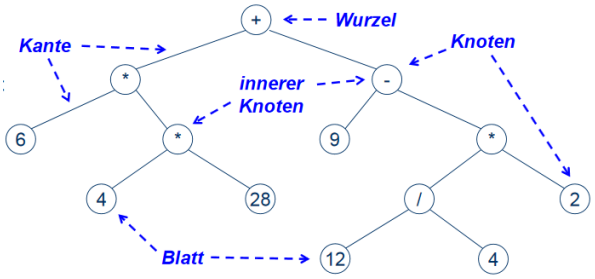
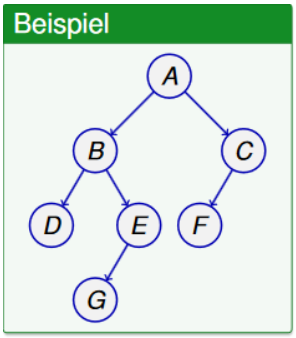
Bäume

Eigenschaften:



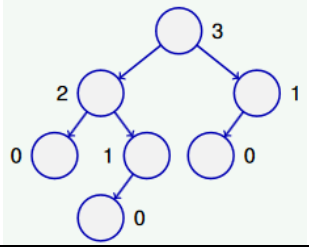
- Keine Zyklen
- Wurzel ist das erste Element und keinen Vorgänger



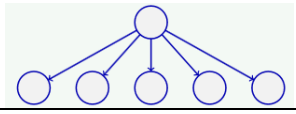
Beziehungen zwischen den Knoten

- *A* ist direkter Vorgänger von *B* und *C*
- *G* ist direkter Nachfolger von *E*
- *B* und *C* sind Geschwisterknoten
- *A* ist indirekter Vorgänger von allen anderen Knoten
- *G* ist indirekter Nachfolger von *A*, *B* und *E*
- *D*, *F* und *G* sind Blätter
- *A*, *B*, *C* und *E* sind innere Knoten

- Höhe eines Baumes ist die Anzahl der Kanten des längsten Baumes zu einem Blatt
- Höhe eines Baumes ist die Höhe seiner Wurzel



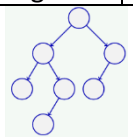
Verzweigungsgrad = Zahl der möglichen Nachfolger pro Element
Bsp: Verzweigungsgrad = 5



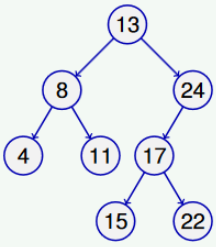
Traversierung: alle Knoten des Baumes ausgehend von der Wurzel „besuchen“

Tiefensuche (DFS depth-first search)	**Breitensuche** (BFS, breadth-first search)
noch zu besuchende Knoten auf Stack ablegen	noch zu besuchende Knoten in Queue ablegen

Binärbaum hat Verzweigungsgrad 2



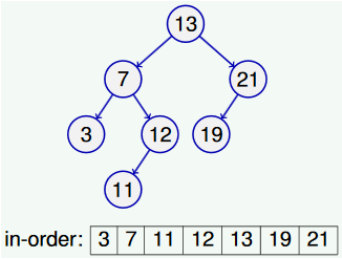
Traversierungsarten



pre-order:	13	8	4	11	24	17	15	22	Bevor das erste Kind besucht wird
in-order:	4	8	11	13	15	17	22	24	Bevor das zweite Kind besucht wird
post-order:	4	11	8	15	22	17	24	13	Nachdem das zweite Kind besucht wurde

Binärer Suchbaum

Ein Binärer Suchbaum ist ein Binärbaum, der die Suchbaumeigenschaft erfüllt: Für jedenKnoten mit Wert X im Baum gilt, dass (bzgl. einer beliebigen, aber festen Ordnungsrelation) der **linke** Teilbaum nur **kleinere** Werte als X und der **rechte** Teilbaum nur **größere** Werte als X beinhaltet.




```java
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;
                }
        }

        Entry root;//Wurzel des Binaerbaums

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

Suche im Binärbaum

- Beginne bei der Wurzel
- Solange gesuchter Wert nicht gefunden:
  - Gesuchter Wert < aktueller Wert: nach links absteigen
  - Gesuchter Wert > aktueller Wert: nach rechts absteigen
  - Kein Kind vorhanden: Wert existiert nicht

```java
//ohne Rekursion
public boolean containsRec(E val) {

        Entry node = root;
                // Baum nicht leer
        while (node != null){

                int result = val.compareTo(node.value);

                if(result == 0) return true; //gefunden

                if(result < 0){
                        //Wert des Suchbaumknotens node "grosser" als Wert val,
                        //im linken Teilbaum weitersuchen
                        node=node.left;
                }else{
                        //Wert des Suchbaumknotesns node "kleiner" als Wert val,
                        //im rechten Teilbaum weitersuchen
                        node=node.right;
                }
        }
        return false;
}
```

```java
//mit Rekursion
public boolean containsRec(E val, Entry node) {
        // Baum leer?
        if (node == null) return false;

        int result = val.compareTo(node.value);

        if(result == 0) return true;

        if(result < 0){
                //Wert des Suchbaumknotens node "grosser" als Wert val,
                //im linken Teilbaum weitersuchen
                return containsRec(val, node.left);
        }else{
                //Wert des Suchbaumknotesns node "kleiner" als Wert val,
                //im rechten Teilbaum weitersuchen
                return containsRec(val, node.right);
        }
}
```

Einfügen eines Wertes

- Suche nach Einfügeposition mittels Suchalgorithmus
- Einhängen des neuen Knotens an dieser Position

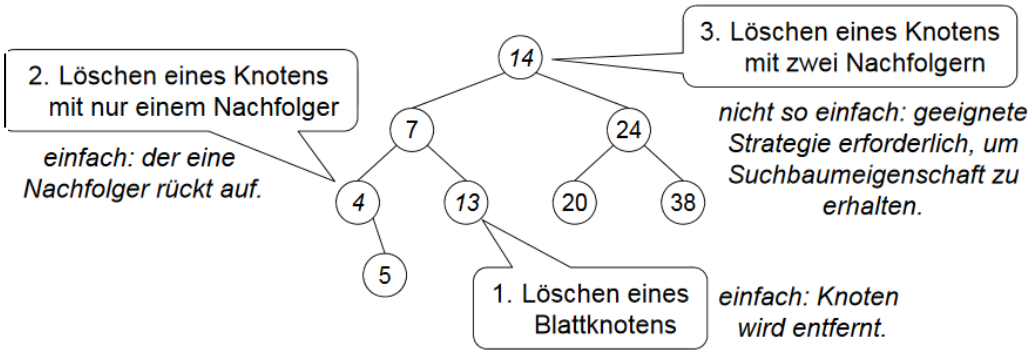
```
public boolean add(E val){
        if (root == null){ //Sonderfall: Einfuegen in leeren Baum
                root = new Entry(val, null);
                return true;
        }
        Entry node = root;
        Entry dragPtr = null; //Schleppzeiger
        int result = 0;

        while (node != null) {
                result = val.compareTo(node.value);
                if (result == 0) return false; // gefunden
                dragPtr = node; //zeigt auf letzen Knoten
                if (result < 0) {
                        // Wert des Suchbaumknotens node "grosser" als Wert val,
                        // im linken Teilbaum weitersuchen
                        node = node.left;
                } else {
                        // Wert des Suchbaumknotesns node "kleiner" als Wert val,
                        // im rechten Teilbaum weitersuchen
                        node = node.right;
                }
                //dragPtr zeigt auf das Blatt, an das anzuhaengen 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;
}
```

Löschen eines Wertes

- Suche nach Knoten mittels Such-Algorithmus
- Zu löschender Knoten Blatt-Knoten: aus Baum entfernen
- Zu löschender Knoten kein Blatt-Knoten: Knoten ersetzen durch..
    - Größten Knoten im linken Teilbaum oder
    - Kleinsten Knoten im rechten Teilbaum

  Dadurch ggf. Weitere Ersetzungen in den Teilbäumen notwendig

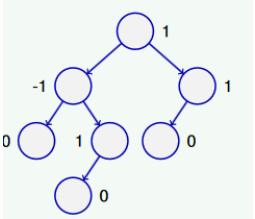


```java
//Methode ermittelt minimalen Schlüsselwert und loescht zugehoerigen Knoten
private E removeMin(){
        E result;
        //wenn linkes Element keinen Nachfolger
        //mehr hat ist kleinster Wert gefunden
        if(left == null){
                result = value;
                //Wert des zu loeschendne Knoten wird auf null gesetzt
                value = null;
        } else{
                //rekursiver Aufruf
                result = left.removeMin();
                //prueft ob Wert des Knoten geloescht wurde
                if(left.value == null){
                        //ersetzen durch rechten Knoten
                        //pruefung ob rechter Knoten existiert fehlt!
                        left = left.right;
                        if(left != null){
                                left.parent = this;
                        }
                }
        }
        return result;
}
//Loeschen aus einem Binaerbaum
public Entry remove(E val){
        int result = val.compareTo(value);
        //linker Teilbaum
        if(result < 0){         //val < result
                if(left != null){
                        left = left.remove(val);
                        //Wurzel des Teilbaums koennte sich
                        //durch zu loeschenden Operant geaendert haben
                        if(left != null) left.parent = this;
                }
                return this;
        //rechter Teilbaum
        }else if(result > 0){   //val > result
                if(right != null){
                        right = right.remove(val);
                        if(right != null) right.parent = this;
                }
                return this;
        }else{                                  //val == result
                if(left == null && right == null) return null;
                else if(left == null) return right;
                else if(right == null) return left;
                else{ //aktueller Knoten hat 2 Kinder
                        value = right.removeMin();
                        if(right.value == null){
                                right = right.right;
                                if(right != null) right.parent = this;
                        }
                }
                return this;
        }
}
```

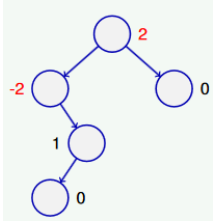
AVL-Baum

Der **Balancefaktor** eines Knotens bezeichnet die Differenz zwischen der Höhe des linken Teilbaums und der des rechten Teilbaums. Blätter haben einen Balancefaktor von 0.

Ein AVL-Baum ist eine spezielle Form eines binären Suchbaums, welcher immer möglichst gut balanciert ist. Dazu stellt der AVL-Baum beim Einfügen und Löschen von Knoten sicher, dass die Balancefaktoren aller Knoten betragsmäßig stets ≤ 1 sind.



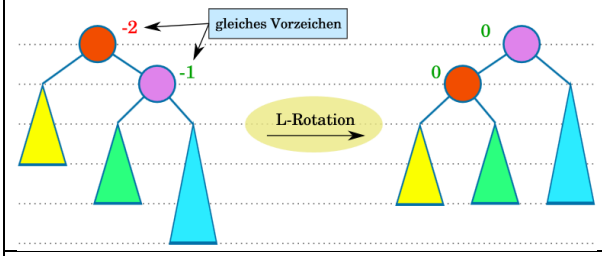
AVL-Baum



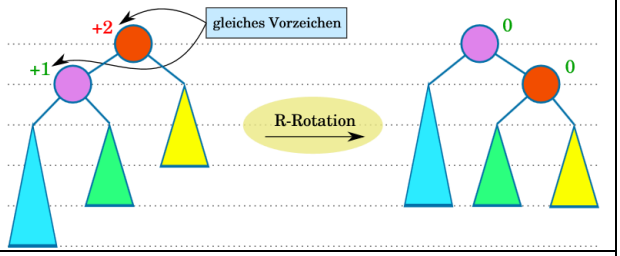
Kein AVL-Baum

- die Operationen Suchen, Einfügen und Löschen funktionieren prinzipiell wie auf einem klassischen Suchbaum
- aber: wenn ein Element hinzugefügt oder gelöscht wird, dann kann sich die Höhe eines (Teil-)Baums um 1 verändern
    - alle Balancefaktoren müssen neu berechnet werden
    - falls mind. ein Balancefaktor betragsmäßig > 1 ist
    - Rebalancierung durch Einzel- oder Doppelrotaionen erfoderlich
- Zusammenhänge für die Höhe h eines AVL-Baums mit n Knoten
    - $log_2(n+1) \leq h < 1{,}441\ log_2(n+2)$
    - Im Verlgeich zum vollständig ausgewogenen Binärbaum (minimaler Höhe) ist der AVL-Baum also höchstens 44% höher
    - Alle Operationen bleiben vom Aufwand O($log_2(n)$).
- Nachteile von AVL-Bäumen
    - Zusätzlicher Platzbedarf in den Knoten zur Speicherung der Höhe oder der Balancefaktoren
    - Komplizierte Implementierung

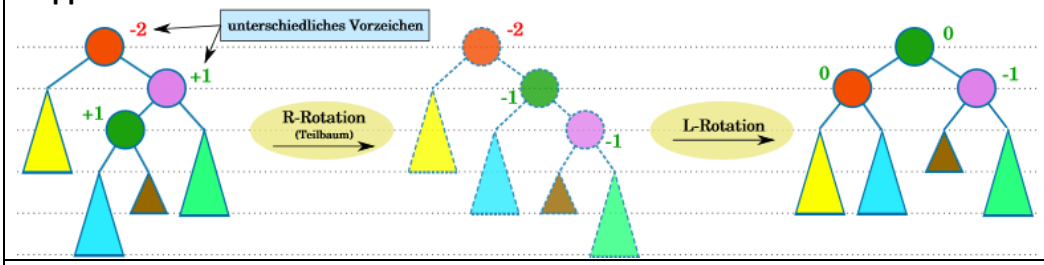
Einzel-Rotation: L-Rotation



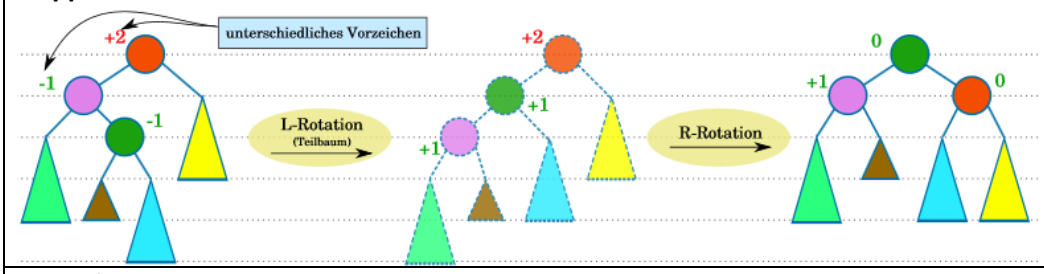
Einzel-Rotation: R-Rotation



Doppel-Rotation: RL-Rotation



Doppel-Rotation: LR-Rotation



Halde/Heap

- Partiell geordneter Baum
- Schnellerer Zugriff auf das größte bzw. kleinste Element (Prioritätswarteschlange)
- Implementierung als Binärbaum möglich

Einfügen eines neuen Elements

- Element an die nächste freie Position in der untersten Ebene einfügen
  - Falls Ebene voll Element wird erster Knoten einer neuen Ebene
- Solgane Haldeneigenschaft in einem Teilbaum verletzt ist
  - Element entsprechend der Haldeneigenschaft nach oben wandern lassen

```
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;
    }
}
```

Löschen eines Elements

- Zu löschendes Element mit dem letzten Element ersetzten
- Letztes Element der untersten Ebene
- Solange Halden-Eigenschaft in einem Teilbaum verletzt ist
- Element entsprechend der Halden-Eigenschaft nach unten wandern lassen
- Min-Heap Tausch mit kleinerem Kind
- Max-Heap Tasuch mit größerem Kind

```
Algorithmus deleteMax(h) {
    // 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;
    }
}
```

Links-Vollständig

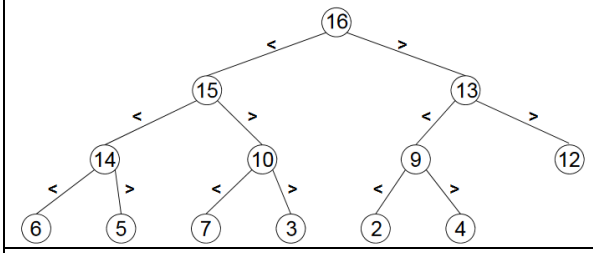
- Alle „Ebenen“ bis auf die unterste sind voll besetzt
- Auf unterster „Ebene“ sitzen alle Knoten so weit links wie möglich

Rechts-Vollständig

- Alle „Ebenen“ bis auf die unterste sind voll besetzt
- Auf unterster „Ebene“ sitzen alle Knoten so weit rechts wie möglich

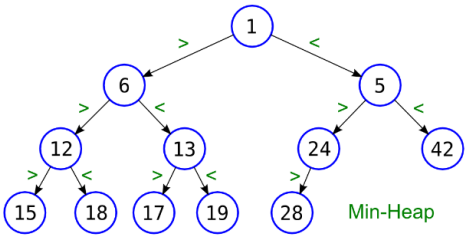
Max-Heap

Wurzel jedes Teilbaums ist größer als andere Knoten des Teilbaums



Min-Heap

Wurzel jedes Teilbaums ist kleiner als ander Knoten des Teilbaums‘



Realisierung mit Halde (k Elemente)

- Entnahme eines Elements O(log k)
- Einfügen eines Elements O(log k)

Begründung:

- Beide Operationen folgen einem Pfad
- Baum ist balanciert
- Realiesierung mit 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)