Algorithmen und Datenstrukturen
Kapitel 7
Dynamische Mengen, das Suchproblem & Binäre Suchbäume

Frank Heitmann
heitmann@informatik.uni-hamburg.de

25. November 2015

Dynamische Mengen

Definition

Bisher hatten wir oft *statische Mengen*, bei denen sich die Anzahl der Elemente nicht ändert (z.B. beim Sortieren).
Oft soll eine Menge aber zeitabhängig wachsen und schrumpfen können. Man spricht dann von *dynamischen Mengen*.
Kennengelernt haben wir bereits den *Stack*, die *Queue* und die *verkettete Liste*.

Anmerkung

Wir beschränken uns oft auf die Darstellung der für unsere Operationen wichtigen Daten. Meist enthalten die Objekte aber zusätzliche *Satellitendaten* z.B. einen *Wächter* am Ende einer verketteten Liste.

Dynamische Mengen

Wichtige Operationen (dynamischer Mengen):

- Search(S, k). Gibt es einen Zeiger x auf ein Element in S mit schlüssel$[x] = k$?
- Insert(S, x).
- Delete(S, x).
- Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x).

Viele hiervon haben wir schon bei z.B. der verketteten Liste gesehen, aber wie schnell waren die?

Das Suchproblem

Definition (Das Suchproblem)

Eingabe: Eine Sequenz $< a_1, a_2, \ldots, a_n >$ von n Zahlen und ein Wert k.
Gesucht: Ein Index i mit $k = A[i]$, falls k in A vorkommt oder der Wert `nil`.

Definition (Das Suchproblem (allgemein))

Eingabe: Eine Menge M und ein Wert k.
Gesucht: Ein Zeiger x auf ein Element in M mit schlüssel$[x] = k$ oder `nil`, falls kein solches Element existiert.

Brute-Force-Suche (Array)

Algorithmus 1 Search($A[1..n], k$)

```
1: for i = 1 to n do
2:     if a[i] == k then
3:         return i
4:     end if
5: end for
6: return nil
```

Analyse

Laufzeit ist in $\Theta(n)$. Korrektheit folgt mit Schleifeninvariante (oder hier auch direkt).

Brute-Force-Suche (Liste)

Algorithmus 2 Search(L, k)

```
1: x = kopf[L]
2: while x ≠ nil && schlüssel[x] ≠ k do
3:     x = nachfolger[x]
4: end while
5: return x
```

Analyse

Laufzeit ist wieder in $\Theta(n)$. Korrektheit zeigt man wie oben.

Binäre Suche

Algorithmus 3 Binäre Suche (Array)

```
1: while first ≤ last ∧ idx < 0 do
2:     m = first + ((last − first)/2)
3:     if a[m] < p then
4:         first = m + 1
5:     else if a[m] > p then
6:         last = m − 1
7:     else
8:         idx = m
9:     end if
10: end while
11: return idx
```

Analyse

Laufzeit ist in $\Theta(\log n)$ ($T(n) = T(n/2) + c$).

Grundlegende Algorithmen - Zusammenfassung

Zusammenfassung

- Die Laufzeit der Brute-Force-Suchalgorithmen war in $\Theta(n)$, insb. da jedes Element betrachtet werden musste.
- Die binäre Suche erlaubt eine Laufzeit in $\Theta(\log n)$, ist aber nicht bei jeder Datenstruktur anwendbar!
- Wir suchen nun Datenstrukturen, die die Suche und auch die anderen Operationen für *dynamische* Mengen in wenig Zeit (z.B. in $O(\log n)$) erlauben.

Pause to Ponder

Unsere bisherigen Datenstrukturen erlauben dies nicht. Man überlege sich dies an Array, Stack, Queue, Liste und Heap...

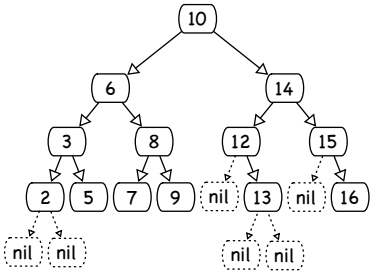
Binäre Suchbäume - Die Idee

Ein *binärer Suchbaum* ist eine Datenstruktur, die viele Operationen auf dynamischen Mengen ermöglicht, insb. die Operationen Search, Minimum, Maximum, Predecessor, Successor, Insert und Delete. Ein binärer Suchbaum ist ein binärer Baum (jeder Knoten hat bis zu zwei Kindern), wobei der Baum noch die spezielle Suchbaum-Eigenschaft erfüllt.

- Der Baum wird (anders als der Heap!) meist durch eine verkettete Datenstruktur (ähnlich der doppelt-verketteten Liste) repräsentiert. Jeder Knoten ist ein Objekt mit den Attributen
  - `schlüssel` (und den Satellitendaten),
  - `links`, `rechts`, `p`, die auf den jeweiligen linken bzw. rechten Nachfolger, bzw. den Vaterknoten zeigen (fehlen diese, sind die entsprechenden Attribute auf `nil` gesetzt).

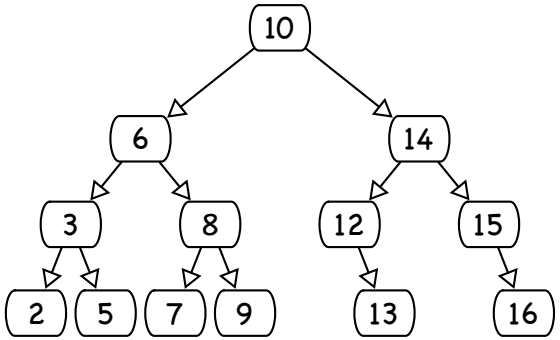
Binäre Suchbäume - Die Idee

- *Binäre Suchbaum-Eigenschaft*: Sei p ein Knoten im binären Suchbaum. Sei l ein Knoten im linken Teilbaum von n, r im rechten, dann gilt stets: $\texttt{schlüssel}[l] \leq \texttt{schlüssel}[p]$ und $\texttt{schlüssel}[p] \leq \texttt{schlüssel}[r]$.

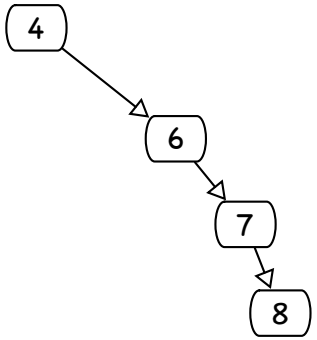


(Im Bild sind nicht alle *nil*-Knoten eingezeichnet, nachfolgend verzichten wir meist ganz auf sie.)

Binäre Suchbäume - Beispiel (gut)



Binäre Suchbäume - Beispiel (schlecht)



Die Operationen

Wir werden nachfolgenden folgende Operationen betrachten:

- InorderTreeWalk(x).
- TreeSearch(x, k), IterativeTreeSearch(x, k).
- TreeMinimum(x), TreeMaximum(x).
- TreeSuccessor(x), TreePredecessor(x).
- TreeInsert(T, x), TreeDelete(T, z).

InorderTreeWalk - Die Idee

Gibt die Schlüssel aus, wobei

- zuerst die Schlüssel des linken,
- dann der Schlüssel der Wurzel und
- dann die Schlüssel des rechten Teilbaumes

ausgegeben werden.

InorderTreeWalk

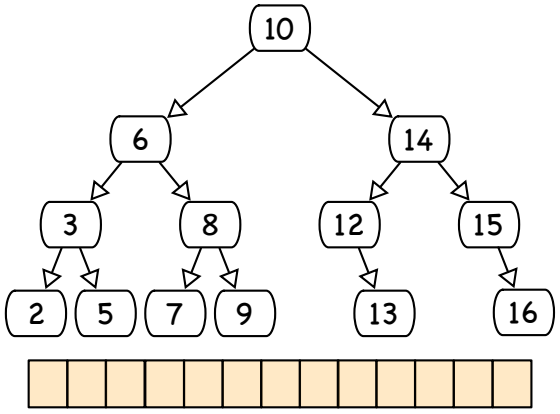
Algorithmus 4 InorderTreeWalk(x)

```
1: if x ≠ nil then
2:     InorderTreeWalk(links[x])
3:     print schlüssel[x]
4:     InorderTreeWalk(rechts[x])
5: end if
```

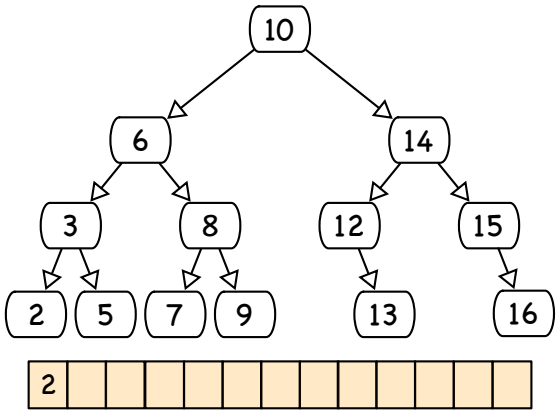
Anmerkung

Ähnlich kann man auch PreorderTreeWalk (Wurzel zuerst) und PostorderTreeWalk (Wurzel zuletzt) schreiben.

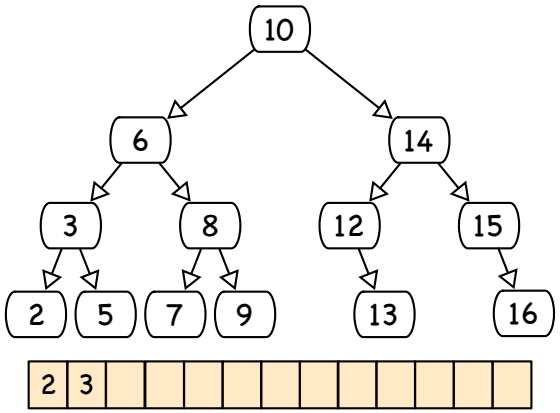
InorderTreeWalk - Beispiel



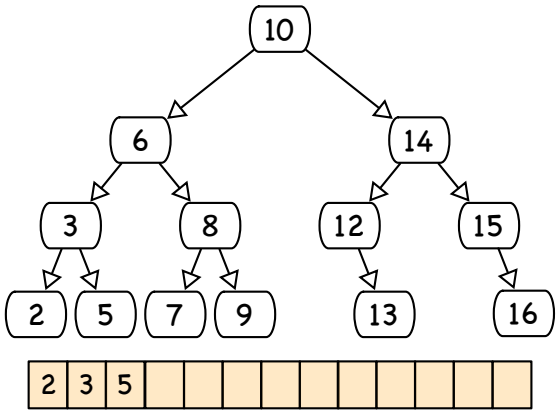
InorderTreeWalk - Beispiel



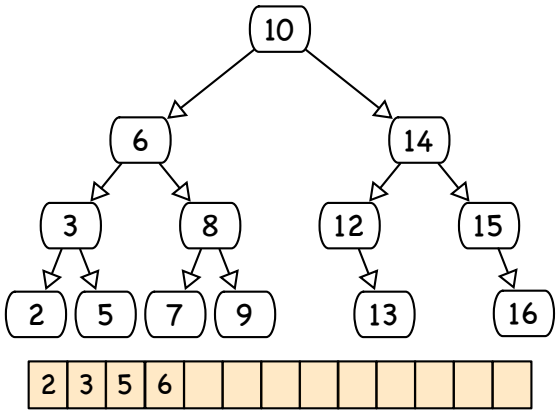
InorderTreeWalk - Beispiel



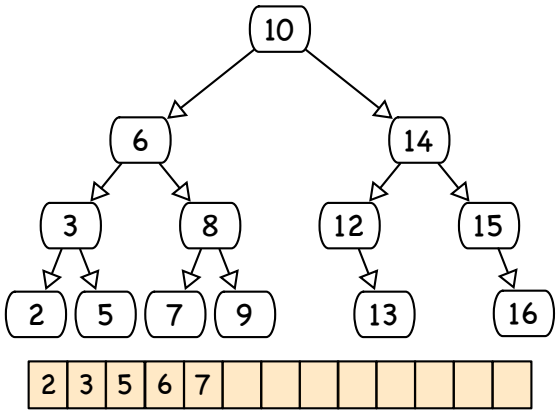
InorderTreeWalk - Beispiel



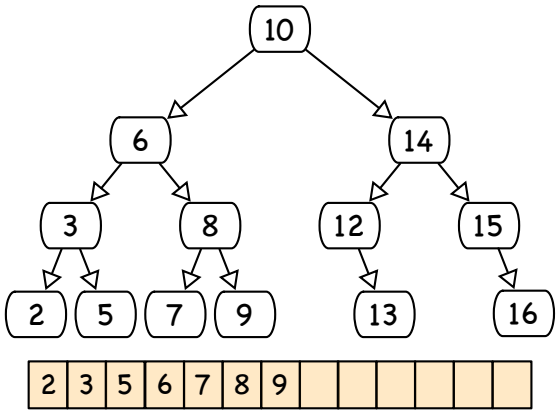
InorderTreeWalk - Beispiel



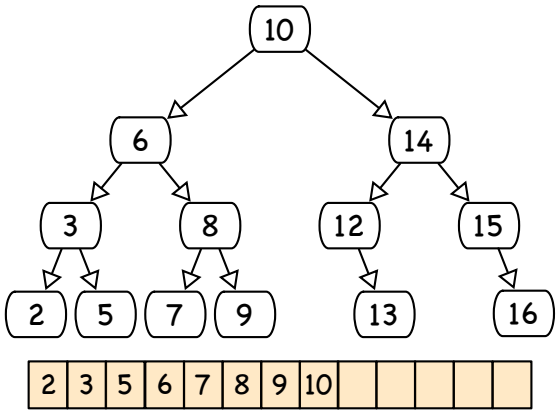
InorderTreeWalk - Beispiel



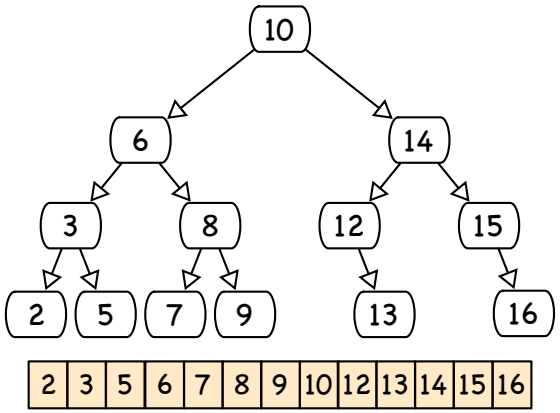
InorderTreeWalk - Beispiel



InorderTreeWalk - Beispiel



InorderTreeWalk - Beispiel



InorderTreeWalk

Anmerkung

Bei einem binären Suchbaum werden die Schlüssel bei InorderTreeWalk *in sortierter Reihenfolge* ausgegeben! (Der Satz, dass dies korrekt ist, folgt gleich. Der Beweis mündlich.)

Pause to Ponder

Laufzeit?

InorderTreeWalk

Algorithmus 5 InorderTreeWalk(x)

```
1: if x ≠ nil then
2:     InorderTreeWalk(links[x])
3:     print schlüssel[x]
4:     InorderTreeWalk(rechts[x])
5: end if
```

Satz

Wenn x die Wurzel eines Teilbaums mit n Knoten ist, dann ist `InorderTreeWalk`(x) *korrekt (folgt per Induktion aus den Eigenschaften binärer Suchbäume).*

InorderTreeWalk - Laufzeitanalyse (1/4)

Satz

Wenn x die Wurzel eines Teilbaums mit n Knoten ist, dann ist `InorderTreeWalk`(x) *in Zeit* $\Theta(n)$.

Beweis.

Aufstellen der Rekurrenzgleichung: Beim einem leeren (Teil-)Baum gilt $T(0) = c$. Ferner gilt bei Aufruf auf einen Knoten dessen linker Teilbaum k und dessen rechter Teilbaum $n-k-1$ Knoten hat $T(n) = T(k) + T(n-k-1) + d$.
Wir vermuten nun $T(n) \in \Theta(n)$, d.h. wir vermuten $T(n) = e \cdot n$ für eine Konstante e. □

InorderTreeWalk - Laufzeitanalyse (2/4)

Beweis.

Behauptung: $T(n) = e \cdot n$.
Wollen wir dies mit Induktion beweisen, so merken wir gleich beim Induktionsanfang $T(0) = e \cdot 0 = 0 \neq c$. Wir müssen unsere Vermutung also verbessern!
Neue Vermutung: $T(n) = e \cdot n + c$. Damit ist immer noch $T(n) \in \Theta(n)$, aber der Induktionsanfang klappt jetzt! □

InorderTreeWalk - Laufzeitanalyse (3/4)

Beweis.

Behauptung: $T(n) = e \cdot n + c$.
Induktionsschritt $(n-1 \Rightarrow n)$:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &= (e \cdot k + c) + (e \cdot (n-k-1) + c) + d \\ &= e \cdot n - e + 2c + d \end{aligned}$$

Setzen wir $e = c + d$ (denn wir wollen ja $T(n) = e \cdot n + c$ erreichen (nicht $T(n+1) = ...$, da wir den Induktionsschritt ja auf n machen)) und verfeinern auf diese Weise also unsere Behauptung erneut (!), so wird gleich alles klappen! □

InorderTreeWalk - Laufzeitanalyse

Anmerkung

Nimmt man als Behauptung gleich $T(n) = (c+d)n+c$, so klappt alles sofort, aber die Gleichung fällt dann 'vom Himmel'. Oben kommt man auf $e = c + d$, weil man ja möchte, dass sich $T(n) = e \cdot n + c$ ergibt.

InorderTreeWalk - Laufzeitanalyse (4/4)

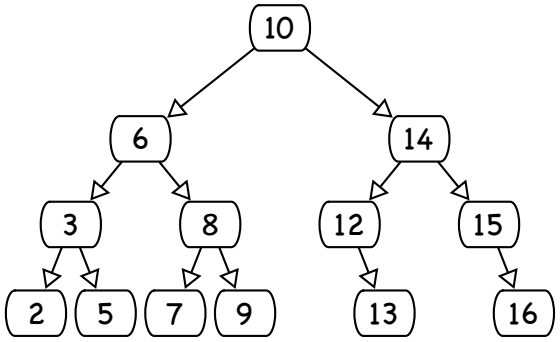
Beweis.

Gegeben (aus dem Code): $T(n)$ mit $T(0) = c$ und $T(n) = T(k) + T(n-k-1) + d$.
Behauptung: $T(n) = (c+d) \cdot n + c$.
Induktionsanfang: $T(0) = c$.
Induktionsannahme: Behauptung gelte schon für Werte kleiner n.
Induktionsschritt $(n-1 \Rightarrow n)$:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &= ((c+d) \cdot k + c) + ((c+d) \cdot (n-k-1) + c) + d \\ &= (c+d) \cdot n - (c+d) + 2c + d \\ &= (c+d) \cdot n + c \end{aligned}$$

Also gilt $T(n) \in \Theta(n)$. □

TreeSearch - Die Idee am Beispiel



TreeSearch 7? 15? 11?

TreeSearch - Die Idee

- Beginne mit der Wurzel. Ist diese kleiner als der Schlüssel, so suche im rechten Teilbaum, ist sie grösser, suche im linken.
- Fahre wie oben mit der Wurzel diese Teilbaumes fort.
- Breche ab, wenn Schlüssel gefunden oder, wenn `nil`-'Knoten' erreicht wird.

TreeSearch - Pseudocode

Algorithmus 6 TreeSearch(x, k)

1: **if** $x ==$ `nil` or $k ==$ schlüssel[x] **then**
2: **return** x
3: **else if** $k <$ schlüssel[x] **then**
4: **return** `TreeSearch`(links[x], k)
5: **else**
6: **return** `TreeSearch`(rechts[x], k)
7: **end if**

Analyse

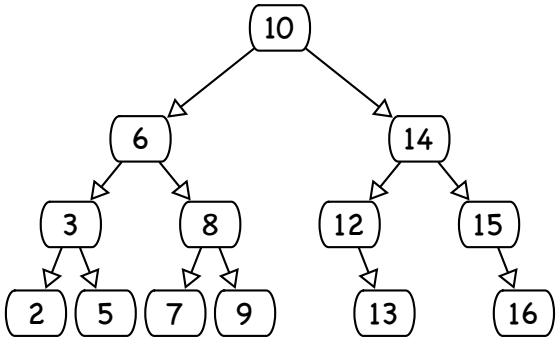
Die Knoten, die während der Rekursion besucht werden, bilden einen Pfad von der Wurzel bis maximal zu einem Blatt. Die Korrektheit folgt aus den Eigenschaften binärer Suchbäume. Die Laufzeit ist in $O(h)$, wobei h die Höhe des Baumes ist.

TreeSearch - Pseudocode (iterativ)

Algorithmus 7 IterativeTreeSearch(x, k)

1: **while** $x \neq$ `nil` and $k \neq$ schlüssel[x] **do**
2: **if** $k <$ schlüssel[x] **then**
3: $x =$ links[x]
4: **else**
5: $x =$ rechts[x]
6: **end if**
7: **end while**
8: **return** x

TreeSearch, Minimum und Maximum am Beispiel



TreeSearch 5? Minimum? Maximum?

Minimum und Maximum

Algorithmus 8 TreeMinimum(x)

1: **while** links[x] $\neq$ `nil` **do**
2: $x =$ links[x]
3: **end while**
4: **return** x

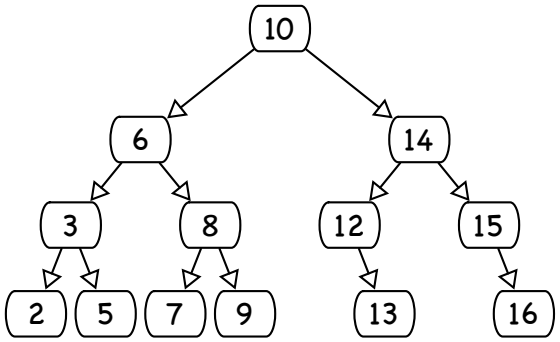
Algorithmus 9 TreeMaximum(x)

1: **while** rechts[x] $\neq$ `nil` **do**
2: $x =$ rechts[x]
3: **end while**
4: **return** x

Analyse (Korrektheit und Laufzeit) wie bei TreeSearch.

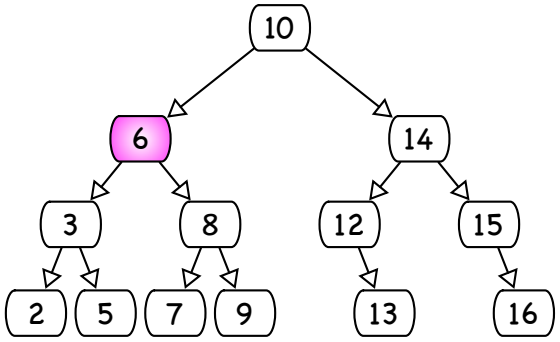
Vorgänger und Nachfolger

Wir wollen den Nachfolger eines Knotens in *der sortierten Reihenfolge* ausgeben. (Den Vorgänger ermittelt man analog.)

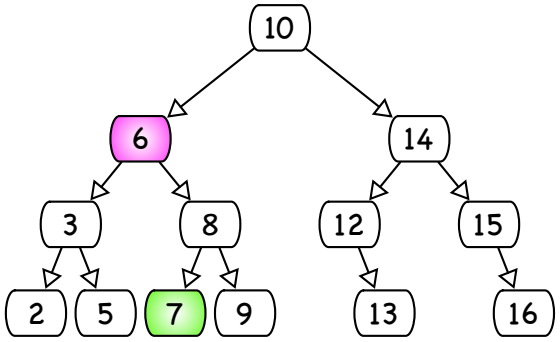


Wir können dies machen ohne jemals Schlüssel zu vergleichen!

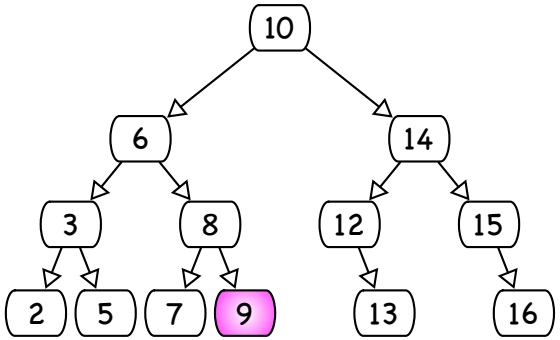
Nachfolger - Beispiel



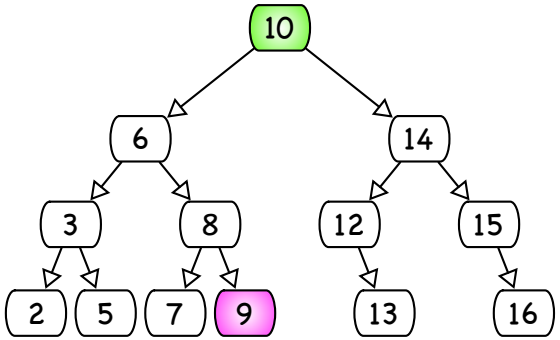
Nachfolger - Beispiel



Nachfolger - Beispiel



Nachfolger - Beispiel



Nachfolger - Die Idee

Der Nachfolger eines Knotens x ist der Knoten mit dem *kleinsten* Schlüssel, der größer ist als schlüssel[x].

- Größer als schlüssel[x] heisst im rechten Teilbaum von x
- und dort am kleinsten heisst das Minimum ('ganz links').

$\Rightarrow$ Problem: x muss gar keinen rechten Teilbaum haben!

- Dann ist der Nachfolger y der jüngste Vorfahre von x, dessen linkes Kind ebenfalls ein Vorfahre von x ist.
- (Geht das auch nicht, dann ist x das größte Element und hat keinen Nachfolger.)

Nachfolger - Die Idee

Dann ist der Nachfolger y der jüngste Vorfahre von x, dessen linkes Kind ebenfalls ein Vorfahre von x ist.

Das heißt wir gehen von x aus nach oben und der *erste* Knoten y auf den wir treffen von dem aus gesehen x in seinem *linken* Teilbaum ist, ist der gesuchte. Dann gilt nämlich gerade schlüssel[x] $<$ schlüssel[y] (wegen '*linken*' oben) und es gibt auch keinen Schlüssel dazwischen (wegen '*erste*' oben).

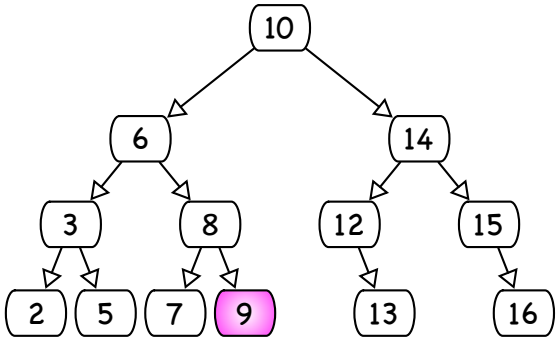
(Obiges kann in einen Beweis um-/ausformuliert werden.)

Bemerkung

Umgangssprachlich: Wir gehen im Baum immer weiter nach oben, wenn wir das erste Mal nach rechts gehen, haben wir den gesuchten Knoten.

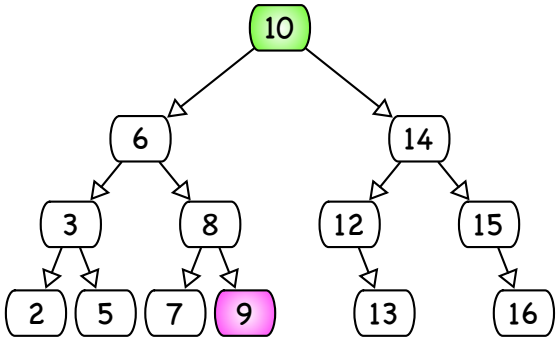
Nachfolger - Die Idee

Dann ist der Nachfolger y der jüngste Vorfahre von x, dessen linkes Kind ebenfalls ein Vorfahre von x ist.



Nachfolger - Die Idee

Dann ist der Nachfolger y der jüngste Vorfahre von x, dessen linkes Kind ebenfalls ein Vorfahre von x ist.



Nachfolger - Pseudocode

Algorithmus 10 TreeSucessor(x)

```
1: if rechts[x] ≠ nil then
2:     return TreeMinimum(rechts[x])
3: end if
4: y = p[x]
5: while y ≠ nil and x == rechts[y] do
6:     x = y
7:     y = p[y]
8: end while
9: return y
```

Analyse

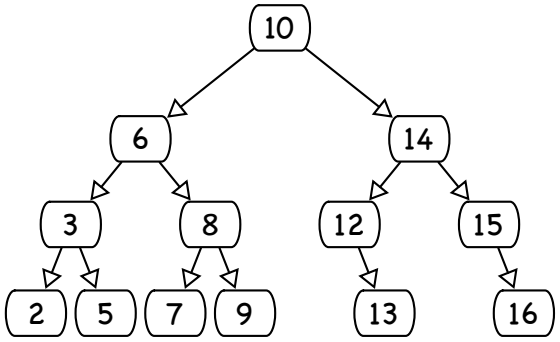
Korrektheit folgt aus obigen Überlegungen. Laufzeit ist wieder in $O(h)$. Die Routine TreePredecessor implementiert man analog.

Einfügen - Die Idee

Wir **manipulieren** nun die (*dynamische*!) Datenstruktur, indem wir ein Element *hinzufügen*. Dabei soll die Suchbaum-Eigenschaft erhalten bleiben.

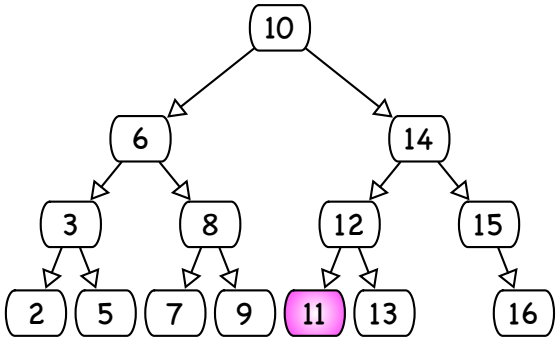
Im Prinzip suchen wir (ähnlich der Suche) den richtigen Platz und fügen das neue Element einfach als Blatt an...

Einfügen - Beispiel

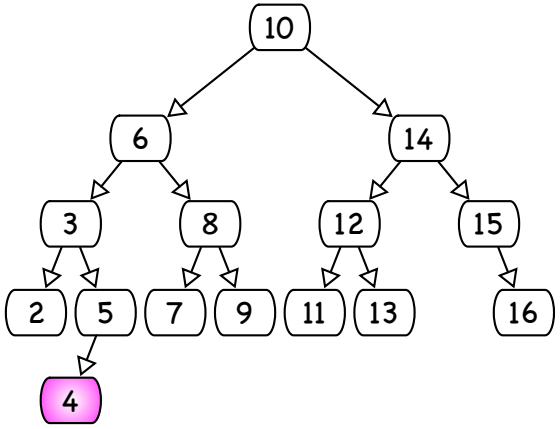


Insert 11? Insert 4?

Einfügen - Beispiel



Einfügen - Beispiel



Einfügen - Die Idee

Im Code halten wir nun einen Zeiger auf x (hier soll das neue Element hin) und einen Zeiger y auf $p[x]$. An y wird dann das neue Element angehängt...

Einfügen - Pseudocode Teil 1

Algorithmus 11 TreeInsert(T, z) - Teil 1

```
1:  y = nil
2:  x = wurzel[T]
3:  while x ≠ nil do
4:      y = x
5:      if schlüssel[z] < schlüssel[x] then
6:          x = links[x]
7:      else
8:          x = rechts[x]
9:      end if
10: end while
```

Erklärung

x zeigt jetzt auf `nil`. y auf den zuletzt besuchten Knoten an den jetzt der neue Knoten z angefügt werden soll. Oft ist y ein Blatt.

Einfügen - Pseudocode Teil 2

Algorithmus 12 TreeInsert(T, z) - Teil 2

```
1:  p[z] = y
2:  if y == nil then
3:      wurzel[T] = z
4:  else
5:      if schlüssel[z] < schlüssel[y] then
6:          links[y] = z
7:      else
8:          rechts[y] = z
9:      end if
10: end if
```

Erklärung

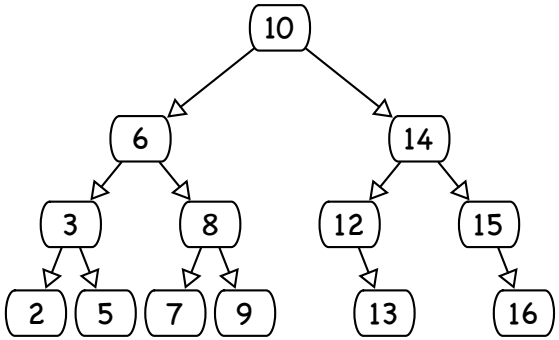
Der Vater von z ist y und z ist linkes oder rechtes Kind von y (je nach Schlüssel). links[z] und rechts[z] sind ferner `nil`.

Löschen - Die Idee

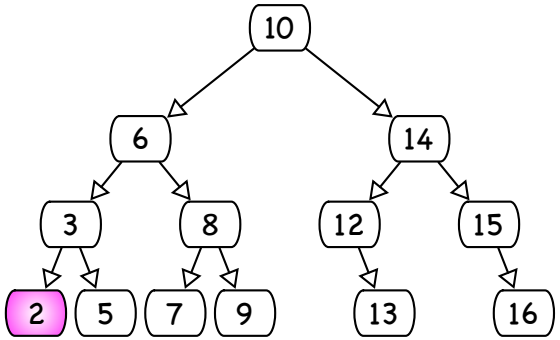
Das Löschen eines Knotens z ist komplizierter. Es gibt drei Fälle:

1. z hat keine Kinder. Dann kann er einfach gelöscht werden. ($p[z]$ zeigt auf `nil` statt auf z).
2. z hat genau ein Kind. Dann werden $p[z]$ und left[z] bzw. right[z] verbunden und damit z 'ausgeschnitten'. (Man vergleiche dies mit der Lösch-Operation bei verketteten Listen!)
3. z hat zwei Kinder. Dann wird z's Nachfolger y genommen (hat kein linkes Kind!), ausgeschnitten (siehe 2. oben) und mit seinen Daten die Daten von z ersetzt.

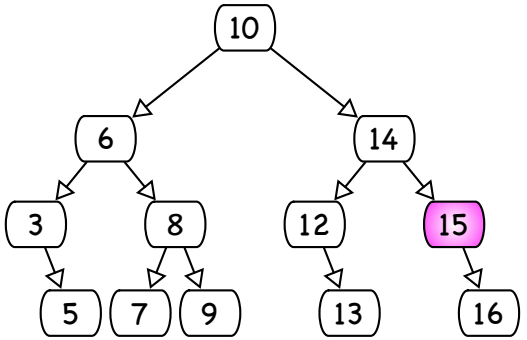
Löschen - Beispiel



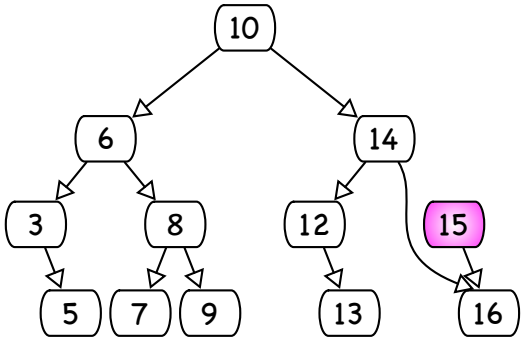
Löschen - Beispiel



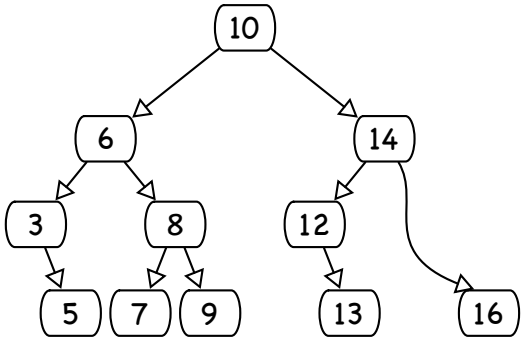
Löschen - Beispiel



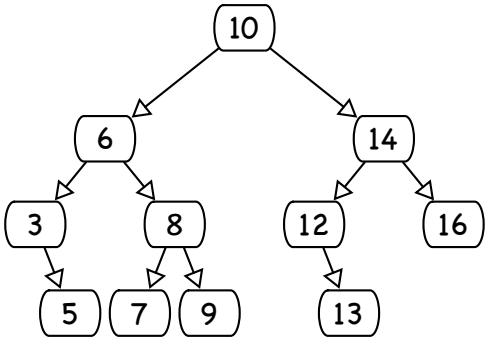
Löschen - Beispiel



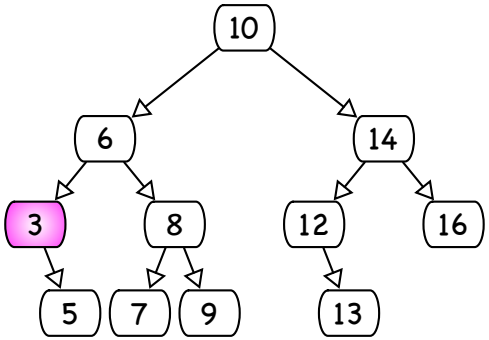
Löschen - Beispiel



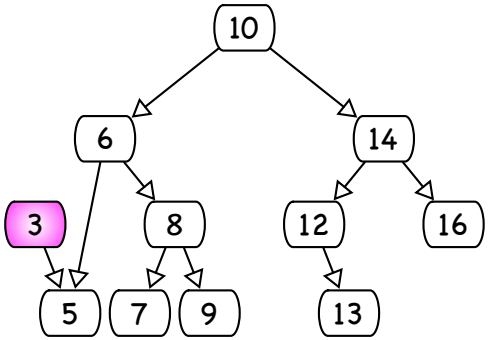
Löschen - Beispiel



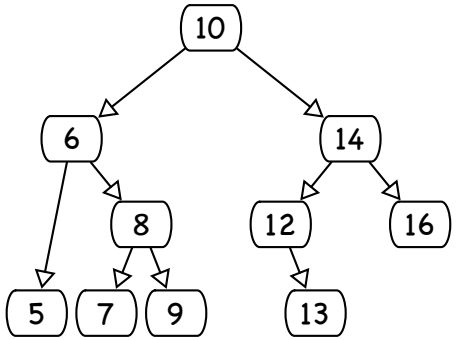
Löschen - Beispiel



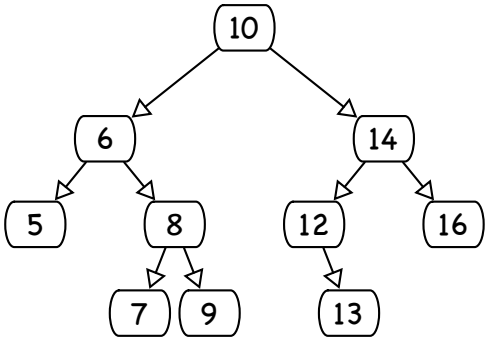
Löschen - Beispiel



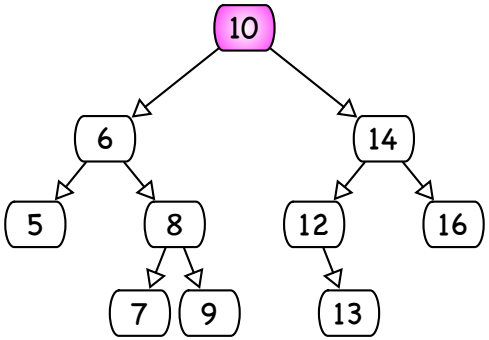
Löschen - Beispiel



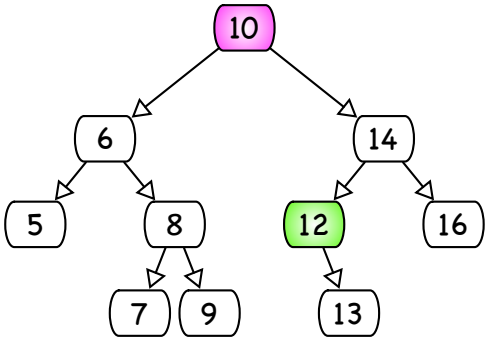
Löschen - Beispiel



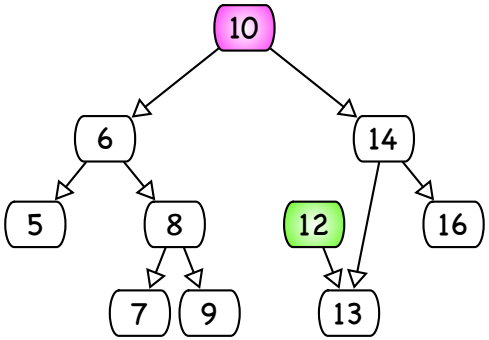
Löschen - Beispiel



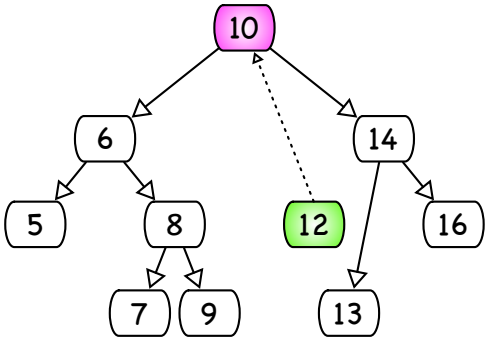
Löschen - Beispiel



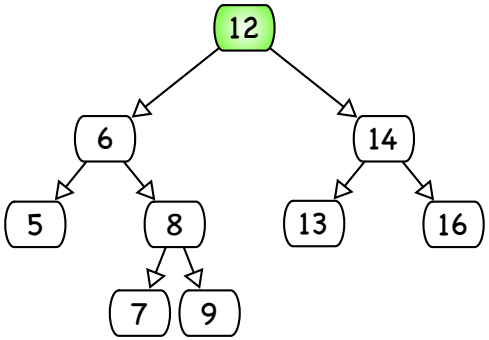
Löschen - Beispiel



Löschen - Beispiel



Löschen - Beispiel



Löschen - Pseudocode

Auf den Sourcecode verzichten wir hier. Ähnlich wie beim Einfügen müssen die Zeigern 'umgebogen' werden.

Weitere Details findet man im [Cormen] in Kapitel 12.3.

Operationen - Zusammenfassung (1/4)

Zusammenfassung der Operationen

- InorderTreeWalk $\approx$ Rekursiv erst die Elemente des linken Teilbaumes ausgeben, dann das Element selbst, dann die Elemente des rechten Teilbaumes. Gibt bei einem binären Suchbaum die Elemente in sortierter Reihenfolge aus.
- TreeSearch $\approx$ Suche im linken Teilbaum weiter, wenn zu suchendes Element kleiner ist als das Wurzelelement, sonst im rechten Teilbaum.

Operationen - Zusammenfassung (2/4)

Zusammenfassung der Operationen

- Minimum $\approx$ Element ganz links.
- Maximum $\approx$ Element ganz rechts.
- Nachfolger (von x) $\approx$ der Knoten mit dem *kleinsten* Schlüssel, der größer ist als schlüssel[x]. Minimum im rechten Teilbaum oder der jüngste Vorfahre von x, dessen linkes Kind ebenfalls ein Vorfahre von x ist.
- Vorgänger (von x) $\approx$ analog ...

Operationen - Zusammenfassung (3/4)

Zusammenfassung der Operationen

- Einfügen $\approx$ Ähnlich wie beim Suchen die geeignete Position suchen und das neue Element als neues Blatt anfügen.
- Löschen $\approx$ Drei Fälle:
  1. z hat keine Kinder. Dann kann er einfach gelöscht werden. ($p[z]$ zeigt auf `nil` statt auf z).
  2. z hat genau ein Kind. Dann werden $p[z]$ und left[z] bzw. right[z] verbunden und damit z 'ausgeschnitten'.
  3. z hat zwei Kinder. Dann wird z's Nachfolger y genommen (hat kein linkes Kind!), ausgeschnitten (siehe 2. oben) und mit seinen Daten die Daten von z ersetzt.

Operationen - Zusammenfassung (4/4)

Zusammenfassung der Operationen

Die Operationen InorderTreeWalk, TreeSearch, Minimum, Maximum, Successor, Insert und Delete sind alle *korrekt*, was man stets recht einfach über die Eigenschaften binärer Suchbäume zeigen kann (ggf. mit Induktion über die Anzahl der Knoten oder die Tiefe des Baumes).
Bei einem binären Suchbaum mit n Knoten der Höhe h, ist die Laufzeit von InorderTreeWalk in $O(n)$, die aller anderen Operationen in $O(h)$.

Anmerkung

Diese Laufzeiten sind zwar gut, da bei einem (ungefähr) ausgeglichenen Baum $h \approx \log n$ ist. Das Problem ist aber, dass binäre Suchbäume nicht ausgeglichen sein müssen! Im worst-case sind obige Operationen alle in $\Theta(n)$!

Rot-Schwarz-Bäume - Die Idee

Rot-Schwarz-Bäume sind binäre Suchbäume, bei denen jeder Knoten eine 'Farbe' hat (rot oder schwarz). Durch Einschränkungen (Rot-Schwarz-Eigenschaften) wird sichergestellt, dass der Baum annähernd balanciert ist, so dass die Operationen dann auch im worst-case in $O(\log n)$ laufen.

Das Problem ist es dann die Operationen wie insb. Einfügen und Löschen so zu implementieren, dass die Rot-Schwarz-Eigenschaften erhalten bleiben (und sie dann trotzdem in $O(\log n)$ sind)!

Rot-Schwarz-Eigenschaften

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ist schwarz.
4. *Wenn ein Knoten rot ist, dann sind seine beiden Kinder schwarz.*
5. *Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten*

Satz

Ein Rot-Schwarz-Baum mit n inneren Knoten hat höchstens die Höhe $2 \cdot \log(n+1)$.

AVL-Bäume - Die Idee

Ein *AVL-Baum* ist ein binärer Suchbaum, der *höhenbalanciert* ist, d.h. für jeden Knoten x unterscheiden sich die Höhen des linken und rechten Teilbaumes maximal um 1. Ist also $h[links[x]]$ die Höhe des linken und $h[rechts[x]]$ die Höhe des rechten Teilbaumes, dann gilt bei einem AVL-Baum für jeden Knoten x

$$h[links[x]] - h[rechts[x]] \in \{-1, 0, 1\}$$

Bei AVL-Bäumen spielt wie auch bei Rot-Schwarz-Bäumen die Rotation eine wichtige Rolle. Mit ihr wird der Baum nach einer Einfügeoperation stets wieder ausbalanciert.

Zusammenfassung

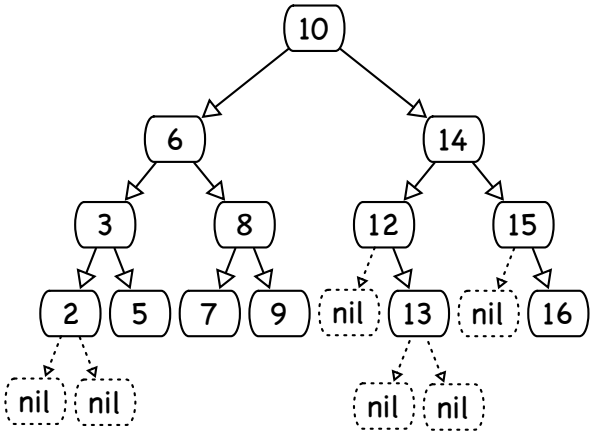
- Grundlagen des Suchens:
  - Brute-Force-Algorithmus
  - Binäres Suchen (Array muss sortiert sein!)
- Suchbäume
  - Definition
  - TreeWalk, Suchen, Minimum, Maximum
  - Vorgänger, Nachfolger
  - Einfügen, Entfernen
- Rot-Schwarz-Bäume
- AVL-Bäume

Anhang

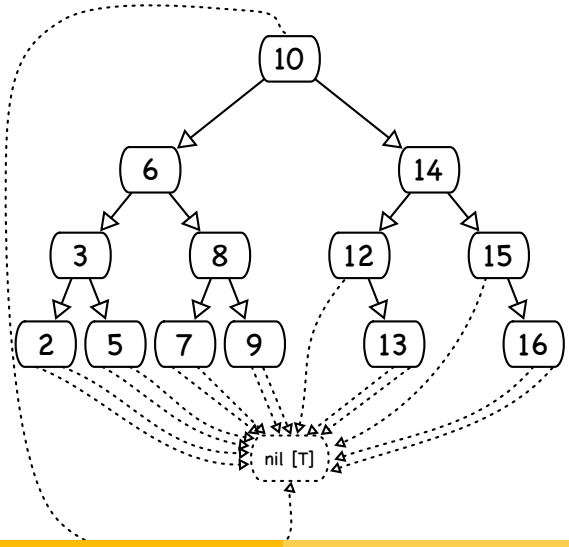
Rot-Schwarz-Eigenschaften

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ist schwarz.
4. *Wenn ein Knoten rot ist, dann sind seine beiden Kinder schwarz.*
5. *Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten*

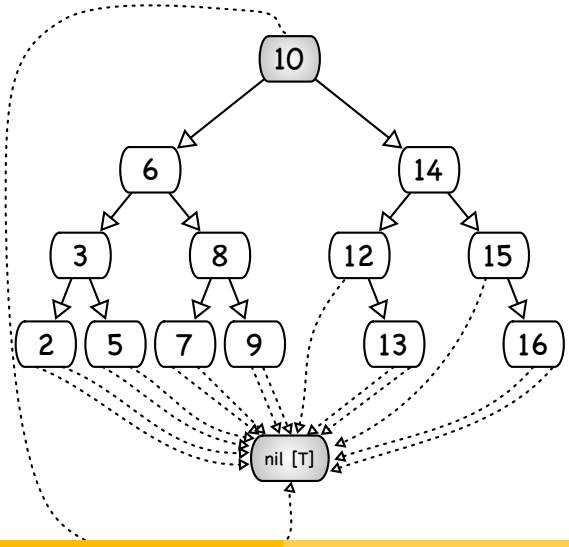
Ein Rot-Schwarz-Baum (noch ohne Färbung)



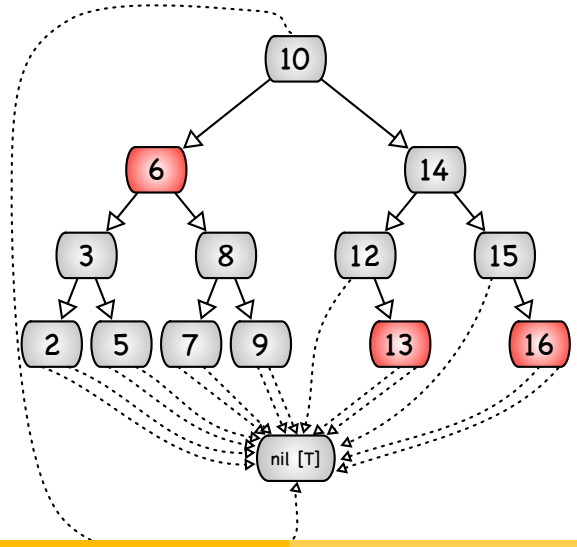
Ein Rot-Schwarz-Baum (noch ohne Färbung)



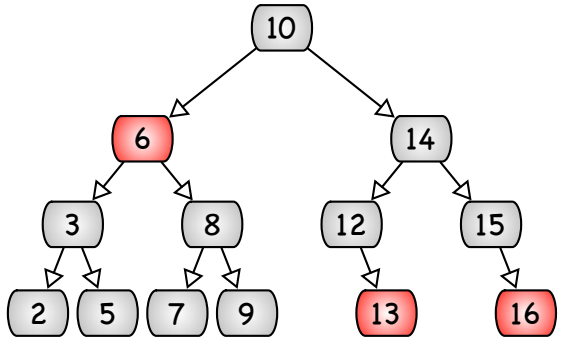
Ein Rot-Schwarz-Baum (Wurzel und Blätter gefärbt)



Ein Rot-Schwarz-Baum (gefärbt)



Ein Rot-Schwarz-Baum (in schön)



Rot-Schwarz-Eigenschaften (Wiederholung)

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ist schwarz.
4. *Wenn ein Knoten rot ist, dann sind seine beiden Kinder schwarz.*
5. *Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten*

Anmerkung

Die Blätter sind *alle* `nil`-Knoten (der Wächter `nil`$[T]$) und schwarz. Die Wurzel hat als Vorgänger auch `nil`$[T]$ (Siehe die beiden vorherigen Bilder.)

Balanciertheit

Satz

Ein Rot-Schwarz-Baum mit n inneren Knoten hat höchstens die Höhe $2 \cdot \log(n+1)$.

Satz

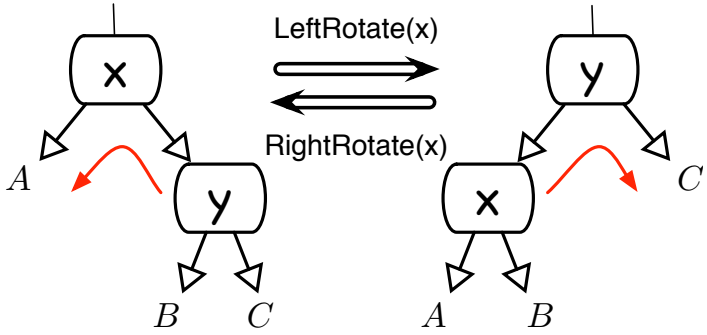
Die Operationen Search, Minimum, Maximum, Successor, Predecessor für dynamische Mengen lassen sich auf Rot-Schwarz-Bäumen in Zeit $O(\log n)$ implementieren, wobei n die Anzahl der Knoten ist.

Rotationen

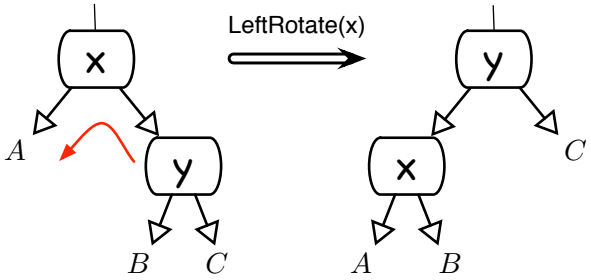
Auch die Operationen TreeInsert und TreeDelete sind in Zeit $O(\log n)$, *aber* da sie den Baum modifizieren, kann es passieren, dass die Rot-Schwarz-Eigenschaften danach nicht mehr erfüllt sind.

Um die Eigenschaften wiederherzustellen, müssen ggf. Farben und die Zeigerstruktur geändert werden. Ein wichtiges Hilfsmittel hierbei (zur Änderung der Zeigerstruktur) ist die *Rotation* ...

Rotation - Bildlich

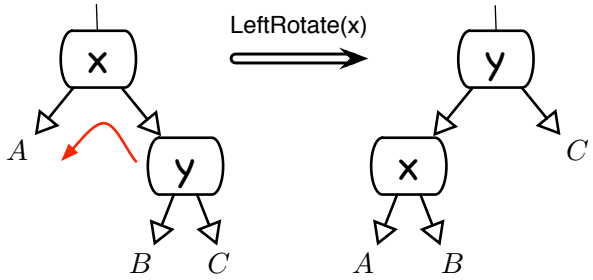


Linksrotation



- Das rechte Kind (y) von x ist nicht `nil`.
- y wird neue Wurzel des Teilbaumes.
- x wird das *linke* Kind von y.
- Das linke Kind von y wird rechtes Kinds von x.

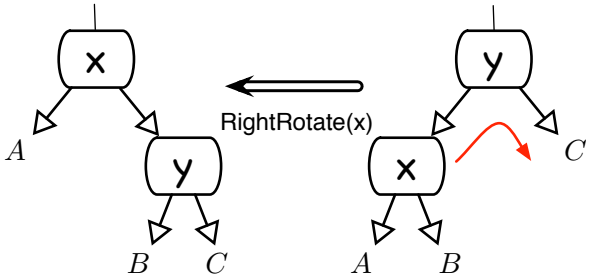
Linksrotation - zum Merken



- Ziehe an A-x-y-C wie an einer Schnur.
- A und x rutschen nach unten, y und C nach oben
- B bleibt übrig. Einziger sinnvoller Platz: rechts von x.

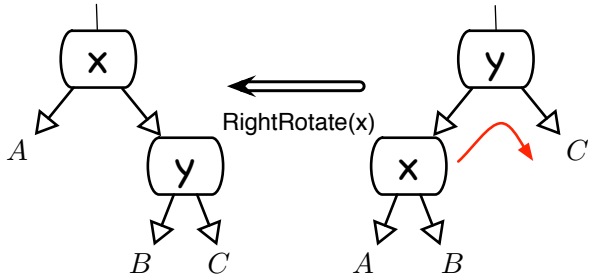
$\Rightarrow$ Erhälte die binäre Suchbaum-Eigenschaft!

Rechtsrotation



- Das linke Kind (x) von y ist nicht `nil`.
- x wird neue Wurzel des Teilbaumes.
- y wird das *rechte* Kind von y.
- Das rechte Kind von x wird linkes Kinds von y.

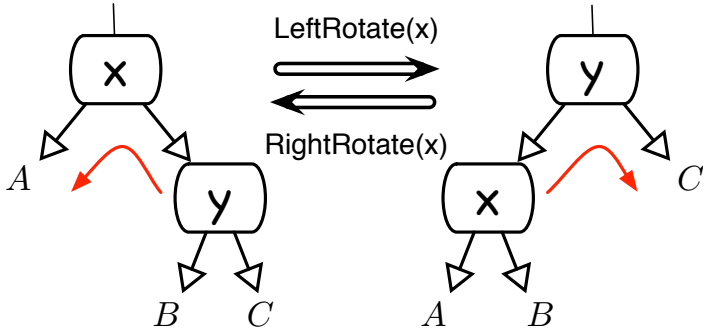
Rechtsrotation - zum Merken



- Ziehe an A-x-y-C wie an einer Schnur.
- C und y rutschen nach unten, x und A nach oben
- B bleibt übrig. Einziger sinnvoller Platz: links von y.

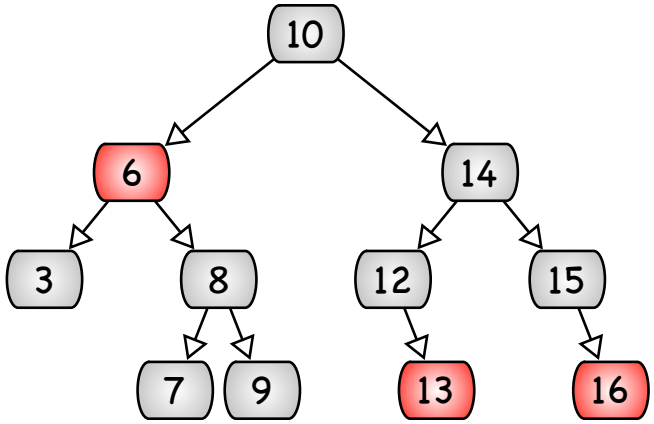
$\Rightarrow$ Erhälte die binäre Suchbaum-Eigenschaft!

Rotation - Bildlich

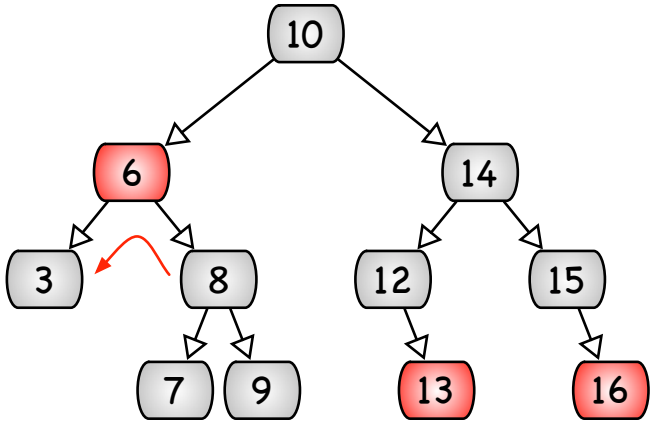


Die Rotation erhält die binäre Suchbaum-Eigenschaft!

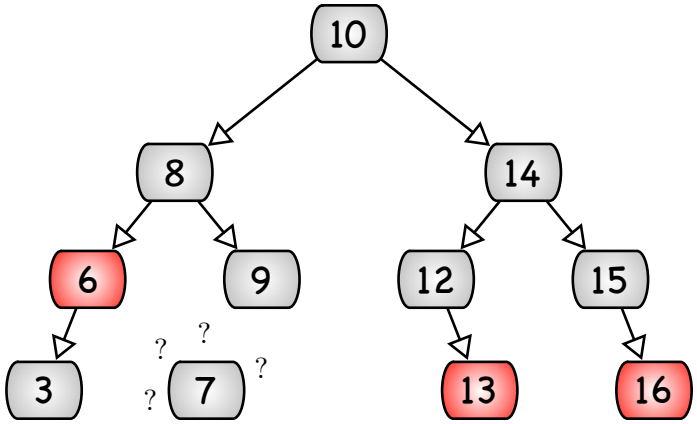
Rotation - Beispiel



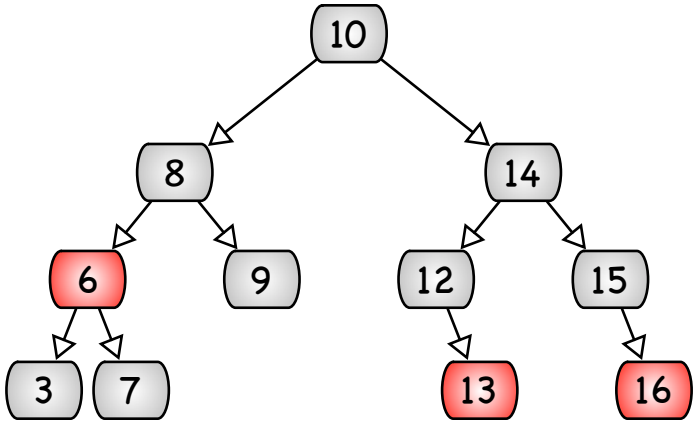
Rotation - Beispiel



Rotation - Beispiel



Rotation - Beispiel



Rotation - Implementation

- Die Implementierung ist wieder 'Zeigerspielerei'. (Ähnlich des Ausschneidens in verketteten Listen.)
- Details sind im [Cormen] zu finden.
- Laufzeit ist in $O(1)$, da nur (eine konstante Anzahl von) Zeiger verändert werden.

Einfügen

Beim Einfügen passiert nun zunächst das gleiche wie bei normalen binären Suchbäumen.

Die Farbe des neu eingefügten Knotens wird auf 'rot' gesetzt.

Dann wird eine Routine aufgerufen, die die Rot-Schwarz-Eigenschaften wiederherstellt...

Einfügen - Grossvater, Onkel und Vater...

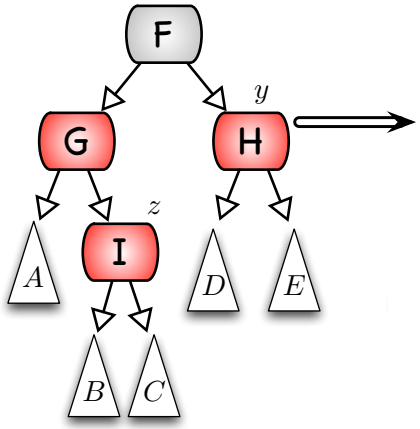
z ist der neu eingefügte, rote Knoten. In einer while-Schleife, in der immer geprüft wird, ob der Vater von z rot ist (nur dann muss man fortfahren), werden nun drei Fälle unterschieden. Die drei Fälle treten 'spiegelverkehrt' auf, je nachdem ob z im linken Teilbaum seines *Grossvaters* ist, oder im rechten. Wir behandeln nachfolgend die erste Variante (z ist im linken Teilbaum seines Grossvaters).

Eine wichtige Rolle spielt der Onkel $y(= \mathit{rechts}[p[p[z]]])$ von z und insb. dessen Farbe sowie, ob z ein linkes oder rechtes Kind ist.

(z hat nachfolgend Kinder, weil in der Routine der Zeiger z nach oben wandern kann.)

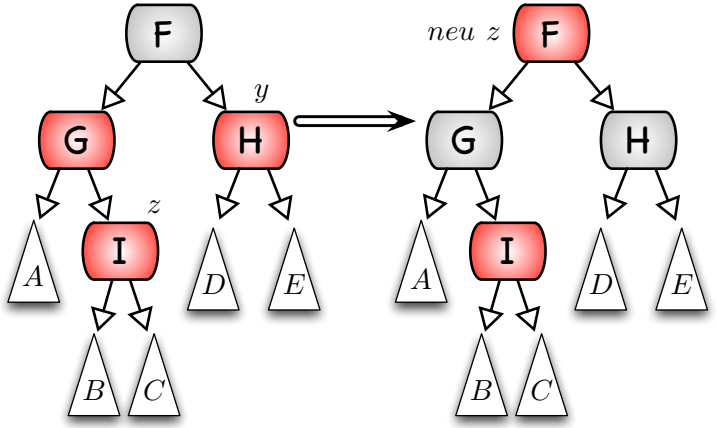
Einfügen - Fall 1

Fall 1: z's Onkel y ist rot. Umfärben und z neu (!) setzen.



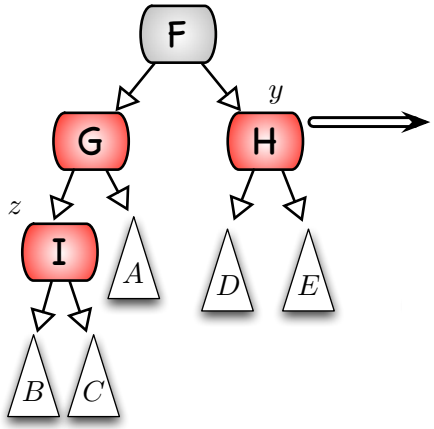
Einfügen - Fall 1

Fall 1: z's Onkel y ist rot. Umfärben und z neu (!) setzen.



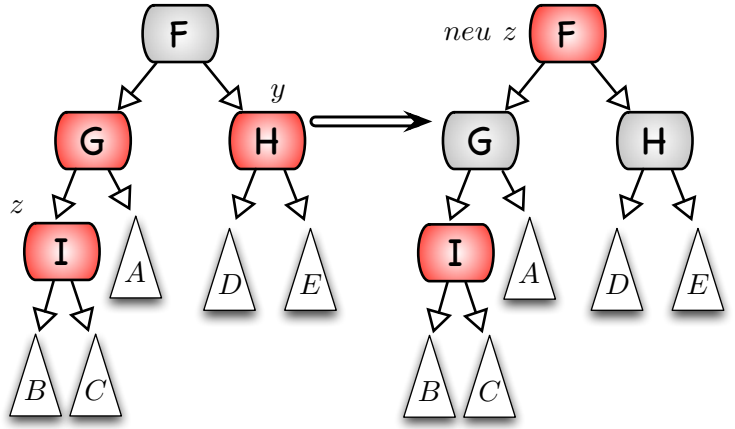
Einfügen - Fall 1b

Fall 1b: z's Onkel y ist rot. Umfärben und z neu (!) setzen.



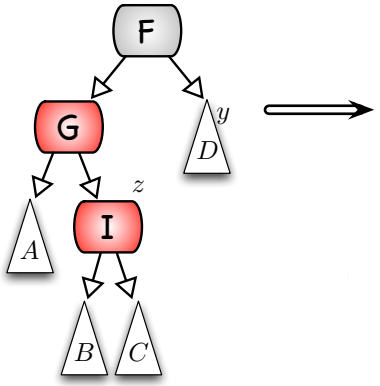
Einfügen - Fall 1b

Fall 1b: z's Onkel y ist rot. Umfärben und z neu (!) setzen.



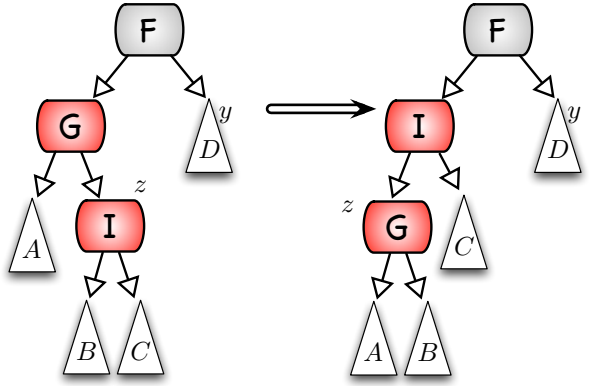
Einfügen - Fall 2

Fall 2: z's Onkel ist schwarz und z ist rechtes Kind seines Vaters. z wird auf $p[z]$ gesetzt und eine Linksrotation wird um (das neue) z gemacht. Dann weiter mit Fall 3. (Der gilt jetzt!)



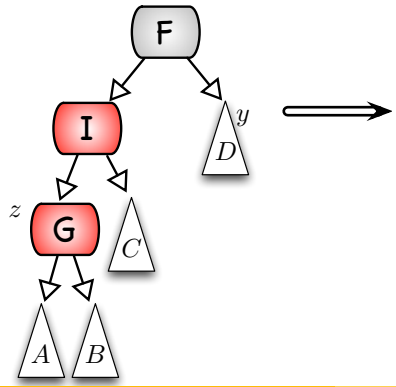
Einfügen - Fall 2

Fall 2: z's Onkel ist schwarz und z ist rechtes Kind seines Vaters. z wird auf $p[z]$ gesetzt und eine Linksrotation wird um (das neue) z gemacht. Dann weiter mit Fall 3. (Der gilt jetzt!)



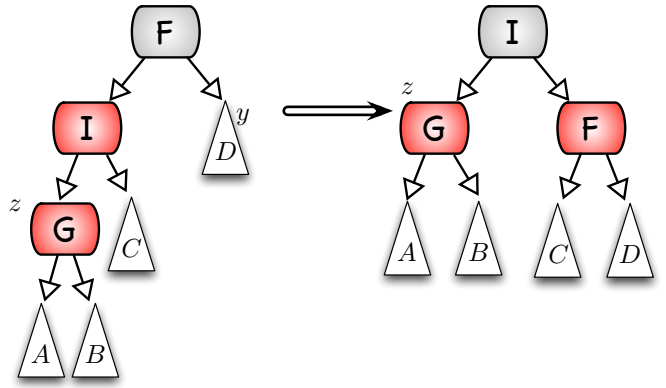
Einfügen - Fall 3

Fall 3: z's Onkel ist schwarz und z ist linkes Kind seines Vaters. Farbe von $p[z]$ auf schwarz (bisher rot), von $p[p[z]]$ auf rot (bisher schwarz) und Rechtsrotation um $p[p[z]]$. (z nicht neu setzen!)

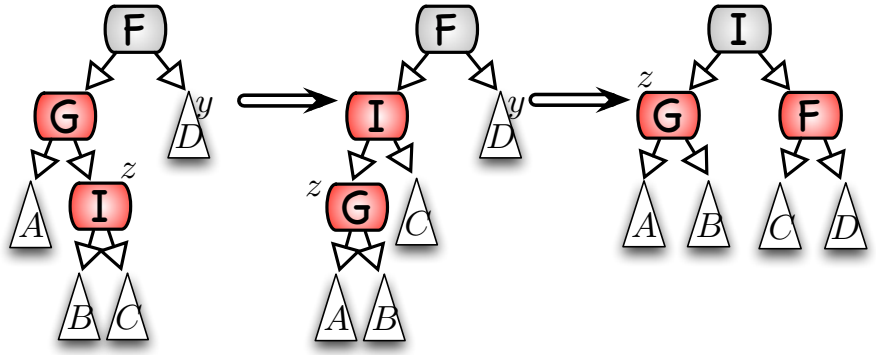


Einfügen - Fall 3

Fall 3: z's Onkel ist schwarz und z ist linkes Kind seines Vaters. Farbe von $p[z]$ auf schwarz (bisher rot), von $p[p[z]]$ auf rot (bisher schwarz) und Rechtsrotation um $p[p[z]]$. (z nicht neu setzen!)



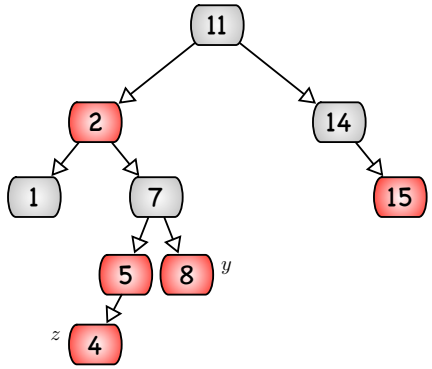
Einfügen - Fall 2 & 3 in Folge



Einfügen - Beispiel

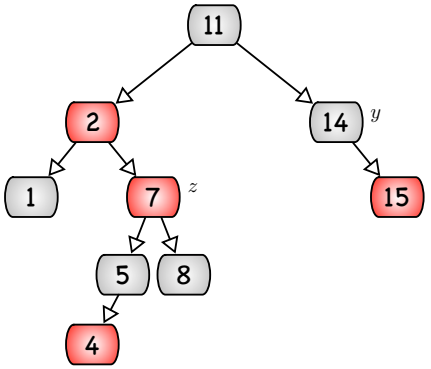
Nun einmal am Beispiel. Der Knoten 4 wird hier neu eingefügt...

Einfügen - Beispiel



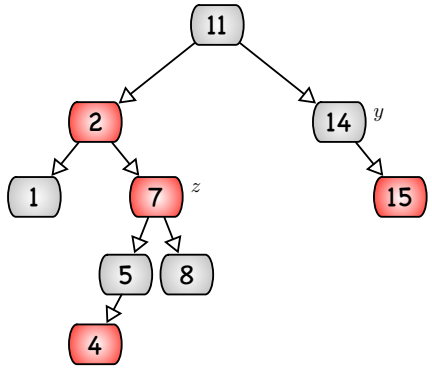
Fall 1: z's Onkel y ist rot. Umfärben und z auf $p[p[z]]$ neu setzen.

Einfügen - Beispiel



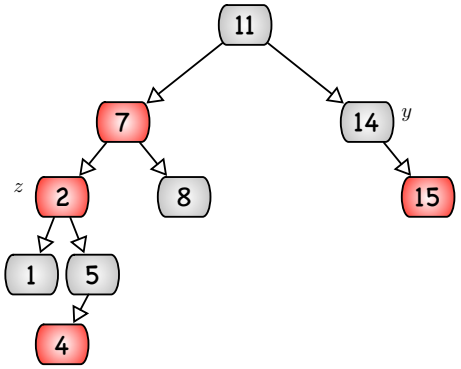
Fall 1: z's Onkel y ist rot. Umfärben und z auf $p[p[z]]$ neu setzen.

Einfügen - Beispiel



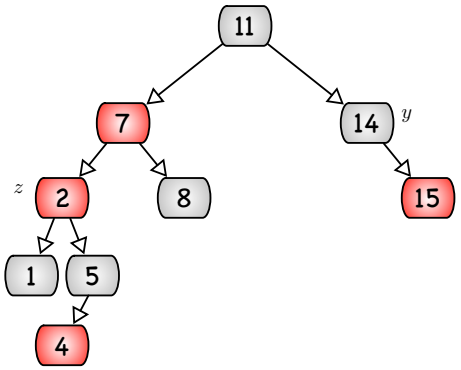
Fall 2: z's Onkel ist schwarz und z ist rechtes Kind seines Vaters. z wird auf $p[z]$ gesetzt und eine Linksrotation wird um (das neue) z gemacht. Dann weiter mit Fall 3. (Der gilt jetzt!)

Einfügen - Beispiel



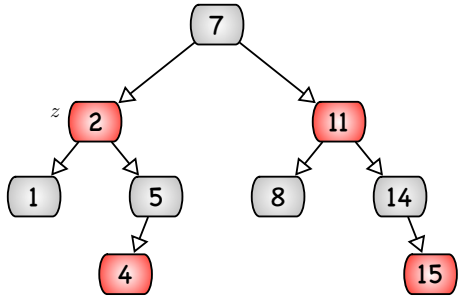
Fall 2: z's Onkel ist schwarz und z ist rechtes Kind seines Vaters. z wird auf $p[z]$ gesetzt und eine Linksrotation wird um (das neue) z gemacht. Dann weiter mit Fall 3. (Der gilt jetzt!)

Einfügen - Beispiel



Fall 3: z's Onkel ist schwarz und z ist linkes Kind seines Vaters. Farbe von $p[z]$ auf schwarz (bisher rot), von $p[p[z]]$ auf rot (bisher schwarz) und Rechtsrotation um $p[p[z]]$.

Einfügen - Beispiel



Fall 3: z's Onkel ist schwarz und z ist linkes Kind seines Vaters. Farbe von $p[z]$ auf schwarz (bisher rot), von $p[p[z]]$ auf rot (bisher schwarz) und Rechtsrotation um $p[p[z]]$.

Einfügen - Zusammenfassung

Der neu eingefüge Knoten z ist im linken Teilbaum seines Grossvaters. z und sein Vater sind rot.

- Fall 1: z's Onkel y ist rot. Umfärben und z auf $p[p[z]]$ neu setzen. (Es gibt hier zwei Fälle, die gleich behandelt werden.)
- Fall 2: z's Onkel ist schwarz und z ist rechtes Kind seines Vaters. z wird auf $p[z]$ gesetzt und eine Linksrotation wird um (das neue) z gemacht. Dann weiter mit Fall 3. (Der gilt jetzt!)
- Fall 3: z's Onkel ist schwarz und z ist linkes Kind seines Vaters. Farbe von $p[z]$ auf schwarz (bisher rot), von $p[p[z]]$ auf rot (bisher schwarz) und Rechtsrotation um $p[p[z]]$. (z nicht neu setzen, die Routine terminiert hier.)

Ist z im rechten Teilbaum seines Grossvaters treten obige Fälle spiegelverkehrt auf. Überall muss dann links und rechts vertauscht werden.

Einfügen - Die Schleifeninvariante

Die Schleifeninvariante:

a) *Der Knoten z ist rot.*
b) *Falls $p[z]$ die Wurzel ist, ist $p[z]$ schwarz.*
c) *Falls es eine Verletzung der Rot-Schwarz-Eigenschaften gibt, gibt es höchstens eine und diese verletzt Eigenschaft 2 oder 4. Ist es 2, so ist z die Wurzel und rot, ist es 4, so sind z und $p[z]$ beide rot.*

Einfügen - Jetzt zu tun...

Wir müssten jetzt - wir **könnten** jetzt...

1. Den Pseudocode ausführlicher schreiben.

⇒ Dabei insb. auf 'Randfälle' achten (leerer Baum, z hat keinen Onkel, ...)

2. Die Schleifeninvariante bei Initialisierung zeigen.
3. Fortsetzung zeigen, wobei hier primär gezeigt werden muss, dass die drei obigen Fälle die Invariante zu Beginn der nächsten Iteration erhalten.
4. Terminierung zeigen und (unter Nutzung der Schleifeninvarianten) dann die Korrektheit des Algorithmus zeigen, d.h. dass wieder ein Rot-Schwarz-Baum vorliegt.

Den genauen Beweis der Schleifeninvarianten und den Pseudocode der Einfügeoperation (sowie der Rotation) findet man im [Cormen]. Dort ist auch eine Behandlung der Löschoperation zu finden.

Zusammenfassung

Zusammenfassung

- Rot-Schwarz-Bäume sind binäre Suchbäume, die die zusätzlichen Rot-Schwarz-Eigenschaften erfüllen.
- Ein Baum der die Rot-Schwarz-Eigenschaften erfüllt ist in etwa ausgeglichen ($h \leq 2 \cdot \log(n+1)$).
- Der Erhalt der Rot-Schwarz-Eigenschaften ist insb. bei der Einfüge- und der Löschoperation zu beachten. Sie können in $O(\log n)$ implementiert werden.
- Damit erlauben Rot-Schwarz-Bäume die wichtigen Operationen für dynamische Mengen alle in $O(\log n)$ Zeit!
- (Allerdings sind sie dafür aufwändiger als normale binäre Suchbäume.)