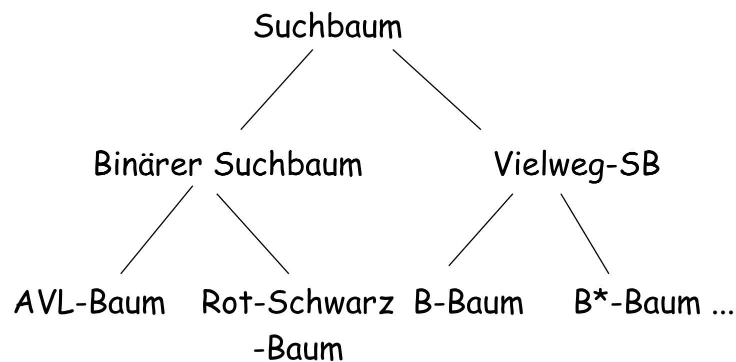
Suchbäume

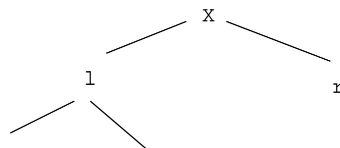


Fast ausgeglichen: Differenz der Blatttiefen begrenzt

Ausgeglichen: alle Blätter besitzen gleiche Tiefe. Wird für Externspeicher-zugiffe verwendet.

Einfügen AVL-Baum

- Annahme: AVL Knoten k enhält Feld mit Höhe des Baumes, dessen Knoten k ist. `(height)`
- Verfahren:
  Rekursives Traversieren zum Einfügepunkt.
  Rückschreitend: Balance prüfen



AVL-Baum: Einfügen

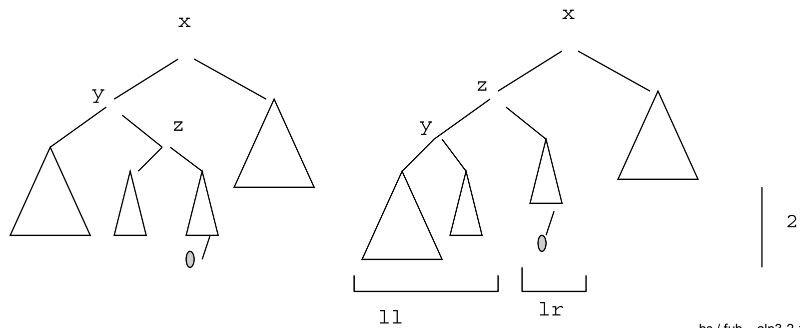
```
AvlNode insert( Comparable x, AvlNode t  {
 if( t == null )  t = new AvlNode( x, null, null );
 else //recursive call of insert()
  if( x.compareTo( t.element ) < 0 ) {
   t.left = insert( x, t.left );
  // Balance check after recursive call
  if( height( t.left ) - height( t.right ) == 2 )
     // Check single or double rotation
     if( x.compareTo( t.left.element ) < 0 )
          t = rotateWithLeftChild( t );
     else t = doubleWithLeftChild( t );
   }
   else if( x.compareTo( t.element ) > 0 )
   ...}      // adjust height
  t.height = max(height(t.left),height(t.right ) ) + 1;
  return t;
}
```


Rotieren

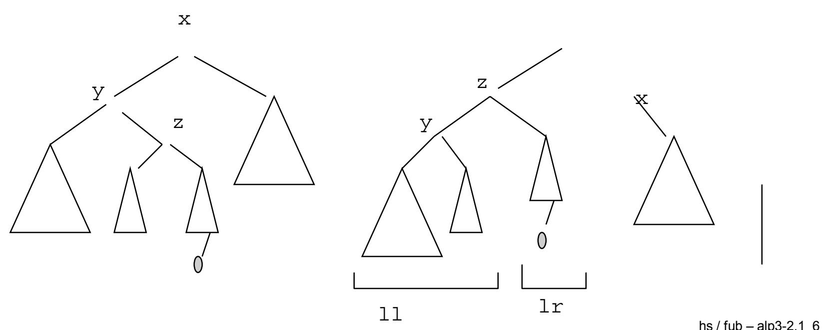
```
AvlNode rotateWithLeftChild( AvlNode x )
  {AvlNode y = x.left;
   x.left = y.right;
   y.right = x;
   x.height = max( height( x.left ),
                   height( x.right ) ) + 1;
  y.height = max( height( y.left ),
                     x.height ) + 1;
  return y;
 }
```

Doppelt rotieren (l r)

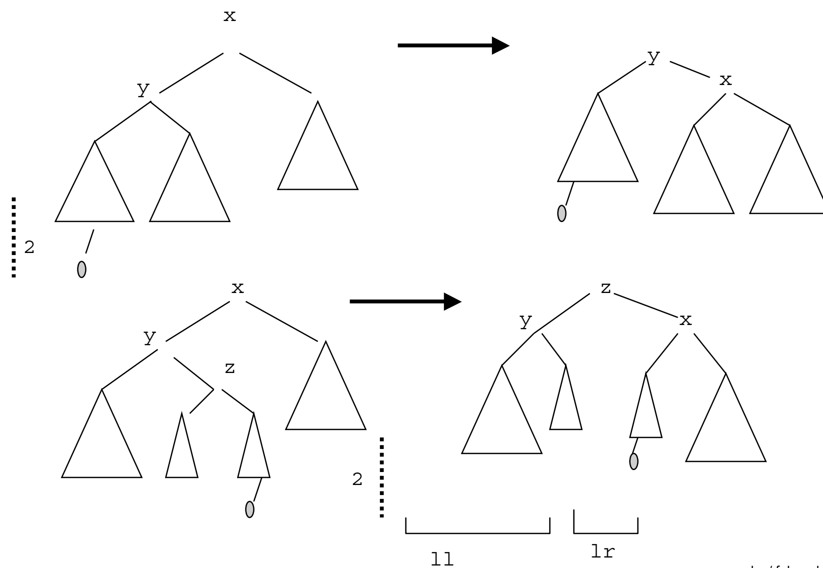
```
AvlNode doubleWithLeftChild( AvlNode x ){
 x.left = rotateWithRightChild( x.left );
 return rotateWithLeftChild( x );        }
```



AVL Bäume: Rotationen

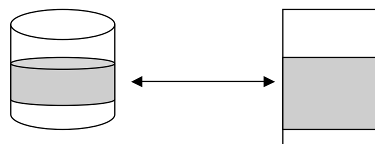


AVL Bäume: Rotationen

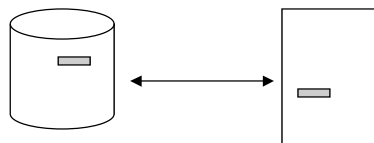


3.3.2.4 B-Bäume - Motivation

- Ziel: Persistente Datenspeicherung mit effizientem Zugriff



" Lösung:"
Daten bei Programmstart in Hauptspeicher laden,
Vor Beendigung serialisiert auf Platte schreiben.
Schlecht!



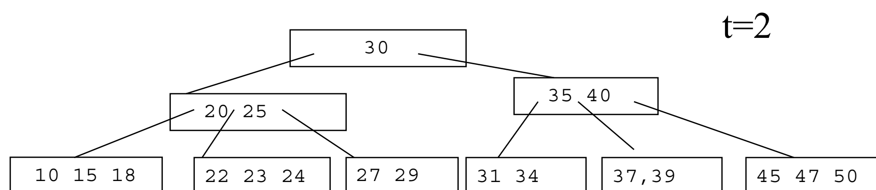
Besser:
Benötigte Daten von der Platte lesen, verändern, zurückschreiben.

Problem: Hohe Lese-/Schreibkosten 1 : 10^4 - 10^5

B-Baum: Suchbaum für Hintergrundspeicher

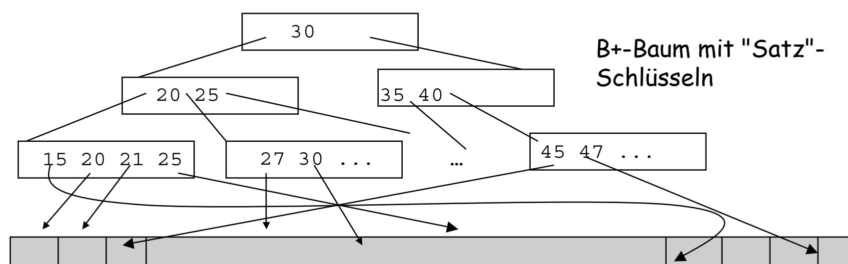
- Mehrweg-Suchbaum mit variablem Grad k,
  $t <= k <= 2t$, t: Minimalgrad
  Wurzel: $1 <= k <= 2t$
- Knoten mit k Unterbäumen enthalten k-1 Werte (Schlüssel, keys)
- alle Blätter haben gleiche Tiefe

t=2



B⁺-Baum (manchmal B*-Baum genannt!)

- Ziel: Großer Verzweigungsgrad innerer Knoten
- Methode: Schlüssel in inneren Knoten, alle Schlüssel-Wert-Paare in Blättern. Wert: Verweis auf Datensatz im Hintergrundspeicher



B+-Baum mit "Satz"-Schlüsseln

Hintergrundspeicher mit Datensätzen (wahlfreier Zugriff)

3.3.5 Hash-Verfahren

- Implementierung von Mengen (und Wörterbücher): Alternative zu Suchbäumen

  to hash: zerhacken, Hackfleisch

- Grundidee: Indexierung der Tabelle mit geeignet transformierten Schlüsselwerten

{30,63,15,11,97,19} ⟶

h(x) = x mod 10

berechnet den den Platz von x in der "Hashtabelle" t (in erster Näherung...)

t

0	30
1	11
2	-
3	63
4	-
5	15
6	-
7	97
8	-
9	19

Hash-Verfahren (1)

- Hash-Verfahren implementiert ein Wörterbuch (Map, dictionnary), das für einen Schlüssel den zugehörigen Wert liefert.
  Beispiele: Telefonbuch: Nachname -> Telefonnr;
  Kasse: Produktcode -> Preis, u.v.m.
- Gegeben
  - Schlüsselmenge V
  - Funktion h : K -> {0, ... ,m-1} (Hashfunktion)
  - Wertemenge V (optional, wenn nur Menge von Schlüsseln verwaltet wird)
  - Tabelle `tab Entry[]` mit `tab.length() == m`

    Implementierung:

    - intern als Feld, extern Folge von Plattenspeicherblöcken
    - Feldelement vom Typ `Entry` speichert Schlüssel k und/oder Wert v

    Tabellenelement ("slots", buckets") kann b Schlüssel k bzw. Schlüssel / Wertpaare (k,v) speichern

    hier: Bucket-Größe b = 1

Hash-Verfahren (2)

- Wichtige Operationen:
  - ```
    boolean insert(Key k, Object v) {
    // requires: k != null
    // effects: if tab[h(k)].value = v'  replace v' by v
    //          else put v into tab[h(k)]
    if (tab[h(k)] == null) tab[h(k)].value = v
           else  ...   ;  //hash collision
    ```
  - ```
    Object find : K -> V
    //z.B.:Feldimplementierung
     Object lookup (Key k){
        if (tab[h(k)].key) == k)
                         return tab[h(k)].value
                      else  ???  //collision or not
                                // found??
    }
    ```


Hash-Kollisionen

- Hash-Kollision liegt vor, wenn
  $h(k_i) = h(k_j)$ für k_i != k_j
- Annahme: für jeden Schlüssel k und
  jedes i, 0<=i<m gilt:
  Wahrscheinlichkeit (h(k) == i) = 1/m
- Wie wahrscheinlich sind Kollisionen?

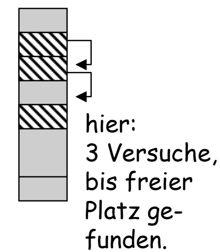
  n Schlüssel einfügen:
  W(mindestens eine Kollision) = 1-W(keine Koll.)
  W(keine K) = 1 * (m-1)/m * (m-2)/m *... * (m-n+1) / m
  $= m! / m^n * (m-n)!$

  Bsp.: m=10, n=8: W= 1-0.0181, m=20, n=10 W= 1-0,065

Hash-Kollisionen

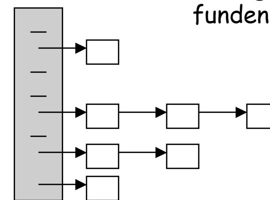
- Gängige Verfahren zur Kollisionsbehandlung:

  - Offene Adressierung
    (= geschlossene Hashtabelle):
    Finde freien Platz in der Tabelle,
    z.B. nächster nicht belegter Platz



hier:
3 Versuche,
bis freier
Platz ge-
funden.

- - Verkettung (Chaining)



Laufzeit bei Verkettung

- Mittlere Laufzeit "Suche mit Schlüssel k"
  tab[].length = m
  n Schlüssel gespeichert
  Füllungsgrad: $\alpha = n/m$
  Annahme: uniformes Hash

  - Nicht erfolgreiche Suche:
    für alle i: tab[i] ist Liste der Länge n/m

    Damit: Suche im Mittel $O(1 + n/m)$
    und mit der Annahme: n/m = const: $O(1)$

Laufzeit bei Verkettung

- Erfolgreiche Suche:

  $1 + n/2*m - 1/2*m = O(1 + \alpha)$ Operationen

- Bewertung:
  unschlagbar schnelles Einfügen / Suchen
  ... wenn "gute" Hash-Funktion und α nicht zu groß
  Sonst z.B. "dynamisches Verändern" der Tabellengröße, aufwendig!
  Sehr einfache Implementierung

Hash Tabelle mit Verkettung

```
public class HashMapSimple  //extends AbstractMap
                            //implements Map {
  private Entry table[];  //Hash Table
  private int count;     // total number of entries

 private static class Entry //implements Map.Entry
  {  int hash;
     Object key;
     Object value;
     Entry next;
     Entry(int hash, Object key, Object value) {
     this.hash = hash;
     this.key = key;
     this.value = value;
  }   ....}
```

Verkettung: Einfügen

```
public Object insert(Object key, Object value) {
 // no duplicate keys not in Hash table.
  Entry tab[] = table;
  int hash = 0;
  int index = 0;
  hash = key.hashCode();
  index = (hash & 0x7FFFFFFF) % tab.length;
  for (Entry e = tab[index] ; e != null ; e = e.next) {
    if ((e.hash == hash) && key.equals(e.key)) {
      Object old = e.value;
      e.value = value;
      return old;        }  }
  // Creates the new entry.
  Entry e = new Entry(hash, key, value);
   e.setNext(tab[index]); //chain in front
  tab[index] = e;
  count++;     return null;
  }
```


Verkettung: Suchen

- Suche

```
public Object lookup(Object key) {
  // search(key) , get(key) ....
  Entry tab[] = table;
  int hash = key.hashCode();
  int index = (hash & 0x7FFFFFFF) %
     tab.length;
  for (Entry e = tab[index]; e != null; e
                                 = e.next)
    if ((e.hash == hash) &&
         key.equals(e.key))
         return e.value;
  return null; }
```