AVL-Bäume – Analyse

Theorem

Ein AVL-Baum der Höhe h besitzt zwischen F_h und $2^h - 1$ viele Knoten.

Definition

Wir definieren die $n$te Fibonaccizahl:

$$F_n = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n > 2 \end{cases}$$

AVL-Bäume – Analyse

Theorem

Ein AVL-Baum der Höhe h besitzt zwischen F_h und $2^h - 1$ viele Knoten.

Definition

Wir definieren die $n$te Fibonaccizahl:

$$F_n = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n > 2 \end{cases}$$

Beweis.

Trivial: Ein vollständiger Binärbaum der Höhe h hat genau $2^h - 1$ viele interne Knoten (und ist ein AVL-Baum).

Ein Baum der Höhe 0 hat keinen internen Knoten und $F_0 = 0$.

Ein Baum der Höhe 1 hat genau einen internen Knoten und $F_1 = 1$.

Falls $h > 1$, dann hat der linke oder rechte Unterbaum Höhe $h - 1$ und damit mindestens F_{h-1} viele interne Knoten.

Der andere Unterbaum hat mindestens Höhe $h - 2$ und damit mindestens F_{h-2} viele interne Knoten.

Insgesamt macht das mindestens $F_{h-1} + F_{h-2} = F_h$ viele Knoten. □

Beweis.

Trivial: Ein vollständiger Binärbaum der Höhe h hat genau $2^h - 1$ viele interne Knoten (und ist ein AVL-Baum).

Ein Baum der Höhe 0 hat keinen internen Knoten und $F_0 = 0$.

Ein Baum der Höhe 1 hat genau einen internen Knoten und $F_1 = 1$.

Falls $h > 1$, dann hat der linke oder rechte Unterbaum Höhe $h - 1$ und damit mindestens F_{h-1} viele interne Knoten.

Der andere Unterbaum hat mindestens Höhe $h - 2$ und damit mindestens F_{h-2} viele interne Knoten.

Insgesamt macht das mindestens $F_{h-1} + F_{h-2} = F_h$ viele Knoten. □

Beweis.

Trivial: Ein vollständiger Binärbaum der Höhe h hat genau $2^h - 1$ viele interne Knoten (und ist ein AVL-Baum).

Ein Baum der Höhe 0 hat keinen internen Knoten und $F_0 = 0$.

Ein Baum der Höhe 1 hat genau einen internen Knoten und $F_1 = 1$.

Falls $h > 1$, dann hat der linke oder rechte Unterbaum Höhe $h - 1$ und damit mindestens F_{h-1} viele interne Knoten.

Der andere Unterbaum hat mindestens Höhe $h - 2$ und damit mindestens F_{h-2} viele interne Knoten.

Insgesamt macht das mindestens $F_{h-1} + F_{h-2} = F_h$ viele Knoten. □

AVL-Bäume – Analyse

Theorem

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

Beweis.

Es gilt $F_h = \Theta(\phi^h)$ mit $\phi = (1 + \sqrt{5})/2$.

$$F_h \leq n \Rightarrow h \log(\phi) + O(1) \leq \log n$$

$$n \leq 2^h - 1 \Rightarrow \log n \leq h + O(1)$$

□

AVL-Bäume – Analyse

Theorem

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

Beweis.

Es gilt $F_h = \Theta(\phi^h)$ mit $\phi = (1 + \sqrt{5})/2$.

$$F_h \leq n \Rightarrow h \log(\phi) + O(1) \leq \log n$$
$$n \leq 2^h - 1 \Rightarrow \log n \leq h + O(1)$$

□

AVL-Bäume – Analyse

Theorem

Einfügen, Suchen und Löschen in einen AVL-Baum mit n Elementen benötigt $O(\log n)$ Schritte.

Beweis.

Die Höhe des AVL-Baumes in $\Theta(\log n)$.
Einfügen, Suchen und Löschen benötigt $O(h)$ Schritte, wenn h die Höhe des Suchbaums ist.
Das Rebalanzieren benötigt ebenfalls $O(h)$ Schritte. □

AVL-Bäume – Analyse

Theorem

Einfügen, Suchen und Löschen in einen AVL-Baum mit n Elementen benötigt $O(\log n)$ Schritte.

Beweis.

Die Höhe des AVL-Baumes in $\Theta(\log n)$.
Einfügen, Suchen und Löschen benötigt $O(h)$ Schritte, wenn h die Höhe des Suchbaums ist.
Das Rebalanzieren benötigt ebenfalls $O(h)$ Schritte. □

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.

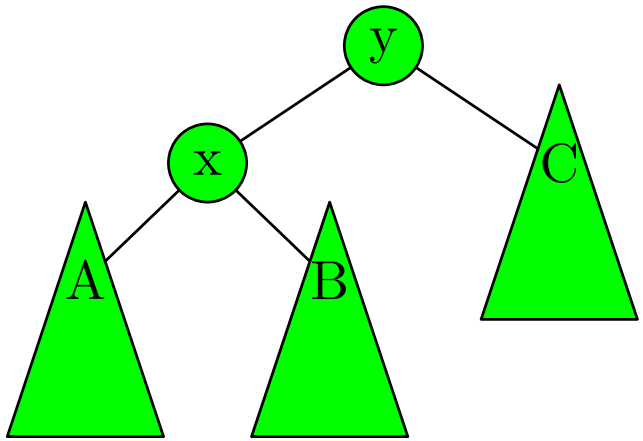
Algorithmus

```
procedure splay(node x) :
  while x ≠ 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

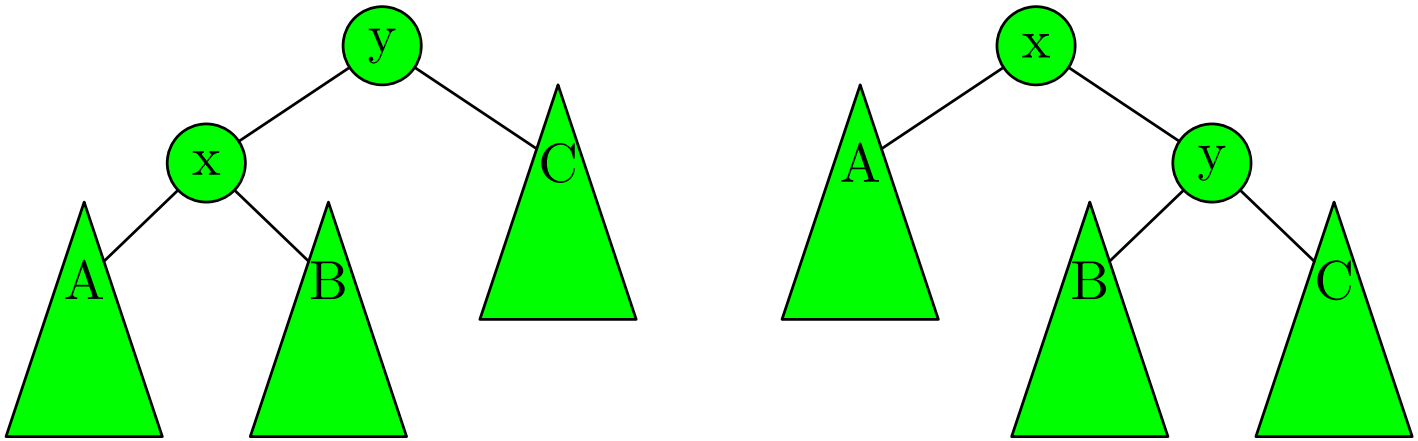


Ein zig auf x ist eine Rechtsrotation des Vaters von x.

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 und Zag

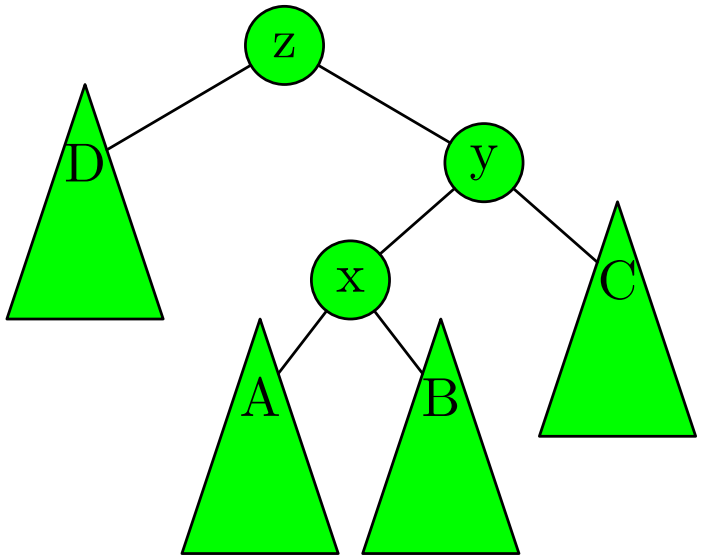


Ein zig auf x ist eine Rechtsrotation des Vaters von x.

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

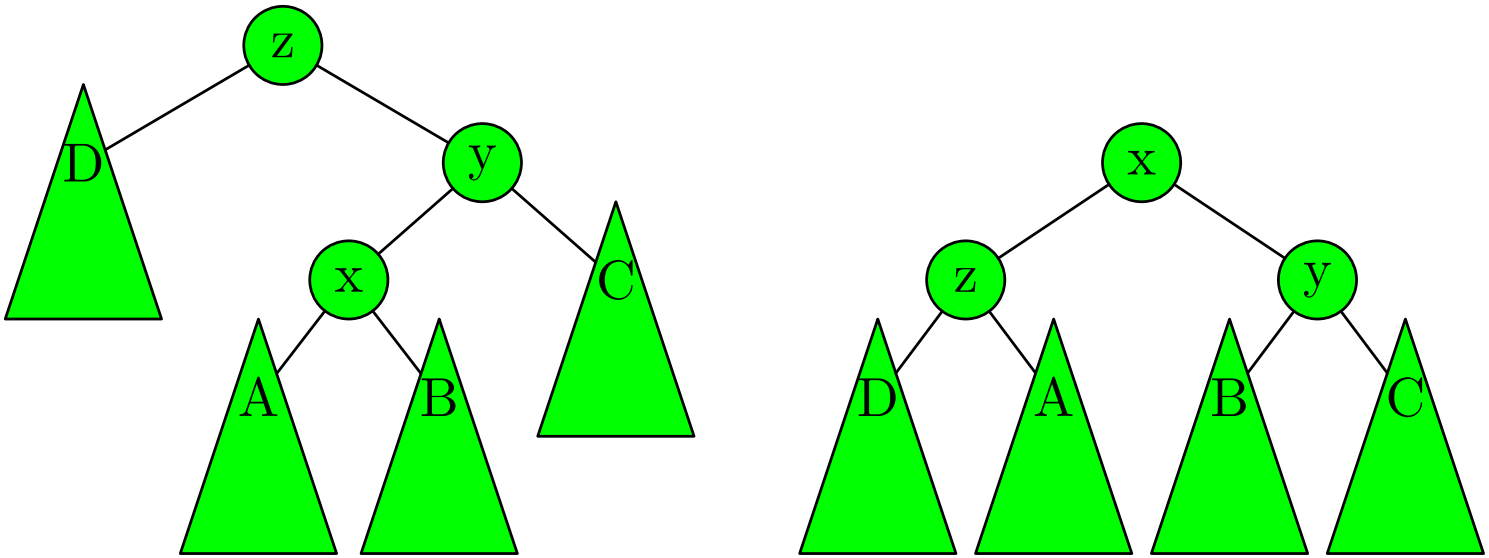


Ein Zig-zag auf x ist eine Rechtsrotation auf y gefolgt von einer Linksrotation auf z.

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

Zag-zig ist symmetrisch hierzu.

Zig-zag und Zag-zig

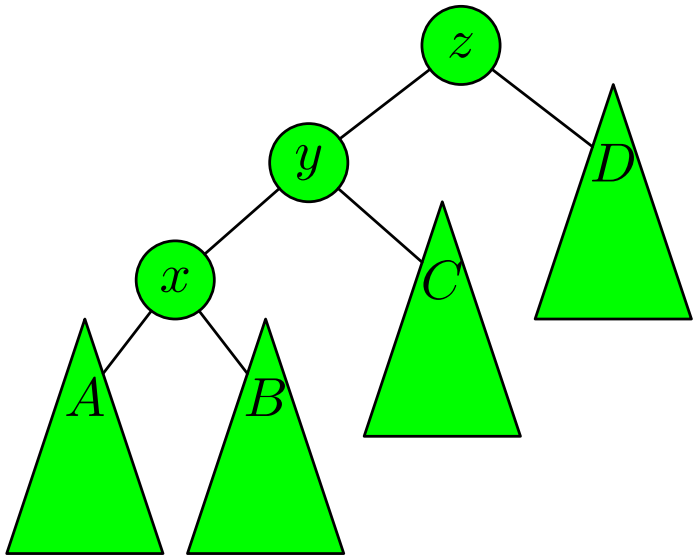


Ein Zig-zag auf x ist eine Rechtsrotation auf y gefolgt von einer Linksrotation auf z.

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



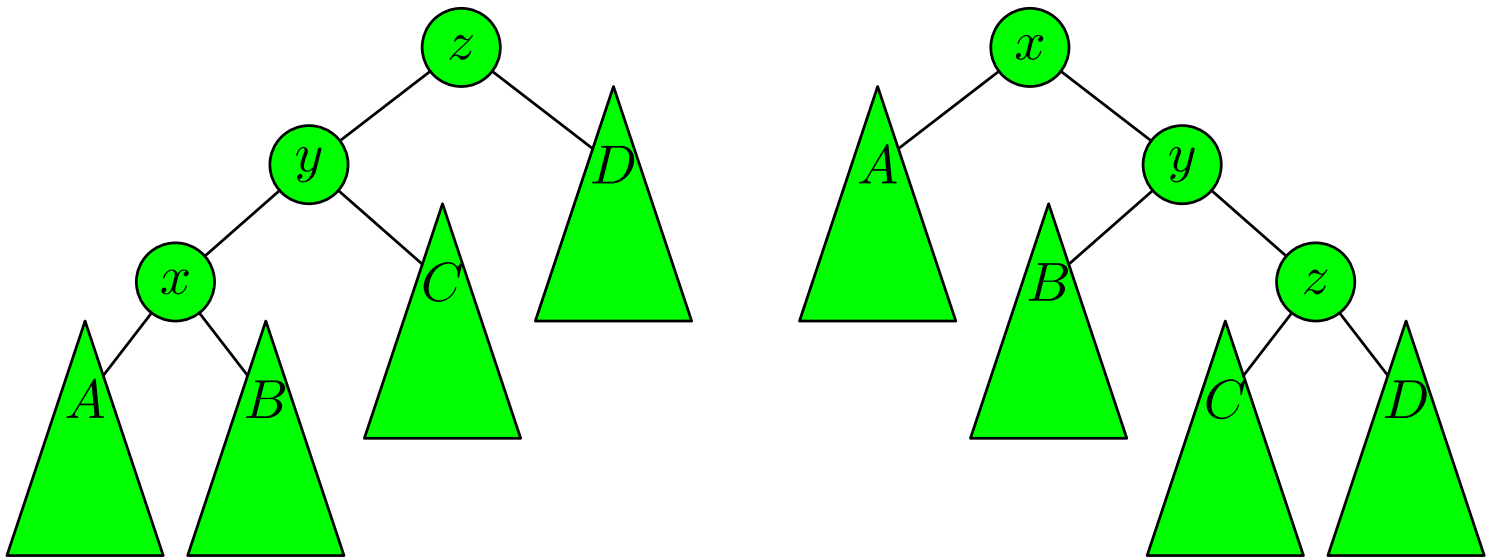
Ein Zig-zig auf x ist eine Rechtsrotation auf z gefolgt von einer Rechtsrotation auf y.

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.

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.

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 – 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 – 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 – 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.) □

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.) □

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.) □

Splay-Bäume – Beispiel

Die Schlüssel von 1 bis 40 werden zufällig eingefügt und gelöscht.

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 containsKey(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);
  containsKey(k);
}
```

Java

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

Java

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