Informatik II
Algorithmen und Datenstrukturen

Klaus Wachtler

29. März 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Text wird alles gesammelt, was über das Skript [v.Helden] hinausgeht.

Form und Inhalt sind:

© 2003 Klaus Wachtler
Breidingstr. 17
29614 Soltau
Tel.: $\bar{0}$5191/70271
email: AnyWare@Wachtler.de
http://www.wachtler.de

Für Verbesserungsvorschläge, Fehlerkorrekturen und ähnliches bin ich jederzeit dankbar.

Letzte Änderungen:

- 20.05.2002 kleinere Korrekturen in **6.1 Nutzen von Sprachmitteln** auf Seite 75
- 20.05.2002 Musterlösung zu einfachen Listen (**A.2.5 Einfach verkettete Liste** auf Seite 133)
- 21.05.2002 bisher war in der Musterlösung für den *Größten Gemeinsamen Teiler* statt der Ausgabe der Anzahlen der ausgeführten Anweisungen nochmals die Laufzeit angegeben (hat aber bisher offenbar niemand bemerkt).
- 21.05.2002 Musterlösung für `Liste eines beliebigen Typs` (C++)
- 23.05.2002 Rechtschreibkorrekturen

- 23.05.2002 Aufgabenstellung **Einfaches Integral** aufgepeppt, Musterlösung dazu (C++ mit Funktionszeigern, Java mit abstrakter Basisklasse ebenso wie mit funktionalen Objekten)
- 23.05.2002 Musterlösung zu **Geschlossenes Hashing** (Java und C++)
- 24.05.2002 C++-Lösungen unter Windows getestet (Visual C++ 6.0)
- 30.05.2002 Musterlösungen zu **A.1.10 Maximum einer Funktion** auf Seite 99 (Java und C++)
- 11.06.2002 Evolutionsstrategien, genetische Algorithmen
- 18.06.2002 Stringvergleiche Knuth,Morris, Pratt
- 20.06.2002 Stringvergleiche Boyer-Moore
- 11.05.2003 Musterlösungen für offenes Hashing in Java; C++-Version der einfach verketteten generischen Liste (**A.2.6.2 C++-Version** auf Seite 161) überarbeitet; C++-Version des geschlossenen Hashing überarbeitet (`HashEintrag` jetzt Unterklasse, Wertsemantik)

Inhalt der Vorlesung sind hauptsächlich die wichtigsten Algorithmen zur Verwaltung von Daten innerhalb von Programmen, sowie die zugehörigen Datenstrukturen.

Die vorgestellten Algorithmen werden bezüglich ihrer Effizienz bewertet.

Soweit wird dies auch in [v.Helden] behandelt.

Darüber hinaus stelle ich auch noch einige klassische Algorithmen vor, die nicht direkt zum Handhaben von Daten nötig sind, aber meiner Meinung nach einem Informatiker bekannt sein sollten. Solche Erweiterungen, die über [v.Helden] hinaus gehen, stehen dann in dem vorliegenden Skript.

Wenn mich der Wahn packt, erweitere ich das Skript vielleicht mal in mittlerer Zukunft dergestalt, daß es eigenständig wird und von [v.Helden] unabhängig. Das wird aber sicher nicht mehr in diesem Semester sein.

Nicht zum Inhalt dieses Skripts und der Vorlesung zählen weite Bereiche, die zu den Algorithmen gehören würden: Numerik, Seminumerik, grafische Verfahren, und vieles andere.

Voraussetzungen:

- virtueller Speicher, lokale Variablen, freier Speicher
- Compiler, Interpreter
- Programmiersprachen: Maschinensprache, Assembler, C, C++, Java, BASIC, FORTRAN, Postscript, LISP, smalltalk, Perl, Python, C#
- virtueller Speicher, lokale Variablen, freier Speicher
- Rekursion
- Programme, Prozesse
- Bibliotheken, statisches/dynamisches Linken

Kapitel 2

Allgemeines

Ein Algorithmus ist eine Rechen- oder Verarbeitungsvorschrift, und beschreibt (möglichst) exakt einen Lösungsweg für ein bestimmtes Problem.

Die Formulierung eines Algorithmus steht damit weit vor dem eigentlichen Programmieren (*Codieren*). Ein Algorithmus ist prinzipiell noch unabhängig von seiner konkreten Ausformulierung, also der Implementierung in einer Programmiersprache.

Dabei kann es für ein gegebenes Problem möglicherweise einen oder mehrere oder keinen (bekannten?) Algorithmus geben.

2.1 Herkunft des Begriffs Algorithmus

Der Begriff „Algorithmus" geht zurück auf den Namen des arabischen Gelehrten **Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî**[1] aus dem neunten Jahrhundert ([Knuth 1], S. 1, [Knuth Sel.]und [Ziegenbalg].).

Sein Buch (*„Über die indischen Zahlen"*) etwa aus dem Jahr 820 trug die in Indien entwickelte Technik des Rechnens mit (nichtnegativen) Dezimalzahlen und einer Ziffer für die 0 in die arabische Welt. Dabei wurde auch die Schreibweise für die

[1] *Vater von Ja'farm, Mohammed, Sohn Moses', aus Khowârizm* (teilweise auch als **alchwarismi** oder ähnlich geschrieben); Khowârizm existiert heute noch unter dem Namen Khiva im Westen von Usbekistan. Vom Titel eines seiner Bücher (**Kitab al jabr w'al-muqabala**) stammt auch der Begriff „Algebra" ab.

Ziffern selbst übernommen[2].

Im elften Jahrhundert wurde dieses Buch in Spanien endlich ins Lateinische übersetzt (und machte damit auch in Europa die indische Rechentechnik indirekt nutzbar). Dieses Buch begann mit den Worten „Dixit Algoritmi…“ (*also sprach Algoritmi…*); die Herkunft des Autors (al-Khowârizmî) wurde offensichtlich als sein Name mißbraucht und zum Ursprung des Begriffs *Algorithmus*.

Durchgesetzt hat sich die Rechnung mit Dezimalzahlen in Europa erst um 1600. Die dabei verwendeten Regeln wurden als *Algorithmen* bezeichnet, und erst im Laufe der Zeit wurde der Begriff für alle Rechenverfahren verwendet.

2.2 Anforderungen an einen Algorithmus

Je nachdem wen man nach den Anforderungen an einen Algorithmus fragt, bekommt man unterschiedliche Antworten.

Relativ unstrittig sind die folgenden Punkte:

- Beschreibung eines durchzuführenden Verfahrens durch Angabe seiner Einzelschritte
- Präzise Formulierung (*Bestimmtheit, definiteness*):
  - Eindeutiger Startpunkt
  - Eindeutiger Endpunkt (entweder die Lösung, oder die Aussage daß das Problem nicht lösbar ist)
  - Jeder Schritt muß ein genau definiertes Verhalten zeigen[3].
  - Für jeden Schritt (außer dem Ende natürlich) ist ein eindeutiger Nachfolgeschritt definiert.

Teilweise werden darüber hinaus folgende Anforderungen an einen Algorithmus gestellt:

[2] Aus dem Arabischen kamen die zehn Ziffern nach Europa, und heißen deshalb *arabische Ziffern*, auch wenn sie eigentlich aus Indien stammen. Noch früher, nämlich in Mesopotamien (Zweistromland, heute etwa Irak) gab es auch bereits ein Schreibzeichen für die Null (allerdings im 60-System statt im Dezimalsystem), nachdem dort lange einfach *kein Zeichen*, also ein Leerzeichen, stattdessen verwendet wurde, was aber offenbar verwirrend war.

[3] spannend ist hier die Frage, ob durch diese Forderung der Aufruf eines Zufallszahlengenerators möglich ist…

- Korrektheit
- Endlichkeit:
  - der Beschreibung des Algorithmus als solcher (*statische Endlichkeit*)
  - sowohl jedes einzelnen Schrittes
  - als auch der gesamten Anweisungsfolge (dynamische Endlichkeit; ein Betriebssystem oder manches Programm mit interaktiver Benutzeroberfläche fällt beispielsweise nicht darunter)
- Verwendung einer Eingabe (*input*)
- Erzeugen einer Ausgabe (*output*)
- Effizienz (*effectiveness*, siehe **2.5 Effizienz von Algorithmen** auf Seite 15)
- Jeder Schritt soll eine einfache Aktion sein, also irgendwie *elementar*

2.3 Wie formuliert man einen Algorithmus?

Ebenso wie es viele Sprachen gibt, in denen man ein Bier bestellen kann, oder mehrere Programmiersprachen um ein Programm zu erstellen, gibt es auch mehrere Möglichkeiten, einen Algorithmus zu formulieren.

Die wichtigsten sind:

- Mehr oder weniger formalisierte textliche Beschreibung.

  Als Beispiel hier der Euklidsche Algorithmus[4] aus [Knuth 1], S. 2, mit dessen Hilfe der größte gemeinsame Teiler (ggT) zweier gegebener positiver ganzer Zahlen m und n bestimmt werden kann:

[4] *Euklid von Alexandria* (ca. 365-300 v.Chr.) gilt als Autor der 13 Bücher „Die Elemente“, die das gesamte damalige mathematische Wissen zusammenfassen sollten und bis ins 19. Jahrhundert unverändert als Lehrbücher verwendet worden sein sollen ([Ziegenbalg].)

Algorithm E (*Euclid's algorithm*). Given two positive integers m and n, find their greatest common divisor, i.e. the largest positive integer which evenly divides both m und n.

E1. [Find remainder.] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)

E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Interchange.] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1. ▮

Abbildung 2.1: Euklidscher Algorithmus aus [Knuth 1]

- Grafische Darstellung als Flußdiagramm (Programmablaufplan)

  Dabei wird jeder Schritt eines Algorithmus in einem Rahmen dargestellt, wobei der Rahmen je nach Art des Schritts (auszuführende Anweisung, Fallunterscheidung etc.) unterschiedliche Formen annehmen kann. Die einzelnen Schritte werden sinnvoll in einem Bild angeordnet und entsprechend ihrer Ausführungsreihenfolge mit Linien oder besser Pfeilen verbunden.

  Das obige Beispiel des Euklidschen Algorithmus sieht bei [Knuth 1] so aus:

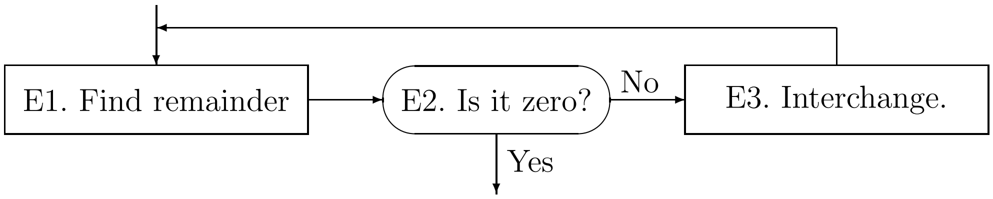


Abbildung 2.2: Flußdiagramm Euklidscher Algorithmus in Notation gem. Knuth

Mithilfe solcher Flußdiagramme (flow chart) lassen sich praktisch alle Algorithmen darstellen, weshalb sie aber auch inzwischen weitgehend verpönt sind; denn durch die freie Platzierung der einzelnen Elemente und die Möglichkeit, Pfeile beliebig zu ziehen, kann man einerseits sehr unübersichtliche Algorithmen darstellen, andererseits auch alleine die Darstellung eines vielleicht vernünftigen Algorithmus unübersichtlich gestalten („Spaghetticode").

Welche Rahmen um die einzelnen Symbole gezeichnet werden, wird etwas uneinheitlich gehandhabt. Bei den Büchern von Knuth werden, wie in Abbildung 2.2 gezeigt, um die eigentlichen Anweisungen gerade Rechtecke gezeichnet, und Fallunterscheidungen stehen in einem zigarrenförmigen abgerundeten Rahmen.

Üblicher ist aber eine andere Kennzeichnung, nämlich ebenfalls Rechtecke um Anweisungen, aber auf einem Eck stehende Rauten für Fallunterscheidungen. Abgerundete Rahmen werden für den Start- und einen oder gegebenenfalls mehrere Endpunkte des Algorithmus verwendet. Der Euklidsche Algorithmus sieht dann so aus:

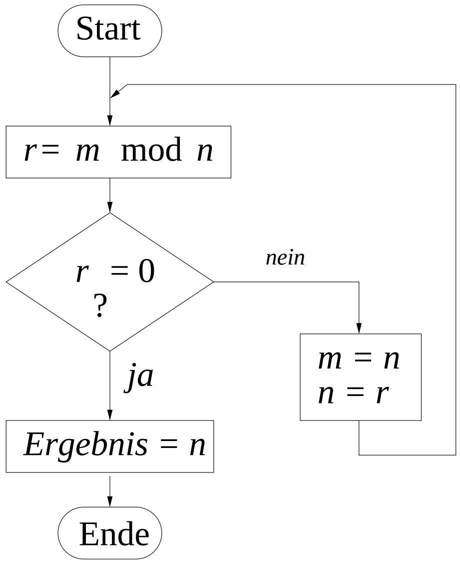


Abbildung 2.3: Flußdiagramm Euklidscher Algorithmus in Standardnotation

Bei solchen Flußdiagrammen gibt es für Schleifen keine besonderen Symbole; man muß Initialisierung, Schleifenbedingung, Schleifenrumpf und gegebenenfalls Reinitialisierung (Weitersetzen des Schleifenzählers beispielsweise) aus den gezeigten einzelnen Elementen zusammensetzen. Alleine dadurch wird die Darstellung schnell unübersichtlich.

Die Symbole sind in DIN 66001 genormt (mit weiteren Symbolen, die hier nicht erwähnt werden, beispielsweise zur Ein- und Ausgabe).

- Struktogramme (Nassi-Shneiderman-Diagramme) nach DIN 66261.

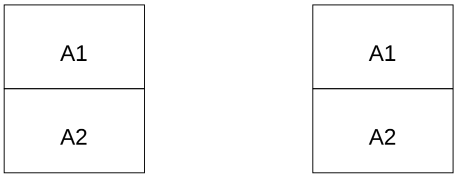
  Hier werden rechteckige Grundelemente für jeweils eine Anweisung, eine Fallunterscheidung, eine Schleife etc. direkt so aneinandergefügt, daß sich für den dargestellten Algorithmus insgesamt ein rechteckiges Diagramm ergibt. Der Start eines Diagramms ist immer am oberen Rand; das Ende zumindest am unteren Rand (oder in einem besonderen Abbruchelement irgendwo innerhalb des Diagramms).

  In einigen der Bauelemente kommen in der „normalen“ Darstellung schräge Linien vor. Da dies manchmal schlecht zu zeichnen ist (beispielsweise beim Zusammensetzen aus Buchstaben), gibt es jeweils eine Ersatzdarstellung ohne schräge Linien. Diese Ersatzdarstellung ist insbesondere auch geeignet, direkt im Quelltext als Kommentar angelegt zu werden!

Es gibt folgende Grundelemente (jeweils links in Normal- und rechts in Ersatzdarstellung):

– Anweisung: Nach Ausführung der Anweisung A wird das darunterliegende Element bearbeitet.

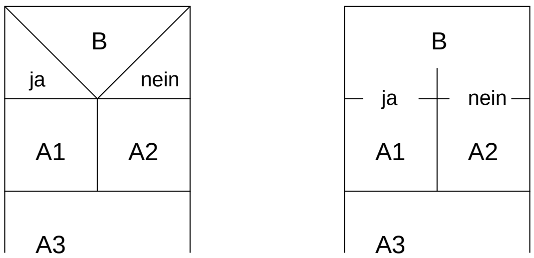
– Anweisungsfolge: Es werden nacheinander die Anweisungen A1, A2 und so fort abgearbeitet.



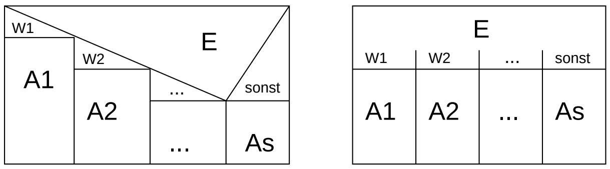
Eine Anweisungsfolge kann überall eingesetzt werden, wo auch eine einfache Anweisung erlaubt ist.

– Fallunterscheidung: Die Bedingung B wird ausgewertet; je nach Ergebnis wird dann entweder die Anweisung (oder Anweisungsfolge) A1 oder die Anweisung A2 ausgeführt.

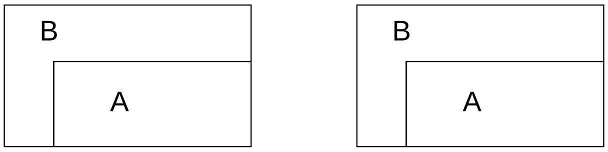
Falls vorhanden, wird anschließend die Anweisung A3 ausgeführt.



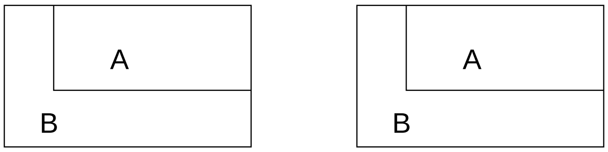
– Mehrfachentscheidung: Der Ausdruck E wird ausgewertet. Falls er den Wert W1 ergibt, wird die darunterliegende Anweisung(sfolge) A1 ausgeführt; analog falls der Ausdruck W_i ergibt, wird A_i ausgeführt. Trifft keiner der angegebenen Werte zu, wird As ausgeführt.



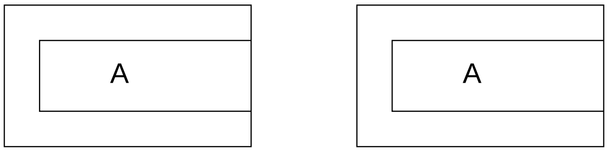
– Abweisende Schleife: Es wird die Bedingung B geprüft. Falls B erfüllt ist, wird A ausgeführt, und anschließend B erneut geprüft, und so weiter.



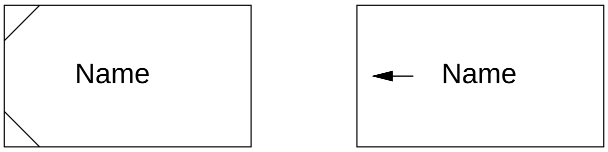
– Nichtabweisende Schleife: Zuerst wird A ausgeführt. Dann wird die Bedingung B geprüft. Falls B erfüllt ist, wird A erneut ausgeführt, und so weiter.



– Unbedingte Schleife: A wird wiederholt ausgeführt.



– Abbruch: Der umgebende Block Name wird beendet.



Nach der Abarbeitung eines Elements wird das darunterliegende, gleich breite Element verwendet. Eine Ausnahme davon sind die einfache Fallunterscheidung und die Mehrfachauswahl. Hier wird jeweils nur einer der darunterliegenden Teilblöcke weiterverfolgt.

Mit einem Nassi-Shneiderman-Diagramm lassen sich nicht alle Algorithmen darstellen; es ist nämlich nicht möglich beliebige Sprünge von einer Stelle eines Algorithmus zu einer anderen Stelle auszudrücken.

Da dies aber ohnehin als sehr unelegant gilt, werden Struktogramme gerne verwendet: elegant formulierte Algorithmen (also solche ohne wilde Sprünge) lassen sich damit sehr gut lesbar darstellen. Zudem ist die Umsetzung einer solchen Darstellung in jede moderne Programmiersprache leicht und elegant.

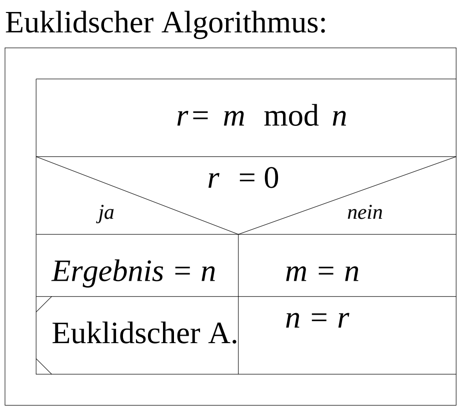


Abbildung 2.4: Struktogramm Euklidscher Algorithmus

```
Euklidscher Algorithmus:
   +----------------------------------------+
   |                                        |
   | +--------------------------------------+
   | | r = m mod n                          |
   | +--------------------------------------+
   | |              r = 0 ?                 |
   | |                  |                   |
   | |----- ja ---------+---- nein --------+
   | |                  |                   |
   | | Ergebnis = n     | m = n             |
   | +------------------+-------------------+
   | | <--Euklidsch. A. | n = r             |
   | +------------------+-------------------+
   |                                        |
   +----------------------------------------+
```

Abbildung 2.5: Struktogramm Euklidscher Algorithmus (Ersatzdarstellung)

2.4 Rekursion

Weil die *Rekursion* in vielen Algorithmen eine zentrale Rolle spielt, soll hier kurz darauf eingegangen werden.

Unter Rekursion versteht man die Selbstbezüglichkeit eines Algorithmus oder eines (Teil-)Programms.

Das übliche Einstiegsbeispiel dazu: die Berechnung der Fakultät einer natürlichen Zahl n.

Die Fakultät einer Zahl ist das Produkt aller Zahlen von 1 bis einschließlich der gegebenen Zahl n, und wird mit einem nachgestellten Ausrufezeichen markiert:

$$n! = 1 \times 2 \times 3 \ldots \times (n-1) \times n \tag{2.1}$$

(Die Fakultät von 0 hat per Definition den Wert 1.)

Neben dieser anschaulichen Definition kann man aber auch Teilausdrücke zusammenfassen: $2! = (1) \times 2$, also $2! = 1! \times 2$. Analog ist $3! = (1 \times 2) \times 3$, also $3! = 2! \times 3$, und so weiter; im allgemeinen Fall also $n! = (1 \times 2 \times \ldots \times (n-1)) \times n$ oder $n! = (n-1)! \times n$.

Damit hat man schon eine rekursive Definition der Fakultät:

$$n! = (n-1)! \times n$$

die allerdings nicht für alle natürlichen Zahlen gilt, sondern nur für $n \geq 1$ (nicht für $n = 0$)

Um jetzt die Definition auf alle natürlichen Zahlen zu erweitern, braucht man eine Art Fallunterscheidung:

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ (n-1)! \times n & \text{ansonsten} \end{cases} \tag{2.2}$$

Die erwähnte Selbstbezüglichkeit drückt sich darin aus, daß zur Definition der Fakultät bereits die Fakultät verwendet wird (im Ausdruck $(n-1)! \times n$).

Typischerweise benötigt eine rekursive Definition immer einen Ankerpunkt, der nicht rekursiv ist (hier für den Fall $n = 0$).

Von manchen Algorithmen gibt es nur eine nichtrekursive Version, von manchen nur eine rekursive Version, es kann auch beide Versionen nebeneinander geben. Welche man dann auswählt, hängt hauptsächlich von der Effizienzbetrachtung ab.

Wenn beide Versionen existieren, dann wird die nichtrekursive Variante meist *iterativ* genannt, weil in aller Regel jede Selbstbezüglichkeit durch einen Iteration, also eine Wiederholung eines Rechenschritts ersetzt wird (hier eine wiederholte Multiplikation $1 \times 2 \times 3 \ldots \times (n-1) \times n$).

Die Umsetzung beider Versionen in ein Programm ist direkt möglich. Der iterativen Version aus Gleichung 2.1 entspricht folgendes Programmfragment:

```
int fakultaet_iterativ( int n )
{
  int i;
  int ergebnis;
  ergebnis = 1;
  for( int i=1; i<=n; i++ )
  {
    ergebnis *= i;
  }
  return ergebnis;
}
```

Dementsprechend kann die rekursive Variante aus Gleichung 2.2 so implementiert werden:

```
int fakultaet_rekursiv( int n )
{
  if( n==0 )   return 1;
  else         return fakultaet_rekursiv( n-1 )*n;
}
```

Die rekursive Implementierung ist -wie so oft- bestechend kurz, einfach und prägnant, und kommt mit weniger Variablen aus. Trotzdem ist sie langsamer, und benötigt mehr Stackspeicher zur Laufzeit.

Nicht alle Programmiersprachen unterstützen rekursive Algorithmen; beispielsweise in FORTRAN und BASIC ist es nicht zulässig, Unterprogramme oder Funktionen direkt oder indirekt sich selbst aufrufen zu lassen.

Zumindest alle Sprachen mit einem modernen Konzept dagegen ermöglichen rekursives Programmieren: ALGOL, Pascal, C/C++, Java, und so weiter.

Wie ein rekursiver Funktionsaufruf technisch gesehen funktioniert, geht über dieses Skript hinaus. Wer sich dafür interessiert, muß sich anderweitig schlau machen. Sinnvoll ist das auf jeden Fall, weil ein Verständnis des Stacks vieles verständlicher macht, und vor allem heftige Programmierfehler vermeiden hilft (*stack overflow*, Sicherheitsprobleme durch *buffer overrun*).

2.5 Effizienz von Algorithmen

Ein wesentliches Merkmal eines Algorithmus ist seine Effizienz, insbesondere wenn man für ein gegebenes Problem mehrere Algorithmen zur Auswahl hat und sich sinnvoll entscheiden muß.

Die Effizienz kann aber nicht einfach als ein Kriterium betrachtet werden, sondern besteht letztlich aus:

- Laufzeiteffizienz: wie schnell kann eine Implementierung sein?
- Speichereffizienz; wieviel Speicher benötigt eine Implementierung zur Laufzeit?
- „kognitive Effizienz": darunter versteht man, wie einfach der Algorithmus geistig zu erfassen ist, und wie natürlich er das Problem löst. Dieser Punkt ist nicht unwichtig, weil die fehlerfreie Implementierung vom Verständnis des Algorithmus abhängt, ebenso wie der Zeitaufwand für die Implementierung.

  Da man diesen Punkt aber nicht in Zahlen fassen kann, wird er meist einfach unterschlagen (in diesem Skript im wesentlichen auch).

  Ein abschreckendes Beispiel in diesem Sinne ist in **B.1 Gaußscher Algorithmus zur Bestimmung des Ostersonntags** auf Seite 217 angegeben.

Laufzeit- und Speichereffizienz sind genau genommen nicht alleine eine Eigenschaft des Algorithmus, sondern werden auch von der konkreten Implementation und der zugrundeliegenden Hardware beeinflußt (sowie von den konkreten Eingabedaten), zumindest sofern man unter Laufzeit tatsächlich eine Zeit in Sekunden versteht, beziehungsweise den Speicherbedarf in Byte ausdrückt.

Erstrebt wird dennoch eine Bewertung unabhängig von Implementation und Hardware. Dies wird erreicht, indem man die Zeit nicht mehr in Sekunden angibt, sondern stattdessen die benötigte Anzahl Rechenschritte (gegebenenfalls aufgeschlüsselt nach Anzahl Vergleiche, Anzahl Additionen etc.), und anstelle des tatsächlichen Speicherbedarfs eine Anzahl Speicherplätze unbestimmter Größe für die Variablen berechnet; dazu später mehr.

In vielen Algorithmen ist eine irgendwie geartete Problemgröße als freier Parameter enthalten („Wieviele Elemente sind zu sortieren?“, „Aus wievielen Elementen soll ein bestimmtes herausgesucht werden?“, „Wie groß ist eine Zahl, die in Primzahlen zerlegt werden soll?“). Diese Problemgröße geht in aller Regel stark in das Laufzeit- und Speicherverhalten ein. Deshalb wird die Effizienz in Abhängigkeit dieser Problemgröße angegeben.

Die kognitive Effizienz ist auf den ersten Blick alleine vom Algorithmus abhängig, weil die Implementation ja erst anschließend kommt. Dennoch ist sie von der jeweiligen Denkweise des Lesers stark beeinflußt, und die wiederum ist ganz klar abhängig von der Programmiersprache (oder dem dahinter stehenden *Paradigma*), mit der der Leser vertraut ist: ein typischer Algorithmus aus der linearen Algebra wird von einem langjährigen Fortranprogrammierer leichter erfaßt werden als von einem Programmierer, der mit Smalltalk aufgewachsen ist; bei einem rekursiven Algorithmus zum Abarbeiten einer Liste mag es umgekehrt sein[5].

2.5.1 Laufzeiteffizienz

Um die Laufzeiten von Algorithmen vergleichen zu können, ist es meist reizlos, alle Algorithmen zu implementieren und einfach die Laufzeiten zu messen, da sowohl die Codierung als auch das zugrundeliegende System (Hardware, Wahl der Sprache, Optimierungen des Compilers und so weiter) einen enormen Einfluß auf die Laufzeiten haben können; vergleiche dazu auch die Laufzeitunterschiede zwischen Java und C in der Beispiellösung **A.2.1 Größter gemeinsamer Teiler** auf Seite 100.

Dazu kommt, daß man natürlich auch gerne die Arbeit vermeidet, etliche Algorithmen erst zu implementieren, um anschließend doch nur einen davon zu verwenden.

[5]…woran man auch gut sehen kann, daß eine Sprache für einen gegebenen Algorithmus mehr oder weniger geeignet sein kann. Üblicherweise tendieren Programmierer dazu, ihre Lieblingssprache zu verwenden; was man kennt das liebt man. Erstrebenswert wäre aber natürlich, die Wahl der Sprache dem Problem und dem Algorithmus anzupassen.

Gesucht wird also eine Möglichkeit, Aussagen zum Laufzeitverhalten von Algorithmen ohne deren Implementation zu machen.

Der erste Ansatz wäre, für einen gegebenen Satz von Eingaben einfach alle Anweisungen des Algorithmus anhand seiner Definition zu simulieren und alle ausgeführten Anweisungen zu zählen.

Das Zählen kann

- einfach über alle Anweisungen erfolgen
- aufgeschlüsselt nach verschiedenen Anweisungen (ganzzahlige Additionen/Subtraktionen, Gleitkommaadditionen/-Subtraktionen, ganzzahlige Multiplikationen, ganzzahlige Divisionen/Modulobildung, Unterprogrammaufrufe etc.)
- vereinfacht nur für bestimmte Grundoperationen (Schleifendurchläufe beispielsweise oder bestimmte arithmetische Operationen); dafür werden sogenannte „kritische“ Operationen gezählt, und alle anderen ignoriert.

erfolgen.

Eine nach bestimmten Gruppen aufgeschlüsselte Zählung kann deshalb interessant sein, weil nicht alle Anweisungstypen gleich aufwendig sind. Es gibt bestimmte „teure“ (also langsame) und dagegen relativ „billige“ (schnelle) Operationen.

Die Details sind systemabhängig, aber allgemein kann man sagen:

- So ziemlich das langsamste (von einer Benutzereingabe abgesehen) ist ein Zugriff auf Dateien. Dabei ist der binäre Direktzugriff noch deutlich schneller als das Lesen oder Schreiben von Text.
- Gleitkommaoperationen sind prinzipiell langsamer als ganzzahlige
- Operationen mit hoher Genauigkeit beziehungsweise großer Stellenzahl sind langsamer als mit kleiner Stellenzahl ()
- Vergleiche, Inkrementieren, Dekrementieren gehen schnell; Multiplikationen sind wesentlich langsamer; Divisionen nochmals viel langsamer als Multiplikationen
- Mathematische Funktionen wie Wurzelziehen, Logarithmenbildung und vor allem Winkelfunktionen sind generell langsamer als elementare algebraische Operationen

- ein Unterprogrammaufruf (oder Funktionsaufruf) ist mindestens so aufwendig wie einige elementare Operationen, je nach Sprache, Anzahl und Art der Parameter kann ein Unterprogrammaufruf aber auch schnell ein Vielfaches der elementaren Operationen kosten.

Will man also eine recht feine Analyse eines Algorithmus durchführen, dann muß man die durchlaufenen Schritte mit einem Faktor gewichten, der den Aufwand beziehungsweise die Laufzeit des einzelnen Schritts reflektiert.

Auch wenn diese Gewichtung theoretisch vom konkreten Zielsystem abhängt, kann man doch davon ausgehen, daß ein Algorithmus schneller ist als ein zweiter, wenn er eine Gleitkommadivision durch wenige ganzzahlige Additionen ersetzen kann, wenn er ansonsten gleichwertig ist.

Ein Beispiel wäre folgende Optimierung: Ein gegebener Algorithmus benötigt einen Vergleich eines Wertes `x` mit der Wurzel eines Wertes `y`:

```
|      ...       |
+----------------+
| x < sqrt(y) ?  |
|       |        |
+- ja --- nein  -+
|       |        |
| ...   |   ...  |
+----------------+
|      ...       |
```

Falls die berechnete Wurzel nur für diesen Vergleich benötigt wird, und nicht mehr im folgenden Teil des Algorithmus, dann kann man die Berechnung beschleunigen:

```
|      ...       |
+----------------+
|   x*x < y ?    |
|       |        |
+- ja --- nein  -+
|       |        |
| ...   |   ...  |
+----------------+
|      ...       |
```

Grund: wohl auf allen gängigen Systemen ist eine Multiplikation deutlich schneller als die Berechnung einer Wurzel.

2.5.2 Speichereffizienz

rekursiv: Stack wird entsprechend der Rekursionstiefe belastet

iterativ: i.d R. wesentlich sparsamer

siehe [v.Helden] S. 5ff.

Kapitel 3

Container und zugehörige Algorithmen

3.1 Allgemeines

Containertypen: (Nutz-)Daten + Verwaltungsinformationen zusammengefaßt, aber getrennt betrachtet; siehe **6 Umsetzung von Algorithmen** auf Seite 75.

3.1.1 Referenzsemantik, Wertsemantik

Container können eine von zwei grundsätzlich unterschiedlichen Verhaltensweisen aufzeigen:

- Bei der *Wertsemantik* legt der Container von allen zum Einfügen angegebenen Nutzdaten jeweils eine Kopie an, und speichert den kopierten Inhalt. Eine danach erfolgte Änderung der zum Einfügen übergebenen Nutzdaten durch den Aufrufer wirkt sich **nicht** auf den Inhalt der Daten im Container aus.

- Wenn der Container dagegen eine *Referenzsemantik* besitzt, werden effektiv nur Referenzen auf die einzufügenden Objekte gespeichert. Eine Änderung der zum Einfügen übergebenen Nutzdaten wirkt sich auch nach dem Einfügen noch auf den Inhalt des Containers aus.

Bei vielen Anwendungen mag es egal sein, welche der beiden Varianten vorliegt; dann ist die Referenzsemantik meist effektiver (weil weniger Daten bewegt werden müssen).

Aber bei vielen Problemstellungen kann sowohl die eine als auch die andere Semantik gewünscht sein. Deshalb sollte eine aus Anwendersicht „vollständige“ Bibliothek von Containern eigentlich beides anbieten.

In Java ist in der API von Sun durchweg eine Referenzsemantik realisiert; eine Umstellung auf Wertsemantik ist aufwendig und in der Regel sehr uneffektiv.

In der STL (Teil der Standardbibliothek von C++, der Container anbietet) dagegen ist durchweg eine Wertsemantik umgesetzt. Dies hängt hauptsächlich damit zusammen, daß C++ keine *garbage collection* kennt, und dadurch bei einer Referenzsemantik Verweise auf Objekte gespeichert würden, deren Lebensdauer vom Container nicht erkennbar oder gar kontrollierbar wäre. Um also Verweise auf nicht mehr existierende Objekte zu vermeiden, wird hier generell mit einer Wertsemantik gearbeitet. Bei Bedarf kann man mit relativ wenig Aufwand eine Referenzsemantik bauen (siehe [KW-C++]); dann muß man allerdings selbst den Überblick über die Lebensdauer der Objekte haben.

Beispiele für jede der beiden Varianten finden sich bei den Musterlösungen in **A.2.6 Einfach verkettete Liste eines beliebigen Typs** auf Seite 152.

3.2 Feld

Feld je nach Sprachen nur einheitlicher Datentyp oder variabel

3.3 Liste

Liste: einfach verkettet, doppelt verkettet, Einfügen/Entfernen an einem oder an beiden Enden (stack, queue, double ended queue), zirkuläre Liste; siehe [v.Helden] S. 38ff. und S. 64ff.

3.4 Map, Set

Set, Map (jeweils unique oder multi)

3.5 Hashing

Hashtabellen: offenes/geschlossenes Hashing, siehe [v.Helden] S. 70ff.

3.6 Bäume

Baum: leer, oder ein Knoten mit zwei (Teil-)Bäumen (also rekursiv definiert) beide Teilbäume leer: Blatt, sonst innerer Knoten, mehr als zwei Teilbäume möglich, Bayerbäume; siehe [v.Helden] S. 91ff.

algebraischer Ausdruck

Binärbaum, sortiert: Suchbaum

Traversierung (Iterator): preorder (Wurzel, links, rechts), postorder (links, rechts, Wurzel), inorder (links, Wurzel, rechts), levelorder

Baumtiefe: Anzahl der Kanten; bei N Knoten: $log_2(N) \dots N-1$

balanciert, entartet, ausgeartet, degeneriert, vollständig

3.6.1 Ausgeglichene Bäume

Wenn man keine weiteren Vorkehrungen trifft, kann ein Baum (beispielsweise ein binärer Suchbaum) im besten Fall halbwegs ausgeglichen sein (im Hinblick auf die Pfadlänge von der Wurzel zu den einzelnen Blättern); dann kann man bei N Knoten (Elementen) schlimmstenfalls $\log N + 1$ Vergleiche erwarten, um ein Element zu finden (was einem Laufzeitverhalten von $O(\log N)$ für die Suche nach einem Element entspricht); ebenso kann der Baum aber auch im ungünstigsten Fall zu einer Liste entarten; dann ist die maximale Pfadlänge N, und die Suchzeit von der Größenordnung $O(N)$.

Um den ungünstigsten Fall zu vermeiden, muß man den Baum ständig ausgeglichen halten. Das bedeutet, daß bei jeder Änderung an der Struktur (Einfügen, Löschen) der Baum möglicherweise reorganisiert werden muß. Durch diesen zusätzlichen Aufwand beim Aufbau des Baums erkauft man sich bessere Suchzeiten, da die Pfadlängen so kurz wie möglich gehalten werden. (Daran kann man natürlich schon erkennen, daß sich der Aufwand nicht lohnt, wenn die Suchzeiten unkritisch sind, und eine Ersparnis hier den Mehraufwand beim Implementieren und Einfügen nicht rechtfertigt.)

Im folgenden werden die wichtigsten Methoden, ausgeglichene Bäume zu bauen, kurz vorgestellt.

Der zusätzliche Verwaltungsaufwand ist in jedem Fall enorm; die einzelnen Verfahren werden deshalb nicht bis ins Detail beschrieben. Mehr zu dem Thema findet man recht verständlich bei [Sedgewick C].

Prinzipiell kann man einen binären Suchbaum, wie er bisher vorgestellt wurde, ausgeglichen halten, indem nach jeder Strukturänderung nach dem mittleren Element (in der Suchreihenfolge) gesucht wird, dieses Element zur Wurzel gemacht wird mit allen kleineren Elementen links davon und allen größeren Elementen rechts davon, und dann anschließend die beiden Teilbäume ebenso in Ordnung bringt. Dadurch bedingt aber jede kleine Änderung eine komplette Neuorganisation des gesamten Baums mit entsprechend vielen Verschiebungen.

Die tatsächlich verwendeten Verfahren dagegen erlauben mehr Freiheit, was den einzelnen Knoten angeht: anstatt nur ein Element und zwei Nachfolger je Knoten zuzulassen, geht man zu größeren Knoten über, die anstatt einem auch mehr Elemente besitzen können, und statt zwei auch mehr Nachfolger. Dadurch kann man Strukturänderungen in den meisten Fällen auf einen kleinen Bereich begrenzen, und spart dadurch im Schnitt viel Rechenaufwand.

3.6.1.1 Top-Down 2-3-4-Bäume

Wenn man Knoten mit

- 2 Nachfolgern und einem Datenelement,
- 3 Nachfolgern und zwei Datenelementen, sowie
- 4 Nachfolgern und drei Datenelementen

zuläßt, kommt man zu sogenannten 2-3-4-Bäumen.

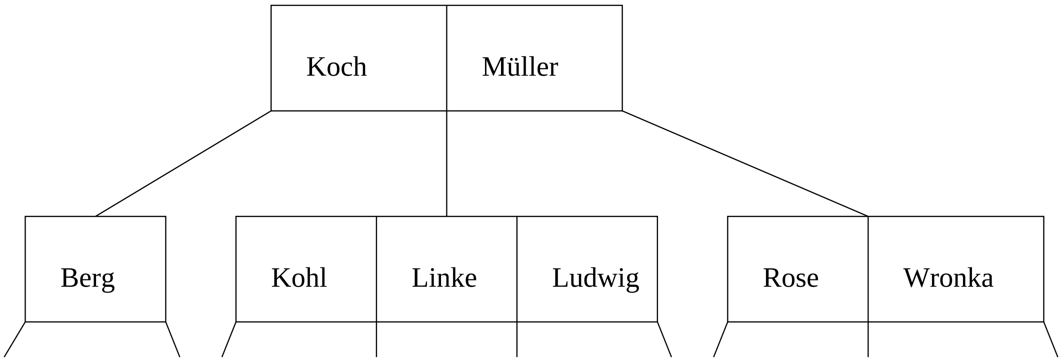


Abbildung 3.1: Beispiel für einen 2-3-4-Baum

In Abbildung 3.1 ist ein kleiner Baum zu sehen, der aus einem 3-Knoten als Wurzel, und je einem 2-, 4- und 3-Knoten als Kinder.

Beim Einfügen eines neuen Elements wird die passende Stelle im Baum gesucht. Wenn hier gerade ein 2- oder 3-Knoten steht, dann wird er um das neue Datenelement erweitert. Trifft man dagegen auf einen 4-Knoten, dann kann der kein weiteres Element aufnehmen, und muß vor dem Einfügen in zwei 2-Knoten zerlegt werden.

Durch das Aufteilen eines 4-Knotens muß aber der nächsthöhere Knoten ein Element mehr aufnehmen. Falls das auch ein 4-Knoten ist, muß der wiederum aufgeteilt, gegebenenfalls dessen Vater ebenfalls, und so weiter. Dieses Aufteilen *bottom up*, also von unten nach oben, kann einerseits im ungünstigsten Fall rechenaufwendig sein, und ist andererseits ungeschickt zu programmieren, weil man entweder bei jedem Element einen Zeiger auf den Vater braucht um sich in Richtung Wurzel hangeln zu können, oder jeweils aufwendig von oben her im Baum suchen muß.

Günstiger ist es daher, *top down* vorzugehen: Wenn man ein Element einfügen will, und von der Wurzel aus nach einem geeigneten Platz sucht, teilt man gleich vorsorglich alle besuchten 4-Knoten in jeweils zwei 2-Knoten auf. Dann ist jedes Aufteilen nur eine lokale Operation, bei der nur der direkte Vater berücksichtigt werden muß.

Die Ausgeglichenheit ergibt sich bei dieser Art von Bäumen automatisch.

Das Implementieren dieser Bäume ist sehr aufwendig, weil einzelne Operationen nur umständlich zu programmieren sind. Deshalb werden Bäume in dieser Form kaum verwendet; aber als Grundlage um ausgefeiltere Strukturen zu beschreiben, sind sie recht nützlich.

3.6.1.2 Red-Black-Trees

So nett die automatische Ausgeglichenheit der 2-3-4-Bäume auch ist: Die Implementation ist sehr aufwendig, weil die Verwaltung einer unterschiedlichen Anzahl Elementen in einem Knoten umständlich ist; ein Binärbaum ist viel einfacher.

Jetzt gibt es aber die Möglichkeit, jeden 4-Knoten eines 2-3-4-Baums (drei Elemente) in einen Knoten mit dem mittleren Element und dem ersten und dritten Element als linken oder rechten Nachfolger aufzulösen.

Ebenso kann man einen 3-Knoten (zwei Elemente) in einen Knoten umsetzen aus dem linken Element und einem rechten Nachfolger aus dem rechten Element; wahlweise kann man auch das rechte Element des ursprünglichen Knotens nehmen, und das linke Element als linken Nachfolger einhängen (Abbildung 3.2).

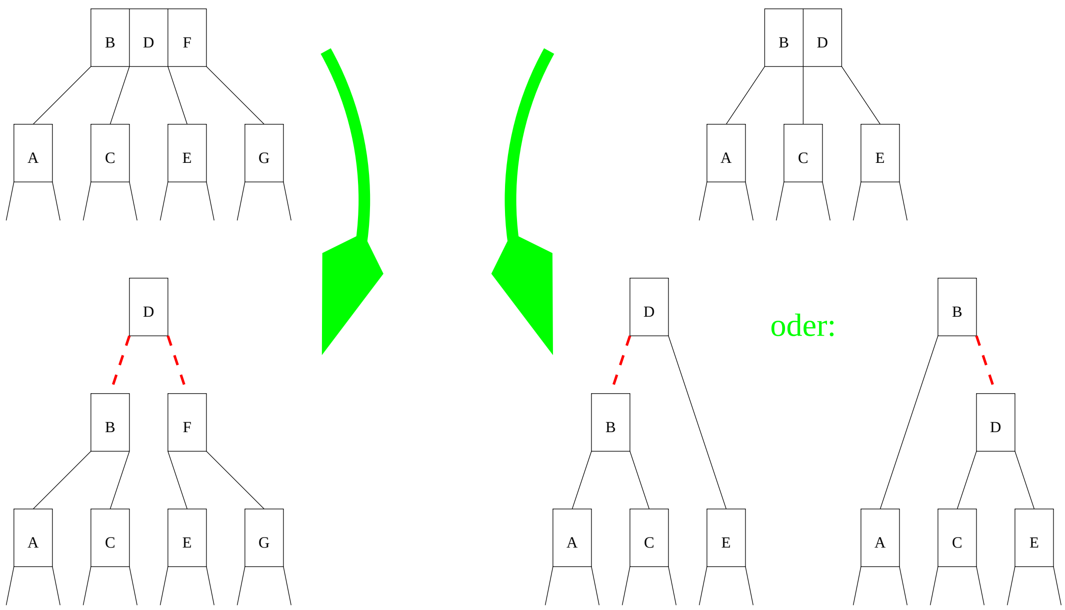


Abbildung 3.2: Umwandlung von 2-3-4-Knoten zu binären Knoten

Der kleine Baum aus Abbildung 3.1 ist in Abbildung 3.3 zu einem Binärbaum umgebaut.

Aufgrund der üblichen Darstellung (beim Auflösen von 3- und 4-Knoten eingeführte Kanten in rot, ursprüngliche Kanten schwarz) heißen solche Bäume *Red-Black-Trees* oder *Rot-Schwarz-Bäume*.

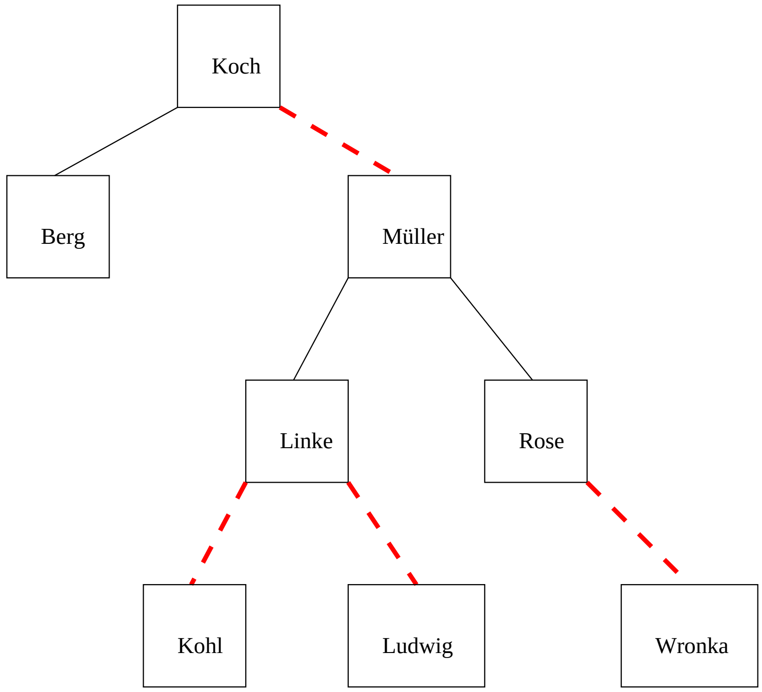


Abbildung 3.3: Der 2-3-4-Baum umgebaut zu einem Binärbaum (Red-Black-Tree)

Die zusätzlichen (roten) Kanten bewirken natürlich, daß der Baum nicht mehr wirklich ausgeglichen ist. Da aber durch jeden umgewandelten Knoten höchstens eine zusätzliche Baumebene eingeführt wird, wächst jeder Pfad um maximal 100%. Die Suchzeit bleibt nach wie in der Ordnung $O(\log N)$. Ausgeglichen ist der Baum nur, wenn man ausschließlich die schwarzen Kanten zählt.

3.6.1.3 AVL-Bäume

Ein AVL-Baum (nach Georgiĭ Maksimovich Adel'son-Vel'skiĭ und Evgeniĭ Mikhaĭlovich Landis) ist ein Binärbaum.

Die Tiefen beider Teilbäume jedes Knotens unterscheiden sich jeweils höchstens um 1. Bei einer Verletzung dieser Bedingung führt zu einer Ausgleichsoperation, die rechenaufwendig ist; dafür werden Rotationen durchgeführt; der Baum muß anhand des Suchwegs zur Einfügestelle nach oben durchlaufen werden (*bottom up*).

Neben dem erhöhten Rechenaufwand zum Ausgleichen ist zusätzlich noch etwas Zusatzinformation pro Knoten nötig (zwei Bit meist, mindestens eines).

3.6.1.4 B-Bäume, Bayerbäume

Bayerbaum

variierbare Anzahl Elemente/Nachfolger pro Knoten; dadurch ist die Knotengröße an effektive Hardwarenutzung anpaßbar (beispielsweise an die Blockgröße eines Dateisystems). Dadurch sehr geeignet für Indexdateien von Datenbanken.

Die einzelnen Knoten sind nicht immer ganz gefüllt, sondern die Belegung jedes Knotens schwankt zwischen einem Kleinst- und einem Größtwert. Dadurch ist in der Regel immer Luft vorhanden, um Strukturänderungen beim Einfügen oder Löschen möglichst lokal zu halten.

Kompakte Einführung in [c't 5/89 Bayerbäume].

3.7 Iteration über Containerelemente

Der Anwender eines Containers wird mit einer gewissen Wahrscheinlichkeit früher oder später eine Möglichkeit brauchen, alle eingefügten Elemente wieder zu erhalten (beispielsweise um alle Elemente in einer Datei zu speichern, oder Summen, Durchschnittswerte und ähnliches über alle Elemente bilden zu können).

Wenn ein Container nur die Mindestoperationen *Einfügen*, *Suchen anhand eines Schlüsselwertes* und *Löschen anhand eines Schlüsselwertes* anbietet, ist es nicht möglich, zu einem späteren Zeitpunkt alle Elemente anfassen zu können, ohne alle Schlüssel zu kennen.

Um mit allen Elementen eines Containers arbeiten zu können (*Iteration über die Elemente*, *Enumeration der Elemente*), ist also eine erweiterte Schnittstelle nötig.

Wenn sich aus der Art des Containers nicht natürlicherweise eine solche Möglichkeit ergibt (wie bei einem Array oder Vektor durch den Zugriff über den Index), hat man dafür im Wesentlichen folgende Möglichkeiten:

1. **Cursor**: Der Container speichert intern einen Verweis auf ein *aktuelles Element*, auch *Cursor* genannt, und bietet in seiner Schnittstelle zum Anwender über Methoden die folgenden Möglichkeiten:

   - den Cursor an den Anfang zu setzen (auf ein Element, das als erstes erscheinen soll, beispielsweise bei einer Liste der Listenanfang)
   - den Cursor von einem Element zum nachfolgenden zu bewegen (in einer Liste also zum Nachfolger wechseln)
   - Zugriff auf die Nutzdaten des Elements, auf das der Cursor gerade verweist
   - eventuell eine Abfragemöglichkeit, ob nach dem aktuellen noch weitere Elemente existieren (man kann auf diese Abfrage auch verzichten, indem beim Weiterbewegen auf das nächste Element durch einen Rückgabewert oder eine Ausnahme das Ende der Iteration signalisiert wird, oder indem der Zugriff auf die Nutzdaten einfach fehlschlägt, wenn man sich bereits über das letzte Element hinausbewegt hat).

   Mit diesen Methoden kann sich der Anwender eine Schleife über alle Elemente konstruieren in der Art (Schreibweise angelehnt an Java):

   ```
   Liste  l;
   l.insert( ... ); // Container füllen
   ```

```
l.insert( ... );
...
Nutzdatentyp aktuell; // Verweis auf jeweils ein Containerelement
l.resetCursor();      // Cursor auf erstes Element richten
while( (aktuell=l.getCursorDaten())!=null ) // Daten holen
{
    // verarbeite das Objekt aktuell...

    l.nextCursor(); // ein Element weitergehen
}
```

2. **Enumerationsobjekt**: Der Container bietet einen eigenen Datentyp zur Enumeration an, von dem sich der Anwender ein Objekt beschafft (beispielsweise mit einer Methode im Container wie etwa `getNewEnumerator()`).

   Dieses Objekt bietet eine Schnittstelle an, wie oben beim Cursor beschrieben.

   Die Verwendung könnte dann so aussehen:

```
Liste  l;
l.insert( ... ); // Container füllen
l.insert( ... );
...
ListenEnumerator aktuell = l.getNewEnumerator(); // Enumerationsobjekt
aktuell.resetCursor();         // Cursor auf erstes Element richten
while( (aktuell.getDaten())!=null ) // Daten holen
{
    // verarbeite das Objekt aktuell.getDaten()...

    aktuell.next(); // ein Element weitergehen
}
```

3. **call back-Funktion**: Der Container bietet eine Methode an (beispielsweise `iterator()`), die über alle Element iteriert, und der Anwender ruft diese Methode auf, anstatt eine eigene Schleife zu bauen. Die Schleife über alle Elemente liegt also nicht im Code des Anwenders, sondern des Containers.

   Weil der Ersteller des Containers aber natürlich nicht wissen kann, was mit den einzelnen Elementen zu geschehen hat, muß der Anwender dies mitteilen. Dazu schreibt er sich eine Funktion beziehungsweise eine Methode, die bei jedem Aufruf ein Nutzdatenelement des Containers erhält, und wie gewünscht verarbeitet. Diese Funktion wird von der `iterator()`- Methode des Containers für jedes Element des Containers aufgerufen, und heißt oft *call back*-Funktion (weil über diese Funktion aus dem Code des Containers beim Anwender sozusagen zurückgerufen wird).

Der Anwendercode für eine Iteration sieht damit etwa so aus:

```
// call back-Funktion:
void tuwasmitmeinenNutzdaten( Nutzdatentyp aktuell )
{
    // verarbeite das Objekt aktuell.getDaten()...
}

...
Liste  l;
l.insert( ... ); // Container füllen
l.insert( ... );
...
l.iteriere( tuwasmitmeinenNutzdaten );
```

Die bei den bisherigen Möglichkeiten zur Iteration (Cursor, Enumerationsobjekt) im Anwendercode vorhandene Schleife über die Elemente ist hierbei in den Container verlagert, und muß gar nicht zwangsweise eine explizite Schleife sein, sondern kann auch durch rekursive Funktionsaufrufe ausgeführt werden.

4. **Funktionsobjekt**: Da nicht alle Programmiersprachen die Übergabe von Funktionen als Parameter ermöglichen (Fortran, Pascal, Ada, C/C++: ja; Java: nein), muß man eventuell auf ein sogenanntes *Funktionsobjekt* (*functional object*, *functionoid*, *functor*) ausweichen. Das ist ein Objekt einer Klasse, die eine Methode anbietet, und sonst in der Regel keine oder nur wenige Daten enthält. Sinn dieses Objekts ist nur, als Parameter übergeben werden zu können, und dabei eine aufzurufende Methode über die Parameterliste zu „schmuggeln“. Aufrufer und Container müssen sich über den Namen der aufzurufenden Methode einig sein; im folgenden Beispiel wird sie einfach `tuwas` genannt.

   Ansonsten ist dieser Weg identisch mit der obigen call back-Funktion; es wird lediglich statt eines Funktion ein Objekt mit angehängter Funktion als Parameter übergeben.

```
// Schnittstelle eines Funktionsobjekts:
interface IFunktionObject
{
    void tuwas( Object o );
}
```

```
// Implementierung des Funktionsobjekts:
class TuwasMitMeinenNutzdaten implements IFunktionObject
{
    void tuwas( Object o )
    {
        // verarbeite das Objekt o...
    }
}

...
Liste  l;
l.insert( ... ); // Container füllen
l.insert( ... );
...
TuwasMitMeinenNutzdaten funktionsobjekt = new TuwasMitMeinenNutzdaten();
l.iteriere( funktionsobjekt );
```

Auch wenn die Vorgehensweise recht ähnlich ist zu einer call back-Funktion, die als Parameter übergeben wird, hat man hier aber zwei Vorteile:

(a) Man kann Funktionsobjekte auch verwenden, wenn Funktionen als Parameter nicht zulässig sind (wie in Java).

(b) Neben der eigentlichen Aufgabe, eine Methode an den Container zu übergeben, kann ein Funktionsobjekt noch mehr Funktionalität bieten.

    Beispielsweise können in der Klasse weitere Attribute definiert werden, die einen internen Status bilden und von Datenelement zu Datenelement manipuliert werden, wie etwa Aufsummieren von Werten, Zählen aller Elemente oder nur solcher mit bestimmten Eigenschaften, und weiteres.

    Hat ein `Student` beispielsweise eine Intelligenz in Form eines Attributs `int iq`, dann kann man die durchschnittliche Intelligenz aller Elemente eines Containers folgendermaßen bilden:

```
// Implementierung des Funktionsobjekts:
class Durchschnittsintelligenz implements IFunktionObject
{
    private int  iq_summe;
    private int  anzahl_aufrufe;

    public Durchschnittsintelligenz()
    {
        iq_summe = 0;
```

```
        anzahl   = 0;
    }

    void tuwas( Object o )
    {
        // Intelligenz summieren, Aufrufe zählen:
        iq_summe += ((Student)o).getIQ();
        anzahl++;
    }

    double getDurchschnittsintelligenz()
    {
        return ((double)iq_summe)/anzahl;
    }
}

...
Liste  l;
l.insert( ... ); // Container füllen
l.insert( ... );
...
Durchschnittsintelligenz durchschnitt = new Durchschnittsintelligenz();
l.iteriere( durchschnitt );
System.out.println( durchschnitt.getDurchschnittsintelligenz() );
```

Verwendet man call back-Funktionen, kann man solche Konstruktionen nicht elegant schreiben, weil man auf irgendeine Form von statischen oder globalen Variablen und wilde Typkonvertierungen zurückgreifen muß.

Hinweis für C++: Da hier (fast) alle Operatoren inklusive dem `operator()` (der wie ein Funktionsaufruf aussieht) überladen werden können, kann man hier auf die Namenskonvention für die aufgerufene Funktion verzichten, und gleich die runden Klammern nehmen. Dann erscheint das Funktionsobjekt tatsächlich wie eine Funktion.

```
// Implementierung des Funktionsobjekts:
class Durchschnittsintelligenz
{
private:
    int  iq_summe;
    private int  anzahl_aufrufe;

public:
    public Durchschnittsintelligenz()
```

```
        : iq_summe( 0 ), anzahl( 0 )
    {
    }

    void operator()( Student o )
    {
        // Intelligenz summieren, Aufrufe zählen:
        iq_summe += ((Student)o).getIQ();
        anzahl++;
    }

    double getDurchschnittsintelligenz()
    {
        return ((double)iq_summe)/anzahl;
    }
}

...
Liste<Student>  l;
l.insert( ... ); // Container füllen
l.insert( ... );
...
Durchschnittsintelligenz durchschnitt;
l.iteriere( durchschnitt );
cout << durchschnitt.getDurchschnittsintelligenz() << endl;
```

Nachdem man jetzt vier verschiedene Möglichkeiten hat, eine Iteration über die Elemente eines Containers zu realisieren, stellt sich die Frage nach den Vor- und Nachteilen.

Die Nutzung eines Cursors direkt im Container ist in der Praxis nicht sinnvoll, weil der Cursor für den gesamten Container gilt, und deshalb vom Anwender sichergestellt werden muß, daß auch nur ein Programmteil zu jedem Zeitpunkt iteriert. Weder kann man zwei ineinander geschachtelte Schleifen verwenden, um alle Elemente eines Containers mit allen anderen zu kombinieren (Kreuzprodukt), noch können zwei oder mehr *threads* eines Prozesses gleichzeitig über alle Elemente iterieren. Der einzige Vorteil dieser Vorgehensweise ist, daß sie mit wenig Quelltext auskommt und daran die prinzipiellen Mechanismen von Containern übersichtlicher dargestellt werden können.

Alle anderen vorgestellten Mechanismen können thread-sicher implementiert werden.

Cursor und Enumerationsobjekte sind nur sinnvoll, wenn innerhalb eines Containers von einem Element zum nächsten navigiert werden kann. Das ist bei einfach verketteten Listen vorwärts möglich, bei doppelt verketteten in beiden Richtungen; bei (offenen) Hashtabellen hängt es von der intern verwendeten Liste ab. Bei Bäumen dagegen bräuchte man für alle gängigen Durchläufe neben den Verweisen von jedem Knoten auf seine Kinder auch Verweise in der anderen Richtung von jedem Kindknoten zu seinem Vaterknoten. Das Durchlaufen einer einfach verketteten Liste in umgekehrter Reihenfolge ist ebenso wenig sinnvoll machbar wie der Durchlauf durch einen Baum, der keine Aufwärtsreferenzen von den Kinder zu den Vätern besitzt.

Bei den Varianten mit call back-Funktion und Funktionsobjekt dagegen muß der Container gar nicht wirklich in einer Schleife durchlaufen werden; oft ist es wesentlich einfacher solche Durchläufe rekursiv zu formulieren (beispielsweise ein in order-Durchlauf durch einen Binärbaum mit der Folge *rekursiver Aufruf mit linkem Kind – Verarbeitung des aktuellen Elements – rekursiver Aufruf mit rechtem Kind*).

Bei Enumerationsobjekten leidet die Effektivität darunter, daß sehr viele Methodenaufrufe nötig sind. Mit call back-Funktionen kann man höhere Geschwindigkeiten erreichen. In C++ sind Funktionsobjekte meist noch schneller, weil alle nötigen Methodenaufrufe vom Compiler durch inline-Aufrufe umgesetzt werden können.

Abgesehen vom höheren Schreibaufwand beim Aufrufer sind in den meisten Fällen dann wohl Funktionsobjekte am sinnvollsten; wenn man den Schreibaufwand beim Anwender mit in Betracht zieht, können call back-Funktionen auch interessant sein.

In der *Standard template library* STL von C++, die ausschließlich auf `template`s aufbaut, ist es an vielen Stellen möglich, wahlweise Zeiger auf call back-Funktionen als auch Funktionsobjekte zu verwenden.

Mit call back-Funktionen oder Funktionsobjekten lassen sich leicht verschiedene Durchläufe (bei Bäumen beispielsweise *in order*, *präorder*, *postorder* und *level order*, oder verschiedene Richtungen bei Listen) durch je eine Methoden realisieren, oder eine gemeinsame Methode mit einem zusätzlichen Parameter, der die Art des Durchlaufs bestimmt.

Kapitel 4

Mustersuche in Feldern

Ein eigenes Kapitel ist die Suche nach einer bestimmten Folge von Elementen in einem Feld. Das kann die Suche nach einer (relativ kurzen) Zeichenfolge in einem (längeren) Text sein, oder ebenso eine bestimmte Folge von Zahlen in einem Feld ganzzahliger Werte oder ähnliches.

Als Beispiel wird aufgrund der Anschaulichkeit die Suche nach einer Zeichenfolge („Muster“) in einer mehr oder weniger langen Folge von Zeichen („Text“) betrachtet; die vorgestellten Verfahren sind aber eindeutig nicht auf Zeichenfolgen beschränkt.

Ein typisches Beispiel ist ein Editor: hier möchte der Benutzer gelegentlich den vorhandenen Text nach einem bestimmten Wort oder einem Teil-Wort durchsuchen.

Der Text soll die Länge N haben; das Muster die Länge M.

Beispiel: in dem N=10 Zeichen langen Text:

`text`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`	`[8]`	`[9]`
	`a`	`x`	`b`	`y`	`c`	`z`	`a`	`x`	`y`	`z`

taucht das M=3 Zeichen lange Muster:

`muster`	`[0]`	`[1]`	`[2]]`
	`a`	`x`	`y`

ab `text[6]` auf.

Gesucht ist also ein Verfahren, um anhand eines Textes und eines Musters die zugehörige Startposition des Musters im Text (hier: 6) zu finden, beziehungsweise die Erkenntnis zu gewinnen, daß das Muster im Text überhaupt nicht vorkommt.

Eine derartige Suche ist häufig nötig, beispielsweise in einem Editor bei der Suche nach einem enthaltenen Wort, oder in einem Virenscanner beim Suchen nach der Signatur bestimmter Viren.

Wenn man sich die mögliche Länge des zu durchsuchenden Textes vorzustellen versucht, wird schnell klar, daß man einen möglichst effizienten Algorithmus benötigt.

4.1 Mustersuche brute force

Mit *brute force* (rohe Gewalt) bezeichnet man Methoden, die ohne viel Nachdenken den direktesten Weg gehen, und dabei gnadenlos Rechenzeit verbraten; beispielsweise um ein Passwort zu knacken einfach alle möglichen Kombinationen durchprobieren.

Bei der Mustersuche in Feldern sieht das folgendermaßen aus: Man geht an die erste Stelle des Textes, und vergleicht ab hier alle Zeichen des Musters paarweise mit der ersten Stelle und den folgenden im Text. Falls bis zum Ende des Musters alle Feldelemente paarweise gleich sind, hat man die gesuchte Stelle gefunden. Ansonsten geht man im Text eine Position weiter, und vergleicht wieder paarweise alle Stellen des Musters mit fortlaufenden Stellen im Text. Dies wird wiederholt, bis man kurz vor dem Ende des Textes ist: hinter der Position N–M kann das Muster nicht mehr beginnen und vollständig enthalten sein.

An jeder Position des Textes kann man abbrechen, wenn eine Stelle nicht zum Muster paßt. Je nach Text und Muster kann das im Schnitt eher oder später der Fall sein.

Das Laufzeitverhalten ist dementsprechend $O((\mathrm{N}-\mathrm{M})\times\mathrm{M})$ für den worst case (wenn immer das ganze Muster verglichen werden muß) bis herunter zu $O(\mathrm{N}-\mathrm{M})$ im best case, wenn bereits immer das erste Zeichen aus dem Muster nicht zutrifft. Wo genau dazwischen der durchschnittliche Fall liegt, hängt stark vom Text und vom Muster ab.

Falls N im Vergleich zu M sehr groß ist, wird die tatsächliche Laufzeit also etwa zwischen $O(\mathrm{N})$ und $O(\mathrm{N}\times\mathrm{M})$ liegen.

Der ungünstigste Fall tritt dann ein, wenn der Text ebenso wie das Muster aus lauter gleichen Zeichen besteht, außer dem jeweils letzten Zeichen. Dann sind alle Kombinationen zu prüfen, um dann an der letzten möglichen Position festzustellen, ob das Muster hier enthalten ist:

text	[0]	[1]	[2]	...	[N-3]	[N-2]	[N-1]
	a	a	a	...	a	a	b

Muster:

muster	[0]	[1]	[2]	...	[M-3]	[M-2]	[M-1]
	a	a	a	...	a	a	b

In C (und sehr ähnlich in Java) könnte die brute force-Suche so aussehen:

```
/* Time-stamp: "20.06.02 17:29 demo.c klaus@lap2.wachtler.de"
 *
 */

#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>


/* sucheteilfeld <brute force>
 * +----------------------------------------------------+
 * |                                                    |
 * | < alle Positionen in text[]: >                     |
 * |                                                    |
 * +----------------------------------------------------+
 * |                                                    |
 * | itext = 0 ... (n-m-1)                              |
 * |  +-------------------------------------------------+
 * |  |                                                 |
 * |  | < alle Positionen im Muster: >                  |
 * |  |                                                 |
 * |  +-------------------------------------------------+
 * |  |                                                 |
 * |  | imuster = 0 ... (m-1)                           |
 * |  |  +----------------------------------------------+
 * |  |  |                                              |
 * |  |  |     text[itext+imuster] != muster[imuster]   |
 * |  |  |                                |             |
 * |  |  +------------- ja ---------------+----- nein ---+
 * |  |  |                                |             |
 * |  |  | <- alle Positionen im Muster   |             |
 * |  |  |                                |             |
 * |  |--+--------------------------------+-------------+
 * |  |                                                 |
 * |  |  Muster gefunden, Lösung ist itext              |
 * |  |                                                 |
 * |  +-------------------------------------------------+
 * |  |                                                 |
 * |  |  <- sucheteilfeld <brute force>                 |
```

```
 * |  |                                              |
 * |--+----------------------------------------------+
 * |                                                 |
 * | Muster nicht gefunden                           |
 * |                                                 |
 * +-------------------------------------------------+
 */

/* liefert einen Index zurück, der die relative Position von muster
 * in text angibt (falls muster in text enthalten ist); sonst -1.
 * Die Länge von text ist m; die Länge von muster ist n.
 */
int sucheteilfeld_brute_force( char text[],
                               int m,
                               char muster[],
                               int n
                               )
{
  int itext, imuster;

  /* Schleife über alle Positionen von text[]: */
  for( itext=0; itext<m-n; itext++ )
  {
    /* zeichenweise text mit muster vergleichen: */
    for( imuster=0; imuster<n; imuster++ )
    {
      if( text[itext+imuster]!=muster[imuster] )
      {
        /* Unterschied gefunden! */
        break;
      }
    }

    if( imuster==n )
    {
      /* innere Schleife bis zum Ende durchlaufen,
       * also Übereinstimmung von text und Muster
       * ab text[itext] gefunden!
       */
      return itext;
    }
  }

  /* nichts gefunden! */
  return -1;
}

int main( int nargs, char **args )
{
  char text[] =
  {
```

```
    'e', 'h', 'n', 'e', ' ',
    'm', 'e', 'h', 'n', 'e', ' ',
    'm', 'u', 'h'
  };

  char  muster[] =
  {
    'm', 'e', 'h', 'n', 'e'
  };

  int position
    = sucheteilfeld_brute_force( text,
                                 sizeof(text)/sizeof(text[0]),
                                 muster,
                                 sizeof(muster)/sizeof(muster[0])
                                 );

  printf( "%d\n", position );
  return 0;
} /* main( int nargs, char **args ) */
```

Achtung! Man ist vielleicht versucht, bei einem festgestellten Unterschied die Suche nicht ein Element nach dem letzten Startpunkt im Text fortzusetzen, sondern aus Geschwindigkeitsgründen hinter der zuletzt verglichenen Stelle („soweit habe ich schon verglichen, also kann ich dahinter weitermachen“).

Dies kann aber dazu führen, daß man ein Vorkommen des Musters übersieht!

Beispiel: Der Text sei:

`text`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`	`[8]`	`[9]`
	`a`	`a`	`b`	`a`	`a`	`b`	`a`	`b`	`y`	`z`

und das Muster sei:

`muster`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`
	`a`	`a`	`b`	`a`	`b`

Dann werden erst die Elemente `text[0]` bis `text[3]` erfolgreich auf Gleichheit mit dem jeweils zugehörigen Element in `muster` getestet; bei `text[4]` wird ein Unterschied zu `muster[4]` festgestellt. Würde man jetzt die Suche fortsetzen, indem man `text[4]` und die folgenden Elemente mit `muster[0]` und seinen Nachfolgern vergleicht, übersieht man das Vorkommen des Musters ab `text[3]`. Solange man keine weiteren Maßnahmen trifft, muß man also sicherheitshalber nach der erfolglosen Suche ab `text[`*i*`]` als nächstes das Muster ab `text[`*i+1*`]` suchen.

4.2 Verfahren von Knuth, Morris und Pratt

Bei der brute force-Methode muß man beim Auftreten eines Unterschieds wie eben beschrieben die Suche ein Zeichen hinter dem letzten Startpunkt im Text fortsetzen, weil man sonst möglicherweise ein Vorkommen des Musters übersieht.

Das ist bedauerlich, weil dadurch viele Zeichen aus `text` mehrfach betrachtet werden müssen.

Je nach Aufbau des Musters kann man allerdings auch weiter springen, bevor man mit erneutem Vergleichen beginnt, und spart sich dadurch einen Teil der Vergleiche.

Beispiel: falls das Muster folgenden Inhalt hat:

`muster`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`
	`a`	`b`	`b`	`b`	`c`

, dann kann bei einem bei `c` festgestellten Unterschied an der Musterposition `[4]` die Suche gleich an der aktuellen Stelle im Text fortgeführt werden; denn ein Vorkommen des Musters muß mit einem `a` beginnen; weil aber im Muster nach dem Anfang kein `a` mehr enthalten ist und bis zu `muster[3]` alle Zeichen übereinstimmten, kann im Text kein `a` stehen, soweit er bereits mit `muster[1]` bis `muster[3]` verglichen wurde.

Ist dagegen das Muster:

`muster`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`
	`a`	`b`	`c`	`a`	`b`	`b`

, und wird ein Unterschied an der Stelle des `muster[5]` festgestellt, dann kann die Suche an der Stelle im Text neu aufgesetzt werden, die mit dem zweiten `a` verglichen wurde (denn dort könnte das mit `a` beginnende Muster theoretisch neu starten).

Anstatt also die Suche jeweils beim Zeichen hinter dem letzten Start neu aufzusetzen, dürfte man unter Umständen auch gleich etwas weiter springen; wie weit hängt erstens vom Muster ab, und zweitens von der erreichten Position im Muster bis zum letzten Abbruch.

Diese Erkenntnis führt zu dem *Verfahren von Knuth, Morris und Pratt*.

Hier wird vor der eigentlichen Suche ein zusätzliches Feld geschaffen, anhand dessen zu jeder möglichen Abbruchposition eine erlaubte Sprungweite hinterlegt wird. Dieses Feld wird für jede Suche aufgrund des Musters einmal bestimmt; bei längeren Texten geht diese Vorabberechnung also nicht in die Laufzeit mit ein.

An jeder Position ist angegeben, wie viele Elemente vor dem aktuellen mit dem Stringanfang übereinstimmen (abgesehen davon, daß die unterste Sprungweite immer mit -1 eingetragen wird).

```
/* Time-stamp: "20.06.02 17:39 demo.c klaus@lap2.wachtler.de"
 *
 */

#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>

/* liefert einen Index zurück, der die relative Position
 * von muster in text angibt (falls Muster in text enthalten ist);
 * sonst -1.
 * Die Länge von text ist n; die Länge von muster ist m.
 */

void init_sprweite( char muster[], int sprweite[], int m )
{
  int  i, j;
  sprweite[0] = -1;
  for( (i=0), (j=-1); i<m; i++, j++,  sprweite[i]=j)
  {
    while( (j>=0) && (muster[i]!=muster[j]) )
    {
      j = sprweite[j];
    }
  }
}


int sucheteilfeld_kmp( char text[], int n,
                       char muster[], int sprweite[], int m
                       )
{
  int index_text, index_muster;

  init_sprweite( muster, sprweite, m );

  for( (index_text=0), (index_muster=0);
       index_muster<m && index_text<n;
       index_text++, index_muster++
       )
  {
    while( index_muster>=0 && text[index_text]!=muster[index_muster] )
    {
      index_muster = sprweite[index_muster];
    }
```

```
  }

  if( index_muster==m )
  {
    return index_text - m;
  }
  else
  {
    return -1;
  }
}

int main( int nargs, char **args )
{
//  char text[] =
//  {
//    'e', 'h', 'n', 'e', ' ',
//    'm', 'e', 'h', 'm', 'e', 'm', 'm', 'e', 'h', 'm', 'e', 'm', 'a', ' ',
//    'm', 'u', 'h'
//  };
//
//  char  muster[] =
//  {
//    'm', 'e', 'h', 'm', 'e', 'm', 'm', 'e', 'h', 'm', 'e', 'm', 'a', ' ',
///* zugehörige Sprungweiten: */
///*   -1,   0,   0,   0,   1,   2,   1,   1,   2,   3,   4,   5,   6,   0,  0 */
//
//  };

//  char text[] =
//  {
//    'z', 'a', 'b', 'z', 'a', 'z', 'a',
//  };
//
//  char  muster[] =
//  {
//    'z', 'a', 'z', 'a',
///* zugehörige Sprungweiten: */
///*  -1,   0,   0,   1,   2  */
//
//  };

  char text[] =
  {
    'z', 'a', 'z', 'z', 'a', 'z', 'a',
  };

  char  muster[] =
  {
    'z', 'a', 'z', 'a',
/* zugehörige Sprungweiten: */
```

```
/*  -1,   0,   0,   1,   2  */

  };

  const int laenge_text = sizeof(text)/sizeof(text[0]);
  const int laenge_muster = sizeof(muster)/sizeof(muster[0]);

  int sprweite[laenge_muster+1];

  int position
    = sucheteilfeld_kmp( text, laenge_text,
                         muster, sprweite, laenge_muster );

  printf( "%d\n", position );

  return 0;

} /* main( int nargs, char **args ) */
```

4.3 Verfahren von Boyer und Moore

1977 veröffentlicht

nicht klausurrelevant

Siehe [Sedgewick C].

4.4 Weiterführende Verweise

Neben den hier beschriebenen Verfahren gibt es noch weitere; teilweise für exakt dieselbe Aufgabenstellung ([Optimal schnell], [Blitzfindig], [Sedgewick C]), teilweise aber auch für Abwandlungen (reguläre Ausdrücke, *pattern matching*:[Mustergültig], [Sedgewick C]).

Kapitel 5

Algorithmen zur Optimierung

Unter **Optimierung** kann man vielerlei verstehen: Optimierung der eigenen Einkünfte durch Gehaltserhöhung oder Anlagebetrug, Optimierung eines Programms zwecks kürzerer Laufzeit oder geringeren Speicherbedarfs, Optimierung eines industriellen Prozesses zur Kosteneinsparung und/oder Qualitätssteigerung, und vieles mehr.

Vor jeder Optimierung steht die Frage, *was* optimiert werden soll. Dazu wird eine **Zielfunktion** (oder auch *Qualitätsfunktion* oder *Fitneßfunktion*) definiert. Diese Zielfunktion ist eine Funktion mindestens einer (und meist vieler) Veränderlicher. Aus der Definitionsmenge, also der Menge aller möglichen Werte der Veränderlichen, soll die Kombination der Veränderlichen herausgefunden werden, mit der sich ein Maximalwert der Zielfunktion ergibt. Diese Kombination der Veränderlichen wird als Optimum bezeichnet; das Auffinden derselben ist Ziel der Optimierung. Eine Optimierung ist demnach gleichbedeutend mit dem *Maximieren* der Zielfunktion.

Die Definition der Zielfunktion ist natürlich vom jeweiligen Problem abhängig.

(Prinzipiell könnte man eine Optimierung auch auffassen als Versuch, eine schädliche Funktion irgendeiner Art (Kosten, Schmerzen, Geräuschemission) zu *minimieren.* Dies läßt sich aber auf das oben beschriebene Maximieren einer Zielfunktion zurückführen, indem man die Zielfunktion als Negierung der vorhandenen schädlichen Funktion definiert, und dann wieder nach dem Maximum sucht.)

Je nach Aufgabenstellung kann man Optimierungsprobleme unter verschiedenen Gesichtspunkten unterscheiden:

- Bei einem diskreten Problem besitzen die variablen Größen keine stetige Definitionsmenge, sondern sind als Menge distinkter Elemente definiert. Andererseits kann das Problem auch auf stetige Funktionen hinauslaufen, die einer mathematischen Analysis zugänglich sind. Es gibt auch Fälle, in denen einige Variablen einer distinkten Menge angehören, andere stetig definiert sind.

  Ein Beispiel für ein diskretes Problem ist die Erstellung eines Stundenplans: es gibt nur distinkte Varianten, die nötigen Unterrichtseinheiten zu kombinieren. Einige Bedingungen sind unbedingt einzuhalten (ein Lehrer kann nicht gleichzeitig mehrere Fächer unterrichten). Für ein optimales Ergebnis sind weitere Nebenbedingungen wünschenswert (möglichst keine leeren Blöcke für die Schüler, die Fächer eines Lehrers an möglichst wenigen Tagen, um mehr freie Tage für die überlasteten Lehrkräfte zu ermöglichen, etc.).

  Typische kontinuierliche Probleme sind die Auslegung von Tragflächenprofilen, Strömungskanälen wie sie in Turbinen oder Verdichtern vorkommen, wirtschaftliche Optimierungen, und viele andere.

- Möglicherweise existieren einzuhaltende Nebenbedingungen (lineare Optimierung, typisches Beispiel ist die Kapazitätsplanung einer Raffinerie mit dem Simplexverfahren; Ziel ist eine maximale Gewinnerzielung).

- Die Anzahl der Variablen entspricht der Anzahl der Dimensionen des Definitionsraumes; nicht alle Verfahren sind für höhere Dimensionen geeignet.

5.1 Geschlossene Lösung

Wenn eine stetig differenzierbare Form der Zielfunktion vorliegt, dann kann man das Optimum finden, indem man einfach das Maximum der Zielfunktion gemäß der klassischen Analysis bestimmt: alle Maxima, Minima und Sattelpunkte der Zielfunktion drücken sich in einer Nullstelle ihrer ersten Ableitung aus. Man bestimmt also die Nullstellen der Ableitung, und ermittelt für die so gefundenen Stellen durch eine Funktionsdiskussion (oder durch Einsetzen in die Zielfunktion) das Maximum.

Eigenschaften dieser Vorgehensweise:

- einfache Differenzierbarkeit der Zielfunktion nötig

- oft eleganter Weg, der gelegentlich auch vereinfachende Zusammenhänge erkennen läßt

- Verständnis der Lösung nötig

5.2 Raten der Lösung

Wenn die Eingangsparameter nicht zu viele mögliche Varianten haben (also nur eine endliche Menge von diskreten Möglichkeiten existiert), dann kann man gelegentlich auch einfach für alle Möglichkeiten jeweils die Zielfunktion berechnen (lassen), und findet so die Lösung.

Auch bei kontinuierlichen Problemen kann man ähnlich vorgehen: Über die möglichen Lösungen wird ein Gitter gelegt mit einer Dimension entsprechend der Anzahl Veränderlicher der Zielfunktion, und dann in einem in Frage kommenden Bereich der Definitionsmenge die Zielfunktion an allen Gitterpunkten berechnet. Eine Funktion einer Veränderlichen wird also in Streifen nicht zu großer Breite zerlegt, eine Funktion zweier Veränderlicher in Rechtecke, und so weiter. Von allen berechneten Werten der Zielfunktion nimmt man den größten, und kann nun (falls die Unterteilung fein genug war) vermuten, daß das tatsächliche Maximum im umgebenden Bereich liegt. Letzteren unterteilt man wiederum mit einem feineren Gitter, und berechnet sich an den Gitterpunkten die Zielfunktion, und so fort – bis das gesuchte Maximum mit einer geforderten Genauigkeit angenähert ist. Siehe dazu auch die Übungsaufgabe **A.1.10 Maximum einer Funktion** auf Seite 99 und die zugehörige Lösung **A.2.10 Maximum einer Funktion** auf Seite 210. Da man durch die schrittweise Verfeinerung nur lokale Maxima finden kann, muß die Anfangsunterteilung fein genug sein, um auch jedes Maximum als einziges in einem Teilbereich zu finden.

Solche Verfahren stossen offensichtlich an ihre Grenzen, sobald die Rechenzeit unerträglich wird, weil man zuwenig Wissen über das Problem hat (oder in die Berechnung einfließen lassen kann).

In Analogie zum Glücksspiel werden Verfahren, bei denen mehr oder weniger geschickt geraten wird, als *Monte-Carlo-Methoden* bezeichnet.

Für die hier behandelte Optimierung sind diese Methoden in der Regel nicht gut geeignet. Gelegentlich kann man sie sinnvoll zur Simulation von Vorgängen verwenden, in denen eine stochastische Komponente enthalten ist. Sie werden hier erwähnt, weil sie eine gedankliche Vorstufe zu den Evolutionsstrategien (**5.3 Genetische Algorithmen, Evolutionsstrategie** auf Seite 50) darstellen.

5.3 Genetische Algorithmen, Evolutionsstrategie

Wie man der Bezeichnung bereits ansehen kann, ist der folgende Abschnitt stark an die natürliche Evolution (wie sie von Charles Darwin begründet wurde) und teilweise an die Genetik angelehnt.

Dazu werden folgende Begriffe aus der Biologie entlehnt (entgegenstehende religiöse Ansichten ignoriere ich hier im Hinblick auf die Zielsetzung dieses Skripts und ohne größere Böswilligkeit):

- Ein **Lebensraum** besteht zu jedem Zeitpunkt aus einer bestimmten Anzahl von **Lebewesen**; die Anzahl kann variabel sein, ist aber in der Regel begrenzt (auch wenn dies nicht immer eingesehen wird).
- Gelegentlich **entstehen** neue Lebewesen.
- Ebenso **sterben** vorhandene und nehmen am weiteren Geschehen nicht mehr teil (es sei denn als Proteinlieferant für begrenzte Zeit).
- Jedes Lebewesen hat einen gewissen Satz **Erbinformationen** (**Gene**), die quasi ein Bauplan dieses Wesens sind. Einzelne Gene oder Gruppen davon (Allele) bewirken die Haarfarbe, die Anzahl der Beine, Nasen, Gesichtszüge, Bau der Organe, chemische Vorgänge und so weiter.
- Jedes der Lebewesen ist **individuell** und wird sich in der Regel von anderen mehr oder weniger unterscheiden. Als Ausprägung seiner Geninformationen nennt man ein solches Individuum auch *Phänotyp*.
- die **Fähigkeiten** eines Lebewesens ergeben sich teilweise aus seinen Erbinformationen (zwei Augen sehen mehr als 10 Ohren, mit vier Beinen kann man besser laufen als mit zweien, aber mit zwei Händen kann man besser den Kühlschrank öffnen als mit vier Hufen), teilweise aber auch aus den **Einwirkungen seiner Umgebung** auf das Lebewesen (Erziehung, Ernährung, Einnahme von Drogen, schwere Kindheit, Training etc.)
- jedes Lebewesen erhält seine Erbinformationen von zwei **Eltern**teilen (die ungeschlechtliche Vermehrung durch Zellteilung, die teilweise bei Bakterien oder vielen Pflanzen (Vermehrung durch Ableger) beliebt ist, soll hier nicht weiter interessieren, weil sie hinsichtlich der Optimierung nicht so erfolgreich und auch aus Sicht der Informatik langweilig ist). Die Gene beider Elternteile werden mehr oder weniger willkürlich so kombiniert, daß sich einer oder mehrere neue Sätze von Erbinformationen ergeben, die jeweils

zum Teile Gene von Vater und zum anderen Teil Gene der Mutter enthalten (**Rekombination**, **crossing over**). Bei der üblicherweise praktizierten Vermehrung wird von diesen beiden rekombinierten Genen nur eines verwendet, das andere verworfen und als Eiweiß entsorgt.

- Gelegentlich verändern sich Gene auch **spontan** (etwa durch Radioaktivität, Vorgänge in der Zelle durch Unzuverlässigkeit der beteiligten Reaktionen, Alkoholeinfluß, Medikamente, Alter…), sodaß weitere Rekombinationen mit einem veränderten Satz von Erbinformationen ausgeführt werden (**Mutation**).

- neben dem erfreulichen Thema Vermehrung **sterben** leider auch Lebewesen, und zwar umso leichter je schlechter sie an ihre Umgebung **angepaßt** sind. Früher oder später sterben sie alle, aber: Es gibt Lebewesen, die eine gute Anpassung an ihre Umgebung besitzen; diese überleben länger (weil sie rechtzeitig das Richtige essen (und zwar mindestens so schnell wie ihre Geschwister), sich schnell genug einen guten Platz suchen wenn die Wölfe kommen, und nicht Samstag abend besoffen aus der Disco heimfahren), und haben mehr Chancen sich zu vermehren (eine Rekombination der Gene ist nur möglich, wenn die Eltern noch leben und dazu willens und in der Lage sind; beispielweise sind in der Natur meist zwei Individuen verschiedener Geschlechter nötig, die sich zumindest kurzzeitig einig sein müssen). Die weniger tauglichen Individuen sterben leichter als die besseren (weil sie zu schnell Motorrad fahren, bekifft vor die Straßenbahn laufen, oder sich auf einen Kampf mit Keule und Tritt in den Unterleib einlassen), oder sind nicht in der Lage sich zu vermehren, bevor sie irgendwann eines mehr oder weniger natürlichen Todes sterben (durch biologische Defekte nicht attraktiv genug oder vermehrungsfähig, homosexuell, gucken lieber Sportschau, oder sind einfach zu dusselig). Je nach den Fähigkeiten eines Lebewesens gibt es also höhere oder niedrigere Chancen, die eigenen Gene in neue Lebewesen einzubringen, die wiederum gut oder schlecht mit ihrer Umwelt zurechtkommen (**natürliche Selektion**, „survival of the fittest“).

- Neben dem Sterben durch Untauglichkeit (oder eremitischen Lebensstil, was evolutionsmäßig auf dasselbe hinausläuft) kommt es auch vor, daß durch irgendwelche zufälligen Vorgänge ein hoffnungsvolles Leben ausgelöscht wird, obwohl es eigentlich recht tauglich war (Meteoriteneinschlag, ein anderer Idiot fährt ein taugliches Wesen versehentlich oder absichtlich um), oder andererseits ein gar nicht so taugliches Wesen auch die Gelegenheit zur Vermehrung bekommt (ein blindes Huhn…). Je nach Weltanschauung nennt man so etwas **Schicksal**, Vorsehung, Gottes Wille etc.; oder einfach „dumm gelaufen“.

- Die eigentliche **Evolution** besteht jetzt darin, daß sich der Lebensraum mit der Bevölkerung so entwickelt, daß die Lebewesen von Generation zu Generation „besser“ werden, indem sich Gene, die in Anbetracht der Umwelt zu einer hohen Tauglichkeit führen, mit der Zeit die unbrauchbaren Gene verdrängen.

Zudem verbessern sich neben den Genen auch noch andere Umstände, die das Überleben und die Vermehrung stark beeinflussen. Beispielweise ist das Wissen, wann Getreide ausgesät werden muß, nicht in den Genen begründet, sondern wird von Individuum zu Individuum irgendwie weitergegeben; es bilden sich Sitten und Gebräuche heraus, die den Bestand günstig beeinflussen (in Grönland die Sitte, Iglus zu bauen; in warmen Gegenden der rituelle Verzicht auf bakteriell verseuchtes Schweinefleisch, dafür der wurmhemmende Genuß von Chillie; in den meisten Gesellschaften der weitgehende Verzicht auf Mord; und so weiter). Solche Effekte sind eng mit dem obigen Punkt „Fähigkeiten“ durch „Einwirkungen seiner Umgebung“ verknüpft, in der Soziologie sicher ein unerschöpfliches Thema, tun aber hier in der Regel nichts zur Sache (auch wenn Effekte aus diesem Gebiet durchaus auch algorithmisch interessant sein können).

Auch wenn beispielsweise von den Eltern an die Nachkommen überlieferte Sitten und Gebräuche, Wissen und so weiter nicht direkt in den Genen begründet liegt, so ist doch zumindest die Fähigkeit, Wissen zu sammeln und weiterzugeben, von den Genen abhängig.

- eine ganz wichtige Eigenheit der Evolution ist die **Konkurrenz**: In einem frühen Stadium können sich noch Lebewesen behaupten, die vom Optimum weit entfernt sind („unter Blinden ist der Einäugige König“). Je weiter sich die Evolution entwickelt, desto besser muß ein Lebewesen angepaßt sein, um nicht von besseren verdrängt zu werden. Dieser steigende Konkurrenzdruck ist also natürlich, und beschleunigt die Evolution deutlich.

Evolution ist demnach nicht eine Optimierung, die irgendwie rechnet, und dann schlagartig ein Ergebnis liefert, sondern ist vielmehr ein kontinuierlicher Prozeß, der eine anfangs schlechte Lösung Schritt für Schritt verbessert.

Was hat das alles mit Informatik zu tun (außer daß sogar Informatiker gelegentlich versuchen, sich fortzupflanzen)?

Wenn man sich den oben geschilderten Vorgang von Werden und Vergehen etwas aus der Entfernung betrachtet, dann ist die natürliche Evolution ein ständig fortgeführtes Experiment, in dem die Erbinformation kontinuierlich optimiert wird. Denn durch Mutation und Rekombination entstehen laufend neue Erbinformationen, welche durch die Selektion einer Prüfung unterworfen werden. Nur die Erbinformationen, die sich für die aktuelle Umgebung als günstig herausstellen, werden wiederum für Mutation und Rekombination verwendet und führen zu weiter variierten Sätzen von Information. Die schlechten Lösungen vergehen recht schnell und meist ohne viel Aufhebens, während gute Lösungen gepflegt und im weiteren sogar noch verbessert werden.

Selbst nach einer recht langen Rechenzeit in der Größenordnung einiger Millionen Jahre kann allerdings noch lange nicht von einem Optimum gesprochen werden angesichts weit verbreiteter Blödheit, Krieg, abstrusen Sekten einschließlich der katholischen Kirche, AIDS und der Bildzeitung; und das bei einem riesigen Platzverbrauch für das Experiment.

Bedenkt man aber den Ausgangszustand in Form von primitiven Einzellern, ist ein enormer Fortschritt zu bekunden, sodaß in den Sechziger Jahren ein findiger Mensch namens John M. Holland auf die Idee kam, ein solches Experiment im Rechner zu simulieren: Die Gene lassen sich als Binärinformation deuten, die Menge der Lebewesen besteht aus einem Feld von Elementen, die im wesentlichen die Gene beinhalten, und die Fähigkeit zu überleben und sich zu vermehren, wird mit einer Tauglichkeitsfunktion (*fitness function*) ausgedrückt. Die Rekombination und Mutation vorhandener Gene läßt sich leicht bewerkstelligen. Selektion ist das Löschen unbrauchbarer Elemente (also solche, für die sich ein schlechter Wert der Tauglichkeitsfunktion ergibt). Eher schwierig ist die Simulation der Umgebung auf die Tauglichkeit (Erziehung etc.). Außerdem ist zur Simulation realer Lebewesen das Wissen um die Bedeutung der Gene vonnöten, was in absehbarer Zukunft nicht erreicht sein wird.

Aber auch wenn man nicht die Welt simulieren kann (und das eigentlich auch nicht unbedingt muß), kann man das Verfahren zur Optimierung anderer Vorgänge verwenden. Die Anwendung eines solchen Verfahrens auf reale Problemstellungen heißt **Evolutionsstrategie**.

Um das Ganze praktisch nutzbar zu machen, weicht man also sinnvollerweise von der tatsächlichen Evolution etwas ab:

- Die Tauglichkeitsfunktion wird vorgegeben, und entspricht der Zielfunktion des zu optimierenden Problems.

- Der Satz an Erbinformationen (die Gene) drückt eine Kombination der variablen Eingangsgrößen des gegebenen Problems aus.

- Die Tauglichkeitsfunktion verwendet nur fest vorgegebene Größen sowie die Erbinformationen eines Lebewesens, nicht Umstände wie Aufenthaltsort im Lebensraum, Sozialverhalten, angesammeltes Wissen in Gemeinschaften oder ähnliches. (Will man solche Informationen auswerten, baut man sie in die Gene mit ein und behandelt sie fortan ebenso.)

- Dementsprechend wirkt sich die Umwelt nicht mehr auf die Lebewesen aus; neben Rekombination und Mutation werden sie nicht weiter beeinflußt (abgesehen von ihrem willkürlichen Tod). Insbesondere können sie ihre Chancen auf Fortpflanzung nicht durch Lernen, Training im Sportstudio oder Schminken verbessern (also keine Rückkopplung der Umwelt auf die Fähigkeiten innerhalb eines Lebens).

- Die Anzahl der Lebewesen ist (meistens) konstant; der Lebensraum ist damit einfach ein Feld von Genen.

- Der Einfachheit halber werden untaugliche Lebewesen üblicherweise (rückstandsfrei) entsorgt, indem sie durch Produkte aus der Rekombination überschrieben werden.

- Die einzelnen Lebewesen haben in der Regel keine Beziehungen untereinander. Sie existieren unabhängig voneinander und werden nur verglichen und gelegentlich entsorgt. Die einzige Stelle, an der sie in Beziehung treten, ist bei der Rekombination: ein erbarmungsloser Algorithmus wählt zwei Lebewesen aus, um ihre Gene neu zu kombinieren. Die ausgewählten Lebewesen verbleiben dabei im Lebensraum; ihre Nachkommen werden zusätzlich abgelegt (möglicherweise als Ersatz für andere ausgeschiedene Wesen).

- Die Umgebung ist meist konstant; es ändern sich also nicht laufend die Bedingungen für die Lebewesen. Dies würde sich in einer laufenden Anpassung der Zielfunktion ausdrücken. Da die Zielfunktion aber dem zu optimierenden Problem entspricht, soll sie sich ja eben nicht laufend ändern. (Es gibt gelegentlich den Fall, daß man eine Optimierung in mehreren Schritten erreichen will; dann kann eine Anpassung der Zielfunktion und eine Weiterverwendung der existierenden Lebewesen durchaus sinnvoll sein. Beispielsweise könnte man bei einer aufwendig zu berechnenden Zielfunktion eine erste Teilevolution veranstalten, die mit einer vereinfachten Zielfunktion relativ schnell eine Ausgangslage für einen zweiten, genaueren Teil liefert.)

  Die Fähigkeit der natürlichen Evolution, auf eine Veränderung der Lebensbedingungen durch Anpassung zu reagieren, wird damit üblicherweise nicht genutzt.

- Es wird meist keine Unterscheidung der Geschlechter gemacht; dies führt schon im realen Leben zu viel Ärger und hat offensichtlich kaum Vorteile (zumindest nicht in der rechnerischen Umsetzung; im realen Leben, wo sich auch Gebräuche einbürgern wie Arbeitsteilung im näheren Umfeld beispielsweise, und die Zöglinge einer gewissen Pflege bedürfen, scheint es in den letzten Jahrmillionen Vorteile geboten zu haben, sonst hätte es sich wohl nicht so entwickelt).

- Die Evolution läuft nicht in Echtzeit ab, sondern deutlich schneller (wenn man nicht gerade in Java oder Basic programmiert).

- Die Evolution wird nicht endlos fortgeführt, sondern irgendwann beendet.

  Das **Ergebnis** der Optimierung besteht dann aus den Genen des besten Lebewesens des letzten Evolutionsschrittes. In diesen Genen drückt sich die beste erreichte Annäherung an das unbekannte Optimum aus.

  Wenn man entweder alle möglichen Gene probiert hat, oder zufällig die richtige Kombination getroffen hat, wird man auch die wirklich optimale Lösung haben (wenn man sie nicht versehentlich durch Mutation wieder zerstört). Normalerweise wird man aber bei komplexen Problemen nicht so lange warten können, sondern nach einer bestimmten Rechenzeit abbrechen, oder wenn die Verbesserung des/der besten Elemente von Evolutionsschritt zu Evolutionsschritt sich nicht mehr nennenswert verbessert. Dadurch erhält man typischerweise nicht *die bestmögliche*, sondern nur eine *gute* Lösung. Wenn man die beste Lösung braucht, und genug Zeit hat, mit einem genetischer Algorithmus darauf zu warten, kann man auch gleich alle Varianten der Eingabeparameter durchprobieren, und kommt damit sicherer zum Ziel.

Der grundsätzliche Ablauf hat damit das Aussehen von Bild 5.1.

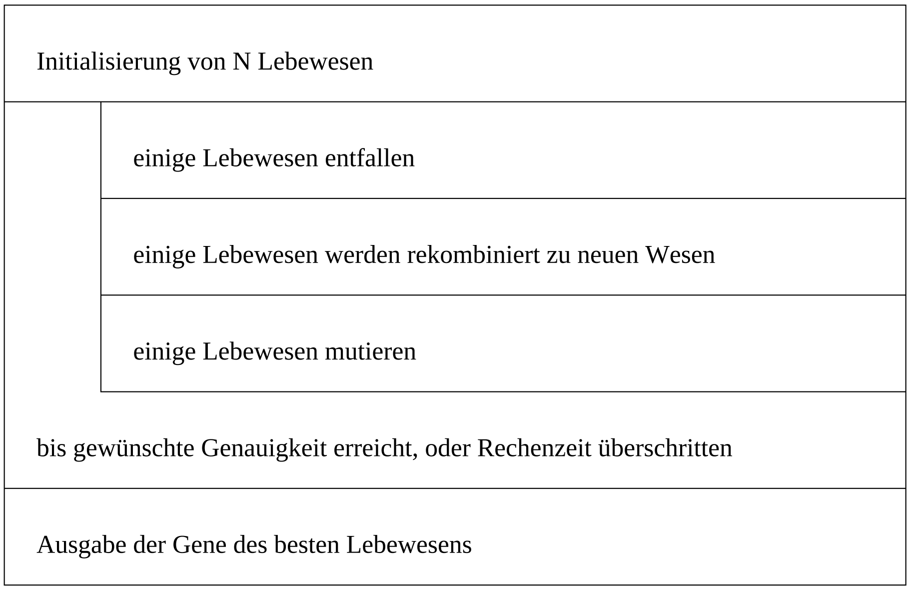


Abbildung 5.1: Prinzipieller Ablauf eines genetischen Algorithmus

Zu den einzelnen Schritten in Abbildung 5.1:

Initialisierung von N Lebewesen: Wenn man kein Wissen über eine sinnvolle Vorbelegung hat, initialisiert man die Gene aller Lebewesen einfach mit zufälligen Zahlen. Es muß nur sichergestellt sein, daß die folgenden Schritte damit auch klarkommen; beispielsweise können zufällig initialisierte Gleitkommazahlen gemäß dem üblicherweise verwendeten Standard IEEE 754 einen ungültigen Wert haben (*positive infinity, negative infinity, verschiedene NAN = not a number*), der bei der weiteren Rechnung je nach System zu Laufzeitfehlern oder unsinnigen Ergebnissen führen kann, wenn nicht gezielt mit geeigneten Prüfungen dieser Fall abgefangen wird (wie im Blechdosenbeispiel **5.4.2 Lösung mithilfe eines genetischen Algorithmus** auf Seite 65 mit der Funktion `isnan()` in der Zielfunktion). Ebenso unsinnig sind natürlich zufällige Werte für Zeigervariablen in C- oder C++-Programmen.

Hat man dagegen bereits ein Vorwissen, wie eine vernünftige Lösung aussehen könnte oder welche Kombinationen sinnvoll sind und welche nicht, dann kann man die ersten Lebewesen natürlich gleich entsprechend initialisieren und dadurch die ersten Evolutionsschritte etwas beschleunigen.

Auf jeden Fall sollte man aber vermeiden, Lebewesen unnötigerweise identisch zu initialisieren; dadurch muß die genetische Vielfalt erst rechenaufwendig durch Mutation erzeugt werden; und: ohne Vielfalt kein Leben! Welchen Sinn hat die Rekombination, wenn alle möglichen Partner gleich sind?

Die Wahl von N ist etwas Glückssache: zuwenige Wesen bedeuten eine genetische Armut (wie in dem einen oder anderen Adelsgeschlecht), zuviele erhöhen den Rechenaufwand pro Evolutionsschritt. In jedem Fall ist es sinnvoll, N wesentlich kleiner zu machen als die Anzahl der Kombinationsmöglichkeiten der Eingangsparameter der Optimierung (also wesentlich kleiner als die Mächtigkeit der Definitionsmenge der Zielfunktion). Dies ergibt sich aber meist von selbst, weil die Vorgehensweise sinnvollerweise für relativ komplexe Probleme mit vielen Kombinationsmöglichkeiten angewendet wird, sodaß die Anzahl der im Arbeitsspeicher darstellbaren Wesen ohnehin klein ist im Vergleich zu allen möglichen Wesen.

einige Lebewesen entfallen entspricht der Selektion und ist meist einfach gelöst, indem die Gesamtzahl der Lebewesen konstant ist und beim Erzeugen neuer Wesen durch Rekombination alte überschrieben werden. Die Auswahl, welche Wesen entfallen sollen, kann unterschiedlich erfolgen. Zumindest sollten vorwiegend schlechte Lebewesen aussortiert werden, um „gute“ Gene bevorzugt zu erhalten, und „schlechte“ mit der Zeit aussterben zu lassen. Häufig wird immer das schlechteste Wesen gelöscht, oft wird auch aus den schlechteren eines willkürlich ausgewählt.

einige Lebewesen werden rekombiniert zu neuen Wesen lehnt sich sehr eng an die natürliche Genetik an (Rekombination, crossing over): es werden jeweils zwei Elternteile gewählt (sinnvollerweise bevorzugt aus den tauglicheren Lebewesen), daraus werden ein oder zwei neue generiert. Bei einem erzeugten Lebewesen erfolgt dies dadurch, daß jedes Gen des neuen Wesens zufällig vom ersten oder vom zweiten Elternteil kopiert wird. Bei zwei neuen Wesen erhält das zweite genau die Gene der Eltern, die für das erste nicht verwendet wurden. Das neue Wesen (oder die beiden neuen) werden dem Lebensraum (Genpool) hinzugefügt, oder überschreiben zu löschende Elemente (siehe oben).

Man sollte nicht nur die beiden besten Lebewesen neu kombinieren, weil auch in den bisher nicht besten Wesen gute Anlagen schlummern können, die sich durch Rekombination oder Mutation vielleicht erst noch entfalten. Ebensowenig sollte man soviel rekombinieren, daß alle nicht so guten Elemente zu schnell verdrängt werden.

Geschickter ist, es einige wenige relativ gute zu rekombinieren, und dadurch einige wenige nicht so gute hinauszuwerfen. Das ganze Spiel lebt wie gesagt von einer gewissen Vielfalt!

einige Lebewesen mutieren: durch Mutation von einem oder wenigen Genen einiger Lebewesen bringt man etwas Vielfalt in den Genpool, der erst zu wirklich neuen Lösungen führen kann.

Aber auch hier ist Augenmaß angebracht: zuviel Mutation zerstört vorhandenes brauchbares Material; zuwenig verlangsamt die Evolution weil keine neuen Ideen in das System eingebracht werden.

bis gewünschte Genauigkeit erreicht, oder Rechenzeit überschritten: Die Genauigkeit der erreichten Lösung ist über die Laufzeit steuerbar.

Je nach Anforderungen führt man die Evolution fort, bis eine gewünschte Genauigkeit erreicht ist, oder bis eine zugebilligte Rechenzeit verbraucht ist. Das Erreichen einer bestimmten Genauigkeit ist allerdings nicht exakt zu erkennen, weil man das genaue (optimale) Ergebnis ja meistens nicht weiß.

Ersatzweise kann man die Genauigkeit anhand der Verbesserung des besten oder einiger der besten Lebewesen im letzten Schritt (oder besser über einige Schritte gemittelt) abschätzen. Wenn der Wert der Zielfunktion des jeweils besten Lebewesens in den ersten Generationen deutlich steigt, und dann im Verlauf weiterer Generationen offensichtlich gegen einen Höchstwert konvergiert, ist man vom Optimum nicht mehr weit entfernt.

$\boxed{\text{Ausgabe der Gene des besten Lebewesens}}$: Die Information, welche Kombination der Eingangsparameter des Problems zur besten Lösung führen, steckt in den Erbinformationen des besten Lebewesens.

(Will man die erreichte Lösung erst bewerten, und dann entscheiden, ob noch weiter optimiert werden soll –eventuell mit anderen Parametern für Mutation etc.–, dann kann es sinnvoll sein, alle Lebewesen des letzten Schrittes zu speichern, um sie gegebenenfalls wieder für die nächste Iteration einlesen zu können.)

Je nachdem, ob man die Genetik mit ins Spiel bringt, oder nicht, teilen sich die Verfahren in zwei Gruppen auf:

- Die eigentlichen Evolutionsstrategien verwenden keine genetischen Vorgänge. Vielmehr wird jedes Individuum als Satz numerischer Werte (entsprechend der Allele) betrachtet; die Optimierung findet statt durch Ändern dieser Werte und Neubewerten der Lösung. Die Änderungen werden nicht auf Bitebene durchgeführt, sondern wirklich als Änderung der numerischen Werte um bestimmte Beträge aufgefaßt. Das klassische Beispiel hierfür ist die Optimierung einer Überschalldüse durch *Rechenberg* und *Schwefel*, indem eine Folge von Ringen variablen Durchmessers einen Strömungskanal ergibt; die einzelnen Durchmesser werden nun reihum so variiert, daß der Wirkungsgrad der Düse optimal wird.

  Der Schwerpunkt liegt hier mehr auf Seite der Mutation, Rekombination kommt eher selten vor. Änderungen der Erbinformation werden in beliebig feinen Schritten durchgeführt (unbeschadet der Tatsache, daß eine reale Implementation natürlich nicht beliebig kleine Differenzen darstellen kann).

- Bei einer zusätzlichen Verwendung der genetischen Vorgänge spricht man von **genetischen Algorithmen**. Nur hier tritt eine Rekombination vorhandener Erbinformationen und willkürliche Mutation auf Bitebene auf. Dies ist die von *Holland* vorgeschlagene Vorgehensweise. Weil die Erbinformation hier nicht nur als ein Satz numerischer Werte entsprechend ihrer Bedeutung aus der Aufgabenstellung aufgefaßt, sondern auch auf Bitebene betrachtet wird, bezeichnet man die Erbinformation hier auch als *digitalen Code*.

  Durch die digitale Sichtweise wird die Erbinformation nicht beliebig fein, sondern ausschließlich in Schritten bestimmter Mindestgröße geändert.

  Der Schwerpunkt der Evolution liegt hier eher bei der Rekombination, weniger bei der Mutation.

Allerdings sind die Grenzen zwischen diesen beiden Gruppen fließend. Beispielweise kann man Mutation bei genetischen Algorithmen durchaus anhand der numerischen Entsprechung der Gene durchführen; ebenso ist bei Evolutionsstrategien eine Rekombination vorhandener Individuen denkbar, indem die numerischen Werte der Allele gemittelt werden (sogenannte *intermediäre Vererbung*). In der Praxis wird eine Mischung der beiden Reinformen meist günstiger sein als eine Form alleine; die Natur kennt keine Ideologie.

Sowohl die beiden Reinformen als auch vermischte (hybride) Verfahren führen zu folgenden charakteristischen Eigenschaften:

- Man erhält in der Regel vermutlich nicht die beste Lösung, sondern (mit Glück oder Geschick) eine gute Näherung.
- Die Zielfunktion muß nicht differenzierbar sein, sondern nur berechenbar.
- Eine Implementation ist gut vektorisierbar und parallelisierbar, und damit auch auf entsprechender Hardware effektiv zu nutzen.
- Es ist kein tiefes Verständnis der Lösung nötig; allerdings erhält man aus der gelieferten Lösung auch kein Verständnis, sondern nur numerische Ergebnisse in Form guter Gene. Insofern ist das Verfahren nicht hilfreicher als ein „gut geraten“.
- Besonders geeignet ist das Verfahren für komplexe Optimierungsprobleme, für die keine geschlossene Lösung gefunden werden kann: aerodynamische Auslegung von Flugzeugen, Tragflügeln, Verdichtern und Turbinen, Festigkeits- oder Schwingungsoptimierung und viele andere technische Aufgabenstellungen, für die zwar eine gegebene Lösung bewertet werden kann, aber kein vernünftiger Weg existiert, zu bewertende Lösungen sinnvoll zu erzeugen; weiterhin rechnerisch schwer zu handhabende Probleme aus der gesamten Informatik wie NP-vollständige Probleme (**5.4.3 Travelling Salesman Problem** auf Seite 73), falls auf die exakte Lösung verzichtet werden kann und eine gute Lösung genügt.

Weitere Anmerkungen:

- Die Zielfunktion wird häufig aufgerufen, und sollte deshalb möglichst schnell berechenbar sein
- Die Zielfunktion sollte auch sehr schlechte Lebewesen möglichst differenziert bewerten, um der Evolution beim Verbessern zu helfen.

Beispielweise ist es bei einer harten Bedingung (ein einzuhaltender Abgasgrenzwert etwa) ungeschickt, alle Lebewesen, die die Grenze überschreiten, mit dem kleinstmöglichen Wert der Zielfunktion abzustrafen. Besser wäre es, das Maß der Überschreitung mit einfließen zu lassen. Dann kann die Optimierung wesentlich zielgerichteter in Richtung besserer Lösungen gehen, die den Grenzwert einhalten.

- Der Rechenaufwand steigt mit dem Umfang der Gene jedes Lebewesens; deshalb sollte die Information in den Genen so gepackt wie möglich sein. Unnötige Redundanz kostet hier echte Rechenzeit durch überflüssige Variationen und effektive Verkleinerung des Genpools.

  Für numerische Aufgabenstellungen kann man deshalb durchaus ein eigenes Zahlenformat verwenden, das nur den in Frage kommenden Zahlenbereich in der nötigen Genauigkeit mit möglichst wenigen Bit kodiert (vielleicht Verwendung von Ganzzahlen/Festkommazahlen statt Gleitkommazahlen?), oder man verwendet Gleitkommazahlen mit der kleinsten sinnvollen Genauigkeit, falls man einen relativ großen Wertebereich abdecken muß.

- Generell gehen Details des Verfahrens (Wahrscheinlichkeiten für Mutation und Rekombination, welche Elemente sollen dafür ausgewählt werden, welche sollen sterben, wieviele Lebewesen soll es geben. . .) stark in die Geschwindigkeit und Qualität der Lösung ein; es gibt aber keine allgemein gültigen Regeln dafür. Vielmehr muß der Anwender sie geschickt wählen beziehungsweise etwas probieren.

  Dabei ist es hilfreich, die Entwicklung des Lebewesen verfolgen zu können (beziehungsweise ihre Tauglichkeit).

  Erschwerend kommt hinzu, daß sich das Ausmaß der einzelnen Operationen im Verlauf der Optimierung ändern sollte:

  – Wenn man zu Beginn der Evolution viele schlechte Wesen hat, kann man mit fleißiger Mutation schnell eine Verbesserung erreichen (egal, was man ändert: viel schlimmer kann es kaum werden, eigentlich nur besser!). Hat man dagegen etwas später bereits viele gute Lebewesen erschaffen, dann werden diese durch zu starke Mutation auch leicht wesentlich schlechter (viel besser können sie ja nicht mehr werden!). Deshalb macht es Sinn, anfangs stärker mutieren zu lassen, und später etwas weniger. In den meisten Fällen ist es auch sinnvoll, spätestens nach einigen Evolutionsschritten nur noch die schlechten Wesen stark zu mutieren, und die besseren etwas weniger.

    Starke Mutation hat also eine sehr enge Verwandtschaft zu den Monte-Carlo-Methoden, und ist zu Anfang besonders hilfreich um überhaupt erst im gesamten Definitionsraum Gebiete mit guten Verbesserungschancen zu finden.

– Analog ist es meist es günstig, sich nach einigen Generationen mehr auf Rekombination und gegebenenfalls intermediäre Vererbung zu stützen.

  Dadurch wird eine bereits erreichte Qualität erhalten und sukzessive verbessert; dies entspricht etwa dem Suchen eines lokalen Optimums ausgehend von einer nahe gelegenen Näherung.

  Es wäre also günstig, nicht nur für die Häufigkeiten von Rekombination und Mutation gute Werte zu finden, sondern für diese auch noch einen guten Verlauf über die Generationen. Dies ist in sich wieder eine Optimierungsaufgabe, die man recht einfach nebenbei erledigen kann: für die zu variierenden Wahrscheinlichkeiten baut man in die Erbinformationen noch sogenannte Regulatorgene ein, die der üblichen Evolution zeitgleich mit den eigentlichen Erbinformationen aus der Aufgabenstellung unterworfen werden (eine Gengruppe gibt in wenigen Bit an, mit welcher Wahrscheinlichkeit das Lebewesen mutiert wird, ein weiteres entsprechend die Dringlichkeit einer Rekombination). Dadurch werden sich neben der eigentlichen Lösung auch gute Werte für die Steuerung des Algorithmus herausbilden. (Diese müssen natürlich vom Algorithmus auch verwendet werden, indem sie zur Entscheidung herangezogen werden, ob mutiert oder rekombiniert werden soll.) Neben der automatischen Anpassung hat man hier auch gleich den Effekt, daß die bereits guten Individuen schwächer mutieren, weil sich das Regulatorgen auf einem niedrigen Wert einpendelt, während die schlechten Individuen einen höheren Wert im Schnitt behalten (bis sie sich verbessern). Die Werte sind also nicht nur in jeder Generation für den gesamten Genpool konstant, sondern auch noch von der Güte der einzelnen Wesen abhängig.

  In solchen Regulatorgenen bildet sich der erwähnte Konkurrenzdruck ab, der im Laufe der Entwicklung steigt.

5.4 Beispiele für Optimierung

5.4.1 Materialverbrauch von Blechdosen

Ein mögliches Beispiel ist die Optimierung des Höhen-/Breitenverhältnisses von Blechdosen.

Beim Verpacken von irgendwelchen Produkten ist der Materialverbrauch ein wesentlicher Kostenfaktor.

Es ist also interessant, in welcher Form ein bestimmtes Produkt möglichst billig verpackt werden kann.

Wenn das Produkt bereits eine vorgegebene Form hat (Buch, Drucker, Kondome), dann muß die Form der Verpackung zwangsläufig der Form des Produkts folgen.

Ist dagegen die Form des Produkts nicht vorgegeben, sondern nur die gewünschte Verpackungsmenge (Tomatenmark, Sauerkraut), dann läßt sich der Materialverbrauch optimieren, indem die Form geschickt gewählt wird.

Im folgenden soll nun versucht werden, 850 ml Tomatenmark optimal zu verpacken.

Optimal heißt in diesem Zusammenhang: billig, also mit minimalem Materialaufwand.

Den kleinstmöglichen Materialaufwand efordert bekanntlich eine Kugel. Aus praktischen Gründen (Aufwand bei der Fertigung, Schwierigkeiten beim restluftfreien Befüllen, Sauerei beim Öffnen, instabile Lage auf einem leicht schrägen Tisch beim Zelten) wird hiervon üblicherweise Abstand genommen.

Einfacher zu fertigen (und praktischer zu verwenden) ist eine zylinderförmige Verpackung: unten und oben je ein ebener kreisförmiger Deckel; dazwischen ein zu einem Zylinder gelöteter Blechstreifen.

Aus einem gegebenen Durchmesser d des Zylinders und der Dosenhöhe h kann man mit der Formel leicht das Volumen V_{Zyl} berechnen:

$$V_{\mathrm{Zyl}} = \frac{d^2\pi}{4}h \tag{5.1}$$

Jetzt ist aber das Volumen V_{Zyl} schon vorgegeben, weil die Verpackungsgrößen EU-weit in bestimmten Stufen genormt sind, und davon die 850 ml-Variante gefordert ist.

Es bleibt also noch, d und h so zu wählen, daß einerseits das geforderte Volumen erreicht wird:

$$V_{\mathrm{Zyl}} = 850\mathrm{cm}^3 \tag{5.2}$$

Andererseits soll der Blechverbrauch, also die Oberfläche A_{Zyl} (Unterseite, Oberseite, Zylinderfläche dazwischen) minimiert werden. Diese ergibt sich wiederum aus d und h über den Umfang U zu:

$$U = d \times \pi \tag{5.3}$$

$$A_{\text{Zyl}} = 2\frac{d^2 \cdot \pi}{4} + U \cdot h \tag{5.4}$$

Ziel der Optimierung ist es, die Gleichung 5.4 zu minimieren.

Um formal eine zu maximierende Zielfunktion zu bekommen, negieren wir die Gleichung 5.4 und erhalten folgende Form der Zielfunktion (U sowie $h = \frac{4V_{\text{Zyl}}}{d^2\pi}$ eingesetzt):

$$Z(d) = -(\frac{1}{2}d^2\pi + \frac{4V_{\text{Zyl}}}{d}) \tag{5.5}$$

(In dieser Funktion hat d die höchste Potenz 2 mit einem negativen Vorfaktor, es existiert also garantiert ein Maximum.)

5.4.1.1 Geschlossene Lösung

Um das Maximum zu finden, benötigt man die Nullstelle(n) der ersten Ableitung von 5.5:

$$\frac{Z(d)}{\mathrm{d}d} = -(\frac{1}{2}2d\pi + (-1)\frac{4V_{\text{Zyl}}}{d^2}) \overset{!}{=} 0 \tag{5.6}$$

Setzt man 5.1 ein, so erhält man:

$$-(d\pi + (-1)\frac{4}{d^2}\frac{d^2\pi}{4}h) \overset{!}{=} 0 \tag{5.7}$$

und damit das Ergebnis:

$$d \overset{!}{=} h \tag{5.8}$$

Erwartungsgemäß ist das Verhältnis d/h für den optimalen Fall unabhängig vom geforderten Volumen; deshalb haben kleine Dosen das selbe Verhältnis d/h wie große.

Mit Gleichung 5.8 kann man Gleichung 5.1 nach d auflösen:

$$d = \sqrt[3]{\frac{4V_{\text{Zyl}}}{\pi}} \tag{5.9}$$

Setzt man in Grundeinheiten (gem. SI) $V_{\text{Zyl}} = 840 \times 10^{-6}\text{m}^3$ ein, so ergibt sich ein optimaler Durchmesser von 0.1026694m oder etwa 10.3cm, was auch gut einer üblichen 850ml-Dose entspricht.

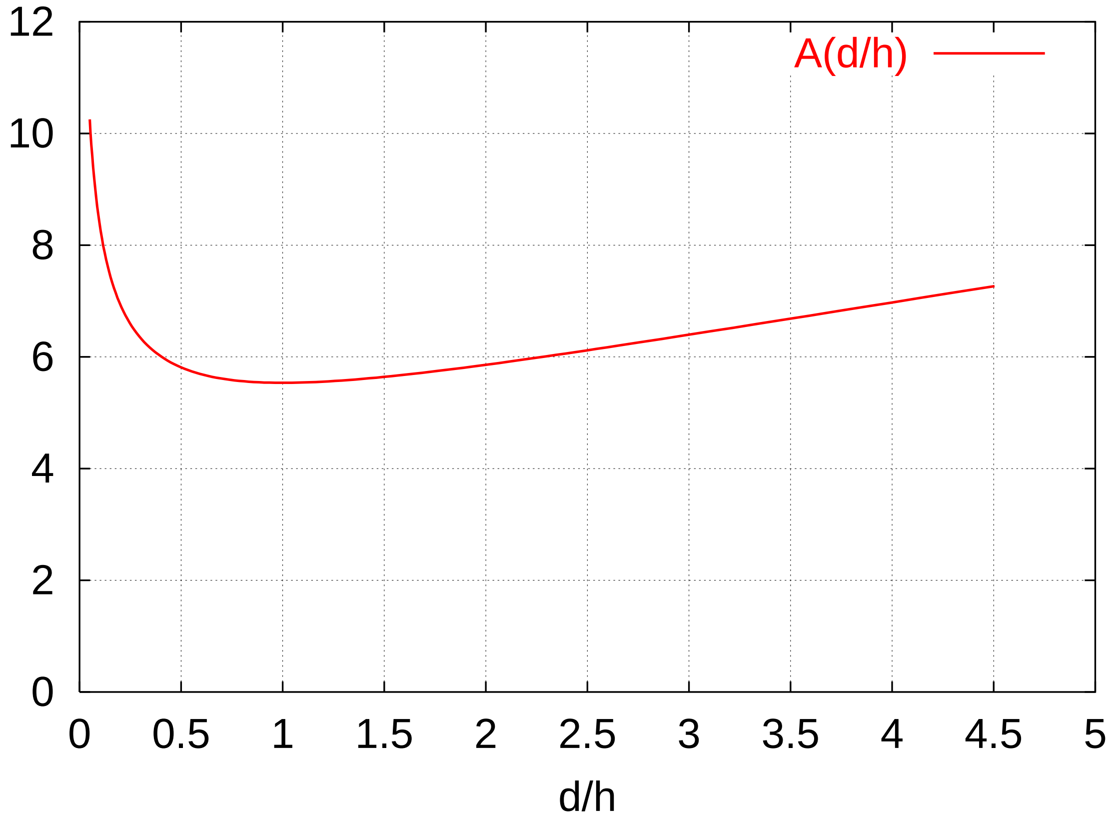


Abbildung 5.2: Fläche einer Dose über d/h

5.4.2 Lösung mithilfe eines genetischen Algorithmus

```
// Time-stamp: "04.05.03 05:42 gen_dose.cpp klaus@wachtler.de"
//
// Findet eine (fast) optimale Lösung der Aufgabenstellung
// "Blechdose eines bestimmten Volumens mit kleinster Oberfläche"
// mithilfe eines genetischen Algorithmus.
//
// Da eine Gleitkommazahl einerseits als Durchmesser, aber andererseits
// als Erbinformation mit 32 Bit verwendet wird, kann dieses Programm
// nur auf Rechnern mit 32-Bit-Darstellung für eine float und
// 4 Byte für eine int verwendet werden.
//
// Weiterhin ist es nur auf GNU-Systemen lauffähig, weil eine
// nicht standardisierte Funktion
//    int isnan(double)
// verwendet wird, um eine Gleitkommazahl auf Gültigkeit zu testen.
//
// Die Gene eines Lebewesens bestehen aus einer Gleitkommazahl,
// die den Durchmesser der Dose in cm repräsentiert (32 bit als float).
//
// Dieses Programm ist keineswegs auf Effektivität getrimmt;
// es soll nur das Prinzip verdeutlichen.
//
// Kompilieren und starten unter Linux:
// g++ -Wall gen_dose.cpp -o gen_dose && gen_dose
//
//
// Änderungen:
//
// 04.05.2003 kw  Kommentare korrigiert

#include <cfloat>
#include <cstdlib>
#include <cstdio>
#include <cmath>

using namespace std;

#define PI  3.1415926534

// liefert mit der übergebenen Wahrscheinlichkeit true, sonst false:
bool vielleicht_ja( double wahrscheinlichkeit )
{
  return rand() < (int)( wahrscheinlichkeit*RAND_MAX );
}

// liefert eine Zufallszahl zwischen 0 und (modwas-1)
int randmod( int modwas )
{
```

```
  return rand()%modwas;
}

// die Standardfunktion rand() liefert üblicherweise nicht 32 Bit
// lange Zufallswerte, sondern typischerweise 16...31 Bit, die mit
// Nullen zu einer int aufgefüllt werden (je nach RAND_MAX).
// Die folgende Funktion liefert dagegen volle 32 zufällige Bit
// (falls RAND_MAX zumindest 255 ist):
unsigned irand()
{
  return (rand()<<24) ^ (rand()<<16) ^ (rand()<<8) ^ rand();
}


#define WRSCH_MUTATION       0.01 // Wahrscheinlichkeit einer Mutation
#define GENZAHL_MUTATION     1    // Höchstzahl mutierter Gene
#define WRSCH_REKOMBINATION  0.4  // Wahrscheinlichkeit einer Rekombination
#define ANZAHL_LEBEWESEN     200  // Größe des Genpools

// Datentyp für die Gene:
typedef union { float d; unsigned igen; } gen_t;

// Datentyp für ein Lebewesen:
// Seine Gene (gen_t) sowie gleich ein Wert für die
// Zielfunktion; dann braucht diese nicht so oft neu berechnet
// zu werden:
typedef struct { gen_t gen; double fitness; } wesen_t;

// erstellt anhand eines übergebenen Gens eine Bewertung des
// Lebewesens: hoher Wert="guter Junge", niedriger Wert="untauglich".
// Diese Zielfunktion ist einfach der negative Wert der
// Dosenoberfläche.
double Zielfunktion( const gen_t &gen )
{
  const double Volumen = 850;  // eine Dose Tomatenmark, [ml] bzw. [cm^3]
  double durchmesser = gen.d;  // Durchmesser [cm] holen

  // negativer Durchmesser? Ist Unfug -> sehr untauglich
  if( isnan( durchmesser ) || durchmesser<=0.0 )
  {
    return -DBL_MAX;  // schlechter gehts nicht!
  }

  // Aus Durchmesser [cm] und Volumen [cm^3] die Höhe [cm] berechnen:
  double hoehe = Volumen/(durchmesser*durchmesser*PI/4.0);
  // Die Fläche der Dose ist 2*Kreisfläche + Zylinder:
  double flaeche     // [cm^2]
    = ( 2.0*durchmesser*durchmesser*PI/4.0   // 2*Kreisfläche
        +
        durchmesser*PI*hoehe                 // Zylinderfläche
        );
```

```
  return -flaeche;  // zu maximierender Wert
}

// Initialisiert die Lebewesen mehr oder weniger sinnvoll:
void init_wesen( wesen_t wesen[], size_t n_wesen )
{
  for( size_t i=0; i<n_wesen; i++ )
  {
    wesen[i].gen.igen = irand();  // zufälliger Wert in der Ursuppe
    wesen[i].fitness  = Zielfunktion( wesen[i].gen );
  }
}

// listet alle Wesen mit ihrer Tauglichkeit:
void zeige_population( wesen_t wesen[], size_t n_wesen )
{
  //// Ausgabe aller Lebewesen:
  //for( size_t i=0; i<n_wesen; i++ )
  //{
  //  printf( "[%3u] 0x%08x/%15.4g => %15.4g\n",
  //          i,
  //          wesen[i].gen.igen,
  //          wesen[i].gen.d,
  //          wesen[i].fitness
  //          );
  //}

  // Ausgabe des besten Werts und des Durchschnitts der 5 besten:
  double dSchnitt = 0.0;
  for( size_t i=( n_wesen>5 ? n_wesen-6 : 0 ); i<n_wesen; i++ )
  {
    dSchnitt += wesen[i].gen.d;
  }
  dSchnitt /= 5.0;
  printf( "%20.8g %20.8g\n", wesen[n_wesen-1].gen.d, dSchnitt );
}

// Vergleichsfunktion für bsearch() und qsort() für wesen_t
// anhand ihrer fitness:
int vergleiche_wesen( const void *p1, const void *p2 )
{
  if( ((wesen_t*)p1)->fitness-((wesen_t*)p2)->fitness < 0.0 )
  {
    return -1;
  }
  else if( ((wesen_t*)p1)->fitness-((wesen_t*)p2)->fitness > 0.0 )
  {
    return +1;
  }
  else
  {
```

```
    return 0;
  }
}


// sortiert die Wesen anhand ihrer Tauglichkeit
void sortiere_wesen( wesen_t wesen[], size_t n_wesen )
{
  qsort( wesen, n_wesen, sizeof(wesen_t), vergleiche_wesen );
}

// Hier wird "Lieber Gott" gespielt:
// Die Population erlebt einen Schritt der Evolution.
void evolution( wesen_t wesen[], size_t n_wesen )
{
  // Vorgehensweise:
  // Jedes Wesen des "besseren Drittels" wird mit einer
  // Wahrscheinlichkeit von WRSCH_REKOMBINATION mit einem beliebigen
  // anderen kombiniert; die beiden entstehenden neuen
  // (mit vertauschter Herkunft jedes Gens) ersetzen jeweils ein
  // Wesen aus den schlechteren zwei Dritteln.
  // Alle Wesen mutieren außerdem mit einer
  // Wahrscheinlichkeit WRSCH_MUTATION. Falls das der Fall ist,
  // werden etwa GENZAHL_MUTATION Gene gekippt.

  size_t n_wesen_1_3 = n_wesen*1/3;  // etwa 1/3 von n_wesen
  size_t n_wesen_2_3 = n_wesen*2/3;  // etwa 2/3 von n_wesen

  // Schleife über alle Wesen:
  for( size_t iWesen=0; iWesen<n_wesen; iWesen++ )
  {
    // besseres Drittel?
    if( iWesen>n_wesen_2_3 )
    {
      // Rekombination?
      if( vielleicht_ja( WRSCH_REKOMBINATION ) )
      {
        // wesen[iWesen] soll rekombiniert werden (cross over).

        // Mit welchem?
        unsigned iWesen2 = rand()%n_wesen;

        // Aber bitte nicht mit sich selbst:
        if( iWesen==iWesen2 )
        {
          --iWesen2;
        }

        // Bitmuster, anhand dessen entschieden wird,
        // welches Gen von iWesen zum ersten Kind kommt (falls Bit=1)
        // oder von iWesen2 (falls 0); jeweils das andere geht
        // zum zweiten Kind:
```

```
      unsigned muster = irand();

      // hier kommen die Kinder hin: es wird jeweils ein Kind
      // im unteren Drittel und eines in den unteren zwei Dritteln
      // abgelegt, indem dort ein je Lebewesen ins Gras beißt
      // (waren ja sowieso nicht die besten => Selektion):
      int   iNeu1 = iWesen%n_wesen_1_3;
      int   iNeu2 = iWesen2%n_wesen_2_3;

      //printf( "cross over (%3d,%3d)->(%3d,%3d); muster=0x%08x\n",
      //        iWesen,
      //        iWesen2,
      //        iNeu1,
      //        iNeu2,
      //        muster
      //        );

      // das erste Kind bekommt die Gene von iWesen, falls das
      // entsprechende Bit in muster gesetzt ist; sonst von iWesen2:
      wesen[iNeu1].gen.igen
        = ( ( wesen[iWesen].gen.igen&muster )     // Bits aus iWesen
            |
            ( wesen[iWesen2].gen.igen&(~muster) ) // Bits aus iWesen2
            );
      // Tauglichkeit neu berechnen:
      wesen[iNeu1].fitness = Zielfunktion( wesen[iNeu1].gen );


      // das zweite Kind bekommt die Gene von iWesen2, falls das
      // entsprechende Bit in muster gesetzt ist; sonst von iWesen:
      wesen[iNeu2].gen.igen
        = ( ( wesen[iWesen2].gen.igen&muster )    // Bits aus iWesen2
            |
            ( wesen[iWesen].gen.igen&(~muster) )  // Bits aus iWesen
            );
      // Tauglichkeit neu berechnen:
      wesen[iNeu2].fitness = Zielfunktion( wesen[iNeu2].gen );


    } // Rekombination

  } // besseres Drittel

  // Mutation?
  if( vielleicht_ja( WRSCH_MUTATION ) )
  {
    int AnzahlMutierterGene = randmod( GENZAHL_MUTATION );
    if( AnzahlMutierterGene<1 )
    {
      AnzahlMutierterGene = 1;
    }
```

```
      unsigned maske = 0;

      for( int i=0; i<AnzahlMutierterGene; i++ )
      {
        // Eine Bitnummer bestimmen, und das entsprechende
        // Bit in maske setzen:
        maske |= 1<<randmod( 32 );
      }

      // printf( "Mutation von %d mit maske 0x%08x\n", iWesen, maske );

      wesen[iWesen].gen.igen ^= maske;
      wesen[iWesen].fitness = Zielfunktion( wesen[iWesen].gen );

    } // Mutation

  } // Schleife über alle Wesen

}


int main( int nargs, char **args )
{
  // Die Lebewesen:
  wesen_t    wesen[ANZAHL_LEBEWESEN];

  init_wesen( wesen, ANZAHL_LEBEWESEN );
  sortiere_wesen( wesen, ANZAHL_LEBEWESEN );

  unsigned anzahl_schritte;
  if( nargs>1 )
  {
    // Argument übergeben; daraus die Anzahl Schritte lesen:
    anzahl_schritte = strtoul( args[1], NULL, 0 );
  }
  else
  {
    // keine Argumente, also abfragen:
    printf( "Schrittzahl:\n" );
    scanf( "%u", &anzahl_schritte );
  }

  for( unsigned iEvolutionsschritt=0;
       iEvolutionsschritt<anzahl_schritte;
       iEvolutionsschritt++
       )
  {
    // printf( "\nEvolutiunsschritt %d\n\n", iEvolutionsschritt );
    evolution( wesen, ANZAHL_LEBEWESEN );
    sortiere_wesen( wesen, ANZAHL_LEBEWESEN );
```

```
    zeige_population( wesen, ANZAHL_LEBEWESEN );
  }


  return 0;
} // main( int nargs, char **args )
```

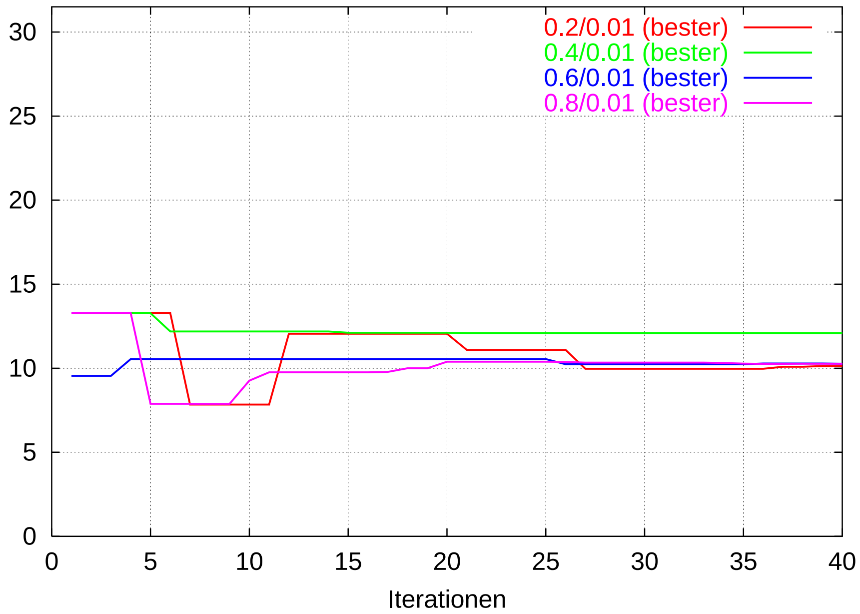


Abbildung 5.3: Variation Wahrscheinlichkeit Rekomb. (Mut.=0.01)

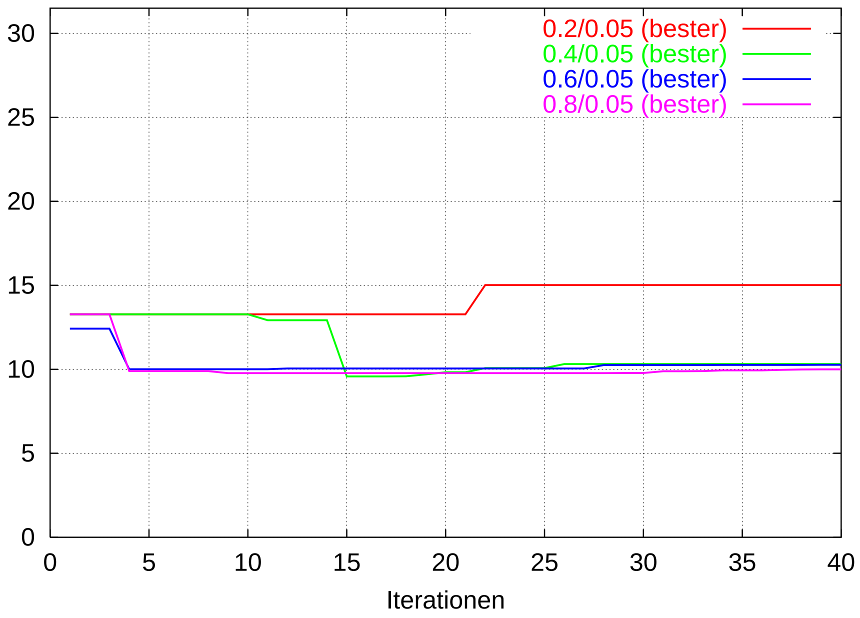


Abbildung 5.4: Variation Wahrscheinlichkeit Rekomb. (Mut.=0.05)

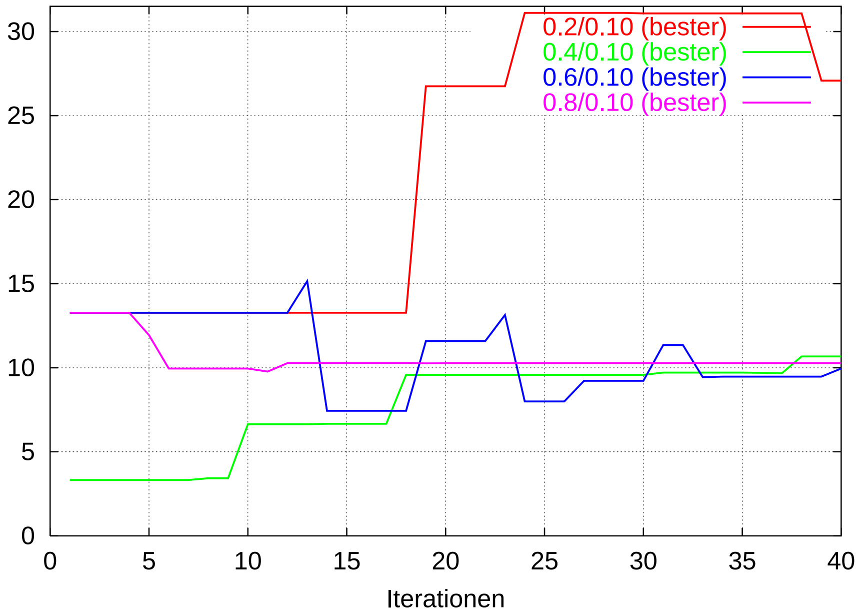


Abbildung 5.5: Variation Wahrscheinlichkeit Rekomb. (Mut.=0.10)

5.4.3 Travelling Salesman Problem

Das *Travelling Salesman Problem* („Problem des Handlungsreisenden“) ist ein prinzipiell einfach zu lösendes Problem – zumindest für kleine Problemgrößen.

Es besteht aus folgender Aufgabenstellung: Ein Handlungsreisender muß eine bestimmte Menge von Orten in einer beliebigen Reihenfolge besuchen. Zum Schluß soll er wieder am Ausgangsort sein; er muß also eine Rundreise machen. Gesucht ist eine Reihenfolge der Orte, die insgesamt einen möglichst kurzen zurückgelegten Weg ergibt.

NP-vollständig, kein schneller exakter Algorithmus bekannt und möglich

Näherungslösung würde in der Praxis in der Regel oft auch reichen.

Kapitel 6

Umsetzung von Algorithmen

6.1 Nutzen von Sprachmitteln

Die hier vorgeführten Datenstrukturen (*Feld*, *Liste*, ...) sind für sich genommen nutzlos. Erst durch die Kombination mit wirklichen Nutzdaten aus einer zu lösenden Problemstellung werden die Strukturen sinnvoll.

Um nicht jede Datenstruktur und die darauf operierenden Algorithmen für jeden möglichen Typ von Nutzdaten (`int`, `double`, `Student`, ...) neu implementieren zu müssen (`FeldInt`, `FeldDouble`, `FeldStudent`, ..., `ListeInt`, ...), strebt man ein möglichst *generisches Programmieren* an[1].

Dieses Ziel kann je nach Programmiersprache und damit je nach Denkweise auf verschiedenen Wegen erreicht werden. Auch wenn der Algorithmus selbst davon nicht beeindruckt sein sollte (er ist ja per Definition von der Implementation unabhängig), ist es doch ungeschickt, einen Algorithmus so zu definieren, daß er bei der später stattfindenden Implementation schwierig oder uneffektiv umzusetzen ist. Auch wenn sich viele Programmiersprachen in einfachen Punkten sehr ähneln (Schleifen, Vergleiche), haben sie doch individuelle Sprachmittel, die man nutzen kann – oder eben nicht, wenn man für die Implementation eines gegebenen Algorithmus die falsche Sprache wählt (falls man die Wahl überhaupt hat).

Wichtige Beispiele für solche Sprachmittel sind:

[1] In den Übungsbeispielen wird dies nur teilweise so umgesetzt, um die Programme möglichst übersichtlich zu halten

- Strukturierung von Daten im Speicher (nicht in Fortran 77; Pascal: `RECORD`, C/C++: `struct`, Java: `class`)

- Verwaltung verschiedener Varianten (Kundenversionen, ausführliche/knappe Ausgaben während der Entwicklung oder danach, Umgehen von Systemabhängigkeiten, etc.) in einem Quelltext durch Verwendung eines Präprozessors (C/C++: standardisiert, für ein Beispiel siehe die Musterlösung**A.2.1.2 C++-Version** auf Seite 105; andere durch externe Präprozessoren wie `m4` je nach Entwicklungsumgebung möglich)

- Übergabe von Daten an Unterprogramme als Kopien (*call by value*, Fortran: nein, Pascal/C/C++: ja, Java: nur bei elementaren Datentypen)

- Übergabe von Daten an Unterprogramme als Referenzen (*call by reference*, Fortran/Pascal/C++: ja, Java: nur bei Objekten), oder als Zeiger (C)

- Möglichkeiten, Daten im Speicher auf verschiedene Arten zu interpretieren (beispielsweise 32 Bit abwechselnd als ganze Zahl, als Folge von 32 ja/nein-Werten und als Gleitkommazahl); dies ist beispielsweise bei genetischen Algorithmen (**5.3 Genetische Algorithmen, Evolutionsstrategie** auf Seite 50) oder beim Datenaustausch mit fremden Rechnerarchitekturen (Meßtechnik) hilfreich, aber generell nicht portabel (Fortran: mit `EQUIVALENCE`, Pascal: variante `RECORDS`, C/C++: mit `union` oder *casts* mit Zeigern, Java: nein)

- Unterstützung von streamorientierten Dateien (nicht in Java-Applets)

- Unterstützung von positionierbaren Dateien (Direktzugriffsdateien, *random access file*, Pascal: nein, Fortran/C/C++/Java: ja, aber nicht in Java-Applets)

- Unterstützung von Binärdateien (unformatierte Dateien), bei denen Daten nicht in ihrer Textrepräsentation, sondern in der rechnerinternen Darstellung abgelegt werden (Pascal: nein, Fortran/C/C++/Java: ja, aber nicht in Java-Applets)

- Festlegen von Feldlängen erst zur Laufzeit (Fortran/Pascal: nein, C/C++: mithilfe von freiem Speicher, Java: ja)

- Objektorientiertes Programmieren, Überladen von Methoden, Vererbung (Fortran/Pascal/C: nein, C++/Java: ja)

- Möglichkeiten, das Anlegen oder Auflösen von Objekten zu beeinflussen (Konstruktoren, Destruktoren; C++: ja, Java: ja mit Einschränkungen beim Auflösen)

- Überladen von Operatoren (Fortran/Pascal/C/Java: nein, C++: ja)

- Schablonen/`template`s für Unterprogramme und Daten (Fortran/Pascal/Java: nein, C: über Makros möglich, C++: ja)
- Die Möglichkeit, komplexe Datenstrukturen als Funktionsergebnis zu liefern (Fortran: nein, Pascal/C/C++/Java: ja)
- Ausnahmebehandlung (mehr oder weniger geordneter Rücksprung zum Aufrufer im Falle eines Fehlers oder anderer besonderer Umstände, Fortran/Pascal: nein, C: bedingt über `longjmp()`, C++/Java: ja)
- Freies Allokieren von Speicher (Fortran: nein, C/C++: ja, Pascal/Java: bedingt)
- Automatische Freigabe von allokiertem Speicher (*garbage collection*, C/C++/Pascal: nein, Java: ja)
- Übergabemöglichkeit von Funktionalitäten an Unterprogramme (Funktionen beziehungsweise ihre Adressen oder Funktionsobjekte; Fortran/Pascal/C/C++/Java: ja)
- Unterstützung von Nebenläufigkeit (*thread*s, Fortran/Pascal: nein, C/C++: ja, aber nicht genormt, Java: ja)
- Standardisierte/portable Bibliotheken für Benutzeroberflächen, IPC, Datenbankzugriff, ...

(Neben den erwähnten Sprachen gibt es natürlich noch eine Unmenge weiterer, deren Nichtnennung keine Wertung sein soll: Forth, Smalltalk, LISP, Perl, Pearl, SQL, diverse Assembler, OCCAM, Spezialsprachen wie `lex`, `yacc`, `awk`, `sed`, ... Auf alle Sprachen gerecht vergleichend einzugehen, würde den Rahmen dieses Skripts völlig sprengen.)

Zusätzlich zu den genannten Ausdrucksmöglichkeiten einer Sprache ist natürlich auch wichtig, wie die hinter der Sprache stehende Denkweise zum Problem und zum Algorithmus paßt (prozedural oder beschreibend/symbolisch, 4GL).

Um diese Unterschiede in den Sprachen und die dadurch zur Verfügung stehenden Implementationsmöglichkeiten zu veranschaulichen, wird im Folgenden ein Feld betrachtet, in welchem anhand eines gegebenen Suchschlüssel ein Feldelement gesucht werden soll.

Jedes Feldelement besteht aus einem Vornamen, einem Nachnamen, und einer ganzzahligen Kennung (beispielsweise die Matrikelnummer eines Studenten).

Um den Kern dieses Abschnitts möglichst klar zu zeigen, wird der einfachste dafür mögliche Algorithmus verwendet, nämlich *lineare Suche*: das Feld wird Element für Element anhand des Suchkriteriums (also Vergleich mit dem Suchschlüssel) verglichen. Bei einer Übereinstimmung wird die Suche beendet und als erfolgreich betrachtet. Wurden alle Elemente geprüft, ohne daß eine Übereinstimmung festgestellt werden konnte, dann wird die Suche als **nicht erfolgreich** beendet.

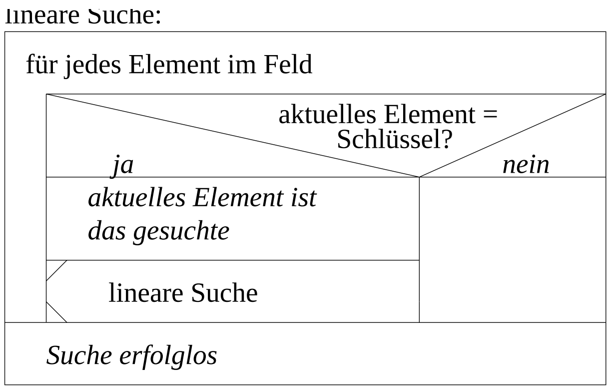


Abbildung 6.1: Struktogramm lineare Suche

TODO: Quelltexte kommentieren und vergleichen!

6.1.1 Fortran-Version

Bei Programmiersprachen aus der Prä-OO-Ära (beispielsweise C gemäß Kernighan und Ritchie oder dann später ANSI-C, ebenso wie Fortran, ALGOL und Pascal) sind Daten und die Operationen darauf nicht zwangsweise miteinander verbunden, sondern müssen bei Bedarf vom Programmierer zusammengeführt werden.

Eine Beispiellösung in Fortran 77 könnte so aussehen:

```
C     Time-stamp: "(13.05.02 19:04) suchestud.for [Klaus@Wachtler.de]"
C     Fortran77-Programm, um nach einem Studenten zu suchen
C     Unter Linux:   g77 -Wall suchestud.for -o suchestud

      PROGRAM SUSTUD

C     Da es in F77 keine Strukturen gibt, kann man auch kein Feld
C     bilden, dessen Elemente jeweils aus Vorname, Nachname, und
C     Matrikelnummer bestehen.
C     Ersatzweise bilden wir halt drei Felder; jeweils eines für jedes
C     Element.
```

```
C     Die vorhandenen Studenten:
      CHARACTER*20 VNAME(5)
      CHARACTER*20 NNAME(5)
      INTEGER      MATR(5)

C     Index für Ergebnis:
      INTEGER      IDXERG

C     Suchfunktion:
      INTEGER      SUCHE

C     Vorbelegung der Felder:
      DATA VNAME / 'Kurt', 'Hein', 'Fritz', 'Klaus', 'Sina' /
      DATA NNAME / 'Pappnase', 'Meier', 'Schmidt', 'Lahm', 'Flink' /
      DATA MATR / 978123, 987456, 998159, 988123, 977111 /

C     Unter den Studenten nach der Matrikelnummer 988123 suchen:
      IDXERG = SUCHE( MATR, 5, 988123 )

      IF( IDXERG .GT. 0 ) THEN
         PRINT*, 'gefunden: ', VNAME(IDXERG), NNAME( IDXERG )
      ELSE
         PRINT*, 'nicht gefunden!'
      ENDIF

      END


C     Sucht in dem Feld MA (mit ANZAHL Matrikelnummern) nach dem
C     Eintrag mit dem Wert KEY, und liefert den Index (1 bis ANZAHL).
C     Wenn nichts gefunden wird, dann wird -1 zurückgegeben:
      INTEGER FUNCTION SUCHE( MA, ANZAHL, KEY )

      INTEGER MA(*), ANZAHL, KEY

      INTEGER I

C     Lineare Suche über das gesamte Feld, bis gefunden:
      DO 100, I = 1, ANZAHL

         IF( MA(I) .EQ. KEY ) THEN
C           Gefunden!
            SUCHE = I
            RETURN
         ENDIF

 100  CONTINUE

C     Nicht gefunden:
      SUCHE = -1
```

```
      RETURN

      END
```

Durch das Fehlen fast aller oben aufgeführten Sprachmöglichkeiten ist das Fortranbeispiel aus heutiger Sicht sehr unelegant. Schon bei kleinen oder mittleren Programmen hat man sehr viele Fehlerquellen, die eine Änderung bestehender Programme gefährlich und aufwendig machen.

6.1.2 Pascal-Version

```
{  Time-stamp: "04.05.03 06:53 suchestud.pas klaus@wachtler.de" }
{  Pascal-Programm, um nach einem Studenten zu suchen                }
{  Unter Linux:   gpc -Wall suchestud.pas -o suchestud               }

PROGRAM Sustud(Input, Output);

{ Vereinbarung von Konstanten: }
CONST

Anzahl = 5;

{ Vereinbarung von Typen: }
TYPE

{ Definition eines Datentypen: ein Student }
Student = RECORD
             Vorname:    string(100);
             Nachname:   string(100);
             Matrikel:   integer;
          END;

{ Definition eines Feldes davon: }
ArrayStudenten = ARRAY [1..5] OF Student;

{ Sucht in dem Feld MA (mit ANZAHL Matrikelnummern) nach dem       }
{ Eintrag mit dem Wert KEY, und liefert den Index (1 bis ANZAHL). }
{ Wenn nichts gefunden wird, dann wird -1 zurückgegeben:           }
FUNCTION SucheStudent( VAR Studenten: ArrayStudenten;
                       indexl: integer;
                       indexh: integer;
                       key:       integer
                     ): integer;

VAR
i: integer;
BEGIN
```

```
    FOR i:=indexl TO indexh DO
    BEGIN
       IF ( Studenten[i].Matrikel = key ) THEN
       BEGIN
           return i;
       END;
    END;
    return -1;
END;

{ Variablen für das Hauptprogramm: }
VAR

  Studenten: ArrayStudenten;
  index_erg: integer;

{ Beginn des Hauptprogramms: }
BEGIN

    { Vorbelegen des Feldes: }
    Studenten[1].Matrikel := 978123;
    Studenten[1].Nachname := 'Pappnase';
    Studenten[1].Vorname  := 'Kurt';
    Studenten[2].Matrikel := 987456;
    Studenten[2].Nachname := 'Meier';
    Studenten[2].Vorname  := 'Hein';
    Studenten[3].Matrikel := 998159;
    Studenten[3].Nachname := 'Schmidt';
    Studenten[3].Vorname  := 'Fritz';
    Studenten[4].Matrikel := 988123;
    Studenten[4].Nachname := 'Lahm';
    Studenten[4].Vorname  := 'Klaus';
    Studenten[5].Matrikel := 977111;
    Studenten[5].Nachname := 'Flink';
    Studenten[5].Vorname  := 'Sina';

    { Unter den Studenten nach der Matrikelnummer 988123 suchen: }
    index_erg := SucheStudent( Studenten, 1, Anzahl, 988123 );

    writeln( 'gefunden: ',
             index_erg, ' ',
             Studenten[index_erg].Vorname, ' ',
             Studenten[index_erg].Nachname
           );
END.
```

6.1.3 C-Version

```
// Time-stamp: "(13.05.02 22:33) suchestud.c [Klaus@Wachtler.de]"
// demonstriert generisches Programmieren (hier: Suche in einem
// Feld mit unbekannten Datentypen) in Anlehnung an bsearch()
// aus der Standardbibliothek von ANSI-C.
//
// Literatur:
// [1] "0 ist false, alles andere true"
//     Klaus Wachtler 2002

#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

// Lineare Suche in einem Feld von Werten unbekannten Inhalts;
// entnommen aus [1];
// Aufruf identisch zu bsearch() aus der Standardbibliothek
// (allerdings muß das Feld nicht sortiert sein, da lineare
// Suche):
void *lsearch( const void *schluessel,
               const void *feldanfang,
               size_t elemente,
               size_t elementgroesse,
               int(* vergleichsfkt)( const void *e1,
                                     const void *e2
                                     )
               )
{
  char       *feld_p = (char *)feldanfang;

  /* Parameter pruefen: */
  if( !( schluessel
         && feldanfang
         && elemente
         && elementgroesse
         && vergleichsfkt
         )
      )
    return NULL;

  /* Schleife ueber alle Elemente: */
  while( elemente-- )
  {
    /* trifft das aktuelle Element zu? */
    if( !(*vergleichsfkt)( schluessel, feld_p ) )
      return feld_p;
    /* Wenn nicht, dann naechstes Element: */
    feld_p += elementgroesse;
  }
```

```

  /* Ende des Feldes und trotzdem nichts gefunden? */
  return NULL;
}


// Testprogramm
// ------------
//

/* Datentyp für einen Studenten: */
typedef struct
{
  char *vname;
  char *nname;
  int   matrikelnummer;
}
student_t;

// Vergleichsfunktion für zwei Studenten (liefert einen Wert
// kleiner 0, wenn die Matrikelnummer des ersten kleiner ist
// als die des zweiten, oder 0, wenn beide gleich sind, sonst
// einen Wert >0):
//
int vergleiche_student_t_matrikelnummer( const void *p1,
                                         const void *p2
                                         )
{
  return ( ((student_t*)p1)->matrikelnummer
           -
           ((student_t*)p2)->matrikelnummer
          );
}

int main( int nargs, char **args )
{
  /* Feld mit ein paar Studenten: */
  student_t    studenten[5] =
  {
    { "Kurt", "Pappnase", 978123 },
    { "Hein", "Meier", 987456 },
    { "Fritz", "Schmidt", 998159 },
    { "Klaus", "Lahm", 988123 },
    { "Sina", "Flink", 977111 },
  };

  /* Zeiger auf gefundenes Element: */
  student_t   *p_student = NULL;

  /* Schlüssel, nach dem gesucht wird (VN+NN egal): */
  student_t    key = { "", "", -1 };
```

```

  /* Suche nach Student mit der Matrikelnummer 988123: */
  key.matrikelnummer = 988123;
  p_student = lsearch( &key,
                       studenten,
                       sizeof(studenten)/sizeof(studenten[0]),
                       sizeof(studenten[0]),
                       vergleiche_student_t_matrikelnummer
                       );

  if( p_student==NULL )
  {
    printf( "nichts gefunden!\n" );
  }
  else
  {
    printf( "gefunden: %s %s %d\n",
            p_student->vname,
            p_student->nname,
            p_student->matrikelnummer
           );
  }

  return 0;
} /* main( int nargs, char **args */
```

6.1.4 C++-Version

```
// Time-stamp: "(13.05.02 22:39) SucheStudenten.cpp [Klaus@Wachtler.de]"
//
// Demonstriert eine Containerklasse in C++ (realisiert als template)

#include <iostream>
#include <string>

using namespace std;

// Klasse der einzelnen Elemente:
class Student
{
public:

  // Konstruktor
  Student( string v = "", string n = "", int m = -1 )
    : Vorname(v ),
      Nachname( n ),
```

```
      Matrikelnummer( m )
  {
  }

  // copy-Konstruktor
  Student( const Student &rechteSeite )
    : Vorname( rechteSeite.Vorname ),
      Nachname( rechteSeite.Nachname ),
      Matrikelnummer( rechteSeite.Matrikelnummer )
  {
  }

  // Destruktor
  virtual ~Student()
  {
  }

  // Zuweisung Student = Student
  Student & operator=( const Student &rechteSeite )
  {
    Vorname = rechteSeite.Vorname;
    Nachname = rechteSeite.Nachname;
    Matrikelnummer = rechteSeite.Matrikelnummer;

    return *this;
  }

  // Ausgabe auf einen ostream
  virtual void streamout( ostream & s ) const
  {
    s << Vorname;
    s << ",";
    s << Nachname;
    s << ",";
    s << Matrikelnummer;
  }

  // Kopie von sich selbst allokieren
  virtual Student &clone() const
  {
    return *(new Student(*this));
  }

  bool operator==( const Student &rS ) const
  {
    return Matrikelnummer==rS.Matrikelnummer;
  }

protected:
```

```
private:

  // Nutzdaten:
  string Vorname;
  string Nachname;
  int    Matrikelnummer;


}; // class Student...

// ggf. nach Student.cpp kopieren:
ostream &operator<<( ostream &s, const Student &obj )
{
  obj.streamout( s );
  return s;
}


template <class T> class Feld
{
public:

  // Konstruktor
  Feld( int n = 0 )
    : Nutzdaten( NULL ),
      Anzahl( n )
  {
    if( n>0 )
    {
      Nutzdaten = new T[n];
    }
  }

  // Destruktor
  virtual ~Feld()
  {
    delete[] Nutzdaten;
  }

  void setzeElement( int index, const T &t )
  {
    Nutzdaten[index] = t;
  }

  const T&sucheLinear( const T&key ) const
  {
    for( int i=0; i<Anzahl; i++ )
    {
      if( key==Nutzdaten[i] )
      {
        return Nutzdaten[i];
```

```
      }
    }
    throw "nix gefunden!";
  }

private:

  // Nutzdaten
  T    *Nutzdaten;

  // Anzahl der Elemente:
  int   Anzahl;


}; // class Feld...



int main( int nargs, char **args )
{
  try
  {
    Feld<Student>  f( 5 );
    f.setzeElement( 0, Student( "Kurt", "Pappnase", 978123 ) );
    f.setzeElement( 1, Student( "Hein", "Meier", 987456 ) );
    f.setzeElement( 2, Student( "Fritz", "Schmidt", 998159 ) );
    f.setzeElement( 3, Student( "Klaus", "Lahm", 988123 ) );
    f.setzeElement( 4, Student( "Sina", "Flink", 977111 ) );

    cout << " gefunden: "
         << f.sucheLinear( Student( "", "", 988123 ) )
         << endl;
  }
  catch( const char *Fehler )
  {
    cerr << Fehler << "\n";
  }
  catch( ... )
  {
    cerr << "???\n";
  }


  return 0;
} // main( int nargs, char **args )
```

6.1.5 Java-Version

In Java können Containertypen implementiert werden, indem man einen Datentyp schafft, der alle nötigen Verwaltungsinformationen für ein Element beinhaltet, sowie eine Referenz auf die eigentlichen Nutzdaten.

Als Testdatentyp sieht der Student folgendermaßen aus (diese Klasse wird übrigens auch für die Musterlösung **A.2.7.1 Java-Version** auf Seite 176 zur Aufgabe **A.1.7 Geschlossenes Hashing** auf Seite 97 und andere verwendet):

```
// Time-stamp: "(13.05.02 21:46) Student.java [Klaus@Wachtler.de]"
//
// Testklasse zum Einfügen in einen Container

import java.util.*;

public class Student
{
    private String   VorName;
    private String   NachName;
    public  int      Matrikelnummer;

    public Student()
    {
        VorName = NachName = "?";
        Matrikelnummer = -1;
    }

    public Student( String v, String n, int m )
    {
        VorName = v;
        NachName = n;
        Matrikelnummer = m;
    }

    public String toString()
    {
        return "<" + VorName + " "
               + NachName + " "
               + Matrikelnummer  + ">";
    }

    // vergleicht zwei Studenten auf Gleichheit
    public boolean equals( Object s )
    {
        return Matrikelnummer==((Student)s).Matrikelnummer;
    }


```

```
} // class Student
```

Meine Implementation des generischen Feldes mit linearer Suche ist folgende:

```
// Time-stamp: "(13.05.02 21:48) Feld.java [Klaus@Wachtler.de]"
//
// Containerklasse für ein einfaches Feld mit linearer Suche

import java.util.*;

public class Feld
{
    private Object[] nutzdaten;
    private int anzahl = 0;

    // Konstruktor:
    // bekommt gewünschte Anzahl Feldelemente übergeben
    public Feld( int n )
    {
        anzahl = n;
        nutzdaten = new Object[anzahl];
    }

    public void setzeElement( int index, Object o )
    {
        nutzdaten[index] = o;
    }

    public Object sucheLinear( Object o )
    {
        for( int i=0; i<anzahl; i++ )
        {
            if( nutzdaten[i].equals( o ) )
            {
                return nutzdaten[i];
            }
        }
        return null;
    }

} // class Feld
```

Das Testprogramm könnte so aussehen:

```
// Time-stamp: "(14.05.02 00:35) teste_SucheStudenten.java [Klaus@Wachtler.de]"
```

```
/*  2 */ //
/*  3 */ // Testprogramm für Container-Klasse Feld
/*  4 */ //
/*  5 */ // javac teste_SucheStudenten.java Student.java Feld.java
/*  6 */ // java teste_SucheStudenten
/*  7 */
/*  8 */ import java.util.*;
/*  9 */
/* 10 */ public class teste_SucheStudenten
/* 11 */ {
/* 12 */     public static void main( String args[] )
/* 13 */     {
/* 14 */         Feld   f = new Feld( 5 );
/* 15 */         f.setzeElement( 0, new Student( "Kurt", "Pappnase", 978123 ) );
/* 16 */         f.setzeElement( 1, new Student( "Hein", "Meier", 987456 ) );
/* 17 */         f.setzeElement( 2, new Student( "Fritz", "Schmidt", 998159 ) );
/* 18 */         f.setzeElement( 3, new Student( "Klaus", "Lahm", 988123 ) );
/* 19 */         f.setzeElement( 4, new Student( "Sina", "Flink", 977111 ) );
/* 20 */
/* 21 */         Student schluessel = new Student( "", "", 987456 );
/* 22 */
/* 23 */         System.out.println( f.sucheLinear( schluessel ) );
/* 24 */
/* 25 */         System.out.println( "ok" );
/* 26 */     }
/* 27 */
/* 28 */ } // public class teste_SucheStudenten
```

Anmerkung: In Java ist es üblich, Container (also allgemeine Datentypen wie Listen, Bäume, Vektoren, etc., die wiederum Mengen von anderen Daten, den eigentlichen Nutzdaten, verwalten) mit einer *Referenzsemantik* auszustatten; siehe **3.1.1 Referenzsemantik, Wertsemantik** auf Seite 21.

Anhang A

Übungen

A.1 Aufgaben

A.1.1 Größter gemeinsamer Teiler

Folgende Teilaufgaben sollen gelöst werden:

- Verwendung des Euklidschen Algorithmus

  1. Der in **2.3 Wie formuliert man einen Algorithmus?** auf Seite 7 vorgestellte Euklidsche Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen soll in Java oder C formuliert werden.

     Ein Testprogramm soll für folgende Gruppen von Wertepaaren den Algorithmus durchführen:

     (A) Alle Kombinationen von $m = 1 \ldots 20$ mit $n = 1 \ldots 20$

     (B) Alle Kombinationen von $m = 1001 \ldots 1020$ mit $n = 1001 \ldots 1020$

     (C) Alle Kombinationen von $m = 1000001 \ldots 1000020$ mit $n = 1000001 \ldots 1000020$

  2. Das Programm soll so erweitert werden, daß die durchschnittliche Laufzeit für jede ggT-Bestimmung jeweils in Gruppe (A), (B) und (C) ausgegeben wird (Hinweis: in Java `long System.currentTimeMillis()`, in C `clock()`)

     Anmerkung: Wahrscheinlich ist die Auflösung der internen Rechneruhr zu grob für eine sinnvolle Messung. Dann muß jeweils um den interessierenden Bereich noch eine Schleife gebaut werden, und die *gemessene Laufzeit durch die Anzahl der Schleifendurchläufe geteilt werden. Weiterhin muß man beim Vergleichen der Werte beachten, daß die Java-Funktion* `currentTimeMillis()` *die insgesamt verstrichene Zeit liefert, also einen höheren Wert bei starker Auslastung des Systems mit weiteren Programmen, während die C-Funktion* `clock()` *nur die Zeit angibt, die für den aktuellen Prozeß verbraucht wird.*

3. Das Programm ist so zu erweitern, daß getrennt für die Gruppen (A), (B) und (C) jeweils die Anzahl
   - der durchgeführten Moduloberechnungen
   - der durchgeführten Vergleiche
   - der durchgeführten Zuweisungen
   - der durchgeführten Additionen und Subtraktionen
   - der durchgeführten Vertauschungen

   gezählt wird und (bezogen auf die Anzahl der ggT-Berechnungen) ausgegeben werden.

   Anmerkung: die gegebenenfalls noch ausgegebenen Laufzeiten sind nicht mehr sinnvoll, weil sie das Zählen der einzelnen Anweisungen beinhalten.

- Zum Vergleich sollen alle obigen Teilaufgaben für folgenden Algorithmus durchgeführt werden, der ebenfalls den ggT zweier Zahlen m und n liefert:

```
eigener ggT:
+-----------------------------------+
|               m>n                 |
|                |                  |
+--- ja  --------+--- nein --------+
| Vertausche m,n |                  |
+-----------------------------------+
|  für i = m ... 1  (-1)            |
|   +-------------------------------+
|   |  m/i ohne Rest teilbar        |
|   |      UND                      |
|   |  n/i ohne Rest teilbar        |
|   |          |                    |
|   +-- nein --+-- ja -------------+
|   |          |                    |
|   |          | ggT = i            |
|   |          +--------------------+
|   |          | <- eigener ggT     |
+---+----------+--------------------+
```

A.1.2 Fakultät

Es ist jeweils die rekursive und iterative Version der Berechnung einer Fakultät zu implementieren.

Dabei ist die Laufzeit zu vergleichen.

A.1.3 Summenbildung

Es ist eine Funktion zu schreiben, die für ein gegebenes n alle Zahlen von 1 bis n aufsummiert.

Weiterhin ist eine Funktion zu schreiben, die dasselbe Ergebnis mit der Formel von Gauß liefert:

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2}$$

Die Laufzeit beider Versionen ist über n grafisch aufzutragen.

A.1.4 Binäre Suche

(aus [v.Helden] übernommen)

Das Suchverfahren „Binäres Suchen“ arbeitet nach folgendem Algorithmus:

Algorithmus: Binäre Suche im Array `a[min .....max]  // min = 0,  max = n`

1. wähle ein mittleres Element des Arrays a[mid]
2. `if (a[mid].key == gesuchterKey)`
   Suche erfolgreich
   `else`
   `if (a[mid].key  < gesuchterKey)`
   betrachte die rechte Hälfte des Arrays `mid .. max`
   `else`
   betrachte die linke Hälfte des Arrays `0 .. mid`
3. `if (betrachtetes Array  == leer)`
   Suche erfolglos
   `else`
   beginne bei Schritt 1

Implementieren Sie den Algorithmus so, daß in einem sortierten Feld von Integer-Zahlen gesucht werden kann. Geben Sie jedesmal, wenn Sie ein Teilfeld durchsuchen, die Indizes der Grenzen und des mittleren Elements aus. Geben Sie am Ende den Index des gefundenen Elements bzw. eine Meldung für die erfolglose Suche aus.

Testen Sie Ihren Algorithmus zunächst mit einfachen Feldern, die Sie direkt im Programmcode initialisieren.

Erweitern Sie ihr Programm so, daß die zu durchsuchende Zahlenfolge aus einer Datei eingelesen werden kann. Hierzu sollen Sie folgende Ein-/Ausgabe realisieren:

- Lesen Sie den Namen der Datei, in der die zu durchsuchende Zahlenfolge steht, von der Tastatur ein.
- Lesen Sie die Zahlenfolge aus der Datei ein.

- Das Programm soll selbständig herausfinden, wieviele Elemente die Zahlenfolge enthält und das Feld entsprechend dimensionieren.
- Gehen Sie davon aus, daß die Elemente in der einzulesenden Datei jeweils in einer Zeile stehen.
- Lesen Sie die zu suchende Zahl von der Tastatur ein.
- Fangen Sie Fehler beim Einlesen ab und überprüfen Sie, ob die eingelesene Zahlenfolge sortiert ist.

Geben Sie zum Schluß den Algorithmus richtig an (im Gegensatz zum oben gegebenen).

A.1.5 Einfach verkettete Liste

Es ist eine einfach verkettete Liste zu implementieren und zu testen, mit den Funktionen:

- Einfügen am Anfang
- Einfügen am Ende
- Einfügen vor einem gegebenen Element
- Einfügen hinter einem gegebenen Element
- Suche nach einem bestimmten Element
- Löschen eines bestimmten Elements

A.1.6 Einfach verkettete Liste eines beliebigen Typs

In der Übungsaufgabe für *einfache Listen* (siehe **A.1.5 Einfach verkettete Liste** auf Seite 96) wurde eine Liste gebaut, die nur jeweils einen bestimmten Typ von Daten (in der Beispiellösung: `int`) verwalten kann.

Diese Liste soll nun so umgebaut werden, daß in der Klasse für die Liste **nicht** vorgegeben wird, welchen Typ die verwalteten Daten haben. Vielmehr soll es die Lösung dieser Aufgabe sinnvoll ermöglichen, eine Liste von Studenten oder eine Liste von Autos oder sonstwas anlegen zu können.

Hinweis Java: Ein möglicher Lösungsweg besteht darin, in jedem Element der Liste als Typ der Nutzdaten nur `Object` zu verwenden, und zur Suche nach einem bestimmten Element die Methode `equals()` der Klasse `Object` zu verwenden.
Zusatzfrage: könnte man analog auch einen binären Suchbaum in Java implementieren, wobei der Typ der Knoten frei ist?

Hinweis C++: Für solche Problemstellungen werden in der Standardbibliothek sogenannte `template`s verwendet, was hier sicher auch sinnvoll ist; siehe [KW-C++].

A.1.7 Geschlossenes Hashing

Es ist eine Hashtabelle zu bauen, in die Objekte einer Klasse `Student` (bestehend aus Vorname, Nachname, und Matrikelnummer) eingefügt werden können, und anhand ihrer Matrikelnummer wiedergefunden werden können. Dabei ist zur Auflösung von Kollisionen *lineares Sondieren zu verwenden*.

A.1.8 Offenes Hashing

Es ist eine Hashtabelle zu bauen, in die Objekte einer Klasse `Student` (bestehend aus Vorname, Nachname, und Matrikelnummer) eingefügt werden können, und anhand ihrer Matrikelnummer wiedergefunden werden können.

Es soll *offenes Hashing* verwendet werden; dafür sollen die selbst implementierten einfach verketteten Listen (siehe Aufgabe **A.1.6 Einfach verkettete Liste eines beliebigen Typs** auf Seite 97 bzw. die Lösungen in **A.2.6 Einfach verkettete Liste eines beliebigen Typs** auf Seite 152) verwendet werden.

A.1.9 Einfache Integration

Es ist eine beliebige Funktion $f(x)$ zu integrieren, indem ein gegebener Bereich $[a, b]$ der Funktion in eine Anzahl von Streifen zerlegt wird, und die Flächen aller Streifen aufsummiert werden (sinnvollerweise kann man als Höhe jeden Streifens den Funktionswert in der Mitte des Streifens verwenden).

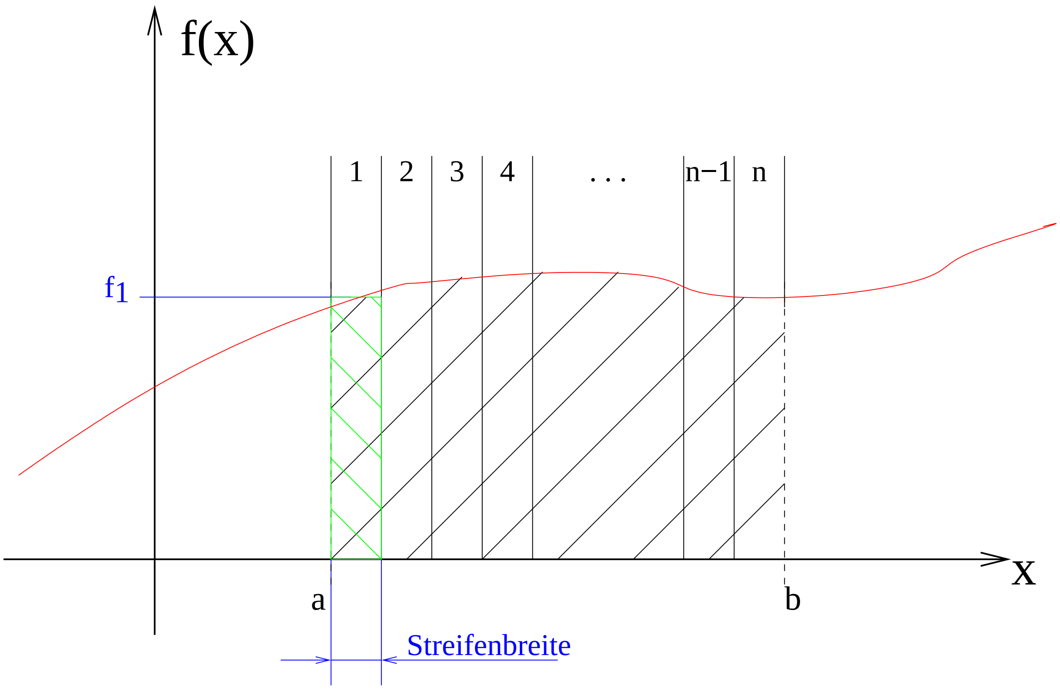


Abbildung A.1: Einfache Integration durch Näherung mit Rechtecken

Mit Δx für die Streifenbreite und n für die Anzahl der Streifen ist eine Näherung des Integrals damit:

$$\int_b^b f(x) \approx \sum_{i=1}^{n} f(x_i) \times \Delta x \tag{A.1}$$

Dabei soll ein vernünftiger Weg umgesetzt werden, die zu integrierende Funktion von der Berechnung des Integrals zu trennen. Das heißt, die zu integrierende Funktion soll möglichst leicht austauschbar sein, ohne das Integrieren an sich ändern zu müssen.

Hinweis Java: Man kann eine abstrakte Basisklasse definieren, in der alles zum Integrieren Nötige implementiert wird; nur eine Methode darin steht für die zu integrierende Funktion und ist nur deklariert. In einer davon abgeleiteten Klasse wird dann genau diese Methode überschrieben mit der tatsächlich zu integrierenden Funktion. Um jetzt andere Funktionen zu integrieren, bildet man von der abstrakten Basisklasse weitere Ableitungen, in denen diese Methode entsprechend implementiert wird.

Alternativ könnte man ein sogenanntes *Funktionales Objekt* verwenden. Das ist ein Objekt einer Klasse, die typischerweise keine (oder nur wenig) Daten enthält, sondern vielmehr nur eine Methode (oder bei Bedarf auch mehrere) beinhaltet. Die Klasse, in der die eigentlich Integration implementiert ist, bekommt über den Konstruktor oder spätestens zum Start der Rechnung ein solches Objekt übergeben, und holt sich über die darin definierte Methode die Funktionswerte.

Hinweis C/C++: Die integrierende Funktion/Methode kann eine Adresse einer anderen Funktion als Parameter übergeben bekommen, die wiederum die zu integrierende Funktion repräsentiert, ähnlich wie die Funktion `bsearch()` (zur binären Suche, siehe **A.2.4.3 C/C++-Version (verallgemeinert)** auf Seite 129 und [KW-C]) aus der Standardbibliothek eine Vergleichsfunktion übergeben bekommt. Beim Aufruf der Funktion zum Integrieren übergibt man dann einfach die zu integrierende Funktion als Parameter.
In C++ könnte man alternativ ebenfalls Funktionsobjekte verwenden, die aber etwas umständlicher zu handhaben sind (und auch etwas mehr Rechenzeit kosten); siehe dazu auch [KW-C++].

(Anmerkung: Um wirklich sinnvoll zu integrieren, ist dieses einfache Verfahren sicher nicht optimal. Bessere Verfahren finden sich beispielsweise in [Muellges/R.].)

A.1.10 Maximum einer Funktion

Es ist mit einem selbstgeschriebenen Programm das Maximum der Funktion:

$$f(x) = 15 + 13x - \frac{1}{2}x^2$$

zu suchen (und zu finden).

A.1.11 Einen String in einem anderen suchen

Gesucht ist eine Methode, in einem String einen zweiten (kürzeren) möglichst effektiv zu suchen.

A.2 Lösungen

A.2.1 Größter gemeinsamer Teiler

Die Musterlösung erfüllt die Aufgabenstellung für C++ vollständig in einem Programm, welches mit Präprozessordirektiven wahlweise so kompiliert werden kann, daß die Laufzeit gemessen wird (durch Aktivieren der Zeile

`#define MODUS  MESSE_LAUFZEIT`

im Quelltext oder durch Aufruf des Compilers mit der Option `-DMODUS=1`), oder so daß die Anzahl der Anweisungen mitprotokolliert wird (mit der Zeile

`#define MODUS  ZAEHLE_ANWEISUNGEN`

beziehungsweise `-DMODUS=2`).

Weiterhin werden in der C++-Version bei der Laufzeitmessung nicht nur die für die Gruppen A, B, und C geforderten Kombinationen von m und n getestet, sondern noch mehr dazwischenliegende; und die so ermittelten Laufzeiten werden (getrennt nach Euklid und dem einfachen Algorithmus) in einer Datei hinterlegt. Aus diesen Dateien wurde dann mit `gnuplot` das Bild A.2 generiert.

Die Java-Version ist dagegen nur so gebaut, daß die Laufzeit bestimmt wird. Die Vorgehensweise ebenso wie ausgegebene Anzahl der Anweisungen würde der C++-Version entsprechen; deshalb habe ich mir die Mühe erspart.

Der erste Aufgabenteil (Umsetzen des Algorithmus) ist jeweils implizit enthalten.

A.2.1.1 Java-Version

```
// Time-stamp: "(24.03.02 21:04) Euklid.java [Klaus@Wachtler.de]"
//
// Kleines Javaprogramm zur Demonstration des Euklidschen Algorithmus
// im Vergleich zu einem einfachen Algorithmus (stumpfisttrumpf()).

import java.util.*;

class Euklid
{

    //
    //       Euklidscher Algorithmus:
    //       +-------------------------------------+
    //       |                                     |
    //       | +-----------------------------------+
```

```
//       | | r = m mod n                       |
//       | +-----------------------------------+
//       | |               r = 0 ?             |
//       | |                  |                |
//       | |----- ja ---------+---- nein --------+
//       | |                  |                |
//       | | Ergebnis = n     | m = n          |
//       | +------------------+-----------------+
//       | | <--Euklidsch. A. | n = r          |
//       | +------------------+-----------------+
//       |                                     |
//       +-------------------------------------+
//

static int euklid( int m, int n )
{
    int r;


    while( true )
    {
        r = m % n;

        if( r==0)
        {
            return n;
        }
        else
        {
            m = n;
            n = r;
        }
    }

} // int euclid( int m, int n )


//
//       stumpfisttrumpf:
//       +----------------------------------+
//       |              m>n                 |
//       |               |                  |
//       +--- ja  --------+--- nein ----------+
//       | Vertausche m,n |                 |
//       +----------------------------------+
//       |  für i = m ... 1  (-1)           |
//       |   +------------------------------+
//       |   |  m/i ohne Rest teilbar       |
//       |   |      UND                     |
//       |   |  n/i ohne Rest teilbar       |
//       |   |          |                   |
```

```
//      |   +-- nein --+-- ja ---------------+
//      |   |          |                     |
//      |   |          | ggT = i             |
//      |   |          +---------------------+
//      |   |          | <- stumpfisttrumpf  |
//      +---+----------+---------------------+
//
//

static int stumpfisttrumpf( int m, int n )
{
    if( m>n )
    {
        int tmp = m;
        m = n;
        n = tmp;
    }

    for( int i=m; m>=1; i-- )
    {
        if( ( m % i ) == 0 )
        {
            if( ( n % i ) == 0 )
            {
                return i;
            }
        }
    }

    return 0; // eigentlich egal, aber vermeidet Compilerwarnungen

} // int stumpfisttrumpf( int m, int n )


public static void kombiniere_m_n_euklid( int mvon, int mbis,
                                          int nvon, int nbis,
                                          int NTIME
                                          )
{
    long start = System.currentTimeMillis();
    for( int i=0; i<NTIME; i++ )
    {
        for( int m=mvon; m<=mbis; m++ )
        {
            for( int n=nvon; n<=nbis; n++ )
            {
                int erg = euklid( m, n );
            }
        }
    }
    long ende = System.currentTimeMillis();
```

```
    System.out.println( "euklid: m = "
                        + mvon
                        + ".."
                        + mbis
                        + "; n = "
                        + nvon
                        + ".."
                        + nbis
                        + "  Zeit = "
                        + ( (double)( ende - start )
                            /NTIME
                            /( (nbis-nvon+1)*(mbis-mvon+1) )
                            )
                        + " msec"
                        );
}

public static void kombiniere_m_n_stumpf( int mvon, int mbis,
                                          int nvon, int nbis,
                                          int NTIME
                                          )
{
    long start = System.currentTimeMillis();
    for( int i=0; i<NTIME; i++ )
    {
        for( int m=mvon; m<=mbis; m++ )
        {
            for( int n=nvon; n<=nbis; n++ )
            {
                int erg = stumpfisttrumpf( m, n );
            }
        }
    }
    long ende = System.currentTimeMillis();
    System.out.println( "stumpf: m = "
                        + mvon
                        + ".."
                        + mbis
                        + "; n = "
                        + nvon
                        + ".."
                        + nbis
                        + "  Zeit = "
                        + ( (double)( ende - start )
                            /NTIME
                            /( (nbis-nvon+1)*(mbis-mvon+1) )
                            )
                        + " msec"
                        );
}
```

```
    public static void main( String args[] )
    {
        kombiniere_m_n_euklid( 1, 20, 1, 20, 1000 );
        kombiniere_m_n_euklid( 1001, 1020, 1001, 1020, 1000 );
        kombiniere_m_n_euklid( 1000001, 1000020, 1000001, 1000020, 1000 );

        kombiniere_m_n_stumpf( 1, 20, 1, 20, 10 );
        kombiniere_m_n_stumpf( 1001, 1020, 1001, 1020, 1 );
        kombiniere_m_n_stumpf( 1000001, 1000020, 1000001, 1000020, 1 );
    }


} // class Euklid
```

Ausgabe der Laufzeiten:

```
euklid: m = 1..20; n = 1..20  Zeit = 1.575E-4 msec
euklid: m = 1001..1020; n = 1001..1020  Zeit = 1.7E-4 msec
euklid: m = 1000001..1000020; n = 1000001..1000020  Zeit = 1.7250000000000002E-4 msec
stumpf: m = 1..20; n = 1..20  Zeit = 0.0010 msec
stumpf: m = 1001..1020; n = 1001..1020  Zeit = 0.035 msec
stumpf: m = 1000001..1000020; n = 1000001..1000020  Zeit = 34.0925 msec
```

A.2.1.2 C++-Version

```
// Time-stamp: "16.03.04 16:46 euklid.cpp klaus@wachtler.de"
//
// Euklidscher Algorithmus oder einfache Bestimmung des ggT;
// wahlweise mit Messung der Laufzeiten
// oder Anzahl der verschiedenen Operationen
//
// ANSI-C++
//
// Linux:    g++ -Wall euklid.cpp && a.out
//
// VC++ 6.0: cl /Zi /GX auklid.cpp
//           .\euklid


#define MESSE_NIX           0
#define MESSE_LAUFZEIT      1
#define ZAEHLE_ANWEISUNGEN  2


// Es kann wahlweise die Laufzeit bestimmt werden,
// oder die Anzahlen bestimmter Anweisungen gezählt werden.
// Dazu entweder vor dem Kompilieren die entsprechende der drei
// folgenden #define-Anweisungen auskommentieren, oder den Compiler
// mit einer der Optionen -DMODUS=1 (für Laufzeit) oder -DMODUS=2
// (für Anzahl Anweisungen) aufrufen:
//#define MODUS  MESSE_NIX
//#define MODUS  MESSE_LAUFZEIT
//#define MODUS  ZAEHLE_ANWEISUNGEN

#include <cstdio>


// Die Laufzeit wird mit clock() gemessen
#if MODUS == MESSE_LAUFZEIT
#include <ctime>
#endif // if MODUS==MESSE_LAUFZEIT



// Es werden
// - Zuweisungen
// - Additionen/Subtraktionen
// - Mulitiplikationen
// - Divisionen/Modulo
// - Vertauschungen
// - Vergleiche
// gezählt:
#if MODUS == ZAEHLE_ANWEISUNGEN
```

```
int AnzZuw;
int AnzAddSub;
int AnzMul;
int AnzDivMod;
int AnzSwap;
int AnzCmp;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN


//
//       Euklidscher Algorithmus:
//       +-------------------------------------+
//       |                                     |
//       | +-----------------------------------+
//       | | r = m mod n                       |
//       | +-----------------------------------+
//       | |              r = 0 ?              |
//       | |                 |                 |
//       | |----- ja ---------+---- nein --------+
//       | |                 |                 |
//       | | Ergebnis = n     | m = n           |
//       | +-----------------+-----------------+
//       | | <--Euklidsch. A. | n = r           |
//       | +-----------------+-----------------+
//       |                                     |
//       +-------------------------------------+
//


int euclid( int m, int n )
{
  int r;


  while( 1 )
    {
#if MODUS == ZAEHLE_ANWEISUNGEN
      AnzDivMod++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
      r = m % n;

#if MODUS == ZAEHLE_ANWEISUNGEN
      AnzCmp++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
      if( r==0)
        {
          return n;
        }
      else
        {
#if MODUS == ZAEHLE_ANWEISUNGEN
```

```
      AnzZuw++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
          m = n;
#if MODUS == ZAEHLE_ANWEISUNGEN
      AnzZuw++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
          n = r;
        }
    }

  return 0; // eigentlich egal, aber vermeidet Compilerwarnungen

} // int euclid( int m, int n )


//
//       stumpfisttrumpf:
//       +-------------------------------------+
//       |                m>n                  |
//       |                 |                   |
//       +--- ja  --------+--- nein ----------+
//       | Vertausche m,n |                    |
//       +-------------------------------------+
//       |  für i = m ... 1  (-1)              |
//       |   +---------------------------------+
//       |   |  m/i ohne Rest teilbar          |
//       |   |      UND                        |
//       |   |  n/i ohne Rest teilbar          |
//       |   |          |                      |
//       |   +-- nein --+-- ja ----------------+
//       |   |          |                      |
//       |   |          | ggT = i              |
//       |   |          +----------------------+
//       |   |          | <- stumpfisttrumpf   |
//       +---+----------+----------------------+
//
//

int stumpfisttrumpf( int m, int n )
{
#if MODUS == ZAEHLE_ANWEISUNGEN
  AnzCmp++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
  if( m>n )
    {
#if MODUS == ZAEHLE_ANWEISUNGEN
      AnzSwap++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
      int tmp = m;
      m = n;
      n = tmp;
```

```
    }


  for( int i=m; m>=1; i-- )
    {
#if MODUS == ZAEHLE_ANWEISUNGEN
      AnzDivMod++;
      AnzCmp++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
      if( ( m % i ) == 0 )
        {
#if MODUS == ZAEHLE_ANWEISUNGEN
          AnzDivMod++;
          AnzCmp++;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN
          if( ( n % i ) == 0 )
            {
              return i;
            }
        }
    }

  return 0; // eigentlich egal, aber vermeidet Compilerwarnungen

} // int stumpfisttrumpf( int m, int n )

//void kombiniere_m_n( ggt_t ggt,
void kombiniere_m_n( int (*ggt)( int m, int n ),
                     int mvon,
                     int mbis,
                     int nvon,
                     int nbis,
                     int NTIME
#if MODUS == MESSE_LAUFZEIT
                     , FILE *f     // Ausgabe der Laufzeiten
#endif // if MODUS==MESSE_LAUFZEIT
                     )
{

#if MODUS == ZAEHLE_ANWEISUNGEN
  AnzZuw = 0;
  AnzAddSub = 0;
  AnzMul = 0;
  AnzDivMod = 0;
  AnzSwap = 0;
  AnzCmp = 0;
#endif // if MODUS==ZAEHLE_ANWEISUNGEN



#if MODUS == MESSE_LAUFZEIT
  clock_t anf, end;
```

```
  anf = clock();

  for( int i=0; i<NTIME; i++ )
    {
#endif // if MODUS==MESSE_LAUFZEIT



      for( int m=mvon; m<=mbis; m++ )
        {
          for( int n=nvon; n<=nbis; n++ )
            {

              /*int erg =*/ ggt( m, n );

            }
        }


#if MODUS == MESSE_LAUFZEIT
    }

  end = clock();
  printf( "Laufzeit m,n=[%8d,%8d],[%8d,%8d]: %g msec/ggT\n",
          mvon, mbis,
          nvon, nbis,
          ( (double)( end - anf )
            /CLOCKS_PER_SEC
            /NTIME
            /( (nbis-nvon+1)*(mbis-mvon+1) ) // Anzahl ggT
            *1000    // Umrechnung von sec auf msec
            )
          );
  fprintf( f,
           "%10d %g\n",
           mvon,
           ( (double)( end - anf )
             /CLOCKS_PER_SEC
             /NTIME
             /( (nbis-nvon+1)*(mbis-mvon+1) ) // Anzahl ggT
             *1000    // Umrechnung von sec auf msec
             )
           );
#endif // if MODUS==MESSE_LAUFZEIT


// Es werden
// - Zuweisungen
// - Additionen/Subtraktionen
// - Multiplikationen
```

```
// - Divisionen/Modulo
// - Vertauschungen
// gezählt:
#if MODUS == ZAEHLE_ANWEISUNGEN
  printf( "Anzahl Anweisungen für m,n=[%8d,%8d],[%8d,%8d]:\n",
          mvon, mbis,
          nvon, nbis
          );
  printf( "AnzZuw    = %9.2f\n",
          (double)AnzZuw/( (nbis-nvon+1)*(mbis-mvon+1) )
          );
  printf( "AnzAddSub = %9.2f\n",
          (double)AnzAddSub/( (nbis-nvon+1)*(mbis-mvon+1) )
          );
  printf( "AnzMul    = %9.2f\n",
          (double)AnzMul/( (nbis-nvon+1)*(mbis-mvon+1) )
          );
  printf( "AnzDivMod = %9.2f\n",
          (double)AnzDivMod/( (nbis-nvon+1)*(mbis-mvon+1) )
          );
  printf( "AnzSwap   = %9.2f\n",
          (double)AnzSwap/( (nbis-nvon+1)*(mbis-mvon+1) )
          );
  printf( "AnzCmp    = %9.2f\n",
          (double)AnzCmp/( (nbis-nvon+1)*(mbis-mvon+1) )
          );
  printf( "\n" );
#endif // if MODUS==ZAEHLE_ANWEISUNGEN

}


int main( int nargs, char **args )
{
#if MODUS == MESSE_LAUFZEIT
  FILE *f_euklid = fopen( "laufzeit_ggt_euklid_delta_ueber_n.xy", "w" );
  FILE *f_einfach = fopen( "laufzeit_ggt_einfach_delta_ueber_n.xy", "w" );
  if( !f_euklid || !f_einfach )
  {
    fprintf( stderr, "kann Dateien nicht öffnen!\n" );
    return 1;
  }
#endif // if MODUS==MESSE_LAUFZEIT

  printf( "\nVersion Euklid\n" );
  kombiniere_m_n( euclid,
                  1, 20, 1, 20,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
```

```
                  );
  kombiniere_m_n( euclid,
                  1001, 1020, 1001, 1020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
                  );
#if MODUS == MESSE_LAUFZEIT
  kombiniere_m_n( euclid,
                  5001, 5020, 5001, 5020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( euclid,
                  10001, 10020, 10001, 10020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( euclid,
                  50001, 50020, 50001, 50020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( euclid,
                  100001, 100020, 100001, 100020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( euclid,
                  500001, 500020, 500001, 500020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
                  );
#endif // if MODUS==MESSE_LAUFZEIT
  kombiniere_m_n( euclid,
                  1000001, 1000020, 1000001, 1000020,
                  100000
#if MODUS == MESSE_LAUFZEIT
                  , f_euklid
#endif // if MODUS==MESSE_LAUFZEIT
```

```
                  );


  printf( "\nVersion einfacher Algorithmus\n" );
  kombiniere_m_n( stumpfisttrumpf,
                  1, 20, 1, 20,
                  5000
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( stumpfisttrumpf,
                  1001, 1020, 1001, 1020,
                  500
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
#if MODUS == MESSE_LAUFZEIT
  kombiniere_m_n( stumpfisttrumpf,
                  5001, 5020, 5001, 5020,
                  100
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( stumpfisttrumpf,
                  10001, 10020, 10001, 10020,
                  10
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( stumpfisttrumpf,
                  50001, 50020, 50001, 50020,
                  1
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( stumpfisttrumpf,
                  100001, 100020, 100001, 100020,
                  1
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
  kombiniere_m_n( stumpfisttrumpf,
                  500001, 500020, 500001, 500020,
                  1
#if MODUS == MESSE_LAUFZEIT
```

```
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );
#endif // if MODUS==MESSE_LAUFZEIT
  kombiniere_m_n( stumpfisttrumpf,
                  1000001, 1000020, 1000001, 1000020,
                  1
#if MODUS == MESSE_LAUFZEIT
                  , f_einfach
#endif // if MODUS==MESSE_LAUFZEIT
                  );



  return 0;
} // main( int nargs, char **args )
```

Ausgabe der Laufzeiten:

```
Version Euklid
Laufzeit m,n=[       1,      20],[       1,      20]: 0.00014575 msec/ggT
Laufzeit m,n=[    1001,    1020],[    1001,    1020]: 0.000217 msec/ggT
Laufzeit m,n=[    5001,    5020],[    5001,    5020]: 0.000218 msec/ggT
Laufzeit m,n=[   10001,   10020],[   10001,   10020]: 0.000223 msec/ggT
Laufzeit m,n=[   50001,   50020],[   50001,   50020]: 0.00022025 msec/ggT
Laufzeit m,n=[  100001,  100020],[  100001,  100020]: 0.00021725 msec/ggT
Laufzeit m,n=[  500001,  500020],[  500001,  500020]: 0.0002185 msec/ggT
Laufzeit m,n=[ 1000001, 1000020],[ 1000001, 1000020]: 0.00022575 msec/ggT

Version einfacher Algorithmus
Laufzeit m,n=[       1,      20],[       1,      20]: 0.000305 msec/ggT
Laufzeit m,n=[    1001,    1020],[    1001,    1020]: 0.02595 msec/ggT
Laufzeit m,n=[    5001,    5020],[    5001,    5020]: 0.12775 msec/ggT
Laufzeit m,n=[   10001,   10020],[   10001,   10020]: 0.2525 msec/ggT
Laufzeit m,n=[   50001,   50020],[   50001,   50020]: 1.3 msec/ggT
Laufzeit m,n=[  100001,  100020],[  100001,  100020]: 2.525 msec/ggT
Laufzeit m,n=[  500001,  500020],[  500001,  500020]: 12.65 msec/ggT
Laufzeit m,n=[ 1000001, 1000020],[ 1000001, 1000020]: 25.3 msec/ggT
```

Ausgabe der Anzahlen der Anweisungen, nach Gruppen getrennt:

```
Version Euklid
Anzahl Anweisungen für m,n=[       1,      20],[       1,      20]:
AnzZuw    =      3.49
```

```
AnzAddSub =      0.00
AnzMul    =      0.00
AnzDivMod =      2.75
AnzSwap   =      0.00
AnzCmp    =      2.75

Anzahl Anweisungen für m,n=[    1001,    1020],[    1001,    1020]:
AnzZuw    =      5.54
AnzAddSub =      0.00
AnzMul    =      0.00
AnzDivMod =      3.77
AnzSwap   =      0.00
AnzCmp    =      3.77

Anzahl Anweisungen für m,n=[ 1000001, 1000020],[ 1000001, 1000020]:
AnzZuw    =      5.65
AnzAddSub =      0.00
AnzMul    =      0.00
AnzDivMod =      3.83
AnzSwap   =      0.00
AnzCmp    =      3.83


Version einfacher Algorithmus
Anzahl Anweisungen für m,n=[       1,      20],[       1,      20]:
AnzZuw    =      0.00
AnzAddSub =      0.00
AnzMul    =      0.00
AnzDivMod =      8.24
AnzSwap   =      0.47
AnzCmp    =      9.24

Anzahl Anweisungen für m,n=[    1001,    1020],[    1001,    1020]:
AnzZuw    =      0.00
AnzAddSub =      0.00
AnzMul    =      0.00
AnzDivMod =    962.33
AnzSwap   =      0.47
AnzCmp    =    963.33

Anzahl Anweisungen für m,n=[ 1000001, 1000020],[ 1000001, 1000020]:
AnzZuw    =      0.00
AnzAddSub =      0.00
AnzMul    =      0.00
AnzDivMod = 950018.44
AnzSwap   =      0.47
AnzCmp    = 950019.44
```

A.2.1.3 Kommentar zu den Ergebnissen

Aus den Ergebnissen kann man leicht folgende grundlegenden Erkenntnisse ziehen, die bei vielen Aufgabenstellungen immer wiederkehren:

- Die Laufzeit verschiedener Algorithmen kann sich ganz wesentlich unterscheiden, und zwar insbesondere bei Zunahme der Problemgröße.

  In den meisten praktischen Fällen ist die genaue Laufzeit zweitrangig. Vielmehr ist besonders wichtig, daß die Laufzeit bei hohen Problemgrößen möglichst wenig zunimmt (falls der Algorithmus auf große Probleme angewendet werden soll zumindest).

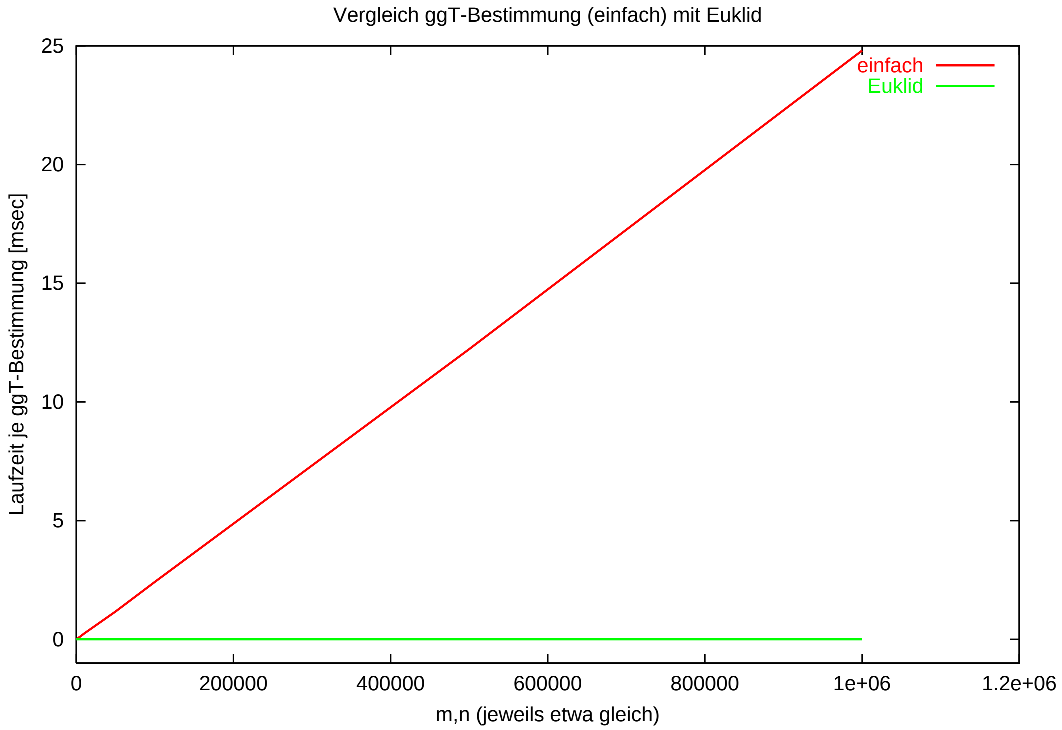


Abbildung A.2: Laufzeit der C++-Version über der Problemgröße

  Bereits ein lineares Verhalten kann in der Praxis kritisch sein. Auf jeden Fall sollte eine überlineare Zunahme der Laufzeit ($O(n^k)$, mit $k > 1$) oder gar ein exponentielles Verhalten unbedingt vermieden werden (soweit möglich).

- Die Wahl des Umgebungssystems geht ganz wesentlich in die Laufzeit ein. Neben der Hardware und der Güte des Compilers (wie gut kann er automatisch optimieren?) ist auch wesentlich, ob das Programm nur interpretiert wird (wie bei klassischen Javasystemen), oder ob echter Maschinencode erzeugt wird. In dem obigen Beispiel läuft das (interpretierte) Javaprogramm etwa dreimal solange wie das (kompilierte) C++-Programm.

A.2.2 Fakultät

A.2.2.1 Javaversion

Der Quelltext für die Javaversion ist folgender:

```
// Time-stamp: "29.03.06 15:46 fakultaet.java klaus@wachtler.de"
//
// Musterlösung der Aufgabe "Fakultät".
// Es wird die Fakultät einer Zahl jeweils iterativ und rekursiv
// berechnet und die dafür benötigten Laufzeiten ausgegeben.
//
// javac fakultaet.java && java fakultaet

import java.util.*;

public class fakultaet
{
    // Berechnet iterativ die Fakultät des übergebenen Parameters
    public static long fakultaet_iterativ( int n )
    {
        long produkt = 1;
        for( int i=2; i<=n; ++i )
        {
            produkt *= i;
        }
        return produkt;
    }

    // Berechnet rekursiv die Fakultät des übergebenen Parameters
    public static long fakultaet_rekursiv( int n )
    {
        return n<=1 ? 1 : n*fakultaet_rekursiv( n-1 );
    }

    public static void main( String args[] )
    {
        final int  anzahl = 1000000; // Soviele Durchläufem um die
                                     // Auflösung der internen Uhr zu
                                     // erhöhen
        long    start, ende;         // der Zeitmessung
```

```
        start = System.currentTimeMillis();
        for( int i=0; i<anzahl; ++i )
        {
            fakultaet_rekursiv( 15 );
        }
        ende = System.currentTimeMillis();
        System.out.println( "rekursiv:   "
                            +
                            ( (double)ende  -start )/anzahl
                            +
                            " msec"
                            );

        start = System.currentTimeMillis();
        for( int i=0; i<anzahl; ++i )
        {
            fakultaet_iterativ( 15 );
        }
        ende = System.currentTimeMillis();
        System.out.println( "iterativ:   "
                            +
                            ( (double)ende  -start )/anzahl
                            +
                            " msec"
                            );


        //System.out.println( "" + n + "! = " + fakultaet_rekursiv( n ) );
        //System.out.println( "" + n + "! = " + fakultaet_iterativ( n ) );
    }

} // public class fakultaet
```

Das Programm erzeugt auf meinem Rechner folgende Ausgabe:

```
rekursiv:   0.009647 msec
iterativ:   0.001996 msec
```

A.2.2.2 C-Version

Der Quelltext für die C-Version ist folgender:

```
/* Time-stamp: "29.03.06 15:58 fakultaet.c klaus@wachtler.de"
 *
 *
 * Musterlösung der Aufgabe "Fakultät"
 *
 */

#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <time.h>


// Berechnet iterativ die Fakultät des übergebenen Parameters
long fakultaet_iterativ( int n )
{
  long produkt = 1;
  int  i;
  for( i=2; i<=n; ++i )
  {
    produkt *= i;
  }
  return produkt;
}

// Berechnet rekursiv die Fakultät des übergebenen Parameters
long fakultaet_rekursiv( int n )
{
  return n<=1 ? 1 : n*fakultaet_rekursiv( n-1 );
}

int main( int nargs, char **args )
{
  const int  anzahl = 10000000; // Soviele Durchläufem um die
                                // Auflösung der internen Uhr zu
                                // erhöhen
  clock_t start, ende;          // der Zeitmessung
  int     i;
```

```
  start = clock();
  for( i=0; i<anzahl; ++i )
  {
    fakultaet_rekursiv( 15 );
  }
  ende = clock();
  printf( "rekursiv:  %f msec\n",
          ( (double)ende  -start )/CLOCKS_PER_SEC/anzahl*1000
          );

  start = clock();
  for( i=0; i<anzahl; ++i )
  {
    fakultaet_iterativ( 15 );
  }
  ende = clock();
  printf( "rekursiv:  %f msec\n",
          ( (double)ende  -start )/CLOCKS_PER_SEC/anzahl*1000
          );

  return 0;
} /* main( int nargs, char **args ) */
```

Das Programm erzeugt auf meinem Rechner folgende Ausgabe:

```
rekursiv:  0.000408 msec
rekursiv:  0.000269 msec
```

A.2.3 Summenbildung

kommt noch...

A.2.4 Binäre Suche

Der Fehler im angegebenen Algorithmus besteht darin, daß nach dem Prüfen von `a[mid]` gegen das gesuchte das Feld auf eines der Teilfelder inklusive dem bereits geprüften Element an der Stelle `mid` geteilt wird (also entweder auf `0 .. mid` oder `mid .. max`).

Das bereits geprüfte Element darf aber nicht im Feld verbleiben (man müßte also reduzieren auf `0 .. mid-1` oder `mid+1 .. max`): erstens ist es uneffektiv, das Feld nicht so schnell wie möglich zu reduzieren, zweitens kann es je nach Eingangsdaten zu einer Endlosschleife kommen (nämlich in dem Fall, daß ein gesuchtes Element nicht im Feld enthalten ist, aber aufgrund der Sortierung zwischen den ersten beiden Elementen des aktuellen Teilfeldes liegen müßte). Dann wird von einem Schritt zum nächsten das Feld nicht mehr reduziert.[1]

A.2.4.1 Java-Version

Hier die Musterlösung in Java (in Form einer einzigen Klasse; der Übersichtlichkeit zuliebe alles in einer Klasse und nur für ganze Zahlen):

```
// Time-stamp: "03.05.03 18:20 binsuch.java klaus@wachtler.de"
//
// Testprogramm zur Binären Suche.
// javac binsuch.java && java binsuch

import java.util.*;
import java.io.*;

public class binsuch
{
    // die eigentliche Suche muß sowohl mit dem ursprünglichen gesamten
    // Feld, als auch mit Teilfeldern durchgeführt werden können.
    // Damit die Teilfelder nicht durch aufwendiges Kopieren erzeugt
    // werden müssen, kann man einfach immer das gesamte Feld an den
    // nächsten Rekursionsschritt übergeben, und als zusätzliche
    // Parameter den Anfangs- und den Endindex des zu verwendenden
    // Teilfeldes.
    // Um den ersten Aufruf einfacher und fehlerfrei zu ermöglichen,
    // wird die Methode binsuch() mit zwei Versionen überlagert:
    // Die erste Version wird nur mit dem Feld un dem zu suchenden
```

[1] In älteren Versionen dieses Skripts -bis Anfang 2003- war noch ein weiterer Fehler im angegebenen Algorithmus, und zwar waren die beiden Zweige der Fallunterscheidung zum Reduzieren auf das linke oder rechte Teilfeld vertauscht. Dies war ein Fehler meinerseits, der in der ursprünglichen Version in [v.Helden] nicht enthalten war.

```
// Wert aufgerufen, die zweite mit den Grenzen für ein Teilfeld.

private static Integer binsuch( Integer Gesamtfeld[],
                                int zusuchen
                                )
{
    // einfach die Parameter auf die eigentliche Suche umsetzen:
    return binsuch( Gesamtfeld, 0, Gesamtfeld.length-1, zusuchen );
}

private static Integer binsuch( Integer Teilfeld[], int von, int bis,
                                int zusuchen
                                )
{
    if( bis<von )
    {
        // leeres Feld? Nichts gefunden!
        return null;
    }
    else if( bis==von )
    {
        // nur ein Element; vergleichen und ggf. zurückgeben:
        return ( (Teilfeld[bis]).intValue()==zusuchen ? Teilfeld[bis] : null );
    }
    else
    {
        // Mehr als ein Element
        // Mitte suchen, und ggf. links oder rechts suchen:
        int index_Mitte = ( von + bis )/2;
        if( Teilfeld[index_Mitte].intValue()==zusuchen )
        {
            // Zufällig getroffen!
            return Teilfeld[index_Mitte];
        }
        else if( Teilfeld[index_Mitte].intValue()>zusuchen )
        {
            // Mitte ist zu groß, also links weitersuchen
            return binsuch( Teilfeld, von, index_Mitte-1, zusuchen );
        }
        else
        {
            // Mitte zu klein, also in rechter Hälfte suchen:
            return binsuch( Teilfeld, index_Mitte+1, bis, zusuchen );
        }
    }
}



private static void einfacherTest()
{
```

```
    // Um Objekte zu haben, nehmen wir nicht int, sondern
    // Integer. Dann kann man als Ergebnis der Suche den Wert null
    // liefern, wenn die Suche erfolglos ist.
    Integer a1[] = new Integer[7];

    a1[0] = new  Integer(12);
    a1[1] = new  Integer(45);
    a1[2] = new  Integer(60);
    a1[3] = new  Integer(70);
    a1[4] = new  Integer(80);
    a1[5] = new Integer(100);
    a1[6] = new Integer(110);

    Integer  gefunden = binsuch( a1, 105 );
    if( gefunden!=null )
    {
        System.out.println( "In a1[] gefunden: " + gefunden );
    }
    else
    {
        System.out.println( "Wert in a1[] nicht gefunden" );
    }
}

private static void TestMitEinlesen()
{

    // Feld für die gelesenen und zu durchsuchenden Werte:
    Integer   a2[] = null;

    try
    {
        // zu lesende Datei ist ein BufferedReader:
        BufferedReader  in
            = new BufferedReader( new FileReader( "werte.dat" ) );

        String zeile;   // hier kommt je eine gelesene Zeile hin

        while( (zeile=in.readLine())!=null )  // Schleife über
                                              // alle Zeile der Datei
        {
            try
            {
                // versuchen, die Zeile in eine int zu verwandeln
                // und speichern (dazu das Feld verlängern):
                Integer neuesFeld[] = new Integer[a2==null ? 1 : a2.length+1];
                for( int i=0; i<neuesFeld.length-1; i++ )
                {
                    neuesFeld[i] = a2[i];
                }
                neuesFeld[neuesFeld.length-1] = new Integer( Integer.parseInt( zeile ) );
```

```
                a2 = neuesFeld;
            }
            catch( Exception e )
            {
                System.out.println( "Konvertieren zu int geht nicht" );
            }

        }

        in.close();   // Datei wieder schließen
    }
    catch( Exception e )
    {
    }

    Integer  gefunden = binsuch( a2, 24 );
    if( gefunden!=null )
    {
        System.out.println( "In a2[] gefunden: " + gefunden );
    }
    else
    {
        System.out.println( "Wert in a2[] nicht gefunden" );
    }


}

public static void main( String args[] )
{
    einfacherTest();
    TestMitEinlesen();
}


} // public class binsuch
```

A.2.4.2 C/C++-Version

Hier die Musterlösung in C++:

```
// Time-stamp: "03.05.03 17:39 binsuch.cpp klaus@wachtler.de"
//
// Musterloesung in C++ zu folgender Aufgabenstellung:
// Das Suchverfahren "‘Binäres Suchen"’ arbeitet nach folgendem Algorithmus:
//
// Algorithmus:  Binäre Suche im Array a[min .....max]  (min = 0,  max = n)
//  (1)  wähle ein mittleres Element des Arrays a[mid]
```

```
//  (2)  if (a[mid].key == gesuchterKey)
//           Suche erfolgreich
//       else
//           if (a[mid].key  < gesuchterKey)
//                betrachte die linke Hälfte des Arrays, also 0 .. mid
//           else
//                betrachte die rechte Hälfte des Arrays, also mid.. max
//  (3)  if (betrachtetes Array  == leer)
//          Suche erfolglos
//       else
//          beginne bei Schritt 1
//
// Implementieren Sie den Algorithmus so, daß in einem sortierten Feld
// von Integer-Zahlen gesucht werden kann. Geben Sie jedesmal, wenn Sie
// ein Teilfeld durchsuchen, die Indizes der Grenzen und des mittleren
// Elements aus. Geben Sie am Ende den Index des gefundenen Elements
// bzw. eine Meldung für die erfolglose Suche aus.
//
// Testen Sie Ihren Algorithmus zunächst mit einfachen Feldern, die Sie
// direkt im Programmcode initialisieren.
//
// Erweitern Sie ihr Programm so, daß die zu durchsuchende Zahlenfolge
// aus einer Datei eingelesen werden kann. Hierzu sollen Sie folgende
// Ein-/Ausgabe realisieren:
//
//  -  Lesen Sie den Namen der Datei, in der die zu durchsuchende
//     Zahlenfolge steht von der Tastatur ein.
//  -  Lesen Sie die Zahlenfolge aus der Datei ein.
//  -  Das Programm soll selbständig herausfinden, wieviele
//     Elemente die Zahlenfolge enthält und das Feld entsprechend
//     dimensionieren.
//  -  Gehen Sie davon aus, daß die Elemente in der
//     einzulesenden Datei jeweils durch Leerzeichen getrennt sind.
//  -  Lesen Sie die zu suchende Zahl von der Tastatur ein.
//  -  Fangen Sie Fehler beim Einlesen ab und überprüfen Sie,
//     ob die eingelesene Zahlenfolge sortiert ist.
//
// Geben Sie zum Schluß den Algorithmus richtig an (im Gegensatz zum oben
// gegebenen).
//
// ANSI-C++
//
// Linux:    g++ -Wall binsuch.cpp && a.out
//
// VC++ 6.0: cl /Zi /GX auklid.cpp
//           .\binsuch

#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

```
int *binsuch( int feld[], size_t l, int zusuchen )
{
  if( l<1 )
  {
    // leeres Feld? Nichts gefunden!
    return NULL;
  }
  else if( l==1 )
  {
    // nur ein Element; vergleichen und ggf. zurückgeben:
    return ( feld[0]==zusuchen ? &feld[0] : NULL );
  }
  else
  {
    // Mehr als ein Element
    // Mitte suchen, und ggf. links oder rechts suchen:
    int index_Mitte = l/2;
    if( feld[index_Mitte]==zusuchen )
    {
      // Zufällig getroffen!
      return &feld[index_Mitte];
    }
    else if( feld[index_Mitte]>zusuchen )
    {
      // Mitte ist zu groß, also links weitersuchen
      return binsuch( feld, index_Mitte, zusuchen );
    }
    else
    {
      // Mitte zu klein, also in rechter Hälfte suchen:
      return binsuch( &feld[index_Mitte+1], l - index_Mitte - 1, zusuchen );
    }
  }

} // int *binsuch( int feld[], size_t l, int zusuchen )


void einfacherTest()
{
  // einfaches, sortiertes Feld zum Testen:
  int   a1[] = { 12, 45, 60, 70, 80, 100, 110 };
  const size_t a1l = sizeof(a1)/sizeof(a1[0]); // Feldlänge

  int *p_gefunden = binsuch( a1, a1l, 46 );
  if( p_gefunden )
  {
    cout << "In a1[] gefunden: " << *p_gefunden << endl;
  }
  else
  {
```

```
    cout << "Wert in a1[] nicht gefunden." << endl;
  }
} // void einfacherTest()


void TestMitEinlesen()
{
  // Datei mit den sortierten Werten:
  string Dateiname;
  cout << "Bitte den Namen einer Datei mit sortierten int-Werten:" << endl;
  cin >> Dateiname;

  int *arr = NULL;
  size_t larr = 0;

  { // Einlesen
    ifstream f( Dateiname.c_str() );
    int wert;
    while( ( f>>wert ) )
    {
      // Feld um ein Element verlängern:
      arr = (int*)realloc( arr, ++larr*sizeof(int) );
      if( !arr )
      {
        cerr << "kein Speicher!\n";
        exit( 1 );
      }
      arr[larr-1] = wert;
    }
  } // Einlesen

  { // Testen, ob sortiert
    for( size_t i=1; i<larr; i++ )
    {
      if( !( arr[i-1]<arr[i] ) )
      {
        cerr << "Feld nicht sortiert!\n";
        exit( 1 );
      }
    }
  } // Testen, ob sortiert


  // Welcher Wert soll gesucht werden?
  cout << "Gesuchten Wert eingeben:" << endl;
  int gesucht;
  cin >> gesucht;

  int *p_gefunden = binsuch( arr, larr, gesucht );
  if( p_gefunden )
  {
    cout << "In " << Dateiname << " gefunden: " << *p_gefunden << endl;
```

```
  }
  else
  {
    cout << "Wert in " << Dateiname << " nicht gefunden." << endl;
  }

  // allokierten Speicher wieder freigeben:
  free( arr ); arr = NULL; larr = 0;

} // void TestMitEinlesen()


int main( int nargs, char **args )
{
  einfacherTest();
  TestMitEinlesen();

  return 0;
} // main( int nargs, char **args )
```

A.2.4.3 C/C++-Version (verallgemeinert)

In der Standardbibliothek von C ist bereits eine Funktion `bsearch()` enthalten, die in Feldern beliebigen Typs suchen kann. Dazu ist beim Aufruf eine Funktion (beziehungsweise ihre Adresse) anzugeben, die anhand von zwei Zeigern auf Feldelemente zurückliefert, welcher der beiden Werte kleiner ist (siehe `man 3 bsearch` unter Unix/Linux oder [KW-C]).

Eine mögliche Implementation mit zugehörigem Testprogramm sieht so aus:

```
// Time-stamp: "24.05.02 11:30 bsearch.cpp Administrator@AW41"
//
// ANSI-C
//
// Linux:    g++ -Wall bsearch.cpp && a.out
// VC++ 6.0: cl

#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

// sucht in einem sortierten Feld (*base)[] nach einem
// Element, für das die Funktion (*compar)() Gleichheit mit dem
// Schlüssel (*key) verspricht, und liefert einen Zeiger auf
// das gefundene Element, oder NULL
```

```
// (entspricht dem Original-bsearch() nach ANSI-C)
void *bsearch( const void *key,
               const void *base,
               size_t nmemb,
               size_t size,
               int (*compar)( const void *, const void * )
               )
{
  switch( nmemb )
  {
    case 0:   // leeres Feld
      return NULL;

    case 1:   //  ein Element: vielleicht ein Treffer
      return ( (*compar)( base, key )==0 ? (void *)base : NULL );

    default : // mehr als zwei Elemente

      // Teilen, und suchen lassen:
      {
        // Etwa die Mitte des Feldes:
        size_t n1 = nmemb/2;
        // Zeiger auf mittleres Element:
        const void  *p1 = (const char*)base+(n1*size);
        // Dieses mittlere Element mit Schlüssel vergleichen:
        int    v1 = (*compar)( p1, key );
        if( v1==0 )
        {
          // Schlüssel genau getroffen
          return(void *) p1;
        }
        else if( v1<0 )
        {
          // base[n1] kleiner als Schlüssel;
          // also in der oberen Hälfte suchen.
          // base[n1] braucht schon nicht mehr betrachtet
          // zu werden.
          return bsearch( key,
                          (const char*)p1+size,
                          nmemb - n1 - 1,
                          size,
                          compar
                          );
        }
        else
        {
          // base[n1] größer als Schlüssel;
          // also in der unteren Hälfte suchen.
          // base[n1] braucht schon nicht mehr betrachtet
          // zu werden.
          return bsearch( key,
```

```
                          base,
                          n1,
                          size,
                          compar
                          );
        }
      }

  } // switch

} // bsearch()

// Testdatentyp für Feldelemente:
typedef struct
{
  double d;
  char   s[100];
} s_t;

// Vergleichsfunktion für obigen Datentyp s_t:
int vergl_f( const void *p1, const void *p2 )
{
  if( ((const s_t*)p1)->d > ((const s_t*)p2)->d )
  {
    return 1;
  }
  else if( ((const s_t*)p1)->d < ((const s_t*)p2)->d )
  {
    return -1;
  }
  else
  {
    return 0;
  }
}

// zu durchsuchendes Feld:
s_t f[] =
{
  { 3.141592654, "pi" },
  { 2.718281828, "e" },
  { 53.00000000, "w" },
  { 1.000000000, "eins" },
  { 2.000000000, "zwoo" },
  { 13.00000000, "drölf" },
};

// Anzahl Feldelemente:
const int l = sizeof(f)/sizeof(f[0]);

// Testprogramm:
```

```
int main( int nargs, char **args )
{
  qsort( f,
         l,
         sizeof(f[0]),
         vergl_f
         );

  // Schleife, bis der Benutzer -1 eingibt:
  while( 1 )
  {
    double wert;
    printf( "Bitte einen Wert (Ende mit -1): " );
    if( scanf( "%lf", &wert )!=1 || wert==-1.0 )
    {
      break;
    }

    s_t key = { wert, "" };
    s_t *p = (s_t *)bsearch( &key,
                             f,
                             l,
                             sizeof(f[0]),
                             vergl_f
                             );

    if( p )
    {
      printf( "Gefunden: %f %s\n", p->d, p->s );
    }
    else
    {
      printf( "nichts gefunden!\n" );
    }


  }


  return 0;
} /* main( int nargs, char **args ) */
```

A.2.5 Einfach verkettete Liste

A.2.5.1 Java-Version

Eine einfache Liste für ganzzahlige Werte mit den Anforderungen aus der Aufgabenstellung (siehe **A.1.5 Einfach verkettete Liste** auf Seite 96) könnte so aussehen:

```
// Time-stamp: "11.05.03 13:38 einfacheListe.java klaus@wachtler.de"
//
// Lösung der Übungsaufgabe: Erstellen einer einfach verketteten Liste

// Die eigentliche Liste besteht aus einer Referenz auf das erste Element.
public class einfacheListe
{
    // Datentyp für ein Element der Liste:
    private class ListenElement
    {
        private int           nutzdaten;  // die Daten eines Elements
        private ListenElement next;       // Verweis auf Nachfolger

        public ListenElement( int daten )
        {
            nutzdaten = daten;
            next = null;
        }

        // fügt hinter sich ein neues Element ein:
        public void einf_dahinter( int wert )
        {
            // dazu den bisherigen Nachfolger merken, um
            // ihn später als Nach-Nachfolger einsetzen zu können:
            ListenElement uebernaechster = next;

            // neues Element schaffen und anhängen:
            next = new ListenElement( wert );

            // bisherigen Nachfolger dahinter setzen:
            next.setNext( uebernaechster );
        }

        public void setDaten( int daten )
        {
            nutzdaten = daten;
        }

        public int getDaten()
        {
```

```
        return nutzdaten;
    }

    public void setNext( ListenElement next_neu )
    {
        next = next_neu;
    }

    public ListenElement getNext()
    {
        return next;
    }


    // sucht nach einem Wert, und liefert das ListenElement
    // (falls gefunden), oder null.
    public ListenElement sucheWert( int wert )
    {
        // Schleife notfalls über alle Elemente:
        ListenElement aktuellesElement = this;
        do
        {
            if( aktuellesElement.getDaten()==wert )
            {
                return aktuellesElement;
            }
            aktuellesElement = aktuellesElement.getNext();
        }
        while( aktuellesElement.getNext()!=null );

        // wenn alle Elemente nicht paßten, dann ist die Suche
        // fehlgeschlagen:
        return null;
    }

    // sucht nach dem Vorgänger eines Werts, und liefert das
    // ListenElement (falls gefunden), oder null.
    public ListenElement sucheVorgaengerVonWert( int wert )
    {
        // Schleife notfalls über alle Elemente:
        ListenElement aktuellesElement = this;
        do
        {
            if( aktuellesElement.next!=null
                &&
                aktuellesElement.next.getDaten()==wert
                )
            {
                return aktuellesElement;
            }
            aktuellesElement = aktuellesElement.getNext();
```

```
        }
        while( aktuellesElement.getNext()!=null );

        // wenn alle Elemente nicht paßten, dann ist die Suche
        // fehlgeschlagen:
        return null;
    }

    // Ausgabe einer Liste in einen String:
    // Element ausgeben, dann ggf. den Rest
    public String toString()
    {
        return ( " El." + nutzdaten
                 + ( next==null ? "." : ", " + next.toString() )
                 );
    }

} // class ListenElement


private ListenElement   anfang;

public einfacheListe()
{
    anfang = null;
}

// fügt einen neuen Wert am Anfang ein:
public void einf_vorn( int wert )
{
    // das bisherige erste Element merken, um es später zum
    // zweiten zu machen (auch wenn das bisherige erste
    // null ist, also die Liste bisher leer ist):
    ListenElement  zweites = anfang;

    // ein neues erstes Element erzeugen, und mit wert
    // initialisieren:
    anfang = new ListenElement( wert );

    // das bisherige erste Element (oder null) hinten
    // anhängen:
    anfang.setNext( zweites );
}

// fügt einen neuen Wert am Ende ein:
public void einf_hinten( int wert )
{
    // Wenn die Liste noch leer ist, dann einfach ein
    // Element an den Anfang setzen:
    if( anfang==null )
    {
```

```
            anfang = new ListenElement( wert );
        }
        else
        {
            // Suche nach dem letzten Element:
            ListenElement letztes = anfang;
            while( letztes.getNext()!=null )
            {
                letztes = letztes.getNext();
            }
            // letztes hat jetzt keinen Nachfolger mehr.
            // Das wird geändert:
            letztes.einf_dahinter( wert );
        }
    }


    // versucht, ein neues Element wert hinter einem gegebenen Wert
    // hinterwas einzusetzen.
    // Das kann natürlich fehlschlagen, wenn das Element, hinter
    // dem eingefügt werden soll, gar nicht existiert.
    // Dementsprechend wird true zurückgegeben, wenn alles geklappt
    // hat; sonst false.
    public boolean einf_HinterWert( int hinterwas, int wert )
    {
        // Leere Liste?
        if( anfang==null )
        {
            return false;
        }
        else
        {
            ListenElement element = anfang.sucheWert( hinterwas );
            if( element==null )
            {
                return false;
            }
            else
            {
                element.einf_dahinter( wert );
                return true;
            }
        }

    }


    // versucht, ein neues Element wert vor einem gegebenen Wert
    // vorwas einzusetzen.
    // Das kann natürlich fehlschlagen, wenn das Element, vor
    // dem eingefügt werden soll, gar nicht existiert.
    // Dementsprechend wird true zurückgegeben, wenn alles geklappt
```

```
// hat; sonst false.
public boolean einf_VorWert( int vorwas, int wert )
{
    // Leere Liste?
    if( anfang==null )
    {
        return false;
    }
    // dann könnte es natürlich sein, daß gleich vor das erste
    // Element eingefügt werden soll:
    else if( anfang.getDaten()==vorwas )
    {
        einf_vorn( wert );
        return true;
    }
    else
    {
        ListenElement element = anfang.sucheVorgaengerVonWert( vorwas );
        if( element==null )
        {
            return false;
        }
        else
        {
            element.einf_dahinter( wert );
            return true;
        }
    }

}


// sucht nach einem Wert, und liefert true, wenn der gesuchte Wert
// enthalten ist:
public boolean sucheWert( int wert )
{
    // Leere Liste?
    if( anfang==null )
    {
        return false;
    }
    else
    {
        return anfang.sucheWert( wert )!=null;
    }

}

// versucht, ein Element wert aus der Liste zu löschen.
// Das kann natürlich fehlschlagen, wenn das Element, das
// gelöscht werden soll, gar nicht existiert.
```

```
    // Dementsprechend wird true zurückgegeben, wenn alles geklappt
    // hat; sonst false.
    public boolean entf_wert( int wert )
    {
        // Leere Liste?
        if( anfang==null )
        {
            return false;
        }
        // dann könnte es natürlich sein, daß gleich das erste
        // Element gelöscht werden soll:
        else if( anfang.getDaten()==wert )
        {
            anfang = anfang.getNext();
            return true;
        }
        else
        {
            // Ansonsten brauchen wir den Vorgänger des zu löschenden
            // Elements; dessen Nachfolger muß gelöscht werden:
            ListenElement element = anfang.sucheVorgaengerVonWert( wert );
            if( element==null )
            {
                return false;
            }
            else
            {
                element.setNext( element.getNext().getNext() );
                return true;
            }
        }

    }


    // Ausgabe einer Liste in einen String:
    // Element ausgeben, dann den Rest
    public String toString()
    {
        return "Liste: " + ( anfang==null ? "-" : anfang.toString() );
    }

} // public class einfacheListe
```

Und hier das zugehörige Testprogramm:

```
// Time-stamp: "(17.05.02 11:53) teste_einfacheListe.java [Klaus@Wachtler.de]"
//
// javac teste_einfacheListe.java einfacheListe.java
```

```
// java teste_einfacheListe
//
// Testprogramm für class einfacheListe

import java.util.*;

public class teste_einfacheListe
{
    public static void main( String args[] )
    {
        einfacheListe   l = new einfacheListe();

        // ein paar jeweils von vorne her einfügen:
        l.einf_vorn( 45 );
        l.einf_vorn( 34 );

        // ein paar von hinten:
        l.einf_hinten( 112 );
        l.einf_hinten( 123 );
        System.out.println( l.toString() );

        // ist der Wert 45 enthalten?
        System.out.println( "45 ist "
                            + ( l.sucheWert( 45 ) ? "" : "nicht " )
                            + "enthalten"
                            );

        // ist der Wert 123 enthalten?
        System.out.println( "123 ist "
                            + ( l.sucheWert( 123 ) ? "" : "nicht " )
                            + "enthalten"
                            );

        // Die (nicht existierende) 30 löschen:
        if( l.entf_wert( 30 ) )
        {
            System.out.println( l.toString() );
        }
        else
        {
            System.out.println( "die 30 kann nicht gelöscht werden!" );
        }

        // Die 123 löschen:
        if( l.entf_wert( 123 ) )
        {
            System.out.println( l.toString() );
        }
        else
        {
            System.out.println( "die 123 kann nicht gelöscht werden!" );
```

```
}

// Die 112 löschen:
if( l.entf_wert( 112 ) )
{
    System.out.println( l.toString() );
}
else
{
    System.out.println( "die 112 kann nicht gelöscht werden!" );
}

// Hinter der 45 noch eine 50 einfügen:
if( l.einf_HinterWert( 45, 50 ) )
{
    System.out.println( l.toString() );
}
else
{
    System.out.println( "hinter 45 kann nichts eingefügt werden!" );
}

// Die 45 löschen:
if( l.entf_wert( 45 ) )
{
    System.out.println( l.toString() );
}
else
{
    System.out.println( "die 45 kann nicht gelöscht werden!" );
}

// Vor der 34 noch eine 30 einfügen:
if( l.einf_VorWert( 34, 30 ) )
{
    System.out.println( l.toString() );
}
else
{
    System.out.println( "vor 34 kann nichts eingefügt werden!" );
}

// Vor der 50 noch eine 48 einfügen:
if( l.einf_VorWert( 50, 48 ) )
{
    System.out.println( l.toString() );
}
else
{
    System.out.println( "vor 50 kann nichts eingefügt werden!" );
}
```

```
        // Vor der nicht existierenden 10 noch eine 8 einfügen
        // (klappt hoffentlich nicht):
        if( l.einf_VorWert( 10, 8 ) )
        {
            System.out.println( l.toString() );
        }
        else
        {
            System.out.println( "vor 10 kann nichts eingefügt werden!" );
        }

    }


} // public class teste_einfacheListe
```

Die Ausgabe sieht folgendermaßen aus:

```
Liste:  El.34,  El.45,  El.112,  El.123.
45 ist enthalten
123 ist nicht enthalten
die 30 kann nicht gelöscht werden!
Liste:  El.34,  El.45,  El.112.
Liste:  El.34,  El.45.
hinter 45 kann nichts eingefügt werden!
Liste:  El.34.
Liste:  El.30,  El.34.
vor 50 kann nichts eingefügt werden!
vor 10 kann nichts eingefügt werden!
```

A.2.5.2 C++-Version

Die Musterlösung in C++ ist auffallend ähnlich der Java-Version (siehe **A.2.5.1 Java-Version** auf Seite 133):

```
// Time-stamp: "11.05.03 17:12 einfacheListe.cpp klaus@wachtler.de"
//
// Lösung der Übungsaufgabe: Erstellen einer einfachen Liste
//
// ANSI-C++
//
// Linux:    g++ -Wall einfacheListe.cpp && ./a.out
//
// VC++ 6.0: cl /Zi /GX einfacheListe.cpp
//           .\einfacheListe
```

```
#include <iostream>
using namespace std;

// Die eigentliche Liste besteht aus einem Zeiger auf das erste Element.
class einfacheListe
{
private:

  // Datentyp für ein Element der Liste:
  class ListenElement
  {
  private:
    int           nutzdaten;  // die Daten eines Elements
    ListenElement *next;      // Verweis auf Nachfolger

  public:

    // Konstruktor
    ListenElement( int daten = 0 )
    {
      nutzdaten = daten;
      next = NULL;
    }

    // Destruktor
    // Achtung! Weil das delete next den Destruktor gleich wieder
    // für den Nachfolger aufruft, in dem wiederum ein delete next
    // steht, wird rekursiv die gesamte Liste ab dem aktuellen Element
    // freigegeben!
    // Wenn man nur ein Element freigeben will, aber nicht seine
    // Nachfolger, dann muß man zuvor sein next zu NULL setzen!
    virtual ~ListenElement()
    {
      delete next;
    }

    // fügt hinter sich ein neues Element ein:
    void einf_dahinter( int wert )
    {
      // dazu den bisherigen Nachfolger merken, um
      // ihn später als Nach-Nachfolger einsetzen zu können:
      ListenElement *uebernaechster = next;

      // neues Element schaffen und anhängen:
      next = new ListenElement( wert );

      // bisherigen Nachfolger dahinter setzen:
      next->setNext( uebernaechster );
    }
```

```
void setDaten( int daten )
{
  nutzdaten = daten;
}

int getDaten()
{
  return nutzdaten;
}

void setNext( ListenElement *next_neu )
{
  next = next_neu;
}

ListenElement *getNext()
{
  return next;
}


// sucht nach einem Wert, und liefert das ListenElement
// (falls gefunden), oder NULL.
ListenElement *sucheWert( int wert )
{
  // Schleife notfalls über alle Elemente:
  ListenElement *aktuellesElement = this;
  do
  {
    if( aktuellesElement->getDaten()==wert )
    {
      return aktuellesElement;
    }
    aktuellesElement = aktuellesElement->getNext();
  }
  while( aktuellesElement->getNext()!=NULL );

  // wenn alle Elemente nicht paßten, dann ist die Suche
  // fehlgeschlagen:
  return NULL;
}

// sucht nach dem Vorgänger eines Werts, und liefert das
// ListenElement (falls gefunden), oder NULL.
ListenElement *sucheVorgaengerVonWert( int wert )
{
  // Schleife notfalls über alle Elemente:
  ListenElement *aktuellesElement = this;
  do
  {
    if( aktuellesElement->next!=NULL
```

```
            &&
            aktuellesElement->next->getDaten()==wert
            )
        {
          return aktuellesElement;
        }
        aktuellesElement = aktuellesElement->getNext();
      }
      while( aktuellesElement->getNext()!=NULL );

      // wenn alle Elemente nicht paßten, dann ist die Suche
      // fehlgeschlagen:
      return NULL;
    }

    // Ausgabe einer Liste in einen ostream:
    // Element ausgeben, dann ggf. den Rest
    virtual void tostream( ostream &o ) const
    {
      o << " El." << nutzdaten;
      if( next==NULL )
      {
        o << ".";
      }
      else
      {
        o << ", ";
        next->tostream( o );
      }
    }

  }; // class ListenElement

  ListenElement   *anfang;

public:

  // Konstruktor
  einfacheListe()
  {
    anfang = NULL;
  }


  // Destruktor
  virtual ~einfacheListe()
  {
    // Listenelemente freigeben:
    delete anfang;
  }

  // fügt einen neuen Wert am Anfang ein:
```

```
void einf_vorn( int wert )
{
  // das bisherige erste Element merken, um es später zum
  // zweiten zu machen (auch wenn das bisherige erste
  // NULL ist, also die Liste bisher leer ist):
  ListenElement  *zweites = anfang;

  // ein neues erstes Element erzeugen, und mit wert
  // initialisieren:
  anfang = new ListenElement( wert );

  // das bisherige erste Element (oder NULL) hinten
  // anhängen:
  anfang->setNext( zweites );
}

// fügt einen neuen Wert am Ende ein:
void einf_hinten( int wert )
{
  // Wenn die Liste noch leer ist, dann einfach ein
  // Element an den Anfang setzen:
  if( anfang==NULL )
  {
    anfang = new ListenElement( wert );
  }
  else
  {
    // Suche nach dem letzten Element:
    ListenElement *letztes = anfang;
    while( letztes->getNext()!=NULL )
    {
      letztes = letztes->getNext();
    }
    // letztes hat jetzt keinen Nachfolger mehr.
    // Das wird geändert:
    letztes->einf_dahinter( wert );
  }
}

// versucht, ein neues Element wert hinter einem gegebenen Wert
// hinterwas einzusetzen.
// Das kann natürlich fehlschlagen, wenn das Element, hinter
// dem eingefügt werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
bool einf_HinterWert( int hinterwas, int wert )
{
  // Leere Liste?
  if( anfang==NULL )
  {
    return false;
```

```
  }
  else
  {
    ListenElement *element = anfang->sucheWert( hinterwas );
    if( element==NULL )
    {
      return false;
    }
    else
    {
      element->einf_dahinter( wert );
      return true;
    }
  }
}


// versucht, ein neues Element wert vor einem gegebenen Wert
// vorwas einzusetzen.
// Das kann natürlich fehlschlagen, wenn das Element, vor
// dem eingefügt werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
bool einf_VorWert( int vorwas, int wert )
{
  // Leere Liste?
  if( anfang==NULL )
  {
    return false;
  }
  // dann könnte es natürlich sein, daß gleich vor das erste
  // Element eingefügt werden soll:
  else if( anfang->getDaten()==vorwas )
  {
    einf_vorn( wert );
    return true;
  }
  else
  {
    ListenElement *element = anfang->sucheVorgaengerVonWert( vorwas );
    if( element==NULL )
    {
      return false;
    }
    else
    {
      element->einf_dahinter( wert );
      return true;
    }
  }
}
```

```
// sucht nach einem Wert, und liefert true, wenn der gesuchte Wert
// enthalten ist:
bool sucheWert( int wert )
{
  // Leere Liste?
  if( anfang==NULL )
  {
    return false;
  }
  else
  {
    return anfang->sucheWert( wert )!=NULL;
  }

}

// versucht, ein Element wert aus der Liste zu löschen.
// Das kann natürlich fehlschlagen, wenn das Element, das
// gelöscht werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
bool entf_wert( int wert )
{
  // Leere Liste?
  if( anfang==NULL )
  {
    return false;
  }
  // dann könnte es natürlich sein, daß gleich das erste
  // Element gelöscht werden soll:
  else if( anfang->getDaten()==wert )
  {
    ListenElement *zweites = anfang->getNext(); // kurz merken

    // Das bisherige erste Element muß gelöscht werden.
    // Dazu:
    // 1. seinen Nachfolger abhängen (damit beim folgenden Frei-
    //    geben nicht der gesamte Listenrest freigegeben wird):
    anfang->setNext( NULL );
    // 2. das erste Element löschen:
    delete anfang;


    // jetzt das zweite Element zum ersten machen:
    anfang = zweites;

    return true;
  }
  else
  {
```

```
      // Ansonsten brauchen wir den Vorgänger des zu löschenden
      // Elements; dessen Nachfolger muß gelöscht werden:
      ListenElement *element = anfang->sucheVorgaengerVonWert( wert );
      if( element==NULL )
      {
        return false;
      }
      else
      {
        // das zu löschende Element ist jetzt element->getNext().
        // Dieses Element kurz merken, damit wir es zum Schluß
        // freigeben können:
        ListenElement *zuloeschen = element->getNext();

        element->setNext( element->getNext()->getNext() );

        // auch hier den Nachfolger plattmachen, damit beim
        // Freigeben von zuloeschen nicht der ganze Listenrest
        // freigegeben wird:
        zuloeschen->setNext( NULL );
        delete zuloeschen;

        return true;
      }
    }
  }


  // Ausgabe einer Liste in einen String:
  // Element ausgeben, dann den Rest
  void tostream( ostream &o ) const
  {
    o<< "Liste: ";
    if( anfang==NULL )  o << "-";
    else                anfang->tostream( o );
  }

}; // class einfacheListe


ostream &operator<<( ostream &o, const einfacheListe &l )
{
    l.tostream( o );
    return o;
}

int main( int nargs, char **args )
{
    einfacheListe   l;

    // ein paar jeweils von vorne her einfügen:
```

```
l.einf_vorn( 45 );
l.einf_vorn( 34 );

// ein paar von hinten:
l.einf_hinten( 112 );
l.einf_hinten( 123 );
cout << l << endl;

// ist der Wert 45 enthalten?
cout << "45 ist "
     << ( l.sucheWert( 45 ) ? "" : "nicht " )
     << "enthalten" << endl;

// ist der Wert 123 enthalten?
cout << "123 ist "
     << ( l.sucheWert( 123 ) ? "" : "nicht " )
     << "enthalten" << endl;

// Die (nicht existierende) 30 löschen:
if( l.entf_wert( 30 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 30 kann nicht gelöscht werden!" << endl;
}

// Die 123 löschen:
if( l.entf_wert( 123 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 123 kann nicht gelöscht werden!" << endl;
}

// Die 112 löschen:
if( l.entf_wert( 112 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 112 kann nicht gelöscht werden!" << endl;
}

// Hinter der 45 noch eine 50 einfügen:
if( l.einf_HinterWert( 45, 50 ) )
{
```

```
        cout << l << endl;
    }
    else
    {
        cout << "hinter 45 kann nichts eingefügt werden!" << endl;
    }

    // Die 45 löschen:
    if( l.entf_wert( 45 ) )
    {
        cout << l << endl;
    }
    else
    {
        cout << "die 45 kann nicht gelöscht werden!" << endl;
    }

    // Vor der 34 noch eine 30 einfügen:
    if( l.einf_VorWert( 34, 30 ) )
    {
        cout << l << endl;
    }
    else
    {
        cout << "vor 34 kann nichts eingefügt werden!" << endl;
    }

    // Vor der 50 noch eine 48 einfügen:
    if( l.einf_VorWert( 50, 48 ) )
    {
        cout << l << endl;
    }
    else
    {
        cout << "vor 50 kann nichts eingefügt werden!" << endl;
    }

    // Vor der nicht existierenden 10 noch eine 8 einfügen
    // (klappt hoffentlich nicht):
    if( l.einf_VorWert( 10, 8 ) )
    {
        cout << l << endl;
    }
    else
    {
        cout << "vor 10 kann nichts eingefügt werden!" << endl;
    }



    return 0;
} // main( int nargs, char **args )
```

Die wesentlichen Unterschiede zur Java-Version sind:

- Die in Java implizit vorhandenen Referenzen jedes Elements auf seinen Nachfolger, auf ein aktuelles Element, etc., werden in der C++-Version explizit durch Zeiger ausgedrückt. Es wäre zwar auch möglich, hier Referenzen zu verwenden; allerdings gäbe es in C++ keine direkte Möglichkeit, eine „ungültige" Referenz auszudrücken (beispielsweise für den Nachfolger des letzten Elements). Deshalb werden hier Zeiger verwendet; ein ungültiger Verweis hat dann den Wert `NULL`.
- Weil man jetzt über Zeiger auf Elemente zugreift, müßte man beispielsweise statt `next.getDaten()` eigentlich `(*next).getDaten()` schreiben, um von dem Zeiger aus (`next`) erst mit `*` zum eigentlichen Objekt zu kommen, von dort aus mit `.` das Element (hier: `getDaten()`) zu erreichen. Es wird hier dann die vereinfachende Schreibweise `next->getDaten()` verwendet.
- Nach dem Ende einer `class`-Definition muß ein Semikolon (`;`) stehen.
- entsprechend der Tatsache, daß es in C++ keine *garbage collection* gibt, muß man beim Löschen eines Elements oder der ganzen Liste mit `delete` die früher mit `new` allokierten Objekte freigeben.
- die Verwandlung eines Objekts in einen String (`toString()` in Java) wird durch die Ausgabe zu einem `ostream` ersetzt.
- Das Testprogramm (`main()`) ist gleich im selben Quelltext enthalten; für eine Verwendung als Modul (ohne eigenes `main()`) wäre der letzte Teil zu löschen, auszukommentieren, oder mit `ifdef TESTPROGRAMM`... stillzulegen.

Die Ausgabe sieht folgendermaßen aus:

```
Liste:  El.34,  El.45,  El.112,  El.123.
45 ist enthalten
123 ist nicht enthalten
die 30 kann nicht gelöscht werden!
Liste:  El.34,  El.45,  El.112.
Liste:  El.34,  El.45.
hinter 45 kann nichts eingefügt werden!
Liste:  El.34.
Liste:  El.30,  El.34.
vor 50 kann nichts eingefügt werden!
vor 10 kann nichts eingefügt werden!
```

A.2.6 Einfach verkettete Liste eines beliebigen Typs

A.2.6.1 Java-Version

Als Beispieldatentyp wird wieder der Student aus **6.1.5 Java-Version** auf Seite 88 verwendet. Zur Demonstration der Referenzsemantik (siehe unten) wird allerdings noch eine Methode `setVorname( String )` eingefügt, um ein Objekt im Nachhinein manipulieren zu können:

```
// Time-stamp: "11.05.03 15:35 Student.java klaus@wachtler.de"
//
// Testklasse zum Einfügen in einen Container

import java.util.*;

public class Student
{
    private String   VorName;
    private String   NachName;
    public  int      Matrikelnummer;

    public Student()
    {
        VorName = NachName = "?";
        Matrikelnummer = -1;
    }

    public Student( String v, String n, int m )
    {
        VorName = v;
        NachName = n;
        Matrikelnummer = m;
    }

    public String toString()
    {
        return "<" + VorName + " "
               + NachName + " "
               + Matrikelnummer  + ">";
    }

    // vergleicht zwei Studenten auf Gleichheit
    public boolean equals( Object s )
    {
        return Matrikelnummer==((Student)s).Matrikelnummer;
    }

    public void setVorname( String neuerVorname )
    {
```

```
        VorName = neuerVorname;
    }

} // class Student
```

Die Lösung besteht ansonsten aus einer leicht modifizierten Version der Musterlösung **A.2.5 Einfach verkettete Liste** auf Seite 133:

Die wesentlichen Änderungen bestehen aus dem Ersatz des `int`-Typs durch Object sowie des Operators `==` durch den Aufruf der `Object`-Methode `equals()`.

```
// Time-stamp: "11.05.03 13:53 einfacheListeGenerisch.java klaus@wachtler.de"
//
// Lösung der Übungsaufgabe: Erstellen einer einfach verketteten Liste
// eines beliebigen Typs
//
// 11.05.2003 kw   Beim Vergleich von Objekten == durch equals() ersetzt


// Die eigentliche Liste besteht aus einer Referenz auf das erste
// Element (diese ist bei einer leeren Liste null);
// jedes Element wiederum hat Nutzdaten und eine Referenz auf
// das nachfolgende Element (oder null).
public class einfacheListeGenerisch
{

    // Datentyp für ein Element der Liste:
    private class ListenElement
    {
        private Object        nutzdaten;  // die Daten eines Elements
        private ListenElement next;       // Verweis auf Nachfolger

        public ListenElement( Object daten )
        {
            nutzdaten = daten;
            next = null;
        }

        // fügt hinter sich ein neues Element ein:
        public void einf_dahinter( Object wert )
        {
            // dazu den bisherigen Nachfolger merken, um
            // ihn später als Nach-Nachfolger einsetzen zu können:
            ListenElement uebernaechster = next;

            // neues Element schaffen und anhängen:
            next = new ListenElement( wert );

            // bisherigen Nachfolger dahinter setzen:
```

```
        next.setNext( uebernaechster );
    }

    public void setDaten( Object daten )
    {
        nutzdaten = daten;
    }

    public Object getDaten()
    {
        return nutzdaten;
    }

    public void setNext( ListenElement next_neu )
    {
        next = next_neu;
    }

    public ListenElement getNext()
    {
        return next;
    }


    // sucht nach einem Wert, und liefert das ListenElement
    // (falls gefunden), oder null.
    public ListenElement sucheWert( Object wert )
    {
        // Schleife notfalls über alle Elemente:
        ListenElement aktuellesElement = this;
        do
        {
            if( aktuellesElement.getDaten().equals( wert ) )
            {
                return aktuellesElement;
            }
            aktuellesElement = aktuellesElement.getNext();
        }
        while( aktuellesElement.getNext()!=null );

        // wenn alle Elemente nicht paßten, dann ist die Suche
        // fehlgeschlagen:
        return null;
    }

    // sucht nach dem Vorgänger eines Werts, und liefert das
    // ListenElement (falls gefunden), oder null.
    public ListenElement sucheVorgaengerVonWert( Object wert )
    {
        // Schleife notfalls über alle Elemente:
        ListenElement aktuellesElement = this;
```

```
        do
        {
            if( aktuellesElement.next!=null
                &&
                aktuellesElement.next.getDaten().equals( wert )
                )
            {
                return aktuellesElement;
            }
            aktuellesElement = aktuellesElement.getNext();
        }
        while( aktuellesElement.getNext()!=null );

        // wenn alle Elemente nicht paßten, dann ist die Suche
        // fehlgeschlagen:
        return null;
    }

    // Ausgabe einer Liste in einen String:
    // Element ausgeben, dann ggf. den Rest
    public String toString()
    {
        return ( " El." + nutzdaten
                 + ( next==null ? "." : ", " + next.toString() )
                 );
    }

} // class ListenElement


private ListenElement   anfang;

public einfacheListeGenerisch()
{
    anfang = null;
}

// fügt einen neuen Wert am Anfang ein:
public void einf_vorn( Object wert )
{
    // das bisherige erste Element merken, um es später zum
    // zweiten zu machen (auch wenn das bisherige erste
    // null ist, also die Liste bisher leer ist):
    ListenElement  zweites = anfang;

    // ein neues erstes Element erzeugen, und mit wert
    // initialisieren:
    anfang = new ListenElement( wert );

    // das bisherige erste Element (oder null) hinten
    // anhängen:
```

```
        anfang.setNext( zweites );
    }

    // fügt einen neuen Wert am Ende ein:
    public void einf_hinten( Object wert )
    {
        // Wenn die Liste noch leer ist, dann einfach ein
        // Element an den Anfang setzen:
        if( anfang==null )
        {
            anfang = new ListenElement( wert );
        }
        else
        {
            // Suche nach dem letzten Element:
            ListenElement letztes = anfang;
            while( letztes.getNext()!=null )
            {
                letztes = letztes.getNext();
            }
            // letztes hat jetzt keinen Nachfolger mehr.
            // Das wird geändert:
            letztes.einf_dahinter( wert );
        }
    }


    // versucht, ein neues Element wert hinter einem gegebenen Wert
    // hinterwas einzusetzen.
    // Das kann natürlich fehlschlagen, wenn das Element, hinter
    // dem eingefügt werden soll, gar nicht existiert.
    // Dementsprechend wird true zurückgegeben, wenn alles geklappt
    // hat; sonst false.
    public boolean einf_HinterWert( Object hinterwas, Object wert )
    {
        // Leere Liste?
        if( anfang==null )
        {
            return false;
        }
        else
        {
            ListenElement element = anfang.sucheWert( hinterwas );
            if( element==null )
            {
                return false;
            }
            else
            {
                element.einf_dahinter( wert );
                return true;
            }
```

```
    }

}


// versucht, ein neues Element wert vor einem gegebenen Wert
// vorwas einzusetzen.
// Das kann natürlich fehlschlagen, wenn das Element, vor
// dem eingefügt werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
public boolean einf_VorWert( Object vorwas, Object wert )
{
    // Leere Liste?
    if( anfang==null )
    {
        return false;
    }
    // dann könnte es natürlich sein, daß gleich vor das erste
    // Element eingefügt werden soll:
    else if( anfang.getDaten().equals( vorwas ) )
    {
        einf_vorn( wert );
        return true;
    }
    else
    {
        ListenElement element = anfang.sucheVorgaengerVonWert( vorwas );
        if( element==null )
        {
            return false;
        }
        else
        {
            element.einf_dahinter( wert );
            return true;
        }
    }

}


// sucht nach einem Wert, und liefert ihn, falls vorhanden:
public Object sucheWert( Object wert )
{
    // Leere Liste?
    if( anfang==null )
    {
        return null;
    }
    else
```

```
    {
        ListenElement   gefunden = anfang.sucheWert( wert );
        return gefunden!=null ? gefunden.getDaten() : null;
    }

}

// versucht, ein Element wert aus der Liste zu löschen.
// Das kann natürlich fehlschlagen, wenn das Element, das
// gelöscht werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
public boolean entf_wert( Object wert )
{
    // Leere Liste?
    if( anfang==null )
    {
        return false;
    }
    // dann könnte es natürlich sein, daß gleich das erste
    // Element gelöscht werden soll:
    else if( anfang.getDaten().equals( wert ) )
    {
        anfang = anfang.getNext();
        return true;
    }
    else
    {
        // Ansonsten brauchen wir den Vorgänger des zu löschenden
        // Elements; dessen Nachfolger muß gelöscht werden:
        ListenElement element = anfang.sucheVorgaengerVonWert( wert );
        if( element==null )
        {
            return false;
        }
        else
        {
            element.setNext( element.getNext().getNext() );
            return true;
        }
    }

}


// Ausgabe einer Liste in einen String:
// Element ausgeben, dann den Rest
public String toString()
{
    return "Liste: " + ( anfang==null ? "-" : anfang.toString() );
}
```

```
} // public class einfacheListeGenerisch
```

Die hier vorgestellte Lösung realisiert eine Referenzsemantik; Wenn der Aufrufer also nach dem Einfügen von Werten die Objekte manipuliert, wirkt sich dies auf den Inhalt der Liste aus.

Kurz vor Ende des Testprogramms wird zur Demonstration einer der vorher eingefügten Werte (`Hein`) geändert, und anschließend der Listeninhalt nochmals ausgegeben:

```
// Time-stamp: "11.05.03 18:37 teste_einfacheListeGenerisch.java klaus@wachtler.de"
//
// javac teste_einfacheListeGenerisch.java einfacheListe.java
// java teste_einfacheListeGenerisch
//
// Testprogramm für class einfacheListeGenerisch

import java.util.*;

public class teste_einfacheListeGenerisch
{
    public static void main( String args[] )
    {
        einfacheListeGenerisch   l = new einfacheListeGenerisch();

        // ein paar jeweils von vorne her einfügen:
        l.einf_vorn( new Student( "K.", "Meier", 123456 ) );
        Student  Hein = new Student( "Hein", "Blöd", 987654 );
        l.einf_vorn( Hein );

        // einen von hinten:
        l.einf_hinten(  new Student( "E.", "Steinberger", 876543 )  );

        System.out.println( l.toString() );

        Student  gefunden;
        // ist der Student mit der Matrikelnummer 987654 enthalten?
        gefunden = (Student)l.sucheWert( new Student( "", "", 987654 ) );
        System.out.println( "987654 ist "
                            + ( ( gefunden!=null )
                                ? "so: " + gefunden.toString()
                                : "nicht"
                                )
                            + " enthalten"
                            );

        // ist der Student mit der Matrikelnummer 333333 enthalten?
```

```
        gefunden = (Student)l.sucheWert( new Student( "", "", 333333 ) );
        System.out.println( "333333 ist "
                            + ( ( gefunden!=null )
                                ? "so: " + gefunden.toString()
                                : "nicht"
                                )
                            + " enthalten"
                            );

        // Die (nicht existierende) 333333 löschen:
        if( l.entf_wert( new Student( "", "", 333333 ) ) )
        {
            System.out.println( "333333 gelöscht; " + l.toString() );
        }
        else
        {
            System.out.println( "der 333333 kann nicht gelöscht werden!" );
        }

        // Die 876543 löschen:
        if( l.entf_wert( new Student( "", "", 876543 ) ) )
        {
            System.out.println( "876543 gelöscht; " + l.toString() );
        }
        else
        {
            System.out.println( "der 987654 kann nicht gelöscht werden!" );
        }


        // Hinter Hein noch einen einfügen:
        if( l.einf_HinterWert( Hein, new Student( "S.", "Schmidt", 542366 ) ) )
        {
            System.out.println( l.toString() );
        }
        else
        {
            System.out.println( "hinter Hein kann nichts eingefügt werden!" );
        }

        System.out.println( l );

        // Der zu Anfang eingefügte Hein wird jetzt geändert.
        // Dadurch ändert sich auch der Inhalt der Liste!
        Hein.setVorname( "Hinnerk" );

        System.out.println( l );
    }


} // public class teste_einfacheListeGenerisch
```

Die Referenzsemantik ist im Vergleich der letzten beiden Zeilen der Ausgabe zu erkennen:

```
Liste:  El.<Hein Blöd 987654>,  El.<K. Meier 123456>,  El.<E. Steinberger 876543>.
987654 ist so: <Hein Blöd 987654> enthalten
333333 ist nicht enthalten
der 333333 kann nicht gelöscht werden!
876543 gelöscht; Liste:  El.<Hein Blöd 987654>,  El.<K. Meier 123456>.
Liste:  El.<Hein Blöd 987654>,  El.<S. Schmidt 542366>,  El.<K. Meier 123456>.
Liste:  El.<Hein Blöd 987654>,  El.<S. Schmidt 542366>,  El.<K. Meier 123456>.
Liste:  El.<Hinnerk Blöd 987654>,  El.<S. Schmidt 542366>,  El.<K. Meier 123456>.
```

A.2.6.2 C++-Version

Um die C++-Version aus **A.2.5 Einfach verkettete Liste** auf Seite 133 generisch zu machen, muß man:

- die jeweilige `class...` durch `template<class T> class..` ersetzen
- innerhalb der Klassen `ListenElement` und `beliebigeListe` (beziehungsweise der `templates`) den festen Typ `int` durch den `template`-Parameter `T` ersetzen
- in `beliebigeListe` den Typ `ListenElement` durch `ListenElement<T>` ersetzen
- die Operatoren zur Ausgabe der einzelnen Klassenelemente zu `template`-Funktionen umwandeln

Wie in C++ eher üblich wird hier eine Wertsemantik implementiert, das heißt bei jedem Einfügevorgang wird eine Kopie des einzufügenden Objekts erzeugt und diese Kopie wird in die Liste eingehängt. Die Kopie wird freigegeben, wenn das Element aus der Liste gelöscht wird, oder die gesamte Liste aufgelöst wird. Dadurch ist die Lebensdauer der Listenelemente vollständig von der Lebensdauer der einzufügenden Objekte entkoppelt.

```
// Time-stamp: "17.09.04 12:28 einfacheListeGenerisch.h klaus@wachtler.de"
//
// Testprogramm zur Übungsaufgabe: Erstellen einer generischen Liste
//
// ANSI-C++
//
```

```
// Änderungen:
// 11.05.2003 kw  aufgeteilt in Headerdatei (einfacheListeGenerisch.h)
//                und Testprogramm (templliste.cpp),
//                Klasse von "beliebigeListe" zu
//                "einfacheListeGenerisch" umbenannt,
//                ListenElement jetzt als Unterklasse in
//                einfacheListeGenerisch

#ifndef _EINFACHELISTE_GENERISCH_H_
#define _EINFACHELISTE_GENERISCH_H_


#include <iostream>
using namespace std;

// Die eigentliche Liste besteht aus einem Zeiger auf das erste Element.
template<class T>class einfacheListeGenerisch
{
 private:

  // Datentyp für ein Element der Liste:
  class ListenElement
    {
    private:

      // Die folgende Zeile entscheidet darüber, daß dieser Container
      // eine Wertsemantik implementiert:
      // (nutzdaten) ist ein komplettes Objekt vom Typ T der
      // Nutzdaten, und erhält beim Erzeugen eines ListenElements
      // durch Zuweisung eine Kopie der einzufügenden Daten (siehe im
      // Konstruktor die Zeile:
      //    nutzdaten = daten;
      // .
      // Eine Referenzsemantik würde man erhalten, wenn hier anstatt
      //    T nutzdaten;
      // stehen würde:
      //    T *nutzdaten_p;
      // und die restlichen Methoden entsprechend angepaßt würden
      // (der Konstruktor ListenElement( const T &daten ) müßte dann
      // beispielsweise enthalten:
      //    nutzdaten_p = &daten;
      // ).
      // Dann wäre in den ListenElementen keine Kopie, sondern ein
      // Zeiger auf die Originaldaten vorhanden, und eine Änderung der
      // eingefügten Objekte ginge einher mit einer Änderung des
      // Containerinhalts.
      T             nutzdaten;  // die Daten eines Elements
      ListenElement *next;       // Verweis auf Nachfolger

    public:
```

```
// Konstruktor
ListenElement( const T &daten )
  {
    nutzdaten = daten;
    next = NULL;
  }

// Destruktor
// Achtung! Weil das delete next den Destruktor gleich wieder
// für den Nachfolger aufruft, in dem wiederum ein delete next
// steht, wird rekursiv die gesamte Liste ab dem aktuellen Element
// freigegeben!
// Wenn man nur ein Element freigeben will, aber nicht seine
// Nachfolger, dann muß man zuvor sein next zu NULL setzen!
virtual ~ListenElement()
  {
    delete next;
  }

// fügt hinter sich ein neues Element ein:
void einf_dahinter( const T &wert )
  {
    // dazu den bisherigen Nachfolger merken, um
    // ihn später als Nach-Nachfolger einsetzen zu können:
    ListenElement *uebernaechster = next;

    // neues Element schaffen und anhängen:
    next = new ListenElement( wert );

    // bisherigen Nachfolger dahinter setzen:
    next->setNext( uebernaechster );
  }

void setDaten( const T &daten )
  {
    nutzdaten = daten;
  }

const T &getDaten()
  {
    return nutzdaten;
  }

void setNext( ListenElement *next_neu )
  {
    next = next_neu;
  }

ListenElement *getNext()
  {
    return next;
```

```
  }


// sucht nach einem Wert, und liefert das ListenElement
// (falls gefunden), oder NULL.
ListenElement *sucheWert( const T &wert )
  {
    // Schleife notfalls über alle Elemente:
    ListenElement *aktuellesElement = this;
    do
    {
      if( aktuellesElement->getDaten()==wert )
      {
        return aktuellesElement;
      }
      aktuellesElement = aktuellesElement->getNext();
    }
    // Änderung 04.07.2002 KW: in der bisherigen Form:
    //  while( aktuellesElement->getNext()!=NULL );
    // wird versehentlich das letzte Element der Liste nicht geprüft!
    // Besser ist es so;
    while( aktuellesElement!=NULL );


    // wenn alle Elemente nicht paßten, dann ist die Suche
    // fehlgeschlagen:
    return NULL;
  }

// sucht nach dem Vorgänger eines Werts, und liefert das
// ListenElement (falls gefunden), oder NULL.
ListenElement *sucheVorgaengerVonWert( const T &wert )
  {
    // Schleife notfalls über alle Elemente:
    ListenElement *aktuellesElement = this;
    do
    {
      if( aktuellesElement->next!=NULL
          &&
          aktuellesElement->next->getDaten()==wert
          )
      {
        return aktuellesElement;
      }
      aktuellesElement = aktuellesElement->getNext();
    }
    while( aktuellesElement->getNext()!=NULL );


    // wenn alle Elemente nicht paßten, dann ist die Suche
    // fehlgeschlagen:
    return NULL;
```

```
      }

    // Ausgabe einer Liste in einen ostream:
    // Element ausgeben, dann ggf. den Rest
    virtual void tostream( ostream &o ) const
      {
        o << " El." << nutzdaten;
        if( next==NULL )
        {
          o << ".";
        }
        else
        {
          o << ", ";
          next->tostream( o );
        }
      }

  }; // class ListenElement

 ListenElement   *anfang;

public:

 // Konstruktor
 einfacheListeGenerisch()
   {
     anfang = NULL;
   }

 // Destruktor
 virtual ~einfacheListeGenerisch()
   {
     // Listenelemente freigeben:
     delete anfang;
   }

 // fügt einen neuen Wert am Anfang ein:
 void einf_vorn( const T &wert )
   {
     // das bisherige erste Element merken, um es später zum
     // zweiten zu machen (auch wenn das bisherige erste
     // NULL ist, also die Liste bisher leer ist):
     ListenElement  *zweites = anfang;

     // ein neues erstes Element erzeugen, und mit wert
     // initialisieren:
     anfang = new ListenElement( wert );

     // das bisherige erste Element (oder NULL) hinten
     // anhängen:
```

```
    anfang->setNext( zweites );
  }

// fügt einen neuen Wert am Ende ein:
void einf_hinten( const T &wert )
  {
    // Wenn die Liste noch leer ist, dann einfach ein
    // Element an den Anfang setzen:
    if( anfang==NULL )
    {
      anfang = new ListenElement( wert );
    }
    else
    {
      // Suche nach dem letzten Element:
      ListenElement *letztes = anfang;
      while( letztes->getNext()!=NULL )
      {
        letztes = letztes->getNext();
      }
      // letztes hat jetzt keinen Nachfolger mehr.
      // Das wird geändert:
      letztes->einf_dahinter( wert );
    }
  }

// versucht, ein neues Element wert hinter einem gegebenen Wert
// hinterwas einzusetzen.
// Das kann natürlich fehlschlagen, wenn das Element, hinter
// dem eingefügt werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
bool einf_HinterWert( const T &hinterwas, const T &wert )
  {
    // Leere Liste?
    if( anfang==NULL )
    {
      return false;
    }
    else
    {
      ListenElement *element = anfang->sucheWert( hinterwas );
      if( element==NULL )
      {
        return false;
      }
      else
      {
        element->einf_dahinter( wert );
        return true;
      }
```

```
    }
  }


// versucht, ein neues Element wert vor einem gegebenen Wert
// vorwas einzusetzen.
// Das kann natürlich fehlschlagen, wenn das Element, vor
// dem eingefügt werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
bool einf_VorWert( const T &vorwas, const T &wert )
  {
    // Leere Liste?
    if( anfang==NULL )
    {
      return false;
    }
    // dann könnte es natürlich sein, daß gleich vor das erste
    // Element eingefügt werden soll:
    else if( anfang->getDaten()==vorwas )
    {
      einf_vorn( wert );
      return true;
    }
    else
    {
      ListenElement *element
        = anfang->sucheVorgaengerVonWert( vorwas );
      if( element==NULL )
      {
        return false;
      }
      else
      {
        element->einf_dahinter( wert );
        return true;
      }
    }
  }


// sucht nach einem Wert, und liefert einen Zeiger darauf, oder NULL:
const T *sucheWert( const T &wert )
  {
    // Leere Liste?
    if( anfang==NULL )
    {
      return NULL;
    }
    else
    {
```

```
      ListenElement *LE_gefunden = anfang->sucheWert( wert );
      return LE_gefunden ? &(LE_gefunden->getDaten()) : NULL;
    }


  }

// versucht, ein Element wert aus der Liste zu löschen.
// Das kann natürlich fehlschlagen, wenn das Element, das
// gelöscht werden soll, gar nicht existiert.
// Dementsprechend wird true zurückgegeben, wenn alles geklappt
// hat; sonst false.
bool entf_wert( const T &wert )
  {
    // Leere Liste?
    if( anfang==NULL )
    {
      return false;
    }
    // dann könnte es natürlich sein, daß gleich das erste
    // Element gelöscht werden soll:
    else if( anfang->getDaten()==wert )
    {
      ListenElement *zweites = anfang->getNext(); // kurz merken

      // Das bisherige erste Element muß gelöscht werden.
      // Dazu:
      // 1. seinen Nachfolger abhängen (damit beim folgenden Frei-
      //    geben nicht der gesamte Listenrest freigegeben wird):
      anfang->setNext( NULL );
      // 2. das erste Element löschen:
      delete anfang;

      // jetzt das zweite Element zum ersten machen:
      anfang = zweites;

      return true;
    }
    else
    {
      // Ansonsten brauchen wir den Vorgänger des zu löschenden
      // Elements; dessen Nachfolger muß gelöscht werden:
      ListenElement *element
        = anfang->sucheVorgaengerVonWert( wert );
      if( element==NULL )
      {
        return false;
      }
      else
      {
        // das zu löschende Element ist jetzt element->getNext().
        // Dieses Element kurz merken, damit wir es zum Schluß
```

```
          // freigeben können:
          ListenElement *zuloeschen = element->getNext();

          element->setNext( element->getNext()->getNext() );

          // auch hier den Nachfolger plattmachen, damit beim
          // Freigeben von zuloeschen nicht der ganze Listenrest
          // freigegeben wird:
          zuloeschen->setNext( NULL );
          delete zuloeschen;

          return true;
        }
      }
    }


  // Ausgabe einer Liste in einen String:
  // Element ausgeben, dann den Rest
  void tostream( ostream &o ) const
    {
      o<< "Liste: ";
      if( anfang==NULL )  o << "-";
      else                anfang->tostream( o );
    }

}; // class einfacheListeGenerisch


template<class T> ostream &operator<<( ostream &o,
                                       const einfacheListeGenerisch<T> &l
                                       )
{
    l.tostream( o );
    return o;
}

#endif /* _EINFACHELISTE_GENERISCH_H_ */
```

Probiert wird im Testprogramm sowohl eine Liste des (elementaren) Typs `int` als auch eine Liste einer eigenen Klasse `Student`:

```
// Time-stamp: "11.05.03 18:37 teste_einfacheListeGenerisch.cpp klaus@wachtler.de"
//
// Lösung der Übungsaufgabe: Erstellen einer generischen Liste
//
// ANSI-C++
//
// Linux: g++ -Wall teste_einfacheListeGenerisch.cpp && ./a.out
//
```

```
// Änderungen:
// 04.07.2002 kw  In ListenElement::sucheWert( T wert )
//                wurde bisher versehentlich das letzte Element
//                nicht mehr geprüft.
//
// 11.05.2003 kw  aufgeteilt in Headerdatei (einfacheListeGenerisch.h)
//                und Testprogramm (teste_einfacheListeGenerisch.cpp),
//                Klasse von "beliebigeListe" zu
//                "einfacheListeGenerisch" umbenannt

#include <iostream>
using namespace std;

#include "einfacheListeGenerisch.h"


#include <string>

class Student
{
public:

  // Konstruktor
  Student( string Vorname = "",
           string Nachname = "",
           int Matrikelnummer = 0
           )
    : Vorname( Vorname ),
      Nachname( Nachname ),
      Matrikelnummer( Matrikelnummer )
  {
  }

  // copy-Konstruktor
  Student( const Student &rechteSeite )
    : Vorname( rechteSeite.Vorname ),
      Nachname( rechteSeite.Nachname ),
      Matrikelnummer( rechteSeite.Matrikelnummer )
  {
  }

  // Destruktor
  virtual ~Student()
  {
  }

  // Zuweisung Student = Student
  Student & operator=( const Student &rechteSeite )
  {
    Vorname = rechteSeite.Vorname;
    Nachname = rechteSeite.Nachname;
```

```
    Matrikelnummer = rechteSeite.Matrikelnummer;

    return *this;
  }

  // Vergleich Student == Student
  bool operator==( const Student &rechteSeite ) const
  {
    return Matrikelnummer == rechteSeite.Matrikelnummer;
  }

  // Ausgabe auf einen ostream
  virtual void streamout( ostream & s ) const
    {
        s << "(" << Vorname
          << "," << Nachname
          << "," << Matrikelnummer << ")";
    }

  void setVorname( string neuerVorname )
  {
    Vorname = neuerVorname;
  }

private:

    string Vorname;
    string Nachname;
    int Matrikelnummer;

}; // class Student...

// ggf. nach Student.cpp kopieren:
ostream &operator<<( ostream &s, const Student &obj )
{
    obj.streamout( s );
    return s;
}



int main( int nargs, char **args )
{
    // Liste mit int-Werten:
    einfacheListeGenerisch<int>   l;

    // ein paar jeweils von vorne her einfügen:
    l.einf_vorn( 45 );
    l.einf_vorn( 34 );

    // ein paar von hinten:
```

```
l.einf_hinten( 112 );
l.einf_hinten( 123 );
cout << l << endl;

// ist der Wert 45 enthalten?
cout << "45 ist "
     << ( l.sucheWert( 45 ) ? "" : "nicht " )
     << "enthalten" << endl;

// ist der Wert 123 enthalten?
cout << "123 ist "
     << ( l.sucheWert( 123 ) ? "" : "nicht " )
     << "enthalten" << endl;

// Die (nicht existierende) 30 löschen:
if( l.entf_wert( 30 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 30 kann nicht gelöscht werden!" << endl;
}

// Die 123 löschen:
if( l.entf_wert( 123 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 123 kann nicht gelöscht werden!" << endl;
}

// Die 112 löschen:
if( l.entf_wert( 112 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 112 kann nicht gelöscht werden!" << endl;
}


// Hinter der 45 noch eine 50 einfügen:
if( l.einf_HinterWert( 45, 50 ) )
{
    cout << l << endl;
}
else
{
```

```
    cout << "hinter 45 kann nichts eingefügt werden!" << endl;
}

// Die 45 löschen:
if( l.entf_wert( 45 ) )
{
    cout << l << endl;
}
else
{
    cout << "die 45 kann nicht gelöscht werden!" << endl;
}

// Vor der 34 noch eine 30 einfügen:
if( l.einf_VorWert( 34, 30 ) )
{
    cout << l << endl;
}
else
{
    cout << "vor 34 kann nichts eingefügt werden!" << endl;
}

// Vor der 50 noch eine 48 einfügen:
if( l.einf_VorWert( 50, 48 ) )
{
    cout << l << endl;
}
else
{
    cout << "vor 50 kann nichts eingefügt werden!" << endl;
}

// Vor der nicht existierenden 10 noch eine 8 einfügen
// (klappt hoffentlich nicht):
if( l.einf_VorWert( 10, 8 ) )
{
    cout << l << endl;
}
else
{
    cout << "vor 10 kann nichts eingefügt werden!" << endl;
}


// Liste mit Studenten:
einfacheListeGenerisch<Student> sl;


// ein paar jeweils von vorne her einfügen:
sl.einf_vorn( Student( "K.", "Meier", 123456 ) );
```

```
Student  Hein( "Hein", "Blöd", 987654 );
sl.einf_vorn( Hein );

// einen von hinten:
sl.einf_hinten(  Student( "E.", "Steinberger", 876543 )  );

cout << "Studentenliste=" << sl << endl;


const Student  *p_gefunden;
// ist der Student mit der Matrikelnummer 987654 enthalten?
p_gefunden = sl.sucheWert( Student( "", "", 987654 ) );
cout << "987654 ist ";
if( p_gefunden )
{
  cout << "so: " << *p_gefunden;
}
else
{
  cout << "nicht";
}
cout << endl;

// ist der Student mit der Matrikelnummer 333333 enthalten?
p_gefunden = sl.sucheWert( Student( "", "", 333333 ) );
cout << "333333 ist ";
if( p_gefunden )
{
  cout << "so: " << *p_gefunden;
}
else
{
  cout << "nicht";
}
cout << " enthalten" << endl;

// Die (nicht existierende) 333333 löschen:
if( sl.entf_wert( Student( "", "", 333333 ) ) )
{
  cout << "333333 gelöscht; Studentenliste=" << sl << endl;
}
else
{
  cout << "die 333333 kann nicht gelöscht werden!" << endl;
}

// Die 876543 löschen:
if( sl.entf_wert( Student( "", "", 876543 ) ) )
{
  cout << "876543 gelöscht; Studentenliste=" << sl << endl;
}
```

```
    else
    {
      cout << "der 876543 kann nicht gelöscht werden!" << endl;
    }
    // Hinter Hein noch einen einfügen:
    if( sl.einf_HinterWert( Hein, Student( "S.", "Schmidt", 542366 ) ) )
    {
      cout << sl << endl;
    }
    else
    {
      cout << "hinter Hein kann nichts eingefügt werden!" << endl;
    }

    // Der zu Anfang eingefügte Hein wird jetzt geändert.
    // Dadurch ändert sich NICHT der Inhalt der Liste, da
    // Wertsemantik!
    Hein.setVorname( "Hinnerk" );
    cout << sl << endl;


    return 0;
} // main( int nargs, char **args )
```

Die letzten beiden Zeilen der Ausgabe sind identisch (im Gegensatz zur Java-Version; siehe **A.2.6.1 Java-Version** auf Seite 152), obwohl zwischenzeitlich der vorher eingefügte `Hein` geändert wurde; also hat dieser Container eine *Wertsemantik*:

```
Liste:  El.34,  El.45,  El.112,  El.123.
45 ist enthalten
123 ist enthalten
die 30 kann nicht gelöscht werden!
Liste:  El.34,  El.45,  El.112.
Liste:  El.34,  El.45.
Liste:  El.34,  El.45,  El.50.
Liste:  El.34,  El.50.
Liste:  El.30,  El.34,  El.50.
Liste:  El.30,  El.34,  El.48,  El.50.
vor 10 kann nichts eingefügt werden!
Studentenliste=Liste:  El.(Hein,Blöd,987654),  El.(K.,Meier,123456),  El.(E.,Steinberger,876543).
987654 ist so: (Hein,Blöd,987654)
333333 ist nicht enthalten
die 333333 kann nicht gelöscht werden!
876543 gelöscht; Studentenliste=Liste:  El.(Hein,Blöd,987654),  El.(K.,Meier,123456).
Liste:  El.(Hein,Blöd,987654),  El.(S.,Schmidt,542366),  El.(K.,Meier,123456).
Liste:  El.(Hein,Blöd,987654),  El.(S.,Schmidt,542366),  El.(K.,Meier,123456).
```

A.2.7 Geschlossenes Hashing

A.2.7.1 Java-Version

Für die Testdaten wird die Klasse `Student` aus **6.1.5 Java-Version** auf Seite 88 wiederverwendet, allerdings erweitert um eine Hashfunktion.

Die Hashfunktion ist in der Schnittstelle `Hashfaehig` deklariert:

```
// Time-stamp: "(23.05.02 20:35) Hashfaehig.java [Klaus@Wachtler.de]"
//
// Interface für eine Klasse, die in eine Hashtabelle eingefügt werden kann
// für die Übungsaufgabe "Hashfunktion".
// Eine Implementation dieser Schnittstelle muß aus ihren
// sonstigen Daten einen Hashwert bilden, und als Rückgabewert
// von hashfunktion() liefern.
// Dieser Wert kann dann als Index in die Hashtabelle verwendet werden.

interface Hashfaehig
{
    int hashfunktion();
}
```

(Anmerkung: Ein Java-`Object` besitzt für Hashing bereits eine Methode namens `hashCode()`, die man hier verwenden könnte. Damit wäre das Interface `Hashfaehig` überflüssig und in der Klasse `Student` wäre lediglich die Methode `hashfunktion()` durch `hashCode()` zu ersetzen. Dies wird hier aber nicht genutzt, um unabhängig von existierenden Bibliotheken den Mechanismus möglichst klar darzustellen.)

Die Klasse für einen Studenten sieht damit so aus:

```
// Time-stamp: "(23.05.02 19:58) Student.java [Klaus@Wachtler.de]"
//
// Testklasse zum Einfügen in eine Hashtabelle

import java.util.*;

public class Student implements Hashfaehig
{
    private String   VorName;
    private String   NachName;
    public  int      Matrikelnummer;

    public Student()
```

```
    {
        VorName = NachName = "?";
        Matrikelnummer = -1;
    }

    public Student( String v, String n, int m )
    {
        VorName = v;
        NachName = n;
        Matrikelnummer = m;
    }

    public String toString()
    {
        return "<" + VorName + " "
            + NachName + " "
            + Matrikelnummer  + ">";
    }

    // vergleicht zwei Studenten auf Gleichheit
    public boolean equals( Object s )
    {
        return Matrikelnummer==((Student)s).Matrikelnummer;
    }

    // Implementation von Hashfaehig:
    // Weil der Suchschlüssel (die Matrikelnummer) bereits ein geeigneter
    // Hashwert ist, wird einfach diese geliefert; eine
    // Modulobildung zum Begrenzen auf die Tabellengröße wird
    // beim Aufrufer erledigt.
    public int hashfunktion()
    {
        return Matrikelnummer;
    }

} // class Student
```

Die Hashtabelle mit linearem Sondieren ist generisch, kann also nicht nur diesen einen Datentypen verwalten, sondern für jeden beliebigen (wichtig ist nur, daß die Methode `equals()` und die Schnittstelle `Hashfaehig` implementiert sind):

```
// Time-stamp: "06.05.03 18:22 Hash.java klaus@wachtler.de"
//
// Hash kann eine Hashtabelle von fast beliebigen Daten verwalten.
// Der zugrundeliegende Typ muß aber folgende Bedingungen erfüllen:
//  - die Methode equals() muß so implementiert sein, daß zwei
//    Objekte anhand des gewünschten Suchschlüssels als gleich oder
```

```
//    ungleich erkannt werden, und
//  - das Interface Hashfaehig muß implementiert sein.
//
// Die Tabellengröße wird beim Erzeugen der Hashtabelle im Konstruktor
// angegeben, und kann im Nachhinein nicht mehr verändert werden.
//
// Um zwischen noch freien, belegten und gelöschten Elementen zu
// unterscheiden, wird folgende Schweinerei gemacht:
//  - ein noch nie benutzter Platz in der Tabelle ist null.
//  - ein belegter Platz ist vom Typ Hashfaehig (also von einem
//    benutzerdefinierten Datentyp, der das Interface Hashfaehig
//    implementiert)
//  - ein früher benutzter, aber bereits wieder gelöschter Eintrag
//    wird auf ein Objekt vom Typ Geloescht gesetzt, und ist somit
//    als "bereits gelöscht" erkennbar.
// Nicht schön, aber praktisch.

import java.util.*;
import java.lang.Class;

public class Hash
{
    static class Geloescht implements Hashfaehig
    {
        public int hashfunktion() { return 0; } // egal, was
    }

    private Hashfaehig[]  Tabelle = null;
    private int           Tabellenlaenge = 0;

    public Hash( int Tabellengroesse )
    {
        Tabelle = new Hashfaehig[Tabellengroesse];
        Tabellenlaenge = Tabellengroesse;
    }

    // Beim Einfügen wird nach dem nächsten freien oder gelöschten
    // Eintrag gesucht; falls dabei bereits ein gültiger Eintrag mit
    // demselben Hashwert gefunden wird, dann wird der überschrieben.
    // Wenn die Tabelle bereits restlos gefüllt ist, dann wird
    // false geliefert; sonst true.
    public boolean insert( Hashfaehig neu )
    {
        // Hier sollte der neue Eintrag landen, falls möglich:
        int wunschindex = neu.hashfunktion() % Tabellenlaenge;

        // Da lineares Sondieren gefordert ist, wird ab dem wunschindex
        // jeder Tabelleneintrag geprüft, bis ein geeigneter Platz
        // gefunden wird:
        for( int i=0; i<Tabellenlaenge; i++ )
        {
```

```
        int index = (wunschindex+i)%Tabellenlaenge;
        if( Tabelle[index]==null                        // unbenutzt?
            ||
            Tabelle[index].getClass()==Geloescht.class // gelöscht?
            ||
            Tabelle[index].equals( neu )               // vorhanden?
            )
        {
            Tabelle[index] = neu;
            return true;
        }
    }

    // Alle Einträge betrachtet, und keinen freien Platz gefunden?
    // Dann ist die Tabelle schon voll!
    return false;
}

// Beim Suchen mit get() wird solange gesucht, bis der Suchschlüssel
// gefunden ist, oder ein unbenutzter Eintrag erreicht ist;
// gelöschte Einträge werden übergangen.
public Object get( Hashfaehig key )
{
    // Hier sollte der gesuchte Eintrag stehen, falls er
    // ohne Kollision eingefügt wurde:
    int wunschindex = key.hashfunktion() % Tabellenlaenge;

    // Da lineares Sondieren gefordert ist, wird ab dem wunschindex
    // jeder Tabelleneintrag geprüft, bis der Schlüssel gefunden
    // ist, oder ein unbenutzter Eintrag erreicht ist:
    for( int i=0; i<Tabellenlaenge; i++ )
    {
        int index = (wunschindex+i)%Tabellenlaenge;
        if( Tabelle[index]==null )                      // unbenutzt?
        {
            return null; // nichts gefunden!
        }



        if( Tabelle[index].getClass()!=Geloescht.class // nicht gelöscht?
            &&
            Tabelle[index].equals( key )               // gleich?
            )
        {
            return Tabelle[index];
        }
    }

    // Alle Einträge betrachtet, und nichts gefunden?
    return null;
```

```
    }

    // Zum Löschen mit delete() wird solange gesucht, bis der Suchschlüssel
    // gefunden ist, oder ein unbenutzter Eintrag erreicht ist;
    // andere bereits gelöschte Einträge werden übergangen.
    public boolean delete( Hashfaehig key )
    {
        // Hier sollte der gesuchte Eintrag stehen, falls er
        // ohne Kollision eingefügt wurde:
        int wunschindex = key.hashfunktion() % Tabellenlaenge;

        // Da lineares Sondieren gefordert ist, wird ab dem wunschindex
        // jeder Tabelleneintrag geprüft, bis der Schlüssel gefunden
        // ist, oder ein unbenutzter Eintrag erreicht ist:
        for( int i=0; i<Tabellenlaenge; i++ )
        {
            int index = (wunschindex+i)%Tabellenlaenge;
            if( Tabelle[index]==null )                      // unbenutzt?
            {
                return false; // nichts gefunden!
            }



            if( Tabelle[index].getClass()!=Geloescht.class // nicht gelöscht?
                &&
                Tabelle[index].equals( key )               // gleich?
                )
            {
                // zu löschenden Eintrag gefunden!
                Tabelle[index] = new Geloescht(); // als gelöscht markieren
                return true;
            }
        }

        // Alle Einträge betrachtet, und nichts gefunden?
        return false;
    }

} // public class Hash
```

Das Testprogramm legt ein paar Studenten in der Tabelle ab, sucht etwas, und löscht dann wieder welche:

```
// Time-stamp: "06.05.03 18:16 TestHash.java klaus@wachtler.de"
//
// javac TestHash.java Hashfaehig.java Hash.java Student.java
// java TestHash
```

```
import java.util.*;

public class TestHash
{
    public static void main( String args[] )
    {
        // eine kleine Tabelle:
        Hash  Studententabelle = new Hash( 10 );

        // etwas einfügen:
        Studententabelle.insert( new Student( "Kurt",
                                              "Warsteiner",
                                              1234567  // hash=7
                                              )
                                 );
        Studententabelle.insert( new Student( "Fritz",
                                              "Flens",
                                              2345678  // hash=8
                                              )
                                 );
        Studententabelle.insert( new Student( "Silke",
                                              "Amaretto",
                                              3456789  // hash=9
                                              )
                                 );
        Studententabelle.insert( new Student( "Marie",
                                              "Huana",
                                              3456787  // hash=7
                                              )
                                 );
        // Anm.: weil die Tabellenplätze 7, 8, und 9 bereits belegt
        // sind, wird der letzte eingefügte Student statt bei 7
        // an der Stelle 0 eingetragen.

        // nach Studenten suchen:
        Student[] key = new Student[5];
        key[0] = new Student( "", "", 1234567 ); // vorhanden
        key[1] = new Student( "", "", 2345678 ); // vorhanden
        key[2] = new Student( "", "", 3456789 ); // vorhanden
        key[3] = new Student( "", "", 3456787 ); // vorhanden
        key[4] = new Student( "", "", 4567812 ); // nicht vorhanden

        for( int i=0; i<5; i++ )
        {
            Student gefunden;
            gefunden = (Student)( Studententabelle.get( key[i] ) );
            if( gefunden==null )
            {
                System.out.println( "Student "
                                    + key[i]
                                    + " nicht gefunden"
```

```
                            );
            }
            else
            {
                System.out.println( "gefunden: " + gefunden );
            }
        }

        System.out.println( "\n" );

        // 2 löschen:
        Studententabelle.delete( key[1] );
        System.out.println( "gelöscht: " + key[1] );
        Studententabelle.delete( key[2] );
        System.out.println( "gelöscht: " + key[2] );

        System.out.println( "\n" );

        // und wieder suchen:
        for( int i=0; i<5; i++ )
        {
            Student gefunden;
            gefunden = (Student)( Studententabelle.get( key[i] ) );
            if( gefunden==null )
            {
                System.out.println( "Student "
                                    + key[i]
                                    + " nicht gefunden"
                                    );
            }
            else
            {
                System.out.println( "gefunden: " + gefunden );
            }
        }

    }

} // public class TestHash
```

A.2.7.2 C++-Version

Die Klasse `Student` ist aus **6.1.4 C++-Version** auf Seite 84 übernommen; auch hier muß man für die Hashfunktion sorgen, sowie für den Vergleich auf Gleichheit (hier mit dem `operator==()`):

```
// Time-stamp: "11.05.03 19:09 Student.h klaus@wachtler.de"
//
// Beispieldatentyp für Hashtabellen.

#ifndef _STUDENT_H_
#define _STUDENT_H_

class Student
{
public:

  // Konstruktor
  Student( string v = "", string n = "", int m = -1 )
    : Vorname(v ),
      Nachname( n ),
      Matrikelnummer( m )
  {
  }

  // copy-Konstruktor
  Student( const Student &rechteSeite )
    : Vorname( rechteSeite.Vorname ),
      Nachname( rechteSeite.Nachname ),
      Matrikelnummer( rechteSeite.Matrikelnummer )
  {
  }

  // Destruktor
  virtual ~Student()
  {
  }

  // Zuweisung Student = Student
  Student & operator=( const Student &rechteSeite )
  {
    Vorname = rechteSeite.Vorname;
    Nachname = rechteSeite.Nachname;
    Matrikelnummer = rechteSeite.Matrikelnummer;

    return *this;
  }

  // Ausgabe auf einen ostream
  virtual void streamout( ostream & s ) const
```

```
  {
    s << Vorname;
    s << ",";
    s << Nachname;
    s << ",";
    s << Matrikelnummer;
  }

  // Kopie von sich selbst allokieren
  virtual Student &clone() const
  {
    return *(new Student(*this));
  }

  bool operator==( const Student &rS ) const
  {
    return Matrikelnummer==rS.Matrikelnummer;
  }

  // Die Hashfunktion liefert einfach die Matrikelnummer;
  // wenn der Aufrufer diesen Wert modulo Tabellengröße rechnet,
  // sollte ein vernünftiger Tabellenplatz herauskommen:
  size_t hashfunktion() const
  {
    return Matrikelnummer;
  }

private:

  // Nutzdaten:
  string Vorname;
  string Nachname;
  int    Matrikelnummer;


}; // class Student...

// ggf. nach Student.cpp kopieren (und static entfernen):
static ostream &operator<<( ostream &s, const Student &obj )
{
  obj.streamout( s );
  return s;
}

#endif /* _STUDENT_H_ */
```

Die C++-Lösung führt dieselben Operationen durch wie die Java-Version. Allerdings ist die Implementation ganz anders: durch die Nutzung von `templates` braucht man weniger Aufwand mit der Verwaltung von gelöschten Elementen treiben.

```
// Time-stamp: "11.05.03 19:28 hash.h klaus@wachtler.de"
//
// C++-Lösung der Übungsaufgabe "Hashtabelle".
//
// Die verwalteten Daten benötigen eine Methode int hashfunktion(),
// die eine ganze Zahl als Tabellenindex liefert, sowie den
// operator==().
// Der von hashfunktion() gelieferte Wert darf beliebig groß sein
// (oder auch negativ); eine Reduzierung auf die Tabellenlänge wird
// automatisch vorgenommen.
//
//
// Änderungen:
//
// 04.05.2003 kw   im Destruktor ~Hash() das delete zu delete[] korrigiert
// 11.05.2003 kw   HashEintrag zu Unterklasse von Hash gemacht;
//                 aufgeteilt in hash.h und testhash.cpp

#ifndef _HASH_H_
#define _HASH_H_


#include <iostream>
#include <string>

using namespace std;

template<class T> class Hash
{
private:

  // Ein Eintrag der Tabelle besteht aus einem Zeiger auf die Nutzdaten,
  // sowie einem flag, ob der aktuelle Eintrag vielleicht gelöscht
  // wurde.
  // Mögliche Einträge:
  //  nutzdaten  |  geloescht |  Bedeutung
  //  -----------+------------+-------------------------------------
  //  NULL       |  false     | Eintrag noch unbenutzt
  //  -----------+------------+-------------------------------------
  //  !=NULL     |  egal      | Eintrag belegt
  //  -----------+------------+-------------------------------------
  //  NULL       |  true      | Eintrag war belegt, und ist gelöscht
  //  -----------+------------+-------------------------------------
  //

  class HashEintrag
  {
  public:

    T    *nutzdaten;
    bool  geloescht;
```

```
    // Konstruktor
    HashEintrag()
    {
      nutzdaten = NULL;
      geloescht = false;
    }

    // Destruktor
    ~HashEintrag()
    {
      delete nutzdaten;
    }
  }; // class HashEintrag


  // Die eigentliche Tabelle ist ein Zeiger auf ein
  // HashEintrag-Feld, das beim Erzeugen mit der richtigen
  // Anzahl allokiert wird:
  HashEintrag       *Tabelle;
  size_t             Tabellenlaenge;

 private:

  // Konstruktor ohne Parameter: private, damit niemand eine
  // leere Tabelle erzeugen kann.
  Hash()
  {
    Tabelle = NULL;
  }

 public:

  // Konstruktor mit Anzahl Einträge
  Hash( size_t Tabellengroesse )
  {
    Tabelle = new HashEintrag[Tabellengroesse];
    Tabellenlaenge = Tabellengroesse;
  }

  // Destruktor
  ~Hash()
  {
    delete[] Tabelle; Tabellenlaenge = 0;
  }

  bool insert( const T &neu )
  {
    size_t wunschindex = neu.hashfunktion() % Tabellenlaenge;

    // Da lineares Sondieren gefordert ist, wird ab dem wunschindex
```

```
  // jeder Tabelleneintrag geprüft, bis ein geeigneter Platz
  // gefunden wird:
  for( unsigned int i=0; i<Tabellenlaenge; i++ )
  {
    size_t index = (wunschindex+i)%Tabellenlaenge;

    if( Tabelle[index].nutzdaten==NULL )
    {
      // Eintrag leer, als hier einfügen!
      // Da hier eine Kopie angelegt wird, hat dieser Container eine
      // Wertsemantik.
      Tabelle[index].nutzdaten = new T( neu );
      return true;
    }
    else if( *(Tabelle[index].nutzdaten)==neu )
    {
      // Eintrag mit diesem Schlüssel schon vorhanden,
      // also überschreiben:
      *(Tabelle[index].nutzdaten) = neu;
      return true;
    }
  }

  // Alle Einträge betrachtet, und keinen freien Platz gefunden?
  // Dann ist die Tabelle schon voll!
  return false;

}

// Die Suche mit get() liefert einen Zeiger auf das gefundene Objekt,
// oder NULL, wenn nichts gefunden wird.
const T *get( const T &key )
{
  size_t wunschindex = key.hashfunktion() % Tabellenlaenge;

  // Da lineares Sondieren gefordert ist, wird ab dem wunschindex
  // jeder Tabelleneintrag geprüft, bis das gesuchte Element
  // gefunden wird:
  for( unsigned int i=0; i<Tabellenlaenge; i++ )
  {
    size_t index = (wunschindex+i)%Tabellenlaenge;

    if( Tabelle[index].nutzdaten==NULL  // nicht belegt!
        &&
        !Tabelle[index].geloescht       // nicht gelöscht
        )
    {
      // dies ist ein jungfräulicher Eintrag,
      // die Suche kann abgebrochen werden!
      return NULL;
    }
```

```
    else if( Tabelle[index].nutzdaten!=NULL   // belegt!
             &&
             *(Tabelle[index].nutzdaten)==key // Treffer!
             )
    {
      return Tabelle[index].nutzdaten;
    }
  }


  // Alle Einträge betrachtet, und keinen freien Platz gefunden?
  // Dann ist die Tabelle schon voll, und nichts gefunden!
  return NULL;
}

// Beim Löschen mit delete() wird false geliefert, wenn nichts gelöscht
// werden kann.
bool del( const T &key )
{
  size_t wunschindex = key.hashfunktion() % Tabellenlaenge;

  // Da lineares Sondieren gefordert ist, wird ab dem wunschindex
  // jeder Tabelleneintrag geprüft, bis das gesuchte Element
  // gefunden wird:
  for( unsigned int i=0; i<Tabellenlaenge; i++ )
  {
    size_t index = (wunschindex+i)%Tabellenlaenge;

    if( Tabelle[index].nutzdaten==NULL  // nicht belegt!
        &&
        !Tabelle[index].geloescht       // nicht gelöscht
        )
    {
      // dies ist ein jungfräulicher Eintrag,
      // die Suche kann abgebrochen werden!
      return false;
    }
    else if( Tabelle[index].nutzdaten!=NULL   // belegt!
             &&
             *(Tabelle[index].nutzdaten)==key // Treffer!
             )
    {
      delete Tabelle[index].nutzdaten; // Speicher freigeben
      Tabelle[index].nutzdaten = NULL; // als "unbenutzt" markieren
      Tabelle[index].geloescht = true; // als "gelöscht" markieren
    }
  }

  // Alle Einträge betrachtet, und keinen freien Platz gefunden?
  // Dann ist die Tabelle schon voll, und nichts gefunden!
  return false;
}
```

```
}; // class Hash


#endif /* _HASH_H_ */
```

Das Testprogramm hierzu:

```
// Time-stamp: "11.05.03 19:13 testhash.cpp klaus@wachtler.de"
//
// Testprogramm zur C++-Lösung der Übungsaufgabe "Hashtabelle".
//
// ANSI-C++
//
// Linux:    g++ -Wall testhash.cpp && ./a.out
//
// VC++ 6.0: cl /Zi /GX testhash.cpp
//           .\testhash
//
// 04.05.2003 kw   im Destruktor ~Hash() das delete zu delete[] korrigiert
// 11.05.2003 kw   HashEintrag zu Unterklasse von Hash gemacht;
//                 aufgeteilt in hash.h und testhash.cpp

#include <iostream>
#include <string>


//////////////////////////////////////////////////////////////////////
//
// Testprogramm:

#include "hash.h"

#include "Student.h"

using namespace std;



int main( int nargs, char **args )
{
  // eine kleine Tabelle:
  Hash<Student>  Studententabelle( 10 );

  // etwas einfügen:
```

```
Studententabelle.insert( Student( "Kurt",
                                  "Warsteiner",
                                  1234567  // hash=7
                                  )
                         );
Studententabelle.insert( Student( "Fritz",
                                  "Flens",
                                  2345678  // hash=8
                                  )
                         );
Studententabelle.insert( Student( "Silke",
                                  "Amaretto",
                                  3456789  // hash=9
                                  )
                         );
Studententabelle.insert( Student( "Marie",
                                  "Huana",
                                  3456787  // hash=7
                                  )
                         );
// Anm.: weil die Tabellenplätze 7, 8, und 9 bereits belegt
// sind, wird der letzte eingefügte Student statt bei 7
// an der Stelle 0 eingetragen.

// nach Studenten suchen:
Student key[5];
key[0] = Student( "", "", 1234567 ); // vorhanden
key[1] = Student( "", "", 2345678 ); // vorhanden
key[2] = Student( "", "", 3456789 ); // vorhanden
key[3] = Student( "", "", 3456787 ); // vorhanden
key[4] = Student( "", "", 4567812 ); // nicht vorhanden

int i;
for( i=0; i<5; i++ )
{
  const Student *gefunden;
  gefunden = Studententabelle.get( key[i] );
  if( gefunden==NULL )
  {
    cout << "Student " << key[i] << " nicht gefunden" << endl;
  }
  else
  {
    cout << "gefunden: " << *gefunden << endl;
  }
}

cout << endl;

// 2 löschen:
Studententabelle.del( key[1] );
```

```
  cout << "gelöscht: " << key[1] << endl;
  Studententabelle.del( key[2] );
  cout << "gelöscht: " << key[2] << endl;

  cout << endl;

  // und wieder suchen:
  for( i=0; i<5; i++ )
  {
    const Student *gefunden;
    gefunden = Studententabelle.get( key[i] );
    if( gefunden==NULL )
    {
      cout << "Student " << key[i] << " nicht gefunden" << endl;
    }
    else
    {
      cout << "gefunden: " << *gefunden << endl;
    }
  }

  return 0;
} // main( int nargs, char **args )
```

A.2.8 Offenes Hashing

A.2.8.1 Java-Version

Für die Testdaten wird die Klasse `Student` aus **6.1.5 Java-Version** auf Seite 88 wiederverwendet, allerdings erweitert um eine Hashfunktion.

Siehe dazu **A.2.7.1 Java-Version** auf Seite 176.

Die Klasse ist generisch, kann also nicht nur diesen einen Datentypen verwalten, sondern jeden beliebigen (wichtig ist nur, daß die Methode `equals()` und die Schnittstelle `Hashfaehig` implementiert sind):

```
// Time-stamp: "11.05.03 08:06 oHash.java klaus@wachtler.de"
//
// oHash kann eine Hashtabelle von fast beliebigen Daten verwalten.
// Der zugrundeliegende Typ muß aber folgende Bedingungen erfüllen:
//  - die Methode equals() muß so implementiert sein, daß zwei
//    Objekte anhand des gewünschten Suchschlüssels als gleich oder
//    ungleich erkannt werden, und
//  - das Interface Hashfaehig muß implementiert sein.
//
// Die Tabellengröße wird beim Erzeugen der Hashtabelle im Konstruktor
// angegeben, und kann im Nachhinein nicht mehr verändert werden.


import java.util.*;
import java.lang.Class;

public class oHash
{
    private einfacheListeGenerisch[]  Tabelle = null;
    private int                       Tabellenlaenge = 0;

    public oHash( int Tabellengroesse )
    {
        Tabelle = new einfacheListeGenerisch[Tabellengroesse];
        Tabellenlaenge = Tabellengroesse;
        for( int i=0; i<Tabellengroesse; i++ )
        {
            Tabelle[i] = new einfacheListeGenerisch();
        }
    }

    // Zum Einfügen wird mit der Hashfunktion der zugehörige Platz
    // bestimmt, und das Objekt an die dort stehende Liste angefügt.
    // Der Einfachheit halber wird nicht geprüft, ob das Element
```

```
    // bereits in der Liste steht.
    // Dadurch wäre mehrfaches Einfügen eines Objekts möglich.
    // Nach dem Berechnen der Tabellenposition wird die gesamte Arbeit
    // der entsprechenden Liste überlassen.
    public void insert( Hashfaehig neu )
    {
        Tabelle[neu.hashfunktion() % Tabellenlaenge].einf_vorn( neu );
    }

    // Suchen eines Elementes: anhand des Hashwertes die richtige Liste
    // finden, und dort suchen lassen.
    public Object get( Hashfaehig key )
    {
        return Tabelle[key.hashfunktion() % Tabellenlaenge].sucheWert( key );
    }

    // Löschen eines Elementes: anhand des Hashwertes die richtige Liste
    // finden, und dort löschen lassen.
    public boolean delete( Hashfaehig key )
    {
        return Tabelle[key.hashfunktion() % Tabellenlaenge].entf_wert( key );
    }

} // public class oHash
```

Das Testprogramm legt ein paar Studenten in der Tabelle ab, sucht etwas, und löscht dann wieder welche:

```
// Time-stamp: "11.05.03 19:44 TestoHash.java klaus@wachtler.de"
//
// javac TestoHash.java Hashfaehig.java oHash.java Student.java \
//       einfacheListeGenerisch.java
// java TestoHash

import java.util.*;

public class TestoHash
{
    public static void main( String args[] )
    {
        // eine kleine Tabelle:
        oHash  Studententabelle = new oHash( 10 );

        // etwas einfügen:
        Studententabelle.insert( new Student( "Kurt",
                                              "Warsteiner",
                                              1234567  // hash=7
                                              )
                                 );
```

```
Studententabelle.insert( new Student( "Fritz",
                                      "Flens",
                                      2345678  // hash=8
                                      )
                         );
Studententabelle.insert( new Student( "Silke",
                                      "Amaretto",
                                      3456789  // hash=9
                                      )
                         );
Studententabelle.insert( new Student( "Marie",
                                      "Huana",
                                      3456787  // hash=7
                                      )
                         );

// nach Studenten suchen:
Student[] key = new Student[5];
key[0] = new Student( "", "", 1234567 ); // vorhanden
key[1] = new Student( "", "", 2345678 ); // vorhanden
key[2] = new Student( "", "", 3456789 ); // vorhanden
key[3] = new Student( "", "", 3456787 ); // vorhanden
key[4] = new Student( "", "", 4567812 ); // nicht vorhanden

for( int i=0; i<5; i++ )
{
    Object gefunden;
    gefunden = ( Studententabelle.get( key[i] ) );
    if( gefunden==null )
    {
        System.out.println( "Student "
                            + key[i]
                            + " nicht gefunden"
                            );
    }
    else
    {
        System.out.println( "gefunden: " + (Student)gefunden );
    }
}

System.out.println( "\n" );

// 2 löschen:
Studententabelle.delete( key[1] );
System.out.println( "gelöscht: " + key[1] );
Studententabelle.delete( key[2] );
System.out.println( "gelöscht: " + key[2] );

System.out.println( "\n" );
```

```
        // und wieder suchen:
        for( int i=0; i<5; i++ )
        {
            Student gefunden;
            gefunden = (Student)( Studententabelle.get( key[i] ) );
            if( gefunden==null )
            {
                System.out.println( "Student "
                                    + key[i]
                                    + " nicht gefunden"
                                    );
            }
            else
            {
                System.out.println( "gefunden: " + gefunden );
            }
        }

    }

} // public class TestoHash
```

A.2.8.2 C++-Version

Die Klasse `Student` für die Testdaten ist wie beim geschlossenen Hashing aus **6.1.4 C++-Version** auf Seite 84 übernommen; auch hier muß man für die Hashfunktion sorgen, sowie für den Vergleich auf Gleichheit (hier mit dem `operator==()`). Siehe dazu die Lösung zum geschlossenen Hashing in **A.2.7.2 C++-Version** auf Seite 183.

Anstelle des linearen Sondierens wird beim geschlossenen Hashing fast die gesamte Arbeit auf die Listen abgewälzt; dadurch wird die Lösung recht kompakt und übersichtlich:

```
// Time-stamp: "11.05.03 19:37 ohash.h klaus@wachtler.de"
//
// C++-Lösung der Übungsaufgabe "Hashtabelle".
//
// Die verwalteten Daten benötigen eine Methode int hashfunktion(),
// die eine ganze Zahl als Tabellenindex liefert, sowie den
// operator==().
// Der von hashfunktion() gelieferte Wert darf beliebig groß sein
// (oder auch negativ); eine Reduzierung auf die Tabellenlänge wird
// automatisch vorgenommen.
//
//
// Änderungen:
//
// 11.05.2003 kw   aus hash.h übernommen und auf offenes Hashing
//                 umgebaut

#ifndef _OHASH_H_
#define _OHASH_H_


#include <iostream>
#include <string>

#include "einfacheListeGenerisch.h"


using namespace std;

template<class T> class oHash
{
 private:

  // Die eigentliche Tabelle ist ein Zeiger auf ein
  // Feld von Listen von T, das beim Erzeugen mit der richtigen
  // Anzahl allokiert wird:
  einfacheListeGenerisch<T>       *Tabelle;
```

```
  size_t                           Tabellenlaenge;

 private:

  // Konstruktor ohne Parameter: private, damit niemand eine
  // leere Tabelle erzeugen kann.
  oHash()
  {
    Tabelle = NULL;
  }

 public:

  // Konstruktor mit Anzahl Einträge
  oHash( size_t Tabellengroesse )
  {
    Tabelle = new einfacheListeGenerisch<T>[Tabellengroesse];
    Tabellenlaenge = Tabellengroesse;
  }

  // Destruktor
  ~oHash()
  {
    delete[] Tabelle; Tabellenlaenge = 0;
  }

  // Da die Listen beim offenen Hashing theoretisch beliebig lang
  // werden können, gibt es keinen Fehler beim Einfügen.
  // Deshalb gibt insert() nichts zurück.
  void insert( const T &neu )
  {
    Tabelle[neu.hashfunktion() % Tabellenlaenge].einf_vorn( neu );
  }

  // Die Suche mit get() liefert einen Zeiger auf das gefundene Objekt,
  // oder NULL, wenn nichts gefunden wird.
  const T *get( const T &key )
  {
    return Tabelle[key.hashfunktion() % Tabellenlaenge].sucheWert( key );
  }

  // Beim Löschen mit delete() wird false geliefert, wenn nichts gelöscht
  // werden kann.
  bool del( const T &key )
  {
    return Tabelle[key.hashfunktion() % Tabellenlaenge].entf_wert( key );
  }

}; // class oHash

#endif /* _OHASH_H_ */
```

Das Testprogramm entspricht **A.2.7.2 C++-Version** auf Seite 183:

```
// Time-stamp: "11.05.03 19:44 testohash.cpp klaus@wachtler.de"
//
// Testprogramm zur C++-Lösung der Übungsaufgabe "Offenes Hashing".
//
// ANSI-C++
//
// Linux:    g++ -Wall testohash.cpp && ./a.out
//
// VC++ 6.0: cl /Zi /GX testohash.cpp
//           .\testohash
//
// 11.05.2003 kw   aus testhash.cpp übernommen und auf offenes Hashing
//                 umgebaut

#include <iostream>
#include <string>


//////////////////////////////////////////////////////////////////////
//
// Testprogramm:

#include "ohash.h"

#include "Student.h"

using namespace std;



int main( int nargs, char **args )
{
  // eine kleine Tabelle:
  oHash<Student>  Studententabelle( 10 );

  // etwas einfügen:
  Studententabelle.insert( Student( "Kurt",
                                    "Warsteiner",
                                    1234567  // hash=7
                                    )
                         );
  Studententabelle.insert( Student( "Fritz",
                                    "Flens",
                                    2345678  // hash=8
                                    )
```

```
                         );
Studententabelle.insert( Student( "Silke",
                                  "Amaretto",
                                  3456789  // hash=9
                                  )
                         );
Studententabelle.insert( Student( "Marie",
                                  "Huana",
                                  3456787  // hash=7
                                  )
                         );

// nach Studenten suchen:
Student key[5];
key[0] = Student( "", "", 1234567 ); // vorhanden
key[1] = Student( "", "", 2345678 ); // vorhanden
key[2] = Student( "", "", 3456789 ); // vorhanden
key[3] = Student( "", "", 3456787 ); // vorhanden
key[4] = Student( "", "", 4567812 ); // nicht vorhanden

int i;
for( i=0; i<5; i++ )
{
  const Student *gefunden;
  gefunden = Studententabelle.get( key[i] );
  if( gefunden==NULL )
  {
    cout << "Student " << key[i] << " nicht gefunden" << endl;
  }
  else
  {
    cout << "gefunden: " << *gefunden << endl;
  }
}

cout << endl;

// 2 löschen:
Studententabelle.del( key[1] );
cout << "gelöscht: " << key[1] << endl;
Studententabelle.del( key[2] );
cout << "gelöscht: " << key[2] << endl;

cout << endl;

// und wieder suchen:
for( i=0; i<5; i++ )
{
  const Student *gefunden;
  gefunden = Studententabelle.get( key[i] );
  if( gefunden==NULL )
```

```
    {
      cout << "Student " << key[i] << " nicht gefunden" << endl;
    }
    else
    {
      cout << "gefunden: " << *gefunden << endl;
    }
  }

  return 0;
} // main( int nargs, char **args )
```

A.2.9 Einfache Integration

Es werden drei verschiedene Lösungen vorgestellt, die sich in der eigentlichen Integration kaum unterscheiden. Allerdings ist die Trennung des Integrierens von der zu integrierenden Funktion jeweils anders gelöst:

- In **A.2.9.1 Java: Verwendung einer abstrakten Basisklasse** auf Seite 201 wird die Implementation der Integration in einer abstrakten Basisklasse definiert, und die zu integrierende Funktion in einer davon abgeleitete Klasse.
- In **A.2.9.2 Java: Verwendung eines funktionalen Objekts** auf Seite 204 wird ein *Interface* mit der zu integrierenden Funktion geschrieben. Der integrierenden Klasse wird ein solches Objekt übergeben, das dann bestimmt, was integriert werden soll.
- In **A.2.9.3 C++: Funktionszeiger** auf Seite 207 wird einfach ein Zeiger auf die zu integrierende Funktion übergeben: schnell zu schreiben, schnell in der Ausführung, und weit entfernt von der gängigen Vorstellung von objektorientiert.

(Anmerkung: allen drei Versionen gemeinsam ist die Tatsache, daß in der Schleife zum Aufsummieren der Teilflächen unnötigerweise eine Multiplikation mit der unveränderlichen Streifenbreite stattfindet. Das Programm ließe sich optimieren, indem in der Schleife nur die Funktionswerte addiert werden, und nach der Schleife dann mit der Intervalllänge multipliziert wird. Dies habe ich aus zwei Gründen unterlassen: zum einen ist die vorliegende Form näher an der Aufgabenstellung und damit vermutlich leichter zu verstehen; zum anderen kann zumindest ein guter C++-Compiler diese Optimierung auch selbständig durchführen.)

A.2.9.1 Java: Verwendung einer abstrakten Basisklasse

Hier wird in einer abstrakten Basisklasse `einfache_integration` die eigentliche Integration implementiert, aber die zu integrierende Funktion nur deklariert, also ihre Implementation weggelassen:

```
// Time-stamp: "27.05.02 23:11 einfache_integration.java klaus@lap2.wachtler.de"
//
// Lösung der Übungsaufgabe "Einfache Integration"
// (Variante mit einer abstrakten Basisklasse zum Integrieren, in
```

```
// deren Ableitungen dann die zu integrierende Funktion überschrieben
// wird)

abstract public class einfache_integration
{

  // zu integrierende Funktion; muß in einer abgeleiteten Klasse
  // überschrieben werden!
  abstract double f( double x );

  // Funktion, die eine Funktion f im Intervall [a,b] näherungsweise
  // integriert, indem das Intervall in gleichbreite Streifen etwa
  // der Breite dx zerlegt wird,
  // für jeden Streifen eine Höhe bestimmt wird (als Funktionswert in
  // der Streifenmitte), und die Flächen aller Streifen aufsummiert
  // werden.
  double integriere( double a, double b, double dx )
  {
    // Wenn a größer ist als b, dann vertauschen:
    if( a>b )
    {
      double tmp = a; a = b; b = tmp;
    }

    // Anzahl der Streifen im Intervall:
    int    AnzahlStreifen = (int)( ( b - a )/dx );

    // Dadurch, daß die Anzahl der Streifen ganzzahlig ist, kann es
    // vorkommen, daß AnzahlStreifen*dx nicht genau die Intervalllänge
    // ergibt, sondern etwas weniger. Deshalb dx anpassen:
    dx = ( b - a )/AnzahlStreifen;

    double Integral = 0.0;        // bisherige Summe
    double xMitte   = a + 0.5*dx; // x in der Mitte des aktuellen Streifens

    for( int i=0; i<AnzahlStreifen ; i++, xMitte+=dx )
    {
      Integral += dx*f( xMitte );
    }

    return Integral;
  }

}
```

Von einer solchen abstrakten Klasse kann kein Objekt erzeugt werden, weil die Implementation der Methode `f( double )` noch fehlt.

Vielmehr muß der Aufrufer erst noch für jede zu integrierende Funktion eine Ableitung bilden, die dann in dieser Methode `f( double )` definiert, was integriert werden soll:

```
// Time-stamp: "(23.05.02 12:11) test_einf_integr.java [Klaus@Wachtler.de]"
//
// Testprogramm zur Lösung der Übungsaufgabe "Einfache Integration"
// (Variante mit einer abstrakten Basisklasse zum Integrieren, in
// deren Ableitungen dann die zu integrierende Funktion überschrieben
// wird)
//
// javac test_einf_integr.java einfache_integration.java
// java test_einf_integr

import java.lang.Math;

// Integrieren der Sinusfunktion:
class einfache_integration_sin extends einfache_integration
{
    double f( double x )
    {
        return Math.sin( x );
    }
}

// Integrieren von x^2:
class einfache_integration_xmalx extends einfache_integration
{
    double f( double x )
    {
        return x*x;
    }
}

public class test_einf_integr
{
    public static void main( String[] args )
    {
        // Ein Stückchen Sinus integrieren:
        einfache_integration_sin Isin
            = new einfache_integration_sin();
        double integral_sin = Isin.integriere( 0.0, 3.14159365, 0.1 );
        System.out.println( "Isin.integriere( 0.0, 3.14159365, 0.1 ) = "
                            +
                            integral_sin
                            );

        // Quadratfunktion integrieren:
        einfache_integration_xmalx Ixmalx
            = new einfache_integration_xmalx();
        double integral_xmalx = Ixmalx.integriere( 0.0, 10.0, 0.1 );
        System.out.println( "Ixmalx.integriere( 0.0, 10.0, 0.1 ) = "
                            +
```

```
                                integral_xmalx
                                );
    }

}
```

Diese Vorgehensweise wirkt ziemlich umständlich; vor allem wenn mehrere Kombinationen aus Integrationsverfahren und zu integrierenden Funktionen durchgespielt werden sollen.

A.2.9.2 Java: Verwendung eines funktionalen Objekts

Durch die Verwendung eines *funktionalen Objekts* (oder auch *Funktionsobjekts*) gelingt eine bessere Trennung des Integrationsverfahrens von der zu integrierenden Funktion.

Dazu ist zuerst eine Klasse oder eine Schnittstelle nötig, mit der die zu integrierende Funktion beschrieben werden kann:

```
// Time-stamp: "30.05.02 13:49 FunktObj.java klaus@lap2.wachtler.de"
//
// Interface eines funktionalen Objekts zur Lösung der
// Übungsaufgabe "Einfache Integration" und anderer ähnlicher Probleme.

interface FunktObj
{
    double f( double x );
}
```

Ein Objekt dieses Typs ist das sogenannte *funktionale Objekt*, und muß die Methode `double f( double x )` implementieren.

Das Integrationsverfahren muß die zu integrierende Funktion nicht kennen, sondern ruft einfach über das funktionale Objekt die darin enthaltene Methode auf:

```
// Time-stamp: "(23.05.02 14:58) einfache_integration.java [Klaus@Wachtler.de]"
//
// Lösung der Übungsaufgabe "Einfache Integration"
// (Variante mit der Übergabe eines funktionalen Objekts)

public class einfache_integration
{
  // Funktionales Objekt, dessen Methode double f( double x ) die
```

```
  // zu integrierende Funktion repräsentiert:
  FunktObj    funobj = null;


  // Dem Konstruktor wird ein funktionales Objekt übergeben, dessen
  // Methode f( double ) bestimmt, was integriert werden soll.
  einfache_integration( FunktObj f ) { funobj = f; }

  // Funktion, die eine Funktion f im Intervall [a,b] näherungsweise
  // integriert, indem das Intervall in gleichbreite Streifen etwa der
  // Breite dx zerlegt wird, für jeden Streifen eine Höhe bestimmt
  // wird (als Funktionswert in der Streifenmitte), und die Flächen
  // aller Streifen aufsummiert werden.
  double integriere( double a, double b, double dx )
  {
    // Wenn a größer ist als b, dann vertauschen:
    if( a>b )
    {
      double tmp = a; a = b; b = tmp;
    }

    // Anzahl der Streifen im Intervall:
    int    AnzahlStreifen = (int)( ( b - a )/dx );

    // Dadurch, daß die Anzahl der Streifen ganzzahlig ist, kann es
    // vorkommen, daß AnzahlStreifen*dx nicht genau die Intervalllänge
    // ergibt, sondern etwas weniger. Deshalb dx anpassen:
    dx = ( b - a )/AnzahlStreifen;

    double Integral = 0.0;        // bisherige Summe
    double xMitte   = a + 0.5*dx; // x in der Mitte des aktuellen Streifens

    for( int i=0; i<AnzahlStreifen ; i++, xMitte+=dx )
    {
      Integral += dx*funobj.f( xMitte );
    }

    return Integral;
  }

}
```

Erst beim Aufrufer werden das Integrationsverfahren und die zu integrierende Funktion (in Form des „funktionalen Objekts“) zusammengebracht:

```
// Time-stamp: "(23.05.02 16:31) test_einf_integr.java [Klaus@Wachtler.de]"
//
// Testprogramm zur Lösung der Übungsaufgabe "Einfache Integration"
// (Variante mit der Übergabe eines funktionalen Objekts)
//
```

```
// javac test_einf_integr.java FunktObj.java einfache_integration.java
// java test_einf_integr

import java.lang.Math;

// funktionales Objekt: Berechnet in f() die Sinusfunktion:
class Sinusfunktion implements FunktObj
{
    public double f( double x )
    {
        return Math.sin( x );
    }
}

// funktionales Objekt: Berechnet in f() die Funktion x^2:
class Quadratfunktion implements FunktObj
{
    public double f( double x )
    {
        return x*x;
    }
}

public class test_einf_integr
{
    public static void main( String[] args )
    {
        // Ein Stückchen Sinus integrieren:
        Sinusfunktion fSin = new Sinusfunktion();
        einfache_integration Isin
            = new einfache_integration( fSin );
        double integral_sin = Isin.integriere( 0.0, 3.14159365, 0.1 );
        System.out.println( "Isin.integriere( 0.0, 3.14159365, 0.1 ) = "
                            +
                            integral_sin
                            );

        // Quadratfunktion integrieren:
        Quadratfunktion fQuadrat = new Quadratfunktion();
        einfache_integration Ixmalx
            = new einfache_integration( fQuadrat );
        double integral_xmalx = Ixmalx.integriere( 0.0, 10.0, 0.1 );
        System.out.println( "Ixmalx.integriere( 0.0, 10.0, 0.1 ) = "
                            +
                            integral_xmalx
                            );
    }

}
```

A.2.9.3 C++: Funktionszeiger

Die beiden in Java beschriebenen Verfahren, das Integrieren von der zu integrierenden Funktion möglichst weit zu trennen, und zur eigentlichen Berechnung dann doch zusammenzuführen, sind in C++ auch möglich[2]. Sie erscheinen hier aber unnötig kompliziert: eigentlich geht es ja nur darum, zum Integrieren einmal die eine Funktion anzugeben und dann bei Bedarf eine andere. Das läßt sich in C und C++ viel einfacher bewerkstelligen, indem man einfach die zu integrierende Funktion (genauer gesagt: ihre Anfangsadresse im Speicher) als Parameter übergibt (siehe dazu auch [KW-C]):

```
// Time-stamp: "24.05.02 12:48 einfache_integration.cpp Administrator@AW41"
//
// Lösung der Übungsaufgabe "Einfache Integration"
//
// ANSI-C++
//
// Linux:    g++ -Wall einfache_integration.cpp && a.out
//
// VC++ 6.0: cl /Zi /GX einfache_integration.cpp
//           .\einfache_integration

#include <iostream>
using namespace std;

// Funktion, die eine Funktion f im Intervall [a,b] näherungsweise
// integriert, indem das Intervall in gleichbreite Streifen etwa der
// Breite dx zerlegt wird, für jeden Streifen eine Höhe bestimmt wird
// (als Funktionswert in der Streifenmitte), und die Flächen aller
// Streifen aufsummiert werden.
//
// Eingabeparameter:
// - double a:       erste Intervallgrenze
// - double b:       zweite Intervallgrenze
//                   (falls a>b ist, werden die beiden vertauscht)
// - double dx:      gewünschte Streifenbreite (wird intern so
//                   angepaßt, daß sich eine ganze Anzahl Streifen
//                   ergibt)
// - double( *f )
//    ( double x ):  f ist ein Zeiger auf eine Funktion, die
//                   einen double-Parameter x entgegennimmt,
//                   und einen double-Wert zurückliefert:
//                   das ist die zu integrierende Funktion
```

[2] C++ geht bei Funktionsobjekten sogar noch weiter, indem „echte“ funktionale Objekte möglich sind, die wie Funktionen verwendet werden können; also statt `funobj.f( xMitte )` beispielsweise könnte man auch `funobj( xMitte )` schreiben, wenn für die Klasse von `funobj` der `operator()` überladen ist; siehe [Strou IV] und [KW-C++].

```
//
// Rückgabewert:
// Näherung für das Integral
double integriere( double a,
                   double b,
                   double dx,
                   double( *f )( double x )
                   )
{
  // Wenn a größer ist als b, dann vertauschen:
  if( a>b )
  {
    double tmp = a; a = b; b = tmp;
  }

  // Anzahl der Streifen im Intervall:
  int    AnzahlStreifen = (int)( ( b - a )/dx );

  // Dadurch, daß die Anzahl der Streifen ganzzahlig ist, kann es
  // vorkommen, daß AnzahlStreifen*dx nicht genau die Intervalllänge
  // ergibt, sondern etwas weniger. Deshalb dx anpassen:
  dx = ( b - a )/AnzahlStreifen;

  double Integral = 0.0;        // bisherige Summe
  double xMitte   = a + 0.5*dx; // x in der Mitte des aktuellen Streifens

  // Jetzt kommt das Integrieren:
  for( int i=0; i<AnzahlStreifen ; i++, xMitte+=dx )
  {
    Integral += dx*f( xMitte );
  }

  return Integral;
}


//////////////////////////////////////////////////////////////////////
//
// Testprogramm:
//

// Quadratfunktion:
double xmalx( double x )
{
  return x*x;
}

#include <cmath> // für die Winkelfunktionen


#define PI 3.14159265
```

```
int main( int nargs, char **args )
{
  cout << "integriere( 0.0, PI, 0.1, sin )      = "
       << integriere( 0.0, PI, 0.1, sin )   // sin() aus math.h
       << endl;
  cout << "integriere( 0.0, PI/2, 0.1, cos )    = "
       << integriere( 0.0, PI/2, 0.1, cos ) // cos() aus math.h
       << endl;
  cout << "integriere( 0.0, 10.0, 0.1, xmalx )  = "
       << integriere( 0.0, 10.0, 0.1, xmalx )
       << endl;
  return 0;
} // main( int nargs, char **args )
```

Diese Vorgehensweise ist nicht nur schnell programmiert, sondern läuft auch unschlagbar schnell. Was will man mehr? Ach ja, es ist nicht so richtig objektorientiert…

A.2.10 Maximum einer Funktion

Sowohl die Java- als auch die C++-Lösung arbeiten nach folgendem Prinzip: beim Aufruf zur Suche der Nullstelle wird eine Funktion angegeben (in Java als Funktionsobjekt, in C++ als Zeiger auf eine Funktion), deren Nullstelle gesucht ist, weiterhin die Grenzen des zu durchsuchenden Intervalls, eine gewünschte Genauigkeit, und eine Anfangsschrittweite. Daraufhin wird das angegebene Intervall gemäß letzterer in Streifen zerlegt, und für jedes dieser Teilintervalle der Funktionswert bestimmt. Der höchste vorkommende Funktionswert und der zugehörige Abszissenwert sind eine erste Näherung für das Maximum; der tatsächliche Wert wird in einem der beiden benachbarten Streifen vermutet. Dementsprechend wird das Intervall auf diese beiden Streifen verkleinert, und mit 100 Teilintervallen darin die Suche analog fortgesetzt (durch einen rekursiven Aufruf). Diese weitere Unterteilung setzt sich fort, bis durch das Verkleinern des Intervalls die Breite kleiner wird als die geforderte Genauigkeit; die Mitte des letzten Intervalls wird dann als Näherung für den Abszissenwert des Maximums geliefert.

A.2.10.1 Java-Version

Das Interface `FunktObj` aus der Musterlösung **A.2.9.2 Java: Verwendung eines funktionalen Objekts** auf Seite 204 zur Aufgabe **A.1.9 Einfache Integration** auf Seite 98 zur Deklaration einer Funktion mit einer Veränderlichen wird hier noch einmal verwendet:

```
// Time-stamp: "30.05.02 13:49 FunktObj.java klaus@lap2.wachtler.de"
//
// Interface eines funktionalen Objekts zur Lösung der
// Übungsaufgabe "Einfache Integration" und anderer ähnlicher Probleme.

interface FunktObj
{
    double f( double x );
}
```

Die eigentliche Musterlösung ist folgende:

```
// Time-stamp: "30.05.02 14:56 maxfun.java klaus@lap2.wachtler.de"
//
// Musterlösung der Aufgabe "Maximum einer Funktion"
//
```

```
class maxfun
{
    private FunktObj    f;

    // Konstruktor
    // Eingabeparameter:
    //
    //   f:            Funktionsobjekt (die Nullstelle dessen Methode f()
    //                 ist gesucht)
    public maxfun( FunktObj funktion )
    {
        f = funktion;
    }

    // Sucht im Intervall [a,b] nach dem Maximum einer Funktion f(x).
    //
    // Eingabeparameter:
    //
    //   a:            linke Intervallgrenze
    //   b;            rechte Intervallgrenze
    //   Genauigkeit:  gewünschte Genauigkeit (in x-Richtung)
    //   Schrittweite: anfängliche Schrittweite, in der das Intervall
    //                 unterteilt wird
    //
    // Rückgabe:
    //   Abszissenwert der gefundenen Nullstelle
    //
    // Vorgehensweise:
    // Beim anfänglichen Aufruf wird das Intervall mit der angegebenen
    // Schrittweite unterteilt, und der größte aller Funktionswerte gesucht.
    // Daraufhin wird das gesuchte Maximum im Intervall vermutet, das um
    // Schrittweite vor dem gefundenen Höchstwert beginnt, und um
    // Schrittweite hinter dem gefundenen Höchstwert endet.
    // maxfun ruft sich dann für dieses Intervall wieder selbst auf mit den
    // neuen Grenzen, der ursprünglichen Genauigkeit, und einer Schrittweite,
    // die als 1/100 der neuen Intervalllänge berechnet wird.
    //
    // Die Rekursion endet, wenn die Intervalllänge kleiner als die
    // gewünschte Genauigkeit ist.

    double max( double a,
                double b,
                double Genauigkeit,
                double Schrittweite
                )
    {
        // a und b ggf. richtig sortieren:
        if( a>b )
        {
            double temp = a; a = b; b = temp;
        }
```

```
        // Wenn die Genauigkeit bereits erreicht ist, dann einfach
        // die Mitte des Intervalls liefern:
        if( b-a < Genauigkeit )
        {
            return ( b + a )/2.0;
        }
        else
        {
            // den jeweils größten gefundenen Funktionswert in maxy und
            // den zugehörigen Abszissenwert in maxx merken!
            // Weil möglicherweise in der Schleife der Endwert des
            // Intervalls nicht genau getroffen wird, ist es am geschicktesten
            // den Maximalwert gleich mit der rechten Grenze zu initialisieren.
            // Damit ist diese auch gleich berücksichtigt!
            double maxx = b;
            double maxy = f.f( b );

            // Schleife über alle Schritte:
            for( ; a<b; a+=Schrittweite )
            {
                if( f.f( a )>maxy )
                {
                    // größeren Wert als bisher gefunden!
                    maxy = f.f( a );
                    maxx = a;
                }
            }

            // der bisherige Maximalwert liegt bei (maxx,maxy).
            // Das darum liegende Intervall genauer durchsuchen:
            return max( maxx-Schrittweite,   // neues a
                        maxx+Schrittweite,   // neues b
                        Genauigkeit,         // bleibt
                        2*Schrittweite/100.0 // feiner unterteilt
                        );
        }
    }

} // class Maxfun
```

Das zugehörige Testprogramm:

```
// Time-stamp: "30.05.02 14:29 test_maxfun.java klaus@lap2.wachtler.de"
//
// Testprogramm zur Musterlösung der Aufgabe "Maximum einer Funktion"
//
// javac test_maxfun.java FunktObj.java maxfun.java
// java test_maxfun
```

```
// Die geforderte Funktion:
class f1 implements FunktObj
{
    public double f( double x )
    {
        return 15.0 + 13.0*x - 0.5*x*x;
    }
}

public class test_maxfun
{
    public static void main( String[] args )
    {
        // Maximum von f() in f1 suchen:
        f1     meinefunktion = new f1();
        maxfun meinmax = new maxfun( meinefunktion );
        double x = meinmax.max( -1000.0,        // Intervall von
                                +1000.0,        // Intervall bis
                                0.0001,         // Genauigkeit
                                1.0             // Anfangsabstand
                                );
        System.out.println( "Maximum f(" + x + ") = "
                            +
                            meinefunktion.f( x )
                            );

    }

}
```

A.2.10.2 C++-Version

```
// Time-stamp: "01.06.02 18:11 maxfun.cpp klaus@lap2.wachtler.de"
//
// Musterlösung der Aufgabe "Maximum einer Funktion"
//
// Kompilieren und starten unter Linux:
// g++ -Wall -DTESTPROGRAMM maxfun.cpp -o maxfun && maxfun
//
// Kompilieren und starten unter Win32/VC++ 6.0:
// cl /Zi /GX -DTESTPROGRAMM maxfun.cpp
// .\maxfun


#include <iostream>

using namespace std;
```

```
// Sucht im Intervall [a,b] nach dem Maximum einer Funktion f(x).
//
// Eingabeparameter:
//
//   f:            (Adresse der) Funktion, deren Nullstelle gesucht ist
//   a:            linke Intervallgrenze
//   b;            rechte Intervallgrenze
//   Genauigkeit:  gewünschte Genauigkeit (in x-Richtung)
//   Schrittweite: anfängliche Schrittweite, in der das Intervall
//                 unterteilt wird
//
// Rückgabe:
//   Abszissenwert der gefundenen Nullstelle
//
// Vorgehensweise:
// Beim anfänglichen Aufruf wird das Intervall mit der angegebenen
// Schrittweite unterteilt, und der größte aller Funktionswerte gesucht.
// Daraufhin wird das gesuchte Maximum im Intervall vermutet, das um
// Schrittweite vor dem gefundenen Höchstwert beginnt, und um
// Schrittweite hinter dem gefundenen Höchstwert endet.
// maxfun ruft sich dann für dieses Intervall wieder selbst auf mit den
// neuen Grenzen, der ursprünglichen Genauigkeit, und einer Schrittweite,
// die als 1/100 der neuen Intervalllänge berechnet wird.
//
// Die Rekursion endet, wenn die Intervalllänge kleiner als die
// gewünschte Genauigkeit ist.

double maxfun( double( *f )( double x ),
               double a,
               double b,
               double Genauigkeit,
               double Schrittweite
               )
{
  // a und b ggf. richtig sortieren:
  if( a>b )
  {
    double temp = a; a = b; b = temp;
  }
  // Wenn die Genauigkeit bereits erreicht ist, dann einfach
  // die Mitte des Intervalls liefern:
  if( b-a < Genauigkeit )
  {
    return ( b + a )/2.0;
  }
  else
  {
    // den jeweils größten gefundenen Funktionswert in maxy und
    // den zugehörigen Abszissenwert in maxx merken!
    // Weil möglicherweise in der Schleife der Endwert des
    // Intervalls nicht genau getroffen wird, ist es am geschicktesten
```

```
    // den Maximalwert gleich mit der rechten Grenze zu initialisieren.
    // Damit ist diese auch gleich berücksichtigt!
    double maxx = b;
    double maxy = f( b );

    // Schleife über alle Schritte:
    for( ; a<b; a+=Schrittweite )
    {
      if( f( a )>maxy )
      {
        // größeren Wert als bisher gefunden!
        maxy = f( a );
        maxx = a;
      }
    }

    // der bisherige Maximalwert liegt bei (maxx,maxy).
    // Das darum liegende Intervall genauer durchsuchen:
    return maxfun( f,                  // Funktion weiterreichen
                   maxx-Schrittweite,   // neues a
                   maxx+Schrittweite,   // neues b
                   Genauigkeit,         // bleibt
                   2*Schrittweite/100.0 // feiner unterteilt
                   );
  }

}

//////////////////////////////////////////////////////////////////////////
//
// Testprogramm:
//

#ifdef TESTPROGRAMM

double f1( double x )
{
  return 15.0 + 13.0*x - 0.5*x*x;
}


int main( int nargs, char **args )
{
  double Nullstelle = maxfun( f1, -1000.0, +1000.0, 0.0001, 1.0 );

  cout << " f(" << Nullstelle << ")=" << f1( Nullstelle )  << endl;

  return 0;
} // main( int nargs, char **args )

#endif // ifdef TESTPROGRAMM
```

A.2.11 Einen String in einem anderen suchen

A.2.11.1 Java-Version

TODO...

A.2.11.2 C++-Version

TODO...

Anhang B

Spezielle Algorithmen

B.1 Gaußscher Algorithmus zur Bestimmung des Ostersonntags

```
// Time-stamp: "(23.04.02 21:10) gauss.cpp [Klaus@Wachtler.de]"
//
// +-----------------------------------------------------------------------+
// |                                                                       |
// | i = jahr/100 - jahr/400 + 4;                                          |
// |                                                                       |
// +-----------------------------------------------------------------------+
// |                                                                       |
// | j = i - jahr/300 + 11;                                                |
// |                                                                       |
// +-----------------------------------------------------------------------+
// |                                                                       |
// | a = ( ( jahr%19 )*19 + j )%30;                                        |
// |                                                                       |
// +-----------------------------------------------------------------------+
// |                                                                       |
// | b = ( ( ( jahr%4 )*2 + 4*jahr + 6*a + i )%7 ) + a - 9;                |
// |                                                                       |
// +-----------------------------------------------------------------------+
// |                                                                       |
// |               b<1                                                     |
// |                |                                                      |
// +----- j --------|---------------- n ----------------------------------+
// |                |                                                      |
// |                |                  (b==26)                             |
// |                |                   ODER                               |
// |                | (a==28 UND b==25 UND ( ( 11*( j + 1 ) )%30 )<19) ) |
```

```
// |                 |                 |                                   |
// |                 +------- j ------+----------- n ---------------------+
// |                 |                 |                                   |
// |   ot = 31 + b;  |  b = b - 7;     |                                   |
// |   om = 3;       |                 |                                   |
// |                 +----------------+-----------------------------------+
// |                 |                                                     |
// |                 |             ot = b;                                 |
// |                 |             om = 4;                                 |
// |                 |                                                     |
// +----------------------------------------------------------------------+
//
// Ostersonntag = 1. Sonntag nach 1. Vollmond im Frühjahr

#include <stdio.h>

void gauss( int jahr )
{
  int i, j, a, b, ot, om;

  i = jahr/100 - jahr/400 + 4;
  j = i - jahr/300 + 11;
  a = ( ( jahr%19 )*19 + j )%30;
  b = ( ( ( jahr%4 )*2 + 4*jahr + 6*a + i )%7 ) + a - 9;

  if( b<1 )
  {
    ot = 31 + b;
    om = 3;
  }
  else
  {
    if( (b==26)
        ||
        (a==28 && b==25 && ( ( 11*( j + 1 ) )%30 )<19)
        )
    {
      b = b - 7;
    }
    ot = b;
    om = 4;
  }

  printf( "%02d.%02d.%04d\n", ot, om, jahr );
}


int main( int nargs, char **args )
{
  for( int i=1980; i<2006; i++ )
  {
```

```
    gauss( i );
  }
  return 0;
}
```

Anmerkung: weitere interessante Informationen zum Kalender, zugehörige Algorithmen und Literaturverweise finden sich in der c't 15/97, S. 312.

Weitere ganz interessante Informationen zu anderen Kalenderrechnungen erhält man über http://www.koenigsmuenster.de/rsk/einfuehrung.htm; beispielsweise kann man unter http://www.koenigsmuenster.de/rsk/dechija.htm studieren, wann das jüdische Jahr beginnt. Dagegen ist Ostern eine leichte Übung.

Literaturverzeichnis

[v.Helden] Prof.Dr. Jürgen Dunkel, Prof.Dr. Josef v. Helden:
Informatik 2
Sommersemester 2000
FH Hannover, FB Informatik

[Knuth 1] Donald E. Knuth:
The Art of Computer Programming
Vol. 1: Fundamental Algorithms
2. Auflage
Addison-Wesley, 1973
ISBN 0-201-03809-9

[Knuth 2] Donald E. Knuth:
The Art of Computer Programming
Vol. 2: Seminumerical Algorithms
2. Auflage
Addison-Wesley, 1971
ISBN 0-201-03822-6

[Knuth 3] Donald E. Knuth:
The Art of Computer Programming
Vol. 3: Sorting and Searching
2. Auflage
Addison-Wesley, 1973
ISBN 0-201-03803-X

[Knuth Sel.] Donald E. Knuth:
Selected Papers on Computer Science
Center for the Study of language and Information (CSLI Publications), Leland Stanford Junior University 1996
ISBN 1-881526-91-7 (paperback)
ISBN 1-881526-92-5 (hardback)

Kein Lehrbuch, aber sehr interessant zu lesen

[Ziegenbalg] Jochen Ziegenbalg:
Algorithmen
Von Hammurapi bis Gödel
Spektrum Akademischer Verlag, 1996
ISBN 3-8274-0114-3

Recht interessant zu lesen, weil neben der nüchternen Beschreibung auch ein paar Geschichten erzählt werden.

[Sedgewick C] Robert Sedgewick:
Algorithmen in C
Addison Wesley 1992
ISBN 3-89319-376-6

Gutes Lehr- und Referenzbuch für einen relativ weiten Bereich (Suchen, Sortieren, Zeichenfolgen, etwas Geometrie, etwas Numerik)

[Sedgewick C (eng.)] Robert Sedgewick:
Algorithms in C
Addison Wesley 1990
ISBN 0-201-51425-7

Originaltext der deutschen Übersetzung [Sedgewick C]

[Strou IV] Bjarne Stroustrup:
Die C++ Programmiersprache
4. Auflage
Addison-Wesley, 1999

[Cline: C++ FAQs] Marshall Cline, Greg Lomow, Mike Girou:
C++ FAQs
2nd Edition
Addison-Wesley, 1999
Eine Sammlung vieler kleinerer Aspekte zu C++, CORBA und COM

[c't 5/89 Bayerbäume] :
Bayer-Bäume
-
c't 5/89, S. 178

[c't 6/89 Winkler/M.] Eckart Winkler, Dirk Meiners:
Starke Typen

Funktions- und Prozedurtypen in modernen Programmiersprachen
c't 6/89, S. 174

Kurze und verständliche Beschreibung der Übergabe von Funktionen und Prozeduren als Parameter in C, Turbo-Pascal, und Modula-2

[c't 6/89 Berchtold] Stefan Berchtold:
Evolution im Computer
Optimierung mit genetischen Algorithmen
c't 6/89, S. 130

Übersichtliche Beschreibung des Konzepts der genetischen Algorithmen, mit einem Beispiel in Pascal (Entwickeln einer Spielstrategie für das Spiel Nimm*)*

[Optimal schnell] Guido Gronek:
Optimal schnell
Schnelle Suche mit „Optimal Mismatch"
c't 3/1995, S. 278ff.

[Mustergültig] Reiner Högel:
Mustergültig
Von Wildcards und regulären Ausdrücken
c't 3/1995, S. 278ff.

[Blitzfindig] Michael Tamm:
Blitzfindig
Texte schnell durchsuchen mit T-Search
c't 1997, S. 292ff.

[Eckel: TIC++ 1] Bruce Eckel:
Thinking in C++
Volume 1
http://www.mindview.net/Books/DownloadSites

[Eckel: TIC++ 2] Bruce Eckel:
Thinking in C++
Volume 2
http://www.mindview.net/Books/DownloadSites

[Eckel: TIJ] Bruce Eckel:
Thinking in Java
http://www.mindview.net/Books/DownloadSites

[KW-C] Klaus Wachtler:
0 ist false, alles andere true
Eine Einführung in ANSI-C
Selbstverlag, 2002
http://www.wachtler.de

[KW-C++] Klaus Wachtler:
Ein Überblick über die Programmierung mit C++
Selbstverlag, 2002
http://www.wachtler.de

[Muellges/R.] G. Engeln-Müllges, F. Reutter:
Numerische Mathematik
Bibliographisches Institut Mannheim
ISBN 3-411-01688-4

Sehr praxisgerechte Darstellung der wichtigsten numerischen Verfahren aus der Technik (Fehleranalyse, Lösen von Gleichungen, numerische Lösung von linearen und nichtlinearen Gleichungssystemen, Approximation, Interpolation, numerisches Integrieren, Anfangswertprobleme).

Als Ergänzung ist eine Formelsammlung [Muellges/R./F77] erhältlich.

[Muellges/R./F77] G. Engeln-Müllges, F. Reutter:
Formelsammlung zur Numerischen Mathematik mit Standard-FORTRAN 77-Programmen
Bibliographisches Institut Mannheim
ISBN 3-411-03125-5

Formelsammlung zum Nachschlagen als Ergänzung zu [Muellges/R.]; enthält alle Verfahren in Fortran 77 formuliert. Analoge Ausgaben sind mit BASIC, Pascal und C erhältlich.

[Al-Khwarizmi] J. J. O'Connor, E. F. Robertson:
Abu Ja'far Muhammad ibn Musa Al-Khwarizmi
The MacTutor History of Mathematics archive
School of Mathematical and Computational Sciences
University of St Andrews

http://www-gap.dcs.st-and.ac.uk/%7Ehistory/Mathematicians/Al-Khwarizmi.html

Index