Datenstrukturen und Algorithmen

Unterlagen zur Lehrveranstaltung

Johann Mitlöhner

2006–2011

1 Der Begriff des Algorithmus

“Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.”
Ludwig Wittgenstein, 1922

Unser Gehirn begünstigt die Entstehung einer bestimmten Art von Sprachen, und damit die Konzepte, die sich damit ausdrücken lassen und über die wir nachdenken können. In ähnlicher Weise begünstigt die heute verbreitete Art des digitalen Computers bestimmte Arten von Algorithmen und damit eng zusammenhängend bestimmte Arten von Datenstrukturen. Hätten wir ganz andere Arten von Rechenmaschinen zur Verfügung, dann könnten wir auch ganz andere Arten von Algorithmen einsetzen. Bevor wir daher über Algorithmen nachdenken, wollen wir zunächst festhalten, auf welchen Rechnern diese laufen sollen.

1.1 Unser Rechnermodell

Aus Algorithmensicht interessieren uns im Rechner nur drei Komponenten: CPU, Hauptspeicher, und Festplatte.

1. Die **CPU** (central processing unit) arbeitet Programme schrittweise ab, z.B. mit einer Taktrate von etwa 3 GHz. Schaltkreise reagieren auf Bitzustände, sodaß Anweisungen in Binärcode ausgeführt werden, d.h. Zustände in der CPU und in anderen Komponenten werden geändert. Die CPU enthält dazu einige wenige Speicherplätze, auf die in Bruchteilen von Nanosekunden zugegriffen wird. Dort finden wir neben der gerade ausgeführten Programmanweisung und der Adresse der nächsten Anweisung noch einige andere Werte, z.B. eine eben berechnete Gleitkommazahl sowie einige Speicheradressen und Zählervariablen.
   Die meisten Desktoprechner und Server enthalten heute mehrere Cores bzw. CPUs, die typischerweise vor allem zur Lastverteilung genutzt werden, indem die Programme auf die vorhandenen Prozessoren aufgeteilt werden. Jedes einzelne Programm läuft auch hier wieder (eine zeitlang) auf einem Prozessor; Software, die Parallelverarbeitung benutzt, bei der ein Algorithmus mehrere Handlungsstränge gleichzeitig verfolgt, ist

wesentlich schwieriger zu entwickeln. Für unsere Überlegungen werden wir von einer CPU ausgehen, d.h. die einzelnen Schritte werden nacheinander ausgeführt und nicht parallel.

2. Der **Hauptspeicher** (RAM, random access memory) erlaubt uns Zugriff auf beliebige Adressen, die wir typischerweise als Byte (8 Bit) lesen und schreiben können. Ein Zugriff dauert einige Nanosekunden; das ergibt sich aus der Frequenz des Front Side Bus, der die CPU mit dem RAM verbindet (z.B. 533 oder 1066 MHz). Hauptspeicher ist nur in engen Grenzen erweiterbar: wir finden auf einem typischen Mainboard z.B. 4 GB RAM. [1]
3. Eine typische **Festplatte** gibt uns z.B. 500 GB Speicherplatz, also etwa 100 mal soviel wie der Hauptspeicher. Auch hier können wir auf beliebige Adressen direkt zugreifen; die Blockgröße ist höher, z.B. 1 KB oder vielfache davon. Auf verschiedene Arten (wie USB, SCSI, und über LAN) können wir eine große Zahl weiterer Festplatten an unseren Rechner hängen. Dieses Speichermedium ist also in der Kapazität viel weniger begrenzt als der Hauptspeicher; durch die mechanische Bewegung des Kopfes dauert ein Zugriff auf einen einzelnen Block aber einige Millisekunden, das ist um den Faktor 1.000.000 langsamer als der Hauptspeicher.[2]

Wir können also vereinfachend festhalten, daß unser Rechner eine Anweisung nach der anderen abarbeitet, und im wesentlichen über zwei Arten von Speicher verfügt: einen relativ schnellen, aber nicht sehr großen, sowie einen relativ langsamen, der aber fast unbegrenzt ist. Solange unser Problem als ganzes in den schnellen Hauptspeicher paßt, können wir oft auch mit sehr einfachen Algorithmen auskommen. Da aber viele Probleme dafür zu groß werden, müssen wir Verfahren finden, die die Anzahl der Zugriffe minimieren. Solche Verfahren wollen wir hier betrachten.
Als Vorbereitung auf den Algorithmusbegriff wollen wir zunächst als Beispiel das Rechnen im römischen Zahlensystem betrachten, das als Additionssystem im wesentlichen ein erweiterter Strichcode ist und einen guten Vergleich mit dem Dezimalsystem gestattet.

1.2 Rechnen im römischen Zahlensystem

Das römische Zahlensystem ist sehr intuitiv und für kleine Zahlen sehr einfach zu verwenden. Es ist ein Additionssystem d.h. der Wert einer Zahl errechnet sich aus der Summe der Ziffern. Bei großen Werten und komplizierteren Operationen wird es aber unbrauchbar:

1. Zählen: Striche nebeneinander schreiben und vereinfachen
   I, II, III, IIII=IV, V, VI, ...
2. Addieren: nebeneinander schreiben und vereinfachen
   CXVI + XXIV = CXVI + XXIIII = CXVIXXIIII = CXXXVIIIII = CXXXX = CXL

[1] Daneben gibt es meist Level 1 und 2 Cache, die in der CPU integriert oder mit ihr über einen schnelleren Bus verbunden sind; doch durch ihre geringe Größe im Vergleich zum restlichen RAM beeinflußen sie unsere Überlegungen nicht wesentlich.

[2] Beim Lesen oder Schreiben von vielen benachbarten Blöcken sinkt der Faktor deutlich, aber nicht unter ca. 1.000.

3. Subtrahieren: wegstreichen und vereinfachen
   CXVI - XXIV = C**XVI** - **XXI**III = CV - XIII = LLIIIII - XIII = LXXXX**XIII**II - XIII = LXXXXII = XCII
4. Multiplizieren: wiederholtes Addieren
   für a * b = c wiederhole c = c + a, b = b - 1 bis b = 0 d.h. leer
5. Dividieren: wiederholtes Subtrahieren
   für a / b = c wiederhole a = a - b, c = c + 1, bis a < b; a ist dann der Rest

Die Verfahren für Multiplizieren und Dividieren sind so mühsam, daß sie für große Zahlen praktisch nicht mehr verwendbar sind. Trotzdem wurden in Europa noch lange nach dem Untergang des römischen Reiches mit den römischen Zahlen gerechnet.

1.3 Vom Binärsystem zu Logik und Elektronik

Digitale Computer arbeiten mit dem binären Zahlensystem. Rechenoperationen können daher mit Hilfe von logischen Funktionen dargestellt werden. Als technische Lösung werden elektronische Schaltungen verwendet. Die Zustände Null und Eins werden dabei als elektrische Spannungsniveaus realisiert.
Als Beispiel soll die Addition von zwei Bits A und B dienen: wie im Dezimalsystem entsteht in bestimmten Situationen ein Übertrag, hier wenn beide Eingabewerte 1 sind. Wir brauchen also logische Funktionen für die Summe S und den Übertrag C, die in Abhängigkeit von den Eingabewerten A und B die richtigen Resultate für S und C liefern:

A	B	S	C	A and B	A or B	A Xor B
0	0	0	0	0	0	0
0	1	1	0	0	1	1
1	0	1	0	0	1	1
1	1	0	1	1	1	0

Die logische Funktion, die uns das Ergebnis für C liefert, ist AND, die Funktion für S ist XOR (eXclusive OR). Aber wie können logische Funktionen implementiert werden, d.h. wie ist es möglich, daß aus Eingabewerten 0 und 1 die Ergebnisse von logischen Funktionen maschinell berechnet werden?
Zur Illustration sollen einfache elektrische Schaltungen mit Batterie, Lampe und Taster dienen. Taster Drücken entspricht 1; Taster nicht gedrückt entspricht 0. Taster sind daher die Eingabeelemente, Lampen übernehmen die Ausgabe. Computer arbeiten zwar nicht mit Tastern und Lampen, aber die Umsetzung mit Transistoren und anderen elektronischen Bauelementen in integrierten Schaltungen ist im Prinzip recht ähnlich.
In Abb. 1 sind die Implementierungen der logischen Funktionen AND, OR und NOT zusammen mit ihren schematischen Darstellungen gezeigt.
In der ersten Schaltung leuchtet die Lampe nur, wenn beide Taster gedrückt sind. Das entspricht der Funktion AND.
In der zweiten Schaltung ist die Funktion OR implementiert: sobald einer der beiden Taster gedrückt ist, leuchtet die Lampe.

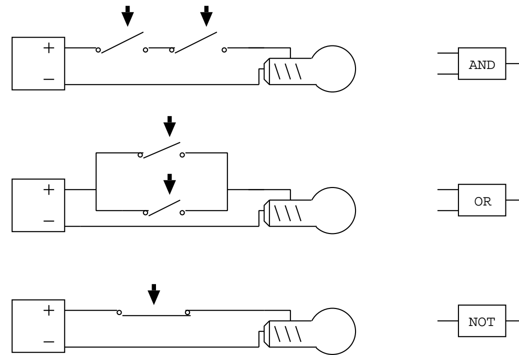
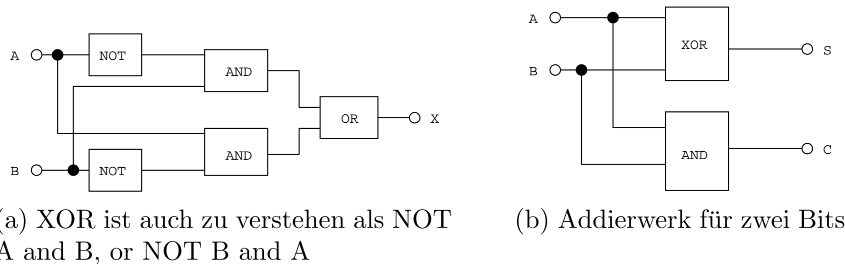


Abbildung 1: Implementierung logischer Funktionen mit elektrischen Schaltungen

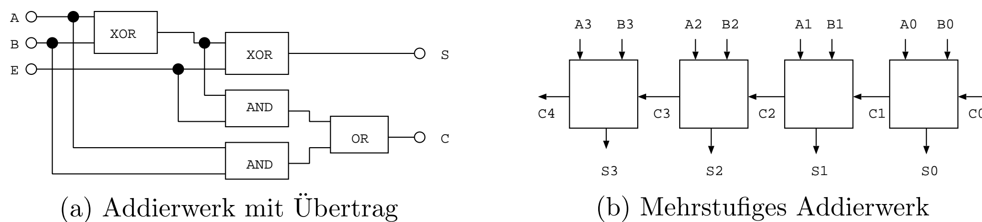


(a) XOR ist auch zu verstehen als NOT A and B, or NOT B and A

(b) Addierwerk für zwei Bits

Abbildung 2: XOR und einfaches Addierwerk ohne eingehendem Übertrag

In der dritten Schaltung arbeitet der Taster als Unterbrecher – die Lampe geht aus, wenn wir den Taster drücken. Das entspricht der Funktion NOT.
Wenn wir statt der Lampe einen elektrisch gesteuerten Taster einsetzen, können wir das Ergebnis eines Schaltkreises als Eingabe für weitere Schaltkreise verwenden. Damit können komplexere logische Funktionen wie NAND (AND gefolgt von NOT) und (mit etwas mehr Aufwand) auch XOR implementiert werden; siehe Abb. 2a. Nun haben wir die nötigen Bausteine für unser einfaches Addierwerk; es ist in Abb. 2b schematisch dargestellt. Leider berücksichtigt es keinen eingehenden Übertrag, ist daher von geringer praktischer Relevanz.



(a) Addierwerk mit Übertrag

(b) Mehrstufiges Addierwerk

Abbildung 3: Addierwerke können als “ripple adder” zusammengesetzt werden

Die etwas aufwendigere Schaltungen in Abb. 3a kann auch einen eingehenden Übertrag berücksichtigen und damit als Baustein von größeren Addierwerken dienen, wie in Abb. 3b dargestellt. In einem solchen “ripple adder” wird der Übertrag in jedem Verarbeitungsschritt

an die nächste Gruppe weitergegeben.
Generell ist die Verarbeitung in der CPU umso schneller, je höher die Taktfrequenz ist; allerdings wird die Kühlung bei hohen Taktraten immer schwieriger. CPUs stossen deshalb bei etwa 3 – 4 GHz an eine Grenze, was mit ein Grund für die Entwicklung von Multicore-Systemen ist. Die Taktfrequenz kann nicht mehr wesentlich erhöht werden, wohl aber die Anzahl der Cores. Die damit entstehenden softwaretechnischen Probleme der Parallelverarbeitung sind aber nach wie vor eine große Hürde, die zusätzliche Komplexität und damit weitere mögliche Fehlerquellen in die ohnehin schon komplexe Softwareentwicklung bringt.

1.4 Intuitiver Algorithmusbegriff

Der arabische Gelehrte Muhammad ibn Musa al-Khwarizmi (780-850) verfaßte um 825 in Baghdad ein Werk in arabischer Sprache über das Rechnen mit indischen Ziffern[3], das im 12. Jahrhundert ins Lateinische übersetzt (Algoritmi de numero Indorum) auch in Europa Verbreitung fand. In seinem Buch gibt er genaue Anweisungen für das Rechnen im Dezimalsystem, d.h. in einem Stellenwertsystem mit der Basis 10, das auch das Konzept der Null beinhaltet, ein enormer praktischer Fortschritt gegenüber dem römischen Zahlensystem.
In Europa sollte es allerdings noch einige hundert Jahre dauern, bis das römische System abgelöst wird: 1202 beschreibt Fibonacci (Leonardo Pisano) in seinem *Liber Abaci* (Buch über das Rechnen) die 'arabischen' Ziffern. Aber erst mit den Werken von Adam Ries, z.B. "Rechnung auff der linihen vnd federn" (1522) wird das Rechnen mit arabischen Ziffern in Europa allgemein bekannt und verdrängt endgültig das Rechnen im römischen System. "auff der linihen" bezieht sich auf das Rechnen mit dem Abakus, "vnd federn" auf das Rechnen am Papier im Dezimalsystem.
Die Anweisungen für die Grundrechenarten Addieren, Subtrahieren, Multiplizieren und Dividieren im Dezimalsystem erfüllen die Bedingungen, die wir an einen "Algorithmus" stellen:
Ein Algorithmus ist:

- ein allgemeines Verfahren zur Lösung eines Problems
- in eindeutig beschriebenen einzelnen Schritten
- dessen Beschreibung endlich lang ist
- das in endlich vielen Schritten und in endlicher Zeit ausgeführt werden kann

Wenn eine unmißverständliche verbale Beschreibung verfügbar ist, so kann ein Mensch den Algorithmus ausführen; liegt die Beschreibung als Programmcode vor, so kann eine Maschine zur Ausführung verwendet werden.
Man könnte nun meinen, es gäbe für jede eindeutig formulierbare Aufgabe auch eine algorithmische Lösung, d.h. wenn die Aufgabe nur klar genug definiert ist, kann sie auch programmiert und damit gelöst werden. Wir können aber mit einem sehr eleganten Beweis zeigen, daß dem <u>nicht</u> so ist.

[3] In Indien war das Dezimalsystem bereits seit ca. 500 bekannt.

1.5 Berechenbarkeit und Halteproblem

Nicht für alle formulierbaren Probleme gibt es Algorithmen, d.h. es gibt Probleme, die nicht 'berechenbar' sind. Das Halteproblem dient oft als Beispiel für ein solches Problem:

1. Annahme: es gibt ein Programm Q, das folgendes leistet:
   Input: beliebiges Programm P und dessen Input X
   Output: Feststellung (in endlicher Zeit!), ob P mit X hält oder nicht
   Frage: Ist ein solches Programm Q möglich?

2. Wir konstruieren ein Programm S mit Input P, das Q verwendet:
   wenn Q(P,P) = Ja, dann geht S in eine Endlosschleife;
   wenn Q(P,P) = Nein, dann terminiert S.

3. Nun versuchen wir S(S):
   Wenn Q(S,S) = Ja, dann heißt das S(S) müßte terminieren
   Wenn Q(S,S) = Nein, dann dürfte S(S) nicht terminieren

In beiden Fällen haben wir einen Widerspruch. Da S offenbar unproblematisch ist, kann der Fehler nur in der Annahme eines Programmes Q liegen. Wir haben bewiesen, daß es das Programm Q nicht geben kann!

1.6 Formaler Algorithmusbegriff

Der intuitive Algorithmusbegriff ist leider etwas unscharf; wann sind die Schritte "eindeutig" beschrieben? Eine mathematisch präzise Definition des Begriffs Algorithmus ist nützlich für Komplexitätsüberlegungen, sowie für die vorhin skizzierte Unentscheidbarkeit des Halteproblems und andere Überlegungen zur Grenzen der Berechenbarkeit. Es gibt eine Reihe von Ansätzen; für die Entwicklung der Informatik ist die Turingmaschine am wichtigsten.
Eine Turingmaschine ist definiert durch

- eine Menge von Zuständen
- eine Menge von Symbolen
- einen Anfangszustand
- eine Menge von Endzuständen
- eine Übergangsfunktion

Die Maschine ist zu jedem Zeitpunkt in genau einem Zustand. Zu Beginn der Verarbeitung ist sie im Anfangszustand. Gelangt sie in einen Endzustand, so ist die Verarbeitung beendet. Ein **Schreib/Lesekopf** steht über einem **Band**, sodaß er jeweils ein Bandsymbol lesen oder schreiben kann und seine Position um jeweils eine Stelle nach links oder rechts bewegt werden kann. Das funktioniert immer, denn das Band ist nach links und rechts unbegrenzt.[4]

[4] Man kann auch sagen: es kann nach Bedarf beliebig verlängert werden.

Die Übergangsfunktion gibt für Kombinationen aus aktuellem Eingabesymbol (über dem der Kopf gerade steht) und Zustand ein Ausgabesymbol (das an die aktuelle Stelle geschrieben wird), einen neuen Zustand und eine Bewegung nach rechts oder links an. Nicht alle möglichen Kombinationen müssen definiert sein; wird kein Übergang gefunden, ist die Verarbeitung beendet.
Damit haben wir ein ganz konkretes Modell für einen "Computer". Wie hängen nun die Begriffe "Algorithmus" und "Computer" zusammen? Die **Church-Turing-These** sagt: "The notion of algorithm can be made precise (in the form of computable functions) and computers can run those algorithms. Furthermore, a computer can theoretically run any algorithm; that is, all ordinary computers are equivalent to each other in terms of theoretical computational power, and it is not possible to build a calculation device that is more powerful than a computer."
Die Turingmaschine wurde von Alan Turing 1936 als Gedankenexperiment für die Grenzen der Berechenbarkeit gebraucht; sie sollte zwar technisch plausibel sein, war aber nicht als praktisch brauchbare Rechenmaschine gedacht; dennoch hat sie die weitere Entwicklung der Datenverarbeitung beeinflußt.

1.7 Die richtungsgebundene Turingmaschine

Diese Art von Turingmaschinen kann besonders einfach gezeichnet werden. Sie hat die gleiche Mächtigkeit wie die nicht richtungsgebundene Turingmaschine.

- Jeder Zustand wird nur von rechts ODER links erreicht.
- Das Band ist nur nach rechts unbegrenzt, am linken Rand steht konventionsgemäß immer das Zeichen Z_0.
- Die Eingabe wird konventionsgemäß rechts durch eine unendliche Folge von e beendet.
- Der Schreib/Lesekopf steht am Beginn der Verarbeitung an der ersten Bandposition über dem Zeichen Z_0.

Aufgaben:

1. Was ist ein Algorithmus? Geben Sie ein Beispiel!
2. Skizzieren Sie die Beweisführung der Unentscheidbarkeit des Halteproblems!
3. Wie ist eine Turingmaschine definiert und wozu dient sie?
4. Konstruieren Sie eine richtungsgebundene Turingmaschine, die feststellt, ob auf dem Band eine (beliebig große) gerade oder ungerade Binärzahl steht.
5. Konstruieren Sie eine Turingmaschine, die eine (beliebig große) Binärzahl verdoppelt.
6. Konstruieren Sie eine Turingmaschine, die eine Zahl im Strichcode halbiert. Divisionsreste sollen wegfallen. Bsp: aus |||| soll || werden, aus ||| soll | werden. Tip: siehe Turingmaschine im Skriptum Panny zum Strichcode verdoppeln.
7. Entwerfen Sie eine Turingmaschine, die eine Binärzahl auf dem Eingabeband durch Zwei dividiert (Divisionsrest fällt weg).

1.8 Dualität Algorithmus/Daten und Abstraktion

Algorithmen und Datenstrukturen stehen nicht isoliert nebeneinander, sondern gehen Hand in Hand: aus einer bestimmten Datenstruktur ergeben sich mehr oder weniger zwangsläufig bestimmte Algorithmen, und umgekehrt.
Moderne Programmiersprachen stellen Möglichkeiten zur Strukturierung von Funktionen und Daten zur Verfügung: benutzerdefinierte Funktionen und Prozeduren, benutzerdefinierte Datentypen.
Damit wird Abstraktion erleichtert, d.h. eine Modellierung, die es ermöglicht, eine bestimmte Implementierung einer Datenstruktur ohne Kenntnis der technischen Details zu verwenden. Hilfreich ist dabei ein Modulkonzept, d.h. die Kapselung von Systemteilen mit einer definierten **Schnittstelle**. Objektorientierte Sprachen wie Java unterstützen diesen Ansatz natürlich besonders.

1.9 Komplexität von Algorithmen

In Abhängigkeit von der Problemgröße verbrauchen verschiedene Algorithmen verschieden viel Resourcen. Die Problemgröße wird meist mit n bezeichnet und ist z.B. beim Sortieren die Anzahl der Elemente.

- Zeitkomplexität $T(n)$: wieviel Zeit bzw. wieviele Schritte benötigt der Algorithmus
- Speicherkomplexität $S(n)$: wieviel Speicher benötigt der Algorithmus

Beide Größen können für den average, worst und best case angegeben werden, und zwar

- exakt, z.B. $T(n) = 1 + n + n^2/2$
- in der O-Notation: es wird nicht die genaue Anzahl der Schritte angegeben, sondern nur die Größenordnung, d.h. der Term, der am schnellsten mit n wächst, im obigen Beispiel $O(n^2)$.

Meist interessiert die Zeitkomplexität im average und worst case, in O-Notation; Beispiele:

$O(1)$	konstant	Einfügen im unsortierten Array
$O(\log n)$	logarithmisch	Suchen im sortierten Binärbaum
$O(n)$	linear	Suchen im unsortierten Array
$O(n \log n)$	$n \log n$	Quicksort, Heapsort
$O(n^2)$	quadratisch	Selection sort
$O(n^c)$	polynomial	oft als Grenze für 'praktische Berechenbarkeit' genannt
$O(c^n)$	exponentiell	Travelling Salesman Problem

Je höher die Zeitkomplexität, desto kleiner der Bereich der Problemgrößen, in dem der Algorithmus noch mit zumutbaren Ausführungszeiten einsetzbar ist. Abb. 4 zeigt einige Kurvenverläufe.

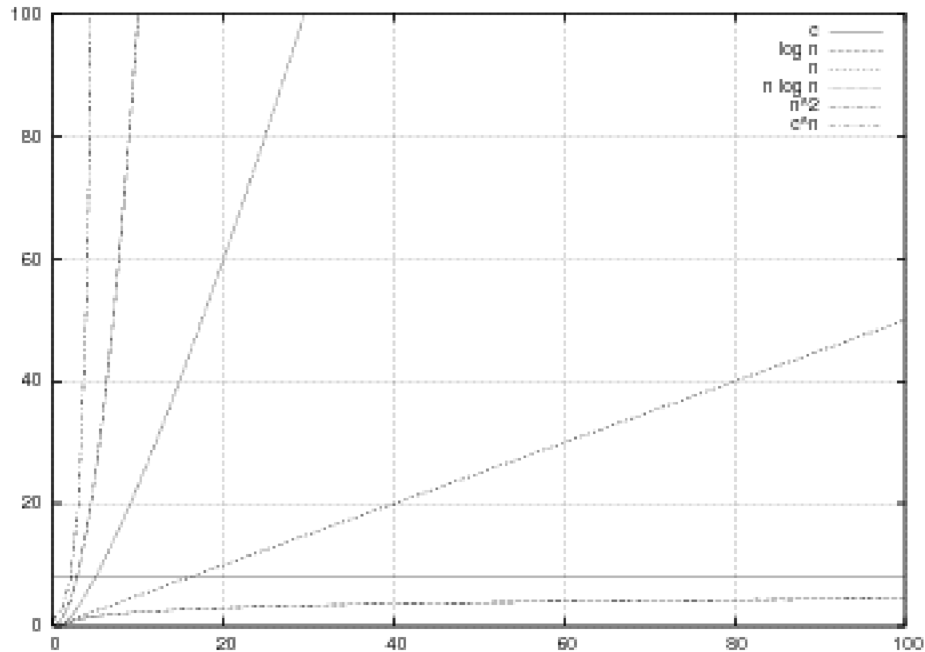


Abbildung 4: Einige Zeitkomplexitäten

1.10 Datenspeicherung und Zugriffsfunktionen

In der Folge betrachten wir verschiedene Datenstrukturen, die eine Anzahl n von Elementen (in unseren Beispielen natürliche Zahlen) speichern können. Die Operationen, deren Algorithmen und Komplexität wir untersuchen werden, sind

- Einfügen
- Suchen
- Löschen
- Sortieren

Die Datenstrukturen sind:

- Arrays
- Listen
- Sortierte Binärbäume
- Heaps
- AVL-Bäume
- Hashtables

Mit jeder Datenstruktur sind bestimmte Algorithmen und damit Komplexitäten für die Zugriffsoperationen verbunden, daher hat jede Datenstruktur Vor- und Nachteile, die für jede Anwendung abzuwägen sind. Die allgemein "beste" Datenstruktur gibt es nicht.

2 Arrays

Das Array ist eine einfache und in unserer Rechnerarchitektur grundlegende Datenstruktur: eine Reihe von Elementen sind im Speicher hintereinander angeordnet, sodaß jedes Element mittels Index zugänglich ist. Die Implementierung in einer Programmiersprache ist einfach, da fast alle Sprachen diesen Datentyp direkt zur Verfügung stellen. Wir definieren ein Array mit einer bestimmten ("wahrscheinlich genügenden") Größe. Mit dieser Datenstruktur können wir die Zugriffsfunktionen Einfügen, Löschen und Suchen implementieren:

- Einfügen: die aktuelle Anzahl der Elemente ist n; wir speichern auf der ersten freien Stelle $a[n] = x$ und setzen $n = n + 1$.
  Überschreiten wir die aktuelle Arraygröße, können wir versuchen, ein größeres Array zu definieren und alle Elemente an den neuen Platz zu kopieren.
- Suchen: wir beginnen bei der ersten Position und vergleichen, bis wir das gesuchte Element gefunden haben (erfolgreiche Suche), oder an der ersten freien Position d.h. am Ende des Arrays angekommen sind (erfolglose Suche).
- Löschen: wir müssen zuerst die Stelle i des zu löschenden Elements x suchen, dann setzen wir das letzte Element an diese Stelle und vermindern n um Eins, also $a[i] = a[n]$ und $n = n - 1$.

2.1 Zeitkomplexität der Suche

Das Array wird wegen seiner einfachen Umsetzung sehr häufig verwendet. Sowohl der Hauptspeicher als auch die meisten anderen Datenträger erlauben eine Form der Adressierung mit Index, daher ist das Array ohne zusätzlichen Aufwand zu implementieren. [5] Die Zeitkomplexität der Suche ist allerdings schlecht.
Die Problemgröße n ist die Anzahl der schon vorhandenen Elemente. Das Einfügen geht in konstanter Zeit unabhängig von n. Für die Suche nehmen wir Gleichverteilung für die Elemente an. Dann erwarten wir als Anzahl der Schritte:[6]

	best	worst	average
erfolgreiche Suche	1	n	$n/2$
erfolglose Suche	n	n	n

Für die Suche haben wir also im average case $O(n)$, was sehr unbefriedigend ist und den Einsatz des unsortierten Arrays für viele Anwendungen ausschließt.

[5] Meist wird zusätzlich zum Hauptspeicher noch Cache-Speicher verwendet, der kleiner ist, aber wesentlich schnelleren Zugriff erlaubt. Adressen werden zuerst im Cache gesucht und, falls sie dort gefunden werden (cache hit), wird der dortige Wert verwendet. Bei geschicktem Ein- und Auslagern kann eine deutliche Performancesteigerung erzielt werden. Das kann sich im Unterschied zu anderen Datenstrukturen besonders beim Array als "locality of reference" auswirken: Elemente, die in benachbarten Speicherstellen stehen, werden oft auch unmittelbar nacheinander bearbeitet, und das geht dann besonders schnell, weil sie oft auch zusammen im Cache-Speicher sind, denn Einträge im Cache (cache lines) enthalten typischerweise etwa 8 bis 512 Bytes, also mehr als ein Array-Element.

[6] Wir zählen nur die Anzahl der Vergleiche; das Hochzählen eines Index wie $i = i + 1$ geschieht um Größenordnungen schneller, sodaß wir es hier vernachlässigen können.

2.2 Binäre Suche im sortierten Array

Wir können ein Array sortiert halten, indem wir bei jedem Einfügen und Löschen die nachfolgenden Elemente um eine Position verschieben, eine sehr aufwendige Variante.
Ist allerdings das Array sortiert, so können wir mit folgendem Verfahren sehr schnell nach Elementen suchen:

- Wir beginnen die Suche an der Stelle i = n/2. Finden wir das gesuchte Element x, so beenden wir.
- Ansonsten suchen wir auf gleiche Weise in der oberen oder unteren Hälfte des Arrays, je nachdem ob x kleiner oder größer a[i] ist.

Mit dieser 'binären Suche' gelangen wir in wenigen Sprüngen zum gesuchten x bzw. der Erkenntnis, daß x nicht im Array enthalten ist.

Aufgaben:

1. Simulieren Sie dieses Verfahren für ein beliebiges Array mit min. 8 Elementen.
2. Formulieren Sie dieses Verfahren in Pseudocode.
3. Beschreiben Sie die Zeitkomplexität des Einfügens im sortierten Array mit der Verschiebemethode.
4. Beschreiben Sie die Zeitkomplexität der binären Suche.

3 Sortierverfahren

Im folgenden betrachten wir Verfahren zum aufsteigenden Sortieren von Elementen in einem Array. Diese Aufgabe tritt sehr häufig auf, sodaß eine große Zahl von Verfahren entwickelt wurden.
Wir gehen bei den Überlegungen zur Zeitkomplexität davon aus, daß die Schritte des Sortierverfahrens sequentiell abgearbeitet werden müssen, d.h. unser Programm wird (auch in einem Mehrprozessor-System) zu jedem Zeitpunkt immer nur von einer einzigen CPU ausgeführt.
Wir unterscheiden zwischen direkten Verfahren (selection sort, insertion sort, bubble sort), die eine avg case Zeitkomplexität von $O(n^2)$ haben, und den höheren Verfahren (quick sort, heap sort), die $O(n \log n)$ erreichen. Es ist bewiesen, daß eine weitere Verbesserung für diese Art von Sortieren nicht möglich ist.

3.1 Sortieren durch Auswahl (selection sort)

Wir bauen im linken Teil des Arrays die schon sortierten Elemente auf. Dazu suchen wir in jedem Durchgang das kleinste Elemente im rechten Bereich und tauschen es mit dem aktuellen Element im linken Bereich.

- Wir suchen das kleinste Element und setzen es (durch Tauschen mit dem dortigen Element) an die Stelle 1. $a[1]$ ist nun sortiert.

- Wir suchen in den verbleibenden $n-1$ Elementen wieder das kleinste Element und tauschen mit $a[2]$. Damit ist $a[1..2]$ nun auch sortiert usw.

Aufgaben:

1. Sortieren Sie eine 'zufällige' Folge von min. 8 Zahlen mit diesem Verfahren.
2. Die Zeitkomplexitäten im worst, best und avg case sind $O(n^2)$. Warum?
3. Formulieren Sie diesen Algorithmus in Pseudocode!
4. Formulieren Sie eine Erweiterung, um die best case Komplexität auf $O(n)$ zu bringen!
5. Welche Stärken und Schwächen hat dieses Verfahren?

3.2 Sortieren durch Einfügen (insertion sort)

Wir fügen die Elemente des unsortierten Arrays a in ein neues, sortiertes Array ein. Dieses entsteht im linken Teil von a. Beim Einfügen von $a[i]$ müssen wir soweit nötig Elemente links von $a[i]$ verschieben, bis wir den richtigen Platz für $a[i]$ finden.

- $a[1]$ ist sortiert.
- $x = a[2]$ wird 'entnommen', sodaß an der Stelle 2 ein Platz frei wird.
- Wenn $x < a[1]$ ist, muß $a[1]$ an die Stelle 2 verschoben werden, damit x auf die Stelle 1 kann.
- Nun ist $a[1..2]$ sortiert, und $a[3]$ wird 'entnommen' usw.

Offenbar müssen wir uns in der äußeren Schleife über die n-1 Elemente von a arbeiten, und in der inneren über die jeweils verbleibenden auf der linken Seite.

Aufgaben:

1. Sortieren Sie eine 'zufällige' Folge von min. 8 Zahlen mit diesem Verfahren.
2. Die Zeitkomplexität im best case $O(n)$ tritt auf, wenn die Element aufsteigend sortiert sind. Warum?
3. Der worst case $O(n^2)$ tritt auf, wenn die Elemente absteigend sortiert sind. Warum?
4. Der avg case ist $O(n^2)$. Warum?
5. Formulieren Sie diesen Algorithmus in Pseudocode!
6. Wo liegen die Stärken und Schwächen dieses Verfahrens?

3.3 Sortieren durch Vertauschen (bubble sort)

Mit etwas Phantasie sieht man, wie die größeren Elemente bei jedem Durchgang weiter nach rechts aufsteigen, ähnlich Luftblasen, daher auch der Name bubble sort.

- Das Array wird von links nach rechts abgearbeitet.
- Wenn ein Element größer als sein Nachfolger ist, wird getauscht.

- Nach dem 1. Durchgang landet das Maximum an der Stelle n, daher muß der nächste Durchgang nur bis $n-1$ gehen.
- Sobald in einem Durchgang kein Tausch nötig war, kann beendet werden.

Aufgaben:

1. Sortieren Sie eine 'zufällige' Folge von min. 8 Zahlen mit diesem Verfahren.
2. Die Zeitkomplexität des best case ist $O(n)$ und tritt auf, wenn die Elemente bereits aufsteigend sortiert sind. Warum?
3. Die Laufzeit liegt nahe dem best case, wenn die Elemente bereits fast sortiert sind. Warum?
4. Der worst case ist $O(n^2)$ und tritt auf, wenn die Element absteigend sortiert sind. Warum?
5. Der average case ist $O(n^2)$. Warum?
6. Formulieren Sie diesen Algorithmus in Pseudocode!
7. Welche Stärken und Schwächen hat dieses Verfahren?

Die einfachen Sortierverfahren haben alle eine Zeitkomplexität von $O(n^2)$ im average case. Das ist für viele praktische Anwendungen unbrauchbar. Daher wurden Verfahren entwickelt, die ein Array $O(n \log n)$ Schritten sortieren; sie sind allerdings auch etwas komplizierter. Das wohl bekannteste Verfahren dieser Art ist der Qicksort.

3.4 Quicksort

Der Quicksort-Algorithmus basiert auf der Grundidee, das zu sortierende Array wiederholt zu partitionieren, sodaß alle Elemente links vom frei wählbaren Partitionselement kleiner und alle rechts davon größer sind. [7]

- Wir wählen $a[1]$ als Partitionselement p.
- Nun kontrollieren wir von rechts kommend ob $a[n] > p$, dann $a[n-1] > p$ usw.
- Ein Element, das bereits auf der richtigen Seite von p ist, wird 'fixiert', d.h. bleibt dort.
- Ansonsten wird mit p getauscht und die Richtung geändert, d.h. wir vergleichen jetzt von links kommend die noch nicht fixierten Elemente.
- Wenn wir bei p ankommen, sind alle übrigen Elemente auf der jeweils richtigen Seite von p, und die Partitionierung ist abgeschlossen.
- Nun wird rekursiv die linke und rechte Seite von p ebenso partitioniert, usw. bis wir bei Partitionen der Länge Eins ankommen, die trivialerweise sortiert sind.

Der Quicksort-Algorithmus benötigt im average case und im best case $O(n \ln n)$ Schritte, was nicht ganz einfach zu zeigen ist ($\ln x$ ist der Logarithmus naturalis von x, also zur Basis $e = 2.718..$). Als Merkhilfe kann man die Partitionierungen in linken und rechten Teil

[7] Quicksort wurde 1961 von Tony Hoare entwickelt.

als 'Binärbaum der rekursiven Aufrufe' verstehen und mit dem Einfügen und Suchen im Binärbaum vergleichen, wo wir ja auch einen logarithmischen Term finden.
Ebenfalls nicht einfach zu zeigen ist, daß eine weitere Verbesserung der Komplexitätsklasse nicht möglich ist, d.h. es wird auch durch noch so ausgefeilte Algorithmen nicht gelingen, z.B. eine Sortierung in $O(n)$ Schritten im average case zu erreichen.
Es gibt auch andere Varianten des Quicksort, die aber alle die gleiche $O(n \log n)$ Komplexität haben. Quicksort ist (wie der Name schon sagt) im average case eines der schnellsten Sortierverfahren; im worst case benötigt er allerdings $O(n^2)$ Schritte. Andere Verfahren wie Heapsort sind hier robuster.

Aufgaben:

1. Sortieren Sie eine 'zufällige' Folge von min. 8 Zahlen mit diesem Verfahren.
2. Formulieren Sie diesen Algorithmus in Pseudocode.
3. Zeigen Sie anhand einer Skizze die average case Zeitkomplexität von $O(n \ln n)$.
4. Wo liegen die Stärken und Schwächen dieses Verfahrens?

3.5 Laufzeiten der Sortierverfahren

Die Sortierverfahren selection, insertion, bubble, quick, merge und heap sort wurden in Java programmiert; in Tab. 1 sind durchschnittliche Laufzeiten in Mikrosekunden mit zufällig initialisierten Arrays dargestellt. Der JIT Compiler der Sun Java VM wurde dazu mit der Option -Xint ausgeschaltet, sodaß auch oft ausgeführter Java Byte Code immer interpretiert wird.[8]
Wie man sieht, schneidet der bubble sort am schlechtesten von allen direkten Verfahren ab, obwohl er gerade in Einführungsveranstaltungen oft gezeigt wird. Der selection sort ist zwar intuitiv am einfachsten, aber der insertion sort ist flotter und bleibt bis 20 Elemente das schnellste Verfahren; erst ab 25 Elementen wird er vom Quicksort geschlagen.
Allen einfachen Verfahren ist die avg case $O(n^2)$ Laufzeit gemeinsam, die natürlich für größere Arrays schnell untragbar wird. Die höheren Sortierverfahren erreichen avg case $O(n \log n)$, sind allerdings auch etwas schwieriger zu verstehen und zu programmieren.

3.6 Marktnischen für die direkten Verfahren

Vergleicht man die Zeitkomplexitäten der direkten und der höheren Sortierverfahren, so überlegt man vielleicht, warum direkte Verfahren überhaupt noch zum Einsatz kommen.
Es ist zu beachten, daß die 'Zeitkomplexität' in der O-Notation die Abhängigkeit der Schrittzahl von der Problemgröße beschreibt. Die Laufzeit in Mikrosekunden kann durch Ausnützen von Tricks bei der Implementierung stark beeinflußt werden.

[8] Normalerweise gibt man die -Xint Option natürlich nicht an, denn man will ja bestmögliche Performance; dann stellt sich hier allerdings ein überraschender Effekt ein. Java Byte Code wird zunächst interpretiert; erst wenn Teile des Codes öfter verwendet werden, entscheidet sich der JIT Compiler dazu, diese zu compilieren, was die Laufzeiten dann drastisch senkt.

n	select	insert	bubble	quick	merge	heap
5	2.7	**2.4**	2.8	3.3	5.5	5.1
10	4.7	**4.0**	5.4	5.4	9.7	9.9
15	7.6	**6.2**	9.8	7.5	14.4	15.0
20	11.8	**9.2**	15.9	9.8	19.3	20.8
25	16.6	13.0	23.6	**12.2**	24.4	26.7
30	22.5	17.5	33.0	**14.7**	29.6	32.8
35	29.5	22.7	44.0	**17.3**	34.7	39.3
40	37.2	28.6	57.1	**19.9**	40.2	46.0
50	56.2	42.7	88.1	**25.8**	51.9	60.5
60	78.8	59.1	126.4	**31.0**	62.9	74.0
80	136.2	102.7	222.8	**43.1**	86.9	104.8
100	208.5	156.4	345.6	**55.6**	111.8	135.8

Tabelle 1: Laufzeiten einiger Sortierverfahren in Mikrosekunden

Durch den Overhead beim rekursiven Funktionsaufruf kann es für kleine n von Vorteil sein, ein direktes Verfahren zu verwenden. Aus diesem Grund werden auch hybride Verfahren eingesetzt, z.B. Quicksort mit Insertion Sort, der für Partitionen mit weniger als ca. 20 Elementen verwendet wird.

4 Listen

Die Liste ist eine in der Informatik sehr häufig verwendete Datenstruktur und stellt eine Art Zwischenschritt zwischen Array und Baum dar: das einzelne Datenelement hat im Array keinen Zeiger auf Vorgänger oder Nachfolger, im Baum hat es beide Zeiger, und in der Liste nur einen, den Zeiger auf den Nachfolger; eine solche Liste bezeichnen wir als linked list.

4.1 Linked List

Wie das Array ist auch die Linked List mit geringem Aufwand umzusetzen; im Unterschied zum Array ermöglicht sie dynamisch wachsende Strukturen. Auch sie wird daher häufig eingesetzt.

- Im Unterschied zum Array werden die einzelnen Elemente an verschiedenen Stellen im Speicher abgelegt: mit jedem Element wird ein Zeiger auf das nächste Element gespeichert (vgl. Abb. 5). Ein Zeiger auf das erste Element repräsentiert die gesamte Liste, da vom ersten Element aus alle übrigen gefunden werden können, indem die Kette der Zeiger verfolgt wird.
  Der Zeiger *null* im letzten Element signalisiert das Ende der Liste. Zeiger können mit realitiv wenig Platzbedarf gespeichert werden (z.B. 32 bit).

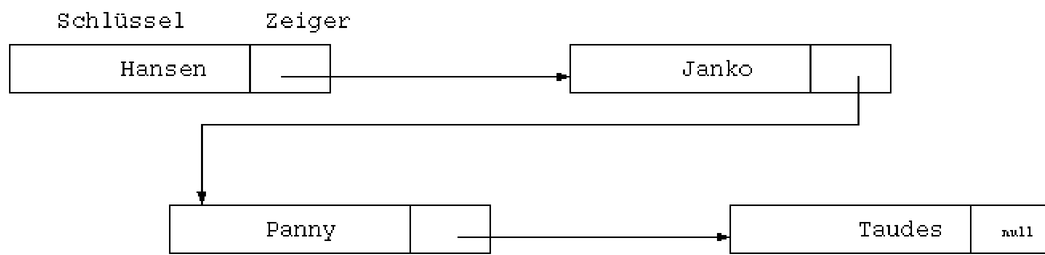


Abbildung 5: Linked List

- Ein direkter Zugriff mit Index auf ein bestimmtes Element wie im Array ist nicht möglich, die Liste muß immer beginnend mit dem ersten Element sequentiell abgearbeitet werden.
- Dafür kann die Struktur problemlos dynamisch wachsen und schrumpfen. Die einzelnen Elemente werden unabhängig von einander eingefügt und gelöscht. Speicherverwaltungen wie der garbage collector in Java oder malloc in C können damit sehr effizient umgehen.

4.2 Zeitkomplexitäten der Operationen in der Linked List

- Einfügen in $O(1)$ durch 'Umbiegen' der Zeiger.
- Suchen in $O(n)$, da die Elemente vom Anfang beginnend durchlaufen werden müssen.
- Löschen durch Umbiegen der Zeiger in $O(1)$, allerdings muß das Element erst gefunden werden, daher insgesamt $O(n)$.
- Ist eine häufige sortierte Ausgabe nötig, kann die Liste auch sortiert gehalten werden, indem bei jedem Einfügen zunächst der passende Platz gesucht wird. Dadurch steigt natürlich die Komplexität des Einfügens auf $O(n)$.
- Eine binäre Suche wie im sortierten Array ist nicht möglich, da wir ja nicht an beliebige Stellen der Liste springen können, sondern immer nur vom Anfang bis zum Schluß durchlaufen können.

Aufgaben:

1. Simulieren Sie Einfügen, Suchen und Löschen in der Linked List an einem Beispiel mit zumindest 4 Elementen!
2. Formulieren Sie Einfügen, Suchen und Löschen in der Linked List in Pseudocode!

4.3 Merge Sort

Dieses Sortierverfahren bietet sich an, wenn nicht Arrays, sondern Listen sortiert werden sollen. Die Idee besteht darin, die zu sortierende Liste rekursiv wiederholt in zwei ungefähr gleich große Hälften zu trennen, bis man bei Länge Eins ankommt; beim folgenden Zusammenführen (merge) nimmt man immer das jeweils kleinere Element zuerst, bis beide Listen

leer sind. [9]
Es soll (3,1,7,5) sortiert werden. Wir trennen in (3,1) und (7,5); dann trennen wir weiter in (3) und (1), sowie in (7) und (5). Nun folgt die Merge Phase. Wir führen (3) und (1) zu (1,3) zusammen, indem wir zuerst von der zweiten Liste 1 entnehmen und danach 3 von der ersten Liste. Ebenso führen wir (7) und (5) zu (5,7) zusammen. Die Listen (1,3) und (5,7) führen wir zusammen, indem wir zuerst 1 von Liste 1 entnehmen, dann 3 wieder von Liste 1, und schließlich die restlichen Elemente dazugeben, womit wir dann (1,3,5,7) erhalten.
Das Verfahren ist einfach und besonders interessant, wenn Daten auf Magnetbändern sortiert werden sollen, und mehrere Laufwerke zur Verfügung stehen. Es bewährt sich aber auch mit Festplatten als Medium.

Aufgaben:

1. Sortieren Sie ein zufälliges Array von min. 7 Elementen mit diesem Verfahren.
2. Beschreiben Sie dieses Verfahren in Pseudocode.
3. Merge Sort hat im best, average und worst case eine Laufzeit von $O(n \log_2 n)$. Warum?

Array und Linked List sind für viele Anwendungen durchaus angemessene Datenstrukturen. Sollen allerdings etwa einige hunderttausend Patientendatensätze gepflegt werden, in denen laufend Einfügungen, Suchen und sortierte Ausgaben nötig sind, so fordern wir eine schnellere Abarbeitung als jedesmal etwa die Hälfte der Datensätze zu durchforsten. Ein Möglichkeit dazu sind die sortierten Binärbäume.

5 Sortierte Binärbäume

Ein **Baum** besteht aus Knoten und gerichteten Kanten, wobei jedem Knoten genau ein Vorgänger und Null bis viele Nachfolger zugeordnet sind, mit Ausnahme des obersten Knotens, der keinen Vorgänger hat und Wurzelknoten genannt wird.
Ein Knoten, der keine Nachfolger hat, wird als **Blattknoten** bezeichnet, die übrigen Knoten heißen **interne Knoten**.
Beim **Binärbaum** sind jedem Knoten maximal zwei Nachfolger zugeordnet. Zwischen der Anzahl der Knoten und der Anzahl der Ebenen (Tiefe) gibt es einen Zusammenhang:

- Der leere Binärbaum hat Null Knoten und Tiefe Null.
- Der Binärbaum mit Tiefe Eins hat einen Knoten.
- Ein Binärbaum mit Tiefe zwei hat zwei oder drei Knoten.
- Ein Binärbaum mit Tiefe drei hat maximal sieben Knoten.
- Ein Binärbaum mit Tiefe m hat maximal $2^m - 1$ Knoten.

[9] Merge Sort wurde bereits 1945 von John von Neumann entdeckt.

- Ein Binärbaum mit n Knoten hat daher mindestens Tiefe $\lfloor \log_2 n \rfloor + 1$, und maximal Tiefe n. [10]

Beim **vollständigen Binärbaum** werden die Ebenen der Reihe nach gefüllt, d.h. Knoten auf Ebene drei werden erst eingefügt, wenn Ebene zwei gefüllt ist usw. Ein vollständiger Binärbaum mit n Knoten hat daher minimale Tiefe.
Beim sortierten Binärbaum ist für jeden Knoten der linke Nachfolger kleiner und der rechte Nachfolger größer.[11]
Beachte: gemäß Definition sind *alle* linken Nachfolger eines Knotens k kleiner als k und *alle* rechten Nachfolger größer k.
Implementiert wird der sortierte Binärbaum z.B. als Record-Struktur mit drei Feldern: Schlüssel sowie Zeiger auf linken und rechten Nachfolger, in Java z.B. so:

```
class Node {
  int key;
  Node left;
  Node right;
}
```

5.1 Interne und externe Pfadlänge

Zählen wir die Anzahl der Kanten vom Wurzelknoten bis einem bestimmten Knoten k, so erhalten wir die Länge des Pfades zu k. Summieren wir die Pfadlängen aller Knoten im Baum, so erhalten wir die **interne Pfadlänge** I des Baums. Wir beobachten, daß sich die interne Pfadlänge bei fixer Knotenzahl je nach Form des Baums ändern.
Ordnen wir jedem der n Knoten soweit möglich zusätzliche 'externe' Knoten zu, so erhalten wird den erweiterten Binärbaum mit s externen Knoten; es gilt:
$s = n + 1$
Zählen wir für jeden externen Knoten k die Anzahl der Kanten vom Wurzelknoten zu k und summieren über alle externen Knoten, so erhalten wir die **externe Pfadlänge** E. Es gilt:
$E = I + 2n$

5.2 Suchen und Einfügen

Beim Suchen im sortierten Binärbaum beginnen wir beim Wurzelknoten und und verleichen mit dem gesuchten Wert. Wir gehen zum linken Nachfolger, wenn der gesuchte Wert kleiner als jener des aktuellen Knotens, und sonst zum rechten Nachfolger. Das machen wir solange,

[10] Der Ausdruck $\log_2 x$ bedeutet Logarithmus dualis von x, das ist der Logarithmus zur Basis 2. Einige Beispiele: $\log_2 8 = 3$ weil $2^3 = 8$, $\log_2 7 \doteq 2.807$ weil $2^{2.807} \doteq 7$. Der Audruck $\lfloor x \rfloor$ steht für den ganzzahligen Teil von x.

[11] Bei Gleichheit muß man eine Konvention treffen, ob man für den linken Nachfolger kleiner gleich definiert oder für den rechten größer gleich; das wollen wir in der Folge ignorieren und annehmen, daß unsere Elemente nicht mehrfach vorkommen.

bis wir den Knoten mit dem gesuchten Wert gefunden haben, oder keinen Nachfolger mehr finden.
Beim Aufbau des Binärbaums fügen wir das erste Element als Wurzelknoten ein. Für jedes folgende Element suchen wir zunächst die richtige Position wie beschrieben und fügen es dann als neuen Nachfolger an dieser Stelle ein.

Aufgaben:

1. Simulieren Sie Suchen und Einfügen in einem Baum mit min. 6 Elementen.
2. Beschreiben Sie das Suchen im sortierten Binärbaum mit Pseudocode.
3. Beschreiben Sie das Einfügen im sortierten Binärbaum mit Pseudocode.
4. Pseudocode für aufsteigend sortierte Ausgabe
5. Pseudocode für absteigend sortierte Ausgabe

5.3 Sortierte Binärbaume: Löschen

Wenn wir einen Knoten k löschen wollen, müssen wir drei Fälle unterscheiden:

1. k ist ein Blattknoten; wir können in problemlos entfernen.
2. k hat einen Nachfolger x: wir setzen x an die Stelle von k
3. k hat zwei Nachfolger: wir suchen den kleinsten Knoten m im rechten Teilbaum, löschen ihn gemäß 1. oder 2. und setzen ihn an die Stelle von k.[12]

Analog kann man im Fall 3 auch den größten Knoten im linken Teilbaum wählen.
Offenbar hängt die Form des resultierenden Baums von den eingefügten und gelöschten Elementen und der Reihenfolge der Operationen ab. Die maximale Anzahl von Vergleichen beim Suchen entspricht der Tiefe des Baums. Bei fixer Knotenzahl n werden daher Bäume mit minimaler Tiefe bevorzugt; solche mit maximaler oder fast maximaler Tiefe werden als 'entartet' bezeichnet.

Aufgaben:

1. Simulieren Sie das Löschen im sortierten Binärbaum mit min. 6 Elementen.
2. Formulieren Sie Pseudocode für das Löschen im sortierten Binärbaum.

5.4 Rotationen

Rotationen können in jedem sortierten Binärbaum vorgenommen werden, um der Entartung entgegenzuwirken. Nehmen wir an, A hat einen rechten Nachfolger B, der nach unten rotiert werden soll. Dann setzen wir

[12] Das Löschen von m geht immer gemäß 1. oder 2., denn wenn m zwei Nachfolger hätte, dann wäre er nicht der kleinste Knoten im rechten Teilbaum.

- $X = B.left$
- $B.left = X.right$
- $X.right = B$
- $A.right = X$

Beachten Sie, wie der rechte Nachfolger von X zum linken Nachfolger von B wird, also bei der Rotation "auf die andere Seite" wechselt. Der linke Nachfolger von X und der rechte Nachfolger von B bleiben unberührt.

5.5 Aufgaben:

1. Simulieren Sie Rotationen nach links und rechts in sortierten Binärbäumen mit min. 6 Elementen.
2. Formulieren Sie die Rotation in die andere Richtung in Pseudocode.
3. Welche Zeitkomplexität hat die Rotation?

6 Heaps

Für viele Anwendungen ist es nicht nötig, alle Elemente jederzeit sortiert im Zugriff zu haben.
Oft genügt ein schneller Zugriff auf das aktuelle Maximum oder Minimum.
Ein vollständiger Binärbaum ist ein Heap-Baum, wenn für jeden Knoten k die Heap-Bedingung gilt: alle Nachfolger sind kleiner/gleich k (Max-Heap) bzw. größer/gleich k (Min-Heap).
Einfügen: das neue Element kommt zunächst auf den nächsten freien Platz, sodaß der Baum vollständig bleibt. In der **Upheap**-Prozedur prüfen wir aufsteigend die Heap-Bedingung und tauschen falls nötig den aktuellen Knoten mit seinem Vorgänger. Haben wir in der Upheap-Prozedur eine Ebene erreicht, wo kein Tausch notwendig ist, können wir abbrechen: auf den Ebenen darüber wird sicher ebenfalls kein Tausch mehr nötig sein.
Max/Min Entnehmen: der Wurzelknoten wird durch das letzte Element ersetzt, sodaß der Baum vollständig bleibt. In der **Downheap**-Prozedur prüfen wir absteigend wieder die Heap-Bedingung und tauschen falls nötig den aktuellen Knoten mit einem seiner Nachfolger; die Heap-Bedingung muß erhalten bleiben, daher tauschen wir in einem Max-Heap mit dem größeren der beiden Nachfolger, und in einem Min-Heap mit dem kleineren.
Verwenden wir das Datum der Ankunft eines Elements (oder eine fortlaufende Nummer) als Schlüssel, so können wir mit dem Max-Heap einen Kellerspeicher (stack) implementieren, und mit dem Min-Heap eine Schlange (queue).

6.1 Array-Darstellung

Die Elemente eines Heaps können wir wie folgt nummerieren: Wurzelknoten = 1, nächste Ebene linker Nachfolger = 2, rechter Nachfolger = 3, nächste Ebene 4, 5, 6, 7 usw.
Daher können wir den Heap in einem Array speichern: für jeden Knoten an der Stelle i (beginnend mit 1 für den Wurzelknoten) setzen wir die beiden Nachfolger auf $2i$ und $2i + 1$.

Umgekehrt erhalten wir die Position des Vorgängers eines Knotens an der Stelle i durch ganzzahlige Division durch 2 (Rest fällt weg).
Es entstehen keine freien Plätze im Array, weil der Heap ja vollständig besetzt ist. Für die Implementierung ist das ein großer Vorteil, weil wir keine Zeiger brauchen.

6.2 Komplexitätsüberlegungen

Bei beiden Algorithmen konzentrieren wir uns auf den den variablen Teil der notwendigen Zeit, d.h. die Prozeduren Upheap und Downheap.
worst case: Die maximale Anzahl der Schritte in Upheap und Downheap hängt offenbar von der Tiefe des Baums ab. Im 'normalen' d.h. nicht allzusehr entarteten Binärbaum hängt die Tiefe d von der Anzahl der Elemente n folgendermaßen ab: [13]
$d(n) \sim \log_2 n$
Mit zunehmender Entartung wächst $d(n)$ bis zur Extremform $d(n) = n$. Da ein Heap ein vollständig besetzter Binärbaum ist, tritt das Problem der Entartung nicht auf, und man kann $d(n)$ exakt angeben: [14]
$d(n) = \lfloor \log_2 n \rfloor + 1$
average case: Da in Upheap und Downheap nicht immer alle Ebenen durchschritten werden müssen, wird die durchschnittliche Anzahl von Schritten geringer sein, aber immer noch von $\log_2 n$ abhängen.
best case: Einfügen eines Elements, das kleiner ist (Max-Heap) als alle schon vorhandenen Elemente; Entnehmen im Max-Heap mit lauter gleichen Elementen.
In der O-Notation erhalten wir daher:

	best	worst	average
Einfügen	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
Entnehmen	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

Ähnliche Überlegungen können wir auch für Suchen, Einfügen und Löschen im sortierten Binärbaum anstellen, wobei hier allerdings wegen der verschiedenen Grade der Entartung nicht so einfach wie beim Heap vorgegangen werden kann; trotzdem können wir auch hier Ergebnisse in der Nähe von $O(\log_2 n)$ erwarten.

Aufgaben:

1. In einen anfangs leeren Max-Heap werden folgende Elemente eingefügt:
   10 30 15 20 12 11
   Zeichnen Sie den Heap-Baum nach jedem Schritt.
2. Aus dem obigen Heap wird das Maximum entnommen. Zeichnen Sie den entstandenen Heap-Baum.
3. Wie 1., aber in Array-Darstellung.

[13] weil ja umgekehrt die Anzahl der Elemente in Abhängigkeit von der Tiefe d mit $n \sim 2^d$ wächst.
[14] Der Ausdruck $\lfloor x \rfloor$ liefert den ganzzahligen Anteil von x.

4. Wie 2., aber ebenfalls in Array-Darstellung.
5. Beschreiben Sie die Vor- und Nachteile der Array-Darstellung für Heaps. Warum können Heaps problemlos als Arrays dargestellt werden, nicht aber sortierte Binärbäume? Welche Probleme würden dabei auftreten?
6. Skizzieren Sie das Einfügen in einen Max-Heap mit Pseudocode! Der Heap soll in der Array-Darstellung gespeichert sein.
7. Skizzieren Sie das Entnehmen des Maximums aus einem Max-Heap mit Pseudocode! Arraydarstellung wie oben.

6.3 Heapsort

Sozusagen als Nebeneffekt liefert die Datenstruktur Heap auch ein interessantes Sortierverfahren: wir bauen zunächst links im Array einen Max-Heap in Array-Darstellung auf und entnehmen dann immer wieder das Maximum und setzen es an die Stelle n, dann auf $n-1$ usw.

- Zu Beginn steht auf $i = 1$ das Maximum eines Heaps mit einem Element.
- Nun fügen wir das Element $i + 1$ in den Heap ein und führen soweit notwendig die Upheap-Prozedur aus, usw bis $i = n$.
- Schließlich enthält das gesamte Array einen Max-Heap in Array-Darstellung.
- Nun entnehmen wir das Maximum und setzen es an die Stelle n; es folgt die Downheap-Prozedur.
- Danach entnehmen wir wieder das Maximum und setzen es auf $n-1$, usw.
- Wenn der Max-Heap leer ist, dann ist das Array sortiert.

Dieses Verfahren hat gegenüber Quicksort den Vorteil, daß es nicht nur im average case, sondern auch im worst case eine günstige Zeitkomplexität hat. Leider ist es im average case um den Faktor 1.4 langsamer als Quicksort.

Aufgaben:

1. Simulieren Sie das Verfahren mit einem Array mit min. 6 Elementen.
2. Bestimmen Sie die Zeitkomplexität für best, average und worst case, und vergleichen Sie das Resultat mit Quicksort!
3. Formulieren Sie den Algorithmus in Pseudocode!

7 AVL-Bäume

Ein AVL-Baum ist ein sortierter Binärbaum, der zusätzlich die **Balance-Bedingung** erfüllt: für jeden Knoten muß gelten, daß die Tiefen der beiden nachfolgenden Teilbäume um höchstens Eins unterschiedlich sind.[15]

[15] Die AVL-Bäume sind nach ihren Erfindern Georgy Adelson-Velsky und Yevgeniy Landis (1962) benannt.

Mit jedem Knoten werden **Balance-Indikatoren** gespeichert, die anzeigen, welcher nachfolgende Teilbaum tiefer ist: der linke (L), der rechte (R), oder beide gleich (0).
Sowohl beim Einfügen als auch beim Löschen kann der Baum aus der Balance geraten; dann muß die Balance durch Rotationen wieder hergestellt werden. [16]

Aufgaben:

1. Was unterscheidet einen AVL-Baum von einem SBB und einem Heap?
2. Was können Sie über die Tiefe von AVL-Bäumen und SBB sagen, und welche Bedeutung hat das?
3. Geben Sie ein Beispiel für einen SBB mit min. 6 Elementen, der auch ein AVL-Baum ist!
4. Geben Sie ein Beispiel für einen SBB mit min. 6 Elementen, der kein AVL-Baum ist, aber gleichzeitig so wenig entartet wie möglich ist!
5. Versehen Sie einen beliebigen AVL-Baum mit min. 6 Elementen mit Balance-Indikatoren!

7.1 Einfügen im AVL-Baum

Mit Hilfe der Balanceindikatoren können wir für jedes Einfügen einen **standardisierten Pfad** konstruieren, indem wir vom eingefügten Knoten aus nach oben aufsteigend jeweils prüfen, ob wir von der tieferen Seite kommen (+), von der weniger tiefen (-), oder keines von beiden (0). Diese Symbole schreiben wir der Reihe nach *von rechts nach links* an.
So erhalten wir eine Folge von $\{+,-,0\}$, die wir in zwei Klasse unterteilen: [17]

$$\begin{array}{ll} \{+,-,0\}*\{-\}\{0\}* \;\cup\; \{0\}* & \text{unproblematisch} \\ \{+,-,0\}*\{+\}\{0\}* & \text{problematisch} \end{array}$$

Das ist auch leicht nachzuvollziehen: wenn wir von rechts kommend (d.h. beim Aufsteigen) Nullen ignorieren und zuerst ein + finden, dann haben wir auf der tieferen Seite eingefügt, und der Baum ist daher aus der Balance geraten.
Beim Einfügen gehen wir zunächst entsprechend dem Einfügen im sortierten Binärbaum vor (Abstiegsphase). In der darauffolgenden Aufstiegsphase prüfen wir die Balanceindikatoren und rotieren, falls nötig, d.h. falls es sich um einen problematischen Pfad handelt.
Der Teilbaum A ist nach dem Einfügen aus der Balance geraten, d.h. seine Nachfolger haben die Tiefen h und h+2. Der tiefere Teilbaum von A sei B, der tiefere Teilbaum von B sei X.

- Fall I: X liegt aussen. In diesem Fall rotieren wir B nach oben und haben die Balance wieder hergestellt. Die Aufstiegsphase ist damit abgeschlossen.

[16] Unter http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm findet sich eine nette Animation für die Operationen im AVL-Baum.

[17] Der **reguläre Ausdruck** $x*$ bedeutet 0 bis beliebig viele Vorkommen des Zeichens x; das Zeichen $\cup$ bedeutet Vereinigung von zwei Mengen. Mit $\{a, b\}$ ist ein a oder ein b gemeint.

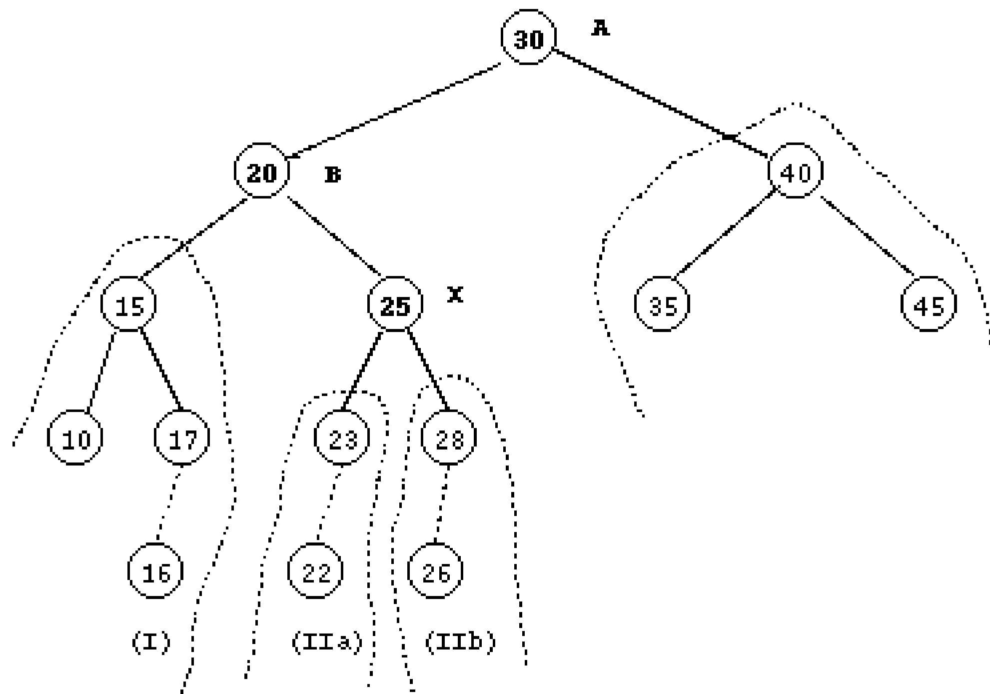


Abbildung 6: Einfügen im AVL-Baum

- Fall II: X liegt innen. Hier müssen wir zunächst X nach oben rotieren; damit haben wir die Situation von Fall I mit X an der Stelle von B und können durch nochmalige Rotation von X nach oben die Balance wieder herstellen. Die Aufstiegsphase ist damit abgeschlossen.

Ein Fall III, wo beide Teilbäume von B gleich tief sind, d.h. die Tiefe h+1 haben, ist nicht möglich, weil der Baum dann schon vor dem Einfügen nicht in Balance gewesen wäre!

7.2 Beispiele

Der Baum in Abb. 6 mit den Einfügungen I, IIa und IIb soll als Ausgangspunkt dienen. [18] Die Großbuchstaben kennzeichnen die Knoten, die rotiert werden. Die strichliert umrandeten Teilbäume bleiben beim Rotieren in sich unverändert, d.h. ihre Struktur und ihre nach dem Einfügen gesetzten Balanceindikatoren werden durch die Rotationen nicht beeinflußt.

Aufgaben

1. Welche Bedeutung für die Balancebedingung haben 'unproblematische' und 'problematische' Pfade beim Einfügen in AVL-Bäumen?

[18] Die Unterscheidung zwischen IIa und IIb ist für die Balanceindikatoren sinnvoll; sie hat keine Bedeutung für die Rotationen.

2. Geben Sie für einen beliebigen AVL-Baum mit min. 6 Elementen und Balance-Indikatoren eine Einfügeoperation an, die in einem unproblematischen Pfad resultiert!
3. Geben Sie für einen beliebigen AVL-Baum mit min. 6 Elementen und Balance-Indikatoren eine Einfügeoperation an, die in einem problematischen Pfad resultiert!
4. Wie sind unproblematische und problematische Einfügepfade allgemein aufgebaut?
5. Führen Sie in einem AVL-Baum mit min. 6 Elementen eine Einfügung gemäß Fall I durch und bringen Sie die Balance-Indikatoren auf den neuen Stand.
6. Das gleiche nach Fall II.
7. Formulieren Sie das Einfügen im AVL-Baum in Pseudocode!

7.3 Löschen im AVL-Baum

Auch hier konstruieren wir wieder den standardisierten Pfad wie beim Einfügen. Dann unterscheiden wir:

$$\begin{array}{ll} \{+,-+*\}\{0\}\{+\}* \ \cup \ \{+\}+ & \text{unproblematisch} \\ \{+,-,0\}\{-\}\{+\}* & \text{problematisch} \end{array}$$

Auch das ist wieder leicht nachzuvollziehen: wenn wir von rechts kommend (d.h. aufsteigend) alle + ignorieren und zuerst ein - finden, dann haben wir auf der weniger tiefen Seite gelöscht und daher den Baum aus der Balance gebracht.
Es wird zunächst wieder entsprechend dem Verfahren im SBB gelöscht. Der resultierende Teilbaum A kann aus der Balance geraten sein, und zwar genau dann, wenn es sich um einen problematischen Pfad handelt. Der tiefere Teilbaum von A sei wieder B, der tiefere Teilbaum von B (falls es ihn gibt) sei X.

- Fall I: X liegt aussen. Rotieren wie in Fall I beim Einfügen, d.h. B hinauf.
- Fall II: X liegt innen. Rotieren wie in Fall II beim Einfügen, d.h. X zweimal hinauf. Der resultierende Baum ist um eine Ebene weniger tief, daher muß die Aufstiegsphase fortgesetzt werden!
- Fall III: beide Teilbäume von B sind gleich tief.[19] Rotieren wie in Fall I, d.h. B hinauf.

Aufgaben:

1. Erklären Sie die unproblematischen und problematischen Pfade beim Löschen!
2. Demonstrieren Sie ein Beispiel für eine unproblematische Löschung in einem AVL-Baum mit mindestens 8 Elementen!
3. wie vorhin, aber Fall I
4. wie vorhin, aber Fall II
5. wie vorhin, aber Fall III
6. Formulieren Sie das Löschen im AVL-Baum in Pseudocode und überprüfen Sie Ihre Lösung anhand einiger Beispiele!

[19] Das kann beim Einfügen nie vorkommen, denn dann wäre der Baum ja schon vorher nicht balanciert.

8 Hashing

Bis jetzt haben wir Datenstrukturen mit Suchen in $O(n)$ und $O(\log_2 n)$ kennengelernt. Die Frage drängt sich auf: geht es nicht noch schneller?
Ist Speicherplatz kein Engpaß, so kann mit einer Hash-Tabelle eine Datenorganisation mit extrem schnellem Zugriff implementiert werden. Zur Veranschaulichung betrachten wir die Zahlen 2, 5, 7, 11, 13, die in einem Array mit *fixer Länge* m abgespeichert werden sollen: steht uns genügend Platz zur Verfügung, so erstellen wir ein Array mit 13 Plätzen und speichern jeden Wert x an der jeweiligen Stelle i wie folgt:

i	1	2	3	4	5	6	7	8	9	10	11	12	13
x		2			5		7				11		13

Wir können nun diese Werte nicht nur mit einem Zugriff abspeichern, sondern auch mit einem Zugriff suchen: wenn der Wert x nicht an der Stelle i = x steht, so ist er im ganzen Array nicht enthalten; z.B. ist die Zahl 8 offenbar nicht im Array, da die Stelle 8 ja leer ist. Wir haben also einen Algorithmus, dessen Zeitkomplexität für Einfügen und Suchen O(1) ist - besser geht es nicht!
Bei diesem 'direct addressing' bleibt allerdings viel Speicherplatz leer, und wir werden auch nicht für beliebige Zahlen ein entsprechend grosses Array erstellen können. Daher müssen wir dieses Verfahren für Arrays realistischer Größe anpassen.
Eine **Hashfunktion** h(x) ordnet jedem Wert x eine gültige Speicherstelle i zu, z.B. mit der Arraygröße m = 7 wählen wir i = h(k) = k mod m + 1, wobei die Modulofunktion den Rest bei ganzzahliger Division liefert. Damit erhalten wir h(2) = 3, h(5) = 6, h(7) = 1, h(11) = 5, h(13) = 7:

i	1	2	3	4	5	6	7
x	7		2		11	5	13

Mit m = 7 haben wir sogar eine **perfekte Hashfunktion** gefunden, da jedem Wert eine eigene Stelle zugeordnet wird. Es ist allerdings nur unter sehr einschränkenden Bedingungen möglich, eine solche Funktion zu finden. Soll z.B. auch noch die Zahl 14 abgespeichert werden, so haben wir h(14) = 1, und diese Stelle ist schon belegt. Damit ist unsere Hashfunktion nicht mehr perfekt. Wir müssen uns überlegen, was wir in einem solchen Fall tun.

8.1 Kollisionen

Eine gute Hashfunktion liefert für 'fast alle' Werte eindeutige Stellen. Für die wenigen Ausnahmen muß ein Verfahren zur **Kollisionsresolution** definiert werden, d.h. es muß bestimmt werden, wo der betreffende Wert abzuspeichern bzw. zu suchen ist, wenn die Stelle aus der Hashfunktion schon belegt ist bzw. nicht den gesuchten Wert enthält.
Die einfache und naheliegende Methode des **linear probing** sucht in der nächsten Stelle: mit h(14) = 1 finden wir die Stelle 1 schon mit dem Wert 7 besetzt, daher erhöhen wir den

Index solange um Eins, bis wir (beim Einfügen) eine freie Stelle finden, oder (beim Suchen) eine leere.
Im Beispiel sind wir bereits bei i = 2 erfolgreich. Der average case hängt natürlich von unserem Geschick bei der Wahl der Hashfunktion ab, wird aber in der Nähe von O(1) liegen. Der worst case tritt auf, wenn das Array nahezu voll besetzt ist, denn dann müssen wir schlimmstenfalls fast alle Stellen kontrollieren, d.h. wir brauchen O(n) Schritte.
Die Kollisionsresolution nach dem **linear probing** zieht den unangenehmen Effekt des **clustering** nach sich, indem die Kollisionsblöcke immer länger werden. Mit anderen Varianten kann dieser Effekt vermindert, aber nie ganz vermieden werden.
Spätestens wenn alle Stellen besetzt sind muß das Array vergrößert werden. Allerdings wird Hashing schon vorher mit zunehmender Besetzung des Arrays immer langsamer, da dann die Häufigkeit der Kollisionen steigt. Es wird daher ab einem bestimmten Besetzungsgrad besser sein, eine Reorganisation vorzunehmen, d.h. das Array zu vergrößern, die Hashingfunktion anzupassen und die Elemente neu zu besetzen. Dieser gelegentliche Aufwand der Größenordnung O(n) muß bei einer genauen Komplexitätsanalyse miteinbezogen werden.

Aufgaben:

1. Entwerfen Sie ein hash table Verfahren für die Abspeicherung von Namen in einem Array fixer Länge m. Testen Sie Ihr Verfahren mit den Namen JANKO, HANSEN, PANNY, TAUDES, NEUMANN, JAMMERNEGG und m = 10.

8.2 Andere Hashfunktionen und Anwendungen

Die Wahl der Hashfunktion bestimmt die Eigenschaften des Verfahrens. Neben der Anwendung in hash tables werden Hashfunktionen auch als **message digest** verwendet. Hier sucht man eine 'Zusammenfassung' der Klartextnachricht m, die wesentlich kürzer als m sein soll, z.B. 160 Bit beim Secure Hash Algorithm SHA-1; ausserdem soll es möglichst aufwendig sein, ein m2 ungleich m zufinden, sodaß h(m2) = h(m).
Damit kann die Integrität einer Nachricht kontrolliert werden, indem sowohl Klartext als auch Digest verschickt werden: der Empfänger vergleicht dazu den erhaltenen Digest mit dem aus der erhaltenen Nachricht generierten.
Eine einfache solche Hashfunktion ist der CRC (cyclic redundancy check). Ähnliche Verfahren können auch für hash tables verwendet werden, indem der Digest als Index interpretiert wird.

```
function crc(bit array bitString[1..len], int polynomial) {
  shiftRegister := initial value // commonly all 0 bits or all 1 bits
  for i from 1 to len {
    if most significant bit of shiftRegister xor bitString[i] = 1
      shiftregister := (shiftregister left shift 1) xor polynomial
    else
      shiftRegister := shiftregister left shift 1
  }
```

```
  return shiftregister
}
```

Diese Hashfunktion wird u.a. zur Kontrolle der Datenintegrität im Ethernet sowie als Kommando cksum in Unix verwendet. Das polynomial ist hier 0x04C11DB7, das Verfahren ist als CRC-32 bekannt (der Digest besteht aus 32 Bit).

8.3 Hashfunktionen als message digest für Kryptographie

Auch in kryptographischen Anwendungen sind Hashfunktionen für message digests in Verwendung. CRC bietet hier aber nicht genügend Schutz vor Mißbrauch, sodaß stattdessen andere, aufwendigere Verfahren wie MD5 (Message Digest, 128 Bit) und SHA-1 (Secure Hash Algorithm, 160 Bit) eingesetzt werden.
Die einfache Digest-Variante in Abb. 7 illustriert die Grundidee. Hier steht M[i] für das i-te Byte im Klartext. Die Bytes A, B, C, D werden mit Null initialisiert und dann in jedem Schritt i wie angegeben neu berechnet. Auch kleine Änderungen im Klartext erzeugen 'fast immer' einen völlig neuen Digest:

```
% java MD "Auch kleine Aenderungen im Klartext erzeugen einen voellig neuen Digest"
16a7c8fb
% java MD "Auch kleine Aenderungen am Klartext erzeugen einen voellig neuen Digest"
46bfc89b
```

Üblicherweise wird der Digest in Hex-Code ausgegeben, denn sonst würde es bei vielen Bitfolgen bei der Ausgabe als Text Probleme geben. Auch hier werden die 32 Bit als 8 Stellen Hex-Code dargestellt. Zum Indizieren in einem hash table der Größe m können die 32 Bit als Integer aufgefaßt und modulo m genommen werden.
Die iterative Berechnung und das Wandern der einzelnen Blöcke nach rechts sieht man am folgenden Beispiel:

```
% java MD "A"
00410000
% java MD "Au"
```

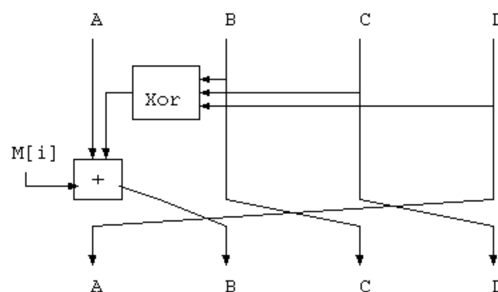


Abbildung 7: Einfacher Message Digest

```
00b64100
% java MD "Auc"
005ab641
% java MD "Auch"
41155ab6
% java MD "Auch "
b65a155a
% java MD "Auch k"
5a365a15
% java MD "Auch kl"
153f365a
```

8.4 Kryptographie

Eine Nachricht m soll in eine verschlüsselte Form c gebracht werden, sodaß aus c bei Kenntnis des passenden Schlüssels wieder m rekonstruiert werden kann.
Symmetrisch: derselbe Schlüssel k für Ver- und Entschlüsselung
Beispiel: m, c, k seien Bitstrings
m wird mit k durch XOR verschlüsselt: xor(m,k) = c, xor(c,k) = m

- Vorteile: einfach, schnell
- Nachteil: beide Seiten müssen den Schlüssel k kennen

Asymmetrisch: verschiedene Schlüssel zum Ver- und Entschlüsseln: public und private key

- Vorteil: der private key ist nur dem Eigentümer bekannt, der public key kann (und soll) allgemein bekannt sein
- Nachteil: so aufwendig, daß nur für ganz kurze m verwendbar, z.B. SHA-1 Digest mit 160 bit

8.5 Asymmetrische Kryptographie

Nur für kurze Bytefolgen praktisch durchführbar, daher Anwendung nicht auf der Originalnachricht, sondern auf symmetrischem Key oder Digest

- Verschlüsselte Übertragung: verschlüsseln mit public key, entschlüsseln mit private key Die Nachricht kann nur jemand entschlüsseln, der den zum public key passenden private key besitzt. Praxis: zuerst zufälligen Key generieren und asymmetrisch übermitteln, dann Nachricht symmetrisch verschlüsselt übertragen
- Signatur: verschlüsseln mit private key, entschlüsseln mit public key Die Signatur konnte nur jemand erzeugen, der den zum public key passenden private key besitzt.
- Praxis: nicht Originalnachricht signieren, sondern Digest

Problem der Authentifizierung: über den Unterzeichner wissen wir nur eines: daß er den zum public key passenden private key besitzt. Aber ist der Besitzer des public key wirklich der, der er vorgibt zu sein?
Lösung: Information über public key und Identität des Ausstellers wird selbst wieder signiert.

1. Certificate Authority signiert die Information über public key und Identität des Ausstellers
   - der Signatur der Certificate Authority muß vertraut werden
   - Kosten, z.B. satte 2400 Euro pro Jahr für a.sign corporate signatur bei atrust.at
   - trotzdem im kommerziellen Bereich verbreitet
2. Web of Trust (PGP), Kette von Certificates ausgehend von der einzelnen Person
   - keine zentrale Stelle
   - keine Kosten
   - im privaten Bereich gelegentlich zu finden

   Aktuelle Entwicklungen: Bankomatkarte mit digitaler Signatur und Bürgerkartenfunktion um ca 15 Euro pro Jahr

8.6 RSA Algorithmus

Ron Rivest, Adi Shamir, Len Adleman (1977)
Plaintexts are positive integers up to 2^{512}. Keys are quadruples (p, q, e, d), with p a 256-bit prime number, q a 258-bit prime number, and d and e large numbers with $(de - 1)$ divisible by $(p - 1)(q - 1)$.
We define $E_K(P) = P^e \ mod\ pq$, $D_K(C) = C^d \ mod\ pq$.
[...] Now E_K is easily computed from the pair (pq, e) — but, *as far as anyone knows*, there is no *easy* way to compute D_K from the pair (pq, e). So whoever generates K can publish (pq, e).
E_K ... encryption function
D_K ... decryption function
P ... plain text (Nachricht im Klartext)
C ... cypher (verschlüsselte Nachricht)

Genaueres siehe z.B.

- http://en.wikipedia.org/wiki/RSA
- http://www.faqs.org/faqs/cryptography-faq/part06/
  insb. Section 6.6 *Is RSA secure?* Nobody knows.

Hinweise auf Fehler sowie andere Kommentare bitte an mitloehn@wu-wien.ac.at