

Diplomarbeit

„Algorithmenbaukasten zur Unterstützung des Blended Learning im Informatikunterricht“

Ausgeführt zum Zweck der Erlangung des akademischen Grades eines
Dipl.-Ing. (FH) für Telekommunikation und Medien
am Fachhochschul-Diplomstudiengang Telekommunikation und Medien St. Pölten

unter der Leitung von

FH-Prof. Dipl.-Ing. Georg Barta

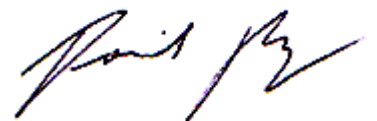
von

Daniel Berger

tm0110038006

St. Pölten, am 10. September 2005

Unterschrift:



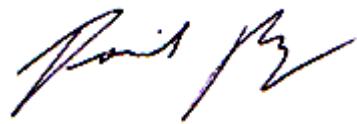
Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der vom Begutachter beurteilten Arbeit überein.

St. Pölten, 10. September 2005



.....
Unterschrift

Inhaltsverzeichnis

1 Zusammenfassung	5
2 Abstract	6
3 Übersicht	7
4 Der Algorithmus.....	10
4.1 Geschichte	12
4.2 Informatik	13
4.3 Erster Computeralgorithmus	15
4.4 Algorithmenanalyse.....	16
5 Blended Learning	17
5.1 Begriffserklärung	19
5.2 Blended Learning im universitären Umfeld	19
6 Methodisches Vorgehen	21
6.1 Problembenennung.....	21
6.2 Zielsetzung.....	21
6.3 Algorithmenauswahl.....	22
6.4 Programmierung	23
6.4.1 HTML.....	23
6.4.2 Javascript	32
6.4.2.1 Hauptseite.....	32
6.4.2.2 Algorithmenseite	36
6.4.2.3 Vergleichsseite	39
6.4.3 CSS	42
7 Vorstellung der Algorithmen	44
7.1 Notationen.....	44
7.2 Sortieralgorithmen.....	47
7.2.1 Bubblesort	48
7.2.2 Insertionsort.....	52

7.2.3 Quicksort	55
7.2.4 Mergesort	58
7.2.5 Shellsort	61
7.2.6 Bucketsort	66
7.2.7 Heapsort.....	70
7.3 Suchalgorithmen	74
7.3.1 Lineare Suche	74
7.3.2 Binäre Suche.....	77
7.3.3 Interpolierte Suche	80
7.4 Hashing.....	83
7.4.1 Direkte Verkettung.....	85
7.4.2 Lineares Sondieren	88
7.4.3 Doppeltes Hashing	92
7.5 Binärbaum.....	96
7.5.1 Suchen	96
7.5.2 Traversieren	99
7.5.2.1 Preorder.....	99
7.5.2.2 Inorder	100
7.5.2.3 Postorder	100
Anhang A: Abkürzungsverzeichnis.....	101
Anhang B: Literaturverzeichnis	102
Anhang C: Internetquellen	103
Anhang D: Internetrecherche.....	103
Anhang E: Abbildungsverzeichnis	104
Anhang F: Stichwortverzeichnis.....	107

1 Zusammenfassung

E-Learning wird in unserer heutigen Informationsgesellschaft immer wichtiger. Studien zufolge kann ein Unternehmen aus dem Wissensstand eines/einer neu eingestellten Mitarbeiters/Mitarbeiterin im Durchschnitt gerade einmal für drei Jahre Nutzen ziehen. Danach ist das Wissen überholt. Kurz zusammengefasst bedeutet es, dass wir ein Leben lang lernen müssen.

Es gibt viele Arten der Weiterbildung. Eine davon ist E-Learning. Unter diesem Begriff, auch Online-Lernen oder Web-based Teaching (WBT) genannt, werden heute die verschiedensten Formen der Verschmelzung von Ausbildung und Internet verstanden, wobei Angebot und Vermittlung von Wissensinhalten unter Einsatz einer zentralen Funktion des Computers realisiert wird, eben der Möglichkeit, das Internet in allen seinen Formen zu nutzen. Die dabei geschaffenen Lernumgebungen basieren weitestgehend auf den Diensten des Internet, also WWW, E-Mail, etc., als primäres Medium für Kommunikation und Präsentation.

Ein Teilbereich des E-Learning, der vorwiegend an universitären Einrichtungen praktiziert wird, ist das Blended Learning oder auf Deutsch Hybrides Lernen. Darunter versteht man den Mix aus dem herkömmlichen, durch einen Lehrerbeauftragten gestalteten Unterricht sowie Onlineunterricht.

Damit beschäftigt sich auch diese Arbeit. Zur Unterstützung des Informatikunterrichts soll ein Algorithmenbaukasten entstehen, mit dessen Hilfe sich die Studierenden die wichtigsten Algorithmen der Informatik zuhause und in Ruhe im Sinne des Blended Learning erarbeiten können.

2 Abstract

The importance of E-Learning in our information society will increase over the next years. Studies have shown that the knowledge a new coworker is able to bring to the workplace is worth about three years. After that period the knowledge is considered old. This means that we should learn for the duration of our whole life.

There are many different ways to learn something new. One possibility is E-Learning. Also known as Online-Learning or Web-based Teaching (WBT), it includes every method that connects education and the different services of the internet, i.e. WWW or emails, for communication and presentation.

A part of E-Learning often used by universities is Blended Learning. This is a mix of the traditional way of teaching, where a tutor teaches a class of students, with online teaching.

The topic of this thesis is to produce a set of tools to teach computer science, so that students can learn the subject matter at home and during their own time.

3 Übersicht

Um sich die in einer Vorlesung oder Übung übermittelten Inhalte besser einprägen zu können, sollten Studierende aufmerksam zuhören und die ihres Erachtens wichtigsten Informationen niederschreiben. Haben sie den Lehrstoff das erste Mal durchgenommen, kann das Verständnis für das Thema fehlen bzw. noch nicht alles wieder abrufbar erlernt sein. Folglich ist es notwendig, das zu erlernende Gebiet zuhause selbst noch einmal aufzubereiten, um es besser erlernen und verstehen zu können.

Dies gilt natürlich auch für die Informatik-Vorlesung, ganz besonders im Bereich der so genannte Algorithmen. Diese gehören zu den grundlegendsten Dingen, die es auf diesem Gebiet gibt. Mit Klassikern wie Bubblesort, Hashing oder Traversieren von Bäumen sind Studierende mit Informatik-Background im Laufe des Studiums in Kontakt gekommen. Und nicht immer hat man dies, wie im letzten Absatz bereits beschrieben, gleich von Anfang an verstanden.

Allgemein wird in Zukunft immer mehr auf Lehrutensilien gesetzt, mit deren Hilfe sich die Studierenden den gehörten Lehrstoff erarbeiten können – Stichwort: Blended Learning.

Im Jahr 2003 wurde die 17. und bisher letzte Sozialerhebung des Deutschen Studentenwerks (DSW) durchgeführt. Hauptanliegen dieser Erhebung ist es, die soziale und wirtschaftliche Lage der Studierenden systematisch zu erfassen. Für diese Arbeit von besonderem Interesse ist die Erfassung des zeitlichen Aufwands den ein/e Studen/in durchschnittlich für das Studium aufbringen muss.

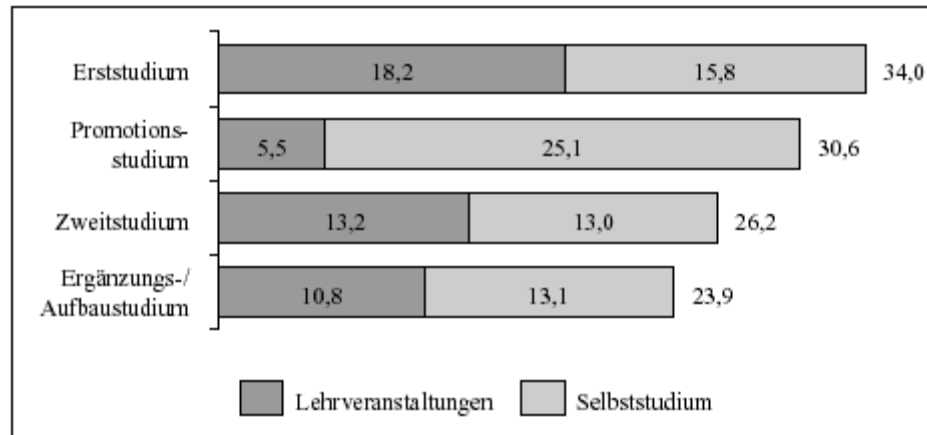


Abb. 3.1: Studienaufwand nach Art des Studiums in Stunden/Woche

Im Schnitt wendet ein/e Student/in im Erststudium, etwas mehr als 18 Wochenstunden für Lehrveranstaltungen auf, das bedeutet er oder sie sitzt in Vorlesungen oder Übungen. Für den Bereich des Selbststudiums wendet er oder sie immerhin noch fast 16 Wochenstunden auf. Darunter fallen per Definition sämtliche Tätigkeiten, die sonst für das Studium aufgewendet werden, beispielsweise Lernen, anfertigen von Haus- oder Abschlussarbeiten, Lektüre von Fachliteratur oder Besuch von Sprechstunden.

Diese Werte wurden durch alle an der Studie beteiligten Studierenden ermittelt. Damit stellen sie Durchschnittswerte dar und es ist nicht von Bedeutung wieviele Hochschulsesemester bereits absolviert wurden, welche Studienrichtung gewählt wurde, oder ob eine Universität oder Fachhochschule besucht wird. Mit anderen Worten charakterisiert dies das Studienverhalten aller Studierenden. [1]

Während der Zeit des Selbststudiums ist der/die Student/in natürlich für jede Art von Unterstützung dankbar, seien es Skripten oder Mitstudenten. Es gibt aber auch Bereiche in denen es unmöglich ist etwas zu lernen bzw. zu erlernen, wenn man etwas nicht versteht. Auch in der Informatik fallen einige Teile des typischen Lehrstoffs darunter. Wie bereits zu Beginn dieses Kapitels angesprochen wurde, gehören Algorithmen dazu.

Wie kann man Studierenden nun während dieser Zeit des Selbststudiums unterstützen? Neben der Empfehlung von Fachlektüre oder dem Verfassen von Skripten, gibt es einen Trend zu Software, mit deren Hilfe gelernt und verstanden werden kann. Dies wird unter dem Fachbegriff Blended Learning als Teil des E-Learning zusammengefasst. Auf die genaue Bedeutung und die Idee, die dahinter steckt, möchte ich in Kapitel 5 „Blended Learning“ eingehen.

Für das Erlernen von Algorithmen drängt sich diese Idee gerade zu auf. Man stelle sich vor, es gäbe ein Programm, mit dessen Hilfe sich ein/e Student/in einen Algorithmus Schritt für Schritt erklären lassen kann. Es wäre ihm möglich sich die Zeit dafür zu nehmen, die er auch wirklich braucht.

Ziel dieser Diplomarbeit ist einen Algorithmenbaukasten im Sinne des Blended Learning zu programmieren, mit dessen Hilfe Studenten die wichtigsten Algorithmen der Informatik erarbeiten können.

4 Der Algorithmus

Wie backe ich einen Marmorkuchen?

Wie wasche ich mir die Haare?

Wie nehme ich meine Lieblingssendung auf Video auf?

Diese und viele andere möglichen Fragestellungen des täglichen Lebens haben etwas gemeinsam. Man kann aus ihnen Algorithmen ableiten. Im Prinzip kann nahezu jeder Vorgang auf einen solchen zurückgeführt werden.

Ganz allgemein kann man unter einem Algorithmus eine genau definierte Handlungs-vorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen verstehen.

Dabei gibt es aber ein paar Richtlinien, die zwingend eingehalten werden müssen. Nur wenn die folgenden Punkte zutreffen, spricht man von einem Algorithmus.

Es muss eine endliche Menge an Anweisungen definiert sein.

Dies bedeutet, dass ein Algorithmus nicht unendlich lang fortgesetzt werden kann. Dabei spielt es allerdings keine Rolle, ob dies ein Mensch während seiner Lebenszeit erleben kann. Es muss lediglich in endlicher Zeit beendet sein. Beispielsweise gibt es keinen Algorithmus, mit dessen Hilfe alle Primzahlen aufgezählt werden können.

Jede Anweisung muss ausführbar sein.

Darunter versteht man, dass eine Anweisung machbar sein muss. Unzulässig wäre es beispielsweise, eine Anweisungsfolge festzulegen, in der verlangt wird, ein Haus am Mars zu bauen. Dies ist natürlich unmöglich – zumindest zum jetzigen Zeitpunkt und mit den heutigen Mitteln und Möglichkeiten.

Jede Angabe genau genug sein.

Bei jeder Anweisung darf es zu keinen Missverständnissen oder Unklarheiten

kommen. Unzulässig wäre beispielsweise folgende Anweisung bei einem Haarwasch-Algorithmus:

„Fügen Sie nun Shampoo in die Haare.“

Dies ist nicht genau genug, weil nicht angegeben wird welche Menge an Shampoo in die Haare gegeben werden soll – ein paar Tropfen, eine Handvoll, der gesamte Flascheninhalt, noch mehr?

Weitere, klassische Beispiele für Algorithmen sind Kochrezepte, zumindest dann, wenn alle Angaben genau genug sind und es für alle Teilaufgaben, wie Braten, Rühren, etc., ebenfalls Algorithmen gibt. Auch Reparatur- und Bedienungsanleitungen oder Hilfen zum Ausfüllen von Formularen sind in der Regel Algorithmen.

Die folgende Tabelle beinhaltet Beispiele für Algorithmen.

Prozess	Algorithmus	Anweisungsbeispiel
Kuchenbacken	Rezept	Füge 300 g Mehl hinzu
Videogerät einstellen	Bedienungsanleitung	Drücke Taste „Record“
Ein Lied spielen	Notenblatt	
Bau eines Radios	Montageanleitung	Verbinde die Basis von Transistor T1 mit dem Kollektor von T5

4.1 Geschichte

Das Wort Algorithmus geht auf den arabischen Mathematiker und Gelehrten Muhammad ibn Musa al-Chwarizmi (* ca. 783, † ca. 850) zurück. 825 verfasste er das Werk „Hisab al-dschabr wa-l-muqabala“ - auf deutsch „Regeln zur Wiederherstellung und Reduktion“. Al-Chwarizmi bedeutet soviel wie "der aus Chwarizm stammende". Chwarizm, bzw. Chiwa, ist ein Ort im heutigen Usbekistan.

Die lateinische Übersetzung seines Werkes bezieht sich eben auf diesen Teil seines Namens. Das Werk beginnt mit „Dixit Algoritmi...“, was soviel bedeutet wie „Algorithmus sprach“, womit er gemeint war. Auch das Wort Algebra lässt sich auf ihn zurückführen, und zwar aus dem Titel seines Werkes. Es entstand aus dem Wort al-dschabr, was eigentlich Einrenkung bedeutet. Ursprünglich stand das Wort Algorithmus nur für die Regeln zur Arithmetik mit arabischen Ziffern. Heute steht es für alle geregelten Prozeduren, mit denen Probleme aller Art gelöst werden können.

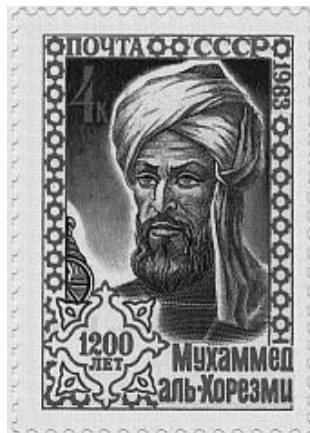


Abb. 4.1: al Chwarizmi

Im Bild 4.1 ist Muhammad ibn Musa al-Chwarizmi auf einer im Jahr 1983 in der damaligen Sowjetunion herausgegebenen Briefmarke zu sehen. [2]

4.2 Informatik

Vor allem aber sind Algorithmen eines der zentralen Themen der Informatik und Gegenstand einiger Spezialgebiete der Theoretischen Informatik, wie der Algorithmentheorie, der Komplexitätstheorie und der Berechenbarkeitstheorie. In Form von Computerprogrammen und elektronischen Schaltkreisen steuern sie Computer und andere Maschinen.

Allgemein kann man sagen, dass ein Algorithmus in der Informatik ein Verfahren zur Problemlösung, das sich für eine Implementierung als Computerprogramm eignet, beschreibt.

Für Algorithmen gibt es unterschiedliche formale Repräsentationen. Der Algorithmus wird beispielsweise als abstraktes Gegenstück zum konkret auf eine Maschine zugeschnittenen Programm verstanden. Das bedeutet, dass die Abstraktion im Weglassen der Details der realen Maschine erfolgt. Das Programm ist also eine konkrete Form des Algorithmus, angepasst an die Notwendigkeiten und Möglichkeiten der realen Maschine. Eine andere Ansicht sagt, dass Algorithmen gerade einmal die Maschinenprogramme von Turingmaschinen seien. Hier erfolgt die Abstraktion bereits in der Verwendung der Turingmaschine an sich, da diese nur eine ideale mathematische Maschine darstellt.

Zur Erklärung: Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing (Abb. 4.2) 1936 entwickeltes mathematisches Modell, um eine Klasse von berechenbaren Funktionen zu bilden. Sie wurde im Rahmen des von David Hilbert im Jahr 1900 formulierten Hilbertprogramms, speziell zur Lösung des so genannten Entscheidungsproblems, in der Schrift "On Computable Numbers, with an Application to the Entscheidungsproblem" vorgestellt. [3]



Abb. 4.2: Alan Turing

Algorithmen sind gerade deshalb von Bedeutung, da es mit ihrer Hilfe möglich ist große Einsparungen zu erzielen. Man denke dabei etwa an eine Anwendung, bei der beispielsweise mehrere Millionen Objekte verwaltet werden. Wenn man dies nun mit einem gut durchdachten Algorithmus durchführt, so kann man die Laufzeit des Programms stark beschleunigen. Dies ist gerade dann sehr wichtig, wenn man bedenkt, dass man etwa durch den Einsatz von neuer Hardware nicht automatisch den gleichen Effekt bewirken kann. Im Normalfall beschleunigt man nur durch diesen Schritt etwa im Bereich von Faktor 10 bis 100. In erster Linie ist es wichtig, die richtigen Methoden, also einen für den konkreten Anwendungsfall passend ausgewählten und korrekt implementieren Algorithmus, zu verwenden.

4.3 *Erster Computeralgorithmus*

Der erste für einen Computer gedachte Algorithmus wurde 1842 von Lady Augusta Ada Byron, Countess of Lovelace (1815-1852), der Tochter des berühmten englischen Poeten Lord Byron, in ihren Notizen zu Charles Babbages Analytical Engine, festgehalten.

Dieser Algorithmus beschrieb die Berechnung von Bernoulli-Zahlen. Sie gilt deshalb als die erste Programmiererin. Weil Charles Babbage seine Analytical Engine aufgrund von fehlenden Finanzierungen seitens der britischen Regierung zu seinen Lebzeiten nicht vollenden konnte, wurde Ada Lovelaces Algorithmus nie darauf implementiert. Nach ihr wurde aber eine Programmiersprache benannt – Ada. Gefördert und unterstützt durch das amerikanische Verteidigungsministerium wird sie vor allem in sicherheitskritischen Bereichen eingesetzt, etwa der Flugsicherung, in Waffensystemen oder Kernkraftwerken. [4]



Abb. 4.3: Lady Augusta Ada Byron, Countess of Lovelace (1815-1852)

4.4 Algorithmenanalyse

Die Erforschung und Analyse von Algorithmen ist Hauptaufgabe der Informatik, und wird meist theoretisch, das bedeutet ohne konkrete Umsetzung in eine Programmiersprache, durchgeführt. Sie ähnelt somit dem Vorgehen in anderen mathematischen Gebieten, in denen die Analyse eher auf die zugrunde liegenden Konzepte als auf konkrete Umsetzungen ausgerichtet ist. Algorithmen werden zur Analyse in eine stark formalisierte Form gebracht und mit den Mitteln der formalen Semantik, also mit mathematischen Methoden, untersucht.

Die Analyse unterteilt sich in verschiedene Teilgebiete. Beispielsweise wird das Verhalten von Algorithmen bezüglich Ressourcenbedarf wie Rechenzeit und Speicherbedarf in der Komplexitätstheorie behandelt, die Ergebnisse werden als asymptotische Laufzeiten angegeben. Der Ressourcenbedarf wird dabei in Abhängigkeit von der Länge der Eingabe ermittelt, das heißt die angegebene Komplexität hängt beispielsweise davon ab, wie groß die Zahlen sind, deren größter gemeinsamer Teiler gesucht wird, oder wie viele Elemente sortiert werden müssen etc.

Das Verhalten bezüglich der Terminierung, das heißt ob der Algorithmus überhaupt jemals erfolgreich beendet werden kann, behandelt die Berechenbarkeitstheorie. Wie es der Name schon vermuten lässt, befasst sich diese mit dem Begriff der Berechenbarkeit, insbesondere welche Probleme mit Hilfe einer Maschine, genauer gesagt: eines mathematischen Modells einer Maschine, etwa der bereits erwähnten Turingmaschine, lösbar sind.

5 Blended Learning

“Blended learning is learning which uses a combination of face-to-face (i.e. taught) and online media. This form of learning is increasingly used at universities, and the presence of both types of learning distinguish blended from wholly online and wholly classroom methods of learning. “ [MIT04] auf Seite 71.

Kurz könnte man dies auch gemäß [RIZ03] im Text „Präsenzunterricht, Fernunterricht“ als „Mix von Fern- und Präsenzunterricht.“ zusammenfassen.

Ganz allgemein ausgedrückt, wird unter "Blended Learning" ein Unterrichtsaufbau, bei dem sich Präsenz- und Online-Phasen abwechseln, verstanden. Ein Teil des Unterrichts findet als übliche Präsenzveranstaltung statt, eine/ein Lehrbeauftragte/r unterrichtet live und vor Ort eine Gruppe an zu Unterrichtenden, der andere Teil wird online gestaltet.

Dieses Modell ist auch für die Planung längerer Lernprozesse geeignet. Mögliche Anwendungsgebiete sind universitäre Lehrveranstaltungen oder verschiedene projektorientierte Unterrichts- und Trainingsvorhaben.

Die Präsenzphasen haben vorrangig folgende Funktionen:

- Koordinierung der Lernprozesse
- Entwicklung des sozialen Systems der Lerngruppe: Kennen lernen von Lehrenden und Lernenden, Klärung von Erwartungen, Interessen, Arbeits- und Kommunikationsregeln
- Abklärung von Organisation, Thema, Zielen und Arbeitsablauf
- Präsentation und Diskussion von Arbeitsergebnissen
- Reflexion und Evaluation von Lern-/Arbeitsprozessen

Die Online-Phasen dienen dagegen vor allem folgenden Funktionen

- (interaktive) Bearbeitung der vereinbarten Aufgaben
- Onlinegestütztes Selbststudium (z.B. Internetrecherche, Lösung interaktiver Aufgaben)
- Kollaboratives Arbeiten in virtuellen Gruppen
- Moderation des Arbeitsprozesses über (themenbezogene) Foren,
- Synchrone Kommunikation zu spezifischen Fragestellungen im Lernprozess

"Blended learning" erfordert einen erhöhten Planungsaufwand in der Vorbereitung und in der Evaluierung der einzelnen Arbeitsschritte. Darunter fallen zum Beispiel Produktion und Implementierung des Content, Auswahl der geeigneten "tools" bzw. Einrichtung der entsprechenden "workspaces" auf einer Lernplattform.

Der Einsatz Neuer Medien stellt im Unterricht eine zusätzliche Herausforderung an die Planungsarbeit der Lehrbeauftragten dar. Die Komplexität der Lernorganisation erhöht sich, damit verbunden wachsen auch die Aufgaben für die Steuerung des Lernprozesses.

5.1 Begriffserklärung

Allgemein hat sich der Begriff „Blended Learning“ erst im Laufe des Jahres 2001 im deutschsprachigen Raum etabliert und bezeichnet mittlerweile einen der vorherrschenden Trends für E-Learning Lösungen. Übersetzt man den englischen Begriff „Blended“, heißt dies soviel wie „vermengt, vermischt, ineinander übergehend“. Ein deutschsprachiges Pendant hat sich nicht entwickelt. Dies mag in erster Linie damit zusammen hängen, dass eine exakte deutsche Übersetzung etwas holprig klingen würde. Was würde man wohl unter dem Begriff „Vermischtes Lernen“ verstehen?

Allerdings hat sich im deutschen Sprachgebrauch der Begriff des „Hybriden Lernens“ verbreitet, der bereits länger als der Trendausdruck „Blended Learning“ existiert und die gleiche Bedeutung hat.

5.2 Blended Learning im universitären Umfeld

Laut der vom US-Elektronik-Konzern IBM in Auftrag gegebenen Studie "The 2003 e-learning readiness ranking" befindet sich Österreich unter den 60 größten Volkswirtschaften auf Platz 15 in Bezug auf Umsetzung von E-Learning-Strategien und – Lösungen. Bewertet wurden dabei alle Bereiche in denen E-Learning angeboten werden kann. Dies beginnt bei der Ausbildung und geht über Staat und Gesellschaft bis hin zum betrieblichen E-Learning. Dies ist immerhin ein Platz im oberen Mittelfeld.
[5]

Lehr- und Lernszenarien, die unter Einsatz von Lernplattformen und anderen Werkzeugen entwickelt werden, können sehr unterschiedlich aussehen: Die Bandbreite reicht von Content-Darbietungen, aufwändigeren Online-Kursen zum selbst-gesteuerten Lernen über kommunikative Szenarien wie virtuelle Gruppenarbeit bis hin zu virtuellen Laboratorien oder Planspielen, in denen individuelles oder team-orientiertes Lernen stattfindet.

Bei der Entwicklung von Blended-Learning-Szenarien im universitären Umfeld stehen die Prozesse des Lehrens und Lernens einer spezifischen Lehrveranstaltung in ihrem Semesterverlauf im Mittelpunkt. In diese Prozesse können vorhandene Materialien oder Medienprodukte eingebunden werden. [6]

6 Methodisches Vorgehen

6.1 Problembenennung

Viele der Standardalgorithmen werden von Studierenden während des Studiums erlernt. Oft tritt allerdings das Problem auf, dass es schwer scheint, diese zu erlernen bzw. zu verstehen, da man zum Selbststudium nur die einzelnen Anweisungen des Algorithmus oder die Funktionen in der jeweiligen Programmiersprache zur Verfügung hat. Es kann sehr umständlich und vor allem extrem aufwändig werden, wenn man versucht diese mit Papier und Bleistift „durchzuspielen“.

Ein anderer Aspekt entsteht aus Sicht des Vortragenden. Da jeder Mensch respektive Student/in unterschiedlich ist, wird auch nicht jeder gleich auf Anhieb einen neuen Algorithmus verstehen. Gerade in Vorlesungen ist es schon alleine aufgrund des Zeitdrucks unmöglich solange zu warten bzw. jedem/jeder Studenten/in während der Vorlesungszeit jeden Algorithmuschritt so lange zu erklären, bis er oder sie es versteht. Man darf schließlich nicht vergessen, dass der vorgegebene Semesterstoff innerhalb der vorgeschriebenen Zeit durchgebracht werden muss.

6.2 Zielsetzung

Leichter zu erlernen wäre ein Algorithmus, wenn man ein Programm hätte, mit dessen Hilfe man sich diesen Schritt für Schritt ansehen, erlernen und natürlich auch verstehen kann. Beispielsweise könnte man sich so durch den Bubblesort-Algorithmus eine beliebige Datenmenge anschaulich sortieren lassen und somit die Funktionsweise verstehen.

Daher soll ein Baukasten entwickelt werden, mit dessen Hilfe zukünftig Studenten die wichtigsten Algorithmen erlernen können. Zusätzlich wird eine Komplexitätsanalyse dieser Algorithmen herangezogen und die Ergebnisse daraus möglichst anschaulich dargestellt.

6.3 *Algorithmenauswahl*

Die wichtigsten Algorithmen der Informatik, die auch üblicherweise während eines Informatikstudiums unterrichtet bzw. erlernt werden, sollten in dem Algorithmenbaukasten enthalten sein.

Eingeteilt in vier Hauptkategorien wurden folgende Algorithmen ausgewählt:

- **Sortieren**
Bubblesort, Insertionsort, Quicksort, Heapsort, Mergesort, Shellsort und Bucketsort
- **Suchen**
lineare Suche, binäre Suche und interpolierte Suche
- **Hashing**
direkte Verkettung, lineares Sondieren und doppeltes Hashing
- **Binärbaum.**
suchen, einfügen und löschen innerhalb eines Binärbaums
Traversieren eines Binärbaums – Preorder, Inorder und Postorder

Im Kapitel 7 „Der Algorithmenbaukasten“ werden die einzelnen Algorithmen genauer vorgestellt.

6.4 *Programmierung*

Der Algorithmenbaukasten wurde mit Javascript, unter Verwendung der Auszeichnungssprache HTML sowie CSS, programmiert. Die im folgenden Abschnitt dargestellten Quellcodesegmente sollen nicht dazu dienen HTML, Javascript oder CSS verständlich zu machen oder gar als Lernhilfe zu dienen. Es wird davon ausgegangen, dass der Leser dieser Diplomarbeit bereits im Umgang damit vertraut ist. Aus diesem Grund wird gezeigt wie der Algorithmenbaukasten aufgebaut ist, nicht aber jeder Teil des Quelltextes bis ins kleinste Detail erklärt.

6.4.1 HTML

Für jede der vier Hauptkategorien wurde dabei eine eigene HTML-Seite entwickelt auf der die User die für den jeweiligen Algorithmus vorgesehenen Aktionen durchführen können. In Abbildung 6.1 ist eine Übersichtsseite anhand der Kategorie Sortieren zu sehen.

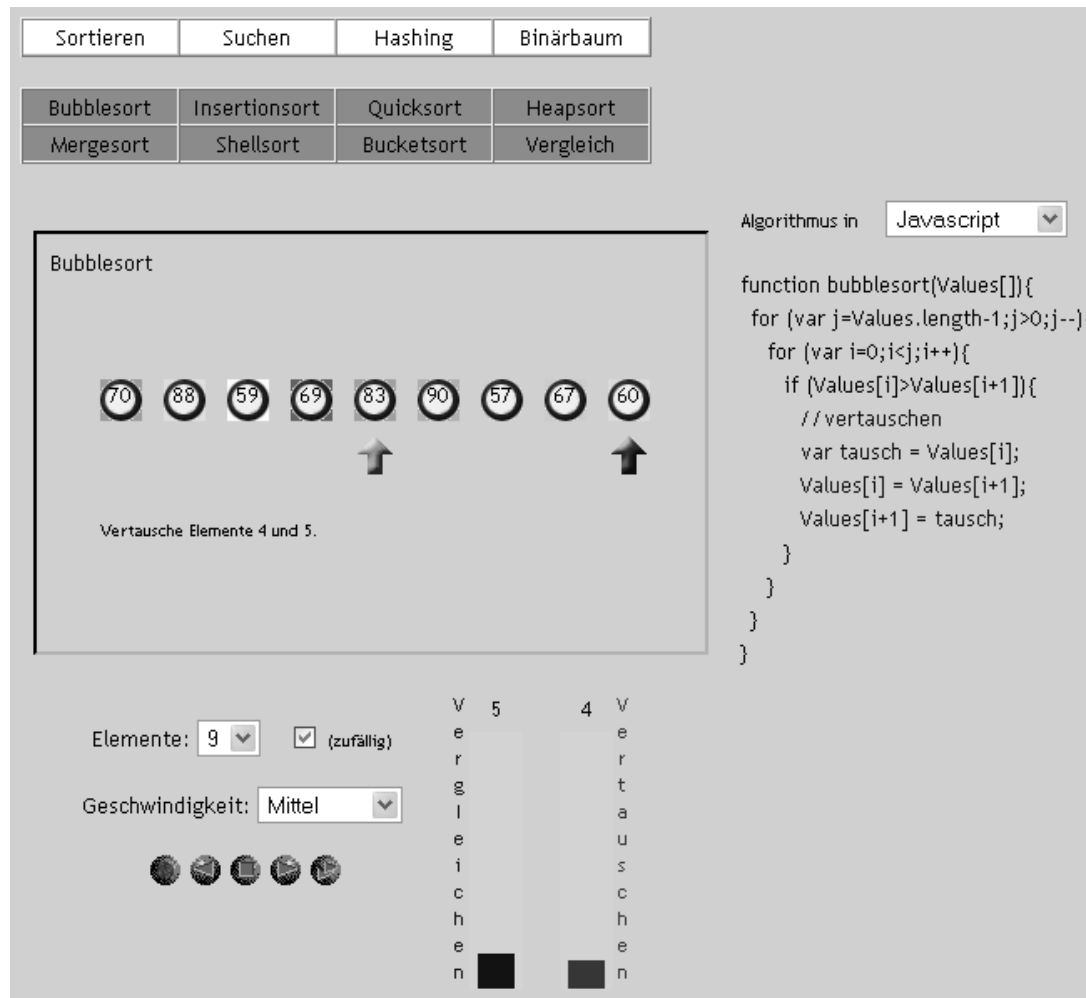


Abb. 6.1: Der Algorithmenbaukasten

Ganz oben befindet sich die Navigation, mit dessen Hilfe alle Seiten des Algorithmenbaukastens angesteuert werden können.

```
<table border="1" cellspacing="0" cellpadding="1">
  <tr bgcolor="#0066FF">
    <td align="center" width="90"><a href="sort.html">Sortieren</a></td>
    <td align="center" width="90"><a href="such.html">Suchen</a></td>
    <td align="center" width="90"><a href="hash.html">Hashing</a></td>
    <td align="center" width="90"><a href="baum.html">Binärbaum</a></td>
  </tr>
</table>
```


Dies ist die Standardtabelle, die auf jeder Seite aufscheint. Durch Mouseclick auf eine der vier Verlinkungen gelangt man zu dem entsprechenden Hauptmenü.

Je nachdem welche Kategorie man aufgerufen hat, erscheint direkt darunter eine Tabelle mit den Verlinkungen zu den Algorithmen, die sich in diesem Bereich befinden bzw. dargestellt werden.

```
<table border="1" cellspacing="0" cellpadding="1">
  <tr>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(1);">Bubblesort</a>
    </td>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(3);">Insertionsort</a>
    </td>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(2);">Quicksort</a>
    </td>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(7);">Heapsort</a>
    </td>
  </tr>
  <tr>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(4);">Mergesort</a>
    </td>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(5);">Shellsort</a>
    </td>
    <td width="90" align="center" bgcolor="#3399FF">
      <a href="javascript:change_iframe(6);">Bucketssort</a>
    </td>
```

```
<td width="90" align="center" bgcolor="#3399FF">
  <a href="sort_vgl.html">Vergleich</a>
</td>
</tr>
</table>
```

Diese Tabelle beispielsweise ermöglicht den Zugang zu den Algorithmen in der Kategorie Sortieren. Hier wird allerdings im Unterschied zur Tabelle mit den Hauptkategorien eine Javascript-Funktion aufgerufen. Auf die genaue Funktionsweise möchte ich im Abschnitt „6.4.2 Javascript“ eingehen. Warum allerdings kein „normaler“ Link in der Form dateiname.html verwendet wird, ist für den nächsten Bereich sehr wichtig.

Direkt unter der Navigation befindet sich das Darstellungsfenster in dem der jeweilige Algorithmus übersichtlich dargestellt wird. Hierfür wird ein iFrame, also ein eingebetteter Frame verwendet. Der folgende Quelltext zeigt dies am Beispiel des Bubblesort.

```
<iframe src="anzeige_bubble.html" width="400" height="250" id="anzeige" name="anzeige"></iframe>
```

Die bereits erwähnte Javascriptfunktion *change_iframe()* wechselt die Quelle aus. Die englische Bezeichnung dafür ist "source" - daher auch das Kürzel src. Je nachdem welcher Algorithmus verlangt wird, wird die entsprechende HTML-Datei im eingebetteten Frame ausgegeben.

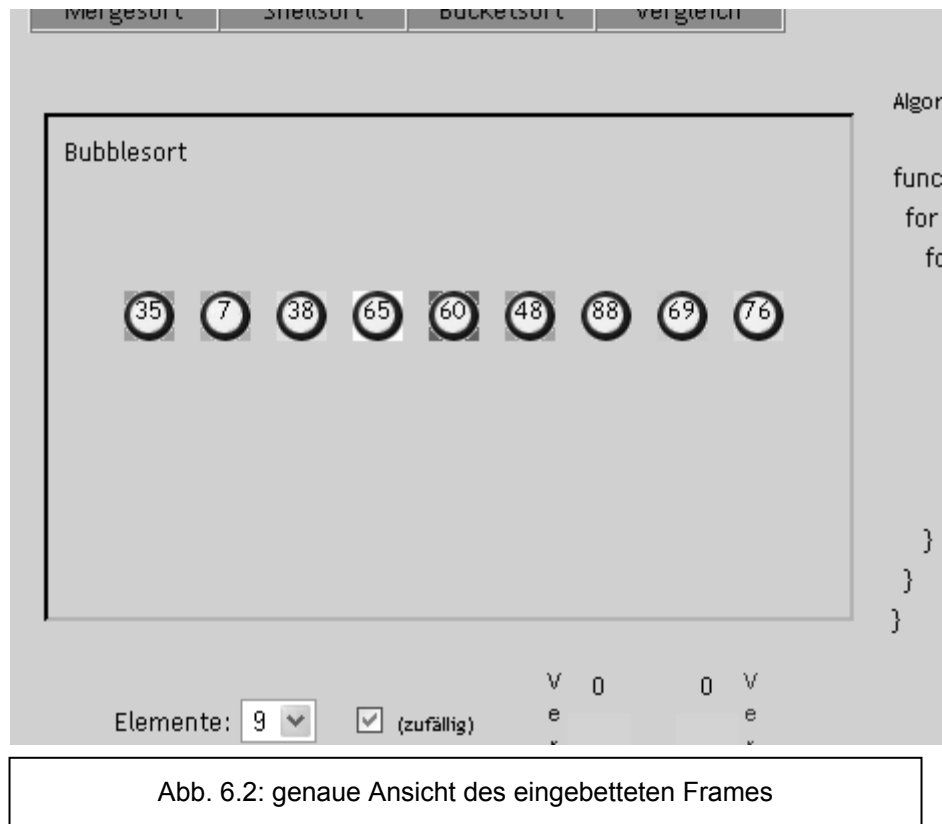


Abb. 6.2: genaue Ansicht des eingebetteten Frames

Grund dafür sind die verwendeten Javascript-Files. Da jeder Algorithmus verständlicherweise anders arbeitet bzw. aufgebaut ist, müssen auch für jeden unterschiedliche Anweisungen implementiert werden.

```
<html>
<head>
<title>Anzeige</title>
<script src="js/bubble.js" type="text/javascript"></script>
</head>
```

Bei jeder Algorithmus-Seite wird ein eigenes Javascript-File inkludiert. Beispielsweise dient bubble.js dazu, dass die angegebene Zahlenreihe auch wirklich mit Hilfe des Bubblesorts sortiert wird. Details zu dem Javascript-File werden im Abschnitt „6.4.2.2 Algorithmenseite“ beschrieben.

Unterhalb dieses eingebetteten Frames befinden sich je nach Algorithmus bzw. Kategorie unterschiedliche Darstellungsoptionen für den Anwender.



Abb. 6.3: Darstellungsoptionen im Bereich Sortieren

In Abbildung 6.3 ist dies am Beispiel der Kategorie Sortieren dargestellt. Neben der Auswahl der darzustellenden Elemente sowie der Geschwindigkeit beim Abspielen eines Algorithmus hat man eben die Möglichkeit sich Schritt für Schritt durch den Ablauf zu klicken oder sich dies als eine Animation anzeigen zu lassen. Des Weiteren wird mitgezählt und übersichtlich anhand eines Säulendiagramms dargestellt, wie oft zwei Elemente verglichen bzw. vertauscht werden müssen. Gesteuert wird jedes dieser Elemente allerdings wieder über Javascript.

Weiters wird auf der rechten Seite der Quelltext des jeweiligen Algorithmus angezeigt. Wie in Abbildung 6.4 dargestellt, stehen dabei die Programmiersprachen Java, Javascript und C++ sowie Pseudocode zur Auswahl .

The image shows a web interface with four panels, each displaying the Bubble Sort algorithm in a different language. Each panel has a dropdown menu labeled 'Algorithmus in' and a code block.

- Javascript:**

```
function bubblesort(Values[]){
  for (var j=Values.length-1;j>0;j--){
    for (var i=0;i<j;i++){
      if (Values[i]>Values[i+1]){
        //vertauschen
        var tausch = Values[i];
        Values[i] = Values[i+1];
        Values[i+1] = tausch;
      }
    }
  }
}
```
- Java:**

```
public void bubblesort(int Values []){
  for (int j=Values.length-1;j>0;j--){
    for (int i=0;i<j;i++){
      if (Values[i]>Values[i+1]){
        //vertauschen
        int tausch = Values[i];
        Values[i] = Values[i+1];
        Values[i+1] = tausch;
      }
    }
  }
}
```
- C++:**

```
void class::bubblesort(int Values []){
  for (int j=Values.length-1;j>0;j--){
    for (int i=0;i<j;i++){
      if (Values[i]>Values[i+1]){
        //vertauschen
        int tausch = Values[i];
        Values[i] = Values[i+1];
        Values[i+1] = tausch;
      }
    }
  }
}
```
- Pseudocode:**

```
PROCEDURE bubblesort(array Values) BEGIN
  FOR j:=Values.length-1 TO 1)
    FOR i:=0 TO j)
      Wenn Values[i] > Values[i+1] ist
        Vertausche
        Values[i] mit Values[i+1]

    FOR END
  FOR END
PROCEDURE END
```

Abb. 6.4: Auswahl der Algorithmenanzeige am Beispiel Bubblesort

In den Bereichen Sortieren und Suchen kann man sich außerdem mit der Leistung der einzelnen Algorithmen etwas genauer auseinandersetzen. Man hat im Sortierbereich, wie in Abbildung 6.5 dargestellt, die Möglichkeit sich eine zufällig generierte

Datenmenge von allen Algorithmen sortieren zu lassen. Hier wird deutlich aufgezeigt, welches Leistungsvermögen die einzelnen Algorithmen haben.

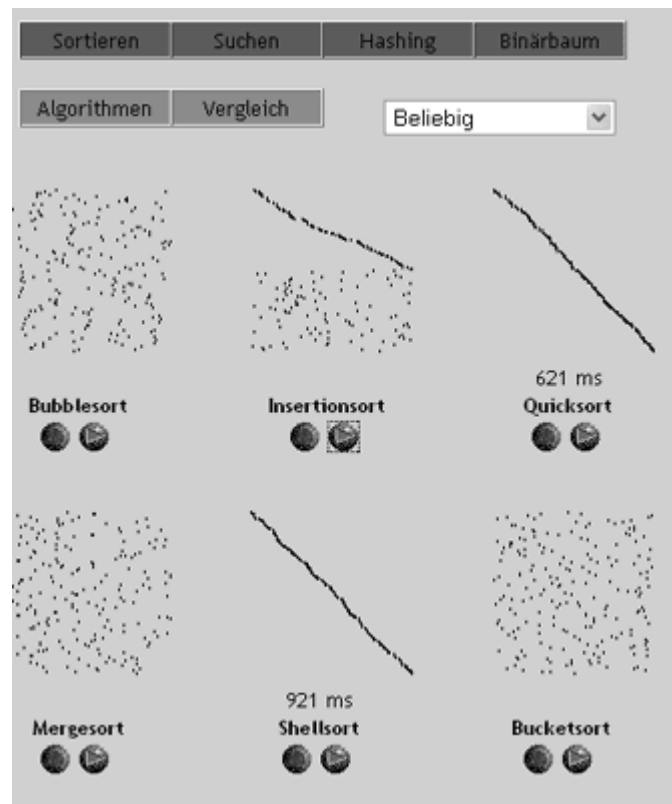


Abb. 6.5: Sortiervergleich im Algorithmenbaukasten

Im Suchbereich gibt es dementsprechend einen ähnlichen Vergleich (Abb. 6.6). Hier wird vor Augen geführt wie schnell die einzelnen Suchalgorithmen eine bestimmte Zahl aus einer ebenfalls zufällig definierten Datenmenge finden können.

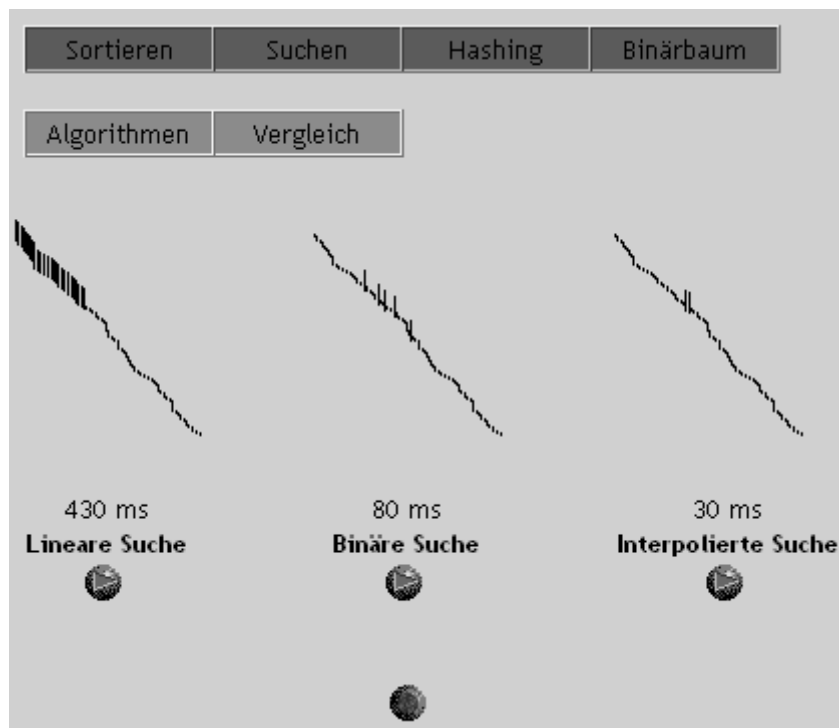


Abb. 6.6: Suchvergleich im Algorithmenbaukasten

6.4.2 Javascript

Die verwendeten und im vorhergehenden Kapitel vorgestellten HTML-Seiten dienen lediglich der Darstellung. Dadurch ist aber noch keine Interaktion des Users mit dem Programm möglich. Deswegen wurde die Hauptprogrammierarbeit mittels Javascript durchgeführt. Erst dadurch ist es dem Anwender beispielsweise möglich eigene Zahlen sortieren zu lassen oder den Algorithmus leicht verändern zu können.

6.4.2.1 Hauptseite

In jeder Seite der vier Hauptkategorien werden zwei Javascript-Files inkludiert. Die Datei „alg.js“ enthält verschiedene Funktionen, die jederzeit und von allen Algorithmen bzw. deren Darstellungen gebraucht werden.

```
function fertig_text(text,text2){
    anzeige.document.getElementById("status").firstChild.data = text;
    anzeige.document.getElementById("status2").firstChild.data = text2;
    anzeige.document.getElementById("status").style.visibility = "visible";
    anzeige.document.getElementById("status2").style.visibility = "visible";
    anzeige.modus="stop";
    keycheck=true;
    fertig=true;
    for (var i=0;i<18;i++){
        document.getElementById("alg"+i).className = "normal";
    }
}
```

Die Funktion „fertig_text()“ dient dazu einen Algorithmus abzuschließen. Beispielsweise wurde ein Array fertig sortiert. Daraus ergibt sich, dass der Anwender keine Möglichkeit mehr haben soll, noch irgendetwas ändern zu können. Er kann lediglich den Algorithmus wieder zurücksetzen und mit neuen Ausgangsdaten starten. Mit Hilfe

der beiden übergebenen Variablen text und text2 werden Informationen über den aktuellen Stand des Algorithmus ausgegeben, beispielsweise, dass dieser abgeschlossen wurde.

```
function hide_this(id){
    document.getElementById(id).style.visibility = "hidden";
}

function set_div(wer,links,oben){
    anzeige.document.getElementById(wer).style.left = links;
    anzeige.document.getElementById(wer).style.top = oben;
    anzeige.document.getElementById(wer).style.visibility = "visible";
}
```

Mit Hilfe der Funktionen hide_this() und set_div() werden DIV-Elemente ein- bzw. ausgeblendet. Damit die sichtbaren Elemente auch die Möglichkeit haben sich zu bewegen, beispielsweise wenn beim Sortieren zwei Elemente ausgetauscht werden sollen, wird dafür eine weitere Funktion gebraucht – slide_thedivs().

```
var sliders = new Array();
var sliderszielleft = new Array();
var sliderszieltop = new Array();
var anzsliders;
function slide_thedivs(){
    var fert=true;
    for (var i=0;i<anzsliders;i++){
        if (parseInt(anzeige.document.getElementById(sliders[i]).style.left)<parseInt(sliderszielleft[i]))
        {
            anzeige.document.getElementById(sliders[i]).style.left =
            parseInt(anzeige.document.getElementById(sliders[i]).style.left)+ 1 +"px";
        }
        if (parseInt(anzeige.document.getElementById(sliders[i]).style.left)>parseInt(sliderszielleft[i]))
        {
            anzeige.document.getElementById(sliders[i]).style.left =
            parseInt(anzeige.document.getElementById(sliders[i]).style.left)- 1 +"px";
        }
    }
}
```

```

    }
    if (parseInt(anzeige.document.getElementById(sliders[i]).style.top)<parseInt(sliderszieltop[i]))
    {
        anzeige.document.getElementById(sliders[i]).style.top =
            parseInt(anzeige.document.getElementById(sliders[i]).style.top)+ 1 +"px";
    }
    if (parseInt(anzeige.document.getElementById(sliders[i]).style.top)>parseInt(sliderszieltop[i]))
    {
        anzeige.document.getElementById(sliders[i]).style.top =
            parseInt(anzeige.document.getElementById(sliders[i]).style.top)- 1 +"px";
    }
}
for (var i=0;i<anzsliders;i++){
    if (parseInt(anzeige.document.getElementById(sliders[i]).style.left)!=parseInt(sliderszielleft[i])
    || parseInt(anzeige.document.getElementById(sliders[i]).style.top)!=parseInt(sliderszieltop[i]))
    {
        fert = false;
        break;
    }
}
if (fert == true) {
    keycheck=true;
}
else{
    setTimeout("slide_thedivs()",10);
}
}

```

Hierbei wird die ID des zu bewegendes Elements sowie die gewünschte Position in die entsprechenden globalen Arrays gespeichert. Aus diesen liest die Funktion die Werte aus und setzt das DIV-Element einen Pixel näher an das Ziel. Wurde dies durchgeführt ruft sich die Funktion mittels Timeout selbst auf. Das geschieht solange bis die Elemente am Ziel angekommen sind.

Die zweite Javascript-Datei, die in den Hauptseiten inkludiert wird, betrifft die Navigation innerhalb einer Kategorie. Darin enthalten ist die Funktion `change_iframe()`. Damit ist es möglich zwischen den einzelnen Algorithmen zu wechseln. Als Beispiel ist im Folgenden diese für den Bereich Sortieren angezeigt.

```
function change_iframe(w){
  switch(w){
    case 1:
      document.getElementById("anzeige").setAttribute("src","anzeige_bubble.html","false");
      break;
    case 2:
      document.getElementById("anzeige").setAttribute("src","anzeige_quick.html","false");
      break;
    case 3:
      document.getElementById("anzeige").setAttribute("src","anzeige_insertion.html","false");
      break;
    case 4:
      document.getElementById("anzeige").setAttribute("src","anzeige_merge.html","false");
      break;
    case 5:
      document.getElementById("anzeige").setAttribute("src","anzeige_shell.html","false");
      break;
    case 6:
      document.getElementById("anzeige").setAttribute("src","anzeige_bucket.html","false");
      break;
    case 7:
      document.getElementById("anzeige").setAttribute("src","anzeige_heap.html","false");
      break;
  }
}
```

6.4.2.2 Algorithmenseite

Wie bereits im Abschnitt 6.4.1 „HTML“ erwähnt, wird jeder Algorithmus in einer eigenen HTML-Seite dargestellt, die in einem eingebetteten Frame ausgegeben wird. In jeder dieser HTML-Seiten wird wiederum ein eigenes Javascript-File inkludiert.

Hier sind alle Funktionen untergebracht, die für jeden Algorithmus individuell angepasst werden müssen. Die wichtigste ist natürlich diejenige, die den Algorithmus an sich ausführt. Dabei wird die selbe Timeout-Technik angewendet, die auch bereits im Abschnitt 6.4.2.1 „Hauptseite“ angesprochen wurde. Die Besonderheit hierbei ist aber, dass nach jeder Aktion, die durchgeführt wird, also beispielsweise dem Austausch von zwei Elementen, der komplette Algorithmus zurückgesetzt wird. Im nächsten Durchgang, das heißt wenn das Timeout vorüber ist und die Funktion erneut aufgerufen wurde, wird diese Aktion übersprungen. Dadurch sieht es für den Anwender aus, als ob der Algorithmus normal weiter laufen würde.

Warum dieses recht verwirrend anmutende Konzept durchgeführt wird lässt sich relativ einfach mit der Tatsache erklären, dass ohne dieser Timeout-Technik keine Animationen mittels Javascript möglich sind. Das folgende Codesegment dient als Beispiel dafür.

```
counttest = vglcount();
if (counttest == true){
    vglcounter++;
    if (document.getElementById("pointer1").style.visibility == "hidden"){
        parent.set_div("pointer1",eval(38*(parseInt(j)+1))+ "px","120px");
        parent.keycheck=true;
    }
    else {
        parent.sliders[0]="pointer1";
        parent.sliderszielleft[0]=38*(parseInt(j)+1);
    }
}
```

```
parent.sliderszieltop[0]=120;
parent.anzsliders = 1;
parent.slide_thedivs();
}
if (mod=="all"&&modus!="stop") parent.ID=parent.setmytimeout(1);
parent.setz_status("Setze rechten Pointer auf Position "+j+".", "");
parent.document.getElementById("alg1").className = "rot";
return;
}
if (code_ik == "j"){myik=j;}
counter++;
```

Zunächst wird mit Hilfe der Funktion `vglcount()` getestet, ob der Algorithmus an der aktuellen Position angelangt ist. Trifft dies zu bedeutet es, dass die aktuelle Animation noch nicht durchgeführt wurde, und somit wird *true* von der Funktion zurückgegeben. Damit ist der Eintritt in die IF-Abfrage ermöglicht und die jeweiligen Aktionen können durchgeführt werden. In diesem Beispiel soll ein DIV-Element, das einen Pointer repräsentiert, bewegt werden. Dazu werden die ID des Elements und die Zielkoordinaten in globale Variablen gespeichert und die bereits aus dem Abschnitt 6.4.2.1 „Hauptseite“ bekannte Funktion `slide_thedivs()` aufgerufen. Anschließend wird der entsprechende Text am Bildschirm ausgegeben und die aktuelle Stelle des Algorithmus mittels CSS markiert. Das Besondere ist, dass die komplette Funktion durch den Befehl `return;` beendet wird und nach der Animation der DIV-Elemente wieder aufgerufen wird, um zur nächsten Anweisung zu gelangen.

Standardmäßig werden den einzelnen Elementen, die zu sortieren oder zu suchen sind, zufällige Zahlen zwischen 0 und 99 zugewiesen. Dem/Der Anwender/in ist es aber auch möglich, diese durch eigene Zahlen zu ersetzen. Wie in Abbildung 6.7 zu sehen, öffnet sich durch den Klick auf eines der DIV-Elemente ein Fenster und eine neue Zahl kann eingegeben werden.

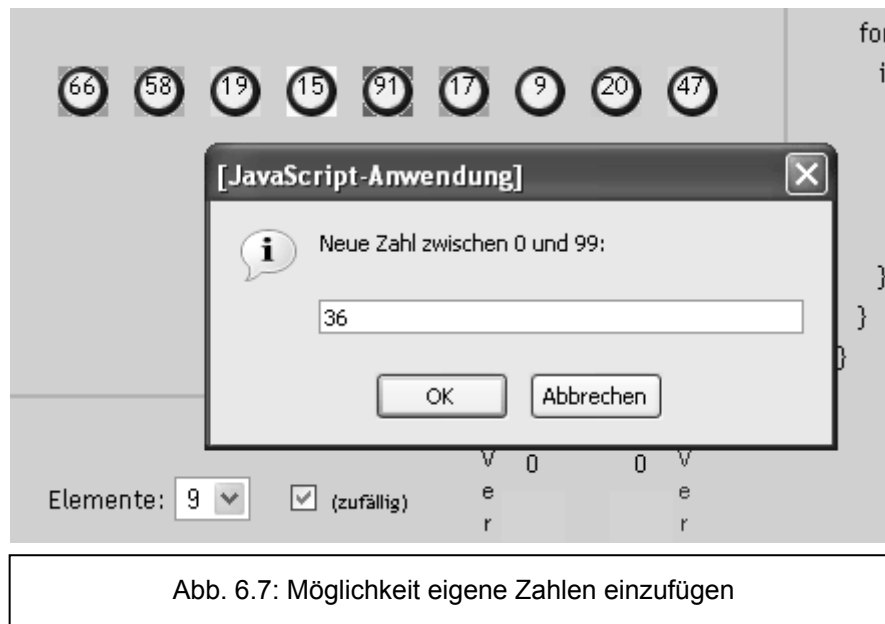


Abb. 6.7: Möglichkeit eigene Zahlen einzufügen

Geregelt wird dies über die Funktion *change_nr()*.

```
function change_nr(id){
    var Check = prompt("Neue Zahl zwischen 0 und 99:","");
    if (isNaN(Check)==false && Check!=null && Check<100 && Check >=0){
        anzeige.document.getElementById(id).firstChild.data = Check;
        arr_ur[id.slice(4,id.length)]=Check;
        arr_sor[id.slice(4,id.length)]=Check;
    }
}
```

Die durch den Anwender eingegebenen Zeichen werden auf ihre Gültigkeit hin überprüft. Das bedeutet, dass nur Zahlen zwischen 0 und 99 akzeptiert werden. Ansonsten wäre ein Sortieren auch nicht sinnvoll, da nicht vergleichbare Elemente vorhanden sein könnten. Ist eine gültige Zahl eingegeben worden, wird diese im DIV-Element ausgegeben und in den globalen Ausgangsarrays gespeichert.

6.4.2.3 Vergleichsseite

Auch bei den beiden HTML-Seiten, die sich mit den Leistungsunterschieden der Algorithmen beschäftigen, wird je eine Javascriptdatei inkludiert. Am wichtigsten sind natürlich die Funktionen, welche die Algorithmen ausführen und damit verbunden die Punkte sortieren bzw. suchen.

Beim Laden der Datei werden für jeden Algorithmus 200 DIV-Elemente erzeugt, die jeweils einen Punkt enthalten. Positioniert werden diese anhand von zuvor ermittelten Werte.

```
document.write("  
  <div id='B'+i+'"  
    style='position:absolute;left:"+eval(0+Math.round(eval(Values_b[i]/2)))+ "px;  
    top:"+eval(100+Math.round(eval(i/2)))+ "px">  
  .  
</div>  
");
```

Einen kleinen Unterschied gibt es dabei selbstverständlich zwischen den Vergleichsseiten von Sortieren und Suchen.

```
Values_b[i]=Math.round(Math.random()*200);
```

Während es beim Sortieren reicht, wie im obigen Beispiel für jedes Array-Element einen beliebigen und damit zufälligen Wert abzuspeichern, muss das komplette Array für die Suchalgorithmen bereits sortiert sein, damit die Algorithmen auch funktionieren. Dies wird schon beim Erzeugen der Zufallszahlen vorgenommen, indem neue Zahlen nicht kleiner sein können als die bisherigen.

```
zahl = zahl+Math.round(Math.random()*2);  
Values_b[nextelement]=zahl;
```

Während des Sortiervorgangs werden die DIV-Elemente entsprechend dem Algorithmus an die entsprechende Position verschoben. Im folgenden Absatz ist dies am Beispiel des Bubblesorts dargestellt.

```
var tausch = Values_b[i];  
Values_b[i] = Values_b[i+1];  
Values_b[i+1] = tausch;  
    var tauscha = document.getElementById("B"+i);  
    var tauschb = document.getElementById("B"+eval(i+1));  
    tauscha.style.top = eval(100+Math.round(eval((i+1)/2)))+ "px";  
    tauschb.style.top = eval(100+Math.round(eval(i/2)))+ "px";  
    tauscha.id = "B"+eval(i+1);  
    tauschb.id = "B"+i;
```

Während der ersten drei Zeilen werden zwei Elemente, wie im Bubblesort gewohnt, ausgetauscht. Im Anschluss werden dann die entsprechenden DIV-Elemente ausgetauscht.

Der/Die Anwender/in kann sich im Bereich „Sortieren“ die Ausgangsposition der einzelnen DIV-Elemente aber nicht nur zufällig ausgeben lassen. Ebenfalls möglich ist auch eine bereits sortierte Liste, sowohl auf- als auch absteigend, wie aus Abbildung 6.8 ersichtlich ist.



Abb. 6.8: Auswahlmöglichkeiten
beim Sortiervergleich

Nach Ändern dieses DropDown-Menüs wird die Funktion *status_select()* aufgerufen.

```
function status_select(id){  
    sortieren = document.getElementById(id).selectedIndex;  
}
```

Diese speichert die aktuell gewünschte Option in einer globalen Variable. Nach deren Wert werden dann die Ausgangspositionen der DIV-Elemente festgelegt.

6.4.3 CSS

In jeder HTML-Datei wird zusätzlich eine Stylesheet-Datei eingebunden – roh2.css. Bei CSS handelt es sich um eine Sprache zur Definition von Formateigenschaften einzelner HTML-Elemente. Damit wird in diesem Programm dafür gesorgt, dass alle Seiten gleich aufgebaut sind. Sei es durch die gleiche Hintergrundfarbe,

```
body {background-color:#ccccff}
```

oder dass immer die gleiche Schriftart sowie der gleiche Schriftgrad verwendet wird.

```
body,table,td,tr,div,p,pre,h1,h2,h3,h4,ul {font-family: "Trebuchet MS", Arial,Helvetica, sans-serif;}  
body,td,div,p,pre,ul {font-size: 13px;}
```

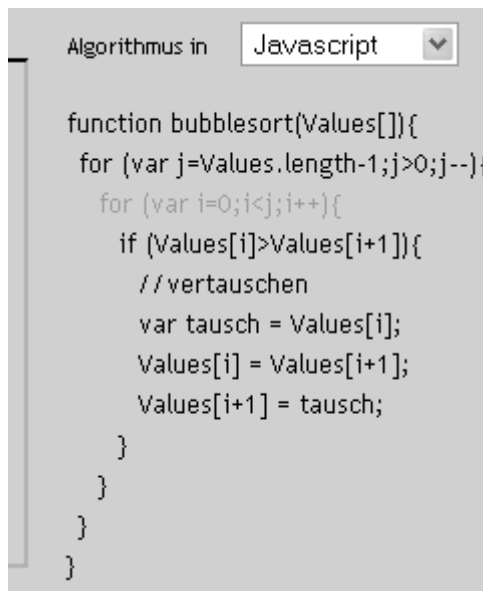
Der wichtigste Einsatz findet sich allerdings in der Animation eines Algorithmus wieder. Mit Hilfe des Javascriptbefehls

```
parent.document.getElementById("alg0").className = "blau";
```

gibt es die Möglichkeit die CSS-Klasse für ein HTML-Element auszutauschen. Hier wird beispielsweise dem Element mit der ID „alg0“ die Klasse blau zugewiesen, die bewirkt, dass die Schriftfarbe blau wird. Angewendet wird dies, um dem Anwender genau zeigen zu können an welcher Stelle des Algorithmus er sich im Moment genau befindet. Dazu werden die Klassen

```
.orange {font-size: 13px; color:#FFCC33;}  
.blau {font-size: 13px; color:#0000dd;}  
.rot {font-size: 13px; color:#dd0000;}  
.grun {font-size: 13px; color:#00dd00;}
```

ausgetauscht, die dem Quelltext die entsprechende Schriftfarbe geben. Abbildung 6.9 zeigt, wie das im Programm aussieht.



```
Algorithmus in Javascript
function bubblesort(Values[]){
  for (var j=Values.length-1;j>0;j--){
    for (var i=0;i<j;i++){
      if (Values[i]>Values[i+1]){
        //vertauschen
        var tausch = Values[i];
        Values[i] = Values[i+1];
        Values[i+1] = tausch;
      }
    }
  }
}
```

Abb. 6.9: CSS-Änderung der Farbe
für ein Element

7 Vorstellung der Algorithmen

Der Algorithmenbaukasten besteht aus verschiedenen grundlegenden Algorithmen der Informatik. Wie bereits im vorherigen Kapitel beschrieben, beinhaltet er neben diversen Sortier- und Suchalgorithmen außerdem einen speziellen Bereich zu den Binärbäumen sowie dem Hashing.

In diesem Kapitel werden die einzelnen Algorithmen vorgestellt.

7.1 Notationen

Zunächst beschäftigt sich dieses Kapitel aber mit den Notationen, die bei der Zusammenfassung der Algorithmen verwendet werden.

Von Bedeutung sind die Notationen bei der Laufzeitkomplexität von Algorithmen. Hierbei berücksichtigt man grundsätzlich drei verschiedene Fälle - den schlechtesten, den besten und den durchschnittlichen bzw. genauen Fall.

Die wichtigste ist die so genannte O-Notation. Diese wurde durch den deutschen Mathematiker Paul Bachmann (1837-1920) im Jahr 1894 in seinem Werk „Analytische Zahlentheorie“ eingeführt. Darin schrieb er: "... wir durch das Zeichen $O(n)$ eine Größe ausdrücken, deren Ordnung in Bezug auf n die Ordnung von n nicht überschreitet; ob sie wirklich Glieder von der Ordnung n in sich enthält, bleibt bei dem bisherigen Schlußverfahren dahingestellt." [BAC1894]

Im Jahr 1909 machte der Zahlentheoretiker Edmund Landau (1877-1938) in seinem Werk „Handbuch der Lehre von der Verteilung der Primzahlen“ von der O-Notation Gebrauch, weshalb diese auch als Schreibweise Landau'scher Symbole bezeichnet wird.



Bild 7.1: Paul Gustav
Heinrich Bachmann



Bild 7.2: Edmund Georg
Hermann Landau

Hierbei verfolgt man drei Grundgedanken. Der erste ist die Betrachtung der Laufzeit in Abhängigkeit von den Eingabedaten des Algorithmus. Diese Abhängigkeitsuntersuchung ist wichtig, da man bei der Laufzeitbetrachtung das Verhalten eines Algorithmus bei wachsender Anzahl der Eingabewerte untersuchen will. Zusätzlich erhält man die Möglichkeit, das prinzipielle Problem mathematisch durch eine Funktion zu beschreiben, deren Parameter man gegen unendlich streben lässt.

Zum Zweiten sollen "unwesentliche" Konstanten außer Acht gelassen werden. Es wird mittels der O-Notation eine Funktion gesucht, die die Laufzeit eines Algorithmus beschreibt, ohne von konstanten Faktoren wie dem verwendeten Rechnermodell, der eingesetzten Programmiersprache und deren Implementierung oder der Taktfrequenz der CPU abhängig zu sein. Es soll dementsprechend ohne Bedeutung sein, wie viel Zeit ein Rechner für eine so genannte "Basis-Operation", beispielsweise eine IF-Abfrage, in Anspruch nimmt, das heißt ob der Algorithmus auf einem Supercomputer oder einem handelsüblichen PC läuft. Wie sich der Algorithmus bei unterschiedlichen Eingabewerten verhält, ist der Zweck dieser Untersuchung.

Der dritte Grundgedanke richtet sich an die Untersuchung einer oberen Schranke: das heißt dass die Funktion, welche mittels O-Notation angegeben wird, multipliziert mit einem rechnerabhängigen konstanten Faktor, stets über der eigentlichen Laufzeitfunktion liegen soll, wenn die Anzahl der Eingabewerte einmal einen bestimmten Wert überschritten haben. Also untersucht man mit der O-Notation immer die Laufzeit im ungünstigsten Fall, im "worst case". [8]

Ebenso wie die O-Notation die obere Schranken für die Laufzeit eines Algorithmus' angibt, definiert die so genannte Ω -Notation die untere Schranken. Die Ω -Notation bildet gemeinsam mit der O-Notation ein elegantes Werkzeug beim Erstellen von Algorithmen. Kann nämlich gezeigt werden, dass $\Omega(f(n))=O(f(n))$, so ist bewiesen, dass es keinen grundlegend schnelleren Algorithmus für das betreffende Problem geben kann.

Des Weiteren beschäftigt sich die Θ -Notation mit der genauen bzw. mittleren Schranke des Algorithmus.

7.2 Sortieralgorithmen

Sortieralgorithmen setzen voraus, dass die zu sortierenden Elemente nach irgendeinem Kriterium vergleichbar sind. Dies kann so einfach sein wie, die Größe von Zahlen oder so kompliziert, wie die Bewertung der Siegchancen einer Stellung in einem Schachspiel. Natürlich kann sich dies nicht nur auf Zahlen beziehen. Weitere Möglichkeiten wären Strings, beispielsweise eine nach den Familiennamen zu sortierende Mitgliedskartei eines Sportvereins, oder Uhrzeit/Datum. Denkbar wäre hier etwa die Abflug- bzw. Landezeiten einzelner Flugzeuge zu sortieren, damit sich ein Passagier am Flughafen schneller seinen Flug herausuchen kann. Für die Sortieralgorithmen selbst spielt die Art der verglichenen Information jedenfalls keine Rolle.

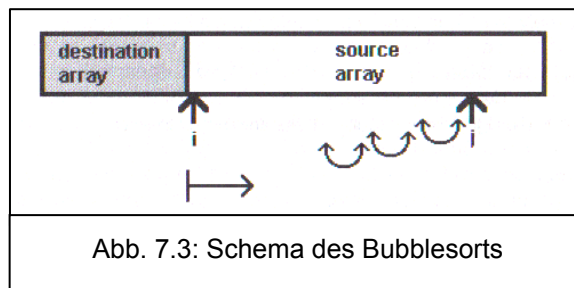
Der Einfachheit halber werden im Algorithmenbaukasten Zahlen von 0 bis 99 sortiert. Hauptaugenmerk soll dementsprechend in erster Linie auf den Algorithmen selbst liegen. So soll sichergestellt werden, dass der/die Anwender/in diese schnellstmöglich erlernen und natürlich auch verstehen kann.

Die Qualität von Sortieralgorithmen lässt sich unter verschiedenen Gesichtspunkten vergleichen. Dabei ist in erster Linie die Geschwindigkeit interessant, die üblicherweise an der Anzahl Elementvergleiche und der Anzahl der Kopieraktionen gemessen wird. Die konkrete Anzahl der Operationen hängt stark davon ab, in wie weit die Daten bereits vorsortiert sind. Bei einer Datenmenge, die beispielsweise schon komplett sortiert ist, werden bei den guten Algorithmen weniger Kopieraktionen durchgeführt, als bei einer Datenmenge, die zufällig angeordnet ist. Der schlimmste bzw. aufwändigste Fall wird auftreten, wenn die zu sortierende Datenmenge bereits sortiert, aber in falscher Reihenfolge vorliegt.

7.2.1 Bubblesort

Bubblesort ist das einfachste Sortierverfahren. Allerdings hat es eine Zeitkomplexität von $\Theta(n^2)$, damit ist es für größere Datenmengen nicht geeignet. Denn bereits bei großen Datenmengen ist Bubblesort erheblich langsamer als ein schnelles Sortierverfahren.

Der Name "Bubblesort" kommt von dem englischen Wort bubble, was auf Deutsch Blase bedeutet, da man seine Funktionsweise sehr gut anhand von langsam aufsteigenden Luftblasen in einer Wassersäule erklären kann. Bubblesort beruht auf folgender Idee: Der Datensatz wird vom Anfang zum Ende durchkämmt und die Elemente dabei paarweise verglichen. Wenn beide in der richtigen Reihenfolge stehen (das kleinere zuerst, das größere hinterher), dann wird mit dem nächsten Paar fortgefahren. Wenn beide in der falschen Reihenfolge stehen, werden sie vertauscht. Das größte Element wandert dabei im Laufe eines Durchgangs an das Ende des Datensatzes.



Dann wird der Prozess wiederholt, aber ohne das letzte Element, da es sich nun an der richtigen Stelle befindet. Das bedeutet, dass nun das insgesamt zweithöchste Element an die vorletzte Stelle wandert. Im nächsten Durchgang werden die letzten beiden Elemente ausgelassen usw. Ersichtlich ist dieser Vorgang in Abbildung 7.3. Das jeweils höchste Element wandert dabei aus dem unsortierten Bereich des Arrays

(source array) in den sortieren Teil (destination array). Der Datensatz ist vollständig sortiert, wenn nur noch ein Element zu "sortieren" ist.

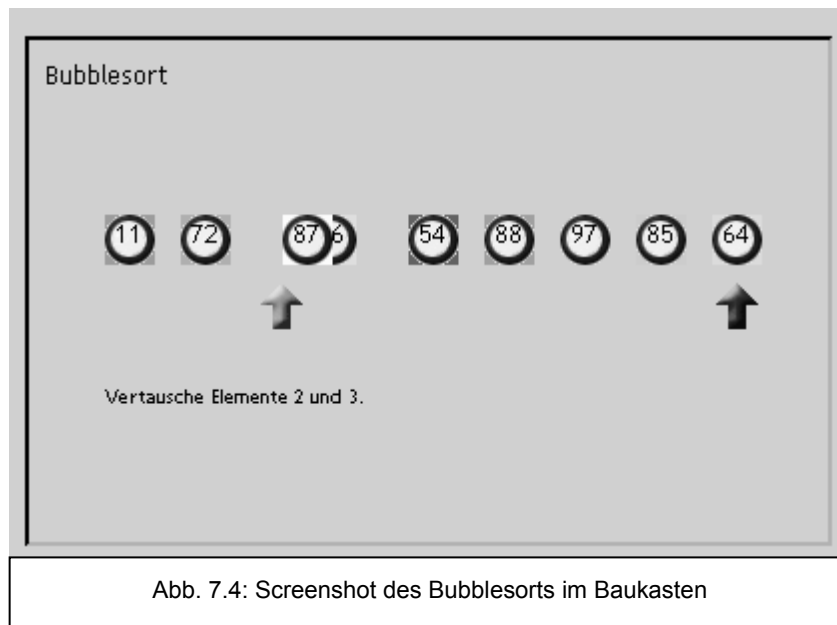
Bubblesort folgt einem so einleuchtenden Mechanismus, dass der Algorithmus oft aus Ungeduld gewählt wird, obwohl bessere Algorithmen verfügbar sind. Für kleine oder weitgehend sortierte Datensätze ist Bubblesort allerdings eine sehr gute Wahl. [LAN02]

Für einen durchschnittlichen Datensatz braucht Bubblesort in der Größenordnung von $n^2/2$ Vergleiche und halb so viele Vertauschungen.

Algorithmus

```
function bubblesort(Values[]){  
  for (var j=Values.length-1;j>0;j--){  
    for (var i=0;i<j;i++){  
      if (Values[i]>Values[i+1]){  
        //vertauschen  
        var tausch = Values[i];  
        Values[i] = Values[i+1];  
        Values[i+1] = tausch;  
      }  
    }  
  }  
}
```

Screenshot



Analyse

Die günstigste Ausgangsbasis für den Bubblesort ist eine bereits in der richtigen Reihenfolge sortierte Datenmenge. In diesem Fall ist das jeweils nächste Element größer und muss nicht ausgetauscht werden.

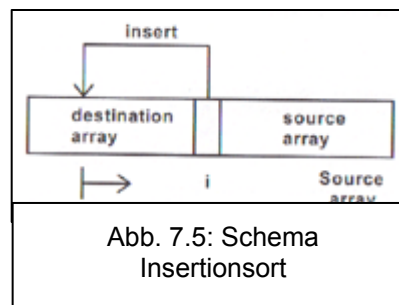
Der ungünstigste Fall ergibt sich demnach ebenfalls bei einer sortierten Datenmenge, allerdings in der verkehrten Reihenfolge. Dadurch ergibt sich, dass jedes folgende Element kleiner ist als sein Vorgänger und sie müssen ausgetauscht werden.

Zusammenfassung [7]

	günstig	mittel	ungünstig
Speicher	$\Omega(1)$	$\Theta(1)$	$O(1)$
Laufzeit	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Vergleiche	$n^2/2$	$n^2/2$	$n^2/2$
Vertauschungen	0	$n^2/4$	$n^2/2$

7.2.2 Insertionsort

Zu Beginn werden die zu sortierenden Daten geteilt, in eine bereits sortierte (das erste Element) und eine unsortierte Menge (der Rest). In Abbildung 7.5 wird dies als destination array bzw. source array bezeichnet. Der Algorithmus entnimmt der unsortierten Eingabemenge bzw. dem source array ein beliebiges Element, meist wird dabei das erste genommen, und fügt es an richtiger Stelle in den geordneten Bereich ein. Das Verfahren arbeitet in-place, als Ergebnis der Sortierung wird also die selbe Menge zurückgegeben. Wird auf einem Array gearbeitet, so müssen die Elemente nach dem neu eingefügten Element verschoben werden.



Algorithmus

```
function insertionSort(Values[]){
  for ( var i=1; i < Values.length; i++){
    var temp = Values[i];
    for (var j = i; (j > 0) && ( Values[j-1] > temp); j--){
      Values[j] = Values[j-1];
    }
    Values[j] = temp;
  }
}
```

Screenshot

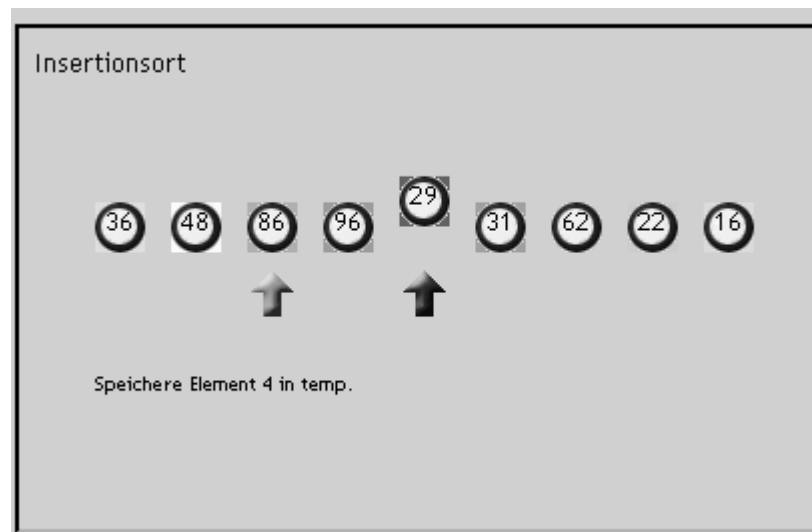


Abb. 7.6: Screenshot des Insertionsorts im Baukasten

Analyse

Im schlechtesten Fall wird der Platz für das einzufügende Element immer erst ganz am Anfang des sortierten Teils gefunden. Dies bedeutet in der äußeren for-Schleife werden Folgen der Länge 1, 2, 3, ..., $n-1$ durchsucht. Insgesamt sind dies $(n-1) * n / 2$ Schritte, also $\Theta(n^2)$ Schritte. Dieser Fall tritt ein, wenn die Folge zu Anfang in absteigender Reihenfolge sortiert ist. Es ginge auch schneller, die Einfügeposition des Elements a_i innerhalb des sortierten Teils $a_0, \dots, a_{i-1}$ zu finden, nämlich mit binärer Suche. Da aber die größeren Elemente alle nach rechts rücken müssen, um die Einfügeposition frei zu machen, ist für das Einfügen ohnehin lineare Zeit erforderlich. [LAN02]

Zusammenfassung [7]

	günstig	mittel	ungünstig
Speicher	$\Omega(1)$	$\Theta(1)$	$O(1)$
Laufzeit	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Vergleiche	$n^2/2$	$n^2/2$	$n^2/2$
Vertauschungen	0	$n^2/4$	$n^2/2$

7.2.3 Quicksort

Quicksort wurde 1960 von Charles Antony Richard Hoare, besser bekannt als Tony Hoare, in seiner Grundform erfunden und seitdem von vielen Forschern weiterentwickelt. Der Algorithmus hat den Vorteil, dass er über eine sehr kurze innere Schleife verfügt, was die Ausführungsgeschwindigkeit stark erhöht. Daher kommt auch der Name. Quick ist das englische Wort für schnell.



Abb. 7.7: Tony Hoare

Quicksort wählt ein Element, meist die Mitte oder das Letzte, aus der zu sortierenden Liste aus, das so genannte Pivotelement, und zerlegt die Liste in zwei Teillisten, eine untere, die alle Elemente kleiner und eine obere, die alle Elemente gleich oder größer

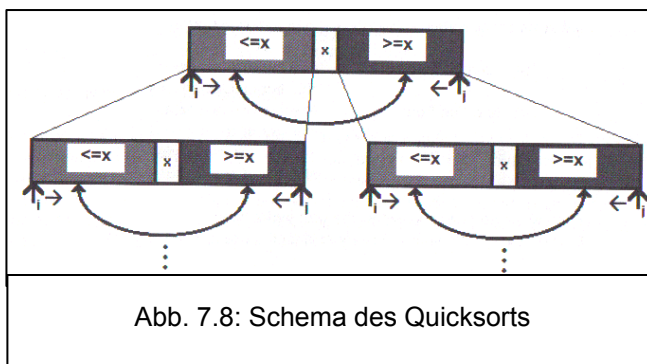


Abb. 7.8: Schema des Quicksorts

dem Pivotelement enthält. Dazu wird zunächst ein Element von unten gesucht, das größer (oder gleich-groß) als das Pivotelement und damit für die untere Liste zu groß ist. Entsprechend wird von oben ein kleineres Element als das Pivot-

element gesucht. Die beiden Elemente werden dann vertauscht und landen damit im jeweils richtigen Bereich. Der Vorgang wird fortgesetzt, bis sich die untere und obere Suche treffen. Damit sind die oben erwähnten Teillisten in einem einzigen Durchlauf entstanden. Suche und Vertauschung können in-place durchgeführt werden.

Die noch unsortierten Teillisten werden über denselben Algorithmus in noch kleinere Teillisten zerlegt (z.B. durch Rekursion, wie auch im Baukasten verwendet) und, sobald nur noch Listen mit je einem Element vorhanden sind, wieder zusammengesetzt. Die Sortierung ist damit abgeschlossen.

Algorithmus

```
var Values = new Array();  
function quicksort(l,r) {  
  var i = l; j = r;  
  var m = Values[Math.round((l+r)/2)];  
  do {  
    while (Values[i]<m) {i++;}  
    while (m<Values[j]) {j--;}  
    if (i<=j) {  
      var tausch = Values[i];  
      Values[i] = Values[j];  
      Values[j] = tausch;  
      i++;  
      j--;  
    }  
  } while (i<=j);  
  if (l<j) {quicksort(l,j)};  
  if (i<r) {quicksort(i,r)};  
}
```

Screenshot

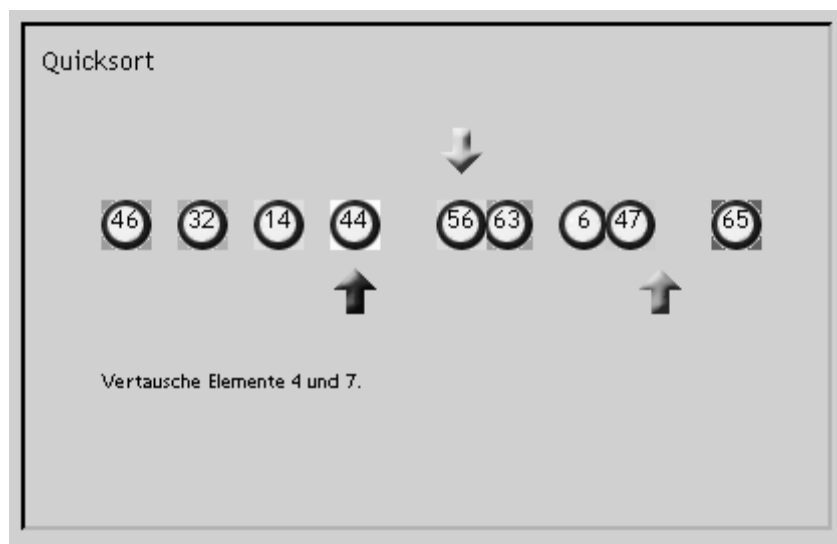


Abb. 7.9: Screenshot des Quicksort im Baukasten

Analyse:

Der Algorithmus verläuft optimal, wenn jeder Aufteilungsschritt im Verlauf der Rekursion jeweils etwa gleichlange Teilstücke erzeugt. In diesem günstigsten Fall benötigt Quicksort $\Theta(n \log(n))$ Schritte, denn die Rekursionstiefe ist $\log(n)$ und in jeder Schicht sind n Elemente zu behandeln. Der ungünstigste Fall tritt ein, wenn ein Teilstück stets nur aus einem Element und das andere aus den restlichen Elementen besteht. Dann ist die Rekursionstiefe $n-1$ und Quicksort benötigt $\Theta(n^2)$ Schritte. In diesem ungünstigsten Fall benötigt Quicksort in etwa genauso lange wie der bereits beschriebene Insertionsort.

In der Praxis wird dies allerdings in Kauf genommen, da sich gezeigt hat, dass auch im Durchschnitt nur $O(n \log(n))$ Schritte benötigt werden. Damit ist es das schnellste Sortierverfahren. [LAN02]

Zusammenfassung [7]

	günstig	mittel	ungünstig
Speicher	$\Omega(\log n)$	$\Theta(\log n)$	$O(n)$
Laufzeit	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Vergleiche	$n \log n$	$1.44 n \log n$	$n^2/2$
Vertauschungen	0	$n \log n / 4.3$	$n^2/4$

7.2.4 Mergesort

Mergesort ist ein rekursiver Sortieralgorithmus, der ähnlich wie Quicksort nach dem Prinzip „Teile und herrsche“ arbeitet. Er wurde erstmals 1945 durch John von Neumann vorgestellt.



Abb. 7.10: John von Neumann

Der Name Mergesort leitet sich von dem englischen Wort (to) merge ab. Auf Deutsch bedeutet dies soviel wie zusammenfügen.

Mergesort betrachtet die zu sortierenden Daten als Liste und zerlegt sie in kleinere Listen, die jede für sich sortiert werden. Im Endeffekt wird die gesamte Datenmenge solange zerteilt, bis jedes einzelne Element eine eigene Liste darstellt. Diese kleinen Listen werden dann in der korrekten Reihenfolge im Reißverschlussverfahren zu größeren Listen zusammengefügt bis wieder eine sortierte Gesamtliste erreicht ist.

Algorithmus

```

function mergesort(links,rechts){
  if (rechts-links>0){
    var mitte = (rechts+1)/2;
    mergesort(links, mitte);
    mergesort(mitte+1, rechts);
    for (i=mitte;i>=links;i--) b[i] = Values[i];
    for (j=mitte+1;j<=rechts;j++) b[rechts+mitte+1-j] = Values[j];
    for (k=links;k<=rechts;k++){
      if (b[i]<b[j]){
        Values[k]=b[j];
        i=i+1;
      }
      else {
        Values[k]=b[i];
        j=j-1;
      }
    }
  }
}

```

Screenshot

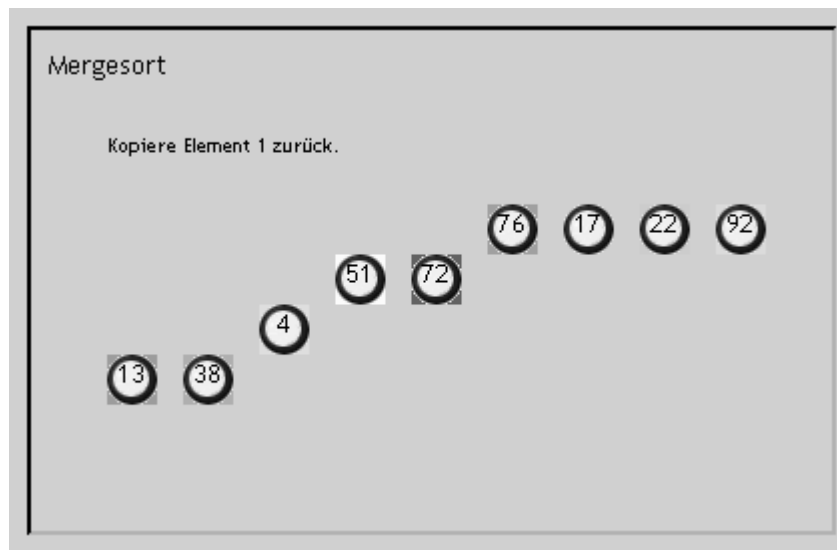


Abb. 7.11: Screenshot des Mergesort im Baukasten

Analyse

Das Sortierverfahren Mergesort hat eine Zeitkomplexität von $\Theta(n \log(n))$ und ist damit optimal. Anders als das ebenfalls optimale Heapsort (Abschnitt 7.2.7) benötigt Mergesort allerdings zusätzlichen Speicherplatz der Größe $\Theta(n)$ für ein Hilfsarray, in das die einzelnen Listen beim anfänglichen Zerschneiden kopiert werden.

Zusammenfassung [7]

	günstig	mittel	ungünstig
Speicher	$\Omega(n)$	$\Theta(n)$	$O(n)$
Laufzeit	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Vergleiche	$n \log n$	$n \log n$	$n \log n$
Vertauschungen	$2n \log n$	$2n \log n$	$2n \log n$

Die Grundidee des Verfahrens ist, die Daten als zweidimensionales Feld zu arrangieren und spaltenweise zu sortieren. Dadurch wird eine Grobsortierung bewirkt. Dann werden die Daten als schmaleres zweidimensionales Feld angeordnet und wiederum spaltenweise sortiert. Dann wird das Feld wiederum schmaler gemacht usw. Zum Schluss besteht das Feld nur noch aus einer Spalte.

Werden die Feldbreiten geschickt gewählt, reichen jedes Mal wenige Sortierschritte aus, um die Daten spaltenweise zu sortieren bzw. am Ende, wenn nur noch eine Spalte vorhanden ist, vollständig zu sortieren.

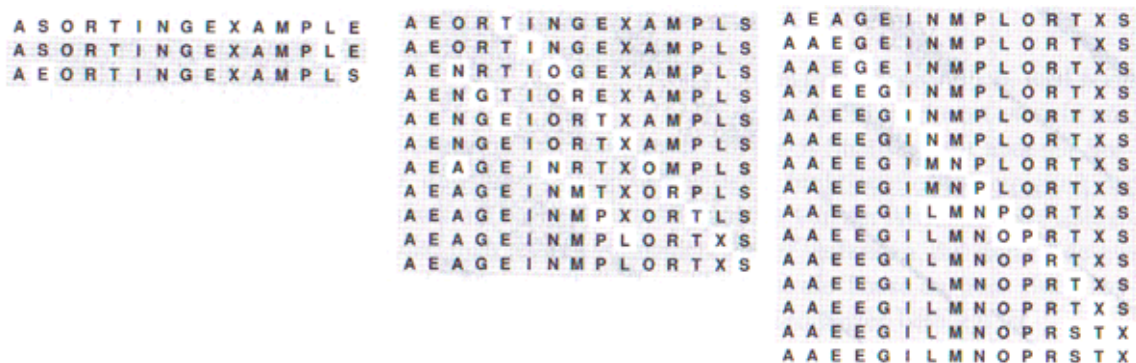


Abb. 7.12: Schema des Shellsorts anhand des Strings „ASORTINGEXAMPLE“

Will man beispielsweise wie in Abbildung 7.12 dargestellt „ASORTINGEXAMPLE“ sortieren und verwendet dazu die Folge:

1, 4, 13, 40, 121, 364, ..

so kann dies innerhalb von drei Durchgängen geschehen.

Die Ausgangsdatenmenge besteht in diesem Beispiel aus 15 Elementen. Die Menge wird in die größtmögliche Anzahl an Spalten geteilt wird, die auch in der verwendeten Folge enthalten ist. Dadurch ergibt sich die Spaltenanzahl 13. In Abbildung 7.12 ist dieser Vorgang im linken Block dargestellt. Bei 15 Elementen, die in 13 Spalten eingeteilt werden, haben lediglich die ersten beiden Spalten mehr als ein Element und müssen daher geordnet werden. In der ersten Spalte befinden sich die Elemente an der Stelle 0 (=A) und 13 (=L) des Array. Da „A“ vor „L“ im Alphabet ist, werden diese nicht vertauscht (zweite Zeile). In der zweiten Spalte befinden sich die Elemente „S“ und „E“, die vertauscht werden (dritte Zeile). Der erste Durchgang ist damit beendet.

In zweiten Durchgang wird das Array gemäß Folge in vier Spalten aufgeteilt, zu sehen im mittleren Block der Abbildung 7.12. Im letzten Durchgang, in dem das Array in eine Spalte „aufgeteilt“ wird müssen aufgrund der Vorgehensweise von Shellsort die einzelnen Elemente nicht mehr weit verschoben bzw. ausgetauscht werden, da bereits in den bisherigen beiden Durchgängen eine sehr gute Vorsortierung durchgeführt wurde.

Algorithmus

```
function shellsort(){  
  var i, j, k, h, t;  
  var cols = new Array(4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1);  
  for (k=0; k<16; k++){  
    h=cols[k];  
    for (i=h; i<n; i++){  
      j=i;
```

```
t=a[j];  
while (j>=h && a[j-h]>t){  
    a[j]=a[j-h];  
    j=j-h;  
}  
a[j]=t;  
}  
}  
}
```

Screenshot

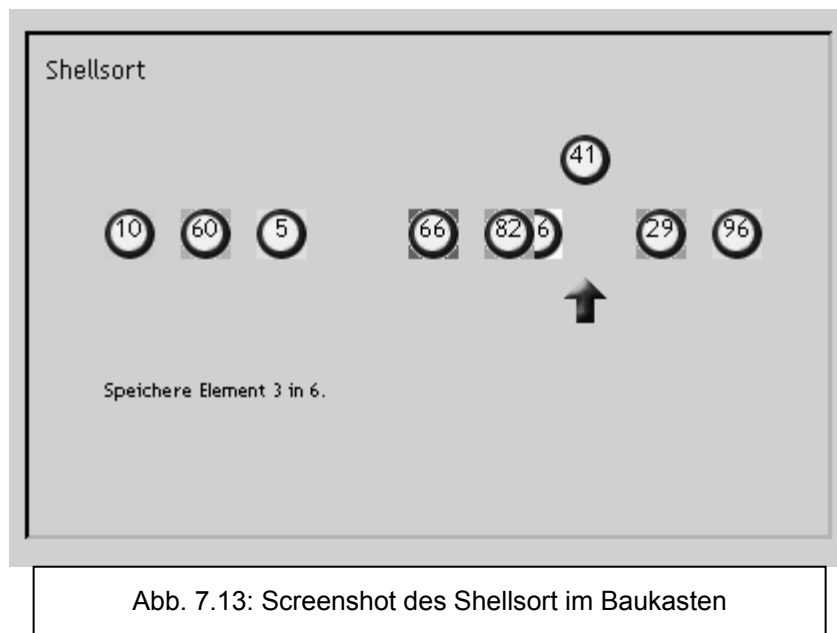


Abb. 7.13: Screenshot des Shellsort im Baukasten

Analyse

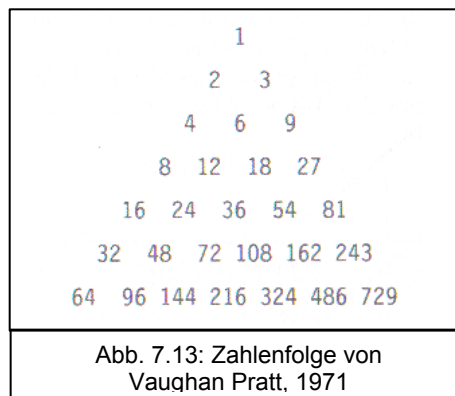
Die Komplexität von Shellsort hängt sehr stark von der Wahl der Folge für die Spaltenanzahl ab. An sich ist dieser Algorithmus eine sehr einfache Art, Daten zu sortieren. Hier kann dies aber auch sehr kompliziert gemacht werden. Wählt man für die Folge nach der sortiert wird zu viele Elemente, wird der Sortiervorgang schnell zu umständlich und lange dauern. Auch bei einer zu kurzen Folge kann dieser Fall eintreten, da dann die Vorsortierung fehlt.

Mit der von Papernov und Stasevich 1965 entwickelten Folge:

$$1, 3, 7, 15, 31, 63, \dots, 2^k - 1$$

wird eine Komplexität von $O(n^{1.5})$ erreicht [PAP65]. Basis dieser ist die Tatsache, dass die einzelnen Elemente der Folge und damit die ausgewählten Spaltenzahlen teilerfremd sind.

Eine andere Art von Folgen setzt genau auf das Gegenteil. Dabei wird bewusst darauf geachtet, dass die einzelnen Elemente nicht teilerfremd sind, bekanntes Beispiel ist die Folge von Vaughan Pratt, entwickelt 1971.



Die Zahlen in diesem Dreieck stehen zueinander in einem besonderen Verhältnis. Der linke Nachfolger entspricht jeweils genau dem doppelten der Ausgangszahl, der rechte Nachfolger dem Dreifachen. Hier wird immerhin eine Komplexität von $O(n \cdot \log(n)^2)$ erreicht. Allerdings hat sich in der Praxis diese Folge nicht durchsetzen können, da es einfach zu viele Elemente gibt und damit zu viele Durchgänge erforderlich sind. Jedoch haben sich Abstandsfolgen, die nach dem gleichen Prinzip aufgebaut sind, aber aus zwei beliebigen teilerfremden Zahlen entstehen, bewährt.

Eine Folge für $O(n \cdot \log(n))$ scheint jedoch nicht zu existieren. Es wurde experimentell belegt, dass die allgemeine Komplexität in $O(n^{1.25})$ liegt. [LAN02]

Zusammenfassung [7]

Die ursprünglich von Donald L. Shell (1959) konzipierte Folge

"1,2,4,8,16,32,64,128,256,...,x" verspricht eine Laufzeit, die in $\Omega(n \log n)$ und $O(n^2)$ liegt. Diese Folge ist sehr ungünstig und sollte nicht angewendet werden.

	günstig	mittel	ungünstig
Speicher	$\Omega(1)$	$\Theta(1)$	$O(1)$
Laufzeit	$\Omega(n \log n)$	$\Theta(n (\log n)^4)$	$O(n^2)$
Vergleiche	$n \log n$	$n (\log n)^{3.0/35}$	???
Vertauschungen	0	$n (\log n)^{3.8/326}$	???

Im Vergleich dazu verspricht die Folge von Papernov-Stasevich (1965)

"1,3,7,15,31,63,127,255,511,...,x" eine Laufzeit, die in $\Omega(n \log n)$ und $O(n^{3/2})$ liegt.

	günstig	mittel	ungünstig
Speicher	$\Omega(1)$	$\Theta(1)$	$O(1)$
Laufzeit	$\Omega(n \log n)$	$\Theta(n (\log n)^2)$	$O(n^{3/2})$
Vergleiche	$n \log n$	$n (\log n)^{1.7/3.42}$	???
Vertauschungen	0	$n (\log n)^{2.1/17.01}$	???

7.2.6 Bucketsort

Bucketsort zählt die Häufigkeit jedes Wertes in einer Datenmenge. Daraus errechnet es die korrekte Position jedes Elements und fügt es in einer zweiten Liste wieder ein.

Die Häufigkeit der Schlüsselwerte wird in einem so genannten Histogramm gespeichert. Dies wird meist als Array implementiert, das so lang ist, wie es mögliche Schlüsselwerte gibt; als Indizes werden dabei die Schlüsselwerte bzw. die ihnen zugeordneten ganzen Zahlen gewählt. Elemente mit gleichem Sortierschlüssel werden dabei in Gruppen, so genannten Buckets, zusammengefasst.

Das Histogramm wird zunächst mit Nullen initialisiert. Dann wird die zu sortierende Datenmenge durchlaufen und bei jedem Element der entsprechende Histogrammeintrag um eins erhöht.

In einem zweiten Array, das ebenso lang ist wie das Histogramm-Array und ebenfalls mit Nullen initialisiert wird, werden nun die aus dem Histogramm errechneten Einfügepositionen gespeichert.

Schließlich wird ein Array gebildet, das ebenso lang ist wie die zu sortierende Datenmenge Elemente besitzt. In dieses werden die Elemente gemäß dem Histogramm-Array nacheinander an den berechneten Positionen eingefügt.

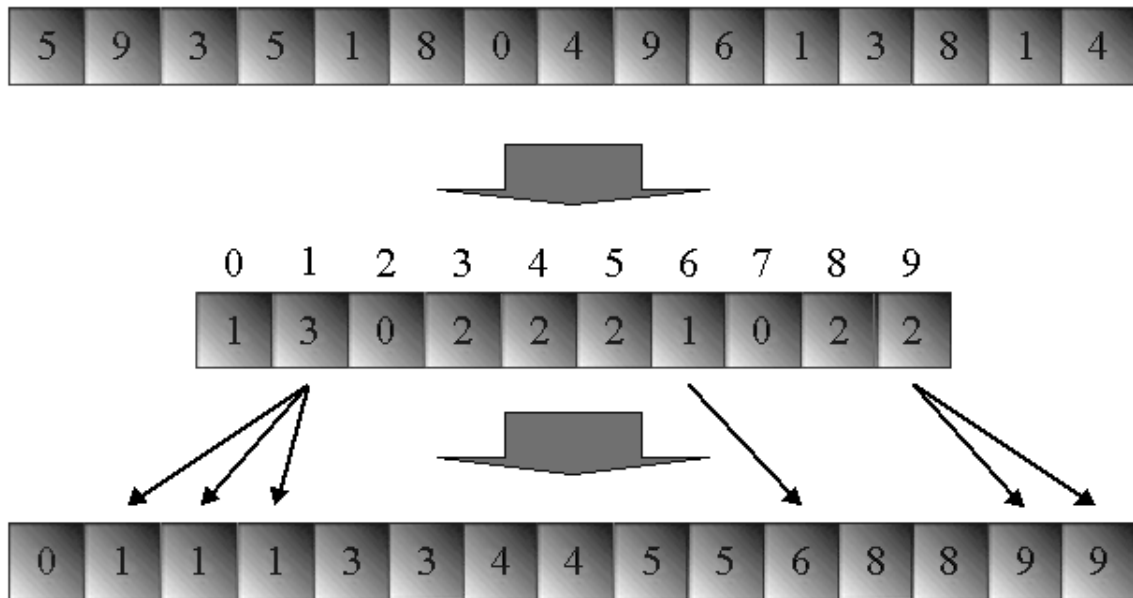


Abb. 7.15: Schema des Bucketsorts

Damit eine Array mit Bucketsort sortierbar ist, muss die Anzahl der von den Sortierschlüsseln annehmbaren Werte gegeben sein.

Bucketsort ist beispielsweise gut geeignet, um eine Adressliste nach Postleitzahlen zu ordnen.

Algorithmus

```
function bucketsort() {
  for (var a=1; a<=m; a++) { // m = Anzahl der möglichen Elemente
    temp[a] = 0;
  }
  for (var a=1; a<=n; a++) { // n = Anzahl der Array-Elemente
    temp[Values[a]]++;
  }
  var x=1;
  for (var a=1; a<m+1; a++) {
    while (temp[a] > 0) {
      Values[x++] = a;
      temp[a]--;
    }
  }
}
```

Screenshot

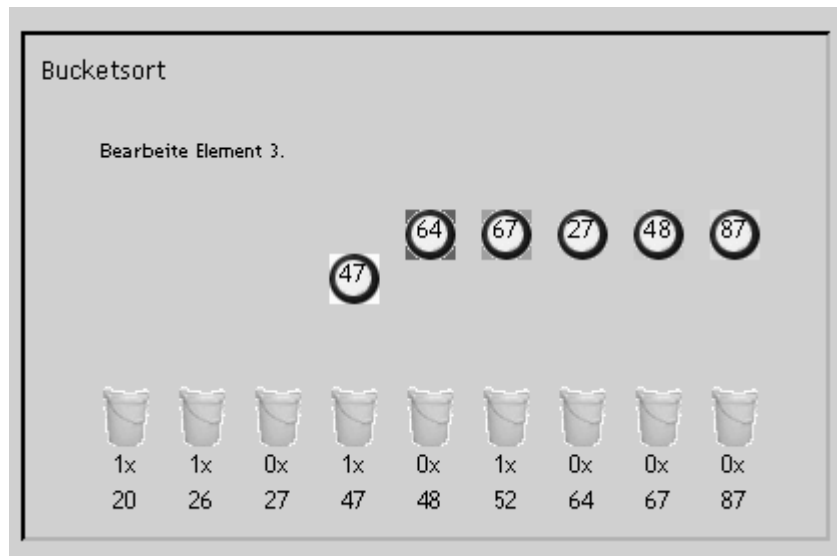


Abb. 7.16: Screenshot des Bucketsort im Baukasten

Analyse

Die asymptotische Laufzeit von Bucketsort ist abhängig von der Länge des zu sortierenden Array und der Anzahl der unterschiedlichen Werte, die angenommen werden können.

Für die Histogramm-Erstellung wird das zu sortierende Array einmal durchlaufen. Die Komplexität wächst also linear mit der Länge des Array. Die Aufwandssklasse der Histogramm-Erstellung ist also $O(n)$.

Um die Einfügepositionen zu berechnen, wird das Histogramm einmal durchlaufen. Da das Histogramm-Array so lang ist, wie es mögliche Werte gibt, ist die Aufwandssklasse für die Erstellung des Einfügestellen-Arrays $O(k)$.

Anschließend wird das zu sortierende Array erneut durchlaufen, um die Elemente im Zielarray zu speichern. Die Aufwandsklasse ist erneut $O(n)$. Der Speicherbedarf im worst case und im best case des Algorithmus unterscheiden sich nicht voneinander. Dadurch ergibt sich, dass der Bucketsort auch im Mittel einen Speicherbedarf von $\Theta(n)$ aufweist.

Zusammenfassung [7]

	günstig	mittel	ungünstig
Speicher	$\Omega(n)$	$\Theta(n)$	$O(n)$
Laufzeit	$\Omega(n)$	$\Theta(n \log(n/k))$	$O(n \log n)$
Vergleiche	n	$n \log(n/k)$	$n \log n$
Kopierungen	$2n$	$2n$	$2n$
Vertauschungen	0	n	n

7.2.7 Heapsort

Heapsort ist ein 1964 von Robert W. Floyd und John Joseph William entwickeltes, relativ schnelles Sortierverfahren. Seine Komplexität ist bei einem Array der Länge n in der Landau-Notation ausgedrückt $O(n \cdot \log(n))$.



Abb. 7.17: Robert W. Floyd

Die Voraussetzung dafür, dass man ein Array von sortierbaren Werten mit Heapsort sortieren kann, ist, dass dieses einen binären Heap repräsentiert, wie in Abbildung 7.18 dargestellt. Ist dies nicht der Fall, so muss man es zuerst in ein Heap überführen.

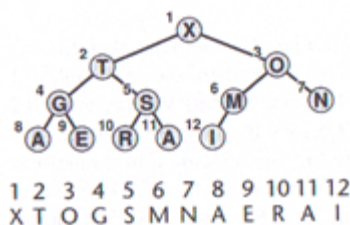


Abb. 7.18: Darstellung eines Heap-geordneten Binärbaums als Array

Um ein Array in einen Heap zu überführen, beginnt man deshalb in der Mitte des Arrays. Man "versickert" nun sukzessive alle davor liegenden Knoten, bis das erste Element versickert wurde. Versickern (auch: absinken) heißt, dass man einen Knoten

mit dem größeren seiner beiden Nachfolgeknoten vertauscht, falls dieser größer ist als er selbst, und dies so lange durchführt, bis er keinen Nachfolgeknoten mehr hat, der größer ist als er selbst.

Man beachte, dass für die zweite Hälfte jedes Arrays die Heap-Eigenschaft bereits erfüllt ist, denn jeder Knoten in der zweiten Hälfte des Arrays entspricht im Heap einem Blatt, hat also keinen Nachfolgeknoten, dessen Wert größer sein könnte als der eigene.

Der eigentliche Sortieralgorithmus nutzt die Tatsache aus, dass die Wurzel eines Heaps stets der größte Knoten ist. Da im fertig sortierten Array der größte Wert ganz rechts stehen soll, vertauscht man das erste mit dem letzten Arrayelement.

Das Element am Ende des Arrays ist nun an der gewünschten Position und bleibt dort. Den Rest des Arrays muss man wieder in einen Heap überführen, indem man das erste Element versickert.

Anschließend vertauscht man das erste mit dem vorletzten Element, d.h. die beiden größten Werte sind wie gewünscht am Ende des Arrays. Man versickert das nun erste Element, usw.

Algorithmus

```
function heapsort() {  
  var n = Values.length;  
  for (i = Math.floor((n-1)/2); i >= 0; i--) sink(i, n-1);  
  for (i = n-1; i >= 1; i--){  
    tauschen(0, i);  
    if (i > 1) sink(0, i-1);  
  }  
}  
  
function tauschen (i, j){  
  if (i != j){  
    Values[i] += Values[j];  
    Values[j] = Values[i] - Values[j];  
    Values[i] -= Values[j];  
  }  
}  
  
function sink(i, t){  
  var k = 2*(i+1);
```

```

var j = k-1;
if (j<=t){
  if (Values[i]>=Values[j]) j=i;
  if (k<=t){
    if (Values[k]>Values[j]) j=k;
  }
  if (i!=j){
    tauschen(i,j);
    sink(j,t);
  }
}
}
}

```

Screenshot

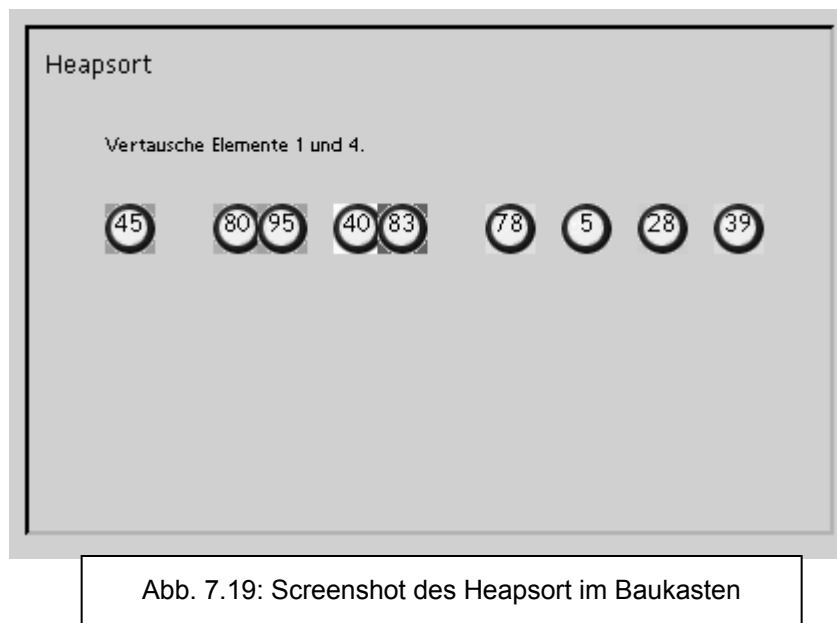


Abb. 7.19: Screenshot des Heapsort im Baukasten

Analyse

Man kann zeigen, dass der Aufbau des Heaps, in Landau-Notation ausgedrückt, in $O(n \cdot \log n)$ Schritten ablaufen kann. Das Versickern eines Elements von der Wurzel benötigt im ungünstigsten Fall $O(\log n)$ Schritte. Insgesamt garantiert HeapSort eine Laufzeit von $O(n \log n)$.

Für eine genügend große Datenmenge ist ein Heapsort im Durchschnitt schneller als Quicksort. Hinzu kommt das bessere Worst Case-Verhalten von $O(n \log n)$ gegenüber $O(n^2)$, so dass bei großen Sortiermengen der Heapsort zu bevorzugen ist.

Zusammenfassung [7]

	günstig	mittel	ungünstig
Speicher	$\Omega(1)$	$\Theta(1)$	$O(1)$
Laufzeit	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$
Vergleiche	n	$n(\log n - 2.5)$	$n \log n$
Vertauschungen	n	$n \log n$	$(n/2) \log n$

7.3 Suchalgorithmen

Das Sortieren von Daten ist meistens nur eine Vorbereitung auf weitere Aktionen, zum Beispiel auf das Suchen.

Das Ziel der Suche ist es, ein bestimmtes Element einer Liste zu finden, von dem der zugehörige Suchschlüssel bekannt ist. Da dieses Problem in der Informatik oft anzutreffen ist, sind die verwendeten Algorithmen - sowie deren Komplexität - sehr gut untersucht. Das Suchen von bestimmten Elementen in einer sortierten Menge geht in der Regel deutlich schneller als in einer Unsortierten, bringt aber insgesamt erst Vorteile, wenn viele Suchanfragen gestellt werden.

7.3.1 Lineare Suche

Lineare Suche ist ein Algorithmus, der auch unter dem Namen sequenzielle Suche bekannt ist. Er ist der einfachste Suchalgorithmus überhaupt.

Man geht dabei die Liste Element für Element durch, bis man es gefunden hat. Der Suchaufwand wächst linear mit der Anzahl der Elemente in der Liste.

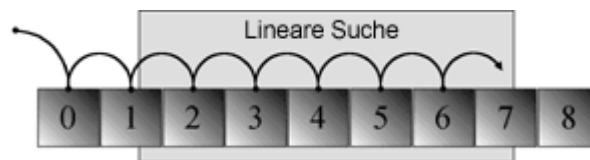


Abb. 7.20: Schema der linearen Suche

Algorithmus

```
function linearsearch(Values,suchzahl){  
  for (var j=0;j<Values.length;j++){  
    if (Values[j]==suchzahl) return true;  
  }  
  return false;  
}
```

Screenshot

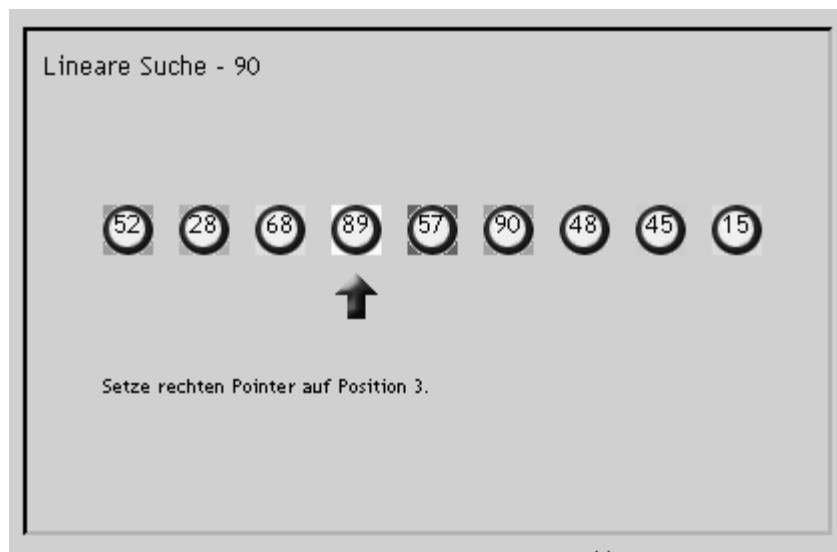


Abb. 7.21: Screenshot der linearen Suche im Baukasten

Analyse

Wenn die Daten zufallsverteilt sind, dann werden im Schnitt $n/2$ Vergleichsoperationen benötigt. Im besten Fall ist das erste Element der Liste das zu suchende Element, im schlechtesten ist es das letzte.

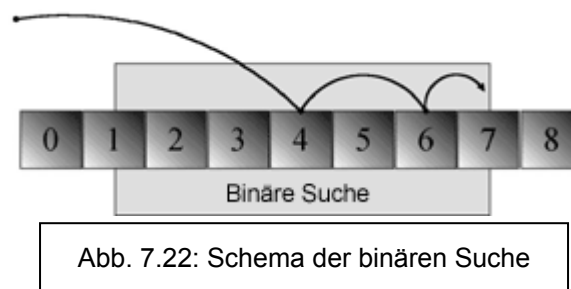
Zusammenfassung [BRE01] Seite 90

	günstig	mittel	ungünstig
Laufzeit	$\Omega(1)$	$\Theta((n+1)/2)$	$O(n)$
Vergleiche	1	$(n+1)/2$	n

7.3.2 Binäre Suche

Die binäre Suche ist ein Algorithmus, der ein gesuchtes Element innerhalb einer Datenmenge im Durchschnitt schneller findet als die lineare Suche. Voraussetzung ist, dass die Elemente der Liste in einer dem Suchbegriff entsprechenden Weise sortiert sind.

Zuerst wird das mittlere Element der Datenmenge überprüft. Es kann kleiner, größer oder gleich dem gesuchten Element sein. Ist es kleiner als das gesuchte Element, muss das gesuchte Element in der hinteren Hälfte stecken, falls es sich dort überhaupt befindet. Ist es hingegen größer, muss nur in der vorderen Hälfte weitergesucht werden. Die jeweils andere Hälfte muss nicht mehr betrachtet werden. Ist es gleich dem gesuchten Element, ist die Suche beendet.



Jede weiterhin zu untersuchende Hälfte wird wieder gleich behandelt: Das mittlere Element liefert wieder die Entscheidung darüber, wo bzw. ob weitergesucht werden muss.

Die Länge des Suchbereiches wird von Schritt zu Schritt halbiert. Spätestens wenn der Suchbereich auf ein Element geschrumpft ist, ist die Suche beendet. Dieses eine Element ist entweder das gesuchte Element, oder das gesuchte Element kommt nicht vor.

Algorithmus

```
var Values = new Array();  
function binaersearch(suchzahl, left, right){  
  var returnwert;  
  var mitte = Math.round((left+right)/2);  
  if Values[mitte]==suchzahl return true;  
  else  
    if left == right return false;  
    if suchzahl < Values[mitte] returnwert = binaersearch (suchzahl, left, mitte-1)  
    if suchzahl > Values[mitte] returnwert = binaersearch (suchzahl, mitte+1, right)  
  return returnwert;  
}
```

Screenshot

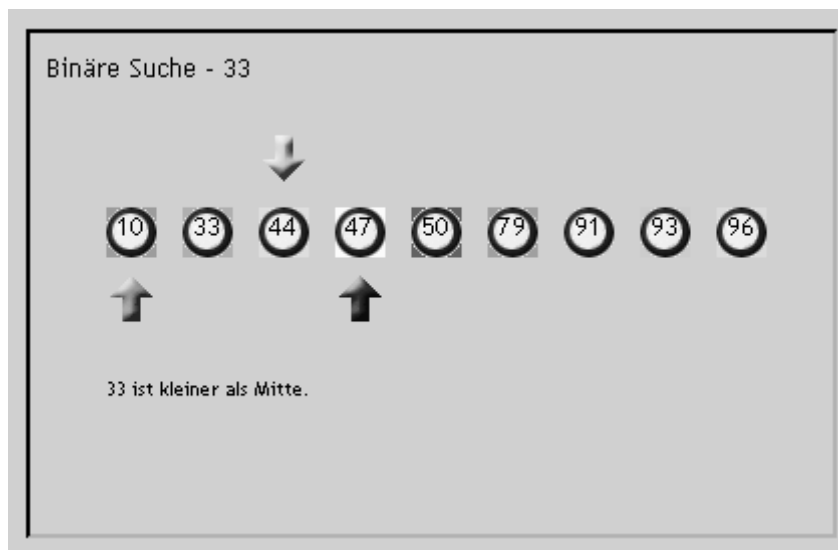


Abb. 7.23: Screenshot der binären Suche im Baukasten

Analyse

Um in einem Array mit n Einträgen ein Element zu finden bzw. festzustellen, dass dieses sich nicht im Array befindet, werden maximal $\log_2(n+1)$ Schritte benötigt. Damit ist sie vor allem bei einer großen Anzahl an Elementen deutlich schneller als die lineare Suche, die aber auch in unsortierten Arrays zur Anwendung gelangen kann.

Zusammenfassung [BRE01] Seite 95

	günstig	mittel	ungünstig
Laufzeit	$\Omega(1)$	$\Theta(\log_2(n+1)-1)$	$O(\log_2(n+1))$
Vergleiche	1	$\log_2(n+1)-1$	$\log_2(n+1)$

7.3.3 Interpolierte Suche

Die Interpolationssuche, auch Intervallsuche genannt, ist ein von der binären Suche abgeleitetes Suchverfahren.

Während der Algorithmus der binären Suche stets das mittlere Element des Suchraums überprüft, versucht der Algorithmus der Interpolationssuche im Suchraum einen günstigeren Teilungspunkt als die Mitte zu erraten. Die Arbeitsweise ist mit der eines Menschen vergleichbar, der ein Wort in einem Wörterbuch sucht: Die Suche nach dem Wort „Tauziehen“ wird üblicherweise eher am Ende des Wörterbuches beginnen, während die Suche nach „Fußball“ im vorderen Bereich begonnen werden dürfte.

Die Interpolationssuche geht ähnlich des Bubblesort von sortierten und gleichverteilten Daten aus. Die Daten werden bei der Interpolationssuche in Abhängigkeit vom Schlüssel geteilt. Hat dieser einen großen Wert, befindet sich das gesuchte Element aufgrund der Vorsortierung im hinteren Teil der Daten. Dementsprechend wird auch im hinteren Teil der Daten die Teilung vorgenommen. Bei einem kleinen Schlüssel wird das Feld entsprechend im vorderen Teil gespalten.

Für alle Daten lässt sich die Teilungsposition berechnen, indem zunächst die Anzahl aller Elemente durch die Anzahl verschiedener Elemente dividiert wird, und anschließend mit dem gesuchten Schlüssel multipliziert wird:

$$Position = \frac{Anzahl\ der\ Elemente}{Anzahl\ verschiedener\ Elemente} \cdot gesuchter\ Wert$$

Die Position des gesuchten Elementes wird somit interpoliert, indem die Gleichverteilung der Daten für eine Abbildung des Schlüssels auf die Liste bzw. das Feld genutzt wird.

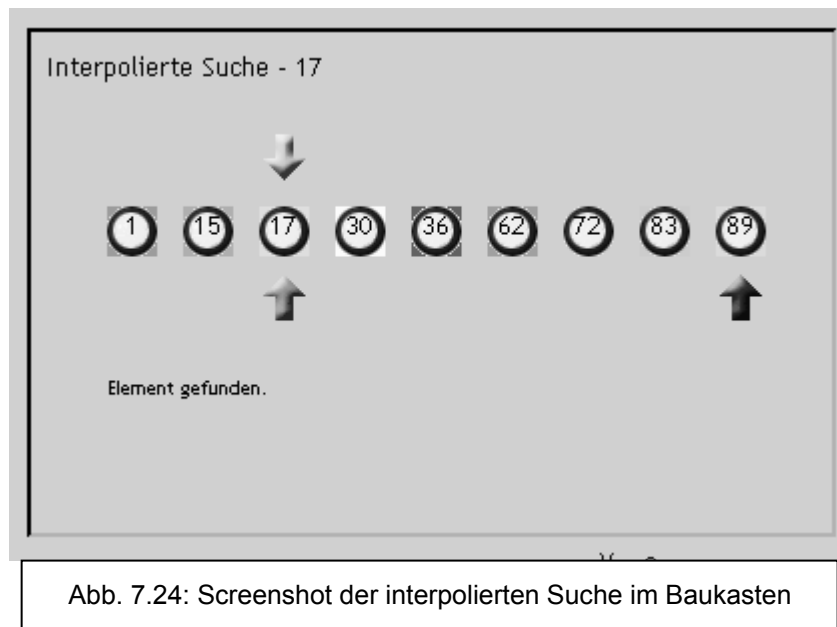
Nun kann überprüft werden, ob der Wert des teilenden Elementes einen größeren oder kleineren Wert als das gesuchte Element hat. Bei identischen Schlüsseln ist die Suche bereits beendet. Wenn das teilende Element einen kleineren Wert hat, wird der rechte Teilbereich untersucht, andernfalls der linke Teilbereich. Die Zahl der Elemente sowie die Zahl der verschiedenen möglichen Werte wird für den neuen Bereich ermittelt, und anschließend eine neue Teilungsposition interpoliert.

Algorithmus

```
function interpolierteSuche(suchzahl, Values){
  var links = 0; // linke Teilfeldbegrenzung
  var rechts = Values.length-1; // rechte Teilfeldbegrenzung
  var versch; // Anzahl verschiedener Elemente
  var pos; // aktuelle Teilungsposition

  while( suchzahl >= Values[links] && suchzahl <= Values[rechts] ){
    versch = Values[rechts] - Values[links];
    pos = links + Math.round((rechts - links) * (suchzahl - Values[links]) / versch);
    if( suchzahl > Values[pos] ) links = pos + 1;
    else if( suchzahl < Values[pos] ) rechts = pos - 1;
    else return pos; // Position zurückgeben
  }
  return -1; // Element nicht gefunden
}
```

Screenshot



Analyse

Eine Untersuchung der Interpolationssuche erweist sich als sehr komplex, als Laufzeit kann jedoch $O(\log(\log n))$ angenommen werden. Theoretisch ist die Interpolationssuche schneller als die binäre Suche, aufgrund der schwierigen Berechnung der Teilungsposition ist sie jedoch nur bei hohen Elementzahlen tatsächlich schneller.

Zusammenfassung [9]

	günstig	mittel	ungünstig
Laufzeit	$\Omega(1)$	$\Theta(\log(\log n))$	$O(\log n)$
Vergleiche	1	$\log(\log n)$	$\log n$

7.4 Hashing

Beim Hashverfahren wird die Datenmenge in eine Tabelle gespeichert. Diese Tabelle wird als die Hashtabelle bezeichnet. Das wesentliche Prinzip des Verfahrens beruht auf dem Einsatz einer Hashfunktion. Diese Funktion berechnet für jedes Element der Menge einen Index in der Hashtabelle. Dieser Index stellt die genaue Position des gesuchten Elementes in der Tabelle dar.

Da Hashfunktionen im Allgemeinen nicht eindeutig sind, können zwei unterschiedliche Schlüssel zum selben Hashwert, also zum selben Feld in der Tabelle führen. Dieses Ereignis wird als Kollision bezeichnet. In diesem Fall müsste die Hashtabelle mehrere Werte an derselben Stelle aufnehmen. Um dieses Problem zu handhaben, gibt es diverse Kollisionsauflösungsstrategien. Drei dieser Möglichkeiten werden in den Abschnitten 7.4.1 bis 7.4.3 vorgestellt.

Vorteile

Trotz der Schwierigkeiten, die sich beispielsweise aus der Kollisionsbehandlung ergeben, bieten Hashtabellen wegen der Vorzüge beim sofortigen Zugriff durch den Hashwert auf die Inhalte in einer Tabelle im Vergleich zu der Suche nach dem Suchschlüssel Vorteile in der Zugriffszeit als auch im benötigten Speicherplatz gegenüber gewöhnlichen Arrays.

Nachteile

Die Daten in einer Hashtabelle sind nicht sortiert, womit die sequentielle Ausgabe eines Teils der Einträge oder aller Einträge erheblich erschwert wird.

Hat die Tabelle einen gewissen Füllgrad überschritten, verfällt der Vorteil der geringeren Suchzeit gegenüber anderen Suchalgorithmen, beispielsweise denen in Kapitel 7.3 „Suchalgorithmen“ beschriebenen. Dann kann nur eine Vergrößerung der Tabelle mit nachfolgender Restrukturierung zu normalem Laufzeitverhalten zurückführen.

Hashfunktion

Sowohl im Algorithmenbaukasten als auch bei den Beispielen in diesem Kapitel wird die folgende Hashfunktion verwendet.

```
function hash(String, M){  
  var h=0, a=31415, b=27183;  
  for(var i=0;i<String.length();i++){  
    h = (a*h + String.charAt(i)) % M;  
    a = a*b % (M-1);  
  }  
  return h;  
}
```

7.4.1 Direkte Verkettung

Eine Möglichkeit der Kollisionsauflösung ist die einzelnen Elemente zu verketteten.

Anstelle der gesuchten Daten enthält die Hashtabelle hier Behälter (Buckets), die alle Daten mit gleichem Hashwert aufnehmen.

Die Tabelle wird als Array implementiert. Wegen der möglichen Kollisionen werden die Daten aber nicht direkt im Array gespeichert, sondern in so genannten Behältern (Buckets), die mehrere Datenelemente mit dem gleichen Hashwert aufnehmen können. Implementieren kann man diese Buckets als Array oder durch Zeiger, die auf das jeweils folgende Element innerhalb des Behälters zeigen. Das Array selbst enthält nur noch einen Verweis auf einen Bucket, der zu diesem Hashwert gehört. Im schlimmsten Fall können Kollisionen zu einer Entartung der Hashtabelle führen, wenn wenige Buckets sehr viele Datenelemente enthalten, während andere Buckets leer bleiben. Um abschließend das genaue Suchelement zu finden, muss dann nur noch eine Suche über die Elemente in einem Bucket durchgeführt werden. Ist die Hash-tabelle ausgewogen, werden aber nur wenige Datensätze in einem Bucket sein und diese Suche benötigt nur geringen Aufwand.

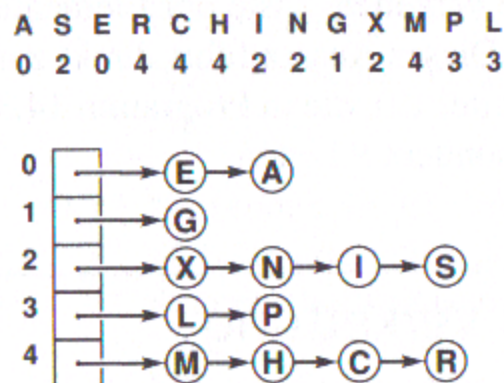


Abb. 7.25: Schema der direkten Verkettung

Algorithmus

```
private Node[] heads;  
private int N, M;  
  
ST(int maxN){ N=0; M=maxN/5; heads=new Node[M]; }  
  
ITEM search(KEY key){ return searchR(heads[hash(key, M)], key); }  
  
void insert(ITEM x){  
    int i = hash(x.key(), M);  
    heads[i] = new Node(x, heads[i]); N++;  
}
```

Screenshot

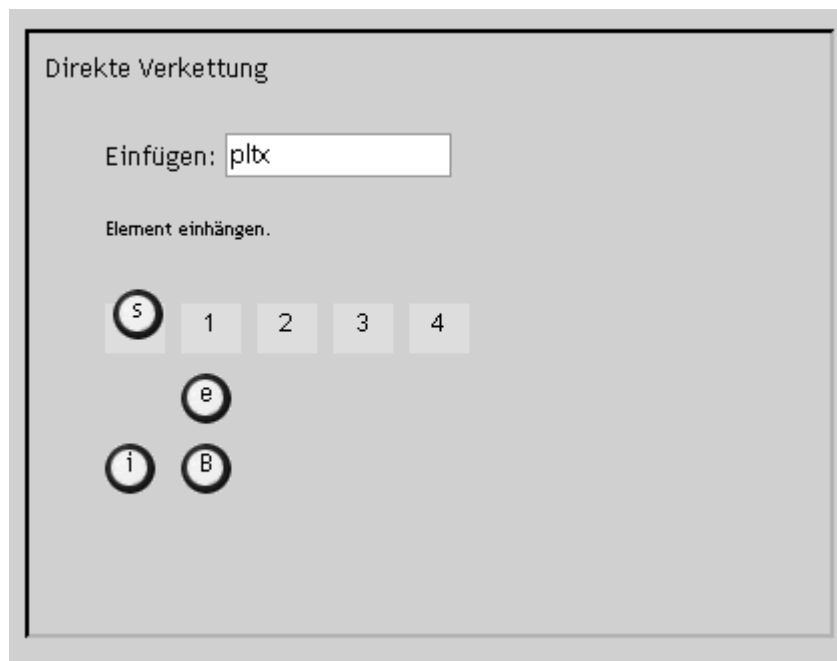


Abb. 7.26: Screenshot der interpolierten Suche im Baukasten

Analyse

Wird nach einem Element gesucht, muss dafür nur ein Bucket herangezogen werden – nämlich derjenige den die Hashfunktion ergibt. Da die Anzahl der Behälter mit m angegeben wird rechnet man durchschnittlich mit einer Laufzeit von $\Theta(n/m/2)$. Im Idealfall befindet sich das gesuchte Element an erster Stelle innerhalb des Buckets – dies entspricht einer Laufzeit von $\Omega(1)$. Der ungünstigste Fall tritt dann ein, wenn sich das gesuchte Element an letzter Stelle des Behälters befindet und aufgrund der Hashfunktion alle Elemente der gesamten Datenmenge in diesen einen Bucket gebracht wurde ($O(n)$). Dieser Fall tritt aber sehr selten auf und sollte bereits durch eine gute Hashfunktion so gut wie ausgeschlossen werden.

Zusammenfassung [SED03] Seite 656ff

	günstig	mittel	ungünstig
Laufzeit	$\Omega(1)$	$\Theta(n/m/2)$	$O(n)$
Vergleiche	1	$n/m/2$	n
Einfügen	1	1	1

7.4.2 Lineares Sondieren

Eine weitere Möglichkeit die Elemente im Kollisionsfall abzuspeichern besteht darin, so lange den jeweils nächsten Behälter zu prüfen, bis man auf einen freien Behälter trifft. Der Bereich (respektive alle Elemente innerhalb dieses Bereichs) zwischen zwei leeren Buckets wird als Cluster bezeichnet.

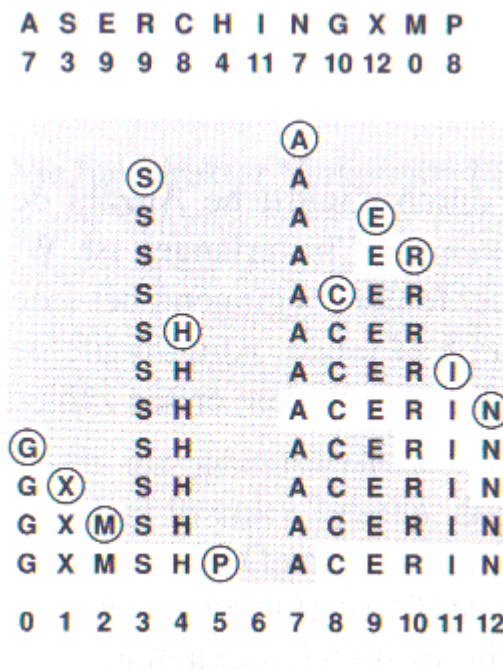


Abb. 7.27: Schema des linearen Sondierens

Wurde der letzte Behälter geprüft, so beginnt man wieder beim ersten Behälter. Wird nach einem Element gesucht, so wird wie beim Einfügen der Hashwert ermittelt. Befindet sich das gesuchte Element nicht an dieser Stelle, so wird solange im folgenden Bucket gesucht bis das Element gefunden wird oder ein leerer Behälter erreicht wird. In diesem Fall befindet sich das gesuchte Element nicht in der Hash-tabelle. Aus diesem Grund ist es wichtig, dass es mehr Buckets gibt als Elemente, da

es sonst zu einer Endlosschleife kommen würde. Zusätzliches Problem dieser Methode ist, dass sich schnell Ketten oder Cluster bilden und die Zugriffszeiten im Bereich solcher Ketten schnell ansteigen. Das lineare Sondieren ist daher wenig effizient.

Algorithmus

```
private ITEM[] st;
private int N, M;

ST(int maxN){ N=0; M=2*maxN; st=new ITEM[M]; }
ITEM search(KEY key){
    int i = hash(key, M);
    while (st[i]!=null){
        if (equals(key, st[i].key())) return st[i];
        else i=(i+1)%M;
    }
    return null;
}

void insert(ITEM x){
    int i = hash(x.key(), M);
    while (st[i] != null) i=(i+1)%M;
    st[i] = x; N++;
}
```



Wird ein Element in der Hashtabelle gesucht, so findet man es im Idealfall im ersten Bucket den man, gemäß der Hashfunktion, untersucht. Dies entspricht also einer Laufzeit von $\Omega(1)$. Der ungünstigste Fall tritt dann ein, wenn sich alle Elemente in der Hashtabelle innerhalb eines einzigen Clusters befinden. Wenn sich das gesuchte Element an der letzten Stelle des Clusters befindet, die Suche aber aufgrund des Hashwerts an der ersten Position beginnt, müssen alle Elemente verglichen werden ($O(n)$).

Seite 90 von 108



Abb. 7.29: Donald Ervin Knuth

Zusammenfassung [SED03] Seite 661ff

	günstig	mittel	ungünstig
Laufzeit	$\Omega(1)$	$\Theta(\frac{1}{2}(1+1/(1-\alpha)))$	$O(n)$
Vergleiche	1	$\frac{1}{2}(1+1/(1-\alpha))$	n
Einfügen	1	$\frac{1}{2}(1+1/(1-\alpha))$	n

7.4.3 Doppeltes Hashing

Beim Doppelten Hashing werden zwei unabhängige Hash-Funktionen h und h' angewandt. Diese heißen unabhängig, wenn die Wahrscheinlichkeit für eine so genannte Doppelkollision, d.h. $h(x)=h(y) \wedge h'(x)=h'(y)$ minimal ist. [BRE01]

A	S	E	R	C	H	I	N	G	X	M	P	L
7	3	9	9	8	4	11	7	10	12	0	8	6
1	3	1	5	5	5	3	3	2	3	5	4	2

0	1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	---	----	----	----

Abb. 7.30: Schema des doppelten Hashings

Zunächst werden die einzelnen Elemente, wie bereits durch das lineare Sondieren bekannt, anhand einer Hashfunktion abgespeichert. Kommt es dabei zu Kollisionen, wird eine zweite Hashfunktion hinzugezogen. Nun wird nicht wie beim Linearen Sondieren jeweils um eine Position, solange bis eine frei ist, weitergegangen, sondern um den Wert der zweiten Hashfunktion.

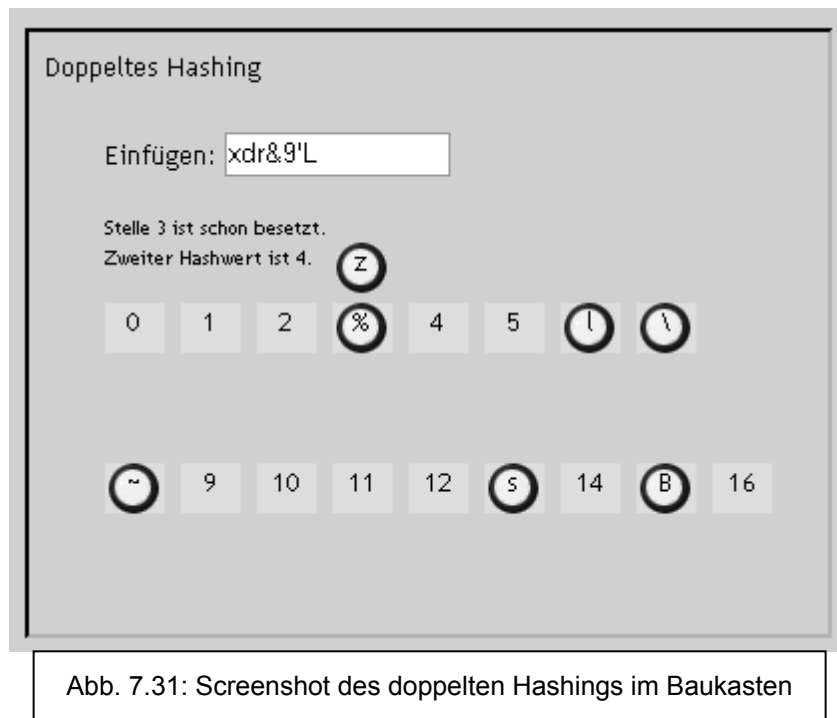
Die sowohl für den Baukasten als auch im Beispiel in Abbildung 7.30 verwendete zweite Hashfunktion lautet:

```
static int hashtwo(int v) { return (v % 97) + 1; }
```

Algorithmus

```
private ITEM[] st;  
private int N, M;  
  
ST(int maxN){ N=0; M=2*maxN; st=new ITEM[M]; }  
  
ITEM search(KEY key){  
    int i = hash(key, M); int k=hashtwo(key);  
    while (st[i]!=null){  
        if (equals(key, st[i].key())) return st[i];  
        else i=(i+k)%M;  
    }  
    return null;  
}  
  
void insert(ITEM x){  
    KEY key = x.key();  
    int i = hash(x.key(), M); int k=hashtwo(key);  
    while (st[i] != null) i=(i+k)%M;  
    st[i] = x; N++;  
}
```

Screenshot



Analyse

Genau wie beim im Abschnitt 7.4.2 beschriebenen linearen Sondieren belaufen sich die günstigsten und ungünstigsten Fälle auf $\Omega(1)$ bzw. $O(n)$.

Der durchschnittliche Fall beläuft sich auf $\Theta(1/\alpha \ln(1/(1-\alpha)))$, wobei α das Verhältnis aus der Anzahl der verfügbaren Buckets und der Anzahl der eingefügten Elemente darstellt.

Zusammenfassung [SED03] Seite 665ff

	günstig	mittel	ungünstig
Laufzeit	$\Omega(1)$	$\Theta(1/\alpha \ln (1/(1-\alpha)))$	$O(n)$
Vergleiche	1	$1/\alpha \ln (1/(1-\alpha))$	n
Einfügen	1	$1/\alpha \ln (1/(1-\alpha))$	n

7.5 Binärbaum

Ein Binärbaum ist in der Graphentheorie ein gewurzelter Baum. Dies bedeutet, dass es genau einen ausgezeichneten Knoten, die so genannte Wurzel gibt. Jeder andere Knoten ist durch genau einen Pfad von dieser aus erreichbar, wobei jeder Knoten des gesamten Baums höchstens zwei Kindknoten besitzen darf. Es wird verlangt, dass sich die Kindknoten eindeutig in linkes und rechtes Kind einteilen lassen. Dies bedeutet, dass ein linker Kindknoten dem jeweiligen Sortiervfahren entsprechend, kleiner sein muss als dessen Wurzel- bzw. Elternknoten. Ein rechter Kindknoten muss dementsprechend größer sein.

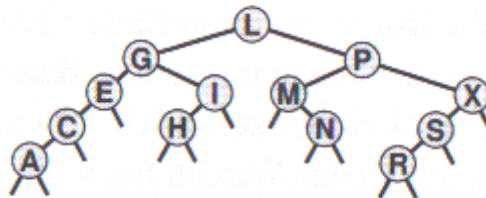


Abb. 7.32: Binärbaum

7.5.1 Suchen

Durchsucht werden die Knoten, unabhängig davon ob der Baum explizit oder implizit (während der Suche generiert) ist. Dabei wird folgendes Prinzip angewendet: Ein Knoten wird aus einer Datenstruktur entnommen. Seine Kindknoten werden untersucht und gegebenenfalls der Datenstruktur hinzugefügt. Je nach Auswahl der Datenstruktur kann der Baum in verschiedenen Reihenfolgen durchsucht werden. Die Verwendung einer Warteschlange führt so zu einer Breitensuche, bei der der Baum Ebene für Ebene durchlaufen wird.

Schema der Breitensuche [10]

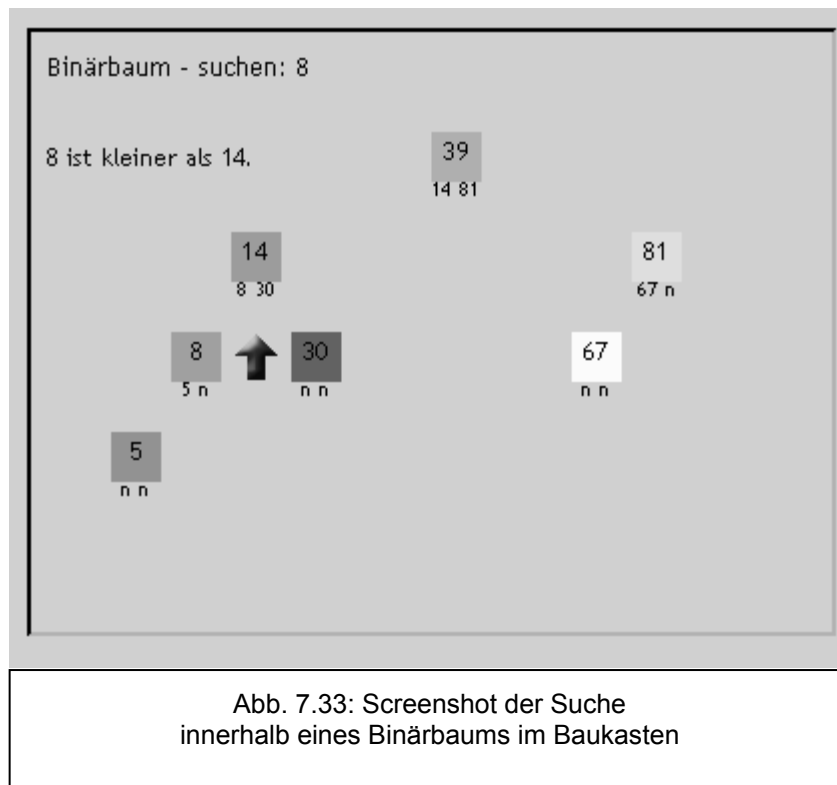
```
enqueue(Wurzelknoten);  
<markiere Wurzelknoten als markiert>  
while(<Schlange nicht leer>) {  
    Knoten=dequeue();  
    <markiere Knoten>  
    if (<Knoten.linker_Nachfolger vorhanden>)  
        enqueue(Knoten.linker_Nachfolger);  
    if (<Knoten.rechter_Nachfolger vorhanden>)  
        enqueue(Knoten.rechter_Nachfolger);  
}
```

Bei Verwendung eines Stacks hingegen, wird jeweils bis zu einem Blatt gesucht und erst anschließend mit dem nächsten Kindknoten fortgefahren. Dies wird als Tiefensuche bezeichnet.

Schema der Tiefensuche [11]

```
markiere_als_besucht(<dieser Knoten>);  
while(<unbesuchte Nachfolger vorhanden>) {  
    tiefensuche(<1. unbesuchter Nachfolger>);  
}
```

Screenshot



Algorithmus

```

public void baumsearch(int suchzahl,knoten k){
    if(suchzahl==k.zahl)
        do_something(); //Element gefunden
    else{
        if(suchzahl>k.zahl)
            if (k.rechts!=null) baumsearch(suchzahl,k.rechts);
        if(suchzahl<k.zahl)
            if (k.links!=null) baumsearch(suchzahl,k.links);
        }
    }
}

```

7.5.2 Traversieren

Es gibt verschiedene Möglichkeiten die Knoten von Binärbäumen zu durchlaufen, also zu traversieren. Diesen Prozess nennt man auch Linearisieren. Man unterscheidet hier:

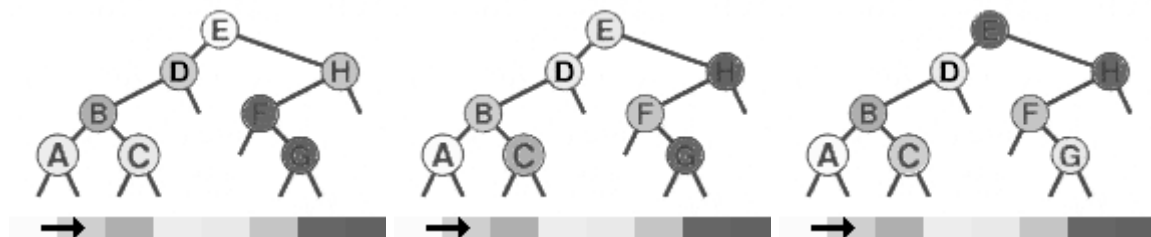


Abb. 7.34: Schematische Darstellung von Preorder (links), Inorder (mitte) und Postorder (rechts)

7.5.2.1 Preorder (W - L - R)

Zuerst wird die Wurzel (W) betrachtet und anschließend zuerst der linke (L), dann der rechte (R) Teilbaum durchlaufen.

In Abbildung 7.34 ergibt dies: E-D-B-A-C-H-F-G

Algorithmus

```
public void traverse( knoten k, int order ){
    if ( k != null ){
        bearbeite( k );
        traverse( k.links );
        traverse( k.rechts );
    }
}
```

7.5.2.2 Inorder (L - W - R)

Zuerst wird der linke (L) Teilbaum durchlaufen, dann die Wurzel (W) betrachtet und anschließend der rechte (R) Teilbaum durchlaufen.

In Abbildung 7.34 ergibt dies: A-B-C-D-E-F-G-H

Algorithmus

```
public void traverse( knoten k, int order ){  
    if ( k != null ){  
        traverse( k.links );  
        bearbeite( k );  
        traverse( k.rechts );  
    }  
}
```

7.5.2.3 Postorder (L - R - W)

Zuerst wird der linke (L), dann der rechte (R) Teilbaum durchlaufen und anschließend die Wurzel (W) betrachtet.

In Abbildung 7.34 ergibt dies: A-C-B-D-G-F-H-E

Algorithmus

```
public void traverse( knoten k, int order ){  
    if ( k != null ){  
        traverse( k.links );  
        traverse( k.rechts );  
        bearbeite( k );  
    }  
}
```

Anhang A: Abkürzungsverzeichnis

ACM	Association for Computing Machinery
CPU	Central Processing Unit – Zentraleinheit, Prozessor
CSS.....	Cascading Style Sheets
DIV	Division = Bereich
DSW.....	Deutsches Studentenwerk
E-Learning.....	Electronic Learning
E-Mail	Electronic Mail
HTML	Hypertext Markup Language
ID	Identifikationsbezeichnung
iFrame	Inline Frame = Rahmen im Textfluss
px	Pixel
src	Source
WBT	Web-based Teaching
WWW	World Wide Web

Anhang B: Literaturverzeichnis

- [BAC1894] Bachmann, Paul, Die analytische Zahlentheorie, Teubner, Leipzig, 1894
- [BRE01] Breutmann Bernd, Data and Algorithms, Fachbuchverlag Leipzig, 2001
- [LAN02] Lang Hans Werner, Algorithmen, Oldenbourg Verlag, München, 2002
- [MIT04] Mitchell Alice, Savill-Smith Carroll, The use of computer and video games for learning, Ultralab, Cambridge, 2004
- [PAP65] Papernov A., Stasevich G., A Method of Information Sorting in Computer Memories. Problems of Information Transmission 1, 63-75 (1965)
- [PRA79] Pratt, Vaughan R., Shellsort and Sorting Networks. Garland, New York (1979)
- [RIZ03] Rizek-Pfister Cornelia, Digitaler Campus, Waxmann, München, 2003
- [SED03] Sedgewick Robert, Algorithmen in Java Teil 1-4, Addison-Wesley, München 2003
- [SHE59] Shell, Donald Lewis, A high-speed sorting procedure. Communications of the ACM 2 (7), 30-32 (1959)

Anhang C: Internetquellen

- [1] <http://www.his.de/Abt2/Foerderung/hb.soz17>, 20/07/05
- [2] http://de.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Chwarizmi, 10/07/05
- [3] <http://www.turing-maschine.de>, 12/07/05
- [4] <http://www.fembio.org/frauen-biographie/ada-lovelace.shtml>, 10/07/05
- [5] <http://www.presetext.at/pte.mc?pte=040211032>, 11/07/05
- [6] http://gonline.univie.ac.at/index_m3.php?lid=1&sid=2702, 11/07/05
- [7] <http://www.sortieralgorithmen.de>, 24/06/05
- [8] <http://www.linux-related.de/index.html?/coding/o-notation.htm>, 09/08/05
- [9] <http://de.wikipedia.org/wiki/Interpolationssuche>, 02/08/05
- [10] http://tud-pc.informatik.tu-darmstadt.de/archive/icpc_seminar/web/algorithmen/breitensuche/, 27/08/05
- [11] http://tud-pc.informatik.tu-darmstadt.de/archive/icpc_seminar/web/algorithmen/tiefensuche/, 27/08/05

Anhang D: Internetrecherche

- <http://www.docjava.dk/datastrukturer/sortering/sortering.htm>, 24/06/05
- http://www.cs.bris.ac.uk/home/sw1653/des_word.html, 10/07/05
- <http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/pictures.html>, 10/07/05
- <http://www.cs.cmu.edu>, 26/06/05
- <http://www.jdl.ac.cn>, 26/06/05
- <http://www-groups.dcs.st-and.ac.uk>, 09/08/05
- <http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Knuth.html>, 27/08/05

Anhang E: Abbildungsverzeichnis

- 3.1: Studienaufwand nach Art des Studiums
Quelle: 17. Sozialerhebung des Deutschen Studentenwerks (DSW), Seite 253
- 4.1: Muhammad ibn Musa al-Chwarizmi
Quelle: <http://inet.lasierra.edu>
- 4.2: Alan Turing
Quelle: <http://fusionanomaly.net/alanturing.html>
- 4.3: Lady Augusta Ada Byron, Countess of Lovelace (1815-1852)
Quelle: <http://www.cs.kuleuven.ac.be/~dirk/ada-belgium/pictures.html>
- 6.1: Der Algorithmenbaukasten
- 6.2: genaue Ansicht des eingebetteten Frames
- 6.3: Darstellungsoptionen im Bereich Sortieren
- 6.4: Auswahl der Algorithmenanzeige
- 6.5: Sortiervergleich im Algorithmenbaukasten
- 6.6: Suchvergleich im Algorithmenbaukasten
- 6.7: Möglichkeit eigene Zahlen einzufügen
- 6.8: Auswahlmöglichkeiten beim Sortiervergleich
- 6.9: CSS-Änderung der Farbe für ein Element
- 7.1: Paul Gustav Heinrich Bachmann
Quelle: <http://www-groups.dcs.st-and.ac.uk/~history/PictDisplay/Bachmann.html>
- 7.2: Edmund Georg Hermann Landau
Quelle: <http://www-groups.dcs.st-and.ac.uk/~history/PictDisplay/Landau.html>
- 7.3: Schema des Bubblesorts
Quelle: „Data and Algorithms“, Seite 43
- 7.4: Screenshot des Bubblesorts im Baukasten
- 7.5: Schema Insertionsort
Quelle: Data and Algorithms, Seite 38
- 7.6: Screenshot des Insertionsorts im Baukasten
- 7.7: Tony Hoare
Quelle: <http://www.cs.cmu.edu>
- 7.8: Schema des Quicksorts
Quelle: Data and Algorithms, Seite 46

- 7.9: Screenshot des Quicksorts im Baukasten
- 7.10: John von Neumann
Quelle: <http://www.cs.cmu.edu>
- 7.11: Screenshot des Mergesorts im Baukasten
- 7.12: Schema des Shellsort anhand des Strings „ASORTINGEXAMPLE“
Quelle: „Algorithmen in Java“, Seite 320
- 7.13: Zahlenfolge von Vaughan Pratt, 1971
Quelle: Algorithmen in Java, Seite 325
- 7.14: Screenshot des Shellsort im Baukasten
- 7.15: Schema des Bucketsorts
Quelle: <http://www.docjava.dk/datastrukturer/sortering/sortering.htm>
- 7.16: Screenshot des Bucketsort im Baukasten
- 7.17: Robert W. Floyd
Quelle: <http://www.jdl.ac.cn>
- 7.18: Darstellung eines Heap-geordneten Binärbaums als Array
Quelle: „Algorithmen in Java“, S. 404
- 7.19: Screenshot des Heapsort im Baukasten
- 7.20: Schema der linearen Suche
Quelle: http://www.cs.bris.ac.uk/home/sw1653/des_word.html
- 7.21: Screenshot der linearen Suche im Baukasten
- 7.22: Schema der binären Suche
Quelle: http://www.cs.bris.ac.uk/home/sw1653/des_word.html
- 7.23: Screenshot der binären Suche im Baukasten
- 7.24: Screenshot der interpolierten Suche im Baukasten
- 7.25: Schema der direkten Verkettung
Quelle: „Algorithmen in Java“, S. 656
- 7.26: Screenshot der interpolierten Suche im Baukasten
- 7.27: Schema des linearen Sondierens
Quelle: „Algorithmen in Java“, S. 661
- 7.28: Screenshot des linearen Sondierens im Baukasten
- 7.29: Donald Ervin Knuth
Quelle: <http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Knuth.html>
- 7.30: Schema des doppelten Hashings
Quelle: „Algorithmen in Java“, S. 667
- 7.31: Screenshot des doppelten Hashings im Baukasten

7.32: Binärbaum

Quelle: „Algorithmen in Java“, S. 598

7.33: Screenshot der Suche innerhalb eines Binärbaums im Baukasten

7.34: Schematische Darstellung von Preorder (links), Inorder (mitte) und Postorder (rechts)

Quelle: „Algorithmen in Java“, S. 261

Anhang F: Stichwortverzeichnis

A

Ada · 15
al-Chwarizmi, Muhammad ibn Musa · 12
Algorithmenanalyse · 16

B

Bachmann, Paul · 44
Binärbaum · 22, 24, 96
Binäre Suche · 22, 77
Blended Learning · 5, 7, 9, 17, 19
Breitensuche · 96
Bubblesort · 7, 21, 22, 25, 40, 48, 49
Bucketssort · 22, 25, 66, 67, 68

C

C++ · 28
CSS · 23, 37, 42

D

Direkte Verkettung · 22, 85
DIV-Element · 34, 37, 38
Doppeltes Hashing · 22, 92

E

Elternknoten · 96

F

Floyd, Robert · 70

H

Hashing · 7, 22, 24, 44, 83
Heap · 70, 71
Heapsort · 22, 25, 60, 70
Hoare, Tony · 55
HTML · 23, 26, 32, 36, 39, 42

I

iFrame · 26
Inorder · 22, 100
Insertionsort · 22, 25, 52, 57, 61
Interpolierte Suche · 22, 80

J

Java · 28
Javascript · 23, 26, 27, 28, 32, 35, 36

K

Kindknoten · 96
Knuth, Donald · 90

L

Lady Augusta Ada Byron · *Siehe Ada*
Landau, Edmund · 44
Lineare Suche · 22, 74
Lineares Sondieren · 22, 88

M

Mergesort · 22, 25, 58, 60

P

Papernov, A. · 64
Postorder · 22, 100
Präsenzunterricht · 17
Pratt, Vaughan · 64
Preorder · 22, 99
Pseudocode · 28

Q

Quicksort · 22, 25, 55, 57, 58

S

sequenzielle Suche · 74
Shell, Donald · 61
Shellsort · 22, 25, 61, 62, 63
Stasevic, G. · 64

T

Tiefensuche · 97
Timeout · 36
Traversieren · 7, 22, 99
Turing, Alan · 13
Turingmaschine · 13, 16

V

von Neumann, John · 58

W

William, John · 70
Wurzel · 71, 99, 100

Θ

Θ -Notation · 46

O

O-Notation · 44

Ω

Ω -Notation · 46