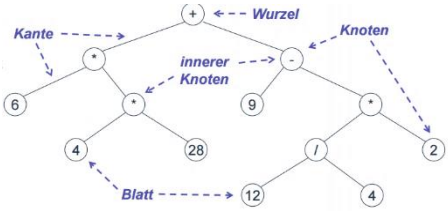
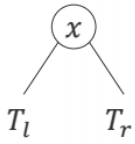
Bäume

Bäume



- Listen: jedes Element hat einen Nachfolger
- Baum: jedes Element hat mehrere Nachfolger
- weitere Beispiele für Baumstrukturen:
  - Gliederung eines Buches in Kapitel, Abschnitte, Unterabschnitte
  - Klammerstrukturen
- Höhe eines Knotens:
  - Die Höhe h eines Knotens entspricht der Anzahl der Kanten des längsten Pfades zu einem von diesem Knoten aus erreichbaren Blatt (= Knoten ohne Nachfolger).
- Die Höhe eines Baumes ist die Höhe der Wurzel
- Bäume enthalten keine Zyklen.
- Verzweigungsgrad = Zahl der möglichen Nachfolger pro Element
- Ein Baum mit n Knoten hat n-1 Kanten

Binärbäume



- Jeder Knoten hat nur zwei Nachfolger (Verzweigungsgrad: 2).
- Definition:
  - Der leere Baum ist ein Binärbaum. (Bezeichnung: ⋄; graphisch: ■)
  - Wenn x ein Knoten ist und T_l u. T_r Binärbäume sind, dann ist auch das Tripel (T_l, x, T_r) ein Binärbaum T.
- x heißt Wurzel, T_l linker Teilbaum, T_r rechter Teilbaum von T.
- Die Wurzeln von T_l und T_r heißen Kinder/Söhne von x, x ist ihr Elternknoten/Vaterknoten.
- Ein Knoten, dessen beide Kinder leere Bäume sind, heißt Blatt.
  - Teilbaum in linearer Notation: ((⋄, 12,⋄ ,/, ⋄, 4,⋄) ,∗, ⋄, 2,⋄)
- Die maximale Höhe eines Binärbaums mit n Knoten ist $n - 1$.
  - Dieser Fall tritt auf, wenn der Baum zu einer Liste entartet.
- Die minimale Höhe eines Binärbaums mit n Knoten ist $\lfloor \log_2 n \rfloor$.
- Falls alle inneren Knoten zwei Kinder haben, so hat ein Binärbaum mit n + 1 Blättern genau n innere Knoten.

Abstrakter Datentyp *Bintree*

adt $BinTree$
sorts $BinTree, E, Boolean$
ops

$create$:		$\rightarrow BinTree$
$maketree$:	$Bintree \times E \times BinTree$	$\rightarrow BinTree$
$value$:	$BinTree$	$\rightarrow E$
$left, right$:	$BinTree$	$\rightarrow BinTree$
$leaf$	E	$\rightarrow BinTree$
$isEmpty$:	$BinTree$	$\rightarrow Boolean$

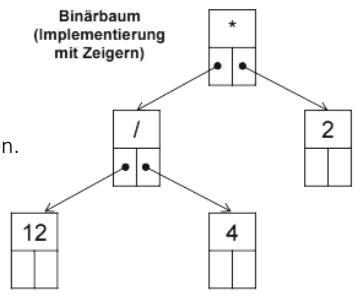
axs

$left(maketree(l,e,r)) = l$
$right(maketree(l,e,r)) = r$
$value(maketree(l,e,r)) = e$
$leaf(v) = maketree(create, v, create)$
$isEmpty(create) = true$
$isEmpty(maketree(l,e,r)) = false$
…

end $BinTree$

Anwendungsbeispiel

- Hilfsfunktion zum Erzeugen eines Blatts:
  - leaf(v) = maketree(create,v,create)



Implementierung des ADTs *BinTree* mit Referenzen

- Implementierung ist ähnlich der von einfach oder doppelt verketteten Listen.
- Jeder Knoten besitzt zwei Referenzen auf die beiden Teilbäume.

Implementierung des ADTs *B*inTree mit Referenzen

```
class Entry<E> {
        E element;
        Entry left, right;
        public Entry(Entry<E> l, E x, Entry<E> r) {
                left = l;
                element = x;
                right = r;
        }
        …
}
class BinTree<E> {
        Entry<E> root;
        …
        // Konstruktor und Methoden
        // der Klasse BinTree
}
```

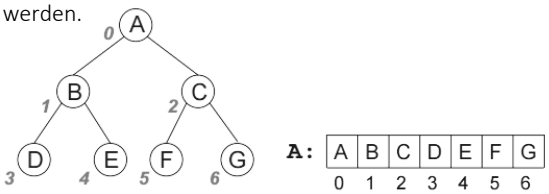
Implementierung des ADTs *B*inTree mit Array-Indizes als Zeigern

Implementierung mit mehrdimensionalem Array und Indizes, ohne explizite Verweise

A	B	C	D	E	F	G	
0	1	2	3	4	5	6	zugeordneter Index
1	3	-1	-1	5	-1	-1	Index linkes Kind
2	4	-1	-1	6	-1	-1	Index rechtes Kind

Implementierung des ADTs *B*inTree durch Einbettung in Array

Betrachte vollständigen Binärbaum der Höhe h: Strategie: durch die Position in einem Feld kann die Position der Nachfolger berechnet werden.



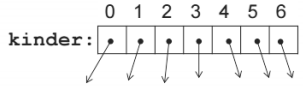
- Baum wird eingebettet in Array A der Länge $2^{h+1} - 1$ (hier: = 7)
  - speichere Wurzel in A[0]
  - linker Nachfolger von Knoten ii steht in A[2*i+1]
  - rechter Nachfolger von Knoten ii steht in A[2*i+2]
  - Vorgänger eines Knotens k steht in A[(k-1)/2]

(Allgemeine) Bäume

- Anzahl der Kinder eines Knotens ist beliebig.
- Keine „festen Positionen“ für Kinder (wie left, right).
- Rekursive Definition:
  - Der leere Baum ist ein (allgemeiner) Baum.
  - Wenn x ein Knoten ist und $T1, \dots, Tk$ Bäume sind, dann ist auch das $(k + 1)$-Tupel $(x, T1, \dots, Tk)$ ein Baum.
- Begriffe der Binärbäume lassen sich übertragen (bis auf die positionsbezogenen, z. B. linker Teilbaum)
- Grad eines Knotens: Anzahl seiner Kinder (für Knoten x: k)
- Grad eines Baumes: maximaler Grad aller Knoten im Baum)

Implementierung allgemeiner Bäume mit Hilfe von ArrayLists

Jeder Knoten speichert eine ArrayList von Referenzen auf seine Kinder.



`kinder.get(i)` liefert Referenz auf das i-te Kind.

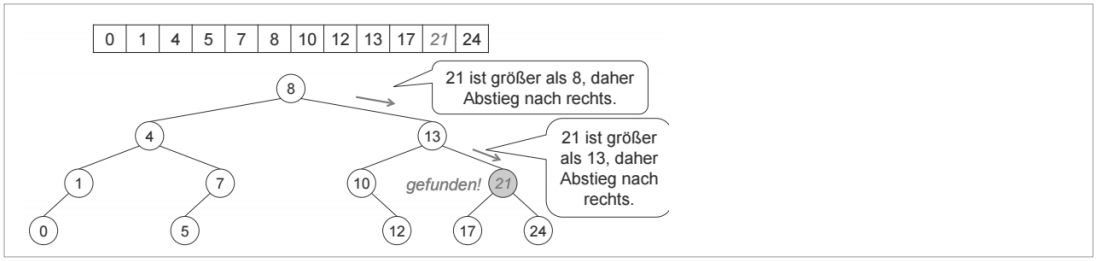
Implementierung allgemeiner Bäume über Binärbäume

- Struktur eines Knotens ist gleich der in einem Binärbaum.
- Aber andere Interpretation der Referenzen:
  - Statt Referenz auf den linken Teilbaum:Referenz auf das am weitesten links stehende Kind.
  - Statt Referenz auf den rechten Teilbaum: Referenz auf das rechte Geschwisterkind.
  - Es ist kein direkter Zugriff auf das i-te Kind möglich.

Suchbäume

Bäume kann man sich als strukturelles Abbild des Teile-und Herrsche-Prinzips vorstellen, wenn man beim Zugriff jeweils pro Knoten nur ein Kind weiterverfolgt.

Eine solche Datenstruktur in Form eines Baums wird Suchbaum genannt. Ein binärer Suchbaum hat max. zwei Kinder je Knoten.



Binäre Suchbäume

- Sei T<E> ein Baum. Eine Knotenmarkierung ist eine Abb. μ: $E \rightarrow D$ für irgendeinen geordneten Wertebereich D.
- Anschaulich:
  - Es wird jedem Knoten E von TTein Wert aus D zugeordnet, der es ermöglicht, die Knotenwerte zu vergleichen.
  - Direkter Vergleich möglich z. B. bei numerischen Werten, Strings
  - Was jedoch z. B. bei Symbolen? Mit μ Ordnungsrelation definieren.
- Ein knotenmarkierter Binärbaum T heißt binärer Suchbaum genau dann, wenn für jeden Teilbaum von T‘ von T, T‘= (Tl,y,Tr) gilt:
  - $\forall x \in Tl$: μ (x) < $\mu(y)$
  - $\forall z \in Tr$: μ (z) > $\mu(y)$

Abstrakter Datentyp SBintree

adt $SBinTree$
sorts $SBinTree, E, Boolean$
ops

$create$: $\rightarrow SBinTree$
$insert$: $SBinTree \times E \rightarrow SBinTree$
$find$: $SBinTree \times E \rightarrow Boolean$
$delete$: $SBinTree \times E \rightarrow SBinTree$

nicht-öffentliche Hilfsfunktion $maketree$ (s. ADT $BinTree$)

Ein binärer Suchbaum enthält keine doppelten Einträge.

axs

$$insert(create, x) = maketree(create, x, create)$$

$$insert(maketree(l, e, r), x) = \begin{cases} maketree(l, e, r) & \text{falls } x = e \\ maketree(insert(l, x), e, r) & \text{falls } x < e \\ maketree(l, e, insert(r, x)) & \text{falls } x > e \end{cases}$$

Einfügen im passenden Teilbaum

$$find(create, x) = false$$

$$find(maketree(l, e, r), x) = \begin{cases} true & \text{falls } x = e \\ find(l, x) & \text{falls } x < e \\ find(r, x) & \text{falls } x > e \end{cases}$$

...

end $SBinTree$

Implementierung von binären Suchbäumen

- hier:
  - Implementierung mit doppelter Verzeigerung
  - Nicht für alle Verwendungen erforderlich, aber vieles (z. B. Iterator) wird leichter.

```
public class SBinTree<E extends Comparable<E>> {

        private class Entry {
                Entry left; // linker Nachfolger
                Entry right; // rechter Nachfolger
                Entry parent; // Elternknoten

                E value; // Nutzdaten

                public Entry(E val, Entry parent) {
                        left = null;
                        right = null;
                        value = val;
                        this.parent = parent;
                }
        }
        // Wurzel des Binaerbaums
        Entry root;

        public SBinTree() {
                root = null;
        }
 ...
```

Suche im binären Suchbaum: rekursive Implementierung

äußere Methode contains

```
public boolean contains(E val) {
       return containsRec(val, root);
}
```

innere Methode containsRec

```
private boolean containsRec(E val, Entry node) {
        // Baum leer?
        if (node == null) return false;
        int result = val.compareTo(node.value);
        if (result == 0) return true; //gefunden
        if (result < 0) {
                // Wert des Suchbaumknotens node "groesser" als Wert val,
               // im linken Teilbaum weitersuchen
               return containsRec(val, node.left);
        } else {
               // Wert des Suchbaumknotens node "kleiner" als Wert val,
                // im rechten Teilbaum weitersuchen
               return containsRec(val, node.right);
        }
}
```

Suche im binären Suchbaum: iterative Implementierung

```
public boolean contains(E val) {
        Entry node = root;
        while (node != null) {
                int result = val.compareTo(node.value);
                if (result == 0) return true; //gefunden
                if (result < 0) {
                // Wert des Suchbaumknotens node "groesser" als Wert val, links
               // weitersuchen
        node = node.left;
        } else {
                // Wert des Suchbaumknotens node
                // "kleiner" als Wert val, rechts weitersuchen
                 node = node.right;
        }
        }
        return false;
}
```

Einfügen in einen binären Suchbaum

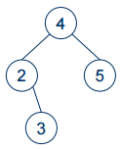
- Aufgabe: Füge 12 in gegebenen Binärbaum ein.
- Grundidee:
  - Suche nach einzufügendem Wert im Binärbaum.
  - Wenn der einzufügende Wert nicht gefunden wird, dann füge ihn als neuen Nachfolger des Knotens ein, an dem die Suche erfolglos endete (und gib true zurück).
  - Sonst: Gib false zurück (Suchbäume enthalten keine Duplikate).

Einfügen in einen binären Suchbaum: iterative Implementierung

```
public boolean add(E val) {
        if (root == null) { // Sonderfall: Einfuegen in leeren Baum
              root = new Entry(val, null);
              return true;
        }
        Entry node = root;
        Entry dragPtr = null; // Schleppzeiger
        int result = 0;
        while (node != null) {
              result = val.compareTo(node.value);
              if (result == 0) return false;
              dragPtr = node; // zeigt auf letzten Knoten
              if (result < 0)
                    node = node.left;
              else
                    node = node.right;
        }
        // dragPtr zeigt auf das Blatt, an das anzu-
        // haengen ist. Neuen Knoten an Blatt anfuegen.
        Entry newNode = new Entry(val, dragPtr);
        if (result < 0) { // result hat Ergebnis des letzten Vergleichs
              dragPtr.left = newNode;
        } else {
              dragPtr.right = newNode;
        }
        return true;
}
```

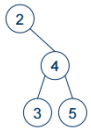
Einfügereihenfolge bedingt das Aussehen

4, 2, 5, 3 nacheinander in zuvor leeren Baum einfügen



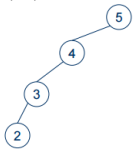
- Höhenunterschiede am Beispiel:
- Knoten 4: ±1, da $h(2) = 1$, $h(5) = 0$
- Knoten 2: ±1, da $h(\text{null}) = -1$, $h(3) = 0$
- ausgewogener/balancierter Baum, da rechte und linke Unterbäume stets etwa die gleiche Höhe (maximaler Unterschied ±1) haben.
- sogar vollständig ausgewogener Baum, da sich die Knotenanzahl in Unterbäumen auch um maximal ±1 unterscheiden.

2, 4, 3, 5 nachHöhenunterschiede am Beispiel:



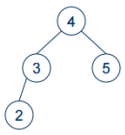
- Knoten 4: 0, da $h(3) = 0$, $h(5) = 0$
- Knoten 2: ±2, da $h(\text{null}) = -1$, $h(4) = 1$
- nicht ausgewogener/unbalancierter Baum.einander in zuvor leeren Baum einfügen

5, 4, 3, 2 nacheinander in zuvor leeren Baum einfügen



- Höhenunterschiede am Beispiel:
- Knoten 5: ±3, da $h(4) = 2$, $h\ (\text{null}) = -1$
- nicht ausgewogener/unbalancierter Baum.
- Extremer Fall, da vorsortierte Eingabe: Suchen, Einfügen haben Aufwand $O(n)$.

4, 3, 2, 5 nacheinander in zuvor leeren Baum einfügen:



- ausgewogener/balancierter Baum.
- vollständig ausgewogener Baum.

Löschen aus einem binären Suchbaum

- Um einen Knoten zu löschen, muss man diesen zuerst finden. Dann sind folgende drei Fälle zu unterscheiden:
  - Löschen eines Blattknotens
    - einfach: Knoten wird entfernt.
  - Löschen eines Knotens mit nur einem Nachfolger
    - einfach: der eine Nachfolger rückt auf.
  - Löschen eines Knotens mit zwei Nachfolgern
    - nicht so einfach: geeignete Strategie erforderlich, um Suchbaumeigenschaft zu erhalten.

Zwei alternative Strategien

- Variante 1
  - Finde im rechten Unterbaum des zu löschenden Knotens den Knoten mit minimalem Wert (= den am weitesten links stehenden Knoten) und merke Dir dessen Wert.
  - Lösche diesen Knoten mit minimalem Wert aus dem Baum.
  - Überschreibe den Wert des ursprünglich zu löschenden Knotens mit dem gemerkten minimalen Wert.
- Variante 2
  - Finde im linken Unterbaum des zu löschenden Knotens den Knoten mit maximalem Wert (= den am weitesten rechts stehenden Knoten) und merke Dir dessen Wert.
  - Lösche diesen Knoten mit maximalem Wert aus dem Baum.
  - Überschreibe den Wert des ursprünglich zu löschenden Knotens mit dem gemerkten maximalen Wert.

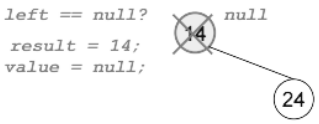
Löschen aus einem binären Suchbaum: rekursive Implementierung

```
...
 private class Entry {
         ...
         private E removeMin() {
                E result;
               if (left == null) {
                       result = value;
                       value = null;
               } else {
                       result = left.removeMin();
                       if (left.value == null) {
                              left = left.right;
                              if (left != null) {
                                      left.parent = this;
                               }
                        }
                }
                return result;
         }
 ..
```

- Hilfsmethode, die minimalen Schlüsselwert in nichtleerem Baum ermittelt und den zugehörigen Knoten löscht.
- Nach rekursivem Abstieg muss im Elternknoten des zu löschenden Knotens Zeiger umgesetzt werden.
- Trick:
  - Im zu löschenden Knoten wird dessen Wert (value) auf null gesetzt.
  - Diese Kennzeichnung kann beim Aufstieg überprüft werden.
- Achtung: Sonderfall „Wurzel des Teilbaums ist zu löschender Min-Knoten“ muss noch separat verarbeitet werden

removeMin am Beispiel

- Achtung: Sonderfall
- zu löschender Knoten ist Wurzel des betrachteten Teilbaums.
- Dieser Sonderfall muss an der aufrufenden Stelle noch separat behandelt werden



Löschen aus einem binären Suchbaum: rekursive Implementierung

```
        ...
         public Entry remove(E val) {
                int result = val.compareTo(value);
                if (result < 0) { // val < value
                       if (left != null) {
                              left = left.remove(val);
                              if (left != null) left.parent = this;
                        }
                        return this;
                } else if (result > 0) { // val > value
                       if (right != null) {
                              right = right.remove(val);
                              if (right != null) right.parent = this;
                       }
                       return this;
                } else { // val == value
                       if ((left == null) && (right == null)) return null;
                        else if (left == null) return right;
                        else if (right == null) return left;
                        else { // der aktuelle Knoten hat zwei Kinder
                              value = right.removeMin();
                               if (right.value == null) {
                                     right = right.right;
                                     if (right != null) right.parent = this;
                               }
                              return this;
                       }
                }
         }
 } // Ende innere Klasse Entry
```

Wurzel muss umgesetzt werden, da sie sich durch die Löschoperation geändert haben könnte.

```
...
 public void remove(E val) {
       root = root.remove(val);
       root.parent = null;
 }
 ..
```

Aufwandsbetrachtung

- Suchen, Einfügen und Löschen folgen jeweils einem einzigen Pfad im Baum von der Wurzel zu einem Blatt oder einem inneren Knoten.
- Aufwand ist proportional zur Länge des Pfades:
  - Balancierte Bäume: Pfadlänge $O(\log n)$
  - (Zur Liste) degenerierte Bäume: Pfadlänge $O(n)$; entstehen
  - insbesondere dann, wenn man die Werte in sortierter Reihenfolge einfügt.
  - Beispiel: 37, 52, 70, 92 in binären Suchbaum einfügen Aufwand für Aufbau des gesamten Baumes $O(n^2)$ also: schlechtes Worst-Case-Verhalten!

Besuchsreihenfolgen für die Knoten eines Baumes

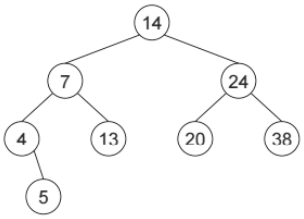
- Drei grundsätzliche Besuchsreihenfolgen: inorder, preorder, postorder
- Auffassbar als Operationen, die Baum in eine Sequenz überführen

```
private class Entry {
 ...
        public void inorder() {
               if (left != null) left.inorder();
               System.out.println("Value:" + value);
               if (right != null) right.inorder();
        }

        public void preorder() {
               System.out.println(value);
               If (left != null) left.preorder();
               if (right != null) right.preorder();
        }
       public void postorder() {
               if (left != null) left.postorder();
               if (right != null) right.postorder();
               System.out.println(value);
        }
 }
```

Besuchsreihenfolgen für die Knoten eines Baumes

- Inorder
  - Besuche linken Unterbaum
  - Aktion auf aktuellem Knoten
  - Besuche rechten Unterbaum
- Preorder
  - Aktion auf aktuellem Knoten
  - Besuche linken Unterbaum
  - Besuche rechten Unterbaum
- Postorder
  - Besuche linken Unterbaum
  - Besuche rechten Unterbaum
  - Aktion auf aktuellem Knoten



- Beispiel:
  - inorder-Durchlauf („links, mitte, rechts“):
    - 4, 5, 7, 13, 14, 20, 24, 38
  - preorder-Durchlauf („mitte, links, rechts“):
    - 14, 7, 4, 5, 13, 24, 20, 38
  - postorder-Durchlauf („links, rechts, mitte“):
    - 5, 4, 13, 7, 20, 38, 24, 14

Klassifikation von Suchbäumen

- Anzahl m der Kinder pro Knoten (Knotengrad):
  - Binärbaum: Anzahl 0 ≤ mm ≤ 2
  - allgemeiner Baum (auch: Vielwegbaum): Anzahl beliebig
- Speicherort der Nutzdaten:
  - natürlicher Baum: Nutzdaten werden bei jedem Knoten gespeichert.
  - Blattbaum/hohler Baum: Nutzdaten werden nur in den Blattknoten gespeichert; innere Knoten dienen nur der Verzweigung.
- Ausgewogenheit/Ausgeglichenheit/Balanciertheit:
  - ausgewogener/ausgeglichener/balancierter Baum:
    - Für alle Knoten gilt: Die Höhen aller ihrer Unterbäume unterscheiden sich höchstens um Δn (meistens ±1) (sog. Höhenbalance).
    - vollständig ausgewogen: Die Anzahl der Knoten in den Unterbäumen unterscheidet sich höchstens um 1.
  - nicht ausgewogener/unbalancierter Baum: keine derartige Einschränkung

AVL-Bäume

- Binäre Suchbäume
  - gutes Durchschnittsverhalten
  - aber: sehr schlechter Worst Case, wenn Elemente in bereits sortierter Reihenfolge eingefügt werden.
- Alternative: AVL-Bäume
  - ermöglichen Einfügen, Löschen und Suchen in $O(\log n)$ auch im Worst Case,
  - sind balancierte, binäre Suchbäume.
  - Grundidee: sog. Strukturinvariante für binären Suchbaum formulieren und diese bei jeder Aktualisierungen (Einfügen, Löschen) aufrecht erhalten

Strukturinvariante der AVL-Bäume (AVL-Bedingung)

- Definition: Ein AVL-Baum ist ein binärer Suchbaum, in dem sich die Höhen seiner zwei Teilbäume höchstens um 1 unterscheiden.
- oder anders ausgedrückt:
  - AVL-Bedingung: Sei ee ein beliebiger Knoten eines binären Suchbaumes, und $h(e)$ die Höhe des Teilbaums mit Wurzel e, dann gilt für die beiden Kinder e.left und $e.\mathrm{right}$ von e: $|h(e.\mathrm{left}) - h(e.\mathrm{right})| \leq 1$
- Konsequenz: Wird diese Strukturinvariante durch Aktualisierungen verletzt, so muss sie durch eine Rebalancieroperation wieder hergestellt werden.

Herstellung von balancierten/ausgewogenen Bäumen

- Möglichkeit 1: globale Reorganisation
  - Alle Elemente aus einem unbalancierten/unausgewogenen Baum auslesen, diese dann so umordnen, dass die neu sortierte Folge beim Einfügen in einen frischen Baum Balanciertheit/Ausgewogenheit liefert.
  - Nachteil: aufwändig
- Möglichkeit 2: lokale Reorganisation durch Rebalancieroperation
  - Balanciertheit/Ausgewogenheit bei jeder Einfüge-/Löschoperation durch lokalen Umbau des Baums sichern.

- Manipulation jeweils einiger Knoten in der Nähe der Wurzel des aus der Balance geratenen Teilbaums: Teilbaum anheben/absenken, um Höhen anzugleichen.
- Das ist die bessere Strategie

Aktualisierung eines AVL-Baums Einfügen/Löschen

- Einfüge-/Löschoperation wird zunächst genauso ausgeführt wie in gewöhnlichem binärem Suchbaum.
- Einfügeoperation
  - Fügt in jedem Fall neues Blatt hinzu.
  - Das neu erzeugte Blatt (= aktuelle Position) gehört zu allen Teilbäumen von Knoten, die auf dem Pfad von der Wurzel zu diesem Blatt liegen.
  - Durch Einfügung können die Höhen dieser Teilbäume um 1 wachsen.
- Löschoperation
  - Entfernt irgendwo im Baum einen Knoten: Blatt oder innerer Knoten.
  - Der Elternknoten des gelöschten Knotens sei aktuelle Position (Sonderfall Wurzel beachten).
  - Teilbäume auf dem Pfad von der Wurzel zu diesem Elternknoten sind betroffen: Deren Höhe könnte sich um 1 verringert haben.
- Danach folgt in beiden Fällen eine Rebalancierphase falls die AVL-Eigenschaft wieder hergestellt werden muss.

Aktualisierung eines AVL-Baums Rebalancierphase

- Rebalancierphase
  - Man läuft von aktueller Position aus den Pfad zur Wurzel zurück.
  - In jedem Knoten wird dessen Balance geprüft: falls Knoten aus der Balance geraten ist, so wird das durch eine der im Folgenden beschriebenen Rebalancieroperationen korrigiert.
  - Wir können also davon ausgehen, dass alle Knoten unterhalb des gerade betrachteten (aus der Balance geratenen) Knotens selbst balanciert sind.
- Hinweis: Suchen im AVL-Baum erfolgt genau so, wie in gewöhnlichem Suchbaum.

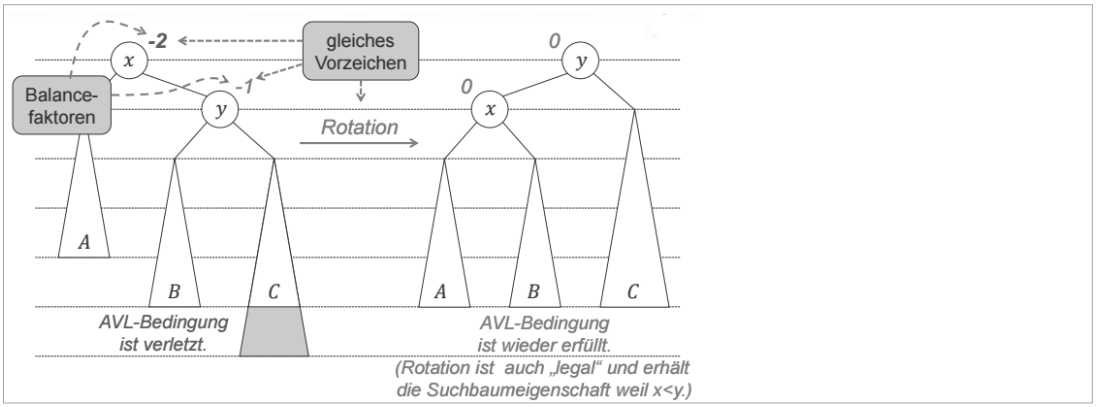
Bestimmung der Höhe und des Balancefaktors eines Knotens

- Balancefaktor bf eines Knotens: Differenz zwischen der Höhe des linken Teilbaums und der Höhe des rechten Teilbaums.
- Induktionsanfang:
  - Höhe h und Balancefaktor bf eines leeren Teilbaums sind -1. (Damit sind Höhe h und Balancefaktor bf eines Blattes 0).
- Induktionshypothese/-annahme:
  - Höhe und Balancefaktoren von Binärbäumen mit weniger als n Knoten können berechnet werden.
- Induktionsschluss:
  - Betrachte die Wurzel eines Baums mit n Knoten.
  - Dessen Teilbäume haben weniger als n Knoten.
  - Laut Induktionshypothese können deren Höhen und Balancefaktoren berechnet werden.
  - Dann gilt für die Höhe h der Wurzel:
  - h (Wurzel) = 1 + max (h (linker Teilbaum) , h (rechter Teilbaum))
  - Und damit für den Balancefaktor bf:
  - bf(Wurzel) = h (linker Teilbaum) – h(rechter Teilbaum)
- Aufwandsbetrachtungen
  - Berechnung von Höhe und Balancefaktor für den ganzen Baum:
    - Die rekursive Implementierung steigt bis zu den Blättern ab und berechnet dann Höhe und Balancefaktoren „von unten nach oben“ im Baum.
    - Jeder Knoten wird dabei einmal besucht: Aufwand: $O(n)$.
  - Neuberechnung bei einer Änderung:
    - Nimmt man im Baum eine Änderung vor (Hinzufügen eines neuen Knotens oder Löschen eines Knotens), dann ändern sich die Höhenangaben und Balancefaktoren nur bei den Knoten, die auf dem Pfad von der Änderungsstelle bis zur Wurzel liegen.
    - Die Korrektur nach einer Änderung hat daher den Aufwand $O(\log 2n)$.

Rebalancieren – Fall a: nach Einfügung in einen AVL-Baum

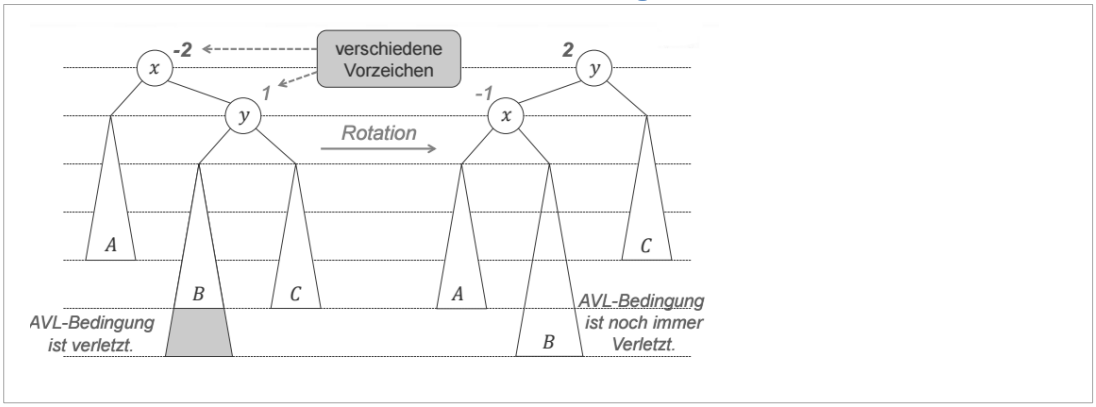
- Annahme:
  - Gerade betrachteter Knoten (Teilbaum) ist durch Einfügung aus der Balance geraten.
  - Also ist die Höhe einer seiner Teilbäume um 1 gewachsen.
  - O. B. d. A. (=ohne Beschränkung der Allgemeinheit) sei der rechte Teilbaum um 2 tiefer (bzw. höher) als der linke.
  - Der andere Fall (linker Teilbaum tiefer) ist symmetrisch und kann analog behandelt werden.
- Zwei Unterfälle unterscheidbar:
  - a1: Innerhalb des rechten Teilbaums ist der rechte Teilbaum gewachsen.
  - a2: Innerhalb des rechten Teilbaums ist der linke Teilbaum gewachsen.

Fall a1: Innerhalb des rechten Teilbaums ist der rechte Teilbaum gewachsen.



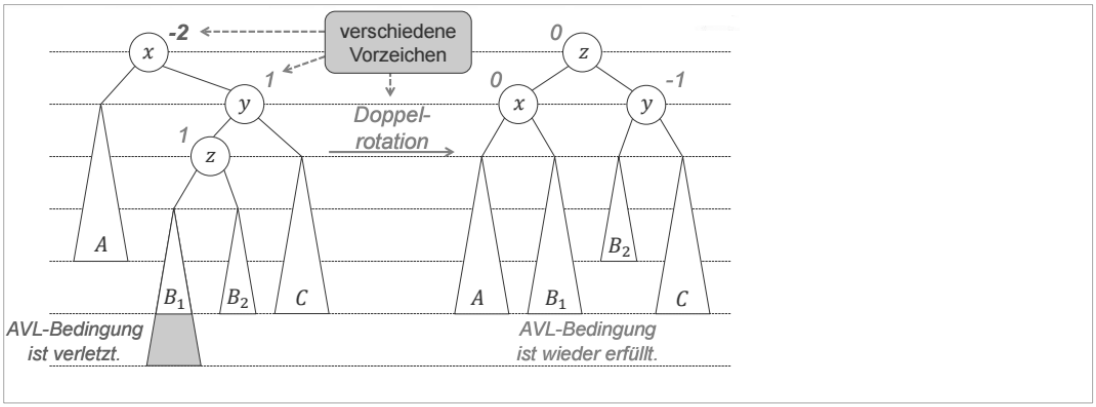
hier: Einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn ausreichend, um Balance wiederherzustellen.

Fall a2: Innerhalb des rechten Teilbaums ist der linke Teilbaum gewachsen



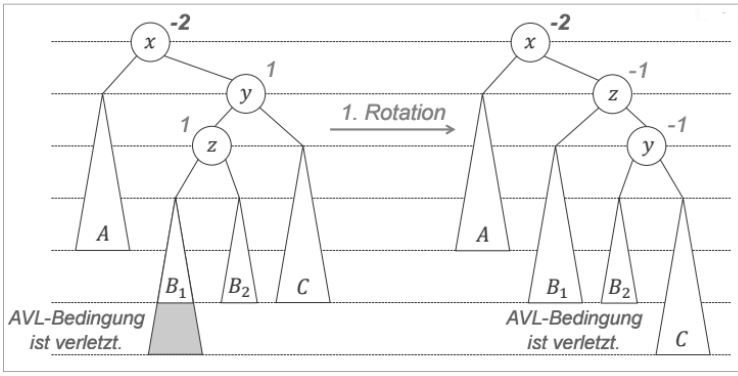
Frage: Was tun? Offenbar genauere Analyse von Fall a2 erforderlich.

Fall a2.1: Innerhalb des rechten Teilbaums ist der linke Teilbaum des linken Teilbaums gewachsen



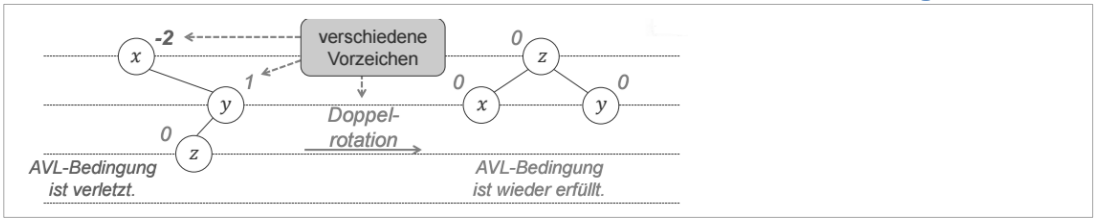
Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren. Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren

Fall a2.1: Innerhalb des rechten Teilbaums ist der linke Teilbaum des linken Teilbaums gewachsen. (II)



Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren. Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren.

Fall a2.2: Innerhalb des rechten Teilbaums ist der rechte Teilbaum des linken Teilbaums gewachsen.



Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren. Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren.

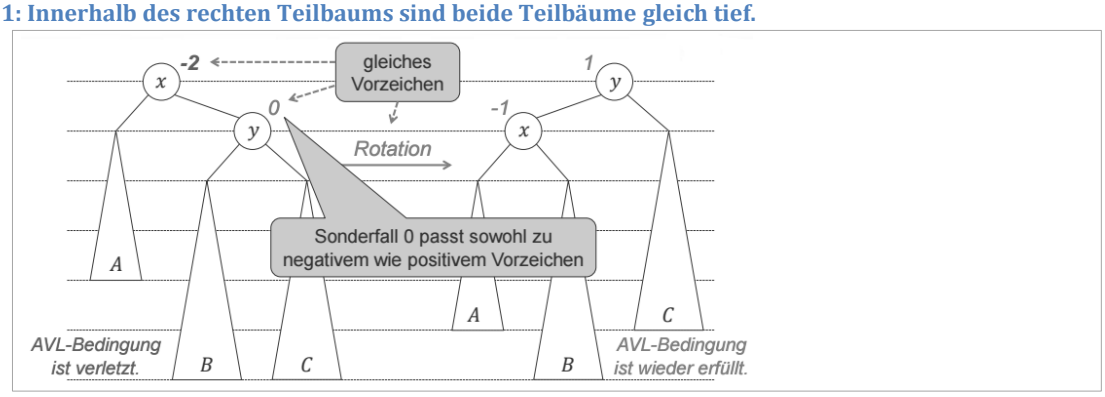
Rebalancieren – Fall a: nach Einfügung in einen AVL-Baum

- also: Nach einer Einfügung genügt eine einzige Rotation oder Doppelrotation, um die Strukturinvariante des AVL-Baums (AVL-Bedingung) wiederherzustellen (und dabei die Suchbaumeigenschaft zu erhalten).
- Beweis (für Interessierte):
  - Eine Rotation oder Doppelrotation im ersten aus der Balance geratenen Knoten e auf dem Pfad von der aktuellen Position zur Wurzel sorgt dafür, dass der Teilbaum mit Wurzel e die gleiche Höhe hat, wie vor der Einfügung.
  - Also haben auch alle Teilbäume, deren Wurzeln Vorfahren von e sind, die gleiche Höhe und keiner dieser Knoten kann jetzt noch unbalanciert sein.

Rebalancieren – Fall b: nach Löschung aus einem AVL-Baum

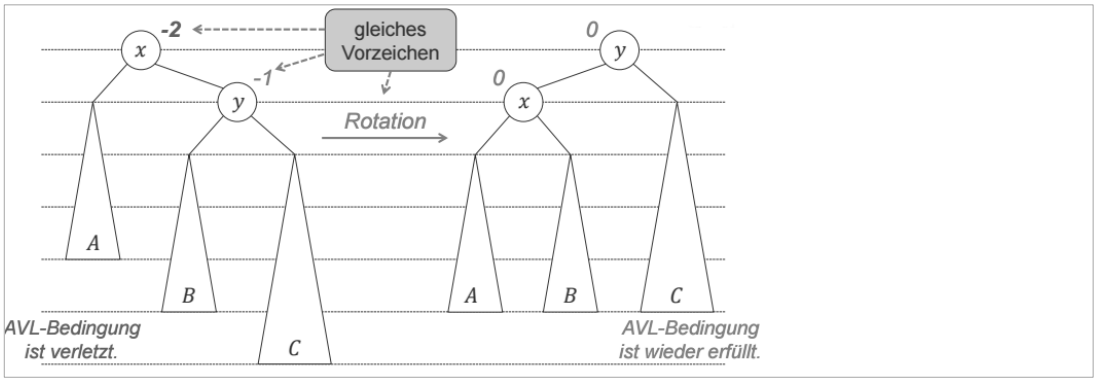
- Annahme:
  - Gerade betrachteter Knoten (Teilbaum) ist durch Löschung aus der Balance geraten.
  - Also ist die Höhe einer seiner Teilbäume um 1 geschrumpft.
  - O. B. d. A. sei der rechte Teilbaum um 2 tiefer (bzw. höher) als der linke.
  - Der andere Fall (linker Teilbaum tiefer) ist symmetrisch und kann analog behandelt werden.
- Drei Unterfälle (nach Löschung) unterscheidbar:
  - b1: Innerhalb des rechten Teilbaums sind beide Teilbäume gleich tief.
  - b2: Innerhalb des rechten Teilbaums ist der rechte Teilbaum tiefer.
  - b3: Innerhalb des rechten Teilbaums ist der linke Teilbaum tiefer.

Fall b1: Innerhalb des rechten Teilbaums sind beide Teilbäume gleich tief.



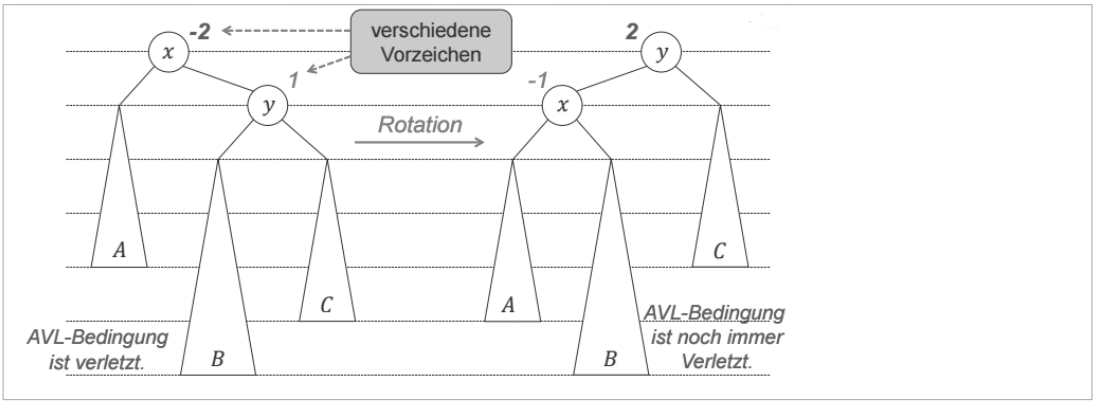
hier: Einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn ausreichend, um Balance wiederherzustellen.

Fall b2: Innerhalb des rechten Teilbaums ist der rechte Teilbaum tiefer.



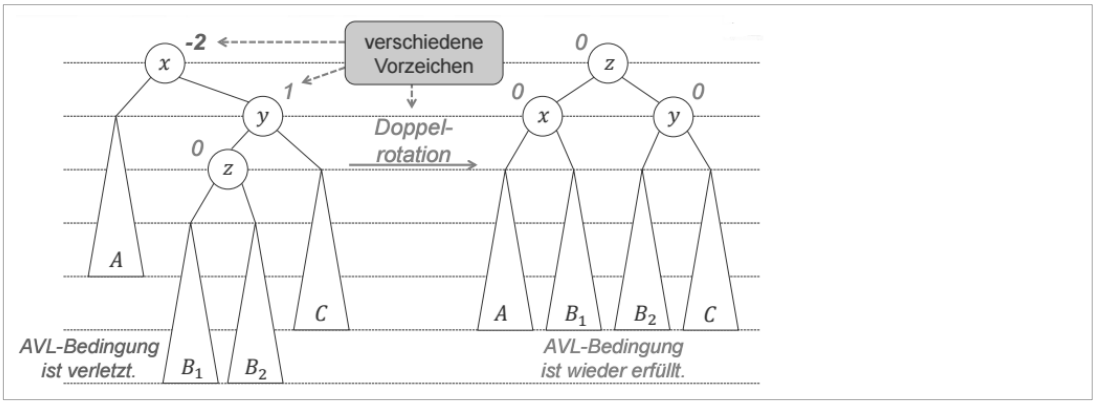
hier: Einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn ausreichend, um Balance wiederherzustellen.

Fall b3: Innerhalb des rechten Teilbaums ist der linke Teilbaum tiefer.



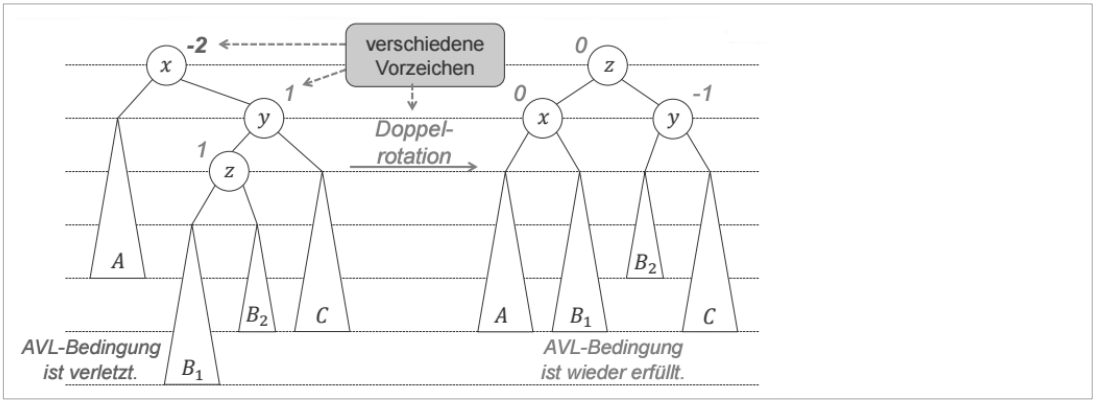
Frage: Was tun? Offenbar wieder genauere Analyse von Fall b3 erforderlich (vgl. a2).

Fall b3.1: Innerhalb des rechten Teilbaums sind beide Teilbäume des linken Teilbaums gleich tief.



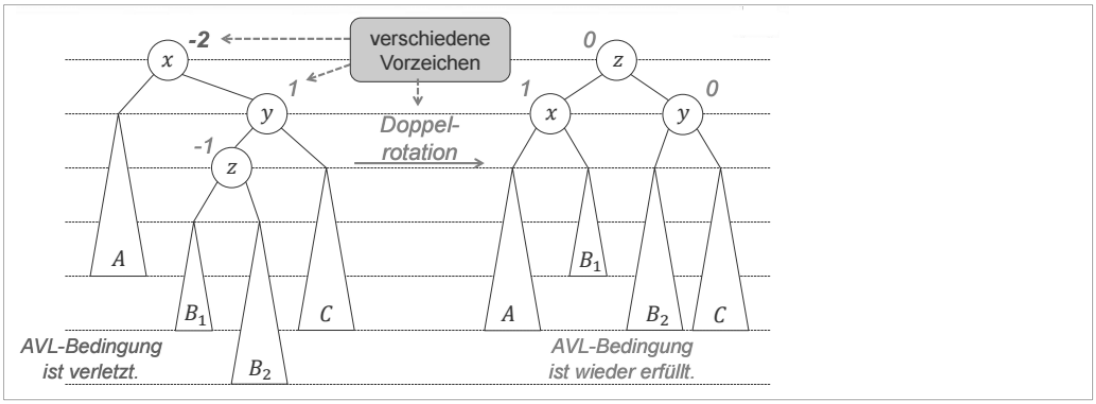
Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren.Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren.

Fall b3.2: Innerhalb des rechten Teilbaums ist der linke Teilbaum des linken Teilbaums tiefer.



Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren

Fall b3.3: Innerhalb des rechten Teilbaums ist der rechte Teilbaum des linken Teilbaums tiefer.



zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren

Rebalancieren – Fall b: nach Löschung aus einem AVL-Baum

also: Ein durch eine Löschoperation aus der Balance geratener (Teil-) Baum kann durch eine Rotation oder Doppelrotation wieder ausgeglichen werden. Es kann aber erforderlich sein, auch Vorgängerbäume bis hin zur Wurzel zu rebalancieren. Die

Zwischenfazit und Hinweis zur Symmetrie

Zwei grundlegende Fälle unterscheidbar:

- Balancefaktoren der Wurzel des aus der Balance geratenen (Teil-) Baums und deren Kind y auf dem Pfad in den zu tiefen Teilbaum haben gleiche Vorzeichen; Lösung: einfache Rotation
  - rechter Teilbaum zu tief: einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn; Fälle: a1, b1, b2
  - linker Teilbaum zu tief: einfache Rotation des gesamten (Teil-) Baums im Uhrzeigersinn; entsprechende Symmetriefälle: a1*, b1*, b2*
  - Beachte: Bei der Betrachtung der Symmetriefälle („Spiegelung der Fälle an Senkrechte durch Wurzel des Baums“, d. h. „linker Teilbaum zu tief“) wird die Rotationsrichtung jeweils umgekehrt.
- Balancefaktoren der Wurzel des aus der Balance geratenen (Teil-) Baums und deren Kind y auf dem Pfad in den zu tiefen Teilbaum haben verschiedene Vorzeichen; Lösung: Doppelrotation
  - rechter Teilbaum zu tief: zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren; danach gesamten (Teil-) Baum gegen den Uhrzeigersinn rotieren; Fälle: a2.1, a2.2, a2.3, b3.1, b3.2, b3.3
  - linker Baum zu tief:
  - zunächst nur Teilbaum mit Wurzel y gegen den Uhrzeigersinn rotieren; danach gesamten (Teil-) Baum im Uhrzeigersinn rotieren; Symmetriefälle: a2.1*, a2.2*, a2.3*, b3.1*, b3.2*, b3.3*
  - Aufwand von Rotation und Doppelrotation: Es sind jeweils eine konstante Anzahl von Referenzen zu ändern, Aufwand daher jeweils: $O(1)$.

Weitere Eigenschaften von AVL-Bäumen

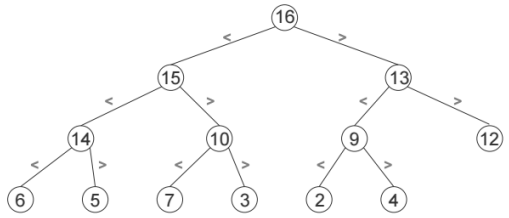
- Zusammenhänge für die Höhe h eines AVL-Baums mit nn Knoten:
  - $\log2(n+1) \leq h < 1{,}441 \log2(n+2)$ Im Vergleich zum vollständig ausgewogenen Binärbaum (mit minimaler Höhe) ist der AVL-Baum also höchstens ~44% höher. Alle Operationen bleiben daher vom Aufwand $O(\log2n)$.
- Nachteile von AVL-Bäumen:
  - Zusätzlicher Platzbedarf in den Knoten zur Speicherung der Höhe oder der Balancefaktoren (der andere Wert kann leicht berechnet werden, weil die Änderungsoperation vom Blatt in Richtung Wurzel läuft).
  - Komplizierte Implementierung.

Halde/Haufen (engl. Heap)

- Ein partiell geordneter Baum ist ein knotenmarkierter Binärbaum T, in dem für jeden Teilbaum T' mit Wurzel x gilt:
  - Variante 1: in der Wurzel steht immer das Minimum des Teilbaums $\forall y \in T' : \mu(x) \leq \mu(y)$
  - Variante 2: in der Wurzel steht immer das Maximum des Teilbaums $\forall y \in T' : \mu(x) \geq \mu(y)$
- Andere Bezeichnungen: Halde/Haufen (engl. Heap)
- Resultierende Halden-/Haufen-/Heap-Eigenschaft: Wert eines Knotens ist bezüglich einer gegebenen Ordnung kleiner oder gleich (Variante 1) bzw. größer oder gleich (Variante 2) dem Wert seiner zwei Nachfolger

Beispiel für max-Halde

Jede Wurzel eines Teilbaums hat einen größeren Wert, als alle übrigen Knoten dieses Teilbaums.



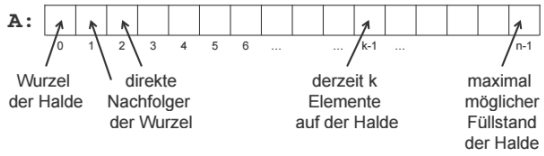
Wir betrachten im Folgenden links-vollständige partiell geordnete Bäume: alle Ebenen bis auf die letzte sind voll besetzt und auf letzter Ebene sitzen Knoten soweit links wie möglich.

Entnahme aus max-Halde

```
Algorithmus deleteMax(h) { // Laufzeit O(logk)
        // loesche das maximale Element aus der Halde h
        // und gib es aus
       entnimm der Wurzel ihren Eintrag und gib ihn als
              Maximum aus;
        nimm den Eintrag der letzten besetzten Position im
              Baum (loesche diesen Knoten) und setze ihn in
               die Wurzel;
        sei p die Wurzel und seien q, r ihre Kinder;
        solange q oder r existieren und
              (μ(p) < μ(q) oder μ(p) < μ(r)) fuehre aus: {
              vertausche den Eintrag in p mit dem groesseren
                      Eintrag der beiden Kinder;
              setze p auf den Knoten, mit dem vertauscht
                     wurde, und q, r auf dessen Kinder;
        }
}
```

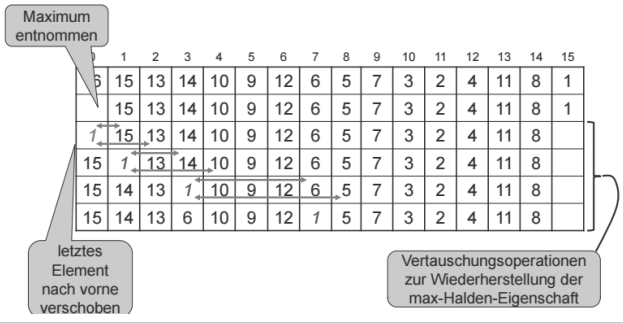
Implementierung einer Halde durch Einbettung in Array

Zur Speicherung einer Halde wird oft die BinärbaumImplementierung „Einbettung in ein Array“ verwendet.



Implementierung: Entnahme aus max-Halde (Array-Einbettung)

Schrittweise Veränderung der in das Array A eingebetteten max-Halde:



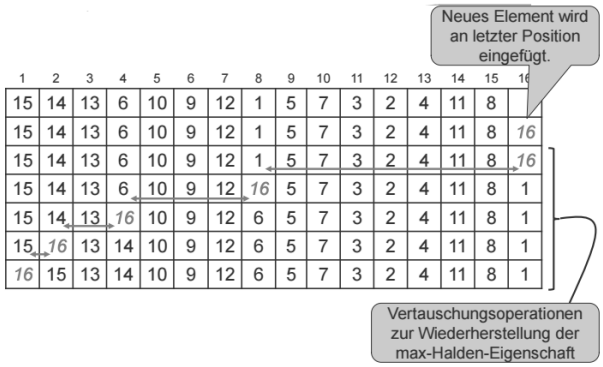
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	15	13	14	10	9	12	6	5	7	3	2	4	11	8	1
	15	13	14	10	9	12	6	5	7	3	2	4	11	8	1
1	15	13	14	10	9	12	6	5	7	3	2	4	11	8	
15	*1*	13	14	10	9	12	6	5	7	3	2	4	11	8	
15	14	13	*1*	10	9	12	6	5	7	3	2	4	11	8	
15	14	13	6	10	9	12	*1*	5	7	3	2	4	11	8	

Einfügen in max-Halde

```
Algorithmus insertMax(h,e) {
        // fuege Element e in den Heap h ein
        erzeuge neuen Knoten q mit Eintrag e;
        fuege q auf der ersten freien Position der
              untersten Ebene ein (falls unterste Ebene voll
              besetzt ist, beginne neue Ebene);
        sei p der Elternknoten von q;
        solange p existiert und μ(q) > μ(p), fuehre aus: {
              vertausche die Eintraege in p und q;
              setze q auf p und p auf den Elternknoten von p;
        }
}
```

Implementierung: Einfügen in max-Halde (Array-Einbettung)

Schrittweise Veränderung der in das Array A eingebetteten max-Halde



Anwendung für Halde

- effiziente Implementierung einer Prioritätswarteschlange
- Realisierung mit Halde (k Elemente)
  - Entnahme eines Elements: $O(\log k)$
  - Einfügen eines Elements: $O(\log k)$
  - Begründung:
    - beide Operationen folgen einem Pfad im Baum
    - Baum ist balanciert
    - Realisierung mittels Array-Einbettung deshalb in $O(\log k)$
- Vergleich dazu Realisierung mit Liste (k Elemente)
  - Entnahme eines Elements: $O(1)$ (zu entnehmendes Element steht vorne)
  - Einfügen eines Elements: $O(k)$

Tafelübung

Baumtraversierung

- Traversierung eines Baumes:
  - alle Knoten des Baumes ausgehend von der Wurzel „besuchen“
- man unterscheidet verschiedene Besuchsreihenfolgen
  - Tiefensuche (DFS, depth-first search):
    - Kinder vor „Geschwistern“ besuchen ; noch zu besuchende Knoten auf einen Stack ablegen
  - Breitensuche (BFS, breadth-first search):
    - „Geschwister“ vor Kindern besuchen noch zu besuchende Knoten in eine Queue ablegen
- Tiefen- und Breitensuche auch auf beliebigen Graphen möglich
  - dann aber notwendig: Markierung der bereits besuchten Knoten

Tiefensuche mit explizitem Stack

```
funktion dfs ( G : Graph ) {
       S := leerer Stack ;
       fuege Wurzel von G in S ein ;
       solange S nicht leer ist fuehre aus {
              a := entferne oberstes Element von S ;
              bearbeite Knoten a ;
              fuer alle Nachfolger n von a fuehre aus {
                     lege n oben auf S ;
              }
       }
}
```

Tiefensuche unter Verwendung des Aufruf-Stacks

```
funktion dfs ( G : Graph ) {
        dfs_helper ( Wurzel von G );
}
funktion dfs_helper ( k : Knoten ) {
        bearbeite k ;
        fuer alle Nachfolger n von k fuehre aus {
                dfs_helper ( n );
        }
}
```

Breitensuche mit Queue

```
funktion bfs ( G : Graph ) {
        Q := leere Queue ;
        fuege Wurzel von G in Q ein ;
        solange Q nicht leer ist fuehre aus {
                a := entferne vorderstes Element aus Q ;
                bearbeite Knoten a ;
                fuer alle Nachfolger n von a fuehre aus {
                        fuege n hinten in Q ein ;
                }
        }
}
```