Algorithmentheorie

04 - Treaps

Robert Elsässer

Das Wörterbuch-Problem

Gegeben: Universum $(U,<)$ von Schlüsseln mit einer totalen Ordnung

Ziel: Verwalte Menge $S \subseteq U$, mit folgenden Operationen

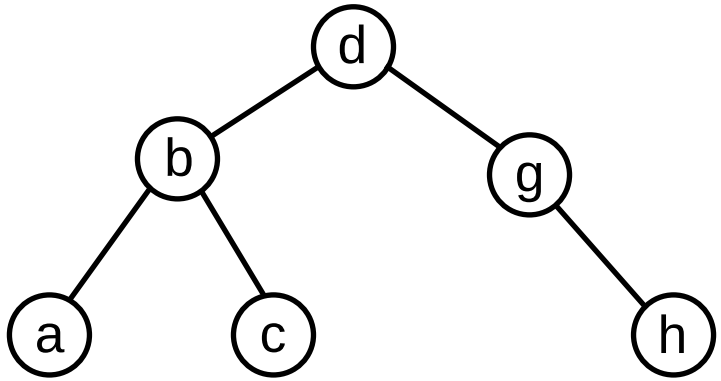
- Suche(x,S): Ist $x \in S$?
- Einfüge(x,S): Füge x zu S hinzu, sofern noch nicht vorhanden.
- Entferne(x,S): Entferne x aus S.

Erweiterte Operationen

- Minimum(S): Finde kleinsten Schlüssel.
- Maximum(S): Finde größten Schlüssel.
- List(S): Gib Einträge in aufsteigender Reihenfolge aus.
- Vereinige(S_1,S_2): Vereinige S_1 und S_2.
  Voraussetzung: $\forall\, x_1 \in S_1\,, x_2 \in S_2$: $x_1 < x_2$
- Spalte(S,x,S_1,S_2): Spalte S in S_1 und S_2.
  $\forall\, x_1 \in S_1\,, x_2 \in S_2$: $x_1 \leq x$ und $x_2 > x$

Bekannte Lösungen

- **Binäre Suchbäume**



Nachteil: Sequenz von Einfügungen kann zu einer linearen Liste führen a, b, c, d, e, f

- **Höhenbalancierte Bäume:** AVL-Bäume, (a,b)-Bäume
  Nachteil: Komplexe Algorithmen oder hoher Speicherbedarf

Ansatz für randomisierte Suchbäume

Werden n Elemente in zufälliger Reihenfolge in einen binären Suchbaum eingefügt, so ist die erwartete Tiefe 1,39 log n.

Idee: Jedes Element x erhält eine zufällig gewählte

Priorität prio(x) $\in R$

Ziel ist es, die folgende Eigenschaft herzustellen.

(*) Der Suchbaum hat die Struktur, die entstanden wäre, wenn die Elemente in der durch die Prioritäten gegebenen Reihenfolge eingefügt worden wären.

Treaps (Tree + Heap)

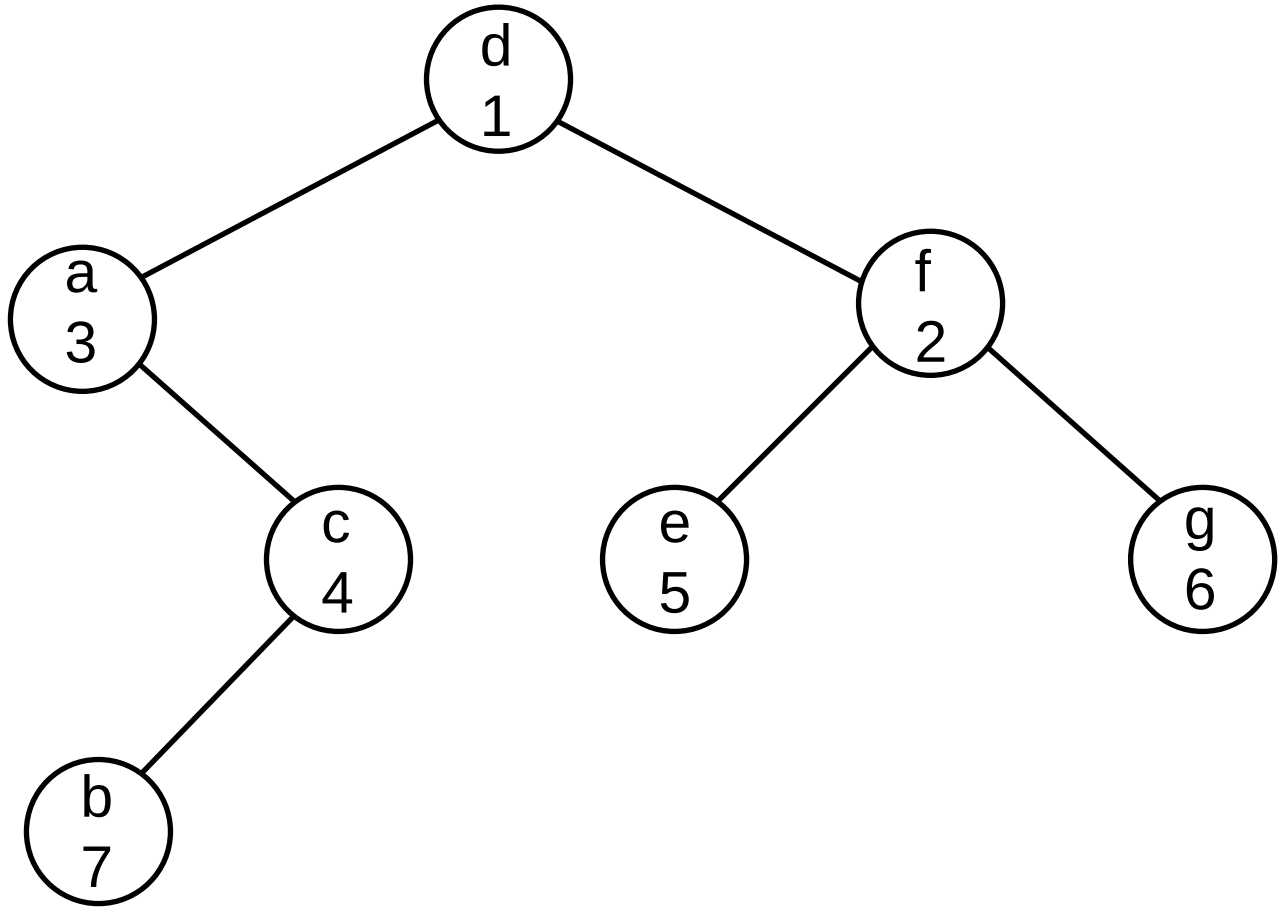
Definition: Ein Treap ist ein binärer Baum.
Jeder Knoten enthält ein Element x mit $key(x) \in U$ und $prio(x) \in R$.
Es gelten die folgenden Eigenschaften.

- Suchbaum-Eigenschaft
  Für jedes Element x gilt:
  - Elemente y im linken Teilbaum von x erfüllen: $key(y) < key(x)$
  - Elemente y im rechten Teilbaum von x erfüllen: $key(y) > key(x)$

- Heap-Eigenschaft
  Für alle Elemente x,y gilt:
  Ist y ein Kind von x, so ist $prio(y) > prio(x)$.
  Alle Prioritäten sind verschieden.

Beispiel

Schlüssel	a	b	c	d	e	f	g
Priorität	3	7	4	1	5	2	6



Eindeutigkeit von Treaps

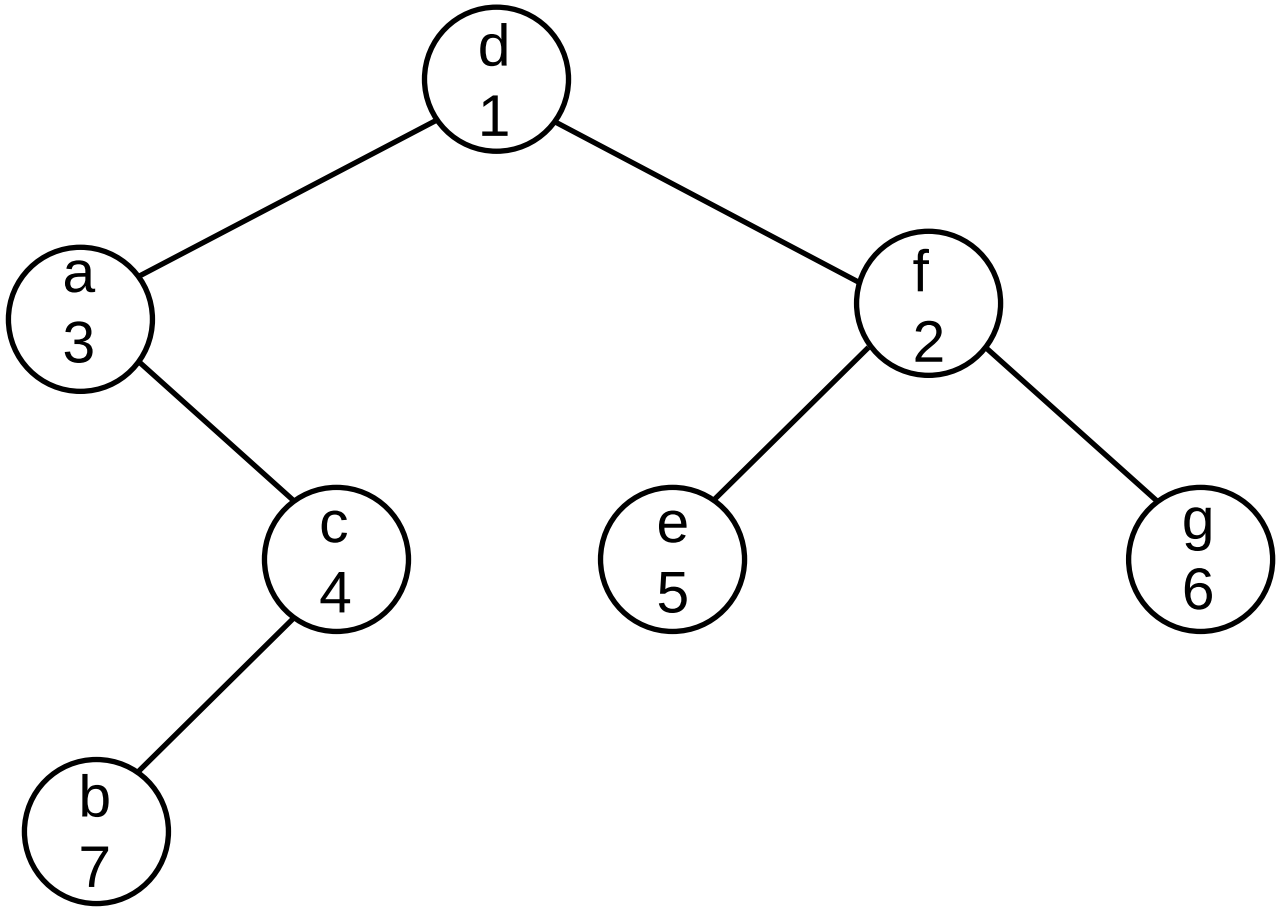
Lemma: Für Elemente $x_1, \ldots, x_n$ mit key(x_i) und prio(x_i) existiert ein eindeutiger Treap. Dieser erfüllt Eigenschaft (*).

Beweis:

n=1: ok

n>1:

Suche nach einem Element



Suche nach Element mit Schüssel *k*

```
1  v := Wurzel;
2  while v ≠ nil do
3     case key(v) =  k : stop;   "Element gefunden" (erfolgreiche Suche)
4             key(v) <  k : v:= RechtesKind(v);
5             key(v) >  k : v:= LinkesKind(v);
6     endcase;
7  endwhile;
8  "Element nicht gefunden" (nicht erfolgreiche Suche)
```

Laufzeit: O(# Elemente auf dem Suchpfad)

Analyse des Suchpfads

Elemente $x_1, \ldots, x_n$ x_i hat *i*-t kleinsten Schlüssel

M sei Teilmenge der Elemente

$P_{min}(M)$ = Element aus *M* mit kleinster Priorität

Lemma:

a) Sei $i<m$. x_i ist Vorfahre von x_m gdw $P_{min}(\{x_i,\ldots,x_m\}) = x_i$

b) Sei $m<i$. x_i ist Vorfahre von x_m gdw $P_{min}(\{x_m,\ldots,x_i\}) = x_i$

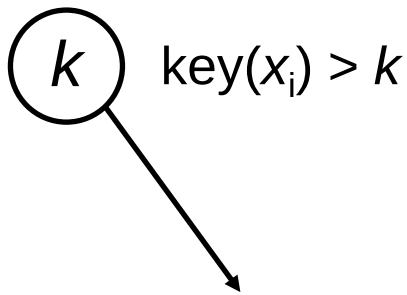
Analyse des Suchpfads

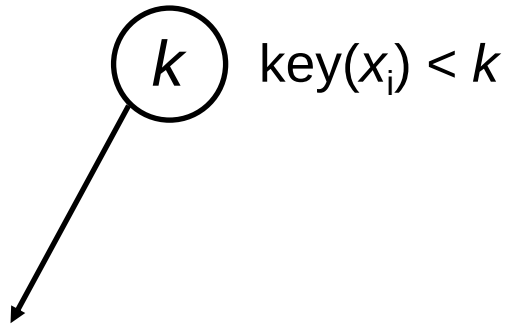
Beweis: a) Verwende (*). El. werden gemäß steigenden Prioritäten eingefügt.

"$\Leftarrow$" $P_{min}(\{x_i,\ldots,x_m\}) = x_i \quad \Rightarrow \quad x_i$ wird als erstes aus $\{x_i,\ldots,x_m\}$ eingefügt

Wenn x_i eingefügt wird, enthält der Baum nur Schlüssel k mit

$k < \text{key}(x_i)$ oder $k > \text{key}(x_m)$





Analyse des Suchpfads

Beweis: a) (Sei $i<m$. x_i ist Vorfahre von x_m gdw $P_{min}(\{x_i,\ldots,x_m\}) = x_i$)

"$\Rightarrow$" Sei $x_j = P_{min}(\{x_i,\ldots,x_m\})$. Z.z. $x_i = x_j$

Annahme: $x_i \neq x_j$

Fall 1: $x_j = x_m$

Fall 2: $x_j \neq x_m$

Teil b) analog.

Analyse der Suche-Operation

Sei T ein Treap mit Elementen $x_1, \ldots, x_n$ x_i hat i-t kleinsten Schlüssel

n-te Harmonische Zahl: $H_n = \sum_{k=1}^{n} 1/k$

Lemma:

1. Erfolgreiche Suche: Die erwartete Anzahl von Knoten auf dem Pfad nach x_m ist $H_m + H_{n-m+1} - 1$.

2. Nicht erfolgreiche Suche: Sei m die Anzahl der Schlüssel, die kleiner als der gesuchte Schlüssel k sind. Die erwartete Anzahl von Knoten auf dem Suchpfad ist $H_m + H_{n-m}$.

Analyse der Suche-Operation

Beweis: Teil 1

$$X_{m,i} = \begin{cases} 1 & x_i \text{ ist Vorfahre von } x_m \\ 0 & \text{sonst} \end{cases}$$

X_m = # Knoten auf Pfad von der Wurzel nach x_m (inkl. x_m)

$$X_m = 1 + \sum_{i<m} X_{m,i} + \sum_{i>m} X_{m,i}$$

$$E[X_m] = 1 + E\left[\sum_{i<m} X_{m,i}\right] + E\left[\sum_{i>m} X_{m,i}\right]$$

Analyse der Suche-Operation

$i < m$:

$$E[X_{m,i}] = \text{Prob}[x_i \text{ ist Vorfahre von } x_m] = 1/(m-i+1)$$

Alle El. aus $\{x_i, \ldots, x_m\}$ haben mit gleicher WSK die kleinste Priorität

$\text{Prob}[P_{min}(\{x_i,\ldots,x_m\}) = x_i] = 1/(m\text{-}i\text{+}1)$

$i > m$:

$$E[X_{m,i}] = 1/(i-m+1)$$

Analyse der Suche-Operation

$$E[X_m] = 1 + \sum_{i<m} \frac{1}{m-i+1} + \sum_{i>m} \frac{1}{i-m+1}$$

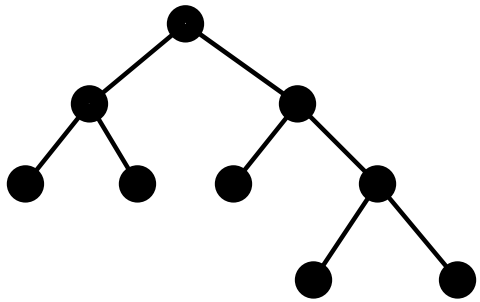
$$= 1 + \frac{1}{m} + \ldots + \frac{1}{2} + \frac{1}{2} + \ldots + \frac{1}{n-m+1}$$

$$= H_m + H_{n-m+1} - 1$$

Teil 2 analog

Einfügen eines neuen Elements *x*

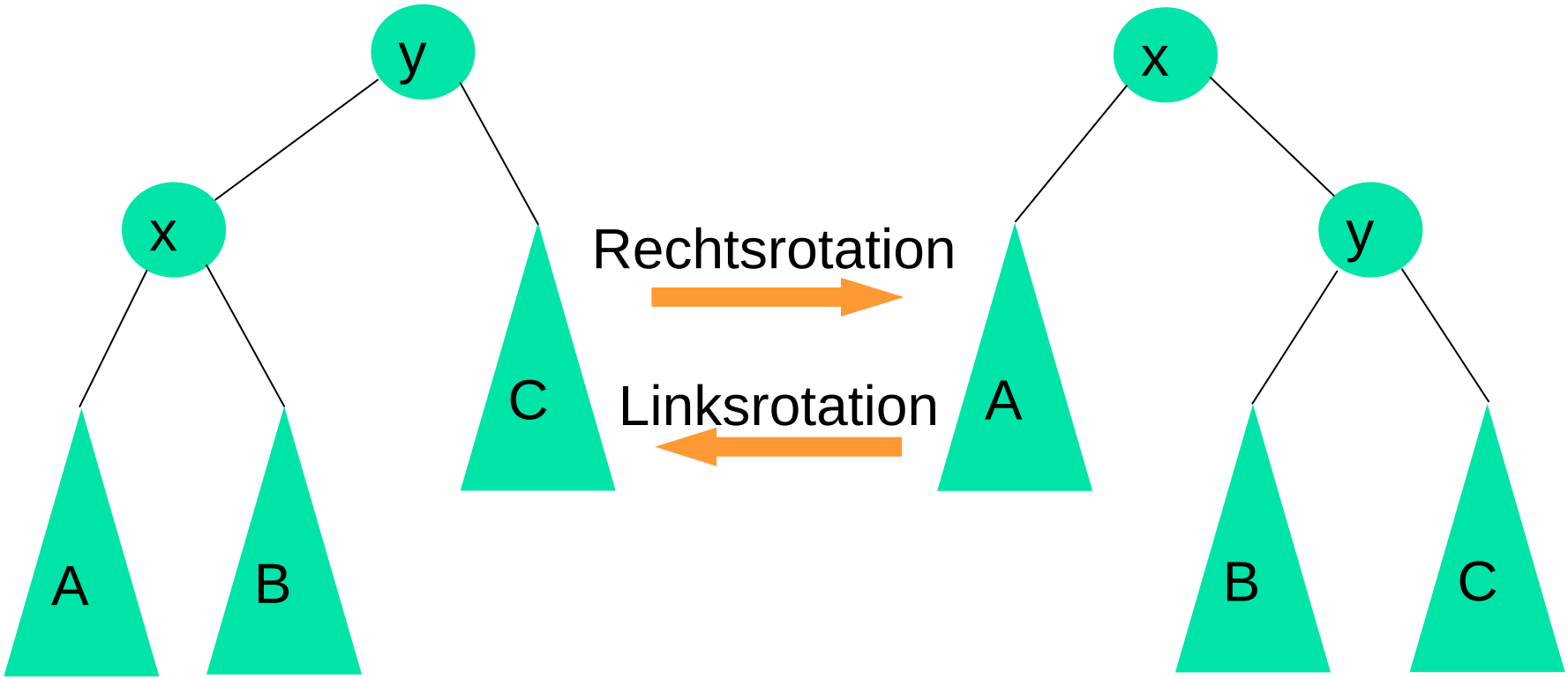
1. Wähle prio(x).
2. Suche nach der Position von *x* im Baum.



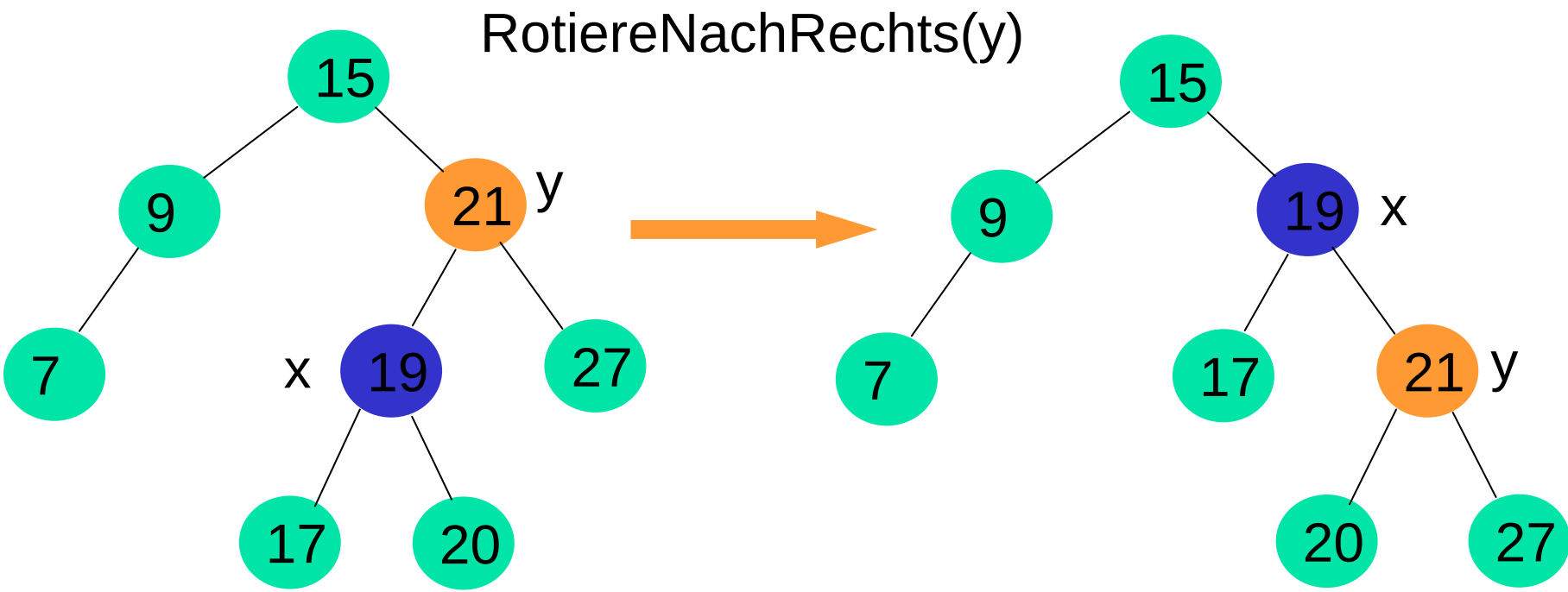
3. Füge *x* als Blatt ein.
4. Stelle Heap-Eigenschaft wieder her.

```
while prio(Elter(x)) > prio(x) do
    if x ist linkes Kind then RotiereNachRechts(Elter(x))
                          else RotiereNachLinks(Elter(x));
    endif
endwhile;
```

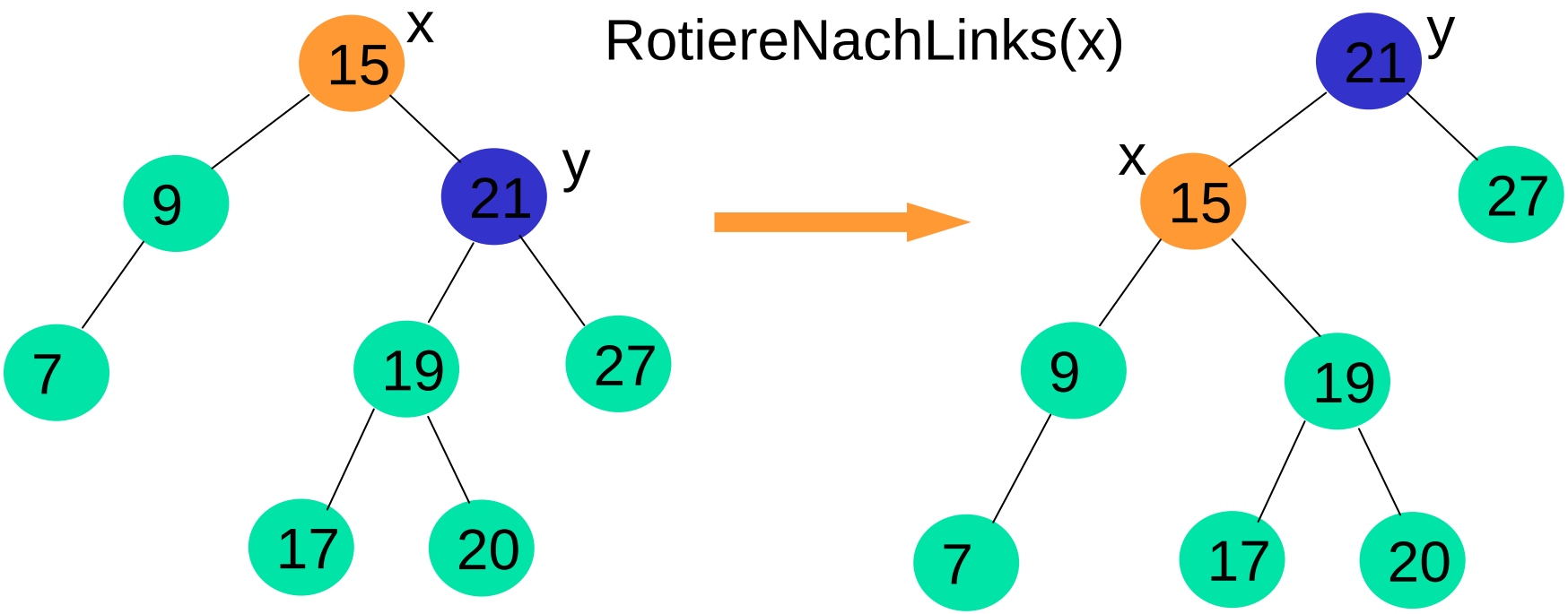
Rotationen



Rotationen


RotiereNachRechts(y)
15
9
21
y
7
x
19
27
17
20
15
9
19
x
7
17
21
y
20
27

Rotationen



Rotationen

RotiereNachLinks(x)

1. y ← RechtesKind(x)
2. RechtesKind(x) ← LinkesKind(y)
3. **if** LinkesKind(y)≠**nil then** Elter(LinkesKind(y)) ← x
4. Elter(y) ← Elter(x)
5. **if** Elter(x)=**nil then** root(T) ← y
6. **else if** x=LinkesKind(Elter(x)) **then** LinkesKind(Elter(x)) ← y
7. **else** RechtesKind(Elter(x)) ← y
8. LinkesKind(y) ← x
9. Elter(x) ← y

Rotationen

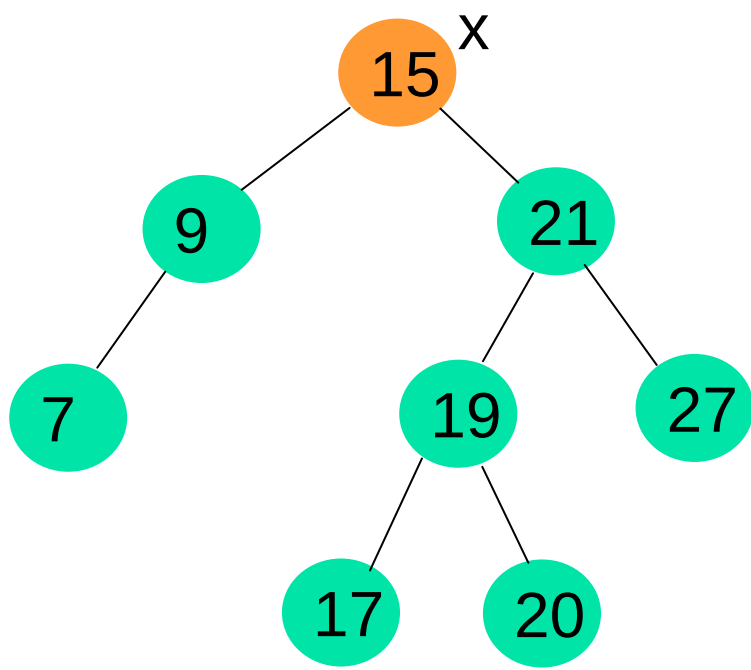
RotiereNachLinks(x) // **lc – LinkesKind**

1. y ← rc(x) // **rc – RechtesKind**
2. rc(x) ← lc(y) // **p - Elter**
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y

Rotationen

RotiereNachLinks(x)

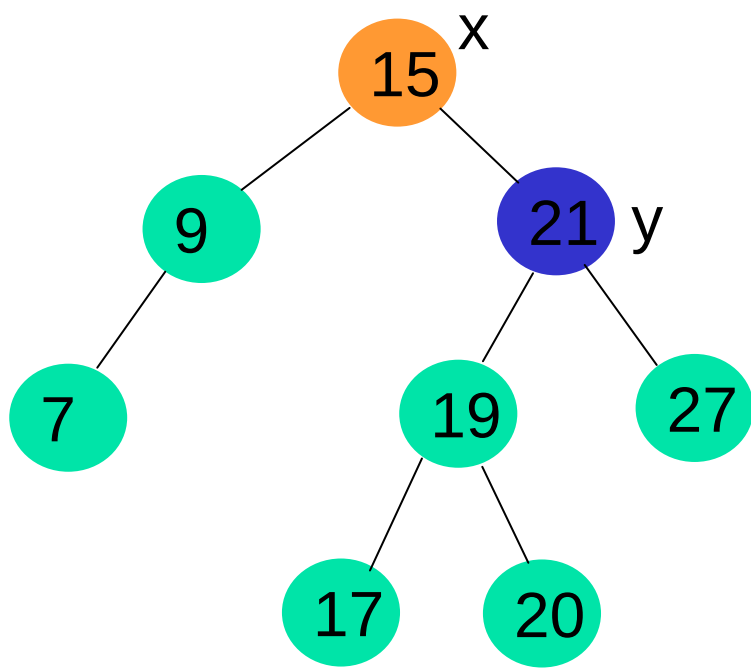
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

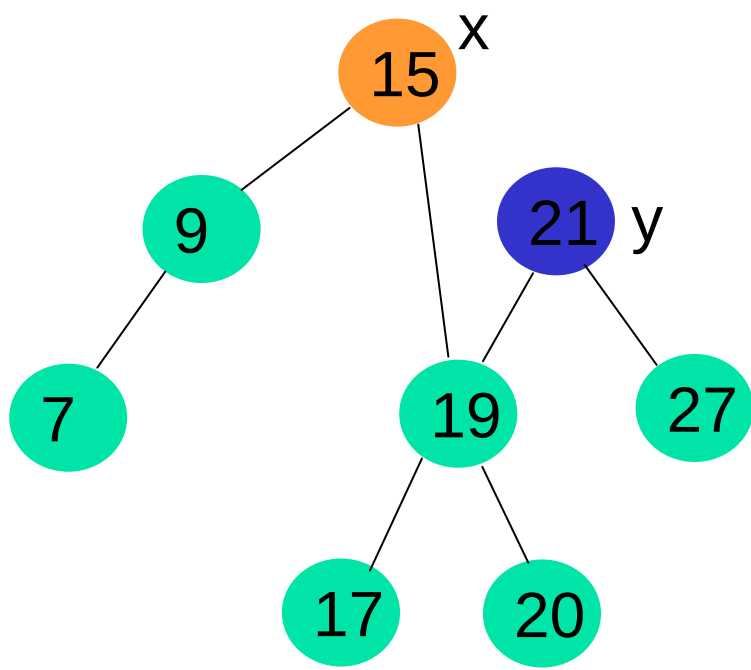
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

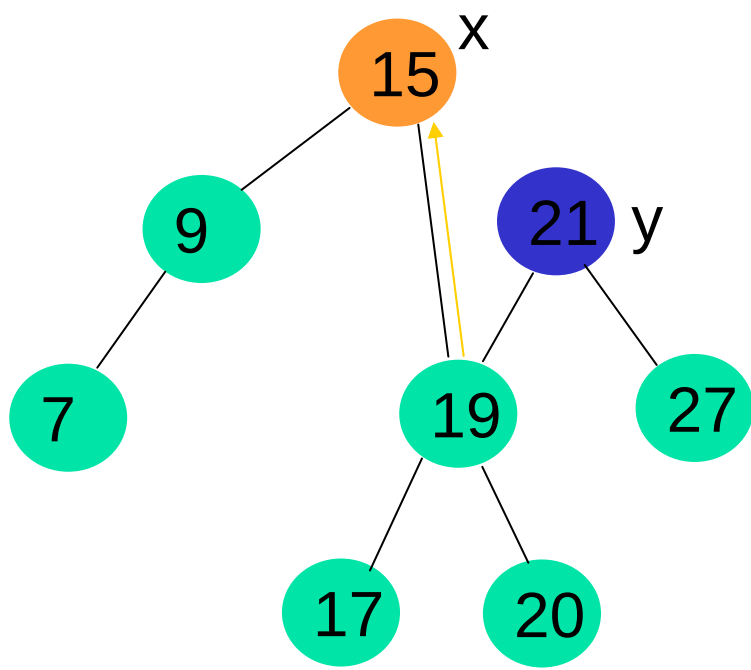
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

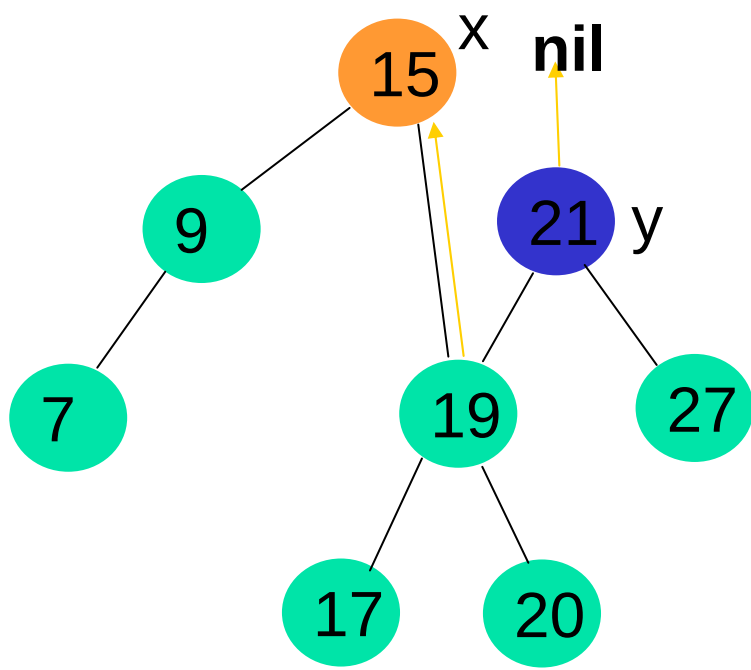
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

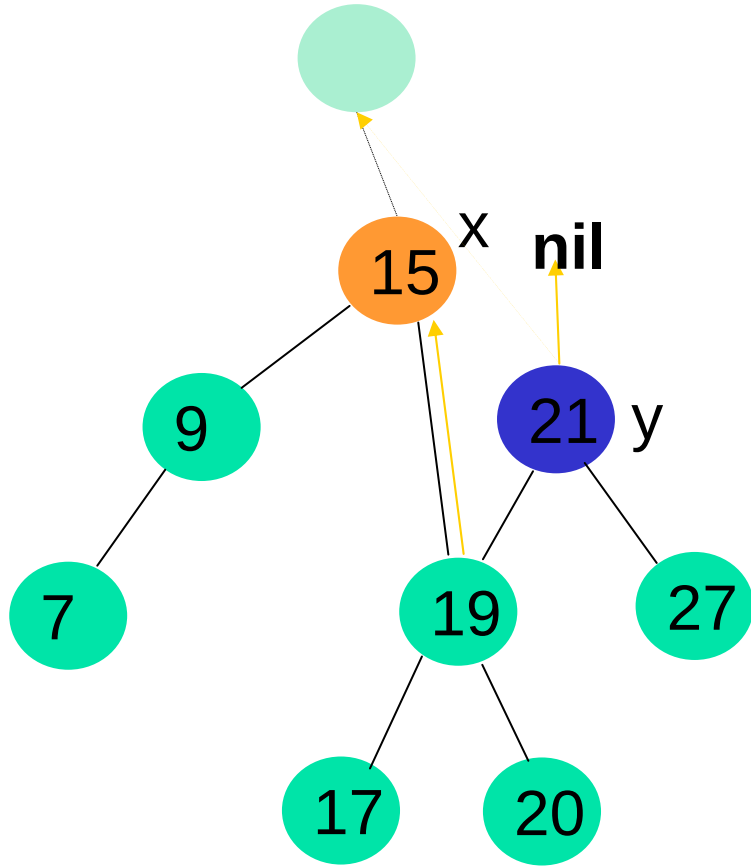
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

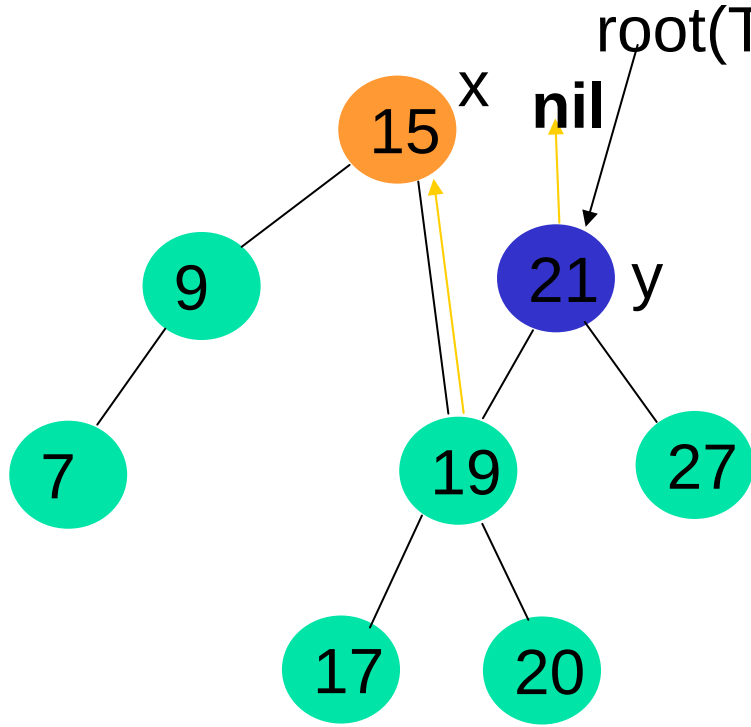
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

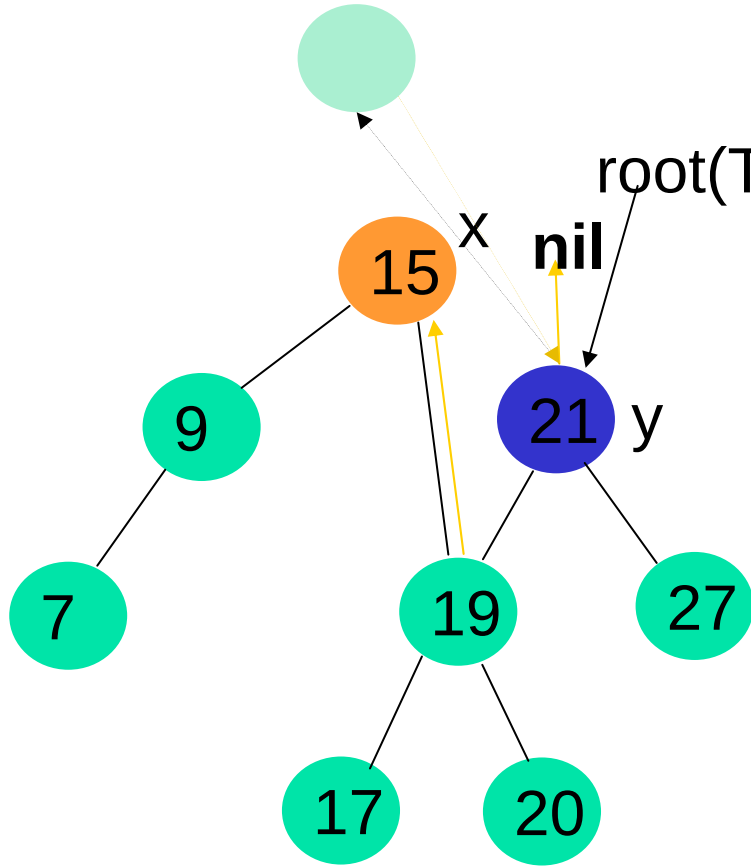
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

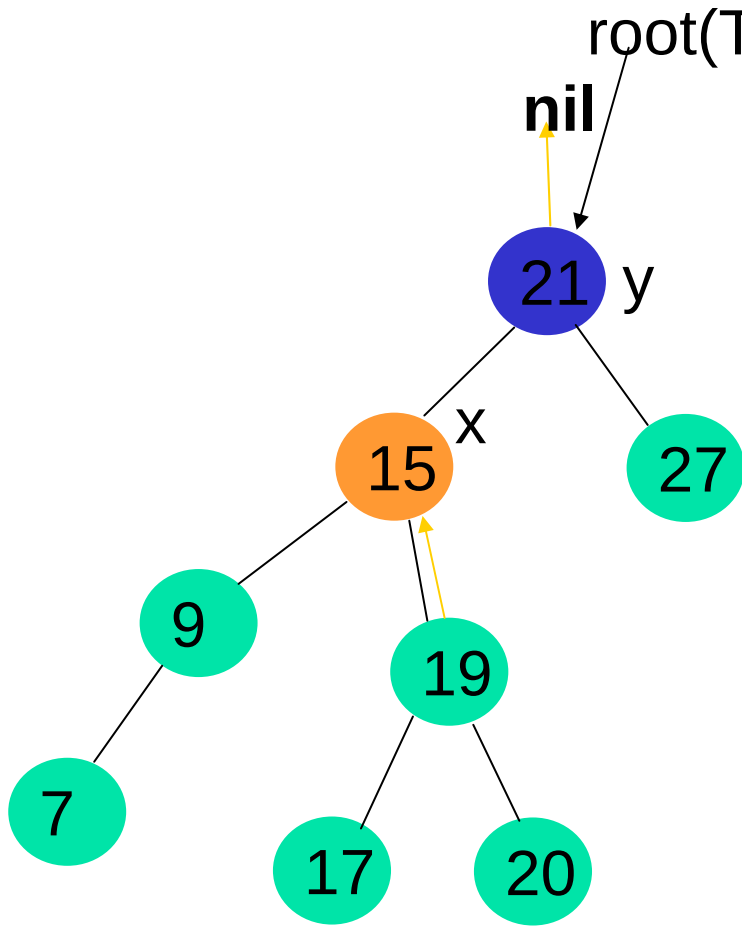
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

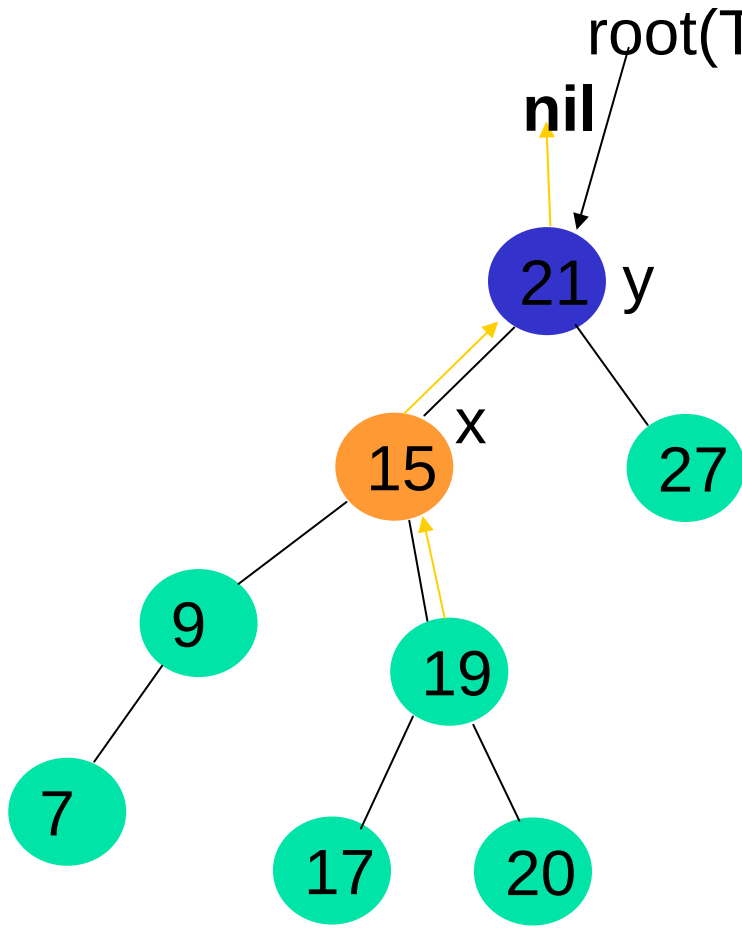
1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y



Rotationen

RotiereNachLinks(x)

1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y

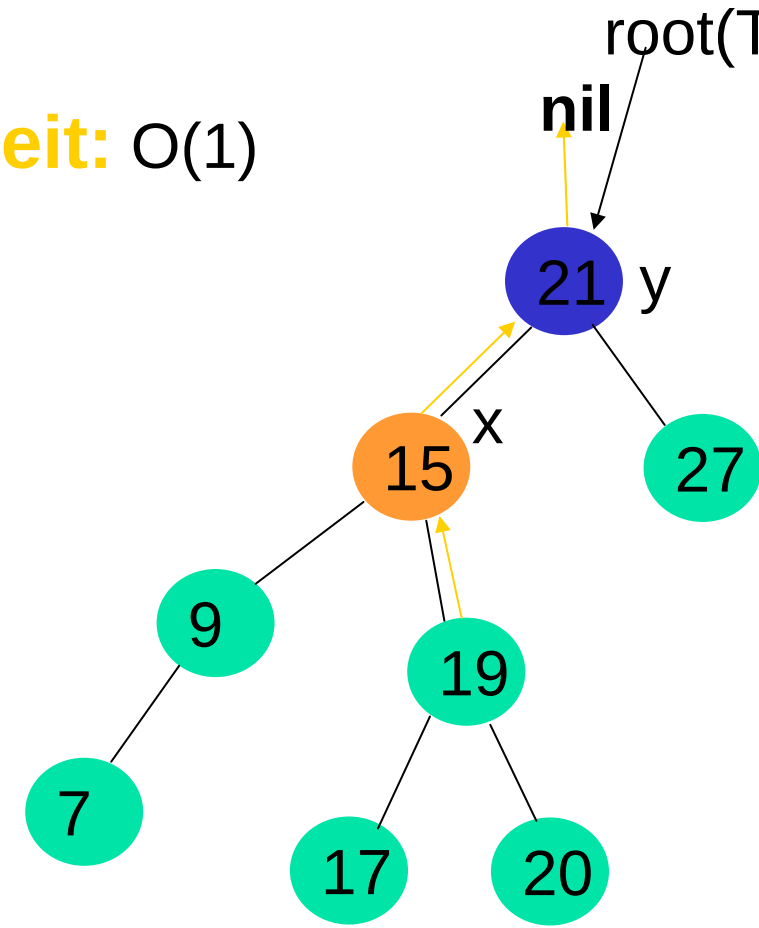


Rotationen

RotiereNachLinks(x)

1. y ← rc(x)
2. rc(x) ← lc(y)
3. **if** lc(y)≠**nil then** p(lc(y)) ← x
4. p(y) ← p(x)
5. **if** p(x)=**nil then** root(T) ← y
6. **else if** x=lc(p(x)) **then** lc(p(x)) ← y
7. **else** rc(p(x)) ← y
8. lc(y) ← x
9. p(x) ← y

Laufzeit: O(1)



Entfernen eines Elements *x*

1. Suche *x* in dem Baum.
2. **while** *x* ist kein Blatt **do**
   u := Kind mit kleinerer Priorität;
   if *u* ist linkes Kind **then** RotiereNachRechts(*x*))
   else RotiereNachLinks(*x*);
   endif;
   endwhile;
3. Entferne x;

Analyse der Einfüge- u. Entferne-Operationen

Lemma: Die erwartete Laufzeit einer Einfüge- bzw. Entferne-Operation ist O(log *n*). Die erwartete Anzahl der Rotationen ist 2.

Beweis: Analyse einer Einfügung (Entfernung ist inverse Operation)

#Rotationen = Tiefe von *x* nach Einfügung als Blatt (1)

\- Tiefe von *x* nach den Rotationen (2)

Sei $x = x_m$

(2) erwartete Tiefe ist $H_m + H_{n-m+1} - 1$

(1) erwartete Tiefe ist $H_{m-1} + H_{n-m} + 1$

Baum enthält *n-1* Elemente, *m-1* sind kleiner.

#Rotationen = $H_{m-1} + H_{n-m} + 1 - (H_m + H_{n-m+1} - 1) < 2$

Erweiterte Operationen

n = Anzahl Elemente im Treap T.

- Minimum(T): Finde kleinsten Schlüssel. O(log n)
- Maximum(T): Finde größten Schlüssel. O(log n)
- List(T): Gib Einträge in aufsteigender Reihenfolge aus. O(n)

- Vereinige(T_1,T_2): Vereinige T_1 und T_2.
  Voraussetzung: $\forall\, x_1 \in T_1 , x_2 \in T_2$: $\text{key}(x_1) < \text{key}(x_2)$
- Spalte(T,k,T_1,T_2): Spalte T in T_1 und T_2.
  $\forall\, x_1 \in T_1 , x_2 \in T_2$: $\text{key}(x_1) \leq k$ und $\text{key}(x_2) > k$

Die Spalte-Operation

Spalte(T,k,T_1,T_2): Spalte T in T_1 und T_2.

$\forall\, x_1 \in T_1\,, x_2 \in T_2$: key($x_1$) $\leq k$ und key(x_2) $> k$

O.B.d.A. Schlüssel k nicht in T.

Andernfalls lösche Element mit Schlüssel k und füge es nach der Spalte-Operation in T_1 ein.

1. Erzeuge neues Element x mit key(x)=k und prio(x) = $-\infty$.
2. Füge x in T ein.
3. Entferne die neue Wurzel. Der linke Unterbaum ist T_1, der rechte T_2.

Die Vereinige-Operation

Vereinige(T_1,T_2): Vereinige T_1 und T_2.

Voraussetzung: $\forall\, x_1 \in T_1\,, x_2 \in T_2$: key($x_1$) < key($x_2$)

1. Ermittle Schlüssel k mit key(x_1) < k < key(x_2)
   für alle $x_1 \in T_1$ und $x_2 \in T_2$.
2. Erzeuge Element x mit key(x)=k und prio(x) = -∞.
3. Erzeuge Treap T mit Wurzel x, linkem Teilbaum T_1 und rechtem Teilbaum T_2.
4. Entferne x aus T.

Analyse

Lemma: Die Operationen Vereinige und Spalte haben eine erwartete Laufzeit von $O(\log n)$.

Praktische Realisierung

Prioritäten aus [0,1)

Prioritäten werden nur im direkten Vergleich benutzt, um zu entscheiden, welches Element die kleinere Priorität hat.

Tritt Gleichheit auf, so würfele für beide Prioritäten weitere Bits aus, bis sie verschieden sind.

$p_1 = 0{,}010111001$
$p_2 = 0{,}010111001$

$p_1 = 0{,}010111001011$
$p_2 = 0{,}010111001010$