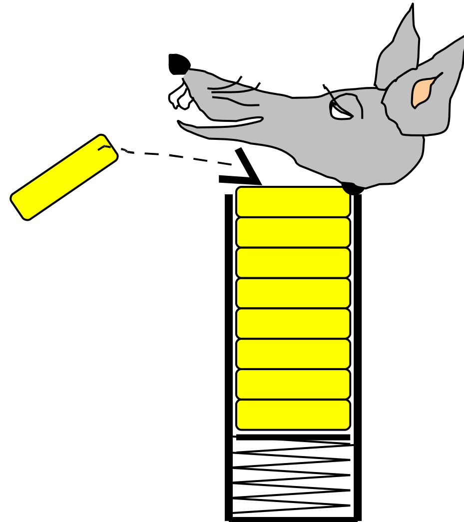
Einführung in die Informatik
- Teil XIII –
Abstrakte Datentypen
und dynamische Klassen



Inhalt:

1. Abstrakte Datentypen
2. Ein Beispiel: ADT Stapel
3. Dynamische Datenstrukturen in Java
4. Listen in Java
5. Generische Listen
6. Aufgaben
7. Bäume
8. Aufgaben

Literaturhinweise:

- Küchlin/Weber: Einführung in die Informatik, Objektorientiert mit Java, Springer 1998
- Krüger, Guido: Handbuch der Java-Programmierung, http://www.javabuch.de oder Addison Wesley 2002

1. Abstrakte Datentypen

Das ziemlich alte Konzept der *abstrakten Datentypen* (ADTs) versucht, die Nutzung eines Datentyps von seiner Implementierung zu trennen. (Zur Information: Es gibt auch *abstrakte Datenstrukturen*. Von diesen existiert dann nur eine einzige Instanz.) Der Vorteil des Konzepts liegt darin, dass die Implementierung geändert werden kann (z. B. bei der Fehlerbeseitigung oder zur Effizienzsteigerung), ohne alle Anwendungsprogramme mit zu ändern. Solange Anwender nur *Zugriffsoperationen* auf den Datentyp nutzen, also eine definierte Schnittstelle, solange haben sie mit der inneren Struktur des ADTs nichts zu tun. **Konsequenterweise werden ADTs über ihre Zugriffsoperationen definiert.** Die Klassen der OOP sind typische Implementierungen von ADTs, solange nicht direkt auf innere Felder zugegriffen wird. (Stattdessen schreibt man besondere Methoden: *get(wert)* bzw. *set(wert)*: man benutzt *Eigenschaften* oder *Properties*.)

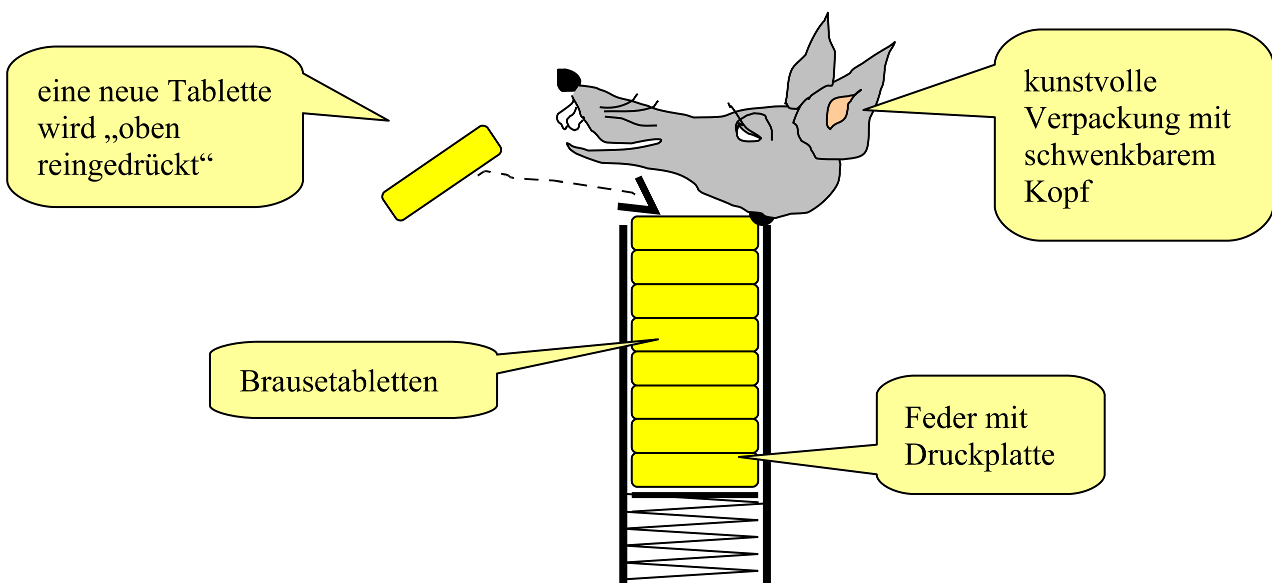
ADTs sind an der Uni ein durchmathematisiertes Konzept. Für die Schule haben sie andere Vorteile:

- Die Trennung von Modell und Implementierung ermöglicht abstraktes Arbeiten im Modell.
- Die Trennung von Nutzung und Implementierung ermöglicht es, die Teilprobleme überschaubar zu halten.
- Das Konzept liefert gute Problemstellungen für Leistungsmessung.
- Arbeitsteiliger Unterricht wird erleichtert.

2. Ein Beispiel: ADT Stapel

Zuerst das Standardbeispiel für ADTs: Wir wollen die Datenstruktur *Stapel* definieren. Was ist ein „Stapel“?

Das Modell: Unabhängig von der tatsächlichen Implementierung benötigen wir eine Vorstellung davon, wie Zugriffsoperationen sich auf einen Stapel auswirken. Wir stellen uns also eine Art „Brausetablettenspender“ vor, dem Tabletten nur oben entnommen oder hinzugefügt werden können.



Versuchen wir einmal eine formale Beschreibung der Vorgänge.

Wir setzen voraus, dass die *Sorten* *e* (Elemente, die auf dem Stapel abgelegt werden) und *b* (boolescher Wahrheitswert) bekannt sind. Dann benötigen wir nur noch die neue

Sorte: *s (Stapel).*

Auf dieser vereinbaren wir die folgenden

Operationen:

neu	*: → s*	liefert einen neuen leeren Stapel „aus dem Nichts“
leer	*: s → b*	stellt fest, ob ein Stapel leer ist
push	*: s x e → s*	legt ein Element e auf dem Stapel s ab
pull	*: s → s*	entfernt das oberste Element vom Stapel s
oben	*: s → e*	liefert das oberste Element e des Stapels s

Jetzt muss noch die genaue Arbeitsweise der Operationen auf den Sorten definiert werden. Das geschieht durch die

Axiome:

leer(neu)	*= true*	ein neuer Stapel ist leer
leer(push(s,e))	*= false*	ein Stapel, auf den man etwas gepackt hat, ist nicht leer
pull(push(neu,e))	*= neu*	usw.
pull(push(push(s,e1),e2))	*= push(pull(push(s,e2)),e1)*	
oben(push(s,e))	*= e*	
oben(push(push(s,e1),e2))	*= oben(push(s,e2))*	

Es stellt sich die Frage, ob dieses Axiomensystem den ADT vollständig beschreibt, und wenn, ob das System minimal ist, ob es äquivalente Axiomensysteme gibt, ... Diese Fragen überlassen wir den Lehrbüchern.

Wenn wir auf Beweise verzichten und stattdessen „nur“ sorgfältig argumentieren, dann ist das Basteln an einem Axiomensystem aber – für ganz wenige Stunden – eine abwechslungsreiche und besonders leistungsstarke Schüler/innen stark motivierende Beschäftigung. Dagegenzurechnen ist die abschreckende Wirkung solcher Beschäftigung auf ebenfalls nicht wenige Schüler/innen, die durch die Formalisierung – ohne Beweise – auch nichts Neues erfahren, denn im System steht nichts, was man nicht auch mit Worten sagen könnte. Man sollte in jedem Fall vermeiden, die formale Beschreibung als Selbstzweck zu betreiben, und man sollte deutlich darauf hinweisen, dass ohne Konsequenzen (also ohne die Anwendung des mathematischen Apparats zum Zwecke des Schlussfolgerns) dieser Formalismus eigentlich leer und hier überflüssig ist.

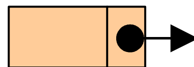
Im Folgenden werden wir ADTs ohne große Formalisierung eher im „schulischen“ Sinne begreifen.

3. Dynamische Datenstrukturen in Java

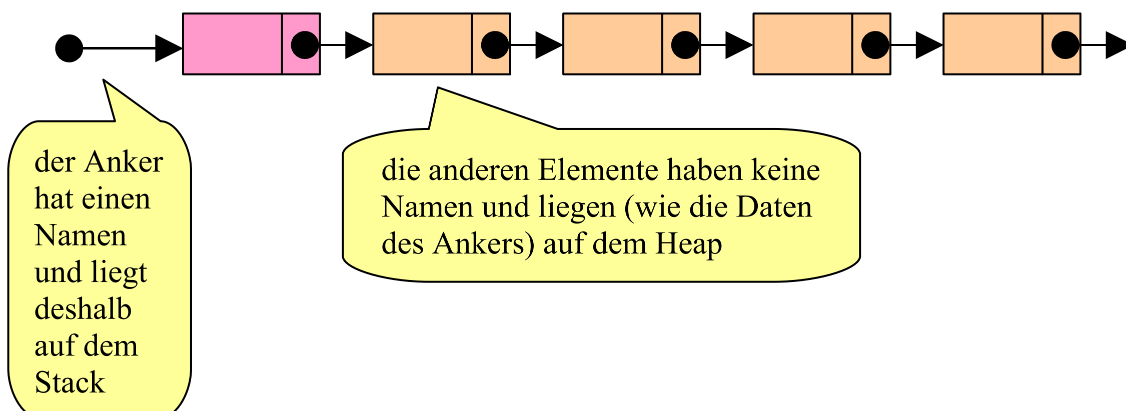
Unter *dynamischen Datenstrukturen* versteht man Gebilde, deren Größe und Gestalt zum Zeitpunkt der Übersetzung des Programms noch nicht festliegen. (In diesem Sinne sind auch Arrays in Java dynamisch.) Vor allem aber können sich Größe und Gestalt während des Programmlaufs ändern. Typische Beispiele sind *Listen*, *Stapel*, *Schlangen*, *Bäume* und *Netze*. Entsprechende Klassen (*Vektor*, *Stack* und diverse *Collections*) sind in Java vorhanden. Da wir aber nicht einen Java-Kurs veranstalten, sondern einen Informatikkurs mithilfe von Java, wollen wir entsprechende Strukturen selbst implementieren. Danach können wir sie mit den vorhandenen vergleichen.

Wenn die Zahl der Daten vor der Übersetzung noch nicht bekannt ist, dann kann man die Daten auch nicht benennen, also einzeln in benannten Variablen unterbringen. Man verwendet also Referenzen (Zeiger), die im Programm gesetzt werden und auf Speicherbereiche zeigen, die die Daten enthalten. Üblicherweise enthalten diese Bereiche dann weitere Zeiger auf andere Daten. Man findet häufig die Aussage „in Java gibt es keine Zeigertypen". Die Aussage stimmt so nicht, denn Java kennt Referenztypen und damit auch Zeiger. Was man mit Java kaum machen kann, sind die „schmutzigen Tricks", die in anderen Sprachen möglich sind. Man kann also Zeiger nicht direkt manipulieren. Für dynamische Strukturen können wir nur die üblichen Javareferenzen benutzen, am besten gleich Referenzen auf Objekte.

Das übliche Basiselement dynamischer Strukturen ist ein Kästchen (Knoten), das Daten und mindestens eine weite Referenz auf andere Knoten (Kanten) enthält. Symbolisieren wir Referenzen durch Pfeile, dann enthalten wir z. B.:



Eins dieser Elemente müssen wir erreichen können, um den Anfang der Struktur zu markieren. Wir benötigen also wenigstens eine benannte Variable dieser Art: den *Anker*. Von diesem aus erreichen wir den Rest durch Verfolgen der Referenzen. Hängen wir einfach Elemente an vorhandene an, dann entsteht eine *Liste*.



In Listen können neue Elemente an beliebiger Stelle eingefügt werden. Beschränken wir die Zugriffe auf ein Ende, dann haben wir einen *Stapel*, fügen wir an einem Ende ein und schneiden am anderen Elemente ab, dann entsteht eine *Schlange*.

Auf dynamischen Datenstrukturen wird üblicherweise mit rekursiven Methoden gearbeitet. Begründet ist das durch den „selbstähnlichen" Aufbau der Strukturen: Der Rest einer Liste ist wieder eine Liste, und die Teile eines Baums sind wiederum Bäume. Folglich werden auch die Strukturen selbst rekursiv definiert.

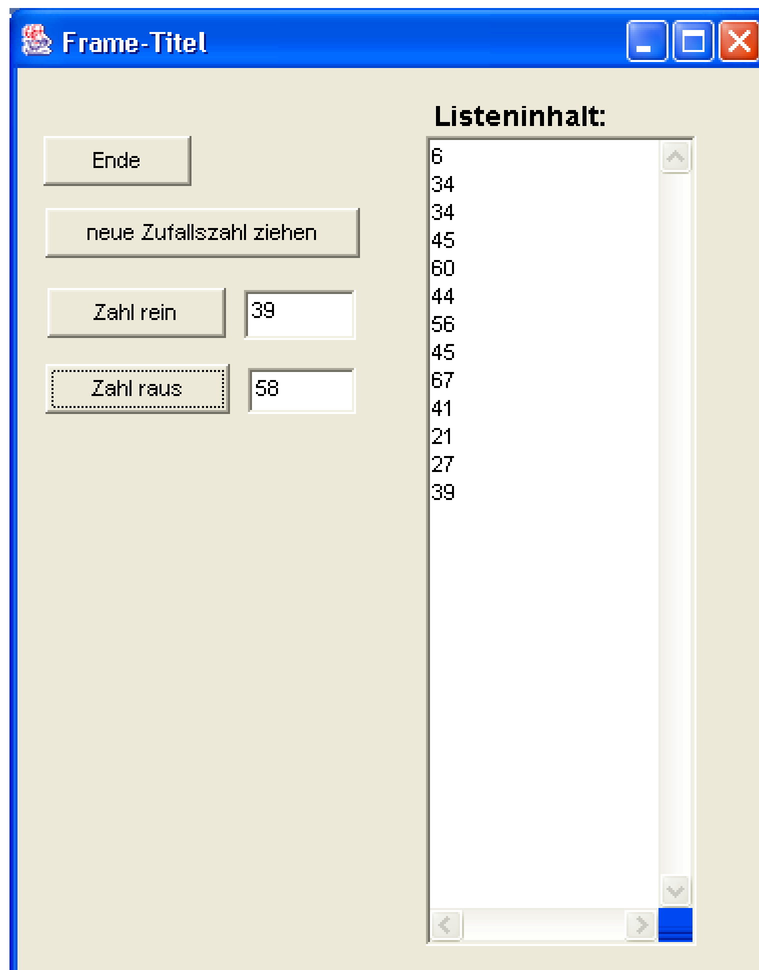
4. Listen in Java

Wir wollen eine einfache Listenstruktur in Java erzeugen. Dazu vereinbaren wir einen abstrakten Datentyp Liste, der durch folgende Zugriffsoperationen definiert ist, deren Arbeitsweise wir nur „intuitiv" erfassen, also als Arbeit im Modell der aneinandergehängten Kästchen.

Der ADT Liste:

Liste()	Konstruktor, erzeugt eine neue leere Liste
istLeer	stellt fest, ob die Liste leer ist
rein	fügt ein Element in die Liste ein
raus	liefert das erste Element der Liste und entfernt es aus dieser
drin	liefert die Anzahl der Elemente der Liste
toString	verwandelt den Inhalt der Liste in ein String-Array, das dann dargestellt werden kann

Verwenden wir ganze Zahlen als Listeninhalte, dann können wir die Liste leicht mit Zufallszahlen füllen. Wir erhalten dann eine Oberfläche wie z. B. diese.



Wir haben dann die üblichen Verhältnisse: In Hintergrund arbeitet eine „intelligente" Struktur, die zu entwickeln einigen intellektuellen Reiz hat. Den Vordergrund „klickt man sich schnell zusammen", ohne dafür zu viel Zeit zu verschwenden. Der Vordergrund dient weitgehend zur Anzeige der Ergebnisse.

Wie arbeitet eine Liste?

Wir wollen ganze Zahlen speichern, also liegt der *Inhalt* der Liste fest. Haben wir eine neue Liste „erschaffen“, dann hat dieser Inhalt einen Wert (wahrscheinlich Null), weil es sich um eine Reihe Bits handelt, die in einem von zwei Zuständen vorliegen. Wir können dieser Bitfolge nicht ansehen, ob sie schon einen gespeicherten Wert darstellt oder eine zufällige Anfangskombination. Deshalb führen wir ein weiteres boolesches Attribut namens *frei* ein, das angibt, ob der Inhalt schon vergeben wurde. Da der angehängte Rest einer Liste wieder eine Liste darstellt, definieren wir Listen rekursiv: Das Attribut *naechster* stellt wieder eine Liste dar. Im Konstruktor belegen wir diese Attribute mit Anfangswerten.

```
public class Liste
{
      private int inhalt;
      private boolean frei;
      private Liste naechster;

      public Liste()
      {
            inhalt = 0;
            frei = true;
            naechster = null;
      }
```

Attribute der Liste

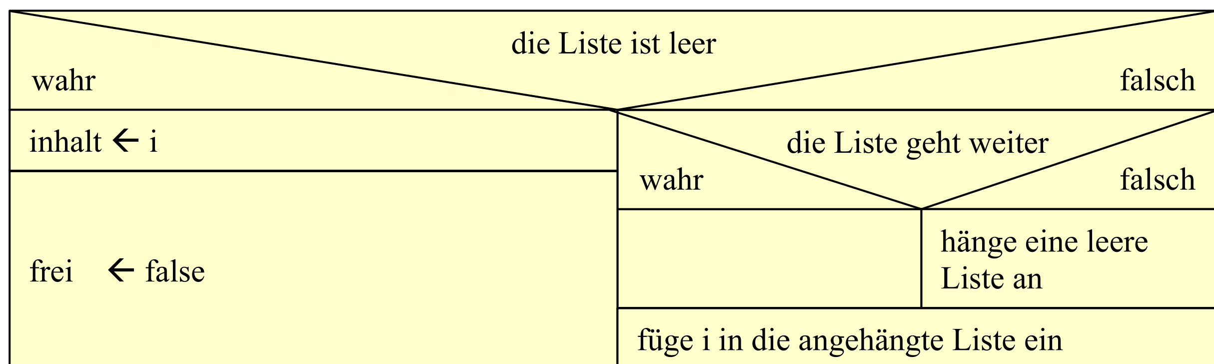
Konstruktor: *null* kennzeichnet eine Referenz „ins Leere“

Es ist ziemlich einfach festzustellen, ob eine Liste leer ist:

```
      public boolean istLeer()
      {
            return frei;
      }
```

Nicht ganz so einfach ist es, neue Elemente in eine Liste einzufügen. Wir machen das rekursiv, das ist kurz, knapp und elegant: Entweder hat die Liste vorne noch Platz (dann können wir die neue Zahl dort speichern) oder die Zahl wird nach hinten durchgereicht.

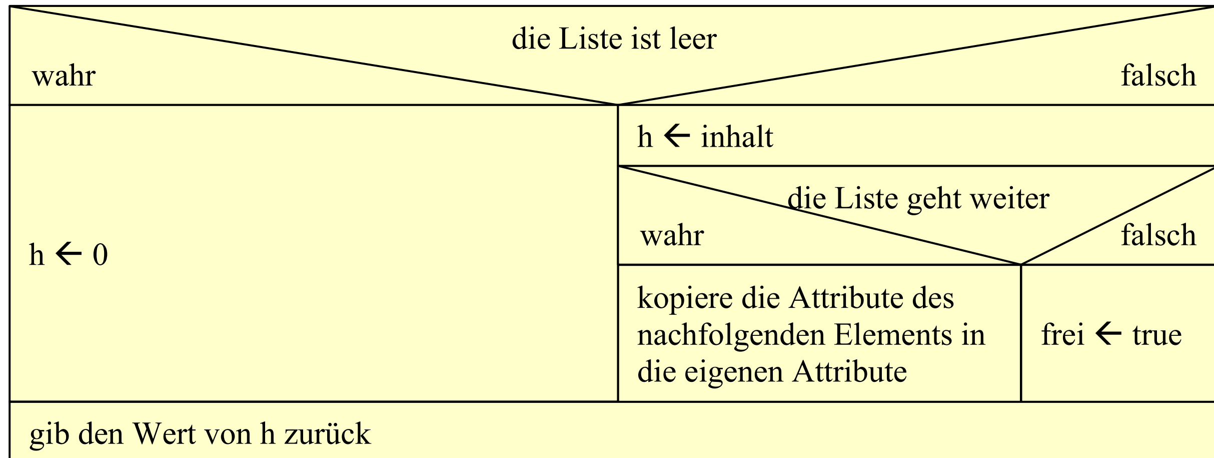
public void rein(int i):




```
      public void rein(int i)
      {
            if(frei)
            {
                  inhalt = i;
                  frei = false;
            }
            else
            {
                  if(naechster==null) naechster = new Liste();
                  naechster.rein(i);
            }
      }
```

Jetzt wird es schwieriger: Da wir den Anfang einer Liste nicht verlieren dürfen, können wir nicht einfach vorne ein Element abschneiden. Wir müssen deshalb bei Bedarf DAS ZWEITE Elemente der Liste löschen, nachdem wir dessen Inhalt in das erste gerettet haben.

public int raus():




```
public int raus()
{
      int h;
      if(!frei)
      {
            h = inhalt;
            if(naechster==null) frei = true;
            else
            {
                  inhalt    = naechster.inhalt;
                  frei      = naechster.frei;
                  naechster = naechster.naechster;
            }
      }
      else h = 0;
      return h;
}
```

Die Anzahl der Elemente der Liste können wir wieder klassisch rekursiv bestimmen:

```
public int drin()
{
      if(frei) return 0;
      else if(naechster==null) return 1;
           else return 1 + naechster.drin();
}
```

Um zu zeigen, dass man in Java Zeiger auch ziemlich direkt benutzen kann, wollen wir den Inhalt der Liste in eine String-Reihung verwandeln. Die kann dann am Bildschirm dargestellt werden. Wir vereinbaren dazu bei Bedarf eine neue Liste (also eine Referenz), die wir die ganze Liste innerhalb einer Schleife durchlaufen lassen. (Das ist zwar nicht fein, funktioniert aber!)

```
public String[] toLines()
{
      int drin = drin();
      String[] lines;
```

```
        if (drin==0)
        {
              lines = new String[1];
              lines[0] = "---";
        }
        else
        {
              lines = new String[drin];
              Liste h = this;
              int i=0;

              while(h!=null)
              {
                    lines[i] = ""+h.inhalt;
                    i++;
                    h = h.naechster;
              }
        }
        return lines;
  }
```

wenn nichts da ist, geben wir auch nichts zurück

jetzt wissen wir, wie groß das Feld sein muss

die Hilfsliste h erhält anfangs den Wert der Liste, die durchlaufen werden soll

Liste durchlaufen und Inhalte in die String-Reihung schreiben

Zur Übersicht die gesamt Klasse:

```
public class Liste
{
      private int inhalt;
      private boolean frei;
      private Liste naechster;

      public Liste()                 // wie angegeben
      public boolean istLeer()       // wie angegeben
      public void rein(int i)        // wie angegeben
      public int raus()              // wie angegeben
      public int drin()              // wie angegeben
      public String[] toLines()      // wie angegeben
}
```

Wie nutzt man jetzt diese Liste?

Wir vereinbaren eine Liste mit `Liste l = new Liste();`

und eine globale Variable Zufallszahl `int zz;`

Haben wir eine neue Zufallszahl gezogen, dann können wir sie in die Liste einfügen

```
  void b_rein_actionPerformed(ActionEvent e)
  {
    l.rein(zz);
    zeigeListe();
  }
```

Entsprechend können wir Zahlen entfernen

```
  void b_raus_actionPerformed(ActionEvent e)
  {
    int h = l.raus();
    tf_raus.setText(""+h);
    zeigeListe();
  }
```

Die Liste zu zeigen ist auch nicht mehr schwierig.

```
private void zeigeListe()
{
    String h = "";
    String[] zeilen = l.toLines();
    for(int i=0;i<l.drin();i++)
    {
      h = h + zeilen[i]+(char)13; //alternativ: ... +"\n";
    }
    anzeige.setText(h);
}
```

5. Generische Listen

Statt Listen zu implementieren, die nur einen bestimmten Datentyp aufnehmen können, sollten wir die Listeneigenschaft „als solche" in einer eigenen Klasse implementieren. Diese kann dann diese Eigenschaft auf Tochterklassen vererben, die genauer spezifizieren, welche Daten aufgenommen werden sollen.

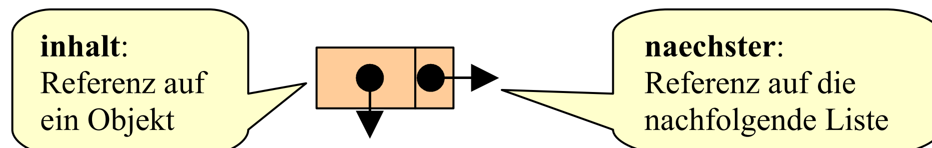
In Java ist ein solches Vorhaben ziemlich einfach. Da alle Objekte von der Urmutter *Object* abgeleitet werden, können wir einfach Objekte in die Liste aufnehmen. Solche Referenzen sind dann mit Tochterklassen kompatibel. Nur wenn primitive Datentypen (*int*, *double*, ...) gespeichert werden sollen, müssen wir sie zuerst in einer *Wrapper-Klasse* verpacken. Wollen wir z. B. die ganze Zahl 37 speichern, dann benutzen wir den Aufruf *new Integer(37);*.

Versuchen wir es mal!

Natürlich verpacken wir generische Listen in einer eigenen Klasse. Diese enthält als Inhalt eine Referenz auf ein beliebiges Objekt und eine weitere Referenz auf die folgende generische Liste.

```
public class GenerischeListe
{
      Object inhalt;
      GenerischeListe naechster;
```

Stellen wir uns die Elemente dieser Klasse wie üblich als Kästchen vor und Referenzen als Zeiger, dann erhalten wir die folgende Form:



Der Konstruktor der Klasse initialisiert die Referenzen: sie zeigen anfangs ins Leere.

```
      public GenerischeListe()
      {
            inhalt = null;
            naechster = null;
      }
```

Damit kann man leicht feststellen, ob die Liste leer ist!

```
      public boolean istLeer()
      {
            return (inhalt == null);
      }
```

Das rekursive Verfahren zum Einfügen neuer Inhalte kennen wir schon.

```
	public void rein(Object neu)
	{
		if(istLeer()) inhalt = neu;
		else
		{
			if(naechster==null) naechster = new GenerischeListe();
			naechster.rein(neu);
		}
	}
```

Und ebenso die Vorgänge beim Entfernen des ersten Elements.

```
	public Object raus()
	{
		Object h;
		if(istLeer()) return null;
		else
		{
			h = inhalt;
			if(naechster == null) inhalt = null;
			else
			{
				inhalt = naechster.inhalt;
				naechster = naechster.naechster;
			}
			return h;
		}
	}
```

Das war’s schon! Jetzt noch mal alles im Zusammenhang:

```
public class GenerischeListe
{
	Object inhalt;
	GenerischeListe naechster;

	public GenerischeListe()
	{
		inhalt = null;
		naechster = null;
	}

	public boolean istLeer()
	{
		return (inhalt == null);
	}

	public void rein(Object neu)
	{
		if(istLeer()) inhalt = neu;
		else
		{
			if(naechster==null) naechster = new GenerischeListe();
			naechster.rein(neu);
		}
	}

	public Object raus()
	{
		Object h;
		if(istLeer()) return null;
		else
```

```
            {
                  h = inhalt;
                  if(naechster==null) inhalt = null;
                  else
                  {
                        inhalt = naechster.inhalt;
                        naechster = naechster.naechster;
                  }
                  return h;
            }
      }
}
```

Jetzt wollen wir die generische Liste nutzen. Zuerst leiten wir eine Stringliste daraus ab. Wenn wir uns die Sache genauer ansehen, dann müssen wir fast nichts tun, da es sich bei Strings ja schon um Referenzen handelt. Nur beim Auslesen der Inhalte müssen wir noch ein Typecasting durchführen, da die Liste nicht „weiß", welche Art von Objekt in ihr gespeichert wurde[1].

```
public class Stringliste extends GenerischeListe
{
     public String stringRaus()
     {
         return (String)super.raus();
     }
```

Typecasting

Bei primitiven Datentypen müssen wir die entsprechenden Wrapperklassen benutzen, deshalb müssen wir auch die Eingabemethode neu schreiben:

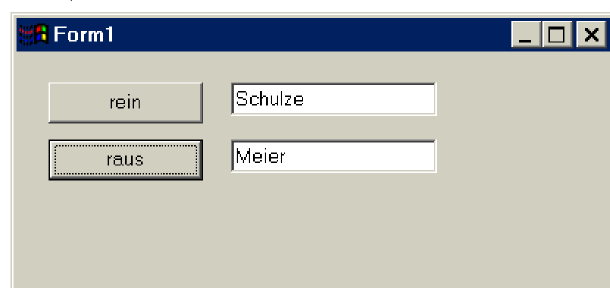
```
public class Integerliste extends GenerischeListe
{

     public void intRein(int i)
     {
         rein(new Integer(i));
     }

     public int intRaus()
     {
         Integer h = (Integer)raus();
         return h.intValue();
     }
}
```

Wrapperklasse für ganze Zahlen

Objekt rausholen und in ganze Zahl verwandeln

Die Aufrufe in Eventhandlern sind dann simpel.

```
Stringliste l = new Stringliste();

void b_rein_actionPerformed(ActionEvent e)
{
     l.rein(textField1.getText());
}

void b_raus_actionPerformed(ActionEvent e)
{
     textField2.setText
                        (l.stringRaus());
}
```



[1] In neueren Java-Versionen geht das eleganter.

6. Aufgaben

1. a: Beschreiben Sie den **ADT Stapel** durch die Zugriffsoperationen *push*, *pull* und *top*.

   b: Implementieren Sie eine Klasse *Intstapel*, die ganze Zahlen speichern kann. Nutzen Sie die Klasse zur Lösung des Problems der „Türme von Hanoi".

   c: Implementieren Sie die generische Klasse Stapel und nutzen Sie diese zur Implementation von Stringstapeln, Stapeln für Fließkommazahlen usw.

   b: Machen Sie sich mit der Arbeitsweise der Javaklasse *Stack* vertraut. Vergleichen Sie diese mit Ihren Ergebnissen.

2. Implementieren Sie die Klasse **GeordneteListe**, die Zahlen aufsteigend sortiert an der richtigen Stelle in die Liste einfügt,

   a: als Klasse für ganze Zahlen

   b: als generische Klasse, aus der sowohl Zahlenlisten wie Stringlisten abgeleitet werden. Wo liegen die Probleme?

   c: Jetzt sollen Datumswerte geordnet in die Liste eingefügt werden.

3. a: Verfahren Sie wie bei Aufgabe 1 für den **ADT Schlange**: Vorne werden Daten eingefügt, hinten abgeschnitten.

   b: Verwalten Sie das Wartezimmer eines Arztes mit diesem Datentyp.

   c: Machen Sie sich mit der Arbeitsweise der Klasse Queue vertraut, die in der Literatur fast überall beschrieben wird. Vergleichen Sie deren Funktionsweise mit der von Ihnen implementierten.

4. Im Beispielprogramm **Physikobjekte** können nicht voraussehbare Mengen von Geräten und Leitungen erzeugt werde.

   a: Diskutieren Sie das Problem in Hinsicht auf die Möglichkeit, dynamische Datenstrukturen einzusetzen.

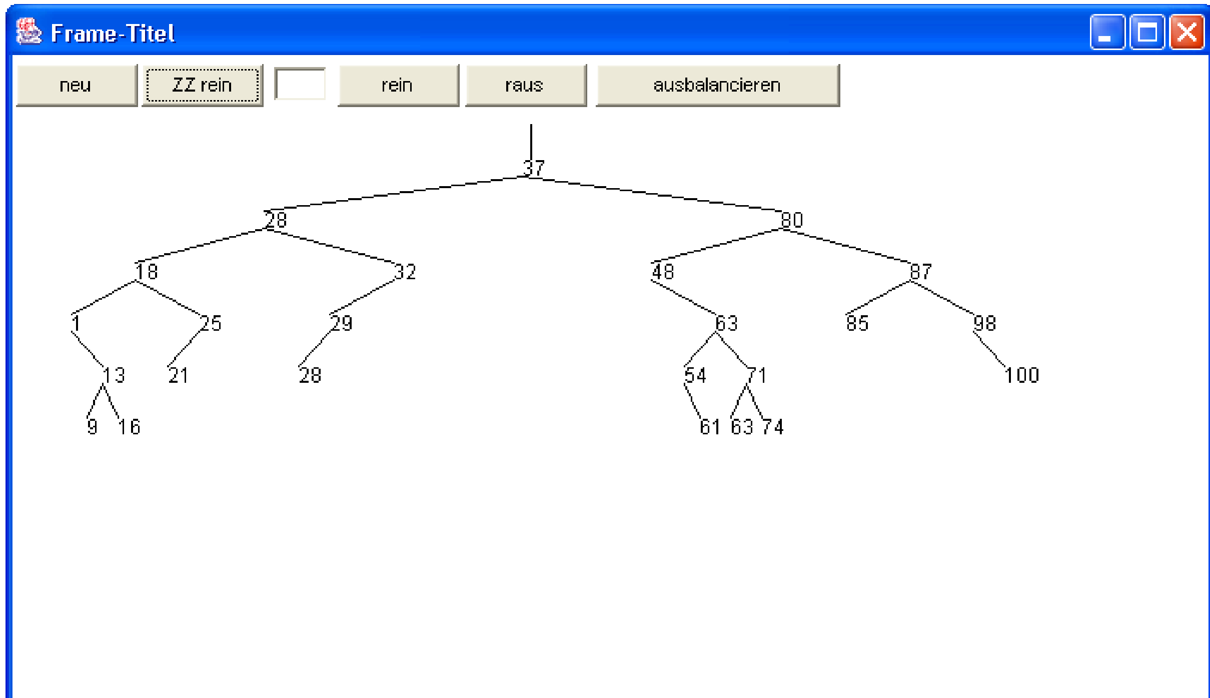
   b: Entwickeln Sie eine Klasse Leitung, die Buchsen der Geräte verbindet. Leitungen sollen durch Mausklicks gesetzt, gezeichnet <u>und verändert</u> werden können. Testen Sie Ihre Lösung.

5. Beschäftigen Sie sich mit dem Datentyp **Menge**. Machen Sie sich in der Literatur mit seiner Funktionalität und seinen Einsatzmöglichkeiten vertraut.

6. Spezifizieren Sie einige der genannten ADTs (z. B. die Schlange oder die geordnete Liste) wie in Teil 2 gezeigt formal über Sorten, Operationen und Axiome. Vergleichen Sie Ihre Ergebnisse mit den im Internet (in den zahlreichen Vorlesungsskripten) oder der Literatur zu findenden.

7. Bäume

Neben Listen, Stapeln und Schlangen spielen Bäume eine große Rolle bei Sortier- und Suchproblemen. Betrachten wir dazu einmal einen *binären Sortierbaum* für ganze Zahlen. Dieser enthält *Blätter*, die eine Zahl aufnehmen können, und zwei weitere *Zweige*, die Referenzen auf Teilbäume darstellen, die auf der einen Seite nur Zahlen enthalten, die kleiner als die im Blatt gespeicherte sind, auf der anderen Seite größer.



Suchen wir in diesem Baum eine Zahl, dann brauchen wir nicht alle gespeicherten Zahlen zu kontrollieren, sondern durchlaufen nur die in Frage kommenden Zweige. Kommen wir bei der Suche an ein Ende – also an einen leeren Ast, dann kann die Zahl sich nicht im Baum befinden. Da in einem Binärbaum pro Ebene 2^n Zahlen Platz finden (wenn man die Zählung bei *0* beginnt), enthält ein Baum mit n Ebenen maximal $2^{n+1}-1$ Blätter.

$$1+2+4+8+\ldots+2^n=\sum_{i=0}^{n}2^i=2^{n+1}-1$$

Fragen wir jetzt umgekehrt nach der Anzahl der Operationen, die bei Suchvorgängen in einer Menge von *N* Zahlen erforderlich sind, die in einem Binärbaum gespeichert wurden, dann erhalten wir:

$$N\le 2^{n+1}-1 \quad\rightarrow\quad N+1\le 2^{n+1} \quad\rightarrow\quad ld(N+1)\le n+1 \quad\rightarrow\quad n\ge ld(N+1)-1$$

Ist der Baum gut organisiert, dann sind etwa *ld(N)* Ebenen und damit etwa *ld(N)* Operationen erforderlich, statt alle *N* Zahlen zu vergleichen. Bei großen Zahlen ist das gravierend:

N	ld N	gerundet
10	3,32192809	3
100	6,64385619	7
1000	9,96578428	10
10000	13,2877124	13
100000	16,6096405	17
1000000	19,9315686	20
10000000	23,2534967	23

Mit 23 Vergleichsvorgängen kann als festgestellt werden, ob eine Zahl unter 10.000.000 zu finden ist. Die riesigen Datenmengen der Suchmaschinen des Internets z. B. sind ohne den Einsatz von Suchbäumen gar nicht in vertretbarer Zeit zu durchwühlen.

Führen wir jetzt also eine Baumklasse zur Speicherung ganzer Zahlen ein. Wir benötigen dafür Platz für die zu speichernde Zahl sowie zwei Referenzen – und aus dem gleichen Grund wie bei den Listen eine boolesche Variable, die angibt, ob der Platz belegt ist. Da wir den Baum auch zeichnen wollen, geben wir den dafür vorgesehenen Grafikkontext in der Zeichenmethode als Parameter an.

```
public class Baum
{
    int inhalt;
    boolean frei;
    Baum links,rechts;
```

Der Konstruktor initialisiert diese Felder.

```
    public Baum()
    {
       inhalt = 0;
       frei = true;
       links = null;
       rechts = null;
    }
```

Jetzt fügen wir Zahlen ein: Ist das Blatt noch frei, dann speichern wir die Zahl und markieren das Blatt als belegt. Ansonsten vergleichen wir die neue Zahl mit dem gespeicherten Inhalt. Ist der größer als die Zahl, dann wird diese links eingefügt, sonst rechts. Dafür müssen wir bei Bedarf neue Bäume erzeugen und den Job rekursiv an diese weiterreichen. (Zahlen sollen nur einmal im Baum auftauchen.)

```
    public void rein(int i)
    {
       if(frei)
       {
             inhalt = i;
             frei = false;
       }
       else
       {
             if(I < inhalt)
             {
                   if(links == null) links = new Baum();
                   links.rein(i);
             }
             else if(I > inhalt)
                    {
                         if(rechts == null) rechts = new Baum();
                         rechts.rein(i);
                    }
       }
    }
```

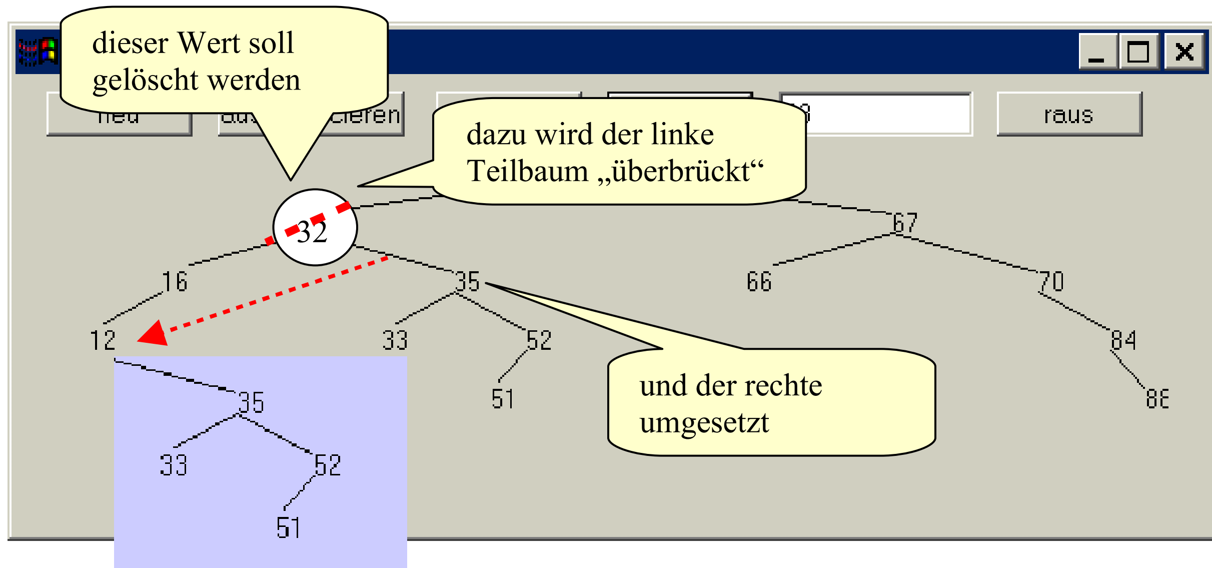
Wert in das leere Blatt einfügen

bei Bedarf neuen Baum erzeugen ...

... und den Job weitergeben.

Komplizierter ist das Löschen von Inhalten, weil wir einerseits im Blatt löschen, aber andererseits dem übergeordneten Blatt mitteilen müssen, ob der Teilbaum jetzt leer ist – denn dann kann die entsprechende Referenz gelöscht werden. Es gibt dafür viele Möglichkeiten. Hier formulieren wird die Methode *raus* als Funktion, die einen booleschen Wert zurückgibt. Ist dieser *true*, dann ist der Teilbaum leer.

Wie löscht man also?



Das Umsetzen von Teilbäumen benötigen wir noch an anderer Stelle, deshalb führen wir zwei Methoden dafür ein:

```
private void rechtsuntenran(Baum b)
{
      if(rechts == null) rechts = b;
      else rechts.rechtsuntenran(b);
}

private void linksuntenran(Baum b)
{
      if(links == null) links = b;
      else links.linksuntenran(b);
}
```

Damit können wir jetzt Inhalte löschen:

```
public boolean raus(int i) {
       if(inhalt == i)
       {
        if(links != null)
        {
            if(rechts == null)rechts = links.rechts;
            else rechts.rechtsuntenran(links.rechts);
            inhalt = links.inhalt;
            links = links.links;
            }
       else
            if(rechts != null)
            {
                if(links == null)links = rechts.links;
                else links.linksuntenran(rechts.links);
                inhalt = rechts.inhalt;
                rechts = rechts.rechts;
            }
            else frei = true;
       }
       if(links != null) if(links.raus(i)) links = null;
       if(rechts != null) if(rechts.raus(i)) rechts = null;
       if((links == null)&&(rechts == null) && frei) return true;
       else return false;
       }
}
```

Wenn die Zahl gefunden wurde ...

... kopieren wir ggf. das linke Blatt (und löschen damit das jetzige) und überbrücken es.

sonst versuchen wir das Gleiche rechts

Klappt auch das nicht, ist der Baum leer

auch in den Teilbäumen löschen

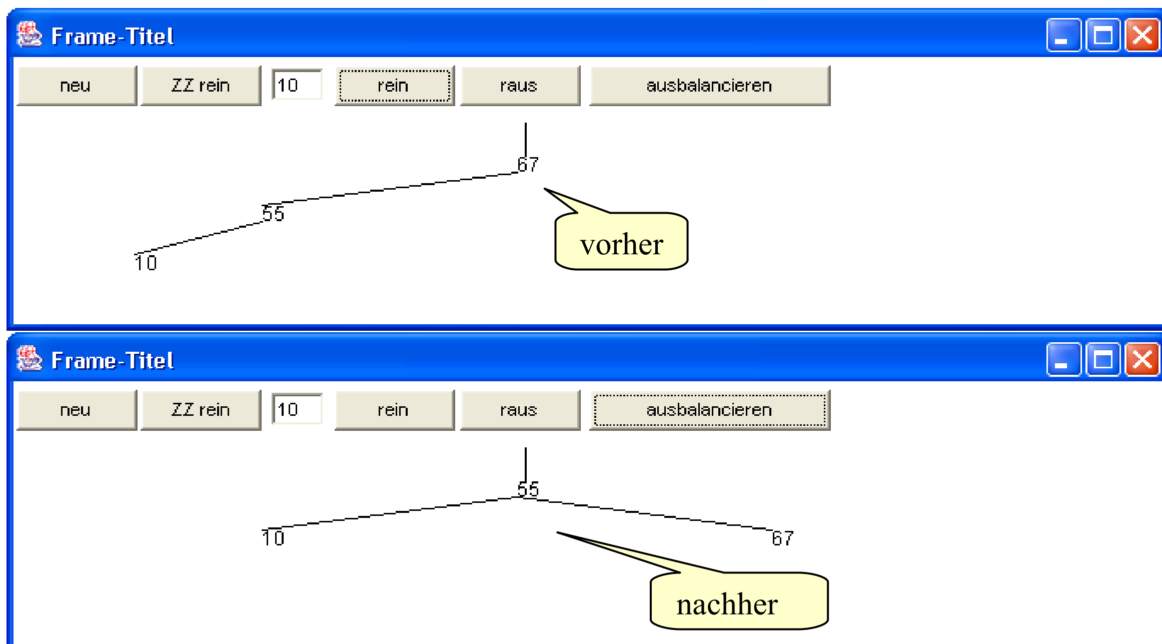
Jetzt wollen wir den Baum zeichnen. Dafür benötigen wir einen Grafikkontext, den das aufrufende Programm zu liefern hat. Weiterhin übergeben wir dem zu zeichnenden (Teil-)Baum die Koordinaten, an denen die Zahl erscheinen soll, und eine Breite, auf die der folgende Baum zu verteilen ist. Wählen wir anfangs für diese Breite das halbe Bildschirmfenster, dann haben die beiden Teilbäume gleichviel Platz auf beiden Seiten. Halbieren diese Teilbäume jeweils den Wert, um wiederum ihre Teilbäume zu zeichnen, dann können wir einen genügend großen Baum auf dem Bildschirm zeigen.

```
public void zeigeBaum(int x, int y, int breite, Graphics g)
{
    g.setColor(Color.black);
    if(!frei)
    {
            g.drawString(""+inhalt,x,y-13);
            int bneu = (int)Math.round(breite/2);
            if(links != null)
            {
                    int xneu = x - bneu;
                    g.drawLine(x,y,xneu,y+20);
                    links.zeigeBaum(xneu,y+30,bneu,g);
            }
            if(rechts != null)
            {
                    int xneu = x + bneu;
                    g.drawLine(x,y,xneu,y+20);
                    rechts.zeigeBaum(xneu,y+30,bneu,g);
            }
    }
}
```

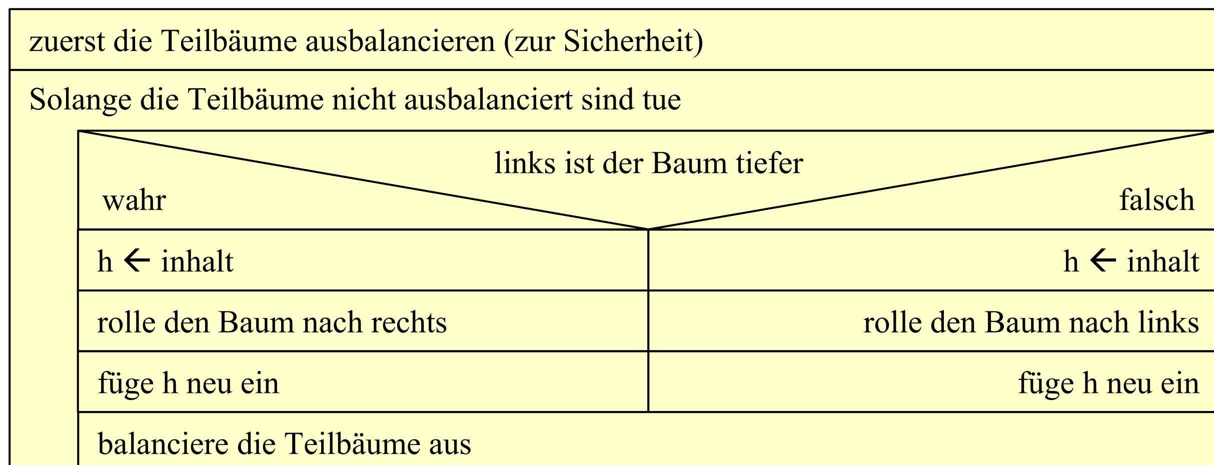
Inhalt zeigen ...

... und beide Teilbäume

Zuletzt wollen wir den Baum ausbalancieren, d. h. bei ungünstig verteilten Inhalten diese so verschieben, dass der Baum weniger Ebenen enthält als vorher. Wir wählen dafür die Methode *des Rollens*: Ist ein Teilbaum wesentlich tiefer als der andere, dann „rollen“ wir den Baum um ein Element auf die andere Seite. Das geschieht ähnlich wie beim Löschen, indem der jetzt „überstehende“ Teilbaum auf der anderen Seite angehängt wird. Dadurch kann der Baum wieder ziemlich aus dem Gleichgewicht geraten. Wir wenden deshalb das Ausbalancieren rekursiv auf die Teilbäume an und wiederholen das Verfahren, bis sich die Tiefen der Teilbäume nur noch um 1 unterscheiden.



balancieren:

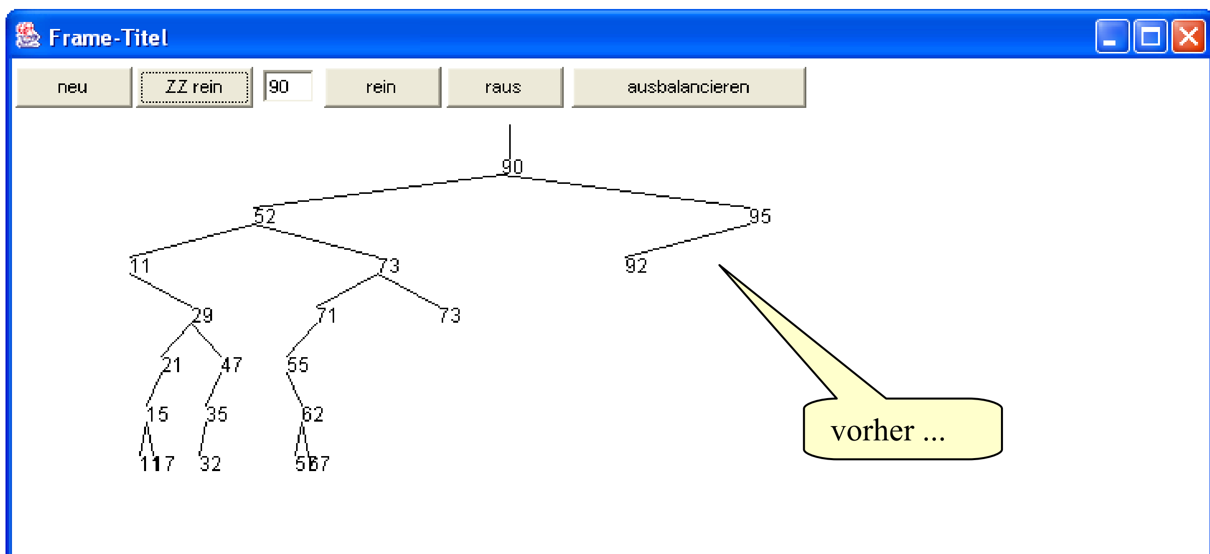


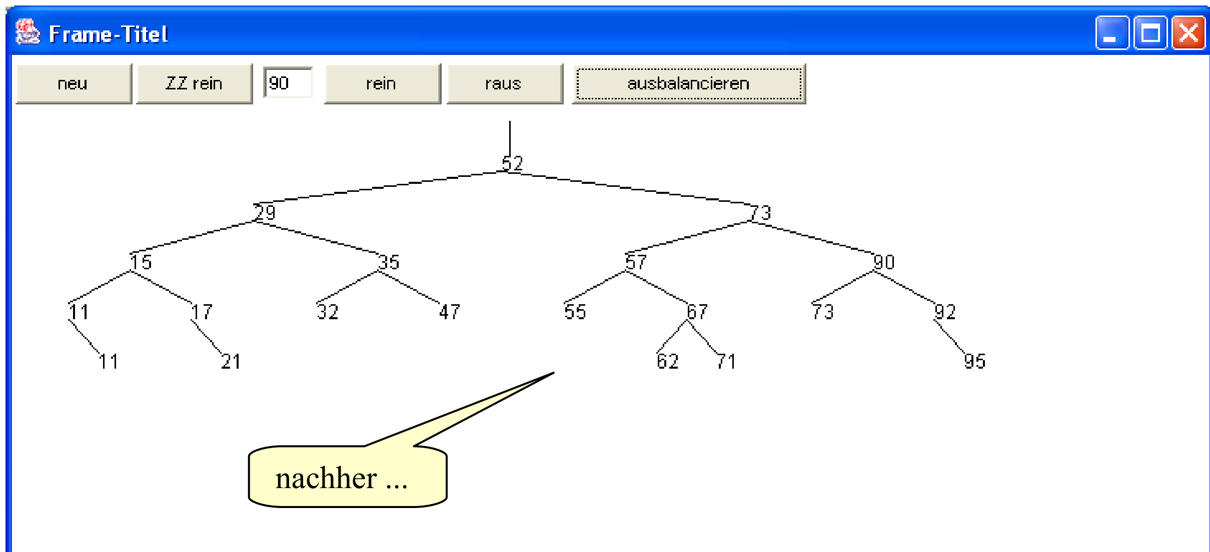

```
{
   int tlinks, trechts,h;

   if(links != null) links.balanciere();
   if(rechts != null) rechts.balanciere();

   if(links == null) tlinks = 0; else tlinks = links.tiefe();
   if(rechts == null) trechts = 0; else trechts = rechts.tiefe();

   while(Math.abs(tlinks-trechts) > 1)
   {
         if(tlinks > trechts)
         {
               h = inhalt;
               inhalt = links.inhalt;
               if(rechts == null) rechts = links.rechts;
               else rechts.linksuntenran(links.rechts);
               links = links.links;
               rein(h);
         }
         else
         {
               h = inhalt;
               inhalt = rechts.inhalt;
               if(links == null) links = rechts.links;
               else links.rechtsuntenran(rechts.links);
               rechts = rechts.rechts;
               rein(h);
         }
         if(links != null) links.balanciere();
         if(rechts != null) rechts.balanciere();

         if(links == null) tlinks = 0; else tlinks=links.tiefe();
         if(rechts == null) trechts = 0;
         else trechts=rechts.tiefe();
   }
}
```

Jetzt müssen wir nur noch die Tiefe eines Baums ermitteln. Das tun wir, indem wir das Maximum der Tiefen der Teilbäume zurückgeben.

```
        private int max(int a, int b)
        {
           if(a > b) return a; else return b;
        }
        private int tiefe()
        {
           int t1,t2;
           if(frei) return 0;
           else
           {
                 if(links == null) t1 = 1; else t1 = 1 + links.tiefe();
                 if(rechts == null) t2 = 1; else t2 = 1 + rechts.tiefe();
                 return max(t1,t2);
           }
        }
    }
```

Und jetzt können wir Bäume ausbalancieren.





Gut, nicht?

Die Nutzung des Baums in einem Anwendungsprogramm ist denkbar einfach. Zuerst müssen wir ein Exemplar des Baums erzeugen.

```
Baum b = new Baum();
```

Jetzt können wir Zahlen einfügen, indem z. B. in einem Event-Handler der Inhalt eines Textfeldes in eine Zahl verwandelt (hier: ohne Fehlerkontrolle!) und dem Baum übergeben wird. Danach wird die Anzeigefläche gelöscht und der Baum wird neu gezeichnet.

```
void b_rein_actionPerformed(ActionEvent e)
{
  String h = textField1.getText();
  int i = Integer.parseInt(h);
  b.rein(i);
  repaint();
}
```

Natürlich können wir auch ganz neu anfangen, wenn uns der Baum zu voll erscheint.

```
void b_neu_actionPerformed(ActionEvent e)
{
  b = new Baum();
  repaint();
}
```

Oder wir geben Zufallszahlen ein.

```
void b_zzrein_actionPerformed(ActionEvent e)
{
  b.rein((int)Math.round(Math.random()*100));
  repaint();
}
```

Das Löschen von Zahlen funktioniert ähnlich wie das Einfügen.

```
void b_raus_actionPerformed(ActionEvent e)
{
  String h = textField1.getText();
  int i = Integer.parseInt(h);
  b.raus(i);
  repaint();
}
```

Auch das Ausbalancieren geschieht mit einem Aufruf.

```
void b_balancieren_actionPerformed(ActionEvent e)
{
  b.balanciere();
  repaint();
}
```

Und wie zeichnet man?

Innerhalb der *paint()*-Methode löscht man zuerst den benutzten Bereich, zeichnet einen kleinen Strich als „Wurzel“ des Baums und überlässt den Rest der Arbeit der *zeigeBaum()*-Methode des Baums. Dieser müssen wir nur die Koordinaten des Anfangspunkts und den Grafikkontext als Parameter übergeben.

```
public void paint(Graphics g) {
  g.setColor(Color.white);
  g.fillRect(0,0,1000,1000);
  g.setColor(Color.black);
  g.drawLine(305,70,305,90);
  b.zeigeBaum(300,100,300,g);
}
```

8. Aufgaben

1. Sichern Sie die Zahleneingaben im Textfeld gegen Eingabefehler ab.

2. Ändern Sie den Baum so, dass
   a: Strings
   b: Datumswerte
   c: Verbunde aus Daten (Name, Vorname, Alter, ...)
   gespeichert werden.

3. Verallgemeinern Sie den Baum zu einer generischen Baumklasse. Wie können die Größenvergleiche dann ausgeführt werden?

4. Geben Sie die Inhalte eines Baum der Größe nach geordnet wieder aus.

5. Nutzen Sie das Ergebnis aus Aufgabe 4, um ein ungeordnetes Zahlenfeld in einen Baum einzulesen und geordnet wieder auszugeben.

6. B-Bäume enthalten pro Blatt Platz für mehrere Inhalte, zwischen denen Zeiger auf Blätter stehen, die die dazwischen liegenden Inhalte aufnehmen. Um Zugriffsoperationen zu optimieren, werden in den Blättern Plätze frei gehalten und die Inhalte bei Bedarf zwischen den Blättern einer Ebene verschoben. Erst wenn eine Ebene ganz gefüllt ist, wird eine neue aufgemacht, die dann anfangs halb leer ist. Entsprechend werden immer ganze Ebenen gelöscht, nachdem möglichst lange Inhalte innerhalb einer Ebene verschoben wurden.
   a: Informieren Sie sich im Internet oder der Literatur über B-Bäume.
   b: Implementieren Sie B-Bäume der Ordnung 2 für ganze Zahlen.
   c: Implementieren Sie eine generische B-Baum-Klasse dieser Ordnung. benutzen Sie dafür auch die Ergebnisse der Aufgabe 3.

7. Implementieren Sie einen „Stammbaum“, also die Generationenfolge einer Familie. Führen Sie insbesondere geeignete Möglichkeiten ein, in diesem Baum zu „blättern“.

8. Diskutieren Sie die Möglichkeiten, Stammbäume nicht als Klasse, sondern als HTML-Dokumentenfolge zu implementieren, in der eine Person jeweils auf einer Seite mit Bild, Text usw. dargestellt wird. Die Zusammenhänge werden durch Links auf andere Seiten implementiert (Eltern, Kinder, Geschwister, ...). Entsteht immer ein Baum?

9. Implementieren Sie einen generischen Suchbaum, dessen Blätter jeweils eine Frage als String aufnehmen. Wird diese mit „ja“ beantwortet, geht es links weiter, sonst rechts. Landet man am Ende eines Astes, dann wird vom „Experten“ eine neue Frage „erfragt“.
   a: Fügen Sie jetzt Inhalte für die Fehlersuche bei Homecomputerproblemen ein.
   b: Ebenso für die Bestimmung von Lebewesen („Hat es vier Beine?“).