



Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Effiziente Algorithmen und
wissensbasierte Systeme



Algorithmen und Datenstrukturen

Wintersemester 2018/19
14. Vorlesung

Rot-Schwarz-Bäume

Dynamische Menge

verwaltet Elemente einer
sich ändernden Menge M



Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code> <code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	<p>} Änderungen</p> <p>} Anfragen</p>

Implementierung: je nachdem...

Binäre Suchbäume

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $O(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

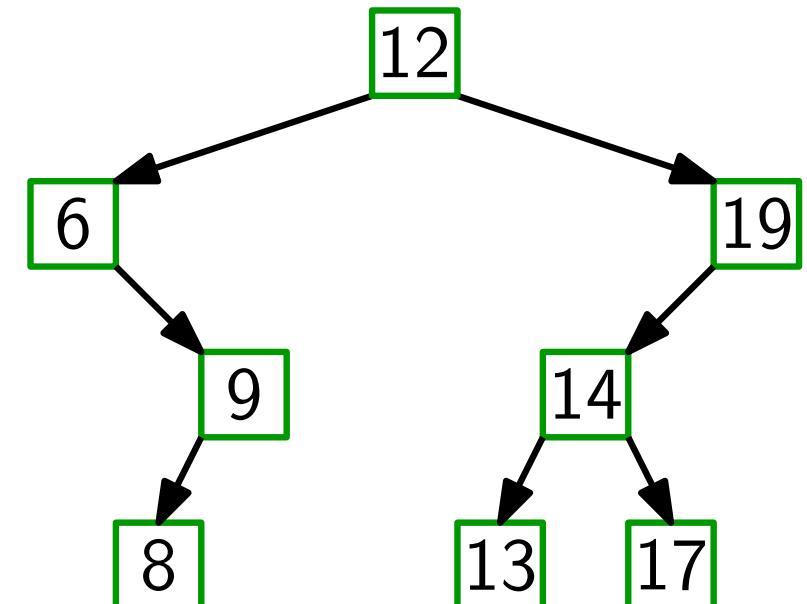
Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume *balancieren!*
 $\Rightarrow h \in O(\log n)$

Binärer-Suchbaum-Eigenschaft:

Für jeden Knoten v gilt:

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
 alle Knoten im rechten Teilbaum von v haben Schlüssel $\geq v.key$



Balanciermethoden

Beispiele

nach **Gewicht**

$BB[\alpha]$ -Bäume

für jeden Knoten ist das Gewicht (= Anzahl der Knoten) von linkem u. rechtem Teilbaum ungefähr gleich.

nach **Höhe**

AVL-Bäume*

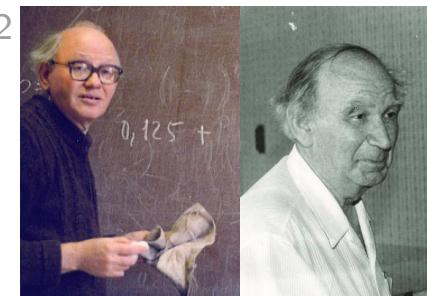
für jeden Knoten ist die Höhe von linkem und rechtem Teilbaum ungefähr gleich.



nach **Grad**

(2, 3)-Bäume

alle Blätter haben dieselbe Tiefe, aber innere Knoten können verschieden viele Kinder haben.



nach **Knotenfarbe**

Rot-Schwarz-Bäume

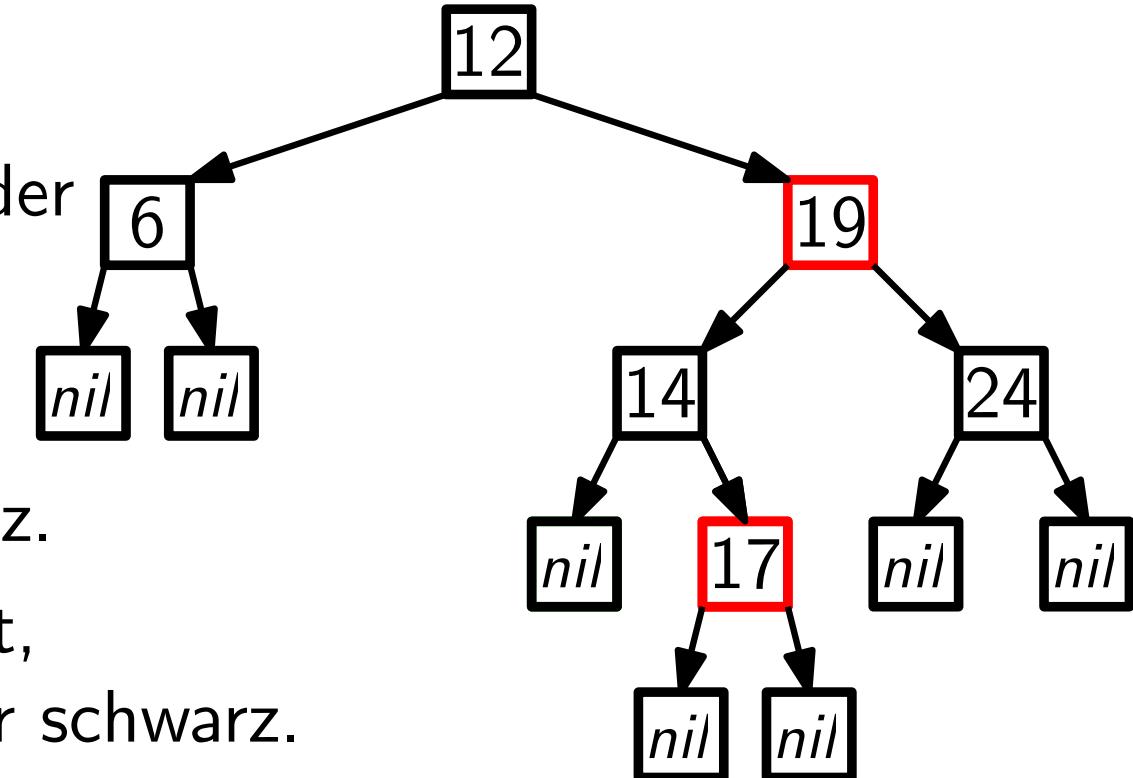
jeder Knoten ist entw. „gut“ oder „schlecht“; der Anteil schlechter Knoten darf in keinem Teilbaum zu groß sein.

Rot-Schwarz-Bäume: Eigenschaften

Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden *Rot-Schwarz-Eigenschaften*:

- (E1) Jeder Knoten ist entweder rot oder schwarz.
- (E2) Die Wurzel ist schwarz.
- (E3) Alle Blätter sind schwarz.
- (E4) Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.
- (E5) Für jeden Teilbaum gilt: alle Wurzel-Blatt-Pfade enthalten dieselbe Anzahl schwarzer Knoten.

Aus (E4) folgt: Auf keinem Wurzel-Blatt-Pfad folgen zwei rote Knoten direkt aufeinander.



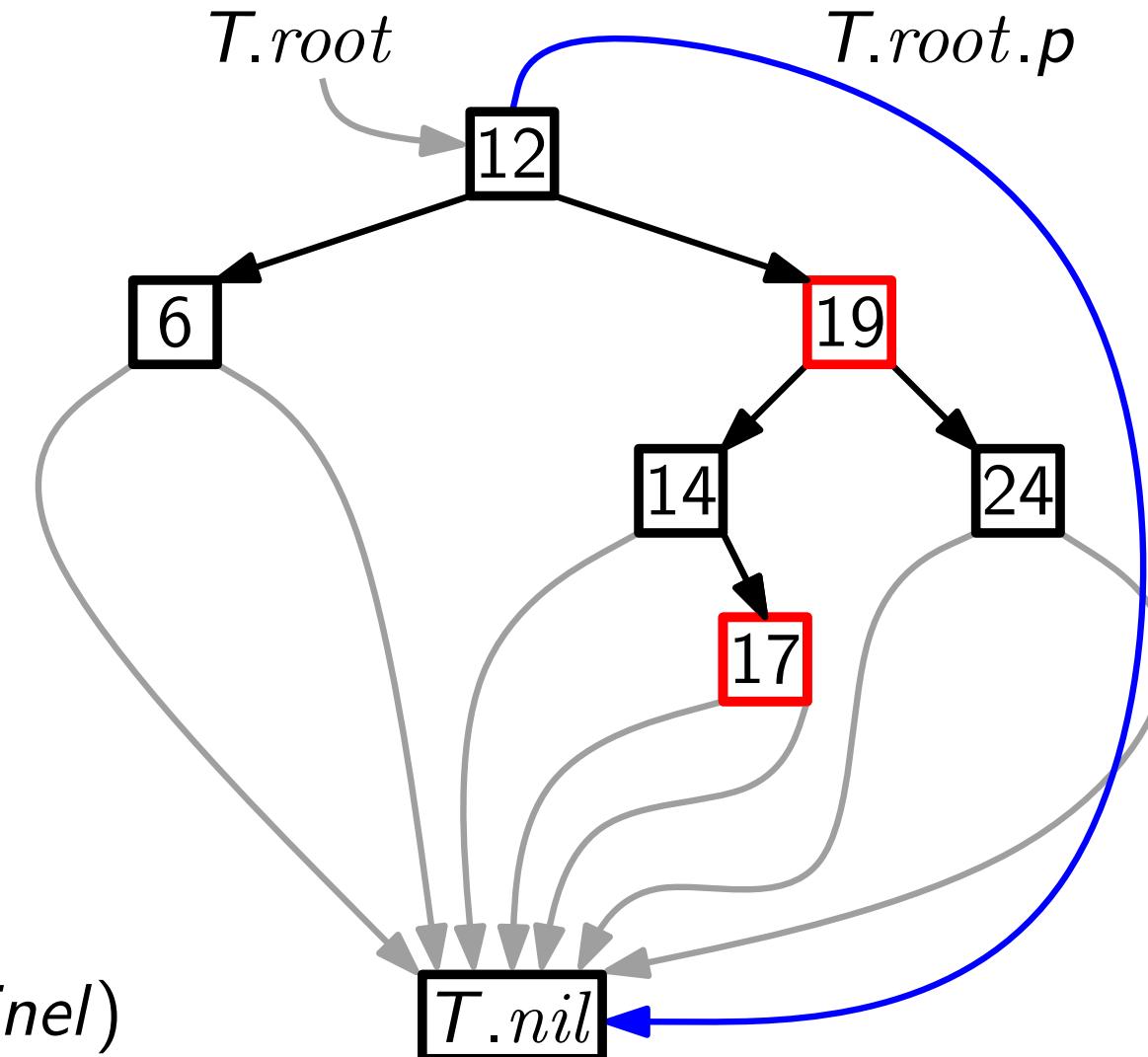
Technisches Detail

<i>Node</i>
<i>Key key</i>
<i>Node left</i>
<i>Node right</i>
<i>Node p</i>

<i>RBNode</i>
<i>Color color</i>

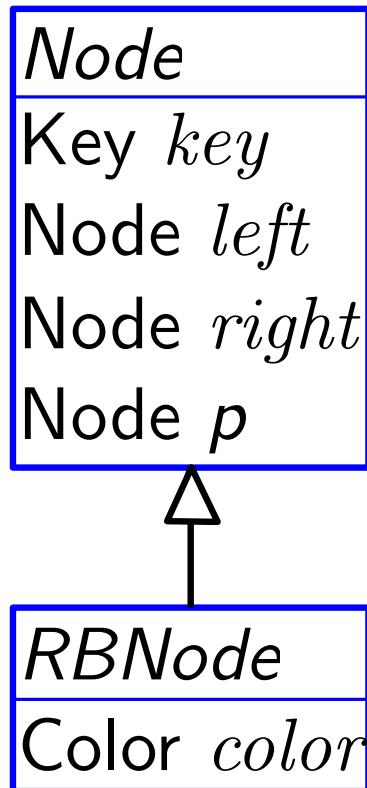
$T.root$, $T.nil$

Dummy-Knoten (engl. *sentinel*)

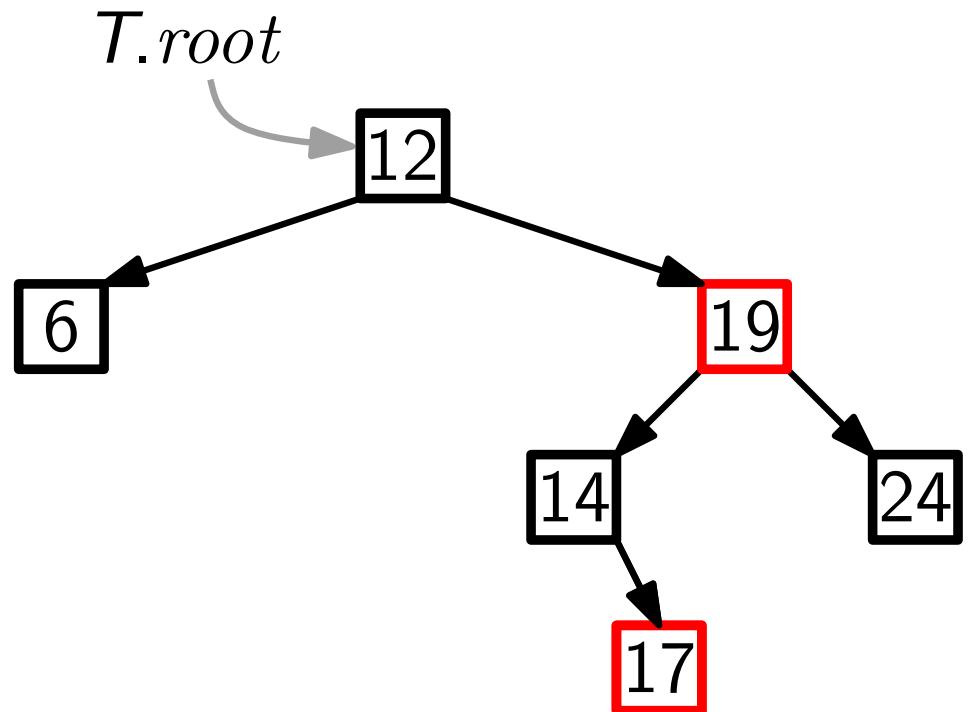


Zweck: um Baum-Operationen prägnanter aufzuschreiben zu können. (Wir zeichnen den Dummy-Knoten i.A. nicht.)

Technisches Detail

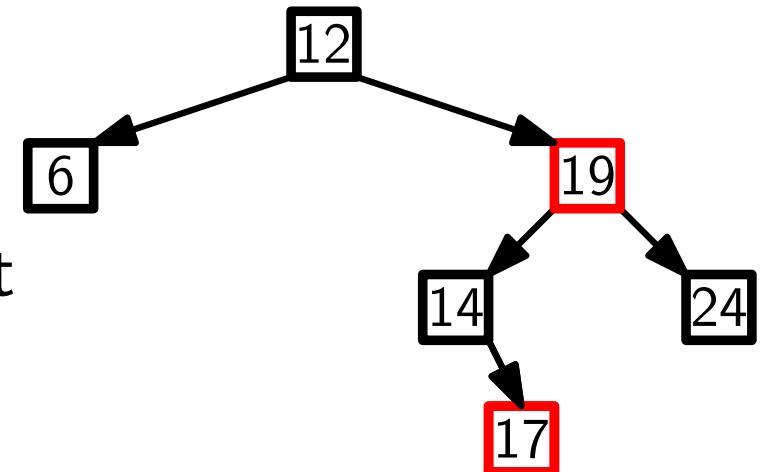


$T.root$, $T.nil$



Zweck: um Baum-Operationen prägnanter aufzuschreiben zu können. (Wir zeichnen den Dummy-Knoten i.A. nicht.)

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

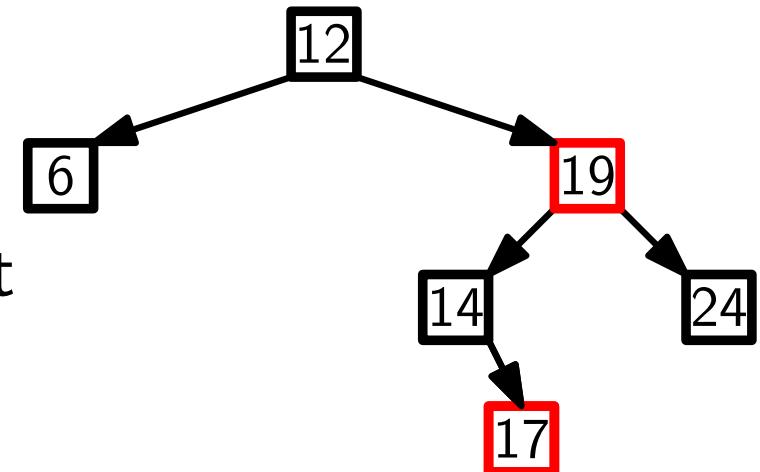
Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition! Die *Höhe* $\text{Höhe}(v)$ eines Knotens v ist die Anz. der Knoten (ohne v) auf dem längsten Pfad zu einem Blatt (inkl. Blatt) in B_v .

Beispiel: „12“ hat Höhe

(Schwarz-) Höhe



Definition: Die *Länge* eines Pfades ist die Anz. seiner Kanten.

Definition: Sei B ein Baum.

Knoten u ist *unter* Knoten v , wenn u in dem Teilbaum B_v von B mit Wurzel v enthalten ist.

Beispiel: „17“ ist unter „19“, „14“ ist *nicht* unter „6“.

Definition: Die *Höhe* eines Knotens v ist die Länge eines längsten Pfads von v zu einem Blatt unter v .

Definition: Die **Schwarz-Höhe** $s\text{Höhe}(v)$ eines Knotens v ist die Anz. der **schwarzen** Knoten (ohne v) auf **jedem** **längsten** Pfad zu einem Blatt (inkl. Blatt) in B_v .

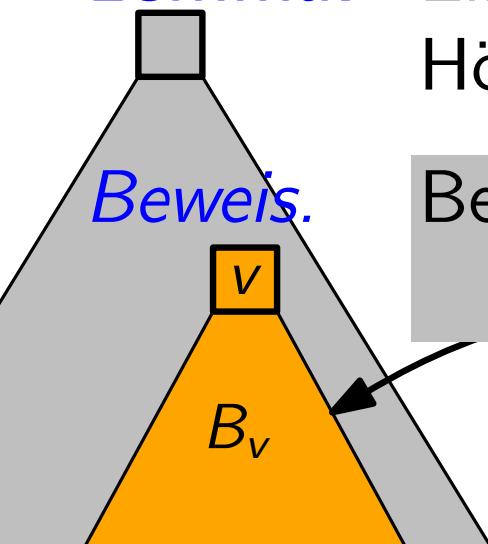
wohl-definiert
wg. (E5)

Beispiel: „12“ hat Höhe 4 (!) und Schwarz-Höhe 2.

Folgerung: \forall Knoten $\Rightarrow s\text{Höhe}(v) \leq \text{Höhe}(v) \leq 2 \cdot s\text{Höhe}(v)$.

Höhe $\in \Theta(\log n)!!$

Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.



Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{\text{Höhe}(v)} - 1$ innere Knoten.

Höhe $\in \Theta(\log n)!!$

Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis. Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{s\text{Höhe}(v)} - 1$ innere Knoten.

Beweis durch vollständige Induktion über Höhe(v).

Höhe(v) = 0. Dann $B_v = B.\text{nil}$ und $s\text{Höhe}(v) = 0$.
 B_v hat $0 = 2^0 - 1$ innere Knoten. 

Höhe(v) > 0. Beide Kinder von v haben Höhe < Höhe(v).
 \Rightarrow können Ind.-Annahme anwenden.

\Rightarrow # innere Knoten von B_v ist mind.

$$2 \cdot (2^{s\text{Höhe}(v)-1} - 1) + 1 = 2^{s\text{Höhe}(v)} - 1.$$

$s\text{Höhe}$ der Kinder von v ist mind.


 Anz. innerer Knoten unter
einem Kind von v 

Höhe $\in \Theta(\log n)!!$

Lemma. Ein Rot-Schwarz-Baum B mit n inneren Knoten hat Höhe $\leq 2 \log_2(n + 1)$.

Beweis. Behauptung: Für jeden Knoten v von B gilt:
 B_v hat $\geq 2^{\text{sHöhe}(v)} - 1$ innere Knoten.

$$v := B.\text{root} \Rightarrow \underbrace{\# \text{ innere Knoten}(B)}_n \geq 2^{\text{sHöhe}(B)} - 1.$$

$$\Rightarrow \text{sHöhe}(B) \leq \log_2(n + 1)$$

Wegen R-S-Eig. (E4) gilt: Höhe(B) $\leq 2 \cdot \text{sHöhe}(B)$.

$$\Rightarrow \text{Höhe}(B) \leq 2 \log_2(n + 1) \quad \square$$

Also: Rot-Schwarz-Bäume sind *balanciert!* Fertig?!
 Nee: Insert & Delete können R-S-Eig. *verletzen!*

Einfügen

Node Insert(key k)

$y = \text{nil}$

$x = \text{root}$

while $x \neq \text{nil}$ **do**

$y = x$

if $k < x.\text{key}$ **then**

$x = x.\text{left}$

else $x = x.\text{right}$

$z = \text{new Node}(k, y)$

if $y == \text{nil}$ **then** $\text{root} = z$

else

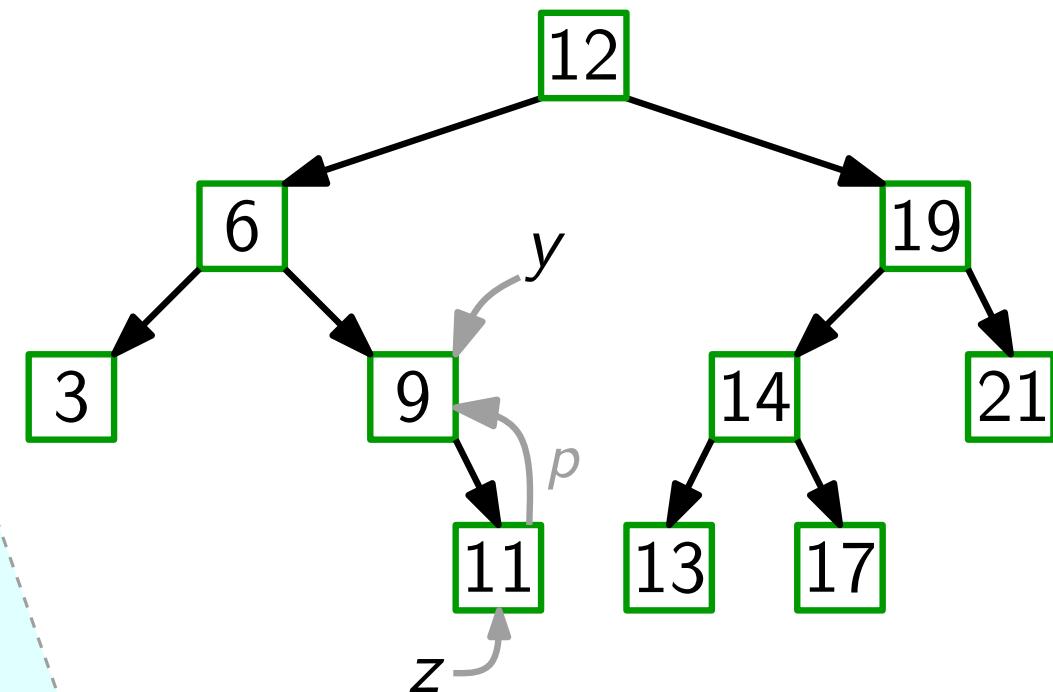
if $k < y.\text{key}$ **then** $y.\text{left} = z$

else

return z

Insert(11)

$x == \text{nil}$



Node(Key k, Node par)
 $\text{key} = k$
 $p = \text{par}$
 $\text{right} = \text{left} = \text{nil}$

Einfügen

Laufzeit? (ohne RBInsertFixup) $O(h) = O(\log n)$

RB .. RB

Node Insert(key k)

$y = T.nil$

$x = root$

while $x \neq T.nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

RB

$z = \text{new Node}(k, y, red)$

if $y == T.nil$ **then** $root = z$

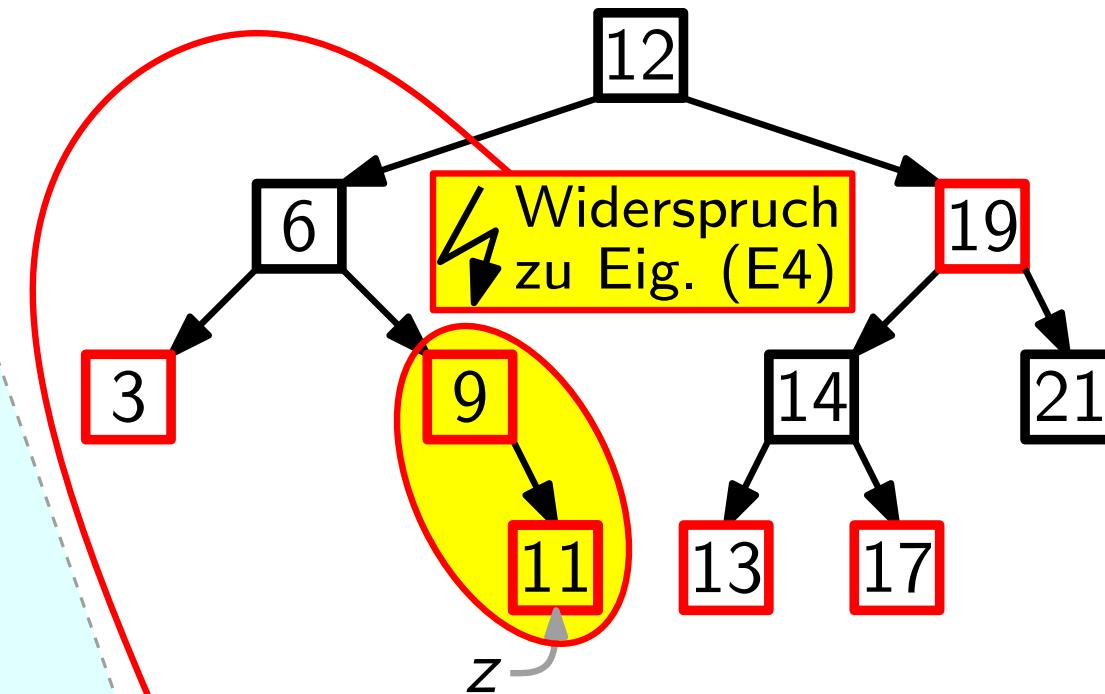
else

if $k < y.key$ **then** $y.left = z$

else

RBInsertFixup(z)

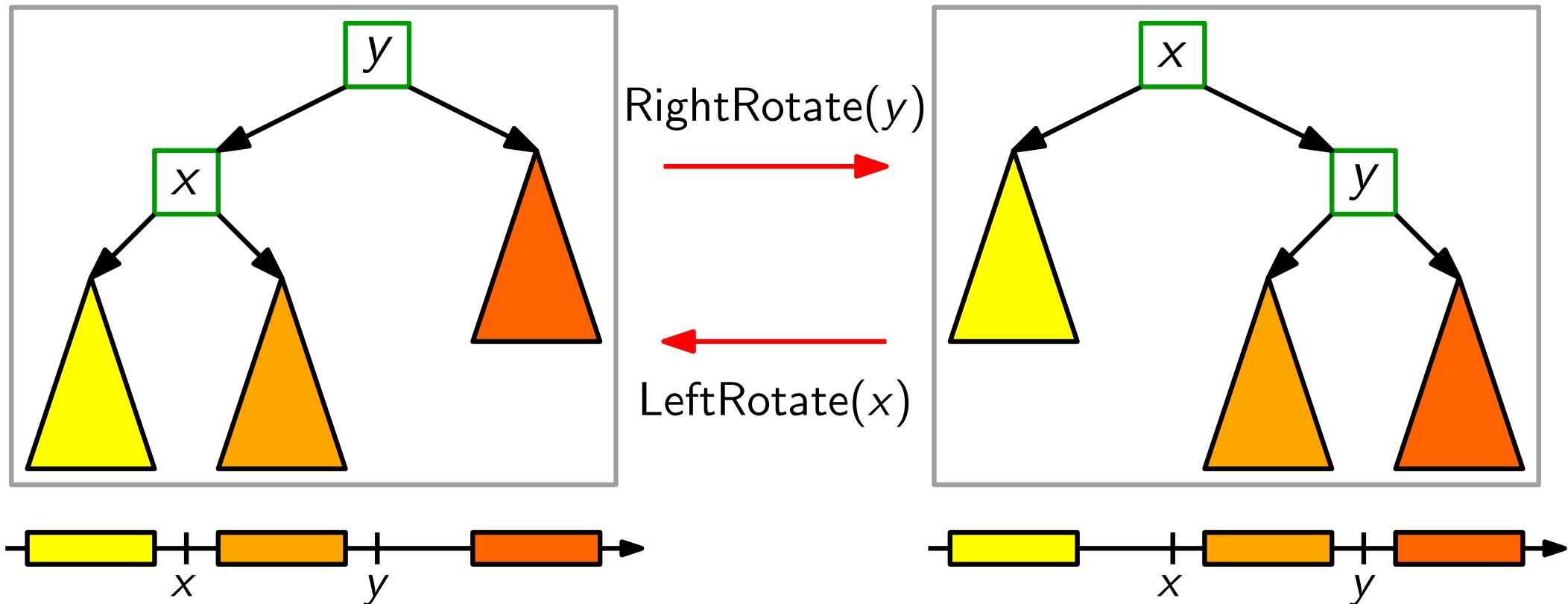
return z



Node(Key k, Node par)
 $key = k$
 $p = par$
 $right = left = T.nil$

RBNode(..., Color c)
 $super(k, par)$
 $color = c$

Exkurs: Rotationen



Also: Binärer-Suchbaum-Eig. bleibt beim Rotieren erhalten!

Aufgabe: Schreiben Sie Pseudocode für LeftRotate(x)!

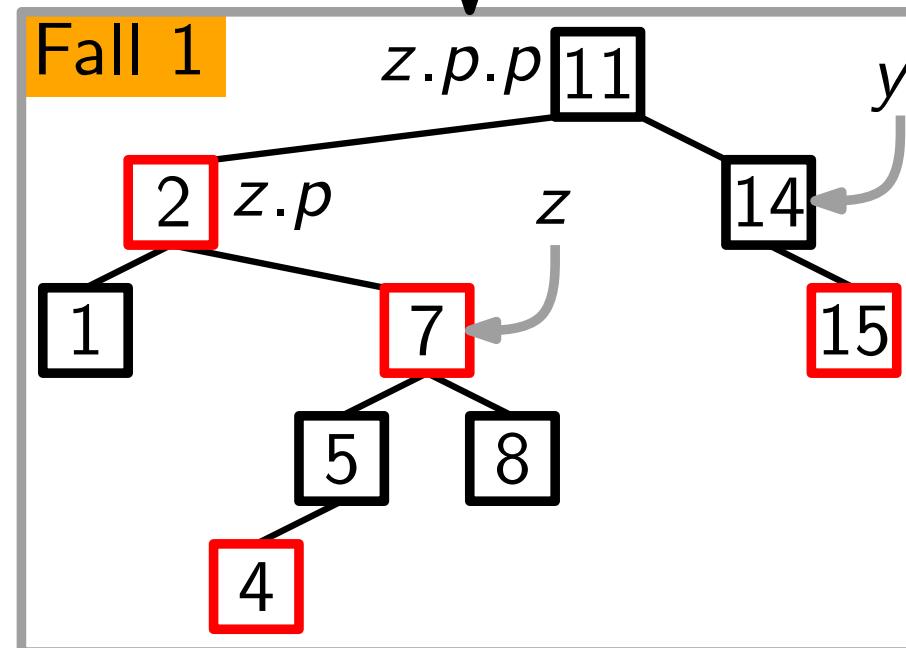
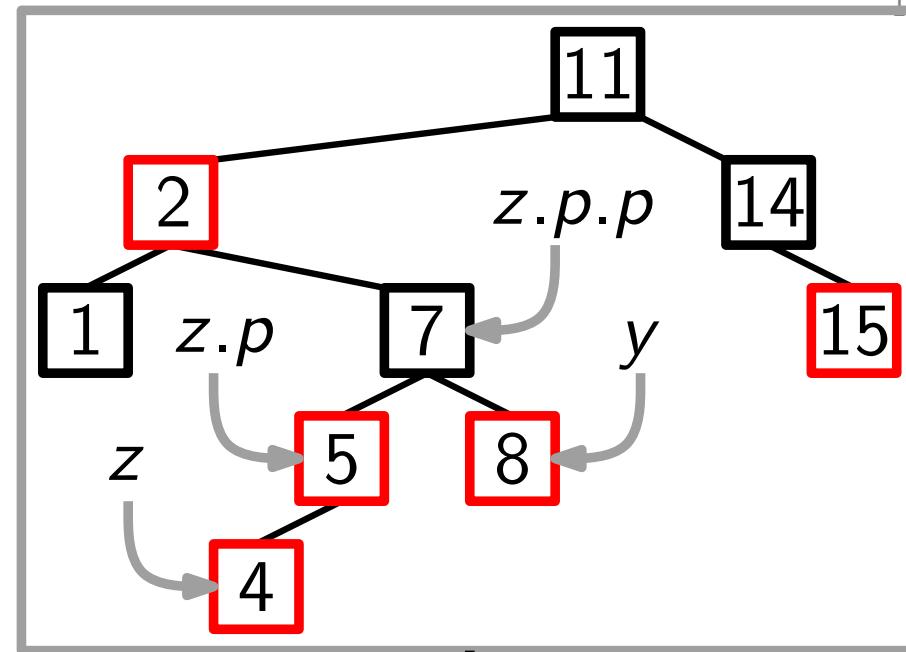
Laufzeit: $O(1)$.



RBIInsertFixup(Node z)

```

while  $z.p.color == \text{red}$  do
    if  $z.p == z.p.p.left$  then
         $y = z.p.p.right$  // Tante von z
        if  $y.color == \text{red}$  then
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
             $y.color = \text{black}$ 
             $z = z.p.p$ 
        else
            if  $z == z.p.right$  then
                 $z = z.p$ 
                LeftRotate( $z$ )
                 $z.p.color = \text{black}$ 
                 $z.p.p.color = \text{red}$ 
                RightRotate( $z.p.p$ )
            else ... // wie oben, aber re. & li. vertauscht
             $root.color = \text{black}$ 
    
```



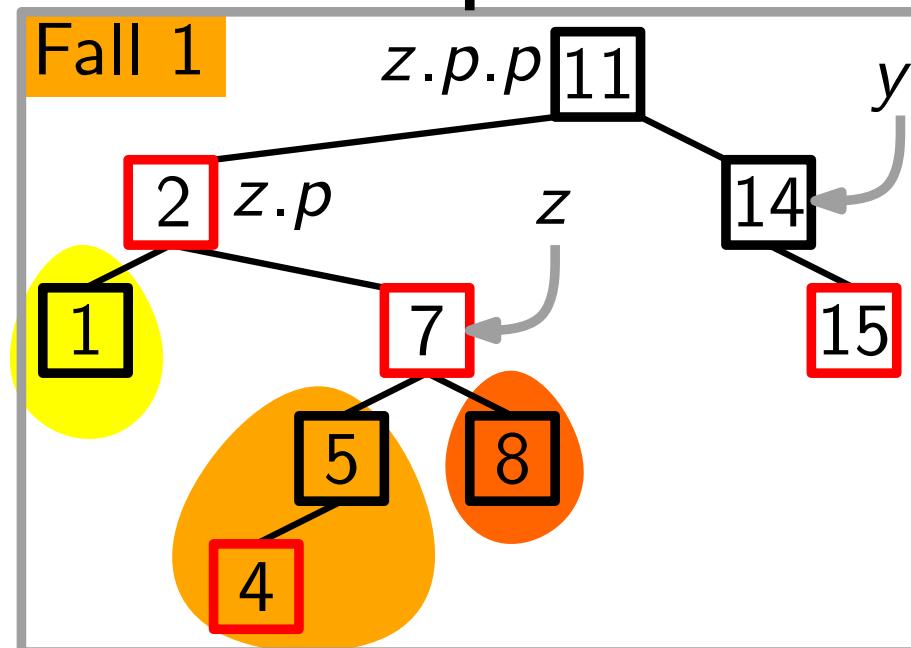
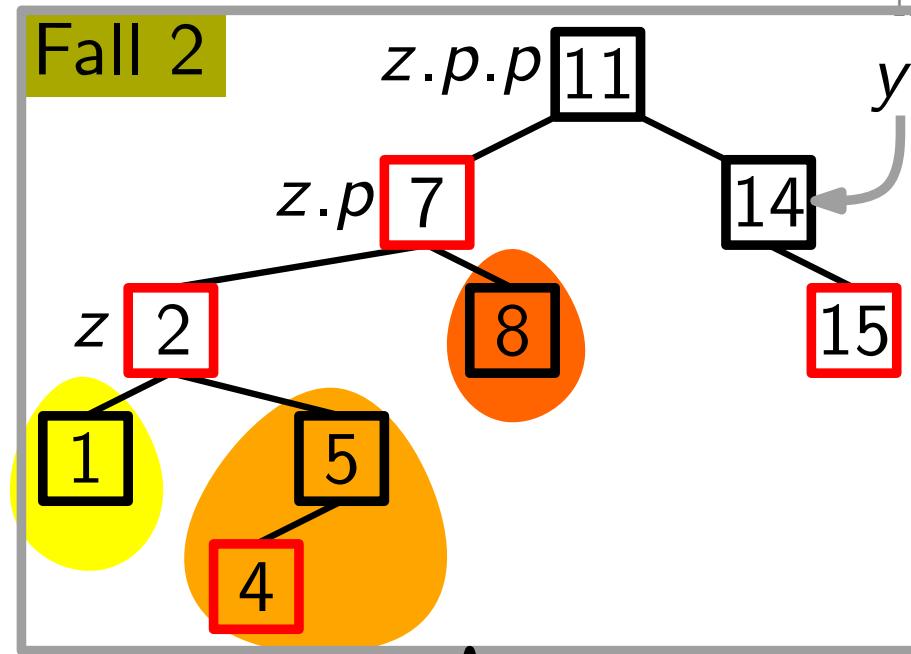
li. vertauscht

RBIInsertFixup(Node z)

```

while  $z.p.color == \text{red}$  do
    if  $z.p == z.p.p.left$  then
         $y = z.p.p.right$  // Tante von z
        if  $y.color == \text{red}$  then
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
             $y.color = \text{black}$ 
             $z = z.p.p$ 
        else
            if  $z == z.p.right$  then
                 $z = z.p$ 
                LeftRotate( $z$ )
                 $z.p.color = \text{black}$ 
                 $z.p.p.color = \text{red}$ 
                RightRotate( $z.p.p$ )
            else ... // wie oben, aber re. & li. vertauscht
     $root.color = \text{black}$ 

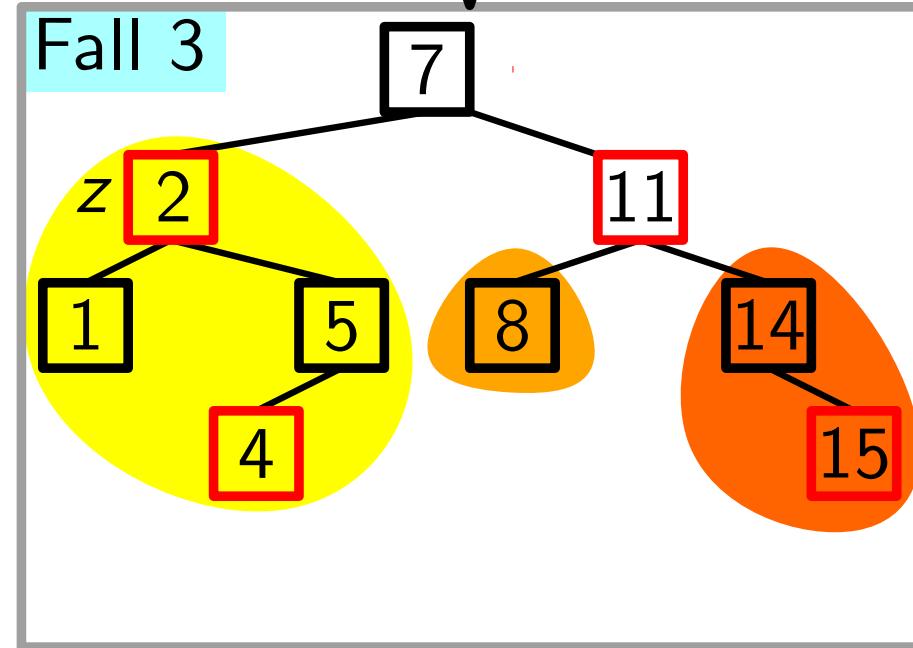
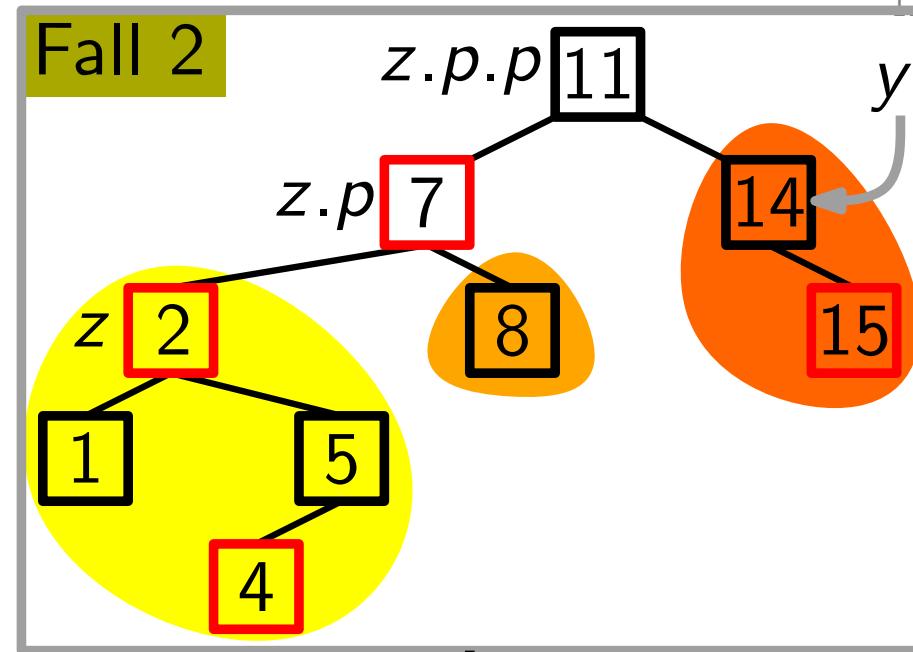
```



li. vertauscht

RBIInsertFixup(Node z)

```
while z.p.color == red do
    if z.p == z.p.p.left then
        y = z.p.p.right // Tante von z
        if y.color == red then
            z.p.color = black
            z.p.p.color = red
            y.color = black
            z = z.p.p
        else
            if z == z.p.right then
                z = z.p
                LeftRotate(z)
                z.p.color = black
                z.p.p.color = red
                RightRotate(z.p.p)
            else ... // wie oben, aber re. & li. vertauscht
    root.color = black
```



li. vertauscht

Korrektheit

Zu zeigen: RBInsertFixup stellt R-S-Eigenschaft wieder her.

Schleifeninvariante (gültig am Anfang der while-Schleife)

- z ist rot.
- Falls $z.p$ die Wurzel ist, dann ist $z.p$ schwarz.
- Falls R-S-Eig. verletzt sind, dann entweder (E2) oder (E4).
 - Falls (E2) verletzt ist, dann weil $z = \text{root}$ und z rot ist.
 - Falls (E4) verletzt ist, dann weil z und $z.p$ rot sind.

Zeige:

- Initialisierung
- Aufrechterhaltung
- Terminierung

Viel Arbeit! Siehe [CLRS, Kapitel 13.3].

Laufzeit RBInsertFixup

```

while  $z.p.color == \text{red}$  do
    if  $z.p == z.p.p.left$  then
         $y = z.p.p.right$ 
        if  $y.color == \text{red}$  then
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
             $y.color = \text{black}$ 
             $z = z.p.p$ 
        }  $O(1)$ 
    else
        if  $z == z.p.right$  then
             $z = z.p$ 
            LeftRotate( $z$ )
             $z.p.color = \text{black}$ 
             $z.p.p.color = \text{red}$ 
            RightRotate( $z.p.p$ )
        }  $O(1)$ 
    }  $O(1)$ 
else ... // wie oben, aber re. & li. vertauscht
root.color = black

```

Insgesamt:

- Fall 1 $O(h)$ mal
- Fall 2 ≤ 1 mal
- Fall 3 ≤ 1 mal

$O(\log n)$ Umfärbungen und ≤ 2 Rotationen

Klettert Baum zwei Ebenen nach oben.

Führt zum Abbruch der while-Schleife.

Löschen in (farblosen) binären Suchbäumen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

- 1.** z hat kein li. Kind.

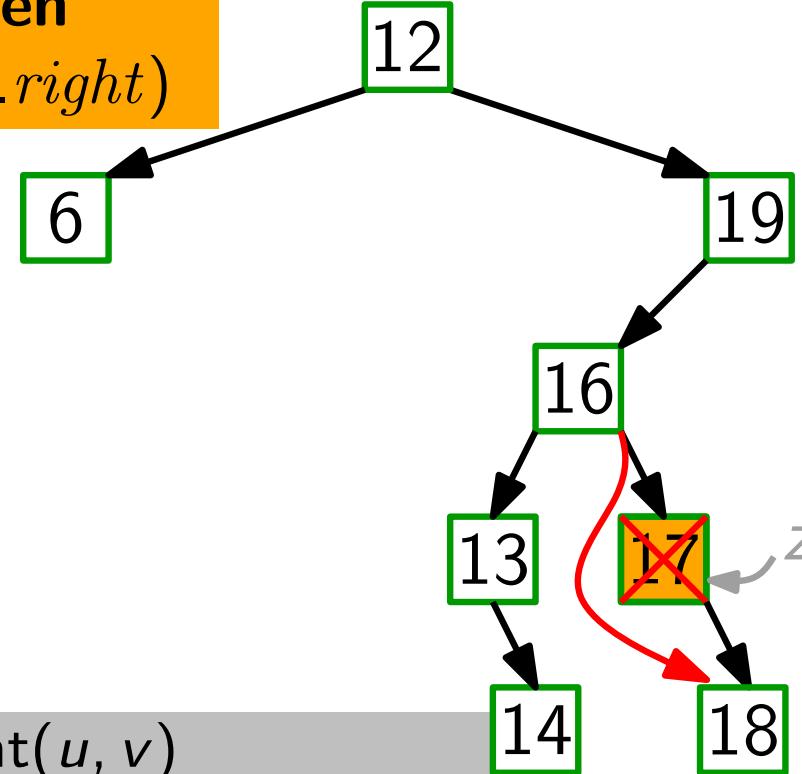
```
if  $z.left == \text{nil}$  then  
    Transplant( $z, z.right$ )
```

Setze $z.right$ an die Stelle von z .

Lösche z .

- 2.** z hat kein re. Kind.

- 3.** z hat zwei Kinder.



Transplant(u, v)

```
if  $u.p == \text{nil}$  then  $\text{root} = v$   
else  
    if  $u == u.p.left$  then  
         $u.p.left = v$   
    else  $u.p.right = v$ 
```

if $v \neq \text{nil}$ then $v.p = u.p$

Setze v
an die
Stelle
von u .

Löschen in (farblosen) binären Suchbäumen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat kein li. Kind.

```
if  $z.left == nil$  then  
    Transplant( $z, z.right$ )
```

Setze $z.right$ an die Stelle von z .

Lösche z .

2. z hat kein re. Kind.

```
else if  $z.right == nil$  then  
    Transplant( $z, z.left$ )
```

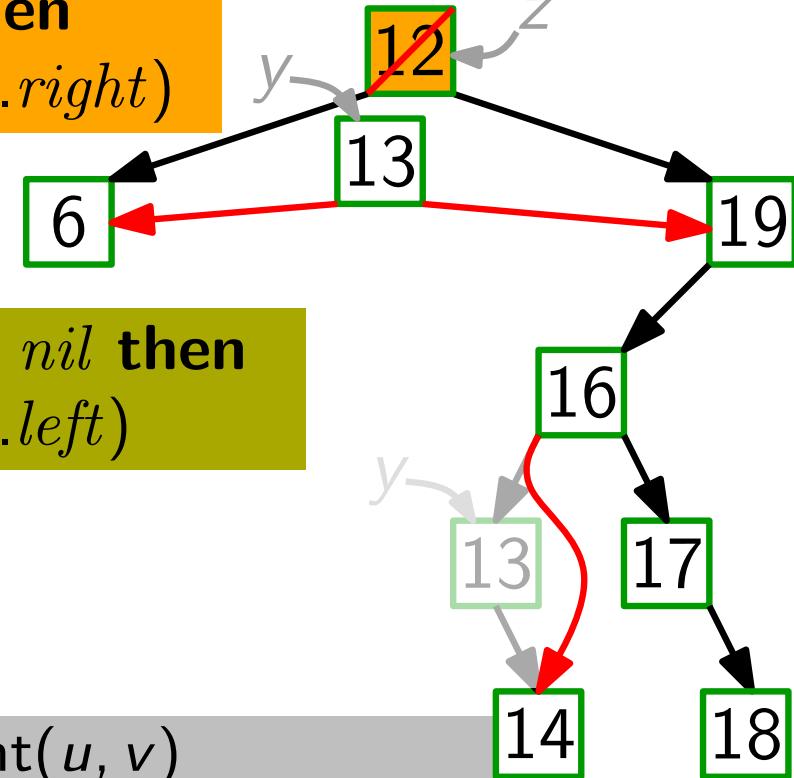
symmetrisch!

3. z hat zwei Kinder.

Setze $y = \text{Successor}(z)$

Falls $y.p \neq z$, setze $y.right$ an die Stelle von y .

Setze y an die Stelle von z



Transplant(u, v)

```
if  $u.p == nil$  then  $root = v$   
else  
    if  $u == u.p.left$  then  
         $u.p.left = v$   
    else  $u.p.right = v$ 
```

```
if  $v \neq nil$  then  $v.p = u.p$ 
```

Setze v an die Stelle von u .

Löschen (Übersicht)

Delete(Node z)

```
if  $z.left == nil$  then // kein linkes Kind
```

```
  | Transplant( $z, z.right$ )
```

```
else
```

```
  if  $z.right == nil$  then // kein rechtes Kind
```

```
    | Transplant( $z, z.left$ )
```

```
  else // zwei Kinder
```

```
     $y = \text{Successor}(z)$ 
```

```
    if  $y.p \neq z$  then
```

```
      | Transplant( $y, y.right$ )
```

```
      |  $y.right = z.right$ 
```

```
      | |  $y.right.p = y$ 
```

```
      | Transplant( $z, y$ )
```

```
      |  $y.left = z.left$ 
```

```
      | |  $y.left.p = y$ 
```

```

RBDelete(Node z)
y = z; origcolor = y.color
if z.left == T.nil then
    x = z.right
    RBTransplant(z, z.right)
else
    if z.right == T.nil then
        x = z.left
        RBTransplant(z, z.left)
    else
        y = Successor(z)
        origcolor = y.color
        x = y.right
        if y.p == z then x.p = y
        else
            RBTransplant(y, y.right)
            y.right = z.right
            y.right.p = y
        RBTransplant(z, y)
        y.left = z.left
        y.left.p = y; y.color = z.color
    if origcolor == black then RBDeleteFixup(x)

```

- y** zeigt auf den Knoten, der entweder gelöscht oder verschoben wird.
- x** zeigt auf den Knoten, der die Stelle von *y* einnimmt – das ist entweder das einzige Kind von *y* oder *T.nil*.
- Falls *y* ursprünglich *rot* war, bleiben alle R-S-Eig. erhalten:
- Keine Schwarzhöhe hat sich verändert.
 - Keine zwei roten Knoten sind Nachbarn geworden.
 - *y* rot \Rightarrow *y* \neq Wurzel \Rightarrow Wurzel bleibt schwarz.

RBDeleteFixup

Was kann schief gehen, wenn y schwarz war?

- (E2) y war Wurzel, und ein rotes Kind von y wurde Wurzel.
- (E4) x und $x.p$ sind rot.
- (E5) Falls y verschoben wurde, haben jetzt alle Pfade, die vorher y enthielten, einen schwarzen Knoten zu wenig.

„Repariere“ Knoten x zählt eine schwarze Einheit extra
(E5): (ist also „rot-schwarz“ oder „doppelt schwarz“)

Ziel: Schiebe die überzählige schwarze Einheit nach oben, bis:

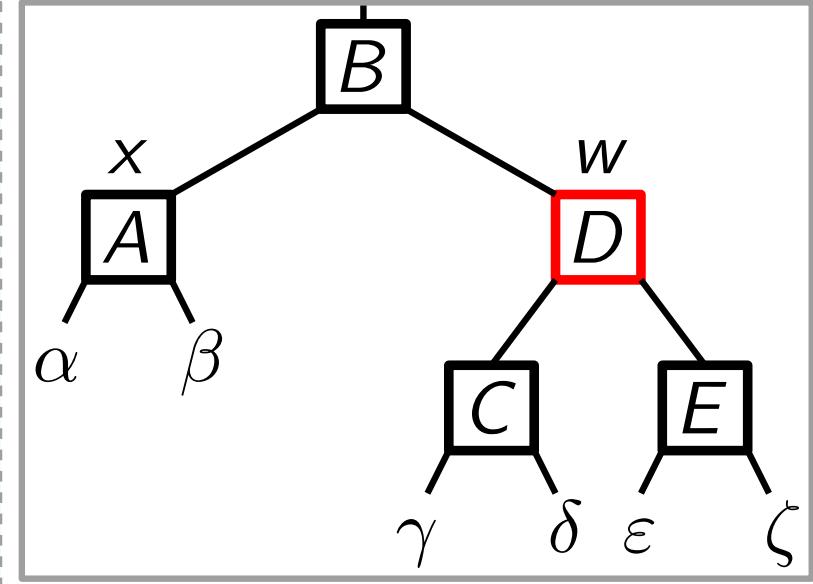
- x ist rot-schwarz \Rightarrow mach x schwarz.
- x ist Wurzel \Rightarrow schwarze Extra-Einheit verfällt.
- Problem wird lokal durch Umfärben & Rotieren gelöst.

RBDeleteFixup(RBNode x)

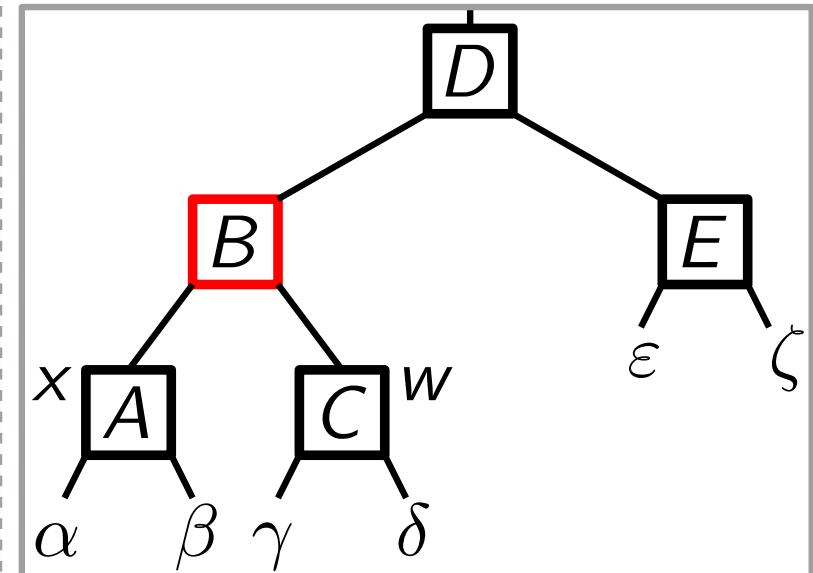
```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
    if  $x == x.p.\text{left}$  then
         $w = x.p.\text{right}$  // Schwester von  $x$ 
        if  $w.\text{color} == \text{red}$  then
             $w.\text{color} = \text{black}$ 
             $x.p.\text{color} = \text{red}$ 
            LeftRotate( $x.p$ )
             $w = x.p.\text{right}$ 
        if  $w.\text{left}.\text{color} == \text{black}$  and
             $w.\text{right}.\text{color} == \text{black}$  then
                 $w.\text{color} = \text{red}$ 
                 $x = x.p$ 
        else // kommt gleich!!
        else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
         $x.\text{color} = \text{black}$ 
    
```

Ziel:
 $w \rightarrow$ schwarz
ohne R-S-Eig.
zu verletzen.



Fall 1



RBDeleteFixup(RBNode x)

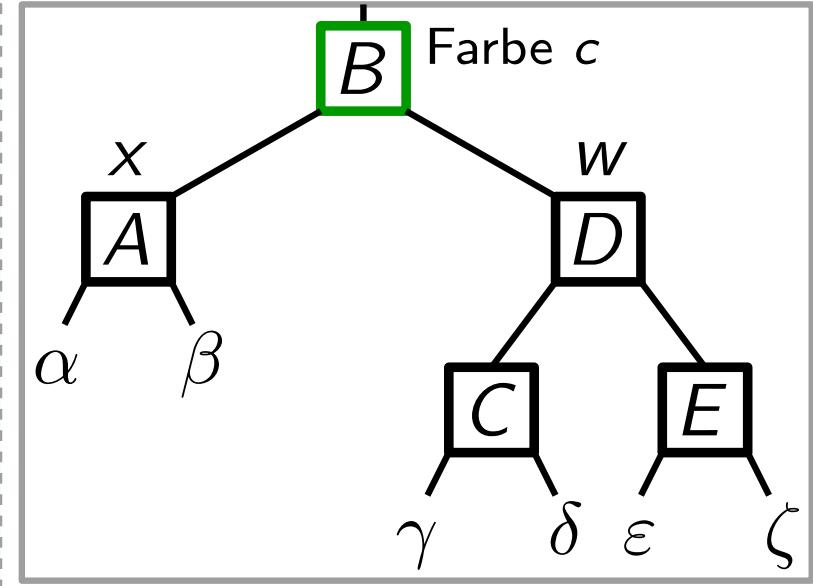
```

while  $x \neq \text{root}$  and  $x.\text{color} == \text{black}$  do
    if  $x == x.p.\text{left}$  then
         $w = x.p.\text{right}$  // Schwester von x
        if  $w.\text{color} == \text{red}$  then
             $w.\text{color} = \text{black}$ 
             $x.p.\text{color} = \text{red}$ 
            LeftRotate( $x.p$ )
             $w = x.p.\text{right}$ 
        if  $w.\text{left.color} == \text{black}$  and
             $w.\text{right.color} == \text{black}$  then
                 $w.\text{color} = \text{red}$ 
                 $x = x.p$ 
        else // kommt gleich!!
    else // wie oben; nur  $\text{left} \leftrightarrow \text{right}$ 
 $x.\text{color} = \text{black}$ 

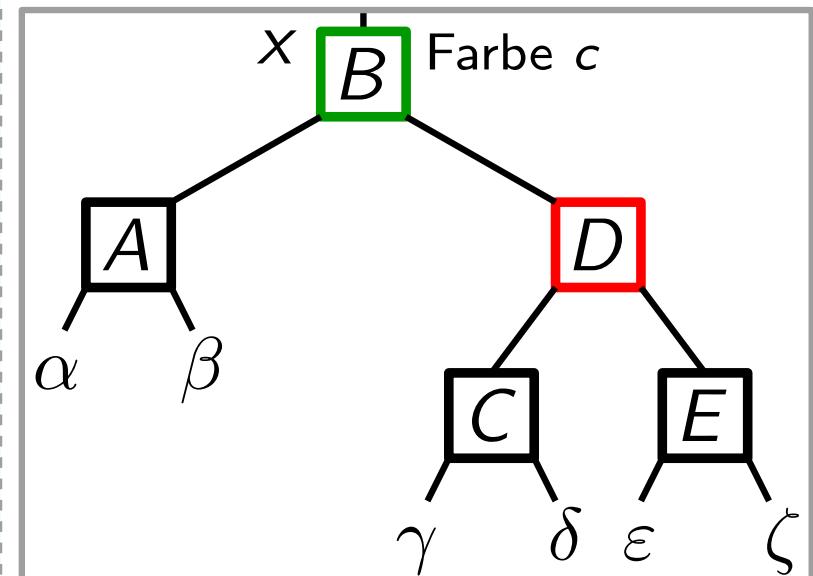
```

Ziel:
 $w \rightarrow$ schwarz
ohne R-S-Eig.
zu verletzen.

Schw. Einheit
raufschreiben.



Fall 2



Bem.: Anz. der schw. Knoten (inkl. Extra-Einh. bei x) bleibt auf allen Pfaden gleich!

RBDeleteFixup (Forts.)

else

if $w.right.color == black$ **then**

$w.left.color = black$

$w.color = red$

RightRotate(w)

$w = x.p.right$

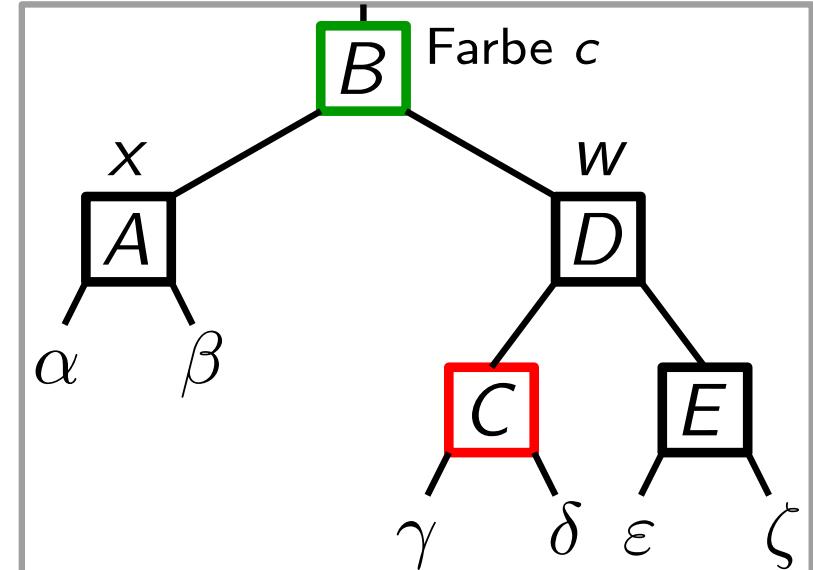
$w.color = x.p.color$

$x.p.color = black$

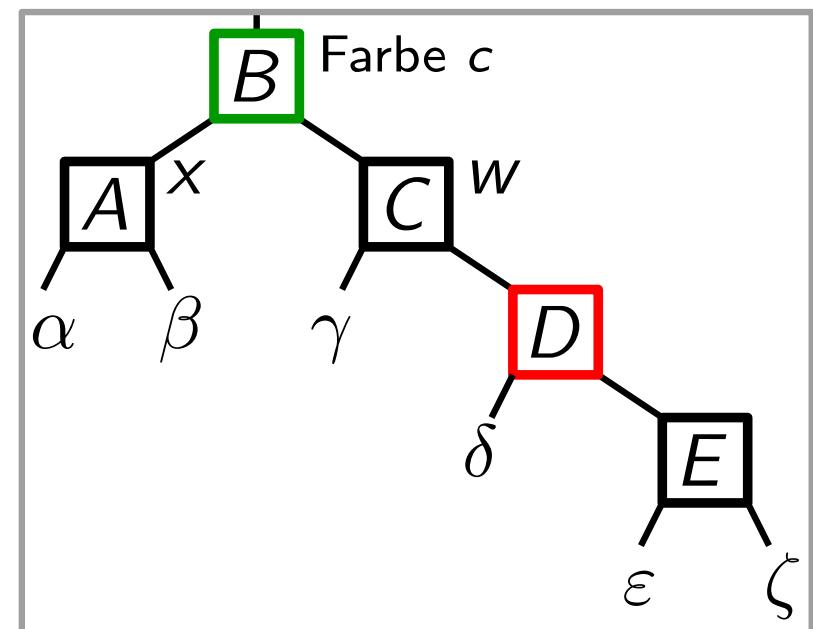
$w.right.color = black$

LeftRotate($x.p$)

$x = root$



Fall 3



RBDeleteFixup (Forts.)

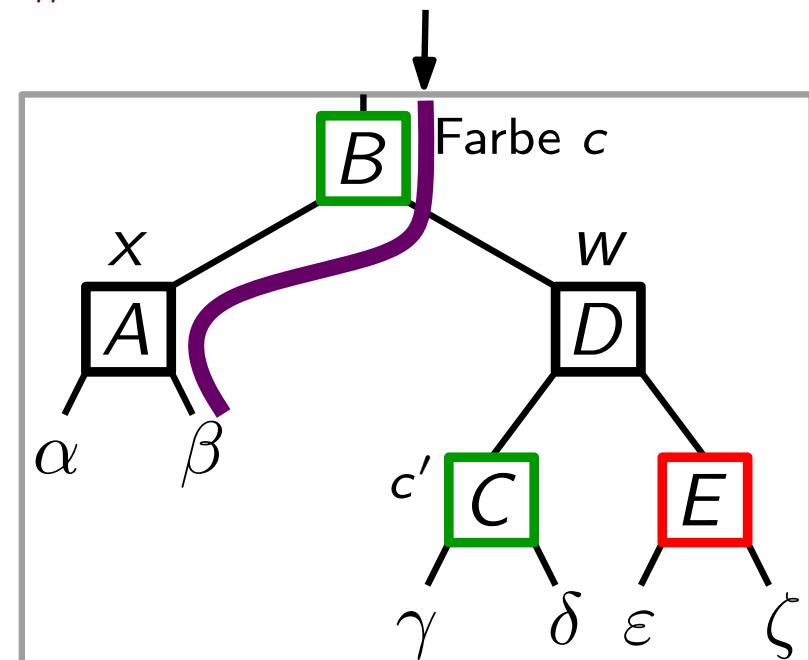
else

```
if w.right.color == black then
    w.left.color = black
    w.color = red
    RightRotate(w)
    w = x.p.right
```

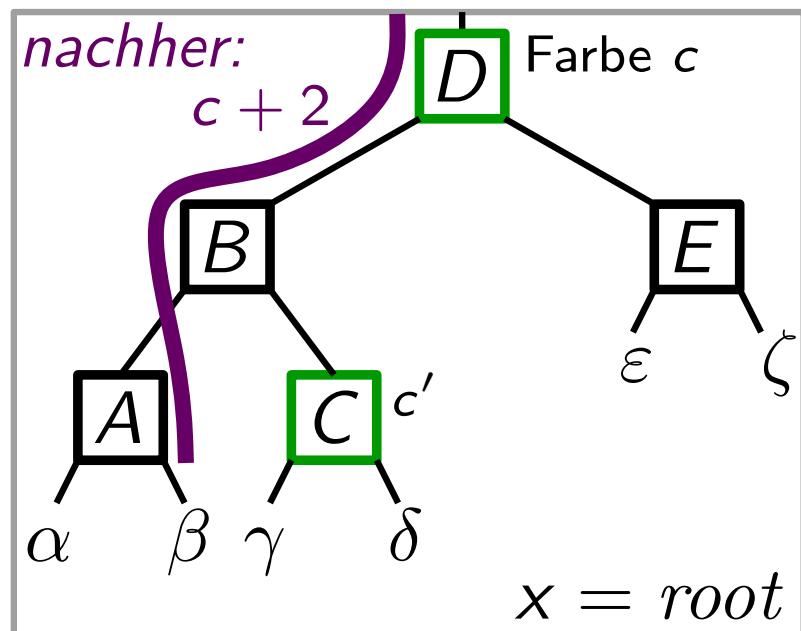
```
w.color = x.p.color
x.p.color = black
w.right.color = black
LeftRotate(x.p)
x = root
```

Bem.: Anz. der schwarzen Knoten
 (inkl. der Extra-Einheit bei x)
 bleibt auf allen Pfaden gleich!

vorher:
 # schwarze Einheiten = $c + 2$



Fall 4



RBDeleteFixup (Forts.)

else

```

if  $w.right.color == black$  then
     $w.left.color = black$ 
     $w.color = red$ 
    RightRotate( $w$ )
     $w = x.p.right$ 

```

```

 $w.color = x.p.color$ 
 $x.p.color = black$ 
 $w.right.color = black$ 
LeftRotate( $x.p$ )
 $x = root$ 

```

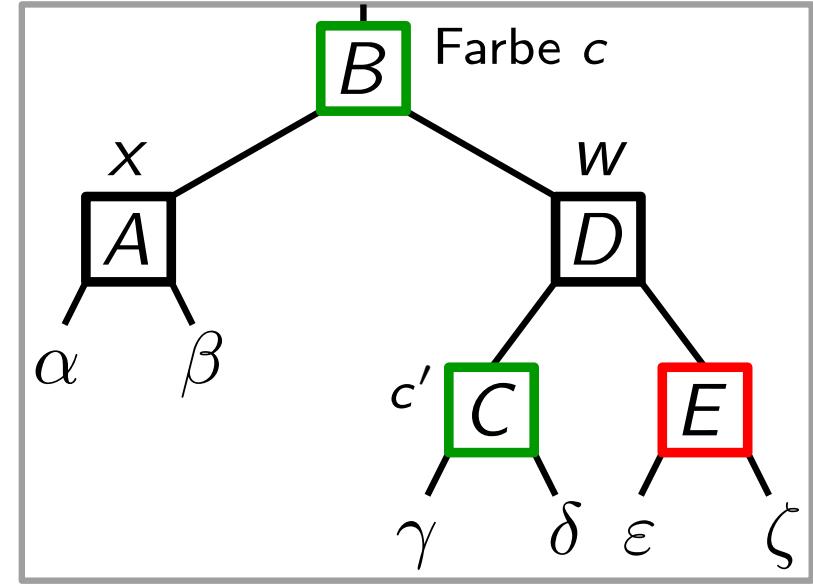
Laufzeit?

Fall 1: 1 Rotation + $O(1)$

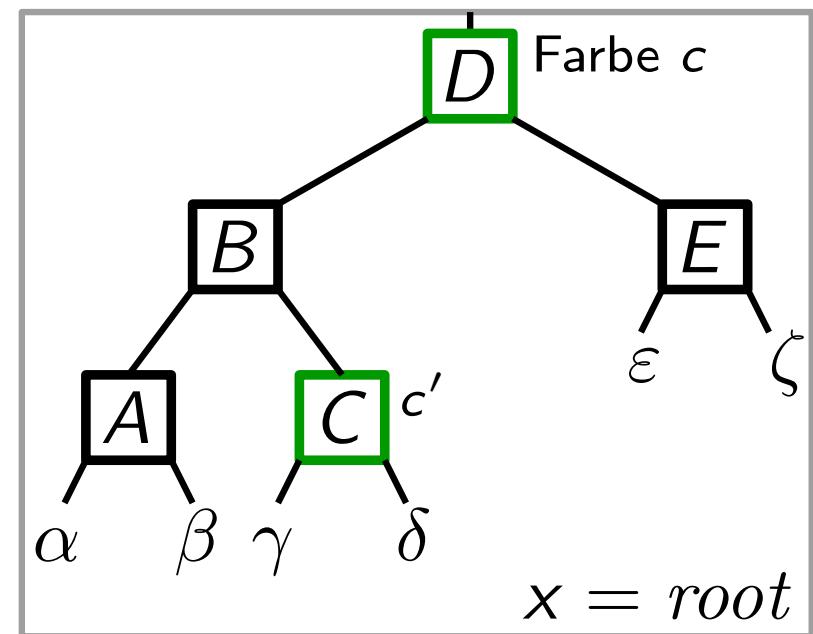
Fall 2: $O(h)$ Umfärbungen

Fall 3: 1 Rotation + $O(1)$

Fall 4: 1 Rotation + $O(1)$



Fall 4



$x = root$

Zusammenfassung

$$\text{Laufzeit RBDelete} \in O(h) + \underbrace{\text{Laufzeit RBDeleteFixup}}_{O(h)} = O(h)$$

RBDelete erhält die Rot-Schwarz-Eigenschaften.

Also gilt (siehe Lemma): $h \in O(\log n)$



Laufzeit RBDelete $\in O(\log n)$

Satz.

Rot-Schwarz-Bäume implementieren alle dynamische-Menge-Operationen in $O(\log n)$ Zeit, wobei n die momentane Anz. der Schlüssel ist.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
15. Vorlesung

Augmentieren von Datenstrukturen

Plan

Wir kennen schon eine ganze Reihe von Datenstrukturen:

- doppelt verkettete Liste
- Stapel
- Hashtabelle
- Heap
- binärer Suchbaum

Allerdings gibt es viele Situationen, wo keine davon **genau** passt.

Herangehensweise: *Augmentieren* von Datenstrukturen,
d.h. wir verändern Datenstrukturen,
indem wir extra Information
hinzufügen und aufrechterhalten.

Ein Beispiel

Bestimme für eine dynamische Menge v. Zahlen den Mittelwert.

1. Welche Ausgangsdatenstruktur?
 - Liste
2. Welche Extrainformation aufrechterhalten?
 - Summe der Elemente (*sum*)
 - Anzahl der Elemente (*size*)
3. Aufwand zur Aufrechterhaltung der Extrainformation?
 - konstanter Aufwand beim Einfügen und Löschen
4. Implementiere neue Operationen!

`getMean()`

`return sum/size`

Ähnlich für Standardabweichung $\sqrt{\frac{1}{n} \sum_{i=1}^n (a_i - \bar{a})^2}$.

Probieren Sie's!

Neues Beispiel

Wir wollen das Auswahlproblem für dynamische Mengen lösen.

D.h. wir wollen zu jedem Zeitpunkt effizient

- das i -kleinste Element (z.B. den Median): Elem `Select(int i)`
- den *Rang* eines Elements: int `Rank(Elem e)`

in einer dynamischen Menge bestimmen können.

Fahrplan: 1. Welche Ausgangsdatenstruktur?

2. Welche Extrainformation aufrechterhalten?

3. Aufwand zur Aufrechterhaltung?

4. Implementiere neue Operationen!

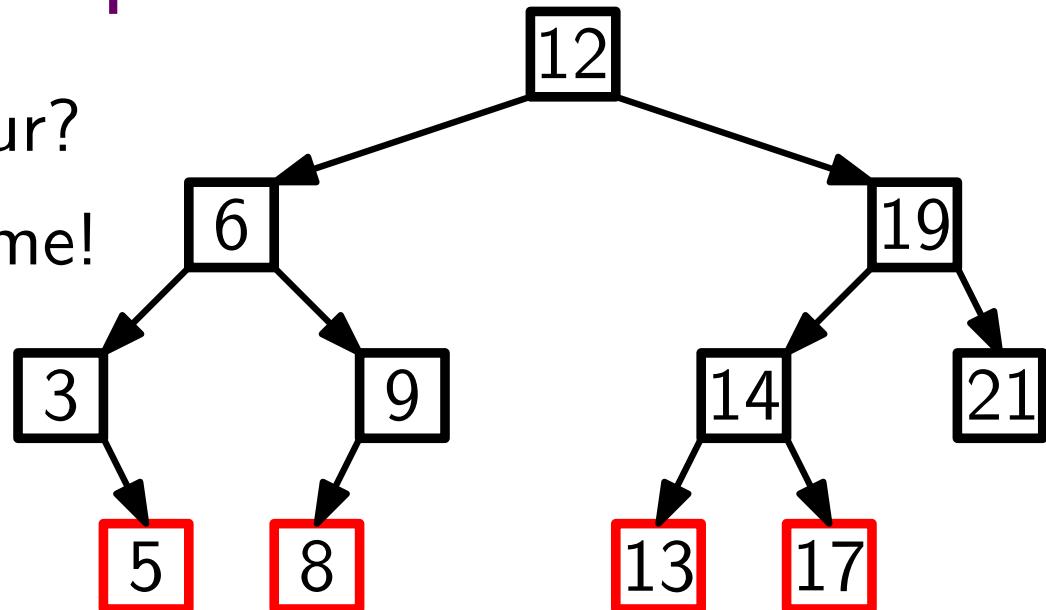
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere neue Operationen!

Select(int i):

```

 $v = \text{Minimum}()$ 
while  $v \neq \text{nil}$  and  $i > 1$  do
|    $v = \text{Successor}(v)$ 
|    $i = i - 1$ 
return  $v$ 
  
```

Rank(Node v):

```

 $j = 0$ 
while  $v \neq \text{nil}$  do
|    $v = \text{Predecessor}(v)$ 
|    $j = j + 1$ 
return  $j$ 
  
```

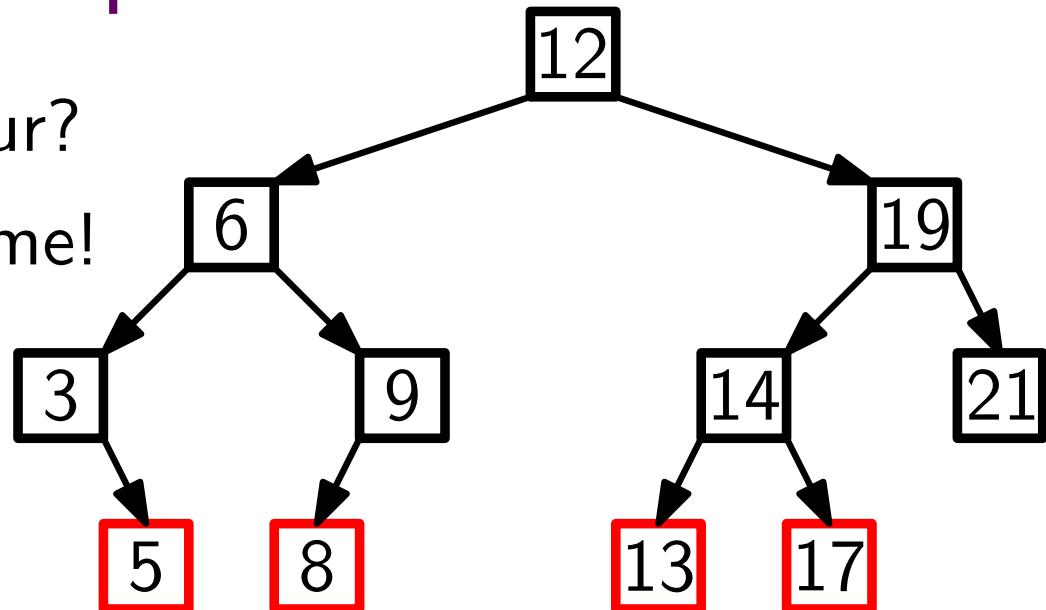
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit? Abschätzung bestmöglich?

Select(int i): $O(i \cdot h)$

```

 $v = \text{Minimum}()$ 
while  $v \neq \text{nil}$  and  $i > 1$  do
|    $v = \text{Successor}(v)$ 
|    $i = i - 1$ 
return  $v$ 
  
```

Rank(Node v): $O(\text{rank} \cdot h)$

```

 $j = 0$ 
while  $v \neq \text{nil}$  do
|    $v = \text{Predecessor}(v)$ 
|    $j = j + 1$ 
return  $j$ 
  
```

Das dynamische Auswahlproblem

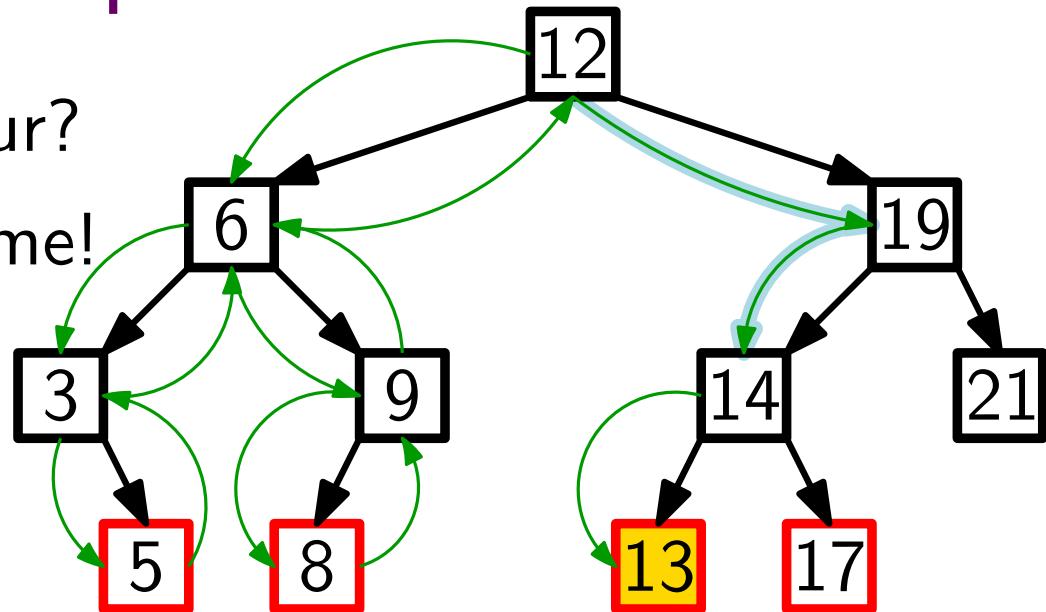
Select(7)

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

Select(int i): $O(\boxed{i} + h)$

```

 $v = \text{Minimum}()$ 
while  $v \neq \text{nil}$  and  $i > 1$  do
     $v = \text{Successor}(v)$ 
     $i = i - 1$ 
return  $v$ 
  
```

Rank(Node v): $O(\text{rank } + h)$

```

 $j = 0$ 
while  $v \neq \text{nil}$  do
     $v = \text{Predecessor}(v)$ 
     $j = j + 1$ 
return  $j$ 
  
```

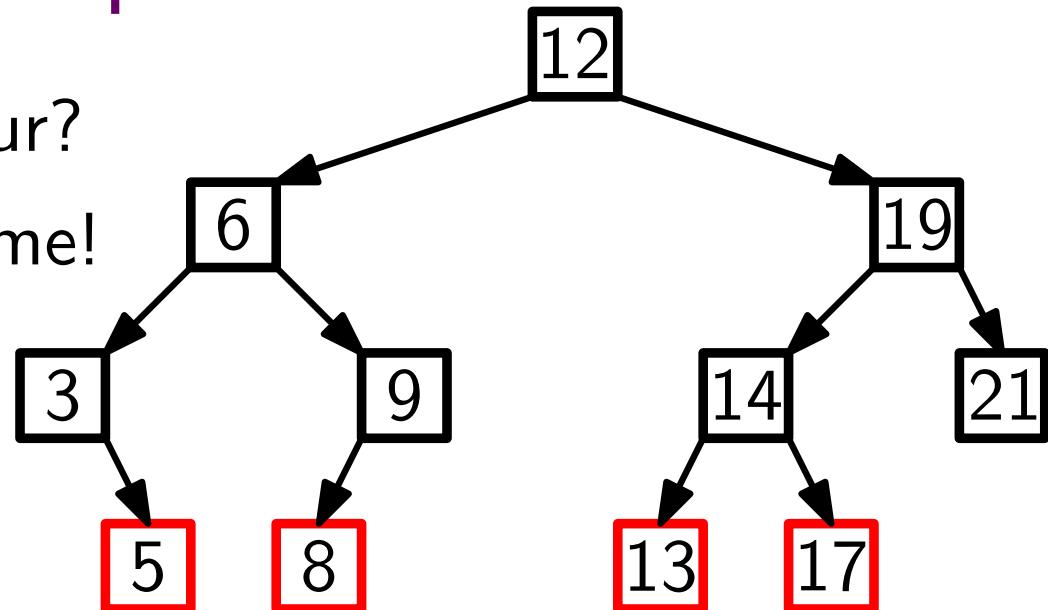
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten? – gar keine?!?

4. Implementiere! Laufzeit?

Select(int i): $O(i + h)$

Rank(Node v): $O(rank + h)$

Problem: Wenn $i \in \Theta(n)$ – z.B. beim Median –, dann ist die Laufzeit linear (wie im statischen Fall!).



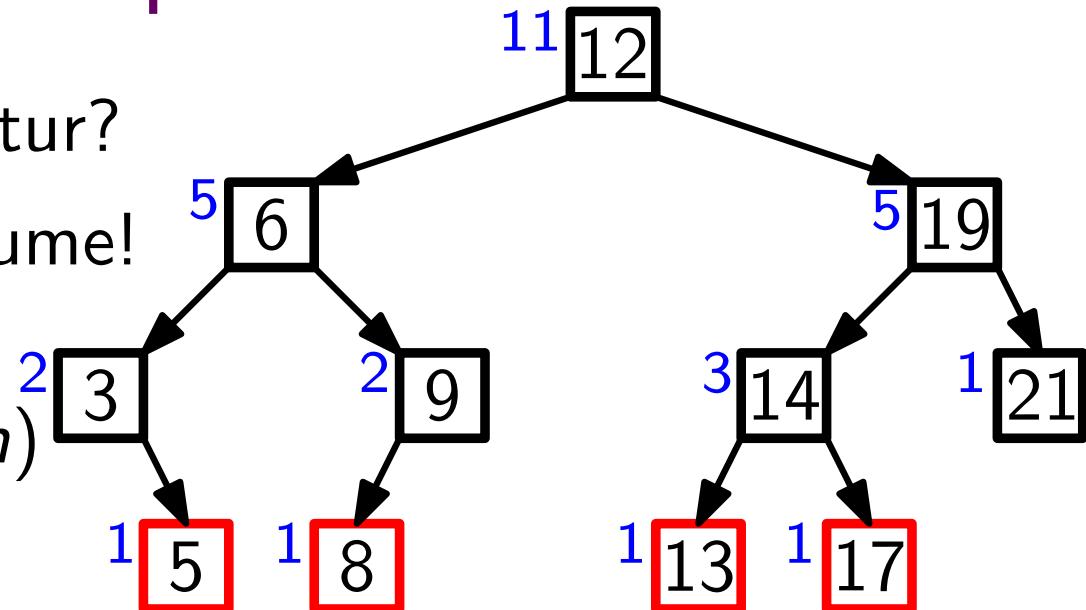
Das dynamische Auswahlproblem

1. Welche Ausgangsdatenstruktur?

- balancierte binäre Suchbäume!

z.B. Rot-Schwarz-Bäume

\Rightarrow Baumhöhe $h \in O(\log n)$



2. Welche Extrainformation aufrechterhalten?

- Größen der Teilbäume: für jeden Knoten v , speichere $v.size$

4. $\text{Select}(\text{Node } v = \text{root}, \text{int } i)$: $O(h)$

```

r = v.left.size + 1
if i == r then return v
else
  if i < r then
    | return Select(v.left, i)
  else
    | return Select(v.right, i - r)
  
```

$\text{Rank}(\text{Node } v)$:

```

r = v.left.size + 1
u = v
while u != root do
  if u == u.p.right then
    | r = r + u.p.left.size + 1
  u = u.p
return r
  
```

(vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife — ist r der Rang von v im Teilbaum mit Wurzel u .

1.) Initialisierung

Vor 1. Iteration gilt $u = v \Rightarrow u\text{-Rang}(v) = v.left.size + 1$. 

Rank(Node v):

```

 $r = v.left.size + 1$ 
 $u = v$ 
while  $u \neq root$  do
    if  $u == u.p.right$  then
         $r = r + u.p.left.size + 1$ 
     $u = u.p$ 
return  $r$  (vorausgesetzt, dass  $T.nil.size = 0$ )

```

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife — ist r der Rang von v im Teilbaum mit Wurzel u .

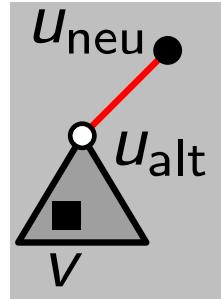
1.) Initialisierung ✓

u-Rang von v

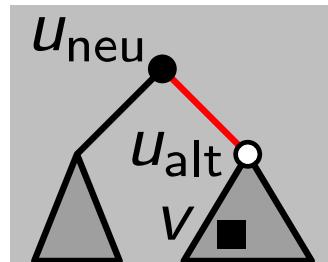
2.) Aufrechterhaltung ✓

Annahme: Inv. galt zu Beginn der aktuellen Iteration.

Zu zeigen: Inv. gilt dann auch am Ende der aktuellen Iter.



1. Fall: u war linkes Kind.
⇒ u -Rang von v bleibt gleich.



2. Fall: u war rechtes Kind.
⇒ u -Rang von v erhöht sich um Größe des li. Teilbaums von u plus 1 (für u selbst).

Rank(Node v):

$r = v.left.size + 1$

$u = v$

while $u \neq root$ **do**

if $u == u.p.right$ **then**

$r = r + u.p.left.size + 1$

$u = u.p$

return r (vorausgesetzt, dass $T.nil.size = 0$)

Korrektheit von Rank()

Invariante: Zu Beginn jeder Iteration der while-Schleife — ist r der Rang von v im Teilbaum mit Wurzel u .

1.) Initialisierung ✓

2.) Aufrechterhaltung ✓

3.) Terminierung ✓

Bei Schleifenabbruch: $u = \text{root}$.

$\Rightarrow r = u\text{-Rang}(v) = \text{Rang}(v)$.

Zusammenfassung:

Die Methode Rank() liefert wie gewünscht den Rang des übergebenen Knotens.

```

Rank(Node v):
    r = v.left.size + 1
    u = v
    while u ≠ root do
        if u == u.p.right then
            r = r + u.p.left.size + 1
        u = u.p
    return r
    (vorausgesetzt, dass T.nil.size = 0)

```

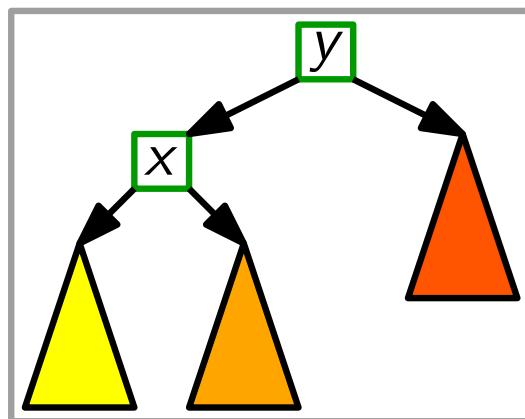
3. Aufwand zur Aufrechterhaltung der Extrainformation?

`RBInsert()` geht in zwei Phasen vor:

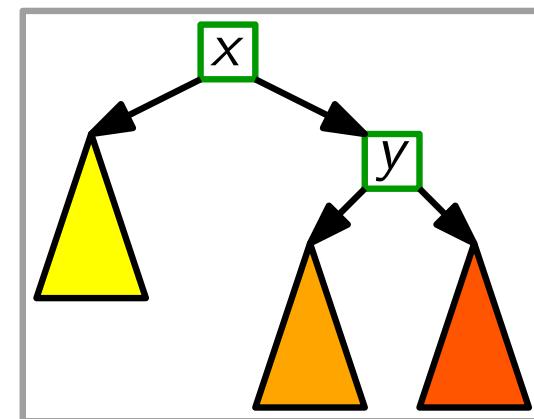
Phase I: Suche der Stelle, wo der neue Knoten z eingefügt wird.

Laufzeit $O(h)$ { Für alle Knoten v auf dem Weg von der Wurzel zu z :
Erhöhe $v.size$ um 1.

Phase II (RBInsertFixup): Strukturänderung nur in ≤ 2 Rotationen:



RightRotate(y)
Laufzeit $O(1)$
LeftRotate(x)



Welche Befehle müssen wir an `RightRotate(Node y)` anhängen, damit nach der Rotation alle `size`-Einträge wieder stimmen?

$$x.size = y.size$$

$$y.size = y.left.size + y.right.size + 1$$

(vorausgesetzt, dass $T.nil.size = 0$)

`RBDelete()` kann man analog „upgraden“.

Ergebnis

Satz. Das dynamische Auswahlproblem kann man so lösen, dass `Select()` und `Rank()` sowie alle gewöhnlichen Operationen für dynamische Mengen in einer Menge von n Elementen in $O(\log n)$ Zeit laufen.

Verallgemeinerung

Satz. Sei f Knotenattribut eines R-S-Baums mit n Knoten.

Falls für jeden Knoten v gilt:

$f(v)$ lässt sich aus Information in v , $v.left$, $v.right$ (inklusive $f(v.left)$ und $f(v.right)$) berechnen.



Dann kann man beim Einfügen und Löschen einzelner Knoten den Wert von f in allen Knoten aufrechterhalten, ohne die asymptotischen Laufzeit $O(\log n)$ der Update-Operationen zu verändern.

Beweisidee. Im Prinzip wie im Spezialfall $f \equiv size$.

Allerdings ist es im Prinzip möglich, dass sich die Veränderungen von einem gewissen veränderten Knoten bis in die Wurzel hochpropagieren. [Details Kapitel 14.2, CLRS]

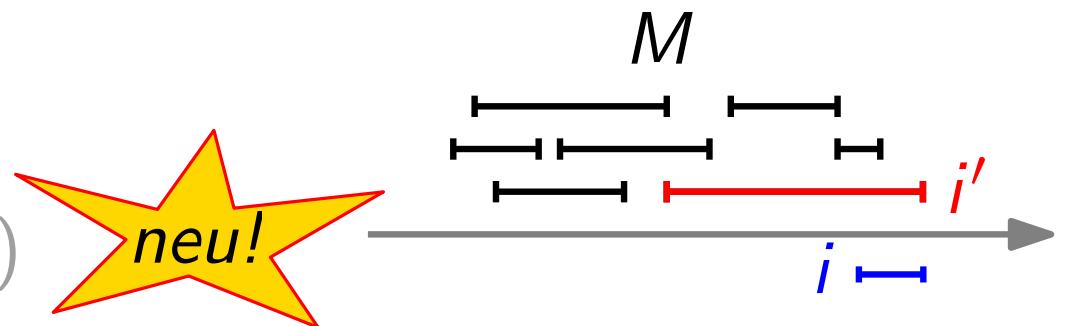
Noch ein Beispiel

zur Augmentierung von Rot-Schwarz-Bäumen (Kapitel 14.3):

Intervall-Baum

verwaltet eine Menge M von Intervallen und bietet Operationen:

- Element $\text{Insert}(\text{Interval } i)$
- Delete(Element e)
- Element $\text{Search}(\text{Interval } i)$



liefert ein Element mit Interval $i' \in M$ mit $i \cap i' \neq \emptyset$, falls ein solches existiert, sonst *nil*.

Bitte lesen Sie's und
stellen Sie Fragen...

Algorithmen und Datenstrukturen

Wintersemester 2018/19
16. Vorlesung

Amortisierte Analyse

Special Guest: Fabian Feitsch

Einstiegsbeispiel: Hash-Tabellen

Frage: Wie groß macht man eine Hash-Tabelle?

Ziel: So groß wie nötig, so klein wie möglich...

Verhindere, dass die Tabelle überläuft
oder dass Operationen ineffizient werden.

Problem: Was tun, wenn man die maximale Anzahl zu speichernder Elemente vorab nicht kennt?

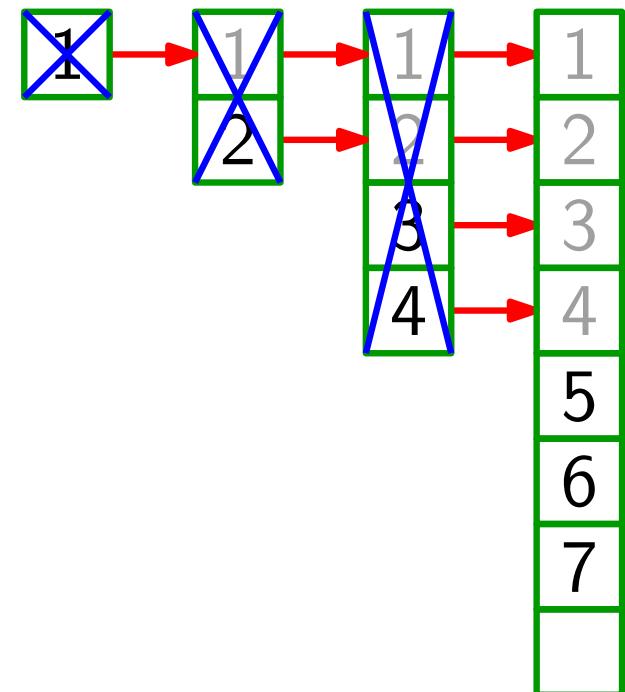
Lösung: *Dynamische Tabellen!*

Dynamische Tabellen

Idee:

- Wenn Tabelle voll, fordere doppelt so große Tabelle an (mit `new`).
- Kopiere alle Einträge von alter in neue Tabelle.
- Gib Speicher für alte Tabelle frei.

Insert(1)
Insert(2)
Insert(3)
Insert(4)
Insert(5)
Insert(6)
Insert(7)
...



Analyse: Welche Laufzeit benötigen n Einfügeoperationen im schlimmsten Fall?

Antwort:

- Tabelle wird genau ($\lceil \log_2 n \rceil$)-mal kopiert.
 - Im schlimmsten (letzten!) Fall ist der Aufwand $\Theta(n)$.
- Also ist der Gesamtaufwand ~~$\Theta(n \log n)$~~ , genauer $\Theta(n)$.
Let's see why...

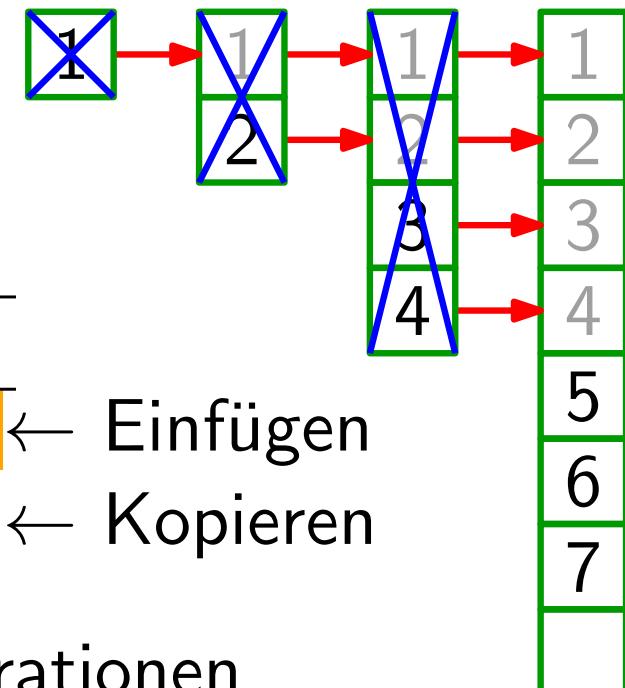
Lüge!

Genauere Abschätzung: Aggregationsmethode

Für $i = 1, \dots, n$ sei

c_i = Kosten fürs i -te Einfügen.

i	1	2	3	4	5	6	7	8	9
$size_i$	1	2	4	4	8	8	8	16	
c_i	1	1	1	1	1	1	1	1	
		1	2		4			8	



Also betragen die Kosten für n Einfügeoperationen

$$\sum_{i=1}^n c_i = n + \sum_{j=0}^{\lfloor \log_2(n-1) \rfloor} 2^j \leq n + \frac{2^{\log_2(n-1)+1} - 1}{2 - 1}$$

$$= n + 2(n - 1) - 1$$

$$< 3n \in \Theta(n)$$

$$\sum_{j=0}^k q^j = \frac{q^{k+1} - 1}{q - 1} \quad \begin{bmatrix} \text{endl.} \\ \text{geom.} \\ \text{Reihe} \end{bmatrix}$$

D.h. die durchschnittlichen („amortisierten“) Kosten sind $\Theta(1)$.

Amortisierte Analyse...

...bedeutet zu zeigen, dass die Operationen einer gegebenen Folge kleine durchschnittliche Kosten haben –
auch wenn einzelne Operationen in der Folge teuer sind!

Auch *randomisierte Analyse* kann man als Durchschnittsbildung (über alle Ereignisse, gewichtet nach Wahrscheinlichkeit) betrachten.

Bei amortisierter Analyse geht es jedoch um die durchschnittliche Laufzeit *im schlechtesten Fall* – nicht im Erwartungswert!

Wir betrachten 3 verschiedene Typen von amortisierter Analyse:

- Aggregationsmethode ✓
- Buchhaltermethode
- Potentialmethode

Buchhaltermethode

- Tendentiell genauer als die Aggregationsmethode
- Verbindet mit jeder Operation op_i amortisierte Kosten \hat{c}_i , die oft nicht mit den tatsächlichen Kosten c_i übereinstimmen.
 $\hat{c}_i > c_i \Rightarrow$ Wir legen etwas beiseite.
 $c_i > \hat{c}_i \Rightarrow$ Wir bezahlen teure Operationen mit vorher Beiseitegelegtem.
- Damit's klappt: wir dürfen nie in die Miesen kommen –

Guthaben $\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$ darf nicht negativ werden!

Dann gilt $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$.

D.h. amortisierte Kosten sind obere Schranke für tatsächliche Kosten!

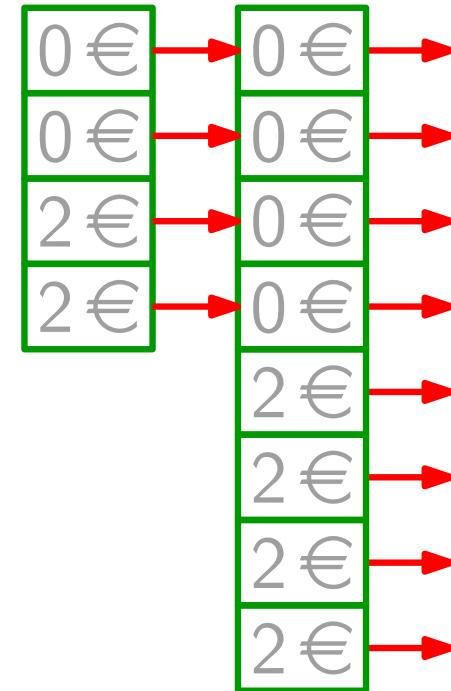
Buchhaltermethode für dynamische Tabellen

Jede Einfügeoperation op_i bezahlt $\hat{c}_i = 3\text{€}$:

- 1€ fürs tatsächliche Einfügen
- 2€ für die nächste Tabellenvergrößerung

Wir verknüpfen die Teilguthaben mit konkreten Objekten der Datenstruktur.

Damit wird deutlich, dass die DS nie Miese macht.



Also sind amortisierte Kosten obere Schranke für tatsächliche Kosten!

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i = 3n = \Theta(n)$$

D.h. die (tats.) Kosten für n Einfügeoperationen betragen $\Theta(n)$.

Buchhaltermethode: noch'n Beispiel



Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abs. Datentyp

```
boolean Empty()  
Push(key k)  
key Pop()  
key Top()
```

```
Multipop(int k)
```



Implementierung

```
while not Empty() and k > 0 do  
    Pop()  
    k = k - 1
```

Buchhaltermethode: Stapel mit Multipop

Betrachte Folge von Push-, Pop- und Multipop-Operationen.

Operation;	tatsächliche Kosten c_i	amortisierte Kosten \hat{c}_i
Push	1	2
Pop	1	0
$\text{Multipop}(k_i)$	$\min\{k_i, \text{size}_i\}$	0

Geht das gut??? – Ja! D.h. Folge von n Op. dauert $\Theta(n)$ Zeit.

Zeige: Amortisierte Kosten „bezahlen“ immer für die echten!

- Jede Push-Operation legt einen Teller auf den Stapel.
Dafür bezahlt sie 1 € und legt noch 1 € auf den Teller.
- Jede (Multi-)Pop-Operation wird mit den Euros auf den Tellern, die sie wegnimmt, komplett bezahlt.

Potentialmethode

Idee: Betrachte Bankguthaben (siehe Buchhaltermethode)
als physikalische Größe,
die den augenblicklichen Zustand der DS beschreibt.

Datenstruktur $D_0 \xrightarrow{op_1} D_1 \xrightarrow{op_2} \dots \xrightarrow{op_n} D_n$

Wähle Potential $\Phi: D_i \rightarrow \mathbb{R}$. O.B.d.A. $\Phi(D_0) = 0$

Ziel: Bank macht keine Miesen.

Also fordern wir $\Phi(D_i) \geq 0$ für $i = 1, \dots, n$.

amortisierte Kosten ↘ *echte Kosten* ↗ *Potentialdifferenz*
Def. $\hat{c}_i = c_i + \Delta\Phi(D_i)$, wobei $\Delta\Phi(D_i) = \Phi(D_i) - \Phi(D_{i-1})$

Folge: $\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1}))$ *teleskopierende Summe*



$$= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^n c_i$$

D.h. amortisierte Kosten „bezahlen“ für echte Kosten.

Potentialmethode: Stapel mit Multipop

To do: Definiere Potentialfunktion –
in Abhängigkeit vom aktuellen Zustand des Stapels!

Idee: Nimm $\Phi(D_i) = \text{size}_i$, also aktuelle Stapelgröße.
 $\Rightarrow \Phi(D_0) = 0$ und $\Phi(D_1), \dots, \Phi(D_n) \geq 0$. 

Prüfe: Falls die i -te Operation eine Push-Operation ist:
 $\Rightarrow \Delta\Phi(D_i) = +1$ und $\hat{c}_i = c_i + \Delta\Phi D_i = 1 + 1 = 2$

Falls die i -te Operation eine Multipop-Operation ist:
 $\Rightarrow \Delta\Phi(D_i) = -\min\{k_i, \text{size}_i\}$
 $c_i = +\min\{k_i, \text{size}_i\}$

$$\hat{c}_i = c_i + \Delta\Phi D_i = 0, \text{ dito mit Pop } (k_i = 1).$$

Also: Amortisierte Kosten pro Operation $O(1)$.
 \Rightarrow Echte Kosten für n Oper. im worst case $O(n)$.

Was
sind die
amort.
Kosten?

Zusammenfassung

Zeige mit **amortisierter Analyse**, dass die Operationen einer gegebenen Folge kleine durchschnittliche Kosten haben – *auch wenn einzelne Operationen in der Folge teuer sind!*

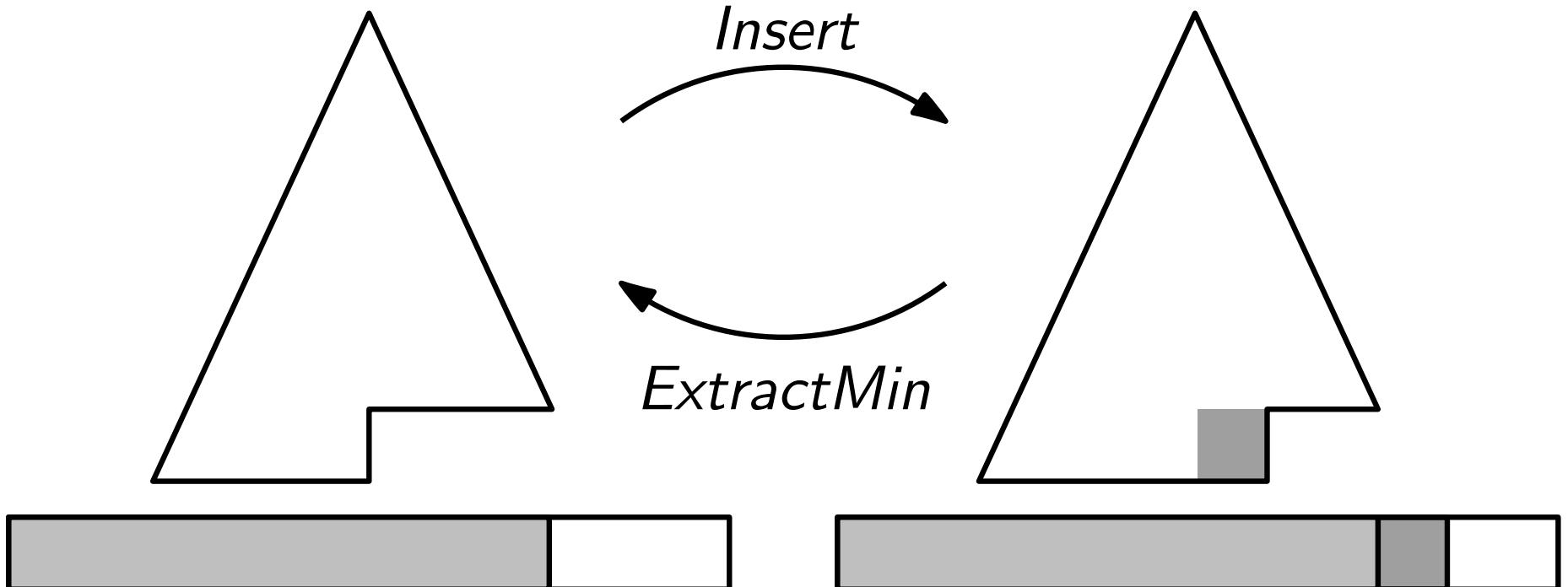
Drei Typen von amortisierter Analyse:

- Aggregationsmethode 
Summiere tatsächliche Kosten (oder obere Schranken dafür) auf.
- Buchhaltermethode 
Verbinde Extrakosten mit konkreten Objekten der DS und bezahle damit teure Operationen.
- Potentialmethode 
Definiere Potential der gesamten DS, so dass mit der Potentialdifferenz teure Operationen bezahlt werden können.

Übungsaufgaben zur amortisierten Analyse (I)

Gegeben sei ein gewöhnlicher MinHeap, dessen Methoden *Insert* und *ExtractMin* im schlechtesten Fall $O(\log n)$ Zeit brauchen.

Zeigen Sie mit der **Potentialmethode**, dass *Insert* amortisiert $O(\log n)$ Zeit und *ExtractMin* amortisiert $O(1)$ Zeit benötigt.



Übungsaufgaben zur amortisierten Analyse (II)

Entwerfen Sie eine Datenstruktur zum Verwalten einer dynamischen Menge von Zahlen. Die DS soll 2 Methoden haben:

- *Insert* zum Einfügen einer Zahl und
- *DeleteLargerHalf* zum Löschen aller Zahlen aus der Datenstruktur, die größer oder gleich dem aktuellen Median der Zahlenmenge sind.

Beide Methoden sollen **amortisiert $O(1)$** Zeit benötigen.

Tipp: Verwenden Sie eine Liste!

1. Beschreiben Sie Ihren Entwurf der Datenstruktur einschließlich der beiden Methoden in Worten.
2. Analysieren Sie mithilfe der **Buchhaltermethode**. Geben Sie die amortisierten Kosten, die Sie mit *Insert* und *DeleteLargerHalf* verbinden, exakt an.

Algorithmen und Datenstrukturen

Wintersemester 2018/19
17. Vorlesung

Nächstes Paar

2. Zwischentest

Bitte melden Sie sich sofort an, falls Sie das noch nicht getan haben.

Die Abstimmung endet heute (Di, 18.12.) um 14:00 Uhr.

Problem:

- Gegeben: Menge P von n Punkten in der Ebene,
jeder Punkt $p \in P$ als (x_p, y_p) .
Finde: Punktpaar $\{p, q\} \subseteq P$ mit kleinstem
(euklidischen) Abstand.
- **Def.** Euklidischer Abstand von p und q ist
$$d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}.$$

Lösung:**Laufzeit:** $\Theta(n^2)$

- Gehe durch alle $\binom{n}{2}$ Punktpaare und berechne ihren Abstand.
- Gib ein Paar mit kleinstem Abstand zurück.

Mach's besser!

Entwurfsparadigma: – inkrementell?

– randomisiert?

– Teile und Herrsche?!

Spezialfall:



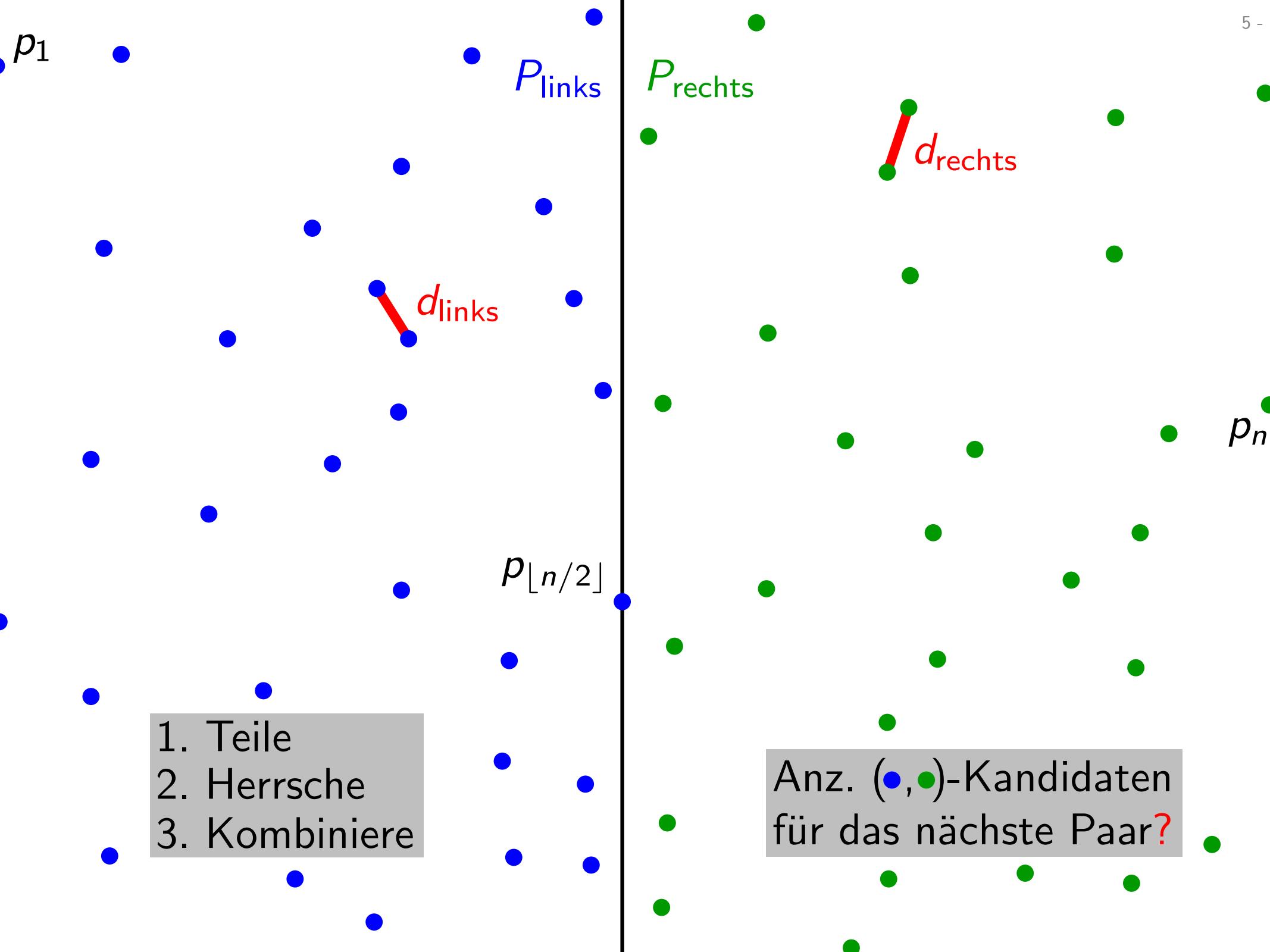
Lösung:

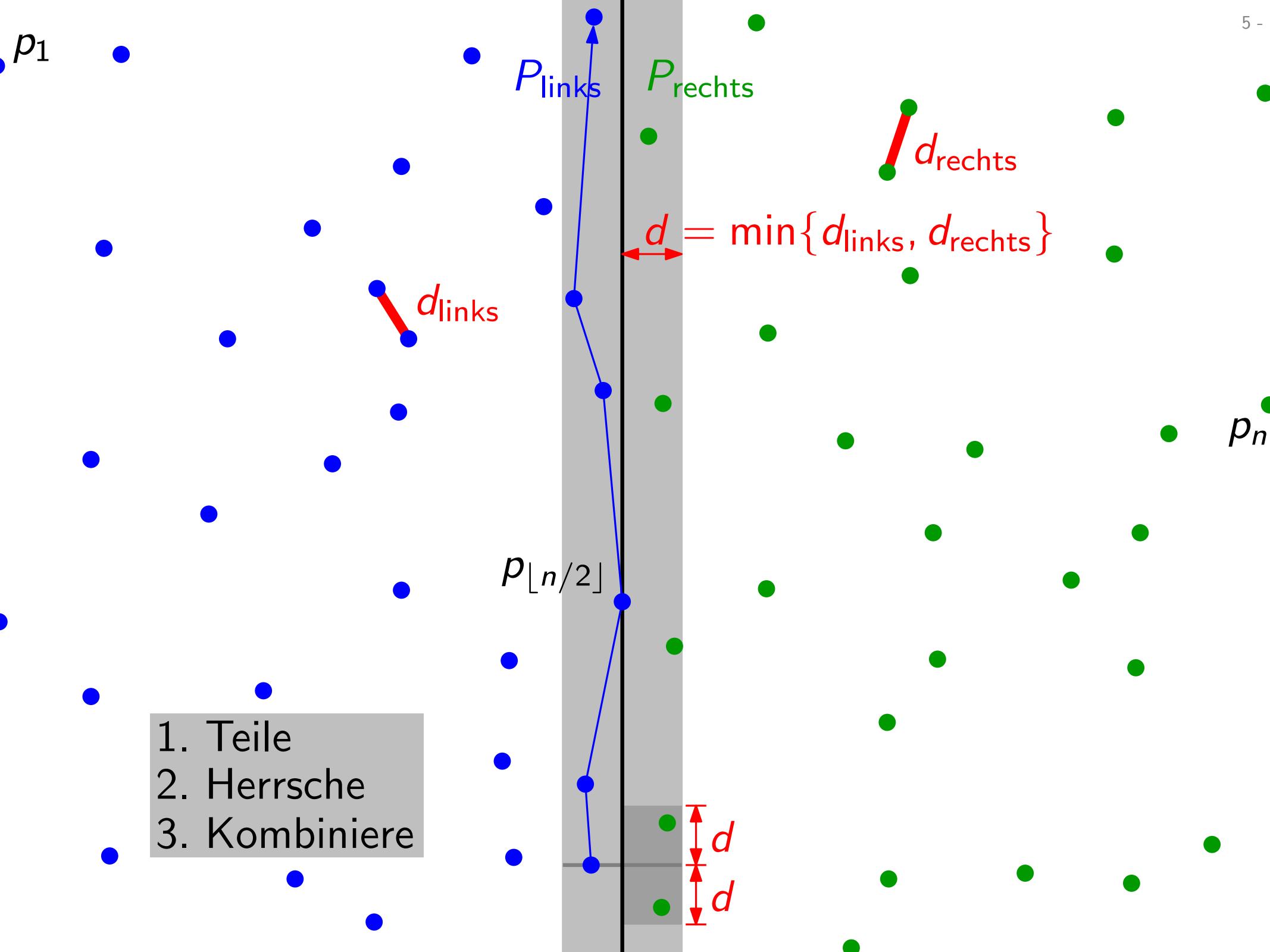
- Sortiere (nach x-Koordinate).
- Berechne Abstände aller *aufeinanderfolgender* Punktpaare.

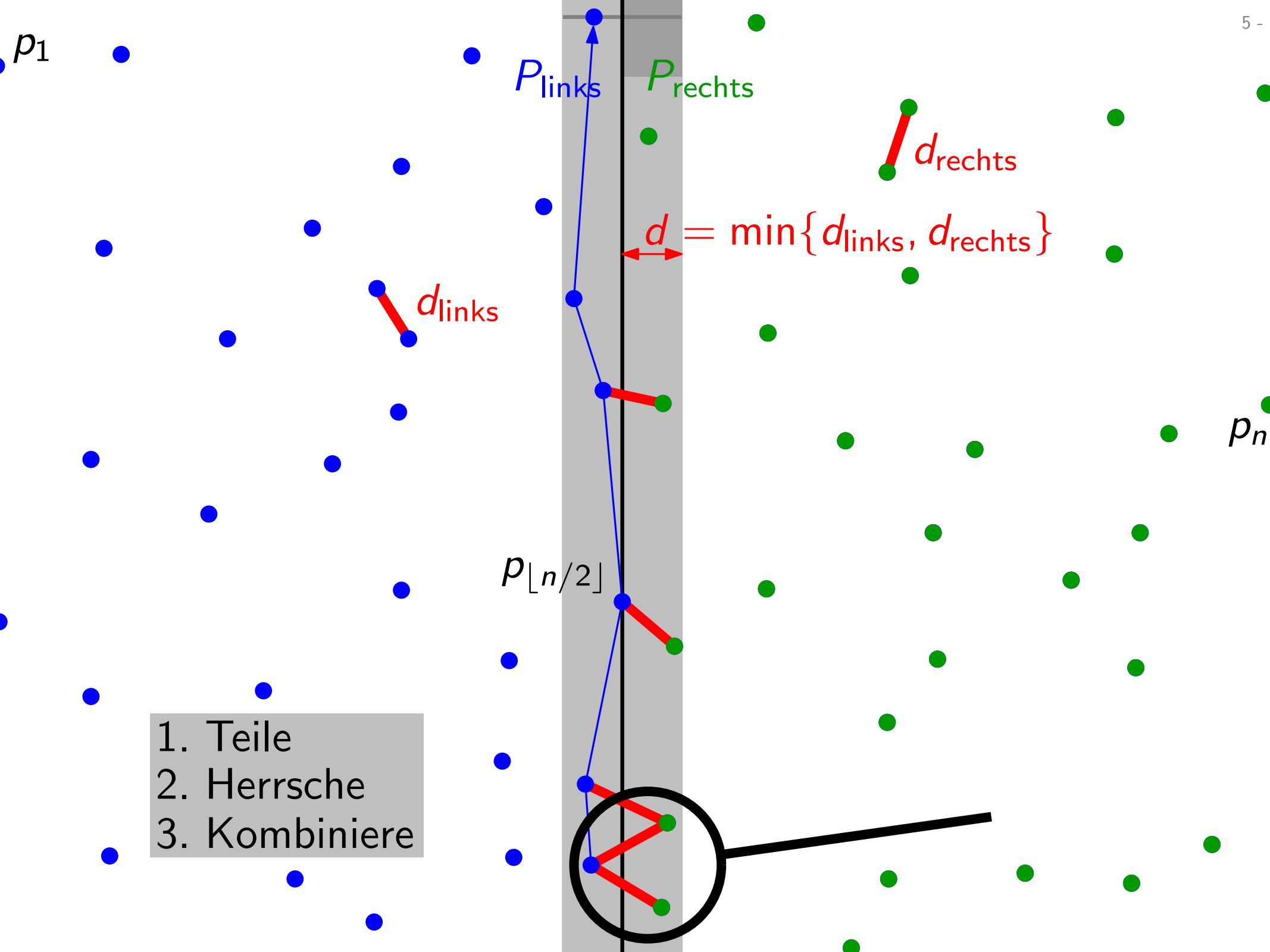
- Bestimme das Minimum dieser Abstände.

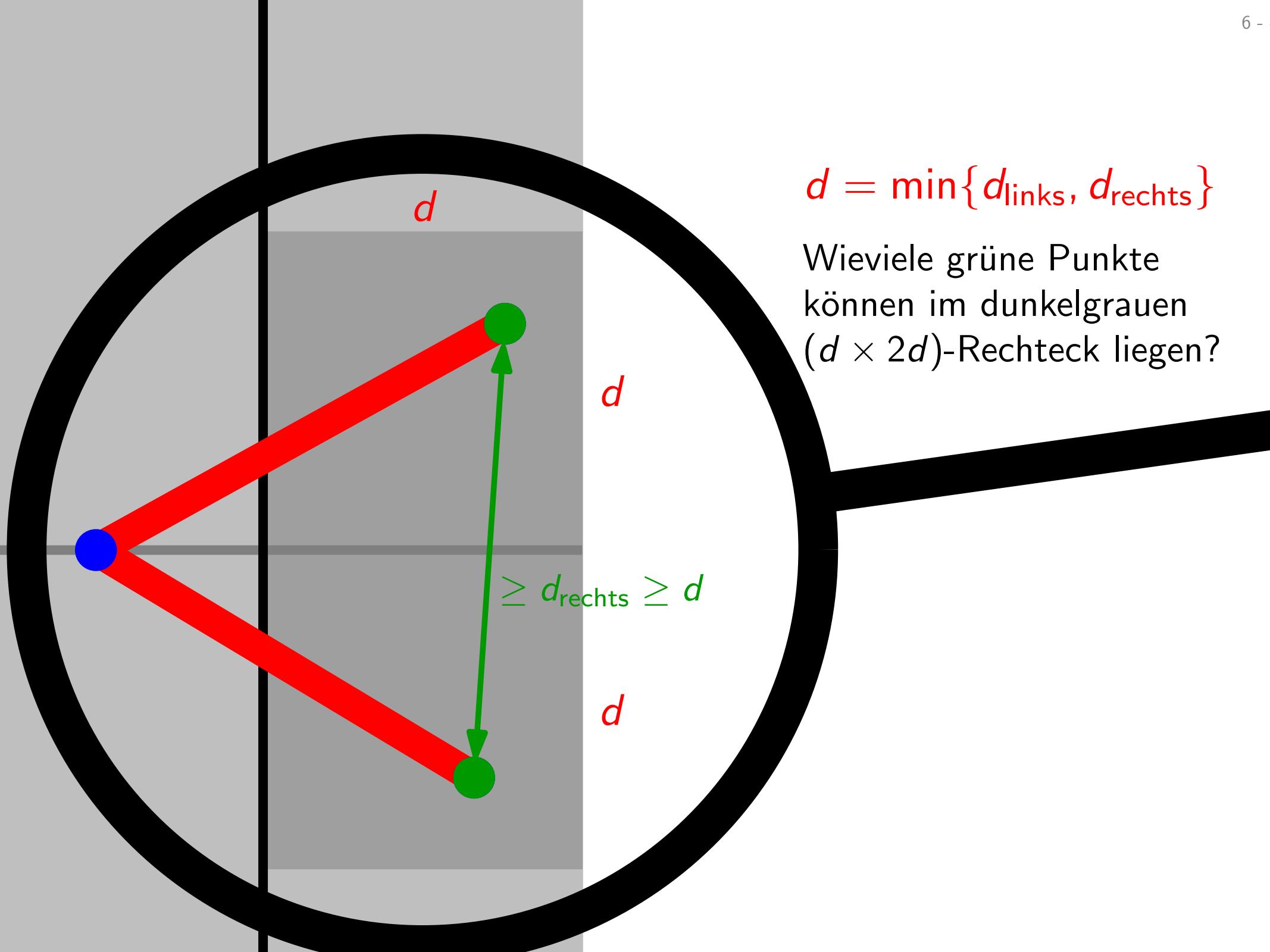
Strukturelle Einsicht:

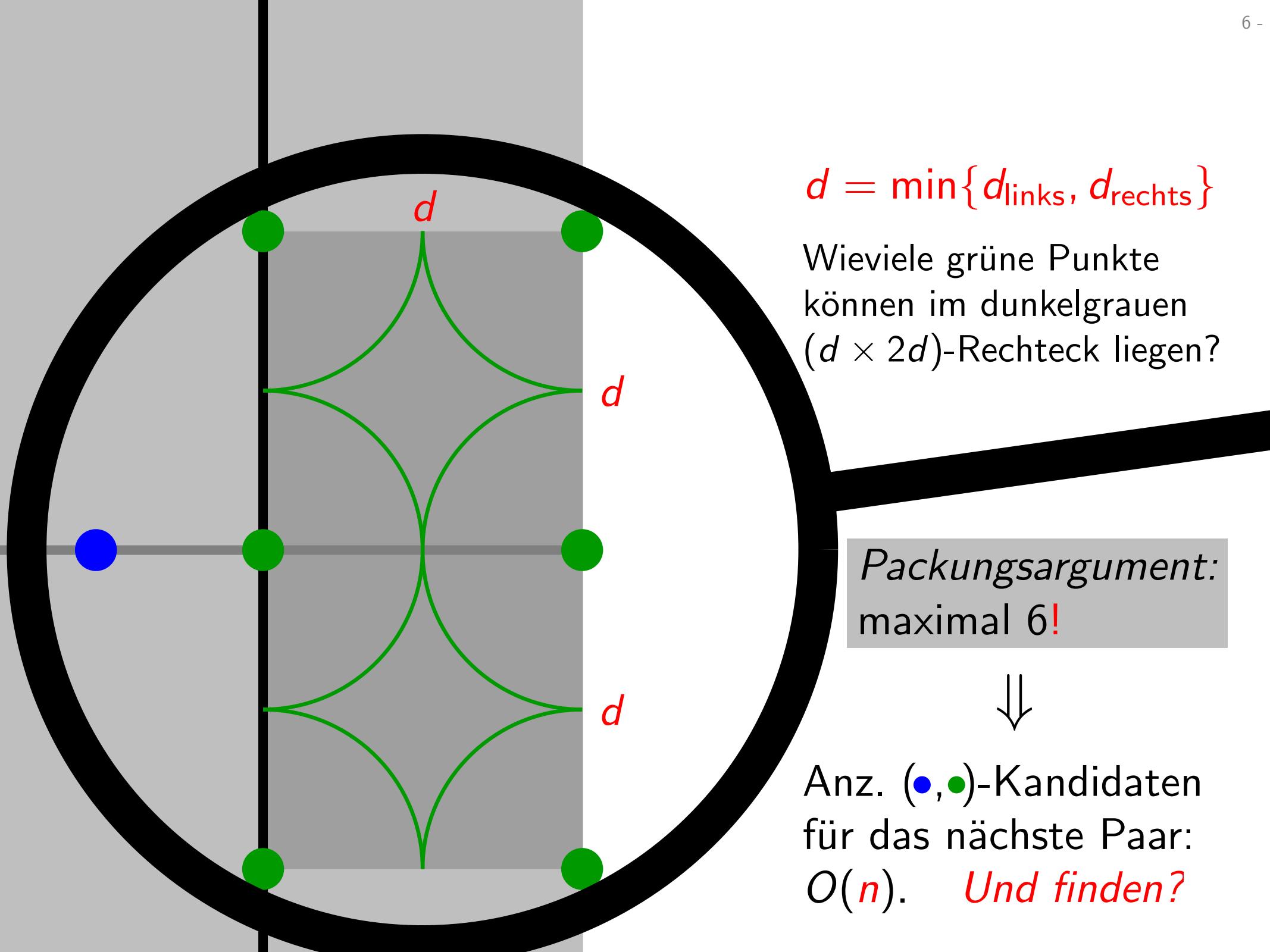
Kandidatenmenge der Größe $n - 1$,
die gesuchtes Objekt enthält.











$$d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$$

Wieviele grüne Punkte können im dunkelgrauen $(d \times 2d)$ -Rechteck liegen?

*Packungsargument:
maximal 6!*



Anz. (\bullet, \bullet) -Kandidaten für das nächste Paar:
 $O(n)$. *Und finden?*

Algorithmus

$$T(n) = \begin{cases} \text{Laufzeit des rekursiven Teils,} \\ \text{d.h. ohne Vorverarbeitung (1.)} \end{cases}$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$

2. Teile: $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$, $P_{\text{rechts}} = P \setminus P_{\text{links}}$

3. Herrsche:

bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}
 d_{rechts} P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$
- sortiere P_{links} und P_{rechts} nach y-Koordinate
- gehe „gleichzeitig“ durch P_{links} und P_{rechts} :
 für jeden Punkt p in P_{links} gehe in P_{rechts} bis y-Koord. $y_p + d$; halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)
- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

Algorithmus

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n)$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$

2. Teile: $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$, $P_{\text{rechts}} = P \setminus P_{\text{links}}$

3. Herrsche:

bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}

d_{rechts}

P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$

$O(1)$

- sortiere P_{links} und P_{rechts} nach y-Koordinate

$O(n \log n)$

- gehe „gleichzeitig“ durch P_{links} und P_{rechts} :

für jeden Punkt p in P_{links} gehe in P_{rechts} bis y-Koord. $y_p + d$; halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)

$O(n)$

- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

Laufzeit

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n)$$

Also $T(n) \approx 2T(n/2) + O(n \log n)$

Rekursionsgleichung mit Master-Theorem lösen?

Bestimme Parameter für das Theorem:

$$a = b = 2, f(n) = O(n \log n).$$

Betrachte $n^{\log_b a} = n^{\log_2 2} = n^1$.

Gilt $f \in \left\{ \begin{array}{ll} O(n^{1-\varepsilon}) & \text{für ein } \varepsilon > 0 \\ \Theta(n^1) & \\ \Omega(n^{1+\varepsilon}) & \text{für ein } \varepsilon > 0 \end{array} \right\}$?

Nein, $f: n \mapsto O(n \log n)$ passt in keinen der drei Fälle.



Die Rekursionsbaummethode liefert... $T(n) = O(n \log^2 n)$.

Noch besser?

$$T(n) \approx 2T(n/2) + O(n \log n) = O(n \log^2 n)$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$

2. Teile: P in $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$ und $P_{\text{rechts}} = P \setminus P_{\text{links}}$

3. Herrsche:

bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}
 $?$! d_{rechts} P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$
- sortiere P_{links} und P_{rechts} nach y-Koordinate

- gehe „gleichzeitig“ durch P_{links} und P_{rechts} :
 - für jeden Punkt p in P_{links} gehe in P_{rechts} bis y-Koord. $y_p + d$;
 - halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)
- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

Noch besser!

$$T(n) \approx 2T(n/2) + O(n \log n) = O(n \log^{\otimes} n)$$

1. Sortiere P nach x-Koordinate $\rightarrow p_1, \dots, p_n$ mit $x_1 \leq \dots \leq x_n$
 und $P' = P$ nach y-Koordinate $\rightarrow p'_1, \dots, p'_n$ mit $y'_1 \leq \dots \leq y'_n$

2. Teile: P in $P_{\text{links}} = \{p_1, \dots, p_{\lfloor n/2 \rfloor}\}$ und $P_{\text{rechts}} = P \setminus P_{\text{links}}$
 P' in P'_{links} und P'_{rechts} (sortiert nach y-Koordinate)

3. Herrsche:
 bestimme rekursiv kleinsten Abstand d_{links} v. Paaren in P_{links}
 d_{rechts} P_{rechts}

4. Kombiniere:

- $d = \min\{d_{\text{links}}, d_{\text{rechts}}\}$
- gehe „gleichzeitig“ durch P'_{links} und P'_{rechts} :
 für jeden Punkt p in P'_{links} gehe in P'_{rechts} bis y-Koord. $y_p + d$; halte die letzten 6 Punkte im grauen Streifen aufrecht ($\rightarrow K_p$)
- bestimme Min. d_{mitte} über alle $d(p, q)$ mit $p \in P'_{\text{links}}$ und $q \in K_p$
- gib Min. von d_{mitte} , d_{links} und d_{rechts} (und entspr. Paar) zurück

$O(n)$

Zusammenfassung

1. Vorverarbeitung (2× Sortieren)	$O(n \log n)$	
2. Teilen	$O(n)$	
3. Herrschen	$2T(n/2)$	
4. Kombinieren	$O(n)$	
<hr/>		
Gesamtlaufzeit	$O(n \log n)$	

Speicherplatzbedarf?

$O(n)$, wenn P' *in situ* in P'_{links} und P'_{rechts} zerlegt wird.

Ist die Laufzeit $O(n \log n)$ optimal?

Def. *Element-Uniqueness-Problem (für natürliche Zahlen)*

Gegeben eine Folge a_1, \dots, a_n von n Zahlen,
kommt jede Zahl nur einmal vor, d.h. $a_i \neq a_j$ für $i \neq j$?

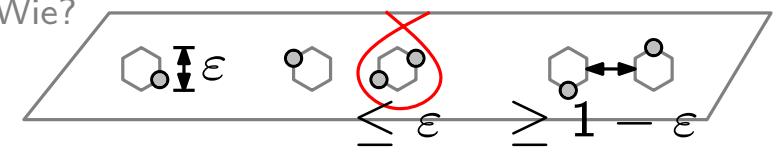
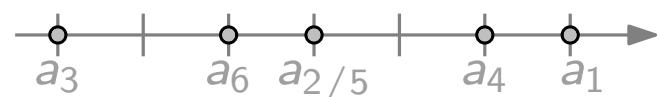
Satz. Das Element-Uniqueness-Problem kann nicht schneller als in $\Omega(n \log n)$ Zeit gelöst werden –
wenn man als Rechenmodell das sogenannte
algebraische Entscheidungsbaummodell zugrunde legt.

Was bedeutet das für das Problem *Nächstes Paar*?

Angenommen wir könnten Nächstes Paar in $o(n \log n)$ Zeit lösen – dann auch Element Uniqueness! ↗

Wie? Teste, ob das nächste Paar Abstand 0 hat!

Genaugenommen muss man die Zahlen a_1, \dots, a_n in eine Menge von (paarweise verschiedenen!) Punkten der Ebene transformieren, aber auch das geht! – Wie?



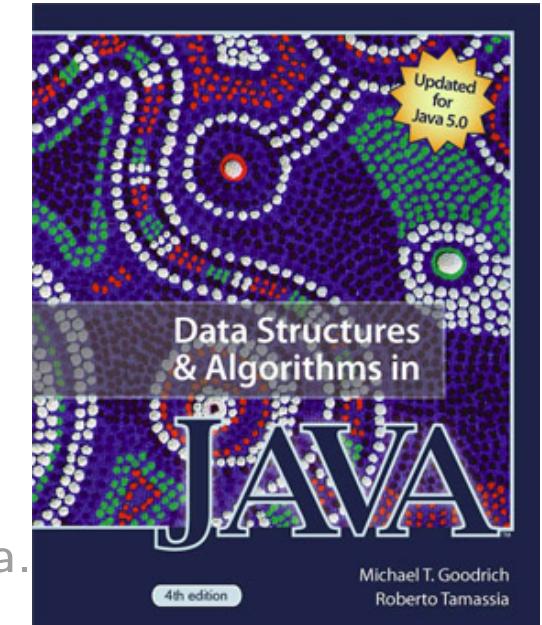
Das heißt. . .

Satz. Das Problem Nächstes Paar kann nicht schneller als in $\Omega(n \log n)$ Zeit gelöst werden, wenn man als Rechenmodell das algebraische Entscheidungsbaummodell zugrunde legt.

Kor. Unser $O(n \log n)$ -Zeit-Algorithmus für das Problem Nächstes Paar ist asymptotisch optimal, wenn man. . . .

Учиться, учиться и учиться

- Implementieren Sie die einfache Brute-Force-Lösung in Java.
- Implementieren Sie einen einfachen Teile-und-Herrsche-Algorithmus, der im Herrsche-Schritt *alle* (quadratisch vielen) (●, ●)-Kandidaten testet. *(Ist der schneller als der Brute-Force-Alg.?)*
- Implementieren Sie den hier vorgestellten Teile-und-Herrsche-Algorithmus, der in $O(n \log^2 n)$ Zeit läuft!
- Implementieren Sie den hier vorgestellten Teile-und-Herrsche-Algorithmus, der in $O(n \log n)$ Zeit läuft!



Goodrich & Tamassia:
Data Structures & Algorithms in Java.
Wiley, 4. Aufl., 2005 (5. Aufl., 2010)

Algorithmen & Datenstrukturen

Lernziele: In dieser Veranstaltung haben Sie schon gelernt...

- die Effizienz von Algorithmen zu messen und miteinander zu vergleichen,
- grundlegende Algorithmen und Datenstrukturen in Java zu implementieren,
- selbst Algorithmen und Datenstrukturen zu entwerfen sowie
- deren Korrektheit und Effizienz zu beweisen.

Inhalt: • Grundlagen und Analysetechniken

- Sortierverfahren

- Java

- Datenstrukturen

- Graphenalgorithmen (kürzeste Wege, min. Spannbäume)
- Systematisches Probieren (dynamisches Progr., Greedy-Alg.)

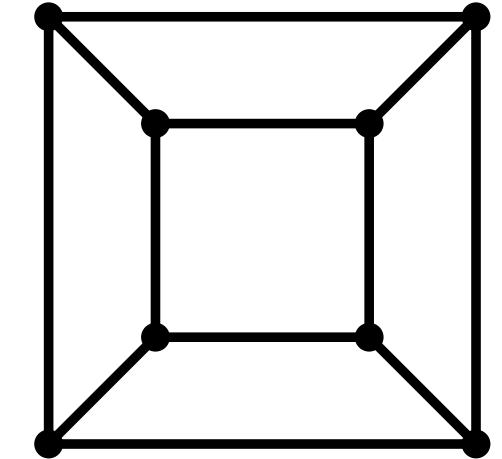
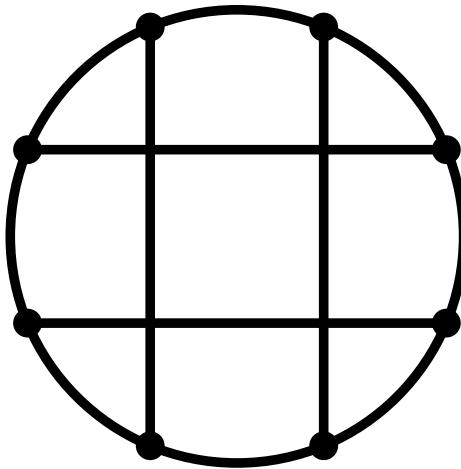
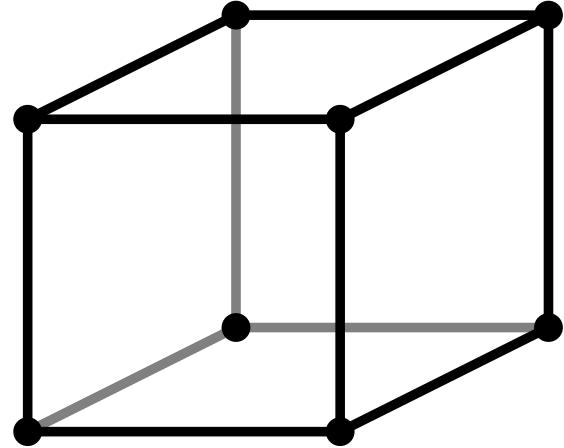
Todo

Algorithmen und Datenstrukturen

Wintersemester 2018/19
18. Vorlesung

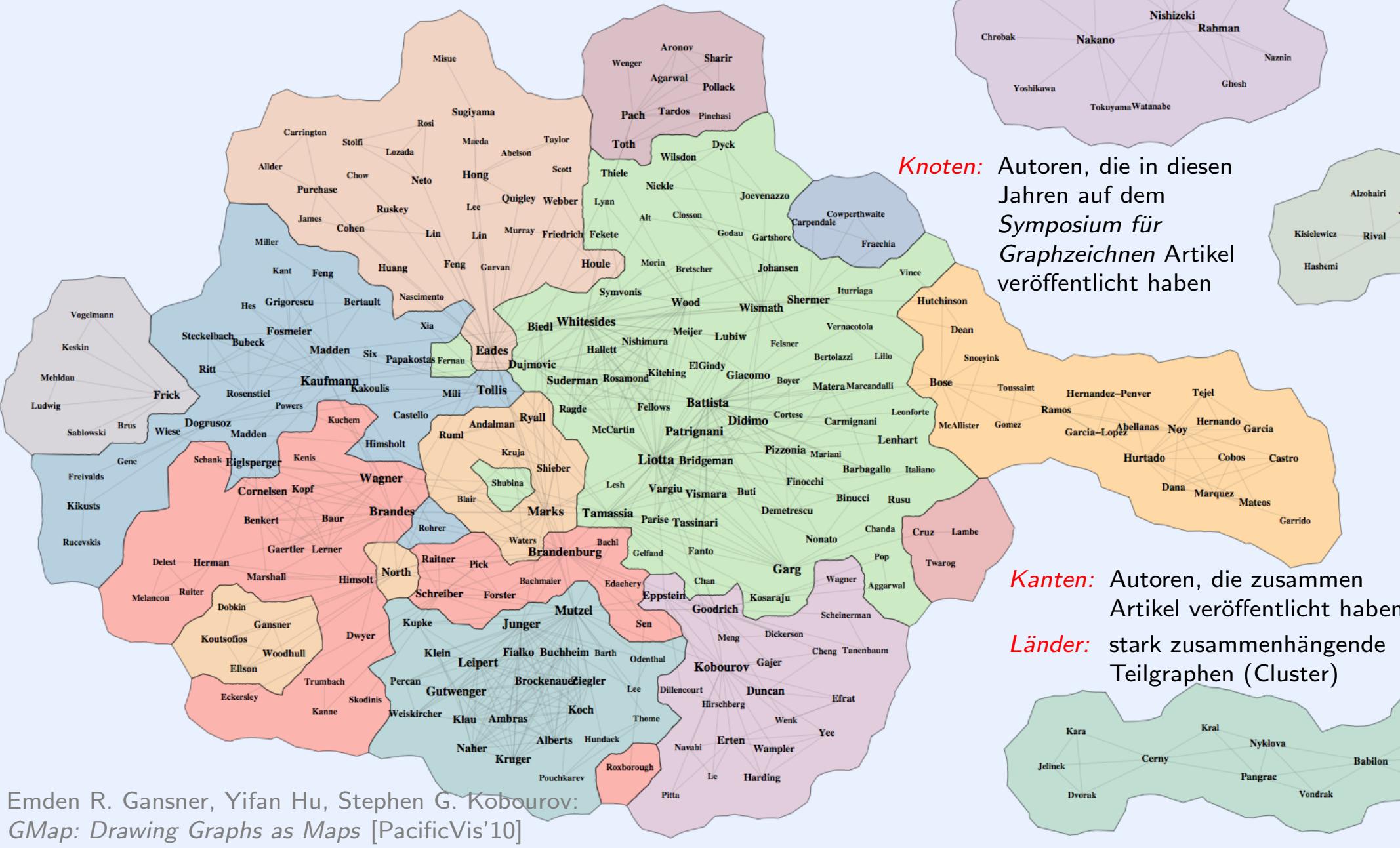
Graphen:
Repräsentation und Durchlaufstrategien

Was ist das?



Ein (und derselbe) *Graph*; der dreidimensionale Hyperwürfel.

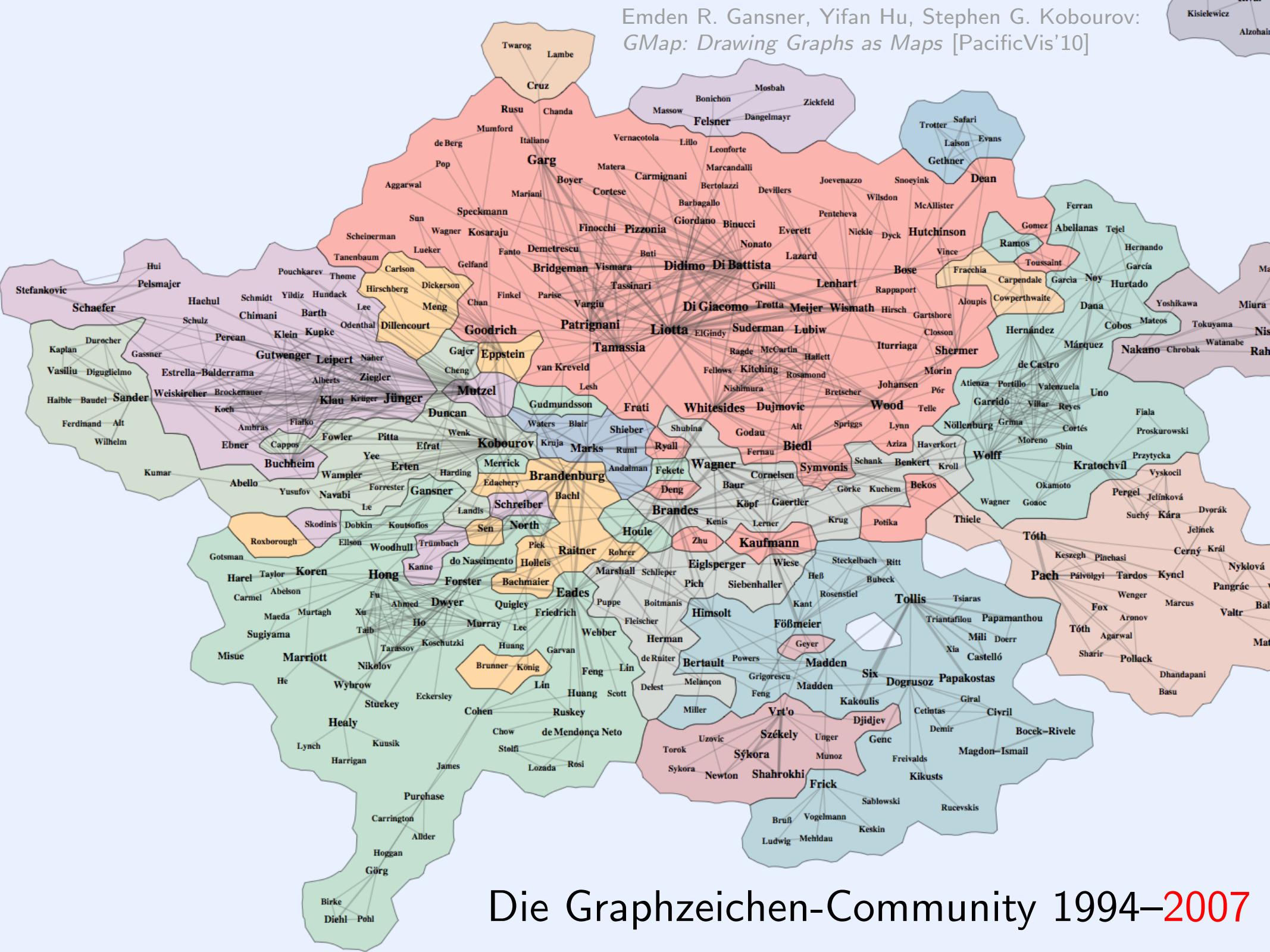
Die Graphzeichen-Community 1994–2004



Knoten: Autoren, die in diesen Jahren auf dem *Symposium für Graphzeichnen* Artikel veröffentlicht haben

Kanten: Autoren, die zusammen Artikel veröffentlicht haben

Länder: stark zusammenhängende Teilgraphen (Cluster)



Die Graphzeichen-Community 1994–2007

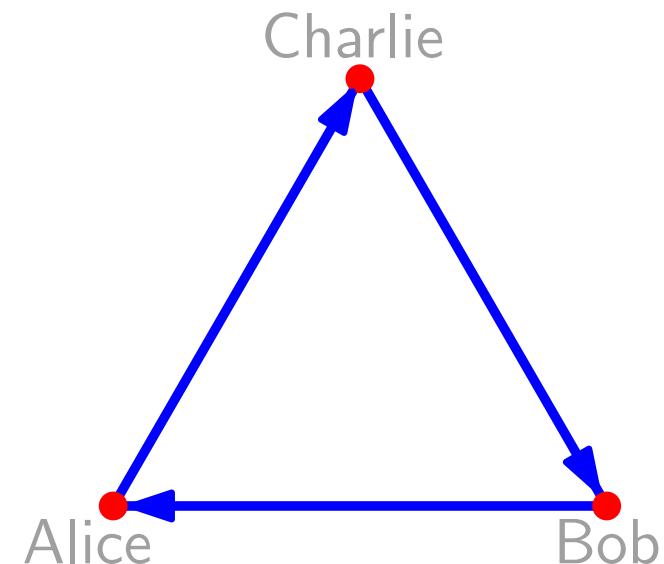
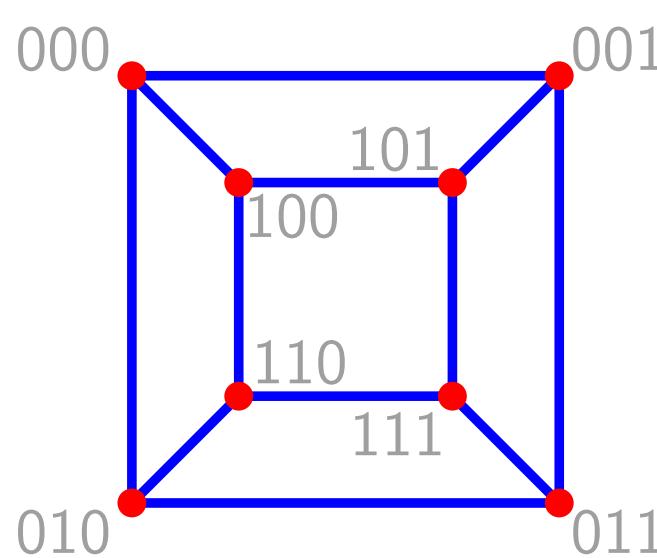
F: Was ist ein Graph?

A₁: Ein (ungerichteter) Graph ist ein Paar (V, E) , wobei

- V Knotenmenge und
- $E \subseteq \binom{V}{2} = \{\{u, v\} \subseteq V \mid u \neq v\}$ Kantenmenge.

$$V = \{000, 001, \dots, 111\}$$

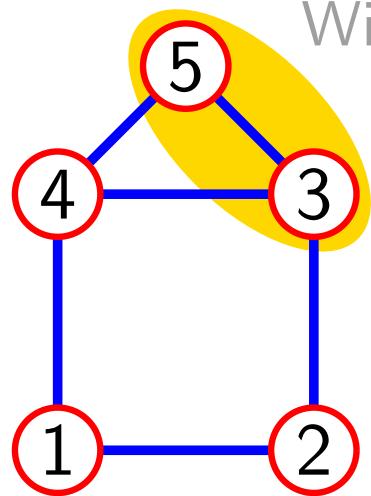
$$\{u, v\} \in E \Leftrightarrow ?$$



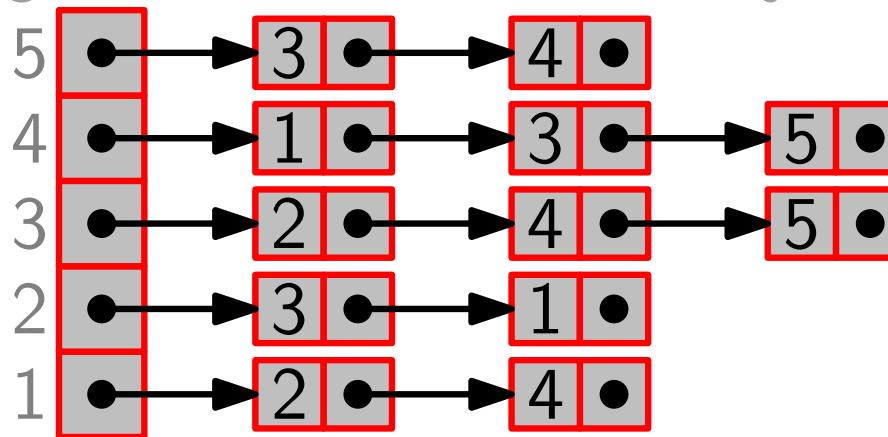
A₂: Ein gerichteter Graph ist ein Paar (V, E) , wobei

- V Knotenmenge und
- $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$ Kantenmenge.

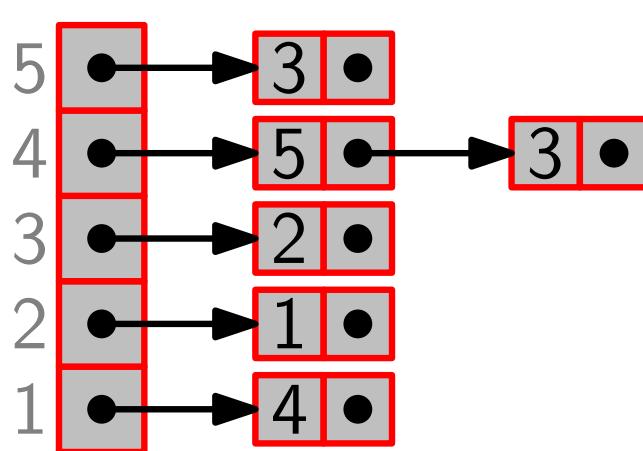
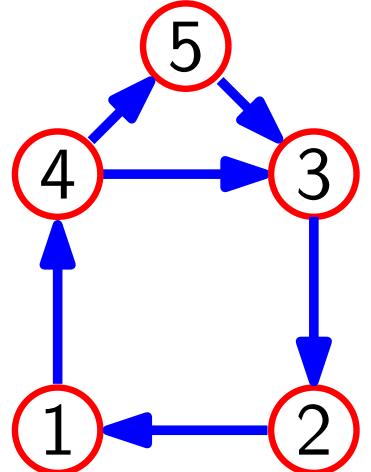
F: Wie repräsentiere ich einen Graphen?



Wir sagen: Knoten 3 und 5 sind *adjazent*.



ungerichtet
Graph



gerichtet
Graph

$$\text{Adj}[i] = \{j \in V \mid (i, j) \in E\}$$

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	0
3	0	1	0	1	1
4	1	0	1	0	1
5	0	0	1	1	0

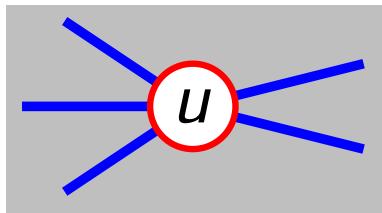
Adjazenzmatrix

	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	0	0

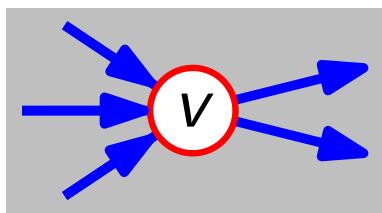
$$a_{ij} = 1 \Leftrightarrow (i, j) \in E$$

Grad eines Knotens

Def.



$$\deg u = |\text{Adj}[u]|$$



$$\text{outdeg } v = |\text{Adj}[v]|$$

$$\text{indeg } v = |\{u \in V : (u, v) \in E\}|$$

Beob.

Sei $G = (V, E)$ ein ungerichteter Graph.

Dann ist die Summe aller Knotengrade $= 2 \cdot |E|$.

Beweis.

Technik des *zweifachen Abzählens*:

Zähle alle Knoten-Kanten-Inzidenzen.

Eine Kante ist *inzident* zu ihren Endknoten.

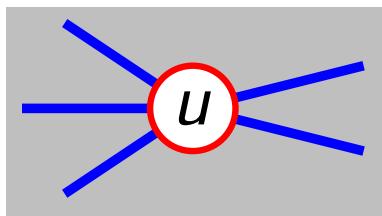
Ein Knoten ist *inzident* zu allen Kanten, deren Endknoten er ist.

Aus Sicht der Knoten: $\sum_{v \in V} \deg v$

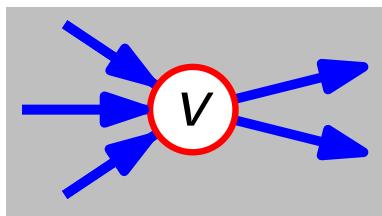
Aus Sicht der Kanten: $2 \cdot |E|$ also gleich!

Grad eines Knotens

Def.



$$\deg u = |\text{Adj}[u]|$$



$$\text{outdeg } v = |\text{Adj}[v]|$$

$$\text{indeg } v = |\{u \in V : (u, v) \in E\}|$$

Beob.

Sei $G = (V, E)$ ein ungerichteter Graph.

Dann ist die Summe aller Knotengrade $= 2 \cdot |E|$.

Sätzle.

Die Anzahl der Knoten ungeraden Grades ist gerade.

Beweis.

$$2 \cdot |E| = \sum_{v \in V} \deg v = \sum_{v \in V_{\text{ger}}} \deg v + \sum_{v \in V_{\text{ung}}} \deg v$$

gerade! *gerade!* *gerade!* \Rightarrow *gerade!*

$$\sum_{v \in V_{\text{ung}}} \deg v \text{ gerade } \Rightarrow |V_{\text{ung}}| \text{ ist gerade!}$$

□

Rundlaufstrategien für ungerichtete Graphen

1. Durchlaufe einen Graphen auf einem Kreis,
so dass jede Kante genau einmal durchlaufen wird.

Charakterisierung: Bei welchen Graphen geht das (nicht)?

Konstruktion: Wie (und in welcher Zeit) finde ich
einen solchen Rundlauf, falls er existiert?

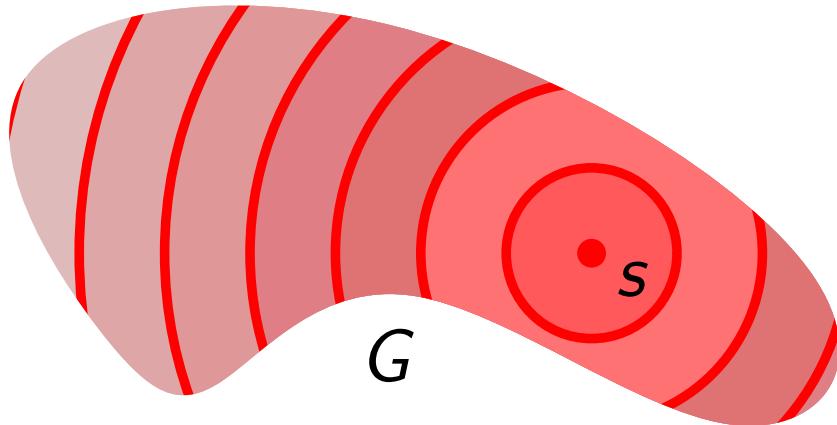
2. Durchlaufe einen Graphen auf einem Kreis,
so dass jeder **Knoten** genau einmal durchlaufen wird.

Charakterisierung: Bei welchen Graphen geht das (nicht)?

Konstruktion: Wie (und in welcher Zeit) finde ich
einen solchen Rundlauf, falls er existiert?

F: Wie durchlaufe ich einen Graphen?

Ideen?



1. wellenförmige Ausbreitung ab einem gegebenen Startknoten s
Breitensuche (breadth-first search, BFS)
2. vom Startknoten s möglichst schnell weit weg
Tiefensuche (depth-first search, DFS)

Breitensuche

BFS(Graph G , Vertex s)

Initialize(G, s)

$Q = \text{new Queue}()$

$Q.\text{Enqueue}(s)$

while not $Q.\text{Empty}()$ **do**

$u = Q.\text{Dequeue}()$

foreach $v \in \text{Adj}[u]$ **do**

if $v.\text{color} == \text{white}$ **then**

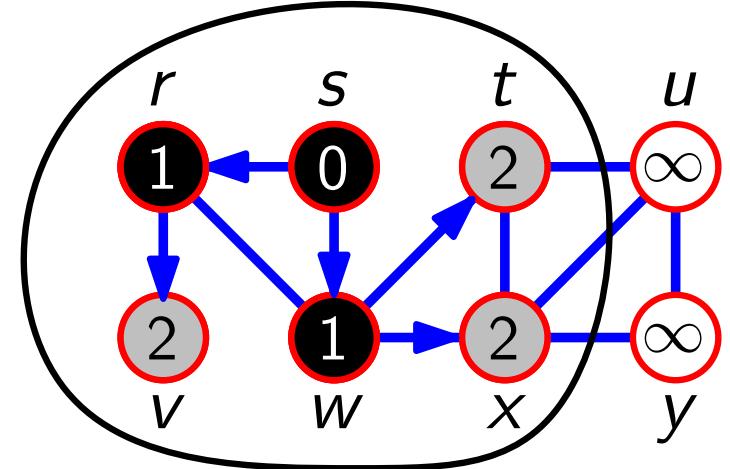
$v.\text{color} = \text{gray}$

$v.d = u.d + 1$

$v.\pi = u$

$Q.\text{Enqueue}(v)$

$u.\text{color} = \text{black}$



$w \quad r \quad | \quad t \quad x \quad v \quad | \quad | \quad | \quad | \quad usw.$

Initialize(Graph G , Vertex s)

foreach $u \in V$ **do**

$u.\text{color} = \text{white}$

$u.d = \infty$

$u.\pi = \text{nil}$

$s.\text{color} = \text{gray}$

$s.d = 0$

Laufzeit?

Initialize	En-/Dequeues	Adjazenzlisten (foreach-Schleifen)
$O(V)$	$+ O(V)$	$+ O(E) = O(V + E)$
		[Beob. über Knotengrade!]

Korrektheit von BFS – Vorbereitung

Definition. Sei $G = (V, E)$ (un)gerichteter Graph, $u, v \in V$.

$\delta(u, v) :=$ Länge eines kürzesten u - v -Wegs,
(falls v von u erreichbar; sonst $\delta(u, v) := \infty$).

Ziel:

Zeige, dass nach $\text{BFS}(G, s)$ für alle $v \in V$ gilt:

$$v.d = \delta(s, v).$$

berechneter Abstand von s *tatsächlicher Abstand von s*

Lemma 1.

(Eigenschaft kürzester Wege)

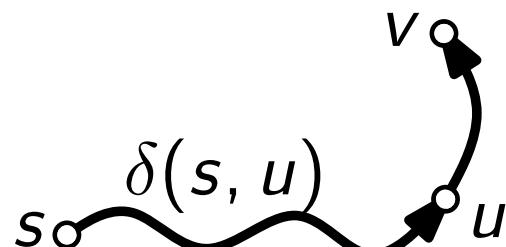
Sei $G = (V, E)$ ein (un)gerichteter Graph, $s \in V$.

Dann gilt für jede Kante $(u, v) \in E$:

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Beweis.

1. Fall: u ist von s erreichbar (d.h. \exists s - u -Weg)



Dieser s - v -Weg hat Länge $\delta(s, u) + 1$.



Kürzester s - v -Weg hat Länge $\leq \delta(s, u) + 1$.

Korrektheit von BFS – Fortsetzung

Lemma 1. Sei $s \in V$. Dann gilt für jede Kante $(u, v) \in E$:
 $\delta(s, v) \leq \delta(s, u) + 1$.



Lemma 2. Sei $G = (V, E)$ ein (un)gerichteter Graph, $s \in V$. Nach $\text{BFS}(G, s)$ gilt für alle $v \in V$: $v.d \geq \delta(s, v)$.

Beweis. Induktion über die Anz. k von Enqueue-Oper.

```
BFS(Graph G, Vertex s)
  Initialize(G, s)
  Q = new Queue()
  Q.Enqueue(s)
  while not Q.Empty() do
    u = Q.Dequeue()
    foreach v ∈ Adj[u] do
      if v.color == white then
        v.color = gray
        v.d = u.d + 1
        v.π = u
        Q.Enqueue(v)
    u.color = black
```

$k = 1$: Situation nach $Q.\text{Enqueue}(s)$:

- $s.d = 0 = \delta(s, s)$
- für alle $v \in V \setminus \{s\}$ gilt $v.d = \infty \geq \delta(s, v)$

$k > 1$: Situation nach $Q.\text{Enqueue}(v)$:

v war gerade noch weiß und ist benachbart zu u .

$v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$

Induktionsannahme für u Lemma 1
 $(u.d$ wurde gesetzt, als Anz. Enqueue-Oper. $< k)$

Jetzt ist v grau. $\Rightarrow v.d$ ändert sich nicht mehr.



Korrektheit von BFS – Fortsetzung

Lemma 2. Sei $G = (V, E)$ ein (un)gerichteter Graph, $s \in V$. Nach $\text{BFS}(G, s)$ gilt für alle $v \in V$: $v.d \geq \delta(s, v)$.



Lemma 3. Sei $Q = \langle v_1, v_2, \dots, v_r \rangle$ während BFS. Dann gilt:

- (A) $v_r.d \leq v_1.d + 1$ und
- (B) $v_i.d \leq v_{i+1}.d$ für $i = 1, \dots, r - 1$.

Also d -Werte der Knoten in Q z.B. $\langle 3, 3, 4, 4, 4 \rangle$.

Korollar. Angenommen u wird früher als v in Q eingefügt, dann gilt $u.d \leq v.d$, wenn v in Q eingefügt wird.

Beweis. Folgt aus Lemma 3 und der Tatsache, dass jeder Knoten $\leq 1 \times$ einen endlichen d -Wert bekommt.

Korrektheit von BFS – Hauptsatz

Satz. Sei G ein (un)gerichteter Graph, s ein Knoten von G . Nach $\text{BFS}(G, s)$ gilt:

- (i) Für alle Knoten $v \in V$ gilt $v.d = \delta(s, v)$.
- (ii) Jeder von s erreichbare Knoten wird entdeckt.
- (iii) Für jeden von s erreichbaren Knoten $v \neq s$ gilt:
es gibt einen kürzesten s - v -Weg, der aus einem
kürzesten s - v . π -Weg und der Kante $(v.\pi, v)$ besteht.

Beweis. (i) \Rightarrow (ii), (iii). Es genügt also (i) zu zeigen.

Lemma 2 $\Rightarrow v.d \geq \delta(s, v)$. Noch z.z.: $v.d \leq \delta(s, v)$.

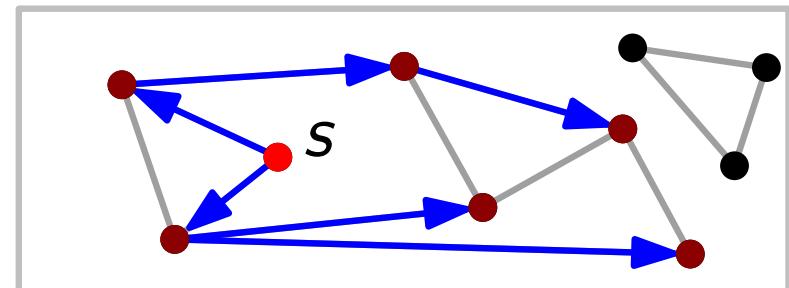
Widerspruchsbeweis mit Wahl des „kleinsten Schurken“.

Siehe Kapitel 22.2 [CLRS].

BFS-Bäume

Betrachte den *Vorgänger-Graphen* $G_\pi = (V_\pi, E_\pi)$ von G :

- $V_\pi = \{v \in V : v.\pi \neq \text{nil}\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V_\pi \setminus \{s\}\}$



Klar: G_π ist ein Baum (da zshg. und $|E_\pi| = |V_\pi| - 1$).

Beh.: G_π ist ein *Kürzeste-Wege-Baum* (oder *BFS-Baum*), d.h.

- $V_\pi = \{v \in V : v$ erreichbar von $s\}$
- für alle $v \in V_\pi$ enthält G_π einen eindeutigen Weg von s nach v , der ein kürzester s - v -Weg ist.

Bew.: Folgt aus (ii) und (iii) im Hauptsatz. □