Datenstrukturen und Algorithmen

Vorlesung 10: Binäre Suchbäume

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

https://moves.rwth-aachen.de/teaching/ss-18/dsal/

28. Mai 2018

Übersicht

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)

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.

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.

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 $\Theta(n)$.

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

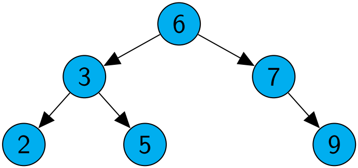
- mindestens so groß wie jeder Schlüssel im linken Teilbaum und
- höchstens so groß wie jeder Schlüssel im rechten Teilbaum ist.

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.

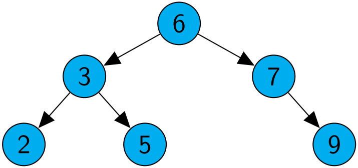


Binäre Suchbäume (I)

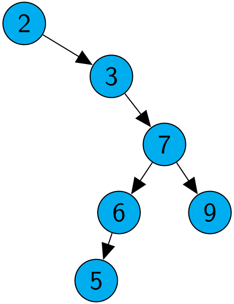
Binärer Suchbaum

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 (II)

Knoten in einem binären Suchbaum bestehen aus vier Feldern:

Binäre Suchbäume (II)

Knoten in einem binären Suchbaum bestehen aus vier Feldern:

- Einem Schlüssel – dem „Wert“ des Knotens,

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

Binäre Suchbäume (II)

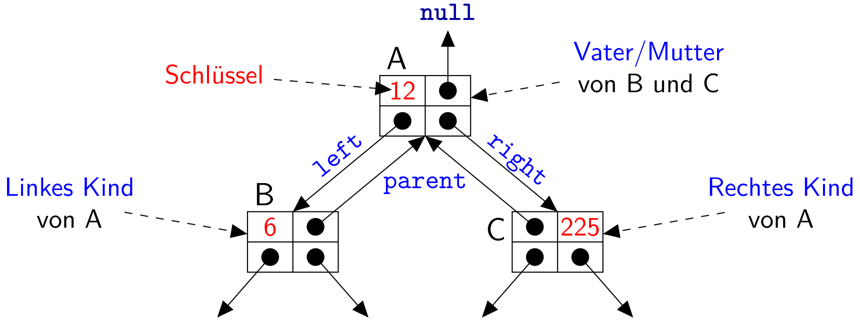
Knoten in einem binären Suchbaum bestehen aus vier Feldern:

- 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).

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).



Binäre Suchbäume (III)

Beispiel (Binärer Suchbaum in C/C++)

```
typedef struct _node* Node;
struct _node {
  int key;
  Node left, right;
  Node parent;
  // ... evtl. eigene Datenfelder
};

struct _tree {
  Node root;
};
typedef struct _tree* Tree;
```

Sortieren in linearer Zeit?

Sortieren

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

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.

Sortieren in linearer Zeit?

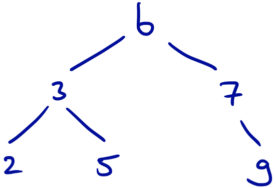
Sortieren

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.



inorder Traversierung!

inorder(3, 2, 5), 6, inorder(7, 9)

= 2, 3, 5, 6, 7, 9

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.

Zeitkomplexität

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

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.

Zeitkomplexität

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

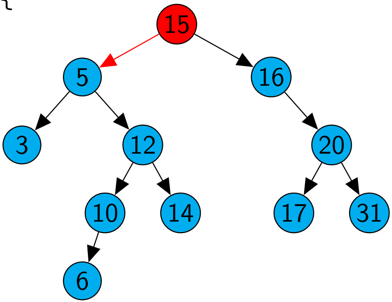
Dies setzt jedoch voraus, dass alle Daten als ein BST gespeichert sind.

Suche nach Schlüssel k im BST

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```

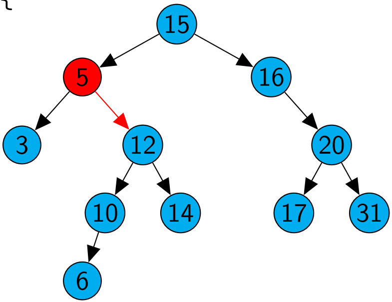
Suche nach Schlüssel k im BST – $k = 10$

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



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

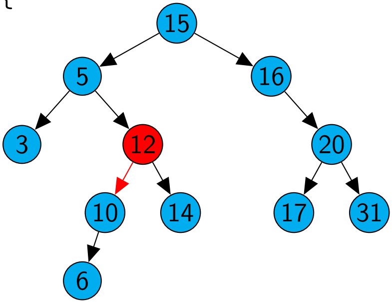
```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



 Binäre Suchbäume

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

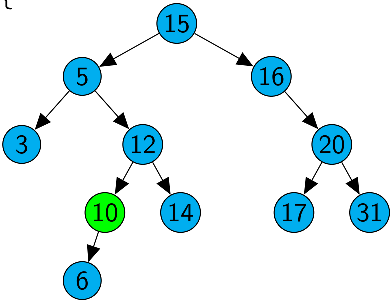
```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



 Binäre Suchbäume

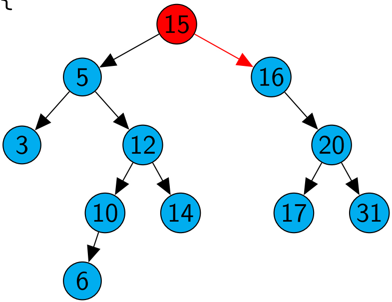
Suche nach Schlüssel k im BST – $k = 10$

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



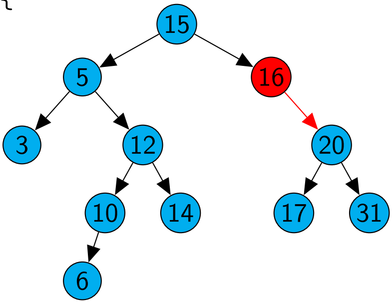
Suche nach Schlüssel k im BST – $k = 18$ (erfolglos)

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



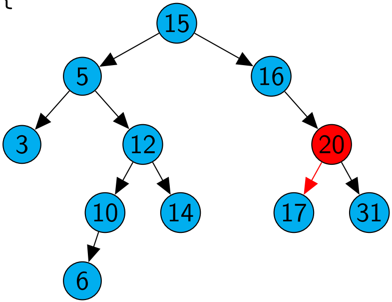
Suche nach Schlüssel k im BST – $k = 18$ (erfolglos)

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



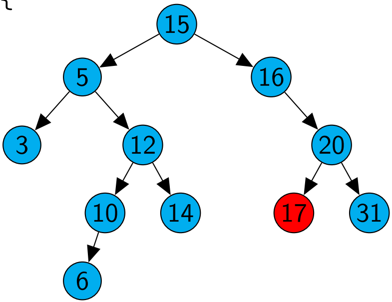
Suche nach Schlüssel k im BST – $k = 18$ (erfolglos)

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



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

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```



 Binäre Suchbäume

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

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```

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

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

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```

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

- Für einen kettenartigen Baum mit n Knoten ergibt das $\Theta(n)$.

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

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```

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 $\Theta(\log(n))$.

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

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```

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?

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

```
Node bstSearch(Node root, int k) {
  while (root) {
    if (k < root.key) {
      root = root.left;
    } else if (k > root.key) {
      root = root.right;
    } else { // k == root.key
      return root;
    }
  }
  return null; // nicht gefunden
}
```

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 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:

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:

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.

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.

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:

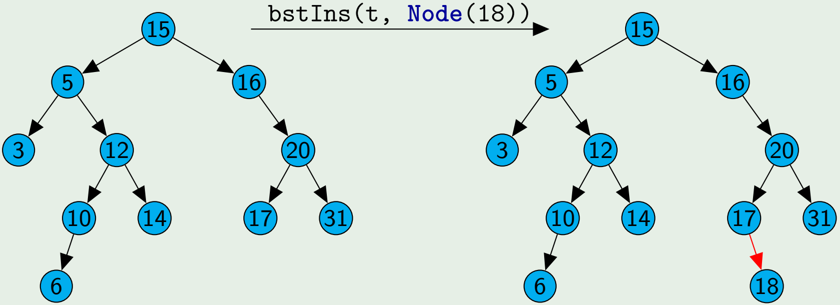
Suche einen geeigneten, freien Platz:
Wie bei der regulären Suche, außer dass, selbst bei gefundenem Schlüssel, weiter abgestiegen wird, bis ein Knoten ohne entsprechendes Kind erreicht ist.

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

```
void bstIns(Tree t, Node node) { // Füge node in den Baum t ein
  // Suche freien Platz
  Node root = t.root, parent = null;
  while (root) {
    parent = root;
    if (node.key < root.key) {
      root = root.left;
    } else {
      root = root.right;
    }
  } // Einfügen
  node.parent = parent;
  if (!parent) { // t war leer => neue Wurzel
    t.root = node;
  } else if (node.key < parent.key) { // richtige Seite ...
    parent.left = node;
  } else {
    parent.right = node;
  }
}
```

Pointers


MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

Abfragen im BST: Minimum

Problem

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

Lösung

```
Node bstMin(Node root) { // root != null
  while (root.left) {
    root = root.left;
  }
  return root;
}
```

- 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.

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`.

Abfragen im BST: Nachfolger (I)

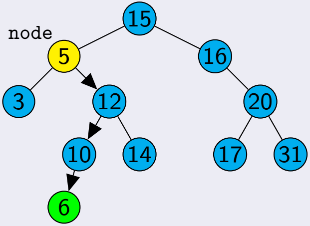
Problem

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`.

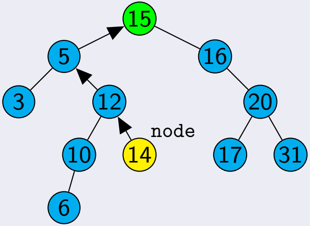
Lösung

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`.

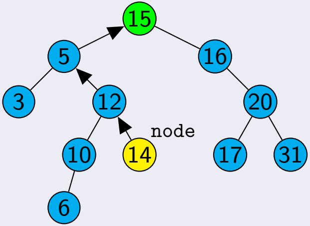
Lösung

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`.

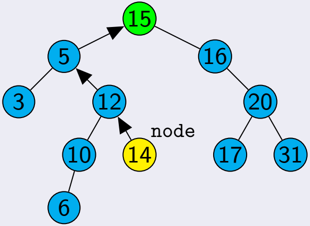
Lösung

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.

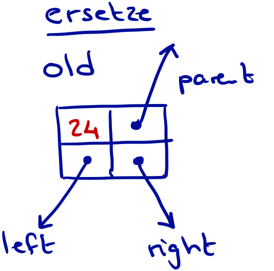
```
Node bstSucc(Node node) { // node != null
  if (node.right) {
    return bstMin(node.right);
  }
  // Abbruch, wenn node nicht mehr rechtes Kind ist (also linkes!)
  // oder node.parent leer ist (also kein Nachfolger existiert).
  while (node.parent && node.parent.right == node) {
    node = node.parent;
  }
  return node.parent;
}
```

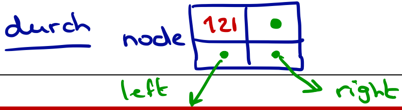
rechter Teilbaum existiert

Ersetzen von Teilbäumen im BST

```
// Ersetzt im Baum t den Teilbaum old durch
// den Teilbaum node (ohne Sortierung!)
void bstReplace(Tree t, Node old, Node node) {
  if (node) { // erlaube node == null!
    node.parent = old.parent;
  }
  if (!old.parent) { // war die Wurzel
    t.root = node;
  } else if (old == old.parent.left) {
    // war linkes Kind
    old.parent.left = node;
  } else { // rechtes Kind
    old.parent.right = node;
  }
}
```

node != null





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

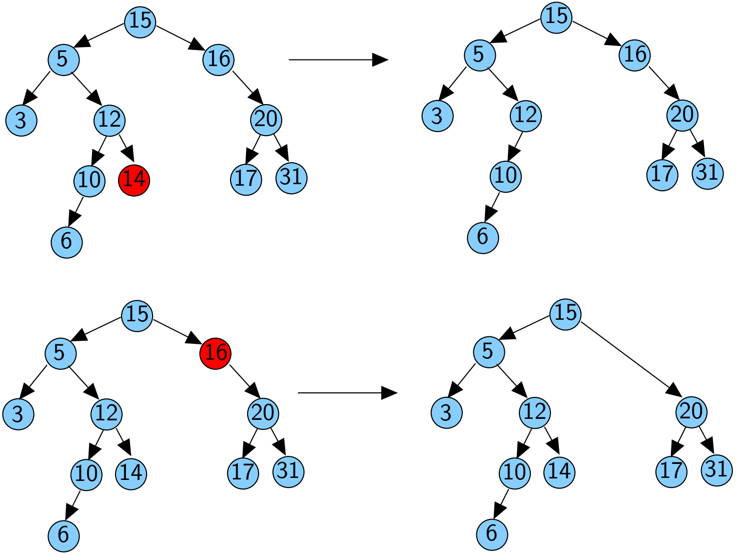
Austauschen von Knoten im BST

```
// Tauscht den Knoten old gegen node aus;
// die Kinder von old sind weiter im BST!
void bstSwap(Tree t, Node old, Node node) {
  // übernimm linken Teilbaum
  node.left = old.left; // auch möglich: swap()
  if (node.left) {
    node.left.parent = node;
  }
  // rechten Teilbaum
  node.right = old.right;
  if (node.right) {
    node.right.parent = node;
  }
  // füge den Knoten ein
  bstReplace(t, old, node);
}
```

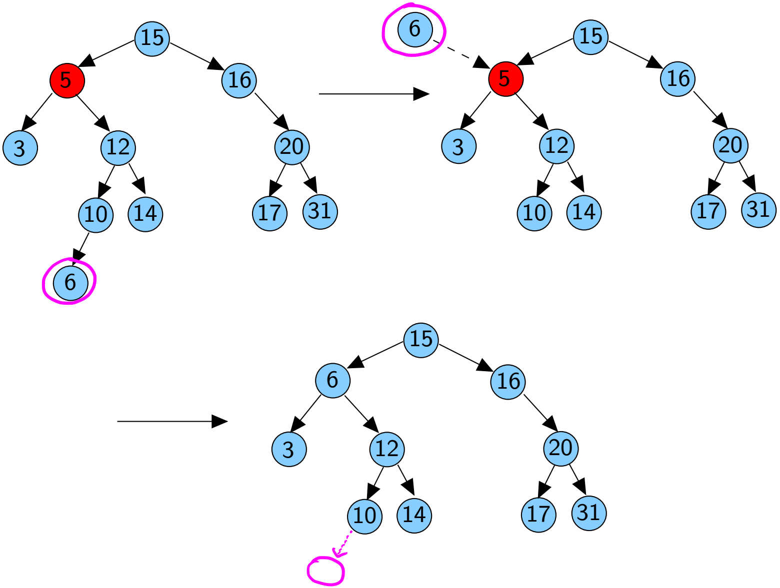
Das Austauschen eines Knotens hat die Zeitkomplexität $\Theta(1)$.

Binäre Suchbäume

Löschen im BST: Die beiden einfachen Fälle



Löschen im BST: Der aufwändigere Fall



Löschen im BST – Strategie

Löschen

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

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`**.

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).

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.

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.

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.

Löschen im BST – Algorithmus

```
// Entfernt node aus dem Baum.
// Danach kann node ggf. auch aus dem Speicher entfernt werden.
void bstDel(Tree t, Node node) {
  if (node.left && node.right) { // zwei Kinder
    Node tmp = bstMin(node.right);
    bstDel(t, tmp); // höchstens ein Kind, rechts
    bstSwap(t, node, tmp);
  } else if (node.left) { // ein Kind, links
    bstReplace(t, node, node.left);
  } else { // ein Kind, oder kein Kind (node.right == null)
    bstReplace(t, node, node.right);
  }
}
```

Komplexität der Operationen auf BSTs

Operation	Zeit
`bstSearch`	$\Theta(h)$
`bstSucc`	$\Theta(h)$
`bstMin`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$

- Alle Operationen sind linear in der Höhe h des BSTs.

Komplexität der Operationen auf BSTs

Operation	Zeit
`bstSearch`	$\Theta(h)$
`bstSucc`	$\Theta(h)$
`bstMin`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$

- Alle Operationen sind linear in der Höhe h des BSTs.
- Die Höhe ist $\log_2(n)$, wenn der Baum nicht zu „unbalanciert" ist.

Komplexität der Operationen auf BSTs

Operation	Zeit
`bstSearch`	$\Theta(h)$
`bstSucc`	$\Theta(h)$
`bstMin`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$

- 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 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.

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 Einfügungsordnungen hat die gleiche Wahrscheinlichkeit.

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 Einfügungsordnungen hat die gleiche Wahrscheinlichkeit.

Theorem (ohne Beweis)

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

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 Einfügungsordnungen hat die gleiche Wahrscheinlichkeit.

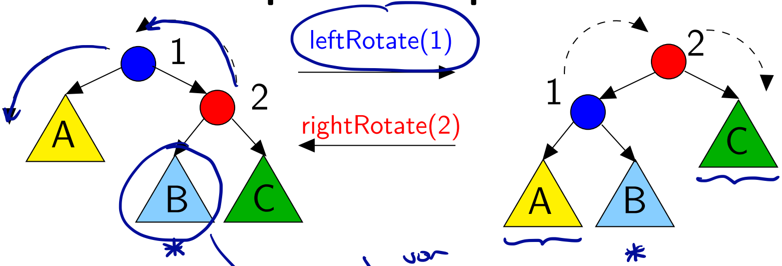
Theorem (ohne Beweis)

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

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

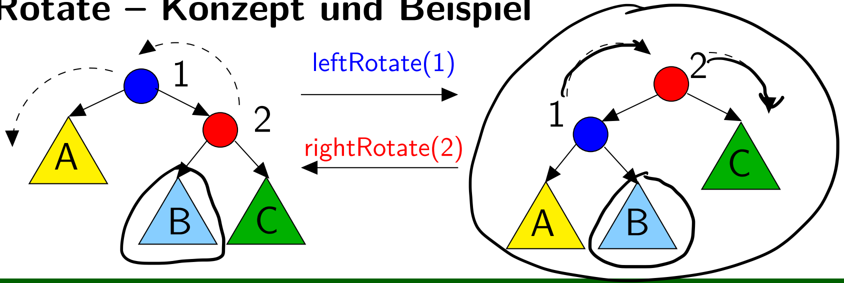
Übersicht

leftRotate – Konzept und Beispiel

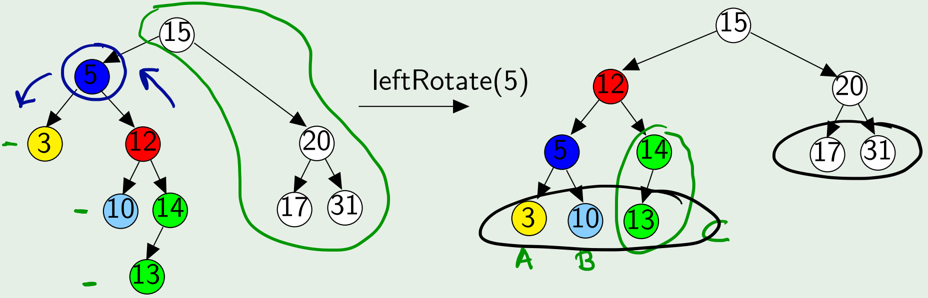


≥ Schlüssel von blau (1)

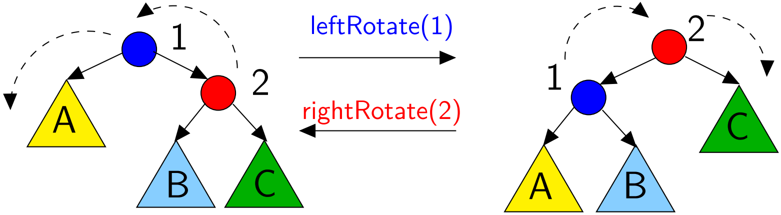
leftRotate – Konzept und Beispiel



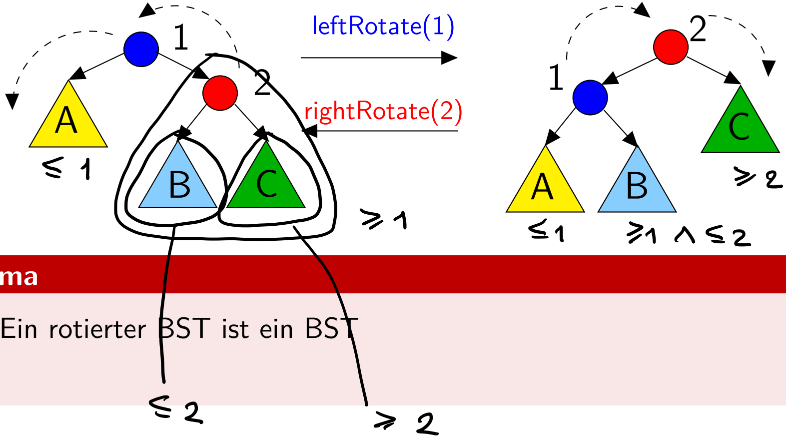
Beispiel



Rotationen: Eigenschaften und Komplexität



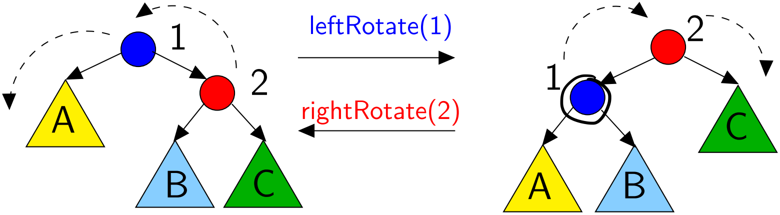
Rotationen: Eigenschaften und Komplexität



Lemma

- Ein rotierter BST ist ein BST

Rotationen: Eigenschaften und Komplexität



ino(A) ; 1 ; ino(B) ; 2 ; ino(C) = ino(A) ; 1 ; ino(B) ; 2 ; ino(C)

Lemma

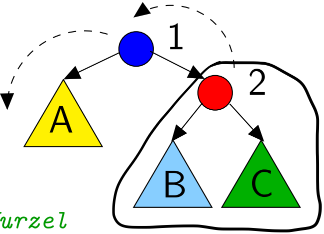
- Ein rotierter BST ist ein BST
- Die Inorder-Traversierung beider Bäume bleibt unverändert.

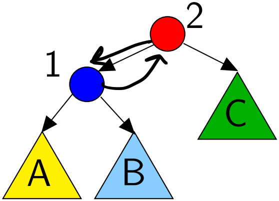
Zeitkomplexität

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

leftRotate – Algorithmus

```
void leftRotate(Tree t, Node node1) { // analog: rightRotate()
  Node node2 = node1.right;
  // Baum B verschieben
  node1.right = node2.left;
  if (node1.right) {
    node1.right.parent = node1;
  }
  // node2 wieder einhängen
  node2.parent = node1.parent;
  if (!node1.parent) { // node1 war die Wurzel
    t.root = node2;
  } else if (node1 == node1.parent.left) { // war linkes Kind
    node2.parent.left = node2;
  } else { // war rechtes Kind
    node2.parent.right = node2;
  }
  // node1 einhängen
  node2.left = node1;
  node1.parent = node2;
}
```





Rotationen – AVL-Baum

An welchen Knoten müssen die Rotationen durchgeführt werden?

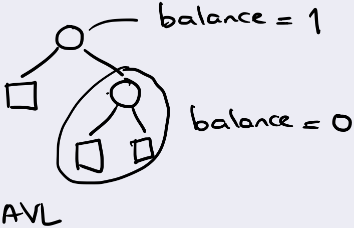
Rotationen – AVL-Baum

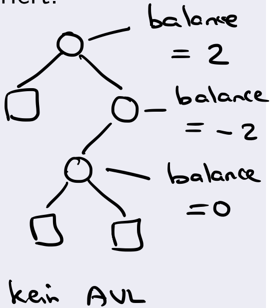
Adelson-Velsky
Landis 1962

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.





balance (x) = Höhe rechter Teilbaum (x) − Höhe linker Teilbaum (x)

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.

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.

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.

löschen
hinzufügen

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 $\underline{\Theta(h)}$ möglich! $h \in \Theta(\log_2(n))$

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.

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.

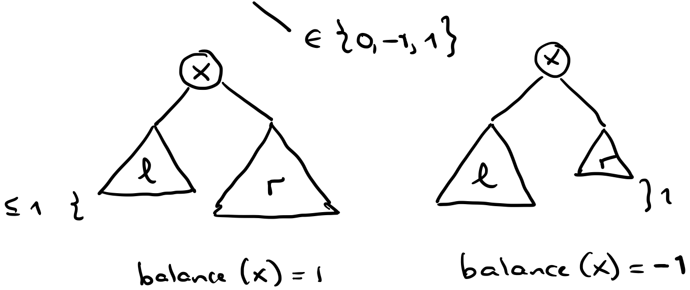
- Eine andere Möglichkeit, um Bäume zu balancieren, sind Rot-Schwarz-Bäume (nächste Vorlesung).

AVL-Bäume: Balancieren nach Einfügen

AVL-Bäume: Balancieren nach Einfügen

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

$$\left|\underbrace{\textit{rechte Teilbaumhöhe} - \textit{linke Teilbaumhöhe}}_{\textit{balance}(x)}\right| \leqslant 1\,.$$



AVL-Bäume: Balancieren nach Einfügen

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

$$\left|\underbrace{\mathit{rechte\ Teilbaumhöhe} - \mathit{linke\ Teilbaumhöhe}}_{\mathit{balance}(x)}\right| \leqslant 1\,.$$

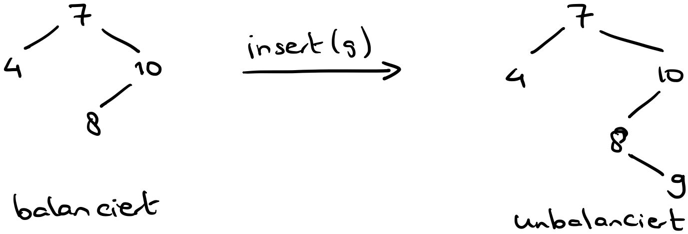
- Wir fügen einen neuen Knoten in den Baum ein.

AVL-Bäume: Balancieren nach Einfügen

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

$$\left|\underbrace{\textit{rechte Teilbaumhöhe} - \textit{linke Teilbaumhöhe}}_{\textit{balance}(x)}\right| \leqslant 1\,.$$

- Wir fügen einen neuen Knoten in den Baum ein.
- Dadurch kann der Baum unbalanciert werden.



AVL-Bäume: Balancieren nach Einfügen

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

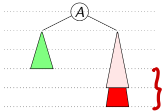
$$\left| \underbrace{\textit{rechte Teilbaumhöhe} - \textit{linke Teilbaumhöhe}}_{\textit{balance}(x)} \right| \leqslant 1\,.$$

- 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.

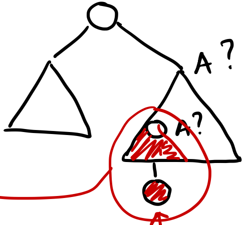
AVL-Bäume: Balancieren nach Einfügen

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



$\}=2$ = balance (A)

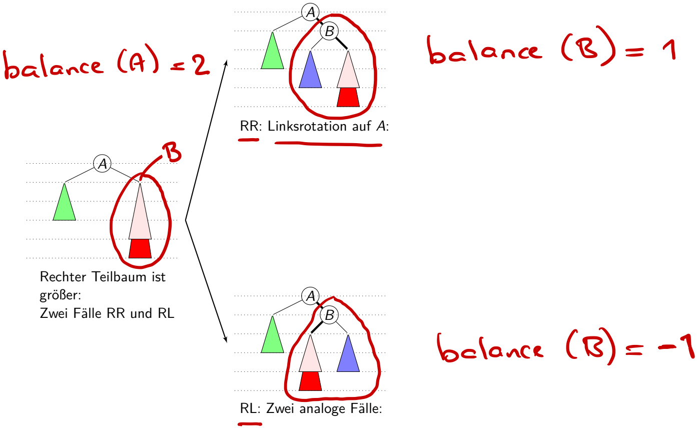
Rechter Teilbaum ist größer:
Zwei Fälle RR und RL



neu eingefügt

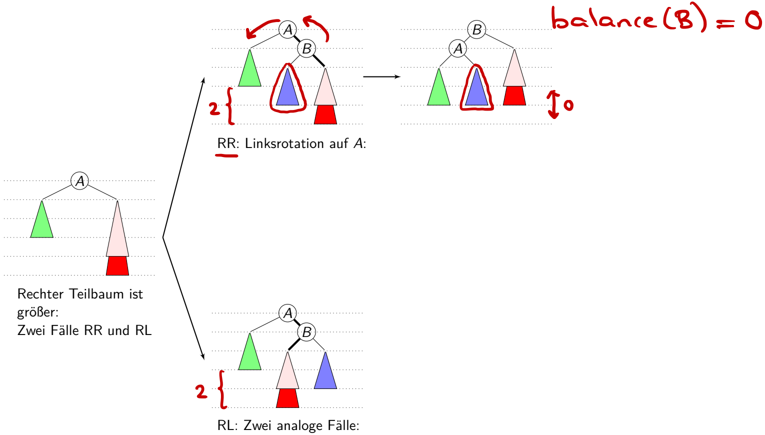
AVL-Bäume: Balancieren nach Einfügen

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



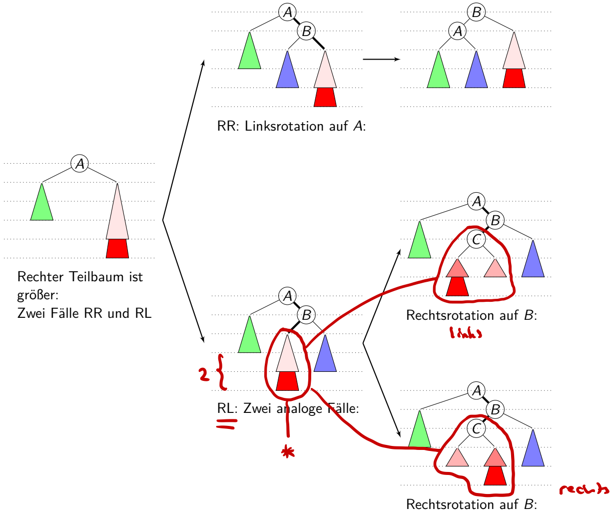
AVL-Bäume: Balancieren nach Einfügen

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



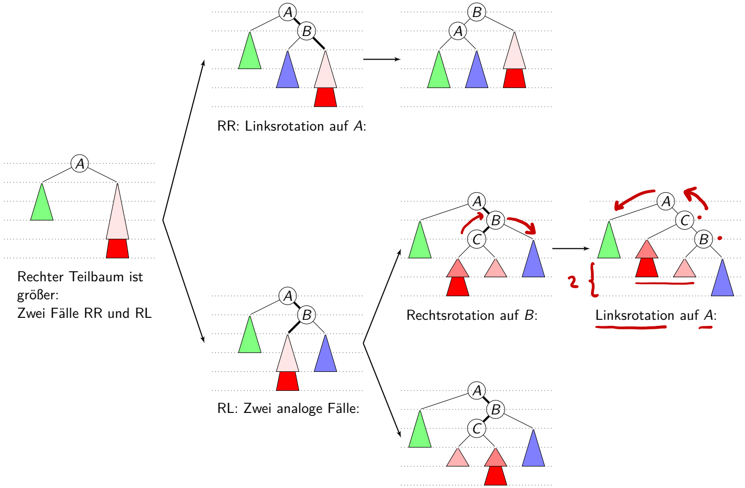
AVL-Bäume: Balancieren nach Einfügen

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



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* $= \pm 2$).



Rechter Teilbaum ist größer:
Zwei Fälle RR und RL

RR: Linksrotation auf A:

RL: Zwei analoge Fälle:

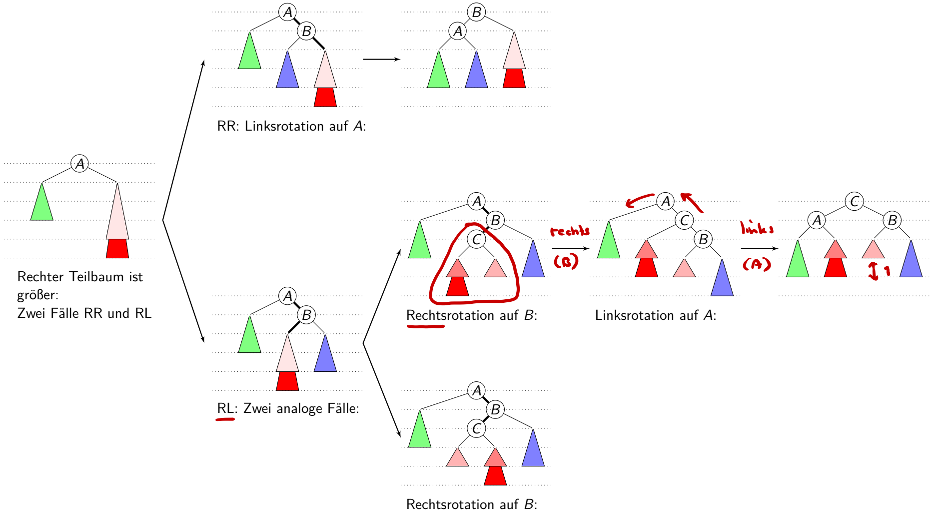
Rechtsrotation auf B:

Linksrotation auf A:

Rechtsrotation auf B:

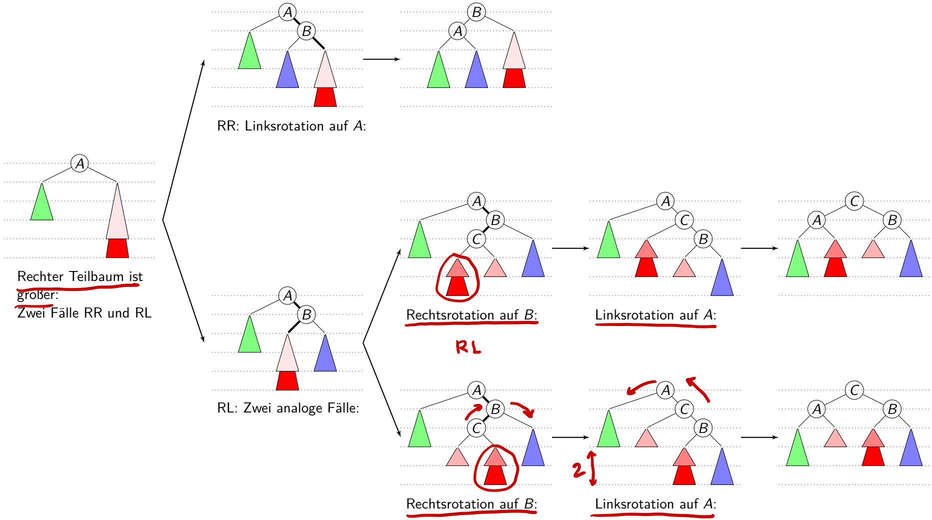
AVL-Bäume: Balancieren nach Einfügen

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



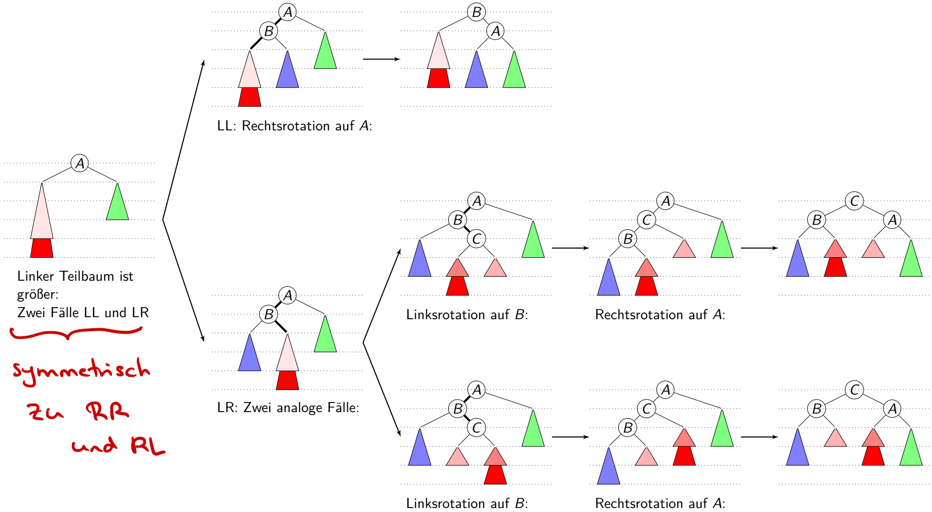
AVL-Bäume: Balancieren nach Einfügen

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



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* $= \pm 2$).



symmetrisch zu RR und RL

AVL-Bäume: Balancieren nach Einfügen

```
void AVLIns(Tree t, Node node) {
  bstIns(t,node);
  //Node deepestUnbalancedNode(Tree t, Node node)
  //gibt null zurück wenn t balanciert ist
  //und den tiefsten unbalancierten Knoten in t sonst
  //(der Parameter node wird zur effizienten Implementierung
  //verwendet)
  Node A = deepestUnbalancedNode(t,node);
  if (A != null) balance(t, A);
}
```

→ Details hier nicht betrachtet

AVL-Bäume: Balancieren nach Einfügen

```
void balance(Tree t, Node A){
  //A ist tiefster unbalancierter Knoten in t
  if (height(A.left) > height(A.right)) {
    if (height(A.left.left) >= height(A.left.right)) { //LL
      rightRotate(t,A);
    } else { //LR
      leftRotate(A.left); rightRotate(A);
    }
  } else {
    if (height(A.right.right) >= height(A.right.left)) { //RR
      leftRotate(t,A);
    } else { //RL
      rightRotate(A.right); leftRotate(A);
    }
  }
}
```

2 Rotationen

AVL-Bäume: Balancieren nach Löschen

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.

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.

AVL-Bäume: Balancieren nach Löschen

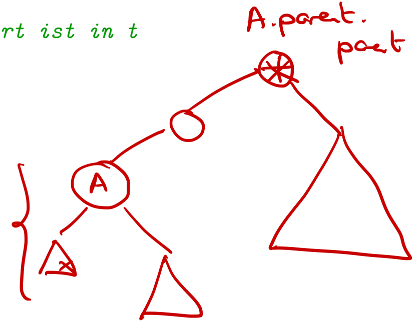
- Baumhöhe von A nach der Rotation ist wieder die gleiche wie vor dem Einfügen des neuen Knotens.
- Das heißt, nach dem Balancieren von A ist der gesamte Baum wieder balanciert.
- Die zweite Operation, die Unbalanciertheit verursachen kann, ist das Löschen eines Knotens.
- Die Balancierung des tiefsten 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)!

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

```
void AVLDel(Tree t, Node node) {
  bstDel(t,node);
  //Node deepestUnbalancedNode(Tree t, Node node)
  //gibt null zurück wenn t balanciert ist
  //und den tiefsten unbalancierten Knoten in t sonst
  //(der Parameter node wird zur effizienten Implementierung
  //verwendet)
  Node A = deepestUnbalancedNode(t,node);
  while (A != null) {
    //bool balanced(Tree t, Node A)
    //gibt true zurück wenn A balanciert ist in t
    //und false sonst
    if (!balanced(t, A)) {
      balance(t, A);
      A = A.parent.parent;
    } else {
      A = A.parent;
    }
  }
}
```



Komplexität der Operationen auf AVL Bäume

visualgo.net

Operation	Zeit
`bstSearch`	$\Theta(h)$
`bstSucc`	$\Theta(h)$
`bstMin`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$

- Da AVL Bäume balanciert sind, gilt $h = \log_2(n)$

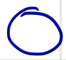
Nachtrag

$Fib(0) = 0 \quad Fib(1) = 1$

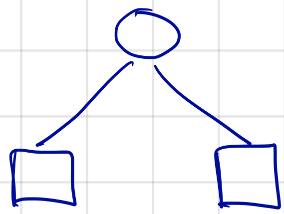
$Fib(i+2) = Fib(i+1) + Fib(i)$

Satz: ein AVL Baum mit Höhe h hat mindestens $Fib(h+2)$ Blätter.

Bew: (skizze) mit vollständiger Induktion.

$h = 0$:  Anzahl Blätter ist $Fib(0+2) = 1$

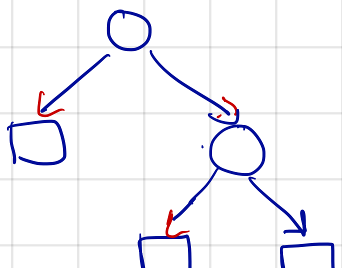
$h = 1$:



AVL Baum mit Höhe 1 und mindest Anzahl Blätter.

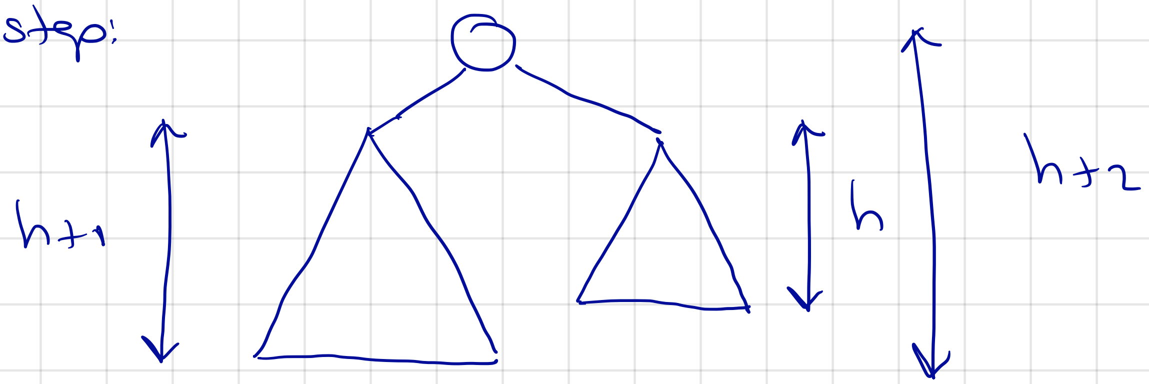
$Fib(1+2) = 2$ Blätter

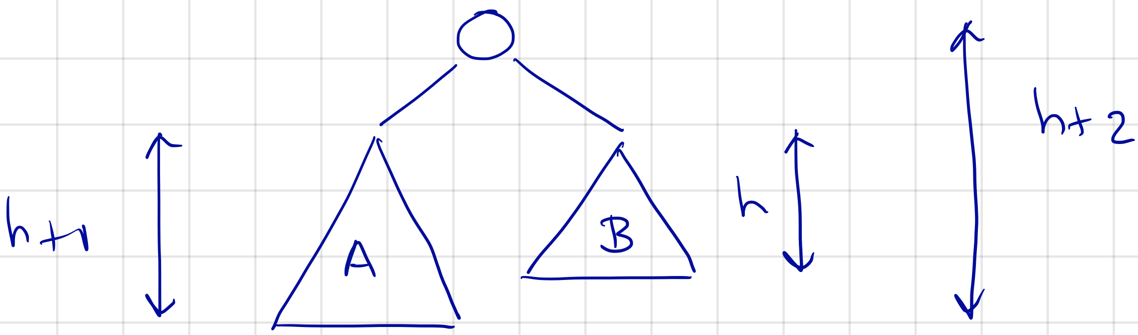
$h = 2$:



AVL Baum mit Höhe 2 und mindest Anzahl Blätter: $Fib(2+2) = 3$ Blätter

Ind. step:





der Teilbaum A hat mind. Fib$(h+1+2)$ Blätter

" " B " " Fib$(h+2)$ Blätter

Also: der ganze AVL-Baum hat mindestens

$$\mathrm{Fib}(h+3) + \mathrm{Fib}(h+2) = \mathrm{Fib}(h+2+2)$$

Blätter $\boxtimes$

Nächste Vorlesung

Nächste Vorlesung

Freittag 1. Juni, 13:15 (Hörsaal H01). Bis dann!