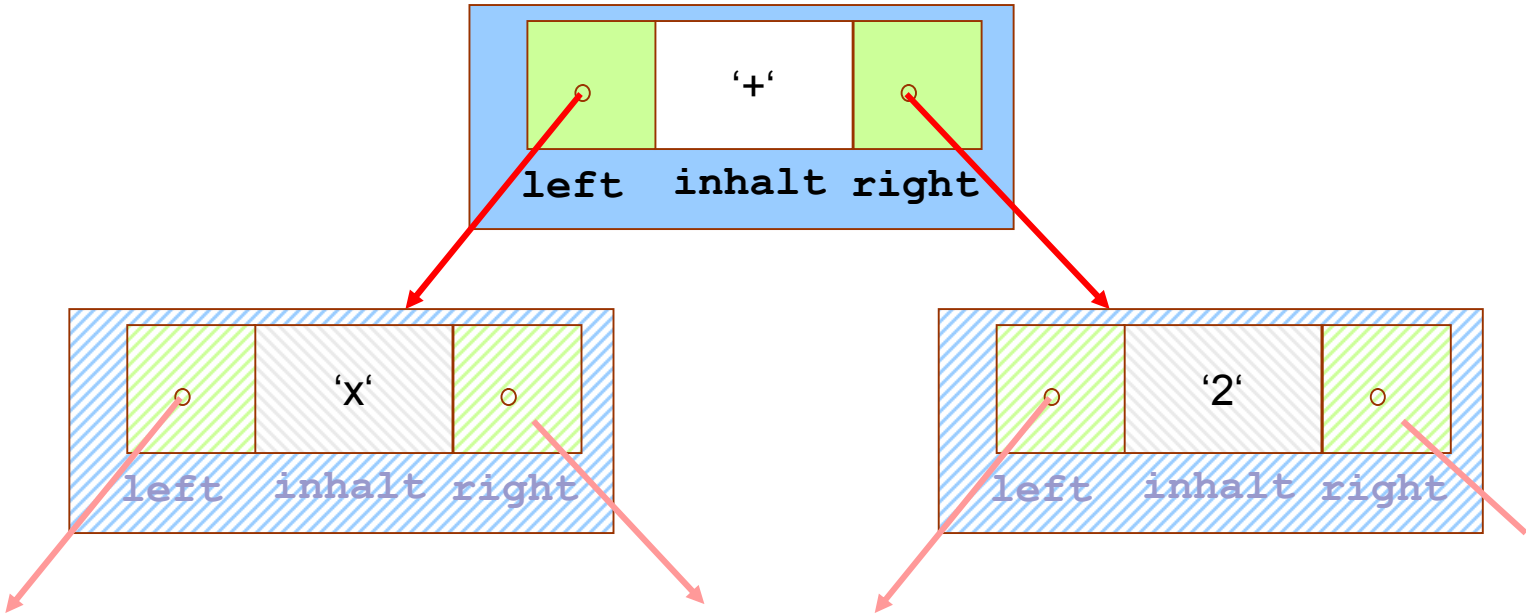
- Beliebte Konvention für Darstellung
  - leere Unterbäume nicht zeichnen
  - innere Knoten rund
  - Blätter rechteckig



Baumknoten

```
class Knoten<E>{
    Knoten left;
    E inhalt;
    Knoten right;
}
```

- Wir implementieren einen Knoten als Zelle mit zwei Zeigern.
- Zur Abwechslung speichern wir Zeichen in den Knoten



Indexkarte für BinTree

- Beliebige BinTree-Operationen können sich auf diesen Methoden abstützen:

BinTree<E>

«Konstructoren»
+ **BinTree**()
+ **BinTree**(BinTree<E>, E, BinTree<E>)

«Prädikate»
+ boolean **isEmpty**( )

«Selektoren»
+ BinTree **left**( )
+ BinTree **right**( )
+ E **content**( )

- Anderen Felder und Methoden werden private erklärt

lichtung

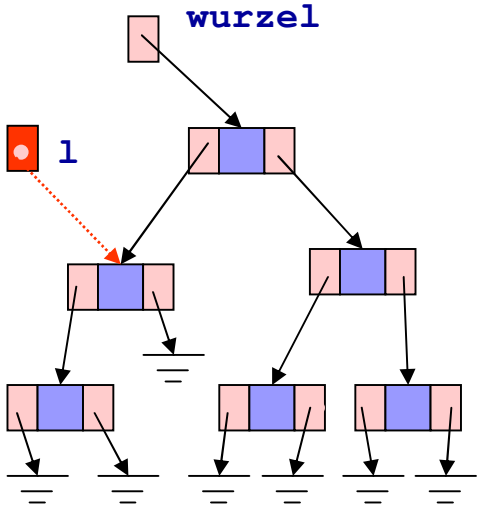
manche meinen
lechts und rinks
kann man nicht velwechsern
werch ein illtum

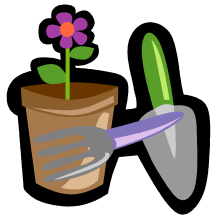
(Ernst Jandl)

Implementierung als Ergebnistyp

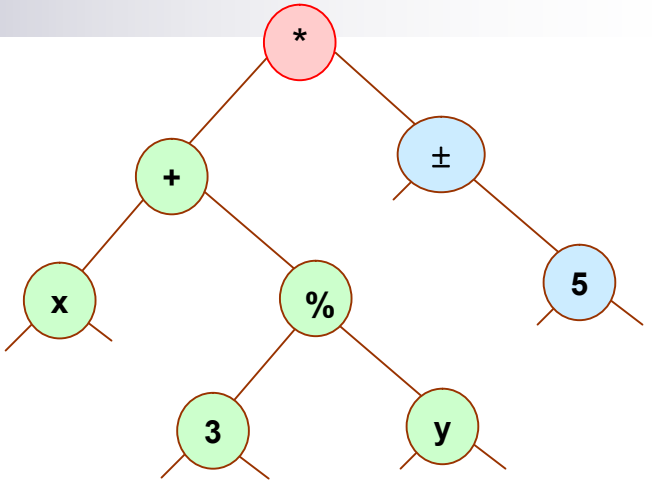
```
class BinTree<E>{
    private Knoten wurzel;

    // Konstruktoren
        BinTree(){};    // der leere Baum

        BinTree(BinTree<E> b1, E c, BinTree<E> b2){
        wurzel  = new Knoten(b1.wurzel,c,b2.wurzel);
        }
    // Prädikat
    boolean isEmpty(){ return wurzel==null; }

    // Selektoren
    BinTree<E> left(){
        BinTree l = new BinTree();
        l.wurzel = this.wurzel.left;   //this nicht nötig
        return l;
    }

    E content(){ return this.wurzel.content; }
```

- Implementierung verläuft analog zu Listen

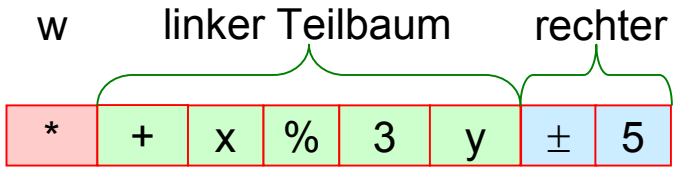


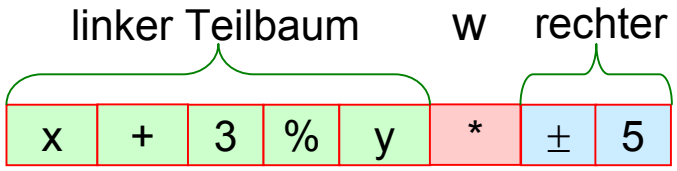


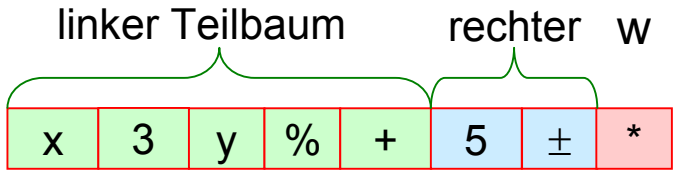
Traversierung



- Systematisches Durchlaufen aller Knoten eines Baumes
  - Preorder:
    - erst die Wurzel (w)
    - dann linker Teilbaum (in Preorder)
    - dann rechter Teilbaum (in Preorder)
  - Inorder:
    - erst linker Teilbaum (in Inorder)
    - dann die Wurzel (w)
    - dann rechter Teilbaum (in Inorder)
  - Postorder
    - erst linker Teilbaum (in Postorder)
    - dann rechter Teilbaum (in Postorder)
    - dann die Wurzel (w)







Traversierungen

```
// Klasse BinTree<E> delegiert in innere Klasse Knoten
  public void preOrder()   { if(!isEmpty()) wurzel.preOrder(); }
  public void inOrder()    { if(!isEmpty()) wurzel.inOrder(); }
  public void postOrder() { if(!isEmpty()) wurzel.postOrder(); }
```

BinTree<E>

- Klasse BinTree<E> behandelt leeren Baum und delegiert an innere Klasse Knoten
- Auf Knotenebene einfache Rekursionen

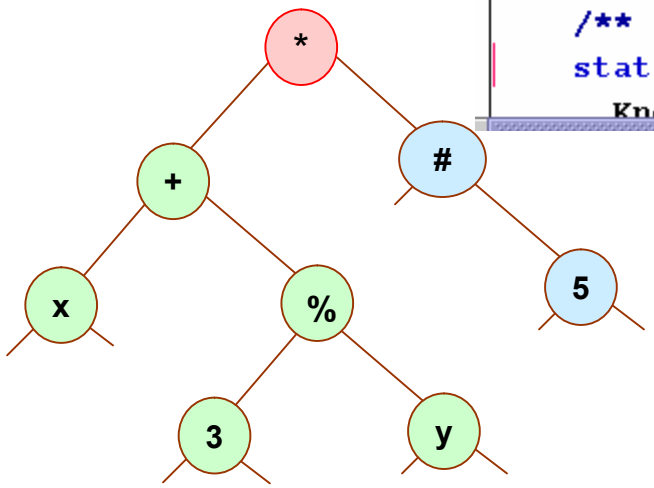
```
// In der Klasse Knoten:
    public void preOrder(){
        tuWas(inhalt);
        if(links!=null)  links.preOrder();
        if(rechts!=null) rechts.preOrder();
    }

    public void inOrder(){
        if(links!=null)  links.inOrder();
        tuWas(inhalt);
        if(rechts!=null) rechts.inOrder();
    }

    public void postOrder(){
        if(links!=null)  links.postOrder();
        if(rechts!=null) rechts.postOrder();
        tuWas(inhalt);
    }

    void tuWas(E e){ System.out.println(e); }
```

BinTree<E>.Knoten

Die äußere Klasse BinTree

- Traversierungen objektorientiert
- Leerer *BinTree* ist nicht *null*, sondern der BinTree mit *wurzel==null*

```
class BinTree{
    Knoten wurzel;
    // Konstruktoren
    BinTree(){ wurzel=null; };              // der leere Baum
    BinTree(Knoten k){ wurzel = k;}         // nichtleerer Baum
    BinTree(BinTree B1,Object o, BinTree B2)
    {wurzel=new Knoten(B1.wurzel,o,B2.wurzel);}

    // Objektorientierte Traversierungen - auch für leeren Baum
    String prefix(){return Knoten.prefix(wurzel);}
    String infix(){return Knoten.infix(wurzel);}
    String postfix(){return Knoten.postfix(wurzel);}

    /** Statische Innere Klasse "BinTree.Knoten". */
    static class Knoten{
      Knoten
```

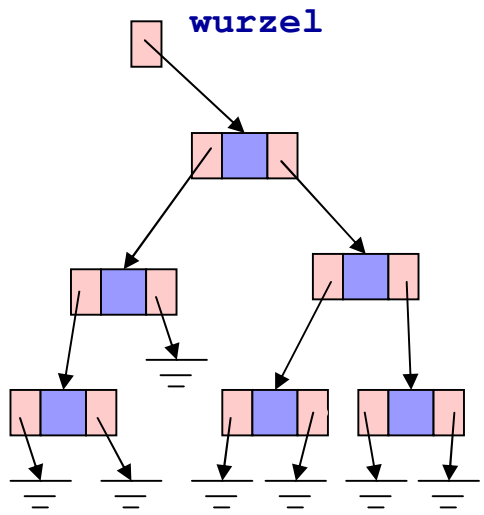


BlueJ: Terminal Window

Options

```
Preorder :  *  +  x  %  3  y  #  5
Inorder : ((( x ) + (( 3 ) % ( y ))) * ( # ( 5 )))
Postorder :  x  3  y  %  +  5  #  *
```

Immer das gleiche Schema:



In Klasse BinTree<E>

```
// Klasse BinTree<E> delegiert in innere Klasse Knoten
  public int  tiefe(){
     if (isEmpty()) return -1; else return wurzel.tiefe(); }

     public boolean contains(E e)
     { return !isEmpty() && wurzel.contains(e);     }
```

In Klasse BinTree<E>.Knoten

- Behandle leeren Baum
- Delegiere an innere Klasse Knoten
- Löse dort rekursiv

Vorsicht:

- null ist kein Objekt, daher Delegation an null abfangen
- NullPointerException

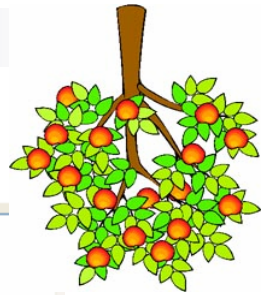
```
    // Tiefe eines Knotens
    public int tiefe(){
        if(links==null && rechts==null) return 0;
        if(links==null)  return 1+rechts.tiefe();
        if(rechts==null) return 1+links.tiefe();
        return 1+Math.max(links.tiefe(),rechts.tiefe());
    }
    // contains
    public boolean contains(E e)
       { return e.equals(inhalt)
                || (links  != null && links.contains(e))
                || (rechts != null && rechts.contains(e));
       }
```

Alternative – nicht objektorientiert

- Traversierungen *static*, damit sie auch für *null* funktionieren
- Innere Klassen mit statischen Methoden müssen selber statisch sein.
- null ist kein Objekt der Klasse Knoten
  - kann keine Methode empfangen
  - darf aber als Parameter auftauchen
- Operatorbaum benötigt keine Klammern für
  - prefix
  - postfix
- Klammern nötig für
  - infix

```
/** Statische Innere Klasse "BinTree.Knoten". */
static class Knoten{
  Knoten left, right;
  Object content;

  Knoten(Knoten l, Object c, Knoten r)
  { left = l; content = c; right =r; }

  static String prefix(Knoten k)
  { return (k==null) ? "" :
      " "+k.content.toString()+" "+prefix(k.left)+prefix(k.right); }

  /* Infix benötigt Klammern */
  static String infix(Knoten k)
  { return (k==null) ? "" :
      "("+infix(k.left)+" "+k.content.toString()+" "+infix(k.right)+")";}

  static String postfix(Knoten k)
  { return (k==null) ? "" :
      postfix(k.left)+postfix(k.right)+" "+k.content.toString()+" "; }

}// Ende innere Klasse Knoten
```

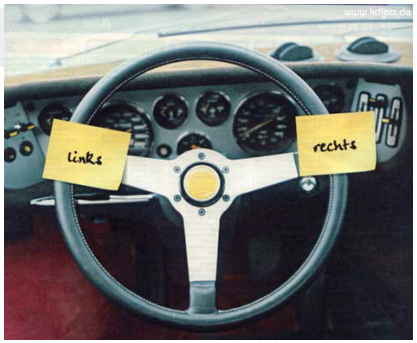
Bäume als Behälter



```
import java.util.Iterator;
import java.util.Stack;
public class TreeContainer<E> extends BinTree<E> implements Iterable<E>{

  public Iterator iterator(){ return new PreOrderIterator();  }
```

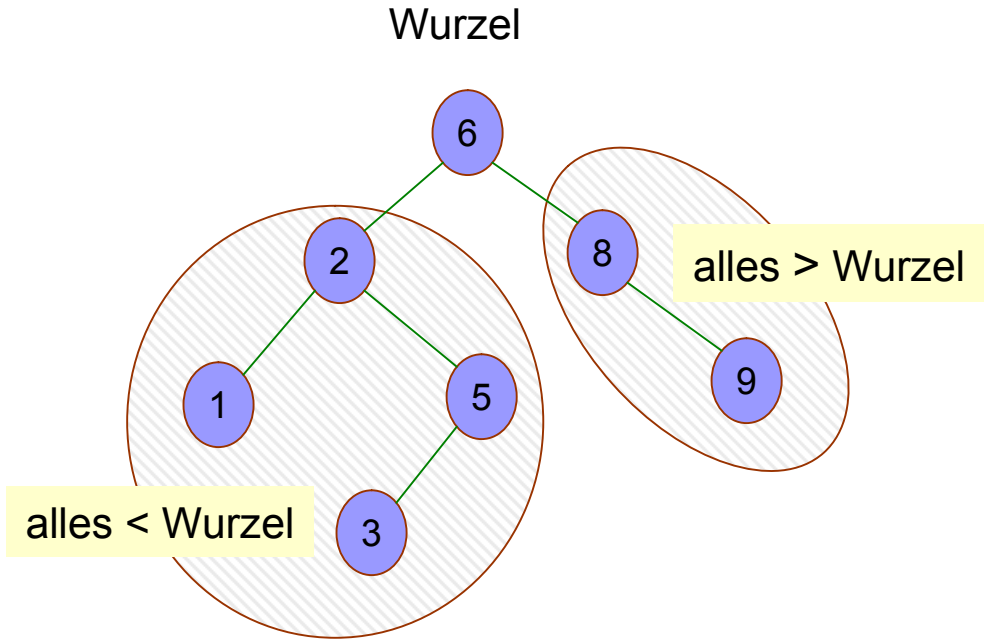
- Durchlaufreihenfolge gegeben durch Iteratoren
- Iteratoren verwenden Standard-Behälter :
  - **Stack**:
    - für preOrder, inOrder, postOrder
  - **Queue**:
    - für levelOrder (schichtweise)
- Beispiel: Inorder
  - Wurzel auf Stack
  - **while** (Stack nicht leer)
    Pop Knoten k
    Push(k.rechts)
    Push(k.links)
    tuWas(k.inhalt)

```
public class PreOrderIterator implements Iterator{
   Stack<Knoten> kStack;

   PreOrderIterator(){
       kStack = new Stack();
       if(wurzel != null)kStack.push(wurzel);
      }

   public void remove(){
        throw new UnsupportedOperationException(); }

   public E next(){
        Knoten top = kStack.pop();
        if(top.rechts != null) kStack.push(top.rechts);
        if(top.links  != null) kStack.push(top.links);
        return top.inhalt;
      }
   public boolean hasNext(){ return !kStack.isEmpty();}
  }
```



Binäre Suchbäume

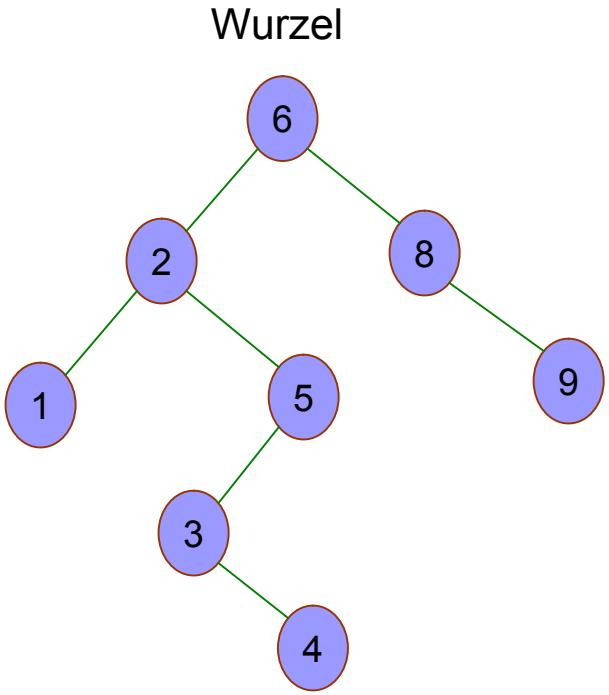
- Binärbäume mit Information in Knoten (und evtl. in Blättern)
- Daten tragen eine Ordnung
  - z.B.: *Comparable*
- Invariante
  - keine Duplikate)*
  - alle Elemente in linkem Teilbaum < Wurzel
  - alle Elemente in rechtem Teilbaum > Wurzel
- Konsequenz:
  - binäre Suche möglich
  - Inorder Traversierung: Daten in geordneter Reihenfolge
  - ⇒ TreeSort



)* wenn man Duplikate zulässt, wird *remove* Operation komplizierter

Suchen im Binären Suchbaum

- suche(Baum b, Element e):
  - falls e = wurzel(b) : gefunden !
  - falls e < wurzel(b) : suche(left(b),e)
  - falls e > wurzel(b) : suche(right(b),e)



- einfügen(Baum b, Element e):
  - falls b=leer:
    - Neuer Baum mit Wurzel e
  - falls e = wurzel(b) : tue nichts
  - falls e < wurzel(b) :
    - falls left(b) leer :
      - Neuer linker Teilbaum mit Wurzel e
    - sonst:
      - einfügen(left(b),e)
  - falls e > wurzel(b) :
    - analog

Implementierung *BSTree*

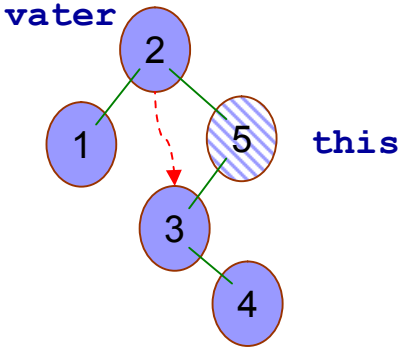
- *wurzel* ist null oder ein Knoten – d.h. ein nichtleerer Baum
- *Daten:* Beliebige Klasse *E*, die *Comparable implementiert*
- *Knoten* repräsentiert nichtleeren Baum
- Die Methoden
  - *insert*
  - *search*
  - *remove*

etc. sind in *Knoten* und in *BSTree* implementiert.

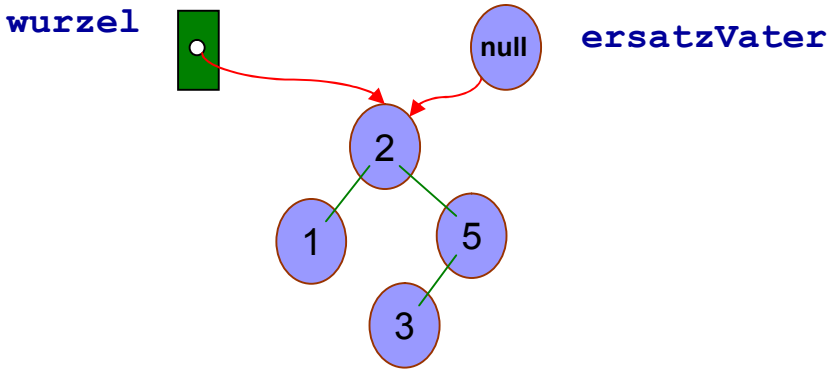
```
public class BSTree<E extends Comparable> {
  Knoten wurzel;

  public void insert(E e){
    if (wurzel==null){
        wurzel=new Knoten(null,e,null);
    }else wurzel.insert(e); //delegiere in Knoten-Klasse
  }// Ende von BSTree.insert

//======= Innere Knotenklasse ==========
  private class Knoten{

 //Objektfelder
    Knoten links;
    E inhalt;
    Knoten rechts;

    /* Konstruktor */
    Knoten(E e){ this(null,e,null); }
    Knoten(Knoten l, E e, Knoten r){
       links = l; inhalt = e; rechts =r;
    }

    public void insert(E e){
      int v = inhalt.compareTo(e);
      if(v > 0)  // e < inhalt
           if (links == null) links = new Knoten(e);
           else links.insert(e);
      else if (rechts == null) rechts = new Knoten(e);
           else rechts.insert(e);
      }// Ende of Knoten.insert
```

Knoten löschen – in *BSTree*

- Beim Löschen eines Knoten muss man dessen Vater in der Hand haben
  - die Verweise des Vaters müssen geändert werden
- Technisch:
  - Vater wird als Parameter mitgeführt
  - *this.remove(e,vater)*



- Wurzel hat aber keinen Vater
  - Wir bauen kurzfristig einen *Ersatzvater*
  - delegieren *remove* in Knotenklasse
  - zum Schluss erhält *wurzel* den linken Sohn des Ersatzvaters (*Warum ist das nötig ?*)




```
public void remove(E e){
   if(contains(e)){
        Knoten ersatzVater =
              new Knoten(wurzel,null,null);
        wurzel.remove(e,ersatzVater);
        wurzel=ersatzVater.links;
       }
```

löschen in *Knoten*

- e mit Knoteninhalt vergleichen
- kleiner:
  - rekursiv in linken Teilbaum
- grösser:
  - rekursiv in rechten Teilbaum
  - this wird neuer Vater
- gleich:
  - nur ein Sohn:
  - verbinde Vater mit Enkel
  - zwei Söhne:

    ersetze inhalt durch kleinstes Element im rechten Teilbaum

    lösche dieses

in Klasse *Knoten*

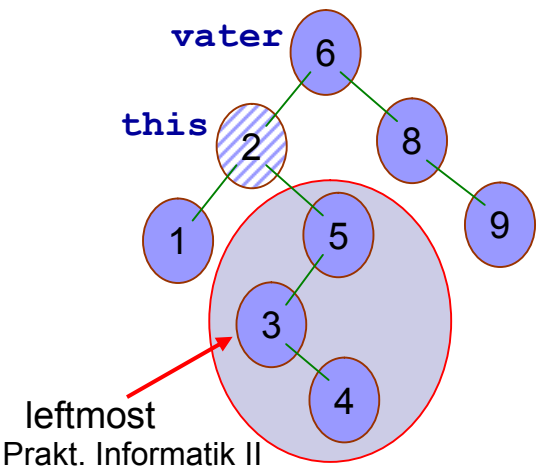
```
void remove(E e, Knoten vater){
   /* Pre */ assert vater.istVaterVon(this);
   int v = e.compareTo(inhalt);

   if      (v < 0) links.remove(e,this);
   else if (v > 0) rechts.remove(e,this);
   else{
      /* Zw.behauptung */ assert this.inhalt.equals(e);
      if     (links==null)  connect(vater,rechts);
      else if(rechts==null) connect(vater,links);
      else{ // min:      Minimum im rechten Teilbaum
            // minVater: Vater von min
            Knoten min = rechts;
            Knoten minVater = this;
            while(min.links != null){
                /* Inv.:*/ assert minVater.istVaterVon(min);
                minVater = min;
                min = min.links;
            } // hochkopieren:
            inhalt = min.inhalt;
              // min löschen
            min.connect(minVater,min.rechts);
          }
      }
   }// end of remove

/** Verbindet vater von this mit Sohn von this */
private void connect(Knoten vater, Knoten enkel){
  if(this == vater.links) vater.links = enkel;
  else vater.rechts = enkel;
}
```

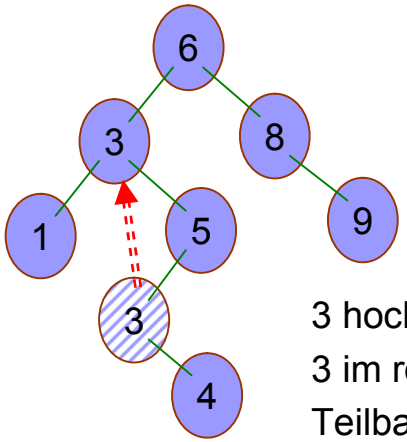
Knoten löschen

- 2. Fall: Knoten **this** hat zwei Söhne:
  - Kopiere kleinstes Element des rechten Teilbaums in k
  - Lösche das Element aus dem rechten Teilbaum

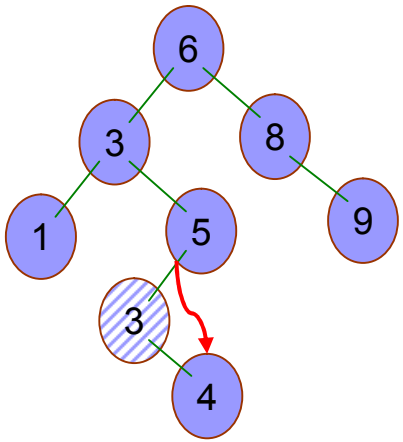
```
public void deleteBelow(Knoten vater, int n){
  if (n < inhalt && links != null)
        links.deleteBelow(this,n);
  else if (n > inhalt && rechts != null)
        rechts.deleteBelow(this,n);
  else if (n==inhalt){
    if (links==null){
        if(this==vater.links) vater.links=rechts;
        else vater.rechts=rechts;
    }else if (rechts==null){
        if(this==vater.links) vater.links=links;
        else vater.rechts=links;
    }else{
        inhalt=leftmost(rechts);
        rechts.deleteBelow(this,inhalt);
    }
  }
}
```

Bsp.: Um die 2 zu löschen :





3 hochkopieren
3 im rechten
Teilbaum löschen

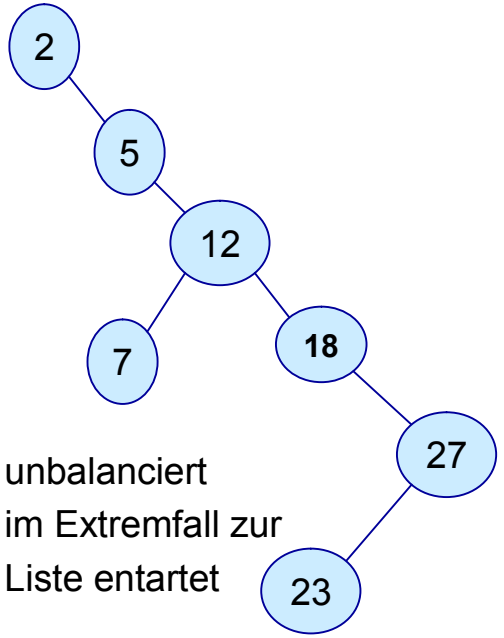


Balance

- Suchbaum nach Einfügen der Elemente

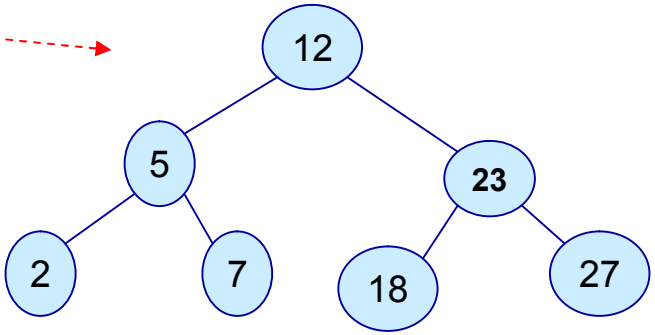
  { 2, 5, 7, 12, 18, 23, 27 }

  - Einfügen in der Reihenfolge
    - 2, 5, 12, 7, 18, 27,23



unbalanciert
im Extremfall zur
Liste entartet

  - Einfügen in der Reihenfolge
    - 12, 23, 5, 2, 18, 27, 7

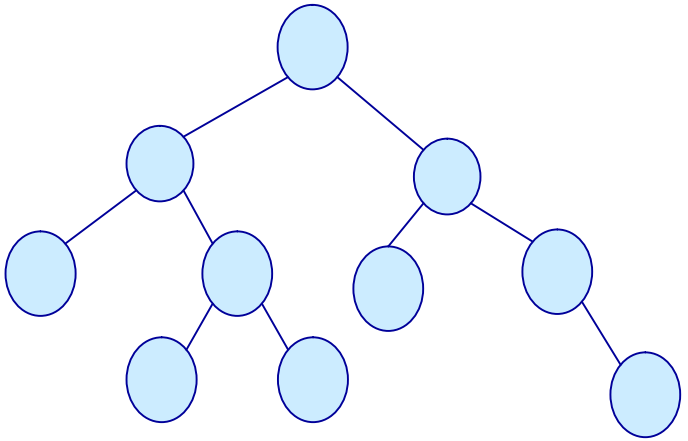


gut balanciert – jedes Element in log N
Schritten von der Wurzel aus erreichbar

  - Fazit: Form des entstandenen Baumes hängt von der Reihenfolge des Einfügens (und Löschens) ab.

Balancierte Bäume

- Ein Binärbaum hat
  - maximal 2^k Knoten der Tiefe k
  - maximal $2^0+2^1+\ldots+2^k = 2^{k+1}-1$ Knoten der Tiefe $\leq$ k
  - d.h. Ein Baum der Tiefe k hat maximal $2^{k+1}-1$ viele Knoten
- Baum mit N Knoten heißt *balanciert*, falls
  - alle Schichten – bis auf die unterste sind voll besetzt
- Für einen N-elementigen balancierten Baum der Tiefe k gilt:
  - $2^k \leq N$
  - $k \leq \log_2 N$,
  - sogar: $k \leq \lfloor \log_2 N \rfloor$ (weil k ganzzahlig ist)
- Beispiel:
  - N=10
  - tiefe $\leq \lfloor \log_2(10) \rfloor = 3$

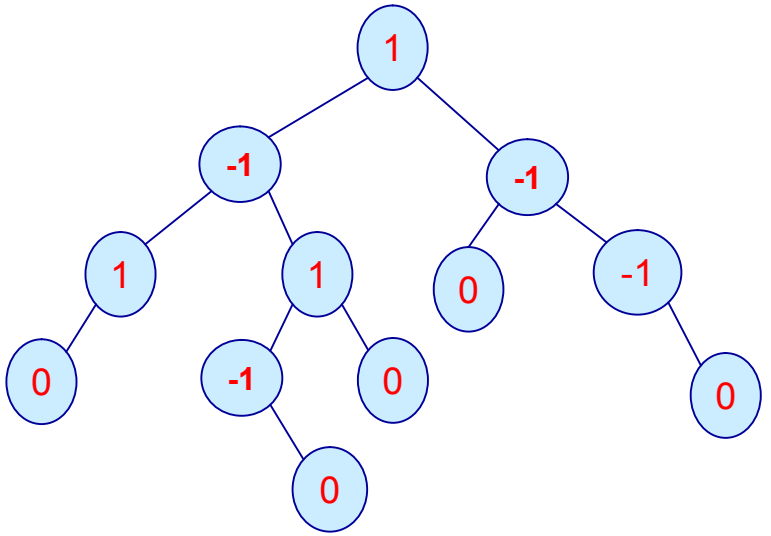


Vorteil balancierter Bäume

- Suchen ist O(log(N))
  - Grund:
    - Anzahl der Vergleiche:
      - falls gesuchtes Element vorhanden:
        - Tiefe des Elementes
      - falls nicht vorhanden:
        - Tiefe des Baumes
    - In jedem Falle
      - ≤ Tiefe des Baumes, also
      - ≤ $\log_2(N)$
- Problem:
  - Wie kann ein Baum *balanciert* bleiben
  - trotz unvorhersehbarer
    - Löschoperationen
    - Einfügeoperationen
- Lösung :
  - Schwäche Balance-Bedingung ab
  - Reorganisiere ggf. nach jedem Einfügen und Löschen
  - Wichtig: AVL-Bäume
    - nach Erfindern Adel‘son-Vel‘skii und Landis

AVL-Bäume

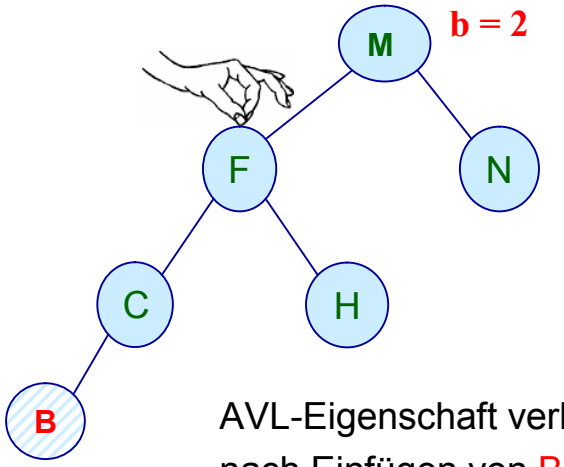
- Binärbäume mit AVL-Eigenschaft
  - AVL: Für jeden Knoten gilt:
    - Die Tiefe von linkem und rechtem Teilbaum unterscheiden sich maximal um 1
- Jedem Knoten k kann man eine Balance-Zahl b(k) zuordnen:
  - b(k) = Tiefe(left(k)) - Tiefe(right(k))
- Für jeden Knoten eines AVL-Baumes muss gelten:
  - b(k) $\in$ { -1, 0, 1}



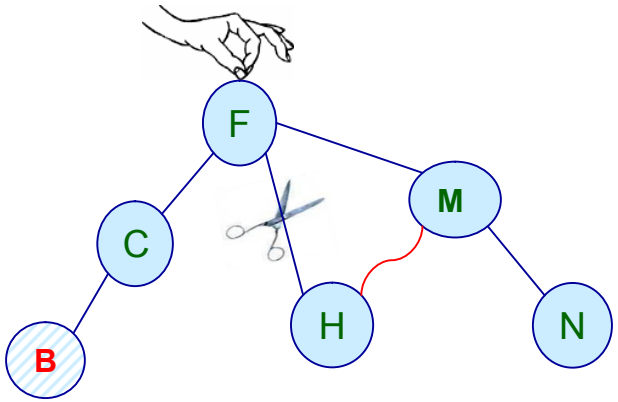
AVL-Baum mit Balance-Werten

Reorganisation

- AVL-Eigenschaft kann temporär zerstört werden
  - durch Einfügen
  - durch Entfernen
- Rettung:
  - Reorganisation durch *Rotationen*
- Rotation:
  - Lokale Reorganisation
  - Erhält Ordnung im Binären Suchbaum



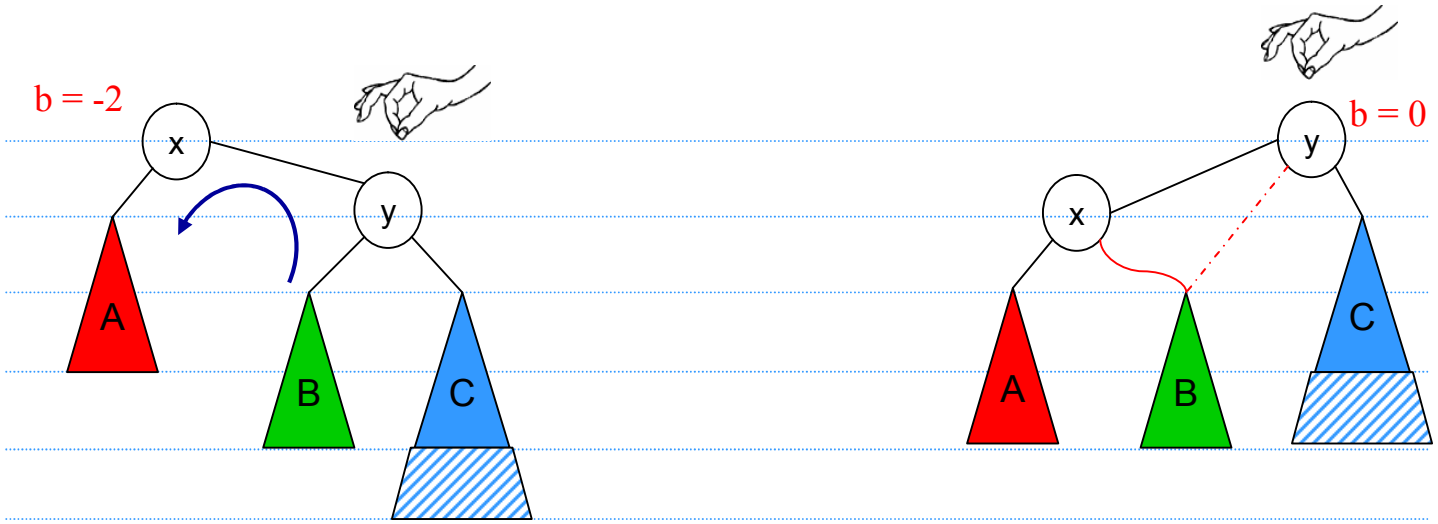
AVL-Eigenschaft verletzt nach Einfügen von B



AVL-Eigenschaft wieder hergestellt - durch *Rotation*

Reorganisation beim Einfügen

- Fall 1: Einfache „Rotation“

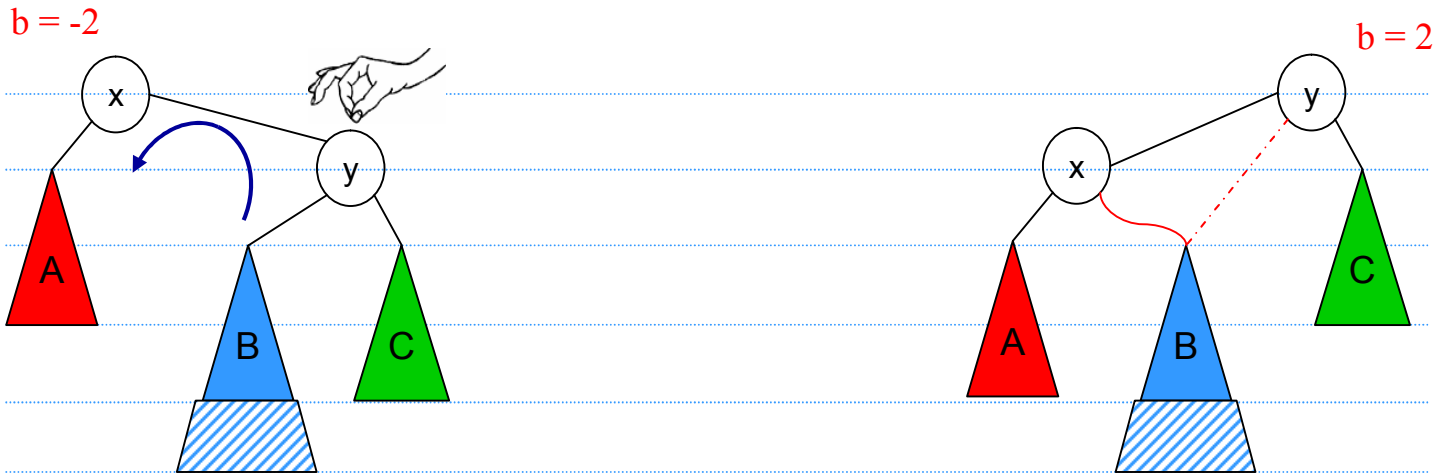


AVL-Eigenschaft verletzt
nach Einfügen rechts von y

AVL-Eigenschaft
wieder hergestellt

Reorganisation beim Einfügen

- Fall 2: Einfache Rotation reicht nicht:

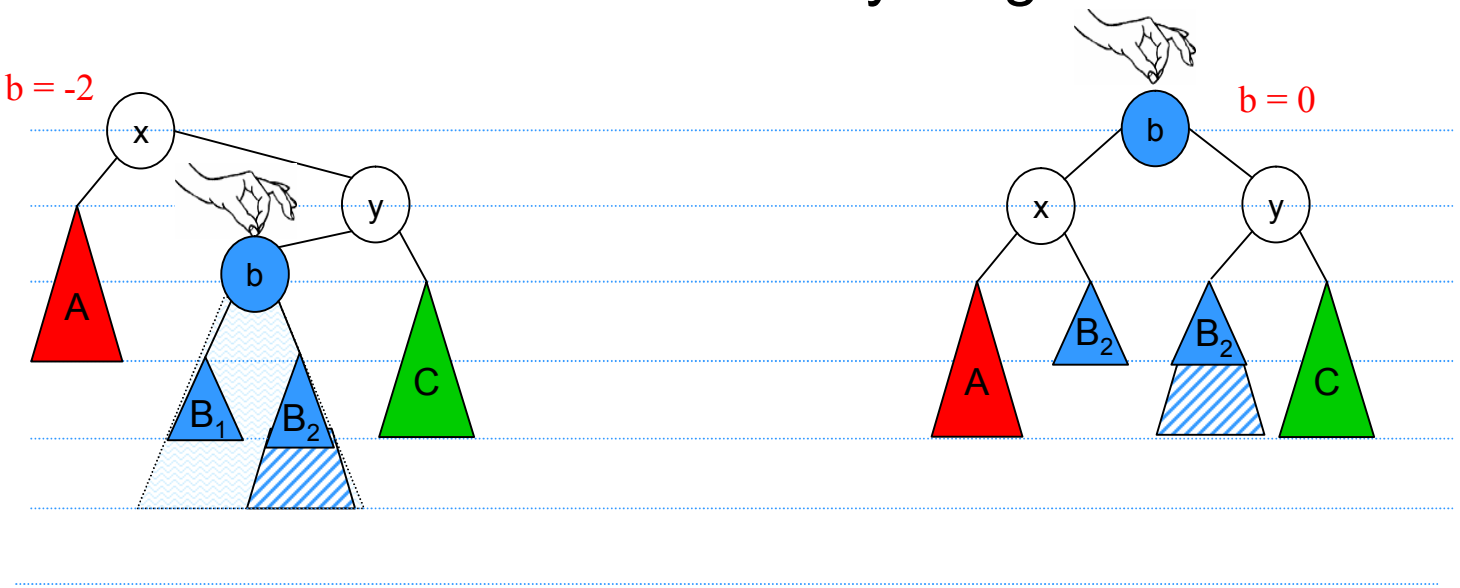


AVL-Eigenschaft verletzt
nach Einfügen links von y

AVL-Eigenschaft immer
noch verletzt

Reorganisation beim Einfügen

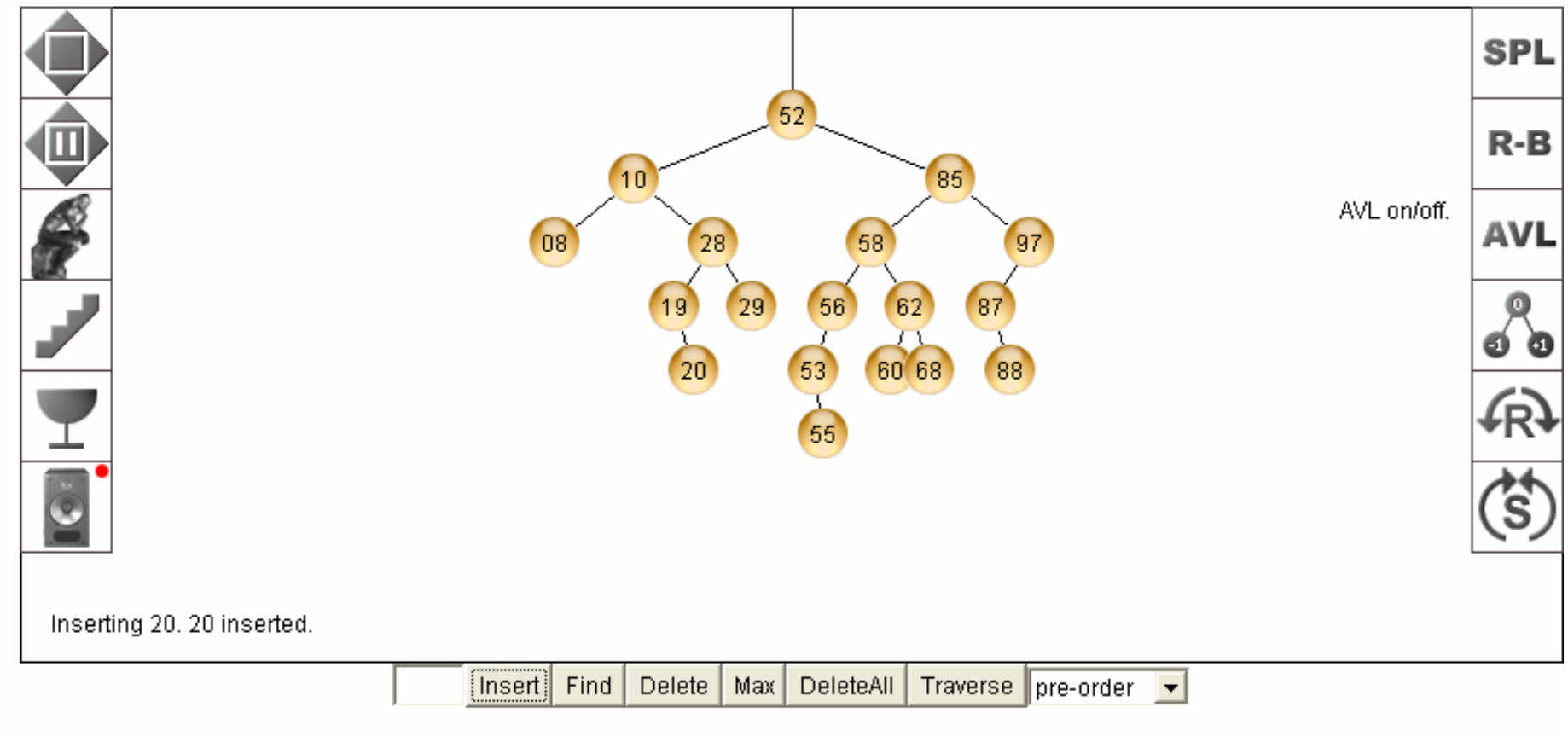
- Fall 2: B wird zwischen x und y aufgeteilt



AVL-Eigenschaft verletzt nach Einfügen zwischen y und z

AVL-Eigenschaft wieder hergestellt

- Analoge Reorganisationen beim Entfernen von Knoten

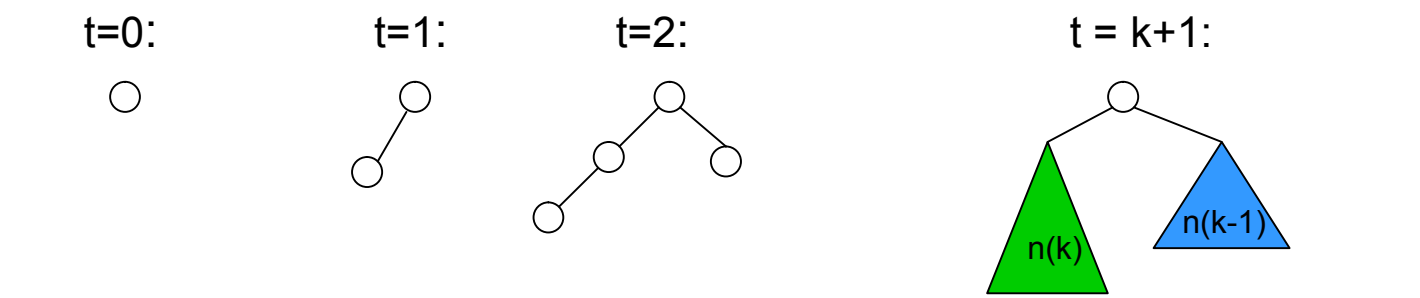


- http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm

Vorteile von AVL-Bäumen

- Vorteil von AVL-Bäumen
  - Nur lokale (begrenzte) Reorganisation notwendig
    - Betrifft nur Knoten zwischen neuem Element und Wurzel
  - Suche in AVL Bäumen effizient, denn
- Satz: Ein AVL-Baum mit N Knoten hat maximal Tiefe $2*\log_2 N$
- Korollar: Suchen, einfügen und entfernen im AVL-Baum ist O(log(N))

Sei n(t) die Mindest-Knotenanzahl eines AVL-Baumes von Tiefe t:



n(0) = 1 n(1) = 2 n(2) = 4 … n(k+1) = n(k) + n(k-1) + 1

Beweis des Satzes

- Satz: Ein AVL-Baum mit n Knoten hat maximal Tiefe $2*\log_2 n$

Wir haben gesehen: Ein AVL-Baum der Tiefe t hat mindestens n(t) Knoten mit

$$n(0) = 1, n(1) = 2,$$
$$n(t) = n(t-1) + n(t-2) + 1 \quad > \quad 2 \times n(t-2)$$

*relativ grobe Abschätzung)**

Für t = 2k+1 ungerade folgt: *(t gerade: Übung)*

$$n(t) > 2 \times n(t-2)$$
$$> 2 \times 2 \times n(t-4)$$
$$> 2^k \times n(t-2k) = 2^k \times n(1) = 2^{k+1}$$

Logarithmieren ergibt:

$$k+1 < \log_2 n(t), \quad also \quad t = 2k+1 < 2 \times \log_2 n(t)$$

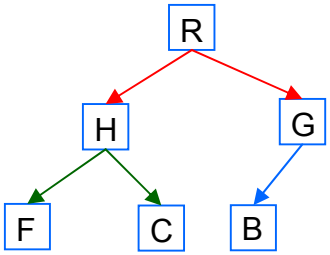
)* Eine genauere Abschätzung liefert am Ende sogar: $t < 1.44 \log_2 n + const$

Vollständige Bäume

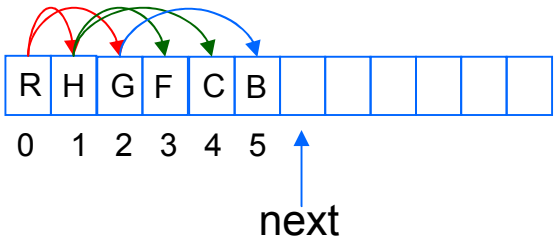
- Ein vollständiger Baum ist ein Binärbaum mit
  - Formeigenschaft:
    - Alle Ebenen bis auf die letzte ist vollbesetzt
    - Die letzte Ebene wird von links nach rechts aufgefüllt

- Ein vollständiger Baum kann als Array B gespeichert werden:
  - Elemente: B[0], B[1], B[2] ….
  - B[i] hat Vater : B[(i - 1) / 2]
  - B[i] hat Söhne : B[2*i+1] und B[2*i+2]

Vollständiger Baum …

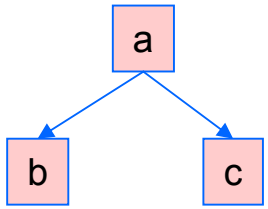


… als Array B:



Heaps

- Ein (Max)Heap ist ein vollständiger Baum, in dem jeder Sohn kleiner als sein Vater ist

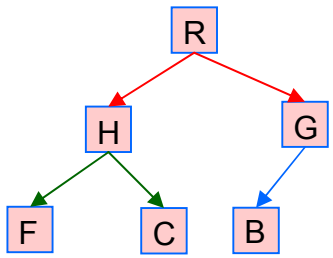


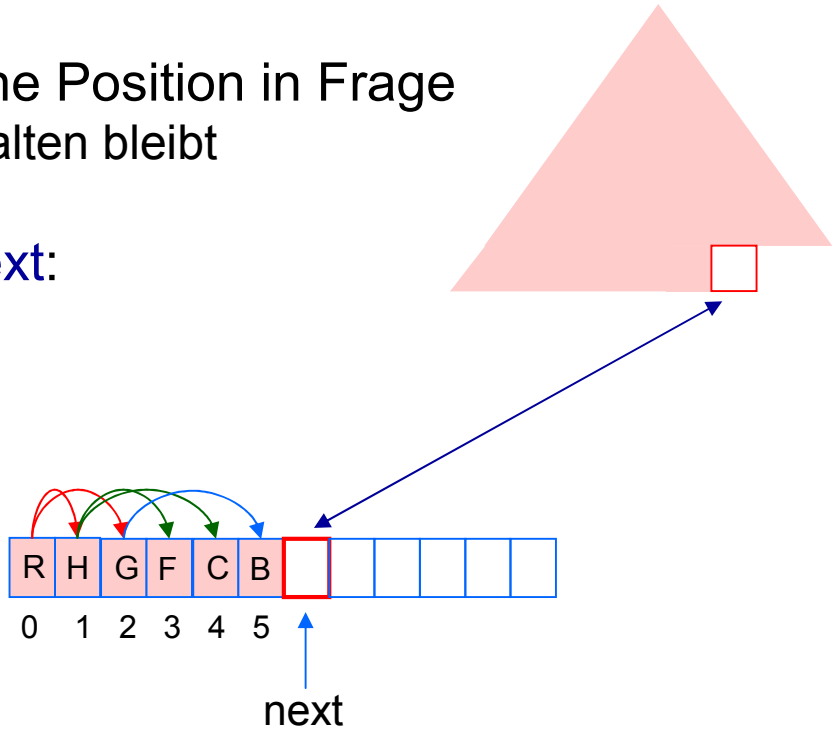
$\Rightarrow a \geq b$ und $a \geq c$

- Die wichtigsten Heap-Operationen
  - insert
    - füge ein beliebiges Element ein
  - deleteMax
    - entferne das größte Element
    - d.h. die gegenwärtige Wurzel
- MinHeap kann man analog definieren:
  Jeder Sohn ist *größer* als der Vater
  - Entsprechende Operationen sind
    - insert und deleteMin

insertHeap

- Für das Einfügen kommt nur eine Position in Frage
  - damit die Formeigenschaft erhalten bleibt
- Im Array ist das die Position next:

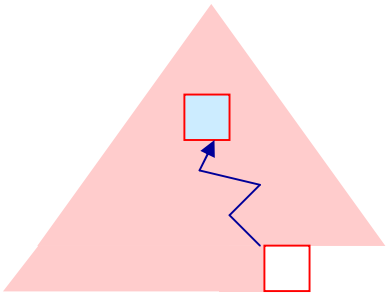




- allerdings kann dabei die Ordnungseigenschaft verletzt werden
  - wir korrigieren dies mit *aufsteigen* (engl.: *upHeap*)

aufsteigen

- Stellt Ordnung wieder her
  - nach Einfügen eines Elementes
  - eine Version von *bubbleUp* im Baum
    - solange Element größer als der Vater
    - vertausche Sohn und Vater
    - bis Element an die richtige Stelle „*gebubbelt*“

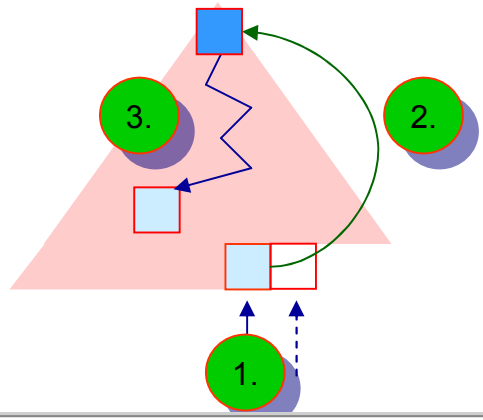


```
public void insert(int n){
  theHeap[nextPos]=n;
  aufsteigen(nextPos);
  nextPos++;
}

/** Sohn steigt auf, solange er größer als der Vater */
private void aufsteigen(int sohn){
  int vater=(sohn-1)/2;
  if ( theHeap[vater] < theHeap[sohn]){
        swap(theHeap,vater,sohn);
        aufsteigen(vater);
  }
}
```

getNext und absickern

- liefert und entfernt größtes Element
  - das ist das Element in der Wurzel

1. Erniedrige nextPos
2. Kopiere letztes Blatt in die Wurzel
   - Form gewahrt – Ordnung verletzt
3. Lass die Wurzel nach unten *absickern* (engl.: *downHeap*)
   - Falls größer als beide Söhne: fertig.
   - Ansonsten vertausche mit größtem Sohn
   - lass weiter absickern



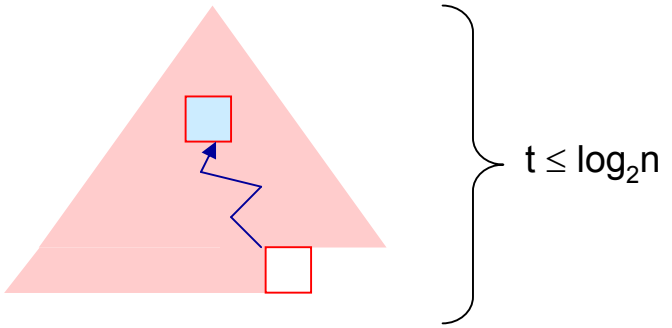

```
public int getNext(){
  int result = theHeap[0];
  nextPos--;
  theHeap[0] = theHeap[nextPos];
  absickern(0);
  return result;
}
```

```
/** Vater sickert abwärts, bis größer als beide Söhne */
private void absickern(int vater){
  int lSohn=2*vater+1;
  int rSohn=lSohn+1;
  if(lSohn >= nextPos) return; // kein Sohn da
  if(rSohn == nextPos){        // nur linker Sohn da
    if(theHeap[vater]<theHeap[lSohn])
      swap(theHeap,vater,lSohn);
    return;
  }else{                       // vater hat zwei Söhne
    int maxSohn =              // bestimme größten
     (theHeap[lSohn]>theHeap[rSohn])? lSohn: rSohn;
    if (theHeap[vater] >= theHeap[maxSohn]) return;
    else { swap(theHeap,vater,maxSohn);
           absickern(maxSohn); // weiter sickern
    }
  }
}
```

Komplexitäten

- Ein Heap der Tiefe t
  - hat mindestens
    $N \geq 1 + 2 + 2^2 + 2^3 + ... + 2^{t-1} + 1$
    $= 2^t$ Elemente

  also
  - $t \leq \log_2 N$

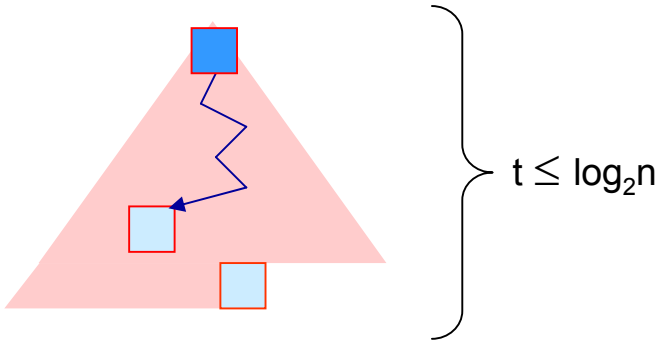


- *insert* ist O(log(n)):
  - upHeap ist proportional der Tiefe

- *getNext* ist O(log(n)):
  - absickern ist proportional zur Tiefe

- Ein Heap ist ein guter Kompromiss:
  - Nicht komplett geordnet

  aber
  - einfügen und entfernen O(log(n))



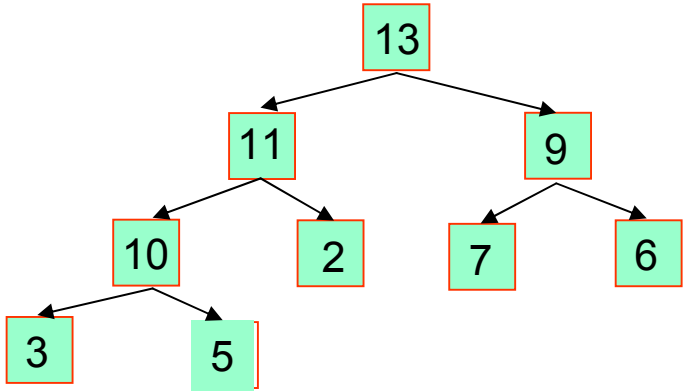
Priority-Queue: Warteschlange mit Prioritäten

- Daten mit verschiedenen Prioritäten werden in eine Queue eingefügt
  - *insert*
- Elemente mit höherer Priorität sollen zuerst drankommen
  - *getNext*
- Lösung: Organisiere die Daten in einem Heap
- Beispiele
  - am Flugschalter
    - Piloten > CabinCrew > First Class > Business Class > Economy
  - Druckaufträge
    - Systemverwalter > Chef > Mitarbeiter > Praktikant
- Problem: Elemente gleicher Priorität sollen FIFO drankommen (First in First out)
  - Führe Zeitstempel ein
  - Ordnung:
    - erst nach Priorität
    - dann nach Zeitstempel

Sortieren mit einem Heap

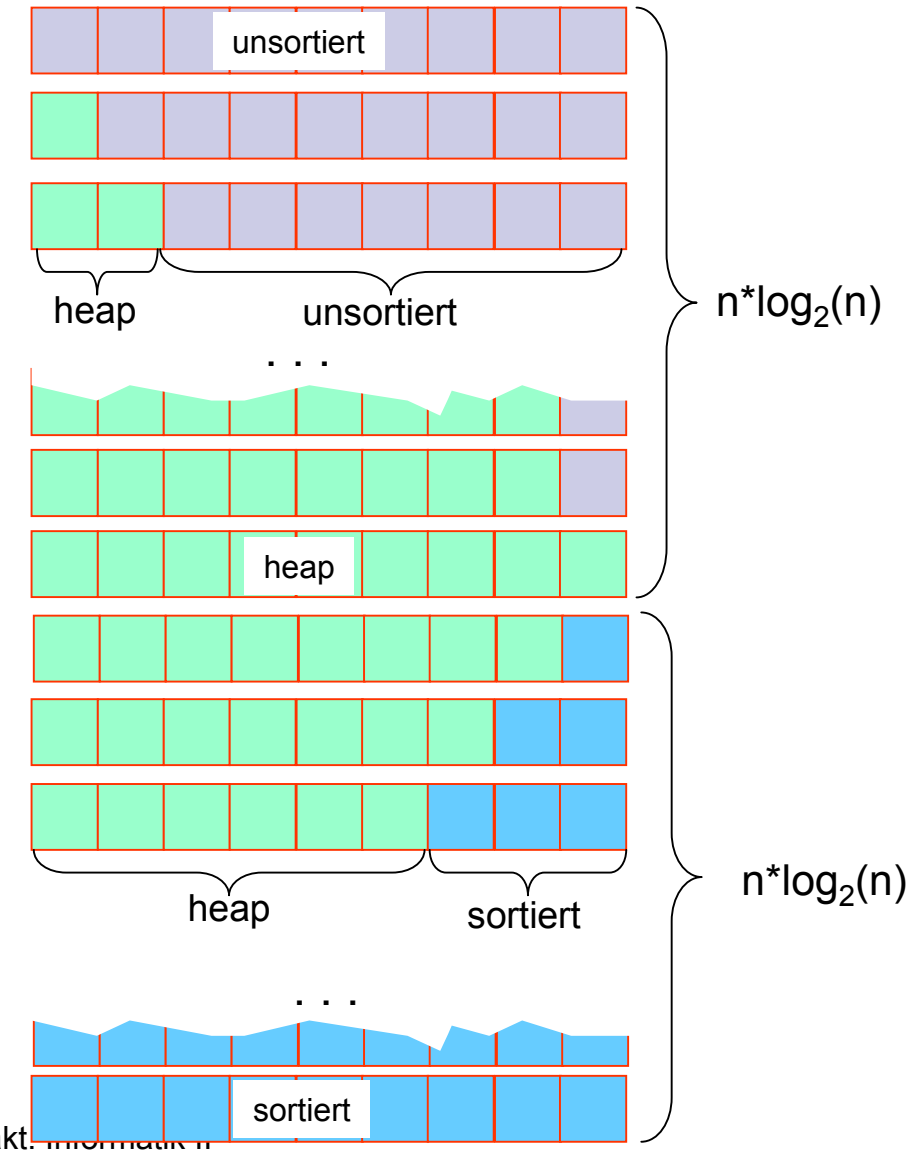
1. Gegeben: Folge von n Elementen

2. Füge sie in einen Heap ein



3. Entnehme die Elemente aus dem Heap
   - Dabei werden sie in (absteigend) sortierter Reihenfolge geliefert

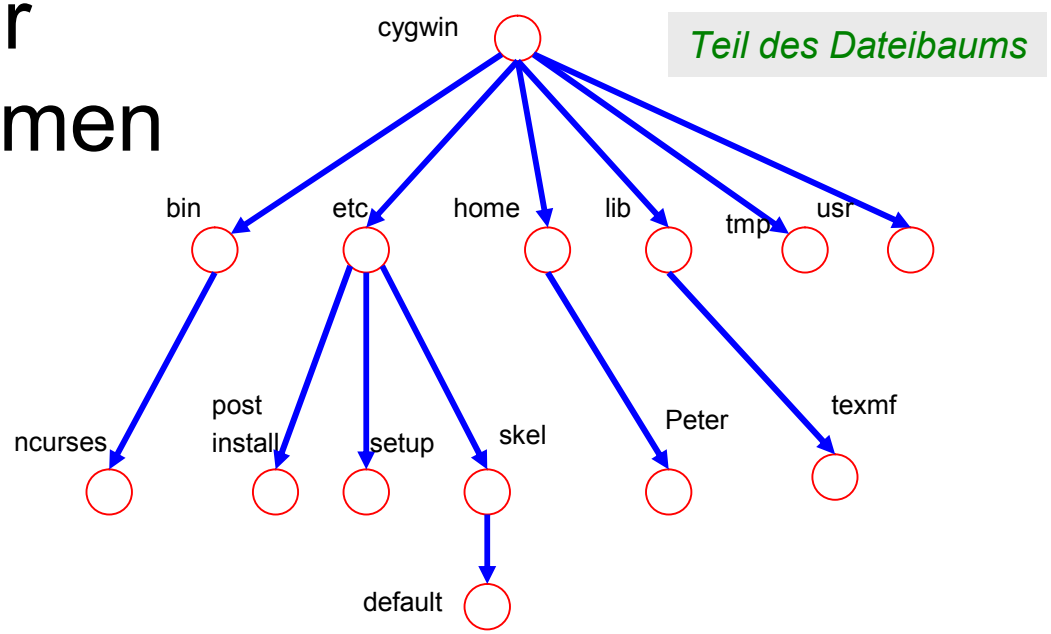
HeapSort



- Sortieren mittels Heap im gleichen Array, in dem sich die Daten befinden
- 1. Phase
  - Baue den Heap im Anfangsabschnitt des Arrays
  - n*O(insert)) = O(n*log(n))
- 2. Phase
  - entnehme die Elemente aus dem Heap und schreibe sie in den Endabschnitt
  - n*O(getNext) = O(n*log(n))
- Komplexität also:
  - 2*O(n*log(n)) = O(n*log(n))

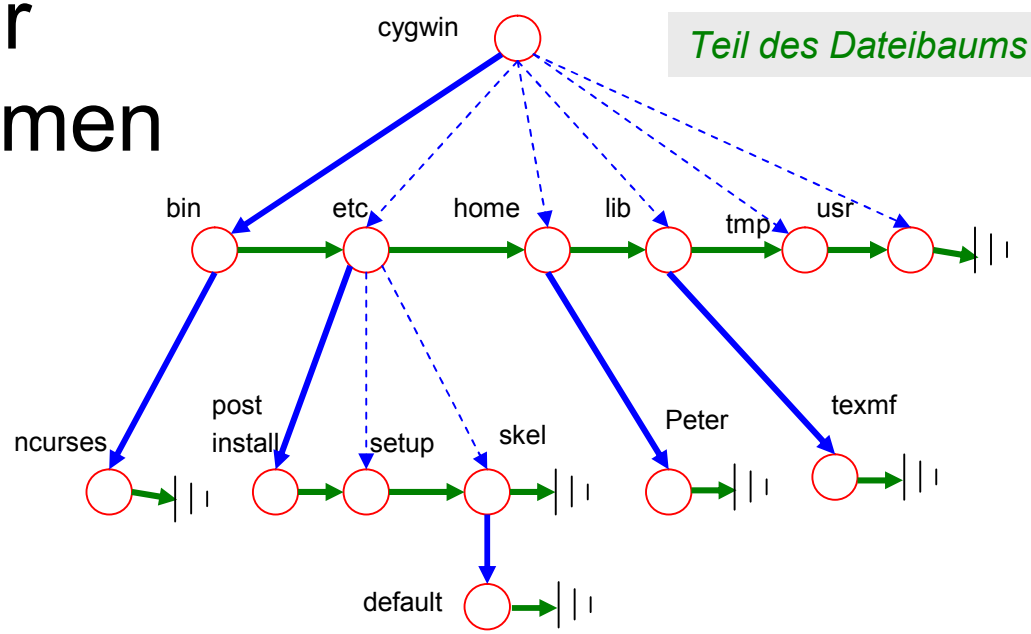
Bäume mit variabler Anzahl von Teilbäumen

- In der Praxis häufig anzutreffen
  - Beispiel: Dateisystem



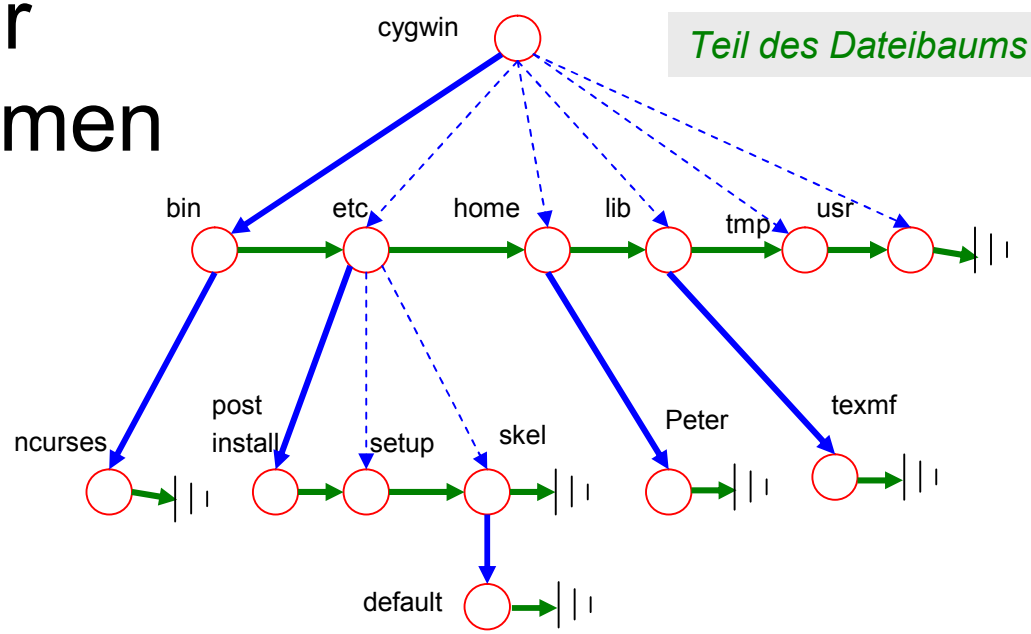
Bäume mit variabler Anzahl von Teilbäumen

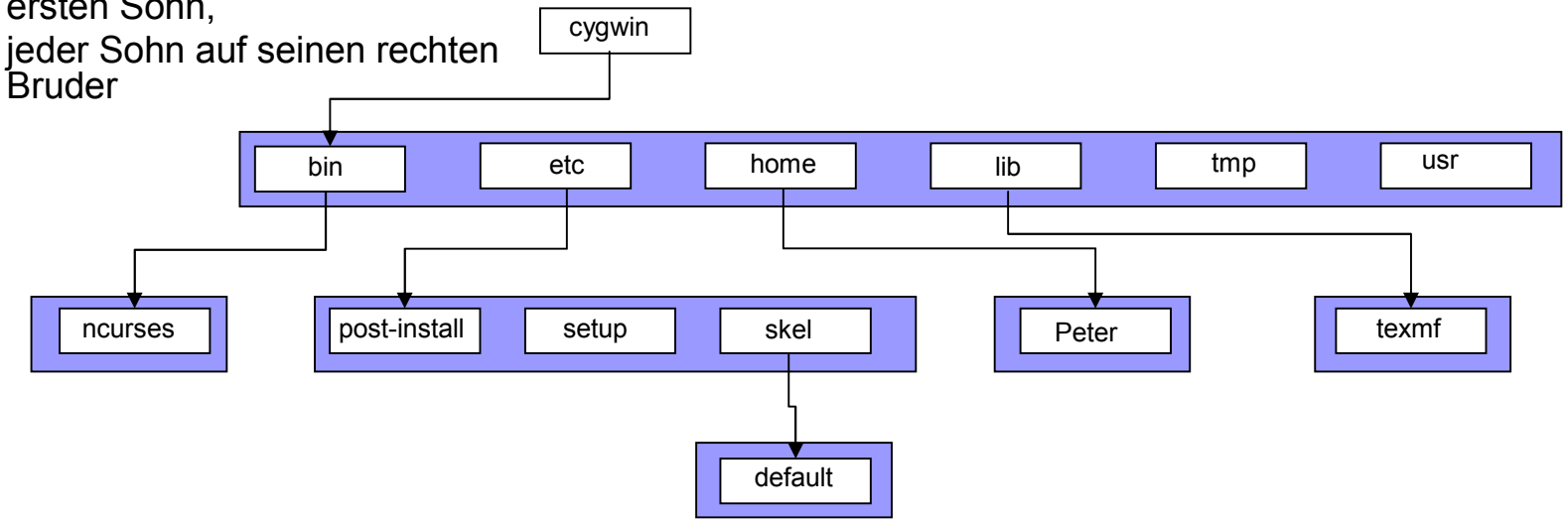
- In der Praxis häufig anzutreffen
  - Beispiel: Dateisystem
- Implementierung
  - Jeder Knoten hat Liste von Söhnen



Bäume mit variabler Anzahl von Teilbäumen

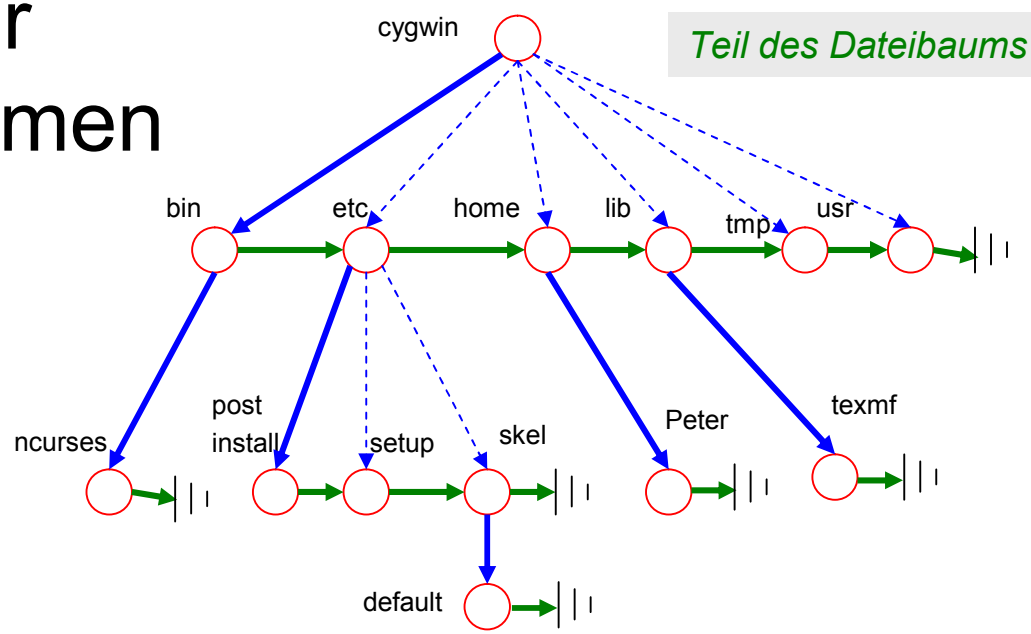
- In der Praxis häufig anzutreffen
  - Beispiel: Dateisystem
- Implementierung:
  - Jeder Knoten hat Liste von Söhnen
- Das bedeutet
  - Jeder Knoten zeigt auf den ersten Sohn,
  - jeder Sohn auf seinen rechten Bruder

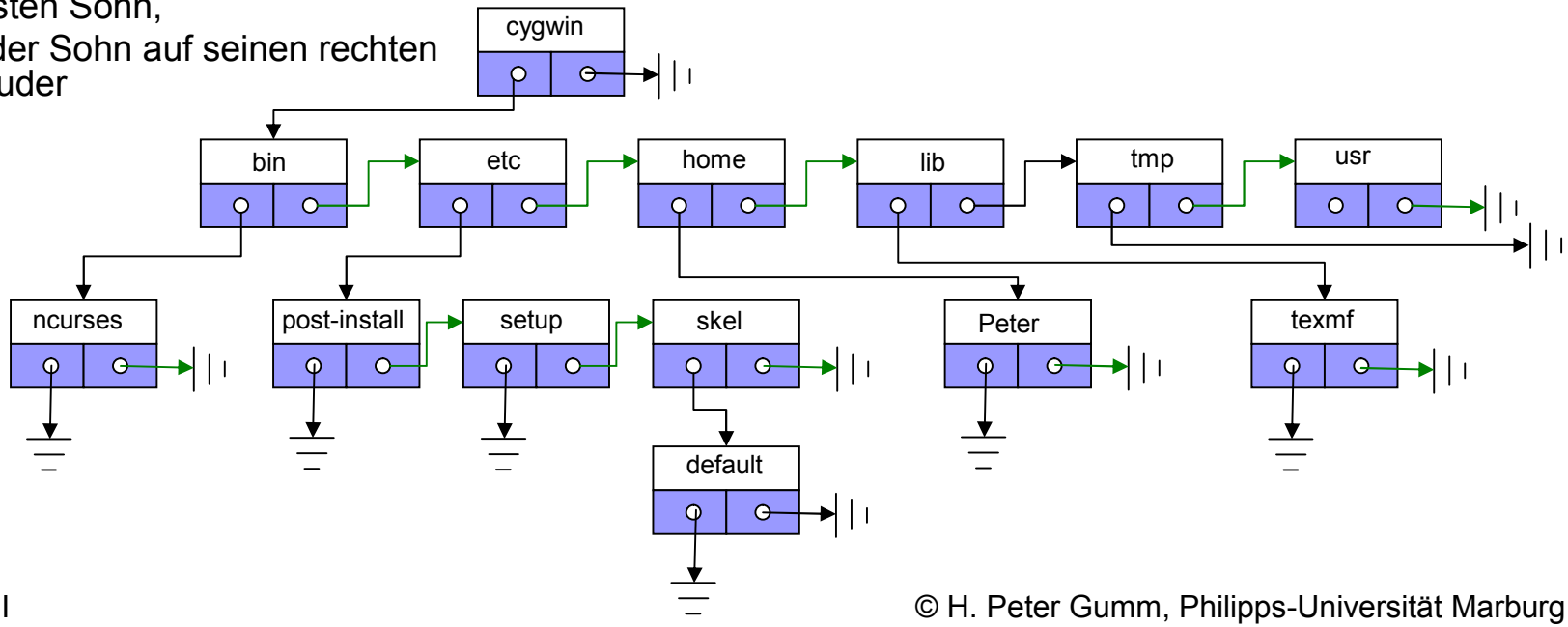




Bäume mit variabler Anzahl von Teilbäumen

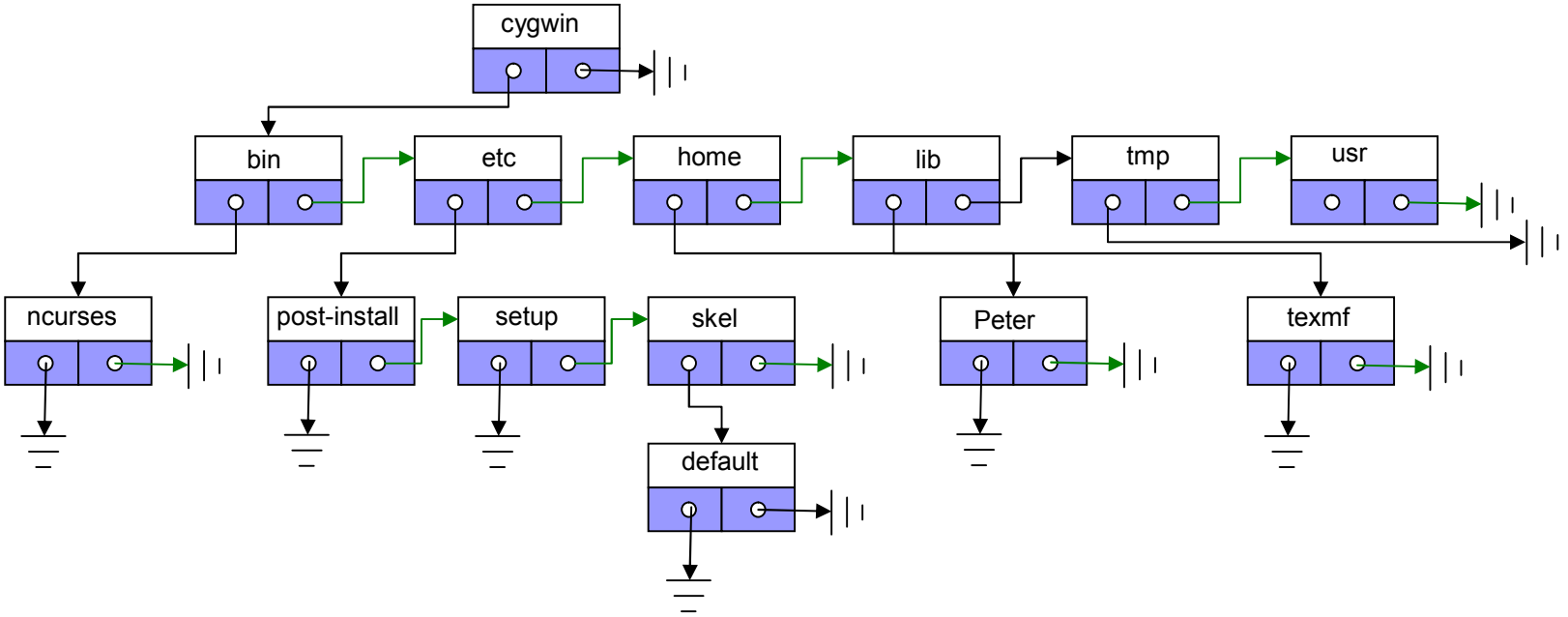
- In der Praxis häufig anzutreffen
  - Beispiel: Dateisystem
- Implementierung:
  - Jeder Knoten hat Liste von Söhnen
- Das bedeutet
  - Jeder Knoten zeigt auf den ersten Sohn,
  - jeder Sohn auf seinen rechten Bruder





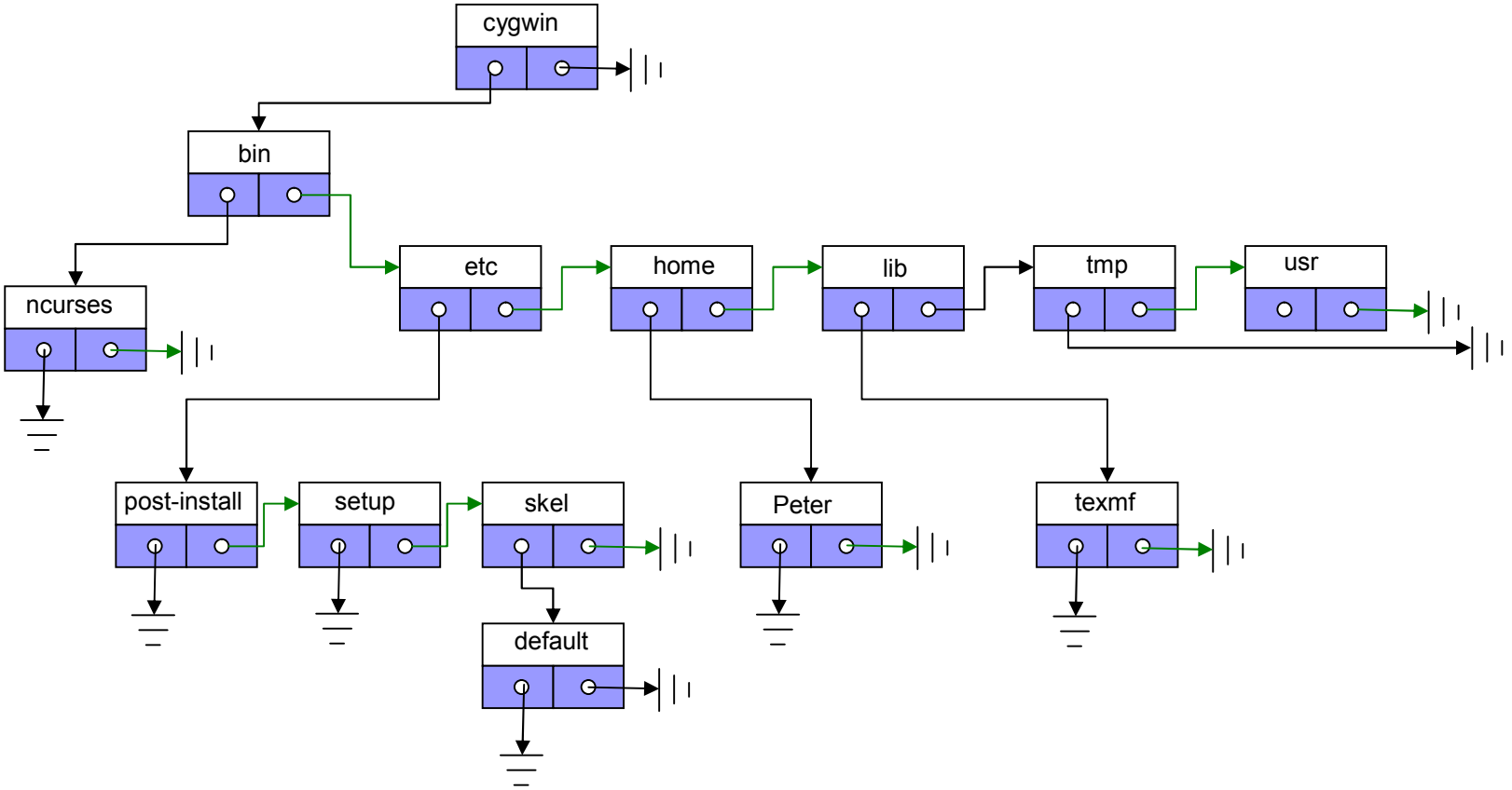
Bäume mit variabler Anzahl von Teilbäumen

- Siehe da – ein Binärbaum



Bäume mit variabler Anzahl von Teilbäumen

- Siehe da – ein Binärbaum



Bäume mit variabler Anzahl von Teilbäumen

- Siehe da – ein Binärbaum