Algorithmen und Datenstrukturen

3. Auflage 2013

Inhaltsverzeichnis

Kapitel 1

Grundlegendes

Dieses Skript versteht sich als Stichwortsammlung des in der Vorlesung präsentierten Stoffes und hat wenigstens im Moment keinen Anspruch, ein vollständiges Skript zu sein. Es genügt daher normalerweise nicht alleine zur Prüfungsvorbereitung oder als Nachschlagewerk! Es sollte sich deshalb auf jeden Fall zumindest mit der aufgeführten Basis-Literatur beschäftigt werden und sich von Zeit zu Zeit auch weiterführende Literatur und aktuelle Zeitschriftenartikel angeschaut werden.

Bemerkung zur Originalität: Dieses Skript ist zum großen Teil aus Skripten anderer Kollegen (auch anderer Hochschulen) und Büchern zusammengestellt! Dieses Skript erhebt daher kein Anspruch auf Originalität, sondern versteht sich als Vorlesungsmitschrift.

1.1 Zur Vorlesung

1.1.1 Einordnung der Vorlesung

Die Vorlesungen der Informatik lassen sich (im Groben) in folgende Lehrgebiete einteilen:

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.

1.1.2 Lernziele

Sie sollen in dieser Veranstaltung Kenntnisse zum selbstständigen Entwurf, selbstständiger Analyse und Anwendung von Algorithmen erwerben und die dazu erforderlichen Datenstrukturen kennen und einsetzen lernen. Insbesondere sollen Sie am Ende des Semester diese Fähigkeiten erworben haben:

Kenntnisse : Wissen reproduzieren können; z.B. Klarheit der Begriffe in der Klausur und Gesprächen im Praktikum

Verständnis : Wissen erläutern können; z.B. Gespräch/Test im Praktikum

Anwendung : Wissen anwenden können; z.B. Freiräume der Praktikumsaufgaben und neue Klausuraufgaben

Analyse : Zusammenhänge analysieren können; z.B. Auswerten der Beobachtungen im Praktikum

Synthese : eigene Problemlösestrategien angeben können; z.B. im Entwurf und Aufgaben in der Klausur

Beurteilung : eigene Problemlösestrategien beurteilen können; z.B. Gespräch im Praktikum und Aufgaben in der Klausur

1.1.3 Praxisbezug

Wer an der Hochschule für Angewandte Wissenschaften (HAW) Hamburg studiert, sollte eigentlich schon wissen, was mit Praxisbezug der HAW's bzw. (ehemaligen) Fachhochschulen gemeint ist. Hier dennoch dazu ein paar klärende Worte.

Ausbildung an einer Universität (oder technischen Universität, TU)

Ausbildungsziel : wissenschaftlichen Nachwuchs ausbilden.

Forschungsziel : Erstellen neues Weltwissens.

Übungen : Benötigen Übung darin, wie man solches neues Weltwissen erstellen und prüfen kann! Die Übungen hier: Trainieren des Entwickelns und Testens Neuen Weltwissens, z.B. Beweisverfahren.

Benutzbar : Frage nach Brauchbarkeit für die Wirtschaft ist zunächst irrelevant!

Ergebnis : Mögliches Ergebnis: Neue Erkenntnis

Ausbildung an der HAW (oder Fachhochschule)

Ausbildungsziel : Ausbildung von Ingenieueren.

Forschungsziel : Erstellen neues Praxiswissen.

Übungen : Benötigen Übung darin, wie man neue Erkenntnisse in die Praxis überführen kann. Die Übungen hier: Trainieren des Transfers und Sammlung praktischer Erfahrungen, z.B. SE.

Benutzbar : Frage nach Brauchbarkeit sehr relevant!

Ergebnis : Mögliches Ergebnis: Neue Designpattern (evtl. formal geprüft!)

Die AbsolventInnen der HAW sollen also die für die Praxis relevanten bzw. inzwischen brauchbaren Teile des Neuen Weltwissens in die tägliche Arbeit einbinden können.

Beide Gebiete haben Wechselwirkung: praktische Erfahrungen können wissenschaftliche Forschung auslösen und neue Forschung kann praktische Erfahrungen absichern. Von daher läßt sich diese grundlegende Sichtweise so klar in der Realität nicht trennen.

Für diese Vorlesung äußert sich dies so: es wird gelehrt, was bei der Konstruktion von Algorithmen zu beachten ist. Dies wird am Beispiel von für InformatikerInnen wichtigen Basisalgorithmen vorgenommen. Formale Verfikation der Algorithmen entfällt. Theoretische Grundlagen, etwa „die Kunst des Zählens“ (Kombinatorik) wird in anderen Vorlesungen vermittelt. Die Komplexitätstheorie wird soweit vermittelt, das die Einordnungen verstanden werden können und ggf. eine grobe (aber nicht formal bewiesene) Abschätzung eines eignen Algorithmus vorgenommen werden kann.

1.1.4 Literatur

Dieses Fach deckt „klassische Themen“ der Ausbildung von InformatikerInnen ab. Es gibt daher viele Lehrbücher und Skripte, die dieses Themengebiet mehr oder weniger verständlich vorstellen. Die meisten Lehrbücher – insbesondere die Besten – wurden vor langer Zeit geschrieben (etwa **der Klassiker** D.E. Knuth: *The Art of Computer Programming*, Volume 1 - 3, Addison-Wesley, 1968; 1969; 1973). In den neueren Ausgaben wurden oft aktuelle Programmiersprachen (etwa Java) aufgenommen, Inhalte an die aktuellste Forschung angepasst und eine verständlichere Darstellung angestrebt.

Dieses Skript steckt inhaltlich nur den in der Vorlesung vorgestellten Teil ab und es ist nicht geplant, dies zu einem allgemeinen Lehrbuch auszuweiten. Es empfiehlt sich daher, eines der aufgeführten Lehrbücher ergänzend zu bearbeiten.

`Aho` .....

A.V. Aho, J. Hopcroft, J.D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley

: ,

Mehlhorn .....
Herbert Klaeren und Michael Sperber
Die Macht der Abstraktion: Einführung in die Programmierung.
B.G. Teubner Verlag

Mehlhorn .....
Kurt Mehlhorn.
Datenstrukturen und Algorithmen.
Band 1 und 2, B.G. Teubner Verlag

Ottmann .....
Thomas Ottmann und Peter Widmayer.
Algorithmen und Datenstrukturen.
Spektrum Akademischer Verlag

Owsnicki .....
Bernd Owsnicki-Klewe.
Algorithmen und Datenstrukturen.
Wißner Verlag

Pareigis .....
Stephan Pareigis.
Algorithmen und Datenstrukturen für Technische Informatiker.
Skript der HAW

Pomberger .....
Gustav Pomberger und Heinz Dobler.
Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung.
Pearson Studium

Sedgewick .....
Robert Sedgewick.
Algorithmen in Java.
Pearson Studium

Solymosi .....
Andreas Solymosi und Ulrich Grude.
Grundkurs Algorithmen und Datenstrukturen in Java.
Vieweg Verlag

1.2 Der Begriff des „Algorithmus“

Informatik ist ein „Kunstwort“ aus den 60ern (Informatik: Information + Technik; oder Informatik: Information + Mathematik). Es war beabsichtigt, einen

Gegensatz zur amerikanischen Bezeichnung „Computer Science“ zu finden: nicht nur auf Computer beschränktes Gebiet. Erweiterungen sind Theoretische, Praktische, Angewandte, Technische Informatik oder auch „Bindestrich-Informatiken“ Informatik hat zentral zu tun mit der systematischen Verarbeitung von Informationen und „Maschinen“, die diese Verarbeitung automatisch leisten (! Computer). Die systematische Verarbeitung wird durch den Begriff Algorithmus präzisiert, und die Information durch den Begriff Daten (hier: maschinenunabhängige Darstellung).

Der Begriff **Algorithmus** ist abgeleitet aus dem Namen *Abu Ja'far Muhammed ibn Musa al Khowarizmi* (persischer Mathematiker, 780-850 n.Chr.). Auf ihn geht die Idee zurück, mathematische Probleme systematisch dadurch zu lösen, dass man eine bestimmte Folge von Anweisungen „blind“ ausführt. Für die Informatik hat der Algorithmus die Bedeutung

- ist eine formale Handlungsvorschrift zur Lösung von Instanzen eines Problems in endlich vielen Schritten.
- einer eindeutigen Vorschrift zur Lösung eines Problems mit Hilfe eines Computers: Die Vorschrift bildet einen Eingangsdatenbereich auf einen Ausgangsdatenbereich ab;
- sind eindeutige Anweisungen für schnelle (aber dumme) Prozessoren (Hochgeschwindigkeitstrottel), die auf einer mathematisch/formalen Grundlage basieren (Rechner „versteht“ nur Bits)
- Ein Algorithmus ist eine vollständige, präzise und in einer Notation oder Sprache mit exakter Definition abgefasste, endliche Beschreibung eines schrittweisen Problemlösungsverfahrens zur Ermittlung gesuchter Datenobjekte (ihre Werte) aus gegebenen Werten von Datenobjekten, in dem jeder Schritt aus einer Anzahl ausführbarer, eindeutiger Aktionen und einer Angabe über den nächsten Schritt besteht.

  Ein Algorithmus heißt terminierend, wenn er die Werte der gesuchten Datenobjekte in endlich vielen Schritten liefert, andernfalls heißt der Algorithmus nicht terminierend.
- ....

Eine präzise Definition dieses Begriffes vermeidet man gerne, da dadurch seine Bedeutung willkürlich eingeschränkt würde. Deshalb soll dieser Begriff hier nicht definiert werden, sondern nur erläutert werden. Als „weicher“ Begriff mit großem Bedeutungsumfang wird er uns nützlicher sein.

Zu nennen sind die Anforderungen an ein Verfahren, das den Namen „Algorithmus“ verdient:

Eindeutigkeit : Jeder elementare Schritt ist unmißverständlich beschrieben und läßt keine Wahlmöglichkeit offen.

Endlichkeit : Die Beschreibung des Algorithmus hat eine endliche Länge (durch Schleifen o.Ä. kann das ablaufende Programm allerdings durchaus unendlich lange brauchen, wenn der Algorithmus fehlerhaft ist).

Terminiertheit : Ein aktuelles Programm, das diesen Algorithmus implementiert, soll in endlicher Zeit beendet sein.

Reproduzierbarkeit : Der Algorithmus soll unabhängig von einer speziellen Implementation sein. Verschiedene (korrekte) Implementationen sollen dasselbe Ergebnis liefern,

Folgende Eigenschaften kann man u.U. beobachten:

- Liefert der Algoritmus 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.

Ein kleiner historischer Rückblick auf die Algorithmen:

- 300 v.Chr.: Euklids Algorithmus zur Bestimmung des ggT, (7. Buch der Elemente): ggT(300; 200) = 100
- 800 n.Chr.: Abu Ja'far Muhammed ibn Musa al Khowarizmi: Aufgabensammlung für Kaufleute und Testamentsvollstrecker (lat.: Liber Algorithmi, Kunstwort aus dem Namen und griechisch „arithmos“ für Zahl)
- 1574: Adam Rieses Rechenbuch
- 1614 Logarithmentafeln (30 Jahre für Berechnung!)
- 1703 Binäres Zahlensystem (Leibnitz)
- 1931 Gödels Unvollständigkeitssatz: Unvollständigkeit der Arithmetik, Unmöglichkeit eines Beweises der Widerspruchsfreiheit der Arithmetik mit arithmetisierbaren Mitteln - macht Aussagen über formale Systeme, genauer über das, was nicht in ihnen ausdrückbar ist.
- 1936 Church'sche These: Die Klasse der intuitiv berechenbaren Funktionen ist gleich der Klasse der Turing-berechenbaren Funktionen.
- danach Ausbau der Algorithmentheorie

Effiziente Algorithmen und Datenstrukturen sind ein zentrales Thema der Informatik. Man macht sich leicht klar, dass ein enger Zusammenhang besteht zwischen der Organisation von Daten (ihrer Strukturierung) und dem Entwurf von Algorithmen, die diese Daten bearbeiten. Für unsere Zwecke sollte man daher auch den Begriff der **Datenstruktur** charakterisieren. Wir verstehen darunter eine Organisation von Daten basierend auf einfachen Datentypen.

1. eine problemgemäße Abbildung der Daten in eine formale rechnergemäße Struktur liefert, und

2. die geeignet ist zur Bearbeitung mit einem bestimmten Algorithmus.

Ein Algorithmus benötigt in der Praxis eine passende von mehreren möglichen Datenstrukturen, um optimal laufen zu können.

Typische Fragestellungen in diesem Gebiet sind:

- Notation für Beschreibung
- Ausdrucksfähigkeit (man vergleiche Notationen der Bienensprache, dressierte Hunde etc.)
- Korrektheit, Genauigkeit, Eindeutigkeit
- Zeitbedarf, Geschwindigkeit

1.3 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.

Kapitel 2

Abstrakte Datentypen

In der Informatik werden - wie in der Mathematik auch - Objekte anhand ihrer Eigenschaften definiert. Spricht man in der Mathematik zum Beispiel von einem Vektor, so meint man ein Objekt, welches man zu einem anderen Vektor addieren kann, oder den man mit einem Skalar multiplizieren kann. Dabei interessiert man sich zunächst nicht dafür, ob es sich im speziellen um Pfeile in der Zeichenebene handelt, oder um Größen, die in einem höher-dimensionalen Raum eine bestimme physikalische Bedeutung haben. Im mathematischen Sinne ist dies eine Algebra (Wertemenge + Operationen darauf). Die Wertemengen können dabei mit Datenstrukturen assoziiert werden und die zugehörigen Operationen mit Algorithmen. Diese Vorlesung könnte also für MathematikerInnen verständlicher in „Algebren der Informatik“ umbenannt werden, in der bekannte Algebren vorgestellt werden und gelehrt wird, wie man selbst Algebren aufbauen kann. Sofern dies zum Verständnis des vermittelnden Stoffs beiträgt, werden wir ggf. auf diese abstrakte Betrachtungsweise zurückgreifen.

Definition 2.1 *Ein Datentyp besteht aus einer Objektmenge (auch Wertebereich genannt) und Operationen, die darauf definiert sind. Die Operationen werden auch Funktionen oder Methoden genannt.*

Nach dem Prinzip der kompositionalen Semantik gibt es atomare Elemente (Bausteine), die aus dieser Sichtweise nicht mehr zerteilt werden können, soll bedeuten, deren Inneres unbekannt bleibt und komplexe Elemente (Gebilde). Die komplexen Elemente werden aus den atomaren Elementen nach bestimmten Regeln zusammengebaut. So sind etwa die in einer Programmiersprache vorgegebenen Datentypen atomare Elemente, aus denen mittels den syntaktischen Regeln der Programmiersprache komplexere Datentypen erstellt/implementiert werden können. Diese komplexen Elemente können für andere Nutzer wieder als atomar angesehen werden!

2.1 Atomare (elementare) Datentypen

Als atomare Datentypen betrachten wir hier solche, die in einer konkreten Programmiersprache bereits vorhanden sind. Dazu einige Beispiele:

Beispiel 2.1

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

Die Operation + ist auf den Ganzen Zahlen als Objektmenge definiert, und hat dabei folgendes Abbildungsverhalten:

$$\begin{aligned} Operation + : \ int \times int &\rightarrow int \\ (a,b) &\mapsto a+b \end{aligned}$$

Gemeint ist hier eine Abbildung, welche das kartesische Produkt - daher das $\times$ - der ganzen Zahlen mit den ganzen Zahlen auf die ganzen Zahlen abbildet. Anders ausgedrückt: es sind zwei Eingabeparameter erforderlich, welche beide vom Typ `int` *sein sollen. Es gibt einen Ausgabeparameter, der wiederum vom Typ* `int` *ist.*

Bemerkung 2.1 *Gleich am Anfang ist ein Bezug zur mathematischen Betrachtungsweise vorgenommen worden. Das der Stoff dieser Vorlesung nicht ganz in dieser Betrachtungsweise vorgenommen wird, liegt an einem wesentlichen Unterschied in der betrachteten Welt: In der Informatik geht man zunächst von endlichen Welten aus, auch wenn unendliche Welten (z.B. unendliche Mengen, Endlosschleifen etc.) behandelt werden können. In der Mathematik ist zunächst alles unendlich groß. Dies sieht man z.B. an obiger Operation +: diese wird meist ohne weitere Prüfung von InformatikerInnen eingesetzt, ohne sich zu vergegenwärtigen, dass sie nicht abgeschlossen ist, d.h. es gibt zwei zugelassene Eingabeparameter, hier mal als $MAXINT$ und $IrgendeinINT$ bezeichnet, die keine Ausgabe erzeugen. Was heißt z.B. keine Ausgabe (im mathematischen Sinne also undefiniert) ? Wenn man in der Informatik Glück hat, wird von der Laufzeitumgebung eine Fehlermeldung generiert, d.h. das Programm bleibt in einem definierten Zustand (wenn das Fehlerauslösende Programm den gemeldeten Fehler verarbeiten kann), wenn man kein Glück hat, endet das Programm in einem „undefinierten“ Zustand, indem z.B. $MAXINT$ Ausgabewert ist oder gar die Laufzeitumgebung abstürzt (beliebte Angriffsziele von Hackern).*

Definition 2.2 *Das Abbildungsverhalten einer Operation nennen wir Signatur*[1]*.*

[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 18.

Als Schreibweise verwenden wir:

Operationsbezeichnung : *Menge der Inputparameter* $\rightarrow$ *Menge der Outputparameter*

Die Menge der Input/Ouputparameter kürzen wir durch die Typenbezeichnung ab. Wird ein Objekt Meintyp erzeugt oder zerstört, so schreiben wir

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

Bemerkung 2.2 *Ein weiterer Unterschied in der Begriffswelt zur Mathematik: in vorangegangener Definition wird eine Operation beschrieben, die im Definitionsbereich und Wertebereich eine leere Menge zulässt. Diese Operation ist keine mathematische Abbildung, da die Eigenschaft linkstotal (Wertebereich leer) und rechtseindeutig (Definitionsbereich leer) verletzt sind. Es handelt sich also um Relationen. Dies ist ein bekanntes Phänomen in der Informatik: Ausnahmen (wie hier) werden oft nicht deutlich gemacht, d.h. diese Operation würde man auch als Funktion bezeichnen. Sollten die Ausnahmen überhand nehmen, etwa mehr als 20% ausmachen, so würde man z.B. zu einer relationalen Programmiersprache (z.B. Prolog) wechseln, d.h. eine Sprache, die in erster Linie nur Relationen kennt, in der dann Abbildungen als solche (im Sinne der Ausnahme) auch nicht besonders ausgezeichnet sind. Ähnliches gibt es im Objektorientierten Bereich: auch hier wird manches als Objekt bezeichnet, was streng genommen keines ist bzw. es sich zumindest drüber diskutieren lässt.*

Weitere Beispiele.

Beispiel 2.2

*Operation **	:	*int*	$\times$	*int*	$\rightarrow$	*int*
Operation ==	:	*int*	$\times$	*int*	$\rightarrow$	*bool*
Operation ++	:			*int*	$\rightarrow$	*int*

Eine Signatur beschreibt einen syntaktischen Aspekt einer Operation. Dies bedeutet, der formale Charakter der Operation wird beschrieben.

Aufgabe 2.1 *Schreiben Sie die Signaturen für folgende Operationen des Typs int auf:*

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

2.2 ADT – (Daten-)Abstraktionsprinzip

Mit der Datenabstraktion stehen Methoden zur Verfügung, mit denen die Art der Verwendung eines Datenobjektes von den Details seiner Konstruktion aus elementareren Datenobjekten getrennt werden kann.

Die Grundidee bei der Datenabstraktion besteht darin, die Programme zur Verwendung von zusammengesetzten Datenobjekten so zu strukturieren, dass sie

mit „abstrakten Daten“ arbeiten. Gleichzeitig wird eine „konkrete“ Darstellung der Daten unabhängig von den die Daten verwendenden Programmen definiert. Die Schnittstelle zwischen diesen beiden Teilen ist eine Menge von Funktionen, die *Konstruktoren* und *Selektoren*, sowie *Mutatoren* genannt werden und die abstrakten Daten in der konkreten Darstellung implementieren.

Die Idee bei der Funktionsabstraktion ist ähnlich der bei der Datenabstraktion und erinnert durchaus an objektorientierte Programmierung: Eine Funktion wird durchaus durch andere Funktionen aufgebaut. Ihre Anwendung ist aber unabhängig von diesen Details. Eine solche Funktion kann auch jederzeit durch eine andere Funktion ersetzt werden, die das gleiche Verhalten zeigt. Dieser Teil ist insbesondere für funktionale Sprachen interessant, wird in dieser Vorlesung jedoch nicht behandelt.

Definition 2.3 *Ein abstrakter Datentyp (ADT) ist ein Datentyp (Objektmengen, auch Wertebereich oder Wertemenge genannt), auf den der Anwender über einen Satz von Funktionen (Operationen) zugreift. Die Funktionen sind durch ihre Signatur (also Input- Outputparameter) und Vor- und Nachbedingungen definiert (Pre-, Postconditions). (siehe unten) Die interne Form (Implementierung und Repräsentation) ist für den Anwender nicht sichtbar (Geheimnisprinzip, Kapselung).*

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 Repräsentation) 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** als Konstruktor: Das Objekt wird erzeugt und ggf. auf bestimmte Weise vorbelegt.

2. **Nicht-destruktive Operationen** als Selektor: Das Objekt wird betrachtet oder gelesen.

3. **Destruktive Operationen** als Mutator: Das Objekt (insbesondere sein innerer Zustand) wird verändert oder beschrieben.

4. **Funktionen** als Konstruktor: Aus einem oder mehreren Objekten entsteht ein neues.

5. **Zerstörung** als Mutator: Beseitigen des Objekts und Freigabe des Speichers.

Der **syntaktische** Aspekt der Operation wird durch die Signatur beschrieben. Um den Inhalt der Operation, also den **semantischen** Aspekt, zu beschreiben, verwenden wir Vor- und Nachbedingungen:

Definition 2.4 *Die Vor- und Nachbedingungen werden oft in einer formalen, z.B. mathematischen Schreibweise spezifiziert, da diese sehr präzise sind und somit Fehlinterpretationen und damit Fehler in der Anwendung vermeiden helfen.*

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 nicht von einer korrekten Eingabe ausgehen und dem Anwender Abfragen in Form von Bedingungen einbauen (assert), die dem Anwender eine Überprüfung der geplanten Eingabewerte erlauben.*

Nachbedingung *(postcondition): Gibt die Eigenschaften des Objekts nach Beendigung der Operation an (das Ergebnis der Operation) und ggf. ob ein zusätzlicher Rückgabewert besteht.*

Es wurde von IBM einmal der Versuch unternommen, für die Beschreibungssprache ein Fragment der Prädikatenlogik erster Stufe zu verwenden, mit dem Ziel, die Beschreibungen dann in PROLOG 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 Beschreibung 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 sicherer zu machen. Meist wird dies aber aus Zeitgründen von den Entwicklern nicht sorgfältig durchgeführt in dem Sinne, ob die Dokumentation überhaupt erstellt wird (haben Sie für alle Ihre Methoden/Funktionen Vor- und Nachbedingungen formal beschrieben ?) und ob die Dokumentation mit der Implementierung abgeglichen bzw. verifiziert wird (haben Sie formal geprüft, ob Ihr

Kommentar und der zugehörige Code übereinstimmen und dies bei Änderungen gepflegt ?). Dies führt z.B. zu neuen SE-Verfahren, z.B. einer modellgetriebenen Entwicklung (z.B. MDA), in der möglichst wenig Dokumentation neben der Implemntierung geschrieben werden muss und in dem Sinne die Idee z.B. von IBM versucht wird zu realisieren: die Dokumentation bzw. Beschreibung des Problems/Lösungswegs/etc. *ist* die Implementierung.

Beispiel 2.3 ADT + und /

$$\begin{aligned} \textit{Operation } +\textit{:} \quad & \textit{int} \times \textit{int} \to \textit{int};\ (a,b) \mapsto a+b \\ \textit{pre:} \quad & \textit{keine} \\ \textit{post:} \quad & a+b \textit{ ist die Summe der Zahlen } a \textit{ und } b \end{aligned}$$

$$\begin{aligned} \textit{Operation } /\textit{:} \quad & \textit{float} \times \textit{float} \to \textit{float};\ (a,b) \mapsto a/b \\ \textit{pre:} \quad & b \neq 0 \\ \textit{post:} \quad & a/b \textit{ ist der Quotient aus den Zahlen } a \textit{ und } b \end{aligned}$$

Im letzten Beispiel könnte man alternativ auch schreiben:

$$\begin{aligned} \textit{Operation } /\textit{:} \quad & \textit{float} \times \textit{float} \to \textit{float} \cup \{\ \textit{error}\ \};\ (a,b) \mapsto a/b \\ \textit{pre:} \quad & \textit{keine} \\ \textit{post:} \quad & a/b \textit{ ist der Quotient aus den Zahlen } a \textit{ und } b \\ & \textit{Falls } b = 0 \textit{ wird error zurückgeliefert} \end{aligned}$$

Dies wäre auch auf die Vorbedingungen zu übertragen. Hier sind keine spezifiziert, weil man implizit durch die Typisierung hier von Zahlen als Eingabe ausgeht. In nicht typisierten Sprachen wäre also in den Vorbedingungen zu formulieren, dass a und b jeweils Zahlen von dem geforderten Typ sind und ggf. bei falscher Eingabe, wie hier im Falle $b \neq 0$ einen Fehlerwert zurückgibt. Man hat dann natürlich das Problem, in welcher Form man den Fehlerwert error zurückliefert, 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 */
```

Beispiel 2.4 ADT Polynom
Auf der Menge der Polynome sind auch gewisse Rechenoperationen erlaubt.

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

Aufgabe 2.2 *Schreiben Sie obiges Beispiel so um, dass die Precondition fallen gelassen werden kann.*

Beispiel 2.5 ADT Student
Auf der Menge der Studenten sind gewisse Operationen definiert.

$$\begin{aligned}
\textit{Operation erzeuge:} &\quad \emptyset \to \textit{Student} \quad ; \quad \textit{Student } s; \\
\textit{pre:} &\quad \textit{Objekt vom Typ HAW wurde vorher erzeugt} \\
\textit{post:} &\quad s.\textit{name} = 0,\ s.\textit{vorname} = 0,\ s.\textit{matr} = -1
\end{aligned}$$

$$\begin{aligned}
\textit{Operation einschreiben:} &\quad \textit{Student} \times \textit{string} \times \textit{string} \times \textit{int} \to \textit{Student} \\
&\quad s = s.\textit{einschreiben(name, vorname, matr);} \\
\textit{pre:} &\quad \textit{name enthält keine Ziffern} \\
\textit{pre:} &\quad \textit{vorname enthält keine Ziffern} \\
\textit{pre:} &\quad \textit{matr} > 0 \\
\textit{post:} &\quad \textit{die Daten von } s \textit{ werden überschieben} \\
&\quad s.\textit{name} = \textit{name} \\
&\quad s.\textit{vorname} = \textit{vorname} \\
&\quad s.\textit{matr} = \textit{matr}
\end{aligned}$$

Natürlich verlangt man von einem ADT, der zu irgendetwas nütze sein soll, bestimmte Eigenschaften, die immer gelten sollen. Triviales Beispiel: Ein sortiertes Array weist eine bestimmte Eigenschaft (Sortierung) auf seinen Elementen auf. Verschwindet diese Eigenschaft, so hört das Objekt auf, ein sortiertes Array zu sein.

Diese Eigenschaft eines ADT nennen wir die **Invariante des ADT**, weil sie durch nichts und niemanden geändert werden darf.

Natürlich ist es kaum zu realisieren (Ausnahmen!), die Invariante zu jedem beliebigen Zeitpunkt der Lebensdauer des Objekts zu garantieren! Viele Algorithmen modifizieren zunächst das gegebene Objekt so, dass die Invariante verletzt wird, basteln sie dann aber sofort wieder hin. Beispiele sind Heaps oder Rot/Schwarz-Bäume, bei denen genau diese Strategie angewendet wird.

Wir verlangen also, dass die Invariante nach jeder Operation auf dem ADT wieder gilt, wenn sie vor der Operation gegolten hat. Dieser Nebensatz ist ganz wichtig, weil sich einige Algorithmen darauf verlassen, dass die Invariante gilt und nur unter dieser Bedingung die Invariante auch wieder hinbekommen.

In vielen Fällen ist die Invariante ziemlich kompliziert. Sie kann z.B. durch eine ziemlich Umfangreiche Definition gegeben werden, auf die wir uns dann bei der Formulierung der Invariante beziehen.

2.3 Lineare Listen

Listen sind ein klassisches Beispiel für einen ADT. Wir werden später aus einer Liste mit eingeschränkter Funktionalität noch weitere ADTs entwickeln. LISP ist z.B. eine Programmiersprache (LISt Processing), die eigentlich nur Listen kennt (und Funktionen, die als Listen notiert werden). Daran sollte deutlich werden, wie mächtig diese recht simple ADT sein kann!

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 $a_1, a_2, \ldots, a_i$. Außerdem sind Duplikate von Elementen erlaubt. Wir betrachten Folgen von Objekten eines bestimmten Grundtyps. Dieser Grundtyp könnte z.B. ein Druckauftrag sein oder Wörterbucheinträge oder Zahlen, was immer man eben in einer Liste abspeichern will. 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)

Dieser Grundtyp stellt schon eine praktischere Betrachtung dar, als manchmal in der Literatur verwendet. Ab einer gewissen Größe der zu verwaltenden Daten komprimiert man den interessanten Inhalt auf einen Schlüssel, der in der Datengröße klein ist gegenüber den inhaltlichen Daten. Verwaltet wird dann nur noch der Schlüssel, über den man dann zu den eigentlichen Daten kommen kann. In der Literatur belässt man es teilweise bei den Schlüsseln, d.h. z.B. bei einer Liste von Zahlen.

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.

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

Definition 2.5 *Der ADT* **Liste** *zeichnet sich durch folgende Operationen aus:*

- Einfügen *eines neuen Elements $x \in G$ (vom Grundtyp) in die Liste L an Position $p \in pos$; dabei muß die Liste bereits mindestens bis Position $p-1$ gefüllt sein. Wenn es bereits ein Element an Position p gibt, werden die Elemente ab dieser Position in der Position um eins nach rechts verschoben.*

- Entfernen *eines Elements $p \in pos$ aus der Liste L; ggf. werden Elmente um eins nach links verschoben.*
- Suchen *der Position $p \in pos$ eines Elements $x \in G$ in Liste L*
- Zugriff *auf Element an der Position $p \in pos$ in Liste L*
- Konkatenation *zweier Listen L_1 und L_2*
- Initialisierung *Erzeugen einer leeren Liste*

Eine Liste hat also keine Lücken! Die Operationen können je nach Anwendung variieren.

Die Signaturen dieser Operationen sind:

$$\begin{array}{llllllll} \mathit{insert} & : & \mathit{list} & \times & \mathit{pos} & \times & \mathit{elem} & \rightarrow & \mathit{list} \\ \mathit{delete} & : & \mathit{list} & \times & \mathit{pos} & & & \rightarrow & \mathit{list} \\ \mathit{find} & : & \mathit{list} & \times & \mathit{elem} & & & \rightarrow & \mathit{pos} \\ \mathit{retrieve} & : & \mathit{list} & \times & \mathit{pos} & & & \rightarrow & \mathit{elem} \\ \mathit{concat} & : & \mathit{list} & \times & \mathit{list} & & & \rightarrow & \mathit{list} \end{array}$$

Vor- und Nachbedingung beispielhaft für insert:

Operation insert: $\mathit{list} \times \mathit{pos} \times \mathit{elem} \rightarrow \mathit{list}$; *L.insert(p,x)*
pre: $p \in \{p_0, \ldots, p_n, p_{n+1}\}$, *eine erlaubte Position,*
Position p_{n+1} sei die leere Position (ϵ das leere Element)
post: *Sei $L = (a_0, \ldots, a_n, \epsilon_{n+1})$ eine Liste.*
Sei a_i das Element an Position p_i. Dann bewirkt
$L.insert(p_i, x) = (a_0, \ldots, a_{i-1}, x, a_i, a_{i+1}, \ldots, a_{n+1}, \epsilon_{n+2})$

Die Invariante bei allen Listenoperationen ist, das die behandelten Objekte Listen sind. Während der Bearbeitung wird die Invariante oft nicht eingehalten, weil z.B. die Liste aufgebrochen wird oder ähnliches.

2.3.1 Implementierung 0: Sequenzielle Speicherung

Datentypen können in statische und dynamische Typen unterschieden werden. Der Bezugspunkt ist der Zeitpunkt der Compilation oder des Programmstarts: bei statischen Typen steht der Platz von Anbeginn zur Verfügung; bei dynamischen Typen wird während des Programmlaufs dynamisch Speicherplatz allokiert oder freigegeben. Es gibt natürlich auch hybride Typen, bei denen ein Teil statisch und ein Teil dynamisch festgelegt wird.

Hier in diesem Abschnitt betrachten wir eine statische Implemntierung, wogegen im nachfolgenden Abschnitt eine dynamische Implementierung betrachtet wird. In diesem Sinne werden gleichwertige Implementierungen mit unterschiedlichen Qualitätsmerkmalen vorgestellt. Letztlich entscheidet die Anwendung, welche die „bessere“ ist, ggf. ist es der Anwendung „egal“.

```
class SeqList
{
    TYP liste[ ]; //TYP ist Grundtyp
  int listSize;
  int maxSize;
public:
    ...
}
```

Listenelemente können an den Positionen $0, \ldots, max - 1$ gespeichert sein. Da hier eine feste Größe verwendet wird, kann man ein Überschreiten der Grenzen durch Vergleich mit den Indizes vermeiden (Invariante: $listSize < max - 1$). Unter Umständen ist in einer oder zwei Variablen festgehalten, welcher Teil der Liste bereits mit sinnvollen Einträgen gefüllt wurde. Hier im Beispiel etwa gibt $listSize$ die Position des letzten Elementes der Liste an.

Position 0 kann als eine uneigentliche Listenposition verwendet werden. 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. In diesem Fall wird $listSize = 0$ als leere Liste interpretiert und $listSize > 0$ beschreibt damit auch die Größe der Liste.

Der Code für die Suchfunktion sieht so aus:

```
int suche(TYP x)
{// liefert von rechts erste Position von x
 // und 0 falls x nicht vorkommt
    t[0] = x; //Stopper
    int pos = listSize;
    while ( t[pos] != x)
        pos = pos - 1;
    return pos;
}
```

Wie am Anfang der Vorlesung erwähnt, werden hier Kernlösungen vorgestellt. Spezialisierungen werden aber meist im konkreten Fall eingesetzt. So gibt es z.B. effizientere Verfahren zum Suchen eines Elements, wenn die Elemente nach aufsteigenden Schlüsselwerten sortiert sind.

Beim Einfügen eines neuen Elements müssen bei der gewählten Implementierung die nachfolgenden Elemente verschoben werden.

- Beim Einfügen und Löschen müssen Elemente verschoben werden. Dies führt bei großen Listen zu viel Rechenaufwand (O(N)). Günstigster Fall: am Ende einfügen. Ungünstigster Fall: am Anfang einfügen.

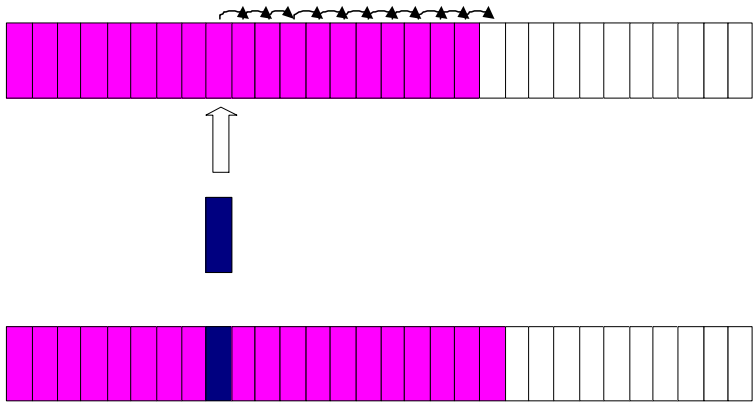


Abbildung 2.1: Einfügeoperation bei Listen als Arrayeinbettung.

- Die Komplexität O(N) ist unabhängig davon, ob die entsprechende Position gegeben ist, oder durch Suchen erst gefunden werden muss. Dazu aber später mehr.

Contiguous memory container So „ungünstig“ ist eine Speicherung im Array nur auf den ersten Blick. Die STL (Standart 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 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. Hier wurde die statische Datenstruktur mit der dynamischen in dem Sinne kombiniert, dass nicht genau der Speicher angefordert wird, den man benötigt, sondern einen grösseren Block. Innerhalb dieses Blockes wird dann dynamisch durch Zeiger (also Adressen) die gewünschte Struktur, z.B. Listen, aufgebaut. Diese „Verzeigerung“ kann z.B. 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 sogenannte *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 sogenannten *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 2.5.3.

2.3.2 Implementierung 1: einfach verkette Listen

Einfach verkettete Liste als Folge von Knoten: jeder Knoten enthält ein Listenelement des Grundtyps und einen Zeiger auf das nächste Listenelement. In dem

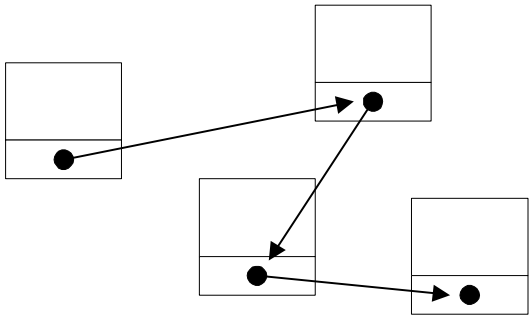


Abbildung 2.2: Struktur einer einfach verketteten Liste.

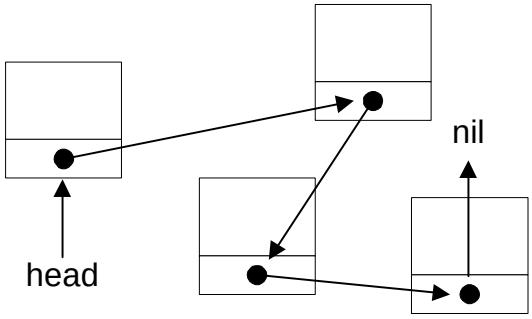


Abbildung 2.3: head- und tail-Zeiger einer einfach verketteten Liste.

Sinne haben wir es hier mit einer rekursiven (und dynamischen) Datenstruktur zu tun: Auf die Definition einer Datenstruktur (Klasse) darf in derselben Definition - in Form einer Referenz - Bezug genommen werden. In manchen Programmiersprachen (z.B. Pascal) geht das erst nach einer sogenannten `forward`-Vereinbarung.

Der Code für einen Knoten sieht folgendermaßen aus:

```
class Knoten
{
    TYP dat;  //TYP ist Grundtyp oder Referenz/Schlüssel auf Daten
    Knoten* next; // Rekursion
public:
    ...
}
```

Wir müssen definieren, was mit dem ersten und letzten Element geschehen soll bzw. wir benötigen eine Verankerung der Liste: Einfach verkettete Liste mit Zeiger `head` auf Listenanfang, Listenende als `nil` markiert. Die Liste in dieser Implementierung stellt sich dann so dar:

```
class List1
{
    Knoten* head;
```

```
public:
    // insert, find, etc.
}
```

Diese Liste ist genau dann leer, wenn `head = nil` gilt.

Positionen werden hier also nicht - wie bei sequentiellen Listen - explizit als laufende Nummer angegeben, sondern implizit durch die Position innerhalb der verketteten Liste.

Ein Listenelement suchen gestaltet sich hier wie folgt:

1. Überprüfen, ob `head = nil`, also ob die Liste leer ist.
2. Überprüfen, ob `Knoten.next = nil`, also ob man das Listenende erreicht hat.
3. Überprüfen, ob `Knoten.dat = x`, also ob das Element gefunden wurde

Hier sind zwei zusätzliche Überprüfungen gegenüber der Version mit Stopper-Element im vorangegangenen Abschnitt.

```
Knoten* suchen(TYP x)
{
    if ( head == nil ) //Liste leer ?
        return 0;
    else
    {
        Knoten* pos = head;
        while ( (pos->dat != x) && (pos->next != nil))
            pos = pos->next;
        // an dieser Stelle ist pos->dat = x oder
        // pos->next = nil
        if ( pos->dat = x )
            return pos;
        else return 0;
    }
}
```

Hier ist zu beachten, das der Wert Null (0) ein spezieller Rückgabewert ist, der wegen den evtl. unterschiedlichem Typ zu Referenzen/Zeigern in manchen Sprachen auch mit $NULL$ bezeichnet wird. In manchen Sprachen ist es aber auch egal. Die $NULL$ macht aber deutlich, das der Entwickler weis, was er hier tut.

Kann man dies nicht „schöner“ machen ? Kommt auf die Anwendung an. Muss man aus Sicht der Anwendung meist am Anfang in der Liste einsteigen, können wir zufrieden sein. Sollte die Anwendung bzw. die Problemstellung jedoch erforderlich machen, dass man „gleichoft“ am Anfang wie am Ende einsteigen muss, so hilft die Implementierung im nächsten Abschnitt weiter.

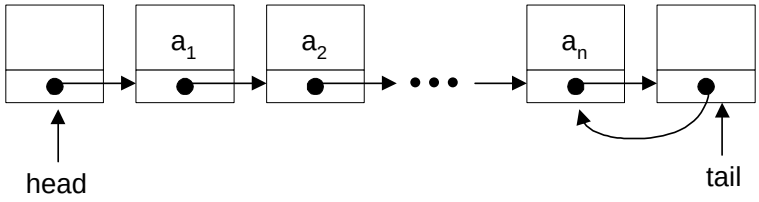


Abbildung 2.4: Dummy-Elemente am Anfang und Ende einer einfach verketteten Liste.

2.3.3 Implementierung 2: einfach Verkettete Listen mit Tail-Zeiger und antizipativer (vorweggenommener) Indizierung

Die Idee ist, sich den Anfang *und* das Ende der Liste zu merken.

- Kopfzeiger head
- Schwanzzeiger 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.
}
```

Dazu ein paar Bemerkungen:

- Liste ist durch Schwanz- und Kopfzeiger gegeben
- der next-Zeiger des letzten Elements zeigt auf das letztes Element (statt nil). Dies erleichtert bei der hier vorgestellten Implementierung die Konkatenation von Listen.

Die Initialisierungsroutine für diese Liste sieht folgendermaßen aus:

```
void List2::init()
{
```

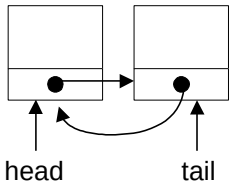


Abbildung 2.5: Leere Liste mit je einem Dummy am Anfang und am Ende.

```
    head = new Knoten;
    tail = new Knoten;
    head->next = tail;
    tail->next = head;
}
```

Suchen durch Stopper-Technik: Gesuchtes Element wird vor Beginn der Suche in das Dummy-Element am Listenende geschrieben.

```
Knoten* suche(TYP x)
{// liefert von links erste Position von x
 // und Pointer auf tail falls x nicht vorkommt
    tail->dat = x;         //x als Stopper einfügen
    Knoten* pos = head;    //beim head starten
    while ( pos->dat != x);
       pos = pos->next;
    if pos == tail
       return 0;
    else return pos;
}
```

Die Vorgehensweise entspricht im Wesentlichen der Vorgehensweise der sequentiellen Implementierung.

Bemerkung 2.3 *Das eigentliche Ziel ist natürlich nicht das Element in der Liste, sondern die Daten, die sich dahinter „verbergen“!*

Einfügen eines Elements an der Position p: Element x soll an Position p eingefügt werden. Zeigt p tatsächlich auf das Element p, so ergibt sich aufgrund der gewählten Implementierung das Problem, die Position p-1 zu finden, für dessen Element der next-Zeiger verändert werden muss. Die Komplexität für die Suche ist 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

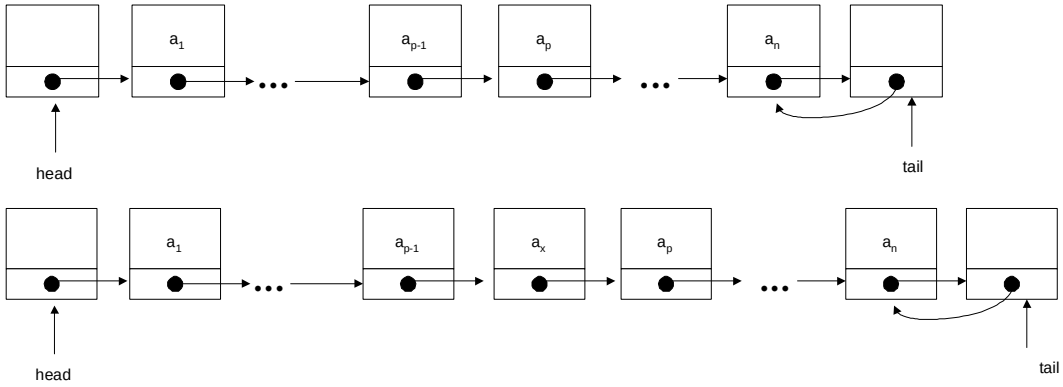


Abbildung 2.6: Einfügen eines Elements in eine einfach verkettete Liste.

das Dummy-Element am Anfang. Man sieht an diesem Beispiel, das bestimmte gewählte Implementierungen nicht für alle benötigten Funktionen unbedingt alle Vorzüge haben. Manchmal wird Effizienz auf Kosten von Verständlichkeit gewonnen und manchmal wird Effizienz bei oft verwendeten Funktionen durch Ineffizienz bei nicht oft verwendeten Funktionen (z.B. Initialisierungsfunktionen) gewonnen etc.

```
insert ( pos p , elem x )
{
    pos q = new pos( );   // neuer Knoten.
    q->dat = x;        // neuer Knoten soll x enthalten
    q->next = p->next->next;
    if ( q ->next == tail ) // letzte Position
        tail->next = q;
    p->next = q;
}
```

Dadurch wird das Einfügen auf eine Komplexität O(1) reduziert.

Listen erlauben in zwei Richtungen sich zu bewegen, nach links bzw. nach rechts. Auch hier kann die Anwendung die gezeigt Implementierung als „unschön“ erscheinen lassen, wenn z.B. man oft „hin und her“ laufen muss. Im nächsten Abschnitt wird dazu eine passendere Implementierung der Liste vorgestellt.

2.3.4 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. Es ergibt sich für die einzelnen Zugriffsmethoden und unterschiedlichen Implementierungen folgende „theoretische“ Effizienz:

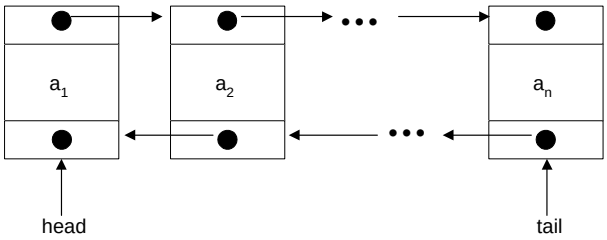


Abbildung 2.7: Doppelt verkettete Liste.

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

Für das Löschen von Listenelementen muß man sich Gedanken machen, was passieren soll. Für die Liste ist erstmal das Resultat in allen Fällen gleich: aus der Liste heraus kann man auf das gelöschte Element nicht mehr als Element zugreifen!

- destruktiv bzgl. Daten: Knoten aus Liste aushängen; Nutzerdaten auf Null setzen bzw. je nach Sicherheitsbedarf explizit überschreiben, was eigene Routinen erfordern kann; Speicherblock ggf. in eine Freiliste einhängen. Sollte nur ein Verweis auf die Nutzerdaten in der Liste vorhanden sein, ist hier zu klären, ob dieser gelöscht werden soll oder auf die verwiesenen Daten.

- nicht destruktiv bzgl. Daten: Knoten aus Liste aushängen; Nutzerdaten nicht löschen, da der Knoten noch in einer anderen Liste oder Anwendung eingehängt ist. Dazu aber den *next*-Zeiger auf Null setzen, da sonst ein „nicht wirklich“ registrierter Verweis in die aktuelle Liste besteht. Sollte nur ein Verweis auf die Nutzerdaten in der Liste vorhanden sein, ist hier zu klären, ob nur die Nutzerdaten als solche erhalten bleiben sollen, was ein Löschen des Verweises erlauben würde, oder ob das Listenelement als solches erhalten bleiben soll.

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

2.4 Stapel und Schlangen (Stack and Queue)

In diesem Abschnitt sollen zwei sehr häufig verwendete Datentypen, nämlich Schlange und Stapel, durch Listen realisiert werden. Das Abstraktionsprinzip

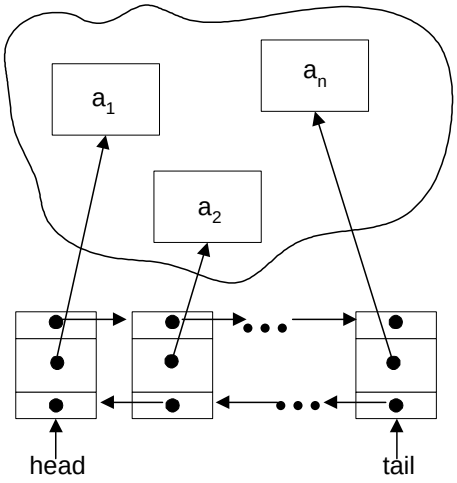


Abbildung 2.8: Die Nutzerdaten werden möglichst in den Knoten nur referenziert.

lässt aber die konkrete Realisierung für die Nutzer dieser ADTs nicht sichtbar werden. Man könnte nun fragen, warum man dann z.B. Listen nicht direkt anbietet ? Die beiden hier behandelten Datenstrukturen zeichnen sich durch eine bestimmte Verhaltensweise aus. Durch Einschränkung der Zugriffsmöglichkeiten stellt man nun die Invariante sicher, dass die verwendete Datenstruktur nicht „versehentlich“ sich für kurze Augenblicke nicht wie vereinbart verhält, z.B. indem sich jemand in der Schlange „vor drängelt“.

Für die Anwendung Schlange bzw. Stapel 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 diese Art von Liste:

- kontrollierte Zugriffspunkte
- leichte Implementierung mit konstanter Schrittzahl

Stapel (Stack) und Schlange (Queue) als ADTs sind sicherlich noch aus anderen Veranstaltungen bekannt. Zur Erinnerung ein Stapel (Stack) verhält sich

nach dem LIFO-Prinzip: Last In First Out und eine Schlange(Queue) nach dem FIFO-Prinzip: First In First Out. Auch hier, wie immer, gibt es Spezialfälle, die davon abweichen können. Hier ein paar **Anwendungen** des ADT's:

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

2.4.1 Stapel (Stack)

Der Stapel, auch als *LIFO*-Puffer bezeichnet, kann als Spezialfall einer Liste angesehen werden. Er enthält nur die Zugriffsfunktionen am Ende der Liste: pushtail und poptail. Meist sagt man nur push und pop. Oft wird ein Stapel als Array implementiert. Da in der Mitte des Stapels nicht eingefügt werden kann, entfällt das Verschieben der Elemente.

ADT Stack

empty:	$\emptyset$	$\rightarrow$	stack
push:	stack $\times$ elem	$\rightarrow$	stack
pop:	stack	$\rightarrow$	stack
top:	stack	$\rightarrow$	elem
isempty:	stack	$\rightarrow$	bool

Operation push:	stack $\times$ elem $\rightarrow$ 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 2.3 *Beschreiben Sie die Operationen*

pop:	*stack*	$\rightarrow$	*stack*
top:	*stack*	$\rightarrow$	*elem*
isempty:	*stack*	$\rightarrow$	*bool*

Einen Stapel 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 Stapel beginnt. Bei einer geschlossenen Klammer wertet man den Ausdruck auf dem Stapel aus und überträgt das Ergebnis auf den vorherigen Stapel. Hier ist die Stapelbelegung:

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

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

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

Aufgabe 2.4 *Wenden Sie diese Regeln auf den unten stehenden Ausdruck an. Notieren Sie sich skizzenhaft die einzelnen Schritte mit den entsprechenden Belegungen der Stapel.*

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

Hier ist eine mögliche Implementierung für einen Stapel.

```
class Stack
{
    int stackSize;
    int top = 0;//zeigt auf das Element über dem obersten.
    Elem a[ ];
public:
    Stack(int sz)
    {
        stackSize = sz;
        a = new Elem ( stackSize );
    }
    push(Elem e)
    {
        if ( top == size - 1 ){throw error;}
        else a[top++] = e;
    }
}
```

Aufgabe 2.5 *Schreiben Sie die pop-Funktion für einen Stapel. Achten Sie auf Sonderfälle!*

2.4.2 Schlangen (Queues)

Eine Queue, auch als *FIFO*-Puffer bezeichnet, enthält nur die Zugriffsfunktionen

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)

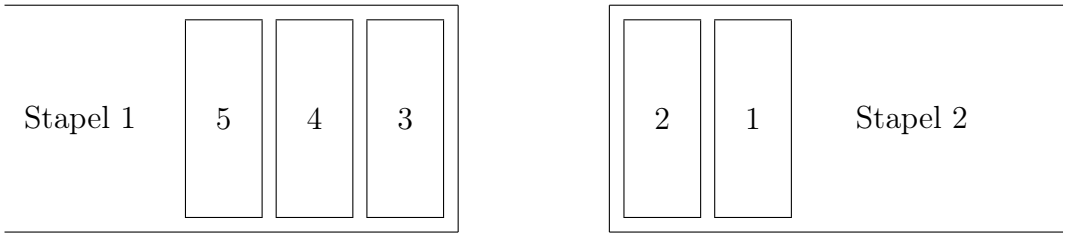
dequeue: Löscht das Element head

isempty: 1 falls die Queue leer ist , 0 sonst.

Man kann also hinten einfügen und vorne auslesen.

```
class Queue
{
    int rear = 0;
    int head = 0;
    int queueSize=0;
    Elem e[ ];
public:
    Queue(int sz)
    {
        queueSize = sz;
        e = new Elem(queueSize);
    }
}
```

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.



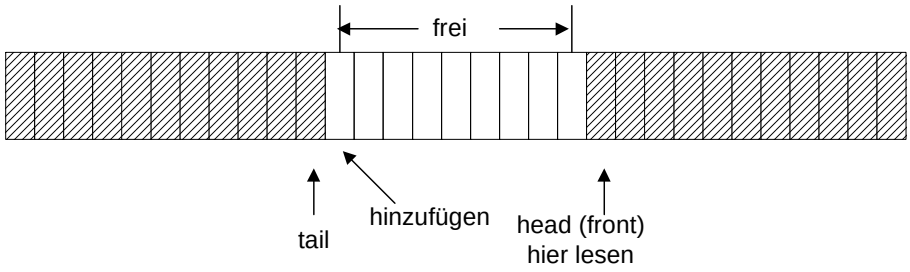


Abbildung 2.9: Eine Queue wird gewöhnlich als Ringbuffer implementiert. Dazu wird ein Array allokiert, in dem ein head- und ein tail-Zeiger die Queue-Elemente eingrenzen.

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.

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.

2.5 Spezielle Anwendungen

2.5.1 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. 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 2.4 *Java Generics und Arrays: Java ermöglicht es seit Version 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 2.5 *Listen in Java: Java bietet verschiedenen Implementierungen für Listen, z.B. ArrayList und LinkedList.*

2.5.2 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:

- Ein gängiges Vorgehen für die Implementierung einer Liste ist, einen Container zu schaffen, welcher Referenzen auf die zu verkettenden Objekte enthält. Die Container selbst sind miteinander durch Referenzen verlinkt. Dies nennt man nicht-intrusive Liste.

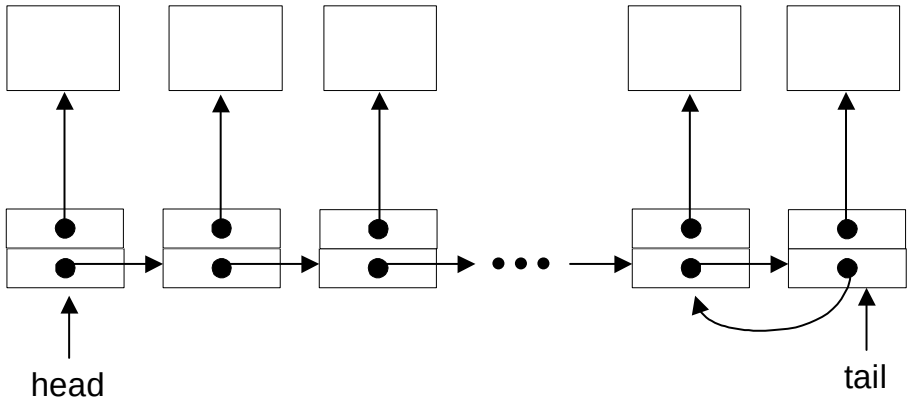


Abbildung 2.10: Nicht-intrusive Implementierung einer verketteten Liste. Die Listen-Container enthalten Referenzen oder Pointer auf die zu verkettenden Objekte.

- Eine intrusive Liste zeichnet sich dadurch aus, dass die zu verkettenden Objekte selbst ihre `next`Pointer (oder Referenzen) enthalten, d.h. die Referenz steht nicht ausserhalb des Objektes, sondern ist in das Objekt „eingedrungen“. In der konkreten Implementierung kann dies auf verschiedenen Wegen erreicht werden. Typisch ist hier auch die Anwendung von Templates in C++.

intrusive	nicht-intrusive
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.

Literatur zu Intrusive-Technologien findet man auf der Homepage von Code Farms.

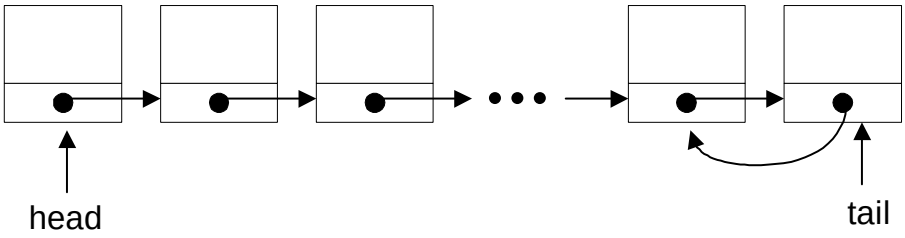


Abbildung 2.11: Intrusive Implementierung einer verketteten Liste. Die next Pointer sind Bestandteil der Objekte, welche verkettetet werden sollen.

2.5.3 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, nicht-dynamischer Speicher	für Datensätze gleicher Größe, Messages
besonders sicher (keine Programmierfehler)	besonders schnell ($O(1)$)

Literatur hierzu: Andrei Alexandrescu: Modernes C++ Desing.

2.5.4 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). Literatur hierzu:

- Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects, Douglas C. Schmidt
- Consensus-based lock-free asynchronous three-buffer Communication, Reto Carrara

- Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Consensus, Chen, Burns, 1998

2.6 Fazit aus Sicht der Konstruktionslehre

Das wesentliche Konstruktionsprinzip hier ist die **Abstraktion**. 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:

> 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.

2.7 Aufgaben

1. **Algorithmus** (? Punkte)

   (a) Beschreiben Sie den Begriff Algorithmus und erklären Sie die vier allgemeinen Anforderungen an einen solchen Algorithmus.

   (b) Erklären Sie folgende vier Eigenschaften eines Algorithmus: *terminiert, determiniert, deterministisch und nicht deterministisch.*

2. **Datenabstraktion** (? Punkte)

   (a) Beschreiben Sie das Prinzip bzw. die Grundidee der Datenabstraktion.

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

(c) In der Vorlesung wurde ein Entwurf einer Schlange vorgestellt, deren Funktionalität mit zwei Stapeln realisiert wurde. Ist es möglich, mit zwei Schlangen die Funktionalität eines Stapels zu realisieren ? Begründen Sie Ihre Antwort, indem Sie auch erklären, wieso die Schlange mittels zwei Stapeln realisiert werden kann!

3. **Datenabstraktion** (? Punkte) 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`).

4. **Datenabstraktion** (? Punkte)
   Definieren Sie durch Angabe der formalen Beschreibung einen Datentyp `Queue`, der eine Schlange nach dem First-In-/First-Out-Prinzip (FiFo) modelliert. Verwenden Sie dazu die Wertebereiche (Typen) `Elements` und `Boolean`.

   (a) Geben Sie für die Operationen `create, isEmpty, enqueue, dequeue` und `front` die allgemeine/formale Typisierung an (z.B. durch Kreuzprodukt).

   (b) Der Datentyp `Queue` mit z.B. Zahlen als Elementen kann mittels eines festen Arrays $A[0, \ldots, m-1]$ und zweier Zähler `head` und `tail` implementiert werden. Geben Sie in Pseudocode die Operationen `enqueue` und `dequeue` an. Die Behandlung des Überlaufs kann mittels `Überlauf behandeln` abstrahiert werden.

   (c) Benennen Sie allgemein den Zeitaufwand für `enqueue` und `dequeue`, falls kein Überlauf auftritt.

   (d) Der Inhalt der Queue sei: $(5, 7, 9, 12, 4, 8)$, m sei 10. Zeichnen Sie das Array und geben Sie den Inhalt von `head` und `tail` an, wenn der erste Eintrag 5 in Position $A[8]$ steht.

5. **Datenstrukturen** (? Punkte)

(a) Wir möchten von einem Strom von Zahlen (z.B. von einer kontinuierlich durchgeführten Messung) die letzten n Zahlen speichern. Eine neue Zahl muss also die am längsten vorgehaltene Zahl verdrängen. Entwickeln Sie für dieses Problem eine Datenstruktur und die dazugehörige Einfüge-Operation in Pseudocode. Die Einfüge-Operation soll eine Laufzeitkomplexität $O(1)$ haben. Nehmen Sie dazu an, dass n bekannt ist. Beachten Sie, dass die Datenstruktur am Anfang der Aufzeichnung noch nicht gefüllt ist.

(b) Nun möchten wir zusätzlich den Durchschnitt aller aktuell vorhandenen Zahlen in dieser Datenstruktur speichern. Erweitern Sie Ihre Einfüge-Operation derart, dass diese nach dem Einfügen der neuen Zahl den neu berechneten Durchschnitt zurückgibt. Die Laufzeitkomplexität darf sich nicht verschlechtern!

(c) Begründen Sie, warum Ihre Implementierung aus Teilaufgabe (b) die Laufzeitkomplexität einhält.

(d) Weisen Sie die Korrektheit Ihrer Einfüge-Operation aus Teilaufgabe (b) nach.

6. **Rekursion** (? Punkte)
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!

7. **Rekursion** (? Punkte)

   (a) 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.

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

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

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

   Beachten Sie: a_n ist gleich der Schreibweise $a(n)$ und damit ist a als Funktion/Methode umzusetzen.

(b) Zeigen Sie den Ablauf der rekursiven Aufrufe für π_2 auf. Eine Berechnung ist dabei nicht durchzuführen! Wie oft werden a und b dabei jeweils aufgerufen?

8. **Rekursion** (? Punkte) Gegeben sei nachfolgende Funktion:

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

(a) Implementieren Sie diese Funktion direkt, also ohne Optimierungsüberlegungen,in Java.

(b) 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.

(c) 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.

9. **Rekursion** (? Punkte)
Zum deterministischen Testen von Anwendungen lassen sich sogenannte Pseudo-Zufallszahlen verwenden. Folgende Berechnungsformel erzeugt für $0 \leq n \in \mathbb{N}^0$ solche Zahlen:

$$f(n) = \begin{cases} n+1 & \text{falls } 3 > n \\ 1 + (((f(n-1) - f(n-3)) * f(n-2)) \bmod 100) & \text{falls } 3 \leq n \end{cases}$$

(a) Implementieren Sie diese Funktion rekursiv ohne Optimierungsüberlegungen, also direkt, in Java in folgendem Rahmen:
`public static int f(int n) { }`

(b) Implementieren Sie die Funktion f linear-rekursiv (rekursiv mit iterativem Ablauf) mit Durchreichen von Zwischenergebnissen mittels der Funktion *linrek* in Java in folgendem Rahmen:

```
public static int f(int n) {
    return linrek(1, 2, 3, n-3);
}
public static int linrek(int a, int b, int c, int steps) {
}
```

(c) Implementieren Sie die Funktion f iterativ (ohne Rekursion) in Java in folgendem Rahmen:
`public static int f(int n) { }`

(d) Welche der drei Lösungen ist bereits linear in der Laufzeitkomplexität und wie kann man die Funktion f in den noch nicht linear in der Laufzeitkomplexität implementierten Varianten realisieren, so dass sie im Zeitaufwand linear werden, also die Komplexität $O(n)$ besitzten ?

Transformieren Sie dazu ggf. Laufzeitaufwand der (jeweiligen) naiven Lösungen in Speicherplatzaufwand, jedoch mit maximal $O(n)$ Speicherplatzaufwand! Skizieren Sie dazu (jeweils) ein Java-Programm und begründen kurz, warum der Aufwand (jeweils) (Laufzeit-/Speicherplatz) linear sein sollte.

10. **Rekursion** (?? Punkte)
Analysieren Sie die folgende rekursive Funktion, wobei gesichert ist, das `A,B` positive ganze Zahlen sind:

```
01 mystery(_A , 0) -> 0;
02 mystery(A , B) when (B rem 2) == 0 ->
03                mystery(A+A,trunc(B/2));
04 mystery(A , B) when (B rem 2) == 1 ->
05                mystery(A+A,trunc(B/2)) + A.
```

(a) Wie lauten die Werte von `mystery(2,25)` und `mystery(3,11)`? Die Berechnung muss nachvollziehbar einen Bezug zu dem gegeben Programmcode haben!

(b) Beschreiben Sie, welchen Wert `mystery(X,Y)` bei gegebenen positiven Zahlen `X,Y` berechnet.

Begründen Sie Ihre Aussage durch einen direkten Bezug zum Programmcode.

(c) Beantworten Sie die Frage in (b) für den Fall, dass in Zeile `01` `mystery(_`A`, 0) -> 1;` ist und alle `+` (Zeilen `03, 05` durch ein `*` ersetzt werden.

Begründen Sie Ihre Aussage durch einen direkten Bezug zum Programmcode.

11. **Array** (?? Punkte)
Schreiben Sie in Pseudo-Code (oder Erlang/OTP) eine Funktion

$$\text{histogram} : \text{array} \times \text{int} \rightarrow \text{array} ,$$

die ein mit ganzen positiven Zahlen gefülltes Array `a[]` und eine positive ganze Zahl `m` als Argumente übernimmt und als Resultat ein Array der Länge `m` zurückliefert, dessen i-ter Eintrag die Anzahl der Vorkommen der Zahl i im Argumentarray angibt. Wenn z.B. die Werte im Argumentarray alle zwischen 0 und $m - 1$ liegen, sollte die Summe der Werte in dem Resultatsarray gleich der Länge dieses Arrays sein. Sollten im Argumentarray nicht abbildbare Werte vorkommen, sind diese zu ignorieren.

Begründen Sie in Bezug auf Ihren Code, warum Ihre Funktion korrekt arbeitet und geben Sie eine Abschätzung der Komplexität Ihrer Funktion an.

Kapitel 3

Komplexität von Algorithmen

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. Dabei handelt es sich um den Charakter des Algorithmus! Wie sich das konkret auf die Laufzeit eines Algorithmus auf einem konkreten Rechner auswirkt hängt von den jeweiligen Randbedingungen ab.

Wir wollen die Komplexität durch die Betrachtung eines Algorithmus zur Bestimmung von Primzahlen motivieren, der im übrigen von Ihnen insbesondere dann verwendet werden könnte, wenn Sie im Internet eine sichere Übertragung gewährleisten wollen.

3.1 Beispiel

Nachfolgend ein erster Algorithmus zur Bestimmung aller Primzahlen kleiner gleich $N \in \mathbb{N}$.

Algorithmus 3.1 Primzahltest mit der Divisionsmethode

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 *und gehe nach 4. sonst gehe nach 3.*
3. *Fahre fort mit dem nächsten* $j = j + 1$ *(möglicher Teiler) solange* $j \leq N$ *und gehe nach 2.*
4. *Fahre fort mit dem nächsten* $i = i + 1$ *(mögliche Primzahl) solange* $i \leq N$ *und gehe nach 2.*

5. alle nicht-gestrichenen Zahlen größer als 1 *sind die gesuchten Primzahlen*

Dieser Algorithmus hat auf offensichtlich mehrere Verbesserungsmöglichkeiten. Dennoch wollen wir zunächst diese einfache Implementation ansehen.

```
primzahlen(int N)
{
      bool a[N];
      for ( int i = 0; i < N ; i++)
           a[i] = true;
      for ( i = 2; i < N; i++)
      {
           for ( int 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
```

Aufgabe 3.1 *Wie oft wird hier die Division mit % ausgeführt? 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.*

Es gibt folgende Verbesserungsmöglichkeiten für das beschriebene Verfahren:

- 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? Betrachten Sie $i = p * q$ und die Situationen $p > \sqrt{i}$ bzw. $p < \sqrt{i}$.
- Die äußere Schleife über i muß nur über alle ungerade i laufen. Es könnte also heißen: `for ( int i = 3; i < N ; i+=2)`.
- 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?

Auf einem Laptop kann man für den nicht-verbesserten Algorithmus folgende Laufzeiten (interner Timer auf Sekunde genau) erhalten:

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

Hier kann man eine quadratische Abhängigkeit erkennen.

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++)
          irgendwas;
}
```

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. Dies gelingt uns erst mit dem folgenden Algorithmus:

Algorithmus 3.2 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: `sieb(int N)`

```
{
     bool* a = new bool(N);
     for(int i=0 ; i < N ; i++)
          a[i] = 1;
     for(int i=1 ; i < sqrt(N) ; i++)
     {
          if ( a[i] == true )
          {
               for (int j = 2 ; i*j < N ; j++)
                    a[i*j] = false;
          }
     }
}
```

Eine empirische Untersuchung auf einem Laptop ergibt 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 $\frac{N}{i}$, da $j * i \leq N \rightarrow j \leq \frac{N}{i}$, dies allerdings nur, wenn nicht schon durch `if (a[i])` vorher ausgefiltert wurde. Durch dieses Filtern wird eine 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++)
          irgendwas;
}
```

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.

Wir fassen die Resultate zusammen:

- 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 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.

[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.

- 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.

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

$$n = \ln N,$$

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^n ersetzen

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

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

3.2 Komplexitätsanalyse

In dieser Vorlesung und speziell 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

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

Das Thema soll hier nicht weiter vertieft werden. Um eine kleine Vorstellung von algorithmischen, theoretisch aber nicht lösbaren Problemen zu geben, sei folgender Algorithmus gegeben: Erstelle eine Datenbank aus dem Dorf X für den Barbier J.R. Hacker, selbst in X wohnend, der alle Männer des Dorfes rasiert, die sich nicht selbst rasieren. Wird Ihre Implementierung den Barbier selbst in die Datenbank eintragen ? Sie sehen, man kann also einfach in Algorithmen Widersprüche formulieren, die leider nicht immer so einfach zu entdecken sind, aber zu einem theoretisch nicht lösbarem algorithmischem Problem führen.

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 schwierig 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 3.1 *Betrachten wir zwei Code-Segmente:*

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

```
for i := 1 to n do
   for j := 1 to n do
       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 3.2 *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, könnte man also nach folgenden Kriterien versuchen zu bestimmen:

- Minimale Laufzeit
- Eine garantierte maximale Laufzeit (hartes Echtzeitproblem)
- Minimaler Hauptspeicherverbrauch
- Minimaler Festplattenspeicherverbrauch
- Minimale Netzbelastung
- Möglichst gut wartbar
- Möglichst universell einsetzbar
- Möglichst gut intuitiv verständlich
- möglichst korrekt **und** vollständig (bei Approximations Algorithmen)

Um einen Algorithmus zu untersuchen, kann man

- theoretische Untersuchungen machen (Komplexitätsanalyse)
- empirische Untersuchungen machen (Stochastik)

Mit theoretischen Untersuchungen werden wir uns im folgenden Beschäftigen. Man vergibt ein Qualitätskriterium, welches sich Resourcenfunktion nennt, und untersucht dieses in Abhängigkeit der Eingabegrösse.

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. Dies wird im Praktikum vertieft.

3.2.1 Aufwandsfunktion

Im Allgemeinen ist eine Ressourcenfunktion zu betrachten, die Speicherplatz und Laufzeit abschätzt. In den meisten Fällen beschränkt man sich auf die Laufzeit, da diese meist die Komplexitätsklasse wesentlich bestimmt.

Definition 3.1 *Eine Aufwandsfunktion $T(N)$ ist eine Funktion der Problemgröße N, welche die Laufzeit eines Algorithmus abschätzt.*

Da die Laufzeit eines Algorithmus natürlich unterschiedlich sein wird auf unterschiedlichen Plattformen, zählt man oft die arithmetischen Operationen, Vergleiche, Zuweisungen, und Funktionsaufrufe. 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
$N - \log N$	$N \log N$
quadratisch	N^2
kubisch	N^3
polynomiell	$N^4, N^5, N^6, ...$
exponentiell	$2^N, 3^N, ...$

3.2.2 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**.

$\mathcal{O}(g)$-Notation

Definition 3.2 *($\mathcal{O}(g)$-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*

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

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

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

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

In der Literatur hat sich jedoch die „Gleichung "$f = \mathcal{O}(g)$durchgesetzt, die eigentlich nur von links nach rechts zu lesen ist.

Beispiel 3.3 *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 3.1 *Seien f und g positive reellwertige Funktionen, so gilt*

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

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

Beispiel 3.4 *$T(n) := 36n^2 - 3n + 7$. 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 3.2 *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 $1 < b$. Die Notation $\mathcal{O}(1)$ ist etwas schlampig, aber durchaus üblich, und suggeriert eine Funktion $f(n) = 1$, also eine konstante Funktion. Intuitiv gesprochen ist jede Funktion in $\mathcal{O}(1)$entweder monoton fallend oder konstant. Da wir aber nur monoton steigende Resourcenfunktionen betrachten, bleiben nur die konstanten Funktionen übrig. Algorithmen mit einer Laufzeit $\mathcal{O}(1)$ laufen also immer gleich lang, egal was man eingibt.

Aufgabe 3.3 *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:

```
{
      komplizierterAlogrithmus.erstellen();
      komplizierterAlogrithmus.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 3.4 *Beweisen Sie, dass gilt* $\mathcal{O}(f) + \mathcal{O}(f) = \mathcal{O}(f)$.

Antwort: 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++ )
      {
           komplizierterAlogrithmus.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_0s von f und g.

Beispiel 3.5 *In folgendem Programmcode interessiert uns 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 3.5 *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. $(n+1)! = \mathcal{O}(n!)$

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

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.

$\Omega(g)$-Notation

Definition 3.3 *(Groß-Omega)*

$$f = \Omega(g) :\Longleftrightarrow \exists\, n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \quad \textit{so dass } \forall\, n \geq n_0 \;:\; f(n) \geq c \cdot g(n).$$

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

Satz 3.2 *Seien $f, g : \mathbb{N} \to \mathbb{R}_0^+$ beliebige Funktionen. Dann gilt*

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

Alle weiteren Eigenschaften der $\mathcal{O}$-Notation gelten (u.U. „umgedreht“) auch für die Ω-Notation.

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

$\Theta(g)$-Notation

Definition 3.4 *(Groß-Theta oder Exakte Ordnung)*

$$f = \Theta(g) :\Longleftrightarrow \exists\, n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}^+ \quad \textit{so dass } \forall\, n \geq n_0 \;:\; c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n).$$

Es gilt folgende Eigenschaft der exakten Ordnung.

Satz 3.3

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

Beispiel 3.6 *Es gilt*

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

denn

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

Letzteres gilt wegen

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

mit Anwendung des Satzes 3.2.

In der Literatur hat sich folgender Gebrauch der Notationen durchgesetzt: Eine Aussage „die Laufzeit ist $\mathcal{O}(f(n))$„ bedeutet, dass $f(n)$ die kleinste obere Schranke der Laufzeit des Algorithmus darstellt. Eine Aussage „die Laufzeit ist $\Omega(f(n))$„ bedeutet, dass $f(n)$ die größte untere Schranke der Laufzeit des Algorithmus darstellt.

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.

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.

Einen kleinen Überblick über das Wachstum bestimmter Komplexitätsfunktionen sollte der Leser sich in einer Tabelle darstellen, etwa n, ln n, n log n, n^2, 2^n und n!.

Aufgabe 3.6 *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.

3.3 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 eine Exponentialfunktion.

Aufgabe 3.7 *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?

3.4 Rekursive Algorithmen

Im Allgemeinen 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:

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

   Stichwort: Horner Schema.

2. Die Berechnung von $N!$ kann auf die Berechnung von $(N-1)!$ zurückgeführt werden

$$N! = N \cdot (N-1)!.$$

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)

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

4. Die „Türme von Hanoi“ ist ein mit drei Stangen und einem Satz von unterschiedlich grossen Scheiben gespieltes Spiel. Die Scheiben werden nach ihrem Durchmesser geordnet und auf die Stangen mittels eines durch das Zentrum von jeder Scheibe gebohrten Lochs aufgesteckt. Das Ziel des Spiels ist, alle Scheiben von der linken Stange (auch als Quelle bezeichnet) auf die mittlere Stange (Ziel) zu versetzen. Die rechte Stange (TMP) kann als eine „Ersatz-“Stange - eine temporäre Stelle für die Scheiben - benutzt werden.

   Jedes Mal, wenn eine Scheibe von einer Stange auf eine andere versetzt wird, müssen zwei Spielregeln beachtet werden:

   (a) Nur die oberste Scheibe auf einer Stange kann versetzt werden, und

   (b) keine Scheibe kann auf eine kleinere Scheibe gesetzt werden.

   Am Anfang sind die Scheiben ihrer Größe nach auf der linken Stange, der Quelle, aufgesteckt: die kleinste ganz oben! Die Aufgabe ist nun, einen Turm der Höhe N (etwa $N = 64$ im Originalbeispiel) von einer Stange (Quelle) zu einer zweiten Stange (Ziel) unter Benutzung einer dritten Stange (TMP) gemäs den Spielregeln zu bewegen.

   Hier eine einfache Strategie, die das Spiel der „Türme von Hanoi“ mit drei Stangen und `N` Scheiben sicher spielt:

   - Das Spielende ist, wenn keine Scheiben mehr auf der linken Stange - der Quelle - stecken und alle Scheiben auf der mittleren Stange - dem Ziel - sind.
   - (a) Sonst sind $N - 1$ Scheiben von der linken Stange (Quelle) auf die rechte Stange (TMP) mit Hilfe der mittleren Stange (Ziel) zu versetzen. Dies ist eine rekursive Strategie!
     (b) Die N-te einzelne Scheibe (also die aktuell größte Scheibe) kann dann von der linken Stange - der Quelle - auf die mittlere Stange - dem Ziel - gesetzt werden.
     (c) Schließlich müssen die $N-1$ Scheiben von der rechten Stange (nun Quelle) auf die mittlere Stange (Ziel) mit Hilfe der linken Stange (nun TMP) versetzt werden.

   Eine mögliche Implementierung im Quasi-Code könnte wie folgt aussehen:

   Algorithmus 3.3 Türme von Hanoi
   Modul hanoi(n, Quelle, Ziel, TMP)

   (a) **falls** $n = 1$ **dann** *bewege Scheibe von Quelle zum Ziel*
   sonst

   i. hanoi(n-1, Quelle, TMP, Ziel)

ii. bewege 1 Scheibe von Quelle zum Ziel

iii. hanoi(n-1, TMP, Ziel, Quelle)

Der Leser möge das Programm in irgendeiner Sprache implementieren. Die Bewegung einer Scheibe kann am besten per `print`-Kommando auf dem Bildschirm ausgegeben werden. Dazu sollten die Stangen als Zeichenketten implementiert werden und mit „Quelle, Ziel“ und „TMP“ bezeichnet werden. Auf dem Bildschirm sind dann die Anweisungen für einen Spieler abzulesen.

Da das Rekursionsprinzip und die mathematische Induktion der selben Quelle entspringen, hier beispielhaft ein Beweis, der per Induktion zeigt, dass man das Spiel „Türme von Hanoi“ mit beliebig vielen Scheiben spielen kann. Zu beachten ist, dass bereits bei 64 Scheiben man ca. 600.000.000.000 Jahre Zeit bei einer Sekunde pro Bewegung einer Scheibe ($2^{64} - 1$ Sekunden!) mitbringen sollte.

Beweis durch vollständige Induktion über N.

Induktionsanfang Sei $N = 1$, dann wird der `dann`-Zweig in Schritt (a) ausgeführt. Da die einzige Scheibe von der Quelle auf das Ziel bewegt wird, arbeitet das Programm daher für $N = 1$ korrekt.

Induktionsbehauptung Die Behauptung sei für ein N gezeigt, d.h. `hanoi(N,"Quelle","Ziel","TMP")` arbeitet für irgendein N korrekt. Aus Sicht der Rekursion: der rekursive Aufruf arbeitet korrekt.

Induktionsschritt Der Aufruf für $N + 1$ (`hanoi(N+1,"Quelle","Ziel","TMP")`) arbeitet wie folgt:

(a) Zunächst wird `n = N+1` auf `n-1 = N` „gesetzt“ bzw. damit werden die rekursiven Aufrufe durchgeführt.

(b) `hanoi(n-1,"Quelle","TMP","Ziel")` arbeitet korrekt nach Induktionsbehauptung.

(c) `bewege...` bewegt eine Scheibe, d.h. die aus Sicht der Rekursion letzte und größte Scheibe von der Quelle auf das Ziel, was dem korrekten Spielzug entspricht.

(d) `hanoi(n-1,"TMP","Ziel","Quelle")` arbeitet korrekt nach Induktionsbehauptung.

Daher arbeitet `hanoi(N,"Quelle","Ziel","TMP")` für alle $N \in \mathbb{N}$ korrekt.

In einem kleinen Nebenbeweis wäre noch explizit zu zeigen, dass die Stangen korrekt bei dem rekursiven Aufruf bezeichnet sind. Das war hier vorausgesetzt.

Um rekursiv definierte Probleme in der Informatik zu lösen, verwendet man 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, also das N-te Element wird behandelt und der Übergang zu dem $N-1$-ten Element. Natürlich benötigt man noch eine Abfrage, ob $N = 1$ (oder 0) ist. Die Routine ruft sich selbst also $N-1$ mal auf. Die Denkweise ist dann prinzipiell einfach: Statt in einer Iteration alle Elemente zu behandeln, behandelt man nur ein Element, kümmert sich um die Verbindung zu den anderen Elementen und lässt dann den Rest durch einen rekursiven Aufruf bearbeiten. Denn schließlich hat man ja eine Funktion implementiert, die N-Elemente bearbeiten kann – also warum diese nicht selbst einsetzen ? Der induktive Beweis entspringt übrigens der gleichen Quelle, wie im Beispiel zu den „Türmen von Hanoi“ gezeigt. Der Induktionsanfang ist dabei vergleichbar mit der Behandlung eines Elementes, hier, indem für z.B. $n = 1$ die Aussage direkt gezeigt wird. Die Induktionsbehauptung spiegelt den rekursiven Aufruf dar: man tut so, als würde die gerade sich in der Implementierungsphase befindende Funktion schon lauffähig sein, hier also tut man so, als wäre die Aussage für irgendein Element gezeigt. Der Induktionsschritt ist dann die Invariante bzw. das Zusammenbringen des einzelnen Elements und der „anderen“ Elemente, hier zeigt man also, das beim Übergang vom n-ten Element zum $n+1$-ten Element die Aussage nicht verloren geht.

Bei der Rekursion sollte man beachten, dass alle errechneten Zwischenergebnisse üblicherweise auf dem BS-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 eine Endrekursion (die, sofern der Compiler „schlau“ ist, ohne Stack auskommt) oder ein iteratives Verfahren und holt sich den erforderlichen Speicher vom Heap (über *new*). Jede Rekursion läßt sich in ein iteratives Verfahren umschreiben und umgekehrt, wenn auch dies nicht immer mal eben so gemacht werden kann.

Beispiel 3.7 *Die beiden Begriffe Rekursion und Iteration sollen anhand der Fakultätsfunktion motiviert werden. Diese Funktion ist u.a. definiert als*

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

Diese Definitionen zeigen schon die unterschiedlichen Darstellungsmöglichkeiten 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. Die linke Definition Π stellt eine Iteration dar, die z.B. mittels eine `for`*-Anweisung leicht realisiert werden kann.*

In den folgenden beiden Abschnitten wird die Analyse rekursiver Algorithmen bzgl. ihrer Laufzeit untersucht. Dies wird darauf hinauslaufen, in einer rekursiven

Gleichung die Rekursion aufzulösen.

Rekurrenz

Die im folgenden beschriebene Rekurrenz (siehe hierzu `Sedgewick`) entsteht bei einem rekursiven Programm, das die Eingabedaten in einer Schleife verarbeitet, um jeweils ein Element zu entfernen:

$$C_N = C_{N-1} + N \text{ für } N \geq 2 \text{ mit } C_1 = 1.$$

C_N ist ungefähr $\frac{N^2}{2}$. Um eine derartige Rekurrenz aufzulösen, ziehen wir sie auseinander, indem wir sie folgendermaßen auf sich selbst anwenden:

$$C_N = C_{N-1} + N = C_{N-2} + (N-1) + N = C_{N-3} + (N-2) + (N-1) + N = \ldots$$

Wenn wir in dieser Weise fortfahren, erhalten wir schließlich:

$$C_N = C_1 + 2 + \ldots + (N-2) + (N-1) + N = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = \frac{N * (N+1)}{2}$$

Die Berechnung der Summe ist elementar („Gaußsche Summenformel“): Das angegebene Ergebnis erhält man, wenn man die Summe zu sich selbst addiert, allerdings Term für Term in umgekehrter Reihenfolge. Dieses Ergebnis - doppelt so groß wie der gesuchte Wert - besteht aus N Termen, die N+1 als Summe liefern.

Diese Rekurrenz entsteht für ein rekursives Programm, das die Eingabe in jedem Schritt halbiert:

$$C_N = C_{N/2} + 1 \text{ für } N \geq 2 \text{ mit } C_1 = 1$$

C_N ist ungefähr ldN. In der angegebenen Form ist diese Gleichung nur sinnvoll, wenn N gerade ist oder wir annehmen, dass $\frac{N}{2}$ eine ganzzahlige Division bezeichnet. Fürs Erste gehen wir von $N = 2^n$ aus, sodass die Rekurrenz stets genau definiert ist. (Beachten Sie, dass $n = ldN$ gilt.) Dann aber lässt sich die Rekurrenz noch einfacher als unsere erste Rekurrenz auseinander ziehen:

$$C_{2^n} = C_{2^{n-1}} + 1 = C_{2^{n-2}} + 1 + 1 = C_{2^{n-3}} + 3 = \ldots = C_{2^0} + n = n + 1$$

Die exakte Lösung für allgemeines N hängt davon ab, wie man $N/2$ interpretiert. Falls $N/2$ den Ausdruck $\lfloor N/2 \rfloor$ darstellt, haben wir eine einfache Lösung: C_N ist die Anzahl der Bits in der binären Darstellung von N und diese Zahl ist gemäß Definition gleich $\lfloor ldN \rfloor + l$. Dieser Schluss folgt sofort aus der Tatsache, dass eine Ganzzahl $N > 0$ in den Wert $\lfloor N/2 \rfloor$ umgewandelt wird, wenn man das am weitesten rechts stehende Bit in der binären Darstellung dieser Zahl eliminiert.

Wir wollen nun am Beispiel der Fibonacci-Zahlen den Zeitaufwand für eine komplexere und konkrete Berechnung durchführen.

Fibonacci-Zahlen

Die Fibonacci-Zahlen treten immer wieder auf bei der Analyse von verschiedenen Algorithmen Sie sind folgendermaßen definiert[4]:

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

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

Algorithmus 3.4 `fibRec(int N)`

```
{
      switch(N)
      {
      case 0:
           return 0;
      case 1:
           return 1;
      default:
           return 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:

[4]Im Allgemeinen: Fibonacci-Zahlen k-ter Ordnung: $fib_k(n) = 0$, falls $0 = n = k - 2$; $fib_k(1) = 1$, falls $n = k - 1$; $fib_k(n) = fib_k(n-1) + fib_k(n-2) + \ldots + fib_k(n-k)$;

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 3.4 *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 $c(1) = 0 = f(2) - 1$.
Induktionsbehauptung. Es sei angenommen, dass $N \geq 2$ und dass folgendes bereits bewiesen ist:

$$c(N) = f(N+1) - 1,$$
$$c(N-1) = f(N) - 1.$$

Induktionsschritt. Es wird gezeigt, dass die Aussage für $c(N+1)$ gilt.

$$\begin{aligned} c(N+1) &\overset{Def.\ vonc}{=} c(N) + c(N-1) + 1 = \\ &\overset{Ind.Beh.}{=} f(N+1) - 1 + f(N) - 1 + 1 = \\ &\overset{ausrechnen}{=} f(N+1) + f(N) - 1 = \\ &\overset{Def.\ vonf}{=} f(N+2) - 1. \end{aligned}$$

□

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 3.5 *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

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

Wollen wir zum Beispiel wissen, wie viele Additionen $c(n)$ für 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 – 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:

Algorithmus 3.5 `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 verwenden kann nennt sich *Divide and Conquer*, also zu deutsch etwa Teile–und–Herrsche. Es wird rekursiv implementiert. Damit kann man die Aufwandsordnung eines Algorithmus tatsächlich verbessern. Der „Altmeister“ Knuth hatte sogar diejenigen sinngemäß als „einfältig“ bezeichnet, die iterativ implementieren. Letztlich sollte man keine Dogmen verbreiten, da es eben mal so und das andere mal so sein kann. Bei der Berechnung von Polynomen ist z.B. das rekursiv definierte Horner Schema wesentlich schneller und exakter, als das iterative Verfahren!

3.5 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

  Im Beispiel: Möchte der Kunde wirklich alle Stellungen des Schachspiels oder evtl. nur des Tic-Tac-Toe Spiels ?

- **Erzwungene Einschränkung** des Problems: „nur“ gutmütige Probleme oder Spezialfälle lassen sich aufschreiben bzw. werden gelöst. Der Auftraggeber muss darauf hingewiesen werden.

  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“).

  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.

  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.

- **Zeitbeschränkte Exploration** des Suchraums: Größe der Instanzen, die in zumutbarer Zeit lösbar sind.

  Im Beispiel: Es werden ausgehend von einer Spielsituation nur alle Stellungen berechnet, die in den nächsten vier Zügen möglich sind.

- **Veränderung** des Algorithmus: Mögliche Redundanzen vermeiden bzw. verkleinern und ggf. andere Operationen für die Problemlösung verwenden.

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

Die letzten Punkte sprechen Probleme an, die in dem Sinne wirklich „nicht effizient lösbar“ sind. In einem solch Fall, meist beschreibbar durch einen super grossen Suchraum, indem die optimale Lösung liegt, kann praktisch bzw. effizient

die optimale Lösung nicht gefunden werden. Es 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).

3.6 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.

3.7 Aufgaben

1. **Begriffsbestimmung** (? Punkte)
   Beschreiben Sie die drei interessanten Fälle **Bester Fall**, **Schlechtester Fall** und **Mittlerer Fall**.

2. **Problembehandlung** (? Punkte)
   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.

3. **MinMax** (?? Punkte)
   Sei $A[0 \ldots n-1]$ ein Array mit natürlichen Zahlen. Geben Sie in Pseudo-Code (oder Erlang/OTP) einen Algorithmus an, der mit maximal $\frac{3*n}{2}$ Elementvergleichen ($\frac{3*n}{2} + 0,5$, falls n ungerade) das Maximum und das Minimum in A bestimmt. Begründen Sie, warum Ihr Algorithmus diese Grenze einhält.

4. **Min-Sort** (? Punkte)
   Gegeben sei der folgende Sortieralgorithmus, der eine Folge $A = (a_1, \ldots, a_n)$ aufsteigend sortiert.

```
 MinSort(A,n)
(1) for i = 1 to n do
(2)     m = i
(3)     for j = i + 1 to n do
(4)         if A[m] > A[j]
(5)            then m = j
(7)     tmp = A[i], A[i] = A[m], A[m] = tmp
```

   (a) Analysieren Sie die Komplexitätsklasse des Algorithmusfür den Worst-Case.

   (b) Zeigen Sie die Korrektheit des Algorithmus. Hinweis: Sie benötigen eine Schleifeninvariante für die innere Schleife und eine für die äußere Schleife.

5. **Break-even Punkt** (? Punkte)
   Für die Aufgabe, n Adressen zu sortieren, stehen drei Verfahren zur Verfügung: Eine Implementierung von Heapsort benötigt eine Zeit von

$192 * log_2(n)\mu s$, eine Implementierung von Insertion Sort genau $2 * n^2 \mu s$ und eine Implementierung von Bubblesort $n^2 + 8n + 180$ μs. Für welche Datenbankgrössen n lohnt sich welches der Sortierverfahren? Bestimmen Sie die „Break-even"-Punkte! Für den Break-even-Punkt zwischen Bubblesort und Heapsort kann kurz vor der formalen Auflösung (bzgl. log_2) mit den beiden Werten $n = 22$ und $n = 23$ getestet werden.

6. **Aufwandabschätzung** (? Punkte)
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.

7. **Aufwandabschätzung** (? Punkte)
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 ; }
}
```

(a) 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!

(b) 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)!

(c) Auf welchem algorithmischen Konstruktionsprinzip basiert die Funktion `foo` ? Denken Sie auch hier an die Begründung Ihrer Antwort!

(d) 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 $n = 2^k, k \in \mathbb{N}_0$ ist.

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

8. **Aufwandabschätzung** (? Punkte)
Gegeben seien folgende Codefragmente; n sei eine positive natürliche Zahl.

```
algorithm alg1(n)
var prod;
begin prod := 1;
      for i := 1 to n do
          prod := prod*i;
      endfor
      return prod;
end alg1;
```

```
algorithm alg2(n)
var p;
begin p := 1;
      while n > 1 do
         p := p*n;
         n := n/3;
      endwhile
      return p;
end alg2;
```

```
algorithm alg3(n)
var p,r;
begin p := 1; r:= 0;
      for i := 1 to n do
          for j := 1 to p do
              r := r + 1;
          endfor
          p := p*2;
      endfor
      return r;
end alg3;
```

```
algorithm alg4(n)
var p,t;
begin p := 1;
      for i := 0 to n do
          for j := i to n do
              t := n;
              while t > 0 do
                 p := p + 1;
                 t := t/2;
              endwhile
          endfor
      endfor
      return p;
end alg4;
```

Geben Sie für diese Algorithmen in Abhängigkeit von n eine Laufzeitkomplexität an. Begründen Sie kurz Ihre Entscheidung. Geben Sie anschließend bzgl. der Komplexität eine Rangordnung der Algorithmen an.

9. **Lesen und Schreiben** (?? Punkte)
Gegeben seien die Datenstrukturen Array A der Länge n, sowie eine einfach verkettete Liste EL der Länge n und eine doppelt verkettete Liste DL der Länge n. Der Head-Zeiger von EL zeigt auf den Anfang der Liste. Der Head-Zeiger von DL zeigt auf den Anfang der Liste, der Tail-Zeiger auf das Ende der Liste DL. Welche Zeitkomplexität haben die folgenden Operationen im Sinne der O-Notation bei den jeweiligen Datenstrukturen (Begründung nicht vergessen!):

   (a) Lesen des ersten und letzten Elementes ?

   (b) Überschreiben eines Elementes, welches sich genau in der Mitte der Datenstruktur befindet ?

Kapitel 4

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 Knuth:1997c 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!

4.1 Definition 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 Arbeitsspeicher. Es wird berücksichtigt, dass die Datenzugriffe besonders teuer sind.

Wir nehmen an, dass jeder Datensatz a_i einen Schlüssel $k_{\pi(i)}$ 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 entweder

$$k_i \leq k_j \text{ oder } k_j \leq k_i$$

gilt.

Sei N die Anzahl der zu sortierenden Datensätze. Wir gehen davon aus, dass wir eine Generierungsfunktion für die zugehörigen Schlüssel $k_{\pi(i)}$ haben, die in irgendeiner Weise eine sinnvolle Sortierung der Datensätze durch eine geeignete Abbildung auf diese Schlüssel herstellt. Dies ist letztlich eine Permutation (bijektive Abbildung) der Datensätze

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

Sollte es nicht sinnvoll sein, die Datensätze in irgendeiner Weise zu sortieren, sollte man von einer Sortierung der Schlüssel absehen. In dieser Problemformulierung sind aber absichtlich viele Details offen gelassen, die für die Lösung durchaus wichtig sind. 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 können Zusatzinformationen den Algorithmus zum Sortieren im Sinne der Effizienz entscheidend beeinflussen. Solche Zusatzinformationen können sein:

- Art der Abspeicherung der Datensätze (extern: CDROM, Magnetband; intern: Hauptspeicher)
- Eingabe der Datensätze (inkrementell: seriell über tcp/ip ein Datensatz pro Sekunde; nicht inkrementell: alle Datensätze im Array verfügbar)
- Art der Schlüssel (p-adische Zahlen z.B. 2-adische Zahlen sind die binäre Darstellung, natürliche Zahlen (sind 10-adische Zahlen))
- Eigenschaften der Schlüssel (lückenlos, gleichverteilt, Max und Min bekannt, Rechenoperationen erlaubt (siehe Radix-Sort Verfahren))
- Eigenschaften der Datensätze (doppelte erlaubt)
- 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 meist, möglichst viel Informationen über die Daten im Sortierverfahren mit zu verwenden.

Bemerkung 4.1 *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 propritäre, also z.B. problemspezifische Lösung, immer die bessere ist oder eine universale, 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), 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

```
class Datensatz
{
      int key;
      DAT daten;
}
Datensatz a[N];//wir zählen im folgenden von 1 bis N
```

- alle Datensätze im Hauptspeicher vorhanden
- Schlüssel sind natürliche Zahlen
- doppelte Datensätze erlaubt (also zwei Datensätze mit gleichen Schlüsseln)
- 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

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

Zur Analyse der Algorithmen wollen wir aber annehmen, dass die Datensätze tatsächlich kopiert werden oder ggf. durch eine Adresse in dem Feld repräsentiert werden. Die Eingabe des unsortierten und Ausgabe des sortierten Datensatzes geschieht also durch

$$a[1], \ldots, a[N]$$

Definition 4.1 *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 4.2 *Wir definieren die Routine* `swap(i,j)`*, welche den i-ten und j-ten Datensatz vertauscht.*

```
swap(i,j)
{
      Datensatz tmp;
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
}
```

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

Definition 4.3 *Ein Sortierverfahren heißt stabil, wenn Objekte mit gleichem Schlüssel ihre relative Position nicht verändern, wenn also gilt*

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

Definition 4.4 *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.

4.2 Elementare Sortierverfahren

Wenn die Anzahl der Datensätze und deren jeweiliger Umfang sich in Grenzen halten, kann man alle Datensätze im Arbeitsspeicher eines Computers sortieren: Man spricht dann von einem internen Sortiervorgang (engl. internal sort). Diese werden in diesem Abschnitt behandelt. Für diese Verfahren ist typisch, dass $\Theta(N^2)$ Vergleichsoperationen von Schlüsseln im schlechtesten Fall ausgeführt werden müssen.

Können nicht alle Datensätze gleichzeitig im Arbeitsspeicher gehalten werden, dann muss ein anderer Sortieralgorithmus gefunden werden: Man spricht dann von einem externen Sortiervorgang (engl. external sort).

Das Problem des externen Sortierens lässt sich auf das Problem des internen Sortierens zurückführen

4.2.1 Sortieren durch Auswahl (Selection Sort)

Methode Nehmen wir als Beispiel ein 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 4.1 *Die unsortierten Karten mögen aussehen wie folgt:*

i:	1	2	3	4	5	6	7	8
a_i:	♣4	♠A	♣K	♡8	♢D	♡D	♡A	♣10

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:	♢D	♠A	♣K	♡8	♣4	♡D	♡A	♣10

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

i:	1	2	3	4	5	6	7	8
a_i:	♢D	♡8	♣K	♠A	♣4	♡D	♡A	♣10

Im nächsten Schritt werden Positionen 3 und 6 vertauscht:

i:	1	2	3	4	5	6	7	8
a_i:	♢D	♡8	♡D	♠A	♣4	♣K	♡A	♣10

Danach Positionen 4 und 7:

i:	1	2	3	4	5	6	7	8
a_i:	♢D	♡8	♡D	♡A	♣4	♣K	♠A	♣10

Folgendes Programm realisiert diesen Algorithmus.

```
SelectionSort()
{
      int minimum = 0;
      for ( int i = 1; i ≤ N-1 ; i++ )
      {
           minimum = findMinStartingAt(i);
           swap(i,minimum);
      }
}
int findMinStartingAt(int i)
{
      int min = i;
      for ( int j = i+1; j≤ N; j++)
      {
           if(a[j].key < a[min].key)
                min = j;
      }
}
```

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 4.1 *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.

4.2.2 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 4.2 *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 erste Element ist das kleinste Element und wir sind fertig. Das zweite Element ist kleiner als das erste, womit wir die beiden vertauschen und erhalten

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

Nun betrachten wir die Karte auf Position 3. Diese steht bereits an der korrekten Stelle und wir betrachten die nächste Karte an Position 4. Diese wird an Position 1 gesetzt und alle anderen um eine Position nach rechts verschoben:

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

Im nächsten Schritt wird die Karte an Position 5 auf Position 1 gesetzt und alle anderen wieder verschoben:

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

Die Karte an Position 6 wird auf Position 3 gesetzt, alle rechts davon um eine Position verschoben:

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

und weiter:

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

Nun folgt das letzte Element: damit sind alle Elemente nach einander an die für die bis dahin sortierte Liste korrekte Position kopiert worden. Alle rechts davon stehenden, sortierten Elemente werden um eine Position nach rechts verschoben.

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

Folgendes Programm realisiert diesen Algorithmus.

```
   InsertionSort()
{
      for ( int i = 2; i ≤ N ; i++ )
      {
           int j = i;
           Datensatz t = a[i];
           int k = t.key;
           while(a[j-1].key > k) && (j > 1)
           {
                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

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

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

4.2.3 Shell Sort

Methode Shell Sort ist ein Verfahren, welches sich Sortieren durch Einfügen zu Hilfe nimmt, dieses jedoch auf bestimmte Teilfolgen anwendet. Diese Teilfolgen bestehen aus den Elementen

$$a[1], a[1+k_i], a[1+2k_i], a[1+3k_i], \ldots$$
$$a[1], a[1+k_{i-1}], a[1+2k_{i-1}], a[1+3k_{i-1}], \ldots$$
$$\text{usw.}$$

Dabei ist k_i 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.

Sind alle der oben aufgeführten Folgen für ein bestimmtes k_i sortiert, so sagt man die Folge ist k_i-sortiert. Leider ist die gesamte Folge dann noch nicht sortiert. Man wählt dann ein kleineres k_{i-1} und erhält neue Folgen die man wieder sortiert. Man fährt solange fort, bis $k_1 = 1$ ist. Diese Werte k_t bilden dann eine sogenannte Folge von abnehmenden Inkrementen

$$k_t \geq k_{t-1} \geq k_{t-2} \geq \ldots \geq k_1 \text{ wobei } k_1 = 1.$$

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

Die Frage stellt sich nun, welche k_t-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. Man nennt es auch *Sortieren mit abnehmenden Inkrementen.*

Beispiel 4.3 *Wir betrachten wieder unser Kartenspiel. Als k-Folge wählen wir*

$$5, 3, 1.$$

Die ursprüngliche Ordnung sei wieder wie folgt:

i:	1	2	3	4	5	6	7	8
a_i:	♣4	♠A	♣K	♡8	♢D	♡D	♡A	♣10

Die 5*-Folgen bestehen aus den Datensätzen an den Positionen*

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

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:	♡D	♡A	♣10	♡8	♢D	♣4	♠A	♣K

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

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

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:	♡8	♢D	♣4	♡D	♡A	♣10	♠A	♣K

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 InsertionSort jetzt leichter.

Beispiele für solche k_t-Folgen sind:

Shell 1, 2, 4, 8, 16, ... (das Original von Shell)

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

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

Pratt 27, 18, 12, 8, 9, 6, 4, 3, 2, 1 Komplexität etwa $\mathcal{O}(n(logn)^2)$ Berechnung: $k_{i,j} = 2^i * 3^j$, wobei $(i, j) : (0, 0); (1, 0), (0, 1); (2, 0), (1, 1), (0, 2); \ldots$

Da wir hier keine Analyse machen wollen, verzichte ich auf die Darstellung des Codes. Den kann man z.B. in `Ottmann` finden.

4.2.4 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 4.4 *Im Kartenspiel haben wir wieder*

i:	1	2	3	4	5	6	7	8
a_i:	♣4	♠A	♣K	♡8	♢D	♡D	♡A	♣10

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

i:	1	2	3	4	5	6	7	8
a_i:	♣4	♠A	♣K	♡8	♢D	♡D	♡A	♣10
:	♠A	♣4	♣K	♡8	♢D	♡D	♡A	♣10
:	♠A	♣4	♡8	♣K	♢D	♡D	♡A	♣10
:	♠A	♣4	♡8	♢D	♣K	♡D	♡A	♣10
:	♠A	♣4	♡8	♢D	♡D	♣K	♡A	♣10
:	♠A	♣4	♡8	♢D	♡D	♡A	♣K	♣10
:	♠A	♣4	♡8	♢D	♡D	♡A	♣10	♣K

Man sieht hier sehr schön, wie der ♣K 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 (Bidirektionales Bubblesort) 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. Combsort (Kamm Sortieren) beginnt mit weit auseinander liegenden Elementen (Gap = Lücke). Die Lücke wird dann immer kleiner. Ähnlich dem Shell Sort Konzept.

4.3 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, d.h. die Daten werden nur soweit sortiert (Prinzip *lazy*-Verarbeitung), dass das größte Element direkt ausgegeben werden kann; vor der nächsten Ausgabe muss zuerst die Heapbedingung wieder hergestellt werden! In jedem Fall sind (*worst-case optimal*) $\mathcal{O}(N \log N)$ Operationen nötig. 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 4.5 *Eine Folge $F = k_1, k_2, \ldots, k_N$ von Schlüsseln nennen wir einen* Heap *wenn*

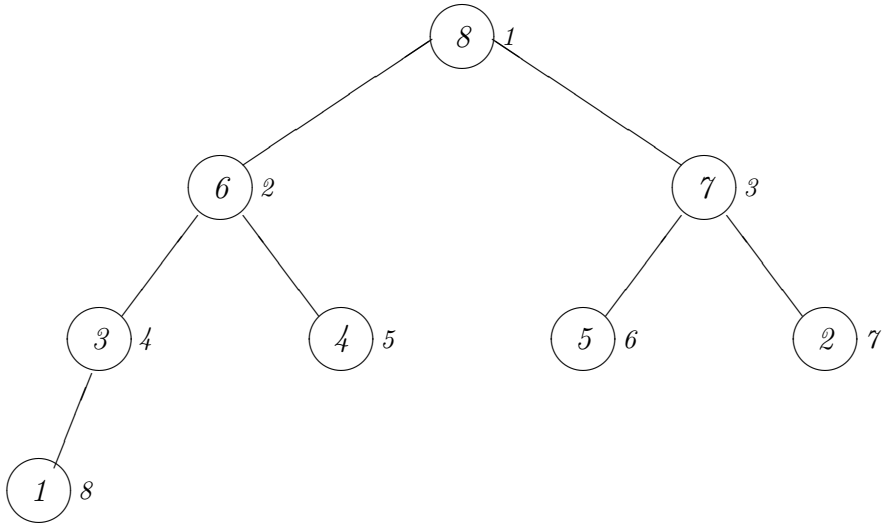
$$k_i \leq k_{\lfloor \frac{i}{2} \rfloor} \text{ für } 2 \leq i \leq N$$

gilt. Gleichbedeutend damit ist

$$k_i \geq k_{2i} \text{ und } k_i \geq k_{2i+1}, \text{ falls } 2i \leq N \text{ und } 2i+1 \leq N.$$

Beispiel 4.5 *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, in den Knoten steht der Schlüssel.

Das vollständige Sortieren mit Hilfe eines Heaps kann in zwei Phasen durchgeführt werden:

Phase 1 : Verwandeln des gegebenen Arrays in einen Heap

Phase 2 : Iteriertes Extrahieren des Maximums: Tauschen des Elementes mit höchstem Index im nicht sortierten Teil an Position1 und versickern lassen im Rest, ähnlich Bubblesort.

4.3.1 Top-Down

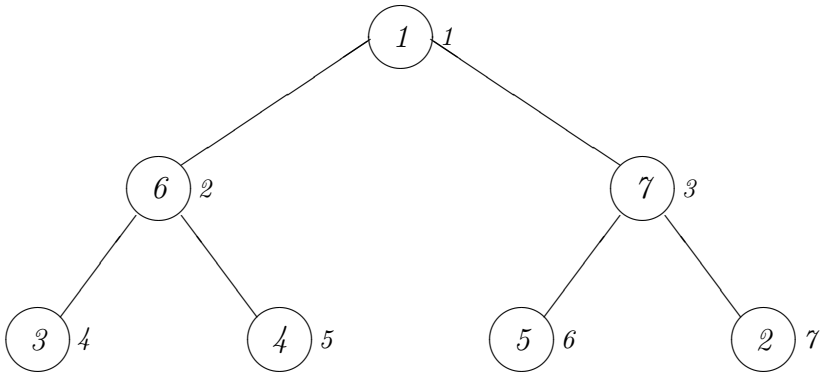
Die nachfolgende Methode ist eine Top-Down Methode, die neben dem hier gezeigten Löschen eines Elementes auch genutzt werden kann, wenn sich z.B. der Schlüssel (als Priorität interpretiert) ändert.

Methode Ist ein Heap gegeben, so ist die Ausgabe des größten Elemtens 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:

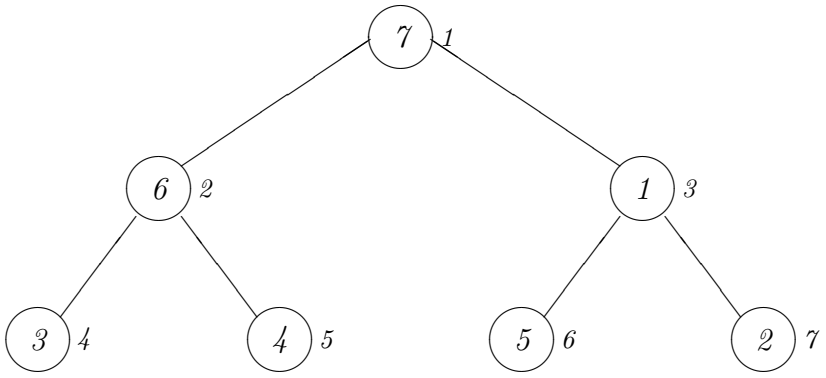
- schreibe das Element mit dem höchsten Index an die Wurzel des Baumes.
- vertausche dieses Element so lange mit dem größeren seiner Söhne, bis alle seine Söhne kleiner sind, oder bis es unten angekommen ist (man sagt, das Element *versickert*).

Dies ist im folgenden am Beispiel fortgeführt.

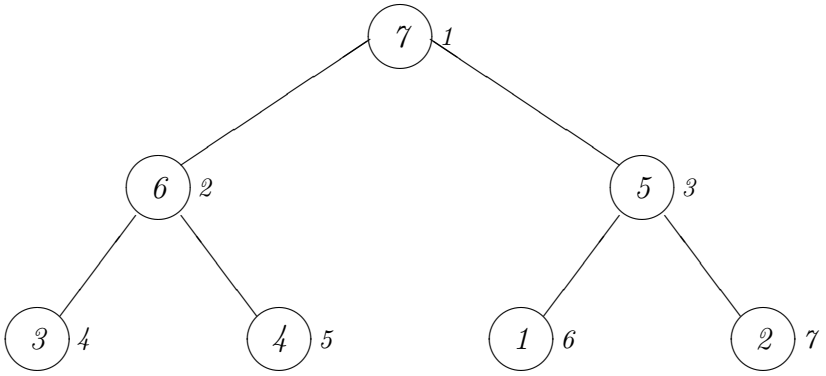
Beispiel 4.6 *Zunächst das Element mit dem höchsten Index an die Wurzel des Baumes schreiben:*



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:

Hier der Algorithmus:

```
int HeapEnde = N;

HeapSeep (int i,int HEnde) {
  while ((2*i) <= HEnde) {
    int j = 2*i;
    int kl = a[2*i].key;
    int kr = kl-1;
    if ((2*i+1) <= HEnde) {
        int kr = a[2*i+1].key; };
    if (kl < kr) j = j+1;
    if (a[i].key < a[j].key) {
    swap(i,j); i = j;
     } else { i = HEnde + 1; };
   } }
```

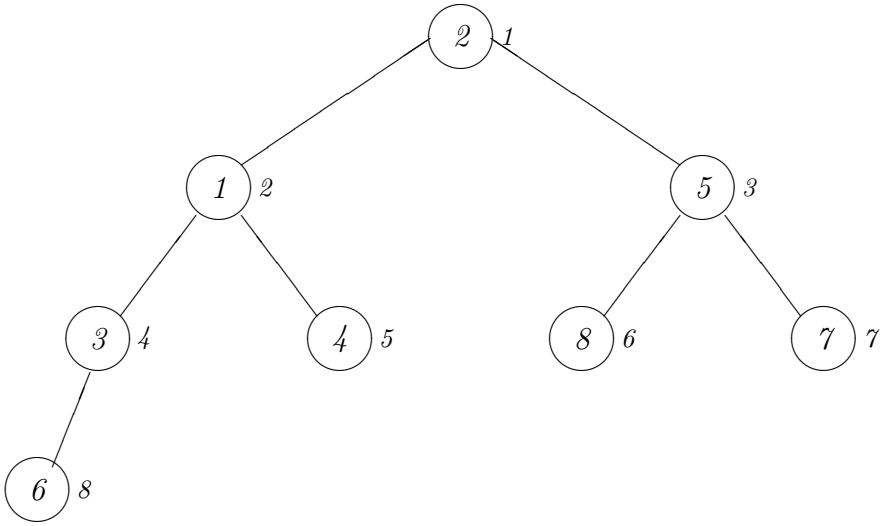
4.3.2 Bottom-Up

Methode Sei $F = k_1, \ldots, k_N$ eine Folge von Schlüsseln. Sie wird in einen Heap umgewandelt, indem die Schlüssel

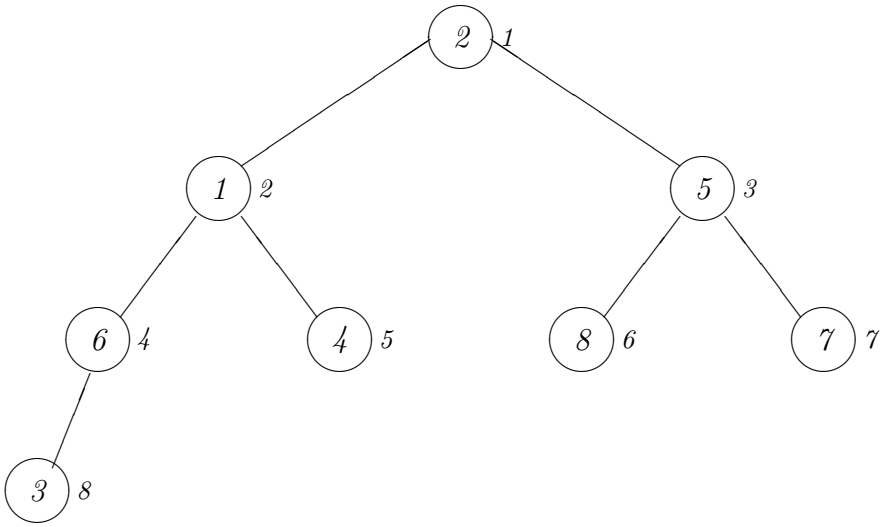
$$k_{\lfloor \frac{N}{2} \rfloor}, \ldots, k_1$$

in dieser Reihenfolge in F versickern.

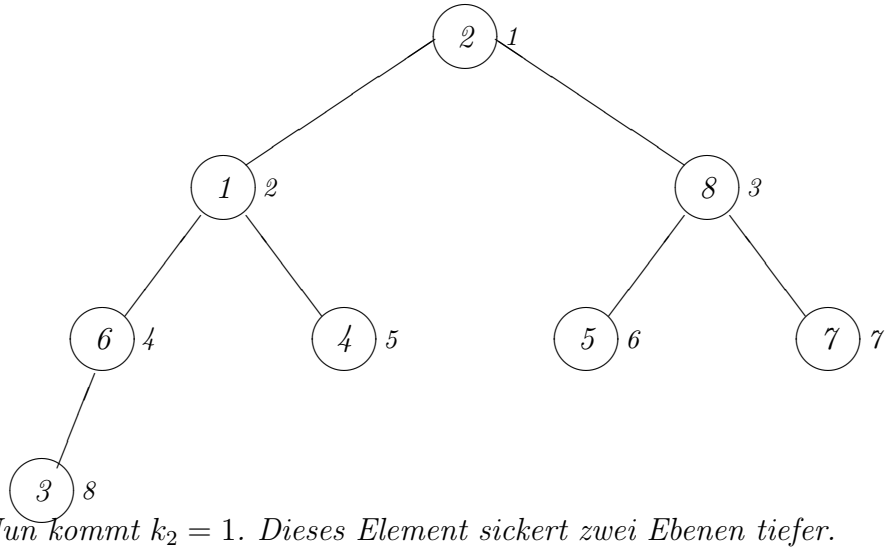
Beispiel 4.7 *Gegeben sei folgende unsortierte Folge in Binärbaumdarstellung:*



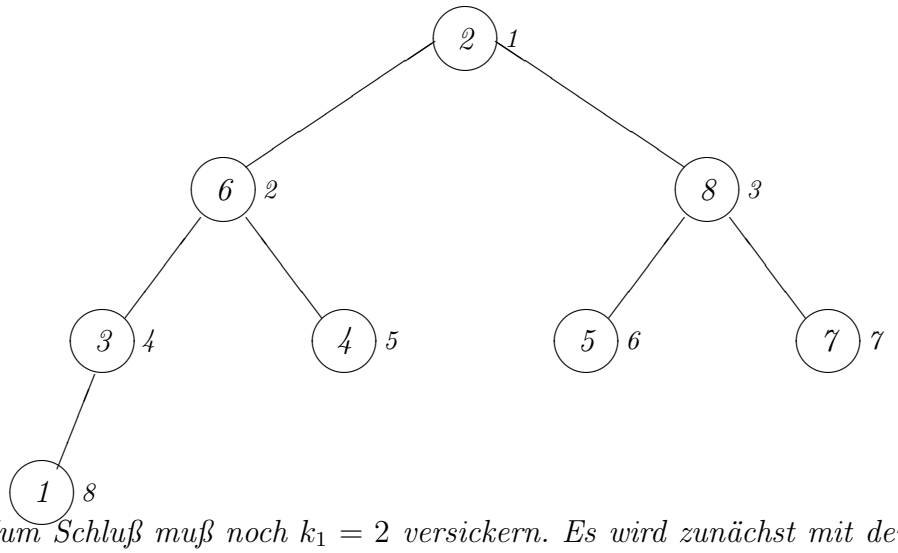
Wir beginnen mit Element $k_{\lfloor \frac{N}{2} \rfloor} = k_4 = 3$*, da durch die Speicherung des binären Baumes in einem Array klar ist, wo die Blätter sind; diese müssen bei diesem Verfahren nicht betrachtet werden! Es versickert nur eine Position weiter runter:*



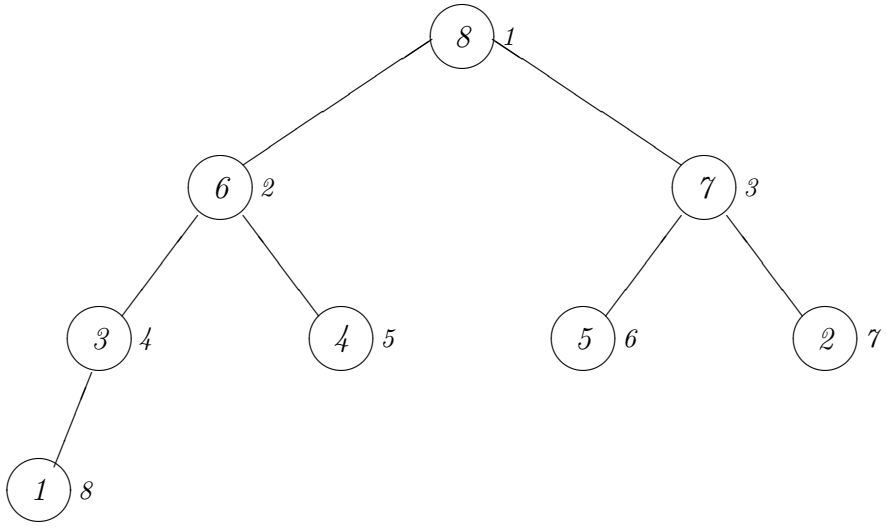
Dann kommt Element $k_3 = 5$*. Es wird mit dem größeren seiner Söhne, also mit* $k_6 = 8$ *getauscht.*



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.

Hier der Algorithmus:

```
reHeap_up( ) {
// Heapeigenschaft herstellen
   for (int i = HeapEnde/2;
        i > 0;
        i--)
     HeapSeep(i,HeapEnde);
}
```

Zum vollständigen Sortieren benötigt man vollgenden Algorithmus:

```
reHeap_down( ) {
// komplette Sortierung!
   for (int i = HeapEnde;
        i > 2;
        i--) {
   swap(1,i);
   HeapSeep(1,i-1); }
}
```

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 der im folgenden Abschnitt vorgestellte Algorithmus 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.

4.4 Quicksort

Quicksort wurde 1962 von C.A.R. Hoare veröffentlicht. 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. Es ist damit rekursiv und gehört zur Teile-und-Herrsche Strategie (*Divide-and-Conquer* bzw. *divide et impera*).

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 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.

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 4.6 *Ein* In-situ[1]*-Sortierverfahren, also ein Sortierverfahren, welches die Datensätze an Ort und Stelle sortiert, benötigt für die Zwischenspeicherung von Datensätzen keinen zusätzlichen Speicher, außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen.*

[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“.

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:	♣10	♠A	♣K	♡8	♢D	♡D	♡A	♣4

Wir wählen ♣4 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:	♡A	♠A	♣K	♡8	♢D	♡D	♣10	♣4

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:	♡A	♠A	♡D	♡8	♢D	♣K	♣10	♣4

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:	♡A	♠A	♡D	♡8	♢D	♣4	♣10	♣K

Die Teilfolgen sind also

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

Im folgenden das Programm dafür.

```
quicksort(int ilinks, int irechts) {
      int pivot,i,j;
      if ( irechts > ilinks ) {
      //quickSwap
           i = ilinks;
```

```
            j = irechts-1;
            pivot = a[irechts].key;
            while(i ≤ j) {
                 while((a[i].key ≤ pivot) && (i < irechts)) i++;
                 //a[i].key > pivot
                 while((a[j].key > pivot) && (ilinks ≤ j)) j-;
                 //a[j].key ≤ pivot
                 if ( i < j ) {
                      swap(i,j);//vertauschen
                 }
            }
            swap(i,irechts);//Pivotelement in die Mitte tauschen
       //quickSwap
            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

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

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 garnichts zu tun.

Der Binärbaum hat dann maximale Höhe, also n. Auf jeder Ebene werden $c \cdot n$ Operationen ausgeführt (nur Vergleiche! Bewegungen werden garnicht 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.

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

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

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

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

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

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

Induktionsschritt Dann folgt

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

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:

$$\frac{2c}{n}\left[\sum_{k=1}^{\frac{n}{2}} k \cdot \log k + \sum_{k=1}^{\frac{n}{2}-1} \left(\frac{n}{2} + k\right) \log\left(\frac{n}{2} + k\right)\right] + bn$$

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

Wir wenden nun die Summenformel

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

an für die beiden Summen und erhalten

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

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.

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 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 als Pivotelement. Das auf diese Weise randomisierte (zufällig gemachte) Quicksort behandelt alle Eingabefolgen fast gleich. Randomisiertes Quicksort ist ein Beispiel für das Paradigma *Zufallsstichproben*. Die Laufzeit des Algorithmus ist eine Zufallsvariable. Randomisiertes Quicksort ist also 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.

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.

4.5 Mergesort

Mergesort ist ein Sortierverfahren, welches sich auch die *Teile-und-Herrsche*-Strategie 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 Liste genommen und der Rest für die zweite Liste. 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.

4.5.1 2-Wege-Mergesort

Methode Eine Folge

$$F = a_1, \ldots, a_N$$

wird in zwei möglichst gleich große Teilfolgen

$$F_1 = a_1, \ldots, a_{\left[\frac{N}{2}\right]} \quad F_2 = a_{\left[\frac{N}{2}+1\right]}, \ldots, a_N$$

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 incrementiert, 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:	♣4	♠A	♣K	♡8	♢D	♡D	♡A	♣10

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

i:	1	2	3	4	5	6	7	8
a_i:	♣4	♠A	♣K	♡8	♢D	♡D	♡A	♣10

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:	♡8	♠A	♣4	♣K	♢D	♡D	♡A	♣10

Danach werden die Teilfolgen zusammengeführt.

i:	1	2	3	4	5	6	7	8						
a_i:	♡8	♠A	♣4	♣K	♢D	♡D	♡A	♣10						
	↑				↑				♢D					
	↑					↑			♢D	♡8				
		↑				↑			♢D	♡8	♡D			
		↑					↑		♢D	♡8	♡D	♡A		
		↑						↑	♢D	♡8	♡D	♡A	♠A	
			↑					↑	♢D	♡8	♡D	♡A	♠A	♣4
				↑				↑	♢D	♡8	♡D	♡A	♠A	♣4	♣K

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.

4.5.2 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 (initiales Sortieren) 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. Die Datensätze beim initalen Sortieren werden auch Run genannt; deren Größe sollte die maximal mögliche Größe betragen. Auf t_3 und t_4 sind nun sortierte Folgen von Datensätzen (die 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 (am Anfang von zwei Runs) entsteht eine sortierte Teilfolge in der Länge der Summe der beiden Ausgangsfolgen (also am Anfang Länge von zwei Runs). 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. Man bezeichnet die dann eingelesenen Teilfolgen als Blöcke. Minimale Größe eines Blocks ist die Größe eines Schlüssels. Beim Erzeugen der sortierten Teilfolgen am Anfang möchte man natürlich so lange Teilfolgen wie möglich erzeugen. Hier wird möglichst viel Hauptspeicher verlangt. Als initialer Sortieralgorithmus kann jeder Sortieralgorithmus verwendet werden! Mergesort ist da nicht unbedingt der geeigneteste.

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!

Weitere Entwicklungen unterscheiden sich in der Organsisation der externen Speicher (auch Bänder genannt), da hier das wesentliche Potential zur Optimierung besteht. So verwendet Mehr-Wege-Mergesort mehr als vier Bänder, um gleichzeitig mehr als zwei Zahlen vergleichen zu können, also um z.B. nach der initialen Sortierung mehr als zwei Runs gleichzeitig zusammen zu führen. Dies kann ab einer bestimmten Anzahl mittels einem (Min/Max-)Heap effizient durchgeführt werden. Das Mehrphasen-Mergesort versucht nun das Manko der vielen passiven Bänder zu beheben: bei z.B. sechs Bändern würden stets zwei Bänder passiv sein! Hier soll stets ein Band nur als Ausgabeband zur Verfügung stehen und alle anderen Bänder als Eingabeband genutzt werden. Damit dies gelingt, müssen jedoch die Runs beim initialen Sortieren gemäß den Fibonacci-Zahlen k-ter Ordnung auf die k Eingabebänder verteilt werden. Das jeweils dann frei werdende Band dient dann als neues Ausgabeband. Damit wird vermieden, dass es zu einer Situation kommt, in der nur noch auf einem Band Zahlen vorhanden sind, diese jedoch noch in zwei (oder mehr) sortierten Teilfolgen stehen.

4.6 Rekursion bei Quick- und Mergesort

Rekursion unter einem anderen Blickwinkel[2]: 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

[2]Dies ist `Sedgewick` entnommen worden.

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 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.

4.7 Fazit aus Sicht der Konstruktionslehre

Die wesentlichen Konstruktionsprinzipien hier sind

naiv : Der Algorithmus wird ohne weitere Betrachtung des Problems „naiv“ umgesetzt. Im Falle von Selection Sort bietet das Verfahren keine Möglichkeit zu Verbesserungen[3]. Im Falle von Insertion Sort kann jedoch durch etwas Nachdenken über das Verfahren eine 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 zunächst durch dieses Prinzip „geteilt wird“. Möchte bzw. muss man noch mehr „rausholen“, so ist die Eigenart des

[3]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!.

Problems selbst zu betrachten. Im Praktikum hat dies zu der schnellsten Lösung geführt.

4.8 Aufgaben

1. **Einfaches Verfahren** (? Punkte)
   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.

2. **Shell Sort** (? Punkte)
   Sotieren Sie die folgende Zahlenreihe mit dem Shell Sort 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 an (Ausführung von Zeile `11`).

   ```
   01   void ShellSort(int N) {
   02      int h = 1, i = 0;
   03      /* Anfangswert für die Distanz nach Knuth */
   04      for ( h = 1; h <= (N/9); h = 3*h+1);
   05      for( ; h>0; h /= 3) {
   06         for(i = h; i <= N ; i=i+h) {
   07            int j = i;
   08            int t = a[i];
   09            int k = t.key;
   10            while(((j-h)>= 0) && (a[j-h] > t)) {
   11               a[j] = a[j-h];
   ```

```
12                j = j-h; }
13             a[j] = t; }
14       } }
```

3. **Insertion Sort** (? Punkte)
   Gegeben sei das Array $A = [\perp, 55, 41, 52, 26, 57, 9, 49, 60]$, sowie der aus der Vorlesung bekannte Sortieralgorithmus

```
InsertionSort(Array) {
      for ( int i = 2; i ≤ Array.length ; i++ ) {
           int j = i;
           Datensatz t = a[i];
           int k = t.key;
           while(a[j-1].key > k) && (j > 1) {
                a[j] = a[j-1];
                j = j-1; }
           a[j] = t; }}
```

   (a) Welche Werte hat das Array A, wenn `Insertion-Sort(A)` aufgerufen wird, die `for`-Schleife jedoch nur zweimal mit dem ersten Wert $i = 2$ und anschließend mit dem zweiten Wert $i = 3$ durchlaufen wird ? Die Dokumentation muß den Ablauf nachvollziehbar darstellen.

   (b) Welche Zeitkomplexität im Sinne der O-Notation hat Insertion-Sort, wenn die Eingabe ein Array der Länge n ist, welches bereits aufsteigend vorsortiert ist bzw. bereits absteigend vorsortiert ist. Begründen Sie Ihre Antwort.

4. **CornSort** (? Punkte)
   Gegeben sei folgender Algorithmus:

```
algorithm CornSort(A : array of sortable items)
  swapped := true;
  while swapped == true do
    swapped := false;
    for i := 0 to length(A) - 2 do
      if A[i] > A[i+1] then swap(A[i],A[i+1]);
                            swapped := true;
      endif endfor
    if swapped == false then exit; endif
    swapped := false;
    for i := length(A) - 2 downto 0 do
      if A[i] > A[i+1] then swap(A[i],A[i+1]);
```

```
                                swapped := true;
      endif endfor
  endwhile
end CornSort;
```

Die Funktion `length(A)` liefert die Größe des Arrays `A`. Arrays werden von 0 bis length(A)-1 indiziert. Die Methode `swap(X,Y)` vertauscht die Inhalte der Variablen `X` und `Y`. Das Schlüsselwort `exit` bewirkt, dass die Bearbeitung des Algorithmus unmittelbar beendet wird.

(a) Wie funktioniert `CornSort` ? Beschreiben Sie die Funktionsweise!

(b) Geben Sie für `CornSort` in Abhängigkeit von `n = length(A)` eine Laufzeitkomplexität an. Es reicht, die Anzahl der Vergleiche zu zählen. Begründen Sie kurz Ihre Entscheidung. Ist `CornSort` ein optimales Sortierverfahren (basierend auf Schlüsselvergleichen) ?

(c) Betrachten und beschreiben Sie die Behandlung des kleinsten und größten Elements der Eingabe unter `CornSort`.

(d) Geben Sie Vorschläge, wie Sie die Laufzeit von `CornSort` verbessern können.

(e) Hat sich durch Ihre Optimierung die Laufzeitkomplexität von `CornSort` verbessern können ? Begründen Sie kurz Ihre Entscheidung.

5. **Sortieralgorithmen** (? Punkte)
Gegeben sei ein anfangs aufsteigend sortiertes Array von n ganzen Zahlen $(a_1, \ldots, a_n)$ mit $a_i \in \mathbb{Z}$. In diesem Array werden nun einige Elemente verringert, indem positive Zahlen abgezogen werden. Wie viele und welche Elemente verringert werden ist nicht bekannt. Ebenso ist nicht bekannt, um welchen Betrag ein Element verringert wird.

   Das Array soll nun in der Laufzeitkomplexität $O(n * k)$ wieder sortiert werden, wobei k die Anzahl der verringerten Elemente ist. Beachten Sie: Wird nur ein einziges Element verringert ($k = 1$), so ist die geforderte Laufzeitkomplexität $O(n)$.

   (a) Geben Sie den Pseudocode eines Sortieralgorithmus an, der die Anforderungen erfüllt.

   (b) Analysieren Sie die Laufzeit des Algorithmus.

6. **Teile-und-Herrsche** (? Punkte)
Erklären Sie kurz das Teile-und-Herrsche-Prinzip. Beschreiben Sie dazu grob den Ablauf.

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.

7. **Teile und Herrsche** (? Punkte)
   Manchmal ist es notwendig, den minimalen oder den maximalen Wert einer Folge zu finden. Eine Verallgemeinerung hiervon ist es, das k-kleinste Element einer Folge zu finden. Eine Möglichkeit, dies zu erreichen ist es, die Folge zunächst zu sortieren und das k-te Element auszuwählen. Der dazu notwendige Aufwand beträgt $O(nlog(n))$ wobei n die Folgenlänge darstellt. Es ist jedoch nicht notwendig, die gesamte Folge zu sortieren, um das gesuchte Element zu finden!

   (a) Entwickeln Sie einen „Teile und Herrsche"-Algorithmus, der das k-kleinste Element einer gegebenen Folge (mit $n > k$) berechnet. Ihr Algorithmus sollte im durchschnittlichen Fall mit $O(n)$ Zeit auskommen, also im durchschnittlichen Fall linear sein. Sie können davon ausgehen, dass alle Werte in der Eingabefolge paarweise verschieden sind.

   (b) Begründen Sie, warum Ihr Algorithmus korrekt arbeitet.

   (c) Nehmen Sie einen Ansatz vor, den Aufwand (Charakter) Ihres Algorithmus zu bestimmen.

8. **Rekursion** (? Punkte)
   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?

9. **Quicksort** (? Punkte)
   Sotieren Sie die folgende Zahlenreihe mit dem im Tipp aufgeführten Quicksort-Algorithmus. Notieren Sie die Zwischenergebnisse nach jeder Tauschoperation.

43	19	81	2	39	41	3	99	12

   Geben Sie die Anzahl der Rekursionsaufrufe und die Rekursionstiefe an

10. **Mehr-Wege-Mergesort** (? Punkte)
    Sortieren Sie die unten angegebene Folge mittels Mehr-Wege-Mergesort für $k = 3$ und einer initialen Run-Länge von 1:

$$T_0 : 43, 71, 12, 35, 11, 80, 99, 47, 62, 15, 77, 26, 69, 93, 19, 45, 23, 38, 56, 91, 10$$

11. **Heapsort** (? Punkte)
    Gegeben sei ein Maximum-Heap, wie in der Vorlesung vorgestellt. Beantworten Sie folgende Fragen und begründen Sie Ihre Antworten:

    (a) Was versteht man hierbei unter dem Begriff „Heap-Eigenschaft“?

    (b) Wie wird die Baumrepräsentation eines Heaps in ein Array kodiert, also wo findet man die Nachfolger eines Knotens im Array?

    (c) Was ist die minimale und maximale Anzahl von Elementen in einem Heap der Höhe h?

    (d) Wo kann sich in einem Heap das kleinste Element nur befinden?

    (e) Welche Elemente in einer unsortierenden Zahlenfolge bilden zu Anfang bereits einen Heap, und warum wird diese Struktur von hinten (rechts) nach vorne (links) in der Array-Repräsentation eines Heaps bzw. von unten nach oben in der Baumrepräsentation eines Heaps als Heap aufgebaut? (Bottom-Up Verfahren)

12. **Heapsort** (?? Punkte)

    (a) Ist das Array (beginnend bei Position 1) mit den Werten $[20, 17, 11, 6, 13, 10, 1, 5, 6, 12]$ ein Max-Heap ? Begründen Sie ihre Antwort.

    (b) Ist ein Array, das in aufsteigend sortierter Reihenfolge vorliegt, ein Min-Heap ? Begründen Sie ihre Antwort.

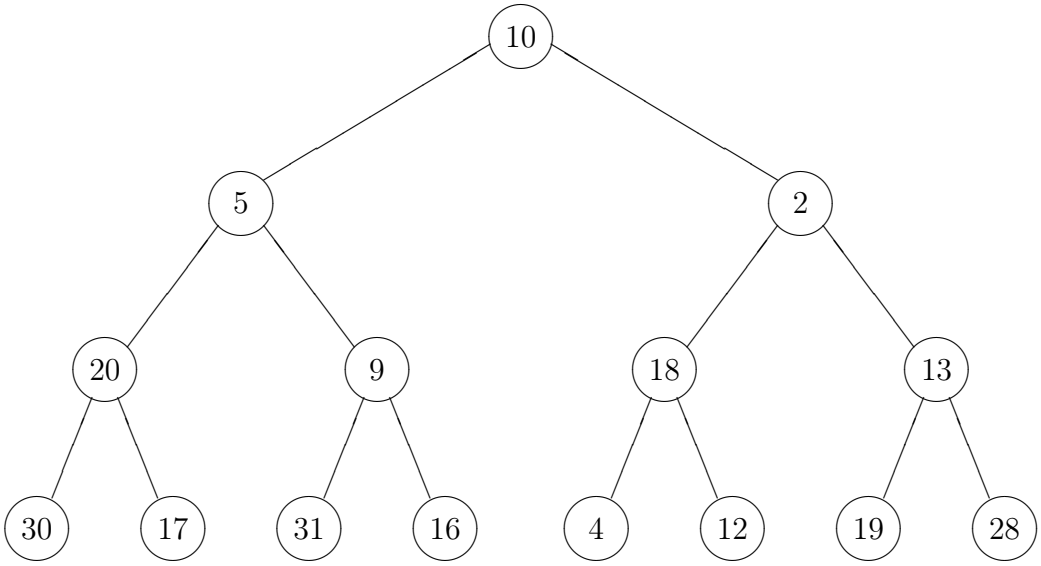
13. **Heapsort** (? Punkte)
    Gegeben sei folgender Maximum-Heap:
    `( 45,31,41,23,15,12,32,10,8,2,13 )`.

    (a) Zeichnen Sie den Heap als Baum.

    (b) 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.

    (c) 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.

14. **Heapsort** (?? Punkte)
    Gegeben sei nachfolgender binärer Baum:



(a) Betten Sie den abgebildeten Baum gemäß der Kodierung beim Heap in ein Array.

(b) Erstellen Sie nun durch Aufruf der Funktion `reHeap_up` aus dem Baum einen Max-Heap. Dokumentieren Sie den Ablauf nachvollziehbar: je Iterationsschritt in `reHeap_up` ist das resultierende Array darzustellen.

(c) Löschen Sie viermal das größte Element und stellen sie jeweils die Heap-Eigenschaft wieder her. Geben Sie die resultierenden Bäume nach jedem Schritt als Array an.

(d) Wie können Sie die Datenstruktur Heap zum Sortieren verwenden ? Geben Sie dazu in Pseudo-Code einen Algorithmus an. Welche Zeitkomplexität im Sinne der O-Notation hat Ihr Algorithmus ?

15. **Heap Increase** (? Punkte)
Schreiben Sie eine Funktion mit der Signatur `heap-increase-key(A,i,k)`. Die Funktion soll in einen bestehenden Heap den Schlüssel an der Stelle i des Arrays A durch den neuen größeren Schlüssel k ersetzen. Die Laufzeitkomplexität soll $O(log(n))$ betragen. Begründen Sie, warum Ihr Algorithmus die geforderte Laufzeitkomplexität einhält und korrekt arbeitet.

Kapitel 5

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 Repräsentation 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 5.1 *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:

   (a) *Die Daten sortieren:*
   Wie bei den Sortieralgorithmen und insbesondere den Prioritätswarteschlangen (z.B. Heap) arbeiten wir mit Daten, die in Datensätze

[1] Stichwort: Abstrakter Datentyp, semantische Netze, Constraint-Netze etc.

oder Elernente 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.

(b) *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).
Um das Suchen nach einer (optimalen) Lösung, zu modellieren, wollen wir uns folgender abstrakter Grundvorstellungen bedienen:

i. *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.*
ii. *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 schaun wir uns in diesem Kapitel genauer an.

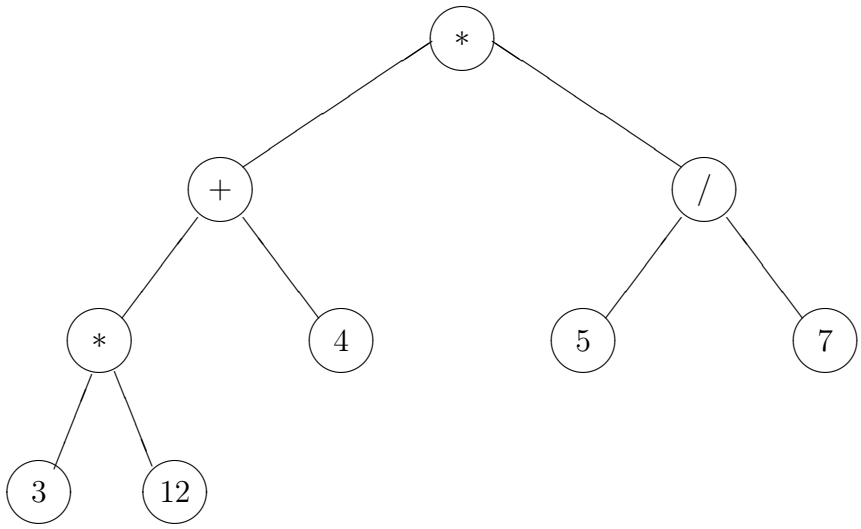
Mit Heapsort haben wir ein Verfahren kennen gelernt, das einmal genutzt werden kann, um Daten lazy *zu sortieren (Bottom-Up Heapsort) und einmal, um Daten sortiert zu halten (Top-Down Heapsort). In beiden Fällen hat Heapsort sehr günstige Laufzeitkonditionen.*

5.1 Definition von Bäumen

Wir haben im vorherigen Kapitel über Sortierverfahren schon gesehen, dass Datentypen 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 als Baum dargestellt werden:



Definition 5.1 *Ein Baum ist eine Menge von Objekten welche man* **Knoten** *nennt, zusammen mit einer Relation auf der Knotenmenge (graphisch durch* **Kanten** *dargestellt), so dass sich eine hierarchische Struktur ergibt. Die Knoten, die unterhalb eines Knotens p liegen und mit p durch Kanten verbunden sind, heißen* **Söhne** *(Nachfolger) von p. p heißt* **Vater** *dieser Knoten.*

Jeder Baum hat eine **Wurzel**. *Das ist der Knoten, der keinen Vater hat. Er dient als Einstieg in den Baum.*

Ein **Pfad** *ist eine Folge von Knoten $p_0, p_1, \ldots, p_n$, so dass jeweils p_i der Vater von p_{i+1} ist. Die Länge dieses Pfades ist n, also die Anzahl der Kanten im Pfad.*

Die Länge des längsten Pfades nennt man die **Höhe** *eines Baumes.*

- **Knoten**
  - **Kante** Verbindungslinie zwischen Knoten

  - **Sohn** unmittelbar nachfolgender und durch Kante verbundener Knoten (nach unten)
  - **Vater** darüberliegender Knoten (durch Kante verbunden)
  - **Wurzel** höchster Knoten im Baum: hat keinen Vater
  - **Grad** Anzahl der Söhne
  - **Tiefe** Abstand vom Wurzelknoten

- **Baum**
  - **Pfad** Folge von Knoten von oben nach unten
  - **Länge eines Pfades** Anzahl der Kanten
  - **Höhe** Länge des längsten Pfads
  - **Grad** Maximum der Grade aller Knoten

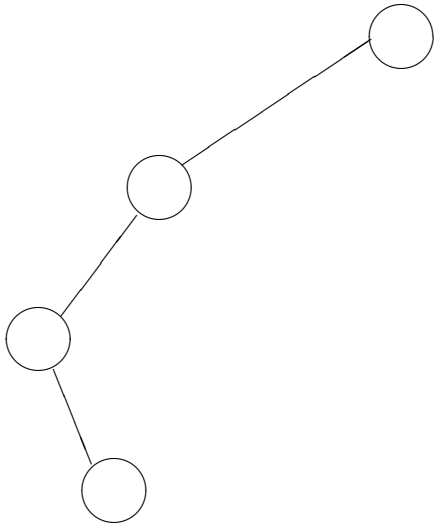
Definition 5.2 *Ein Baum heißt* binär*, wenn die Wurzel und alle anderen Knoten 2 Nachfolger oder weniger Nachfolger hat. Die Knoten mit 0 Nachfolgern sind die Blätter des binären Baums. Ein binärer Baum heißt* voll*, wenn zwischen jedem Blatt und der Wurzel dieselbe Anzahl von Kanten liegt und mit Ausnahme 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. Im folgenden betrachten wir nur binäre Bäume.

Wir wollen nun Eigenschaften von binären Bäumen untersuchen. Zunächst interessiert uns die Höhe eines Baumes mit einer bestimmten Anzahl von Knoten. Dazu gibt es folgende Aussagen:

Satz 5.1 *(maximale und minimale Höhe eines Baums)*

1. *Die maximale Höhe eines Baumes mit N Knoten ist $N - 1$. Dieser Fall tritt ein, wenn der Baum zu einer Liste entartet.*

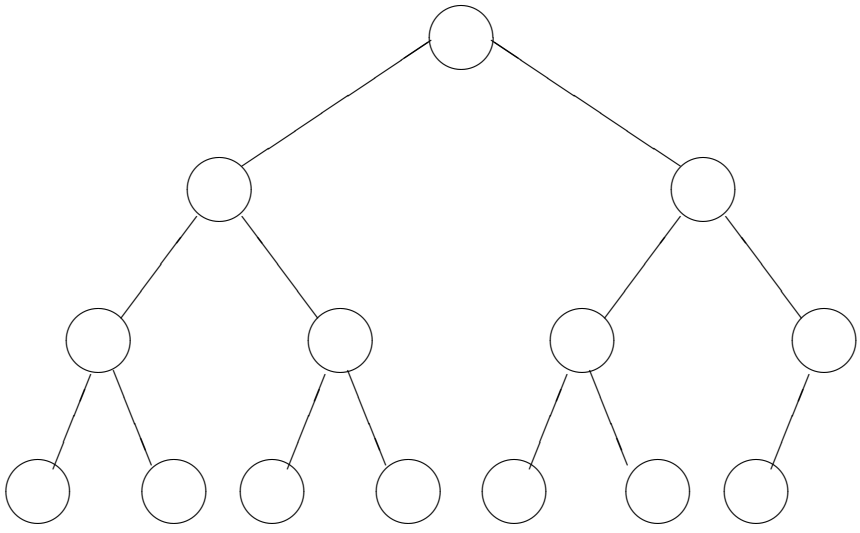


2. *Die minimale Höhe eines Baumes vom Grad* 2 *mit* N *Knoten ist* $\lfloor \log_2 n \rfloor$. *Dieser Fall tritt ein, wenn bis auf einige der untersten Knoten alle Knoten besetzt sind. Ist ein binärer Baum mit Höhe* h *nämlich vollbesetzt, so hat er*

$$1 + 2 + 4 + 8 + 16 + 32 + \ldots + 2^h = 2^{h+1} - 1 = n$$

Knoten. In diesem Fall gilt:

$$h = \lfloor \log_2(2^{h+1} - 1) \rfloor$$



Auf Bäumen kann man verschiedene Reihenfolgen von Knoten definieren. Drei Reihenfolgenen haben einen bestimmten Namen: inorder (symmetrische Reihenfolge), postorder (Nebenreihenfolge) und preorder-Reihenfolge (Hauptreihenfolge). Sei dazu T_k ein Teilbaum, der als Wurzel einen 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

$$\begin{aligned} \text{inorder} &: \text{Teilbaum} \longrightarrow \text{Folge von Knoten} \\ \text{preorder} &: \text{Teilbaum} \longrightarrow \text{Folge von Knoten} \\ \text{postorder} &: \text{Teilbaum} \longrightarrow \text{Folge von Knoten} \end{aligned}$$

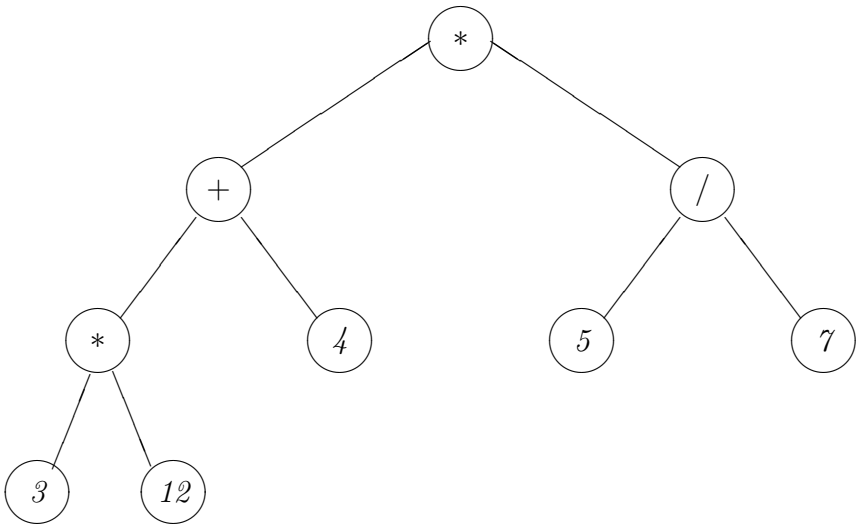
Wir definieren die Reihenfolgen dann rekursiv.

- $\text{inorder}(T_k) = (\ \text{inorder}(T_{k1})\ ,\ k\ ,\ \text{inorder}(T_{k2})\ )$
- $\text{preorder}(T_k) = (\ k\ ,\ \text{preorder}(T_{k1})\ ,\ \text{preorder}(T_{k2})\ )$
- $\text{postorder}(T_k) = (\ \text{postorder}(T_{k1})\ ,\ \text{postorder}(T_{k2})\ ,\ k\ )$

Folgende Bezeichnung verwendet man außerdem für die Reihenfolge:

- inorder = symmetrisch
- preorder = Hauptreihenfolge
- postorder = Nebenreihenfolge

Aufgabe 5.1 *Geben Sie die Inorder-, Preorder- und Postorder-Reihenfolgen von folgendem Baum an:*



5.2 Implementierungen

5.2.1 Dynamische Implementierung

Man kann Bäume implementieren wie verkettete Listen. Die Knoten sind Objekte, welche mit Referenzen (Zeigern) auf ihre Söhne zeigen. Die Klasse Knoten hat also zwei Zeiger auf ihre Söhne. Möchte man die Bäume bequem von unten nach oben durchgehen, so benötigt man noch einen Vater-Zeiger.

```
class Knoten
{
      DAT daten;
      Knoten links,rechts,vater;
}
```

Dabei enthalten `daten` die Daten des Knotens, also zum Beispiel eine Operation oder einen Wert.

Bemerkung 5.2 `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.*

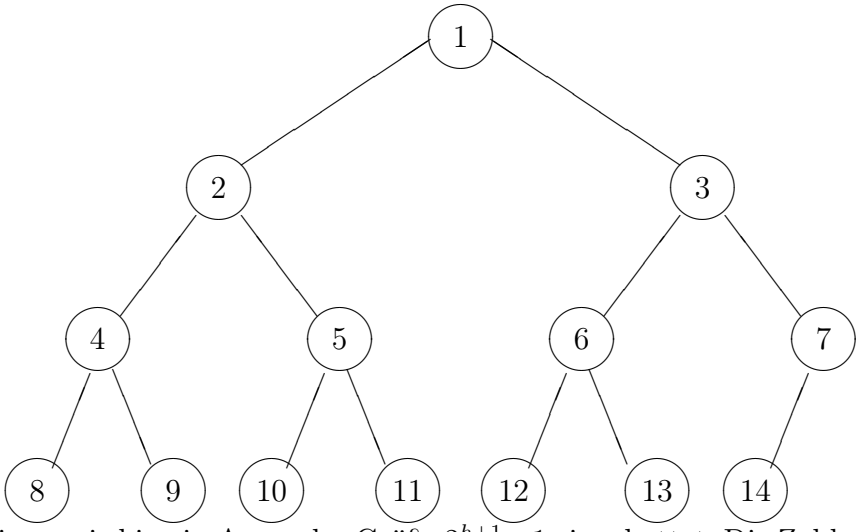
Man benötigt einen weiteren Datentyp, der die Wurzel des Baumes kennzeichnet. Dies ist nötig, da in den Algorithmen die Wurzeln entfernt werden (Initialisiesierungsproblem bzw. Problem des Anfangs). Dabei erhält man zwei neue Bäume je wieder mit einer Wurzel.

```
class Baum
{
      Knoten wurzel;
}
```

5.2.2 Statische Implementierung

Natürlich kann man die oben definierten Knoten einfach als Array allokieren und danach entsprechend verlinken. Mit der statischen Implementierung ist jedoch eine Implementation gemeint, die völlig ohne Zeiger auskommt. Die Struktur entsteht durch Indexberechnung im Array, wie wir es beim Heap schon kennen gelernt haben.

Dazu betrachten wir einen weitgehend vollständigen binären Baum der Höhe h.



Dieser wird in ein Array der Größe $2^{h+1}-1$ eingebettet. Die Zahlen in den Knoten in obiger Abbildung geben den Index im Array an. 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 liegt. Zum Vater von i kommt man durch

$$\text{Vater}(i) = \left\lfloor \frac{i}{2} \right\rfloor,$$

falls das Ergebnis nicht Null ist, dann war i schon die Wurzel des Baumes.

Falls die darzustellenden Bäume vollständig sind, ist die Realisierung mittels Array eine gute Implementierung. Sind die Bäume dagegen nur schwach besetzt, so wird viel Speicherplatz verschwendet.

Aufgabe Stellen Sie einen vorgegebenen Baum als Array dar. Stellen Sie ein vorgegebenen Array als Baum dar.

5.2.3 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, welche der entsprechenden Durchlaufordnung entspricht (pre-, in- oder postorder). Da die Durchlaufordnungen rekursiv definiert sind, können sie auch entsprechend implementiert werden. Hier ein Pseudocodeprogramm für die Herstellung einer Postorderreihenfolge.

   Algorithmus 5.1 Postorder(Teilbaum B)

   - *Falls B keine Söhne hat, so gib B zurück*

- *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 (*post-Ausgabe*)*

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 und/oder durch Verwendung zusätzlich Zeiger realisiert werden bzw. im Falle der Verwendung eines Arrays durch entsprechende Berechnungsvorschriften. Mit einem Array wäre z.B. die Preorder einfach auszugeben, weil man eine triviale `for`-Schleife verwenden könnte.

5.3 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 $\mathbb{N}$

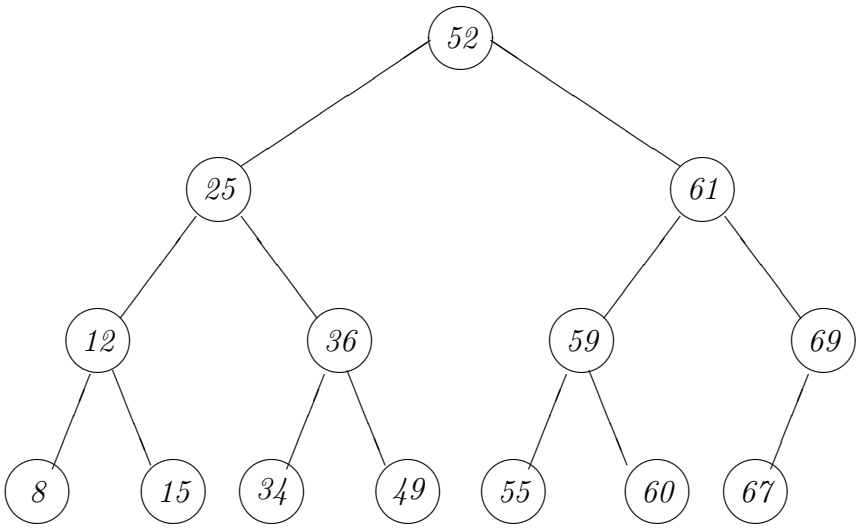
$$\mu \ : \ T \longrightarrow \mathbb{N}.$$

Definition 5.3 *Ein binärer Baum T mit einer Knotenmarkierung μ heißt* **binärer Suchbaum** *genau dann, wenn für jeden Teilbaum T' von T mit $T' = (T'_l, k, T'_r)$ gilt:*

$$\begin{aligned} \forall \text{ Schlüssel } x \in T'_l \ &: \ \mu(x) < \mu(k) \\ \forall \text{ Schlüssel } z \in T'_r \ &: \ \mu(z) > \mu(k) \end{aligned}$$

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 5.1



Bemerkung 5.3 *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
{
      int key;
      Knoten* links,rechts;
}
```

Dann kann eine Memberfunktion definiert werden, die feststellt, ob ein Knoten mit einem bestimmten Key vorkommt.

```
bool exist(int k)
{
      if(k == key) return TRUE;
      else
      {
           if(k < key)
                if(links == 0) return FALSE;
                else return links->exist(x);
           else
                if(rechts == 0) return FALSE;
                else return rechts->exist(x);
```

```
      }
}
```

Diese Routine wird für den Wurzelknoten des Baumes aufgerufen und liefert TRUE zurück, wenn der gefragte Knoten existiert. Wir erweitern nun diese Routine um zu einer Einfügeroutine zu gelangen. Dabei wird der neue Knoten eingehängt, wenn wir für `links` oder `rechts` auf einen NULL-Zeiger stoßen.

```
Knoten* insert(int k)
{
      if(k == key) return this;
      else
      {
           if(k < key)
           {
                if(links == 0)
                {
                     links = new Knoten(k);
                     return links;
                }
                else return links->insert(x);
           }
           else
           {
                if(rechts == 0)
                {
                     rechts = new Knoten(k);
                     return rechts;
                }
                else return rechts->insert(x);
           }
      }
}
```

Diese Routine liefert einen Pointer 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 aufwändiger. Beim Löschen eines inneren Knotens muss die Suchstruktur des Baumes erhalten bleiben.

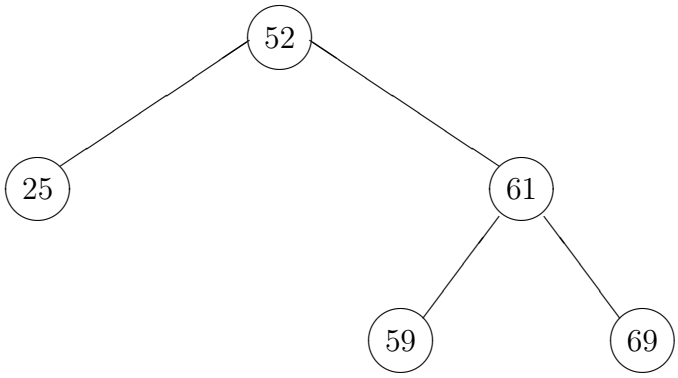
5.4 AVL-Bäume

Der erste Vorschlag zu höhenbalancierten Bäumen wurde 1962 von Adelson-Velskii und Landis gemacht. Man nennt sie AVL-Bäume und sie sind folgendermaßen definiert:

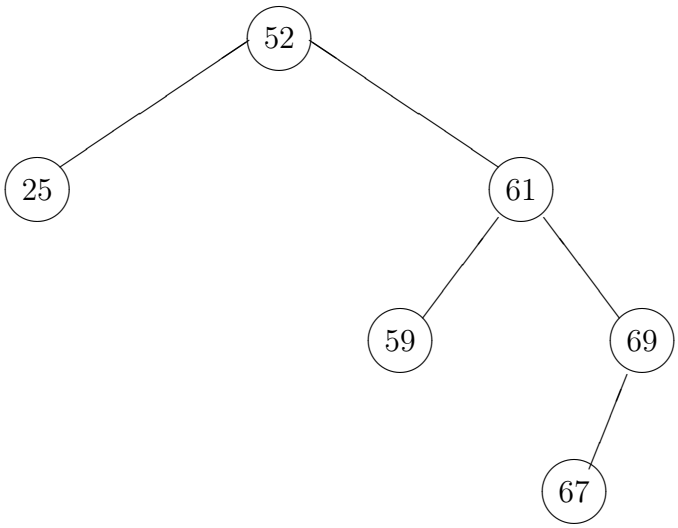
Definition 5.4 *Ein binärer Baum heißt AVL-Baum, wenn für jeden Knoten p gilt, dass sich die Höhen des linken ($h(Tl)$)und rechten ($h(Tr)$) Teilbaums höchstens um 1 unterscheiden. Wir definieren den Balance-Faktor durch: $bal(p) = h(Tr) - h(Tl) \in \{-1, 0, 1\}$. Falls $bal(p) \notin \{-1, 0, 1\}$, also $bal(p) \in \{-2, 2\}$ gilt, liegt ein Fehler vor.*

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.

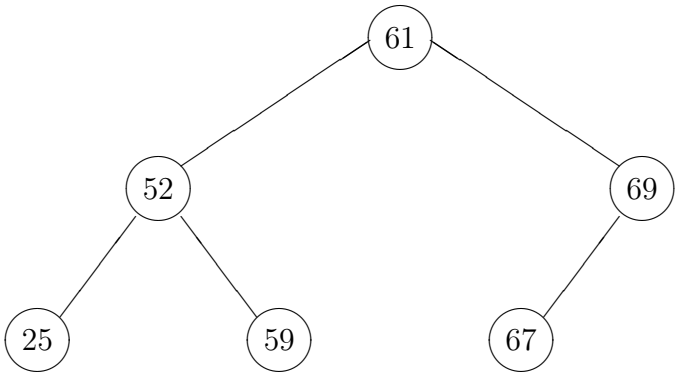
Bei jedem Einfügen und Löschen eines Knotens in einen AVL-Baum wird durch eine **Rebalancieroperation** wieder die AVL-Bedingung hergestellt. Hier müssen verschieden Fälle unterschieden werden. Ein einfacher Fall wird hier zunächst vorgeführt. Dazu betrachten wir einen Teilbaum, der durch Einfügen eines Knotens die AVL-Bedingung verletzt. Hier der ursprüngliche Baum:



Nun soll ein Knoten mit Wert 67 eingetragen werden. Das ergibt:



Der rechte Teilbaum hat nun Höhe 3, während der linke Teilbaum (der Knoten mit Wert 25) Höhe 1 hat. Wir rotieren den Baum nun gegen den Uhrzeigersinn. Der Knoten mit dem Wert 61 wird dabei die Wurzel. Knoten 59 wird nun der frei werdende rechten Sohn von 52.



Er ist sinnvoll, wenn nach einer aufwendigen Aufbauphase sehr viele Zugriffe auf die eingefügten Daten vorgenommen werden. Die Zugriffszeit ist `O(log(n))`, also die Höhe des binären balancierten Baumes; die Zeit zum balancieren beträgt ebenfalls `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!

Für diesen Algorithmus werden Rotationen benötigt, die in Abbildung 5.4 (Seite 129) 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.

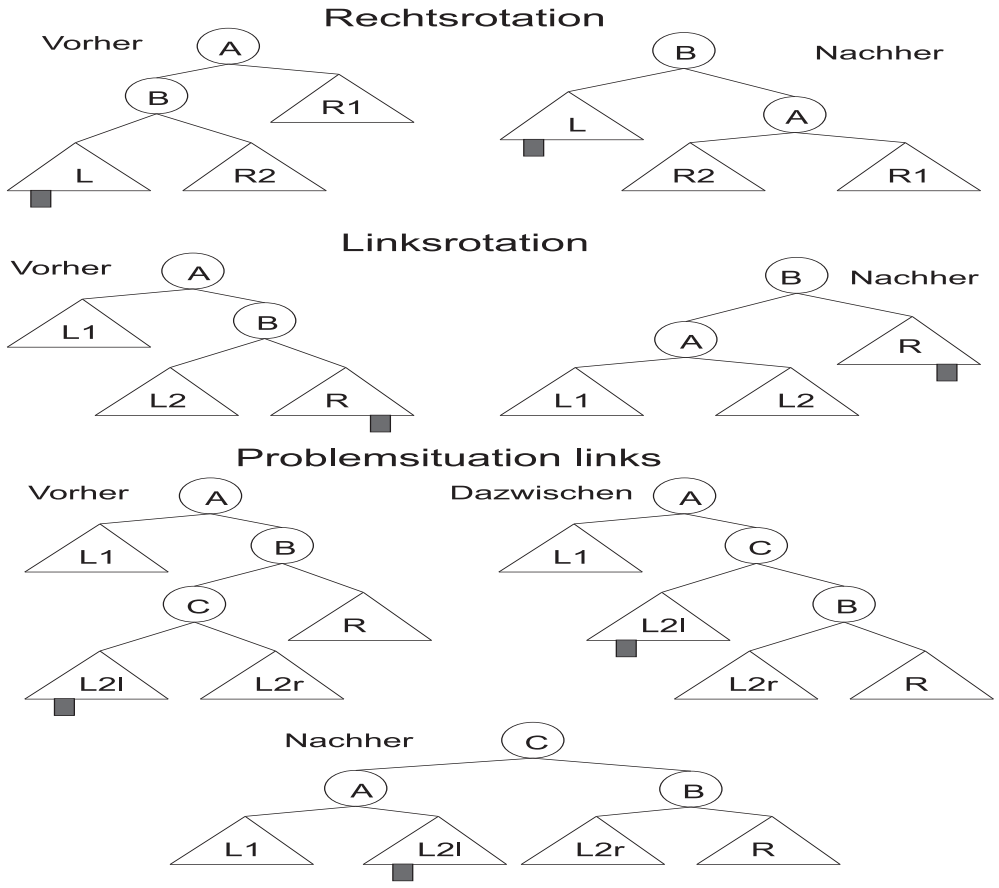
4. **Problemsituation rechts**: (Doopelte Rechtsrotation) Die Höhe des Teilbaums `R1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `R2` begründet. Diese Situation ist in der Abbildung nicht dargestellt. Sei dieser Teilbaum detailierter mit Wurzel `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 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 5.4 (Seite 129) sind alle AVL-Bäume.

Betrachten wir nun die Höhe des Baumes bzw. seiner Teilbäume für die Abbildung 5.4 (Seite 129).

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`. *Nacher*: 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`.
Nacher: 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`[2]. Die GesamtHöhe ist `n + 3`.
Nacher: 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.

[2]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!

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 5.4 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 der 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 5.2 *Welches ist die Mindestanzahl von Knoten, die ein AVL-Baum der Höhe h hat?*

5.5 Rot-Schwarz-Baum

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 beschrieben, welcher sie *symmetric binary B-trees* nannte. Der heutige Name geht auf Leo J. 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.

5.5.1 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:

1. Jeder Knoten im Baum ist entweder rot oder schwarz. Wenn ein Nachfolger eines Knotens nicht vorhanden ist, dann wird entsprechendes Feld auf `NIL` (`NotInList`) gesetzt.

2. Die Wurzel des Baums ist schwarz.

3. Jedes Blatt (NIL-Knoten) ist schwarz.

4. Die Kinder eines roten Knotens sind alle schwarz.

5. Für jeden Knoten x enthält jeder abwärts gerichtete Pfad zu einem Blattknoten (NIL-Knoten) die gleiche Anzahl an schwarzen Knoten. (Schwarztiefe)

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 5.5 *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. NIL-Blatt) eingeführt. Somit sind alle Knoten mit Suchschlüssel innere Knoten (und haben genau zwei Kinder) und alle Blätter NIL-Knoten.*

5.5.2 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 5.6 *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 NIL-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 NIL-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 verletzt sein könnte, weswegen nun (rekursiv) 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 NIL-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 5.7 *Im folgenden werden wir also nur noch Knoten mit mindestens einem Kind betrachten. Hierbei werden wir eventuelle NIL-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 5.8 *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 5.2 *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.

5.6 Graphen

5.6.1 Einführung und Definition

Bäume sind spezielle Graphen. Bei einem Graphen sind beliebige Beziehungen (Relationen) der Knoten untereinander erlaubt, nicht nur von Vater zu Sohn[3]. 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

[3]Es gibt auch Darstellungen, in denen die Knoten mit Relationen und die Kanten mit Objekten assoziiert werden, z.B. bei den *Constraint Satisfaction Problems* (CSP)

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 5.5 *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 5.6 *Sei* $G = (V, E)$ *ein Graph.*

- *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* v_{i_j} *paarweise verschieden sind.*

- *Ein* **zyklenfreier Graph** *ist ein Graph, der keinen Zyklus enthält.*

Nun können wir auch wieder Bäume definieren, indem wir die Definition von Graphen verwenden:

Definition 5.7 *Ein* **Baum** *ist ein zusammenhängender und zyklenfreier Graph.*

5.6.2 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 im Wesentlichen 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*.

5.6.3 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 boolsche $n \times n$-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.

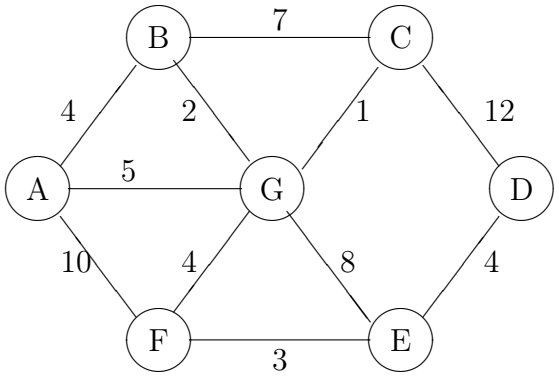
Ein Nachteil ist der hohe Speicherplatzbedarf von $\mathcal{O}(n^2)$. Dies wiegt umso schwerer, je weniger Kanten es im Vergleich zu Knoten gibt.
Adjazenzlisten Bei dieser Implementation verwaltet man eine Liste von Nachbarn eines jeden Knotens. Man hat einen Array von Knoten, die als Einstiegspunkte 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 Implementierungen verwendet man für Graphen die statisch sind, also nach ihrer Erzeugung nicht mehr verändert werden müssen.

Aufgabe Gegeben Sie die Adjazenzmatrix und die Ajdazenzliste für folgenden Graphen an.



5.6.4 Kürzeste Wege in bewerteten Graphen

Zunächst definieren wir einen bewerteten Graphen.

Definition 5.8 *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 versuchen zu 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. Warum ist dem so ? Letztlich muss man fast alle Wege kennen, um garantiert zu wissen, welches

der kürzeste Weg ist. Dies ist dann rekursiv für alle Knoten auf diesem Weg anzuwenden. 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, zu welchem Nachbarknoten sie gehen müssen, um den preiswertesten Weg zu σ zu finden. Dies spiegelt die eben erwähnte Rekursion wider.

Jeder Knoten hat drei (zusätzliche) Attribute[4]:

- *pred* für den Vorgänger Knoten zu dem man gehen muß, um am preiswertesten nach σ zu kommen,
- *cost* hält die bisher minimalen Kosten auf dem Weg zu σ
- *marked* merkt sich, ob der Knoten schon abschließend behandelt wurde, d.h. die minimalsten Kosten feststehen.

```
class Knoten
{
      DAT daten;
      Knoten pred;
      float cost;
      bool marked;
}
```

Der Algorithmus hat nun folgenden Ablauf:

Algorithmus 5.2 *1.* **Initialisierung:** *alle Knoten κ außer dem Startknoten σ bekommen folgende Initialisierung:*

- $\kappa.pred = undefiniert;$
- $\kappa.cost = \infty;$
- $\kappa.marked = false;$

Der Startknoten σ wird wie folgt initialisiert

- $\sigma.pred = \sigma;$
- $\sigma.cost = 0;$
- $\sigma.marked = true;$

2. Bestimme den Rand R, der aus adjazenten Knoten zu σ besteht.

[4]Alternativ kann dies in einer geeigneten Tabelle dargestellt werden.

3. **while** $R \neq \emptyset$ **do**

   - *wähle* $\nu \in R$ *so dass* $\nu.cost$ *minimal, und entferne* ν *aus* R
   - $\nu.marked = true$
   - *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

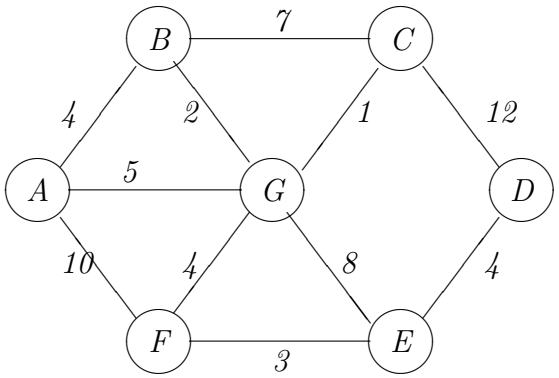
$$\kappa.cost = \nu.cost + c(\nu, \kappa)$$

   und

$$\kappa.pred = \nu.$$

3. *nimm* κ *in* R *mit auf.*

Beispiel 5.2 *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 5.3 *Bestimmen Sie die Ablauffolge (Randmenge, markierte Knoten) für den Algorithmus von Dijkstra für den Startpunkt B.*

5.7 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.

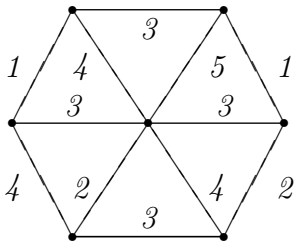
Definition 5.9 *Sei $G(V, E)$ ein ungerichteter Graph. Ein Baum $H(V', E')$ heißt ein* Gerüst *von G, wenn H ein Teilgraph von G ist und alle Knoten von G enthält (wenn also gilt $E' \subseteq E$ und $V' = V$). Wenn H ein Gerüst von G ist, sagt man auch: „G wird von H aufgespannt“.*

Es gibt ein Algorithmus, der ein bzgl. irgendwelchen Kosten minimales Gerüst findet. Dieser Algorithmus, nachfolgend beschrieben, gehört zur Klasse der Greedy-Algorithmen.

Algorithmus 5.3 (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.*

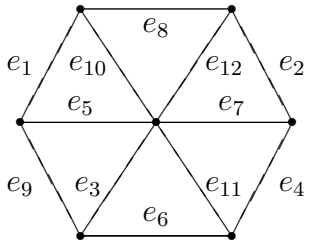
- *Nummeriere die Kanten $e_1, \ldots, e_{|E|}$ nach aufsteigender Länge. Setze $F := \emptyset$.*
- *Für $i := 1, \ldots, |E|$:*
  - *Falls $F \cup \{e_i\}$ nicht die Kantenmenge eines Kreises in G enthält, setze $F := F \cup \{e_i\}$.*

Beispiel 5.3 *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:* $e_8, e_1, e_5, e_3, e_6, e_4, e_2$

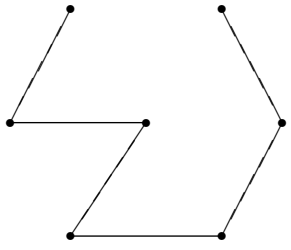
$i = 9$ *Kreis:* e_9, e_3, e_5

$i = 10$ *Kreis:* e_{10}, e_1, e_5

$i = 11$ *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* $F := \{e_1, e_2, e_3, e_4, e_5, e_6\}$*:*



5.8 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 komprimieren. Die AVL-Bäume verwenden als Indiz für eine notwendige Reparatur direkt die Höhe des Baumes wogegen die Rot-Schwarz-Bäume eine eigene Höhe - die „Schwarztiefe“ verwenden (die letztlich von der Baumhöhe abgeleitet wird).

Beide 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:

Algorithmus 5.4 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.

5.9 Aufgaben

1. **Greedy-Algorithmen** (? Punkte)
   Bestimmen Sie für nachfolgendes Problem folgende Elemente des im Tipp aufgeführten 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 30kg. Versuchen Sie gemäß Ihrem Algorithmus den Rucksack mit folgenden Teilen zu bepacken: 7kg,12kg,14kg,16kg,20kg.

Welche Teile würden Sie „von Hand“ (also ohne erkennbares Prinzip) auswählen?

2. **optimale Wege** (? Punkte)
Führen Sie den Algorithmus von Dijkstra mit dem Graphen in Abbildung 5.1 (Seite 147) 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.

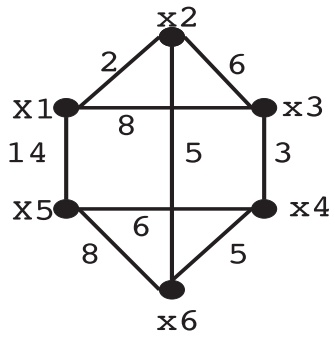


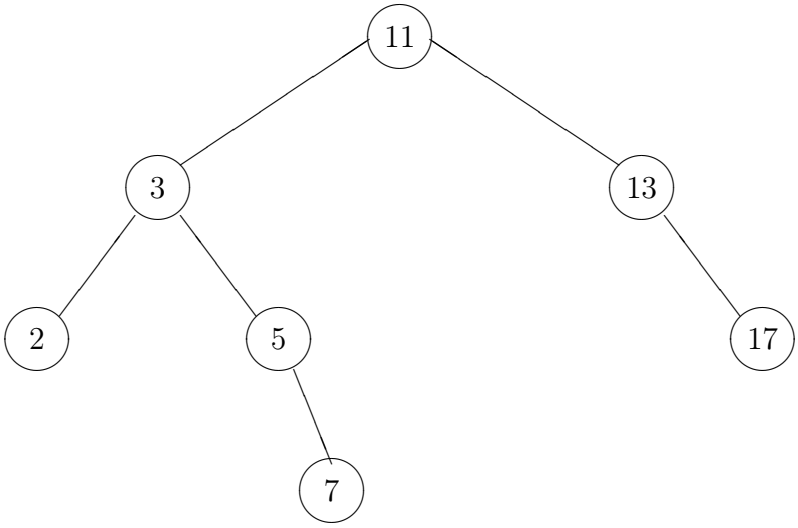
Abbildung 5.1: Optimaler Weg gesucht!

3. **Allgemeiner Baum** (? Punkte)
Ein allgemeiner Baum kann pro Knoten beliebig viele Söhne haben und kann auch als gerichteter Graph ohne Zyklen aufgefasst werden. Gesucht ist nun ein solcher allgemeiner Baum mit folgenden Eigenschaften:

   - Ein preorder-Durchlauf beginnend am Knoten mit Markierung C ergibt die Knotenfolge $C, X, R, T, S, U, W, J, K, L, V, M$.
   - Ein postorder-Durchlauf durch den Baum erzeugt die Konotenfolge $R, T, U, W, S, X, J, V, L, M, K, C$.

   Zeichnen Sie den durch die beiden Folgen gegebenen allgemeinen Baum und begründen Sie, wie Sie dies aus der Vorgabe abgeleitet haben. Eine Zeichnung ohne diese Herleitung wird nicht gewertet!

4. **Suchbaum** (? Punkte)
Gegeben sei folgender binärer Suchbaum:



(a) Der Algorithmus `Tree-Minimum(x)` gibt das Element mit dem kleinsten Schlüssel in einem binären Suchbaum zurück.

```
Tree-Minimum(x)
1 while x.links =!= NIL
2       x = x.links
3 return x
```

Geben Sie eine rekursive Version des Algorithmus `Tree-Minimum(x)` an. Der Algorithmus ist in Pseudo-Code (oder Erlang/OTP) anzugeben.

(b) Gegeben sei folgender Traversierungsalgorithmus:

```
  Tree-Walk(x)
1 if x =!= NIL
2    Tree-Walk(x.rechts)
3    print x.schlüssel
4    Tree-Walk(x.links)
```

Welche Schlüsselfolge erhält man, wenn `Tree-Walk(x)` auf der Wurzel des obigen Baums aufgerufen wird ?

5. **AVL-Baum** (? Punkte)

(a) Definieren Sie den Begriff *AVL-Baum*.

(b) Beschreiben Sie ein Verfahren in Pseudocode, mit dem man testen kann, ob ein vorgegebener binärer Suchbaum ein AVL-Baum ist. Geben Sie die Laufzeitkomplexität Ihres Verfahrens an.

(c) Gilt für jeden Knoten eines AVL-Baums, dass sich die Anzahl der Knoten seines linken und rechten Teilbaums um höchstens 1 unterscheidet ? Begründen Sie Ihre Antwort!

6. **AVL-Bäume** (? Punkte)
In dieser Aufgabe ist ein AVL-Baum zu erstellen. Fügen Sie in der folgenden, vorgegebenen Reihenfolge die Elemente in einen leeren AVL-Baum ein. Als Ordnungsrelation gilt die lexikographische Ordnung. Erläutern Sie, wann und welche Rotationen stattfinden (Der Ablauf des Einfügens muss klar sein)! Zeichnen Sie den Baum nach jeder Rotation neu auf.

Söndre_Strömfjord, Berlin, Casablanca, Dakar, El_Salvador, Oslo, Windhuk

7. **AVL-Bäume** (? Punkte)
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.

8. **AVL-Baum** (? Punkte)
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.

9. **AVL-Baum** (? Punkte)
   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.

10. **AVL Baum** (? Punkte)
    Student Karl behauptet, dass in einem AVL-Baum der Höhe h sich jedes Blatt in der Ebene h oder h-1 befindet. Widerlegen Sie Karl durch ein Gegenbeispiel oder zeigen Sie diese Behauptung durch einen Beweis mittels vollständiger Induktion. Die Wurzel sei dabei Ebene eins und deren Söhne auf Ebene zwei usw.

11. **AVL-Baum** (? Punkte)
    Fügen Sie folgende Zahlen **in der vorgegebenen Reihenfolge** in einen leeren AVL-Baum ein: $(7, 6, 5, 1, 3, 9, 8)$ und löschen Sie dann die Zahlen $(1, 5, 3)$ **in der vorgegebenen Reihenfolge**. Stellen Sie die AVL-Bäume dar, die sich als Zwischenschritte ergeben. Geben Sie die durchgeführten Rotationsoperationen und die zuvor vorhandenen Höhenunterschiede für die relevanten Knoten an.

12. **Datenverwaltung** (? Punkte)
Gesucht ist eine Datenstruktur zur Verwaltung einer dynamischen Menge von Schlüsseln x mit $x \in \mathbb{N}$. Die Datenstruktur soll die folgenden drei Operationen jeweils in worst-case Laufzeit $O(logn)$ und best-case Laufzeit $\Omega(1)$ bereitstellen. n bezeichnet dabei die Anzahl der Schlüssel, die sich aktuell in der dynamischen Menge befinden.

Einfügen(x) : Falls sich der Schlüssel x noch nicht in der dynamischen Menge befindet, so wird x eingefügt. Ansonsten bleibt die Menge unverändert.

Löschen(x) : Falls sich ein Schlüssel x in der dynamischen Menge befindet, so wird dieser aus der dynamischen Menge entfernt. Ansonsten bleibt die Menge unverändert.

Suchen(x) : Es wird genau dann *true* zurückgegeben, wenn sich der Schlüssel x in der dynamischen Menge befindet.

Beschreiben Sie, wie Ihre Datenstruktur aufgebaut ist und wie die obigen Operationen realisiert werden. Setzen Sie anschließend Ihre Lösungsskizze in Pseudocode um, sofern Sie nicht eine allgemein, aus der Vorlesung bekannte Datenstruktur verwenden oder bei einer Kombination von diesen, um deren Zusammenwirken darzustellen. Erläutern sie die Korrektheit Ihrer Lösung und wieso die gegebenen Laufzeitschranken eingehalten werden.

Hinweis: Verwenden Sie aus der Vorlesung bekannte Strukturen geschickt wieder, also nicht unbedingt nur diejenigen, die diesem Kapitel zugeordnet sind. Sie dürfen darüber hinaus davon ausgehen, dass für n immer eine obere Schranke abgeschätzt werden kann.

Kapitel 6

Hashverfahren

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 meist 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. Wenn die Schlüssel dazu nicht direkt verwendbar sind, wird ein Berechnungsverfahren eingesetzt: eine Hashfunktion. Mit einer Hashfunktion h wird aus dem Schlüssel k eine Hashadresse $h(k)$ (positive ganze Zahl) berechnet. Die Hashadresse gibt den Index in einem Array (Feld) an, wo der Datensatz abgespeichert werden kann bzw. abgespeichert ist. Das Feld wird auch Hashtabelle genannt. 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.

6.1 Einführung

Eine Hashfunktion unterliegt folgenden Bewertungskriterien:

Gleichverteilung/Zufälligkeit (geringe Kollisionsgefahr): Ideal ist injektive Hashfunktion (verschiedene Objekte haben verschiedene Schlüssel). Dies benötigt jedoch meist zu viel Platz. Daher ist es nicht ausgeschlossen, dass zwei verschiedene Objekte den selben Schlüssel haben: Kollision (spezielle Überlaufverfahren erforderlich)

Geringer Speicherbedarf : Ideal ist surjektive Hashfunktion (das ganze Feld abdecken), d.h. im Schnitt hoher Belegungsfaktor der Hashtabelle: (Anzahl Elemente)/(Anzahl der verfügbaren Speicherplätze) ist dicht an 1.

Streuwirkung : Ähnliche Eingabewerte liefern völlig verschiedene Hashwerte (gute Streuwirkung), d.h. die Anzahl der Kollisionen ist gering, da die Hashfunktion dadurch Häufungen fast gleicher Schlüssel möglichst gleichmäßig auf den Adressbereich streut.

Effizienz - Die Funktion muss schnell (und damit meist einfach) berechenbar sein.

Später werden wir Strategien besprechen, um mit Kollisionen umzugehen. Zunächst wollen wir aber Hashfunktionen finden, so dass möglichst wenig Kollisionen auftreten.

6.1.1 Beispiel mit Matrikelnummern

Wir wollen den Unterschied zwischen einer guten und einer schlechten Hashfunktion kennen lernen. Dazu bilden wir ein Array als Prüfungsdatenbank, in das Informationen von Studierenden eingetragen werden. Zur Einsortierung (als Arrayindex) wird die erste Ziffer der Matrikelnummern verwendet und in eine Tabelle (0-9) eingetragen. Möglicherweise erhalten wir dabei viele Kollisionen, ohne das die Tabelle in jedem Index Einträge hat und bzgl. jedem Index gleich viele Einträge.

Dann wird die letzte Ziffer der Matrikelnummern als Arrayindex verwendet und wieder in eine Tabelle (0-9) eingetragen. Dies sollte eine wesentlich bessere Verteilung ergeben. Die Tabelle wird wahrscheinlich fast gleichmäßig aufgefüllt. Dies ist die gute Hashfunktion, da zwar immer noch Kollissionen auftreten, diese aber minimiert sind.

Die minimale Anzahl an Kollissionen in diesem Beispiel ist *Anzahl Studierende - 10*, z.B. bei 30 Studierenden wären 20 Kollissionen (3 Einträge pro Arrayindex) minimal. Die maximale Anzahl ist *Anzahl Studierende*, z.B. bei 30 Studierenden 30 Kollissionen (30 Einträge in einem Arrayindex). Die beste Hashfunktione erreicht also die minimale Anzahl an Kollissionen.

Eine gute Hashfunktion sollte also folgende Eigenschaften haben:

- möglichst surjektiv sein (also das ganze Array abdecken)
- möglichst injektiv sein (also keine Kollisionen bzw. pro Arrayindex die gleiche Anzahl an Kollissionen)
- die Berechnung muss schnell (und damit meist 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.

6.1.2 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

$$P(1) = 1 = \frac{m-0}{m}.$$

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

$$P_{keineKollision} = P(1) \cdot P(2) \cdot \ldots \cdot P(n) = \prod_{i=1}^{n} P(i),$$

oder

$$P_{Kollision} = 1 - \frac{m(m-1)\ldots(m-(n-1))}{m^n} = 1 - \frac{\prod_{i=1}^{n}(m-(i-1))}{m^n} = 1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n}.$$

Hiermit kann man z.B. die Wahrscheinlichkeit ausrechnen, mit der zwei oder mehr Leute am gleichen Tag Geburtstag haben, bei n anwesenden Personen:

$$P_{Kollision} = 1 - \frac{\prod_{i=0}^{n-1}(365 - i)}{365^n}.$$

Z.B. bei zwei Personen:

$$P_{Kollision} = 1 - \frac{\prod_{i=0}^{2-1}(365 - i)}{365^2} = 1 - \frac{(365 - 0)(365 - 1)}{365^2} = 0,00274 \approx 0,274\%.$$

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. Für das Hashing bedeutet dies, dass Kollissionen praktisch unvermeidbar sind.

6.2 Hashfunktionen

Welche Hashfunktion 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

$$\begin{array}{rcl} h : D & \longrightarrow & \{0, \ldots, m-1\} \\ k & \mapsto & h(k) \end{array}$$

6.2.1 Divisions(-Rest-)methode

Hier definieren wir die Hashfunktion als

$$h(k) = k \bmod m.$$

Die Qualität von dieser Hashfunktion hängt von m ab. Diese Funktion hat die Eigenschaft, dass sie aufeinanderfolgende Schlüssel auf aufeinanderfolgende Indizes abbildet. Das kann ungünstig sein. 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 von m als Primzahl 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.

6.2.2 Multiplikative Methode

Bei diesem Verfahren wird der Schlüssel k mit einer irrationalen Zahl Θ multipliziert, um eine willkürliche Zahl bzw. Zufallszahl zu simulieren. Als Wert der Hashfunktion nimmt man einige der Nachkommaziffern vom Ergebnis der Multiplikation bzw. sofern die Multiplikation zu große Werte erzeugt wird diese Methode mit der Division-Rest-Methode kombiniert[1]. Besonders gleichmäßig werden die Schlüssel verteilt, wenn man für Θ den goldenen Schnitt

$$\Theta = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

wählt. 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 $(k \cdot \Theta) mod 1000$.

6.2.3 Mittel-Quadrat-Methode

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.

Beispiel 6.1 *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.*

[1] Der Trick besteht darin, statt (a * b * c) mod m zu rechnen, ((((a mod m) * (b mod m)) mod m) * (c mod m)) mod m zu berechnen.

k	$k \ mod \ m$	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 dem Faktor Zufall versucht wird, eine gleichmässige Verteilung zu erreichen, damit die minimale Anzahl an Kollissionen erreicht wird. Wünschenswert wäre eine Hashfunktion, die unabhängig von der Charakteristik der Schlüssel, also z.B. der Verteilung der Schlüssel selbst[2], eine ausgeglichene Verteilung auf der Hashtabelle erzeugt (und in dem Sinne die Verteilung der Schlüssel selbst neutralisiert).

6.3 Kollisionsvermeidungsstrategien

Kollisionen lassen sich im allgemeinen nicht gänzlich vermeiden. Daher muss man sich damit beschäftigen, was im Falle einer Kollision passieren soll. Für das neu einzutragende Element, dessen Platz im Array schon belegt ist, muss irgendwo anders untergebracht werden. Dabei soll man es später auch leicht wiederfinden. Grundsätzlich gibt es zwei Methoden: Sie werden oft als *Offenes Hashing* und *Geschlossenes Hashing* bezeichnet.

1. Man organisiert die Hashtabelle so, dass die mehrfachen Einträge zu einem Index unter einer Adresse als verkettete Liste realisiert werden (Chaining, offenes Hashing)

2. Man errechnet aus dem Original-Hashwert einen neuen Hashwert und versucht den Schlüssel dort einzutragen (Re-Hashing, Probing, Sondierung, geschlossenes Hashing, offene Adressierung)

Diese Begriffe *Offenes Hashing* und *Geschlossenes Hashing* werden in der Literatur nicht einheitlich verwendet. Daher verwenden wir hier die Begriffe *Separate Chaining* und *offene Adressierung*. Diese Begriffe sind in der Literatur einheitlich.

6.3.1 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 (Fächer), welche Elemente von verketteten Listen sind. Der Wert der Hashfunktion, der Hashcode, ist ein Index auf ein Array, der Referenzen auf Anfangselemente von verketteten Listen hält. Ist dieses Anfangselement noch frei, dann wird

[2]Wenn z.B. die ASCII-Werte der Zeichen eines Schlüssels aufaddiert werden, könnte ein enges Intervall an Zahlen entstehen.

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 6.2 *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*			

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.

6.3.2 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 Adrssierung 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

$$h_0(), h_1(), h_2(), \ldots, h_r().$$

Dabei bezeichnet $h_j(key) = h(key) - s(j, key)$ den j-ten Sondierungsversuch mit der Sondierungsfunktion $s(j, key$. Findet bei Anwendung aller Hashfunktionen eine Kollision statt, so kann das neue Element nicht einsortiert werden. Im „Normalfall“ ist die Folge der Hashfunktionen so gestaltet, dass dieser Fall als Überlauf des Arrays bzw. der größenbeschränkten Hashtabelle gewertet werden kann.

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, d.h. mit einem Flag markierte

Felder stellen nicht das Ende der Suche dar: Damit das Suchen also wohldefiniert ist, muss beim Löschen dieser Flag gesetzt werden. Im Falle eines neuen Eintrags kann ein mit einem Flag besetztes Feld als leer interpretiert werden.

Im folgenden beschreiben wir einige Verfahren um eine neue Adresse im Falle einer Kollision zu finden.

6.3.3 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

$$h_i(k) = (h_0(k) - i) \bmod m.$$

, wobei $s(j,k) = j$ ist. Dieses komplizierte Konstukt 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

$$h_i(k) = (k - i) \bmod m.$$

Eigentlich verallgemeinert man das lineare Sondieren zu folgender Formel mit gegebener Konstante $c \in \mathbb{N}$

$$h_i(k) = (k - c \cdot i) \bmod m.$$

, also $s(j,k) = c \cdot j$. Dies bringt jedoch keine wirkliche Verbesserung des Verfahrens. Voraussetzung ist außerdem, dass c und m teilerfremd sind, damit alle Zellen getroffen werden.

Das Verfahren führt dazu, dass im Falle von Kollisionen Ketten mit Abstand c entstehen.

6.3.4 Quadratisches Sondieren

Beim quadratischen Sondieren werden neue Adressen mit quadratischem Abstand erzeugt. Dies führt dazu, dass keine Ketten mehr entstehen. Die Folge der Hashfunktionen lautet hier

$$h_i(k) = (h_0(k) - s(j,k)) \bmod m.$$

mit $s(j,k) = (-1)^j \cdot \left\lceil \frac{j}{2} \right\rceil^2$ Wählt man für m eine Primzahl der Form $m = 4 \cdot j + 3$, so werden alle Zellen getroffen (Güting/Dieker geben hier als Referenz einen Artikel

von Radke an). Konkret hat man also z.B. für $m = 1019$ (Primzahl und bei Division durch 4 Rest 3) und für

$$h_0(k) = k \bmod m$$

$$\begin{aligned} h_0(k) &= k \bmod 1019 \\ h_1(k) &= k - 1 \bmod 1019 \\ h_2(k) &= k + 1 \bmod 1019 \\ h_3(k) &= k - 4 \bmod 1019 \\ h_4(k) &= k + 4 \bmod 1019 \end{aligned}$$

Beispiel 6.3 *Gegeben seien Elemente mit den folgenden Schlüsseln:*

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Die Arrayelemente nennen wir $a[.]$*. Als Hashfunktion* $h()$ *verwenden wir*

$$h(k) = k \ mod \ m \quad mit \ m = 7.$$

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*

$$h_1(77) = (h_0(77) - 1) \ mod \ 7 = 6.$$

Diese Zelle ist noch frei und wir bekommen

0	*1*	*2*	*3*	*4*	*5*	*6*
63		*51*	*24*			*77*

Das Element $k = 85$ *liefert*

$$h_0(90) = 6.$$

Diese Zelle ist nun gerade belegt worden. Hätte man 90 vor 77 einsortiert, dann wäre 90 an die Position 6 gekommen. Man sieht, dass die Belegung der Hashtabelle von der Reihenfolge der Eingabedaten abhängt. Wir wenden also h_1 *an und erhalten*

$$h_1(90) = (h_0(90) - 1) \ mod \ 7 = (6 - 1) \ mod \ 7 = 5.$$

Hier sortieren wir also 90 ein

0	*1*	*2*	*3*	*4*	*5*	*6*
63		*51*	*24*		*90*	*77*

Nun soll noch die 83 *untergebracht werden. Anwendung von* h_0 *ergibt*

$$h_0(83) = 83 \ mod \ 7 = 6.$$

Diese Zelle ist aber schon vergeben. Welches h_i *bringt* 83 *auf einen freien Platz?*

Ein zahlentheoretisches Ergebnis von Radke aus dem Jahr 1970 zeigt, dass alle Zellen getroffen werden, wenn m eine Primzahl der Form

$$m = 4 \cdot j + 3$$

ist.

6.3.5 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 eine Kollision zu erzeugen unabhängig sind. Oft begnügt man sich damit, eine *intuitiv* unabhängige zweite Hashfunktion $h'(.)$ zu finden.

Ist m eine Primzahl und

$$h(k) = k \bmod m,$$

dann ist

$$h'(k) = 1 + (k \bmod (m-2))$$

eine gute Wahl. Die Folge der Hashfunktionen definiert man dann wie folgt:

$$h_j(k) = (h(k) - s(j,k) \bmod m.$$

mit $s(j,k) = j \cdot h'(k)$. Hieran sieht man, dass h' nicht Null sein darf (sonst ändert sich an den h_j im Gegensatz zu h ja nichts. Dafür wird bei h' auch 1 addiert.

Beispiel 6.4 *Gegeben seien Elemente mit den folgenden Schlüsseln:*

$$\kappa = \{24, 51, 63, 77, 85, 99\}.$$

Die Arrayelemente nennen wir $a[.]$. *Als Hashfunktion* $h()$ *verwenden wir*

$$h(k) = k \ mod \ m \quad mit \ m = 7.$$

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.*

$h_1(77) = (h(77) - (1 + h'(77)) \ mod \ 7 = (0 - (1 + (77 \ mod \ 5))) \ mod \ 7 = (0 - (1+2) \ mod \ 7) = 4.$

Dies ergibt eine Kollision. Wir wenden nun die nächste Hashfunktion h_2 *an und erhalten:*

$$h_2(77) = (h(77) - 2 \cdot h'(77)) \ mod \ 7 = (0 - 2 \cdot 3) \ mod \ 7 = -6 \ mod \ 7 = 1.$$

Diese Zelle ist frei und wir tragen ein

0	*1*	*2*	*3*	*4*	*5*	*6*
63	*77*	*51*	*24*			

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 erreichen.

6.4 Löschen von Elementen

Das Löschen von Elementen aus einer Hashtabelle mit offener Adressierung erfordert einen Kniff. Folgendes Beispiel beschreibt die Problematik und den Kniff.

Beispiel 6.5 *Gegeben seien Elemente mit den folgenden Schlüsseln:*

$$\kappa = \{63, 77\}.$$

Als Hashfunktion verwenden wir double hashing mit

$$h(k) = k \ mod \ 7 \ und \ h'(k) = 1 + (k \ mod \ 5).$$

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*
$14^{Flag=X}$	$Flag=0$	$Flag=0$	$77^{Flag=0}$	$Flag=0$	$Flag=0$	$Flag=0$

6.5 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 $\bmod m$ hat man nun $\bmod m_{neu}$.

6.6 Fazit aus Sicht der Konstruktionslehre

Suchalgorithmen[3], 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

[3]Dies ist `Sedgewick` entnommen worden.

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.

6.7 Aufgaben

1. **Konstruktionsprinzip** (? Punkte)

   (a) Suchalgorithmen, die mit Hashing arbeiten, bestehen aus zwei Teilen. Welche sind dies?

   (b) Welche Ziele verfolgen die einzelnen Teile?

   (c) Warum ist Hashing ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf?

   (d) Ist Hashing eine geeignete Datenstruktur für die Speicherung von Daten, wenn bekannt ist, dass die gespeicherten Daten häufiger sortiert ausgegeben werden müssen ? Begründen Sie Ihre Antwort.

2. **Geschlossenes Hashing** (? Punkte)

   (a) Gegeben sei folgende Hashfunktion

   $$H : \mathbb{N}_0 \to \mathbb{Z}_{11}, H(k) = -3k \ mod \ 11$$

   Welche Index-Plätze in der Hashtabelle werden niemals belegt ? Begründen Sie ihre Antwort!

(b) Seien X und Y endliche Mengen. Kann es kollisionsfreies Hashing geben, wenn $H : X \to Y$ mit $|X| \geq |Y|$, wobei alle Schlüssel aus X benutzt werden ?

3. **Geschlossenes Hashing** (? Punkte)
Fügen Sie die Zahlen $13, 5, 27, 60, 2, 65, 8, 37, 18$ in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod 7$ in eine Hashtabelle der Größe $N = 7$ (nummeriert von 0 bis 6) 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.

4. **Geschlossenes Hashing** (? Punkte)
Gegeben sei eine leere Hashtabelle mit $m = 13$. Nachfolgend betrachten wir Hashing mit verketteten Listen zur Kollisionsverwaltung. Wählen Sie unter den unten aufgeführten Hashfunktionen eine gute aus und begründen Sie ihre Wahl.

   Stellen Sie weiterhin die Hashtabelle nach dem Einfügen der Schlüssel in der Reihenfolge $12, 23, 13, 56, 26, 45, 10$ dar. Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h_* an.

   $$h_{11}(x) := (3x + 1) \bmod m \qquad h_{12}(x) := ((2x + 7) \bmod 11) \bmod m$$

5. **Offenes Hashing** (? Punkte)
Gegeben sei eine leere Hashtabelle mit $m = 13$. Nachfolgend betrachten wir Hashing mit offener Adressierung. Wählen Sie unter den unten aufgeführten Hashfunktionen eine gute aus und begründen Sie ihre Wahl.

   Stellen Sie weiterhin die Hashtabelle nach dem Einfügen der Schlüssel in der Reihenfolge $12, 23, 13, 56, 26, 45, 10$ dar. Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h_* an.

   $$h_{21}(x; i) := (2 * h_{12}(x) + 26 * i) \bmod m \qquad h_{22}(x; i) := (h_{11}(x) + i) \bmod m$$

6. **Lineares Sondieren** (? Punkte)
Fügen Sie die Zahlen $12, 7, 22, 14, 8, 4, 11, 3$ in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod 8$ in eine Hashtabelle der Größe $N = 8$ (nummeriert von 0 bis 7) ein. Die Kollisionsbehandlung erfolge durch lineares Sondieren ($h_i(k) = (h(k) + i) \bmod 8$).

   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.

7. **Kollisionsbehandlung** (? Punkte)
Fügen Sie die Zahlen

$$107, 20, 37, 46, 1, 2, 4, 50, 0$$

in dieser Reihenfolge in eine Hashtabelle ein. m gebe dabei die Größe der Hashtabelle an und b die Anzahl der möglichen Elemente pro Index. Die Kollisionsbehandlung ist eine Kombination von Verkettung und offener Adressierung. Als Hashfunktion soll die Divisions-(Rest-)Methode verwendet werden.

   (a) Es sei $b = 2, m = 5$ und verwenden Sie lineares Sondieren ($s(j, k) = j$).

   (b) Es sei $b = 1, m = 11$ und verwenden Sie Double Hashing ($s(j, k) = j * h'(k)$) mit $h'(x) = 2 * h(x)$ und $h'(x) = 1$, falls $2 * h(x) = 0$.

   (c) Welche Eigenschaften sollte eine Hashfunktion haben und was muß zusätzlich für $h(x), h'(x)$ beim Double Hashing gelten ? Beurteilen Sie die grundsätzliche Eignung von $h'(x)$ aus Aufgabenteil (b) als Hashfunktion sowie im Zusammenhang mit Double Hashing.

Das Einfügen ist jeweils so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie die dazu die Zwischenergebnisse der Sondierungsfolge $h_i(x)$ an und abschließend die Hashtabelle.

8. **Quadratisches Sondieren** (? Punkte)
Fügen Sie die Zahlen

$$9, 11, 40, 22, 26, 43, 36, 14, 5$$

in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = kmod9$ in eine Hashtabelle der Größe $N = 9$ (nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge durch quadratisches Sondieren ($h_i(k) = (h(k) + i^2) \bmod 9$).

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. **Mittel-Quadrat-Methode** (? Punkte)
Fügen Sie die Zahlen

$$13, 6, 27, 11, 16, 17, 19, 18, 42$$

in dieser Reihenfolge unter Verwendung folgender 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.

10. **Personalnummer** (? Punkte)
Die etwa 8000 Mitarbeiter eines Unternehmens mit mehreren Standorten in Deutschland sollen mit einer Hashtabelle verwaltet werde. Als Schlüssel wird die Personalnummer gewählt, die folgenden Aufbau besitzt: `ttmmjjjjaaappppp`. `ttmmjjjj` stellt das Geburtsdatum dar, bestehend aus Tag, Monat und Jahr. `aaa` gibt die Abteilung an, der der Mitarbeiter zugeordnet ist, und `ppppp` die Postleitzahl des Standorts. Es soll ein offenes Hashverfahren mit quadratischem Sondieren gewählt werden.

    (a) Welches Problem entsteht, wenn als Tabellengröße $m = 10000$ und die übliche Hashfunktion $h(k) = k \bmod m$ gewählt wird.

    (b) Geben Sie eine Hashfunktion an, die das Problem aus (a) vermeidet. Die Tabellengröße $m = 10000$ soll dabei unverändert bleiben. Begründen Sie Ihre Entscheidung.

    (c) Wie lässt sich das Problem aus (a) vermeiden, falls die Tabellengröße leicht verändert werden darf ?

11. **Offenes Hashing** (? Punkte)
In einem kleinen Programmkino mit $m = 23$ Plätzen wird ein Film gezeigt. Um dem Ansturm Herr zu werden, sollen die Plätze mit einem offenen Hashverfahren auf die Wartenden verteilt werden. Als Schlüssel werden dabei nur die beiden letzten Ziffern der Personalausweisnummer verwendet. Betrachten Sie die folgenden beiden Hashfunktionen:

$$h_1(x) := \text{ Quersumme}(x) \qquad h_2(x) := x \bmod 23$$

    (a) Diskutieren Sie, inwieweit h_1 und h_2 die vier Bedingungen, die an eine sinnvolle Hashfunktion gestellt werden, erfüllen.

    (b) Welche Platznummern erhalten die Besucher, wenn sie in der nachfolgend gegebenen Reihenfolge das Kino betreten ?
    $7; 16; 61; 87; 69; 22; 4; 4$
    Verwenden Sie jeweils folgende Hashfunktionen:

    - lineares Sondieren mit h_1 als Hashfunktion (und $c = 1$)
    - lineares Sondieren mit h_2 als Hashfunktion (und $c = 1$)
    - Doppelhashing mit h_1 als erster und h_2 als zweiter Hashfunktion. Inwieweit muss h_2 geändert werden, um Probleme zu vermeiden ?

(c) Warum ist ein geschlossenes Hashing, in dem geschildertem Szenario, nicht sinnvoll ?