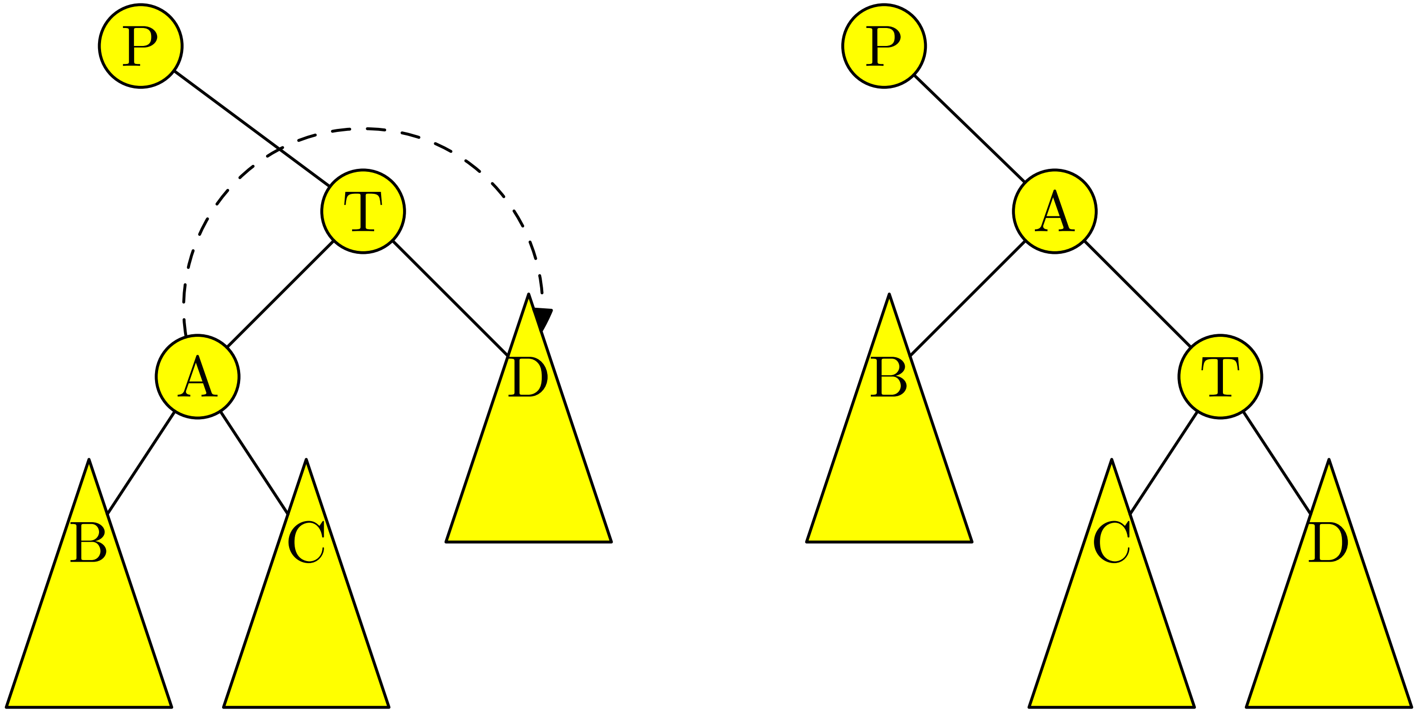
Rotate-Down



Die Heapeigenschaft sei erfüllt, außer für T:
T habe eine zu große Priorität.

Nach der Rotation gilt wieder, daß die Heapeigenschaft erfüllt ist, außer für T.

Wir können T nach unten rotieren, bis es ein Blatt ist.

Rotate-Down

Java

```
void rotate_down(Treapnode⟨K, D⟩ n) {
  while(true) {
    if(n.left ≠ null) {
      if(n.right ≡ null ||
        ((Treapnode⟨K, D⟩)n.left).weight ≤
        ((Treapnode⟨K, D⟩)n.right).weight) n.rotateright();
      else n.rotateleft();
    }
    else if(n.right ≡ null) break;
    else n.rotateleft();
  }
  repair_root();
}
```

Treaps – Löschen

Java

```
public void delete(K k) {
    if(root ≡ null) return;
    Treapnode⟨K, D⟩ n = (Treapnode⟨K, D⟩)root.findsubtree(k);
    if(n ≡ null) return;
    rotate_down(n);
    super.delete(k);
}
```

Wir rotieren den Knoten nach unten und entfernen ihn dann.

Treaps – Einfügen

Einfügen ist das Gegenteil von Löschen.

Löschen:

1. Knoten nach unten rotieren
2. Als Blatt entfernen

Einfügen:

1. Knoten als Blatt einfügen
2. Nach oben zur richtigen Position rotieren

Treaps – Einfügen

Java

```
void rotate_up(Treapnode⟨K, D⟩ n) {
  while(true) {
    if(n.parent ≡ null) break;
    if(((Treapnode⟨K, D⟩)n.parent).weight ≤ n.weight) break;
    if(n.parent.right ≡ n) n.parent.rotateleft();
    else n.parent.rotateright();
  }
  repair_root();
}
```

Wir rotieren nach oben bis wir die Wurzel erreichen oder der Elternknoten kleinere Priorität hat.

Treaps – Einfügen

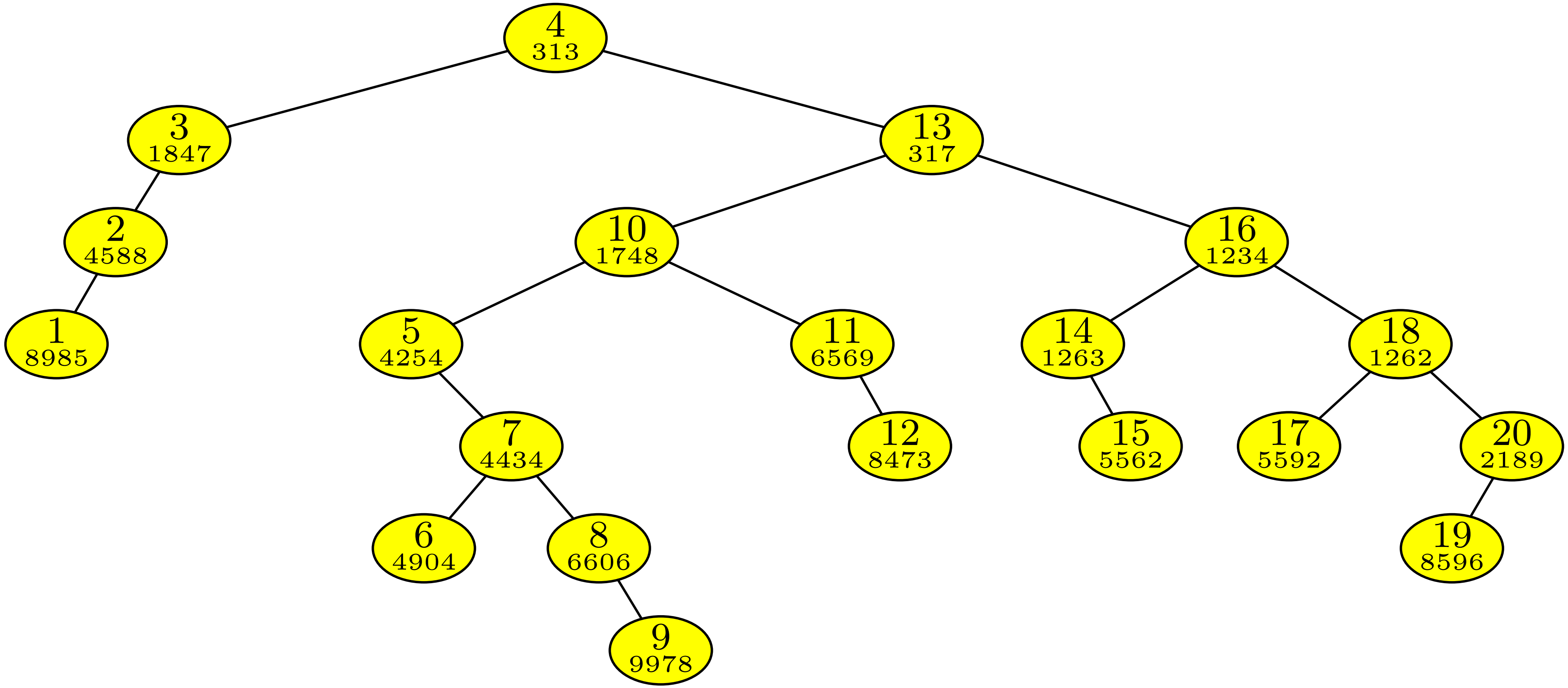
Java

```
public void insert(K k, D d) {
  if(root == null) root = new Treapnode<K, D>(k, d, generator);
  else {
    root.insert(new Treapnode<K, D>(k, d, generator));
    rotate_up((Treapnode<K, D>)root.findsubtree(k));
  }
}
```

1. Normal als Blatt einfügen
2. Hochrotieren

Einfügen von 20 Elementen in einen Treap

Die Prioritäten sind zufällig gewählt.



Treaps

Theorem

Einfügen, Suchen und Löschen in einen Treap mit n Elementen benötigt $O(\log n)$ Schritte im Mittel, falls die Prioritäten aus einem ausreichend großen Bereich zufällig gewählt werden.

- Treaps sind in der Praxis sehr schnell.
- Standardimplementation für assoziative Arrays in LEDA.
- Einfach zu analysieren.
- Einfach zu implementieren.
- Sehr hübsch.

Splay-Bäume

- Splay-Bäume sind eine selbstorganisierende Datenstruktur.
- Basiert auf binären Suchbäumen.
- Restrukturiert durch Rotationen.
- Keine Zusatzinformation in Knoten.
- Nur amortisiert effizient.
- Einfach zu implementieren.
- Viele angenehme Eigenschaften (z.B. „selbstlernend“)
- Nur eine komplizierte Operation: splay

Die Splay-Operation

Gegeben ist ein binärer Suchbaum und ein Knoten x.

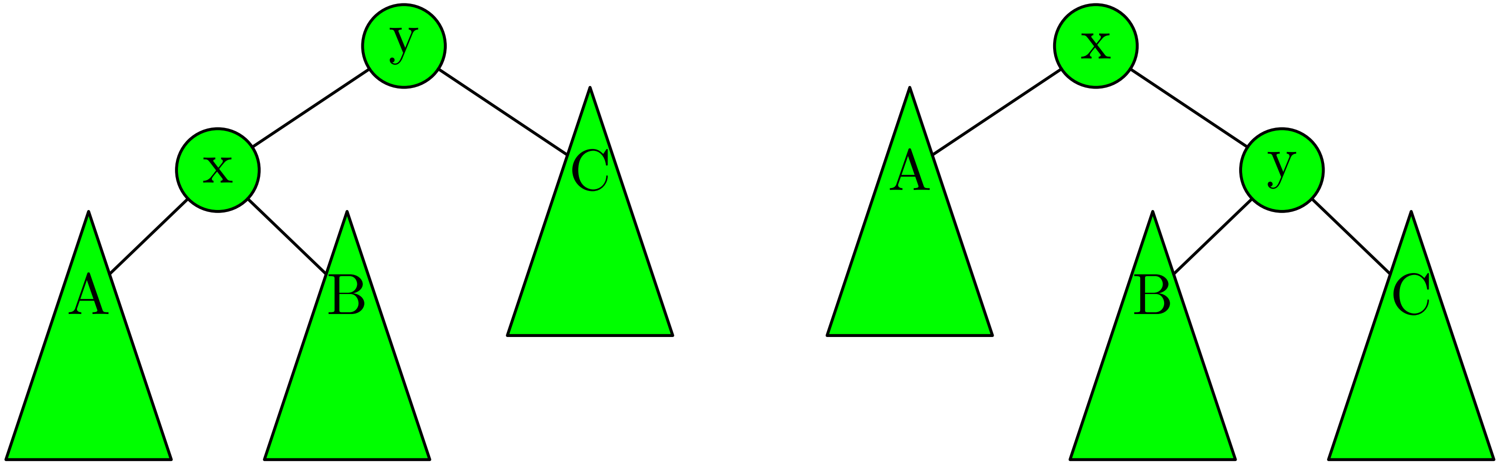
Algorithmus

procedure $splay(node\ x)$:
  while $x \neq root$ **do**
    $splaystep(x)$
  od

Wir führen Splay-Schritte auf x aus, bis es zur Wurzel wird.

Ein Splay-Schritt ist ein zig, zag, zig-zig, zig-zag, zag-zig oder zag-zag.

Zig und Zag

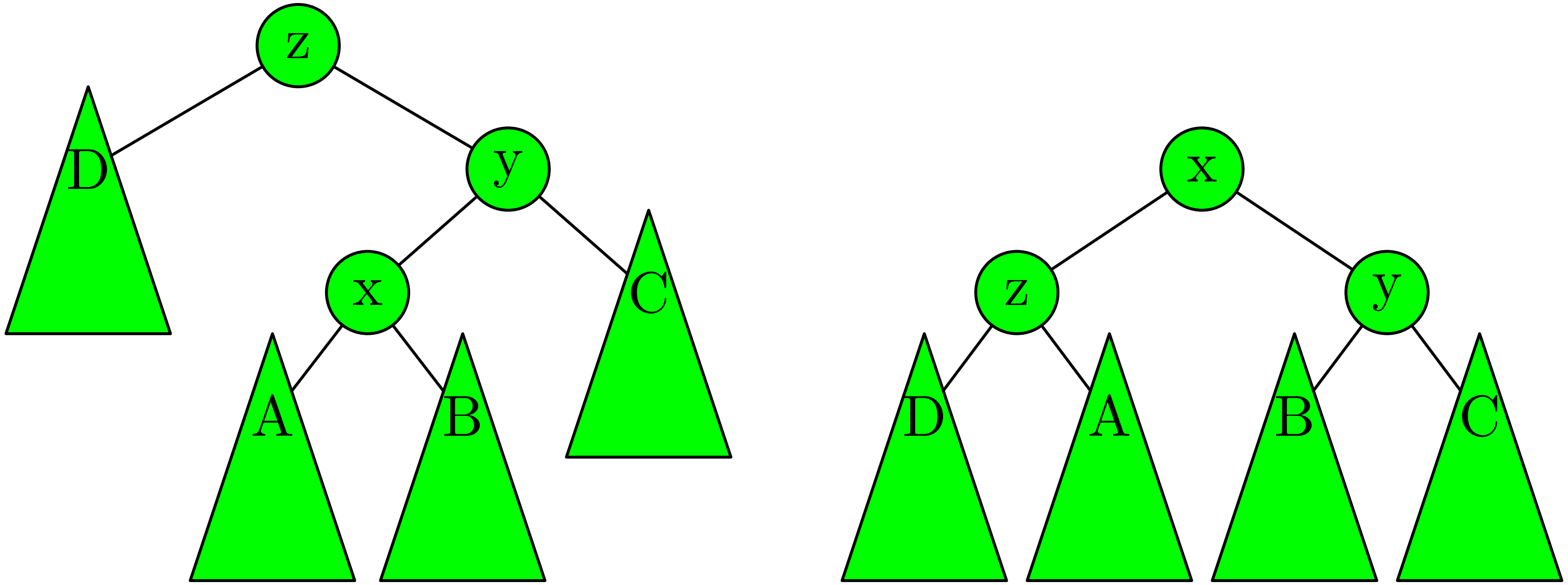


Ein zig auf x ist eine Rechtsrotation des Vaters von x.

Sie wird nur ausgeführt, wenn x das linke Kind der Wurzel ist.

Ein zag ist eine Linksrotation des Vaters, wenn x das rechte Kind der Wurzel ist.

Zig-zag und Zag-zig

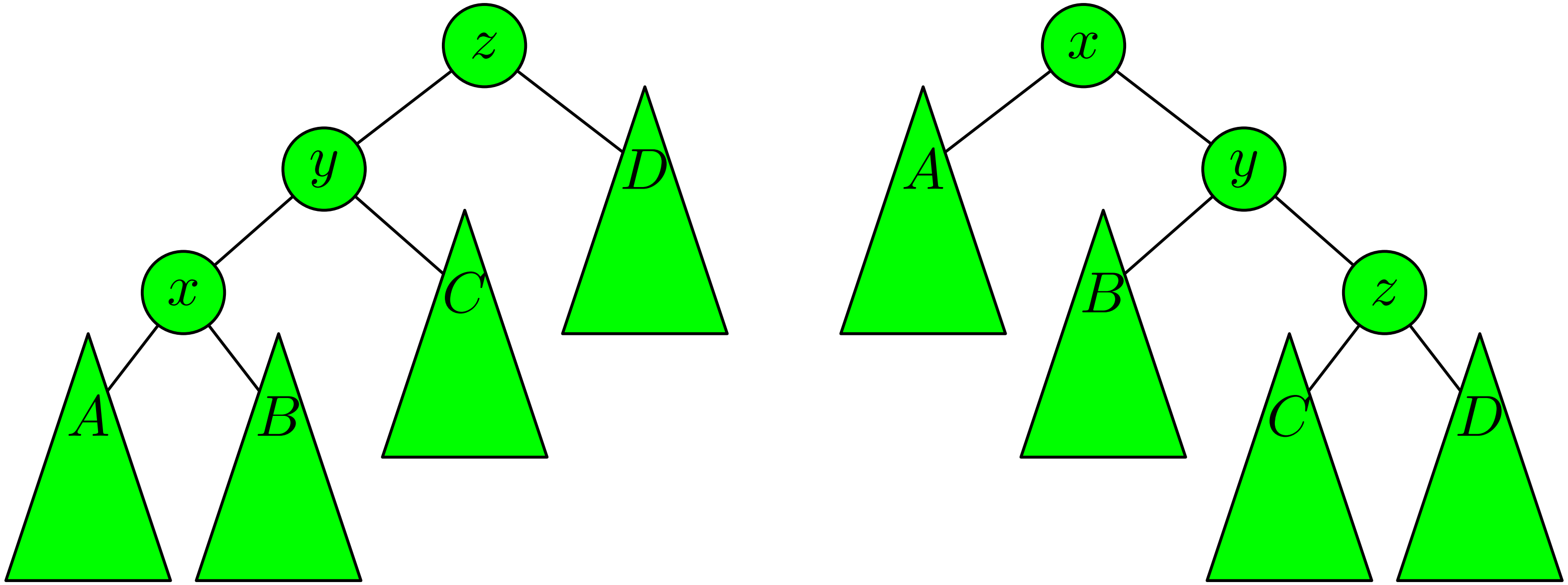


Ein Zig-zag auf x ist eine Rechtsrotation auf y gefolgt von einer Linksrotation auf z.

Dabei muß y das rechte Kind von z und x das linke Kind von y sein.

Zag-zig ist symmetrisch hierzu.

Zig-zig und Zag-zag



Ein Zig-zig auf x ist eine Rechtsrotation auf z gefolgt von einer Rechtsrotation auf y.

Dabei muß y das linke Kind von z und x das linke Kind von y sein.

Diese Operation sieht unerwartet aus!

Zag-zag ist wieder symmetrisch hierzu.

Amortisierte Analyse

Jeder Knoten x habe ein Gewicht $g(x) \in \mathbf{N}$.

Definition

- Der Kontostand eines Splay-Baums T ist $\sum_{v \in T} r(v)$.
- $r(v) = \left\lfloor \log(\bar{g}(v)) \right\rfloor$.
- $\bar{g}(v) = \sum_{u \in T(v)} g(u)$.
- $T(v)$ ist der Unterbaum mit Wurzel v

Amortisierte Analyse

Definition

Gegeben sei ein Splay-Schritt, der einen Splay-Baum T in einen Splay-Baum T' verwandelt.
Die amortisierten Kosten dieses Schrittes betragen

$$\sum_{v \in T'} r(v) - \sum_{v \in T} r(v) + 1.$$

Ein Schritt sind die tatsächlichen Kosten.

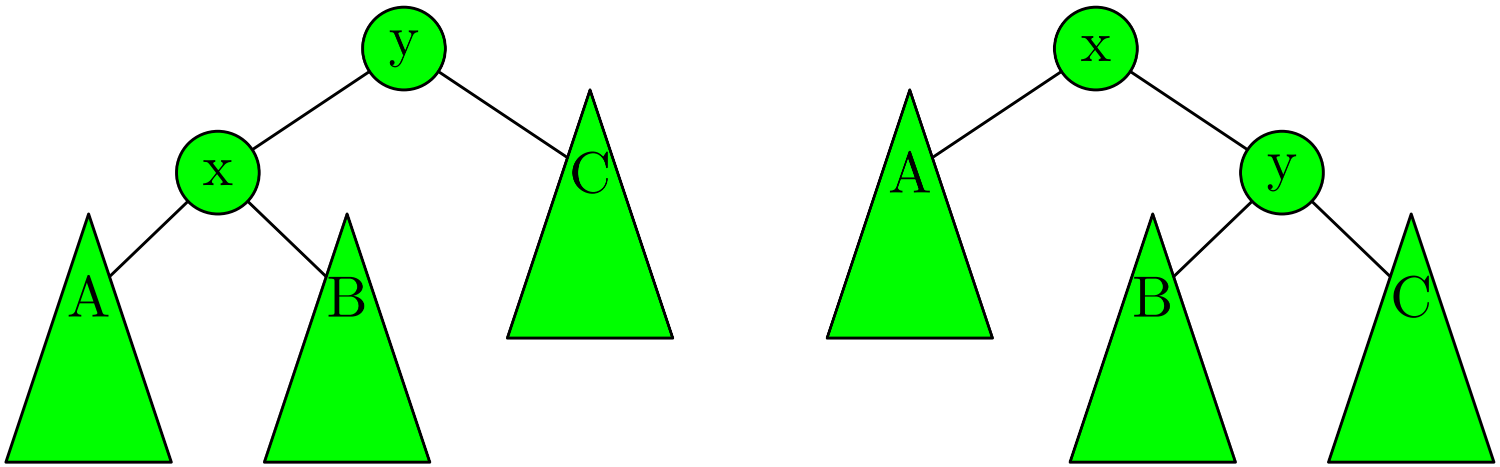
Der Rest ist eine Einzahlung oder Abhebung vom Konto.

Lemma

Die amortisierten Kosten eines

- *zig sind* $\leq 1 + 3(r(y) - r(x))$,
- *zig-zag sind* $\leq 3(r(z) - r(x))$,
- *zig-zig sind* $\leq 3(r(z) - r(x))$.

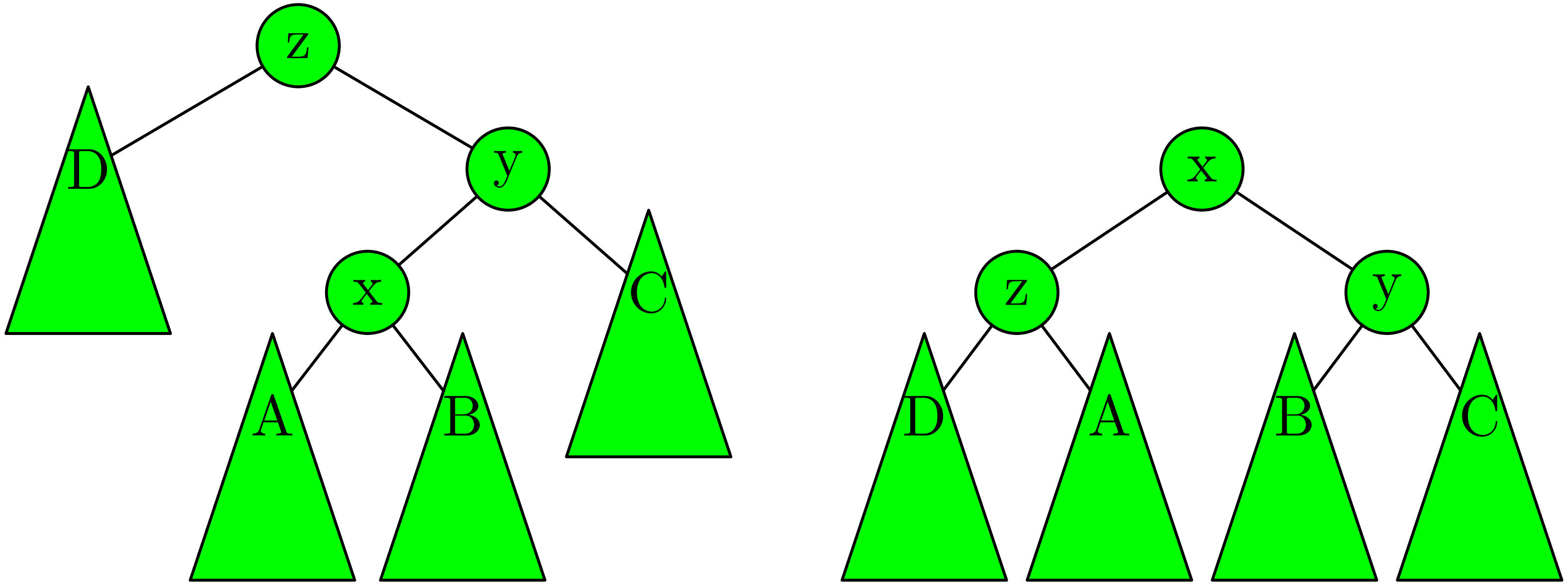
x, y, z sind die Knoten in den entsprechenden Zeichnungen.
Auf x wird die Operation ausgeführt.
z ist Großvater, y ist Vater von x.



- $r'(x) = r(y)$
- $r'(y) \leq r(y)$
- $r(y) \geq r(x)$

Die amortisierten Kosten eines zig sind

$$\begin{aligned} 1 + r'(x) - r(x) + r'(y) - r(y) &\leq 1 + r'(y) - r(x) \\ &\leq 1 + r(y) - r(x) \leq 1 + 3(r(y) - r(x)). \end{aligned}$$

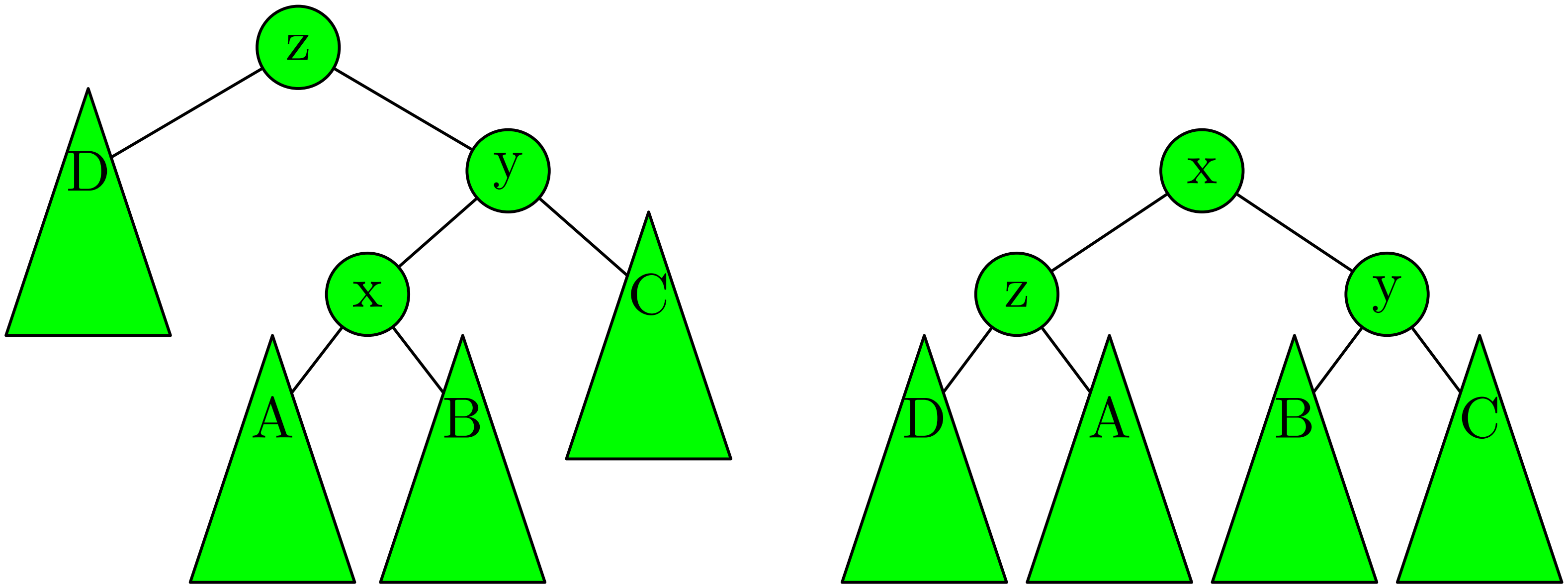


Wir nehmen erst einmal $r(z) > r(x)$ an.

- $r'(x) = r(z)$
- $r'(y) \leq r'(x) = r(z)$
- $r'(z) \leq r'(x) = r(z)$

- $r(y) \geq r(x)$
- $1 \leq r(z) - r(x)$

$$1+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z) \leq 1+r'(y)+r'(z)-r(x)-r(y)$$
$$\leq r(z) - r(x) + r(z) + r(z) - r(x) - r(x) = 3(r(z) - r(x)).$$



Jetzt nehmen wir $r(z) = r(x)$ an.

- $r'(x) = r(z)$
- $r'(y) < r(x)$ oder $r'(z) < r(x)$ (sonst $r'(x) > r(z)$)

$$
\begin{aligned}
1 &+ r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&\leq 1 + r'(y) + r'(z) - r(x) - r(y) \leq 0 = 3(r(z) - r(x))
\end{aligned}
$$

(Zig-zig ähnlich)

Splay-Bäume – Analyse

Theorem

Die amortisierten Kosten einer Splay-Operation auf einem Knoten x sind $O\bigl(\log(\bar g(w)/\bar g(x))\bigr)$, wenn w die Wurzel des Baums ist.

Beweis.

Die amortisierten Kosten sind höchstens

$$\begin{aligned}1+3(r(v_t)-r(v_{t-1}))+3(r(v_{t-1})-r(v_{t-2}))+\\ +\ldots+3(r(v_2)-r(v_1))=1+3r(v_t)-3r(v_1),\end{aligned}$$

wobei $v_t = w$ und $v_1 = x$.

$$1+3r(w)-3r(x) = O(\log(\bar g(w))-\log(\bar g(x))) = O(\log(\bar g(w)/\bar g(x)))$$

$\square$

Splay-Bäume – Suchen

Wir suchen folgendermaßen nach Schlüssel k:

1. Normale Suche im Suchbaum
2. Endet in Knoten x mit Schlüssel k oder in x^+ oder x^-
3. Wende Splay auf den gefundenen Knoten an
4. Siehe nach, ob k in Wurzel

Amortisierte Laufzeit:

$$O\left(\log\left(\frac{(\bar{g}(w))}{\bar{g}(x)}\right)\right) \text{ erfolgreiche Suche}$$

$$O\left(\log\left(\frac{\log(\bar{g}(w))}{\min\{\bar{g}(x^+), \bar{g}(x^-)\}}\right)\right) \text{ erfolglose Suche}$$

Splay-Bäume – Einfügen

Wir fügen einen Schlüssel k mit Gewicht a ein, der noch nicht vorhanden ist:

- Normale Suche im Suchbaum
- Einfügen eines neuen Knotens als Blatt
- Splay-Operation auf diesen Knoten

Amortisierte Laufzeit:

Das Konto wird zunächst erhöht.
x sei der neu eingefügte Knoten.
Die Splay-Operation benötigt $O(\log(\bar{g}(\bar{w}))/\bar{g}(x))$.

Splay-Bäume – Löschen

Wir löschen einen Schlüssel k:

1. Suche nach dem Schlüssel k
2. Siehe nach k in der Wurzel ist
3. Splay-Operation auf dem größten Knoten im linken Unterbaum
4. Klassisches Löschen von k

Amortisierte Laufzeit:
Zuerst wie Suche:

$$O\left(\log\left(\frac{(\bar{g}(w))}{\bar{g}(x)}\right)\right) \text{ erfolgreiche Suche}$$

$$O\left(\log\left(\frac{\log(\bar{g}(w))}{\min\{\bar{g}(x^+), \bar{g}(x^-)\}}\right)\right) \text{ erfolglose Suche}$$

Dann sinkt der Kontostand, was wir aber nicht ausnutzen.

Splay-Bäume als assoziatives Array

Theorem

In einem anfänglich leeren Splay-Baum können n Operationen (Suchen, Einfügen, Löschen) in $O(n \log n)$ Schritten ausgeführt werden.

Beweis.

Wir setzen $g(x) = 1$.

Dann ist $\bar{g}(T)$ die Anzahl der Knoten im Baum.
Also ist $\bar{g}(T) \leq n$.
Die amortisierten Kosten einer Operation sind
$O(\log(\bar{g}(T)) = O(\log n)$.
(Beim Einfügen kommt zur Splay-Operation noch die Erhöhung des Kontostands um $O(\log n)$ hinzu.) □

Amortisierte Analyse – Wiederholung

Eine Datenstruktur werde durch n Operationen verändert:

$$D_0 \xrightarrow{t_1} D_1 \xrightarrow{t_2} D_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} D_n$$

Die Zeit dafür ist $\sum_{k=1}^{n} t_i$.

Klassische Analyse:

- Jede Operation benötigt höchstens $f(n)$ Schritte
- Die Gesamtzeit ist $O(f(n)n)$.

Problematisch, wenn t_i sehr schwankend.

Amortisierte Analyse – Wiederholung

$$D_0 \xrightarrow{t_1} D_1 \xrightarrow{t_2} D_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} D_n$$

Wir definieren uns eine Potentialfunktion Φ mit:

1. $\Phi(D_0) = 0$
2. $\Phi(D_i) \geq 0$
3. $a_i := t_i + \Phi(D_i) - \Phi(D_{i-1})$ für $i > 0$
4. a_i nicht sehr schwankend

Amortisierte Analyse:

- Zeige, daß $a_i \leq g(n)$
- Gesamtzeit höchstens $O(ng(n))$

$$ng(n) \geq \sum_{k=1}^{n} a_i = \sum_{k=1}^{n} t_i + \Phi(D_n) + \Phi(D_0) \geq \sum_{k=1}^{n} t_i$$

Splaytrees und Optimale Suchbäume

Theorem

Gegeben sei ein Suchbaum T für n Schlüssel. Eine bestimmte Folge von m Suchoperationen in T benötige Zeit t.
Wenn wir die n Schlüssel in einen Splay-Baum einfügen und dann dieselbe Folge von Suchoperatioen ausführen, dauert dies $O(n^2 + t)$.

Bedeutung:

Asymptotisch verhalten sich Splay-Bäume ebensogut wie optimale Suchbäume.

Der Splay-Baum benötigt aber Zeit, um die Zugriffshäufigkeiten zu „lernen“.

Beweis.

Sei $d(k)$ die Tiefe des Knotens mit Schlüssel k in T und d die Gesamttiefe von T.
Wir definieren $g(k) = 3^{d-d(k)}$ als Gewichtsfunktion.

Die amortisierten Kosten einer Suche nach k sind:

$$O\left(\log\left(\frac{(\bar{g}(w))}{\bar{g}(k)}\right)\right)$$

Es gilt $\bar{g}(w) \leq \sum_{i=1}^{n} 3^{d-d(k_i)} \leq \sum_{j=0}^{d} 2^j 3^{d-j} \leq 3^{d+1}$ und
$\bar{g}(k) \geq g(k) = 3^{d-d(k)}$.
Die Kosten sind daher höchstens $O(\log(3^{d+1}/3^{d-d(k)}) = O(d(k))$.

Die Suche in T benötigt aber $\Omega(d(k))$ Zeit.

Das Aufbauen des Splaytrees geht in $O(n^2)$. $\square$

Java

```
private void splay(Searchtreenode⟨K, D⟩ t) {
 while(t.parent ≠ null) {
   if(t.parent.parent ≡ null) {
     if(t ≡ t.parent.left) t.parent.rotateright(); //Zig
     else t.parent.rotateleft(); }//Zag
   else if(t ≡ t.parent.left && t.parent ≡ t.parent.parent.left) {
     t.parent.parent.rotateright(); //Zig − zig
     t.parent.rotateright(); }
   else if(t ≡ t.parent.left && t.parent ≡ t.parent.parent.right) {
       t.parent.rotateright(); //Zig − zag
       t.parent.rotateleft(); }
   else if(t ≡ t.parent.right && t.parent ≡ t.parent.parent.right) {
     t.parent.parent.rotateleft(); //Zag − zag
     t.parent.rotateleft(); }
   else if(t ≡ t.parent.right && t.parent ≡ t.parent.parent.left) {
       t.parent.rotateleft(); //Zag − zig
       t.parent.rotateright(); }
 }
 root = t;
}
```

Java

```
public boolean iselement(K k) {
  if(root ≡ null) return false;
  Searchtreenode⟨K, D⟩ n = root, last = root;
  int c;
  while(n ≠ null) {
    last = n;
    c = k.compareTo(n.key);
    if(c < 0) n = n.left;
    else if(c > 0) n = n.right;
    else {splay(n); return true; }
  }
  splay(last); return false;
}
```

Java

```
public void insert(K k, D d) {
  super.insert(k, d);
  iselement(k);
}
```

Java

```
public D find(K k) {
  iselement(k);
  if(root ≠ null && root.key.equals(k)) return root.data;
  return null;
}
```

Java

```
public void delete(K k) {
    if(!iselement(k)) return;
    if(root.left ≠ null) {
        Searchtreenode⟨K, D⟩ max = root.left;
        while(max.right ≠ null) max = max.right;
        splay(max);
    }
    super.delete(k);
}
```