Datenstrukturen und Algorithmen

Vorlesung 9: Binäre Suchbäume (K12)

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

15. Mai 2014

RWTH AACHEN UNIVERSITY

Übersicht

Übersicht

ADTs für dynamische Multimengen: Arrays und Listen

Eine Multimenge ist ähnlich zu einer Menge, kann aber Elemente mehrfach enthalten.

Beispiel (Vor- und Nachteile von Arrays)

\+ Direkter Zugriff über Indices ⇒ Binäre Suche für sortierte Arrays in $\mathcal{O}(\log n)$ möglich

– Statische Größe ⇒ `insert/delete` problematisch

Beispiel (Vor- und Nachteile von Listen)

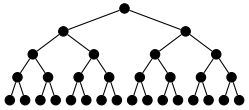
\+ $\mathcal{O}(1)$ für `insert/delete` (auf Elemente in doppelt verketteter Implementierung)

– Kein Direktzugriff über Indices ⇒ Suche in $\mathcal{O}(n)$ sogar für sortierte Listen

ADTs für dynamische Multimengen: Binärbäume

Definition (Binärbaum)

Ein Binärbaum (binary tree) ist ein Baum, in dem jeder Knoten höchstens zwei ausgehenden Kanten hat, die geordnet sind.



Definition

Ein Binärbaum ist (höhen-)balanciert, wenn sich für jeden Knoten die Höhen der Teilbäume der beiden Kinder höchstens um eine bestimmte Differenz unterscheiden.

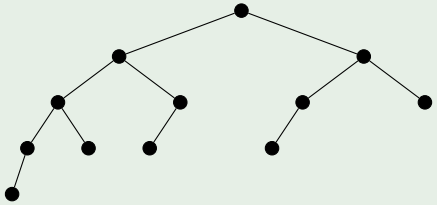
Beispiel (Vorteile balancierter Binärbäume)

In einem balancierten Binärbaum mit n Knoten kann jedes Element von der Wurzel aus in $\mathcal{O}(\log n)$ Schritten erreicht werden (statt $\mathcal{O}(n)$ bei Listen).

Balancierte Binärbäume

Beispiel (Fibonacci-Baum)

$Fib(0)$:	Leerer Baum
$Fib(1)$:	Nur eine Wurzel
$Fib(h)$ für $h \geqslant 2$:	Eine Wurzel, linkes Kind $Fib(h-1)$, rechtes Kind $Fib(h-2)$

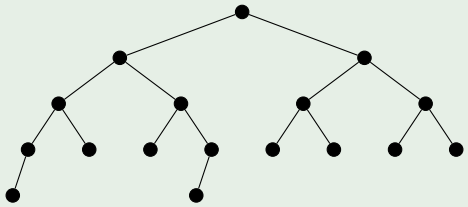


Vollständige Binärbäume

Definition

Ein vollständig balancierter (vollständiger) Binärbaum ist ein balancierter Binärbaum in dem sich die Tiefen zweier beliebiger Blätter höchstens um 1 unterscheiden.

Beispiel

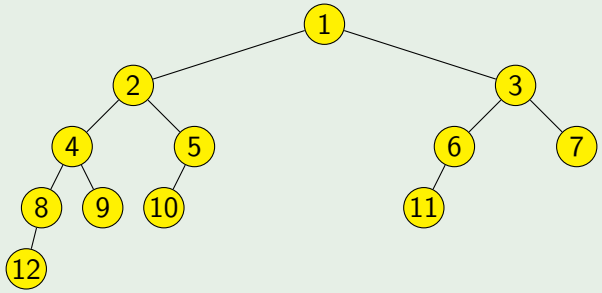


Alle Ebenen, bis auf die letzte, sind vollständig gefüllt.

Binärbäume mit Schlüsseln

Beispiel

Wenn wir die Knoten von Binärbäumen mit Schlüsseln (Werten) versehen, erhalten wir Datentypen für Multimengen.



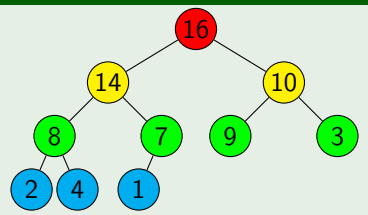
Die Balanciertheit des Baumes und die Anordnung der Schlüsseln im Baum bestimmen, wie einfach oder schwer es ist, bestimmte Operationen wie z.B. Suchen, Einfügen, Löschen oder sortiert Ausgeben ausgeführt werden können.

ADTs für dynamische Multimengen: Heaps

Definition (Heap)

Ein Heap ist ein vollständiger Binärbaum mit nach links geordneten Blättern, so dass der Schüssel jedes Vaterknotens größer oder gleich der Schlüssel seiner Kinder ist.

Beispiel (Vorteile von Heaps)



Verwendung finden Heaps vor allem dort, wo es darauf ankommt, schnell ein Element mit größtem Schlüssel aus dem Heap zu entnehmen (HIFO-Prinzip).

ADTs für dynamische Multimengen: Vergleich

Komplexität einiger Operationen

Operation	Array	Liste	sortiertes Array	sortierte Liste	Heap	Nett wäre...
`findMax`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
`findMin`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`find`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`insert`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`delete`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`print Sorted`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

Annahmen:

- Array: statisch, `insert`/`delete` benötigt Kopieren
- Liste: doppelt verkettete Implementierung, Zeiger auf Ende (`tail`)
- Hier: `insert`/`delete` wird auf ein Element (nicht auf einen Schüssel) angewandt.

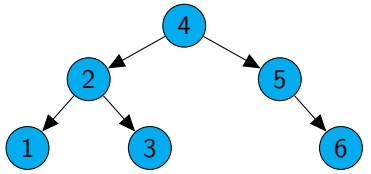
Übersicht

Binäre Suchbäume

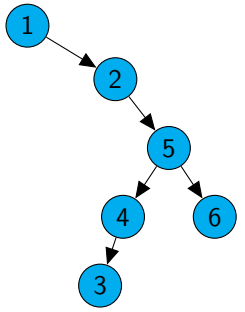
Binärer Suchbaum

Ein binärer Suchbaum (binary search tree BST) ist ein Binärbaum, der Elemente mit Schlüsseln als Knoten enthält, wobei der Schlüssel jedes Knotens

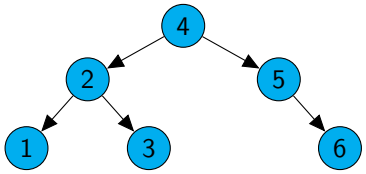
- mindestens so groß ist, wie jeder Schlüssel im linken Teilbaum und
- höchstens so groß ist, wie jeder Schlüssel im rechten Teilbaum.

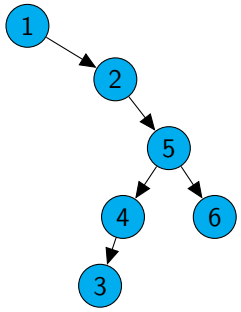


Zwei binäre Suchbäume, die jeweils die Schlüssel 1, 2, 3, 4, 5, 6 enthalten.

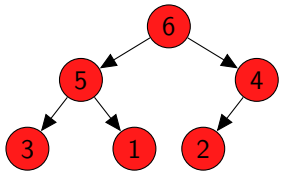


Binäre Suchbäume – Unterschiede zu Heaps





Zwei binäre Suchbäume, die jeweils die Schlüssel 1, 2, 3, 4, 5, 6 enthalten.

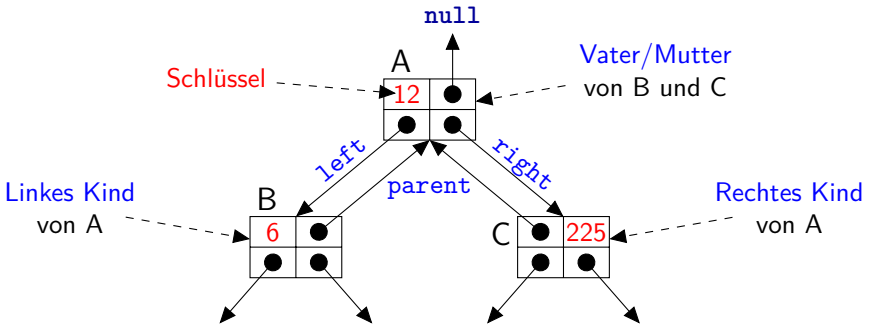


Ein Heap mit den gleichen Schlüsseln.

Binäre Suchbäume – Implementierung

Knoten in einem binären Suchbaum bestehen aus vier Feldern:

- Einem Schlüssel – dem „Wert“ des Knotens,
- Zeiger auf den linken und den rechten Teilbaum (evtl. *null*), sowie
- einem Zeiger auf den Vater-/Mutterknoten (bei der Wurzel leer)
  ⇒ doppelt verkettete Implementierung.



Binäre Suchbäume – Implementierung

Beispiel

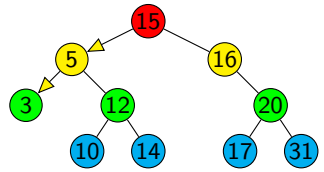
```
1 class Node {
2   int key;
3   Node left, right;
4   Node parent;
5   // ... evtl. eigene Datenfelder
6 };

8 class Tree {
9   Node root;
10 };
```


Abfragen im BST: Minimum

Problem

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

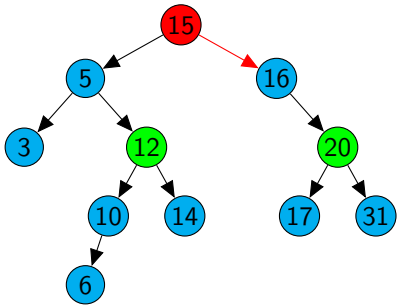
```
1 Node bstMin(Node root) { // root != null
2   while (root.left != null) {
3     root = root.left;
4   }
5   return root;
6 }
```



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

Suche nach Schlüssel $k = 18$ im BST

```
1  Node bstSearch(Node root, int k)
       {
2    while (root != null) {
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: $\Theta(h)$.

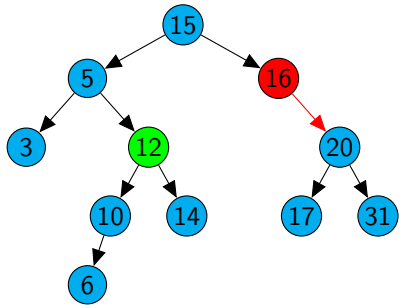
- Für einen kettenartigen Baum mit n Knoten ergibt das $\Theta(n)$.
- Ist der BST balanciert, erhält man $\Theta(\log n)$.

Funktioniert dieses Suchverfahren auch bei Heaps? Nein.

Binäre Suchbäume Binäre Suchbäume

Suche nach Schlüssel $k = 18$ im BST

1 Node bstSearch(Node root, int k)
{
2 while (root != null) {
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 balanciert, erhält man $\Theta(\log n)$.

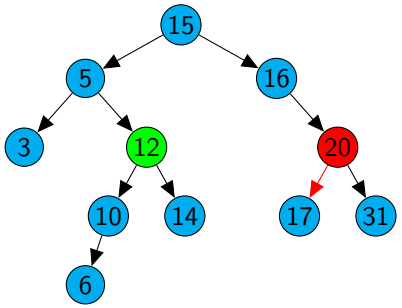
Funktioniert dieses Suchverfahren auch bei Heaps? Nein.

Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 17/40

Binäre Suchbäume Binäre Suchbäume

Suche nach Schlüssel $k = 18$ im BST

1 Node bstSearch(Node root, int k)
{
2 while (root != null) {
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 balanciert, erhält man $\Theta(\log n)$.

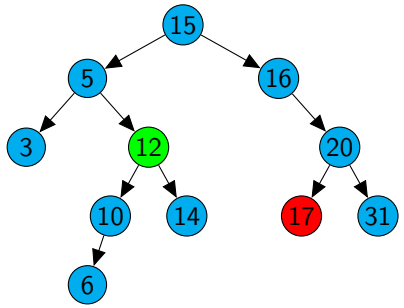
Funktioniert dieses Suchverfahren auch bei Heaps? Nein.

Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 17/40

Binäre Suchbäume Binäre Suchbäume

Suche nach Schlüssel $k = 18$ im BST

1 Node bstSearch(Node root, int k)
{
2 while (root != null) {
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 balanciert, erhält man $\Theta(\log n)$.

Funktioniert dieses Suchverfahren auch bei Heaps? Nein.

Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 17/40

Einfügen eines Knotens mit Schlüssel k – Strategie

Einfügen

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

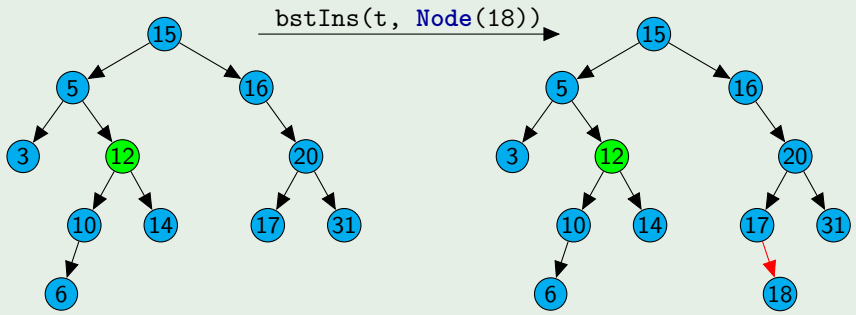
Suche einen geeigneten, freien Platz:
Wie bei der regulären Suche, außer dass, selbst bei gefundenem Schlüssel, weiter abgestiegen wird, bis ein Knoten ohne entsprechendes Kind erreicht ist.

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

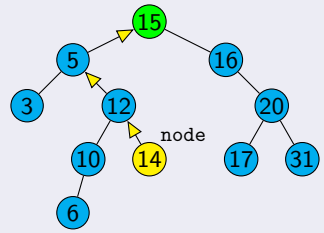
```
1  void bstIns(Tree t, Node node) { // Fuege node in den Baum t
     ein
2    // Suche freien Platz
3    Node root = t.root, parent = null;
4    while (root != null) {
5      parent = root;
6      if (node.key < root.key) {
7        root = root.left;
8      } else {
9        root = root.right;
10     }
11   } // Einfuegen
12   node.parent = parent; node.left = null; node.right = null;
13   if (parent == null) { // 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

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

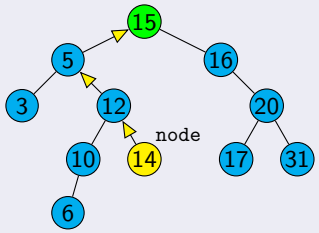


Abfragen im BST: Nachfolger

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



Binäre Suchbäume Binäre Suchbäume

Abfragen im BST: Nachfolger

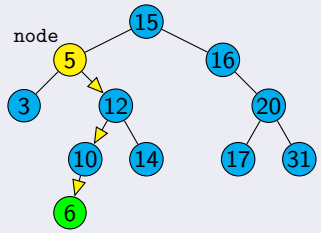
Problem

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.



Prof. Dr. Erika Ábrahám Datenstrukturen und Algorithmen 21/40

Binäre Suchbäume Binäre Suchbäume

Abfragen im BST: Nachfolger

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`.

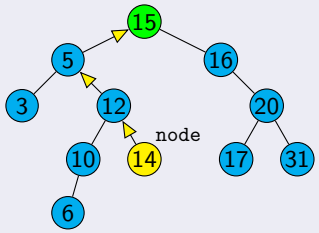
Lösung

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.



Binäre Suchbäume Binäre Suchbäume

Abfragen im BST: Nachfolger

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`.

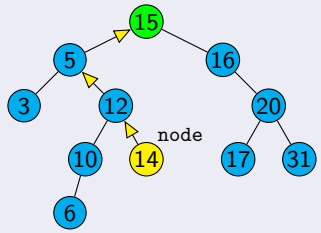
Lösung

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

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`.

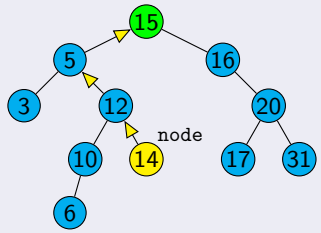
Lösung

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

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

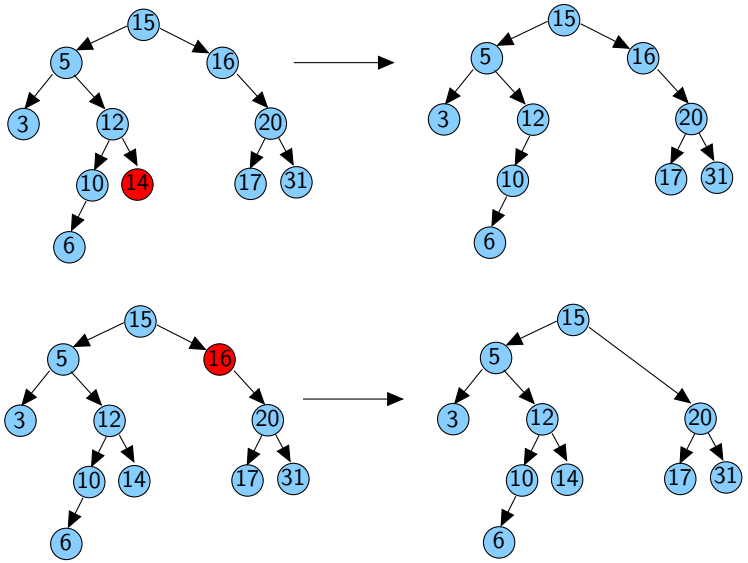
Das Ersetzen eines Teilbaums hat die Zeitkomplexität $\Theta(1)$.

Austauschen von Knoten im BST

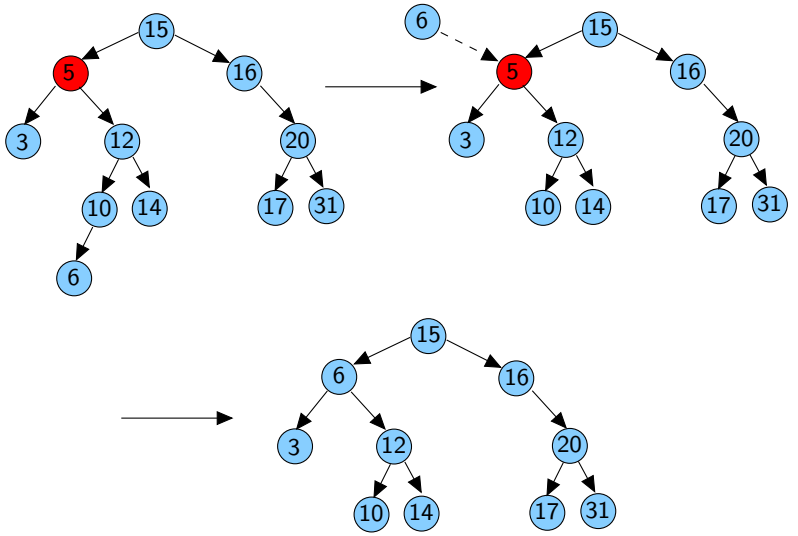
```
1 // Tauscht den Knoten oldNode gegen newNode aus;
2 // die Kinder von oldNode sind weiter im BST!
3 void bstSwap(Tree t, Node oldNode, Node newNode) {
4   // übernimm linken Teilbaum
5   newNode.left = oldNode.left; // auch möglich: swap()
6   if (newNode.left != null) {
7     newNode.left.parent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


Sortieren im BST 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.

Zeitkomplexität

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

Komplexität der Operationen auf BSTs

Operation	Zeit
`bstMin/bstMax`	$\Theta(h)$
`bstSearch`	$\Theta(h)$
`bstSucc/bstPred`	$\Theta(h)$
`bstIns`	$\Theta(h)$
`bstDel`	$\Theta(h)$
`bstSort`	$\Theta(n)$

- Sortieren ist in linearer Zeit.
- Alle anderen Operationen sind linear in der Höhe h des BSTs.
- Die Höhe ist in $\mathcal{O}(\log n)$, wenn der Baum balanciert ist.
- Man kann einen unbalancierten binären Baum mittels Rotationen wieder balancieren.

Zufällig erzeugte binäre Suchbäume

Zufällig erzeugte 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 $O(\log n)$.

Fazit: Im Schnitt verhält sich ein binärer Suchbaum wie ein (fast) balancierte Suchbaum.

Übersicht

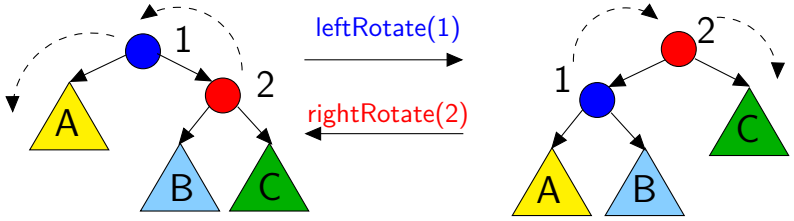
Rotationen – AVL-Baum

AVL-Baum

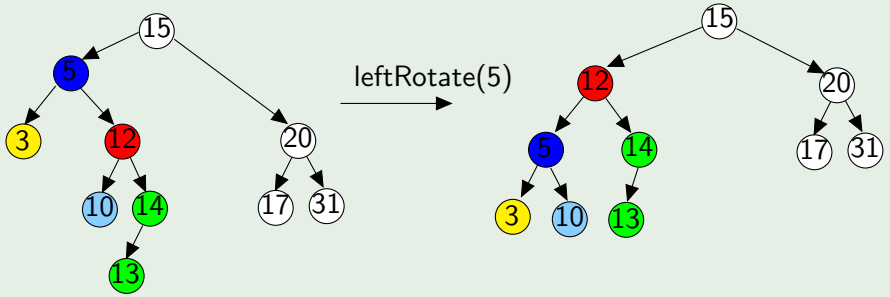
- Ein AVL-Baum ist ein balancierter BST, bei dem für jeden Knoten die Höhe der beiden Teilbäume höchstens um 1 differiert.
- Dazu wird (in einem zusätzlichem Datenfeld) an jedem Knoten über die Höhe dieses Unterbaums Buch geführt.
- Nach jeder (kritischen) Operation wird die Balance wiederhergestellt. Dies ist in $\Theta(h)$ möglich!
- Dadurch bleibt stets $h = \Theta(\log n)$ und $\Theta(\log n)$ kann für die Operationen auf dem BST garantiert werden.
- Der AVL-Baum ist benannt nach den sowjetischen Mathematikern Georgi Maximowitsch Adelson-Velski und Jewgeni Michailowitsch Landis, die die Datenstruktur im Jahr 1962 vorstellten.

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

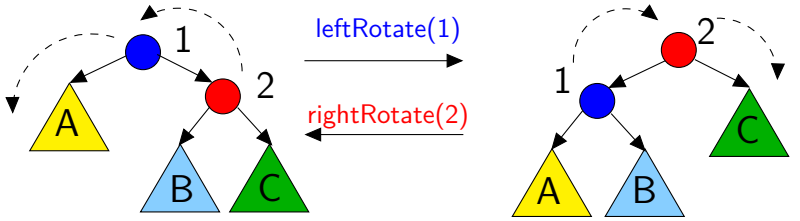
leftRotate – Konzept und Beispiel



Beispiel



Rotationen: Eigenschaften und Komplexität



Lemma

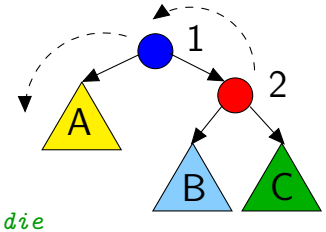
- Ein rotierter BST ist ein BST
- Die Inorder-Traversierung beider Bäume bleibt unverändert.

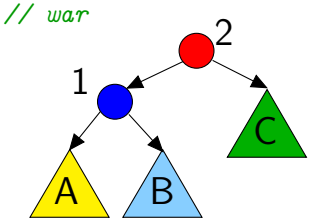
Zeitkomplexität

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

leftRotate – Algorithmus (rightRotate() ist analog)

```
1  void leftRotate(Tree t, Node node1) {
2    Node node2 = node1.right;
3    // Baum B verschieben
4    node1.right = node2.left;
5    if (node1.right != null) {
6      node1.right.parent = node1;
7    }
8    // node2 wieder einhängen
9    node2.parent = node1.parent;
10   if (node1.parent == null) { // 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 }
```





Übersicht

AVL-Bäume: Balancieren nach Einfügen

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

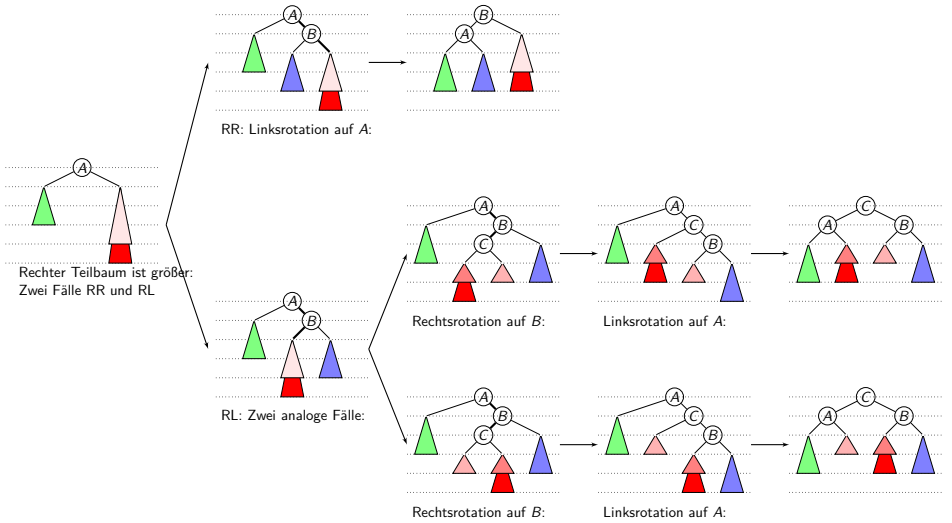
$$\underbrace{|\textit{rechte Teilbaumhöhe} - \textit{linke Teilbaumhöhe}|}_{\textit{balance}(x)} \leqslant 1\,.$$

- 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.

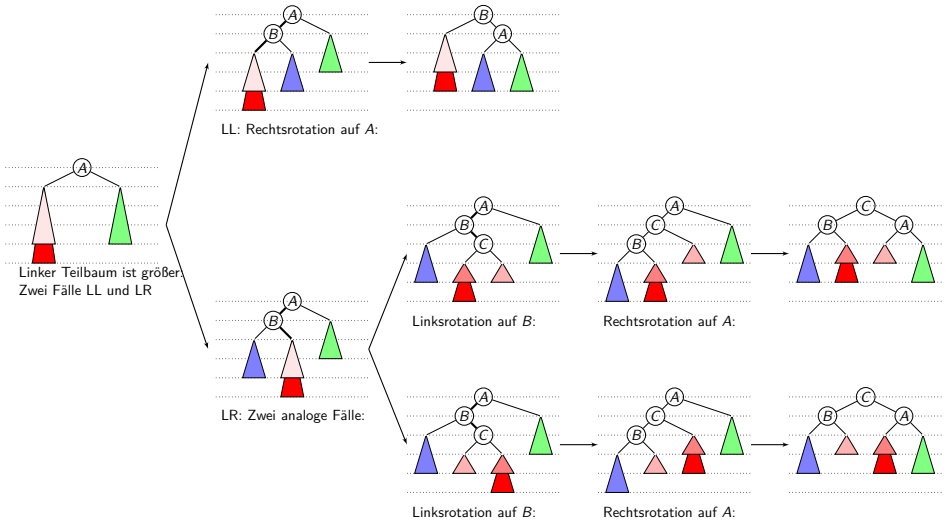
AVL-Bäume: Balancieren nach Einfügen

Sei A der tiefste unbalancierte Knoten auf dem Pfad von der Wurzel zum neu eingefügten Knoten (unbalanciert: *linke Teilbaumhöhe* − *rechte Teilbaumhöhe* = ±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 Löschen

- Baumhöhe von A nach der Rotation ist wieder die gleiche wir vor dem Einfügen des neuen Knotens.
- Das heißt, nach dem Balancieren von A ist der gesamte Baum wieder balanciert.
- Der zweite Operation, der 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.