

Datenstrukturen und Algorithmen

Vorlesung 10: Binäre Suchbäume

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<http://moves.rwth-aachen.de/teaching/ss-15/dsa1/>

8. Mai 2015



Übersicht

1 Binäre Suchbäume

- Suche
- Einfügen
- Einige Operationen (die das Löschen vereinfachen)
- Löschen

2 Rotationen

Übersicht

1 Binäre Suchbäume

- Suche
- Einfügen
- Einige Operationen (die das Löschen vereinfachen)
- Löschen

2 Rotationen

Motivation

Suchbäume unterstützen Operationen auf **dynamischen** Mengen, wie:

- ▶ Suchen, Einfügen, Löschen, Abfragen (z. B. Nachfolger oder minimales Element)

Die Basisoperationen auf binären Suchbäumen benötigen eine Laufzeit, die proportional zur Höhe des Baums ist.

Für vollständige binäre Bäume mit n Elementen liefert dies eine Laufzeit in $\Theta(\log(n))$ für eine Basisoperation.

Für einen Baum, der einer linearen Kette entspricht, ist dies jedoch in $\Theta(n)$.

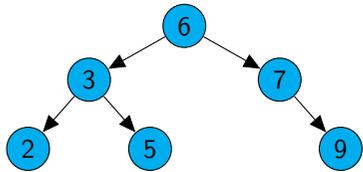
Wir werden später binäre Suchbäume kennen lernen, deren Operationen immer Laufzeiten in $\Theta(\log(n))$ haben (s. nächste Vorlesung).

Binäre Suchbäume (I)

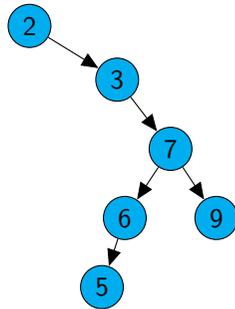
Binärer Suchbaum

Ein **binärer Suchbaum** (BST) ist ein Binärbaum, der Elemente mit Schlüsseln enthält, wobei der Schlüssel jedes Knotens

- ▶ **mindestens** so groß wie jeder Schlüssel im **linken** Teilbaum und
- ▶ **höchstens** so groß wie jeder Schlüssel im **rechten** Teilbaum ist.



Zwei binäre Suchbäume, die jeweils die Schlüssel 2, 3, 5, 6, 7, 9 enthalten.



Binäre Suchbäume (III)

Beispiel (Binärer Suchbaum in C/C++)

```

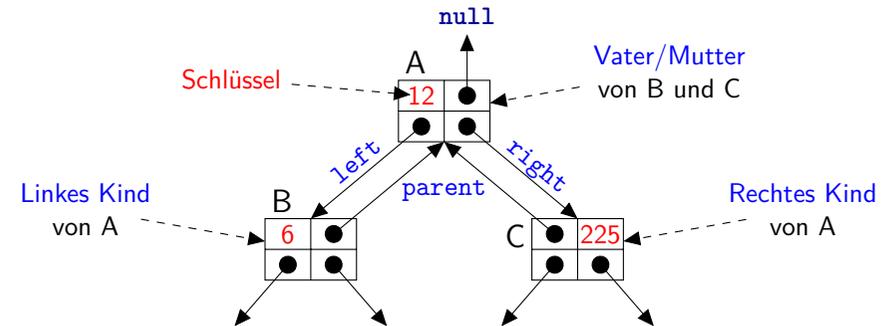
1 typedef struct _node* Node;
2 struct _node {
3     int key;
4     Node left, right;
5     Node parent;
6     // ... evtl. eigene Datenfelder
7 };

9 struct _tree {
10    Node root;
11 };
12 typedef struct _tree* Tree;
  
```

Binäre Suchbäume (II)

Knoten in einem binären Suchbaum bestehen aus vier Feldern:

- ▶ Einem **Schlüssel** – dem „Wert“ des Knotens,
- ▶ einem (möglicherweise leeren) **linken** und **rechten** Teilbaum (bzw. Zeiger darauf) sowie
- ▶ einem Zeiger auf den Vater-/Mutterknoten (bei der Wurzel leer).



Sortieren in linearer Zeit?

Sortieren

Eine **Inorder** Traversierung eines binären Suchbaumes gibt alle Schlüssel im Suchbaum in **sortierter** Reihenfolge aus.

Die Korrektheit dieses Sortierverfahrens folgt per Induktion direkt aus der BST-Eigenschaft.

Beispiel

Beispiel Inorder Traversierung BST am Overheadprojektor.

Zeitkomplexität

Da die Zeitkomplexität einer Inorder Traversierung eines Baumes mit n Knoten $\Theta(n)$ ist, liefert uns dies einen Sortieralgorithmus in $\Theta(n)$.

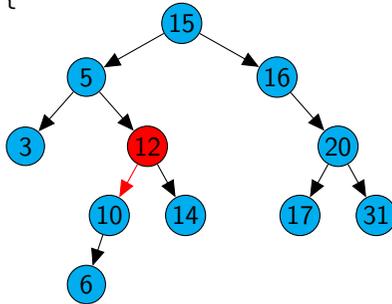
Dies setzt jedoch voraus, dass alle Daten als ein BST gespeichert sind.

Suche nach Schlüssel k im BST – $k = 10$

```

1 Node bstSearch(Node root, int k) {
2   while (root) {
3     if (k < root.key) {
4       root = root.left;
5     } else if (k > root.key) {
6       root = root.right;
7     } else { // k == root.key
8       return root;
9     }
10  }
11  return null; // nicht gefunden
12 }

```



Die Worst-Case Komplexität ist **linear** in der Höhe h des Baumes: $\Theta(h)$.

- ▶ Für einen kettenartigen Baum mit n Knoten ergibt das $\Theta(n)$.
- ▶ Ist der BST so balanciert wie möglich, erhält man $\Theta(\log(n))$.

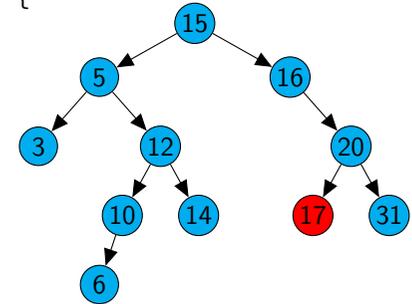
Funktioniert dieses Suchverfahren auch bei Heaps? **Nein**.

Suche nach Schlüssel k im BST – $k = 18$ (erfolglos)

```

1 Node bstSearch(Node root, int k) {
2   while (root) {
3     if (k < root.key) {
4       root = root.left;
5     } else if (k > root.key) {
6       root = root.right;
7     } else { // k == root.key
8       return root;
9     }
10  }
11  return null; // nicht gefunden
12 }

```



Die Worst-Case Komplexität ist **linear** in der Höhe h des Baumes: $\Theta(h)$.

- ▶ Für einen kettenartigen Baum mit n Knoten ergibt das $\Theta(n)$.
- ▶ Ist der BST so balanciert wie möglich, erhält man $\Theta(\log(n))$.

Funktioniert dieses Suchverfahren auch bei Heaps? **Nein**.

Einfügen eines Knotens mit Schlüssel k – Strategie

Einfügen

Man kann einen neuen Knoten mit Schlüssel k in den BST t einfügen, ohne die BST-Eigenschaft zu zerstören:

Suche einen geeigneten, freien Platz:

Wie bei der regulären Suche, außer dass, **selbst bei gefundenem Schlüssel**, weiter abgestiegen wird, bis ein Knoten ohne entsprechendes Kind erreicht ist.

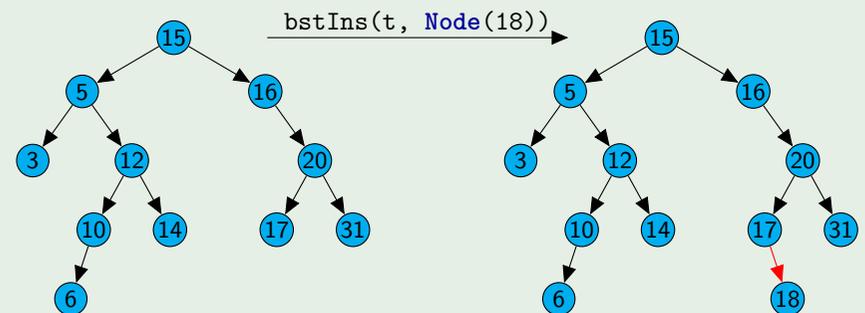
Hänge den neuen Knoten an:

Verbinde den neuen Knoten mit dem gefundenen Vaterknoten.

- ▶ Komplexität: $\Theta(h)$, wegen der Suche.

Einfügen von 18 in den BST t – Beispiel

Beispiel



Einfügen in einen BST – Algorithmus

```

1 void bstIns(Tree t, Node node) { // Füge node in den Baum t ein
2   // Suche freien Platz
3   Node root = t.root, parent = null;
4   while (root) {
5     parent = root;
6     if (node.key < root.key) {
7       root = root.left;
8     } else {
9       root = root.right;
10    }
11  } // Einfügen
12  node.parent = parent;
13  if (!parent) { // t war leer => neue Wurzel
14    t.root = node;
15  } else if (node.key < parent.key) { // richtige Seite ...
16    parent.left = node;
17  } else {
18    parent.right = node;
19  }
20 }

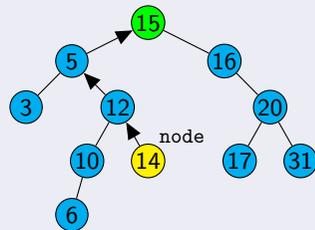
```

Abfragen im BST: Nachfolger (I)

Problem

Wir suchen den **Nachfolger**-Knoten von `node`, also den bei Inorder-Traversierung als nächstes zu besuchenden Knoten. Dessen Schlüssel ist mindestens so groß wie `node.key`.

Lösung



Abfragen im BST: Minimum

Problem

Wir suchen den Knoten mit **kleinstem Schlüssel** im durch `root` gegebenen (Teil-)Baum.

Lösung

```

1 Node bstMin(Node root) { // root != null
2   while (root.left) {
3     root = root.left;
4   }
5   return root;
6 }

```

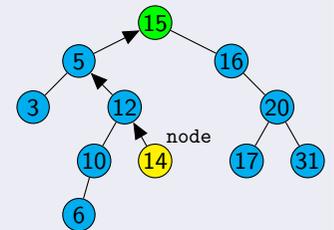
- ▶ Komplexität: $\Theta(h)$ bei Baumhöhe h .
- ▶ Analog kann das Maximum gefunden werden.

Abfragen im BST: Nachfolger (I)

Problem

Wir suchen den **Nachfolger**-Knoten von `node`, also den bei Inorder-Traversierung als nächstes zu besuchenden Knoten. Dessen Schlüssel ist mindestens so groß wie `node.key`.

Lösung



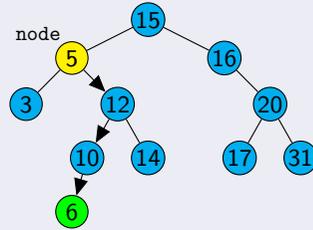
Abfragen im BST: Nachfolger (I)

Problem

Wir suchen den **Nachfolger**-Knoten von `node`, also den bei Inorder-Traversierung als nächstes zu besuchenden Knoten. Dessen Schlüssel ist mindestens so groß wie `node.key`.

Lösung

Der rechte Teilbaum existiert:
Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.



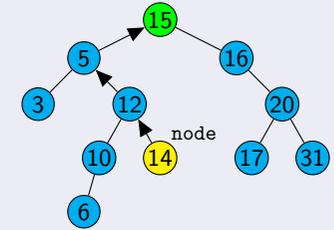
Abfragen im BST: Nachfolger (I)

Problem

Wir suchen den **Nachfolger**-Knoten von `node`, also den bei Inorder-Traversierung als nächstes zu besuchenden Knoten. Dessen Schlüssel ist mindestens so groß wie `node.key`.

Lösung

Der rechte Teilbaum existiert:
Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.
Andernfalls:
Der Nachfolger ist der jüngste Vorfahre, dessen *linker* Teilbaum `node` enthält.



Abfragen im BST: Nachfolger (I)

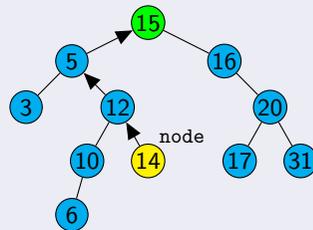
Problem

Wir suchen den **Nachfolger**-Knoten von `node`, also den bei Inorder-Traversierung als nächstes zu besuchenden Knoten. Dessen Schlüssel ist mindestens so groß wie `node.key`.

Lösung

Der rechte Teilbaum existiert:
Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.

Andernfalls:
Der Nachfolger ist der jüngste Vorfahre, dessen *linker* Teilbaum `node` enthält.



- Komplexität: $\Theta(h)$ bei Baumhöhe h .

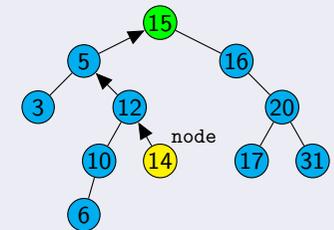
Abfragen im BST: Nachfolger (I)

Problem

Wir suchen den **Nachfolger**-Knoten von `node`, also den bei Inorder-Traversierung als nächstes zu besuchenden Knoten. Dessen Schlüssel ist mindestens so groß wie `node.key`.

Lösung

Der rechte Teilbaum existiert:
Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.
Andernfalls:
Der Nachfolger ist der jüngste Vorfahre, dessen *linker* Teilbaum `node` enthält.



- Komplexität: $\Theta(h)$ bei Baumhöhe h .
- Analog kann der Vorgänger gefunden werden.

Abfragen im BST: Nachfolger (II)

Der rechte Teilbaum existiert:

Der Nachfolger ist der kleinste Knoten im rechten Teilbaum.

Andernfalls:

Der Nachfolger ist der jüngste Vorfahre, dessen linker Teilbaum `node` enthält.

```

1 Node bstSucc(Node node) { // node != null
2   if (node.right) {
3     return bstMin(node.right);
4   }
5   // Abbruch, wenn node nicht mehr rechtes Kind ist (also linkes!)
6   // oder node.parent leer ist (also kein Nachfolger existiert).
7   while (node.parent && node.parent.right == node) {
8     node = node.parent;
9   }
10  return node.parent;
11 }

```

Ersetzen von Teilbäumen im BST

```

1 // Ersetzt im Baum t den Teilbaum old durch
2 // den Teilbaum node (ohne Sortierung!)
3 void bstReplace(Tree t, Node old, Node node) {
4   if (node) { // erlaube node == null!
5     node.parent = old.parent;
6   }
7   if (!old.parent) { // war die Wurzel
8     t.root = node;
9   } else if (old == old.parent.left) {
10    // war linkes Kind
11    old.parent.left = node;
12  } else { // rechtes Kind
13    old.parent.right = node;
14  }
15 }

```

Das Ersetzen eines Teilbaums hat die Zeitkomplexität $\Theta(1)$.

Austauschen von Knoten im BST

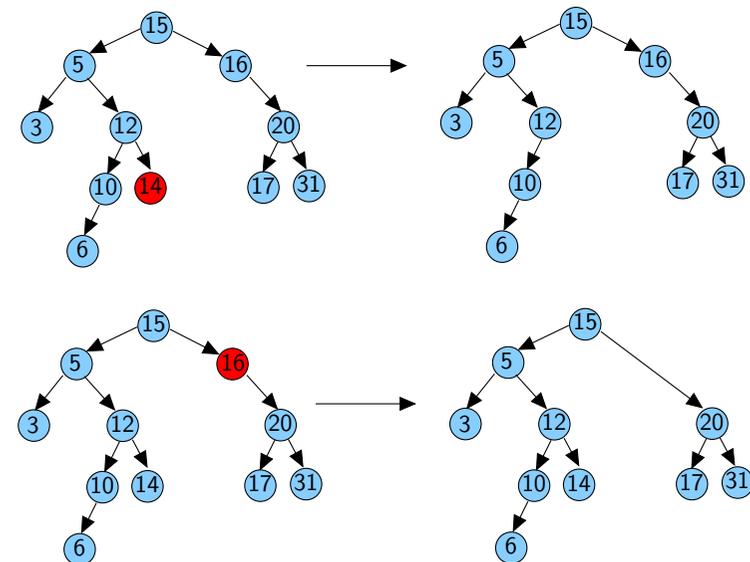
```

1 // Tauscht den Knoten old gegen node aus;
2 // die Kinder von old sind weiter im BST!
3 void bstSwap(Tree t, Node old, Node node) {
4   // übernimm linken Teilbaum
5   node.left = old.left; // auch möglich: swap()
6   if (node.left) {
7     node.left.parent = node;
8   }
9   // rechten Teilbaum
10  node.right = old.right;
11  if (node.right) {
12    node.right.parent = node;
13  }
14  // füge den Knoten ein
15  bstReplace(t, old, node);
16 }

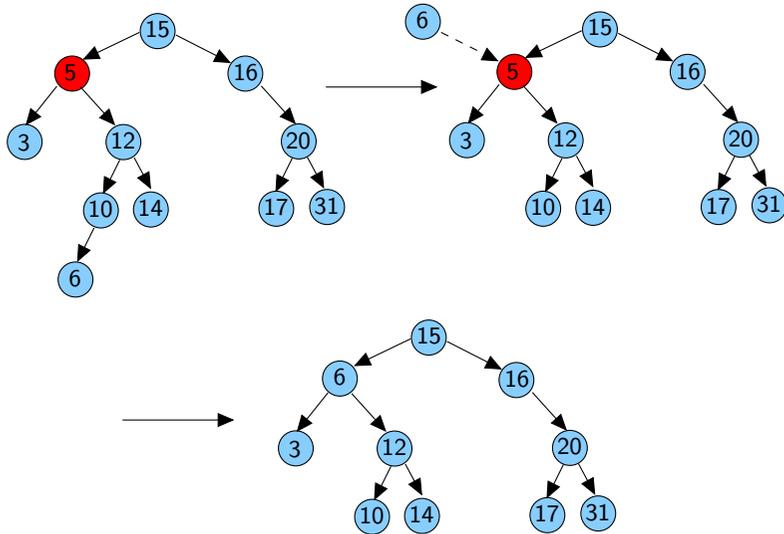
```

Das Austauschen eines Knotens hat die Zeitkomplexität $\Theta(1)$.

Löschen im BST: Die beiden einfachen Fälle



Löschen im BST: Der aufwändigere Fall



Löschen im BST – Algorithmus

```

1 // Entfernt node aus dem Baum.
2 // Danach kann node ggf. auch aus dem Speicher entfernt werden.
3 void bstDel(Tree t, Node node) {
4   if (node.left && node.right) { // zwei Kinder
5     Node tmp = bstMin(node.right);
6     bstDel(t, tmp); // höchstens ein Kind, rechts
7     bstSwap(t, node, tmp);
8   } else if (node.left) { // ein Kind, links
9     bstReplace(t, node, node.left);
10  } else { // ein Kind, oder kein Kind (node.right == null)
11    bstReplace(t, node, node.right);
12  }
13 }

```

Löschen im BST – Strategie

Löschen

Um Knoten `node` aus dem BST zu löschen, verfahren wir folgendermaßen:

node hat keine Kinder:

Ersetze im Vaterknoten von `node` den Zeiger auf `node` durch `null`.

node hat ein Kind:

Wir schneiden `node` aus, indem wir den Vater und das Kind direkt miteinander verbinden (den Teilbaum **ersetzen**).

node hat zwei Kinder:

Wir finden den **Nachfolger** von `node`, entfernen ihn aus seiner ursprünglichen Position und **tauschen** `node` gegen den Nachfolger.

- ▶ Es tritt nur der erste Fall (`bstMin(node.right)`) aus `bstSucc` auf.
- ▶ Der gesuchte Nachfolger hat **kein linkes Kind**.

Komplexität der Operationen auf BSTs

Operation	Zeit
<code>bstSearch</code>	$\Theta(h)$
<code>bstSucc</code>	$\Theta(h)$
<code>bstMin</code>	$\Theta(h)$
<code>bstIns</code>	$\Theta(h)$
<code>bstDel</code>	$\Theta(h)$

- ▶ Alle Operationen sind **linear** in der Höhe h des BSTs.
- ▶ Die Höhe ist $\log_2(n)$, wenn der Baum nicht zu „unbalanciert“ ist.
- ▶ Man kann einen binären Baum mittels **Rotationen** wieder balancieren.

Zufällig erzeugte binäre Suchbäume

Zufällig erzeugter BST

Ein zufällig erzeugter BST mit n Elementen ist ein BST, der durch das Einfügen von n (unterschiedlichen) Schlüsseln in zufälliger Reihenfolge in einen anfangs leeren Baum entsteht.

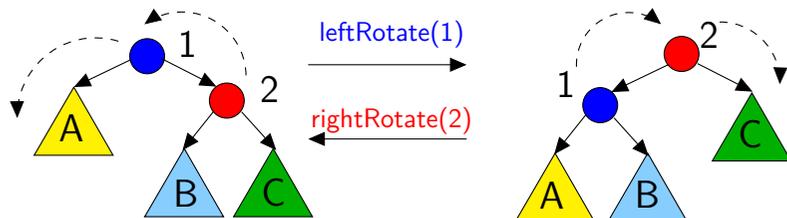
Annahme: jede der $n!$ möglichen Einfügensordnungen hat die gleiche Wahrscheinlichkeit.

Theorem (ohne Beweis)

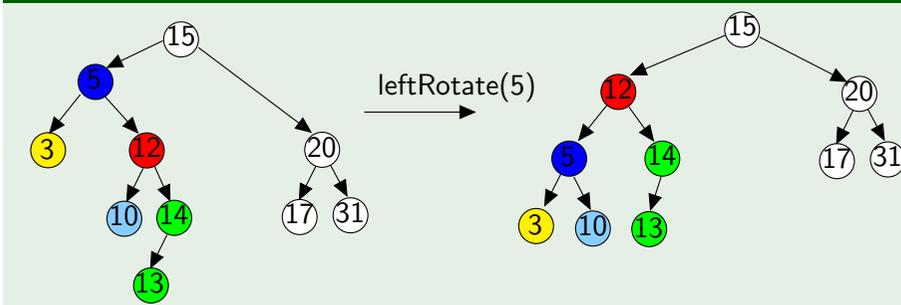
Die erwartete Höhe eines zufällig erzeugten BSTs mit n Elementen ist $O(\log(n))$.

Fazit: Im Schnitt verhält sich eine binäre Suchbaum wie ein (fast) balancierter Suchbaum.

leftRotate – Konzept und Beispiel



Beispiel

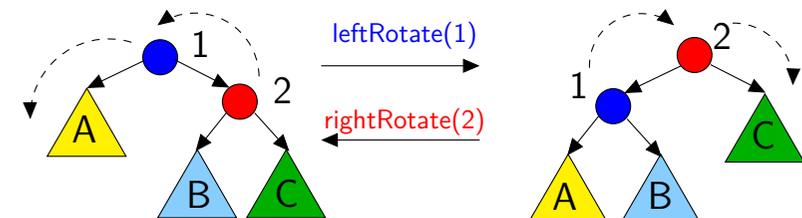


Übersicht

- 1 Binäre Suchbäume
 - Suche
 - Einfügen
 - Einige Operationen (die das Löschen vereinfachen)
 - Löschen

- 2 Rotationen

Rotationen: Eigenschaften und Komplexität



Lemma

- ▶ Ein rotierter BST ist ein BST
- ▶ Die Inorder-Traversierung beider Bäume bleibt **unverändert**.

Zeitkomplexität

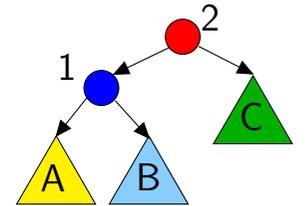
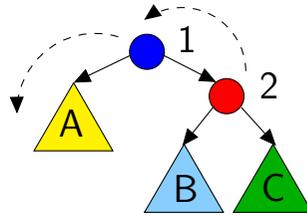
Die Zeitkomplexität von Links- oder Rechtsrotieren ist in $\Theta(1)$.

leftRotate – Algorithmus

```

1 void leftRotate(Tree t, Node node1) { // analog: rightRotate()
2   Node node2 = node1.right;
3   // Baum B verschieben
4   node1.right = node2.left;
5   if (node1.right) {
6     node1.right.parent = node1;
7   }
8   // node2 wieder einhängen
9   node2.parent = node1.parent;
10  if (!node1.parent) { // node1 war die Wurzel
11    t.root = node2;
12  } else if (node1 == node1.parent.left) { // war linkes Kind
13    node2.parent.left = node2;
14  } else { // war rechtes Kind
15    node2.parent.right = node2;
16  }
17  // node1 einhängen
18  node2.left = node1;
19  node1.parent = node2;
20 }

```



AVL-Bäume: Balancieren nach Einfügen

- ▶ Betrachten wir einen **AVL-Baum**.
- ▶ Jeder AVL-Baum ist (**höhen-**)**balanciert**, d. h., für alle Knoten x :

$$\underbrace{|\text{rechte Teilbaumhöhe} - \text{linke Teilbaumhöhe}|}_{\text{balance}(x)} \leq 1.$$

- ▶ Wir fügen einen neuen Knoten in den Baum ein.
- ▶ Dadurch kann der Baum **unbalanciert** werden.
- ▶ Balancierung durch **Rotation**.
- ▶ **Einfachrotation**, wenn die tieferen Blätter "außen" liegen.
- ▶ **Doppelrotation**, wenn die tieferen Blätter "innen" liegen.

Rotationen – AVL-Baum

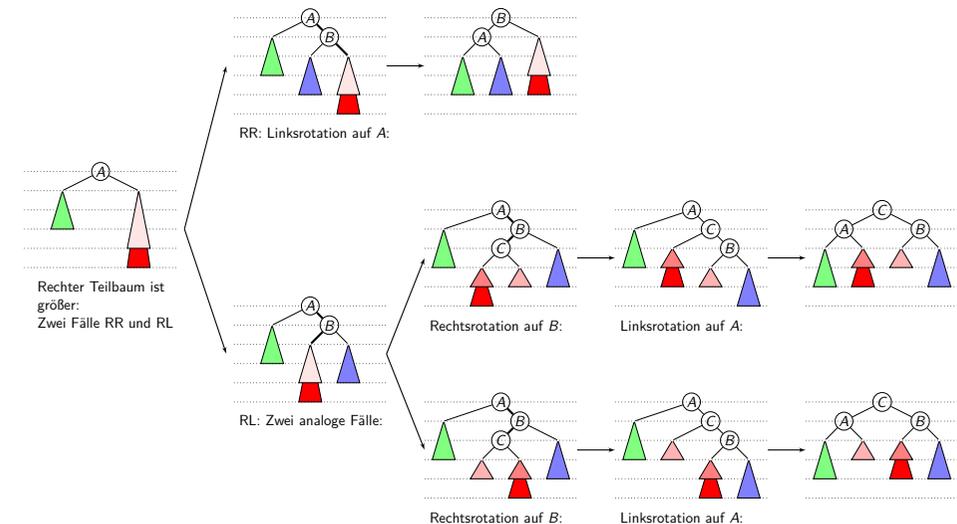
An welchen Knoten müssen die Rotationen durchgeführt werden?

AVL-Baum

- ▶ Ein **AVL-Baum** ist ein balancierter BST, bei dem für jeden Knoten die Höhe der beiden Teilbäume höchstens um 1 differiert.
- ▶ Bei **AVL-Bäumen** wird die **Höhe** der Teilbäume der Knoten **balanciert**.
- ▶ Dazu wird (in einem zusätzlichem Datenfeld) an jedem Knoten über die Höhe dieses Unterbaums Buch geführt.
- ▶ Nach jeder (kritischen) Operation wird die Balance wiederhergestellt. **Dies ist in $\Theta(h)$ möglich!**
- ▶ Dadurch bleibt stets $h \in \Theta(\log(n))$ und $\Theta(\log(n))$ kann für die Operationen auf dem BST **garantiert** werden.
- ▶ Eine andere Möglichkeit, um Bäume zu balancieren, sind **Rot-Schwarz-Bäume** (nächste Vorlesung).

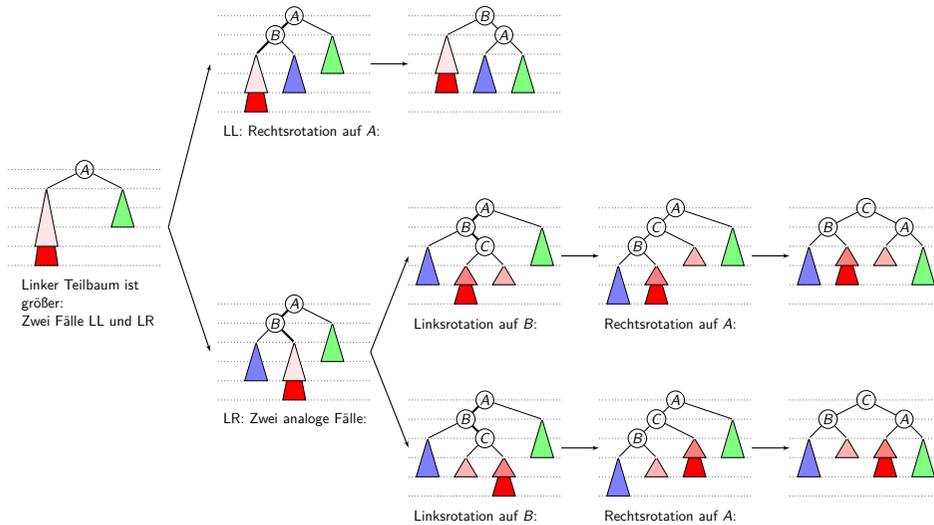
AVL-Bäume: Balancieren nach Einfügen

Sei A der tiefste unbalancierte Knoten auf dem Pfad von der Wurzel zum neu eingefügten Knoten (unbalanciert: $\text{linke Teilbaumhöhe} - \text{rechte Teilbaumhöhe} = \pm 2$).



AVL-Bäume: Balancieren nach Einfügen

Sei A der tiefste unbalancierte Knoten auf dem Pfad von der Wurzel zum neu eingefügten Knoten (unbalanciert: *linke Teilbaumhöhe* – *rechte Teilbaumhöhe* = ± 2).



AVL-Bäume: Balancieren nach Einfügen

```

1 void AVLIns(Tree t, Node node) {
2   bstIns(t,node);
3   //Node deepestUnbalancedNode(Tree t, Node node)
4   //gibt null zurück wenn t balanciert ist
5   //und den tiefsten unbalancierten Knoten in t sonst
6   //(der Parameter node wird zur effizienten Implementierung
7   //verwendet)
8   Node A = deepestUnbalancedNode(t,node);
9   if (A != null) balance(t, A);
10 }

```

AVL-Bäume: Balancieren nach Einfügen

```

1 void balance(Tree t, Node A){
2   //A ist tiefster unbalancierter Knoten in t
3   if (height(A.left) > height(A.right)) {
4     if (height(A.left.left) >= height(A.left.right)) { //LL
5       rightRotate(t,A);
6     } else { //LR
7       leftRotate(A.left); rightRotate(A);
8     }
9   } else {
10    if (height(A.right.right) >= height(A.right.left)) { //RR
11      leftRotate(t,A);
12    } else { //RL
13      rightRotate(A.right); leftRotate(A);
14    }
15  }
16 }

```

AVL-Bäume: Balancieren nach Löschen

- ▶ Baumhöhe von A nach der Rotation ist **wieder die gleiche** wie vor dem Einfügen des neuen Knotens.
- ▶ Das heißt, nach dem Balancieren von A ist der **gesamte** Baum wieder balanciert.
- ▶ Die zweite Operation, die Unbalanciertheit verursachen kann, ist das **Löschen** eines Knotens.
- ▶ Die Balancierung des tiefsten unbalancierten Knotens kann auf die gleiche Weise erreicht werden wie beim Einfügen.
- ▶ **Aber:** der Teilbaum hat **nicht** die gleiche Höhe wie vor dem Löschen (sie ist um 1 kleiner)!
- ▶ Im schlimmsten Fall müssen **alle** unbalancierten Knoten einzeln balanciert werden.
- ▶ Da aber die Balancierung eines Knotens nur einen konstanten Aufwand erfordert und es nur $\mathcal{O}(\log(n))$ unbalancierte Knoten geben kann, ist der Aufwand immer noch **logarithmisch**.

AVL-Bäume: Balancieren nach Löschen

```
1 void AVLDel(Tree t, Node node) {
2   bstDel(t,node);
3   //Node deepestUnbalancedNode(Tree t, Node node)
4   //gibt null zurück wenn t balanciert ist
5   //und den tiefsten unbalancierten Knoten in t sonst
6   //(der Parameter node wird zur effizienten Implementierung
7   //verwendet)
8   Node A = deepestUnbalancedNode(t,node);
9   while (A != null) {
10    //bool balanced(Tree t, Node A)
11    //gibt true zurück wenn A balanciert ist in t
12    //und false sonst
13    if (!balanced(t, A)) {
14      balance(t, A);
15      A = A.parent.parent;
16    } else {
17      A = A.parent;
18    }
19  }
20 }
```