Inhalt

- 3 Datenstrukturen
  - 3.1 Folgen
  - 3.2 Bäume
    - 3.2.1 Grundlagen
    - 3.2.2. Funktonale Spezifikation und einfache Algorithmen
    - 3.2.3 Anwendung von binären Bäumen (Huffman-Code)
    - 3.2.4 Imperative Implementierungen von Bäumen
  - 3.3 Effiziente Repräsentation von Mengen und Relationen
    - 3.3.1 Spezifikation
    - 3.3.2 Suchbäume
      Binäre Suchbäume, AVL-Bäume, B-Bäume
    - 3.3.5 Hashverfahren
    - .....
  - 3.4 Graphen und Graphalgorithmen

3.3 Effiziente Repräsentation von Mengen

- 3.3.1 Spezifikation (funktional)

  Modell: data Set t = {t} -- math. Mengen

  Invariante: - -- ggf. card s <= MAX)

  Signatur:

```
insert :: t -> {t} -> {t}       -- enter
delete :: t -> {t} -> {t}       -- remove
contains :: t -> {t} -> Bool    -- elem
card   :: {t} -> int            -- size
iterate :: {t} -> {t}           -- traverse
find :: t -> {t} -> t           -- search
```

  Semantik: naive Mengenlehre

Imperativ

```
interface Set {
 void insert (Object val);
  // requires: this.card < MAX
  // effects:  no effect, if this
  //           contains val,
  //           otherwise contains(val)
 ....
 boolean contains (Object val);
  //requires: -
  // effects: result: set semantics;
  //          no other effects
```


3.3.2 Mengenrepräsentation

- 3.3.2.1 Übersicht
  Lineare Repräsentationen
  - Listen
  - Feld
    Mittlere Laufzeit fast aller Operationen: O(n)
    Denn: das an i-ter Position befindliche Element zu finden braucht man i Zugriffe.
    Jedes der n Elemente wird mit gleicher Wahrscheinlichkeit gesucht. Damit im Mittel 1/n * S i , i=1..n = (n+1)/2 Zugriffe.
    Im Mittel muss etwa die Hälfte der Elemente durchsucht werden.
  - Falls Menge geordnet, bessere Algorithmen, z.B. binäre Suche.

Mengenrepräsentation

- Bitlisten

  Besonders effektiv, wenn Wertebereich beschränkt, ganzzahlig

```
bitlist = new Bitset(MAX+1);  // Java class lib
void insert (int val){
  //requires: 0 <= val <= MAX
 // effects: ...
bitlist.set(val);
}
boolean contains(int val){
  //...
  return bitlist.get(j);
}
```


Mengenrepräsentation

- Hashverfahren

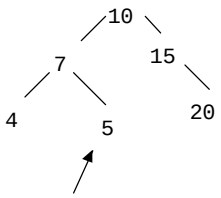
{30,63,15,11,97,19} ▸ **h(x) = x mod 10**

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

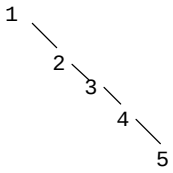
t	
0	30
1	11
2	-
3	63
4	-
5	15
6	-
7	97
8	-
9	19

Suchbäume

- Invarianzeigenschaft (für binären Suchbaum)
  Im linken Teilbaum befinden sich nur Werte, die kleiner gleich dem Knotenwert sind, im rechten Teilbaum nur solche, die größer sind.
  und jeder Teilbaum besitzt die Invarianzeigenschaft.

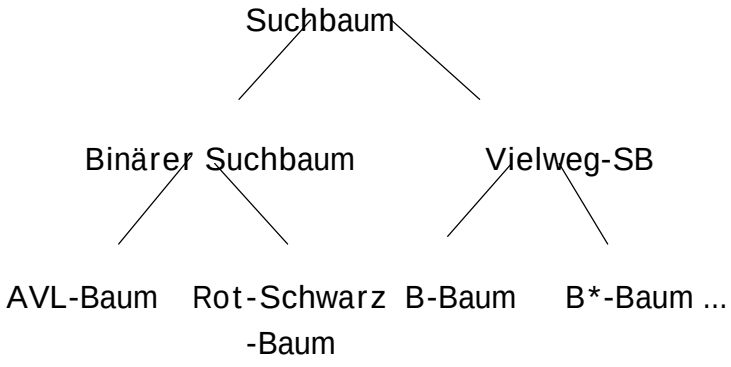


KEIN binärer Suchbaum



Binärer Suchbaum
Unschön (warum?)
"entarteter Baum"

Suchbäume



Fast ausgeglichen: Differenz der Blatttiefen begrenzt

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