

Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 12 (17.6.2014)

Binäre Suchbäume II
(Treaps, Rot-Schwarz-Bäume)



**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

Zufällige Binäre Suchbäume...

“Typischer” Fall:

- Falls die Schlüssel in zufälliger Reihenfolge eingefügt werden, hat der Baum Tiefe $O(\log n)$
- Operationen haben Laufzeit $O(\log n)$

Problem:

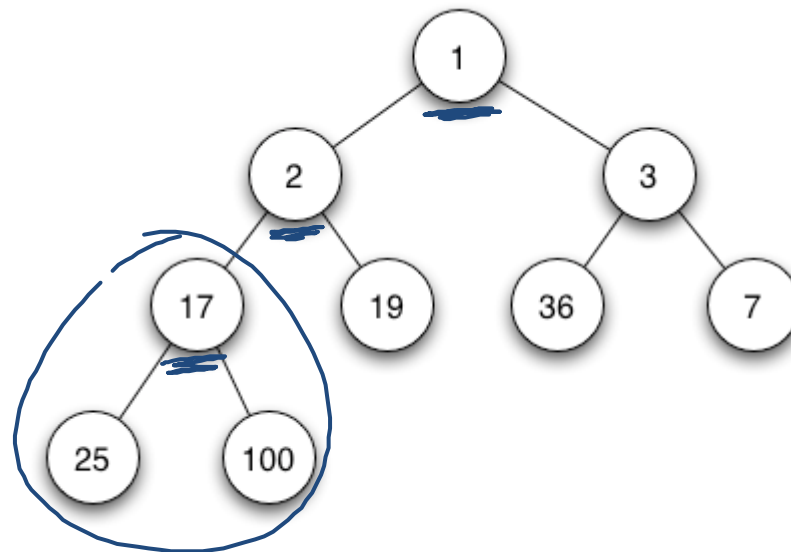
- Zufällige Reihenfolge ist nicht unbedingt der typische Fall!
- Vorsortierte Werte kann genau so typisch sein
 - Das ergibt einen sehr schlechten binären Suchbaum

Idee:

- Können wir zufällige Reihenfolge erzwingen?
- Schlüssel werden in beliebiger Reihenfolge eingefügt, aber Struktur soll immer wie bei zufälliger Reihenfolge sein!

Heap (Min-Heap) Eigenschaft:

- Gegeben ein Baum, jeder Knoten einen Schlüssel
- Ein Baum hat die Min-Heap Eigenschaft, falls **in jedem Teilbaum**, die **Wurzel** den **kleinsten Schlüssel** hat



- Heaps sind auch die “richtige” Datenstruktur, um **Prioritätswarteschlangen** zu implementieren
 - werden wir noch behandeln

Kombination Binary Search Tree / Heap

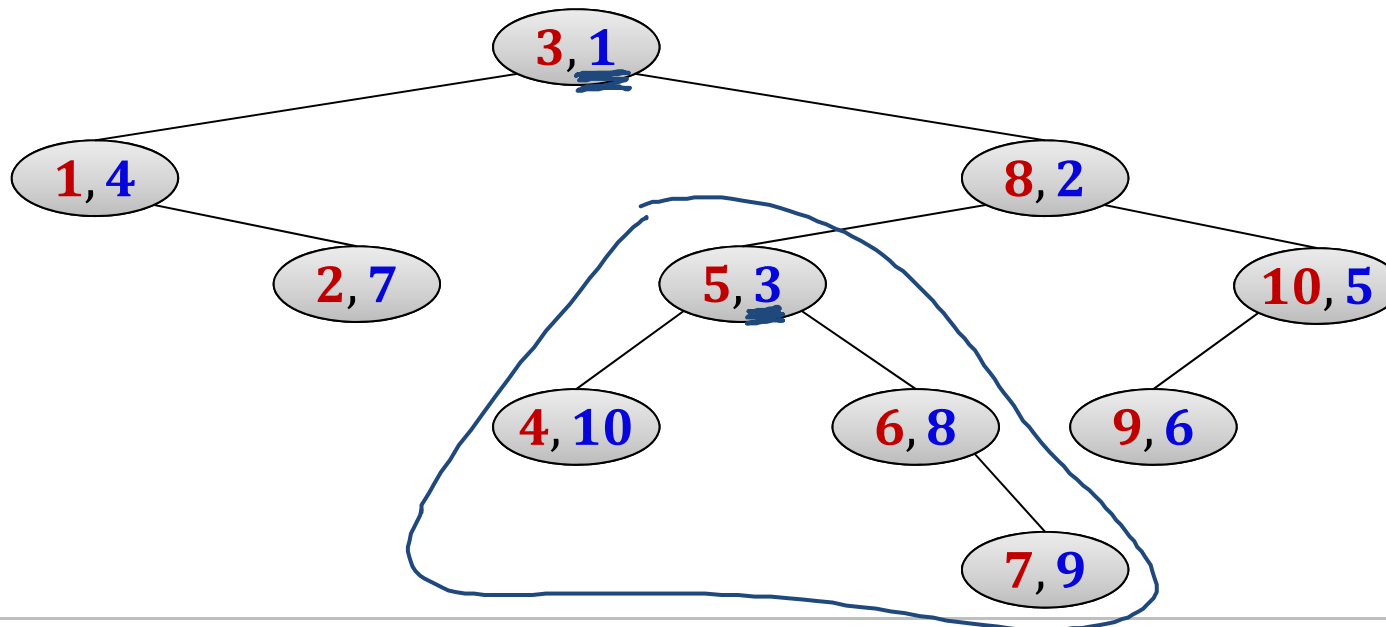
Annahme:

- Jedes Element hat zwei eindeutige Schlüssel key1 und key2

Ziel:

- Binärer Suchbaum bezüglich key1
- Einfügen in Reihenfolge, welche durch key2 gegeben ist

Beispiel: ^{4.}(1,4), ^{1.}(2,7), ^{3.}(3,1), ^{2.}(4,10), (5,3), (6,8), (7,9), (8,2), (9,6), (10,5)



Treap: Kombination BST / Heap

Annahme:

- Jedes Element hat zwei eindeutige Schlüssel $key1$ und $key2$

Treap:

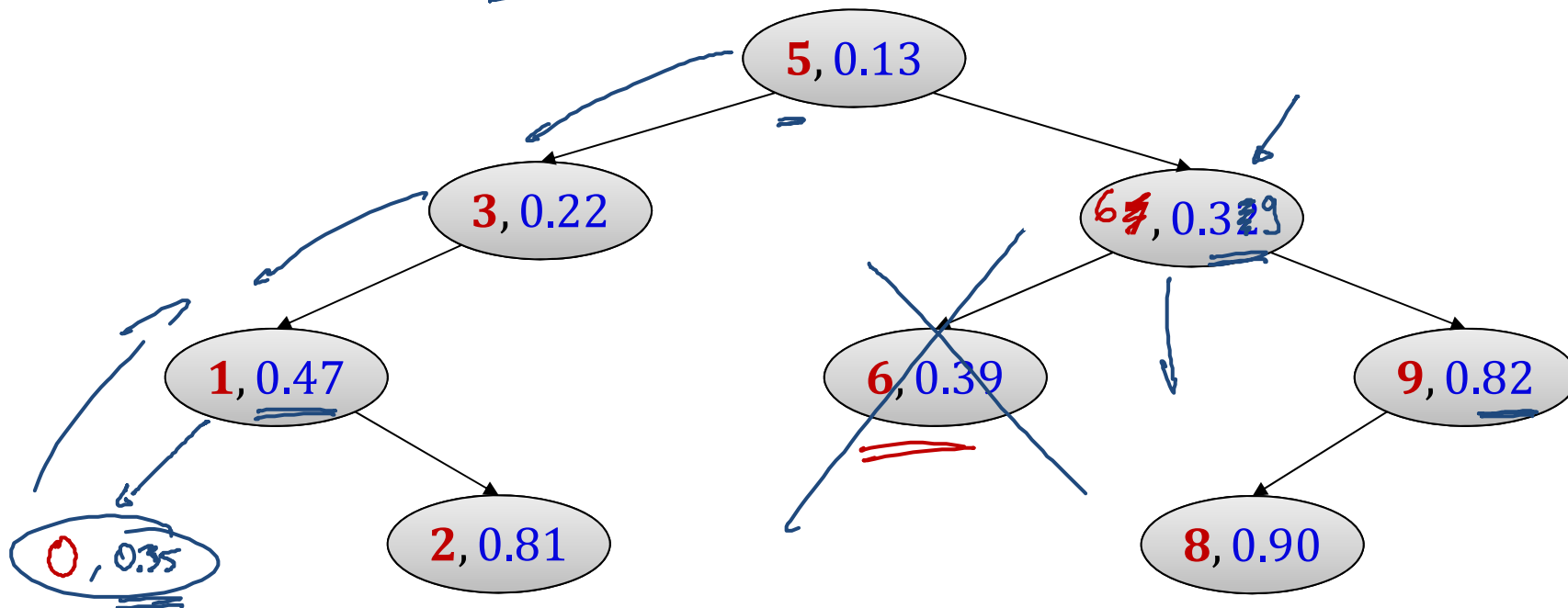
- **Binärer Suchbaum** bezüglich $key1$
- **Min-Heap** bezüglich $key2$
- Entspricht bin. Suchbaum der Schlüssel $key1$, in welchen die Schlüssel in der durch $key2$ geg. Reihenfolge eingefügt wurden

Ziel:

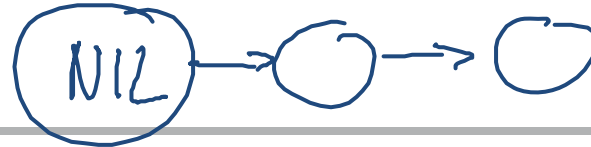
- Zu jedem Primärschlüssel ($key1$) wird zusätzlich ein zufälliger Schlüssel $key2 \in [0,1]$ bestimmt
- **Stelle bei jedem insert / delete sicher, dass der Baum ein Treap bezüglich der Schlüssel $key1$ und $key2$ ist!**
- Entspricht bin. Suchbaum mit zufälliger Einfügereihenfolge

Treap: Einfügen und Löschen

- Einfügen und Löschen funktioniert erstmal gleich, wie bei einem normalen binären Suchbaum
- Allerdings kann dann allenfalls die Min-Heap-Eigenschaft bezüglich key2 nicht mehr erfüllt sein
- Beispiel: $\text{insert}(0, 0.35)$, $\text{delete}(7)$



Sentinel



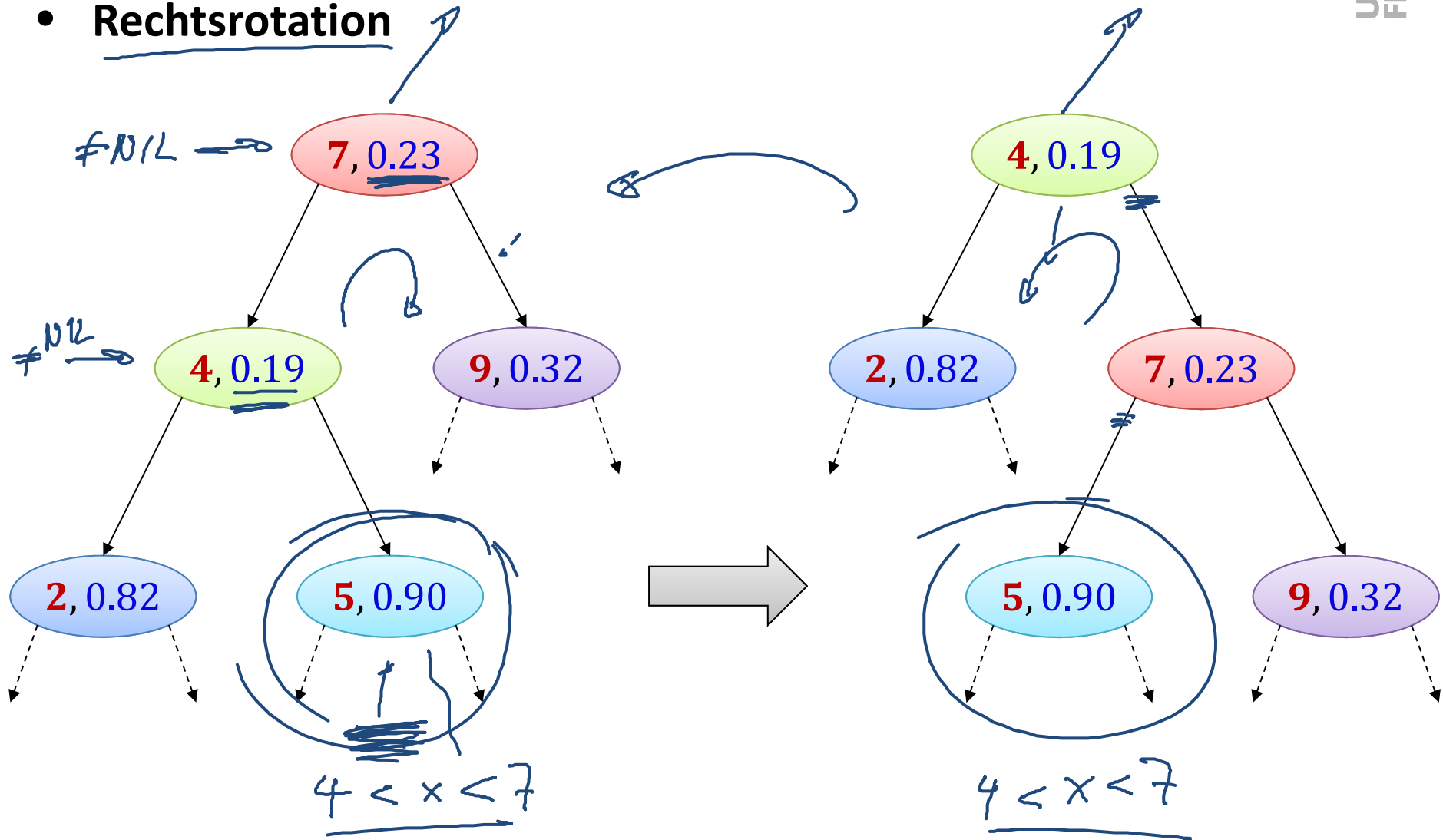
Um den Code zu vereinfachen...

Sentinel-Knoten: NIL

- Ersetzt alle NULL-Pointer
- *NIL.key1* ist nicht definiert (wird nie gesetzt oder gelesen)
- *NIL.key2* = ~~2.0~~^{-1.0} (und damit immer grösser als alle anderen)
 - wird nie verändert ←
- *NIL.left*, *NIL.right*, *NIL.parent* können beliebig gesetzt sein
 - Wir müssen darauf achten, dass sie nie ausgelesen werden
 - Wenn es den Code vereinfacht, kann man *NIL.parent*, ... neu setzen

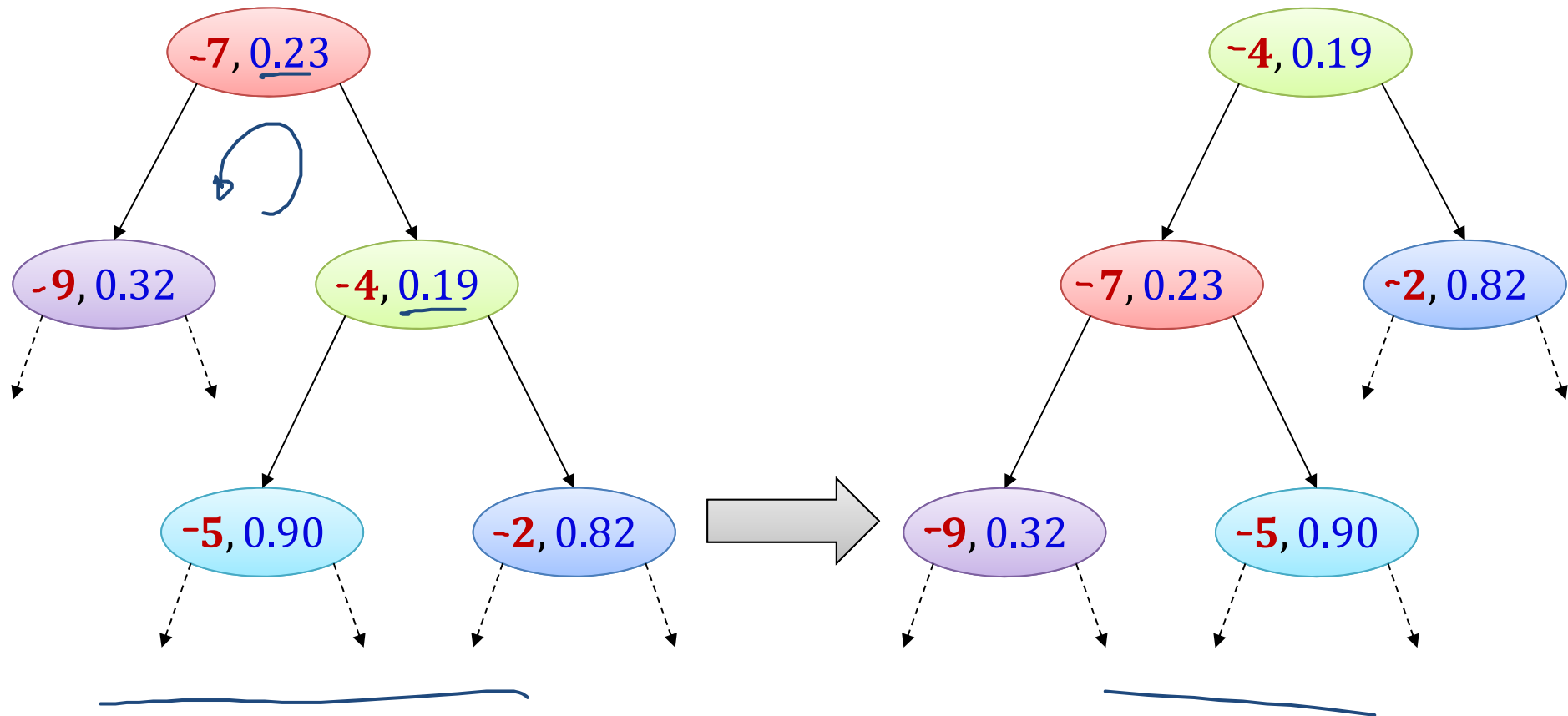
Rotationen

- Rechtsrotation



Rotationen

- Linksrotation



Rechtsrotation

right-rotate(u, v):

`u.left = v.right` ←

`u.left.parent = u` ←

`v.right = u` ← *w = u.parent*

`u.parent = v` ←

if `u == root` then

`root = v` ←

else

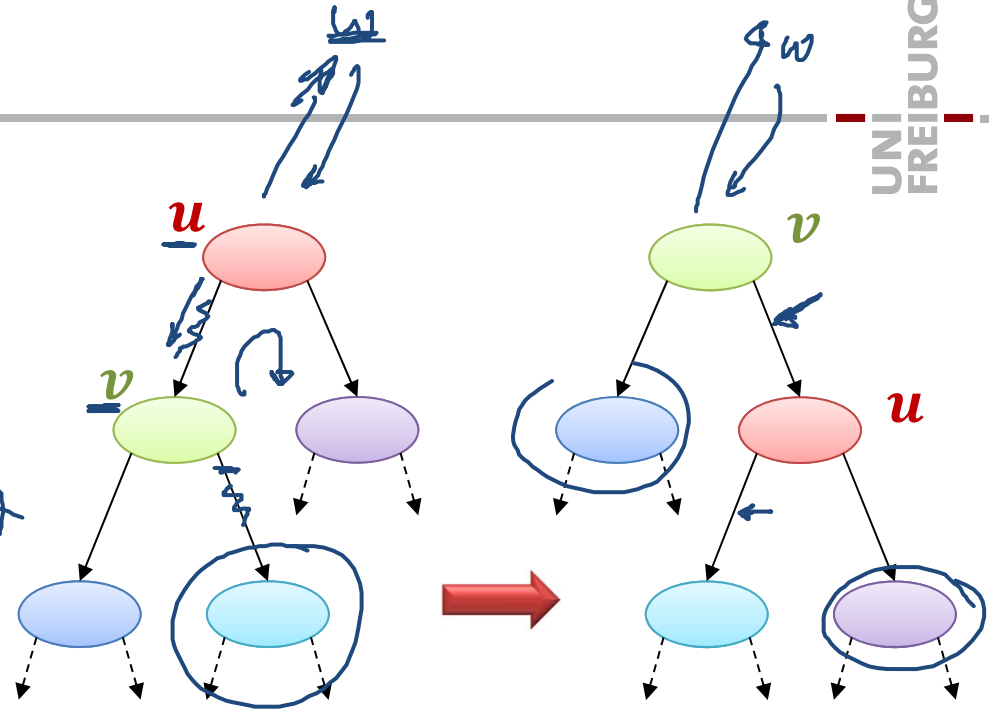
 if `u = u.parent.left` then

~~`u.parent`~~.left = `v`

 else

~~`u.parent`~~.right = `v`

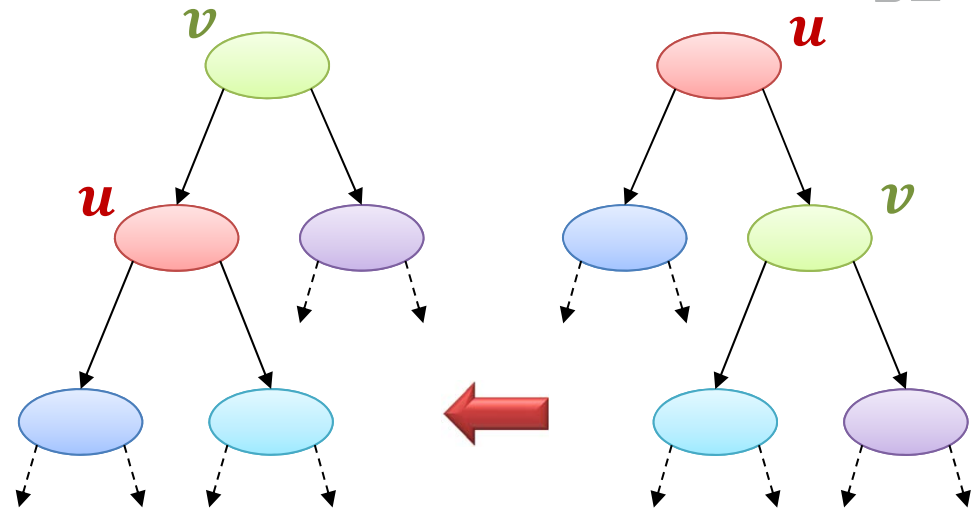
`v.parent = u.parent`



Linksrotation

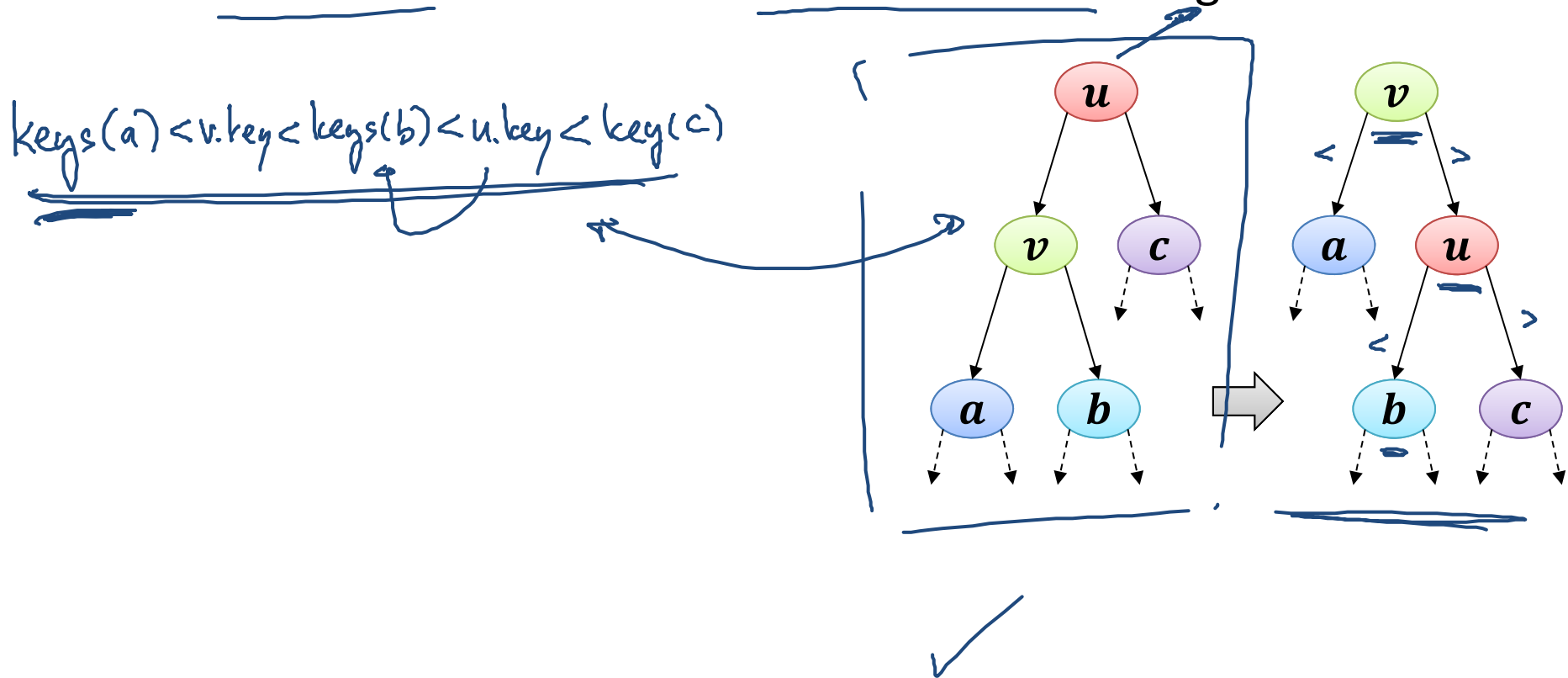
left-rotate(u,v):

```
{ u.right = v.left
  u.right.parent = u
  v.left = u u = u.parent
  u.parent = v
  if u == root then
    root = v
  else
    if u = u.parentw.left then
      u.parentw.left = v
    else
      u.parentw.right = v
  v.parent = u.parentw
```



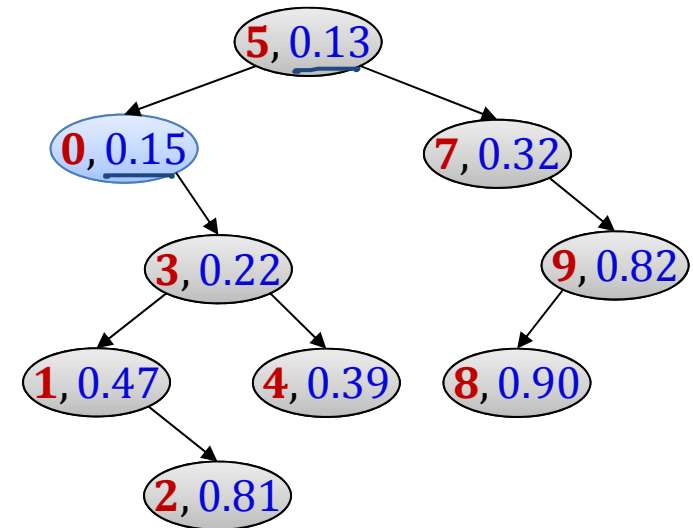
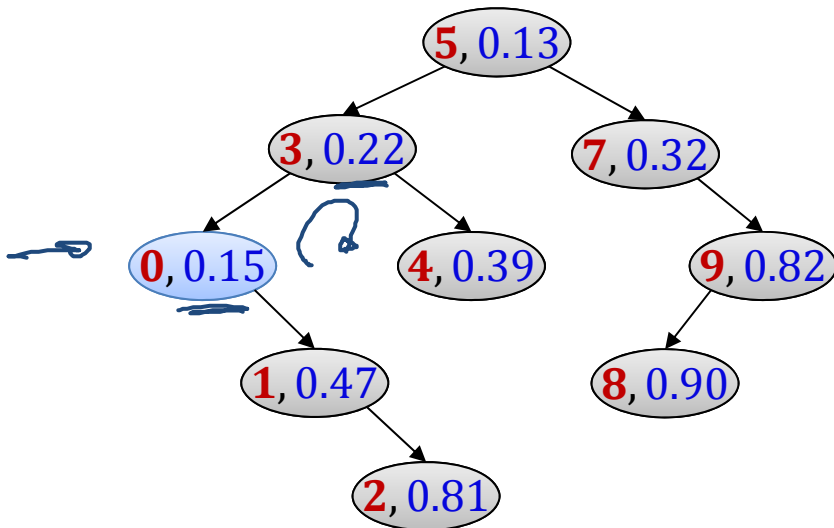
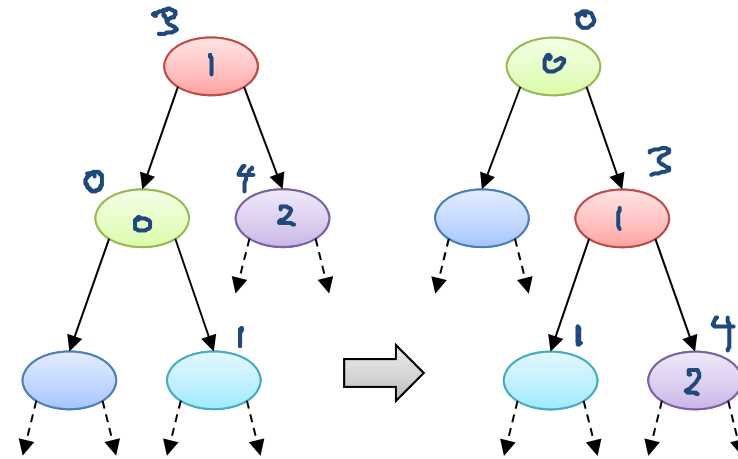
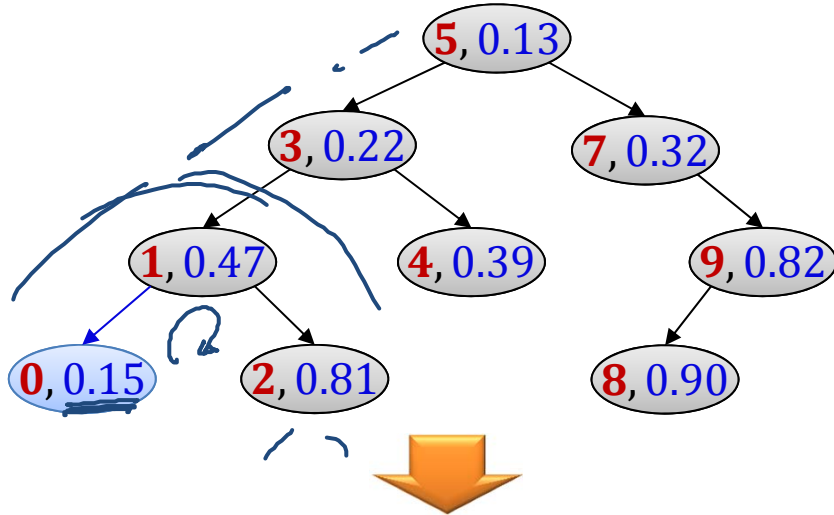
Korrektheit: Rotationen

Lemma: Rotationen erhalten die "Bin. Search Tree"-Eigenschaft



Treap: Einfügen

- Beispiel: $insert(0, 0.15)$



Treap: Einfügen

root.parent = NIL

Ziel: Füge Schlüssel x ein ($key1$)

$NIL.key = \text{~~0~~ } -1.0$

1. Bestimme zufälligen zweiten Schlüssel y ($key2$)
2. Füge (x, y) wie üblich in den bin. Suchbaum (bez. x) ein
3. Annahme:

Knoten mit Schlüssel x heisst v , $u = v.parent$

$u.key2 = -1.0$

```
while  $v.key2$  <  $u.key2$  do
  if  $v == u.left$  then
    right-rotate( $u, v$ )
  else
    left-rotate( $u, v$ )
 $u = v.parent$ 
```

\uparrow
NIL



Treap insert(x)

```
// assumption: treap is not empty
v = root; y = random(0,1)
while v != NIL and v.key1 != x do
  if v.key1 > v.x then
    if v.left == NIL then
      w = new TreeNode(x,y,v,NIL,NIL); v.left = w
      v = v.left
    else
      if v.right == NIL then
        w = new TreeNode(x,y,v,NIL,NIL); v.right = w
        v = v.right
```

w.key1
w.key2
w.parent
w.left w.right

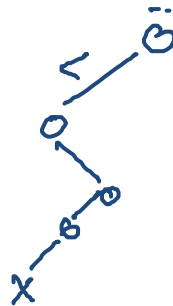
```
// v now is the node with v.key1 == x
u = v.parent
while v.key2 < u.key2 do
  if v == u.left then
    right-rotate(u,v)
  else
    left-rotate(u,v)
  u = v.parent
```

Treap: Einfügen, Korrektheit

Lemma: Der Treap ist zu jeder Zeit während dem Einfügen von x ein korrekter binärer Suchbaum bezügl. $key1$

Beweis:

- Der neue Knoten wird am Anfang wie bisher eingefügt
 - da, wo man bei der Suche nach x auf NIL trifft



- Rotationen erhalten die BST-Eigenschaft

Treap: Terminierung

Annahme: Nach dem Einflügen von x ist der Schlüssel x im Knoten v gespeichert und der Pfad von der Wurzel zu v ist:

$$root = u_1, u_2, \dots, u_k, v$$

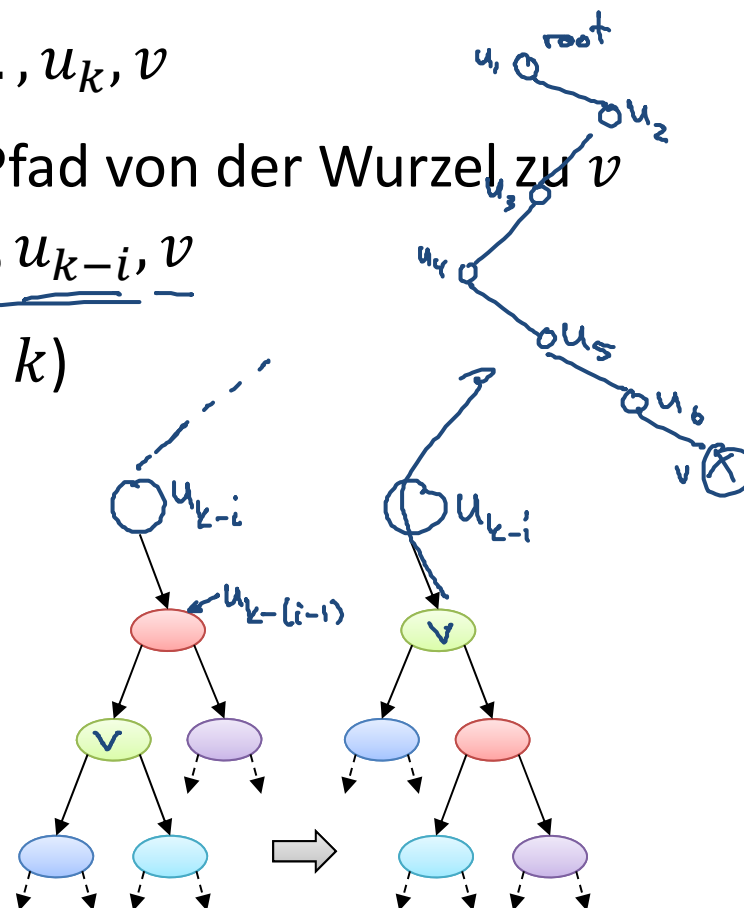
Lemma: Nach $i \leq k$ Rotationen ist der Pfad von der Wurzel zu v

$$root = u_1, u_2, \dots, u_{k-i}, v$$

Beweis: Per Induktion (für alle $i = 0, \dots, k$)

- Verankerung $i = 0$ ✓
- Induktionsschritt:

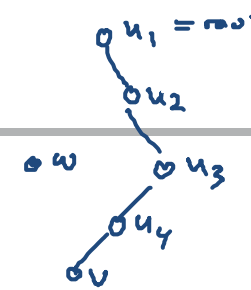
$$i-1 \rightarrow i \quad \checkmark$$



Korollar: Alg. terminiert nach $\leq k$ Rotationen

Treap: Heap-Eigenschaft

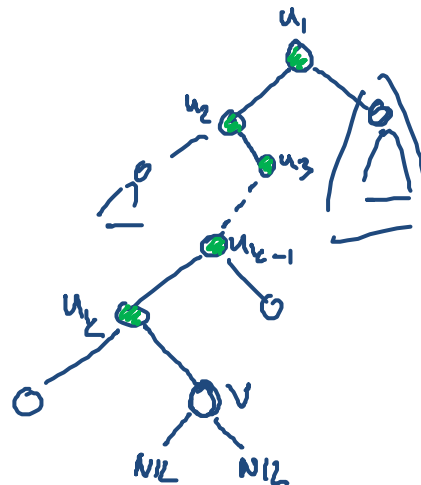
Lemma: Nach $i \leq k$ Rotationen:



- 1) Pfad von der Wurzel zu v : $root = u_1, u_2, \dots, u_{k-i}, v$
- 2) Teilbäume aller Knoten $w \neq u_1, \dots, u_{k-i}$ erfüllen Heap-Eigenschaft
- 3) Für alle Knoten $u_j, j \in \{1, \dots, k - i\}$ und alle Knoten $w \neq v$ im Teilbaum von u_j gilt: $u_j.key_2 \leq w.key_2$

Beweis: Induktion (für alle $i = 0, \dots, k$)

- Verankerung: $i=0$



Treap: Heap-Eigenschaft

Lemma: Nach $i \leq k$ Rotationen:

- 1) Pfad von der Wurzel zu v : $root = u_1, u_2, \dots, u_{k-i}, v$
- 2) Teilbäume aller Knoten $w \neq u_1, \dots, u_{k-i}$ erfüllen Heap-Eigenschaft
- 3) Für $u_j, j \in \{1, \dots, k - i\}$ und $w \neq v$ im Teilbaum von u_j gilt: $u_j.key2 \leq w.key2$

Beweis: Induktion (für alle $i = 0, \dots, k$)

- Induktionsschritt:

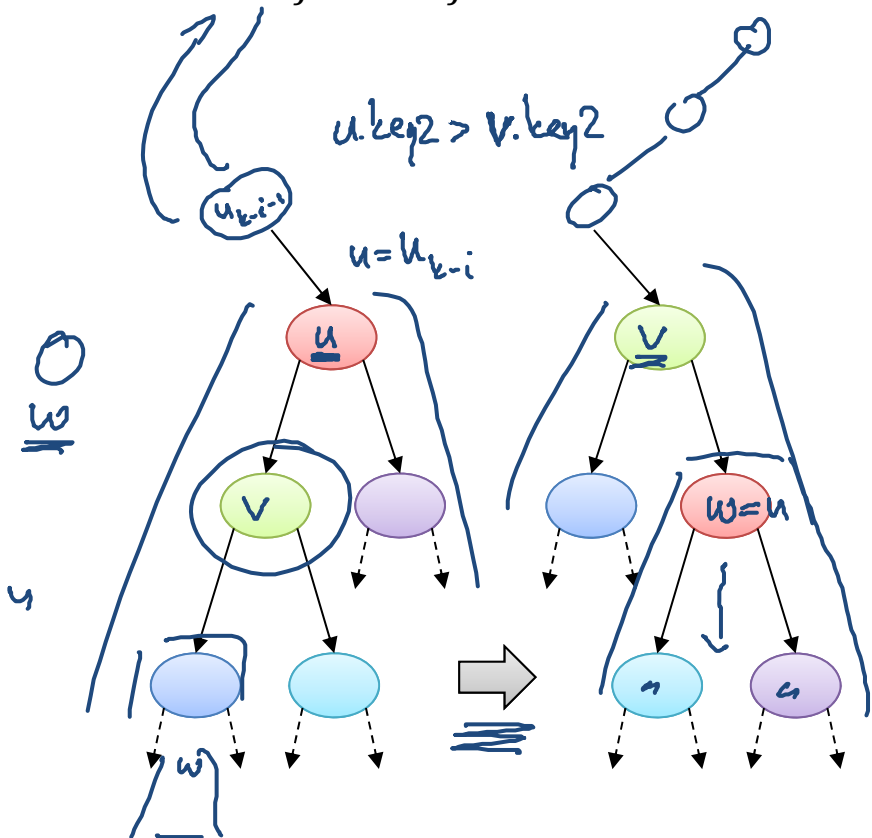
$$i \rightarrow i+1$$

- 2) w ist im Teilbaum von u
($w \neq u$)

w 's Teilbaum wird nicht verändert

$w = u$: nach Rot. erfüllt Teilbaum von u die Min-heap-Eigenschaft

- 3) Ind. - Annahme 3)



Treap: Einfügen

Theorem: Beim Einfügen eines Schlüssels bleibt die Treap-Eigenschaft erhalten. Laufzeit des Einfügens: $O(\text{Tiefe})$ des Baums

bin. search tree ✓

heap-Eigenschaft: Lemma auf der letzten Folie
+ Abbruchbed. beim Rotieren

Laufzeit ✓

Korollar: Einfügen hat mit hoher Wahrscheinlichkeit Laufzeit $O(\log n)$.

Bin. Suchbaum : Struktur, wie bei zufälligem Einfügen

Treap: Löschen

Ziel: Lösche Schlüssel x

Gehe wie üblich vor:

1. Suche Knoten v mit Schlüssel x

2. Falls v ein Blatt ist oder nur 1 Kind hat, lösche v

3. Falls v zwei Kinder hat:

– Finde Vorgänger u von v (grösster Schlüssel im linken Teilbaum)

• alternativ: finde Nachfolger...

– $\overset{v}{u}.key1 = \overset{u}{v}.key1$; $\overset{v}{u}.key2 = \overset{u}{v}.key2$

– **Dieser Schritt kann die Min-Heap Eigenschaft bezügl. $key2$ verletzen!**

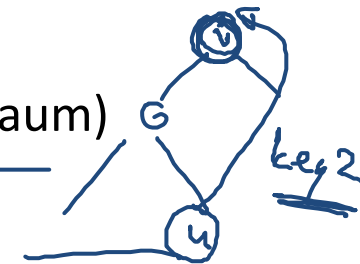
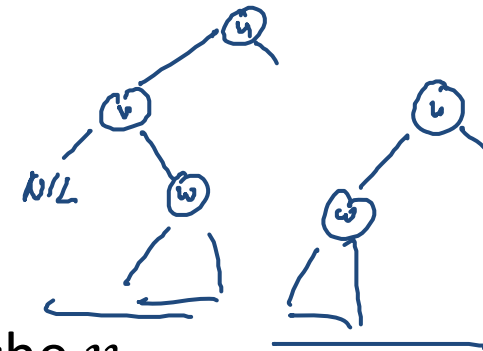
– Lösche Knoten u (u hat höchstens 1 Kind)

– Falls nötig, stelle Min-Heap Eigenschaft mit Hilfe von Rotationen wieder her

– Mit jeder Rotation wandert der "Fehler" weiter von der Wurzel weg

– **Anzahl Rotationen \leq Tiefe des Baums (nach dem Löschen)**

– **Details: Übungsaufgabe**



Treap: Eigenschaften

- Zwei Schlüssel $key1$ und $key2$
 - $key1$ ist der normale Schlüssel für die Dictionary-Operationen
 - $key2$ ist ein zufälliger Schlüssel, um den Baum balanciert zu halten
- Immer ein bin. Suchbaum bez. $key1$ und ein Min-Heap bez. $key2$

Solange $key2$ unabhängig von allen $insert/delete$ -Operationen gewählt wird, hat der Baum immer die Struktur eines binären Suchbaums mit zufälliger Einfügereihenfolge.

- **Mit hoher Wahrscheinlichkeit: $Tiefe \in \Theta(\log n)$**

Die Operationen $find$, $insert$, $delete$, min , max , $succ$, $pred$ haben alle

$Laufzeit O(\log n)$

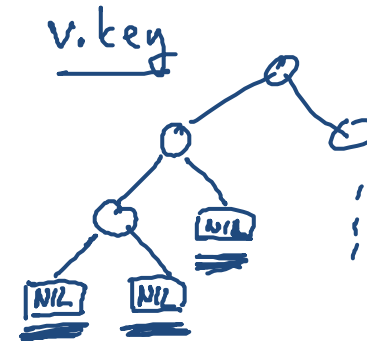
- Wie man das auch deterministisch garantieren kann, werden wir als nächstes sehen...

Ziel: Binäre Suchbäume, welche immer balanciert sind

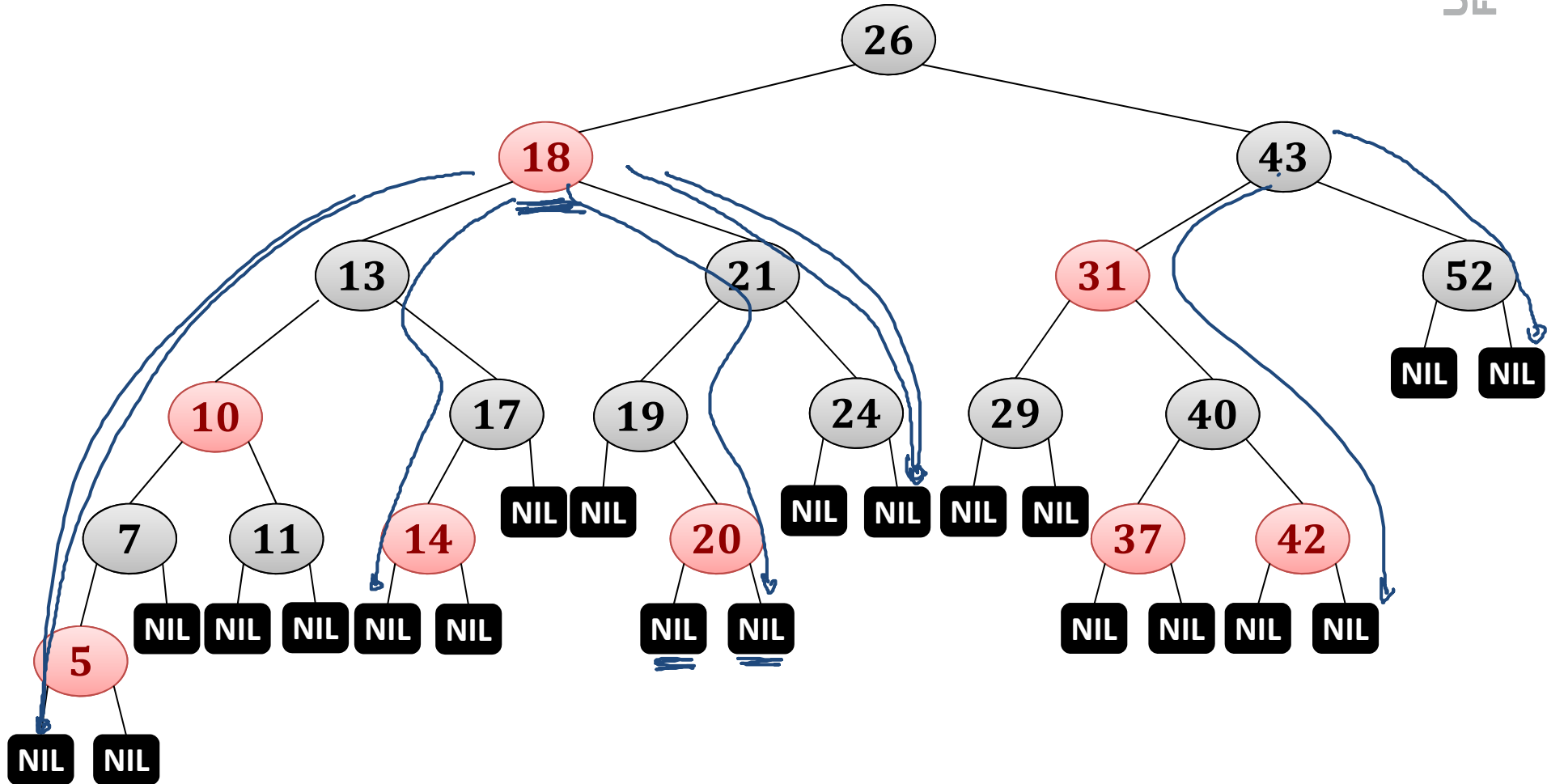
- balanciert, intuitiv: in jedem Teilbaum, links & rechts \approx gleich gross
- balanciert, formal: Teilbaum mit k Knoten hat Tiefe $O(\log k)$

Rot-Schwarz-Bäume sind binäre Suchbäume, fuer welche

- 1) Alle Knoten sind rot oder schwarz
- 2) Wurzel ist schwarz
- 3) Blätter (= NIL-Knoten) sind schwarz
- 4) Rote Knoten haben zwei schwarze Kinder
- 5) Von jedem Knoten v aus, haben alle (direkten) Pfade zu Blättern (NIL) im Teilbaum von v die gleiche Anzahl schwarze Knoten



Rot-Schwarz-Bäume: Beispiel

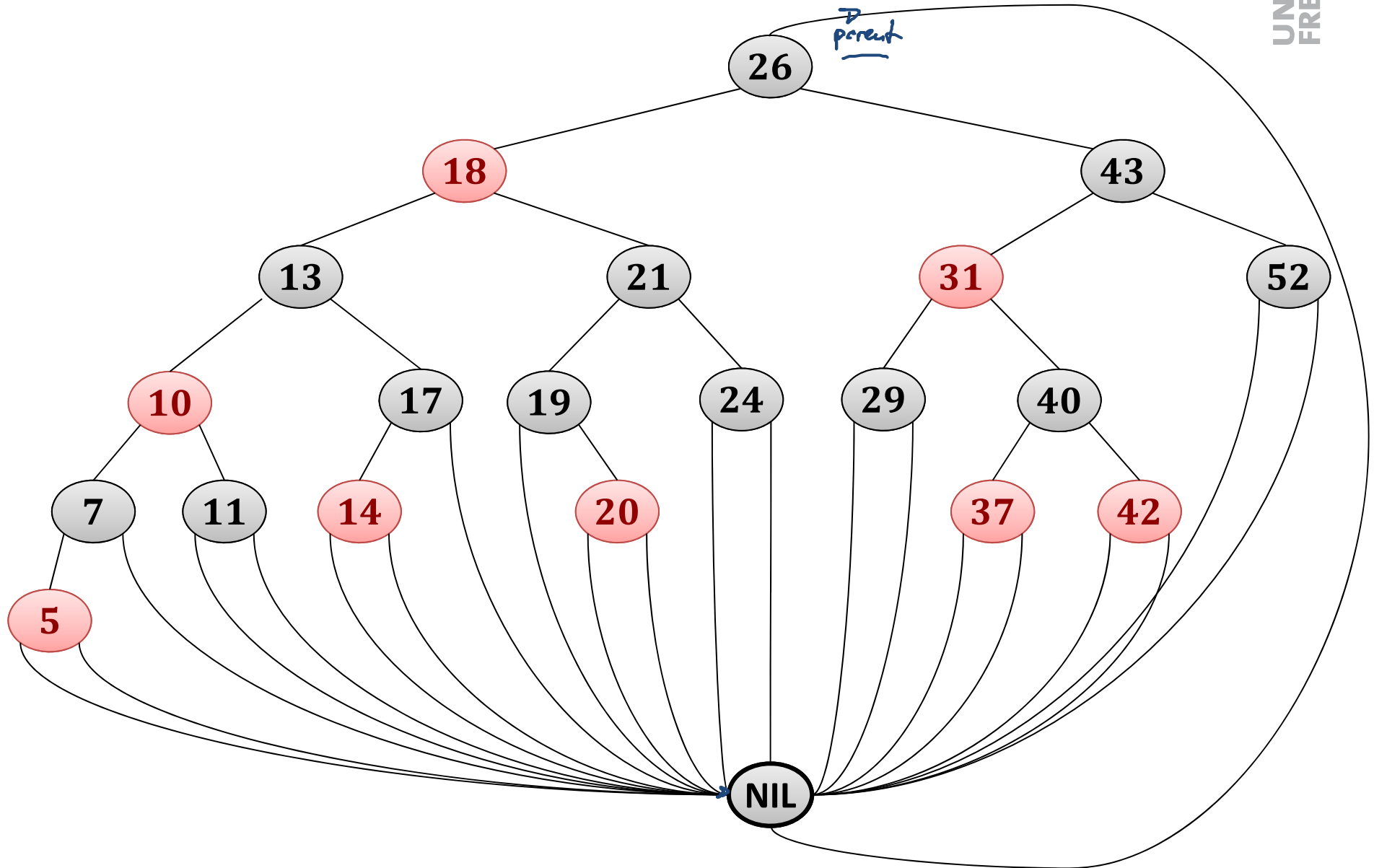


Wie bei den Treaps, um den Code zu vereinfachen...

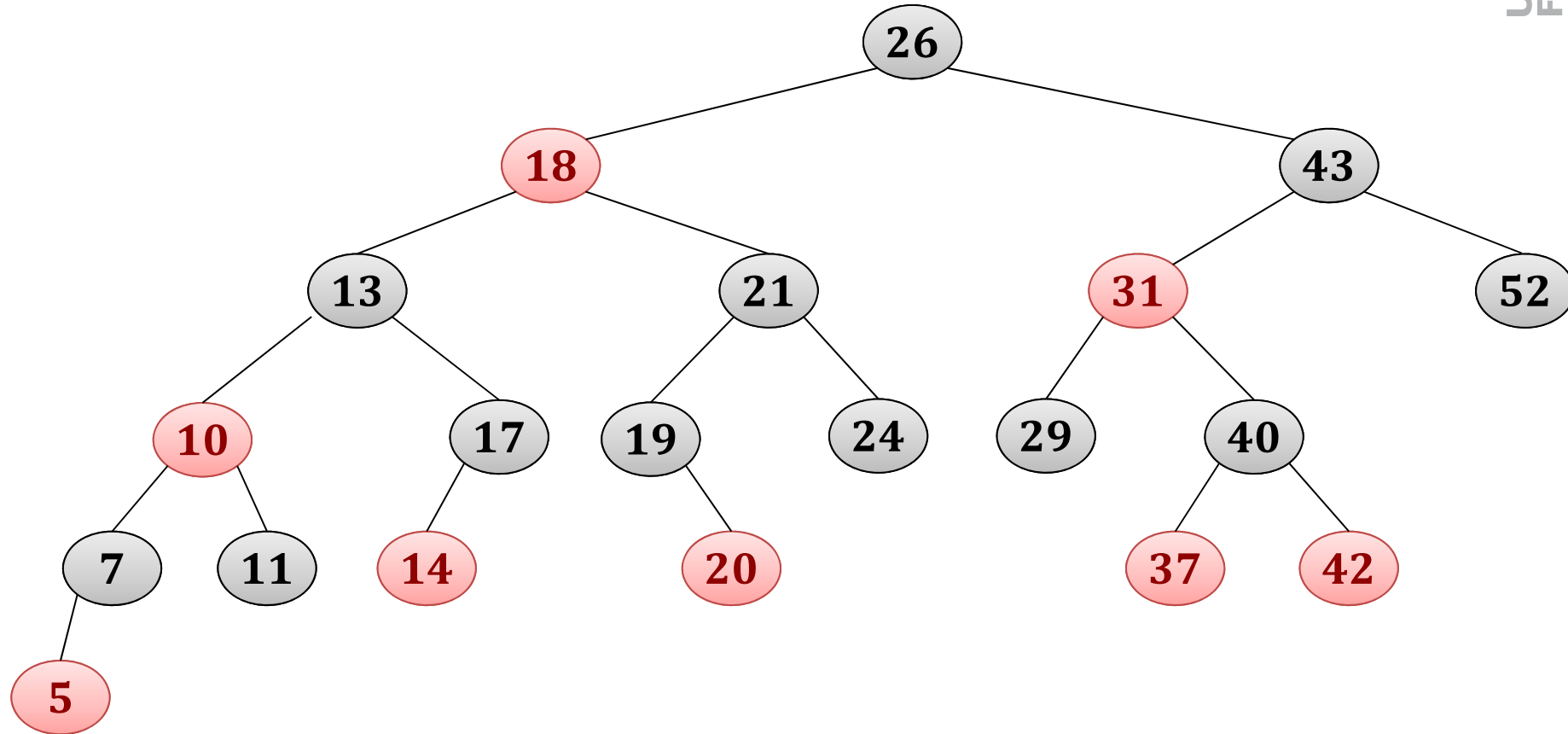
Sentinel-Knoten: *NIL*

- Ersetzt alle NULL-Pointer
- *NIL.key* ist nicht definiert (wird nie gesetzt oder gelesen)
- *NIL.color* = *black*
 - Als Blätter des Baumes verstehen wir die NIL-Knoten (sind alle schwarz)
 - repräsentiert alle Blätter des Baumes
- *NIL.left*, *NIL.right*, *NIL.parent* können beliebig gesetzt sein
 - Wir müssen darau achten, dass sie nie ausgelesen werden
 - Wenn es den Code vereinfacht, kann man *NIL.parent*, ... neu setzen

Rot-Schwarz-Bäume: Sentinel



Rot-Schwarz-Bäume: Repräsentation

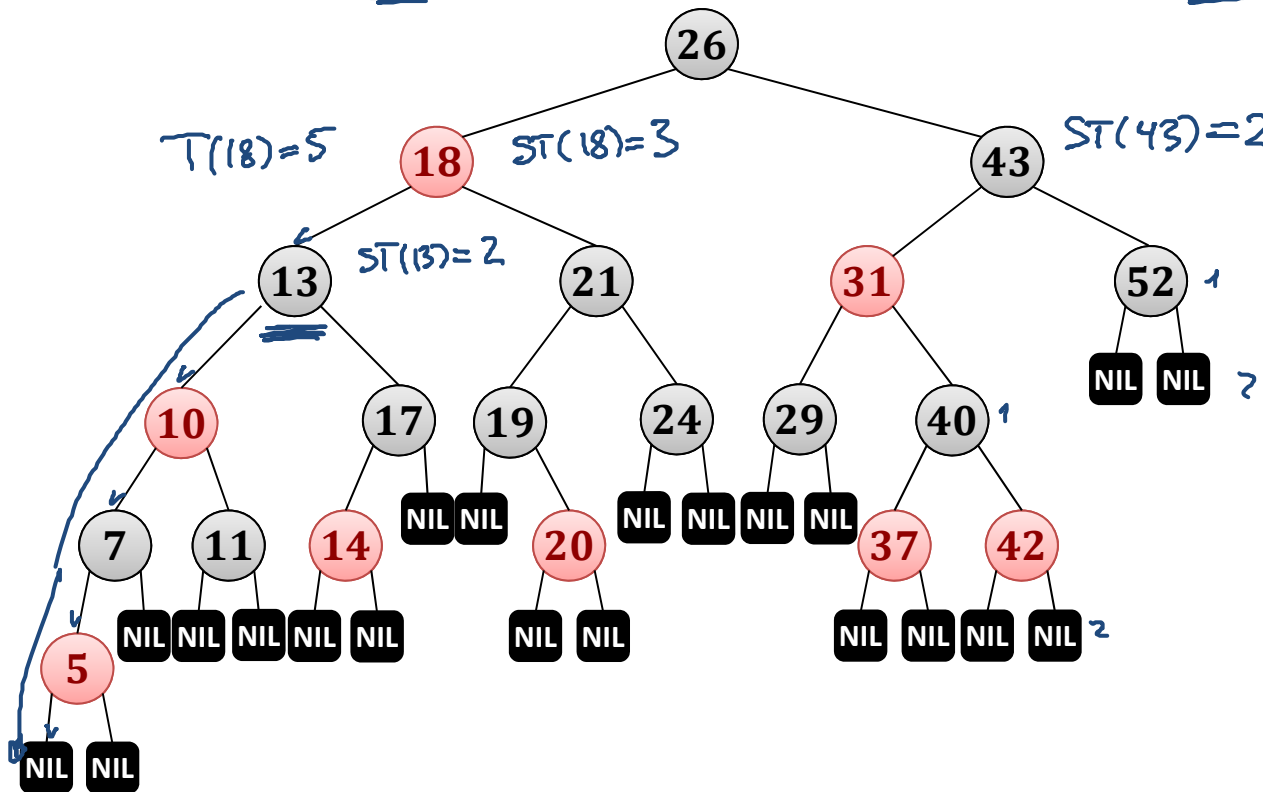


Tiefe / Schwarz-Tiefe

Definition: Die **Tiefe (T)** eines Knoten v ist die maximale Länge eines direkten Pfades von v zu einem Blatt (NIL).

Definition: Die **Schwarz-Tiefe (ST)** eines Knoten v ist die Anzahl schwarzer Knoten auf jedem direkten Pfad von v zu einem Blatt (NIL)

- Der Knoten v wird dabei nicht gezählt, das Blatt (NIL, falls $\neq v$) jedoch schon!



$ST(v) \leq T(v)$

$T(v) \leq 2ST(v)$

