



hochschule mannheim

Fakultät für Informatik Programmierung 2 (PR2)

06 - Datenstrukturen

Prof. Dr. Frank Dopatka



Hochschule Mannheim University of Applied Sciences



hochschule mannheim



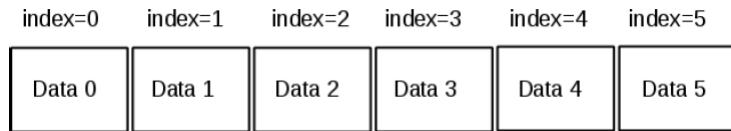
Verkettete Listen



Was haben wir bislang?
Arrays!



- Ein Array ist Speicher eines vorgegebenen Datentyps.
- Jedes Speicherelement bekommt einen Index, über den man das Speicherelement abrufen oder beschreiben kann.
- Die Größe eines Arrays wird bei seiner Erstellung definiert und ist dann unveränderlich.
- Der lesende und schreibende Zugriff auf ein Array-Element ist unglaublich schnell mit $O(1)$.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

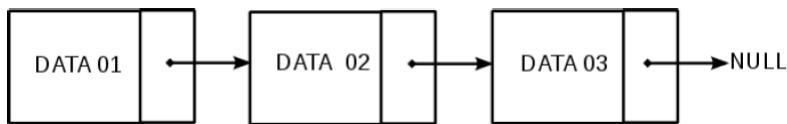
3



Was ist eine verkettete Liste?



- Eine verkettete Liste ist eine Kette von Speicherelementen eines vorgegebenen Datentyps, die jeweils auf ihren Nachfolger zeigen.
- Bei einer doppelt verketteten Liste zeigt ein Speicherelement jeweils sowohl auf seinen Vorgänger, als auch auf seinen Nachfolger.
- Möchte man ein Element finden, so muss man die Kette vom Anfang aus, über die Zeiger der Elemente, bis zu dem zu suchenden Element entlang hangeln:



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

4





hochschule mannheim
Array vs. Liste

- Vorteile einer Liste:
 - Elemente können sehr schnell am Anfang der Liste eingefügt werden.
 - Speicherbedarf nur wenig höher als bei einem Array.
 - Im Gegensatz zu einem Array ist das Einfügen und Löschen problemlos möglich, ohne dass der komplette Datensatz bei jeder Vergrößerung umkopiert werden muss.
- Nachteile einer Liste:
 - Es ist aufwändig, nach Daten zu suchen und die Liste zu sortieren, da über jedes einzelne Element gegangen (iteriert) werden muss.
 - Das Einfügen an der ersten und der letzten Stelle muss gesondert behandelt werden.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

5



hochschule mannheim
Eine einfach verkettete Liste:
Datenknoten

```
public class Knoten {  
    private Object o;  
    private Knoten nächster;  
  
    public Knoten(Object o, Knoten nächster){  
        this.setData(o);  
        this.setNext(nächster);  
    }  
    public Object getData() {  
        return o;  
    }  
    public void setData(Object o) {  
        this.o = o;  
    }  
}
```



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

6


hochschule mannheim
Eine einfach verkettete Liste:
Datenknoten



```

public Knoten getNext() {
    return nächster;
}
public void setNext(Knoten nächster) {
    this.nächster = nächster;
}
@Override
public String toString(){
    if (this.getData()==null) return null;
    return this.getData().toString();
}
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

7


hochschule mannheim
Eine einfach verkettete Liste:
Die Liste selbst



```

public class EinfachVorketteteListe {
    private Knoten wurzel;

    public EinfachVorketteteListe(Object o){
        wurzel=new Knoten(o,null);
    }

    public Object getFirst(){
        if (wurzel==null) return null;
        return wurzel.getData();
    }

    public Object get(int pos){
        int anz=this.getAnzKnoten();
        if ((pos<1)|| (pos>anz))
            throw new RuntimeException("Position ungültig!");
        Knoten k=wurzel;
        for (int i=1;i<pos;i++){
            k=k.getNext();
        }
        return k.getData();
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

8

 hochschule mannheim

Eine einfach verkettete Liste: Die Liste selbst



```
public int getAnzKnoten(){
    if (wurzel==null) return 0;
    Knoten k=wurzel;
    Knoten nächster;
    int anz=0;
    do{
        nächster=k.getNext();
        k=nächster;
        anz++;
    }while (nächster!=null);
    return anz;
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

9

 hochschule mannheim

Eine einfach verkettete Liste: Die Liste selbst

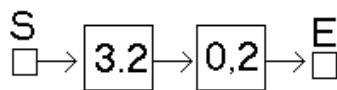
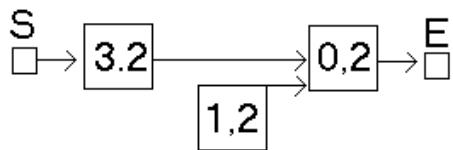
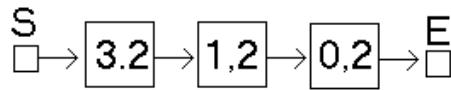


```
public Object[] getAll(){
    int anz=this.getAnzKnoten();
    if (anz==0) return new Object[0];
    Object[] array=new Object[anz];
    array[0]=wurzel.getData();
    Knoten k=wurzel.getNext();
    for (int i=1;i<anz;i++){
        array[i]=k.getData();
        if (k!=null) k=k.getNext();
    }
    return array;
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

10


**Eine einfach verkettete Liste:
Die Liste selbst – Knoten löschen**

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 11


**Eine einfach verkettete Liste:
Die Liste selbst – Knoten löschen**


```

public void removeFirst() {
    if (wurzel!=null) wurzel=wurzel.getNext();
}

public void removeLast() {
    if (wurzel==null) return;
    if (wurzel.getNext()==null)
        wurzel=null;
    else{
        Knoten k=wurzel;
        Knoten nächster=null;
        Knoten vorheriger=null;
        do{
            nächster=k.getNext();
            if (nächster!=null){
                vorheriger=k;
                k=nächster;
            }
        }while (nächster!=null);
        vorheriger.setNext(null);
    }
}
  
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 12


hochschule mannheim
Eine einfach verkettete Liste:
Die Liste selbst – Knoten löschen

```

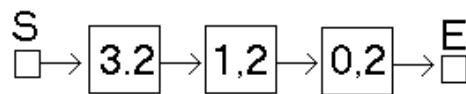
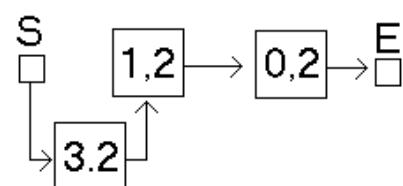
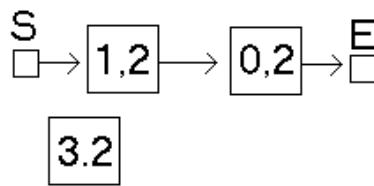
public void remove(int pos){
    int anz=this.getAnzKnoten();
    if ((pos<1)|| (pos>anz)) return;
    if (pos==1){
        this.removeFirst(); return;
    }
    if (pos==anz){
        this.removeLast(); return;
    }
    Knoten k=wurzel;
    Knoten nächster=null;
    Knoten vorheriger=null;
    for (int i=1;i<pos;i++){
        vorheriger=k;
        k=k.getNext();
    }
    nächster=k.getNext();
    vorheriger.setNext(nächster);
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

13


hochschule mannheim
Eine einfach verkettete Liste:
Die Liste selbst – Knoten einfügen



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

14


hochschule mannheim
Eine einfach verkettete Liste:
Die Liste selbst – Knoten einfügen



```

public void addFirst(Object o){
    Knoten k=new Knoten(o,wurzel);
    wurzel=k;
}

public void addLast(Object o) {
    Knoten k=wurzel;
    Knoten nächster;
    do{
        nächster=k.getNext();
        if (nächster!=null) k=nächster;
    }while (nächster!=null);
    nächster=new Knoten(o,null);
    k.setNext(nächster);
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 15


hochschule mannheim
Eine einfach verkettete Liste:
Die Liste selbst – Knoten einfügen



```

public void add(int pos, Object o) {
    int anz=this.getAnzKnoten();
    if ((pos<1) || (pos>anz+1))
        throw new RuntimeException("Position ungültig!");
    if (pos==1){
        this.addFirst(o); return;
    }
    if (pos==anz+1){
        this.addLast(o); return;
    }
    Knoten k=wurzel;
    for (int i=1;i<pos-1;i++){
        k=k.getNext();
    }
    // neuen Knoten zwischendrin einfügen
    Knoten neu=new Knoten(o,k.getNext());
    k.setNext(neu);
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 16

hochschule mannheim

Eine einfach verkettete Liste: Das Testprogramm



```

public class TestListe {
    public static void alleAusgeben(EinfachV рketteteListe x) {
        Knoten[] alle=x.getAll();
        if (alle!=null) {
            System.out.println("-- Liste aller Knoten --");
            for(Knoten k:alle) System.out.println(" "+k);
            System.out.println("-- Liste Ende --");
        }
        else{
            System.out.println("-- Liste: Keine Knoten vorhanden! --");
        }
    }

    public static void main(String[] args) {
        EinfachV рketteteListe l=new EinfachV рketteteListe("Frank");
        l.addLast("Dopatka"); alleAusgeben(l);
        l.addFirst("Prof. Dr."); alleAusgeben(l);
        l.add(3, "999"); alleAusgeben(l);
        l.remove(3); alleAusgeben(l);
        System.out.println(l.get(3));
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

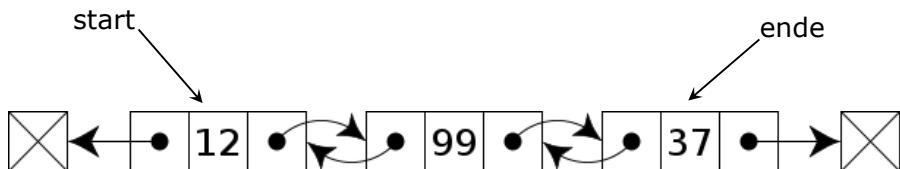
17

hochschule mannheim

Skizze einer doppelt verketteten Liste



- Im Gegensatz zur einfache-verketteten Liste hat jedes Element sowohl eine Referenz auf das nachfolgende als auch auf das vorhergehende Element.
- Die Vorgänger-Referenz des ersten und der Nachfolge-Referenz des letzten Elementes zeigen auf den Wert **null**.
- Dieses besondere Element dient zum Auffinden des Anfangs und des Endes der doppelt verketteten Liste.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

18

hochschule mannheim
einfach vs. doppelt verkettete Liste

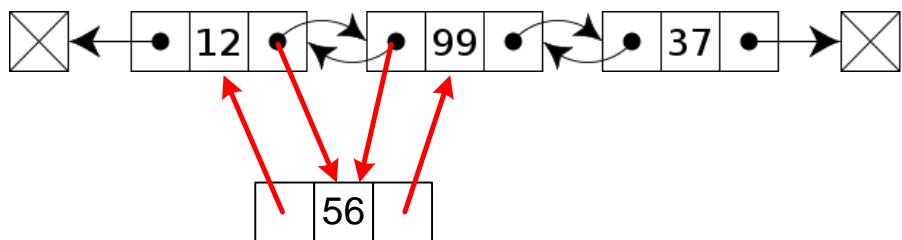


- Vorteile einer doppelt verketteten Liste:
 - Einfaches Navigieren durch die Liste: Man kann hin- und her-laufen.
 - Über die Liste kann von hinten nach vorne iteriert werden, wenn das Verwaltungsprogramm sich neben dem ersten auch das letzte Element der Liste merkt.
- Nachteile einer doppelt verketteten Liste:
 - Es gibt einen erhöhten Speicheraufwand im Vergleich zur einfach verketteten Liste mit fast 100% mehr Referenzen.
 - Das Einfügen und Löschen ist etwas schwieriger.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

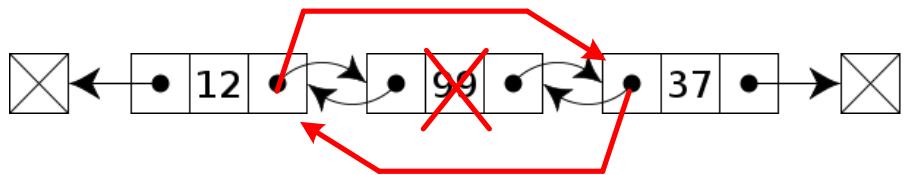
19

hochschule mannheim
**Skizze einer doppelt verketteten Liste:
Einfügen eines Elementes mittendrin**



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

20



21



FiFo & LiFo



22

Was ist ein FiFo?

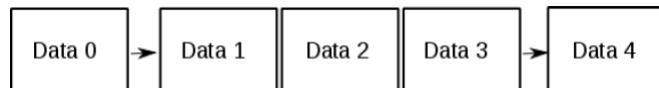
- FiFo steht für „First In, First Out“.
- Ein FiFo wird auch als Queue oder Warteschlange bezeichnet.
- Im Bereich der Warenwirtschaft ist FiFo das übliche Verfahren, da die ältesten Bestände auch nach Möglichkeit zuerst verbraucht werden sollten.
- Im Bereich von Betriebssystemen werden Datenverbindungen, die nach dem FiFo-Prinzip organisiert sind, Pipes genannt.
- Ein praktischer Bereich in der Informatik, in dem ein FiFo zum Einsatz kommt, sind Controller von seriellen Schnittstellen wie RS-232 oder USB.
 - Der gepufferte Chip sorgt durch ein FiFo-Verfahren dafür, dass das erste an der seriellen Schnittstelle ankommende Byte als erstes durch Software im Rechner verarbeitet wird.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

23

Was ist ein FiFo?

- Man stellt sich an der einen Seite an und die andere Seite wird abgearbeitet / bedient / verarbeitet:



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

24


hochschule mannheim
Referenzimplementierung eines FiFo



```

public class FiFo {
    private EinfachVerketteteListe puffer=null;

    public FiFo(){
    }

    public void put(Object o){
        if (puffer==null)
            puffer=new EinfachVerketteteListe(o);
        else
            puffer.addFirst(o);
    }

    public Object get(){
        if ((puffer==null)|| (puffer.getAnzKnoten()==0)){
            // Alternative: Exception werfen
            return null;
        }
        Object o=puffer.getLast();
        puffer.removeLast();
        return o;
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

25


hochschule mannheim
Referenzimplementierung eines FiFo



```

public class TestFiFo {
    public static void main(String[] args) {
        FiFo f=new FiFo();
        f.put("Prof. Dr.");
        f.put("Frank");
        f.put("Dopatka");
        System.out.println(f.get());
        System.out.println(f.get());
        System.out.println(f.get());
        System.out.println(f.get());
        System.out.println(f.get());
        System.out.println(f.get());
    }
}

```

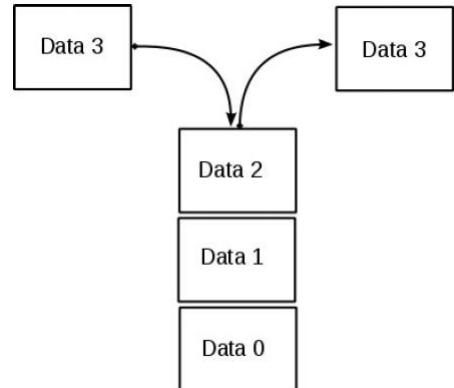
Problems	@ Javadoc	Console
<terminated> TestFiFo [Java Application]		
		Prof. Dr.
		Frank
		Dopatka
		null
		null
		null

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

26

Was ist ein LiFo?

- LiFo steht für „Last In, First Out“.
- Ein LiFo wird auch als Stack, Stapselspeicher oder Kellerspeicher bezeichnet.
- Jedes neue Speicherelement wird oben auf den Stapel gelegt.
- Wenn man ein Speicherelement auslesen möchte, so kann man nur das zuletzt aufgelegte Speicherelement wieder vom LiFo nehmen.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

Methoden eines LiFo

- Die Operation, bei der ein neues Objekt auf den Stapel gelegt wird, nennt man „push“.
- Bei der Operation „pop“ wird das oberste Objekt gelesen und gleichzeitig gelöscht.
- Sollte das oberste Objekt nur gelesen werden und nicht gelöscht, so wird eine Operation namens „peek“ benutzt.
- Die Bezeichnungen push und pop entstanden aus der Analogie zu einem Stapel von Tablets in einer Cafeteria:
 - Wird ein Tablet auf den Stapel gestellt, wird der Stapel hinuntergedrückt (engl. to push, drücken).
 - Wird ein Tablet vom Stapel entfernt, so „poppt“ der Stapel nach oben.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

28

hochschule mannheim



Die Bedeutung von Stapelspeichern

- In der Informatik bezeichnet ein Stapelspeicher eine häufig eingesetzte Datenstruktur.
- Sie wird von den meisten Mikroprozessoren in der Hardware direkt unterstützt.
- Der Stapel des Mikroprozessors wird oft von diesem selbst bei Aufruf und Rücksprung von Unterprogrammen zur Speicherung der Rücksprungadresse genutzt, ohne dass ein zusätzliches push oder pop zum Ablegen oder Holen dieser Rücksprungadresse nötig ist, da die entsprechenden Anweisungen das Register selbst richtig setzen.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

29

hochschule mannheim



Die Bedeutung von Stapelspeichern: Rekursiver Aufruf von 4!

- Rekursion wird in der Regel durch einen Stack implementiert, der die Rücksprungadressen, aber auch alle lokalen Variablen und evtl. Funktionsergebnisse aufnimmt.
- Berechnet man die Fakultät von 4, so legt jeder Aufruf folgende Informationen auf den Stapel:
 1. Platz für Ergebnis
 2. Argument x
 3. Rücksprungadresse
- Zunächst wird im Hauptprogramm **fak(4)** aufgerufen und damit die folgenden Informationen auf den Stapel gelegt:

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
Stapelzeiger →	3 Rücksprungadresse ins Hauptprogramm	

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

30

hochschule mannheim

Die Bedeutung von Stapelspeichern: Rekursiver Aufruf von 4!



- Die Fakultätsfunktion prüft jetzt, ob das Argument 0 ist.
- Da dies nicht der Fall ist, wird **4*fak(3)** berechnet.
- Zunächst muss also **fak** mit dem Argument 3 aufgerufen werden.
- Achtung: In diesem Beispiel „wächst“ der Stapel nach unten!

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
	3	Rücksprungadresse ins Hauptprogramm
	4	Platz für Ergebnis
	5	3 (Argument)
Stapelzeiger →	6	Rücksprungadresse in die Fakultätsfunktion

hochschule mannheim

Die Bedeutung von Stapelspeichern: Rekursiver Aufruf von 4!



- Das Argument ist wieder ungleich 0, also geht's weiter mit **3*fak(2)**:

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
	3	Rücksprungadresse ins Hauptprogramm
	4	Platz für Ergebnis
	5	3 (Argument)
	6	Rücksprungadresse in die Fakultätsfunktion
	7	Platz für Ergebnis
	8	2 (Argument)
Stapelzeiger →	9	Rücksprungadresse in die Fakultätsfunktion


hochschule mannheim
Die Bedeutung von Stackspeichern:
Rekursiver Aufruf von 4!



- Das Argument ist wieder ungleich 0, also $2 * \text{fak}(1)$:

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
	3	Rücksprungadresse ins Hauptprogramm
	4	Platz für Ergebnis
	5	3 (Argument)
	6	Rücksprungadresse in die Fakultätsfunktion
	7	Platz für Ergebnis
	8	2 (Argument)
	9	Rücksprungadresse in die Fakultätsfunktion
	10	Platz für Ergebnis
	11	1 (Argument)
Stapelzeiger →	12	Rücksprungadresse in die Fakultätsfunktion


hochschule mannheim
Die Bedeutung von Stackspeichern:
Rekursiver Aufruf von 4!



- Das Argument ist wieder ungleich 0, also $1 * \text{fak}(0)$:

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
	3	Rücksprungadresse ins Hauptprogramm
	4	Platz für Ergebnis
	5	3 (Argument)
	6	Rücksprungadresse in die Fakultätsfunktion
	7	Platz für Ergebnis
	8	2 (Argument)
	9	Rücksprungadresse in die Fakultätsfunktion
	10	Platz für Ergebnis
	11	1 (Argument)
	12	Rücksprungadresse in die Fakultätsfunktion
	13	Platz für Ergebnis
	14	0 (Argument)
Stapelzeiger →	15	Rücksprungadresse in die Fakultätsfunktion

hochschule mannheim

Die Bedeutung von Stapelspeichern: Rekursiver Aufruf von 4!



- Jetzt ist das Argument 0, das Ergebnis also 1.

- Wir holen die Rücksprungadresse und das Argument vom Stapel und schreiben die 1 in den dafür vorgesehenen Platz.
- Der Rücksprung führt in die Fakultätsfunktion zurück:

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
	3	Rücksprungadresse ins Hauptprogramm
	4	Platz für Ergebnis
	5	3 (Argument)
	6	Rücksprungadresse in die Fakultätsfunktion
	7	Platz für Ergebnis
	8	2 (Argument)
	9	Rücksprungadresse in die Fakultätsfunktion
	10	Platz für Ergebnis
	11	1 (Argument)
	12	Rücksprungadresse in die Fakultätsfunktion
Stapelzeiger →	13	1 (Ergebnis)

hochschule mannheim

Die Bedeutung von Stapelspeichern: Rekursiver Aufruf von 4!



- Jetzt kann man das Ergebnis mit dem Argument multiplizieren ($1 * 1$).
- Das neue Ergebnis ist wieder 1.
- Die Rücksprungadresse und das Argument werden vom Stapel geholt und das neue Ergebnis in den dafür vorgesehenen Platz geschrieben.

- Rücksprung in die Fakultätsfunktion:

Stapelanfang →	1	Platz für Ergebnis
	2	4 (Argument)
	3	Rücksprungadresse ins Hauptprogramm
	4	Platz für Ergebnis
	5	3 (Argument)
	6	Rücksprungadresse in die Fakultätsfunktion
	7	Platz für Ergebnis
	8	2 (Argument)
	9	Rücksprungadresse in die Fakultätsfunktion
Stapelzeiger →	10	1 (Ergebnis)

hochschule mannheim

Die Bedeutung von Stackspeichern: Rekursiver Aufruf von 4!



- Wiederum wird das Ergebnis mit dem Argument multipliziert ($1*2$).
- Zurück in die Fakultätsfunktion:

Stapelanfang	→	1	Platz für Ergebnis
		2	4 (Argument)
		3	Rücksprungadresse ins Hauptprogramm
		4	Platz für Ergebnis
		5	3 (Argument)
		6	Rücksprungadresse in die Fakultätsfunktion
Stapelzeiger	→	7	2 (Ergebnis)

hochschule mannheim

Die Bedeutung von Stackspeichern: Rekursiver Aufruf von 4!



- Das Ergebnis wird mit dem Argument multipliziert ($2*3$).

• Zurück in die Fakultätsfunktion: Stapelanfang → 1 Platz für Ergebnis

2 4 (Argument)

3 Rücksprungadresse ins Hauptprogramm

Stapelzeiger → 4 6 (Ergebnis)

- Das Ergebnis wird mit dem Argument multipliziert ($6*4$).

• Zurück ins Hauptprogramm:

Stapelanfang → 1 24 (Ergebnis)
Stapelzeiger

- Das Hauptprogramm muss dann nur noch das Ergebnis 24 vom Stapel holen.



hochschule mannheim
Implementierung eines LiFo

- Die geforderten Methoden lauten
 - **push** (einkellern)
legt das Objekt oben auf den Stapel.
 - **pop** (auskellern)
holt und entfernt das oberste Objekt vom Stapel.
 - **peek** (nachsehen)
holt das oberste Objekt, ohne es zu entfernen.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

39



hochschule mannheim
Implementierung eines LiFo
auf Basis einer verketteten Liste

```
public class LiFo {  
    private EinfachVerketteteListe puffer=null;  
  
    public LiFo(){  
    }  
  
    public void push(Object o){  
        if (puffer==null)  
            puffer=new EinfachVerketteteListe(o);  
        else  
            puffer.addFirst(o);  
    }  
  
    public Object pop(){  
        Object o=this.peek();  
        puffer.removeFirst();  
        return o;  
    }  
  
    public Object peek(){  
        if (puffer==null) return null;  
        return puffer.getFirst();  
    }  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

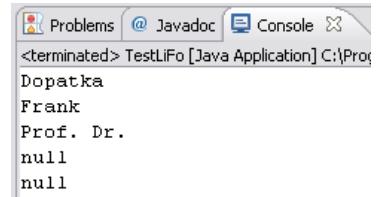
40



hochschule mannheim
Test der Implementierung



```
public class TestLiFo {  
    public static void main(String[] args) {  
        LiFo l=new LiFo();  
        l.push("Prof. Dr.");  
        l.push("Frank");  
        l.push("Dopatka");  
        System.out.println(l.pop());  
        System.out.println(l.pop());  
        System.out.println(l.pop());  
        System.out.println(l.pop());  
        System.out.println(l.pop());  
    }  
}
```



Problems @ Javadoc Console X
<terminated> TestLiFo [Java Application] C:\Proj
Dopatka
Frank
Prof. Dr.
null
null

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

41



hochschule mannheim



Hashes

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

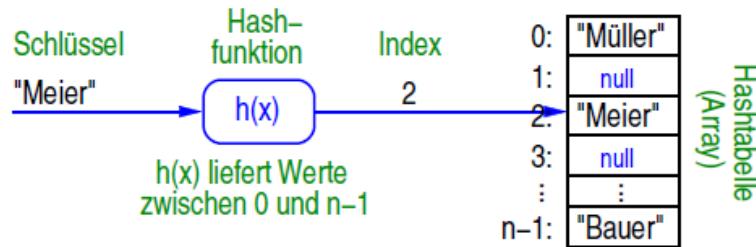
42

Was ist Hashing?

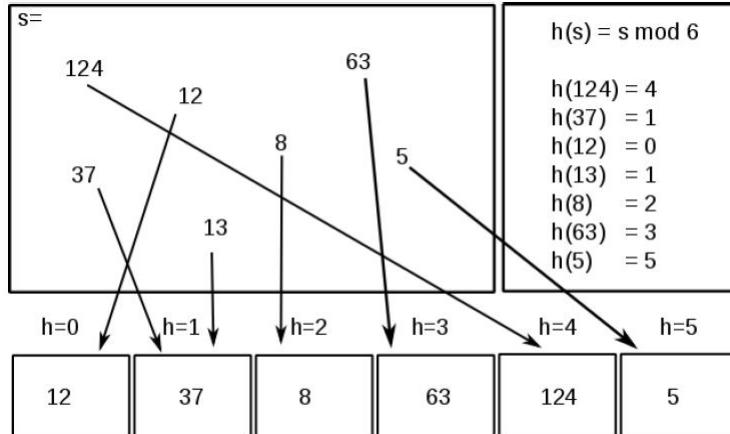
- Möchte man eine Menge von Daten in einer Hashtabelle speichern, so kann man n Speicherplätze reservieren und mittels einer Hashfunktion eine Adresse für diese Daten berechnen.
- Eine Hashfunktion liefert zu einem Datenwert immer denselben Zielwert einer bestimmten Größe.
- Aus dem Zielwert kann man aber nichtmehr auf den Datenwert zurückschließen.
- Daher werden Hashfunktion auch als Einweg-Funktionen bezeichnet.
- Die erzeugte Adresse wird dann ähnlich dem Index eines Arrays angesprochen, um zu dem gespeicherten Element zu gelangen.

Ein einfaches Hashing

- Eine einfache Hashfunktion ist die Modulo-Operation, bei der der Rest der Division zurückgegeben wird: $h(s) = s \bmod n$
- Dabei ist s der zu hashende Datenwert und n der Teiler, den man vorgeben muss, z.B. $n=6$.
- Dadurch kann ein Rest von 0 bis 5 entstehen.
- Dieser Rest kann dann die Ablage in einem konventionellen Array sein, welches 6 Elemente groß ist und über den Index 0 bis 5 angesprochen wird.




Ein einfaches Hashing




Kollisionen

- $h(13)$ und $h(37)$ erzeugen dummerweise denselben Hashwert; dies nennt man eine Kollision.
- Die Wahrscheinlichkeit einer Kollision ist umso höher,
 - je kleiner der Bereich für die Datenablage ist.
 - je höher der Füllgrad, also die Auslastung, der Datenablage ist.
- Bei einer Kollision gibt es verschiedene Möglichkeiten:
 1. Man kann das alte Element löschen und mit dem neuen überschreiben.
 2. Man kann eine „HashException“ werfen.
 3. Man kann die Datenablage so konstruieren, dass es sich um ein Array von verketteten Listen handelt, so dass das nächste Element mit diesem Hashwert hinter das vorherige abgespeichert wird.



- Als weitere Möglichkeit kann man „sondieren“, indem man nach einem festen Verfahren einen neuen Hashwert berechnen, bis man auf einen freien Speicherplatz trifft:

$$h^i(s) = (h(s) + i) \bmod n$$

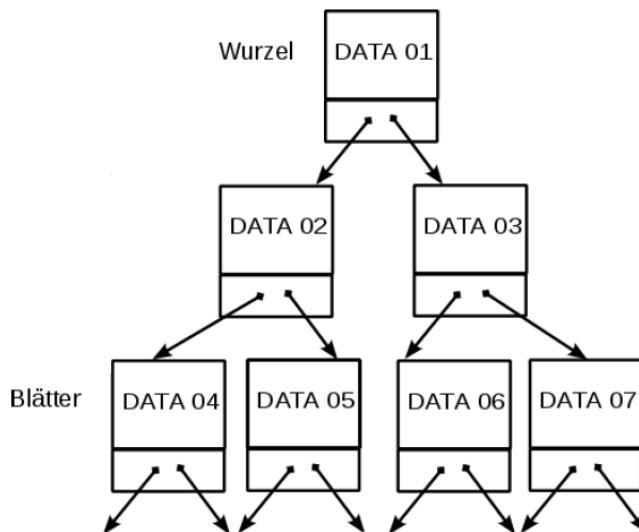


Bäume: Trees & Tries



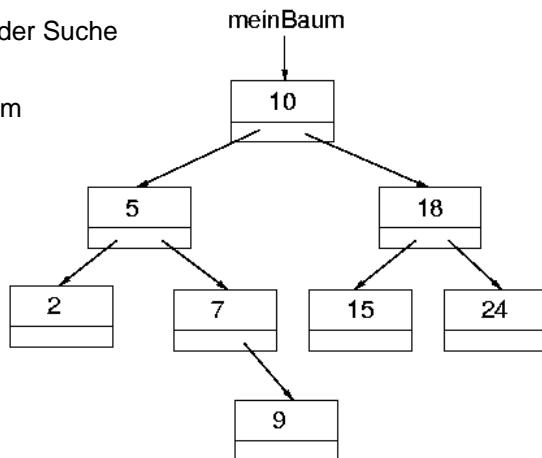


- Ein Baum hat ein Speicherelement als Wurzel, welches auf mehrere Nachfolgeelemente zeigt.
- Die wiederum zeigen auf mehrere Nachfolgeelemente.
- Bei einem Binärbaum gibt es ein linkes und ein rechtes Nachfolgeelement.
- Bei einem sortierten Binärbaum schreibt man immer
 - das kleinere Element in das linke Nachfolgeelement und
 - das größere Element in das rechte Nachfolgeelement.
- Das geht so weiter, bis die letzten Elemente, die Blätter, erreicht sind.
- Um einen Baum zu durchsuchen, kann man ihn traversieren, d.h. in einer vordefinierten Weise durchlaufen.




Ein Baum mit sortierten Zahlen


- Wie groß ist der Aufwand zu prüfen, ob eine Zahl im Baum enthalten ist oder nicht (z.B. die 6)?
- Wovon ist die Komplexität der Suche abhängig?
- Vergleichen Sie das mit dem Aufwand in einer Liste!




Begriffe rund um den Binärbaum


- Ein Binärbaum heißt geordnet, wenn jeder innere Knoten ein linkes und eventuell zusätzlich ein rechtes Kind besitzt (und nicht etwa nur ein rechtes Kind), sowie der linke Knoten kleiner, der rechte Knoten größer als der Betrachtungsknoten ist.
- Man bezeichnet ihn als voll, wenn jeder Knoten entweder Blatt ist (also kein Kind besitzt), oder aber zwei (also sowohl ein linkes wie ein rechtes) Kinder besitzt.
- Man bezeichnet volle Binäräume als vollständig, wenn alle Blätter die gleiche Höhe haben, wobei die Höhe hier als die Anzahl übergeordneter Knoten definiert sei.
- Beispiel: Ein Baum, der nur aus der Wurzel besteht, hat die Höhe 0.
- Der Binärbaum ist entartet, wenn jeder Knoten entweder Blatt ist (Anzahl Kinder ist 0) oder nur ein Kind besitzt.
- In diesem Fall stellt der Baum eine Liste dar.



hochschule mannheim
Was ist ein binärer Suchbaum?

- Ein binärer Suchbaum ist ein binärer Baum, bei dem die Knoten des linken Teilbaums eines Knotens nur kleinere Werte und die Knoten des rechten Teilbaums eines Knotens nur größere Werte als der Knoten selbst besitzen.
- Die Interpretation der Begriffe „kleiner“ und „größer“ ist dem Anwender überlassen.
- Auch ob es sich um nur einen „Schlüssel“ oder eine Kombination von Feldern handeln soll, ist Sache des Anwenders.
- Hat ein Knoten keinen Vater-Knoten, so ist er die Wurzel des Baums.
- Gilt `links==null` und `rechts==null`, so hat der Baum keine Kind-Knoten und ist damit ein Blatt.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

53



hochschule mannheim
Zeichenketten-Knoten eines Baums

- Ein binärer Suchbaum ist ein binärer Baum, bei dem die Knoten des linken Teilbaums eines Knotens nur kleinere Werte und die Knoten des rechten Teilbaums eines Knotens nur größere Werte als der Knoten selbst besitzen.
- Die Interpretation der Begriffe „kleiner“ und „größer“ ist dem Anwender überlassen.
- Auch ob es sich um nur einen „Schlüssel“ oder eine Kombination von Feldern handeln soll, ist Sache des Anwenders.
- Hat ein Knoten keinen Vater-Knoten, so ist er die Wurzel des Baums.
- Gilt `links==null` und `rechts==null`, so hat der Baum keine Kind-Knoten und ist damit ein Blatt.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

54


hochschule mannheim
Zeichenketten-Knoten eines Baums



```

public class KnotenString{
    private String s=null;
    private KnotenString links=null;
    private KnotenString rechts=null;
    private boolean markiert=false;

    public KnotenString(String s){
        this.setData(s);
    }
    public String getData() {
        return s;
    }
    public void setData(String s) {
        this.s = s;
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

55


hochschule mannheim
Zeichenketten-Knoten eines Baums



```

public KnotenString getLinks() {
    return links;
}
public void setLinks(KnotenString k) {
    this.links = k;
}
public KnotenString getRechts() {
    return rechts;
}
public void setRechts(KnotenString k) {
    this.rechts = k;
}
public void setMarkiert(boolean markiert) {
    this.markiert = markiert;
}
public boolean isMarkiert() {
    return markiert;
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

56

```
@Override
public String toString(){
    if (this.getData()==null) return null;
    return this.getData().toString();
}
```

```
public class BinärbaumNamen {
    private KnotenString wurzel=null;

    public BinärbaumNamen(String s){
        wurzel=new KnotenString(s);
    }

    public int getHöhe(){
        return getHöheRekursion(0,wurzel);
    }

    private int getHöheRekursion(int aktHöhe,KnotenString k){
        if(k!=null){
            return Math.max( // der tiefere Zweig zählt
                // links absteigen:
                getHöheRekursion(aktHöhe+1, k.getLinks()),
                // rechts absteigen
                getHöheRekursion(aktHöhe+1, k.getRechts())
            );
        }
        return aktHöhe;
    }
}
```



Ermitteln des kleinsten Elementes



```
public String getKleinster(){
    KnotenString k=getKleinster(wurzel);
    if (k==null)
        return null;
    else
        return k.getData();
}
private KnotenString getKleinster(KnotenString start){
    if(start!=null){
        while(start.getLinks() !=null) start=start.getLinks();
    }
    return start;
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

59



Ermitteln des größten Elementes

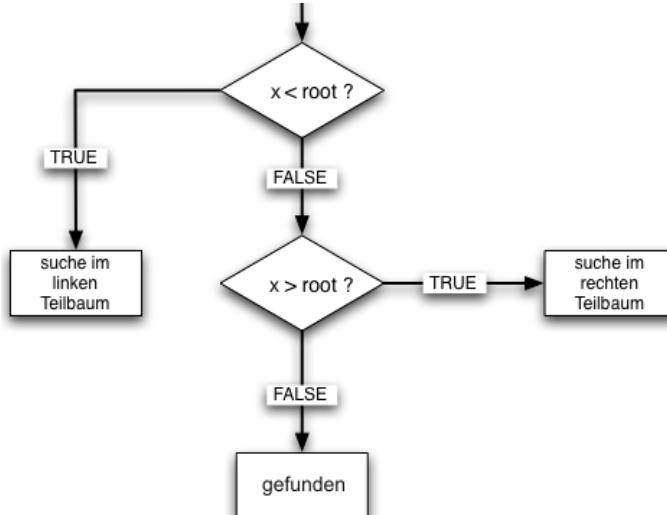


```
public String getGrößter(){
    KnotenString k=getGrößter(wurzel);
    if (k==null)
        return null;
    else
        return k.getData();
}
private KnotenString getGrößter(KnotenString start){
    if(start!=null){
        while(start.getRechts() !=null) start=start.getRechts();
    }
    return start;
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

60

hochschule mannheim
Suchen eines Knotens (PAP)



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

61

hochschule mannheim
Suchen eines Knotens

```

public KnotenString getKnoten(String s){
    return getKnotenRekursion(wurzel,s);
}

private KnotenString getKnotenRekursion(KnotenString k, String s){
    if (k==null) return null;
    if ((s.compareTo(k.getData())==0) return k; // gefunden
    if ((s.compareTo(k.getData())<0) { // links weiter suchen
        if (k.getLinks()==null) return null;
        return getKnotenRekursion(k.getLinks(),s);
    }
    if ((s.compareTo(k.getData())>0){ // rechts weiter suchen
        if (k.getRechts()==null) return null;
        return getKnotenRekursion(k.getRechts(),s);
    }
    return null;
}
  
```

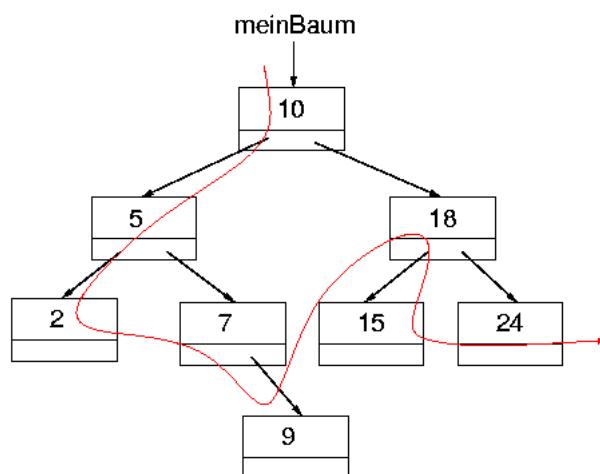
Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

62

Was ist Traversierung?

- Traversierung bezeichnet das Untersuchen der Knoten des Baumes in einer bestimmten Reihenfolge.
- Es gibt verschiedene Möglichkeiten, die Knoten von Binäräbäumen zu durchlaufen. Man unterscheidet:
 - pre-order oder Hauptreihenfolge (W–L–R):
wobei zuerst die Wurzel (W) betrachtet wird und anschließend zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen wird,
 - in-order oder symmetrische Reihenfolge (L–W–R):
wobei zuerst der linke (L) Teilbaum durchlaufen wird, dann die Wurzel (W) betrachtet wird und anschließend der rechte (R) Teilbaum durchlaufen wird sowie
 - post-order oder Nebenreihenfolge (L–R–W):
wobei zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen wird und anschließend die Wurzel (W) betrachtet wird.

pre-order Durchlauf (W-L-R)

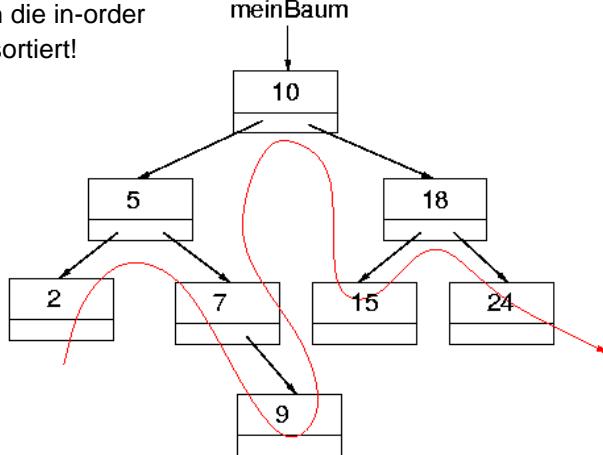


hochschule mannheim
pre-order Durchlauf (W-L-R)
rekursiver Pseudocode

```
preOrder( knoten ):  
  
    print( knoten )  
  
    if knoten.links ≠ null then  
        preOrder( knoten.links )  
  
    if knoten.rechts ≠ null then  
        preOrder( knoten.rechts )
```

hochschule mannheim
in-order Durchlauf (L-W-R)

- Ist der Baum sortiert,
so ist auch die in-order
Ausgabe sortiert!



 hochschule mannheim
in-order Durchlauf (L-W-R)
rekursiver Pseudocode

```
inOrder( knoten ):  
  
    if knoten.links ≠ null then  
        inOrder( knoten.links )  
  
    print( knoten )  
  
    if knoten.rechts ≠ null then  
        inOrder( knoten.rechts )
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

67

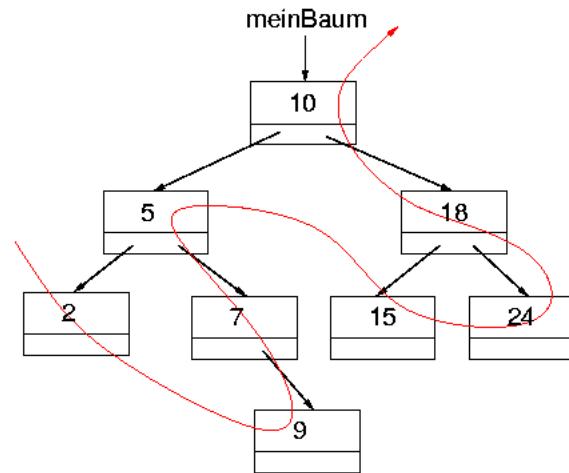
 hochschule mannheim
in-order Durchlauf (L-W-R)
Java-Implementierung

```
public String inOrderAusgabe(){  
    return inOrderRekursion(wurzel);  
}  
private String inOrderRekursion(KnotenString k){  
    String s="";  
    if (k==null) return "<leer>";  
    // L  
    if(k.getLinks() !=null)  
        s+=inOrderRekursion(k.getLinks());  
    // W  
    s+=k.getData()+" ";  
    // R  
    if(k.getRechts() !=null)  
        s+=inOrderRekursion(k.getRechts());  
    return s;  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

68

hochschule mannheim
post-order Durchlauf (L-R-W)



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 69



hochschule mannheim
post-order Durchlauf (L-R-W)
rekursiver Pseudocode

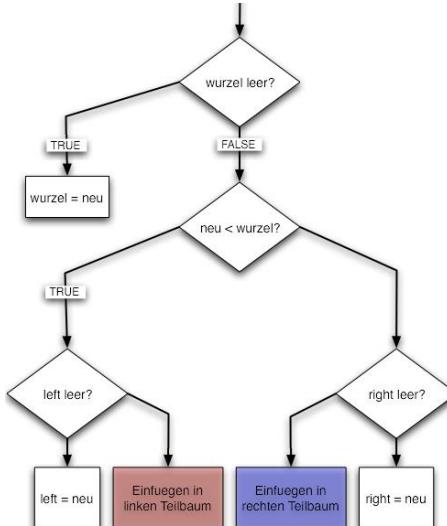


```
postOrder( knoten ):  
    if knoten.links ≠ null then  
        postOrder( knoten.links )  
  
    if knoten.rechts ≠ null then  
        postOrder( knoten.rechts )  
  
    print( knoten )
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 70



hochschule mannheim
Programmablaufplan
zum Einfügen neuer Elemente



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

71

hochschule mannheim
Java-Code
zum Einfügen neuer Elemente

```

public void einfügen(String s){
    // Einträge werden nicht doppelt gespeichert
    if (wurzel==null) wurzel=new KnotenString(s);
    this.einfügenRekursion(wurzel,s);
}
private void einfügenRekursion(KnotenString k,String s){
    if (k==null)
        k=new KnotenString(s); if (wurzel==null) wurzel=k;
    else{
        if ((s.compareTo(k.getData())<0){
            if (k.getLinks()!=null)
                einfügenRekursion(k.getLinks(),s);
            else
                k.setLinks(new KnotenString(s));
        }
        if ((s.compareTo(k.getData())>0){
            if (k.getRechts()!=null)
                einfügenRekursion(k.getRechts(),s);
            else
                k.setRechts(new KnotenString(s));
        }
    }
}
  
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

72

hochschule mannheim
Löschen eines Blattes

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

73

hochschule mannheim
Löschen eines Blattes

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

74



Löschen eines Blattes



- Der Knoten mit der 60 existiert zwar weiterhin im Arbeitsspeicher, ist aber nicht mehr in den Binärbaum integriert und kann daher nicht "wiedergefunden" werden.
- Bei der Ausgabe, beim rekursiven Einfügen, bei der Suche und bei anderen Operationen wird die 60 nicht mehr berücksichtigt.
- Es ist so, als ob der Knoten gar nicht mehr da wäre.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

75



Löschen eines Blattes:
Java-Code

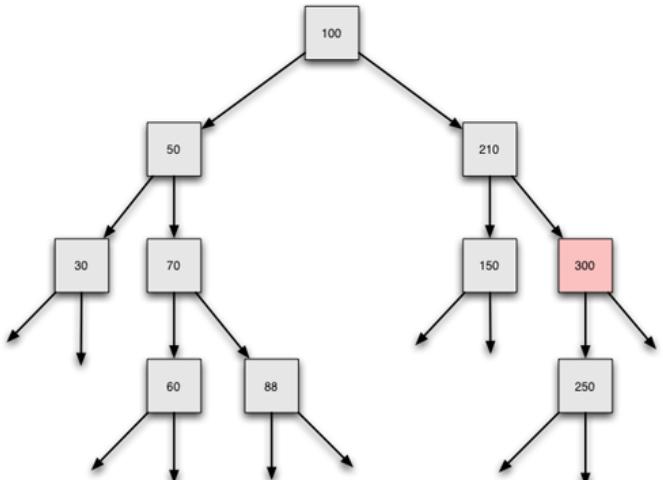


```
public boolean löschen(String s){  
    return löschenRekursion(wurzel,s,null,false);  
}  
private boolean löschenRekursion(  
    KnotenString k,String s,KnotenString vater,boolean binLinksVomVater){  
    if (k==null) return false;  
    if ((s.compareTo(k.getData()))==0){  
        // gefunden: Knoten k ist zu löschen!  
        // k ist Blatt  
        if ((k.getLinks()==null) && (k.getRechts()==null)) {  
            if (vater==null)  
                wurzel=null;  
            else  
                hängeUnterVater(vater,binLinksVomVater,null);  
            return true;  
        }  
    }  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

76

hochschule mannheim
**Löschen eines Knotens
mit genau einem Nachfolger**

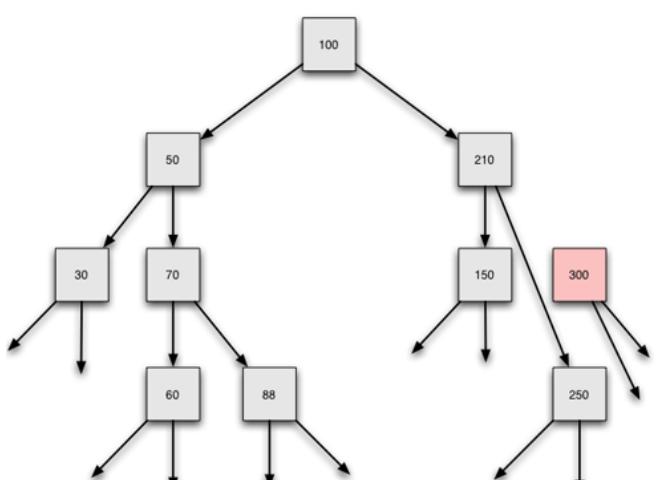


A binary search tree diagram illustrating the deletion of a node with exactly one successor. The root node is 100. It has two children: 50 (left) and 210 (right). Node 50 has two children: 30 (left) and 70 (right). Node 70 has two children: 60 (left) and 88 (right). Node 210 has two children: 150 (left) and 300 (right). Node 300 has one child: 250 (left). All nodes except 300 are gray, while 300 is pink.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

77

hochschule mannheim
**Löschen eines Knotens
mit genau einem Nachfolger**



The same binary search tree diagram as the previous slide, showing the state before or after a deletion operation. The root node is 100. It has two children: 50 (left) and 210 (right). Node 50 has two children: 30 (left) and 70 (right). Node 70 has two children: 60 (left) and 88 (right). Node 210 has two children: 150 (left) and 300 (right). Node 300 has one child: 250 (left). All nodes except 300 are gray; node 300 is pink.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

78

hochschule mannheim

Löschen eines Knotens mit genau einem Nachfolger

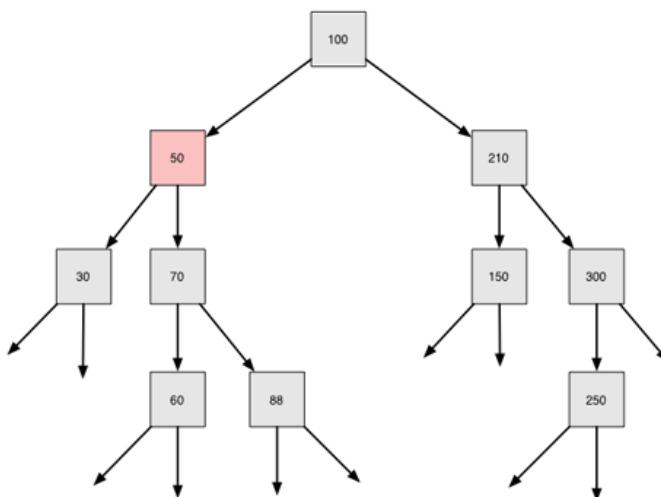


```
// k hat nur rechts einen Unter-Knoten -> umbiegen
if ((k.getLinks() ==null) &&(k.getRechts() !=null)){
    if (vater==null)
        wurzel=k.getRechts();
    else
        hängeUnterVater(vater,binLinksVomVater,k.getRechts());
    return true;
}
// k hat nur links einen Unter-Knoten -> umbiegen
if ((k.getLinks() !=null) &&(k.getRechts() ==null)){
    if (vater==null)
        wurzel=k.getLinks();
    else
        hängeUnterVater(vater,binLinksVomVater,k.getLinks());
    return true;
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 79

hochschule mannheim

Löschen eines Knotens mit mit 2 Nachfolgern



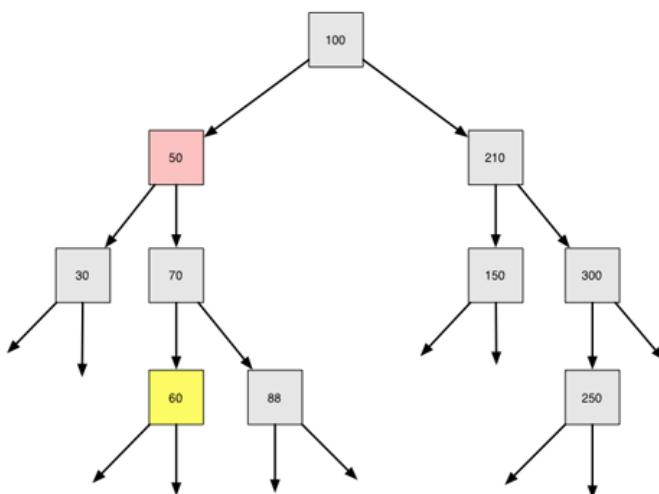
Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 80

hochschule mannheim
**Löschen eines Knotens
mit mit 2 Nachfolgern**

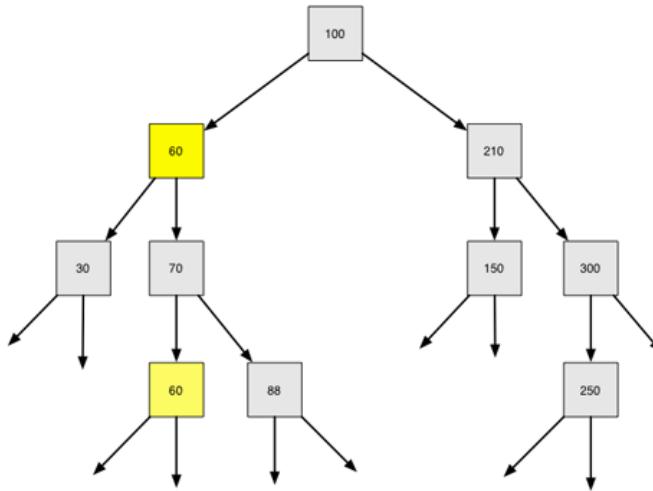


- Ersetze den Inhalt des zu löschenen Knotens durch den größten Wert des linken Teilbaums oder alternativ durch den kleinsten Wert des rechten Teilbaumes.
- Lösche anschließend den entsprechenden Knoten mit einer der beiden zuvor beschriebenen Methoden.

hochschule mannheim
**Löschen eines Knotens
mit mit 2 Nachfolgern**



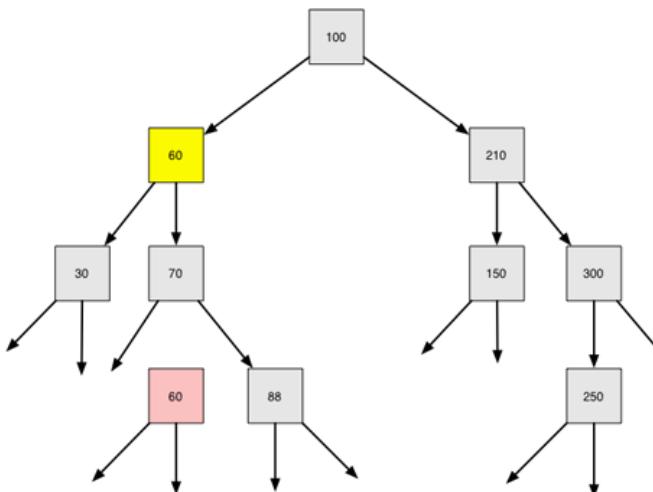
hochschule mannheim
**Löschen eines Knotens
mit mit 2 Nachfolgern**



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

83

hochschule mannheim
**Löschen eines Knotens
mit mit 2 Nachfolgern**



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

84

hochschule mannheim

Löschen eines Knotens mit mit 2 Nachfolgern



```
// k hat sowohl links, als auch rechts noch einen Unter-Knoten
// Ersetze den Inhalt des zu löschenen Knotens durch den kleinsten Wert
// des rechten Teilbaumes.
// Lösche den entsprechenden unteren Knoten
KnotenString x=getKleinster(k.getRechts());
System.out.println("X:"+x);
löschen(x.getData());
if (vater==null){
    x.setLinks(wurzel.getLinks());
    x.setRechts(wurzel.getRechts());
    wurzel=x;
}
else{
    x.setLinks(k.getLinks());
    x.setRechts(k.getRechts());
    hängeUnterVater(vater,binLinksVomVater,x);
}
}
```

hochschule mannheim

Noch nichts gefunden? Weitersuchen!



```
// nicht gefunden: links weiter suchen
if ((s.compareTo(k.getData())<0)){
    if (k.getLinks()==null) return false;
    return löschenRekursion(k.getLinks(),s,k,true);
}
// nicht gefunden: rechts weiter suchen
if ((s.compareTo(k.getData())>0)){
    if (k.getRechts()==null) return false;
    return löschenRekursion(k.getRechts(),s,k,false);
}
return false;
}
```


hochschule mannheim
Neue Zuweisungen von Knoten:
Referenzen „umbiegen“



```

private void hängeUnterVater(KnotenString vater,boolean links,KnotenString k) {
    if (links) // das Kind hängt links vom Vater...
        vater.setLinks(k);
    else // ...oder rechts
        vater.setRechts(k);
}
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 87


hochschule mannheim
1. Test der Baum-Implementierung



```

public class TestBinärbaumNamen {
    public static void main(String[] args) {
        BinärbaumNamen bn=new BinärbaumNamen("Dopatka");
        bn.einfügen("Meier");
        bn.einfügen("Schmidt");
        bn.einfügen("Arenson");
        bn.einfügen("Müller");
        bn.einfügen("Wusserow");
        bn.einfügen("Schulz");
        System.out.println("Höhe:"+bn.getHöhe());

        bn.löschen("Wusserow");
        System.out.println(bn.inOrderAusgabe());
        System.out.println(bn.getKleinster());
        System.out.println(bn.getKnoten("Meier"));
        System.out.println("L:"+bn.getKnoten("Meier").getLinks());
        System.out.println("R:"+bn.getKnoten("Meier").getRechts());
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 88

hochschule mannheim

1. Test der Baum-Implementierung



```
Problems @ Javadoc Console X
<terminated> TestBinärbaumNamen [Java Application] C:\Programme\jre\bin\javaw.exe
Höhe:5
Arenson Dopatka Meier Müller Schmidt Schulz
Arenson
Meier
L:null
R:Schmidt
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 89

hochschule mannheim

2. Test der Baum-Implementierung: Ein günstiger Baum



```
public class TestGünstig {
    public static void main(String[] args) {
        BinärbaumNamen bn=new BinärbaumNamen("Müller");
        bn.einfügen("Dopatka");
        bn.einfügen("Schulz");
        bn.einfügen("Arenson");
        bn.einfügen("Meier");
        bn.einfügen("Wusserow");
        bn.einfügen("Schmidt");
        System.out.println("Höhe:"+bn.getHöhe());
        System.out.println(bn.inOrderAusgabe());
    }
}
```

```
Problems @ Javadoc Console X
<terminated> TestGünstig [Java Application] C:\Programme\jre\bin\javaw.exe (05.07.2015)
Höhe:3
Arenson Dopatka Meier Müller Schmidt Schulz Wusserow
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 90

hochschule mannheim

3. Test der Baum-Implementierung: Ein günstiger Baum



```

public class TestUngünstig {
    public static void main(String[] args) {
        BinärbaumNamen bn=new BinärbaumNamen("Wusserow");
        bn.einfügen("Schulz");
        bn.einfügen("Schmidt");
        bn.einfügen("Müller");
        bn.einfügen("Meier");
        bn.einfügen("Dopatka");
        bn.einfügen("Arenson");
        System.out.println("Höhe:"+bn.getHöhe());
        System.out.println(bn.inOrderAusgabe());
    }
}

```

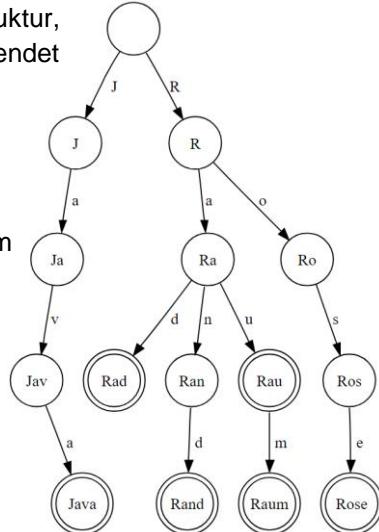


Höhe:7
Arenson Dopatka Meier Müller Schmidt Schulz Wusserow

- hochschule mannheim
- ### Probleme von Binärbäumen
- 
- Bei einer ungünstigen Einfügung von Elementen „entartet“ der Binärbaum zu einer verketteten Liste.
 - Ungünstig bedeutet hier dummerweise „sortiert“.
 - Die Zugriffe sind dann aufgrund des Baum-Overheads und der Rekursionen noch etwas langsamer als bei einer Liste.
 - Man kann das Problem beheben, indem man den Baum „ausgleicht“.
 - Sogenannte AVL-Knoten speichern nicht nur den Wert und Referenzen auf den linken bzw. rechten Teilbaum, sondern zusätzlich noch die Differenz der Höhen der Teilbäume!
 - Sind diese Differenzen zu groß, können Algorithmen zum „Ausgleichen“ bzw. „Ausbalancieren“ von Bäumen angewendet werden,
s. <http://magazin.c-plusplus.de/artikel/AVL-Baum>

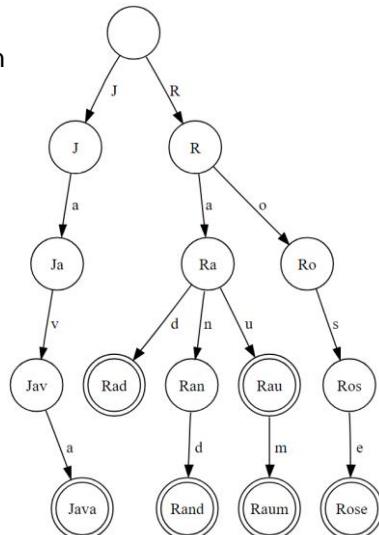

Und was sind jetzt Tries?

- Ein Trie oder Präfixbaum ist eine Datenstruktur, die zum Suchen nach Zeichenketten verwendet wird.
- Es handelt sich dabei um einen speziellen Suchbaum zur gleichzeitigen Speicherung mehrerer Zeichenketten.
- Statt 2 Kindknoten beim binären Suchbaum sind hier n Kindknoten möglich.
- Ein Trie realisiert Datenkompression, da gemeinsame Präfixe der Zeichenketten nur einmal gespeichert werden.
- Ein Trie wird stets über eine Menge von beliebigen Zeichenketten aufgebaut.




Und was sind jetzt Tries?

- Jede ausgehende Kante eines Knotens innerhalb eines tries ist mit einem Zeichen versehen, sodass ein Pfad beginnend bei der Wurzel bis zu einem Blatt im tries eine der Zeichenketten darstellt, aus der der Baum konstruiert worden ist.
- tries eignen sich zur Indexierung von Texten, um effizient bestimmte Anfragen an den Text zu beantworten.




Und was sind jetzt Tries?


- Eine mögliche Verwendung kann die Realisierung von Suchanfragen sein.
- Mit Hilfe des Tries kann eine Personensuche nach Namen erfolgen über Existenzanfragen.
- Auf dieser Basis ist auch ein Auto-Vervollständigen von Wörtern realisierbar.



95

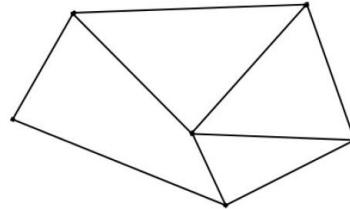




Graphen

Was ist ein Graph in der Graphentheorie?

- Ein Graph besteht in der Graphentheorie anschaulich aus einer Menge von Punkten, zwischen denen Linien verlaufen.
- Die Punkte nennt man Knoten oder Ecken, die Linien nennt man Kanten.
- Besitzen Kanten im Graphen Richtungspfeile, so spricht man von einem gerichteten Graphen, ansonsten von einem ungerichteten Graphen.
- Die Anzahl der Kanten an einem Knoten nennt man Grad des Knotens.
- Der maximale Grad von Knoten in einem Graph nennt man Grad des Graphen.



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

97

Beispiel eines Graphen

- In einem U-Bahn-Netz stellt jeder Knoten eine U-Bahn-Station dar und jede Kante eine direkte Zugverbindung zwischen zwei Stationen:





hochschule mannheim
Implementierung einer Kante

```
public class Kante {  
    private Knoten knoten1=null;  
    private Knoten knoten2=null;  
  
    public Kante(Knoten kn1,Knoten kn2){  
        knoten1=kn1; knoten2=kn2;  
        kn1.addKante(this); kn2.addKante(this);  
    }  
  
    public Knoten getAnderesEnde(Knoten kn){  
        if (knoten1==kn)  
            return knoten2;  
        else  
            return knoten1;  
    }  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 99



hochschule mannheim
Implementierung eines Knotens

```
import java.util.ArrayList;  
public class Knoten {  
    private int id=0;  
    private ArrayList<Kante> kantenListe=new ArrayList<Kante>();  
  
    public Knoten(int id){  
        this.id=id;  
    }  
  
    public int getId(){  
        return this.id;  
    }  
  
    public int getGrad(){  
        return kantenListe.size();  
    }  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 100



hochschule mannheim
Implementierung eines Knotens

```
public void addKante(Kante ka){  
    if (ka==null)  
        throw new RuntimeException("addKante: Ungültige Kante!");  
    kantenListe.add(ka);  
}  
  
public ArrayList<Kante> getKanten(){  
    return kantenListe;  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

101



hochschule mannheim
Implementierung eines Knotens

```
public ArrayList<Knoten> getNachbarKnoten(){  
    ArrayList<Kante> kanten=getKanten();  
    if (kanten==null) return null;  
    ArrayList<Knoten> aus=new ArrayList<Knoten>();  
    for(Kante ka:kanten){  
        aus.add(ka.getAnderesEnde(this));  
    }  
    return aus;  
}  
  
@Override  
public String toString(){  
    return ""+this.getId();  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

102



hochschule mannheim
Implementierung des Graphen

```

import java.util.ArrayList;
public class Graph {
    private ArrayList<Knoten> knotenListe=new ArrayList<Knoten>();

    public Graph(){
    }

    public void addKnoten(Knoten kn) {
        if (kn==null)
            throw new RuntimeException("addKnoten: Ungültiger Knoten!");
        knotenListe.add(kn);
    }
    public boolean hatKnoten(Knoten kn) {
        if (kn==null) return false;
        for(Knoten knAusListe:knotenListe){
            if (knAusListe==kn) return true;
        }
        return false;
    }
    public int getAnzKnoten(){
        return knotenListe.size();
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

103



hochschule mannheim
Implementierung des Graphen

```

public Knoten getKnoten(int id){
    for(Knoten knAusListe:knotenListe){
        if (knAusListe.getId()==id) return knAusListe;
    }
    return null;
}
public ArrayList<Knoten> getKnoten() {
    return knotenListe;
}
public int getGrad(){
    int max=0;
    for(Knoten knAusListe:knotenListe){
        if (knAusListe.getGrad(>max) max=knAusListe.getGrad());
    }
    return max;
}
public void setKnoten(ArrayList<Knoten> liste){
    if ((liste==null)|| (liste.size(<1))
        throw new RuntimeException("setKnoten: Ungültige Liste!");
    knotenListe=liste;
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

104

hochschule mannheim

Implementierung des Graphen: Test



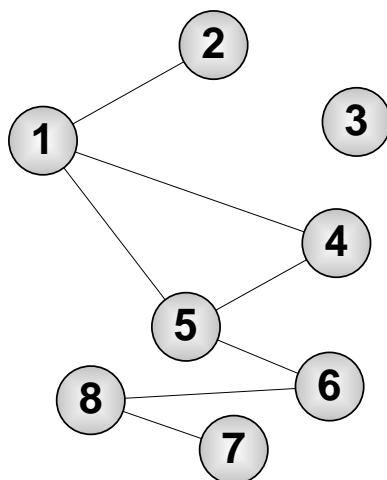
```
public class TestGraph {  
    public static void main(String[] args) {  
        // Graph erzeugen und Konten hinzufügen  
        Graph g=new Graph();  
        for (int i=1;i<=8;i++) g.addKnoten(new Knoten(i));  
        // Kanten hinzufügen  
        new Kante(g.getKnoten(1),g.getKnoten(2));  
        new Kante(g.getKnoten(1),g.getKnoten(4));  
        new Kante(g.getKnoten(1),g.getKnoten(5));  
        new Kante(g.getKnoten(4),g.getKnoten(5));  
        new Kante(g.getKnoten(5),g.getKnoten(6));  
        new Kante(g.getKnoten(6),g.getKnoten(8));  
        new Kante(g.getKnoten(8),g.getKnoten(7));  
  
        System.out.println("Grad Graph: "+g.getGrad());  
        Knoten k=g.getKnoten(2);  
        System.out.println("Grad Knoten2: "+k.getGrad());  
    }  
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

105

hochschule mannheim

Implementierung des Graphen: Test



```
Console X  
<terminated> TestGraph (1) [Java Application]  
Grad Graph: 3  
Grad Knoten2: 1
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

106



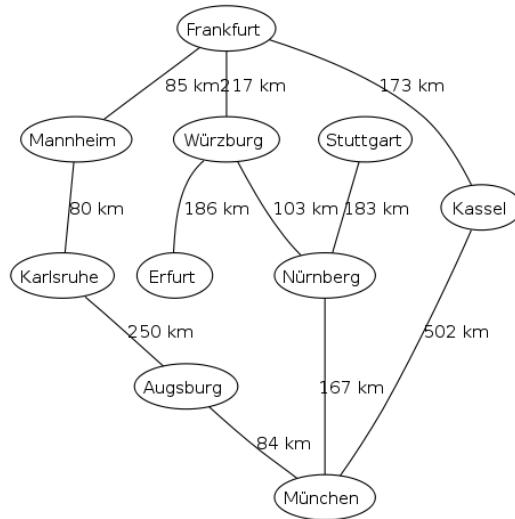
hochschule mannheim
Der kürzeste Weg mit Dijkstra

- Der Algorithmus von Dijkstra (nach seinem Erfinder Edsger W. Dijkstra) ist ein Algorithmus aus der Klasse der Greedy-Algorithmen und dient der Berechnung eines kürzesten Pfades zwischen einem Startknoten und einem oder mehreren beliebigen Knoten in einem kantengewichteten Graphen.
- Die Kantengewichte des Graphen dürfen dabei nicht negativ sein.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 107



hochschule mannheim
Der kürzeste Weg mit Dijkstra



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 108



hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Die Idee – sprachlich formuliert

- Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus verspricht.
- Andere Kanten werden erst dann verfolgt, wenn alle kürzeren Streckenabschnitte beachtet wurden. Dieses Vorgehen gewährleistet, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existieren kann.
- Dieses Vorgehen wird fortgesetzt, bis die Distanz des Zielknotens berechnet wurde.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

109



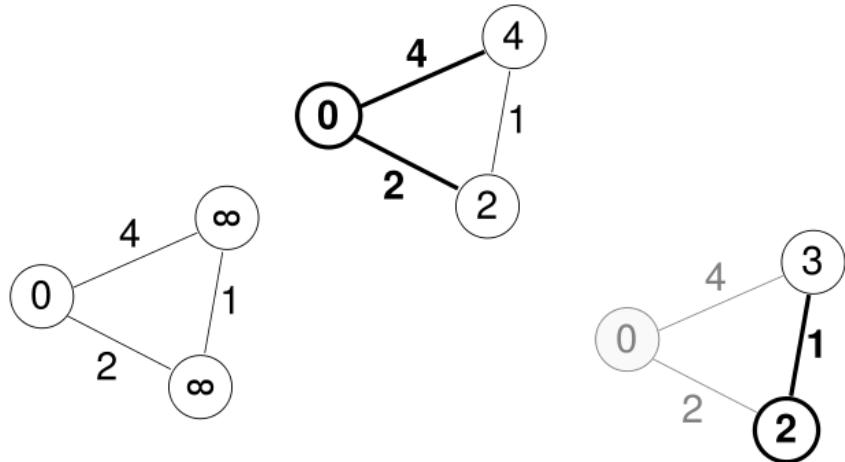
hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Die Idee – sprachlich formuliert

1. Weise allen Knoten die beiden Eigenschaften "Distanz" und "Vorgänger" zu.
2. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞ .
3. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
 1. Speichere, dass dieser Knoten schon besucht wurde.
 2. Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten.
 3. Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.
4. Ist der gesuchte Knoten der Zielknoten, so breche ab.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

110


**Der kürzeste Weg mit Dijkstra:
Die Idee am Beispiel**




**Der kürzeste Weg mit Dijkstra:
Die Idee als Pseudocode**

```

Funktion init(Graph,Startknoten,abstand[],vorgänger[],Q)
  für jeden Knoten v in Graph:
    abstand[v] := unendlich
    vorgänger[v] := null
    abstand[Startknoten] := 0
  ende // für
  Q := Die Menge aller Knoten in Graph
ende // funktion
    
```


hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Die Idee als Pseudocode



```

Funktion Dijkstra(Graph, Startknoten) :
  init (Graph,Startknoten,abstand[],vorgänger[],Q)
  solange Q nicht leer: // Der eigentliche Algorithmus
    u := Knoten in Q mit kleinstem Wert in abstand[]
    entferne u aus Q // u kennt jetzt kürzesten Weg
    wenn u=Zielknoten dann return vorgänger[]
    für jeden Nachbarn v von u:
      falls v in Q:
        distanzupdate(u,v,abstand[],vorgänger[])
      ende // falls
    ende // für
  ende // solange
ende // funktion

```


hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Die Idee als Pseudocode



```

Funktion distanzupdate(u,v,abstand[],vorgänger[])
  alternativ := abstand[u] + abstand_zwischen(u, v)
  falls alternativ < abstand[v]
    abstand[v] := alternativ
    vorgänger[v] := u
  ende // falls
ende // funktion

```


 hochschule mannheim
Der kürzeste Weg mit Dijkstra:
 Die Idee als Pseudocode



```

Funktion erstellePfad(Zielknoten,vorgänger[])
  Weg[] := [Zielknoten]
  u := Zielknoten
  // Vorgänger des Startknotens ist null
  solange vorgänger[u] nicht null
    u := vorgänger[u]
    füge u am Anfang von Weg[] ein
  ende // solange
  return Weg[]
ende // funktion

```


 hochschule mannheim
Der kürzeste Weg mit Dijkstra:
 Java-Code der Kante



```

import graph.*;
public class KanteDijkstra extends Kante {
  private int distanz=Integer.MAX_VALUE;

  public KanteDijkstra(Knoten kn1, Knoten kn2) {
    super(kn1, kn2);
  }
  public KanteDijkstra(Knoten kn1, Knoten kn2, int distanz) {
    this(kn1,kn2);
    setDistanz(distanz);
  }

  public void setDistanz(int distanz) {
    if (distanz<0)
      throw new RuntimeException("Distanz darf nicht negativ sein!");
    this.distanz = distanz;
  }
  public int getDistanz() {
    return distanz;
  }
}

```


hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Java-Code des Knotens



```

import graph.*;
public class KnotenDijkstra extends Knoten {
    private int distanz=Integer.MAX_VALUE;
    private KnotenDijkstra vorgänger=null;

    public KnotenDijkstra(int id) {
        super(id);
    }
    public void setDistanz(int distanz) {
        if (distanz<0)
            throw new RuntimeException("Distanz ungültig");
        this.distanz = distanz;
    }
    public int getDistanz() {
        return distanz;
    }

    public void setVorgänger(KnotenDijkstra vorgänger) {
        this.vorgänger = vorgänger;
    }
    public KnotenDijkstra getVorgänger() {
        return vorgänger;
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

117


hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Java-Code des Graphen



```

import graph.*;
import java.util.ArrayList;
public class GraphDijkstra extends Graph{
    public GraphDijkstra() {
        super();
    }
    public void init(){
        for(Knoten knAusListe:getKnoten()){
            KnotenDijkstra kd=(KnotenDijkstra)knAusListe;
            kd.setVorgänger(null);
            kd.setDistanz(Integer.MAX_VALUE);
        }
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

118


hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Java-Code des Graphen



```

public ArrayList<KnotenDijkstra> kürzesterWeg(int start,int ende){
    init();
    KnotenDijkstra s=(KnotenDijkstra) getKnoten(start);
    KnotenDijkstra e=(KnotenDijkstra) getKnoten(ende);
    if ((s==null) || (e==null)) return null;
    if (s==e) return null;
    s.setDistanz(0);
    ArrayList<KnotenDijkstra> Q=new ArrayList<KnotenDijkstra>();
    for(Knoten kn:getKnoten()){
        Q.add((KnotenDijkstra) kn);
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 119


hochschule mannheim
Der kürzeste Weg mit Dijkstra:
Java-Code des Graphen



```

do{
    KnotenDijkstra gewählt=null;
    for(KnotenDijkstra x:Q){
        if (gewählt==null) gewählt=x;
        if (gewählt.getDistanz()>x.getDistanz()) gewählt=x;
    }
    Q.remove(gewählt);
    for(Kante ka:gewählt.getKanten()){
        KanteDijkstra kaD=(KanteDijkstra) ka;
        KnotenDijkstra knD=(KnotenDijkstra) ka.getAnderesEnde(gewählt);
        if (Q.contains(knD)){
            if (gewählt.getDistanz()+kaD.getDistanz()<knD.getDistanz()){
                knD.setDistanz(gewählt.getDistanz()+kaD.getDistanz());
                knD.setVorgänger(gewählt);
            }
        }
    }
}while(Q.size()>0);

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 120


Der kürzeste Weg mit Dijkstra:
Java-Code des Graphen



```

ArrayList<KnotenDijkstra> aus=new ArrayList<KnotenDijkstra>();
KnotenDijkstra x=e;
do{
    aus.add(0, x);
    x=x.getVorgänger();
}while (x!=null);
return aus;
}
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 121


Der kürzeste Weg mit Dijkstra:
Testen



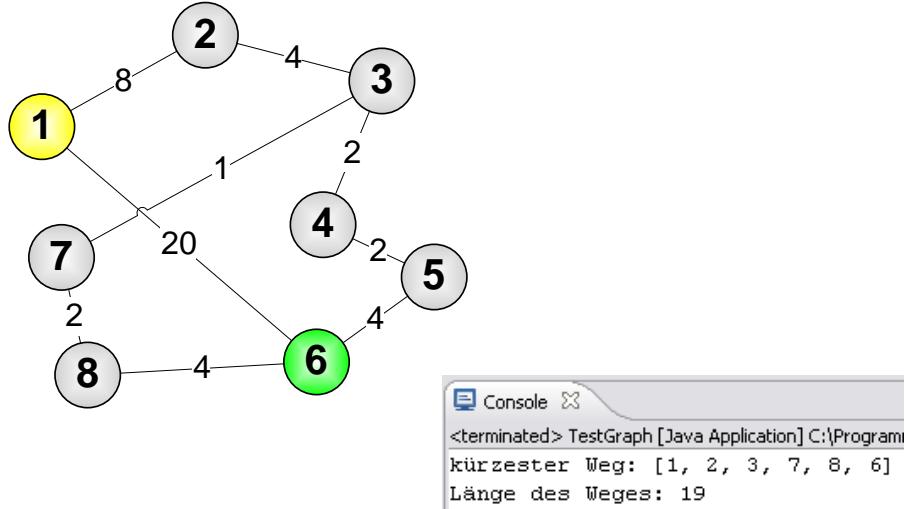
```

import java.util.ArrayList;
public class TestGraph {
    public static void main(String[] args) {
        // Graph erzeugen und Konten hinzufügen
        GraphDijkstra g=new GraphDijkstra();
        for (int i=1;i<=8;i++) g.addKnoten(new KnotenDijkstra(i));
        // Kanten hinzufügen
        new KanteDijkstra(g.getKnoten(1),g.getKnoten(2),8);
        new KanteDijkstra(g.getKnoten(2),g.getKnoten(3),4);
        new KanteDijkstra(g.getKnoten(3),g.getKnoten(4),2);
        new KanteDijkstra(g.getKnoten(4),g.getKnoten(5),2);
        new KanteDijkstra(g.getKnoten(5),g.getKnoten(6),4);
        new KanteDijkstra(g.getKnoten(6),g.getKnoten(8),4);
        new KanteDijkstra(g.getKnoten(8),g.getKnoten(7),2);
        new KanteDijkstra(g.getKnoten(7),g.getKnoten(3),1);
        new KanteDijkstra(g.getKnoten(1),g.getKnoten(6),20);
        ArrayList<KnotenDijkstra> weg=g.kürzesterWeg(1, 6);
        System.out.println("kürzester Weg: "+weg);
        System.out.println("Länge des Weges: "+
            ((KnotenDijkstra)g.getKnoten(6)).getDistanz());
    }
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka 122


Der kürzeste Weg mit Dijkstra:
Testen




Was ist Graphenfärbung?

- Eine Knotenfärbung eines ungerichteten Graphen ordnet jedem Knoten im Graphen eine Farbe zu.
- Eine Farbe ist eine ganze Zahl größer als 0.
- In der Graphentheorie beschäftigt man sich meist nur mit „gültigen“ Färbungen und versucht, Algorithmen zu entwickeln, die für einen vorgegebenen Graphen eine gültige Färbung mit möglichst wenigen Farben finden.
- Probleme aus der diskreten Mathematik, aber auch außer-mathematische Fragestellungen lassen sich oft in ein Färbungsproblem übersetzen, daher sind solche Algorithmen auch außerhalb der Graphentheorie von Interesse.



Was ist eine gültige Färbung?



- Eine Knotenfärbung ist gültig, falls je zwei beliebige benachbarte Knoten eine andere Farbe besitzen.
- Ein Graph ist k -knotenfärbbar, falls es eine gültige Knotenfärbung des Graphen gibt, bei der nicht mehr als k Farben verwendet wurden.
- Wenn ein Graph färbbar ist, gibt es eine kleinste Zahl x , sodass der Graph x -knotenfärbbar ist.
 - Diese Zahl wird die chromatische Zahl oder Knotenfärbungszahl des Graphen genannt.
- Die Bestimmung der chromatischen Zahl eines Graphen ist NP-schwer.
- Das heißt aus Sicht der Komplexitätstheorie, dass sehr wahrscheinlich kein Algorithmus existiert, der dieses Problem effizient löst.

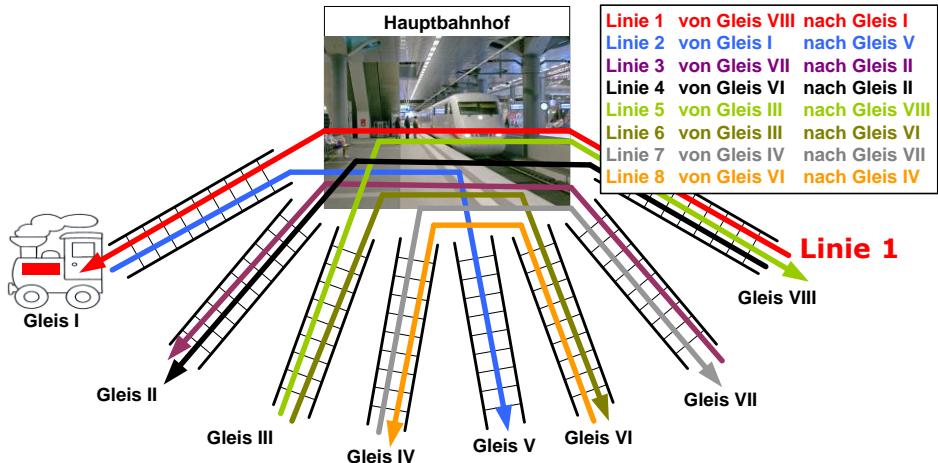


Was hat dies mit Stundenplänen und Fahrplänen zu tun?



- Stundenplanprobleme lassen sich als Graphfärungsprobleme formulieren:
 - Die Knoten des Graphen sind dabei die zu platzierenden Veranstaltungen, und eine Kante wird zwischen zwei Veranstaltungen eingefügt, die nicht gleichzeitig stattfinden können.
 - In der Schule wären das z. B. Stunden, die von demselben Lehrer unterrichtet werden sowie Stunden in derselben Klasse.
 - Die möglichen Farben entsprechen den zuteilbaren Zeitfenstern.
- In der gleichen Weise können beispielsweise
 - Register-Zuweisungs-Probleme in Prozessoren,
 - Bandbreiten-Zuweisung-Probleme oder
 - Fahrpläneals Graphenfärungsprobleme formuliert werden.

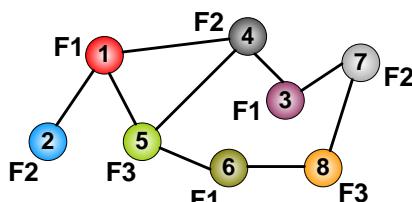
hochschule mannheim
Ein Fahrplan-Problem



- Welche Linien dürfen gleichzeitig fahren?
- Wie viele Fahrplan-Slots braucht man mindestens?

hochschule mannheim
Umwandlung in ein Graphenproblem

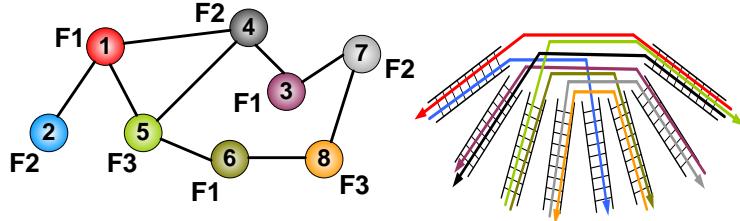
- Eine Zug-Linie:
Ein Knoten im Graph
- Ein Konflikt zweier Linien:
Eine Kante im Graph



- Färbung:
 - Jeder Knoten bekommt eine Farb-Nummer.
 - Knoten, die durch eine Kante verbunden sind, dürfen nicht dieselbe Farb-Nummer bekommen.
- Wie würden Sie möglichst einfach vorgehen?




Greedy-Lösungsansatz

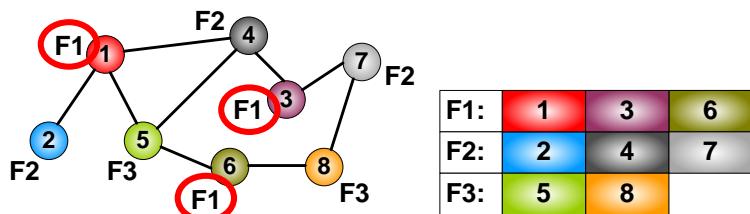
```

Greedy-Algorithmus:
while (nicht alle Knoten gefärbt) {
    x := nächster nicht gefärbter Knoten
    M := Menge der Farben der Nachbarn von x
    farbe(x) := kleinste Farbe, die nicht in M ist
}

```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka
129


Von der Färbung zur Schedule

- Knoten (Zug-Linien) mit derselben Farb-Nummer sind nicht durch eine Kante verbunden:
 - Sie stehen nicht in Konflikt zueinander.
 - Genau die können gleichzeitig ausgeführt werden!
- Ziel ist es, die Anzahl der Farb-Nummern zu minimieren:
 - Chromatische Zahl mit passender Färbung finden.
 - Optimale Ausnutzung der Gleis-Kapazitäten.
 - Kürzeste Taktung des Fahrplans.

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka
130

hochschule mannheim

Java-Implementierung der Färbung: Keine Änderung bei den Kanten



```
import graph.*;
public class KanteFärbung extends Kante {
    public KanteFärbung(Knoten kn1, Knoten kn2) {
        super(kn1, kn2);
    }
}
```

hochschule mannheim

Java-Implementierung der Färbung: Färbbare Knoten



```
import graph.*;
public class KnotenFärbung extends Knoten {
    private int farbe=0;

    public KnotenFärbung(int id) {
        super(id);
    }

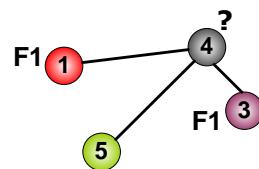
    public void setFarbe(int farbe) {
        if (farbe<0)
            throw new RuntimeException("Ungültige Farbe!");
        this.farbe = farbe;
    }
    public int getFarbe() {
        return farbe;
    }
}
```

hochschule mannheim

Java-Implementierung der Färbung: Färbbare Knoten



```
public int getNächsteFreieFarbe() {
    int freieFarbe=1;
    boolean bereitsGesetzt;
    do{
        bereitsGesetzt=false;
        for (Kante ka:getKanten()){
            int gesetzt=((KnotenFärbung)ka.getAnderesEnde(this)).getFarbe();
            if (gesetzt==freieFarbe){
                freieFarbe++;
                bereitsGesetzt=true;
                break;
            }
        }
    } while (bereitsGesetzt);
    return freieFarbe;
}
```



hochschule mannheim

Java-Implementierung der Färbung: Der Graph



```
import java.util.ArrayList;
import graph.*;
public class GraphFärbung extends Graph{
    public GraphFärbung() {
        super();
    }

    public int getAnzFarben(){
        int max=0;
        for(Knoten kn:getKnoten()){
            KnotenFärbung kf=(KnotenFärbung)kn;
            if (max<kf.getFarbe()) max=kf.getFarbe();
        }
        return max;
    }
}
```



hochschule mannheim
Java-Implementierung der Färbung:
Der Graph

```
public void färben(){
    // Liste der Knoten durchlaufen
    for(Knoten kn:getKnoten()){
        // Greedy: nächste freie Farbe zuweisen
        KnotenFärbung kf=(KnotenFärbung)kn;
        kf.setFarbe(kf.getNächsteFreieFarbe());
    }
}

public String getSchedule(){
    int farben=getAnzFarben();
    String aus[] =new String[farben];
    for(int i=0;i<farben;i++) aus[i]="";
    String ausgabe="";
    for(Knoten kn:getKnoten()){
        KnotenFärbung kf=(KnotenFärbung)kn;
        aus[kf.getFarbe()-1]+=" "+kf.getId()+" ";
    }
    for(int i=0;i<farben;i++) ausgabe+="Slot "+(i+1)+": "+aus[i]+\n";
    return ausgabe;
}
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

135



hochschule mannheim
Java-Implementierung der Färbung:
Die Test-Klasse

```
public class TestGraph {
    public static void main(String[] args) {
        // Graph erzeugen und Konten hinzufügen
        GraphFärbung g=new GraphFärbung();
        for (int i=1;i<=8;i++) g.addKnoten(new KnotenFärbung(i));
        // Kanten hinzufügen
        new KanteFärbung(g.getKnoten(1),g.getKnoten(2));
        new KanteFärbung(g.getKnoten(1),g.getKnoten(4));
        new KanteFärbung(g.getKnoten(1),g.getKnoten(5));
        new KanteFärbung(g.getKnoten(4),g.getKnoten(5));
        new KanteFärbung(g.getKnoten(5),g.getKnoten(6));
        new KanteFärbung(g.getKnoten(6),g.getKnoten(8));
        new KanteFärbung(g.getKnoten(8),g.getKnoten(7));
        new KanteFärbung(g.getKnoten(7),g.getKnoten(3));
        new KanteFärbung(g.getKnoten(3),g.getKnoten(4));
        // Greedy-Färbung
        g.färben();
```

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

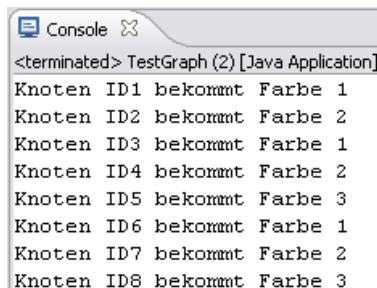
136

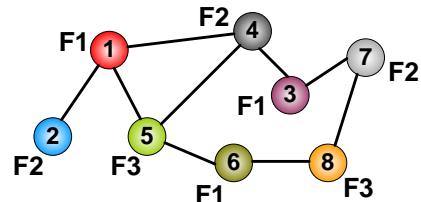
hochschule mannheim

Java-Implementierung der Färbung: Die Test-Klasse



```
// Ausgabe der Farben der einzelnen Knoten
for (int i=1;i<=8;i++){
    System.out.println("Knoten ID"+g.getKnoten(i).getId()
        +" bekommt Farbe "
        +((KnotenFärbung)g.getKnoten(i)).getFarbe());
}
```

 Console <terminated> TestGraph (2) [Java Application]
Knoten ID1 bekommt Farbe 1
Knoten ID2 bekommt Farbe 2
Knoten ID3 bekommt Farbe 1
Knoten ID4 bekommt Farbe 2
Knoten ID5 bekommt Farbe 3
Knoten ID6 bekommt Farbe 1
Knoten ID7 bekommt Farbe 2
Knoten ID8 bekommt Farbe 3



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

137

hochschule mannheim

Java-Implementierung der Färbung: Die Test-Klasse



```
// Ausgabe der Schedule
System.out.println("Schedule greedy-gefärbt:");
System.out.println(g.getSchedule());
}

Schedule greedy-gefärbt:
Slot 1: 1 3 6
Slot 2: 2 4 7
Slot 3: 5 8
```

F1:	1	3	6
F2:	2	4	7
F3:	5	8	

Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

138