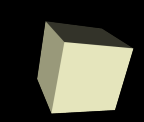
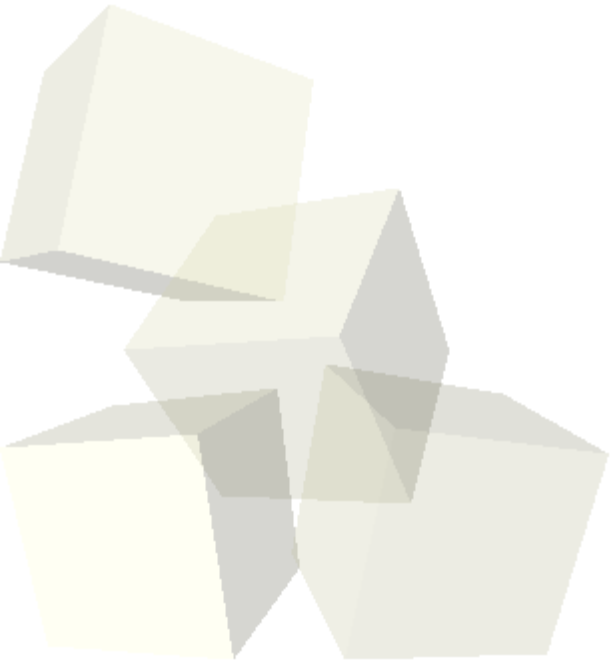


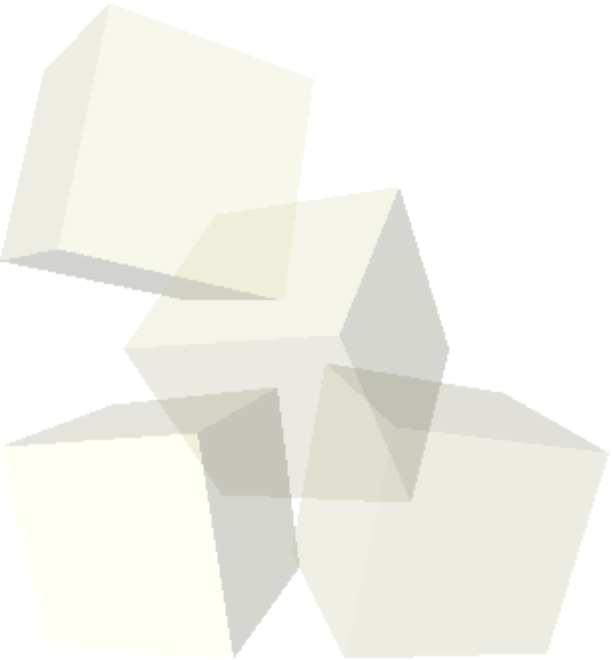
C-Schulung Tag 7

Herzlich willkommen



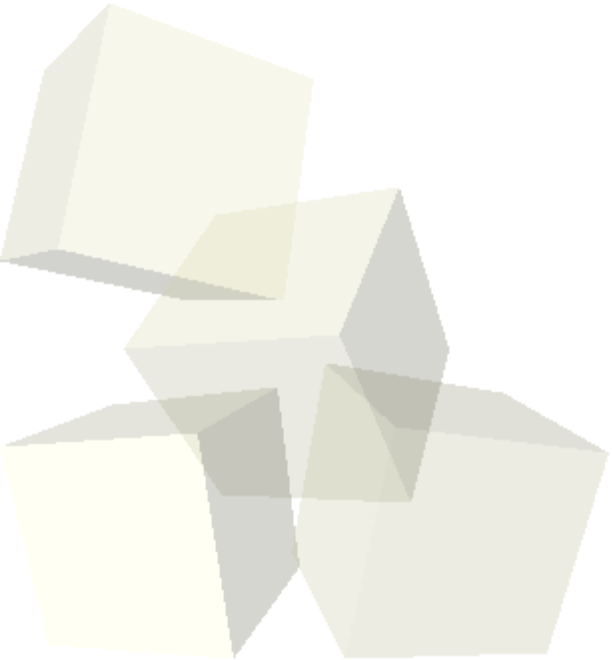
Agenda

- Listen als Mengen
- Binäre Suchbäume
- Hashverfahren



Motivation

- Wie viele unterschiedliche Wörter befinden sich im Roman *Moby Dick* von Herman Melville?
  - mobydick.txt



Listen als Mengen

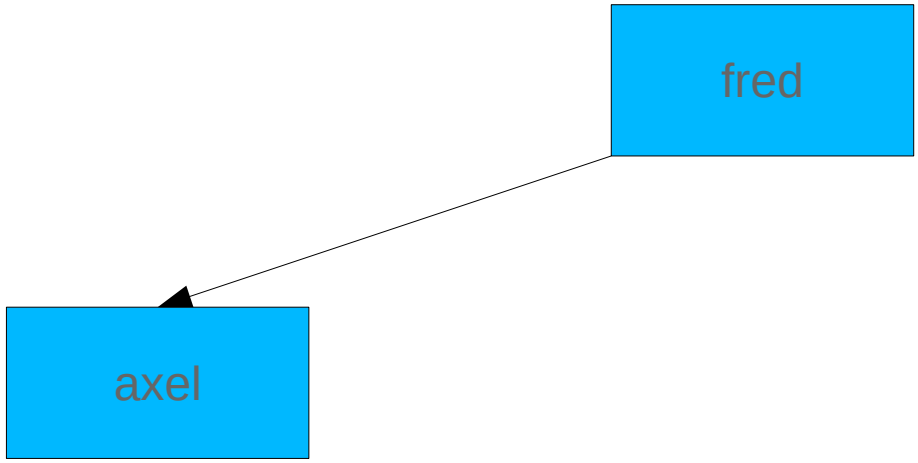
- In Java könnte man die Wörter einfach nacheinander in ein Set einfügen; Duplikate werden von Sets automatisch ignoriert.
- In C gibt es von Haus aus keine Sets; stattdessen können wir eine eigene Array-basierte Liste verwenden und vor jedem Einfügen manuell prüfen, ob das Wort schon enthalten ist.
- Performanceproblem: Für jedes einzufügende Wort muss die komplette Liste untersucht werden, weil das Wort überall in der Liste stehen könnte.
  - Komplexitätsklasse O(n)

Binäre Suchbäume

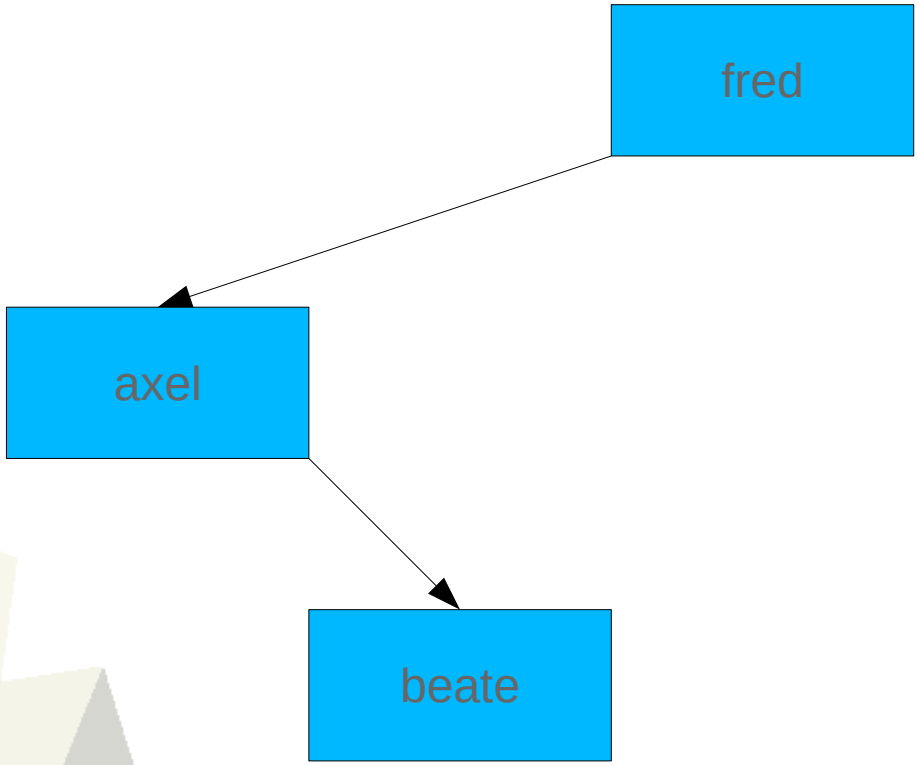
- Wir haben bereits gelernt, dass die binäre Suche deutlich schneller ist als die lineare Suche.
- Allerdings ist eine sortierte Liste keine geeignete Datenstruktur für eine veränderliche Menge, weil bei jedem Einfügen alle Wörter hinter dem eingefügten Wort verschoben werden müssten.
- Stattdessen verwendet man eine rekursive Baumstruktur, die aus verketteten Knoten besteht; dann muss beim Einfügen lediglich ein weiterer Knoten erzeugt und geeignet eingehängt werden.

Fred einfügen

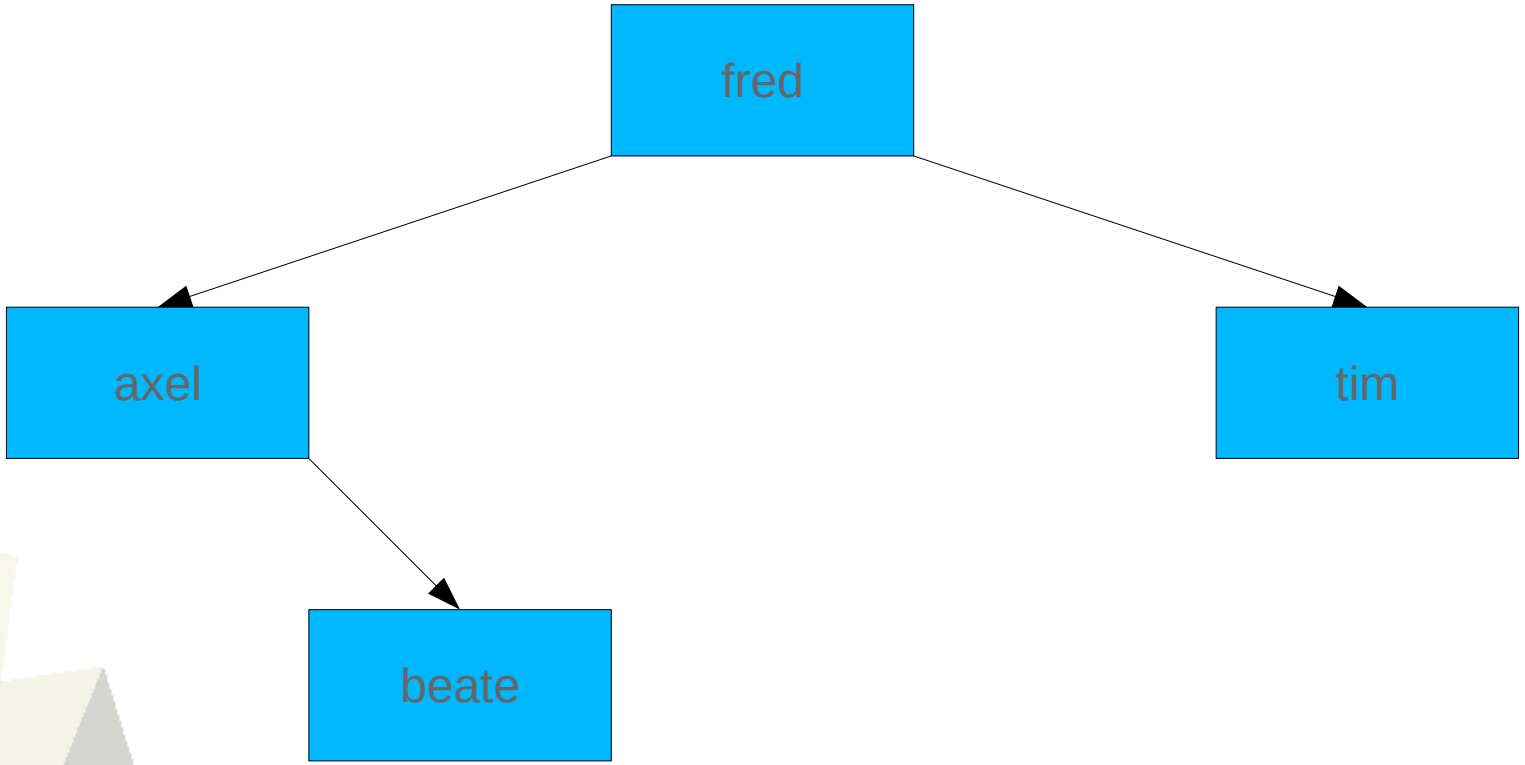
Axel einfügen



Beate einfügen



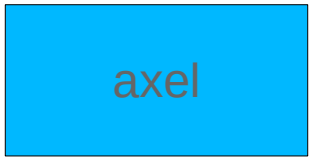
Tim einfügen



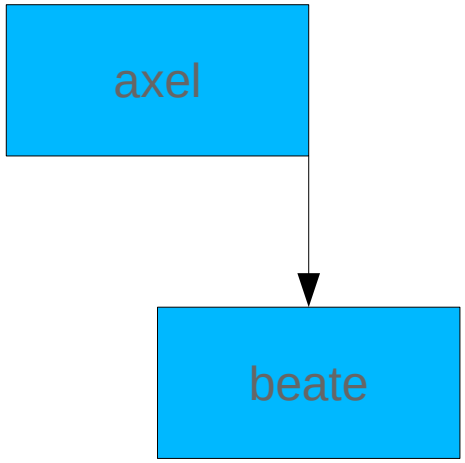
Entartung

- Die Effizienz der Suche eines Elements in einem binären Suchbaum hängt von der Tiefe des eingeschlagenen Suchpfads ab.
- Wenn alle Pfade ungefähr gleich lang sind, dann ist binäre Suche mit O(log n) recht effizient.
- Wenn man die Elemente in aufsteigender Reihenfolge einfügt, dann entartet der binäre Suchbaum zu einer Liste, und die Suche ist dann mit O(n) nicht mehr schneller als bei einer Liste.

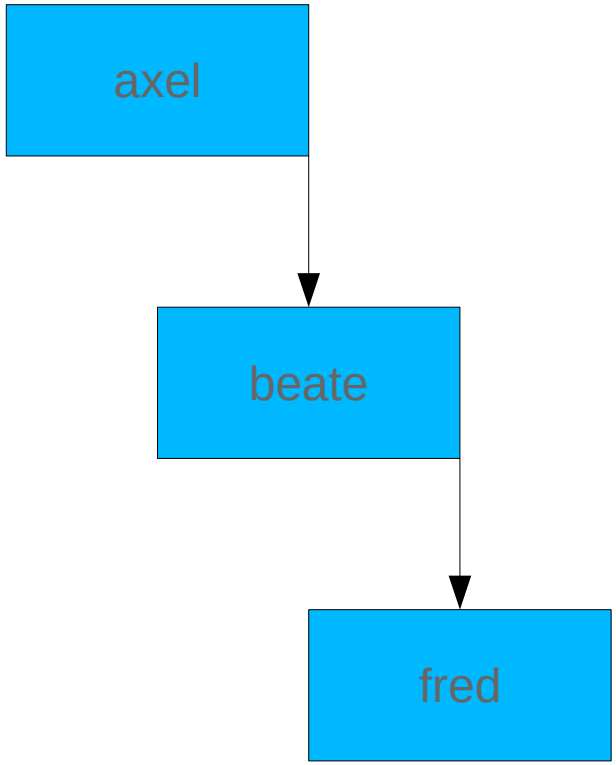
Axel einfügen



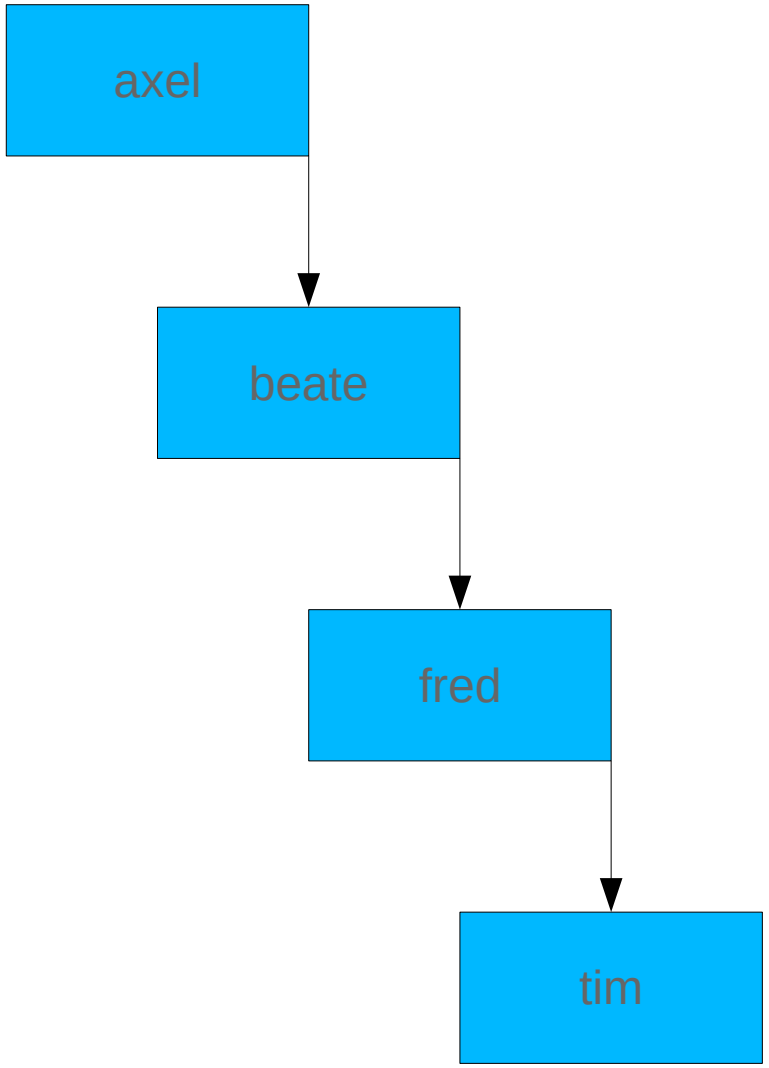
Beate einfügen



Fred einfügen



Tim einfügen



Balancierte Binäre Suchbäume

- Balancierte Binäre Suchbäume können nicht entarten, sind aber schwieriger zu implementieren.

- Zum Selbststudium:
  - https://de.wikipedia.org/wiki/Rot-Schwarz-Baum

Hashverfahren

- Die Grundidee von Hashing ist, dass aus jedem Element ein Hashcode berechnet werden kann:
  - hash("fred") = (('f' * 31 + 'r') * 31 + 'e') * 31 + 'd'
- Aus diesem Hashcode kann wiederum ein Index in ein Array (z.B. der Größe 5) berechnet werden:
  - index("fred") = hash("fred") % 5 = 3151467 % 5 = 2
- Idealerweise ist die Komplexität dann O(1).

0	1	2	3	4
tim	axel	fred	beate	

Kollisionen

- Falls mehrere Wörter auf denselben Index abgebildet werden, spricht man von Kollisionen.

- Kollisionen sind i.A. unvermeidbar. Es gibt zwei Möglichkeiten, mit Kollisionen umzugehen:
  - Offenes Hashing
  - Geschlossenes Hashing

0	1	2	3	4
tim	axel guido	fred heinz	beate julian	

Offenes vs. geschlossenes Hashing

- Beim *offenen Hashing* ist die Hashtabelle eine **Array von Wortlisten** (2D); alle Wörter mit demselben Index landen dann in derselben Liste.
  - Diese Listen sollten nicht zu lang werden.

- Beim *geschlossenen Hashing* ist die Hashtabelle ein **Array von Wörtern** (1D); bei einer Kollision wird einfach die nächste freie Stelle gesucht.
  - Je voller das Array wird, desto langsamer wird die Suche nach einer freien Stelle.
  - Außerdem ist das Entfernen eines Worts nicht-trivial.