Informatik 2

Übungsblatt 10

Ablauf der Übungsstunde

Werkzeugkasten
Wiederholung des für die Übungsaufgaben relevanten Stoffs

Übungsserien
Teilweise Lösung der aktuellen Übungsserie an der Tafel

Prüfungsvorbereitung
Besonderer Schwerpunkt auf für die Prüfung relevante Aufgaben

Präsenzstunde

diese Woche
im HIL F15.4

Ziel der heutigen Übungsstunde

Themen

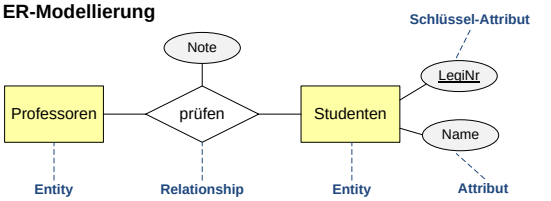
- Verstehen wie man Daten modelliert
- Entity-Relationship (ER) Diagramm zeichnen können
  - Funktionalitäten

Wichtig für die Prüfung

- Entity-Relationship Diagramm für gegebenes Szenario aufzeichnen können
- Funktionalitäten korrekt einsetzen

Werkzeugkasten für aktuelle Serie

ER-Modellierung

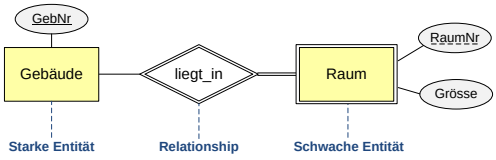


Was „kennt“ ER?

- **Entities (Gegenstände)**
- **Relationships**
- **Attribute und Rollen**

Werkzeugkasten für aktuelle Serie (2)

Existenzabhängige (schwache) Entitytypen



Merke

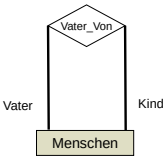
- **Schwache Entitäten** können nicht ohne **starke Entitäten** leben und sind nur mit starken Entitäten eindeutig identifizierbar

Funktionalitäten

- Trick zum besseren Verständnis: bei 1:1 oder 1:N Funktionalität immer die (partiellen) Funktionen aufschreiben
- Richtung der Funktion immer "zur 1"

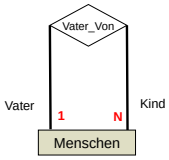
Funktionalitäten

- Welche Funktionalitäten?



Funktionalitäten

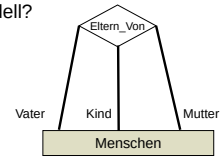
- Welche Funktionalitäten?



Vater_von: Kind: Menschen → Vater: Menschen

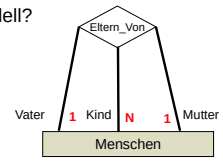
Funktionalitäten

- Welche Funktionalitäten?
- Gutes Modell?



Funktionalitäten

- Welche Funktionalitäten?
- Gutes Modell?



Eltern_Von:
Kind: Menschen × Vater: Menschen → Mutter: Menschen
Kind: Menschen × Mutter: Menschen → Vater: Menschen

Problem:
(Anna, Bernhard) ↦ Clara
(Anna, Doris) ↦ Emil

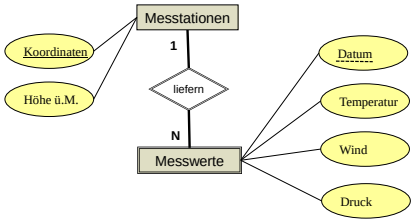
→ Inkonsistenz möglich
(Bernhard, Anna, Clara),
(Emil, Anna, Doris)

Beispiel Messstationen

Modelliere

- Messstationen mit bekannter Lage (Koordinaten und Höhe ü.M.) liefern Messwerte Temperatur, Wind und Druck zu bestimmten Zeiten

Beispiel Messstationen



Aktuelle Serie

Discussion Assignment 9

Felix Friedrich, Lars Widmer
TA lecture, Informatics II D-BAUG
May 7, 2014

Outline

Recursion Repeated

- **Recursion** is when a method **calls itself**. It can be puzzling but in the end it's **simple** and very **powerful**.
- It's very **natural** to use recursion in order to **traverse a tree**. Every recursion brings us a level further down on the current branch. So, that's what we choose for the example solution in the following slides.
- But first let's have a very quick look at recursion itself.

Recursive Example

```
1  private static void callMyself(int n) {
2      System.out.print(n+"␣");
3      if (n > 0) {
4          callMyself(n-1);
5      }
6  }
7
8  public static void main(String[] args) {
9      callMyself(5);
10 }
```

What's the output? → 5 4 3 2 1 0

Example Execution, *starting with 2*

```
1  callMyself(2) {
2      System.out.print("2␣");
3      if (2 > 0) {
4          callMyself(2-1);
5          // ...
6  }   }
```


Example Execution, *starting with 2*

```
1  callMyself(2) {
2      System.out.print("2␣");
3      if (2 > 0) {
4          callMyself(2-1);
5          ==> callMyself(1) {
6              System.out.print("1␣");
7              if (1 > 0) {
8                  callMyself(1-1);
9                  // ...
10 }   }   }   }
```


Example Execution, *starting with 2*

```
1  callMyself(2) {
2      System.out.print("2␣");
3      if (2 > 0) {
4          callMyself(2-1);
5          ==> callMyself(1) {
6              System.out.print("1␣");
7              if (1 > 0) {
8                  callMyself(1-1);
9                  ==> callMyself(0) {
10                     System.out.print("0␣");
11                     if (0 > 0) { // false
12                         // stop therefore
13 }   }   }   }   }   }
```


Outline

TreeElement, *our data structure*

```
1  public class TreeElement
2  {    public String data;
3       public TreeElement lower;
4       public TreeElement higher;
5       public TreeElement(String str)
6       {    data = str;
7            lower = null;
8            higher = null;
9       }
10      public String toString()
11      {    return data+"␣";
12      }
13 }
```


Traverse *in Order* (for toString)

```
1  private String traverseInOrder(TreeElement iter) {
2      if (iter == null) {
3          return "";
4      }
5      return traverseInOrder(iter.lower)
6         +iter+traverseInOrder(iter.higher);
7  }
8
9  public String toString() {
10     return traverseInOrder(root); // start from top
11 }
```


Traverse *in Order* Explained

```
1  private String traverseInOrder(TreeElement iter) {
2      if (iter == null) {
3          return "";
4      }
5      return traverseInOrder(iter.lower)
6         +iter+traverseInOrder(iter.higher);
7  }
```

- Each call we do two recursions, one for the `lower` branch, one for the `higher`.
- When we arrive at a `null`-reference, we're done for this branch.
- To get the right order, we concatinate the result of the `lower` branch before our `data` of the current node and at the end we put the result of the `higher` branch.

Locate the Node where to `Add`

```
1  private void followToAdd(TreeElement iter,
2                           TreeElement newTE) {
3
4      // compare the two elements:
5      int comp = newTE.data.compareTo(iter.data);
6      // ...
7              followToAdd(iter.higher, newTE);
8              // ...
9              followToAdd(iter.lower, newTE);
10             // ...
11 }
```

- When traversing the tree in order to `add` a new element, we don't have to traverse the whole tree.
- At every node we can decide on which further branch the new element belongs. The other branch can be neglected.
- When we arrive at a `null`-reference, we found the place to actually place the element.

Locate the Node where to `Add`

```
1  private void followToAdd(TreeElement iter,
2                           TreeElement newTE) {
3
4      // compare the two elements:
5      int comp = newTE.data.compareTo(iter.data);
6
7      if (comp == 0) { // Check for duplicates
8          return; // found duplicate
9      }
10
11     // Carry on if the new element is higher ...
12     // Carry on if the new element is lower ...
13 }
```


Locate the Node where to `Add`

```
1  // ...
2  // Carry on if the new element is higher:
3  if (comp > 0) { // higher
4      if (iter.higher == null) {
5          iter.higher = newTE;
6      } else { // not free => recursion
7          followToAdd(iter.higher, newTE);
8      }
9  } else { // lower
10     // ...
```


Locate the Node where to `Add`

```
1      // ...
2      // Carry on if the new element is lower:
3      } else { // lower
4          if (iter.lower == null) {
5              iter.lower = newTE;
6          } else { // not free => recursion
7              followToAdd(iter.lower, newTE);
8          }
9      }
10 }
```


`Add` Element

Using the traversing method, adding now is quite simple.

```
1  public void add(String str) {
2      TreeElement newTE = new TreeElement(str);
3      if (root == null) {
4          root = newTE;
5      } else {
6          followToAdd(root, newTE);
7      }
8  }
```

Traverse for Maximum Depth

We use a recursive helper method to determine the maximum depth of the tree.

```
1 public int depth() {
2     return traverseForMaxDepth(root, 0);
3 }
```


Traverse for Maximum Depth

```
1  private int traverseForMaxDepth
2          (TreeElement iter, int level) {
3      if (iter == null) {
4          return level;
5      }
6      ++level;
7      return Math.max(
8          traverseForMaxDepth(iter.lower, level),
9          traverseForMaxDepth(iter.higher, level));
10 }
```


Outline

Extending `Highscore`, Class Structure

```
1 public class FiledHighscore extends Highscore {
2
3     public FiledHighscore(int s) {
4         super(s);
5     }
6
7     // ...
8
9 }
```


Class `FiledHighscore`, Method `store`

```
1  public void store() {
2      try {
3          BufferedWriter out = new BufferedWriter
4              (new FileWriter("highscore.txt"));
5          for (Player p : highscore) {
6              out.write(p.score);
7              out.write(p.name);
8              out.write('\n');
9          }
10         out.close();
11     } catch (IOException e) {
12         e.printStackTrace();
13     }
14 }
```


Class `FiledHighscore`, Method `load`

```
1  public void load() {
2      try {
3          // open file, read all, close
4          // ...
5      } catch (IOException e) {
6          e.printStackTrace();
7      }
8  }
```

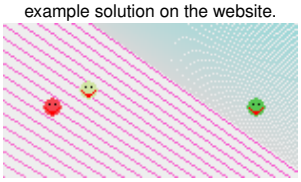

Method `load`

```
1  // open file, read all, close
2  BufferedReader in = new BufferedReader(
3          new FileReader("highscore.txt"));
4  highscore.clear(); // empty current list
5  while (in.ready()) {
6      int score = in.read();
7      String name = in.readLine();
8      Player p = new Player(name,score);
9      highscore.add(p);
10 }
11 in.close();
```


Outline

Bouncing Smileys!

You can find the full project for this example solution on the website.



If the authors agree, we might also publish **cool student projects** on the website.

Outline

Feel free ...

Ladies & Gentlemen!
... Please name any

- Questions?
- Feedback?
- Additions?
- Remarks?
- Wishes?
- ...

All the Best!