Allgemeine Informatik I / II
(für WiMa, E-Technik, . . .)

F. Schweiggert

23. April 2006

Fakultät Mathematik u. Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung
Vorlesungsbegleiter (gültig ab Wintersemester 2005/2006)

Hinweise:

- Teile dieser Unterlagen stammen vom gleichnamigen Skript von Dr. A. Borchert und sind mit dessen Genehmigung eingearbeitet worden
- Viele verwendete LaTeX-Macros stammen von Dr. A. Borchert und Dr. J. Mayer
- Die enthaltenen Beispiele sind teilweise unter Solaris und teilweise unter Linux erstellt – im Einzelfall sind immer die jeweiligen Manualseiten zu befragen!

! Dieser Vorlesungsbegleiter ersetzt weder den Besuch der Vorlesung noch den der Übungen!

Inhaltsverzeichnis

Kapitel 1

Vorbemerkungen

1.1 Ziele der Vorlesung

Die Vorlesung verfolgt im Kern zwei wesentliche Ziele:

- Solide Einführung in das Fach der Informatik, so dass später weiterführende Veranstaltungen besucht werden können.

  Alle Kerngebiete der Informatik – theoretische, praktische und angewandte Informatik – werden mit ausgewählten Aspekten vertreten sein.

- Erlernung des fundierten praktischen Umgangs mit Rechnern.

  Das geht deutlich über das Finden des richtigen Menüs und des richtigen Buttons hinaus (“Mausschieberei”).

  Es geht darum, das Innenleben eines Rechners / Betriebssystems prinzipiell verstehen zu lernen, um so u.a. in der Lage zu sein, für eigene Arbeit selbst notwendige Werkzeuge erstellen und somit viel effizienter und effektiver mit Rechnern umgehen zu können.

1.2 Organisation der Vorlesung

Zu der Veranstaltung Allgemeine Informatik I zählen

- die Vorlesung,
- die Übungen,
- die Tutorien, die gruppenweise jede Woche an mit dem jeweiligen Tutor verabredeten Terminen stattfinden und
- die aktive Beteiligung durch das Nacharbeiten der Vorlesung, das Lösen von Übungsaufgaben, die Teilnahme an den Tutorien und das Stellen von Fragen in allen Veranstaltungen, per E-Mail oder in den Sprechstunden.

Teilnahme an den Übungen:

- Da die Vorlesung weitgehend im Präsentationsstil gehalten wird, droht gerade in der Informatik die Gefahr, dass die Notwendigkeit einer aktiven Beteiligung **unterschätzt** wird.
- Viele Vorlesungsinhalte wirken zunächst durchaus "einleuchtend" und es ist zu Beginn auch nicht schwierig, ihnen zu folgen. Das gilt insbesondere dann, wenn zu Beginn bereits Erfahrungen mit dem Umgang mit Computern vorliegen.
- Es liegt jedoch in der Natur dieses Faches, dass viele subtile aber wesentliche Feinheiten nur dann auffallen, wenn die Übungsaufgaben zeitnah erledigt werden.
- Später werden die aus den Übungen zu gewinnenden Kenntnisse selbstverständlich vorausgesetzt.
- Diejenigen, die die Übungsteilnahme zu Beginn unterschätzt haben, weil die Vorlesung so leicht erscheint, erfahren dann irgendwann im Laufe des Wintersemesters einen Moment, ab dem sie in der Vorlesung **völlig abgehängt** werden.
  - Das ist dann sehr frustrierend – für alle Beteiligten.
  - Ein "Das lerne ich dann in den Semesterferien nach" hat bislang bei den wenigsten funktioniert!
  - **Allgemeine Informatik II basiert zu 100% auf Allgemeine Informatik I!**
- **Studieren heißt „sich um die Inhalte selbst zu bemühen"** (lat.: studere = streben (nach etw.), sich (um etw. bemühen)
- **Es gibt nichts Gutes, außer man tut es!** (Erich Kästner)

Anmeldeverfahren:

- Sie reihen sich zu Ihrem Termin in eine Schlange vor einem der Räume O27/211 bzw. O27/213 und warten, bis einer der Tutoren frei wird.
- Der Tutor nimmt Ihre persönlichen Daten entgegen (Name, Matrikelnummer, Studiengang) und gibt Ihnen Gelegenheit, ein erstes Passwort einzutragen.
- Achten Sie bitte darauf, dass Ihr Name korrekt eingetragen wird. Sie werden genau die gleiche Schreibweise später auf Ihrem Schein vorfinden.
- Der Tutor lässt die Benutzungsrichtlinien ausdrucken, die Sie unterschreiben müssen. Der Tutor zeichnet die Richtlinien gegen.
- Die unterschriebenen Benutzungsrichtlinien werden gesammelt und bei unserem Sekretariat abgegeben, das alle ordnungsgemäßen Anträge freischaltet.
- Wenn alles klappt, haben Sie Ihren Zugang binnen ein oder zwei Tagen. Das heißt, dass es spätestens Ende dieser Woche klappen sollte.
- Kontaktieren Sie uns bitte, falls es dabei Probleme geben sollte.

1.3 Rechnerzugang, Rechnerbetrieb

Wahl des Passworts:

- Mit einer Benutzungsberechtigung bei uns übernehmen Sie persönlich die Verantwortung für Ihren Zugang.
- Diese beginnt mit der Wahl eines sicheren Passworts.
- Das ist keine triviale Hürde, da gute Passwörter nicht leicht zu erfinden sind und sich nur schwer einprägen lassen.

Folgende Regeln sind zu beachten:

- Das Passwort sollte genau 8 Zeichen lang sein. (Kürzer ist zu kurz und alles hinter dem achten Zeichen wird unglücklicherweise ignoriert).
- Zugelassen sind alle Zeichen, die die Tastatur hergibt: Klein- und Großbuchstaben, Ziffern, Sonderzeichen.
- Vermeiden Sie Umlaute, "ß" und Funktionstasten.
- Das Passwort darf keinem bekanntem Wort ähneln, egal aus welcher Sprache. Genauso sind Namen, Ortsbezeichnungen, Geburtsdaten, Auto-Kennzeichen, Telefonnummern usw. allesamt tabu.
- Bei uns muss das Passwort mindestens einen Kleinbuchstaben und einen Großbuchstaben haben und eines der ersten sieben Zeichen muss ein Sonderzeichen sein.
- Hinweis: Achten Sie darauf, dass nicht versehentlich Caps-Lock oder Num-Lock aktiv sind!

- Wenn wir feststellen, dass wir Ihr Passwort “knacken” können, wird Ihr Zugang gesperrt, bis Sie bei uns persönlich vorbeischauen.

Schlechte Passwörter

101076	Somak4	challeng	holsten1	mephisto	sonne1
123asd	Susanne!	check-ma	hxnmstr	michael2	sonnensc
161065	TAbay1	claudia	inges.	micra1	sphinx
161072	Ukraine1	clemente	island!	mircea5	ssroessn
161278	Zhang!!!	cleo19	jager.	mkell1	stierle2
1anne1	abrahas	cocis20	jkjkjk	mopper	striker
1lichtwa	af5011	corvus	joasia	morisset	struppi!
1quinn	alegria	cumulus1	joker1	mousse24	sunpol
1qwert	alex95	departur	jsschrst	niomniom	tania2
2beijing	alexandr	dg1nfv0	julius.	orkork	taraxacu
2callent	algarve	diekts	junkie	ortho9	thpolt1
2emacs	angel!	dingmin	karmann	oyster7	tiger1
2mannan	antonia!	eduardo!	kashmir	patrick1	trabitra
8baume	anul99	elzbieta	kermit	peacock	tsvg11
Allgaeu0	apmats	eminem!	kickers2	pepper1	tuartor
Antimon!	apollo13	erleucht	konichiw	peugeot3	tueanh
Baller1	asterix0	euro97	kontroll	popen7	tw0477
Berlin?	avatar1	f7260h	laminar5	popo96	verdik.
Brckstdt	babis1	fabio?	leblanc	prodigy	vergil3
Cardix!	barbara!	fafnir!	lespaul	quasar	vietnam
EgkBaH	basti1	falcon3	lichen/	quattro	viktoria
Gwaihir	beast!	front242	lothar1	quoniam	voodoo2
Honey7	biblia1	furioso	loulou1	rack21	wave43
Kinderzi	bini11	ganter	lucent	radiatio	werner
LIEBE6	birten	garfield	madlen	radies1	winter96
Mailand!	blumensc	ge1706	maicel	revilo	xxxxxx
Moritz1	bmb850	gery4	mailman1	rotar!	zaborav1
Ninja1	bochum!	hammet2	mamusia	roxana	zeppez
Ninurta	bonzo9	harpe.	marsh5	sanjuan1	zeppi.
Pasquale	boreal!	hedwig!	maruca	scarface	zhangyin
Peppermi	bullfrog	heidi1	mathe1	sherry	
Philo!	butterfl	hijack	matthi78	simone2	
Reginald	butthead	himmelbl	me3203	sobaka	
Sharon!	carmen	hofbraeu	melanie8	sodom666	

Gute Passwörter

- Gute Beispiele: `iFm=AmSt` oder `rLK/1oeT`
  aber bitte ja keines der beiden wählen!
- Spruch-Methode: Man wähle irgendeinen Satz, beispielsweise “Ich freue mich auf mein Studium”, nimmt die Anfangsbuchstaben, variiert Groß- und Kleinschreibung und wirft ein Sonderzeichen ein.
- Zweiwort-Methode: Man wähle zwei Begriffe, beispielsweise “Erlkönig und Goethe”, nimmt daraus nur Fragmente und lässt die beiden verbleibenden Teile durch Sonderzeichen verbinden. Hier sollte ebenfalls Groß- und Kleinschreibung variiert werden.

Umgang mit Passwörtern

Es ist nicht nur wichtig, dass Sie sich ein gutes Passwort ausdenken, sondern dass Sie damit auch richtig umgehen:

- Nie aufschreiben!

  Es ist nicht schlimm, wenn Sie Ihr Passwort vergessen. Wenn Sie Ihren Studentenausweis dabei haben, können Sie einen von uns aufsuchen und sich selbst ein neues Passwort eintragen. Es empfiehlt sich, ein neues Passwort einzuüben, indem Sie sich ein Dutzend mal hintereinander anmelden.
- Lassen Sie sich nicht über die Schulter gucken bei der Eingabe des Passworts. Wenn Sie den Verdacht haben, dass jemand Ihr Passwort erspähen konnte, sollten Sie sich sofort ein neues geben. Dies geht mit dem **passwd**-Kommando.
- Senden Sie Ihr Passwort nie im Klartext über das Netzwerk. SSH (*secure shell*) ist gut, da hier alles verschlüsselt wird. Bei POP und FTP werden andere Passwörter verwendet.
- Benutzen Sie nie potentiell mit Viren oder Würmern verseuchte Rechner zur Anmeldung bei uns, da dann mit Schnüffelprogrammen gerechnet werden muss, die die Eingabe von Passwörtern abfangen.
- Ein periodischer Passwort-Wechsel pro Jahr genügt.

Wahl des Benutzernamens

Sie können bei uns frei einen Benutzernamen unter Beachtung folgender Regeln wählen:

- Länge zwischen 2 und 8 Zeichen.
- Besteht aus Kleinbuchstaben und Ziffern und muss mit einem Kleinbuchstaben beginnen.
- Er darf noch nicht vergeben sein.
- Er wird nie wieder geändert!

Beachten Sie bitte die letzte Regel: Wir ändern unter keinen Umständen den Benutzernamen.

Im Falle von Namensänderungen (Urkunde/Ausweis bitte mitbringen) wird nur der ebenfalls eingetragene volle Name geändert, jedoch nie der Benutzername.

Zugang zu unseren Rechnern

Sie haben viele Möglichkeiten, an unsere Rechner zu kommen:

- Öffentliche Pool-Räume mit Chipkarten-Zugang:

O27/211	24 Plätze, gut ausgestattet mit Sun Ultras
O27/213	24 Plätze mit älteren Maschinen

- Öffentliche Pool-Räume in der Helmholtzstraße 18:

E44	Zugang über die Mathematik-Bibliothek
140	reserviert für Diplomanden und Doktoranden

- Über SSH (*secure shell*) von einem beliebigen (hoffentlich sicheren) Rechner auf einen unserer Server:

Theseus	ist vorzuziehen; hier liegen auch Ihre Daten
Turing	sehr alter Server
Turan	Geheimtip, da bislang kaum genutzt
Thales	schnell, aber primär für die Fakultät

  Alle diese Namen gehören zur Domain `mathematik.uni-ulm.de`.

  Unter `http://ssh.mathematik.uni-ulm.de/` gibt es Hinweise zur Verwendung der SSH.

Arbeiten im Rechnerraum / am Rechner

- Nehmen Sie das, was Sie in den Raum hineinnehmen, bei Verlassen wieder mit oder entsorgen Sie die Abfälle im Abfalleimer!
- Verhalten Sie sich bitte stets so, dass Sie andere nicht stören!
- Halten Sie Tastatur und Maus sauber! Dies gilt auch und besonders für den (Flach-) Bildschirm!
- Benutzen Sie bitte die Kleiderhaken!
- **Rechner dürfen auf gar keinen Fall ausgeschaltet werden!**
- Wenn Sie etwas ausdrucken, nehmen Sie es auf jeden Fall mit!
- **Bevor Sie etwas drucken, denken Sie bitte nach:** Auf welchen Drucker? Ist das, was Sie drucken, eine Postscript-Datei? Mit welchem Programm wollen Sie drucken? Auf welchen Drucker wollen Sie die Ausgabe lenken?
- Lassen Sie Ihre mailbox nicht zu sehr anwachsen - es gibt bei jedem Mail-Programm ein(en) Delete-Kommando / -Button oder Save-Kommando / -Button (zum Abspeichern in den Heimatkatalog)!
- Belästigung Dritter direkt oder via email oder sonstwie sind tunlichst zu unterlassen – wir reagieren darauf typischerweise mit **Sperrung des logins!**
- Zur Belästigung Dritter können auch entsprechende Hintergrundbilder am Monitor dienen – also keine Pinup's und dgl.

- Surfen im Internet ist (noch) frei – **Vorrang hat aber der Lehrbetrieb!** Machen Sie also Ihr „Surf-Brett“ frei, wenn Bedarf ist!
- Surfen im Internet sollte primär dem Studium dienen – Seiten mit pornografischem, rassistischem, faschistischem (o.dgl.) Inhalt können Sie wo auch immer, aber nicht bei uns anschauen geschweige denn erstellen! Auch dies führt zur Sperrung des Login! Herunterladen von mp3-Dateien kann gegen das Urheberrechtsgesetz verstoßen und somit **strafbar** sein!
- Mutwillige Störung des Rechnerbetriebs ebenso wie mutwillige Zerstörungen werden nicht toleriert (Anzeige, Sperrung des Login!)

Freiheiten müssen durch verstärktes
Verantwortungsbewusstsein ausgeglichen werden
James P. Comer

Benutzungsrichtlinien In unseren Richtlinien geht es um folgende Punkte:

- Es muss immer sichergestellt sein, wer für welchen Zugang die Verantwortung trägt. Deswegen dürfen Sie nie Ihren Zugang mit jemand anders teilen oder auch nur temporär zur Verfügung stellen.
- Sie dürfen die Sicherheit unserer Rechner nicht gefährden. Das wäre beispielsweise der Fall, wenn es jemand anders gelänge, an Ihren Zugang zu kommen.
- Sie dürfen andere in der Benutzung unserer Rechner-Ressourcen nicht behindern. Rechen- und speicherintensive Anwendungen unterliegen daher strengen Richtlinien. Auch der zur Verfügung stehende Plattenplatz ist aus diesem Grunde reglementiert.
- Der Zugang bei uns darf nur für Studienzwecke genutzt werden.

Signifikante Verletzungen der Richtlinien führen zur temporären Sperrung eines Zugangs, bis der Vorfall durch einen persönlichen Besuch bei uns geklärt ist.

Plattenplatz

- Sie dürfen ohne weitere Rückfragen 50 Megabyte auf Dauer belegen.
- Kurzfristig darf es auch deutlich mehr sein. Es gibt keine Quota bei uns.
- Wenn Sie über 50 Megabyte an einem Wochenende belegen, gibt es eine warnende automatisch generierte E-Mail.
- Wenn Sie diese E-Mails mehrfach ignorieren, führt dies zur temporären Sperrung.
- Wenn Sie für Studienzwecke wirklich mehr Plattenplatz benötigen, erhöhen wir gerne Ihre Schranke. Eine E-Mail an uns genügt.
- Feststellen des eigenen Verbrauchs: Im Heimatkatalog (wird noch erläutert) das Kommando **du** aufrufen!
- Reduzieren des eigenen Verbrauchs:
  - Löschen nicht mehr benötigter / leicht wieder erzeugbarer Daten (Kommando **rm**
  - Komprimieren von Dateien (Kommandos wie **zip** oder **gzip** oder **tar**

Benutzung der Drucker

- Es stehen mehrere Laser-Drucker zur allgemeinen Verwendung zur Verfügung:

Gutenberg	O27/213
Garamond	O27/211
Merian	Helmholtzstraße 18, E44

- Sie erhalten in jedem Semester ein Kontingent von 200 Seiten, das in zwei Teilen zu jeweils 100 Seiten vergeben wird. Das heißt, dass die zweite Hälfte erst irgendwann in der Mitte des Semesters freigegeben wird.
- Dieses Kontingent ist strikt nur für Studienzwecke zu verwenden. **Vorlesungsskripte** dürfen nicht über unsere Drucker ausgedruckt werden. Zulässig ist beispielsweise der Ausdruck von Übungsblättern oder Ihrer Lösung zu einem Übungsblatt.
- Es wird dringend empfohlen, von den **Druck-Möglichkeiten beim KIZ** Gebrauch zu machen (über Ihren KIZ-Zugang). Für diese Drucker können Sie auch über den Unishop beliebig weitere Kontingente nachkaufen. Entsprechend sind die Drucker beim KIZ auch für den Ausdruck von Vorlesungsskripten geeignet.

Will man den Inhalt einer Datei ausdrucken (bitte prüfen, ob das überhaupt Sinn macht — ausführbare Programme kann kein Mensch lesen!), so muss man beachten, dass die Drucker im WiMa–Net nur die Seitenbeschreibungssprache **Postscript** verstehen – manche Formate wie das übliche PDF werden implizit umgewandelt!

Feststellen, ob der Inhalt einer Datei *myfile* Postscript ist:

```
turing$ head myfile
%!PS-Adobe-2.0
%%Creator: dvips(k) 5.86 Copyright 1999 Radical Eye Software
%%Title: all.dvi
%%Pages: 257
%%PageOrder: Ascend
%%BoundingBox: 0 0 596 842
%%DocumentFonts: Times-Roman Times-Bold Times-Italic Courier
%%EndComments
%DVIPSWebPage: (www.radicaleye.com)
%DVIPSCommandLine: dvips -o all.ps all.dvi
```

Ist der Inhalt *myfile* einer Datei „Postscript" (kann man übrigens mit dem Editor anschauen), so kann wie folgt ausgedruckt werden:

```
lp -dgaramond XY
```

Damit wird die Datei auf dem Drucker **garamond** ausgedruckt (**-d** : **d** steht für *destination*).

In den **Shell-Variablen** `PRINTER` bzw. `LPDEST` ist eine **Standardeinstellung** definiert; lässt man also oben die Angabe *-dgaramond* weg, so wird der Standarddrucker angesteuert!

Ist der Inhalt einer Datei normaler (ASCII-) Text, z.B. ein mit einem Editor erstellter Programmtext, so wird dieser mit **a2ps** in Postscript umgewandelt und direkt und sofort auf den Drucker geschickt:

```
a2ps -Pgutenberg datei
```

Auf die Angabe *-Pgutenberg* kann man verzichten – es wird auf den Standarddrucker (siehe oben: Shell-Variable LPDEST / PRINTER) ausgegeben

Statt auf den Drucker kann das auch in eine Datei gehen – die erzeugte Postcript-Datei nennt man am besten wie die Ausgangsdatei gefolgt von der Endung **.ps**, also

```
a2ps -o datei.ps datei
```

Druckjob abbrechen: lprm oder cancel

E-Mails

Sie erhalten mit der Einschreibung auch eine Email-Adresse, meist in der Form `Vorname.Name@uni-ulm.de`.

Da Sie zu den Rechner der Fakultät für Mathematik und Wirtschaftswissenschaften (Übungen, Punkteserver, ...), brauchen Sie auch einen `Login` bei uns. Damit verbunden ist eine weitere Email-Adresse der Form `loginname@mathematik.uni-ulm.de`. Es wird aber in absehbarer Zeit (spätestens bis Ende 2006) nur noch die erstgenannte Adresse geben!

- Wir und die Tutoren verwenden ausschließlich Ihre E-Mail-Adressen bei uns. Wenn Sie E-Mails weitergeleitet haben möchten, finden Sie Hinweise dazu unter: `http://www.mathematik.uni-ulm.de/admin/qmail/`
- Wir empfehlen Ihnen jedoch, E-Mails für Studienzwecke direkt bei uns zu lesen und zu versenden. Das ist viel zuverlässiger als die zahllosen Free-Mailer und ist auch von außen über eine SSH erreichbar.
- POP wird ebenfalls unterstützt, muss jedoch von Ihnen selbst konfiguriert werden: `http://www.mathematik.uni-ulm.de/admin/qmail/pop.html`
- Wenn Sie auf einer Web-Schnittstelle bestehen, empfiehlt sich der entsprechende Dienst beim KIZ.
- Bitte versenden Sie keine E-Mails mit umfangreichen Anhängen. Das übliche Limit liegt bei einem Megabyte.
- Wenn Sie größere Datenmengen mit jemanden austauschen möchten, geht dies auch mit individuellen FTP-Zugängen. Mehr dazu unter: `http://www.mathematik.uni-ulm.de/admin/adis-ftp/`

Etwas Netiquette:

Es gibt zig Webseiten, die über **Etiquette** im Internetverkehr (kurz „Netiquette“ genannt) informieren. Hier einige Aspekte im Kontex mit Emails:

- Emails sind wie Briefe – nur eben elektronisch!
- Die “Subject:”- oder “Betreff:”-Zeile muss vorhanden sein und muss auch für den Inhalt sprechen. Es gibt Personen, die hunderte von Emails pro Tag erhalten – hier hilft ein klarer Titel beim Sortieren.
- Humor, Ironie oder Sarkasmus werden vom Empfänger oft nicht so verstanden. Wie in Briefen auch fehlen eben Körpersignale wie Schmunzeln oder Augenzwinkern. Smilies wie :-) waren eine Zeitlang Mode, können aber kindisch wirken.
- Zeilen, die mehr als 70 Zeichen lang sind, sind schwer zu lesen.
- GROSS GESCHRIEBENE PASSAGEN SIND SCHWIERIG ZU LESEN!
- Groß- und Kleinschreibung: REINE GROSS-SCHREIBUNG WIRKT SO, ALS OB MAN SCHREIEN WÜRDE, konsequente kleinschreibung zeugt von bequemlichkeit und desinteresse.
- Wie bei Briefen auch auf Orthographie oder Grammatikfehler achten.

- HTML Formatierungen
  - sind unnötig,
  - machen die Email grösser,
  - werden nicht von allen Mailprogrammen (sinnvoll) angezeigt und
  - nerven Empfänger, die Email als Text archivieren.

  Ein Eingriff in die Privatsphäre passiert, wenn in HTML eingebaute unsichtbare Bilder Links zu externen Seiten enthalten. Dadurch kann Drittpersonen mitgeteilt werden, wann und wo Sie Ihre Email gelesen haben. Viele Direktwerber verwenden solche Tricks.

  Manche Menschen löschen solche Emails ohne sie zu lesen!

- Also: Emails als reinen Text verschicken!

- Auch reine Textinhalte können sauber und verständlich formatiert werden.

- *Carbon Copies* (**CC**) nur wenn nötig verwenden. Die Addressaten auf der CC Liste sollten wissen, warum sie die Email erhalten. Sie können auch verwirren, wenn die Empfänger nicht wissen, wer antworten soll.

- Auch *Blind Carbon Copies* (**BCC**) können für den Empfänger irritierend sein, also sparsam verwenden!

- Nicht auf Spam (Bulk-Email) antworten. Durch eine Antwort wird signalisiert, dass die Addresse aktiv ist – Folge: noch mehr Spams

- ...

Exotik unserer Rechner-Infrastruktur?

Unsere Rechner und die Werkzeuge, die wir darauf einsetzen, wirken für viele Neulinge exotisch:

- Bei unseren Rechnern handelt es sich um Arbeitsplätze und Servern von Sun Microsystems. Diese Rechner haben SPARC-Prozessoren, die zu der Familie der Intel-Prozessoren in keiner Weise kompatibel sind.
- Entsprechend ist es beispielsweise unmöglich, Produkte von Microsoft darauf laufen zu lassen.
- Als Betriebssystem wird Solaris eingesetzt. Dabei handelt es sich um eine Variante des originalen UNIX. Linux und das GNU-Projekt haben sich UNIX zum Vorbild genommen. Bei uns haben Sie Gelegenheit, das Original kennenzulernen.
- Nicht exotisch ist die Wahl der Programmiersprache, die für die Einführung in die Software-Entwicklung verwendet wird :

  Java[1] gibt es sowohl für Microsoft-Systeme wie auch für UNIX-Systeme!

Warum so eine Umgebung?

Bedenken Sie, dass wir keinen Volkshochschulkurs anbieten. Ziel ist es bei uns, dass Sie solide Grundlagen in der Informatik erhalten, die über die nur kurzfristig Nutzen bringende Vertrautheit mit zur Zeit populären Anwendungen und Programmiersprachen hinausgeht.

Wichtig für die Auswahl unserer praktischen Umgebung waren für uns folgende Kriterien:

- Sie ist einfach. Wir wollen keine kostbare Vorlesungszeit mit der Einführung irrelevanter Details und Komplexitäten verlieren, die in kürzester Zeit wieder veraltet sind.
- Sie folgt weitgehend internationalen Standards. Für die Arbeitsumgebung unter UNIX oder Linux gibt es einen gemeinsam befolgten IEEE-Standard.
- Wenn es Sie interessiert, können Sie hinter die Kulissen schauen und bis zum letzten Bit herausbekommen, wie es dahinter funktioniert.
- Bei Linux (das in der Benutzung Solaris weitgehend ähnelt) haben Sie die Möglichkeit, zu sehen, wie ein Betriebssystemkern funktioniert und bei den GNU-Werkzeugen sind ebenfalls die Quellen allesamt öffentlich.
- Es gibt sehr gute Fachliteratur aus renommierten Verlagen!

[1] Java ist eine objektorientierte Programmiersprache und als solche ein eingetragenes Warenzeichen der Firma Sun Microsystems

1.4 Zum Inhalt der Vorlesung

Folgende Themen gehören zu Allgemeine Informatik I:

- Kurze Einführung in unsere Arbeitsumgebung einschließlich einer Einführung in das UNIX-Dateisystem und wichtige Werkzeuge unter UNIX.
- Einführung in die praktische Programmierung mit Java einschließlich Datentypen (Basistypen, Arrays und Records), Schleifen, Prozeduren, Rekursion, Ein- und Ausgabe und die Einbettung in die UNIX-Umgebung. Objektorientierte Konzepte werden hier noch hintenangestellt!
- Einführung in Sortier-Algorithmen.
- Einführung in formale Sprachen und endliche Automaten.

Im Anschluss werden im Sommersemester in "Allgemeine Informatik II" folgende Themen angeboten:

- Fortgeschrittene Rekursionstechniken einschließlich Recursive-Descent-Parsing, Backtracking und Branch-And-Bound-Verfahren.
- Mehr zu formalen Sprachen.
- Einführung in dynamische Datenstrukturen einschließlich linearen Listen, Bäumen und Hash-Verfahren.
- Objekt-orientierte Programmierung

1.5 Etwas zur Informatik

- **technische Informatik:**
  Aufbau von Computern mit vorhandenen technologischen Mitteln nach vorgegebener Spezifikation in Abhängigkeit von Preis, Geschwindigkeit, Kapazität und Zuverlässigkeit
- **theoretische Informatik:**
  Untersuchung von „intuitiv" gewonnenen Methoden, Schaffung von mathematisch fundierten Theorien und Methoden, Untersuchung von Algorithmen z.B. auf ihre Komplexität, Fragen der Berechenbarkeit schlechthin, ...
- **angewandte Informatik:**
  Einsatz existierender Rechner mit vorgegebenen Eigenschaften in Produktion, Verwaltung, etc zur Lösung konkreter Probleme – Fragen der Analyse, Modellierung, Spezifikation – Erstellen von Anwendungssoftware unter Berücksichtigung von Qualitätsaspekten wie z. B. Benutzungsfreundlichkeit, Wiederverwendbarkeit, Pflegbarkeit oder Prüfbarkeit.
  Stichworte: *Software Engineering, Datenbanken*
- **Kerninformatik:**
  Bindeglied zwischen technischer Informatik und angewandter Informatik – Betriebssysteme, Datenbankmanagementsysteme, Compilerbau, Tools, ...
- andere Gebiete: *KI – Künstliche Intelligenz, Expertensysteme, Neuro-Informatik, ...*

Informatikabteilungen an der Uni Ulm

- Abteilung Angewandte Informationsverarbeitung
  Leiter: Prof. Schweiggert – Mitglied der Fakultät für Mathematik und Wirtschaftswissenschaften sowie kooptiertes Mitglied der Fakultät für Informatik
- Abteilungen der Fakultät für Informatik (www.informatik.uni-ulm.de)
  - Datenbanken und Informationssysteme – Leiter: Prof. Dadam
  - Künstliche Intelligenz – Leiter: Prof. v.Henke
  - Neuroinformatik – Leiter: Prof. Palm
  - Programmiermethodik und Compilerbau – Leiter: Prof. Partsch
  - Theoretische Informatik – Leiter: Prof. Schöning
  - Verteilte Systeme – Leiter: Prof. Schulthess
  - Medieninformatik – Leiter: Prof. Weber

Basis-Vorlesungen der SAI

- Allgemeine Informatik I (2/2) – WS
- Allgemeine Informatik II (2/2) – SS
  Diese Veranstaltung setzt die Allgemeine Informatik I nahtlos fort!
- Allgemeine Informatik III (2/2) – WS – identisch mit „Systemnahe Software I“
  Diese Veranstaltung verlangt (insb. praktische) Kenntnisse, die in „Allgemeiner Informatik I / II“ vermittelt werden.
- Systemnahe Software (2/2) – SS – identisch mit „Systemnahe Software II“
  Diese Veranstaltung setzt die „Allgemeine Informatik III“ voraus!

1.6 Am Rechner anmelden und abmelden

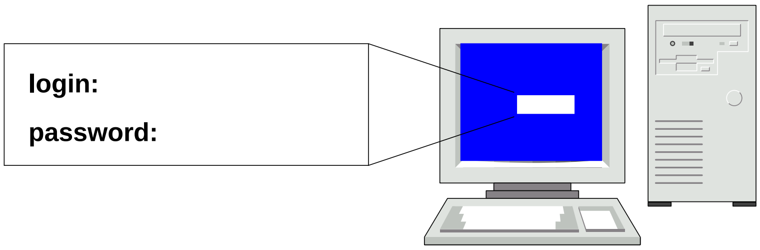


Abbildung 1.1: Workstation anmeldebereit

- ist der Bildschirm dunkel, so Maus bewegen
- ist der Monitor immer noch dunkel, so Monitor ggf. einschalten

Nach Eingabe der login-Namens und des Passworts (jeweils mit Enter/Return abschliessen) erscheint ein Bildschirm wie in Abb. 1.2:

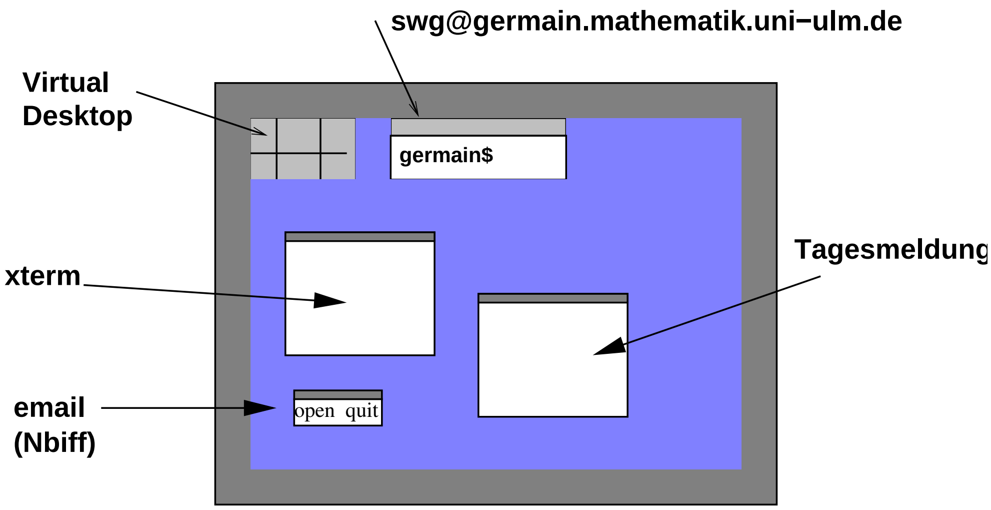


Abbildung 1.2: Workstation nach der Anmeldung

Ein xterm auf einem Rechner starten:

- Mit der Maus auf den blauen (Hintergrund-) Bereich gehen und rechte Maustaste kurz drücken oder gedrückt halten ⟶ Auswahlmenü **workspace**.

Abbildung 1.3: auf einen bestimmten Rechner gehen

- Gewünschten Rechner mit rechter Maustaste anklicken und es erscheint das Arbeitsfenster (xterm) für den Rechner **turing**
- Die Schrift ist wohl noch etwas klein auf diesem Fenster, also mit der Maus in dieses Fenster fahren (weiße Fläche) — dort die Taste “Control” (oder “Ctrl”, zu deutsch “Steuerung”) drücken und gedrückt halten, gleichzeitig rechte Maustaste drücken und gedrückt halten, so erscheint dort wieder ein Auswahlmenü (s.u.), in dem man durch Ziehen der Maus auswählen kann.
- Wer mehr über Eigenschaften und Möglichkeit eines “xterm” erfahren will, kann dies mit dem Kommando “**man** xterm” erfahren – dazu aber später mehr!

Abmelden: Mit der rechten Maustaste auf blauen Hintergurnd und auf *Exit* ziehen!

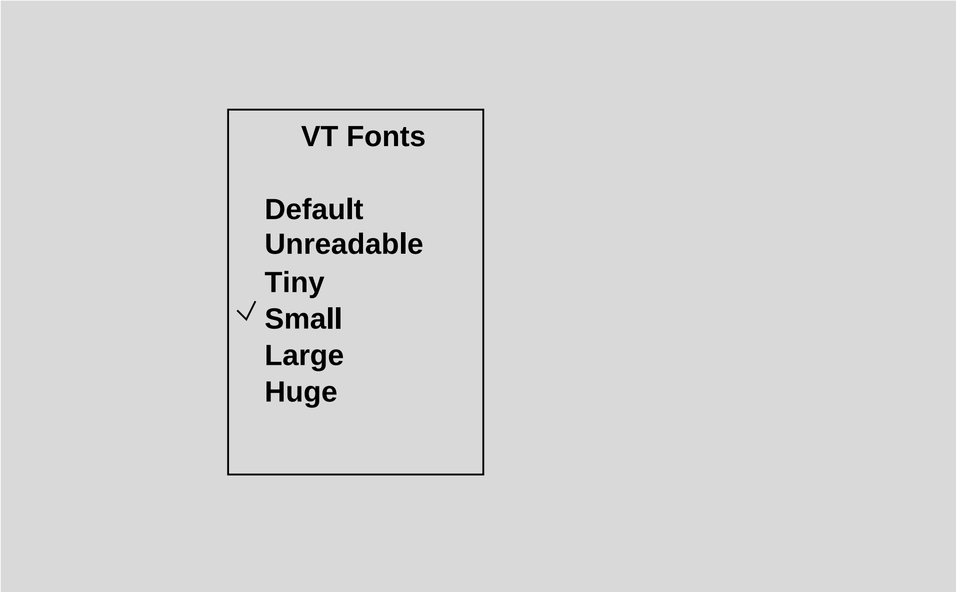


Abbildung 1.4: Schriftgröße ändern

1.7 Login von außerhalb

Von Uni-Pools ins WiMa Net:

- Zugangsberechtigungen müssen dort geholt werden
- Zugansgberechtigung für das WiMa Net braucht man dennoch!
- Auf dem jeweiligen Arbeitsplatz muss entsprechende Software installiert sein – am besten und sichersten ist **ssh** (*secure shell*), bei Win32-Systemen **PuTTY**
- Von einem Rechner in der E-Technik kann man mit

  ```
  ssh turing.mathematik.uni-ulm.de
  ```

  eine Shell auf der *turing* starten. Mehr dazu ist beim Betreuer des jeweiligen Rechners zu erfahren.
- Einige Wohnheime hängen direkt am Uni-Netz - auch von dort aus kann man auf das WiMa-Net!
- Auf dem Campus gibt es fast flächendeckend ein WLAN (*Wireless Local Area Network*), so dass mit einem enstprechenden Laptop "überall" auf unseren Rechnern arbeiten können `:-)`

Wenn Sie mehr wissen wollen:

- Fragen Sie Ihre Nachbarin / Ihren Nachbarn!
- Probieren Sie es einfach mal aus!
- Wenn nichts mehr geht:
  - Nicht einfach weggehen!
  - Auf keinen Fall den Rechner ausschalten!
  - Suchen Sie im Raum jemanden, der helfen kann!
  - Suchen Sie einen freien Arbeitsplatz und schicken Sie eine entsprechende Nachricht an **trouble**!

1.8 Sie sind willkommen

Bitte scheuen Sie sich nicht,

- mitten in der Vorlesung oder in den Übungen Fragen zu stellen,
- Ihren Tutor, den Übungsleiter Herrn Heidenbluth oder mich mit fragenden oder kommentierenden E-Mails zu überschütten und
- unsere Sprechstunden zu nutzen.

Eine erfolgreiche Vorlesungsveranstaltung ist nur möglich, wenn die Kommunikation beidseitig funktioniert.

So erreichen Sie mich (siehe auch Abteilungs-Homepage):

E-Mail:	franz.schweiggert (at) uni-ulm.de, franz (at) schweiggert.de
Büro:	Helmholtzstraße 18, Zimmer E04
Telefon:	0731/50-23570
Sekretariat:	Gisela Richter, -23571, E03, besetzt von 8-12 Uhr

So erreichen Sie Norbert Heidenbluth (s.a. Abteilungs-Homepage):

E-Mail:	norbert.heidenbluth (at) uni-ulm.de
Büro:	Helmholtzstraße 18, Zimmer E24
Telefon:	0731/50-23574

Bei Problemen in der Rechnerbenutzung wenden Sie sich bitte an

`trouble@mathematik.uni-ulm.de`

Nützliche Informationen finden Sie unter:

`www.mathematik.uni-ulm.de/sai/ws05/ai1/` (Vorlesungsseite)
`www.mathematik.uni-ulm.de/sai/` (SAI Homepage)
mit dem Link *Information For Our Students*

Darunter finden Sie unter "Weitere nützliche Informationen" z.Zt. folgende Links:

- Administratives, Tagesmeldungen, etc.
  Tagesmeldungen – etwas zum Thema Passwortwahl – Umgang mit zweifelhaften Mails – Infos zur Anpassung der Systemeinstellung – Hinweise für die ersten Schritte im Internet – Umgang mit rechen- / speicherintensiven Jobs – Thema Drucken – Dokumentation nützlicher Software-Pakete
- SSH: Useful Information and Downloads (auch für Win32-Systeme)

Kapitel 2

UNIX - die ersten Schritte

2.1 Betriebssysteme

2.1.1 Übersicht

> „Zum Betriebssystem zählen die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen“ (nach DIN 44300).

Hauptaufgaben:

- **Vermitteln** zwischen Benutzer/Benutzerprogramm und der Rechner-Hardware – Betriebssystem stellt alle Leistungen des Rechners den Anwenderprogrammen zur Verfügung, **schützt** aber gleichzeitig auch Hardware und Software vor unberechtigtem Gebrauch

- Verwalten der Ressourcen eines Rechners (**Resource Manager**):

  – Kontrolle aller Hardware- und Software-Komponenten eines Rechners und effiziente Zuteilung an die einzelnen Nachfragern

  – Stellt Basis-Dienstleistungen (Dateizugriff, Prozessmanagement) bereit, auf denen Anwendungsprogramme aufsetzen können

- Stellt abstrakte Sicht auf den Rechner bereit (eine **virtuelle Maschine**):

  – Eine (oder mehrere) Schichten von Software, die über der „nackten“ Hardware liegen

- Diese ist einfacher zu verstehen und zu programmieren, komplexe Hardware-Details verbergen sich hinter einfachen und einheitlichen Instruktionen

- Programmierer erhält vom Betriebssystem eine angenehmere Schnittstelle zur Hardware als von der reinen Hardware (Instruktionssatz, Register-Struktur, Speicher-Organisation, E/A-Struktur, Bus-Struktur, ...), er muss sich nicht mehr um sämtliche maschinenabhängige Details kümmern.

2.1.2 Etwas Geschichte

- Die Entwicklung von UNIX begann 1969 als Thompson eine wenig benutzte PDP-7 für die Entwicklung seines "Space Travel"-Projekts requirierte. Mehr dazu: `http://www.bell-labs.com/history/unix/pdp7.html`

- Für das Spiel wurde ein minimales Betriebssystem einschließlich einem Dateisystem benötigt. Damit wurde die Grundlage für UNIX gelegt.

- Seit den 80er Jahren gibt es den sogenannten POSIX-Standard für die UNIX-Umgebung, um möglichst viele Gemeinsamkeiten unter den vielen existierenden UNIX-Varianten zu pflegen (POSIX: *Portable Operating System for Computer Environments*. Die aktuelle Fassung davon ist der IEEE Standard 1003.1 in der Fassung von 2004 zu finden unter `http://www.opengroup.org/onlinepubs/009695399/toc.htm` (IEEE: *Institute of Electrical and Electronical Engineers*

- Es gibt viele populäre Implementierungen davon, zu denen beispielsweise Solaris gehört (leitet sich vom originalen UNIX ab), GNU/Linux, FreeBSD und die anderen BSD-Varianten. (GNU: rekursives Akronym für *GNU's Not UNIX*, siehe `www.gnu.org` – BSD: *Berkeley System / Software Distribution*

- Selbst für Microsoft Windows gibt es durch das **Cygwin**-Projekt eine POSIX-Umgebung – allerdings mit Einschränkungen.

2.1.3 Aufbau eines UNIX-Betriebssystems

- Der Erfolg von UNIX begründet sich auf die Verwendung von **wenigen** und sehr **einfachen** Abstraktionen.

- Abstraktion: eine möglichst einfache und flexible Schnittstelle zu einem Dienst – hinter der Schnittstelle können sich viele verschiedene komplexe Implementierungen verbergen.

- In folgenden Punkten bietet UNIX konkurrenzlos einfache Abstraktionen an:
  - hierarchischer Namensraum (mit einer Wurzel), bestehend aus vielen verschiedenen Dateisystemen
  - Dateien (einfache uninterpretierte Sequenz von Bytes)
  - Ein- und Ausgabeverbindungen funktionieren gleichermaßen für Dateien, interaktive Verbindungen zum Benutzer, zu Geräten und bei Netzwerkverbindungen
  - Ein sehr einfaches Rechtesystem, das im wesentlichen nur eine Benutzer-ID (UID), eine Gruppen-ID (GID) und eine Liste weiterer Gruppenzugehörigkeiten berücksichtigt
  - Aufrufschnittstelle für Programme.

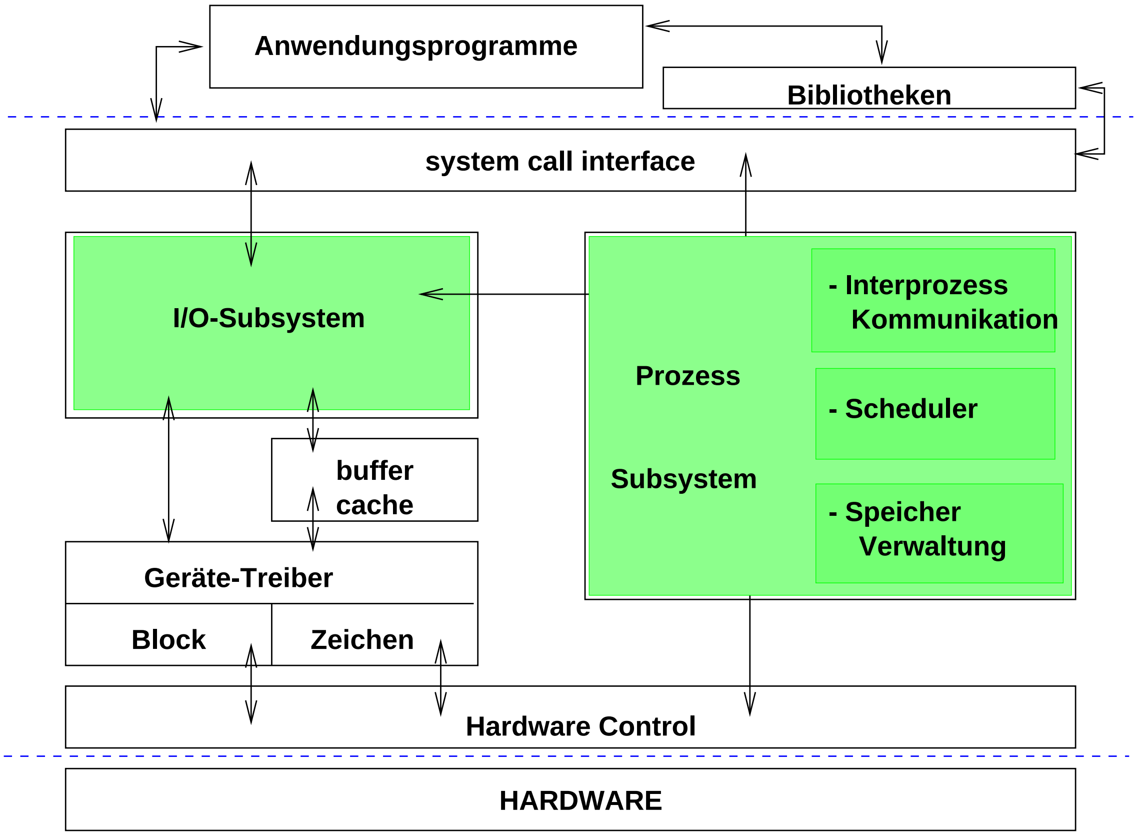


Abbildung 2.1: Architektur von UNIX

Das I/O-(Input-/Output)-Subsystem

Aufgaben:

- Verwalten der statischen Objekte (zentraler Begriff “Datei”)
- Zugriff auf Geräte / Speichermedien (Dateien, Mechanismen zur *InterProcess Communication*), Ein-/Ausgabe-Geräte wie Tastatur oder Bildschirm, Netzwerke über Datenstrukturen (Verwaltungstabellen, interne Datenstrukturen, Datenstrukturen zur Netzwerk-Kommunikation, z.B. *sockets*) ermöglichen
- Kontrolle der Zugriffsrechte (welcher Benutzer darf z.B. welche Dateien verändern?)
- Speichern und Wiederbeschaffen von Benutzerdaten
- Bereitstellen von Platten-Speicherplatz
- Verwalten von freiem Plattenplatz

Benutzt einen Puffermechanismus beim Zugriff auf Daten, die auf einem **block device** (z.B. Festplatte) lagern: **Buffer Cache**.

Block devices: Geräte, die Daten in beliebig adressierbaren Blöcken (z.B. in Einheiten von 4kByte) speichern und übertragen können (Beispiel: Festplatten)

Char devices: Geräte, auf die ungepuffert zugegriffen wird, gemäß dem Modell „unformatierter sequentieller Bytestrom“ (Beispiele: Terminal, Drucker).

Geräte-Treiber (*Device Driver* oder *Controller*) wie Video-Karten-Treiber, Netzwerk-Karten-Treiber, Platten-Controller, ..., heißen die Module, die die Operationen der physikalischen Geräte im Detail kontrollieren. Sie haben eine geräte-unabhängige Schnittstelle nach oben zu den übrigen Betriebssystem-Routinen und eine geräte-abhängige Schnittstelle nach unten zur Hardware hin.

Das Prozess-Subsystem:

Die Ausführung eines Programms heißt **Prozess**.

Dazu gehört mehr als nur der Programmtext und die Daten, die der Programmierer "geschrieben" hat; die Ausführungsumgebung eines Prozesses (oft auch als **Kontext** eines Prozesses bezeichnet) enthält Datenstrukturen zur Verwaltung der Dateiverbindungen dieses Prozesses oder Tabellen, die die Reaktion des Prozesses auf eintreffende Signale (Interrupts) definieren.

Aufgaben des Prozess-Subsystems:

- Prozesse managen (kreieren, terminieren, synchronisieren)

- Verschiedenen rechenwilligen Prozessen fair die CPU zuteilen (*Scheduling*)

- Kommunikation zwischen Prozessen (IPC — InterProcess Communication) ermöglichen, z.B. Protokolle der unteren Netzwerk-Schichten bereitstellen

- Hauptspeicher verwalten, Speicherschutz herstellen

Benötigt Dienstleistungen vom I/O-Subsystem, wenn ein Programm gestartet (vom Compiler generiertes Programm von Platte in Speicher laden) oder wenn Prozessdaten aus-/eingelagert werden müssen (*swapping*).

- **Speicher-Verwaltungs-Teil:**
  - kümmert sich um die Zuteilung des Hauptspeichers an die einzelnen Prozesse
  - übernimmt das Ein- und Auslagern von Prozessdaten vom Primärspeicher zum Sekundärspeicher und umgekehrt
- **Scheduler:**
  - teilt die CPU fair den einzelnen Prozessen zu
  - entscheidet anhand von Prioritäten und der Uhr (Zeitscheiben-Verfahren), welcher Prozess die CPU zugeteilt bekommt und damit ausgeführt wird
- **IPC-Teil:**
  - versorgt alle Prozesse mit Dienstleistungen zur Kommunikation untereinander: Signale (*interrupts*), *shared memory* (gemeinsam nutzbarer Hauptspeicherbereich), FIFO-Strukturen wie *pipes* (*First-In-First-Out*), Datenstrukturen und Kommunikationsendpunkte zur Netzwerk-Kommunikation (*sockets* und *ports*) u.a.m.

2.1.4 Das I/O-System von UNIX

Eine der Hauptaufgaben eines jeden DV-Systems ist die Datenspeicherung. In einem UNIX-System wird jede Art von Information in Dateien (**files**) gespeichert. Eine **Datei** ist schlicht ein Behälter für Information (Bytes) (*A file ist just a sequence of bytes*).

Dateiarten:

- Gewöhnliche oder reguläre Datei (*ordinary* oder *regular file*): Container für jede Art von Daten (Texte, Daten in ASCII; ausführbare Programme — binäre Daten; u.a.).

- Katalogdatei (*directory* oder *directory file*): Verzeichnis für die Namen von Dateien samt Verweisen auf eine Verwaltungsstruktur, in der vom Betriebssystem die wichtigen Informationen zu Dateien samt Verweisen auf ihren Speicherort geführt werden (*Inode List*, *file allocation table*).

- spezielle Katalogdateien:

  - **Arbeitskatalog** (*working directory*: Jeder Benutzer / Prozess hat zu einem bestimmten Zeitpunkt immer genau einen Katalog als aktuellen Arbeitskatalog zugeordnet

  - **Heimatkatalog** (*home directory*): Nach dem Anmelden erhält ein Benutzer diesen Katalog als ersten Arbeitskatalog zugewiesen, er "gehört" ihm!

- Gerätedatei (*device special file*): Geräte und Hardware allgemein, zu Dateien abstrahiert (Terminal, Drucker, Platten, Memory, Netzwerkkarte, Graphikkarte).

- Weitere Dateitypen z.B. zu IPC (*InterProcess Communication*), u.a.m.

Dateisystem

Ein **Dateisystem** (*File system*) ist eine geordnete Zusammenfassung von Dateien zu einer Einheit. UNIX kann mehrere Dateisysteme verwalten; ein spezielles Dateisystem ist das sog. **Root-File-System**: dieses wird beim Hochfahren des Betriebssystems als erstes bekannt gemacht, alle anderen müssen explizit geladen werden.

Den grundlegender Aufbau eines Unix-Dateisystems (UFS) zeigt schematisch Abb. 2.2 (S. 27)

Abbildung 2.2: Dateisystem aus logischer Sicht

- Der **boot block** enthält beim *Root-Filesystem* den sog. *bootstrap code*, der beim „Hochfahren" der Maschine zuerst geladen und ausgeführt wird; er dient zur Initialisierung des Systems, von diesem werden weitere Programmteile des Betriebssystems (liegen in normalen Dateien dieses Datei-Systems) gestartet.

- Der **super block** enthält Verwaltungsinformationen zu diesem Dateisystem

- Inode-Liste
  Eine Inode ist eine Datenstruktur, die alle relevanten Informationen zu einer Datei enthält (s.u.).

- Datenblöcke
  Diese enthalten in Einheiten z.B. von 1Kbyte die Dateiinhalte.

Das Dateisystem aus Benutzersicht stellt eine hierarchisch aufgebaute Ordnungsstruktur (beispielhaft in Abb. 2.3, Seite 28 dargestellt) über alle Dateien her.

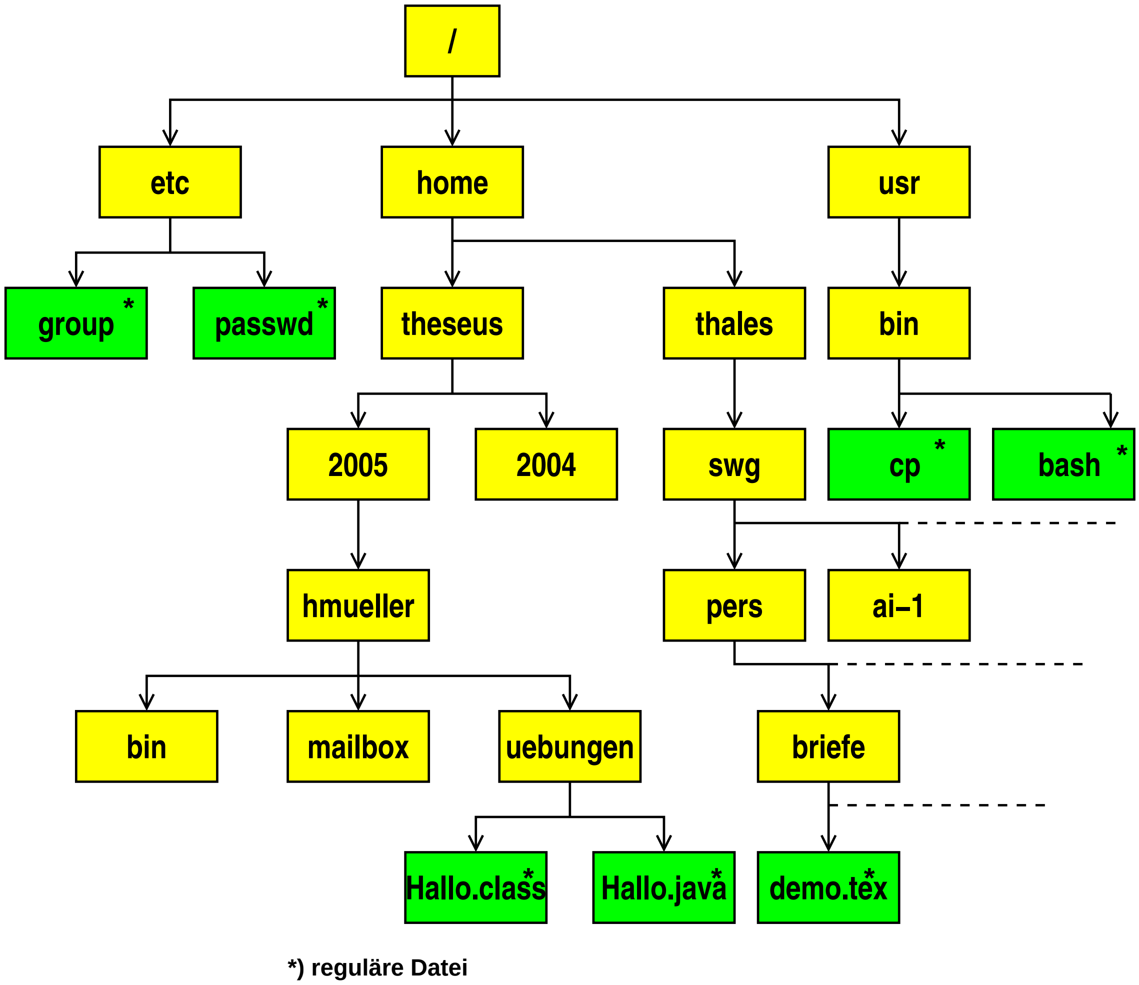


Abbildung 2.3: Dateisystem aus Benutzersicht

Dateinamen:

- sind als Eintrag in einem Katalog (Directory, Ordner) enthalten
  - fast „beliebige“ Zeichenfolgen, auf keinen Fall Schrägstrich:
    * Schrägstrich ist der Name des Wurzelkatalogs (*root*)
    * Schrägstrich trennt einzelne Namen in einer Pfadangabe (s.u.)
  - **Vorsicht** bei Zeichen mit Sonderbedeutung für die Shell
- **lokaler Name**: vom Katalog aus gesehen, in dem dieser Name eingetragen ist
  also wenn `/home/thales/swg/pers/briefe` der aktuelle Arbeitskatalog ist, so ist die darin enthaltene Datei `demo.tex` direkt mit `demo.tex` ansprechbar
- **absoluter Name** oder **Pfadname**: Auflistung der Dateien auf dem Weg von der Wurzel zu dieser Datei, die jeweiligen Dateinamen durch Schrägstrich getrennt, also

  `/home/thales/swg/pers/briefe/demo.tex`

- **relativer Name**: von einem Punkt im Dateibaum aus gesehen
  - da der aktuelle Katalog auch den Namen „ . “ (Punkt) hat, kann sie als `./demo.tex` angesprochen werden
  - da der Heimatkatalog eines jeden Users den Namen `$HOME` (genauer: `HOME`) bzw. den Namen ~ (Tilde) hat, kann die Datei vom User “swg” von jedem Punkt aus mit `$HOME/pers/briefe/demo.tex` bzw. mit
    `~/pers/briefe/demo.tex` angesprochen werden
  - Wenn der Arbeitskatalog der Katalog `/home/thales/swg/ai1/ws05` ist, so kann diese Datei mit `../../pers/briefe/demo.tex` angesprochen werden („ .. “ ist der Name des übergeordneten Katalogs)

Die Inode

- Datenstruktur, die alle wesentlichen Informationen zu einer Datei enthält

- Name einer Datei stellt lediglich eine Verbindung zu einer Inode her – dies erfolgt in der Katalogdatei (s.u.) durch einen Eintrag der Form

```
Dateiname        Inode-Nummer
```

- eine Datei kann mehrere Namen haben, die eindeutige Identifikation einer Datei erfolgt über die Inode, genauer über einen Index (Zahl) in die Liste der verfügbaren Inodes

- Inhalt einer Inode (nicht vollständig):

  - Identifikation des Besitzers der Datei
  - Identifikation der Besitzergruppe
  - Datei-Typ
  - Zugriffsrechte auf die Datei
  - Zeitstempel (zuletzt zugegriffen / modifiziert)
  - Anzahl der Namen für diese Datei
  - Größe der Datei
  - ...
  - Adressen der Datenblöcke dieser Datei

Directory / Katalogdatei:

Dateien vom Typ „Directory" dienen i.w. zwei Zwecken:

- Zusammenfassung verschiedener Dateien zu einer organisatorischen Einheit - damit entsteht die für den Benutzer relevante hierarchische Struktur (Baum) des Datei-Systems
- Verbindung von Dateinamen zu Datei, d.h. Zuordnung einer Inode-Nummer zu einem Katalognamen:

Dateiname	Inode
.	4711
..	3122
filename_1	4738
filename_2	4789

Der Name „ **.** " ist der Name des Katalogs selbst, der Name „ **..** " ist der Name des übergeordneten Katalogs — diese beiden Einträge sind in jedem Katalog (ausgenommen dem Wurzelkatalog) enthalten!

2.2 Shell

- Die wichtigste Schnittstelle unter UNIX für den Benutzer ist die **Kommandozeile**.

- Ein Programm, das eine interaktive Kommandozeile anbietet, wird unter UNIX **Shell** genannt.

- Bekannt sind insbesondere die *Bourne Shell* **sh** (praktisch unverändert seit ca. 1980), die *Korn Shell* **ksh** (entstand Ende der 80er Jahre) und die *Bourne Again Shell* **bash** aus dem GNU-Projekt. Zeitweilig populär war auch noch die C-Shell **csh**, die sich aber in der verwendeten Syntax deutlich von den anderen Shells unterscheidet.

- Per Voreinstellung starten bei uns neue Zugänge mit der **bash**.

Eine interaktive Shell ist unermüdlich darin,

- eine Eingabe-Aufforderung auszugeben (genannt Prompt, besteht bei uns per Voreinstellung aus dem Namen des Rechners, auf dem Sie gerade arbeiten, einem Dollar-Zeichen und einem Leerzeichen),

- eine Zeile einzulesen,

- dies als Aufruf eines Programms mit einigen Parametern zu interpretieren,

- das ausgewählte Programm mit den Parametern zur Ausführung zu bringen und

- darauf zu warten, dass das Program beendet ist.

Eine interaktive Shell endet nur dann, wenn die Eingabe endet oder sie explizit terminiert wird.

Aufgaben einer Shell

- Kommando-Zeilen-Interpretation (Variablen- / Kommando- / Dateinamen-Substitution, Ein-/Ausgabe-Umlenkung - mehr dazu später)

- Zerlegen der Zeile in einzelne Worte: Trenner ist ein oder mehrere Leerzeichen

- Starten von Kommandos

Eingabe unter UNIX

Wenn Sie unter UNIX interaktiv etwas eingeben, haben normalerweise einige Zeichen eine besondere Bedeutung:

- **BACKSPACE** löscht das zuletzt eingegebene Zeichen in der aktuellen Eingabezeile.
- **CTRL-u** (zuerst die "Control"-Taste drücken und während sie noch gedrückt bleibt, die Taste "u" drücken) löscht die gesamte Eingabezeile.
- **CTRL-w** löscht das zuletzt eingegebene Wort.
- **RETURN** beendet die aktuelle Eingabe mit einem Zeilentrenner.
- **CTRL-d** beendet die aktuelle Eingabe ohne einen Zeilentrenner. Geschieht dies zu Beginn einer Zeile (also ohne bislang eingetippten Zeileninhalt) wird dies als Eingabe-Ende interpretiert.
- **CTRL-c** sendet ein Signal an das gerade laufende Programm, das normalerweise zur vorzeitigen Terminierung führt.
- **CTRL-s** stoppt die Ausgabe, die sich danach nur mit **CTRL-q** wieder fortsetzen lässt.
- **CTRL-v** nimmt dem folgenden Zeichen die Sonderbedeutung.

All diese Funktionen können auf andere Zeichen gelegt werden. Hier sind nur unsere Voreinstellungen genannt.

Kommandozeile

- Eine Kommandozeile in einer Shell besteht im einfachsten Falle aus
  - dem Namen eines Programms und
  - beliebig vielen sogenannten Argumenten (oder Parametern).
- Der Programmname und die Argumente werden durch Leerzeichen (und Tabs) voneinander getrennt.

- Beispiel:

```
euclid$ cal
   September 2005
So Mo Di Mi Do Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

euclid$ cal 11 2005
    November 2005
So Mo Di Mi Do Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

@euclid$
```

Fehler in der Kommandozeile

- Die Mehrheit der zur Verfügung stehenden Programme geben kurze hilfreiche Hinweise (*Usage*-Meldung), wenn sie falsch aufgerufen worden sind.

- Beispiel:

```
euclid$ cal 2005 11
cal: bad month
usage: cal [ [month] year ]
euclid$
```

  Hier beschwert sich **cal** darüber, dass es keinen Monat mit der Nummer 2005 gibt.

- Die sogenannten Usage-Zeilen geben an, ob und wenn ja, was für Argumente (Parameter) erwartet werden. Die eckigen Klammern umfassen dabei optionale Argumente.

- Im konkreten Falle kann **cal** ganz ohne Argumente aufgerufen werden (liefert den Kalender für den aktuellen Monat) oder mit nur einem Argument (wird als Jahr interpretiert und liefert den Kalender für ein ganzes Jahr) oder mit zwei Parametern (Monat und Jahr; liefert dann den Kalender für den Monat im genannten Jahr).

Für die folgenden Beispiele gilt: Die Zeichen auf einer Zeile nach dem **#**-Zeichen (bis zum abschließenden **return**) (**enter**) sind Kommentar und werden von der Shell ignoriert.

Beispiele: date pwd ls

```
turing$ # Kommentar - ohne Bedeutung
turing$ date  # Datum und Uhrzeit?
Mon Sep 22 16:42:23 MEST 2003
turing$ pwd   # In welchem Katalog bin ich?
/home/swg/allgInfo/2/bsp
turing$ ls    # Eintraege im aktuellen Katalog?
typescript
turing$ ls -l # mit etwas mehr Info
total 0
-rw-r--r--   1 swg      users           0 Sep 15 19:18 typescript
turing$ ls -al # ???
total 2
drwxr-xr-x   2 swg      users        1024 Sep 15 19:18 .
drwxr-xr-x   4 swg      users        1024 Sep 15 19:18 ..
-rw-r--r--   1 swg      users           0 Sep 15 19:18 typescript
turing$
```

Bedeutung der Ausgabe des `ls`-Kommandos:

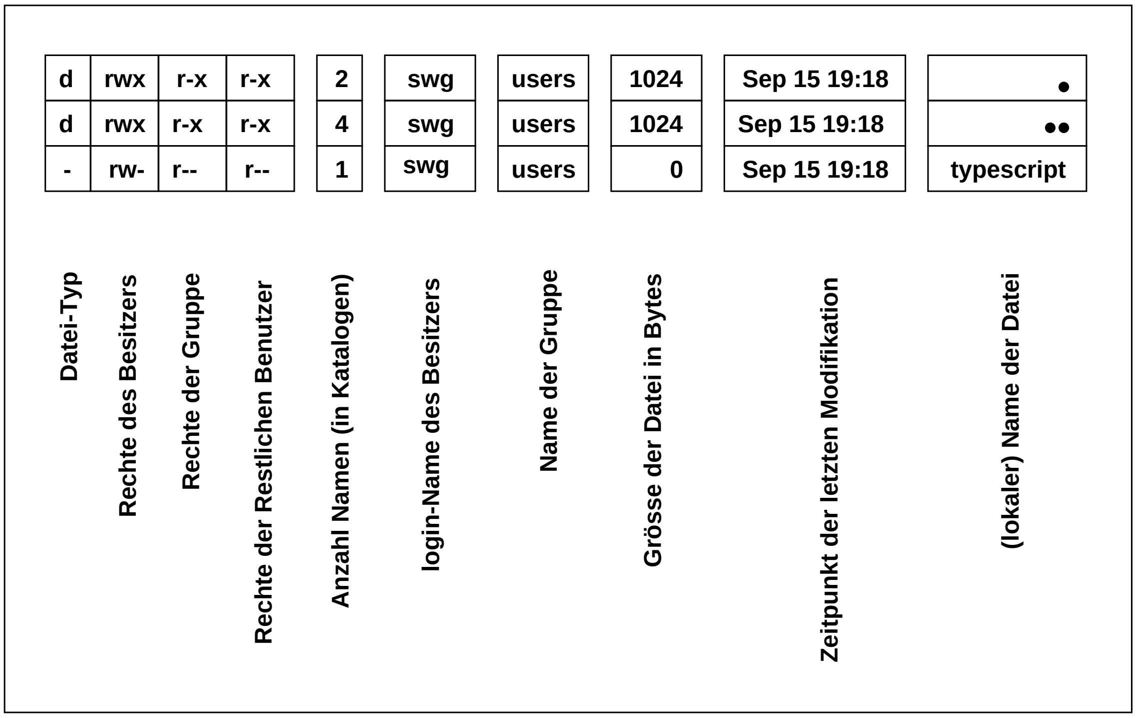


Abbildung 2.4: ls — Dateiattribute

Mit:

d	Datei-Typ Directory
-	bei Datei-Typ: reguläre Datei bei Rechten: das an dieser Stelle stehende Recht ist **nicht** gesetzt
r	Recht zu Lesen (read)
w	Recht zu Schreiben bei regulären Dateien Recht eine Datei anzulegen bei Directories
x	bei regulären Dateien: ausführbar bei Directories: darf durchsucht werden

Zeichen mit Sonderbedeutung

Es gibt eine ganze Reihe von Zeichen (auch Zeichenpaare), die für die Shell eine Sonderbedeutung haben: sie werden von ihr **interpretiert**.

Dazu zählen Zeichen wie “Leertaste” oder “Tabulator” sowie:

<table><tr><td>( )</td><td>{ }</td><td>?</td><td>*</td><td>" "</td><td>' '</td><td>` `</td><td><</td><td><<</td><td>></td><td>>></td><td>|</td><td>#</td><td>$</td><td>\</td><td>return</td></tr></table>

Welche Bedeutung diese im Einzelnen haben, werden Sie so nach und nach kennenlernen. Vorläufig nur verwenden, wenn die Bedeutung klar ist!

Beispiel:

```
turing$ date   " ein versehentlicher
> Apostroph
> und nun ?
> einfach noch einen "

date: bad conversion
turing$
```

Ein Paar von doppelten Apostrophen oder einfachen Apostrophen verhindern die Interpretation der Zeichen dazwischen — das Paar einfacher Apostrophen die aller Zeichen, das Paar doppelter die von fast allen. Insbesondere das `return`-Zeichen wird nicht mehr als Abschluß des Kommando’s erkannt, die Shell erwartet die Fortsetzung auf der nächsten Bildschirmzeile und deutet dies mit dem sog. Sekundärprompt „>“

Noch ein Beispiel:

```
euclid$ ls -l
insgesamt 0
-rw-r--r--    1 swg      users           0 2005-09-10 11:52 $
-rw-r--r--    1 swg      users           0 2005-09-10 11:52 $x
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 ?
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 a
-rw-r--r--    1 swg      users           0 2005-09-10 11:40 anton huber
-rw-r--r--    1 swg      users           0 2005-09-10 11:41 brief-01
-rw-r--r--    1 swg      users           0 2005-09-10 11:53 typescript
euclid$ ls -l anton huber
ls: anton: Datei oder Verzeichnis nicht gefunden
ls: huber: Datei oder Verzeichnis nicht gefunden
euclid$ ls -l "anton huber"
-rw-r--r--    1 swg      users           0 2005-09-10 11:40 anton huber
euclid$ ls -l ?
-rw-r--r--    1 swg      users           0 2005-09-10 11:52 $
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 ?
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 a
euclid$ ls -l "?"
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 ?
euclid$ ls -l $x
insgesamt 0
-rw-r--r--    1 swg      users           0 2005-09-10 11:52 $
-rw-r--r--    1 swg      users           0 2005-09-10 11:52 $x
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 ?
-rw-r--r--    1 swg      users           0 2005-09-10 11:50 a
-rw-r--r--    1 swg      users           0 2005-09-10 11:40 anton huber
-rw-r--r--    1 swg      users           0 2005-09-10 11:41 brief-01
-rw-r--r--    1 swg      users           0 2005-09-10 11:53 typescript
euclid$ ls -l "$x"
ls: : Datei oder Verzeichnis nicht gefunden
euclid$ ls -l '$x'
-rw-r--r--    1 swg      users           0 2005-09-10 11:52 $x
euclid$
```

Erläuterungen??? Dazu gibt's die Vorlesung!

2.3 Einige UNIX-Kommandos

Philosophie dieser Kommandos: Was der Benutzer möchte, wird klaglos erledigt. Er ist intelligent und weiß, was er möchte.

2.3.1 Informationen zu einem Kommando — man

- Für (fast) alle Kommandos gibt es sogenannte Manual-Seiten, die kurz und präzise jeweils ein Kommando mitsamt all seinen Aufrufmöglichkeiten erklären.
- Diese lassen sich am einfachsten mit dem **man**-Kommando abrufen. Als Argument wird dabei der Name des Kommandos übergeben, zu dem die Dokumentation gewünscht wird.
- Beispiel: `man cal` (gekürzte Ausgabe unter Solaris)

```
NAME
     cal - display a calendar

SYNOPSIS
     cal [ [month] year]

DESCRIPTION
     The cal utility writes a Gregorian calendar to standard
     output.  If the year operand is specified, a calendar
     for that year is written. If no operands are  specified,
     a  calendar for the current month is written.

OPERANDS
     The following operands are supported:

     month Specify the month to be displayed,  represented
           as a decimal integer from 1 (January) to 12
           (December). The default is the current month.

     year  Specify the year for which the calendar is
           displayed, represented as a decimal integer from
           1 to 9999. The default is the current year.
```

Versuchen Sie erst gar nicht, dies auszudrucken!!!

Aufbau einer Manual-Seite (man page)

- **NAME** zeigt den Namen des Kommandos zusammen mit einem Einzeiler, der das Kommando beschreibt.
- **SYNOPSIS** gibt die Aufruf-Syntax des Programms an analog zur Usage-Meldung.
- **DESCRIPTION** beschreibt detailliert das Kommando und spezifiziert genau wie die Argumente interpretiert werden.

- **SEE ALSO** verweist auf andere Manual-Seiten, die in diesem Kontext interessant sind.
- **BUGS** beschreibt Einschränkungen und verbliebene bekannte Probleme des Kommandos.

Betrachten einer Manual-Seite

- Wenn Sie bei uns mit dem **man**-Kommando eine Manualseite betrachten, landen Sie automatisch in einem Programm, mit dem Sie den Text seitenweise betrachten können.

- Als Seitenbetrachter kommt bei uns per Voreinstellung das Programm **less** zum Zuge.

- Wichtige interaktive Kommandos innerhalb von **less**:

q	steht für "quit" und beendet die Ausführung von **less**.
SPACE	zeigt die nächste Seite an.
b	geht eine Seite zurück ("back").
RETURN	geht eine nur eine Zeile weiter.
h	liefert einen Überblick weiterer Kommandos von **less**.

Zum Kommando **man** gibt es natürlich auch eine Manualseite: `man man`

2.3.2 Dateiinhalt auflisten: cat

```
NAME
   cat - concatenate files and print on the standard output
SYNOPSIS
   cat [-benstuvAET] [--number] [--number-nonblank]
   [--squeeze-blank] [--show-nonprinting] [--show-ends]
   [--show-tabs]  [--show-all] [--help] [--version]
   [file...]
DESCRIPTION
   cat writes the contents of each given file, or the
   standard input  if none are  given  or when a file
   named '-' is given, to the standard output.
   ...
```

2.3.3 Katalog wechseln: cd

```
NAME
   cd - Change working directory
SYNOPSIS
  cd [ dirName ]
DESCRIPTION
  Change the  current working directory  to dirName, or to
  the home directory (as specified in the HOME environment
  variable) if dirName is not given.
  ...
```

2.3.4 Dateien kopieren: cp

```
NAME
   cp - copy files
SYNOPSIS
   cp source dest
   cp source { source } directory
DESCRIPTION
   If the last argument names an existing directory, cp
   copies  each  other  given file into a file with the
   same name in that directory (Attention: if dest file
   already exists, it will be overwritten without warn-
   ing!).  Otherwise, if only two  files are given,  it
   copies the first onto the second. It is an  error if
   the  last argument  is not a directory and more than
   two  files are given. By default,  it does  not copy
   directories.
   ...
```

2.3.5 Plattenplatzbelegung ermitteln: du

```
NAME
       du - estimate file space usage

SYNOPSIS
       du [OPTION] ... [FILE] ...

DESCRIPTION
       Summarize  disk usage of each FILE, recursively for directories.

       ...
```

2.3.6 Katalog erzeugen: mkdir

```
NAME
   mkdir - make directories
SYNOPSIS
   mkdir dir { dir }
DESCRIPTION
   mkdir  creates  a  directory  with  each  given  name.
   By  default,  the  mode of created directories is 0777
   minus the bits set in the umask (see umask).
   ...
```

2.3.7 Dateien umbenennen / verschieben: mv

```
NAME
   mv - rename files
SYNOPSIS
   mv source dest
   mv source { source } directory
DESCRIPTION
   If the last argument names an existing directory,  mv
   moves each other given file into a file with the same
   name in that directory. Otherwise, if only two  files
   are given, it moves the first onto the second. It  is
   an error if the last argument is not a directory  and
   more  than  two  files  are  given.  It can move only
   regular files across filesystems.
   ...
```

2.3.8 Datei byteweise ausgeben: od

```
NAME
   od - dump files in octal and other formats
SYNOPSIS
   od  [-bcloxv]  [...] [file...]
DESCRIPTION
   od writes to the standard output the contents of
   the given files, or of the standard input if the
   name  `-'  is  given.
   ...
```

2.3.9 Arbeitskatalog anzeigen: pwd

```
NAME
   pwd - print name of current/working directory
SYNOPSIS
   pwd
DESCRIPTION
   pwd prints the fully resolved name of the current
   directory. pwd is a shell builtin.
```

2.3.10 Datei löschen: rm

```
NAME
   rm - remove files
SYNOPSIS
   rm name { name }
DESCRIPTION
   rm removes  each specified file. By default, it
   does  not  remove  directories. If once removed
   there is no way back! Attention with an asterix
   (*) on this command line!
   ...
```

VORSICHT: Was weg ist, ist weg!

2.3.11 Katalog entfernen: rmdir

```
NAME
   rmdir - remove empty directories
SYNOPSIS
   rmdir dir { dir }
DESCRIPTION
   rmdir removes each given empty  directory. If
   any  argument  does  not refer to an existing
   empty directory, it is an error.
   ...
```

2.3.12 Dateien komprimieren und verpacken: zip

```
NAME
       zip,  zipcloak,  zipnote,  zipsplit - package and compress
       (archive) files

SYNOPSIS
       zip  [-aABcdDeEfFghjkKlLmoOqrRSTuvVwXyz!@$]  [-b path]
       [-n suffixes] [-t mmddyyyy] [-tt mmddyyyy] [ zipfile [
       file1 file2 ...]] [-xi list]

       zipcloak [-dhL] [-b path] zipfile

       zipnote [-hwL] [-b path] zipfile

       zipsplit [-hiLpst] [-n size] [-b path] zipfile

DESCRIPTION
       zip is a compression and file packaging utility for  Unix,
       VMS,  MSDOS, OS/2, Windows NT, Minix, Atari and Macintosh,
       Amiga and Acorn RISC OS.

       It is analogous to a  combination  of  the  UNIX  commands
       tar(1)  and compress(1) and is compatible with PKZIP (Phil
       Katz's ZIP for MSDOS systems).

       A companion program  (unzip(1L)),  unpacks  zip  archives.
       The zip and unzip(1L) programs can work with archives produced
       by PKZIP,  and  PKZIP  and  PKUNZIP  can  work  with
       archives  produced  by zip.  zip version 2.3 is compatible
       with PKZIP 2.04.  Note that PKUNZIP  1.10  cannot  extract
       files  produced  by  PKZIP  2.04  or zip 2.3. You must use
       PKUNZIP 2.04g  or  unzip  5.0p1  (or  later  versions)  to
       extract them.

       For a brief help on zip and unzip, run each without specifying
       any parameters on the command line.

       ...
```

2.3.13 Konventionen auf der Kommandozeile

Viele Programme unterstützen folgende Konventionen:

- Die ersten Argumente sind **Optionen** , d.h. optionale Argumente. Diese werden üblicherweise mit einem Bindestrich eingeleitet und von einem Buchstaben gefolgt. Beispiel: `ls -l` oder `wc -l`
- Wenn mehrere Optionen angegeben werden, können diese zusammengefasst oder auch getrennt voneinander angegeben werden.
  Beispiel: `ls -a -l`
  (`-a` bittet **ls** darum, alle Namen aufzuzählen, auch solche die mit einem Punkt beginnen).
  Alternativ: `ls -la`
- Nach den Optionen folgen Dateinamen, aus denen das Programm einliest. Wenn keine Dateinamen angegeben wird, erfolgt die Eingabe aus der Standard-Eingabe. Beispiel: `cat`
- Wenn der erste Dateiname mit einem Bindestrich beginnt, kann mit einem vorausgehenden doppelten Bindestrich `--` die Interpretation als Dateiname erzwungen werden.
  Beispiel: `rm -- -x` (Löscht die Datei mit dem Namen “-x”).

Für den Interessierten an weiteren Kommandos: man intro

2.4 Shell: Dateinamen-Substitution

`*`	wird im gegebenen Kontext durch beliebige viele Zeichen (auch keins) ersetzt, so dass Namen existierender Dateien entstehen
`?`	wird im gegebenen Kontext durch genau ein Zeichen ersetzt, so dass Namen existierender Dateien entstehen
`[ ... ]`	eines der Zeichen in den eckigen Klammern wird genommen, um im gegebenen Kontext einen Namen einer existierenden Datei zu erzeugen

Achtung: Ein `rm` mit einem darauffolgenden Wort bestehend aus einem `*` löscht alle Dateien – nimmt man zu `rm` noch die Option `-r` hinzu, ist u.U. noch viel mehr weg!

2.5 Einige Shell-Variablen

Die Shell enthält eine Vielzahl von Variablen, deren Wert von diversen Kommandos benutzt wird - wer deren Bedeutung nicht kennt, sollte diese Werte tunlichst **nicht** verändern!

Variable	Bedeutung
HOME	Pfadname des Heimatkatalogs
PATH	Sammlung von Pfaden, in denen nach einem auszuführenden Kommando gesucht wird
HOSTNAME	Name des Rechners
PRINTER	Name des Standarddruckers

Welchen Wert hat die Variable? (**Shell-Variablen-Substitution**)

```
hypatia$ echo $HOME
/home/swg
hypatia$ echo $PATH
/home/swg/bin:/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin
:/usr/openwin/bin :/usr/lib/java/bin:/usr/games/bin:/usr/
games:.:/usr/bin/TeX
hypatia$
```

2.6 stdin – stdout – stderr

Die meisten UNIX-Kommandos sind so gebaut, dass sie — wenn beim Aufruf nichts anderes bestimmt wird — von der **Standardeingabe**(stdin) (Bildschirm) lesen, ihr Ergebnis auf die **Standardausgabe** (stdout) (Bildschirm) sowie Fehlermeldungen auf die **Diagnoseausgabe** (stderr) (Bildschirm) schreiben.

"Bildschirm"	Identifikator
Standardeingabe	0
Standardausgabe	1
Diagnoseausgabe	2

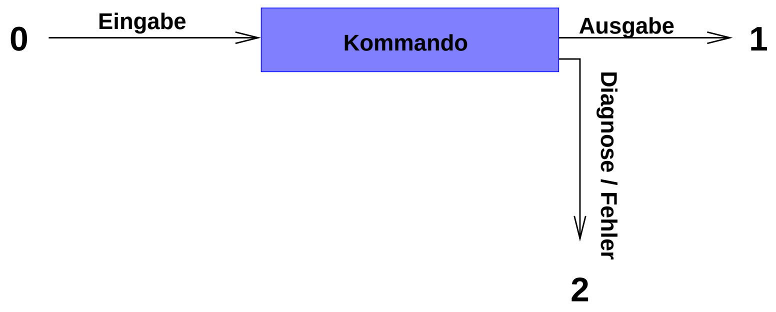


Abbildung 2.5: Filter — Programm

2.7 I/O-Umlenkung

Symbol	Bedeutung	Beispiel
<	stdin von "Datei nehmen"	`cat < file`
>	stdout auf "Datei legen" Achtung: diese Datei wird vorher "gelöscht"!	`cat file > newfile`
>>	stdout an "Datei anfügen"	`cat file >> newfile`
2>	stderr auf Datei legen (wie >)	`cat file 2>error`
2>>	wie >> für stderr	

Diese Umlenkung führt die Shell durch, **bevor** sie das Kommando zur Ausführung bringt!

Achtung bei >:

```
euclid$ cat catfile
Von der Stirne heiss
rinnen muss der Schweiss
soll das Werk den Meister loben ...
euclid$ catfile
cat: catfile: Eingabedatei und Ausgabedatei sind gleich
euclid$ cat catfile
euclid$
```

2.8 Pipes & Filters

Pipes (FIFO - *First–In–First–Out*) werden von der Shell erzeugt, um eine Verbindung zwischen zwei Kommandos herzustellen:

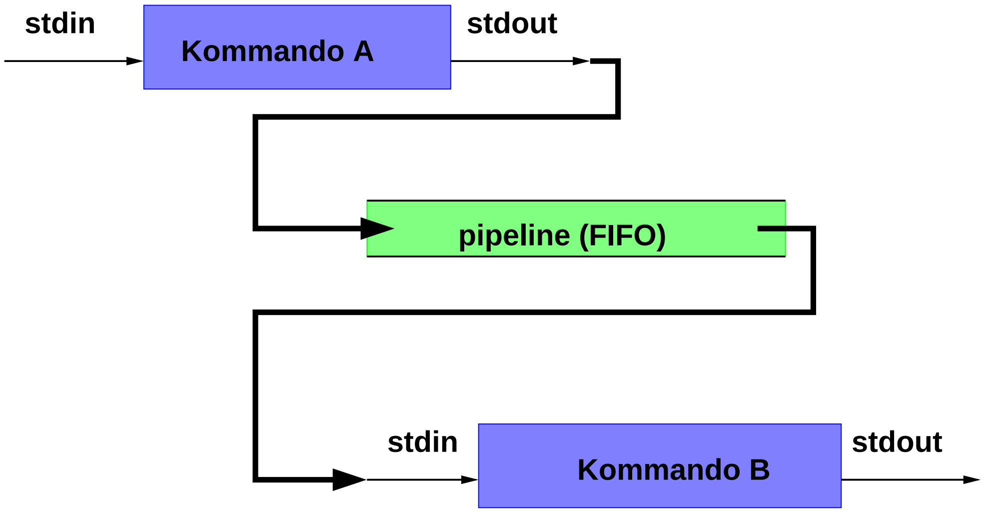


Abbildung 2.6: Pipeline zwischen Kommandos

Diese Umlenkung der Standardausgabe eines Programmes in Standardeingabe eines anderen Programmes nimmt die Shell vor:

```
Kommando_A | Kommando_B
```

Man könnte sagen, dass `Kommando_B` entsprechend seiner Funktionalität etwas aus der Ausgabe von `Kommando_A` heraus**filter**t (**Filter**)!

2.9 Hintergrund / Vordergrund

Eine Kommandozeile wird mit einem **return** abgeschlossen. Die Shell nimmt zunächst Substitutionen (z.B. Dateinamen-Substition) und ggf. I/O-Umlenkungen (Datei, Pipes) vor, bringt dann das Kommando (das danach verbleibende erste Wort) mit seinen Optionen und Argumenten zur Ausführung und wartet auf dessen Beendigung — dies ist daran zu erkennen, dass die Shell ihren **Prompt** erst nach Beendigung des gestarteten Programm wieder liefert, also damit die Aufforderung zur erneuten Eingabe liefert. Man sagt, dass Shell und Kommando **synchron** laufen!

Enthält die Kommandozeile vor dem *return* ein **&**, so wird das Kommando wie oben gestartet, die Shell wartet aber **nicht**, sondern liefert sofort wieder ihren Prompt (davor gibt sie allerdings noch eine Zahl aus, die **Prozessnummer** des gestarteten Kommandos). Man sagt, die Shell und das Kommando laufen **asynchron**!

Beispiel: Eine Datei `zahlen` soll numerisch (Option `-n`), absteigend (revers, Option `-r`) sortiert werden, das Ergebnis soll in der Datei `zahlen` stehen (Output, Option `-o`):

```
euclid$ cat zahlen
50
1
100
13
25
10
euclid$ sort -nr -o zahlen zahlen &
[1] 3234
euclid cat zahlen
100
50
25
13
10
1
euclid$
```

Anmerkung:

- Die Zahl in eckigen Klammern (hier 1) ist die Nummer des Jobs, der im Hintergrund läuft
- Die andere Zahl (hier 3234) ist die Nummer des Prozesses – falls der Job eine Pipeline darstellt, ist es die Nummer des "hintersten" Prozesses!

Beispiel: Datei `zahlen` sortieren und die ersten 5 Zeilen des Ergebnisses ausgeben (Kommando **head**):

```
euclid$ cp zahlen.org zahlen
euclid$ sort zahlen | head -5 &
[1] 3270
euclid$ 1
10
100
13
25
```

Anmerkung: Wie man hier leicht sieht, "streiten" sich Shell und das Kommando (hier das letzte in der Pipe) um den Ausgabe-Bildschirm! Die Zeile `[1] 3270` enthält Jobnummer und Prozessnummer (von `head`), die die Shell ausgibt – die Zahl `1` kommt von `head`, der Prompt von der Shell, die weiteren Zahlen kommen von `head`!

2.10 Kommandos abbrechen: ctrl-c — ps — kill

Beispiele:

```
hypatia$ cat   # auszugebende Datei vergessen
^c
hypatia$
```

In diesem Aufruf wartet **cat** lange auf Eingaben (so lange bis das Ende der Eingabe erreicht ist - s.u.) - durch Eingabe von **Control-c** (Tasten `control` und `c` gleichzeitig gedrückt — häufig dargestellt als `^c`) wird das laufende Programm abgebrochen!

```
hypatia$ cat #stdin ausgeben
xxxxxxxxxxx
xxxxxxxxxxx
^d
hypatia$
```

Control-d (dargestellt als `^d`) signalisiert Ende der Eingabe!

Es kommt immer wieder — vor allem bei Anfängern — vor, dass Programme ungewollt **endlos** laufen:

```
thales$ cat endlos.c
int main() {
  while (1);
}
thales$ gcc -Wall -o endlos endlos.c
thales$ endlos
^c
thales$ endlos
^\ Quit (core dumped)
thales$
```

Hinter der Eingabe von `ctrl-c` oder `ctrl-\` verbirgt sich die Erzeugung von **Signalen**, die vom Betriebssystem dem Prozess zugestellt werden; die Reaktion darauf ist die Termination!

Erzeugung von Signalen — Kommando kill:

```
NAME
   kill - terminate a process
SYNOPSIS
   kill [ -s signal | -p ]  [[ --aa ]] pid ...
   kill -l [ signal ]
DESCRIPTION
   kill sends the specified signal to the specified process.
   If no signal is specified, the TERM signal is  sent. The
   TERM  signal will kill processes which do not catch this
   signal. For other processes, if may be necessary to use
   the KILL (9) signal, since this signal cannot be caught.
   Most modern shells have a builtin kill function.
OPTIONS
   pid ...
      Specify the list of processes that kill should signal.
      Each pid can be one of four things.  A  process  name
      in which case processes called that will be signaled.
      n where n is larger than 0.  The process with pid n
      will be signaled.  -1 in which case all processes
      from MAX_INT to 2 will  be  signaled,
      as  allowed  by  the  issuing  user.  -n where n is
      larger than 1, in which case processes  in  process
      group  n  are signaled.   IF a negative argument is
      given the signal must be specified first, otherwise
      it will be taken as the signal to send.

   -s Specify  the  signal  to  send.   The signal may be
      given as a signal name or number.

   -p Specify that kill should only print the process  id
      (pid)  of the named process, and should not send it
      a signal.

   -l Print a list of signal names.  These are  found  in
      /usr/include/signal.h
```

Mit dem Kommando **kill** können Signale (identifiziert über eine **Signalnummer**) gezielt an Prozesse verschickt werden, an Vordergrund-Prozesse allerdings nur von einem anderen Fenster aus. Man benutzt es im allgemeinen, um Hintergrundprozesse zu terminieren.

Das folgende Beispiel zeigt ein Programm, das selbst keine Möglichkeit zur Beendigung anbietet (so was soll's geben!):

```
hypatia$ xtel &
1842
hypatia$  ps
  PID TTY STAT TIME COMMAND
  140   1 S    0:00 -bash
  200  p1 S    0:00 -bash
  340  p0 S    0:02 gv out
 1808  p0 S    0:00 vi kap.2
 1833  p2 S    0:00 bash -i
 1842  p2 S    0:00 xtel
 1845  p2 R    0:00 ps

hypatia$ ps | grep xtel
 1842  p2 S    0:00 xtel
 1849  p2 S    0:00 grep xtel

hypatia$ kill 1842
Terminated              xtel
hypatia$
```

Anmerkungen:

- Das **&** am Ende der Kommandozeile veranlasst die Shell, das Kommando als Hintergrund-Kommando auszuführen und nicht auf seine Termination zu warten.
- Die Shell gibt eine Nummer (hier: 1842) aus - diese identfiziert den gestarteten Prozess eindeutig (**PID –** *Process ID*).
- Falls man diese nicht mehr weiß, so kann man mit dem Kommando **ps** etwas über die laufenden Prozesse erfahren, insbesondere deren PID! Da hier sehr viele Informationen kommen, filtern wir mit dem Suchprogramm **grep** (siehe `man grep`) das richtige heraus!
- Wenn `kill` wie im Beispiel angegeben nicht zur Termination des identifizierten Prozesses führt, dann hilft

  `kill -9 1842`

  Wenn das nicht hilft, so haben Sie versucht einen Prozess zu terminieren, der Ihnen überhaupt nicht gehört!

Anwendung:

Sie haben ein Fenster mit einer Shell, bei der nichts mehr geht; öffnen Sie auf dem gleichen Rechner erneut ein Fenster mit einer Shell; deren PID läßt sich z.B. über die Shell-Variable `$` mit

`echo $$`

herausfinden; jetzt versuchen wir mit `ps` herauszufinden, welche Shells noch laufen und “schießen” (kill) diese ab.

Kapitel 3

Technische Grundlagen

3.1 Bits & Bytes

- Elementare Speichereinheit eines Rechners: zwei-wertig (z.B. über / unter Schwellwert)

- Elementare Informationseinheit – das **Bit** – ebenfalls zweiwertig (*binär*); oft dargestellt als **falsch / wahr** oder **0 / 1**

- Acht Bits zusammengefasst zu neuer Einheit: **Byte**

- Rechnerabhängig werden 16 oder 32 Bits (2 oder 4 Byte) zu einer adressierbaren Speichereinheit (**Wort**) zusammengefaßt (16-Bit-, 32-Bit-, 64-Bit-Rechner)

- Ein Byte $\hookrightarrow$ 256 verschiedene Kombinationen der Bit-Belegungen

- verwendet zur rechnerinternen Darstellung – Codierung – von Zeichen (Buchstaben, Ziffern, Interpunktionszeichen, Sonder- und Steuerzeichen)

- Zum Austausch so codierter Zeichen zwischen Rechnern ist eine gleiche Interpretation der Belegung eines Bytes (Folge der Bit-Werte) notwendig:

  - **ASCII-Code**: American Standard Code of Information Interchange, 1968 als 7-Bit-Code für 128 Zeichen definiert, erweitert zu einem 8-Bit-Code (**ISO 8859-1**)
    — PC's, Workstations
  - **EBCDIC**: Extended Binary-Coded Decimal Interchange Code
    — Großrechner, auch mittlere Rechner)
  - heute: Uni-Code (später)

ASCII (ISO 8859-1)

- Festlegung der Byte-Belegung (Codierung) von Buchstaben, Ziffern, Sonder- und Steuerzeichen
- Festlegung einer Ordnung auf der Menge der Zeichen

Anmerkung: Leider halten sich nicht alle Hersteller (von Software) an ISO 8859-1, insbesondere bei nationalen Zeichen (z.B. deutsche Umlaute), was dann beim Austausch von Daten zu merkwürdigen Darstellungen führt.

Byte-Belegung kann auch als Dualzahl (Basis 2) interpretiert werden.

Dezimalzahlen:

- 6037 wird gelesen als $6 \times 10^3 + 0 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$
  10^0 ist definiert als 1!
- Basis: 10 — Ziffern: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$

Dualzahlen:

- Basis: 2 — Ziffern: $0, 1$
- Beispiel:

Ziffern	1	1	0	0	1
Gewicht	2^4	2^3	2^2	2^1	2^0
Potenz	4	3	2	1	0

- Dezimal ausgewertet:
  $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 25$

Oktalzahlen:

- Basis: 8 — Ziffern: $0, 1, 2, 3, 4, 5, 6, 7$
- Übergang von Dualzahl zu Oktalzahl durch Bilden von Dreier-Gruppen von rechts her:

Dual	01	101	110
Oktal	1	5	6

- Übergang von Oktal nach Dual: Umsetzen der einzelnen Ziffern in Dual

Oktal	7	2	3
Dual	111	010	011

Hexadezimalzahlen:

- Basis: 16 — Ziffern: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$
- Übergang von Dual nach Hexadezimal durch Bilden von Vierer-Gruppen von rechts her:

Dual	0110	1110
Hex	6	E

ASCII-Tabelle (Ausschnitt):

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
0	0	0x00	NUL(\0)	053	43	0x2B	+	0126	86	0x56	V
01	1	0x01	SOH	054	44	0x2C	,	0127	87	0x57	W
02	2	0x02	STX	055	45	0x2D	-	0130	88	0x58	X
03	3	0x03	ETX	056	46	0x2E	.	0131	89	0x59	Y
04	4	0x04	EOT	057	47	0x2F	/	0132	90	0x5A	Z
05	5	0x05	ENQ	060	48	0x30	0	0133	91	0x5B	[
06	6	0x06	ACK	061	49	0x31	1	0134	92	0x5C	\
07	7	0x07	BEL	062	50	0x32	2	0135	93	0x5D	]
010	8	0x08	BS(\b)	063	51	0x33	3	0136	94	0x5E	^
011	9	0x09	HT(\t)	064	52	0x34	4	0137	95	0x5F	_
012	10	0x0A	LF(\n)	065	53	0x35	5	0140	96	0x60	‘
013	11	0x0B	VT	066	54	0x36	6	0141	97	0x61	a
014	12	0x0C	FF(\f)	067	55	0x37	7	0142	98	0x62	b
015	13	0x0D	CR(\r)	070	56	0x38	8	0143	99	0x63	c
016	14	0x0E	SO	071	57	0x39	9	0144	100	0x64	d
017	15	0x0F	SI	072	58	0x3A	:	0145	101	0x65	e

Unicode-Zeichensatz

- Der ASCII-Code wie auch die Zeichensätze der ISO-8859-Familie basieren auf einer 8-Bit-Darstellung (256 Zeichen)

- Der Unicode verwendet 16 Bit (65536 Zeichen!)

- Welcher Code wird welchem Zeichen (mathematische Symbole wie Integralzeichen, griechische Buchstaben, japanisches Schriftzeichen, ...) zugewiesen?
  $\hookrightarrow$ **www.unicode.org**

- Beispiel für den Buchstaben **a** (Ordnungsnummer 97):

Zeichenformat	**Bitnotation**
8-Bit, z.B. iso-8859-1	0110001
16-Bit Unicode	000000000110001

- Konsequenz: Die meisten herkömmlichen Textdateien (mit Zeichen mit kleinen Ordnungsnummern) wären doppelt so groß!

- $\hookrightarrow$ UTF-8 Codierung

  - UTF-8 codiert Zeichen nur bei Bedarf mit mehr als 8 Bit.

  - Byte-Darstellung mit einer führenden 0, z.B. `01000001`, definiert das Byte als Zeichen aus dem Bereich der ASCII-Ordnungsnummern von 0 bis 127

  - Byte-Darstellung mit einer führenden 1, z.B. `10110010`, sagt, dass das Byte nur als Teil eines Zeichens ist, welches nur mit anderen Bytes zusammengelesen werden kann

  - Die führende 1 „maskiert" also etwas.

- Der Unicodebereich von 0 bis 127 wird durch ein Byte, der von 128 bis 2047 wird durch 2 Byte und der Bereich von 2048 bis 65535 durch 3 Byte dargestellt.

- Maskierungssequenzen:
    1. "10" bedeutet: dieses Zeichen beinhaltet Füllbits und ist nur im Zusammenhang mit vorangegangen Zeichen lesbar.
    2. "110" dieses Zeichen leitet ein Unicodezeichen bestehend aus 2 Byte ein.
    3. "1110" dieses Zeichen leitet ein Unicodezeichen bestehend aus 3 Byte ein.

- in der folgenden Tabelle ist $x \in \{0,1\}$

Zeichenbereich	**Bitnotation**
$0 \ldots 127$	$0xxx.xxx$
$128 \ldots 2047$	$110x.xxxx10xx.xxxx$
$2048 \ldots 65535$	$1110.xxxx10xx.xxxx10xx.xxxx$

- UTF-8 wandelt also den Uni-Code in einen Bytestrom um

- reiner ASCII- oder Latin-1-Text stellt bereits einen gültigen UTF-8-Bytestrom dar

3.2 Rechenmaschinen

- Monotone Rechenaufgaben haben schon seit vielen Jahrhunderten Tüftler bewegt, Maschinen zu konstruieren, die diese Aufgabe abnehmen können.

- Im 16. Jahrhundert entwarf Leonardo da Vinci eine Rechenmaschine. Entsprechende Notizen wurden 1967 entdeckt. Es gibt jedoch keine Hinweise, daß diese zu Zeiten von Leonardo da Vinci gebaut worden wäre.

- 1623 entwickelte und baute der Astronom und Mathematiker Wilhelm Schickard die erste bekannte Rechenmaschine, die alle vier Grundrechenarten beherrschte.

- 1645 wurde von Blaise Pascal die zweite bekannte Rechenmaschine gebaut.

- 1668 wurde die erste nicht-dezimale Addiermaschine für die englische Währung von Sir Samuel Morland gefertigt.

- 1673 baute Gottfried Leibniz eine Rechenmaschine, die neben den Grundrechenarten auch Wurzeln ziehen konnte.

- 1786 entstand die "Differenzmaschine" von J. M. Mueller entwickelt, die es erlaubte, Polynome zu tabulieren.

- 1801 entstand die automatisierte Web-Maschine von Joseph-Maire Jacuard, die mit Lochkarten programmiert wurde.

- 1820 entstand der erste in Massen produzierte Rechner von Charles Xavier Thomas de Colmar, der über 90 Jahre lang verkauft wurde.

- Da nicht jeder eine Rechenmaschine zur Verfügung hatte, gewannen viele Rechenhilfe große Popularität, zu denen insbesondere Tafeln für Logarithmen und trigonometrische Funktionen gehörten.

- Eine der ersten Logarithmentafeln entstand 1620 von Henry Briggs.

- Mit zunehmenden Ansprüchen aus Astronomie, Vermessungswesen und dem Ingenieurswesen wurden immer umfangreichere Tabellenwerke benötigt, die in mühseliger Handarbeit von Scharen an Mitarbeitern erstellt wurden. Das war zeitaufwendig, teuer und auch sehr fehleranfällig.

- Charles Babbage (1791-1871) versuchte, den ersten generell programmierbaren Rechner zu bauen einschließlich Lochkarten, frei nutzbarem Speicher und einem Drucker. Seine "Analytic Engine" wurde jedoch zu seinen Lebzeiten nie funktionsfähig. Seine hinterlassenen Teile und Aufzeichnungen ermöglichten später einen Nachbau durch das London Science Museum, der die Funktionsfähigkeit seines Entwurfs belegte.

- Ada Lovelace unterstützte Babbage und entwarf auf Papier Programme für die "Analytic Engine". Sie ist damit die erste Programmiererin in der Geschichte.

- 1938 baute Konrad Zuse einen mechanischen Rechner, das Binärsystem verwendete (Z1).

- 1939 entwickelten John V. Atanasoff und Clifford Berry eine 16-Bit-Addiermaschine auf Basis von Röhren.

- 1939 konstruierte Konrad Zuse eine Rechenmaschine auf Basis von elektrischen Relays (Z2).

- 1940 wurde bei den Bell Laboratories der "Complex Number Calculator" entwickelt, für den Bauteile von Telefon-Verbindungsanlagen verwendet wurden und der über Fernschreiber bedient werden konnte.

- 1941 entstand der erste programmierbare Rechner, der jedoch noch keine bedingten Sprünge unterstützte von Konrad Zuse (Z3).

- 1943 entstand "Harvard Mark I" als programmierbarer Rechner von Howard H. Aiken und seinem Team.

- 1943: "I think there is a world market for maybe five computers.", Thomas Watson, Vorstandsvorsitzender von IBM.

- 1943 wurde ein spezialisierter Rechner zum Knacken von Chiffren entwickelt von Max Newman, Wynn-Williams und deren Team, zu dem auch Alan Turing gehörte. Später wurde "Collossus" gebaut, der es erlaubte eine programmierbare logische Funktion auf ein Eingabe-Band mit einer Geschwindigkeit von 5000 Zeichen pro Sekunde auszuführen.

- 1946 entstand der erste vollständige elektronische Computer (ENIAC = Electronic Numerical Integrator and Computer) von John W. Mauchly und J. Presper Eckert am Ballistic Research Laboratory.

- 1948 entstand der erste Rechner, der sowohl das Programm als auch die Daten im Speicher verwaltete (SSEM = Small Scale Experimental Machine) am Manchester University.

- 1951 wurde der erste universell programmierbare Rechner (UNIVAC-1) von J. Presper Eckert and John Mauchly entwickelt und gebaut.

Modelle für Rechenmaschinen

Programmierbare Rechenmaschinen benötigen ein Funktionsmodell, das beschreibt, wie die Programm-Instruktionen ausgeführt werden. Es gibt zahlreiche verschiedene Funktionsmodelle:

Theoretische Modelle:

- Lambda-Kalkül, wurde in den 30er Jahren von Alonzo Church und Stephen Kleene entwickelt zur theoretischen Untersuchung der Berechenbarkeit.
- Turing-Maschine, wurde von Alan Turing entwickelt, ebenfalls für theoretische Untersuchungen.
- Zelluläre Automaten, die insbesondere durch die Arbeiten von John Horton Conway bekannt wurden.

Praktische Rechner-Architekturen:

- Die John-von-Neumann-Maschine die ist zugrundeliegende Architektur fast aller Rechner von heute. Wesentliches Merkmale sind die gemeinsame Nutzung des Speichers für Daten und Programme und die Trennung zwischen Recheneinheit und Speicher.
- Im Vergleich dazu sieht die Harvard-Architektur die Trennung zwischen Daten und Programmen vor. Namensgeber war die "Harvard Mark I".

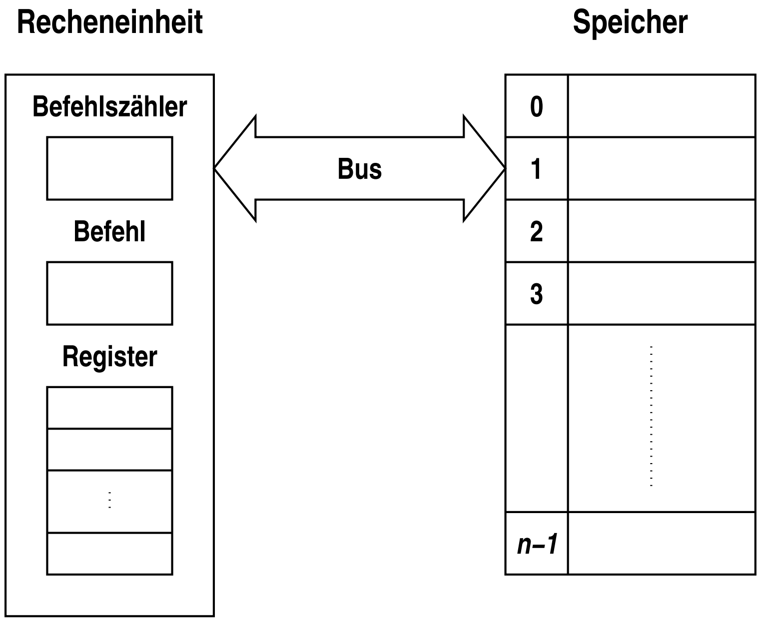


Abbildung 3.1: John-von-Neumann-Maschine

- Die John-von-Neumann-Maschine (siehe Abb. 3.1, S. 58) wurde von John von Neumann, John William Mauchly und J. Presper Eckert beim ENIAC-Projekt entwickelt.

- Der Speicher besteht aus n Speicherzellen, die individuell addressiert werden können.

- Über das Bus-System ist es möglich, einen Lesebefehl und eine Adresse aus dem Bereich $[0..n-1]$ anzugeben. Kurz darauf kopiert das Speichersystem den Inhalt der entsprechenden Zelle in den Bus. Umgekehrt kann auch über den Bus ein Schreibbefehl zusammen mit einer Adresse und dem Wert einer Speicherzelle gegeben werden und der Speicher kopiert dann den Wert in die entsprechende Zelle.

Ausführungszyklus

- Die Recheneinheit hat weitere Speicherzellen, die ihr direkt zugänglich sind. Diese werden Register genannt. Dazu zählen insbesondere der Befehlszähler, ein Register für den aktuellen Befehl und weitere allgemein verwendbare Register.

- Zu Beginn eines Rechenschritts wird die vom Befehlszähler adressierte Speicherzelle in das Befehls-Register geladen.

- Danach wird der Befehlszähler um 1 erhöht.

- Der Befehl wird dann dekodiert und ausgeführt.

- Im Rahmen eines Befehls ist es u.a. möglich,
  - Daten aus dem Speicher in eines der Register zu laden,
  - den Inhalt eines Registers in eine Speicherzelle zu schreiben,
  - arithmetische Operationen auf den Registern durchzuführen,
  - Werte in den Registern miteinander zu vergleichen und
  - den Befehlszähler zu verändern, ggf. in Abhängigkeit eines vorher stattgefundenen Vergleichs.

- Wenn der Befehl fertig ausgeführt ist, wiederholt sich diese Abfolge.

Rechner-Architekturen

Bei Rechner-Architekturen gibt es zwei Sichten:

- Sicht des Programmierers: Hier ist das Modell (abstrakte Architektur) relevant einschließlich der zur Verfügung stehenden Register, des Adreßraumes, des Speichers und insbesondere die zur Verfügung stehenden Instruktionen.

- Sicht des Hardware-Ingenieurs: Hier geht es darum, wie ein Modell konkret durch Hardware implementiert werden kann. Viele Einzelkomponenten sind notwendig, um ein funktionierendes System zu bauen: Prozessor, Hauptplatine, Speicher-Chips, Ein- und Ausgabegeräte.

Zu einer abstrakten Architektur gibt es häufig zahlreiche Implementierungen. So läuft beispielsweise ein Programm, das für den Intel 80386-Prozessor in den 80er Jahren geschrieben worden ist, auch heute noch auf einem System mit einem Pentium-IV-Prozessor.

Es gibt zahlreiche verschiedene abstrakte Architekturen, die nicht miteinander kompatibel sind. Unsere Maschinen von Sun verwenden beispielsweise die SPARC-Architektur. Programme für die SPARC-Architektur laufen nicht auf den Intel-Prozessoren und umgekehrt.

Redcode-Architektur

Redcode ist ein kleines übersichtliches Beispiel für eine abstrakte Rechner-Architektur:

- Redcode wurde zum ersten Mal vorgestellt in der Kolumne "Computer Recreations" der Zeitschrift Scientific American im Mai 1984 von A. K. Dewdney.

- Das klassische Redcode-Modell kommt mit 8000 Speicherzellen und 8 verschiedenen Instruktionen aus.

- Auf allgemeine Register wurde verzichtet. Stattdessen werden arithmetische Operationen direkt auf den Speicherzellen ausgeführt. Dies wäre weniger praktisch für eine Architektur, die in Hardware gegossen wird, aber Redcode wird nur emuliert, d.h.&,durch Programme auf normalen Maschinen simuliert.

Redcode-Befehle:

Befehle haben in Redcode ein oder zwei Operanden, die jeweils entweder einen Wert direkt angeben oder eine Speicherzelle direkt oder indirekt adressieren:

MOV A, B	Kopiere den Wert von A nach B
ADD A, B	Ersetze den Wert von B durch die Summe von A und B.
SUB A, B	Ersetze den Wert von B durch die Differenz von A und B.
JMP A	Springe nach A.
JMZ A, B	Springe nach A, falls B den Wert 0 hat.
JMG A, B	Springe nach A, falls B einen Wert ungleich 0 hat.
DJN A, B	Dekrementiere B um 1 und springe nach A, falls B noch nicht 0 wurde.
CMP A, B	Vergleiche A und B und überspringe die nächste Instruktion, falls die beiden Werte nicht übereinstimmen.

Hinweis: Bei Sprüngen wird ein neuer Wert in den Befehlszähler geschrieben.

Adressierungs-Modi:

Jeder der beiden Operanden eines Befehls kann auf verschiedene Weise adressiert werden:

- Konstante: Es wird keine Speicherzelle adressiert, sondern der gewünschte Wert ist direkt in dem Befehl enthalten.

- Direkt: In dem Befehl steht eine Adresse, die *relativ* zum (alten, noch nicht inkrementierten) Wert des Befehlszählers interpretiert wird. Das heißt, daß implizit der Befehlszähler zur Adresse addiert wird. Ist das Resultat größer oder gleich 8000, wird durch 8000 geteilt und der Rest genommen (aus 8000 wird so 0, aus 8001 entsprechend 1 usw.).

- Indirekt: Zunächst wird wie bei der direkten Adressierung der Inhalt einer Speicherzelle geladen. Dann wird diese ebenfalls als Adresse interpretiert, die *relativ zur vorher adressierten Speicherzelle* interpretiert wird.

Solche Adressierungsmodi sind auch typisch für real existierende Prozessoren. Allerdings wird typischerweise mit Hilfe von Registern indirekt adressiert.

Kapitel 4

Formale Sprachen

4.1 Grammatiken

4.1.1 Formale Sprache

- Es sei ein Vokabular V_T gegeben das aus einer endlichen Menge von lexikalischen Symbolen besteht. (Lexikalische Symbole werden auch Terminal-Symbole genannt, daher die Bezeichnung V_T).

- Die Menge V_T^* ist dann die Menge aller endlichen Folgen aus Symbolen aus V_T, einschließlich der leeren Folge.

- Eine Sprache L auf Basis des Vokabulars V_T ist dann eine Teilmenge von V_T^*.

- Problem: Wie kann die Menge L mit endlichem Aufwand definiert werden, wenn L unendlich viele Folgen enthält?

- Lösung: Es werden Grammatiken verwendet, die aus endlich vielen Bauvorschriften (genannt Produktionen) bestehen, die rekursiv jede Folge aus L konstruieren können.

Beispiel: Darstellung von ganzen Zahlen mit "Tausender-Punkt"

Vokabular: $V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$, die Menge der (arabischen) Ziffern, ergänzt um den "Punkt".

daraus Zeichenfolgen: 123.456 oder 12.1 **(?)**

L intuitiv definiert:

> keine führende 0, am Anfang eine oder zwei oder drei Ziffern, danach können kommen ein Punkt gefolgt von drei beliebigen Ziffern

4.1.2 Produktionen

- Produktionen sind Ersetzungsregeln in der Form

$$\langle \mathrm{A} \rangle \quad \longrightarrow \quad \alpha$$

- Auf der linken Seite steht genau ein sogenanntes Nicht-Terminal-Symbol (hier $\langle \mathrm{A} \rangle \in V_N$). Es gilt: $V_T \cap V_N = \emptyset$. Weiter sei $V := V_T \cup V_N$.

- Die Nicht-Terminal-Symbole dienen als Bezeichnung komplexerer "Bauteile", die aus kleineren Teilen zusammengesetzt sind. Ihnen werden normalerweise Namen gegeben, die das "Bauteil" beschreiben.

- Die rechte Seite besteht aus einer Sequenz von Terminal oder Non-Terminal-Symbolen: $\alpha \in V^*$. Diese Sequenz kann auch die Länge 0 haben.

- Griechische Buchstaben stehen für Sequenzen von Symbolen aus V^*.

- ϵ steht für eine Folge der Länge 0.

4.1.3 Grammatik

Eine Grammatik G ist ein 4-Tupel (V_T, V_N, P, S):

- V_T ist die endliche Menge der lexikalischen Symbole (Terminal-Symbole).

- V_N ist die endliche Menge der Nicht-Terminal-Symbole.

- $V_T \cap V_N = \emptyset$.

- P repräsentiert die Produktionen und ist eine endliche Teilmenge aus $V_N \times V^*$.

- Statt der Tupel-Notation $(\langle \mathrm{A} \rangle, \alpha) \in P$ wird die Ersetzungsregelform $\langle \mathrm{A} \rangle \longrightarrow \alpha$ verwendet.

- $\forall \langle \mathrm{A} \rangle \in V_N : \exists\, \alpha : (\langle \mathrm{A} \rangle, \alpha) \in P$.
(Für jedes Nicht-Terminal-Symbol gibt es mindestens eine Produktionsregel, bei der das Symbol auf der linken Seite verwendet wird).

- $S \in V_N$ ist das Startsymbol.

4.1.4 Sätze und Sprachen

Gegeben sei eine Grammatik $G = (V_T, V_N, P, S)$:

- α **produziert** β direkt: $\boxed{\alpha \;\rightarrow\; \beta}$
  falls sich α und β folgendermaßen zusammensetzen

$$\begin{aligned} \alpha &= \gamma\,\langle\mathrm{A}\rangle\,\delta \quad \alpha,\beta,\gamma,\delta \in V^*,\ \langle\mathrm{A}\rangle \in V_N \\ \beta &= \gamma\,\omega\,\delta \quad\ \ \omega \in V^* \end{aligned}$$

  und wenn es eine Produktion $\langle\mathrm{A}\rangle \longrightarrow \omega$ gibt.

- α **produziert** β: $\boxed{\alpha \;\rightarrow^+\; \beta}$
  falls es $\omega_1, \omega_2, \ldots, \omega_n \in V^*$ gibt, so daß

$$\alpha \rightarrow \omega_1 \rightarrow \omega_2 \rightarrow \ldots \rightarrow \omega_n \rightarrow \beta$$

  Man schreibt $\boxed{\alpha \;\rightarrow^*\; \beta}$, wenn $\alpha = \beta$ zulässig ist.

- **Satzform** für eine Grammatik:
  Jede Folge α mit $S \rightarrow^* \alpha$ für $\alpha \in V^*$, S Startsymbol.

- **Satz** für eine Grammatik:
  Jede Satzform $\alpha \in V_T^*$.

- **Sprache**, erzeugt von einer Grammatik:
  Die Menge aller möglichen Sätze.

Anmerkungen:

- Durch die Ableitung ergibt sich eine hierarchische Struktur, die graphisch dargestellt werden kann.
- Grammatiken dienen nicht nur zur Klärung, ob eine Folge aus V_T zu Sprache gehört, sondern liefern im Erfolgsfalle auch eine Struktur.
- Allerdings sind manche Grammatiken mehrdeutig, d.h. sie bieten mehrere Ableitungen für einen Satz an.

4.1.5 Beispiele

Tausenderzahlen:

- Menge der Terminal-Symbole: $V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$
- Menge der Nicht-Terminal-Symbole $V_N = \{S, A, B, N, Z\}$, S sei das Startsymbol
- Regeln:

Regel 1:	$S \longrightarrow A$	Regel 7:	$B \longrightarrow .ZZZ$	Regel 13:	$N \longrightarrow 4$
Regel 2:	$S \longrightarrow AB$	Regel 8:	$Z \longrightarrow 0$	Regel 14:	$N \longrightarrow 5$
Regel 3:	$A \longrightarrow N$	Regel 9:	$Z \longrightarrow N$	Regel 15:	$N \longrightarrow 6$
Regel 4:	$A \longrightarrow NZ$	Regel 10:	$N \longrightarrow 1$	Regel 16:	$N \longrightarrow 7$
Regel 5:	$A \longrightarrow NZZ$	Regel 11:	$N \longrightarrow 2$	Regel 17:	$N \longrightarrow 8$
Regel 6:	$B \longrightarrow BB$	Regel 12:	$N \longrightarrow 3$	Regel 18:	$N \longrightarrow 9$

- Beispiel für eine Ersetzungsfolge:

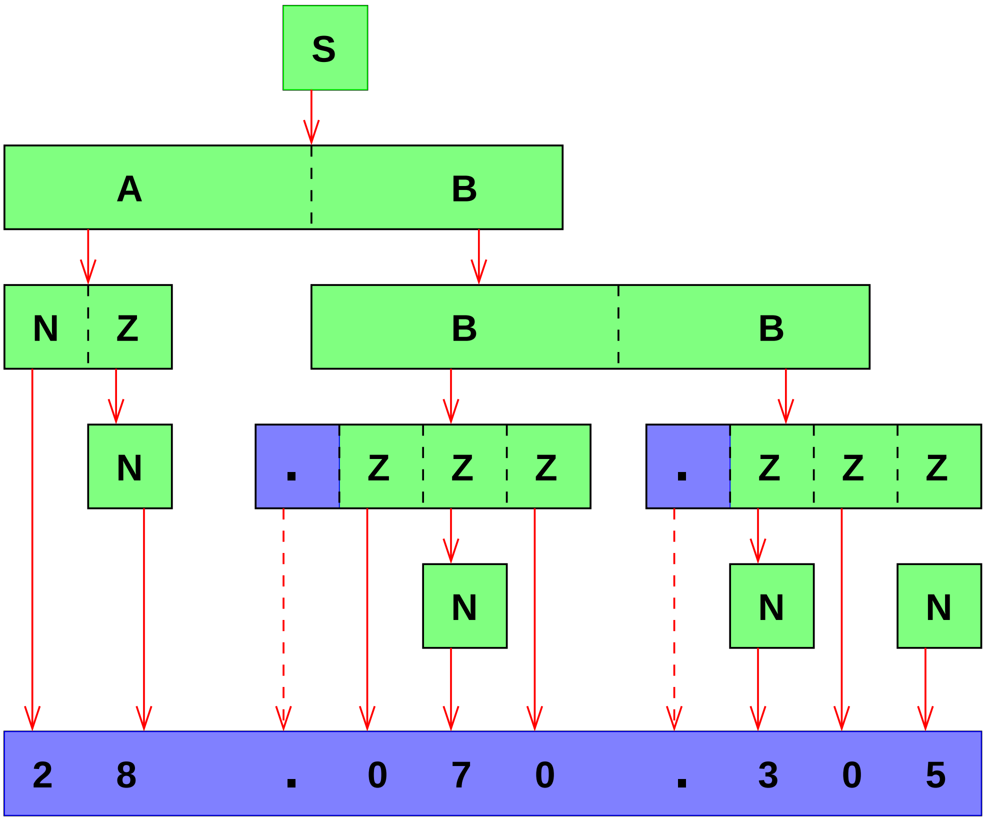


Abbildung 4.1: Ableitung “Tausenderzahlen”

Beispiel: Gleitkommazahlen

- Verbale Beschreibung: Eine Gleitkommazahl besteht aus einer nicht-leeren Folge von Ziffern, optional gefolgt von einem Komma und einer weiteren nicht-leeren Folge von Ziffern.
- $V_T = \{\text{“0”}, \text{“1”}, \text{“2”}, \text{“3”}, \text{“4”}, \text{“5”}, \text{“6”}, \text{“7”}, \text{“8”}, \text{“9”}, \text{“,”}\}$
- $V_N = \{\langle\text{Gleitkommazahl}\rangle,\ \langle\text{Vorkommastellen}\rangle,$ $\langle\text{Nachkommastellen}\rangle,\ \langle\text{Ziffernfolge}\rangle,\ \langle\text{Ziffer}\rangle\}$
- Produktionsregeln P:

$$
\begin{array}{rcl}
\langle\text{Gleitkommazahl}\rangle & \longrightarrow & \langle\text{Vorkommastellen}\rangle\ \langle\text{Nachkommastellen}\rangle \\
\langle\text{Vorkommastellen}\rangle & \longrightarrow & \langle\text{Ziffernfolge}\rangle \\
\langle\text{Nachkommastellen}\rangle & \longrightarrow & \epsilon \\
\langle\text{Nachkommastellen}\rangle & \longrightarrow & \text{“,”}\langle\text{Ziffernfolge}\rangle \\
\langle\text{Ziffernfolge}\rangle & \longrightarrow & \langle\text{Ziffer}\rangle \\
\langle\text{Ziffernfolge}\rangle & \longrightarrow & \langle\text{Ziffer}\rangle\ \langle\text{Ziffernfolge}\rangle \\
\langle\text{Ziffer}\rangle & \longrightarrow & \text{“0”} \\
 & \cdots & \\
\langle\text{Ziffer}\rangle & \longrightarrow & \text{“9”}
\end{array}
$$

- Startsymbol $S = \langle\text{Gleitkommazahl}\rangle$

- Behauptung: “12,3” ist ein Satz.
- Beweis durch die Angabe der entsprechenden Ableitung:

$$
\begin{array}{rcl}
\langle\text{Gleitkommazahl}\rangle & \longrightarrow & \langle\text{Vorkommastellen}\rangle\ \langle\text{Nachkommastellen}\rangle \\
 & \longrightarrow & \langle\text{Ziffernfolge}\rangle\ \langle\text{Nachkommastellen}\rangle \\
 & \longrightarrow & \langle\text{Ziffer}\rangle\ \langle\text{Ziffernfolge}\rangle\ \langle\text{Nachkommastellen}\rangle \\
 & \longrightarrow & \text{“1”}\ \langle\text{Ziffernfolge}\rangle\ \langle\text{Nachkommastellen}\rangle \\
 & \longrightarrow & \text{“1”}\ \langle\text{Ziffer}\rangle\ \langle\text{Nachkommastellen}\rangle \\
 & \longrightarrow & \text{“1”}\ \text{“2”}\ \langle\text{Nachkommastellen}\rangle \\
 & \longrightarrow & \text{“1”}\ \text{“2”}\ \text{“,”}\ \langle\text{Ziffernfolge}\rangle \\
 & \longrightarrow & \text{“1”}\ \text{“2”}\ \text{“,”}\ \langle\text{Ziffer}\rangle \\
 & \longrightarrow & \text{“1”}\ \text{“2”}\ \text{“,”}\ \text{“3”}
\end{array}
$$

Beispiel: Einfache Ausdrücke

- $V_T = \{\langle\text{Zahl}\rangle, \text{“}+\text{”}, \text{“}*\text{”}\}$
- $V_N = \{\langle\text{Summe}\rangle, \langle\text{Produkt}\rangle\}$
- Produktionsregeln P:

$$
\begin{array}{rcl}
\langle\text{Summe}\rangle & \longrightarrow & \langle\text{Produkt}\rangle \\
\langle\text{Summe}\rangle & \longrightarrow & \langle\text{Produkt}\rangle \text{ “}+\text{” } \langle\text{Summe}\rangle \\
\langle\text{Produkt}\rangle & \longrightarrow & \langle\text{Zahl}\rangle \\
\langle\text{Produkt}\rangle & \longrightarrow & \langle\text{Zahl}\rangle \text{ “}*\text{” } \langle\text{Produkt}\rangle
\end{array}
$$

- Startsymbol $S = \langle\text{Summe}\rangle$

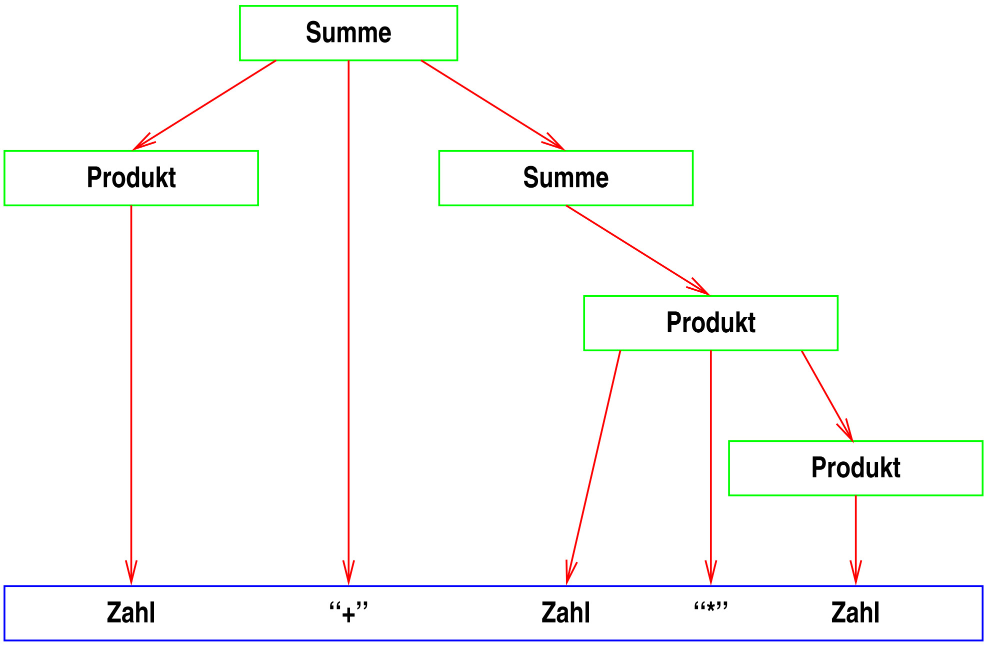


Abbildung 4.2: Ableitungsbaum für Einfache Ausdrücke

- Grammatiken können durch die Ableitungsstruktur festlegen, welche Operanden ein Operator erhält.
- In diesem konkreten Beispiel wird die Multiplikation zuerst durchgeführt, bevor es zur Addition kommt.

4.2 (Erweiterte) Backus-Naur-Form

- 1960 wurde von John Backus und Peter Naur bei der Beschreibung von Algol-60 zum ersten Male eine formale Grammatik für eine Programmiersprache spezifiziert.
- Bei der formalen Notation von Backus und Naur (kurz BNF = Backus-Naur-Form genannt) kommt zur Verringerung der Anzahl der Produktionsregeln noch "|" als Alternativsymbol hinzu.
- Die Kurzform $\langle \mathrm{A} \rangle \longrightarrow \alpha \mid \beta$ ist dabei äquivalent zu

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & \longrightarrow & \alpha \\ \langle \mathrm{A} \rangle & \longrightarrow & \beta \end{array}$$

Die erweiterte Backus-Naur-Form (**EBNF**) fügt weitere Notationen hinzu, um die Zahl der Produktionsregeln weiter reduzieren zu können:

- Optionalität: $\langle \mathrm{A} \rangle \longrightarrow \alpha\ [\ \beta\ ]\ \gamma$ entspricht

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & \longrightarrow & \alpha\ \langle \mathrm{A}' \rangle\ \gamma \\ \langle \mathrm{A}' \rangle & \longrightarrow & \beta \mid \epsilon \end{array}$$

- Wiederholung (0 bis beliebig oft): $\langle \mathrm{A} \rangle \longrightarrow \alpha\ \{\ \beta\ \}\ \gamma$ entspricht

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & \longrightarrow & \alpha\ \langle \mathrm{A}' \rangle \\ \langle \mathrm{A}' \rangle & \longrightarrow & \beta\ \langle \mathrm{A}' \rangle \mid \gamma \end{array}$$

- Klammerung: $\langle \mathrm{A} \rangle \longrightarrow \alpha\ (\ \beta \mid \gamma\ )\ \delta$ entspricht

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & \longrightarrow & \alpha\ \langle \mathrm{A}' \rangle\ \delta \\ \langle \mathrm{A}' \rangle & \longrightarrow & \beta \mid \gamma \end{array}$$

- Durch die Verwendung von EBNF geht ein Teil der Struktur verloren, die mit BNF noch zum Ausdruck kam.

Das "Tausenderpunktbeispiel":

Regeln 1, 2 und 6 werden zu: $S \longrightarrow A\{B\}$
Regeln 3, 4 und 5 werden zu: $A \longrightarrow N|NZ|NZZ$
Regel 7 bleibt
Regeln 8 und 9 werden zu: $Z \longrightarrow 0|N$
Regeln 10 bis 18 werden zu: $N \longrightarrow 1|2|3|4|5|6|7|8|9$

Die Grammatik von EBNF

- $V_T = \{\langle\text{Symbol}\rangle, \text{“} \longrightarrow \text{”}, \text{“} \mid \text{”}, \text{“[”}, \text{“]”}, \text{“\{”}, \text{“\}”}, \text{“(”}, \text{“)”}\}$
- $V_N = \{\langle\text{Produktionsregeln}\rangle,\ \langle\text{Produktionsregel}\rangle,$
  $\langle\text{Alternativen}\rangle,\ \langle\text{Sequenz}\rangle,\ \langle\text{Element}\rangle\}$
- Produktionsregeln P:

$$
\begin{array}{rcl}
\langle\text{Produktionsregeln}\rangle & \longrightarrow & \langle\text{Produktionsregel}\rangle \\
\langle\text{Produktionsregeln}\rangle & \longrightarrow & \langle\text{Produktionsregel}\rangle\ \langle\text{Produktionsregeln}\rangle \\
\langle\text{Produktionsregel}\rangle & \longrightarrow & \langle\text{Symbol}\rangle\text{“} \longrightarrow \text{”}\langle\text{Alternativen}\rangle \\
\langle\text{Alternativen}\rangle & \longrightarrow & \langle\text{Sequenz}\rangle \\
\langle\text{Alternativen}\rangle & \longrightarrow & \langle\text{Sequenz}\rangle\ \text{“} \mid \text{”}\ \langle\text{Alternativen}\rangle \\
\langle\text{Sequenz}\rangle & \longrightarrow & \langle\text{Element}\rangle \\
\langle\text{Sequenz}\rangle & \longrightarrow & \langle\text{Element}\rangle\ \langle\text{Sequenz}\rangle \\
\langle\text{Element}\rangle & \longrightarrow & \langle\text{Symbol}\rangle \\
\langle\text{Element}\rangle & \longrightarrow & \text{“(”}\langle\text{Alternativen}\rangle\text{“)”} \\
\langle\text{Element}\rangle & \longrightarrow & \text{“[”}\langle\text{Alternativen}\rangle\text{“]”} \\
\langle\text{Element}\rangle & \longrightarrow & \text{“\{”}\langle\text{Alternativen}\rangle\text{“\}”}
\end{array}
$$

- Startsymbol $S\ =\ \langle\text{Produktionsregeln}\rangle$

4.3 Endliche Automaten

Ein deterministischer endlicher Automat A ist ein 5-Tupel (Z, V, δ, s, E):

- Z ist eine endliche Menge von Zuständen.
- V ist eine endliche Menge von Symbolen.
- δ ist eine Übergangsfunktion $\delta : Z \times V \to Z$.
- $s \in Z$ ist der Startzustand.
- $E \subset Z$ ist die Menge der zulässigen Endzustände.

Wenn eine Zeichenfolge $\alpha = (\alpha_1, \cdots, \alpha_n) \in V^*(n \geq 0)$ gegeben ist, ergibt sich entsprechend eine Folge von Zuständen $\zeta_i \in Z$ mit

$$
\begin{array}{rcl}
\zeta_0 & = & s \\
\zeta_i & = & \delta(\zeta_{i-1}, \alpha_i) \qquad (i \geq 1)
\end{array}
$$

Ein endlicher Automat A erkennt die Zeichenfolge $\alpha \in V^*$, falls gilt: $\zeta_n \in E$.

Beispiel: Gleitkommazahlen

- $Z = \{z_1, \cdots, z_5\}$
- $V = \{“0”, \cdots, “9”, “,”\}$
-

δ	z_1	z_2	z_3	z_4	z_5
“0” ⋯ ”9”	z_2	z_2	z_4	z_4	z_5
“,”	z_5	z_3	z_5	z_5	z_5

- $s = z_1$
- $E = \{z_2, z_4\}$

Hinweis: z_5 dient als Fehlerzustand. Wenn an einer unpassenden Stelle ein Komma kommt (z.B. zu Beginn oder nachdem bereits ein Komma gesehen worden ist), dann bleibt der Automat bis zum Ende der Eingabefolge in diesem Fehlerzustand.

Graphische Darstellung endlicher Automaten

- Jeder Zustand aus Z wird durch einen Kreis repräsentiert.
- Sei $V_{i,j}$ die Menge aller $v \in V$ für die gilt: $\delta(z_i, v) = z_j$. Falls $V_{i,j} \neq \emptyset$, dann ist ein gerichteter Pfeil zu zeichnen von dem Kreis für z_i zu dem Kreis von z_j, der mit der Menge $V_{i,j}$ beschriftet wird.
- Der Anfangszustand z_1 wird durch ein “S” markiert.
- Zulässige Endzustände werden durch einen doppelten Kreis markiert.

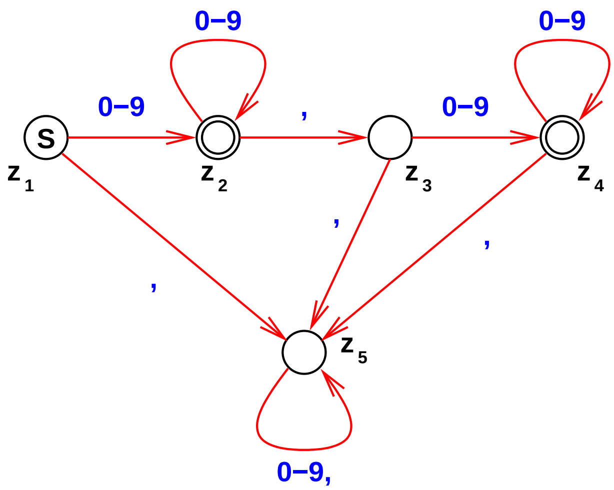


Abbildung 4.3: Automat für Gleitkommabeispiel

4.4 Endliche Automaten mit Textausgabe

Es gibt endliche Automaten, die Ausgabetext produzieren in Abhängigkeit

- von dem aktuellen Zustand: Moore-Maschine.
- von dem aktuellen Zustand und dem nächsten Eingabe-Symbol: Mealy-Maschine.

In Ergänzung oder an Stelle der Ausgabe sind natürlich auch beliebige andere Anweisungen denkbar, die einen Zustand außerhalb den des Automaten manipulieren. Endliche Automaten als Grundstruktur sind vielfach in Gebrauch in der Kommunikation mit anderen Programmen (z.B. über ein Netzwerk) oder mit Hardware.

Ein Beispiel für eine Moore-Maschine findet sich in Abb. 4.4, S. 72.

- Ein Tennis-Spiel strukturiert sich hierarchisch in ein Match, das aus mehreren Sätzen besteht, die wiederum aus Spielen zusammengesetzt sind, die aus einer Reihe von Punkten bestehen.
- Das Diagramm zeigt einen Moore-Automaten für ein einzelnes Tennis-Spiel.
- $V = \{\text{“A”}, \text{“B”}\}$
  (“A” heißt ein Punkt ging an Spieler A, “B” weist entsprechend einen gewonnen Punkt an Spieler B zu).

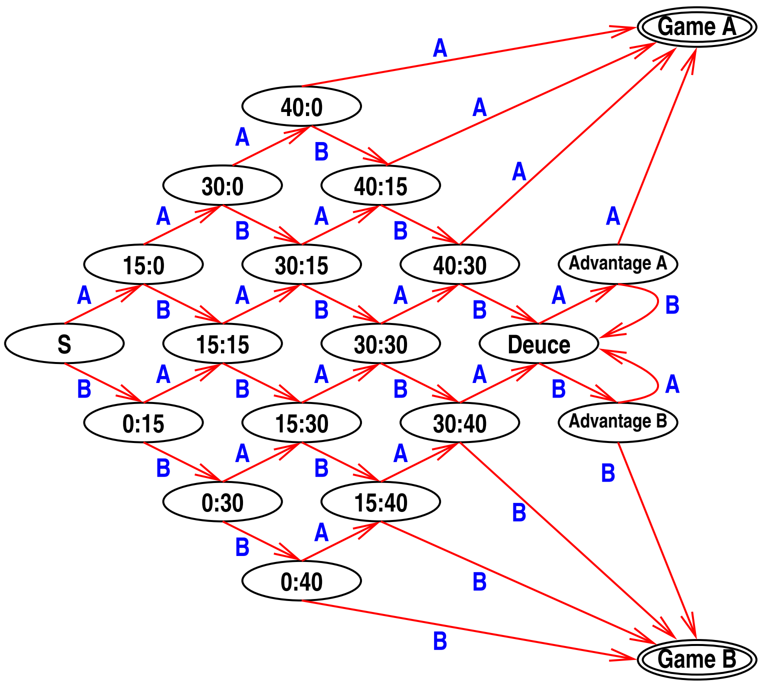


Abbildung 4.4: Moore-Maschine

4.5 Reguläre Sprachen

Sprachen, die mit einem endlichen Automaten definiert werden können, werden **reguläre** Sprachen genannt. Die entsprechende Grammatik läßt sich mit folgendem Verfahren ableiten:

- $V_T = V$ (die Menge der Terminal-Symbole der Grammatik entspricht genau der Menge der Symbole des Automaten)
- $V_N = Z$ (aus jedem Zustand des Automaten wird je ein Nicht-Terminal-Symbol der Grammatik)
- Für jeden Zustand $z \in E$ (Endzustand) gibt es eine Produktionsregel:
  $\langle z \rangle \longrightarrow \epsilon$
- Für alle Zustandskombinationen $(z_i, z_j) \in Z \times Z$, wird für jedes Symbol v aus $V_{i,j}$ eine Produktionsregel gebildet:
  $\langle z_i \rangle \longrightarrow v \langle z_j \rangle$
- $S = \langle s \rangle$ (das Startsymbol des Automaten wird zum Startsymbol der Grammatik)

Beispiel:

Gegeben sei der in Abb. 4.5 (S. 73) dargestellte Automat über dem Vokabular $V = \{0, 1\}$ mit der Zustandsmenge $Z = \{S, A, B, C, D\}$. Nicht dargestellte Übergänge sind Fehlerübergänge.

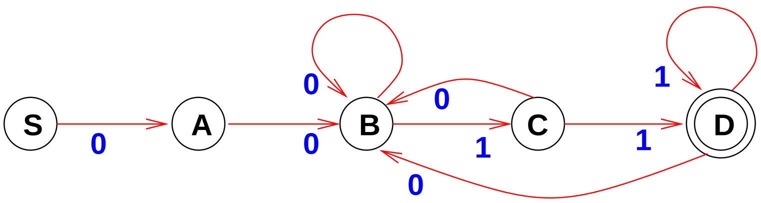


Abbildung 4.5: Ein 0/1-Automat

Grammatik:

$V_T = V$, $V_N = Z$, Startzymbol: S

Produktionsregeln:

$S \longrightarrow 0A$	$A \longrightarrow 0B$	$B \longrightarrow 0B$
$B \longrightarrow 1C$	$C \longrightarrow 0B$	$C \longrightarrow 1D$
$D \longrightarrow 0B$	$D \longrightarrow 1D$	$D \longrightarrow \epsilon$

Beispiel: Gleitkommazahlen

Der endliche Automat für Gleitkommazahlen (Abb. 4.3, S. 71) läßt sich in entsprechend der vorgestellten Transformationsregel in eine Grammatik überführen:

- $V_T = \{“0”, \cdots, “9”, “,”\}$
- $V_N = \{\langle z_1\rangle, \cdots, \langle z_5\rangle\}$
- Produktionsregeln P:

$$\begin{array}{lcl}
\langle z_2\rangle & \longrightarrow & \epsilon \\
\langle z_4\rangle & \longrightarrow & \epsilon \\
\langle z_1\rangle & \longrightarrow & “0”\langle z_2\rangle \\
 & \cdots & \\
\langle z_1\rangle & \longrightarrow & “9”\langle z_2\rangle \\
\langle z_1\rangle & \longrightarrow & “,”\langle z_5\rangle \\
\langle z_2\rangle & \longrightarrow & “0”\langle z_2\rangle \\
 & \cdots & \\
\langle z_2\rangle & \longrightarrow & “9”\langle z_2\rangle \\
\langle z_2\rangle & \longrightarrow & “,”\langle z_3\rangle
\end{array}
\qquad
\begin{array}{lcl}
\langle z_3\rangle & \longrightarrow & “0”\langle z_4\rangle \\
 & \cdots & \\
\langle z_3\rangle & \longrightarrow & “9”\langle z_4\rangle \\
\langle z_3\rangle & \longrightarrow & “,”\langle z_5\rangle \\
\langle z_4\rangle & \longrightarrow & “0”\langle z_4\rangle \\
 & \cdots & \\
\langle z_4\rangle & \longrightarrow & “9”\langle z_4\rangle \\
\langle z_4\rangle & \longrightarrow & “,”\langle z_5\rangle \\
\langle z_5\rangle & \longrightarrow & “0”\langle z_5\rangle \\
 & \cdots & \\
\langle z_5\rangle & \longrightarrow & “9”\langle z_5\rangle \\
\langle z_5\rangle & \longrightarrow & “,”\langle z_5\rangle
\end{array}$$

- $S = \langle z_1\rangle$

Reguläre Grammatik:

Bei einer regulären Grammatik sind alle Regeln von der Form

$$\boxed{X \longrightarrow bY, X \longrightarrow b}$$

oder von der Form

$$\boxed{X \longrightarrow Yb, X \longrightarrow b}$$

mit X und Y je **ein** Non-Terminalsymbol, b eine (auch leere) Folge von Terminalsymbolen

Eine Sprache ist eine **reguläre Sprache**, wenn es eine reguläre Grammatik gibt, die diese definiert.

Beispiel: Ganze Zahlen mit Tausender-Punkt

Menge der Terminalsymbole wird ergänzt um das "leere" Symbol (ϵ)

- $V = T = \{\epsilon, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$
- $NT = \{S, Z, H, T, U, V, W\}$
- Produktionsregeln (EBNF):

$S \longrightarrow 1Z\|2Z\|3Z\|4Z\|5Z\|6Z\|7Z\|8Z\|9Z$ $Z \longrightarrow \epsilon\|0H\|1H\|2H\|3H\|4H\|5H\|6H\|7H\|8H\|9H$ $H \longrightarrow \epsilon\|0T\|1T\|2T\|3T\|4T\|5T\|6T\|7T\|8T\|9T$
$T \longrightarrow \epsilon\|.U$
$U \longrightarrow 0V\|1V\|2V\|3V\|4V\|5V\|6V\|7V\|8V\|9V$ $V \longrightarrow 0W\|1W\|2W\|3W\|4W\|5W\|6W\|7W\|8W\|9W$ $W \longrightarrow 0T\|1T\|2T\|3T\|4T\|5T\|6T\|7T\|8T\|9T$

- Startsymbol: S

4.6 Nicht-reguläre Sprachen

Können alle durch Grammatiken beschreibbare Sprachen auch durch endliche Automaten beschrieben werden?

Hier ist eine Grammatik für eine Sprache, für die kein endlicher Automat gefunden werden kann:

- $V_T = \{``(", ``)"\}$
- $V_N = \{\langle A \rangle, \langle B \rangle\}$
- Produktionsregeln P:

$$\begin{array}{rcl} \langle A \rangle & \longrightarrow & \epsilon \\ \langle A \rangle & \longrightarrow & \langle B \rangle \; \langle A \rangle \\ \langle B \rangle & \longrightarrow & ``(" \langle A \rangle ``)" \end{array}$$

4.7 Reguläre Ausdrücke

Die Definition **regulärer Ausdrücke** erfolgt induktiv:

- Das leere Wort (ϵ) ist ein regulärer Ausdruck
- Jedes Terminalsymbol ist ein regulärer Ausdruck
- Sind α und β reguläre Ausdrücke, so auch $\alpha\beta$ (Konkatenation), $\alpha|\beta$ (Auswahl) sowie $\{\alpha\}$ (Folge)

  Im letzten Fall verwendet man die Schreibweise $(\alpha)*$ mit dem Meta-Symbol $*$ zusätzlich zu den Meta-Symbolen (); damit können die geschweiften Klammern fortan wieder für Mengen verwendet werden.

- Meta-Symbole sind hier also: `(` `)` `*` $|$

Zusammenhang zu **regulären Sprachen**:

Seien A, α, β reguläre Ausdrücke und bezeichne $L(A)$ die von A definierte Sprache (Menge von korrekten Symbolfolgen); dann gilt:

- Ist $A = \epsilon$ (leeres Wort), so ist $L(A) = \{\epsilon\}$.
- Ist $A = a$ (a Terminalsymbol), so ist $L(A) = \{a\}$.
- Ist $A = \alpha\beta$, so ist $L(A) = L(\alpha)L(\beta)$.
  (das kartesische Produkt mit dem Konkatenationsoperator von $L(\alpha)$ und $L(\beta)$)
- Ist $A = (\alpha|\beta)$, so ist $L(A) = L(\alpha) \cup L(\beta)$.
- Ist $A = (\alpha)*$, so ist $L(A) = L(\alpha)^*$.

Beispiel: Ganze Zahlen mit Tausender-Punkt
statt $0|1|2|3|4|5|6|7|8|9$ schreiben wir vereinfachend $[0-9]$ (Intervall)

$$\boxed{([1-9]|[1-9][0-9]|[1-9][0-9][0-9])(.[0-9][0-9][0-9])*}$$

Anmerkungen:

- Meta-Symbole: `( ) [ ] *` $|$

- die Zeichen zwischen den eckigen Klammern stellen ein Intervall (s.o.) dar, können aber auch eine Aufzählung bedeuten; $[axp]$ bedeutet also: eines der in der Aufzählung genannten Zeichen (a oder x oder p).

- Sollten diese Meta-Symbole als Symbole der zu definierenden Sprache vorkommen, so wird ihnen ein **backslash** (\) vorangestellt; damit wird der *backslash* selbst zu einem Meta-Symbol!

Weitere Meta-Symbole zur Wiederholung: $\{n, m\} \rightarrow$ das Vorangehende mindestens n-mal, höchstens m-mal wiederholen

$$\boxed{([1-9][0-9]\{0,2\})(.[0-9][0-9][0-9])*}$$

4.8 Reguläre Ausdrücke und UNIX-Tools (egrep)

Die folgende Tabelle gibt eine Übersicht über reguläre Ausdrücke, wie sie viele UNIX-Kommandos kennen.

`c`	Terminalzeichen `c`
`\c`	Escapesequenz, oder falls `c` Meta-Zeichen, `c` als Terminalsymbol
`^`	Anfang (Zeile, Wort) – Meta-Zeichen
`$`	Ende (Zeile, Wort) – Meta-Zeichen
`.`	beliebiges Einzel-Zeichen – Meta-Zeichen
`[abc...]`	ein Terminalzeichen aus der Aufzählung zwischen `[]`
`[^abc...]`	alle Zeichen als Terminalzeichen außer den Aufgezählten in `[]`
`[a-f]`	alle Zeichen zwischen `a` und `f` nach ASCII-Ordnung als Terminalzeichen
`[^a-f]`	alle Zeichen außer denen zwischen `a` und `f` als Terminalzeichen
`r1` | `r2`	`r1` oder `r2`
`(r1)(r2)`	`r1` gefolgt von `r2`
`(r)*`	beliebige viele `r`'s hintereinander (auch keins)
`(r)+`	beliebig viele, aber mindestens ein `r`
`(r)?`	ein oder kein `r`
`(r)`	Klammern dienen der Gruppierung – Meta-Zeichen

Metazeichen sind also: `( ) [ ] \ . ^ + * ? $`

Ein Fragezeichen als Terminalsymbol muss demnach entweder als `\?` oder als `[?]` (Aufzählung) definiert werden. Ein *backslash* (\) muss demnach als `\\` definiert werden. Die Zeichen `^` und `$` definieren abhängig von der Zielsetzung oder Eigenschaften des jeweiligen Tools Anfang / Ende einer Zeile oder eines Wortes oder …

egrep ist ein Kommando, das in einer Datenmenge (Standardeingabe, Datei) suchen kann:

Typischer Aufruf: `egrep options pattern {file}`

`pattern` ist dabei ein regulärer Ausdruck, der mit obigen Meta-Symbolen gebildet werden kann.

Achtung: Die Shell interpretiert die Kommandozeile, **bevor** *egrep* zur Ausführung gebracht wird!

Beispiele:
Kommando **wc** kann Zeichen, Worte oder Zeilen **zählen** (*word count*).

```
hypatia$ cat text
Dies ist ein Text fuer egrep:

123.500 3334

^$

\fBUNIX\fP ist ein Multi-Tasking-System - viele nehmen Linux
Linux ist eine gute Alternative zu WindowsXP
hypatia$ egrep '^$' text | wc -l  # Wieviele Leerzeilen?
      3
hypatia$ egrep '\^' text # pattern mit '' quotieren!
^$
hypatia$ egrep '\\fB' text
\fBUNIX\fP ist ein Multi-Tasking-System - viele nehmen Linux
hypatia$ egrep Linux text
\fBUNIX\fP ist ein Multi-Tasking-System - viele nehmen Linux
Linux ist eine gute Alternative zu WindowsXP
hypatia$ egrep 'Linux$' text
\fBUNIX\fP ist ein Multi-Tasking-System - viele nehmen Linux
hypatia$ egrep Linux$ text
\fBUNIX\fP ist ein Multi-Tasking-System - viele nehmen Linux
hypatia$ egrep '^[0-9][0-9][0-9]\.([0-9][0-9][0- 9])*'  text
123.500 3334
hypatia$
```

```
hypatia$ cat nochnText
12:Borchert:23572
Die folgende Zeile enthaelt ein paar blanks

:13:Schweiggert:23570
14:Richter:23571
15:Melzer:23574


hypatia$ egrep '^[ ]*$' nochnText | wc -l
      3
hypatia$ egrep '^$' nochnText | wc -l
      2
hypatia$ egrep -v '^$' nochnText | wc -l
      6
hypatia$ wc -l nochnText
      8 nochnText
hypatia$ egrep '^[^:].*:' nochnText
12:Borchert:23572
14:Richter:23571
15:Melzer:23574
hypatia$ egrep Zeile text nochnText
nochnText:Die folgende Zeile enthaelt ein paar blanks
hypatia$ egrep ein Text nochnText
egrep: Text: No such file or directory
nochnText:Die folgende Zeile enthaelt ein paar blanks
hypatia$ egrep ein text nochnText
text:Dies ist ein Text fuer egrep:
text:\fBUNIX\fP ist ein Multi-Tasking-System - viele nehmen Linux
text:Linux ist eine gute Alternative zu WindowsXP
nochnText:Die folgende Zeile enthaelt ein paar blanks
hypatia$
```

4.9 Algorithmen

4.9.1 Einführung

Gegeben ist ein "Problem":

- eine Menge von Zahlen soll sortiert werden,
- auf einem Schachbrett sollen 8 Damen so plaziert werden, dass sich je zwei nicht bedrohen,
- ein Handlungsreisender soll n Städte besuchen – finde "optimale" Route,
- ...

Aufgaben:

- Finde eine Repräsentation der Problemdaten in den von einer Programmiersprache zur Verfügung gestellten Sprachmitteln (wie soll ein Schachbrett dargestellt werden?)
- Finde einen Algorithmus und formuliere diesen in den von dieser Sprache zur Verfügung gestellten Sprachmitteln

Diese beiden Schritte hängen eng zusammen!

4.9.2 Was ist ein Algorithmus?

Informell und intuitiv:

- Eine Beschreibung zur Lösung eines Problems, die in Schritten ausführbar (operativ, effektiv) ist, die also einen **Prozess** (Vorgang) definiert, und zwar so, dass "jemand" – der **Prozessor** – diesen ausführen kann
- Diese Beschreibung ist genau: alle Schritte und die Reihenfolge sind unmissverständlich (für den Prozessor)
- Die Formulierung muss auf einer wohlgewählten Ebene der Details – **Abstraktionsebene** – aufhören!
- Die Beschreibung muss "endlich" sein (also kein "...")!
- Die Termination muss gegeben sein: der beschriebene Prozess muss nach endlich vielen Schritten zum Ende kommen!
- Der Algorithmus (Prozess) arbeitet auf benannten Objekten!

Beispiele für Algorithmen (aus dem täglichen Leben):

- Gebrauchsanweisungen (Prozessor ?)
- Kochrezepte
- Strickanleitungen

4.9.3 Beispiel: Berechnung der Potenz

Zu berechnen sei die Potenz einer reellen Zahl – der zur Verfügung stehende Prozessor "verstehe" nur die Grundrechenarten $+, -, *, /$!

Gegeben seien also die benannten Objekte b (Basis, reelle Zahl) und n (Exponent, positive ganze Zahl)! Zu berechnen ist b^n!

Naheliegender Algorithmus:

Multipliziere b n-mal mit sich selbst!

Ein Programm dazu:

```
(* Initialisierung *)
Read(b,n);
(* der eigentliche Algorithmus: *)
p := 1.0;
WHILE n >= 1 DO
   p := p * b; n := n - 1;
END;
(* Ausgabe: *)
Write(p);
```

Dieser Algorithmus führt also n Multiplikationen aus, die zwar sehr schnell gehen, aber dennoch im Vergleich zu Additionen sehr aufwendig sind!

4.9.4 Komplexität (time)

Alternativer Algorithmus:

Feststellung:

- Ist der Exponent eine gerade Zahl, so gilt:

  $b^n = (b^2)^{\frac{n}{2}}$

- Es sind also $\frac{n}{2} + 1$ Multiplikation durchzuführen!

Ein Programm hierzu:

```
Read(b,n);
p := 1.0;
WHILE n >= 1 DO
  IF ODD(n) (* n ungerade? *) THEN
     p := p * b; n := n - 1;
  ELSE
     n := n DIV 2 ; (* ganzzahlige Division *)
     b := b * b;
  END;
END;
Write(p);
```

Zahl der Multiplikationen?

- Im besten Fall ist n immer geradzahlig, bis irgendwann $n = 1$ wird – dann sind es $log_2(n) + 1$ Multiplikationen!
- Im schlechtesten Fall ist n immer alternierend gerad- / ungeradzahlig – dann sind es etwa $2 * log_2(n)$ Multiplikationen!
- Interessant ist die Aussage nur in Abhängigkeit von n – wie steigt die Anzahl der Multiplikationen in Abhängigkeit von n? Konstante Summanden (oben $+1$) sind bei großen n uninteressant!
- Also wächst bei diesem Algorithmus die Zahl der Multiplikationen zur Berechnung von b^n mit $log_2(n)$
- Beim vorherigen Algorithmus wächst die Zahl der Multiplikationen zur Berechnung von b^n (linear) mit n

Man sagt: Die (Laufzeit-) **Komplexität** des ersten Algorithmus ist höher als die vom zweiten – oder die Komplexität des ersten ist $O(n)$, die des zweiten ist $O(log_2(n))$, wobei n die "Problemgröße" beschreibt (die hier durch den Exponenten bestimmt wird).

Wenn also die exakte Zahl der Multiplikationen durch die Funktion $f(n)$ beschrieben ist, so bedeutet die Komplexitätsangabe $O(log_2(n))$:

$$\lim_{n\to\infty} \frac{f(n)}{log_2(n)} \leq M$$

mit einer Konstanten M!

Das $O()$ ist das sog. Landau-Symbol!

Am Rechner:

Die obigen beiden Algorithmen sind jetzt jeweils in einem Programm in eine Schleife eingebettet, in der sie für $n = 10000, (1), 100000$ berechnet werden. Die Zeitmessung mit dem UNIX-Kommando **time** liefert:

1. Algorithmus	2. Algorithmus
6 min 20.8 sec	0 min 0.23 sec

Die absoluten Zahlen sind hier unbedeutend (hängen von vielen Faktoren ab). Beide Messungen wurden unter i.w. gleichen Bedingungen durchgeführt.

Kapitel 5

Ein Editor: vi

5.1 Etwas vorab

In der Welt von UNIX bzw. GNU/Linux gibt es zwei Familien von mächtigen Editoren, die große Popularität erlangten: Emacs und vi.

Literatur: Christ, J.: **TerminalBuch vi - Effizient editieren unter UNIX.** Oldenbourg Verlag München Wien 1989

Ein **Editor** (lat. *edire*) ist ein Programm zur Texterstellung und Textbearbeitung – dazu zählt auch die Navigation im Text:

- Programmtexte
- "Normale" Texte, emails
- Kommandozeilen
- u.a.m.

(Meine subjektiven) Anforderungen:

- Ein Editor für alle Zwecke ⟶ integrierbar in Mailtool und Shell!
- Auf (fast) allen Betriebssystemen verfügbar!
- Ergonomisch (wenig / kein Wechsel zwischen Tastatur und Maus)!
- Schnell und sicher!
- Hohe Funktionalität! Insbesondere: Reguläre Ausdrücke!!!
- Einfach / billig / umsonst zu haben!

5.2 Varianten des vi

- Der vi wurde Ende der 70er Jahre und Anfang der 80er Jahre kontinuierlich weiterentwickelt und gehört seitdem zu den Standardwerkzeugen unter UNIX.
- Leider war der Quelltext für den **vi** durch die Kooperation mit AT&T proprietär, so dass für freie Software-Systeme (wie GNU/Linux) der vi neu entwickelt werden musste. All diese Varianten bemühten sich (mehr oder weniger erfolgreich), die originalen Kommandos des vi zu implementieren und zusätzliche Funktionalitäten anzubieten.
- Besondere Popularität genießt hier heute der **vim** (*vi improved*). Diesen Editor gibt es für alle gängigen Plattformen unter `http://www.vim.org/`.
- Auf unseren Systemen steht sowohl das Original (**vi**) als auch der in der Funktionalität weit darüber hinausgehende **vim** zur Verfügung.
- Benutzern von Microsoft-Windows-Systemen sei es aber empfohlen, vim im Rahmen von Cygwin zu installieren.
  Siehe `http://www.cygwin.com/`
- Einen Überblick über alle bekannten Versionen des vi liefert die Seite von Sven Guckes unter
  `http://www.vi-editor.org/`

5.3 vi in Schritten

Der vi arbeitet grundsätzlich mit seiner eigenen Kopie des zu bearbeitenden Textes. Eine Sicherung der vi-eigenen Kopie in eine Datei erfolgt nur durch ausdrückliche Kommandos.

Aufruf und Ausstieg (Kommandomodus, s.u.):

Aufruf	Effekt
`vi`	Dateiname wird beim Ausstieg (Schreiben) angegeben
`vi name`	Aufruf der Datei `name` am Anfang
`vi +n name`	Aufruf der Datei `name` bei Zeile `n`
`vi + name`	Aufruf der Datei `name` am Ende
`vi name1,name2,...`	Aufruf der Datei `name1`, weiter mit `:n`
`vi -t tag`	Aufruf der Datei, die `tag` enthält
`vi +/muster name`	Aufruf der Datei `name` mit Positionierung auf erste Zeile die `muster` enthält
`view name`	nur Lesen der Datei `name`
Beenden	Effekt
`ZZ`	Schreiben und Beenden
`:w`	Schreiben
`:q`	Beenden (bedingt: falls keine Änderungen)
`:q!`	unbedingtes Beenden, kein Schreiben
`:wq`	Schreiben und Beenden

Aufbau des Bildschirms beim vi

- Die unterste Zeile ist für Statusangaben reserviert und die Eingabe längerer Kommandos.
- Alle anderen Zeilen dienen zur Anzeige des zu bearbeitenden Textes.
- Um leere Zeilen vom Textende unterscheiden zu können, werden nicht im Text vorhandene Zeilen mit einem führenden “ ” markiert.
- Zu Beginn wird oben der Anfang des eingelesenen Textes angezeigt und der Cursor steht auf dem ersten Zeichen der ersten Zeile.

Der Editor kennt zwei Modi:

- **Kommando-Modus**
- **Text- oder Schreibmodus**

Wechseln vom Kommando-Modus zum Schreibmodus durch Eingabe eines Kommandos zur Texteingabe:

Kommando	Wirkung
`a`	`append`: nach der Stelle, auf der der Cursor steht, werden die im Folgenden eingegebenen Zeichen eingefügt
`A`	nach dem Zeilenende einfügen (Zeile: s.u.) – wie `$a`
`i`	`insert`: vor der Stelle auf der der Cursor steht, werden die im Folgenden eingegebenen Zeichen eingefügt
`I`	Am Zeilenanfang einfügen - wie `^i`
`o`	neue Zeile **nach** aktueller Zeile und dort `a`
`O`	neue Zeile vor aktueller Zeile und dort `a`

- Wechseln vom Schreibmodus in den Kommando-Modus: Drücken der **Escape-Taste** (*Esc*).
- Drücken der Esc-Taste im Kommandomodus beläßt den Kommandomodus – also im Zweifelsfall die Esc-Taste mehrfach drücken!
- Unmittelbar nach dem Aufruf ist der *vi* im Kommando-Modus!

Begriffe:

- **Zeile:** durch *return* (*newline*) abgeschlossene Zeichenfolge
- **Wort:** Zeichenfolge ohne Leerraum (*blank, tab, newline*)

Zeilenummern sichtbar machen: :`set nu` (*numbering*)
Zeilennummerierung wieder ausschalten: :`set nonu`

Cursor-Bewegungen:

- nur im Kommando-Modus
- mit den Pfeiltasten (allerdings bei manchen Installationen nicht möglich)
- ansonsten: Kommandoorientiert

Kommando	Bedeutung
`h`	ein Zeichen nach links, max. bis Zeilenanfang
`j`	ein Zeichen nach unten (nächste Zeile)
`k`	eine Zeile nach oben
`l`	ein Zeichen nach rechts, max. bis Zeilenende
`4l`	4 Zeichen nach rechts, analog für oben, unten, links
`H`	zum Bildschirmanfang
`M`	zur Bildschirmmitte
`L`	zum Bildschirmende (unterste Zeile auf dem Bildschirm)
`+`	nächste Zeile
`-`	vorhergehende Zeile
`return`	wie `+`
`^`	auf erstes Nicht-Leerzeichen (i.a. Zeilenanfang)
`0`	Zeilenanfang
`$`	Zeilenende
`w`	zum Anfang des nächsten Wortes
`W`	wie `w`, der Punkt wird als Worttrenner ignoriert
`b`	zum Anfang des vorausgehenden Wortes (ohne Sonderzeichen)
`B`	wie `b`, analog zu `W`
`e`	zum nächsten Wortende
`E`	wie `e`, analog `W`
`)`	auf nächsten Satzanfang
`(`	auf vorigen Satzanfang
`f`**x**	zum nächsten **x**
`F`**x**	auf letztes **x**
`t`**x**	vor nächstes **x**
`T`**x**	vor letztes **x**
`;`	wiederholt letztes `f F t T`

Cursor-Bewegungen:

Kommando	Bedeutung
`ctrl-f`	um eine Bildschirmseite weiter (*forward scrolling*)
`ctrl-b`	um eine Bildschirmseite zurück (*backward scrolling*)
`ctrl-d`	um eine halbe Bildschirmseite weiter (*down*)
`ctrl-u`	um eine halbe bildschirmseite zurück (*up*)
`ctrl-e`	um eine Zeile vorwärts
`ctrl-y`	um eine Zeile rückwärts
`z+`	neue Seite mit der aktuellen Zeile oben
`z.`	neue Seite mit der aktuellen Zeile in der Mitte
`z-`	neue Seite mit der aktuellen Zeile unten
`:$`	zum Dateiende
`:1`	zum Dateianfang (erste Zeile)
`:5`	zur fünften Zeile
`:zahl`	zur `zahl`-ten Zeile
`:/string/`	zur nächsten Zeile, die `string` enthält

Text verändern:

Kommando	Bedeutung
`x`	Zeichen, auf dem der Cursor steht, löschen
`dw`	Wort löschen (ab der Cursor-Position)
`3dw`	die nächsten drei Worte löschen
`dd`	Zeile, in der der Cursor steht, löschen
`3dd`	die nächsten drei Zeilen löschen
`d$`	vom Cursor bis Zeilenende löschen
`D`	wie `d$`
`:2,7d`	Zeile 2 bis Zeile 7 löschen
`:.,$d`	von aktueller Zeile bis Ende alles löschen (wie dG)
`d)`	lösche nächsten Satz
`dG`	alles von aktueller Pos. bis Dateiende löschen
`u`	letztes Kommando rückgängig machen
`:r file`	Inhalt von `file` nach aktueller Zeile einfügen
`yy`	(*yank*) Zeile in einen Puffer schreiben (siehe `p`)
`3yy`	aktuelle und folgende 2 Zeilen (also 3) in den Puffer schreiben
`p`	(*put*) Puffer nach aktueller Zeile einfügen
`P`	wie `p`, nur **vor** der aktuellen Zeile
`J`	aktuelle Zeile durch folgende verlängern (zusammenfügen)
`cw`	*change word*: am Wortende erscheint ein `$`, Wort kann überschrieben werden
`c$`	am Zeilenende erscheint ein `$`, Zeichen bis zum Zeilenende werden überschrieben
`:s/str1/str2/`	erstes Vorkommen von `str1` in der aktuellen Zeile wird durch `str2` ersetzt (*substitute*)
`:s/str1/str2/g`	alle Vorkommen von `str1` in der aktuellen Zeile werden durch `str2` ersetzt (*substitute global*)
`:1,$s/str1/str2/`	im gesamten Text `1,$` wird in jeder Zeile das erste Vorkommen von `str1` durch `str2` ersetzt
`:1,$s/str1/str2/g`	im gesamten Text `1,$` werden **alle** Vorkommen von `str1` durch `str2` ersetzt
`3,7w file`	Text von der 3. bis 7. Zeile in Datei `file` schreiben

Textsuche:

Der zu suchende Text (`string`) kann entweder direkt angegeben werden oder über ein **Muster** beschrieben werden.

- suche nächste / vorhergende Stelle:

  steht im Folgenden ein Schrägstrich ("/"), so heißt dies: „suche nächstes Vorkommen“
  steht statt des Schrägstrichs ein Fragezeichen ("?"), so heißt dies: „suche letztes Vorkommen“

- fester String: `:/anton`
  „suche nächste Stelle, an der "anton" vorkommt“

- **reguläre Ausdrücke:**

  `^` steht für "am Zeilenanfang"
  `$` für "am Zeilenende"

  Beispiel:
  `:/^anton$` steht für "suche Zeile, die nur aus "anton" besteht“

  `\< \>` stehen für Wortanfang bzw. Wortende

  Anmerkung: die Zeichen `^ $ / ? * . [ ] \ < >` haben also eine Sonderbedeutung: will man diese selbst suchen, so ist ihnen ein **backslash** voranzustellen.

- **Weitersuchen:** Eingabe von `n` — wiederholt letztes / oder `?`

Zwischendurch etwas anderes machen:

Kommando	Bedeutung
`:!cmd`	Shell-Kommando `cmd` zur Ausführung bringen
`:e file`	`file` editieren

Anmerkungen:

- alle Kommandos mit einem **:** am Anfang werden auf der untersten Bildschirmzeile eingegeben (bewirkt der Doppelpunkt) und müssen mit *return* abgeschlossen werden.
- Dahinter verbirgt sich übrigens ein anderer Editor namens **ex** – dies müssen wir aber nicht weiter beachten!

Kapitel 6

Java – erste Schritte

Software-Entwicklung heißt Modelle bilden – auf unterschiedlichen Ebenen der Abstraktion!

Wir interessieren uns hier vorrangig für die Ebene der Programmiersprache!

6.1 Systeme und Modelle

- Die **Systemtheorie** stellt Definitionen für eine abstrahierende Sichtweise auf beliebige Sachverhalte zur Verfügung

- **System**: Anordnung von Gebilden (**Objekten**, Elementen) – für die jeweilige Sicht abgrenzbare Elemente–, die aufeinander durch Relationen einwirken bzw. miteinander in Relationen stehen und die durch eine Hüllfläche, die Systemgrenze, von ihrer Umgebung abgegrenzt sind (offene vs. geschlossene Systeme)

- **Systemkonzepte**:
  funktionale, strukturelle, hierarchische Betrachtung (ergänzende Sichtweisen)

  – Beim **funktionalen** Konzept steht die Wandlung von Eingabeoperanden zu Ausgabeoperanden, durch die die Funktion des Systems beschrieben wird, im Betrachtungsmittelpunkt

  – Das **strukturelle** Konzept stellt allgemein „Elemente“, ihr „Verhalten“ und die sie verknüpfenden „Relationen“ in den Vordergrund. Ergänzend wird das System gegenüber der Umgebung abgegrenzt.

  – Das **hierarchische** Konzept beschreibt ein System durch sukzessives Zerlegen in Teilsysteme

Modell:

- Ein Modell ist ein System, das ein anderes (reales) System abbildet.
- Eine **Modellbildung** erfordert geeignete Modellierungsmethoden.
- Eine **Methode** ist ein Vorgehensprinzip zur Lösung von Aufgaben.
- Eine Modellierungsmethode umfasst Konstrukte und eine Vorgehensweise, die beschreibt, wie diese Konstrukte wirkungsvoll bei der Modellbildung anzuwenden sind.
- Konstrukte umfassen die Elemente einer „Beschreibungssprache“ und die Regeln für die Verknüpfung der Elemente.

6.2 Objektorientierung – kurz gefasst

6.2.1 Übersicht

Generell: **Strukturelle** Betrachtung eines Systems

- Was sind die relevanten, konkreten Elemente im System? Welche sind passiver, welche aktiver Natur?
- Worüber werden Information benötigt?
- Bildung von **Klassen**: alle „gleichartigen“ Elemente werden zu einer Klasse zusammengefasst
- Was sind die Eigenschaften / Attribute der Objekte / Klassen?
- Welches Verhalten zeigen die Objekte / Klassen?
- ...

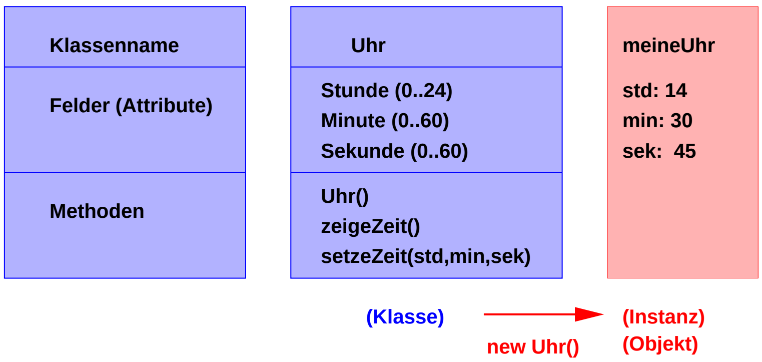


Abbildung 6.1: Klasse: Beispiel Uhr

Klasse: allgemein eine Gruppe von Dingen, Lebewesen oder Begriffen, die gemeinsame Merkmale oder Eigenschaften haben!

- Man kann bei der Analyse eines Systems die Objekte, ihr Verhalten / ihre Methoden und ihren Zusammenhang (strukturell, kausal, ...) in den Mittelpunkt stellen: **Objektorientierte Analyse (OOA)**
- Man kann dies beim Entwurf eines Systems tun: **Objektorientierter Design (OOD)**
- Man kann dies bei der Programmierung tun: **Objektorientierte Programmierung (OOP)**

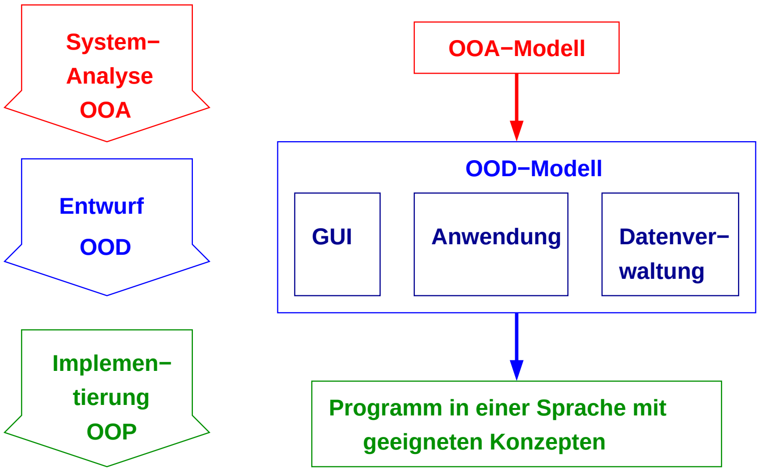


Abbildung 6.2: OOA, OOD, OOP

NB: Der Fokus ist auf den unterschiedlichen Betrachtungsebenen unterschiedlich – auf der Ebene der OOA sind "Methoden" z.B. betriebswirtschaftliche Funktionen, auf der Ebene des OOD z.B. DV-technische Funktionen ("abspeichern", "erzeugen", "freigeben"), auf der Ebene der OOP Hilfsfunktionen zur Realisierung der OOD-Methoden!

6.2.2 OOA

Ziel des Analyseprozesses ist es, ein **System von Objekten** zu finden und zu arrangieren, die **im gemeinsamen Zusammenspiel** das reale System abbilden und die gestellte Aufgabe **mit verteilten Verantwortlichkeiten** erledigen

Beispielklassen: Transportfahrzeuge, Produkte, Lagerplätze, Transportanforderungen, ...

Also:

- zunächst sind die Objekte und Klassen zu finden,
- dann ihre Attribute und Funktionen zu bestimmen
- und zuletzt die Beziehungen der Objekte untereinander festzulegen

Grundregel:

$\boxed{\text{Das einzige Strukturierungsmittel ist das Objekt}}$

Vorgehensweise (z.B.):

- Analyse der (zukünftigen) Benutzerschnittstelle zum System
- Analyse der Anforderungsdokumentation, z.B.
  1. Bestimmung der Klassen: Welche Objekte gibt es im System? (Hauptwörter weisen auf potenzielle Objekte / Klassen)
  2. Gefundene Objekte modellieren: Welche Attribute haben sie? Was sind die identifizierenden / beschreibenden Eigenschaften?
  3. Bestimmung der Methoden der Objekte: Was tun sie, was kann man mit ihnen tun? Welches „Wissen“ haben die Objekte und für welche Operationen sind sie deshalb zuständig?
  4. Art der Beziehung zwischen den Objekten untersuchen (**Klassenbeziehungen**)

6.2.3 OOD

Ziel des Entwurfsprozesses ist es, die endgültige Architektur festzulegen z.B. durch

- Anbindung der Fachklassen an die Benutzungsoberfläche
- Anbindung an die Datenhaltung (Datenbanklösung oder Programmierkonzepte)
- Nutzung von (eigenen) Klassenbibliotheken
- Anpassung und Optimierung auf die Programmiersprache,

6.3 Vorbemerkungen

- Ein Java-Programm besteht aus einer Menge von Klassen
- Aus den einzelnen Klassen werden konkrete Objekte (Instanzen der Klasse) erzeugt
- Jede Klasse kann Attribute und Methoden haben, die klassenspezifisch oder instanzspezifisch sind
- Viele Klassen existieren in Form von Bibliotheken
- Wir haben zur Vereinfachung für den Anfänger eine Klasse zur einfachen Ein- / Ausgabe erstellt – es gibt hier Methoden zum Einlesen von Zahlen, Zeichen oder Zeichenfolgen (Strings), genauso zum Ausgeben
- Java-Programme in Dateien mit der Endung **.java** werden vom Java-Compiler (**javac**) in eine Zwischensprache in einer Datei mit der Endung **.class** übersetzt und können vom Java-Interpreter (**java**) ausgeführt werden (man sagt auch: sie werden auf der *Java Virtual Machine* ausgeführt. Auch hierzu haben wir für Sie als “Anfänger” eine Vereinfachung gebaut.
- Wie Sie diese Vereinfachung bekommen und nutzen können, wird in den Übungen erläutert!

6.4 Wie immer am Anfang: Hello World

Programm 6.1: Hello World (*Hello.java*)

```java
import IOulm.*; // alle unsere Klassen im Paket IOulm zur Ein- / Ausgabe

public class Hello { // alles ist eine Klasse

   public static void main(String [] args) {

      Write.String("Hello World!\n"); // Ausgabe einer Zeichenfolge (String)
      Write.Line("Hello World!"); // Ausgabe einer Zeile

   } // Ende der main()-Methode
} // Ende der Klasse
```

- Name des Programms: **Hello**
- Die Klasse ist als **public** deklariert, d.h. jeder kann sie verwenden – wir werden zunächst hier immer *public* verwenden
- Jedes selbständige Java-Programm benötigt eine **main()**-Methode
- Die Methode ist statisch (*static*), d.h. sie operiert auf der Klasse als solches und nicht auf einer Instanz (Objekt) der Klasse (später mehr dazu!)
- Die Methode liefert keinen Rückgabewert, was durch das dem Methodennamen vorangestellte **void**) ausgedrückt wird
- Die Methode hat einen formalen Parameter namens *args*, der vom Typ "Vektor von Zeichenketten" (Vektor von Strings, Array von Strings) ist – die eckigen Klammern vor `args` besagen, dass `args` ein Vektor ist, das davorstehende Wort `String` besagt, dass dessen Elemente *Strings* sind
- `args` dient dazu, einem Java-Programm beim Auruf auf der Kommandozeile Argumente übergeben zu können
- Der Name der Datei ohne die Endung **.java** muss mit dem in dieser Datei definierten Klassennamen übereinstimmen
- Die beiden Schrägstrich "//" leiten einen **Kommentar** ein – ein Text bis zum Zeilenende, der für den Leser gedacht ist und vom Java-Compiler ignoriert wird
- In unserem **Klassenpaket** (*package*) **IOulm** gibt es eine Klasse namens **Write** – diese enthält u.a. Methoden namens **String** zur Ausgabe einer Zeichenfolge und **Line** zur Ausgabe einer Zeichenfolge als Zeile (d.h. mit einem impliziten *newline* – Ersatzdarstellung des Zeichens ist `\n`) am Ende
- Die geschweiften Klammern stellen eine **Blockstruktur** her
- Die Semikolons schließen einzelne Anweisungen ab
- Groß-/Kleinschreibung ist signifikant

Übersetzung und Ausführung

- Wir haben Ihnen zum Herunterladen einmal die Datei `IOulm.jar` zur Verfügung gestellt. Dabei handelt es sich um ein sog. **Java-Archiv** (Endung **jar**, das die die Klassen `Write` (Einfache Ausgabemethoden für Java) und `Urc` (*Ulmer reader class*: einfache Lesemethoden) bereitstellt.
- Wir haben Ihnen ebenfalls zur Verfügung gestellt die Datei `carj.sh` (`carj` steht für *compile and run java*, `sh` steht für Shell). Damit kann der Übersetzungs- und Ausführungsvorgang vereinfacht werden
- Diese beiden Dateien wie auch die Datei `Hello.java` seien in unserem momentanen Arbeitskatalog abgelegt
- Sollten Sie die `IOulm.jar` nicht in Ihrem aktuellen Arbeitskatalog (also bei Hello.java) abgelegt haben, so ist in der Datei `carj.sh` eine Anpassung vorzunehmen: in der Zeile

  `myClasspath=.:$HOME/ai1/java/ulm/IOulm.jar`

  muss der Teil nach `$HOME` den relativen Pfadnamen der Datei `IOulm.jar` sein. Mit `myClasspath` wird sowohl dem Java-Compiler wie auch dem Java-Interpreter angegeben, wo er nach den Klassen suchen soll – entweder im aktuellen Katalog (das sagt der Punkt) oder eben im nächsten (nach dem Dopplepunkt) Pfad.

Ausgangssituation bei mir:

```
spatz$ pwd
/home/swg/skripte/ai1.05/6/progs
spatz$ ls
Hello.java
spatz$ ls $HOME/lib
IOulm.jar
spatz$ ls $HOME/bin/carj*
/home/swg/bin/carj.sh
spatz$ egrep myClasspath $HOME/bin/carj.sh
myClasspath=.:$HOME/lib/IOulm.jar
javac -classpath $myClasspath $1.java
   java -classpath $myClasspath $1
spatz$
```

- Ich habe mir in meinem Heimatverzeichnis einen Katalog `lib` angelegt und dort `IOulm.jar` abgelegt
- Die Datei `carj.sh` liegt im Katalog `~/bin`
- In dieser Datei ist die o.g. Zeile auf `myClasspath=.:$HOME/lib/IOulm.jar` gesetzt

Übersetzung:

```
spatz$ ls Hello.*
Hello.java
spatz$ carj.sh Hello
Compilation successful!
spatz$ ls Hello.*
Hello.class  Hello.java
spatz$
```

Es entsteht die Datei mit der Endung **.class** – diese enthält das Programm im sog. Byte-Code

Übersetzung mit Ausführung:

```
spatz$ carj.sh -e Hello
Compilation successful!
Starting program...
Hello World!
Hello World!
spatz$
```

Dasselbe "zu Fuß":

```
spatz$ javac -classpath $HOME/lib/IOulm.jar Hello.java
spatz$ java -classpath .:$HOME/lib/IOulm.jar Hello
Hello World!
Hello World!
spatz$
```

Wer es etwas eleganter will:

```
spatz$ export CLASSPATH=".:$HOME/lib/IOulm.jar"
spatz$ echo $CLASSPATH
.:/home/swg/lib/IOulm.jar
spatz$ javac Hello.java
spatz$ java Hello
Hello World!
Hello World!
spatz$
```

Noch besser ist, dies ein für allemal zu erledigen:

```
spatz$ cd
spatz$ cp .bashrc .bashrc.old
spatz$ echo 'CLASSPATH=.:$HOME/lib/IOulm.jar' >> .bashrc
spatz$ echo 'export CLASSPATH' >> .bashrc
spatz$
```

Ich selbst bevorzuge letzteren Weg! Aber bitte aufpassen!!!

Meist gelingt es nicht, im ersten Ansatz ein Programm ohne jeden Syntaxfehler einzugeben – dann “hagelt” es mehr oder weniger Fehlermeldungen vom Compiler.

Programm 6.2 (S. 98) ist eine verfälschte Kopie des Programmes 6.1 (S. 95).

Programm 6.2: Ein Programm mit Syntaxfehlern (*Hello1.java*)

```
1  import IOUlm.*;
2
3  public class Hello {
4
5     public static void main(String [] args) {
6
7        Write.String("Hello World!\n");
8        Write.Line("Hello World!");
9     }
10 }
```

Erster Übersetzungsversuch:

```
spatz$ javac Hello1.java
Hello1.java:9: ';' expected
    }
    ^
1 error
spatz$ exit
```

Dies sieht harmlos aus: Der Compiler vermisst spätestens in Zeile 9 ein Semikolon – wir sollten die Anweisung in Zeile 8 mit einem Semikolon abschliessen.

Der zweite Versuch (mit obiger Korrektur):

```
spatz$ javac Hello1.java
Hello1.java:3: class Hello is public, should be declared in a file
named Hello.java
public class Hello {
       ^
Hello1.java:1: package IOUlm does not exist
import IOUlm.*;
^
Hello1.java:7: cannot resolve symbol
symbol  : variable Write
location: class Hello
      Write.String("Hello World!\n");
      ^
Hello1.java:8: cannot resolve symbol
symbol  : variable Write
location: class Hello
      Write.Line("Hello World!");
      ^
4 errors
spatz$
```

Das sieht ja plötzlich viel schlimmer aus – ist es aber nicht:

1. Der Fehler, der sich auf Zeile 3 bezieht, ist offenkundig: **Der Klassenname muss identisch sein mit dem Datenamen** (ohne Endung) – entweder wir nennen die Klasse `Hello1` oder die Datei `Hello.java`

2. Der Fehler, der sich auf Zeile 1 bezieht, ist subtiler: Unser Klassenpaket scheint nicht zu existieren

   - Ist der Klassenpfad (`CLASSPATH` bzw. Angabe bei `myClasspath` in `carj.sh` falsch?
   - Ist das nach `import` angegebene Paket falsch geschrieben?

   Letzteres ist der Fall: das `U` muss ein kleines `u` sein!

3. Die anderen Fehler mit "`cannot resolve symbol`" sind Folgefehler – schließlich ist das Symbol "`Write`" in `IOulm.jar` definiert, was der Compiler aufgrund der falschen Schreibweise (siehe Punkt 1) aber nicht lokalisieren konnte!

6.5 Noch ein Beispiel: GGT

Berechnung des größten gemeinsamen Teilers GGT zweier positiver ganzer Zahlen (x,y):

Problemwissen:

- $GGT(x, x) = x$
- $GGT(x, y) = GGT(y, x)$
- $x > y \Longrightarrow GGT(x, y) = GGT(x - y, y)$
  Die letzte Aussage bedürfte eines Beweises – wir glauben Sie einfach!

Daraus ergibt sich leicht der Algorithmus, der in Abb. 6.3 (S. 99) dargestellt ist.

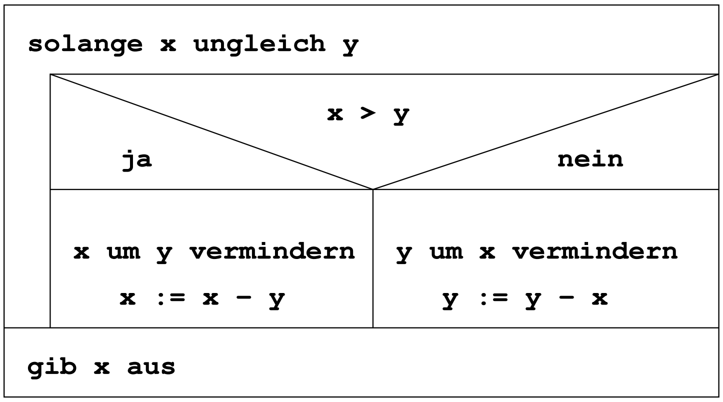


Abbildung 6.3: GGT – Nassi-Shneidermann-Diagramm

Programm 6.3: GGT – Erste Version (*GGT.java*)

```
1  /*
2   * Berechnung des GGT
3   */
4  import IOulm.*;
5
6  public class GGT{
7
8     public static void main(String [] args) {
9
10     int x, x0, y, y0;
11     // Eingabe:
12     Write.String("Gib␣zwei␣ganze␣Zahlen:␣");
13     if ( Urc.readInt() ) { x0 = Urc.getInt(); } else { return; }
14     if ( Urc.readInt() ) { y0 = Urc.getInt(); } else { return; }
15
16     // Verarbeitung :
17     x = x0; y = y0;
18     if ( ( x <= 0 ) || (y <= 0) ) {
19        Write.Line("Fehler␣in␣der␣Eingabe!");
20        Write.String("x␣=␣"); Write.Int(x); Write.String(";␣");
21        Write.String("y␣=␣"); Write.Int(y); Write.Ln();
22        return;
23     };
24     while ( x != y ) {
25         if ( x > y ) { x = x - y; }
26         else { y = y - x; }
27     };
28
29     // Ausgabe
30     Write.String("Der␣GGT␣von␣"); Write.Int(x0);
31     Write.String("␣und␣"); Write.Int(y0);
32     Write.String("␣lautet:␣");
33     Write.Int(x); Write.Ln();
34
35
36     }
37  }
```

Erläuterungen:

1-3 Mit `/*` und `*/` kann ein **mehrzeiliger Kommentar** gefasst werden

10 Hier werden vier Variable `x`, `x0`, `y` und `y0` vom Typ `int` (Ganze Zahl, Integer-Zahl) eingeführt (NB: in der Methode `main`!)

13 Die Klasse **Urc** (*Ulmer reader class*) enthält diverse Methoden zum Einlesen von der Standardeingabe. Generell gilt: Lesen von der Standardeingabe ist sehr diffizil und hängt neben programmiersprachlichen Eigenheiten sowohl von der Implementierung der entsprechenden Methoden wie auch von systemspezifischen Gegebenheiten ab!

- Mit der Methode `Urc.readInt()` werden alle Zahlen einer Zeile der Standardeingabe eingelesen und sukzessive bereitgestellt; diese Methode liefert den **Wahrheitswert** `true`, falls mindestens eine Zahl bereitgestellt werden konnte, ansonsten `false` – an die Stelle dieses Methodenaufrufs wird nach Beendigung dieser Methode der Rückgabewert gestellt.

– wenn (engl.: **if**) also `Urc.readInt()` den Wert `true` lieferte, so geht es weiter bei `x0 = Urc.getInt();` die Methode `Urc.getInt()` liefert die (nächste) bereitgestellte Zahl zurückgeliefert (wird also wieder an die Aufrufstelle gesetzt) und danach wird dieser Wert in die Variable `x0` per Wertzuweisung (Operator "=") geschrieben Diesen Sachverhalt skizziert Abb. 6.4 (S. 101).

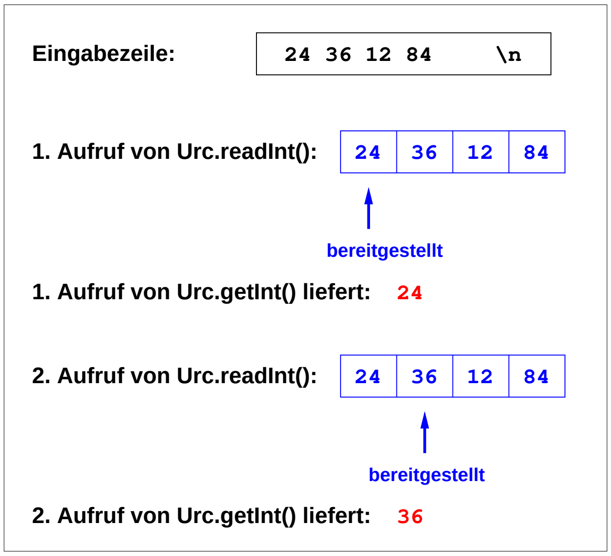


Abbildung 6.4: Einlesen mit Urc.readInt()

– wenn `Urc.readInt()` den Wert `false` lieferte, geht es weiter mit dem "ansonsten" (also **else**): die Anweisung **return** bewirkt die Beendigung der umgebenden Methoden, hier also von *main()* – damit wird das gesamt Programm beendet.

– Die Bedingung, nach der bei `if` verzweigt wird, muss in runden Klammern stehen!

14 wie Zeile 13

17 Die Anweisung `x = x0;` weist der Variablen x den Wert der Variable `x0` zu (analog `y`)

18 Die Bedingung lautet hier: "wenn x kleiner oder gleich 0 oder y kleiner oder gleich 0 ist"

Operator "<=" bedeutet "kleiner oder gleich"

Operator "||" bedeutet "oder"

21 Die Methode `Write.Ln()` gibt einen Zeilenvorschub aus

24-27 Mit `while bedingung { ... }` wird eine **Schleife** definiert: solange die Bedingung wahr ist, wird der Teil zwischen { und } wiederholt

24 Der Operator "`!=`" besagt "ungleich"

25 Die Anweisung `x = x - y;` besagt: "werte den Ausdruck auf der rechten Seite des = aus und nimm diesen Wert als neuen Wert von x"

37 Jede öffnende Klammer braucht ihre schließende Klammer!

Meist ist es geschickter, statt "langatmigen" Eingabedialogen Programm als **Filter** zu schreiben – um es hier noch einfacher zu halten schreiben wir auch die Fehlermeldungen an die Standardausgabe statt an die Diagnoseausgabe. Solange wir also ein (korrektes) Zahlenpaar lesen können, wird deren GGT bestimmt.

Programm 6.4: GGT als Filter (*GGTmultiple.java*)

```java
/*
 * Berechnung des GGT
 */
import IOulm.*;

public class GGTmultiple {

  public static void main(String[] args) {

    int x, x0, y, y0;
    while ( Urc.readInt() ) {
      x0 = Urc.getInt();
      if ( Urc.readInt() ) { y0 = Urc.getInt(); } else { return ; }

      x = x0; y = y0;
      if ( ( x <= 0 ) || (y <= 0) ) {
        Write.String("Fehler in der Eingabe: ");
        Write.String("x = "); Write.Int(x); Write.String("; ");
        Write.String("y = "); Write.Int(y); Write.Ln();
      } else {
        while ( x != y ) {
          if ( x > y ) { x = x - y; } else { y = y - x; }
        };
        Write.String("GGT von "); Write.Int(x0);
        Write.String(" und "); Write.Int(y0);
        Write.String(": "); Write.Int(x); Write.Ln();
      }
    }
  }
}
```

```
spatz$ javac GGTmultiple.java
spatz$ cat zahlen
24 36 40 80
88 66
-4 9
45 65
12
16
13
spatz$ java GGTmultiple < zahlen
GGT von 24 und 36: 12
GGT von 40 und 80: 40
GGT von 88 und 66: 22
Fehler in der Eingabe: x = -4; y = 9
GGT von 45 und 65: 5
GGT von 12 und 16: 4
spatz$
spatz$
```

In Programm 6.4 (S. 102) sind die Strukturen bereits ziemlich verschachtelt – die hierarchische Struktur wird in Abbildung 6.5 (S. 103) deutlich.

```
public class GGTmultiple {

   public static void main( String [] args) {

      int x, x0, y, y0;
      while ( Urc.readInt() ) {

         x0 = Urc.getInt();

         if ( Urc.readInt() ) { y0 = Urc.getint() } else {return; }
         x = x0; y = y0;
         if ( (x <= 0) || (y <= 0) ) {
            Write.String("Fehler in der Eingabe: ");
            Write.String(x = "); Write.Int(x); Write.String("; ");
            Write.String(y = "); Write.Int(y); Write.Ln();
         } else {
            while ( x!= y ) {
               if ( x > y ) { x = x - y; } else { y = y - x;};
            }
         }
      }
   }
}
```

Abbildung 6.5: GGT als Filter – hierarchische Anweisungsstruktur

6.6 Lesbarkeit von Programmen

Programm 6.5: Korrekt – aber lesbar? (*Hello2.java*)

```java
import IOulm.*;

public class Hello2 {
   public static
void main(String [] args) {   Write.String("Hello World!\n")
; Write.Line("Hello World!");
} }
```

- Im Vordergrund steht die Lesbarkeit des Programmtexts für den Entwickler und nicht für den Übersetzer.
- Deswegen ist die disziplinierte Verwendung eines auf Lesbarkeit ausgerichteten Stils essentiell.
- Schopenhauer: “Wer nachlässigt schreibt, legt dadurch zunächst das Bekenntnis ab, daß er selbst seinen Gedanken keinen großen Wert beilegt!”
- Nietzsche: “Den Stil verbessern — das heißt den Gedanken verbessern und nichts weiter.”
- L. Reiners: “[...] der Kampf um den Ausdruck ist ein Kampf um den Inhalt. Um einen Gedanken knapp und kristallklar zu formulieren, muß man ihn bis zum Ende durchdacht haben; [...] die Form ist der Prüfstein des Gehalts.”

Kapitel 7

Die Sprache etwas detaillierter

Die wichtigste Web-Site ist **java.sun.com**

Wichtige URLs:

- http://java.sun.com/reference/docs/index.html
- http://java.sun.com/j2se/1.4.2/docs/api/java/math/package-summary.html
  für das Paket **java.math**
- http://java.sun.com/j2se/1.4.2/docs/api/java/util/package-summary.html
  für das Paket **java.util**
  u.a. für verschiedene Containerklassen (Listen: dynamische Arrays, verlinkte Listen, ...)
- http://java.sun.com/j2se/1.4.2/docs/api/java/io/File.html
  für die Klasse **File**: **java.io**

Zu Java gehört eine umfangreiche Klassenbibliothek. Dem Programmierer wird damit eine einheitliche, vom zugrunde liegenden Betriebssystem unabhängige Schnittstelle (*Application programming interface, API*) angeboten.

7.1 Der Zeichensatz

Java arbeitet mit dem Unicode-Zeichensatz:

- Einbetten von Unicode-Zeichen in Java-Programme: `\uxxxx`, x ein Hex-Zeichen (Bsp.: `\u0020` ist das Blank, `\u03c0` ist das Zeichen π
- Unicode-Zeichen können an beliebiger Stelle in Java-Programmen verwendet werden, in Kommentaren, Literalen wie auch in Variablennamen
- Man kann natuerlich einfach im ASCII-Zeichenbereich bleiben: Bereich 32 bis 126 und zusätzlich Leerzeichen und Zeilentrenner

Literale (ganz allgemein):

- Ein Literal ist ein durch seine formale Sprache festgelegter Name eines Wertes.
- Ein Literal ist ein elementarer Ausdruck. Dabei bedeutet "elementar", dass ein Literal sich nicht in Teilausdrücke zerlegen lässt.
- Der Wert eines Literals ist durch die verwendete formale Sprache eindeutig bestimmt. Das bedeutet, dass
  - der Wert eines Literals bereits durch den Quelltext festgelegt ist und nicht erst später (etwa während einer Interpretation) bestimmt wird
  - der Wert eines Literals nicht irgendwie "umdefiniert" werden kann, sondern in jedem Zusammenhang der gleiche ist.
- Falls der Wert eine Zahl ist, dann spricht man auch von einem **Numeral**.
- Literale werden gelegentlich auch als "Konstanten" bezeichnet, doch sollte die Bezeichnung "Konstante" denjenigen Namen vorbehalten werden, deren Wert nicht bereits durch ihre formale Sprache definiert ist. Der Wert einer Konstante kann beispielsweise in verschiedenen Texten einer formalen Sprache unterschiedlich definiert sein.
- Der Wert des Numerals "0" ist in vielen formalen Sprachen gleich dem Wert des Numerals "00". Es sind also zwei verschiedene Numerale, die aber den gleichen Wert bezeichnen. Literale sind also nicht das gleiche wie Werte.

7.2 Bezeichner, reservierte Schlüsselworte

Bezeichner:

- beginnt mit einem Buchstaben, dem *underscore*(„_“), oder einem Unicode-Währungssymbol (letzeres sollte man vermeiden, da diese per Konvention für Bezeichner reserviert sind, die vom Compiler automatisch generiert werden)
- gefolgt von einer beliebigen Folge von Buchstaben, *underscore*, Ziffern oder Unicode-Währungssymbolen

Reservierte Worte:

abstract	assert***	boolean	break	byte
case	catch	char	class	const *
continue	default	do	double	else
enum ****	extends	final	finally	float
for	goto *	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp **	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

* nicht benutzt / ** hinzugekommen in 1.2 /

*** hinzugekommen in 1.4 / **** hinzugekommen in 5.0

(nach: `java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html`)

7.3 Einfache Datentypen

7.3.1 Übersicht

Typ	**enthält**	**Standardwert**	Größe	Wertebereich
boolean	true, false	false	1 Bit	–
char	Unicodezeichen	`\u0000`	16 Bit	`\u0000` ... `\uffff`
byte	ganze Zahl	0	8 Bit	-128 ... 127
short	ganze Zahl	0	16 Bit	-32768 ... 32767
int	ganze Zahl	0	32 Bit	$-2^{32-1} \dots 2^{32-1} - 1$
long	ganze Zahl	0	64 Bit	$-2^{64-1} \dots 2^{64-1} - 1$
float	Gleitpunktzahl	0.0	32 Bit	$\pm 1.4E - 45 \dots \pm 3.4028235E + 38$
double	Gleitpunktzahl	0.0	64 Bit	$\pm 4.9E - 324 \dots \pm 1.7976931348623157E306$

Anm.: Zu diesen einfachen Datentypen gibt es jeweils auch entsprechende Klassen (sog. **Wrapper-**Klassen , die soweit notwendig angesprochen werden. Sie enthalten insbesondere eine Reihe nützlicher Methoden.

7.3.2 boolean

Die Werte sind *false* und *true* und entsprechenden den üblichen Wahrheitswerten der Logik. An dieser Stelle sein an die elementaren Operatoren und Regeln der Logik erinnert.

Seien `a,b,c` Variable vom Typ `boolean`. Dann sind folgende logische Verknüpfungen definiert:

Negation

a	$\neg a$
true	**false**
false	**true**

Konjunktion

a	b	$a \wedge b$
true	**true**	**true**
true	**false**	**false**
false	**true**	**false**
false	**false**	**false**

Disjunktion

a	b	$a \vee b$
true	**true**	**true**
true	**false**	**true**
false	**true**	**true**
false	**false**	**false**

Äquivalenz

a	b	$a \equiv b$
true	**true**	**true**
true	**false**	**false**
false	**true**	**false**
false	**false**	**true**

Implikation

a	b	$a \rightarrow b$
true	**true**	**true**
true	**false**	**false**
false	**true**	**true**
false	**false**	**true**

In der mathematischen Aussagenlogik gelten die folgenden Regeln:

1.	$\neg\neg a \equiv a$	Idempotenz
2.	$a \wedge b \equiv b \wedge a, a \vee b \equiv b \vee a$	Kommutativität
3.	$(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$ $(a \vee b) \vee c \equiv a \vee (b \vee c)$	Assoziativität
4.	$(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$ $(a \vee b) \wedge c \equiv (a \wedge c) \vee (b \wedge c)$	Distributivität
5.	$a \wedge true \equiv a, a \wedge false \equiv false, a \wedge a \equiv a$ $a \vee true \equiv true, a \vee false \equiv a, a \vee a \equiv a$	Kürzungsregeln
6.	$a \wedge (\neg a) \equiv false, a \vee (\neg a) \equiv true$	
7.	$a \rightarrow b \equiv (\neg a) \vee b$	
8.	$\neg(a \wedge b) \equiv (\neg a) \vee (\neg b)$ $\neg(a \vee b) \equiv (\neg a) \wedge (\neg b)$	Regeln von de Morgan

In Java: Negation ($\neg$) ist **!**, Konjuktion ($\wedge$) ist **&&**, Disjunktion ($\vee$) ist **||**

Variable oder Ausdrücke vom Typ *boolean* erhalten ihren Wert sehr häufig als Ergebnis von Vergleichen. Dies kann dazu führen, dass das Ergebnis eines Vergleiches überhaupt nicht definiert ist.

Beispiel: `x / y > 0` – was passiert, wenn `x` den Wert 5, aber `y` den Wert 0 hat? Der Wert des Vergleiches ist weder `true` noch `false`, sondern undefiniert.

Kurzschlussbewertung

- Während in der mathematischen Notation eine boolsche Variable oder ein boolscher Ausdruck nur die Werte **true** und **false** annehmen kann, gibt es bei Programmen auch noch die Möglichkeit, dass die Evaluation einer der Operanden eines boolschen Operators entweder undefiniert ist oder zu einem Laufzeitfehler führt.
- Beispiel: Was passiert bei der Bewertung von
  `(j > 0) && (i / j = 0)` falls `j = 0`?

  Hier würde die Bewertung des rechten Operanden des `&&`-Operators zu einem Laufzeitfehler führen.

- Java (und viele andere Programmiersprachen) lassen dies zu. Sie bewerten bei `&&` und `||` zunächst den linken Operanden und betrachten den rechten Operanden dann und nur dann, wenn das Resultat tatsächlich davon abhängt. Das wird als **Kurzschlussbewertung** bezeichnet.

- Interpretation von `a && b`:

```
if ( a ) {
   result := b; /* b wird bewertet */
} else {
   result := false; /* b bleibt unbewertet */
        }
```

- Interpretation von `a || b`:

```
if ( a ) {
   result := true; /* b bleibt unbewertet */
} else {
   result := b; /* b wird bewertet */
        }
```

- Entsprechend verlieren diese Operatoren die Eigenschaft der Kommutativität in Java.

Programm 7.1 (S 109) soll dies verdeutlichen.

Programm 7.1: Kurzschlussbewertung (*Logik.java*)

```java
import IOulm.*;
public class Logik {
   public static void main(String[] args) {

      boolean a = false;
      int x = 5, y = 0;

      if ( !a || (x/y == 1) ) { Write.Line("Hurra"); }
      if ( (x/y == 1) || !a ) { Write.Line("???"); }

   }
}
```

Übersetzung und Ausführung:

```
spatz$ javac Logik.java
spatz$ java Logik
Hurra
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Logik.main(Logik.java:9)
spatz$
```

Hinweis: Java kennt sowohl ein Konditionales UND / ODER (`&&` / `||`) (Kurzschlussbewertung) als auch ein logisches UND / ODER (`&` / `|` – keine Kurzschluss-, sondern vollständige Bewertung) – siehe dazu Programm 7.2, S. 110, dessen Ausführung liefert:

```
spatz$ javac Logik1.java
spatz$ java Logik1
Hurra -- Kurzschluss :-)
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Logik1.main(Logik1.java:8)
spatz$
```

Programm 7.2: Konditional vs. Logisch (*Logik1.java*)

```java
import IOulm.*;
public class Logik1 {
    public static void main(String[] args) {

        boolean a = false;
        int x = 5, y = 0;

        if ( !a || (x/y == 1) ) { Write.Line("Hurra -- Kurzschluss :-)"); }
        if ( !a | (x/y == 1) ) { Write.Line("Hurra -- die Logik :-("); }

    }
}
```

7.3.3 char

- **char** repräsentiert Unicode-Zeichen; Literale werden in einfache Apostrophen gefasst, über eine Unicode-Escape-Sequenz (`\uxxxx`, x Hexzahl) oder über eine einfache Escape-Sequenz (siehe Tabelle) definiert

- Beispiel:

```
char tab = '\t', apostrophe = '\'', nul = '\000',
     aleph = '\u05D0';
```

  aleph: ℵ (semitisches Zeichen)

- Escape-Sequenzen

Escape-Sequenz	Zeichenwert
`\b`	*backspace*
`\t`	horizontaler Tab
`\n`	*newline*
`\r`	"Wagenrücklauf"
`\"`	Doppelapostroph
`\'`	einfacher Apostroph
`\\`	*backslash*
`\xxx`	Das Latin-1-Zeichen mit der Oktalcodierung `xxx`; die Angabe von drei Oktalziffern ist empfehlenswert aber nicht notwendig
`\uxxxx`	Das Unicode-Zeichen mit der Hex-Codierung `xxxx`; können an beliebiger Stelle in Java-Programmen stehen, nicht nur in Zeichen- oder String-Literalen

- *char* ist eine **vorzeichenlose** 16-Bit-Integer!

- Klasse **Character** definiert eine Reihe statischer Methoden wie *getNumericValue()*, *isDigit()*, *isLetter()*, *isLowerCase()* oder *toUpperCase()* (siehe Programm 7.3, S. 112)

  – nachzulesen z.B. in `java.sun.com/j2se/1.3/docs/api/java/lang/Character.html`

- Unsere Klassen `IOulm.Write` und `Urc.Read` enthalten Ein-/Ausgabe-Funktionen:
  - in Urc: `static boolean readChar()`
    stellt im Erfolgsfall ein Zeichen aus der Standardeingabe bereit
  - in Urc `static char getChar()`
    liefert ein bereitgestelltes Zeichen zurück
  - in Write: `static void Char(char ch)`
    gibt das Zeichen "ch" aus

Programm 7.3: Datentyp char (*Char.java*)

```java
import IOulm.*;

public class Char {

    public static void main(String[] args) {
        int l = 0, d = 0, n = 0, sum = 0; char ch;

        while ( Urc.readChar()) {
            ch = Urc.getChar();
            sum = sum + Character.getNumericValue(ch);
            if ( Character.isDigit(ch) ) d = d + 1;
            if ( Character.isLetter(ch) ) l = l + 1;
            if ( ch == '\n' ) n = n + 1;
            if (Character.isLowerCase(ch))
                ch = Character.toUpperCase(ch);
            Write.Char(ch);
        }
        Write.Line("========================================");
        Write.Line("Summary: ");
        Write.String("Number of Letters: "); Write.Int(l); Write.Ln();
        Write.String("Number of Digits:  "); Write.Int(d); Write.Ln();
        Write.String("Number of Lines:   "); Write.Int(n); Write.Ln();
        Write.String("Checksum:            "); Write.Int(sum); Write.Ln();
    }
}
```

Ausführung von Programm 7.3, S. 112:

```
spatz$ cat text
Dies ist ein
Text mit

4 Zeilen!
spatz$ javac Char.java
spatz$ java Char < text
DIES IST EIN
TEXT MIT

4 ZEILEN!
========================================
Summary:
Number of Letters: 23
Number of Digits:  1
Number of Lines:   4
Checksum:            497
spatz$
```

Beispiel: In einem Text, der von der Standardeingabe zu lesen ist, sind Ziffernfolgen (Dezimal) enthalten. Diese sollen jeweils in eine Zahl umgewandelt und aufsummiert werden, Die Summe ist auszugeben. Es kann davon ausgegangen werden, dass sowohl die einzelnen Zahlen wie auch die entstehende Summe nicht zu groß werden, also in ein Objekt vom Typ `int` passen!

Lösung:

- Die Erkennung einer Zeichenfolge als Ziffernfolge kann sehr leicht mit einem (7.1, S. 113) Automaten beschrieben werden.

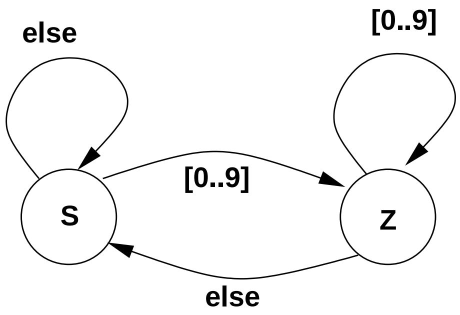


Abbildung 7.1: Zeichenfolge als Ziffernfolge erkennen

- Die beiden Zustände `Z` und `S` können interpretiert werden als "innerhalb einer Zifferfolge" bzw. "nicht innerhalb einer Zifferfolge", was sich wiederum sehr leicht über eine Variable `inNumber` vom Typ `boolean` beschreiben lässt.
- Bei der Abarbeitung des Automaten gibt es also zunächst nur die Unterscheidung zwischen den beiden Zuständen:

```
if ( inNumber ) { // Zustand Z ...
} else { // Zustand S ...
}
```

- Bei jedem der beiden Zustände ist wiederum eine (einfache) Fallunterscheidung zu treffen: entweder ist das gelesene Zeichen eine Ziffer (`[0..9]`) oder keine Ziffer, d.h. innerhalb obiger Fallunterscheidungen gibt es wiederum eine (einfache) Fallunterscheidung nach dem gelesenen Zeichen.
- Das Wandeln einer Ziffernfolgen in eine Zahl folgt dem Hornerschema.

  Eine Ziffernfolge $z_N z_{N-1} \dots z_1 z_0$ ist eine Kurzschreibweise für das Polynom

  $\sum_{i=0}^{N} z_i * 10^i = (((z_N * 10 + z_{N-1}) * 10 + z_{N-2}) * 10 + \dots) * 10 + z_1) * 10 + z_0$
- Beim Übergang von `S` nach `Z` initialisieren wir eine Summen-Variable mit dem Zahlenwert der gelesenen Ziffer – diesen erhält man mit der Methode

  ```
  static int digit(char ch, int radix)
  ```

  aus der Char-Klasse `Character`.

  Bei jeder erneut gelesenen Ziffer wird die bisherige Summe mit 10 multipliziert und der Zahlenwert der jetzt gelesenen Ziffer hinzuaddiert.

- Bei jedem Übergang von Z nach S wird die aufgesammelte Zahl zur Gesamtsumme addiert.

Programm 7.4: Ziffernfolge in Zahl wandeln (*ToNumber.java*)

```java
import IOulm.*;

public class ToNumber {

    public static void main(String[] args) {
        char ch; int z = 0, sum = 0; boolean inNumber = false;

        while ( Urc.readChar() ) {
            ch = Urc.getChar();
            if ( inNumber ) {
                if ( Character.isDigit(ch) ) {
                    z = z * 10 + Character.digit(ch,10);
                } else {
                    sum = sum + z; inNumber = false; z = 0;
                }
            } else {
                if ( Character.isDigit(ch) ) {
                    z = Character.digit(ch,10); inNumber = true;
                }
                else { /* inNumber bleibt false */
                };
            }
        };
        Write.String("Summe: "); Write.Int(sum); Write.Ln();
    }
}
```

Test von Programm 7.4 (S. 114):

```
spatz$ cat zahlentext
Wir haben +28 Rinder und
016 Esel sowie 256 Huehner.
Es sind 4-2- Katzen.
spatz$ java ToNumber < zahlentext
Summe: 306
spatz$
```

7.3.4 Integer-Typen

$$byte \subset short \subset int \subseteq long \ (\subset float \subset double)^{1}$$

- Unterscheidung: darstellbarer Zahlbereich / Speicherplatzbedarf
- **Alle** sind mit Vorzeichen, es gibt kein vorzeichenlosen (also nicht-negativen) Zahlen!
- Literale sind Folgen von Ziffern, ein Minuszeichen davor ist der unäre Operator und gehört nicht zur Zahl!
  - ist das erste Zeichen eine Null und das zweite in "x", so folgen Hexadezimalziffern
  - ist das erste Zeichen eine Null, das zweite kein "x", so folgen Oktalziffern
  - alles andere sind Dezimalzahlen
- Ein Integer-Literal ist zunächst vom Typ *int* (also 32 Bit) – wird ein **L** oder **l** angehängt, wird es zum Typ *long* (also 64 Bit)
- *byte* und *short*-Literale haben keine spezielle Syntax

Programm 7.5 (S. 115) zeigt einige Beispiele mit Literalen.

Programm 7.5: Integer-Literale (*MyInt.java*)

```java
import IOulm.*;
public class MyInt{
   public static void main(String [] args) {
      byte b = 0177; short s = 0x7fff;
      int i = 01000; long l = 0x7FFFFFFFFFFFFFFFl;
//    long l1 = 0x7FFFFFFFFFFFFFFF;
      Write.String("b = "); Write.Byte(b); Write.Ln();
      Write.String("s = "); Write.Short(s); Write.Ln();
      Write.String("i = "); Write.Int(i); Write.Ln();
      Write.String("l = "); Write.Long(l); Write.Ln();
//    Write.String("l1 = "); Write.Long(l1); Write.Ln();
   }
}
```

Übersetzung und Ausführung von Programm 7.5, S. 115:

```
spatz$ javac MyInt.java
spatz$ java MyInt
b = 127
s = 32767
i = 512
l = 9223372036854775807
spatz$
```

[1] float und double sind Gleitkommatypen und werden im nächsten Abschnitt behandelt

Nimmt man in den Zeilen 6 und 11 die Kommentare weg, so erhält man bei der Übersetzung:

```
spatz$ javac MyInt-a.java
MyInt-a.java:6: integer number too large: 7FFFFFFFFFFFFFFF
      long l1 = 0x7FFFFFFFFFFFFFFF;
                ^
1 error
spatz$
```

Integer-Klassen

- Klasse **Byte**
- Klasse **Short**
- Klasse **Integer** (für *int*)
- Klasse **Long**

jeweils mit Definition der Konstanten **MIN_VALUE** und **MAX_VALUE** sowie diversen Methoden

Programm 7.6 (S. 116) verwendet alle Datentypen und zeigt deren jeweiligen Wertebereich auf einem Intel-PC.

Programm 7.6: Integer-Zahlbereiche (*MyInt1.java*)

```java
import IOulm.*;
public class MyInt1{
  public static void main(String[] args) {

    Write.String("byte:  "); Write.Byte(Byte.MIN_VALUE);
    Write.String(" bis "); Write.Byte(Byte.MAX_VALUE);
    Write.Ln();

    Write.String("short: "); Write.Short(Short.MIN_VALUE);
    Write.String(" bis "); Write.Short(Short.MAX_VALUE);
    Write.Ln();

    Write.String("int:   "); Write.Int(Integer.MIN_VALUE);
    Write.String(" bis "); Write.Int(Integer.MAX_VALUE);
    Write.Ln();

    Write.String("long:  "); Write.Long(Long.MIN_VALUE);
    Write.String(" bis "); Write.Long(Long.MAX_VALUE);
    Write.Ln();
  }
}
```

Ausführung von Programm 7.6 (S. 116):

```
spatz$ java MyInt1
byte:  -128 bis 127
short: -32768 bis 32767
int:   -2147483648 bis 2147483647
long:  -9223372036854775808 bis 9223372036854775807
spatz$
```

Darstellung negativer ganzer Zahlen

- Vorzeichen + Absolutbetrag (Spiegelung am Nullpunkt)?

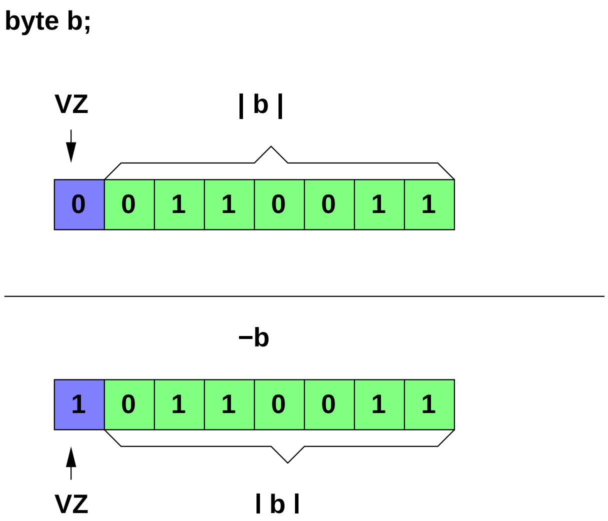


Abbildung 7.2: Negative Zahl: Vorzeichen + Absolutbetrag

Nachteil: Spezielle Subtraktion nötig!

- Lineartransformation: 1-er-Komplement (Basis-1-Komplement)

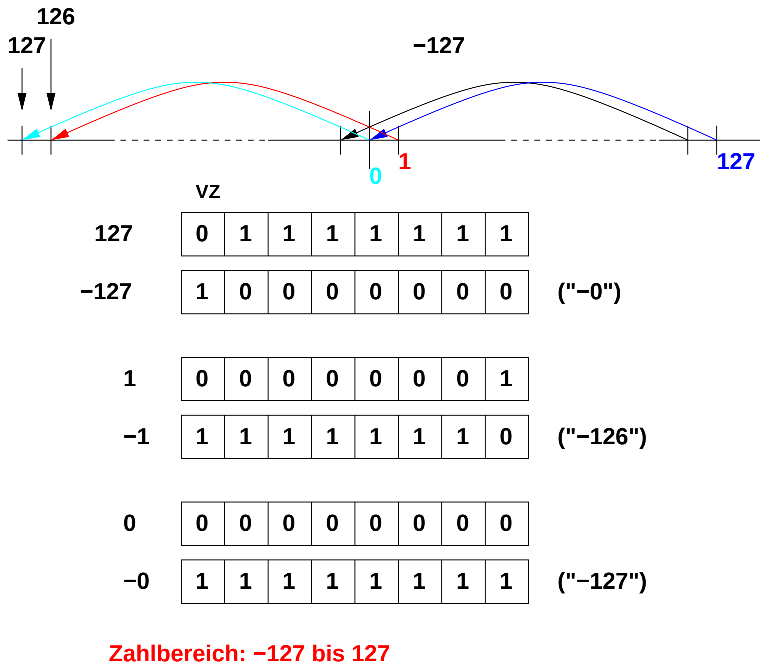


Abbildung 7.3: 1-er-Komplement

Vorgehen: Bits “kippen”
Subtraktion wird zu Addition (vereinfacht, ohne Sonderfälle):

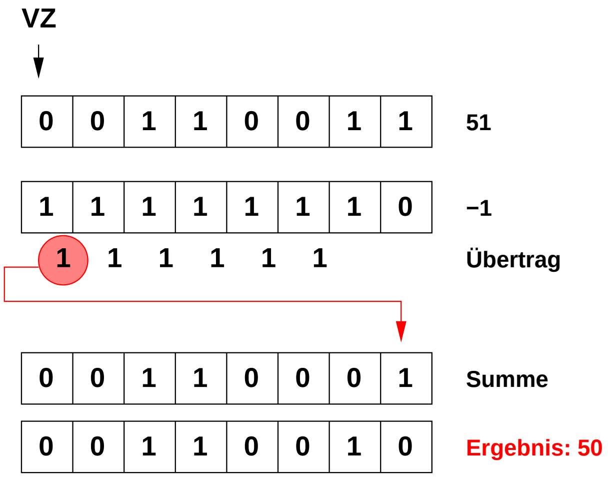


Abbildung 7.4: Subtraktion bei 1-er-Komplement

- Lineartransformation: 2-er-Komplement (Basis-Komplement)

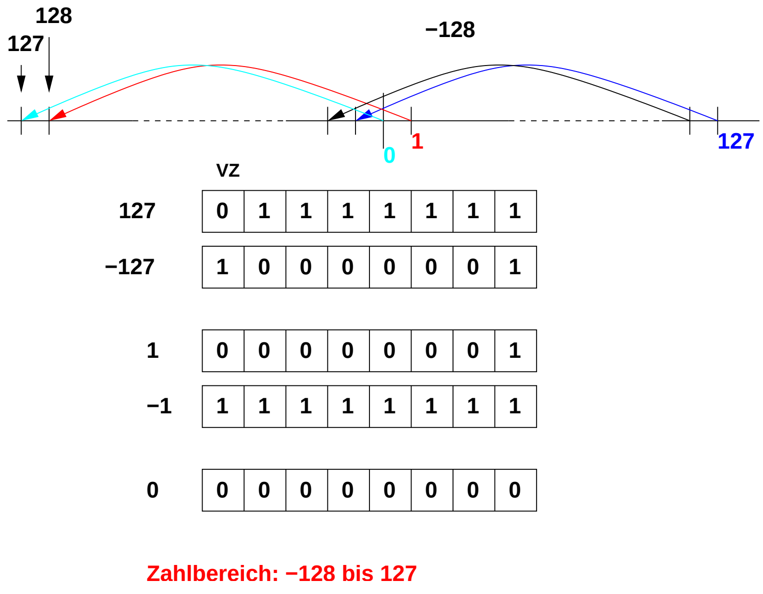


Abbildung 7.5: 2-er-Komplement

Vorgehen: Bits "kippen" und 1 addieren
Subtraktion wird zu Addition (vereinfacht, ohne Sonderfälle):

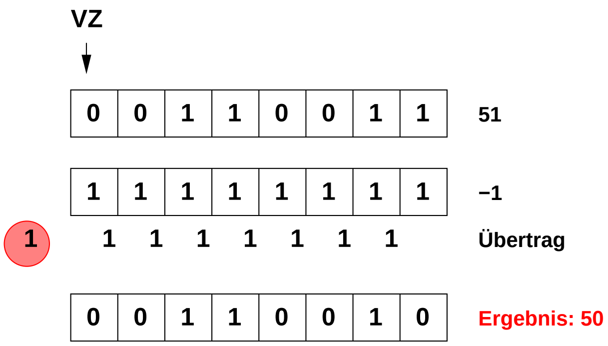


Abbildung 7.6: Subtraktion bei 2-er-Komplement

7.3.5 Reelle Zahlen / Gleitkommatypen: float, double

- Darstellung und arithmetisches Verhalten entsprechend IEEE 754-1985
- Literale: 123.456 – 0.0 – .01 – 0. – $1e-6$ – 1.2E9
- per default sind Literale *double*
- *float*-Literale: nachgestelltes **F** oder **f**

Genauigkeit

Programm 7.7, S. 120, zeigt die Problematik: Eine Formel wird auf 4 verschiedenen, mathematisch identischen Wegen ausgewertet. Man beachte aber, dass ein Compiler meist Optimierungen vornimmt, die die Reihenfolge der Auswertung durchaus verändern können.

Programm 7.7: Genauigkeitsprobleme (*Reals.java*)

```java
// float, double: Berechnung von 9*x*x*x*x - y*y*y*y + 2*y*y;

import IOulm.*;
public class Reals {
  public static void main(String[] args) {
     float x,y,z1,z2,z3,z4; int i,j;

     x = 10864f; y = 18817.0f; i = 10864; j = 18817;

     z1 = 9*x*x*x*x - y*y*y*y; z1 = z1 + 2*y*y;
     Write.String("1. Ergebnis: "); Write.Real(z1); Write.Ln();

     z2 = 9*x*x*x*x; z2 = z2 + 2*y*y; z2 = z2 - y*y*y*y;
     Write.String("2. Ergebnis: "); Write.Real(z2); Write.Ln();

     z3= 2*y*y; z3 = z3 - y*y*y*y; z3 = z3 + 9*x*x*x*x;
     Write.String("3. Ergebnis: "); Write.Real(z3); Write.Ln();

     z4 = 9*x*x*x*x - y*y*y*y + 2*y*y;
     Write.String("4. Ergebnis: "); Write.Real(z4); Write.Ln();

     Write.String("Ganzzahlig gerechnet: ");
     Write.Int(9*i*i*i*i - j*j*j*j + 2*j*j); Write.Ln();
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac Reals.java
spatz$ java Reals
1. Ergebnis: 7.08158976E8
2. Ergebnis: 0.0
3. Ergebnis: 0.0
4. Ergebnis: 7.08158976E8
Ganzzahlig gerechnet: 1
spatz$
```

Woran liegt dies?

- Wegen der endlichen Stellenzahl in einem Rechner können reelle Zahlen nicht immer genau dargestellt werden!
- Die interne Darstellung einer reellen Zahl x erfolgt in der Form

$$x = (\sum_{i=1}^{N} g_i * 2^{-i}) * 2^{exp}$$

mit festem N und einer festen Stellenzahl zur Darstellung des ganzzahligen Exponenten `exp` — die g_i sind 0 oder 1 (g_1 ist 1, sofern $x \neq 0$).

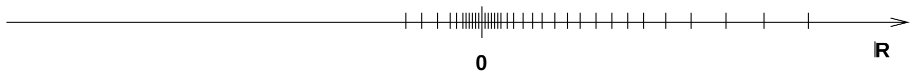


Abbildung 7.7: Maschinenzahlen

Klar ist, dass es nur endlich viele reelle Zahlen im Rechner (sog. Maschinenzahlen) gibt. Weiterhin sieht man leicht, dass der Abstand zwischen zwei benachbarten Maschinenzahlen immer größer wird, je größer die Zahlen sind. Das bedeutet aber auch, dass ein kleiner Fehler (das letzte Bit der Mantisse wird "gekippt") bei großen Zahlen eine sehr große Wirkung hat.

Zur Erläuterung sei ein Dezimalrechner mit $N = 3$ und zwei Stellen für den Exponenten unterstellt (Abb. 7.8, S. 121)

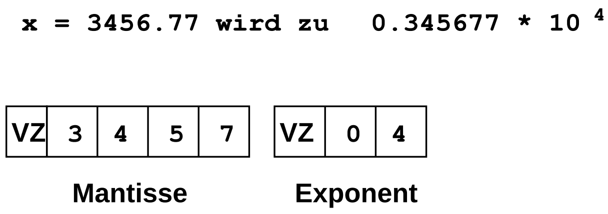


Abbildung 7.8: Darstellung reeller Zahlen

Betrachten wir die Addition zweier reeller Zahlen:

$1234.0 + 0.1234 = 0.1234 * 10^4 + 0.1234 * 10^0 =$

$0.1234 * 10^4 + 0.00001234 * 10^4 =$

(wegen der Beschränkung auf 4 Stellen für die Mantisse)
$0.1234 * 10^4 + 0.0000 * 10^4 = 0.1234 * 10^4$

Dies ist ziemlich falsch!!!

Beim Arbeiten mit reellen Zahlen macht es auch keinen Sinn zwei Variablen auf Gleichheit zu testen (siehe Programm 7.8, S. 122) – die Gleichheitsrelation ist ==

Programm 7.8: Gleichheit bei float / double (*Equals.java*)

```java
import IOulm.*;
public class Equals{
  public static void main(String[] args) {

     double x = 0.3333333333333333E-20; double z = 0.7777777777777777;
     double v = 9999999999999999.0;
     double y = x * v + (1.0 - z);
     y = ( y + z - 1 ) / v;
     // y = { [ x * v + (1.0 - z) ] + z - 1 } / v = x
     if ( x == y )
           Write.Line("Gleich!");
     else
           Write.Line("Ungleich!");
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac Equals.java
spatz$ java Equals
Ungleich!
spatz$
```

NB: Der Divisionsoperator **/** ist bei ganzzahligen Operanden die ganzahlige Division – 1/3 gibt 0!

Statt auf Gleichheit zu testen, sollte man sich eine Genauigkeitsschranke vorgeben und verlangen, dass der Absolutbetrag der Differenz kleiner als diese Schranke ist (siehe Programm 7.9, S. 122).

Programm 7.9: Gleichheit bei float / double: Genauigkeitsschranke (*Equals1.java*)

```java
import IOulm.*;
public class Equals1{
  public static void main(String[] args) {

     double eps = 1E-12;
     double x = 0.3333333333333333E-20; double z = 0.7777777777777777;
     double v = 9999999999999999.0;
     double y = x * v + (1.0 - z);
     y = ( y + z - 1 ) / v;
     if ( Math.abs(x - y ) < eps )
           Write.Line("Gleich!");
     else
           Write.Line("Ungleich!");
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac Equals1.java
spatz$ java Equals1
Gleich!
spatz$
```

Anm.: Die Java-Klasse **Math** stellte eine Reihe mathematischer Funktionen bereit, so unter anderem den Absolutbetrag **abs()**
(siehe z.B. `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html`)

Spezielle Werte: positiv und negativ unendlich (**Infinity**, **-Infinity**, negative Null (**-0.0**) und „keine Zahl“ (**NaN**) – siehe dazu Programm 7.10, S. 123.

Programm 7.10: float, double: Spezielle Werte (*Unendlich.java*)

```java
import IOulm.*;

public class Unendlich{

  public static void main(String [] args) {
      double inf = 1.0/0.0;
      double neginf = -1.0/0.0;
      double negzero = -1.0/inf;
      double Not_a_Number = 0.0/0.0;
      double x = 2.5, y;

     Write.String("positiv unendlich: "); Write.Real(inf); Write.Ln();
     Write.String("negativ unendlich: "); Write.Real(neginf); Write.Ln();
     Write.String("negative Null: "); Write.Real(negzero); Write.Ln();
     Write.String("not a number: "); Write.Real(Not_a_Number); Write.Ln();
     y = x / inf;
     Write.String("x / inf = "); Write.Real(y); Write.Ln();

     if ( inf == Double.POSITIVE_INFINITY ) Write.Line("Hurra zum Ersten");
     if ( neginf == Double.NEGATIVE_INFINITY ) Write.Line("Hurra zum Zweiten");
     if ( negzero == 0.0) Write.Line("Hurra zum Dritten");

     y = inf + neginf;
     Write.String("inf + neginf  = "); Write.Real(y); Write.Ln();
     if ( Double.isNaN(y) )
        Write.Line(" unendlich + -unendlich ist undefiniert");
     if ( negzero == 0.0 )
        Write.Line("-0 ist gleich +0");
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac Unendlich.java
spatz$ java Unendlich
positiv unendlich: Infinity
negativ unendlich: -Infinity
negative Null: -0.0
not a number: NaN
x / inf = 0.0
inf + neginf  = NaN
 unendlich + -unendlich ist undefiniert
-0 ist gleich +0
spatz$
```

Korrespondierende Klassen: Float, Double

definieren u.a. Konstanten **MIN_VALUE**, **MAX_VALUE**, **NEGATIVE_INFINITY**, **POSITIVE_INFINITY**, **NaN** (Not a Number)

Beispiel: Quadratische Gleichung

$ax^2 + bx + c = 0$

Allgemeine Lösung: $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Programm 7.11: Lösung einer quadratischen Gleichung (*QuadrGl.java*)

```
// a*x^2 + b*x + c = 0
import IOulm.*;

public class QuadrGl{

  public static void main(String[] args) {
     double a , b, c, d, x1, x2;
     double eps = 1.0E-6;
     Write.String("a (x^2) = ");
     if ( Urc.readReal() ) a = Urc.getReal(); else return ;
     Write.String("b (x^1) = ");
     if ( Urc.readReal() ) b = Urc.getReal(); else return ;
     Write.String("c (x^0) = ");
     if ( Urc.readReal() ) c = Urc.getReal(); else return ;
     Write.Real(a); Write.String("x^2 + ");
     Write.Real(b); Write.String("x + ");
     Write.Real(c); Write.Line(" = 0");
     if ( (Math.abs(a)<eps) && (Math.abs(b)<eps) && (Math.abs(c)<eps) ) {
         Write.Line("Unendlich viele Loesungen!"); return;
     }
     if ( (Math.abs(a)<eps) && (Math.abs(b)<eps) && (Math.abs(c)>=eps) ) {
         Write.Line("Keine Loesung!"); return;
     }
     if ( (Math.abs(a)<eps) && (Math.abs(b)>=eps) && (Math.abs(c)<eps) ) {
        Write.String("Eine Loesung: 0.0"); Write.Ln();
        return;
     }
     if ( (Math.abs(a)<eps) && (Math.abs(b)>=eps) && (Math.abs(c)>=eps) ) {
        Write.String("Eine Loesung: "); Write.Real(- c / b); Write.Ln();
        return;
     }
     d = b*b - 4*a*c;
     if (Math.abs(d) < eps) {
         Write.String("Zusammenfallende Loesungen: ");
         Write.Real( -b / (2 * a)); Write.Ln(); return;
     }
     if ( d < 0.0 ) {
         Write.Line("Keine Loesung!"); return;
     }
     d = Math.sqrt(d);
     x1 = ( -b + d )/(2*a);
     x2 = ( -b - d )/(2*a);
     Write.String("Erste Loesung: "); Write.Real(x1); Write.Ln();
     Write.String("Zweite Loesung: "); Write.Real(x2); Write.Ln();
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac QuadrGl.java
spatz$ java QuadrGl
a (x^2) = 4
b (x^1) = 0
c (x^0) = 4
4.0x^2 + 0.0x + 4.0 = 0
Keine Loesung!
spatz$ java QuadrGl
a (x^2) = 4
b (x^1) = 0
c (x^0) = -4
4.0x^2 + 0.0x + -4.0 = 0
Erste Loesung: 1.0
Zweite Loesung: -1.0
spatz$ java QuadrGl
a (x^2) = 1
b (x^1) = 5e-1
c (x^0) = -3
1.0x^2 + 0.5x + -3.0 = 0
Erste Loesung: 1.5
Zweite Loesung: -2.0
spatz$
```

7.3.6 Typkonvertierungen

- Mit Ausnahme von *boolean* können alle primitiven Typen ineinander konvertiert werden
- Vergrößernde (verbreiternde) Konvertierung (*widening conversion*): Konvertierung in einen Typ mit einem größeren Wertebereich
- Verkleinernde (verengende) Konvertierung (*narrowing conversion*): Konvertierung in einen Typ, dessen Wertebereich nicht größer ist (nicht immer sicher)

Programm 7.12 (S. 127) zeigt einige Beispiele.

Programm 7.12: Typkonvertierungen (*Casts.java*)

```java
// Typ-Konvertierungen
import IOulm.*;

public class Casts {

  public static void main(String [] args) {
    double x = 1234.56; int i;
    Write.String("x = "); Write.Real(x); Write.Ln();
    Write.String("(int) x = "); Write.Int( (int) x ); Write.Ln();
    i = (int) Math.round(x); // nach den uebl. Rundungsregeln
    Write.String("(int) Math.round(x) = "); Write.Int(i); Write.Ln();
    i = (int) Math.ceil(x);  // naechst groessere Integer
    Write.String("(int) Math.ceil(x) = "); Write.Int(i); Write.Ln();
    i = (int) Math.floor(x); // naechst kleinere Integer
    Write.String("(int) Math.floor(x) = "); Write.Int(i); Write.Ln();
    char c = '\uffff';       // ein Unicode (2 Byte mit 1 besetzt
    Write.String(" c = "); Write.Char(c); Write.Ln();
    Write.String(" c + 0 = "); Write.Int(c + 0); Write.Ln();
    short k = (short) c;     // jetzt der Wert -1
    Write.String("(short) c = "); Write.Short(k); Write.Ln();
  }
}
```

Die Ausgabe von Programm 7.12 (S. 127):

```
spatz$ javac Casts.java
spatz$ java Casts
x = 1234.56
(int) x = 1234
(int) Math.round(x) = 1235
(int) Math.ceil(x) = 1235
(int) Math.floor(x) = 1234
 c = ï££
 c + 0 = 65535
(short) c = -1
spatz$
```

Typ-Konvertierung:

Von	nach							
	boolean	byte	short	char	int	long	float	double
boolean	–	N	N	N	N	N	N	N
byte	N	–	I	C	I	I	I	I
short	N	C	–	C	I	I	I	I
char	N	C	C	–	I	I	I	I
int	N	C	C	C	–	I	I*	I
long	N	C	C	C	C	–	I*	I*
float	N	C	C	C	C	C	–	I
double	N	C	C	C	C	C	C	–

N	Konvertierung nicht möglich
C	Expliziter Cast für verkleinernde Konvertierung
I	Implizite erweiternde Konvertierung
I*	wie I, aber Stellen können verloren gehen

Programm 7.13 (S. 128) zeigt einige Beispiele für automatische Konvertierung mit Genauigkeitsverlust.

Programm 7.13: Typkonvertierungen mit Verlust (*Conversions.java*)

```java
// Typ-Konvertierungen
import IOulm.*;

public class Conversions {

  public static void main(String[] args) {
     int i = Integer.MAX_VALUE;
     long j = Long.MAX_VALUE;
     float x = i;
     float y = j;
     double z = j;
     Write.String("i = "); Write.Int(i);
     Write.String(" | x = "); Write.Float(x); Write.Ln();

     Write.String("j = "); Write.Long(j);
     Write.String(" | y = "); Write.Float(y);
     Write.String(" | z = "); Write.Real(z); Write.Ln();
  }
}
```

Die Ausgabe von Programm 7.13 (S. 128):

```
spatz$ javac Conversions.java
spatz$ java Conversions
i = 2147483647 | x = 2.14748365E9
j = 9223372036854775807 | y = 9.223372E18 | z = 9.223372036854776E18
spatz$
```

7.4 Die Klasse String

String-Literale sind beliebige, in Doppelapostrophen gefasste Zeichenfolgen, die beliebige Escape-Sequenzen enthalten können. String ist **kein** Typ der Sprache – es handelt sich um eine Klasse (dazu aber später mehr)!

Programm 7.14 (S. 129 zeigt ein einfaches Beispiel. Der Operator + bedeutet bei Strings die **Konkatenation**.

Programm 7.14: Klasse String (*MyStrings.java*)

```java
import IOulm.*;

public class MyStrings{

  public static void main(String[] args) {

        int i = 3; double p = 3.14;
        String s1, s2 = "Anton";
        s1 = s2 + " Huber";
        Write.Line(s1);
        s1 = s1 + ": " + i + " / " + p;
        Write.Line(s1);
        Write.Line("i = " + i);
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac MyStrings.java
spatz$ java MyStrings
Anton Huber
Anton Huber: 3 / 3.14
i = 3
spatz$
```

7.5 Operatoren

Die folgende Tabelle enthält eine ’Ubersicht über **alle** Operatoren – wir werden zunächst nur auf einige eingehen!

<table>
<tr><th>Ass.</th><th>Operator</th><th>Operand(en)</th><th>Operation</th></tr>
<tr><td>links</td><td>.
[]
(args)
++, --</td><td>Objekt, Member
Array, Integer
Methode, Argumentliste
Variable</td><td>Zugriff auf Objekt-Member
Zugriff auf Array-Element
Methodenaufruf
Post-Inkrement, Post-Dekrement</td></tr>
<tr><td>rechts</td><td>++, --
+, -
~
!</td><td>Variable
Zahl
integer
boolean</td><td>Prä-Inkrement, Prä-Dekrement
unäres Plus / Minus
Bitweises Komplement
Negation</td></tr>
<tr><td>rechts</td><td>new
(Typ)</td><td>Klasse, Argumentliste
Typ, beliebig</td><td>Objekterzeugung
Cast (Typkonvertierung)</td></tr>
<tr><td>links</td><td>*, /, %</td><td>Zahl, Zahl</td><td>Multiplikation, Division, Rest</td></tr>
<tr><td>links</td><td>+, -
+</td><td>Zahl, Zahl
String, beliebig</td><td>Addition, Subtraktion
String-Konkatenation</td></tr>
<tr><td>links</td><td><<
>>

>>></td><td>Integer, Integer
Integer, Integer

Integer, Integer</td><td>Shift nach links
Shift nach rechts mit Vorzeichenerweiterung
Shift nach rechts mit Null-Erweiterung</td></tr>
<tr><td>links</td><td><, <=
>, >=
instanceof</td><td>Zahl, Zahl
Zahl, Zahl
Referenz, Typ</td><td>Größenvergleiche
Größenvergleiche
Typvergleich</td></tr>
<tr><td>links</td><td>==
!=
==
!=</td><td>primitiv, primitiv
primitiv, primitiv
Referenz, Referenz
Referenz, Referenz</td><td>Identität
Ungleichheit
Referenzen auf dasselbe Objekt
Referenzen auf verschiedene Objekte</td></tr>
<tr><td>links</td><td>&
&</td><td>Integer, Integer
boolean, boolean</td><td>Bitweises UND
Logisches UND</td></tr>
<tr><td>links</td><td>^
^</td><td>Integer, Integer
boolean, boolean</td><td>Bitweises exkl. ODER (XOR)
Logisches exkl. ODER (XOR)</td></tr>
<tr><td>links</td><td>|
|</td><td>Integer, Integer
boolean, boolean</td><td>Bitweises ODER
Logisches ODER</td></tr>
<tr><td>links</td><td>&&</td><td>boolean, boolean</td><td>Konditionales UND</td></tr>
<tr><td>links</td><td>||</td><td>boolean, boolean</td><td>Konditionales ODER</td></tr>
<tr><td>rechts</td><td>?:</td><td>boolean, beliebig, beliebig</td><td>bedingter (ternärer) Operator</td></tr>
<tr><td>rechts</td><td>=
*=, /=, %=,
+=, -=, <<=,
>>=, >>>=,
&=, ^=, |=</td><td>Variable, beliebig
Variable, beliebig</td><td>Zuweisung
Zuweisung mit Operation</td></tr>
</table>

Anm.:

- Zahl: jeder primitive Typ außer *boolean* (also Integer, Gleitkommazahl oder Zeichenwert)
- Integer: alle Zahlentypen – bei Vektorzugriffen ist *long* **nicht** erlaubt!
- Referenz: ein Objekt (Instanz einer Klasse) oder Array
- Variable: etwas, dem ein Wert zugewiesen werden kann

Präzedenz (Vorrang):

- von oben nach unten absteigend – d.h. `*` (Zeile 4) kommt vor + (Zeile 5)
- Klammern zur Veränderung des Vorrangs (im Zweifelsfall immer anzuraten)

Assoziativität: (erste Spalte "Ass.")

regelt, wie mehrere Operatoren bei gleichem Vorrang in einem Ausdruck ausgewertet werden

Beispiel: `a = b = c = 8` ist äquivalent mit `a = ( b = (c = 8) )`, d.h.

- es wird zunächst `c = 8` bewertet: c wird 8 zugewiesen, der Wert der Zuweisung ist der zugewiesene Wert, also wieder 8
- dann wird b der Wert des Ausdrucks `c = 8` zugewiesen, also 8 – der Wert dieser Zuweisung ist wieder 8
- schließlich wird a der Wert des Ausdrucks `b = ( c = 8)` zugewiesen, also ebenfalls 8 – der Wert dieses Ausdrucks wäre wiederum 8

Rückgabe- / Ergebnistyp:

- Die arithmetischen Operatoren geben
  - *double* zurück, wenn wenigstens einer der Operanden *double* war,
  - *float* zurück, wenn wenigstens einer der Operanden *float* war,
  - *long* zurück, wenn wenigstens einer der Operanden *long* war,
  - *int* zurück, wenn keiner der oberen Fälle zutrifft, auch wenn die Operanden kleiner als *int* waren.
- Vergleichsoperatoren geben *boolean* zurück
- Zuweisungsoperatoren geben den Wert zurück, der zugewiesen wurde und von einem Typ ist, der mit dem der Variablen auf der linken Seite kompatibel ist
- Der bedingte Operator gibt den Wert des zweiten oder dritten Arguments zurück (beide müssen von einem Typ sein, der mit dem Typ der linken Seiten kompatibel ist – siehe als Beispiel Programm 7.15, S. 132

Programm 7.15: Bedingter Operator (*BedOperator.java*)

```
// Bedingter Operator
import IOulm.*;

public class BedOperator{

  public static void main(String[] args) {

     double x = (3 > 2)? 1.2 : 'A';
     double y = (3 <= 2)? 1.2 : 'A';
     Write.String("x = "); Write.Real(x); Write.Ln();
     Write.String("y = "); Write.Real(y); Write.Ln();
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac BedOperator.java
spatz$ java BedOperator
x = 1.2
y = 65.0
spatz$
```

Auswertungsreihenfolge

- Reihenfolge der Operandenbewertung ergibt sich durch Klammerung, Präzedenz und Assoziativität
- zuerst werden die Operanden ausgewertet, von links nach rechts
- es werden nicht notwendig alle ausgewertet: Kurzschlussbewertung!

Arithmetische Operatoren

- **Addition** (+):

  Ist einer der Operatoren ein String, so wird daraus die Konkatenation (Siehe Programm 7.14, S. 129).

- **Division** (/):

  Sind beide Operanden ganzzahlig, so erfolgt die ganzzahlige Division!

- **Modulo** (%):

  Der Rest, der entsteht, wenn der zweite Operator so oft wie möglich vom ersten subtrahiert wird – das Vorzeichen ist das des ersten Operanden (Programm 7.16, S. 133).

Programm 7.16: Modulo (*Modulo.java*)

```java
import IOulm.*;

public class Modulo{

  public static void main(String[] args) {

      Write.Line("11%3          = " + 11%3);
      Write.Line("11%3.5        = " + 11%3.5);
      Write.Line("-11%3         = " + (-11%3));
      Write.Line("11%-3         = " + (11%-3));
      Write.Line("-11%-3        = " + (-11%-3));
      Write.Line("11.0%0.0      = " + 11.0%0.0);
      Write.Line("(1.0/0.0)%0.0 = " + (1.0/0.0)%0.0);
      Write.Line("11%0          = " + 11%0); // Fehler
  }
}
```

Ausführung liefert:

```
spatz$ javac Modulo.java
spatz$ java Modulo
11%3          = 2
11%3.5        = 0.5
-11%3         = -2
11%-3         = 2
-11%-3        = -2
11.0%0.0      = NaN
(1.0/0.0)%0.0 = NaN
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Modulo.main(Modulo.java:14)
spatz$
```

- Vergleichsoperatoren (==, `!=`):
  - Vorsicht bei Referenztypen: es werden nicht die referenzierten Objekte verglichen (dazu später mehr)
  - Bei Zahlen erfolgt eine entsprechenden Konvertierung (der schmalere Typ in den breiteren Typ)
- Logische Operatoren (siehe Programm 7.2, S. 110)
  - Konditionale Operatoren (`&&, ||, !`): bewertet wird von links nach rechts; sobald Resultat feststeht, werden die weiteren Operanden nicht mehr ausgewertet (Kurzschlussbewertung)
  - Logische Operatoren im Logik-Sinn (`&, |, ^`): bewertet werden **alle** Operanden
- Der Operator **instanceof**

  erwartet als linken Operanden ein Objekt oder ein Array und als rechten Operanden den Namen eines Referenztyps (später mehr dazu)

7.6 Anweisungen

7.6.1 Übersicht

Anweisung	**Zweck**	**Syntax**

Ausdruck	Seiteneffekte	`var = expression;`
		`i++;`
		`meth();`
		new `typ();`

Zusammengesetzte Anweisung	Zusammenfassung zu einer Anweisung / Gruppierung	{ *anweisung {*; *anweisung}* }
leere Anweisung		;
benannte Anweisung	Anweisung benennen	*label* **:** *anweisung*
Deklaration	Variable deklarieren	*[***final***] typ name [= wert] {*,*name = wert}*;
if	Verzweigung	**if (***ausdruck* **)** *anweisung* *[* **else** *anweisung]*
switch	Mehrfach-Verzweigung	**switch (** *ausdr* **) {** *{* **case :** *anweisung }* *[* **default :** *anweisungen]*
while	Wiederholung	**while (***ausdruck* **)** *anweisung*
do	Wiederholung	**do** *anweisung* **while (***ausdruck***)**
for	Wiederholung	**for(** *ausdruck* **;** *bedingung* **;** *ausdruck* **)** *anweisung*
break	Block verlassen	**break** *[label]*
continue	Schleife von neuem beginnen	**continue** *[label]*
return	Methode verlassen	**return** *[ausdruck]*
synchronized	kritischen Abschnitt schützen	**synchronized (** *ausdruck* **) {** *anweisung* **}**
throw	Ausnahme auslösen	**throw** *ausdruck*
try	Ausnahme verarbeiten	**try {** *anweisungen* **}** *{* **catch (** *typ name* **) {** *anweisungen* **}** *}* *[* **finally {** *anweisungen* **}** *]*
assert	Invariante prüfen	**assert** *invariante [* **:** *fehler]*

7.6.2 Wiederholungsanweisungen

- **while (** `bedingung` **)** `anweisung`

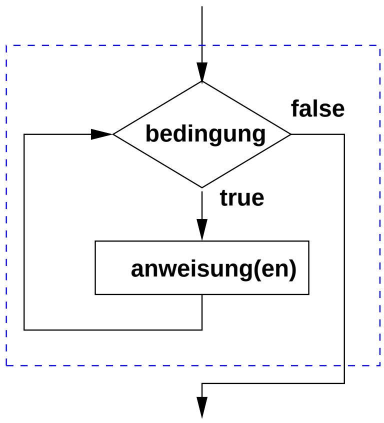


Abbildung 7.9: while-Schleife

- **do** `anweisung` **while (** `bedingung` **)**

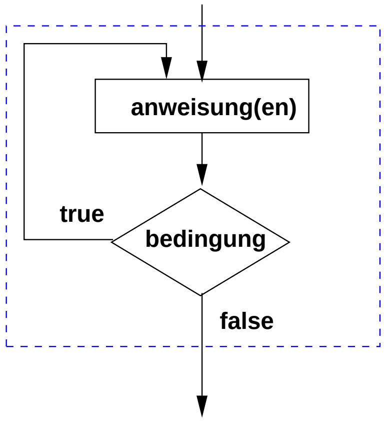


Abbildung 7.10: do-while-Schleife

Es sollte in beiden Fällen sichergestellt sein, dass die (Wiederholungs-) Bedingung irgendwann einmal wahr wird – außer man will wirklich eine Endlosschleife!

Programm 7.17 (S. 137) zeigt ein Beispiel, dessen Ausführung liefert:

```
spatz$ javac Schleife.java
spatz$ java Schleife
0 1 4 9 16 25 36 49 64 81
nach der Schleife: i = 10
100 81 64 49 36 25 16 9 4 1
nach der Schleife: i = 0
spatz$
```

Programm 7.17: while und do–while (*Schleife.java*)

```
import IOulm.*;
public class Schleife {
  public static void main(String [] args) {
      int i = 0; int N = 10;

      while ( i < N ) {
          Write.Int(i*i); Write.Char(' ');
          i = i + 1;
      }
      Write.Line("\nnach der Schleife: i = " + i);

      do {
          Write.Int(i*i); Write.Char(' ');
          i = i - 1;
      } while ( i > 0 );
      Write.Line("\nnach der Schleife: i = " + i);
  }
}
```

- **for (** `initialisierung;iterationsbedingung;variablenveraenderung`**)** `anweisung`

  Programm 7.18 (S. 137) zeigt ein einfaches Beispiel, dessen Ausführung liefert:

```
spatz$ javac For.java
spatz$ java For
0 1 4 9 16 25 36 49 64 81
nach der Schleife: i = 10
100 81 64 49 36 25 16 9 4 1
nach der Schleife: i = 0
spatz$
```

Programm 7.18: for-Schleife (*For.java*)

```
import IOulm.*;
public class For{
  public static void main(String [] args) {
      int i; int N = 10;

      for (i = 0; i < N; i = i + 1) {
          Write.Int(i*i); Write.Char(' ');
      }
      Write.Line("\nnach der Schleife: i = " + i);

      for ( i = N; i > 0; i = i -1) {
          Write.Int(i*i); Write.Char(' ');
      };
      Write.Line("\nnach der Schleife: i = " + i);
  }
}
```

Mehr dazu auch in Programm 7.21, S. 140!

7.6.3 Verzweigungen

Die **if**-Anweisung sollte eigentlich selbsterklärend sein, sie enthält allerdings eine kleine Falle – siehe Programm 7.19, S. 138.

Programm 7.19: if – else (*If.java*)

```
import IOulm.*;
public class If {
  public static void main(String [] args) {
    boolean a, b; int i = 0;
    a = ( i == 0);
    b = false;
    if (a)
        if (b)
            Write.Line(" ?????????? ");
     else
        Write.Line("   !!!!!!!!!    ");
// ========================================
    if (a)
        if (b)
            Write.Line(" ?????????? ");
        else
            Write.Line("   !!!!!!!!!    ");
  }
}
```

Das **else** gehört immer zum nächsten **if** (*dangling else*)!

Die **switch**-Anweisung (Mehrfachverzweigung)

```
switch (ausdruck) {
  case constant-1:
    anweisung;
  case constant-2:
    anweisung;
  ...
  default:
    anweisung;
}
```

Zunächst wird der Ausdruck `ausdruck`, der vom Typ `byte`, `short`, `char` oder `int` sein muss, ausgewertet. In Abhängigkeit vom Ergebnis wird dann die Sprungmarke (`case`) angesprungen, deren Konstante mit dem Ergebnis des Ausdrucks übereinstimmt. Die Konstante und der Ausdruck müssen dabei zuweisungskompatibel sein.

Das optionale **default**-Label wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird. Ist kein `default`-Label vorhanden und wird auch keine passende Sprungmarke gefunden, so wird keine der Anweisungen innerhalb der switch-Anweisung ausgeführt. Jede Konstante eines `case`-Labels darf nur einmal auftauchen. Das `default`-Label darf maximal einmal verwendet werden.

Achtung:

Nachdem ein `case`- oder `default`-Label angesprungen wurde, werden **alle** dahinterstehenden Anweisungen ausgeführt. Im Gegensatz zu Sprachen wie PASCAL erfolgt auch dann keine Unterbrechung, wenn das nächste Label erreicht wird. Wenn dies erwünscht ist (ist wohl meist der Fall), so muss der Kontrollfluss mit Hilfe einer **break**-Anweisung unterbrochen werden. Jedes `break` innerhalb einer switch-Anweisung führt dazu, dass zum Ende der `switch`-Anweisung verzweigt wird.

Programm 7.20, S. 139, zeigt ein typisches Beispiel.

Programm 7.20: switch-Anweisung (*Switch.java*)

```java
import IOulm.*;
public class Switch{
  public static void main(String[] args) {
      char ch;
      while ( Urc.readChar() ) {
          ch = Urc.getChar();
          if ( ((ch >= 'a') && (ch <= 'z')) ||
               ((ch >= 'A') && (ch <= 'Z')) )
              Write.Line(" letter ");
          else
              switch (ch) {
                  case '!':
                    Write.Line("exclamation mark"); break;
                  case '?':
                    Write.Line("question mark"); break;
                  case ',':
                    Write.Line("comma"); break;
                  case '.':
                    Write.Line("period or dot or point"); break;
                  case ':':
                    Write.Line("colon"); break;
                  case ';':
                    Write.Line("semicolon"); break;
                  case '0':
                  case '1':
                  case '2':
                  case '3':
                  case '4':
                  case '5':
                  case '6':
                  case '7':
                  case '8':
                  case '9':
                    Write.Line(" digit "); break;
                  case '\n':
                    break;
                  default:
                    Write.Line("Don't know :-(");
              }
          }
  }
}
```

7.6.4 Benannte Anweisungen

Benannte Anweisungen werden eher selten benötigt, sollen aber der V ollständigkeit halber an einem einfachen Beispiel (nur bedingt sinnvoll) zusammen mit der Anweisung **break** in Programm 7.21, S. 140, demonstriert werden.

Programm 7.21: Benannte Anweisung, break (*Label.java*)

```java
// benannte Anweisungen
import IOulm.*;

public class Label {

  public static void main(String[] args) {
     int OG = 5, i = 0, j = 0;
     rows: for (i=0; i < OG; i = i + 1) {
           cols: for (j=OG; j >= 0; j = j - 1) {
                      if ( i * j > 2 * OG ) break rows;
                      else
                            Write.Line("i = " + i + " / j = " + j);
                 }
           Write.Line("---------------------------");
           }
  }
}
```

Übersetzung und Ausführung von Programm 7.21, S. 140:

```
spatz$ javac Label.java
spatz$ java Label
i = 0 / j = 5
i = 0 / j = 4
i = 0 / j = 3
i = 0 / j = 2
i = 0 / j = 1
i = 0 / j = 0
--------------------------
i = 1 / j = 5
i = 1 / j = 4
i = 1 / j = 3
i = 1 / j = 2
i = 1 / j = 1
i = 1 / j = 0
--------------------------
i = 2 / j = 5
i = 2 / j = 4
i = 2 / j = 3
i = 2 / j = 2
i = 2 / j = 1
i = 2 / j = 0
--------------------------
spatz$
```

Anmerkungen zu Programm 7.21, S. 140:

1. Die Erhöhung der Variablen `i` in Zeile 8 um 1 wie auch die Erniedrigung der Variablen `j` in Zeile 9 um 1 wird in Java typischerweise mit dem Operator **++** bzw. **- -** formuliert.

   Die Stellung des Operators **vor** bzw. **nach** der Variablen ändert die Semantik – dies soll Programm 7.22, S. 141 demonstrieren.

   Programm 7.22: Die Operatoren ++ und - - (*PlusPlus.java*)

   ```java
   // benannte Anweisungen
   import IOulm.*;

   public class PlusPlus {

     public static void main(String [] args) {
           int i = 1, j = 1, m,n;
           m = i++; n = ++j;
           Write.Line("m = " + m + " / n = " + n);
           Write.Line("m++ = " + m++ + " / ++n = " + ++n);
           Write.Line("m = " + m + " / n = " + n);

     }
   }
   ```

   Übersetzung und Ausführung von 7.22, S. 141:

   ```
   spatz$ javac PlusPlus.java
   spatz$ java PlusPlus
   m = 1 / n = 2
   m++ = 1 / ++n = 3
   m = 2 / n = 3
   spatz$
   ```

2. Die beiden Variablen `i` und `j` werden offenkundig nur innerhalb der jeweiligen Schleifen benutzt (`i` in Zeilen 8-15, `j` in Zeilen 9-13). In diesem Fall können wir sie auch lokal definieren und damit ihre Sichtbarkeit wie ihre Lebensdauer auf die jeweilige Schleife beschränken.

Programm 7.23 (S. 142) fasst diese beiden Punkte zusammen.

Programm 7.23: Programm 7.21 von S. 140 modifiziert (*Label1.java*)

```java
import IOulm.*;
public class Label1 {
  public static void main(String [] args) {
      int OG = 5;
      rows: for (int i=0; i < OG; i++) {
            cols : for (int j=OG; j >= 0; j--) {
                  if ( i * j > 2 * OG ) break rows;
                  else
                        Write.Line("i = " + i + " / j = " + j );
            }
            Write.Line("-------------------------");
            }
            i++; // ausserhalb von Anweisung rows
  }
}
```

Übersetzung von Programm 7.23, S. 142:

```
spatz$ javac Label1.java
Label1.java:13: cannot find symbol
symbol   : variable i
location: class Label1
           i++; //ausserhalb von Anweisung rows
           ^
1 error
spatz$
```

7.6.5 Finale Initialisierung: final

In Programm 7.21 (S. 140) hat die Variable *OG* den Charakter einer Konstanten – dies kann wie in Programm 7.24 (S. 143) zum Ausdruck gebracht werden und auch sichergestellt werden.

Programm 7.24: Programm 7.21 von S. 140 modifiziert (*Label2.java*)

```java
import IOulm.*;
public class Label2 {
  public static void main(String[] args) {
      final int OG = 5;
      rows: for (int i=0; i < OG; i++) {
            cols : for (int j=OG; j >= 0; j--) {
                        if ( i * j > 2 * OG ) break rows;
                        else
                              Write.Line("i = " + i + " / j = " + j);
                  }
              Write.Line("---------------------------");
              }
              OG++;
  }
}
```

Übersetzung von Programm 7.24, S. 143:

```
spatz$ javac Label2.java
Label2.java:13: cannot assign a value to final variable OG
           OG++;
           ^
1 error
spatz$
```

7.7 Array

7.7.1 Konzept

Das **Array** stellt in Java einen wichtigen Aspekt in der Verwaltung typgleicher Variablen und Objekte dar. Es handelt sich also um ein Feld oder einen Container, das bzw. der als Variable dazu in der Lage ist, mehrere Objekte vom gleichen Typ aufzunehmen und zu verwalten. Die darin enthaltenen Objekte sind speziellen Eigenschaften und Zugriffsmöglichkeiten unterworfen.

Arrays sind genau betrachtet keine eigenen Datentypen, sondern werden in Java durch eine interne Klasse repräsentiert. Abb. 7.11 (S. 144) soll schematisch das Speichermanegement und die Anordnung der Containerelemente im Speicher zeigen. Diese werden möglichst sequentiell angeordnet, um einen möglichst schnellen Zugriff auf alle Elemente zu erhalten.

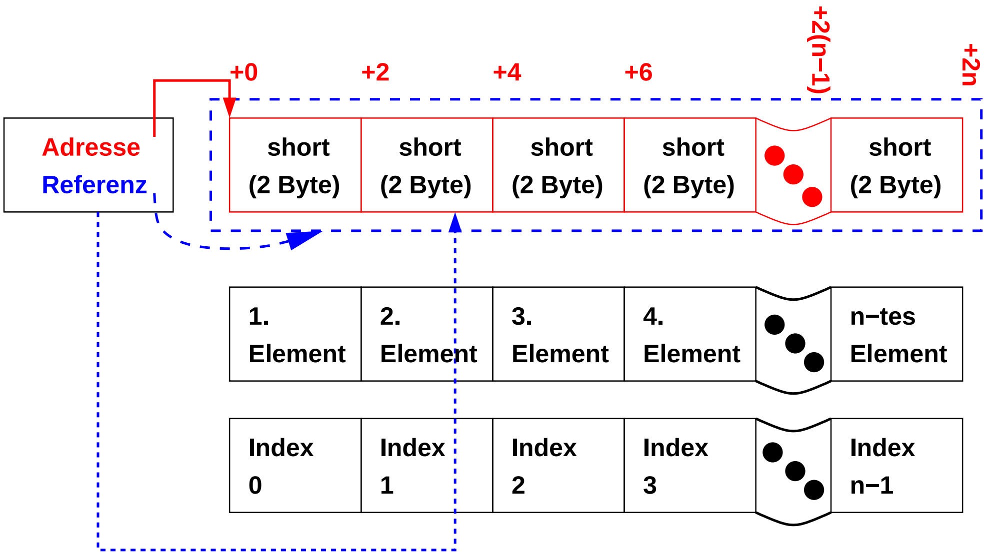


Abbildung 7.11: Speicherung / Darstellung eines Array

Wie bereits erwähnt werden Arrays in Java durch eine spezielle Klasse repräsentiert: so kann ein Array spezielle Methoden und Operationen zur Verfügung stellen, um die enthaltenen Daten möglichst effektiv verwalten und manipulieren zu können. Zudem ist das Abfragen von Statusdaten eines Array (z.B. die Anzahl seiner Elemente) möglich.

Daraus resultieren aber auch einige Besonderheiten beim Anlegen eines Array im Speicher und beim Kopieren (dazu später mehr).

7.7.2 Deklaration

Bei der Deklaration eines Array wird der eigentliche Container erzeugt. Das bedeutet, dass das Array zunächst in Größe und Typ genau bestimmt wird. Möglich ist die Verwendung aller elementaren und benutzerdefinierten Datentypen.

Die Deklaration wird mit dem Datentyp eingeleitet, gefolgt vom Namen des Arrays. Wichtig ist die Kennzeichnung des Arrays mit den eckigen Klammeroperatoren ([]), welche eine Array-Variable ausweisen. Diese können nach dem Typ oder nach dem Namen erscheinen. Zum Abschluss wird das Array noch mit dem **new**-Operator im Speicher erzeugt (Speicher wird alloziert) und mit dem **Konstruktor** der Elementklasse instanziiert. Zu allen bisher behandelten Datentypen gibt es ja auch jeweils eine passende Klasse (**Wrapper**), die einen solchen Konstruktor (Methode ohne Rückgabewert, gleicher Name wie die Klasse) besitzt.

Syntax:

`type`**[]** `Name` **= new** `type`**[**`Anzahl`**]**

`type Name` **[] = new** `type`**[**`Anzahl`**]**

Programm 7.25 (S. 145) zeigt ein erstes Beispiel.

Programm 7.25: Array Deklaration (*Array1.java*)

```java
1  import IOulm.*;
2
3  public class Array1{
4
5    public static void main(String [] args) {
6          final int N = 9;
7          int [] v = new int[N]; // <-- so
8          int w[] =   {1,2,3,4,5,6,7,8,9};     // <-- oder so
9          for (int i = 0; i < N; i++)
10             v[i] = (i+1) * w[i];
11
12         for(int i = N-1; i>= 0; i--)
13             Write.Line(" " + (i+1) + " zum Quadrat ist " + v[i]);
14   }
15 }
```

Anmerkungen zu Programm 7.25 (S. 145):

Zeile 7: Es wird die Variable `v` als Array von Elementen des Typs `int` deklariert; dieser wird eine mit `new` erzeugte Referenz auf ein Array mit `N` Elementen vom Typ `int` zugewiesen; die allozierte Speicherfläche dieses Arrays ist mit `0` initialisiert

Zeile 8: Es wird die Variable `w` als Array von Elementen des Typs `int` deklariert und mit den Werten auf der rechten Seite initialisiert – die Dimension ergibt sich über deren Anzahl. Die Stellung der eckigen Klammern nach dem Variablennamen ist zwar zulässig, aber eher nicht zu empfehlen, da die Typ-Information links vom Variablennamen steht (hier `int`) – `[]` gehört zur Typ-Information!

Zeile 10: Der Zugriff auf ein Array-Element erfolgt über den Index: `v[i]`

Der Referenzwert des Arrays (Wert von `v`) ist ohne Bedeutung, ausgenommen das spezielle Literal `null`, das die “Abwesenheit” eines Arrays ausdrückt!

Mit dem Array verbunden ist die Eigenschaft `length` , die die Anzahl der Elemente angibt – Programm 7.26 (S. 146) zeigt eine Anwendung.

Programm 7.26: Array-Länge (*Array2.java*)

```java
import IOulm.*;

public class Array2{

  public static void main(String[] args) {
        int w[] =   {1,2,3,4,5,6,7,8,9};
        for (int i = 0; i < w.length; i++)
            w[i] = (i+1) * w[i];

        for(int i = w.length - 1; i>= 0; i--)
             Write.Line(" " + (i+1) + " zum Quadrat ist " + w[i]);
  }
}
```

Arrays können beliebige Typen aufnehmen: Programm 7.27, S. 146, zeigt ein Array von Strings.

Programm 7.27: Array mit Strings (*ArrayOfStrings.java*)

```java
import IOulm.*;

public class ArrayOfStrings {
    public static void main(String[] args) {
        String[] anArray = { "1. String", "2. String", "3. String" };

        for (int i = 0; i < anArray.length; i++) {
            Write.Line(anArray[i]);
        }
    }
}
```

Beispiel: "beliebig" große Zahl mit einer einstelligen Zahl multiplizieren

- (ganze) Zahl als Folge von Ziffern (Typ `char`) in ein Array `char [] zahl` einlesen und am Ende das Null-Byte anfügen
- da die Zahl bei der Multiplikation um eine Stelle größer werden kann, wird die Zahl "umgedreht": die Einerstelle kommt auf Indexposition 0, die Zehnerstelle auf 1, ...
- nach den üblichen Regeln multiplizieren

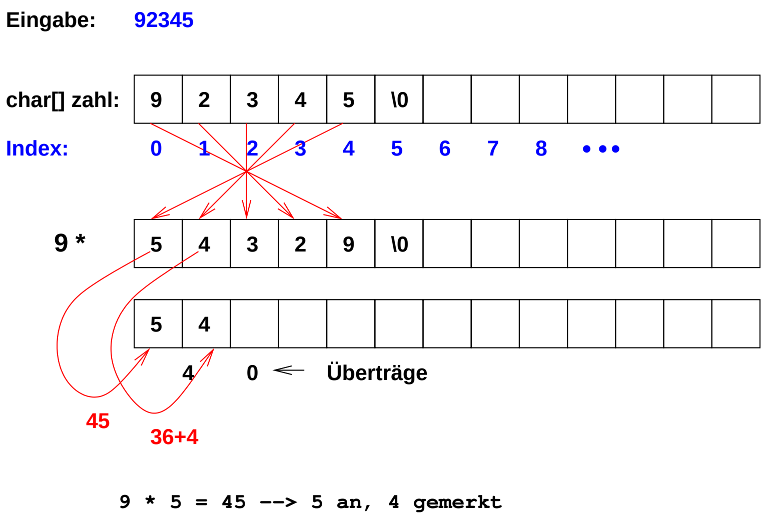


Abbildung 7.12: Multiplikation einer großen Zahl

Programm 7.28 (S. 147) zeigt eine Implementierung.

Programm 7.28: Multiplikation einer großen Zahl (*Multiply.java*)

```
import IOulm.*;
public class Multiply {
  public static void main(String [] args) {
        final int N = 100; int i = 0, ende = 0, p, an, gemerkt; char z;
        char [] zahl = new char[N];

        // Einlesen :
        while ( Urc.readChar () && ( i < N-1 ) ) {
           z = Urc.getChar ();
           if ( (z < '0') || (z > '9') )
                   break;
           zahl [i] = z ; i++;
        }
        zahl [i] = '\0'; ende = i-1;

        // Umdrehen:
        for ( i = 0; i <= ende / 2; i++) {
            z = zahl [i];  zahl [i] = zahl [ende-i];
            zahl [ende-i] = z;
        }
```

```

        // Multiplizieren
        z = '9'; gemerkt = 0;
        for(i=0; i<=ende; i++) {
            p = ( zahl[i] - '0' ) * ( z - '0') + gemerkt;
            an = p % 10; zahl[i] = (char) (an + '0');
            gemerkt = p / 10;
        }
        if (gemerkt != 0 ) {
            zahl[++i] = (char) (gemerkt + '0');
            ende = i;
            zahl[++i] = '\0';
        }

        // Ausgeben
        for (i = ende; i >= 0; i--)
            Write.Char(zahl[i]);
        Write.Ln();
    }
}
```

7.7.3 Mehrdimensionale Arrays

Die Elemente von Arrays können insbesondere auch Referenzen auf Arrays sein – dies führt zu Strukturen wie in Abb. 7.13 (S. 149) dargestellt.

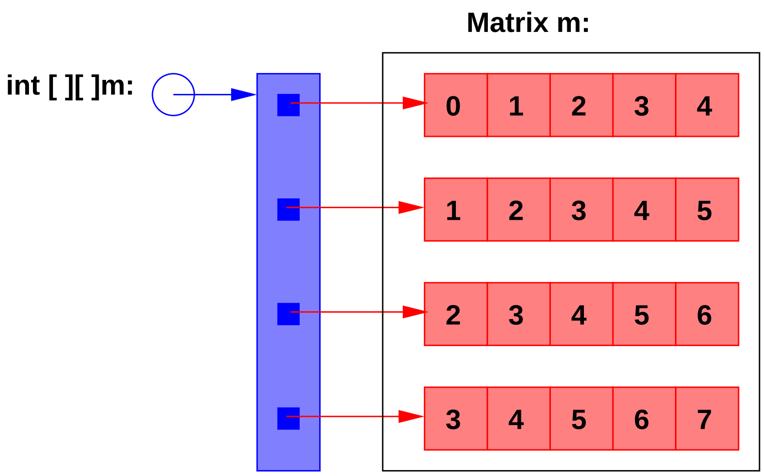


Abbildung 7.13: Zweidimensionale Matrix

Programm 7.29 (S. 149) zeigt die Deklaration, Initialisierung und Verwendung eines mehrdimensionalen Arrays.

Programm 7.29: Mehrdimensionales Array (*ArrayOfArrays0.java*)

```java
import IOulm.*;

public class ArrayOfArrays0 {
    public static void main(String [] args) {
        // 4x5-Matrix
        int [][] aMatrix = new int [4][5];

        //  Initialisierung
        for (int i = 0; i < aMatrix.length; i++) {
            for (int j = 0; j < aMatrix[i].length; j++) {
                aMatrix[i][j] = i + j;
            }
        }

        // Ausgabe
        for (int i = 0; i < aMatrix.length; i++) {
            for (int j = 0; j < aMatrix[i].length; j++) {
                Write.String(aMatrix[i][j] + " ");
            }
            Write.Ln();
        }
    }
}
```

Übersetzung und Ausführung von Programm 7.29 (S. 149):

```
spatz$ javac ArrayOfArrays0.java
spatz$ java ArrayOfArrays0
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
spatz$
```

Programm 7.30 (S. 150) zeigt eine Alternative zu Programm 7.29 (S. 149).

Programm 7.30: Mehrdimensionales Array (*ArrayOfArrays1.java*)

```java
import IOulm.*;

public class ArrayOfArrays1 {
    public static void main(String[] args) {
        int[][] aMatrix = new int[4][];

        // Initialisierung
        for (int i = 0; i < aMatrix.length; i++) {
            aMatrix[i] = new int[5];     // Erzeuge Zeilen-Array
            for (int j = 0; j < aMatrix[i].length; j++) {
                aMatrix[i][j] = i + j;
            }
        }

        // Ausgabe
        for (int i = 0; i < aMatrix.length; i++) {
            for (int j = 0; j < aMatrix[i].length; j++) {
                Write.String(aMatrix[i][j] + " ");
            }
            Write.Ln();
        }
    }
}
```

Die Dimensionen der Teilfelder müssen nicht identisch sein – dies zeigen Abb. 7.14 (S. 151) und Programm 7.31 (S. 151).

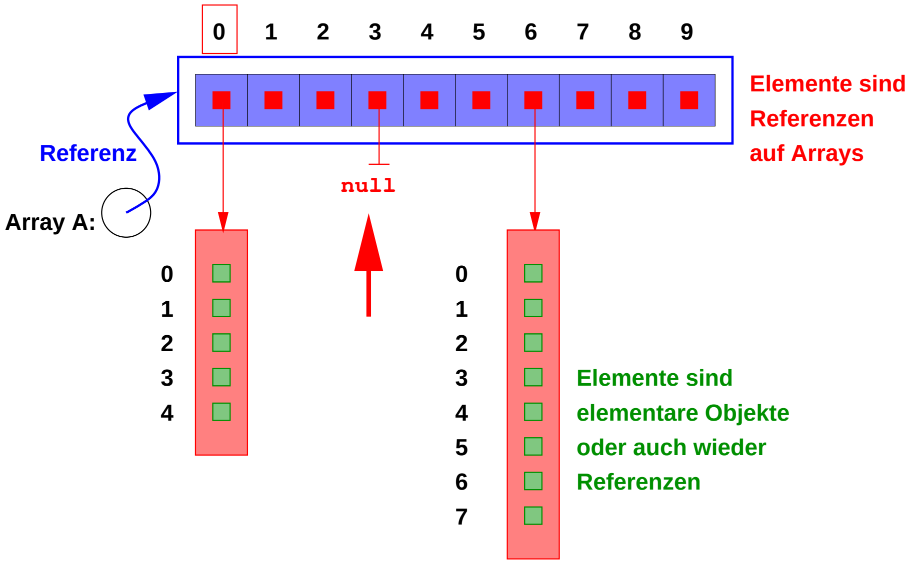


Abbildung 7.14: Mehrdimensionale Arrays

Programm 7.31: Mehrdimensionales Array mit unterschiedlichen Dimensionen (*ArrayOfArrays2.java*)

```
import IOulm.*;

public class ArrayOfArrays2 {
    public static void main(String[] args) {
        String[][] cartoons =
        { { "Flintstones", "Fred", "Wilma", "Pebbles", "Dino" },
            { "Rubbles", "Barney", "Betty", "Bam Bam" },
            { "Jetsons", "George", "Jane",
              "Elroy", "Judy", "Rosie", "Astro" },
            { "Scooby Doo Gang", "Scooby Doo",
              "Shaggy", "Velma", "Fred", "Daphne" }
        };
        for (int i = 0; i < cartoons.length; i++) {
            Write.String(cartoons[i][0] + ": ");
            for (int j = 1; j < cartoons[i].length; j++) {
                Write.String(cartoons[i][j] + " ");
            }
            Write.Ln();
        }
    }
}
```

Übersetzung und Ausführung von Programm 7.31 (S. 151):

```
spatz$ javac ArrayOfArrays2.java
spatz$ java ArrayOfArrays2
Flintstones: Fred Wilma Pebbles Dino
Rubbles: Barney Betty Bam Bam
Jetsons: George Jane Elroy Judy Rosie Astro
Scooby Doo Gang: Scooby Doo Shaggy Velma Fred Daphne
spatz$
```

7.7.4 Kopieren von Arrays

Bei der Wertzuweisung (Kopieren) einer Array-Variablen `w` an die Array-Variable `v` – Dimension und Elementtyp gleich – wird lediglich die Referenz kopiert, nicht das Array als Sammlung von Elementen (siehe Abb. 7.15 (S. 152)).

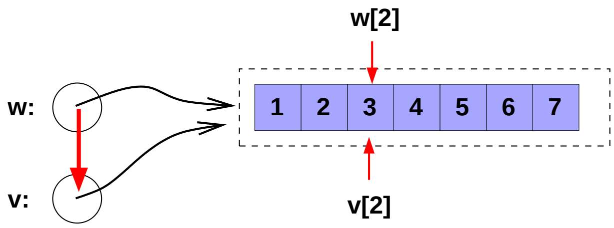


Abbildung 7.15: Kopieren von Arrays

Um eine "tiefe" Kopie zu erhalten, kann man die Methode **arraycopy()** verwenden (siehe Abb. 7.16 (S. 152). Die Methode befindet sich im System-Package von Java.

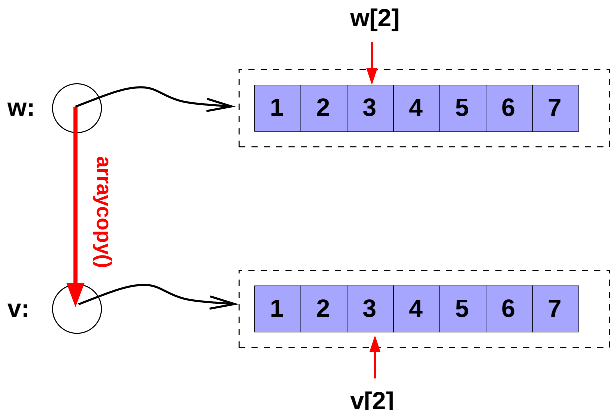


Abbildung 7.16: Kopieren von Arrays: arraycopy()

Die Methode `arraycopy()` hat 5 Parameter und keinen Rückgabewert:

- der erste Parameter gibt die Quelle für den Kopiervorgang an
- der zweite gibt die Startposition zum Kopieren in der Quelle an
- der dritte benennt das Ziel
- der vierte gibt die Startadresse im Zielbereich an
- der fünfte gibt die Länge des zu kopierenden Bereichs an

Programm 7.32 (S. 153) zeigt ein kleines Beispiel.

Programm 7.32: Kopieren von Arrays: arraycopy() (*ArrCopy1.java*)

```java
import IOulm.*;

public class ArrCopy1 {
  public static void main(String[] args) {
    char[] w = {'a','b','c','d','e','f','g'};
    char [] v = {'X','X','X','X','X','X','X'};

    System.arraycopy(w,1,v,2,4);

    Write.Line("i:  w[i]:  v[i]:");
    for(int i=0; i<v.length; i++)
      Write.Line("" + i + "    " + w[i] + "      " + v[i]);
  }
}
```

Übersetzung und Ausführung von Programm 7.32 (S. 153):

```
spatz$ javac ArrCopy1.java
spatz$ java ArrCopy1
i:  w[i]:  v[i]:
0    a      X
1    b      X
2    c      b
3    d      c
4    e      d
5    f      e
6    g      X
spatz$
```

Tiefes Kopieren bei mehrdimensionalen Arrays:

Programm 7.33 (S. 154) zeigt ein Beispiel mit gleichen Dimensionen.

Programm 7.33: Kopieren mehrdimensionaler Arrays (*ArrCopy2.java*)

```java
import IOulm.*;

public class ArrCopy2 {
  public static void main(String[] args) {
    char[][] w = { {'A','A','A','A','A','A'}, {'B','B','B','B','B','B'},
                   {'C','C','C','C','C','C'}, {'D','D','D','D','D','D'},
                   {'E','E','E','E','E','E'}
                 };
    char[][] v = { {'X','X','X','X','X','X'}, {'X','X','X','X','X','X'},
                   {'X','X','X','X','X','X'}, {'X','X','X','X','X','X'},
                   {'X','X','X','X','X','X'}
                 };

    System.arraycopy(v,1,w,2,2);

    Write.Line("w nach dem Kopieren:");
    for(int i=0; i<w.length; i++) {
        Write.String("w[" + i + "]: ");
      for (int j=0; j < w[i].length; j++)
        Write.String("  " + w[i][j]);
      Write.Ln();
    }

  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac ArrCopy2.java
spatz$ java ArrCopy2
w nach dem Kopieren:
w[0]:   A  A  A  A  A  A
w[1]:   B  B  B  B  B  B
w[2]:   X  X  X  X  X  X
w[3]:   X  X  X  X  X  X
w[4]:   E  E  E  E  E  E
spatz$
```

Programm 7.34 (S. 155) zeigt ein Beispiel mit ungleichen Dimensionen.

Programm 7.34: Kopieren mehrdimensionaler Arrays (*ArrCopy3.java*)

```
import IOulm.*;

public class ArrCopy3 {
  public static void main(String [] args) {
    char [][] w = { {'A','A','A','A','A','A','A','A','A','A','A','A'},
                    {'B','B','B','B'}, {'C','C'}, {'D'}
                  };
    char [][] v = { {'X','X','X','X','X','X','X','X','X'},
                    {'Y'}, {'Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z'}
                  };
    char [][] u = new char[3][1];

    System.arraycopy(v,0,w,1,2);

    Write.Line("w␣nach␣dem␣Kopieren:");
    for(int i=0; i<w.length; i++) {
      Write.String("w[" + i + "]:␣");
      for (int j=0; j < w[i].length; j++)
      Write.String("␣␣" + w[i][j]);
      Write.Ln();
    }

    System.arraycopy(v,0,u,1,2);

    Write.Line("u␣nach␣dem␣Kopieren:");
    for(int i=0; i<u.length; i++) {
      Write.String("u[" + i + "]:␣");
      for (int j=0; j < u[i].length; j++)
      Write.String("␣␣" + u[i][j]);
      Write.Ln();
     }

  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac ArrCopy3.java
spatz$ java ArrCopy3
w nach dem Kopieren:
w[0]:   A  A  A  A  A  A  A  A  A  A  A  A
w[1]:   X  X  X  X  X  X  X  X  X
w[2]:   Y
w[3]:   D
u nach dem Kopieren:
u[0]:   \0
u[1]:   X  X  X  X  X  X  X  X  X
u[2]:   Y
spatz$
```

Aber Achtung:

Ist im Ziel-Array die “innere Dimension” bei der Deklaration nicht angegeben, so ist dort zunächst die `null`-Referenz eingetragen – wird diese beim Kopieren nicht ersetzt, so liefert der Zugriff einen Fehler. Dies demonstriert Programm 7.35, S. 156.

Programm 7.35: Kopieren mit Fehler (*ArrCopy4.java*)

```
import IOulm.*;

public class ArrCopy4 {
  public static void main(String [] args) {
    char [][] v = { {'X','X','X','X','X','X','X','X','X'},
                    {'Y'}, {'Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z','Z'}
                  };
    char [][] u = new char [3][]; // <------

    System.arraycopy(v,0,u,1,2);

    Write.Line("u nach dem Kopieren:");
    for(int i=0; i<u.length; i++) {
      Write.String("u[" + i + "]: ");
      for (int j=0; j < u[i].length; j++)
      Write.String("  " + u[i][j]);
      Write.Ln();
    }
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac ArrCopy4.java
spatz$ java ArrCopy4
u nach dem Kopieren:
u[0]: Exception in thread "main" java.lang.NullPointerException
        at ArrCopy4.main(ArrCopy4.java:15)
spatz$
```

7.7.5 Klonen von Arrays

Statt wie bislang Arrays zu kopieren, kann man mit der Methode **clone()** ein identisches Abbild im Speicher erstellen. So wie die Wertzuweisung nur die Referenzen kopiert, so vergleicht der Gleichheits-Operator **==** auch nur die Referenzen auf Gleichheit. Für Arrays gibt es hier die Methode **Arrays.equals()** .

Programm 7.36, S. 157, zeigt eine kleine Anwendung.

Programm 7.36: Klonen und Vergleichen (*ArrayClone.java*)

```java
import IOulm.*;
import java.util.*;

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

      int[] a = {1,2,3};
      int[] b = (int[]) a.clone();
      if (a == b )
        Write.Line("a and b are equal");
      else
        Write.Line("a and b are not equal");
      if (Arrays.equals(a,b) )
        Write.Line("a and b are equivalent");
      else
        Write.Line("a and b are not equivalent");

      b[1] = 100;
      if (Arrays.equals(a,b) )
        Write.Line("a and b are equivalent");
      else
        Write.Line("a and b are no longer equivalent");

      for (int i = 0; i < a.length; i++)
          Write.Line("a[" + i + "] = " + a[i] + "/ b[" + i + "] = " + b[i]);
       Write.Ln();
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac ArrayClone.java
spatz$ java ArrayClone
a and b are not equal
a and b are equivalent
a and b are no longer equivalent
a[0] = 1/ b[0] = 1
a[1] = 2/ b[1] = 100
a[2] = 3/ b[2] = 3
spatz$
```

Klonen bei mehrdimensionalen Arrays zeigt beispielhaft Programm 7.37, S. 158.

Programm 7.37: Klonen mehrdimensionaler Arrays (*ArrayClone1.java*)

```java
import IOulm.*;
import java.util.*;

public class ArrayClone1 {
  public static void main(String[] args) {

     int [][] data = {{1,2,3,4}, {5,6}, {7,8,9}};
      // Kopie vorbereiten
      int [][] copy = new int[data.length][];
      // Referenzen klonen
      for(int i =0; i < data.length; i++)
         copy[i] = (int []) data[i].clone();
      if (Arrays.equals(data,copy))
        Write.Line("data and copy are equivalent");
      else
        Write.Line("data and copy are not equivalent");
      for(int i = 0; i < copy.length; i++) {
        for(int j = 0; j < copy[i].length; j++)
           Write.String("" + copy[i][j] + " ");
        Write.Ln();
      }
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac ArrayClone1.java
spatz$ java ArrayClone1
data and copy are not equivalent
1 2 3 4
5 6
7 8 9
spatz$
```

Anm.: Es ist klar, dass `data` und `copy` nicht identisch sind – die Referenzen in der ersten Dimension sind verschieden!

7.7.6 Strings und char-Arrays

In der Klasse String gibt es einige nützliche Methoden, um Strings in char-Arrays und umgekehrt zu verwandeln:

(siehe: `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html`

`String(char[] value)`
Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

`String(String original)`
Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

`String(char[] value, int offset, int count)`
Allocates a new String that contains characters from a subarray of the character array argument.

`void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
Copies characters from this string into the destination character array.

Programm 7.38 (S. 159) zeigt einige Beispiele.

Programm 7.38: Strings und char-Arrays (*StringArray.java*)

```java
import IOulm.*;

public class StringArray {
  public static void main(String[] args) {
    char[] w = {'a','b','c','d','e','f','g'};
    char[] v = new char[20];
    String s1 = new String(w); String s2 = new String(w,2,3);
    String s3 = "abcdefg"; String s4 = "Hello World";

    Write.Line("s1: " + s1); Write.Line("s2: " + s2);

    if ( s1.equals(s3) ) Write.Line("gleich");
    else Write.Line("ungleich");

    s4.getChars(0,4,v,0);
    for (int i = 0; i < v.length; i++) Write.Char(v[i]);
    Write.Ln();
  }
}
```

Übersetzung und Ausführung:

```
spatz$ javac StringArray.java
spatz$ java StringArray
s1: abcdefg
s2: cde
gleich
Hell\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
spatz$
```

7.8 Methoden (Prozeduren, Funktionen)

[...] procedures are one of the most powerful features of a high level language, in that they simplify the programming task and shorten the object code.

C.A.R. Hoare, 1981

7.8.1 Unterprogrammtechnik

- Allgemeiner Begriff: **Unterprogramm**
- analog zu Funktionen in der Mathematik:

$$sin(x) \; = \; \sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!}$$

- Programm zur Berechnung des Sinus in Abhängigkeit eines formalen Parameters wird nur einmal geschrieben – wird der Sinus zu einem aktuellen Parameter a an anderer Stelle benötigt, so wird
  - der Text zur Sinusberechnung mit Ersetzung von x durch a an diese Stelle kopiert (offener Einbau, **Makrotechnik**)
  - der Text zur Sinusberechnung "angesprungen", der formale Parameter wird an den aktuellen Parameter "gebunden", und nach Beendigung der Berechnung geht es nach der Aufrufstelle weiter – der Aufruf wird durch den Ergebniswert ersetzt (siehe Abb. 7.17, 161).

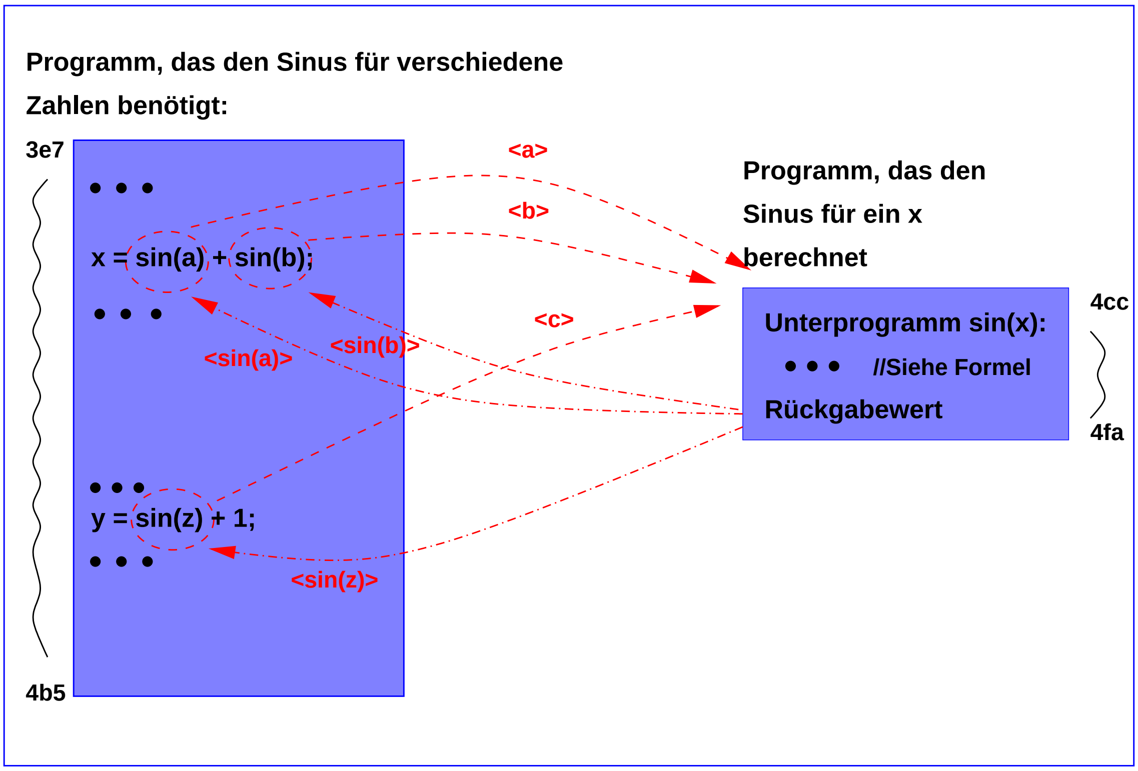


Abbildung 7.17: Unterprogrammtechnik mit Ansprung

7.8.2 Konzepte und Terminologie

- Ein Unterprogramm ist ein mit einem Namen versehener, parametrisierter Anweisungsblock (**Prozedurrumpf**), bestehend aus einem Vereinbarungsteil für lokale Objekte und der Berechnungsvorschrift
- Vor dem Rumpf steht der **Prozedurkopf**, bestehend aus dem Namen der Prozedur, den Namen und Typen der **formalen Parameter** und ggf. dem **Ergebnistyp**
- Die formalen Parameter gelten wie die im Rumpf deklarierten Objekte als **lokal**
- Der Block kann mit **passenden** aktuellen Parametern über seinen Namen aufgerufen werden
- Bei Prozeduren mit Ergebnis wird dessen Typ explizit im Prozedurkopf genannt: in manchen Sprachen (wie Java oder C) **vor** dem Prozedurnamen, in anderen (wie Modula-2, Oberon) **nach** der Parameterliste; der Ergebniswert wird meist durch eine **return**-Anweisung definiert
- Prozeduren mit Ergebniswert nennt man meist Funktionsprozeduren
- In OO-Sprachen wie Java spricht man statt von Prozeduren von **Methoden**. Man kennt hier zwei Arten von Methoden: solche, die an die Existenz einer konkreten Instanz einer Klasse gebunden sind und **auf dieser Instanz** ausgeführt werden (**Instanz-Methoden**), und solche, die für die Klasse als Gesamtes definiert sind (**Klassenmethoden**). Zunächst wird nur von Klassenmethoden die Rede sein – diesen wird immer das Schlüsselwort **static** vorangestellt!.

- Innerhalb von Methoden können Variable deklariert werden, jedoch keine Funktionen!
- Innerhalb von Methoden können andere Methoden aufgerufen werden, auch die Methode selbst (**rekursiver Aufruf**)!

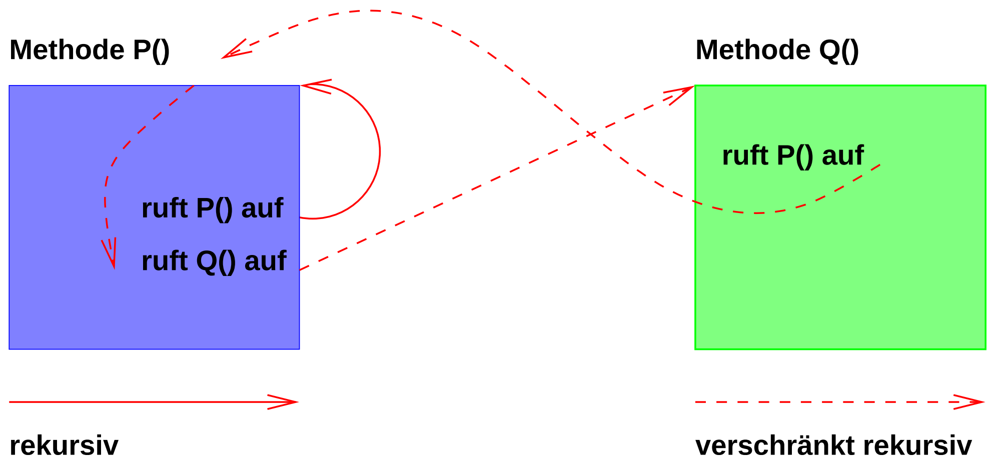


Abbildung 7.18: Rekursion bei Unterprogrammen

Beispiel: Bestimmung der harmonischen Reihe

$$harm(k) = \sum_{i=1}^{k} \frac{1}{i}$$

Das Programm 7.39, S. 163, enthält zwei Unterprogramme *harm1()* und *harm2()*.

Programm 7.39: Harmonische Reihe (*HarmReihe.java*)

```java
import IOulm.*;

public class HarmReihe{

 static float harm1 ( int k) {
      int i = 1;
      float res = 0.0f;
      while ( i <= k ) {
           res += 1.0/i; // why not 1/i ?
           i++;
      }
      return res;
}

 static float harm2 ( int k) {
      float res = 0.0f;
      while ( k >= 1 ) {
           res += 1.0/k;
           k--;
      }
      return res;
}


  public static void main(String[] args) {
      int i;
      if ( Urc.readInt() ) i = Urc.getInt() ; else return;
      Write.Line("Harmonische Reihe bis " + i);
      Write.Line(harm1(i) + " (von gross nach klein)");
      Write.Line(harm2(i) + " (von klein nach gross)");
  }
}
```

```
thales$ java HarmReihe
100000000
Harmonische Reihe bis 100000000: 18.997896413852555
Andersherum:                     18.997896413853447
thales$
```

Beispiel: Integer-Zahlen als Bitmuster ausgeben / über Bitpositionen erstellen

Programm 7.40: Bitmuster und Integer (*BitHandling.java*)

```
1 import IOulm.*;
2
3 public class BitHandling {
4
5   static int composeBitPattern( int[] bitPosition ) {
6     int res = 0;
7     for (int i = 0; i < bitPosition.length; i++)
8         res |= (01 << bitPosition[i]);
9     return res;
10  }
11
12  static void printBitPattern(int z) {
13    for (int mask = 1 << 31; mask != 0; mask >>>= 1) {
14      if ( (z & mask) != 0 )
15        Write.Char('1');
16      else
17        Write.Char('0');
18    }
19    Write.Ln();
20  }
21
22  public static void main(String[] args) {
23
24    int[] bitPosition = new int[] {0, 3, 7, 28, 31};
25    printBitPattern(1000000);
26    printBitPattern(composeBitPattern(bitPosition));
27
28  }
29 }
```

Anmerkungen:

Zeile 8: Der Operator „|=“ in „`a |= b`“ steht für „`a = a | b`“, der Operator „|“ ist das bitweise inklusive Oder

Zeilen 8, 13: Der Operator „<<“ ist der Shift-Operator nach links!

Zeile 13: Der Ausdruck „`mask >>>= 1`“ ist eine Kurzschreibweise für „`mask = mask >>> 1`“; der Operator „>>>“ bedeutet vorzeichenloser Rechtsshift (von Links werden Nullen nachgezogen)

Zeile 14: Der Operator „`&`“ ist das bitweise Und

7.8.3 Signaturen

Allgemein bilden Name, Anzahl und Reihenfolge der Parameter sowie ihrer Typen und u.U. auch der Typ des Resultats die **Signatur** eines Unterprogramms (auch als **Aufrufschnittstelle** bezeichnet). Sie gibt die Syntax des Aufrufs wieder. Unterprogramme werden anhand ihrer Signatur unterschieden. Haben sie gleichen Namen, aber unterschiedliche Signatur, so spricht man von **Überladen** (*overloading*) des Namens.

Beispiel:

Seien die Funktionen `f(int,long)` und `f(long,long)` definiert. Die erste Variante ist **spezieller** als die zweite, da jeder Aufruf der ersten auch durch die zweite ausgeführt werden kann. Ist weiter `f(long,int)` gegeben, so ist auch dieser spezieller als die zweite Variante, während diese mit der ersten Variante unvergleichbar ist (was dann dazu führt, dass der Aufruf `f(1,2)` ungültig ist.

Programm 7.41: Signaturen und Überladen (*Signature.java*)

```java
1  import IOulm.*;
2
3  public class Signature {
4
5    static long f(int i, long l) {
6      return i + l;
7    }
8    static long f(long i, long l) {
9      return i + l;
10   }
11   static long f(long i, int l) {
12     return i + l;
13   }
14   public static void main(String[] args) {
15         Write.Long(f (1,2));   Write.Ln();
16   }
17 }
```

```
sperber$ javac Signature.java
Signature.java:15: reference to f is ambiguous, both method f(int,long)
in Signature and method f(long,int) in Signature match
       Write.Long(f(1,2)); Write.Ln();
                    ^
1 error
sperber$
```

Beachte: Zwei Java-Methoden können sich nicht lediglich in ihrem Resultatstyp unterscheiden!

7.8.4 Parameterübergabe

- Werteaufruf (*call by value*) → Java, C/C++, Modula-2, Oberon, u.a.

  Der aktuelle Parameter wird ausgewertet – der dabei entstehende Wert wird dem formalen Parameter (als lokale Variable) zugewiesen

- Referenzaufruf (*call by reference*) → C++, Perl, Modula-2, Oberon, u.a.

  Der aktuelle Parameter ist eine Variable oder ein Ausdruck, der als Linkswert vorkommen kann, also auf der linken Seite einer Wertzuweisung stehen kann (z.B. x[i+2]). Dieser Linkswert wird für den Linkswert des formalen Parameters eingesetzt. Der formale Parameter wird damit ein Alias für den aktuellen Parameter. Der Wert des aktuellen Parameters wird damit auch **nicht kopiert**, der formale Parameter verweist auf diesen Wert.

- Namensaufruf (*call by name*) → Algol68

  Der aktuelle Parameter ist ein beliebiger Ausdruck, der ohne jede Auswertung textuell für den formalen Parameter substituiert wird. Der Ausdruck wird dann im Kontext des Unterprogramms immer dort evaluiert, wo der formale Parameter vorkommt. Damit darf der Ausdruck nur solche Bezeichner enthalten, die innerhalb des Unterprogramms Gültigkeit haben (können durchaus auch ausserhalb definiert sein). Der Namensaufruf entstammt dem sog. λ-Kalkül (ein formales Berechnungsmodell), spielt in heutigen Programmiersprachen keine Rolle mehr.

Anhand von Programm 7.42 (S. 166) sollen diese drei Konzepte erläutert werden.

Programm 7.42: Parameterübergabe – Allgemeine Konzepte (*Parameter.java*)

```
import IOulm.*;

public class Parameter {

  static int i = 0;
  static int [] a = new int[] {10,11};

  static void f(int x) {
     i++; x = 5 * x;
  }

 public static void main(String [] args) {
    f(a[i]);
    Write.Line("a[0] = " + a[0] + ", a[1] = " + a[1]);
  }
}
```

call by value Zu Beginn von f wird implizit die Zuweisung $x = 10$ ausgeführt; dann wird die **globale Variable** um 1 erhöht und die **lokale Variable** x verfünffacht – dies hat auf $a[0]$ keine Auswirkung, ausgegeben würde also

```
a[0] = 10, a[1] = 11
```

call by reference Hier wäre x ein Alias für $a[0]$ (häufig als $x \equiv a[0]$ geschrieben). In f würde also $a[0] = 5 * a[0]$ ausgeführt werden. Die Ausgabe wäre dann

```
a[0] = 50, a[1] = 11
```

call by name In f wird x textuell durch $a[i]$ ersetzt – der Prozedurrumpf wäre dann aktuell:

```
i++; a[i] = 5 * a[i];
```

Die Ausgabe wäre dann

```
a[0] = 10, a[1] = 55
```

Programm 7.43, S. 167, zeigt einige Spielereien.

Programm 7.43: Parameterübergabe (*Params.java*)

```java
import IOulm.*;

public class Params{

  static void f(int i){ i++; }
  static void g(int v[]) {
    v[0] = 9; v[1] = 8;
  }
  static void h(String[] sv) {
    sv[0] = ">>" + sv[0] + "Mist bleibt Mist" + "<<";
  }
  static void h1(String str) {
    str = ">>" + str + "Mist bleibt Mist" + "<<";
  }

  public static void main(String[] args) {
      int k = 10;
      int[] a = new int[5];
      for (int i = 0; i < a.length; i++) a[i]=i;
      String[] str = new String[] {"Stroh", "Heu", "Gras"};
      String s = "Stroh";
      Write.String("1 k (vorher):  " + k + " | ");
      f(k);
      Write.Line("k (nachher): " + k);
      Write.String("2 a (vorher):  ");
      for(int i = 0; i < a.length; i++)
         Write.String("  " + a[i]);
      Write.Ln();
      g(a);
      Write.String("3 a (nachher): ");
      for(int i = 0; i < a.length; i++)
         Write.String("  " + a[i]);
      Write.Ln();
      Write.String("4 " + str[0] + " wird zu ");
      h(str);
      Write.Line(str[0]);
      Write.String("5 s (vorher):  " + s + " | ");
      h1(s);
      Write.Line("s (nachher): " + s);
  }
}
```

Übersetzung und Ausführung:

```
sperber$  javac Params.java
sperber$  java Params
1 k (vorher):  10 | k (nachher): 10
2 a (vorher):    0  1  2  3  4
3 a (nachher):   9  8  2  3  4
4 Stroh wird zu >>StrohMist bleibt Mist<<
5 s (vorher):  Stroh | s (nachher): Stroh
sperber$
```

Erläuterungen:

Ausgabezeile 1: Klar, k bleibt unverändert, da in $f()$ eine Kopie mit dem Wert von k angelegt; Veränderung der Kopie lässt das Orginal unverändert.

Ausgabezeilen 2 und 3: Der Vektorname a ist eine Referenz, von der eine Kopie an $g()$ übergeben wird und dort v heisst. Es ist gilt: a **und** v **beziehen sich auf dasselbe Objekt!** Der Durchgriff über v zu den ersten beiden Elementen des Array ist genauso wie wenn er über a erfolgen würde!

Ausgabezeile 4: (siehe Programmzeilen 20 sowie 34,35,36) – Erklärung wie zuvor!

Ausgabezeile 5: s ist eine Referenz auf das konstante (unveränderliche) Objekt "*Stroh*", eine Kopie dieser Referenz wird in $h1()$ mit dem Namen *str* verwendet. An *str* wird nun eine neue Referenz zugewiesen, die Referenz in s bleibt natürlich unverändert!

7.8.5 Exkurs: Blöcke, Gültigkeitsbereich und Lebensdauer

Ein **Block** ist eine Zusammenfassung von Anweisungen und gilt dann logisch als eine Anweisung – die Zusammenfassung erfolgt durch Klammerung mit geschweiften Klammern.

- Blöcke können beliebig ineinander verschachtelt werden (Achtung: Lesbarkeit!)
- Moderne Programmiersprachen lassen in jedem Block die Deklaration von neuen Objekten (Variable, Klassen) durch Angabe von Name und Typ zu!
- Die **Gültigkeit** der in einem Block deklarierten Objekte ist auf den Block beschränkt, das Objekt ist **lokal** zu seinem Block!
- Der Rumpf einer Klasse kann als Block angesehen werden:

```
public class Params{
   // Variablendeklarationen
   // (im OO-Jargon: Klassen- / Objekt-Datenfelder
   // Prozedurdeklarationen
   // (im OO-Jargon: Klassen- / Objekt-Methoden
}
```

- Auch der Rumpf einer Funktion kann als Block angesehen werden: in ihm (so in Java) können nur Variable deklariert werden! Schleifen bilden innerhalb von Funktionen neue Blöcke!
- In Java ist die Gültigkeit einer Variablen der Block (und alle dazu inneren Blöcke), in dem sie deklariert wurde. In anderen Sprachen kann die Gültigkeit in einem inneren Block aufgehoben sein, wenn dort ein neues Objekt gleichen Namens deklariert wird.

Programm 7.44: Verletzung des Gültigkeitsbereichs (*Valid1.java*)

```java
public class Valid1 {

  { int x = 1;
    int y;
    {  int x = 2;
          y = x;
    }
    y = x;
  }

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

Übersetzung von Programm 7.44 (S. 169) liefert:

```
sperber$ javac Valid1.java
Valid1.java:5: x is already defined in
    {  int x = 2;
             ^
1 error
sperber$
```

Programm 7.45 (S. 170) zeigt eine korrekte Verwendung von Blöcken. Im Block 1.1 sind die Variablen c und y sowie auch die Objekte im umgebenden Block (also x) sichtbar, nicht jedoch die in Block 1.2 und auch nicht die in Block 1.1.1 deklarierten Variablen. Im Block 1 ist nur die Variable x sichtbar! Bei geschachtelten Blöcken geht die Sichtbarkeit von innen nach außen, nicht jedoch von außen nach innen!

Programm 7.45: Korrekte Gültigkeitsbereiche (*Valid2.java*)

```java
public class Valid2 {

  { // Beginn Block 1
    int x;
    { // Beginn Block 1.1
      int y = 1; char c = 'q';
      { // Beginn Block 1.1.1
        int k = 10;
      }
      x = y;
    } // Ende Block 1.1
    { // Beginn Block 1.2
      int y = 2; double d;
      x = y;
    } // Ende Block 1.2
  } // Ende Block 1

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

- Die **Lebensdauer** einer Variablen ist die Zeit vom Eintreten bis zum Verlassen des Blockes, in dem diese deklariert wurde. In Programm 7.45 (S. 170) existiert die Variable x mit Beginn von Block 1 bis zu seinem Ende; die Variable y in Block 1.1 hat mit der in Block 1.2 nichts zu tun!

Speicherverwaltung:

Jedem Block entspricht ein Speicherbereich (**Frame**) auf dem Laufzeitstapel (**run-time stack**), wo unter anderem die Werte der lokalen Variablen abgelegt werden. Beim Eintreten in einen Block wird ein entsprechender Frame oben auf dem Stack alloziert, beim Verlassen wird er wieder freigegeben. Die Verwaltung dieser Frames erfolgt nach dem **LIFO**-Prinzip : *last in, first out*. Dies zeigt für das Programm 7.45 (S. 170) die Abbildung 7.19 (S. 171).

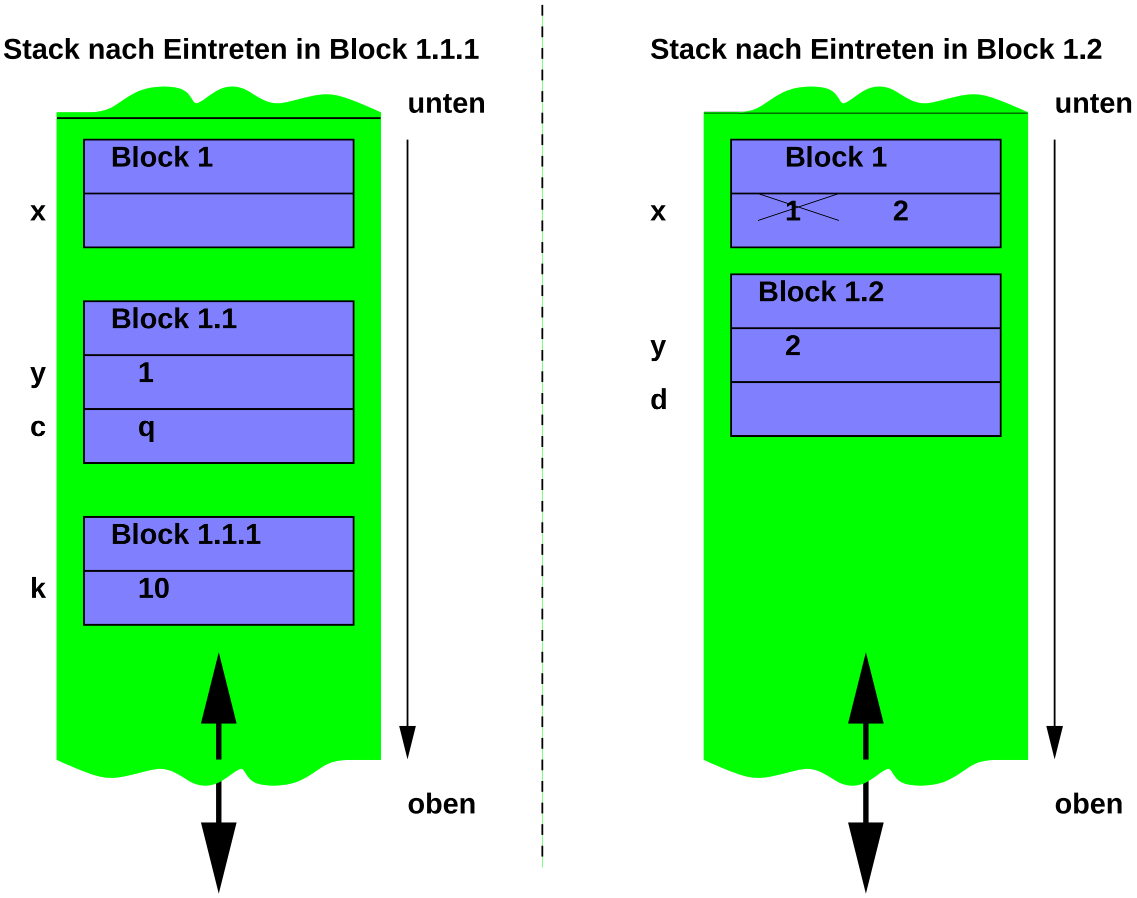


Abbildung 7.19: Stack für Blockstruktur

Tritt der Kontrollfluss in einen neuen Block – das kann durch einen Prozeduraufruf passieren – so wird ein neuer Frame auf dem Stack angelegt. Bei einem rekursiven Eintreten in den Block (rekursiver Prozeduraufruf) wird jedesmal ein neuer Frame angelegt, jede Variable erfährt eine neue **Inkarnation**: ihr Name wird an einen neuen Speicherplatz gebunden. Wird ein Block verlassen, so wird der entsprechende Frame freigegeben (Pulsieren des Stack).

Der **Haldenspeicher** (**Heap**) kennt weder Frames noch ein Pulsieren; er enthält alle mit **new** erzeugten Klassenobjekte. Das Objekt auf der Halde lebt weiter, auch wenn die entsprechende Referenzvariable vom Stack verschwindet. Gibt es zu einem Objekt keine Referenz mehr, ist es "Müll" (**garbage**) und kann vom **garbage collector** eingesammelt werden und der zugehörige Speicher kann freigegeben werden.

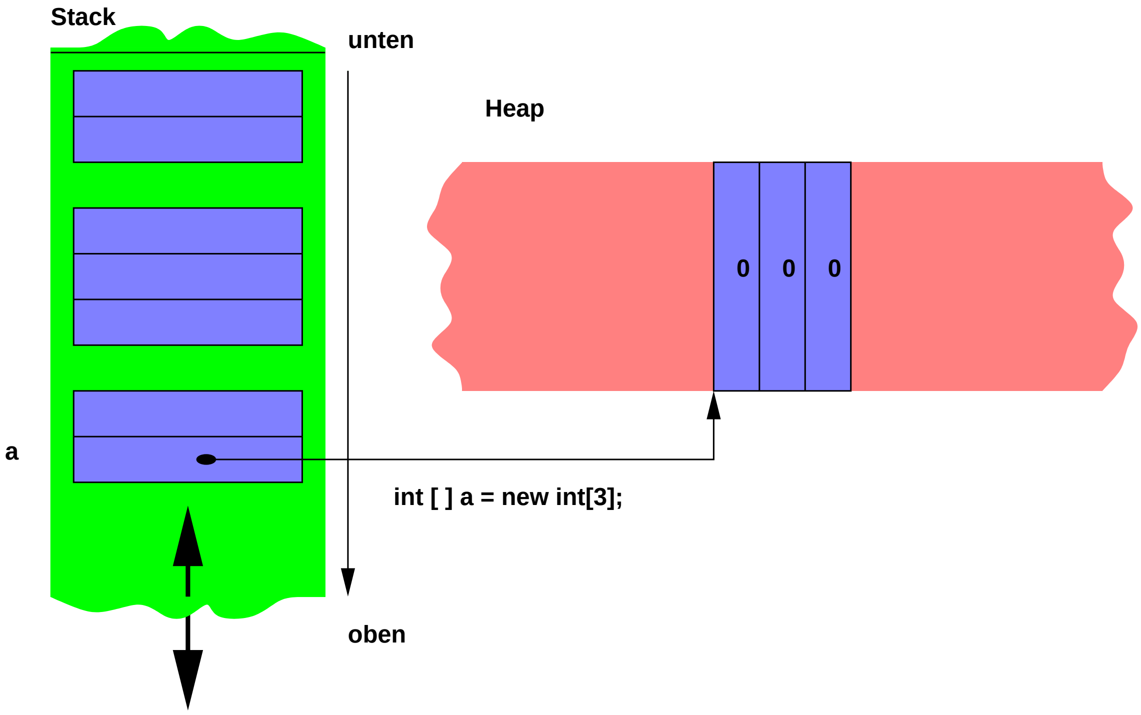


Abbildung 7.20: Stack und Heap

Es ist aber insbesondere möglich, eine Referenz auf das Haldenobjekt an eine Variable in einem äußeren Block zuzuweisen und damit ein Objekt "nach draußen" weiterzureichen.

7.8.6 Dokumentationskommentare: javadoc

Bei Klassen und bei Methoden sollte immer möglichst präzise angegeben werden, wie sie zu handhaben sind, welche Voraussetzungen für die Benutzung gestellt sind und welche Bedingungen nach Beendigung der Leistungserbringung erfüllt sind. Für eine Methode $f()$ ist also anzugeben, welche **Zusicherungen** (*assertion, input assertion*) $f()$ an sein Ergebnis wahr macht und welche Anforderungen (*requirement, input assertion*) $f()$ dafür an seine Parameter (und ggf. andere implizite Eingabevariablen) stellt. Zusammen mit der Signatur von $f()$ bilden sie den **Kontrakt** der Funktion.

Programm 7.46 (S. 173) zeigt ein einfaches Beispiel.

Programm 7.46: Programm mit Dokumentationskommentaren (*doc/MyMath.java*)

```
/**
 * Class for some mathematical functions
 *
 * @see       Math
 * @author    swg @ SAI
 * @version   1.1
 */

public class MyMath{

/**
 * computes the power a^b;
 *  attention : there is no check for possible overflow
 * @see Math.pow
 * @param a Base, --
 * @param b Exponent, b &ge; 0
 * @return a^b
 * @author Franz Schweiggert
 * @version 1.1, Nov 2005
 */
 public static int power(int a, int b) {
     int p = 1;
     while (b > 0) {
         p *= a; b--;
     }
     return p;
 }
}
```

Kommentare, die mit `/**` beginnen (und mit `*/` enden), werden als Dokumentationskommentare bezeichnet. Diese können mit dem Programm **javadoc** zu einer HTML-basierten Dokumentation extrahiert werden. *javadoc* kennt weitere Schlüsselworte (z.B. `@param` oder `@author`). Zusätzlich können HTML-Befehle verwendet werden.

Was sollte wie dokumentiert werden?

- Anforderungen und Zusicherungen, die sich aus den Typen in der Signatur ergeben, werden nicht wiederholt.
- Es sind grundsätzliche **alle** Parameter aufzuführen – falls es für einen Parameter keine Anforderung gibt, so dies durch die Formel *true* oder einen Leerstrich ausgedrückt!
- Der Spezifikation kann ein Kurzkommentar vorausgehen, der z.B. Hinweise über Sinn, Zweck und Motivation enthält. Weitere Hinweise können der Spezifikation folgen: Autor, Datum, Programmiermethoden, Literaturreferenzen, …
- globale Variable, die verwendet / verändert werden, sind wie Eingabe- / Ausgabeparameter zu beschreiben.

7.8.7 Rekursion

- Ein Unterprogramm $p()$ heißt **rekursiv**, wenn in seinem Rumpf ein Aufruf von $p()$ erfolgt.
- Zwei Unterprogramme $p()$ und $q()$ sind wechselseitig rekursiv, wenn im Rumpf von $p()$ ein Aufruf von $q()$ erfolgt und im Rumpf von $q()$ ein Aufruf von $p()$ erfolgt.

Rekursion bedeutet also Wiederholung, und wie jede Schleife, so muss die Rekursion terminieren. Dies geschieht auch hier über eine Abbruchbedingung.

Programm 7.47 (S. 174) zeigt eine iterative und eine gleichwertige rekursive Berechnung von

$$\sum_{i=0}^{n} i$$

.

Programm 7.47: Iterativ und Rekursiv (*SumRek.java*)

```java
import IOulm.*;

public class SumRek{
  static int iterativSum (int n) {
    int i, s = 0;
    for(i = n; i > 0; i--)
       s += i;
    return s;
  }
  static int rekursivSum(int n) {
     if ( n > 0 ) return (n + rekursivSum(n-1) );
     else return 0;
  }

  public static void main(String[] args) {
       int n;
       if ( Urc.readInt() ) {
          n = Urc.getInt();
          Write.Line(" Iterativ : " + iterativSum(n));
          Write.Line("Rekursiv: " + rekursivSum(n));
       }
       else return;

  }
}
```

Übersetzung und Ausführung:

```
sperber$ javac SumRek.java
sperber$ java SumRek
10000
Iterativ: 50005000
Rekursiv: 50005000
sperber$ java SumRek
1000000
Iterativ: 1784293664
Exception in thread "main" java.lang.StackOverflowError
sperber$
```

Dass es bei $n = 1000000$ zu einem Programmabbruch kommt mit dem Hinweis auf *StackOverflow*, muss nicht verwundern – der Stack ist in seiner Größe beschränkt (siehe Abb. 7.21, S. 175).

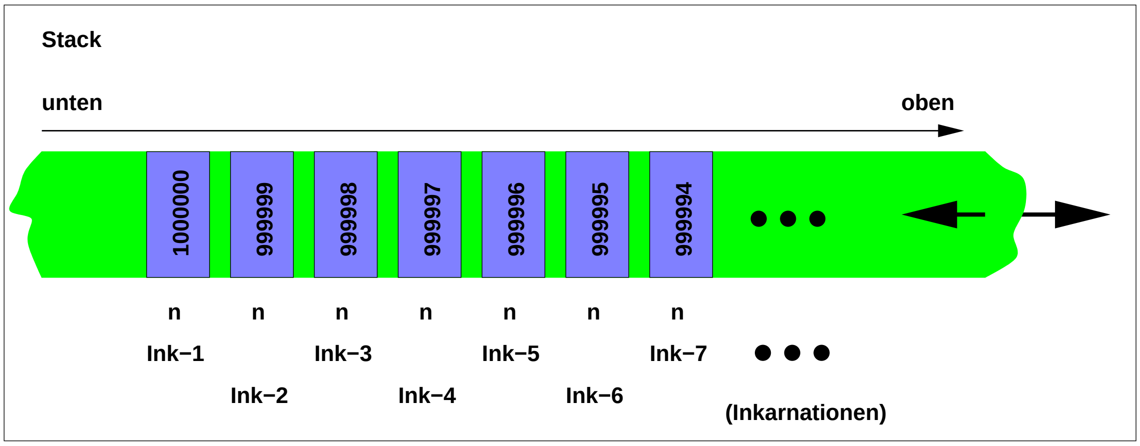


Abbildung 7.21: Stack Overflow

Bei der Rekursion stellen sich analog zum Beweisen mit vollständiger Induktion folgende Fragen:

1. Was ist der Basisfall und wie wird er gelöst?
   a) Ist es der absolute Trivialfall?
      Im Beispiel: $n < 0$ soll zum Ergebnis 0 führen
   b) Ist es der einfachste, nichttriviale Fall?
      Im Beispiel: $n = 0$ hat als Ergebnis 0
2. Wie kann der allgemeine Fall der Größe n auf die Lösung für eine Größe $n' < n$ reduziert werden?

   Im Beispiel:

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$

Dies führt zu der in Programm 7.48 (S. 175) enthaltenen verbesserten Version.

Programm 7.48: Rekursion und Induktion (*SumRek1.java*)

```java
import IOulm.*;

public class SumRek1{
  static int rekursivSum(int n) {
     if ( n < 0 ) return 0;
     else if (n == 0) return 0;
     else return (n + rekursivSum(n-1) );
  }

  public static void main(String[] args) {
       int n;
       if ( Urc.readInt () ) {
          n = Urc.getInt ();
```

```
            Write.Line("Summe: " + rekursivSum(n));
        }
        else return;

    }
}
```

Beispiel: Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen

Für $a \in N, b \in N$ gilt: $a > b \Rightarrow ggT(a,b) = ggT(b, a\%b)$

Beweis:

Sei $r = GGT(b, a\%b)$, d.h. r teilt sowohl b als auch $a\%b$; wegen

$$a = (a \div b) \cdot b + a\%b \quad (*)$$

teilt dann r auch a ($\div$ sei die ganzzahlige Division).

Sei r' ein gemeinsamer Teiler von a und b; dann ist r' wegen $(*)$ auch Teiler von b und $a\%b$, d.h. r' teilt auch r, also ist $r' \leq r$.

q.e.d.

Die Implementierung ist in Programm 7.49 (S. 176) dargestellt.

Programm 7.49: Größter gemeinsamer Teiler – rekursiv (*GGTRek.java*)

```
import IOulm.*;

public class GGTRek{

  /**
   * Groesster gemeinsamer Teiler
   * @param a, a &gt; 0, a &ge; b
   * @param b, b &ge; 0, b &le; a
   * @return result, die groesste Zahl mit result &le; a,
   *         result &le; b und a%result == 0, b%result == 0
   */
   public static int ggT(int a, int b) {
      // Trivialfall
      if ( b == 0) return a;

      // Reduktion und Rekursion
      return ggT(b, a %b);
   }

  public static void main(String[] args) {
      int n, m, res;
      if ( Urc.readInt() ) { n = Urc.getInt(); }
      else return ;
      if ( Urc.readInt() ) { m = Urc.getInt(); }
```

```
        else return ;
        if ( (n <= 0) || (m <= 0) ) return;
        res = ( n >= m ) ? ggT(n,m) : ggT(m,n);
        Write.Line("ggT von " + n + " und " + m + " ist " + res );
    }
}
```

Endrekursion (*Tail Recursion*)

Schleife vs. Rekursion:

- Bei einer Schleife sind die Variablen des Schleifenrumpfes nur einmal existent – eine Zuweisung überschreibt ihre Werte!
- Bei rekursiven Prozeduren werden die lokalen Variablen bei jedem Aufruf neu erzeugt – genauer: jede Variable erhält eine neue "Inkarnation"!
- Nach Rückkehr aus einem Aufruf stehen die alten Werte wieder zur Verfügung (siehe Abb. 7.21, S. 175, bzw. Abb. 7.22, S. 177)

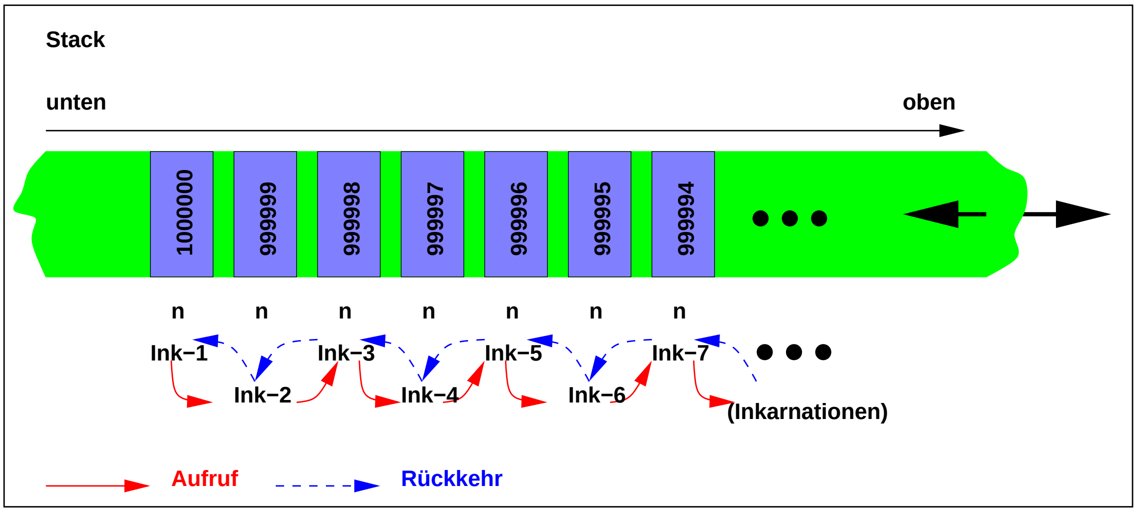


Abbildung 7.22: Inkarnationen

- Ein Aufruf heißt **endrekursiv** (*tail recursive*), wenn er **die letzte Operation** dieser Prozedur ist!
- Bei Prozeduren mit einem einzigen endrekursiven Aufruf werden die lokalen Variablen nach der Rückkehr des rekursiven Aufrufs nicht mehr benötigt, sie könnten auch überschrieben werden. Diese Art der Rekursion kann ohne Schwierigkeit durch eine Schleife ersetzt werden (was auch viele Compiler implizit so umsetzen)!

Beispiel: die Funktion `RekSum1()` in Programm 7.48, S. 175, ist **nicht** endrekursiv, da nach der Rückkehr aus dem rekursiven Aufruf noch die Addition `n + RekursivSum(n-1)` durchgeführt werden muss!

Programm 7.50 (S. 178) zeigt eine Funktion `sum()`, die eine endrekursive Funktion `tailSum()` aufruft!

Programm 7.50: Endrekursive Funktion (*SumRek2.java*)

```
import IOulm.*;

public class SumRek2{
   static int tailSum(int n, int res) {
      if ( n == 0 ) return res;
      else return tailSum(n-1, n+res);
   }
   static int sum(int n) {
      if ( n < 0 ) return 0;
      else return tailSum(n,0);
   }

  public static void main(String[] args) {
       int n;
       if ( Urc.readInt() ) {
          n = Urc.getInt();
          Write.Line("Summe: " + sum(n));
       }
       else return;

   }
}
```

Endrekursive Funktionen lassen sich leicht in eine iterative Form umformen: man fügt eine Schleife ein, ersetzt den rekursiven Aufruf durch Zuweisungen an die lokalen Variablen entsprechend der Parameterübergabe. Dies geht, da nach dem rekursiven Aufruf die lokalen Variablen nicht mehr benötigt werden (Endrekursivität); statt neue Inkarnationen anzulegen, können die Variablen also wiederverwendet werden.

Programm 7.51 (S. 178) zeigt dies für Programm 7.50 (S. 178).

Programm 7.51: Endrekursivierte Funktion (*SumRek3.java*)

```
import IOulm.*;

public class SumRek3{
   static int iterSum(int n, int res) {
      for (;;) { // Schleife als Ersatz der Rekursion
         if ( n == 0 ) return res;
         else { res = n + res; n = n - 1; // tailSum(n-1, n+res);
         }
      }
   }
   static int sum(int n) {
      if ( n < 0 ) return 0;
      else return iterSum(n,0);
```

```
  }

  public static void main(String [] args) {
      int n;
      if ( Urc.readInt () ) {
         n = Urc.getInt ();
         Write.Line("Summe: " + sum(n));
      }
      else return;

  }
}
```

Programm 7.52: Endrekursivierter GGT aus Programm 7.49, S. 176 (GGTIter.java)

```
import IOulm.*;

public class GGTIter{

  /**
   * Groesster gemeinsamer Teiler
   * @param a, a > 0, a >= b
   * @param b, b >= 0, b <= a
   * @return result , die groesste Zahl mit result <= a,
   *             result <= b und a%result == 0, b%result == 0
   */
   public static int ggTIter(int a, int b) {
      int tmp;
      for (;;) {
         if ( b == 0) return a;
         tmp = a; a = b; b = tmp % b; // return ggT(b, a %b);
      }
    }

  public static void main(String [] args) {
      int n, m, res;
      if ( Urc.readInt () ) { n = Urc.getInt (); }
      else return ;
      if ( Urc.readInt () ) { m = Urc.getInt (); }
      else return ;
      if ( (n <= 0) || (m <= 0) ) return;
      res = ( n >= m ) ? ggTIter(n,m) : ggTIter(m,n);
      Write.Line("ggT von " + n + " und " + m + " ist " + res );
  }
}
```

- Endrekursive Funktionen sollte man besser iterativ implementieren (Performance ist besser, *Stack overflow* kann vermieden werden)
- Rekursivität ist dann besonders elegant, wenn mehrere rekursive Aufrufe nötig sind – iterative Lösungen verlangen vom Programmierer eine eigene “Stackverwaltung”.

Ackermann- / Peter-Funktion

1926 vermutete David Hilbert, dass jede berechenbare Funktion sich aus wenigen sehr einfachen Regeln zusammensetzen lässt und sich die Dauer der Berechnung abschätzen lässt (primitiv rekursive Funktionen). Wilhelm Ackermann fand im gleichen Jahr eine Funktion, die diese Vermutung widerlegte (1928 veröffentlicht). 1955 konstruierte R. Peter eine vereinfachte Version. Diese wird heute meist Ackermann-, manchmal Peter- oder auch Ackermann-Peter-Funktion genannt. Sie ist definiert als $P : \mathbb{N} x \mathbb{N} \rightarrow \mathbb{N}$ mit

$$P(n,m) = \begin{cases} m+1 & \text{falls } n = 0 \\ P(n-1,1) & \text{falls } m = 0 \\ P(n-1, P(n,m-1)) & \text{sonst} \end{cases}$$

Programm 7.53: Ackermann-/Peter-Funktion (*Peter.java*)

```java
import IOulm.*;

public class Peter {

   public static int p(int n, int m) {
        int res;
        if (n == 0) res = m+1;
        else
            if (m == 0) res = p(n-1,1); // rek. Aufruf 1
            else res = p(n-1,           // rek. Aufruf 2
                          p(n,m-1));    // rek. Aufruf 3
        return res;
   }

  public static void main(String[] args) {
       int n, m;
       if ( Urc.readInt() ) { n = Urc.getInt(); }
       else return ;
       if ( Urc.readInt() ) { m = Urc.getInt(); }
       else return ;
       if ( (n < 0) || (m < 0) ) return;
       Write.Line("Ergebnis: " + p(n,m));
  }
}
```

Kapitel 8

Abstrakte Datentypen (FIFO und LIFO) mit erster Klassenbildung

8.1 Übersicht

Bei der Organisation von Daten benötigt man oft eine spezielle Zugriffsorganisation:

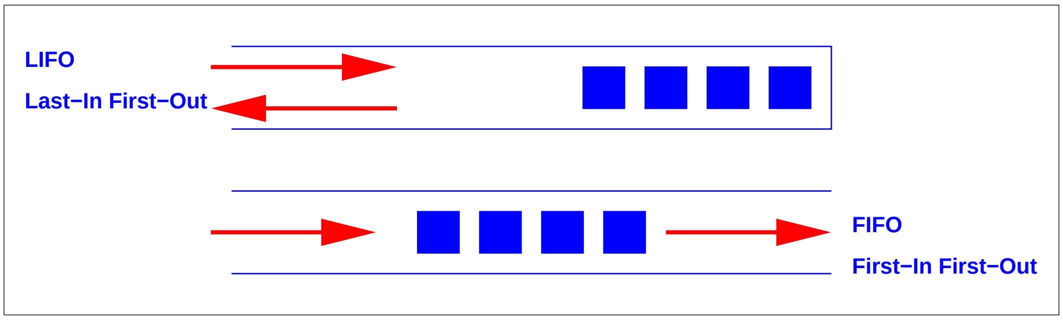


Abbildung 8.1: LIFO / FIFO

Die Struktur **LIFO** wird meist auch **Stack**, manchmal auch Kellerspeicher genannt; die Struktur **FIFO** wird häufig auch **Queue**, **Pipe** oder Warteschlange genannt. Manchmal benötigt man *Queues*, bei denen an beiden Enden hinzugefügt und entfernt werden kann; diese werden meist als **Deque** (*double ended queue*) bezeichnet.

Im folgenden sollen einige grundlegende Implementierungsaspekte dargestellt werden. Die Elemente in der Datenstruktur werden einfach ganze Zahlen sein, zur Implementierung wird ein (beschränktes) Array verwendet. Mit den Konzepten von modernen Programmiersprachen können natürlich auch unbeschränkte Strukturen realisiert werden, der Elementtyp wird dann auch erst bei Benutzung / Deklaration der Struktur bestimmt. Zudem wird man besser eine eigene Klasse *Stack* einführen, so dass andere Programme darauf aufbauend entsprechende Objekte erzeugen können!

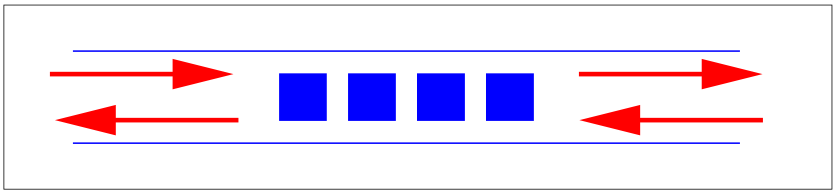


Abbildung 8.2: Deque – double ended queue

8.2 LIFO (Stack)

Die zentralen Operationen sind

- $push(x)$ – Element auf den Stack legen
- $x = top()$ – oberstes Element zurückliefern
- $pop()$ – oberstes Element entfernen
- $isEmpty()$ – Stack leer?
- $isFull()$ – Stack voll?

Um von den monolithischen Klassen wegzukommen, wird der Stack – seine Datenfelder und seine Methoden – in einer eigenen Klasse implementiert. Zum Test dieser Implementierung wird eine eigene Klasse erstellt – diese enthält die *main*-Methode.

Programm 8.1: Stack – primitive Implementierung (*PrimStack/PrimStack.java*)

```java
import IOulm.*;

public class PrimStack{
    static private final int size = 10; // Klassenfeld
    private int st [];                  // Instanzfeld
    private int top;                    // Instanzfeld

    public PrimStack() {                // Konstruktor
            st = new int[ size ]; top = 0;
    }
    public void print () {
     for(int i = 0; i < top; i++)
        Write.String(" " + st[i] + " -> ");
     Write.Ln(); return;
    }
    public boolean isEmpty() { return ( top == 0 ); }
    public boolean isFull () { return ( top >= size ); }
    public boolean push(int x) {
          if ( isFull () ) return false;
          st[top] = x; top++; return true;
    }
    public boolean pop() {
          if ( isEmpty() ) return false;
```

```
24            top--; return true;
25     }
26     public int top() {
27         // vorab: der Aufrufer muss sicherstellen, dass der
28         // Stack nicht leer ist !!!
29         if ( isEmpty() )
30             return Integer.MIN_VALUE; // keine gute Loesung !!!
31         return st[top-1];
32     }
33 }
```

Anmerkungen zu Programm 8.1 (S. 182):

- Zeile 5: Die Variable (genauer: Konstante) *size* wird nur ein einziges Mal angelegt, egal wie viele Instanzen der Klasse *PrimStack* auch immer erzeugt werden
- Zeilen 6,7: Die beiden Größen *st* und *top* sind nur innerhalb dieser Klasse sichtbar (Schlüsselwort **private**); beides sind Instanzfelder, d.h. jede erzeugt Instanz hat lokal bei sich diese Objekte
- Zeilen 9-12: Hier wird ein **Konstruktor** definiert, der beim Erzeugen einer Instanz aufgerufen wird und diese (ihre Felder) initialisiert. Ein Konstruktor ist immer zugänglich (**public**) und hat keinen Rückgabewert. Er heisst immer so wie die Klasse selbst. Verschiedene Konstruktoren unterscheiden sich dann in ihrer Signatur.
- Zeilen 13ff: Alle Methoden sind nun **Instanzmethoden**
- Zeilen 37ff: Das typische Dilemma: als potentieller Rückgabewert kommt jede Integer in Frage – wie soll man über den Rückgabewert einen aufgetretenen Fehler oder eine aufgetretene Ausnahmesituation signalisieren?

Programm 8.2 auf S. 8.2 zeigt ein Anwendung dieser Klasse. Beide Klassen liegen im selben Verzeichnis, unsere Variable *CLASSPATH* hat den Punkt-Eintrag, somit wird die Klasse *PrimStack* in *TestPrimStack* auch gefunden.

Programm 8.2: Stack – Test der primitive Implementierung (*PrimStack/TestPrimStack.java*)

```
import IOulm.*;

public class TestPrimStack {
  public static void main(String[] args) {
     PrimStack st = new PrimStack();
     char action =' '; int number = 0;
     while (true) {
        Write.String("add (a) / top (t) / remove (r) / print (p) / exit (e)? ");
        if ( Urc.readChar() ) {
            action = Urc.getChar();
            Urc.readString(); // Rest, insbesondere '\n' weglesen
            switch (action) {
              case 'a':
                   Write.String("Give number: ");
                   if (Urc.readInt()) {
                      number = Urc.getInt();
                      if ( st.push(number) ) Write.Line("done");
```

```
18                      else Write.Line("Overflow!");
19                    }
20                    break;
21               case 'r':
22                    if (st.pop()) Write.Line("done");
23                    else Write.Line("Empty!");
24                    break;
25                case 't':
26                    if ( st.isEmpty() ) Write.Line("Empty!");
27                    else Write.Line("got: " + st.top());
28                    break;
29                 case 'p':
30                      st.print();
31                      break;
32                 case 'e':
33                        return;
34                 default:
35                        Write.Line(" Illegal action");
36            }
37        } else return;
38      }
39   }
40 }
```

Anmerkungen zu Programm 8.2 (S. 183):

- Zeile 5: Hier wird eine Instanz *st* der Klasse *PrimStack* erzeugt und initialisiert
- Zeile 17ff: Der Aufruf der (Instanz-) Methoden erfolgt an der Instanzvariablen (**st.push()**)

Abstrakte Datentypen (ADTs):

- Der Stack ist das typische Beispiel für einen Datentyp, den man abstrakt, d.h. implementierungsunabhängig definieren kann – man nennt dies einen **Abstrakten Datentyp**, kurz **ADT**
- Der Datentyp wird ausschliesslich über seine Zugriffsoperationen und nicht über seine innere Struktur definiert. Letztere wird verborgen (**Information Hiding**): Diese Struktur kann geändert werden – solange die Zugriffsoperation in Syntax und Semantik gleich bleiben, müssen die Programme, die diesen Typ nutzen, nicht geändert werden!
- Die Definition kann in Analogie zu mathematischen Strukturen (Algebren) erfolgen:

types	$Stack, Object, Boolean$
operations	$new : \emptyset \longrightarrow Stack$
	$push : Stack \times Object \longrightarrow Stack$
	$pop : Stack \longrightarrow Stack$
	$top : Stack \longrightarrow Object$
	$isEmpty : Stack \longrightarrow Boolean$
var	$s : Stack; o : Object$
preconditions	$pop(s) : NOT\ \ isEmpty(s)$
	$top(s) : NOT\ \ isEmpty(s)$
axioms	$isEmpty(new()) = true$
	$isEmpty(push(s,o)) = false$
	$pop(push(s,o)) = s$
	$top(push(s,o)) = o$

Diese Definition geht von einem unbeschränkten Stack aus. Dies ist auch bei großen Speicherkapazitäten unrealistisch:

types	$FStack, Object, Boolean$
operations	$new : \emptyset \rightarrow FStack$
	$push : FStack \times Object \rightarrow FStack$
	$pop : FStack \rightarrow FStack$
	$top : FStack \rightarrow Object$
	$isEmpty : FStack \rightarrow Boolean$
	$isFull : FStack \rightarrow Boolean$
var	$s : FStack; o : Object$
preconditions	$pop(s) : NOT\ \ isEmpty(s)$
	$top(s) : NOT\ \ isEmpty(s)$
	$push(s, o) : NOT\ \ isFull(s)$
axioms	$isEmpty(new()) = true$
	$isEmpty(push(s, o)) = false$
	$pop(push(s, o)) = s$
	$top(push(s, o)) = o$

- Die konkrete Umsetzung dieser Definitionen in Java kann über ein **Interface** erfolgen; dieses kann dann verschiedene Implementierung besitzen. Da sich in Java alle Klassen und damit alle Objekte von einer Basisklasse namens **Object** ableiten, muss bei der Implementierung eines konkreten Stacks der Typ nicht festgelegt werden – dies macht erst die Anwendung!

- Schwierigkeiten macht zunächst die Umsetzung der Vorbedingungen (*preconditions*); eine Lösung bietet Java in Form von sog. **Exceptions** an.

Programm 8.3: Interface für den ADT Stack (*STACK/ADTStack.java*)

```java
import java.util.EmptyStackException;
public interface ADTStack {
    public boolean isEmpty();
    public boolean isFull();
    public Object top()
        throws EmptyStackException;
    public void push(Object element)
        throws MyStackOverflowException;
    public void pop()
        throws EmptyStackException;
}
```

Um vom *Exception Handling* an dieser Stelle wegzukommen, soll eine Vereinfachung vorgenommen werden: Programm 8.4, S. 185

Programm 8.4: Vereinfachtes Interface für den ADT Stack (*STACK/ADTSimpleStack.java*)

```java
public interface ADTSimpleStack {
    public boolean isEmpty();
    public boolean isFull();
    public Object top();
    public boolean push(Object element);
    public boolean pop();
}
```

Eine mögliche Implementierung zeigt Programm 8.5, S. 186.

Programm 8.5: Implementierung zum Stack-Interface (*STACK/SimpleStack.java*)

```
public class SimpleStack implements ADTSimpleStack {
    static private final int size = 10;
    private Object st [];
    private int top ;

    public SimpleStack () {
        st = new Object[ size ]; top = 0;
    }
    public boolean isEmpty() { return ( top == 0 ); }
    public boolean isFull () { return ( top >= size ); }

    public boolean push(Object element ) {
        if ( isFull () ) return false ;
        st [ top++] = element ; return true ;
    }
      public boolean pop() {
          if ( isEmpty() ) return false ;
          top--; return true ;
    }
    public Object top () {
        if ( isEmpty () ) return null ;
        return st [ top-1];
    }
}
```

Eine einfache Anwendung zeigt 8.6, S. 186.

Programm 8.6: Anwendung der Implementierung zum Stack-Interface (*STACK/TestSimpleStack.java*)

```
import IOulm.*;

public class TestSimpleStack {
    public static void main(String args[ ]) {
        ADTSimpleStack st = new SimpleStack ();
        if ( st .push(new Integer (10)) ) Write . Line ("push(10)" );
        else return ;
        if ( st .push(new Character ('a')) ) Write . Line ("push('a') " );
        else return ;
        if ( st .push(new String("Zeichenfolge")) ) Write . Line ("push(\"Zeichenfolge\")" );
        else return ;
        Write . Line ("top: " + st . top ());
        Write . Line ("pop:" ); st .pop ();
        Write . Line ("top: " + st . top ());
        Write . Line ("pop:" ); st .pop ();
        Write . Line ("top: " + st . top ());
        Write . Line ("pop:" ); st .pop ();
    }
}
```

Anmerkungen zu 8.6, S. 186:

- Zeile 5: Hier wird der Konstruktor unseres Stack aufgerufen – die Variable *st* vom Typ *SimpleStack* hat als Wert eine **Referenz** auf eine Instanz der Klasse *SimpleStack* – diese wird von *new* und dem Aufruf des Konstruktor dieser Klasse erzeugt!
- Zeile 6: Die **Instanzmethode** *push*() wird an der Instanz *st* aufgerufen – diese erwartet als Argument eine Referenz auf ein Objekt vom Typ *Object* (siehe Programm 8.5, S. 186, bzw. die Schnittstelle in 8.4, S. 185) – oder eine typverträgliche Referenz

  Alle Klassen in Java liegen in einer gemeinsamen Hierarchie (**Klassenhierarchie**):

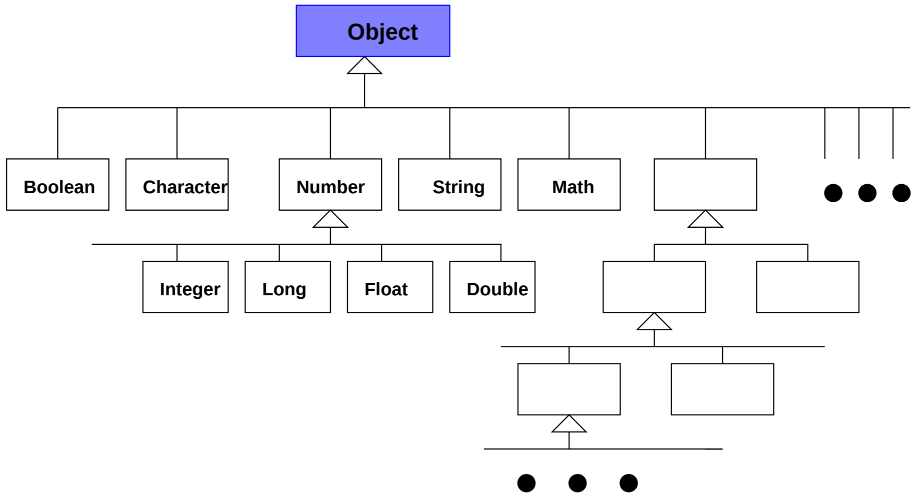
  

  Abbildung 8.3: Klassenhierarchie (Ausschnitt)

  Die tieferliegenden Klassen erhalten mit dem sehr grundlegenden Konzept der **Vererbung** Eigenschaften der darüberliegenden Klasse(n) – insbesondere sind die Referenzen typverträglich, also sind Referenzen auf Instanzen der Klassen *Integer* (Zeile 6), *Character* (Zeile 8) und *String* (Zeile 10) verträglich mit dem formalen Parameter (Referenz auf *Object*)

  Mehr dazu im Zusammenhang mit Klassenerweiterung / Vererbung!

- Zeile 6: *st.push*() erhält eine Referenz auf eine Instanz von *Integer*, die mit dem Wert 10 initialisiert wurde
- Zeile 8: *st.push*() erhält eine Referenz auf eine Instanz von *Character*, die mit dem Zeichen *a* initialisiert wurde
- Zeile 10: *st.push*() erhält eine Referenz auf eine Instanz von *String*, die mit dem Wert *Zeichenfolge* initialisiert wurde
- Die Klassen *Integer*, *Character*, *String* (u.v.a.) erben eine Methode *toString*(), die in der jeweiligen Klasse "sinnvoll" überschrieben wird (mehr dazu später) – *st.top*() liefert in den Zeilen 12,14,16 eine Referenz zurück, implizit wird darauf die Methode *toString*() aufgerufen.
- Einen Stack mit Elementen unterschiedlichen Typs zu unterhalten, macht meist wenig Sinn – Programm 8.6, S. 186 soll hier kein Vorbild sein!

8.3 FIFO

FIFO – *First in First Out*, meist mit **Queue** bezeichnet

Beispiele: Pipes in Unix, Warteschlangen (Drucker-Queue), . . .

Die zentralen Operationen sind:

- $add()$ – am Ende Element hinzufügen
- $rm()$ – am Anfang Eelement entfernen
- $isEmpty()$ – Queue leer?
- $isFull()$ – Queue voll?

Auch hier lassen sich leicht Preconditions und Axiome finden, die die Semantik genau definieren!

Wie vorher beim Stack lässt sich auf dies in einem Interface festhalten (Programm 8.7, S. 8.7).

Programm 8.7: Einfaches Interface zu einer Queue (*QUEUE/ADTSimpleQueue.java*)

```java
public interface ADTSimpleQueue {
    public boolean isEmpty ();
    public boolean isFull ();
    public Object remove ();
    public boolean add( Object element );
}
```

Für eine Realisierung mit den bislang bekannten Mitteln bietet sich ein Array mit Dimension *size* an.

Abbildung 8.4 (S. 188) zeigt eine erste Variante: der Bedienschalter ist fest auf Indexposition 0, das Ende (sprich: der nächste freie Platz) wird über einen Index *tail* geführt.

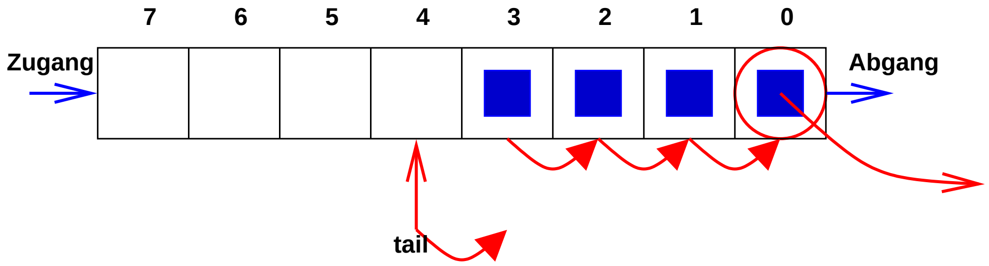


Abbildung 8.4: Array als Queue – Variante 1

Die Operationen (mit $int[]\ queue\ =\ new\ int[size]$):

$$
\begin{array}{ll}
isEmpty(queue): & return(tail\ \ ==\ \ 0)\\
isFull(queue): & return(tail\ \ >= size)\\
add(queue, x): & queue[tail] = x, tail++\\
x = remove(queue) & x = queue[0];\\
 & for(i = 0; i < tail; i++)\ queue[i] = queue[i+1]\\
 & tail--;\\
 & return\ \ x
\end{array}
$$

Die Operation *remove*() ist sehr aufwändig!

Eine Alternative ergibt sich dadurch, dass man nicht die Elemente in der Queue verschiebt, sondern den “Bedienschalter” – dazu wird das Array “kreisförmig geschlossen” (siehe dazu Abbildung 8.5, S. 189).

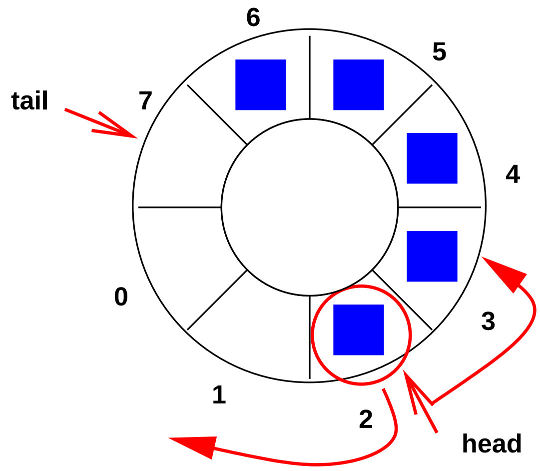


Abbildung 8.5: Zyklisches Array als Queue – Variante 2

Zur einfacheren Verwaltung wird zusätzlich eine Varibale *count* geführt, in der die Zahl der Elemente in der Queue festgehalten wird.

Die Operationen jetzt:

$$\begin{array}{ll} isEmpty(queue): & return(count == 0); \\ isFull(queue): & return(count == size); \\ add(queue, x): & queue[tail] = x; tail = (tail+1)\%size; count++; \\ x = remove(queue): & x = queue[head]; head = (head+1)\%size; count--; return\ x; \end{array}$$

Programm 8.8 (S. 190) zeigt eine einfache Implementierung, ein kleines Testprogramm ist Programm 8.9 (S. 190)

Programm 8.8: Queue als zyklisches Array (*QUEUE/SimpleQueue.java*)

```java
public class SimpleQueue implements ADTSimpleQueue {
    static private final int size = 10;
    private Object queue [];
    private int head, tail, count;
    public SimpleQueue() {
        queue = new Object[ size ]; tail = head = count = 0;
    }
    public boolean isEmpty() { return ( count == 0 ); }
    public boolean isFull () { return ( count >= size ); }
    public boolean add(Object element) {
        if ( isFull () ) return false;
        queue[ tail ] = element; tail = ( tail + 1) % size; count++; return true;
    }
    public Object remove() {
        Object x;
        if ( isEmpty() ) return null;
        x = queue[head]; head = ( head + 1) % size; count--; return x;
    }
}
```

Programm 8.9: Anwendung von Programm 8.8, S. 190 (*QUEUE/TestSimpleQueue.java*)

```java
import IOulm.*;
public class TestSimpleQueue{
  public static void main(String [] args) {
    ADTSimpleQueue q = new SimpleQueue();
    char action =' '; int number = 0;
    while (true) {
      Write. String ("add (a) / remove (r) / exit (e)? ");
      if ( Urc.readChar () ) {
          action = Urc.getChar ();
          Urc. readString (); // Rest, insbesondere '\n' weglesen
          switch ( action ) {
            case 'a':
                Write. String ("Give number: ");
                if (Urc. readInt ()){
                  number = Urc. getInt ();
                  if ( q.add(new Integer (number)) ) Write. Line ("done");
                  else Write. Line ("Overflow!");
                }
                break;
            case 'r':
                if ( q.isEmpty() ) Write. Line ("Empty!");
                else Write. Line ("got: " + q.remove ());
                break;
            case 'e': return;
            default: Write. Line (" Illegal action");
          }
      } else return;
    }
  }
}
```

Kapitel 9

Algorithmen zum Suchen und Sortieren

9.1 Vorbemerkungen

Die im Folgenden vorgenommene Beschränkung auf das Sortieren von / Suchen in Zahlenfolgen ändert nichts an den Algorithmen. Soll etwas anderes (Texte, Persondendaten) sortiert / gesucht werden, so müssen entsprechende Größer-/Kleiner-/Gleichheits-Vergleiche (als Funktionen) bereitgestellt werden, das Umkopieren kann komplizierter werden. Diese Algorithmen sollen Konstruktions- und Vorgehensweisen erläutern; in allen Programmiersprachen gibt es in den Bibliotheken ausgereifte Implementierungen, das Unix-Kommando `sort` leistet hervorragende Dienste!

Sortierproblem:

Sei $n \in \mathbb{N}$ und $a = (a_0, a_1, \ldots, a_{n-1})$ eine Folge mit $a_i \in \mathbb{N}, \forall i \in I = \{0, 1, \ldots, n-1\}$
Gesucht ist eine bijektive Abbildung (Permutation) $\varphi : I \to I$, so dass gilt:

$$a_{\varphi(i)} \leq a_{\varphi(j)}, \forall i, j \in I \; mit \; i < j$$

Beispiel:

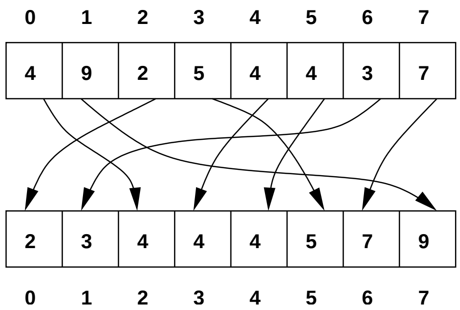


Abbildung 9.1: Sortieren

$$a_{\varphi(0)} = 2, a_{\varphi(1)} = 7, a_{\varphi(2)} = 0, a_{\varphi(3)} = 5, a_{\varphi(4)} = 3, a_{\varphi(5)} = 4, a_{\varphi(6)} = 1, a_{\varphi(7)} = 6$$

Suchproblem:

Gegeben sie obige Folge a und ein x aus dem Wertebereich der Feldelemente. Gesucht ist $i \in I$ mit $x = a_i$

9.2 Suchen

9.2.1 Lineares Suchen

Ist das Feld, in dem zu suchen ist, unsortiert, geht man von Anfang her durch das Feld und vergleicht Element für Element mit dem gesuchten Wert.

Programm 9.1: Lineare Suche (*LinearSearch.java*)

```java
public class LinearSearch {

   public static int linearSearch (int a[], int x) {
      int i;
      int n = a.length;
      for(i = 0; i < n; i++)
         if ( a[i] == x ) return i;
      return -1; // not found
   }
}
```

Programm 9.2: Test der linearen Suche (*TestLinearSearch.java*)

```java
import IOulm.*;

public class TestLinearSearch {

    static int a[] = { 4,9,2,5,4,4,3,7 };
    static int x;
   public static void main(String[] args) {

      Write.String("Give element to look for: ");
      if ( Urc.readInt() ) x = Urc.getInt(); else return ;

      x = LinearSearch.linearSearch(a,x);

      if ( x < 0 ) Write.Line("not found");
      else Write.Line("Found at index " + x );
   }
}
```

Bei einem Feld der Länge n werden im besten Fall 1 Vergleich, im schlechtesten Fall n und im Mittel $n/2$ Vergleiche benötigt, die Komplexität kann mit $O(n)$[1] angegeben werden.

[1] der Algorithmus benötigt höchstens linear viele Schritte

9.2.2 Binäre Suche

Ist das Feld, in dem man sucht bereits (aufsteigend) sortiert, so kann man wie "in einem Lexikon" suchen:

Sei $I' = I = \{0, 1, \ldots, n-1\}$

1. Wenn $I' = \{\}$, so **fertig, nicht enthalten**
2. Sei m das mittlere Element von I';
   - wenn $x = a_m$, so **fertig: der gesuchte Index ist** m
   - wenn $x < a_m$, setze $I' = \{i \in I' | i < m\}$ und mache weiter bei 1;
   - wenn $x > a_m$, setze $I' = \{i \in I' | i > m\}$ und mache weiter bei 1;

Grafisch lässt sich das Verfahren wie in Abb. 9.2 (S. 193) dargestellt beschreiben. Wie mal leicht sieht benötigt das Verfahren bei n Elementen im schlimmsten Fall die komplette Suche in dem dargestellten Graphen, die Komplexität beträgt $O(log_2(n))$!

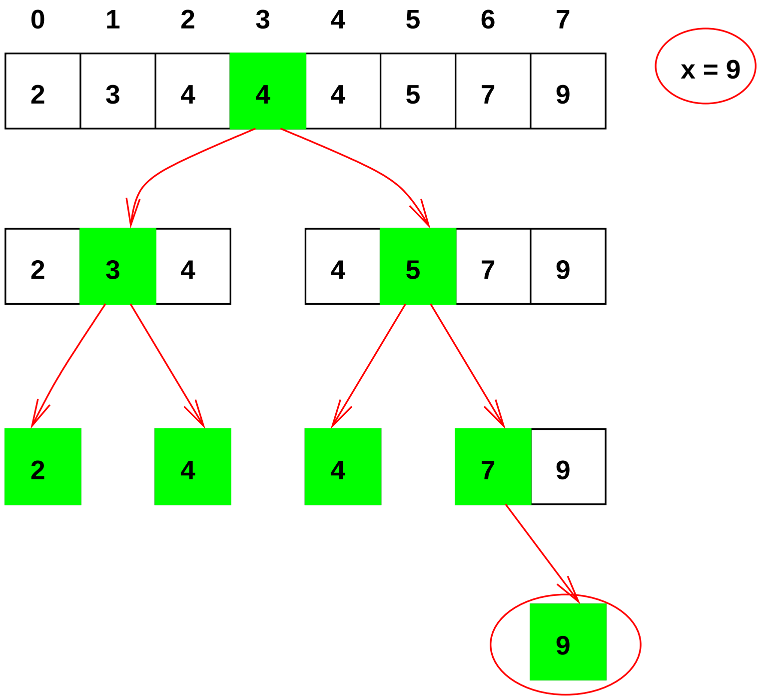


Abbildung 9.2: Binäres Suchen – grafisch dargestellt

Die Implementierung kann sowohl iterativ wie auch rekursiv serfolgen (siehe Programm 9.3, S. 194.

Programm 9.3: Binäre Suche – Iterativ und Rekursiv (*BinarySearch.java*)

```java
public class BinarySearch {

   public static int binarySearchIterativ (int a[], int x) {
      int i,j,m,n;
      n = a.length; i = 0; j = n - 1;
      while (i <= j) {
            m = (i + j) / 2;
            if (a[m] == x) return m;
            if (x < a[m]) j = m-1;
            else /* x > a[am] */ i = m+1;
      }
      return -1; // not found
   }

   public static int binarySearchRecursive (int a[], int x) {
         return recSearch(a,x,0,a.length-1);
   }
   private static int recSearch(int a[], int x, int i, int j) {
         int m;
         if ( i > j ) return -1;
         m = ( i + j ) / 2;
         if ( a[m] == x ) return m;
         if ( x < a[m] ) return recSearch(a,x,i,m-1);
         else return recSearch(a,x,m+1,j);
   }
}
```

Programm 9.4: Testprogramm für Programm 9.3, S. 194 (*TestBinarySearch.java*)

```java
import IOulm.*;

public class TestBinarySearch {

    static int a[] = { 2,3,4,4,4,5,7,9 };
    static int x;
   public static void main(String[] args) {
      Write.String("Give element to look for: ");
      if ( Urc.readInt() ) x = Urc.getInt(); else return ;
      x = BinarySearch.binarySearchIterativ (a,x);
      if ( x < 0 ) Write.Line("not found");
      else Write.Line("Found at index " + x );

      Write.String("Give element to look for: ");
      if ( Urc.readInt() ) x = Urc.getInt(); else return ;
      x = BinarySearch.binarySearchRecursive (a,x);
      if ( x < 0 ) Write.Line("not found");
      else Write.Line("Found at index " + x );
   }
}
```

9.3 Insertion Sort

In jedem Schritt i ($i = 1, 2, \ldots, n-1$) des Verfahrens besteht das Array a aus zwei Teilen:

- $a_0, \ldots, a_{i-1}$ ist bereits aufsteigend sortiert
- $a_i, \ldots, a_{n-1}$ ist noch unsortiert

Wir merken uns in einer Variablen t das Element a_i. Um dieses Element in den bereits sortierten Teil einzufügen, verschieben wir die Element $a_{i-1}, a_{i-2}, \ldots$ um je eine Position nach links, bis wir den richtigen Platz für das gemerkte Element t gefunden haben.

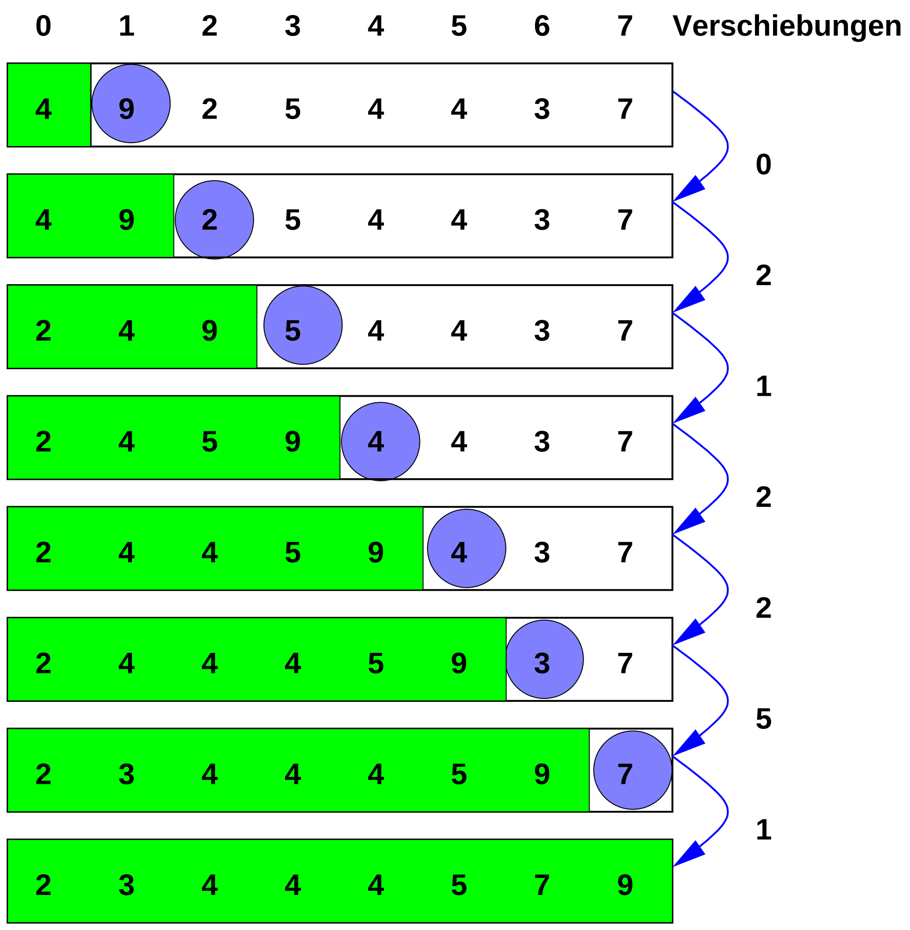


Abbildung 9.3: Insertion Sort

Programm 9.5 (S. 196) zeigt eine Implementierung dieses Verfahrens als Klassenmethode (Schlüsselwort: *static*).

Programm 9.5: Insertion Sort (*InsertionSort.java*)

```java
public class InsertionSort {

  public static void insertionSort (int a[]) {
    int i,j,t;
    int n = a.length;
    for(i = 1; i < n; i++) {
      j = i; t = a[j];
      while ( j > 0 && a[j-1] > t ) {
        a[j] = a[j-1];
        j--;
      }
      a[j] = t;
    }
  }
}
```

Programm 9.6: Test von InsertionSort (*TestInsertionSort.java*)

```java
import IOulm.*;

public class TestInsertionSort {

   static int a[] = { 4,9,2,5,4,4,3,7 };
   static int n = a.length;

  public static void printArray() {
   for(int i = 0; i < n; i++) {
      Write.String(" " + a[i]);
   }
   Write.Ln();
   return;
  }

 public static void main(String[] args) {

      InsertionSort.insertionSort(a);
      printArray();
      return;

 }
}
```

Analyse

- Im schlechtesten Fall (*worst case*) muss das neu einzufügende Element immer am linken Ende eingefügt werden: beim i-ten Schritt sind dies i Vergleiche und i Verschiebe-Operationen (Kopiervorgänge), mit $i = 1, 2, 3, \ldots, n-1$. Insgesamt sind dies also $(n-1) \cdot n/2$ Operationen, wir haben also einen Algorithmus mit einer Komplexität von $O(n^2)$.
- Dieser Fall tritt ein, wenn das Feld bereits absteigend sortiert ist!
- Die genaue Anzahl der Schritt kann über die sog. **Inversionszahl** bestimmt:

  Für eine Folge $a = (a_0, a_1, \ldots, a_{n-1})$ ist die Inversionszahl z_a wie folgt definiert:

  $$z_a = |\{ (i,j) \mid i < j \wedge a_i > a_j \}|$$

  Betrachtet man die Größe

  $$z_a(j) = |\{ i \mid i < j \wedge a_i > a_j \}|,$$

  so erhält man für jedes Element im j-ten Schritt (in Abb. 9.3, S. 195, eingekreist), die Anzahl der Verschiebungen und Vergleiche im sortierten linken Feld. Addiert man diese Zahlen auf, so erhält man z_a!

- Im Beispiel ist

  $$z_a = |\{(0,2),(0,6),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(3,4),(3,5),(3,6),(4,6),(5,6)\}| = 13$$

  bzw.

$z_a(0) = 0$	$z_a(1) = 0$	$z_a(2) = 2$	$z_a(3) = 1$	$z_a(4) = 2$	$z_a(5) = 2$	$z_a(6) = 5$	$z_a(7) = 1$

9.4 Quicksort

- eines der schnellsten und auch einfachsten Sortierverfahren
- C.A.R. Hoare: *Quicksort*. Computer Journal, Vol. 5, 1, 10-15 (1962)
- rekursiv nach dem Prinzip *divide et impera* (*divide and conquer*, teile und beherrsche):

1)	***Divide***	Problem wird in (kleinere) Teilprobleme zerlegt
2)	***Conquer***	Teilprobleme werden gelöst
3)	***Combine***	Teilproblemlösungen werden zur Gesamtlösung zusammengefügt

  Funktioniert immer dann, wenn Zusammenfügen der Teilproblemlösung die Gesamtlösung ergibt!

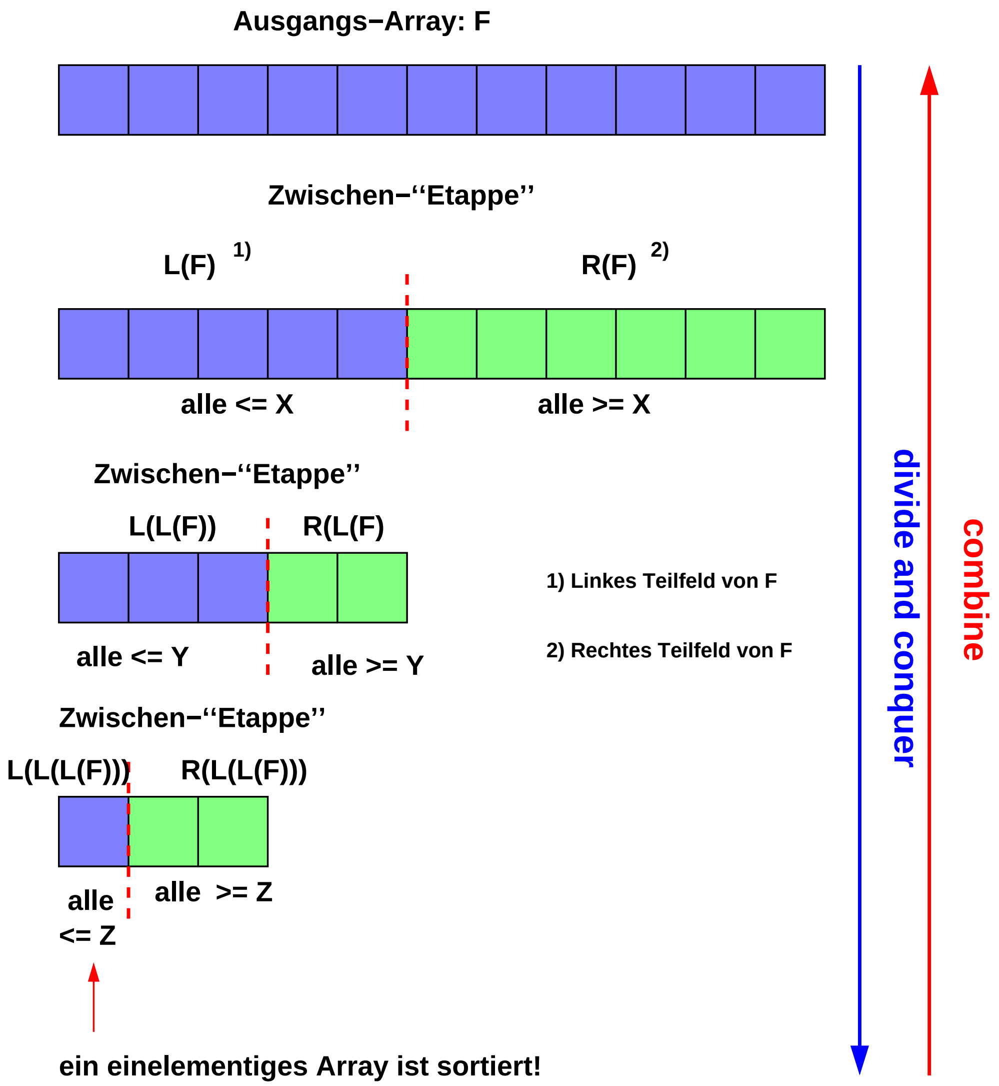


Abbildung 9.4: Quicksort – Prinzip

Partitionierungs-Algorithmus für ein Feld $int[]a = new\ \ int[n];$:

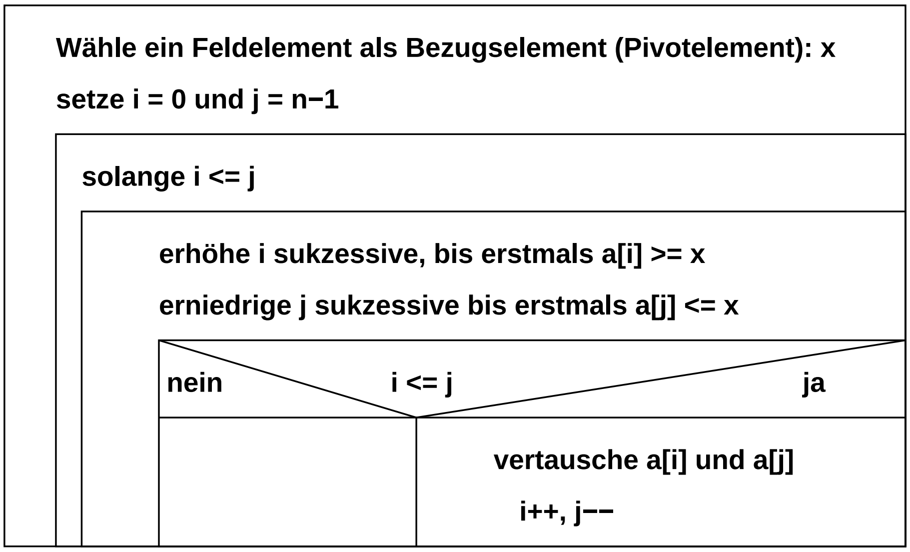


Abbildung 9.5: Quicksort – Partitionierung

Die Rekursionstiefe hängt natürlich ganz entscheidend von der Wahl des Pivotelementes ab. Im günstigsten Fall erfolgt immer einer Halbierung des zu sortierenden Teilfeldes – die Rekursionstiefe ist somit $log_2(n)$ (siehe Abb. 9.6 a), S. 200); in jeder Schicht sind n Elemente zu behandeln, d.h. der Quicksort benötigt $\Theta(n \cdot log_2(n))$ Schritte[2].

Im ungünstigsten Fall wird immer ein einelementiges Feld abgespalten (siehe Abb. 9.6 b), S. 200 – die Rekursionstiefe ist $n - 1$, der Quicksort benötigt dann $\Theta(n^2)$ Schritte.

Im Mittel wird eine Partitionierung wie in Abb. 9.6 c) (S. 200) dargestellt zustandekommen.

Wahl des Pivotelementes:

- Ein beliebiges Feldelement, z.B. das erste, das mittlere oder das letzte – ist das Feld bereits aufsteigend (absteigend) sortiert, so führt die die Wahl des ersten (letzten) Elements zum schlechtesten Fall!
- Der Median (das Element, das bei aufsteigender Sortierung der Elemente an Position $\frac{n-1}{2}$ steht[3] – in diesem Fall erhält man die optimale Partitionierung[4] .
- Der Median von drei beliebigen Elementen, z.B. des ersten, des mittleren und des letzten

Man kann zeigen, dass die Zeitkomplexität im Mittel ebenfalls $\Theta(n \cdot log_2(n))$ beträgt!

[2] $\Theta(n^2)$ bedeutet, dass der Algorithmus mindestens, aber auch höchstens proportional zu n^2 Schritte benötigt
[3] bei geradem n wird als Median oft der Mittelwert der Elemente an Position $\frac{n-1}{2}$ und $\frac{n-1}{2} + 1$ genommen
[4] Die Bestimmung des Median kann in linearer Zeit erfolgen!

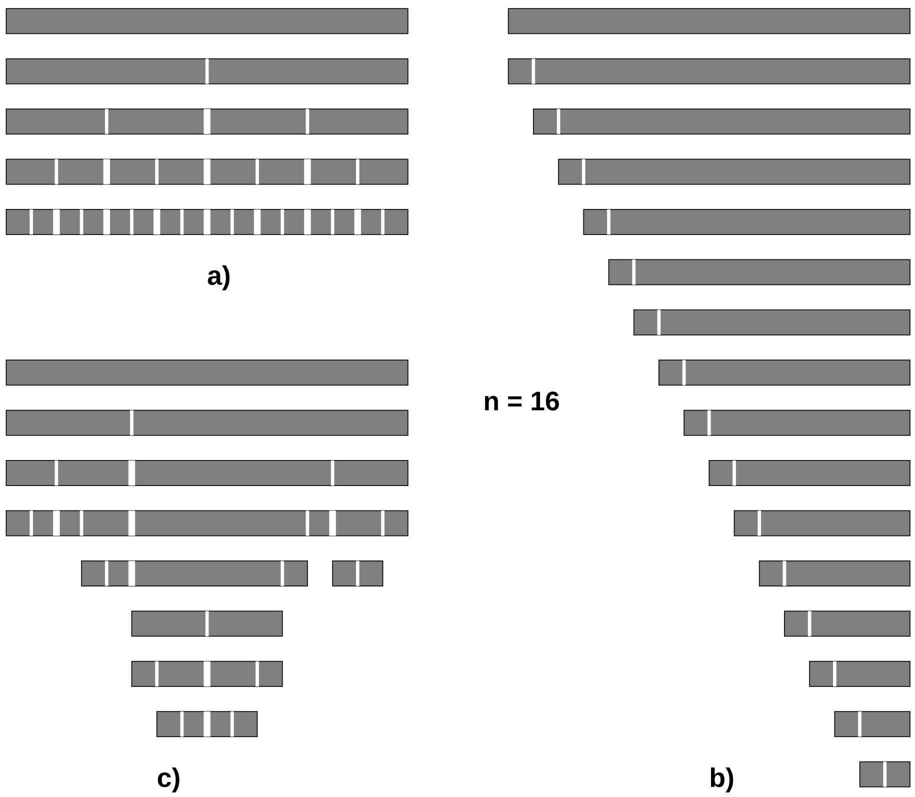


Abbildung 9.6: Quicksort – Rekursionstiefe

Programm 9.7: Quicksort – rekursiv (*Quicksort.java*)

```java
public class Quicksort {

  private static int a[];
  public static void quickSort(int[] a0) {
    a = a0;
    if ( a.length == 0 ) return;
    qsort(0, a.length-1);
  }

  private static void qsort(int low, int high) {
    int i = low, j = high;
    int ref = a[ (low + high) / 2 ];

    // partitioning
    while ( i <= j ) {
      while ( a[i] < ref ) i++;
      while ( a[j] > ref ) j--;
      if ( i <= j ) { swap(i,j); i++; j--;}
    }
    Write.Ln();

    // recursion
    if ( low < j ) qsort(low,j);
```

```
      if ( i < high ) qsort (i ,high );
   }

   private static void swap(int i , int j ) {
      int tmp = a[ j ]; a[ j ] = a[ i ]; a[ i ] = tmp;
   }
}
```

Programm 9.8: Quicksort – Testprogramm (*TestQuicksort.java*)

```
import IOulm.∗;

public class TestQuicksort {

   static int [] a = { 4,9,2,5,4,4,3,7 };
   static int n = a.length ;

   public static void printArray () {
    for(int i = 0; i < n; i++) {
       Write.String ("␣" + a[ i ]);
    }
    Write.Ln();
    return;
   }

  public static void main(String [] args ) {

       Quicksort . quickSort (a );
       printArray ();
       return;

  }
}
```

Der Quicksort-Algorithmus läßt sich auch relativ leicht iterativ mit eigener Stackverwaltung implementieren. Der jeweils zu sortierende Feldausschnitt wird über die beiden begrenzenden Indizes bestimmt. Zu Beginn wird das gesamt Feld auf den Stack gelegt (jeweils zwei Positionen: die erste ist der linke, die zweite der rechte Index). Der Algorithmus holt sich immer den obersten Feldausschnitt von Stack, partioniert diesen in zwei Teile, legt den rechten Teil auf den Stack (für die spätere Bearbeitung) und macht mit der Partitionierung mit dem linken Teil weiter.

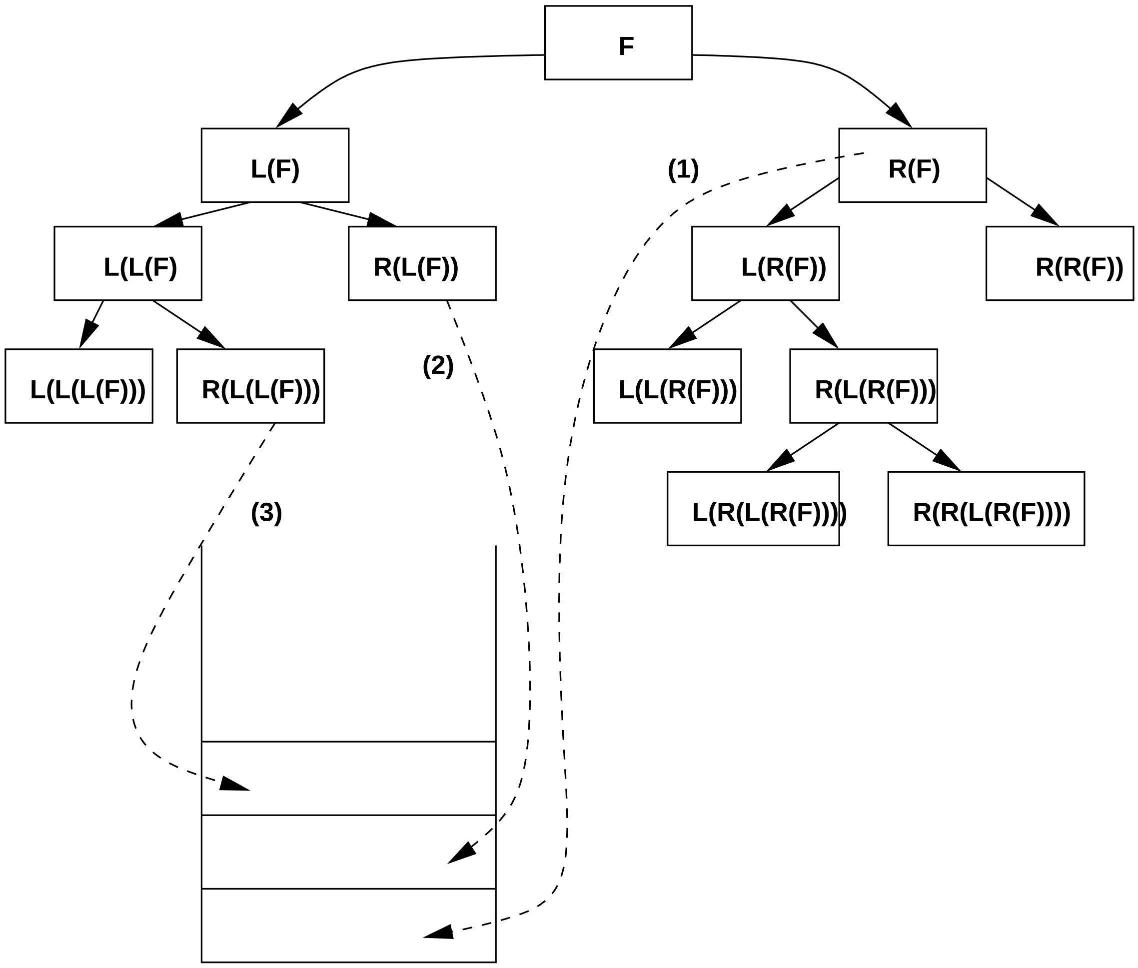


Abbildung 9.7: Quicksort – Iterativ

Programm 9.9: Quicksort – Iterativ (*QuicksortIterativ.java*)

```
public class QuicksortIterativ {

  private static int a[];
  private static int depth = 256; // pairs of indices
  private static int[] stack = new int[depth];
  private static int top = 0; // next free

  public static void quickSort(int a0[]) {
    int i,j, left, right, ref,tmp;
    a = a0;
    if ( a.length == 0 ) return;
    stack[top] = 0; stack[++top] = a.length-1; top++;
    do {
     // get partition from top of stack
     right = stack[--top];
     left = stack[--top];
     do { // partition of a[ left ] .. a[ right ]
        ref = a[ ( left + right ) / 2];
        i = left ; j = right ;
        do {
           while ( a[i] < ref ) i++;
           while ( a[j] > ref ) j--;
           if ( i <= j ) {
             swap(i,j); i++; j--;
           }
        } while (i <= j);
        if ( i < right ) { // stacking boundaries of right part
           stack[top] = i; stack[++top] = right; top++;
           if ( top >= depth-1 ) return;
        }
        right = j;
     } while ( left < right );
    } while (top > 0);
  }

  private static void swap(int i, int j) {
    int tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
  }
}
```

9.5 Heapsort

J.W.J. Williams: *Algorithm 232: Heapsort*. Communications of the ACM, Vol. 7,6, 347-348 (1964)

9.5.1 Grundlagen

Graphen

Definition: Ein **(gerichteter) Graph** ist ein Paar $G = (V, E)$ mit: V ist eine endliche, nichtleere Menge von Knoten (*Vertices*) und $E \subseteq V \times V$ eine Relation auf V, die Menge der Kanten (*Edges*).

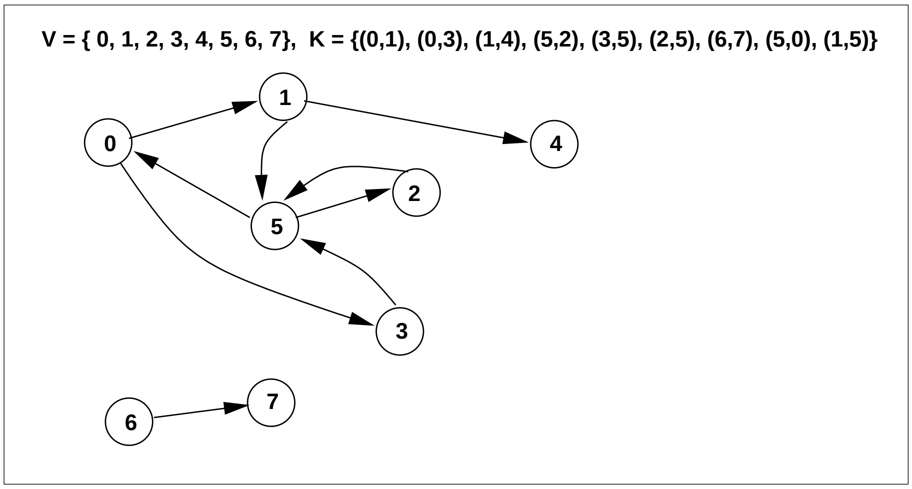


Abbildung 9.8: Ein Graph als Relation und in grafischer Darstellung

Sei $G = (V, E)$ ein Graph und $u \in V$:

Definition: Ein Knoten $v \in V$ heißt **direkt erreichbar**, wenn $(u, v) \in E$; v heisst dann **direkter Nachfolger** von u, u heisst **direkter Vorgänger** von v

Definition:

Ausgangsgrad $o(u) := |\{v \in V | (u, v) \in E\}|$

Eingangsgrad $i(u) := |\{v \in V | (v, u) \in E\}|$

Definition:

Ein **Pfad** in einem Graphen ist eine endliche Folge von Kanten

$p = (u_0, v_0), (u_1, v_1), \ldots (u_{n-1}, v_{n-1})$ mit $n \in \mathbb{N}_0$ und $u_i = v_{i-1}$, $\forall i \in \{1, \ldots n-1\}$

n ist die Länge des Pfades – $n = 0$ ist der leere Pfad (Länge 0)

u_0 ist der Anfangsknoten, v_{n-1} der Endknoten von p, alle anderen Knoten heißen innere Knoten

Sei $p = (u_0, v_0), (u_1, v_1), \ldots (u_{n-1}, v_{n-1})$ ein Pfad in einem Graphen G:

Definition:

p heißt **einfach**, wenn alle Kanten paarweise verschieden sind.

p heißt **Zyklus**, wenn $v_{n-1} = u_0$

p heißt **Weg**, wenn gilt: $\forall i, j \in \{0, \ldots, n-1\}, i \neq j$ ist $u_i \neq u_j, v_i \neq v_j$

p heißt **Kreis**, wenn er ein Zyklus und ein Weg ist.

Bäume

Definition: Ein Graph $T = (V, E)$ heißt **Baum** (**Tree**), wenn gilt:

1. T enthält keine Zyklen
2. Es gibt genau einen Knoten mit Eingangsgrad 0 – dieser ist die **Wurzel** des Baumes
3. Alle anderen Knoten haben den Eingangsgrad 1

Es gilt: In einem Baum ist jeder Pfad ein Weg!

Sei $T = (V, E)$ ein Baum mit Wurzel $r \in V$.

Definition:

Ein Knoten $w \in V$ heißt **Nachfolger** von $u \in V$, wenn es einen nichtleeren Pfad von u nach v gibt!

Ein Knoten $b \in V$ heißt **Blatt**, wenn $o(b) = 0$. Alle anderen Knoten heißen **innere Knoten**.

Die **Höhe** $h(v)$ **eine Knotens** $v \in V$ ist die Länge des Weges von der Wurzel r zu v.

Die **Höhe** $h(T)$ **eines Baumes** ist definiert als $max\{h(v)|v \in T - \{r\}\}$

Bäume werden in der Informatik unter anderem verwendet zur Organisation großer Datenmengen (Bsp: Dateisysteme) bzw. zum schnellen Zugriff auf große Datenmengen.

Definition: Ein Baum T heißt **Binärbaum**, wenn jeder Knoten einen Ausgangsgrad $\leq$ 2 hat

Abb. 9.9 (S. 206) zeigt einen Baum mit 11 Knoten – in dieser Art der Darstellung (schichtenweise von oben nach unten) kann man auf die Pfeile verzichten. Statt von „Höhe“ spricht man manchmal auch von „Tiefe“.

Definition: Ein Binärbaum T heißt **vollständig**, wenn

1. jedes Blatt die Höhe $h(T)$ oder $h(T) - 1$ hat,
2. jeder Knoten mit Ausgangsgrad 1 die Höhe $h(T) - 1$ hat.

Es gilt: Ein vollständiger Binärbaum mit n Knoten hat die Höhe $int(log_2(n))$!

Der Baum in Abb. 9.9 (S. 206) ist vollständig!

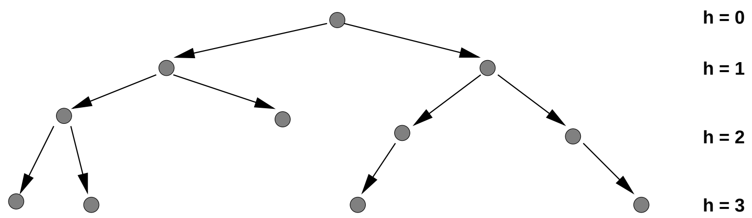


Abbildung 9.9: Ein Binärbaum mit 11 Knoten

Heapsort verwendet eine besondere Datenstruktur, die als **Heap**[5] bezeichnet wird.

Sei $T = (V, E)$ ein vollständiger Binärbaum mit einer Knotenmarkierung $\alpha : V \to M$, die jedem Knoten $u \in V$ eine Markierung $\alpha(u)$ aus einer geordneten Menge $(M, \leq)$ zuordnet.

Definition:

Ein Knoten $u \in V$ hat die **Heap-Eigenschaft**, wenn gilt:

$\forall v \in V : (u, v) \in E \Rightarrow \alpha(u) \geq \alpha(v)$

T heißt **Heap**, wenn alle Knoten die Heap-Eigenschaft haben, d.h. es gilt: $\forall (u, v) \in E : \alpha(u) \geq \alpha(v)$

T heißt **Semi-Heap**, wenn alle Knoten außer der Wurzel die Heap-Eigenschaft haben.

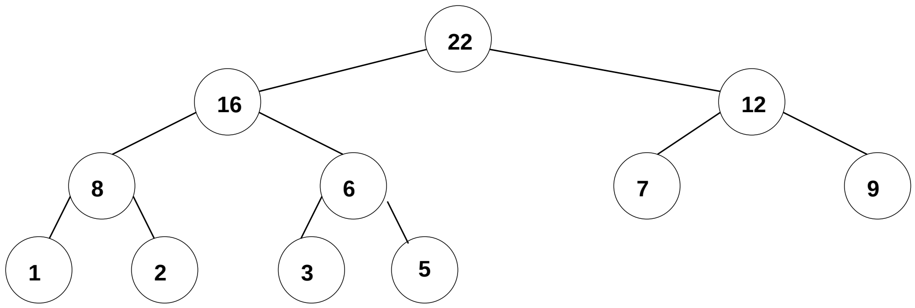


Abbildung 9.10: Heap mit 11 Knoten, $M = \mathbb{N}$

[5] hat nichts mit dem Speicherbereich zu tun, in dem dynamisch allozierte Objekte abgelegt werden

Was hat ein Array mit einem Baum zu tun?

Gegeben sei ein Array mit n Elementen; die Wurzel steht an Indexposition 0, die beiden Nachfolger des Knotens zur Indexposition i stehen an den Positionen $2 \cdot i + 1$ und $2 \cdot i + 2$. Alle Knoten an den Positionen $0, \ldots n \div 2 - 1$ sind innere Knoten, die an den Positionen $n \div 2, \ldots, n - 1$ sind Blätter.

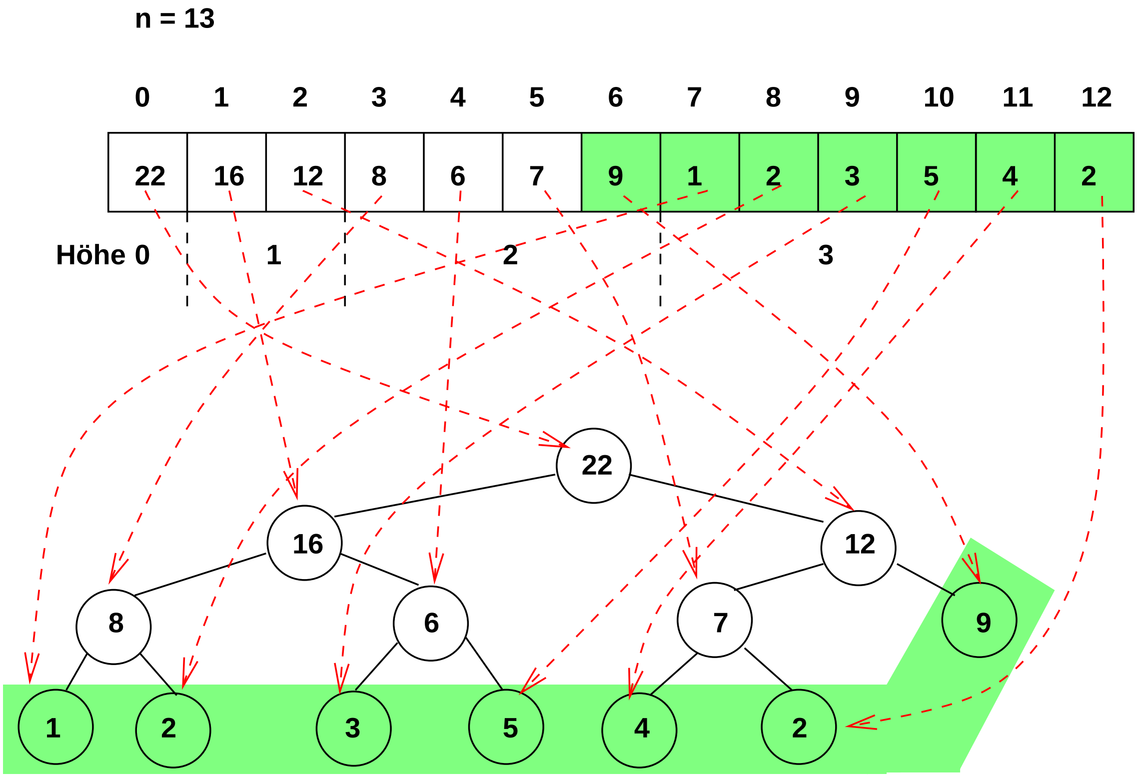


Abbildung 9.11: Vom Array zum Baum

9.5.2 Sortierverfahren

- Ausgangspunkt: die zu sortierenden Daten sind als Heap organisiert (Abb. 9.10, S. 206) und die Daten (Markierungen) sollen absteigend sortiert werden.
- Das größte Element steht in der Wurzel → entnehmen und ausgeben.
- Neuordnung des Baumes als Heap:
  - Suche Blatt b mit maximaler Höhe, kopiere seine Markierung in die Wurzel und lösche das Blatt b
    Ergebnis ist ein Semi-Heap (Abb. 9.12, S. 208)

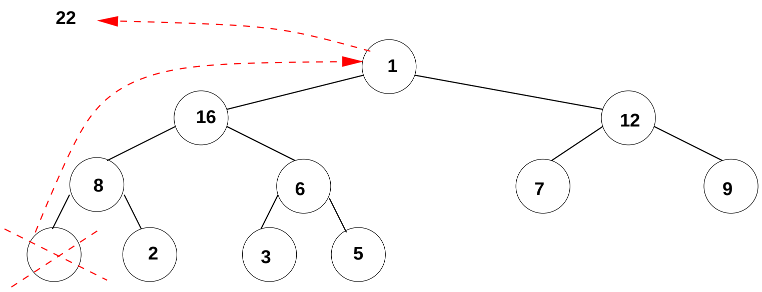


Abbildung 9.12: Schritt 2 von Heapsort – Semi-Heap

  - Aus dem Semi-Heap einen Heap machen:
    Sei v die Wurzel des Baumes (eines Teilbaumes), der einen Semi-Heap bildet.

    Prozedur `semiToHeap(v)`
    Eingabe: Semi-Heap mit Wurzel v
    Ergebnis: Heap durch Umordnung der Knotenmarken (siehe Abb. 9.14, 209)
    Algorithmus: (siehe Abb. 9.13, 208)

```
solange v nicht die Heap-Eigenschaft
    waehle Nachfolgerknoten w mit maximaler Markierung
    vertausche die Markierungen von w und v
    setze v = w
```

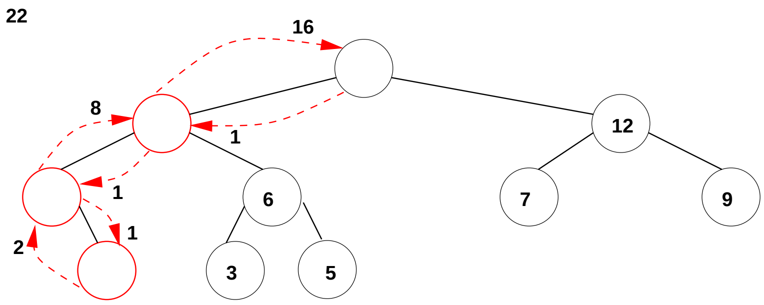


Abbildung 9.13: Schritt 3 von Heapsort – Semi-Heap nach Heap umsetzen

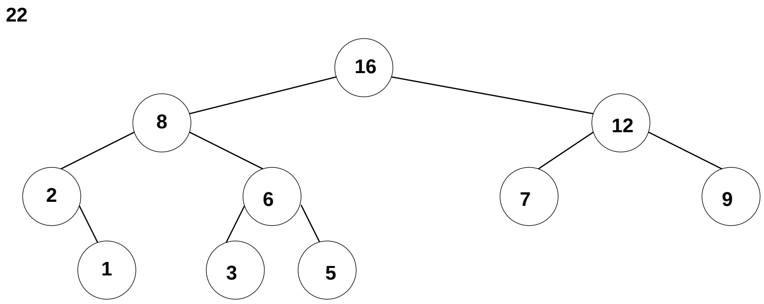


Abbildung 9.14: Ergebnis von Schritt 3 von Heapsort

- Umsetzung eines vollständigen Binärbaums in einen Heap:

  Prozedur `buildHeap`

  Eingabe: vollständiger Binärbaum mit Knotenmarkierung α

  Ergebnis: Heap (durch Umordnung der Knotenmarken)

  Algorithmus:

```
fuer i = h(T) - 1 bis 0 in Schritten von -1
     fuer alle inneren Knoten v der Hoehe h(v) == i
         semiToHeap(v)
```

- **Algorithmus Heapsort:**

  Eingabe: Vollständiger Binärbaum mit Wurzel r und Knotenmarkierung α

  Ausgabe: Absteigend sortierte Folge der Knotenmarken

  Algorithmus:

```
buildHeap;
solange r kein Blatt:

     gib α(r) aus;
     waehle Blatt b mit maximaler Tiefe;
     markiere r mit α(b);
     loesche Blatt b;
     semiToHeap(r);
gib α(r) aus;
```

Komplexitätsbetrachtungen:

- Ein vollständiger Binärbaum T mit n Knoten hat die Höhe $h(T) \leq log_2(n)$ – die Prozedur *semiToHeap* braucht daher maximal $log_2(n)$ Schritte!
- Der Algorithmus *buildHeap* ruft *semiToHeap* maximal n mal auf
- Dies ergibt die Komplexität $O(n \cdot log_2(n))$

9.5.3 Implementierung

Speicherung eines vollständigen Binärbaums in einem Array:

- Die Wurzel steht an Indexposition 0
- Die beiden Nachfolger eines Knotens, der an Indexposition i steht, stehen an den Positionen $2 \cdot i + 1$ und $2 \cdot i + 2$

Alle Knoten an den Positionen $0, \ldots, n \div 2 - 1$ sind innere Knoten, alle an den Positionen $n \div 2, \ldots n$ sind Blätter.

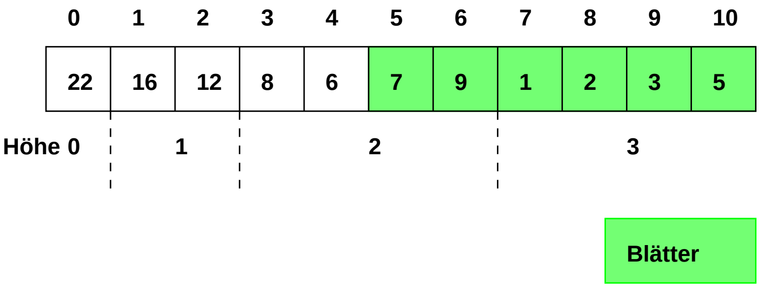


Abbildung 9.15: Array-Repräsentation des Heap von Abb. 9.10, S. 206

Statt nun die Marke der Wurzel auszugeben, wird diese mit der des zu löschenden Blattes vertauscht. Das “Löschen” des Blattes erfolgt dadurch, dass die Anzahl der noch zu berücksichtigenden Knoten n um 1 reduziert wird. So steht am Ende die aufsteigend sortierte Folge im Array.

Programm 9.10: Heapsort (*Heapsort.java*)

```
public class Heapsort {

  private static int a[];
  private static int n;

  public static void heapSort(int[] a0) {
    a = a0; n = a.length;
    if ( n == 0 ) return;
    sort ();
  }

  private static void semiToHeap(int v) {
     int w = 2 * v + 1; // first successor of v
     while ( w < n ) {
        if ( w + 1 < n )  // another successor ?
           if ( a[w+1] > a[w] ) w++;
        // now w is that successor of v with maximum label
        if ( a[v] >= a[w] ) return; // v has heap property
        swap(v,w); // exchange labels
        v = w; w = 2*v + 1;
        }
     }
```

```
    }

    private static void buildHeap (){
        for(int v = n / 2 - 1; v >= 0; v--)
            semiToHeap(v);
    }

    private static void sort () {
        buildHeap ();
        while ( n > 1 ) {
            n--;
            swap(0,n);
            semiToHeap (0);
        }
    }

    private static void swap(int i, int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

Programm 9.11: Anwendung von Programm 9.10, S. 210 (*TestHeapsort.java*)

```
import IOulm.*;

public class TestHeapsort {

    static int [] a = { 22,16,12,8,6,7,9,1,2,3,5 };
    static int n = a.length;

   public static void printArray () {
    for(int i = 0; i < n; i++) {
        Write.String(" " + a[i]);
    }
    Write.Ln();
    return;
   }

  public static void main(String[] args) {

        Heapsort.heapSort(a);
        printArray ();
        return;

  }
}
```

9.6 Shellsort

Benannt nach dem Urheber D.L. Shell: *A high-speed sorting procedure.* Communications of the ACM 2 (7), 30-32, 1959

Idee:

1. Die Elemente werden in einem zweidimensionalen Feld angeordnet
2. Die Elemente innerhalb einer Spalte werden sortiert (mit *InsertionSort*)
3. War es nur eine Spalte, so fertig!
4. Die so entstandene vorsortierte Folge wird wieder in einem zweidimensionalen Feld mit größerer Spaltenlänge angeordnet
5. weiter bei 2

Bei geschickter Wahl der Spaltenzahl ist das Verfahren sehr effizient.

Beispiel:

Gegeben sei die Folge

12	3	5	1	21	7	5	0	9	2	4	5	11	0	2	3	4	6	1	13

Anordnung in 7 Spalten:

$$\begin{array}{rrrrrrr} 12 & 3 & 5 & 1 & 21 & 7 & 5 \\ 0 & 9 & 2 & 4 & 5 & 11 & 0 \\ 2 & 3 & 4 & 6 & 1 & 13 & \end{array} \longrightarrow \begin{array}{rrrrrrr} 0 & 3 & 2 & 1 & 1 & 7 & 0 \\ 2 & 3 & 4 & 4 & 5 & 11 & 5 \\ 12 & 9 & 5 & 6 & 21 & 13 & \end{array}$$

Wieder linearisiert ergibt

0	3	2	1	1	7	0	2	3	4	4	5	11	5	12	9	5	6	21	13

Anordnen in 3 Spalten

$$\begin{array}{rrr} 0 & 3 & 2 \\ 1 & 1 & 7 \\ 0 & 2 & 3 \\ 4 & 4 & 5 \\ 11 & 5 & 12 \\ 9 & 5 & 6 \\ 21 & 13 & \end{array} \longrightarrow \begin{array}{rrr} 0 & 1 & 2 \\ 0 & 2 & 3 \\ 1 & 3 & 5 \\ 4 & 4 & 6 \\ 9 & 5 & 7 \\ 11 & 5 & 12 \\ 21 & 13 & \end{array}$$

Wieder linearisiert ergibt

0	1	2	0	2	3	1	3	5	4	4	6	9	5	7	11	5	12	21	13

Hier sind nur noch wenige Verschiebungen notwendig!

Nimmt man in einem Feld mit n Elementen eine wie im Beispiel oben vorgenommene Aufteilung in $h(=7)$ Spalten vor, so sind die Elemente der i-ten Spalte ($i = 0, \ldots, h-1$) die Elemente

$$a_{i+0\cdot h}, a_{i+1\cdot h}, a_{i+2\cdot h}, \ldots, a_{i+(n\div h)\cdot h} \quad \text{(sofern } i + (n \div h)\dot{h} < n\text{)}$$

Verwendet man *InsertionSort*, so startet man damit, dass das einelementige Teilfeld am Anfang einer jeden Spalte als sortiert gilt; d.h., die Elemente ab Indexposition h müssen in das Teilfeld ihrer Spalte einsortiert werden. Man muss hier nicht spaltenweise vorgehen, sondern kann nacheinander alle Elemente a_i, $i \in \{h, \ldots n-1\}$ in ihr Teilfeld $a_{i-j\cdot h}, j = 1, \ldots$ (so dass $i - j \cdot h \geq 0$) einsortieren.

Die verwendete Folge von Spaltenanzahlen bestimmt die Effizienz des Verfahrens – mehr dazu in der einschlägigen Literatur.

Programm 9.12: Shellsort (*Shellsort.java*)

```java
public class Shellsort {

   public static void shellSort (int [] a) {
      int i,j,k,h,t;
      int n = a.length;
//       int [] cols = { 1391376, 463792, 198768, 86961, 33936,
//                          13776, 4592, 1968, 861, 336, 112, 48,
//                           21, 7, 3, 1};
      int [] cols = { 9841, 3280, 1093, 364, 121, 40, 13, 4, 1};
          // Folge nach D. Knuth
      for( k =0; k < cols.length; k++) {
         h = cols[k];
         for(i = h; i < n; i++) {
            j = i; t = a[j];
            while ( j >= h && a[j-h] > t ) {
               a[j] = a[j-h];
               j = j - h;
            }
            a[j] = t;
         }
      }
   }
}
```

9.7 Bestimmung des Median

Definition: Sei $(M, \leq)$ eine Menge mit einer Ordnungsrelation und $a = a_0, \ldots, a_{n-1}$ eine endliche Folge mit $a_i \in M$. Ein Element a_i der Folge hat den **Rang** $k \in \{0, \ldots, n-1\}$ (ist das k-größte Element), wenn es in der sortierten Folge an Position k steht!

Für $k = 0$ erhält man das Minimum, für $k = n - 1$ das Maximum. Deren Bestimmung ist letztlich trivial und kann in $O(n)$ erfolgen. Sind alle Elemente verschieden, ist der Rang jeden Elementes eindeutig.

Der einfachste Weg, das Element mit Rang k zu bestimmen, ist, das Feld zu sortieren (Komplexität $O(n \cdot log(n))$).

Ein erster Algorithmus kann sich an der Partitionierung beim Quicksort orientieren. Beginnend beim ganzen Feld wird sukzessive partitioniert (siehe Abb. 9.16 (S. 214). Beim Suchen nach dem k-kleinsten Element müssen wir in dem Baum nicht alle Weg durchlaufen, sondern nur den Pfad aufbauen der uns zum Ziel führt.

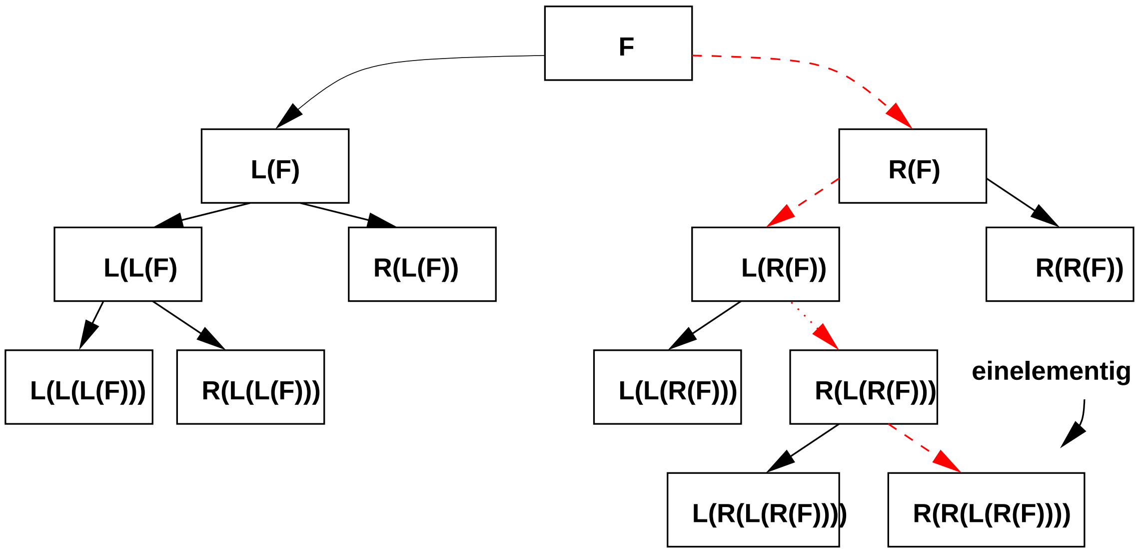


Abbildung 9.16: Suchbaum zur Rangbestimmung

Programm 9.13: Rangbestimmung (*Rank.java*)

```java
public class Rank{

  private static int[] a0;
  public static int rank(int[] a, int k) {
    // returns element with rank k''
    // the k-th smallest element
    int n = a.length;
    if ( n == 0 || k < 0 || k >= n ) return Integer.MIN_VALUE;
          // no good solution !
    a0 = new int[n];
    System.arraycopy(a,0,a0,0,n);
    return select(0, n-1,k);
  }

  private static int select(int left, int right, int k) {
    int ref = a0[ (left + right) / 2 ];
    int i = left, j = right;
    // partitioning
    while ( i <= j ) {
      while ( a0[i] < ref ) i++;
      while ( a0[j] > ref ) j--;
      if ( i <= j ) { swap(i,j); i++; j--;}
    }
    /* a[l] <= ref, f.a. l < i;
       a[l] >= ref, f.a. l > j;
       a[l] = ref, f.a. l > j && l < i
       i = j+1 oder i = j+2
     */

    if (k <= j) return select(left, j, k);
    if (k >= i) return select(i, right, k);
    return a0[k];
  }

 private static void swap(int i, int j) {
    int tmp = a0[j]; a0[j] = a0[i]; a0[i] = tmp;
 }
}
```

9.8 Textsuche

9.8.1 Problemstellung

Gegeben sei ein Alphabet **A** und Zeichenfolgen $s = s_0, s_1, \ldots, s_{n-1}$ und $p = p_0, p_1, \ldots, p_{m-1}$ der Längen n und m über **A** (sinnvollerweise sei $m \leq n$).

Definitionen:

1. Ein Fenster w_i^k der Länge k ist eine Teilfolge von s: $w_i^k = s_i, s_{i+1}, \ldots, s_{i+k-1}$, mit $i \geq 0, k > 0, i + k - 1 < n$
2. Ein Fenster $w_i^m = p$ heißt Vorkommen des Musters p im String s an der Position i.
3. Ein **Mismatch** in einem Fenster w_i^m ist eine Position j, an der das Muster nicht mit dem Fenster übereinstimmt: $p_j \neq (w_i^m)_j$

Definition: String-Matching-Problem

Finde alle Position von Vorkommen von p in s, d.h. $\{i | p = w_i\}$

Abb. 9.17, S. 216, zeigt ein Beispiel mit überlappendem Matching.

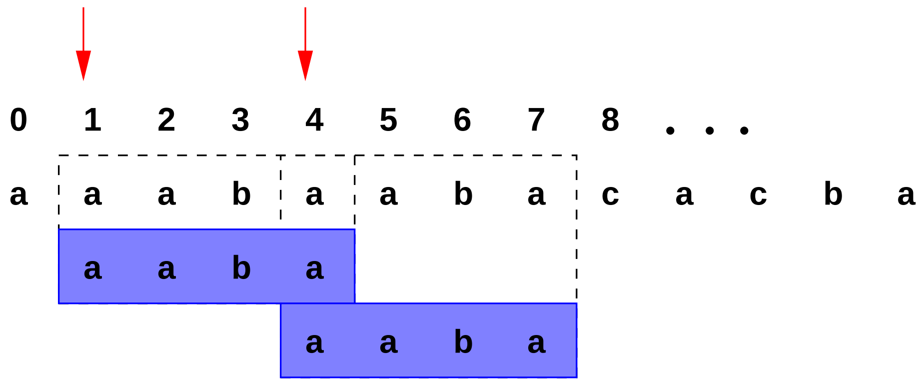


Abbildung 9.17: Pattern Matching

Es gibt verschiedene Strategien zur Lösung dieses Problems, von denen im Folgenden nur zwei dargestellt werden sollen.

9.8.2 Ein einfacher Algorithmus

Man überprüft an jeder Position des Textes, ob ab dieser der Suchbegriff vorkommt:

```
boolean found = false;
int i = 0;
while (!found && (i < n - m) {
   for(int j = 0; j < m; j++)
       if (s[i+j] != p[j]) break;
   if ( j >= m ) found = true;
   else i++;
}
```

Die Methode `naiveSearch()` in Programm 9.14, S. 217, bestimmt **alle** Vorkommen des Suchbegriffs und gibt jeweils die Startposition des Vorkommens aus.

Programm 9.14: Naive Textsuche (*TextSearch1.java*)

```
import IOulm.*;

public class TextSearch1 {

  static void naiveSearch (char[] s, char[] p) {
     // p Substring, pattern
     // prints all indices where p begins in s
     int i = 0, j, n = s.length, m = p.length;
     boolean first = true;
     while (i <= n - m) {
        j = 0;
        while( (j <m) && (s[i+j] == p[j]) ) j++;
        if ( j == m )
           if ( first ) { Write.String(" " + i); first = false; }
           else Write.String(", " + i);
        i++;
     }
     Write.Ln();
  }

  public static void main(String[] args) {
     String s, p; int n,m;
     char [] sTmp, pTmp = {};
     int line = 0;
     // get patternstring as first argument
     if ((args != null) && (args.length == 1)) {
          p = args[0]; m = p.length();
          pTmp = new char[m]; p.getChars(0,m,pTmp,0);
     } else {
          Write.Line("exactly one argument needed: searchString");
          System.exit(1);
     }

     // read string in which to search from stdin
```

```
    while (Urc.readString()) {
            s = Urc.getString();
            n = s.length();
            sTmp = new char[n]; s.getChars(0,n,sTmp,0);
//          Write.Line(s);
            Write.String("Line " + line + ": ");
            naiveSearch(sTmp,pTmp);
            Write.Ln();
            line++;
    }
    return ;
  }
}
```

Der zu suchende String wird hier als **Argument** übergeben. Da wir den Zugriff auf die einzelnen Zeichen des Strings benötigen, wird der String in ein Char-Array konvertiert. Der zu durchsuchende Text wird zeilenweise von der Standardeingabe (durch Umlenkung auch aus einer Datei) eingelesen.

Analyse:

- schlechtester Fall:

  Die äußere Schleife (Variable i) wird $n - m + 1$ mal durchlaufen, die innere Schleife (Variable j) wird höchstens m mal durchlaufen.

  Dies ergibt für die Anzahl der Vergleiche: $V = (n - m + 1) \cdot m$, also $V \in O(n \cdot m)$

  Ist das Muster kurz im Vergleich zum Text ($m \ll n$), so ist

  $V = (n - m + 1) \cdot m \geq (n - \frac{n}{2} + 1) \cdot m \geq \frac{n}{2} \cdot m$, also $V \in \Theta(n \cdot m)$[6]

- günstigster Fall:

  Dieser liegt dann vor, wenn der erste Vergleich bereits einen Unterschied bringt; in diesem Fall werden insgesamt $n - m + 1$ Vergleiche getätigt!

- Verhalten im Durchschnitt:

  Dies lässt sich abhängig vom Muster und der Wahrscheinlichkeitsverteilung der Symbole des zu druchsuchenden Textes wie folgt ermitteln:

  Sei h_j die Wahrscheinlichkeit, mit der das j-te Symbol des Musters im zu durchsuchenden Text vorkommt (ist j-te Symbol ein e und sind 15% der Symbole im Text e's, so wäre $h_j = 0.15$

  Sei v die durchschnittliche Anzahl der Vergleiche für jede Position i des zu durchsuchenden Textes – also die mittlere Anzahl der Iterationen in der inneren Schleife (Variable j).

  Dann gilt:

  Das erste Symbol des Musters wird immer verglichen – Vergleiche: 1

  In h_0 aller Fälle stimmt das erste Symbol überein, somit muss dann das zweite Symbol verglichen werden: Vergleiche: h_0.

  In h_1 aller Fälle stimmt auch das zweite überein, so dass auch dass dritte Verglichen werden muss: Vergleiche $h_0 \cdot h_1$

  Insgesamt: $v = 1 + h_0 + h_0 \cdot h_1 + \ldots h_0 \cdot h_1 \cdot \ldots h_{m-2}$

[6] $\Theta(n \cdot m)$ bedeutet, dass der Algorithmus mindestens, aber auch höchstens proportional zu $n \cdot m$ Schritte benötigt

Abschätzung:
Sei $h = max(h_j) < 1$ (der Text besteht also nicht nur aus einem einzigen Symbol).
Dann gilt: $v \leq \sum_{k=0}^{m-1} h^k \leq \frac{1}{1-h}$
Für die Anzahl der Vergleiche ergibt sich somit: $V \leq \frac{n-m+1}{1-h} \in O(n)$

Dieser Algorithmus hat also lineare Laufzeit!

9.8.3 Knuth-Morris-Pratt-Algorithmus (KMP)

Der KMP-Algorithmus nutzt die bei Übereinstimmungen gewonnene Information aus.

In Abb. 9.18 (S. 220) kommt es nach 5 Übereinstimmungen zwischen Suchbegriff und Text erstmals zu einer Differenz (**Mismatch**).

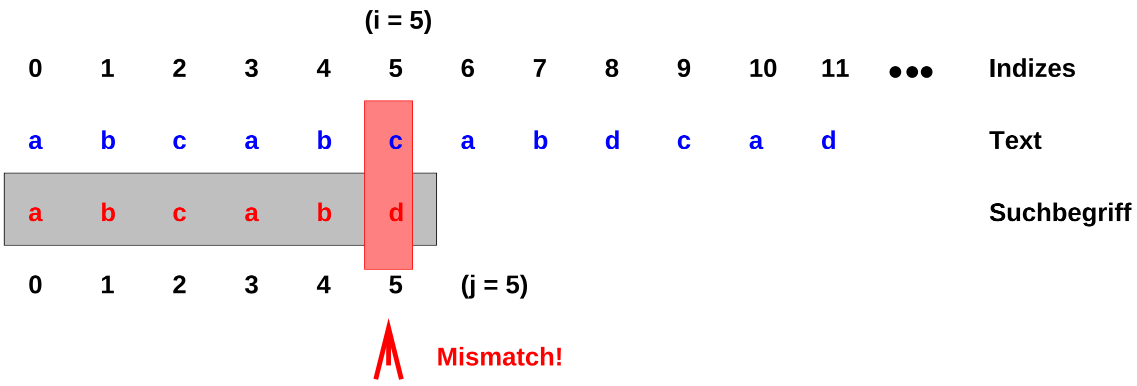


Abbildung 9.18: Mismatch

Beim naiven Algorithmus wurde der Suchbegriff um eine Indexposition im Text (hier: Position 1) weitergeschoben und der Vergleich startete erneut beim ersten Symbol im Suchbegriff. Aus der Kenntnis des Suchbegriffs heraus ist aber klar, das an der nächsten Indexposition (hier also 1) ein b steht und auch der übernächsten (also: 2) kein a steht. Ein Chance besteht erstmals auf Indexposition 3, die umso größer ist, da danach ein b im Suchbegriff kommt, was wir auch nach dem a im Suchbegriff erwarten. Das auf Indexposition 5 stehende c kennen wir nicht, da wir ja die Symbol immer nur auf Gleichheit vergleichen.

Es macht als Sinn, den nächsten Vergleich mit der aktuellen Textposition ($i = 5$) ab Indexposition $j = 2$ im Suchbegriff zu starten.

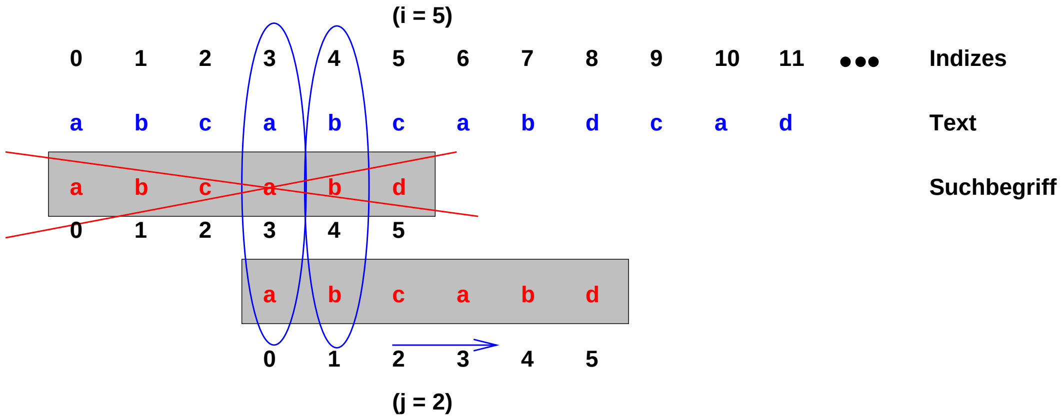


Abbildung 9.19: Neuer Start nach Mismatch

Es lohnt sich also, vorab den Suchbegriff zu analysieren!

Grundbegriffe:

Sei A ein Alphabet und $x = x_0 x_1 \ldots x_{k-1}$, $k \in \mathbb{N}$ eine Zeichenfolge der Länge k über A.

$|\,x\,| \;:=\; k$

Ein **Präfix** der Länge b von x ist eine Teilzeichenreihe $u = x_0 \ldots x_{b-1}$ mit $0 \leq b \leq k$ (Anfangswort)
Ist $u \neq x$ ($b < k$, so ist u ein **echter Präfix**

Ein **Suffix** der Länge b von x ist eine Teilzeichenreihe $u = x_{k-b} \ldots x_{k-1}$ mit $0 \leq b \leq k$ (Endwort)
Ist $u \neq x$ ($b < k$, so ist u ein **echter Suffix**

Ein **Rand** der **Breite** b von x ist eine Teilzeichenreihe r mit
$r = x_0 \ldots x_{b-1}$ **und** $r = x_{k-b} \ldots x_{k-1}$ mit $0 \leq b < k$

Ein Rand ist also gleichzeitig Präfix wie auch Suffix!

Beispiel: $x = abacab$

- Echte Präfixe von x sind: $\epsilon,\ a,\ ab,\ aba,\ abac,\ abaca$
- Echte Suffixe von x sind: $\epsilon,\ b,\ ab,\ cab,\ acab,\ bacab$
- Ränder von x sind ϵ, ab

Ist ein Rand des bislang übereinstimmenden Präfixes des Suchbegriffs bekannt, so kann das "Verschieben" des Suchbegriffs über dem zu durchsuchenden Text in größeren Schritten erfolgen, am besten ist der breiteste Rand dieses bislang übereinstimmenden Präfixes (Abb. 9.20, S. 221)!

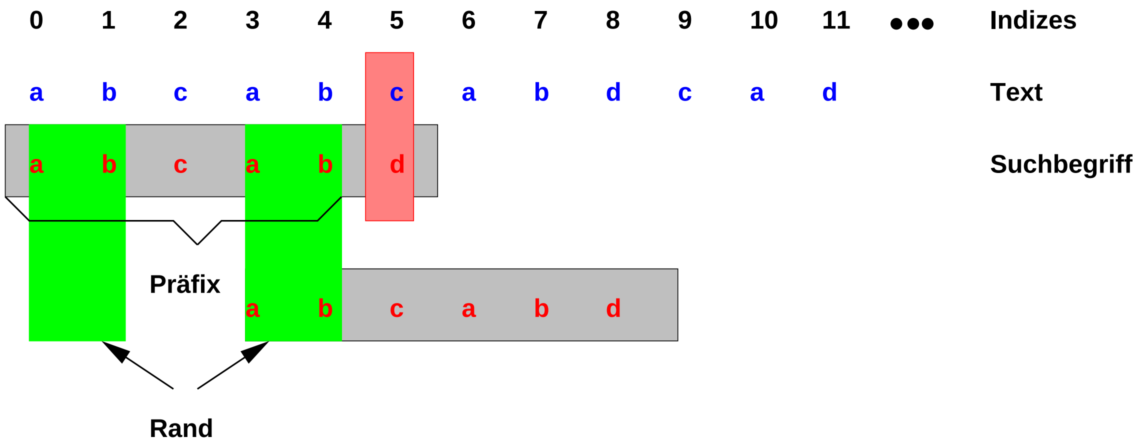


Abbildung 9.20: Schiebedistanz beim KMP-Algorithmus

Es gilt: Sind r und s Ränder von x mit $\mid r \mid \leq \mid s \mid$, so ist r ein Rand von s.

Ohne Beweis, die Skizze in Abb. 9.21, S. 222, möge genügen!

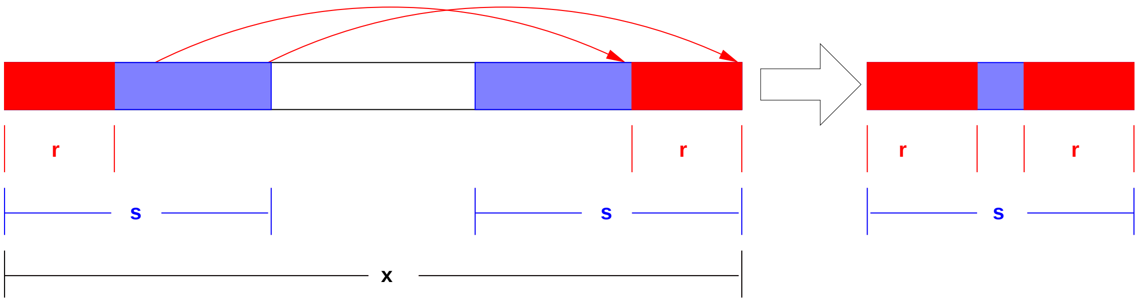


Abbildung 9.21: Breitester Rand (KMP-Algorithmus)

Es gilt weiter: Ist r der breiteste Rand von x, so erhält man den nächst breitesten Rand von x als breitesten Rand von $r, \ldots$

Sei $u = p_0 \ldots p_{i-1}$ ein echtes Präfix des Musters. Ein Rand r der Breite j von u lässt sich durch das Symbol p_i **fortsetzen**, wenn $p_j = p_i$ ist.

Dies zeigt anschaulich die Abb. 9.22, S. 222. Damit erggibt sich ra als Rand von ua!

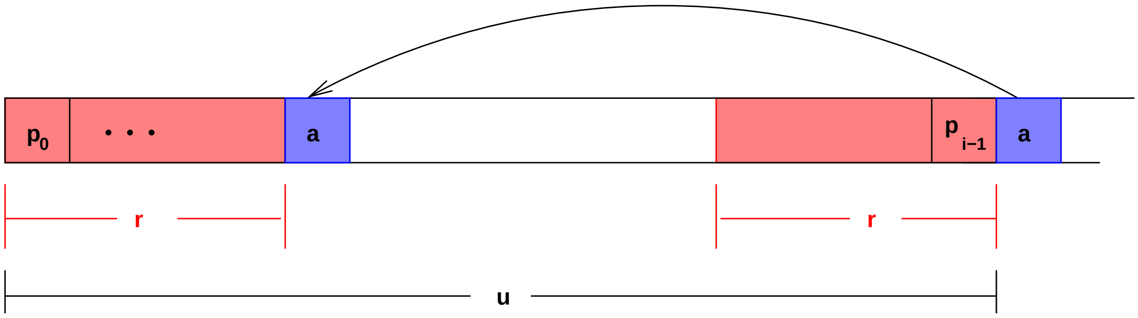


Abbildung 9.22: Randfortsetzung (KMP-Algorithmus)

Vor der eigentlichen Mustersuche wird in einer Vorlaufphase ein Array b der Länge $m + 1$ (m: Musterlänge) wie folgt berechnet:

$b[i]$ ist Länge des breitesten Randes des Präfix der Länge i, für $i = 1, \ldots, m$; $b[0]$ wird auf -1 gesetzt, da das Präfix der Länge 0 keinen Rand hat. Damit weiss man für jeden übereinstimmenden Präfix die Schiebedistanz!

Algorithmus zur Bestimmung des Arrays b:

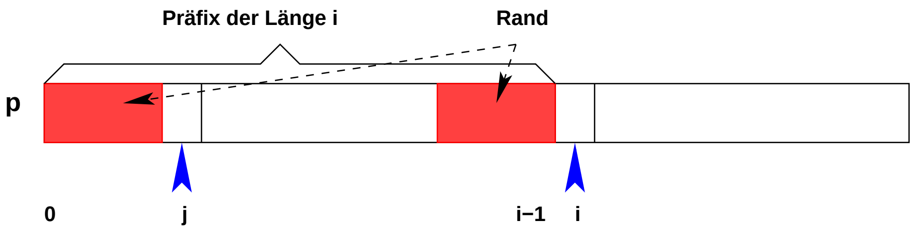


Abbildung 9.23: Vorlaufphase des KMP-Algorithmus

(1) Der Algorithmus durchläuft das Muster mit den beiden Indizes j und i

(2) j und i zeigen jeweils hinter den breitesten Rand des Präfixes $p_0 \dots p_{i-1}$

(3) Ist $p_j = p_i$, so lässt sich der Rand fortsetzen:

- (3a) erhöhe j und i jeweils um 1
- (3b) j und i zeigen wieder jeweils hinter den breitesten Rand des Präfixes $p_0 \dots p_{i-1}$
- (3c) als Länge des breitesten Randes des Präfixes des Länge i wird $b[i] = j$ gesetzt.

(4) Ist $p_j \neq p_i$, so lässt sich der Rand nicht fortsetzen.

- (4a) Es wird versucht, den nächst breitesten Rand fortzusetzen
- (4b) Dies ist der nächst breiteste Rand von $p_0 \dots p_{j-1}$ – dessen Breite steht in $b[j]$
- (4c) also wird $j = b[j]$ gesetzt: weiter bei (3)

(5) Lässt sich kein Rand fortsetzen, so bricht die Iteration bei $j = -1$ ab! In jedem Fall wird nach dem Erhöhen von j und i (Schritt (3a)) der Eintrag $b[i] = j$ vorgenommen!

Programm 9.15: KMP – Vorverarbeitung (*KMPPreprocessing.java*)

```
import IOulm.*;

public class KMPPreprocessing{

  static void preprocess (char [] p, int [] b){
      int i = 0, j = - 1;
      b[i] = j;
      while ( i < p.length ) {
          while ( j >= 0 && p[i] != p[j] ) j = b[j];
          i++; j++;
          b[i] = j;
      }
  }

  public static void main(String [] args) {
    int j, m = 6;
    char []p = {'a','b','a','b','a','a'};
    int []b = new int[m+1];
    preprocess (p,b);

```

```
    Write.String("   j: ");
    for(j=0; j <= m; j++) Write.String(" " + j );
    Write.Ln();

    Write.String("p[j]: ");
    for(j=0; j < m; j++) Write.String(" " + p[j] );
    Write.Ln();

    Write.String("b[j]:");
    for(j=0; j <= m; j++) Write.String(" " + b[j] );
    Write.Ln();

    return ;
  }
}
```

Ergebnis:

```
   j:  0 1 2 3 4 5 6
p[j]:  a b a b a a
b[j]: -1 0 0 1 2 3 1
```

Interpretation:

Für ein gegebenes j (Index im Suchbegriff), $0 \leq j < m$, ist $b[j]$ die Länge des breitestens Randes im Präfix der Länge j:

j	Präfix	Rand	Randlänge
0	ϵ	ϵ	-1
1	a	ϵ	0
2	ab	ϵ	0
3	aba	a	1
4	$abab$	ab	2
5	$ababa$	aba	3
6	$ababaa$	a	1

Tritt nun beim Vergleich von Textposition i mit der Position j des Suchstrings ein Mismatch auf, so kann ein erneuter Vergleich mit Textposition i ab Position $b[j]$ (falls $b[j] \geq 0$) starten. In Abb. 9.24, S. 225, sind die verschiedenen Situation zu obigem Beispiel aufgelistet.

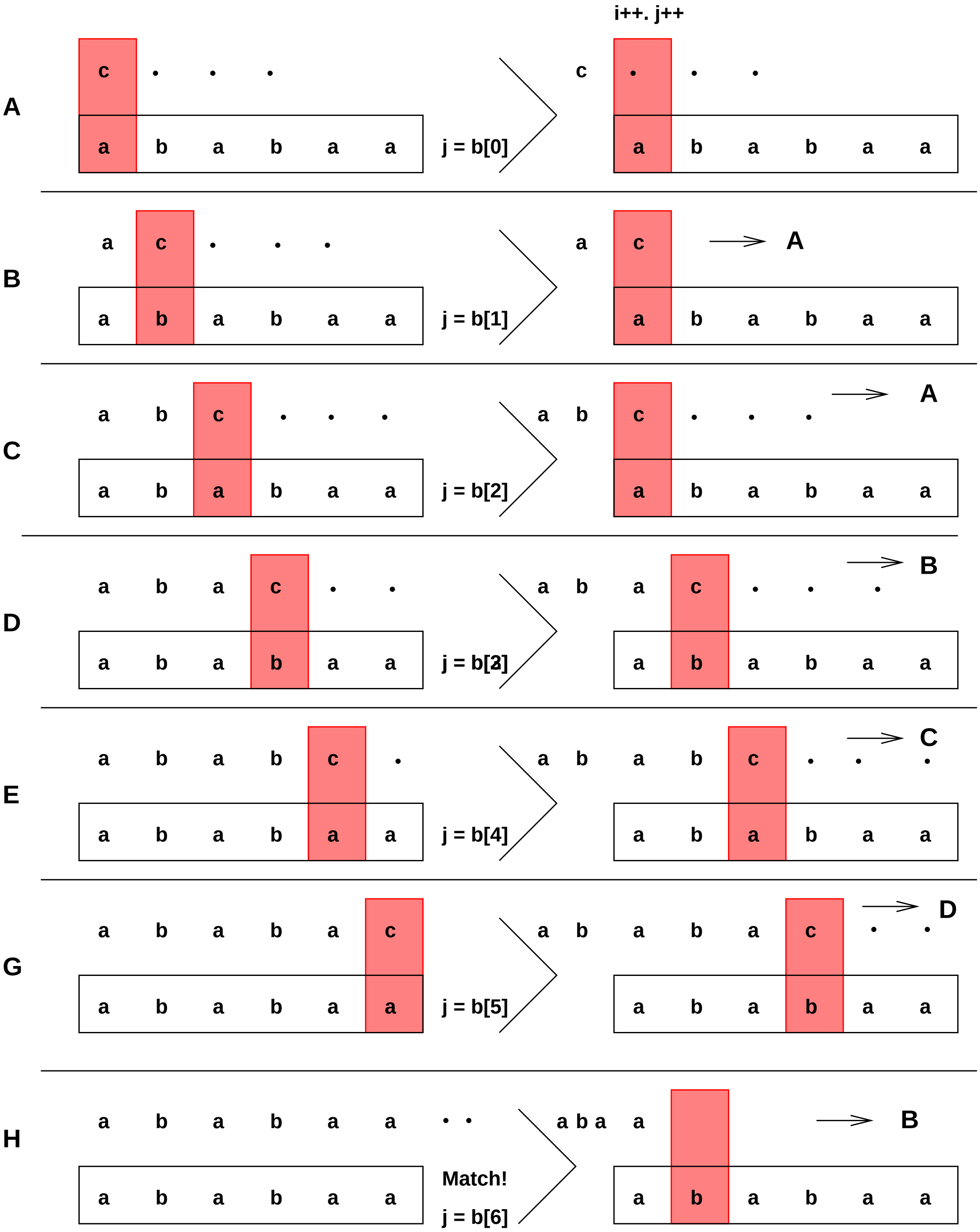


Abbildung 9.24: Verschieben am Beispiel

Die eigentliche Suche:

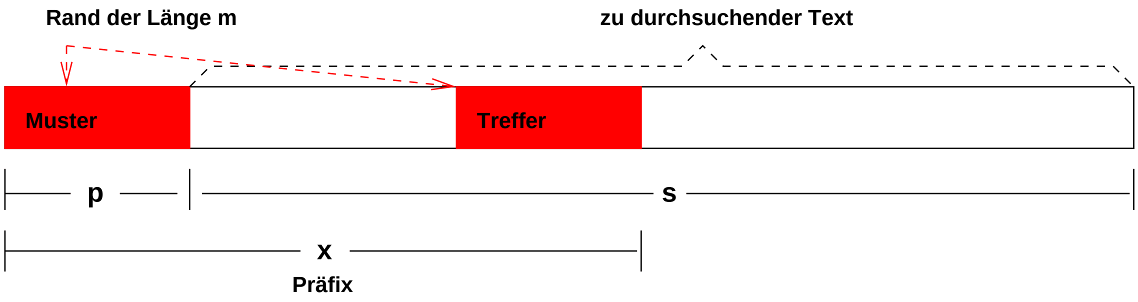


Abbildung 9.25: KMP-Suche: Rand der Breite m eines Präfixes x von ps

Wie aus Abb. 9.24, S. 225, bereits zu vermuten ist, kann der Vorverarbeitungsalgorithmus (Methode *preprocess()* in Programm 9.15, S. 9.15) konzeptionell statt auf p auf die Zeichenkette ps (s der zu durchsuchende Text) angewandt werden. Zugelassen sind natürlich nur Ränder der maximalen Länge m – ein Rand des Präfixes x der Breite m ist dann ein Treffer (Match), sofern sich der Rand nicht überschneidet!

Programm 9.16: KMP – das Programm (*KMP.java*)

```
// Knuth−Morris−Pratt−Algorithmus
import IOulm.*;

public class KMP{

  static private void preprocess (char [] p, int [] b){
      int i = 0, j = − 1;
      b[i] = j;
      while ( i < p.length ) {
          while ( j >= 0 && p[i] != p[j] ) j = b[j];
          i++; j++;
          b[i] = j;
      }
  }

  static public void search (char[] s, char[] p) {
    boolean first = true;
    int [] b = new int[p.length + 1];
    int i = 0, j = 0, n = s.length, m = p.length;
    preprocess (p,b);
    while ( i < n ) {
        while ( j >= 0 && s[i] != p[j]) j = b[j];
        i++; j++;
        if ( j == m ) {
          if ( first ) { Write.String(" " + (i−j) ); first = false; }
          else Write.String(", " + (i−j));
          j = b[j];
        }
    }
  }

}
```

Erläuterungen:

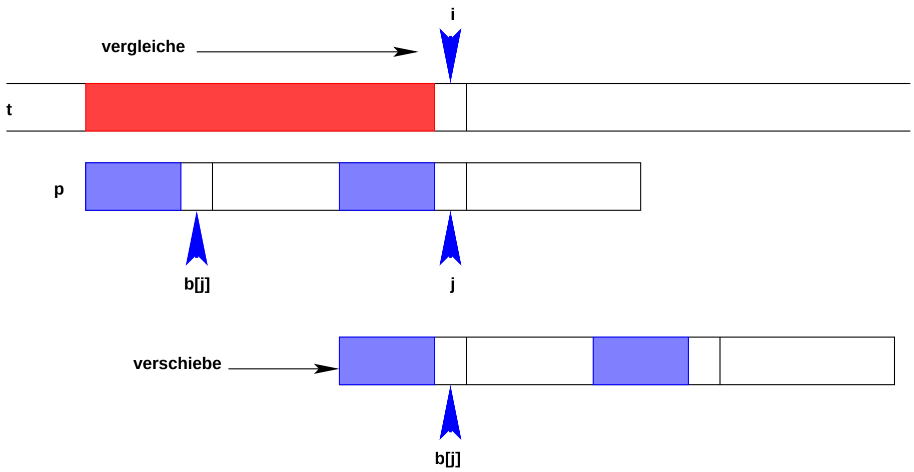


Abbildung 9.26: KMP-Algorithmus: Mismatch

- In der innersten Schleife (Zeile 22) wird bei einem Mismatch an Position j der breiteste Rand des übereinstimmenden Präfixes des Musters betrachtet (Abb. 9.26) – dieser hat die Breite $b[j]$
- Es wird weiterverglichen an Position $b[j]$ – damit ergibt sich eine neue Ausrichtung des Musters
- Diese Ausrichtung berücksichtigt das breiteste Präfix des Musters, das bereits übereingestimmt hat
- Die Schleife iteriert, bis Übereinstimmung vorliegt oder bis kein Rand mehr vorhanden ist ($j = -1$).
- Wenn alle m Symbole des Musters übereinstimmen (Zeile 24), so wird die Positionsnummer des Starts der Übereinstimmung ausgegeben.

Beispiel:

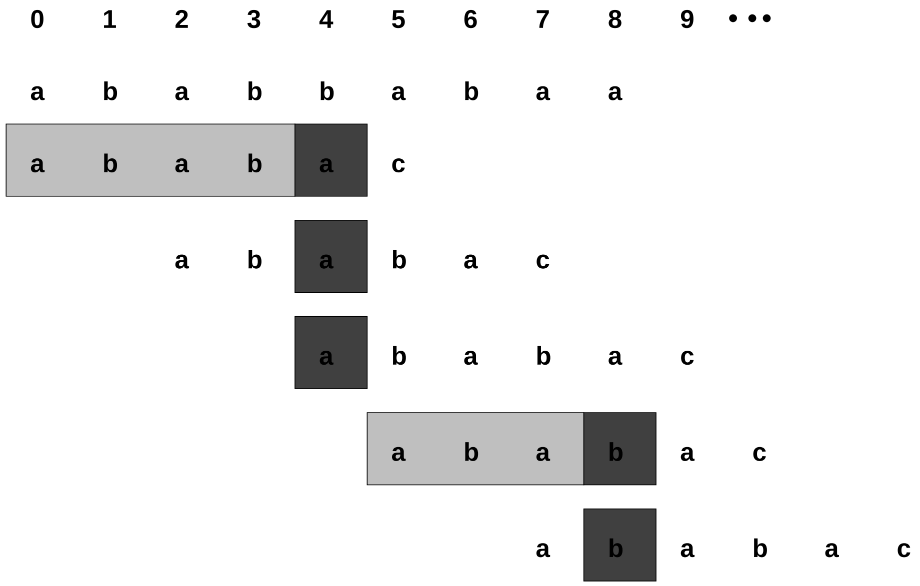


Abbildung 9.27: KMP-Algorithmus: Beispiel für Behandlung von Mismatches

Übereinstimmungen sind hellgrau, Mismatches dunkelgrau hinterlegt!

- Beim ersten Mismatch verschiebt der Suchalgorithmus verschiebt das Muster soweit, wie der breiteste Rand des übereinstimmenden Präfixes es zulässt.
- Das übereinstimmende Präfix ist *abab*, sein breitester Rand ist *ab*.
- Es kommt zu einem erneuten Mismatch, da *ab* durch *a* fortgesetzt werden kann.
- Der nächstbreiteste Rand von *abab* ist ϵ, auch dieser kann durch *a* fortgesetzt werden, es kommt zum nächsten Mismatch!

Insgesamt wird *a* im Muster 3-mal mit einem *b* im zu durchsuchenden Text verglichen. Dies lässt sich wie folgt beheben:

- Im Array *b* wird für jedes echte Präfix des Musters nicht generell die Länge des breitesten Randes vermerkt, sondern die Länge des breitesten Randes, der sich nicht durch das darauffolgende Symbol fortsetzen lässt oder -1, falls es keinen solchen Rand gibt!
- Im Vorverarbeitungsteil wird der jeweils gefundene breiteste Rand nicht durch $b[i] = j$ nur dann eingetragen, wenn $p_i \neq p_j$, der Rand sich also nicht durch das darauffolgende Symbol p_i fortsetzen lässt.
- Ist dagegen $p_i = p_j$, so muss der nächstbreiteste Rand gefunden werden, der sich nicht durch p_i fortsetzen lässt. Der Eintrag $b[j]$ verweist auf den breitesten Rand des Präfixes $p_0 \ldots p_{j-1}$, der sich nicht durch p_j fortsetzen lässt, also ist $b[i] = b[j]$ die richtige Wahl!

Im Beispiel (Abb. 9.28, S. 229) sieht das wie folgt aus:

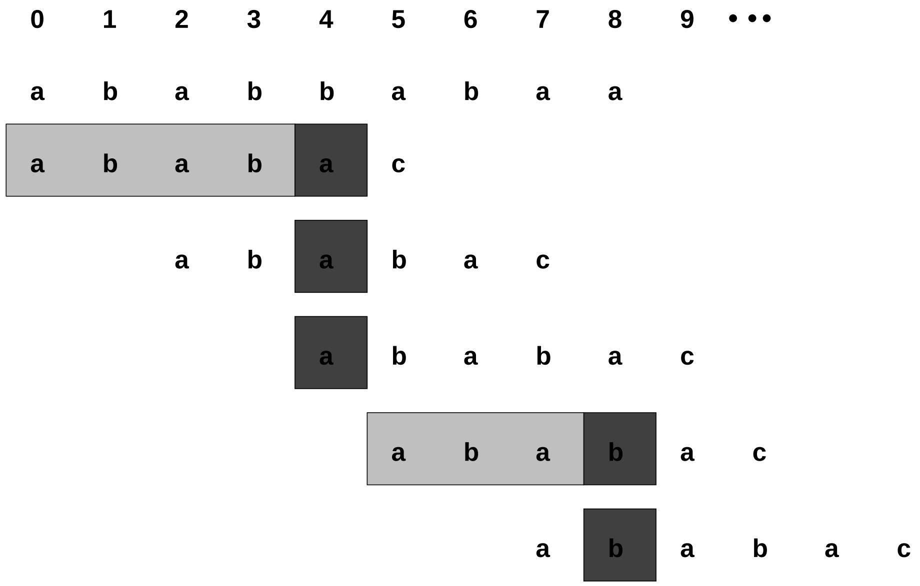


Abbildung 9.28: KMP-Algorithmus: Beispiel für verbesserte Behandlung von Mismatches

Der verbesserte Vorverarbeitungsalgorithmus:

Programm 9.17: KMP mit verbesserter Vorverarbeitung (*KMP-improved*)

```java
static void preprocess (char [] p, int [] b){
    int i = 0, j = - 1;
    b[i] = j;
    while ( i < p.length ) {
        while ( j >= 0 && p[i] != p[j] ) j = b[j];
        i++; j++;
        if (i < p.length && p[i] == p[j])
            b[i] = b[j];
        else
            b[i] = j;
    }
}
```

Auch der KMP-Algorithmus ist in $O(n)$.

Kapitel 10

Dynamische Datenstrukturen

10.1 Listen

10.1.1 Konzepte, Terminologie

- Datenstrukturen sind mehr als eine Anhäufung von Daten.
- Spielen Zugriffszeiten eine Rolle, dann sollten die Daten strukturell so angeordnet werden, dass alle wichtigen Zugriffs-Operationen effizient unterstützt werden. Entsprechend spiegeln sich häufig die strukturellen Beziehungen der Daten und die Zugriffsmethoden auch in der Vernetzung der Daten wider.
- Entsprechend hängt die Wahl einer geeigneten Datenstruktur von der Art und Häufigkeit der benötigten Zugriffsoperationen ab.
- Zu den einfachsten Datenstrukturen zählen lineare Listen, bei denen eine Reihe von Elementen a_i in einer geordneten Form repräsentiert werden:

$$a_1\ a_2\ \dots\ a_{i-1}\ a_i\ a_{i+1}\ \dots\ a_n$$

- Zu den typischen Operationen auf linearen Listen zählen
  - das Hinzufügen eines Elements am Anfang oder oder am Ende der Liste,
  - das Entfernen eines Elements,
  - das Betrachten des i-ten Elements (insbesondere für $i = 1$ oder $i = n$) und
  - das Betrachten des nächsten oder vorherigen Elements (also a_{i+1} oder a_{i-1} ausgehend von a_i).

Wir hatten bislang Arrays zur Zusammenfassung von Daten gleichen Typs zu einer neuen Einheit. Arrays sind von Haus aus statisch, d.h. ihre Größe wird einmal bestimmt und bleibt dann konstant. Statt Zusammenfassung sagt man häufig auch **Container** oder auch **Collection**.

Listen sind spezielle Datenstrukturen, die sich speziell zur Zusammenfassung von Objekten (gleichen Typs? → später mehr dazu) eignen, deren Anzahl dynamisch wachsen und auch schrumpfen kann. Java stellt im Paket **java.util** den Datentyp Liste als eine spezielle Sammlung (*collection*) bereit. **java.util.List** ist eine Methoden-Schnittstelle (*interface*), die durch die Klassen **LinkedList** und **ArrayList** implementiert wird. In der im folgenden aufgebauten Terminologie ist *LinkedList* eine doppelt verkettete Liste; *ArrayList* ist eine alternative Implementierung mit den für Arrays typischen Eigenschaften.

Beispiele für Listen

- Aufträge: Jeder Eintrag entspricht einem Auftrag, der angibt, welcher Artikel an wen zu versenden ist. Neue Aufträge werden an das Ende der Liste angefügt. Der erste Auftrag wird bearbeitet und nach Erledigung entfernt.
- Spielzüge: Jeder Eintrag entspricht einem Spielzug. Wenn normal weitergezogen wird, kommt ein neuer Spielzug hinzu, der an das Ende der Liste anzufügen ist. Wenn jedoch ein Spielzug zurückgenommen werden soll, wird der letzte Eintrag aus der Liste entfernt und dazu verwendet, den alten Spielstand zu restaurieren.
- Terminkalender: Jeder Eintrag besteht aus einer Zeitangabe und einer kurzen Information. Alle Einträge werden nach der Zeit sortiert angeordnet. Betrachtet wird nur der erste Eintrag, der als nächster relevant ist. Hinzukommende Einträge werden entsprechend ihrer Zeit einsortiert. Einträge am Anfang der Liste, die nicht mehr aktuell sind, werden entfernt.

Gegeben sei eine Objektklasse *K*; enthalten Objekte vom Typ *K* eine Referenzvariable vom selben Typ als Feld, so kann man Objekte vom Typ *K* über diese Referenzen zu einer primitiven Liste verknüpfen (siehe Abb. 10.1, S. 232). In nicht 00-Sprachen ist damit bereits eine verkettete Liste (die Referenzen heißen dann meist Zeiger) definiert. Im OO-Kontext spricht man zunächst nur von einen Knoten-Kette, da die eigentliche Liste in einer (Container-) Klasse samt den zugehörigen Listen-Methoden zusammengefasst wird.

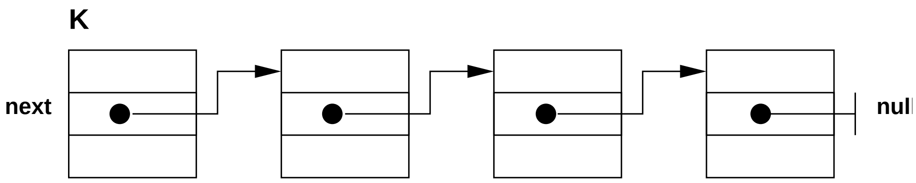


Abbildung 10.1: Primitive Liste

Implementierungsskizze zu Abb. 10.1, S. 232:

```
class Node {
   Object content; // was auch immer in einem Listenelement
                   // gehalten wird
   Node next;      // Referenz auf ein Objekt vom Typ K
}
```

Hier stellt sich zunächst die Frage nach der **leeren Liste** und damit verbunden auch die nach dem Konstruktor. Naheliegend wäre folgender Konstruktor:

```
public Node(){ content = null; next = null; }
```

Damit könnte prinzipiell eine leere Liste wie folgt erzeugt werden:

```
Node list = new Node();
```

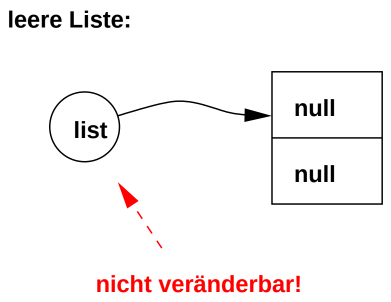


Abbildung 10.2: Primitive Liste – die leere Liste

Die Erkennung einer Liste als leere Liste ist nun über die *content*-Referenz möglich, alternativ könnte man auch einen Zähler führen.

Genau betrachtet sieht das Listenelement aus Abb. 10.2 (S. 233) wie in Abb. 10.3 (S. 233) dargestellt aus.

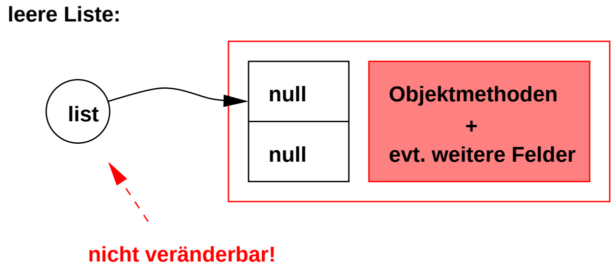


Abbildung 10.3: Primitive Liste – die leere Liste, genau betrachtet

Ein erstes Element einfügen geht leicht:

```
public void addFirstEl(Object el){ content = el; }
```

Einfügen in eine nicht leere Liste:

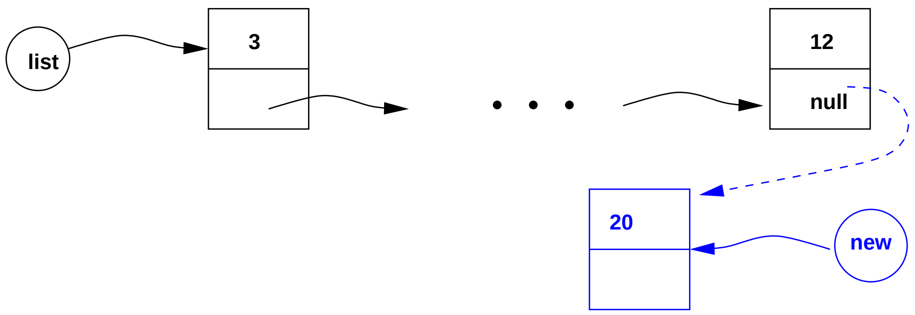


Abbildung 10.4: Primitive Liste – Einfügen

Genau betrachtet sieht die Liste aus Abb. 10.4 (S. 234) wie in Abb. 10.5 (S. 234) dargestellt aus.

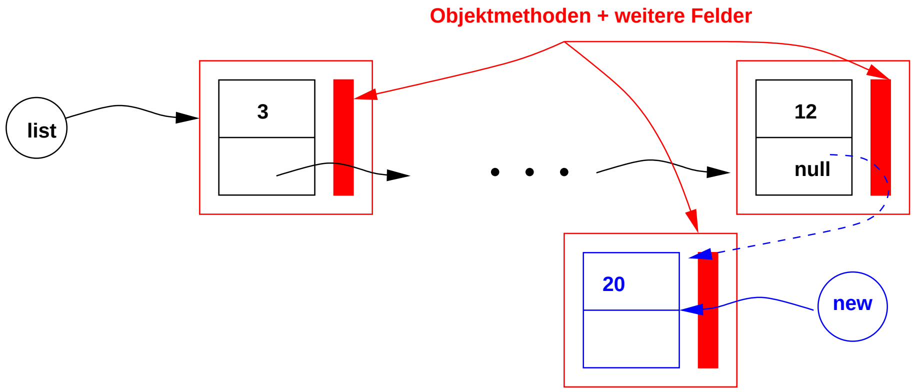


Abbildung 10.5: Primitive Liste – genau betrachtet

Es macht wenig Sinn, damit weiterzuarbeiten!

Bessere Alternative:

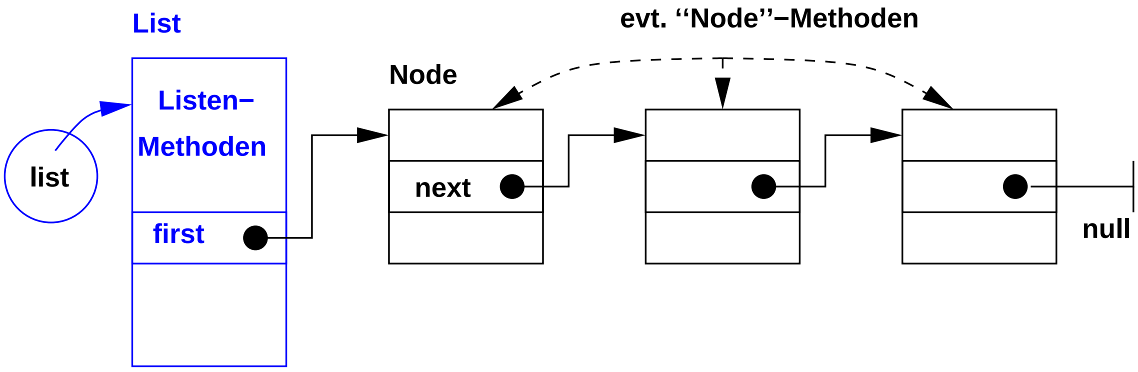


Abbildung 10.6: Einfache Liste (einfach verkettet)

Implementierungsskizze zu Abb. 10.6, S. 235:

```
class List{
       private class Node {
           Object content;
           Node next;
           Node(Object o){ content = o; next = null; }
       }
       private Node first;
}
```

Die Klasse *List* enthält die Definition einer **lokalen Klasse** – diese beschreibt ein allgemeines Listenelement samt dessen Konstruktor (und ggf. weiterer Methoden). Das Feld *first* enthält eine Referenz auf das erste eigentliche Listenelement (siehe Abbildung 10.6, S. 235). In der Klasse *List* könnten weitere Felder (z.B. Zähler, der die Anzahl der Elemente eines Listenobjekts enthält) und natürlich auch Methoden enthalten sein.

10.1.2 Implementierung von Listen

Programm 10.1 (S. 236) enthält die Implementierung einer ersten Listenklasse.

Programm 10.1: Erste Listenimplementierung (*SimpleList/List.java*)

```
1  public class List {
2
3        private class Node {
4              Object content;
5              Node next;
6              Node(Object o){ content = o; next = null; }
7        }
8
9        private int count;
10       private Node first;
11
12       public List(){
13          this.count = 0;
14          this.first = null;
15       }
16
17       public void add(Object el){ // am Anfang einfuegen
18          Node tmp = new Node(el);
19          tmp.next = first;
20          first = tmp;
21          count++;
22       }
23
24       public String toString() {
25          StringBuffer sb = new StringBuffer(); // veraenderliche String-Puffer
26          sb.append("(");
27          Node x = first;
28          while ( x != null ) {
29             sb.append(x.content.toString()); // Der Inhalt muss eine
30                                              // toString() Methode
31                                              // implementiert haben!
32             x = x.next;
33             if ( x != null ) sb.append(", ");
34          }
35          // abschliessen
36          sb.append(")");
37          return sb.toString();
38       }
39 }
```

Erläuterungen:

- Zeilen 3-7
  Klasse für die eigentlichen Listenelemente – **private**, weil für die "Außenwelt" unwichtig und damit unsichtbar gemacht; dies gilt auch für den Konstruktor!

- Zeilen 9-10

  die Felder unseres "Kopf"-Elementes (Abb. 10.6, S. 235)

- Zeilen 12-15

  Der Konstruktor setzt die Anzahl der eigentlichen Listenelemente auf 0 und den Zeiger auf das erste Listenelement auf *null*! Die leere Liste kann dann sowohl durch `count == 0` wie auch durch `first == null` erkannt werden – natürlich nicht von außen, sondern über einer in der Klasse `List` noch zu definierende Methode (`isEmpty()`).

- Die Methode *add()* – siehe Abb. 10.7, S. 10.7)

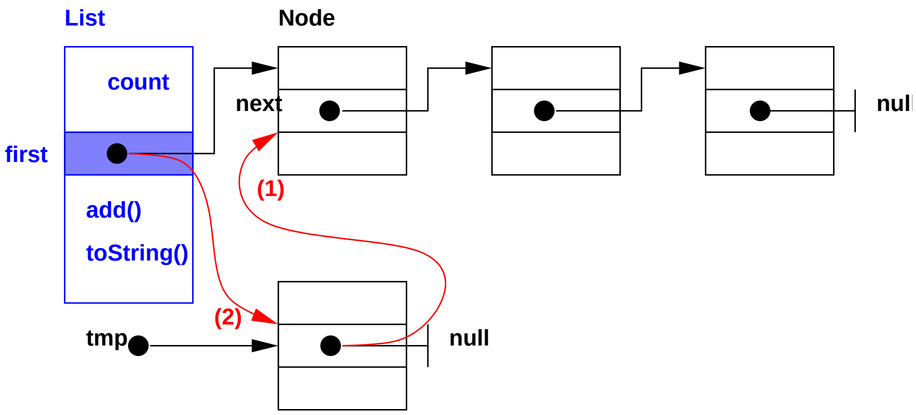


Abbildung 10.7: Element am Anfang der Liste einfügen

Man beachte die Reihenfolge der Referenz-Zuweisungen!

- Zeilen 25-39

  Es ist fast immer sinnvoll, für ein Datenobjekt eine Methode **toString()** zur Verfügung zu stellen. Im vorliegenden Fall sollen die Listenelemente in einem String zur Verfügung gestellt werden: die Elemente selbst sind durch Komma getrennt aufgelistet und das Ganze wird in runden Klammern gefasst.

  NB: Unabdingbar ist, dass der Inhalt der Listenelemente (das Feld *content*) auch eine Methode *toString()* besitzt!

  Um den String aufzubauen, benötigt man einen **variablen String** – dieser Typ wird über die Klasse **StringBuffer***StringBuffer* bereitgestellt. Diese Klasse besitzt selbst einige Methoden, von denen wir hier zwei benötigen:

  - **append()**: fügt einen konstanten String an den variablen String an!
  - **toString()**: wandelt den variablen String in einen konstanten Strin

- Zeilen 28-35

  Die typische Struktur, eine Liste vom Anfang her zu durchlaufen!

Programm 10.2 (S. 238) zeigt eine einfache Anwendung der obigen Klasse.

Programm 10.2: Anwendung von Programm 10.1, S. 236 (*SimpleList/TestList.java*)

```
1  import IOulm.*;
2
3  public class TestList {
4     public static void main(String[] args) {
5        List l = new List();;
6        int i;
7        while( Urc.readInt() ) {
8           i = Urc.getInt();
9           Integer el = new Integer(i);
10          l.add(el);
11       }
12       Write.Line("Liste: " + l);
13    }
14 }
```

Anwendung:

```
euclid$ cat daten
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
euclid$ java TestList < daten
Liste: (15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
euclid$
```

Anmerkung:

- Zeile 9

  Wir hatten das schon einmal: der Basistyp *int* muss in eine Integer-Klasse umgesetzt werden – diese hat natürlich eine *toString()*-Methode!

Einschub: Generische Methoden

In Java ist jede Klasse automatisch eine Unterklasse der Klasse **Object**. Dies erfolgt implizit, detaillierte Kenntnis des Unterklassenkonzepts ist zum Verständnis zunächst noch nicht notwendig!

Datentypen, wie Stacks und Queues (in einem früheren Kapitel behandelt), aber auch die hier behandelten Listen heißen **generisch** (bezüglich ihrer Elementtypen). So lassen sich aus eine Liste mit Elmentyp *Object* Listen über allen möglichen Klassentypen erzeugen, da die Zuweisung `Object data = d;` für jedes *d* eines Klassentyps gültig ist und da die Objekte zur Laufzeit wieder zu ihrer Objektklasse spezialisiert werden können. Für die elementaren Datentypen gibt es auch jeweils eine **Hüllklasse** (*wrapper class*); damit können auch diese als Elemente in Listen verwendet werden (siehe Programm 10.2, S. 238).

Mit dem Konzept des generischen Programmierens verfolgt man das Ziel, Code zu erzeugen, der für unterschiedliche Datentypen verwendet werden kann.

Generische Methoden:

Zu den generischen Datentypen gehören auch generische Methoden. Diese arbeiten auf allen Spezialisierungen des generischen Typs, müssen dazu aber zur Übersetzungszeit oder auch erst zur Laufzeit spezialisiert werden.

So ist die Methode **toString()** eine in **Object** definierte Methode, die standardmäßig (im wesentlichen) die Objektreferenz als String ausgibt. Subtypen (also alle Klassen) können *toString()* neu definieren (überschreiben), um sich selbst in Strings verwandeln zu können (siehe Programm 10.1, S. 236).

Die Subtypen sollten alle eine Methode *toString()* mit **derselben** Signatur aber unterschiedlichem Programmtext haben (für die Wrapperklassen ist dies der Fall!). Welcher Programmtext aufgerufen wird, hängt vom Subtyp des konkreten Datenelements ab. Dies kann zur Laufzeit erkannt werden und könnte zum Beispiel für jedes Datenelement in unserer Liste unterschiedlich sein. Zu dem Zeitpunkt, an dem die generische Ausgabefunktion geschrieben wird, müssen noch nicht einmal alle Subtypen bekannt sein, auf denen später *toString()* aufgerufen wird. Es genügt, dass die Signatur bekannt ist und dass **jedes Objekt** eine passende Implementierung von *toString* hat!

Generische Vergleiche:

Viele Algorithmen zum Suchen und Sortieren können als generische Algorithmen implemtiert werden (was wir zunächst noch nicht gemacht hatten). Dazu müssen sie aber durch einen Vergleichsoperator parametrisiert werden!

In Java wird hierzu ein Interface **Comparable** im Paket *java.lang* bereitgestellt. Darin ist eine Vergleichsmethode **compareTo(Object a)** wie folgt spezifiziert:

compareTo() liefert eine Zahl

- kleiner 0, falls das Objekt, auf dem *compareTo()* läuft, kleiner als *a* ist,
- gleich 0, falls das Objekt, auf dem *compareTo()* läuft, gleich *a* ist,
- größer 0, falls das Objekt, auf dem *compareTo()* läuft, größer als *a* ist.

Hat *a* nicht den gleichen Typ, so wird eine Ausnahme erzeugt (dazu später mehr, also zuvor achtgeben!)

In den generischen Algorithmen werden Variablen vom Referenztyp *Comparable* verwendet. Bei deren Instantiierung zeigt eine Referenzvariable auf ein Objekt einer Klasse, die das Interface *Comparable* implementiert. Die Hüllklassen der elementaren Datentypen implementieren dieses Interface!

Wenn wir später mit generischen sortierten Listen arbeiten wollen, so brauchen wir den Elementvergleich. Im Programm 10.3 (S. 240) wird ein Listenelement mit einer *compareTo()*-Methode definiert.

Programm 10.3: Klasse mit compareTo()-Methode (*SimpleList/MyListObject.java*)

```
1  public class MyListObject implements Comparable {
2     private String name;
3     private int alter;
4     public MyListObject(String name, int alter){
5        this.name = name; this.alter = alter;
6     }
7     public int compareTo(Object a){
8         return (this.alter - ((MyListObject) a).alter);
9     }
10    public String toString() {
11          return "[" + name + ", " + alter + "]";
12    }
13 }
```

Erläuterungen:

- Zeile 1: Die Klasse *MyListObject* muss das Interface *Comparable* implementieren!
- Zeilen 7-9: Gemäss Interface *Comparable* hat die Methode *compareTo()* einen Parameter vom Typ *Object*. In Zeile 8 wird daraus via *cast* ein Objekt vom aktuellen Typ *MyListObject*; der Vergleich erfolgt über die Altersangabe (Komponente *alter*).
- Zeilen 10-12: Die *toString()*-Methode ist immer gut!

Damit könnte die bisherige Listenklasse 10.1, S. 236, um eine Methode *search()* erweitert werden:

```
public boolean search(Comparable el){
  for(Node tmp = first; tmp != null; tmp = tmp.next)
       if (el.compareTo(tmp.content) == 0) return true;

  return false;
}
```

10.1.3 Implementierung eines dynamischen Stack

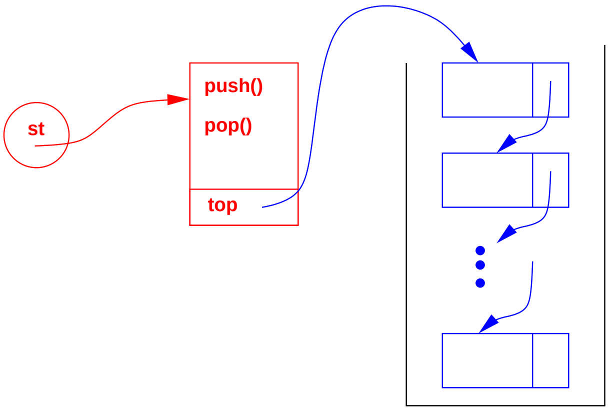


Abbildung 10.8: Stack als Liste

Dazu verwenden wir wieder unseren Abstrakten Datentyp bzw. das entsprechende Interface aus einem früheren Kapitel (siehe 10.4, S. 241).

Programm 10.4: Noch einmal das Interface zu einem Stack (*Stack/ADTSimpleStack.java*)

```
public interface ADTSimpleStack {
    public boolean isEmpty();
    public boolean isFull();
    public Object top();
    public boolean push(Object element);
    public boolean pop();
}
```

Zur Implementierung als Liste:

- Die Verkettung vom obersten Element zum nächsten darunterliegenden Element und so weiter ist von Vorteil: das Hinzufügen ist sehr einfach!
- Ein Verkettung von "unten nach oben" hieße beim Hinzufügen erst einmal an das Ende zu laufen und dort das neue Element anzufügen!
- Das Lesen wie auch das Entfernen des obersten Elements ist bei der dargestellten Verkettung ebenfalls trivial: Lesen – das gesuchte Element ist im direkten Zugriff, Entfernen – siehe Abb. 10.9, S. 242

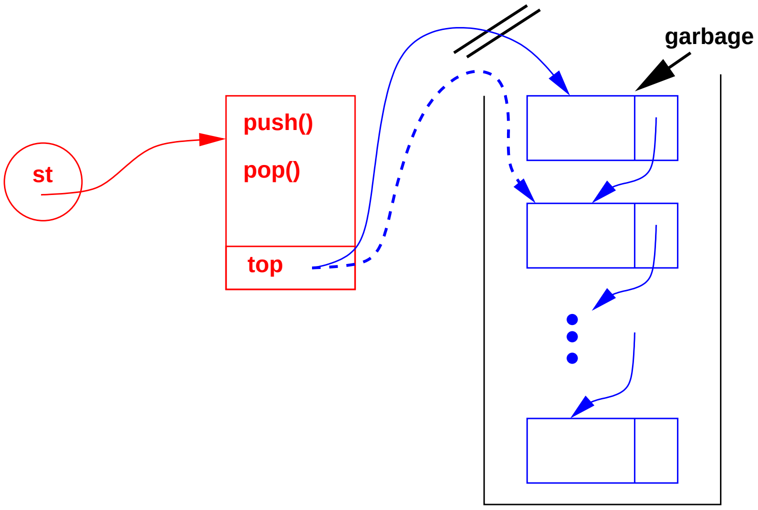


Abbildung 10.9: Stack als Liste – pop()

Programm 10.5: Dynamischer Stack als Implementierung des Stack-Interface (*Stack/DynStack.java*)

```java
public class DynStack implements ADTSimpleStack{

        private class Node {
              Object content;
              Node next;
              Node(Object o){ content = o; next = null; }
        }

        private int count; private Node top;

        public DynStack(){
           this.count = 0; this.top = null;
        }

        public boolean isEmpty() { return count == 0; }
        public boolean isFull () { return false; }

        public boolean push(Object el){
            Node tmp = new Node(el);
            tmp.next = top; top = tmp; count++;
            return true;
        }

        public Object top() {
              if (isEmpty()) return null;
              return top.content;
        }

        public boolean pop() {
              if ( isEmpty() ) return true;
              top = top.next; count--;
```

```java
            return true;
        }

        public String toString () {
            StringBuffer sb = new StringBuffer(); // veraenderliche String-Puffer
            sb.append("(");
            Node x = top;
            while ( x != null ) {
                sb.append(x.content.toString()); // Der Inhalt muss eine
                                                 // toString () Methode
                                                 // implementiert haben!
                x = x.next;
                if ( x != null ) sb.append(", ");
            }
            // abschliessen
            sb.append(")");
            return sb.toString();
        }
}
```

Programm 10.6: Testanwendung zu 10.5 (*Stack/TestStack.java*)

```java
import IOulm.*;
public class TestStack {
  public static void main(String[] args) {
     ADTSimpleStack st = new DynStack();
     char action =' '; int number = 0;
     while (true) {
        Write.String("add (a) / remove (r) / show (s) / print (p) / exit (e)? ");
        if ( Urc.readChar() ) {
           action = Urc.getChar();
           Urc.readString(); // remove rest of line, especially '\n'
           switch ( action ) {
              case 'a': Write.String("Give number: ");
                        if (Urc.readInt()){
                           number = Urc.getInt();
                           if ( st.push(new Integer(number)) ) Write.Line("done");
                           else Write.Line("Overflow!");
                        }
                        break;
              case 'r': if ( st.isEmpty() ) Write.Line("Empty!");
                        if (st.pop()) Write.Line("done");
                        break;
              case 's': if ( st.isEmpty() ) Write.Line("Empty!");
                        else Write.Line("got: " + st.top());
                        break;
              case 'p': Write.Line("Total: " + st );
                        break;
              case 'e': return;
              default:  Write.Line(" Illegal command");
           }
        } else return;
     }
  }
```

```
}
```

10.1.4 Implementierung einer dynamischen Queue

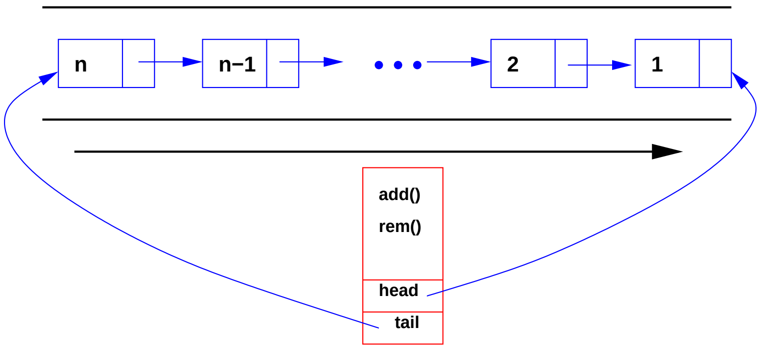


Abbildung 10.10: Queue als Liste – verkettet von "hinten nach vorne"

Beim Element mit Referenz *head* erfolgt der "Abgang" (Methode *rem()*), bei dem mit Referenz *tail* erfolgt der Zugang (Methode *add()*).

Bei der Verkettung in Abb. 10.10 (S. 245) ist das Hinzufügen (*add()*) sehr einfach, das Entfernen dagegen nicht: die Referenz des Elementes mit Eintrag "2" müsste auf *null* gesetzt werden – dazu müssen wir von "tail" dieses Element aufsuchen!

Alternative:

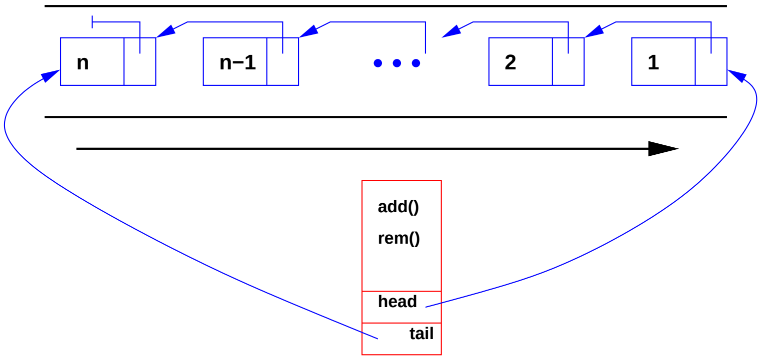


Abbildung 10.11: Queue als Liste – verkettet von "vorne nach hinten"

Entfernen:

```
head = head.next
```

Hinzufügen:

```
Node newNode = new Node(el);
tail.next = newNode;
tail = newNode;
```

Hierbei sind natürlich noch die Grenzfälle (leere Queue, nur ein Element in der Queue) zu betrachten

Grundlage ist wieder unser abstrakter Datentyp von früher (10.7), erweitert um eine Methode *get()*.

Programm 10.7: Abstrakter Datentyp Queue (*Queue/ADTSimpleQueue.java*)

```java
public interface ADTSimpleQueue {
    public boolean isEmpty ();
    public boolean isFull ();
    public Object remove ();
    public Object get ();
    public boolean append( Object element );
}
```

Programm 10.8: Implementierung des ADT Queue als dynamische Liste (*Queue/DynQueue.java*)

```java
public class DynQueue implements ADTSimpleQueue{

    private class Node {
        Object content ;
        Node next ;
        Node(Object o ){ content = o ; next = null ; }
    }

    private int count ; private Node head , tail ;

    public DynQueue(){
      this .count = 0; this .head = this . tail = null ;
    }

    public boolean isEmpty() { return count == 0; }
    public boolean isFull () { return false ; }

    public boolean append( Object el ){
        Node tmp = new Node(el );
        if ( isEmpty() ) head = tail = tmp;
        else { tail .next = tmp; tail = tmp; }
        count++;
        return true ;
    }

    public Object remove () {
        if ( isEmpty() ) return null ;
        Object o = head . content ;
        head = head . next ;
        if ( head == null) tail = null ;
        count--;
        return o ;
    }

    public Object get () {
        if ( isEmpty() ) return null ;
        Object o = head . content ;
        return o ;
    }
```

```java

        public String toString() {
            StringBuffer sb = new StringBuffer(); // veraenderliche String-Puffer
            sb.append("(");
            Node x = head;
            while ( x != null ) {
                sb.append(x.content.toString()); // Der Inhalt muss eine
                                                 // toString() Methode
                                                 // implementiert haben!
                x = x.next;
                if ( x != null ) sb.append(", ");
            }
            // abschliessen
            sb.append(")");
            return sb.toString();
        }
}
```

Programm 10.9: Testprogramm zu 10.8 (*Queue/TestDynQueue.java*)

```java
import IOulm.*;
public class TestDynQueue{
  public static void main(String[] args) {
     ADTSimpleQueue q = new DynQueue();
     char action =' '; int number = 0;
     while (true) {
        Write.String("append (a) / remove (r) / print (p) / exit (e)? ");
        if ( Urc.readChar() ) {
            action = Urc.getChar();
            Urc.readString(); // Rest, insbesondere '\n' weglesen
            switch (action) {
              case 'a':
                    Write.String("Give number: ");
                    if (Urc.readInt()){
                       number = Urc.getInt();
                       if ( q.append(new Integer(number)) ) Write.Line("done");
                       else Write.Line("Overflow!");
                    }
                    break;
               case 'r':
                    if ( q.isEmpty() ) Write.Line("Empty!");
                    else Write.Line("got: " + q.remove());
                    break;
               case 'p':
                    Write.Line("Total: " + q);
                    break;
               case 'e': return;
               default: Write.Line(" Illegal action");
            }
        } else return;
     }
  }
}
```

Die einzige kleine Unschönheit ist, dass uns die Methode *toString()* in unserer Queue-Implementierung die Queue so ausgibt, dass der Anfang der Liste (Ende zum Wegnehmen) links und das Ende (Ende zum Anfügen) rechts steht.

Dies lässt sich aber ändern: man durchläuft die Liste rekursiv zum Ende durch und beim Rückwärtsgehen in der Rekursion wird der String zusammengefügt: 10.10, S. 248

Programm 10.10: Dynamische Queue – Liste rückwärts ausgeben (*Queue/reverse*)

```
StringBuffer sb;

private void reverse(boolean first, Node x) {
    if ( x == null) return;
    if (x != null) reverse(false, x.next);
    sb.append(x.content.toString());
    if (!first) sb.append(", ");
}

public String toString() {
    sb = new StringBuffer();
    sb.append("(");
    reverse(true, head);
    // abschliessen
    sb.append(")");
    return sb.toString();
}
```

10.1.5 Sortierte Listen

- Bei sortierten Listen besitzt jedes Element einen Schlüssel, für den es eine vollständige Ordnungs-Relation $\leq$ gibt.
- Eine Ordnungs-Relation $\leq$ ist vollständig, wenn
  - $a \leq a$ (Reflexivität)
  - $a \leq b \wedge b \leq a \Rightarrow a = b$ (Antisymmetrie)
  - $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (Transitivität)

Methoden:

- `void insertElement(Object element)`

  fügt *element* so ein, dass alle Elemente weiterhin in der Liste sortiert bleiben – da hierzu die Methode *compareTo()* verwendet werden muss, sollte die Signatur so gewählt werden:

  `void insertElement(Comparable element)`

- `void deleteElement(Object element)`

  löscht *element*, so dass alle verbleibenden Elemente weiterhin in der Liste sortiert sind – hier gilt dasselbe wie oben, daher

  `void deleteElement(Comparable element)`

- `Object getFirstElement()`

  liefert das erste Element aus der Liste (ohne es zu entfernen)

- `void iterate()`

  iterate setzt den *cursor* auf das erste Element der Liste. Mit *iterate()* und *getElement* ist es möglich, sich alle Elemente der Liste in sortierter Reihenfolge anzusehen

- `Object getElement()`

  liefert das Element, auf das der aktuelle *cursor* – initial gesetzt von der Methode *iterate()* – zeigt und setzt den *cursor* auf das nachfolgende Element

- `int length()`

  liefert die Anzahl der Elemente in der Liste

Für die folgenden Implementierungen verwenden wir zum Test die in Programm 10.11 (S. 249) dargestellte Klasse.

Programm 10.11: Elementklasse für die sortierte Liste (*SortedList/MyObject.java*)

```java
public class MyObject implements Comparable {
    private String name;
    private int key;
    public MyObject(String name, int key){
        this.name = name; this.key = key;
    }
    public int compareTo(Object o){
        return (this.key - ((MyObject) o).key);
    }
    public String toString() {
        return "[" + key + ", " + name + "]";
    }
    public void changeName(String name){
        this.name = name;
    }
}
```

Die Listenstruktur ist wie in Abb. 10.12 (S. 250) dargestellt einfach verkettet.

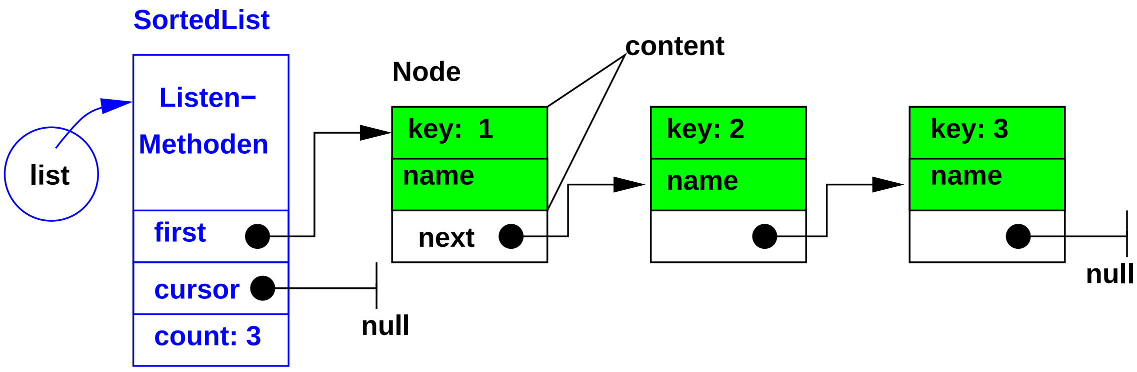


Abbildung 10.12: Sortierte Liste – Struktur

Einfügen eines Elementes:

- Ist die Liste leer (*count* == 0), so muss ein neuer Knoten erzeugt und der Inhalt eingetragen werden; danach wird *first* auf diesen Knoten gesetzt
- Ist die Liste nicht leer, so muss erst die Stelle gesucht werden, an der das neue Element entsprechend der Sortierung einzufügen ist. Das Einfügen **nach** einem Knoten ist einfach (siehe Abb. 10.13, S. 250).

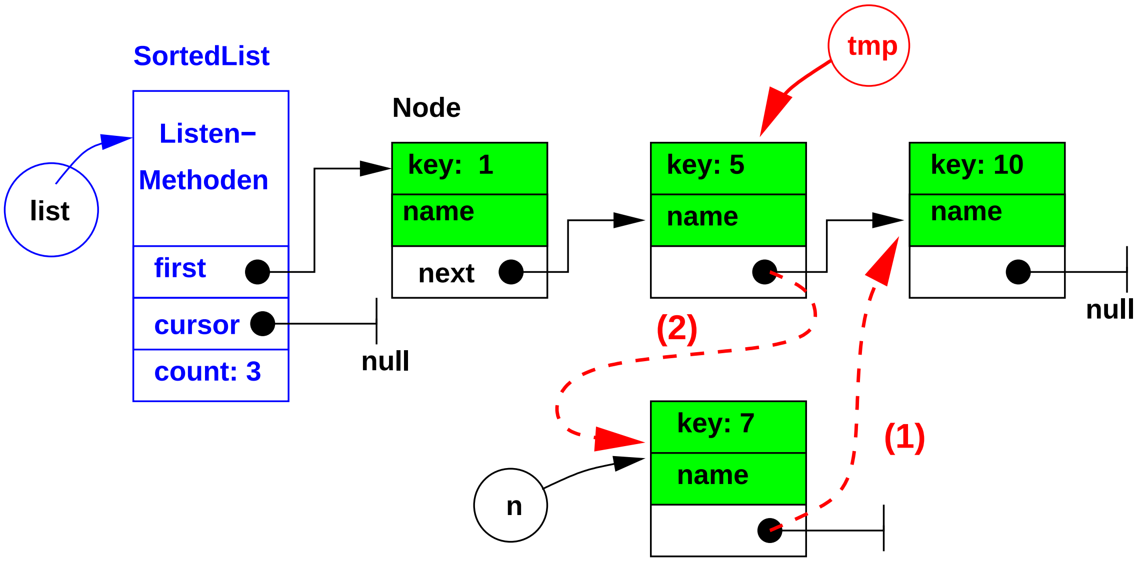


Abbildung 10.13: Sortierte Liste – Element einfügen

– Wir suchen zunächst die passende Position: das größte Element in der Liste, das kleiner ist als das einzufügende Element

```
private Node search(Comparable el)
// sucht nach dem groessten Element, das kleiner als el ist
```

- Da an dieser Stelle die Liste nicht leer ist, gibt es als Ergebnis zwei Möglichkeiten:
  * Das neue Element ist kleiner als alle Listenelemente – dazu vergleichen wir das neue Element einfach mit dem ersten Listenelement. In diesem Fall soll *search* die *null*-Referenz liefern, das Einfügen geschieht dann am Anfang der Liste (trivial) und erfolgt wie im Fall der leeren Liste.
  * Das neue Element muss nicht am Anfang eingefügt werden – dann liefert uns *search* eine Referenz auf das Element, **nachdem** das neue Element eingefügt werden muss. Das geschieht ganz leicht wie in Abb. 10.13 (S. 250) beschrieben.

Programm 10.12 (S. 252) zeigt eine erste Implementierung.

Programm 10.12: Sortierte Liste – Methode insertElement() (*SortedList/SortedList1.java*)

```
public class SortedList1 {

    private class Node {
        Object content; Node next;
        Node(Object o){ content = o; next = null; }
    }

    private int count;
    private Node first, cursor;

    public SortedList1 (){
        this.count = 0; first = cursor = null;
    }

    // sucht nach dem groessten Element, das kleiner als el ist
    private Node search(Comparable el){
        Node tmp = first;
        if (count == 0) return null; // leere Liste
        if (el.compareTo(tmp.content) < 0) return null;
        while (tmp.next != null && el.compareTo(tmp.next.content) > 0)
            tmp = tmp.next;
        return tmp;
    }

    public void insertElement(Comparable el){
        Node n = new Node((Object) el);
        Node tmp = search(el);
        if ( tmp == null) {
            n.next = first; first = n;
        } else {
            n.next = tmp.next; tmp.next = n;
        }
        count++;
    }

    public int length() { return count; }

    public String toString() {
        StringBuffer sb = new StringBuffer(); //
        sb.append("(");
        Node x = first;
        while ( x != null ) {
            sb.append(x.content.toString());
            x = x.next;
            if ( x != null ) sb.append(", ");
        }
        sb.append(")");
        return sb.toString();
    }
}
```

Löschen eines Elementes:

- Ist die Liste leer, so ist nichts zu tun!
- Mit der Methode *search*() suchen wir den Vorgänger des zu löschenden Elements.

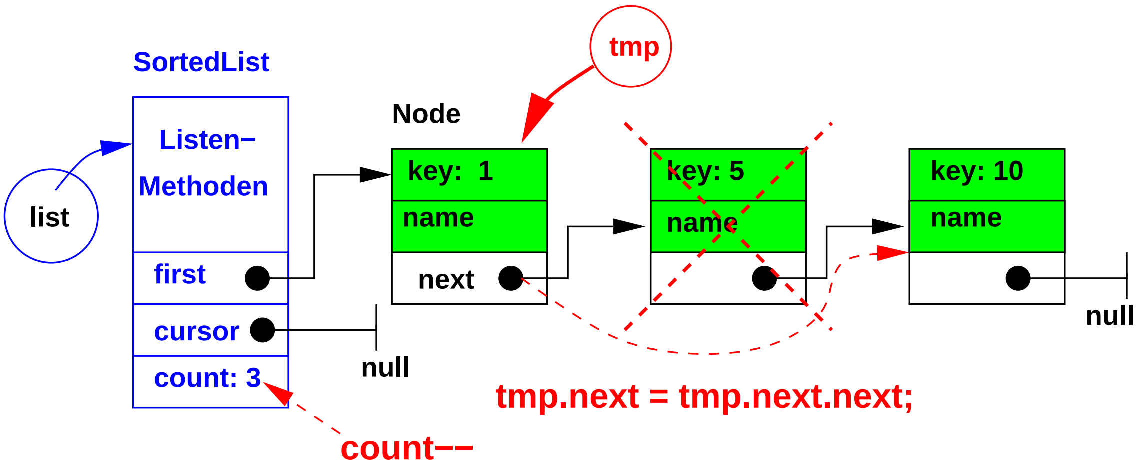


Abbildung 10.14: Sortierte Liste – Element löschen

Die weiteren Schritte sind wie in Abb. 10.14 (S. 253) dargestellt zu vollziehen.

Programm 10.13: Sortierte Liste – Methode deleteElement() (*SortedList/delete.frag*)

```
public void deleteElement (Comparable el ) {
    if ( count == 0 ) return;
    Node tmp = search ( el );
    if ( tmp == null ) tmp = first ;
    if ( el .compareTo( first . content ) == 0 ) {
        first = first .next ; count−−; return;
    }
    while (tmp.next != null)
        if ( el .compareTo(tmp.next . content ) == 0 ){
            tmp.next = tmp.next .next ; count−−; return;
        } else { tmp = tmp.next ; }
}
```

Erstes Element holen:

```
public Object getFirstElement() {
   if ( first == null ) return null;
   else return first.content;
}
```

Iterator setzen:

```
public void iterate() { cursor = first; }
```

Element holen:

```
public Object getElement() {
    if (cursor == null) return null;
    else { Node tmp = cursor;
           cursor = cursor.next;
           return tmp.content;
    }
}
```

10.2 Bäume

10.2.1 Konzepte, Terminologie

- Wir hatten Bäume bereits im Zusammenhang mit *Heapsort* eingeführt, allerdings in eher graphentheoretischer Sichtweise.

- Bäume gehören zu den wichtigsten Datenstrukturen.

- Es gibt sie in sehr vielen Varianten, wovon wir nur einige im Rahmen dieser Vorlesung kennenlernen werden.

- Eine allgemeine Definition von Bäumen nach Knuth:
  Ein **Baum** ist eine endliche Menge T von einem oder mehreren Knoten, so dass
  - es einen speziell ausgezeichnenten Knoten gibt, der als **Wurzel** bezeichnet wird: **root**(T); und
  - die verbleibenden Knoten (ohne die Wurzel) in $m \geq 0$ disjunkte Mengen $T_1, \ldots, T_m$ aufgeteilt sind, wobei jede dieser Mengen wiederum ein Baum ist. Die Bäume $T_1, \ldots, T_m$ werden die **Unterbäume** der Wurzel genannt.

- Diese Definition ist rekursiv. Allerdings ist es kein Zirkelschluss, da die Gesamtmenge der Knoten endlich ist und sich die Menge der Knoten bei jedem Rekursionsschritt um mindestens eins vermindert, so dass am Ende Bäume mit nur einem Knoten verbleiben.

- Bäume lassen sich auch nicht-rekursiv definieren, jedoch passt Rekursion zur Natur der Bäume, da viele Algorithmen auf Bäumen rekursiv arbeiten.

- Hinweis: Diese Seite und die folgenden lehnen sich sehr eng an das Kapitel 2.3 von “The Art of Computer Programming”, Band 1, von Donald E. Knuth an. Insbesondere wurden auch einige Abbildungen mehr oder weniger direkt übernommen.

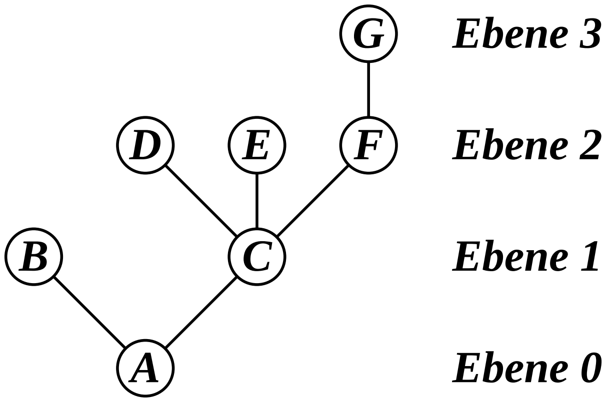


Abbildung 10.15: Ein erster Baum

- Jeder Knoten eines Baumes ist die Wurzel eines der Unterbäume innerhalb des Baumes.

- Die Zahl der Unterbäume eines Knotens wird als **Grad** bezeichnet.

- Ein Knoten mit dem Grad 0 wird als *Endknoten* oder als **Blatt** bezeichnet.

- **Verzweigungsknoten** sind Knoten mit einem Grad > 0.

- Die **Ebene** eines Knotens in Bezug auf T wird rekursiv definiert:

  - Die Ebene von **root**(T) ist 0.

  - Die Ebene eines anderen Knotens ist um 1 größer als die Ebene des Knotens, der die Wurzel des unmittelbar übergeordneten Unterbaumes ist.

Geordnete und orientierte Bäume

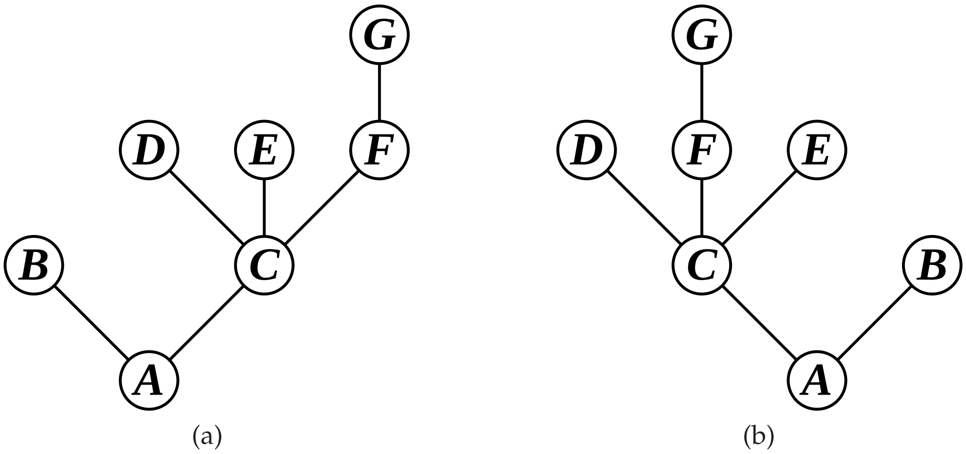


Abbildung 10.16: Zwei unterschiedliche Bäume?

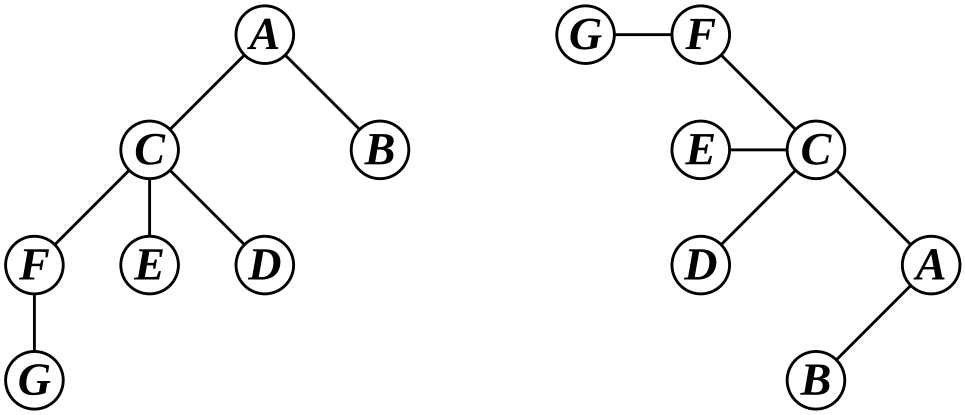
- Wenn die relative Ordnung der Unterbäume $T_1, \ldots, T_m$ relevant ist, wird von einem **geordneten Baum** gesprochen. Entsprechend ist dann beispielsweise T_2 der zweite Unterbaum der Wurzel.

- Wenn hingegen die Ordnung keine Rolle spielt, wird von **orientiertem Baum** gesprochen.

- Die beiden Bäume im obigen Beispiel sind verschieden, wenn sie als geordnete Bäume betrachtet werden, jedoch äquivalente Darstellungen, wenn sie als orientierte Bäume zu verstehen sind.

- Normalerweise sind alle Bäume geordnet, da sich das relativ naturgemäß aus der Datenstruktur im Rechner ergibt.

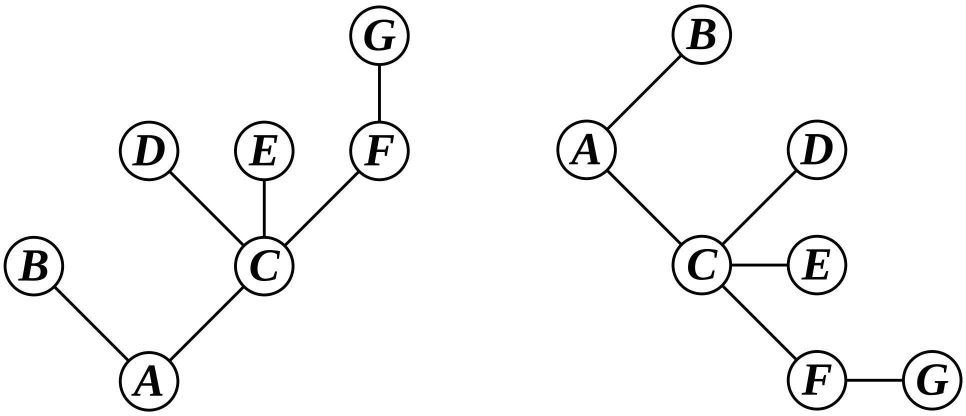
Darstellung von Bäumen:



(a) (b)

Abbildung 10.17: von Oben nach unten / von rechts nach links wachsende Bäume?

- Bäume können in vielerlei Weisen dargestellt werden.
- Da häufig von "über", "höher als" bei Knoten gesprochen wird, empfiehlt es sich, von einer einheitlichen Darstellungsweise auszugehen.
- Obwohl Bäume in der Natur von unten nach oben wachsen, hat sich diese Vorstellung bei den Informatikern nicht durchgesetzt. Stattdessen ist üblicherweise die Wurzel ganz oben (siehe Diagramm oben links) und von dort wächst der Baum nach unten.



(a) (b)

Abbildung 10.18: von unten nach oben / von links nach rechts wachsende Bäume?

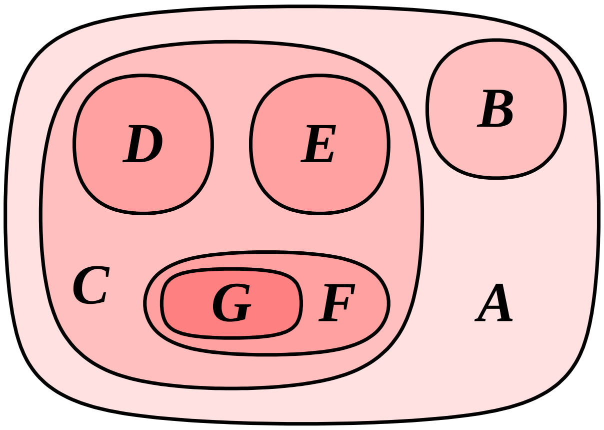


Abbildung 10.19: Baum als Menge

- Es gibt zahlreiche weitere Möglichkeiten, Bäume darzustellen:
  - Die Darstellung in Form verschachtelter Mengen ist insbesondere für orientierte Bäume geeignet.
  - Für die Eingabe ist eine Syntax denkbar, die mit Klammern die Verschachtelung vorgibt: `(A(B)(C(D)(E)(F(G))))`
  - Bäume entsprechen Hierarchien und entsprechend können auch geordnete Bäume so dargestellt werden, daß für jeden Knoten eine entsprechend eingerückte Zeile verwendet wird (je höher die Ebene, desto weiter ist es einzurücken).

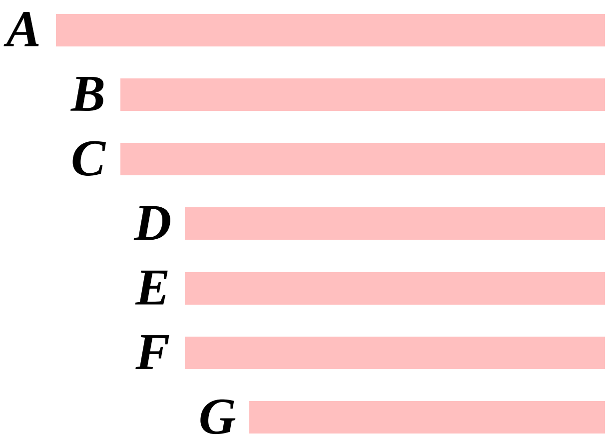


Abbildung 10.20: Baum in eingerückter Form

Familienbäume: Nachfahren

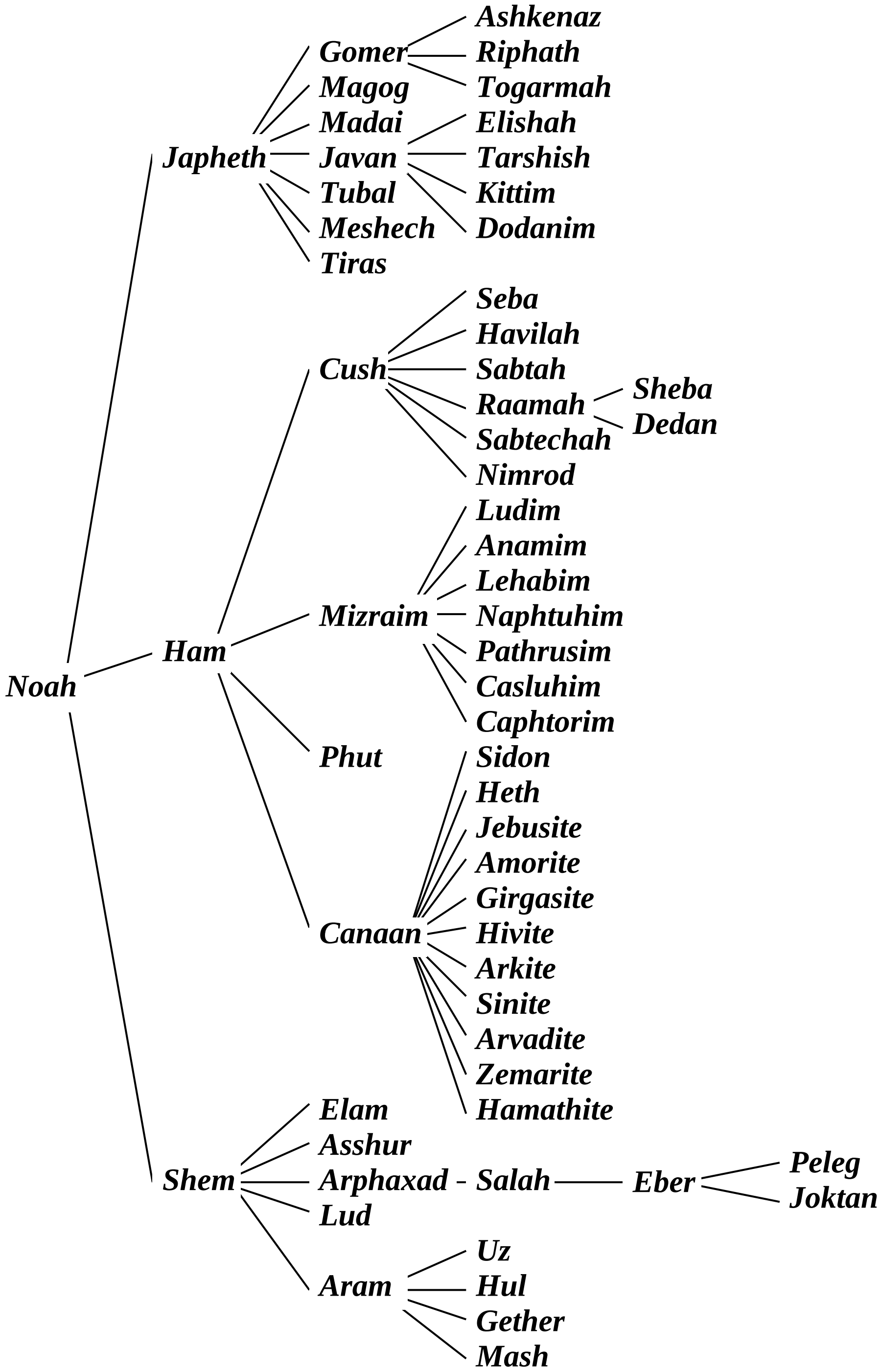


Abbildung 10.21: Baum der Nachfahrenschaft

Familienbäume: Vorfahren

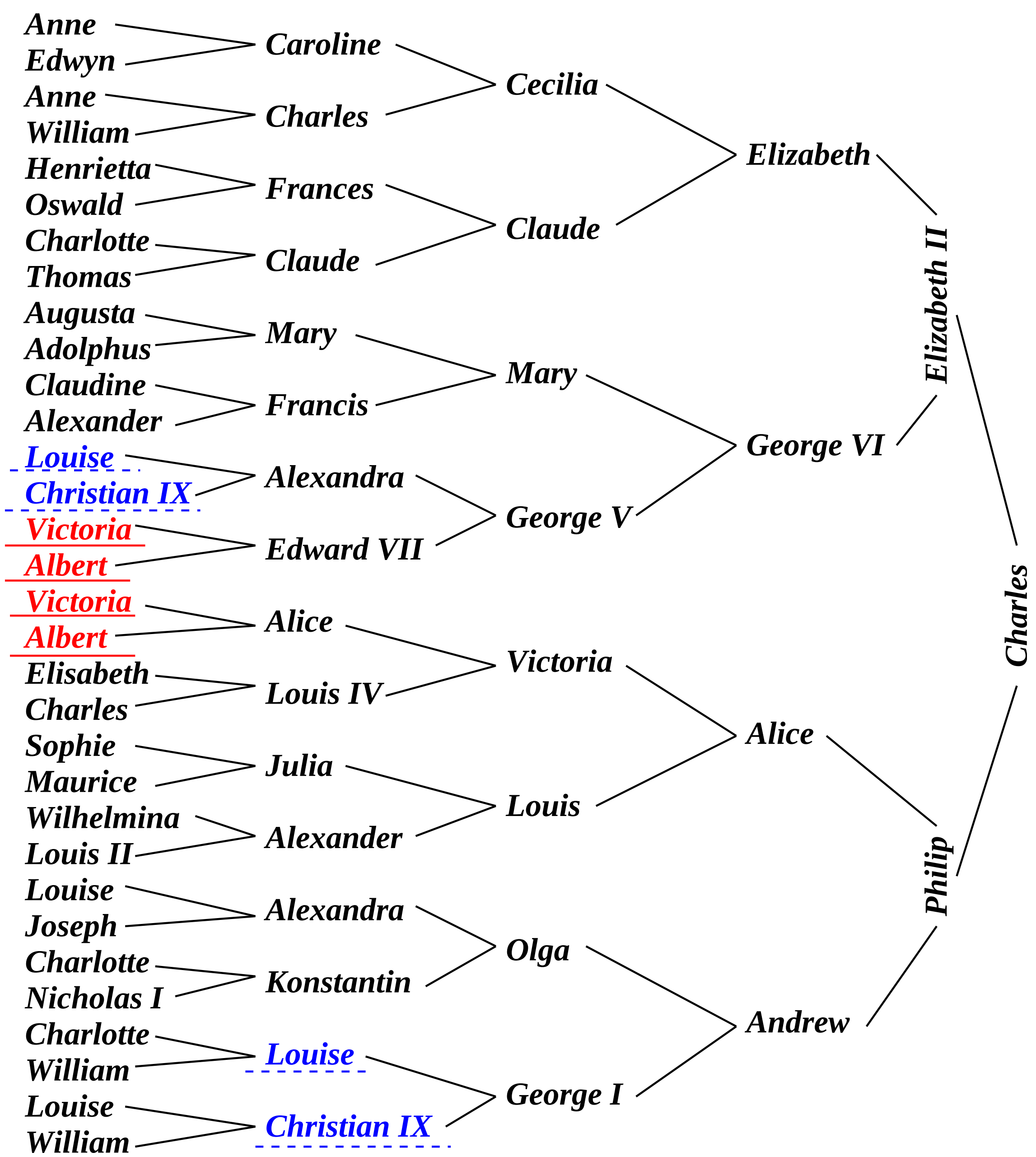


Abbildung 10.22: Stammbaum

- Viele Bezeichnungen bei Bäumen lehnen sich an die der Familienbäume an.

- Es gibt zwei Varianten von Familienbäumen:
  - Ausgehend von einer Wurzel werden alle Nachfahren (bis zu einer gewissen Ebene) angegeben.
  - Ausgehend von einer Wurzel werden alle Vorfahren (bis zu einer gewissen Ebene) aufgeführt.

- Familienbäume sind nicht notwendigerweise Bäume im Sinne der Definition, wenn es zu Eheschließungen innerhalb der Verwandtschaft kommt. So zählt sowohl Elizabeth II als auch Philip das Ehepaar von Königin Victora und Prinz Albert zu ihren Ururgroßeltern (siehe rote Farbe – durchgezogen unterstrichen). Und auch König Christian IX und Königin Louise zählen beidseitig zu den Vorfahren (siehe blaue Farbe – gestrichelte Linie).

- Das Problem lässt sich beheben, wenn die Knoten nicht die jeweiligen Personen repräsentieren, sondern die jeweilige Rolle (z.B. als Großvater mütterlicherseits).

Von Familienbäumen abgeleitete Terminologie

(vgl. 10.21, S. 260)

- Die Wurzel wird als *Elternteil* der Wurzeln ihrer Teilbäume bezeichnet.

- Letztere sind *Kinder* ihres Elternteils bzw. *Geschwister* untereinander.

- Die Begriffe *Vorfahre* und *Nachfahre* bezeichnen Beziehungen, die sich über mehrere Ebenen erstrecken können.

Binärbäume

- Ein binärer Baum ist eine endliche Menge von Knoten, die entweder leer ist oder aus einer Wurzel und den Elementen zweier disjunkter binärer Bäume besteht, die linker und rechter Teilbaum der Wurzel genannt werden.

- Binäre Bäume sind **kein** Spezialfall eines generellen Baumes:

  - So ist die leere Menge als binärer Baum zulässig, jedoch nicht als allgemeiner Baum, der immer eine Wurzel besitzt.
  - Die zwei binäre Bäume in Abb. 10.23, S. 263, sind nicht identisch.

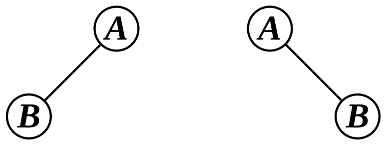


Abbildung 10.23: Zwei Binärbäume

  In dem einen Fall liegt nur ein linker Teilbaum vor, in dem anderen Fall nur ein rechter.

- Der Familienbaum mit den Vorfahren (10.22, S. 261) ist ein binärer Baum. Ein weiteres Beispiel findet sich bei den Ausscheidungskämpfen bei Tennisturnieren: Der Gewinner ist die Wurzel, auf der 1. Ebene sind die Teilnehmer des Finales, auf der 2. Ebene die der Halbfinale usw.

Repräsentierung binärer Bäume:

Programm 10.14: Definition eines Binärbaums (*BinTrees/BinTree.java*)

```java
public class BinTree {

        private class Node {
            Object content;
            Node left, right;
            Node(Object o){ content = o; left = right = null; }
        }

        private int count;
        private Node root;

        public BinTree(){
            count = 0;
            root = null;
        }
}
```

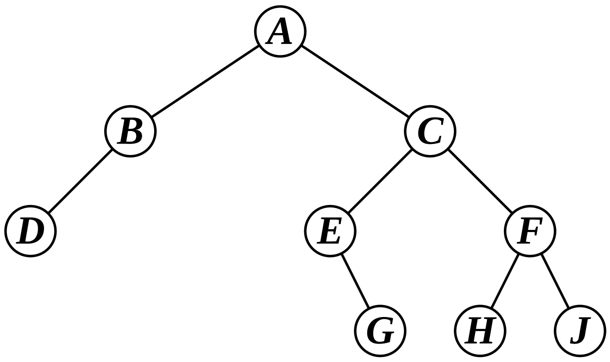


Abbildung 10.24: Binärbaum: Beispiel

Die Repräsentierung des Binärbaumes in Abb. 10.24 (S. 264) ist in Abb. 10.25 (S. **??**) gezeigt.

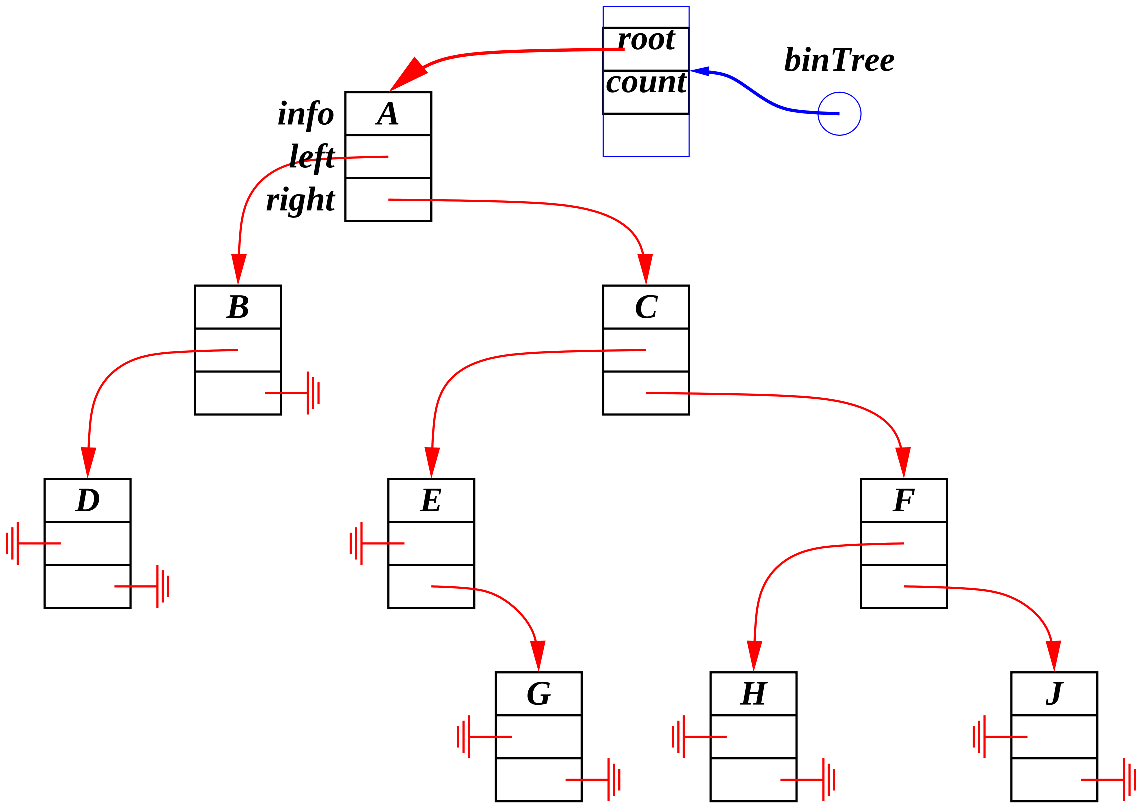


Abbildung 10.25: Binärbaum: Repräsentation

Ausgeglichene Binärbäume

- Ein Baum T ist **in der Höhe ausgeglichen** (*height-balanced*), wenn entweder T leer ist oder gilt, dass

    - $\mid h(T_1) - h(T_2) \mid \leq 1$ und
    - sowohl T_1 als auch T_2 in der Höhe ausgeglichen sind.

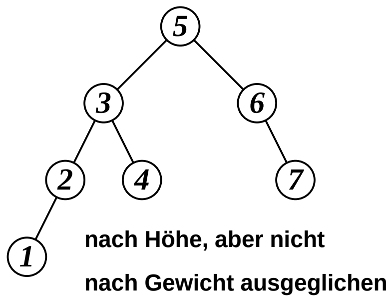
- Ein Baum T ist **nach dem Gewicht ausgeglichen** (vollständig ausgeglichen oder *weight-balanced*), wenn entweder T leer ist oder gilt, dass

    - $\mid \mathrm{card}(T_1) - \mathrm{card}(T_2) \mid \leq 1$[1] und
    - sowohl T_1 als auch T_2 nach dem Gewicht ausgeglichen sind.

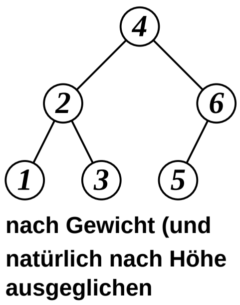
- Wenn ein Baum T nach dem Gewicht ausgeglichen ist, so folgt daraus, dass er auch in der Höhe ausgeglichen ist.

- Nach dem Satz von Adelson-Velsky und Landis gilt für einen in der Höhe ausgeglichenen Baum T:
  $\lceil \log_2(n+1) \rceil \leq h(T) \leq 1.4404 \log_2(n+2) - 0.3277$

- Somit sind Suchpfade im höhenausgeglichene Bäume niemals um mehr als 45% länger im Vergleich zum optimalen Fall.



(a)



(b)

Abbildung 10.26: Ausgeglichene Binärbäume

[1] *card(T)*: Anzahl der Knoten im Baum T

10.2.2 Traversieren eines Binärbaumes

- Es gibt drei prinzipielle Möglichkeiten, einen binären Baum zu **traversieren**: *preorder*, *inorder* und *postorder*.
- Wenn der binäre Baum leer ist (d.h. die *root*-Referenz ist gleich **null**), gibt es bezüglich der Traverse nichts weiter zu tun.
- Ist der Baum nicht leer, so definieren sich die einzelnen Traversen wie folgt:

preorder	Besuche die Wurzel Traversiere den linken Teilbaum Traversiere den rechten Teilbaum
inorder	Traversiere den linken Teilbaum Besuche die Wurzel Traversiere den rechten Teilbaum
postorder	Traversiere den linken Teilbaum Traversiere den rechten Teilbaum Besuche die Wurzel

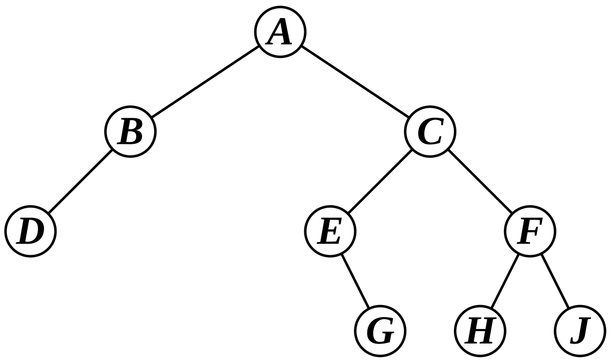


Abbildung 10.27: Traversieren

- Folgende Reihenfolgen ergeben sich, wenn die Knoten entsprechend den vorgestellten Traversen besucht werden:

preorder	A B D C E G F H J
inorder	D B A E G C H F J
postorder	D B G E H J F C A

10.2.3 Aufbau eines ausgeglichenen Binärbaums

Wenn die Zahl n der Knoten bekannt ist, so lässt sich ein vollständig ausgeglichener Baum sehr leicht aufbauen:

```
Node buildTree(int n) {
   if (n <= 0 ) return null;
   nLeft = n / 2; nRight = n - nLeft -1;
   Node newNode = new Node();
   newNode.left = buildTree(nLeft);
   newNode.right = buildTree(nRight);
   return newNode;
}

root = buildTree(n);
```

Diese Vorgehensweise lehnt sich an die Preorder-Traversierung an!

Diesen Baum könnte man anschließend wie auch immer traversieren und als Aktion "Besuche die Wurzel" zum Beispiel einen Inhalt in den Knoten eintragen. Eine *toString*()-Methode lässt sich leicht von jeder der drei Traversierungsarten ableiten. Allerdings ist im folgenden die Struktur des Baumes immer interessant, so wir eine zweidimensionale Ausgabe für nützlicher halten. An einfachsten geht dies, lässt man den Baum von links nach rechts wachsen. Dazu wählt man eine Traversierung ähnlich der Inorder-Traversierung, allerdings "rechts" und links" vertauscht. Je tiefer in der Rekursion eine Knoten "besucht" wird, desto weiter eingerückt muss er ausgegeben werden.

Programm 10.15: Aufbauen und Ausgeben ein Binärbaums (*BinTrees/BalBinTree.java*)

```java
import IOulm.*;

public class BalBinTree {

        private class Node {
             Object content;
             Node left, right;
             Node(Object o){ content = o; left = right = null; }
        }

        private int count;
        private Node root;

        public BalBinTree (){
           count = 0;
           root = null;
        }

        private Node build (int number) {
           Node newNode; int numberLeft, numberRight;
           if ( number <= 0 ) return null;
           numberLeft = number / 2;
           numberRight = number - numberLeft - 1;
           count++;
           newNode = new Node(new Integer(count)); //some content
           newNode.left = build (numberLeft);
           newNode.right = build (numberRight);
           return newNode;
       }

        public void buildBal (int number) {
              root = build (number);
        }

        private void printTree (Node r, int indent) {
             int i;
             if ( r != null) {
                  printTree (r.right, indent + 5);
                  /**** Besuche die Wurzel *****/
                  for( i = 1; i <= indent; i++)
                      Write.Char(' ');
                  Write.Line("" + r.content);
                  /***************************/
                  printTree (r.left, indent + 5);
             }
        }

        public void print (int indent) {
              printTree (root, indent);
        }

}
```

Programm 10.16: Anwendung von Programm 10.15 (*BinTrees/TestBalBinTree.java*)

```
import IOulm.*;

public class TestBalBinTree {
   public static void main(String[] args) {
      BalBinTree t = new BalBinTree();
      t.buildBal(20);
      t.print(4);
   }
}
```

```
swg@sperber$ java TestBalBinTree
                   20
              17
                   18
                        19
         12
                   16
              13
                   14
                        15
    1
                   11
              8
                   9
                        10
         2
                   6
                        7
              3
                   4
                        5
swg@sperber$
```

Statt beim Aufbauen des Baumes bereits etwas einzutragen, könnte man den Inhalt zunächst freilassen und einem zweiten Traversierungsdurchgang die Knoteninhalte eintragen:

Programm 10.17: Variante zu 10.15 (S. 268) (*BinTrees/BalBinTree1.java*)

```java
import IOulm.*;

public class BalBinTree1 {

        private class Node {
             Object content;
             Node left, right;
             Node(Object o){ content = o; left = right = null; }
        }

        private int count;
        private Node root;

        public BalBinTree1 (){
           count = 0;
           root = null;
        }

        private Node build (int number) {
           Node newNode; int numberLeft, numberRight;
           if ( number <= 0 ) return null;
           numberLeft = number / 2;
           numberRight = number - numberLeft - 1;
           count++;
           newNode = new Node(null); //some content
           newNode.left = build (numberLeft);
           newNode.right = build (numberRight);
           return newNode;
       }

        public void buildBal (int number) {
              root = build (number);
        }

        int i = 0;
        private void fillTreeInorder (Node r){
             if ( r == null ) return;
              fillTreeInorder (r.left);
             i++;
             r.content = new Integer(i);
              fillTreeInorder (r.right);
        }

        public void fillInorder () {
              fillTreeInorder (root);
        }

        private void printTree (Node r, int indent) {
             int i;
             if ( r != null) {
```

```
                printTree (r.right, indent + 5);
                /**** Besuche die Wurzel ******/
                for( i = 1; i <= indent; i++)
                    Write.Char(' ');
                Write.Line("" + r.content);
                /****************************/
                printTree (r.left, indent + 5);
            }
        }

        public void print (int indent) {
                printTree (root, indent);
        }

}
```

Programm 10.18: Anwendung von Programm 10.17 (BinTrees/TestBalBinTree1.java)

```
import IOulm.*;

public class TestBalBinTree1 {
    public static void main(String[] args) {
        BalBinTree1 t = new BalBinTree1();
        t.buildBal(20);
        t.fillInorder();
        t.print(4);
    }
}
```

```
swg@sperber$ java TestBalBinTree1
                   20
              19
                   18
                        17
         16
                   15
              14
                   13
                        12
    11
                   10
              9
                   8
                        7
         6
                   5
                        4
              3
                   2
                        1
swg@sperber$
```

10.2.4 Breitentraversierung

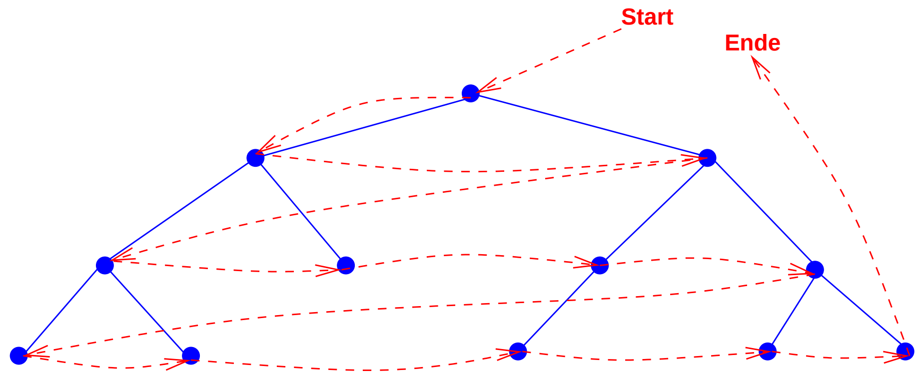


Abbildung 10.28: Leveltraversierung

Bei diesem Verfahren geht man nicht wie beim Preorder-Traversieren zuerst in die Tiefe, sondern die Strategie lautet hier: Breite zuerst (**breadth first** statt **depth first**)!

Anwendung:

- Suchbäume, bei denen die Knoten Spielsituationen und die Kanten mögliche Spielzüge darstellen
- Gesucht: eine Gewinnsituation, die in möglichst wenigen Spielzügen erreichbar ist
- Solche Suchbäume werden oft sehr tief und werden daher nur begleitend zur Suche Schicht für Schicht aufgebaut

Vorgehensweise bei der Breitensuche eines bereits vollständig aufgebauten Binärbaumes:

- alle Knoten einer Ebene müssen “gemerkt” werden
- für jeden Knoten müssen für den nächsten Durchlauf dessen Nachfolger gemerkt werden
- : ideale Datenstruktur: Queue
- Länge der Queue im schlimmsten Fall: $\frac{n}{2}$ bei einem vollständig ausgeglichenen Binärbaum mit n Knoten[2]

[2]bei den rekursiven Methode *pre-*, *in-*, *postorder* wird der Stack so groß wie die Tiefe des betrachteten Baumes

Programm 10.19: Breitensuche (*BinTrees/levelOrder*)

```
public void levelOrder (){
    DynQueue q = new DynQueue();
    DynQueue out = new DynQueue();
    q.append( root );
    while (!q.isEmpty()) {
      Node tmp = (Node) q.remove ();
      if (tmp != null) {
          out.append(tmp.content );
          q.append(tmp. left );
          q.append(tmp. right );
      }
    };
    Write.Line("Levelorder: " + out );
}
```

10.2.5 Sortierte Binärbäume

- Ein Tupel (T, v, V, R) ist ein **sortierter binärer Baum**
    - bezüglich der Wertefunktion $v : T \to V$ und
    - bezüglich der vollständigen Ordnungsrelation R für $V \times V$,

  wenn T ein binärer Baum ist, der entweder leer ist oder für den bezüglich des linken Teilbaums T_1 und des rechten Teilbaums T_2 gilt, dass

    - $v(\text{root}(T)) >_R v(t) \quad \forall\, t \in T_1$,
    - $v(\text{root}(T)) \leq_R v(t) \quad \forall\, t \in T_2$ und daß
    - sowohl (T_1, v, V, R) als auch (T_2, v, V, R) sortierte binäre Bäume sind.

- Gelegentlich ist es auch sinnvoll, darauf zu bestehen, dass

$$v(t_1) \neq_R v(t_2) \,\forall\, t_1, t_2 \in T, t_1 \neq t_2$$

  In diesem Falle ist die Wertefunktion $v : T \to V_T$ bijektiv, wobei
  $V_T = \{v \mid \exists t \in T : V(t) = v\}$, d.h. mit Hilfe eines Wertes $v \in V_T$ kann ein Knoten $t \in T$ eindeutig bestimmt werden.

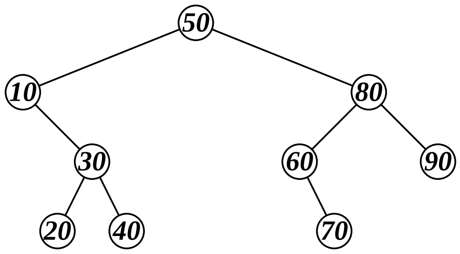


Abbildung 10.29: Sortierter Binärbäum

- Wenn ein sortierter binärer Baum in *inorder* traversiert wird, dann werden alle Werte in sortierter Form durchlaufen.
- Die Höhe $h : T \to \mathbb{N}_0$ eines Baumes definiert sich wie folgt: Falls T leer ist, dann ist $h(T) = 0$, ansonsten gilt
  $h(T) = max(h(T_1), h(T_2)) + 1$.
- $\lceil \log_2(n+1) \rceil \leq h(T) \leq n$
  mit $n = \text{card}(T)$ (Anzahl der Elemente in T).

Suche in einem sortierten Binärbaum

- Gegeben sei ein Wert $k \in V$ und gesucht wird ein Knoten $t \in T$ mit $v(t) = k$:

  1. Wenn T leer ist, dann ist der gesuchte Wert nicht vorhanden.
  2. Sei $t = \text{root}(T)$. Wenn $v(t) = k$, dann ist der gesuchte Wert gefunden.
  3. Es geht weiter mit dem ersten Schritt, wobei
     - $T = T_1$, falls $k < v(t)$, und
     - $T = T_2$, falls $k \geq v(t)$.

- Diese Suche hat einen Aufwand von $O(h(T))$.

Einfügen in einen sortierten Binärbaum

- Gegeben sei ein neues Element u mit $k = v(u)$, das in T einzufügen ist:

  1. Erzeuge einen neuen binären sortieren Baum U, der nur aus der Wurzel u besteht.
  2. Wenn T leer ist, dann ist T durch U zu ersetzen.
  3. Es geht weiter mit dem zweiten Schritt, wobei
     - $T = T_1$, falls $k < v(t)$, und
     - $T = T_2$, falls $k \geq v(t)$.

- Das Einfügen hat einen Aufwand von $O(h(T))$.

- Wenn Elemente in sortierter Reihenfolge eingefügt werden, degeneriert der Baum zur linearen Liste mit $h(T) = \mathrm{card}(T)$. Dieses "*worst case*"-Szenario läßt sich durch diverse Algorithmen vermeiden, die den Baum bei Bedarf umbauen (z.B. die Höhenausgeglichenheit sicherstellen).

Um ein neues Element einzufügen, muss der Knoten im Baum gesucht werden, an den das neue Element angehängt werden kann. Dieser Knoten hat maximal einen Nachfolger, angefügt werden muss entsprechend der Ordnungsrelation, und dieser Zeiger ist auf jeden Fall *null*!

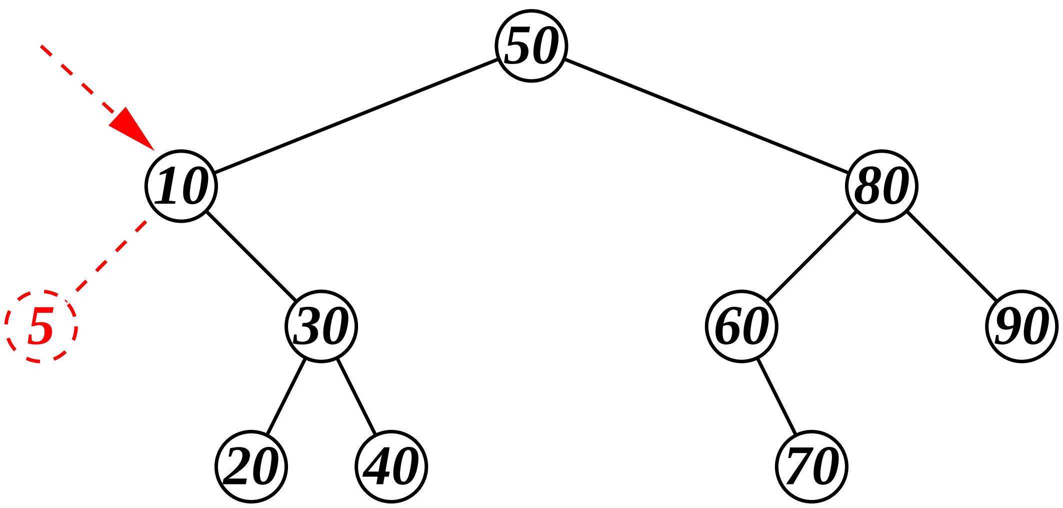


Abbildung 10.30: Einfügen in einen sortierter Binärbäum

Programm 10.20: Einfügen in einen sortierten Binärbaum (*SortBinTrees/SortBinTree1.java*)

```
public class SortBinTree1 {

        private class Node {
              Object content;
              Node left, right;
              Node(Object o){ content = o; left = right = null; }
        }
        private class Edge{
            Node parent, child;
            boolean leftEdge;
            Edge(){ parent = child = null;}
        }

        private int count;
        private Node root;

        public SortBinTree1(){
            count = 0;
            root = null;
        }
```

```

    private Edge find(Comparable o, Node root){
        /* returns edge to Node o'' in tree with root root''
         * if o'' not in tree, edge.child is null and edge.parent
         * is that node at which o'' could be added according
         * to the given order;
         * left == true denotes that it's from edge.parent to the
         * left, false denotes to the right
         */
        if (root == null) return null;
        Edge edge = new Edge();
        edge.child = root; edge.parent = null;
        while(edge.child != null){
          if (o.compareTo(edge.child.content) == 0) { return edge; }
          edge.parent = edge.child;
          if (o.compareTo(edge.child.content) < 0) {
                  edge.child = edge.child.left; edge.leftEdge = true;
          } else {
                  edge.child = edge.child.right; edge.leftEdge = false;
          }
        }
        edge.child = null; //not contained
        return edge;
     }

    public void insert(Comparable o){
         Edge np;
         if (count == 0) { root = new Node(o); count++; return; }
         np = find(o, root);
         if (np.child != null) return; // already in tree
         if (np.leftEdge) np.parent.left = new Node(o);
         else np.parent.right = new Node(o);
         count++;
     }

    private void printTree(Node r, int indent) {
        int i;
        if ( r != null) {
            printTree(r.right, indent + 5);
            for( i = 1; i <= indent; i++)
               Write.Char(' ');
            Write.Line("" + r.content);
            printTree(r.left, indent + 5);
        }
    }

    public void print(int indent) {
          printTree(root, indent);
    }
}
```

Bei den Operationen auf Bäumen ist es oft sinnvoll, mit Kanten zu arbeiten. Dazu wurde eine entsprechende interne Klasse beigefügt!

Programm 10.21: Anwendung von Programm 10.20 (*SortBinTrees/TestSortTree1.java*)

```java
import IOulm.*;

public class TestSortTree1 {
    public static void main(String[] args) {
        int i;
        SortBinTree1 t = new SortBinTree1();
        while (Urc.readInt()) {
            i = Urc.getInt();
            t.insert(new Integer(i));
        }
        t.print(4);
    }
}
```

```
euclid$ cat daten
50 20 60 30 40 80 10 25 45 35
euclid$ java TestSortTree1 < daten
              80
          60
    50
                        45
                   40
                        35
              30
                   25
          20
              10
euclid$
```

Löschen in einem binären sortierten Baum

- Der Knoten $u \in T$ ist aus T zu entfernen:
  1. Sei U der Teilbaum von T mit $\text{root}(U) = u$.
  2. Falls U_1 leer ist, dann ist U durch U_2 zu ersetzen.
  3. Falls U_2 leer ist, dann ist U durch U_1 zu ersetzen.

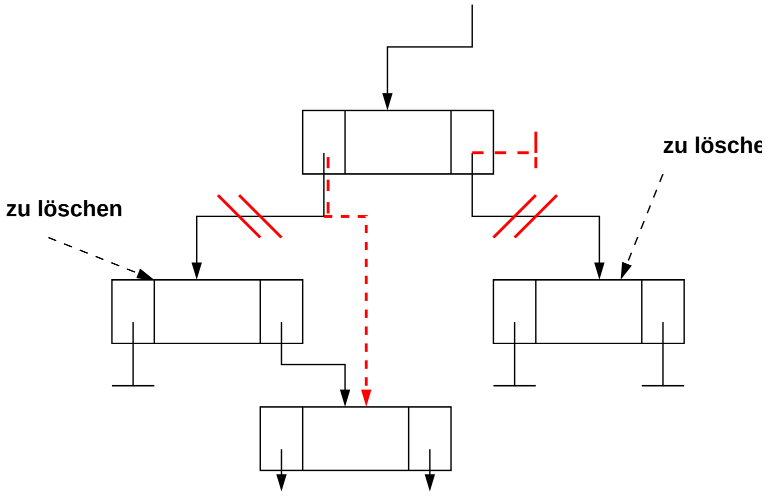


Abbildung 10.31: Löschen eines Knotens mit maximal einem Nachfolger

  4. Wenn jedoch U_1 und U_2 nicht leer sind, dann
     - ist $u_2 \in U_2$ so auszuwählen, dass $v(u_2) \leq v(t)\, \forall t \in U_2$ (einfach nur links absteigen, bis es nicht weitergeht oder vom rechten Teilbaum den kleinsten wählen) – alternativ: vom linken Teilbaum U_1 den größten wählen
     - u_2 aus U_2 zu entfernen (recht einfach, da zumindest der linke Teilbaum leer ist) und
     - u_2 als Ersatz für den Knoten u zu verwenden (siehe Abb. 10.32, S. 279)

- Das Löschen hat einen Aufwand von $O(h(T))$, der im wesentlichen durch die Suche nach dem minimalen Element in U_2 bestimmt ist.

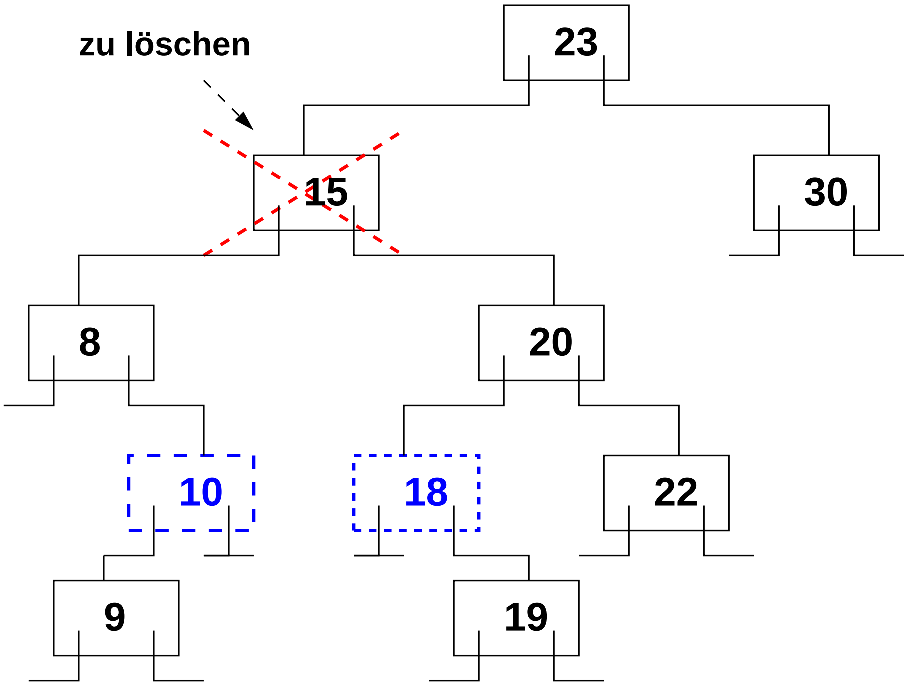


Abbildung 10.32: Löschen eines Knotens mit zwei Nachfolgern

Programm 10.22: Löschen in einem sortierten Binärbaum (*SortBinTrees/SortBinTree2.frag*)

```
import IOulm.*;

public class SortBinTree2 {

        private class Node {
            Object content;
            Node left, right;
            Node(Object o){ content = o; left = right = null; }
        }
        private class Edge{
            Node parent, child;
            boolean leftEdge;
            Edge(){ parent = child = null;}
        }

        private int count;
        private Node root;

        public SortBinTree2(){
            count = 0;
            root = null;
        }

        private Edge find(Comparable o, Node root){ // as before }

        public void insert(Comparable o){ // as before }

```

```
/*****************************************************/
        private Edge e; // global for delete () and deleteNode ()
        private void deleteNode (){
            Node parent, lowest;
            if (e.child.left == null)
                if (e.leftEdge) e.parent.left = e.child.right;
                else e.parent.right = e.child.right;
            else
                if (e.child.right == null)
                        if (e.leftEdge) e.parent.left = e.child.left;
                        else e.parent.right = e.child.left;
                else { // find lowest node on right branch
                        lowest = e.child.right; parent = e.child;
                        while (lowest.left != null) {
                            parent = lowest; lowest = lowest.left;
                        }
                        // move it upwards
                        e.child.content = lowest.content;
                        if (lowest == e.child.right) e.leftEdge = false;
                        else e.leftEdge = true;

                        e.child = lowest; e.parent = parent;
                        deleteNode ();
                    }
            }

        public void delete (Comparable o){
            e = find(o, root);
            if (e == null) return;
            if (e.child == null) // not contained
                    return;
            deleteNode ();
        }

/*****************************************************/

}
```

Kapitel 11

Hashing

11.1 Grundlagen

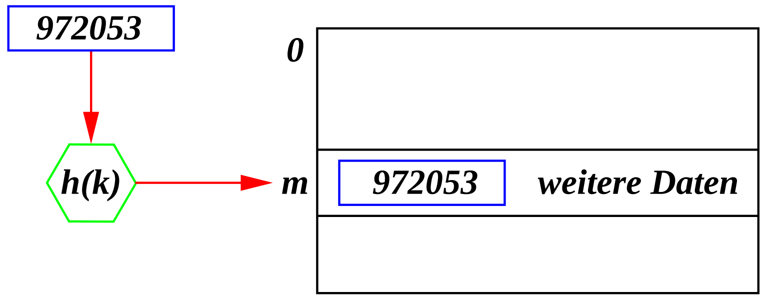


Abbildung 11.1: Hash-Funktion

- Hash-Verfahren versuchen, die Ermittlung eines Datensatzes in Abhängigkeit eines Schlüssels $k \in K$ mit Hilfe einer **Hash-Funktion** $h : K \rightarrow M$ zu vereinfachen, die einen Schlüssel k in eine Speicherposition $m \in M = [0..n-1]$ abbildet.
- Leider ist es nicht trivial, selbst für eine auch nur relativ kleine vorgegebene Menge von Schlüsseln K eine Hash-Funktion zu finden, die injektiv ist, d.h.

$$\forall k_1, k_2 \in K : h(k_1) \neq h(k_2)$$

- Entsprechend sind **Kollisionen** auch bei geringen Mengen an Schlüsseln sehr wahrscheinlich (Geburtstagsproblem[1]).
- Problemstellungen:
    - Wie wird mit Kollisionen umgegangen?
    - Wie könnte eine "gute" Hash-Funktion aussehen?

[1]Wenn 23 Menschen in einem Raum sind, so ist die Wahrscheinlichkeit, dass zwei am selben Tag Geburtstag haben, größer als 50% – eine mathematische Tatsache, die der menschlichen Intuition widerspricht!

Kollisionen

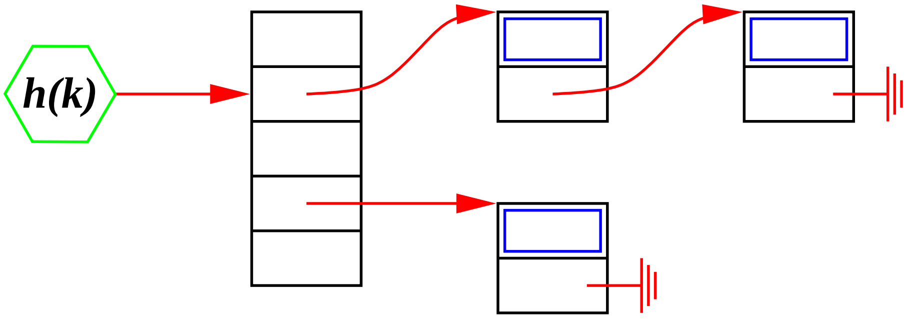


Abbildung 11.2: Bucket Table

- Die einfachste (und gängigste) Technik zur Lösung von Kollisionen besteht in der Bildung von linearen Listen. In der eigentlichen Tabelle (über den Indexbereich M) sind dann nur Objekt-Referenzen, die dann entweder **null** sind (bislang noch kein k mit $h(k) = m$ gesehen) oder Anker für lineare Listen.

- Eine solche Tabelle wird *bucket table* genannt.

- Im einfachsten Falle werden neue Elemente bei der jeweiligen Liste ganz vorne eingefügt. Es ist aber auch denkbar, die Listen sortiert zu verwalten oder entsprechend der Zugriffshäufigkeit zu organisieren, um im Falle längerer Kollisionslisten die Zugriffszeit zu reduzieren.

- Es gibt auch zahlreiche Verfahren, bei denen die Datensätze direkt in der durch M indizierten Tabelle untergebracht werden, wo dann im Falle einer Kollision andere, noch freie, Plätze ausgesucht werden. Nachteile: Limitierte Anzahl von Datensätzen (oder die Notwendigkeit von Überlauftabellen), Suchzeit wird u.U. länger und das Löschen aufwendiger.

Anforderungen an Hash-Funktionen:

- generell
  1. Effiziente Berechenbarkeit,
  2. Surjektivität (ansonsten wird Speicherplatz innerhalb von M verschwendet) und
  3. gute "Streuung", d.h. auch relativ "ähnliche" Schlüssel sollten verschiedenen Lokationen aus M zugeordnet werden.

- Hash-Funktionen müssen in Abhängigkeit von K geeignet bestimmt werden.

- Kryptographische Hash-Funktionen wie MD5 und SHA-1 erfüllen zwar den 2. und 3. Punkt in hervorragender Weise für beliebige Schlüsselmengen K, sind jedoch leider nicht effizient genug. Das Ziel hierbei ist aber auch ein ganz anderes: man möchte aus Nachrichten einen nichtmanipulierbaren Fingerabdruck gewinnen. Besondere Anforderung ist hier, dass es "sehr lange" dauern muss, bis man zwei Nachrichten k_1 und k_2 mit $k_1 \neq k_2$ gefunden hat, für die $h(k_1) = h(k_2)$ ("Kollisionsresistenz").

Anforderungen an solche Hashfunktionen sind u.a. **Kollisionsresistenz**, Repräsentativität (jedes Bit eines Textes muss in die Berechnung eingehen) und Kryptografische Sicherheit (für einen gegebenen Hashwert muss es praktisch unmöglich sein den zugehörigen Text herzustellen bzw. zurückzugewinnen).

Hashfunktionen, die diese Eigenschaften erfüllen werden im Englischen als *Message Digest* (MD) bezeichnet (*Digest*: Auszug, Zusammenfassung). Ein *Message Digest* ist der (digitale) Fingerabdruck einer Nachricht, bei der mit Hilfe von "einfach" berechenbaren Funktionen ein Wert ermittelt wird, der kürzer ist als die Originalnachricht.

Ron Rivest entwickelte – zusammen mit anderen Mitarbeitern der RSA Data Security – eine Reihe von Hashfunktionen MD1(?), MD2, MD3, MD4 bis MD5, die meist auch als Synonym für den Message Digest gelten.

Zusammen mit der National Security Agency (NSA) entwickelte das NIST (National Institute of Standards and Technology) einen zum Signieren gedachten Hash-Standard für den Digital Signature Standard (DSS). Der Standard heißt Secure Hash Standard (SHS) und spezifiziert den "Secure Hash Algorithm" (SHA) mit Hashwert der Länge 160 Bit. Der weit verbreitete Algorithmus ähnelt im Aufbau stark dem Message Digest von RSA.

- Wenn k eine ganze Zahl ist, gibt es zwei bewährte Varianten:
  - Division: $h(k) = k \bmod n$.
  - Multiplikation: $h(k) = \left\lfloor n \left(\left(\frac{A}{w} k \right) \bmod 1 \right) \right\rfloor$

  ($M = [0..n-1], w = \mathbf{MAX}(\mathbf{INTEGER}) + 1$, und A ist eine natürliche Zahl, die teilerfremd zu w ist)

11.2 Hash-Funktionen

11.2.1 Auf Division basierende Hash-Funktionen

$$h(k) = k \bmod n$$

- Die Streuung dieser Variante hängt von der Wahl von n ab.
- Wenn beispielsweise n gerade ist, dann wäre $h(k)$ genau dann gerade, wenn k gerade ist.
- Wenn n eine Zweierpotenz wäre, würden einfach die höherwertigen Bits von k abgeschnitten werden.

11.2.2 Auf Multiplikation basierende Hash-Funktionen

$$h(k) = \left\lfloor n\left(\left(\frac{A}{w}k\right) \bmod 1\right)\right\rfloor$$

- Hierbei ist $w = \mathbf{MAX(INTEGER)} + 1$ und und A eine natürliche Zahl, die teilerfremd zu w ist.
- Die binäre Repräsentierung von A kann dann als A/w betrachtet werden, wenn gedanklich links von A eine Null, gefolgt von einem Komma, steht.
- Die Multiplikation von A und k würde bei einer Wortbreite von 32 Bit insgesamt 64 Bit benötigen.
- Wenn jedoch n eine Zweier-Potenz 2^t ist, dann sind nur die führenden t Bits der niedrigwertigeren 32 Bit des Ergebnisses der Multiplikation interessant.

Programm 11.1: Auf Multiplikation basierende Hash-Funktion (*Hash1.java*)

```
import IOulm.*;

public class Hash1{
    final static int bitsPerWord = 32;
    static int t = 8;
    static int A = 1327217883;
    public static int hash(int k) {
        return (short) ( (A * k) >>> (bitsPerWord - t));
    }
    public static void main(String [] args) {
        Write.Line("Hashwert von " + 87654321 + " : " + hash(87654321));
        Write.Line("Hashwert von " + 77654321 + " : " + hash(77654321));
        Write.Line("Hashwert von " + 54321 + " : " + hash(54321));
        Write.Line("Hashwert von " + 54321000 + " : " + hash(54321000));
        Write.Line("Hashwert von " + 1 + " : " + hash(1));
        Write.Line("Hashwert von " + 1000000 + " : " + hash(1000000));
    }
}
```

```
euclid$ java Hash1
Hashwert von 87654321 : 200
Hashwert von 77654321 : 216
Hashwert von 54321 : 28
Hashwert von 54321000 : 33
Hashwert von 1 : 79
Hashwert von 1000000 : 254
euclid$
```

Das Resultat liegt im Bereich $[0, 2^t - 1]$.

Satz von Hugo Steinhaus und Vera Turán Sós:

Sei θ eine irrationale Zahl. Wenn die Punkte $\{\theta\}, \{2\theta\}, \ldots, \{n\theta\}$ in das Intervall $[0,1]$ plaziert werden (jeweils mod 1), dann haben die dadurch entstandenen $n+1$ Teilintervalle von $[0,1]$ maximal drei verschiedene Längen. Wenn der nächste Punkt $\{(n+1)\theta\}$ hinzugefügt wird, dann fällt er in eines der längsten Segmente.

- Dieser Satz stellt sicher, dass diese Punkte sehr gleichmäßig über das Intervall von $[0,1]$ verteilt werden und damit darauf basierende Hash-Funktionen eine gute Streuwirkung besitzen.
- Allerdings gibt es durchaus Unterschiede in der Wahl von θ, die sich in den möglichen Größenunterschieden der Intervalle bemerkbar machen. Wenn beispielsweise θ nahe bei 0 oder 1 gewählt wird, würde es zu Beginn viele winzige Teilintervalle und ein großes geben.
- Der goldene Schnitt $\phi^{-1} = (\sqrt{5}-1)/2$ ist ein idealer Kandidat für θ, wenn eine möglichst gleichmäßige Verteilung gewünscht wird.
- A sollte dann als die nächstgelegene ganze Zahl zu $\phi^{-1}w$ gewählt werden, die teilerfremd zu w ist.
- Somit ist beispielsweise $A = 1327217883$ geeignet für $w = 2^{31}$.

11.2.3 Andere Schlüsseltypen

- In der Praxis kommen viele weitere Schlüsseltypen neben **Integer** vor, so daß zuvor eine Konvertierung stattfinden muß.
- Längere Schlüssel (insbesondere Zeichenketten) müssen dabei so geschickt zu einer **Integer** verdichtet werden, daß auch die Reihenfolge relevant ist, d.h. `"AB"` sollte einen anderen Hash-Wert haben als `"BA"`.
- Sehr beliebt ist es daher, Summation und Rotation zu verbinden.

Programm 11.2: Hash-Funktion für Strings analog zu hashCode() (*Hash2.java*)

```java
import IOulm.*;

public class Hash2{
   final static int MAXHASHVAL = 1001;
   public static int hash(String s) {
      int sum = 0;
      for(int j = 0; j < s.length(); j++)
        sum = sum * 31 + s.charAt(j);
      return (sum >>> 1) % MAXHASHVAL;
   }

   public static void main(String[] args) {
      String s = "BayernMuenchen";
      Write.Line("Value: " + hash(s));
      Write.Line("hashValue(): " + (s.hashCode() >>> 1) % MAXHASHVAL);
    }
}
```

Programm 11.3: Hash-Funktion für Strings mit Shift und Rotation (*Hash3.java*)

```java
import IOulm.*;

public class Hash3{
    final static int MAXHASHVAL = 1001;

    static int rotShift (int i, int k) {
            // 1<=k<=31
            int tmp = 1;
            for(int j = 1; j < k; j++) tmp = (tmp << 1) | 1;
            tmp = tmp & i;
            tmp = tmp << 32 - k;
            i = i >>> k;
            return (i | tmp) ;
        }

    public static int hash(String s) {
        int hashval = 0; char ch; int ordval;
        for(int j = 0; j < s.length(); j++){
          ch = s.charAt(j);
          if ( ch >= ' ' ) ordval = ch - ' ';
          else ordval = Character.MAX_VALUE - ' ' + ch;
          hashval = (short) rotShift (hashval,5) + ordval;
        }
        return (short) (1327217883 * hashval >>> 10);
    }

    public static void main(String[] args) {
        String s = "BayernMuenchen";
        Write.Line("Value: " + hash(s) % MAXHASHVAL);
     }
}
```

11.3 Assoziative Arrays

- Assoziative Arrays (andere Namen: *hashes* und *dictionaries*) sind in den Grundoperationen vergleichbar mit normalen Arrays, abgesehen davon, dass
  - der Index-Typ beliebig sein kann (z.B. Zeichenketten) und
  - nicht notwendigerweise eine vollständige Ordnung für den Index-Typ existiert.
- In vielen Anwendungen sind assoziative Arrays sehr praktisch. So ist es z.B. bei einer Adressdatenbank sinnvoll, wenn der Name als Index-Typ dient. Häufig wird der Index auch Schlüssel genannt.
- Manchmal sind auch Zugriffe über mehrere Schlüssel interessant. Bei Adressdatenbanken könnte es beispielsweise nützlich sein, von der Telefon-Nummer oder der E-Mail-Adresse auf einen Datensatz zu schließen. Das sind dann sekundäre Schlüssel, die nicht notwendigerweise bei jedem Datensatz definiert sind.
- Das Hash-Verfahren ist ideal, um assoziative Arrays zu implementieren.
- Ist auf dem Schlüssel eine Ordnung definiert, so kann das assoziative Array auch als sortierter Baum implementiert werden.

Kapitel 12

Ausnahmen und Zusicherungen

12.1 Ausnahmebehandlung

- Ausnahmen oder Fehler werden in Prozeduren häufig mit Rückgabewert -1 signalisiert
- Fehler oder irreguläre Situation behandeln macht den Kontrollfluss u.U. sehr kompliziert!
- Explizite Ausnahmebehandlung: Eine Ausnahme (*exception*) ist eine Art Signal, das das Auftreten eines Fehlers oder einer irregulären Situation kennzeichnet.
- In Java werden Ausnahmen mit der **throw**-Anweisung ausgelöst.
- Ausnahmen können abgefangen und behandelt werden: *Exception Handler*
- Ausnahmebehandlungsroutinen werden mit der **try – catch – finally** Anweisung geschrieben

> The paradigm for using exceptions is the *try–catch–finally* sequence: you *try* something; if that something throws an exception, you *catch* the exception; and *finally* you clean up from either the normal code path or the exception code path, whichever actually happened.
> *The Java Programming Language, 3rd Edition, K. Arnold, J. Gosling, D. Holmes, Addison Wesley 2001, S. 28*

Programm 12.1, S. 290, zeigt ein erstes Beispiel.

Programm 12.1: Ausnahmebehandlung mit try – catch (*FactorialExcept.java*)

```java
/**
 * Berechnung der Fakultaet einer Zahl, uebergeben als Argument
 *  Aufruffehler werden mit try/catch behandelt
 */

public class FactorialExcept {

 public static void main( String[] args ) {

   // Versuche die Fakultaet zu berechnen --
   // was schief geht, wird in den nachfolgenden catch-Anweisungen behandelt

   try {
          int n = Integer.parseInt(args[0]);
          System.out.println(n +"! = " + fac(n));
   }
   // kein Argument angegeben, args[0] ist undefiniert
   catch (ArrayIndexOutOfBoundsException e) {
     System.err.println("Argument muss angegeben werden");
     System.err.println("Usage: java FactorialExcept <zahl>");
     System.exit(1);
   }

   // Argument ist da, aber keine Zahl
   // wird von Integer.parseInt() ausgeloest
   catch (NumberFormatException e) {
     System.err.println("Argument muss eine Integer sein");
     System.exit(2);
   }

   // Argument ist kleiner als 0
   // wird von fac() ausgeloest
   catch ( IllegalArgumentException e) {
     // von der Methode fac() gesendete Nachricht anzeigen
     System.err.println("Ungueltiges Argument: " + e.getMessage());
     System.exit(3);
   }
   finally {
     System.err.println("Zum guten Schluss!");
     // System.exit(4);
   }
   System.err.println("Zum allerletzten Schluss!");
   System.exit(0);
 }

 public static double fac(int k) {
    if ( k < 0 ) throw new IllegalArgumentException("Parameter muss >= 0 sein");
    double f = 1.0;
    while ( k > 1 ) { f *= k; k--;}
    return f;
 }
}
```

Erläuterungen zu Programm 12.1 (S. 290):

- Zeile 14:

  Wir versuchen aus einem String eine Zahl zu machen – die Wrapper-Klasse *Integer* hat eine Methode **parseInt()** ; das kann auch schief gehen, weswegen wir diesen Aufruf in einem **try**-Block absichern!

- Zeile 15:

  Da wir nun das Thema "Ausnahmenbehandlung" angehen und dies im Kontext mit Ein-/Ausgabe wichtig ist, verlassen wir die Ulmer "Spezialklassen" und kehren zum Java-Standard zurück – **System.out** ist ein automatisch erzeugtes Objekt, das eine Verbindung zum Bildschirm (**stdout**) repräsentiert und eine Methode **println()** zur Verfügung stellt – mehr dazu im letzten Abschnitt dieses Kapitels.

- Zeile 19:

  Fehler gehören an die Diagnoseausgabe (**stderr**) – mit dem automatisch erzeugten Objekt **System.err** können wir das bewerkstelligen – auch dazu mehr am Ende dieses Kapitels

- Die folgenden drei **catch**-Blöcke gehen vom speziellen zum allgemeinen Ausnahmefall: kein Argument angegeben, falsches Argument angegeben, unzulässiges Argument angegeben

  Ausnahmen sind Objekte einer entsprechenden Ausnahmenklasse – in der *catch*-Anweisung muss ein solches deklariert werden!

  Zeile 18: Objekt e der Ausnahmeklasse $ArrayIndexOutOfBoundsException$

- Zeile 35:

  Die Ausnahmeklassen haben natürlich alle einen Konstruktor, mit dem ein Ausnahmeobjekt erzeugt werden kann. Dieser hat typischerweise ein String-Argument, welches andererseits mit einer Methode **getMessage()** abgefragt werden kann (Rückgabewert).

- Zeile 47:

  Eine Ausnahme bei Methodenaufrufen besteht häufig darin, dass unzulässige Parameter übergeben werden – dies behandelt die Ausnahmeklasse **IllegalArgumentException**.

  hier: für negative ganze Zahlen erzeugen wir eine Ausnahme (mit **new** und **Konstruktoraufruf**), die wir an den Aufrufer unserer Methode $fac()$ "werfen".

Ausführung von Programm 12.1, S. 290:

```
euclid$ java FactorialExcept
Argument muss angegeben werden
euclid$
euclid$ echo $?
1
euclid$ java FactorialExcept aaaaaa
Argument muss eine Integer sein
euclid$ echo $?
2
euclid$ java FactorialExcept -4
euclid$= 0 sein
euclid$ echo $?
3
euclid$ java FactorialExcept 4
4! = 24.0
Zum guten Schluss!
Zum allerletzten Schluss!
euclid$ echo $?
0
euclid$
```

Grundstruktur der try/catch/finally-Anweisung

```
try {
  /* Im Normalfall wird dieser Code-Block wie geschrieben
   * abgearbeitet; u.U. kann er aber eine Ausnahme ausloesen:
   * direkt ueber eine throw-Anweisung oder
   * indirekt durch Aufruf einer Methode, die eine Ausnahme ausloest
   */
}

catch ( FirstException e1 ) {
  /* Dieser Block enthaelt Anweisungen, die Ausnahmen (Ausnahmeobjekte)
   * des Typs >>FirstException<< oder einer davon abgeleiteten Klasse
   * behandeln. Anweisungen in diesem Block koennen unter dem Namen
   * >>e1<< auf dieses Ausnahmeobjekt zugreifen
   */
}

catch ( AnotherException e2 ) {
  /* Dieser Block enthaelt Anweisungen, die Ausnahmen (Ausnahmeobjekte)
   * des Typs >>AnotherException<< oder einer davon abgeleiteten Klasse
   * behandeln. Anweisungen in diesem Block koennen unter dem Namen
   * >>e2<< auf dieses Ausnahmeobjekt zugreifen
   */
}

finally {
  /* Dieser Block enthaelt Anweisungen, die nach dem Verlassen
   * des try-Blocks immer ausgefuehrt werden, egal wie der
   * try-Block verlassen wurde, ob wegen Erreichen des Blockendes,
   * ob wegen Erreichen einer break- / continue- oder return-Anweisung,
   * ob wegen einer Ausnahme, die mit einem catch-Block abgefangen wurde
   * oder ob wegen einer nicht abgefangenen Ausnahme;
   * nicht aber bei Aufruf von System.exit() im try-Block
}
```

Ausführung einer throw-Anweisung

- die normale Programmausführung wird unmittelbar angehalten
- der Interpreter sucht nach einer Ausnahmebehandlungsroutine, die diese Ausnahme abfangen (behandeln) kann
  - zunächst wird im umgebenden Code-Block gesucht
  - wird eine passende gefunden, so wird der Code-Block beendet und die Ausnahmebehandlung des Code-Blocks gestartet
  - nach deren Beendigung geht es mit der Ausführung nach deren Code weiter
- Wenn der umgebende Code-Block keine passende Ausnahmebehandlung hat, so wird im nächst höheren umgebenden Codeblock gesucht, . . . !
- Wenn die Methode **keine** Ausnahmebehandlungsroutine enthält, die die mit *throw* ausgelöste Ausnahme behandeln kann:
  - die Methode wird abgebrochen und es geht zum Aufrufer zurück
  - jetzt wird hier nach einer passenden Ausnahmebehandlungsroutine gesucht
  - ggf. geht es im Aufrufstack entsprechend nach oben
  - wird die Ausnahme nirgendwo behandelt, so geht es irgendwann zur *main()*-Methode
  - wird sie auch hier nicht behandelt, so gibt der Java-Interpreter eine Fehlermeldung und einen Stack-Trace aus (der zeigt wo die Ausnahme ausgelöst wurde); danach Termination (siehe Programm 12.2, S. 295, als Modifikation von Programm 12.1, S. 290.

Programm 12.2: Ausnahmen ohne Behandlung (*FactorialExcept1.java*)

```java
/**
 * Berechnung der Fakultaet einer Zahl -- diese wird als Argument uebergeben
 */

public class FactorialExcept1 {

  public static void main( String[] args ) {

    int n = Integer.parseInt(args[0]);
    System.out.println(n +"! = " + fac(n));
    System.exit(0);
  }

  public static double fac(int k) {
    if ( k < 0 )
      throw new IllegalArgumentException("Parameter muss >= 0 sein");
    double fact = 1.0;
    while ( k > 1 ) { fact *= k; k--;}
    return fact;
  }
}
```

Ausführung:

```
euclid$  java FactorialExcept1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsE
xception: 0
   at FactorialExcept1.main(FactorialExcept1.java:9)
euclid$  java FactorialExcept1 aaaa
Exception in thread "main" java.lang.NumberFormatException: For
input string: "aaaa"
   at java.lang.NumberFormatException.forInputString(NumberFormat
Exception.java:48)
   at java.lang.Integer.parseInt(Integer.java:446)
   at java.lang.Integer.parseInt(Integer.java:496)
   at FactorialExcept1.main(FactorialExcept1.java:9)
euclid$  java FactorialExcept1 -4
Exception in thread "main" java.lang.IllegalArgumentException:
Parameter muss >= 0 sein
        at FactorialExcept1.fac(FactorialExcept1.java:16)
        at FactorialExcept1.main(FactorialExcept1.java:10)
euclid$  java FactorialExcept1 4
4! = 24.0
euclid$
```

Ausnahmetypen:

- **Jede Ausnahme ist ein Objekt!**
- Sein Typ ist **java.lang.Throwable** oder eine daraus abgeleitete Klasse
- Standardmäßig abgeleitete Klassen:
  - **java.lang.Error** : meist nicht behebbare Fehler (**Error-Exceptions**
  - **java.lang.Exception**: weisen auf weniger schwerwiegende Zustände hin **Exceptions**
    Dazu gehören:
    java.io.EOFException (End-of-File)
    java.lang.ArrayIndexOutOfBoundsException (ein Programm versucht, über das Ende eines Arrays hinaus zu lesen / zu schreiben
- Ausnahmen als Objekt können Daten und Methoden, die auf diesen arbeiten, enthalten
  - Klasse **Throwable** und alle davon abgeleiteten Klassen enthalten String-Feld mit lesbarer Fehlermeldung zum Ausnahmezustand
  - wird eingetragen, wenn das Objekt erzeugt wird
  - kann mit der Methode **getMessage()** gelesen werden

Deklaration von Ausnahmen:

- Unterscheidungen:
  - *Error*-Ausnahmen vs. *Exception*-Ausnahmen (s.o)
  - während der Übersetzung **geprüfte** (*checked*), also per Deklaration anzugebende Ausnahmen, und **nicht geprüfte** (*unchecked*) Ausnahmen
- Es gilt:
  - Jedes Ausnahmeobjekt, dessen Klasse von *Error* abgeleitet ist, ist ungeprüft!
  - Jedes Ausnahmeobjekt, dessen Klasse von *Exception* abgeleitet ist, ist geprüft

    Ausnahme:
    es ist eine von *java.lang.RuntimeException* abgeleitete Klasse; in diesem Fall ist sie ungeprüft (*RuntimeException* ist von *Exception* abgeleitet)
- praktisch jede Methode kann ungeprüfte (unprüfbare) Ausnahmen auslösen (*OutOfMemoryError* oder *NullPointerException* bei Methoden, die Arrays oder Objekte verwenden (*null*-Argument))
- geprüfte Ausnahmen treten nur in wohldefinierten Situationen auf (z.B. beim Lesen aus Dateien: *FileNotFoundException* – man muss damit rechnen, dass die gesuchte Datei nicht gefunden werden kann)

Arbeiten mit geprüften / ungeprüften Ausnahmen:

- Beim Schreiben einer Methode, die eine geprüfte Methode auslöst, muss eine **throws**-Anweisung verwendet werden, um diese Ausnahme im Methodenkopf zu deklarieren; der Java-Compiler prüft, ob diese Ausnahmen im Methoden-Kopf deklariert wurden (daher "geprüfte" Ausnahmen).

  Beispiele:

  – siehe Programm 12.3, S. 298:
  `public static void main(String[] args) throws IOException{`

  – siehe Programm 12.1, S. 290:

  ```
  public static double fac(int k) {
    if ( k < 0 )
      throw new IllegalArgumentException("Parameter muss >= 0 sein");
    double f = 1.0;
    while ( k > 1 ) { f *= k; k--; }
    return f;
  ```

  Hier wird eine Ausnahme des Typs *java.lang.IllegalArgumentException* ausgelöst – diese ist von RuntimeException abgeleitet, daher **ungeprüft** und deswegen muss sie nicht deklariert werden.
  Hätte aber auch deklariert werden können:

  ```
  public static double fac(int k) throws IllegalArgumentException {
    if ( k < 0 )
      throw new IllegalArgumentException("Parameter muss >= 0 sein");
    double fact = 1.0;
    while ( k > 1 ) { fact *= k; k--; }
    return fact;
  }
  ```

- Müssen "geprüfte" Ausnahmen deklariert werden? $\hookrightarrow$ Ja!

  – eine (eigene) Methode ruft eine andere Methode auf – diese könnte prinzipiell eine geprüfte Ausnahme auslösen

  – die (eigene) Methode **muss** dann diese Ausnahme entweder mit einer eigenen Routine behandeln oder diese mit *throws* deklarieren, um zu zeigen, dass die (eigene) Methode die Ausnahme weiterreichen kann.

- Woher weiss man, ob eine Methode, die man aufruft, eine Ausnahme auslösen kann?

  $\hookrightarrow$ Methodenkopf anschauen

  $\hookrightarrow$ den Compiler "fragen" (siehe nachfolgendes Beispiel).

Im Programm 12.3 (S. 298) lesen wir mit der Java-Standardbibliothek (mehr dazu im letzten Abschnitt dieses Kapitels) von *stdin*, in gleicher Weise wie man auch aus Dateien liest. Beim Lesen aus Dateien sind grundsätzlich Ausnahmen zu erwarten:

Programm 12.3: Ohne throws-Deklaration (*Factorial2.java*)

```java
1 /**
2  * Integer aus Stdin
3  * Fakultaet iterativ berechnen
4  */
5
6 import java.io.*; // Alle Klassen im Paket java.io importieren
7
8 public class Factorial2 {
9
10   public static void main(String[] args) throws IOException {
11
12     BufferedReader in =
13         new BufferedReader(new InputStreamReader(System.in));
14     for (;;) {
15       System.out.print("Gib Zahl: ");
16       String line = in.readLine();
17       if( ( line == null ) || (line.equals("quit") ) )
18         break;
19       try {
20         int n = Integer.parseInt(line);
21         System.out.println(n + "! = " + factorial(n));
22       }
23       catch(Exception e) {
24         System.out.println("Fehler in Eingabe <<" + e.getMessage() + ">>"); }
25     }
26   }
27
28   public static double factorial(int k) {
29     if ( k < 0 ) return 0.0;
30     double fact = 1.0;
31     while ( k > 1 ) {
32       fact *= k;
33       k--;
34     }
35     return fact;
36   }
37 }
```

Erläuterungen:

- Zeile 12-13:

  Aus dem Objekt *System.in* (Verbindung zi *stdin*) erzeugen wir ein *InputStreamReader*-Objekt, aus diesem wiederum ein *BufferedReader*-Objekt.

- Zeile 16:

  BufferedReader-Objekte kennen eine Methode *readline()*, mit der eine Zeile (*String*-Objekt) gelesen werden kann.

Lässt man in Programm 12.3 (S. 298) im Methodenkopf von *main()* die **throws**-Deklaration weg, so meldet der Java-Compiler:

```
Factorial2.java:16: unreported exception java.io.IOException;
must be caught or declared to be thrown
     String line = in.readLine();
                          ^
1 error
```

Noch ein Beispiel: Berechnung großer Zahlen (**java.math.BigInteger** und **java.util**

Programm 12.4: Große Fakultäten (*FacBig.java*)

```java
// Berechnung der Fakultaeten von grossen Zahlen
import java.math.BigInteger;
import java.util.*;

public class FacBig {
 private static ArrayList table = new ArrayList();
 // ArrayList ist dynamisches Array
 static { // erstes Element mit 0!=1 initialisieren
        table.add(BigInteger.valueOf(1));
      }
 public static BigInteger fac(int k) {
   if ( k < 0 )
     throw new IllegalArgumentException("Parameter muss >= 0 sein");
   for (int size = table.size(); size <= k; size++) {
     BigInteger lastFac = (BigInteger) table.get(size-1);
     BigInteger nextFac = lastFac.multiply(BigInteger.valueOf(size));
     table.add(nextFac);
   }
   return (BigInteger) table.get(k);
 }

 public static void main( String[] args ) {
  try { int n = Integer.parseInt(args[0]);
        for (int i = 1; i <= n; i++)
          System.out.println(i +"! = " + fac(i));
      }
  // kein Argument angegeben, args[0] ist undefiniert
  catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Usage: java FactorialExcept <zahl>");
    System.exit(1);
  }
  catch ( IllegalArgumentException e) {
    System.out.println("Ungueltiges Argument: " + e.getMessage());
    System.exit(2);
  }
  System.exit(0);
 }
}
```

Erläuterungen:

- Zeile 2:

  Die Klasse **java.math.BigInteger** enthält Methoden zum Arbeiten mit sehr großen ganzen Zahlen.

- Zeile 6:

  Die Klasse **ArrayList** implementiert ein Listen-Interface

- Zeilen 8-10:

  mit der Methode *add()* der Klasse *ArrayList* wird eine erste große Integer-Zahl (Wert von 1) hinzugefügt!

- Zeilen 11-20:

  Berechnung von *k*-Fakultät als große Integer.

  Mit der Methode *size()* erhalten wir die Anzahl der Elemente in unserer *ArrayList* – es werden die weiteren Fakultäten *size* bis *k* berechnet und im Array abgelegt!

finally-Block:

- wird ausgeführt, auch wenn der try-Block mit break, continue oder return verlassen wird
- erst wird ein catch-Block (sofern passend) angesprungen und dann der finally-Block
- gibt es keinen lokalen catch-Block, so wird der finally-Block ausgeführt, **bevor** die Kontrolle an den nächsten umgebenden catch-Block, der die Ausnahme behandeln kann, weitergegeben wird
- wird im finally-Block die Kontrolle mit einer return, break, continue, oder throw-Anweisung abgegeben oder wird eine Methode aufgerufen, die eine Ausnahme auslöst, so wird die ausstehende Kontrollabgabe verworfen und die neue verarbeitet, d.h. z.B.:
  - in einem finally-Block wird eine Ausnahme ausgelöst, so ersetzt diese Ausnahme eine evt. gerade in Verarbeitung befindliche Ausnahme
  - enthält der finally-Block eine return-Anweisung, dann kehrt die Methode normal zurück, auch wenn eine Ausnahme ausgelöst aber noch nicht behandelt wurde.

Beispiel:

Programm 12.5: try–finally (*Continue.java*)

```java
// Continue in Schleifen

public class Continue{

  public static void main(String[] args) {

    for(int k = 0; k <= 9; k++){
      if (k % 2 == 0) continue;
      System.out.println("k = " + k);
    }
    int k = 0;
    while (k <= 9 ) {
      try {
        if (k % 2 == 0) continue;
        System.out.println("k = " + k);
      }
      finally {
        k++;
      }
    }
  }
}
```

12.2 Assertions

Ab Version 1.4!

- Zusicherungen sind Bedingungen, die an beliebigen Stellen einen korrekten Programmzustand beschreiben sollen: Schlüsselwort **assert**.
- Zusicherungen müssen beim Übersetzen aktiviert werden, ansonsten sind sie nichts anderes als Kommentar!
- Ist eine Zusicherung aktiviert und schlägt sie fehl, so wird eine Ausnahme erzeugt (**java.lang.AssertionError**)
- Die assert-Anweisung kann ein Argument (eine Bedingung) oder zwei Argumente (zusätzlich: Fehlernachricht an den Konstruktor *AssertionError()*) haben

Programm 12.6 (S. 302) zeigt ein Beispiel.

Programm 12.6: Assertion – Zusicherung (*MyAssert.java*)

```
// Assertions

import java.io.*; // Alle Klassen im Paket java.io importieren

public class MyAssert {

  public static void main(String[] args) throws IOException{
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    for (;;) {
      System.out.print("Gib Zahl: ");
      String line = in.readLine();
      if( ( line == null ) || ( line.equals("quit") ) )
        break;
      try {
        int n = Integer.parseInt(line);
        System.out.println(n + "! = " + factorial(n));
      }
      catch(Exception e) {
        System.out.println("Fehler in Eingabe <<" + e.getMessage() + ">>"); }
    }
  }

  public static double factorial(int k) {
    assert (k >= 0) : "got " + ">>" + k + "<<";
    double fact = 1.0;
    while ( k > 1 ) { fact *= k; k--;}
    return fact;
  }
}
```

Übersetzung und Ausführung:

```
thales$ javac -source 1.4 MyAssert.java
thales$ java -ea:MyAssert MyAssert
Gib Zahl: 9
9! = 362880.0
Gib Zahl: -1
Exception in thread "main" java.lang.AssertionError: -1
        at MyAssert.factorial(MyAssert.java:30)
        at MyAssert.main(MyAssert.java:22)
thales$
thales$ java -ea:MyAssert
# Diagnose-Meldungen, u.a.
#  -ea[:<packagename>...|:<classname>]
thales$
thales$ java -ea MyAssert
Gib Zahl: -1
Exception in thread "main" java.lang.AssertionError: -1
        at MyAssert.factorial(MyAssert.java:30)
        at MyAssert.main(MyAssert.java:22)
thales$
```

- **Übersetzen von Zusicherungen:**

  `javac -source 1.4 MyAssert.java`

- Aktivieren von Zusicherungen
  - Zusicherungen sind zunächst deaktiviert (*disabled*) – Performance!
  - Anm.: (Gute) Zusicherungen sind auch guter Kommentar!
  - Die Aktivierung von Zusicherungen in allen Klassen außer Systemklassen erfolgt mit der Option **-ea**
  - Sollen auch die Assertions in den Systemklassen aktiviert werden: Option **-esa**
  - Deaktivieren von Zusicherungen: Option **-da**

12.3 File-IO

Die grundlegende Abstraktion zum **File-IO** sind **Streams**. Streams sind abstrakte Konzepte, mit welchen man Ein-und Ausgabekanäle modelliert. Man unterscheidet zwischen Output Streams (Ausgabeströmen), in welche man Bytes schreiben, und Input Streams (Eingabeströmen), von welchen man Bytes lesen kann.

Einen Stream kann man sich als (unendliche) Folge von Bytes vorstellen:

- Ausgabe eines Textes auf einen Bildschirm kann als die Folge der Bytes angesehen werden, aus denen der Text besteht.
- Beim Lesen einer Datei ist der Stream einfach die Folge der Bytes, aus denen die Datei besteht.
- Beim Versenden eines Bildes über ein Netzwerk besteht der Stream aus den Bytes, aus denen das Bild zusammengesetzt ist

Das Stream-Konzept ist stark abstrahierend von der Frage, woher oder wohin die Bytes strömen oder was sie eigentlich darstellen.

In Java sind die abstrakten Klassen **java.io.OutputStream** und **java.io.InputStream** definiert, welche Aus- bzw. Eingabe-Ströme repräsentieren. Diese Klassen sind abstrakt, weil das Konzept der strömenden Bytes ziemlich fundamental ist und nicht von der Quelle (File, Netzwerk, Tastatur ...) oder der Senke (File, Netzwerk, Bildschirm, ...) abhängt.

Einige Methoden dieser Klassen:

InputStream

`public void close();`	Schliessen des Eingabestroms und Freigeben von damit verbundenen Systemressourcen
`public abstract int read();`	Lesen eines Bytes. Rückgabewert: Das gelesene Byte, oder -1 wenn das Ende des Stroms erreicht wurde. Diese Methode wird in jeder abgeleiteten Klasse implementiert.
`public int read(byte[] byteArray);`	Lesen aller verfügbaren Bytes des Eingabestroms in ein Byte-Array. Rückgabewert: Die Anzahl gelesener Bytes, oder -1 wenn das Ende des Stroms erreicht wurde. Diese Methode wird in jeder abgeleiteten Klasse implementiert.
`public int skip(long numBytes);`	Überspringt die nächsten `numBytes` Bytes.

OutputStream

`public void close();`	wie InputStream
`public abstract void write(byte b);`	Schreiben eines einzelnen Bytes. Diese Methode wird in jeder abgeleiteten Klasse implementiert
`public void write(byte[] byteArray);`	Schreiben eines byte-Arrays
`public void flush();`	Leert Puffer, bewirkt also, dass ggf. gepufferte Bytes geschrieben / geschickt werden.

Für konkrete Datenquellen sind auch “konkrete” Klassen implementiert:

- **java.io.FileInputStream** für das Lesen von Dateien
- **java.io.ByteArrayInputStream** um Daten aus einem Array im Arbeitsspeicher so zu lesen, wie wenn sie aus einer Datei oder über das Netz kämen
- **java.io.FileOutputStream** für das Schreiben einer Datei
- **java.io.ByteArrayOutputStream** für das Schreiben in ein Array

Reader und Writer

Diese verleihen einem Eingabe- / Ausgabestrom eine gewisse Struktur.

Reader:

Wenn man zum Beispiel eine Textdatei lesen will, so ist man eher daran interessiert, ganze Zeilen als Strings einlesen zu können als ein Array von Bytes. Von einem *Reader*-Objekt kann man einzelne Chars, Arrays von Chars oder Strings beziehen. Viele von *Reader* abgeleitete Klassen sorgen dafür, dass ein Strom von Bytes in Unicode Zeichen umgewandelt wird.

Jeder *Reader* besitzt einen **InputStream**; die meisten *Reader* erwarten einen *InputStream* als Parameter für ihren Konstruktor. Dieser *InputStream* ist der eigentliche Strom, auf dem die Bytes hereinkommen, bevor sie vom *Reader* in Zeichen (Characters) umgewandelt werden.

Eine der praktischsten *Reader* ist die Klasse **BufferedReader**, welche unter anderem eine Methode `String readLine()` besitzt. Diese Methode liest die Bytes ihres Eingabestroms, setzt sie zu *Chars* zusammen und fügt sie nacheinander in einen String. Sobald ein Zeilenumbruch kommt, ist das Sammeln der Bytes vorbei und der String wird (ohne das *newline*) zurückgegeben. Um einen *BufferedReader* zu erzeugen, muss man dem Konstruktor ein anderes *Reader*-Objekt übergeben.

Writer:

In vielen Fällen ist es ziemlich umständlich, eine gewünschte Ausgabe in einen Strom von Bytes zu verwandeln. Ein *Writer*-Objekt ist in der Lage *Chars* und *Strings* in *Bytes* umzuwandeln (was z.B. bei Unicode Zeichen sehr nützlich ist).

Beim Schreiben vom Textdateien werden Zeilentrenner je nach Betriebssystem unterschiedlich (`\n` oder auch `\r\n`) repräsentiert. Ein *Writer* für eine Textdatei fügt automatisch den richtigen Zeilentrenner ein.

Jeder *Writer* besitzt einen *OutputStream*; die meisten *Writer* erwarten einen *OutputStream* als Parameter für ihren Konstruktor. Dieser *OutputStream* ist der eigentliche Strom, auf dem die Bytes hinausgeschrieben werden, die vom *Writer* aus den *Chars* und *Strings* erzeugt worden sind.

Einer der praktischsten *Writer* ist die Klasse **PrintWriter**, welche für jeden primitiven Datentyp eine entsprechende **println()**-Methode besitzt. Diese konvertieren den Parameter in einen String; dieser wird in eine Folge von Bytes umgewandelt und zusätzlich wird noch ein Zeilentrenner angefügt.

Um einen *PrintWriter* zu erzeugen, muss man dem Konstruktor ein anderes Writer-Objekt übergeben.

Beispiele:

Wir hatten bisher mit der speziell präparierten Klasse *Read* aus dem Paket *IOulm* gearbeitet. Diese basiert natürlich auch auf dem oben skizzierten Konzept.

Beim Lesen von Zeichen-Strömen geht man normalerweise so vor, dass man zunächst eine Instanz eines *InputStream* (d.h. einer von *InputStream* abgeleiteten Klasse) erzeugt. Danach erzeugt man eine Instanz einer von *Reader* abgeleiteten Klasse. Diesen *Reader* benutzt man schliesslich für das Erstellen eines *BufferedReader*.

Programm 12.7: Lesen von stdin und schreiben nach stdout (*Read1.java*)

```
1  import java.io.*;
2
3  public class Read1{
4
5    public static void main(String[] args) {
6          String inline;
7          InputStreamReader instream = new InputStreamReader(System.in);
8          BufferedReader buff = new BufferedReader(instream);
9          try {
10             while ( (inline = buff.readLine()) != null)
11                     System.out.println(inline);
12
13             }
14             catch(Exception e) {
15                   System.err.println(e.getMessage());
16                   System.exit(1);
17             }
18   }
19 }
```

Erläuterungen zu Programm 12.7, S. 306:

- Das Objekt **System.in** ist ein *InputStream*, die Objekte **System.out** und **System.err** und sind beide *OutputStream*s, die wie man es von Unix her kennt, standardmäßig vom System bereits geöffnet und vorbereitet sind.
- Aus dem *InputStream* machen wir jetzt einen *InputStreamReader* (ein *InputStreamReader* konvertiert Bytes zu Chars): Zeile 7
- Um jetzt auf angenehme Weise Zeilen von stdin lesen zu können, erzeugen wir einen BufferedReader: Zeile 8
- Um die nächste Zeile zu lesen, verwenden wir die *readLine()*-Methode: Zeile 10

  NB: Die Methode *readLine()* blockiert, bis die Zeile mit Return abgeschlossen wird. Erst dann wird der String mit den eingegebenen Zeichen gefüllt. Der null-Referenz wird bei Ende der Eingabe geliefert.
- Die Zeilen 7 und 8 lassen sich auch zusammenfassen zu

  `BufferedReader buff = new BufferedReader(new InputStreamReader(System.in));`
- Ausgabeseitig ist nichts anzumerken!

Lesen aus einer Textdatei:

Programm 12.8: Lesen aus einer Textdatei (*ReadFile.java*)

```java
import java.io.*;

public class ReadFile {

  public static void main(String[] args) {
      String inline, infile;
      FileInputStream fstream = null;
      if (args.length == 0){
            System.err.println("usage: java ReadFile filename"); System.exit(1);
      }
      infile = args[0];
      try { fstream = new FileInputStream(infile);
      } catch (FileNotFoundException fnfe) {
            System.err.println(fnfe.getMessage()); System.exit(2);
      }

      BufferedReader buff = new BufferedReader(new InputStreamReader(fstream));
      try {
          while ((inline = buff.readLine()) != null) System.out.println(inline);
      } catch(Exception e) {
          System.err.println(e.getMessage()); System.exit(1);
      }
  }
}
```

Ausführung:

```
euclid$ java ReadFile
usage: java ReadFile filename
euclid$ java ReadFile xxx
xxx (No such file or directory)
euclid$ java ReadFile einText
Read1.class
Read1.java
einText
euclid$ cat einText
Read1.class
Read1.java
einText
euclid$ java ReadFile einText1
einText1 (Permission denied)
euclid$ ll einText1
--w-r--r--    1 swg      users          31 2006-04-18 19:56 einText1
euclid$
```

Für das Lesen von Dateien gibt es die von InputStreamReader abgeleitete Klasse **FileReader**, die sich direkt über einen Dateinamens erzeugen lässt – Programm 12.9, S. 308.

Programm 12.9: Lesen aus einer Textdatei – vereinfacht (*ReadFile1.java*)

```
1 import java.io.*;
2
3 public class ReadFile1 {
4
5   public static void main(String[] args) {
6       String inline, infile;
7       FileReader fr = null;
8       if (args.length == 0){
9               System.err.println("usage: java ReadFile filename"); System.exit(1);
10      }
11      infile = args[0];
12      try { fr = new FileReader(infile);
13      } catch (FileNotFoundException fnfe) {
14              System.err.println(fnfe.getMessage()); System.exit(2);
15      }
16      BufferedReader buff = new BufferedReader(fr);
17      try {
18          while ((inline = buff.readLine()) != null) System.out.println(inline);
19      }
20      catch(Exception e) {
21          System.err.println(e.getMessage()); System.exit(1);
22      }
23  }
24 }
```

Erläuterungen:

- Da ein *FileReader* ein *InputStreamReader* ist, können wir direkt einen *BufferedReader* herstellen – Zeile 16.

Beim Schreiben von Zeichen-Strömen (Char) geht man normalerweise so vor, dass man zunächst eine Instanz eines **FileOutputStream** (einer von *OutputStream* abgeleiteten Klasse) erzeugt. Danach erzeugt man eine Instanz der Klasse **OutputStreamWriter**, einer von **Writer** abgeleiteten Klasse. Diesen *Writer* benutzt man schliesslich für das Erstellen eines **PrintWriter**s.

Dies kann über die Klasse **FileWriter**, abgeleitet von **OutputStreamWriter**, die sich direkt mit Hilfe eines Dateinamens erzeugen lässt, weider abkürzen (siehe Programm **??**, S. **??**.

Programm 12.10: Lesen aus einer und Schreiben in eine Textdatei (*ReadWriteFile.java*)

```java
import java.io.*;

public class ReadWriteFile {

  public static void main(String[] args) {
     String inline, infile, outfile;
     FileReader fr = null; FileWriter fw = null;
     if (args.length != 2){
           System.err.println("usage: java ReadWriteFile infile outfile");
           System.exit(1);
     }
     infile = args[0]; outfile = args[1];
     try { fr = new FileReader(infile);
     } catch (IOException fnfe) {
          System.err.println(fnfe.getMessage()); System.exit(2);
     }
     BufferedReader buff = new BufferedReader(fr);

     try {
       fw = new FileWriter(outfile);
     } catch (IOException fnfe) {
          System.err.println(fnfe.getMessage()); System.exit(3);
     }
     PrintWriter pw = new PrintWriter(fw);

     int i = 0;
     try {
          while ( (inline = buff.readLine()) != null) {
               i++; pw.println("" + i + ": " + inline);
          }
     }
     catch(Exception e) {
          System.err.println(e.getMessage()); System.exit(1);
     } finally {
          pw.close();
     }
  }
}
```

```
euclid$  ls *Text
einText
euclid$  cat einText
Read1.class
Read1.java
einText
euclid$  java ReadWriteFile
usage: java ReadWriteFile infile outfile
euclid$  java ReadWriteFile einText
usage: java ReadWriteFile infile outfile
euclid$  java ReadWriteFile einText ausText
euclid$  cat ausText
1: Read1.class
2: Read1.java
3: einText
euclid$  chmod -w ausText
euclid$  java ReadWriteFile einText ausText
ausText (Permission denied)
euclid$  chmod +w ausText
euclid$  java ReadWriteFile einText ausText
euclid$  cat ausText
1: Read1.class
2: Read1.java
3: einText
euclid$
```

Anm.: Damit sind auch **System.in**, **System.out** und **System.err** klar

- *System.in* ist eine Instanz eines **InputStream** (automatisch erzeigt)
- *System.out* ist ein (automatisch erzeugtes) **PrintStream**-Objekt und besitzt deshalb die diversen Methoden für die Ausgabe von primitiven Datentypen wie z.B. die verschiedenen verb|println()|-Varianten
- für *System.err* gilt dasselbe

Die Klasse **FileWriter** besitzt noch einen Konstruktor, mit dem man eine Datei zum Schreiben im Sinne von hinten Anfügen öffnen kann:

```
FileWriter(File file, boolean append)
          Constructs a FileWriter object given a File object.
```

Anhang

Literatur

[1] K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language Third Edition*. Addison Wesley 2001.

[2] J. Bloch: *Effective Java Programming Language Guide* Addison Wesley 2001

[3] J. Christ: *TerminalBuch vi - Effizient editieren unter UNIX*. Oldenbourg Verlag München Wien 1989.

[4] D. Flanagan: *Java Examples in a Nutshell*. O'Reilly 2001

[5] D. Flanagan: *Java in a Nutshell*. O'Reilly 2003

[6] J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java Language Specification Seconde Edition*. Addison Wesley 2000

[7] H. Herold: *Linux-Unix-Shells*. Addison-Wesley, 1999.

[8] B. W. Kernighan und R. Pike: *Der UNIX-Werkzeugkasten*. Hanser, 1986.

[9] W. Küchlin, A. Weber: *Einführung in die Informatik*. Springer 2005

[10] H.W. Lang: *Algorithmen in Java*. Oldenbourg 2003

[11] Ch. Ullenboom: *Java ist auch eine Insel*. Galileo Computing

⇝ **jeweils auf aktuellste Ausgabe achten!**

Abbildungsverzeichnis

Beispiel-Programme

Index