



Bäume

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



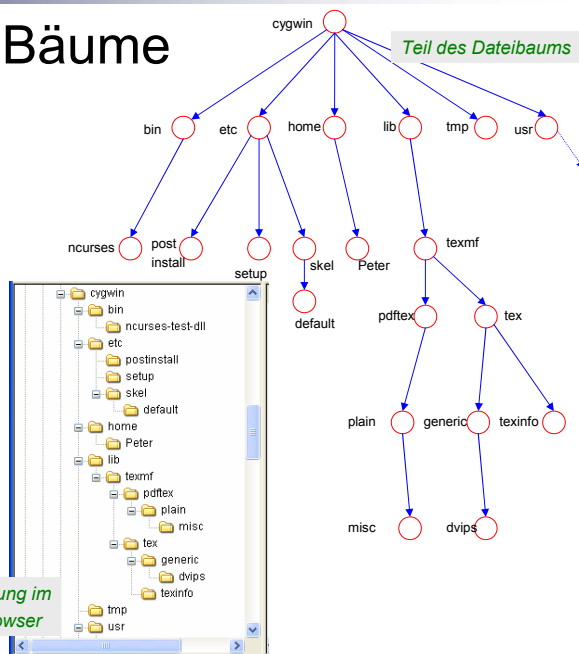
Hierarchien - Bäume

■ 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

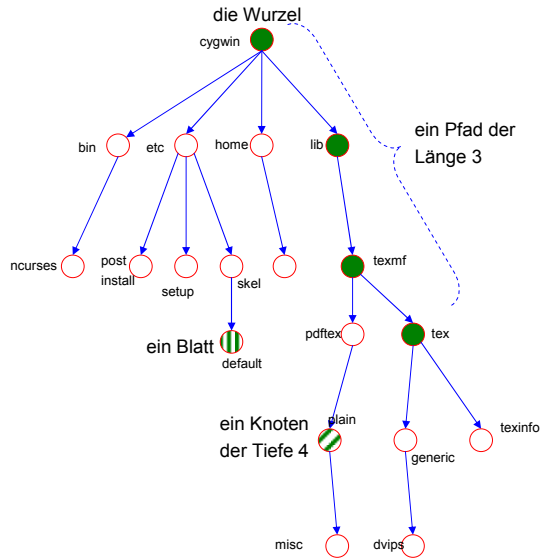
- Knoten :
 - Verzeichnisse
- Kanten:
 - Unterverzeichnis
- Blätter:
 - Dateien
- Wurzel:
 - Laufwerk (C:)





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
 - Formal:
 - leerer Baum hat Tiefe -1.
- **Tiefe des Baumes:**
 - max. Tiefe eines Knoten
 - = max. Länge eines Pfades

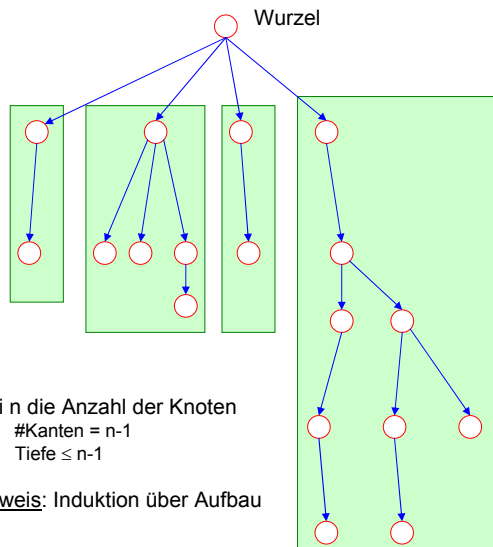


der Baum hat Tiefe 5



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

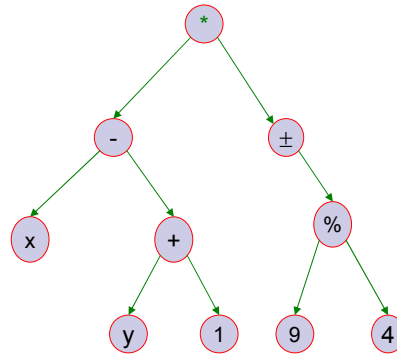


- Sei n die Anzahl der Knoten
 - #Kanten = $n-1$
 - Tiefe $\leq n-1$
- Beweis: Induktion über Aufbau



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

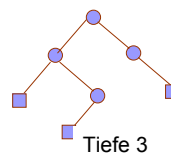
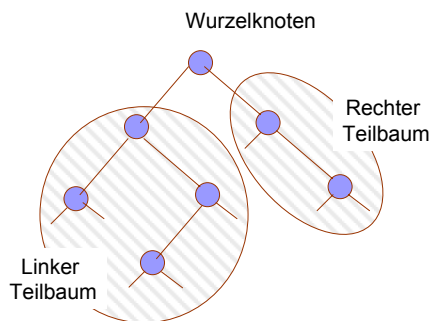


Operatorbaum für
 $(x - (y + 1)) * (- (9 \% 4))$



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

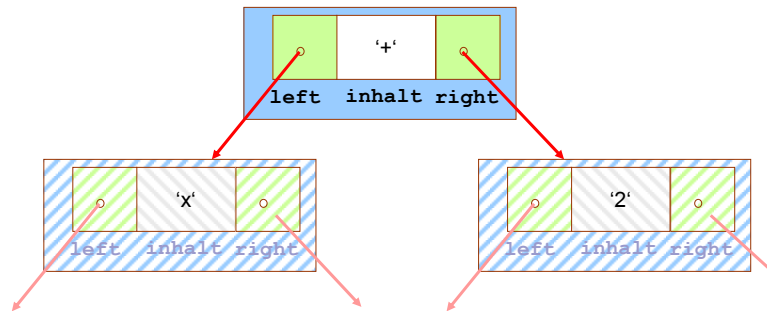




Baumknoten

```
class Knoten{
    Knoten left;
    char content;
    Knoten right;
}
```

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



Konstruktoren, Prädikate, Selektoren

■ Konstruktoren

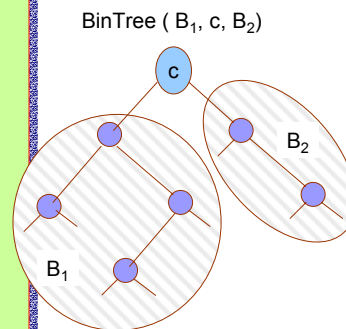
- `BinTree()`
 - Der leere Baum
- `BinTree(B1, c, B2)`
 - neuer `BinTree` mit
 - linkem Teilbaum `B1`
 - rechtem Teilbaum `B2`
 - Inhalt der Wurzel: `c`

■ Prädikat `isEmpty`

- Testet, ob ein `BinTree` leer ist, oder einen rechten und linken Teilbaum enthält

■ Selektoren `left`, `right`, `content`

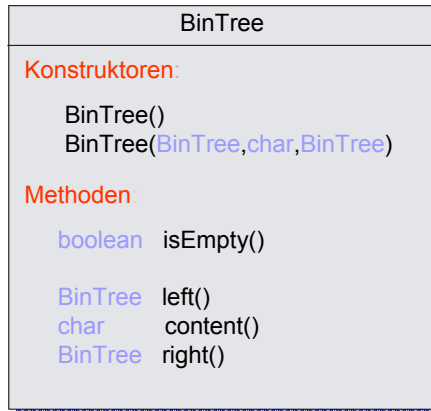
- Falls der `BinTree` nicht leer ist, liefert
 - `left()` den linken Teilbaum
 - `right()` den rechten Teilbaum
 - `content()` den Inhalt der Wurzel





Indexkarte für BinTree

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



lichtung
 manche meinen
 lechts und rinks
 kann man nicht verwechsellern
 werch ein illtum
 (Ernst Jandl)

- Wir verstecken alle anderen Felder und Methoden hinter dem Schlüsselwort **private**



Implementierung als Ergebnistyp

```
class BinTree{
  private Knoten wurzel;

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

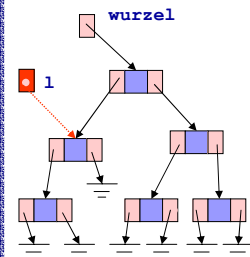
  BinTree(BinTree b1, char c, BinTree b2){
    wurzel = new Knoten(b1.wurzel,c,b2.wurzel);
  }

  // Prädikat
  boolean isEmpty(){ return wurzel==null; }

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

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

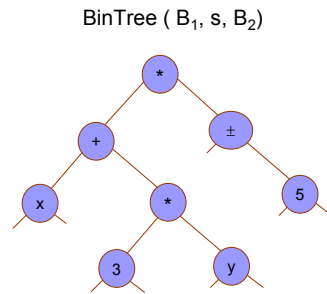
- Implementierung verläuft analog zu Listen





Rekursion auf Binärbäumen

- Binärbäume sind induktiv definiert
 - Der leere Baum ist ein Binärbaum
 - Sind B_1 und B_2 Binärbäume und s ein String, dann ist der Baum mit Wurzel s , linkem Teilbaum B_1 und rechtem Teilbaum B_2 ein Binärbaum
- Operationen f auf Binärbäumen sind rekursiv
 - Falls `isEmpty(B)`: gib $f(B)$ direkt an
 - Ansonsten beschreibe wie sich der Wert von f aus $f(B_1)$, $f(B_2)$ und s ergibt.
- Beispiel: tiefe (engl.: depth)
 - Falls `isEmpty(B)`: 0
 - Ansonsten $\max(\text{left().depth()}, \text{right().depth()}) + 1$
- Beispiel: `exists(char c)` (ist c vorhanden?)
 - Falls `isEmpty(B)`: false
 - Ansonsten: `left().exists(c) || content()== c || right().exists(c)`



depth: 4

Dieser Baum repräsentiert $(x+3*y)*(-5)$



Einfache Baumoperationen



```

class XBinTree extends BinTree{

    int depth(){
        if(isEmpty()) return -1;
        else return
            max( ((XBinTree)left()).depth()
                ((XBinTree)right()).depth() ) + 1;
    }

    boolean exists(char c){
        if(isEmpty()) return false;
        else return
            ((XBinTree)left()).exists(c)
            || content()==c
            || ((XBinTree)right()).exists(c);
    }

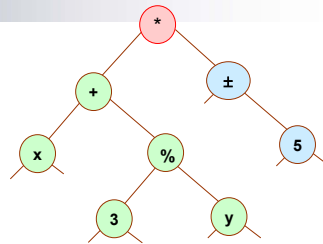
    private static int max(int x, int y) {
        return (x < y)? y : x;
    }
}

```

- `isEmpty()`, `left()`, `right()` werden aus der Oberklasse geerbt.
- `left()` liefert einen `BinTree()`
- `depth()` ist nur in der Unterklasse definiert – für `XBinTrees`
- Wir brauchen casts um die `BinTrees` in `XBinTrees` zu verwandeln
- Hilfsmethode `max` ist nicht an ein Objekt gebunden. Wir erklären sie als Klassenmethode, also `static`.
- Da sie nur hier gebraucht wird, verstecken wir sie mit `private`.

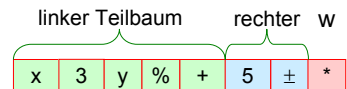
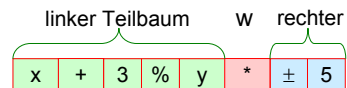
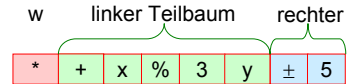


Traversierung



- 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)



Die innere Klasse BinTree.Knoten

- 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)+")"; }

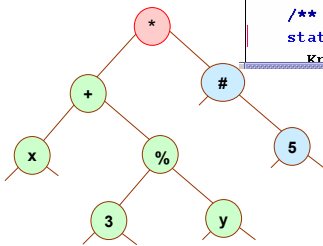
    static String postfix(Knoten k)
    { return (k==null) ? "" :
      postfix(k.left)+postfix(k.right)+" "+k.content.toString()+" "; }
} // Ende innere Klasse Knoten

```



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); }

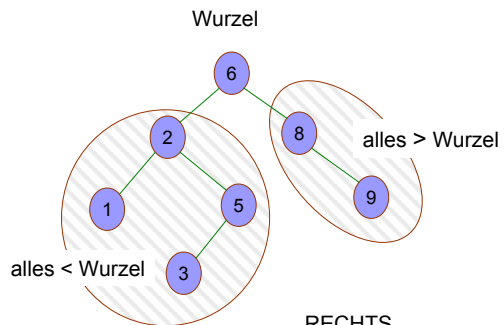
    /** Statische Innere Klasse "BinTree.Knoten". */
    static class Knoten{
        Knoten
    }
}
```

```
BlueJ: Terminal Window
Options
Preorder : * + x % 3 y # 5
Inorder : ((( x ) + (( 3 ) % ( y ))) * ( # ( 5 )))
Postorder : x 3 y % + 5 # *
```



Binäre Suchbäume

- Binärbäume
- Information in Knoten und Blättern
- Invariante
 - alle Elemente in linkem Teilbaum < Wurzel
 - alle Elemente in rechtem Teilbaum > Wurzel

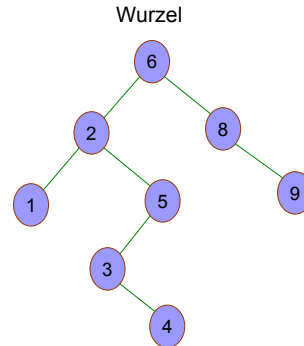


RECHTS
manche meinen,
lechts und rinks
kann man nicht velwechsern,
werch ein illtum
(ernst jandl)



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*

- *Knoten* repräsentiert nichtleeren Baum
- *wurzel* ist null oder ein Knoten – d.h. ein nichtleerer Baum
- Die Methoden
 - `insert`
 - `search`etc. sind in *Knoten* und in *BSTree* implementiert.

```
public class BSTree{
    Knoten wurzel;

    public void insert(int n){
        if (wurzel == null) wurzel = new Knoten(n);
        else wurzel.insert(n);
    }// Ende von BSTree.insert

    private class Knoten{
        Knoten links;
        int inhalt;
        Knoten rechts;

        /* Zwei Konstruktoren */
        Knoten(int i){links=null; inhalt=i; rechts=null;}
        Knoten(Knoten l, int i, Knoten r)
        { links = l; inhalt = i; rechts = r; }

        public void insert(int n){
            if (n < inhalt){
                if(links==null){ links = new Knoten(n);}
                else links.insert(n);
            }else if (n > inhalt){
                if(rechts==null){ rechts = new Knoten(n);}
                else rechts.insert(n);
            }
        }
    }// Ende von Knoten.insert
}
```



Knoten löschen

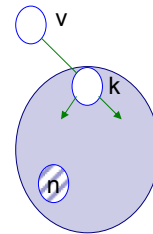
- 1. Fall:
 - Wurzel muss gelöscht werden
- Ansonsten:
 - innerer Knoten zu löschen
 - Führe immer Zeiger auf Vater mit
 - siehe später

```
import java.util.*;
public class BSTree{
    Knoten wurzel;

    public void delete(int n){
        if (wurzel == null) return;
        else if (wurzel.inhalt == n){
            if (wurzel.links==null) wurzel=wurzel.rechts;
            else if (wurzel.rechts==null) wurzel= wurzel.links;
            else wurzel.deleteBelow(wurzel, n);
        }else wurzel.deleteBelow(wurzel, n);
    }
}
```

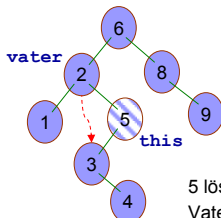
k.deleteBelow(Knoten v, int n)

löscht Knoten mit Inhalt n im Teilbaum mit Wurzel k
Dabei ist v der Vater von k



Knoten k löschen – in Klasse Knoten

- Rekursion in linken oder rechten Teilbaum
- Knoten gefunden
- 1. Fall: Knoten **this** hat nur einen Sohn s
 - ⇒ verbinde s direkt mit seinem Vater



5 löschen:
Vater zeigt auf Sohn

```
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);
        }
    }
}
```

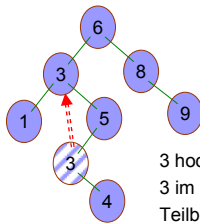
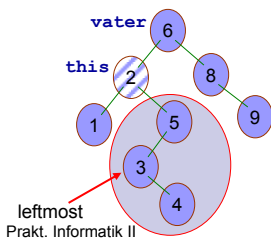


Knoten löschen

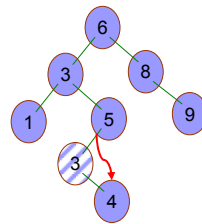
- 2. Fall: Knoten **this** hat zwei Söhne:
 - Kopiere kleinstes Element des rechten Teilbaums in k
 - Lösche das Element aus dem rechten Teilbaum

```
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

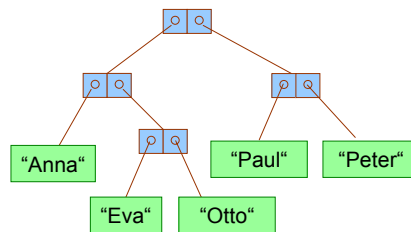


© H. Peter Gumm, Philipps-Universität Marburg



Bäume mit Blättern

- Jeder Zweig soll in einem Blatt enden
- Die Information speichern wir **nur** in den Blättern
- Jeder Knoten hat zwei Unterbäume



```
class Knoten{
  Baum links;
  Baum rechts;
}
```

```
class Blatt{
  String info;
}
```

- Wir wollen Blatt und Knoten zu einer Klasse Baum zusammenfassen. Wir kriegen ein Problem: Wir müssen auch zulassen :

```
Blatt links;
Blatt rechts;
```

... aber ein Blatt ist kein Knoten !



Wie kann man Klassen vereinigen ?

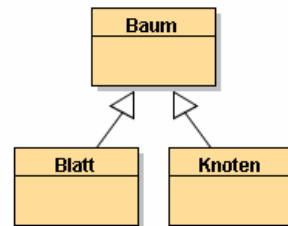
- Wir wollen Blatt und Knoten zu einer Klasse Baum zusammenfassen.

- Blatt wird Unterklasse von Baum
- Knoten wird Unterklasse von Baum



```
class Blatt extends Baum
class Knoten extends Baum
```

- Viele Methoden müssen für alle Bäume funktionieren
 - istBlatt()
 - istKnoten
 - depth()
 - draw()



Default-Methoden redefinieren

- In **Baum** definieren wir die Methoden irgendwie:

```
boolean isBlatt(){
    return false; // äähm na ja ...
}
void draw(){ } // tu nix
```

- In den Unterklassen redefinieren wir sie wieder

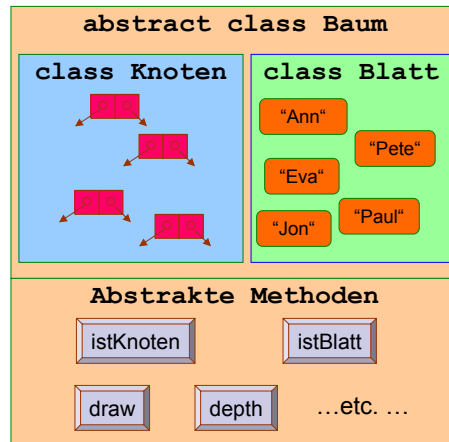
```
// z.B. in Blatt:
boolean isBlatt {
    return true;
}
void draw(
    System.out.println(info);
}
```





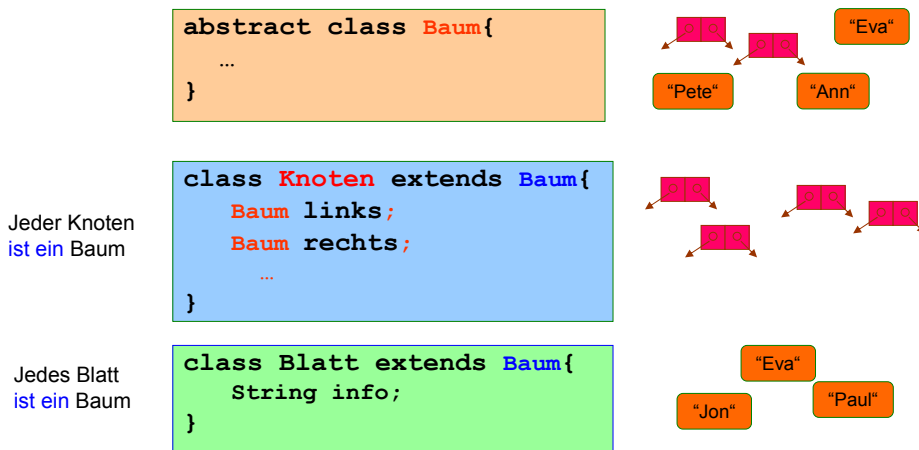
besser: Abstrakte Klassen

- Vereinigung von Unterklassen
- Gemeinsame Methoden
 - In der Oberklasse *abstrakt* erklärt
 - Nur die Signatur wird aufgeführt
 - In jeder nicht abstrakten Unterklasse *implementiert*
- Beispiel
 - Jedes Blatt ist ein Baum
 - Jeder Knoten ist ein Baum
 - Definiere Baum als *abstrakte Klasse*, die Blatt und Knoten umfasst



Abstrakte Klasse Baum

- Klassen werden wechselseitig rekursiv





Implementierung

- Abstrakte Methoden **müssen** in (konkreten) Unterklassen implementiert werden
- Wird vom Compiler geprüft

```
abstract class Baum{
    abstract boolean istBlatt();
    abstract int depth();
    ...
}
```

```
class Knoten extends Baum{
    Baum links, rechts;
    boolean istBlatt(){
        return false;}
    int depth(){
        return
        1+max(links.depth(),
            rechts.depth());
    }
}
```

```
class Blatt extends Baum{
    String info;
    boolean istBlatt(){
        return true;}
    int depth(){ return 0; }
    ...
}
```



Balance

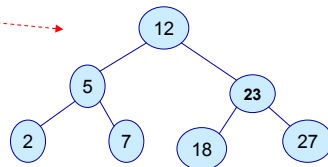
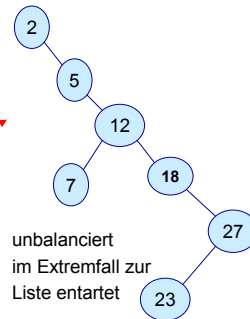
- Suchbaum nach Einfügen der Elemente

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

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

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


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



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



Balancierte Bäume

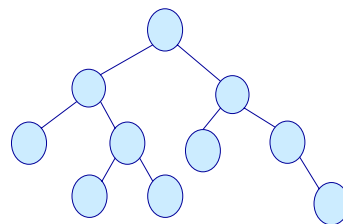
lichtung 
manche meinen
lechts und rinks
kann man nicht velwechsern
werch ein illtum
(Ernst Jandl)

- Ein Binärbaum hat
 - maximal 2^k Knoten der Tiefe k
 - maximal $2^0+2^1+\dots+2^k = 2^{k+1}-1$ 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

- Folgerung: Ein N -elementiger Baum der Tiefe k ist balanciert, gdw.
 - $2^k \leq N$
 - $k \leq \log_2 N$,
 - 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 Elt. vorhanden:
 - Tiefe des Elementes
 - falls nicht vorhanden:
 - Tiefe des Baumes
 - In jedem Falle
 - \leq Tiefe des Baumes, also
 - $\leq \log_2(N)$
 - Problem:
 - Wie kann ein Baum *balanciert* bleiben
 - trotz unvorhersehbarer
 - Löschooperationen
 - 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

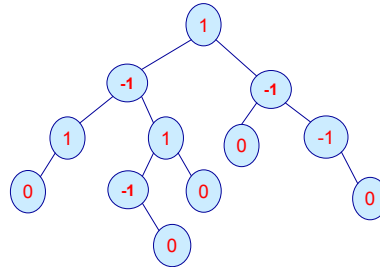
- AVL: Für jeden Knoten gilt:
 - Die Tiefe von linkem und rechtem Teilbaum unterscheiden sich maximal um 1

■ Jedem Knoten k kann man eine Balance-Zahl $b(k)$ zuordnen:

- $b(k) = \text{Tiefe}(\text{left}(k)) - \text{Tiefe}(\text{right}(k))$

■ 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

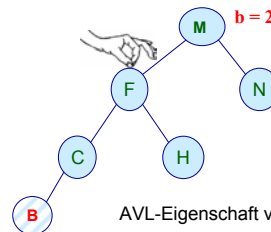
- durch Einfügen
- durch Entfernen

■ Rettung:

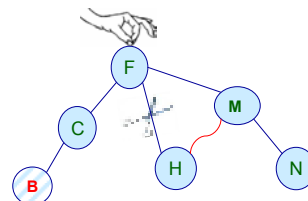
- Reorganisation durch *Rotationen*

■ Rotation:

- Lokale Reorganisation
- Erhält Ordnung im Binären Suchbaum



AVL-Eigenschaft verletzt nach Einfügen von B

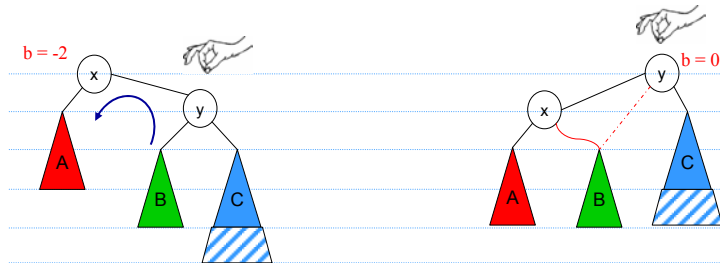


AVL-Eigenschaft wieder hergestellt - durch *Rotation*



Reorganisation beim Einfügen

■ Fall 1: Einfache „Rotation“



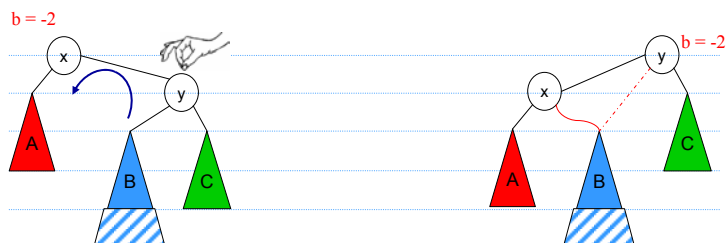
AVL-Eigenschaft verletzt
nach Einfügen rechts von y

AVL-Eigenschaft
wieder hergestellt



Reorganisation beim Einfügen

■ Fall 2: Einfache Rotation reicht nicht:



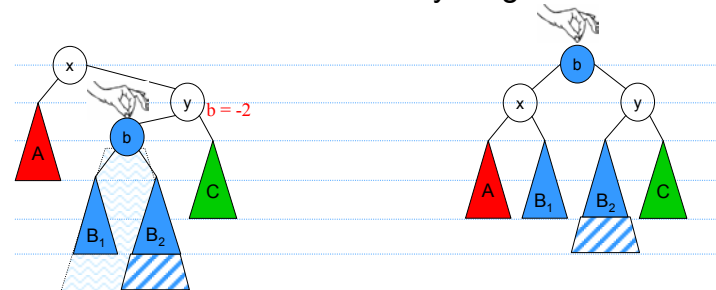
AVL-Eigenschaft verletzt
nach Einfügen links von y

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



Zum Experimentieren

Inserting 20. 20 inserted.

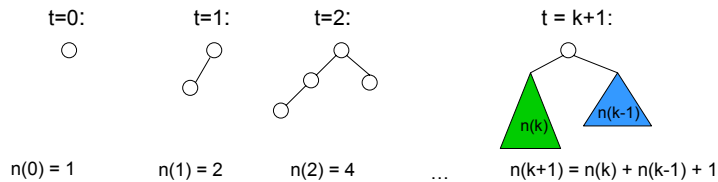
- <http://www.seanet.com/users/arsen/avltree.html>



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 $2 \cdot \log_2 N$
- Korollar: Suchen, einfügen und entfernen im AVL-Baum ist $O(\log(N))$

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



Beweis des Satzes

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

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

$$n(0) = 1, n(1) = 2,$$

$$n(t) = n(t-1) + n(t-2) + 1 > 2 \times n(t-2) \quad \text{relativ grobe Abschätzung)*}$$

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:

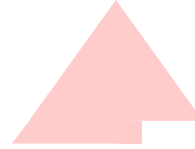
$$k + 1 < \log_2 n(t), \text{ also } \underline{k + 1} < \underline{2k + 1} < \underline{2 \times \log_2 n(t)}$$

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

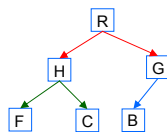


Vollständige Bäume

- 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: $B[0], B[1], B[2] \dots$
 - $B[i]$ hat Vater : $B[(i - 1) / 2]$
 - $B[i]$ hat Söhne : $B[2*i+1]$ und $B[2*i+2]$

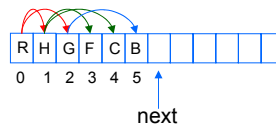


Vollständiger Baum ...



Prakt. Informatik II

... als Array A:

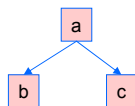


© H. Peter Gumm, Philipps-Universität Marburg



Heaps

- Ein 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

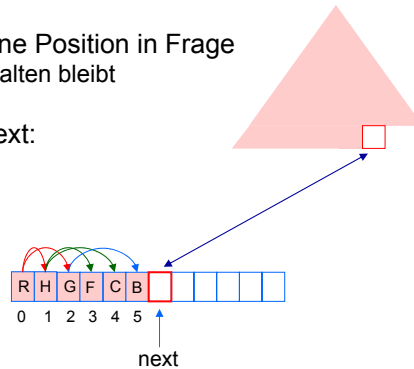
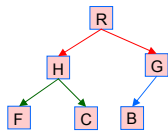
Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



insertHeap

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

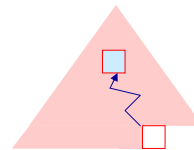


- allerdings kann dabei die Ordnungseigenschaft verloren gehen
 - wir korrigieren dies mit *aufsteigen* (engl.: *upHeap*)



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“



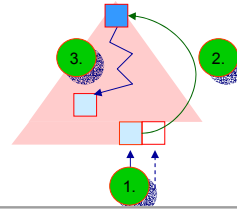
```
public void insert(int n) {
    theHeap[nextPos]=n;
    aufsteigen(nextPos);
    nextPos++;
}

/** 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



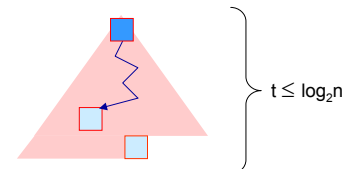
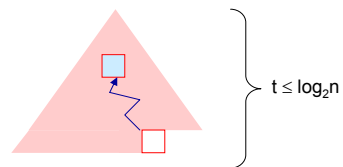
```
public int getNext(){
    int result = theHeap[0];
    nextPos--;
    theHeap[0] = theHeap[nextPos];
    absickern(0);
    return result;
}
```

```
/** 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 $N \geq 1 + 2 + 2^2 + 2^3 + \dots + 2^{t-1} + 1 = 2^t$ 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 $O(\log(n))$



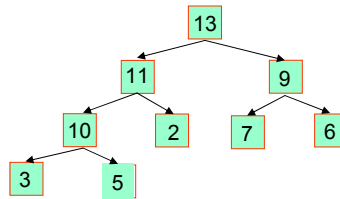


Sortieren mit einem Heap

1. Gegeben: Folge von n Elementen



2. Füge sie in einen Heap ein

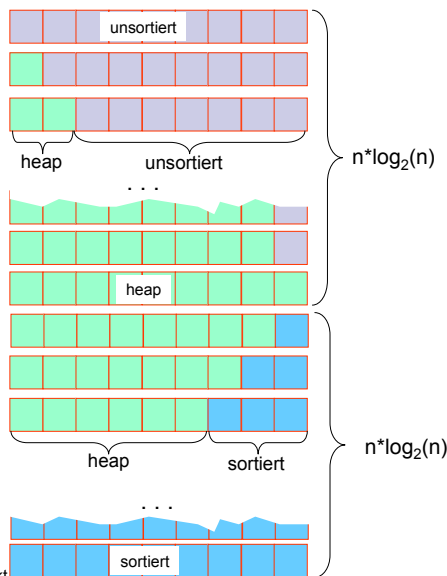


3. Entnehme die Elemente aus dem Heap

- 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
- $n \cdot O(\text{insert}) = O(n \cdot \log(n))$

■ 2. Phase

- entnehme die Elemente aus dem Heap und schreibe sie in den Endabschnitt
- $n \cdot O(\text{getNext}) = O(n \cdot \log(n))$

■ Komplexität also:

- $2 \cdot O(n \cdot \log(n)) = O(n \cdot \log(n))$



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



Bäume mit variabler Anzahl von Teilbäumen

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

