Algorithmen und Datenstrukturen für Angewandte und Technische Informatiker

Vorlesungsskript

Bernd Kahlbrandt[1], Christoph Klauck[2], Stephan Pareigis[3]
Department für Informatik
Hochschule für Angewandte Wissenschaften Hamburg

Sommersemester 2012

[1]bernd.kahlbrandt@informatik.haw-hamburg.de
[2]christoph.klauck@informatik.haw-hamburg.de
[3]stephan.pareigis@informatik.haw-hamburg.de

Versiongeschichte

- 1. Februar 2005 erste Version
- 11. April 2006 Sieb des E. äußerer Schleifenzähler i ab 2
- 2.Juni 2006 neues Kapitel Hashtabellen: Löschen von Elementen
- 2.Juni 2006 Ergänzung im Kapitel Hashtabellen: load factor, capacity, resize
- 2.Juni 2006 neues Kapitel minimale perfekte Hashfunktionen
- 13.September 2006 Index eingeführt
- 15.September 2006 Kapitel 1 ADT komplett überarbeitet
- Überarbeitung für das Wintersemester 2007/8 (Bernd Kahlbrandt)
- Überarbeitung für das SS 2008 (Bernd Kahlbrandt, unter Verwendung von Material von Christoph Klauck)
- Überarbeitung für das WS 2008/9 (Bernd Kahlbrandt)
- Überarbeitung für das SS 2009 (Bernd Kahlbrandt/Christoph Klauck)
- Überarbeitung für das SS 2012 (Bernd Kahlbrandt)

Vorwort

Dieses Skript soll Sie bei der Vor- und Nachbereitung von Vorlesung und Praktikum AD unterstützen. Ich möchte Sie aber auch motivieren weitere Quellen selbstständig zu erschließen. Alle von mir verwendeten Quellen finden Sie im Literaturverzeichnis. Besonders möchte ich Sie aber auf Cormen et al. *Introduction to Algorithms* [CLR94] und Don Knuth, *The Art of Computer Programming* [Knuff] hinweisen. Insbesondere letzteres ist eine Fundgrube für Praktikums- und Klausuraufgaben. Die Wahrscheinlichkeit ist klein, dass eine Aufgabe, die ich in Praktikum oder Klausur stelle, *nicht* in irgendeiner Form in diesem Werk sich findet. Auch auf Don Knuths Homepage www-cs-faculty.stanford.edu/~knuth/ möchte ich Sie explizit hinweisen. Hier finden Sie auch erste Teile von *Volume IV*, so lange Sie noch nicht als Buch erschienen sind.

Für das Wintersemester 2008/9 hat Bernd Kahlbrandt das Skript nochmals überarbeitet und hoffentliche weitere Schreibfehler und missverständliche Formulierungen beseitigt. Das Kapitel 6 wurde vervollständigt. Die Anzahl der Übungsaufgaben wurde weiter erhöht.

Auch für das SS 2009 wurden einige kleinere Änderungen, Ergänzungen und hoffentlich Verbesserungen vorgenommen.

Für das SS 2010 und das S 2010/11 sind nur geringfügige Änderungen vorgenommen worden.

Für das SS 2012 wurden einige Abschnitte vervollständigt und weitere kleinere Verbesserungen vorgenommen.

Einordnung der Vorlesung

Die Vorlesungen der Informatik können (im Groben) in folgende Lehrgebiete eingeteilt werden:

Anwendungen Informatik : Dieses Gebiet ist durch die Anwendungen, etwa der Medizin, Biologie, Psychologie, Banken etc. geprägt. Hier findet man spezielle Anwendungssysteme, wie etwa kaufmännische oder technische Informationssysteme.

In diesem Gebiet sind konkrete Lösungen, etwa aus der Konstruktionslehre oder der Theorie, zu finden. Die Veränderung hängt einmal von dem Lebenszyklus der Anwendung ab und dem Lebenszyklus der verwendeten Systeme.

Stellt dieses Gebiet den Schwerpunkt des Studiengangs dar, werden diese meist als „Bindestrich“-Informatik bezeichnet, etwa Medizin-Informatik oder Maschinenbau-Informatik.

Konstruktionslehre der Informatik : Dieses Gebiet führt eine Methodenlehre durch, die sich dadurch auszeichnet, anwendungsunabhängige Konstruktionsprinzipien vorzustellen, was durch Design Pattern, Basisalgorithmen etc. vorgenommen wird. Hier findet man allgemeine Methoden und Werkzeuge, System- und Anwendungskomponenten, Qualitätssicherung, Engineeringsmethoden etc.

Die Veränderungen in diesem Gebiet hängen von den „Paradigmen“’ der Informatik ab und haben aber auch darüber hinaus noch Gültigkeit. Bekannteste Beispiele wären die Objektorientiertheit oder das Client/Server Prinzip.

Stellt dieses Gebiet den Schwerpunkt des Studiengangs dar, wird dieser meist als Informatik bezeichnet.

Praktische Informatik : In diesem Gebiet werden aktuelle allgemeine Werkzeuge vorgestellt, wie etwa Betriebssysteme, Datenbanksysteme, Programmiersysteme oder auch Kommunikationssysteme.

Zu den aktuellen Werkzeugen gehören zum Einen die auf dem Markt neu erscheinenden Werkzeuge und zum Anderen auch die auf dem Markt vornehmlich vorhandenen Werkzeuge. Die Veränderungen in diesem Gebiet sind also recht schnell.

Wegen der rasanten Entwicklung wird dieses Gebiet nicht als Schwerpunkt eines Studiengangs verwendet.

Technische Informatik : In diesem Gebiet werden aktuelle technische Werkzeuge vorgestellt, wie etwa die Rechnertechnologie oder auch Netzwerktechnologie.

Die Veränderungen in diesem Gebiet sind im Moment rasant.

Stellt dieses Gebiet den Schwerpunkt des Studiengangs dar, wird dieser meist als Technische Informatik bezeichnet.

Theoretische Informatik : In diesem Gebiet werden die allgemeinen Rahmenbedingungen und die Grundlagen der Informatik vorgestellt, wie etwa Formale Sprachen, die Abstraktionstheorie oder mathematische Verfahren.

Die Veränderungen in diesem Gebiet sind recht langsam, was einmal an dem hier verwendeten Abstraktionsgrad liegt und an der für „Neuentwicklungen“ benötigten Zeit.

Stellt dieses Gebiet den Schwerpunkt des Studiengangs dar, wird dieser meist als Diskrete Mathematik oder theoretische Informatik bezeichnet.

Diese Vorlesung ordnet sich in den Bereich der Konstruktionslehre ein. Dabei werden natürlich für die praktischen Aufgaben Werkzeuge aus dem Bereich der praktischen Informatik verwendet.

Lernziele

Sie sollen in dieser Veranstaltung Kenntnisse zur Bewertung und selbstständigen Entwicklung von Algorithmen erwerben und die dazu erforderlichen Datenstrukturen kennen und einsetzen lernen.[1] Insbesondere sollen Sie am Ende des Semester diese Fähigkeiten erworben haben:

- Elementare Datenstrukturen wie Liste, Stack, Queue, Binärbaum etc. kennen und unter Berücksichtigung ihrer Vor- und Nachteile einsetzen können.
- Die Komplexität von Algorithmen untersuchen und daraus die richtigen Konsequenzen ziehen können.
- Grundprinzipien von Sortierverfahren und die Standard-Sortierverfahren und ihre Eigenschaften kennen.
- Baumstrukturen kennen und einsetzen können.
- Ausgewählte Graphenalgorithmen (Dijkstra, Kruskal) kennen.
- Grundlegende Hashverfahren kennen.
- Ausgewählte Verfahren zur Komprimierung und zur Verschlüsselung kennen.
- Die vorgestellten Strukturen und Algorithmen softwaretechnisch angemessen implementieren können.

[1] Aus der Modulbeschreibung TI AD

Inhaltsübersicht

Abstrakte Datentypen

- Abstrakte Datentypen, Signatur, Vor- und Nachbedingungen
- Lineare Listen, Stack, Queue

Hier besprechen wir Abstrakte Datentypen (ADT). Wir lernen die grundlegenden Strukturen *Liste, Stack* und *Queue* kennen, lernen verschiedene Implementierungsmöglichkeiten und deren Vor- und Nachteile kennen.

Algorithmen und Komplexität

- Komplexität, Aufwandsfunktion, asymptotischer Aufwand
- Landau-Notation ($\mathcal{O}$ (Groß O), Ω, Θ)
- Darstellung in logarithmischen Skalen
- Rekursive Verfahren

Hier geht es um Algorithmen: Wie *komplex* (aufwändig) ist ein Algorithmus? Wie misst man das? Wie vergleicht man die Komplexität von Algorithmen? Hier benötigen wir Kenntnisse aus der Analysis.

Sortierverfahren

- Nicht-rekursive Sortierverfahren und deren Komplexität
- Rekursive Sortierverfahren (Quicksort und Mergesort) und deren Komplexität

Hier herrschen eigentlich klare Verhältnisse. Totzdem muss ein Informatiker die Grundprinzipien verstanden haben, denn für spezielle Aufgaben geht es ggf. auch besser. Das Sortierverfahren *Heapsort* bildet dann eine natürliche Überleitung zum nächsten Kapitel.

Bäume und Graphen

- Implementierungsmöglichkeiten
- Binäre Suchbäume und Komplexität
- Graphen und kürzeste Wege (Dijkstra-Algorithmus)

Hier geht es wieder um Strukturen: diesmal um Bäume und Graphen. Wir lernen wichtige Anwendungen kennen. Die Komplexitätsanalyse kommt uns auch hier wieder zugute.

Hashfunktionen

- offene Adressierung und separate chaining
- Kollisionsvermeidungsstrategien

Dieses wichtige Konzept erlaubt uns, besonders schnell an unsere Daten zu kommen. Dabei gibt es jedoch ein paar Tricks und Kniffe.

Komprimierung und Kryptographie

- Verlustfreie Komprimierung
- Ausgewählte Kryptographieverfahren

Komprimierung und Verschlüsselung von Daten ist angesichts der anfallenden Fülle von Daten und Datenübertragungen in wichtiges Thema. Einige der grundlegenden Prinzipien sollte jeder Informatiker kennen.

Fehlerbehandlung

Selbst wenn ein Algorithmus gemäß seiner Spezifikation korrekt ist, so gibt es mehr als genug Fälle, in denen eine konkrete Implementation dennoch abstürzt, weil irgendwelche Daten doch nicht den Anforderungen genügen. Jeder Mensch schützt sich, das Programm und die Umwelt vor solchen Unglücken durch eine ordentliche Fehlerbehandlung. Wie jeder weiß, macht eine solche Fehlerbehandlung in der Praxis einen recht großen Teil des fertigen Programmcodes aus und der algorithmische Kern ist dann evtl. nur unter Schwierigkeiten zu erkennen.

Genau das soll hier natürlich nicht passieren. Aus diesem Grund wird in (fast) allen Algorithmen auf eine explizite Fehlerbehandlung verzichtet! Jeder, der den einen oder anderen Algorithmus implementiert, ist aufgefordert, eine geeignete Fehlerbehandlung selbst vorzunehmen. In diesem Sinne sind in dieser Vorlesung SE-Verfahren anzuwenden, insbesondere im Praktikum. Diese werden hier jedoch nicht vermittelt.

Konstruktive Kritik und Verbesserungs- bzw. Ergänzungsvorschläge oder auch Lösungsvorschläge für Aufgaben sind herzlich willkommen. Sie können sie an den jeweiligen Veranstalter schicken.

André Jestel hat im WS 2008/9 viele konstruktive Beiträge zur Verbesserung des Skripts geleistet. Im SS 2009 kamen viele konstruktive Beiträge von Mehmet Bulut.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Abstrakte Datentypen

1.1 Definitionen

In der Informatik werden - wie in der Mathematik auch - Strukturen anhand ihrer Eigenschaften definiert. Konkrete Beispiele der Strukturen werden dann als Instanzen, Objekte oder Implementierungen bezeichnet. Sie kennen dieses Konzept schon aus der objektorientierten Programmierung: eine *Klasse* beschreibt, wie die konkreten Instanzen, die *Objekte*, auszusehen und sich zu verhalten haben.

Wir definieren einen Datentyp indem wir bestimmte Eigenschaften angeben, die jede Implementierung des Datentyps haben soll. Die Eigenschaften werden durch Operationen auf einer Wertemenge definiert.

Definition 1.1 (Abstrakter Datentyp)
Ein *Abstrakter Datentyp* besteht aus einer oder mehreren Objektmengen (auch Wertebereich oder Wertemenge genannt) und Operationen, die darauf definiert sind. Die Operationen werden auch Funktionen oder Methoden genannt. Man spezifiert die Operationen über ihre **Signatur** zusammen mit einer **Semantik**. Möchte man die Spezifikation (Signatur, Semantik) von der Implementierung unterscheiden, so wird letztere oft Methode genannt. Eine Methode ist dann die Implementierung einer Operation. Wichtig ist hier besonders die Unabhängigkeit von einer konkreten Programmiersprache oder Implementierung. ◄

Man bekommt also eine Abstraktionsebene, bei der Objekte anhand ihrer Eigenschaften identifiziert werden. Tatsächlich haben wir es mit drei Abstraktionsebenen zu tun:

Abstraktionsebene	Name	Beispiel
2	Abstrakter Datentyp	Integer
1	Implementierung	32-bit Integer
0	Objekt	`int i = 5;`

Sehen wir uns erstmal die Eigenschaften der atomaren Datentypen an. *Atomare Datentypen* sind Datentypen, die in einer konkreten Programmiersprache bereits vorhanden sind. Hier einige Beispiele:

Bezeichnung	Datentyp	Elementare Operationen
Wahrheitswerte	`bool`	and , or ,=
Ganze Zahlen	`int`	+,−,∗, /, abs(), ++, <,>,! =
Reelle Zahlen	`float`, `double`	+, −,∗, /, sin(), tan(), <, >
Buchstaben	`char`	=, ! =, atoi()
Zeiger	`void*`	=, ! =, ∗

Beispiel 1.2 (Datentyp Integer)
Der Datentyp Integer `int` kann wie folgt dargestellt werden:

Objektmengen	int
	bool
Operationen	$+ : \text{int} \times \text{int} \rightarrow \text{int}$
	$(a, b) \mapsto a + b$
	...

An dieser Stelle müssten jetzt alle Operationen aufgelistet werden.

Gemeint ist bei der Operation + eine Abbildung $+ : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$, welche das kartesische Produkt - daher das $\times$ Zeichen - der ganzen Zahlen mit den ganzen Zahlen in die ganzen Zahlen abbildet. Anders ausgedrückt: es sind zwei Eingabeparameter erforderlich, welche beide vom Typ `int` sein sollen. Es gibt einen Ausgabeparameter, der wiederum ein `int` ist. Durch die Menge aller Operationen (und deren Signatur und Semantik) die man mit einem Integer durchführen kann, wird der Datentyp *Integer* definiert. Diese Definition ist unabhängig von jeglicher Programmiersprache oder Implementierung. ◀

Bemerkung 1.3 (Parameter-Arten)
Aus der Programmierung kennen Sie den Unterschied von Referenz- und Wertsemantik. In UML kann man bei der Spezifikation von Operationen noch weiter unterscheiden: *in*, *out* oder *inout*, um zum Ausdruck zu bringen, dass ein Parameter übergeben, aber nicht verändert wird, nicht übergeben, aber zurückgegegeben, bzw. übergeben und u. U. verändert zurückgegeben wird. ◀

Aufgabe 1.4
Welche wichtigen Unterschiede gibt es zwischen $\mathbb{Z}$ und dem Datentyp `int` in einer Programmiersprache wie C++ oder Java? Welche Unterschiede gibt es zwischen $\mathbb{R}$ und den Datentypen `float` bzw. `double` in Java bzw. C++? ◀

Definition 1.5 (Signatur)
Das Abbildungsverhalten einer Operation nennen wir Signatur[1]. Als Schreibweise verwenden wir: Operationsbezeichnung : Menge Inputparameter → Menge Outputparameter Als Input- und Outputparameter verwenden wir Elemente des Wertebereichs. Diese bezeichnen wir mit Typenbezeichnungen wie sie uns aus der Programmierung bekannt sind.

Wird ein Objekt *Meintyp* erzeugt oder zerstört[2], so schreiben wir

$$\text{ctor} : \emptyset \rightarrow \text{Meintyp oder delete} : \text{Meintyp} \rightarrow \emptyset$$

◀

Beispiel 1.6 (Signatur)
Weitere Beispiele:

Operation *	:	int	×	int	→	int
Operation ==	:	int	×	int	→	bool
Operation ++	:			int	→	int

◀

Eine Signatur beschreibt den syntaktischen Aspekt einer Operation. Es wird das Eingabe/Ausgabeverhalten definiert. Darüber hinaus benötigt man meist aber auch eine Beschreibung der Semantik. Dies kann durch Nachbedingungen geschehen, aber in vielen Fällen ist eine textuelle Beschreibung oder in mathematischen Kontexten eine Formel durchaus angemessen.

[1] Aus implementatorischer Sicht ist die Signatur einer Funktion anders definiert, nämlich als Funktionsname, das n-Tupel der Übergabeparameter und Rückgabewert. Siehe Beispiel auf Seite 5. Bibliothekare definieren Signatur ganz anders und das, was auf Ihrem HAW-Bibliotheksausweis so bezeichnet wird, ist einfach eine Unterschrift.

[2] Aus mathematischer Sicht ist eine Abbildung, deren Definitions- oder Wertebereich die leere Menge ist, nicht sinnvoll. Dies ist also nicht als Abbildung aufzufassen, sondern eher als eine Art Pseudocode oder Stenogramm.

Aufgabe 1.7
Schreiben Sie die Signaturen für folgende Operationen des Typs int auf:

$$*= \qquad \leq \qquad \%$$

◀

Beispiel 1.8 (Stack)
ADT Stack:

Objektmengen	STACK	
	ELEMENT	
	BOOL	
Operationen	$push : STACK \times ELEMENT$	$\rightarrow STACK$
	$pop : STACK$	$\rightarrow STACK$
	...	

Letzteres ist die Signatur für eine seiner Operationen *push*. Ein Stack (in dem evtl. schon bestimmte Elemente gespeichert sind) zusammen mit einem Element ergeben einen Stack, in dem ganz bestimmte Elemente gespeichert sind. Welche das genau sind und was das Element beim Stack bewirkt, wird erst in der Semantik definiert. ◀

Es gibt viele verschiedene Möglichkeiten, die Operationen auf einem Stack festzulegen. Wofür man sich entscheidet, hängt von der jeweiligen Situation ab. Für die Operation *push* gibt es wohl keine anderen Möglichkeiten als die in Beispiel 1.8 angegebene. Bei der Operation *pop* hat man mindestens zwei Möglichkeiten:

1. $pop : Stack \longrightarrow Stack$, also nur Entfernen und kein Zurückgeben des obersten Elements bzw. einer Referenz darauf. Dann benötigt man auch eine Operation $top : Stack \longrightarrow Element$. Diese liefert dann dass oberste Element zurück, ohne es zu entfernen. Diese Operation wird auch *peek* genannt.

2. $pop : Stack \longrightarrow Stack \times Element$, also das Zusammenfassen von *pop* und *top* in einer Operation.

Die üblichen Kriterien aus dem Softwareengineering sprechen für die Variante 1, wegen des höheren Zusammenhalts der Operationen. Dies entspricht auch dem verbreiteten Programmierstil, dass eine verändernde Operation nichts zurückliefert und eine Operation, die etwas zurückliefert, keine Veränderung vornimmt ([Mey97]).

Bemerkung 1.9 (Stack und Synonyme)
Für *Stack* waren oder sind viele andere Begriffe in Gebrauch. Aufbauend auf [Knu97a] seien hier genannt: *push-down list*, *reversion storage*, *Keller* (*cellar*), *nesting store*, *pile*, *LIFO-Liste*, *Jo-Jo Liste*. Die vielen Begriffe deuten darauf hin, das dieses Konzept in vielen Kontexten einsetzbar ist. Besonders ist mir der Begriff Keller-Speicher aufgefallen, der mir zunächst überhaupt nicht einleuchtete, weil ich ihn mit dem Namen einer Person in Verbindung brachte. Ein Keller ist für mich ein zwar möglicherweise schlecht belichtetes, kühles, eventuell auch feuchtes, Untergeschoss, aber immerhin ein Geschoss. Ein Stack ist dann eher ein Kriechkeller, wie er in Gegenden mit schweren oder feuchten Böden, z. B. in Dithmarschen vorkommen. Aus einem solchen entnehme ich vielleicht wirklich als nächste Flasche die zu letzt eingelagerte. ◀

Es gibt verschiedene Möglichkeiten zur Beschreibung der Semantik einer Operation: axiomatisch oder algebraisch. Wir wollen nur die algebraische Methoden verwenden. Für eine systematische Behandlung der axiomatischen Beschreibung von abstrakten Datentypen sei z. B. auf [GZ06] verwiesen. Hier nimmt man die exakte mathematische Notation zuhilfe um die Semantik einer Operation zu beschreiben. In Anlehnung an OCL wollen wir mit Vor- und Nachbedingungen (Pre-, Postconditions) arbeiten, wo dies als sinnvoll erscheint.

Aufgabe 1.10
Finden Sie bitte Antworten auf folgende Fragen:

1. Wofür steht die Abkürzung OCL?
2. Von welcher größeren Spezifikation ist OCL ein Teil oder welche Spezifikation wird durch OCL ergänzt?
3. Von wem wurde OCL entwickelt?
4. Für welchen Zweck ist OCL entwickelt worden?
5. Geben Sie ein Beispiel für OCL an!

◄

Definition 1.11 (Vorbedingung, Nachbedingung)
Eine *Vorbedingung* (precondition) gibt Eigenschaften der Eingabeparameter an, mit denen die Operation fehlerfrei anwendbar ist. Der Anwender des ADT muss sicher gehen, dass diese Eigenschaften erfüllt sind. Zur Sicherheit sollte der Entwickler des ADTs Abfragen einbauen (assert).

Eine *Nachbedingung* (postcondition) gibt die Eigenschaften des Objekts nach Beendigung der Operation an (das Ergebnis der Operation). Dies wird oft in mathematischer Schreibweise spezifiziert, da sie sehr präzise ist und man mathematische Aussagen oft leicht in ein Computerprogramm umwandeln kann. ◄

Bemerkung 1.12
(Object Constraint Language) ist eine Sprache, die formal aufgebaut ist und es erlaubt, Bedingungen an Objekte und Operationen zu formulieren. Wir wollen uns nicht streng an OCL halten, sie kann uns aber eine gute Richtschnur sein. Was hier sinnvoll verwendet werden kann, wird bei Bedarf eingeführt. Für Einzelheiten sei auf die Literatur verwiesen. ◄

Bemerkung 1.13
Es wurde von IBM einmal der Versuch unternommen, für die Spezifikationssprache ein Fragment der Prädikatenlogik erster Stufe zu verwenden, mit dem Ziel, die Beschreibungen dann in PROLOG (einer logischen Programmiersprache) 1:1 zu implementieren. Das funktionierte auch hervorragend. Das IBM dies nicht wiederholt hat, liegt letztlich an dem Problem, das der Übergang der informellen Objekte im Kopf zu den formalen Objekten im Rechner nun von der Implementierung in die Spezifikation der Vor- und Nachbedingungen „vorverlagert“ wurde und ihre Entwickler in dem Sinne nun auf z.B. die Programmiersprache PROLOG umgeschult werden mussten. Von daher ist auch zu beobachten, dass in der Praxis häufig eine informale Sprache zur Beschreibung dieser Bedingungen verwendet wird (da einfacher zu formulieren) und die Bedingungen nicht immer an einer Stelle explizit zu finden sind, sondern sich oft in den Kommentaren des Codes verteilen.

Hier liegt ein Dilemma der Informatik: Die Beschreibung der Vor- und Nachbedingungen sind zusätzliche Arbeiten zu der eigentlichen Implementierung, d.h. man versucht durch teilweise recht aufwendige Dokumentation die Systeme (funktions-)sicherer zu machen. Meist wird dies aber aus Zeitgründen von den Entwicklern nicht sorgfältig durchgeführt, bzw. überhaupt durchgeführt. Haben Sie für alle Ihre Methoden/Funktionen Vor- und Nachbedingungen formal beschrieben ? Oder haben Sie die Dokumentation mit der Implementierung abgeglichen bzw. geprüft ob Ihr Kommentar und der zugehörige Code übereinstimmen und dies bei Änderungen gepflegt ?.

Dies führt zu neuen IDE-Systemen, die hier möglichst viel automatisieren und dies führt zu neuen SE-Verfahren, z.B. einer modellgetriebenen Entwicklung (z.B. MDA), mit der Idee (siehe IBM): die Dokumentation bzw. Spezifikation des Problems/Lösungswegs/etc. *ist* die Implementierung. ◄

Beispiel 1.14 (ADT Stack)
Nochmal das Beispiel Stack. Wir definieren

$$\text{STACK} \quad s = (x_1, \ldots, x_k), \quad x_i \in \text{ELEMENT}.$$

Außerdem: Maximale Stackgröße MAX. Nun definieren wir den abstrakten Datentypen zunächst mit der Operation *push*:

Objektmengen		STACK
		ELEMENT
		BOOL
Operationen	$push$:	$STACK \times ELEMENT \to STACK$
	pre:	$k <$ MAX
	post:	$push(s, e) = (x_1, \ldots, x_k, e)$ wobei $s \in$ STACK,
		$e \in$ ELEMENT

Die Operation *pop* spezifizieren Sie bitte als Übungsaufgabe. ◀

Bemerkung 1.15 (Mathematik und OO)
Beachten Sie die Unterschiede in dieser Schreibweise und in objektorientierten Programmiersprachen: Hier betrachten wir eine Abbildung `push` von `Stack`×`Element` in `Stack`. In Java oder C++ schreiben Sie aber für ein Objekt `myStack` der Klasse `Stack`: `myStack.push(e)`. Die Operation operiert auf den Attributwerten des Objekt und den übergebenen Parametern. ◀

Beispiel 1.16 (Spezifikation von Operationen)
Weitere Beispiele für die Spezifikation von Operationen:

Operation +:	int × int → int ; $(a, b) \mapsto a + b$
pre:	keine
post:	$a + b$ ist die Summe der Zahlen a und b

Operation /:	float × float → float ; $(a, b) \mapsto a/b$
pre:	$b \neq 0$
post:	a/b ist der Quotient aus den Zahlen a und b

Im letzten Beispiel könnte man alternativ auch schreiben:

Operation /:	float × float → float ∪ { error } ; $(a, b) \mapsto a/b$
pre:	keine
post:	a/b ist der Quotient aus den Zahlen a und b
	Falls $b = 0$ wird *error* zurückgeliefert

Hier hat man natürlich das Problem, in welcher Form man *error* zurückliefern möchte, z. B.

```
float division(float a, float b, ErrObj* errobj);
```

Es ginge aber auch

```
ErrObj* division(float a, float b, float* erg);
```

Welche der beiden Möglichkeiten gemeint ist, geht aus der Schreibweise nicht hervor. Aus implementatorischer Sicht haben beide Funktionen natürlich eine unterschiedliche Signatur, obwohl das Input / Output-Verhalten gleich ist.

Außerdem zwingt man den Anwender dieser Funktion auch, den Rückgabewert jedesmal zu überprüfen, z. B.

```
erg = division(a,b,errobj);
if ( errobj != 0 ) /* hier muss eine Fehlerbehandlung her */
```

Die Einführung von Exceptions diente unter anderem dazu, dieses Problem zu entschärfen. ◀

Beispiel 1.17 (ADT Polynom)
Auf der Menge der Polynome sind auch gewisse Rechenoperationen erlaubt. Zum Beispiel:

$$\begin{aligned}
\text{Operation } +: &\quad Polynom \times Polynom \to Polynom \\
&\quad (p, q) \mapsto p + q \\
\text{pre:} &\quad \text{Grad der Polynome gleich } \operatorname{grad}(p) = \operatorname{grad}(q) =: N \\
\text{post:} &\quad \text{sei } p = \sum_{i=0}^{N} a_i \cdot x^i,\ q = \sum_{i=0}^{N} b_i \cdot x^i, \\
&\quad \text{so ist: } p + q = \sum_{i=0}^{N} (a_i + b_i) \cdot x^i
\end{aligned}$$

◀

Aufgabe 1.18
Schreiben Sie obiges Beispiel so um, dass die Vorbedingung fallen gelassen werden kann! ◀

Auch Klassen in nicht-algorithmischen Anwendungen haben den Aspekt des ADT.

Beispiel 1.19 (ADT Student)
Abbildung 1.1 zeigt einen trivialisierten Ausschnitt aus einem landesweiten Hamburgischen Stu-

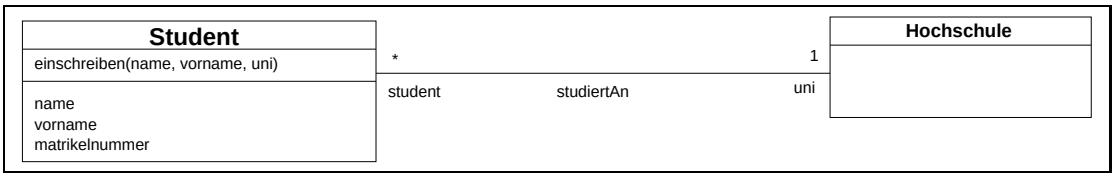


Abbildung 1.1: Ausschnitt Studierendenverfahren

dierendenverfahren. Auf der Menge der Studenten sind gewisse Operationen definiert. Bei der Wahl der Schnittstelle hat man verschiedene Möglichkeiten. Hier ist die Variante gezeigt, in der es eine create-Operation gibt, die ein leeres Objekt erzeugt.

$$\begin{aligned}
\text{Operation erzeuge:} &\quad \emptyset \times \text{Hochschule} \to \text{Student} \quad ; \quad \text{Student s;} \\
\text{pre:} &\quad \text{Objekt vom Typ Hochschule wurde vorher erzeugt} \\
\text{post:} &\quad \text{s.name =null, s.vorname = null, s.matr = null;}
\end{aligned}$$

Die Vorbedingung kommt hier aus der Multiplizität am *uni*-Ende der binären Assoziation *studiertAn*.

$$\begin{aligned}
\text{Operation einschreiben:} &\quad \text{Student} \times \text{string} \times \text{string} \to \text{Student} \\
&\quad \text{s = s.einschreiben(name, vorname, uni);} \\
\text{pre:} &\quad \text{name enthält keine Ziffern} \\
\text{pre:} &\quad \text{vorname enthält keine Ziffern} \\
\text{post:} &\quad \text{die Daten von } s \text{ werden überschrieben} \\
&\quad \text{s.name = name} \\
&\quad \text{s.vorname = vorname} \\
&\quad \text{s.matr = neu und eindeutig generiert}
\end{aligned}$$

◀

ADTs sollen folgende Eigenschaften haben:

1. **Universalität** Dies bedeutet, der Datentyp soll von Implementierung und Programmiersprache unabhängig sein. Auch innerhalb einer Programmiersprache soll ein einmal entwickelter ADT in jedem beliebigen Programm verwendet werden können.

2. **Präzise Beschreibung** Durch eine exakte algebraische oder axiomatische Beschreibung lässt sich ein ADT gut implementieren.

3. **Kapselung** Die interne Form (Implementierung und Representation) ist für den Anwender nicht sichtbar (Geheimnisprinzip, Kapselung). Er sieht nur das Interface.

Folgende Operationen werden in der Regel von einem ADT benötigt:

1. **Initialisierung** Das Objekt wird erzeugt und auf bestimmte Weise vorbelegt.
2. **Nicht-destruktive Operationen** Das Objekt wird betrachtet oder gelesen.
3. **Destruktive Operationen** Das Objekt (insb. sein innerer Zustand) wird verändert oder beschrieben.
4. **Funktionen** Aus einem oder mehreren Objekten entsteht ein neues.
5. **Zerstörung** Beseitigen des Objekts und Freigabe des Speichers.

1.2 Lineare Listen

Listen sind ein klassisches Beispiel für einen ADT und auch die Grundlage für viele weitere ADTs. Wir werden später aus einer Liste mit eingeschränkter Funktionalität noch weitere ADTs entwickeln.

Lineare Listen basieren auf dem mathematischen Konzept einer (endlichen) Folge. Im Unterschied zu Mengen gibt es auf Folgen eine Ordnung: es gibt ein erstes, zweites, drittes Element etc. Die Elemente bezeichnen wir mit $a_1, a_2, \ldots, a_n$. Außerdem sind Duplikate von Elementen erlaubt. Wir betrachten Folgen von Objekten eines bestimmten Grundtyps. Dieser Grundtyp könnte z. B. ein Druckauftrag sein oder Wörterbucheinträge, was immer man eben in einer Liste abspeichern will.

Listen treten in vielen Situationen auf, ob nun ein Rechner beteiligt ist oder nicht. Sind Sie etwas dabei etwas zu erledigen, so kann es passieren, dass Sie weiteres zur Erledigung bekommen oder selbst entdecken. Sie können dies aber oft nicht gleich in Angriff nehmen. Also tragen Sie es in eine Aufgabenliste („Todo-List“) ein. Je nach Art der Aufgaben und Ihren eigenen Neigungen mag es sinnvoll sein, diese Aufgaben in der Reihenfolge, wie sie aufgetreten sind zu bearbeiten (FIFO, first in first out) oder das noch neueste zu erst, dann das zweitletzte etc (LIFO, last in first out). Erste Strategie führt auf eine *Queue*, letztere auf einen *Stack*.

Unabhängig vom Grundtyp wollen wir die Eigenschaften der Liste untersuchen.

Voraussetzung: Beliebiger fester Grundtyp G mit folgenden Merkmalen:

- Schlüssel (z.B. Integer)
- eigentliche Information (beliebiger benutzerdefinierter Typ)

Als Eigenschaften für den ADT Liste wünscht man sich, dass man neue Elemente an gegebener Position einfügen und entfernen kann. Außerdem möchte man Elemente suchen können.

Definition 1.20 (ADT Liste)
Im folgenden sei

- POS (für Position) die Menge der erlaubten Schlüssel,
- $ELEM$ die Menge der erlaubten Elemente vom Grundtyp
- $LIST$ die Menge der möglichen Listen

Der ADT Liste zeichnet sich durch folgende Operationen aus:

- Einfügen eines neuen Elements $x \in G$ (vom Grundtyp) in die Liste L an Position $p \in pos$ (insert oder insertAt)
- Entfernen eines Elements $p \in pos$ aus der Liste L (delete oder deleteAt)

- Suchen der Position $p \in pos$ eines Elements $x \in G$ in Liste L (find)
- Zugriff auf Element an der Position $p \in pos$ in Liste L (retrieve)
- Verkettung zweier Listen L_1 und L_2 (concat)

Hier die Definition des

ADT Liste

Objektmengen POS
ELEM
LIST

Operationen

insert	:	LIST	×	POS	×	ELEM	→	LIST
delete	:	LIST	×	POS			→	LIST
find	:	LIST	×	ELEM			→	POS
retrieve	:	LIST	×	POS			→	ELEM
concat	:	LIST	×	LIST			→	LIST

Vor- und Nachbedingung beispielhaft für insert:

Operation insert:	LIST × POS × ELEM → LIST; L.insert(p,x)
pre:	$p \in \{p_0, \ldots, p_n\}$ erlaubte Positionen
post:	Sei $L = (a_0, \ldots, a_n)$ eine Liste. Sei a_i das Element an Position p_i. Dann bewirkt $L.insert(p_i, x) = (a_0, \ldots, a_{i-1}, x, a_i, a_{i+1}, \ldots, a_{n+1})$

◀

In der Literatur gibt es verschiedene Definitionen für Operationen des ADT Liste. Viele der folgenden zählt Knuth in [Knu73a] auf:

1. Zugriff auf den k-ten Knoten, um dessen Daten zu lesen oder zu ändern. (siehe retrieve)
2. Ersetzen des k-ten Knotens durch einen neuen Knoten.
3. Füge einen Knoten vor oder nach dem k-ten ein (Präzisierung von insert)
4. Lösche den k-ten Knoten (delete)
5. Füge zwei Listen zusammen (concat)
6. Zerlege eine Liste in zwei (divide)
7. Kopiere eine Liste (clone)
8. Bestimme die Anzahl Knoten in einer Liste (count)
9. Sortiere die Knoten aufsteigend (sort)
10. Durchsuche die Liste nach einem Wert in den Daten der Knoten (find)

Je nach Anwendung wird man die eine oder die andere Operation favorisieren und besonders effizient implementieren wollen. Das kann dann den Ausschlag für die Wahl sowohl der Schnittstelle als auch der Implementierung geben.

Das Interface `List` in Java umfasst z. B. 25 Operationen. Implementiert wird dieses Interface von `AbstractList`, `ArrayList`, und `LinkedList` Auch `Vector` implementiert dieses Interface, ist aber veraltet und sollte nicht mehr verwendet werden.

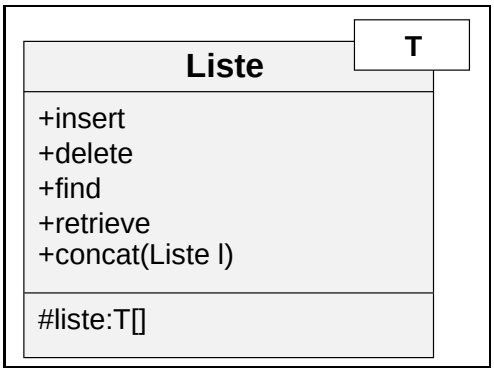


Abbildung 1.2: Array-Implementierung einer Liste

1.2.1 Implementierung 0: Sequenzielle Speicherung

Eine einfache Implementierung verwendet ein Array zur Speicherung der Listenelemente und wird sequentielle Speicherung genannt.

```
public class List0<T>{
  public ...
  private T liste[];
  private int listSize;
  private int maxSize;
}
```

Listenelemente sind gespeichert an den Positionen $1, \ldots, listSize$

Position 0 ist eine uneigentliche Listenposition. Sie ermöglicht eine Suchoperation mit Stopper: Vor Beginn der Suche wird das gesuchte Element x an die Position 0 geschrieben. Gesucht wird dann ab *listSize* rückwärts. Position 0 dient dann als Stopper im Falle einer erfolglosen Suche.

Der Code für die Suchfunktion sieht so aus:

```
public int find(T o) {
   int i=listSize;
   liste[0] = o;
   while(liste[i]!=o)
   {
       i--;
   }
   return i;
}
```

Die Stoppertechnik vermeidet die sonst notwendige Abfrage, ob die Laufvariable `i` noch im zulässigen Bereich liegt. Sind die Elemente nach aufsteigenden Schlüsselwerten sortiert, so gibt es effiziente Verfahren zum Suchen eines Elements.

Beim Einfügen eines neuen Elements müssen die nachfolgenden Elemente verschoben werden.

- Beim Einfügen und Löschen müssen Elemente verschoben werden. Dies führt bei großen Listen zu viel Rechenaufwand, der proportional zur Anzahl n der Elemente wächst ($\mathcal{O}(n)$). Günstigster Fall: am Ende einfügen. Ungünstigster Fall: am Anfang einfügen.

- Die Komplexität $\mathcal{O}(n)$ ist unabhängig davon, ob die entsprechende Position gegeben ist, oder durch Suchen erst gefunden werden muß.

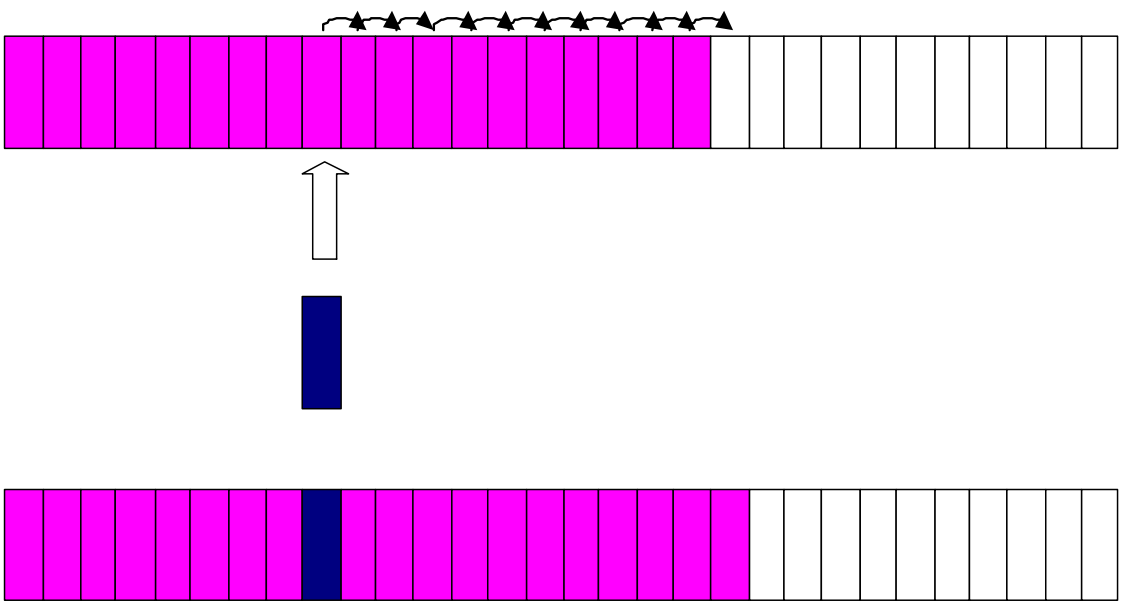


Abbildung 1.3: Einfügeoperation bei Listen als Arrayeinbettung.

Die Bedeutung von $\mathcal{O}$ wird später genau definiert.

Dies mag auf den ersten Blick ungünstig erscheinen, aber so ungünstig ist eine Speicherung im Array nicht. Es hängt immer davon ab, welche Operationen man favorisieren will oder muss. Die STL (Standard Template Library in C++) implementiert z. B. einen `vector` so, dass sie einen grossen Speicherblock allokiert, und dann die Elemente dort hineinkopiert. Auch für eine Liste eignet sich unter Umständen ein sogenannter *contiguous memory block*. In diesem Fall ist die Reihenfolge der Elemente allerdings nicht durch das Array vorgegeben, sondern durch Zeiger in jedem Element, welche auf das folgende Element zeigen. Diese Verzeigerung kann natürlich auch durch Indexrechnung im Array geschehen. Wird ein Element der Liste gelöscht, dann werden keine Elemente verschoben, sondern das frei gewordene Element wird in eine zweite Liste eingehängt, die sog. *Freiliste*, die angibt, dass das Element frei ist und wieder verwendet werden kann. Das Array wirkt also als eine Speicherverwaltung. Dies ist eine Implementierung des sog. *Pool Allocation Patterns*, bei dem ein Pool von gleich großen Speicherblöcken allokiert wird. Jedes Element im Array wird als ein Container für die eigentlichen Elemente verwendet (daher contiguous memory container). Diese Speicherverwaltung hat gegenüber einer eingebauten Speicherverwaltung den Vorteil, dass sie besonders schnell ist ($O(1)$). Siehe dazu den Abschnitt 1.4.3.

Java bietet im Paket *java.util* eine Implementierung mit sequenzieller Speicherung in der Klasse *LinkedList*.

1.2.2 Implementierung 1: einfach verkette Listen

Abbildung 1.4 zeigt die Struktur einer einfach verketteten Liste. Eine solche Liste ist eine Folge von Knoten: Jeder Knoten enthält ein Listenelement des Grundtyps und einen Zeiger auf das nächste Listenelement. Der Code für einen Knoten sieht etwa folgendermaßen aus:

```
class Knoten{
      public T daten;
      public Knoten next;
      public Knoten(T d){
         daten = d;
         next = null;
      }
```

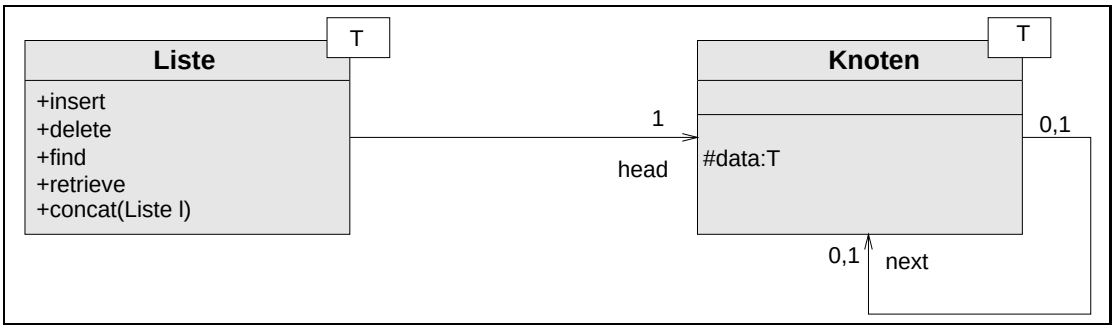


Abbildung 1.4: Verkette Liste, Variante 1

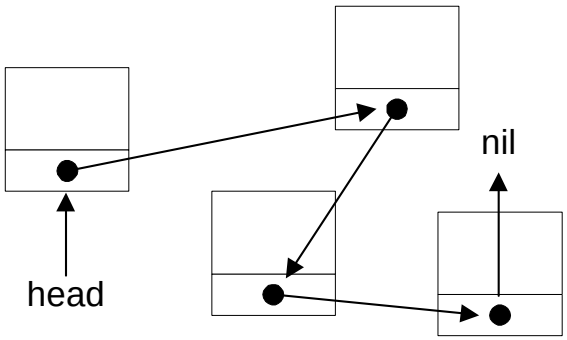


Abbildung 1.5: head- und tail-Referenzen bzw. Zeiger einer einfach verketteten Liste.

```
}
```

Ich habe diese Definition von `Knoten` aus einem Kontext kopiert, in dem ich diesen als innere Klasse einer Klasse Liste verwende. Die Attribute sind als öffentlich deklariert, um die Code-Beispiele schlank zu halten. Die Template-Variable heißt `T`, kommt aus `List1<T>`, `Knoten` wird nicht „extra“ parametrisiert.

Bemerkung 1.21 (Implementierung von Knoten)
Die genauen Details der Implementierung einer Klassen *Knoten* hängt von der Umgebung ab. In Java ist oft eine statische innere Klasse sinnvoll. ◄

Die Liste ist eine Template-Klasse (parametrisierte Klasse), die eine Referenz oder einen Pointer auf ein Objekt der mit dem Grundtyp T parametrisierte Klasse Knoten hat. Ein Objekt der Klasse Knoten hat eine Referenz auf den nächsten Knoten oder *null*, wenn es das letzte Element der Liste ist. Wir müssen definieren, was mit dem ersten und letzten Element geschehen soll: Wir nehmen hier eine einfach verkettete Liste mit Zeigern `head` auf Listenanfang. Das Listenende wird als `null` markiert, d. h. der letzte Knoten zeigt auf den Nachfolger (`next`) `null`. Die Liste in dieser Implementierung stellt sich dann so dar:

```
public class List1<T>{
// insert, delete find etc.
   ...
   private Knoten<T> head;
   ...
}
```

Die Liste ist genau dann leer, wenn `head = null` ist.

Die Operation *Listenelement suchen* muss dann folgendes tun:

1. Überprüfen, ob `head = null`, also ob die Liste leer ist.
2. Überprüfen, ob `Knoten.next = null`, also ob man das Listenende erreicht hat.

Dies sind zwei zusätzliche Überprüfungen gegenüber der Version mit Stopper-Element.

```
public Knoten find(T d) {
   if(head == null)
      return null;
   else
     {
        Knoten<T> pos = head;
        while ( (pos.daten != d) && (pos.next != null))
             pos = pos.next;
             // an dieser Stelle ist pos.daten = d oder
     // pos.next = null
     if ( pos.daten ==d)
         return pos;
     else
         return null;
     }
}
```

Vergleicht man die sequentielle Implementierung und die mit einfacher Verkettung, so stellt man fest, dass das Aufsuchen einer gegebenen Position in der sequentiellen Implementierung einfacher erscheint. Bei der sequentiellen Implementierung wird Speicher genutzt, auch wenn er noch nicht belegt ist; die Implementierung mit verketteten Knoten erlaubt ein einfaches dynamisches Wachstum. Mit dem Konzept des Iterators kann können aber einige Operationen effizienter gestaltet werden.

Kann man das nicht noch besser machen, etwa mit weniger Aufwand, je nach favorisierter Operation? Ja, das geht! Einige Operationen lassen sich mit folgender Implementierung drastisch beschleunigen.

1.2.3 Implementierung 2: einfach verkettete Liste mit tail-Pointer und antizipativer Indizierung

Bei dieser Technik wird die Position ganz anders verwendet: Nicht als ganzzahlige Position, sondern als Referenz, Pointer o. ä. auf einen Knoten. Position werden hier also nicht - wie bei sequentiellen Listen - als laufende Nummer angegeben, sondern durch einen Zeiger (Referenz, Iterator) auf das Listenelement. Die Klasse Liste in Abb. 1.6 enthält:

- Referenz auf head
- Referenz auf tail
- beide zeigen auf Dummyelemente
- eigentliche Listenelemente liegen dazwischen

Die Knoten werden identisch zu Implementierung 1 realisiert. Die Liste selbst stellt sich so dar:

```
class List2
{
      Knoten head;
      Knoten tail;
public:
     // insert, find, etc.
}
```

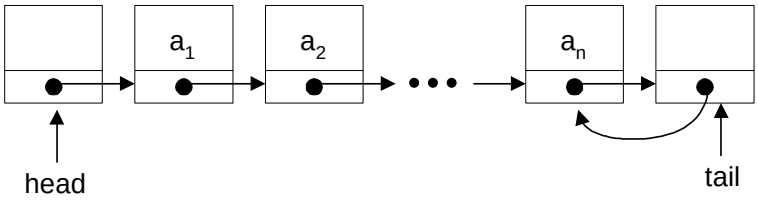


Abbildung 1.6: Dummy-Elemente am Anfang und Ende einer einfach verketteten Liste.

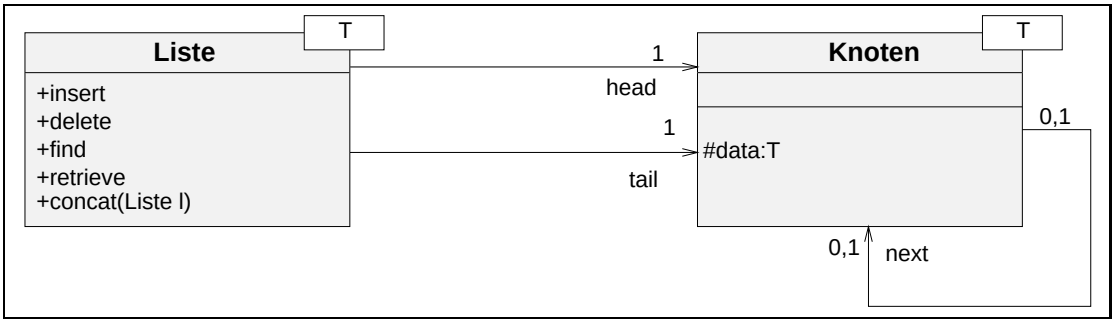


Abbildung 1.7: Einfache verkettete Liste mit head und tail

- Die Liste ist durch head- und tail-pointer gegeben
- next-Zeiger des tails zeigt auf das letzte Element. Dies erleichtert Konkatenation.
- In Abb. 1.8 ist die leere Liste abgebildet

Die Initialisierungsroutine für diese Liste sieht folgendermaßen aus:

```
public List2()
{
head = new Node();
tail = new Node();
head.next = tail;
tail.next = head;
}
```

Suchen durch Stoppertechnik: Gesuchtes Element wird vor Beginn der Suche in das Dummy-Element (`tail`) am Listenende geschrieben. Als Ergebnis wird eine Referenz auf den Knoten als Position zurückgeliefert,

```
public Knoten find(T o) {
   tail.daten = o;
   Knoten pos = head;
```

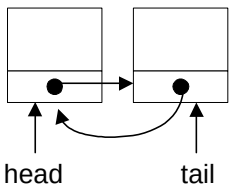


Abbildung 1.8: Leere Liste mit je einem Dummy am Anfang und am Ende.

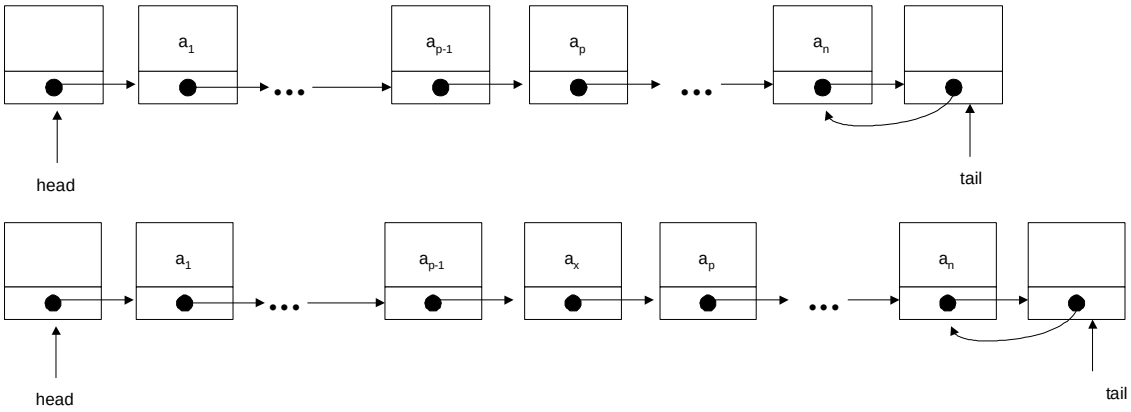


Abbildung 1.9: Einfügen eines Elements in eine einfach verkettete Liste.

```
    do pos = pos.next;
    while(pos.daten != o);
    return pos;
}
```

wenn der Wert gefunden wurde, anderenfalls eine Referenz auf `tail`. Das hat den Vorteil, dass man ggf. nun leicht am Ende einfügen kann.

Einfügen eines Elements an der Position p:

Element x soll an Position p eingefügt werden. Zeigt p tatsächlich auf das Element a_p, so ergibt sich das Problem, die Position $p-1$ zu finden, für dessen Element den next-Zeiger verändern muß. Komplexität für die Suche ist $\mathcal{O}(N)$.

Antizipative Indizierung: Das Element a_p wird indiziert durch den next-Zeiger des Elements a_{p-1}. Um an a_1 zu kommen wird also `head.next` verwendet, um an a_2 zu kommen, wird a_1`.next.` verwendet etc. Wenn im Funktionsaufruf p übergeben wird, muß intern mit `p->next` gearbeitet werden. Daher auch das Dummy-Element am Anfang. Wir verwenden jetzt also keine ganzen Zahlen mehr als Positionen, sondern Referenzen auf Knoten bzw. in Sprachen wir C++ Pointer auf Knoten.

```
public void insert(Knoten p,T t) {
    Knoten q = new Knoten(p);
    q.daten = t;
    q.next =p.next;
    p.next = q;
    if(q.next==tail)
        tail.next=q;
}
```

Dadurch wird das Einfügen auf eine Komplexität $\mathcal{O}(1)$ reduziert.

1.2.4 Implementierung 3: doppelt verkettete Liste

Wenn man für Listenelemente oft den Vorgänger und Nachfolger (predecessor, successor) benötigt, so empfiehlt es sich, die Liste doppelt zu verketten.

Für das Löschen von Listenelementen muß man sich Gedanken machen, was passieren soll. Bei Java mag man dem Garbage Collector vertrauen, bei C++ hat man mehr Wahlmöglichkeiten:

- gesamten Knoten einschließlich Nutzerdaten vom Heap mit delete entfernen
- Knoten aus Liste aushängen; Nutzerdaten auf Null setzen; Speicherblock in Freiliste einhängen

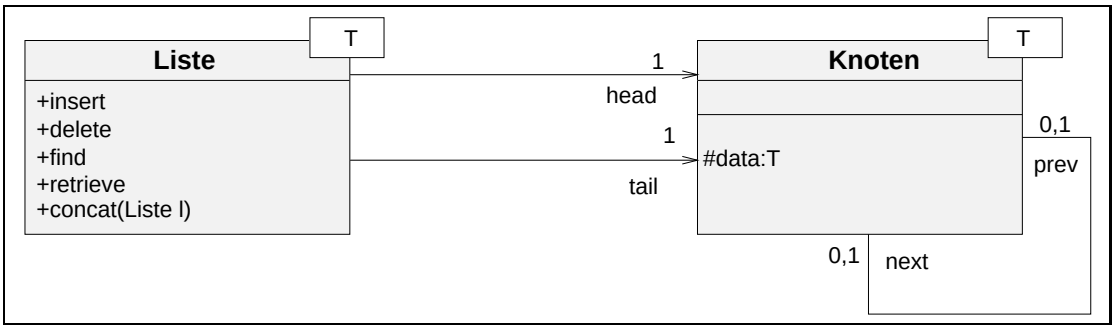


Abbildung 1.10: Doppelt verkettete Liste

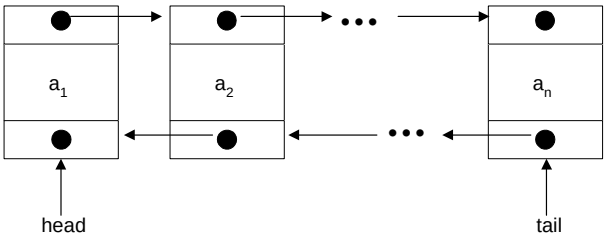


Abbildung 1.11: Doppelt verkettete Liste.

- Knoten aus Liste aushängen; Nutzerdaten nicht löschen, da der Knoten noch in einer anderen (Master-)Liste eingehängt ist

Realisierung mit Zeigern: Die Nutzerdaten sollten nicht in den Knoten direkt gespeichert werden, sondern im Knoten nur referenziert werden.

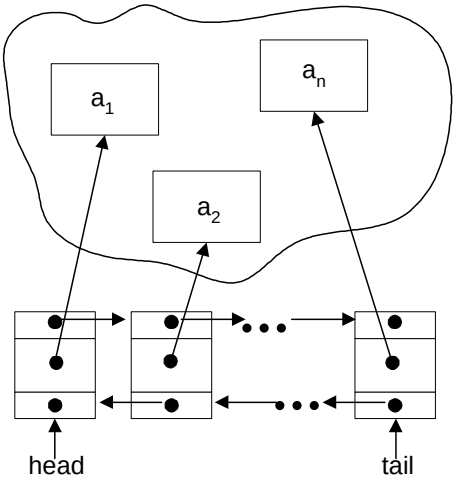


Abbildung 1.12: Die Nutzerdaten werden möglichst in den Knoten nur referenziert.

Es ergibt sich für die einzelnen Zugriffsmethoden und unterschiedlichen Implementierungen die Effizienz entsprechend der folgenden Tabelle. Bei der Variante 3 sei dabei antizipative Indizierung unterstellt.

	0	1	2	3
insert	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
delete	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
find	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
concat	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

1.3 Stapel und Schlangen (Stack and Queue)

Oft genügt es, Elemente nur am Anfang oder Ende einer Liste einzufügen, zu lesen oder zu entfernen.

- pushhead (list L , elem x): fügt x am Anfang von L ein
- top (list L): liefert das erste Element von Liste L
- pushtail (list L , elem x): fügt x am Ende von L ein
- bottom (list L): liefert letztes Element von Liste L
- pophead (list L, elem x): entfernt erstes Element und weist es x zu
- poptail (list L, elem x) : entfernt letztes Element und weist es x zu

Man erhält folgende Eigenschaften für Datentypen mit diesen Operationen:

- kontrollierte Zugriffspunkte
- einfache Implementierung mit konstantem Aufwand
- Spezialfälle: Stapel (Stack) und Schlange (Queue)
- Charakterisierung für Stack: LIFO (Last In First Out)
- Charakterisierung für Schlange: FIFO (First In First Out)

Für Stack und Queue gibt es eine Fülle von Anwendungen. Hier eine Auswahl:

Stapel (Stack)	Schlange (Queue)
Auswertung wohlgeformter Klammerausdrücke	Warteschlangen (Kunden vor Kassen, Druckaufträge, Befehlsfolgen)
Realisierung von Unterprogramm-aufrufen	Message-Passing zwischen Threads oder Prozessen
Auflösung rekursiver Funktionen in iterative	Vorrangwarteschlangen (priority queues)
Tiefensuche in Graphen	Breitensuche in Graphen

1.3.1 Stack (Stapel)

Bei einem Stack wird nur am Ende eingefügt und auch nur am Ende entnommen (Li-Fo, last in, first out). Dies kann man als Spezialfall einer Liste ansehen. Ein Stack enthält also nur die Zugriffsfunktionen am Ende der Liste: pushtail und poptail. Meist sagt man nur push und pop. Oft wird ein Stack als Array implementiert. Da in der Mitte des Stacks nicht eingefügt werden kann, entfällt das Verschieben der Elemente.

ADT Stack

create:	$\emptyset$	$\rightarrow$	stack
push:	stack $\times$ elem	$\rightarrow$	stack
pop:	stack	$\rightarrow$	stack
top:	stack	$\rightarrow$	elem
isEmpty:	stack	$\rightarrow$	bool

push:	stack × elem → stack; s.push(b)
pre:	keine
post:	ist $s = (a_1, \ldots, a_n)$, so bewirkt s.push(b), dass $s = (a_1, \ldots, a_n, b)$
	ist $s = (\ )$ also leer, so bewirkt s.push(b), dass $s = (b)$

Aufgabe 1.22
Beschreiben Sie die Operationen

pop:	stack	→	stack
top:	stack	→	elem
isEmpty:	stack	→	bool

◀

Einen Stack verwendet man zum Beispiel, um vollständig geklammerte Ausdrücke auszuwerten:

$$((6 * (4 * 28)) + (9 - ((12/4) * 2)))$$

Man kann diesen Ausdruck auswerten, indem man ihn von links nach rechts durchgeht, und bei jeder geöffneten Klammer einen neuen Stack beginnt. Bei einer geschlossenen Klammer wertet man den Ausdruck auf dem Stack aus und überträgt das Ergebnis auf den vorherigen Stack. Hier ist die Stackbelegung **mitten** in dieser Rechnung:

```
Stack 1 ( 672         +
Stack 2 (6  * 112   )  ( 9 -
Stack 3     (4*28)          (   3 *
Stack 4                      ( 12/4 )
```

Eine andere Implementierung wäre wie folgt: zwei Stacks: Operandenstack und Operatorenstack

- öffnende Klammer: ignorieren
- Operand: auf Operandenstack legen
- Operator: auf Operatorenstack legen
- schließende Klammer: ignorieren
- obersten Operator vom Operatorenstack nehmen
- zwei Operanden vom Operandenstack nehmen
- Ausdruck auswerten
- Ergebnis auf Operandenstack schreiben

Aufgabe 1.23
Wenden Sie diese Regeln auf den unten stehenden Ausdruck an. Notieren Sie sich skizzenhaft die einzelnen Schritte mit den entsprechenden Belegungen der Stacks.

$$((6 * (4 * 28)) + (9 - ((12/4) * 2)))$$

◀

Hier ist eine mögliche Teil-Implementierung für einen Stack.

```
public class Stack<T> {
   public Stack(int sz){
      stackSize = sz;
      T[] ts = (T [])(new Object[sz]);//See Bug ID:     5098163
      a = ts;
   }
   public void push(T e)throws Exception{
      if ( top == stackSize - 1)
         {throw new Exception("Stacksize exceeded");}
      else a[top++] = e;
   }
      ...
}
```

Aufgabe 1.24
Schreiben Sie die pop-Funktion für einen Stack. Achten Sie auf Sonderfälle! ◀

1.3.2 Schlangen (Queues)

Eine Queue entspricht der Schlange, die Sie aus dem Supermarkt, der Mensa etc. kennen werden. Elemente werden am Ende eingefügt und am Anfang entnommen. Eine Queue enthält also nur die Zugriffsfunktionen um am Ende einzufügen und am Anfang zu entnehmen. Im Gegensatz zum *Stack* handelt es sich bei der *Queue* um einen *Fi-Fo*, First in, First out Mechanismus.

ADT Queue						
front	:	queue			→	elem
enqueue	:	queue	×	elem	→	queue
dequeue	:	queue			→	queue
isEmpty	:	queue			→	bool

front: Liefert das Element head zurück ohne es aus der Queue zu löschen (nicht destruktiv)

enqueue: Fügt neues Element hinten ein (rear)

dequeue: Löscht das Element head

isEmpty: **true** falls die Queue leer ist , **false** sonst.

Man kann also am hinteren Ende einfügen und am vorderen auslesen.

Wie schon beim Stack angemerkt, gibt es auch bei der Queue mehrere sinnvolle Möglichkeiten, die Operationen zu definieren. So könnte man dequeue auch so defínieren, dass das Element zurück geliefert und gelöscht wird, also *front* und *dequeue* zusammenfassen.

```
class Queue<T>
{
private int tail = 0;
private int head = 0;
private bool contrary = false;
private int queueSize=0;
private T [] a;
public Queue(int sz)
    {
        queueSize = sz;
        a = (T[]) (new Object[sz+1]);
        tail = queueSize;
        head = tail;
```

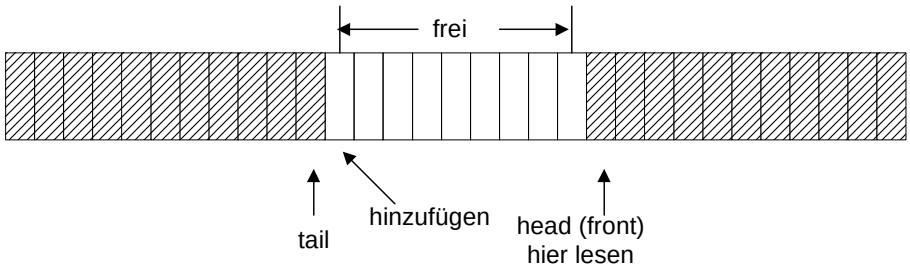
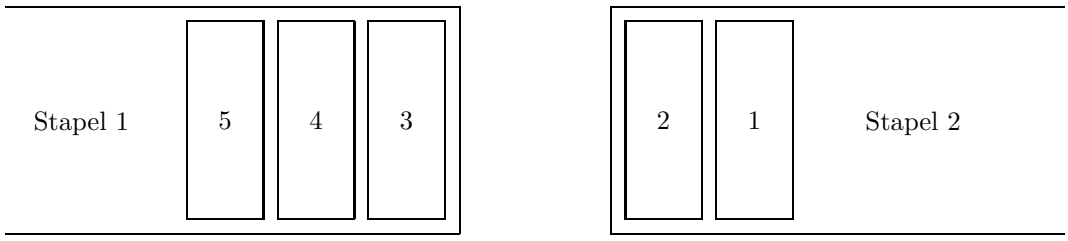


Abbildung 1.13: Queue als Ringbuffer

```
    }
public enqueue(Elem e) {
        if ((!contrary && ((head - tail) == queueSize)) ||
            ( contrary && ((head - tail) == 0))           )
           {throw error;}
        else if (tail == 0)) {
                contrary = !contrary;
                tail = queueSize; };
        a[tail] = e;
        tail--;}
             }
  }
public dequeue( ) {
        if (head == tail) {throw error;}
        else if (head == 1) {
                contrary = !contrary;
                head = queueSize+1; };
        head--;
        a[head] = null;
  }
}
```

Eine Queue wird gewöhnlich als Ringbuffer implementiert. Dazu wird ein Array allokiert, in dem ein head- und ein tail-Pointer die Queue-Elemente eingrenzen.

Den Effekt der Datenabstraktion wird hier am Beispiel der Schlange aufgezeigt, indem die Schlange durch zwei Stapel realisiert wird. In der nachfolgenden Abbildung ist das Prinzip veranschaulicht.



Das Einfügen der Elemente wird durch das Einfügen in `Stapel 1` vorgenommen, dass Auslesen durch das Auslesen aus `Stapel 2`. Wenn `Stapel 2` leer ist, in `Stapel 1` sich aber Elemente befinden, werden diese komplett in `Stapel 2` „umgestapelt“, also jeweils ein Element aus `Stapel 1` ausgelesen und sofort in `Stapel 2` eingelagert, bis `Stapel 1` komplett leer ist! Optisch, wie in der Zeichnung angedeutet, verändert sich die Reihenfolge der Elemente nicht. Sie sind jedoch statt in `Stapel 1` nun in `Stapel 2` enthalten.

Welchen Vorteil hätte man durch diese Realisierung ? Die Verwaltung des Speichers würde auf den Zeitpunkt konzentriert, indem „umgestapelt“ wird. In der Rest der Zeit wäre der Zugriff auf die Schlange genauso effizient, wie der auf einen Stapel! Wenn man statt einem tatsächlichen „umstapeln“ die Stapel geschickt tauscht (Anfang und Ende des nicht leeren Stapels müssen in der zugrunde liegenden Liste getauscht werden), dann kann man sogar diesen Aufwand noch reduzieren.

Umgekehrt geht es auch: mittels zwei Schlangen kann die Funktionalität eines Stapels realisiert werden: Dazu müsste man die Reihenfolge der gespeicherten Elemente „umdrehen“, damit das älteste Element einer Schlange zum jüngsten Element einer anderen Schlange wird. Da Schlangen, also FIFO-Speicher, aber die Reihenfolge der gespeicherten Elemente in der Ausgabe beibehalten, besteht zunächst keine Möglichkeit diese „umzudrehen“. Wenn man nun eine der beiden Schlangen stets leer hält und dort das neuste Element einträgt, leert man anschliessend die andere Schlange und fügt die Elemente in die erste Schlange ein. Damit wird das neuste Element stets das erste eingefügte Element in dieser Schlange sein und somit nach außen als zuletzt eingefügtes Element wieder als erstes ausgelesen werden.

1.4 Spezielle Techniken

1.4.1 Generische Typen

Generische Typen sind Datentypen die Typen als Parameter enthalten. In Java wird eine generische Liste so definiert:

```
List<T>
```

Dabei ist T der Typparameter. Setzt man für T z. B. `Integer` ein, dann bekommt man eine Liste von Integern. Ich werde versuchen in dieser Veranstaltung konsequent generische Klassen zu verwenden. In C++ lautet die Syntax

```
template<class T>
class List
```

Eine Objekt *list* von Liste mit Integern als Elemente würde dann so erzeugt

```
List<int> list;
```

Java verwendet dynamische Bindung um generische Typen zu unterstützen. Das heißt, der Methodenaufruf wird zur Laufzeit anhand des tatsächlichen Typen aufgelöst. In C++ wird der Typparameter (Template) zur Compilezeit ersetzt und es werden konkrete Typen erzeugt.

Generische Typen sollten Java-Programmierer kennen und wo es sinnvoll ist einsetzen.

Bemerkung 1.25 (Java Generics und Arrays)
Java ermöglicht es seit 1.5 generische Typen zu verwenden. Allerdings kann man kein Array eines generischen Typs erstellen:

```
liste = new T[maxListSize];
```

sondern muss zu dem Cast

```
liste = (T[])new Object[maxListSize];
```

greifen, was zu der Warnung

```
Type safety: The cast from Object[] to T[] is actually checking
against the erased type Object[]
```

führt. Dies wird als Bug 5098163 geführt. Die Behebung dieser Schwäche erfolgt eventuell in Java 7 („Under discussion as a potential feature in Java SE 7“.) ◄

Bemerkung 1.26 (Listen in Java)
Java bietet verschiedenen Implementierungen für Listen, z. B. ArrayList und LinkedList. Die Struktur von LinkedList zeigt Abb. 1.14 ◄

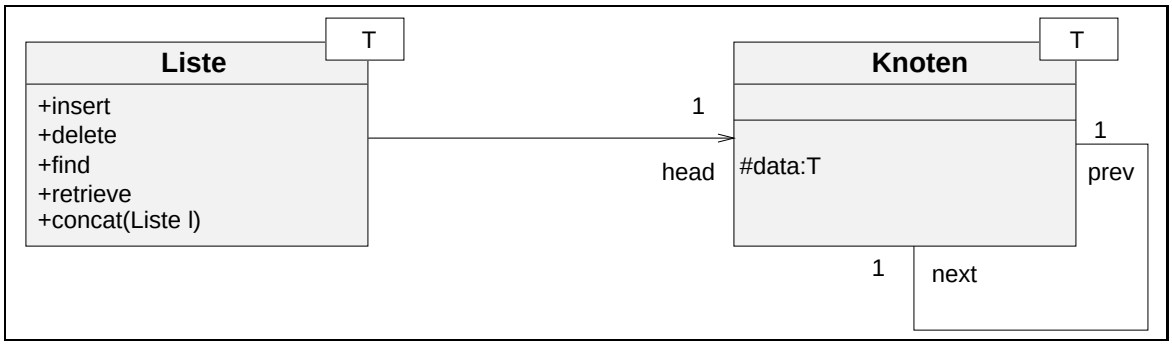


Abbildung 1.14: LinkedList in Java

1.4.2 Intrusive Datenstrukturen

Hier handelt es sich um eine Implementierungsstrategie, die besonders für Echtzeitsysteme und Systeme mit wenig Speicher angewendet wird. Am Beispiel einer Liste soll kurz das Prinzip erläutert werden: zunächst ist ein übliches Vorgehen für die Implementierung einer Liste, einen Container zu schaffen, welcher Referenzen (oder Pointer) auf die zu verkettenden Objekte enthält. Die Container selbst sind miteinander verlinkt. Dies nennt man nicht-intrusive Liste. Abbildung 1.15 illustriert das Prinzip: Die Listen-Container enthalten Referenzen oder Pointer auf die zu verkettenden Objekte. In UML kann man dies wie ein Abb. 1.16 darstellen: Die next Pointer sind

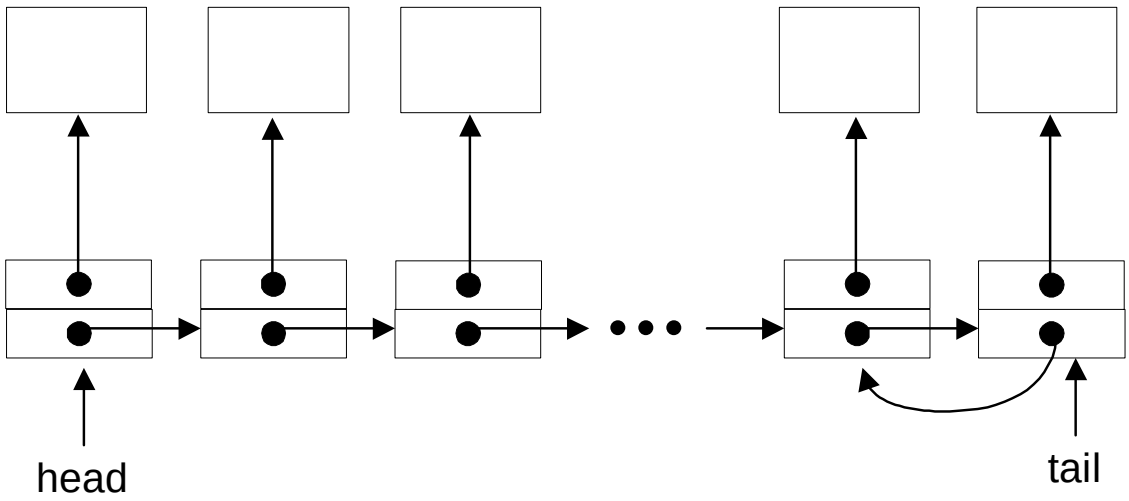


Abbildung 1.15: Nicht-intrusive Implementierung einer verketteten Liste.

Bestandteil der Objekte, welche verkettetet werden sollen. Eine intrusive Liste zeichnet sich im Unterschied dadurch aus, dass die zu verkettenden Objekte selbst ihre `next`-Pointer (oder Referenzen) enthalten. In der konkreten Implementierung kann dies auf verschiedenen Wegen erreicht werden. Typisch ist hier auch die Anwendung von Templates in C++. Die intrusive-Technologie besitzt entscheidene Vorteile gegenüber nicht-intrusiven Datenstrukturen.

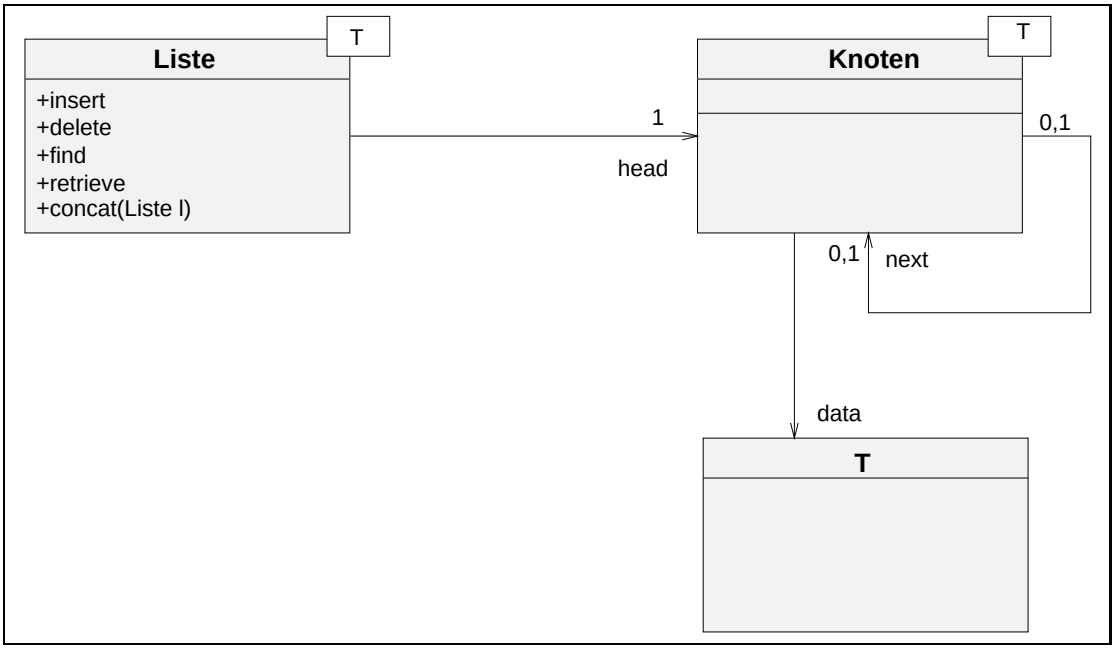


Abbildung 1.16: Nicht-intrusive Liste, UML

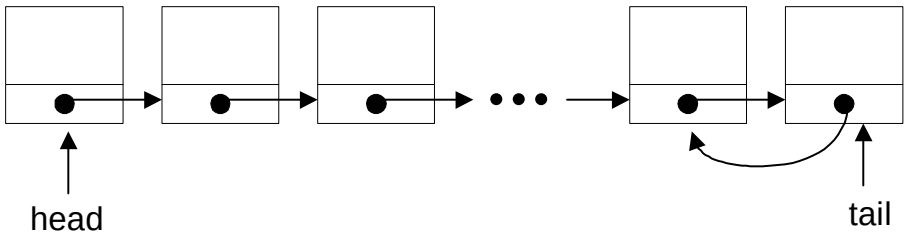


Abbildung 1.17: Intrusive Implementierung einer verketteten Liste.

Aufgabe 1.27 (Intrusiv UML)
Stellen Sie eine intrusive Implemntierung einer Liste analog zu Abb. 1.16 dar! ◄

intrusiv	nicht-intrusiv
weniger Speicherbedarf. Gerade bei Verkettung von kleinen Objekt besonders wirkungsvoll	jeder Container enthält einen 4 Byte großen Pointer auf das Objekt
direkter schneller Zugriff auf Objekte	zusätzlich Dereferenzierungsebene
weniger Objekte, dadurch geringere Beanspruchung der internen Speicherverwaltung	bei Vorallokierung der Container kann konstante Allokationszeit nicht garantiert werden, außer durch zusätzliche Freiliste → mehr Speicher
heterogene Listen schwierig	heterogene Listen möglich
Objekte können nur in einer Liste hängen	Objekte können in mehreren Listen gleichzeitig hängen

Als Beispiel für den letzten Punkt sei eine Struktur genannt, welche in einer Liste alle Studenten abspeichert, die eine bestimmte Vorlesung besuchen. Dies werde nun für alle Vorlesungen gemacht. Dadurch sind natürlich viele Studenten in mehreren Listen.

Intrusive Datenstrukturen kann man auf unterschiedliche Art und Weise implementieren. Ihre Eigenschaften und Auswirkungen sind vorsichtig abzuwägen gegenüber anderen Implementierungen. Allgemein kann man sagen, dass sie sich für Systeme mit Echtzeitanforderungen und wenig Speicher besonders eignen.

Literatur zu Intrusive-Technologien findet man auf der Homepage von Code Farms(www.codefarms.com).

1.4.3 Speicherverwaltung

Oft wird die Speicherverwaltung in echtzeitkritischen eingebetteten System den Anforderungen entsprechend selbst geschrieben. Typisch sind vor allem Static Allocation und Pool Allocation.

static allocation	pool allocation
Speicherplatz von Anfang an fest zugewiesen	Pool von Memoryblöcken vorallokiert
Reihenfolge der Initialisierung durch Allokator	Verwaltung der Memoryblöcke durch Freiliste
für Treiber, Konfigurationsdaten, nichtdynamischer Speicher	für Datensätze gleicher Größe, Messages
besonders sicher (keine Programmierfehler)	besonders schnell ($O(1)$)

Eine hocheffiziente Speicherverwaltung in C++, die eine einfach verkettete Liste in einer raffinierten Form verwendet, wird in [Ale03] beschrieben.

1.4.4 Message Queue

Hier noch ein Beispiel für eine Queue. Um Nachrichten (Messages) zwischen unterschiedlichen Kontexten (Threads, Prozesse) zu puffern, wird oft eine Queue verwendet. Die Message-Queue muß Unterlauf und Überlauf behandeln. Dies geschieht mit Zählsemaphoren. Außerdem muß der Schreib- und Lesevorgang geschützt werden. Dies geschieht mit Mutexen. In der Vorlesung Betriebssysteme wird dies ausführlich erläutert. Ist die Message-Queue leer, so blockiert ein Thread (Zählsemaphore für Unterlauf). Wie man eine Message-Queue dennoch annehmbar beenden kann, wird auf einigen Folien gezeigt. Außerdem zeige ich eine Alternative zum Message-Passing welche den lesenden und schreibenden Thread nicht blockiert. Literatur hierzu:

- Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects, Douglas C. Schmidt ([SSRB00])

- Consensus-based lock-free asynchronous three-buffer Communication, Reto Carrara, siehe http://www.carrara.ch/ fachbeitraege/ Consensus.pdf, zu letzt besucht am 15.03.2008.
- J. Chen, A. Burns, Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Concensus, ecrts, p. 2, 10th Euromicro Workshop on Real Time Systems, 1998

1.5 Fazit aus Sicht der Konstruktionslehre

Das wesentliche Konstruktionsprinzip hier ist die **Abstraktion** (für Daten wie auch für Funktionen). Abstraktion ist ein zentraler Begriff der konstruktiven Systemtheorie: Sie hilft, die komplexen Systeme überschaubar und damit handhabbar zu halten. Sicherlich beschreibt Abstraktion das Verhältnis zwischen Objekt und Instanz, kann aber nicht darauf reduziert werden. In der Informatik bestehen die Systeme teilweise aus Millionen von Einzelteilen, wenn man eine Zeile Programmcode einer herkömmlichen höheren Programmiersprache als ein Element betrachtet. Diese Systeme lassen sich nur durch Abstraktion handhaben, von der physikalischen Ebene bis hin zu der Applikationsebene. Von daher hat sich theoretisch betrachtet seit dem Modell der Turingmaschine bzgl. Berechenbarkeit in der Informatik nichts verändert. Die unterschiedlichen Abstraktionen, von Registermodellen über Objektorientiertheit und Web-Diensten bis hin zu Agentensystemen, erlauben aber erst die Konstruktion und Realisierung komplexer Systeme, wie wir sie heute kennen und immer mehr benötigen.

Zum Abschluss eine allgemeine, philosophische Definition zu diesem Thema:

> Abstraktion (Abziehung, Absonderung) ist die Heraushebung eines Erkenntnisinhalts durch die willkürliche, aktive Aufmerksamkeit (Apperzeption), das willkürliche, absichtliche, zweckbewußte Festhalten bestimmter Vorstellungsmerkmale unter gleichzeitiger Vernachlässigung, Zurückdrängung, Hemmung; anderer Merkmale. Der Gegensatz zur Abstraktion im engeren Sinne, d.h. zur Erweiterung des Begriffsinhalts, ist die logische Determination.

1.6 Aufgaben

1. ([03])Welche wichtigen Unterschiede gibt es zwischen $\mathbb{Z}$ und dem Datentyp `int` in einer Programmiersprache wie C++ oder Java?
2. ([04]) Welche Unterschiede gibt es zwischen $\mathbb{R}$ und den Datentypen `float` bzw. `double` in Java bzw. C++?
3. ([03])Schreiben Sie die Signaturen für folgende Operationen des Typs int auf: $++$ $*=$ $\leq$ $\%$
4. ([03]) Wie werden Vor- und Nachbedingungen in den folgenden Sprachen geschrieben?
   4.1. UML
   4.2. C++
   4.3. Java

   Finden Sie heraus, wie dies in anderen Sprachen gemacht wird!
5. ([04]) Beschreiben Sie die Operationen push, pop, top und isEmpty eines Stacks durch Angaben von Signatur, Vor- und Nachbedingung!
6. ([04]) Beschreiben Sie die Operationen front, enqueue, dequeue, isEmpty einer Queue durch Angaben von Signatur, Vor- und Nachbedingung!
7. ([03]) Schreiben Sie Beispiel 1.17 so um, dass die Vorbedingung entfallen kann!

8. ([25]) Implementieren Sie eine Liste mittels eines Arrays (Implementierung 0) und als doppelt verkettete Liste in Java! Vergleichen Sie die Performance Ihrer Implementierung mit der in den Java Klassenbibliotheken!

9. ([15]) Der ADT *deque* ist eine Liste, bei der Einfügungen und Entfernungen am Anfang oder Ende der Liste vorgenommen werden, ebenso wie viele der Zugriffe. Die fundamentalen Operationen fügen am Anfang bzw. Ende ein und entnehmen entsprechend: insertFront, insertRear, deleteFront, deleteRear. Ferner gibt es die lesenden Operationen getFront, getRear. Geben Sie an, wie dieser ADT mit einem Array und wie er mit einer Liste implementiert werden kann. (Vorzugsweise Java, C++, andere Sprachen können akzeptabel sein, im Zweifel: fragen!).

10. Fragen zum Begriff des Algorithmus:

    10.1. Beschreiben Sie den Begriff Algorithmus und erklären Sie die vier allgemeinen Anforderungen an einen solchen Algorithmus!

    10.2. Erklären Sie folgende vier Eigenschaften eines Algorithmus: *terminiert, determiniert, deterministisch und nicht deterministisch*!

11. Beschreiben Sie das Prinzip bzw. die Grundidee der Datenabstraktion!

12. Welche vier Typen von Funktionen benötigt man im Allgemeinen? (unabhängig von der konkreten Datenstruktur als „Schnittstelle“) Geben Sie für einen Stapel jeweils ein erklärendes kleines Beispiel an. Es genügt der Name einer Funktion/Methode und eine kurze Erklärung, was sie macht.

13. Definieren Sie durch Angabe der formalen Beschreibung einen Datentyp `Stack`, der einen Stapel nach dem Last-In-/First-Out-Prinzip (LiFo) modelliert. Geben Sie dazu die Wertebereiche (Typen) und die Operationen an. Bei den Operationen ist die allgemeine/formale Typisierung anzugeben (z.B. durch Kreuzprodukt) sowie die Pre- und Postcondition.

    Definieren Sie Operationen zum Ablegen eines Elementes (`push`), zum Entfernen des ersten Elementes (`pop`), zum Abfragen des ersten Elementes (`top`) sowie zur Bestimmung der Höhe des Stack (`high`).

14. Die Addition kann durch Verwendung der Nachfolger- (`succ`) und Vorgänger-(`pred`) Funktionen realisiert werden. Im Falle der Addition sind das die Funktionen `+1` und `-1`.

    Schreiben Sie in einer (Pseudo-)Programmiersprache Ihrer Wahl **zwei Versionen** der Addition: eine Version, die einen **rekursiven Ablauf** beschreibt, und eine Version, die einen **iterativen Ablauf** beschreibt. Beide Versionen sollen jeweils nur genau **zwei ganze Zahlen** addieren können und dies ausschließlich mittels der beiden genannten Funktionen!

15. Berechnung von π:

    15.1. Implementieren Sie folgendes Verfahren in z.B. Java. Dieses Verfahren zur Berechnung der Zahl π ist aus dem Jahr 1976 und stammt von den Herren Salamin und Brent.

$$a_m := \frac{a_{m-1} + b_{m-1}}{2} \text{ mit } a_0 := 1$$

$$b_j := \sqrt[2]{a_{j-1} * b_{j-1}} \text{ mit } b_0 := \sqrt[2]{0.5}$$

$$\pi_n := \frac{(a_n + b_n)^2}{(1 - \sum_{k=0}^{n} [2^k * (a_k - b_k)^2])}$$

    15.2. Zeigen Sie den Ablauf der rekursiven Aufrufe für π_2 auf. Eine Berechnung ist dabei nicht durchzuführen! Wie oft werden a_n und b_n dabei jeweils aufgerufen?

16. Gegeben sei nachfolgende Funktion:

$$f(n) = \begin{cases} 1 & \text{falls } \forall n \in Z : n \leq 1 \\ f(n-1) + f(n-2) & \text{falls } \forall n \in Z : n > 1 \end{cases}$$

    16.1. Implementieren Sie diese Funktion direkt, also ohne Optimierungsüberlegungen, in Java!

    16.2. Wie groß ist die Anzahl der Additionen $A(n)$ die beim Aufruf von $f(n)$ ausgeführt werden ? Berechnen Sie dazu zuerst die Anzahl der Additionen für alle $n \in \{1, 2, 3, 4, 5, 6, 7\}$. Versuchen Sie dann eine allgemeine Formel aufzustellen.

    16.3. Wie kann man die Funktion f **rekursiv** so implementieren, dass sie im Zeitaufwand linear läuft, also $O(n)$ besitzt ? Transformieren Sie dazu ggf. Laufzeitaufwand der naiven Lösung in Speicherplatzaufwand, jedoch mit maximal $O(n)$ Speicherplatzaufwand! Geben Sie dazu ein Java-Programm an und begründen kurz, warum der Aufwand jeweils (Laufzeit-/Speicherplatz) linear ist!

17. Geben Sie bitte die Vor- und Nachteile einer doppelt verketteten Liste im Vergleich mit einer einfach verketteten Liste an!

18. Welches sind die wichtigsten Funktionen des ADT *Queue*?

19. Geben Sie bitte für verschiedene Implementierungsmöglichkeiten einer Klasse *Knoten* für eine verkettete Liste die Vor- und Nachteile an! Begründen Sie Ihre Ansichten!

Kapitel 2

Algorithmen und Komplexität

2.1 Übersicht

Ziel dieses Kapitels ist es, Ihnen einige grundlegende Begriffe zu vermitteln, die bei der Analyse von Algorithmen nützlich sind. Dabei geht es um den Ressourcenverbrauch, z. B. die Anzahl Schleifendurchläufe, Vergleiche, Rechenoperationen, den Speicherbedarf, oder was immer als repräsentativ für den Aufwand sein mag, den ein Algorithmus benötigt. Wie sich das konkret auf die Laufzeit auf einem konkreten Rechner auswirkt hängt dann nur noch von den jeweiligen Randbedingungen ab.

2.2 Algorithmen

Ich beginne dieses Kapitel mit drei ganz elementaren Verfahren zur Ermittlung von Primzahlen, die Einige von Ihnen vielleicht noch aus der Schule erinnern. Dabei geht es darum einige Grundprinzipien an ganz einfachen Beispielen zu erläutern.

Beispiel 2.1 (Primzahltest mit der Divisionsmethode)
Bestimmung aller Primzahlen kleiner $n \in \mathbb{N}$.

1. beginne bei $i = 2$
2. zu prüfen ist, ob i eine Primzahl ist. Beginne mit $j = 2$. Falls $j \neq i$ ist, dann teste, ob i durch j teilbar ist. Wenn ja, dann streiche dieses i.
3. Fahre fort mit dem nächsten $j = j + 1$ (möglicher Teiler) solange $j \leq N$.
4. Fahre fort mit dem nächsten $i = i + 1$ (mögliche Primzahl) solange $i \leq N$.
5. alle nicht-gestrichenen Zahlen größer als 1 sind die gesuchten Primzahlen

◀

Für diesen Algorithmus gibt es offensichtlich mehrere Verbesserungsmöglichkeiten. Dennoch wollen wir zunächst diese einfache Implementierung ansehen.

```
public static void primzahlen(int n)
{
    boolean a[] = new boolean[n];
    int i, j;
    for ( i = 0; i < n ; i++)
          a[i] = true;
    for ( i = 2; i < n; i++)
```

```
        {
            for (j = 2; j < n; j++)
                 if ( (i%j == 0) && (j!=i) ) a[i] = false;
        }
    }// alle i für die a[i] noch auf true steht, sind Primzahlen
```

Beispiel 2.1 ist ein „brute-force“ Algorithmus: Es werden einfach alle Möglichkeiten berechnet. Einige elementare Verbesserungen kann man daran vornehmen, aber das ändert nicht viel am Grundprinzip. Diese Art von Algorithmen ist *eine* häufig vorkommende. Ein anderes Extrem zeigt die Formel zur Berechnung der ersten n positiven ganzen Zahlen (Bsp. 2.28). Hier existiert ein Algorithmus, der es ermöglicht das Ergebnis ohne alle einzelnen Additionen anzugeben. Eine weitere Art von Algorithmen sind *heuristische* Algorithmen, die zwar nicht beweisbar oder sicher ein optimales Ergebnis liefern, nach aller Erfahrung aber ein gutes Ergebnis.

Aufgabe 2.2
Wie oft wird hier die Division mit % ausgeführt? ◀

Antwort: Die innere Schleife wird $n-2$ mal durchlaufen. Die äußere Schleife wird auch $n-2$ mal durchlaufen. Insgesamt erhält man $(n-2)*(n-2) = n^2-4n+4$. Betrachtet man nun sehr große n, so macht sich besonders das n^2 bemerkbar. Wir reden hier von quadratischer Komplexität oder quadratischem Aufwand.

An diesem Verfahren sind einige Verbesserungen leicht möglich:

- Es genügt, die innere Schleife abzubrechen, wenn bereits einmal eine Division geklappt hat.
- Die innere Schleife muß nur bis i laufen, denn i ist ja sicher nicht durch Zahlen größer als es selbst teilbar.
- Die innere Schleife muß sogar nur bis $\sqrt{i}$ laufen. Warum?
- Die äußere Schleife über i muß nur über alle ungerade i laufen. Es könnte also heißen: `for ( int i = 3; i < n ; i+=2)`. In dem Fall muss dann natürlich noch `a[i+1] = false;` im Code eingefügt werden.
- Die äußere Schleife über i kann auch schon diejenigen i ausfiltern, welche schon als Nicht-Primzahlen erkannt worden sind. z.B. mit `if ( a[i] ) ...` . Warum?
- Bei diesem Algorithmus werden Zeugen gesucht (die j's im Algorithmus), die bestätigen können, ob der aktuelle Kandidat (das aktuelle i im Algorithmus) keine Primzahl ist. Eine Verbesserung wäre die Strategie zu ändern: ein gefundener Zeuge (wegen der Primfaktorzerlegung also unsere sicher gefundnen Primzahlen) wird komplett ausgefragt über alle (bis n) Nicht-Primzahlen, die er kennt: siehe Sieb des Eratosthenes).

Ein Versuch auf einem Laptop ergab sich für den nicht-verbesserten Algorithmus folgende Laufzeiten (interner Timer auf Sekunde genau):

Problemgröße N	Sekunden
1E03	0
5E03	1
10E03	4
15E03	9
20E03	16
25E03	25
30E03	36

Hier kann man gut eine quadratische Abhängigkeit erkennen. (Woran erkennen Sie dies?)

Wir betrachten nun die beiden geschachtelten Schleifen mit einer der vorgeschlagenen Verbesserungen. Ich habe die Startwerte bei 1 anfangen lassen und die Schleifen bis einschließlich n bzw. i gemacht, um die nachfolgende Untersuchung einfacher zu machen.

```
for ( i = 1; i <= n; i++)
{
     for ( int j = 1; j <= i; j++)
          irgendetwas;
}
```

Für $i = 1$ wird die innere Schleife nur einmal durchlaufen. Für $i = 2$ zwei mal, für $i = 3$ dreimal und so weiter. Insgesamt ergibt das

$$1 + 2 + 3 + ... + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

Man hat also wieder eine Abhängigkeit von n^2. Diesmal ist aber ein Faktor $\frac{1}{2}$ davor. Der Algorithmus ist zwar besser, rutscht aber nicht in eine neue Klasse. Es bleibt bei quadratischem Aufwand. Eine Verbesserung in eine neue, etwas bessere Klasse gelingt uns erst mit dem folgenden Algorithmus:

Beispiel 2.3 (Sieb des Eratosthenes)
Zweck des Algorithmus: finde alle Primzahlen bis zu einer gegebenen Zahl $n \in \mathbb{N}$.

1. beginne bei $i = 2$
2. streiche alle Vielfachen von i bis zur Zahl n (außer i selbst)
3. erhöhe i um eins
4. ist $i \leq \sqrt{n}$ und ist i noch nicht gestrichen, so gehe zu 2
5. alle nicht-gestrichenen Zahlen größer als 1 sind die gesuchten Primzahlen

◀

So könnte das Programm dazu aussehen, wenn ich den Algorithmus in einer Klassen-Operation einer Klasse implementiere (`a` ist eine Klassenvariable):

```
public static void sieb(int n){
  a = new boolean[n];
  int i,j;
  for(i=0 ; i < n ; i++)
      a[i] = true;
      for(i=2 ; i < Math.sqrt(n) ; i++){
      if ( a[i] == true ){
          for (j = 2; i*j < n ; j++)
               a[i*j] = false;
          }
  }
}
```

Eine empirische Untersuchung auf einem Laptop ergab einmal folgende Zahlen:

Problemgröße N	Sekunden
1E03	0
1E04	0
1E05	0
1E06	1
10E06	3
20E06	5
30E06	8
40E06	10
50E06	14
100E06	29

Hier läuft die äußere Schleife nur bis $\sqrt{n}$. Die innere Schleife läuft bis $\frac{n}{i}$, dies allerdings nur, wenn nicht schon durch `if (a[i])` vorher herausgefiltert wurde. Durch dieses Herausfiltern wird eine genaue Komplexitätsabschätzung äußerst schwierig, aber man kann immerhin eine Abschätzung nach oben vornehmen. Wir untersuchen folgendes Programm:

```
for ( i = 1; i <= sqrt(n); i++)
{
      for ( int j = 1; j*i <= n; j++)
irgendetwas;
}
```

Die innere Schleife läuft immer nur bis $\frac{n}{i}$, wobei i durch die äußere Schleife von 1 bis $\sqrt{n}$ läuft. Für $i = 1$ durchläuft die innere Schleife n-mal, für $i = 2$ durchläuft sie $\frac{n}{2}$-mal, für $i = 3$ $\frac{N}{3}$-mal usw. Eigentlich dürften wir hier nur ganzzahlige Werte nehmen. Dies vernachlässigen wir aber der Einfachheit halber. Wir müssen also rechnen:

$$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + ... + \frac{n}{\sqrt{n}} = \sum_{i=1}^{\sqrt{n}} \frac{n}{i} = n \sum_{i=1}^{\sqrt{n}} \frac{1}{i}.$$

Wir haben es also mit einer harmonischen Reihe zu tun. Angenommen, wir wüssten, dass man die harmonische Reihe nach oben durch den Logarithmus abschätzen kann:

$$\ln k < \sum_{i=1}^{k} \frac{1}{i} < 1 + \ln k.^1$$

Dann erhalten wir

$$n \ln \sqrt{n} = \frac{1}{2} n \ln n.$$

Wir haben also für das Sieb des Eratosthenes ein Verhalten von $\frac{1}{2} n \ln n$. Man spricht vom Aufwand n log n, oder man sagt *superlinearer* Aufwand, weil er etwas schlechter ist als linear. Dies ist deutlich besser als der quadratische Aufwand im vorherigen Algorithmus: eine neue Klasse.

Zusammenfassung

- Beide Verfahren liefern das gleiche Ergebnis. Dennoch ist das letzte Verfahren in einer besseren Klasse einzuordnen. Im folgenden werden wir den Aufwand von Algorithmen noch genauer beschreiben.
- Das Sieb des Eratosthenes könnte man noch weiter verbessern, z.B. könnte die äußere Schleife nur alle ungeradzahlige i durchlaufen. Wir verbessern dadurch die Konstante in der Aufwandsabschätzung
- Eine Aufwandsabschätzung (Komplexitätsanalyse) kann sehr kompliziert bis unmöglich sein. Sehr oft kann man jedoch untere oder obere Grenzen für den Aufwand angeben, zum Beispiel wenn Schleifen in der besprochenen Form auftreten.
- Beide Algorithmen erfordern nicht unerheblich Speicherplatz. Gerade auf einem embedded System kann dies von großer Bedeutung sein. Man muß daher genau definieren, was vom Algorithmus gefordert ist. Ein Primzahltest mit einer einzelnen Zahl kann z.B. mit erheblich weniger Speicheranforderung implementiert werden.
- Oft kann man durch Verwendung von zusätzlichem Speicherplatz den Algorithmus beschleunigen. Sind die Speicherressourcen knapp, so kann man oft einen Algorithmus finden, der zwar weniger Platz benötigt, dafür aber langsamer ist.

[1] Das Integral $\int \frac{1}{x} = ln$ läßt sich von unten und oben durch eine Treppenfunktion approximieren, deren Stufenbreite 1 ist und damit eine Partialsumme der harmonischen Reihe darstellt.

- In der Kryptographie wird der Aufwand oft als eine Größe der Dezimalstellenanzahl der Eingangsgröße angegeben. Es wird also

$$m = \ln n,$$

die Anzahl der Binärstellen von n, als Problemgröße angesehen[2]. Wenn wir vorher den Aufwand hatten von $n \ln n$, so haben wir nun, indem wir n durch 2^m ersetzen

$$n \ln n = 2^m \ln 2^m = 2^m m = 2^{m+1}.$$

Wir haben also exponentiellen Aufwand, wenn die Problemgröße die Anzahl der Binärstellen ist.

Im Jahr 2002 wurde ein Verfahren zum Primzahltest gefunden, dass sich besser als bisher bekannte verhält: Das AKS-Verfahren von Manindra Agrawal, Neeraj Kayal und Nitin Saxena [AKS04].

Definition 2.4
Der Begriff *Algorithmus* wurde vermutlich von dem arabischen Mathematiker Al-Khowârizmî (ca. 790-840 n.Chr.) geprägt. Ein **Algorithmus** ist ein Verfahren zur Lösung eines Problems. Die Verarbeitungsvorschrift muß dabei so präzise formuliert sein, daß sie von einer Maschine (etwa einem Computer) oder auch von einem Menschen durchgeführt werden kann. Er hat folgende Eigenschaften:

- **Eindeutigkeit** Er ist eindeutig und unmissverständlich formuliert.
- **Endlichkeit** Er ist in endlich vielen Schritten beschrieben.

Außerdem können Algorithmen zusätzlich folgende Eigenschaften haben:

- Liefert der Algorithmus ein Resultat nach endlich vielen Schritten, so sagt man, der Algorithmus **terminiert**.
- Ein Algorithmus heißt **determiniert**, falls er bei gleichen Eingaben und Startbedingungen stets dasselbe Ergebnis liefert.
- Ein Algorithmus heißt **deterministisch**, wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht.

◀

Beispiel 2.5 (Algorithmen)
- Computerprogramme
- Kochrezepte
- Aufbauanleitung für IKEA-Möbel („Ikeabana“)
- Pseudocode zur Formulierung von Algorithmen
- Mathematik zur Formulierung von Algorithmen
- Umgangssprache zur Formulierung von Algorithmen

Es ist in manchen dieser Fälle strittig, ob die angegebenen Algorithmen determiniert, determininistisch, eindeutig, oder endlich sind. ◀

[2] Eigentlich $m = \log_2 n$. Da aber $\log_2 n = \frac{1}{ln2} lnn$ und unter Vernachlässigung konstanter Faktoren, schreibe ich der vereinfachten Rechnung wegen $m = \ln n$.

2.3 Komplexitätsanalyse

In diesem Abschnitt befassen wir uns mit algorithmischen Problemen, d.h. Problemen, die sich durch einen Algorithmus zur Lösung beschreiben lassen. Da ein Algorithmus nur „einmal“ entwickelt wird und dann „sehr häufig“ ausgeführt wird, ist der Ausführungsaufwand eines Algorithmus im Allgemeinen viel wichtiger als der Entwicklungsaufwand. Letztlich lässt sich aber durch intensivierung des Entwicklungsaufwands und damit einem höherem Zeitaufwand dafür, z.B. durch Entwicklung gleichwertiger, aber schnellerer Algorithmen, teilweise sehr viel Zeit bei der Ausführung einsparen. Von daher ist also zu unterscheiden zwischen

Entwicklungsaufwand und dem

Ausführungsaufwand auch als Komplexität bezeichnet.

Die Klasse aller algorithmischen Probleme enthält aber nicht nur lösbare Probleme, sondern kann selbst wieder unterteilt werden in

Klasse 1 : algorithmische Probleme, die aus theoretischen Gründen nicht lösbar sind

Klasse 2 : algorithmische Probleme, die zumindest theoretisch lösbar sind

Das Thema soll hier nicht weiter vertieft werden.

Theoretisch lösbar oder nicht impliziert die Frage, ob es diese Unterscheidung auch für die Praxis gibt: Ja, ist aber schwierig zu beantworten. In der Praxis ist es die Anwendung, die etwas als lösbar einstuft oder nicht lösbar einstuft, was manchmal nur an ein paar ms hängen kann. Dies lässt sich kaum allgemein formulieren, weshalb in der Komplexitätstheorie dieser Punkt unter dem Stichwort „effizient lösbar“ betrachtet wird. Bei der Betrachtung von Algorithmen werden wir daher ganz entscheidend mit der Frage ihrer Effizienz konfrontiert. Die algorithmische Komplexitätstheorie behandelt Methoden und Notationen, mit denen man den Aufwand eines Verfahrens formal beschreiben kann. Grundlegende Annahme der Komplexitätsanalyse ist

> Klassifikation von Problemen (generische Fragestellung mit verschiedenen Instanzen) anhand des Bedarfs an Berechnungsresourcen (Rechenzeit und Speicherbedarf) in Abhängigkeit von der Größe der Eingabe.

Als Analyse von Algorithmen bezeichnen wir die Untersuchung von Speicherplatzbedarf und Rechenzeit in Abhängigkeit von Anzahl und Art der Eingangsdaten mit dem Ziel der Effizienzverbesserung. **Effizient Lösbar** ist ein Problem, wenn ein Polynom existiert, dass die Anzahl der Rechenschritte auf einer deterministischen, sequentiellen Maschine in Abhängigkeit von der Instanzengröße nach oben abschätzt. Die Menge dieser Probleme wird mir **P** für polynomial lösbar bezeichnet.

Speicherplatzbedarf und benötigte Rechenzeit werden auch als **Komplexität** bezeichnet, wobei meistens der Schwerpunkt auf der Rechenzeit liegt. Der Aufwand für irgendein Verfahren wird (bis auf wenige angenehme Ausnahmen) von der Größe des Eingangsdatenbereichs, z.B. der Anzahl der zu sortierenden Elementen, abhängen.

Die Aufgabe der Algorithmenanalyse besteht darin, ohne ein aktuelles Programm, ohne Rechner und ohne Stoppuhr Aussagen über die zu erwartende Effizienz des Algorithmus zu machen. Das ist in vielen Fällen nicht ganz einfach, weil nichttriviale Algorithmen „je nach Lage der Dinge“ ganz unterschiedliches Laufzeitverhalten zeigen. So unterscheidet man normalerweise drei Fälle:

Der beste Fall (best case): Das ist diejenige Struktur der Eingabedaten, für die der Algorithmus am effektivsten ist. So ist z.B. für einige Sortierverfahren eine bereits sortierte Sequenz der beste Fall (Bubblesort), für andere eine genau umgekehrt sortierte Sequenz (Heapsort).

Weil beste Fälle ziemlich selten auftreten, werden wir uns um ihre Behandlung nicht sehr häufig kümmern.

Der schlechteste Fall (worst case): Diejenige Eingabestruktur, für die der Algorithmus am wenigsten effektiv ist. Sucht man z.B. ein Element in einem Array, so ist der schlechteste Fall immer der, dass das Element gar nicht drin ist.

In gewisser Weise ist dies der interessanteste Fall für uns.

Der mittlere Fall (average case): Die zu erwartende Effizienz bei beliebiger Strukturierung der Eingabedaten.

Dieser Fall ist in der Praxis ausgesprochen interessant, hat aber den gewichtigen Nachteil, dass seine Behandlung mathematisch ausgesprochen kompliziert ist: Man braucht dazu in erster Linie Verfahren der Wahrscheinlichkeitsrechnung. Wir werden uns nicht so oft mit diesem Problem befassen.

Natürlich haben z.B. Programmiersprache, Compiler, Betriebssystem und verwendeter Rechner einen Einfluss auf die effektive Laufzeit des fertigen Programms, aber mit einer einfachen Überlegung erkennt man doch, dass es u.U. andere Kriterien für die Effizienz eines Algorithmus gibt. Dazu betrachten wir zwei kleinere Beispiele.

Beispiel 2.6
Betrachten wir zwei Code-Segmente:

```
for i := 1 to n do          for i := 1 to n do
    for j := 1 to n do         for j := 1 to n do
        s := s + 1                 for k := 1 to n do
                                       s := s + 1
```

Nehmen wir an, das linke Fragment liefe auf einem uralten Sinclair Z80, der 10^3 Operationen „s := s + 1“ in der Sekunde ausführen kann und nehmen wir ferner an, das rechte auf einer CRAY XMP-4, die 10^7 davon in der Sekunde schafft. Kurzes Nachrechnen ergibt, dass für $n = 10^4$ beide Programme die gleiche Zeit brauchen und dass für größere n das Programm auf dem Sinclair schneller ist. Unter dem Gesichtspunkt des Verhaltens bei wachsender Größe der Eingabe ist der linke Algorithmus also effizienter als der rechte. Dieser Gesichtspunkt wird durch die so genannte asymptotische Analyse präzisiert. Man sucht dabei für eine gegebene Ressourcenfunktion nach Vergleichsfunktionen und vergleicht beide im Hinblick darauf, wie sie sich bei wachsendem Argument verhalten. ◄

Beispiel 2.7
Wir betrachten zwei Probleme:

1. Problem M_1: n^2-Rechenschritte bis Instanzgröße m (bei m also m^2)
2. Problem M_2: 2^n-Rechenschritte bis Instanzgröße m (bei m also 2^m)

Es wird nun eine Erneuerung durchgeführt und eine 10^6 schnellere Technologie verwendet. Die Auswirkung auf unsere Probleme:

1. Problem M_1: Nun $1000 * m$ Instanzen möglich! (da $(1000 * m)^2 = (10^3)^2 * m^2 = 10^6 * m^2$).
2. Problem M_2: Nun $20 + m$ Instanzen möglich! (da $2^{20+m} = 2^{20} * 2^m = (10^6 + 48576) * 2^m$)

◄

Die Güte eines Algorithmus, der als Computerprogramm implementiert ist, kann man nach verschiedenen Kriterien bestimmen, z. B. :

- Minimale Laufzeit
- Minimaler Hauptspeicherverbrauch
- Minimaler Festplattenspeicherverbrauch

- Minimale Netzbelastung
- Eine garantierte maximale Laufzeit (hartes Echtzeitproblem)
- Möglichst gut wartbar
- Möglichst universell einsetzbar
- Möglichst gut intuitiv verständlich

Um einen Algorithmus zu untersuchen, kann man

- theoretische Untersuchungen machen
- empirische Untersuchungen machen

Mit theoretischen Untersuchungen werden wir uns im folgenden beschäftigen. Man vergibt ein Qualitätskriterium, welches sich Aufwands- oder Resourcenfunktion nennt, und untersucht dieses.

Empirische Untersuchungen sind Tests und Experimente mit dem Algorithmus. Sie sind in der Softwareentwicklung unumgänglich. Mit ihnen kann man nicht nur Fehler finden, sondern auch seine theoretischen Untersuchungen bestätigen. Auch damit werden wir uns in der Vorlesung und im Praktikum beschäftigen.

Aufwandsfunktion

Definition 2.8 (Aufwandsfunktion)
Eine Aufwandsfunktion (Ressourcenfunktion) $T(n)$ ist eine Funktion der Problemgröße n, welche die Laufzeit eines Algorithmus angibt. ◀

Da die Laufzeit eines Algorithmus natürlich unterschiedlich sein wird auf unterschiedlichen Plattformen, zählt man oft die arithmetischen Operationen, Vergleiche, Zuweisungen, und Funktionsaufrufe. Das heißt, man verwendet die Anzahl dieser Operationen als Kennzahl für die Laufzeit des Algorithmus. Im obigen Beispiel vom Sieb des Eratosthenes haben wir einen Aufwand von ungefähr

$$T(n) \approx \frac{1}{2} n \, ln \, n.$$

Wir interessieren uns in der Regel für die Art des Wachstums der Aufwandsfunktion und vernachlässigen konstante Faktoren. Daher sprechen wir hier von einem Aufwand von $n \log n$, wobei die Basis des Logarithmus nicht angegeben ist. Logarithmen mit verschiedener Basis unterscheiden sich nur durch einen Faktor. Wir einigen uns auf die Basis 2 für alle Logarithmen.

Die wichtigsten Aufwandsfunktionen sind

konstant	1
logarithmisch	$\log n$
linear	n
$n \cdot \log n$	$n \log n$
quadratisch	n^2
kubisch	n^3
polynomiell	$n^4, n^5, n^6, \ldots$
exponentiell	$2^n, 3^n, \ldots$

Asymptotischer Aufwand

Die Vernachlässigung in der Aufwandsfunktion von konstanten Faktoren und Summanden hat einen Namen: asymptotischer Aufwand oder asymptotische Aufwandsordnung. Man beschreibt ihn mit der *Landau-Notation*.

Definition 2.9 ($\mathcal{O}$-Notation oder Groß-O-Notation)
Seien $f : \mathbb{N} \to \mathbb{R}^+$ und $g : \mathbb{N} \to \mathbb{R}^+$ positive reellwertige Funktionen auf $\mathbb{N}$. Wir definieren

$$f = \mathcal{O}(g) :\Longleftrightarrow \exists\, n_0 \in \mathbb{N}, c \in \mathbb{R}^+, c > 0 \text{ so dass } \forall\, n \geq n_0 \;:\; f(n) \leq c \cdot g(n).$$

Wir sagen auch f hat die Ordnung $\mathcal{O}(g)$. ◀

Dies bedeutet: f wächst höchstens so schnell wie g. Eine präzisere Schreibweise ist $f \in \mathcal{O}(g)$, wobei

$$\mathcal{O}(g) := \left\{ f : \mathbb{N} \to \mathbb{R}^+ \mid \exists\, n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0 \text{ so dass } \forall\, n \geq n_0 \;:\; f(n) \leq c \cdot g(n) \right\}.$$

Man kann daher die „Gleichung "$f = \mathcal{O}(g)$ nur von links nach rechts lesen.

Beispiel 2.10

1. $T(n) := n + 5 = \mathcal{O}(n)$. Um dies nachzuweisen, müssen wir Konstanten n_0 und $c > 0$ finden, so dass
$$T(n) = n + 5 \leq c \cdot n$$
gilt, für $n \geq n_0$. Wir formen $n + 5 \leq c \cdot n$ um zu
$$5 \leq (c - 1) \cdot n.$$
Wählen wir $c = 2$ ($c = 1$ würde wenig Sinn machen, ansonsten ist die Wahl willkürlich), so erhalten wir
$$5 \leq n.$$
Also muß n nur größer als 5 sein, und somit haben wir schon n_0, nämlich
$$n_0 = 5.$$
2. $T(n) := 28 \cdot n + 42 = \mathcal{O}(n)$. Auch hier findet man Konstanten c und n, nur eben mit anderen Werten als im ersten Beispiel.
3. $T(n) := 10^{12} \cdot n + 2^{52} = \mathcal{O}(n)$. Hier werden die Konstanten sehr groß sein.
4. $T(n) := n^2 + n - 1 = \mathcal{O}(n^2)$. Dies ist ein Beispiel für quadratische Ordnung.

◀

Man muß nicht unbedingt die passenden Konstanten finden, um heraus zu finden von welcher Ordnung eine Funktion f ist. Folgende Aussage macht uns das Leben leichter:

Satz 2.11 ($\mathcal{O}$ und Grenzwert)
Seien f und g positive reellwertige Funktionen, so gilt

$$\text{Falls der Grenzwert } \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \text{ existiert, so gilt } f = \mathcal{O}(g).$$

◀

Beweis: Sei $\kappa = \lim_{n\to\infty} \frac{f(n)}{g(n)}$. Dann gilt

$$\forall \varepsilon > 0 : \exists n_0 : \forall n \geq n_0 : \left| \frac{f(n)}{g(n)} - \kappa \right| < \varepsilon.$$

Sei nun $\varepsilon = 1$. Dann ergibt sich

$$\left| \frac{f(n)}{g(n)} \right| - |\kappa| \leq \left| \frac{f(n)}{g(n)} - \kappa \right| < 1.$$

Also ist ($\frac{f}{g}$ ist positiv, da f und g positiv sind)

$$\frac{f(n)}{g(n)} \leq 1 + \kappa.$$

Wähle nun $c = 1 + \kappa$. Das n_0 ist dadurch entsprechend gegeben und wir bekommen

$$f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0.$$

□

Man kann außerdem sagen, wenn $\frac{f}{g}$ über alle Grenzen wächst, dann gilt $f = \mathcal{O}(g)$ *nicht*.

Beispiel 2.12
$T(n) := 36n^2 - 3n + 7$. Es gilt

$$\lim_{n\to\infty} \frac{36n^2 - 3n + 7}{n^2} = 36.$$

Der Grenzwert existiert also und es gilt daher $36n^2 - 3n + 7 = \mathcal{O}(n^2)$. ◀

Bei Polynomen setzt sich also der höchste Exponent durch. Allgemein kann man sagen, dass ein Polynom k-ten Grades sich verhält wie $\mathcal{O}(n^k)$.

Aufgabe 2.13

1. Gilt $2\sqrt{n} + 5n = \mathcal{O}(n)$? Antwort: Der höchste Exponent, also $5n$, setzt sich durch. Es gilt ja $\lim \frac{2\sqrt{n}+5n}{n} = 5$. Also ist die Aussage richtig.
2. Gilt $2\sqrt{n} + 5 = \mathcal{O}(n)$? Antwort: Diesmal ist der Grenzwert des Quotienten sogar 0. Die Aussage ist richtig. Man kann aber sogar noch eine bessere Aussage machen, nämlich $2\sqrt{n} + 5 = \mathcal{O}(\sqrt{n})$.

◀

In der letzten Aufgabe haben wir gesehen, dass Funktionen $\mathcal{O}(n)$ als auch $\mathcal{O}(\sqrt{n})$ sein können. Tatsächlich gibt es auf den $\mathcal{O}$-Mengen eine Ordnung, nämlich

$$\mathcal{O}(1) \subset \mathcal{O}(\ln n) \subset \mathcal{O}(n^a) \subset \mathcal{O}(n) \subset \mathcal{O}(n \ln n) \subset \mathcal{O}(n^b) \subset \mathcal{O}(2^n),$$

wobei $0 < a < 1$ und $1 < b$.

Aufgabe 2.14
Gilt $\ln n = \mathcal{O}(n)$? Beweisen Sie dies! Antwort: Ja, dies gilt. Bei der Limesbildung des Quotienten erhalten wir wieder mit der l´Hôpitalschen Regel, dass der Grenzwert existiert und Null ist. ◀

Rechenregeln für $\mathcal{O}$-Notation Betrachten wir folgenden Programmabschnitt:

```
{
     komplizierterAlgorithmus.erstellen();
     komplizierterAlgorithmus.ausrechnen();
}
```

Von den Funktionen `erstellen()` und `ausrechnen()` ist bekannt, dass sie einen Aufwand von $\mathcal{O}(f)$ bzw. $\mathcal{O}(g)$ haben. Welchen Aufwand hat dann der gegebene Programmabschnitt?

Antwort: Der Aufwand summiert sich. Man bekommt also $\mathcal{O}(f) + \mathcal{O}(g)$. Angenommen, einer der beiden Aufwände ist dominant, wir haben also

$$f = \mathcal{O}(g) \text{ oder } g = \mathcal{O}(f).$$

Dann gilt:

$$T(n) = \begin{cases} \mathcal{O}(f) & falls \quad g = \mathcal{O}(f) \\ \mathcal{O}(g) & falls \quad f = \mathcal{O}(g) \end{cases}$$

Falls beide Aufwände von der Ordnung $\mathcal{O}(f)$ sind, so erhalten wir als Gesamtaufwand:

$$T(n) = \mathcal{O}(f) + \mathcal{O}(f) = \mathcal{O}(f).$$

Aufgabe 2.15
Beweisen Sie, dass gilt $\mathcal{O}(f) + \mathcal{O}(f) = \mathcal{O}(f)$.

Beweis: Gemeint ist ja folgendes: seien g und h Funktionen mit $g = \mathcal{O}(f)$ und $h = \mathcal{O}(f)$, dann gilt

$$g + h = \mathcal{O}(f).$$

Die Konstante c, die hier nach Definition gefordert ist, ergibt sich als

$$c = c_g + c_h$$

wobei c_g und c_h die Konstanten für g und h sind. Als n_0 wählt man das Maximum der entsprechenden Werte von g und h. ◀

Diesmal betrachten wir folgendes Programm:

```
{
   for ( int i = 0; i < N ; i++ )
   {
      komplizierterAlgorithmus.ausrechnen();
   }
}
```

Angenommen, die Routine `ausrechnen()` hat eine Aufwandsordnung von $\mathcal{O}(f(N))$. Wie groß ist dann die Aufwandsordnung des gegebenen Programmabschnitts?

Antwort: Diesmal bekommen wir das Produkt

$$N \cdot \mathcal{O}(f(N)).$$

Allgemeiner möchte man also wissen, was das Produkt von zwei Funktionen mit bekannten Ordnungen für eine Ordnung hat. Es gilt

$$\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g).$$

Gemeint ist wieder: seien ϕ und γ zwei Funktionen mit $\phi = \mathcal{O}(f)$ und $\gamma = \mathcal{O}(g)$, so ist

$$\phi \cdot \gamma = \mathcal{O}(f \cdot g).$$

Die Konstante $c_{\phi \cdot \gamma}$ ergibt sich diesmal als

$$c_{\phi \cdot \gamma} = c_f \cdot c_g,$$

und das n_0 für $\phi \cdot \gamma$ ist wieder das Maximum der entsprechenden n_0's von f und g.

Beispiel 2.16
In folgendem Programmcode untersuchen wir die Anzahl der Aufrufe von `j += i` in der Routine `ausrechnen()`[3].

```
{
   for ( int i = 0; i < 2*n ; i++ )
   {
       int k = 0;
       while ( k++ < 500 )
               komplizierterAlgorithmus.ausrechnen(k);
   }
}
komplizierterAlgorithmus::ausrechnen(int j)
{
   for ( int i = 0; i < n ; i++ )
   {
       j += i;
   }
}
```

Die Schleife in `ausrechnen()` wird n-mal durchlaufen. Die Routine `ausrechnen()` wird 500 Mal durchlaufen. Das ergibt $500 \cdot n$. Das Ganze wird $2 \cdot n$ mal durchlaufen; ergibt $2 \cdot n \cdot 500 \cdot n$. Dies ergibt eine Aufwandsfunktion der Ordnung

$$\mathcal{O}(n^2).$$

◀

Aufgabe 2.17
Welche Aussagen sind wahr?

1. $n^2 = \mathcal{O}(n^3)$
2. $n^3 = \mathcal{O}(n^2)$
3. $2^{n-1} = \mathcal{O}(n^2)$
4. $(n+1)! = \mathcal{O}(n!)$
5. $\sqrt{n} = \mathcal{O}(\ln n)$
6. $28 = \mathcal{O}(1)$
7. $12 \cdot n + 5 = \mathcal{O}(n \ln n)$
8. $n \ln n = \mathcal{O}(n^2)$

◀

Wir wollen die Aufwandsfunktion auch nach unten abschätzen, das heißt den Aufwand im besten Fall beschreiben. Dazu gibt es folgende Schreibweise.

Definition 2.18 (Groß-Omega)
$g = \Omega(f) :\Longleftrightarrow \exists c \in \mathbb{R}^+, c > 0\ n_0 \in \mathbb{N}$ so dass $\forall n \geq n_0\ c \cdot f(n) \leq g(n)$. ◀

Es gibt eine Symmetrie zwischen $\mathcal{O}$ und Ω, die in folgendem Satz zum Ausdruck kommt.

[3]Das Programm ergibt nicht viel Sinn. Es geht mir um geschachtelte Schleifen.

Satz 2.19
Seien $f, g : \mathbb{N} \to \mathbb{R}_0^+$ beliebige Funktionen. Dann gilt

$$f = \mathcal{O}(g) \iff g = \Omega(f).$$

◀

Beweis als Übungsaufgabe. □

Im Idealfall kann eine Aufwandsfunktion sowohl nach oben ($\mathcal{O}$) als auch nach unten (Ω) durch die selbe Funktion abgeschätzt werden. Man verwendet dann die Notation Groß-Theta (Θ):

Definition 2.20 (Groß-Theta oder Exakte Ordnung)
$g = \Theta(f) : \iff \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N}$ so dass$\forall n \geq n_0 : c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$. ◀

Es gilt folgende Eigenschaft der exakten Ordnung.

Satz 2.21
$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$. ◀

Beispiel 2.22
Es gilt

$$f(n) := \frac{1}{2}n^2 - 3n = \Theta(n^2),$$

denn

$$f = \mathcal{O}(n^2) \text{ und } f = \Omega(n^2).$$

Letzteres gilt wegen

$$n^2 = \mathcal{O}(\frac{1}{2}n^2 - 3n) = \mathcal{O}(n^2)$$

mit Anwendung des Satzes 2.19. ◀

Häufig auftretende Ressourcenanforderungen

- O(1) (konstant). Ein Algorithmus ohne Schleifen. die von der Anzahl der Eingabedaten abhängen (Laufzeit) oder der nur einfache Variablen oder solche mit konstanter Größe verwendet (Speicherbedarf).
- O(log n) (logarithmisch, unabhängig von der Basis). Ein Algorithmus, der (z.B. durch fortgesetzte Halbierung der Eingabe) nur logarithmisch viele Elemente anguckt (Laufzeit). Logarithmischer Speicherbedarf kann durch entsprechende Rekursion (auf dem Laufzeitkeller) zustande kommen.
- O(n) (linear). Ein Algorithmus, der jedes Eingabeelement einmal oder öfter anguckt (Laufzeit) und einmal oder öfter speichert (Speicherbedarf).
- 0(n log n.) (überlogarithmisch). Entsteht häufig durch Unterteilung in kleinere Unterprobleme (Quicksort).
- O(n^2) (quadratisch). Typisch für paarweise Verarbeitung, z.B. einfache Sortierverfahren („jeder mit jedem“).
- 0(n^3) (kubisch). Einige Verfahren zum Lösen von Gleichungssystemen, einige Optimierungs- und Parsingverfahren.
- O(2^n) (exponentiell). Klassische Beispiele: Aufzählen der Potenzmenge einer Menge mit n Elementen (die selber überabzählbar groß ist), Aufzählen der möglichen Belegungen von n booleschen Variablen mit den beiden Wahrheitswerten.

- O(n!) (faktoriell). Beispiel: Aufzählen aller Permutationen von n Elementen.

Man kann sich eine kleine Faustregel ins Gedächtnis einprägen: Mit einer Komplexität unterhalb von O(n), also etwa O(1) oder 0(log n), kann man kaum etwas nicht-triviales anstellen.

Der Grund besteht darin, dass ein Algorithmus von solcher Komplexität nicht einmal alle seine Eingabedaten angucken kann. Dafür braucht er ja schon n Schritte. Entweder tut der Algorithmus irgendetwas fürchterlich uninteressantes – irgendein Element aus einem array lesen ist O(1) – oder seine Eingabedaten sind schon sehr hoch strukturiert – Binärsuche in einem sortierten Array ist O(log n) – und man fragt sich, wie teuer es war, diese Struktur aufzubauen. Oder er „weis“ (explizit durch „nachdenken“ oder implizit durch das know how des Entwicklers), warum er sich bestimmte Eingabedaten nicht angucken muss.

Im Allgemeinen bezeichnet man alle Algorithmen, deren Komplexität nach oben durch ein Polynom beschränkt ist, also O(P(n)) mit einem Polynom P, als handhabbar (engl. tractable), während etwa ein $O(2^n)$-Algorithmus als nicht handhabbar (engl. intractable) gilt. Man mag darüber streiten, ob ein Algorithmus mit der Komplexität $O(n^{93})$ als „handhabbar“ bezeichnet werden kann. Interessanterweise sind allerdings solche Polynome in der Praxis recht selten – oder fällt Ihnen ein Verfahren mit dieser Komplexität ein ? Erfahrungsgemäß gilt für polynomial beschränkte Algorithmen, dass der Grad des Polynoms meistens ≤ 3 ist.

Andererseits ist ein $O(n^2)$ Verfahren manchmal auch schon nicht mehr praktikabel, wenn es sich um große n handelt. Typische Beispiele sind eben die bekannten Sortierprobleme, bei denen der Schritt vom quadratischen zum überlogarithrnischen Verfahren ganz entscheidend sein kann, wenn man große Mengen sortieren will.

Aufgabe 2.23
Entwickeln Sie eine Idee, mit der Sie den angenäherten Wert $10000! = 3 * 10^{35659}$ auf einem „normalen“ Computer bestimmen können. Für Besitzer eines Mathematikbuchs: Gucken Sie im Stichwortverzeichnis unter dem Begriff „Stirlingsche Formel“ nach! Wenn Ihr Buch diesen Begriff nicht kennt, werfen Sie es weg. ◄

Man könnte verzweifeln, wenn man nur das Wachstum von n log n sich anschaut! Anscheinend sind sowieso nur die logarithmischen Verfahren in der Praxis brauchbar, vielleicht noch die linearen. Alles „darüber“ scheint schrecklich zu sein. Aber das lässt sich leider nicht immer realisieren. Meistens sind wir schon zufrieden, wenn wir es mit einem quadratischen Algorithmus zu tun haben. Wie man zunächst mit solchen Problemen umgehen kann, finden Sie am Ende dieses Kapitels.

2.4 Darstellungen

Im Zusammenhang mit der Analyse von Algorithmen verwendet man oft eine doppelt-logarithmische Darstellung der Aufwandsfunktion. Dies bedeutet, dass sowohl die x-Achse, als auch die y-Achse mit einer logarithmischen Skala versehen sind. Hier verwendet man oft einen dekadischen oder dyadischen Logarithmus. In diesem doppelt-logarithmischen Koordinatensystem sehen die bekannten Funktionen natürlich anders aus.

Betrachten wir die Funktion

$$y = a \cdot x^k$$

wobei $k \in \mathbb{N}$. Die Achsen der doppelt-logarithmischen Achse nennen wir ξ und ζ. Es ist dann also

$$x = 10^{\xi} \text{ und } y = 10^{\zeta}.$$

Dies setzen wir in unsere Funktionengleichung ein und erhalten

$$y = 10^{\zeta} = a \cdot (10^{\xi})^k.$$

Umformungen ergeben

$$10^{\zeta} = a \cdot (10^{\xi})^k = a \cdot 10^{\xi \cdot k}.$$

Nun lösen wir nach ζ auf, um zu erkennen, wie sich ζ in Abhängigkeit von ξ darstellt:

$$\zeta = \log_{10}(a \cdot 10^{\xi \cdot k}) = \log_{10} a + \log_{10}(10^{\xi \cdot k}) = \log_{10} a + k \cdot \xi.$$

ζ ist eine lineare Funktion in ξ, wobei k die Steigung angibt und $\log_{10} a$ den Schnittpunkt mit der ζ-Achse.

Ein allgemeines Polynom vom Grad k der Form

$$p(x) = \sum_{i=1}^{k} a_i x^i$$

ist keine Gerade in der doppelt-logarithmischen Darstellung. Je größer ξ wird, desto mehr sieht der Graph wie eine Gerade aus, mit einer Steigung die dem höchsten Exponenten – also dem Grad des Polynoms – entspricht.

Interessant sind auch folgende Funktionen. Betrachten wir

$$y = \log x.$$

Mit $x = 10^{\xi}$ und $y = 10^{\zeta}$ wie oben erhalten wir

$$10^{\zeta} = \log 10^{\xi} = \xi.$$

Auflösen nach ζ ergibt

$$\zeta = \log \xi.$$

Ein Logarithmus bleibt also ein Logarithmus. Egal in welcher Darstellung. Genauso hält es sich mit der Funktion

$$y = 10^x.$$

Auch hier erhält man

$$\zeta = 10^{\xi}.$$

Eine Exponentialfunktion bleibt also in doppelt-logarithmischer Darstellung eine Exponentialfunktion.

Aufgabe 2.24

1. Wie sieht eine Gerade $y = a \cdot x + b$ in doppelt-logarithmischer Darstellung aus? (insb. für $b = 0$).
2. Wie sieht die Wurzelfunktion $y = \sqrt{x}$ in doppelt-logarithmischer Darstellung aus?
3. Wie sieht die Exponentialfunktion $y = e^x$ in doppelt-logarithmischer Darstellung aus?

◀

2.5 Rekursive Algorithmen

Oft kann man ein Problem das von einem Parameter N abhängt zurückführen auf ein gleichartiges Problem, in dem $N - 1$ oder ein anderer Wert kleiner als N vorkommt. Zum Beispiel:

1. Die Berechnung eines Polynoms vom Grad N kann auf Berechnung eines Polynoms vom Grad $N - 1$ zurückgeführt werden:

$$\underbrace{\sum_{i=0}^{N} a_i x^i}_{\text{Grad } N} = (x \cdot \underbrace{\sum_{i=1}^{N} a_i x^{i-1}}_{\text{Grad } N-1}) + a_0.$$

2. Die beiden Begriffe Rekursion und Iteration sollen anhand der Fakultätsfunktion motiviert werden. Diese Funktion ist u.a. definiert als

$$n! \;=\; \Pi_{i=1}^{n} i \;=\; n \cdot (n-1) \cdot \ldots * 2 \cdot 1 \;=\; n \cdot (n-1)!$$

Diese Definitionen zeigen schon die unterschiedlichen Definitionsmöglichkeiten auf. Zunächst soll die letzte Definition aufgegriffen werden, da sie eine rekursive Definition ist: die Fakultät einer Zahl n wird berechnet, indem diese Zahl mit der Fakultät der Zahl $n-1$ multipliziert wird. Eine ähnliche Definition kennen wir von der Liste: eine Liste besteht aus einem Element und einer (Rest-)Liste. In der Implementierung könnte dies 1:1 durch eine rekursiv implementierte Funktion umgesetzt werden.

Die linke Definition Π stellt eine Iteration dar, die z.B. mittels einer `for`-Anweisung leicht realisiert werden kann. Die mittlere Definition beinhaltet genau genommen mit den ... ebenfalls eine Schleife, da nur Schleifen uns eine „dynamische Anpassung“ des Programms bzw. der Programmlänge an die Eingabegröße erlauben.

3. Die N-te Zeile im Pascalschen Dreieck kann berechnet werden, indem man die $N-1$-ste Zeile verwendet:

$$\begin{array}{c}
1 \\
1 \quad 1 \\
1 \quad 2 \quad 1 \\
1 \quad 3 \quad 3 \quad 1 \\
1 \quad 4 \quad 6 \quad 4 \quad 1 \\
1 \quad 5 \quad 10 \quad 10 \quad 5 \quad 1 \\
1 \quad 6 \quad 15 \quad 20 \quad 15 \quad 6 \quad 1
\end{array}$$

Ist N die aktuelle Zeile, und k die aktuelle Spalte, so lautet die Formel zur Berechnung des Wertes P an der Stelle N, k ($k \geq 2$, für $k = 1$ ist P immer 1)

$$P(N,k) = P(N-1,k-1) + P(N-1,k).$$

Meist schreibt man $P(m,n) = \binom{n}{m}$, ausgesprochen n über m.

Um solche Probleme in der Informatik zu lösen, verwendet man auf eine Technik, die sich *Rekursion* nennt. Man schreibt eine Routine, die das Problem für N löst. In dieser Routine wird aber die Routine selbst mit $N-1$ aufgerufen. Es wird also nur der Rechenschritt von $N-1$ bis N in der Routine implementiert. Natürlich benötigt man noch eine Abfrage, ob $N = 1$ (oder 0) ist. Die Routine ruft sich selbst also $N-1$ mal auf.

Hierbei sollte man beachten, dass alle errechneten Zwischenergebnisse auf dem Stack abgelegt werden. Hat man also nur eingeschränkte Ressourcen (z.B. auf embedded Systemen) zur Verfügung, so sollte man entweder die Größe des Problems (also N) limitieren. Oder man verwendet nicht-rekursive Verfahren und holt sich den erforderlichen Speicher vom Heap (über *new*).

Wir wollen am Beispiel der Fibonacci-Zahlen den Zeitaufwand für die Berechnung durchführen.

Fibonacci-Zahlen

Die Fibonacci-Zahlen treten immer wieder auf bei der Analyse von verschiedenen Algorithmen aber auch im alltäglichen Leben. Sie sind folgendermaßen definiert:

$$\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(n) &= f(n-1) + f(n-2), \qquad \text{falls } n \geq 2.
\end{aligned}$$

Man sieht, dass die n-te Fibonacci-Zahl auf die $n-1$-te und $n-2$-te Fibonacci-Zahl zurückgeführt werden kann. Dies legt nahe, einen rekursiven Algorithmus zur Berechnung der n-ten Fibonacci-Zahl zu verwenden.

```
fibRec(int n)
{
     switch(n)
     {
     case 0:
          return 0;
     case 1:
          return 1;
     default:
          return fibRec(n-1) + fibRec(n-2);
     }
}
```

Kürzer und unter Einhaltung der Java Code-Konventionen sieht das so aus:

```
return (n <= 1) ? n : fibRec(n - 1) + fibRec(n - 2);
```

Betrachten wir zunächst die Zahlen, die dabei entstehen. Setzt man der Reihe nach $N = 0, 1, 2, 3, 4, \ldots$ ein, so erhalten wir die Fibonacci-Zahlen

n	f(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

Hier einige interessante Beispiele:

1. Das Klavier hat in einer Oktave (von C bis C) 5 schwarze Tasten in einer Zweiergruppe und einer Dreiergruppe (2 und 3 und 5 jeweils aus der Fibonacci-Folge). Außerdem gibt es 8 weiße Tasten, so dass man zusammen 13 Töne pro Oktave hat (hierbei ist das C doppelt gezählt, damit das Beispiel paßt), also $5 + 8 = 13$.

2. In Amerika (Illinois) gibt es ein Gänseblümchen (Echinacea purpurea), das hat 89 Blumenblätter. Davon 55 die in die eine Richtung, 34 die in die andere Richtung zeigen. Auch Sonnenblumen weisen ein solches Muster auf, welches je nach Größe der Pflanze Blätteranzahlen der Fibonacci-Folge hat.

3. Man kann eine Figur aus Quadraten malen, wobei man zwei Einer-Quadrate nebeneinander malt. Darüber direkt anschließend ein Zweier-Quadrat, rechts daneben ein Dreier-Quadrat, darunter ein Fünfer-Quadrat welches die beiden Einer und das Dreier berührt usw. Die entstehenden Rechtecke nennt man Fibonacci-Rechtecke. Setzt man die Kantenlängen eines der Rechtecke ins Verhältnis, so erhält man eine Zahl, die gegen den goldenen Schnitt

$$\Phi = \lim_{n \to \infty} \frac{f(n+1)}{f(n)} = \frac{1 + \sqrt{5}}{2}$$

strebt. Dieser hat die Eigenschaft: lange Seite zu kurzer Seite verhalten sich wie Summe der beiden zur längeren.

Um den rekursiven Algorithmus zu analysieren, betrachten wir die Zeile nach `default:`. Hier wird eine Addition ausgeführt. Seien $c(n-1)$ und $c(n-2)$ die Anzahl der Additionen die zur Berechnung von `fibRec(n-1)` und `fibRec(n-2)` benötigt werden. Dann ist die Anzahl $c(N)$ der Additionen zur Bestimmung von `fibRec(n)`

$$c(n) = c(n-1) + c(n-2) + 1.$$

Die Anzahl der Additionen $c(0)$ und $c(1)$ sind beide Male 0, denn `fibRec(0)` und `fibRec(1)` liefert die Ergebnisse ohne Additionen ((`return`) zurück. Man bekommt also eine Rekursion der Form:

$$\begin{aligned} c(0) &= 0 \\ c(1) &= 0 \\ c(n) &= c(n-1) + c(n-2) + 1 \end{aligned}$$

Satz 2.25
Für die Anzahl der Additionen zur rekursiven Berechnung der Fibonacci-Zahlen gilt:

$$c(n) = f(n+1) - 1,$$

wobei $f(n+1)$ die $(n+1)$-ste Fibonacci-Zahl ist. ◄

Beweis: Durch vollständige Induktion.
Induktionsanfang: $c(0) = 0 = f(1) - 1$ und $c(1) = 0 = f(2) - 1$.
Induktionsannahme: Es sei angenommen, dass $n \geq 2$ und dass folgendes bereits bewiesen ist:

$$c(n) = f(n+1) - 1,$$

$$c(n-1) = f(n) - 1.$$

Induktionsschluss: Es wird gezeigt, dass die Aussage für $c(n+1)$ gilt.

$$\begin{aligned} c(n+1) &\overset{\text{Def. von c}}{=} c(n) + c(n-1) + 1 \\ &\overset{\text{Ind. Ann.}}{=} f(n+1) - 1 + f(n) - 1 + 1 \\ &\overset{\text{ausrechnen}}{=} f(n+1) + f(n) - 1 \\ &\overset{\text{Def. von f}}{=} f(n+2) - 1. \end{aligned}$$

□

Die Anzahl der Addition entspricht also praktisch den Fibonacci-Zahlen. Um den Aufwand zur Berechnung der N-ten Fibonacci-Zahl zu bestimmen, müssen wir eine geschlossene Darstellung der Fibonacci-Zahlen finden.

Satz 2.26
Für die Fibonacci-Zahlen gibt es eine geschlossene Darstellung. Seien

$$\Phi = \frac{1+\sqrt{5}}{2} \text{ und } \hat{\Phi} = \frac{1-\sqrt{5}}{2}.$$

Dann gilt

$$f(n) = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n).$$

◄

Beweis: Wir benötigen die Tatsache, dass

$$\Phi^2 = \Phi + 1 \text{ und } \hat{\Phi}^2 = \hat{\Phi} + 1.$$

Dies kann man leicht nachrechnen. Die Aussage selbst beweist man wieder mit vollständiger Induktion.

Induktionsanfang:

$$f(0) = \frac{1}{\sqrt{5}}(\Phi^0 - \hat{\Phi}^0) = 0$$

und

$$f(1) = \frac{1}{\sqrt{5}}(\Phi^1 - \hat{\Phi}^1) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}-1+\sqrt{5}}{2}) = 1.$$

Induktionsannahme: Wir nehmen an, dass $n \geq 2$ und dass folgendes bereits bewiesen ist:

$$f(n) = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n)$$

und

$$f(n-1) = \frac{1}{\sqrt{5}}(\Phi^{n-1} - \hat{\Phi}^{n-1})$$

Induktionsschluss: Wir zeigen die Aussage für $n+1$.

$$\begin{aligned}
f(n+1) &\overset{\text{Def von f}}{=} f(n) + f(n-1) \\
&\overset{\text{Ind.Ann.}}{=} \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n) + \frac{1}{\sqrt{5}}(\Phi^{n-1} - \hat{\Phi}^{n-1}) \\
&\overset{\text{Umrechnen}}{=} \frac{1}{\sqrt{5}}\left((\Phi+1)\Phi^{n-1} - (\hat{\Phi}+1)\hat{\Phi}^{n-1}\right) \\
&\overset{\text{Tatsache von oben}}{=} \frac{1}{\sqrt{5}}\left(\Phi^2 \cdot \Phi^{n-1} - \hat{\Phi}^2 \cdot \hat{\Phi}^{n-1}\right) \\
&\overset{\text{multipliziert}}{=} \frac{1}{\sqrt{5}}\left(\Phi^{n+1} - \hat{\Phi}^{n+1}\right)
\end{aligned}$$

Damit ist die Behauptung bewiesen. □

Wollen wir zum Beispiel wissen, wie viele Additionen $c(n)$ die Berechnung der n-ten Fibonacci-Zahl für $n = 69$ benötigt, so müssen wir rechnen:

$$c(69) = f(70) - 1.$$

Wir erhalten mit dem letzten Satz, dass

$$f(70) \approx 1,9 \cdot 10^{14}.$$

Wenn wir annehmen, dass wir auf einem $2,4$GHz-Rechner pro Sekunde $2.4 \cdot 10^9$ Additionen machen können, so erfordert die Rechnung eine Rechenzeit von

$$\frac{1,9 \cdot 10^{14}}{2,4 \cdot 10^9} = \frac{1,9}{2,4} 10^5 \approx 22h.$$

Dies ist sehr lang und wir werden gleich sehen, dass es auch schneller geht. Zunächst wollen wir betrachten, wie man auf die geschlossene Form der Fibonacci-Zahlen kommt. Dazu benötigt man schon etwas Geschick. Man kann ein Computeralgebrasystem (CAS) – z. B. Maple oder Matlab – zu Hilfe nehmen. Es gibt auch eine Theorie der Differenzengleichungen, mit der man diese Gleichungen manchmal lösen kann.

Nun kann man die 69-ste Fibonacci-Zahl auch schneller bestimmen. Dazu betrachten wir folgendes Programm:

```
fibSeriell(int n)
{
   int a[70];
   a[0]=0;
   a[1]=1;
   for(int i = 2; i < 70; i++)
   {
      a[i] = a[i-1] + a[i-2];
   }
}
```

Dieses Programm führt nur 68 Additionen durch. Auf dem gleichen Rechner wie oben benötigt dieses Programm nur 28 Nano-Sekunden. Das sind $0,028$ Mikro-Sekunden. Es dürfte also etwas schneller ablaufen als das rekursive Verfahren.

Dennoch sollten wir rekursive Verfahren nicht als langsam abstempeln. Wir werden bei den Sortierverfahren im nächsten Kapitel sehen, dass rekursive Verfahren durchaus ihre Berechtigung haben. Das Verfahren welches man verwendet nennt sich *Divide and Conquer*, also zu deutsch Teile-und-herrsche. Es wird rekursiv implementiert. Damit kann man die Aufwandsordnung eines Algorithmus tatsächlich in manchen Fällen verbessern.

Beispiel 2.27 (Türme von Hanoi)
Die „Türme von Hanoi“ sind ein beliebtes Beispiel für rekursive Programmierung. Ein Stapel von Scheiben liegt auf einem Stab. Die Scheiben sind dabei mit von unten nach oben abnehmendem Durchmesser auf den Stab gesteckt . Sie haben die Aufgabe, diese Scheiben auf einen anderen Stab zu legen. Dabei haben Sie folgende Regeln einzuhalten:

- Sie dürfen nur eine Scheibe zur Zeit bewegen.
- Eine Scheibe darf nur auf einer anderen mit größerem Durchmesser liegen.
- Sie haben einen weiteren Stab als „Zwischenablage“.

Der Algorithmus zum Lösen der Aufgabe der Türme von Hanoi läßt sich in Pseudocode so beschreiben: Dabei bewirke die Prozedur `hanoi(n,start,ziel,ablage)` „bewege n Scheiben regelgerecht von `start` nach `ziel` unter Verwendung des Hilfsstabs`ablage`“.

```
hanoi(n, start, ziel, ablage)
        if (n == 1) „move disk from start to ziel\
        else
          hanoi(n-1, start, ablage, ziel)
          „move disk from start to ziel\
          hanoi(n-1, ablage, ziel, start)
```

◀

Wie ist dieser Algorithmus nun zu analysieren? Sei dazu T(n) die Anzahl Züge, die notwendig sind um n Scheiben von einem Stab auf den anderen zu bewegen (unter Verwendung des dritten Stabs) Dann ist

1. $T(1) = 1$
2. $T(n) = 2T(n-1) + 1$

Was heißt das?

$$T(1) = 1 \text{ wegen dem Rekursionsabbruch}$$

$$T(n) = 2^n - 1$$

Aber wie wie kommt man darauf? Generell gilt:

- Das Verhalten rekursiver Algorithmen muss untersucht werden
- Eine geschlossene Formel für das Verhalten muss hergeleitet werden, d.h. nicht iterativ, sondern direkt aus den Eingaben

Manchmal kann man der Folge ihr Bildungsgesetz ansehen: Hier sind die ersten Elemente der Folge

$$1, 3, 7, 15, 31, 63, 127, \ldots$$

Das sieht ja schon mal sehr nach $2^n - 1$ aus. Etwas systematischer bildet man die Differenzen zwischen den Folgenelementen:

$$\begin{array}{ccccccccccccc} 1 & & 3 & & 7 & & 15 & & 31 & & 63 & & 127 \\ & 2 & & 4 & & 8 & & 16 & & 32 & & 64 & \\ & & 2 & & 4 & & 8 & & 16 & & 32 & & \end{array}$$

Ab den ersten Differenzen ändert sich nichts mehr. Gegenüber der Ausgangsgleichung sehen wir hieraus

$$T(n+1) = T(n) + 2^n$$

Also

$$\begin{aligned} T(n+1) &= 2^n + T(n) \\ &= \ldots \\ &= 2^n + 2^{n-1} + \cdots + 2^2 + 1 \\ &= \sum_{i=0}^{n} 2^i \\ &= \frac{1 - 2^{n+1}}{1 - 2} \\ &= 2^{n+1} - 1 \end{aligned}$$

Eine ähnliche Möglichkeit ist das sogenannte „Sondieren“ der Rekursion:

$$T(1) = 1 \text{ wegen dem Rekursionsabbruch}$$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2 \cdot (2 \cdot T(n-2) + 1) + 1 \\ &= 2^2 \cdot T(n-2) + 2^1 + 2^0 \\ &= 2^3 \cdot T(n-3) + 2^2 + 2^1 + 2^0 \\ &= \ldots \\ &= 2^k \cdot T(n-k) + 2^k - 1 \\ &= \ldots \\ &= 2^{n-1} \cdot T(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} \cdot T(1) + 2^{n-1} - 1 \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 (?) \end{aligned}$$

Besser ist es natürlich, die Theorie der Differenzengleichungen anzuwenden. Satz A.26 ist die Lösung von

$$t_{n+1} = 2 \cdot t_n + 1 \tag{2.1}$$

gerade

$$t_n = 2^n - 1 \tag{2.2}$$

Unter Bezug auf Satz A.26 sind wir damit fertig.

Haben wir aber eines der heuristischen Verfahren angewandt, so sind wir gut beraten, die Formel zu beweisen. Wie macht man das? Mittels vollständiger Induktion!

Machen wir das doch einfach: Wir beweisen die Aussage

$$T(n) = 2^n - 1$$

für die Türme von Hanoi mittels vollständiger Induktion. Dabei bezeichnet IV die Induktionsverankerung, IA die Induktionsannahme und IS den Induktionsschluss.

$$\begin{array}{lll} IV: & T(1) & = 2^1 - 1 \\ & & = 1 \\ IA: & T(n_0) & = 2^{n_0} - 1 \\ IS: & T(n_0+1) & = 2 \cdot T(n_0) + 1 \\ & & = 2(2^{n_0} - 1) + 1 \texttt{ (aufgrund der IA)} \\ & & = 2^{n_0+1} - 2 + 1 \\ & & = 2^{n_0+1} - 1 \end{array}$$

Oft hilft bei den heuristischen Verfahren Intuition, z. B.

1. Man addiere auf beiden Seiten der Rekursionsgleichung 1.

$$\begin{array}{lll} T(1) + 1 & = & 2 \\ T(n) + 1 & = & 2T(n-1) + 2 \end{array}$$

2. Ersetze U(n) = T(n) + 1 und setze dies in die Rekursionsgleichung ein:

$$\begin{array}{lll} U(1) & = & 2 \\ U(n) & = & 2U(n-1) \end{array}$$

3. Nun „sieht“ man die Lösung

$$U(n) = 2^n$$

4. Einsetzen in die Ausgangsgleichung liefert:

$$\begin{array}{lll} T(n) & = & U(n) - 1 \\ & = & 2^n - 1 \end{array}$$

(Ein Teil des Beweises des genannten Satzes läuft übrigens ganz analog.)

Beispiel 2.28 (Summe der ersten n Zahlen)
Die Folge wird definiert durch:

1. T(1) = 1
2. T(n) = T(n-1) + n

Sondieren liefert

$$\begin{array}{lll} T(n) & = & T(n-1) + n \\ & = & T(n-2) + (n-1) + n \\ & = & T(n-3) + (n-2) + (n-1) + n \\ & = & \dots \\ & = & T(1) + 2 + \dots + (n-2) + (n-1) + n \\ & = & 1 + 2 + \dots + (n-2) + (n-1) + n \\ & = & (1+n) + (2+n-1) + \dots + \\ & = & (n+1) + (n+1) + \dots \\ & = & \frac{n(n+1)}{2} \end{array}$$

Der Beweis mittels vollständiger Induktion ist einfach und wurde in der Vorlesung vorgeführt. ◀

Problembehandlung bei komplexen „Problemen“

Nachfolgende Möglichkeiten gibt es, um mit komplexen Algorithmen umzugehen. Wir betrachten dies an einem Beispiel, für das Schachspiel in einem Baum alle möglichen Stellungen zu berechnen. Für dieses Problem gibt es einen Algorithmus, der jedoch so viele Resourcen (Rechenzeit und Platz) benötigt, dass sie bis dato noch nicht zur Verfügung stehen. Von daher ist dies ein interessantes Beispiel, weil es einen Algorithmus gibt, der die Lösung exakt/optimal berechnen kann, die Ausführungszeit jedoch nicht effizient und (bis dato) damit nicht praktikabel ist.

- **Natürliche Einschränkung** des Problems: Evtl. muss „nur“ ein Spezialfall gelöst werden. Hier hilft ein Nachfragen beim Auftraggeber bzw. eine genauere Analyse des gestellten Problems, d.h. die Strategie ist, genauer hinzuhören, was der Auftraggeber wirklich will bzw.durch eine genauere Analyse des Problems dessen Charakter und damit die Problemklasse genauer bestimmen zu können.

  Im Beispiel: Möchte der Kunde wirklich alle Stellungen des Schachspiels oder evtl. nur Endspielsituationen ?

- **Erzwungene Einschränkung** des Problems: „nur“ gutmütige Probleme oder Spezialfälle lassen sich aufschreiben bzw. werden gelöst. Der Auftraggeber muss darauf hingewiesen werden. Ggf. kann die „Auslagerung“ des worst case erreicht werden, indem ein vorgelagertes Programm z.B. zu einem für den Auftraggeber uninteressantem Zeitpunkt eine Vorstrukturierung der Daten vornimmt und dadurch zur „Hauptarbeitszeit“ die gewünschte Effizienz erreicht.

  Im Beispiel: Es werden ausgehend von nur einer Spielsituation alle Stellungen berechnet, die durch Züge der Bauern möglich sind.

- **Redefinition des Problems**: Betrachtet man das Problem aus einer anderen Sichtweise ist die Komplexitätsanalyse einfacher (quasi Vermeidung des „mit Kanonen auf Spatzen schießen“). So kann z.B. statt nach allen möglichen Situationen zu suchen, es effizienter sein, nach allen nicht möglichen Situationen zu suchen. Im Wesentlichen muss man für eine Redefinition das Problem selbst intensiv analysieren oder (genereller Ansatz) die Sichtweise (das Konzept) selbst genauer analysieren.

  Im Beispiel: Könnte man das Schachspiel auch als Tic-Tac-Toe Spiel auffassen ? Oder als Problem des minimalen Gerüstes ?

- **Approximationen**: Reduktion der Ansprüche an die Qualität der Antworten; Aufgabe des Optimalitätsanspruches. Dies ist auch eine erzwungene Einschränkung: da ein exaktes bzw. optimales Ergebnis nicht erreicht werden kann, wird nur eine Annäherung geliefert.

  Im Beispiel: Es werden nur wenige Stellungen berechnet, um daraus abzuschätzen, was wohl der beste Zug sein könnte. Das Ergebnis kann ein sehr guter bis hin zu einem sehr schlechten Zug sein.

- **Veränderung** des Algorithmus: Mögliche Redundanzen vermeiden bzw. verkleinern und ggf. andere Operationen für die Problemlösung verwenden. Evtl. auch mit anderen Strategien arbeiten. Hier können z.B. generelle Strategien zum Einsatz kommen, wie etwa Teile-und-Herrsche oder auch problemspezifische Strategien, wie z.B. bei der Suche nach Primzahlen.

  Im Beispiel: Es werden keine Stellungen mehrfach berechnet. Durch z.B. eine Stellungs-DB wird zunächst geprüft, ob die Stellung schon einmal bewertet wurde. Als weitere Möglichkeit werden die Stellungen selbst in einer anderen, effizienteren Form dargestellt, z.B. binäres Muster und die Züge werden durch `shift`-Operationen realisiert.

Bei den Approximationsalgorithmen werden also Verfahren benötigt, die „nur noch“ eine gute Lösung liefern, was auch immer „gut“ bedeutet. Dies führt zu nichtdeterministischen Programmen bzw. Approximationsalgorithmen, da eine Auswahl getroffen werden muss, d.h. es muss entschieden werden, welche Suchpfade (den Lösungsprozess hier als Suchprozess betrachtet) beschritten

werden, d.h. es muss in weitestem Sinne approximiert werden. Zwei bzw. drei Begriffe werden in diesem Zusammenhang wichtig: „Korrektheit“ (*saftey*), „Vollständigkeit“ (*lifeness*) und „Korrekt- und Vollständigkeit“. Die drei Begriffe seien am Beispiel eines Garbage Collector erklärt:

Korrektheit Ein Garbage Collector (GC) arbeitet korrekt, wenn das, was er freigibt, auch wirklich garbage ist. Dies bedingt jedoch nicht, dass jeder garbage freigegeben wird!

Vollständigkeit Ein Garbage Collector (GC) arbeitet vollständig, wenn er jeden garbage freigibt. Dies bedingt jedoch nicht, dass das, was er freigibt, auch wirklich garbage ist!

Korrekt- und Vollständigkeit Ein Garbage Collector (GC) arbeitet korrekt und vollständig, wenn er jeden garbage freigibt und wenn das, was er freigibt auch wirklich garbage ist.

Sehr oft wird die Vollständigkeit zu Gunsten der Korrektheit aufgegeben. Im Falle von Suchmaschinen (hier mit den Begriffen „precision“ und „recall“ arbeitend) wird sogar auf beides verzichtet: nicht jedes gelieferte Dokument auf eine Frage ist präzise/korrekt und nicht alle präzisen/korrekten Dokuemnte werden geliefert (recall).

2.6 Fazit aus Sicht der Konstruktionslehre

Die wesentlichen Punkte sind die theoretische Laufzeitklasse, die praktische Laufzeitklasse und die Möglichkeiten, beide Laufzeiten zu verbessern. Bei den Laufzeitklassen ist zu prüfen, ob die Klasse des Algorithmus (der Problemlösemethode) mit der Klasse des Problems übereinstimmt. Der mögliche Unterschied hier gibt den Rahmen für Verbesserungen vor, es sei denn, man verändert das Problem.

Für die Konstruktion von Systemen ist zunächst die theoretische Laufzeitklasse wichtig, um abschätzen zu können, was einen erwartet: dies nicht im Sinne einer konkreten Laufzeit sondern im Sinne einer Verfügbarkeit des Dienstes bei steigender Problemgrösse. Die praktische Laufzeit entscheidet dann, ob der Dienst überhaupt verwendet wird.

Genügt eine der Laufzeitklassen nicht den Anforderungen der Anwendung (z.B. in der maximal zu bearbeitenden Problemgrösse (etwa 10.000 Zugriffe auf SUCHE pro Sekunde) oder z.B. der Laufzeit einer Problemlösung (etwa Zeit zwischen Anfrage und Antwort)), müssen die Strategien für eine mögliche Verbesserung angewendet werden.

2.7 Aufgaben

1. ([03]) Ermitteln Sie die Aufwandsfunktion für die Operationen `pop` und `push` eines Stacks!
2. ([04]) Es sei ein ADT X gegeben (denken Sie an Liste, Array, Tablespace ...). Dessen Implementierung sei so, dass eine Vergrößerung einen fixen Aufwand f erfordert und für jedes Element, für das Platz geschaffen wird, ein variabler Aufwand von v anfällt. Untersuchen Sie die folgenden Wachstumstrategien auf ihre Effizient:
   2.1. Minimaler Speicherplatz: Der ADT wird leer angelegt und immer, wenn ein Element eingefügt wird, um (den Platz für) ein Element vergrößert.
   2.2. Prozentuale Vergrößerung: Ist der Platz gefüllt, so wird der ADT (den Platz für) p% mehr Elemente als vorher vergrößert.
3. ([03]) Zeigen und beweisen Sie, dass Stack- und Queue-Operationen mit konstantem Aufwand implementiert werden können!
4. ([03]) Beweisen Sie Satz 2.21!
$$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$$
5. ([03]) Wie sieht die Wurzelfunktion $y = \sqrt{x}$ in doppelt logarithmischer Darstellung aus?

6. ([03]) Wie sieht die Exponentialfunktion $y = e^x$ in doppelt logarithmischer Darstellung aus?
7. ([03]) Wie sieht die Funktion $y = \frac{1}{x}$ in doppelt logarithmischer Darstellung aus?
8. ([06]) Analog die Darstellung in lin-log, log-lin. Welche der drei Darstellungen sind für welche Zwecke sinnvoll?
9. ([06]) Beweisen Sie die folgende Aussage!

$$\sum_{k=1}^{n} \lfloor \frac{k}{2} \rfloor = \lfloor \frac{n^2}{4} \rfloor$$

10. ([06]) Beweisen Sie die folgende Aussage!

$$\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil = \lceil \frac{n \cdot (n+2)}{4} \rceil$$

11. ([03]) Beweisen Sie bitte folgende Aussage!

$$\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2 \ \forall n \in \mathbb{N}^+$$

12. ([03]) Beweisen Sie bitte die folgende Aussage!

$$\sum_{k=1}^{n} 2k = n \cdot (n+1) \ \forall n \in \mathbb{N}^+$$

13. ([03]) Beweisen Sie bitte die folgende Aussage!

$$\sum_{k=1}^{n} (2k-1) = n^2 \ \forall n \in \mathbb{N}^+$$

14. Gegeben sei folgendes Codefragment; n sei eine positive natürliche Zahl.

```
int j = 0, i = n;
while (i > 0) {
 for(j = i; j <= n; j++) {
  AnweisungY; }
 for(j = n; j > 0 ; j--) {
  AnweisungY; }
 AnweisungY;
 i--; }
```

Geben Sie eine Funktion f an, die in Abhängigkeit von n bestimmt, wie oft *AnweisungY* ausgeführt wird. Ordnen Sie den Algorithmus der bestmöglichen (niedrigsten) Komplexitätsklasse O in Abhängigkeit von n zu.

15. Für die Aufgabe, n Adressen zu sortieren, stehen drei Verfahren zur Verfügung: Eine Implementierung von Heapsort benötigt eine Zeit von $240 * log_2(n)\mu s$, eine Implementierung von Bubblesort genau $3 * n^2 \mu s$ und eine Implementierung von Insertion Sort $2 * n^2 + 6n + 16\ \mu s$. Für welche Datenbankgrössen n lohnt sich welches der Sortierverfahren? Bestimmen Sie die „Break-even"-Punkte! Für den Break-even-Punkt mit Heapsort kann kurz vor der formalen Auflösung, d.h. einer Gleichung ohne log_2, mit den beiden Werten $n = 22$ und $n = 21$ getestet werden.

16. Gegeben sei folgendes Codefragment; n sei eine positive natürliche Zahl.

```
01 public static double foo(double a[], int i, int j) {
02 int mid;
03 double foo1, foo2 ;
04 if ( i == j ) return a[ i ] ;
05 else {
06    mid = (int) (((double) (i + j) ) / 2.0) ;
07    foo1 = foo( a, i, mid ) ;
08    foo2 = foo( a, mid+1, j ) ;
09    if ( foo1 > foo2 ) return foo1 ;
10   else return foo2 ; }
}
```

    16.1. Welchen Wert liefert der Aufruf `foo(a,1,8)` zurück, wenn die Elemente `a[1], ... , a[8]`, mit den Werten `3.5, 5.5, 12.5, 4.5, 6.5, 2.5, 0.5, 7.5` belegt sind ? Begründen Sie Ihre Antwort durch Angabe des Ablaufes!

    16.2. Was berechnet der Aufruf `foo( a, 1, n )` im Allgemeinen ? (Die Feldelemente a[1], ..., a[n] seien mit reellen Zahlen vorbelegt.). Begründen Sie Ihre Antwort durch allgemeine Erklärungen an Hand des Algorithmus (dazu ggf. die Zeilennummern verwenden)!

    16.3. Auf welchem algorithmischen Konstruktionsprinzip basiert die Funktion `foo` ? Denken Sie auch hier an die Begründung Ihrer Antwort!

    16.4. Geben Sie eine möglichst kleine O-Schranke für die Zeitkomplexität des Aufrufs `foo( a, 1, n )` (in Abhängigkeit von n) an. Zur Vereinfachung dürfen Sie annehmen, dass n von der Form $n = 2^k, k \in N_0$ ist.

    Sie können den Aufwand einer Rückgabe, Division und eines Vergleiches jeweils mit 1 berechnen.

17. Was ist ein Algorithmus?

18. Welche charakteristischen Merkmale hat ein Algorithmus?

19. Nennen Sie bitte einige Beispiele für Algorithmen!

20. Wie funktioniert das Sieb des Erathostenes?

Kapitel 3

Sortieren

Sortieren ist eine Aufgabe, die in der IT in vielen Varianten auftritt. Es kann notwendig oder sinnvoll sein, eine mehr oder weniger große Menge von Objekten temporär in eine bestimmte Reihenfolge zu bringen, die für eine Verarbeitung notwendig oder wünschenswert ist. In anderen Fällen möchte man eine bestimmte Reihenfolge dauerhaft erhalten, auch wenn neue Objekte eingefügt oder vorhandene geändert oder gelöscht werden.

Untersuchungen von Computerherstellern und -nutzern zeigen seit vielen Jahren, dass mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge entfällt. Es ist daher nicht erstaunlich, dass große Anstrengungen unternommen wurden möglichst effiziente Verfahren zum Sortieren von Daten mit Hilfe von Computern zu entwickeln. Das gesammelte Wissen über Sortierverfahren füllt inzwischen Bände. Allein der Klassiker von Knuth [Knu97c] enthält 391 Seiten zu diesem Thema. Noch immer erscheinen neue Erkenntnisse über das Sortieren in wissenschaftlichen Fachzeitschriften und zahlreiche theoretisch und praktisch wichtige Probleme im Zusammenhang mit dem Problem eine Menge von Daten zu sortieren sind ungelöst!

3.1 Definitionen und Schreibweisen

Sortierverfahren kann man grob in zwei Klassen einteilen: interne und externe Sortierverfahren. Ein internes Sortierverfahren ist darauf ausgerichtet, eine Datenmenge zu sortieren, welche vollständig in den Arbeitsspeicher passt. Ein externes Sortierverfahren ist darauf spezialisiert, große Datenmengen zu sortieren, welche auf externen Datenträgern gespeichert sind. Es passen also nicht alle Daten gleichzeitig in den Speicher. Es wird berücksichtigt, dass die Datenzugriffe besonders teuer sind.

Wir nehmen an, dass jeder Datensatz a_i einen Schlüssel k_i hat. Dieser Schlüssel wird zum Sortieren benutzt, d. h. wir nehmen an, dass auf den Schlüsseln eine *totale Ordnung* besteht. Für zwei beliebige Schlüssel k_i und k_j können wir also sagen, dass $k_i \leq k_j$ oder $k_j \leq k_i$ gilt. Sei n die Anzahl der zu sortierenden Datensätze. Um eine Anordnung auf den Datensätzen in aufsteigender Folge der totalen Ordnung zu erhalten, benötigen wir also eine Permutation (also eine *bijektive Abbildung*)

$$\pi \ : \ \{1, \ldots, n\} \rightarrow \{1, \ldots, n\},$$

so dass gilt

$$k_{\pi(1)} \leq \cdots \leq k_{\pi(n)}.$$

In dieser Problemformulierung sind absichtlich viele Details offen gelassen. Was heißt es, dass eine Folge von Sätzen „gegeben“ ist: Ist damit gemeint, dass sie in schriftlicher Form oder auf Magnetband, Platte, Diskette oder CDROM vorliegen? Ist die Anzahl der Sätze bekannt? Ist das Spektrum der vorkommenden Schlüssel bekannt ? Welche Operationen sind erlaubt um die Permutation π zu bestimmen? Wie geschieht die „Umordnung“? Sollen die Datensätze physisch bewegt werden oder genügt es eine Information zu berechnen, die das Durchlaufen, insbesondere

die Ausgabe der Datensätze in aufsteigender Reihenfolge erlaubt? Wir können unmöglich alle in der Realität auftretenden Parameter des Sortierproblems berücksichtigen. Vielmehr wollen wir uns auf einige prinzipielle Aspekte des Problems beschränken.

Häufig kann Zusatzinformation den Algorithmus und dessen Geschwindigkeit entscheidend beeinflussen. Solche Zusatzinformationen können sein:

- Art der Abspeicherung der Datensätze (CDROM, Hauptspeicher, Magnetband)
- Eingabe der Datensätze (seriell über tcp/ip ein Datensatz pro Sekunde oder alle Datensätze im Array verfügbar)
- Art der Schlüssel (*m-adisch*, natürliche Zahlen)
- Eigenschaften der Schlüssel (lückenlos, gleichverteilt, Max und Min bekannt, Rechenoperationen erlaubt)
- Eigenschaften der Datensätze (doppelte erlaubt oder nicht)
- Weitere Nutzung der Daten (Ausgabe auf Bildschirm des gesamten Datensatzes oder nur lokal; Bestimmung des mittleren Elements)

Zum Beispiel könnte man Datensätze, die seriell und einzeln ankommen, sofort einsortieren, ohne zu warten bis alle Datensätze vollständig verfügbar sind. Es lohnt sich immer, möglichst viel Information über die Daten im Sortierverfahren mit zu verwenden.

Bemerkung 3.1
Bei der Entwicklung von Algorithmen besteht die Frage, welche Informationen über das Problem helfen, die Aufgabe im Sinne der Anwendung effizient und (bei komplexen Problemen) optimal bzw. „so gut wie möglich“ zu lösen. Man kann nicht generell sagen, dass eine proprietäre, also z.B. problemspezifische Lösung, immer die bessere ist oder eine universelle, also z.B. möglichst generelle Lösung, immer schlechter ist. Ist man sehr speziell kann man evtl. sehr schnell und gut sein, aber im schlechtesten Fall nur ein Problem lösen. Ist man zu universell kann man evtl. alle Probleme lösen (siehe hierzu den GPS: Generell Problem Solver von Herbert Simon und Allen Newel), ist aber zu langsam und zu schlecht. Wichtig ist eine umfassende Behandlung des Problems, zumindest bis die Anwendung mit der Lösung bzgl. Effizienz und Qualität zufrieden ist, da man diese sonst vertrösten müsste. . . ◀

Hier wollen wir auf einige prinzipielle Aspekte des Sortierens eingehen. Dazu machen wir folgende Annahmen:

- der Datensatz hat folgende Form, etwa in Java:

```
public class Datensatz<T> {
    int key;
    T daten;
}
```

- Wir arbeiten mit einem Array von Datensatz:

```
Datensatz<T> a[n]
```

  Wir zählen hierbei von 1 bis n.

- alle Datensätze sind im Hauptspeicher vorhanden
- die Schlüssel sind natürliche Zahlen

- doppelte Datensätze sind erlaubt (also zwei oder mehr Datensätze mit gleichen Schlüsseln)
- das Ergebnis ist ein zusammenhängender Speicherbereich mit aufsteigend sortierten Schlüsseln

Die sortierte Permutation könnte auch in Form eines Arrays $p[i]$ gegeben sein, also

$$\begin{array}{rcl} p : \{1,\ldots,n\} & \to & \{1,\ldots,n\} \\ i & \mapsto & p[i]. \end{array}$$

Dies vermeidet natürlich die möglicherweise aufwendigen Kopiervorgänge von Datensätzen. Nimmt man an, dass die Datensätze in der Form $a[1],\ldots,a[n]$ gegeben sind, dann ist der sortierte Datensatz gegeben durch

$$a[p[1]],\ldots,a[p[n]] \text{ mit } a[p[1]].key \leq \cdots \leq a[p[n]].$$

Zur Analyse der Algorithmen wollen wir aber annehmen, dass die Datensätze tatsächlich kopiert werden. Die Eingabe des unsortierten und Ausgabe des sortierten Datensatzes geschieht also durch

$$a[1],\ldots,a[n].$$

Definition 3.2
Im Programm und in der Analyse verwenden wir leicht unterschiedliche Schreibweisen.

	Programm	Analyse
Datensatz	$a[i]$	a_i
unsortierte Eingabe	$a[1],\ldots,a[n]$	$a_1,\ldots,a_n$
sortierte Ausgabe	$a[1],\ldots,a[n]$	$a_{\pi(1)},\ldots,a_{\pi(n)}$
Schlüssel	$a[i].key < a[j].key$	$k_i < k_j$

◀

Definition 3.3
Wir definieren die Routine `swap(i,j)`, welche den i-ten und j-ten Datensatz vertauscht.

```
private void swap(int i, int j){
  Datensatz<T> tmp;
  tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

◀

Falls Datensätze doppelt vorkommen (also mit gleichem Schlüssel) macht folgende Definition Sinn.

Definition 3.4
Ein Sortierverfahren heißt *stabil*, wenn Objekte mit gleichem Schlüssel ihre relative Position nicht verändern, wenn also gilt

$$\text{ist } i < j \text{ und } k_i = k_j \text{ so gilt } \pi(i) < \pi(j).$$

◀

Definition 3.5
Wir führen noch folgende Schreibweise ein, die wir in der Analyse der Sortierverfahren benötigen.

1. Die Anzahl der zu sortierenden Elemente bezeichnen wir als Problemgröße und mit dem Buchstaben n.

2. Mit $C_{min}(n)$, $C_{max}(n)$, $C_{avg}(n)$ bezeichnen wir die Anzahl der Schlüsselvergleiche (comparisons) des Sortierverfahrens. Dabei unterscheiden wir den besten Fall (best case: min), den schlechtesten Fall (worst case: max), und einen durchschnittlichen Fall (average case: avg).

3. Mit $M_{min}(n)$, $M_{max}(n)$, $M_{avg}(n)$ bezeichnen wir die Anzahl der Bewegungen (movements). Auch hier wird wieder nach best case, worst case und average case getrennt.

Der average case wird üblicherweise bezogen auf alle $n!$ möglichen Ausgangsordnungen der Datensätze. ◄

3.2 Elementare Sortierverfahren

In diesem Abschnitt besprechen wir einfache interne Sortierverfahren. Für diese Verfahren ist typisch, dass $\Theta(n^2)$ Vergleichsoperationen von Schlüsseln im schlechtesten Fall ausgeführt werden müssen.

Sortieren durch Auswahl (Selection Sort)

Methode. Nehmen wir als Beispiel das Kartenspiel. Das Prinzip von Selection Sort besteht darin, die Karte mit dem kleinsten Wert zu finden, und diese mit der ersten Karte zu tauschen. Beim Kartenspiel würde man die kleinste Karte einfach vorne rein stecken. Algorithmisch bedeutet dies aber, dass die folgenden Karten (bis zu der Stelle an der die kleinste Karte war) eine Stelle weiter rutschen. Da dies viele Bewegungen erfordert, tauschen wir die erste Karte mit der kleinsten. Dies erfordert drei Bewegungen (da ja zum Vertauschen ein Hilfsspeicher benötigt wird).

Nun muss das Kartenspiel ab der zweiten Karte sortiert werden. Dabei verfahren wir genauso. Wir suchen die kleinste Karte ab der zweiten Karte und tauschen sie mit der zweiten Karte. Danach muss das Kartenspiel ab der dritten Karte sortiert werden.

Beispiel Die unsortierten Karten mögen aussehen wie folgt:

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8

Der Einfachheit halber alles Karo und nur Zahlen.

Das kleinste Element steht an Position 5. Wir vertauschen also Position 5 und Position 1 miteinander und erhalten

i:	1	2	3	4	5	6	7	8
a_i:	♢2	♢6	♢9	♢3	♢7	♢4	♢5	♢8

Nun betrachten wir die Karten ab Position 2. Das kleinste verbleibende Element steht an Position 4. Also werden Position 4 und 2 miteinander vertauscht.

i:	1	2	3	4	5	6	7	8
a_i:	♢2	♢3	♢9	♢6	♢7	♢4	♢5	♢8

Im nächsten Schritt werden Positionen 3 und 6 vertauscht:

i:	1	2	3	4	5	6	7	8
a_i:	♢2	♢3	♢4	♢6	♢7	♢9	♢5	♢8

Danach Positionen 4 und 7:

i:	1	2	3	4	5	6	7	8
a_i:	♢2	♢3	♢4	♢5	♢7	♢9	♢6	♢8

Folgendes Programm realisiert diesen Algorithmus. Dabei ist unterstellt, dass das Array `a` etwa ein (Klassen-) Attribut der Klasse ist, zu der die Operationen gehören.

```
public void selectionSort(){
   int minimum = 0;
   for(int i = 1; i<= n-1;i++){
       minimum = findMinStartingAt(i);
       swap(i,minimum);
   }
}
public int findMinStartingAt(int i){
   int min  = i;
   for(int j = i+1;j<=n;j++){
       if(a[j].key < a[min].key)
          min = j;
   }
   return min;
}
```

Analyse. In der Routine `findMinStartingAt(int)` werden Schlüsselvergleiche gemacht. Die entsprechende Zeile wird $n - i$ mal aufgerufen. Die Routine `findMinStartingAt(int)` wird zunächst mit $i = 1$, dann mit $i = 2$ usw., und schließlich mit $i = n - 1$ aufgerufen. Die Anzahl der Schlüsselvergleiche ist unabhängig von der Ausgangsordnung der Datensätze. Wir können also schreiben:

$$C_{min}(n) = C_{max}(n) = C_{avg}(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2).$$

Die Routine `swap(int,int)` wird $n-1$ mal aufgerufen. In jedem Aufruf haben wir drei Bewegungen. Damit ergibt sich

$$M_{min}(n) = M_{max}(n) = M_{avg}(n) = 3(n-1) = \Theta(n).$$

Insgesamt, mit allen Bewegungen und Vergleichen, erhält man also eine Aufwandsordnung im best case, worst case und average case von

$$\Theta(n^2) + \Theta(n) = \Theta(n^2).$$

Kann man den Algorithmus verbessern, indem man die Minimumsuche verbessert?

Satz 3.6
Jeder Algorithmus zur Bestimmung des Minimums von n Schlüsseln, der allein auf Schlüsselvergleichen basiert, muss mindestens $n - 1$ Schlüsselvergleiche ausführen. ◀

Beweis: Angenommen man käme mit weniger Vergleichen aus. Dann gibt es Datensätze, mit denen kein Vergleich durchgeführt wurde. Solch ein Datensatz könnte aber gerade der Kleinste sein. Es gibt keine Möglichkeit außer Schlüsselvergleichen, um dies festzustellen. Also muss jeder Datensatz in mindestens einen Vergleich eingebunden sein. □

Wir halten fest, dass bei Selection Sort $\Theta(n^2)$ Vergleichsoperationen durchgeführt werden und $\Theta(n)$ Bewegungen. Sind Bewegung relativ teuer (weil z.B. von einem externen Medium gelesen werden muss), so kann Selection Sort besser sein als ein Verfahren, welches weniger Vergleichsoperationen benötigt, da Selection Sort nur linear viele Bewegungen benötigt.

Sortieren durch Einfügen (Insertion Sort)

Methode Die Datensätze werden der Reihe nach durchgegangen, und in die sortierte, anfangs leere Teilfolge an der richtigen Stelle einsortiert. Nehmen wir als Beispiel wieder ein Kartenspiel. Die Karten liegen verdeckt auf dem Tisch und man nimmt eine Karte nach der anderen auf. Das Einfügen funktioniert dann so: Angenommen man hat Karten $1, \ldots, i-1$ sortiert in der Hand. Man

nimmt die i-te Karte, und vergleicht sie der Reihe nach mit der ersten, zweiten, dritten usw Karte in der Hand. Sobald die Karte in der Hand einen größeren Kartenwert hat als die aufgenommene i-te Karte, fügt man die i-te Karte davor ein.

Beispiel Die unsortierten Karten mögen aussehen wie folgt:

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8

Als erstes nimmt man die Karte 1 vom Tisch auf. Es ist eine ♢7. Als zweites nimmt man ein ♢6 auf. Man vergleicht es mit der in der Hand befindlichen ♢7. Da ♢6 kleiner ist, sortiert man es davor ein und erhält

i:	1	2	3	4	5	6	7	8
a_i:	♢6	♢7	♢9	♢3	♢2	♢4	♢5	♢8

Als nächstes nimmt man Karte 3 auf: ♢9. Man vergleicht sie mit Karte 1 und Karte 2 und sortiert sie hinter ♢7 ein.

i:	1	2	3	4	5	6	7	8
a_i:	♢6	♢7	♢9	♢3	♢2	♢4	♢5	♢8

Dann nimmt man Karte 4 auf: ♢3. Beim Vergleich mit den in der Hand befindlichen Karten bekommt man schon gleich beim ersten Vergleich die Einfügestelle:

i:	1	2	3	4	5	6	7	8
a_i:	♢3	♢6	♢7	♢9	♢2	♢4	♢5	♢8

Die in der Hand befindlichen Karten rutschen eine Stelle weiter: 3 Verschiebungen. Als nächstes haben wir

i:	1	2	3	4	5	6	7	8
a_i:	♢2	♢3	♢6	♢7	♢9	♢4	♢5	♢8

Das sind 4 Verschiebungen.

i:	1	2	3	4	5	6	7	8
a_i:	♢2	♢3	♢4	♢6	♢7	♢9	♢5	♢8

Das sind 3 Verschiebungen.

Folgendes Programm realisiert diesen Algorithmus.

```
public void insertionSort(){
    for ( int i = 2; i <= n ; i++ ){
        int j = i;
        Datensatz<T> t = a[i];
        int k = t.key;
        while(a[j-1].key > k)
        {
          a[j] = a[j-1];
          j = j-1;
        }
        a[j] = t;
    }
}
```

Die while-Schleife in dieser Routine terminiert dann nicht richtig, wenn der i-te Datensatz einen kleineren Schlüssel hat als alle Datensätze $a_1, \ldots, a_{i-1}$. Damit die While-Schleife richtig terminiert nehmen wir an, dass wir das linke Element $a[0]$ mit einem Stopper belegt haben. Dies ist ein Dummy-Datensatz, dessen Schlüssel kleiner ist als alle existierenden Schlüssel. Wir haben damit erreicht, dass in der While-Schleife nur eine Abfrage statt zweien vorkommt (vgl. Suche in Listen).

Analyse. Zum Einfügen eines Datensatzes benötigen wir mindestens ein und höchstens i Schlüsselvergleiche. Außerdem werden zwei oder höchstens $i+1$ Bewegungen ausgeführt. Das Ganze wird in einer `for`-Schleife $n-1$-mal durchlaufen. Wir erhalten also

$$C_{min}(n) = n-1; \qquad C_{max}(n) = \sum_{i=2}^{n} i = \Theta(n^2)$$

$$M_{min}(n) = 2(n-1); \qquad M_{max}(n) = \sum_{i=2}^{n} (i+1) = \Theta(n^2).$$

Wie sieht es nun mit den durchschnittlichen Anzahlen aus: C_{avg} und M_{avg}? Dazu nehmen wir an, dass alle $n!$ Permutationen des Eingabe-Arrays gleich wahrscheinlich sind. Sind dann auch noch alle Schlüsselwerte verschieden, so benötigen wir nach den vorstehenden Überlegungen gerade den Mittelwert der Extremwerte:

$$\begin{aligned} C_{avg}(n) &= \Theta(n^2) \\ M_{avg}(n) &= \Theta(n^2) \end{aligned}$$

Shellsort

Methode. Shellsort ist ein Verfahren, welches sich „Sortieren durch Einfügen (Insertion Sort)“ zu Hilfe nimmt, dieses jedoch auf bestimmte Teilfolgen anwendet. Diese Teilfolgen bestehen aus den Elementen

$$a[1], a[1+h], a[1+2h], a[1+3h], \ldots$$
$$a[2], a[2+h], a[2+2h], a[2+3h], \ldots$$
$$a[3], a[3+h], a[3+2h], a[3+3h], \ldots$$
$$\text{usw.}$$

Dabei ist h auf bestimmte Weise gewählt. Die Idee ist, dass dadurch große Sprünge schneller geschafft werden. Ein Datensatz mit kleinem Schlüssel, der am falschen Ende steht, muss also nicht einschrittweise umkopiert werden bis er an der richtigen Stelle steht. Abbildung 3.1 zeigt dies am Beispiel der Folge (7,5,8,1,5,2,6,4). Im ersten Schritt wird diese Folge 4-sortiert. Anschließend wird sie 3-sortiert und dann 2-sortiert. Im letzten Schritt braucht man nur noch bei Bedarf benachbarte Elemente zu vertauschen, um die Sortierung herzustellen.

Bemerkung 3.7
Die Idee bezieht sich nicht nur auf die großen Schritte, sondern auch auf die Art und Weise, in der diese Schritte kleiner gemacht werden: wie oft wird man bei der Verkleinerung der Schrittgrösse die gleichen Elemente nochmal sortieren? Wie sieht also die Verteilung der Zugriffe auf die einzelnen Elemente aus? Und welche Verteilung ist gut bzw. optimal? ◄

Sind alle der oben aufgeführten Folgen für ein bestimmtes h sortiert, so sagt man die Folge ist h-sortiert. Leider ist die gesamte Folge dann noch nicht sortiert. Man wählt dann ein kleineres h und erhält neue Folgen die man wieder sortiert. Man fährt solange fort, bis $h = 1$ ist. Diese Werte h bilden dann eine sog. Folge von abnehmenden Inkrementen

$$h_t \geq h_{t-1} \geq h_{t-2} \geq \cdots \geq h_1 \text{ wobei } h_1 = 1.$$

Es werden also der Reihe nach eine h_t-sortierte Folge hergestellt, danach h_{t-1}-sortierte Folge, usw. bis wir eine 1-sortierte Folge haben, welche also sortiert im gewöhnlichen Sinn ist.

Die Frage stellt sich nun, welche h-Folge man wählen soll. Hier gibt es eine Reihe Antworten, die jedoch nur unvollständig sind. Verschiedene interessante Einzelergebnisse lassen sich in diesem Zusammenhang beweisen.

Dieses Verfahren wurde von D. L. Shell entwickelt [She59]. Man nennt es auch *Sortieren mit abnehmenden Inkrementen*.

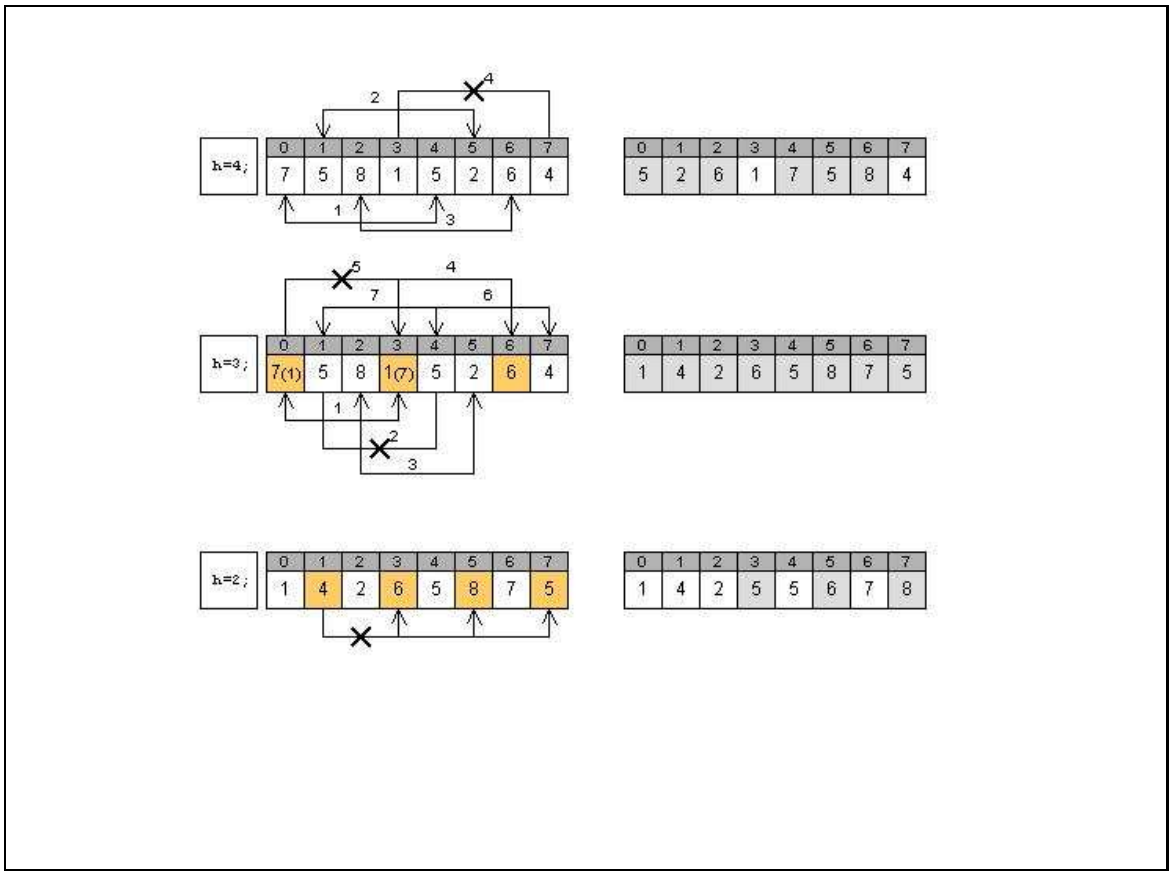


Abbildung 3.1: h-Sort für h = 4, 3, 2

Beispiel 3.8
Wir betrachten wieder unser Kartenspiel. Als h-Folge wählen wir

$$5, 3, 1.$$

Die ursprüngliche Ordnung sei wieder wie folgt:

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8

Die 5-Folgen bestehen aus den Datensätzen an den Positionen

$$1, 6 \text{ und } 2, 7 \text{ und } 3, 8.$$

Wir vertauschen 1 und 6, 2 und 7, und 3 und 8 und erhalten

i:	1	2	3	4	5	6	7	8
a_i:	♢4	♢5	♢8	♢3	♢2	♢7	♢6	♢9

Diese Folge ist nun 5-sortiert. Nun wollen wir sie 3-sortieren. Die 3-Folgen sind

$$1, 4, 7 \text{ und } 2, 5, 8 \text{ und } 3, 6.$$

Wir vertauschen 1 und 4 (7 ist schon richtig), 2 und 5 und 3 und 6.

i:	1	2	3	4	5	6	7	8
a_i:	♢3	♢2	♢7	♢4	♢5	♢8	♢6	♢9

Diese Folge muss nun noch durch gewöhnliches Sortieren durch Einfügen sortiert werden. Wir haben mit 6 Vertauschungen erreicht, dass einige der kleinen Karten links stehen und die hohen Karten im Prinzip rechts. Unter Umständen hat es Insertion Sort jetzt leichter. ◀

Der folgende Code implementiert die h-Sortierung:

```
public int [] hSort( int [] a, int h ) {
     for( int actIndex = h; actIndex < a.length; actIndex++ ) {
         int moveIndex = actIndex - h;
         int tmp = a[ actIndex ];
         ...
         // platz schaffen
         while( (moveIndex >= 0) && (a[ moveIndex ] > tmp) ) {
             a[moveIndex + h] = a[moveIndex];
             moveIndex -= h;
         }
         // einfügen.
         a[ moveIndex + h ] = tmp;
     }
     return a;
 }
```

Die Idee: man sucht eine geeignete Folge für h, z.B.: $\ldots, 16, 8, 4, 2, 1$ (erstes Element je nach Grösse der zu sortierenden Liste), und führt durch sukzessives Anwenden des h-Sort Verfahrens mit diesen Werten für h die gesamte Sortierung durch. Dieses Verfahren wurde zuerst von Donald Shell in [She59] beschrieben.

Beispiele für solche Folgen sind:

Shell 1, 2, 4, 8, 16, ... (das Original von Shell aus [She59])

Knuth 1, 4, 13, 40, ... ([Knu73b] Komplexität etwa $\mathcal{O}(n^{\frac{3}{2}})$ Berechnung: $h_{i+1} = 3 * h_i + 1$

```
int h;
for(h=1; h < (start-end)/9; h = 3*h+1);
for( ; h > 0; h /= 3 )
{
     hSort ... //h=40,13,4,1
}
```

Sedgewick 1, 8, 23, 77, 281, 1073, 4193, 16577, ... Komplexität etwa $\mathcal{O}(n^{\frac{4}{3}})$ Berechnung: $h_i = 4^{i+1} + 3 * 2^i + 1$

Pratt 27, 18, 12, 8, 9, 6, 4, 3, 2, 1 Komplexität etwa $\mathcal{O}(n(logn)^2)$

Satz 3.9 (Vererbung der H-Sortierung)
Eine Folge $(a_i), i = 1, \ldots n$ sei h-sortiert. Wird sie nun k-sortiert, so ist sie anschließen h- und k-sortiert ◀

Insbesondere gilt: Ist eine Folge 3-sortiert und 2-sortiert, so müssen zum Abschließen der Sortierung nur noch benachbarte Elemente gegeneinander vertauscht werden. Dementsprechend erscheint es logisch, dass bei der h-Sortierung einer Liste, die $2 * h$- und $3 * h$-sortiert ist, ebenfalls nur „benachbarte“ Elemente ($Abstand = h$) gegeneinander vertauscht werden müssen. Daraus ergibt sich eine weitere Folge für h, die von Pratt, s. o., die mit einer für dieses Verfahren idealen Anzahl von Vertauschungsoperationen auskommt. Sie ergibt sich so ähnlich wie das bekannte Pascalsche Dreieck, nur das jetzt nach links unten mit 2 und nach recht unten mit drei multipliziert wird

					1					
				2		3				
			4		6		9			
		8		12		18		27		
	16		24		36		54		81	
					...					

Ordnet man die Matrix etwas anders an, so erkennt man das einfache Bildungsgesetz leichter:

		$\longrightarrow *3$						
			1	2	3	4	5	...
	0	1	3	9	27	81	...	
	1	2	6	18	54	...		
	2	4	12	36	...			
$*2$	3	8	24	...				
	4	16	...					

Sie ergibt sich aus dieser Dreiecksmatrix von Elementen zu $h_{ij} = 2^i * 3^j$. In dem Ausgangsdreick werden die Zahlen von rechts unten nach links oben verwendet.

Auf die genauere Untersuchung des Aufwands von Shellsort seien Sie auf [Sed92, OW02, Knu73b] verwiesen. Die Quelle [Sed92] gebe ich an, weil Sie bei mir im Regal steht, es gibt neuere Bücher z. B. [Sed02] in denen sich die Informationen wahrscheinlich auch finden lassen.

Bubblesort

Methode. Hier nimmt man an, dass nur Bewegungen (swap) zwischen benachbarten Datensätzen vorgenommen werden dürfen. Man durchläuft die Liste $a_1, \ldots, a_n$ der Datensätze von 1 bis n und vergleicht zwei nebeneinander liegende Datensätze. Diese bringt man jeweils durch Vertauschen in die richtige Reihenfolge. Nach dem ersten Durchlauf ist offenbar der Datensatz mit dem größten Schlüssel oben angelangt (man schleppt ihn immer mit). Dann geht man die Folge erneut durch.

Da der größte Datensatz schon oben angekommen ist, muss man nur noch bis $n-1$ gehen. Wenn keine Vertauschen mehr vorgenommen werden müssen, ist die Folge sortiert.

Beispiel Im Kartenspiel haben wir wieder

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8

Der erste Durchlauf geht von 1 bis 8 und man erhält folgendes:

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8
:	♢6	♢7	♢9	♢3	♢2	♢4	♢5	♢8
:	♢6	♢7	♢3	♢9	♢2	♢4	♢5	♢8
:	♢6	♢7	♢3	♢2	♢9	♢4	♢5	♢8
:	♢6	♢7	♢3	♢2	♢4	♢9	♢5	♢8
:	♢6	♢7	♢3	♢2	♢4	♢5	♢9	♢8
:	♢6	♢7	♢3	♢2	♢4	♢5	♢8	♢9

Man sieht hier sehr schön, wie der ♢9 nach oben blubbert. Daher auch der Name des Verfahrens.

Analyse. Wir verzichten auf eine detaillierte Analyse. Man kann zeigen, dass Bubblesort folgenden Aufwand hat

$$C_{max}(n) = \Theta(n^2)$$
$$M_{max}(n) = \Theta(n^2).$$

Es gilt auch

$$C_{avg}(n) = M_{avg}(n) = \Theta(n^2).$$

Damit ist Bubblesort ein schlechtes elementares Sortierverfahren. Einige in der Literatur vorgeschlagenen Verbesserungen bringen keine nennenswerten Vorteile. Bei Shakersort wird zum Beispiel die Folge abwechselnd von links und von rechts durchlaufen. Dies soll den Nachteil beheben, dass kleine Elemente nur sehr schwer von rechts nach links kommen.

3.3 Quicksort

Quicksort wurde 1962 von C. A. R. Hoare veröffentlicht [Hoa62]. Die Grundidee besteht darin, die zu sortierende Folge von Datensätzen in zwei Folgen aufzuteilen, und dann jede für sich zu sortieren. Dieses Verfahren kann man wiederholt anwenden. Abbildung 3.2 zeigt, was man dadurch möglicherweise erreichen kann: Die zu sortierende Folge F wird möglichst gleichmäßig in zwei Teilfolgen F_1 und F_2 aufgeteilt, dass gilt: $f_i \leq f_j \ \forall \ f_i \in F_1$ *und* $f_j \in F_2$. Das gleiche Verfahren wird nun auf die Teilfolgen angewendet, bis man Teilfolgen von jeweils einem Element hat. Im Ergebnis ist die Folge dann sortiert.

Aufgrund unserer bisherigen Erfahrungen können wir hoffen, dass das Zerlegen in zwei Teilfolgen einen $\mathcal{O}(n)$ Aufwand benötigt. Gelingt es, immer eine Aufteilung in zwei gleichgroße Teilfolgen zu erreichen, so benötigen wir dafür $log_2\ n$ Schritte. Insgesamt können wir also auf ein Verhalten mit $\mathcal{O}(n \cdot log_2 n)$ hoffen. Wir werden sehen, dass dieses rekursive Verfahren sich tatsächlich im Wesentlichen so verhält. Der schlechteste Fall ist allerdings deutlich schlechter.

Das bei Quicksort verwendete Verfahren ist ein Beispiel einer *Divide-and-Conquer*-Strategie (*teile und herrsche*), eine Strategie, die bereits in der Renaissance als *divide et impera* formuliert wurde und urspünglich dem französischen König Ludwig XI. (1423-1483) zugeschrieben wird.

Methode Die Aufteilung in zwei Teilfolgen geschieht so, dass beide sortierten Teilfolgen ohne weiteres Sortieren aneinander gesetzt werden können und dann eine ganze sortierte Folge ergeben. Dafür nimmt man einen Schlüsselwert k und teilt die Datensätze auf in zwei Teilfolgen F_1 und F_2 wovon F_1 nur Datensätze mit Schlüsseln kleiner als k hat, die andere Teilfolge F_2 enthält Datensätze mit Schlüsseln größer als k. Am liebsten hätte man beide Teilfolgen gleich groß. Wir wollen annehmen, dass wir den Schlüssel k eines bestimmten Datensatzes a_k zur Aufteilung in

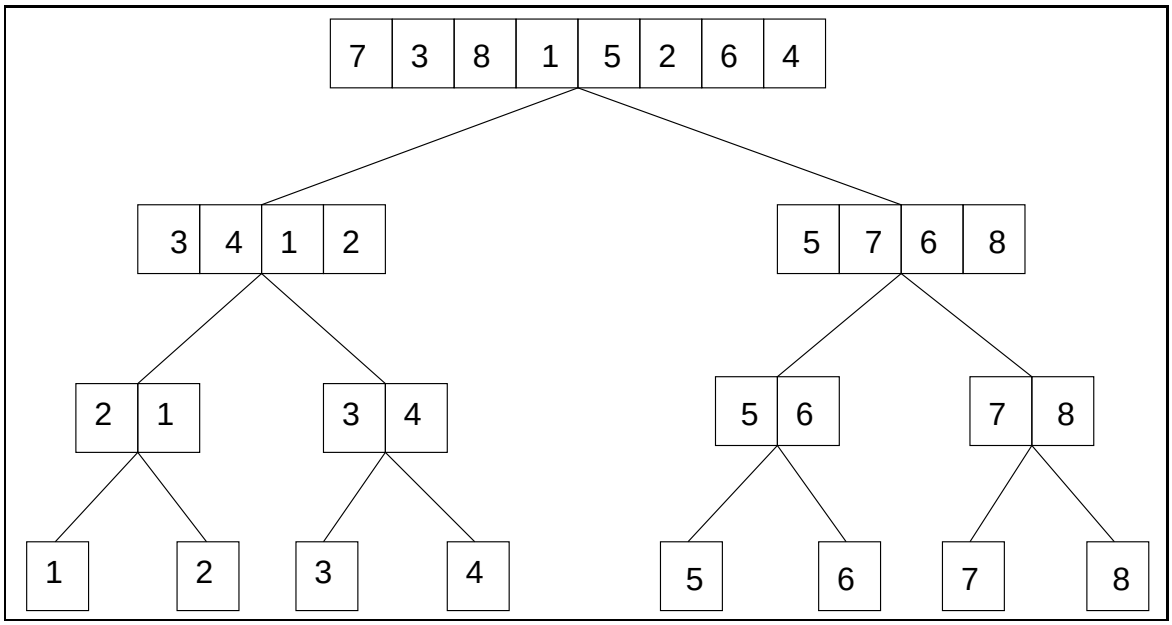


Abbildung 3.2: Grundprinzip von Quicksort

Teilfolgen verwenden. Dieses a_k nennen wir *Pivotelement*. F_1 und F_2 werden selbst wieder mit Quicksort sortiert, also für sich in je zwei Teilfolgen aufgeteilt.

Die Zusammensetzung geschieht dann wie folgt:

$$\text{gesamte sortierte Folge F } = (F_1, a_k, F_2)\,.$$

Die Rekursion verkleinert die Folgen bis sie einen oder keinen Datensatz mehr enthalten.

Optimal für die Aufteilung der Ausgangsfolge wäre eine Trennung am *Median*.

Definition 3.10 (Median)
Sei $a_1, \ldots, a_n$ eine Folge von Objekten mit vollständiger Ordnung vom Typ T. Ein Objekt z für das gilt: ($P : Probability$, $Wahrscheinlichkeit$)

$$\begin{aligned} P(\{i|a_i \leq z\}) &\leq 0.5 \; und \\ P(\{i|a_i \geq z\}) &\leq 0.5 \end{aligned}$$

heißt *Median* dieser Folge. ◀

Speziell für den hier betrachteten Fall heißt das: Die Hälfte der Datensätze hat einen Schlüsselwert kleiner oder gleich dem Median und die andere Hälfte einen Schlüsselwert größer oder gleich dem Median. Wir bestimmen einen Wert *median*, der dem *Median* der $a_i, i = 1, \ldots, n$ möglichst nahe kommt. Dann suchen wir l, r mit folgenden Eigenschaften:

1. $1 \leq l \leq r \leq n$
2. $a_i \leq \; median \; \forall \; i < r \; \wedge \; a_i \; \geq \; median \; \forall \; i \; > l$

Dabei verwenden wir das Konzept der *Schleifeninvarianten*. Dieses Prinzip lautet allgemein formuliert so:

Gegeben Sie eine Bedingung P und eine Bedingung Q Wir starten mit einer Situation, in der gilt $P = true$ Dann versuchen wir innerhalb einer Schleife diese Eigenschaft zu erhalten und zusätzlich $Q = true$ zu erreichen:

```
P = true
loop while Q == false
  erhalte P == true
  versuche  Q == true zu erreichen
```

Verlassen wir die Schleife, so gilt $(P \wedge Q) = true$.

In diesem Fall verwenden wir die Bedingung 2 als Schleifeninvariante und versuchen die Bedingung 1 innerhalb einer Schleife zu realisiersen. Damit lässt sich der Quicksort-Algorithmus Pseudo-Codeartig so beschreiben:

```
quicksort(a:Array, start:int, ende:int)
{
//Bestimme l und r wie oben angegeben
// Sortiere die Partitionen
   quicksort(a, start,l);
   quicksort(a,r,ende);
}
```

Um die Bestimmung von l und r zu starten machen wir uns eine elementare Sache zu nutze:

Ist $A(x)$ irgend eine Aussage die für $x \in X$ wahr oder falsch sein kann, so ist die Aussage „A(x) ist $\forall x \in X$ wahr“, bestimmt wahr, wenn X die leere Menge $\emptyset$ ist.

Setzen wir zunächst

$$\begin{aligned} r &:= start \\ l &:= ende \end{aligned}$$

so ist zwar die Bedingung

$$l \leq r$$

verletzt, es gilt aber — sogar unabhängig davon, wie wir den Wert *median* bestimmt haben —

$$\begin{aligned} a_i &\leq median \; \forall \; i < r \\ a_i &\geq median \; \forall \; i > l \end{aligned}$$

Wie können wir nun die Bedingung 1, $l \leq r$ sicherstellen? Dazu schieben wir r nach rechts und l nach links. Allerdings müssen wir dabei sicherstellen, das die Bedingung 1 (die mit dem *median*) nicht verletzt wird. Das erreichen wir so:

```
1  while (a[r] < median)
2     r++;
3  while (a[l] > median)
4     l--;
5  if (r <= l)
6     swap(l,r);
```

In Zeile 1 prüfen wir ob die Bedingung $a_i < median$ für $i = r$ erfüllt ist. in diesem Fall können wir r mindestens um eine Position nach rechts schieben. Ganz analog können wir l um eine Position nach links schieben, wenn die Bedingung in Zeile 3 erfüllt ist. Beides machen wir so lange, wie „es gut geht“. Sind beide Schleifen verlassen, so prüfen wir in Zeile 5, ob unsere Bedingung 1 bereits erfüllt ist oder nicht. Ist sie noch nicht erfüllt, d. h. $r <= l$, so gilt wegen der Bedingung in Zeile $1 : a_r \geq \; median$ und wegen der in Zeile $3 : a_l \leq \; median$. Vertauschen wir die beiden, so sind wir unserem Ziel, die Bedingung 1 sicherzustellen, einen Schritt näher gekommen.

Nun brauchen wir nur noch die äußere Schleife:

```
while (r < l)
   while (a[r] < median)
      r++;
   while (a[l] > median)
      l--;
   if (r <= l)
      swap(a[l],a[r]);
```

Wir gut das funktioniert hängt stark von der Wahl des Wertes *median* ab. Bewährte Strategien sind:

- Wähle den ersten, mittleren oder den letzten Wert
- Wähle den Median des ersten, mittleren und letzten Elements
- Wähle einen zufälligen Indexwert, und nehme den Wert an dieser Position

Führen wir das doch mal am obigen Beispiel aus Abb. 3.2 einfach durch. Wir bilden den Median als Mittelwert von

$$\begin{aligned} median &= \frac{a_1 + a_4 + a_8}{3} \\ &= \frac{7+1+4}{3} \\ &= \frac{12}{3} \\ &= 4 \end{aligned}$$

Die erste Partitionierung zeigt Abb. 3.3

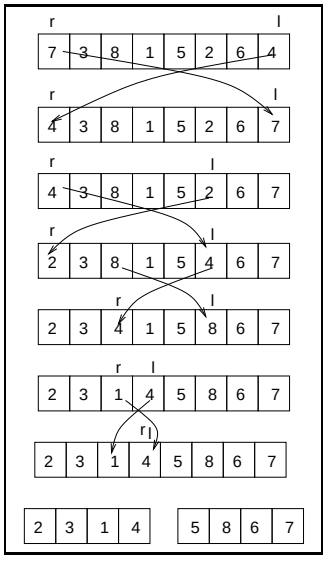


Abbildung 3.3: Erste Partitionierung

Den Algorithmus kann man etwa so formulieren:
Algorithmus Quicksort (Folge F) Falls F einen oder keinen Datensatz hat tue nichts.
sonst
Divide Wähle ein Pivotelement p(z. B. das letzte Folgenelement).
Teile F auf in zwei Folgen F_1 und F_2, so dass

$$\forall a \in F_1 \ : \ a.key \leq p.key \text{ und } \forall b \in F_2 \ : \ p.key < b.key.$$

Conquer Quicksort (F_1) : Quicksort (F_2) (an dieser Stelle sind F_1 und F_2 sortiert)
Merge bilde zusammengesetzte Folge

$$F = (F_1, p, F_2)\,.$$

Der Knackpunkt im Verfahren ist die Aufteilung der ursprünglichen Folge F in zwei Teilfolgen F_1 und F_2 mit den geforderten Eigenschaften bzgl. eines Pivotelements p.

Wir wollen annehmen, dass

$$p = a_n,$$

also der am rechten Ende stehende Datensatz ist. Wir wollen F_1 und F_2 erzeugen, ohne ein neues Array für Datensätze anlegen zu müssen. Hierfür folgende

Definition 3.11 (In-situ Sortierverfahren)
Ein *In-situ*[1]-Sortierverfahren benötigt für die Zwischenspeicherung von Datensätzen keinen zusätzlichen Speicher, außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen. ◀

Wir wollen also F_1 und F_2 In-situ erzeugen. Hierfür also die
Methode zur Aufteilung in Teilfolgen mit Pivotelement p Sei $a_l, \ldots, a_n$ die Teilfolge die aufgeteilt werden muss, und sei $p = a_n$ das Pivotelement. Dann laufe man von links die Datensätze durch, bis ein Datensatz kommt, dessen Schlüssel größer ist als der des Pivotelements. Gleichzeitig laufe man von rechts durch die Datensätze durch, bis man auf einen Datensatz stößt, dessen Schlüssel kleiner ist als der des Pivotelements. Diese Datensätze vertausche man. Stoßen die Iteratoren von links und rechts zusammen, dann ist man fertig.
Beispiel Am Beispiel Kartenspiel sieht das aus wie folgt

i:	1	2	3	4	5	6	7	8
a_i:	♢8	♢6	♢9	♢3	♢2	♢4	♢5	♢7

Wir wählen ♢7 als Pivotelement. Schon gleich der erste Datensatz an Position 1 ist größer als das Pivotelement, der Datensatz an Position 7 ist kleiner. Diese beiden werden vertauscht.

i:	1	2	3	4	5	6	7	8
a_i:	♢5	♢6	♢9	♢3	♢2	♢4	♢8	♢7

Weiter gehts von links mit Position 2. Dieser Datensatz ist nun schon kleiner als das Pivotelement. Dann kommt Position 3. Dieser Datensatz ist größer als das Pivotelement. Von rechts kommt Position 6 welches kleiner ist als das Pivotelement, also werden 3 und 6 vertauscht.

i:	1	2	3	4	5	6	7	8
a_i:	♢5	♢6	♢4	♢3	♢2	♢9	♢8	♢7

Von links kommend sind nun Positionen 4 und 5 schon kleiner als das Pivotelement. Damit stoßen die Iteratoren von links und rechts zusammen. Wir bringen nun noch das Pivotelement an seine endgültige Position indem wir Positionen 6 und 8 vertauschen

i:	1	2	3	4	5	6	7	8
a_i:	♢5	♢6	♢4	♢3	♢2	♢7	♢8	♢9

Die Teilfolgen sind also

$$F_1 = \{a_1, \ldots, a_5\} \text{ und } F_2 = \{a_7, a_8\}.$$

Im folgenden das Programm dafür. `a` ist dabei ein Array von `Datensatz<T>` der Länge `n+1`.

[1] In-situ bedeutet „an Ort und Stelle“ bzw. im übertragenen Sinn „an der gegebenen anatomischen Position“ oder „in der richtigen anatomischen Lage“. *Ex-situ* bedeutet „nicht an Ort und Stelle“ bzw. im übertragenen Sinn „außerhalb der natürlichen anatomischen Lage“.

```
void quicksort(int ilinks, int irechts)
{
     int pivot,i,j;
     Datensatz<T> tmp;
     if ( irechts > ilinks )
     {
          i = ilinks;
          j = irechts-1;
          pivot = a[irechts].key;
          while(1)
          {
               while(a[i].key < pivot) i++;
               while(a[j].key >= pivot)
                    j--;//Vorsicht: Stop-Element einbauen
               if ( i >= j ) break;//in der Mitte getroffen
               swap(i,j);//vertauschen
          }
          swap(i,irechts);//Pivotelement in die Mitte tauschen
          quicksort(ilinks,i-1);
          quicksort(i+1,irechts);
     }
}
```

Man beachte, dass die erste while-Schleife sicher terminiert. Die zweite while-Schleife terminiert dann nicht, wenn das Pivotelement das kleinste Element der Teilfolge ist. Es muss für die gesamte Folge ein linkes Stopelement eingefügt werden. Dann hat man auch für jede Teilfolge ein solches.

Die Abbruchbedingung in den inneren while-Schleifen ist so gestaltet, dass auch Datensätze mit mehreren gleichen Schlüsseln vorkommen können. Allerdings werden hierbei unnötige Vertauschungen vorgenommen. Das Verfahren ist nicht stabil.

Analyse. Wir fangen mit dem *best case* an. Idealerweise werden dabei die Listen immer in zwei gleich große Teillisten aufgeteilt. Wir nehmen an, dass der Aufwand für die Sortierung einer Teilliste $T(n)$ beträgt. Dabei soll T die Aufwandsfunktion sein, die die Summe der Bewegungen und Vergleiche darstellt. Wird eine Teilliste wieder in zwei gleich großte Teillisten aufgeteilt, so benötigt man zur Aufteilung selbst $\mathcal{O}(n)$ Bewegungen und Vergleiche. Um diese halb so großen Teillisten zu sortieren, wird jeweils ein Aufwand

$$T\left(\frac{n}{2}\right)$$

benötigt. Insgesamt erhält man so eine Rekursionsformel der Form

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n. \end{aligned}$$

Setzt man diese Rekursionsformel in ein CAS ein (z.B. Maple) so erhält man als geschlossene Darstellung für T

$$T(n) = c \cdot n \cdot \log_2 n = \mathcal{O}(n \log_2 n).$$

Auch aus Satz A.25 ergibt sich dies durch einfaches Einsetzen: $a = 2, c = c, b = c, m = 1$. Damit hat man dann $T(n) = \Theta(n \log_2 n)$. Dieses Ergebnis kann man auch einsehen, wenn man sich den Aufrufbaum mit den Teilfolgen klar macht. Jede Teilfolge wird in zwei gleich große Teilfolgen aufgeteilt, welche wiederum in zwei gleich große Teilfolgen aufgeteilt werden. Man erhält einen Binärbaum mit minimaler Höhe. Die Höhe des Binärbaumes ist $\log_2 n$. In jeder Ebene müssen $c \cdot n$ Operationen (Bewegungen und Vergleiche) durchgeführt werden. Also hat man $c \cdot n \cdot \log_2 n$.

Im *worst case*, also im schlechtesten Fall, werden die Teilfolgen ganz ungleich aufgeteilt. Eine Teilfolge ist leer, die andere enthält alle Elemente bis auf das Pivotelement. Dieser Fall tritt auf,

wenn die Folge schon fertig sortiert ist und als Pivotelement immer das ganz rechts (also das größte Element) genommen wird. Dies ist besonders bitter, denn die Folge ist ja schon fertig sortiert und es gäbe eigentlich gar nichts zu tun.

Der Binärbaum hat dann maximale Höhe, also n. Auf jeder Ebene werden $c \cdot n$ Operationen ausgeführt (nur Vergleiche! Bewegungen werden gar nicht gemacht). Man erhält also im worst case

$$T(n) = n \cdot c \cdot n = \mathcal{O}(n^2).$$

Im *average case*, also im durchschnittlichen Fall, gehen wir davon aus, dass

1. alle Schlüssel paarweise verschieden sind und
2. dass jede der $n!$ Anordnungen gleich wahrscheinlich ist[2].

Aus Annahme 2. folgt, dass jedes Element mit gleicher Wahrscheinlichkeit $1/n$ als Pivotelement gewählt wird. Es werden zwei Teilfolgen aus einer Folge der Länge n erzeugt, welche die Längen $k-1$ und $n-k$ haben. Jede dieser Teilfolgenaufteilungen ist gleich wahrscheinlich und wir mitteln über den Aufwand. Das ergibt[3]

$$T(n) = \frac{1}{n} \cdot \sum_{k=1}^{n} (T(k-1) + T(n-k)) + bn.$$

Dabei bezeichnet b den Aufteilungsaufwand. Durch Umsortieren der Summanden erhalten wir

$$T(n) = \frac{2}{n} \cdot \sum_{k=1}^{n-1} T(k) + bn.$$

Dabei ist

$$T(0) = T(1) = 0,$$

also eine Folge ohne oder mit nur einem Element erzeugt keinen Aufwand. Wir zeigen durch Induktion, dass gilt

$$T(n) \leq c \cdot n \log n.$$

Dabei sei c genügend groß gewählt und wir nehmen an, dass n gerade ist (der ungerade Fall geht analog).

Induktionsanfang Es gilt $T(1) = 0$. Damit gilt für $n = 2$

$$T(2) = \frac{2}{2} \cdot \sum_{k=1}^{2-1} T(k) + 2b = T(1) + 2b = 2b \leq c \cdot n \log n = c2 \log 2 = 2c.$$

Für $T(3)$ hat man

$$T(3) = \frac{2}{3} (T(1) + T(2)) + 3b = \frac{13}{3} b \leq c \cdot 3 \log 3 \leq 3c \log 4 = 6c.$$

Wir werden später sehen, dass $c \geq 4b$ gewählt werden muss. Mit dieser Wahl ist auch der Induktionsanfang gültig.

Induktionsannahme Sei nun $n \geq 3$. Wir nehmen an, dass für alle $i < n$ gilt

$$T(i) \leq c \cdot i \log i.$$

[2] Wichtig hier ist, dass man mit der Annahme richtig liegt. Die nachfolgende Berechnung merkt nicht, ob eine andere Verteilung der Realität näher kommen würde!

[3] Hier wird nun der Erwartungswert bei angenommener Verteilung berechnet.Bei z.B. einem fairen Würfel erwattet man z.B. im Mittel eine Augenzahl von 3,5.

Dann folgt

$$\begin{aligned} T(n) &\leq \frac{2}{n}\left[\sum_{k=1}^{n-1} T(k)\right] + bn \\ &\overset{Ind.\ Ann.}{\leq} \frac{2c}{n}\left[\sum_{k=1}^{n-1} k \cdot \log k\right] + bn \\ &\overset{Summe\ auseinander}{\leq} \frac{2c}{n}\left[\sum_{k=1}^{\frac{n}{2}} k \cdot \log k + \sum_{k=1}^{\frac{n}{2}-1} \left(\frac{n}{2}+k\right) \log\left(\frac{n}{2}+k\right)\right] + bn \end{aligned}$$

Wir verwenden nun, dass k in der ersten Summe nicht größer wird als $\frac{n}{2}$. Wir können damit abschätzen

$$\log k \leq \log \frac{n}{2} = \log n - \log 2 = \log n - 1,$$

da wir den dyadischen Logarithmus verwenden. In der zweiten Summe wird das Argument im Logarithmus nie größer als n und so haben wir

$$\log\left(\frac{n}{2}+k\right) \leq \log n \text{ für alle } k = 1, \ldots, \frac{n}{2} - 1.$$

So können wir in unserer Abschätzung weiter machen und erhalten:

$$\begin{aligned} & \frac{2c}{n}\left[\sum_{k=1}^{\frac{n}{2}} k \cdot \underbrace{\log k}_{\leq \log n - 1} + \sum_{k=1}^{\frac{n}{2}-1} \left(\frac{n}{2}+k\right) \underbrace{\log\left(\frac{n}{2}+k\right)}_{\leq \log n}\right] + bn \\ \leq\ & \frac{2c}{n}\left[(\log n - 1)\sum_{k=1}^{\frac{n}{2}} k + \log n \sum_{k=1}^{\frac{n}{2}-1} \left(\frac{n}{2}+k\right)\right] + bn \\ =\ & \frac{2c}{n}\left[(\log n - 1)\sum_{k=1}^{\frac{n}{2}} k + \log n \left(\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \log n \sum_{k=1}^{\frac{n}{2}-1} k\right] + bn \end{aligned}$$

Wir wenden nun die Summenformel

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

für die beiden Summen an und erhalten

$$\begin{aligned} =\ & \frac{2c}{n}\left[(\log n - 1)\left(\frac{\frac{n}{2}(\frac{n}{2}+1)}{2}\right) + \log n \left(\frac{n}{2}-1\right)\frac{n}{2} + \log n \frac{(\frac{n}{2}-1)\frac{n}{2}}{2}\right] + bn \\ =\ & \frac{2c}{n}\left[\log n(\frac{\frac{n}{2}(\frac{n}{2}+1)}{2} + \frac{n}{2}(\frac{n}{2}-1) + \frac{(\frac{n}{2}-1)\frac{n}{2}}{2}) - \frac{\frac{n}{2}(\frac{n}{2}+1)}{2}\right] + bn \\ =\ & \frac{2c}{n}\left[\left(\frac{n^2}{2} - \frac{n}{2}\right)\log n - \frac{n^2}{8} - \frac{n}{4}\right] + bn \\ =\ & c \cdot n \log n - \underbrace{c \cdot \log n}_{\geq 0} - \frac{cn}{4} - \underbrace{\frac{c}{2}}_{\geq 0} + bn \\ \leq\ & c \cdot n \log n - \frac{cn}{4} + bn \\ \leq\ & c \cdot n \log n + \left(b - \frac{c}{4}\right) n \\ \leq\ & c \cdot n \log n \end{aligned}$$

Der letzte Schritt folgt mit $\frac{c}{4} \geq b$. Damit wäre gezeigt, dass

$$T(n) \leq c \cdot n \log n.$$

Das heißt, dass Quicksort auch im Mittel (average case) eine Aufwandsordnung von $\mathcal{O}(n \log n)$ hat.

Wie verhält sich Quicksort im schlechtesten Fall? Im besten Fall wird das Array immer in der Mitte geteilt. Im schlechtesten Fall wird immer nur ein Element abgespalten. Für die Anzahl Schlüsselvergleiche ergibt sich dann:

$$C_{max}(n) \geq \sum_{i=2}^{n}(i+1) = \frac{(n-1)(n+4)}{2} = \Omega(n^2)$$

Quicksort in der Praxis: Ein Problem besteht darin, ein geeignetes Pivotelement zu finden. Am günstigsten wäre ein Pivotelement mit einem Schlüssel, der die Ausgangsfolge in genau zwei gleich große Folgen aufteilt (sog. Median). Im allgemeinen (d.h. ohne Vorkenntnisse über die zu sortierende Folge) wird man so einen Schlüssel jedoch nicht finden können. Es gibt u. a. folgende Methoden:

- *3-Median-Strategie* Man wählt drei Datensätze aus und bestimmt hiervon den Median, also den mittleren Datensatz. Eine mögliche Wahl für die drei Datensätze sind der erste, letzte und ein an mittlerer Position liegender Datensatz.
- *Zufalls-Strategie* Man wählt aus einer Folge ein zufälliges Element aus und benutzt dessen Schüssel. Das auf diese Weise randomisierte (zufällig gemachte) Quicksort[4] behandelt alle Eingabefolgen fast gleich.

Typischerweise lohnt sich Quicksort nicht für weniger als 25-30 Elemente. Die Konstanten in der Aufwandsabschätzung machen sich für kleine n bemerkbar. Man sollte in der Rekursion hier also abbrechen. Man kann diese Folgen mit Insertionsort sortieren. Eine Möglichkeit ist auch, die Teilfolgen am Ende der Rekursion unsortiert zu lassen, und nach Zusammensetzen der Teilfolgen die gesamte Folge mit Insertionsort zu sortieren.

3.4 Mergesort

Mergesort ist ein weiteres Sortierverfahren, welches sich das *Divide-and-Conquer*-Prinzip zu Nutze macht. Die gesamte Folge wird in zwei Teilfolgen aufgeteilt, wobei jede für sich sortiert wird. Im Unterschied zu Quicksort werden die Teilfolgen nicht an einem Pivotelement getrennt. Es werden die ersten $n/2$ Positionen für die eine Folge genommen und der Rest für die zweite Folge. Man muss beim Zusammenfügen der Teilfolgen allerdings die Datensätze *einfädeln* (engl. merge).

Mergesort eignet sich besonders für externes Sortieren, also für Sortierprobleme bei denen die Datensätze auf externen Speichermedien abgelegt sind. Im Prinzip holt man sich eine bestimmte Anzahl von Datensätzen (soviel wie in den Speicher passen), sortiert sie, legt sie auf dem externen Speichermedium ab und holt sich neue Datensätze. Am Schluss werden die Datensätze durch mergen zusammengeführt und es entsteht eine sortierte Liste auf dem externen Speichermedium. Wir beginnen jedoch zunächst mit dem allgemeinen Verfahren.

2-Wege-Mergesort

Methode: Eine Folge

$$F = a_1, \ldots, a_n$$

[4]Randomisiertes Quicksort ist ein Beispiel für das Paradigma *Zufallsstichproben*. Randomisiertes Quicksort ist ein *LasVegasAlgorithmus*, denn er liefert immer ein korrektes Ergebnis, nur die Laufzeit ist eine Zufallsvariable. Es lässt sich zeigen, dass der Algorithmus mit „sehr hoher Wahrscheinlichkeit“ nicht viel mehr als die erwartete Laufzeit benötigt.

wird in zwei möglichst gleich große Teilfolgen

$$F_1 = a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor} \quad F_2 = a_{\lfloor \frac{n}{2}+1 \rfloor}, \ldots, a_n$$

aufgeteilt. Jede der Teilfolgen wird mit Mergesort sortiert. Das Zusammenfügen geschieht durch Verschmelzen der beiden sortierten Teilfolgen. Beim Verschmelzen geht man durch beide Teilfolgen von links durch, je mit einem Positionszeiger, und übernimmt den jeweils kleinern Datensatz in die Resultatfolge. Der Positionszeiger wird inkrementiert, von dessen Teilfolge das Element übernommen wurde. Ist eine Folge erschöpft, so übernimmt man den Rest der anderen Folge in die Resultatfolge.

Beispiel Wir nehmen wieder das Beispiel mit dem Kartenspiel.

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8

Zuerst wird die Folge in zwei Teilfolgen in der Mitte geteilt. Also so:

i:	1	2	3	4	5	6	7	8
a_i:	♢7	♢6	♢9	♢3	♢2	♢4	♢5	♢8

Danach wird jede Folge für sich sortiert. Wir nehmen mal an, dass das im folgenden Schritt schon geschehen sei:

i:	1	2	3	4	5	6	7	8
a_i:	♢3	♢6	♢7	♢9	♢2	♢4	♢5	♢8

Danach werden die Teilfolgen zusammengeführt.

i:	1	2	3	4	5	6	7	8							
a_i:	♢3	♢6	♢7	♢9	♢2	♢4	♢5	♢8							
	↑				↑				♢2						
	↑					↑			♢2	♢3					
		↑				↑			♢2	♢3	♢4				
		↑					↑		♢2	♢3	♢4	♢5			
		↑						↑	♢2	♢3	♢4	♢5	♢6		
			↑					↑	♢2	♢3	♢4	♢5	♢6	♢7	
				↑				↑	♢2	♢3	♢4	♢5	♢6	♢7	♢9

Algorithmus Mergesort(Folge F)
Falls F einen oder keinen Datensatz enthält, tue nichts
sonst:
Divide Teile F in zwei gleich große Teilfolgen F_1 und F_2.
Conquer Mergesort(F_1); Mergesort(F_2)
Merge Bilde Resultatfolge durch Zusammenführen von F_1 und F_2.

Analyse. Mergesort teilt im Gegensatz zu Quicksort eine gegebene Teilfolge immer in fast zwei gleichgroße Teilfolgen auf. Die Tiefe des Aufteilungsbaumes und somit die Rekursionstiefe sind also logarithmisch beschränkt. Diese Überlegung verdeutlicht, dass es für Mergesort keinen wirklichen *worst case* gibt. Für eine vorsortierte Folge ist die Rekursionstiefe genauso groß wie für jede andere Startsequenz. Die Anzahl der Bewegungen kann natürlich abweichen.

Das Verschmelzen zweier Teilfolgen geschieht in linearem Aufwand. Man erhält also wieder eine Rekursionsformel für den Aufwand in Form

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Die Lösung hatten wir schon bei Quicksort gesehen. Man erhält

$$T(n) = \Theta(n \log n).$$

Beim Verschmelzen von Teilfolgen werden jedoch viele Datenbewegungen gemacht. Obwohl der asymptotische Aufwand der gleiche ist, ist daher Quicksort im allgemeinen schneller. Mergesort eignet sich vielmehr für das Sortieren auf externem Speicher.

Mergesort auf externem Speicher

Wir nehmen nun an, dass die Anzahl n der Datensätze so groß ist, dass nicht alle Datensätze gleichzeitig in den Hauptspeicher passen. Wir werden daher im folgenden Teilfolgen einlesen, sie sortieren, und danach wieder auf den externen Speicher zurück schreiben. Am Ende werden die Teilfolgen zusammengeführt.

Methode Es werden vier externe Speicher t_1, t_2, t_3, t_4 verwendet. Zunächst werden wiederholt Datensätze von t_1 gelesen, intern sortiert, und abwechselnd so lange auf t_3 und t_4 geschrieben, bis t_1 erschöpft ist. Die Größe der Datensätze die wir jeweils einlesen sei I. Auf t_3 und t_4 sind nun sortierte Folgen von Datensätzen (Runs) vorhanden. Auf jedem der Speicher sind dies etwa $[n/(2I)]$.

Nun werden die Datensätze auf t_3 und t_4 zusammengeführt. Dabei schreiben wir die Ergebnisfolgen abwechselnd auf t_1 und t_2. Danach werden die Teilfolgen von t_1 und t_2 gelesen und auf t_3 und t_4 wieder zusammengeführt. Beim Zusammenführen von zwei sortierten Teilfolgen entsteht eine sortierte Teilfolge in der Länge der Summe der beiden Ausgangsfolgen. Die sortierten Teilfolgen werden immer länger, die Anzahl der verschiedenen sortierten Teilfolgen wird immer kleiner.

Beim Zusammenführen von sortierten Teilfolgen wird wenig interner Speicher benötigt. Beim Erzeugen der sortierten Teilfolgen am Anfang möchte man natürlich so lange Teilfolgen wie möglich erzeugen. Hier wird möglichst viel Hauptspeicher verlangt.

Die Verwendung von 4 externen Speichern sorgt dafür, dass der Lese-/Schreibkopf stets kontinuierlich von links nach rechts wandert und nicht evtl. bei jedem Zugriff erneut positioniert werden muss!

3.5 Rekursion bei Quick- und Mergesort

Rekursion unter einem anderen Blickwinkel[5]: Die Programme Mergesort und Quicksort sind typisch für Implementierungen von „Teile und Herrsche“-Algorithmen.

Quicksort sollte man vielleicht besser als „Herrsche und Teile“-Algorithmus bezeichnen: In einer rekursiven Implementierung wird der größte Teil der Arbeit für eine bestimmte Aktivierung vor den rekursiven Aufrufen erledigt. Andererseits hat der rekursive Mergesort eher den Geist von teilen und herrschen: Zuerst wird die Datei in zwei Teile aufgeteilt, dann wird jeder Teil einzeln beherrscht. Das erste Problem, für das Mergesort die Verarbeitung durchführt, ist klein; die größte Teildatei wird am Schluss verarbeitet. Quicksort beginnt mit der größten Teildatei und schließt mit der kleinsten ab. Es ist interessant, die Algorithmen im Kontext der Verwaltungsanalogie gegenüberzustellen: Bei Quicksort muss jeder Manager die richtige Entscheidung treffen, wie die Aufgabe zu gliedern ist, sodass ein komplettes Ergebnis vorliegt, wenn die Teilaufgaben erledigt sind. Bei Mergesort dagegen entscheidet jeder Manager ohne Nachzudenken, dass die Aufgabe zu halbieren ist, und muss sich dann mit den Konsequenzen herumschlagen, nachdem die Teilaufgaben fertig gestellt sind.

Der Unterschied manifestiert sich in den nichtrekursiven Implementierungen der beiden Verfahren. Quicksort muss einen Stack verwalten, weil große Teilprobleme zu speichern sind, die abhängig von den Daten aufgegliedert werden. Mergesort erlaubt eine einfache nichtrekursive Version, weil die Aufteilung der Dateien unabhängig von den Daten erfolgt, sodass wir die Reihenfolge, in der die Teilprobleme abgearbeitet werden, neu ordnen können, um das Programm zu vereinfachen. Man könnte Quicksort eher den Top-Down-Algorithmen zuordnen, weil er an der Spitze des Rekursionsbaums arbeitet und dann nach unten weitergeht, um das Sortieren zu komplettieren.

Wir haben festgestellt, dass sich Mergesort und Quicksort hinsichtlich der Stabilität unterscheiden. Wenn wir bei Mergesort annehmen, dass die Teildateien stabil sortiert werden, müssen wir nur noch sicherstellen, dass die Mischoperation stabil erfolgt, was sich leicht einrichten lässt. Die rekursive Struktur des Algorithmus führt sofort zu einem induktiven Beweis der Stabilität. Bei einer arraybasierten Implementierung von Quicksort bietet sich kein einfacher Weg für eine stabile Unterteilung an, sodass die Stabilität von vornherein ausgeschlossen ist, selbst bevor die

[5] Dies ist [Sed02] entnommen worden.

Rekursion ins Spiel kommt. Die geradlinige Implementierung von Quicksort für verkettete Listen ist dagegen stabil.

Algorithmen mit nur einem rekursiven Aufruf reduzieren sich praktisch zu einer Schleife, während Algorithmen mit zwei rekursiven Aufrufen wie Mergesort und Quicksort das Tor zu „Teile und Herrsche"-Algorithmen und Baumstrukturen öffnen, wo viele unserer besten Algorithmen angesiedelt sind. Mergesort und Quicksort sind eine sorgfältige Untersuchung wert, nicht nur aufgrund ihrer praktischen Bedeutung als Sortieralgorithmen, sondern auch, weil sie Einblicke in das Wesen der Rekursion bieten, was uns wiederum hilft, andere rekursive Algorithmen zu entwickeln und zu verstehen.

3.6 Heapsort

Heapsort ist eine Sortiermethode, die sich eine besondere Datenstruktur zu Hilfe nimmt: den Heap. Ein Heap ist ein binärer Baum, der eine bestimmte Bedingung erfüllt: die Heapbedingung. Das Sortierverfahren ermöglicht es, die Daten in absteigender Folge auszugeben, wobei in jedem Fall (*worst-case optimal*) $\mathcal{O}(n \log n)$ Operationen nötig sind. Dies geht jedoch nur, wenn die Daten bereits in einem Heap angeordnet sind. Doch dafür gibt es auch ein Verfahren der Ordnung $\mathcal{O}(n \log n)$.

Definition 3.12 (Heap)
Eine Folge $F = k_1, k_2, \ldots, k_n$ von Schlüsseln nennen wir einen *Heap* (genauer: Max-Heap) wenn

$$k_i \leq k_{\lfloor \frac{i}{2} \rfloor} \text{ für } 2 \leq i \leq n$$

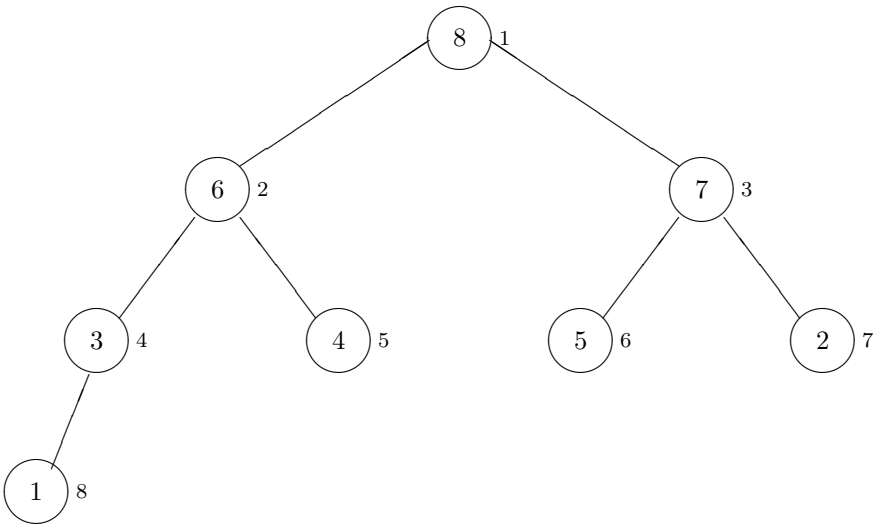
gilt. Gleichbedeutend damit ist

$$k_i \geq k_{2i} \text{ und } k_i \geq k_{2i+1}, \text{ falls } 2i \leq n \text{ und } 2i+1 \leq n.$$

Ein Min-Heap wird ganz analog definiert. ◀

Beispiel Betrachten wir folgendes Beispiel. Die Folge $F = 8, 6, 7, 3, 4, 5, 2, 1$ genügt der Heap-Bedingung, denn es gilt

$$8 \geq 6,\ 8 \geq 7,\ 6 \geq 3,\ 6 \geq 4,\ 7 \geq 5,\ 7 \geq 2,\ 3 \geq 1.$$



In der Darstellung stehen neben den Knoten die Positionen in der Liste, also der Index im Array; in den Knoten steht der Schlüssel. In einem Heap kann man also den linken und den rechten

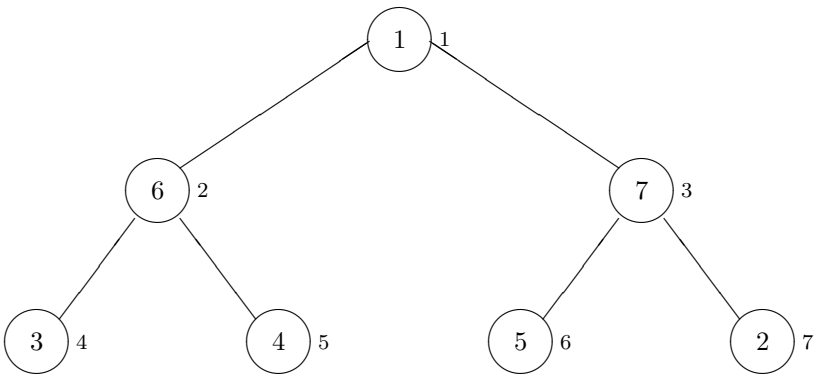
Nachfolger eines Knotens und seinen Vorgänger leicht berechnen:

$$\begin{aligned} \text{linkerNachfolger}(k_i) &= k_{2i} \\ \text{rechterNachfolger}(k_i) &= k_{2i+1} \\ \text{Vorgänger}(k_i) &= k_{\lfloor \frac{i}{2} \rfloor} \end{aligned}$$

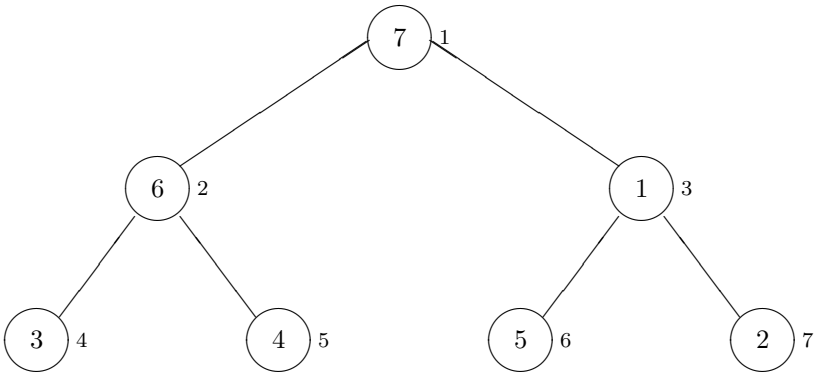
Methode Ist ein Heap gegeben, so ist die Ausgabe des größten Elements einfach. Es steht an der ersten Stelle k_1. Wie bekommt man nun das nächst kleinere Element? Man entfernt die Wurzel des Heaps, also das Element mit dem größten Schlüssel. Dadurch bekommt man zwei Heaps. Diese zwei Heaps fügt man wieder zu einem Baum zusammen, so dass die Heap Bedingung wieder erfüllt ist. Das geht folgendermaßen:

- schreibe das Element mit dem höchsten Index an die Wurzel des Baumes.
- vertausche dieses Element so lange mit dem größeren seiner Söhne, bis alle seine Söhne kleiner sind, oder bis es unten angekommen ist (man sagt, das Element *versickert*).

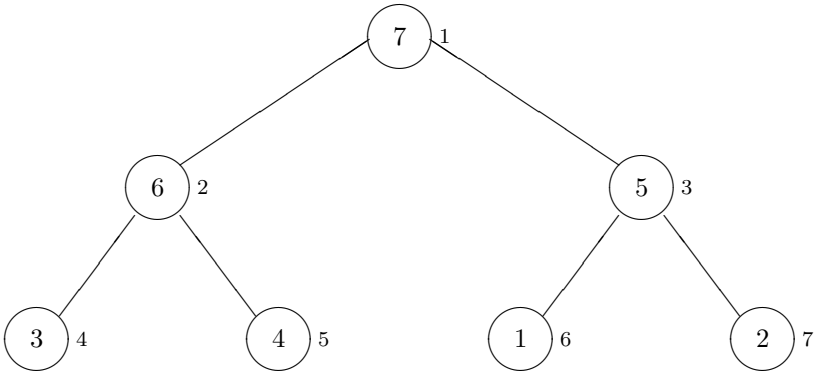
Dies ist im folgenden dargestellt. Zunächst das Element mit dem höchsten Index an die Wurzel des Baumes schreiben:



Nun wird das Element an Position 1 (die 1) mit seinem Sohn an Position 3 vertauscht (der 7), denn dieses ist das größere der Söhne.



Nun wird noch die 1 (Position 3) mit der 5 (Position 6) vertauscht.



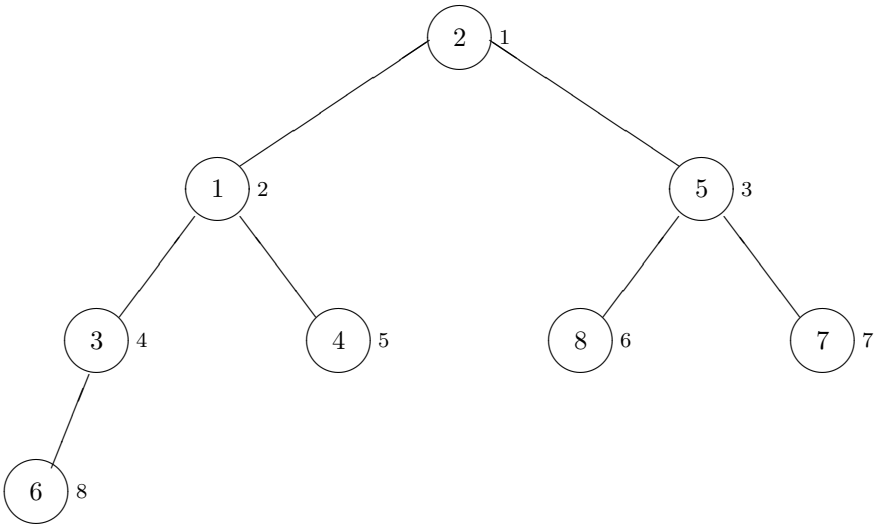
Der Ergebnisbaum ist wieder ein Heap. Nun kann die Wurzel ausgegeben werden als größtes Element des jetztigen Heaps. Dann wiederholt sich der Vorgang bis alle Elemente ausgegeben sind.

Nun muss noch die ursprüngliche, unsortierte Folge in einen Heap umgewandelt werden. Dazu das Verfahren:
Methode Sei $F = k_1, \ldots, k_n$ eine Folge von Schlüsseln. Sie wird in einen Heap umgewandelt, indem die Schlüssel

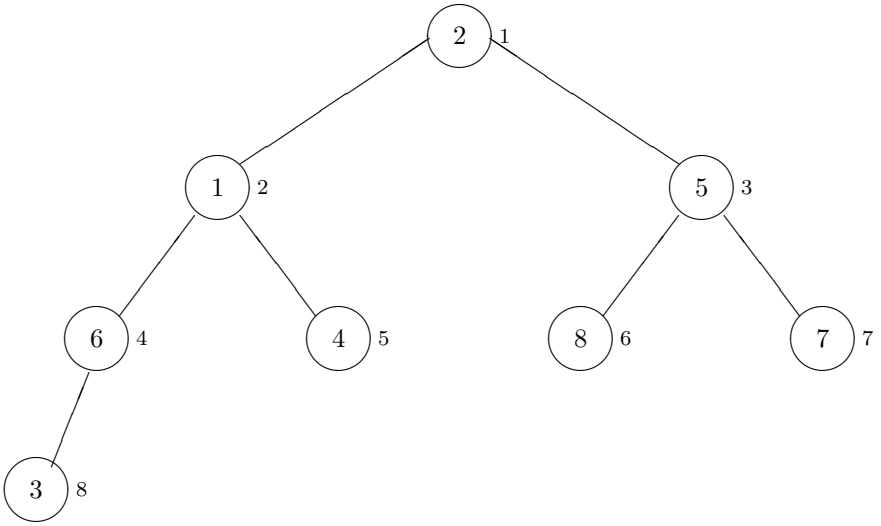
$$k_{\lfloor \frac{n}{2} \rfloor}, \ldots, k_1$$

in dieser Reihenfolge in F versickern.

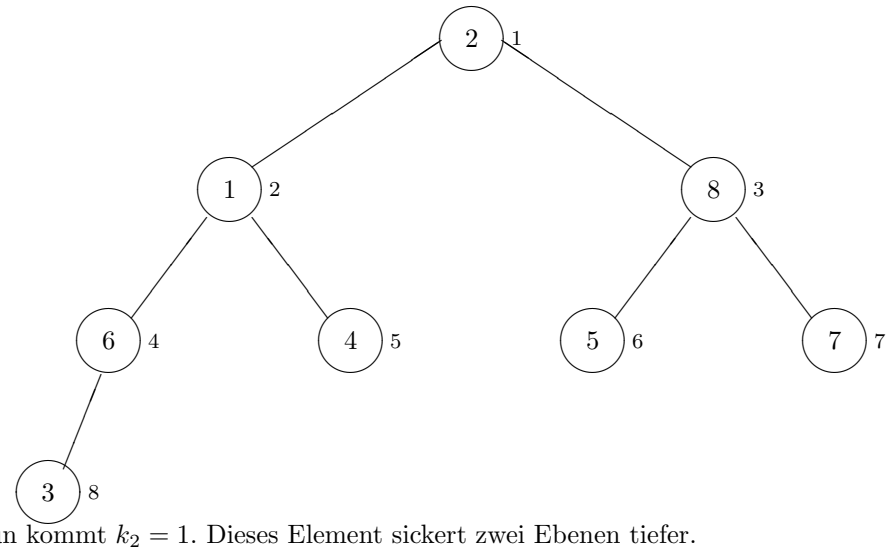
Beispiel Gegeben sei folgende unsortierte Folge in Binärbaumdarstellung:



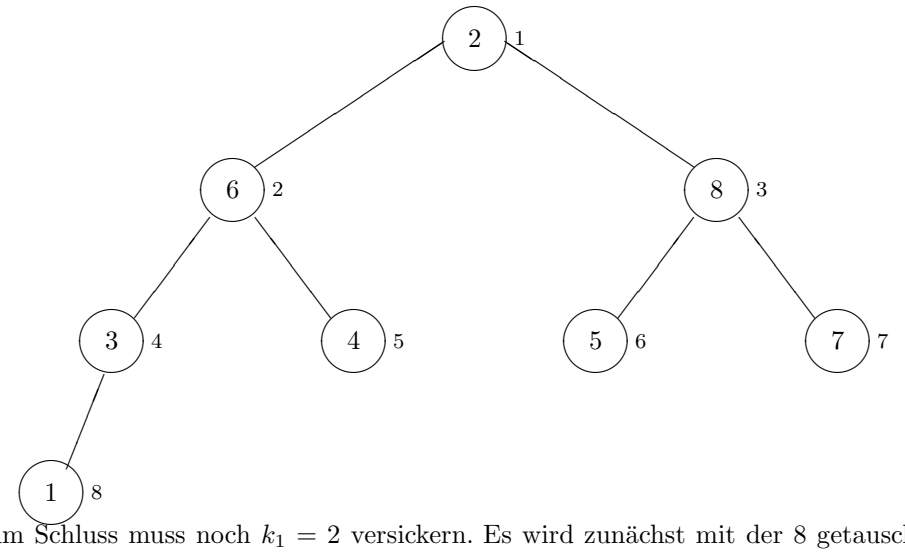
Wir beginnen mit Element $k_{\lfloor \frac{n}{2} \rfloor} = k_4 = 3$. Es versickert nur eine Position weiter runter:



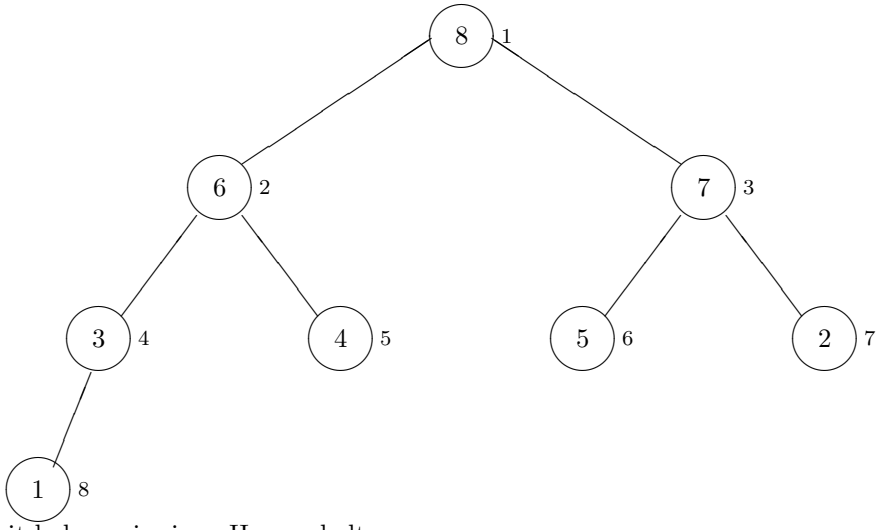
Dann kommt Element $k_3 = 5$. Es wird mit dem größeren seiner Söhne, also mit $k_6 = 8$ getauscht.



Nun kommt $k_2 = 1$. Dieses Element sickert zwei Ebenen tiefer.



Zum Schluss muss noch $k_1 = 2$ versickern. Es wird zunächst mit der 8 getauscht und dann mit $k_7 = 7$.



Somit haben wir einen Heap erhalten.

Analyse. Heapsort kann aus einer unsortierten Folge einen Heap machen mit Aufwand

$$\mathcal{O}(n).$$

Für das Zusammenfügen von mehreren Heaps zu einem Heap wird ein Aufwand von

$$\mathcal{O}(n \log n)$$

benötigt. Dies ist dann auch der asymptotische Aufwand für Heapsort. Heapsort ist im schlechtesten Fall asymptotisch optimal. Trotzdem ist Quicksort im allgemeinen (im Durchschnitt) schneller. Es gibt Varianten von Heapsort, die ausnützen, dass eine gegebene Folge vorsortiert ist (das angegebene Verfahren tut das nicht). Ein solches Verfahren benötigt $\mathcal{O}(n)$ Aufwand für eine vorsortierte Folge

Eine wichtige Anwendung für Heaps ist eine Prioritätswarteschlange (priority queue):

Definition 3.13 (Prioritätswarteschlange)
Eine Prioritätswarteschlange ist eine highest-in-first-out Warteschlange. Dies setzt voraus, dass auf den Einträgen eine Ordnung besteht. Diese Definition ist eindeutig, wenn jeder Schlüssel nut einmal vorkommt. Für den Fall, dass Schlüssel doppelt vorkommen können definiere ich: Der höchste Wert und unter gleichen der zuerst eingefügte wird als erste entnommen. Hier sind viele andere Definitionen denkbar. ◀

3.7 Fazit aus Sicht der Konstruktionslehre

Die wesentlichen Konstruktionsprinzipien hier sind

naiv : Der Algorithmus wird ohne weitere Betrachtung des Problems „naiv“ umgesetzt. Im Wesentlichen wird die zu sortierende Liste von links nach rechts so durchlaufen, dass alles links von der aktuellen Position als „sortiert“ eingestuft wird, und alles rechts von der aktuellen Position als „unsortiert“ eingestuft wird. Selection Sort arbeitet dabei im Wesentlichen im unsortierten Teil und wählt dort das kleinste Element aus; Insertion Sort arbeitet im Wesentlichen im sortiereten Teil und fügt das nächste, unsortierte Element an seiner (vorläufig) korrekten Stelle ein. Im Falle von Selection Sort bietet das Verfahren keine wirkliche Möglichkeit zu Verbesserungen[6]. Man kann Bubblesort als einen Versuch zur Verbesserung ansehen: hier wird bei „der Suche nach dem kleinsten Element“ durch Vertauschung der unsortierte

[6]Ein Beispiel dafür, dass ein Problemlöseverfahren in einer anderen Klasse als das Problem spielt. Trotzdem gibt es Situationen, in denen die praktische Laufzeitklasse dieses Verfahrens interessant wird!

Teil im Laufe der Zeit schon etwas mitsortiert. Im Falle von Insertion Sort kann jedoch durch etwas Nachdenken über das Verfahren eine wirkliche Verbesserung resultierend in Shell Sort erreicht werden.

Teile und Herrsche : Der Algorithmus teilt das Problem in Teilprobleme auf. Damit ist zwar das eigentliche Problem nicht näher betrachtet worden, dafür aber die Strategie des Problemlösens selbst.

Allgemeine Problemlösungsstrategien, wie das vorgestellte „Teile und Herrsche"-Prinzip, sind erste gute Ansätze, um Algorithmen zu beschleunigen, selbst wenn, wie im Praktikum zu sehen, die Lösung selbst zunächst durch dieses Prinzip „geteilt wird". Möchte bzw. muss man noch mehr „rausholen", so sollte die Eigenart des Problems selbst intensiver betracht werden. Im Praktikum hat dies zu der schnellsten Lösung geführt.

3.8 Aufgaben

1. Wie ist das asymptotische Verhalten von Insertion Sort im durchschnittlichen Fall (C_{avg}, M_{avg})? Machen Sie ggf. vereinfachende Annahmen, um zu einem zumindest unter diesen Annahmen gültigen Ergebnis zu kommen!
2. ([10]) Ist Insertion Sort stabil?
3. ([10]) Ist Selection Sort stabil?
4. ([15]) Don Knuth beschreibt in [Knu73b] *Algorithmus S* „straight selection sort". Dabei wird zunächst der Datensatz mit dem größten Schlüssel selektiert, dann der mit dem zweitgrößten. Warum ist das „more convenient"?
5. ([10]) Ist *Algorithmus S* stabil?
6. ([25]) Untersuchen Sie, wann Selection Sort *besser* ist als Quicksort!
7. ([25]) Untersuchen Sie, wann Selection Sort und wann Insertion Sort theoretisch besser ist und wann praktisch!
8. ([25]) Untersuchen Sie das Verhalten von Insertion Sort, wenn Schlüsselwerte mehrfach vorkommen!
9. ([25]) Ermitteln Sie bitte $C_{min}, C_{avg}, C_{max}$ und $M_{min}, M_{avg}, M_{max}$ für Bubblesort.
10. ([10])Ist Bubblesort stabil?
11. Implementieren Sie (inklusive der Funktion `swap`) ein einfaches Sortierverfahren für integer-Arrays, das nach folgendem Grundprinzip funktioniert:

    ```
    Solange gilt: Es gibt einen Index i
    in dem Array a mit a[i] > a[i+1] tue
                Führe swap(i, i+1) aus.
    ```

    Erklären Sie Ihre Umsetzung in konkretem Bezug zu diesem Grundprinzip. Bei der Implementierung darf nur ein Schleifenkonstrukt verwendet werden!

    Sortieren Sie mit Ihrer Implementierung folgendes array: $a = \{32, 3, 86, 0\}$. Geben Sie nach jedem Durchgang Ihrer Schleife den Zustand des aktuellen arrays an.
12. Sortieren Sie die folgende Zahlenreihe mit dem Shell Sort Algorithmus aus dem Tipp (nicht mit dem ggf. in Ihrem Gedächtnis vorhandenem Algorithmus!). Notieren Sie die Zwischenergebnisse nach jeder Einfügeoperation (, d.h. den Zustand nach Ausführung von Zeile `13`) sowie die jeweilige Distanz h.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
46	45	21	6	119	99	12	118	74	41

Geben Sie die insgesamte Anzahl der vorgenommenen Vertauschungen (Ausführung von Zeile `11`) an.

13. Gegeben sei ein Maximum-Heap, wie in der Vorlesung vorgestellt. Beantworten Sie folgende Fragen und begründen Sie Ihre Antworten:

    13.1. Was versteht man hierbei unter dem Begriff „Heap-Eigenschaft“?

    13.2. Wie wird die Baumrepräsentation eines Heaps in ein Array kodiert, also wo findet man die Nachfolger eines Knotens im Array?

    13.3. Was ist die minimale und maximale Anzahl von Elementen in einem Heap der Höhe h?

    13.4. Wo kann sich in einem Heap das kleinste Element nur befinden?

    13.5. Für welche Elemente in einer unsortierenden Zahlenfolge ist zu Anfang bereits die Heap-Bedingung erfüllt und warum wird diese Struktur von hinten (rechts) nach vorne (links) in der Array-Repräsentation eines Heaps aufgebaut?

14. Gegeben sei folgender Maximum-Heap:
    `( 45,31,41,23,15,12,32,10,8,2,13 )`.

    14.1. Zeichnen Sie den Heap als Baum.

    14.2. Fügen Sie nacheinander die Elemente 28 und 60 ein. Der Vorgang ist so zu dokumentieren, dass der Einfügeprozess nachvollzogen werden kann. Geben Sie dazu z.B. die vorgenommenen Vertauschungen an.

    14.3. Löschen Sie das Wurzelelement des nach dem Einfügen entstandenen Heaps, und stellen Sie die „Heap-Eigenschaft“ wieder her. Der Vorgang ist so zu dokumentieren, dass der Einfügeprozess nachvollzogen werden kann. Geben Sie dazu z.B. die vorgenommenen Vertauschungen an.

15. 15.1. Erklären Sie kurz das Teile-und-Herrsche-Prinzip. Beschreiben Sie dazu grob den Ablauf.

    15.2. Welches Problem könnte bei der Durchführung bestehen ? Denken Sie dabei z.B. an die Praktikumsaufgabe, in der die Teilsumme einer Folge von Zahlen zu bestimmen war (siehe Homepage von Christoph Klauck).

16. Quicksort und Mergesort unterscheidet nicht nur die Anwendung: Quicksort wird zu den „internen Sortierverfahren“ gezählt und Mergesort zu den „externen Sortierverfahren“.

    Beide arbeiten nach dem Teile-und-Herrsche-Prinzip. Was unterscheidet dennoch beide Algorithmen im Punkt Rekursion? Welche Auswirkung hat dies für eine iterative Variante von Quicksort?

17. Sortieren Sie die folgende Zahlenreihe mit dem Quicksort-Algorithmus. Notieren Sie die Zwischenergebnisse nach jeder Tauschoperation.

9	11	40	22	26	43	36	14	4	49

    Geben Sie die Anzahl der Rekursionsaufrufe und die Rekursionstiefe an.

Kapitel 4

Bäume und Graphen

Die Graphentheorie ist ein Teilgebiet der Mathematik, dessen Anfänge bis ins 18. Jahrhundert zurückreichen (Leonhard Eulers „Königsberger Brückenproblem“), das aber erst im 20. Jahrhundert größeres Interesse auf sich zu ziehen vermochte. Sie hat sich als nützliches Instrument zur Lösung verschiedenartigster Probleme erwiesen, etwa in den Wirtschaftswissenschaften, den Sozialwissenschaften oder der Informatik. Gerade für die Informatik ist die Graphentheorie ein wichtiges Gebiet, da dort Graphen einerseits zur rechnerinternen Darstellung von Informationen und Daten häufig verwendet werden[1] und andererseits zur Visualisierung von bestimmten Sachverhalten dienen.

In diesem Kapitel verwenden wir einmal Bäume (als spezielle Graphen), um Daten so abzulegen, das auf sie effizient zugegriffen werden kann, und wir verwenden Graphen als typische Datenstruktur um darin eine oft durchzuführende Aufgabe zu betrachten: das Suchen. Für komplexe Probleme ist diese Aufgabe so aufwendig, dass z. B. eigene Bücher nur zu diesem Thema verfasst wurden.

Bemerkung 4.1
Das Abrufen bestimmter Informationseinheiten aus größeren vorher gespeicherten Datenbeständen ist eine fundamentale Operation, die man als **Suchen** bezeichnet. Diese Operation spielt in sehr vielen Aufgaben der Datenverarbeitung eine zentrale Rolle. Auch ein Programmablauf kann als Suche beschrieben werden, insbesondere dann, wenn man keine vollständigen und/oder korrekten Algorithmen mehr verwenden kann.

Es können folgende Fälle unterschieden werden:

1. Die Daten sind **nicht sortiert**.
   Man hat nun im Wesentlichen zwei Möglichkeiten:

   1.1. Die Daten sortieren:
   Wie bei den Sortieralgorithmen und insbesondere den Prioritätswarteschlangen (z. B. Heap) arbeiten wir mit Daten, die in Datensätze oder Elemente gegliedert sind. Jedes Element verfügt dabei über einen Schlüssel, der in Suchoperationen verwendet wird. Das Ziel der Suche besteht darin, Elemente zu finden, deren Schlüssel mit einem gegebenen Suchschlüssel übereinstimmen. Im Ergebnis der Suche greift man normalerweise auf die Informationen innerhalb des Elements (und nicht nur auf den Schlüssel) zu, um die Informationen weiterzuverarbeiten.

   1.2. Die Daten nicht sortieren:
   Es sind Methoden von Interesse, die solche unsortierten „Auswahlmöglichkeiten“ einschränken und auf möglichst effiziente Weise nach der richtigen Lösung suchen. Um die Lösung zu finden, macht man sich einen Plan, konstruiert damit also ein neues Objekt. Die Schwierigkeit ist stets, eine Abschätzung dafür zu geben, ob man sich einer guten Lösung genähert hat und ob man sich schnell nähert (Konvergenzverhalten).

[1] Stichwort: Abstrakter Datentyp, semantische Netze, Constraint-Netze etc.

Um das Suchen nach einer (optimalen) Lösung, zu modellieren, wollen wir uns folgender abstrakter Grundvorstellungen bedienen:

- Das Bergsteigermodell: Wir bewegen uns in einem Gebirge mit dem Ziel, den höchsten Gipfel zu erklimmen; die erreichte Höhe gibt dabei den Gütegrad unserer bisherigen Lösung oder Teillösung an.
- Das Graphensuchmodell: Hier denken wir uns die noch zu lösenden Teilprobleme an den Knoten eines Graphen angeheftet, wobei ein Schritt entlang einer Kante uns der Lösung näher bringt, uns also ein neues (hoffentlich reduziertes) Teilproblem liefert.

Diese beiden Modellvorstellungen stehen nicht in Konkurrenz zueinander, sondern sie ergänzen sich. Bei erstem Modell wird die Art der Bewertung modelliert und oft durch Gradientenabstiegsverfahren realisiert. Beim zweitem Modell lassen sich Buchführungsmethoden über das Fortschreiten überhaupt beschreiben. Hier werden explizite Heuristiken oft eingesetzt. Letztlich ist dies ein Gebiet, das im Bereich der Künstlichen Intelligenz ausgiebig untersucht wird.

2. Die Daten sind **sortiert**.
   Hier werden die Daten in einer Art Wörterbuch (Symboltabelle) gehalten, die im Wesentlichen zwei Operationen unterstützt: Einfügen eines neuen Elementes und Zurückgeben eines Elementes mit einem gegebenen Schlüssel. Die Idee ist, die Daten weitestgehend sortiert zu halten. Das schauen wir uns in diesem Kapitel genauer an.

Mit Heapsort haben wir ein Verfahren kennen gelernt, das einmal genutzt werden kann, um Daten *lazy* zu sortieren (Bottom-Up Heapsort) und einmal, um Daten sortiert zu halten (Top-Down Heapsort). In beiden Fällen hat Heapsort sehr günstige Laufzeitkonditionen. ◀

4.1 Definition von Bäumen

Wir haben im vorherigen Kapitel über Sortierverfahren schon gesehen, dass Datenstrukturen in Baumstruktur hilfreich sein können. Aber mit Bäumen lassen sich auch hierarchische Strukturen darstellen, so wie man sie in vielen Bereichen antrifft:

- Organigramm eines Unternehmens; also Aufteilung in Abteilungen, Gruppen und deren Mitarbeiter
- Gliederung eines Buches in Kapitel, Unterkapitel, Abschnitte
- Aufteilung von Deutschland in Bundesländer, Kreise, Gemeinden
- Abstammungsbaum eines Menschen: Eltern, Großeltern, Urgroßeltern

Auch ein vollständig geklammerter Ausdruck kann wie in Abb. 4.1 als Baum dargestellt werden: Die Operanden und Operatoren des Ausdrucks $(3 * 12 + 4) * (5/7)$ werden dabei von innen nach außen in der Baum eingetragen. $3 * 12$ ist der unterste linke Teilbaum, $+4$ steht „darüber“. $5/7$ steht im rechten unteren Teilbaum und der Gesamtausdruck wird über die Wurzel verknüpft. Wie man dies systematisch behandelt, werden wir auf S. 86 sehen.

Definition 4.2 (Baum)
Ein Baum $T = (V, E)$ (für *Tree*) ist eine Menge V (für *vertices*) von **Knoten**, zusammen mit einer Relation

$$E \subset V \times V$$

(für edge) auf der Knotenmenge V (graphisch durch **Kanten** dargestellt), so dass sich eine hierarchische Struktur ergibt. Ein Knoten hat also höchstens einen vorangehenden Knoten und keinen, einen oder viele nachfolgende Knoten. ◀

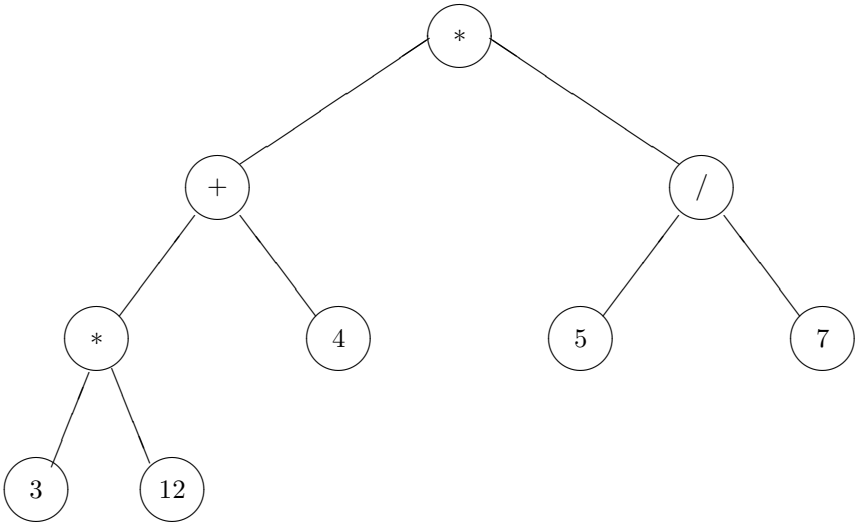


Abbildung 4.1: Ein vollständig geklammerter Ausdruck

Man kann den Begriff des Baumes auch rekursiv definieren:

Bemerkung 4.3 (Rekursive Definition von Baum)
Ein Baum ist eine endliche Menge V von Knoten mit folgenden Eigenschaften:

1. Ein Knoten ist die Wurzel (root)
2. Die anderen Knoten (außer der Wurzel) sind in $m > 0$ disjunkte Teilmengen $V_1, \ldots, V_m$ zerlegt, von denen jede wiederum ein Baum ist. Diese heißen Teilbäume.

◀

Die Bezeichnugen für die Elemente eines Baumes leiten sich zum Teil von der Anwendung zur Darstellung von Familienstammbäumen ab. Die folgende Definition fasst sie zusammen:

Definition 4.4 (Elemente eines Baums)
Knoten Für Knoten sind folgende Definitionen üblich

- **Kante** Ein Element aus E (Verbindungslinie zwischen Knoten)
- **Sohn** (oder **Kind**) eines Knotens $v \in V$ ist ein Knoten $s \in V$, so dass

  $$(v, s) \in E$$

  ,d. h. ein unmittelbar nachfolgender und durch Kante verbundener Knoten (nach „unten“ in der üblichen Darstellung von Bäumen).
- **Vater** eines Knotens $x \in V$ ist der Knoten $v \in V$, so dass

  $$(v, x) \in E$$

  gilt, d. h. dies ist der darüberliegende Knoten (durch Kante verbunden).
- **Grad** eines Knotens $x \in V$ ist

  $$|\{s \in V : (x, s) \in E\}|,$$

  d. h. die Anzahl der Söhne oder Kinder.

Baum Auf dieser Ebene sind folgende Definitionen sinnvoll und üblich:

- **Wurzel** eines Baums ist ein Knoten $w \in V$, so dass

$$\forall x \in V : (x, w) \notin E$$

  (höchster Knoten im Baum: hat keinen Vater)

- **Pfad** ist eine Folge von Knoten $p_0, \ldots, p_n$, so dass

$$(p_i, p_{i+1}) \in E \text{ für alle } i = 1, \ldots, n-1.$$

- **Länge eines Pfades** Anzahl der Kanten
- **Tiefe** eines Knotens ist die Länge des Pfades zum Wurzelknoten
- **Höhe** Länge des längsten Pfades ausgehend von der Wurzel bis zu einem Blatt
- **Grad** Maximum der Grade aller Knoten

◀

Im Folgenden betrachten wir binäre Bäume. Das sind Bäume, bei denen jeder Knoten höchstens zwei Söhne hat, also Bäume vom Grad 2. Diese sind einfacher als allgemeine Bäume, bei denen für jeden Knoten eine maximale Anzahl $d \in \mathbb{N}$ von Söhnen erlaubt ist.

Definition 4.5 (Binärbaum)
Ein Baum heißt *binär*, wenn die Wurzel und alle anderen Knoten 2 Nachfolger oder weniger Nachfolger hat. Die Knoten mit 0 Nachfolgern sind die Blätter des binären Baums. Ein binärer Baum heißt *voll*, wenn zwischen jedem Blatt und der Wurzel dieselbe Anzahl von Kanten liegt und mit Ausnahme der Blätter jeder Knoten genau zwei Söhne besitzt. Von diesen wird einer als *rechter Sohn* und der andere als *linker Sohn* bezeichnet. Ein binärer Baum heißt *vollständig*, wenn der Baum auf jeder Ebene bis auf die unterste vollständig besetzt ist und auf der untersten Ebene höchstens Knoten ganz rechts fehlen. Ein binärer Baum heißt *höhenbalanciert*, wenn die Höhen der beiden Teilbäume jeden Knotens sich maximal um 1 unterscheiden. ◀

Ein Heap ist also ein vollständiger binärer Baum.

Bemerkung 4.6 (Binärbaum)
Durch die Unterscheidung von linkem und rechtem Sohn ist ein Binärbaum etwas Anderes als ein allgemeiner Baum vom Grad 2. So sind die beiden Bäume in Abb. 4.2 als Bäume vom Grad 2

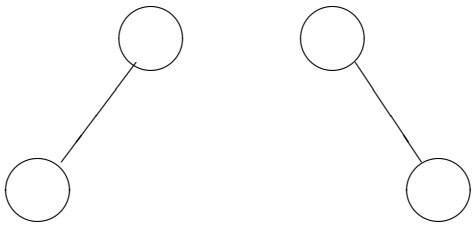


Abbildung 4.2: Gleiche Bäume — ungleiche Binärbäume

identisch, als Binärbäume aber verschieden, da die Wurzel des linken nur einen linken Sohn und die des rechten nur einen rechten Sohn besitzt. ◀

4.2 Eigenschaften von Bäumen

Wir wollen nun Eigenschaften von Bäumen untersuchen. Zunächst interessiert uns die Höhe eines Baumes mit einer bestimmten Anzahl von Knoten. Dazu gibt es folgende Aussagen:

Satz 4.7 (maximale und minimale Höhe eines Baums)

1. Die maximale Höhe eines Baumes mit n Knoten ist $n-1$.
2. Die minimale Höhe eines Baumes vom Grad 2 mit n Knoten ist $\lfloor \log_2 n \rfloor$.

◀

Beweis:

1. Dieser Fall tritt ein, wenn der Baum zu einer Liste entartet. Abbildung 4.3 zeigt das Prinzip. Für $n = 1$ und $n = 2$ ist die Aussage klar. Ist sie für $n = n_0$ wahr, so folgt sie für n_0+1 durch

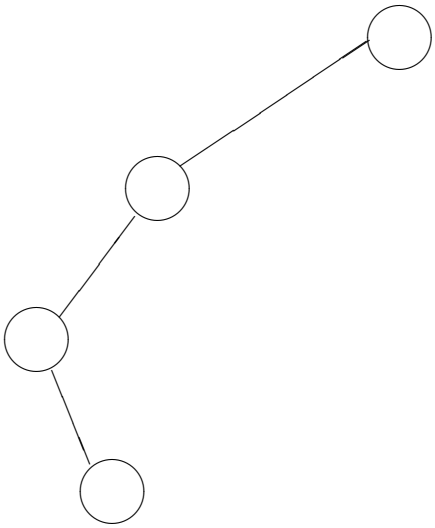


Abbildung 4.3: Zu einer Liste entarteter Baum

einfügen eines Kindes unter dem untersten Knoten des „darüberliegenden“ Teilbaums. Man kann dies auch direkt einsehen: Nach Definition der *Höhe* eines Baumes ist dies die Länge des längstens Pfads. Außer der Wurzel gibt es aber nur $n-1$ weitere Knoten. Das Beispiel zeigt, dass diese Höhe auch erreichbar ist.

2. Dieser Fall tritt ein, wenn bis auf einige der untersten Knoten alle Knoten besetzt sind. Das Prinzip zeigt Abb. 4.4 Ist ein binärer Baum mit Höhe h nämlich vollständig, so hat er

$$1 + 2 + 4 + 8 + 16 + 32 + \cdots + 2^h = 2^{h+1} - 1 = n$$

Knoten. In diesem Fall gilt: $h = \lfloor \log_2(2^{h+1} - 1) \rfloor = \lfloor \log_2 n \rfloor$

Aufgabe 4.8
Wieviele Knoten n hat ein vollbesetzter Baum vom Grad d und Höhe h? Welche minimale Höhe h hat ein Baum vom Grad d mit n Knoten? ◀

4.2.1 Reihenfolgedurchläufe

Auf Bäumen kann man verschiedene Reihenfolgen von Knoten definieren. Drei Reihenfolgen haben einen bestimmten Namen: inorder (symmetrische Reihenfolge), postorder (Nebenreihenfolge) und preorder-Reihenfolge (Hauptreihenfolge) . Sei dazu T_k ein Teilbaum, der als Wurzel einen

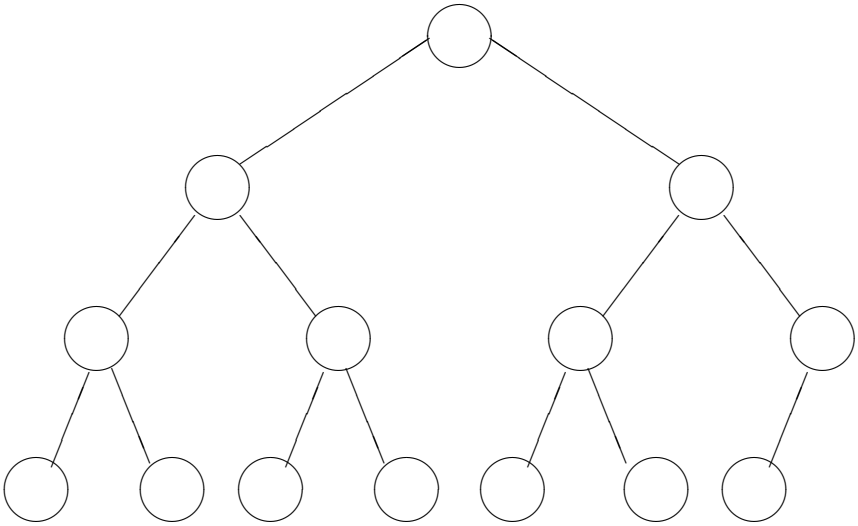


Abbildung 4.4: Baum minimaler Höhe

bestimmten Knoten k hat. Dann gibt es zwei Teilbäume unterhalb von k, nämlich T_{k1} und T_{k2}, wobei $k1$ der linke und $k2$ der rechte Sohne von k ist. Seien inorder(), preorder() und postorder() Abbildungen, die einen Teilbaum als Argument nehmen, und eine Folge von Knoten zurückliefern, also

$$\begin{aligned} \text{inorder} &: \text{Teilbaum} \longrightarrow \text{Folge von Knoten} \\ \text{preorder} &: \text{Teilbaum} \longrightarrow \text{Folge von Knoten} \\ \text{postorder} &: \text{Teilbaum} \longrightarrow \text{Folge von Knoten} \end{aligned}$$

Wir definieren die Reihenfolgen dann rekursiv.

- inorder(T_k) = (inorder(T_{k1}), k, inorder(T_{k2}))
- preorder(T_k) = (k, preorder(T_{k1}), preorder(T_{k2}))
- postorder(T_k) = (postorder(T_{k1}), postorder(T_{k2}), k)

Folgende Bezeichnung verwendet man außerdem für die Reihenfolge:

- inorder = symmetrisch
- preorder = Hauptreihenfolge
- postorder = Nebenreihenfolge

Aufgabe 4.9
Geben Sie die Inorder-, Preorder- und Postorder-Reihenfolgen des Baums in Abb. 4.5 an. ◀

4.3 Implementierungen

4.3.1 Dynamische Implementierung

Man kann Bäume so implementieren wie Listen. Die Knoten sind Objekte, welche mit Referenzen (Zeigern) auf ihre Söhne zeigen. Die Klasse Knoten hat also bei Binärbäumen zwei Zeiger auf ihre Söhne. Möchte man die Bäume auch von unten nach oben durchgehen, so benötigt man noch eine Referenz oder einen Zeiger auf den Vater.

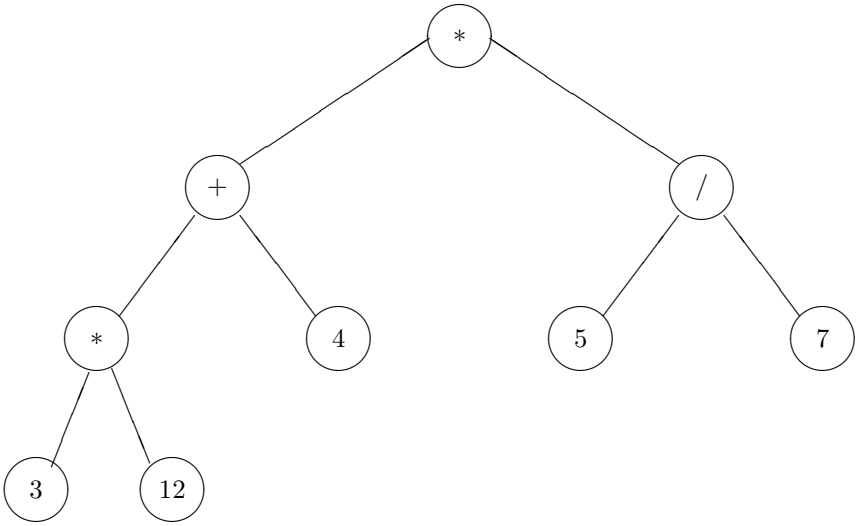


Abbildung 4.5: Baum für geklammerten Ausdruck

```
class Knoten<T>
{
T daten;
int key;
Knoten<T> links, rechts, vater;
}
```

Dabei enthalten `daten` die Daten des Knotens, also zum Beispiel eine Operation oder einen Wert.

Bemerkung 4.10
`vater` ist eine Information, die bei rekursiver Programmierung nicht notwendig ist, da die darin enthaltene Information (Adresse des Vorgängers) im Stack abgespeichert ist. Bei iterativer bzw. nicht rekursiver Programmierung bietet dieser explizite Zeiger ähnliche Vorteile, wie bei doppelt verketteten Listen. ◄

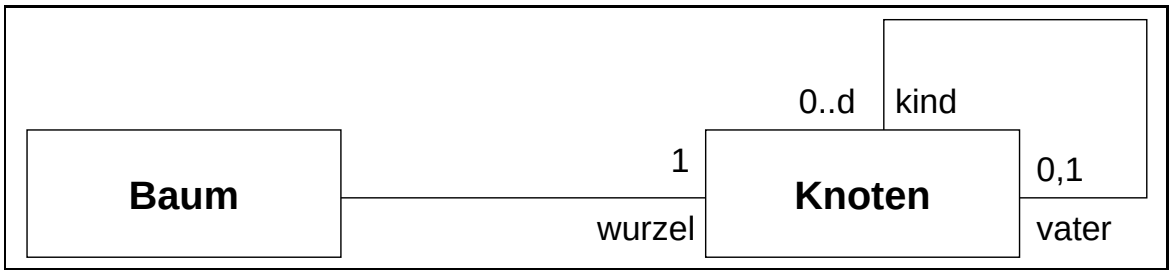


Abbildung 4.6: UML-Darstellung der Zeiger- oder Referenzimplementation eines Baums

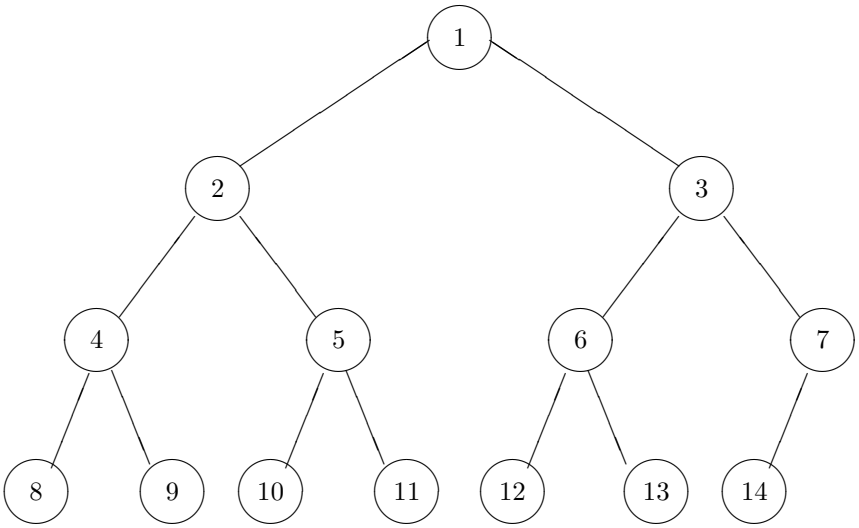
Man benötigt einen weiteren Datentyp, der die Wurzel des Baumes kennzeichnet. Dies ist nötig, da in den Algorithmen (siehe z. B. Heap Sort) die Wurzeln entfernt werden. Dabei erhält man zwei neue Bäume je wieder mit einer Wurzel.

```
class Baum<T>
{
    Knoten<T> wurzel;
}
```

4.3.2 Statische Implementierung

Natürlich kann man die oben definierten Knoten einfach als Array allokieren und danach entsprechend verlinken. Mit Array-Einbettung ist jedoch eine Implementierung gemeint, die völlig ohne Zeiger oder Referenzen auskommt. Die Struktur entsteht durch Indexrechnung im Array.

Dazu betrachten wir einen weitgehend vollständigen binären Baum der Höhe h. Dieser wird



in ein Array der Größe 2^{h+1} eingebettet[2]. Die Zahlen in den Knoten in obiger Abbildung geben den Index im Array wieder. Sei $i \in \{1, \ldots, 2^{h+1} - 1\}$ ein beliebiger Knoten. Dann kommt man zu seinem Sohn durch die Rechnung

$$\text{SohnLinks}(i) = 2i \quad \text{SohnRechts}(i) = 2i + 1,$$

falls $2i$ bzw. $2i + 1$ noch im Bereich des Arrays liegen. Zum Vater von i kommt man durch

$$\text{Vater}(i) = \left\lfloor \frac{i}{2} \right\rfloor,$$

falls das Ergebnis nicht Null ist, denn dann war i schon die Wurzel des Baumes.

Wenn die darzustellenden Bäume fast vollständig besetzt sind, dann ist die Array-Einbettung eine gute Implementierung. Sind die Bäume dagegen nur schwach besetzt, so wird viel Speicherplatz verschwendet.

Aufgabe 4.11 (Baum als Array, Array als Baum)
Stellen Sie einen vorgegebenen Baum als Array dar. Stellen Sie ein vorgegebenen Array als Baum dar! ◀

4.3.3 Implementierung für Reihenfolgendurchläufe

Für die Traversierung von binären Bäumen gibt es im Wesentlichen zwei Implementierungen.

1. **Rekursive Implementierung** Es soll eine Liste von Knoten erzeugt werden, die der entsprechenden Durchlaufordnung entspricht (in-, pre- oder postorder). Da die Durchlaufordnungen rekursiv definiert wurden, können sie auch entsprechend implementiert werden. Hier ein Pseudocodeprogramm für Herstellung einer Postorderreihenfolge.
   Postorder(Teilbaum B)

   - Falls B keine Söhne hat, so gib B zurück

[2] Die Position $i = 0$ bleibt unbenutzt. Das kann man aber auch anders handhaben.

- Ansonsten bestimme Wurzelknoten w von B
- Bestimme den linken Teilbaum l von w
- Bestimme den rechten Teilbaum r von w
- Gib die Folge $(Postorder(l), Postorder(r), v)$ zurück

Bei *pre-Ausgabe* wäre die Ausgabe vor den rekursiven Aufrufen zu tätigen; bei *in-Ausgabe* zwischen den beiden rekursiven Aufrufen.

2. **Nicht-rekursive Implementierung** Diese kann durch explizite Implementierung eines Stacks oder zusätzliche Zeiger geschehen, die die Reihenfolge angeben. Von jedem Knoten gibt es einen Zeiger auf seinen Nachfolger bezüglich der entsprechenden Reihenfolgeordnung.

   Im Falle der symmetrischen (inorder) Reihenfolgeordnung sind zusätzliche Zeiger nicht nötig. Hier lautet die Rekursionsdefinition

   $$\text{inorder}(k) = (\text{inorder}(linkerTeilbaum), k, \text{inorder}(rechterTeilbaum)).$$

   Der Nachfolger eines Knotens k bestimmt sich durch den am weitesten linken Knoten l seines rechten Sohn-Teilbaums. Diesen kann man leicht finden, indem man die bestehende Baumstruktur ausnutzt.

   Sollte der rechte Teilbaum nicht existieren, so kann man den vorhandenen Zeiger für den rechten Teilbaum (der dann üblicherweise auf `null` steht) auf den entsprechenden Nachfolger zeigen lassen. Diese Struktur nennt man *gefädelter Baum* (threaded tree).

   Mit einem Array ist nur die Breitentraversierung einfach auszugeben, weil man eine triviale `for`-Schleife verwenden könnte. Für z. B. eine Preorder reicht eine `for`-Schleife nicht aus!

4.4 Binäre Suchbäume

Im Folgenden nehmen wir an, dass jeder Knoten im binären Baum mit einer natürlichen Zahl markiert ist. Wir haben also eine Abbildung (Knotenmarkierung) von allen Knoten $k \in T$ im Baum nach $\mathbb{N}$

$$\mu \ : \ T \longrightarrow \mathbb{N}.$$

Definition 4.12
Ein binärer Baum mit einer Knotenmarkierung μ heißt **binärer Suchbaum** genau dann, wenn für jeden Teilbaum T' von T mit $T' = (T_l, y, T_r)$ gilt:

$$\begin{aligned} \forall x \in T_l \quad &: \quad \mu(x) < \mu(y) \\ \forall z \in T_r \quad &: \quad \mu(z) > \mu(y) \end{aligned}$$

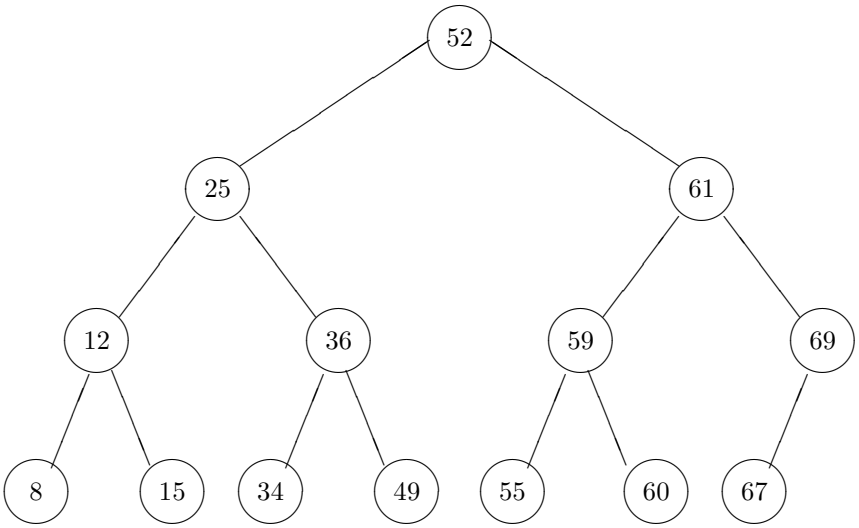
◀

Alle Werte (der Knoten) im linken Teilbaum seien also kleiner als die Wurzel des Teilbaums, und alle Werte im rechten Teilbaum seien größer als die Wurzel des Teilbaums.

Beispiel 4.13 (Binärer Suchbaum)
Die Abb. 4.13 zeigt ein einfaches Beispiel eines binären Suchbaums. ◀

Bemerkung 4.14 (Inorder)
Die Inorder-Reihenfolge (symmetrisch) liefert bei binären Suchbäumen eine geordnete Folge der Knoten. ◀

Implementierung Wir wollen uns die Einfüge-Funktion (insert) bei einem binären Suchbaum ansehen. Angenommen die Klasse für den Knoten sieht aus wie folgt:




```
class Knoten<T>
{
   int key;
   Knoten<T> links,rechts,vater;
}
```

Dann kann eine Methode definiert werden, die feststellt, ob ein Knoten mit einem bestimmten Key vorkommt.

```
public boolean exists(int k){
     if(k == key) return true;
     else
     {
          if(k < key)
               if(links == null) return false;
               else return links.exists(k);
          else
               if(rechts == null) return false;
               else return rechts.exists(k);
     }
}
```

Diese Routine wird für den Wurzelknoten des Baumes aufgerufen und liefert **true** zurück, wenn der gefragte Knoten existiert.

Um einfach zugreifen zu können, kann man natürlich eine entsprechende Operation `exists` der Klasse Baum definieren, die lediglich die entsprechende Operation des Wurzelobjekts der Klasse `Knoten` aufruft. Entsprechendes gilt für die nun zu entwerfende Operation `insert`.

Wir erweitern nun diese Routine um zu einer Einfügeroutine zu gelangen. Dabei wird der neue Knoten eingehängt, wenn wir für `links` oder `rechts` auf eine NULL-Referenz stoßen.

```
Knoten<T> insert(T d, int k)
{
     if(k == key) return this;
     else
```

```
    {
         if(k < key)
         {
              if(links == null)
              {
          links = new Knoten<T>(d,k);
          return links;
              }
              else return links.insert(d, k);
         }
         else
         {
              if(rechts == null)
              {
          rechts = new Knoten<T>(d,k);
          return rechts;
              }
              else return rechts.insert(d,k);
         }
    }
}
```

Diese Routine liefert eine Referenz auf einen Knoten zurück, der entweder bereits den einzufügenden Wert besitzt, oder der gerade erzeugt worden ist um den Wert einzufügen.

Die Löschroutine zum Entfernen eines Knotens ist etwas aufwendiger. Beim Löschen eines Blattes, wird einfach die Referenz auf den Sohn auf Null gesetzt. In einer Sprache wie C++ ist ggf. das Knotenobjekt zu löschen. Hat der zu löschende Knoten nur ein Kind, so wird er ausgehängt: Die Referenz des Vaters wird auf das Kind gesetzt. Hat der Knoten zwei Nachfolger, so vertauschen wir ihn mit seinem ersten Nachfolger, der höchstens ein Kind hat. So sorgen wir dafür, dass beim Löschen eines Knotens die Suchstruktur des Baumes erhalten bleibt. Dies will ich hier nicht ausführen und verweise auf die Literatur (z. B. [Knu97a], [GD04], [CLR94]).

4.5 AVL-Bäume

Einfache Beispiele $(575, 28, 4711, 69, 1234)$ zeigen, dass auch binäre (Such-) Bäume zu Listen entarten können. Es gibt deshalb viele Ideen, wie man dies verhindern kann.

Der erste Vorschlag zu höhenbalancierten Bäumen wurde 1962 von Adelson-Velskiĭ und Landis gemacht. Man nennt sie AVL-Bäume und sie sind folgendermaßen definiert:

Definition 4.15 (AVL-Baum)
Ein binärer Suchbaum heißt AVL-Baum, wenn für jeden Knoten p gilt, dass sich die Höhen des linken und rechten Teilbaums höchstens um 1 unterscheiden. ◀

Mit AVL-Bäumen soll vermieden werden, dass Bäume zu Listen entarten. Dadurch soll das Suchen in einem binären Baum einen Aufwand $\mathcal{O}(\log n)$ haben. Wichtig ist hier, dass ein AVL-Baum von Anfang an aufgebaut werden muss, d.h. ein beliebiger Suchbaum kann nur in einen AVL-Baum transformiert werden, indem alle Elemente neu eingefügt werden.

Bei jedem Einfügen und Löschen eines Knotens in einen AVL-Baum wird durch eine *Rebalancieroperation* wieder die AVL-Bedingung hergestellt. Hier müssen verschieden Fälle unterschieden werden. Ein einfacher Fall wird hier zunächst vorgeführt. Dazu betrachten wir einen Teilbaum, der durch Einfügen eines Knotens die AVL-Bedingung verletzt. Hier der ursprüngliche Baum:

Nun soll ein Knoten mit Wert 67 eingetragen werden. Das ergibt zunächst — wie in Abb. 4.8 gezeigt — eine Verletzung der AVL-Bedingung. Der rechte Teilbaum hat nun Höhe 2, während der linke Teilbaum (der Knoten mit Wert 25) Höhe 0 hat. Wir rotieren den Baum nun gegen den

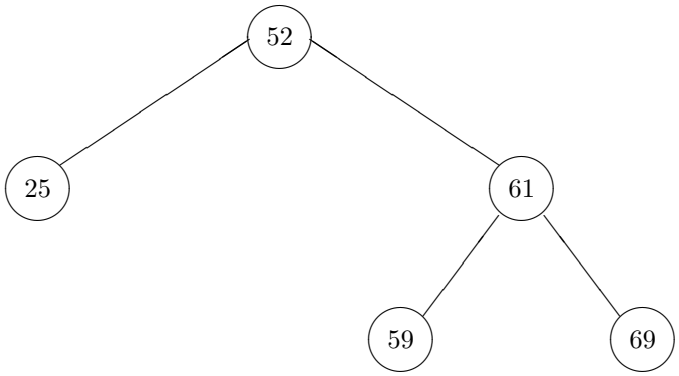


Abbildung 4.7: Der Ausgangsbaum

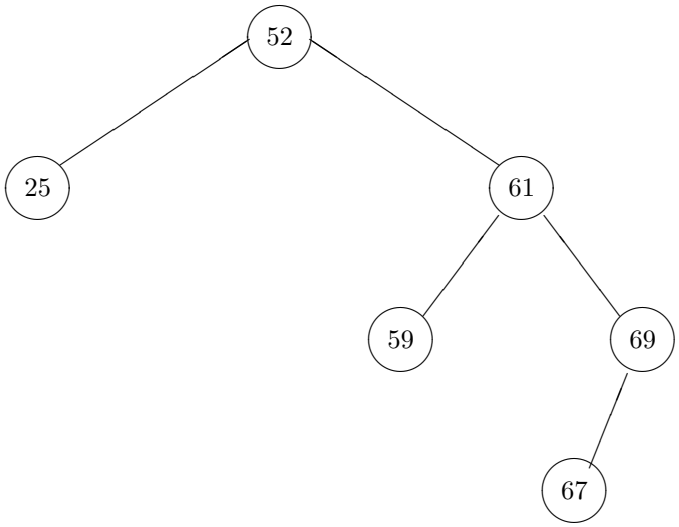


Abbildung 4.8: AVL Bedingung verletzt

Uhrzeigersinn. Der Knoten mit dem Wert 61 wird dabei die Wurzel. Knoten 59 wird nun zum frei werdenden rechten Sohn von 52 und der Baum ist wieder balanciert, siehe Abb. 4.9.

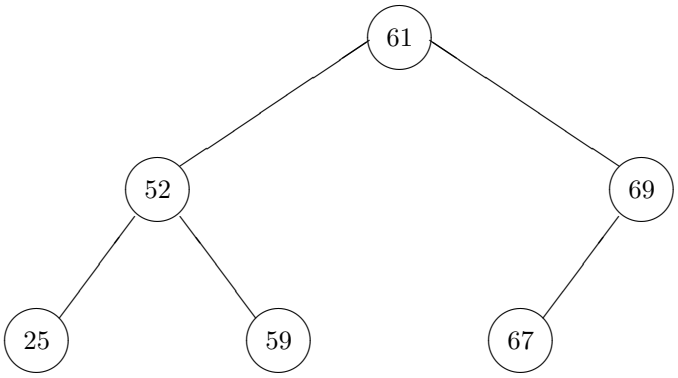


Abbildung 4.9: AVL Bedingung durch Links-Rotation wieder hergestellt

Die Verwendung eines AVL-Baumes ist sinnvoll, wenn nach einer aufwendigen Aufbauphase sehr viele Zugriffe auf die eingefügten Daten vorgenommen werden. Die Zugriffszeit ist $\mathcal{O}(\log(n))$, also die Höhe des binären balancierten Baumes; die Zeit zum balancieren beträgt ebenfalls $\mathcal{O}(\log(n))$. Löschen ist auch möglich in AVL-Bäumen. Die AVL-Bäume werden z. B. von Linux teilweise für die Verwaltung von Speicherplätzen verwendet (vm_area_struct).

Für diesen Algorithmus werden Rotationen benötigt, die in Abbildung 4.10 (Seite 94) dargestellt sind. Kreise stehen dabei für einzelne Ecken, Dreiecke für Teilbäume, der kleine dunkle Kasten für das neu eingefügte Element.

Wichtig sind die Bedingungen, wann eine Rotation vorgenommen werden muss:

1. **Rechtsrotation**: Die Höhe des Teilbaums `R1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `L` begründet.

2. **Linksrotation**: Die Höhe des Teilbaums `L1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `R` begründet.

3. **Problemsituation links**: (Doppelte Linksrotation) Die Höhe des Teilbaums `L1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `L2` begründet. Dieser Teilbaum ist in der Abbildung detaillierter mit Wurzel `C` und den beiden Teilbäumen `L2l` und `L2r` dargestellt. In diesem Fall ist zuerst in dem Teilbaum mit Wurzel `B` eine Rechtsrotation durchzuführen (siehe in der Abbildung die Darstellung `Dazwischen`) und dann in dem Baum mit Wurzel `A` eine Linksrotation durchzuführen.

4. **Problemsituation rechts**: (Doppelte Rechtsrotation) Die Höhe des Teilbaums `R1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `R2` begründet. Diese Situation ist in der Abbildung nicht dargestellt. Sei dieser Teilbaum detaillierter mit Wurzel `C` und den beiden Teilbäumen `R2l` und `R2r` dargestellt. In diesem Fall ist zuerst in dem Teilbaum mit Wurzel `B` eine Linkssrotation durchzuführen und dann in dem Baum mit Wurzel `A` eine Rechtsrotation durchzuführen.

Der Algorithmus selbst arbeitet (im Groben) so, dass er nach dem Einfügen eines Elementes „bottom“ up überprüft, ob eine der vier beschriebenen Situationen vorliegt. Es wird dann eine

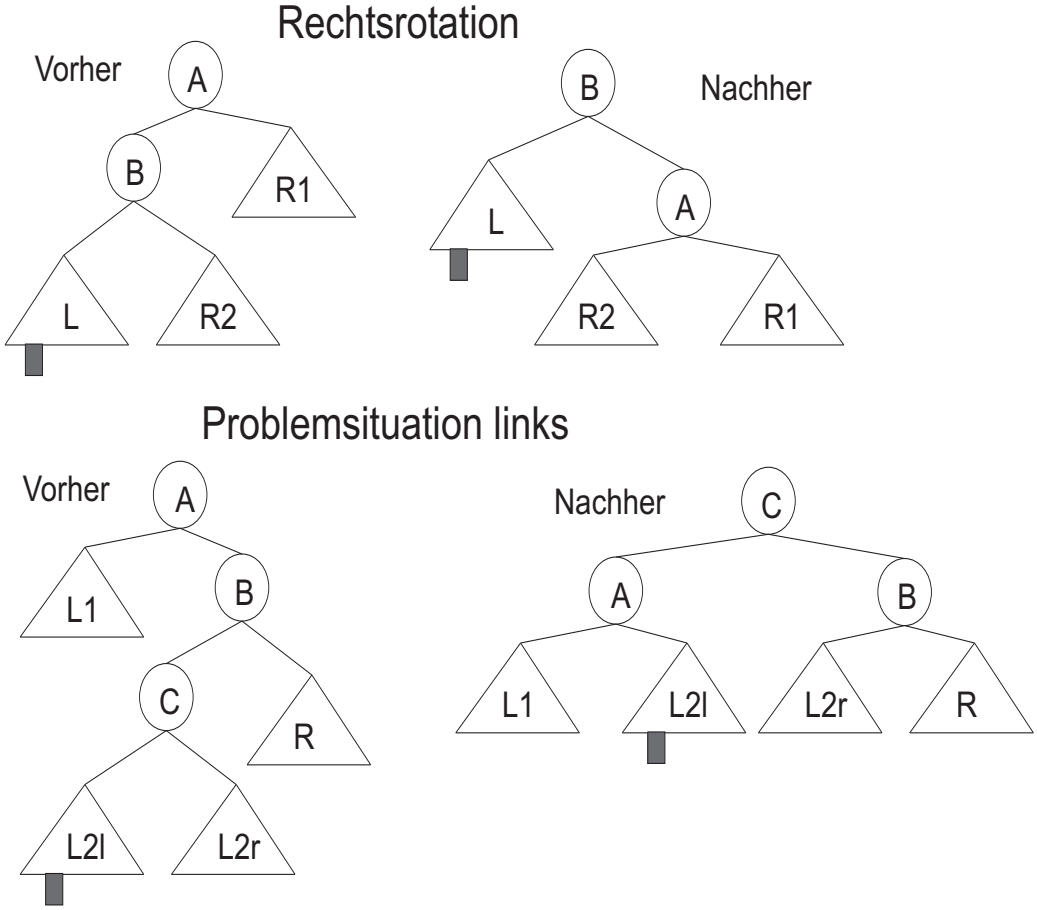


Abbildung 4.10: Die Rotationsarten eines AVL-Baumes

entsprechende Rotation bzw. Balancierung des Baumes vorgenommen. Diese Bedingung, dass der Algorithmus „bottom up“ arbeitet ist die zentrale Bedingung für AVL-Bäume! Die Stärke der AVL-Bäume liegt also darin, dass sie lokal agieren, also dort ausbalancieren, wo das neue Element eingefügt bzw. gelöscht wurde.

Die Teilbäume in Abbildung 4.10 (Seite 94) sind alle AVL-Bäume. Betrachten wir nun die Höhe des Baumes bzw. seiner Teilbäume für die Abbildung 4.10 genauer:

Rechtsrotation *Vorher*: Die Höhe von `R1` sei `n`. Dann ist die Höhe von `R2` gleich `n` oder `n - 1` und die Höhe von `L` gleich `n + 1`. Die Gesamthöhe ist `n + 3`.
Nachher: Die Höhe von dem Teilbaum mit Wurzel `A` ist `n + 1`, also genauso wie die Höhe von `L`! Die GesamtHöhe ist `n + 2`.

Linksrotation *Vorher*: Die Höhe von `L1` sei `n`. Dann ist die Höhe von `L2` gleich `n` oder `n - 1` und die Höhe von `R` gleich `n + 1`. Die GesamtHöhe ist `n + 3`.
Nachher: Die Höhe von dem Teilbaum mit Wurzel `A` ist `n + 1`, also genauso wie die Höhe von `R`! Die GesamtHöhe ist `n + 2`.

Problemsituation links *Vorher*: Die Höhe von `L1` sei `n`. Dann ist die Höhe von `R` gleich `n` oder `n - 1` und die Höhe von `L2` (Teilbaum mit Wurzel `C`) gleich `n + 1`. In der dargestellten Situation ist die Höhe von `L2l` `n`; die Höhe von `L2r` ist dann `n - 1`[3]. Die GesamtHöhe ist `n`

[3] Wäre die Höhe `n - 2` würde beim Aufbau des AVL-Baumes bereits an dieser Stelle eine Rechtsrotation durchgeführt werden. Die beschriebene Problemsituation würde nicht mehr auftreten!

`+ 3`.
Nachher: Die Höhe von dem Teilbaum mit Wurzel `A` ist `n + 1` und die Höhe von dem Teilbaum mit Wurzel `B` ist `n` oder `n + 1`. Die GesamtHöhe ist `n + 2`.

Problemsituation rechts Die Situation ist analog, bzw. genaugenommen eine Spiegelung, zu der Problemsituation links.

Das Löschen funktioniert nach folgendem Prinzip: In dem linken bzw. rechten Teilbaum des zu löschenden Knotens wird das grösste (linker Teilbaum) bzw. kleinste (rechter Teilbaum) Element an die Stelle des zu löschenden Knotens kopiert. Dann wird dieses Element gelöscht (rekursiv). Es sollte jeweils der höchste Teilbaum ausgewählt werden, um die Anzahl der Rotationen zu minimieren. Ist das zu löschende Element ein Blatt (Rekursionsende), so wird das Element gelöscht und der Baum (bis zur Wurzel!) „bottom up“ durchlaufen und ggf. ausgeglichen.

Bemerkung 4.16 (Balancierter oder nichtbalancierter Baum)

1. Im allgemeinen ist es sehr aufwendig, Datensätze in einem *balancierten* binären Baum anzuordnen. Falls Änderungen am Datenbestand ähnlich häufig vorkommen wie das Suchen nach bestimmten Datensätzen, verzichtet man auf die Balanciertheit des binären Baumes und versucht nur, zu erreichen, dass der entstehende Baum einem balancierten Baum „ähnlicher“ sieht als einer linearen Liste. Im Falle einer Liste der Telefonvorwahlen wird jedoch der Datenbestand nach der erstmaligen Festlegung nur noch selten geändert, aber häufig abgefragt. Hier lohnt sich der Aufwand für die Konstruktion eines balancierten Baumes wegen des dadurch erreichten optimalen Suchaufwandes.

2. Ein bewusster Übergang zu einem nichtbalancierten binären Baum bietet sich ausserdem immer dann an, wenn die Datensätze mit sehr unterschiedlicher Häufigkeit erfragt werden. Zu häufig aufgerufenen Datensätzen sollten von der Wurzel aus Wege aus wenigen Kanten führen, während selten benötigte Datensätze weiter von der Wurzel entfernt stehen sollten. Das obige Anordnungsprinzip lässt dann in der Regel keinen *balancierten* binären Baum zu.

◀

Aufgabe 4.17
Welches ist die Mindestanzahl von Knoten, die ein AVL-Baum der Höhe h hat? ◀

4.6 Rot-Schwarz-Bäume

Ein Rot-Schwarz-Baum ist eine vom binären Suchbaum abgeleitete Datenstruktur, welche sehr schnellen Zugriff auf die in ihr gespeicherten Werte garantiert. Rot-Schwarz Bäume wurden zuerst 1972 von Rudolf Bayer [Bay72] beschrieben, welcher sie *symmetric binary B-trees* nannte. Der heutige Name geht auf Leo I. Guibas und Robert Sedgewick zurück, welche 1978 die rot-schwarze Farbkonvention einführten. Die schnellen Zugriffzeiten auf die einzelnen im Rot-Schwarz-Baum gespeicherten Elemente werden durch fünf Eigenschaften erreicht, welche zusammen garantieren, dass ein Rot-Schwarz-Baum immer balanciert ist, wodurch die Höhe eines Rot-Schwarz-Baumes mit n Werten nie größer wird als $O(log_2 n)$. Somit können die wichtigsten Operationen in Suchbäumen - suchen, einfügen und löschen - garantiert in $O(log_2 n)$ ausgeführt werden.

4.6.1 Eigenschaften

Ein Rot-Schwarz-Baum ist ein (annähernd) ausgeglichener, binärer Suchbaum, in dem jeder Knoten eine Zusatzinformation - seine Farbe - trägt. Wie der Name vermuten lässt, arbeitet ein Rot-Schwarz-Baum mit den Farben seiner Knoten, um sich auszugleichen. Neben den Bedingungen, die an binäre Suchbäume gestellt werden, wird an Rot-Schwarz-Bäume jedoch noch die Forderung gestellt, folgende fünf Eigenschaften immer zu erfüllen:

Definition 4.18 (Rot-Schwarz-Baum)
Ein Rot-Schwarz-Baum ist ein Binärbaum, der folgende fünf Eigenschaften hat:

1. Jeder Knoten im Baum ist entweder rot oder schwarz.
2. Die Wurzel des Baums ist schwarz.
3. Jedes Blatt (NIL-Knoten) ist schwarz.
4. Kein roter Knoten hat ein rotes Kind.
5. Die Anzahl der schwarzen Knoten von jedem beliebigen Knoten zu einem Blatt (Schwarztiefe) ist auf allen Pfaden gleich.

◀

Durch diese fünf Bedingungen wird die wichtigste Eigenschaft von Rot-Schwarz-Bäumen sichergestellt: Der längste Pfad von der Wurzel zu einem Blatt ist nie mehr als doppelt so lang wie der kürzeste Pfad von der Wurzel zu einem Blatt. Hierdurch ist ein Rot-Schwarz-Baum immer annähernd balanciert. Da die Höhe dadurch minimiert wird, wird somit ebenfalls die Laufzeit der oben genannten Operationen minimiert. Somit kann man für Rot-Schwarz-Bäume - im Gegensatz zu normalen binären Suchbäumen - eine obere Schranke für die Laufzeit diese Operationen garantieren.

Um zu verstehen, warum diese fünf Eigenschaften eine obere Schranke für die Laufzeit garantieren, reicht es sich zu verdeutlichen, dass aufgrund der vierten Eigenschaft auf keinem Pfad zwei rote Knoten aufeinander folgen dürfen, weswegen sich auf dem längsten Pfad immer ein roter Knoten mit einem schwarzen Knoten abwechselt, während auf dem kürzesten Pfad nur schwarze Knoten vorhanden sind. Da die fünfte Eigenschaft jedoch festlegt, dass die Anzahl der schwarzen Knoten auf allen Pfaden gleich sein muss kann der Pfad, auf dem sich jeweils ein roter mit einem schwarzen Knoten abwechselt maximal doppelt so lang sein wie der Pfad auf dem nur schwarze Knoten sind.

Bemerkung 4.19
Während es auch möglich ist, Binärbäume zu betrachten, bei denen die Knoten nicht immer genau zwei Kinder haben müssen, betracht dieser Abschnitt der Einfachheit halber nur Bäume, welche immer genau zwei Kinder haben. Hierzu werden eventuell fehlende Kinder als schwarzes Blatt ohne Suchschlüssel (sog. null-Blatt) eingeführt. Somit sind alle Knoten mit Suchschlüssel innere Knoten (und haben genau zwei Kinder) und alle Blätter null-Knoten. ◀

4.6.2 Operationen

Suchen

Die Suchoperation erben Rot-Schwarz-Bäume von den allgemeinen binären Suchbäumen. Für eine genaue Beschreibung des Algorithmus siehe dort.

Einfügen

Das Einfügen in den Rot-Schwarz-Baum funktioniert wie das Einfügen in einen binären Suchbaum, wobei der neue Knoten rot gefärbt wird, damit die Schwarztiefe des Baumes erhalten bleibt. Nach dem Einfügen können jedoch eventuell die zweite oder - was wahrscheinlicher ist - die vierte Eigenschaft des Rot-Schwarz-Baumes verletzt sein, weswegen es nötig werden kann, den Baum zu reparieren. Hierbei unterscheidet man insgesamt fünf Fälle, welche im folgenden genauer betrachtet werden.

Bemerkung 4.20
Wenn im folgenden von Vater, Großvater und Onkel die Rede ist, so sind diese jeweils relativ zum neu einzufügenden Knoten (N) zu sehen.

In den Fällen 3 bis 5 kann angenommen werden, dass der einzufügende Knoten einen Großvater hat, da sein Vater rot ist, und somit nicht selbst die Wurzel sein kann (Die Wurzel des Baums

ist schwarz). Da es sich aber bei einem Rot-Schwarz-Baum um einen Binärbaum handelt, hat der Großvater auf jeden Fall noch ein Kind (auch wenn es sich bei diesem um einen null-Knoten handeln kann).

In den Fällen 4 bis 5 wird der Einfachheit halber angenommen, dass der Vaterknoten das linke Kind seines Vaters (also des Großvaters des einzufügenden Knotens) ist. Sollte er das rechte Kind seines Vaters sein, so müssen in den beiden folgenden Fällen jeweils links und rechts vertauscht werden. ◄

Fall 1 : Der neu eingefügte Knoten ist die Wurzel des Baumes. Da hierdurch die zweite Eigenschaft verletzt wird (Die Wurzel des Baums ist schwarz) färbt man die Wurzel einfach um. Da dieser Fall nur eintritt, falls man ein Element in den leeren Baum einfügt, braucht man sich nicht um weitere Reparaturen zu kümmern, da es im Baum nach dem Einfügen nur diesen einen Knoten gibt, weswegen keine der weiteren Eigenschaften verletzt werden kann.

Fall 2 : Der Vater des neuen Knotens ist schwarz. Hierdurch wird die fünfte Eigenschaft gefährdet, da der neue Knoten selbst wieder zwei schwarze NIL-Knoten mitbringt und somit die Schwarztiefe auf einem der Pfade um eins erhöht wird. Da der eingefügte Knoten selbst aber rot ist, und beim Einfügen einen schwarzen null-Knoten verdrängt hat, bleibt die Schwarztiefe auf allen Pfaden erhalten.

Fall 3 : Sowohl der Onkel als auch der Vater des neuen Knotens sind rot. In diesem Fall kann man beide Knoten einfach schwarz färben, und im Gegenzug den Großvater rot färben, wodurch die fünfte Eigenschaft wiederhergestellt wird. Durch diese Aktion wird das Problem um ein Level nach oben verschoben, da durch den nun rot gefärbten Großvater die zweite oder vierte Eigenschaft verletzten sein könnte, weswegen nun der Großvater betrachtet werden muss. Dieses Vorgehen wird solange rekursiv fortgesetzt, bis keine der Regeln mehr verletzt wird.

Fall 4 : Der neue Knoten hat einen schwarzen Onkel und ist das rechte Kind seines roten Vaters. In diesem Fall kann man eine Linksrotation um den Vater ausführen, welche die Rolle des einzufügenden Knotens und seines Vaters vertauscht. Danach kann man den ehemaligen Vaterknoten mit Hilfe des fünften Falles bearbeiten. Durch die oben ausgeführte Rotation wurde ein Pfad so verändert, dass er nun durch einen zusätzlichen Knoten führt, während ein anderer Pfad so verändert wurde, dass er nun einen Knoten weniger hat. Da es sich jedoch in beiden Fällen um rote Knoten handelt, ändert sich hierdurch an der Schwarztiefe nichts, womit die fünfte Eigenschaft erhalten bleibt.

Fall 5 : Der neue Knoten hat einen schwarzen Onkel und ist das linke Kind seines roten Vaters. In diesem Fall kann man eine Rechtsrotation um den Großvater ausführen, wodurch der ursprüngliche Vater nun der Vater von sowohl dem neu einzufügenden Knoten als auch dem ehemaligen Großvater ist. Da der Vater rot war, muss nach der vierten Eigenschaft (Kein roter Knoten hat ein rotes Kind) der Großvater schwarz sein. Vertauscht man nun die Farben des ehemaligen Großvaters bzw. Vaters, so ist in dem dadurch entstehenden Baum die vierte Eigenschaft wieder gewahrt. Die fünfte Eigenschaft bleibt ebenfalls gewahrt, da alle Pfade, welche durch einen dieser drei Knoten laufen, vorher durch den Großvater liefen, und nun alle durch den ehemaligen Vater laufen, welcher inzwischen - wie der Großvater vor der Transformation - der einzige schwarze der drei Knoten ist.

Löschen

Das Löschen eines Knotens aus einem Rot-Schwarz-Baum erfolgt analog zum Löschen eines Knotens aus binären Suchbäumen. Falls der zu löschende Knoten zwei Kinder hat (keine Null-Knoten), so sucht man entweder den maximalen Wert im linken Teilbaum oder den minimalen Wert im rechten Teilbaum des zu löschenden Knotens, schreibt diesen Wert in den eigentlich zu löschenden Knoten, und entfernt den gefundenen Knoten einfach aus dem Rot-Schwarz-Baum. Dies kann man immer ohne Probleme machen, da der gelöschte Knoten maximal ein Kind gehabt haben kann,

da sein Wert sonst nicht maximal respektive minimal gewesen wäre. Somit lässt sich das Problem auf das Löschen von Knoten mit maximal einem Kind vereinfachen.

Bemerkung 4.21
Im folgenden werden wir also nur noch Knoten mit mindestens einem Kind betrachten. Hierbei werden wir eventuelle null-Knoten falls nötig ebenfalls als Kinder bezeichnen. (falls der Knoten sonst keine weiteren Kinder haben sollte). ◄

Will man einen roten Knoten löschen, so kann man diesen einfach durch sein Kind ersetzen, welches nach der vierten Eigenschaft (Kein roter Knoten hat ein rotes Kind) schwarz sein muss. Da der Vater des gelöschten Knotens ebenfalls aufgrund derselben Eigenschaft schwarz gewesen sein muss, wird die vierte Eigenschaft somit nicht mehr verletzt. Da alle Pfade, die ursprünglich durch den gelöschten roten Knoten verliefen, nun durch einen roten Knoten weniger verlaufen, ändert sich an der Schwarztiefe ebenfalls nichts, und die fünfte Eigenschaft (Die Anzahl der schwarzen Knoten von jedem beliebigen Knoten zu einem Blatt ist auf allen Pfaden gleich) bleibt auch erhalten. Ebenfalls noch einfach abzuarbeiten ist der Fall, dass der zu löschende Knoten schwarz ist, aber ein rotes Kind hat. Würden in diesem Fall einfach der schwarze Knoten gelöscht werden, könnte dadurch sowohl die vierte als auch die fünfte Eigenschaft verletzt werden, was jedoch umgangen werden kann, indem das Kind schwarz gefärbt wird. Somit treffen garantiert keine zwei roten Knoten aufeinander (der eventuell rote Vater des gelöschten Knotens und sein rotes Kind) und alle Pfade, welche durch den gelöschten schwarzen Knoten verliefen, verlaufen nun durch sein schwarzes Kind, wodurch beide Eigenschaften erhalten bleiben.

Falls sowohl der zu löschende Knoten als auch sein Kind schwarz sind, ersetzt man zuerst den zu löschenden Knoten mit seinem Kind, und löscht danach den Knoten. Nun verletzt dieser Knoten (im folgenden Konfliktknoten genannt) jedoch die Eigenschaften eines Rot-Schwarz-Baumes, da es nun einen Pfad gibt welcher vorher durch zwei schwarze Knoten führte, jetzt aber nur noch durch einen führt. Somit ist die fünfte Regel (Die Anzahl der schwarzen Knoten von jedem beliebigen Knoten zu einem Blatt ist auf allen Pfaden gleich) verletzt. Je nach Ausgangslage werden nun sechs verschiedene Fälle unterschieden wie der Baum wieder zu reparieren ist, welche im folgenden genauer betrachtet werden.

Bemerkung 4.22
Wenn im folgenden von Vater (parent), Bruder (sibling), Großvater (grandparent) und Onkel (uncle) die Rede ist, so sind diese jeweils relativ zum ehemaligen Kind des zu löschenden Knoten N (Konfliktknoten) zu sehen, welcher nach dem Platztausch jetzt an der Stelle steht an der der zu löschende Knoten selbst ursprünglich stand.

Für die Fälle 2, 5 und 6 sei der Konfliktknoten (N) das linke Kind seines Vaters. Sollte er das rechte Kind sein, so müssen in den drei Fällen jeweils links und rechts vertauscht werden. ◄

Fall 1 : Der Konfliktknoten (N) ist die neue Wurzel. In diesem Fall ist man fertig, da ein schwarzer Knoten von jedem Pfad entfernt wurde und die neue Wurzel schwarz ist, womit alle Eigenschaften erhalten bleiben.

Fall 2 : Der Bruder (S) des Konfliktknotens ist rot. In diesem Fall kann man die Farben des Vaters und des Bruders des Konfliktknotens invertieren und anschließend eine Linksrotation seinen Vater ausführen, wodurch der Bruder des Konfliktknotens zu dessen Großvater wird. Alle Pfade haben weiterhin die selbe Anzahl an schwarzen Knoten, aber der Konfliktknoten hat nun einen schwarzen Bruder und einen roten Vater, weswegen man nun zu Fall 4, 5, oder 6 weitergehen kann.

Fall 3 : Der Vater (P) des Konfliktknotens, sein Bruder (S) und die Kinder seines Bruders (SL respektive SR) sind alle schwarz. In diesem Fall kann man einfach den Bruder rot färben, wodurch alle Pfade die durch diesen Bruder führen - welches genau die Pfade sind welche nicht durch den Konfliktknoten selbst führen - einen schwarzen Knoten weniger haben, wodurch die ursprüngliche Ungleichheit wieder ausgeglichen wird. Jedoch haben alle Pfade welche durch den Vater laufen nun einen schwarzen Knoten weniger als jene Pfade die nicht

durch den Vater laufen, wodurch die fünfte Eigenschaft immer noch verletzt wird. Um dies zu reparieren versucht man nun den Vaterknoten zu reparieren indem man versucht einen der sechs Fälle - angefangen bei Fall 1 - anzuwenden.

Fall 4 : Sowohl der Bruder des Konfliktknotens als auch die Kinder des Bruders (SL respektive SR) sind schwarz, aber der Vater (P) des Konfliktknotens rot. In diesem Fall reicht es aus, die Farben des Vaters und des Bruders zu tauschen. Hierdurch bleibt die Anzahl der schwarzen Knoten auf den Pfaden welche nicht durch den Konfliktknoten laufen unverändert, fügt aber einen schwarzen Knoten auf allen Pfaden welche durch den Konfliktknoten führen hinzu, und gleicht somit den gelöschten schwarzen Knoten auf diesen Pfaden aus.

Fall 5 : Das linke Kind (SL) des Bruders (S) ist rot, das rechte Kind (SR) wie auch der Bruder des Konfliktknotens (N) sind jedoch schwarz und der Konfliktknoten selbst ist das linke Kind seines Vaters. In diesem Fall kann man eine Rechtsrotation um den Bruder ausführen, sodass das linke Kind (SL) des Bruders dessen neuer Vater wird, und damit der Bruder des Konfliktknotens wird. Danach vertauscht man die Farben des Bruders und seines neuen Vaters. Nun haben alle Pfade immer noch die gleiche Anzahl an schwarzen Knoten, aber der Konfliktknoten hat einen schwarzen Bruder dessen rechtes Kind rot ist, womit man nun zum sechsten Fall weitergehen kann. Weder der Konfliktknoten selbst noch sein Vater werden durch diese Transformation beeinflusst.

Fall 6 : Der Bruder (S) des Konfliktknotens (N) ist schwarz, das rechte Kind des Bruders (SR) rot und der Konfliktknoten selbst ist das linke Kind seines Vaters. In diesem Fall kann man eine Linksrotation um den Vater des Konfliktknotens ausführen, sodass der Bruder der Großvater des Konfliktknotens, und der Vater seines ehemaligen rechten Kindes (SR) wird. Nun reicht es die die Farben des Bruders und des Vaters des Konfliktknotens zu tauschen und das rechte Kind des Bruders schwarz zu färben. Der Unterbaum hat nun in der Wurzel immer noch die selbe Farbe wodurch die vierte Eigenschaft erhalten bleibt. Aber der Konfliktknoten hat nun einen weiteren schwarzen Vorfahren: Falls sein Vater vor der Transformation noch nicht schwarz war, so ist er nach der Transformation schwarz, und falls sein Vater schon schwarz war, so hat der Konfliktknoten nun seinen ehemaligen Bruder (S) als schwarzen Großvater, weswegen die Pfade welche durch den Konfliktknoten laufen nun einen zusätzlichen schwarzen Knoten passieren.

Falls nun ein Pfad nicht durch den Konfliktknoten verläuft, so gibt es zwei Möglichkeiten:

- Der Pfad verläuft durch seinen neuen Bruder. Ist dies der Fall, so muss der Pfad sowohl vor als auch nach der Transformation durch den alten Bruder (S) und den neuen Vater des Konfliktknotens laufen. Da die beiden Knoten aber nur ihre Farben vertauscht haben ändert sich an der Schwarztiefe auf dem Pfad nichts.
- Der Pfad verläuft durch den neuen Onkel des Konfliktknotens welcher das rechte Kind des Bruders (S) ist. In diesem Fall ging der Pfad vorher sowohl durch seinen Bruder, dessen Vater, und das rechte Kind des Bruders(SR). Nach der Transformation geht er aber nur noch durch den Bruder (S) selbst - welcher nun die Farbe seines ehemaligen Vaters angenommen hat - und das rechte Kind des Bruders, welches seine Farbe von rot auf schwarz geändert hat. Insgesamt betrachtet hat sich an der Schwarztiefe dieses Pfades also nichts geändert.

In beiden Fällen verändert sich die Anzahl der schwarzen Knoten auf den Pfaden also nicht, wodurch die vierte Eigenschaft wiederhergestellt werden konnte.

Höhenbeweis

Wie schon in der Einleitung motiviert ist die besondere Eigenschaft von Rot-Schwarz Bäumen dass sie in logarithmischer Zeit - genauer in $O(log_2 n)$ - ein Element im Baum suchen, löschen oder

einfügen können. Diese Operationen sind auf allen binären Suchbäumen von der Höhe h des Baumes abhängig. Je niedriger nun die Höhe des Baumes ist, desto schneller laufen die Operationen. Kann man nun beweisen dass ein binärer Suchbaum mit n Elementen nie eine gewisse Höhe (im Falle des Rot-Schwarz Baumes 2 $log_2(n+1)$) überschreitet, so hat man bewiesen dass die oben genannten Operationen im schlimmsten Fall logarithmische Kosten haben, nämlich die genannten Kosten von $2log_2(n+1)$ für einen Baum in dem n Elemente gespeichert sind. Somit muss gezeigt werden, dass folgende Aussage gilt:

Satz 4.23
Für die Höhe h eines Ein Rot-Schwarz-Baumes, der n Schlüssel speichert, gilt: $h = 2log_2(n+1)$ ◀

Beweisidee: Zum Beweis dieser Eigenschaft muss man zuerst einen Hilfssatz über die Anzahl der inneren Knoten im Baum beweisen, und verbindet diese später mit der vierten Eigenschaft von Rot-Schwarz Bäumen (es folgen nie zwei rote Knoten aufeinander) um die oben genannte Eigenschaft zu beweisen.

4.7 B-Bäume

4.7.1 Übersicht

B-Bäume sind einerseits ein im Nachhinein naheliegende Verallgemeinerung von Binärbäumen, andererseits bringen sie einige „Komplikationen“ mit sich. Eines ihrer wichtigen Einsatzgebiete ist es, bei der Unterstützung der Indexierung von Plattendateien mitzuwirken. Daher haben sie auch eine andere Struktur, als diejenigen, die wir zum Suchen in Hauptspeicherbereichen betrachtet haben. Abbildung 4.11 zeigt die prinzipieller Architektur von Platten. Platten sind in vielen Fällen

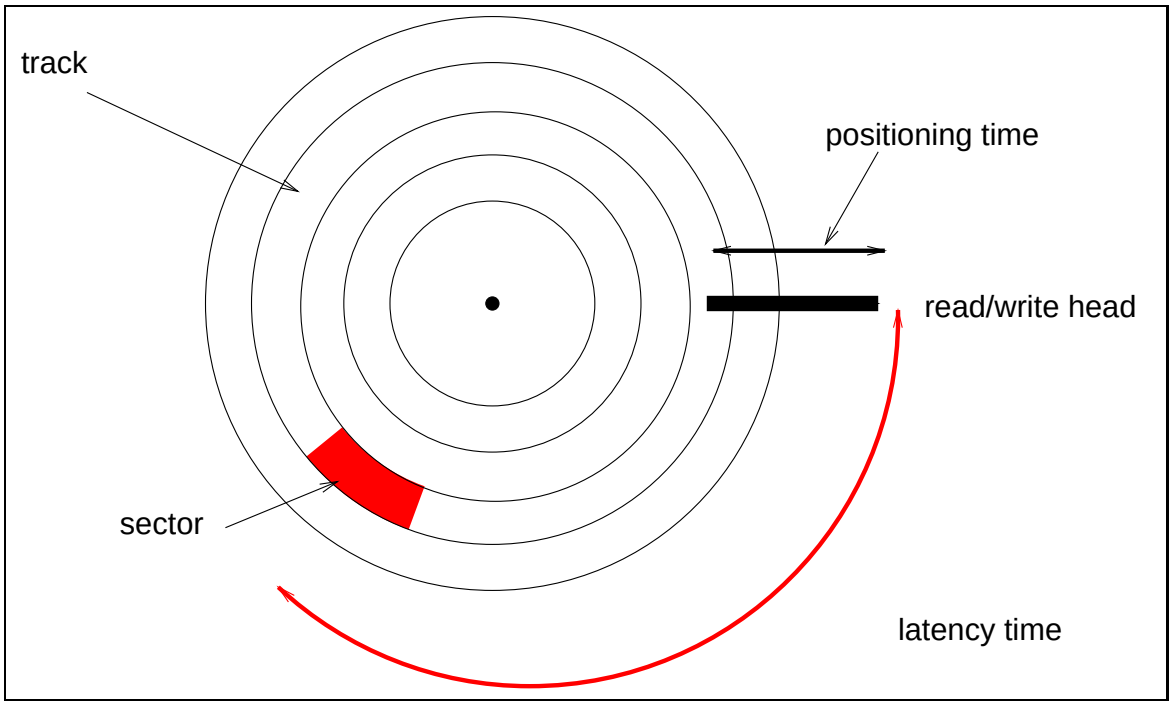


Abbildung 4.11: Faktoren beim Plattenzugriff

in *Spur*en (tracks) unterteilt. Jede Spur besteht aus *Sektor*en. Gelesen wird von einer Platte mittels Magnetköpfen, wobei wir uns hier nicht den Kopf darüber zerbrechen, ob es mit Lese-, Schreib-

oder Schreib-Leseköpfen zu tun haben. Der Sektor ist die kleinste Übertragungseinheit. Um auf einen Sektor zugreifen zu können, muss der Kopf zunächst auf der Spur positioniert werden. Die hierfür erforderliche *Positionierungszeit* bildet die eine Komponente der tatsächlichen Zugriffszeit. Ist der Kopf auf der Spur positioniert, so muss abgewartet werden, bis der gesuchte Sektor unter dem Kopf erscheint. Das Zeitintervall von der abgeschlossen Positionierung über der Spur bis zum Erscheinen des Beginns des Sektors unter dem Kopf heißt *Latenzzeit*.

B-Bäume wurden entwickelt um einfach suchen zu können, wenn die Suchstruktur nicht ganz in den Hauptspeicher passt. Ich zeige hier die Theorie und so weit dies möglich ist, gebe ich einige Hinweise zu ihrer Anwendung in Datenbanken.

4.7.2 Lernziele

- Den Begriff des B-Baums kennen
- Die wesentlichen Operationen auf B-Bäumen kennen
- Grundprinzipien der implementierung von B-Bäumen kennen
- Anwendungen von B-Bäumen kennen.

4.7.3 B-Bäume: Grundbegriffe

In Kap. 3 haben wir zunächst Algorithmen betrachtet, mit denen Strukturen im Hauptspeicher sortiert werden können. Zum Abschluss haben wir Mergesort als eine Möglichkeit zum Sortieren größerer Datenbestände betrachtet, die nicht mehr im Ganzen im Hauptspeicher bearbeitet werden können. Auch beim Suchen muss man diese Fälle unterscheiden. Wir betrachten nun einen Spezialfall: In (relationalen) Datenbanken verwendet man Indizes, um den Zugriff über ausgewählte Zugriffspfade zu unterstützen. Logisch handelt es sich bei einem Index um eine Datei, in der Schlüsselwerte sortiert abgelegt sind und zu jedem Schlüsselwert ein oder mehrere Pointer irgend welcher Art auf Daten gespeichert sind. Ein gegebenen Schlüsselwert ist so leicht und effizient zu finden und anschließend können die Daten sehr einfach gelesen werden. Was passiert aber nun im Prinzip, wenn ein Zugriff auf eine Datenbank erfolgt, etwa mit einem SELECT Befehl? Im Prinzip läuft dabei folgendes ab:

- Das DBMS entscheidet über einen geeigneten Zugangspfad.

  Handelt es sich um ein `SELECT` auf einer Tabelle ohne `WHERE` oder `ORDER BY` Bedingung, so kann es eine sinnvolle Entscheidung sein, die Tabelle einfach unter Umgehung aller Indizes physisch zu lesen.

  Je nach dem, welche Spalten in der `WHERE`- bzw. `ORDER BY`-Bedingung verwendet werden und wie die Werte eingegrenzt werden, kann einer der Indizes als geeigneter Zugriffspfad ausgewählt werden.

- Der geeignete Index wird gelesen. Wenn Indexteile in der `WHERE`-Bedingung vorkommen, kann bereits hier selektiert werden. Das Lesen bedeutet genauer: Einlesen eine Indexblocks, wenn er noch nicht im Hauptspeicher ist. Dann wird dieser Block durchsucht und ggf. weitere Blöcke geladen usw.
- Lesen der Daten, d. h. einlesen Datenblock, falls noch nicht im Hauptspeicher.
- Rückgabe der ausgewählten Spalten der selektierten Zeilen an die Anwendung.

Bei einem vollständigen Binärbaum, haben wir es mit einer Höhe von $log\, n$ zu tun, wenn der Baum n-Knoten hat. Im schlechtesten Fall kann man befürchten, auch entsprechend viele Indexblöcke lesen zu müssen um einen gesuchten Indexwert zu finden und dann auf die Daten zugreifen zu können. Das wäre für praktische Zwecke völlig unakzeptabel. Hier nun kommt die Idee von Edward M. McCreight und Rudolf Bayer zum tragen: Man definiere einen m-Weg Baum: Jeder Knoten

kann m Nachfolger haben. In den Blattknoten finden wir zu Schlüsseln Verweise auf die zugehörigen Datensätze. Darüber geschichtet liegt eine möglichst flache Hierarchie innerer Knoten. Ein innerer Knoten hat Verweise auf seine maximal m Nachfolger. Die Struktur wird also ähnlich aussehen, wie in Abb. 4.12. Das Wachstum eines B-Baums wird man sich also in etwa folgendermaßen vorstellen.

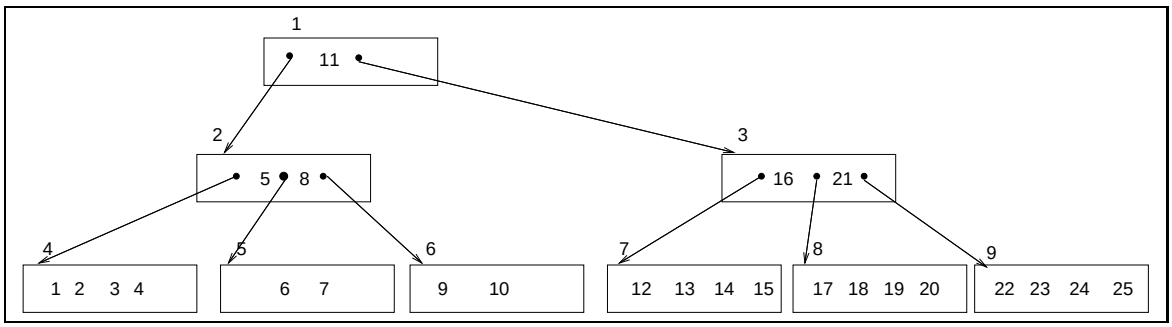


Abbildung 4.12: Ein B-Baum mit kleinem Grad

Zunächst ist der Baum leer. Mit dem ersten eingefügten Knoten wird ein externer Knoten und der Wurelknoten mit einem Verweis auf den Nachfolger eingefügt. Das kann nun eine Weile so weitergehen, bis ein Knoten voll wird. Nehmen wir mal an, dies sei ein Blattknoten. Dann wird dieser in zwei Blöcke auf gespalten, die Schlüssel dem Median entsprechend je zur Hälfte auf die beiden verteilt. Im Wurzelknoten entsteht so ein neuer Verweis. Ist auch der Wurzelknoten voll, so wird er ebenso gespalten und es entsteht eine weitere Ebene mit einem neuen Wurzelknoten. Ein B-Baum hat also zwei Merkmale, die ihn von allen bisher dahin betrachteten Bäumen unterscheiden:

- Sie wachsen von unten nach oben, nicht von oben nach unten. Rudolf Bayer berichtet in [BD02], er habe inzwischen eine Pflanze, die Drachenweide (english fishtail willow) gefunden, der ähnlich zu wachsen scheint. Ich bin kein Botaniker und habe dies nicht selber nachgeprüft.
- Aufgrund des Vorgehens beim Aufspalten kann es vorkommen, dass ein B-Baum zu 50% leer ist. Zur Zeit ihrer ersten Definition war diese Verschwendung teueren Plattenspeichers ein Sakrileg.

Von daher wurde das Prinzip zunächst als völlig unpraktikabel abgelehnt. Die erste Zeitschrift, in der der Artikel veröffentlicht werden sollte, wies das Konzept als abstrus zurück. Durch den Einsatz von Don Knuth wurde er dann in der renomierten Zeitschrift acta informatica veröffentlicht [BM72]. Noch 1984 veröffentlichte der Scientific American einen Artikel über B-Bäume, der völlig irreführende Abbildungen enthielt. Ein Beispiel zeigt Abb. 4.13

Definition 4.24 (B-Baum)
Sei $h \in \mathbb{N}, h \geq 0$. Ein *B-Baum* ist ein *Wurzelbaum* mit folgenden Eigenschaften. Dabei sei x.n die Anzahl Schlüssel in einem Knoten

1. Jeder *Knoten* x enthält $x.n$ Schlüssel in monoton wachsender Folge $x.k_1 \leq x.k_2 \leq \cdots \leq x.k_{x.n}$
2. Ist x ein interner *Knoten*, so enthält er $x.n + 1$ Zeiger $x.p_1, x.p_2, \ldots, x.p_{x.n+1}$ auf seine Nachfolger.
3. Die Schlüssel zerlegen die Schlüsselbereiche der Teilbäume: Ist k_i ein Schlüssel aus dem Teilbaum unter $x.p_i, i = 1, \ldots, x.n + 1$, so gilt:

$$k_1 \leq x.k_1 \leq k_2 \leq x.k_2 \leq \cdots \leq x.k_{x.n} \leq k_{x.n+1}$$

4. Jedes Blatt hat dieselbe Tiefe h.

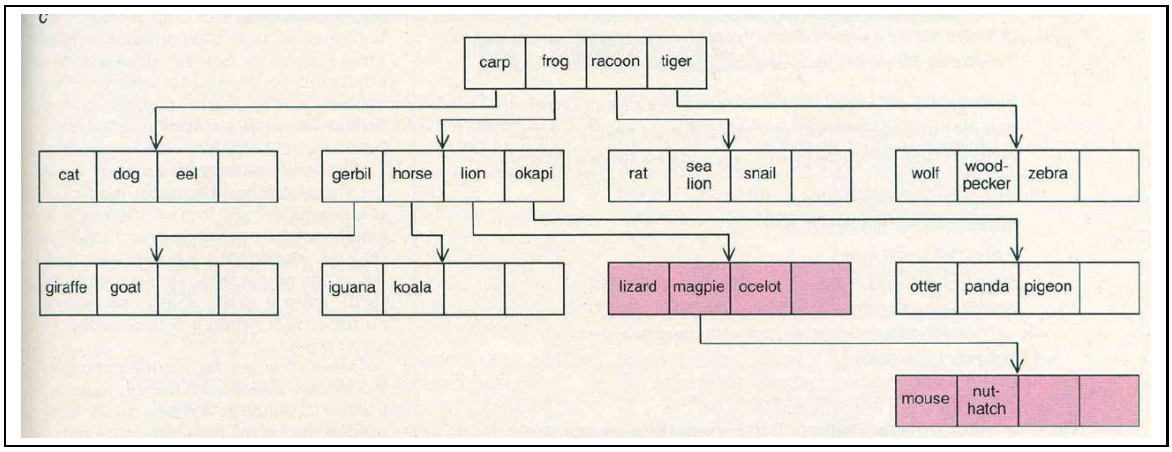


Abbildung 4.13: Aus dem Scientific American

5. Die Anzahl Schlüssel in einem Knoten ist nach unten und oben durch $t > 1$ beschränkt:
   - Jeder Knoten außer der Wurzel enthält mindestens $t - 1$ Schlüssel.
   - Jeder Knoten kann höchstens $2t - 1$ Schlüssel enthalten.
   - Wenn der Baum nicht leer ist, enthält die Wurzel mindestens einen Schlüssel.

Dabei heißt t *Grad*, *Ordnung* oder *Klasse* des B-Baums. [CLR94] ◄

Abbildung 4.14 zeigt ein Beispiel eines B-Baums der Ordnung 2.

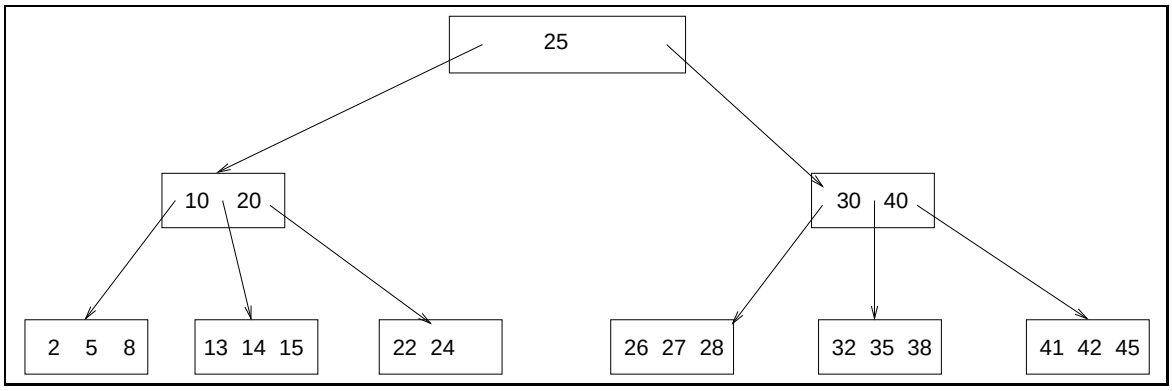


Abbildung 4.14: Ein B-Baum der Ordnung 2

Bemerkung 4.25 (Abweichende Definition)
Es gibt auch eine leicht abweichende Definition, die eine gerade maximale Anzahl von Schlüsseln in einem Knoten zulässt. In diesem Fall wird die Ordnung als M definiert, die Untergrenze ist dann $\frac{M}{2}$, die Obergrenze M. [Sed02] ◄

Satz 4.26 (Höhe B-Baum)
Zwischen der Höhe h, der Ordnung t und der Anzahl Schlüssel n in einem B-Baum gilt die folgende Beziehung:

$$h \leq log_t \frac{n+1}{2}.$$

◀

Beweis: Ist die Höhe h des Baumes gegeben, so ist die Minimal-Anzahl der Knoten (aufgrund der Untergrenze) gegeben, wenn die Wurzel einen Schlüssel und alle anderen Knoten $t-1$ Schlüssel enthalten. In diesem Fall gibt es auf der Ebene i $2t^{i-1}$ Knoten, $i = 1, \ldots, h$. Für die Anzahl n der Schlüssel gilt also

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1} h2t^{i-1} \\ &= 1 + 2(t-1)(\frac{t^h - 1}{t-1}) \\ &= 2t^h - 1 \end{aligned}$$

$\Longrightarrow$

$$t^h \leq \frac{n+1}{2}$$

$\Longrightarrow$

$$h \leq log_t \frac{n+1}{2}$$

q.e.d.

Aus Satz 4.26 folgen viele der positiven Eigenschaften von B-Bäumen. Man erkennt z. B. sofort, dass die Suche über die Knoten im Wesentlichen ein $O(log_t n)$ ist. Hinzu kommt natürlich noch die Suche innerhalb eines Knotens, die wir später abschätzen.

Bereits aus dieser Definition kann man Einiges für den praktischen Einsatz ableiten. Suchen wir nach einem Schlüssel, so setzt sich die Zeit hierfür etwa so zusammen:

- Konstante Zeit für den Zugriff auf einen Block α (Latenzzeit)
- Übertragungszeit pro Blockeintrag β
- Zugriffszeit innerhalb eines Blocks $\gamma\ ln(\nu k + 1)$, $1 \leq \nu \leq 2t - 1$ ist hier der Kehrwert der Seitenbelegungsquote, die ja zwischen 50% und 100% liegt.

Wir nehmen an, dass die Anzahl Seiten, die innerhalb einer Transaktion gelesen und geschrieben werden müssen, proportional zur Höhe h des Baums ist. Für die Zeit, die für die Zugriffe auf den B-Baum benötigt wird, gilt also

$$T \approx \delta h(\alpha + \beta(2k+1) + \gamma\ ln(\nu k + 1))$$

Approximiert man h durch $\log_{\nu k+1}(I+1)$, I die Indexgröße, so ist

$$T_a \approx \delta \log_{\nu k+1}(I+1)(\alpha + \beta(2k+1) + \gamma\ ln(\nu k + 1))$$

Hieraus kann man mit einigem Aufwand nachrechnen, dass T_a minimal wird, wenn k so gewählt wird, das gilt:

$$\frac{\alpha}{\beta} = 2(\frac{\nu k + 1}{\nu} ln(\nu k + 1)) - (2k+1)$$

Dies ergibt heute etwa $k \approx 1000$

4.7.4 Operationen auf B-Bäumen

Suchen in B-Bäumen

Das Suchen in B-Bäumen ist eine einfache Verallgemeinerung des Suchens in Binärbäumen. Wir müssen uns zunächst überlegen, wie wir die Suche in Binärbäumen auf n-Weg-Bäume so umformulieren, dass n Ausgänge möglich sind.

Ungefähr so wäre dies zu formulieren

```
class BTree ...
{
   void Node search(Key k, Node x)
   {
        int i = 1;
        while((x.k[i]< k)and (i<=x.n))
           i++;
        if(x.k[i]==k)
           fertig
        if(x.leaf) //nichts gefunden
           return null
        diskread(p[i]); //Folgeknoten lesen
        return search(k,x.p[i]);
    }
}
```

Der vorstehende Pseudocode verwendet lineare Suche. Dies können wir natürlich durch binäre Suche deutlich verbessern. Abbildung 4.15 zeigt das Aktivitätsdiagramm hierzu aus [BM72] in

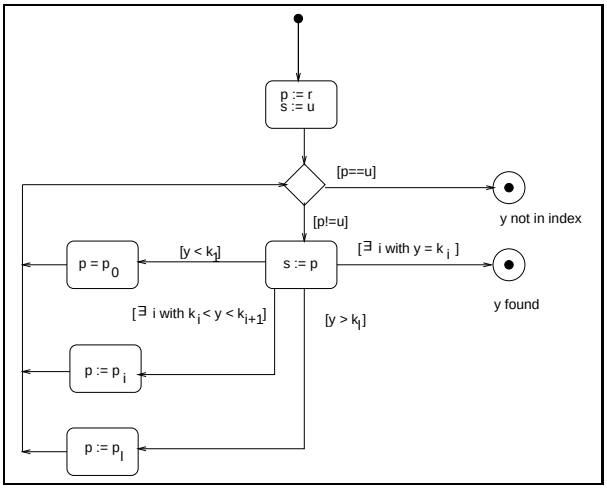


Abbildung 4.15: Suchen im B-Baum

aktueller Notation. Dabei sind p,r s pointer (oder Referenz-) Variablen, die auch undefiniert (u) sein können. r zeigt auf die Wurzel und ist = u, wenn der Baum leer ist. s wird erst beim Insert verwendet.

Einfügen in B-Bäume

Als erstes müssen wir einen leeren Baum anlegen.

```
void bTreeCreate()
{
```

```
  root = new node();
  root.leaf = true;
  diskWrite(root);
}
```

Beim Einfügen in einen B-Baum sind verschiedene Fälle zu unterscheiden. Ich werde für jeden Fall einen Algorithmus zum Einfügen angeben. Ich weise aber bereits jetzt darauf hin, dass in den Details abweichende Algorithmen gewählt werden können.

1. Einfügen eines Schlüssels in ein Blatt b mit $b.n < 2t - 2$. In diesem Fall wird der Schlüssel einfach eingefügt.

2. Einfügen eines Schlüssels in ein Blatt mit $b.n = 2t - 2$. In diesem Fall wird der Knoten durch das Einfügen voll und deshalb in zwei Blöcke aufgespalten. Ein Beispiel zeigt Abb. 4.16. Wird der Knoten darüber dadurch noch nicht voll, so sind wir fertig. Andernfalls muss

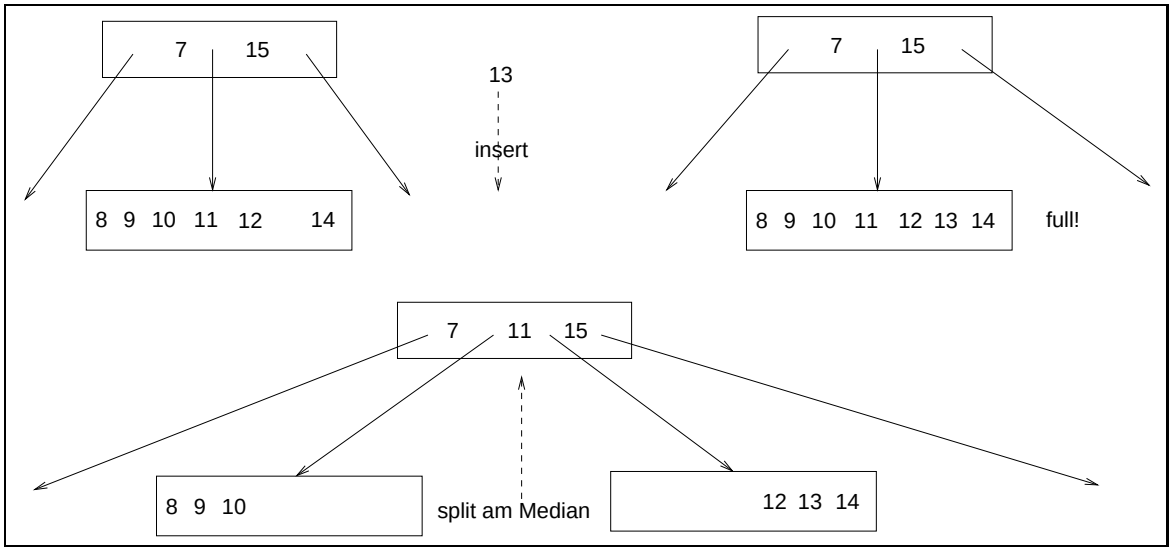


Abbildung 4.16: Einfügen in B-Baum, Oberer Knoten nicht voll

das Aufspalten fortgesetzt werden. Ist auch der Wurzelknoten voll, so wächst der Baum um eine Ebene. Die Entscheidung, wann ein Split erfolgt kann natürlich auch anders getroffen werden. Einige elementare Möglichkeiten, die sich nicht ausschließen, sind:

- Splitten erst wenn der Knoten tatsächlich voll ist.
- Gar nicht Splitten, sondern den Knoten in eine „to split Queue“ einfügen und den Split vornehmen, wenn der Knoten ansonsten nicht benutzt wird.

Man achte darauf, dass wir hier über das Einfügen von Schlüsseln sprechen. Es ist Aufgabe des Einfügealgorithmus einen Schlüssel so in den Baum einzufügen, dass die B-Baum-Eigenschaften erhalten bleiben. Die Obergrenze $2t-1$ sorgt dabei dafür, dass der Baum gleichmäßig wächst und nicht etwa einzelne Knoten viele Schlüssel enthalten und die Untergrenze $t - 1$ verhindert, dass andere sehr wenige Schlüssel enthalten.

Abbildung 4.17 zeigt den Ablauf des Einfügens. Dabei ist s eine Variable, die vom Suchalgorithmus auf den zu letzt gelesenen Knoten (Seite) gesetzt wird. Die folgenden drei Abbildungen zeigen ein Beispiel hierzu. Lediglich der Split der Wurzel fehlt in dieser Darstellung noch.

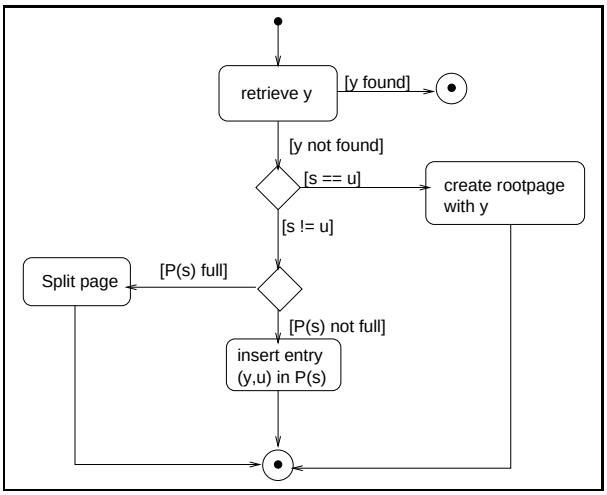


Abbildung 4.17: Einfügen in B-Baum

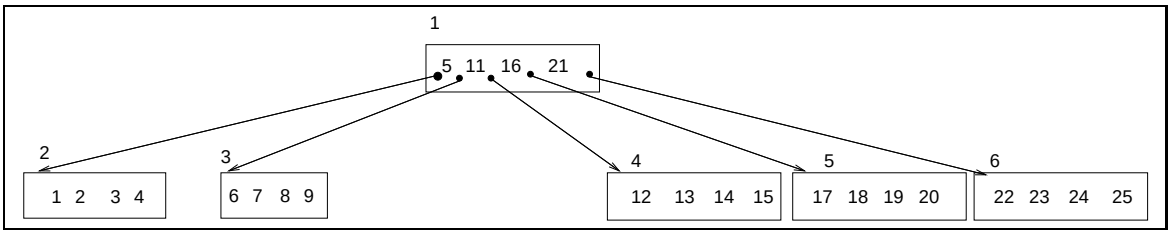


Abbildung 4.18: Eine 8 fällt „vom Himmel“

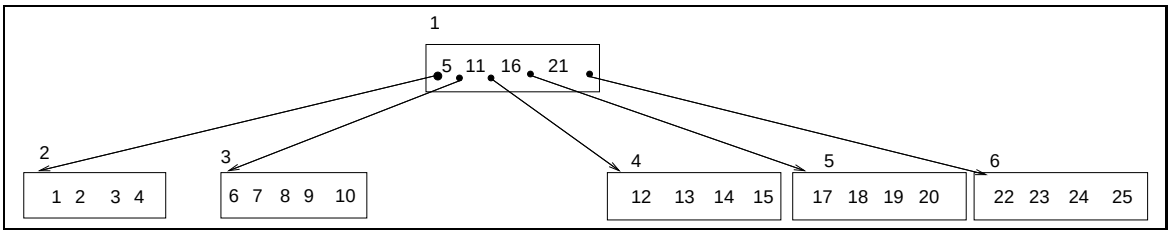


Abbildung 4.19: Knoten 3 wird voll

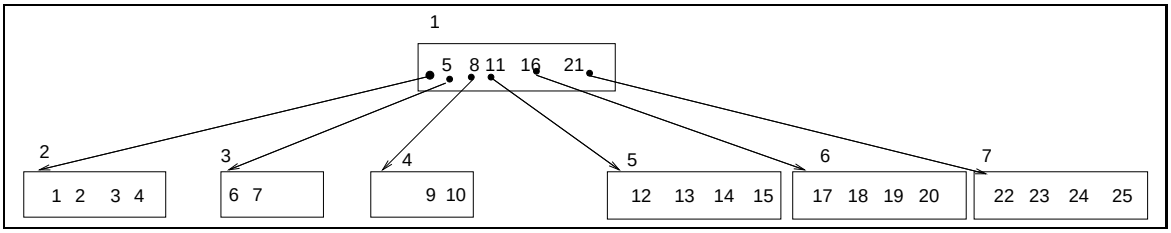


Abbildung 4.20: Split, Wurzel voll

Löschen in B-Bäumen

Ähnlich wie beim Einfügen die Obergrenze sorgt beim Löschen die Untergrenze $t - 1$ für die Anzahl Schlüssel in einem Knoten für die Balance. Sie verhindert, das besonders „magere“ Knoten entstehen. Hier der Ablauf des Löschens als Aktivitätsdiagramm in Abb. 4.21.

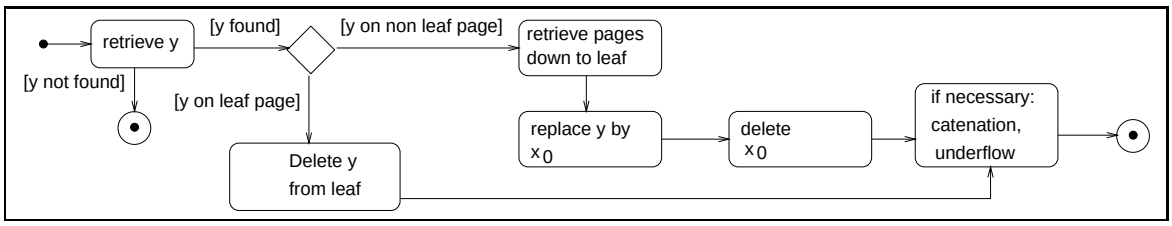


Abbildung 4.21: Löschen im B-Baum

4.7.5 Einsatz

In diesem Abschn. beschreibe ich (zumindest) eine der wichtigsten Anwendungen von B-Bäumen. Um gezielt auf Sätze in einer Datenbanktabelle mit bestimmten Werten in einigen Spalten zugreifen zu können, benötigt man einen Index. In einer Indexdatei werden Werte gespeichert und zu einem Wert eine Adresse, Pointer o. ä. was den schnellen Zugriff auf den entsprechenden Datensatz ermöglicht. Wird der Index groß, so kann man den Index nicht mehr sequentiell durchsuchen. Auch die effizienten Datenstrukturen zur sortierten Organisation im Hauptspeicher helfen hier nicht mehr weiter. B-Bäume sind hier ein gute Lösung, wenn keinen näheren Eigenschaften über die Nutzungscharakteristik vorliegen.

Bemerkung 4.27 (Sequentieller und wahlfreier Zugriff)
Es ist für mich erstaunlich, wie oft in der Praxis sequentiell gesucht wird. Hier einige Beispiele aus alter und neuerer Zeit:

CICS Eine Reihe von Kontrollblöcken in CICS wurden zumindest bis in die 1990er Jahre hinein sequentiell durchsucht. Das war bei Tabellen wie der PCT (Program Control Table) oder der PPT (Program Processing Table) weitgehend unkritisch, da diese nur beim Start einer Transaktion und eines Programms benutzt wurden. Aber auch die sogenannte Filetable für DATACOM/DB, auf die bei jedem Zugriff auf eine Datenbanktabelle zugegriffen werden musste, wurde von der DATACOM CICS Service Facility sequentiell durchsucht. Standen einige sehr oft benutzte Tabellen weit hinten in der Filetable, so konnte man dramatische Performanceverbesserungen durch einfaches Umsortieren erreichen.

ACL Access Control List in CISCO Routern wie dem Catalyst 7500 werden ACLs (Access Control List) sequentiell durchsucht.

RAS In der default-Konfiguration wurde in dem RAS (Remote Access Server) der HAW beim Einloggen eines Benutzers sequentiell nach der Userid gesucht, um diese zu verifizieren. Bei einigen tausend Benutzern ist das natürlich nicht tragbar, so dass eine Datenbank verwendet werden muss.

Mit den heute zur Verfügung stehenden Techniken und Entwicklungsumgebungen sollten hier noch viele Verbesserungen möglich sein. ◀

In der Praxis bestehen Indizes oft aus zwei Teilen:

- Dem B-Baum, wie er in diesem Kapitel beschrieben wurde. Diese liefern einen schnellen Zugriff auf den Teil des Index, über den dann die Daten erreicht werden können.

- Einer untersten Schicht, in der dann die Verweise auf die Daten stehen.

In der Darstellung von [Knu73b] heißt der erste Teil „index set“ und der untere „sequence set.“
Siehe hierzu auch [Dat98]
Für die genaue Gestaltung gibt es verschiedene weitere Varianten:

Präfix-B-Baum Auf den oberen Ebenen wird nicht der ganze Schlüssel gespeichert, sondern nur der Teil, der benötigt wird um die Teilbäume zu zerlegen.

Komprimierter Präfix-B-Baum Zusätzlich zu der Verwendung von Präfixen werden diese komprimiert: Ein Prefix einer höheren Ebene, der auf der unteren unverändert auftritt, wird nicht dort nicht wiederholt.

Schlüsselstruktur

1. Man kann pro Schlüssel eine Indexdatei anlegen. Dies hat den Nachteil, dass viele Indexdateien entstehen können und die Leistungsfähigkeit von B-Bäumen nicht vollständig ausgeschöpft wird.
2. Man kann alle Schlüssel einer Datei in einer Indexdatei speichern. Zusätzlich wird vor den Wert noch eine Id für den Schlüssel gesetzt.
3. Man kann alle gleichartigen Schlüssel in einer Indexdatei speichern. Gleichartig heißt hier, z. B. Fremdschlüssel und Primärschlüssel. Dies mag Vorteile bei Joins bringen.
4. Man kann alle Schlüssel einer Datenbank (im Sinne etwa von DB2 oder DATACOM) in einer Indexdatei speichern. So kann die Datei zwar viele Schlüssel enthalten, die Leistungsfähigkeit von B-Bäumen wirkt einem Leistungsverlust aber wirkungsvoll entgegen. Bei geeigneter Wahl der Ids für die Schlüssel kann man ähnliche Wirkungen bei Joins erreichen wie bei der vorstehenden Möglichkeit.

4.8 Graphen

Einführung und Definition

Bäume sind spezielle Graphen. Bei einem Graphen sind beliebige Beziehungen (Relationen) der Knoten untereinander erlaubt, nicht nur von Vater zu Sohn. Beispiele können sein:

- Personen, die sich untereinander kennen
- Orte, die durch Wege miteinander verbunden sind
- Computer, die miteinander verbunden sind
- Stellungen in einem Spiel (z. B. Schach) die auseinander hervorgehen

Bei Graphen sind Richtungen in den Beziehungen zwischen den Knoten erlaubt. So geht zwar eine Schachstellung aus einer anderern hervor, aber da man die Bauern z. B. nicht zurück ziehen darf, kommt man nicht zur alten Stellung wieder zurück. Graphen, die Richtungen in den Beziehungen nutzen, nennt man **gerichtete Graphen**. Graphen, bei denen jede Beziehung bidirektional ist (also in beide Richtungen), nennt man **ungerichtete Graphen**. Einen gerichteten Graphen kann man wie folgt definieren:

Definition 4.28 (Graph)
Ein **gerichteter Graph** (engl. *digraph* für directed graph) $G = (V, E)$ besteht aus

- einer Menge V von Knoten (vertices) und
- einer Menge $E \subseteq V \times V$ (edges) von Kanten.

Ein Element $(v_i, v_j) \in E$ heißt Kante und v_i und v_j heißen adjazent. ◄

Weiterhin definiert man folgendes:

Definition 4.29
Sei $G = (V, E)$ ein Graph.

- Die Anzahl $n = |V|$ der Knoten heißt *Ordnung* des Graphen.
- Die Anzahl $e = |E|$ der Kanten heißt *Größe* des Graphen.
- Zwei Graphen $G = (V, E), G' = (V', E')$ heißen *isomorph*, wenn es eine *bijektive* Abbildung $\varphi : V \longrightarrow V'$ gibt und es gilt $(u, v) \in E \Longleftrightarrow (\varphi(u), \varphi(v)) \in E'$.
- Eine Folge von Kanten
$$(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \ldots, (v_{i_{n-1}}, v_{i_n})$$
heißt **Pfad** der Länge $n - 1$ von v_{i_1} nach v_{i_n}.
- Ein Graph heißt *stark zusammenhängend*, wenn je zwei Knoten durch mindestens einen Pfad miteinander verbunden sind. Ein Graph heißt *zusammenhängend*, wenn je zwei Kanten durch mindestens einen Pfad im unterliegenden ungerichteten Graphen verbunden sind.
- Ein *Zyklus* in einem Graphen ist ein Pfad
$$(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \ldots, (v_{i_{n-1}}, v_{i_n})$$
mit $v_{i_1} = v_{i_n}$, wobei alle anderen v_{i_j} paarweise verschieden sind.
- Ein *zyklenfreier Graph* ist ein Graph, der keinen Zyklus enthält.
- Der *Umfang* eines Graphen ist die Länge seines kürzesten Zyklus. Sie ist ∞, wenn der Graph zyklenfrei ist.
- Der *Abstand* (distance) d(u,v) zweier Knoten u und v ist die minimale Länge eine Pfades von u nach v, er ist ∞, wenn es keinen Pfad von u nach v gibt.
- Der *Durchmesser* (diameter) eines Graphen ist das Maximum von d(u,v) über alle Ecken u, v.
- Der Grad (degree) eines Knotens (vertex) v in einem ungerichteten Graph ist die Anzahl der Kanten (edges), die in v beginnen; Kanten, die in v enden und beginnen zählen dabei doppelt. Der out-degree (in-degree) eines Knotens in einem gerichteten Graphen ist die Anzahl Kanten, die diesen Knoten verlassen (die zu diesem Knoten führen). Der Grad eines Knotens in einem gerichteten Graph ist die out-degree + in-degree. Der Grad eines Graphen ist das Maximum der Grade seiner Knoten.

◄

Sei $V_n := \{1, 2, \ldots, n\}$. Dann sei

$$K_n := (V_n, E_n),\ E_n = \{(u, v) \in V_n^2 | 1 \leq u < v \leq n\}$$

der vollständige Graph der Ordnung n.

$$P_n := (V_n, E_n),\ E_n = \{(v, v + 1) \in V_n^2 | 1 \leq v < n\}$$

ist dann der Pfad der Ordnung n und hat Größe $n - 1$.

$$C_n := (V_n, E_n),\ E_n = \{(v, (vmodn) + 1) \in V_n^2 | 1 \leq v < n\}, n \geq 3$$

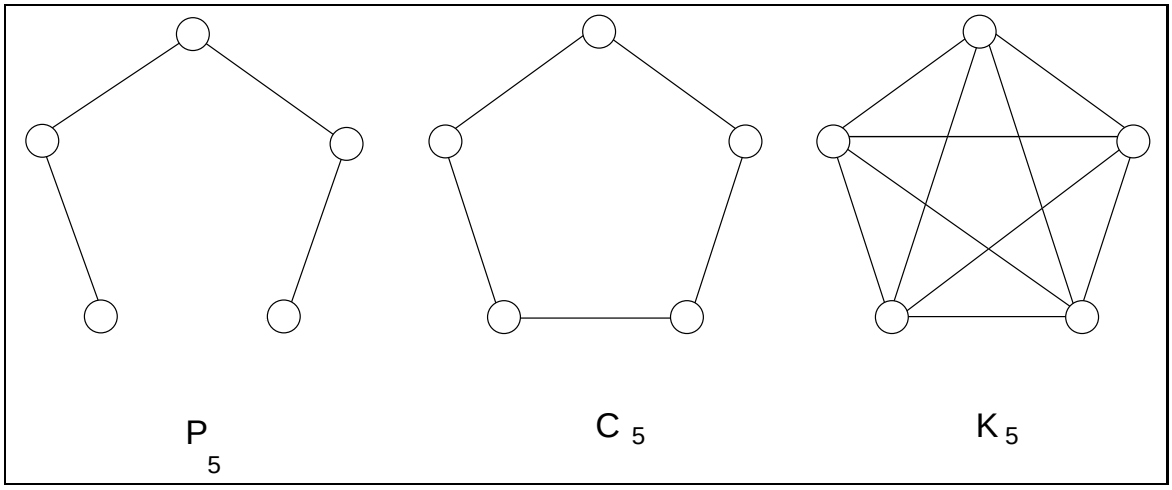


Abbildung 4.22: Einige elementare Graphen

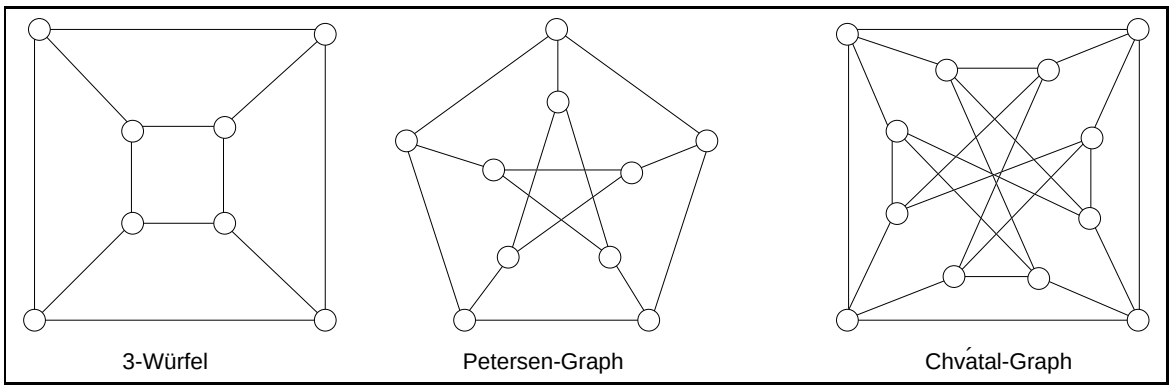


Abbildung 4.23: 3-Würfel, Petersen- und Chvátal-Graph

ist der Zyklus (cycle) der Länge n. Hier ist $n = \text{Ordnung}(C_n) = \text{Größe}(C_n)$. [Knu08a] Abbildung 4.22 zeigt einige Beispiele für verschiedene n. Eine Reihe von Graphen wurden entworfen um Vermutungen zu widerlegen und Beispiele für die grundlegenden Begriffe zu liefern. Interessant sind unter anderem der 3-Würfel, Petersen und der Chvátal Graph (Abb. 4.23).

Nun können wir auch wieder Bäume definieren, indem wir die Definition von Graphen verwenden:

Definition 4.30 (Baum)
Ein **Baum** ist ein zusammenhängender und zyklenfreier Graph. ◀

Typische Graph-Algorithmen

Wir wollen keine Graph-Algorithmen im Detail besprechen, sondern nur aufzeigen, welche Probleme es gibt und welche Aufgaben man mit Graphen lösen kann.

1. Eine Aufgabe besteht darin, einen Algorithmus zu finden der in einem gegebenen Graphen alle Knoten durchläuft. Dazu gibt es zwei Möglichkeiten: den *Tiefendurchlauf* und den *Breitendurchlauf*. Nimmt man an, dass man eine Wurzel r hat, von der aus man jeden Knoten erreichen kann, so kann man einen Baum erzeugen, der alle Pfade enthält, die in dieser Wurzel starten. Diesen Baum kann man nun nach *Preorder*-Reihenfolge durchlaufen, oder ebenenweise.

2. Bestimmung kürzester Wege: die Kanten des Graphen seien mit positiven reellen Zahlen (Kosten) versehen. Von einem gegebenen Knoten aus sollen nun die Wege zu allen Knoten bestimmt werden, so dass die Summe der Kosten der durchlaufenen Kanten minimal ist. Man spricht vom *single source shortest path*-Problem. Eine Lösung hierfür ist der Algorithmus von *Dijkstra*.

3. Man kann fragen, ob ein Graph Zyklen enthält. Dies ist bei großen Graphen nicht immer offensichtlich.

4. Man sucht einen Subgraphen, der nur aus einer Teilmenge des gegebenen Graphen besteht, und alle Knoten miteinander verbindet. Der Subgraph soll möglichst wenig Kanten enthalten. Man spricht von einem *minimalen Spannbaum*.

Implementierungen

Für die Implementierung von Graphen möchte ich zwei Möglichkeiten angeben.
Adjazenzmatrix Sei $G = (V, E)$ ein Graph mit n Knoten, $V = \{1, 2, \ldots, n\}$. Die *Adjazenzmatrix* von G ist die boolesche $n \times n$-Matrix A

$$A_{ij} = \begin{cases} \mathit{true} & \text{falls}(i,j) \in E \\ \mathit{false} & \text{sonst} \end{cases}$$

Bei Graphen, deren Kanten einen Wert haben (Kosten) kann direkt der Wert in die Adjazenzmatrix eingetragen werden. Die Adjazenzmatrix hat den Vorteil, dass man mit $\mathcal{O}(1)$ Aufwand feststellen kann, ob eine Kante von einem gegebenen Knoten zu einem anderen gegebenen Knoten führt.

Ein Nachteil ist der hohe Speicherplatzbedarf von $\mathcal{O}(n^2)$. Dies wiegt umso schwerer, je weniger Kanten es im Vergleich zu Knoten gibt.

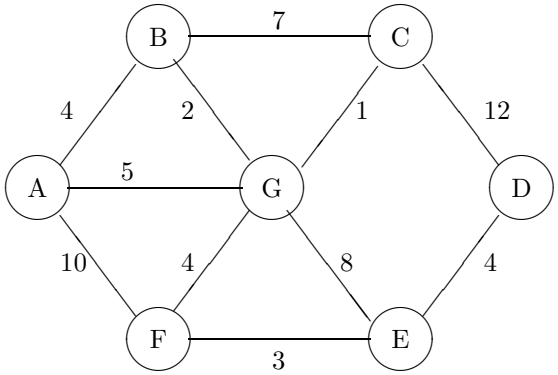
Adjazenzlisten: Bei dieser Implementatierung verwaltet man eine Liste von Nachbarn eines jeden Knotens. Man hat einen Array von Knoten, die als Einstiegspunkte in jede dieser Listen dient. Hier ist der Platzbedarf besonders gering, nämlich $\mathcal{O}(|V| + |E|)$.

Ein Test, ob zwei Knoten ν und κ benachbart sind, ist hier schwieriger, denn man muss im schlimmsten Fall beide Listen, die von ν und die von κ durchlaufen.

□

Beide Implementationen verwendet man für Graphen die statisch sind, also nach ihrer Erzeugung nicht mehr verändert werden müssen.

Aufgabe 4.31 (Adjazenzmatrix- und liste)
Gegeben Sie die Adjazenzmatrix und die Ajdazenzliste für folgenden Graphen an.



◀

Kürzeste Wege in bewerteten Graphen

Zunächst definieren wir einen bewerteten Graphen.

Definition 4.32 (Bewerteter Graph)
Ein bewerteter Graph ist ein Graph $G = (V, E)$ zusammen mit einer Abbildung

$$c : E \longrightarrow \mathbb{R}_0^+.$$

Diese Abbildung nennt sich Kostenfunktion (engl. cost). Sie beschreibt die Kosten, die anfallen, wenn die Kante (engl. edge) durchlaufen wird. ◀

Wie schon angesprochen können Kosten in der Adjazenzmatrix eingetragen werden. So hat man eine kompakte Darstellung eines bewerteten Graphen. Wir wollen uns nun um folgende Fragestellung kümmern:

Für einen Startknoten $\sigma \in V$ sollen die kürzesten Wege zu allen anderen Knoten bestimmt werden. Ein kürzester Weg von σ zu einem Knoten κ ist ein Pfad von σ nach κ, so dass die Summe der Kosten aller durchlaufenen Kanten kleiner ist als bei jedem anderen Pfad von σ nach κ.

Man könnte auch fragen, welches der kürzeste Weg zwischen zwei Knoten σ und κ ist, anstatt die Antwort gleich für alle Knoten wissen zu wollen. Interessanterweise ist der Aufwand aber nicht größer, wenn man gleich die kürzesten Wege von σ zu jedem anderen Knoten bestimmt. Der Algorithmus, der dies leistet, nennt sich

Algorithmus von Dijkstra Die Idee des Algorithmus von Dijkstra besteht darin, wellenförmig vom Startknoten σ aus alle anderen Knoten zu erforschen (Breitendurchlauf). Dabei werden die Kosten zu den Nachbarknoten registriert. Die Nachbarknoten werden dann bewertet mit den minimalen Kosten die anfallen um σ zu erreichen. Außerdem merken sich die Knoten noch, auf zu welchem Nachbarknoten sie gehen müssen um den preiswertesten Weg zu σ zu finden.

Jeder Knoten hat drei (zusätzliche) Attribute:

- *pred* für den Vorgänger Knoten zu dem man gehen muss um am preiswertesten nach σ zu kommen,
- *cost* hält die minimalen Kosten auf dem Weg zu σ

- *marked* merkt sich, ob der Knoten schon abschließend behandelt wurde.

```
class Knoten<T>
{
     T daten;
     Knoten<T> pred;
     float cost;
     bool marked;
     ...
}
```

Der Algorithmus hat nun folgenden Ablauf:

1. **Initialisierung:** alle Knoten κ außer dem Startknoten σ bekommen folgende Initialisierung:

   - κ.pred = undefiniert;
   - κ.cost = ∞;
   - κ.marked = false;

   Der Startknoten σ wird wie folgt initialisiert

   - σ.pred = σ;
   - σ.cost = 0;
   - σ.marked = true;

2. Bestimme den Rand R, der aus adjazenten Knoten zu σ besteht.

3. **while** $R \neq \emptyset$ **do**

   - wähle $\nu \in R$ so dass ν.cost minimal, und entferne ν aus R
   - ν.marked = true
   - ergänze Rand R bei ν

Beim Ergänzen des Randes R in einem Knoten ν muss folgendes gemacht werden:

1. wähle adjazente Knoten κ von ν die nicht markiert sind.

2. für jeden dieser Knoten κ finde heraus, ob

   $$\kappa.cost > \nu.cost + c(\nu, \kappa).$$

   Wenn dem so ist, dann setze

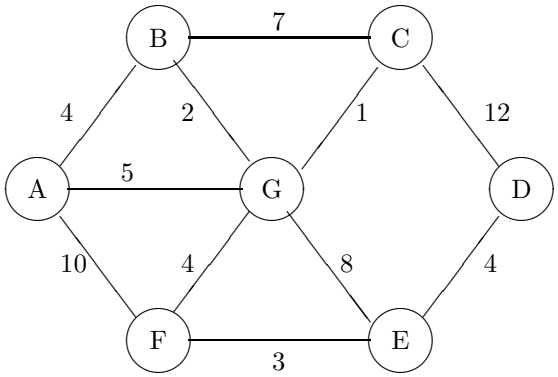
   $$\kappa.cost = \nu.cost + c(\nu, \kappa)$$

   und

   $$\kappa.pred = \nu.$$

3. nimm κ in R mit auf.

Beispiel 4.33
Wir betrachten folgenden (ungerichteten) bewerteten Graphen:



Für diesen Graphen erzeugt der Algorithmus von Dijkstra für den Startpunkt A folgende Randmengen R und markierte Knoten. Die Tripel bedeuten folgendes:

$$Knoten.(pred, cost, marked)$$

.

markiert	Randmenge
A.(A,0,1)	B.(A,4,0) G.(A,5,0) F.(A,10,0)
B.(A,4,1)	G.(A,5,0) C.(B,11,0) F.(A,10,0)
G.(A,5,1)	C.(G,6,0) F.(G,9,0) E.(G,13,0)
C.(G,6,1)	D.(C,18,0) F.(G,9,0) E.(G,13,0)
F.(G,9,1)	E.(F,12,0) D.(C,18,0)
E.(F,12,1)	D.(E,16,0)
D.(E,16,1)	

◄

Aufgabe 4.34
Bestimmen Sie die Ablauffolge (Randmenge, markierte Knoten) für den Algorithmus von Dijkstra für den Startpunkt B! ◄

4.9 Algorithmus von Kruskal

In der Mathematik wie auch in der Informatik kann durch „Verdichtung“ der Informationen die Bearbeitung dieser Informationen erleichtert werden. Der/die LeserIn mögen z. B. an Flächen denken, die durch drei Punkte beschrieben werden können. Bei einer Verschiebung einer Fläche müssen nun nicht alle ihre Punkte einzeln verschoben werden, sondern nur die drei sie beschreibenden Punkte.

In ähnlicher Weise kann der Nutzen eines Gerüstes angesehen werden.

Definition 4.35
Sei $G(V, E)$ ein ungerichteter Graph. Ein Baum $H(V', E')$ heißt ein *Gerüst* (spanning subgraph) von G, wenn H ein Teilgraph von G ist und alle Knoten von G enthält ,d. h. wenn gilt $E' \subseteq E$ und $V' = V$. Wenn H ein Gerüst von G ist, sagt man auch: „G wird von H aufgespannt“. ◄

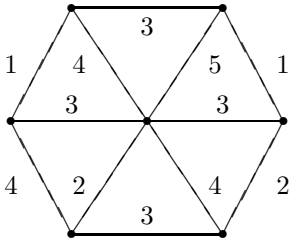
Es gibt einen Algorithmus, der ein bzgl. irgendwelchen Kosten minimales Gerüst findet. Dieser Algorithmus, nachfolgend beschrieben, gehört zur Klasse der Greedy[4]-Algorithmen.

Definition 4.36 (Algorithmus von Kruskal)
Eingabe: Eine Menge E der Kanten mit ihren Längen (Kosten). Ausgabe: Minimales Gerüst von $G(V, E)$, d.h. eine dafür geeignete Teilmenge E' der Kantenmenge.

- Nummeriere die Kanten $e_1, \ldots, e_{|E|}$ nach aufsteigender Länge. Setze $F := \emptyset$.
- Für $i := 1, \ldots, |E|$:
  - Falls $F \cup \{e_i\}$ nicht die Kantenmenge eines Kreises in G enthält, setze $F := F \cup \{e_i\}$.

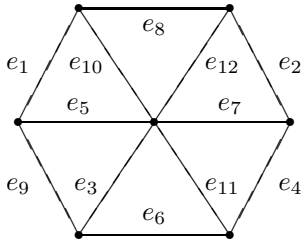
◀

Beispiel 4.37
Wir bestimmen ein Minimalgerüst des folgenden Graphen:



Hier der Ablauf:

Initialisierung F := {}. Die Nummerierung ist wie folgt:



$i = 1$ F := $\{e_1\}$

$i = 2$ F := $\{e_1, e_2\}$

$i = 3$ F := $\{e_1, e_2, e_3\}$

$i = 4$ F := $\{e_1, e_2, e_3, e_4\}$

$i = 5$ F := $\{e_1, e_2, e_3, e_4, e_5\}$

$i = 6$ F := $\{e_1, e_2, e_3, e_4, e_5, e_6\}$

$i = 7$ Kreis: e_7, e_4, e_6, e_3

$i = 8$ Kreis: $e_8, e_1, e_5, e_3, e_6, e_4, e_2$

$i = 9$ Kreis: e_9, e_3, e_5

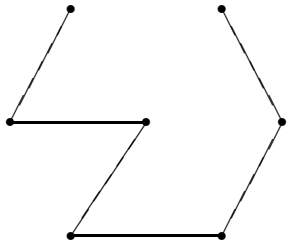
$i = 10$ Kreis: e_{10}, e_1, e_5

[4] greedy: gierig

$i = 11$ Kreis: e_{11}, e_3, e_6

$i = 12$ Kreis: $e_{12}, e_3, e_6, e_4, e_2$

Ende Das minimale Gerüst mit Gesamtlänge ist F := $\{e_1, e_2, e_3, e_4, e_5, e_6\}$:



◀

4.10 Fazit aus Sicht der Konstruktionslehre

Die binären Suchbäume wurden verwendet, um Daten, nach denen oft gesucht werden soll, komprimiert darzustellen. Der Trick besteht in der Sortierung der Nachfolger eines Knotens bzgl. dem Knoteninhalt. Die Beobachtung zeigt, dass die Suchbäume jedoch entarten können, d.h. die eigentliche Baumstruktur kann verloren gehen und damit der Laufzeitvorteil.

Daher wurden hier zwei Verfahren vorgestellt, die nach dem Löschen bzw. Einfügen eines Elementes den evtl. entarteten Baum „reparieren“ bzw. erneut balancieren.

Die AVL-Bäume verwenden als Indiz für eine notwendige Reparatur explizit die Höhe des Baumes als eine Art Verwaltung. Der balance-Faktor stellt schon eine implizite „Markierung“ der Knoten dar, wobei noch deutlich wird, in welchem Teilbaum eine linksseitige oder rechtsseitige (Über-)Last besteht. Die Rot-Schwarz-Bäume arbeiten nur noch mit einer Markierung, also einer kodierten, impliziten Verwaltung, die in dem Sinne eine eigene, implizite Höhe - die „Schwarztiefe“ verwenden. Dieses Konstruktionsprinzip verwenden wir auch an anderen Stellen: explizite Verwaltung mit expliziten Daten (meist gut zu verstehen, und meist viel Speicher und ggf. spürbare Laufzeit) oder impliziten Verwaltung, d.h. einer Kodierung der Daten in Markierungen, z.B. Farben. Diese benötigen gegenüber der expliziten Verwaltung meist wenig Speicher, sind jedoch nicht immer leicht zu verstehen. Die Trennung ist jedoch nicht immer so scharf! Wichtig ist dann, wie effizient der ALgorithmus (also der Anwender dieser Repräsentation) mit der jeweiligen Art arbeiten kann!

Beide Verfahren konzentrieren sich bei der Reparatur auf die „Verursachung“, d.h. sie arbeiten lokal beginnend bei der Stelle, an der ein Element gelöscht/eingefügt wurde. Im schlimmsten Fall arbeiten sie sich bis zur Wurzel, jedoch nur lokal den jeweiligen Knoten und seine (direkten) Nachfolger/Vorgänger betrachtend. Der komplette Baum wird niemals betrachtet. Dadurch fallen diese Reparaturalgorithmen in die $O(nlogn)$-Klasse.

Die permanente Reparatur, d.h. das Aufrechterhalten der Invarianten der AVL-/Schwarz-Rot-Baumeigenschaft, erweist sich als sehr effizient bzgl. der Reparaturdauer und bzgl. der Verfügbarkeit der Daten bzw. des Suchdienstes.

Bei den Graphen haben wir das Konstruktionsprinzip der Greedy-Algorithmen kennen gelernt. Zunächst betrachten wir die Arbeitsweise eines Greedy-Algorithmus abstrakt. Normalerweise haben wir folgendes zur Verfügung:

1. Eine Menge von Kandidaten c, aus denen wir die Lösung konstruieren wollen.

2. Eine Teilmenge $s \subseteq c$, die wir bereits ausgewählt haben.

3. Eine boolesche Funktion *solution*, die uns sagt, ob eine Menge von Kandidaten eine legale Lösung des Problems darstellt, unabhängig davon, ob diese Lösung optimal ist.

4. Eine Testfunktion *feasible*, die uns sagt, ob eine gewisse Teillösung u.U. zu einer kompletten legalen Lösung erweitert werden kann.

5. Eine Auswahlfunktion *select*, die uns denjenigen noch unbenutzten Kandidaten liefert, der im Sinne der Greedy-Strategie der erfolgversprechendste ist.

6. Eine Zielfunktion *val*, die uns den Wert einer gewissen Lösung angibt.

Die Greedy-Strategie startet mit einer leeren Lösungsmenge und erweitert sie schrittweise um das höchstwertige passende Element der Kandidatenmenge:

Definition 4.38
Greedy-Algorithmen
◀

```
function greedy(c) : set
 {c ist die gesamte Menge der Kandidaten}
 {Gesucht ist eine "optimale" Teilmenge von c}
 {Die wird in die Loesungsmenge s hineinkonstruiert}

 s := emptyset {Loesungsmenge ist am Anfang leer}

 while not solution(s) and c <> emptyset do
   x := {dasjenige Element von c, das select maximiert}
   c := c - {x}  {x wird nur einmal angefasst und}
                 {aus c unwiderruflich entfernt}
   if feasible(s + {x}) then    {Geht es?}
     s := s + {x}    {Geht! Es bleibt auf ewig in s drin!}

 if solution(s) then
   return s
 else
   {Es gibt keine Loesung!}
```

Am Ende liefert $val(s)$ den Wert der gefundenen Lösung. Dass diese Lösung den Wert von *val* tatsächlich optimiert, ist die wesentliche Eigenschaft *greedy-lösbarer* Probleme. Einige Optimierungsprobleme auf Graphen lassen sich *greedy-lösen*: die bekanntesten sind die Algorithmen von Kruskal und Dijkstra, die wir hier kennen gelernt haben.

4.11 Historische Anmerkungen

B-Bäume wurden 1969 von Rudolf Bayer und E. M. McCreight erfunden. Die bei der damals renommiertesten Informatik-Zeitschrift, den Communications of the ACM (CACM) eingereichte Arbeit wurde von den Gutachtern nicht akzeptiert. Don Knuth schrieb damals an der ersten Auflage des dritten Bandes von TAOCP [Knu73b] und kannte die Arbeiten von Bayer und McCreight. Er sorgte dafür, dass die Arbeit in Acta Informatica erscheinen konnte [BM72]. Wieder abgedruckt wurde der Artikel [BM72] in [BD02].

4.12 Aufgaben

1. Welches ist die Mindestanzahl von Knoten, die ein AVL-Baum der Höhe h hat?

2. Bestimmen Sie die Ablauffolge (Randmenge, markierte Knoten) für den Algorithmus von Dijkstra für den Startpunkt B!

3. Warum ist in der Definition des B-Baums $t > 1$ gefordert?

4. Schreiben Sie den Code für die nicht-rekursive Implementierung der drei beschriebenen Reihenfolge-Durchläufe bei Binärbäumen!

5. Schreiben Sie den Code für die Löschfunktion in einem binären Suchbaum!

6. Wie ändern sich die `exists` und `insert` Operationen für einen binären Suchbaum, wenn Schlüsselwerte mehrfach vorkommen können?

7. Wieviele Knoten N hat ein vollbesetzter Baum vom Grad d und Höhe h? Welche minimale Höhe h hat ein Baum vom Grad d mit N Knoten?

8. (M21) Ein 0-2-Baum sei definiert als ein Baum, in dem jeder Knoten genau 0 oder 2 Söhne hat. Beweisen Sie bitte:

   8.1. Jeder 0-2-Baum hat eine ungerade Anzahl Knoten!

   8.2. Zu jedem Binärbaum mit n Knoten gibt es genau einen (geordneten) 0-2-Baum mit $2n + 1$ Knoten![Knu97a]

9. Arbeiten Sie die Lösch-Operation (vgl. S. 91) in einem binären Suchbaum so aus, dass Sie leicht und sicher implementieren können!

10. Wenn in einem AVL-Baum beim Löschen oder Einfügen an einer Position entdeckt wird, dass die Balance nicht mehr stimmt, so wird um diese Position gemäß den Regeln rotiert. Zeigen Sie: die Höhe dieser Position ist nach der Rotation gleich der Höhe vor dem Einfügen oder Löschen. Es genügt dies für das Einfügen sowie die Rechts- bzw. Linksrotation und die doppelte Rechtsrotation bzw. doppelte Linksrotation zu zeigen.

11. Nachfolgende Implementierung für den AVL-Baum muß fertig gestellt werden. Folgender, für die Implementierung benötigter Code liegt dazu vor:

```
public class AVLnode {
  int key = -1, hoehe = -1, counter = 0;
  AVLnode links = null, rechts = null;

  public AVLnode (int wert){
     key = wert; hoehe = 0; counter = 1;
     links = rechts = null; } }


public class AVLtree {
   private static AVLnode wurzel = null;
   ...
   public static void insert(int c) {
      wurzel = insert(wurzel, c); }
   public static AVLnode insert(AVLnode n, int insert) {
      /* TODO */ }

   public static AVLnode rotiererechts(AVLnode n) {
      /* TODO: Rechtsrotation UND Hoehen neu bestimmen!  */ }

   public static AVLnode rotierelinks(AVLnode n) {
      /* TODO: Linksrotation UND Hoehen neu bestimmen!  */ }

   public static int hoehe(AVLnode n) {
      if (n == null) return -1;
```

```
    else return n.hoehe; }
 public static AVLnode pruefen(AVLnode n) {
    /* FERTIG: prueft Balance und rotiert ggf. für Knoten n */
    ... } }
```

Implementieren Sie rekursiv den fehlenden Code für die Funktion `insert`, die einen Wert in den AVL-Baum einfügt und auf eine Rebalancierung prüft (linker Teilbaum ist für kleinere Werte zuständig). Bei Mehrfachvorkommen eines Wertes ist ein Zähler in dem zugehörigen Knoten zu inkrementieren. Implementieren Sie zudem die beiden Rotationen `rotierelinks` für die Links- bzw. `rotiererechts` für die Rechtsrotation (Siehe Abbildung 4.10, Seite 94).

12. **AVL-Baum** Fügen Sie mit dem in der vorangegangenen Aufgabe von Ihnen implementierten Algorithmus folgende Zahlen **in der vorgegebenen Reihenfolge** in einen leeren AVL-Baum ein: $(4, 5, 8, 6, 9, 7, 1, 0, 2, 3)$. Geben Sie die Anzahl der durchgeführten Rechts- und Linksrotationen an. Geben Sie am Ende den Baum in Inorder als eine Zeile aus.

    Sofern Sie die vorangegangene Aufgabe (noch) nicht gelöst haben, geben Sie bitte in Pseudo-Code an, nach welcher Methode Sie in den AVL-Baum Elemente einfügen.

13. Gegeben sei folgender allgemeiner Greedy-Algorithmus:

```
function greedy(c) : set
 R := emptyset
 while not solution(R) and C <> emptyset do
 { x := select(C,R)
   C := C - {x}
   if feasible(R + {x}) then
      R := R + {x}
  } // end-while
 if solution(R) then
    return R
   else
    {Es gibt keine Loesung!}
```

    Bestimmen Sie für nachfolgendes Problem folgende Elemente des allgemeinen Greedy-Algorithmus:

    - Eine Menge von Kandidaten **C**, aus der die Lösung konstruiert wird;
    - Die Lösungsmenge **S**$\subseteq C$;
    - Eine boolesche Funktion **solution**, die angibt, ob eine Menge von Kandidaten eine legale Lösung des Problems darstellt;
    - Eine Testfunktion **feasible**, die angibt, ob eine Teillösung noch zu einer kompletten legalen Lösung erweitert werden kann;
    - Eine Auswahlfunktion **select**, die einen noch unbenutzten Kandidaten liefert, der im Sinne der Greedy-Strategie der erfolgversprechendste ist;
    - Eine Zielfunktion **val**, die den Wert einer gewissen Lösung angibt.

    Wenden Sie den daraus resultierenden Algorithmus auf folgendes „Bepackungsproblem“ an: Gegeben sei ein leerer Rucksack mit maximalem Fassungsvermögen von 40kg. Versuchen Sie gemäß Ihrem Algorithmus den Rucksack mit folgenden Teilen zu bepacken: 4kg,9kg,11kg,14kg,22kg,26kg.

    Welche Teile würden Sie „von Hand“ (also ohne erkennbares Prinzip) auswählen ?

14. Führen Sie den Algorithmus von Dijkstra mit dem Graphen in Abbildung 4.24 (Seite 121) durch (bis zum Abbruch!). Wählen Sie als Ausgangspunkt den Knoten x_1. Fertigen Sie, ähnlich wie in der Vorlesung, eine Dokumentation an, aus der der Ablauf deutlich wird. Bei Wahlmöglichkeiten ist immer die lexikographisch niedrigste Ecke zu wählen.

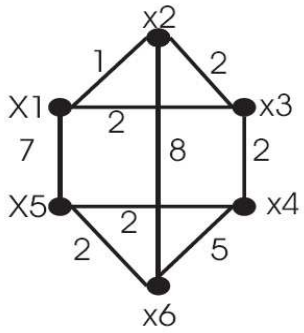


Abbildung 4.24: Optimaler Weg gesucht!

Kapitel 5

Hash-Verfahren

Im vorherigen Kapitel haben wir gelernt, dass Bäume — insbesondere Suchbäume — gut dazu geeignet sind, Mengen von Objekten zu verwalten, in denen man bestimmte Stellen schnell finden muss. Diese Verfahren basieren im Wesentlichen auf einer abstrakten Vergleichsoperation und benötigen einen Aufwand von $\mathcal{O}(\log n)$. Unter Umständen geht es aber noch schneller bestimmte Einträge zu finden, nämlich mit dem Aufwand $\mathcal{O}(1)$. Dies ist der Fall, wenn man aus dem Schlüssel des Objekts direkt die Speicherstelle — meistens den Index in einem Array — berechnen kann. Sind die Schlüssel ganzzahlige Werte wäre eine direkte schlüsselindizierte Suche möglich, d. h. der Schlüssel wird als Adresse (in einem Array) verwendet[1]. Wenn die Schlüssel dazu nicht direkt verwendbar sind, wird ein Berechnungsverfahren eingesetzt: eine Hashfunktion. Das Array, in dem die Objekte an der durch die Hashfunktion berechneten Adresse gespeichert werden, nennt man Hashtabelle. Hier schauen wir uns also ein ganz anderes Verfahren an: anstatt durch Wörterbuchdatenstrukturen zu navigieren (mittels vergleichsbasierten Verfahren), versuchen wir hier, die Elemente in einer Tabelle direkt zu referenzieren, indem Schlüssel durch arithmetische Operationen in Tabellenadressen transformiert werden.

5.1 Einführung

Eine Hashfunktion sollte die Eigenschaft haben, dass die Schlüssel der zu indizierenden Objekte möglichst gleichmäßig auf das Array verteilt wird. Werden zwei Objekte auf den gleichen Arrayindex abgebildet, dann kommt es zu einer Kollision. Später werden wir Strategien besprechen, um mit Kollisionen umzugehen. Zunächst wollen wir aber Hashfunktionen finden, so dass möglichst wenig Kollisionen auftreten.

Beispiel mit Matrikelnummern

Wir wollen den Unterschied zwischen einer guten und einer schlechten Hashfunktion kennen lernen. Dazu bilde ich ein Array, in das ich Studenten einordne. Dazu lasse ich mir die ersten drei Ziffern der Matrikelnummer nennen und trage sie in eine Tabelle (0-9) ein, entsprechend der ersten der genannten drei Ziffern. Möglicherweise erhalten wir dabei viele Kollisionen und eine sehr ungleichmäßige Verteilung.

Dann lasse ich mir die letzten drei Ziffern der Matrikelnummern der Studenten nennen und trage sie wieder in eine Tabelle (0-9) ein. Dies sollte eine wesentlich bessere Verteilung geben. Die Tabelle wird fast gleichmäßig aufgefüllt. Dies ist die gute Hashfunktion: Es treten zwar immer noch Kollisionen auf, aber diese werden minimiert.

Die minimale Anzahl an Kollisionen in diesem Beispiel ist *Anzahl Studierende - 10*, z. B. bei 30 Studierenden wären 20 Kollisionen (3 Einträge pro Arrayindex) minimal. Die maximale Anzahl ist

[1] Bei Assoziationsmatrizen greift man direkt mittels Inhalt auf die gewünschten(assoziierten) Daten zu; diese finden z.B. bei neuronalen Netzen Verwendung.

Anzahl Studierende, z.B. bei 30 Studierenden 30 Kollisionen (30 Einträge in einem Arrayindex). Die beste Hashfunktion erreicht also die minimale Anzahl an Kollisionen.

Eine gute Hashfunktion sollte also folgende Eigenschaften haben:

- möglichst surjektiv sein (also das ganze Array abdecken)
- möglichst injektiv sein (also keine Kollisionen)
- die Berechnung muss einfach sein.

Natürlich kann man mit den Begriffen *möglichst surjektiv* und *möglichst injektiv* nicht viel anfangen, aber wir bekommen schon mal ein Gefühl dafür, was gebraucht wird.

Geburtstagsparadoxon

Wir wollen die Wahrscheinlichkeit ausrechnen, mit der eine Kollision auftritt. Dabei nehmen wir an, dass es m Behälter gibt auf die die Objekte verteilt werden sollen (das Array hat also Größe m). Es sollen n Objekte verteilt werden. Wir nehmen an, dass die Hashfunktion in dem Sinne *ideal* ist, dass sie die n Schlüsselwerte gleichmäßig auf die m Behälter verteilt. Die Wahrscheinlichkeit, dass eine Kollision eintritt, bezeichnen wir mit

$$P_{Kollision}.$$

Offenbar gilt

$$P_{Kollision} = 1 - P_{keineKollision}.$$

Wir bezeichnen mit $P(i)$ die Wahrscheinlichkeit, mit der das i-te Objekt auf einen freien Behälter abgebildet wird, wenn alle vorherigen Schlüssel ebenfalls auf freie Behälter abgebildet wurden. Für das erste Objekt gilt, dass keine Kollision auftreten kann, denn es sind ja noch keine Objekte in den Behältern. Also gilt

$$P(1) = 1.$$

Für das zweite Objekt gilt, dass es einen von $m-1$ Behältern treffen muss, damit keine Kollision stattfindet. Also

$$P(2) = \frac{m-1}{m}.$$

Nun sind zwei Behälter belegt da wir annehmen, dass es zu keiner Kollision kommt. Dann gilt für das dritte Objekt

$$P(3) = \frac{m-2}{m}.$$

Allgemein kann man erkennen, dass gilt

$$P(i) = \frac{m-i+1}{m}.$$

Nun sollen alle diese Ereignisse eintreten. Die Wahrscheinlichkeit dafür ist

$$P_{keineKollision} = P(1) \cdot P(2) \cdot \dots \cdot P(n),$$

oder

$$P_{Kollision} = 1 - \frac{m(m-1)(m-2)\dots(m-n+1)}{m^n}.$$

Hiermit kann man die Wahrscheinlichkeit ausrechnen, mit der zwei Leute am gleichen Tag Geburtstag haben. Bei n anwesenden Personen erhält man:

$$P_{Kollision} = 1 - \frac{\prod_{i=0}^{n-1}(365-i)}{365^n}.$$

Bei zwei Personen also:

$$P_{Kollision} = 1 - \frac{\prod_{i=0}^{2-1}(365-i)}{365^2} = 1 - \frac{(365-0)(365-1)}{365^2} \approx 0,00274 = 0,274\%.$$

Als Ergebnis bekommt man, dass bereits bei 23 anwesenden Personen eine Wahrscheinlichkeit von über 50 Prozent besteht, dass zwei Personen am gleichen Tag Geburtstag haben. Bei 50 anwesenden Personen ist die Wahrscheinlichkeit schon 97 Prozent.

5.2 Hashfunktionen

Welche Hashfunktionen sind denn nun konkret möglich? Wir stellen mehrere Möglichkeiten vor. Der Bereich aus dem die Schlüsselwerte stammen nennen wir D. Wir bezeichnen die Hashfunktion mit

$$\begin{array}{rcl} h : D & \longrightarrow & \{0, \ldots, m-1\} \\ k & \mapsto & h(k) \end{array}$$

Warum wird hier nicht nur eine, die „beste“ Möglichkeit vorgestellt ? Wir haben hier das Dilema, dass eine auf alle möglichen Verteilungen der Schlüsselwerte sehr gut reagierende Hashfunktion in der Berechnung sehr aufwendig ist und damit den Zeitvorteil wieder aufbraucht. Daher wird die für die zu erwartende Verteilung der Schlüsselwerte schnellste Hashfunktion mit minimaler Anzahl an Kollisionen gesucht. Im folgenden Abschnitt betrachten wir z.B. ein Verfahren, dass hervorragend geeignet ist, wenn die Schlüsselwerte selbst gleichmäßig verteilt sind.

Divisions(-Rest)-Methode

Hier definieren wir die Hashfunktion als

$$h(k) = k \bmod m.$$

Die Qualität von dieser Hashfunktion hängt von m und der „Verteilung“ der Schlüsselwerte ab. Diese Funktion hat die Eigenschaft, dass sie aufeinanderfolgende Schlüssel auf aufeinanderfolgende Indizes abbildet. Das kann ungünstig sein, wenn die Schlüsselwerte in dem Sinne ungünstig verteilt sind. Den Grund werden wir erkennen, wenn wir die Kollisionsvermeidungsstrategien besprechen.

Sehr oft wird aus Effizienzgründen (schnellere Berechnung durch Bit-shift-Operationen) eine Darstellung der Schlüssel zu einer Basis 2^n gewählt. Wenn nun m einen 2^k-Teiler beinhaltet, werden nicht mehr alle Bits der Schlüssel berücksichtigt, was sich nachteilig auf die Hashfunktion auswirkt. Dies wird am besten durch die Wahl einer Primzahl für m vermieden.

Neben der Berechnung mittels *mod*-Funktion kann noch eine Transformation des Schlüssels in eine Zahl benötigt werden, sofern man nicht die binäre Darstellung direkt als solche auffasst.

Multiplikative Methode

Sollten die Schlüsselwerte ungünstig verteilt sein, benötigt man in der Hashfunktion einen „simulierten Zufall“ als Störungsfaktor, um diese ungünstige Verteilung in eine Gleichverteilung zu wandeln[2]. Bei diesem Verfahren wird dazu der Schlüssel k mit einer irrationalen Zahl Θ multipliziert:

$$k \cdot \Theta$$

Irrationale Zahlen haben den Vorteil, durch ihr Berechnungsverfahren beliebig viele Zahlen liefern zu können und wegen ihrer Veranlagung eine Art „Zufallsreihenfolge“ von Ziffernfolgen zu liefern.

[2] Ein echter Zufall wäre schlecht, da er ja für die Suche nachgestellt werden muss.

Als Wert der Hashfunktion nimmt man nun einige der Nachkommaziffern vom Ergebnis der Multiplikation. Besonders gleichmäßig werden die Schlüssel verteilt, wenn man für Θ den goldenen Schnitt

$$\Theta = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

wählt. Betrachten wir den gebrochenen Anteil von Θ, 2Θ, 3Θ ..., so sehen wir, dass die neu berechneten Werte stets in eines der größeren verbliebenen der nach goldenem Schnitt geteilten Intervalle fallen [Knu73b], S. 510. Benötigt man z.B. einen Hashwert zwischen 0 und 999, also einen dreistelligen Hashwert für jeden Schlüssel, so nimmt man die erste, zweite und dritte Nachkommastelle des Produkts $k \cdot \Theta$ oder berechnet $(f(k \cdot \Theta)) \bmod 1000$, wobei f die gewünschten Nachkommastellen extrahiert, d.h. man kombiniert die Methode mit der Division-Rest-Methode. Sofern die Multiplikation zu große Werte erzeugt, nutzt man folgende Beziehung aus:

$$(a \cdot b \cdot c) \bmod m = ((((a \bmod m) \cdot (b \bmod m)) \bmod m) \cdot (c \bmod m)) \bmod m$$

Mittel-Quadrat-Methode

In diesem Verfahren wird als „simulierten Zufall“ bzw. als Störungsfaktor die Zifferndarstellung der Schlüsselwerte verwendet sowie ein nur den Schlüsselwert verwendeten Berechnungsverfahren.

Sei k eine Ziffernfolge gegeben durch

$$k = k_r k_{r-1} \ldots k_1.$$

Es wird nun k^2 gebildet. Die Zifferndarstellung von k^2 sei

$$k^2 = s_{2r} s_{2r-1} \ldots s_1.$$

Von dieser Ziffernfolge nehme man nun einen mittleren Block von Ziffern, etwa

$$s_i s_{i-1} \ldots s_j$$

für

$$2r \geq i > j \geq 1.$$

Ein mittlerer Block von Ziffern hängt von *allen* Ziffern in k ab. Dadurch werden aufeinanderfolgende Werte von k besser gestreut.

Bemerkung 5.1 (Mittel-Quadrat-Methode)
Die Mittel-Quadrat-Methode wurde ursprünglich von John von Neumann zur Generierung von Zufallszahlen vorgeschlagen. Hat die Anfangszahl 10 Stellen, so wird sie quadriert, die mittleren 10 Stellen sind die erste Zufallszahl, die nächste erhält man dann wieder durch Quadrieren und nimmt wieder die mittleren 10 Stellen etc. Diese Methode hat sich aber nicht bewährt, siehe z. B. [Knu97b]. ◄

Beispiel 5.2
Wir betrachten die Divisionsmethode und die Mittel-Quadrat-Methode im Vergleich. Bei der Divisionsmethode wählen wir die Primzahl $m = 101$. Bei der Mittel-Quadrat-Methode wählen wir die zweite und dritte Ziffer von rechts des Quadrates von k. Bei der multiplikativen Methode ist $\Theta = 0.6180339887$ gewählt. Als Hashfunktion werden die ersten beiden Nachkommastellen des Produkts $k \cdot \Theta$ genommen.

k	$k \bmod m$	k^2	$h_{Mittel-Quadrat}(k)$	$k \cdot \Theta$	$h_{mult}(k)$
722	15	521284	28	446,2205398414	22
723	16	522729	72	446,8385738301	83
724	17	524176	17	447,4566078188	45

Vergleich von Hashfunktionen

◀

Letztlich lässt sich erkennen, dass mittels des Faktors Zufall versucht wird, eine gleichmässige Verteilung zu erreichen, damit die minimale Anzahl an Kollisionen erreicht wird. Wünschenswert wäre eine Hashfunktion, die unabhängig von der Charakteristik der Schlüssel, also z.B. der Verteilung der Schlüssel selbst[3], eine ausgeglichene Verteilung auf der Hashtabelle erzeugt (und in dem Sinne die Verteilung der Schlüssel selbst neutralisiert).

Minimale perfekte Hashfunktionen

Eine optimale Hashfunktion bildet eine Schlüsselmenge D auf die Indexmenge $\{0, \ldots, n\}$ ohne Kollisionen und mit minimalem Speicherbedarf ab. Dies kann funktionieren, wenn die Schlüsselmenge D im voraus bekannt ist[4].

Definition 5.3
Eine Hashfunktion

$$h : D \to \{0, \ldots, n\}$$

heißt minimal, wenn h surjektiv ist. Das Array hat also die Größe $|D|$ und es wird kein zusätzlicher Speicher verbraucht.

Eine Hashfunktion h heißt perfekt, wenn h injektiv ist. Es treten also keine Kollisionen auf. ◀

Beispiel 5.4
Sei $D := \{2, 5, 7\}$. Definiere

$$h : D \to \{0, \ldots, 2\}$$

$$h(k) = \left\lfloor \frac{5}{k} \right\rfloor \mod 3.$$

Dann ist

$$\begin{aligned} h(2) &= 2 \\ h(5) &= 1 \\ h(7) &= 0 \end{aligned}$$

Dies ist eine minimale, perfekte Hashfunktion für diese spezielle Menge D. ◀

Die Methode, die im vorangegangenen Beispiel angewendet wurde, nennt sich *reciprocal hashing*. Sie hat folgende allgemeine Form:

$$h(k) = \frac{A}{B \cdot k} + C.$$

Dabei sind A, B, C so zu bestimmen, dass h minimal und perfekt ist. Oft wird zunächst $B = 1$ und $C = 0$ gesetzt. Diese Hashverfahren werden oft eingesetzt um Schlüsselworte von Compilern zu hashen (siehe hierzu z. B. das GNU Projekt gperf).

Dieses Verfahren wurde 1981 von G. Jaeschke beschrieben [Jae81].

[3] Wenn z.B. die ASCII-Werte der Zeichen eines Schlüssels aufaddiert werden, könnte ein enges Intervall an Zahlen entstehen.

[4] Hier erkennt man wieder das Konstruktionsprinzip, bei Notwendigkeit eine sehr genaue Untersuchung des Problems vorzunehmen, d.h. „maßgeschneiderte“ Lösungen zu erstellen, und nicht die Lösung von der Stange zu nehmen.

5.3 Kollisionsvermeidungsstrategien

Kollisionen lassen sich im Allgemeinen nicht gänzlich vermeiden. Daher muss man sich damit beschäftigen, was im Falle einer Kollision passieren soll. Das neu einzutragende Element, dessen Platz im Array schon belegt ist, muss irgendwo anders untergebracht werden. Dabei soll man es später trotzdem leicht wiederfinden können, d.h. die Methode muss versuchen, den Suchaufwand möglichst gering zu halten. Grundsätzlich gibt es zwei Methoden.

1. Man organisiert die Hashtabelle so, dass die mehrfachen Einträge zu einem Index unter einer Adresse als verkettete Liste realisiert werden[5].

2. Man errechnet aus dem Original-Hashwert einen neuen Hashwert und versucht den Schlüssel dort einzutragen.

Sie werden oft als *offenes Hashing* und *Geschlossenes Hashing* bezeichnet. Diese Begriffe werden jedoch in der Literatur gegensätzlich verwendet. Daher verwende ich hier die Begriffe *Separate Chaining* und *offene Adressierung*. Diese Begriffe werden in der Literatur konsistent verwendet.

Separate Chaining

Wie der Begriff sagt, wird hier eine separate Kette, also eine zusätzliche verkettete Liste angelegt. Diese kommt zum Einsatz, wenn eine Kollision entsteht. Die Elemente werden also nicht in einem Array gespeichert, sondern in Buckets, welche Elemente von verketteten Listen sind. Der Wert der Hashfunktion, der Hashcode, ist ein Index auf einen Array, der Referenzen auf Anfangselemente von verketteten Listen hält. Ist dieses Anfangselement noch frei, dann wird ein neues Element dort einsortiert. Ist es schon vergeben, dann entsteht also eine Kollision. Das neue Element wird dann in der entsprechenden Liste hinten angehängt.

Beispiel 5.5 (Separate Chaining)
Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 95\}.$$

Als Hashfunktion $h()$ verwenden wir

$$h(k) = k \bmod m \quad \text{mit } m = 7.$$

Den Array mit den Referenzen auf die verketteten Listen nennen wir

$$a[i], \quad i = 0, \ldots, 6.$$

Das erste Element mit $k = 24$ wird einsortiert bei

$$a[h(24)] = a[3].$$

0	1	2	3	4	5	6
			24			

Das zweite Element mit $k = 51$ wird einsortiert bei

$$a[h(51)] = a[2].$$

0	1	2	3	4	5	6
		51	24			

[5] Hier könnten auch AVL-Bäume oder ähnliches eingesetzt werden. Jedoch ist ab einer gewissen Größe zu prüfen, ob man überhaupt noch mit einem Hashverfahren arbeiten will.

Das Element mit $k = 63$ wird bei

$$a[h(63)] = a[0]$$

einsortiert.

0	1	2	3	4	5	6
63		51	24			

Beim Element $k = 77$ ist der Hashcode $h(77) = 0$. Hier haben wir eine Kollision. Der Platz $a[0]$ ist schon belegt. Daher wird das Element $k = 77$ hinter das Element $k = 63$ in die Liste eingehängt. Die Liste, welche hinter $a[0]$ hängt, besteht also aus den Elementen $k = 63$ und $k = 77$.

0	1	2	3	4	5	6
63		51	24			
77						

Um nun z.B. das Element $k = 77$ wiederzufinden, wird zunächst in $a[h(77)] = a[0]$ nachgeschaut. Dort findet man zuerst das Element $k = 63$. Man muss sich dann an der verketteten Liste entlanghangeln und kommt dann auf das Element $k = 77$. ◄

In Abhängigkeit des Verhältnisses κ von *Anzahl Schlüssel* zu *Grösse der Hashtabelle* kann die Liste recht lang werden, so dass hier andere, effizientere Strukturen, wie etwa balancierte Suchbäume, zum Einsatz kommen können. Im Prinzip wird mittels Hashfunktion ein erster, konstanter, aber evtl. „grober“ Zugriff vorgenommen, d.h. der Suchraum für den Schlüssel wird in Abhängigkeit von κ eingeschränkt. Das in der Literatur hier oft verkette Listen aufgeführt werden, macht deutlich, dass man zunächst von wenig Kollisionen pro Arrayindex ausgeht, also von einem „kleinem“ κ.

Offene Adressierung

In diesem Fall besteht die Hashtabelle aus einem Array von Referenzen auf die Elemente. Keine zusätzlichen Listen werden verwendet. Die Kapazität der Hashtabelle ist also so groß, wie das Array selbst. Um Kollisionen aufzulösen, muss also im selben Array ein anderer freier Platz für das neue Element gefunden werden. Man muss eine neue Adresse finden. Diesen Vorgang nennt man *Offene Adressierung*. Offene Addressierung wird manchmal *Offenes Hashing* genannt. Man findet aber auch den Begriff *geschlossenes Hashing*, weil die Kapazität der Hashtabelle beschrängt ist.

Im Prinzip wird im Falle einer Kollision ein neuer Hashcode mit einer weiteren Hashfunktion bestimmt. Entsteht dann wieder eine Kollision, so benötigen wir eine weitere Hashfunktion. Man benötigt also eine Folge von Hashfunktionen, wobei man im Falle einer Kollision immer die nächst folgende Hashfunktion verwendet. Die Folge von Hashfunktionen bezeichnen wir mit

$$h_0(), h_1(), h_2(), \ldots, h_r().$$

Findet bei Anwendung aller Hashfunktionen eine Kollision statt, so kann das neue Element nicht einsortiert werden.

Soll ein Element gelöscht werden, so wird zunächst nur ein Flag gesetzt, das besagt, dass das Element nicht mehr existiert. Beim Suchen nach einem Element mit Schlüssel k werden der Reihe nach die Hashfunktionen

$$h_1(k), \ldots, h_r(k)$$

durchprobiert, solange, bis man entweder das Element gefunden hat, oder bis man ein wirklich leeren Arrayeintrag gefunden hat. Damit das Suchen wohldefiniert ist, muss beim Löschen eben der besprochene Flag gesetzt werden.

Im folgenden beschreiben wir einige Verfahren um eine neue Adresse im Falle einer Kollision zu finden.

Lineares Sondieren

Bei dieser Methode wird im Falle einer Kollision an der Stelle $a[i]$ einfach im Array die Stelle $a[i+1]$ untersucht. Wenn sie frei ist, kann das neue Element hier einsortiert werden. Ist sie nicht frei, dann wird wieder eine Stelle weitergegangen. Die Folge der Hashfunktionen sieht dann so aus

$$h_i(k) = (h_0(k) + i) \bmod m.$$

Dieses komplizierte Konstrukt muss sein, denn es kann ja sein, dass $h_0(k) + i$ über den adressierbaren Bereich hinausläuft. Ist

$$h_0(k) = k \bmod m$$

gewählt, so ergibt sich

$$h_i(k) = (k + i) \bmod m.$$

Das lineare Sondieren kann zu folgender Formel verallgemeinert werden: Mit einer gegebener Konstante $c \in \mathbb{N}$ setzt man

$$h_i(k) = (k + c \cdot i) \bmod m.$$

Das Verfahren führt dazu, dass im Falle von Kollisionen Ketten mit Abstand c entstehen. Dies bringt jedoch keine wirkliche Verbesserung des Verfahrens. Voraussetzung ist außerdem, dass c und m teilerfremd sind, damit alle Zellen getroffen werden.

Quadratisches Sondieren

In den vorangegangenen Verfahren wird mit einer konstanten Schrittweite gearbeitet. Beim quadratischen Sondieren werden neue Adressen mit quadratischem Abstand erzeugt. Die Idee ist, im Falle einer Kollision, nicht nur die unmittelbare Nachbarschaft zu prüfen, sondern in der Annahme dort auch vorwiegend auf besetzte Felder zu stossen, die Schrittweite mit der Anzahl der Kollisionen zu erhöhen.Dies führt dazu, dass keine Ketten mehr entstehen. Die Folge der Hashfunktionen lautet hier

$$h_i(k) = (h_0(k) + i^2) \bmod m.$$

Wählt man für m eine Primzahl der Form $m = 4 \cdot j + 3$, so werden alle Zellen getroffen ([Rad70]). Konkret hat man also z.B. für $m = 1019$ (Primzahl und bei Division durch 4 Rest 3) und für

$$h_0(k) = k \bmod m$$

$$\begin{aligned} h_0(k) &= k \bmod 1019 \\ h_1(k) &= k + 1 \bmod 1019 \\ h_2(k) &= k + 4 \bmod 1019 \\ h_3(k) &= k + 9 \bmod 1019 \\ h_4(k) &= k + 16 \bmod 1019 \end{aligned}$$

Beispiel 5.6 (Quadratisches Sondieren)
Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Die Arrayelemente nennen wir $a[i]$. Als Hashfunktion $h()$ verwenden wir

$$h(k) = k \bmod m \quad \text{mit } m = 7.$$

Wir wollen bei Kollision quadratisch sondieren. Zunächst erhalten wir mit Hashfunktion h_0

0	1	2	3	4	5	6
63		51	24			

für die Elemente 24, 51 und 63. Bei 77 entsteht eine Kollision bei $a[0]$. Die lösen wir auf, indem wir Hashfunktion $h_1(77)$ anwenden

$$h_1(77) = (h_0(77) + 1) \bmod 7 = 1.$$

Diese Zelle ist noch frei und wir bekommen

0	1	2	3	4	5	6
63	77	51	24			

Das Element $k = 85$ liefert

$$h_0(85) = 1.$$

Diese Zelle ist nun gerade belegt worden. Hätte man 85 vor 77 einsortiert, dann wäre 85 an die Position 1 gekommen. Man sieht, dass die Belegung der Hashtabelle von der Reihenfolge der Eingabedaten abhängt. Wir wenden also h_1 an und erhalten

$$h_1(85) = (h_0(85) + 1) \bmod 7 = (1 + 1) \bmod 7 = 2.$$

Diese Zelle ist auch schon belegt. Also wenden wir h_2 an

$$h_2(85) = (h_0(85) + 4) \bmod 7 = (1 + 4) \bmod 7 = 5.$$

Hier sortieren wir also 85 ein

0	1	2	3	4	5	6
63	77	51	24		85	

Nun soll noch die 99 untergebracht werden. Anwendung von h_0 ergibt

$$h_0(99) = 99 \bmod 7 = 1.$$

Diese Zelle ist aber schon vergeben. Welches h_i bringt 99 auf einen freien Platz? ◀

Bemerkung 5.7 (Quadratisches Sondieren)
Das quadratische Sondieren wird tatsächlich noch etwas anders definiert: man macht alternierend Schritte in quadratischem Abstand nach vorne und nach hinten. Dies ergibt folgende Folge von Hashfunktionen:

$$\begin{aligned}
h_0(k) &= k \bmod m \\
h_1(k) &= k + 1 \bmod m \\
h_2(k) &= k - 1 \bmod m \\
h_3(k) &= k + 4 \bmod m \\
h_4(k) &= k - 4 \bmod m \\
h_5(k) &= k + 9 \bmod m \\
h_6(k) &= k - 9 \bmod m \\
h_{2i-1}(k) &= k + i^2 \bmod m \\
h_{2i}(k) &= k - i^2 \bmod m
\end{aligned}$$

Das vorherige Beispiel zeigt deutlich, warum dies nötig ist. Ein zahlentheoretisches Ergebnis von Radke aus dem Jahr 1970 [Rad70] zeigt, dass alle Zellen getroffen werden, wenn m eine Primzahl der Form

$$m = 4 \cdot j + 3$$

ist. In [CLRS07] wird quadratisches Sondieren so definiert:

$$h_i(k) = (h_0(k) + c_1 i + c_2 i^2) \; mod \; m$$

Dies könnte man zu polynomialem Sondieren verallgemeinern, dies hat sich bisher aber nicht als sinnvoll erwiesen. ◀

Double Hashing

In diesem Fall wird zusätzlich zur Hashfunktion h eine zweite Hashfunktion h' verwendet, welche zur ersten *unabhängig* ist. Dies bedeutet, dass die Wahrscheinlichkeiten bei beiden Hashfunktionen gleichzeitig eine Kollision zu erzeugen unabhängig sind. Oft begnügt man sich damit, eine *intuitiv* unabhängige zweite Hashfunktion h' zu finden. Die Idee ist also, den Schlüsselwerten, die unter der ersten Funktion h eine Kollision haben, mit der zweiten Funktion h' eine individuelle Schrittweite zu geben.

Ist m eine Primzahl und

$$h(k) = k \mod m,$$

dann ist

$$h'(k) = 1 + (k \mod (m-2))$$

eine gute Wahl. Die Folge der Hashfunktionen definiert man dann wie folgt:

$$h_i(k) = (h(k) + h'(k) \cdot i^2) \mod m.$$

Hieran sieht man, dass h' nicht Null sein darf (sonst ändert sich an den h_i im Gegensatz zu h ja nichts. Dafür wird bei h' auch 1 addiert. Erkennt man in den vorangegangenen Verfahren am Index i die Anzahl der Kollisionen, so sollte hier das i klein gegenüber dieser Anzahl sein, womit der Suchaufwand wieder klein gehalten wird! In dem Sinne soll h' zunächst die Kollisionen streuen, bevor mittels dem i in einem festen Abstand freie Feldplätze gesucht werden.

Beispiel 5.8
Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Die Arrayelemente nennen wir $a[.]$. Als Hashfunktion $h()$ verwenden wir

$$h(k) = k \bmod m \quad \text{mit } m = 7.$$

Wir wollen bei Kollision mit der Hashfunktion

$$h'(k) = 1 + (k \mod 5)$$

sondieren. Bei den ersten drei Elementen gibt es keine Kollision

0	1	2	3	4	5	6
63		51	24			

Bei 77 gibt es eine Kollision an Position 0. Diese versuchen wir nun mit h_1 aufzulösen.

$$h_1(77) = (h(77) + h'(77)) \mod 7 = (0 + (1 + (77 \mod 5))) \mod 7 = (0 + (1 + 2)) = 3.$$

Dies ergibt eine Kollision. Wir wenden nun die nächste Hashfunktion h_2 an und erhalten:

$$h_2(77) = (h(77) + h'(77) \cdot 2^2) \mod 7 = (0 + 3 \cdot 2^2) \mod 7 = 12 \mod 7 = 5.$$

Diese Zelle ist frei und wir tragen ein

0	1	2	3	4	5	6
63		51	24		77	

Für 85 erhalten wir

$$h_0(85) = 85 \bmod 7 = 1$$

und für die 99, die in Bsp. 5.6 Probleme machte erhalten wir im ersten Anlauf

$$h_0(99) = 99 \bmod 7 = 1$$

wieder eine Kollisionen. Der zweite Anlauf liefert

$$h_1(99) = (h(99) + h'(99)) \bmod 7 = (1 + (1 + (99 \bmod 5))) \bmod 7 = (1 + (1 + 4)) = 6.$$

und sind nun mit

0	1	2	3	4	5	6
63	85	51	24		77	99

erfolgreich. ◄

Das Schwierige bei der Kollisionsvermeidung bzw. Handhabung der Kollisionen mit offener Adressierung ist, dass sich diese mit der Hashfunktion selbst „vertragen muss“. Eine Kollision kann hier zwei Ursachen haben: Einmal, dass zwei Schlüssel zum selben Hashindex führen und zum Anderen, dass der Hashindex eines Schlüssels wegen Kollisionsbehandlung eines anderen Hashindex bereits besetzt ist. Daher muss auch diese Funktion versuchen, die minimale Anzahl an Kollisionen zu ereichen.

5.4 Löschen von Elementen

Das Löschen von Elementen aus einer Hash-Tabelle mit offener Adressierung erfordert einen Kniff. Folgendes Beispiel beschreibt die Problematik und den Kniff.

Beispiel 5.9
Gegeben seien Elemente mit den folgenden Schlüsseln:

$$\kappa = \{63, 77\}.$$

Als Hashfunktion verwenden wir double hashing aus dem vorherigen Beispiel mit

$$h(k) = k \mod 7 \text{ und } h'(k) = 1 + (k \mod 5).$$

Bei 77 gibt es eine Kollision mit 63. Also fügen wir 77 ein mit $h_1(77) = 3$.

0	1	2	3	4	5	6
63			77			

Nun soll 63 gelöscht werden:

0	1	2	3	4	5	6
			77			

Jetzt wollen wir 77 wiederfinden. Wir suchen bei $h_0(77) = 0$. Dort finden wir die 77 nicht und schließen fälschlicherweise daraus, dass die 77 nicht in der Hashtabelle enthalten ist. Wir wollen ja auch nicht alle Hashfunktionen $h_0, \ldots, h_r$ durchprobieren.

Der Kniff besteht nun darin, an der gelöschten Stelle ein Flag zu setzen, welches kennzeichnet, dass dort einmal ein Element stand. Wir löschen 63 und setzen das Flag:

0	1	2	3	4	5	6
$Flag{=}X$			77			

Jetzt wollen wir 77 wiederfinden. Wir suchen bei $h_0(77) = 0$. Dort finden wir das $Flag = X$. Dies bedeutet, wir müssen bei $h_1(77) = 3$ auch noch suchen und finden dort die 77.

Nun wollen wir 14 einfügen. Es ist $h_0(14) = 0$. Dort ist die Stelle frei. Das Flag beachten wir nicht. Wir können die 14 reinschreiben. Das Flag wird nicht zurückgesetzt. Wir erhalten:

0	1	2	3	4	5	6
$14^{Flag=X}$			77			

Tatsächlich benötigt man in jedem Feld ein Flag. Die Hashtabelle sieht also so aus:

0	1	2	3	4	5	6
$14^{Flag=X}$	$Flag{=}0$	$Flag{=}0$	$77^{Flag=0}$	$Flag{=}0$	$Flag{=}0$	$Flag{=}0$

◄

5.5 load factor, capacity, resize

Die Begriffe *load factor* und *capacity* (Kapazität) beschreiben die Auslastung und Größe einer Hashtabelle. Ist eine Hashtabelle zu voll, dann muss ein *resize* durchgeführt werden. Die folgenden Bemerkungen beziehen sich auf beide Kollisionsvermeidungsstrategien: separate chaining und offene Adressierung.

load factor

load factor gibt an, wie weit eine Hashtabelle ausgelastet ist. Der *load factor* liegt zwischen 0.0 und 1.0. Er bestimmt sich als

$$\textit{load factor} = \frac{\text{Anzahl der belegten Felder}}{\text{Gesamtgröße } m \text{ der Hashtabelle}}.$$

Ist ein maximaler *load factor*, beispielsweise 0,8 bzw. 80%, überschritten, so wird die Hashtabelle ineffektiv und muss vergrößert werden.

Wir ein minimaler *load factor* unterschritten, so wird zu viel Speicher verbraucht.

capacity

Die *capacity* ist einfach die Größe m der Hashtabelle.

Die Hashtabelle soll automatisch vergrößert werden wenn der *load factor* einen maximalen *load factor*, beispielsweise 80%, überschritten hat. Man erhält eine neue *capacity* m_{neu}. In diesem Fall spricht man vom *resize* einer Hashtabelle.

Die Defaultwerte in Java sind für den *load factor* 0.75 und für die *capacity* 11, d.h. Java vergrössert die Hashtabelle, wenn die bestehende Tabelle zu 75% ausgelastet ist.

resize

Beim *resize* wird eine neue *capacity* m_{neu} bestimmt. Diese wird so gewählt, dass die bereits eingefügten Elemente einen neuen minimalen *load factor*, beispielsweise 50%, ergeben.

Es ist zu beachten, dass beim *resize* alle bereits eingefügten Elemente neu eingefügt werden müssen. Die Hashfunktionen haben sich ja nun verändert. Statt $\mod m$ hat man nun $\mod m_{neu}$.

5.6 Analyse der Hashverfahren

Bei den hier betrachteten Hashverfahren ist der worst case schlecht. Wer die verwendete Hashfunktionen kennt, kann Werte wählen, die alle auf den gleichen Wert abgebildet werden. Wird etwa die Divisionsmethode mit einer Primzahl p und lineares Sondieren gewählt, so erhält man für die Werte $k \cdot p, k = 1, \ldots p$ im ersten Versuch immer 0, im zweiten 1, im $k - ten$ k, so dass die durchschnittliche Suchzeit asymptotisch ein $\Theta(n)$ ist, wenn $n < p$ Werte gehashed werden. Analoge Überlegungen gelten für separate chaining als Kollisionsvermeidungsstrategie.

Ein Mittel dagegen ist die zufällige Auswahl der Hash-Funktionen aus einer geeignet gewählten Menge, siehe [CLRS07] (Stichwort: universal hashing).

Für separate chaining betrachten wir eine Hash-Tabelle mit m Plätzen, die n Elemente enthält. Der load factor ist dann $\alpha = \frac{n}{m}$. In diesem Fall kann α also auch größer als 1 werden. Wir untersuchen nun wie sich das Verhalten für konstantes α und $n, m \longrightarrow \infty$ entwickelt. Der worst case Fall ist wieder der zum Anfang dieses Abschnitts beschriebene: Alle Schlüssel kommen auf einen Platz der Tabelle in eine Liste der Länge n. Dann ist die Zeit zum Suchen $\theta(n)$ und es muss noch die Hash-Funktion berechnet werden.

Das durchschnittliche Verhalten beim Suchen können wir unter der Annahme, dass die Hash-Funktion alle Schlüssel mit gleicher Wahrscheinlichkeit auf die Tabellenplätze verteilt, relativ einfach analysieren. Unter dieser Annahme befinden sich im Durchschnitt $\alpha = \frac{n}{m}$ Einträge in

der Liste an einem Tabellenplatz. Sie nach einem Eintrag zu durchsuchen ist also ein $\Theta(\alpha)$, wenn die Suche nicht erfolgreich ist, denn dann müssen wir bis zum Ende der Liste suchen. Für die Berechnung der Hashfunktion nehmen wir, an dass diese einen konstanten Aufwand verursacht: $\Theta(1)$. Damit haben wir bewiesen:

Satz 5.10 (Erfolgloses Suchen mit separate chaining)
Unter der Voraussetzung, dass die Hash-Funktion die Schlüssel mit gleicher Wahrscheinlichkeit auf die Tabellenplätze verteilt, und Kollisionen durch separate chaining aufgelöst werden, ist der durchschnittliche Aufwand für eine erfolglose Suchoperation $\Theta(1+\alpha)$. ◄

Dieses Ergebnis gilt aber auch für die erfolgreiche Suche:

Satz 5.11 (Erfolgreiches Suchen mit separate chaining)
Unter der Voraussetzung, dass die Hash-Funktion die Schlüssel mit gleicher Wahrscheinlichkeit auf die Tabellenplätze verteilt, und Kollisionen durch separate chaining aufgelöst werden, ist der durchschnittliche Aufwand für eine erfolgreiche Suchoperation $\Theta(1+\alpha)$. ◄

Beweis: Wir nehmen an, dass die Schlüssel gleichwahrscheinlich auf die Plätze gebracht werden. Ferner können wir ohne Einschränkung annehmen, dass die insert Operation das Element am Ende einer Liste einfügt. Aus Kap. 1 wissen wir ja, dass dies das asymptotische Verhalten nicht beeinflusst. Ist das gesuchte Element das i-te Element auf einem Platz, so ergibt sich für die durchschnittliche Anzahl, die durchsucht werden muss:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m}) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+(\frac{1}{nm})(\frac{(n-1)n}{2})\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}\end{aligned}$$

Der Gesamtaufwand für eine erfolgreiche Suche ist damit $\Theta(2+\frac{\alpha}{2}-\frac{1}{2m}) = \Theta(1+\alpha)$.

Aus diesen Überlegungen ergibt sich nun dies: Ist die Anzahl Plätze in der Hash-Table proportional zur Anzahl Einträge, also $n = \mathcal{O}(m)$, so ist $\alpha = \frac{n}{m} = \frac{\mathcal{O}(m)}{m} = \mathcal{O}(1)$. Damit können alle Operationen im Durchschnitt mit einem Aufwand von $\mathcal{O}(1)$ unterstützt werden.

Kommen wir nun zur Untersuchung des Verhaltens bei der offenen Adressierung. Die Position eines Schlüssels in der Tabelle hängt von der Reihenfolge beim Einfügen ab. Wir nehmen für die Analyse deshalb an, dass für jeden Schlüssel k die Folge der Hash-Funktionen $h_0(k), h_1(k), \ldots h_{m-1}(k)$mit gleicher Wahrscheinlichkeit eine der $m!$ Permutationen von $\{0,1,2,\ldots m-1\}$ ist. Diese Annahme bezeichnen wir als *uniform hashing*.

Satz 5.12 (Offene Adressierung, erfolglose Suche)
Unter der Voraussetzung des uniform hashing gilt für eine Hash-Tabelle mit load factor $\alpha = \frac{n}{m} < 1$ ist die erwartete Sondierungstiefe (Erwartungswert) bei einer erfolglosen Suche höchstens $\frac{1}{1-\alpha}$. ◄

Aus Satz 5.12 folgt sofort:

Satz 5.13 (Offene Adressierung, einfügen)
Unter der Voraussetzung des uniform hashing gilt für eine Hash-Tabelle mit load factor $\alpha = \frac{n}{m} < 1$ dass die erwartete Sondierungstiefe (Erwartungswert) beim Einfügen höchstens $\frac{1}{1-\alpha}$ ist. ◄

Satz 5.14 (Offene Adressierung, erfolgreiche Suche)
Unter der Voraussetzung des uniform hashing gilt für eine Hash-Tabelle mit load factor $\alpha = \frac{n}{m} < 1$ ist die erwartete Sondierungstiefe (Erwartungswert) bei einer erfolgreichen Suche höchstens $\frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}$. ◄

Für die Beweise von Satz 5.12 – 5.14 verweise ich auf [CLRS07].

5.7 Fazit aus Sicht der Konstruktionslehre

Suchalgorithmen[6], die mit Hashing arbeiten, bestehen aus zwei Teilen.

1. Im ersten Schritt transformiert der Algorithmus den Suchschlüssel mithilfe einer Hashfunktion in eine Tabellenadresse.

   Diese Funktion bildet im Idealfall unterschiedliche Schlüssel auf unterschiedliche Adressen ab. Oftmals können aber auch zwei oder mehrere unterschiedliche Schlüssel zur gleichen Tabellenadresse führen.

2. Somit führt eine Hashing-Suche im zweiten Schritt eine Kollisionsbeseitigung durch, die sich mit derartigen Schlüsseln befasst.

Ein hier behandeltes Verfahren zur Kollisionsbeseitigung verwendet verkettete Listen und ist somit unmittelbar in solchen Situationen nützlich, bei denen sich die Anzahl der Suchschlüssel nur schwer im Voraus angeben lässt. Das andere Verfahren zur Kollisionsbeseitigung erzielt kurze Suchzeiten für Elemente, die in einem Array fester Größe gespeichert sind.

Hashing ist ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf. Wenn es keine Speicherbeschränkung gäbe, könnten wir jede Suche mit nur einem einzigen Speicherzugriff realisieren, indem wir einfach den Schlüssel als Speicheradresse analog zu einer schlüsselindizierten Suche verwenden. Allerdings lässt sich dieser Idealzustand meistens nicht verwirklichen, weil der Speicherbedarf besonders bei langen Schlüsseln viel zu groß ist. Gäbe es andererseits keine Zeitbeschränkung, könnten wir mit einem Minimum an Speicher auskommen, indem wir ein sequenzielles Suchverfahren einsetzen. Hashing erlaubt es, zwischen diesen beiden Extremen ein vertretbares Maß sowohl für die Zeit als auch den Speicherplatz zu finden. Insbesondere können wir jedes beliebige Verhältnis einstellen, indem wir lediglich die Größe der Hashtabelle anpassen; dazu brauchen wir weder Code neu zu schreiben noch auf andere Algorithmen auszuweichen.

Hashing ist ein klassisches Problem der Informatik: Die verschiedenen Algorithmen sind recht gründlich untersucht worden und haben weite Verbreitung gefunden. Grob betrachtet können wir davon ausgehen, dass Hashing die Operationen Suchen und Einfügen von Symboltabellen in konstanter Zeit unabhängig von der Größe der Tabelle unterstützt. Diese Erwartung ist die theoretische Optimalleistung für jede Implementierung von Symboltabellen, wobei aber Hashing aus zwei Gründen kein Wundermittel ist. Erstens hängt die Laufzeit von der Länge des Schlüssels ab, was in praktischen Anwendungen mit langen Schlüsseln ein Hindernis darstellen kann. Zweitens bietet Hashing keine effizienten Implementierungen für andere Operationen der Symboltabellen wie etwa Auswählen oder Sortieren.

5.8 Kryptograpische Hashverfahren

Laden Sie sich eine Datei aus dem Internet auf Ihren Rechner, so möchten Sie sicherstellen

- dass es sich um diese Datei handelt und keine veränderte
- dass sie auf dem Weg vom Server zu Ihrem Rechner nicht verändert wurde.

Zu diesem Zweck finden Sie auf manchem Servern entsprechende Codes, z. B. MD5. Es gibt frei verfügbare Software wie CrypTool oder Jacksum, mit denen Sie diese Codes selbst überprüfen können.

Um diese Codes zu berechnen werden ebenfalls Hash-Funktionen eingesetzt, die Zielrichtung ist aber eine ganz andere als bei den bisher betrachteten. Ein Hash-Funktion, wie sie in diesem Abschnitt behandelt wird, berechnet aus einer Datei einen Wert, der etwa 32 oder 64 Bit lang ist. Dieser Wert soll so beschaffen sein, dass nach einer Änderung der Datei ein anderer Wert errechnet wird. Anders formuliert: Es wird angestrebt, dass verschiedene Dateien auch zu verschiedenen Werten führen, es also keine Kollisionen gibt. So soll vermieden werden, dass einem eine andere

[6] Dies ist [Sed02] entnommen worden.

Datei — sozusagen ein Trojaner — untergeschoben wird. Wie wir von den bisher betrachteten Hash-Funktionen wissen, kann man Kollisionen nicht vermeiden. Eine Kollision hat hier aber eine andere Bedeutung, als bei den anderen Hash-Funktionen, bei denen sie durch die beschriebenen Verfahren aufgelöst werden kann: Das Erzeugen einer Kollision kompromittiert das Verfahren.

Um die Grundprinzipien dieser Verfahren zu erklären, beginnen ich mit dem einfachsten *Message Digest Algorithmus* MD2.

Dieser Algorithmus wird in RFC 1115 und in RFC 1319 beschrieben. Ich folge in dieser Darstellung letzterem. Er ermittelt aus einer beliebigen Byte-Folge 128 Bit „Fingerabdruck“ oder „message digest“. Die Hoffnung war, dass es nicht möglich sei zwei verschiedene Nachrichten zu erzeugen, die den gleichen Fingerabdruck liefern.

Ich verwende zur Beschreibung folgende Konvention:

- Die Nachricht, die zu verarbeiten ist besteht aus einer Folge $(m_i)_{i=0,\ldots,b-1}$von Bytes.

Der MD2 Algorithmus arbeitet in fünf Schritten:

Schritt 1: Die Nachricht wird mit 1 bis 16 Bytes so aufgefüllt, dass ihre Länge ein Vielfaches von 16 ist. Selbst wenn die Länge b ein Vielfaches von 16 ist geschieht dies.

Der Algorithmus hierfür ist: Ist $b \; mod \; 16 = c$ so ist

$$b_i = 16 - c \forall i = b, \ldots, b + 16 - c.$$

Die Länge der Nachricht ist nun n.

Schritt 2 Nun wird für die verlängerte Nachricht eine 16 Byte lange Prüfsumme berechnet und ebenfalls angefügt. Zur Berechnung der Prüfsumme wird eine Permutation der Zahlen von $0 \ldots 255$ verwendet, die aus den Nachkommastellen von π konstruiert wurde. Sie folgt hier,

```
41, 46, 67, 201, 162, 216, 124, 1, 61, 54, 84, 161, 236, 240, 6,
19, 98, 167, 5, 243, 192, 199, 115, 140, 152, 147, 43, 217, 188,
76, 130, 202, 30, 155, 87, 60, 253, 212, 224, 22, 103, 66, 111, 24,
138, 23, 229, 18, 190, 78, 196, 214, 218, 158, 222, 73, 160, 251,
245, 142, 187, 47, 238, 122, 169, 104, 121, 145, 21, 178, 7, 63,
148, 194, 16, 137, 11, 34, 95, 33, 128, 127, 93, 154, 90, 144, 50,
39, 53, 62, 204, 231, 191, 247, 151, 3, 255, 25, 48, 179, 72, 165,
181, 209, 215, 94, 146, 42, 172, 86, 170, 198, 79, 184, 56, 210,
150, 164, 125, 182, 118, 252, 107, 226, 156, 116, 4, 241, 69, 157,
112, 89, 100, 113, 135, 32, 134, 91, 207, 101, 230, 45, 168, 2, 27,
96, 37, 173, 174, 176, 185, 246, 28, 70, 97, 105, 52, 64, 126, 15,
85, 71, 163, 35, 221, 81, 175, 58, 195, 92, 249, 206, 186, 197,
234, 38, 44, 83, 13, 110, 133, 40, 132, 9, 211, 223, 205, 244, 65,
129, 77, 82, 106, 220, 55, 200, 108, 193, 171, 250, 36, 225, 123,
8, 12, 189, 177, 74, 120, 136, 149, 139, 227, 99, 232, 109, 233,
203, 213, 254, 59, 0, 29, 57, 242, 239, 183, 14, 102, 88, 208, 228,
166, 119, 114, 248, 235, 117, 75, 10, 49, 68, 80, 180, 143, 237,
31, 26, 219, 153, 141, 51, 159, 17, 131, 20
```

die Erklärung ihres Zustandekommens ist für die nächste Auflage geplant. Nun wird in 16-Byte Blöcken (innere Schleife) die Folge der 16 Bytes der Prüfsumme ermittelt:

```
/* Clear checksum. */
For i = 0 to 15 do:
   Set C[i] to 0.
end /* of loop on i */
```

```
Set L to 0.
  For i = 0 to N/16-1 do

   /* Checksum block i. */
   For j = 0 to 15 do
      Set c to M[i*16+j].
      Set C[j] to S[c xor L].
      Set L to C[j].
    end /* of loop on j */
 end /* of loop on i */
```

Diese 16 Bytes der Prüfziffer werden der Nachricht ebenfalls hinzugefügt, so dass wir nun eine Folge der Länge $n' = n + 16$ haben.

Schritt 3 Initialisiere den message digest Buffer: Eine Folge $(x_i)_{i=0\ldots48}$ wird mit 0 initialisiert.

Schritt 4 Dann werden die Elemente dieser Folge so berechnet:

```
  For i = 0 to N'/16-1 do
   /* Copy block i into X. */
   For j = 0 to 15 do
      Set X[16+j] to M[i*16+j].
      Set X[32+j] to (X[16+j] xor X[j]).
    end /* of loop on j */

   Set t to 0.

   /* Do 18 rounds. */
   For j = 0 to 17 do

      /* Round j. */
      For k = 0 to 47 do
         Set t and X[k] to (X[k] xor S[t]).
      end /* of loop on k */

      Set t to (t+j) modulo 256.
   end /* of loop on j */
end /* of loop on i */
```

Schritt 5 Ausgabe: Der message digest steht in $(x_i)_{i=1,\ldots,15}$ und wird nun ausgegeben.

Beispiel 5.15 (MD2)
Wir betrachten die Nachricht

Der Advokat
aß grad Salat
als ihm ein Schraat
die Saat
zertrat

Da es auch von den Blanks etc. abhängt schreibe ich das Gedicht in einen Satz

```
          1         2         3         4         5         6
i 012345678901234567890123456789012345678901234567890123456789012345678901
b Der Advokat aß grad Salat als ihm ein Schraat die Saat zertrat
```

Als MD2 message digest erhält man nun z. B. mittel *jacksum* oder *CrypTool*:
7B 72 90 BA D6 1A 2E 3B 6D 10 88 B2 FD B0 E3 BC ◀

5.9 Historische Anmerkungen

Der MD2 Algorithmus wurde 1988 von Ronald L. Rivest veröffentlicht. Eine der frühen Veröffentlichungen des Geburtstagsparadoxon ist [Fel50], Abschnitt II.3. Der Autor beklagt sich im Vorwort scherzhaft, über ungenaue Zitate: Er habe viele Beispiele aus nichtmathematischen Arbeiten aufgegriffen. Inzwischen würden neuere Arbeiten so zitieren, als wenn die Originalquellen seine Beispiele verwendet hätten.

5.10 Aufgaben

1. Beweisen Sie bitte:

$$(a \cdot b \cdot c) \bmod m = ((((a \bmod m) \cdot (b \bmod m)) \bmod m) \cdot (c \bmod m)) \bmod m$$

   !

2. Welches h_i bringt in Beispiel 5.6 die 99 auf einen freien Platz?

3. Was passiert genau, wenn man als Hash-Funktion

$$h(k) = k \bmod 2^m$$

   $m > 0$ wählt? Unter welchen Bedingungen ist dies sinnvoll?

4. Ist eine Hash-Funktion der Form

$$h(k) = k \bmod 3 \cdot m$$

   zum Hashen von alphabetischen Schlüsseln sinnvoll?

5. Unter welchen Bedingungen ist eine Hash-Funktion der Form

$$h(k) = k \bmod 2 \cdot m$$

   sinnvoll und unter welchen nicht?

6. 6.1. Suchalgorithmen, die mit Hashing arbeiten, bestehen aus zwei Teilen. Welche sind dies?
   6.2. Welche Ziele verfolgen die einzelnen Teile?
   6.3. Warum ist Hashing ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf?

7. Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

   in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = kmod5$ in eine Hashtabelle der Größe $N = 5$ (nummeriert von 0 bis 4) ein. Die Kollisionsbehandlung erfolge durch verkettete Listen.

   Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion an.

8. Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod\ 9$ in eine Hashtabelle der Größe $N = 9$ (nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge durch quadratisches Sondieren ($h_i(k) = (h(k) + i^2)$ mod 9).

Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h bzw. bei Kollisionen der Hashfunktionen h_i an.

9. Fügen Sie die Zahlen

$$13, 6, 27, 11, 16, 17, 19, 18, 42$$

in dieser Reihenfolge unter Verwendung folgender (Mittel-Quadrat-Methode) Hashfunktion in eine Hashtabelle der Größe $N = 11$ (nummeriert von 0 bis 10) ein:

$$qk = k^2; \quad st = qk[3, 2]; \quad h(st) = st\ mod\ 11$$

Dabei liefert $qk[3, 2]$ die Stellen 3 und 2 (von rechts zählend), also z.B. $5291763[3, 2] = 76$ oder $54321[3, 2] = 32$ sowie $12[3, 2] = 1$. Die Kollisionsbehandlung erfolge durch $h_i(k) = (h_0(k) + i^2)\ mod\ 11$.

Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch jeweils das Ergebnis der Hashfunktion an.

Kapitel 6

Komprimierung und Kryptographie

6.1 Übersicht

Manche meinen zwar, Plattenplatz würde „nichts mehr kosten“. Wäre diese Behauptung korrekt, so könnte man auf Komprimierung vielleicht verzichten. Andererseits sind die Größen von Speichermedien (Platten) schneller gewachsen als die Zugriffsgeschwindigkeit (vgl. den Vortrag von Rudolf Bayer in [BD02]). Es dauert also länger als früher, die größte verfügbare Platte zu durchsuchen. Andere meinen, Komprimierungsalgorithmen könnten noch so leistungsfähig werden, dass man Videokonferenzen über 9.600 Baud Leitungen abhalten könne. Derart allgemeine Aussagen sind wohl nicht belegbar. Auf jeden Fall stellen wir aber fest, dass sowohl Verfahren, die gespeicherte Daten als statische Objekte komprimieren als auch Verfahren, die dynamische übertragene Daten (Stichwort: Video) komprimieren, nützlich sein können. Ob sich Komprimierung etwas auf einer Platte lohnt, hängt von der Verarbeitungscharakteristik ab. Komprimierung ist sinnvoll:

- Wenn Sie hinreichend viel Plattenplatz einsparen. Die Formulierung „hinreichend“ weist bereits auf den Beurteilungsspielraum hin.
- Wenn Operationen, wie das Öffnen einer Datei, nach Komprimierung nicht wesentlich langsamer, vielleicht sogar schneller ablaufen. Dabei ist letztendlich gegenüberzustellen:
  - Benötigte Zeit, um eine nichtkomprimierte Datei zu laden bzw. zu schreiben.
  - Benötigte Zeit, um eine komprimierte Datei zu laden zuzüglich der Zeit zum Dekomprimieren zum Bearbeiten und Komprimieren beim Speichern.

Komprimiert man einen Text, so wird man erwarten, nach Dekomprimierung den Text wieder unverändert vorzufinden. In anderen Fällen kann man auf die exakte Reproduktion verzichten. So kann man ein Bild statt hochauflösend auch mit einer geringeren Auflösung speichern. Das reduziert das Datenvolumen, allerdings unter Verzicht auf Details, die nur bei hoher Auflösung erkennbar sind. Für manche Zwecke mag dies ausreichen. Ferner kann man mit festen Tabellen für die Komprimierung arbeiten (statisch) oder die Daten analysieren um zu einer guten Kompression zu kommen (dynamisch). So kann man Komprimierungsverfahren nach diesen zwei Kriterien klassifizieren:

	verlustfrei	verlustbehaftet
statisch	RLE	MPEG
		JPEG
		Fraktal
dynamisch	Huffman	JPEG

Im Laufe dieses Kapitels werde ich eine Reihe von Verfahren darstellen und damit die Namen in dieser Matrix zumindest teilweise mit Inhalt füllen.

Im letzten Abschnitt des letzten Kapitels wurden Message Digest Algorithmen vorgestellt. Dabei musste die Frage offenbleiben, wie die Hash-Werte zur Kontrolle sicher übertragen werden können. Hier werden nun einige Verschlüsselungsverfahren vorgestellt. In dieser Version werde ich mich dabei stark auf die freie Software CrypTool stützen, die im AI-Labor installiert ist.

6.2 Lernziele

- Die wichtigsten Prinzipien der verlustfreien und verlustbehafteten Komprimierung kennen.
- Die Standardverfahren der verlustfreien Komprimierung kennen und implementieren können.
- Beurteilen können wann Komprimierung sinnvoll eingesetzt werden kann und wann nicht.

6.3 Verlustfreie Komprimierung

Bemerkung 6.1 (Nicht jede Datei läßt sich verkleinern)
Für jedes allgemein einsetzbare, verlustfreie Komprimierungsverfahren gibt es eine Datei, die durch seine Anwendung nicht verkleinert wird.

Das dies so ist zeigt man durch einen Widerspruchsbeweis: Angenommen ein Verfahren K würde jede Datei verkleinern. Sei nun D eine beliebige Datei und $size(D) =: N$ hier Größe (z. B. in Bits). Nach Annahme ist $size(K(E)) < size(E)$ für jede Datei E, insbesondere für alle $K^n(D), n = 1, \ldots, N$. Damit wäre dann aber $size(K^N(D)) = 0$ und daraus lässt sich D mit Sicherheit nicht rekonstruieren. ◄

Anderer seits sagt man ja auch: „Jedes Programm hat (mindesten) einen Fehler.“ Würde Bem. 6.1 nicht gelten, so könnte man ja jedes Programm auf eine Zeile verkürzen, die fehlerhaft ist, also nicht funktioniert.

6.3.1 Lauflängenkodierung

Ein einfaches und nützliches Verfahren nutzt Folgen gleicher Zeichen aus: Je länger eine solche Folge ist, um so mehr spart man ein, wenn man sie durch das Zeichen und die vorkommende Anzahl ersetzt. Enthält ein Text keine Ziffern, so kann man einfach „*aaaaaaaaaa*“ durch „10*a*“ ersetzen. Enthält der Text Zahlen, so benötigt man eine Escape-Sequenz. Im einfachsten Fall ist dies ein Zeichen, dass nicht im Text vorkommt. Kann man z. B. „\“ hierfür wählen, so lohnt sich eine Ersetzung ab drei gleichen Zeichen. Nun kann man aber nicht davon ausgehen, dass irgendein Zeichen nie in einer zu komprimierenden Datei vorkommt. Es könnte ja jemand versuchen, eine bereits komprimierte Datei erneut mit dem gleichen Verfahren zu komprimieren.

Für binäre Dateien kann man dieses Verfahren noch weiter optimieren. Binäre Dateien enthalten nur die Zeichen „0“ und „1“. Auf einen Lauf von „0-en“ folgt also ein lauf von „1-en“ usw. Man kann also auf die Speicherung der Werte im Wesentlichen verzichten. Man muss sich nur merken, ob es mit „0“ oder „1“ losgeht und dann die Längen der Läufe speichern.

Grob sieht der Algorithmus etwa so aus:

```
procedure bbRLE(inFile,outFile)
        ch1 := get(inFile)
        ch2 := get(inFile)
        while (not done) do
                if (ch1 = ch2) then
                 Start of run!
                        put(outFile,ch1)
                        put(outFile,ch2)
```

```
                count := 0
                 How many follow?
                repeat
                        ch1 := get(inFile)
                        if (ch1 = ch2) then
                        count := count + 1
                until (ch1 != ch2)
                 End of run
                put(outFile,count)
                ch2 := get(inFile)
        else
                 No run: check next pair
                put(outFile,ch1)
                ch1 := ch2
                ch2 := get(inFile)
```

6.3.2 Komprimierung unter Berücksichtigung von Häufigkeiten

Wenn die Zeichen nicht gleichwahrscheinlich über den den Text verteilt sind, so muss man nicht jedes Zeichen mit der gleichen Anzahl Bits kodieren. Eine Vorstellung von den Auswirkungen geben Häufigkeitsstatistiken für Buchstaben in verschiedenen Sprachen [Sin00].

	Häufigkeit in %			Häufigkeit in %	
Buchstabe	Deutsch	Britisch	Buchstabe	Deutsch	Britisch
a	6,51	8,2	n	9,78	6,7
b	1,89	1,5	o	2,51	7,5
c	3,06	2,8	p	0,79	1,9
d	5,08	4,3	q	0,02	0,1
e	17,40	12,7	r	7,00	6,0
f	1,66	2,2	s	7,27	6,3
g	3,01	2,0	t	6,15	9,1
h	4,76	6,1	u	4,35	2,8
i	7,55	7,0	v	0,67	1,0
j	0,27	0,2	w	1,89	2,4
k	1,21	0,8	x	0,03	0,2
l	3,44	4,0	y	0,04	2,0
m	2,53	2,4	z	1,13	0,1

Es macht den Eindruck, als wenn diese Verteilung sehr langfristig stabil ist. Sie verschiebt sich nur graduell mit Rechtschreibreformen und dem sich wandelnden Sprachgebrauch, sieht man einmal von unterschiedlichen Dialekten ab, wie etwa Pidgin English etc.

Schnelle radikale Änderungen, wie sie Mark Twain vorgeschlagen haben soll, dürften sich kaum so schnell durchsetzen.

> A Plan for the Improvement of English Spelling
>
> For example, in Year 1 that useless letter “c” would be dropped to be replased either by “k” or “s”, and likewise “x” would no longer be part of the alphabet. The only kase in which “c” would be retained would be the “ch” formation, which will be dealt with later. Year 2 might reform “w” spelling, so that “which” and “one” would take the same konsonant, wile Year 3 might well abolish “y” replasing it with “i” and Iear 4 might fiks the “g/j” anomali wonse and for all. Jenerally, then, the improvement would kontinue iear bai iear with Iear 5 doing awai with useless double konsonants, and Iears 6-12 or so modifaiing vowlz and the rimeining voist and unvoist konsonants. Bai Iear 15 or sou, it wud fainali bi posibl tu meik ius ov thi ridandant letez “c”, “y”

> and “x” – bai now jast a memori in the maindz ov ould doderez – tu riplais “ch”, “sh”, and “th” rispektivli. Fainali, xen, aafte sam 20 iers ov orxogrefkl riform, wi wud hev a lojikl, kohirnt speling in ius xrewawt xe Ingliy-spiking werld.

Eine ernstzunehmende Quelle hierfür habe ich nicht, Hinweise werden gerne genommen.

Eines der bekanntesten Verfahren, dass die Häufigkeit der Zeichen berücksichtigt, ist das von Huffman. Dieses Verfahren ermittelt zunächst die Häufigkeit der einzelnen Zeichen in der zu kodierenden Zeichenfolge. Dazu werden die entsprechenden Anzahlen in einem Array festgehalten. Betrachten wir das lateinische Alphabet ohne Umlaute und nur Großbuchstaben, so hat man ein Array `count[27]` zu füllen, wobei `count[0]` für das Leerzeichen verwendet wird. Entsprechend der Häufigkeiten der Zeichen werden die Codierungen festgelegt.

Dabei müssen wir auf Einiges achten:

Beispiel 6.2 (Huffman - Eindeutigkeit)
Wir betrachten die folgende Tabelle aus dem Anfang des deutschen Alphabets. Nach der vorstehend angegebenen Häufigkeitsverteilung ist die absteigende Häufigkeit dieser vier Zeichen a, d, c, b. Wir codieren also wie folgt

Zeichen	Coodierung
a	0
b	11
c	01
d	1

Hier taucht aber ein Problem auf: Was ist, wenn die ersten Zeichen der Codierung übereinstimmen? Was bedeutet z. B. „0111“? Das ist absolut nicht klar! Hier einige Möglichkeiten:

- addd (0 1 1 1)
- adb (0 1 11)
- abd (0 11 1)
- cdd (01 1 1)
- cb (01 11)

Wir müssen als darauf achten, dass keine zwei Codierungen gleich beginnen. Dies erreichen wir durch die folgende Codierung

Zeichen	Coodierung
a	0
b	101
c	100
d	11

In diesem Fall ist eindeutig, was z. B. „0111“ bedeutet: a (0) + irgendwas. Die Abb. 6.1 zeigt diese Codierung als Binärbaum. ◀

Um diese Idee zu einem Verfahren auszubauen, wird zunächst ein Binärbaum nach folgendem Algorithmus aufgebaut, der in seinen Blättern die Zeichen und ihre Häufigkeiten enthält. In einem zweiten Schritt bestimmen wir dann die Kodierung aus den Häufigkeiten. Wir beginnen mit Blattknoten, die ein Zeichen und die Anzahl dessen Auftretens enthalten. Diese arbeiten wir von kleinen zu großen Häufigkeiten ab.

1. Schritt: Feststellen der Häufigkeiten
2. Schritt: Codebaum „von unten“ aufbauen

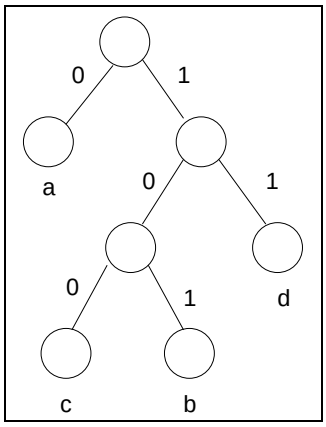


Abbildung 6.1: Binärbaum dieser Codierung

2.1. Knoten sind (zunächst) nur Zeichen mit ihrer Häufigkeit!

2.2. Die beiden Knoten mit den kleinsten Werten mit einem neuen Vorgänger verbinden. Dessen Wert ist die Summe der beiden Werte!

2.3. Die beiden gewählten Knoten durch den neuen ersetzen. Bei mehr als einer Möglichkeit: Zufällig auswählen! Links oder rechts ist gleichgültig.

2.4. Dies solange wiederholen, bis nur noch ein Knoten vorhanden ist.Der letzte Knoten ist die Wurzel des Codebaums!

Anschließend ordnen wir die Codezeichen den Blättern zu: Einmal nach links ist 0, einmal nach rechts ist 1.

Beispiel 6.3 (Aufbau Huffman-Tree)
Als Beispiel wählen wir den Text „MISSISSIPPI-MISSOURI“. Die folgende Tabelle zeigt die Häufigkeiten mit der die einzelnen Zeichen vorkommen.

M	I	S	P	-	O	U	R
2	6	6	2	1	1	1	1

Diese nehmen wir als Basis für den Aufbau des Baums. Wir sehen als Erstes in Abb. 6.2. Diese

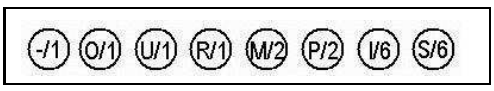


Abbildung 6.2: Huffman: Die „unterste“ Ebene

enthält die Zeichen, zum besseren Verständnis und zum Aufbau des Baums noch mit ihren Häufigkeiten versehen sind. Im nächsten Schritt bauen wir uns aus den Knoten (-,1) und (0,1) den Knoten mit der Häufigkeit $2 = 1 + 2$. Diesen setzen wir an Stelle der bisherigen beiden Knoten und machen diese zu Söhnen[1]. Das Ergebnis sehen wir in Abb. 6.3. Anschließend nehmen wir uns die Knoten mit den nächst größeren Häufigkeiten vor. Das sind hier (U,1), (R,1). Wieder entsteht ein neuer Knoten und unser Baum wächst. Unter den so entstandenen Knoten suchen wir wieder die beiden mit den niedrigsten Häufigkeiten. man beachte, dass uns nun nur noch die Häufigkeiten interessieren. Aus diesen bilden wir einen neuen Knoten, wieder mit der Summe der Häufigkeiten. Dies Verfahren setzen wir nun weiter fort mit (M,2),(P,2) und bilden auch über diesen einen neuen

[1] Hier sieht man mal wieder die Gefahren des Anthropomorphismus: Wie kann man jemand zum Sohn machen? Durch Adoption? Zum Vater schon eher ...

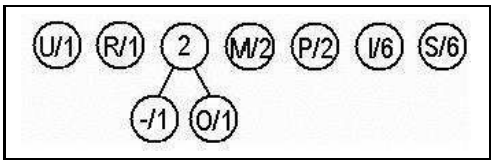


Abbildung 6.3: Huffman: Schritt 2

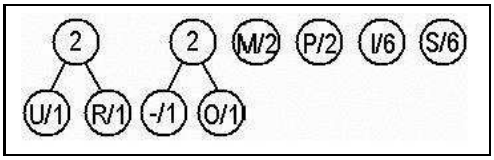


Abbildung 6.4: Huffman: Schritt 3

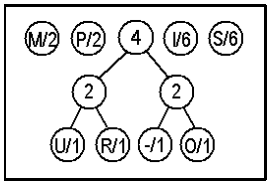


Abbildung 6.5: Huffman: Schritt 4

Knoten. Über diesen beiden neuen Knoten bilden wir nun nach dem bereits beschriebenen Schema den nächsten Knoten, siehe Abb. 6.6 Ganz entsprechend geht es nun über Abb. 6.7, weiter, bis

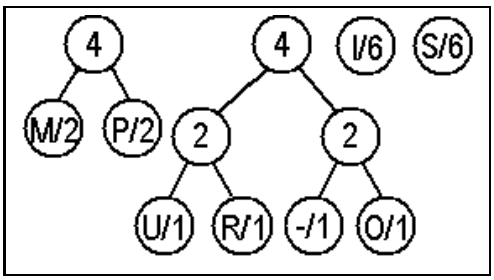


Abbildung 6.6: Huffman: Schritt 5

der ganze Codebaum aufgebaut ist. Nun haben wir unsere Verschlüsselung:

I	S	M	P	-	O	U	R
10	11	000	001	0110	0111	0100	0101

◀

Dies baut man nun so aus, dass die inneren Knoten die Codierung enthalten und in den Blättern das durch sie codierte Zeichen steht. Dieser Baum wird so aufgebaut, das die erwartete Dekodierungszeit minimal wird. Wie wir wissen, hängen Operationen auf Binärbäumen vor allem von der Höhe des Baumes ab. Sei d_i die Tiefe des Knotens für das $i-$te Zeichen und

$$q_i = \frac{count[i]}{\sum_{i=0}^{26} count[i]}$$

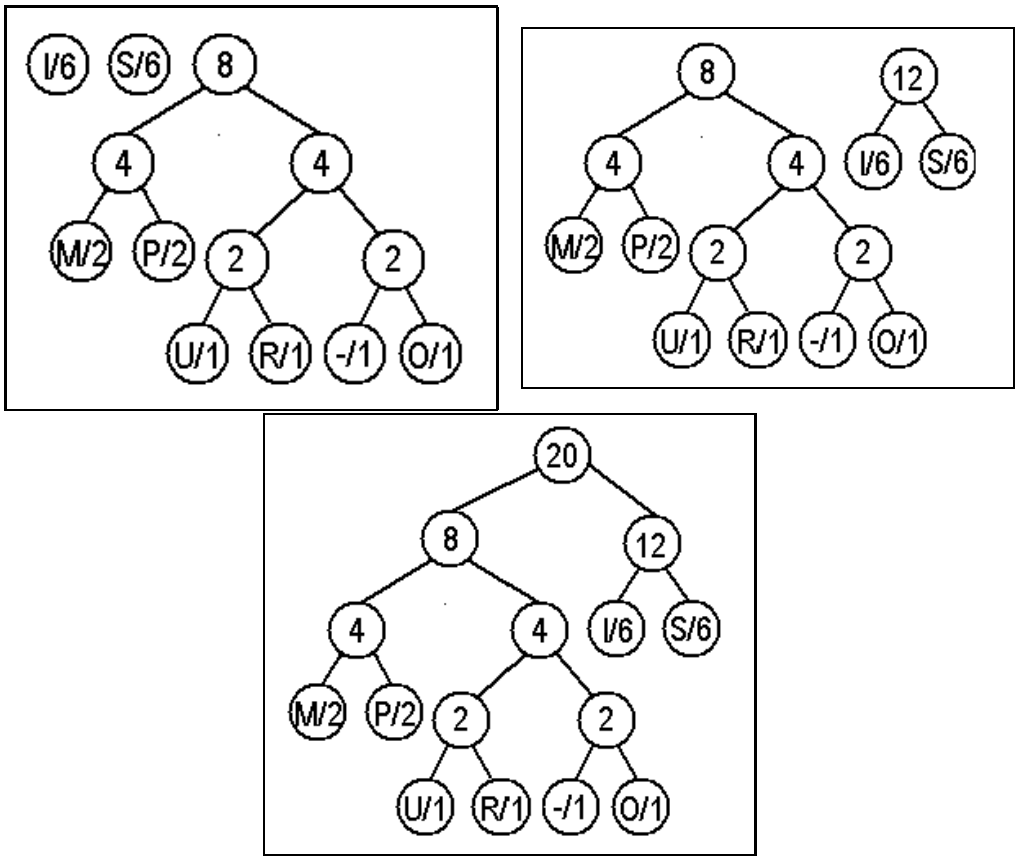


Abbildung 6.7: Die nächsten Schritte

die Häufigkeit seines Auftretens. Dann ist die erwartete Dekodierungszeit:

$$\sum_{i=0}^{26} q_i d_i$$

Diese wird minimal, wenn die mit den Häufigkeiten gewichtete Pfadlänge bis zu den Blättern minimal wird.

Wie machen wir das nun praktisch? Zunächst die Codierung.

1. Wir zählen die Häufigkeiten der Zeichen im Text und tragen diese in in ein Array freq ein. Das ist die Frequenztabelle.
2. Dann bauen wir uns einen min-Heap auf, in dem die Knoten folgende Attribute haben:
   - leaf, enthält true, wenn es sich um ein Blatt handelt.
   - char, Das Zeichen falls der Knoten ein Blatt ist.
   - freq, kumulierte Häufigkeit der Zeichen, die unter diesem Knoten „hängen“.
   - left, Verweis auf linken Nachfolger.
   - right, Verweis auf rechten Nachfolger
3. Aus diesem Baum konstruieren wir zwei Arrays,
   - code, mit den Codierungen der Zeichen,
   - len, mit den zugehörigen Längen der Codierungen
4. Diese beiden Array schreiben wir uns für den späteren Versand weg. Anschließend durchlaufen wir die zu komprimierende Datei erneut, codieren die Zeichen und schreiben sie weg.

Erhalten wir eine derart behandelte Datei und die Arrays freq, code und len, so decodieren wir wie folgt:

1. Wir bauen den Baum auf. Das geht jetzt ganz einfach: Die Bits sagen uns ob wir links (0) oder rechts (1) einfügen müssen, und die Länge wie weit wir runter müssen. In die Blätter tragen wir das Zeichen ein.
2. Wir lesen ein Code-Zeichen, klettern runter bis zum Blatt, geben das dort gefundene Zeichen aus, gehen zurück zur Wurzel, lesen das nächste Zeichen etc.

6.4 Kryptographie

In der Informatik wird heute das Geheimnisprinzip „hochgehalten“. Es gibt aber auch viele Situationen in anderen Bereichen, in denen Informationen nur ausgewählten Personen zugänglich sein sollen. Hier einige Beispiele:

- Firmengeheimnisse, die auf einem Rechner gespeichert sind, sollen beim Abhanden kommen des Rechners oder beim Eindringen von Unbefugten in den Rechner nicht lesbar sein.
- Geheime Informationen sollen übertragen werden. Für den Fall, dass die Nachricht während der Übertragung abgefangen wird, soll derjendige, der die Information erhält, nichts damit anfangen können.
- In vielen Verhandlungen geht es darum herauszubekommen, über welche Informationen der Verhandlungspartner verfügt ohne seine eigenen Informationen preiszugeben.
- ...

In den ersten beiden Fällen wird man ein Verfahren wählen, bei dem die Information verschlüsselt wird und mit einem geeigneten Schlüssel wieder entschlüsselt werden kann.

Bei den alten klassischen Verfahren verwendet man einen Schlüssel, der angibt wie die Zeichen der Nachricht durch andere ersetzt werden sollen. Dieser Schlüssel muss auf sichere Weise zum Empfänger der Nachricht gelangen, d. h. so, dass er nicht in die Hände anderer gelangen kann. Die verschlüsselte Nachricht kann irgendwie geschickt werden, es sei denn, man will auch geheim halten, dass es überhaupt eine Verbindung zwischen Sender und Empfänger gibt. Der Empfänger kann mittels des Schlüssels die Nachricht wieder in ds Original „übersetzen“.

Beispiele für Verfahren dieser Art sind

- Die Caesar-Chiffre, bei der jeder Buchstabe der Nachricht durch den drei Stellen später im Alphabet stehenden ersetzt wird.
- Die Vigenère-Verschlüsselung, bei der z. B. 26 (A. . . Z) Geheimtextalphabete nach einem vereinbarten Schema zur Verschlüsselung verwendet werden.

Für die Einzelheiten derartiger Verfahren sei auf [Sin00] und das Handbuch zu CrypTool verwiesen. verwiesen.

Unmittelbar klar sollten aber auch ohne genauere Diskussion folgende Punkte sein:

1. Die Verteilung der Schlüssel (Distanz der Verschiebung bei der Caesar-Chiffre etc.) ist eine gravierende Schwachstelle.
2. Ist die Nachricht hinreichend lang, so sind diese Verschlüssellungen durchaus in endlicher Zeit zu „knacken“. In einfachen Fällen hilft dabei schon die Häufigkeit der Buchstaben in einer Sprache (s. o. S. 143).

Da diese Verfahren auf der Seite des Senders und Empfängers den gleichen Schlüssel verwenden, nennt man sie *symmetrisch*. Diese symmetrischen Verfahren werden in ihrer klassischen Form daher nicht mehr für ernsthafte Anwendungen eingesetzt.

Praktisch eingesetzt werden für viele Zwecke asymmetrische Verfahren oder eine Kombination von symmetrischen und asymmetrischen Verfahren.

Das bekannte PGP ist eines von vielen Beispielen hierfür.

Für diese Version verweise ich für alles weitere auf CrypTool und die damit ausgelieferte Dokumentation, die Sie unter www.cryptool.org/ finden.

6.5 Historische Anmerkungen

Siehe [Sin00] und das Skript zu CrypTool.

6.6 Aufgaben

Anhang A

Mathematische Grundlagen

A.1 Übersicht

Elementare Boolesche Algebra sollte jeder beherrschen, Informatiker vielleicht deutlich mehr als „Otto Normalverbraucher“. Man muss diese Kenntnisse ja nicht bei jeder privaten Unterhaltung herausstreichen. Wenn man programmiert ist es einfach wichtig, dies zu beherschen.

In vielen Situationen benötige ich bei der Untersuchung von Algorithmen Aussagen über Folgen von natürlichen Zahlen. Die Fähigkeit, mit natürlichen Zahlen „arbeiten“ zu können, also

- rechnen können, auch mit anderen Basen als 10
- Beweise mit vollständiger Induktion führen können
- Konvergenz von Folgen untersuchen können

sind eine notwendige Voraussetzung für das Studium von Algorithmen und werden in diesem Skript vorausgesetzt. Dieses Kapitel fasst u. a. die wichtigsten Aussagen, die aus diesem Teilgebiet der Mathematik benötigt werden, zusammen. Damit soll den Lesern und Leserinnen, die sich nicht mehr an das gesamte Schulwissen erinnern, eine einfache Möglichkeit zum Nachschlagen geboten werden.

Um Rekursionsgleichungen zu lösen sind Kenntnisse über die Grundbegriffe und Lösungsstrategien und -techniken für Differenzengleichungen unter informatischen Gesichtspunkten nützlich. Ich betrachte Differenzengleichungen hier also eher unter dem Gesichtpunkt des Algorithmus im Sinn von Def. 2.4 auf Seite 31 oder der Rekursionsgleichung, als unter dem Gesichtspunkt etwa der Diskretisierung von Differentialgleichungen. Als ich anfing, diesen Abschnitt zu schreiben, verwendete ich zunächst [Rom86] als Quelle.

A.2 Lernziele

- Grundbegriffe der Mengenlehre kennen.
- Beweise mit vollständiger Induktion führen können.
- Folgen und Reihen auf Konvergenz untersuchen können.
- Wissen, was eine Differenzengleichung ist.
- Eigenschaften der Lösung(en) von Differenzengleichungen aus den Eigenschaften der Gleichung herleiten können.
- Differenzengleichungen lösen können.
- Beweisen können, dass eine gefundene Lösung einer Differenzengleichung tatsächlich eine Lösung ist.

A.3 Boolesche Algebra

Es verbietet sich aus vielen Gründen hier [Knu08b, Knu07] abzuschreiben. Ich empfehle aber dringends sich diese anzusehen, am besten, solange Sie noch frei verfügbar im Internet stehen. Dann können Sie nämlich US$ 2,56 pro Fehler verdienen und Spaß sollte die Lektüre auch machen! Sie finden in [Knu08b] auf Seite 3 eine Tabelle aller binären booleschen Funktionen von zwei Variablen. Ich stelle hier nur einige wichtige Regeln für boolesche Operatoren zusammen.

$x \wedge y$	$=$	$y \wedge x$	Kommutativgesetze
$x \vee y$	$=$	$y \vee x$	
$x \oplus y$	$=$	$y \oplus x$	
$x \wedge (y \wedge z)$	$=$	$(x \wedge y) \wedge z$	Assoziativgesetze
$x \vee (y \vee z)$	$=$	$(x \vee y) \vee z$	
$x \oplus (y \oplus z)$	$=$	$(x \oplus y) \oplus z$	
$(x \vee y) \wedge z$	$=$	$(x \wedge z) \vee (y \wedge z)$	Distributivgesetze
$(x \wedge y) \vee z$	$=$	$(x \vee z) \wedge (y \vee z)$	
$(x \oplus y) \wedge z$	$=$	$(x \wedge z) \oplus (y \wedge z)$	
$(x \wedge y) \vee x$	$=$	x	Absorptionsgesetze
$(x \vee y) \wedge x$	$=$	x	
$(x \oplus y) \oplus x$	$=$	y	
$(x \oplus y) \oplus y$	$=$	x	
$\neg(x \wedge y)$	$=$	$\neg x \vee \neg y$	De Morgansche Gesetze
$\neg(x \vee y)$	$=$	$\neg x \wedge \neg y$	

A.4 Mengen

Definition A.1 (Mengentheoretische Begriffe)

- $S = \{e_1, e_2, \ldots, e_n\}$, endliche Menge, angegeben durch Aufzählung ihrer Elemente.
- $|S|$ Kardinalität: Anzahl Element in S; der Einfachheit halber seien endliche Mengen S unterstellt.
- $S \cup T := \{x | x \in S \vee x \in T\}$, Vereinigung von S und T.
- $S \cap T := \{x | x \in S \wedge x \in T\}$, Durchschnitt von S und T.
- $S \backslash T := \{x | x \in S \wedge x \notin T\}$, Differenz von S und T.
- $S \Delta T := (S \cup T) \backslash (S \cap T)$, symmetrische Differenz von S und T.

◄

A.5 Abbildungen

Definition A.2 (Abbildung)
Seien A und B Mengen. Eine Abbildung f von A nach B ordnet jedem Element von A genau ein Element von B zu. Wir schreiben:

$$f : \begin{cases} A \longrightarrow B \\ x \mapsto f(x) \end{cases}$$

◄

Definition A.3 (Injektiv, surjektiv, bijektiv)
Sei $f : A \longrightarrow B$ eine Abbildung. f heißt

injektiv, wenn gilt:

$$x_1 \neq x_2 \Longrightarrow f(x_1) \neq f(x_2)$$

surjektiv, wenn gilt: $f(A) = B$, mit anderen Worten

$$\forall b \in B \; \exists a \in A \text{ mit } f(a) = b.$$

bijektiv, wenn f injektiv und bijektiv ist.

◀

Satz A.4 (Injektivität)
Eine Abbildung $f : A \longrightarrow B$ ist genau dann injektiv, wenn 1. oder 2. gilt:

1. $\forall b \in f(A) \exists_1 a \in A$ mit $f(a) = b$
2. $\forall x_1, x_2 \in A : f(x_1) = f(x_2) \Longrightarrow x_1 = x_2$

◀

A.6 Intervalle

$$\begin{aligned}
[a, b] &:= \{x | a \leq x \; \leq b\} \\
[a, b[&:= \{x | a \leq x \; < b\} \\
&=: [a, b) \\
]a, b] &:= \{x | a < x \; \leq b\} \\
&=: (a, b] \\
]a, b[&:= \{x | a < x \; < b\} \\
&=: (a, b)
\end{aligned}$$

A.7 Axiome

Die natürlichen Zahlen $\mathbb{N} = \{1, 2, 3, \ldots\}$ sind jedem von Kindeszeiten her geläufig. Vielleicht hat es gerade deshalb so lange gedauert, bis eine präzise mathematische Beschreibung formuliert wurde. Heute charakterisiert man de Menge der natürlichen Zahlen $\mathbb{N}$ durch die Peanoschen Axiome.

Definition A.5 (Peano Axiome)
Die Menge $\mathbb{N}$ der natürlichen Zahlen wird durch die *Peano Axiome* charakterisiert:

1. $1 \in \mathbb{N}$
2. Jede Zahl $a \in \mathbb{N}$ hat einen eindeutig bestimmten Nachfolger $a' \in \mathbb{N}$
3. Es gibt keine natürliche Zahl mit dem Nachfolger 1.
4. $a' = b' \Longrightarrow a = b$
5. Jede Menge M natürlicher Zahlen ($M \subset \mathbb{N}$), die 1 und zu jeder natürlichen Zahl a auch deren Nachfolger a' enthält, enthält alle natürlichen Zahlen. In Formeln: $((1 \in M) \wedge (a \in M \Longrightarrow a' \in M)) \Longrightarrow M = \mathbb{N}$ (Prinzip der vollständigen Induktion)

◀

Für den Nachfolger a' schreibt man unter Mathematikern $a + 1$, unter (C etc.) Programmierern $a + +$.

Insbesondere das fünfte Peanosche Axiom aus Def. A.5 wird oft benötigt, um eine Aussage $A(n)$ für alle $n \in \mathbb{N}$zu beweisen. Üblicherweise geschieht dies in der folgenden Form:

IV Induktionsverankerung oder Induktionanfang: Hier wird gezeigt, dass die Aussage für $n = 1$ wahr ist, d. h $A(1)$ ist wahr.

IA Induktionsannahme oder Induktionshypothese. Es wird angenommen, dass die Aussage für ein $n_0 \in \mathbb{N}$ wahr sei. Die Menge der $n_0 \in \mathbb{N}$, für die dies gilt, ist nicht leer, denn nach **IV** gilt dies für $n_0 = 1$. In manchen Fällen nimmt man auch an, die Aussage sei für alle $k \leq n_0$ wahr.

IS Induktionschluss: In diesem Teil wird logisch zwingend aus der Induktionsannahme (**IA**) und der Induktionsverankerung (**IV**) hergeleitet, dass die Aussage auch für $n_0 + 1$ gilt, d. h. dass auch $A(n_0 + 1)$ wahr ist.

Die Induktionsverankerung ist dabei entscheidend: Sie stellt sicher, dass nicht etwas für die leere Menge ($\emptyset$) bewiesen wird. Über die leere Menge ist jede Aussage wahr!

A.8 Beweise mit vollständiger Induktion

Hier nun einige Beispiele, die illustrieren, wie das geht bzw. was schief gehen kann.

Einer Anekdote nach fand Gauss im Schulunterricht diese Formel, als der Lehrer die Schüler aufforderte,die Zahlen von 1 bis 100 zu addieren, um eine Zeit lang seine Ruhe zu haben.

Satz A.6 (Gauss'sche Formel)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \ \forall \ n \in \mathbb{N} \tag{A.1}$$

Der Beweis mit vollständiger Induktion ist einfach (siehe Beispiel A.7). ◀

Beispiel A.7 (vollständige Induktion)
Als ganz einfaches Beispiel eines Beweises mit vollständiger Induktion sei hier der Beweis von Satz A.6 vorgeführt.

IV Ich zeige zunächst, dass die Formel A.1 für $n = 1$ wahr ist. Dazu sind nur die linke und die rechte Seite auszurechnen:

$$\sum_{i=1}^{1} i = 1 \tag{A.2}$$

$$\frac{1(1+1)}{2} = \frac{2}{2} = 1 \tag{A.3}$$

IA Die Formel A.1 sei für $n = n_0$ wahr, ausgeschrieben

$$\sum_{i=1}^{n_0} i = 1 + 2 + \cdots + n_0 = \frac{n_0(n_0+1)}{2} \tag{A.4}$$

Nun kommt es darauf an, aus dem was wir jetzt haben, unter Verwendung von Wissen, das keiner mit Erfolg anzweifeln kann, die Formel auch für $n_0 + 1$ herzuleiten. In einem so einfachen Fall wie diesem, führt die einfachste Strategie oft zum Erfolg: Man schreibt einfach

die linke Seite hin und rechnet so lange, bis man das gewünsche Ergebnis hat:

$$\sum_{i=1}^{n_0+1} i = \sum_{i=1}^{n_0} i + (n_0+1) \tag{A.5}$$

$$= \frac{n_0(n_0+1)}{2} + (n_0+1)(IA) \tag{A.6}$$

$$= \frac{n_0(n_0+1)+2(n_0+1)}{2} \tag{A.7}$$

$$= \frac{(n_0+1)n_0+(n_0+1)2}{2} \tag{A.8}$$

$$= \frac{(n_0+1)(n_0+2)}{2} \tag{A.9}$$

In diesem Fall ist es mir gelungen einfach „von links nach rechts“ das gewünschte Ergebnis auszurechnen. Das klappt nicht immer auf Anhieb. Komme ich in's Stocken, so hat es sich für mich bewährt auch auf der rechten Seite loszurechnen. Vielleicht klappt es ja dann sich in der Mitte zu treffen.

In manchen Lösungsvorschlägen habe ich etwa folgende Argumentation gelesen:

$$\sum_{i=1}^{n_0+1} i = \frac{(n_0+1)(n_0+2)}{2} \Longleftrightarrow \tag{A.10}$$

$$\sum_{i=1}^{n_0} i + (n_0+1) = \frac{n_0(n_0+1)}{2} + (n_0+1) \overset{(IA)}{\Longleftrightarrow} \tag{A.11}$$

$$\sum_{i=1}^{n_0} i + (n_0+1) = \sum_{i=1}^{n_0} i + (n_0+1) \Longleftrightarrow \tag{A.12}$$

$$(n_0+1) = (n_0+1) \tag{A.13}$$

$$\tag{A.14}$$

Die letzte Aussage ist offenbar wahr und damit ist die Behauptung bewiesen. Das ist nicht falsch, aber erfordert nicht weniger, vielleicht sogar mehr Schreibarbeit (Denken Sie an Zeit und Fehler in einer Klausur), erfordert ggf. eine Begründung für die Äquivalenzumformung und ist nach meinem Eindruck nicht nur unter Mathematikern unüblich. ◀

Definition A.8 (Logarithmen)

- $\ln x : y = \ln x \Leftrightarrow e^y = x$ Natürlicher Logarithmus (Basis e = 2.71828...)
- $ld\ x : y = ld\ x \Leftrightarrow 2^y = x$ binärer oder dualer Logarithmus (Basis 2)
- $log\ x : y = log\ x \Leftrightarrow 10^y = x$ Dekadischer Logarithmus (Basis 10)

◀

Rechenregeln für Logarithmen

$$log_a x = log_a b \cdot log_b x \tag{A.15}$$

$$log(x \cdot y) = log(x) + log(y) \tag{A.16}$$

$$log(x^y) = y \cdot log(x) \tag{A.17}$$

Die folgende Definition fasst die wichtigsten Symbole zusammen, die für den Umgang mit Summen und Produkten benötigt werden.

Definition A.9 (Summen und Produkte)

1.

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$$

2.
$$\prod_{i=1}^{n} a_i = a_1 \times a_2 \times \cdots \times a_n$$

3.
$$c^n := \prod_{i=1}^{n} c = c \times c \times \cdots \times c$$

4.
$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n$$

5. Die leere Summe ist 0, etwa:
$$\sum_{i=1}^{0} a_i = 0$$

6. Das leere Produkt ist 1, etwa:
$$\prod_{i=1}^{0} a_i = 1$$

◀

Satz A.10 (Rechenregeln für Summen und Produkte)

1. Distributivgesetz
$$(\sum_{R(i)} a_i)(\sum_{S(j)} b_j) = \sum_{R(i)} (\sum_{S(j)} a_i b_j)$$

2. Wechsel der Variablen
$$\sum_{R(i)} a_i = \sum_{R(j)} a_j = \sum_{R(p(j))} a_{p(j)}$$

3. Summationsreihenfolge
$$\sum_{R(i)} \sum_{S(j)} a_{ij} = \sum_{S(j)} \sum_{R(i)} a_{ij}$$

◀

Definition A.11 (Binomialkoeffizienten)
Der Binomialkoeffizient $\binom{n}{r}$ *n über r* ist definiert als

$$\binom{n}{r} := \frac{n!}{(n-r)!r!}$$

Wegen $0! = 1$ ist $\binom{n}{0} = 1$ ◀

Satz A.12 (Formeln für Binomialkoeffizienten)

$$
\begin{aligned}
\binom{n}{r} &= 0 \text{ für } r > n; \\
\binom{n}{n} &= 1; \\
\binom{n}{r} &= \binom{n}{n-r}; \\
\binom{n+1}{r+1} &= \binom{n}{r} + \binom{n-1}{r} + \binom{n-2}{r} + \cdots + \binom{r}{r}; \\
\binom{n+1}{r} &= \binom{n}{r} + \binom{n}{r-1} = \binom{n}{r} \cdot \frac{n+1}{n-r+1} \\
\binom{m+n}{r} &= \binom{m}{r}\binom{n}{0} + \binom{m}{r-1}\binom{n}{1} + \cdots + \binom{m}{1}\binom{n}{r-1} + \binom{m}{0}\binom{n}{r}; \\
(a \pm b)^n &= \binom{n}{0}a^n \pm \binom{n}{1}a^{n-1}b + \ldots (\pm)^n \binom{n}{n}b^n
\end{aligned}
$$

◀

A.9 Endliche Reihen

Es gibt eine Fülle nützlicher Formeln für endliche Reihen. Der Inhalt dieses Abschnitts enthält einige Dinge, die ich in den letzten Jahren aus irgend einem Grund brauchen konnte.

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{A.18}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{A.19}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4} \tag{A.20}$$

$$\sum_{k=1}^{n} 2k-1 = n^2 \tag{A.21}$$

$$\sum_{k=1}^{n} 2k = n(n+1) \tag{A.22}$$

$$\sum_{k=1}^{n} (2k-1)^2 = \frac{n(4n^2-1)}{3} \tag{A.23}$$

$$\sum_{k=1}^{n} (2k-1)^3 = n^2(2n^2-1) \tag{A.24}$$

$$\sum_{k=1}^{n} a^{k-1} = \frac{1-a^n}{1-a} \tag{A.25}$$

$$\sum_{k=1}^{n} \frac{1}{k} = C + ln\ n + \frac{1}{2n} - \frac{a_2}{n(n+1)} - \frac{a_3}{n(n+1)(n+2)} - \cdots \tag{A.26}$$

$$C = 0,5772156649 \ldots \text{Eulersche Konstante} \tag{A.27}$$

$$a_k = \frac{1}{k} \int_o^1 x(1-x)(2-x)\cdots k-1-x)dx \tag{A.28}$$

$$\tag{A.29}$$

A.10 Grenzwerte

Definition A.13 (Konvergenz)

1. Eine Folge $(a_i)_{i=1,2,\ldots}$ konvergiert genau dann gegen einen Grenzwert a, wenn gilt $\forall\, \epsilon > 0\ \exists\, n_0$ mit $|a - a_i| < \epsilon\ \forall\ i \geq n_0$.
2. Eine unendliche Summe $\sum_{i=1}^{\infty} a_i$ konvergiert genau dann gegen einen Wert S, wenn die Folge der Teilsummen $s_i := \sum_{j=1}^{i} a_j$ konvergiert.

◀

A.11 Beträge, Normen und Metriken

Definition A.14 (Metrik)
Sei M eine Menge. Eine Abbildung

$$d : \left\{ \begin{array}{lll} M \times M & \longrightarrow & \mathbb{R}_0^+ \\ (x,y) & \mapsto & d(x,y) \end{array} \right.$$

heißt Metrik, wenn gilt:

$$\begin{aligned} d(x,x) &= 0\ \forall\ x \in M \\ d(x,y) &= d(y,x) \neq 0\ \forall\ x,y \in M, x \neq y (Symmetrie) \\ d(x,z) &\leq d(x,y) + d(y,z)\ \forall\ x,y,z \in M (Dreiecksungleichung) \end{aligned}$$

◀

Der absolute Betrag || auf den reellen Zahlen $\mathbb{R}$ ist eine Metrik, insbesondere gilt:

Satz A.15 (Dreiecksungleichung)

$$|x| - |y| \leq |x - y|$$

◀

A.12 Ganzzahlige Operationen

Definition A.16 (Einige ganzzahlige Funktionen)

- $\lfloor x \rfloor$ Untere Gaussklammer (floor): $max\{n \in \mathbb{Z} | n \leq x\}$
- $\lceil x \rceil$ Obere Gaussklammer (ceiling): $min\{n \in \mathbb{Z} | n \geq x\}$
- x mod y: x modulo y, der „Rest“: $x - y\lfloor \frac{x}{y} \rfloor$
- $[x]$ gebrochener Anteil $xmod1$ (Achtung: In der Mathematik gibt es auch die Bedeutung, [ ] = floor)
- gcd(x,y) „greatest common divisor“, größter gemeinsamer Teiler, deutsch auch ggt(x,y) abgekürzt.
- lcm(x,y) „least common multiple“, kleinstes gemeinsames Vielfaches deutsch auch kgv(x,y) abgekürzt.
- $a \perp b$ „teilerfremd“ (relatively prime): gcd(x,y) = 1
- $a|b$ a ist Teiler von b.

◀

Satz A.17 (Rechenregeln für ganzzahlige Funktionen)

1. $\lceil x \rceil = \lfloor x \rfloor \Longleftrightarrow x \in \mathbb{Z}$
2. $\lceil x \rceil = \lfloor x \rfloor + 1 \Longleftrightarrow x \in \mathbb{R} \backslash \mathbb{Z}$
3. $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
4. Sei $n \in \mathbb{N}$ und $x \in \mathbb{R}$. Dann gilt:
   - 4.1. $\lfloor x \rfloor < n \Leftrightarrow x < n$
   - 4.2. $n \leq \lfloor x \rfloor \Leftrightarrow n \leq x$
   - 4.3. $\lceil x \rceil \leq n \Leftrightarrow x \leq n$
   - 4.4. $n < \lfloor x \rfloor \Leftrightarrow n < x$
   - 4.5. $\lfloor x \rfloor = n \Leftrightarrow x - a < n \leq x \Leftrightarrow n - 1 \leq x < n + 1$
   - 4.6. $\lceil x \rceil = n \Leftrightarrow x \leq n < x + 1 \Leftrightarrow n - 1 < x \leq n$

   ([Knu97a], S. 41)
5. $\lfloor -x \rfloor = -\lceil x \rceil$ ([Knu97a], S. 41)
6. $$\sum_{k=1}^{n} \lfloor \frac{k}{2} \rfloor = \lfloor \frac{n^2}{4} \rfloor$$

   ([Knu97a], S. 43)
7. $$\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil = \lceil \frac{n(n+2)}{4} \rceil$$

   ([Knu97a], S. 43)

◀

A.13 Laufzeituntersuchung

Wir betrachten zunächst die geschachtelten Schleifen aus Bsp. 2.3:

```
a[1] = 0;
  for (i = 2; i <= N; i++)
     a[i] = 1;
  for (i = 2; i <= sqrt(N); i++)
         for (j = 2; j <= sqrt(N); j++)
           a[i*j] = 0;
```

Abschätzung der Laufzeit:

Code	Abschätzung der Laufzeit
for (i = 2; i ¡= N; i++)	$N-1$
a[i] = 1;	1
for (i = 2; i ≤sqrt(N); i++)	$\sqrt{N}-1$
for (j = 2; j≤ sqrt(N); j++)	$\sqrt{N}-1$
a[i*j] = 0;	1

Konsequenzen

Loop 1 $(N-1)*1 = N-1$

Loop 2 $(\sqrt{N}-1)*(\sqrt{N}-1)*1 \leq N-2\sqrt{N}+1$

Satz A.18 ($\mathcal{O}$ und Grenzwert)
Seien f und g positive reellwertige Funktionen, so gilt

$$\text{Der Grenzwert } \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \text{ existiert } \iff f = \mathcal{O}(g).$$

◄

A.14 Differenzengleichungen

Definition A.19 (Erste Differenz)
Seien G, H *Gruppe*n, $D \subset G$ und y eine *Abbildung*

$$y : D \longrightarrow H$$

Dann ist für $x \in D, h \in G, x+h \in D$ die *erste Differenz* von y definiert als

$$\Delta_h y(x) := y(x+h) - y(x).$$

Δ_h heißt Differenzenoperator. ◄

Definition A.20 (Zweite Differenz)
Sei $\Delta_h y(x)$ die erste Differenz von y nach Def. A.19. Dann ist die zweite Differenz $\Delta_h^2 y(x)$von y definiert als erste Differenz von $\Delta_h y(x)$, d. h.

$$\Delta_h^2 y(x) := \Delta_h(\Delta_h y(x))$$

Analog werden Differenzen höherer Ordnung definiert. ◄

Definition A.21 (Gewöhnliche Differenzengleichung)
Eine *gewöhnliche Differenzengleichung* ist eine Beziehung zwischen einer unabhängigen Variablen x, einer Funktion $y(x)$ und einer oder mehrerer ihrer Differenzen $\Delta_h^n y(x)$. Dabei ist h eine willkürlich gewählte Konstante. ◄

Definition A.22 (Grad einer Differenzengleichung)
Der *Grad einer Differenzengleichung* ist der Exponent der höchsten Potenz, in der die höchste Differenz auftritt. ◄

Ist der Grad 1 so spricht man von einer linearen Differenzengleichung.

Definition A.23 (Ordnung einer Differenzengleichung)
Die *Ordnung einer Differenzengleichung* ist n, wenn die Differenz zwischem dem größten und dem kleinsten wirklich vorkommenden Argument gleich nh ist. ◄

In der Regel betrachtet man h = 1 und so werde ich hier auch verfahren. Die Fälle, in denen andere Abstände betrachtet werden müssen, kann man durch Substitution behandeln.

Definition A.24 (Lineare Differenzengleichung)
Eine Differenzengleichung heißt linear, wenn sie in der Form

$$f_n(k)y_{k+n} + f_{n-1}(k)y_{k+n} + \cdots + f_1(k)y_{k+1} + f_0(k)y_k - g(k) = 0$$

geschrieben werden kann. Ist $g \equiv 0$, so spricht man von einer linearen homogenen Differenzengleichung. ◄

Für lineare Differenzengleichungen erster und zweiter Ordnung gibt es Lösungsstrategien, die denen für lineare Gleichungssystem bzw. linearen Differentialgleichungen ähneln (nun ja, zumindest entfernt).

Satz A.25 (Existenz- und Eindeutigsatz)
Die lineare Differenzengleichung n-ter Ordnung

$$f_n(k)y_{k+n} + f_{n-1}(k)y_{k+n} + \cdots + f_1(k)y_{k+1} + f_0(k)y_k - g(k) = 0 \tag{A.30}$$

über einer Menge S aufeinanderfolgender ganzzahliger Werte von k hat genau eine Lösung y, deren Funktionswerte für n aufeinanderfolgende k-Werte willkürlich vorgegeben sind. ◄

Lineare Differenzengleichungen 1. Ordnung

Satz A.26 (Lineare Differenzengleichung 1. Ordnung)
Die allgemeine Lösung der linearen homogenen Differenzengleichung

$$y_{k+1} = a_k \cdot y_k, y_0 = c; \tag{A.31}$$

ist

$$y_k = c \prod_{i=0}^{k-1} a_i \tag{A.32}$$

Die Lösung der inhomogenen Gleichung

$$y_{k+1} = a_k \cdot y_k + b_k, y_0 = c \tag{A.33}$$

ist

$$y_k = c \prod_{i=0}^{k-1} a_i + \sum_{m=0}^{k-1} \frac{b_m}{\prod_{j=0}^{m} a_j} \prod_{i=0}^{k-1} a_i \tag{A.34}$$

◄

Bemerkung A.27 (Konstante Koeffizienten)
Für konstante Koeefizienten vereinfachen sich Lösungen gemäß Satz A.25 wie folgt: Ist $a = a_i, b = b_i \ \forall i$ so ist die Lösung der homogenen Gleichung:

$$y_k = c \cdot a^k \tag{A.35}$$

und entsprechend die der inhomogenen:

$$y_k = ca^k + b\frac{1-a^k}{1-a} \tag{A.36}$$

◄

Beispiel A.28 (Türme von Hanoi)
Als Beispiel für die Anwendung von Satz A.26 lösen wir die Rekursionsgleichung, die z. B. bei der Anzahl der Züge beim Spiel *Türme von Hanoi* auftritt:

$$\begin{aligned} T_0 &= 1 \\ T_{n+1} &= 2 \cdot T_n + 1. \end{aligned}$$

Hier ist $a_k = 2, k = 0, 1, \ldots, c = 1$. Die Lösung ist nach A.27:

$$\begin{aligned} T_n &= 2^n + \frac{1-2^n}{1-2}) \\ &= 2^n - 1 + 2^n \\ &= 2^{n+1} - 1 \end{aligned}$$

Für die übliche Anwendung müssen wir nur noch bei 1 statt bei 0 beginnen. Dann ist

$$T_n = 2^n - 1$$

◄

Satz A.29 (Teile und herrsche)
Die Aufwandsfunktion von divide and conquere Algorithmen genügt der Differenzengleichung

$$T(n) = aT(\frac{n}{c}) + bn^m.$$

Für die Lösung gilt:

$$T(n) = \begin{cases} \Theta(n^m), & \forall\ a < c^m \\ \Theta(n^m log\ n), & \forall\ a = c^m \\ \Theta(n^{loc_c a}), & \forall\ a > c^m \end{cases}$$

[CFR05] ◄

Satz A.30 (Hôspitalsche Regel)
Seien $f, g : \mathbb{R} \longrightarrow \mathbb{R}$ Funktionen mit

$$\lim_{x \to a} f(x) = \lim_{x \to a} g(x) = 0$$

und gilt $f'(a) = \ldots = f^{(n-1)} = 0, g'(a) = \ldots g^{(n-1)} = 0, g^{(n)}(a) \neq 0$, dann ist

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \frac{f^{(n)}(a)}{g^{(n)}(a)}$$

sofern der Grenzwert existiert [Mes71]. ◄

A.15 Wahrscheinlichkeitsrechung

Definition A.31 (Zufallsvariable, Ereignis)
Eine diskrete Zufallsvariable X ist eine Abbildung einer endlichen oder abzählbarunendlichen Menge S in die rellen Zahlen $\mathbb{R}$, also:

$$X : S \longrightarrow \mathbb{R}$$

Das Ereignis $X = x$ ist definiert als $\{s \in S : X(s) = x\}$ ◀

Definition A.32 (Erwartungswert)
Der Erwartungswert einer einer diskreten Zufallsvariablen X ist

$$E(X) = \sum_x x \cdot Pr\{X = x\}$$

sofern diese Summe existiert, d. h. endlich ist oder absolut konvergiert. Pr{X=x} ist dabei die Wahrscheinlichkeit von $X = x$. ◀

A.16 Aufgaben

1. ([01]) Wie rechnet man einen Logarithmus in einen anderen (d. h. mit einer anderen Basis) um?
2. ([05])Lösen Sie die folgenden Differenzengleichungen
   - 2.1. $y_{n+1} - (n+1)y_n = 0, y_0 = c.$
   - 2.2. $y_{n+1} - 3^n y_n = 0, y_0 = c.$
   - 2.3. $y_{n+1} - e^{2n} y_n = 0, y_0 = c.$
   - 2.4. $y_{n+1} - \frac{n}{n+1} y_n = 0, n \geq y_1 = c.$

A.17 Das griechische Alphabet

α	alpha	β	beta	γ, Γ	gamma	δ, Δ	delta
ϵ, ε	epsilon	ζ	zeta	η	eta	$\theta, \vartheta, \Theta$	theta
ι	jota	κ	kappa	λ, Λ	lambda	μ	my
ν	ny	ξ, Ξ	xi	o	omikron	π, Π	pi
ρ, ϱ	rho	$\sigma, \varsigma, \Sigma$	sigma	τ	tau	υ, Υ	ypsilon
ϕ, φ, Φ	phi	χ	chi	ψ, Ψ	psi	ω, Ω	omega

A.18 Das deutsche Fraktur-Alphabet

$\mathfrak{a}, \mathfrak{A}$	a, A	$\mathfrak{b}, \mathfrak{B}$	beta	$\mathfrak{c}, \mathfrak{C}$	c, C	$\mathfrak{d}, \mathfrak{D}$	d, D	$\mathfrak{e}, \mathfrak{E}$	e, E
$\mathfrak{f}, \mathfrak{F}$	f, F	$\mathfrak{g}, \mathfrak{G}$	g, G	$\mathfrak{h}, \mathfrak{H}$	h, H	$\mathfrak{i}, \mathfrak{I}$	i, I	$\mathfrak{j}, \mathfrak{J}$	j, J
$\mathfrak{k}, \mathfrak{K}$	k, K	$\mathfrak{l}, \mathfrak{L}$	l, L	$\mathfrak{m}, \mathfrak{M}$	m, M	$\mathfrak{n}, \mathfrak{N}$	n, N	$\mathfrak{o}, \mathfrak{O}$	o, O
$\mathfrak{p}, \mathfrak{P}$	p, P	$\mathfrak{q}, \mathfrak{Q}$	q, Q	$\mathfrak{r}, \mathfrak{R}$	r, R	$\mathfrak{s}, \mathfrak{S}$	s, S	$\mathfrak{t}, \mathfrak{T}$	t, T
$\mathfrak{u}, \mathfrak{U}$	u, U	$\mathfrak{v}, \mathfrak{V}$	v, V	$\mathfrak{w}, \mathfrak{W}$	w, W	$\mathfrak{x}, \mathfrak{X}$	x, X	$\mathfrak{y}, \mathfrak{Y}$	y, Y
$\mathfrak{z}, \mathfrak{Z}$	z, Z								

Anhang B

Übungsaufgaben

1. Welche wichtigen Unterschiede gibt es zwischen $\mathbb{Z}$ und dem Datentyp `int` in einer Programmiersprache wie C++ oder Java?
2. Schreiben Sie die Signaturen für folgende Operationen des Typs int auf: $++$ $*=$ $\leq$ $\%$
3. Ermitteln Sie die Aufwandsfunktion für die Operationen `pop` und `push` eines Stacks bei der Implementierung als verkettete Liste!
4. Finden Sie bitte Antworten auf folgende Fragen:
   4.1. Wofür steht die Abkürzung OCL?
   4.2. Von welcher größeren Spezifikation ist OCL ein Teil oder welche Spezifikation wird durch OCL ergänzt?
   4.3. Von wem wurde OCL entwickelt?
   4.4. Für welchen Zweck ist OCL entwickelt worden?
   4.5. Geben Sie ein Beispiel für OCL an!
5. Wie werden Vor- und Nachbedingungen in den folgenden Sprachen geschrieben?
   5.1. UML
   5.2. C++
   5.3. Java

   Finden Sie heraus, wie dies in anderen Sprachen gemacht wird!
6. Beschreiben Sie die Operationen push, pop, top und isEmpty eines Stacks durch Angaben von Signatur, Vor- und Nachbedingung!
7. Beschreiben Sie die Operationen front, enqueue, dequeue, isEmpty einer Queue durch Angaben von Signatur, Vor- und Nachbedingung!
8. Schreiben Sie Beispiel 1.17 so um, dass die Vorbedingung entfallen kann!
9. Es sei ein ADT X gegeben (denken Sie an Liste, Array, Tablespace ...). Dessen Implementierung sei so, dass eine Vergrößerung einen fixen Aufwand f erfordert und für jedes Element, für das Platz geschaffen wird, ein variabler Aufwand von v anfällt. Untersuchen Sie die folgenden Wachstumstrategien auf ihre Effizient:

9.1. Minimaler Speicherplatz: Der ADT wird leer angelegt und immer, wenn ein Element eingefügt wird, um (den Platz für) ein Element vergrößert.

9.2. Prozentuale Vergrößerung: Ist der Platz gefüllt, so wird der ADT (den Platz für) p% mehr Elemente als vorher vergrößert.

10. Beweisen Sie, dass die angegebenen Aufwandsfunktionen (Θ) für die Operationen in den verschiedenen Implementationen einer Liste (Implementationen 0–3) korrekt sind!

11. Zeigen und beweisen Sie, dass Stack- und Queue-Operationen mit konstantem Aufwand implementiert werden können!

12. Beweisen Sie Satz 2.21!
$$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$$

13. Beantworten Sie die Fragen aus Aufgabe 2.17 und beweisen Sie Ihre Behauptungen:

13.1. $n^2 = \mathcal{O}(n^3)$

13.2. $n^3 = \mathcal{O}(n^2)$

13.3. $2^{n-1} = \mathcal{O}(n^2)$

13.4. $(n+1)! = \mathcal{O}(n!)$

13.5. $\sqrt{n} = \mathcal{O}(\ln n)$

13.6. $28 = \mathcal{O}(1)$

13.7. $12 \cdot n + 5 = \mathcal{O}(n \ln n)$

13.8. $n \ln n = \mathcal{O}(n^2)$

14. Sehr oft wird eine Operation swap eingesetzt. Diese benötigt drei Zuweisungen, um die Vertauschung der Elemente vorzunehmen. Wie und mit wievielen Vertauschungen überführen Sie:

14.1. (a,b,c) in (c, b, a)?

14.2. (a,b,c,d) in (b,c,d,a)?

15. Wie sieht eine Gerade $y = a \cdot x + b$ in doppelt logarithmischer Darstellung aus?

16. Wie sieht die Wurzelfunktion $y = \sqrt{x}$ in doppelt logarithmischer Darstellung aus?

17. Wie sieht die Exponentialfunktion $y = e^x$ in doppelt logarithmischer Darstellung aus?

18. Wie sieht die Funktion $y = \frac{1}{x}$ in doppelt logarithmischer Darstellung aus?

19. Analog die Darstellung in lin-log, log-lin. Welche der drei Darstellungen sind für welche Zwecke sinnvoll?

20. Beweisen Sie die folgende Aussage!
$$\sum_{k=1}^{n} \lfloor \frac{k}{2} \rfloor = \lfloor \frac{n^2}{4} \rfloor$$

21. Beweisen Sie bitte die folgende Aussage!
$$\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil = \lceil \frac{n \cdot (n+2)}{4} \rceil$$

22. Beweisen Sie bitte folgende Aussage!

$$\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2$$

23. Der ADT *deque* ist eine Liste, bei der Einfügungen und Entfernungen am Anfang oder Ende der Liste vorgenommen werden, ebenso wie viele der Zugriffe. Die fundamentalen Operationen fügen am Anfang bzw. Ende ein und entnehmen entsprechend: insertFront, insertRear, deleteFront, deleteRear. Ferner gibt es die lesenden Operationen getFront, getRear. Geben Sie an, wie dieser ADT mit einem Array und wie er mit einer Liste implementiert werden kann. (Vorzugsweise Java, C++, andere Sprachen können akzeptabel sein, im Zweifel: fragen!).

24. 24.1. Beschreiben Sie den Begriff Algorithmus und erklären Sie die vier allgemeinen Anforderungen an einen solchen Algorithmus.

    24.2. Erklären Sie folgende vier Eigenschaften eines Algorithmus: *terminiert, determiniert, deterministisch und nicht deterministisch.*

25. 25.1. Beschreiben Sie das Prinzip bzw. die Grundidee der Datenabstraktion.

    25.2. Welche vier Typen von Funktionen benötigt man im Allgemeinen? (unabhängig von der konkreten Datenstruktur als „Schnittstelle“) Geben Sie für einen Stapel jeweils ein erklärendes kleines Beispiel an. Es genügt der Name einer Funktion/Methode und eine kurze Erklärung, was sie macht.

    25.3. Definieren Sie durch Angabe der formalen Beschreibung einen Datentyp `Stack`, der einen Stapel nach dem Last-In-/First-Out-Prinzip (LiFo) modelliert. Geben Sie dazu die Wertebereiche (Typen) und die Operationen an. Bei den Operationen ist die allgemeine/formale Typisierung anzugeben (z.B. durch Kreuzprodukt) sowie die Pre- und Postcondition.

    Definieren Sie Operationen zum Ablegen eines Elementes (`push`), zum Entfernen des ersten Elementes (`pop`), zum Abfragen des ersten Elementes (`top`) sowie zur Bestimmung der Höhe des Stack (`high`).

26. In der Vorlesung wurden sechs Möglichkeiten genannt, wie man auf ein komplexes Problem reagieren kann bzw. das Problem handhabbar machen kann. Nennen und erklären Sie diese evtl. auch mit Hilfe eines Beispieles.

27. Die Addition kann durch Verwendung der Nachfolger- (`succ`) und Vorgänger-(`pred`) Funktionen realisiert werden. Im Falle der Addition sind das die Funktionen `+1` und `-1`.

    Schreiben Sie in einer (Pseudo-)Programmiersprache Ihrer Wahl **zwei Versionen** der Addition: eine Version, die einen **rekursiven Ablauf** beschreibt, und eine Version, die einen **iterativen Ablauf** beschreibt. Beide Versionen sollen jeweils nur genau **zwei ganze Zahlen** addieren können und dies ausschließlich mittels der beiden genannten Funktionen!

28. 28.1. Implementieren Sie folgendes Verfahren in z.B. Java. Dieses Verfahren zur Berechnung der Zahl π ist aus dem Jahr 1976 und stammt von den Herren Salamin und Brent.

$$a_m := \frac{a_{m-1} + b_{m-1}}{2} \text{ mit } a_0 := 1$$

$$b_j := \sqrt[2]{a_{j-1} * b_{j-1}} \text{ mit } b_0 := \sqrt[2]{0.5}$$

$$\pi_n := \frac{(a_n + b_n)^2}{(1 - \sum_{k=0}^{n} [2^k * (a_k - b_k)^2])}$$

    28.2. Zeigen Sie den Ablauf der rekursiven Aufrufe für π_2 auf. Eine Berechnung ist dabei nicht durchzuführen! Wie oft werden a_n und b_n dabei jeweils aufgerufen?

29. Gegeben sei nachfolgende Funktion:

$$f(n) = \begin{cases} 1 & \text{falls } \forall x \in Z : x \leq 1 \\ f(n-1) + f(n-2) & \text{falls } \forall x \in Z : x > 1 \end{cases}$$

    29.1. Implementieren Sie diese Funktion direkt, also ohne Optimierungsüberlegungen,in Java.

    29.2. Wie groß ist die Anzahl der Additionen $A(n)$ die beim Aufruf von $f(n)$ ausgeführt werden ? Berechnen Sie dazu zuerst die Anzahl der Additionen für alle $n \in \{1, 2, 3, 4, 5, 6, 7\}$. Versuchen Sie dann eine allgemeine Formel aufzustellen.

    29.3. Wie kann man die Funktion f **rekursiv** so implementieren, dass sie im Zeitaufwand linear läuft, also $O(n)$ besitzt ? Transformieren Sie dazu ggf. Laufzeitaufwand der naiven Lösung in Speicherplatzaufwand, jedoch mit maximal $O(n)$ Speicherplatzaufwand! Geben Sie dazu ein Java-Programm an und begründen kurz, warum der Aufwand jeweils (Laufzeit-/Speicherplatz) linear ist.

30. Beweisen Sie bitte die folgende Aussage!

$$\sum_{k=1}^{n}(2k-1) = n^2$$

31. Beweisen Sie bitte die folgende Aussage!

$$\sum_{k=1}^{n} 2k = n \cdot (n+1)$$

32. Geben Sie die Inorder-, Preorder- und Postorder-Reihenfolgen des Baums in Abb. 4.5 an!

Definition B.1 (Rot-Schwarz-Baum)

33. Ein *Rot-Schwarz-Baum* ist ein sortierter Binärbaum mit folgenden Eigenschaften: Außer den internen Knoten des Baums gibt es externe Knoten, die an die Stelle fehlender Söhne treten und die Rolle von Nachfolgern für Blätter spielen.

    33.1. Jeder Knoten ist entweder rot oder schwarz.

    33.2. Jedes externe Blatt ist schwarz.

    33.3. Wenn ein Knoten rot ist, so sind seine beiden Söhne schwarz.

    33.4. Alle Pfade von einem Knoten zu darunterliegenden Blättern enthalten dieselbe Anzahl schwarzer Knoten.

◀

    33.1. Zu welcher Implementierung raten Sie für die „externen“ Knoten?

    33.2. Angenommen, die Wurzel eines Rot-Schwarz-Baumes sei rot. Bleibt es dann ein Rot-Schwarz-Baum, wenn wir sie schwarz färben?

    33.3. Beweisen Sie: Die Höhe eines Rot-Schwarz-Baumes ist nicht größer als $2 \cdot log_2(n+1)$!

    33.4. Der Pfad von der Wurzel zu einem Blatt ist nie mehr als doppelt so lang wie der kürzeste Weg dieser Art!

34. Diese Frage hat mit dem Thema zu tun, wenn auch nicht mit Informatik. Welche botanische Neuschöpfung zeigt die Rückseite der Deutschen 1, 2 und 5 Cent Münzen? Für diejenigen, die so kleine Münzen nicht im Portemonaie oder der Tasche haben, zeigt Abb. B.1 ein Beispiel.

Abbildung B.1: Rückseite deutscher Cent-Münzen

35. Nehmen Sie an, wir haben einen sortierten Binärbaum mit den Zahlen zwischen 1 und 1000 als Schlüssel und suchen darin die Zahl 363. Welche der Folgen können nicht die Folge der besuchten Knoten sein? Begründen Sie Ihre Antwort.

    35.1. 2, 252, 401, 398, 330, 344, 397, 363.

    35.2. 924, 220, 911, 244, 898, 258, 362, 363.

    35.3. 925, 202, 911, 240, 912, 245, 363.

    35.4. 2, 399, 387, 219, 266, 382, 381, 278, 363.

    35.5. 935, 278, 247, 621, 299, 392, 358, 363.

36. Zeigen Sie, wie man den Wert eines Polynoms vom Grad n mit n Multiplikationen und n Additionen berechnen kann!.

37. Gegeben die Zahlen von 1 bis 15 (beide einschließlich). Geben Sie eine Reihenfolge dieser Zahlen an, bei der der durch Einfügen dieser Zahlenfolge entstehende sortierte Binärbaum perfekt balanciert ist und zeichnen Sie den Baum auf.

38. Gegeben die Zahlen von 1 bis 15 (beide einschließlich). Geben Sie eine Reihenfolge dieser Zahlen an, bei der der durch Einfügen dieser Zahlenfolge entstehende sortierte Binärbaum zu einer Liste entartet!

39. Gegeben folgende Frequenztabelle für eine Datei:

a	b	d	e	n	w
5	2	1	6	3	9

    39.1. Konstruieren Sie den Codebaum, den der Huffman-Algorithmus ergibt (die Zuordnung von „links“ und „rechts“ ist Ihnen freigestellt)

    39.2. Geben Sie die resultierende Codetabelle an (die Zuordnung von 0 und 1 ist Ihnen ebenfalls freigestellt).

    39.3. Codieren Sie die Nachricht „badewanne“!

40. Geben Sie (in Pseudocode, Java ...) einen Algorithmus an, der für einen vollständigen Binärbaum (gespeichert als Array) überprüft, ob es sich um einen Max-Heap handelt. Der Algorithmus soll in O(n) Schritten arbeiten. Erläutern Sie in Worten die Arbeitsweise Ihres Algorithmus.

41. Welche asymptotische Laufzeit hat Heapsort auf einem absteigend sortierten Array? Wie sieht das praktisch aus?

42. Nehmen Sie die Zahlenfolge 12, 4, 7, 1, 9, 5, 11, 2, 3, 10, 6, 8 und fügen Sie diese Folge in einen min-Heap und in einen sortierten Binärbaum ein!

43. Geben Sie einen Algorithmus an, der für einen beliebigen (!) Binärbaum entscheidet, ob er voll ist. Beschreiben Sie zunächst, wann das der Fall ist und verwenden Sie diese Ihre Definition zur Implementation.

44. Welches ist die Mindestanzahl von Knoten, die ein AVL-Baum der Höhe h hat?

45. Beweisen Sie bitte: $\forall n, m \in \mathbb{N}$ gilt:

$$\lfloor \frac{n+m}{2} \rfloor + \lfloor \frac{n-m+1}{2} \rfloor = \lfloor \frac{2n+1}{2} \rfloor$$

46. Beweisen Sie bitte: $\forall n, m \in \mathbb{N}$ gilt:

$$\lceil \frac{n+m}{2} \rceil + \lceil \frac{n-m+1}{2} \rceil = \lceil \frac{2n+1}{2} \rceil$$

47. Beweisen Sie bitte:

$$\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x+y \rfloor$$

Unter welchen Bedingungen gilt „=“?

48. Beweisen Sie bitte:

$$\lceil x \rceil + \lceil y \rceil \geq \lceil x+y \rceil$$

Unter welchen Bedingungen gilt „=“?

49. ([M20]) Zeigen Sie, dass ein stabiler Sortieralgorithmus genau eine Permutation von n Elementen liefern kann.

50. Diese Aufgabe besteht aus zwei Teilen:

    50.1. ([M45]) Beweisen Sie

$$\pi = \frac{9801}{\sqrt{8}} \left(\sum_{n=0}^{\infty} \frac{(4n)!(1103+26390n)}{(n!)^4 396^{4n}} \right)^{-1}$$

    50.2. ([20]) Ermitteln Sie die Aufwandsfunktion für die Ermittlung einer Näherung von π in sinnvollen Größen!

51. Ramanchandra Kaprekar (1905 - 1986) war einer der berühmtesten indischen Mathematiker. Er entdeckte eine Besonderheit der Ziffernfolge $1, 4, 6, 7$.

    Satz von Kaprekar:

    Man nehme eine beliebige vierziffrige Zahl (z. B.:2195), bei der nicht alle vier Ziffern gleich sind (also nicht $1111, 2222, 3333, \ldots$) und ordne die vier Ziffern der Größe nach absteigend (im Beispiel: 9521). Von der so entstehenden Zahl subtrahiere man die Zahl, die bei aufsteigender Anordnung der Ziffern entsteht (im Beispiel: 1259). Mit der entstandenen neuen Zahl (im Beispiel: $9521 - 1259 = 8262$) wird der Prozess wiederholt. Nach einigen Schritten landet man immer bei der Zahl 6174, die sich dann bei weiteren Schritten stets selbst reproduziert. Besonders kurz ist dieser Prozess für 2004. Finden Sie heraus, wie sich das für andere Startwerte entwickelt.

52. Die Binomialkoeffizienten n über k (siehe auch Pascalsches Dreieck) sind definiert als

$$\binom{n}{k} := \frac{n!}{k!(n-k)!} \; \forall k, n \; \geq 0$$

    52.1. Untersuchen Sie verschiedene Algorithmen zur Berechnung der Binomialkoeffizienten darauf, für welche Werte n, k und insbesondere bis für welche Werte von $\binom{n}{k}$ sie ein korrektes Ergebnis für `int` bzw. `long` in Java bzw. C++ liefern.

    52.2. Ermitteln Sie, wie viele Rechenoperationen die verschiedenen Verfahren benötigen!

    52.3. Visualisieren Sie das Ergebnis!

    52.4. Können Sie das Ergebnis auf allgemeine 8, 16, 32, 64 Bit Rechner verallgemeinern?

53. Geben Sie die ersten 40 binären Stellen von π an!

54. Arbeiten Sie die Lösch-Operation (vgl. S. 91 in einem binären Suchbaum so aus, dass Sie leicht und sicher implementieren können!

55. Zeigen Sie, wie ein Stack mittels zweier Queues implementiert werden kann! Analysieren Sie das asymptotische Verhalten dieser Stack-Implementierung!

56. Zeigen Sie, wie eine Queue mittels zweier Stacks implementiert werden kann! Analysieren Sie das asymptotische Verhalten dieser Stack-Implementierung!

57. Eine Freundin rief mich am Tag vor dem Schuljahresbeginn 2009 an und bat um Hilfe bei folgendem Problem:

    > Es gibt fünf neue Parallelklassen a, b, c, d, e. Jede von diesen hat einen Klassenraum Ra, Rb, Rc, Rd, Re. Damit sich die Schüler und Schülerinnen über Klassengrenzen hinweg und auch die verschiedenen Klassenlehrer kennenlernen, sollen die Schüler für fünf Projekte an fünf Terminen gemischt werden. Jede Klasse wird in fünf feste (Farb)Gruppen aufgeteilt (weiß, gelb, grün, blau, rot). Jedes Projekt findet in einem der Klassenräume statt. Die Gruppen sollen nun so für die fünf Termine gemischt werden, dass folgende Bedingungen erfüllt werden:

    57.1. Keine Farbgruppen aus einer Parallelklasse in einem Projekttermin und Raum.

    57.2. Keine Farbgruppe mehrfach in einem der Räume.

    57.3. Keine Farbgruppe mehrfach mit einer anderen in einem Projekt.

    Entwickeln Sie einen Algorithmus, um festzustellen, ob dieses Problem eine Lösung hat und ggf. alle Lösungen findet.

58. Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

    in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod\ 9$ in eine Hashtabelle der Größe $N = 9$ (nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge durch quadratisches Sondieren ($h_i(k) = (h(k) + i^2)$ mod 9).

    Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h bzw. bei Kollisionen der Hashfunktionen h_i an.

59. Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

    in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod\ 9$ in eine Hashtabelle der Größe $N = 9$ (nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge Verkettung.

    Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h bzw. bei Kollisionen der Hashfunktionen h_i an.

60.

Lösungen

1. Welche wichtigen Unterschiede gibt es zwischen $\mathbb{Z}$ und dem Datentyp `int` in einer Programmiersprache wie C++ oder Java?

   Antwort:

Kriterium	$\mathbb{Z}$	`int`
Wertebereich	$(-\infty, +\infty)$	[-MAXINT,+MAXINT]
Addition Subtraktion Multiplikation	exakt	Nur im Bereich Ansonsten Overflow
Division	Nur auf Teilbereich	ganzzahlige

2. Schreiben Sie die Signaturen für folgende Operationen des Typs `int` auf:

$$++ \quad *= \quad \leq \quad \%$$

   Antwort:

$$\begin{array}{lcccccc} ++ & : & & & \text{int} & \rightarrow & \text{int} \\ *= & : & \text{int} & \times & \text{int} & \rightarrow & \text{int} \\ \leq & : & \text{int} & \times & \text{int} & \rightarrow & \text{bool} \\ \% & : & \text{int} & \times & \text{int} & \rightarrow & \text{int} \end{array}$$

3. Ermitteln Sie die Aufwandsfunktion für die Operationen `pop` und `push` eines Stacks!

   Antwort: Ich entscheide mich dafür, den Stack mittels einer verketteten Liste zu implementieren. `push` besteht dann im verketten eines neuen Elements am Listenkopf, `pop` ist dann das entfernen des Listenkopfs. Beide Operationen erfordern die Manipulation einer Referenz (oder pointers). Beide Operationen sind also ein $\mathcal{O}(1)$. Die Aufgabe hätte also besser gelautet: Ermitteln Sie für eine Implementierung die Aufwandsfunktion für die Operationen `pop` und `push` eines Stacks!

4. Finden Sie bitte Antworten auf folgende Fragen:

   4.1. Wofür steht die Abkürzung OCL?

   4.2. Von welcher größeren Spezifikation ist OCL ein Teil oder welche Spezifikation wird durch OCL ergänzt?

   4.3. Von wem wurde OCL entwickelt?

   4.4. Für welchen Zweck ist OCL entwickelt worden?

   4.5. Geben Sie ein Beispiel für OCL an!

   Antwort:

   4.1. Object Constraint Language

4.2. UML

4.3. [WK99]

4.4. OCL wurde für die Spezifikation von Bedingungen in objektorientierten Modellen entwickelt.

4.5. Sei X eine Klasse, a ein Attribut und op eine Operation von X. Dann kann man die Nachbedingung dass a nach ausführen von op das Atrribut a größer als Null ist, z. B. so formulieren: @post: $a > 0$.

5. Wie werden Vor- und Nachbedingungen in den folgenden Sprachen geschrieben?

   5.1. UML

   5.2. C++

   5.3. Java

   Finden Sie heraus, wie dies in anderen Sprachen gemacht wird.

   Antwort:

   5.1. pre ..., post ...

   5.2. Beide können über `assert` abgeprüft werden. Es können Exceptions geworfen werden.

   5.3. Beide können über `assert` abgeprüft werden. Es können Exceptions geworfen werden.

6. Beschreiben Sie die Operationen push, pop, top und isEmpty eines Stacks durch Angaben von Signatur, Vor- und Nachbedingung!

   Antwort: Bezeichnen wir die Klasse der Elemente des Stacks mit „Element“, so kann man die geforderten Angaben wie folgt gestalten.

push	Signatur	Stack×Element	⟶	Stack
	Vorbedingung			
	Nachbedingung			
pop	Signatur	Stack	⟶	Stack×Element
	Vorbedingung	Stack not empty		
	Nachbedingung	letztes Element entfernt		
isEmpty	Signatur	Stack	⟶	bool
	Vorbedingung			
	Nachbedingung			

   Hierbei habe ich mich auf die schmalste denkbare Schnittstelle beschränkt. So entfernt „pop“ nicht nur das oberste Element, sondern liefert es auch zurück.

7. Beschreiben Sie die Operationen front, enqueue, dequeue, isEmpty einer Queue durch Angaben von Signatur, Vor- und Nachbedingung!

enqueue	Signatur	Queue×Element	⟶	Queue
	Vorbedingung			
	Nachbedingung			
front	Signatur	Queue	⟶	Element
	Vorbedingung	not empty		
	Nachbedingung			
dequeue	Signatur	Queue	⟶	Queue×Element
	Vorbedingung	not empty		
	Nachbedingung			
isEmpty	Signatur	Queue	⟶	bool
	Vorbedingung			
	Nachbedingung			

8. Schreiben Sie Beispiel 1.14 so um, dass die Vorbedingung entfallen kann!

   Antwort: Wir können ohne Einschränkung annehmen

   $$P := grad(p) < Q =: grad(q).$$

   Wäre dies nämlich nicht der Fall, so brauchen wir nur p und q zu vertauschen.

   Damit lautet dann die Nachbedingung ohne eine Vorbedingung zu benötigen.

   $$\begin{aligned} \text{post:} \quad & \text{sei } p = \sum_{i=0}^{N} a_i \cdot x^i,\ q = \sum_{i=0}^{N} b_i \cdot x^i,\ b_i = 0, i = p+1, \ldots, q, \\ & \text{so ist } p + q = \sum_{i=0}^{Q} (a_i + b_i) \cdot x^i \end{aligned}$$

9. Es sei ein ADT X gegeben (denken Sie an Liste, Array, Tablespace ...). Dessen Implementierung sei so, dass eine Vergrößerung einen fixen Aufwand f erfordert und für jedes Element, für das Platz geschaffen wird ein variabler Aufwand von v anfällt. Untersuchen Sie die folgenden Wachstumstrategien auf ihre Effizient:

   9.1. Minimaler Speicherplatz: Der ADT wird leer angelegt und immer, wenn ein Element eingefügt wird, um (den Platz für) ein Element vergrößert.

   9.2. Prozentuale Vergrößerung: Ist der Platz gefüllt, so wird der ADT (den Platz für) p% mehr Elemente als vorher vergrößert.

   Antwort: Bei der Strategie des minimalen Speicherplatzes wird bei jedem Einfügen vergrößert. Der Aufwand zum Einfügen von n Elementen ist hier $(f + v) \cdot n$.

   Bei der Strategie der prozentualen Vergrößerung betrachte ich einen ADT der mit der Größe n_0 beginnt und die Vergrößerung auf $n = n_0 * (1 + p)^k$. Der Aufwand hierfür ist $k * f + v * (n - n_0)$. Damit ist der Gesamtaufwand durch $f * log\ n + v * \log(1 + p) log\ n$ abzuschätzen.

   Im ersten Fall haben wir ein lineares, im zweiten ein logarithmisches Wachstum des Aufwandes mit n.

   Vergleichen wir dies für einige Werte von n und $n_0 = 1$, $p = 1$

n	minimal	prozentual	Differenz
1	f+v	f+v	0
2	$2 \cdot (f + v)$	$2 \cdot (f + v)$	0
4	$4 \cdot (f + v)$	$3 \cdot f + 4 \cdot v)$	$1 \cdot f$
8	$8 \cdot (f + v)$	$4 \cdot f + 8 \cdot v)$	$4 \cdot f$
16	$16 \cdot (f + v)$	$5 \cdot f + 16 \cdot v)$	$11 \cdot f$

10. Beweisen Sie, dass die angegebenen Aufwandsfunktionen (Θ) für die Operationen in den verschiedenen Implementationen einer Liste (Implementationen 0–3) korrekt sind!

    Antwort: Dies sind $16 = 4 \cdot 4$ Aufgaben, die alle nach dem gleichen Schema zu lösen sind. Ich beginne einfach mal mit den ersten und werde die Liste im Laufe der Zeit vervollständigen:

    Implementierung 0 Dies ist die Array-Implementierung.

    10.1. *insert*-Operation: Der Code für die Implementierung dieser Operation lautet hier so:

```
insert(int p, Datensatz e)
{
  for(int i = listsize;i>=p;i--)
     t[i+1] = t[i];
}
t[p] = e;
```

Hierbei ist angenommen, dass $listSize < max$ ist. Anderfalls komt noch der Vergrößerungsaufwand hinzu. Sind die p gleichverteilt, so sind also im Durchschnitt $\frac{n}{2}$ Zuweisungen (Kopien) und eine weitere Zuweisung notwendig, also $\Theta(N)$.

Implementation 3 Verkettete Liste mit Kopf- und Schwanzzeiger und antizipativer Indizierung. Hier ist es ganz wichtig, dass die „Position", die bisher einen Schlüssel repräsentierte, „umgedeutet" wird. Die Menge der Positionen ist nun eine Menge von Pointern oder Referenzen auf Elemente. Das erste Element wird mit dem `next`-Attributwert des Kopf-Knotens indiziert, das p-te mit dem `next`-Attributwert des $p-1$-ten etc. `tail.next` zeigt auf das letzte Element.

Nach diesen Vorbemerkungen können alle Behauptungen leicht bewiesen werden. Man muss vor allem, die Implementierungen der Operationen angeben.

10.1. Einfügen: Die `insert(p, e)` Operation kann so implementiert werden (vergleiche auch Skript):

```
q = new Knoten(p);
q.dat = e;
q.next = p.next;
p.next = q;
if(q.next == tail)
   tail.next = q;
```

Pro Einfügevorgang haben wir so einen neuen Knoten zu erzeugen, einen Vergleich und $3-4$ Zuweisungen. Also ist der Aufwand zum Einfügen konstant, also $\Theta(1)$, unabhängig von der Größe der Liste.

11. Zeigen und beweisen Sie, dass Stack- und Queue-Operationen mit konstantem Aufwand implementiert werden können!

Antwort: Dies kann man natürlich wie bei der Liste direkt beweisen. Da wir die Eigenschaften für die Listenoperationen bereits bewiesen haben, können wir es auch auf diese zurückführen. Ich implementiere Stack oder Queue einfach mittels einer Liste (Aggregation oder Komposition). Das Attribut mit der Referenz auf diese Datenstruktur bezeichne ich mit `liste`. Dann sieht der entscheidende Teil der Implementierung so aus:

```
class Queue{
  //...
  List liste;
  //...
  push(Element e){
    liste.insert(list.front,e);
  }
  Element pop(){
    liste.retrieve(list.front);
    liste.delete(list.front);
  }
}
```

Sowohl `push` als auch `pop` benötigen also konstanten Aufwand, also $\mathcal{O}(1)$

12. Beweisen Sie Satz 2.21!

$$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$$

Antwort: Dies ist ein einfaches Nachrechnen der Definitionen:

$$\begin{aligned} & g \in \mathcal{O}(f) \cap \Omega(f) \\ :\Longleftrightarrow \quad & \exists n_0, c : g(x) \leq c \cdot f(x) \forall x \geq n_0 \wedge \exists m_0, d : d \cdot f(x) \leq g(x) \forall x \geq m_0 \\ \Longleftrightarrow \quad & \exists n_0, c, d : d \cdot f(x) \leq g(x) \leq c \cdot f(x) \forall x \geq n_0 \\ \Longleftrightarrow : \quad & g \in \Theta(f). \end{aligned}$$

Wir brauchen ja nur das Maximum der von n_0 und m_0 aus der zweiten Zeile zu nehmen.

13. Beantworten Sie die Fragen aus Aufgabe 2.17 und beweisen Sie Ihre Behauptungen:

13.1. $n^2 = \mathcal{O}(n^3)$
Diese Aussage ist wahr. Es gilt $n^2 \leq n^3 \ \forall \ n \geq 1$. Also ist $c = 1, n_0 = 1$.

13.2. $n^3 = \mathcal{O}(n^2)$
Nach dem vorstehenden Beweis ist diese Aussage falsch.

13.3. $2^{n-1} = \mathcal{O}(n^2)$
Diese Aussage ist falsch. Für $n > 7$ ist $2^{n-1} > n^2$

13.4. $(n+1)! = \mathcal{O}(n!)$
Diese Aussage ist falsch: Für $n > c$ ist $(n+1)! > cn!$. Die Umkehrung ist allerdings richtig, wenn auch nicht nützlich:

$$\lim_{n \to \infty} \frac{n!}{n+1!} = \lim_{n \to \infty} \frac{1}{n+1} = 0.$$

13.5. $\sqrt{n} = \mathcal{O}(\ln n)$
Dies Aussage ist falsch: Es gilt $\sqrt{n} \geq \ln n \ \forall \ n \geq 1$

13.6. $28 = \mathcal{O}(1)$
Diese Aussage ist offensichtlich wahr mit $c = 28$ und $n_0 = 1$

13.7. $12 \cdot n + 5 = \mathcal{O}(n \ln n)$
Diese Aussage ist wahr aber nicht nützlich:
$12 \cdot n + 5 <= (n \ln n) \ \forall \ n > e^{13} \approx 442.414$

13.8. $n \ln n = \mathcal{O}(n^2)$
Diese Aussage ist wahr, aber ebenfalls kaum nützlich:
$n \ln n \leq n^2 \ \forall n > 1$

14. Sehr oft wird eine Operation swap eingesetzt. Diese benötigt drei Zuweisungen, um die Vertauschung der Elemente vorzunehmen. Wie und mit wievielen Vertauschungen überführen Sie:

14.1. (a,b,c) in (c, b, a)?

14.2. (a,b,c,d) in (b,c,d,a)?

Antworten: Die einfachsten Antworten dürften folgende sein:

14.1. swap(a,c) leistet dies mit einer Vertauschung, also drei Zuweisungen.

14.2. Hierfür brauchen wir diese Folge (t temporäre Variable) von Vertauschungen:

$$t \leftarrow a, a \leftarrow b, b \leftarrow c, c \leftarrow d, d \leftarrow t$$

Insgesamt also fünf Zuweisungen.

15. Wie sieht eine Gerade $y = a \cdot x + b$ in doppelt logarithmischer Darstellung aus?

    Antwort: Ich substituiere $x = 10^{\xi}$, $y = 10^{\chi}$

    Dann ist in diesem Fall

    $$10^{\chi} = y = a \cdot x + b = a \cdot 10^{\xi} + b10^{\chi} = a \cdot 10^{\xi} + b\chi = log(a \cdot 10^{\xi} + b)$$

    Für $b = 0$ ergibt sich damit einfach:

    $$\chi = log(a \cdot 10^{\xi}) = \xi + loga$$

    Das ist eine Gerade mit Steigung 1, die die χ Achse bei $\log a$ schneidet.

16. Wie sieht die Wurzelfunktion $y = \sqrt{x}$ in doppelt logarithmischer Darstellung aus?

    Antwort: Substitution von $y = 10^{\chi}$ und $x = 10^{\xi}$ liefert:

    $$\chi = log10 \cdot \frac{\xi}{2},$$

    d. h. eine Gerade durch den Ursprung des $\xi - \chi$-Koordinatensystems mit Steigung $0,5$.

17. Wie sieht die Exponentialfunktion $y = e^x$ in doppelt logarithmischer Darstellung aus?

    Antwort: Ich substituiere $x = 10^{\xi}$, $y = 10^{\chi}$.

    Dann ist in diesem Fall

    $$10^{\xi} = e^{10^{\chi}}$$

    Logarithmieren auf beiden Seiten liefert:

    $$\xi =$$

18. Wie sieht die Funktion $y = \frac{1}{x}$ in doppelt logarithmischer Darstellung aus?

    Antwort: Ich substituiere $x = 10^{\xi}$, $y = 10^{\chi}$. Das liefert

    $$\chi = log\frac{1}{10^{\xi}} = -\xi$$

    Also eine Gerade mit der Steigung -1 durch das $\xi - \chi$-Korrdinatensystem.

19. Analog die Darstellung in lin-log, log-lin. Welche der drei Darstellungen sind für welche Zwecke sinnvoll?

20. Beweisen Sie die folgende Aussage!

    $$\sum_{k=1}^{n} \lfloor \frac{k}{2} \rfloor = \lfloor \frac{n^2}{4} \rfloor$$

    Beweis: Wie bei vielen Aussagen über natürliche Zahlen liegt hier ein Versuch eines Beweises mit nvollständiger Induktion nahe. Versuchen wir es also auf diese Weise:

    I.V.: Für n = 1 gilt:

    $$\sum_{k=1}^{1} \lfloor \frac{k}{2} \rfloor = \lfloor \frac{1}{2} \rfloor = 0 = \lfloor \frac{1^2}{4} \rfloor = 0$$

    I.A.: Die Aussage sei bereits für alle $n \leq n_0$ bewiesen.

I.S.: Ich fange zunächst einfach an dies nachzurechnen:

$$\begin{aligned}\sum_{k=1}^{n_0+1}\lfloor\frac{k}{2}\rfloor &= \sum_{k=1}^{n_0}\lfloor\frac{k}{2}\rfloor + \lfloor\frac{n_0+1}{2}\rfloor \\ &= \lfloor\frac{n_0^2}{4}\rfloor + \lfloor\frac{n_0+1}{2}\rfloor \\ &= \lfloor\frac{n_0^2}{4}\rfloor + \lfloor\frac{2(n_0+1)}{4}\rfloor\end{aligned}$$

Der letzte Ausdruck sieht schon sehr nach $\frac{(n_0+1)^2}{2}$ aus, wenn da nicht die Abrundung auf die nächst kleinere ganze Zahl wäre. Hier liegt also eine Fallunterscheidung nach n_0 gerade bzw. ungerade nahe.

Ist n_0 gerade, so ist n_0^2 durch 4 teilbar. Der gebrochene Anteil kommt also aus dem zweiten Summanden, dessen ersten Teil durch vier Teilbar ist und ist $\frac{1}{2}$. Das ist aber gerade $\lfloor\frac{(n_0+1)^2}{2}\rfloor$.

Ist n_0 ungerade, so ist n_0+1 gerade und der zweite Ausdruck durch vier teilbar. Der gebrochene Anteil stammt aús dem ersten Ausdruck. Ein ungerades n_0 hat die Form $n_0 = 2\cdot k+1$. Dann ist

$$\begin{aligned}\lfloor\frac{n_0^2}{4}\rfloor &= \lfloor\frac{(2\cdot k+1)}{4}\rfloor \\ &= \lfloor\frac{4\cdot k^2+4\cdot k+1}{4}\rfloor\end{aligned}$$

Auch hier wird durch das Abschneiden des gebrochenen Anteils das gleiche Ergebnis erzielt.

Es geht aber auch ohne vollständige Induktion ([Knu73a]):

Sei $n = 2t$. Dann ist

$$\begin{aligned}\sum_{k=1}^{n}\lfloor\frac{k}{2}\rfloor &= \sum_{k=1}^{n}\lfloor\frac{n+1-k}{2}\rfloor \\ &= \frac{1}{2}\sum_{k=1}^{n}\underbrace{(\lfloor\frac{k}{2}\rfloor + \lfloor\frac{n+1-k}{2}\rfloor)}_{\begin{array}{l}\frac{k}{2}+\frac{n-k}{2},\text{ wenn k gerade}\\ \frac{k-1}{2}+\frac{n+1-k}{2},\text{ wenn k ungerade}\end{array}} \\ &= \frac{1}{2}\sum_{k=1}^{n}\frac{n}{2} \\ &= \frac{1}{2}n\cdot\frac{n}{2} \\ &= \frac{n^2}{4}\end{aligned}$$

Für $n = 2t + 1$ ist $t = \frac{n-1}{2}$

$$\begin{aligned}
\sum_{k=1}^{2t+1} \lfloor \frac{k}{2} \rfloor &= \sum_{k=1}^{2t} \lfloor \frac{k}{2} \rfloor + \lfloor \frac{2t+1}{2} \rfloor \\
&= t^2 + t \\
&= \frac{(n-1)^2}{4} + \frac{2n-2}{4} \\
&= \frac{n^2}{4} - \frac{1}{4} \\
&= \lfloor \frac{n^2}{4} \rfloor
\end{aligned}$$

21. Beweisen Sie die folgende Aussage!

$$\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil = \lceil \frac{n \cdot (n+2)}{4} \rceil$$

Beweis: Ganz analog zur vorstehenden Aufgabe können wir auch hier rechnen. Sei zunächst $n = 2t$ gerade:

$$\begin{aligned}
\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil &= \sum_{k=1}^{n} \lceil \frac{n+1-k}{2} \rceil \\
&= \frac{1}{2} \sum_{k=1}^{n} \underbrace{(\lceil \frac{k}{2} \rceil + \lceil \frac{n+1-k}{2} \rceil)}_{} \\
&\quad = \frac{k}{2} + \frac{n+2-k}{2}, \text{ wenn k gerade} \\
&\quad = \frac{k+1}{2} + \frac{n+1-k}{2}, \text{ wenn k ungerade} \\
&= \frac{1}{2} \sum_{k=1}^{n} \lceil \frac{2t+2}{2} \rceil \\
&= \frac{1}{2} 2t \cdot \frac{2t+2}{2} \\
&= \frac{n(n+2)}{4} \\
&= \lceil \frac{n(n+2)}{4} \rceil
\end{aligned}$$

Die letzte Umformung gilt, da mit n auch $n + 2$ gerade ist.

Sei nun $n = 2t + 1$. Dann gilt

$$\begin{aligned}
\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil &= \sum_{k=1}^{2t} \lceil \frac{k}{2} \rceil + \lceil \frac{2t+1}{2} \rceil \\
&= \frac{1}{2} 2t \cdot \frac{2t+2}{2} + \lceil \frac{2t+1}{2} \rceil \\
&= \frac{1}{2} 2t \cdot \frac{2t+2}{2} + \frac{2t+2}{2} \\
&= \frac{(n-1)(n+1) + 2n + 1}{4} \\
&= \frac{n^2 - 1 + 2n + 1}{4} \\
&= \frac{n(n+2)}{4}
\end{aligned}$$

Hier noch eine weitere Beweisvariante von Christian Braun aus dem SS 2010:
Zunächst einmal ist folgendes wahr:

$$\lceil \frac{k}{2} \rceil = \frac{k}{2} + \frac{1}{4}(1 - (-1)^k).$$

Nun kann man unter Verwendung der bekannten Summenformel einfach rechnen:

$$\begin{aligned}
\sum_{k=1}^{n} \lceil \frac{k}{2} \rceil &= \sum_{k=1}^{n} (\frac{k}{2} + \frac{1}{4}(1 - (-1)^k) \\
&= \sum_{k=1}^{n} \frac{k}{2} + \frac{n}{4} - \sum_{k=1}^{n} (-1)^k \\
&= \frac{n(n+1)}{4} + \frac{n}{4} + frac12(1 - (-1)^n). \\
&= \frac{n(n+1)}{4} + \frac{n}{4}, \text{n gerade} \\
&= \frac{n(n+1)}{4} + \frac{n}{4} + 1, \text{n ungerade} \\
&= \frac{n(n+2)}{4}, \text{n gerade} \\
&= \frac{n(n+2)}{4} + 1, \text{n ungerade}
\end{aligned}$$

22. Beweisen Sie bitte folgende Aussage!

$$\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2$$

Beweis: Noch ein Beispiel für vollständige Induktion. Es wird sich aber als nützlich erweisen, von der Formel

$$\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$$

Gebrauch zu machen. Ich beweise also

$$\sum_{k=1}^{n} k^3 = (\frac{n \cdot (n+1)}{2})^2$$

I.V.: Für $n = 1$ ist

$$\begin{aligned}\sum_{k=1}^{1} k^3 &= 1 \\ &= \frac{1 \cdot 2}{2}\end{aligned}$$

I.A.: Die Aussage sei für alle $n \leq n_0$ bewiesen:

$$\sum_{k=1}^{n_0} k^3 = (\frac{n_0 \cdot (n_0 + 1)}{2})^2$$

I.S.:

$$\begin{aligned}\sum_{k=1}^{n_0+1} k^3 &= (\frac{n_0 \cdot (n_0 + 1)}{2})^2 + (n_0 + 1)^3 \\ &= \frac{(n_0 + 1)^2}{4} \cdot (n_0 + \cdot 4 \cdot (n_0 + 1) \\ &= \frac{(n_0 + 1)^2}{4} \cdot (n_0 + 2)^2 \\ &= (\frac{(n_0 + 1) \cdot (n_0 + 2)}{2})^2\end{aligned}$$

Sie können die Gaussschen Summenformel aber auch direkt in der Rechnung einsetzen: An der Induktionsverankerung ändert sich nichts. Der Rest geht so:

I.V.: Für $n = 1$ ist

$$\begin{aligned}\sum_{k=1}^{1} k^3 &= 1 \\ &= \frac{1 \cdot 2}{2}\end{aligned}$$

I.A.: Die Aussage sei für alle $n \leq n_0$ bewiesen:

$$\sum_{k=1}^{n_0} k^3 = (\sum_{k=1}^{n_0} k)^2$$

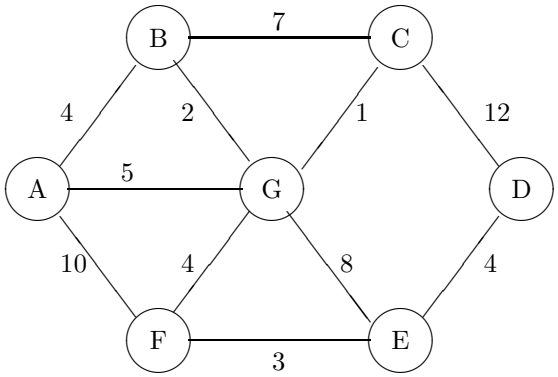


Abbildung B.2: Ein Projektplan

I.S.:

$$\begin{aligned}
\sum_{k=1}^{n_0+1} k^3 &= (\sum_{k=1}^{n_0} k)^2 + (n_0+1)^3 \\
&= (\sum_{k=1}^{n_0} k)^2 + (n_0+1)^2 n_0 + (n_0+1)^2 \\
&= (\sum_{k=1}^{n_0} k)^2 + (n_0+1)(n_0+1)n_0 + (n_0+1)^2 \\
&= (\sum_{k=1}^{n_0} k)^2 + 2(n_0+1)\frac{(n_0+1)n_0}{2} + (n_0+1)^2 \\
&= (\sum_{k=1}^{n_0} k)^2 + 2(\sum_{k=1}^{n_0} k)(n_0+1) + (n_0+1)^2 \\
&= (\sum_{k=1}^{n_0+1} k)^2
\end{aligned}$$

23. Der ADT *deque* ist eine Liste, bei der Einfügungen und Entfernungen am Anfang oder Ende der Liste vorgenommen werden, ebenso wie viele der Zugriffe. Die fundamentalen Operationen fügen am Anfang bzw. Ende ein und entnehmen entsprechend: insertFront, insertRear, deleteFront, deleteRear. Ferner gibt es die lesenden Operationen getFront, getRear. Geben Sie an, wie dieser ADT mit einem Array und wie er mit einer Liste implementiert werden kann. (Vorzugsweise Java, C++, andere Sprachen können akzeptabel sein, im Zweifel: fragen!).

24. In Kapitel 4 wurde der Algorithmus von Dijkstra vorgestellt. Formulieren Sie auf dieser Basis einen Algorithmus, der *längste* Wege findet. Wenden Sie ihn auf den Graphen

    aus Kap. 5 des Skripts (Siehe Abb. B.2) an, um den längsten Weg von A nach D zu finden. Verwenden Sie dazu eine Tabelle, wie im Skript!

Lösung: Den Algorithmus formuliere ich analog zum Skript. Die Idee des Algorithmus zur bestimmung längster Wege besteht darin, wellenförmig vom Startknoten σ aus alle anderen Knoten zu erforschen (Breitendurchlauf). Dabei werden die Kosten zu den Nachbarknoten registriert. Die Nachbarknoten werden dann bewertet mit den maximalen Kosten die anfallen um σ zu erreichen. Außerdem merken sich die Knoten noch, auf zu welchem Nachbarknoten sie gehen müssen um den teuersten Weg zu σ zu finden. Um Zyklen sicher zu vermeiden nehme ich an, das der Graph gerichtet ist.

Jeder Knoten hat drei (zusätzliche) Attribute:

- *pred* für den Vorgänger Knoten zu dem man gehen muß um am teuersten nach σ zu kommen,
- *cost* hält die maximalen Kosten auf dem Weg zu σ
- *marked* merkt sich, ob der Knoten schon abschließend behandelt wurde.

Der Algorithmus hat nun folgenden Ablauf:

24.1. **Initialisierung:** alle Knoten κ außer dem Startknoten σ bekommen folgende Initialisierung:

- κ.pred = undefiniert;
- κ.cost = 0;
- κ.marked = false;

Der Startknoten σ wird wie folgt initialisiert

- σ.pred = σ;
- σ.cost = 0;
- σ.marked = true;

24.2. Bestimme den Rand R, der aus adjazenten Knoten zu σ besteht.

24.3. **while** $R \neq \emptyset$ **do**

- wähle $\nu \in R$ so dass ν.cost maximal, und entferne ν aus R
- ν.marked = true
- ergänze Rand R bei ν

Beim Ergänzen des Randes R in einem Knoten ν muß folgendes gemacht werden:

24.1. wähle adjazente Knoten κ von ν die nicht markiert sind.

24.2. für jeden dieser Knoten κ finde heraus, ob

$$\kappa.cost < \nu.cost + c_{max}(\nu, \kappa).$$

Achtung: Hier muss beachtet werden, dass es auch in der ausgewählten Randmenge länger Wege geben kann. Wir ermitteln $c(\nu, \kappa)$ indem wir das Maximum über $\{c(\nu, \kappa), c(\nu, \rho) + c(\rho, \kappa) | \rho \in R\}$ Wenn dem so ist, dann setze

$$\kappa.cost = \nu.cost + c(\nu, \kappa)$$

und

$$\kappa.pred = \nu \; bzw. \; \rho.$$

, je nachdem, wo das Maximum gefunden wurde

24.3. nimm κ in R mit auf.

markiert	Randmenge
A.(A,0,1)	B.(A,4,0) G.(A,5,0) F.(A,10,0)
F.(A,10,1)	B.(A,4,0)G.(A,5,0) C.(B,11,0)
G.(A,5,1)	C.(G,6,0) F.(G,9,0) E.(G,13,0)
C.(G,6,1)	D.(C,18,0) F.(G,9,0) E.(G,13,0)
F.(G,9,1)	E.(F,12,0) D.(C,18,0)
E.(F,12,1)	D.(E,16,0)
D.(E,16,1)	

25. Beweisen Sie bitte die folgende Aussage!

$$\sum_{k=1}^{n}(2k-1) = n^2$$

Beweis:

I.V.: $1 = 1^2$

I.A.:

$$\sum_{k=1}^{n_0}(2k-1) = {n_0}^2$$

I.S.:

$$\begin{aligned}\sum_{k=1}^{n_0+1}(2k-1) &= \sum_{k=1}^{n_0}(2k-1) + (2\cdot n_0 - 1)\\ &= n_0^2 + (2\cdot n_0 - 1)\\ &= (n_0+1)^2\end{aligned}$$

Direkt aus der bekannten Formel

$$\sum_{k=1}^{n} k = \frac{n\cdot(n+1)}{2}$$

kann man das natürlich auch herleiten:

$$\sum_{k=1}^{n}(2k-1) = \sum_{k=1}^{n} 2k - n = n\cdot(n+1) - n = n^2$$

26. Beweisen Sie bitte die folgende Aussage!

$$\sum_{k=1}^{n} 2k = n\cdot(n+1)$$

Beweis:

I.V.: Für n=1 ist

$$\begin{aligned}\sum_{k=1}^{1} 2k &= 2\\ &= 1\cdot 2\end{aligned}$$

I.A.: Angenommen, die Aussage sei $\forall n \leq n_0$ bereits bewiesen, insbesondere:

$$\sum_{k=1}^{n_0} 2k \quad = \quad n_0 \cdot (n_0 + 1)$$

I.S.:

$$\begin{aligned} \sum_{k=1}^{n_0+1} 2k &= \sum_{k=1}^{n_0} 2k + 2 \cdot n_0 + 2 \\ &= n_0 \cdot (n_0 + 1) + 2(\cdot n_0 + 1) \\ &= (n_0 + 1) \cdot (n_0 + 2) \end{aligned}$$

27. Geben Sie die Inorder-, Preorder- und Postorder-Reihenfolgen des Baums in Abb. 4.5 an!

    Antwort:

    pre-order $* + *3124/57$(also: links-polnische Notation)

    inorder $3 * 12 + 4 * 5/7$ Bis auf die Klammern also die übliche mathematische Schreibweise. Mit Klammern also $((3 * 12) + 4) * (5/7)$

    postorder $312 + 4 + 57/*$(also: rechts-polnische Notation)

28. Fragen zu Rot-Schwarz-Bäumen

    28.1. Zu welcher Implementierung raten Sie für die „externen“ Knoten?

    28.2. Angenommen, die Wurzel eines Rot-Schwarz-Baumes sei rot. Bleibt es dann ein Rot-Schwarz-Baum, wenn wir sie schwarz färben?

    28.3. Beweisen Sie: Die Höhe eines Rot-Schwarz-Baumes ist nicht größer als $log_2 n$!

    28.4. Beweisen Sie: Der Pfad von der Wurzel zu einem Blatt ist nie mehr als doppelt so lang wie der kürzeste Weg dieser Art!

29. Diese Frage hat mit dem Thema zu tun, wenn auch nicht mit Informatik. Welche botanische Neuschöpfung zeigt die Rückseite der Deutschen 1, 2 und 5 Cent Münzen?

    Antwort: Auf diese Frage kam ich durch [Wan02]. Es soll sich offenbar um eine Eiche handeln. Die verbreitetsten Eichenarten in Europa sind Stieleiche (Quercus robur) und Traubeneiche (Quercus petraea). Stieleichen haben Eicheln mit deutlich ausgeprägten Stielen (wie in der Abbildung), aber ungestielte Blätter, die direkt am Ast ansetzen. Die Traubeneiche hat traubenförmig wachsende Eicheln an kurzen Stielen, aber gestielte Blätter. Die hier gezeigte hat gestielte Blätter. Zumindest bei dem Blatt unten rechts ist dies klar so gezeichnet. Da diese auch noch wechselständig sind, soll es sich ganz offenbar um eine Traubeneiche handeln. Auch die Blattform widerspricht dem zumindest nicht. Traubeneichen haben aber ungestielte Früchte (Eicheln). Gestielte Früchte, wie hier gezeigt, haben aber z. B. Stieleichen. Die haben aber ungestielte Blätter. [MW87], www.baumkunde.de

30. Nehmen Sie an, wir haben einen sortierten Binärbaum mit den Zahlen zwischen 1 und 1000 als Schlüssel und suchen darin die Zahl 363. Welche der Folgen können nicht die Folge der besuchten Knoten sein? Begründen Sie Ihre Antwort.

    30.1. 2, 252, 401, 398, 330, 344, 397, 363.

    30.2. 924, 220, 911, 244, 898, 258, 362, 363.

    30.3. 925, 202, 911, 240, 912, 245, 363.

    30.4. 2, 399, 387, 219, 266, 382, 381, 278, 363.

    30.5. 935, 278, 247, 621, 299, 392, 358, 363.

31. Zeigen Sie, wie man den Wert eines Polynoms vom Grad n mit n Multiplikationen und n Additionen berechnen kann!.

    Lösungsvorschlag: Das geht mit dem Horner-Schema

32. Gegeben die Zahlen von 1 bis 15 (beide einschließlich). Geben Sie eine Reihenfolge dieser Zahlen an, bei der der durch Einfügen dieser Zahlenfolge entstehende sortierte Binärbaum perfekt balanciert ist und zeichnen Sie den Baum auf

33. Gegeben die Zahlen von 1 bis 15 (beide einschließlich). Geben Sie eine Reihenfolge dieser Zahlen an, bei der der durch Einfügen dieser Zahlenfolge entstehende sortierte Binärbaum zu einer Liste entartet!

34. Gegeben folgende Frequenztabelle für eine Datei:

a	b	d	e	n	w
5	2	1	6	3	9

    34.1. Konstruieren Sie den Codebaum, den der Huffman-Algorithmus ergibt (die Zuordnung von „links“ und „rechts“ ist Ihnen freigestellt)

    34.2. Geben Sie die resultierende Codetabelle an (die Zuordnung von 0 und 1 ist Ihnen ebenfalls freigestellt).

    34.3. Codieren Sie die Nachricht „badewanne“!

    Lösung:

    34.1. Wir gehen vor wie in der Vorlesung und bauen den Baum schrittweise, beginnend mit den geringsten Häufigkeiten auf:
    d/1 b/2 11
    3 8
    d/1 b/2 n/3 a/5 28
    11 17
    3 8 w/9
    d/1 b/2 n/3 a/5

35. Geben Sie (in Pseudocode, Java ...) einen Algorithmus an, der für einen vollständigen Binärbaum (gespeichert als Array) überprüft, ob es sich um einen Max-Heap handelt. Der Algorithmus soll in O(n) Schritten arbeiten. Erläutern Sie in Worten die Arbeitsweise Ihres Algorithmus.

36. Welche asymptotische Laufzeit hat Heapsort auf einem absteigend sortierten Array? Wie sieht das praktisch aus?

37. Nehmen Sie die Zahlenfolge 12, 4, 7, 1, 9, 5, 11, 2, 3, 10, 6, 8 und fügen Sie diese Folge in einen min-Heap und in einen sortierten Binärbaum ein!

38. Geben Sie einen Algorithmus an, der für einen beliebigen (!) Binärbaum entscheidet, ob er voll ist. Beschreiben Sie zunächst, wann das der Fall ist und verwenden Sie diese Ihre Definition zur Implementation.

39. Was drückt Abb. 1.4 präzise aus? Was bleibt offen?

    Antwort: Die Antwort auf die Frage hängt zum Teil von der UML Version ab

    Versionsunabhängig
    - Bis auf den Rollennamen „head“ bei der Aggregationsraute bei „LISTE“ fehlen Rollennamen. Für die Rollennamen gibt es einen sinnvollen Defaultwert: Klassenname mit kleinem Anfangsbuchstaben. Das ist nicht weiter tragisch, aber so werden Hinweise nicht gegeben, die zumindest unerfahrene Entwickler brauchen könnten.

- Die rekursive Aggregation von Knoten zu Knoten hat keine Rollennamen. Das wird durch die Kennzeichnung der Assoziation nicht ausgeglichen!

prä UML 2.0 Sind Multiplizitäten nicht spezifiziert, so heißt dies: Multiplizität = 1. Dies heißt für das Diagramm in Abb. 1.4:

- Zu einer Liste gibt es genau einen Knoten. Der sollte auch als erster gekennzeichnet werden. Dort würde der Rollenname „head“ dann auch passen.
- Ein Knoten hängt in genau einer Liste. Das ist nicht aus der Definition einer Liste klar, sondern eine Design-Entscheidung.
- Zu einem „Ganzen“ Knoten gibt es genau einen darunter hängenden Knoten. Was ist mit dem Letzten? Was ist mit dem Ersten?

UML 2.0

40. Warum muss die innere Schleife beim Divisionsverfahren zur Primzahlbestimmung (S. 27) nur bis $\sqrt{i}$ laufen?

    Es sei $\sqrt{i} = N$

41. $$lnk < \sum_{i=1}^{k} \frac{1}{i} < 1 + lnk$$

    Beweis: Die angegebene Summe ist gerade die Approximation des Integrals $\int \frac{1}{x} = \ln$ über das Intervall $[1, k]$ durch eine Treppenfunktion von oben, also gilt

    $$\ln k < \sum_{i=1}^{k} \frac{1}{i}$$

    Verschieben wir die Treppenfunktion um 1 nach unten, so ist:

    $$\ln k > \sum_{i=1}^{k} \frac{1}{i} - 1$$

    also

    $$\sum_{i=1}^{k} \frac{1}{i} < 1 + \ln k$$

42. Beweisen Sie Satz 2.19 Es geht um ein Nachrechnen der Definition

    $\Rightarrow$

    $$f = \mathcal{O}(g) :\Leftrightarrow \exists n_0 \in \mathbb{N}, c \in \mathbb{R} mit f(n) \leq c \cdot g(n) \forall n > n_0$$

    $\Leftarrow$

43. Beweisen Sie Satz 2.21

    $$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$$

    Es geht um ein Nachrechnen der Definition

44. Lösung von Aufgabe 4.8

    44.1. In einem vollbesetzten Baum vom Grad d hat jeder innere Knoten d Söhne. Ist die Höhe h, so ist die Anzahl der Knoten also:

    $$\begin{aligned} N &= \sum_{k=0}^{h} d^k \\ &= \frac{d^{h+1} - 1}{d - 1} \end{aligned}$$

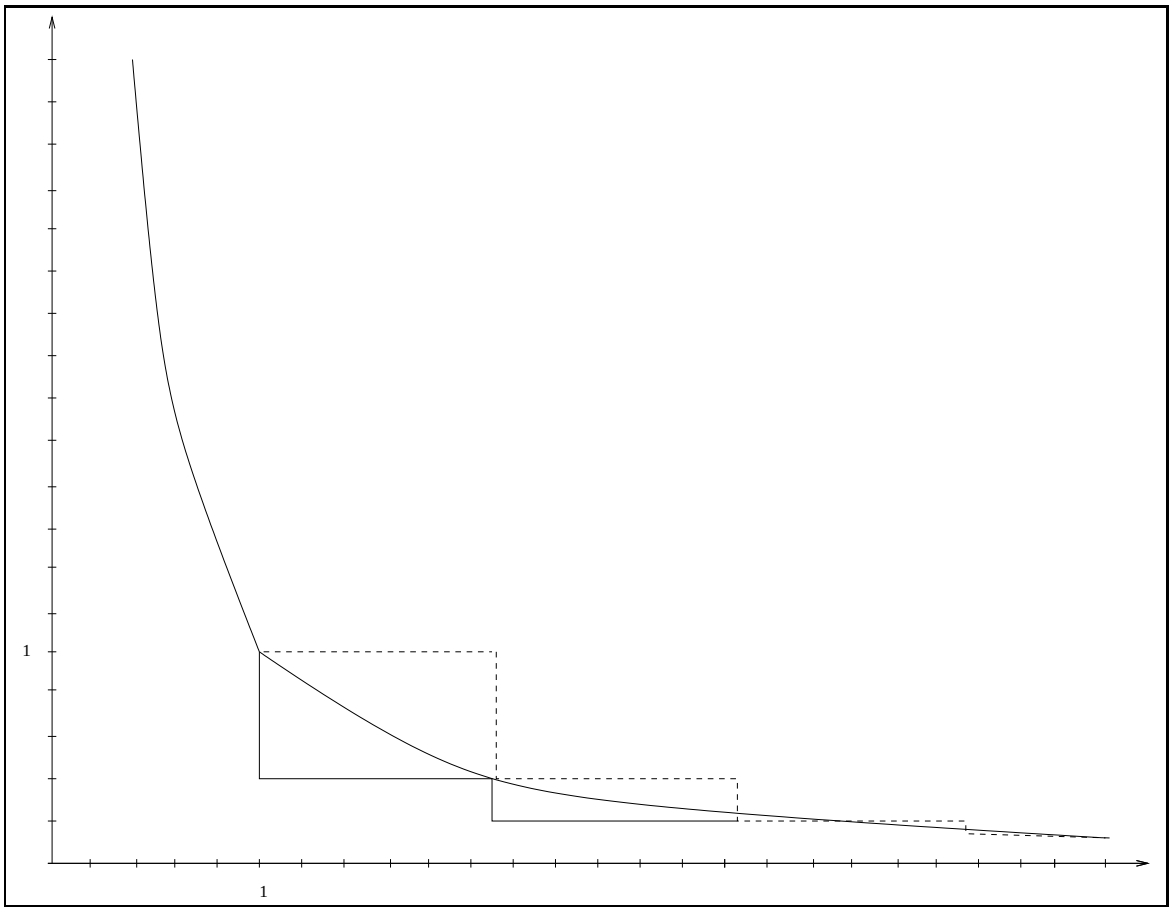


Abbildung B.3: Obere bzw. untere Treppenfunktion

44.2. Die minimale Höhe hat der vollbesetzte Baum. Wir brauchen also nur nach d^{h+1} aufzulösen und auf beiden Seiten zu logarithmieren:

$$h = log_d(N \cdot d - N + 1) - 1$$

45. Lösung von Aufgabe 4.9:

inorder 3,*,12,+,4,*,5,/,7

preorder *,+,*,3,12,4,/5,7 (linkspolnische Notation)

postorder 3,12,*,4,+,5,7,/ (rechtspolnische Notation)

46. Lösung von Aufgabe ex:baumarray: Die Wurzel kommt auf $a[0]$. Für $i \geq 1$ werden die maximal d Kinder dann in $a[d * i], a[d * i + 1], \ldots, a[d * i + d - 1$. Umgekehrt werden die Elemente des Arrays entsprechend diesem Schema in den Baum „eingebaut“.

47. Mindestanzahl von Knoten eines AVL Baumes der Höhe h.

Wir beginnen mit einem AVL-Baum der Höhe $h = 1$. Dieser hat mindestens ein Blatt und maximal zwei. Die Mindestanzahl von Knoten ist also 2.

Ein AVL-Baum der Höhe $h = 2$ hat mindestens zwei und maximal vier Blätter, insgesamt also mindestens 5 Knoten.

Einen AVL-Baum mit minimaler Blatt- und Knotenzahl der Höhe $h + 2$ erhält man, in dem man einen AVL-Baum mit minimaler Knotenanzahl der Höhe h und der Höhe $h + 1$ zusammenfügt. Die Anzahl der Knoten dieses neuen AVL-Baumes ergibt sich durch Addition der Knotenanzahlen der Teilbäume.

48. Wie ist das asymptotische Verhalten von Insertion Sort im durchschnittlichen Fall (C_{avg}, M_{avg})? Machen Sie ggf. vereinfachende Annahmen, um zu einem zumindest unter diesen Annahmen gültigen Ergebnis zu kommen!

Lösungsvorschlag:

49. ([10]) Ist Insertion Sort stabil?

Lösungsvorschlag:

50. ([10]) Ist Selection Sort stabil?

Lösungsvorschlag:

51. ([15]) Don Knuth beschreibt in [Knu73b] *Algorithmus S* „straight selection sort“. Dabei wird zunächst der Datensatz mit dem größten Schlüssel selektiert, dann der mit dem zweitgrößten. Warum ist das „more convenient“?

52. ([10]) Ist *Algorithmus S* stabil?

53. ([25]) Untersuchen Sie, wann Selection Sort *besser* ist als Quicksort!

54. ([25]) Untersuchen Sie, wann Selection Sort und wann Insertion Sort theoretisch besser ist und wann praktisch!

55. ([25]) Untersuchen Sie das Verhalten von Insertion Sort, wenn Schlüsselwerte mehrfach vorkommen!

56. ([25]) Ermitteln Sie bitte $C_{min}, C_{avg}, C_{max}$ und $M_{min}, M_{avg}, M_{max}$ für Bubblesort.

57. ([10])Ist Bubblesort stabil?

58. (M20) Zeigen Sie, dass ein stabiler Sortieralgorithmus genau eine Permutation von n Elementen liefern kann.

    Beweis: Angenommen, es gäbe zwei mögliche, verschiedene korrekte Sortierfolgen $p_1, p_2, \ldots, p_n$ und $q_1, q_2, \ldots, q_n$. Sei nun $i := min\{i|p_i \neq q_i\}$. Dann gibt es j, k mit $p_i = q_j$ und $q_i = p_k$. Nach Definition von i ist $p_r = q_r \forall 1 \leq r < i$. also ist $p_i < p_k$

59. Diese Aufgabe besteht aus zwei Teilen:

    59.1. ([M45]) Beweisen Sie

    $$\pi = \frac{9801}{\sqrt{8}}(\sum_{n=0}^{\infty} \frac{(4n)!(1103 + 26390n)}{(n!)^4 396^{4n}})^{-1}$$

    59.2. ([20]) Ermitteln Sie die Aufwandsfunktion für die Ermittlung einer Näherung von π in sinnvollen Größen!

60. Ramanchandra Kaprekar (1905 - 1986) war einer der berühmtesten indischen Mathematiker. Er entdeckte eine Besonderheit der Ziffernfolge $1, 4, 6, 7$.

    Satz von Kaprekar:

    Man nehme eine beliebige vierziffrige Zahl (z. B.:2195), bei der nicht alle vier Ziffern gleich sind (also nicht $1111, 2222, 3333, \ldots$) und ordne die vier Ziffern der Größe nach absteigend (im Beispiel: 9521). Von der so entstehenden Zahl subtrahiere man die Zahl, die bei aufsteigender Anordnung der Ziffern entsteht (im Beispiel: 1259). Mit der entstandenen neuen Zahl (im Beispiel: $9521 - 1259 = 8262$) wird der Prozess wiederholt. Nach einigen Schritten landet man immer bei der Zahl 6174, die sich dann bei weiteren Schritten stets selbst reproduziert. Besonders kurz ist dieser Prozess für 2004. Finden Sie heraus, wie sich das für andere Startwerte entwickelt.

61. Die Binomialkoeffizienten n über k (siehe auch Pascalsches Dreieck) sind definiert als

    $$\binom{n}{k} := \frac{n!}{k!(n-k)!} \; \forall k, n \; \geq 0$$

    61.1. Untersuchen Sie verschiedene Algorithmen zur Berechnung der Binomialkoeffizienten darauf, für welche Werte n, k und insbesondere bis für welche Werte von $\binom{n}{k}$ sie ein korrektes Ergebnis für `int` bzw. `long` in Java bzw. C++ liefern.

    61.2. Ermitteln Sie, wie viele Rechenoperationen die verschiedenen Verfahren benötigen!

    61.3. Visualisieren Sie das Ergebnis!

    61.4. Können Sie das Ergebnis auf allgemeine 8, 16, 32, 64 Rechner verallgemeinern?
    Sie sollten nicht unbedingt endlos CPU-Zyklen Ihres Rechners verbrauchen, sondern lieber *vorher nachdenken.*

    Lösungsvorschlag: Ich beginne mit der Berechnung

    61.1. Ein ganz naiver Ansatz rechnet einfach die Formel aus:

Operand(en)	Formel	Operationen
$n!$	$1 \cdot 2 \cdot \ldots n$	$n-1$ Multiplikationen
$k!$	$1 \cdot 2 \cdot \ldots k$	$k-1$ Multiplikationen
$(n-k)!$	$1 \cdot 2 \cdot \ldots n-k$	$n-k-1$ Multiplikationen
$k! \cdot (n-k)!$		1 Multiplikation
$\frac{n!}{k! \cdot (n-k)!}$		1 Division
Summe		$2n-2$ Multiplikationen
		1 Division

kann man natürlich die drei Multiplikationen mit 1 einsparen und landet damit bei $2n-5$ Multiplikationen und einer Division.

Etwas mehr kann man einsparen, wenn man sieht, dass die Multiplikationen bis k bzw. $n-k$ nicht mehrfach vorgenommen werden müssen. Man muss dann nur noch $k < n-k$ prüfen und kann dann die Schleife zur Berechnung von $n!$ „unterbrechen“ und das Ergebnis zwischenspeichern. So reduziert sich der Aufwand zur Berechnung von $n!$,$k!$ und $(n-k)!$ auf $n-1$ Multiplikationen, so dass ri auf insgesamt n Multiplikationen und eine Division.

61.2. Man kann n über k unter Verwendung der Formel (z. B. [Rot60]

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

rekursiv berechnen.

Das geht allerdings auf Kosten der Laufzeit: Die Anzahl Aufrufe a(n,k) zur rekursiven Berechnung von $\binom{n}{k}$ ist:

$$a(n,k) = a(n-1,k) + a(n-1,k-1)$$

62. Wieviele Knoten N hat ein vollbesetzter Baum vom Grad d und Höhe h? Welche minimale Höhe h hat ein Baum vom Grad d mit N Knoten?

62.1. Auf jeder Ebene i hat ein solcher Baum d^i Knoten. Insgesamt also:

$$\begin{aligned} \text{Anzahl Knoten} &= \sum_{i=0}^{N} d^i \\ &= \frac{1-d^{N+1}}{1-d} \\ &= \frac{d^{N+1}-1}{d-1} \end{aligned}$$

62.2. Die Höhe ist minimal, wenn höchstens die unterste Ebene nicht voll besetzt ist. Aus der maximalen Knotenanzahl ergibt sich also:

$$\begin{aligned} \text{Höhe} &\leq \lfloor log_d \frac{(d^{N+1}-1)}{d-1} \rfloor \\ &= \lfloor log_d(d^{N+1}-1) - log_d(d-1) \rfloor \\ &\leq \lfloor log_d(d^{N+1}) \rfloor \end{aligned}$$

63. Gegeben die Zahlen von 1 bis 15 (beide einschließlich). Geben Sie eine Reihenfolge dieser Zahlen an, bei der der durch Einfügen dieser Zahlenfolge entstehende sortierte Binärbaum perfekt balanciert ist und zeichnen Sie den Baum auf. **Antwort:** Eine Idee zur Konstruktion einer solchen Folge ist diese: Die Schlüsselwerte im linken Teilbaum eines Knotens sind kleiner als dessen Schlüssel und die im rechten Teilbaum größer. Dies gilt auch für die Wurzel. Ich starte also bei der mittleren Zahl (8) der Folge und teile die Teilfolgen entsprechend weiter und füge immer abwechselnd von linke und von rechts ein, also:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
							8							
			4								12			
	2				6				10				14	
1		3		5		7		9		11		13		15

Ich füge also in dieser Reihenfolge ein:
8,4,12,2,6,10,14,1,3,5,7,9,11,13,15
und erhalte folgenden Baum

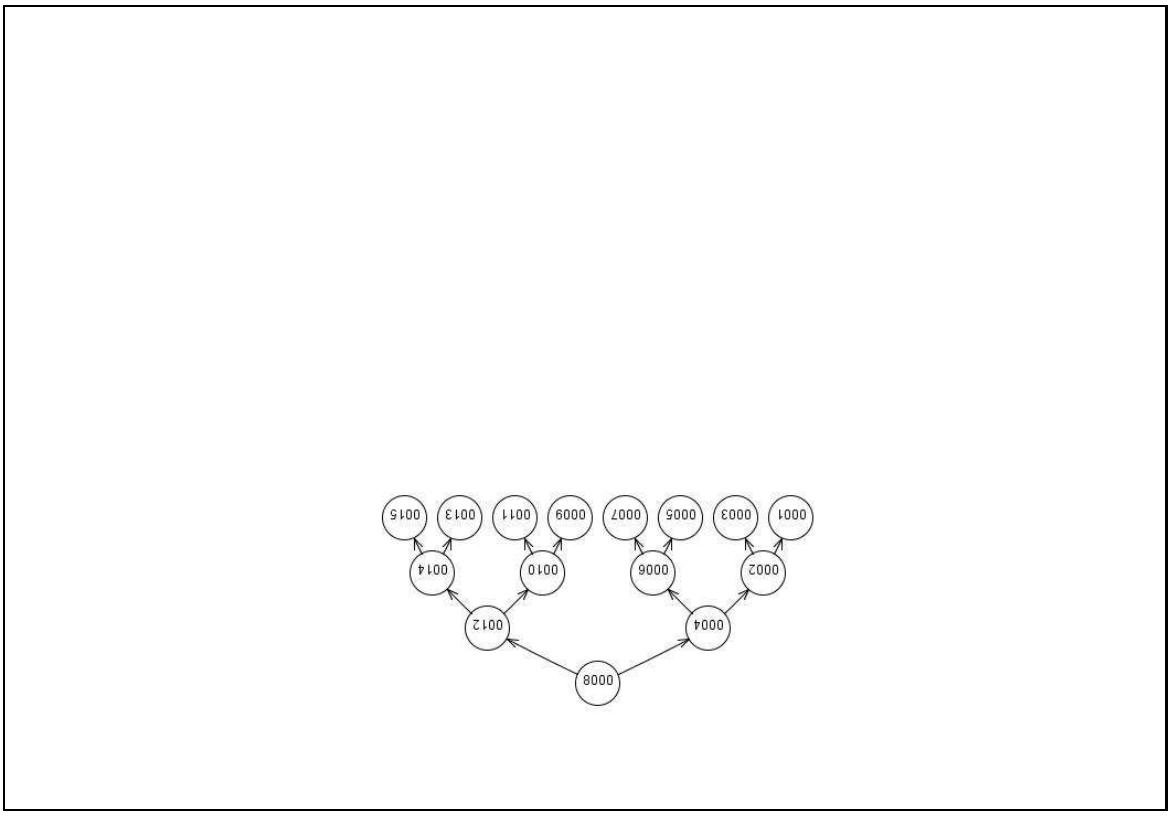


Abbildung B.4: Balancierter Suchbaum

64. Gegeben die Zahlen von 1 bis 15 (beide einschließlich). Geben Sie eine Reihenfolge dieser Zahlen an, bei der der durch Einfügen dieser Zahlenfolge entstehende sortierte Binärbaum zu einer Liste entartet! Erläutern Sie, wie Sie die Folge konstruiert haben!

    Antwort:

65. Gegeben folgende Frequenztabelle für eine Datei:

a	b	d	e	n	w
5	2	1	6	3	9

    65.1. Konstruieren Sie den Codebaum, den der Huffman-Algorithmus ergibt (die Zuordnung von „links“ und „rechts“ ist Ihnen freigestellt)

    65.2. Geben Sie die resultierende Codetabelle an (die Zuordnung von 0 und 1 ist Ihnen ebenfalls freigestellt).

    65.3. Codieren Sie die Nachricht „badewanne“!

66. $\forall n, m \in \mathbb{N}$ gilt:

$$\lfloor \frac{n+m}{2} \rfloor + \lfloor \frac{n-m+1}{2} \rfloor \quad = \quad \lfloor \frac{2n+1}{2} \rfloor$$

67. $\forall n, m \in \mathbb{N}$ gilt:

$$\lceil \frac{n+m}{2} \rceil + \lceil \frac{n-m+1}{2} \rceil \quad = \quad \lceil \frac{2n+1}{2} \rceil$$

68. Beweisen Sie bitte:

$$\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x+y \rfloor$$

Unter welchen Bedingungen gilt „=“?

69. Beweisen Sie bitte:

$$\lceil x \rceil + \lceil y \rceil \geq \lceil x+y \rceil$$

Unter welchen Bedingungen gilt „=“?

70. Geben Sie die ersten 40 binären Stellen von π an! Der Merkspruch um Rechenzeit einzusparen ist:

> But I, I *know* the octonal facts about pi.

Die liefert den Beginn der Darstellung von π im Oktalsystem, die man nun leicht dual hinschreiben kann

$$\begin{aligned} \pi_8 &= 3,11037552 \\ \pi_2 &= 11,00100100100001111110110101010001000100001011010000110000 \end{aligned}$$

71. Wieviele Knoten N hat ein vollbesetzter Baum vom Grad d und Höhe h? Welche minimale Höhe h hat ein Baum vom Grad d mit N Knoten?

Antworten:

71.1. Ich überlege mir das für ein $h = 0, 1, 2, \ldots$. Wenn alles gut geht habe ich dann sogar schon einen Induktionsbeweis.

h=0 Dann hat der Baum nur die Wurzel und somit $N = 1$ Knoten.

h=1 Wenn der Baum vollbesetzt ist, hat jeder Knoten außer den Blattknoten d direkte Nachfolger. Der Baum hat also $N = d + 1$ Knoten.

h beliebig aber fest

$$N = \sum_{k=0}^{h} d^k$$

Der Wert dieser geometrischen Reihe ist bekannt (z. B. [Rot60])

$$N = \sum_{k=0}^{h} d^k = \frac{d^{h+1} - 1}{d - 1}$$

Die minimale Höhe ergibt sich aus der vorstehenden Überlegung: Sie liegt vor, wenn der Baum vollständig besetzt ist oder höchstens auf der letzten Ebene Lücken vorliegen. Wie bei Binärbäumen ist die Höhe dann $h = \lfloor \log_d N \rfloor$

72. Arbeiten Sie die Lösch-Operation (vgl. S. 91 in einem binären Suchbaum so aus, dass Sie leicht und sicher implementieren können!

Antwort: „Im Prinzip“ steht alles auf S. 91 Ich arbeite mich vom einfachsten zum schwierigsten Fall vor und versuche daraus eine schlüssige Strategie abzuleiten. Zunächst muss der zu löschende Knoten gefunden werden. Dazu verwende ich eine Operation der Klasse Baum, die den Knoten findet und zurückliefert. Gibt es ihn nicht (return Wert **null**) so ist nichts zu tun. Andernfalls muss ich den Knoten löschen. Dazu nun die folgenden Überlegungen:

72.1. Ist der Knoten ein Blatt, so muss nur der Verweis vom Vater auf diesen auf **null** gesetzt werden. Dazu muss ich feststellen, ob es der linke oder der rechte Sohn ist. Das kann ich direkt abfragen oder die Suchbaumeigenschaft ausnutzen: Ist der Schlüssel des Vaters kleiner, so ist es der rechte Sohn, andernfalls der linke.

72.2. Hat der Knoten genau einen Sohn, so muss ich diesen „nach oben“ ziehen. Dieser bekommt einen neuen Vater, der Vater des Knotens einen neuen linken bzw. rechten Sohn.

72.3. Wenn der Knoten zwei Söhne hat, muss ich einen Weg finden, die Suchbaumeigenschaft zu erhalten: Alle Knoten im linken Teilbaum haben kleiner, alle im rechten größere Schlüssel. Auch der direkte Nachfolger s findet sich also im rechten Teilbaum. Dieser hat höchstens einen Sohn. Andernfalls wäre ja der Schlüssel im linken Sohn von s kleiner als der Schlüssel von s und größer als der Schlüssel des zu löschenden Knotens. Diesen rechten Sohn kann ich (so er vorhanden ist) also an den Vater von s hängen und den zu löschenden Knoten durch s ersetzen. Dazu muss ich dann die entsprechenden Referenzen wie gewohnt ändern.

73. Beweisen Sie bitte:

$$(a \cdot b \cdot c) \bmod m = ((((a \bmod m) \cdot (b \bmod m)) \bmod m) \cdot (c \bmod m)) \bmod m$$

!

Beweis: Ich zeige

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m.$$

Die Behauptung folgt dann z. B. mit $a' = a \cdot b$ durch Einsetzen.

Zunächst führe ich folgende Bezeichnungen ein:

$$\begin{aligned} a \cdot b \bmod m &= a \cdot b - f \cdot m \\ a \bmod m &= a - f_a \cdot m \\ b \bmod m &= b - f_b \cdot m \end{aligned}$$

Nun rechne ich wie folgt:

$$\begin{aligned} (a \bmod m) \cdot (b \bmod m) &= ((a - f_a m)(b - f_b m)) \\ &= (ab - bf_a - af_b m + f_a f_b m^2) \\ &= (ab - (f_a b + f_b a - f_a f_b m)m) \\ &= ab \end{aligned}$$

Es geht aber auch so:

$$\begin{aligned} (a \bmod m)(b \bmod m) \bmod m &= (a - \lfloor \frac{a}{m} \rfloor m)(b - \lfloor \frac{b}{m} \rfloor m) \bmod m \\ &= (ab - b\lfloor \frac{a}{m} \rfloor m - a\lfloor \frac{b}{m} \rfloor m - \lfloor \frac{a}{m} \rfloor \lfloor \frac{b}{m} \rfloor m^2)) \bmod m \\ &= (ab - (b\lfloor \frac{a}{m} \rfloor + a\lfloor \frac{b}{m} \rfloor - \lfloor \frac{a}{m} \rfloor \lfloor \frac{b}{m} \rfloor m)m) \bmod m \end{aligned}$$

74. Welches h_i bringt in Beispiel 5.6 auf Seite 130 die 99 auf einen freien Platz?

Hier zur Erinnerung der Stand nach dem Einfügen der ersten 5 Zahlen:

0	1	2	3	4	5	6
63	77	51	24		85	

Bei dem Versuch die 99 unterzubringen erhält man:

i	$(99 + i^2) mod\ 7$	$h_i(99)$
0	99 mod 7	1
1	(99 +1) mod 7	2
2	(99 +4) mod 7	5
3	(99 +9) mod 7	3
4	(99 +16) mod 7	3
5	(99 +25) mod 7	5
6	(99 +36) mod 7	2
7	(99 +49) mod 7	1
8	(99 +64) mod 7	2
9	(99 +81) mod 7	5
10	(99 +100) mod 7	3
11	(99 +121) mod 7	3
12	(99 +144) mod 7	5
13	(99 +169) mod 7	2
14	(99 +196) mod 7	1

Wir erkennen hier einen Zyklus: 1, 2, 5, 3, 3, 5, 2, 1, der sich von nun an immer wiederholt. Woran liegt das?

75. Was passiert genau, wenn man als Hash-Funktion

$$h(k) = k \bmod 2^m$$

$m > 0$ wählt?

Antwort:

76. Ist eine Hash-Funktion der Form

$$h(k) = k \bmod 3 \cdot m$$

zum Hashen von alphabetischen Schlüsseln sinnvoll?

Antwort:

77. Unter welchen Bedingungen ist eine Hash-Funktion der Form

$$h(k) = k \bmod 2 \cdot m$$

sinnvoll und unter welchen nicht?

Antwort:

78. Zeigen Sie, wie ein Stack mittels zweier Queues implementiert werden kann! Analysieren Sie das asymptotische Verhalten dieser Stack-Implementierung!

    Ich definiere den Stack mit den üblichen Operationen push und pop.

79. Zeigen Sie, wie eine Queue mittels zweier Stacks implementiert werden kann! Analysieren Sie das asymptotische Verhalten dieser Stack-Implementierung!

80. Eine Freundin rief mich am Tag vor dem Schuljahresbeginn 2009 an und bat um Hilfe bei folgendem Problem:

    > Es gibt fünf neue Parallelklassen a, b, c, d, e. Jede von diesen hat einen Klassenraum Ra, Rb, Rc, Rd, Re. Damit sich die Schüler und Schülerinnen über Klassengrenzen hinweg und auch die verschiedenen Klassenlehrer kennenlernen, sollen

die Schüler für fünf Projekte an fünf Terminen gemischt werden. Jede Klasse wird in fünf feste (Farb)Gruppen aufgeteilt (weiß, gelb, grün, blau, rot). Jedes Projekt findet in einem der Klassenräume statt. Die Gruppen sollen nun so für die fünf Termine gemischt werden, dass folgende Bedingungen erfüllt werden:

80.1. Keine Farbgruppen aus einer Parallelklasse in einem Projekttermin und Raum.

80.2. Keine Farbgruppe mehrfach in einem der Räume.

80.3. Keine Farbgruppe mehrfach mit einer anderen in einem Projekt.

Entwickeln Sie einen Algorithmus, um festzustellen, ob dieses Problem eine Lösung hat und ggf. alle Lösungen findet.

Lösung: Eine systematische Lösung könnte der Algorithmus der Dancing links liefern, der in Abschn. 7.2.2.1 von TAOCP beschrieben werden soll.

81. Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod\ 9$ in eine Hashtabelle der Größe $N = 9$ (nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge durch quadratisches Sondieren ($h_i(k) = (h(k) + i^2)$ mod 9).

Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h bzw. bei Kollisionen der Hashfunktionen h_i an.

Lösung:

82. Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod\ 9$ in eine Hashtabelle der Größe $N = 9$ (nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge Verkettung.

Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h bzw. bei Kollisionen der Hashfunktionen h_i an.

Lösung: Es ergibt sich folgende Tabelle. Die linke Spalte enthält die Indizes, die weiteren die eingetragenen Schlüssel.

Index	Schlüsselliste
0	9,36
1	
2	11
3	
4	40,22
5	14,5
6	
7	43
8	26

83. Bereits in der vorstehenden Aufgabe wurde an die Definition der Fibonacci-Zahlen erinnert. Beweisen Sie bitte die folgende Aussage über die Fibonacci-Zahlen:

$$F_{n+1} \cdot F_{n-1} - F_n^2 = (-1)^n \ \forall\ n > 0.$$

Beweis (mit vollst. Induktion:

Induktionsanfang: Für $n = 1$ ist $F_2 \cdot F_0 - F_1^2 = 1 \cdot 0 - 1 = -1 = (-1)^1$

Induktionsannahme: Die Aussage sei bereits für alle $k \leq n$ bewiesen:

$$F_{k+1} \cdot F_{k-1} - F_k^2 = (-1)^k \; \forall \; 0 < k \leq n.$$

Induktionsschluss:

$$\begin{aligned}
F_{n+2} \cdot F_n - F_{n+1}^2 &= (F_{n+1} + F_n) \cdot F_n - F_{n+1}^2, \text{nach Definition} \\
&= F_{n+1} \cdot F_n + F_n^2 - F_{n+1}^2 \\
&= F_{n+1}(F_n - F_{n+1}) + F_n^2 \\
&= F_{n+1}(F_n - F_n - F_{n-1}) + F_n^2, \text{nach Definition} \\
&= F_{n+1}(-F_{n-1}) + F_n^2 \\
&= -F_{n+1} \cdot F_{n-1} + F_n^2 \\
&= -(F_{n+1} \cdot F_{n-1} - F_n^2 \\
&= -(-1)^n, \text{nach Iduktionsannahme} \\
&= (-1)^{n+1}
\end{aligned}$$

Literaturverzeichnis

[AKS04] Manindra Agrawal, Neeraj Kayal und Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[Ale03] Andrei Alexandrescu. *Modernes C++ Design. Generische Programmierung und Entwurfsmuster angewendet.* mitp-Verlag, Bonn, 2003, 424 Seiten. ISBN 3-8266-1347-3.

[Bay72] Rudolf Bayer. Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.

[BD02] Manfred Broy und Ernst Denert, Hrsg. *Software Pioneers. Contributions to Software Engineering.* Springer-Verlag, Berlin, Heidelberg, New York, NY, 2002, 727 Seiten. ISBN 3-540-43081-4. 4 DVDs.

[BM72] Rudolf R. Bayer und M. McCreight, E.˙ Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972. Diese Arbeit wurde in [BD02] nachgedruckt. Der dortige Beitrag von Rudolf Bayer enthält viele Informationen zur Entstehungsgeschichte.

[CFR05] P. Cull, M. Flahive und R. Robson. *Difference Equations. From Rabbits to Chaos.* Springer-Verlag, Berlin, Heidelberg, New York, NY, 2005, xiii+392 Seiten. ISBN 0-387-23233-8. Auch als Softcover-Ausgabe erhältlich, isbn 0-387-23234-6, Math 230.55.

[CLR94] Thomas H. Cormen, Charles E. Leiserson und Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1. Auflage, 14. Druck, 1994, 1028 Seiten. ISBN 0-262-53091-0. Es gibt wohl wenige Informatik Bücher, die es in kaum 5 Jahren auf 14 Drucke bringen! Mein erster Eindruck ist hervorragend. Natürlich in LaTeX gesetzt! Seit Ende 2004 gibt es eine Deutsche Übersetzung, die ich mir aber nicht angesehen habe.

[CLRS07] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Algorithmen — Eine Einführung.* Oldenbourg Wissenschaftsverlag, München, 2. korrigierte Auflage, 2007, xxii+1188 Seiten.

[Dat98] Hugh Date, Chris J.and Darwen. *SQL - Der Standard SQL/92 mit den Erweiterungen CLI und PSM.* Addison Wesley Longman, Reading, MA, 1998.

[Fel50] William Feller. *An Introduction to Probability Theory and its Applications.* Wiley, New York, Chichester, Brisbane, Toronto, Singapore, 3. Auflage, 1950, xviii+509 Seiten.

[GD04] Ralf Hartmut Güting und Stefan Dieker. *Datenstrukturen und Algorithmen.* Leitfäden der Informatik. Teubner, Wiesbaden, 3. neu bearb. Auflage, 2004, xvi+377 Seiten.

[GZ06] Gerhard Goos und Wolf Zimmermann. *Vorlesungen über Informatik, Band 1: Grundlagen und funktionales Programmieren.* Springer-Verlag, Berlin, Heidelberg, New York, NY, 4. überarbeitete Auflage, 2006, xiii+400 Seiten. ISBN 3-540-24405-0.

[Hoa62] Charles Antony Richard Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[Jae81] G. Jaeschke. Reciprocal hashing: a method for generating minimal perfect hashing functions. *Communications of the ACM*, 24(12):829–833, Dezember 1981.

[Knu73a] Donald E. Knuth. *The Art of Computer Programming I: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 2. Auflage, 1973, xix+634 Seiten. ISBN 0-201-03821-8. Ein Klassiker, den jeder Informatiker gelesen haben sollte. Der Stil bis hin zu den Aufgaben ist legendär. Dieser Band ist im Gegensatz zu den weiteren auch als Paperback lieferbar.

[Knu73b] Donald E. Knuth. *The Art of Computer Programming III, Sorting.* Addison-Wesley, Reading, MA, 1. Auflage, 1973. ISBN 0-201-03803-X.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming I: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 3. Auflage, 1997, xix+650 Seiten. ISBN 0-201-89683-4. Neue, erheblich überarbeitete Auflage von [Knu73a].

[Knu97b] Donald E. Knuth. *The Art of Computer Programming II, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 3. Auflage, 1997. ISBN 0-201-89684-2. Die letzte Revision vor Fertigstellung der auf den dritten folgenden Bände.

[Knu97c] Donald E. Knuth. *The Art of Computer Programming III, Sorting.* Addison-Wesley, Reading, MA, 2. Auflage, 1997. ISBN 0-201-89685-0. Die letzte Revision vor Fertigstellung der auf den dritten folgenden Bände.

[Knu07] Donald Ervin Knuth. *The Art of Computer Programming Volume 4, pre-Fascicle 0B. A Draft of Section 7.1.2: Boolean Evaluation.* Addison-Wesley, 2007, v+61 Seiten. Zeroth printing (revision 7), 20 May 2007. www-cs-faculty.stanford.edu/ knuth/, seit dem 01.05.2008 nicht mehr online Verfügbar.

[Knu08a] Donald Ervin Knuth. *The Art of Computer Programming Volume 4, pre-Fascicle 0A. A Draft of Section 7: Introduction to Combinatorial Searching.* Addison-Wesley, 2008, xv+74 Seiten. Zeroth printing (revision 3), 30 January 2008. www-cs-faculty.stanford.edu/ knuth/, seit dem 01.05.2008 nicht mehr online Verfügbar.

[Knu08b] Donald Ervin Knuth. *The Art of Computer Programming Volume 4, pre-Fascicle 0C. A Draft of Section 7.1.1: Boolean Basics.* Addison-Wesley, 2008, iv+84 Seiten. Zeroth printing (revision 13), 28 April 2007. www-cs-faculty.stanford.edu/ knuth/, seit dem 01.05.2008 nicht mehr online Verfügbar.

[Knuff] Donald E. Knuth. *The Art Of Computer Programming.* Addison-Wesley, Reading, MA, 1973ff. Das Lebenswerk von Don Knuth. Ich hoffe er bleibt solange gesund bis er es abgeschlossen hat und darüber hinaus! Ich erinnere mich aber an die Besprechung von Jerrold E. Marsden[Mar80] von Jean Dieudonnés *Foundations of Analysis*: This... illustrates a basic theorem in mathematical writing which even experts find hard to swallow: *to find the true length (measured in pages or years) of a writing project multiply your initial estimate by a factor of at least three.* In fact, if one takes a project with large enough scope, the writing process can become stationary: *even though you write feverishly, your project is at all times half-completed.* Dieudonnés position is probably almost as bad as the theorem indicates and could even be as bad as stationary, ... Diese Beschreibung könnte auch auf dieses Werk zutreffen.

[Mar80] Jerrold E. Marsden. Book Review: Treatise on analysis. *Bulletin of the American Mathematical Society*, 3(1, July 1980):719–724, 1980.

[Mes71] Herbert Meschkowski. *Mathematisches Begriffswörterbuch.* Nr. 99 in Hochschultaschenbücher. Bibliographisches Institut, Mannheim, Wien, Zürich, 3. erweiterte Auflage, 1971. Inzwischen natürlich etwas antik. Ich benutze es aber immer noch, wenn ich etwas mathematisches nachschlagen möchte, über das ich keine Spezialliteratur habe.

[Mey88a] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mey88b] Bertrand Meyer. *Objektorientierte Software Entwicklung*. Hanser, Prentice-Hall, München, Wien, London, 1988. Deutsche Übersetzung von [Mey88a]. Einer der Klassiker über objektorientierte Entwicklung. Stark mit Blick auf EIFFEL geschrieben.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, NJ, 2. Auflage, 1997. ISBN 0-13-629155-4. Eine dramatisch erweiterte Ausgabe von [Mey88b].

[MW87] Alan Mitchell und John Wilkinson. *Pareys Buch der Bäume*. Verlag Paul Parey, Hamburg, Berlin, 2. durchgesehene Auflage, 1987, 271 Seiten. ISBN 3-490-19518-3.

[OW02] Thomas Ottmann und Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 4. Auflage, 2002, xix+716 Seiten. ISBN 3-8274-1029-0.

[Rad70] Charles E. Radke. The use of quadratic residue research. *Communications of the ACM*, 13(2):103–107, Februar 1970.

[Rom86] Heinrich Rommelfanger. *Differenzengleichungen*. Bibliographisches Institut, Mannheim, Wien, Zürich, 1986, 156 Seiten. ISBN 3-411-03136-0. Math 230.18.

[Rot60] Karl Rottmann. *Mathematische Formelsammlung*, Band 13 von *Hochschultaschenbücher*. Bibliographisches Institut, Mannheim, Wien, Zürich, 2. Auflage, 1960, 176 Seiten. ISBN 3-411-00013-9.

[Sed92] Robert Sedgewick. *Algorithmen in C++*. Addison-Wesley, Bonn, Paris, Reading, MA, 1992.

[Sed02] Robert Sedgewick. *Algorithms in Java - Parts 1-4. Fundamentals / Data Structures / Sorting / Searching*. Addison-Wesley, Reading, MA, 3. Auflage, 2002, xix+737 Seiten. ISBN 0-201-36120-5.

[She59] Donald L. Shell. A High-Speed Sorting Procedure. *Communications of the ACM*, 2(7), Juli 1959.

[Sin00] Simon Singh. *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet*. Carl Hanser Verlag, München, Wien, 2000, 475 Seiten. ISBN 3-446-19873-3.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert und Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2. Patterns for Concurrent and Networked Objects*. Wiley, New York, Chichester, Brisbane, Toronto, Singapore, 2000, xxix+633 Seiten. ISBN 0-471-60695-2.

[Wan02] Reinhard Wandtner. Stielblüte mit Eichlaub. *FAZ*, S. 9, 12.04.2002.

[WK99] Jos B. Warmer und Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, Reading, MA, 1999.

Index