Kapitel 11

Erweiterte und komplexere Datentypen

Fachgebiet Eingebettete Systeme und ihre Anwendungen
Prof. Dr. Andreas Koch

TECHNISCHE UNIVERSITÄT DARMSTADT

Inhalt von Kapitel 11
Erweiterte und komplexere Datentypen

11.1 Rekursive Datentypen

- Mengen und Listen
- Methodenoperationen auf Mengen

11.2 Weitere komplexere Strukturen

- Bäume
- Komplexere Strukturen

11.1 Rekursive Datentypen

```
public class Element {

     int info;
     Element naechster;

}

...

Element element = new Element();
element.info = 1;
element.naechster = new Element();
element.naechster.info = 2;
```

Erläuterung:

- Objekte einer Klasse können auch Verweise auf Objekte **derselben** Klasse als Variable enthalten.
- Eine solche Klasse nennt man *rekursiv*.

Das Beispiel „Menge“

```
public class Menge {

     private Element erstesElement;

     public int groesse ( ) { ... }
     public boolean istEnthalten ( int info ) { ... }
     public boolean fuegeEin ( int info ) { ... }
     public boolean entferne ( int info ) { ... }

}
```

Erläuterung:

- Prinzipiell gibt der Namen einer Methode schon Informationen darüber, was Sie genau tut.
- Das **`private`** anstelle des gewohnten **`public`** sorgt dafür, dass **`erstesElement`** nur von den Methoden der Klasse **`Menge`** angesprochen werden kann.

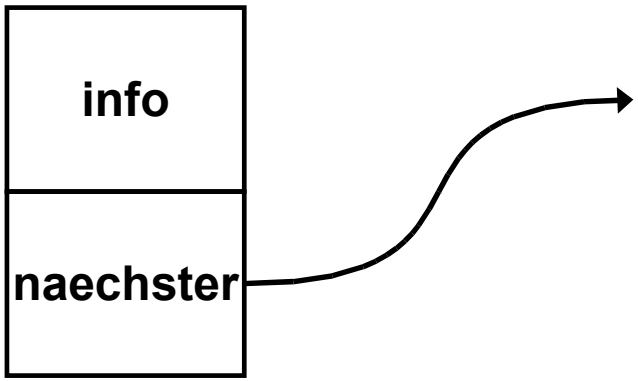
Methoden der Klasse Menge

- **`public int groesse()`**
  - Gibt die Anzahl der Elemente in der Menge zurück.
- **`public boolean istEnthalten (int info)`**
  - Überprüft, ob das Element **`info`** in der Menge enthalten ist.
- **`public boolean fuegeEin (int info)`**
  - Füge das Element **`info`** in die Menge ein.
    - Wenn **`info`** schon in der Menge enthalten ist, wird es nicht noch einmal eingefügt.
    - Der Rückgabewert ist **`true`**, wenn **`info`** noch nicht in der Menge enthalten ist (und somit eingefügt wird).

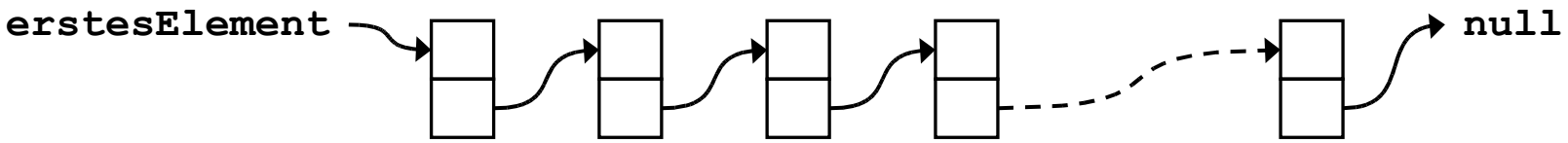
Methoden der Klasse Menge

- **`public boolean entferne (int info)`**
  - Entferne **`info`** aus der Menge
    - Wenn **`info`** nicht in der Menge enthalten ist, wird es natürlich nicht entfernt.
    - Der Rückgabewert ist **`true`**, wenn **`info`** in der Menge enthalten war (und somit entfernt wurde).

- Visualisierungsmöglichkeit eines Objekts der Klasse **`Element`**



Schematische Darstellung



Die einzelnen Elemente der Menge bilden eine Art Kette, die durch Verweis **`naechster`** zusammengehalten wird.

- Das Ende dieser Kette wird sinnvollerweise durch **`naechster==null`** angezeigt.

  ➔ Da für den letzten Wert der Kette noch kein Nachfolge-Objekt angelegt worden ist

- Eine Kette dieser Art heißt in der Informatik ***Liste***.

Methode `groesse`

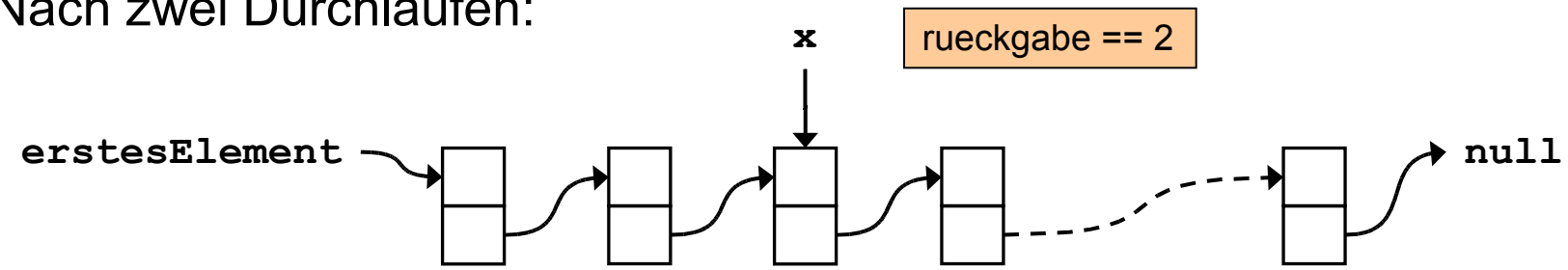
```
public int groesse (){
     Element x = erstesElement;
     int rueckgabe = 0;
     while ( x != null ){
        x = x.naechster;
        rueckgabe++;
     }
     return rueckgabe;
}
```

Bis **x** das letzte **Element** erreicht hat

x hüpft von einem **Element** zum Nächsten

Bei jedem Hüpfer wird **rueckgabe** um 1 erhöht

Nach zwei Durchläufen:



Methode `istEnthalten`

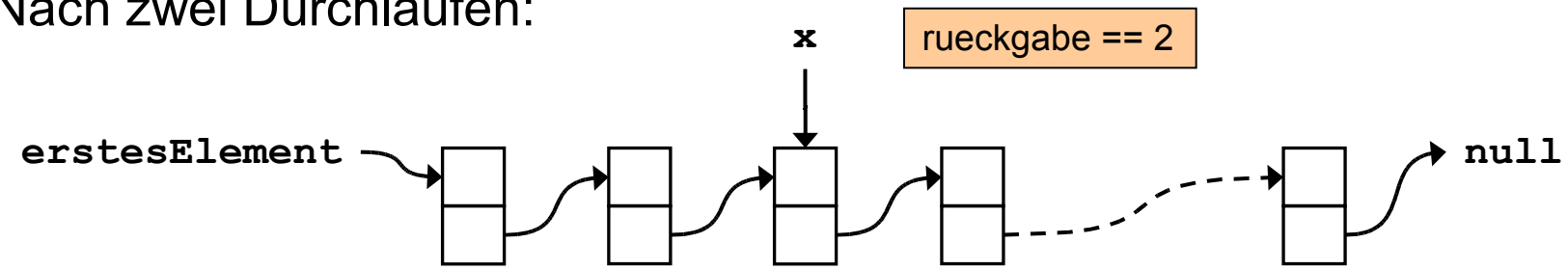
```
public boolean istEnthalten ( int info ){
     Element x = erstesElement;
     while ( x != null ){
        if ( x.info == info )
                return true;
        x = x.naechster;
     }
return false;
}
```

Bis **x** den gesuchten Wert **info** enthält

x hüpft von einem **Element** zum Nächsten

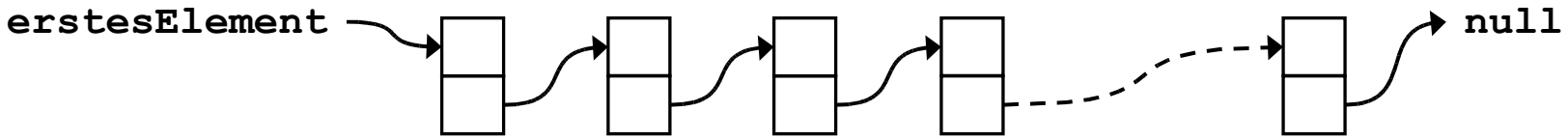
oder der gesuchten Wert **info** nicht gefunden wurde (Ende der Liste)

Nach zwei Durchläufen:

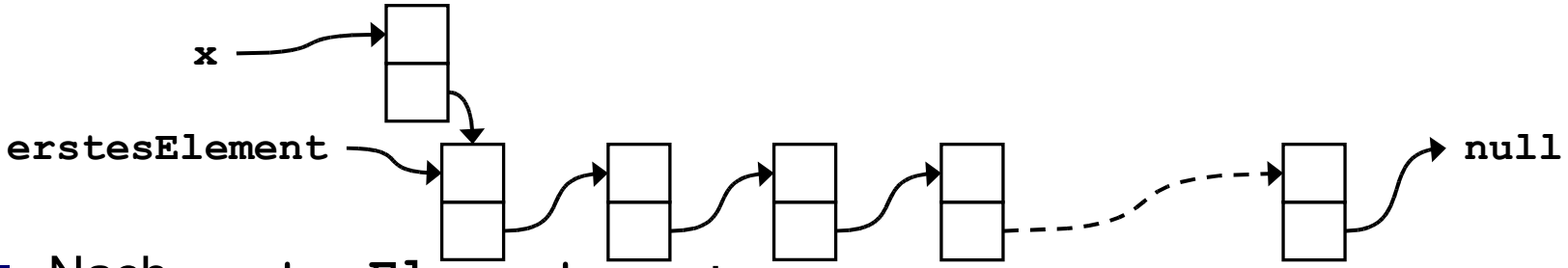


Methode `fuegeEin` Veranschaulichung

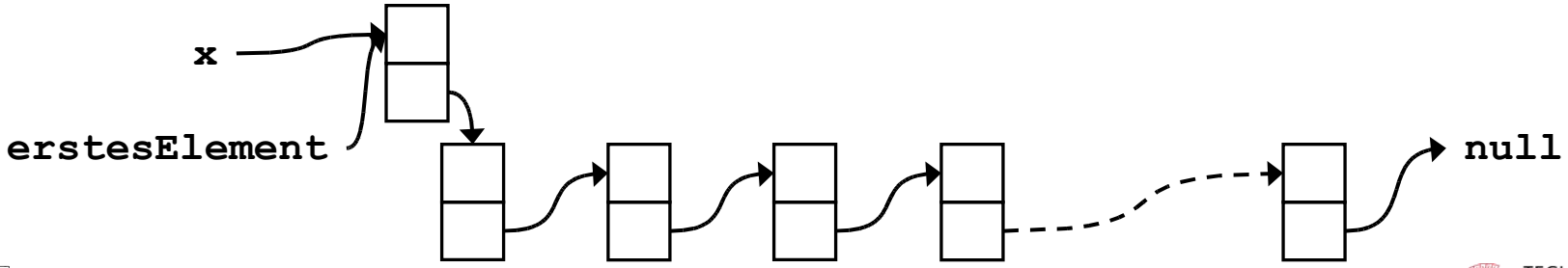
- Ausgangsliste:



- Nach `x.naechster = erstesElement`:



- Nach `erstesElement = x`:



Methode `fuegeEin`

```
public boolean fuegeEin (int info){

     if ( istEnthalten(info) )
        return false;
     Element x = new Element();
     x.info = info;
     x.naechster = erstesElement;
     erstesElement = x;
     return true;
}
```

wenn **`info`** enthalten, wird nichts getan

ansonsten Anlegen ein neuen Elementes `x`

das **`info`** enthält
und auf das allererste Element zeigt

`x` wird dann zum neuen allerersten Element

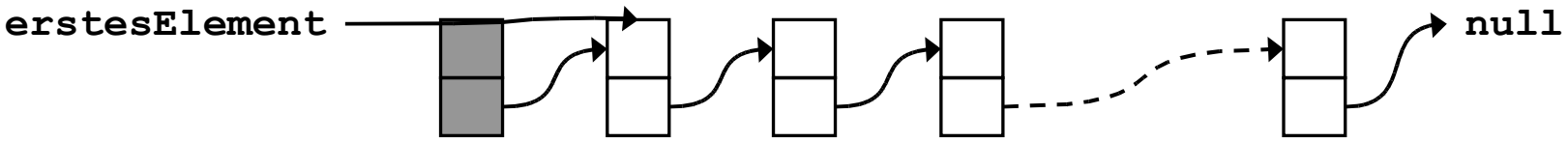
Methode `entferne`

- Die Idee ist, das Element mit der gesuchten **`info`** einfach aus der Liste auszukoppeln.
  - Hinterher gibt es dann keinen Verweis mehr auf dieses Element.
  - *Erinnerung*: Ein solches Element wird vom ***Garbage Collector*** aus dem Speicher entfernt.
- *Methodisches Problem*:
  - Um ein Element aus der Liste zu entkoppeln, muss man Komponente ”**`naechster`** seines Vorgängers” ändern.
  - Beim Durchlauf durch die Liste, um das Element zu finden, muss man also immer um ein Element zurückbleiben.
  - Falls das zu löschende Element das allererste ist, geht die Entkopplung ganz einfach:

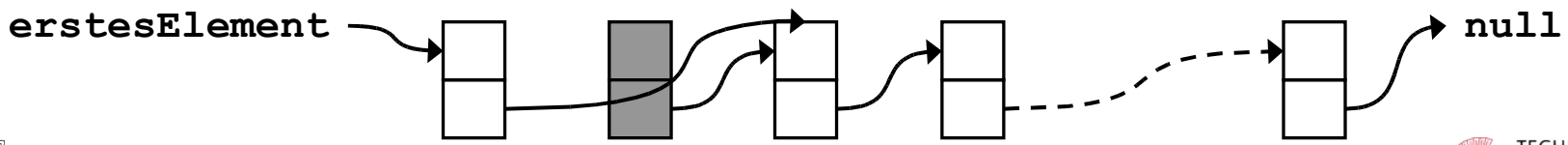
```
erstesElement = erstesElement.naechster;
```

Methode `fuegeEin` Veranschaulichung

- Fall 1: Element ist das erste in der Liste
  - Nach **`erstesElement = erstesElement.naechster`** im Fall **`erstesElement.info == info`** :



- Fall 2: Element ist eines innerhalb der Liste
  - Nach **`x.naechster = x.naechster.naechster`** im Fall **`x.naechster.info == info`**
  - hier schon im ersten Durchlauf der **`while`**–Schleife



Methode `entferne`

```
public boolean entferne ( int info ){
     if ( erstesElement == null )
        return false;
     if ( erstesElement.info == info ) {
        erstesElement = erstesElement.naechster;
        return true;
     }
     Element x = erstesElement;
     while ( x.naechster != null ) {
        if ( x.naechster.info == info ){
              x.naechster = x.naechster.naechster;
              return true;
        }
        x = x.naechster;
     }
     return false;
}
```

aus einer leeren Liste kann man nichts entfernen

erstes Element entfernen

Iteriere über alle nächsten Elemente

Wenn das nächste das gesuchte ist

wird das übernächste Element zum nächsten gemacht

Andere mögliche Methoden

Es sind noch weitere Methoden denkbar, die auf Listen oder Mengen operieren:

- Vermischen zweier Listen: **`merge`** bzw. **`union`**
- Schnittmenge zweier Mengen: **`intersect`**
- Eine Liste an eine andere hängen: **`append`**
- Eine Liste sortieren: **`sort`**
- Eine Liste ausgeben: **`print`**

Bäume

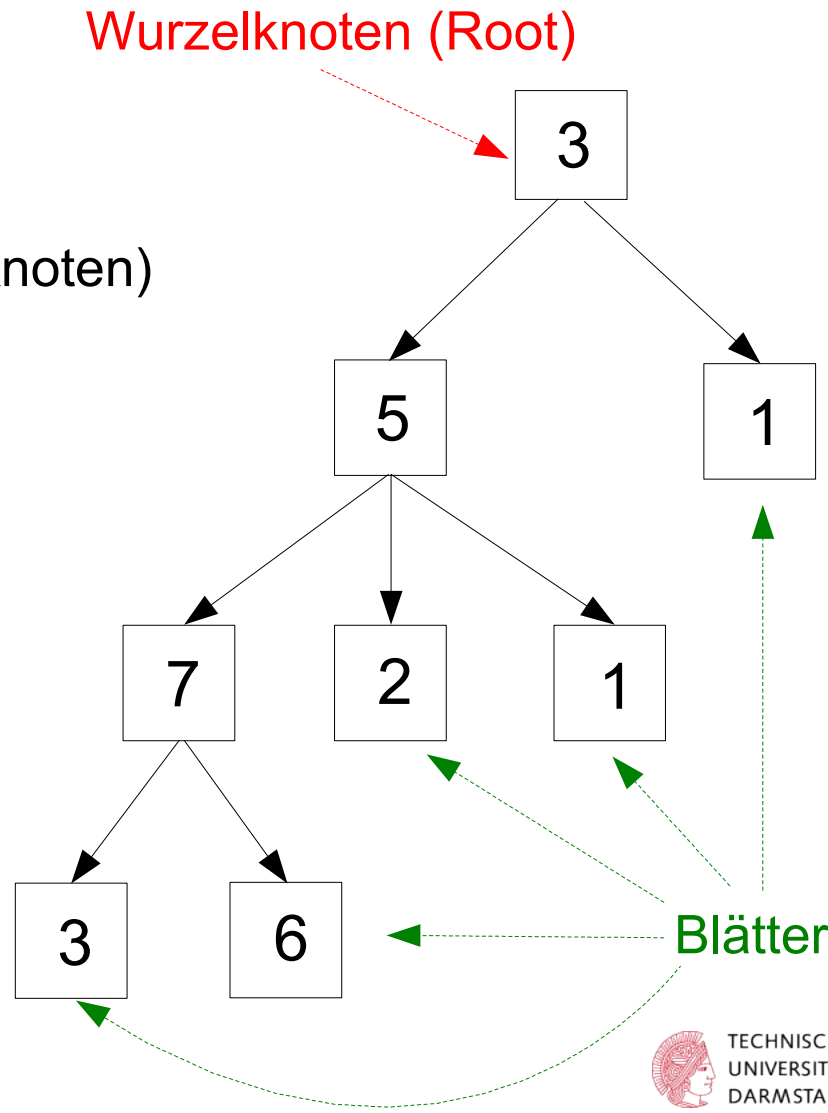
Datenstrukturen, bestehend aus

- Knoten (enthalten Informationen)
- Kanten (Verbindungen zwischen Knoten)

Einschränkungen

- jeder Knoten mehrere Nachbarn haben kann
- keine Zyklen erlaubt sind
  (dann wären es allgemeine Graphen)

Anm: Die Information hat in diesem Beispiel keine besondere Bedeutung



Binäre Bäume

Einschränkung:

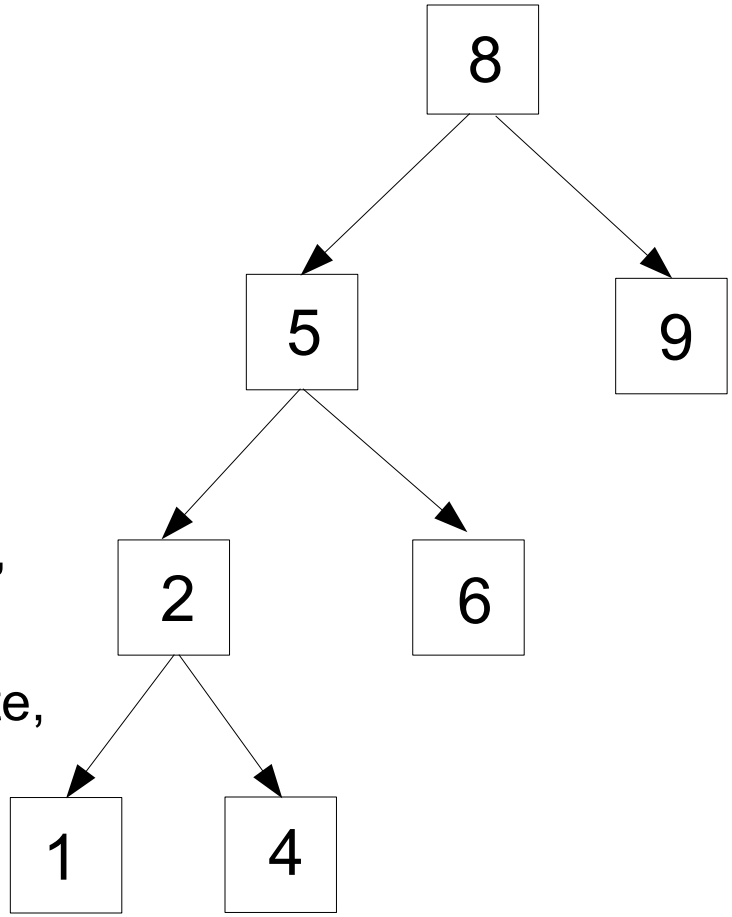
- jeder Knoten hat 2 Nachfolge-Knoten (ausgenommen Blätter)

Zahlreiche Anwendungen in der Informatik

z.B. Sortierbäume:

- der linke Teilbaum enthält nur Elemente, die kleiner sind als die Wurzel
- der rechte Teilbaum enthält nur Elemente, die größer sind als die Wurzel

Das gilt rekursiv für jeden Teilbaum!



Realisierung

```
public class Baum {

    // Datenkomponenten
    private int info;
    private Baum linkerTeilbaum;
    private Baum rechterTeilbaum;

    // Konstruktor
    public Baum(int info) {
       this.info = info;              // Setzen der info-Komponente
       linkerTeilbaum = null;         // Initialisierung der beiden
       rechterTeilbaum = null;        // Teilbaeume mit null-Referenz
    }

    // Methoden
    public int groesse ( ) { ... }
    public boolean istEnthalten ( int info ) { ... }
    public void fuegeEin ( int info ) { ... }
    public void entferne ( int info ) { ... }
}
```

Implementierung

Die Implementierung der Methoden richtet sich nach dem jeweiligen Anwendungszweck

- z.B. bei einem sortierten Binärbaum muß darauf geachtet werden, daß eine neue **`info`**-Komponente an der richtigen Stelle des Baums eingefügt wird.
- Das ist ein gutes Beispiel dafür, daß man die Datenkomponenten nicht direkt zugreifbar macht (also **`private`**), damit keine Unterklasse durch ein falsches Einfügen die Sortierung zerstören kann.
  - Einfügen eines Knotens ist nur mit Methode **`fuegeEin`** möglich, die darauf achtet, daß die Sortierordnung erhalten bleibt.

Implementierung `fuegeEin` für sortierten Baum

```
public void fuegeEin ( int info )
   // Entscheidung ob im linken oder im rechten Teilbaum
   if (this.info < info) {
      // Sind wir an einem Blatt angelangt?
      if (rechterTeilbaum == null) {
         // dann einen neuen Knoten anlegen
         rechterTeilbaum = new Baum(info);
      } else {
         // ansonsten im rechten Teilbaum einfügen
         rechterTeilbaum.fuegeEin(info);
      }
   }
   else {
      ...
      // analog für den linken Teilbaum
   }
}
```

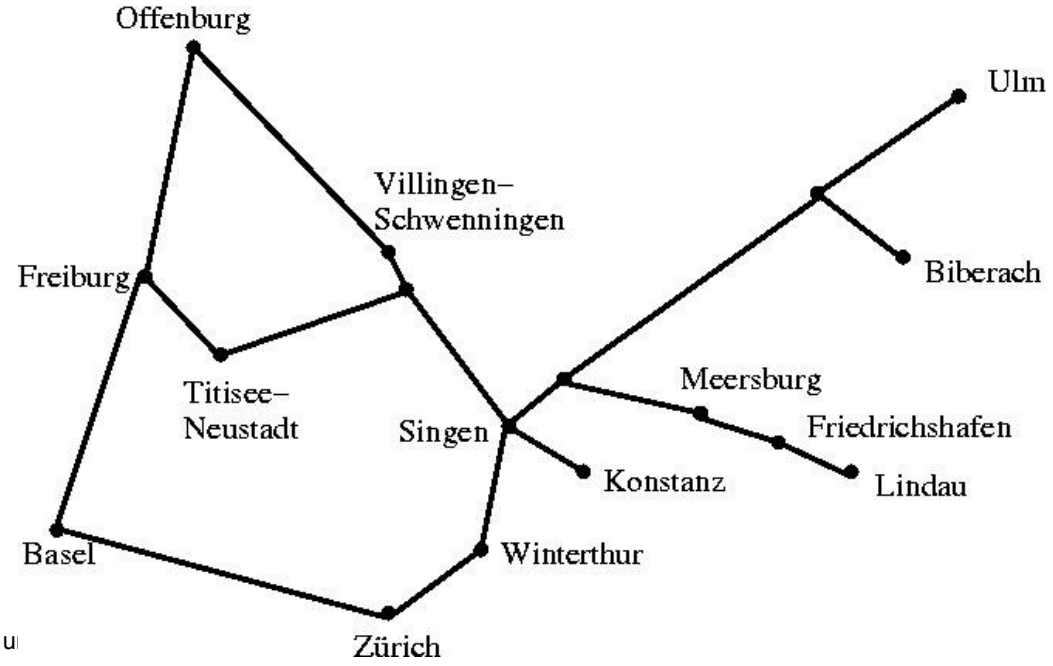
Rekursiver Aufruf

Komplexere Strukturen

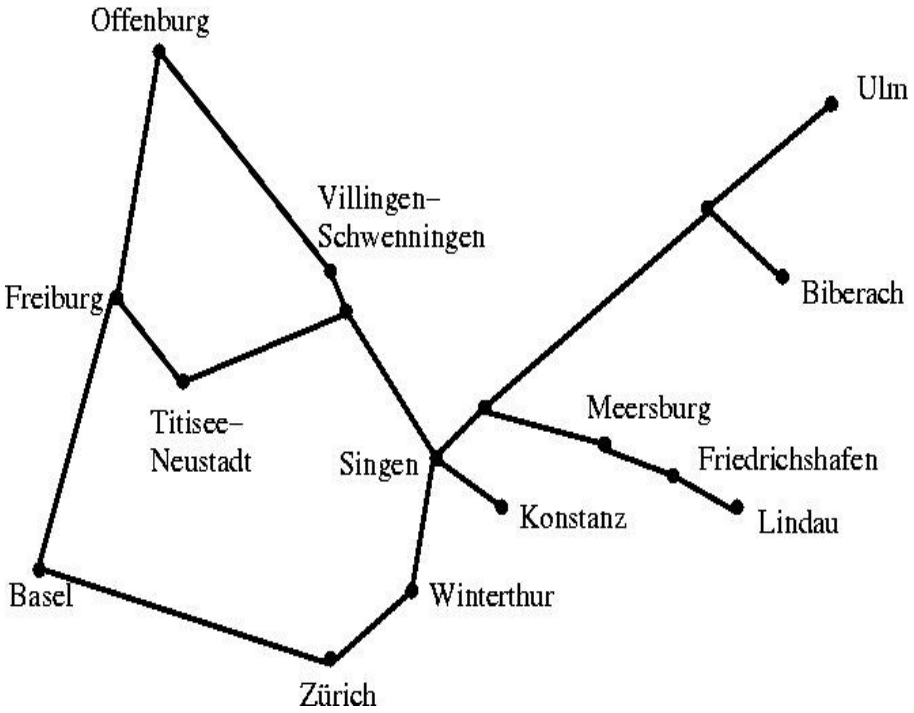
Beispiel: Autobahnnetz

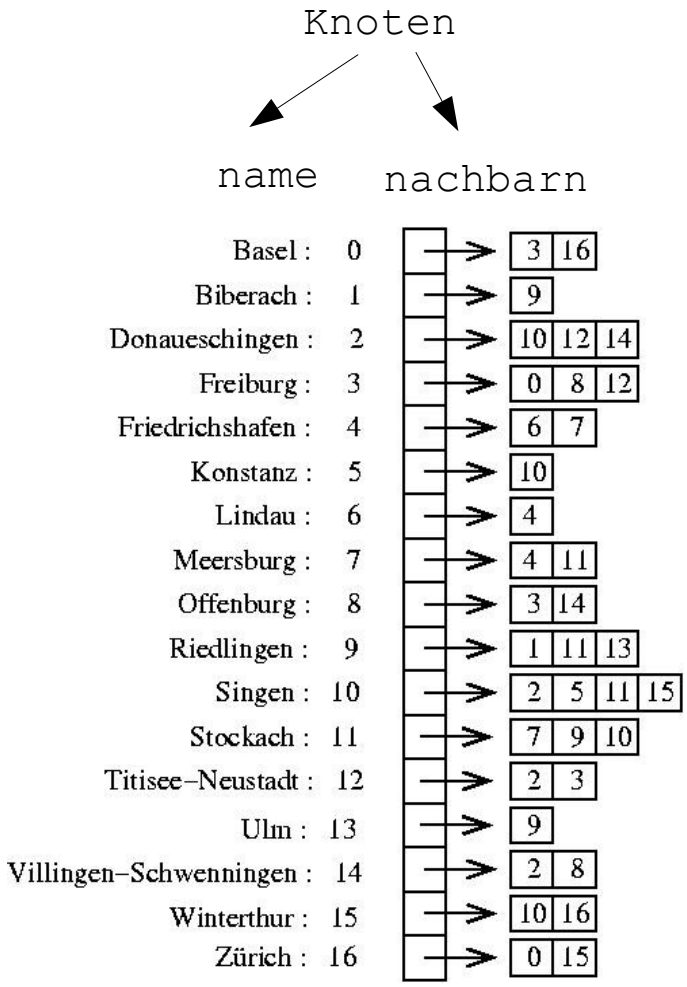
Idee:

- ◊ Ein Array von Knotenpunkten
- ◊ Für jeden Knotenpunkt eine Liste von Nachbarknotenpunkten.



Veranschaulichung





Veranschaulichung

```
public class Knoten
{
   public String name;
   public Menge nachbarn;
}

// Verwendung, z.B. in main
   Knoten[] autobahnnetz = new Knoten[17];
   autobahnnetz[0] = new Knoten();
   autobahnnetz[0].name = "Basel";
   autobahnnetz[0].nachbarn.fuegeEin(3);
   autobahnnetz[0].nachbarn.fuegeEin(16);

   autobahnnetz[1] = new Knoten();
   autobahnnetz[1].name = "Biberach";
   autobahnnetz[1].nachbarn.fuegeEin(9);
   ...
}
```