Informatik B

Michael Karcher

23. Juli 2000

Allgemeines

0.1 Haftungsausschluss

Dieses Skript ist nach bestem Wissen und Gewissen aus meinem Gehirn an meine Hände weitergeleitet worden. Was um sonstwieviel Uhr mein bestes Wissen ist, will ich lieber nicht wissen, ebenso ist undefiniert, was meine Hände beim Tippen und der Computer beim TEXen des Dokumentes daraus macht. Wer einer Fehler findet, kann ihn mir (`Michael.Karcher@writeme.com`) ruhig zumailen, ich werde ihn wahrscheinlich korrigieren. Wer wegen eines Fehlers in diesem Skript durch die Klausur fällt, ist selber Schuld, da er besser in der Vorlesung hätte aufpassen können.

0.2 Markennamen

In diesem Skript vorkommende Namen können Markennamen sein. In diesem Fall gehören sie dem jeweiligen Inhaber und unterliegen dem Markenschutz.

0.3 Copyright

Jeder darf mit diesem Skript machen, was er will, solange er kein Geld damit verdient. Wenn jemand mich beteiligen will, sieht es anders aus. In diesem Fall bitte ich um eine Mail. `:-)`

Kapitel 1

Grundlagen

1.1 Einführung

1.1.1 Was ist Java?

Java ist eine Programmiersprache, die die Firma Sun entwickelt hat. Die wichtigsten Merkmale dieser Programmiersprache sind, dass Java

- objektorient
- interpretiert
- archtekturneutral

ist. Mit Objektorientierung ist eine Art der Programmierung gemeint, wie wir später noch sehen werden. Da Java interpretiert ist, wird ein Javaprogramm nicht in Maschinencode, der für jeden Prozessor anders ist, übersetzt wird, sondern in den sogenannten *Bytecode*, der von einem bestimmten Programm, nämlich der *JVM - Java Virtual Machine* interpretiert wird. Diese JVM gibt es für fast alle Computer, so dass ein kompiliertes Java-Programm auf fast allen Computern funktioniert, womit ich auch schon die Eigenschaft der Architekturneutralität beschrieben habe.

1.1.2 Wie mache ich etwas mit Java?

Wie mit jeder anderen Programmiersprache, arbeitet man in Java, in dem man mit einem Texteditor eine Programmdatei erstellt, die danach mit einem *Compiler* in eine ausführbarere Form übersetzt wird. Die klassischen Compilersprachen erzeugen hier eine Datei, die der Prozessor direkt versteht. Java erzeugt hier den oben schon erwähnten Bytecode, der nicht für irgend einen realen Prozessor gedacht ist, sondern für den virtuellen Prozessor in der JVM. Der dritte und letzte Schritt, wie man ein Javaprogramm zum laufen bekommt, ist es, die JVM zu starten.

Sowohl die JVM als auch der Java-Compiler sind Teile des Java Devopment Kits, das üblicherweise einfach JDK genannt wird. Das JDK existiert für alle wichtigen Plattformen (damit ist die Kombination aus Prozessor, umgebender Hardware und Betriebssystem, also beispielsweise „Linux auf einem Intel-Prozessor in einem IBM-PC“ gemeint), auf denen Java-Programme laufen sollen. Das

JDK selber enthält keine Oberfläche, in der man das Programm eintippt, eine Funktionstaste benutzt, und automatisch der Compiler gestartet wird, der einem dann direkt im Quelltext die Fehler zeigt, sondern besteht aus einigen Kommandozeilenprogrammen. Die beiden wichtigsten sind `javac` und `java`. `javac` ist der Java-Compiler, der Quellcode in Bytecode übersetzt. `java` startet die virtuelle Java-Maschine, und führt die `main`-Funktion einer angegebenen Klasse aus. Sehen wir uns mal ein Beispiel an:

Hello.java

```
public class Hello {
  public static void main(String args[]) {
    System.out.println("Hello, world!");
  }
}
```

Dieser Quelltext gehört in die Datei `Hello.java`, da alle Java-Quelltexte die Erweiterung `java` tragen müssen, und so heissen müssen wie die erste Klasse in ihnen.

In diesem Falle heißt die Klasse `Hello`, und ist für jeden zugänglich (`public`). Sie enthält genau eine Funktion, nämlich `main`. Der Name ist wichtig, denn die Funktion `main` ist die Funktion, mit der die JVM bei der Ausführung anfängt. Das einzige was getan wird, ist der Aufruf der Funktion `System.out.println`, die einen Text als Parameter erhält, und diesen ausgibt (normalerweise auf den Bildschirm). Jetzt muss man diesen Quelltext compilieren. In diesem Skript verwende ich Mitschnitte aus meinem zu Hause stehenden Linux-System, unter Windows kann es anders aussehen, wenn man eine Umgebung wie den JBuilder oder VisualJ++ verwendet. Ich schreibe generell kursiv, was der Benutzer eintippt, wogegen normale Schrift die Ausgabe des Systems darstellt.

```
mi@spartacus:~ > javac Hello.java
mi@spartacus:~ >
```

Das Ergebnis ist in diesem Fall eine Datei mit dem Namen `Hello.class`. Generell wird eine Klasse immer in eine Datei, deren Name der Klassenname ist, compiliert. Die Erweiterung des Namens lautet stehts `.class`. Jetzt geht es darum, die Klasse auszuführen, das heißt, wie schon zwei mal gesagt, die Funktion `main` dieser Klasse zu starten. Das tut man mit dem Programm `java`, das als Argument den Namen der Klasse erwartet:

```
mi@spartacus:~ > java Hello
Hello, world!
mi@spartacus:~ >
```

Programmierer sind natürlich selten perfekt, und daher kann es passieren, dass einmal ein Fehler in einem Programm ist. Ich ändere daher `Hello.java` so, dass das `S` in `System.out` klein geschrieben ist, und starte javac erneut:

```
mi@spartacus:~ > javac Hello.java
Hello.java:4: Undefined variable, class, or package name: system
                system.out.println("Hello, world!");
                ^
1 error
mi@spartacus:~ >
```

Hiermit teilt mir der Compiler mit, dass in der vierten Zeile der Datei `Hello.java` ein Fehler ist, und zwar, dass er `system` nicht kennt. Also, merkt es euch: *Java unterscheidet zwischen Groß- und Kleinschreibung!* und *Es ist häufig von Nutzen, die Fehlermeldung zu lesen, und nicht bis zur nächsten Übungsstunde zu warten, oder zu raten, was falsch sein könnte.* Als letztes fasst der Compiler zusammen, das genau ein Fehler aufgetreten ist. Die Datei `Hello.class` wird in diesem Fall vom Compiler nicht angefasst. Wenn sie vorher schon existierte, bleibt sie bestehen, wenn sie noch nicht existierte, erstellt der Compiler sie nicht.

1.2 Programmieren in Java

1.2.1 Kommentare

In Java können überall, außer mitten in Zeichenketten, im Quelltext Kommentare stehen. Es gibt zwei verschiedene Varianten, Kommentare in den Quelltext einzubinden. Zum einen die Möglichkeit, alles bis zum Zeilenende als Kommentar zu erklären, was nützlich ist, wenn man zu einer Programmzeile am rechten Rand eine Anmerkung machen möchte, zum anderen kann man Kommentarblöcke verwenden, die mit `/*` anfangen und `*/` aufhören.

Toast.java

```
/* Dieses ist ein Beispiel: Ein Java-Programm mit Kommentaren
   Kommentare, die über mehrere Zeilen gehen, werden, wie man
   sieht mit Slash-Stern eingeleitet und enden mit Stern-Slash
   */

/* Achtung! Diese Kommentarart kann man nicht verschachteln! Ich
   kann hier jetzt /* /* /* /* schreiben, und der Kommentar ist
   trotzdem hier zuende: */

public class Toast {       // Immer nur "Test" ist langweilig!
  public /*geht auch hier*/ static void main(String argv[]) {
    System.out.println(argv[0]);  // Gib das erste Argument aus
  }
  // Wie man sieht, ist ab dem // bis zum Zeilenende Kommentar
}
```

1.2.2 Das Fakultätsprogramm

Factorial.java

```
1  /**
2   * This program computes the factorial of a number
3   */
4  public class Factorial {                    // Define a class
5    public static void main(String[] args) { // The program starts here
6      int input = Integer.parseInt(args[0]); // Get the user's input
7      double result = factorial(input);      // Compute the factorial
8      System.out.println(result);            // Print out the result
9    }                                        // The main() method ends here
10
11   public static double factorial(int x) {  // This method computes x!
12     if (x < 0)                             // Check for bad input
13       return 0.0;                          //   if bad, return 0
14     double fact = 1.0;                     // Begin with an initial value
```

```
15      while(x > 1) {                       // Loop until x equals 1
16        fact = fact * x;                   //   multiply by x each time     \
17        x = x - 1;                         //   and then decrement x        \
18      }                                    // Jump back to the start of loop\
19      return fact;                         // Return the result            \
20    }                                      // factorial() ends here        \
21  }                                        // The class ends here
```

Dieses Programm berechnet die Fakultät des Argumentes, eine genauere Erläuterung folgt: In Zeile 1 sieht man einen Kommentar, der mit `/**` anfängt. Für den Javacompilere `javac` ist das nichts besonderes, das erste Zeichen des Kommentars ist halt ein Stern. Das Programm `javadoc`, was Dokumentation zu Klassen erstellt, verwendet diese Kommentare um sie in die Dokumentation zu übernehmen. Wie `javadoc` genau arbeitet, ist uns erst einmal egal, solange unsere Projekte noch klein und übersichtlich sind.

Zur Übersichtlichkeit eines Programmes tragen zwei Punkte entscheidend bei:

Kommentare Zum eine eine vernünftige Kommentierung. Die Zeile `x=3;` mit dem Kommentar „Hier wird x auf 3 gesetzt" zu versehen ist *nicht* vernünftig. Ein Kommentar: „3, damit x ungerade ist", falls das eine Rolle spielt, kann dagegen sehr wohl vernünftig sein.

Formatierung Es gibt etwa doppelt so viele Ansichten, wie ein Programm am besten formatiert ist, wie es Personen gibt, die den Quelltext zu diesem Programm je lesen wollen. Daher schreibe ich in diesem Skript nur allgemeine Richtlinien zur Formatierung. Ob man jetzt lieber zwei oder drei Zeichen einrückt ist recht egal.

1.2.3 Klassendefinitionen

Wir sind den Klassendefinitionen schon häufiger begegnet, da in Java alles in Klassen liegt. Genauer erklärt haben wir diese Definition aber noch nicht. Ich betrachte die Bestandteile einer Klassendefinition:

Modifier Modifier beschreiben die Art der Klasse genauer. Was es an Modifiern gibt, werden wir später noch lernen. Man muß gar keine Modifier angeben, es dürfen aber beliebig viele durch Zwischenraum[1] getrennte Modifier hintereinander stehen. In diesem Falle ist der einzige Modifier `public`, der besagt, dass jeder auf die Klasse zugreifen darf.

Typ Bei Klassendefinitionen lautet der Typ immer `class`.

Name Wenn man etwas definiert, gibt man dem Definierten einen Namen. Dieser Name steht direkt hinter dem Typ.

Geltungsbereich Der Geltungsbereich ist ein Block, der durch geschweifte Klammern begonnen und beendet wird. Alles, was in diesem Block definiert wird, gehört zu der Klasse, die durch die Klassendefinition definiert wird. Dieser Block wird auch einfach *Bereich* der Definition oder auf Englisch „scope" genannt.

[1] So genannter „white space", das sind Leerzeichen, Tabulatoren, Zeilenumbrüche und auf manchen Systemen auch noch weitere Zeichen

Die innerhalb einer Klasse definierten Objekte, in diesem Fall sind es zwei Methoden (man sagt auch Elementfunktionen), `main` und `factorial`, heißen Mitglieder („member“) der Klasse. Schauen wir uns die Methodendefinitionen einmal genauer an:

1.2.4 Methodendefinitionen

Eine Methodendefinition ist vom Prinzip her einer Klassendefinition sehr ähnlich. Ich erläutere das Prinzip erst einmal an der Methode `factorial`:

Modifier Wie schon bei den Klassen, kann man mit den Modifiern die Feinheiten bestimmen. `public` heißt hier wieder, das von überall diese Methode aufgerufen werden darf, und `static` wird später noch erläutert.

Typ Hier steht der Typ, den die Funktion zurückgibt. Dass es sich um eine Methodendefinition handelt, sieht der Compiler an der Parameterliste. `factorial` liefert eine Gleitkommazahl mit doppelter Genauigkeit zurück, der Name dieses Typs ist `double`.

Name Die Methode heißt `factorial`. Das ist nun mal so, und bedarf keiner Erklärung.

Parameterliste Methoden, die eine Funktion berechnen, brauchen häufig Ausgangsdaten. So auch die Fakultätsfunktion, denn irgendwoher muß sie ja wissen, wovon die Fakultät berechnet werden soll. Dazu übergibt man der Funktion Parameter (auch Argumente genannt). Die Parameter werden innerhalb von runden Klammern genauso wie Variablen definiert, daher erläutere ich das hier noch nicht.

Geltungsbereich Alles, in dem Block nach der Parameterliste (man erinnere sich, ein Block wird durch geschweifte Klammern begrenzt), sind Definitionen oder Befehle, die nur zu dieser Methode gehören.

1.2.5 Variablendefinitionen

Das Schema einer Definition erneut hinzuschreiben ist schon fast etwas langweilig, aber trotzdem. Als Beispiel nehme ich Zeile 14:

```
double fact = 1.0;
```

Modifier Nun, wer hätte es geahnt, auch bei Variablen kann man Modifier angeben, sie bestimmen wie immer die eine oder andere Eigenschaft der Variable. In diesem Fall brauchen wir keine, denn die Methode, in der die Variable definiert wird, darf natürlich drauf zugreifen, und außerhalb dieser Methode existiert die Variable sowieso nicht.

Typ Gibt, wie man intuitiv erwartet hat, den Typ der Daten an, die man in der Variable speichern kann. In diesem Fall ist es wieder `double`.

Name Man will doch die Variable auch ansprechen können, oder? Genau dazu braucht sie den Namen.

Initialisierung Bei einer Variablendefinition kann nach dem Namen noch ein `=` folgen, und dahinter ein Ausdruck, der angibt, mit welchem Wert die Variable belegt sein soll. Wenn die Initialisierung wegfällt, wird die Variable erst einmal auf einem undefiniertem Wert belassen. Wenn der Java-Compiler sich nicht sicher ist, ob eine Variable initialisiert wird, bevor sie irgendwie verwendet wird, gibt es eine Fehlermeldung, solange es sich nicht um statische Variablen handelt. Diese erhalten einen Standardwert.

1.2.6 Datentypen

In Java haben alle Variablen einen festen Typ. Es können nur Daten dieses Typs in der Variable gespeichert werden. Es gibt sogenannte primitive und komplexe Typen, zuerst beschreibe ich die wichtigsten primitiven Typen.

Typ	Wofür
`boolean`	Wahrheitswert: `true` oder `false`
`int`	Ganze Zahlen
`float`	Kleine und ungenaue Gleitkommazahlen
`double`	Gleitkommazahlen
`char`	ein Zeichen aus dem 16-Bit-Zeichensatz

Um Variablen zu initialisieren, braucht man einen Ausdruck, der den richtigen Typ als Ergebnis hat, das einfachste sind Konstanten. Je nach Typ sehen Konstanten verschieden aus:

- `boolean`-Konstanten lauten entweder `true` oder `false`
- `int`-Konstanten bestehen nur aus Ziffern und einem `-` davor bei negativen Zahlen
- `double`-Konstanten bestehen aus einer Zeichenfolge aus Ziffern und einem Dezimalpunkt, wie `1.0` oder `0.003`, aber auch `1.` oder `.5`, oder aus einer Ziffernfolge, die einen Dezimalpunkt enthalten kann, auf die ohne Zwischenraum ein `e` und eine weitere Ziffernfolge folgt. Dabei steht `1.23e23` für $1.23 \cdot 10^{23}$.
- `float`-Konstanten sind `double`-Konstanten, an die ein `f` angehängt wird, wie in `float var = 1.2f`. Beim Versuch, einer `float`-Variablen einen `double`-Wert zuzuweisen, gibt es einen Compilerfehler (den man durch eine explizite Typumwandlung auch beseitigen kann, kommt noch...)
- `char`-Konstanten stehen zwischen einfachen Anführungszeichen, wie in `'X'`. Entweder handelt es sich um einen einzelnen Buchstaben, oder um eine sogenannte Escape-Sequenz, die mit einem Backslash `\` anfängt. Falls man den Backslash selber braucht, nimmt man `'\\'`. Weitere Escapesequenzen sind `'\n'`, was den Computer oder Drucker anweist, in eine neue Zeile zu springen und `'\uXXXX'`, wobei XXXX ein vierstelliger Code im Hexadezimalsystem ist. Damit gibt man die Zeichennummer im Unicodezeichensatz an. Beispielsweise steht `'\x03c0'` für das π.

Man kann die meisten Unicode-Zeichen auch in Variablennamen benutzen, also zum Beispiel

```
final double \u03c0=3.141592
```

wobei `final` ein Modifier ist, der sagt, dass der Wert endgültig ist, und im Programm nicht modifiziert werden kann – sprich, der aus einer Variablen eine Konstante macht.

Komplexe Datentypen funktionieren im Prinzip genauso:

```
Termin t;          // Typ oder Klasse Termin
                   // muß definiert sein
String[] args;     // Siehe auch Parameter
                   // von main
```

Es ist eine Konvention (aber im Gegensatz zu Haskell ist es nicht Pflicht), dass komplexe Datentypen groß geschrieben werden. Der Typ `String` ist in der API vordefiniert. API steht für Application Program Interface, also Schnittstelle für Anwendungsprogramme. Sie ist die Sammlung von Klassen und Typen, die in Java direkt eingebaut sind. Die eckigen Klammern bedeuten, dass es sich um ein Array handelt, also nicht ein String, sondern viele Strings (so wie Listen in Haskell) in `args` gespeichert werden.

1.2.7 Methoden in der API

Es ist an dieser Stelle völlig fehl am Platze, sämtliche Methoden sämtlicher Klassen der API aufzulisten. Es ist aber wichtig, dass man weiß, dass es solche Auflistungen (genannt API-Referenz) gibt.[2] In einer solchen Referenz findet man die Erklärung zu der Programmzeile

```
int input = Integer.parseInt(args[0]);
```

Es gibt nämlich die vordefinierte Klasse `Integer`, die eine Methode `parseInt` enthält. Sie nimmt einen String als Parameter (mit args[0] greift man auf den ersten Kommandozeilenparameter zu), und konvertiert ihn in eine Ganzzahl, sofern das möglich ist, ansonsten wird ein Programmausnahme (Exception) ausgelöst.

Die Funktion `System.out.println` ist ebenfalls ein Teil der API, sie nimmt einen String, und schreibt ihn auf den Bildschirm. Man kann Strings mit `+` aneinanderhängen. Alle primitiven Typen (wenigstens die, die oben stehen) werden von Java automatisch in einen String konvertiert, falls man sie in einem Zusammenhang verwendert, der einen String erfordert. Umgekehrt ist das nicht der Fall.

Man kann die Ausgabe des Fakultätsprogramms dadurch verschönern, dass man die Zeile 8 in

```
8     System.out.println(input + "! = " + result);
```

ändert.

1.2.8 Arithmetik

In Java kann man ganz normal mit den vier Grundrechenarten (`+`,`-`,`*`,`/`) rechnen, allerdings muss man mit dem Additions- und dem Divisionsoperator vorsichtig

[2] `http://java.sun.com/products/jdk/1.2/docs/api/index.html`

sein, da diese beiden Operatoren sich je nach beteiligten Typen verschieden verhalten. Das Pluszeichen dient nämlich außerdem zur Stringverkettung, die Division wird nur, wenn Fließkommazahlen beteiligt sind, als Fließkommadivision ausgeführt, ansonsten wird gegen 0 gerundet (sprich: abgehackt). Natürlich achtet Java auf die mathematischen Regeln, was zuerst aufgeführt werden soll, also wird Punktrechnung vor Strichrechnug gemacht, wenn keine Klammern gesetzt sind. Was dann noch nicht feststeht wird von links nach rechts geklammert (also ist der Ausdruck `3-4+5`, wie es üblich ist `(3-4)+5`).

Das Ganze ist vor allem deshalb wichtig, weil `+` nur für zwei Strings oder zwei numerische Typen definiert ist. Dass `"Die Antwort ist "+42` zum trotzdem funktiniert, liegt daran, dass, wie oben gesagt, Zahlen automatisch in Strings konvertiert werden, wenn sie in einem Zusammenhang auftauchen, wo Strings benötigt werden. So ergibt `"Test"+3+4` aufgrund der automatischen Klammerung von links nach rechts `("Test"+3)+4`, wobei dann die `3` in einen String konvertiert wird, so dass der Zwischenschritt `"Test3"+4` lautet, und das Ergebnis `"Test34"` ist. Schreibt man `"Test"+(3+4)` wird zuerst die Klammer ausgewertet, also kommt der Computer auf `"Test"+7`, und das Ergebnis wird `"Test7"` sein. Wenn man die Zahlen vor dem String hat, also `3+4+"Test"` schreibt, wird wieder von links nach rechts geklammert, so dass sich `(3+4)+"Test"` ergibt, was zu `"7Test"` ausgewertet wird.

Bei der Division verhält es sich ähnlich: Sobald einer der beiden Operanden eine Fließkommazahl ist, wird der andere ebenfalls in eine Fließkommazahl konvertiert, und eine Fließkommadivision ausgeführt. Sind beide Operanden Ganzzahlen oder Ganzzahlvariablen, erhält man immer eine Ganzzahl als Ergebnis.

`Division.java`

```
/**
 * Programm zur Demonstration der Ganzzahldivision
 */

public class Division {
  public static void main(String[] args)
  {
    int x = 20;
    int y = 12;
    double z = x / y;      // z ist jetzt 1
    System.out.println(z);
  }
}
```

Dieses Testprogramm gibt `1.0` auf den Bildschirm aus. Es wirkt so, als ob die Nachkommastellen berechnet worden wären. Das ist allerdings nicht der Fall. Die Division `x/y` wird als Ganzzahl berechnet, wobei `1` herauskommt, das *Ergebnis* wird dann in eine Fließkommazahl konvertiert.

Außerdem gibt es in Java auch noch den Modulo-Operator, der den Rest bei der Division berechnet. Er ist nicht nur auf Ganzzahlen, sondern auch auf Fließkommawerte definiert. Allerdings muß man dabei vorsichtig sein, da bei der Modulorechnung manchmal Rundungsfehler auftreten, die man sonst nicht merken würde. Ich mache ein Beispiel:

`Modulo.java`

```
/**
```

```
 * Programm zur Verdeutlichung der
 * Gefahr des Modulos auf Fließkommazahlen
 */

public class Modulo {
  public static void main(String[] args) {
    double eins = 1;
    System.out.println(eins%(1.0/3));
    System.out.println(eins%(1.0/5));
    System.out.println(eins%(1.0/7));
  }
}
```

Ausgabe des Programms

```
mi@spartacus:~/javascript > java Modulo
5.551115123125783E-17
0.19999999999999996
5.551115123125783E-17
mi@spartacus:~/javascript >
```

Man würde zuerst erwarten, dass die Ausgabe des Programmes in allen drei Fällen null wäre. Wie man sieht, ist sie es in keinem, was daran liegt, daß die Kehrwehrte nicht exakt dargestellt werden können. In diesem Programm muss man die Kehrwehrte mit `1.0/3`, `1.0/5`, `1.0/7` berechnen, da sonst, wie oben schon erläutert, das Ergebnis, was bekanntlich kleiner eins ist, einfach auf Null abgehackt würde. Im ersten und dritten Fall ist das Ergebnis wenigstens fast null, so dass man bei einem Vergleich gegenüber Null mit etwas Toleranz die Nullen noch erkannt. Im zweiten Fall kommt aber absolut nicht null heraus, was daran liegt, dass `1.0/5` etwas zu groß ist, und daher nicht ganz fünf mal in eins reingeht, so dass noch etwas weniger als ein Fünftel übrig bleibt. In den anderen beiden Fällen war der Kehrwert etwas zu klein, so dass nur ein winziger Rest von der Eins bei der Division übrig geblieben ist.

Als Besonderheit gibt es in Java den aus `C` und `C++` übernommenen Inkrement- und Dekrementoperator. Der Inkrementoperator lautet `++`, und kann vor oder hinter einen Variablennamen geschrieben werden. Sowohl `++i;` wie `i++;` sind Anweisungen, die die Variable `i` um eins erhöhen. Allerdings ist noch mehr möglich, denn der Ausdruck `i++` hat einen Wert, den man verwenden kann, und zwar den alten Wert von `i` (Merke: wenn das `++` *nach* i steht, wird erst der Wert genommen, und da*nach* wird i erhöht). Im Gegensatz dazu hat `++i` den neuen Wert von i. Das folgende Programm gibt also `i=3,j=1,k=3` aus:

Inkrement.java

```
public class Inkrement {
  public static void main(String[] args)
  {
    int i=1;
    int j=i++; // Erst i nehmen, dann erhöhen
    int k=++i; // umgekehrt
    System.out.println( "i=" + i +
                        ",j=" + j +
                        ",k=" + k);
```

```
  }
}
```

Der Dekrementoperator `--` funktioniert genauso, nur das er die Variable um eins erniedrigt anstatt zu erhöhen.

1.2.9 Kontrollstrukturen

Bedingte Ausführung mit `if`

```
if(Bedingung) {
  Block A
}
else {
  Block B
}
```

Die *Bedingung* wird ausgewertet, es muss dabei ein Boolscher Ausdruck herauskommen (Moment, die Vergleichsoperatoren hat der Prof noch gar nicht erläutert!). Wenn die Bedingung wahr ist, wird *Block A* ausgeführt, ansonsten *Block B*. Es ist nicht nötig, einen `else`-Block anzugeben, alles nach der ersten geschweiften Klammer zu kann entfallen, dann wird *Block A* ausgelassen, falls *Bedingung* falsch ist, ansonsten wird alles audgeführt.

Schleifen mit `while`

```
while(Bedingung) {
  Block
}
```

Solange die *Bedingung* zutrifft, wird der Block *Block* ausgeführt. Dabei wird die Bedingung bereits vor dem ersten Durchlauf geprüft. Ist sie von Anfang an falsch, wird die Schleife gar nicht durchlaufen. Eine `while`-Schleife wurde in der Funktion `factorial` verwendet. Ich drucke hier den Schleifeninhalt noch einmal ab:

Factorial.java

```
14    double fact = 1.0;                      // Begin with an initial value
15    while(x > 1) {                          // Loop until x equals 1
16      fact = fact * x;                      //   multiply by x each time     \
17      x = x - 1;                            //   and then decrement x        \
18    }                                       // Jump back to the start of loop\
19    return fact;                            // Return the result             \
```

Zuerst wird `result` auf eins gesetzt. Danach wird `result` in der Schleife mit allen ganzen Zahlen zwischen zwei und `x` multipliziert, und zwar mit `x` als erstes, und dann abwärts. In jedem Schleifendurchlauf (das heißt, bei jeder Ausführung des Blocks) findet eine Multiplikation statt, und x wird erniedrigt. Wenn x nicht mehr größer als eins ist, ist die Funktion fertig. Wenn x von Anfang an 1 oder 0 war, wird die Schleife gar nicht erst ausgeführt, es findet also gar keine Multiplikation statt.

Schleifen mit `do-while`

```
do {
  Block
} while(Bedingung);
```

Vom Prinzip her funktioniert die `do-while`-Schleife genauso wie die `while`-Schleife. Allerdings wird die Schleife mindestens einmal durchlaufen. Eine sinnvolle Anwendung der `do-while`-Schleife ist zum Beispiel ein interaktives Programm, was am Anfang der Schleife fragt, was der Benutzer tun möchte, dann die gewünschte Aktivität ausführt, und am Ende prüft, ob der Benutzer den Befehl zum Beenden gegeben hat. Vor dem ersten Durchlauf kann er ihn noch nicht gegeben haben, so dass es nicht sinnvoll wäre, denn der Benutzer wird ja erst in der Schleife gefragt.

Schleifen mit `for`

```
for(Vorher;Bedingung;Schritt) {
  Block
} while();
```

Die `for`-Schleife ist eine komplexe Formulierung, die einem erlaubt, alles was zu der Schleife gehört innerhalb der Schleifenanweisung zu schreiben. Ich gebe erst einmal ein typisches Beispiel, bevor ich es genauer erkläre:

ForSchleife.java

```
/**
 * Beispiel für eine For-Schleife
 */

public class ForSchleife {
  public static void main(String[] args)
  {
    // Zähle von 1 bis 10
    for(int x = 1;x <= 10;x++)
      System.out.println(x);
  }
}
```

Dieses Programm gibt die Zahlen von eins bis zehn hintereinander in einzelnen Zeilen aus. Und zwar wird zuerst der *Vorher*-Block ausgeführt, in dem die Variable `x` definiert und auf eins gesetzt wird. Danach wird die *Bedingung* geprüft, wenn sie wahr ist, wird der *Block* durchlaufen. (der auch eine einfache, mit Semikolon abgeschlossene Anweisung sein kann, dann entfallen die geschweiften Blockklammern) Hier prüfen wir, ob `x` immernoch kleiner als zehn ist, was natürlich der Fall ist. In der Schleife wird der aktuelle Wert von `x` ausgegeben. Wenn ein Durchlauf fertig ist, wird *Schritt* ausgeführt, um den nächsten Durchlauf vorzubereiten. Danach wird wieder die *Bedingung* geprüft und falls sie immernoch wahr ist, der *Block* und der *Schritt* erneut ausgeführt. Hier wird im *Schritt* `x` erhöht, damit im nächsten Durchlauf die nächste Zahl in `x` steht.

1.2.10 Ein komplexeres Fakultätsprogramm

Factorial3.java

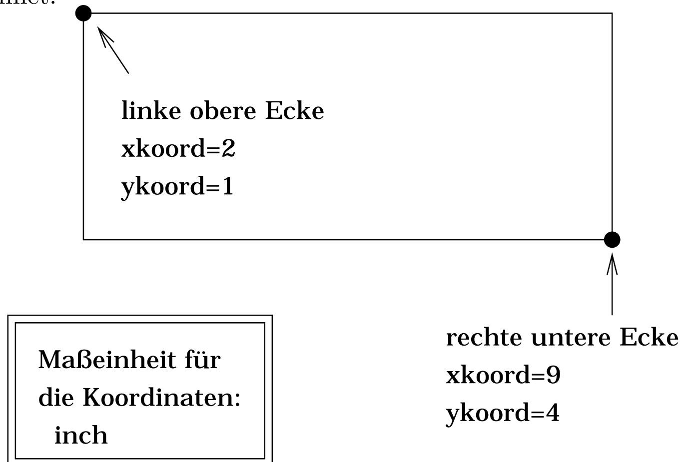
```java
import java.io.*;  // Import all classes in java.io package.  Saves typing.

/**
 * This class computes factorials and caches the results in a table for reuse.
 * 12! is as high as we can go using the int data type
 * The example also shows how to handle exceptions (e.g. bad input)
 **/

public class Factorial3 {
  // Create an array to cache values 0! through 12!.
  static int[] table = new int[13];
  // A "static initializer": initialize the first value in the array
  static { table[0] = 1; }  // factorial of 0 is 1.
  // Remember the highest initialized value in the array
  static int last = 0;

  public static int factorial(int x) {
    // Check if x is too big or too small.  Throw an exception if so.
    if (x >= table.length)   // ".length" returns length of any array
      throw new IllegalArgumentException("Overflow; x is too large.");
    if (x < 0) throw new IllegalArgumentException("x must be non-negative.");

    // Compute and cache any values that are not yet cached.
    while(last < x) {
      table[last + 1] = table[last] * (last + 1);
      last++;
    }
    // Now return the cached factorial of x.
    return table[x];
  }

  public static void main(String[] args) {
    // This is how we set things up to read lines of text from the user.
    BufferedReader inbuff = new BufferedReader(new InputStreamReader(System.in));
    int x=1;
    String line=null;
    // Loop forever
    for(;;) {
      // Display a prompt to the user
      System.out.print("Neues Argument: ");
      // Read a line from the user
      try{ line = inbuff.readLine(); }
      catch(Exception e) {
      System.out.println("IOProblem: " + e.getMessage());
      }
      // If we reach the end-of-file, or if the user types "quit", then quit
      if ((line == null) || line.equals("quit")) break;
      // Try to parse the user's input, and compute and print the factorial
      try {
        x = Integer.parseInt(line);
        System.out.println(x + "! = " + factorial(x));
      }
      // If anything goes wrong in parsing, display an error message
      catch(NumberFormatException e) {
        System.out.println("The argument must be an integer");
        }
      // The argument is < 0 or > length.  Thrown by factorial()
      catch (IllegalArgumentException e) {
      // Display the message sent by the factorial() method:
      System.out.println("Bad argument: " + e.getMessage());
```

```
61          }
62        }
63      }
64    }
```

Nachdem das erste Programm einigermaßen verständlich ist, schauen wir uns das nächste an. Auch hier werden die einzelnen Sprachelemente besprochen, die verwendet werden. Das erste ist das `import` am Anfang, was dem Java-Compiler sagt, dass sämtliche Klassen aus der API, deren Name mit `java.io` anfängt, in diesem Programm verwendet werden können. Diese Anweisung ist deshalb nötig, da es ewig dauern würde, bei jedem Compilerstart sämtliche Klassen der API (die im JDK 1.2 etwa 8 Megabyte im Bytecode sind) zu laden.

1.2.11 Klassen und Instanzen

Bis jetzt haben wir Klassen nur dazu gebraucht, um Funktionen darin unterzubringen. Klassen können aber außer Funktionen auch noch Daten enthalten, und außerdem kann man Objekte erzeugen, deren Typ eine Klasse ist. Machen wir mal ein einfaches Beispiel: Eine Klasse, die einen Punkt beschreibt. Jeder Punkt ist ein Objekt, dessen Typ diese Klasse ist. Man nennt Objekte, deren Typ eine Klasse, ist Instanzen der Klasse. Jede Instanz der Klasse Punkt hat ihre eigenen Koordinaten, allerdings hat die Klasse selbst die Eigenschaft „Maßeinheit", da es ein ziemliches Durcheinander geben würde, wenn man gleichzeitig mit Punkten, deren Koordinaten in Inch, in Zentimetern oder in Pixeln angegeben ist, rechnet.



In Java gehören Eigenschaften normalerweise jeder Instanz einer Klasse. Man kann allerdings mit dem Modifier `static` erklären, dass die modifizierte Eigenschaft oder Funktion der Klasse nichts mit den einzelnen Instanzen am Hut hat. Eine als `static` deklarierte Funktion kann man auch aufrufen, ohne eine Instanz der Klasse zu kennen. So sollte man zum Beispiel eine Methode, die fragt, ob man lieber in Inch oder in Zentimetern arbeitet, ausführen, bevor der erste Punkt erzeugt wird, also bevor es überhaupt eine Instanz der Klasse gibt. Natürlich können `static`-Funktionen nur auf `static`-Variablen der Klasse zugreifen, da sie unabhängig von Instanzen arbeiten, und jede Instanz ihre eigene Kopie der nichtstatischen Eigenschaften hat.

Das erklärt jetzt, wieso wir `main` immer als `static` deklariert haben: Wenn die JVM startet, gibt es noch gar keine Objekte. Also gibt es auch keine Instanz der Hauptklasse unseres Programms. Und solange es keine Instanz gibt, kann keine Funktion aufgerufen werden, die nicht statisch ist.

1.2.12 Arrays

Arrays werden auch Felder genannt, und sind eine nummerierte Ansammlung von Objekten des gleichen Typs, der sowohl primitiv, als auch eine Klasse sein kann. Da wir über Klassen und Objekte noch nicht so viel gesprochen haben, mache ich die Beispiele hauptsächlich mit primitiven Typen.

Arrays werden deklariert, indem man hinter den Typnamen ein Paar eckiger Klammern setzt (wie in `String[] args` als Parameter von `main`). Ein Array, in dem man 13 ganze Zahlen speichern kann, deklariert man als

```
int table[] = new int[13];
```

Mit dem `new int[13]` wird Speicherplatz für 13 `int`s besorgt, auf den man über `table` zugreifen kann. Die einzelnen Elemente heißen `table[0]` bis `table[12]`, da in Java Arrays generell mit 0 angefangen werden.

Man kann Arrays auch gleich bei der Definition initialisieren, wie in der folgenden Zeile:

```
int[] zPot = {1,2,4,8,16,32,64};
```

Hier wird ein Array mit dem Namen `zPot` angelegt, was die ersten sieben Potenzen von zwei, angefangen bei 2^0 speichert. Anstatt das Array direkt zu initialisieren, kann man auch Code schreiben, der die Zweierpotenzen berechnet:

```
int[] zPot = new int[7];
zPot[0] = 1;              // Startwert
for(int i = 1;i < zPot.length;i++) {
  zPot[i] = zPot[i-1]*2;  // nimm die letzte
                          // Potenz mit 2 mal
}
```

`length` ist ein reserviertes Wort in Java, was einem die Anzahl der Elemente des Arrays zurückgibt, auf das es angewandt wird. In diesem Fall ist `zPot.length` also 7. Falls man beschließt, die ersten neun Zweierpotenzen zu benötigen, muss man nur die erste Zeile ändern.

Man kann auch mehrdimensionale Arrays definieren, ich gebe hierzu ein Beispiel mit einer Differenztabelle an. (In der Vorlesung hatten wir eine Multiplikationstabelle, an der war nicht ersichtlich, welcher Index zuerst kommt.)

Array.java

```
/**
 * Programm zur Demonstration von
 * zweidimensionalen Arrays
 */

public class Array {
  public static void main(String[] args)
  {
```

```
    // diffTab[a][b] soll a-b sein
    int diffTab[][] = {{ 0,-1,-2,-3,-4},
                       { 1, 0,-1,-2,-3},
                       { 2, 1, 0,-1,-2},
                       { 3, 2, 1, 0,-1},
                       { 4, 3, 2, 1, 0}};
    System.out.println("0 - 2 = " +
                          diffTab[0][2]);
    System.out.println("3 - 1 = " +
                          diffTab[3][1]);
  }
}
```

Diese Programm gibt die Differenzen unter der Verwendung der Tabelle aus. Für die Reihenfolge der Indezes kann man sich ein zweidimensionales Array als Array aus eindimensionalen Arrays denken (das ist es ja auch!), und dann wird klar, dass der erste Index auswählt, welche der fünf Zeilen gewählt werden muss, und der zweite die Position darin angibt.

Arrays sind Referenzen

Das bedeutet, das `int bla[];` keinen Platz für ein Array belegt, weshalb wir oben auch `new` benutzt haben, sondern nur einen Verweis auf ein Array aufnehmen kann. In Java sind Arrays generell Referenzen. Welche Konsequenz das hat, veranschauliche ich mit einem Beispiel:

Referenz.java

```
1  /**
2   * Programm zur Verdeutlichung, dass Arrays
3   * Referenzen sind
4   */
5
6  public class Referenz {
7    public static void main(String[] args)
8    {
9      char[] original = {'T','a','n','n','e'};
10     char[] referenz = original;
11     System.out.println("original="+original);
12     referenz[1] = 'o';
13     System.out.println("original="+original);
14     char[] dolly = (char[])original.clone();
15     dolly[1] = 'a';
16     System.out.println("original="+original);
17     System.out.println("dolly="+dolly);
18   }
19 }
```

Ausgabe des Programms

```
original=Tanne
original=Tonne
original=Tonne
dolly=Tanne
```

In Zeile 12 des Programmes wird der Wert von `referenz` verändert. Da `referenz` und `original` aber Referenzen auf das *gleiche* Array im Speicher sind, wird der Wert von `original` ebenfalls verändert. Mit `clone` erzeugt man dagegen eine Kopie eines Objektes (die genauere Bedeutung der einzelnen Elemente aus Zeile 14 werden wir noch besprechen), die unabhängig vom Original sind. Daher ändert sich trotz der Zuweisung in Zeile 15 der Wert von `original` nicht mehr.

Wichtig ist es auch beim Vergleich von Arrays, denn anstatt den Inhalt zu vergleichen, macht die Befehlszeile `if ( arraya == arrayb )` einen Vergleich, ob `arraya` und `arrayb` das gleiche Objekt referenzieren. Wenn `arrayb` ein Klon von `arraya` ist, sind `arraya` und `arrayb` in diesem Sinne also ungleich. Um zu vergleichen, ob zwei Objekte gleich sind, verwendet man die Methode equals, die jedes Objekt hat, und gibt ihr das andere Objekt als Argument: `if ( arraya.equals ( arrayb ))`

1.2.13 Exceptions

Eine Exception ist ein Signal, das auf eine Ausnahmesituation oder einen Fehler hinweist. Exceptions werden an einer Stelle ausgelöst, und können an einer anderen Stelle abgefangen werden, werden sie nicht abgefangen, führt das zum Programmabbruch. Um eine Exception auszulösen, muss man erst einmal ein Objekt erzeugen, was diese Exception beschreibt. Die Objekte für Exceptions heißen normalerweise irgendwie mit `Exception` am Ende. Sie werden mit `new` erzeugt und danach mit throw geworfen:

beipiel

```
if(radikand < 0)
  throw new InvalidArgumentException
              ("Das ist mir zu komplex!");
```

Um eine Exception aufzufangen, verwendet man `try` und `catch`. Auf `try` folgt ein Block, der die Befehle enthält, die eine Exceptions auslösen können. In darauffolgenden `catch`-Blöcken können die Exceptions abgefangen werden, die im `try`-Block aufgetreten sind: Der `throw`-Befehl springt direkt in den `catch`-Block, der die entsprechende Exception abfängt. Der `catch`-Block muss nicht in der gleichen Prozedur stehen, wie der throw- Befehl, er kann es aber durchaus tun. Um zu sehen, was man mit Exceptions alles anstellen kann, siehe API-Referenz unter `java.lang.Throwable`.

1.2.14 noch etwas IO

IO oder I/O steht für Input/Output, also Ein- und Ausgabe. Da in diesem Programm mit dem Benutzer kommuniziert werden soll, bedarf es etwas mehr als `System.out.println`, was wir schon kennen. Zur Eingabe erzeugt man ein Objekt der Klasse `BufferedReader`, das die Eingabepufferung übernimmt. Bei Pufferung geht es darum, dass das Programm größere Blöcke vom Computer anfordert, und dann einzeln verarbeitet, als sich die Zeichen einzeln abzuholen (man geht ja auch nicht für jeden Satz in einem Buch erneut in die Bibliothek, sondern leiht sich das Buch aus. . .)

`System.out.print` gibt einen Text aus, ohne in die nächste Zeile zu wechseln, ist also für Eingabeaufforderungen recht gut geeignet. Die Eingabe des

Benutzers wird mit `inbuff.readLine` erledigt. Die Beschreibung in der API-Referenz findet man unter `java.io.BufferedReader`. Im Fall, dass bei der Eingabe in Zeile 42 etwas schief geht, was eigentlich nicht schiefgehen sollte (Ich habe gerade kein Beispiel provozieren können), gibt es aus dem `catch`-Block in den Zeilen 43 – 45 eine Meldung `IOProblem: `. `getMessage` ist eine Methode, die jede Exception hat, siehe API-Referenz unter `java.lang.Throwable`.

Nachdem geklärt ist, dass bei der Eingabe nicht die Welt untergegangen ist, wird geprüft, ob das Programm beendet werden soll. Dabei wird die schon im Abschnitt über Arrays erwähnte Methode `equals` verwendet, die zwei Objekte vergleicht. `null` ist ein Spezialwert, der für undefiniert steht, und dann von `readLine` erzeugt wird, wenn die Eingabe zu Ende ist (unter echten Betriebssystemen kann man nämlich die Eingabe eines Programmes umlenken, so dass es nicht von der Tastatur, sondern aus einer Datei liest).

Der Befehl `break` in Zeile 47 hinter dem `if` ist uns noch nicht über den Weg gelaufen. Er bricht eine Schleife einfach aus heiterem Himmel ab, und macht hinter der Schleife weiter. Man kann ja jetzt noch einmal einen Blick auf die Schleifenanweisung in Zeile 38 werfen. Dort steht `for(;;)`, was ein Standardbefehl für eine Endlosschleife ist. Es wird vor dem ersten Semikolon keine Initialisierung gemacht, zwischen den Semikolons steht keine Abbruchbedigung, und nach dem zweiten Semikolon steht auch kein Befehl, der den nächsten Durchlauf vorbereitet. Also wird einfach nur der folgende Block solange ausgeführt, bis etwas besonderes geschieht, normalerweise ein `break`.

Wie man in den Zeilen 54 – 61 sehen kann, können mehrere `catch`-Blöcke auf ein `try` folgen, wobei dann der entsprechende Block für den geworfenen Exception-Typ ausgeführt wird.

Als letztes erläutere ich `finally`. Auf jeden `try`-Block kann ein `finally`-Block folgen. Der `finally`-Block wird in dem Moment ausgeführt, wo der `try`-Block verlassen wird, egal auf welche Art. Wenn zum Beispiel ein `break` in einem `try`-Block auftaucht, wird der `try`-Block an der Stelle abgebrochen. Bevor hinter die Schleife gesprungen wird, wird erst noch der `finally`-Block ausgeführt. Zwischen `try` und `finally` können auch noch einige `catch`-Blöcke stehen. Falls dann eine Exception auftritt, wird der `try`-Block dort abgebrochen, wo die Exception entstand, dann der entsprechende `catch`-Block ausgeführt, und als letztes der `finally`-Block.

1.3 Objektorientiert Programmieren

1.3.1 Ein Beispiel für eine Klasse

Den Begriff Objekt verwende ich absichtlich nicht, da bei ihm nicht klar wird, ob eine Klasse, eine Instanz einer Klasse, oder nur einfach irgendein zusammengehöriger Speicherbereich gemeint ist. Das ist Geschmacksache.

Bis jetzt haben wir uns noch nicht genauer um Klassen gekümmert, gebraucht haben wir sie bis jetzt nur deshalb, weil alles in Klassen enthalten sein muss. Allerdings eröffnet einem objektorientierte Programmierung Möglichkeiten, die in der imperativen Programmierung nicht so einfach nachzubilden sind (sie sind es, man kann in imperativen Sprachen jeden Mist machen, es ist sogar mit fiesen Tricks möglich, in C die Möglichkeiten von Haskell zu bekommen, aber das nur ganz am Rande).

Eine Klasse ist eine Ansammlung von Variablen und Funktionen, die man als Felder und Methoden bezeichnet. Man kann von einer Klasse Instanzen erzeugen, die eine gewisse Unabhängigkeit voneinander haben.

Man könnte zum Beispiel eine Klasse Computer definieren. Instanzen dieser Klasse beschreiben dann verschiedene Computermodelle. Wichtige Felder in dieser Klasse sind dann Angaben wie Prozessortyp, Taktfrequenz Betriebssystem(e), Festplattengröße, Bildschirmtyp, Hersteller, Modellname. Als Methode könnte man eine Funktion schreiben, die berechnet, für welche Computerspiele der Computer tauglich ist. Man kann aber auch eine Methode schreiben, die prüft, ob der Einsatz von Computern in einem bestimmten Gebiet lohnt - ganz unabhängig davon, welche es sind, und dazu Statistiken in die Klasse packen.

Das ist aber schon ein sehr komplexes Beispiel, daher betrachten wir erst einmal die Beispielklasse `Kreis`, deren Instanzen verschieden große Kreise beschreiben:

Kreis.java

```
1  /**
2   * Kreis - eine Beispielklasse
3   */
4
5  public class Kreis {
6    /* Der Wert fuer Pi */
7    public final static double PI=3.141592653;
8
9    /* Der Radius des Kreises */
10   public double r;
11
12   /* Rechnet einen Winkel im Bogenmaß in
13      Grad um */
14   static double bogenInGrad(double bogenmaß) {
15     return bogenmaß/PI*180.0;
16   }
17
18   /* Berechnet die Fläche des Kreises */
19   double fläche() {
20     return PI*r*r;
21   }
22 }
```

Diese Klasse enthält die Konstante `PI`. Da sie allgemeingültig für alle Kreise gleich ist, braucht man sie nur einmal zu speichern. Daher wird sie als `static` erklärt, und wird damit zu einem *Klassenfeld*, was die gesamte Klasse nur einmal hat.

Der Radius dagegen kann für jeden Kreis anders sein. Daher wird er nicht als `static` deklariert und ist ein *Instanzfeld*. Das bedeutet, das jede Instanz der Klasse Kreis ihren eigenen Radius hat.

Die Funktion `bogenInGrad` ist vom Radius des Kreises unabhängig, da sie Winkel im Bogenmaß in Grad umrechnet. Würde sie den Kreisbogen eines bestimmten Kreises in Grad umrechnen, wäre das Ergebnis für verschiedene Instanzen der Klasse Kreis verschieden. Da sie das aber gerade nicht tut, kann sie als `static` deklariert sein, was bedeutet, dass sie unabhängig von jeden Instanz

arbeitet. Natürlich darf sie dann auch nur Klassenfelder und keine Instanzfelder benutzen. Eine solche Methode nennt man *Klassenmethode*.

Die Funktion `fläche` soll einem sagen, welchen Flächeninhalt ein Kreis hat. Dazu braucht man natürlich einen ganz speziellen Kreis mit einem bestimmten Radius. Also funktioniert `fläche` nur mit einer Instanz der Klasse Kreis, und nich alleine. Eine Methode, die eine Instanz zum Arbeiten benötigt, nennt man *Instanzmethode*.

So. Toll. Jetzt haben wir eine Klasse. Und wir haben zwei Datenfelder und zwei Methoden. Aber was macht man jetzt damit? Um die Frage zu klären, gibt es ein Beispielprogramm `TestKreis`:

TestKreis.java

```
/**
 * TestKreis - eine Klasse, die zeigt, wie man die
 * Kreisklasse benutzen kann
 */

public class TestKreis {
  public static void main(String[] args)
  {
    /* Benutze ein Klassenfeld */
    System.out.println("Pi ist "+Kreis.PI);

    /* Benutze eine Klassenmethode */
    System.out.println("1 im Bogenmaß sind " +
                 Kreis.bogenInGrad(1) + " Grad");

    /* Erzeuge eine Instanz */
    Kreis k = new Kreis();

    /* Setze ein Instanzfeld */
    k.r = 3;

    /* Benutze ein Instanzmethode */
    System.out.println("Der Flächeninhalt des"
                  + " Kreises ist " + k.fläche());
  }
}
```

Eigentlich sollten die Kommentare im Programm schon deutlich machen, worum es eigentlich geht, daher fasse ich mich in der Erläuterung etwas kürzer, und erkläre es gleich allgemeiner: Auf Klassenfelder und Klassenmethoden wird zugegriffen, indem man den Namen der Klasse schreibt, dann einen Punkt und den Namen des Feldes oder der Methode. Auf Instanzfelder wird zugegriffen, indem man den Namen der Instanz, einen Punkt und den Namen von Feld oder Methode angibt.

Interessant ist allerdings Zeile 17, in der eine Instanz erzeugt wird. Wie Arrays sind auch Variablen, deren Typ eine Klasse ist, eigentlich nur Referenzen auf Instanzen dieser Klasse. Diese wird mit `Kreis k` erklärt. Mit dem `new Kreis();` wird ein neues Objekt der Klasse Kreis eingerichtet, auf das die Referenz `k` dann verweist.

Da das Beispielprogramm bis auf diese `new`-Zeile trivial ist, erkläre ich es für erklärt, und beschreibe die Klasse Kreis noch etwas genauer. Das Klassenfeld `PI` ist als `final` deklariert, und daher eine Konstante. Das sollte uns alles bekannt vorkommen. Durch das `static` wird, wie oben erwähnt, die Zugehörigkeit zur Klasse festgelegt.

In der nächsten Deklaration wird der Radius `r` als double deklariert. Auch das ist eigentlich nichts besonderes. Da er nicht `static` ist, existiert der Radius allerdings innerhalb der Instanzen der Klasse, und nicht wie `PI` in der Klasse selber.

Die Funktion `bogenInGrad` benutzt das Klassenfeld `PI`. Moment! Gerade wurde doch erklärt, dass man es mit `Kreis.PI` erreicht. Das stimmt natürlich auch, aber innerhalb der Klasse `Kreis` ist die Wahrscheinlichkeit recht groß, dass man auf Felder und Methoden dieser Klasse zugreifen will, und daher kann man das `Kreis.` weglassen. Diese Funktion arbeitet, wie erklärt, für alle Kreise gleich (eigentlich hat sie auch nichts mit Kreisen zu tun, aber in irgendeinem Objekt muss sie stecken), und kann daher als `static` erklärt werden.

Die Funktion `fläche` benutzt nicht nur das Klassenfeld `PI`, auf das ohne `Kreis.` zugegriffen wird, da wir uns innerhalb der Klasse befinden, sondern auch das Instanzfeld `r`. Java weiß, wo `r` zu finden ist, weil `fläche` als Methode einer Instanz aufgerufen werden muss, und diese Instanz genau ein r enthält. Die eigene Instanz ist auch über den Namen `this` ansprechbar, also kann die Funktion `fläche` auch die Zeile

```
return this.r*this.r*Kreis.PI;
```

enthalten. Hier sieht man deutlich, dass `PI` aus der Klasse `Kreis` stammt, wogegen `r` aus der Instanz, die die Funktion aufgerufen hat, kommt.

1.3.2 Konstruktoren

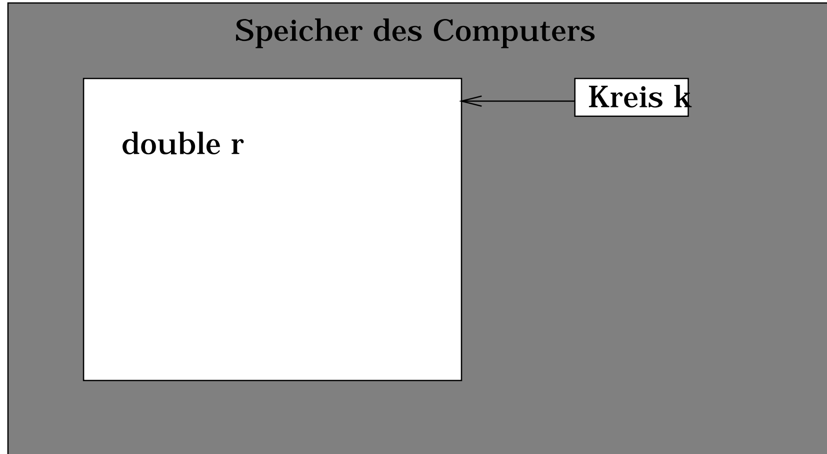
Wenn eine Instanz einer Klasse angelegt wird, kann es nötig sein, die Instanzfelder auf bestimmte Werte zu initialisieren, oder auf andere Art die Arbeit der Klasse in dieser Instanz vorzubereiten. Dazu gibt es die sogenannten Konstruktoren. Wenn man sich das Stück `new Kreis();` genau ansieht, wirkt das wie ein Funktionsaufruf. Genau das ist es auch, nämlich der Aufruf eines Konstruktors der Klasse Kreis, der keine Parameter erhält. Aber wir haben ja gar keinen Konstruktor definiert, wir wissen ja noch nicht einmal wie das geht! Das ist kein Problem, denn Java erzeugt automatisch einen Konstruktor, der einfach nichts tut.

Vielleicht hält man es für sinnvoll, direkt beim Erzeugen des Kreises den Radius angeben zu können. Es ist auf jeden Fall sinnvoll, daran zu zeigen, wie Konstruktoren definiert werden können:

```
class Kreis {
  ...
  Kreis() {
    r = 1.0;
  }
  Kreis(double r) {
    this.r = r;
```

```
  }
}
```

Wenn man jetzt `new Kreis()` schreibt, wird mit `new` Speicherplatz für das Objekt belegt, und danach der Konstruktor für dieses neue Objekt aufgerufen. Da keine Parameter angegeben sind, wird der erste Konstruktor benutzt, und der Radius auf eins gesetzt. Recht anschaulich wird das ganze mit einem Kasten dargestellt:



Der Kasten wird von `new` angelegt. Der Konstruktor der Klasse sorgt dafür, dass in dem Kasten die Ordnung herrscht, die für die Klasse `Kreis` benötigt wird. Schreibt man `Kreis k = new Kreis()`, so wird hinterher in der Referenz `k` eingetragen, wo sich der Kasten befindet.

Mit `Kreis k2 = new Kreis(1.5)` erzeugt man einen weiteren Kreis, der unabhängig vom ersten ist, und den Radius 1.5 hat. Dabei wird der zweite Konstruktor verwendet, der den Radius des Kreises auf den Wert setzt, der ihm als Parameter mitgegeben wurde. Dass dazu ein `this` benötigt wird, liegt daran, dass der Parameter `r` genauso heißt wie das Feld `r`. Gibt man einfach nur `r` an, sieht man den Parameter, er „verdeckt" das Feld. Schreibt man aber explizit `this` davor, muss es sich um ein Feld handeln, und dann wird das Feld `r` benutzt.

1.3.3 Destruktoren

Wenn Klassen mit einer Methode angelegt werden, liegt es Nahe, auch beim Abbau der Klasse wieder eine Methode aufzurufen. Üblicherweise werden solche Methoden als Destruktoren bezeichnet. In Java ist die Funktion ein Destruktor, die `finalize` heißt. Sobald ein Objekt im Programm nicht mehr benutzt wird (oder etwas später, wenn der Speicher wieder benötigt wird), wird der Destruktor aufgerufen.

Da wir gelernt haben, dass in Java sämtlich Objekte mit `new` angelegt werden, und die Referenzen darauf beliebig an andere Objekte weitergegeben werden können, kann man nicht sagen, dass ein Objekt, was in der Funktion A erzeugt wurden, auch am Ende der Funktion A wieder abgeräumt werden kann.

Daher gibt es in Java den so genannten „Garbage Collector", der prüft, ob es Objekte gibt, auf die keine Referenz mehr zeigt. Er läuft im Hintergrund und

daher ist nicht vorherzusagen, wann genau ein Objekt als unbenutzt erkannt wird. Der Garbage Collector ist so programmiert, dass er auch geschlossene Ketten von Objekten findet, die jeweils auf das nächste verweisen, so dass auch sie sauber abgeräumt werden können.

1.4 Geltungsbereich

Bereits im Abschnitt über Definitionen wurde der Begriff Geltungsbereich verwendet. Er beschreibt den Teil des Programmes, in dem eine bestimmte Definition wirksam ist. Mit dem Geltungsbereich einer Variablen oder einer Funktion meint man normalerweise den Bereich, in dem sie ansprechbar ist. Fast immer ist der Geltungsbereich der aktuelle Block (mit geschweiften Klammern begrenzt), mit allen Unterblöcken, sofern diese nicht eine genauso benannte Variable enthalten:

```
{
  int i = 3;
  {
    int i = 5;
    System.out.println("Innen ist i "+i);
  }
  System.out.println("Außen ist i "+i);
}
/* System.out.println("Gibt einen Compilerfehler "+i); */
```

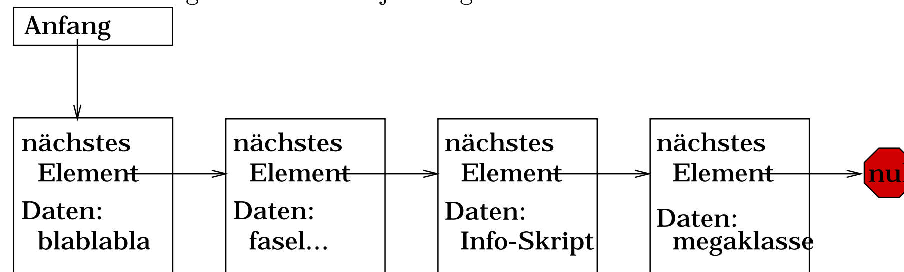
gibt erst 5, und dann 3 aus, weil im inneren Block das `i` aus dem äußeren Block verdeckt ist. Außerhalb des äußeren Blockes ist gar kein `i` bekannt, so dass es einen Compilerfehler gibt, wenn man die Kommentarzeichen entfernt.

Eine Ausnahme von dieser Regel ist die for-Schleife. Variablen, die im Initialisierungsabschnitt definiert werden, sind nur innerhalb der Schleife, inklusive Bedingung und Schrittanweisung vorhanden.

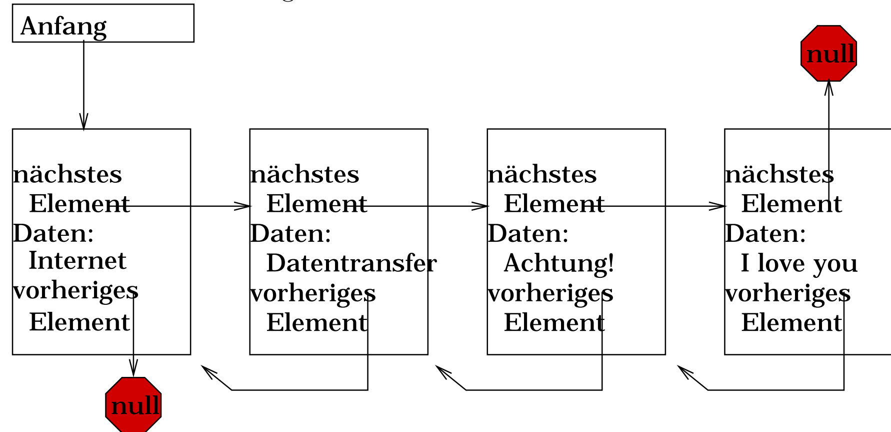
1.5 Listen

Dieser Abschnitt ist recht kurz gehalten, da er nur einem eine Vorstellung geben soll, wie eine Liste ungefähr funktioniert. Also gerade so weit, dass man kapiert, was in etwa mit der Übungsaufgabe 3-4 gemeint sein könnte.

Eine Liste ist eine Kette von Objekten, die jeweils einen Verweis auf das nächste Element der Liste enthalten. Eine solche Liste heißt „einfach verkettet“, da es eine einzige Kette von Objekten gibt.



Wenn man auch rückwärts durch die Liste gehen will, ist es sehr ungünstig, wenn man in einem Objekt immer nur eine Verweis auf das nächste stehen hat, weil man dann die ganze Liste nach dem Objekt durchsuchen müsste, dass auf das aktuelle Objekt zeigt. Daher benutzt man in solchen Fällen doppelt verkettete Listen, die außer einer Referenz auf das nächste Element auch einen Verweis auf das vorherige Element enthalten.



1.6 Brüche

Fraction.java

```
/**
 * The Fraction class implements some aspects of fractions.
 * Each fraction is a pair numerator/denominator of integers
 * We represent fractions in a standardized way
 * (1) denominator > 0  (2) gcd(num,demom)=1
 */
public class Fraction implements Cloneable{

    /**
     * the numerator
     */
    private int num;

    /**
     * the denominator.  It is always > 0
     */
    private int denom;

    /**
     * Constructor without arguments, constructs 0 as fraction 0/1
     */
    public Fraction() {
        num = 0; denom = 1;
    }

    /**
     * Constructs Fraction object from an int
     * Parameter: a yields the fraction a/1
     */
    public Fraction( int a ) {
        num = a; denom = 1;
    }
```

```
/**
 * Constructor with two integer argument a and b,
 * constructs the fraction a/b and simplifies it.
 * Parameter: a the numerator
 * Parameter: b the denominator
 */
public Fraction( int a, int b ) {
    if ( b == 0 ) throw new ArithmeticException(
                   "Division by zero in constructor" );
    if ( a == 0 ) { num = 0; denom = 1; }
    else { int tmp = gcd( a, b ); num = a/tmp; denom = b/tmp; }
    if ( denom < 0 ) { num = - num; denom = -denom; }
}

/**
 * Calculates the greatest common divisor of a and b
 * Parameter: a
 * Parameter: b
 * return the greatest common divisor of
 * a and b
 */
private static int gcd( int a, int b ) {
    if ( a == 0 || b == 0 ) throw new ArithmeticException(
                   "Zero argument in gcd calculation" );
    a = Math.abs( a ); b = Math.abs( b );
    while ( a != b ) if ( a > b ) a = a - b; else b = b - a;
    return a;
}

/**
 * Returns a clone of the fraction
 * This method overrides the corresponding method
 * from the Object class -> return type is Object
 */
public Object clone(){
      Fraction s = new Fraction(num,denom);
      return s;
}

/**
 * Multiplies this fraction with other fraction r and
 * simplifies the result.
 * Parameter: r the fraction to be multiplied with.
 * return product of the fractions in simplified form.
 */
public Fraction multiply( Fraction r ) {
    num = num * r.num; denom = denom * r.denom;
    int tmp = gcd( num, denom );
    num = num/tmp; denom = denom/tmp;
    return this;
}

/**
 * Multiplies  fraction r and fraction s
 * this is a class method and therefore static
 */
public static Fraction multiply( Fraction r, Fraction s ) {
    Fraction f = (Fraction) r.clone(); // cast required!
    return f.multiply(s);
}
```

```
    /**
     * Returns the double value of this fraction.
     * return num/denum
     */
    public double doubleValue() {
        return ( double ) num / ( double ) denom;
    }

    /**
     * Returns a string representation of this fraction.
     * return fraction in the form  "num/denum"
     */
    public String toString() {
        return Integer.toString( num ) + "/" + Integer.toString( denom );
    }

}
```

Mit der Klasse `Fraction` haben wir eine Klasse betrachtet, die es erlaubt, mit Brüchen zu rechnen, wobei Zähler und Nenner jeweils als `int` deklariert sind. Im Konstruktor wird der Bruch auf 0, eine natürliche Zahl, oder einen Bruch gesetzt, je nachdem, mit wie vielen Parametern der Konstruktor aufgerufen wird. Die Klasse `Fraction` enthält eine statische Funktion `gcd`, die den größten gemeinsamen Teiler zweier Zahlen bestimmt. Das braucht man, um die Brüche zu kürzen. Desweiteren existiert eine Instanzmethode `multiply`, einen Bruch mit einem anderen multipliziert, und eine Klassenmethode multiply, die das Produkt zweier Brüche berechnet. Dazu kopiert sie erst einen Bruch, und multipliziert diese Kopie dann mit dem zweiten Bruch. Das Ergebnis gibt sie dann zurück. Um den Bruch zu kopieren wird die Methode `clone` aufgerufen. `clone` ist ein von Java vordefinierter Begriff, der das Erstellen einer Kopie bedeutet. Jede Klasse, die die `clone`-Fähigkeit haben soll, muss mit `implements Clonable` definiert werden. Wenn eine Objekt geklont werden soll, dessen Klasse `Clonable` nicht implementiert, dann gibt es eine Exception.

1.7 Vererbung und Typecasts

Unter Vererbung versteht man die Möglichkeit, eine Klasse zu erweitern oder zu spezialisieren, in dem man alles aus einer Klasse übernimmt und eigene Methoden und Felder hinzufügt oder vorhandene umdefiniert. Als Beispiel nehmen wir ruhig die schon oben erwähnte Klasse für Computer. Natürlich ist es durchaus möglich, sämtliche Arten von Computern mit einer Klasse zu beschreiben. Allerdings kann es sinnvoll sein, bestimmte Methoden nur für bestimmte Typen anzubieten (Es macht zum Beispiel keinen Sinn, eine Playstation danach zu fragen, was die größte Festplatte ist, die sie kennt), oder einfach Methoden, die nach Typ verschieden sind, nicht in der Klasse Computer jedes Mal prüfen zu lassen, welcher Computertyp es ist, sondern in Unterklassen zu stecken. So könnte man zum Beispiel von der Klasse Computer die Klassen Spielkonsole, PC, Macintosh, Workstation und Unix-Server ableiten.

Der Witz an der Vererbung ist, dass die Klassen dabei Spezialfähigkeiten

erhalten, aber ihren gemeinsamen Ursprung dazu ausnutzen können, in ein Array, was allgemein Computer enthält, eingetragen werden zu können, obwohl es verschiedene Klassen sind.

Da wir als Beispiel für eine Klasse den Kreis benutzt haben, gibt es jetzt eine Beispielklasse, die von der Klasse Kreis erbt, nämlich `EbenerKreis`. Diese Klasse beschreibt Kreise in einer Ebene, die also zusätzlich zu ihrem Radius auch noch einen Mittelpunkt (x- und y-Koordinate) speichern. Es gibt darin eine Methode `enthält`, die prüft, ob ein Punkt innerhalb des Kreises liegt.

EbenerKreis.java

```
/**
 * Kreis, der in einer Ebene liegt
 */

/* Kreis2 ist die Klasse Kreis mit Konstruktoren */
public class EbenerKreis extends Kreis2 {
  public double mx,my;

  /* Konstruktor fuer einen Kreis unter Angabe
     von Radius und Mittelpunkt */
  public EbenerKreis(double r, double x, double y) {
    super(r);
    mx = x;
    my = y;
  }

  /* Prüft, ob ein Punkt im Kreis liegt. Algorithmus
     nach Pythagoras, dem Griechen. Genaueres siehe
     Mathematiklehrbücher der Mittelstufe */

  public boolean enthält(double x, double y) {
    double dx = mx - x;
    double dy = my - y;
    double mpa =              // Mittelpunktsabstand
      Math.sqrt(dx*dx+dy*dy);
    return mpa <= r;
  }
}
```

Man nennt `Kreis` (im Beispiel steht `Kreis2`, weil die Klasse `Kreis` das Beispiel ohne Konstruktor ist) die Superklasse oder Oberklasse von `EbenerKreis`. Der Konstruktor der Superklasse wird mit `super` aufgerufen. Hier wird nur der Radius nach oben weitergereicht, der Mittelpunkt ist für `Kreis` ja auch uninteressant.

Man kann die Klasse ganz normal benutzen:

```
EbenerKreis ek = new ek(5,3,4);
Kreis k = ek;
if(k.enthält(1,1)){...}  // GEHT NICHT
if( ((EbenerKreis) k).enthält(1,1))
   { ...}
```

In der ersten Zeile wird ein Kreis mit dem Radius 5 um den Punkt $(3, 4)$ angelegt. In Zeile 2 wird eine Referenz auf ein allgemeines Kreisobjekt erzeugt, das auf die Instanz von `ek` der Klasse `EbenerKreis` verweist. Das klappt, weil `EbenerKreis` eine Erweiterung von `Kreis` ist, und daher auch als normaler Kreis funktioniert. In Zeile 3 dagegen gibt es einen Compilerfehler, denn für einen Kreis ist die Methode `enthält` gar nicht definiert. Wenn man sich sicher ist, das `k` eigentlich auf ein `EbenerKreis`-Objekt verweist, dann kann man `k` „casten", das heißt wörtlich übersetzt umgießen, also der JVM klar machen, dass man eine Instanz des Typs `EbenerKreis` erwartet. Für so einen Cast (oder Typecast) schreibt man den Namen des Typs, als der die Variable interpretiert werden soll, in runde Klammern. Genau das wird in Zeile 4 auch gemacht. Ist jetzt aber `k` gar keine Referenz auf eine `EbenerKreis`-Instanz, gibt es eine `ClassCastException`.

Ein Cast auf Superklassen ist dagegen nie nötig, da Java das von alleine erledigt, da es immer funktioniert.

Casts sind außerdem nötig, wenn eine allgemeine Liste zum Beispiel Integer enthält und sortiert werden soll. In diesem Fall bekommt man aus der Liste Referenzen auf `Object`, einer ganz allgemeinen Klasse, und muss sie erst in Referenzen auf `Integer` casten, bevor sie verglichen werden können. Dabei muss man auch noch einmal aufpassen: Es gibt den primitiven Typ `int` und die Klasse `Integer`, die zum Speichern eines `int`s als Objekt benutzt wird. Der Vergleich zwischen zwei `Object`s, die `Integer`s sind, funktiniert allerdings etwas anders als in der Vorlesung angegeben, das was da stand geht nur auf `BigInt` und `BigDecimal`, zwei Objekttypen für beliebig große Zahlen.

CastTest.java

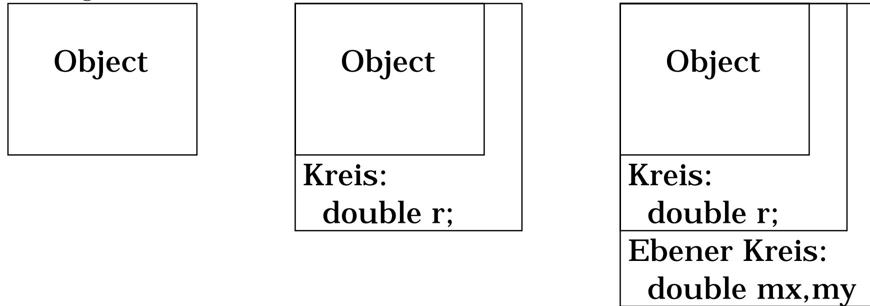
```
1  public class CastTest {
2    public static void main(String[] args) {
3      int i=3;
4      int j=4;
5      Object oi = new Integer(i);
6      Object oj = new Integer(j);
7      System.out.println(
8         ((Integer)oi).intValue() <
9         ((Integer)oj).intValue());
10   }
11 }
```

Jede Klasse hat genau eine Superklasse. Gibt man keine Superklasse an, wird automatisch `Object` als Superklasse eingesetzt. Daher kommt es, dass sämtliche Klassen unter Java eine Baumstruktur bilden.

1.7.1 Konstruktorverkettung

Es ist, wie wir bereits im Beispiel `EbenerKreis` gesehen haben, möglich, die Konstruktoren der Superklassen aus dem Konstruktor aufzurufen. Genau genommen ist es nicht nur möglich, sondern sogar zwingend, ebenso wie es zwingend ist, dass jede Klasse einen Konstruktor hat. Wenn man selber keinen Konstruktor definiert, wird ein Konstruktor generiert, der den Konstruktor der Superklasse ohne Parameter aufruft. Definiert man einen Konstruktor ohne Aufruf des übergeordneten Konstruktors, wird der Aufruf automatisch vor dem ersten Befehl

eingefügt. Der automatische Aufruf eines übergeordneten Konstruktors macht dann Probleme, wenn es in der Superklasse gar keinen Konstruktor ohne Parameter gibt.



Konstruktor.java

```
/**
 * Beispiel für die Konstruktorverkettung
 */



/* Superklasse hat selber keine Superklasse
   angegeben, also ist sie direkt von Object
   abgeleitet */

class Superklasse {
  public Superklasse() {
    System.out.println("Superklasse();");
  }
  public Superklasse(int i) {
    System.out.println("Superklasse("+i+")");
  }
}

/* Subklasse1 enthält gar keinen Konstruktor */

class Subklasse1 extends Superklasse {
}

/* Subklasse2 enthält einen Konstruktor ohne Aufruf des
   Konstruktors von Superklasse */

class Subklasse2 extends Superklasse {
  public Subklasse2() {
    System.out.println("Subklasse2()");
  }
}

/* Subklasse3 enthält nur einen Konstruktor mit einem Parameter */

class Subklasse3 extends Superklasse {
  public Subklasse3(int i) {
```

```
    super(i);
    System.out.println("Subklasse3("+i+")");
  }
}

/* Sub3Subklasse enthält keinen Konstruktor und ist von
   Subklasse3 abgeleitet. Es gibt Ärger mit dem Compiler */

//class Sub3Subklasse extends Subklasse3 {
//}

public class Konstruktor {
  public static void main(String[] args) {
    System.out.println("Lege ein Objekt von Superklasse an:");
    Object o1 = new Superklasse();
    System.out.println("Lege ein Objekt von Subklasse1 an:");
    Object o2 = new Subklasse1();
    System.out.println("Lege ein Objekt von Subklasse2 an:");
    Object o3 = new Subklasse2();
    System.out.println("Lege ein Objekt von Subklasse3 an:");
    Object o4 = new Subklasse3(15);
//    System.out.println("Lege ein Objekt von Sub3Subklasse an:");
//    Object o5 = new Sub3Subklasse();
  }
}
```

1.7.2 Beschatten und Überschreiben

Man kann in Klassen Variablen definieren, die genauso heißen, wie Variablen in einer Superklasse dieser Klasse. Angenommen, die Klasse `B` sei von `A` abgeleitet, und beide enthalten eine Variable `x`. Jetzt sei `BInstanz` eine Instanz der Klasse `B`. Dann benutzt `BInstanz.x` die Variable `x` aus der Klasse `B`. Wenn man aber eine Referenz `ARef` auf eine Klasse `A` auf `BInstanz` zeigen läßt, wird mit `ARef.x` das `x` aus der Klasse `A` verwendet. In `BInstanz` sind also beide Variablen `x` enthalten, und je nach dem, auf welche Methode man darauf zugreift, erhält man die eine oder die andere Variable. Man sagt `x` aus `B` *beschattet* `x` aus `A`, weil `x` aus `A` zwar vorhanden ist, aber normalerweise nicht gesehen wird. Innerhalb von Methoden von `B` wird auf das `x` aus `B` zugegriffen, es ist aber dort ebenfalls möglich, das `x` aus `A` zu erhalten, in dem man entweder `super.x` schreibt, oder `this` auf eine Referenz auf `A` castet. In einer komplexeren Hierarchie mit mehreren Stufen, wo eine Variable `x` existiert, bleibt einem nur der Weg, `this` zu casten, wenn man auf übergeodnete `xe` zugreifen will.

Schatten.java

```
/**
 * Programm zur Demonstration der Beschattung
 */

/* Die Kommentare in diesem Programm entfallen,
   da alles im Skript beschrieben steht */
```

```
7
8   class Superklasse {
9     String text;
10    public Superklasse() {
11      text = new String("Dies ist Superklassse");
12    }
13
14    /* Diese Methode wird immer den Superklassen-
15       text ausgeben, da sie gar nicht weiss, dass
16       text irgendwo beschattet wird */
17
18    public void printtext() {
19      System.out.println(text);
20    }
21  }
22
23  class Subklasse extends Superklasse {
24    String text;   /* Überschattet den alten text */
25    public Subklasse() {
26      text = new String("Dies ist Subklasse");
27    }
28
29    /* Diese Methode gibt zuerst den Subklassentext,
30       und dann zwei mal den Superklassentext aus */
31
32    public void printsubtext() {
33      System.out.println("text:"+text);
34      System.out.println("super.text:"+super.text);
35      System.out.println("((Superklasse)this).text"+
36                            ((Superklasse)this).text);
37    }
38  }
39
40  class Subsubklasse extends Subklasse {
41    String text;
42    public Subsubklasse() {
43      text = new String("Dies ist Subsubklasse");
44    }
45
46    /* Diese Methode gibt alle drei Texte aus,
47       sie sieht genauso aus, wie die Methode
48       in Subklasse */
49
50    public void printsubsubtext() {
51      System.out.println("text:"+text);
52      System.out.println("super.text:"+super.text);
53      System.out.println("((Superklasse)this).text"+
54                            ((Superklasse)this).text);
55    }
56  }
```

```
57
58  public class Schatten {
59    public static void main(String[] args) {
60      Subsubklasse x = new Subsubklasse();
61      System.out.println("Rufe printtext auf:");
62      x.printtext();
63      System.out.println("Rufe printsubtext auf:");
64      x.printsubtext();
65      System.out.println("Rufe printsubsubtext auf:");
66      x.printsubsubtext();
67    }
68  }
```

Ausgabe des Programms

```
1   Rufe printtext auf:
2   Dies ist Superklassse
3   Rufe printsubtext auf:
4   text:Dies ist Subklasse
5   super.text:Dies ist Superklassse
6   ((Superklasse)this).textDies ist Superklassse
7   Rufe printsubsubtext auf:
8   text:Dies ist Subsubklasse
9   super.text:Dies ist Subklasse
10  ((Superklasse)this).textDies ist Superklassse
```

Man kann nicht nur Variablen, sondern auch Methoden in untergeordneten Klassen erzeugen, die genauso heißen, wie Methoden der Superklasse. Allerdings wird dann automatisch beim Aufruf der Methoden die Methode aufgerufen, die zu der Klasse gehört, aus der die gegebene *Instanz* stammt. Bei den beschatteten Variablen wurde die Variable genommen, die zu der Klasse gehört, aus der die *Referenz* stammt. Dieses Verhalten wird „dynamic method lookup“ genannt, da erst in dem Moment, wo das Objekt feststeht, dessen Methode aufgerufen wird, auch feststeht, welcher Methode aufgerufen wird.

Schreiben.java

```
1   /**
2    * Beispiel zum Überschreiben von Methoden
3    */
4
5   class Klasse1 {
6     public String getName() {
7       return new String("Klasse1");
8     }
9     public void printName() {
10      System.out.println("Diese Instanz gehört zu "+
11                                        getName());
12    }
13  }
14
15  class Klasse2 extends Klasse1 {
16
```

```
17   /* Überschreibt die Methode von oben */
18   public String getName() {
19     return new String("Klasse2");
20   }
21 }
22
23 class Klasse3 extends Klasse1 {
24
25   /* Überschreibt die Methode von oben und benutzt
26      sie innerhalb */
27   public String getName() {
28     return new String("Klasse3 (extends "
29                       +super.getName()+")");
30   }
31 }
32
33 public class Schreiben {
34   public static void main(String[] args) {
35     Klasse2 meineKlasse = new Klasse2();
36     Klasse1 Klasse1ref = meineKlasse;
37     meineKlasse.printName();
38     Klasse1ref.printName();
39
40     Klasse3 nochnTest = new Klasse3();
41     nochnTest.printName();
42   }
43 }
```

Ausgabe des Programms

```
1 Diese Instanz gehört zu Klasse2
2 Diese Instanz gehört zu Klasse2
3 Diese Instanz gehört zu Klasse3 (extends Klasse1)
```

1.8 Verkapselung

Unter Verkapselung versteht man, dass nicht jede Funktion auf jede Variable zugreifen darf, sondern es genau kontrolliert wird, wer was tut, damit es einfacher wird, Fehler zu finden, da man recht bald sehen kann, welche Funktionen daran Schuld sein können, dass eine Variable den falschen Wert hat. Es gibt aber auch Variablen, bei denen beabsichtigt ist, dass sie von vielen Stellen aus geändert werden können. Daher gibt es in Java verschiedene Schutzstufen für Variablen, wobei in Prozeduren definierte Variablen aus dem Schema herausfallen – bei denen ist sowieso klar, dass sie nur von der Prozedur benutzt und verändert werden, in der sie definiert sind.

Die Variablen und Methoden, die in Klassen definiert sind (damit meine ich sowohl Klassen-, als auch Instanzvariablen und -methoden) können mit den Modifiern `public`, `protected` und `private` versehen werden. Ist eine Methode oder Variable als `public` deklariert, darf von jeder Stelle des Programms diese Methode aufgerufen bzw. auf diese Variable zugegriffen werden. Ist eine Variable

oder Methode als `private` deklariert, dürfen nur Methoden des Objektes, in dem die Variable bzw. Methode deklariert ist, diese verwenden. Ist gar kein Modifier angegeben, kann jede Klasse des Pakets die Funktion bzw. Variable benutzen. Außerhalb des Paketes ist die Variable tabu. Bei der Verwendung von `protected` erweitert sich der Bereich, in dem die Variable oder die Methode benutzt werden kann, auf alle Unterklassen des Objektes. Oder kurz in Tabellenform:

Modifier	Benutzbar in der eigenen Klasse	dem gleichen Paket	Unterklassen in anderen Paketen	anderen Orten
`public`	JA	JA	JA	JA
`protected`	JA	JA	JA	NEIN
keiner	JA	JA	NEIN	NEIN
`private`	JA	NEIN	NEIN	NEIN

1.8.1 Pakete

Pakete sind eine Ansammlung zusammengehöriger Klassen. Innerhalb von Paketen sind die Zugriffsbeschränkungen deutlich geringer, wie aus der Tabelle zu sehen ist. Daher ist es sinnvoll, Klassen, die miteinander zu tun haben in dasselbe Paket zu legen. Das ist insbesondere dann wichtig, wenn zwei Klassen nicht voneinander abgeleitet sind (wie zum Beispiel `ListElement` und `DoubLiList`), aber dennoch eng verknüpft arbeiten. Nur mit Paketen ist es möglich, solche unabhängigen Klassen aufeinander Zugriff zu gewähren, ohne das alle Welt auf die Felder Zugriff hat.

1.8.2 Abstrakte Klassen

Wenn wir ein Geometrieprogramm entwickeln, werden wir verschiedene geometrische Formen behandeln wollen, die aber alle die Möglichkeit haben, Umfang und Flächeninhalt zu berechnen. Dazu könnte man einfach eine Klasse `Form` definieren, die diese Methoden hat, und die Klassen `Dreieck`, `Kreis` und `Rechteck` davon ableiten. Allerdings ist die Frage, was dann mit Instanzen der allgemeinen Klasse Form passieren soll. Und was sollen bitte die Funktionen `umfang` und `fläche` in der Klasse `Form` tun?

Als eine elegante Lösung bietet Java dazu abstrakte Klassen und Methoden an. Eine abstrakte Klasse kann per Definition keine Instanzen haben, und sie darf abstrakte Methoden enthalten. Abstrakte Methoden sind eine Art Platzhalter für Methoden, die in abgeleiteten Klassen überschrieben werden müssen. Werden in einer Klasse, die von einer abstrakten Klasse abgeleitet ist, nicht alle abstrakten Methoden überschrieben, so muss die Klasse ebenfalls als abstrakt definiert werden, da sie noch unausgefüllte Platzhalter enthält.

Abstrakte Klassen definiert man, in dem man den Modifier `abstract` verwendet, für abstrakte Methoden gilt das gleiche. Die Klasse `Form`, die als Basis für die geometrischen Objekte dient, sieht dann so aus:

Form.java

```
1 public abstract class Form {
2   public abstract double umfang();
3   public abstract double fläche();
4 }
```

In den davon abgeleiteten Klassen muss man nichts weiter beachten, als dass die Methoden `umfang` und `fläche` definiert werden müssen. Es gibt keine besonderen Syntaxkonstrukte dafür, die Angabe `extends Form` in der Klassendefinition ist ausreichend:

FKreis.java

```
/**
 * FKreis - eine von Form abgeleitete Klasse
 */

public class FKreis extends Form {
  /* Der Wert fuer Pi */
  public final static double PI=3.141592653;

  /* Der Radius des Kreises */
  public double r;

  /* Konstruktor, der den Radius als Argument
     nimmt */
  public FKreis(double newr) {
    r = newr;
  }

  /* Berechnet die Fläche des Kreises */
  public double fläche() {
    return PI*r*r;
  }

  /* Berechnet den Umfang des Kreises */
  public double umfang() {
    return 2*PI*r;
  }
}
```

Der Sinn davon, alle Formen von einer Klasse abzuleiten liegt darin, dass man dann eine allgemeine Referenz auf eine Form haben kann, und man sich nicht weiter darum zu kümmern braucht, was für eine Form das ist. Ein Programmfragment könnte etwa so aussehen:

```
  Form[] fo = new Form[10];  // Ein Array von 10
                             // beliebigen Formen

  /* Erzeugen von 10 Formen... */
  fo[0] = new Kreis(3);
  fo[1] = new Rechteck(5,20);
  //...
  fo[9] = new Dreieck(3,4,5);

  /* Berechne die Summe aller Flächen */
  double gesamtfläche = 0;
```

```
12    for(int i = 0;i < fo.length;i++)
13      gesamtfläche += fo[i].fläche();
```

Dabei wird in Zeile 13 immer die korrekte Methode `fläche` des Objekts aufgerufen, da, wie oben erläutert, Methoden überschrieben werden, wenn sie in abgeleiteten Klassen erneut definiert werden, und die Eigenschaft von überschriebenen Methoden nicht verloren geht, wenn man eine Referenz auf die Klasse hat, wo die Originalmethode (in diesem Falle sogar nur ein Platzhalter) stand.

1.9 Interfaces

Jetzt könnte man auf die Idee kommen, die Klasse `EbenerKreis` mit in unser Geometrieprogramm einzubauen. Die Klasse existiert ja schon, und außerdem wird, wenn man das ganze auf dem Bildschirm darstellen will, es sowieso nützlich sein, eine Position zu dem Objekt zu haben. Genauso wird man dann auch Klassen für Dreiecke und Rechtecke haben wollen, die eine feste Position haben.

Es spricht natürlich nichts dagegen, die Klassen jeweils von `Kreis`, `Rechteck` und `Dreieck` abzuleiten, aber dann hat man ein Problem, und zwar haben wir gerade die schöne abstrakte Klasse `Form` eingeführt, um alle Formen zu vereinheitlichen. In dieser Klasse `Form` gibt es ganz absichtlich keinen Mittelpunkt und keine Funktion, die prüft, ob ein Punkt im Objekt ist, da diese Klasse unabhängig von Koordinaten arbeitet, also können wir nicht wie oben einfach eine Methode `enthält` überschreiben, so dass man für eine beliebige gegebene Form prüfen kann, ob sie einen Punkt enthält.

Für diesen Zweck muss eine neue Superklasse her, die die abstrakte Methode `enthält` enthält, und von der die neuen Klassen `EbenerKreis`, `EbenesDreieck` und `EbenesRechteck` abgeleitet sind. Doch Moment! In Java kann eine Klasse immer nur von einer Basisklasse abgeleitet sein, also kann `EbenerKreis` dann nicht mehr von `Kreis` abgeleitet sein. Das soll aber auch nicht der Fall sein, denn dann kann man die Methode Fläche aus `Kreis` in `EbenerKreis` nicht einfach übernehmen.

Daher gibt es einen Spezialfall von abstrakten Klassen mit dem Namen Interface. Ein Interface ist eine abstrakte Klasse, die *nur* abstrakte Methoden enthält. Eine Klasse kann nämlich zusätzlich zur Abstammung von einer Basisklasse noch beliebig viele Interfaces implementieren, das bedeutet, dass sie die Methoden aus dem Interface überschreibt. Genauso wie Referenzen auf Klassen (also die üblichen Klassenvariablen) kann man auch Referenzen auf Interfaces definieren, und zum Beispiel Arrays daraus anlegen.

Wir definieren jetzt also ein Interface `Eben`, wozu man anstatt des Wortes `class` das Wort `interface` schreibt, und keine Methode wirklich definiert (allerdings ist die Angabe des Modifiers `abstract` an dieser Stelle nicht nötig):

Eben.java

```
1  /**
2   * Interface für geometrische Formen in der Ebene
3   */
4
5  public interface Eben {
6    /* prüft, ob ein Punkt im Objekt liegt */
7    public boolean enthält(double x, double y);
```

```
  /* Geben die X- und Y-Koordinate des
     Mittelpunktes zurück */
  public double getX();
  public double getY();

  /* Verschiebt ein Objekt */
  public void setMittelpunkt(double x, double y);
}

```

Jetzt sehen wir uns die Variante von EbenerKreis an, die auf der Klasse `Kreis` und dem Interface `Eben` basiert:

FEbenerKreis.java

```
/**
 * Beispiel für eine Klasse, die ein Interface
 * implementiert
 */

public class FEbenerKreis extends FKreis
            implements Eben {
  private double mx,my;

  public FEbenerKreis(double r, double x, double y){
    super(r);
    mx = x;
    my = y;
  }

  public double getX() {return mx;}
  public double getY() {return my;}

  public void setMittelpunkt(double x,double y) {
    mx = x;
    my = y;
  }

  public boolean enthält(double x, double y) {
    double dx = mx - x;
    double dy = my - y;
    return dx*dx+dy*dy <= r*r;
  }
}
```

Kapitel 2

Elementare Datenstrukturen

In diesem Kapitel behandeln wir allgemein, wie bestimmte Datenstrukturen funktionieren, mehr oder weniger unabhängig von der Programmiersprache Java. Der hier angegeben Code ist eine Art Pseudocode – er läßt sich mit wahrscheinlich keinem einzigen Compiler übersetzen, aber jeder, der ihn liest und ein bisschen Ahnung von Programmieren hat, sollte ihn verstehen. Das gute am Pseudocode ist, dass man sich darin nur Menschen gegenüber verständlich machen muss, die häufig selber etwas besser mitdenken als ein Computer, daher hat der Pseudocode hier keine so besonders festen Regeln. Wichtig ist eigentlich nur, dass hier (in diesem Skript und an der Tafel bei Herrn Felsner) keine geschweiften Klammern die Blöcke kennzeichnen, sondern *nur* die Einrückung.

2.1 Der Stack

2.1.1 Was ist ein Stack?

Ein Stack verwaltet eine geordmete Menge von Objekten, wobei man ein Element anfügen kann, oder das letzte Element wieder wegnehmen. Beispiele für Stack (auf Deutsch Stapel oder Keller) sind:

Der PEZ-Spender Wer kennt ihn nicht aus der Kindheit? Dieses Plastikteil, wo man einen Stapel süße Brausetabletten reintut, und wenn man den Kopf nach hinten klappt, kommt eine Tabelette raus. Wenn der PEZ-Spender leer ist, tut man wieder einen neuen Stapel hinein.

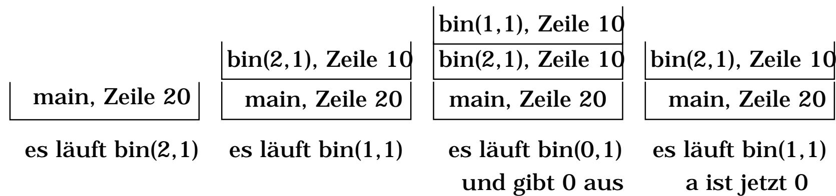
Back-Knopf in Netscape Beim Eingeben einer URL oder dem Anklicken eines Links wird ein Element auf den Stack gelegt, durch Back wird es wieder heruntergenommen und ausgewertet, und wenn man in der Menuleiste Go anklickt, wird der gesamte Inhalt des Stacks auf dem Bildschirm dargestellt

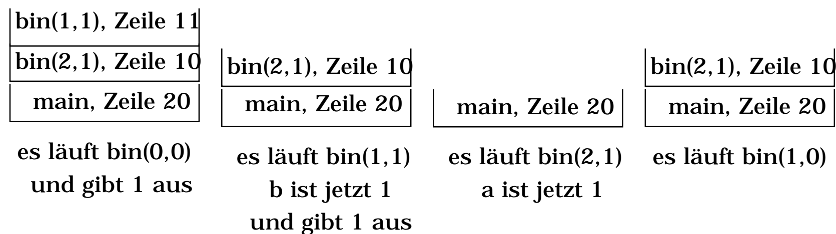
Rekursive Programmierung Bei der Rekursiven Programmierung wird immer, wenn eine Funktion aufgerufen wird, die Position gespeichert, wo der Aufruf stand. Wenn eine Funktion zu Ende augeführt wurde, wird

nachgeguckt, von wo der letzte Aufruf stattgefunden hat, und an der Stelle weitergemacht, und die Positionsangabe wird wieder entfernt. Schauen wir uns dazu einmal eine recht ineffiziente Methode zum Berechnen von $\binom{n}{k}$ an:

```
1  public int binomial(int n, int k)
2  {
3    if(n == 0)
4      if(k == 0)
5        return 1;
6      else
7        return 0;
8    else
9    {
10     int a = binomial(n-1,k)
11     int b = binomial(n-1,k-1);
12     return a+b;
13   }
14 }
```

Dann enthält der Funktionsstack am Anfang zum Beispiel nur die Information „main, Zeile 20“. Wenn man jetzt nachvollzieht, wie die Funktion sich dauernd selbst aufruft, kommt man zu etwa folgendem Verlauf:





2.1.2 Methoden des Stacks

Was ein Stack im allgemeinen tut, habe ich oben schon erläutert. Der Zugriff erfolgt üblicherweise über folgende Methoden:

`push(`*`Objekt`*`)`	fügt *`Objekt`* als oberstes Element dem Stack hinzu
`Objekt pop()`	gibt das oberste Element des Stacks aus und entfernt es
`bool isEmpty()`	prüft, ob der Stack leer ist

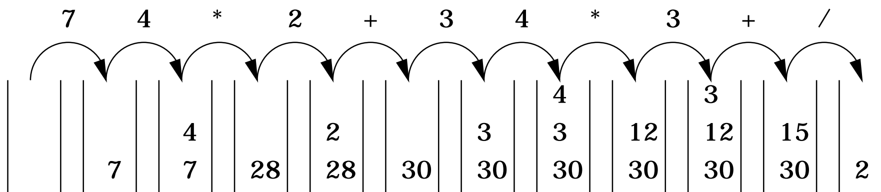
2.1.3 Was macht man mit einem Stack?

Man kann einen Stack beispielsweise dazu verwenden, einen Rechenausdruck in umgekehrter polnischer Notation (UPN) auszuwerten. In UPN wird nicht `a + b` sondern `a b +` geschrieben. Wenn man sich das an einigen Audrücken verdeutlicht, merkt man, dass in UPN keine Klammern benötigt werden, und eine Regel wie Punkt-vor-Strich ebenfalls nicht.

Normal	`3 + 4`
UPN	`3 4 +`
Normal	`(3 * 2) + 4`
UPN	`3 2 * 4 +`
Normal	`3 * (2 + 4)`
UPN	`3 2 4 + *`

Um so einen Ausdruck auszuwerten eignet sich ein Stack prima, auf den die Zahlen gepusht werden, und, sobald man einen Operator findet, die beiden obersten Zahlen poppt, miteinander verknüpft und das Ergebnis wieder pusht. Ich gebe hier ein Beipiel:

Rechenausdruck: 7 4 * 2 + 3 4 * 3 + / (Das ist (7*4+2)/(3*4+3)



Jetzt entwerfen wir einen Algorithmus, der das tut, was in der Grafik angedeutet ist: Die Eingabe ist ein Ausdruck $E = (e_1, e_2, \ldots, e_n)$, die Ausgabe der Wert des Ausdrucks E. Es wird davon ausgegangen, dass E ein korrekter Ausdruck ist.

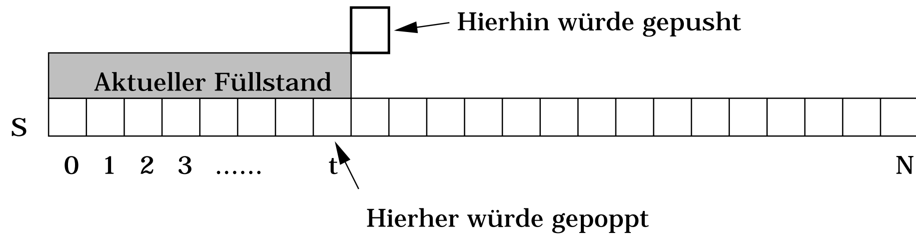
UPN-Auswertung

```
evaluate(E)
  for(i = 0;i <= n;i++)
    if(ei is number)
      push(ei)
    else
      a = pop()
      b = pop()
      push (b ei a)  // ei ist Operator
  return pop
```

2.1.4 Wie realisiert man Stacks?

Stack im Array

Idee: Wir speichern die einzelnen Stackelemente in einem Array, und merken uns, wie viele Objekte zur Zeit in dem Array sind. Dann ist das aktive Ende des Stacks definiert. Das untere Ende ist einfach der Arrayanfang.

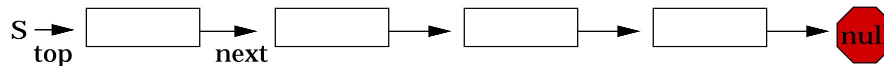


`S`: das Array selbst
`N`: die Anzahl der Elemente, die maximal in das Array passen
`t`: Index im Array, bei dem das letzte Element steht (-1 falls leer)

Entwurf für die Methoden

Stack im Array
```
isEmpty()
  return t == -1

push(Obj)
  if(t == N)
    throw StackFullException
  t++
  S[t] = Obj

pop(Obj)
  if(t == -1)
    throw StackEmptyException
  t--
  return S[t+1]
```

Stack über verkettete Listen



Wir verwenden eine verkettete Liste, wobei jedes Listenelement ein Objekt auf dem Stack darstellt. Wenn man ein neues Objekt einfügen will, erzeugt man ein neues Element, worauf die `top`-Referenz dann zeigt, und setzt deren `next`-Referenzu auf die alte Stackspitze. Die Listenelemente haben ihrerseits die Methoden `Objekt getData()`, `setData(`*`Obj`*`)`, `ListElem getNext()` und `setNext(`*`ListElem`*`)`. Ein Entwurf der Methoden sieht zum Beispiel so aus:

Liste als Stack
```
Klasse Stack:
  private ListElem top

  Konstruktor:
    top = null

  isEmpty()
    return (top == null)

  push(Obj)
```

```
    ListElem newtop = new ListElem
    newtop.setData(Obj)
    newtop.setNext(top)
    top = newtop

  pop()
    LE OldTop = top
    top = top.getNext()
    return OldTop.getData()
```

Vergleich der beiden Lösungen

Wir vergleichen die Laufzeit, die für die verschiedenen Methoden benötigt wird. Dazu zählt man die Elementaroperationen (was das genau ist, steht in gewissem Rahmen frei), die man braucht, um ein Element zu pushen, zu poppen oder zu prüfen, ob der Stack leer ist. In diesem Fall zählen wir einfach die Programmzeilen als Operationen, was so lange auch recht gut klappt, wie keine komplexen Unterfunktionen aufgerufen werden.

	`isEmpty`	`push`	`pop`
Array	1	3	3
Liste	1	4	3

An der Tabelle sieht man, dass es sich um effiziente Implementierungen handelt, denn die Zahl der Elementaroperationen ist klein und konstant.

Außer der Laufzeit kann man (das wurde in der Vorlesung aber nicht gemacht) den Speicherverbrauch der Lösungen vergleichen. Dabei stellt man fest, dass die Variante mit dem Array pro möglichem Objekt auf dem Stack genau eine Referenz speichern muss, nämlich den Arrayeintrag. Da die Größe des Stacks bei der Arraylösung fest ist, und man einen Überlauf vermeiden will, wird man eher für mehr Objekte Platz freihalten, als wirklich gebraucht ist. Das ist natürlich nicht optimal.

In der zweiten Lösung gibt es (außer der Speichergröße) keine Grenze für die Größe des Stacks. Es werden daher auch keine ungenutzten Speicherbereiche belegt, die nur darauf warten, irgendwann vielleicht einmal eine Referenz aufzunehmen. Da erscheint diese Lösung also besser. Allerdings muss für jedes *vorhandene* Objekt auf dem Stack nicht nur eine Referenz auf das Objekt, sondern auch eine Referenz auf das nächste Element gespeichert werden, wobei von dem Speicheaufwand, der einfach schon dadurch entsteht, dass viel mehr Objekte (zu jedem Datenobjekt ein eigenes ListElem), schon abgesehen wurde. Es werden also doppelt so viel Referenzen wie bei der Arraylösung angelegt, also verbraucht ein Listenstack bereits so viel Speicher wie ein nur halb gefüllter Arraystack. Auch das ist nicht optimal, aber man kann sowieso nicht Flexibilität, Speicherverbrauch und Laufzeit gleichzeitig optimal bekommen.

Dynamische Arrays

In `java.util.Stack` ist bereits ein Stack für Java mitgeliefert. In diesem Stack wird ein Array, wie in der ersten Methode verwendet, allerdings erlaubt sich der Computer, wenn ihm das Array viel zu groß oder zu klein vorkommt, sich ein

neues Array zu holen, die Daten zu kopieren, und das alte Array wegzuschmeissen. Dazu gibt es drei Regeln:

- Beginne mit einem Array der Größe N, wobei N vernünftig gewählt werden sollte.
- Wenn ein Objekt gepusht wird, und das Array voll ist, verdoppele die Größe des Arrays.
- Wenn im Array nur noch ein Viertel der Plätze belegt sind, und das Array noch mehr als N Elemente hat, halbiere die Größe des Arrays

Im Gegensatz zu den ersten beiden Lösungen kann hier eine push- oder pop-Operation unter Umständen beliebig lange dauern, da der Stack beliebig groß werden kann, und diese beliebige Größe irgendwann einmal wieder kopiert werden muss. Allerdings stellt man fest, das nach der Veränderung der Größe des Stacks man garantiert eine gewisse Zeit lang Ruhe davon haben wird. Angenommen, der Stack ist gerade auf die Größe M verändert worden. Dann ist er halb voll, hat also $\frac{M}{2}$ Elemente. Die nächste Änderung tritt ein, wenn er entweder nur noch $\frac{M}{4}$ Elemente hat, oder auf M Elemente gefüllt wurde. Dazu werden im ersten Fall mindestens $\frac{M}{4}$ Pops benötigt, im zweiten Fall $\frac{M}{2}$ Pushes.

Annahme: Das Vergrößern des Stacks auf die Größe M benötigt M Elementarperationen, dann können wir diese M Elementarperationen auf die nächsten $\frac{M}{4}$ Stackoperationen umlegen, die dann eben vier Elemntarperationen länger brauchen. Dadurch erreicht man auch hier eine Statistik, bei der push und pop recht schnell gehen, also ist auch diese Variante eine laufzeiteffiziente Implementierung.

Was die Speichereffizienz betrifft ist diese Variante ebenfalls gut, denn sie schmeißt weder mit Mini-Objekten um sich, wie die Listenvariante, noch muss man für Stacks, die manchmal groß werden, die ganze Zeit über viel Speicher opfern.

2.2 Queue

2.2.1 Watt'n dat?

Eine weitere wichtige Datenstruktur ist die Queue (auf Deutsch: Warteschlange). Im Gegensatz zum Stack, wo das Objekt, was als letztes darauf gelegt wurde, als erstes herunter genommen wird, kommt aus einer Queue das Objekt zuerst heraus, was auch als erstes hineingesteckt wurde. Daher werden Queues auch als FIFOs bezeichnet (first in – first out), Stacks dagegen als LIFOs. Im täglichen Leben begegnet einem eine Queue überall dort, wo man sich anstellen muss[1]. Im Computer wird zum Beispiel für die Tastatureingaben eine Queue verwendet. Jeder Tastendruck wird in der Queue gespeichert, und wenn der Computer Zeit hat, holt er die Daten aus der Queue ab, und reagiert auf die Eingabe.

[1] Denkste! Überall, wo man sich anstellen muss, drängeln sich Leute vor. In einer einfachen Queue ist das unmöglich...

2.2.2 Methoden der Queue

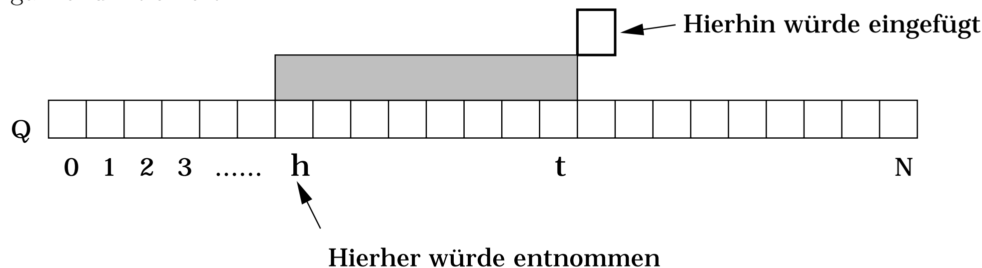
Wie der Stack, hat auch die Queue drei Methoden - eine um Elemente einzufügen, eine, um Elemente abzuholen, und die Überprüfung, ob die Queue überhaupt Daten enthält.

`enqueue(`*`Objekt`*`)`	fügt *`Objekt`* als hinterstes Element der Queue an
`Objekt dequeue()`	gibt das vorderste Element der Queue aus und entfernt es
`bool isEmpty()`	prüft, ob die Queue leer ist

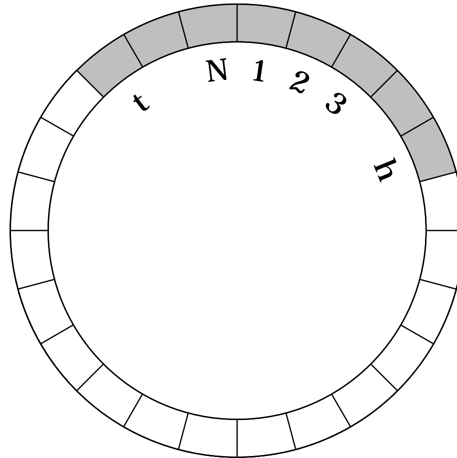
2.2.3 Queue im Array

Wenn man einen Stack im Array implementieren kann, sollte das auch mit einer Queue funktionieren. Wie wir schon beim Stack gelernt haben, speichert man sich einen Index im Array, der das letzte in die Queue gestellte Element angibt. Diesen Index nennen wir `tail`, oder kurz `t`, da er das hintere Ende der Queue bezeichnet. Das erste Element steht am Anfang auf Position 0. Wenn man das Element abgeholt hat, ist entweder vorne ein Loch und das erste Element steht nicht mehr an Position 0, oder man schiebt alle Elemente um eins nach vorne.

Der zweite Ansatz mag einfacher zu programmieren zu sein, aber er ist sicher nicht effizient, da im schlimmsten Fall bei jedem dequeue-Aufruf fast der ganze Arrayinhalt um eine Position verschoben werden muss (beim abwechselnden Einfügen und Entfernen aus einer fast vollen Queue). Also speichert man auch noch, wo das vorderste Element steht in einem weiteren Index, den man `head` oder einfach `h` nennt. Damit haben wir bereits eine vernünftige Idee, wie das ganze funktioniert.



Allerdings kann es passieren, dass irgendwann `head` und `tail` ziemlich am Ende des Arrays stehen. Dann ist es natürlich ungünstig, wenn man nichts mehr einfügen kann, obwohl am Anfang noch so viel Platz ist. Daher macht man das lineare Array einfach zum Ring, in dem hinter dem N-ten Element wieder das erste kommt, und die Queue kann dann durchaus hinten anfangen, über das Ende hinaus gehen, und vorne enden. Da es etwas kompliziet wird, zu zählen, wie viele Objekte in der Queue sind, führen wir einfach noch `n` ein, was angibt, wie viele es sind.



Jetzt brauchen wir nur noch ein paar Zeilen Pseudocode:

Queue im Array

```
Klasse Queue:
  private int h, t, n
  private Object[] Q

  Konstruktor:
    h=0
    t=0
    n=0

  isEmpty()
    return (n==0)

  enqueue(Obj)
    if(n < N)
      n = n + 1
      if(t < N - 1)
        t = t+1
      else
        t = 0
      Q[t] = Obj
    else
      throw QueueFullException

  dequeue()
    if(n > 0)
      n = n - 1
      if(h < N - 1)
        t = t+1
        return Q[t-1]
      else
        t = 0
        return Q[N]
```

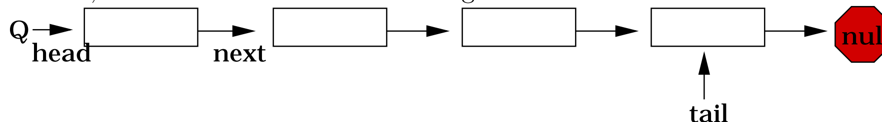
```
    else
      throw QueueEmptyException
```

Queue in einer Liste

Wir konnten beim Stack mit einer einfach verketteten Liste arbeiten, also versuchen wir das auch mit der Queue. Dazu muss man natürlich wissen, wo welches Ende ist, also ob die Referenzen vom Kopf zum Schwanz oder umgedreht zeigen. Angenommen, die Referenzen gingen vom Schwanz zum Kopf. Dann zeigt das Element, was als erstes entfernt werden muss, auf `null`. Wenn man dieses Element entfernt, muss das Element, was vorher auf dieses Element verwiesen hat, jetzt auf `null` zeigen.

Dabei gibt es ein Problem: *Wie* gelangen wir an das zweite Element der Queue? Wir wissen, wo die Referenzenkette anfängt, also das letzte Element, wir kennen das Anfangselement, aber um das Element dahinter zu finden, bleibt uns nichts übrig, als die Queue zu durchsuchen. Das ist ineffizient. Daher probieren wir es andersherum.

Die Referenzen zeigen also vom Kopf zum Schwanz. Dann verweist das letzte Element der Queue auf `null`. Dahinter ein weiteres Element einzufügen ist einfach, denn das letzte Element haben wir schon, und müssen in diesem Element nur die Referenz ändern. Das entfernen am Anfang der Zeigerkette ist ebenfalls einfach, das haben wir schon im Stack gemacht.



Bis jetzt sind wir davon ausgegangen, dass wir `tail` kennen. Das ist typisch mathematisch[2], denn wir gehen von irgendetwas aus, ohne es zu vorher gezeigt zu haben, indem wir es einfach behaupten. Wie findet man den jetzt das Ende? Man kann vom `head` aus einfach allen Zeigern folgen, und irgendwann ist man da:

```
getTail()
  n = head
  while(n.getNext() != null)
    b = n.getNext()
  return n
```

Darauf basierend schreibt man dann die einfachen Methoden, die einem bis auf kleine Änderungen schon vom Stack bekannt sein sollten:

```
Queue in Liste
Klasse Queue:
  private ListElem head;

  Konstruktor:
    head = null

  isEmpty()
```

[2]So hat sich Herr Felsner ausgedrückt

```
    return (head == null)

  enqueue(Obj)
    ListElem t = new ListElem
    t.setData(Obj)
    t.setNext(null);
    getTail().setNext(top)

  dequeue()
    ListElem OldHead = head;
    head = head.getNext();
    return OldHead.getData();
```

Diese Lösung scheint effizient zu sein, da alle Methoden kurz sind. Allerdings ist sie das *nicht*, da die Methode `getTail()` eine Schleife enthält, die sooft durchlaufen wird, wie es Elemente in der Queue gibt, und `getTail()` bei jedem Einfügevorgang aufgerufen wird. Im günstigsten Falle braucht man pro Durchlauf eine Elementaroperation, und bereit dann benötigt man für 100 Elemente etwa 5000 Operationen, also 50 Operationen pro `enqueue`, wenn man es gleichverteilt rechnet. Will man 1000 Elemente in die Queue packen, dauert das bereits 500000 Operationen, pro Element also sogar 500. Das ist nicht effizient.

Daher speichert man sich `tail` als Instanzvariable, und setzt diese Variable in `enqueue` immer auf den neuesten Wert. Die Implementation sei dem Hörer[3] zur Übung überlassen.

2.2.4 Die Variante Dequeue

Dequeue ist ein Kurzwort für „double ended queue“. Eine Dequeue kann sowohl am Anfang, wie am Ende, Daten aufnehmen oder abgeben. Damit sind die einfache Queue, und der Stack beides Spezialfälle der Dequeue. Um eine Dequeue mit Listen effizient zu implementieren, braucht man doppelt verkettete Listen, da wie oben zu zeigen versucht wurde, das Entfernen am Ende der Referenzkette nicht effizient möglich ist.

[3] in diesem Falle eher Leser

Kapitel 3

Algorithmen

Nachdem wir wissen, wie man programmiert, und außerdem ein wenig über einfache Datenstrukturen gesprochen haben, wird es Zeit, dieses Wissen anzuwenden, indem man Algorithmen entwickelt. Ein Algorithmus ist eine Vorschrift, was mit den eingegebenen Daten zu tun ist. Je mehr Schritte der Algorithmus braucht, um aus den eingegebenen Daten die Ausgabe zu berechnen, um so mehr Zeit benötigt die Anwendung, und genau darauf wollen wir jetz achten.

3.1 Laufzeitanalyse

Sicherlich ist ein Algorithmus im allgemeinen schneller, wenn es weniger Daten gibt, die zu verarbeiten sind, und braucht länger, wenn man große Datenmengen verarbeiten will. Außerdem hängt die Laufzeit bei einigen Algorithmen zusätzlich von den Daten selbst ab. Daher unterscheidet man den günstigsten Fall („best case"), die durchschnittliche Dauer („average case") und die Zeit, die bei den denkbar ungünstigsten Daten auftritt („worst case"). Der Idealfall interessiert uns nicht, da man jeden Algorithmus einfach erweitern kann, daß er zuerst die Eingabe mit einer Vorgabe vergleicht, und dann ein vorberechnetes Ergebnis ausspuckt. Die durschnittliche Laufzeit ist dagegen sehr interessant, aber nur mit der Wahrscheinlichkeitsrechnung aus der höheren Mathematik zu berechnen. Der schlechteste Fall ist normalerweise nicht so interessant, aber macht eine definitive Aussage: „Länger als so lange *kann* es nicht dauern". Wenn dieser Algorithmus zwischen Betätigung des Bremspedals und dem Einsetzen der Bremswirkung ablaufen muss, ist es sogar sehr wichtig, zu garantieren, dass er schnell genug fertig ist.

3.1.1 Die O-Notation

Um die Laufzeit zu beschreiben, die ein Algorithmus braucht, gibt man keine Formel an, die berechnet, wie viele Elementaroperationen ein Algorithmus braucht, wenn die genauen Eingabedaten gegeben sind, denn diese Formel wird sehr kompliziert, und man sieht ihr auch nicht direkt an, was zum Beispiel passiert, wenn man doppelt so viele Daten hereinsteckt.

Etwas sinnvoller wäre es schon, eine Formel anzugeben, die aus aus der Größe der Eingabedaten berechnet, wie viele Operationen gebraucht werden. Aller-

dings ist die Anzahl der Operationen als solche auch nicht sehr aussagekräftig, weil schlechte Compiler häufig für den gleichen Quelltext häufig Code erstellen, der mehr Zeit braucht, als der Code, den ein guter Compiler erstellt.

Richtig interessant, und trotzdem nicht zu kompliziert ist es, eine Funktion zu beschreiben, die genauso stark (oder stärker) steigt, als die Laufzeit des Algorithmus. Mathematisch exakt macht man es mit der O-Notation. Unter $O(g)$ versteht man eine Menge von Funktionen, die nicht stärker steigt als $g(n)$. Die mathematische Definition sieht so aus:

$$f \in O(g) \Longleftrightarrow \exists c > 0 : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : f(n) \leq cg(n)$$

Das bedeutet, dass ab einem (endlichen) n_0 die Funktion f innerhalb bestimmter Grenzen bleibt. Am Anfang darf die Funktion f machen, was immer sie will. Und es muss nicht so sein, dass $f(n)$ kleiner als $g(n)$ ist, sondern es reicht, dass ein (endliches) Vielfaches, nämlich das c-fache von $g(n)$ stets über $f(n)$ liegt. Das c muss aber unabhängig von n festgelegt werden können.

Es ist also $f(n) = n^2$ nicht in $O(n)$ drin, da jedes Vielfache von n, also cn kleiner als n^2 wird, wenn nur n groß genug ist, in diesem Falle größer als c. Allerdings ist $f(n) = 3n + 300$ sehr wohl in $O(n)$, da man mit $c = 303$ bereits ab $n_0 = 1$ bei $f(n)$ einen Wert erhält, der nicht größer als $cg(n)$ ist. Man könnte auch $c = 4$ und $n_0 = 300$ wählen. Es reicht ein einziges Paar von c und n_0, mit dem f und g die Gleichung erfüllen, damit $f \in O(g)$ gilt.

In der Informatik steht n bei der Laufzeit normalerweise für die Anzahl der Elemente in der Menge von Eingabedaten. Als Beispiel berechnen wir die Funktionsklasse, in der die Laufzeit des Algorithmus `MaxArray` liegt, der die größte unter n Zahlen ermittelt.

MaxArray

```
MaxArray(A):
  max=A[0]
  for(i = 1;i < n; i++)
    if(max < A[i])
      max = A[i]
  return max
```

Die erste und letzte Zeile werden je einmal ausgeführt, und sind simpel, können also mit einer Elementaroperation berechnet werden. Die Schleife in der Mitte wird aber $n-1$ mal durchlaufen, und braucht dafür in diesem Fall wieder ein paar (ob jetzt 2, 3 oder 4 ist Ansichtssache, jedenfalls klein und konstant) Elementaroperationen.

Allgemein gilt für einen solchen Algorithmus, der in einer Schleife pro Durchlauf s Operationen braucht, und t außerhalb, das seine Laufzeit $f(n) = (n-1)s+t$ in $O(n)$ liegt. Man sagt: „Wir haben einen $O(n)$-Algorithmus entworfen“. Wenn die Laufzeit eines Algorithmus konstant ist, redet man von $O(1)$.

Wichtige Funktionen für die Laufzeitabschätzung und ihre Anordnung in der O-Hierarchie sind:

$$1 <_O \log n <_O n <_O n \log n <_0 n^2 <_O \underbrace{n^k}_{k \geq 3} <_O \underbrace{a^n}_{a>1} <_O n!$$

wobei mit $f <_O g$ gemeint ist, dass $f \in O(g)$, aber nicht $g \in O(f)$. Die meisten Funktionen daraus haben einen Namen, den man in der Laufzeitangebe

häufig verwendet, man sagt statt „ein $O(n)$-Algorithmus“ zum Beispiel auch „ein Algorithmus mit linearer Laufzeit“.

$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n^2)$	quadratisch
$O(n^k)$	polynomiell
$O(a^n)$	exponentiell

Es gilt, dass falls $f \in O(h)$ und $g \in O(h)$, auch $(f+g) \in O(h)$ ist, und ebenso, dass aus $f \in O(g)$ und $g \in O(h)$ folgt, dass $f \in O(h)$ ist. Damit kann man Summen von Funktionen vereinfachen:

Beispiel:

$$f(n) = \underbrace{\underbrace{\underbrace{12n^2}_{\in O(n^2)} + \underbrace{6n^3}_{\in O(n^3)}}_{\in O(n^3)} + \underbrace{\underbrace{4n\log n}_{\in O(n\log n)} + \underbrace{2^10}_{\in O(1)}}_{\in O(n \log n)}}_{\in O(n^3)}$$

Daran sieht man die Regel, dass man in Summen alle Summanden bis auf den in der O-Hierarchie am höchsten stehenden vernachlässigen kann.

3.1.2 Beispiel: maximale Teilsumme

Um verschiedene Herangehensweisen an die Implementierung eines Algorithmus zu beschreiben, nehmen wir ein Beispiel: Einen Algorithmus, der einem die größtmögliche Summe von aufeinanderfolgenden Elementen in einem Array `A` mit `n` Elementen liefert. Das macht natürlich nur Sinn, wenn auch negative Zahlen im Array sind, da sonst einfach alle Zahlen zusammen die größtmögliche Summe ergeben.

Berechnen wir sie einfach!

Es liegt ja recht nahe, wenn man die größte Teilsumme haben will, einfach *alle* zu berechnen, und die größte davon zu nehmen. Das ganze kann man auch schnell in einen Algorithmus schreiben:

MaxSubArray arschlahm

```
1  MaxSubArray(A)
2    // Wenn man 0 Elemente addiert, erhält man
3    // als Summe 0. Daher ist 0 in jedem Array
4    // eine mögliche Teilsumme
5    maxsum = 0
6    for(start = 0; start < n; start++)
7      for(stop = start+1; stop < n; stop++)
8        summe = 0
9        for(idx = start; idx < stop; idx++)
10         summe = summe + A[idx]
11       if(summe > maxsum)
12         maxsum = summe
13   return maxsum
```

Dass diese Funktion das gewünschte leistet, ist recht einfach zu verstehen. Allerdings ist sie nicht besonders effizient. Man sollte bei verschachtelten Schleifen sowieso misstrauisch werden. Die äußerste Schleife wird nämlich n-mal durchlaufen, die mittlere Schleife im Schnitt $\frac{n}{2}$-mal pro Durchlauf der äußeren Schleife, also ist die Anzahl der Durchläufe der inneren Schleife bereits in $O(n^2)$. Die Anzahl der Durchläufe der inneren Schleife pro Durchlauf der mittleren Schleife läßt sich auch berechnen, aber ich denke, man glaubt mir, wenn ich behaupte, dass auch das wieder mit n geht. Daher hat die Funktion einen Gesamtaufwand $O(n^3)$. Das ist keine besonders tolle Laufzeit, denn für die zehnfache Größe das Arrays braucht die Funktion bereits tausendmal länger. Das ist einfach schlecht.

3.1.3 Zwischenergebnisse behalten

Daher schaut man sich einmal genauer an, was in der innersten Schleife passiert. Dort werden immer die Arrayelemente von den Indizes `start` bis `stop` aufsummiert. Dabei bleibt `start` lange Zeit gleich, während sich `stop` immer um eins erhöht. Also ist es eigentlich Mist, die Summer jedesmal neu von vorne zu berechnen. Damit erhält man dann folgenden Algorithmus, der nur noch $O(n^2)$ als Laufzeit hat:

MaxSubArray nicht ganz doof

```
1 MaxSubArray(A)
2   maxsum = 0
3   for(start = 0; start < n; start++)
4     summe = 0
5     for(stop = start; stop < n; stop++)
6       summe = summe + A[stop]
7       if(summe > maxsum)
8         maxsum = summe
9   return maxsum
```

3.1.4 Divide & Conquer

Wir können das Problem auch mit einer ganz anderen Herangehensweise lösen, die in der Informatik auch recht verbreitet ist, nämlich das Problem dadurch zu beherrschen (das heißt hier: in vernünftiger Zeit zu lösen), indem man es in Teilprobleme zerhackt, diese getrennt löst, und die Ergebnisse zusammenfügt. In diesem Fall zerteilen wir das Array in der Mitte, und haben dann genau drei Möglichkeiten:

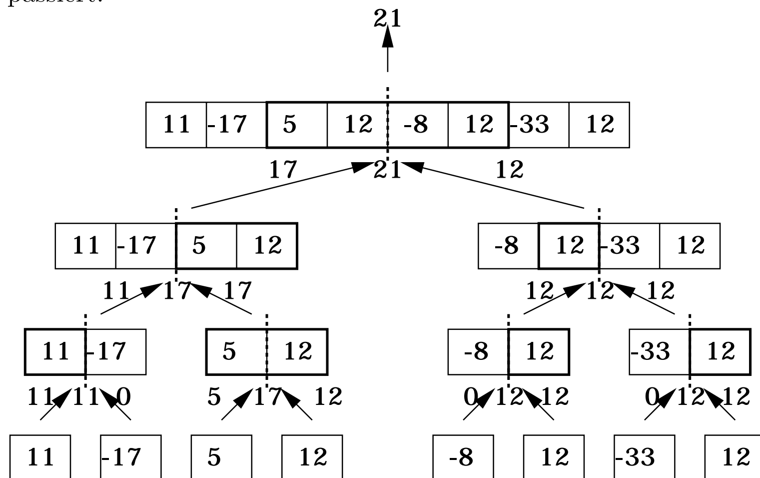
- Die maximale Teilsumme besteht nur aus Elementen des linken Teilarrays
- Die maximale Teilsumme besteht nur aus Elementen des rechten Teilarrays
- Die maximale Teilsumme besteht aus Elementen beider Arrays, ist also die Summe aus maximaler Endsumme des linken, und maximaler Anfangssumme des rechten Arrays.

Der Algorithmus, um die maximale Teilsumme von `A` zwischen den Indizes `l` und `r` zu berechnen, sieht also im Prinzip so aus:

MaxSubArray mit D&C

```
MaxAnfsumme(A,l,r)
  maxsum = 0;
  summe = 0;
  for(i = l;i <= r;i++)
    summe += A[i];
    if(summe > maxsum)
      maxsum = summe
  return maxsum

MaxEndsumme(A,l,r)
  // geht analog

MaxSubArray(A,l,r)
  if(l == r)
    return max(A[l],0);

  m = abrunden((l+r)/2)   // Such Dir die Mitte raus
  maxL = MaxSubArray(A,l,  m)
  maxR = MaxSubArray(A,m+1,r)
  maxM = MaxEndsumme(A,l,  m) +
         MaxAnfsumme(A,m+1,r)
  return max(maxL,maxR,maxM);
```

Man kann sich anschauen, wie die Funktion sich selbst rekursiv aufruft, wenn man mal ein Diagramm zeichnet, in dem zu sehen ist, was mit welchem Teil passiert:



Die Funktionen `MaxEndsumme` und `MaxAnfsumme` sind von der Laufzeit her in $O(n)$, was offensichtlich ist, dann die Schleife hat genau n Durchläufe. Interessanter ist die Frage, welche Laufzeit `MaxSubArray` selbst hat. Für den Fall $n = 1$ ist offensichtlich die Laufzeit in $O(1)$, ansonsten gilt, dass die Laufzeit L das Verhalten $L(n) = 2O(\frac{n}{2}) + 2L(\frac{n}{2})$ hat. Die Zwei kann man in das $O(\frac{n}{2})$ hineinziehen, dann erhält man $L(n) = O(n) + 2L(\frac{n}{2})$. Damit man diese Rekursionsformel lösen kann, setze ich für O(n) einfach cn ein.

Die Lösung kann man entweder raten, ausprobieren, oder sich von einem Computeralgebrasystem wie Maple berechnen lassen. Es kommt die Funktion $T(1)n + cn\log_2(n)$ heraus, die in $O(n \log n)$ liegt. Man kann das $O(n \log n)$ auch aus dem Baum sehen, denn es gibt $\log_2 n$ Ebenen im Baum, auf der jeweils die maximalen End– bzw. Anfangssummen berechnet werden, wobei jedes Element nur ein Mal behandelt wird, also der Aufwand pro Ebene $O(n)$ ist.

Augen auf und durch

Über Arrays kann man häufig noch effizientere Algorithmen implementieren, wenn man einfach in einer Schleife von vorne nach hinten durchläuft. Auch dieses Problem läßt sich auf diese Art lösen, denn ist das Problem für die ersten k Elemente gelöst, dann ist die Lösung für die ersten $k+1$ Elemente entweder:

- Dieselbe
- oder sie ist die maximale Endsumme des neuen Arrays.

Für den ersten Fall braucht man nichts weiter zu tun, als ihn zu erkennen. Im zweiten Fall ist die maximale Endsumme des neuen Arrays gefragt. Diese ist entweder die maximale Endsumme des alten Arrays, zu der das neue Element addiert wird, oder, falls das kleiner 0 wird, einfach 0, da dann das günstigste für die Endsumme ist, über *gar kein* Element zu summieren. Die Endsumme kann man also beim Durchlaufen des Arrays mitbestimmen. Der Algorithmus steht hier nicht, da er in einer Übungsaufgabe gefragt ist, und ich vor der Abgabe der letzten Gruppe hier die Aufgaben nicht weiter erläutern will, als der Professor in der Vorlesung.

3.1.5 Sortieren

Ein ganzes Teilgebiet der Informatik beschäftigt sich mit dem Sortieren von Daten. Das Problem dazu lautet einfach: Gegeben sei eine Folge $(a_1, a_2, a_3, \ldots, a_n)$, deren Elemente einer Totalordnung unterliegen. Dabei heißt Totalordnung, dass zwei Elemente stets vergleichbar sind, und wenn $a_1 < a_2$ als auch $a_2 < a_3$ gilt, immer $a_1 < a_3$ gilt. Bei den reellen, rationalen und natürlichen Zahlen ist das beispielsweise der Fall, ebenso bei Strings, indem man als Vergleichskriterium die Ordnung im Telefonbuch nimmt.

Sortieren lassen sich nicht nur Folgen von Elementen, die auf diese Art vergleichbar sind, sondern jede Folge, deren Elementen man einen Schlüssel (oder Key) zuordnen kann, der aus einer total geordneten Menge stammt, denn dann kann man die Totalordnung dieser Menge auf die Elemente, die zu sortieren sind, übernehmen. Es ist auch nicht so, dass eindeutig festgelegt sein muss, wie eine Folge sortiert wird. So könnte beispielsweise das Telefonbuch auch nach Vornamen oder Adresse sortiert sein, anstatt nach dem Nachnamen, wenn man einfach eine andere Methode zur Erzeugung des Sortierschlüssels nehmen würde.

Insertion-Sort

Ein recht einfacher Algorithmus, der durch die Folge linear vorgeht, basiert auf der Idee, dass ersten k Elemte bereits sortiert seien. Dann nimmt man sich das $k+1$-te Element, und steckt es an die Stelle zwischen die k Elemente, wo es

der Sortierreihenfolge nach hingehört. Dann sind $k + 1$ Elemente sortiert. Der Algorithmus sieht so aus:

Insertion-Sort
```
1  InsertSort(A)
2    // Die Schleife fängt bei 1 an, da die
3    // Folge aus dem ersten Element immer
4    // sortiert ist.
5    for(k = 1; k < n; k++)
6      // Merke dir das neue Element
7      x = A[k]
8      // Gehe von hinten nach vorne durch, bis
9      // Du ein kleineres Element als x findest,
10     // oder am Anfang bist
11     for(i = k-1;i >= 0 && A[i] > k; i++)
12       A[i+1] = A[i];  // Schieb' nach hinten
13
14     A[i+1] = x;  // Setz' das neue Element ein
```

Betrachten wir einmal genauer, was beim Einfügen passiert, wenn zum Beispiel schon 5 Elemente sortiert sind. Zwischen den beiden einfachen waagerechten Linien liegt die innere Schleife.

a_0	a_1	a_2	a_3	a_4	a_5	...	x	i
1	5	7	12	17	3	...	–	–
1	5	7	12	17	3	...	3	4
1	5	7	12	17	17	...	3	3
1	5	7	12	12	17	...	3	2
1	5	7	7	12	17	...	3	1
1	5	5	7	12	17	...	3	0
1	3	5	7	12	17	...	–	0

Wie eine Laufzeitbestimmung ergibt, wird die äußere Schleife n mal durchlaufen, und die innere höchstens $2 + 3 + \cdots + n = \sum_{j=2}^{n} j$ mal. Das bedeutet, dass die Laufzeit, die hauptsächlich von den Kopieroperationen bestimmt ist, in $O(n^2)$ liegt.

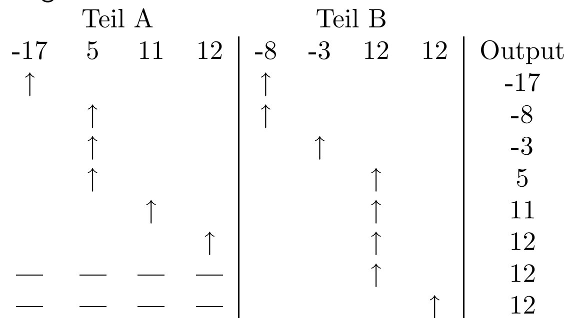
3.1.6 Mergesort

Das Problem der maximalen Teilsumme haben wir mit Divide & Conquer auf $O(n \log n)$ zurechtstutzen können. Das soll uns beim Sortieren auch gelingen. Also setzen wir genauso an: Um ein Array zu sortieren, spalten wir es in zwei Teile auf, sortieren die beiden einzelnen Teile, und fügen danach die bereits sortierten Teile wieder zusammen. Das zusammenfügen der sortierten Teile wird ium Englischen „merge“ genannt, daher kommt der Name für diesen Algorithmus. Er sieht etwa so:

mergesort
```
1  mergeSort(A,l,r)
2    // 1 Element ist bereits sortiert
3    if(l == r)
4      return;
5    m = abrunden(l+r/2)
6    mergeSort(A,l,m);
```

```
7   mergeSort(A,m+1,r);
8   merge(A,l,m,r);
```

Den Algorithmus zu `merge` darf ich leider hier ebenfalls nicht hinschreiben, da noch nicht alle Übungsgruppen den Zettel abgegeben haben, wo das gefragt ist. Allerdings wurde erläutert, wie `merge` funktioniert: Man nehme die beiden Arrays, die zu vermischen sind, und schaue sich das erste Element darin an. Das kleinere von den beiden ist das erste Element der Mischung, und wird aus dem Array entfernt, wo es her kam. Jetzt machen wir das ganze so lange weiter, bis alle Elemente in der Ausgabe liegen. In der Praxis löscht man die Elemente allerdings nicht, sondern springt einfach mit dem Index eins weiter, weil das Löschen in einem Array eine sehr aufwendige Operation ist. Eine Verlaufsskizze für `merge` sieht etwa so aus:



Teil A				Teil B				
-17	5	11	12	-8	-3	12	12	Output
↑				↑				-17
	↑			↑				-8
	↑				↑			-3
	↑					↑		5
		↑				↑		11
			↑			↑		12
—	—	—	—			↑		12
—	—	—	—				↑	12

Die Laufzeit entsteht zum einen aus dem `merge`, was in $O(n)$ läuft, und dem Aufwand, den den untergeordnenten `mergeSort`-Aufrufe verbrauchen. Das läuft wieder auf $T(n) = 2T(\frac{n}{2}) + cn$ hinaus, was wir schon bei `MaxSubArray` hatten. Die Laufzeit davon ist (immernoch) $O(n \log n)$, was deutlich besser als $O(n^2)$ ist.

3.2 Bäume

Brr! Die Vorlesung ist leider nicht so schön in Kapitel gliederbar. Eigentlich gehören Bäume nämlich unter Datenstrukturen und nicht unter Algorithmen einsortiert, aber da kann ich jetzt auch nichts gegen machen.

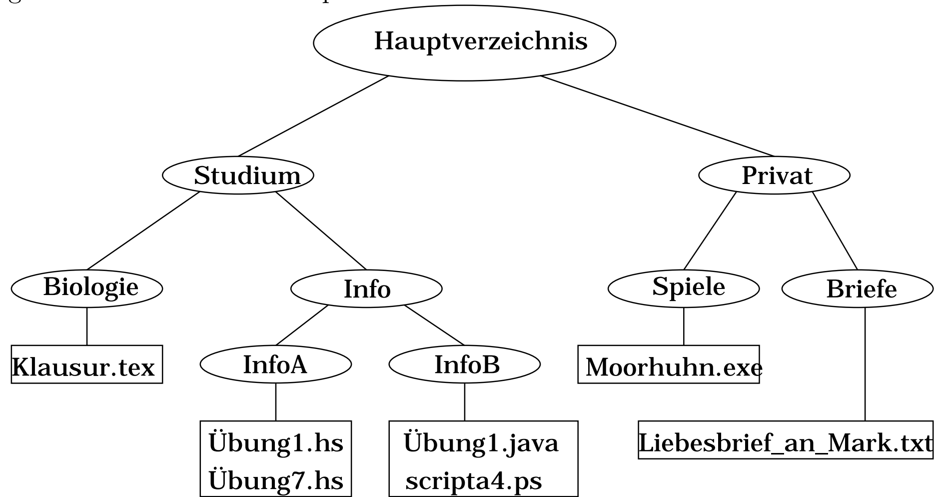
3.2.1 Wozu das ganze?

Man sehe sich zum Beispiel eine Festplatte an. Dort könnte man die Dateien natürlich in einer Liste speichern, die dann etwa so aussieht:

```
Bioklausur.doc
Liebesbrief_an_Mark.txt
Moorhuhn.exe
Uebung1_InfA.hs
Uebung1_InfB.java
Uebung7_InfA.hs
```

Wie man leicht sieht, ergibt sich dabei früher oder später ein ziemliches Durcheinander. Daher wird das Dateisystem in Verzeichnisse gegliedert, die sel-

ber wieder Dateien oder Verzeichnisse enthalten können, so dass die oben angegebene List dann zum Beispiel so aussieht:

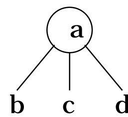


3.2.2 Defintion eines Baumes

Eine Menge B von Knoten heißt Baum, wenn sie

1. einen besonderen Knoten $R \in B$ enthält, der Wurzel heißt
2. Jeder Knoten aus B auuser der Wurzel R genau einen Vater hat, die Wurzel aber keinen
3. Die Wurzel Vorfahre von jedem Knoten ist

Dabei sind die Vorfahren eines Knotens der Knoten selber und alle Vorfahren seines Vaters. Als Beispiel für einen einfachen Baum gebe ich die Menge $B = a, b, c, d$ an, mit der Beziehung Wurzel $= a, \text{Vater}(b) = a, \text{Vater}(c) = a, \text{Vater}(d) = a$. Als Graph sieht das dann so aus:



In einem Baum benutzt man folgende Bezeichnungen (wobei $u, v \in B$):

- u heißt **Kind** von v, genau dann, wenn v Vater von u ist.
- u und v heißen **benachbart**, wenn sie den gleichen Vater haben
- u heißt **Nachfahre** von v, genau dann, wenn v Vorfahre von u ist.
- Knoten ohne Kindern heissen **äußere Knoten** oder **Blätter**
- Knoten mit Kinder heissen **innere Knoten**

geordnete Bäume

In geordneten Bäumen kann man vom ersten, zweiten, dritten, ... Kind eines Knotens sprechen. In ungeordneten Bäumen sind dagegen alle Kinder gleichwertig, es gibt dort, wenn ein Knoten mehrere Kinder hat, kein Kind, das man „erstes Kind“ nennen kann.

Teilbaum

Eine Menge $\overline{B} \in B$ heißt Teilbaum, wenn sie selber ein Baum ist, und alle Nachfahren, die dessen Wurzel in B hat, ebenfalls enthält.

3.2.3 Methoden des Datentyps Baum

Wichtig: Um in diesem Skript die Konsistenz zu wahren, formuliere ich hier einiges um, damit es in das objektorientierte, javaartige Modell von Herrn Felsner passt. Dass die Sachen hier an manchen Stellen ganz anders aussehen heißt also nicht, dass alles falsch war, was wir in der Vorlesung gehört haben, sondern nur, dass die Vertretung ein anderes Konzept hatte

Wenn man einen Baum objektorientiert programmiert, führt man ein Objekt `Knoten` ein, was einen Knoten des Baumes (also innerer Knoten oder Blatt) darstellt, und eine Objekt `Baum`. Das entspricht dem Vorgehen bei den Listen, bei denen wir `DoubLiList` und `ListElement` eingeführt hatten.

Zuerst betrachte ich die Methoden eines Knotens:

`vater()`	gibt den Vater des Knotens zurück
`bool` `istWurzel()`	prüft, ob der Knoten die Wurzel des Baumes ist
`Knoten[]` `kinder()`	gibt die Menge der Kinder des Knotens zurück. In ungeordneten Bäumen in beliebiger Reihenfolge, sonst vom ersten bis zum letzten.
`bool` `istBlatt()`	prüft, ob der Knoten ein Blatt des Baumes ist

Ein Baum braucht eingentlich nur eine einzige Methode, nämlich `wurzel`, die die Wurzel des Baumes zurückgibt.

Die Methoden lassen sich alle in $O(1)$ implementieren, abgesehen von der Methode `Kinder`, da dort das Array im allgemeinen erstellt oder kopiert wird, was $O(c_v)$ benötigt, wobei c_v die Kinder des Knotens v bezeichnet. Man kann auch Bäume so implementieren, dass auch diese Methode in $O(1)$ funktioniert, aber das ist keine Eigenschaft der Datenstruktur an sich.

3.2.4 Grundlegende Algorithmen auf Bäume

Unter der **Tiefe** eines Knotens versteht man den Abstand zur Wurzel, was man auch als die Anzahl der Vorfahren dieses Knotens ohne sich selber auffassen kann. Mathematisch ausgedruckt sieht das so aus:

$$\text{Tiefe}(v) = \begin{cases} 0 & \text{falls } v \text{ die Wurzel ist} \\ 1 + \text{Tiefe}(\text{Vater}(v)) & \text{sonst} \end{cases}$$

In dem Pseudocode, wie Herr Felsner ihn schreibt, sieht es dann so aus:

```
Tiefenberechnung
Tiefe(v)
  if(v.istWurzel())
    return 0;
  else
    return 1+Tiefe(v.vater())
```

Die Laufzeit dieses Algorithmus ist offensichtlich: Er braucht genau so lange, wie der Knoten tief liegt. Also ist der Aufwand $O(\text{Tiefe}(v))$.

Die **Höhe** eines Knoten gibt an, wie viele Generationen noch unterhalb dieses Knotens liegen. Unter der Höhe eines Baumes versteht man die Höhe der Wurzel, die natürlich die meisten Nachfahren hat.

$$\text{Höhe}(v) = \begin{cases} 0 & \text{falls } v \text{ Blatt ist} \\ 1 + \max_{u \in v.\text{kinder}()} \text{Tiefe}(u) & \text{sonst} \end{cases}$$

$$\text{Höhe}(B) = \text{Höhe}(B.\text{wurzel}()) = \max_{v \in B} \text{Tiefe}(v)$$

Falls man die Höhe eines Baumes tatsächlich dadurch berechnet, erhält man eine Laufzeit von $O(n^2)$, da für n Knoten die Tiefe berechnet werden muss, und im blödesten Fall alle Knoten untereinander hängen, und daher die Tiefenberechnung auch $O(n)$ ist. Im Durchschnitt kommt man aber etwas besser weg. Wenn man sich aber überlegt, was dabei geschieht, kann man sich eigentlich nur vor den Kopf schlagen, denn von jedem Teilbaum, der nicht direkt an der Wurzel hängt, wird dann die Höhe gleich mehrmals berechnet. Besser ist es, für die Höhenberechnung wirklich nur die Kinder der Wurzel zu nehmen, und die Höhe der anderen Nachfahren nicht noch einmal extra zu berechnen. Dann erhält man folgenen Algorithmus

```
Höhenberechnung
Höhe(v)
  if(v.istBlatt())
    return 0
  else
    maxh = 0
    Knoten[] k = kinder()
    for(i = 0;i < k.length;i++)
      h = Höhe(k[i])
      if(h > maxh)
        maxh = h
    return maxh+1
```

Dort wird, wenn man es sich genauer überlegt, die `for`-Schleife für jedes Kind des Knotens einmal durchlaufen. Da es auch für die Kinder der Kinder gilt, wird die `for`-Schleife – zählt man die rekursiven Aufrufe mit – genauso so oft durchlaufen, wie der Knoten Nachfahren außer sich selbst hat. Also geht die Laufzeit dieses Algorithmus linear mit der Anzahl der Knoten, er hat also nur $O(n)$.

3.3 Durchlaufen von Bäumen

Häufig will man irgendetwas mit den Daten, die in einem Baum gespeichert sind, anstellen. Dazu muss man alle Knoten nacheinander verarbeiten, man spricht von „besuchen“. Nun gibt es verschiedene Möglichkeiten, in welcher Reihenfogle die einzelnen Knoten besucht werden. Bei allemeinen Bäumen sind Preorder und Postorder die wichtigsten Verfahren.

3.3.1 Preorder

Wie das Pre- in Preorder schon andeutet, wird hier irgendetwas zuerst gemacht. Und zwar geht es um das besuchen der Wurzel jedes Teilbaums. Im Preorder-Durchlauf wird zuerst die Wurzel besucht, danach alle Teilbäume, die durch die Kinder der Wurzel gebildet werden. In diesen Teilbäumen (die natürlich disjunkt sind), wird wieder zuerst die Wurzel besucht, und dann alle Teilbäume, die als Wurzel die Kinder der Wurzel des gerade bearbeiteten Teilbaums haben. Im Algorithmus sieht das so aus:

Preorder-Durchlauf

```
1 Preorder(v)
2   besuche(v);
3   Knoten k = k.kinder();
4   for(i = 0;i < k.length;i++)
5     Preorder(k[i]);
```

3.3.2 Postorder

Man ahnt es schon: Bei Postorder wird die Wurzel als letztes besucht. Der Algorithmus sieht dann so aus:

Postorder-Durchlauf

```
1 Preorder(v)
2   Knoten k = k.kinder();
3   for(i = 0;i < k.length;i++)
4     Preorder(k[i]);
5   besuche(v);
```

3.3.3 Binäre Bäume

In einem binären Baum haben alle inneren Knoten genau 2 Kinder. Eine mögliche Anwendung für einen binären Baum ist ein Ratespiel, wo die Fragen immer nur mit „ ja“ oder „nein“ beantwortet werden.

Wenn zum Beispiel eine Person von Beate, Carla, Anne, Detlef und Erik gesucht wird, wobei Beate klein ist, und rote Haare hat, Carla schwarze Haare, und Anne wieder rote Haare hat, Detlef kurze blonde, und Erik lange schwarze Haare hat, kann einem folgender Baum den Fragevorgang verdeutlichen:

```
Ist die gesuchte Person weiblich?
JA:
  Haarfarbe rot?
  JA:
```

```
    Klein?
    JA:
      Beate
    NEIN:
      Anne
  NEIN:
    Carla
NEIN:
  kurze Haare?
    JA:
      Detlef
    NEIN:
      Erik
```

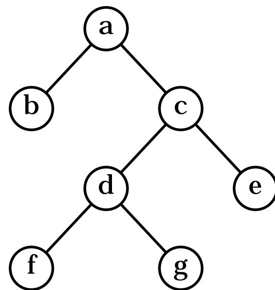
In den inneren Knoten dieses binären Baumes stehen die Fragen, die Antworten stehen in den Blättern. Die Anzahl der zu stellenden Fragen ist die Höhe des Entscheidungsbaum minus 1.

In diesem Fall folgen wir in einen Teilbaum, wenn eine Frage mit „ja" beantwortet wurde, ansonsten in den anderen. Es ist sehr wichtig, diese Teilbäume nicht zu verwechseln: Binäre Bäume sind geordnet, es gibt ein eindeutiges linkes und rechtes Glied, daher stellen wir die Methoden `links` und `rechts` im Objekt `BKnoten` zur Verfügung.

3.3.4 Baumdurchlauf Inorder

Für binäre Bäume, bei denen jeder innere Knoten genau zwei Kinder hat, kann es sinnvoll sein, den inneren Knoten zu besuchen, nachdem der eine, und bevor der andere Unterbaum besucht wird.

```
Inorder(v)
  Inorder(v.links() )
  besuche(v)
  Inorder(v.rechts())
```



Inorder `b a f d g c e`

Preorder `a b c d f g e`

Postorder `b f g d e c a`

Eulertour

Die Eulertour läuft um den Baum so eng wie möglich herum, und notiert alle Knoten, an denen sie vorbeikommt. Im Beispiel sind die Folge so aus

```
 a b a c d f d g d c e c a
   X X     X X X   X X     - Inorder-Auswahl (von unten)
```

Man besucht alle Knoten einmal von links, einmal von rechts und einmal von unten, wobei diese Richtungen daher kommen, dass man sich auf dieser Tour um den Baum einmal links neben jedem Knoten, einmal unter jedem Knoten und einmal rechts neben jedem Knoten steht. Daher gibt es die Alternative, die Blätter drei mal hintereinander zu notieren.

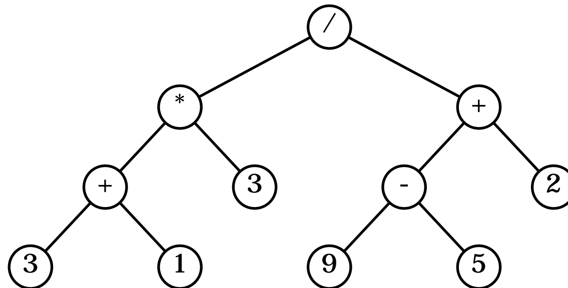
```
Eulertour(v)
  Besuche(v,links)
  if(v.innererKnoten())
    Eulertour(v.links())
  Besuche(v,unten)
  if(v.innererKnoten())
    Eulertour(v.rechts())
  Besuche(v,rechts)
```

Die Inorderfolge beschränkt sich auf die von unten besuchten Knoten, bei Preorder schreibt man die von links, und bei Postorder die von rechts besuchten Knoten auf.

Man kann in der Eulertour die Anzahl der Nachfahren eines Knoten sehr einfach abzählen: Man zählt alle Knoten zwischen dem ersten und letztzen Auftauchen des Knotens, dessen Nachfahren wir zählen wollen, inklusive des Knotens selbst, und dividieren die Anzahl durch drei, da jeder Knoten von links, rechts und oben besucht wurde.

3.3.5 Ausdrucksbäume

Wir benutzen der Einfachheit halber nur die Operatoren +,-,*,/. $[(3+1)*3]/[(9-5)+2]$ ergibt dann folgenden Baum:

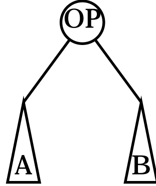


Die Werte stehen in den Blättern, die inneren Knoten beschreiben, was mit den Ergebnissen der Teilbäume zu tun ist. Die Postorder-Folge des Baumes ergibt den Ausdruck in UPN, die Inorder-Folge ergibt den Ausdruck in der üblichen Notation – aber ohne Klammern! Der Ausdruck in Inorder-Form lässt sich also nicht eindeutig auswerten. Jetzt betrachten wir die Eulertour durch den

Baum, wobei wir Blätter nur einmal notieren, und von links besuchte Knoten durch `(` ersetzen, und von rechts besuchte Knoten durch `)`

$$(_/(_*(_+3+1)_+ * 3)_*/(_+(_-9-5)_- + 2)_+)_/$$

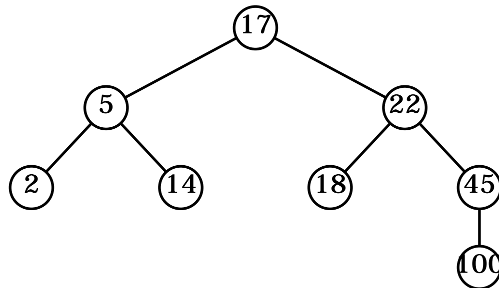
Was passiert dabei eigentlich? Betrachten wir mal einen Ausschnitt:



Das wird von der Eulertour umgeformt in $((A)$ OP $(B))$.

3.3.6 Binäre Suchbäume

Man kann Bäume benutzen, um in den Knoten Daten zu speichern. Dazu erhält jeder Knoten die Methode `getData`, mit der man den Wert dieses Knoten erfragen kann. Man braucht für Suchbäume eine total geordnete Menge. Es gilt die Regel, das im linken Teilbaum alle Werte kleiner oder gleich sind, als der Wert in der Wurzel dieses Teilbaums, und im rechten Teilbaum alles Werte größer oder gleich sind als der Wert in der Wurzel



Das entspricht nicht ganz unserer bisherigen Erklärung von binären Bäumen, nach denen ein Knoten immer *genau* zwei Blätter hat, daher hängen wir an alle offenen Enden noch ein Blatt an, was gar keinen Wert hat, und an alle anderen Enden zwei Blätter, die gar keinen Wert haben. Die Daten sind also in diesem Falle nur in den inneren Knoten gespeichert. Um die Zahlen in sortierter Folge auszugeben, gibt man den Baum in Inorder-Folge aus. Denn der linke Teilbaum, der vor der Wurzel steht, wird vor der Wurzel ausgeben, und erst danach, der rechte Teilbaum, dessen Werte alle größer als die Wurzel sind.

Suchen im Suchbaum

Wir fragen uns durch den Baum, genau wie im Beispiel mit den Personen. Hier würde die erste Frage also lauten „Ist die Zahl größer, kleiner oder gleich 17?“ Danach entscheiden wir, ob wir nach links oder rechts gehen müssen, oder das Element bereits gefunden haben

```
// Sucht x im Teilbaum mit der Wurzel v
binäreBaumsuche(x,v)
```

```
if(v.getData().key == x.key)
  return v
else if(v.istBlatt())
  throw new NotFoundException("Gibt es nicht");
else if(x.key < v.getData().key)
  return binäreBaumsuche(x,v.links())
else
  return binäreBaumsuche(x,v.rechts())
```

Dieser Algorithmus hat eine Laufzeit, die $O(\mathsf{Tiefe})$ liegt. In der gleichen Zeit kann man das Minimum und das Maximum finden, indem man an jedem Knoten nach links bzw. rechts abbiegt, bis es nicht mehr geht.

Einen binären Suchbaum erstellt man, indem man jede Zahl dorthin packt, wo sie hingehört, und das ganze über alle Zahlen iteriert, die in den Suchbaum eingefügt wird. Das Einfügen ist vergleichsweise einfach, denn die neuen Zahlen können immer als Blatt erzeugt werden.

Wenn man einzelne Elemente löschen will (zum Beispiel, weil sie falsch waren, und man eine korrigierte Version eintragen will), kann drei wesentlich verschiedene Fälle geben.

Zwei null-Kinder In diesem Fall ist das Einfügen ganz einfach, denn dann gibt es keine Kinder mehr, die wir betrachten müssen.

Löschen ohne echte Kinder

```
1  if (v==v.Vater.linkesKind)
2    v.Vater.linkesKind = null;
3  else
4    v.Vater.rechtesKind = null;
```

Ein null-Kind Dieser Fall ist immernoch recht einfach: Der zu löschende Knoten hat einen Vater und ein Kind. Die kann man auch direkt verbinden.

Löschen mit einem Kind

```
1  if(v.linkesKind == null)
2    u = v.rechtesKind;
3  else
4    u = v.linkesKind;
5  if (v==v.Vater.linkesKind)
6    v.Vater.linkesKind = u;
7  else
8    v.Vater.rechtesKind = u;
9  u.Vater = v.Vater;
```

Kein null-Kind Dieser Fall ist etwas komplexer, da sich dort nicht einfach der Baum zusammenziehen lässt. Üblicherweise tauscht man ein Glied des Baumes an die Stelle, wo man löschen will. Als Ersatzkonten für den zu löschenden Knoten kommen genau zwei Werte in Frage. Wir brauchen nämlich einen Wert, der kleiner ist als der rechtet Teilbaum, und größer als der linke. Das klappt nur, wenn man das Maximum aus dem linken oder das Minimum aus dem rechten Baum herausnimmt und an die Stelle des zu löschenden Elements setzt. Ich erinnere noch einmal daran, dass man das Minimum in einem binären Baum dadurch findet, dass man den „linkesten" Knoten heraussucht.

```
t = v.rechtesKind.minimum();
v.setData(t.getData());
t.delete();
```

3.3.7 Gleichungen über Bäume

- #Blätter = #innereKnoten+1

  Beweis: Wir löschen einen Knoten, der ein Blatt als Kind hat, zusammen mit diesem Blatt. Dabei verringern wir die die Anzahl der Knoten und die Anzahl der Blätter um eins. Jetzt wendet man vollständige Induktion an, und stellt fest, dass der Satz gilt, da am Ende ein einzelnes Blatt, nämlich die Wurzel übrig ist, und bis dorthin genausoviele Knoten wie Blätter entfernt sind.

- Level i etnhält höchstens 2^i Knoten

  Beweis: $|\mathsf{Level(i)}| \leq 2|\mathsf{level(i-1)}| \leq 2^i|\mathsf{Level(0)}| = 2^i$

- $2h - 1 \leq$ #Knoten $\leq 2^{n+1}$

  Beweis: Das erste folgt daraus, dass, um die Höhe h zu erreichen, mindestens h innere Knoten benötigt, der Rest folgt aus Satz 1. Das zweite erhält man durch die Summierung des Satzes 2 über alle Levels.

- $h \leq$ #Blätter $\leq 2^{n+1}$

- $\log(n+1) - 1 \leq \mathsf{Höhe} \leq \frac{n-1}{2}$

3.3.8 Darstellung von binären Bäumen

In Arrays

Wir definieren eine Funktion $p(v)$, die uns sagt, welche Position im Array B zu welchem Knoten gehört. Damit wir beliebige Bäume darstellen können, ist die Funktion p vom Baum abhängig. Häufig wird $p(v)$ für binäre Bäume so definiert:

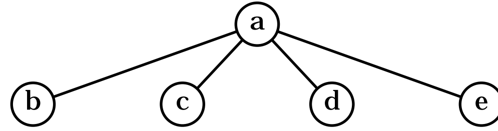
```
p(Wurzel) = 1
p(v.links()) = 2p(v)
p(v.rechts()) = 2p(v)+1
```

Die Methoden funktionieren alle recht schnell:

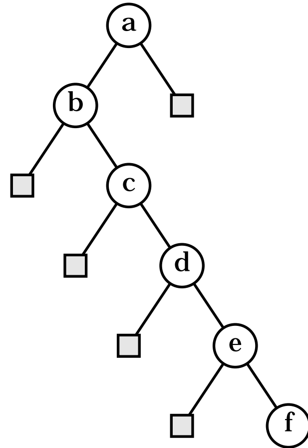
```
istBlatt(v) = B[2p(v)] == null && B[2p(v)+1] == null
Vater(v) = B[p(v)/2];   // Javamäßiges abrunden!
```

Der Speicheraufwand ist dagegen erheblich, da uns nur selten voll besetzte Bäume begegnen, aber wir immer dafür Platz reservieren müssen. Der Platzverbrauch steigt exponentiell mit der maximalen Höhe des Baumes, ungeachtet der tatsächlichen Auslastung des Baumes.

3.3.9 Darstellung beliebiger Bäume als binäre Bäume

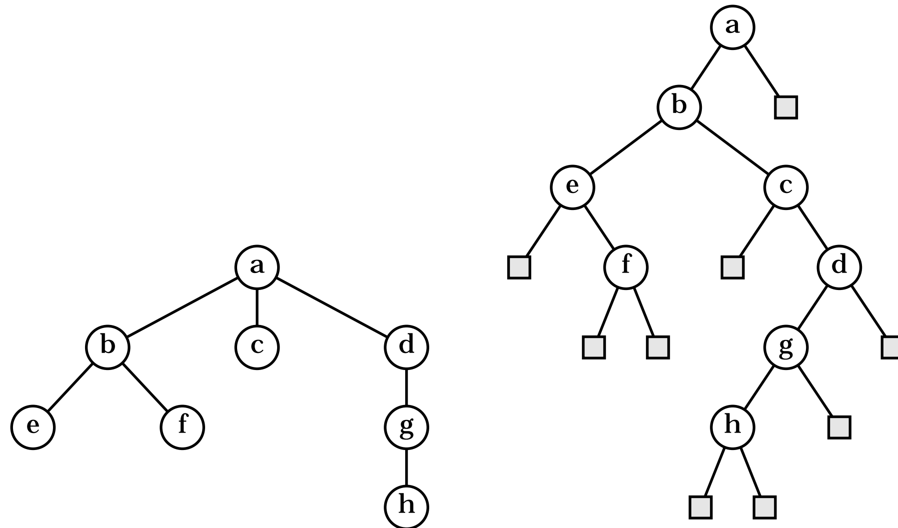


wird zum Beispiel zu



Man kann beliebige Bäume als binäre Bäume darstellen, in dem man das erste Kind eines Knotens als linkes Kind erklärt, und alle weiteren jeweils als rechtes Kind des vorherigen Kindes, wie in der Grafik dargestellt.

Ein komplexeres Beispiel wäre folgendes:



3.3.10 Vollständiger binärer Baum

Definition: Hat T die höhe h, also die Level $0, \ldots, h$, dann heißt T vollständig, wenn sich in Level i genau 2^i Knoten befinden für $i \leq h-1$, und in Level h sich alle Knoten ganz links befinden.

Satz: Sei $m = \#$innere des vollständigen binären Baumes T, dann gilt für seine Höhe h: $2^{h-1} \leq m \leq 2^h - 1$.

Beweis: Schlechtester Fall: Auf der letzten Ebene ist nur ein Knoten, dann gilt $m = \sum_{i=0} h - 22^i + 1 = 2^{h-1} + 1 - 1 = 2^{h-1}$

Satz: $log(m+1) \leq h \leq logm + 1$

Beweis: der Satz folgt durch Logarithmieren des vorherigen Satzes, und Umstellen der Ungleichung.

Corrolar: $h = \lceil \log_2(m+1) \rceil$

Beweis: $\log(m+1) \leq h \leq \log(m) + 1 < \log(m+1) + 1$

Also liegt h zwischen $\log(m+1)$ und $\log(m+1)+1$. Dazwischen gibt es nur eine ganze Zahl, die man dann erhält, wenn man $\log(m+1)$ aufrundet. Wenn $\log(m+1)$ ganzzahlig ist, ist $h = \log(m+1)$.

3.4 Folgen, Sequenzen, ...

Der Datentyp `enumeration` oder `iterator` beschreibt eine geordnete Folge von Daten. Er muss die Methoden `isEmpty()` und `nextElement(v)` unterstützen. In einem allgemeinen Baum in der oben beschriebenen Implementierung ist `Kinder(v)` vom Typ `enumeration`.

3.5 Priority Queue

Auf Deutsch gibt es dafür den umständlichen Begriff Prioritätswarteschlange, und damit wird eine Queue beschrieben, in der die Elemente nicht unbedingt nach der Reihenfolge herauskommen, in der sie hineingesteckt wurden, sondern die Daten eine Priorität haben, und immer das wichtigste Element als erstes herausgegeben wird. Da der Begriff selbst im Englischen recht lang ist, wird er häufig mit PQ abgekürzt.

3.5.1 abstrakter Datentyp

Eine Menge von Daten $a_1, \ldots, a_n \in A$, und eine Menge von Schlüsseln $k \in K$, dabei ist K total geordnet. Die Methoden:

`einfügen(a,k)`	fügt `a` mit der Priorität `k` ein
`entferne_Min`	gibt das Element mit dem kleinsten `k` aus
`ist_leer`	`true`, falls die Liste leer ist
`min_Schlüssel`	gibt den Wert des kleinsten Schlüssels aus
`min_Element`	gibt das Element mit dem kleinsten `k` aus, ohne es zu entfernen

3.5.2 Sortieren mit PQs

Ordne jedem Datum seinen Sortierschlüssel als Priorität zu. Dann füge diese Elemente in die Priority-Queue ein. Sie kommen von selbst in sortierter Reihenfolge wieder heraus

3.5.3 Implementation mit Folgen

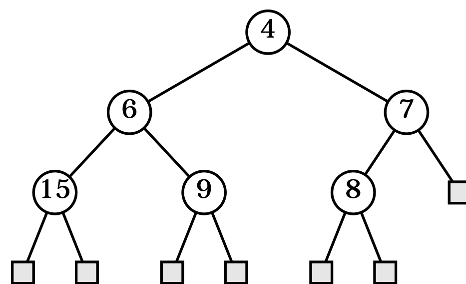
Man kann entweder die Folge unsortiert lassen, oder jederzeit dafür sorgen, dass sie stets sortiert ist. Zuerst betrachten wir den Fall, in dem die Folge unsortiert ist, und neue Elemente hinten angefügt werden. Die Einfügemethode läuft sehr schnell mit `O(1)`, da das Ende bekannt ist. Allerdings muss man zum Heraussuchen des Element mit dem kleinsten Schlüssel suchen, was `O(n)` benötigt

Wenn man mit einer sortierten Folge arbeitet, dauert das Entfernen des ersten Elements nur `O(1)`, aber dafür braucht das Einfügen in dieser Variante länger, nämlich `O(n)`.

Die Laufzeit beim Sortieren mit den Folgen ist im ersten Falle `O(n^2)`, da `n` Elemente herausgesucht werden müssen, und es jedes mal `O(n)` dauert. Im zweiten Fall dauert es ebenfalls `O(n^2)` , da des Einfügen in die Queue für jedes der `n` Elemente `O(n)` dauert.

3.5.4 Implementation mit Heaps

Auf Deutsch heißen Heaps „Halde“oder „Haufen“. Sie sind eine gute Datenstruktur für Prioritätswarteschlangen. Ein Heap verwaltet eine Menge von Elementen. Die Methoden sind `insert` zum Einfügen und `extract` zum Herausholen des Elements mit der höchsten (numerisch kleinsten) Priorität. Jedes Element eines Heaps besteht aus einem Datensatz und einem Schlüssel. Für jeden inneren Knoten in einem Heap gilt die Ungleichung $\mathsf{Key}(v) \geq \mathsf{Key}(\text{Vater}(v))$. Daraus folgt, dass der kleinste Schlüssel immer in der Wurzel steht. Weiterhin sind Heaps vollständige binäre Bäume, das heißt, alle Levels, bis auf den letzten sind voll belegt (oder vollständig), und im letzten Level sind die restlichen Knoten von links nach rechts ohne Lücke angeordnet (man sagt auch linksbündig gefüllt). Heaps benötigen für `insert` und `extract` $O(\log n)$, daher geht das Sortieren mit $O(n \log n)$.



3.5.5 Implementierung

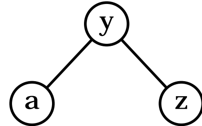
- Alle Blätter sind `null`
- Zeiger `last` auf das letzte Element

Nachdem wir überlegt haben, wie ein Heap grundsätzlich funktionieren soll, und wir ein paar Details festgelegt haben, die uns bei der Implementierung helfen sollen, können wir anfangen, die Methoden zu schreiben.

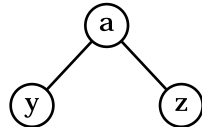
Zum Einfügen von einem Datum `a` mit dem Schlüssel `k` in einen Heap braucht man folgende Schritte

1. Finde das Blatt, bei dem eingefügt werden soll (also: Suche die `null`-Referenz, die durch den Knoten mit (a, k) ersetzt werden soll). Dazu geht man von `last` solange nach oben, bis man einen Knoten von links antrifft. Von diesem Knoten geht man nach rechts, und dann wieder nach links, bis man auf ein Blatt trift. Es gibt einen Spezialfal, wenn der letzte Level ganz voll ist, dann muss ganz links in einer neuen Ebene den neuen Knoten anlegen.

2. Speichere (a, k) in dem neuen inneren Knoten. Stelle danach die Heapeigenschaft (Sortierung der Äste) wieder her, indem man das neue Element zur Wurzel hin hochtauscht.

   Wenn jetzt $a < y$ gilt, dann kann man a und y tauschen. Da $y < z$ sowieso schon galt, gilt nach dem Tausch die Heapeigenschaft wieder:

3. Setze last auf das neue Element

Das Suchen der Einfügeposition (also von `last` aus nach oben wandern, bis man die Wurzel trifft oder von links kommt, und dann wieder nach unten gehen), hat maximal einen Aufwand, der der Höhe des Baumes entspricht. Das ist $O(\log n)$. Weiterhin muss man danach im Baum unter Umständen Elemente tauschen. Dabei ist im kompliziertesten Fall das neueste Element das kleinste, und muss bis in die Wurzel getauscht werden. Da es bei jeder Tauschaktion eine Ebene höher rutscht, geht auch das gesamte Tauschen in $O(\log n)$ Zeit von statten.

Zum Löschen des Elements mit dem geringsten Schlüssel, tu folgendes:

1. Kopiere die Daten aus `last` in die Wurzel, und entferne `last`. Da last nur zwei Nullblätter als Kinder hat, ist das kein Problem. Setze `last` auf die neue Position. Das geht analog zum Suchen des nächsten freien Blattes beim Einfügen, nur halt andersherum.

2. Stelle die Heapeigenschaft wieder her, indem das mit hoher Warscsheinlichkeit zu große Element aus der Wurzel nach unten getauscht wird. Dabei muss das zu große Element stets gegen das kleinere der Kinder getauscht werden, da es ja dann über dem anderen Kind liegt. Den Weg, den das Element beim nach unten Tauschen zurücklegt, wird üblicherweise der „kleinstes-Kind-Weg“ genannt, weil immer gegen das kleinste Kind getauscht wird.

Die Laufzeit beim Entfernen eines Elementes ist proportional zur Höhe des Baumes, die logarithmisch mit der Anzahl der Elemente geht, ist der Aufwand in $O(\log n)$.

3.5.6 Heapsort

Wie der Name schon sagt, wird mit Hilfe eines Heaps sortiert. Dabei werden alle zu sortierenden Elemente in den Heap eingefügt, was für n Elemente je $O(\log n)$ dauert, also insgesamt $O(n \log n)$. Für das Entfernen, bei dem die n Elemente wieder aus dem Heap entfernt werden, braucht man wieder n Operationen zu je $O(\log n)$.

Streng genommen war die Rechnung oben zu einfach, denn der Aufwand zum Löschen und Einfügen im Baum hängt eigentlich nicht von der Gesamtzahl der Elemente, sondern von der Zahl der im Baum vorhandenen Elemente ab. Da es aber unproblematisch ist, nach oben abzuschätzen, können wir die Zahl der Elemente im Baum, die stets kleiner oder gleich der Zahl der Elemente überhaupt ist, durch die Gesamtzahl abschätzen, was die obige Rechnung legitimiert.

Beim Sortieren sind die n Elemente schon am Anfang bekannt, damit kann man den Heap schneller als in $O(n \log n)$ aufbauen. Dazu benutzt man so genannte „Bottom-Up-Heaps“. Die Eingabe habe *genau* $n = 2^k - 1$ Elemente aus einer totalgeordneten Menge, die Ausgabe ist ein Heap aus diesen Elementen. Falls man keine „glatte Anzahl“ Elemente hat, kann man sich das ganze mit $+\infty$ auffüllen.

Dieser Algorithmus baut aus den Elementen zuerst einen vollständigen binären Baum ohne Heap-Eigenschaft. Wir gehen jetzt von unten nach oben durch den Baum. In der untersten Ebene ist die Heapeigenschaft trivialerweise erfüllt. Wir gehen jetzt einen Level höher. Dort können wir die Wurzel wie beim Wiederherstellen nach dem Löschen eines Elementes nach unten auf dem kleinstes-Kind-Weg durchtauschen. Das machen wir mit allen Ebenen, bis wir am Ende auf der obersten Ebene angekommen sind.

Der Aufwand für das herabsinkenlassen der neuen Wurzel entlang des kleinstes-Kind-Weges ist proportional zur Länge dieses Weges. Wenn man die Länge jedes Weges mit der Gesamthöhe des Baumes abschätzt, erhält man für das Einsortieren eines Knotens die Laufzeit $O(\log n)$, also für alle Knoten die Laufzeit $O(n \log n)$. Da wir zeigen wollen, dass die Bottom-Up-Heaps schneller sind, brauchen wir eine bessere Abschätzung der Weglänge. Der Aufwand ist, wenn man die Anzahl der Knoten in dem Baum mit $n = 2^h - 1$ ansetzt:

$$
\begin{aligned}
&\sum_{v \text{ Knoten}} \text{Länge des Weges für } v \\
&= \sum_{i=0}^{\log n} (\text{Anzahl Knoten auf Rang } i)(\log n - i) \\
&= \sum_{i=0}^{h-1} 2^i (h - i)
\end{aligned}
$$

Ein visuelles Argument zu Abschätzung

Der Aufwand für den Knoten v ist proportional zur Länge des kleinstes- Kind-Weges, der nie länger als der längste absteigende Weg ist. Wähle daher für v einen speziellen absteigenden Weg, und zwar: Einen Schritt nach links, danach nur noch nach rechts. Solche Wege werden `lr*`-Wege genannt.

Man sieht, dass jede Kante nur auf höchstens einem der `lr*`-Wege liegt, also die Anzahl der Tauschoperationen stets kleiner als die der Kanten im Baum ist. Die Anzahl der Kanten im Baum mit n Elementen ist $2n$, wenn man auch noch die `null`-Blätter mitzählt. Also wird der Bottom-Up-Heap in $O(n)$ aufgebaut.

3.6 Sortieren

Wir kennen Verfahren, die eine Menge in $O(n \log n)$ sortieren, nämlich MergeSort und QuickSort. Geht es denn überhaupt schneller? Falls man die Elemente der zu sortierenden Menge untereinander vergleicht, geht es im schlechtesten Fall *nicht* schneller.

Eine ganz andere Idee ist möglich, wenn die Schlüssel eine geeignete Feinstruktur aufweisen, oder nur wenige Schlüssel vorkommen. In diesem Fall kann man sogar in linearer Zeit (d. h. $O(n)$) sortieren. Wenige Schlüssel bedeuten dabei konstant viele, also zum Beispiel, wenn eine Liste von Personen nach ihrem Alter in Jahren sortiert werden soll, kommt man mit den Schlüsseln 0 bis 150 aus. Ebenso kommt man mit 26 Schlüsseln aus, wenn man nach dem Anfangsbuchstaben des Nachnamens sortieren will. Schlüssel, die sich aus Namen bilden lassen, bilden dagegen eine geeignete Feinstruktur, die das Sortieren in linearer Zeit ermöglichen.

3.6.1 CountingSort

CountingSort ist ein Algorithmus, der den Fall, in dem es nur wenige Schlüssel gibt, gut löst.

Annahme: Die Schlüssel sind die Zahlen 1 bis k

Idee: Zähle für jedes Element x, wieviele Elemente kleiner sind. Damit kann man bestimmen, wohin das Element x in der Ausgabe gepackt werden muss.

```
// Es gibt die Schlüssel 1..k im Array A der zu
// sortierenden Elemente.
CountingSort(k,A)
  B = new item[A.length]
  C = new int[k+1]

  for(i = 0; i < A.length; i++)
    C[A[i].key]++;
  // Jetzt ist C[t] die Anzahl der Items in A
  // mit dem Schlüssel t

  for(i = 1; i < k; i++)
    C[i] = C[i] + C[i-1];
  // Jetzt ist C[t] die Anzahl der Items im A
  // mit einem Schlüssel keiner oder gleich t

  for(i = 0;i < A.length;i++)
    C[A[i].key] = C[A[i].key]-1;
    B[C[A[i].key]] = A[i];

  return B;
```

Der Aufwand für die erste Schleife ist $O(n)$, der für die zweite $O(k)$. Die dritte Schleife benötigt wieder $O(n)$. Insgesamt ergibt das einen Aufwand von $O(n+k)$, und falls k konstant ist, einfach $O(n)$.

Beispiel:

$$A = [1_a, 4_a, 2_1, 1_b, 2_b, 2_c, 1_c]$$

$$C = [0, 3, 3, 0, 1] \quad \text{nach der ersten Schleife}$$

$$C = [0, 3, 6, 6, 7] \quad \text{nach der zweiten Schleife}$$

A	B							C				
	0	1	2	3	4	5	6	0	1	2	3	4
								0	3	6	6	7
1_a			1_a					0	2	6	6	7
4_a			1_a				4_a	0	2	6	6	6
2_a			1_a			2_a	4_a	0	2	5	6	6
1_b		1_b	1_a			2_a	4_a	0	1	5	6	6
2_b		1_b	1_a		2_b	2_a	4_a	0	1	4	6	6
2_c		1_b	1_a	2_c	2_b	2_a	4_a	0	1	3	6	6
1_c	1_c	1_b	1_a	2_c	2_b	2_a	4_a	0	0	3	6	6

3.7 Dictionaries

Auf Deutsch auch Wörterbücher genannt, speichern sie Objekte mit Schlüsseln. Ein Wörterbucheintrag („item“) besteht also aus einem Schlüssel („key“) k und einem Element e. In Wörterbüchern sind die wichtigsten Operationen des Einfügen, das Nachschlagen und das Löschen von Einträgen. Beispiele für ein Lexikon sind:

Anwendung	**Schlüssel**	**Wert**
Lexikon	Suchbegriff	Erlärung
Duden	Deutsches Wort	Beugungsformen
Telefonbuch	Name und Vorname	Telefonnummer
Kundendaten	Kundennummer	Infos über ihn
Wetterdatei	Koordinaten	Messwerte

Es gibt geordnete und ungeordnete Wörterbücher. In geordneten Wörterbüchern sind die Schlüssel total geordnet (zur Erinnerung: man kann je zwei Wörter vergleichen, und wenn x kleiner als y und y kleiner als z ist, dann ist auch x kleiner als z).

3.7.1 Methoden eines Wörterbuches

`int size()`	Gibt die Anzahl der Einträge zurück
`bool isEmpty()`	Testet, ob das Wörterbuch leer ist
`Element find(k)`	Gibt ein Element mit dem Schlüssel k zurück, falls kein solcher Eintrag existiert, wird ein spezieller Wert zurückgegeben.
`Element[] findAll(k)`	Gibt ein Array aller Elemente mit dem Schlüssel k zurück
`insert(k,e)`	Trägt den Eintrag (k, e) in das Wörterbuch ein.
`Element remove(k)`	Löscht (irgend-)ein Element mit dem Schlüssel k, und gibt es zurück, existiert kein Element mit diesem Schlüssel, wird ein spezieller Wert zurückgegeben
`Element[] removeAll(k)`	Streicht alle Elemente mit Schlüssel k und gibt ein Array mit den gestrichenen Objekten zurück.

Der oben erwähnte spezielle Wert kann sich je nach Implementation des Wörterbuches unterscheiden. In einer Java-Implementation werden ziemlich sicher Objekte im Wörterbuch gespeichert, weswegen der Wert `null` für diesen Fall prädestiniert ist. Man könnte auch eine Exception, aber das ist häufig eine etwas zu starke Maßnahme, oder ein spezielles Element, was im Wörterbuch nicht vorkommen kann (falls man ein Wörterbuch über primitive Typen implementiert). Ab jetzt wird diesers Wert mit `NO_SUCH_KEY` bezeichnet.

3.7.2 Beispiel

`insertItem(6,C)`	$\{(6, C)\}$	—
`insertItem(8,A)`	$\{(6, C), (8, A)\}$	—
`insertItem(3,B)`	$\{(6, C), (8, A), (3, B)\}$	—
`insertItem(6,D)`	$\{(6, C), (8, A), (3, B), (6, D)\}$	—
`findElement(8)`	unverändert	`A`
`findElement(6)`	unverändert	`C` oder `D`
`findAllElements(6)`	unverändert	`{C,D}`
`remove(7)`	unverändert	`NO_SUCH_KEY`
`removeAll(6)`	$\{(8, A), (3, B)\}$	`{C,D}`

3.7.3 Geordnete Wörterbücher

Wenn die Schlüssel einer Totalordnung unterliegen, dann definiert man auch noch folgende Methoden:

`key KeyBefore(k)`	gibt den größten Schlüssel, der kleiner oder gleich k ist, zurück.
`Element ElementBefore(k)`	gibt das Element mit dem größten Schlüssel, der kleiner oder gleich k ist, zurück.
`key KeyAfter(k)`	gibt den größten Schlüssel, der kleiner oder gleich k ist, zurück.
`Element ElementAfter(k)`	gibt das Element mit dem größten Schlüssel, der kleiner oder gleich k ist, zurück.

3.7.4 Implementierungen

Implementation im Array

Bei ungeordneten Wörterbüchern hängt die `insert`-Funktion das Element hinten an, und benötigt $O(1)$, `find` und `remove` müssen dagegen das ganze Array durchlaufen, und benötigen daher $O(n)$. Man kann dieses Implementation dann verwenden, wenn es sehr unwahrscheinlich ist, dass überhaupt jemals ein `find` gemacht werden muss, zum Beispiel, um die Zwischenergebnisse einer Rechnung zu sichern, für den Fall, dass der Rechner abstürzt. Dort müssen dauernd neue Zwischenergebnisse eingefügt werden, aber da der Computer normalerweise nicht abstürzt, ist egal, wie lange das Suchen darin dauert.

In geordneten Wörterbüchern sind die Einträge stets nach dem Schlüssel sortiert. `findElement` kann mit einer binären Suche arbeiten. Das bedeutet, dass man in der Mitte des Arrays aufsetzt, und dann prüft, ob der gesuchte Schlüssel in der vorderen oder hinteren Hälfte ist. Danach wird die entsprechende Hälfte genommen, die Mitte angesehen, und geprüft, ob der Eintrag im ersten oder zweiten Viertel liegt. Das macht man so lange, bis nur noch ein Element existiert. Es handelt sich also wieder um eine Art Divide & Conquer-Algorithmus. (eigentlich nicht, denn es wird ja nicht danach in beiden Hälften gesucht)

Binäre Suche

```
// Am Anfang setze l auf 0 und r auf n-1
binarySearch(k,l,r)
  // l größer klingt zwar schwachsinning, tritt aber
  // tatsächlich auf, wenn der Schlüssel nicht existiert,
  // da dann Intervalle aus keinem Element durchsucht
  // werden (irgendwo zwischen n und n+1 muss er doch
  // liegen, also fange ich die Suche bei n+1 an, und kann
  // schon bei n aufhören)
  if(l > r)
    return NO_SUCH_KEY;

  m = (l+r)/2           // Abrunden
  if(k == keys[mid])
    return elems[mid]
  else if(k > keys[mid])
    return binarySearch(k,l,m-1);
  else
    return binarySearch(k,m+1,l);
```

Wir analysieren jetzt die Anzahl der rekursiven Aufrufe, da jeder Aufruf konstante Zeit benötigt, und die Zeit pro Aufruf einfach nur ein konstanter Faktor wird. Dazu betrachtet man die Anzahl der zu durchsuchenden Elemente, das ist $(r-l)+1$. Am Anfang ist das n, und in jedem Schritt wird die Größe dieses Intervalls auf die Hälfte oder noch stärker beschränkt, da immer nur die Elemente zwischen einem Randelement und der Mitte betrachtet werden, das Element in der Mitte is ausgeschlossen. Formelmäßig sieht es so aus:

$$
\begin{aligned}
& k < \texttt{key}[m]: \\
(m-1)-l+1 &= \left\lfloor \frac{l+r}{2} \right\rfloor - l \leq \frac{l+r}{2} - \frac{2l}{2} \\
&= \frac{r-l}{2} \leq \frac{r-l+1}{2} \\
& k > \texttt{key}[m]: \\
r-(m+1)+1 &= r - \left\lfloor \frac{l+r}{2} \right\rfloor \leq \frac{2r}{2} - \frac{l+r}{2} \\
&= \frac{r-l}{2} \leq \frac{r-l+1}{2}
\end{aligned}
$$

Nach m Operationen ist das Problem auf $\frac{n}{2^m}$ reduziert, und nach $\log_2 n$ ist das Problem auf ein Element reduziert in diesem Fall ist dann $r = l = m$, und wenn der gesuchte Schlüssel nicht existiert, dann wird die Funktion mit l und $l-1$ oder $r+1$ und r aufgerufen, was zur Abbruchbedingung führt. `binarySearch` ist in $O(\log n)$.

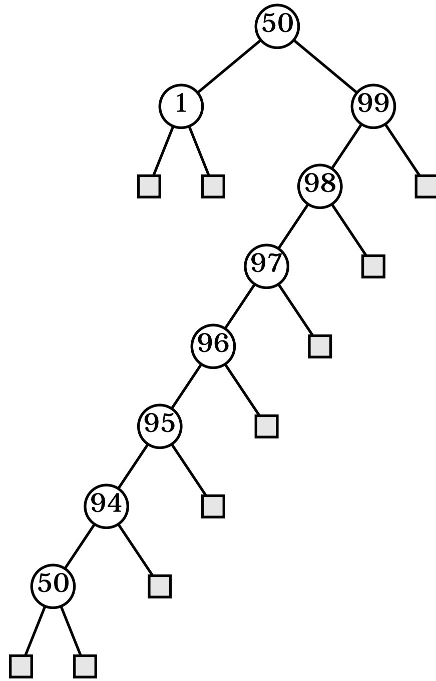
Um Einzufügen, sucht man zuerst die Stelle, wo das Element eingefügt werden soll, mit `closestKeyBefore`, und verschiebt dann alle Elemente dahinter um eine Position. Dann kann man das neue Element an die neue Stelle einfügen. Da man das Verschieben des Arrays elementweise machen muss, benötigt das ganze $O(n)$. Das ist vergleichsweise ungünstig, und für das Streichen sieht es analog aus. Solange man nur den Brockhaus von CD-ROM verwendet, bei dem die Daten naturgemäß nicht verändert werden, kann uns das aber egal sein.

Binäre Suchbäume

3.7.5 Laufzeit im Suchbaum

`findElement(k)` benötigt $O(h)$, wobei h die Tiefe des gesuchten Knotens ist. Dummerweise kann unser Baum auch zu einer Geraden entarten (immer nur linke Blätter benutzt), daher ist der schlechteste Fall $O(n)$. Für `insertItem(k,e)` und `remove(k)` muss man auch erst einmal die Stelle suchen, an der gearbeitet werden soll. Im Durchschnitt erhalten wir aber eine Laufzeit in $O(\log n)$, garantieren kann man das aber nicht.

`findAllElements(k)` wird laut vielen Büchern in $O(h+s)$ erledigt, wobei h die Tiefe eines Knotens mit Schlüssel k und s die Anzahl der gefundenen Einträge ist. Diese Aussage ist falsch.



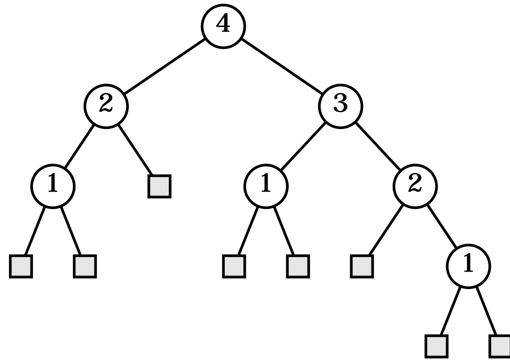
Wie man sieht, muss man den rechten Teilbaum bis ganz nach unten durchgehen, um die zweite 50 zu finden. Daher behaupten wir, dass h halt die maximale Tiefe eines Elements mit dem Knoten h ist. Jetzt kommt aber der Baum ins Spiel, wo unten keine 50 steht, was man aber auch erst feststellt, wenn man unten angekommen ist.

Für dieses Problem gibt es zwei Lösungsmöglichkeiten:

- Man sorgt beim Einfügen dafür, das gleiche Elemente stets beienander gespeichert werden. Man fügt also eine zweite 50 nicht unten ein, sondern als Knoten mit einem Nullblatt direkt zwischen der ersten 50 und der 99. Solche Sachen modifizieren aber die Baumstruktur erheblich, und sind bei der Weiterentwicklung eher hinderlich
- Man hält bei jedem Knoten außer einer Referenz auf den Parent noch eine Referenz auf den in Inorderfolge links beziehungsweise rechts liegenden Knoten. Dann braucht man nur einen zu finden, und kann danach entlang der Inorderfolge (die schon sortiert ist) laufen, um die anderen zu finden

3.8 AVL-Bäume

Die AVL-Bäume heissen nach Adel'son Vel'ski und Landis. Wir erinnern uns noch einmal an die Definition der Höhe eines Knotens, die bekanntlich die Höhe des Teilbaums mit diesem Knoten als Wurzel ist. Anders ausgedrückt ist es die maximale Länge eines absteigenden Weges von diesem Knoten zu einem Blatt. AVL-Bäume sindn Suchbäume mit der sogenannten Höhen-Balance-Eigenschaft, die besagt, dass für jeden inneren Knoten die Höhe der Kinder höchstens um 1 differiert.



In der Grafik sieht man einen AVL-Baum (die Zahlen sind nicht die Werte in den Knoten, sondern die Höhe. Es ist aus der Grafik ersichtlich, dass es Blätter auf mehr als zwei verschiedenen Levels gibt.

Satz: Die Höhe eines AVL-Baumes mit n Einträgen ist in $O(\log n)$.

Beweis: $n(h)$ sei die minimale Größe (Anzahl der inneren Knoten) eines AVL-Baumes mit der Höhe h. Wir sehen im oberen Baum: $n(1) = 1$, $n(2) = 2$, $n(3) = 4$.

Sei jetzt h die Höhe einies Baumes mit der Wurzel r. Dann hat mindestens ein Kind die Höhe $h-1$ und das zweite Kind die Höhe $h-2$. Dann folgt also

$$n(h) \geq \underbrace{1}_{\text{Wurzel}} + \underbrace{n(h-1)}_{\text{erster Teilbaum}} + \underbrace{n(h-2)}_{\text{zweiter Teilbaum}} \geq 2n(h-2)$$

. Also ist $n(h) \geq \sqrt{2}^{h-1} \geq 2^{\lfloor \frac{h-1}{2} \rfloor}$. Wenn man das ganze nach h auflöst, erhält man $\lfloor \frac{h-1}{2} \rfloor \leq \log(n(h))$. Wegen $\frac{h-2}{2} \leq \lfloor \frac{h-1}{2} \rfloor \leq log(n)$. Daraus erhält man $h \leq 2\log(n) + 2$. Das ist ein erfreuliches Ergebnis, denn das bedeutet, dass die Höhe des Baumes logartihmisch mit der Anzahl der Elemente geht, und daher `findElement(k)` in $O(n)$ liegt.

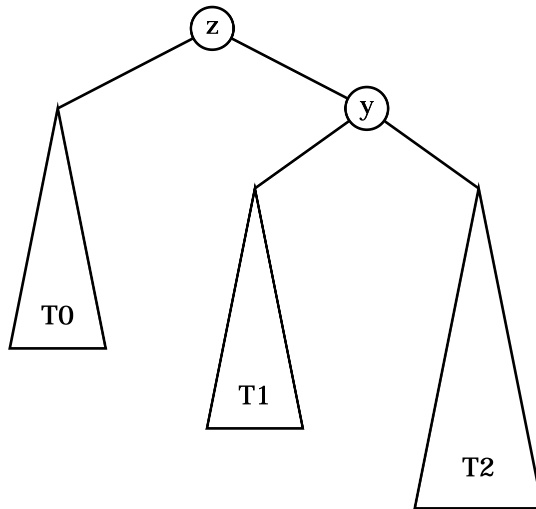
Einfügen in den AVL-Baum

Dazu führt man folgende Schritte aus:

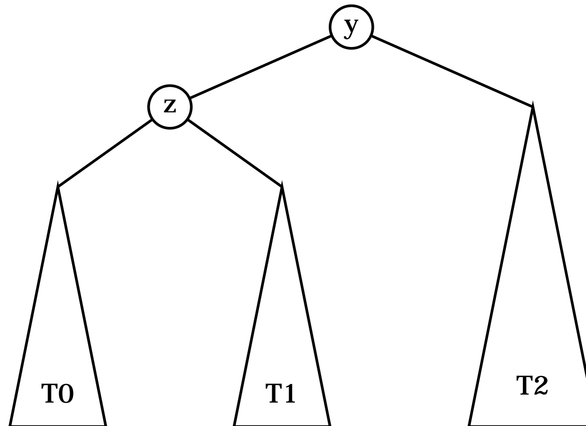
1. Einfügen in den Suchbaum
2. Wiederherstellung der Höhenbalance

Wenn ein Knoten aus der Balance fällt, muss man eine Rotation ausführen. Bei der Rotation bleibt die Ordnung erhalten, und die Höhenbalance wird wieder hergestellt.

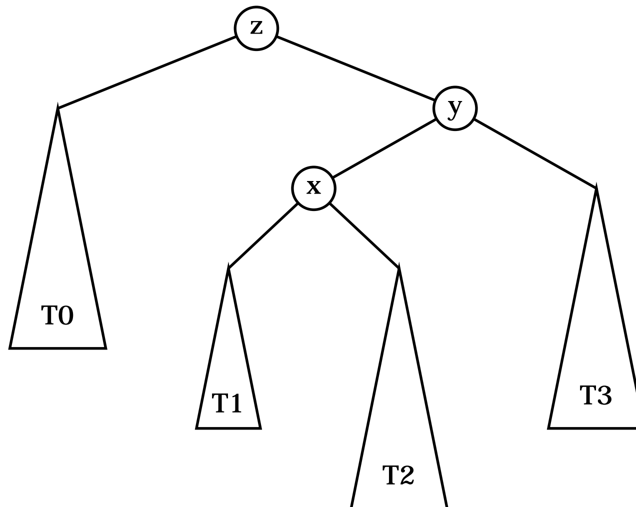
Man muss nach dem Einfügen eines Knotens den Weg von diesem Knoten zur Wurzel durchgehen, und prüfen, ob einer von diesen Knoten unbalanciert ist. Wenn diese nicht der Fall ist, haben wir Glück gehabt. Die dann auftretende Situation sieht häufig wie in folgender Grafik aus:

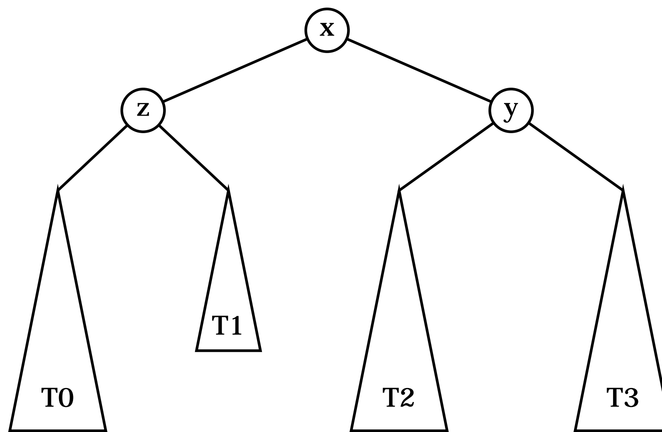


Nach der Rotation der Kante zwischen z und y sieht das dann so aus:



Dieser neue Baum hat die gleiche Tiefe, wie der Baum vor dem Einfügen des neuen Elements, also ist nach der Rotation die Balance im Baum wieder hergestellt. Diese Operation wird „einfache Rotation“ genannt. Der gespiegelte Fall geht analog. Interessant ist aber noch dieser Fall, in dem eine „doppelte Rotation“ notwendig ist:





Auch hierbei bleibt die Reihenfolge erhalten.

Algorithmus restructure

Eingabe: x hat Vater y und Großvater z. Seien T_0, T_1, T_2, T_3 die 4 Unterbäume nach ihren Schlüsseln geordnet, und a, b, c die Knoten x, y, z in ihrer inorder-Folge, also ebenfalls sortiert.

Ersetze jetzt den Unterbaum mit der Wurzel z durchc einen Unterbaum mit Wurzel b. Danach wird a das linke Kind von b mit T_0 und T_1 als Unterbäume, und c das rechte Kind von b mit T_2 und T_3 als Unterbäume.

Folgerung: Einfügen ist in $O(\log n)$ möglich, da das Einfügen in den binären Suchbaum $O(\log n)$ dauert, und das suchen einer Stelle, wo rotiert werden muss ebenfalls. Die Rotation selber kann vernachlässigt werden.

Löschen in AVL-Bäumen

1. Lösche das Element im Suchbaum

2. Stelle die Höhenbalance wieder her

Diese mal werden maximal $O(\log n)$ Rotationen benötigt, da es passieren kann, das ein Baumteil plötzlich so viel zu klein wird, dass mehrfach hineinrotiert werden muss.

Ergebnis

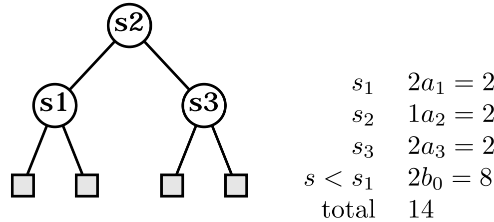
Alle Operationen gehen jetzt in $O(\log n)$ zu erledigen

3.9 Optimale Suchbäume

Gegeben seien n Einträgen (Items) mit Schlüsseln aus einer totalgeordneten Menge. Die Schlüssel seien in aufsteigender Folge durchnummeriert, also $s_1 < s_2 < \ldots < s_3$. Zu jedem Item ist eine Anfragehäufigkeit (das kann man als Wahrscheinlichkeit auffassen) gegeben, nämlich a_i sei die Anzahl der Anfragen nach dem Schlüssel s_i. Weiterhin sei b_i die Anfragehäufigkeit für Schlüssel, die zwischen s_i und s_{i+1} liegt. b_0 ist die Anzahl der Abfragen, deren Schlüssel kleiner als alle Schlüssel im Wörterbuch ist, und b_n analog die Anzahl der Abfragen nach einem Schlüssel, der größer als alle Schlüssel im Wörterbuch ist. Die mit den b_i gezählten Anfragen sind erfolglos.

Gesucht ist ein Suchbaum, mit dem die Anfragen mit minimalen Aufwand bearbeitet werden können.

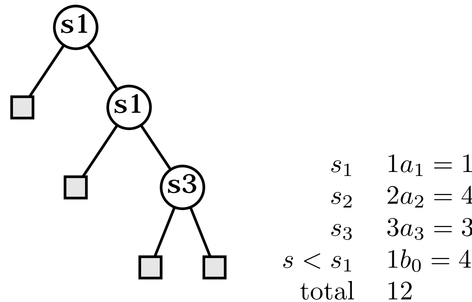
Als Beispiel benutzen wir die Schlüsselmenge $S = s_1, s_2, s_3$ mit $s_1 < s_2 < s_3$. Es gelte $a_1 = 1, a_2 = 2, a_3 = 1$ und $b_0 = 4, b_1 = b_2 = b_3 = 0$. Betrachtet man jetzt den Suchbaum T_1:



s_1	$2a_1 = 2$
s_2	$1a_2 = 2$
s_3	$2a_3 = 2$
$s < s_1$	$2b_0 = 8$
total	14

Wenn wir die Anzahl der Vergleiche zählen, dann erhält für die Anzahl der Vergleiche die neben dem Baum stehende Tabelle.

Mit dem Baum T_2 ergibt sich folgendes Bild:



s_1	$1a_1 = 1$
s_2	$2a_2 = 4$
s_3	$3a_3 = 3$
$s < s_1$	$1b_0 = 4$
total	12

Offensichtlich ist für genau dieses Problem T_2 besser geeignet als T_1, obwohl T_1 viel besser balanciert ist. Das liegt daran, dass wir jetzt nicht den schlechtesten Fall betrachten, der bei T_2 natürlich länger braucht als bei T_1, sondern genau die Anfragen, die tatsächlich gemacht werden, was dazu führen kann, das

zu einer Gerade degenerierte Bäume in manchen Fällen als Ideallösung herauskommen.

Der Leser möge sich an Informatik A[1] erinnern, wo wir die Huffmannbäume besprochen haben. Diese Bäume sorgten dafür, dass ein Text mit gegebener Buchstabenverteilung möglichst kurz kodiert werden muss. Dabei sind die Kosten in der Textlänge die Länge der Pfade im Baum zu den Blättern, in denen die Buchstaben gespeichert sind. Die Anordnung der Knoten im Huffmannbaum ist beliebig.

In unserem Fall sind die Suchwerte in den Knoten des Baumes, in den Blättern (die ja sowieso nur zur Vervollständigung der Knoten eingebaut wurden) steht dagegen nichts. In Suchbäumen muss darauf geachtet werden, dass die Inorderausgabe immer sortiert ist, also die Anordnung der Knoten in keinem Fall beliebig genannt werden kann. Der Huffmann-Algorithmus ist daher nicht anwendbar, da er einfach irgendwelche Teilbäume zusammenpflückt, ohne sich im geringsten um die Ordnung zu scheren.

Wir entwerfen also einen anderen Algorithmus. In einem Baum T nennen wir die Tiefe von s_i l_i und die Tiefe des Blattes $]s_i; s_{i+1}[$ nennen wir λ_i. Die Kosten für T sind $P(T) = \sum a_i(l_i + 1) + \sum b_i(\lambda_i)$. Man nennt $P(T)$ die *gewichtete Pfadlänge*. Mit OSB bezeichnet man einen optimalen Subbaum, also eine Teilbaum, dessen gewichtete Pfadlänge minimal ist.

Jeder binäre (Teil-)Baum besteht aus einer Wurzel und zwei Teilbäumen, nämlich dem linken und dem rechten. Wenn in unserem Fall also s_i in der Wurzel steht, ist der linke Teilbaum ein Suchbaum für $s_1 \dots s_{i-1}$, der rechte einer für $s_{i+1} \dots s_n$. Wenn jetzt der Gesamtbaum ein OSB ist, dann müssen auch die Teilbäume OSBe sein, da die Laufzeiten der Teilbäume in die Gesamtlaufzeit linear einfließen. Es gilt:

$$\begin{aligned} P(T) &= P(T_{\text{links}}) + \sum_{j=0}^{i-1}(a_j + b_j) \\ &+ P(T_{\text{rechts}}) + \underbrace{\sum_{j=i} n(a_j + b_j)}_{\text{konstant}} \end{aligned}$$

Man sieht also auch noch einmal aus der Gleichung, dass außer einem Term, der sich direkt aus der Aufgabenstellung ergibt, in die Gesamtdauer nur die gewichteten Pfadlängen der Subbäume einfließen.

Wenn man mit $P(i, j)$ die gewichtete Pfadlänge eines OSB für den Bereich zwischen s_i und s_j bezeichnet, gilt für die minimalen Kosten eines Suchbaums für $s_1 \dots s_n$ mit der Wurzel i: $P(T) = P(1, i-1) + P(i+1, n) + \sum_{j=1}^{n} a_j + \sum_{j=0}^{n} b_j$. Den optimalen Suchbaum erhält man durch das Ausprobieren aller Wurzeln. Wieso Herr Felsner diesen Algorithmus als so einfach und simpel lobt, kann nur einen Grund haben: Er will uns mal wieder mit einem furchtbar ineffizienten Algorithmus abschrecken.

Lösung durch Rekursion:

```
OptSB(i,k)
  if(k < i)
```

[1] oder siehe `http://www.inf.fu-berlin.de/~lichtebl/infa`

```
    return null
  min=∞
  C = b[i-1]
  for(j = i;j <= k; j++)
    C += a[i]+b[i]
    Q = OptSB(i,j-1)+OptSB(j+1,k)
    if(Q < min)
      min = Q
  return min + C
```

Wie ich schon ahnte, eröffnet uns Herr Felsner jetzt, dass diese Lösung extrem ineffizient ist, nämlich exponentielle Laufzeit (dass bedeutet $O(a^n)$) benötigt. In Informatik A hatten wir so einen Schnarchalgorithmus zur Berechnung der Fibonacci-Zahlen. Bei den Fibonacci-Zahlen war das Problem, dass dasselbe Subproblem dauernd neu gelöst wird, eine Abhilfe dagegen ist eine Tabelle, in der die Zahlen zwischengespeichert werden.

Genau so etwas bietet sich auch hier an. Man muss vermeiden, dass $P(i, k)$ mehrfach die aufwändige Berechnung durchführt, indem Zwischenergebnisse gespeichert werden.

Es gilt, dass man, um einen Aufruf der Intervalllänge n zu behandeln, mindestens 2 rekursive Aufrufe braucht, die die Intervallänge $n-1$ behandeln (und üblicherweise noch viele viele Aufrufe mehr), daher ist $A(n)$ die Anzahl der rekursiven Aufrufe für n Elemente mindestens doppelt so groß wie der für $n-1$ Elemente. Setzt man den Aufwand für ein Element auf 2, dann gilt, dass der Aufwand für n Elemente größer als 2^n ist.

Man beobachtet, dass die Anzahl der verschiedenen Paare (i, k) mit denen `OptSB` aufgerufen wird, nicht sehr groß ist. Es sind nämlich alle Paare (i, k) : $1 \leq i \leq k \leq n$, davon gibt es weniger als n^2. Für diese Elemente erzeugen wir eine Tabelle `P[i][k]`, die die Ergebnisse zwischenspeichert. Um diese Tabelle „on the fly“aufzubauen, legen wir eine weitere Tabelle `Def[i][k]` an, die `bools` enthält, und sagt, ob `P[i][k]` bereits berechnet wurde. Der Algorithmus wird so modifiziert, dass er die Tabelle benutzt. `Def[i][k]` sei am Anfang auf `false` initialisiert:

```
OptSB(i,k)
  if(k < i)
    return null
  if(Def[i][k])
    return P[i][k]
  min=∞
  C = b[i-1]
  for(j = i;j <= k; j++)
    C += a[i]+b[i]
    Q = OptSB(i,j-1)+OptSB(j+1,k)
    if(Q < min)
      min = Q
  P[i][k] = min + C
  def[i][k] = true
  return min + C
```

Die Laufzeit beträgt jetzt $O(n^3)$, da es n Tabelleneinträge gibt, der jeder

mit dem Aufwand $O(n^2)$ berechnet werden muss. Allerdings ist es jetzt recht speicheraufwendig, und wenn der Computer dabei der Speicher ausgeht, und zu swappen anfängt, dann können wir uns das $O(n)$ sonstwo hin packen.

3.10 Dynamisches Programmieren

Dynamisches Programmieren ist ein Entwurfsparadigma wir Divide & Conquer, was man auf Probleme anwenden kann, für die es eine offensichtliche rekursive Lösung gibt, und sich die Subprobleme überlappen (das bedeutet, im rekursiven Aufrufbaum wird mehrfach das gleiche gelöst).

3.11 Fibonacci-Zahlen

$F(k)$ sei die Anzahl der Möglichkeiten, k als Summe der Zahlen 1 und 2 zu schreiben, wobei es auf die Reihenfolge ankommt. Jede Darstellung von k entsteht aus einer Darstellung von $k-1$ mit einem angefügten `+1` oder einer Darstellung von $k-2$ mit einem angefügten `+2`. Daraus erkennt man, dass $F(k) = F(k-1) + F(k-2)$.

```
fib(k)
  if(k == 1) return 1
  if(k == 2) return 2
  return fib(k-1,k-2)
```

Wir haben diesen Algorithmus schon so oft besprochen (nicht unbedingt hier im Skript, aber im allgemeinen), dass ich mir die Herleitung der Laufzeit spare. Es kommt heraus, dass die Laufzeit von `fib(n)` durch `fib(n-1)` angesetzt werden kann. Es gilt $F(n-1) \geq 2^{n-1}$.

3.11.1 Aufbrechen der Rekursion

Top-Down-Aufbau

Eine Möglichkeit ist wie oben, zwei Tabellen zu erstellen, die `F[i]` und `def[i]` heissen. Die Bedeutung davon sei wie oben.

```
fib_td(k)
  if(def[k])
    return F[k]
  def[k] = true
  F[k] = fib_td(k-1) + fib_td(k-2)
  return F[k]
```

Damit das funktioniert, muss `F[1]` und `F[2]` schon vorher existieren, und das im `def`-Array auch gekennzeichzet sein. Diese Funktion arbeitet jetzt in $O(n)$, da nur n Tabelleneinträge berechnet werden müssen, die (vorausgesetzt, die vorherigen Einträge sind schon berechnet), sich in $O(1)$ berechnen lassen.

3.11.2 Bottom-Up-Aufbau

Idee: Systematisches Füllen der Tabelle.

```
fib-bu(k)
  F[1] = 1
  F[2] = 2
  for(i = 3; i <= k; i++)
    F[i] = F[i-1] + F[i-2]
  return F[k]
```

3.11.3 Longest common subsequence

$X = x_1x_2 \dots x_n$

$Y = y_1y_2 \dots \dots x_m$

Teilfolge von X isit eine Folge $x_{i_1}x_{i_2} \dots x_{i_k}$, mit $i_1 \leq i_2 \leq \cdots \leq i_k$. Diese Teilfolge hat die Länge k.

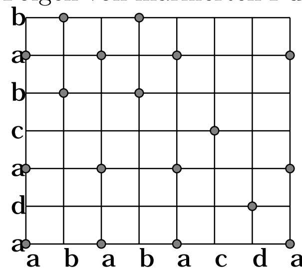
Beispiel:

```
X = a b a b a c d a
Y = a d a c b a b
```

die gemeinsamen Teilfolgen davon sind unter anderem

- `a`
- `a b`
- `a a c`
- `a a b a`
- `b a b`

Man kann eine Matrix anlegen, die entlang der einen Achse mit der Folge X beschriftet ist, die andere mit der Folge Y. Wenn man alle Punkte markiert, bei denen die x-Beschriftung gleich der y-Beschrichtung ist, dann sind alle gemeinsamen Teilfolgen Folgen von markierten Punkten, die von links unten nach rechts oben verlaufen.



$T(k, l)$ sei die Länge der längsten gemeinsamen Teilfolge von $x_1 \dots x_k$ und $y_1 \dots y_l$.

Wenn $x_k \neq y_l$ dann ist

$$T(k, l) = \max \left\{T(k-1, l), T(k, l-1)\right\}$$

Wenn $x_k = y_l$ dann ist

$$T(k, l) = \max \left\{T(k-1, l), T(k, l-1), T(k-1, l-1) + 1\right\}$$

3.12 Graphen

3.12.1 Was ist ein Graph?

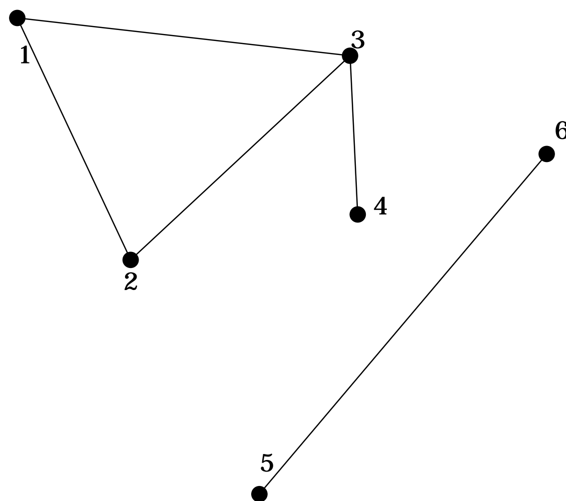
Ein Graph besteht aus Knoten und Kanten, wobei die Kanten Paare von Knoten verbinden. Graphen begegnen uns andauernd, zum Beispiel:

- Straßen– und Schienennetze, Labyrinthe
- Schaltpläne in der Elektrik
- Strukturformeln in der Chemie
- Gitternetze in der Geometrie

3.12.2 Und so sieht das der Mathematiker

Ein Graph G besteht aus einer Menge von Kanten (vertices) V und Knoten (edges) E. Daher schreibt man $G = (V, E)$. Dabei ist E einen Teilmenge der 2-elementigen Teilmengen von V, wer es will, kann es so schreiben: $E \subset \binom{V}{2}$.

Beispiel:
$V = 1, 2, 3, 4, 5, 6 \quad E = \big\{\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{5,6\}\big\}$



3.12.3 Begriffe

Zwei Knoten $x, y \in V$ heißen benachbart oder adjazent, genau dann, wenn $x, y \in E$.

Ein Knoten $x \in V$ und eine Kante $e \in E$ heißen inzident, genau dann wenn $x \in e$, also x einer der beiden Endpunkte ist.

Der Grad eines Knotens $v \in V$ ist die Anzahl der inzidenten Kanten, oder die Anzahl der adjazenten Knoten. Er wird (wegen englisch degree) mit $\mathrm{d}(x)$ bezeichnet. Im Beispiel oben ist zum Beispiel $\mathrm{d}(2) = 3, \mathrm{d}(4) = 1, \mathrm{d}(1) = 2$.

Proposition: Für jeden Graphen $G = (V, E)$ gilt:

$$\sum_{v \in V} \mathrm{d}(v) = 2|E|$$

Beweis: Jede Kante hat zwei Eckpunkte, also wird sie in der Summe der Grade doppelt gezählt.

Folgerung: In jedem Graphen ist die Anzahl der Knoten ungeraden Grades gerade.

Beweis: Zerlege die Summe aus der Proposition in zwei Teile:

$$\begin{aligned}
&\sum_{v \in V} \mathrm{d}v \\
&= \sum_{\substack{v \in V \\ d}} (v)\text{gerade} + \sum_{\substack{v \in V \\ d}} (v)\text{ungerade} \\
&= 2|E|
\end{aligned}$$

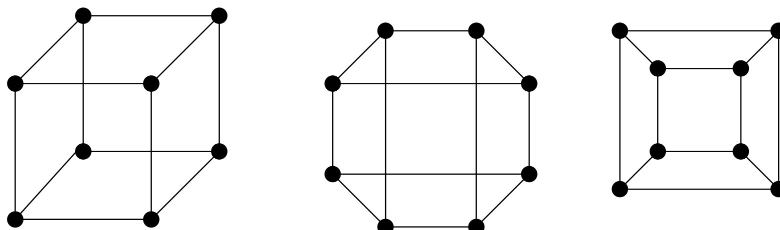
Die rechte Seite ist offensichtlich gerade, der erste Summand ebenfalls, da dort über gerade Zahlen summiert wird. Der zweite Summend muss daher auch gerade sein. Da er eine Summe über ungerade Zahlen ist, muss es eine gerade Anzahl Summanden geben.

3.12.4 Spezielle Graphen

vollständige Graphen $E = \binom{V}{2}$, es sind also alle Knoten direkt durch eine Kante miteinander verbunden. Diese Graphen werden mit K_n bezeichnet. Je nach der Knotenmenge, die ja beliebig sein kann, gibt es natürlich genau genommen verschiedene vollständige Graphen mit der gleichen Knotenzahl. Alle diese Graphen sind allerdings isomorph, das bebdeutet, sie unterscheiden sich nur durch die Wahl der Menge aller Knoten, und können durch Umbenennug der Knoten ineinander übergeführt werden. Isomorphe Graphen werden üblicherweise als gleich betrachtet.

vollständig bipartite Graphen Easy: $V = U \cup W$, $U \cap W = 0$, $E = U \times W$. Anschaulich heißt es, dass man die Menge der Knoten in zwei disjunkte Teile teilen kann, und dann von jedem Knoten Kanten zu jedem Knoten im anderen Teil ausgehen, aber keine zu einem Knoten im selben Teil. Diese Graphen werden mit $K_{n,m}$ bezeichnet, wobei $n = |U|$ und $m = |W|$.

planare Graphen $G = (V, E)$ heißt planar, wenn er kreuzungsfrei in die Ebene einbettbar ist. (Das heißt einfach: Wenn man eine Skizze zeichnet, dann kann man die Knoten so anordnen, dass sich keine Verbindungslinie schneidet). K_3 und K_4 sind planar, da es eine kreuzungsfreie Darstellung gibt, K_5 dagegen nicht.

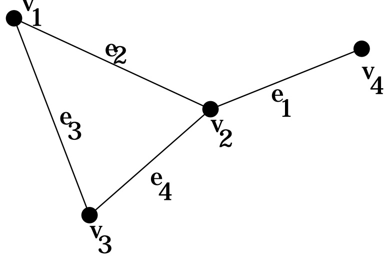


Wie man sieht, ist die Darstellung eines Graphen nicht eindeutig, sondern je nachdem, welche Eigenschaft man hervorheben will, kann man einen Graph

verschieden zeichnen. Im ersten Bild sieht man zum Beispiel die Struktur eines 3D-Würfels, das zweite Bild veranschaulicht, dass es einen Rundweg gibt, auf dem keine Kante zweimal benutzt wird (indem man einfach aussen lang geht), und im dritten Bild sieht man, dass der Graph planar ist.

3.12.5 Algorithmen für Graphen

Als Beispiel für die Datenstrukturen benutze ich diesen Graphen:



Datenstrukturen einen Graphen

Sei $G(V, E)$ ein Graph und $|V| = n, |E| = m$:

1. Adjazenzmatrix: A ist eine $n \times n$-Matrix mit

$$a_{ij} = \begin{cases} 1 & \text{für } v_i, v_j \in E \\ 0 & sonst \end{cases}$$

Damit das funktioniert, müssen alle Knoten auf igendeine Art durchnummerierbar sein, da sonst v_i, also der i-te Knoten nicht definierbar ist.

	v_1	v_2	v_3	v_4
v_1	x	1	1	0
v_2	1	x	1	1
v_3	1	1	x	0
v_4	0	1	0	x

Diese Datenstruktur benötig $O(n^2)$ Speicherzellen.

2. Inzidenzmatrix: N ist eine $n \times m$-Matrix mit

$$n_{ij} = \begin{cases} 1 & \text{für } v_i \in e_j \\ 0 & sonst \end{cases}$$

In diesem Fall muss man außer den Knoten auch die Kanten abzählen, und die einzelnen Kanten mit $e_1 \ldots e_m$ bezeichnen.

	e_1	e_2	e_3	e_4
v_1	0	1	1	0
v_2	1	1	0	1
v_3	0	0	1	1
v_4	1	0	0	0

Da m in der Größenordnung n^2 sein kann (zum Beispiel bei vollständigen Graphen) muss der Speicheraufwand mit $O(n^2)$ abgeschätzt werden.

3. Adjazenzlisten: Für jeden Knoten v_i existiert eine Liste der adjazenten Knoten. Referenzen auf diese Listen werden in einem Array gespeichert. Die Anzahl der Listenelemente ist in $O(m)$, da für jede Kante zwei Einträge (nämlich in der Adjazenzliste beider Endpunkte) gemacht werden müssen. Das Array hat die Größe $O(n)$. Der Gesamtaufwand beträgt also $O(n+m)$. Im worst case geht m mit $O(n^2)$, also ist diese Methode bezüglich des Speicherverbrauches nicht schlechter als die Adjazenzmatrix (Konstanten wie immer vernachlässigt). Allerdings ist es in der Adjazenzmatrix recht schwierig zu prüfen, ob zwei Knoten benachbart sind, da die ganze Liste der adjazenten Knoten durchgegangen werden muss, also $O(n)$ benötigen.

Mehr Infos!

Häufig braucht man in Graphen auch noch Daten, so könnte man bei einem Fernstraßennetz die Länge der Straßen und die Namen der Orte speichern, in einem Schienennetz die Fahrpreise und die zulässige Höchstgeschwindigkeit auf den Strecken, in einer chemischen Strukturformel die Art der Bindung und den Typ der Atome...

gerichtete Graphen

In einem Fußballturnier kann man eine Graph zeichnen, dessen Knoten die teilnehmenden Mannschaften sind. Da wir mit Verlängerung und Elfmeterschießen spielen, kann man zwischen zwei Mannschaften, die gegeneinander gespielt haben, eine Kante einzeichnen, die *gerichtet* ist, und vom Verlierer zum Sieger zeigt (oder umgekehrt, Hauptsache konsistent).

In der Adjazenzmatrix kann man ein Vorzeichen einführen, dass sagt, ob die verbindende Kante bei dem Knotenpaar anfängt oder aufhört:

$$a_{ij} = \begin{cases} +1 & \text{für } v_i \to v_j \in E \\ -1 & \text{für } v_j \to v_i \in E \\ 0 & \textit{sonst} \end{cases}$$

In der Inzidenzmatrix gibt mit dem Vorzeichen an, ob die angegebene Kante am angegebenen Knoten endet oder anfängt. Man spricht davon, dass der Knoten, von dem die Kante ausgeht, der Fuß der Kante und der Knoten, wo die Kante endet, der Kopf der Kante ist.

$$n_{ij} = \begin{cases} +1 & \text{für } v_i \text{ Kopf von } e_j \\ -1 & \text{für } v_i \text{ Fuß von } e_j \\ 0 & \textit{sonst} \end{cases}$$

3.12.6 Graphen durchmustern

Anwendungen:

- Finde alle Knoten, die von $v_0 \in V$ aus erreichbar sind.
- Bestimmen aller Zusammenhangskomponenten des Graphen.
- Für gerichtete Graphen: Bestimmung, ob der Graph zyklisch ist. Für gerichtet Graphen ist es von Bedeutung, ob es nicht nur von a nach b einen

Weg gibt, sondern auch umgekehrt. In diesem Fall heißt der Graph zyklisch, da man dort im Kreis laufen kann.

Stelle Dir die Aufgabe vor: im Physikgebäude haben einige Leute Wände und Türen auch auf den Gängen installiert, und dafür gibt es zwischen einigen Zimmern Türen. Wem noch nicht graust, der sollte bitte mal zur Phyisk kommen, und sich das Gebäude ansehen. Jetzt sagt jemand: „Deine Vorlesung beginnt in 10 Minuten in Hörsaal C“. Das ist dann tatsächlich ein Problem, denn der Hörsaal C ist auf keinem Plan verzeichnet[2], aber um alle Räume zu finden ist einiger Aufwand zu treiben, wahrscheinlich findet man dann auch, wo die Vorlesung wirklich ist.

Beim dem durchsuchen eines Graphen muss man drei Arten von Knoten unterscheiden:

- unbekannte Knoten
- bekannt und abgearbeitet, das heißt, alle Nachbarknoten bekannt.
- bekannt und noch nicht abgearbeitet

Die bekannte Knoten heissen W. Eine Kante $e = u, v$ heißt brauchbar, falls $u \in W$ und $v \notin W$. Unsere Idee ist es, die Menge der bekannten Knoten zu vergrössern, indem wir den brauchbaren Kanten folgen. Die Menge U im folgenden Beispiel enthält alle bekannten und noch nicht abgearbeiteten Knoten, also die, von denen brauchbare Kanten ausgehen (können), und ein Knoten ist `marked`, sobald er bekannt wird, also in W ist. Die Abfrage auf `brauchbar` prüft also, ob ein Ende der Kante ein markierter und das andere Ende ein nicht markierter Knoten ist.

```
search(v0,G)
  forall(v in V) marked[v] = false
  marked[v0] = true
  U.add(v0)
  while(!U.isEmpty())
    w = U.selectElement()
    if exists e={w,v} brauchbar
      marked(v) = true
      T.addEdge(v,w)
      U.add(v)
    else
      U.delete(w)
```

Lemma: Nach Ablauf von `search(`v_0`,`G`)` gilt

1. `marked[v]==true`, genau dann, wenn v von v_0 aus erreichbar
2. T ist die Kantenmenge eines Baumes, der alle von v_0 aus erreichbaren Knoten enthält.

Beweis von (1):

[2] ihn gibt es nämlich nicht

v gegeben mit `marked[v]==true`
v ist von einem v_1 aus erreicht worden, da früher markiert wurde.
v_1 ist von einem v_2 aus erreicht worden, da früher markiert wurde.
usw.
dieser Prozess endet, wenn $v_r = v_0$ ist.

Die Folge $v_0, v_{i-1}, v_{i-2}, \ldots, v_2, v_1, v$ ist ein Weg, der v_0 und v verbindet. Die Gegenrichtung beweist man mit: „Angenommen ist einen Weg $v_0 \ldots v$, aber `marked[v]==false`".

Sei w der letzte markierte Knoten des Weges von v_0 nach v. Dann muss eine Kante von w in Richtung auf v existieren, die noch brauchbar ist, also muss w noch in U sein, also ist der Algorithmus noch nicht fertig.

Beweis von (2):

Für jeden erreichbaren Knoten $v \neq v_0$ gibt es genau eine Kante in T die auf v zeigt, über die v markiert wurde. Insgesamt sind das $n-1$ Kanten, wenn insgesammt n Knoten erreicht wurden.

Herr Felsner ist der Meinung, dass sich Bäume in der Graphentheorie von den weiter oben besprochenen Bäumen unterscheiden. Der eigentliche Unterschied ist aber nur, dass jetzt die Zeichnung nicht unbedingt von oben nach unten geht. Die wichtigen Eigenschaften eines Baums wiederhole ich noch einmal:

- $(n-1)$ Kanten
- zusammenhängend
- kreisfrei

zwei von diesen Eigenschaften reichen aus, damit die Datenstruktur aus n Knoten ein Baum ist, die dritte folgt daraus.

In unserem Fall wissen wir, dass $n-1$ Kanten vorhanden sind, und nach Teil (1) des Lemmas die Struktur zusammenhängend ist.

Wenn man die Effizienz dieses Algorithmus berechnen will, muss man klären, wie die Menge U verwaltet wird, und man brauchbare Kanten findet. Jetzt kommt mal wieder einer dieser coolen Informatikertricks: Wir wollen unter Umständen am Ende sowieso eine Menge mit W haben. Daher hängen wir Queuemäßig an W die neuen Knoten an. Wenn wir einen Knoten untersuchen wollen, liefert uns `selectElement` immer den ersten Knoten, der noch brauchbare Kanten haben kann. Bei dem Aufruf von `deleteElement` wird der Zeiger des ersten interessanten Knotens um eins vorgeschoben.

Insgesammt wird die jede Kante im Graphen auf Brauchbarkeit geprüft, dazu braucht man $O(m)$. Desweiteren werden alle Knoten einmal durch U durchgeschleift, das macht $O(n)$. Daher haben wir einen Gesamtaufwand von $O(n+m)$.

Wir können U zum Beispiel als Queue oder Stack implementieren. Wenn wir eine Queue nehmen, erhalten wir eine sogenannte Breitensuche, bei der wir die kürzesten Wege (das heißt, die wenigsten Schritte) finden, nimmt man einen Stack, ergibt sich der Baum für eine Tiefensuche, der andere Vorteile aufweist (zum Beispiel hat er üblicherweise weniger Kinder pro Knoten).

3.12.7 Kürzeste Wege

$G = (V, E)$ gerichteter Graph. Dazu gibt es $w : E \to \mathbf{R}$, eine Gewichts– oder Längenfunktion.

Ein Weg $p = v_0 \ v_1 \ v_2 \ \dots \ v_l$ ist dann wirklich ein Weg, wenn $(v_i, v_{i+1}) \in E$ für alle $i = 0 \dots l-1$. Die Länge eines Weges ist $w(p) := \sum_{i=0}^{l-1} w((v_i, v_{i+1}))$ Der Abstand zweier Knoten u und v aus E ist definiert als:

$$\delta(u,v) = \begin{cases} \min_{p \text{ Weg von } u \text{ nach } v} w(p) \\ \infty \quad \text{wenn es keinen Weg gibt} \end{cases}$$

Es gibt ein Problem, dass nicht immer ein Minimum existiert, nämlich, wenn w die Frechheit hat, negativ zu werden. Wenn es einen Kreis im Graphen gibt, der insgesamt einen negativen Umfang hat, dann kann man ihn beliebig oft durchlaufen, und erhält immer noch kürzere Wege. Daher werden diese negativen Kreise häufig verboten.

Es gibt verschiedene Arten von Problemen, bei denen man kürzeste Wege braucht:

- Komplette Abstandsmatrix, also $\delta(u,v)$ für alle Knotenpaare
- Abstände von einem Knoten s zu allen anderen, also ein Zeile der Abstandsmatrix
- Abstand $\delta(u,v)$ für nur ein Knotenpaar

Gegeben sei ein Graph $G = (V,E)$, die Abstandsfunktion $w : E \to R$, Startknoten $s \in V$. Gesucht ist $d*(u) = \delta(s,u)$ für $u \in V \setminus s$. Weiterhin interessieren wir uns für einen kürzesten Weg p_u für jedes $u \in V \setminus s$. Es genügt dazu, zu jedem Knoten u einen Vorgängerknoten $\pi(u)$ auf einem kürzesten Weg von s nach u anzugeben, da der kürzeste Weg von s nach u über $\pi(u)$ natürlich den kürzesten Weg von s nach $\pi(u)$ enthält.

Algorithmische Idee: „sukzessives Verbessern".

Starten mit Schätzungen für $d(u)$, die sicher grösser als $d*(u)$ sind. Es soll die ganze Zeit über $d(u) \geq d*(u)$ gelten, während wir $d(u)$ sukzessive verbessern (das heißt verkleinern). Dabei ist es dann unser Ziel, am Ende überall die Gleichheit zu erreichen, also $d* = d$.

Der Schritt des Verbesserns sieht, wenn man eine Kante von u nach v betrachtet, besagt, dass man mit dieser Kante einen Weg von kürzesten s nach u zu einem nach v erweitern kann.

$$d*(v) \leq d*(u) + w(u,v) \leq d(u) + w(u,v)$$

Die rechte Seite ist eine erlaute Schätzung für $d*(v)$, da sie immer noch größer oder gleich $d*(v)$ selbst ist.

```
Relax(u,v)
  if(d(v) > d(u) + w(u,v)
    d(v) = d(u) + w(u,v)
    π(v) = u
```

Jetzt muss man nur noch wissen, wann man fertig ist, und welche Kanten eigentlich überhaupt betrachtet werden müssen. Für das erstere gibt es sogenannte Optimalitätsbedingungen, die bei $d*$ erfüllt sind:

1. für alle $(u,v) \in E$ gilt $d*(v) \leq d*(u) + w(u,v)$

2. (u,v) ist auf einem kürzesten Weg von s nach v ist äquivalent zu $d*(v) = d*(u) + w(u,v)$

Der Beweis zum ersten Punkt ist bereits erfolgt, als wir uns angeguckt haben, dass die Verbesserung durch `Relax` erlaubt ist. Der zweite Punkt ist auch schon ziemlich einfach: $\Rightarrow$ beweist man durch folgende überlegung: Sei p_v ein kürzester Weg von s nach v, dessen vorletzter Knoten u ist. Also ist $q = p$ ohne (u,v) ein Weg von s nach u.

$$\underbrace{w(q)}_{\geq d*(u)} = \underbrace{w(p)}_{=d*(v)} - w(u,v)$$

.

Angenommen, $w(q) > d*(u)$, dann ist $d*(u) + w(u,v) < d*(v)$. Das ist ein Widerspruch zur ersten Behauptung.

Die Gegenrichtung beweist man, in dem man einen kürzesten Weg von s nach u nimmt. Dann ist $w(q) = d*(u)$. Sei p der um (u,v) verlängerte Weg q, dann ist es ein Weg von s nach v. $w(p) = d*(u) + w(u,v) = d*(v)$. Also ist p ein kürzester Weg von s nach u.

Daraus erhält man die Folgerung, dass jeder Algorithmus, der `Relax` so lange anwendet, wie sich die Werte von $d(v)$ verändern lassen, ist ein korrekter Algorithmus zur Bestimmung der kürzesten Wege. Das erste Beispiel ist der Bellman-Ford Algorithmus:

```
for all v ∈ V do d(v) = ∞
d(s) = 0
for i = 1... n-1 do
  for all e ∈ E do Relax(e)
for all e = (u, v) ∈ E do
  if d(v) > d(u) + w(u,v) then
    throw "Negativer Kreis!"
return(d,π)
```

Die erste Zeile ist $O(n)$, die zweite $O(1)$, die dritte sorgt dafür, dass die vierte O(n) mal ausgeführt wird, die an sich schon $O(m)$ mal `Relax` aufruft. `Relax` enthält keine Schleifen oder Rekursionen, ist also von der Laufzeit in $O(1)$. Also brauchen dritte und vierte Zeile zusammen $O(n \cdot m)$. die letzte Schleife wird $O(m)$ mal durchlaufen, und macht eine Abfrage, die in $O(1)$ Zeit erfolgen kann. Also ist sie in $O(m)$. Es dominierert eindeutig die doppelte Schleife in der Mitte, also liegt die Laufzeit des Algorithmus in $O(n \cdot m)$. Im schlimmsten Fall handelt es sich um einen vollständigen Graphen mit $m = O(n^2)$, also kann die Laufzeit $O(n^3)$ werden.

Wie gut, das wir jetzt wissen, wieviel Zeit der Algorithmus verwendet, aber leider wissen wir noch nicht, ob dort wirklich das richtige herauskommt. Das tut es sogar. Der kürzeste Weg zwischen zwei Knoten besteht aus maximal $n - 1$ Kanten, da ein Knoten nur einmal vorkommt. Wäre es günstig, einen Knoten zwei mal vorkommen zu lassen, dann liegt an der Stelle ein negativer Kreis vor, und es gibt keine definierte Abstandsfunktion für diesen Weg.

Jeder der Durchlälufe der Schleife in Zeile 3 findet alle kürzesten Wege, die einen Knoten mehr haben, als die, die man vorher hatte. Am Anfang hat man nur $d(0) = 0$. In der ersten Phase wird an diesen optimalen Weg von s nach s jede Kante angehängt, und die Länge dieser Wege ergibt sich aus der Länge

der Kanten. Im zweiten Schritt stellt man fest, welche Wege es gibt, die zwei Kanten enthalten. Die Wege, die länger sind, als ein schon vorhandener Weg werden durch `Relax` ignoriert. Nach $n-1$ Durchläufen sind alle optimalen Wege mit n oder weniger Knoten bereits betrachtet worden.

Falls es einen negativen Kreis gibt, hört der Algorithmus nicht auf, kürzere Wege zu finden, und hat auch nach $n-1$ Schritten noch mindestens eine Kante, die man in einen optimalen Weg einbauen sollte, damit man etwas besseres erhält. Genau das wird in Zeile 6 für alle Knoten geprüft.

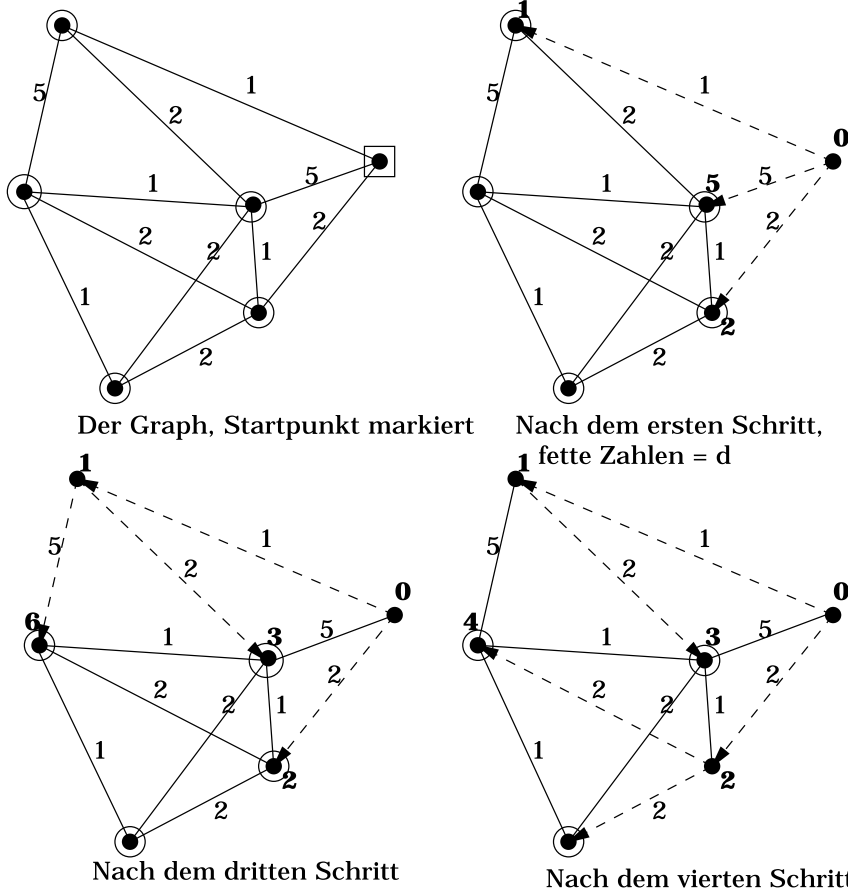
An der Tafel wurde der Algorithmus über einige Ungleichungen bewiesen, von denen ich das Gefühl hatte, man liest sie einmal, kann sie wahrscheinlich noch nachvollziehen, und vier Stunden nach der Vorlesung ist alles wieder vergessen.

3.12.8 Der Dijkstra-Algorithmus

Wenn wir annehmen, dass $w : E \to \mathbf{R}_0^+$ ist, also alle Gewichte größer oder gleich null sind, dann kann man den Algorithmus von Dijkstra (sprich „Deikstra“) verwenden. In jedem Graphen gibt es einen Baum der kürzesten Wege. Jetzt zeigen wir etwas geniales: Man kann unter Kenntnis des Baumes die optimalen Wege sehr schnell berechnen! Wenn man diesen Baum kantenweise durchläuft, dann erhält man sofort die optimalen Wege, da man zu jedem Knoten auf dem optimalen Wege ankommt.

Ganz so gut ist Dijkstra natürlich nicht, denn er hat den Baum der idealen Wege noch nicht gegeben. Der Algo von Dijkstra funktioniert jetzt so:

```
Dijkstra(G,s)
  for all v ∈ V
    d(v) = ∞
  d(s) = 0
  H = V
  while(!H.isEmpty)
    u = node in H with d(u) minimal
    for all (u,v) ∈ E
      Relax(u,v)
    H.delete(u)
  return d
```



Der Graph, Startpunkt markiert **Nach dem ersten Schritt, fette Zahlen = d**

Nach dem dritten Schritt **Nach dem vierten Schritt**

Der Algorithmus ist korrekt, denn wenn u bearbeitet wird, ist $d(u)$ bereits $d*(u)$. Diese Behauptung gilt (das ist so!). Angenommen, es wäre nicht so. Sei v der erste Knoten, für den es nicht gilt. Sei p der kürzeste Weg von s nach v. p enthält eine erste Kantet (x, y), ,mit $x \in H$, $y \notin H$. Wegen der Wahl von v ist $d*(x) = d(x)$. Wegen $\mathsf{Relax}(x, y)$ ist $d(y) \leq d*(x) + w(x, y)$.

$$d(v) \underbrace{\geq d*(x) + w(x,y)}_{=d(y)} + \underbrace{w(\quad) + \cdots + w(\quad)}_{\text{weitereKantenaufPfad}\geq 0}$$

v und y sind in H, und $d(v) \geq d(y)$. Da wir v aus H löschen, muss $d(v) = d(y)$ sein. Also ist $d(v)$ keiner oder gleich der Länge des Weges p. Dieser ist ein optimaler Weg, also ist $d(v) \leq d*(v)$. Da aber für den ganzen Algorithmus gilt, dass $d(v) \geq d*(v)$ muss also $d(v) = d*(v)$ sein. Der Knoten v ist also **nicht** der erste Knoten, der auf einem zu langen Weg erreicht wird.

Wenn man die Mathematik oben vernachlässigt, kann man sich das ganze auch vorstellen: $d(v)$ ist die Länge eines kürzesten Weges von s nach v, der nur Knoten aus H enthält. Würde man einen Knoten benutzen wollen, der nicht in H liegt, ist der Weg zu diesem Knoten bereits weiter als der Weg, der bereits bekannt ist (diese Überlegung klappt natürlich nur, wenn es keine Teilabschnitte negativer Länge gibt, daher die Einschränkung am Anfang)

H wird als Heap implementiert, der zusätzlich die Methode `decreaseKey` unterstützt. Wir können jetzt die Laufzeit bestimmen. Dazu berechnen wir die

Laufzeit der verschiedenen Schritte. Wir müssen alle n Elemente aus dem Heap entfernen, das dauert bekanntlich $n \log n$. Weiterhin wird m mal `Relax` aufgerufen. In Relax wird `decreaseKey` verwendet. Das benötigt $O(\log n)$. Da jede Kante einmal relaxed wird, kommt $O(m \log n)$ hinzu. Mit normalen Heaps benötigt Dijkstras Algorithmus also

$$O((n + m) \log n)$$

Es geht mit irgendwelchen verrückten Datenstrukturen, wie zum Beispiel Fibonacci-Heaps sogar noch schneller (dort ist die Operation `decreaseKey` in konstanter Laufzeit möglich), aber das müssen nur Hauptfach-Informatiker können.

Und zum Abschied wünsche ich euch allen

Frohe FERIEN!

Inhaltsverzeichnis

0.1 Haftungsausschluss . . . 1
0.2 Markennamen . . . 1
0.3 Copyright . . . 1

1 Grundlagen **2**
1.1 Einführung . . . 2
1.1.1 Was ist Java? . . . 2
1.1.2 Wie mache ich etwas mit Java? . . . 2
1.2 Programmieren in Java . . . 4
1.2.1 Kommentare . . . 4
1.2.2 Das Fakultätsprogramm . . . 4
1.2.3 Klassendefinitionen . . . 5
1.2.4 Methodendefinitionen . . . 6
1.2.5 Variablendefinitionen . . . 6
1.2.6 Datentypen . . . 7
1.2.7 Methoden in der API . . . 8
1.2.8 Arithmetik . . . 8
1.2.9 Kontrollstrukturen . . . 11
1.2.10 Ein komplexeres Fakultätsprogramm . . . 13
1.2.11 Klassen und Instanzen . . . 14
1.2.12 Arrays . . . 15
1.2.13 Exceptions . . . 17
1.2.14 noch etwas IO . . . 17
1.3 Objektorientiert Programmieren . . . 18
1.3.1 Ein Beispiel für eine Klasse . . . 18
1.3.2 Konstruktoren . . . 21
1.3.3 Destruktoren . . . 22
1.4 Geltungsbereich . . . 23
1.5 Listen . . . 23
1.6 Brüche . . . 24
1.7 Vererbung und Typecasts . . . 26
1.7.1 Konstruktorverkettung . . . 28
1.7.2 Beschatten und Überschreiben . . . 30
1.8 Verkapselung . . . 33
1.8.1 Pakete . . . 34
1.8.2 Abstrakte Klassen . . . 34
1.9 Interfaces . . . 36

2 Elementare Datenstrukturen **38**
2.1 Der Stack . . . 38
2.1.1 Was ist ein Stack? . . . 38
2.1.2 Methoden des Stacks . . . 39
2.1.3 Was macht man mit einem Stack? . . . 40
2.1.4 Wie realisiert man Stacks? . . . 40
2.2 Queue . . . 43
2.2.1 Watt'n dat? . . . 43
2.2.2 Methoden der Queue . . . 44
2.2.3 Queue im Array . . . 44
2.2.4 Die Variante Dequeue . . . 47

3 Algorithmen **48**
3.1 Laufzeitanalyse . . . 48
3.1.1 Die O-Notation . . . 48
3.1.2 Beispiel: maximale Teilsumme . . . 50
3.1.3 Zwischenergebnisse behalten . . . 51
3.1.4 Divide & Conquer . . . 51
3.1.5 Sortieren . . . 53
3.1.6 Mergesort . . . 54
3.2 Bäume . . . 55
3.2.1 Wozu das ganze? . . . 55
3.2.2 Defintion eines Baumes . . . 56
3.2.3 Methoden des Datentyps Baum . . . 57
3.2.4 Grundlegende Algorithmen auf Bäume . . . 57
3.3 Durchlaufen von Bäumen . . . 59
3.3.1 Preorder . . . 59
3.3.2 Postorder . . . 59
3.3.3 Binäre Bäume . . . 59
3.3.4 Baumdurchlauf Inorder . . . 60
3.3.5 Ausdrucksbäume . . . 61
3.3.6 Binäre Suchbäume . . . 62
3.3.7 Gleichungen über Bäume . . . 64
3.3.8 Darstellung von binären Bäumen . . . 64
3.3.9 Darstellung beliebiger Bäume als binäre Bäume . . . 65
3.3.10 Vollständiger binärer Baum . . . 65
3.4 Folgen, Sequenzen, ... 66
3.5 Priority Queue . . . 66
3.5.1 abstrakter Datentyp . . . 66
3.5.2 Sortieren mit PQs . . . 66
3.5.3 Implementation mit Folgen . . . 66
3.5.4 Implementation mit Heaps . . . 67
3.5.5 Implementierung . . . 67
3.5.6 Heapsort . . . 68
3.6 Sortieren . . . 70
3.6.1 CountingSort . . . 70
3.7 Dictionaries . . . 71
3.7.1 Methoden eines Wörterbuches . . . 71
3.7.2 Beispiel . . . 72
3.7.3 Geordnete Wörterbücher . . . 72

3.7.4 Implementierungen 72
3.7.5 Laufzeit im Suchbaum 74
3.8 AVL-Bäume 75
3.9 Optimale Suchbäume 78
3.10 Dynamisches Programmieren 81
3.11 Fibonacci-Zahlen 81
3.11.1 Aufbrechen der Rekursion 82
3.11.2 Bottom-Up-Aufbau 82
3.11.3 Longest common subsequence 82
3.12 Graphen 83
3.12.1 Was ist ein Graph? 83
3.12.2 Und so sieht das der Mathematiker 83
3.12.3 Begriffe 84
3.12.4 Spezielle Graphen 85
3.12.5 Algorithmen für Graphen 85
3.12.6 Graphen durchmustern 87
3.12.7 Kürzeste Wege 89
3.12.8 Der Dijkstra-Algorithmus 91