

Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 10 (28.5.2018)

Binäre Suchbäume II



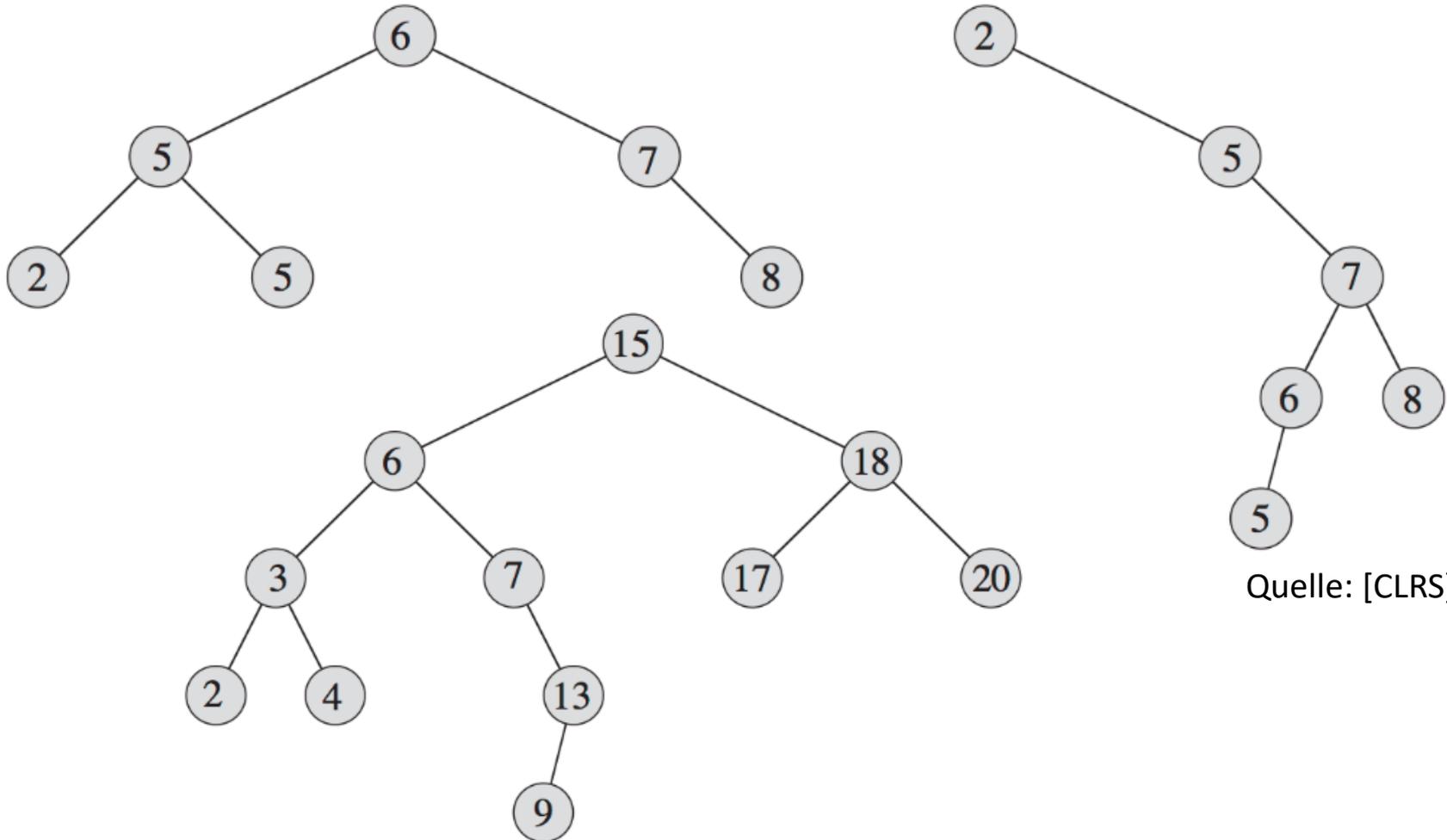
**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

Binäre Suchbäume

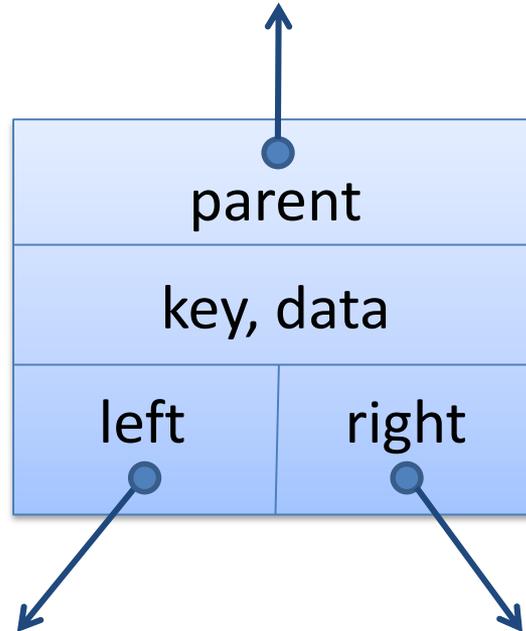
- Binäre Suchbäume müssen nicht immer so schön symmetrisch sein...



Quelle: [CLRS]

Binärer Suchbaum : Elemente

TreeElement:

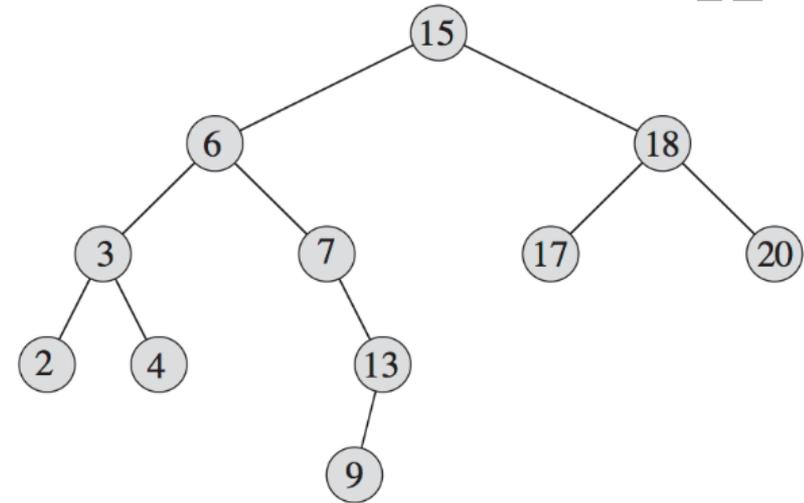


Implementierung: gleich wie bei den Listen-Elementen

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`:

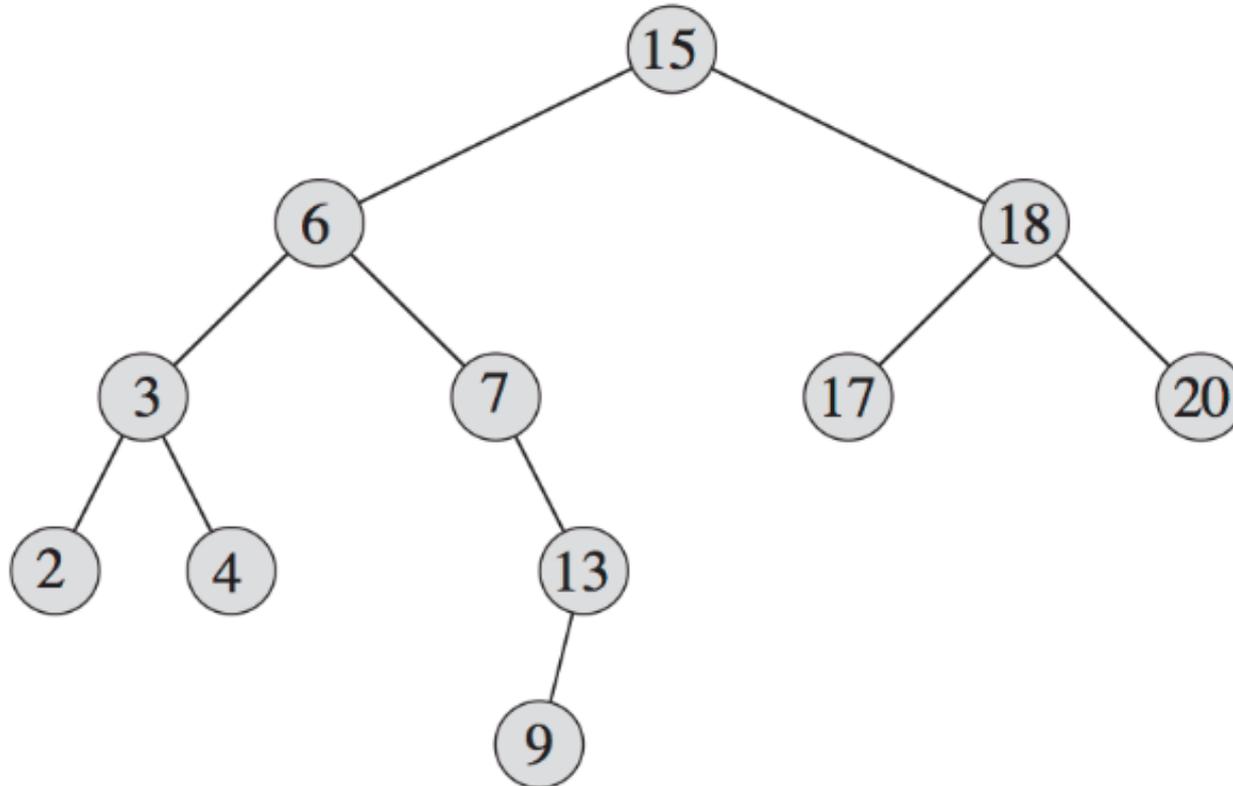
`current = current.left`

else:

`current = current.right`

Einfügen eines Schlüssels

Füge Schlüssel 1, 5, 14, 6.5, 19 ein...



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 $v.left = \text{None}$
 - 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!

Die Operationen

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

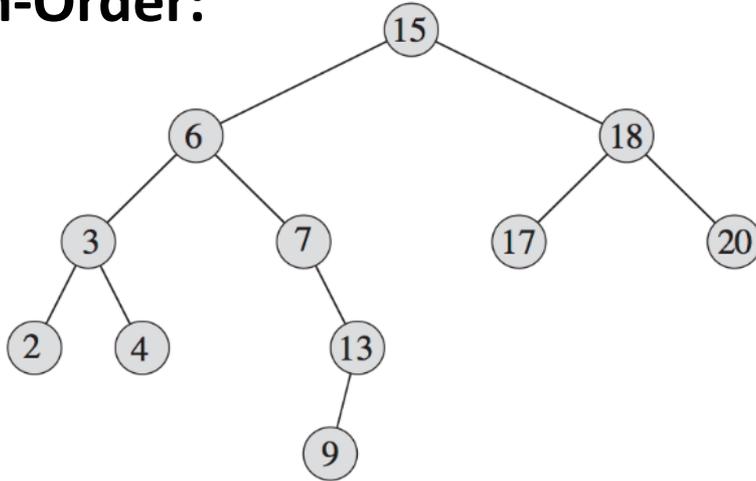
haben alle **Laufzeit $O(\text{Tiefe des Baums})$** .

Was ist die Tiefe eines binären Suchbaums?

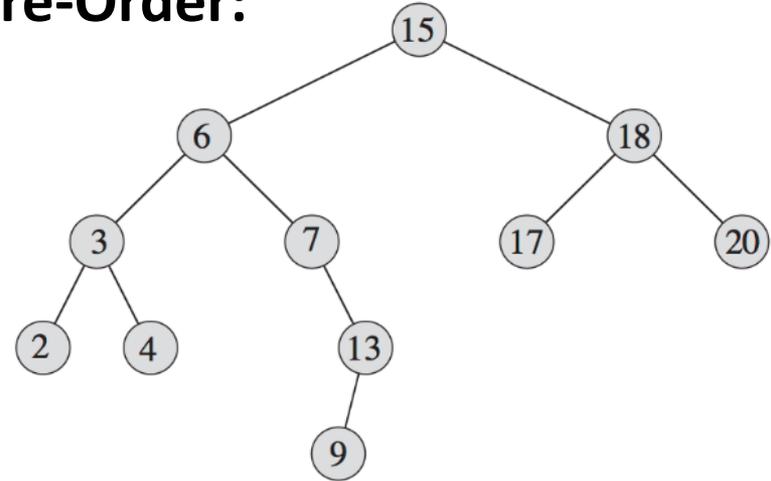
Traversieren eines binären Suchbaums

Ziel: Besuche alle Knoten eines binären Suchbaums einmal

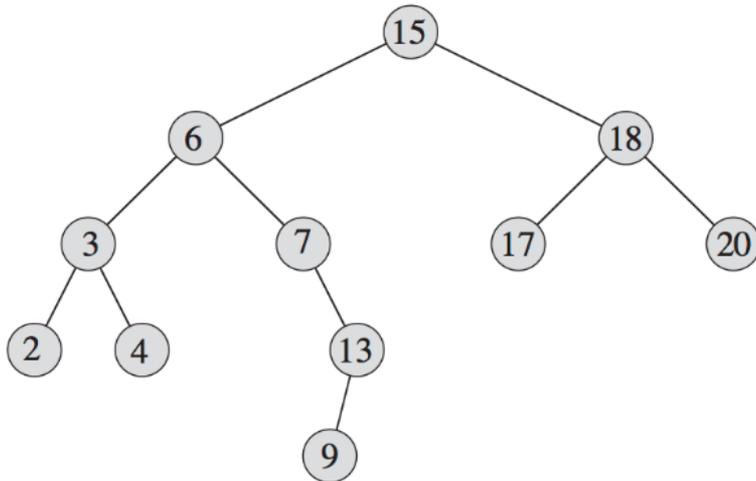
In-Order:



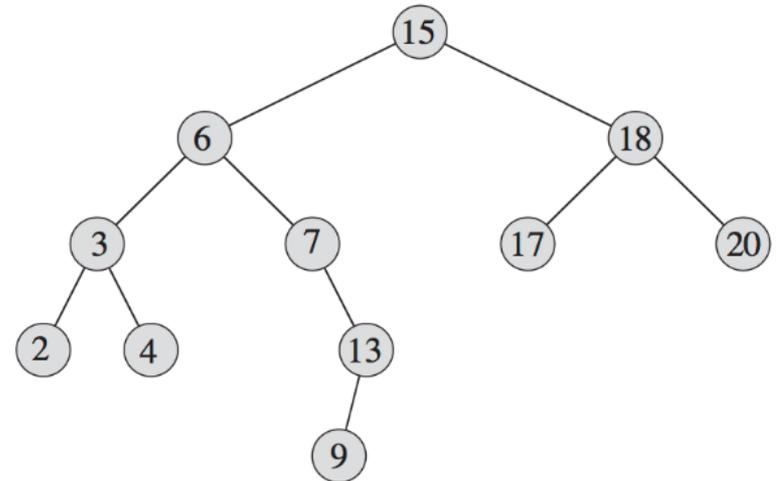
Pre-Order:



Post-Order:



Level-Order:



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)
```

Tiefensuche:

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

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: $O(n)$

Worst-Case Laufzeit der Operationen in binären Suchbäumen:

$O(\text{Tiefe des Baums})$

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

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?

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: $4.311 \cdot \ln n$

“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:

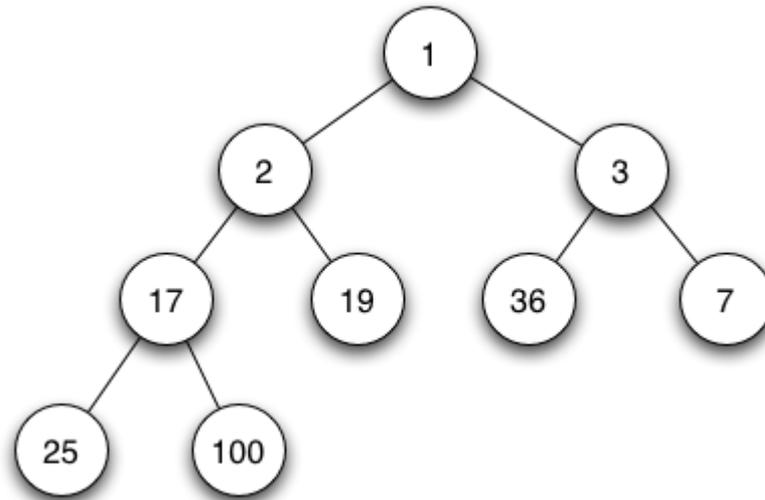
- Zufällige Reihenfolge ist nicht unbedingt der typische Fall!
- Vorsortierte Werte kann genau so typisch sein
 - Das ergibt einen sehr schlechten binären Suchbaum

Idee:

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

Heap (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

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)

Annahme:

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

Treap:

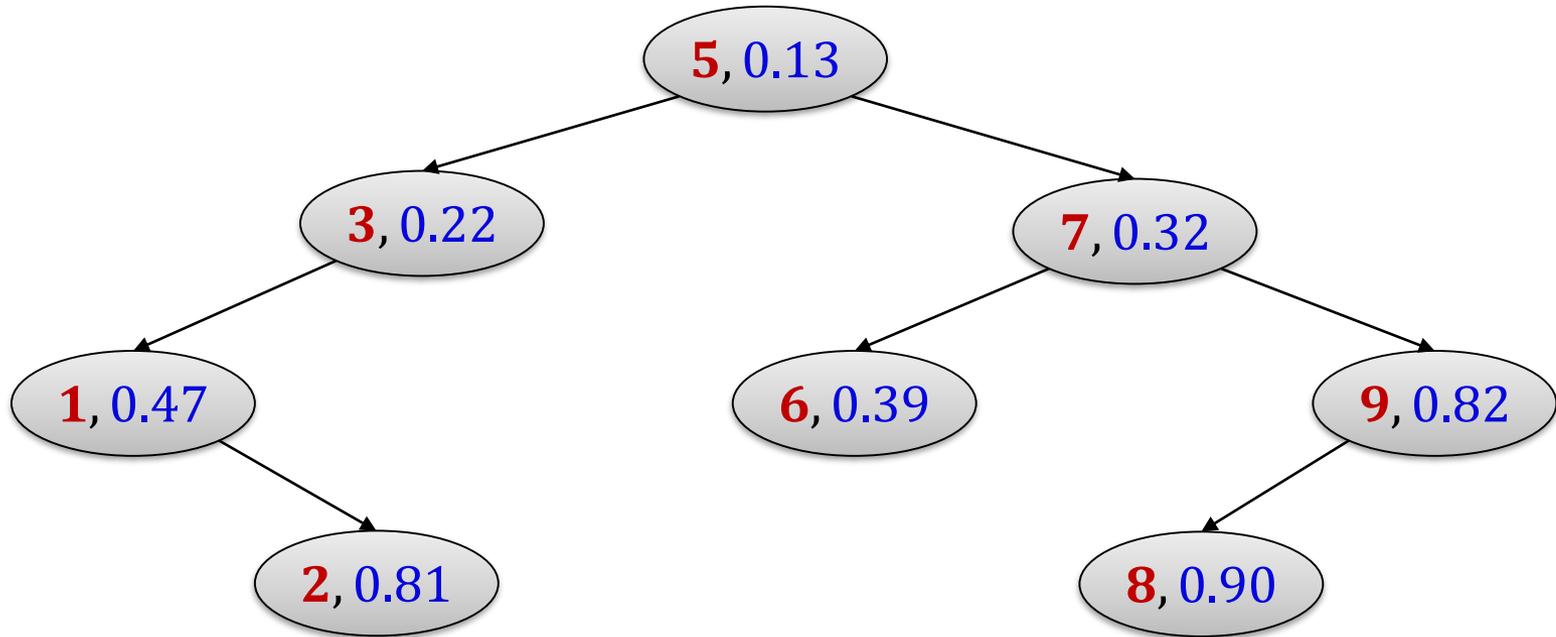
- **Binärer Suchbaum** bezüglich *key1*
- **Min-Heap** bezüglich *key2*
- Entspricht bin. Suchbaum der Schlüssel *key1*, in welchen die Schlüssel in der durch *key2* geg. Reihenfolge eingefügt wurden

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: *insert(0, 0.35)*, *delete(7)*

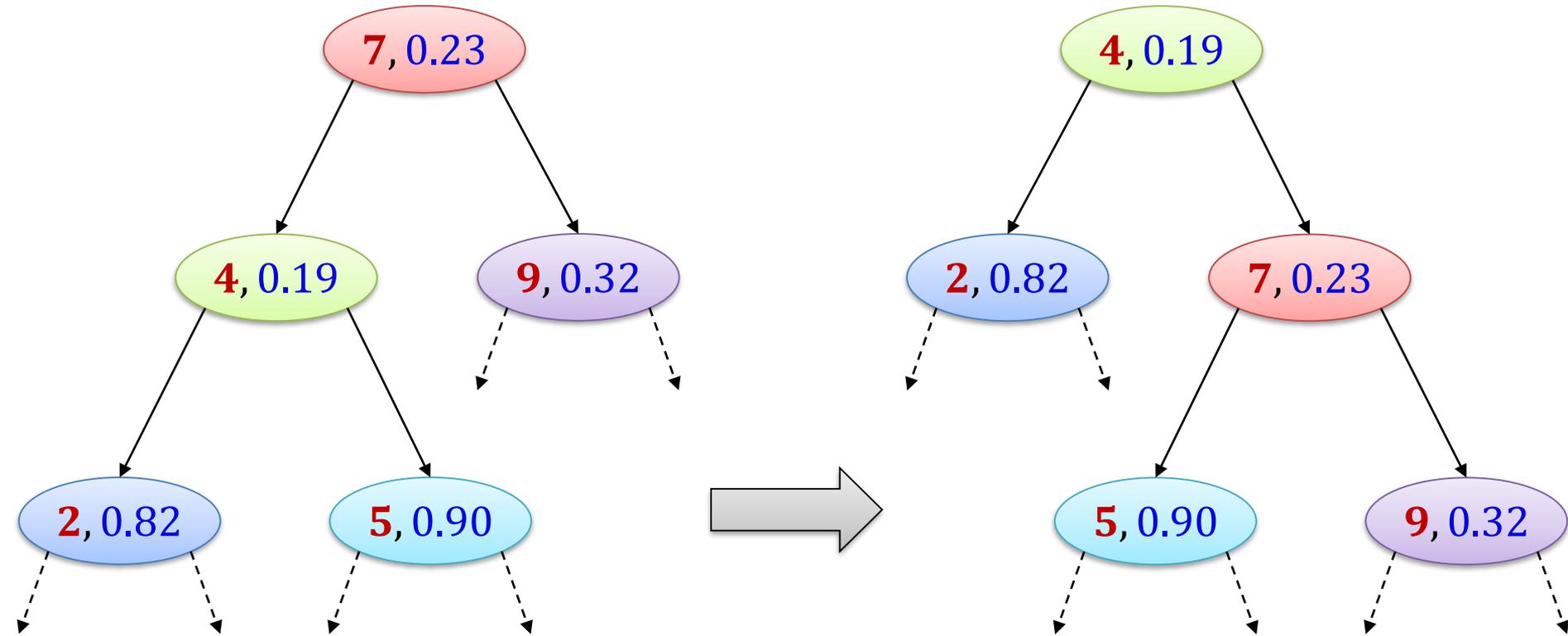


Um den Code zu vereinfachen...

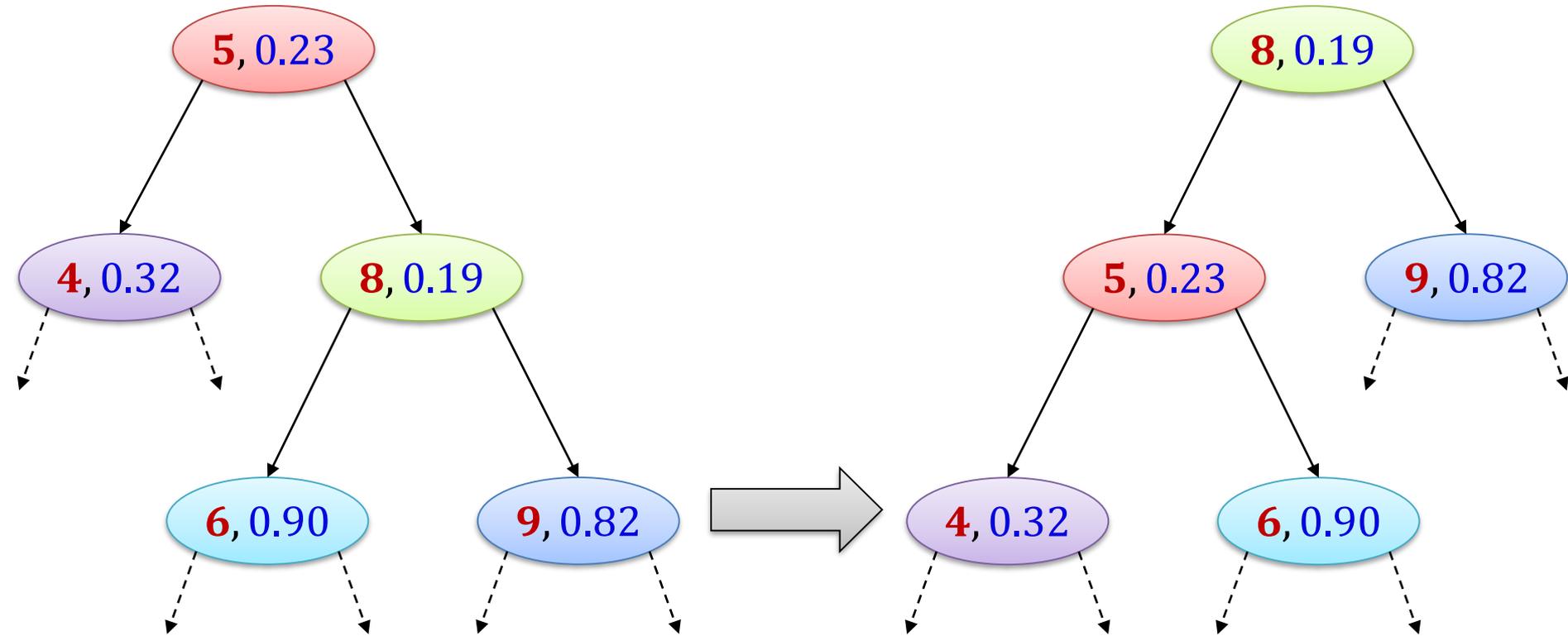
Sentinel-Knoten: *NIL*

- 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

- Rechtsrotation



- Linksrotation



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`

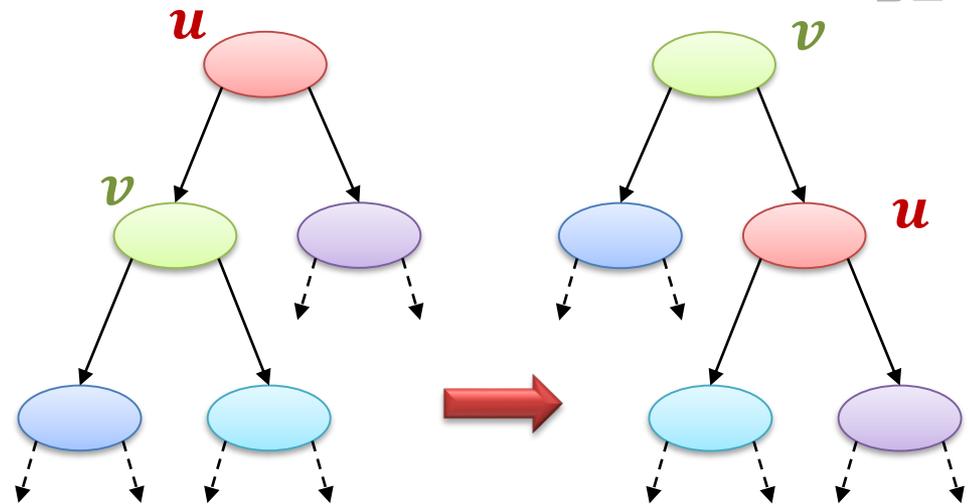
`else`

`u.parent.right = v`

`v.parent = u.parent`

`v.right = u`

`u.parent = v`



Linksrotation

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
```

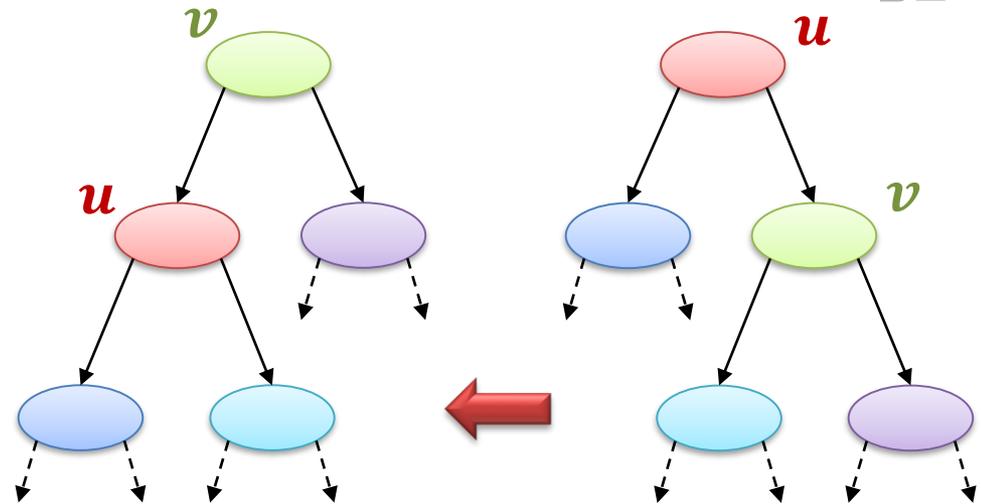
```
    else
```

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

```
v.parent = u.parent
```

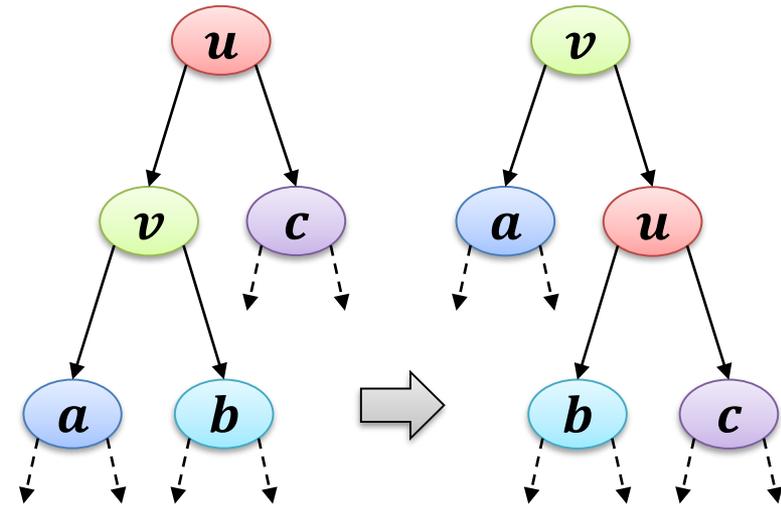
```
v.left = u
```

```
u.parent = v
```



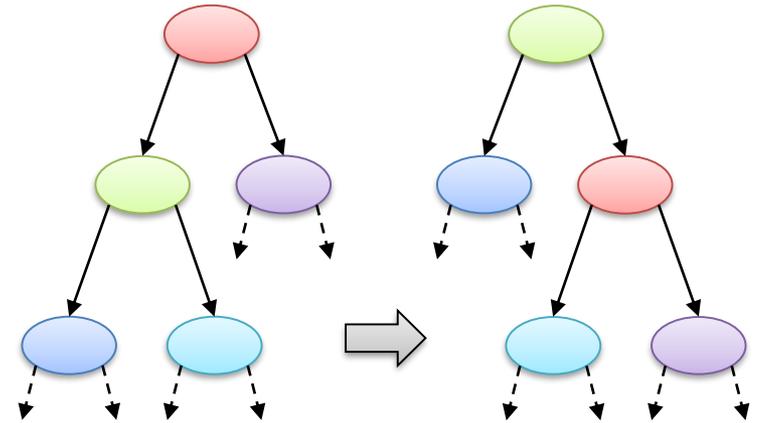
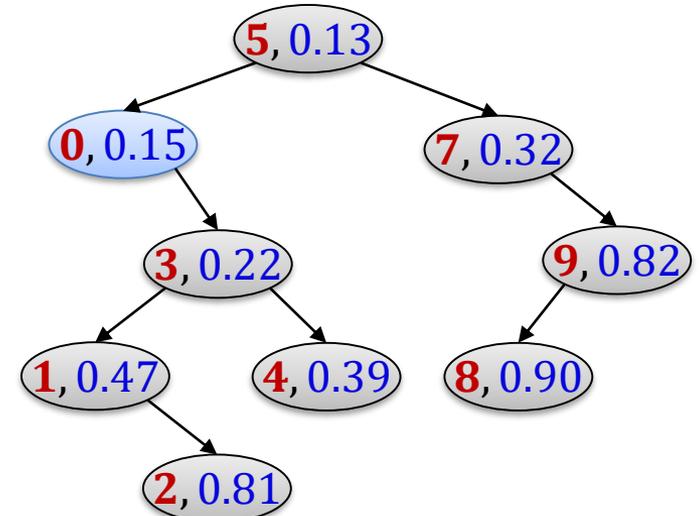
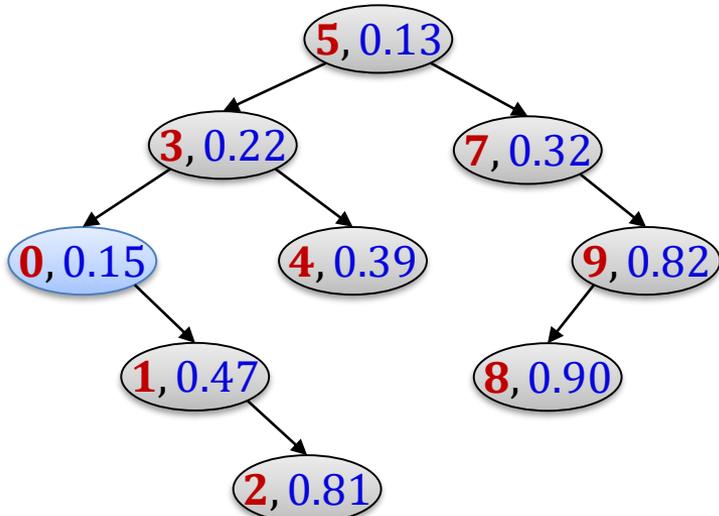
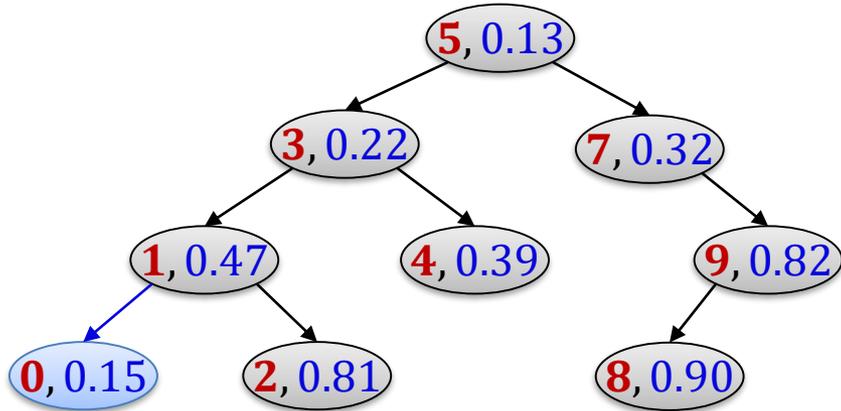
Korrektheit: Rotationen

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



Treap: Einfügen

- Beispiel: $insert(0, 0.15)$



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 , $u = v.\text{parent}$

```
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 = 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
```

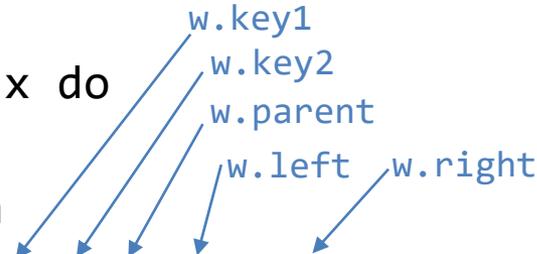
```
  if v == u.left then
```

```
    right-rotate(u,v)
```

```
  else
```

```
    left-rotate(u,v)
```

```
  u = v.parent
```



Treap: Terminierung

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

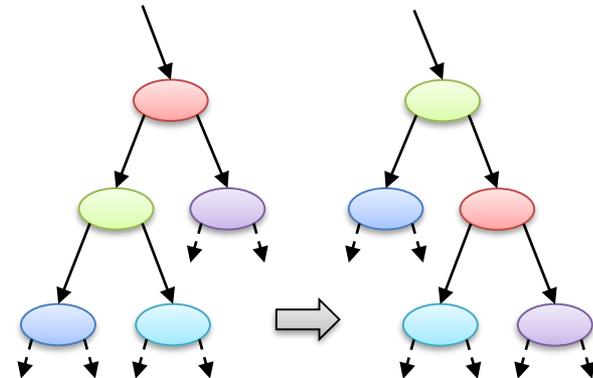
$$root = u_1, u_2, \dots, u_k, v$$

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:



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

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:

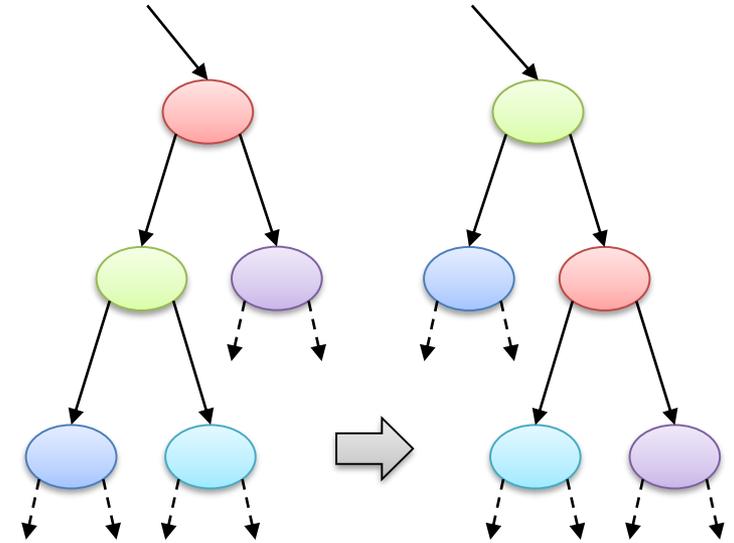
Treap: Heap-Eigenschaft

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:



Treap: Einfügen

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

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

Ziel: Lösche Schlüssel x

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

- 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...

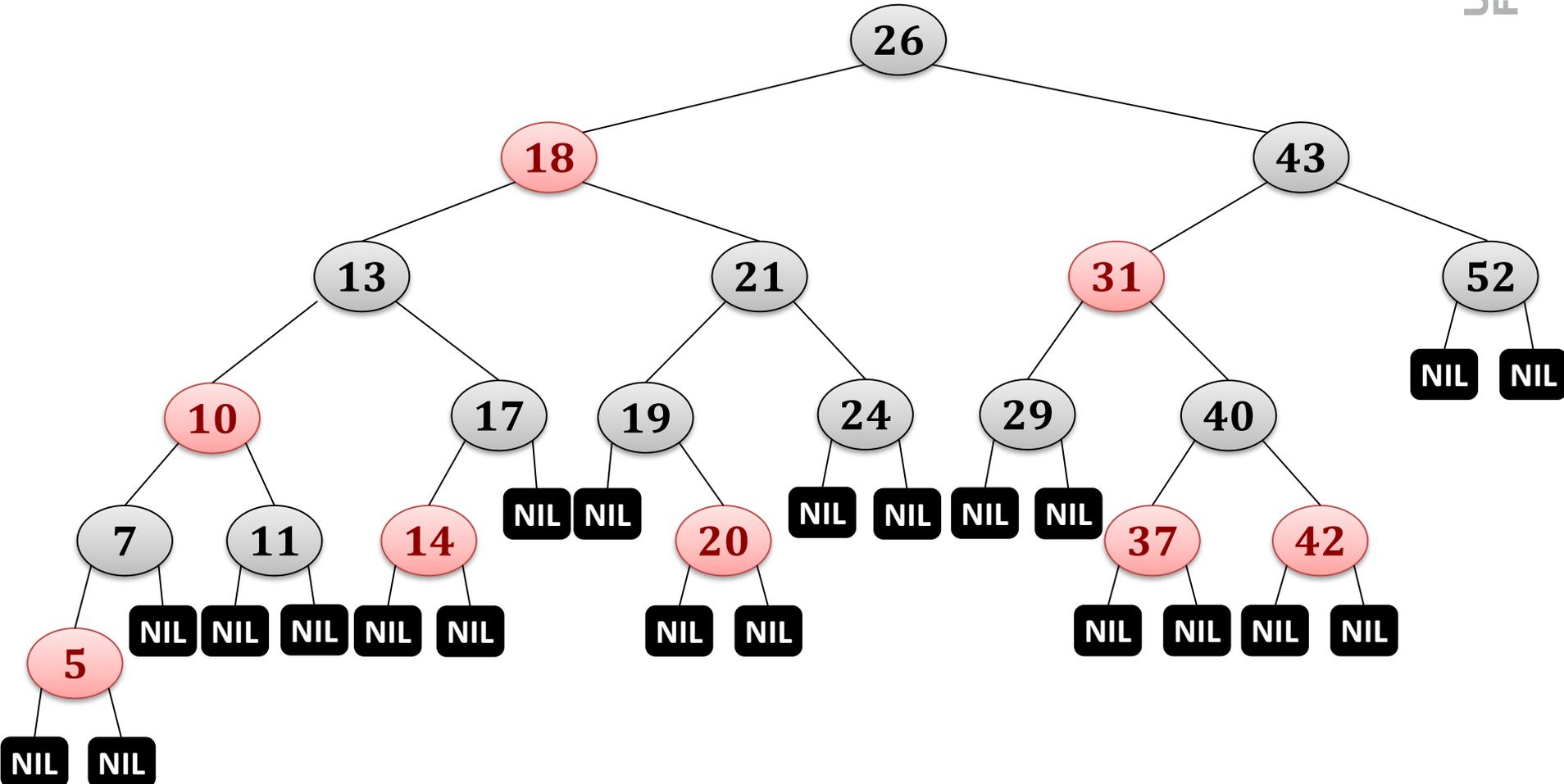
Ziel: Binäre Suchbäume, welche **immer balanciert** sind

- 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, 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

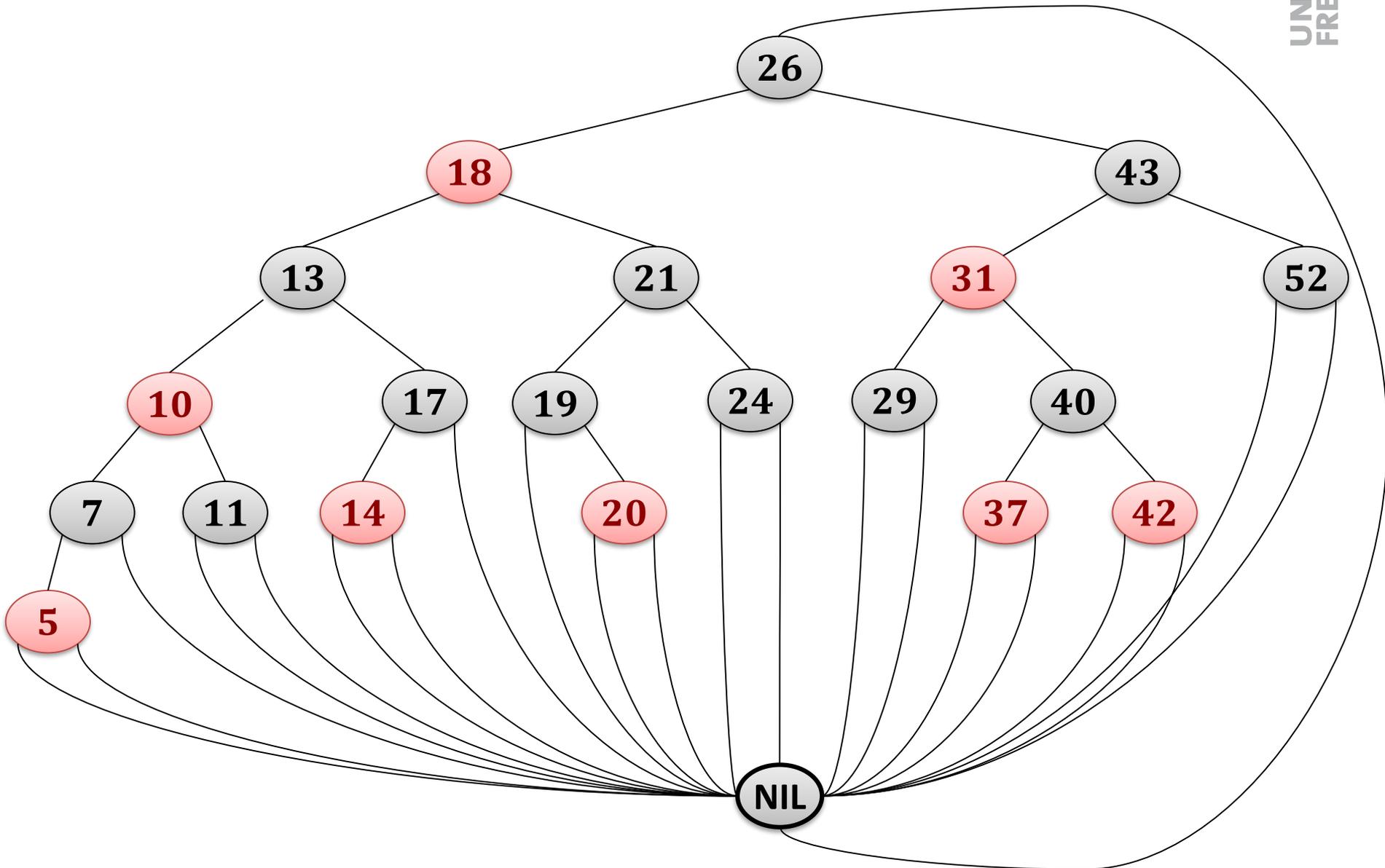


Wie bei den Treaps, um den Code zu vereinfachen...

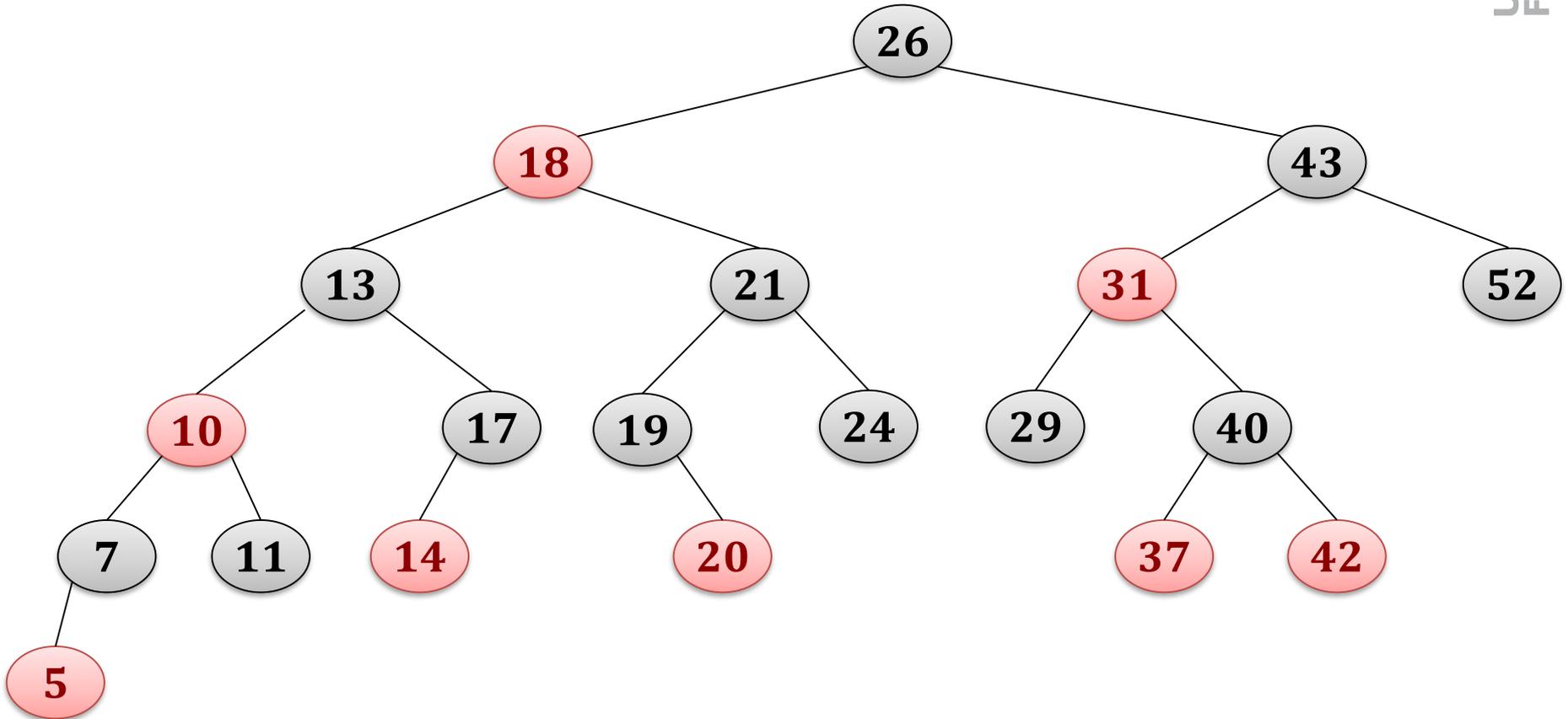
Sentinel-Knoten: *NIL*

- 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



Schwarz-Tiefe \leftrightarrow Anzahl Knoten

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

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

Beweis:

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

Schwarz-Tiefe \leftrightarrow Anzahl Knoten

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

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

Beweis:

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

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:

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