Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 10 (28.5.2018)

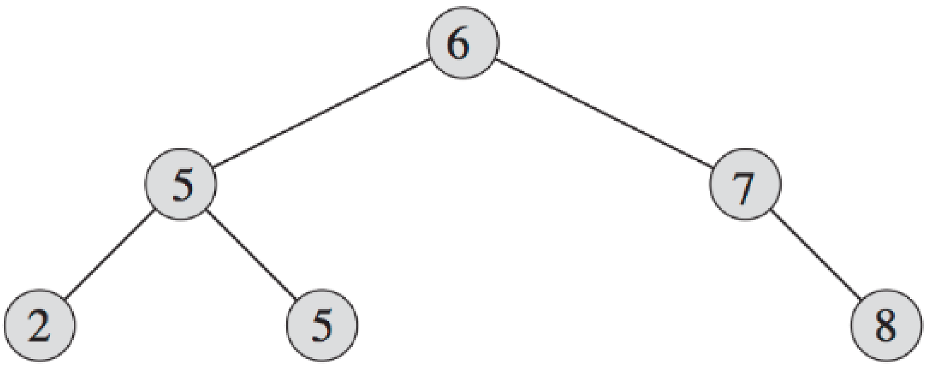
Binäre Suchbäume II

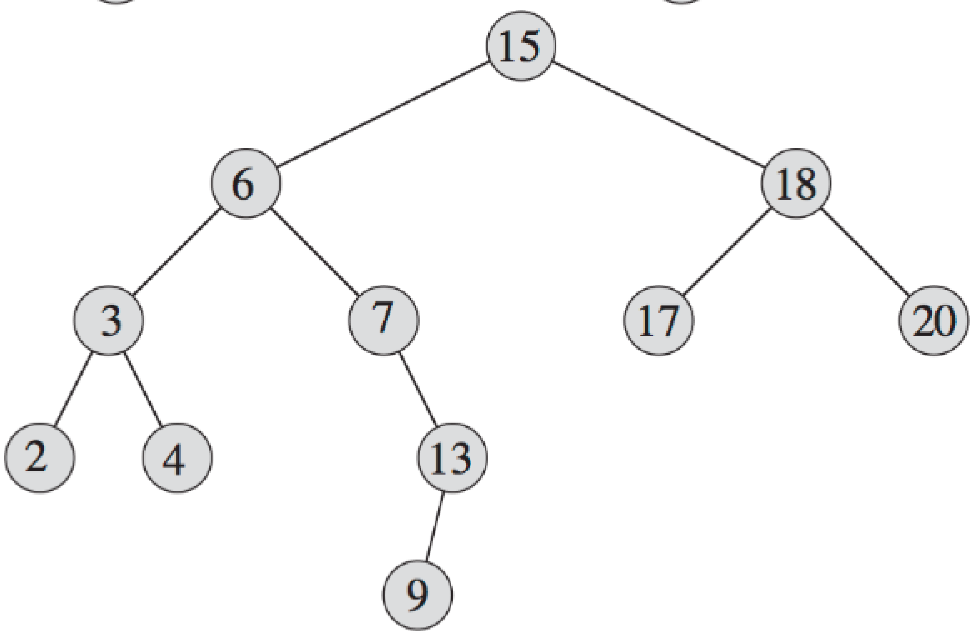
Fabian Kuhn
Algorithmen und Komplexität

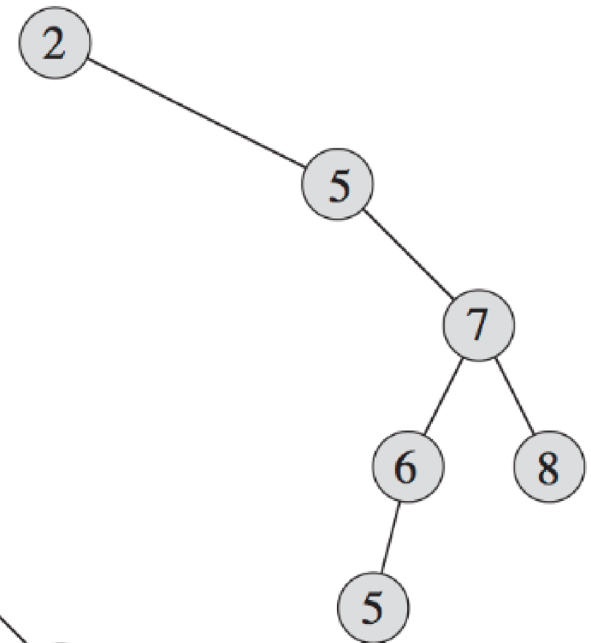
UNI FREIBURG

Binäre Suchbäume

- Binäre Suchbäume müssen nicht immer so schön symmetrisch sein…



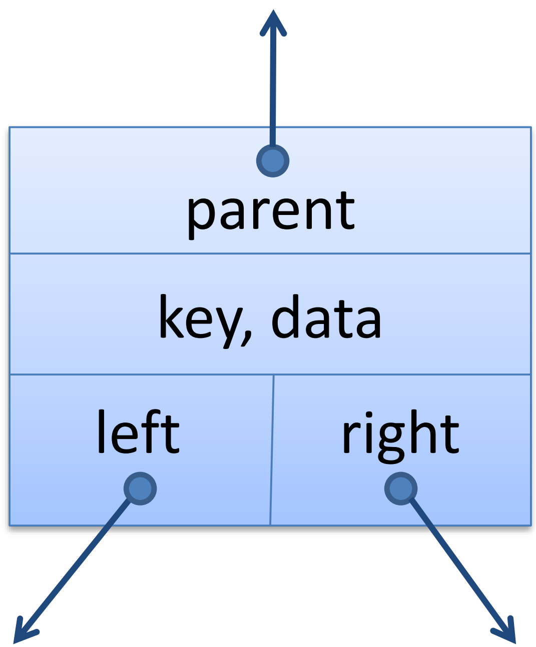




Quelle: [CLRS]

Binärer Suchbaum : Elemente

TreeElement:

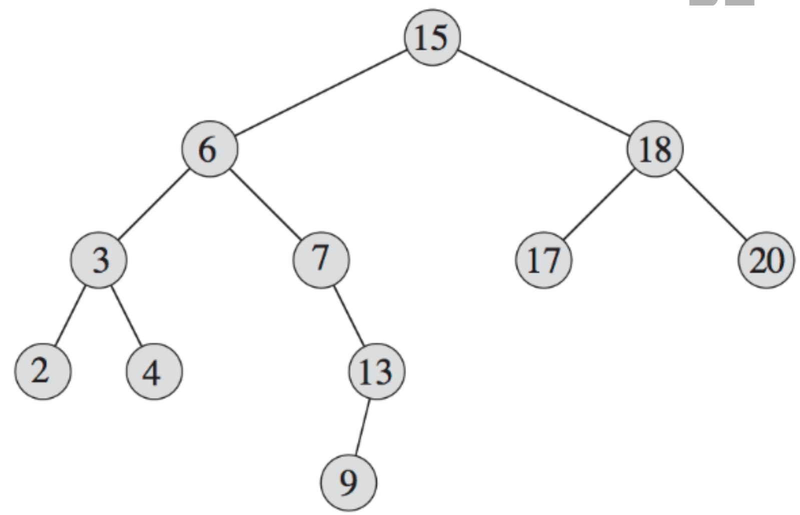


Implementierung: gleich wie bei den Listen-Elementen

Suche in einem binären Suchbaum

Suche nach Schlüssel x

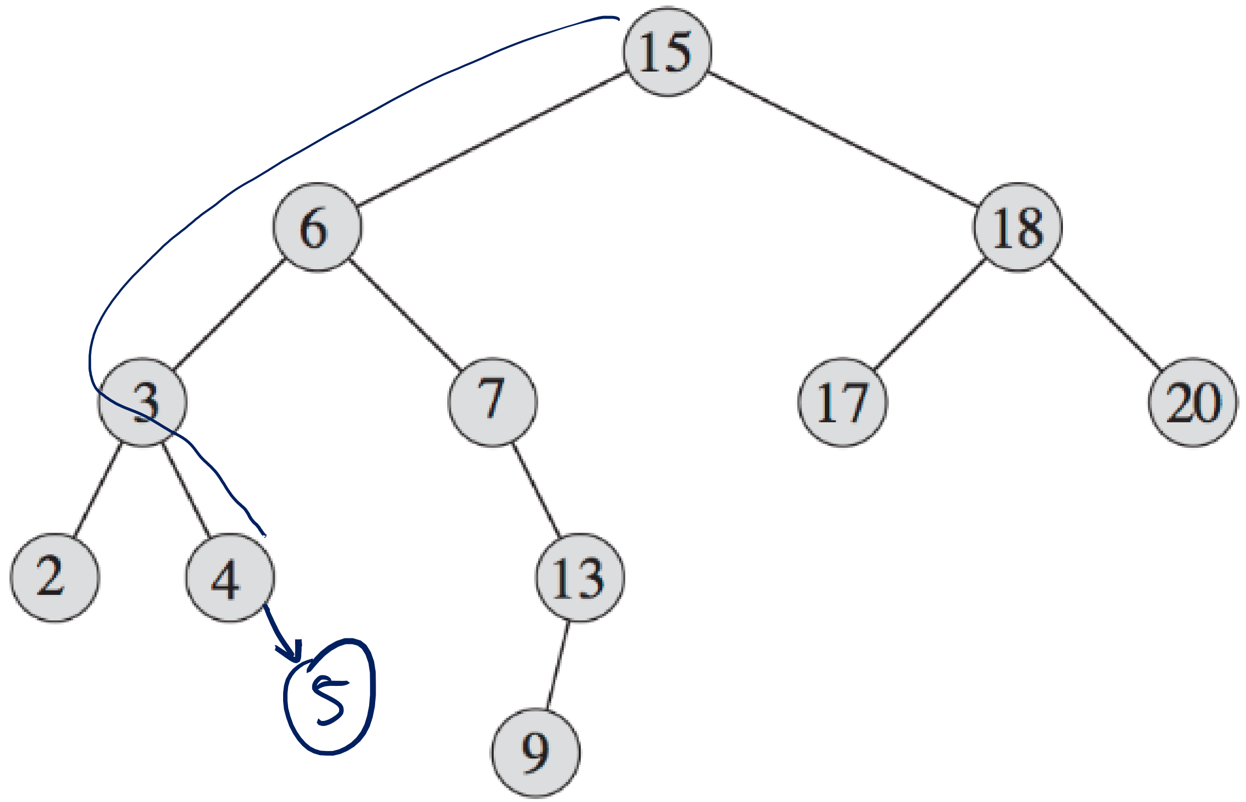
- Benutze binäre Suche
  (darum heisst's binärer Suchbaum…)




```
current = root
while current is not None and current.key != x:
    if current.key > x:
        current = current.left
    else:
        current = current.right
```

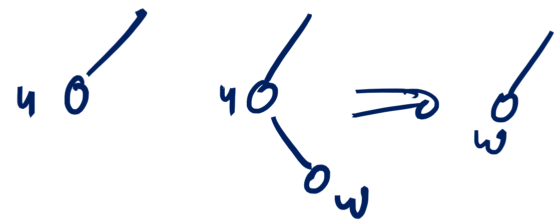
Einfügen eines Schlüssels

Füge Schlüssel 1, 5, 14, 6.5, 19 ein…

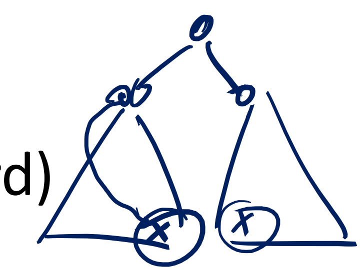


Löschen eines Schlüssels

Lösche Schlüssel x:



1. Finde Knoten u mit $u.key = x$
   – wie üblich mit binärer Suche
2. Falls u nicht 2 Kinder hat, lösche Knoten u
   – Annahme: v ist Parent von u, u ist linkes Kind von v (anderer Fall analog)
   – Fall u ein Blatt ist, wird $v.left =$ None
   – Falls u ein Kind w hat, wird $v.left = w$
3. Falls u zwei Kinder hat, dann bestimme Vorgängerknoten v
   – Funktioniert auch mit Nachfolgerknoten
4. Setze $u.key = v.key$ und $u.data = v.data$
5. Lösche Knoten v (gleich, wie oben u gelöscht wird)
   – Knoten v hat höchstens 1 Kind!



Laufzeit Binärer Suchbaum

Die Operationen

find, min, max, predecessor, successor, insert, delete

haben alle **Laufzeit O(Tiefe des Baums)**.

Was ist die Tiefe eines binären Suchbaums?

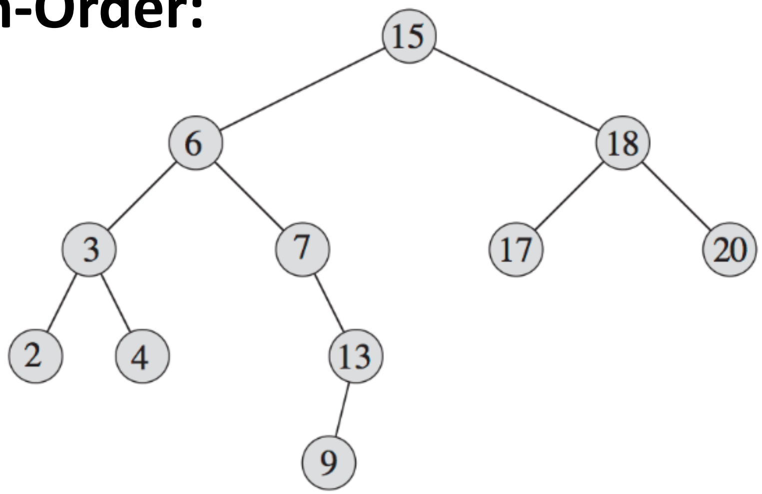
worst-case : $O(n)$

typischer Fall : $O(\log n)$

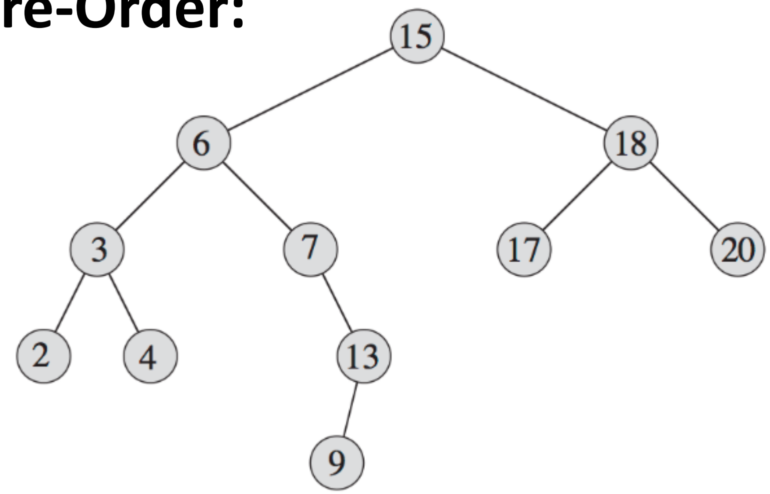
Traversieren eines binären Suchbaums

Ziel: Besuche alle Knoten eines binären Suchbaums einmal

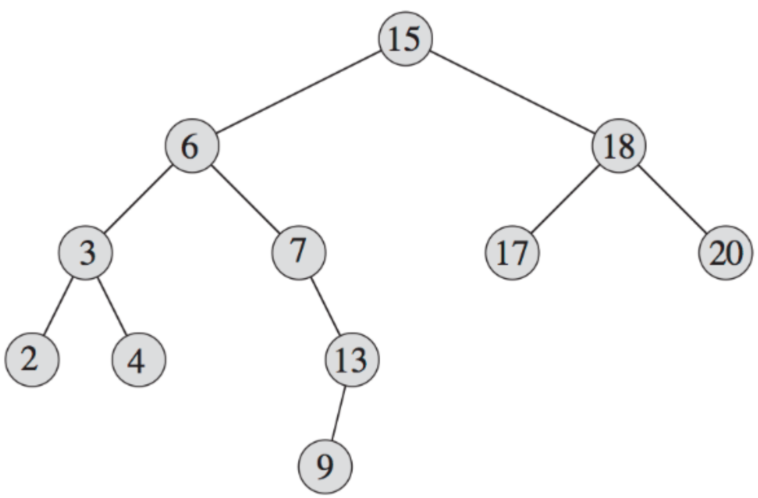
In-Order:



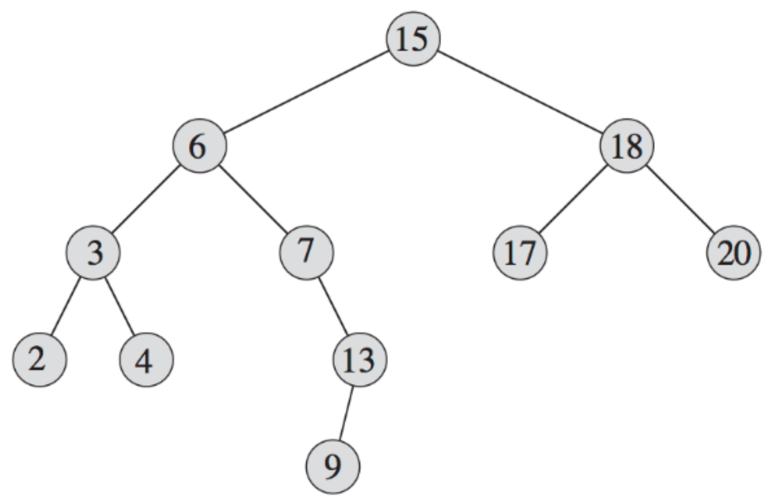
Pre-Order:



Post-Order:



Level-Order:



Tiefensuche / DFS Traversierung

```
preorder(node):
    if node != None
        visit(node)
        preorder(node.left)
        preorder(node.right)

inorder(node):
    if node != None
        inorder(node.left)
        visit(node)
        inorder(node.right)

postorder(node):
    if node != None
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche
- Lösung mit einer Warteschlange:
  - Wenn ein Knoten besucht wird, werden seine Kinder in die Queue eingereiht

```
BFS-Traversal:
    Q = new Queue()
    Q.enqueue(root)
    while not Q.empty() do
        node = Q.dequeue()
        visit(node)
        if node.left != None
            Q.enqueue(node.left)
        if node.right != None
            Q.enqueue(node.right)
```

Analyse Baum-Traversierung

Tiefensuche:

- Jeder Knoten wird genau einmal besucht
- Kosten pro Knoten: $O(1)$
- **Gesamtzeit** für DFS Traversierung: $\boldsymbol{O(n)}$

Breitensuche:

- Jeder Knoten wird genau einmal besucht
  - Kosten pro Knoten ist linear in der Anzahl Kinder
  - Aber: Jeder Knoten wird genau einmal in die FIFO-Queue eingefügt
- Kosten pro Knoten (amortisiert): $O(1)$
- **Gesamtzeit** für BFS Traversierung: $\boldsymbol{O(n)}$

Tiefe eines binären Suchbaums

Worst-Case Laufzeit der Operationen in binären Suchbäumen:

$$\boldsymbol{O}(\textbf{Tiefe des Baums})$$

- Im **besten Fall** ist die Tiefe $\boldsymbol{\log_2 n}$
  - Definition Tiefe: Länge des längsten Pfades von der Wurzel zu einem Blatt
- Im **schlechtesten Fall** ist die Tiefe $\boldsymbol{n-1}$

Was ist die Tiefe in einem typischen Fall?

- Was ist ein typischer Fall?

Ist es möglich, in einem binären Suchbaum immer Tiefe $O(\log n)$ zu garantieren?

"Typischer" Fall

Zufälliger binärer Suchbaum:

- **n Schlüssel werden in zufälliger Reihenfolge eingefügt**

Beobachtung:

- Mit Wahrscheinlichkeit $1/3$ haben beide Teilbäume der Wurzel mindestens $n/3$ Knoten.
- Analoges gilt auch für alle Teilbäume
- Im Durchschnitt wird deshalb auf jedem 3. Schritt von der Wurzel Richtung eines Blattes, der Teilbaum um einen Faktor $2/3$ kleiner!
- Verkleinern um einen Faktor $2/3$ geht nur $O(\log n)$ oft.
- Tiefe eines zufälligen binären Suchbaums ist deshalb $O(\log n)$
- Genaue Rechnung ergibt:

  Erwartete Tiefe eines zufälligen bin. Suchbaums: $\boxed{\mathbf{4.311 \cdot \ln n}}$

"Typischen" Fall erzwingen?

"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

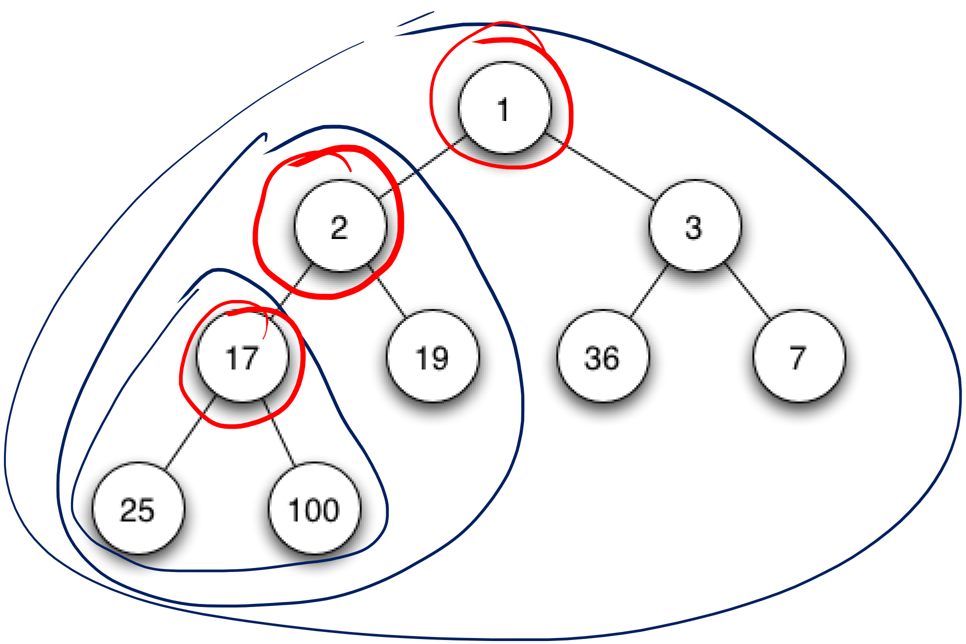
Idee:

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

Heap

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

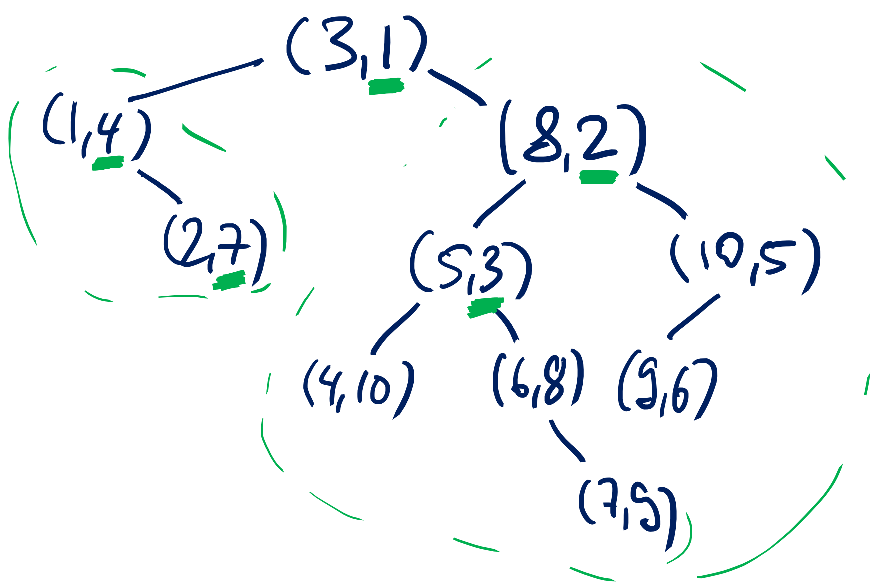
Annahme:

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

Ziel:

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

Beispiel: (1,4), (2,7), (3,1), (4,10), (5,3), (6,8), (7,9), (8,2), (9,6), (10,5)



bezüglich key2
↳ min-heap

Treap: Kombination BST / Heap

Annahme:

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

Treap:

- Binärer Suchbaum bezüglich *key1*
- Min-Heap bezüglich *key2*

TREAP

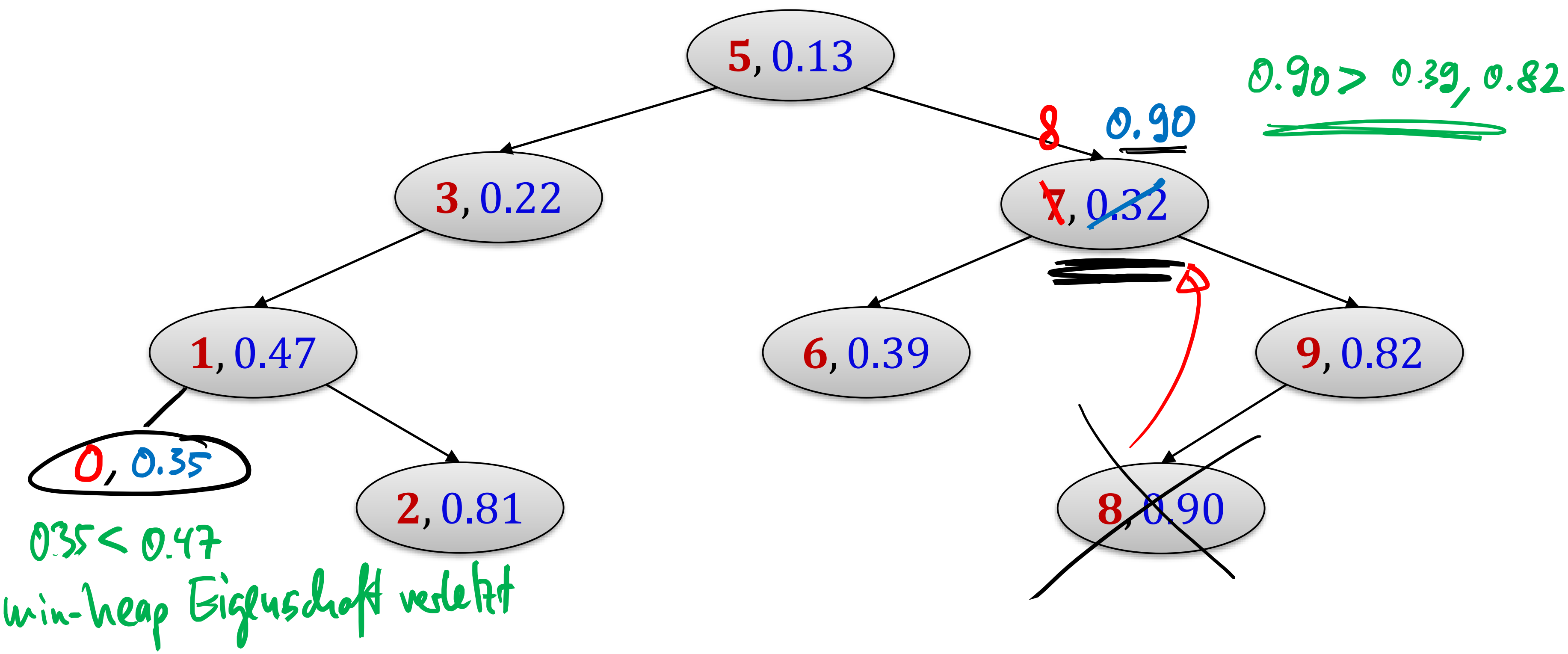
- Entspricht bin. Suchbaum der Schlüssel *key1*, in welchen die Schlüssel in der durch *key2* geg. Reihenfolge eingefügt wurden

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 beüglich der Schlüssel *key1* und *key2* ist!
- Enspricht 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: *insert*(**0**, 0.35), *delete*(**7**)



Sentinel

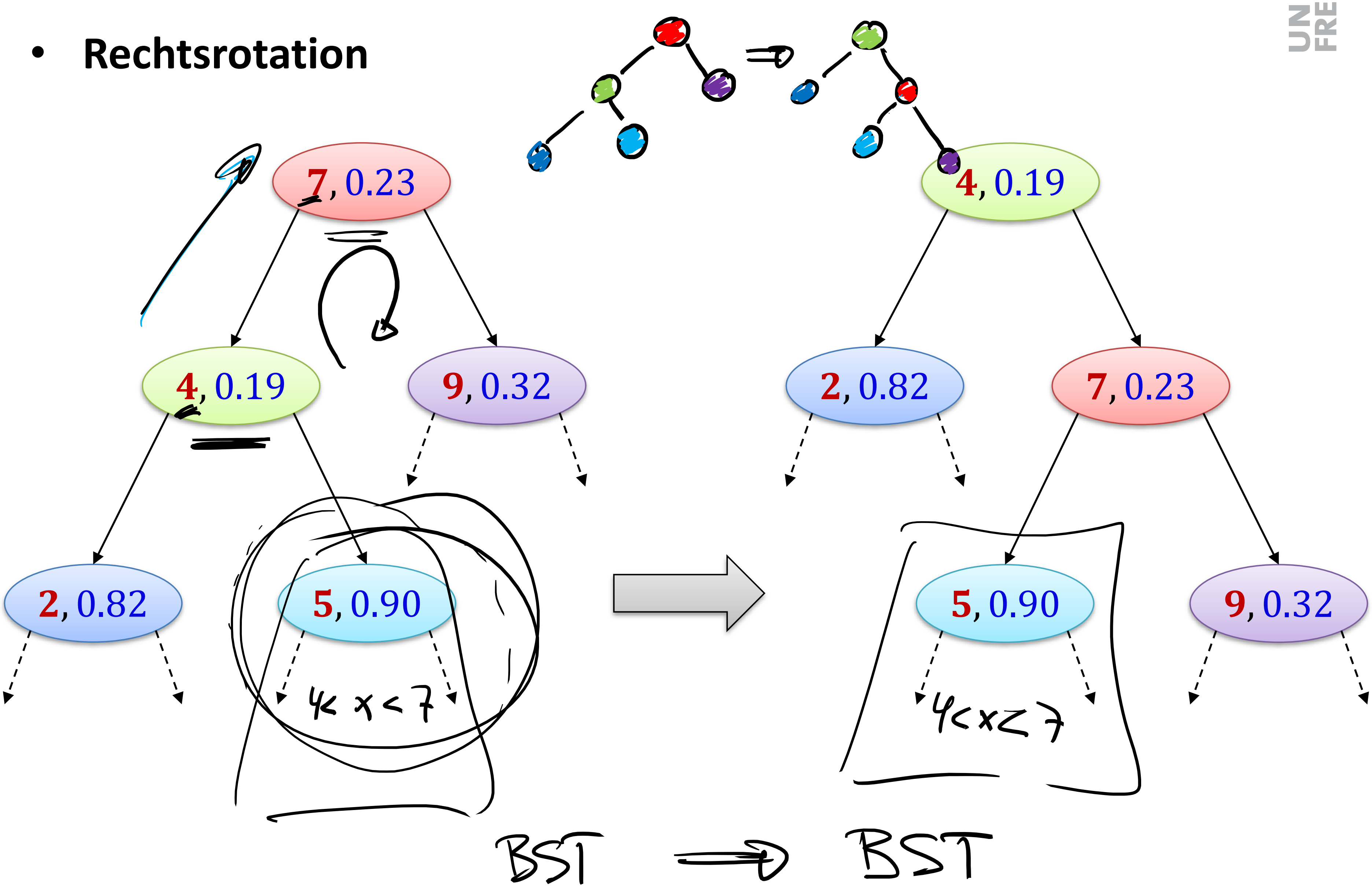
Um den Code zu vereinfachen…

Sentinel-Knoten: *NIL*

- Ersetzt alle None(null)-Pointer
- *NIL.key1* ist nicht definiert (wird nie gesetzt oder gelesen)
- *NIL.key2 = -1.0* (und damit immer kleiner als alle anderen)
  - wird nie verändert
- *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

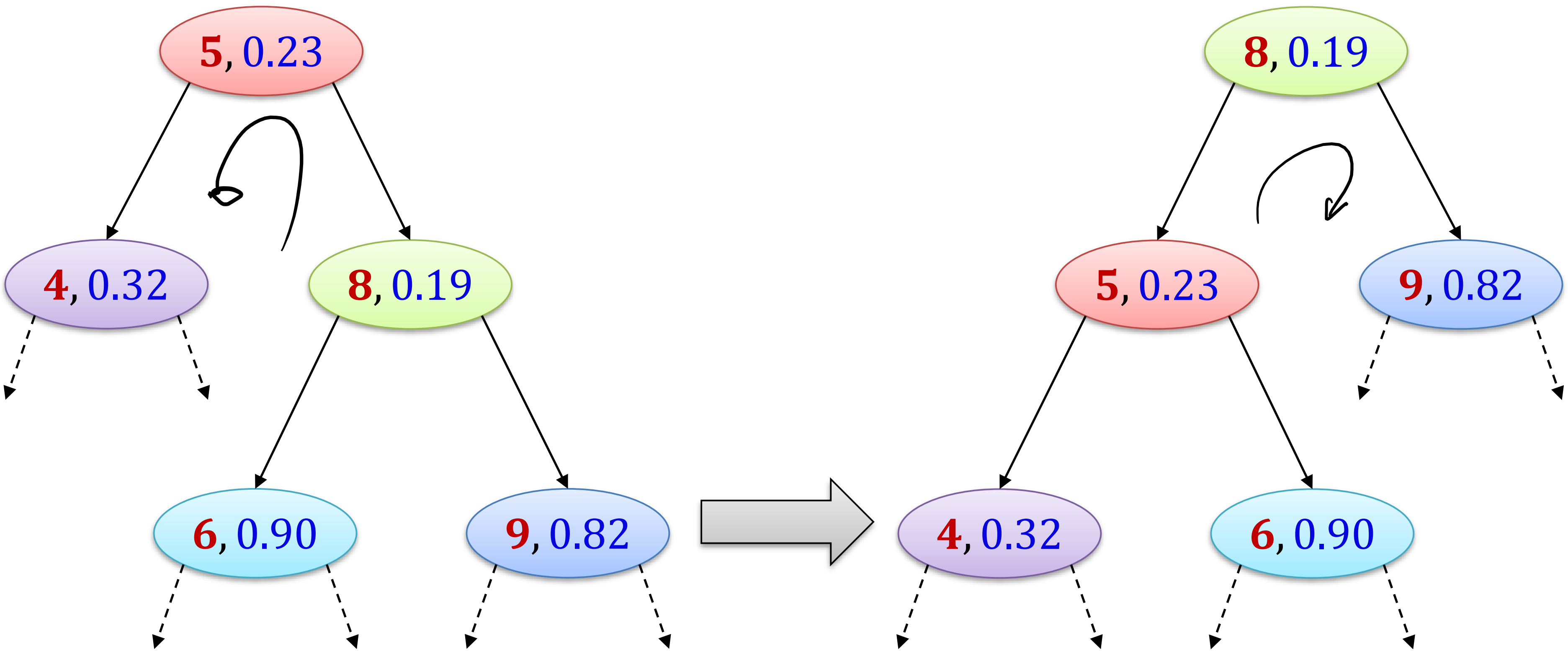
Rotationen

- **Rechtsrotation**



Rotationen

- **Linksrotation**

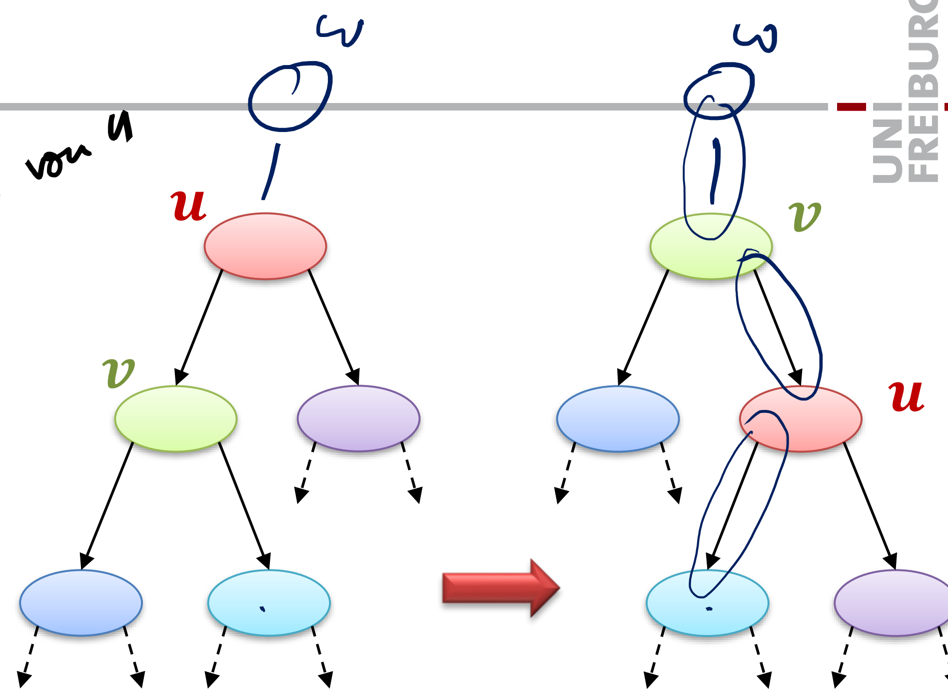


Rechtsrotation

Wurzel des Teilbaum

linkes Kind von u

```
right-rotate(u,v):
   u.left = v.right
   u.left.parent = u
   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
   v.right = u
   u.parent = v
```

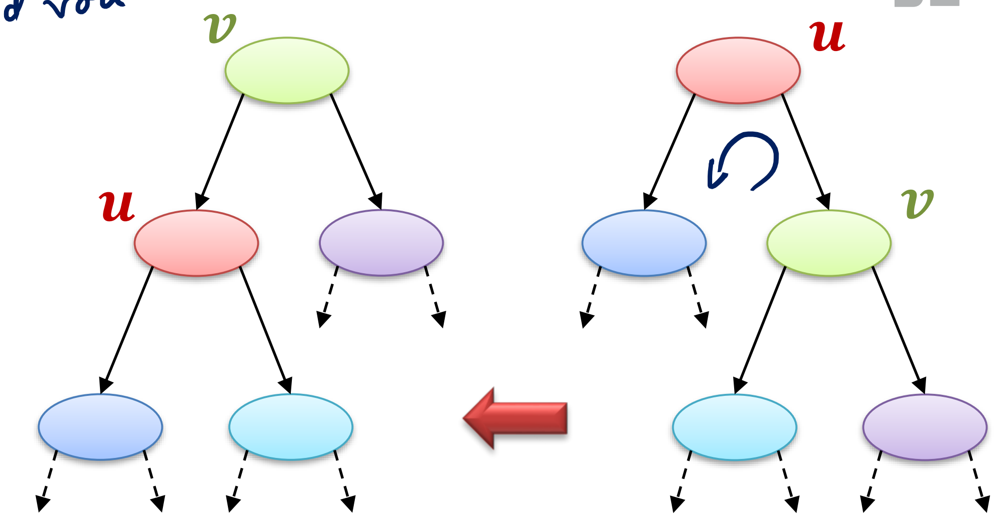


Linksrotation

Wurzel des Teilbaums

rechtes Kind von u

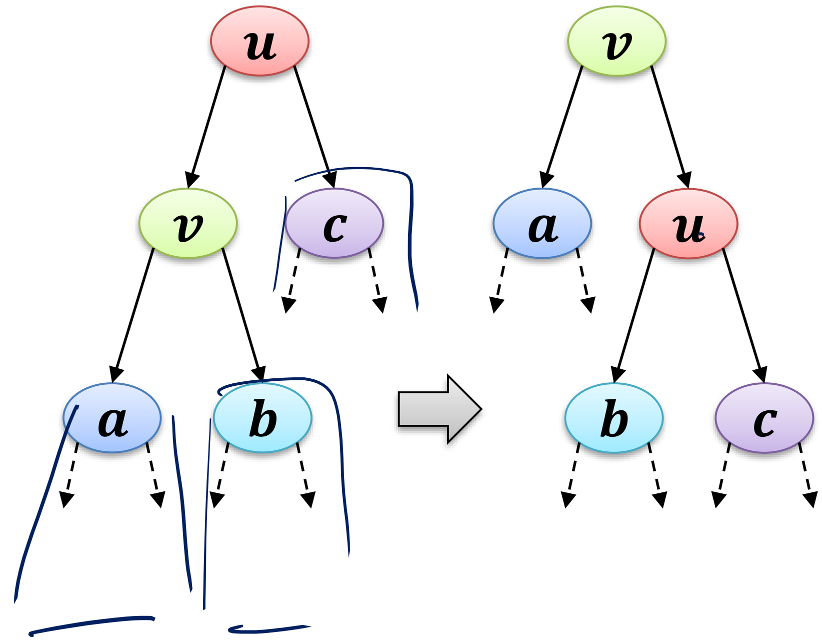
```
left-rotate(u,v):
   u.right = v.left
   u.right.parent = u
   if u == root then
      root = v
   else
      if u == u.parent.left then
         u.parent.left = v
      else
         u.parent.right = v
   v.parent = u.parent
   v.left = u
   u.parent = v
```



Korrektheit: Rotationen

Lemma: Rotationen erhalten die “Bin. Search Tree”-Eigenschaft

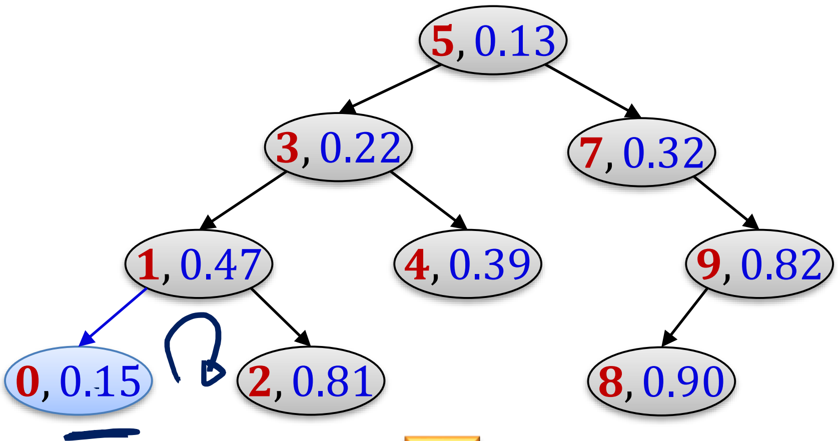
$$\underbrace{\text{keys}(a) < v.\text{key} < \text{keys}(b) < u.\text{key} < \text{keys}(c)}_{\text{vor Rot.}}$$

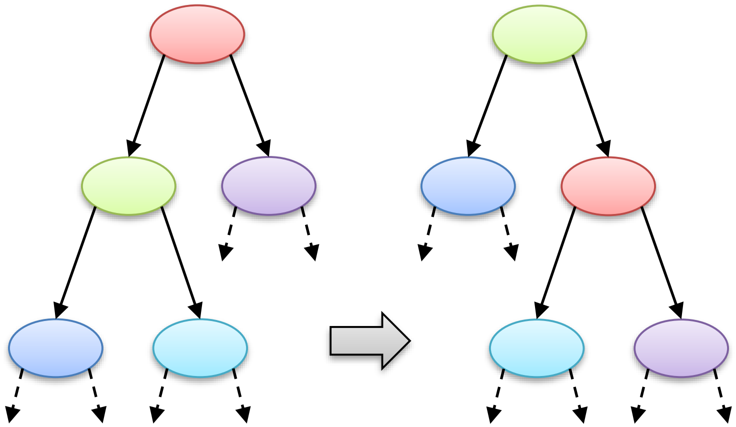


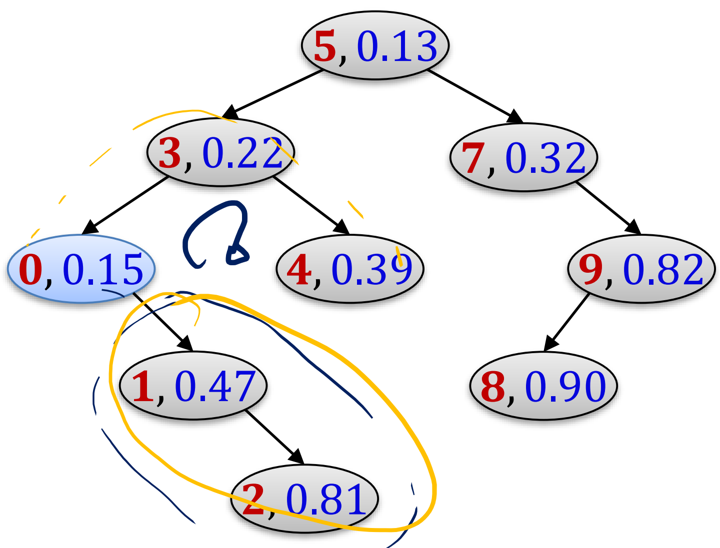
$$\text{keys}(a) < v.\text{key} < \text{keys}(b) < u.\text{key} < \text{keys}(c)$$

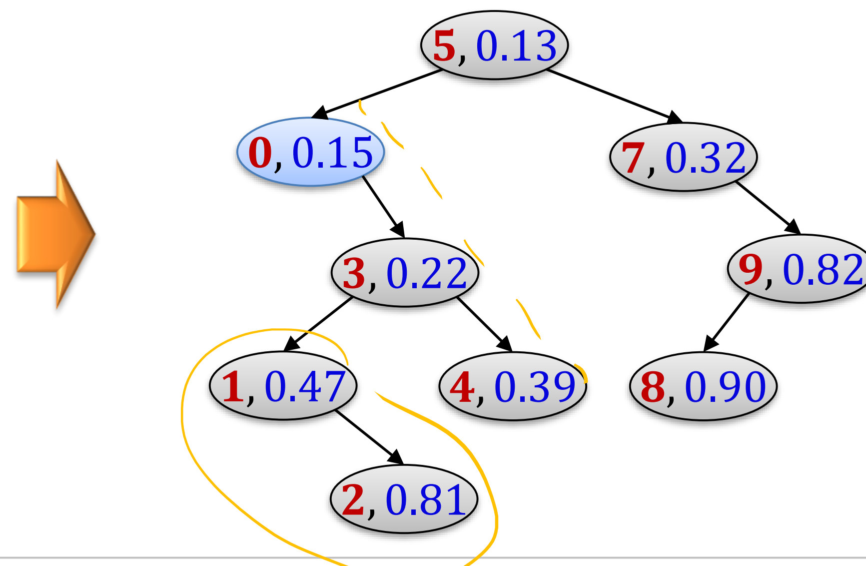
Treap: Einfügen

- Beispiel: *insert*(**0**, 0.15)





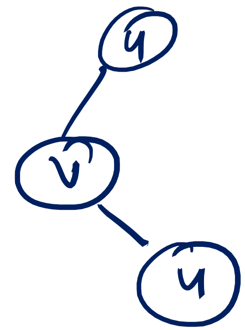




Treap: Einfügen

Ziel: Füge Schlüssel x ein

1. Bestimme zufälligen zweiten Schlüssel y
2. Füge (x, y) wie üblich in den bin. Suchbaum (bez. x) ein
3. Annahme:
   Knoten mit Schlüssel x heisst v, $\quad u = v.\mathrm{parent}$




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

Treap insert(x)

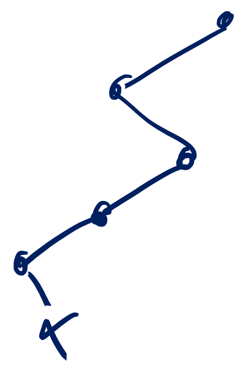
```
// assumption: treap is not empty
v = root; y = random(0,1)
while v != NIL and v.key1 != x do
    if v.key1 > v.x then
        if v.left == NIL then
                                           w.key1 w.key2 w.parent w.left w.right
            w = new TreeNode(x,y,v,NIL,NIL); v.left = w
        v = v.left
    else
        if v.right == NIL then
            w = new TreeNode(x,y,v,NIL,NIL); v.right = w
        v = v.right

// 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

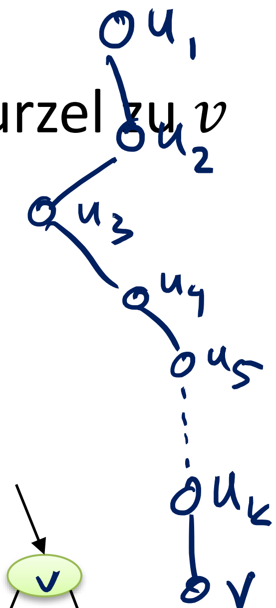


- Rotationen erhalten die BST-Eigenschaft

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:

$$\underline{root = u_1, u_2, \dots, u_k, v}$$



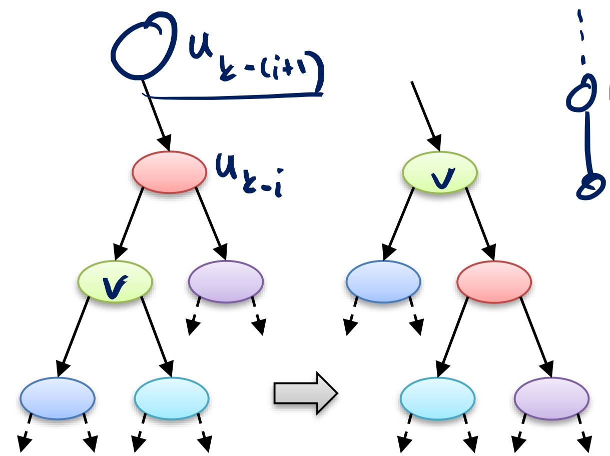
Lemma: Nach $i \leq k$ Rotationen ist der Pfad von der Wurzel zu v

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

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

- Verankerung $i = 0$ ✔
- Induktionsschritt:

$i \to i+1$ ✓



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: $u_j.key2 \leq w.key2$

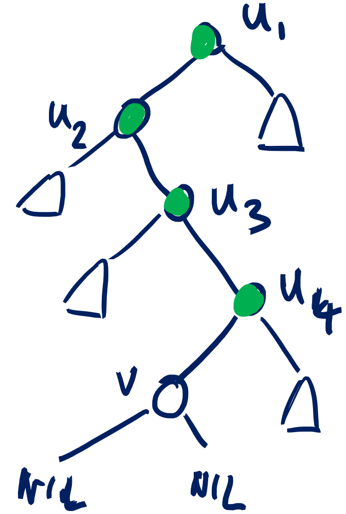
i u_j

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

- Verankerung: $i=0$

(2) ✓

(3) ✓



(im Bsp. $k=4$)

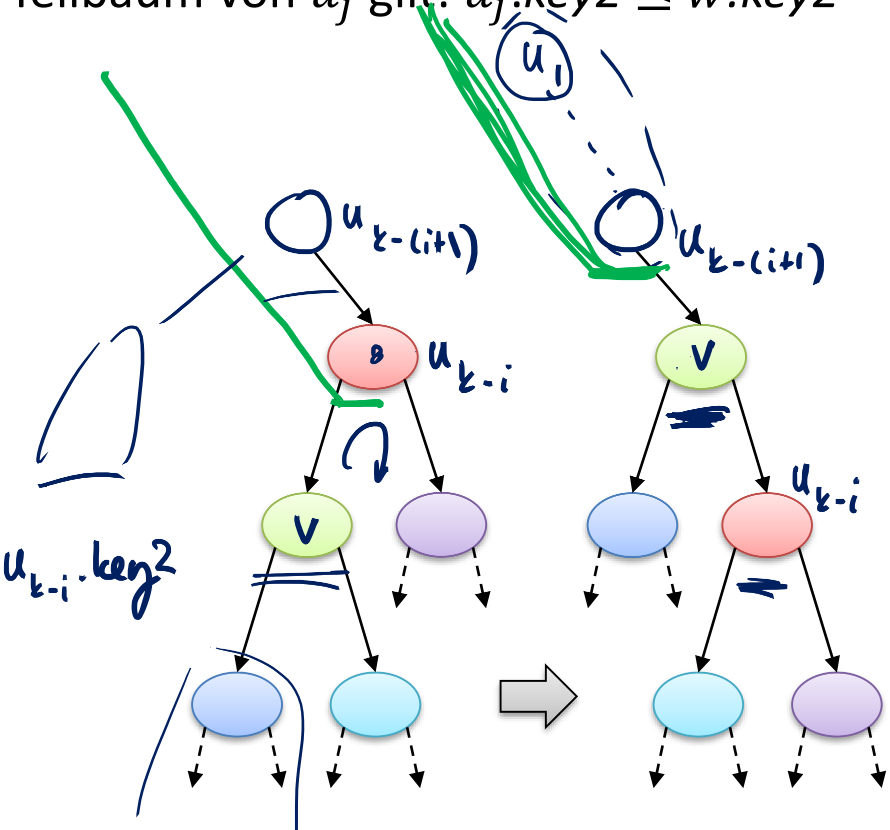
Treap: Heap-Eigenschaft

Lemma: Nach $i \leq k$ Rotationen:

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

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

- Induktionsschritt: $i \to i+1$



(2) folgt aus (2) und (3) für i (Ind.-Voraussetzung) ✓

(3) folgt aus (3) für i

Treap: Einfügen

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

folgt direkt aus den Lemmas

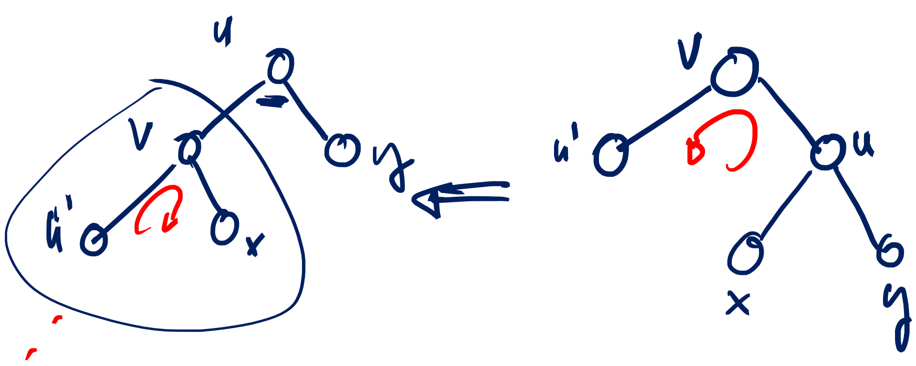
Korollar: Einfügen hat mit hoher Wahrscheinlichkeit Laufzeit $O(\log n)$.

Struktur wie bei zuf. Einfügereihenfolge

Treap: Löschen

Ziel: Lösche Schlüssel x

1. Suche Knoten v mit Schlüssel x
2. Solange v zwei Kinder hat:
   - w ist das Kind mit kleinerem $\mathrm{key2}$
   - Falls $w == v.\mathrm{left}$, dann mache Rechtsrot.: `right-rotate(v,w)`
   - Falls $w == v.\mathrm{right}$, dann mache Linksrot.: `left-rotate(v,w)`
3. Lösche v
   - v hat höchstens ein Kind und kann daher einfach gelöscht warden (ohne die min-heap Eigenschaft zu verletzen)



u.key2 < v.key2

u.'key2 < u'.key2

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…

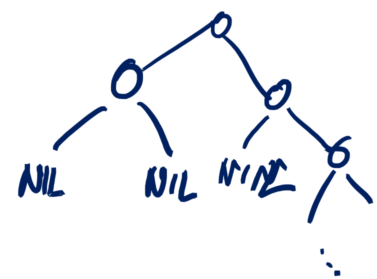
Rot-Schwarz-Bäume

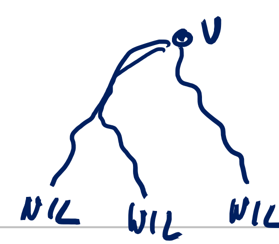
Ziel: Binäre Suchbäume, welche **immer balanciert** sind

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

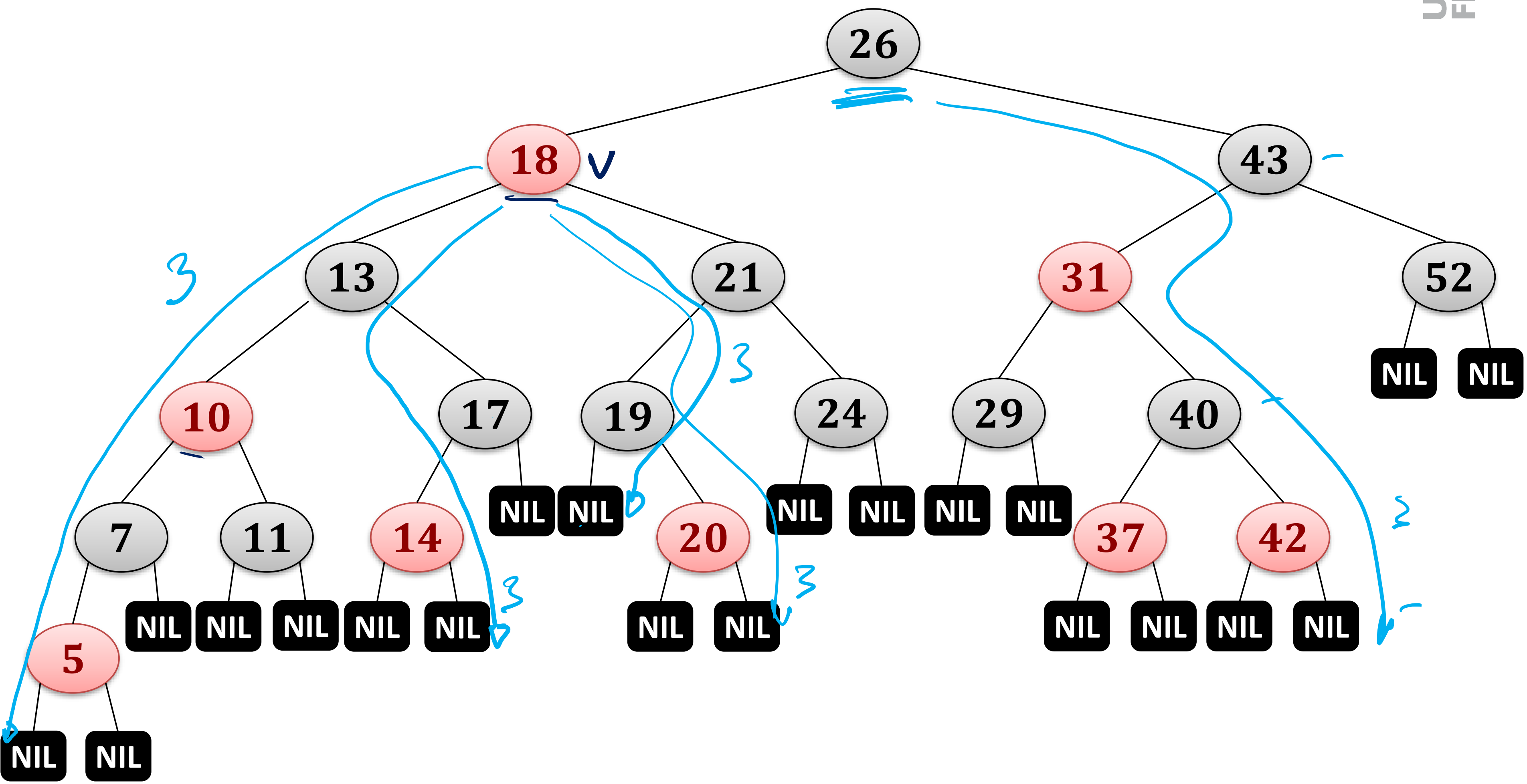
Rot-Schwarz-Bäume sind binäre Suchbäume, für die gilt:

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



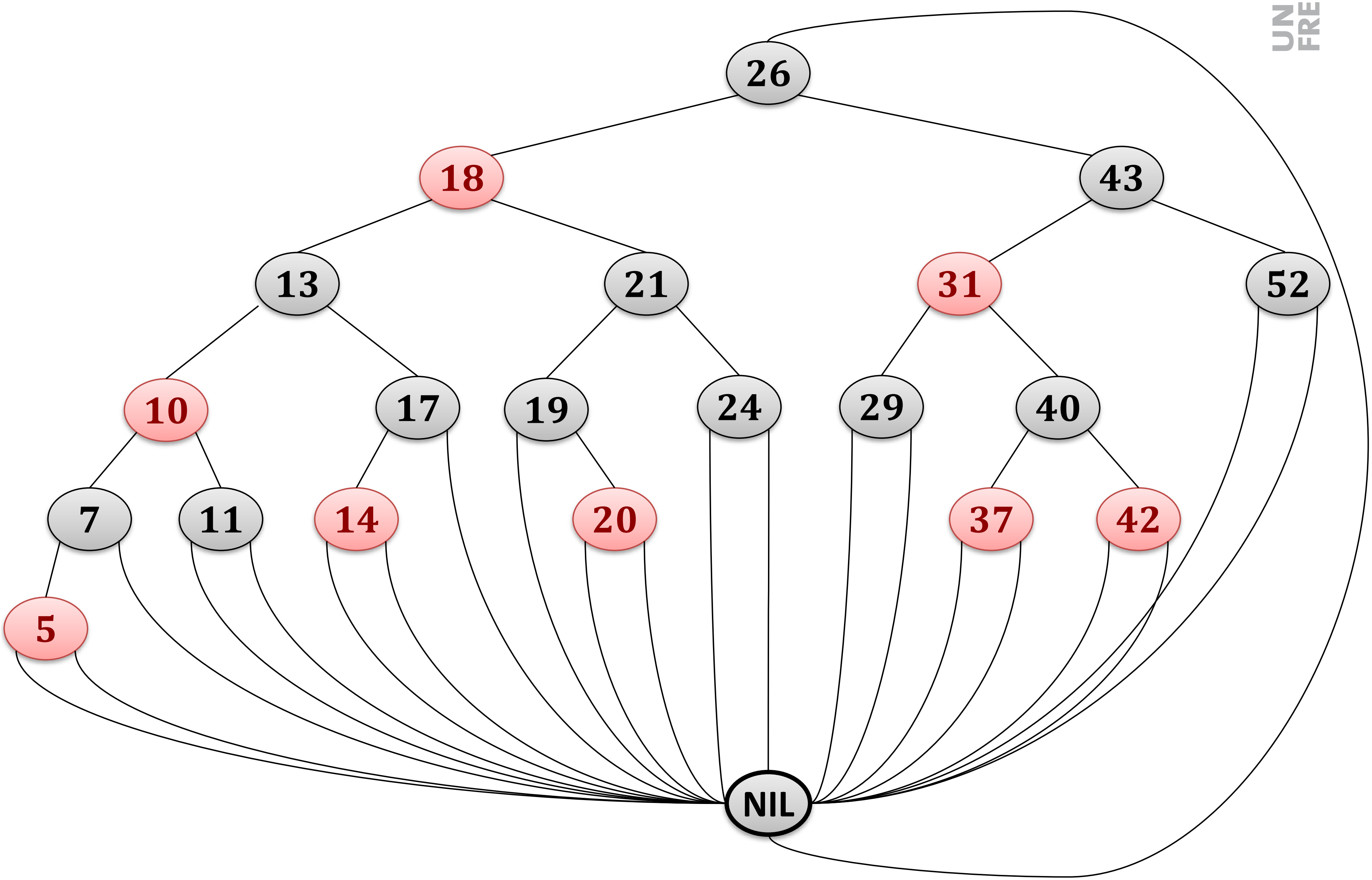
Sentinel

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

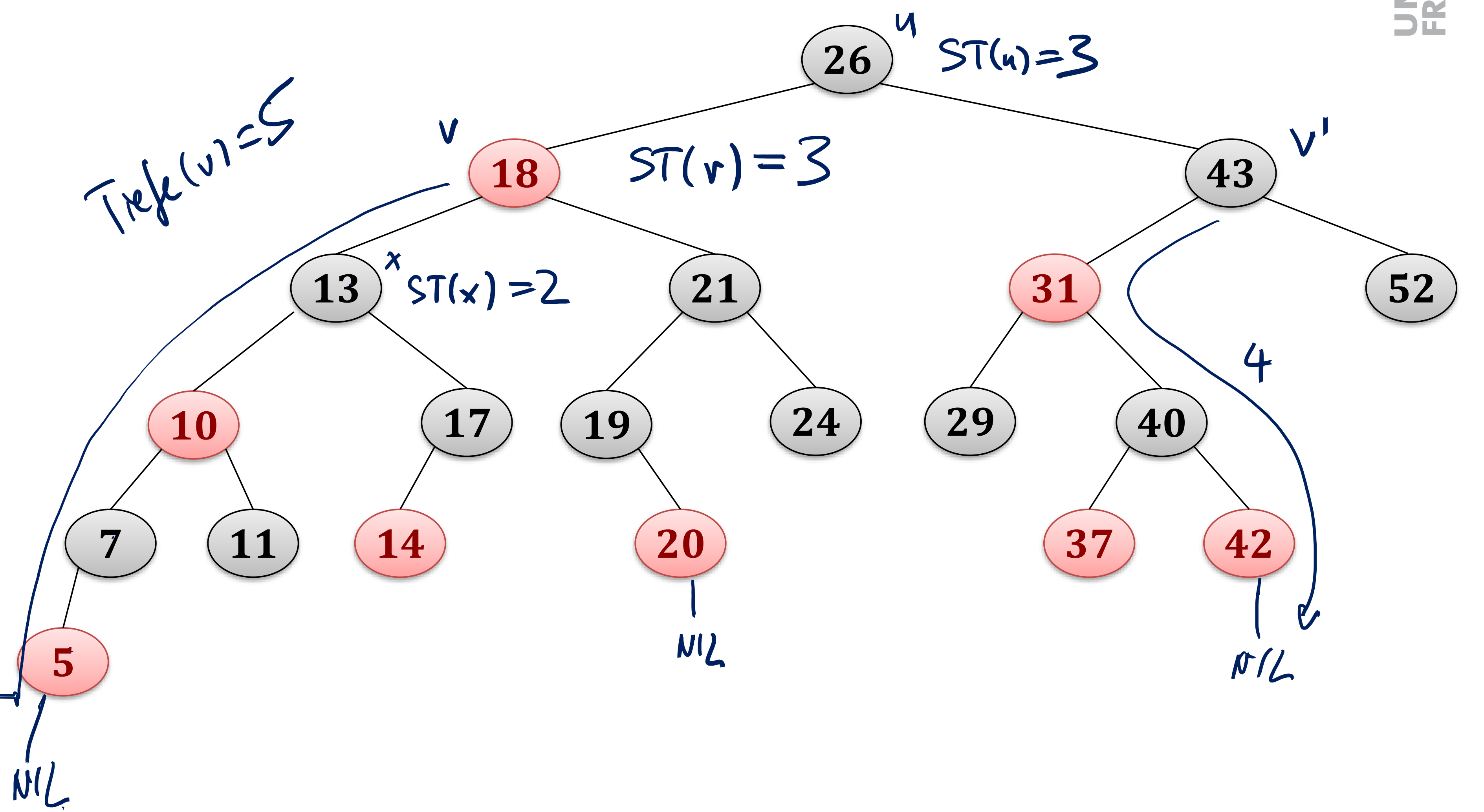
Sentinel-Knoten: *NIL*

- Ersetzt alle None/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 darauf 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

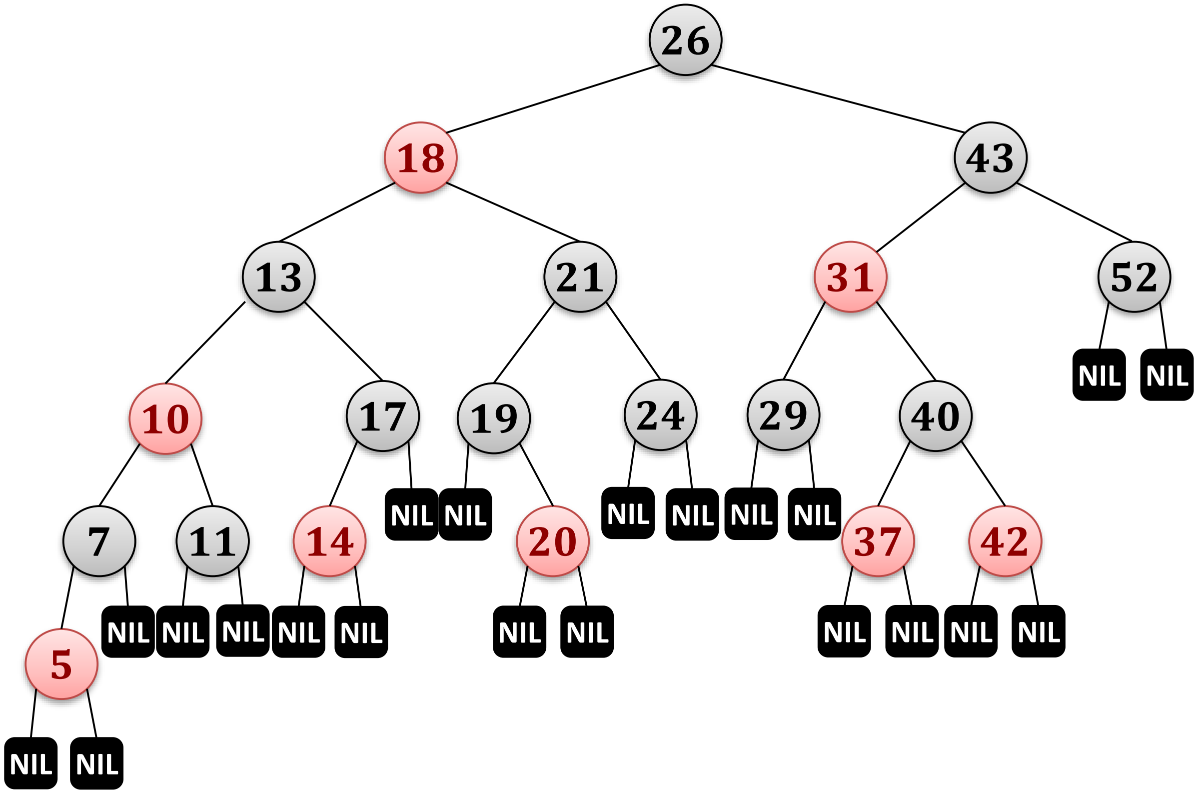


Tiefe / Schwarz-Tiefe

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!



Schwarz-Tiefe ↔ Anzahl Knoten

Lemma: Im Teilbaum eines Knoten v mit **Schwarz-Tiefe $ST(v)$** ist die **Anzahl innerer Knoten** $\geq 2^{ST(v)} - 1$

nicht NIL-Knoten

\# Knoten $\geq 2^{ST(v)} - 1$

Beweis:

- Per Induktion über die Tiefe $T(v)$ von v

Für einen Knoten mit Tiefe t und Schwarztiefe s gilt: $\frac{t}{2} \leq s \leq t$

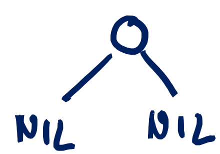
Teilb. mit Tiefe t und Schw.-tiefe s hat $\geq 2^s - 1$ innere Knoten

Verankerung: $t = 0 \Rightarrow s = 0$

NIL $\rightarrow$ 0 innere Knoten

$2^0 - 1 = 0$ ✓

$t = 1 \rightarrow s = 1$



$2^1 - 1 = 1$ ✓

Schritt:

$t \geq 1$: Aussage für $t \geq 0$ gilt, dann gilt sie auch für $t+1$

Schwarz-Tiefe ↔ Anzahl Knoten

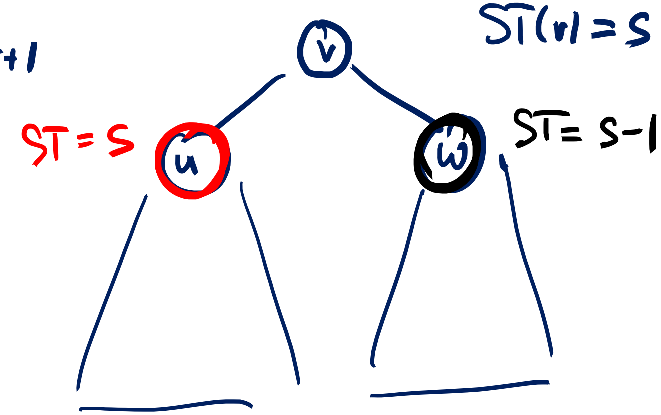
Lemma: Im Teilbaum eines Knoten v mit **Schwarz-Tiefe** $\boldsymbol{ST(v)}$ ist die **Anzahl innerer Knoten**

$$\geq \mathbf{2^{ST(v)} - 1}$$

Beweis:

- Per Induktion über die Tiefe $T(v)$ von v

$t \to t+1$



Tiefe $= t+1$

$T(u) \leq t$, $T(w) \leq t$ ← können Ind.-V. anwenden

$n(u) \geq 2^{s-1} - 1$

$n(w) \geq 2^{s-1} - 1$

$ST(u) \geq s-1$ $\quad ST(w) \geq s-1$

Bed. (5) von RS-Bäumen

$n(v) = 1 + n(u) + n(w) = 1 + 2 \cdot (2^{s-1} - 1) = \underline{\underline{2^s - 1}}$ ✓

Tiefe eins Rot-Schwarz-Baumes

Theorem:

Die **Tiefe** eines Rot-Schwarz-Baumes ist $\leq 2\log_2(n+1)$.

Beweis:

- Anzahl innerer Knoten : n (alle ausser den NIL-Knoten)
- Lemma:

$$n \geq 2^{ST(root)} - 1$$

$$\frac{T(root)}{2} \leq ST(root) \leq \log_2(n+1)$$

$$T(root) \leq \underline{\underline{2 \cdot \log_2(n+1)}}$$