Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik VI

Algorithmen und Datenstrukturen

Vorlesungsmitschrift zur Vorlesung im SS 2000

Prof. Dr.-Ing. H. Ney

Ausgearbeitet von
Maximilian Bisani,
Jörg Dahmen,
Wolfgang Macherey und
Sonja Nießen

Nicht-autorisierte Vorabversion

Letzte Überarbeitung: 5. März 2001

Inhaltsverzeichnis

1 Grundlagen **6**

1.1 Einführung 6

1.1.1 Problemstellungen 6

1.1.2 Aktualität des Themas 7

1.1.3 Ziele der Vorlesung 7

1.1.4 Hinweis auf das didaktische Problem der Vorlesung 7

1.1.5 Vorgehensweise 8

1.1.6 Datenstrukturen 8

1.2 Algorithmen und Komplexität 9

1.2.1 Random-Access-Machine (RAM) 11

1.2.2 Komplexitätsklassen 14

1.2.3 O-Notation für asymptotische Aussagen 16

1.2.4 Regeln für die Laufzeitanalyse 18

1.2.5 Beispiel: Fibonacci-Zahlen 19

1.2.6 Beispiel: Fakultät 21

1.3 Datenstrukturen 22

1.3.1 Datentypen 22

1.3.2 Listen 25

1.3.3 Stacks 31

1.3.4 Queues 33

1.3.5 Bäume 35

1.3.6 Abstrakte Datentypen 46

1.4 Entwurfsmethoden 48
1.4.1 Divide-and-Conquer-Strategie 48
1.4.2 Dynamische Programmierung 51
1.4.3 Memoization 54
1.5 Rekursionsgleichungen 55
1.5.1 Sukzessives Einsetzen 55
1.5.2 Master-Theorem für Rekursionsgleichungen 56
1.5.3 Rekursionsungleichungen 60

2 Sortieren **62**
2.1 Einführung 62
2.2 Elementare Sortierverfahren 65
2.2.1 SelectionSort 65
2.2.2 InsertionSort 67
2.2.3 BubbleSort 70
2.2.4 Indirektes Sortieren 74
2.2.5 BucketSort 76
2.3 QuickSort 78
2.3.1 Beweis der Schranken von $\sum_{k=m}^{N} 1/k$ mittels Integral-Methode . 88
2.4 HeapSort 89
2.4.1 Komplexitätsanalyse von HeapSort 93
2.5 Untere und obere Schranken für das Sortierproblem 95
2.6 Schranken für $N!$ 98
2.7 MergeSort 101
2.8 Zusammenfassung 101

3 Suchen in Mengen **102**
3.1 Problemstellung 102
3.2 Einfache Implementierungen 104

3.2.1 Ungeordnete Arrays und Listen 104
3.2.2 Vergleichsbasierte Methoden 104
3.2.3 Bitvektordarstellung (Kleines Universum) 107
3.2.4 Spezielle Array-Implementierung 108
3.3 Hashing 110
3.3.1 Begriffe und Unterscheidung 110
3.3.2 Hashfunktionen 112
3.3.3 Wahrscheinlichkeit von Kollisionen 113
3.3.4 Offenes Hashing (Hashing mit Verkettung) 116
3.3.5 Geschlossene Hashverfahren (Hashing mit offener Adressierung) . 118
3.3.6 Zusammenfassung der Hashverfahren 123
3.4 Binäre Suchbäume 123
3.4.1 Allgemeine binäre Suchbäume 123
3.4.2 Der optimale binäre Suchbaum 136
3.5 Balancierte Bäume 142
3.5.1 AVL-Bäume 142
3.5.2 (a,b)-Bäume 148
3.6 Priority Queue und Heap 150

4 Graphen **152**
4.1 Motivation: Wozu braucht man Graphen? 152
4.2 Definitionen und Graph-Darstellungen 153
4.2.1 Graph-Darstellungen 154
4.2.2 Programme zu den Darstellungen 156
4.3 Graph-Durchlauf 157
4.3.1 Programm für den Graph-Durchlauf 158
4.4 Kürzeste Wege 163
4.4.1 Dijkstra-Algorithmus (Single-Source Best Path) 163
4.4.2 Floyd-Algorithmus (All Pairs Best Path) 167

4.4.3 Warshall-Algorithmus 170

4.5 Minimaler Spannbaum 172

4.5.1 Definitionen 172

4.5.2 MST Property 172

4.5.3 Prim-Algorithmus 173

4.6 Zusammenfassung 175

Literaturverzeichnis

[Cormen et al.] T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms.* MIT press / McGraw Hill, 10th printing, 1993.

[Sedgewick 88] R. Sedgewick: *Algorithms.* 2nd ed., Addison-Wesley, 1988.

[Sedgewick 93] R. Sedgewick: *Algorithms in Modula-3.* Addison-Wesley, 1993.

[Schöning] U. Schöning: *Algorithmen – kurz gefasst.* Spektrum Akad. Verl., 1997

[Güting] R.H. Güting: *Datenstrukturen und Algorithmen.* Teubner, 1992.

[Aho et al. 83] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Aho et al. 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Mehlhorn] K. Mehlhorn: *Datenstrukturen und effiziente Algorithmen.* Bde 1,2,3; (primär Band 1: „Sortieren und Suchen“), Teubner 1988. (Bde 2+3 vergriffen)

[Wirth] N. Wirth: *Algorithmen und Datenstrukturen mit Modula-2.* 4. Auflage, Teubner 1986.

[Aigner] M. Aigner: *Diskrete Mathematik.* Vieweg Studium, 1993.

[Schinzel] B. Schinzel: *Datenstrukturen und Algorithmen.* Augustinus-Buchhandlung Aachen, 1991

[Ottmann] T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen.* 2.Auflage, Wissenschaftsverlag, 1993

Hinweise zur Literatur

[Cormen et al.]	beste und modernste Darstellung
[Sedgewick 88, Sedgewick 93]	konkrete Programme
[Schöning]	sehr kompakte und klare Darstellung
[Güting]	schöne Darstellung des Suchens in Mengen
[Aho et al. 83]	als Einführung sehr schön
[Aho et al. 74]	sehr breit; deckt auch numerische Verfahren ab
[Mehlhorn]	Motivation und Analyse der Algorithmen
[Wirth]	viel Modula-2
allgemein:	DUDEN Verlag: Schülerduden INFORMATIK, bzw. Duden INFORMATIK (fast identisch)

Diese Seite ist absichtlich leer.

1 Grundlagen

1.1 Einführung

1.1.1 Problemstellungen

Diese Vorlesung befaßt sich mit Algorithmen und den zugehörigen Datenstrukturen. Dabei werden sowohl konkrete Implementierungen erarbeitet als auch Analysen ihrer Komplexität bezüglich Rechenzeit und Speicherplatzbedarf angestellt. Wir werden folgende Problemstellungen der Informatik betrachten:

Sortieren

- Sortierung einer Folge von Zahlen
- Bestimmung des Median

Suchen in Mengen

- Symboltabellen für Compiler (enthalten vordefinierte Wörter, Variablen, etc.)
- Autorenverzeichnis einer Bibliothek
- Kontenverzeichnis einer Bank
- allgemein Suchen in Datenbanken

Pfade in einem Graphen

- kürzester Pfad von A nach B
- Minimaler Spannbaum (kürzester Pfad, der alle Knoten miteinander verbindet)
- kürzeste Rundreise (Traveling-Salesman-Problem (TSP))

String-Matching
Suche nach Zeichenfolgen im Text.

1.1.2 Aktualität des Themas

Die Suche nach guten Algorithmen und ihrer effizienten Implementierung ist stets aktuell gewesen. Auch in Zeiten immer schneller wachsender Speicher- und Rechenkapazitäten ist es aus ökonomischen Gründen unerläßlich, diese möglichst effizient zu nutzen. Wie wir sehen werden, können ineffiziente Algorithmen auch noch so große Kapazitäten sehr schnell erschöpfen.

Auf der theoretischen Seite existieren immer noch zahlreiche ungeklärte Fragestellungen: So ist zum Beispiel bisher keine effiziente, exakte Lösung für das Traveling-Salesman-Problem bekannt. Daneben gibt es in der angewandten Informatik viele Gebiete, die nach extrem effizienten Algorithmen verlangen, z.B.: Compilerbau, Datenbanken, künstliche Intelligenz, Bild- und Sprachverarbeitung und Erkennung.

1.1.3 Ziele der Vorlesung

Sie sollten am Ende der Vorlesung Kenntnisse und Fähigkeiten in den folgenden Bereichen erlangt haben:

Theoretische Kenntnisse: Was zeichnet effiziente Verfahren aus, und welche Algorithmen und Datenstrukturen stecken dahinter? Wie ist der Aufwand (Rechenzeit, Speicherplatz) eines Verfahrens definiert? Wie kann man diesen Aufwand in Komplexitätsmaße fassen, und wie wird ein Algorithmus analysiert?

Praktische Fähigkeiten: Wie analysiert man ein Problem und bildet es auf Datenstrukturen und Algorithmen ab? Wie implementiert man den Algorithmus als lauffähiges Programm und testet dieses dann?

1.1.4 Hinweis auf das didaktische Problem der Vorlesung

Es ist wesentlich schwieriger zu beschreiben, wie man von der Problemstellung zum fertigen Programm kommt, als im nachhinein das fertige Programm zu erklären und nachzuvollziehen. Daher konzentrieren sich die meisten Bücher mehr auf die fertigen Programme als auf die Motivation und den Weg zum Programm.

Gelegentlich werden wir auch in der Vorlesung diesen Weg gehen. Versuchen Sie dann selbst, die Herleitung und Implementierung dieser Standard-Algorithmen nachzuvollziehen. Implementieren Sie diese vielleicht nach einigen Tagen noch einmal aus freier Hand. Verifizieren Sie insbesondere, daß Sie auch Details wie z.B. Index-Grenzen richtig hinbekommen, denn der Teufel steckt im Detail.

Aus den folgenden Gebieten der *Mathematik* werden immer wieder Anleihen gemacht, achten Sie deshalb darauf, daß Ihnen deren Grundlagen vertraut sind:

- Kombinatorik (Fakultät, Binomialkoeffizienten)
- Wahrscheinlichkeitsrechnung
- elementare Analysis (hauptsächlich Grenzwerte)

1.1.5 Vorgehensweise

Bei der Behandlung von Algorithmen und Datenstrukturen werden zwei Ebenen unterschieden:

Die algorithmische Ebene umfaßt eine möglichst allgemeine und maschinen-unabhängige Beschreibung der Objekte (zu manipulierende Daten) und des Algorithmus. Dies hat den Vorteil, daß man sich auf das Wesentliche konzentrieren kann, und sich nicht mit maschinen- und programmiersprachen-spezifischen Details aufhalten muß.

Die programmiersprachliche Ebene umfaßt die konkrete Implementierung. Der Übersichtlichkeit wegen werden in der Vorlesung nicht immer vollständig lauffähige Programme vorgestellt. Zum Verständnis der Programmtexte ist die grundlegende Kenntnis einer imperativen Programmiersprache Voraussetzung. Programme und Algorithmen sind hier in Modula-3 oder Modula-3-ähnlichem Pseudocode formuliert.

1.1.6 Datenstrukturen

Datenstrukturen und Algorithmen sind unmittelbar miteinander verknüpft und können nicht getrennt voneinander betrachtet werden, da ein Algorithmus mit den Methoden arbeiten muß, die auf einer Datenstruktur definiert (und implementiert) sind. In den folgenden Kapiteln werden Datenstrukturen aus den folgenden Kategorien vorgestellt:

- Sequenzen (Folgen, Listen) (Abschnitt 1.3.2)
- Mengen (speziell Wörterbücher (dictionaries)) (Abschnitt 3)
- Graphen (speziell Bäume) (Abschnitt 1.3.5)

1.2 Algorithmen und Komplexität

Die Komplexitätsordnung eines Algorithmus bestimmt maßgeblich die Laufzeit und Platzbedarf der konkreten Implementierung. Diese hängen aber, außer vom Algorithmus selbst, u.a. noch von den folgenden Größen ab:

- Eingabedaten
- Rechner-Hardware
- Qualität des Compilers
- Betriebssystem

Die letzten drei Punkte sollen uns hier nicht interessieren. Wir betrachten nur die *abstrakte Laufzeit* $T(n)$ in Abhängigkeit von den Eingabedaten. Dabei drückt der Parameter n die Größe des Problems aus, die von Fall zu Fall unterschiedlich aufgefaßt werden kann:

- Beim *Sortieren* ist n die Anzahl der zu sortierenden Werte $a_1, \ldots, a_n$.
- Bei der *Suche nach Primzahlen* gibt n z.B. die obere Grenze des abzusuchenden Zahlenbereichs an.

Beispiel: Sortieren durch Auswählen

Zur Veranschaulichung betrachten wir ein einfaches Sortierverfahren, *Sortieren durch Auswählen*. Es besteht aus zwei verschachtelten Schleifen, und kann wie folgt implementiert werden:

```
PROCEDURE SelectionSort(VAR a: ARRAY [1..N] OF ItemType) =
VAR min : INTEGER ;
    t   : ItemType ;
BEGIN
    FOR i := 1 TO N - 1 DO
        min := i ;
        FOR j := i + 1 TO N DO
            IF a[j] < a[min] THEN
                min := j ;
            END ;
        END ;
        t := a[min] ; a[min] := a[i] ; a[i] := t ;
    END ;
END SelectionSort ;
```

Die Rechenzeit $T(n)$ wird hier bestimmt durch den Aufwand für die elementaren Operationen *vergleichen* und *vertauschen*. Deren Anzahl können wir in Abhängigkeit von der Anzahl n der zu sortierenden Elemente wie folgt berechnen:

- Anzahl der Vergleiche (compares):

$$C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- Anzahl der Vertauschungen (exchanges, swaps):

$$E(n) = n - 1$$

  Dies gilt, weil Vertauschungen nur in der äußeren Schleife ausgeführt werden, und diese genau $(n-1)$-mal durchlaufen wird.

- Bezeichnet man den relativen Aufwand für Vergleiche und Vertauschungen mit α_1 bzw. α_2, dann ergibt sich der folgende Gesamtaufwand:

$$\begin{aligned} T(n) &= \alpha_1 C(n) + \alpha_2 E(n) \\ &= \alpha_1 \frac{n(n-1)}{2} + \alpha_2(n-1) \\ &\approx \alpha_1 \frac{n^2}{2} \quad \text{für große } n \quad (n \gg \frac{\alpha_2}{\alpha_1}) \end{aligned}$$

Diese Aussage gilt für große n. Wir sagen, daß das *asymptotische Verhalten* von $T(n)$ dem von n^2 entspricht, oder daß $T(n)$ quadratische Komplexität hat.

1.2.1 Random-Access-Machine (RAM)

Die abstrakte Laufzeit ergibt sich aus der Zahl der elementaren Operationen. Um genaue, vergleichbare Aussagen über die Komplexität eines Programms machen zu können, muß definiert werden, was unter einer elementaren Operation zu verstehen ist. Dazu eignet sich ein axiomatisch definiertes Rechnermodell. Dieser Modell-Rechner dient dann als Vergleichsmaßstab.

Ein solches Rechnermodel stellt die *verallgemeinerte Registermaschine* (random access machine, RAM) dar. Eine RAM wird durch folgende Elemente definiert (vgl. Abb. 1.1):

- adressierbare Speicherzellen (random access registers), deren Inhalte ganzzahlige Werte (integers) sind
- Ein- und Ausgabebänder, die nur von links nach rechts bewegt werden können
- zentrale Recheneinheit mit:
  - Akkumulator
  - Befehlszähler (program counter)
- Programm (Der Programmcode soll sich selbst nicht verändern können und wird daher außerhalb des adressierbaren Speichers abgelegt.)

Das Programm wird in einem sehr rudimentären Befehlssatz (abstrakter Assembler) verfaßt (vgl. Tabelle 1.1). Einzelheiten können variieren; so ist der Akkumulator oft kein eigener Speicher, sondern einfach die Speicherzelle mit der Nummer 0. Variationen gibt es auch beim Befehlssatz, den möglichen Datentypen u.v.m.

Anmerkung: Die (moderneren) RISC-Prozessoren (reduced instruction set computer) sind in ihrer Architektur einer RAM sehr ähnlich. Die konkurrierenden CISC-Architekturen (complex instruction set computer) haben einen wesentlich umfangreicheren Befehlssatz, und zerlegen die komplexen Befehle bei der Ausführung in einfachere Arbeitsschritte.

Kostenabschätzung

Je nach Zielsetzung kann man zwischen zwei Kostenmaßen wählen:

Einheitskostenmaß: Jeder Instruktion wird ein konstanter Aufwand zugeordnet, unabhänging von den jeweiligen Operanden. Dies entspricht dem Fall, daß die zu verarbeitenden Werte die Wortlänge des Rechners nicht überschreiten.

Logarithmisches Kostenmaß: (auch Bit-Kostenmaß) Die Kosten sind von der Länge des Operanden (Anzahl der Bits) abhängig. Dies ist angebracht, wenn die Operanden mehr als je eine Speicherzelle (Wort) belegen.

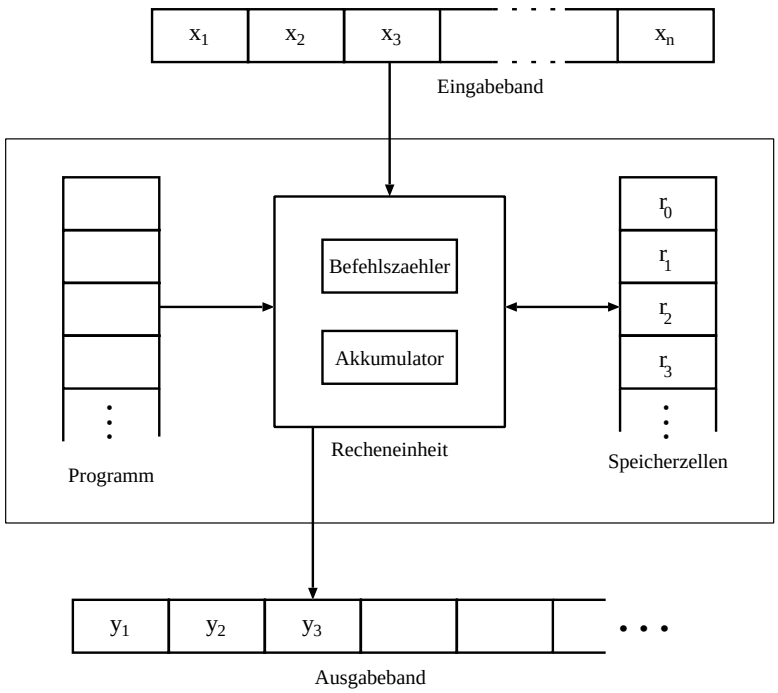


Abbildung 1.1: Schematische Darstellung der Random-Access-Machine (RAM)

LOAD a	$c(A) \leftarrow a$
STORE a	$a \leftarrow c(A)$
ADD a	$c(A) \leftarrow c(A) + a$
SUB a	$c(A) \leftarrow c(A) - a$
MUL a	$c(A) \leftarrow c(A) * a$
DIV a	$c(A) \leftarrow \lfloor c(A)/a \rfloor$
READ	$c(A) \leftarrow$ aktuelles Eingabesymbol
WRITE	aktuelles Ausgabesymbol $\leftarrow c(a)$
JUMP b	Programmzähler wird auf Anweisung Nummer b gesetzt.
JZERO b	Programmzähler wird auf Anweisung Nummer b gesetzt, falls $c(A) = 0$, sonst auf die nächste Anweisung.
JGTZ b	Programmzähler wird auf Anweisung Nummer b gesetzt, falls $c(A) > 0$, sonst auf die nächste Anweisung.
HALT	Verarbeitung wird beendet.

Tabelle 1.1: Befehlssatz einer Random-Access-Machine (RAM): Es steht $c(\cdot)$ für „Inhalt von“, A für „Akkumulator“. Der Operand a ist entweder i, $c(i)$, oder $c(c(i))$ mit $i \in \mathbb{N}$, entsprechend den Adressierungsarten immediate, direkt und indirekt.

Das logarithmische Kostenmaß entspricht dem Komplexitätsbegriff auf einer Turing-Maschine.

Die Kosten, die einem Befehl zugeordnet werden, sind wenig standardisiert. Man versucht in der Regel eine möglichst realistische Schätzung zu finden. Folgende Faktoren spielen dabei eine Rolle:

- Art des Speicherzugriffs: keiner, direkt, indirekt
- Art des Befehls
  - Arithmetische Operationen: MUL und DIV können wesentlich teurer sein als ADD und SUB.
  - Sprungbefehle: Bedingte Sprünge (JZERO) können teurer sein als unbedingte (JUMP)

Die RAM erfaßt (ziemlich) genau die im realen Rechner anfallenden Speicher-Operationen. Externe Speicher wie z.B. Festplatten werden nicht betrachtet.

Prozeduraufrufe und Rekursion

Der Befehlssatz der RAM enthält kein Konstrukt zur Definition von Unterprogrammen (Funktionen, Prozeduren), die insbesondere für rekursive Algorithmen benötigt werden. Prozeduren, wie sie in Modula-3 und vielen anderen Sprachen zur Verfügung stehen, werden vom Compiler nach folgender Methode in elementare Anweisungen der RAM umgewandelt:

1. Bei jedem Aufruf (Inkarnation, Aktivierung) wird ein Speicherbereich, der sogenannte Aktivierungsblock (Versorgungsblock), (auf dem sog. call-stack) reserviert. Dieser enthält Platz für
   - die aktuellen Parameter,
   - die Rücksprungadresse und
   - die lokalen Variablen der Prozedur.
2. Das aufrufende Programm legt die aktuellen Parameter und die Rücksprungadresse im Aktivierungsblock ab und setzt den Programmzähler auf den Anfang der aufzurufenden Prozedur.
3. Nach Beendigung der Prozedur wird der zugehörige Aktivierungsblock entfernt und die Kontrolle an das aufrufende Programm zurückgegeben.

Der Speicherbedarf für die Aktivierungsblöcke muß natürlich bei der Analyse der Speicherkomplexität berücksichtigt werden. Bei rekursiven Algorithmen wächst der Speicherbedarf proportional zur Rekursionstiefe, da für jeden individuellen Aufruf ein eigener Aktivierungsblock angelegt wird.

1.2.2 Komplexitätsklassen

Um Algorithmen bezüglich ihrer Komplexität vergleichen zu können teilt man diese in Komplexitätsklassen ein. In der folgenden Tabelle sind typische Zeitkomplexitätsklassen aufgeführt.

Klasse	Bezeichnung	Beispiel
1	konstant	elementarer Befehl
$\log(\log n)$	doppelt logarithmisch	Interpolationssuche
$\log n$	logarithmisch	binäre Suche
n	linear	lineare Suche, Minimum einer Folge
$n \log n$	überlinear	Divide-and-Conquer-Strategien, effiziente Sortierverfahren, schnelle Fourier-Transformation (FFT)
n^2	quadratisch	einfache Sortierverfahren
n^3	kubisch	Matrizen-Inversion, CYK-Parsing
n^k	polynomiell vom Grad k	lineare Programmierung
2^n	exponentiell	erschöpfende Suche (exhaustive search), Backtracking
$n!$	Fakultät	Zahl der Permutationen (Traveling-Salesman-Problem)
n^n		

Bei den logarithmischen Komplexitätsklassen spielt die Basis in der Regel keine Rolle, da ein Basiswechsel einer Multiplikation mit einem konstanten Faktor entspricht. Wenn eine Basis nicht explizit angegeben wird, dann geht diese für $\log(n)$ meist aus dem Zusammenhang hervor. Außerdem sind folgende Schreibweisen üblich:

$$\begin{array}{ll} \mathrm{ld} := \log_2 & \text{Logarithmus dualis} \\ \mathrm{ln} := \log_{\mathrm{e}} & \text{natürlicher Logarithmus} \\ \mathrm{lg} := \log_{10} & \text{dekadischer Logarithmus} \end{array}$$

Betrachtet man die Laufzeit einiger hypothetischer Algorithmen (siehe Abb. 1.2), so wird schnell klar, daß in den meisten Fällen nur Algorithmen mit maximal polynomieller Komplexität praktikabel sind. Laufzeiten proportional 2^n oder $n!$ nehmen schon bei moderaten Problemgrößen astronomische Werte an.

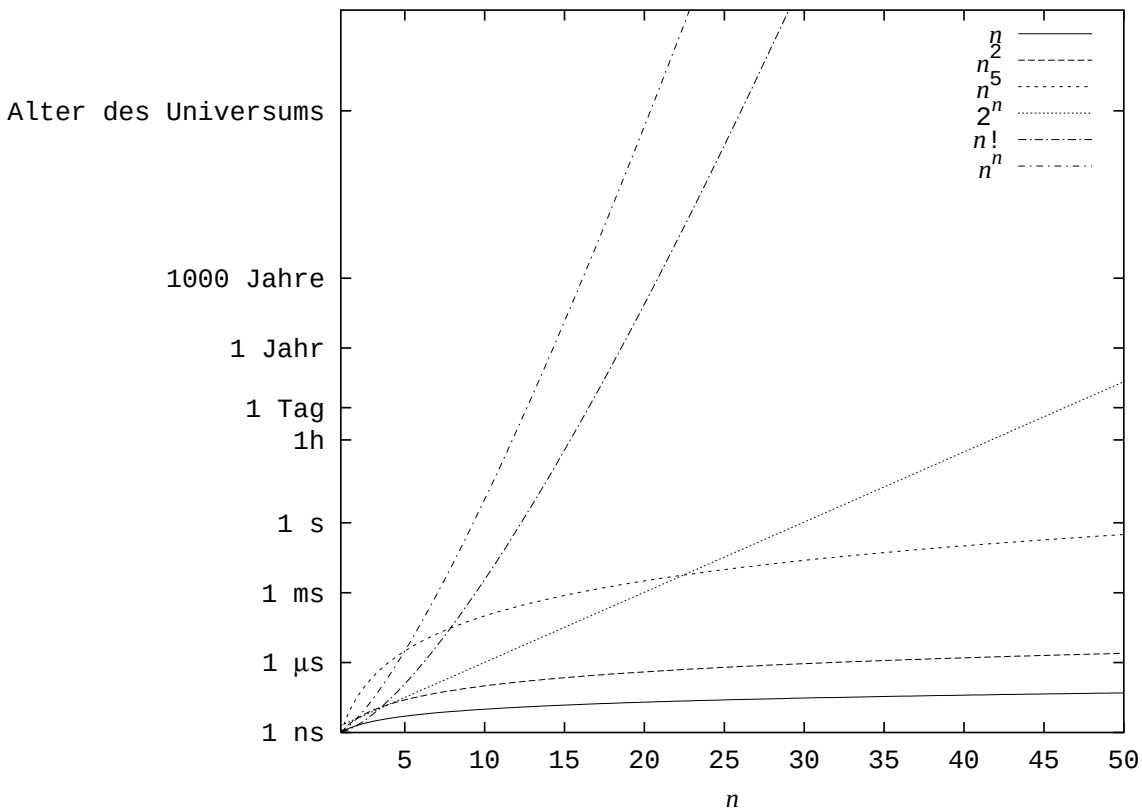


Abbildung 1.2: Wachstumsverhalten einiger typischer Komplexitätsklassen. Dargestellt ist der Laufzeitbedarf von hypothetischen Algorithmen mit den angegebenden Komplexitäten, wobei angenommen wird, daß der Rechner für jede elementare Operation 1 ns (1 ns = 10^{-9} Sekunden) benötigt. (Die Zeitachse ist logarithmisch.)

1.2.3 O-Notation für asymptotische Aussagen

Beschreibt eine Funktion die Laufzeit eines Programms, so ist es oft ausreichend, nur ihr asymptotisches Verhalten zu untersuchen. Konstante Faktoren werden vernachlässigt, und die Betrachtung beschränkt sich auf eine „einfache“ Funktion. Zugleich sollen aber möglichst enge Schranken gefunden werden.

Um asymptotische Aussagen mathematisch konkret zu fassen, definiert man die folgenden Mengen (sog. *O-Notation*):

Definition Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion, dann ist

$$\begin{aligned}
O(f) &:= \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\, \exists n_0 > 0 : \quad g(n) \leq c \cdot f(n)\, \forall n \geq n_0\} \\
\Omega(f) &:= \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\, \exists n_0 > 0 : \quad g(n) \geq c \cdot f(n)\, \forall n \geq n_0\} \\
\Theta(f) &:= \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\, \exists n_0 > 0 : \quad \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\, \forall n \geq n_0\} \\
o(f) &:= \{g : \mathbb{N} \to \mathbb{R}^+ \mid \forall c > 0\, \exists n_0 > 0 : \quad 0 \leq g(n) < cf(n)\, \forall n \geq n_0\} \\
\omega(f) &:= \{g : \mathbb{N} \to \mathbb{R}^+ \mid \forall c > 0\, \exists n_0 > 0 : \quad 0 \leq cf(n) < g(n)\, \forall n \geq n_0\}
\end{aligned}$$

Übliche Sprechweise:

$g \in O(f)$: f ist *obere Schranke* von g.
g wächst höchstens so schnell wie f.
$g \in \Omega(f)$: f ist *untere Schranke* von g.
g wächst mindestens so schnell wie f.
$g \in \Theta(f)$: f ist die *Wachstumsrate* von g.
g wächst wie f

Statt $g \in O(f)$ schreibt man oft (mathematisch nicht korrekt) $g(n) \in O(f(n))$ um die Bezeichnung der Variablen deutlich zu machen und die Definition zusätzlicher „Hilfsfunktionen“ zu vermeiden. So steht bespielsweise „$O(n^2)$“ für „$O(f)$ mit $f(n) = n^2$“. Auch die Schreibweise $g = O(f)$ ist üblich.

Allgemeine Aussagen und Rechenregeln

Für das Rechnen mit den oben definierten Mengen gelten einige einfache Regeln. Seien f und g Funktionen $\mathbb{N} \to \mathbb{R}^+$, dann gilt:

1. Linearität:
   Falls $g(n) = \alpha f(n) + \beta$ mit $\alpha, \beta \in \mathbb{R}^+$ und $f \in \Omega(1)$, dann gilt: $g \in O(f)$

2. Addition:
$$f + g \in O(\max\{f, g\}) = \begin{cases} O(g) & \text{falls} \quad f \in O(g) \\ O(f) & \text{falls} \quad g \in O(f) \end{cases}$$

3. Multiplikation:
$$a \in O(f) \wedge b \in O(g) \Rightarrow a \cdot b \in O(f \cdot g)$$

4. Falls der Grenzwert $\lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)}$ existiert, so ist $g \in O(f)$. (Der Umkehrschluß gilt nicht!)

5. Es gilt:
$$\Theta(f) = \Omega(f) \cap O(f)$$

Dabei sind Addition, Multiplikation, Maximumbildung von zwei Funktionen *bildweise* zu verstehen, also z.B. $(f + g)(n) = f(n) + g(n)$.

Beweis zu 1: Wegen $f \in \Omega(1)$ gibt es $c' > 0$ und $n' > 0$, mit $f(n) \geq c' \cdot 1 \ \forall\, n \geq n'$. Wähle $c = \alpha + \dfrac{\beta}{c'}$ und $n_0 = n'$, dann gilt:
$$g(n) := \alpha \cdot f(n) + \beta \leq c \cdot f(n) \ \forall\, n \geq n_0$$
D.h. $g \in O(f)$.

Beispiele:

a) Sei $f(n) = 7n + 3$. Dann gilt:
$$7n + 3 \leq c \cdot n \ \forall\, n \geq n_0 \text{ mit } c = 8 \text{ und } n_0 = 3$$
und damit $f(n) \in O(n)$

b) Sei $f(n) = \sum_{k=0}^{K} a_k n^k$ mit $a_k > 0$. Dann ist $f \in O(n^K)$, denn
$$f(n) \leq cn^K \ \forall\, n \geq n_0 \text{ mit } c = \sum_{k=0}^{K} a_k \text{ und } n_0 = 1$$

c) Sei $T(n) = 3^n$. Dann gilt: $T(n) \notin O(2^n)$, denn es gibt kein Paar $c > 0$, $n_0 > 0$, so daß
$$3^n \leq c \cdot 2^n \quad \forall\, n \geq n_0$$

d) Seien $T_1(n) \in O(n^2)$, $T_2(n) \in O(n^3)$ und $T_3(n) \in O(n^2 \log n)$, dann gilt:
$$T_1(n) + T_2(n) + T_3(n) \in O(n^3)$$

e) Seien $f(n) = n^2$ und $g(n) = 5n^2 + 100 \log n$. Dann gilt:
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{5n^2 + 100 \log n}{n^2} = 5$$
und damit $g \in O(f)$.

1.2.4 Regeln für die Laufzeitanalyse

Elementare Anweisungen sind in $O(1)$.

Es ist wichtig zu beachten, was im gewählten Rechnermodell (z.B. RAM) als elementare Anweisung gilt. Höhere Programmiersprachen stellen oft (scheinbar elementare) Anweisungen zur Verfügung, die vom Compiler in komplexere Anweisungsfolgen umgesetzt werden.

Folgen von Anweisungen: Setzt sich ein Programm aus mehreren Teilen zusammen, die hintereinander ausgeführt werden, so wird das asymptotische Verhalten der Laufzeit allein von dem Teil bestimmt, der den größten Aufwand hat (vgl. *Addition*).

Seien A und B zwei Anweisungen mit Laufzeiten $T_A \in O(f)$ und $T_B \in O(g)$, dann hat die Hintereinanderausführung

```
A ; B ;
```

die Laufzeit $T = T_A + T_B \in O(\max\{f, g\})$.

Schleifen: Die Gesamtlaufzeit ergibt sich als Summe über die einzelnen Durchläufe.

Oft ist die Laufzeit des Schleifenkörpers unabhängig davon, um den wievielten Durchlauf es sich handelt. In diesem Fall kann man einfach die Laufzeit des Schleifenkörpers $T_K(n)$ mit der Anzahl der Durchläufe $d(n)$ multiplizieren.

Im allgemeinen berechnet man entweder die Summe über die einzelnen Durchläufe explizit oder sucht obere Schranken für T_K und d: Wenn $T_k \in O(f)$ und $d \in O(g)$ so ist $T \in O(f \cdot g)$ (vgl. *Multiplikation*).

Bedingte Anweisungen: A und B seien zwei Prozeduren mit Laufzeiten $T_A \in O(f)$ und $T_B \in O(g)$, dann hat die Anweisung

`IF` *Bedingung* `THEN A() ELSE B() END ;`

eine Laufzeit in $O(1)+O(g+f)$, falls die Bedingung in konstanter Zeit ausgewertet wird. (vgl. *Addition*)

Prozeduraufrufe müssen nach nicht-rekursiven und rekursiven unterschieden werden.

nicht-rekursiv: Jede Prozedur kann separat analysiert werden. Zusätzlich zu den Anweisungen der Prozedur tritt nur noch ein (in der Regel kleiner) konstanter Overhead für Auf- und Abbau des Aktivierungsblocks auf.

rekursiv: Eine einfache Analyse ist nicht möglich, man muß eine Rekursionsgleichung aufstellen und diese detailliert analysieren (siehe Abschnitt 1.5).

1.2.5 Beispiel: Fibonacci-Zahlen

Wir wollen nun Laufzeitanalysen am Beispiel verschiedener Programme zur Berechnung der Fibonacci-Zahlen durchführen.

Definition der Fibonacci-Zahlen

$$f(n) := \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Damit ergibt sich die Folge: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$

Behauptung

1. $f(n) \in O(2^n)$
2. $f(n) \in \Omega(2^{n/2})$

Beweis

a) $f(n)$ ist positiv: $f(n) > 0 \quad \forall\, n > 0$

b) $f(n)$ ist für $n > 2$ streng monoton wachsend: $f(n) < f(n+1)$

c) Abschätzung nach oben: Für alle $n > 3$ gilt

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &< 2 \cdot f(n-1) \\ &< 4 \cdot f(n-2) \\ &< \ldots \\ &< 2^{n-1} \cdot f(1) = \frac{1}{2} 2^n \end{aligned}$$

Also gilt $f(n) \in O(2^n)$.

d) Abschätzung nach unten: Für alle $n > 3$ gilt

$$\begin{aligned} f(n) &> 2 \cdot f(n-2) \\ &> 4 \cdot f(n-4) \\ &\vdots \\ &> \begin{cases} 2^{\frac{n-1}{2}} \cdot f(1) & \text{, für } n \text{ ungerade} \\ 2^{\frac{n}{2}-1} \cdot f(2) & \text{, für } n \text{ gerade} \end{cases} \end{aligned}$$

Also gilt $f(n) \in \Omega(2^{n/2})$.

In Abschnitt 1.5.3 werden wir zeigen, daß $f \in \Theta(c^n)$ mit $c = \frac{1+\sqrt{5}}{2}$.

Zwei Algorithmen zur Berechnung der Fibonacci-Zahlen

Zur Berechnung der Fibonacci-Zahlen betrachten wir einen *rekursiven* und einen *iterativen* Algorithmus.

Naiver, rekursiver Algorithmus

Hier wurde die Definition der Fibonacci-Zahlen einfach übernommen und in ein Programm umgesetzt:

```
PROCEDURE Fibonacci(n : CARDINAL) : CARDINAL =
BEGIN
    IF n <= 1 THEN
        RETURN n ;
    ELSE
        RETURN Fibonacci(n-1) + Fibonacci(n-2) ;
    END ;
END Fibonacci ;
```

Sei $a > 0$ die Dauer für Funktionsaufruf, Verzweigung (`IF`) und die `RETURN`-Anweisung. Dann gilt für die Laufzeit $T(n)$ dieses Algorithmus:

$$T(n) = \begin{cases} a & \text{für } n \leq 1 \\ a + T(n-1) + T(n-2) & \text{für } n > 1 \end{cases}$$

Offensichtlich ist $T(n) = a, a, 3a, 5a, 9a, \ldots \geq af(n+1)$. Daher ist

$$T(n) \in \Omega(2^{n/2})$$

Stellt man die rekursiven Aufrufe dieser Prozedur für $f(5)$ als Baum dar (siehe Abb. 1.3), wird klar, warum dieser Algorithmus sehr aufwendig ist: Man betrachte nur die Wiederholungen der Aufrufe von $f(0)$ und $f(1)$.

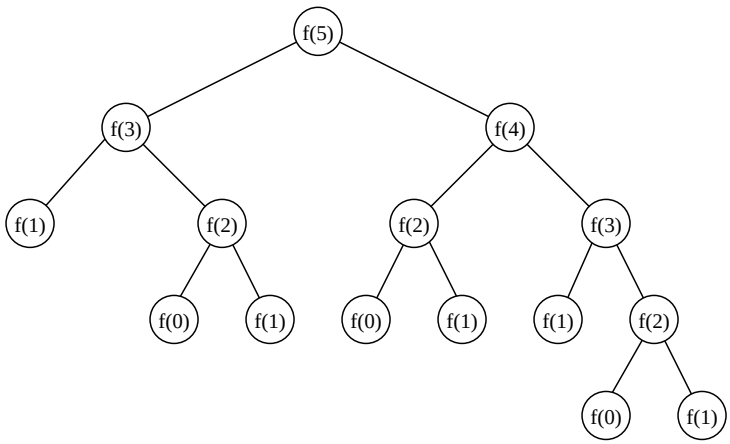


Abbildung 1.3: Baumdarstellung der Aufrufe des rekursiven Fibonacci-Programms

Iterativer Algorithmus

Der iterative Algorithmus basiert darauf, die Fibonacci-Zahlen aufzuzählen und jede neue Zahl als Summe ihrer beiden Vorgänger zu berechnen. Das spart die redundanten Aufrufe zur Berechnung von Fibonacci-Zahlen, die schon einmal berechnet wurden.

```
PROCEDURE Fibonacci(n : CARDINAL) : CARDINAL =
VAR fib_new, fib_old, t : CARDINAL ;
BEGIN
    fib_new := 0 ; fib_old := 1 ;
    FOR i := 1 TO n DO
        t := fib_new ;
        fib_new := fib_new + fib_old ;
        fib_old := t ;
    END ;
    RETURN fib_new ;
END Fibonacci ;
```

Die Laufzeit dieses Algorithmus ist offensichtlich $\Theta(n)$. Das bedeutet eine erhebliche Verbesserung gegenüber der naiven Implementierung.

1.2.6 Beispiel: Fakultät

Wir wollen nun noch am Beispiel der Fakultät zeigen, daß in manchen Fällen auch ein rekursiver Algorithmus, abgesehen vom Overhead für die Funktionsaufrufe und dem Speicherbedarf für die Versorgungsblöcke, keine Nachteile hat.

Definition der Fakultät

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

Für die Laufzeit $T(n)$ des entsprechenden rekursiven Programms ergibt sich mit geeigneten Konstanten $a, b > 0$:

$$\begin{aligned} T(n) &= \begin{cases} a & n = 0 \\ b + T(n-1) & n > 0 \end{cases} \\ T(n) &= b + T(n-1) \\ &= 2 \cdot b + T(n-2) \\ &= \ldots \\ &= n \cdot b + T(0) \\ &= n \cdot b + a \\ &\in \Theta(n) \end{aligned}$$

Der Algorithmus hat eine Laufzeit $T(n)$, die linear in n ist, und hat damit (bis auf einen konstanten Faktor) das gleiche Laufzeitverhalten wie eine entsprechende iterative Implementierung.

1.3 Datenstrukturen

1.3.1 Datentypen

Die in Modula-3-ähnlichen Programmiersprachen verfügbaren Datentypen lassen sich in drei Gruppen unterteilen: elementare und zusammengesetzte Typen, sowie Zeiger.

Elementare (atomare) Datentypen

- `INTEGER` : Ganze Zahlen
- `CARDINAL` : Natürliche Zahlen
- `CHARACTER` : Zeichen (Buchstaben etc.)
- `REAL` : Dezimalzahlen
- `BOOLEAN` : Wahrheitswerte (`TRUE` oder `FALSE`)

- zusätzlich existieren in Pascal und Modula-3 noch
  - Aufzählungstypen
    z.B. `TYPE Wochentag = {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag} ;`
  - Unterbereichstypen
    z.B. `TYPE Lottozahl = [1..49] ;`

Operationen auf diesen Typen werden in der Regel durch elementare Hardware-Instruktionen realisiert.

Zusammengesetze Typen

Mit folgenden Typkonstruktoren lassen sich aus elementaren Datentypen zusammengesetzte Typen (compound types) bilden.

- `ARRAY` : Feld
- `RECORD` : Datensatz, Verbund, Struktur
- `SET` : Menge

Diese Typen erlauben direkten Zugriff auf ihre Komponenten, d.h. der Aufwand ist konstant, und nicht, wie z.B. bei verketteten Listen (siehe Abs. 1.3.2), von der Größe der Struktur abhängig.

Zeigertypen (Pointer)

Unter einem Zeiger versteht man eine Referenz, die auf eine andere Variable verweist. Dabei handelt es sich de facto um eine Variable, deren Inhalt die Speicheradresse der *referenzierten Variable* ist. Der Typ der referenzierten Variablen heißt *Bezugstyp*. Die Verwendung von Zeigern erlaubt es, *dynamische Datenstrukturen* anzulegen, deren Speicher erst bei Bedarf angefordert wird.

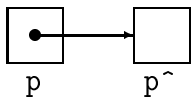
Erzeugung eines Pointers

Eine Zeigervariable wird angelegt, wenn eine Variable als Zeiger auf einen Bezugstyp deklariert wird. In Modula-3 dient dazu das Schlüsselwort `REF`.

```
TYPE ZeigerTyp :  REF BezugsTyp ;
VAR p :  ZeigerTyp ;
```

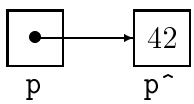
Erzeugung einer Variablen vom Bezugstyp
Die Funktion `NEW` alloziert dynamisch, d.h. zur Laufzeit, Speicherplatz für eine Variable vom Bezugstyp und liefert die entsprechende Adresse zurück. Der Wert der referenzierten Variablen `p^` ist noch unbestimmt.

```
p := NEW(ZeigerTyp) ;
```



Dereferenzierung
Zugriff auf die referenzierte Variable erfolgt mit dem Dereferenzierungs-Operator `^`:

```
p^ := 42 ;
```



Zuweisungen an einen Pointer

`p := q`	`p` und `q` müssen denselben Bezugstyp haben.
`p := NIL`	Die Konstante `NIL` repräsentiert den Fall, daß ein Zeiger auf keine Variable zeigt.

Freigabe einer Variablen vom Bezugstyp
In Modula-3 wird eine mit `NEW` erzeugte Variable automatisch entfernt, wenn kein Zeiger mehr auf sie verweist (*gargabe collection*). In vielen andere Sprachen (Pascal, Modula-2, C) muss man den belegten Speicher dagegen explizit freigeben. In Pascal geschieht dies z.B. mit der Prozedur `dispose`.

Benutzerdefinierte Datentypen

Mit Hilfe der Konstruktoren `ARRAY` und `RECORD` sowie der Zeiger lassen sich weitere, komplexere Datentypen konstruieren. Einige wichtige davon, die wir in den nächsten Abschnitten besprechen werden, sind die folgenden:

list: Liste, Sequenz, Folge

stack: Stapel, Keller, LIFO-Memory (last-in, first-out)

queue: Schlange, FIFO-Memory (first-in, first-out),

tree: Baum

Die genannten Datenstrukturen werden in der Regel dazu verwendet, Elemente eines bestimmten Typs zu speichern. Man spricht daher auch von *Container-Typen*. Den Typ

des enthaltenen Elements wollen wir nicht festlegen und verwenden in den folgenden Beispielen die Bezeichnung `ItemType`, die mittels einer Deklaration folgenden Typs mit Inhalt zu füllen ist:

```
TYPE ItemType = ... ;
```

Die Beispielprogramme wurden zum großen Teil [Sedgewick 93] entnommen.

1.3.2 Listen

Eine Liste stellt eine Sequenz oder Folge von Elementen $a_1, \ldots, a_n$ dar. Listen werden oft als Verkettung von Records mittels Zeigern (*linked list*) implementiert. Wenn eine Liste nicht verzweigt ist, nennt man sie auch *lineare* Liste. Die folgenden Operationen sind auf einer Liste definiert:

insert : Einfügen an beliebiger Position.

delete : Entfernen an beliebiger Position.

read : Zugriff auf beliebige Position.

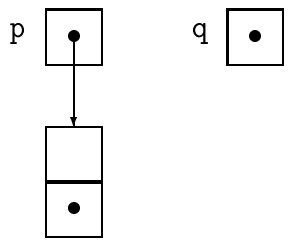
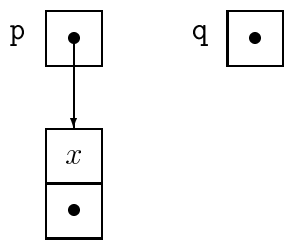
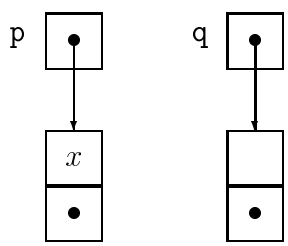
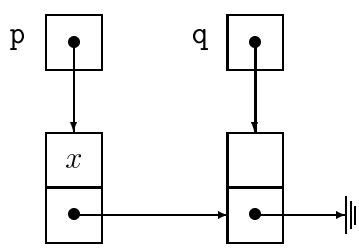
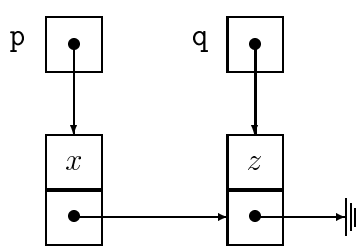
Bei der Implementierung kann man je nach Zielsetzung mit Pointern oder mit Arrays arbeiten.

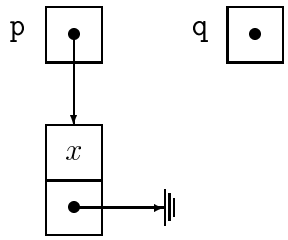
Zeiger-Implementierung von Listen (verkettete Liste)

In der Pointer-Implementierung wird eine Liste aus mehreren *Knoten* aufgebaut. Jeder Knoten enthält den Inhalt des Listenelements (key) und einen Zeiger auf den nachfolgenden Knoten (next).

```
TYPE link = REF node ;
     node = RECORD
         key  : ItemType ;
         next : link ;
     END ;
```

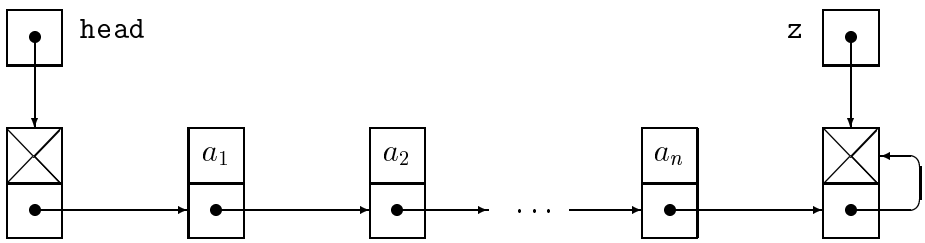
Beispiel:

Anweisung	**Wirkung**
`VAR p, q : link ;`	
`p := NEW(link) ;`	
`p^.key :=` x `;`	
`q := NEW(link) ;`	
`p^.next := q ;` `q^.next := NIL ;`	
`q^.key :=` z `;`	

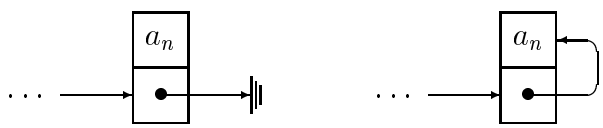
Anweisung	**Wirkung**
`p^.next := NIL ;` `q := NIL ;`	

Schematische Darstellung

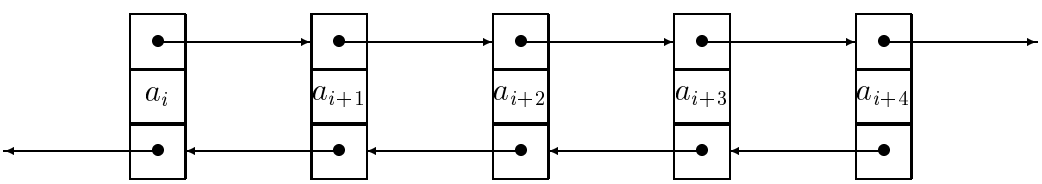
Eine einfach verkettete Liste sieht so aus:



Bei der Darstellung des Anfangs und Endes der Liste gibt es einige Varianten. In der hier vorgestellten Variante werden zwei zusätzliche Knoten `head` und `z` verwendet, die Anfang bzw. Ende der Liste repräsentieren und selbst kein Listenelement speichern. Andere Implementierungen verzichten auf ein explizites Kopf-Element. Beim Listenende sind auch folgende Varianten üblich:



Eine *doppelt verkettete* Liste lässt sich so darstellen:



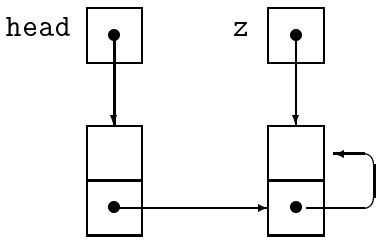
Andere Varianten sind *zyklische Listen* und *Skip-Listen*.

Programme

```
VAR head, z: link ;
```

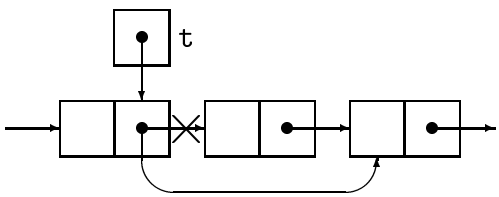
leere Liste anlegen

```
PROCEDURE ListInitialize() =
BEGIN
    head := NEW(link) ;
    z    := NEW(link) ;
    head^.next := z ;
    z^.next    := z ;
END ListInitialize ;
```



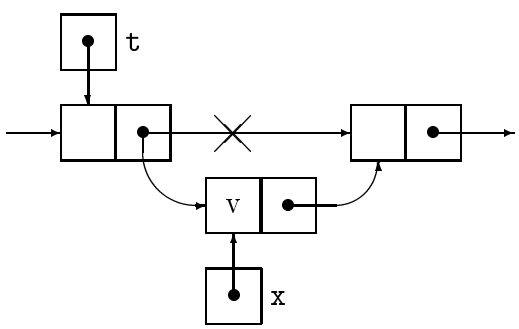
Nachfolger von `t^` löschen

```
PROCEDURE DeleteNext(t : link) =
BEGIN
    t^.next := t^.next^.next ;
END DeleteNext ;
```



Nachfolger von `t^` einfügen

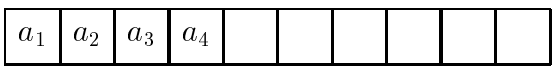
```
PROCEDURE InsertAfter(v : ItemType ;
                      t : link) =
VAR x : link ;
BEGIN
    x := NEW(link) ;
    x^.key := v ;
    x^.next := t^.next ;
    t^.next := x ;
END InsertAfter ;
```



Array-Implementierung von Listen

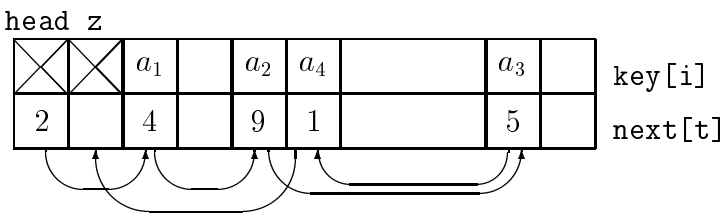
Zwei auf Arrays basierende Implementierungen der linearen Liste sind das sequentielle Array und die Cursor-Darstellung.

Sequentielles Array



Cursor-Darstellung

Die Cursor-Darstellung ist an die Zeiger-Implementierung angelehnt und verwendet ein zweites Array für die Indizes der Nachfolgeelemente.



Die Listenmethoden `InsertAfter` und `DeleteNext` lassen sich in der Cursor-Darstellung wie folgt implementieren. Dabei ist `z` wieder das Listenende; `x` bezeichnet die nächste freie Position im Array (sehr primitive Speicherverwaltung).

```
VAR key  : ARRAY [0..N] OF ItemType ;
    next : ARRAY [0..N] OF CARDINAL ;
    x, head, z : CARDINAL ;

PROCEDURE ListInitialize() =
BEGIN
    head := 0 ; z := 1 ; x := 1 ;
    next[head] := z ; next[z] := z ;
END ListInitialize ;

PROCEDURE DeleteNext(t : INTEGER) =
BEGIN
    next[t] := next[next[t]] ;
END DeleteNext ;
```

```
PROCEDURE InsertAfter(v : ItemType ; t : INTEGER) =
BEGIN
    x := x + 1 ;
    key[x] := v ;
    next[x] := next[t] ;
    next[t] := x ;
END InsertAfter ;
```

Anmerkungen:

- `InsertAfter` testet nicht ob das Array schon voll ist.
- Mit `DeleteNext` freigegebene Positionen im Array werden nicht wiederverwendet. (Dafür müßte zusätzlich eine Freispeicherliste verwaltet werden.)

Vor- und Nachteile der verschiedenen Implementierungen

sequentielles Array:

- vorgegebene maximale Größe
- sehr starr: Der Aufwand für Insert und Delete wächst linear mit der Größe der Liste.
- Daher nur bei „statischer“ Objektmenge zu empfehlen, d.h. wenn relativ wenige Insert- und Delete-Operationen nötig sind.
+ belegt keinen zusätzlichen Speicherplatz

Cursor-Darstellung:

- vorgegebene maximale Größe
- spezielle Freispeicherverwaltung erforderlich
+ Fehlerkontrolle leichter

Pointer-Implementierung:

+ maximale Anzahl der Elemente offen
+ Freispeicherverwaltung wird vom System übernommen
- Overhead durch Systemaufrufe
- Fehlerkontrolle schwierig

1.3.3 Stacks

Einen Stack kann man sich als Stapel von Elementen vorstellen (siehe Abb. 1.4). Stacks speichern (so wie Listen) Elemente in sequentieller Ordnung, der Zugriff ist jedoch auf das erste Element der Folge beschränkt. Die zulässigen Operationen sind:

Push : Neues Element wird am oberen Ende (Top) hinzugefügt.

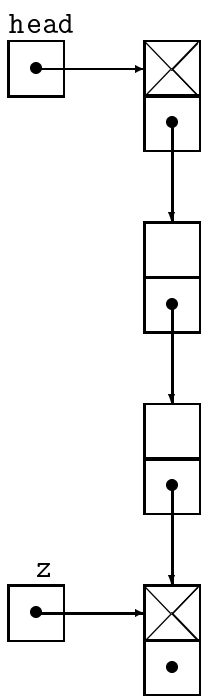
Pop : Oberstes Element wird vom Stack entfernt und zurückgeliefert.

Da das von Pop zurückgelieferte Element immer dasjenige ist, das *zuletzt* eingefügt wurde, spricht man auch von einem LIFO-Memory (last-in, first-out). Andere Bezeichnungen für „Stack“ sind „Stapel“, „Kellerspeicher“ und „pushdown stack“.

Implementierung mit Pointern

Der Stack verwendet die gleiche Datenstruktur wie die Liste, nur die Operationen unterscheiden sich folgendermaßen:

```
PROCEDURE Push(v : ItemType) =
VAR t : link ;
BEGIN
    t := NEW(link) ;
    t^.key := v ;
    t^.next := head^.next ;
    head^.next := t ;
END Push ;

PROCEDURE Pop() : ItemType =
VAR t : link ;
BEGIN
    t := head^.next ;
    head^.next := t^.next ;
    RETURN t^.key ;
END Pop ;

PROCEDURE StackEmpty() : BOOLEAN =
BEGIN
    RETURN head^.next = z ;
END StackEmpty ;
```



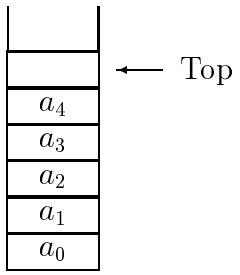


Abbildung 1.4: Schematische Darstellung eines Stacks

Implementierung mit Arrays

```
CONST stack_max = 100 ;
VAR   stack : ARRAY [0..stack_max] OF ItemType ;
      top : CARDINAL ;

PROCEDURE StackInitialize() =
BEGIN top := 0 ;
END StackInit ;

PROCEDURE Push(v : ItemType) =
BEGIN
    stack[top] := v ;
    top := top + 1 ;
END Push ;

PROCEDURE Pop() : ItemType =
BEGIN
    top := top - 1 ;
    RETURN stack[top] ;
END Pop ;

PROCEDURE StackEmpty() : BOOLEAN =
BEGIN
    RETURN top = 0 ;
END StackEmpty ;

PROCEDURE StackFull() : BOOLEAN =
BEGIN
    RETURN top > stack_max ;
END StackFull ;
```

Beachte: Über- und Unterlauf des Stacks müssen abgefangen werden.

1.3.4 Queues

Geläufige Namen für „Queue“ sind auch „Warteschlange“ und „Ringspeicher“. Bei der Queue ist das Einfügen und Entfernen von Elementen nur wie folgt möglich:

Put : Neues Element wird am Ende der Queue hinzugefügt.

Get : Vorderstes Element wird aus der Queue entfernt und zurückgeliefert.

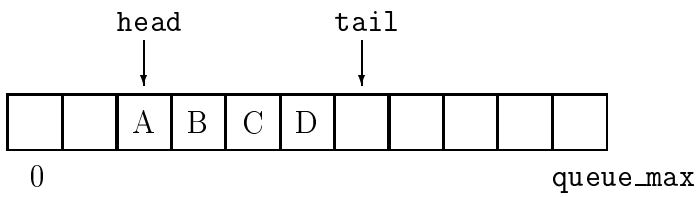
Da das von Get zurückgelieferte Element immer dasjenige ist, das *zuerst* eingefügt wurde, spricht man auch von einem FIFO-Memory (first-in, first-out).

Implementierung als Array:

```
CONST queue_max = 100 ;
VAR   queue : ARRAY [0..queue_max] OF ItemType ;
      head, tail : CARDINAL ;

PROCEDURE QueueInitialize() =
BEGIN
    head := 0 ;
    tail := 0 ;
END QueueInitialize ;

PROCEDURE Put(v : ItemType) =
BEGIN
    queue[tail] := v ;
    tail := tail + 1 ;
    IF tail > queue_max THEN tail := 0 ; END ;
END Put ;

PROCEDURE Get() : ItemType =
VAR t : ItemType ;
BEGIN
    t := queue[head] ;
    head := head + 1 ;
    IF head > queue_max THEN head := 0 ; END ;
    RETURN t ;
END Get ;

PROCEDURE QueueEmpty() : BOOLEAN =
BEGIN
    RETURN head = tail ;
END QueueEmpty ;
```

```
PROCEDURE QueueFull() : BOOLEAN =
BEGIN
    RETURN head = (tail+1) MOD (queue_max+1) ;
END QueueFull ;
```

Beachte: Über- und Unterlauf der Queue müssen abgefangen werden.



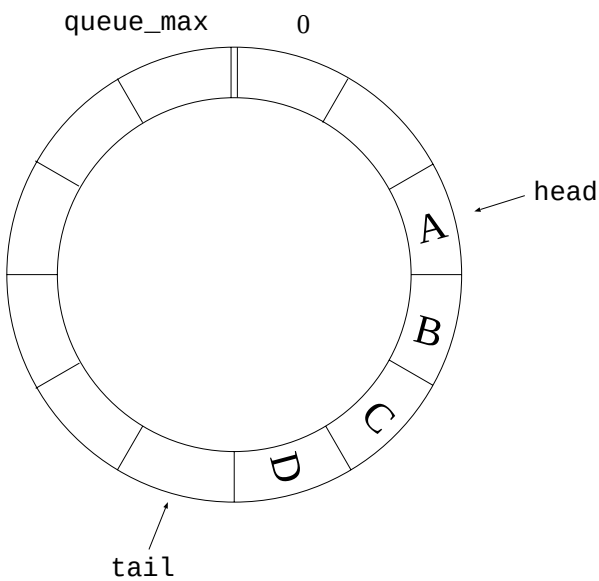


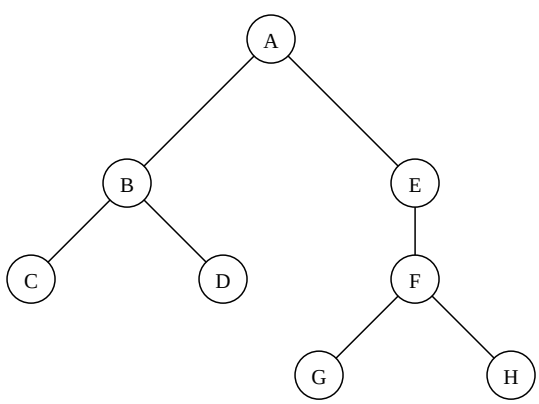
Abbildung 1.5: Schematische Darstellung eines Ringpuffers (Queue)

1.3.5 Bäume

Ein *Baum* besteht aus einer Menge von Knoten und einer Relation, die über der Knotenmenge eine Hierarchie definiert:

- Jeder Knoten, außer dem Wurzelknoten (*Wurzel*, root), hat einen unmittelbaren Vorgänger (*Vater*, parent, father).
- Eine *Kante* (Zweig, Ast, branch) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus.
- Der *Grad* eines Knotens ist die Zahl seiner unmittelbaren Nachfolger.
- Ein *Blatt* (Blattknoten, leaf) ist ein Knoten ohne Nachfolger, also mit Grad 0. Mit anderen Worten: außer den Blattknoten hat jeder Knoten mindestens einen unmittelbaren Nachfolger.
- *Siblings* (*Brüder*, Geschwister) sind alle unmittelbaren Nachfolger des gleichen Vaterknotens.
- Ein *Pfad* (Weg) ist eine Folge von Knoten $n_1, \ldots, n_k$, wobei n_i unmittelbarer Nachfolger von n_{i-1} ist. (Es gibt keine Zyklen.)
  Die *Länge eines Pfades* ist die Zahl seiner Kanten = die Zahl seiner Knoten -1.
- Die *Tiefe* oder auch *Höhe eines Knotens* ist die Länge des Pfades von der Wurzel zu diesem Knoten.
- Die *Höhe eines Baumes* ist die Länge des Pfades von der Wurzel zum tiefsten Blatt.
- Der *Grad eines Baumes* ist das Maximum der Grade seiner Knoten.
  Spezialfall mit Grad = 2: Binärbaum.

Beispiel

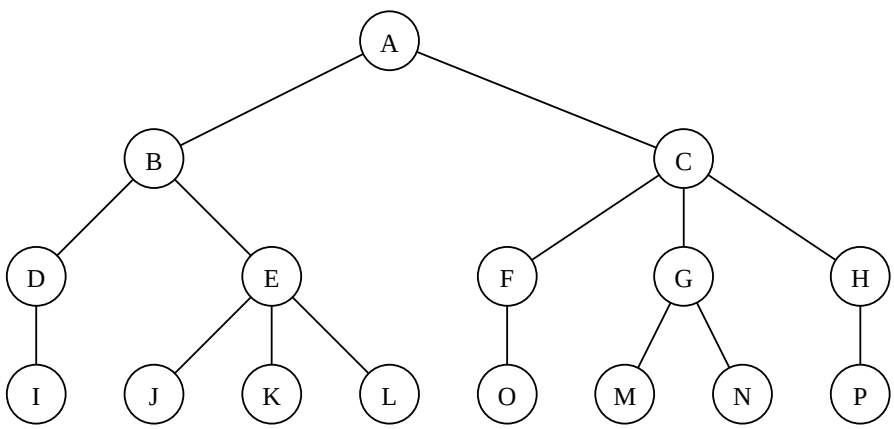


Wurzel	:	A
Vater	:	A ist Vater von E und B
Brüder	:	B und E sind Brüder
Blätter	:	C, D, G, H
Tiefe	:	F hat Tiefe 2
Höhe des Baumes	:	3
Pfad von A nach G	:	(A, E, F, G)

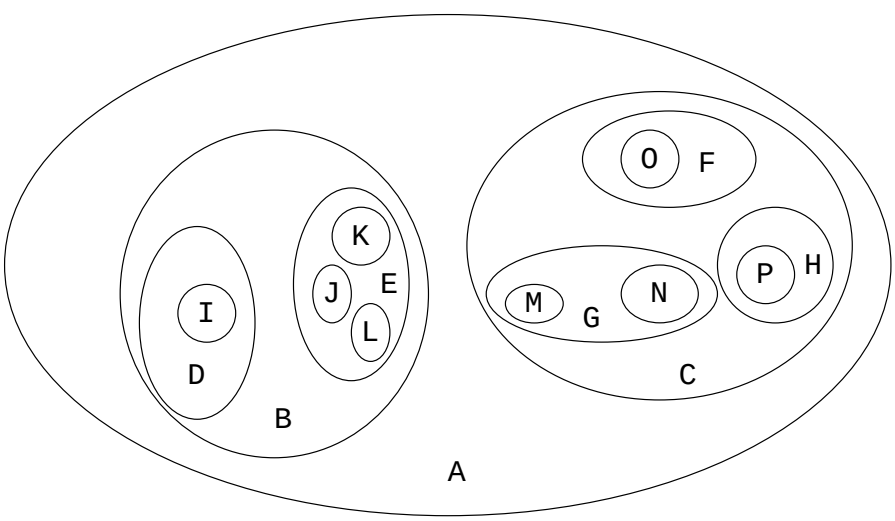
Graphische Darstellungen

Bäume können auf unterschiedliche Arten dargestellt werden. Die vier folgenden Darstellungen beschreiben dieselbe hierarchische Struktur:

1. Darstellung als Graph



2. geschachtelte Mengen



3. Darstellung durch Einrückung

```
A
    B
        D
            I
        E
            J
            K
            L
    C
        F
            O
        G
            M
            N
        H
            P
```

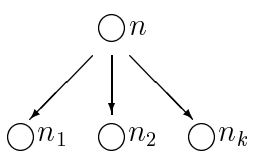
4. geschachtelte Klammern

(A(B(D(I),E(J,K,L)),C(F(O),G(M,N),H(P))))

Induktive Definition

1. Ein einzelner Knoten ist ein Baum. Dieser Knoten ist gleichzeitig der Wurzelknoten.

2. Sei n ein Knoten und seien $T_1, T_2, ..., T_k$ Bäume mit den zugehörigen Wurzelknoten $n_1, n_2, ..., n_k$. Dann wird ein neuer Baum konstruiert, indem $n_1, n_2, ..., n_k$ zu unmittelbaren Nachfolgern von n gemacht werden. T_k heißt dann k-ter Teilbaum (Unterbaum) des Knotens n.



Minimale und maximale Höhe eines Baumes

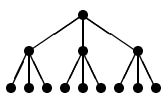
Sei T ein Baum vom Grad $d \geq 2$ und n Knoten, dann gilt:

- maximale Höhe $= n - 1$
- minimale Höhe $= \lceil \log_d (n(d-1)+1) \rceil - 1 \leq \lceil \log_d n \rceil$,

Beweis

maximale Höhe: klar, da es in T nur n Knoten gibt, es also maximal $n-1$ Kanten geben kann. Ein Baum mit maximaler Höhe ist zu einer linearen Liste entartet.

minimale Höhe: Ein *maximaler Baum* ist ein Baum, der bei gegebener Höhe h und gegebenem Grad d die maximale Knotenzahl $N(h)$ hat.

1 Knoten	
d Knoten	
d^2 Knoten	
$\vdots$	$\vdots$

Also gilt für die maximale Knotenzahl $N(h)$:

$$\begin{aligned} N(h) &= 1 + d + d^2 + ... + d^h \quad \text{(geometrische Reihe)} \\ &= \frac{d^{h+1}-1}{d-1} \end{aligned}$$

Bei fester Knotenzahl hat ein maximal gefüllter Baum minimale Höhe. Ein Baum, dessen Ebenen bis auf die letzte komplett gefüllt sind, hat offensichtlich minimale Höhe. Also liegt die Anzahl n der Knoten, die man in einem Baum gegebener minimaler Höhe h unterbringen kann, zwischen der für einen maximalen Baum der Höhe h und einem maximalen Baum der Höhe $h-1$.

$$\begin{aligned} N(h-1) &< n &\leq N(h) \\ \frac{d^h-1}{d-1} &< n &\leq \frac{d^{h+1}-1}{d-1} \\ d^h &< n(d-1)+1 &\leq d^{h+1} \\ h &< \log_d (n(d-1)+1) &\leq h+1 \end{aligned}$$

und wegen $n(d-1)+1 < nd \;\; \forall n > 1$ folgt:

$$\begin{aligned} h &= \lceil \log_d (n(d-1)+1) \rceil - 1 \\ &\leq \lceil \log_d (nd) \rceil - 1 \\ &= \lceil \log_d n \rceil \end{aligned}$$

Spezialfall $d = 2$ (Binärbaum):

$$h = \lceil \mathrm{ld}(n+1) \rceil - 1 \leq \lceil \mathrm{ld}\, n \rceil$$

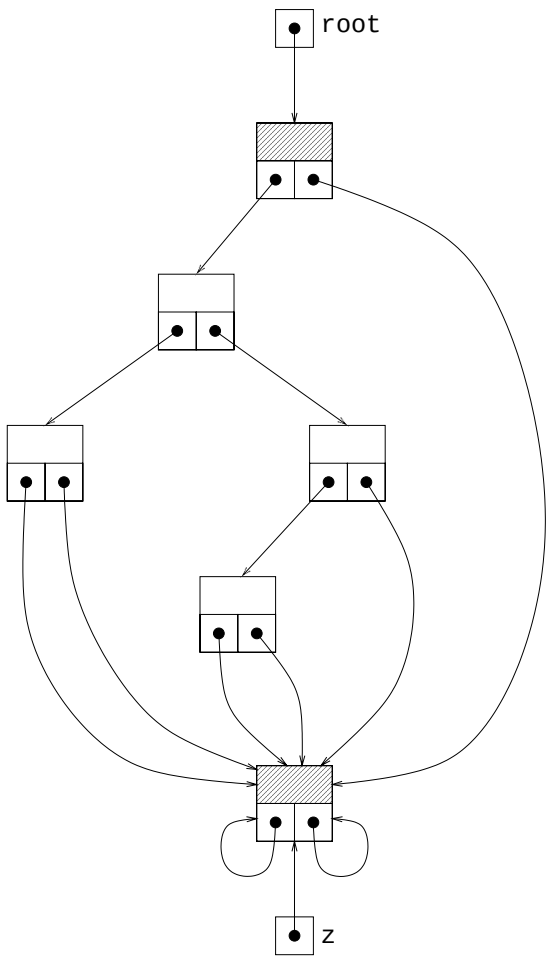
Implementierung von Binärbäumen

Zwei geläufige Implementierungen von Binärbäumen sind die Pointer-Darstellung und die Array-Einbettung.

Pointer-Realisierung

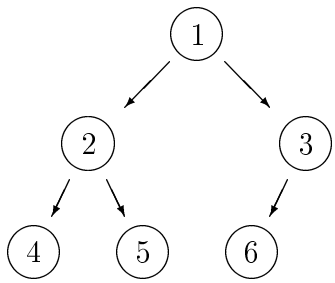
```
TYPE node = REF RECORD
                key : ItemType ;
                left, right : node ;
            END ;
```

Zur Kennzeichnung der Wurzel und der Blätter werden (analog zur Implementierung der verketteten Listen) zwei spezielle Knoten `root` und `z` verwendet.



Array-Einbettung

Die Array-Einbettung eigent sich am besten zur Darstellung vollständiger Bäume. Ein Binärbaum heißt *vollständig*, wenn alle Ebenen bis auf die letzte vollständig besetzt sind, und die letzte Ebene von links nach rechts aufgefüllt ist (auch: links-vollständig).



Numeriert man die Knoten wie hier gezeigt ebenenweise durch, so lassen sich die Positionen von Vorgängern und Nachfolgern auf folgende Weise einfach berechnen:

Vorgänger: $\text{Pred}(n) = \left\lfloor \frac{n}{2} \right\rfloor$

Nachfolger: $\text{Succ}(n) = \begin{cases} 2n & \text{linker Sohn} \\ 2n+1 & \text{rechter Sohn} \end{cases}$ (falls diese existieren)

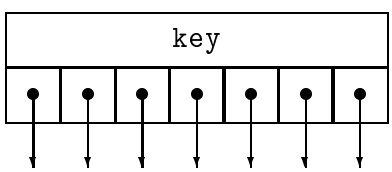
Man prüft die Existenz der Nachfolger über die Arraygrenzen.

Implementierung von Bäumen mit Grad $d > 2$

Array von Pointern

Darstellung durch Knoten, die ein Array von Pointern auf ihre Nachfolgeknoten enthalten.

```
TYPE node = REF RECORD
                key  : ItemType ;
                sons : ARRAY [1..d] OF node ;
            END ;
```



Nachteile

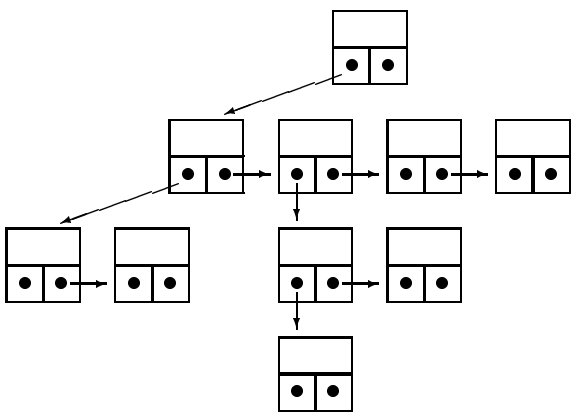
- der maximale Grad ist vorgegeben
- Platzverschwendung, falls der Grad der einzelnen Knoten stark variiert.

Darstellung durch Pseudo-Binärbaum

Strategie: Geschwister sind untereinander zu einer linearen Liste verkettet. Jeder Knoten verweist nur auf sein erstes Kind und seinen rechten Bruder (leftmost-child, right-sibling).

```
TYPE node = REF RECORD
                key : ItemType ;
                leftmostchild : node ;
                rightsibling  : node ;
            END ;
```

`leftmostchild` zeigt also auf das erste Element einer verketteten Liste der Geschwister mit den Nachfolgezeigern `rightsibling`.



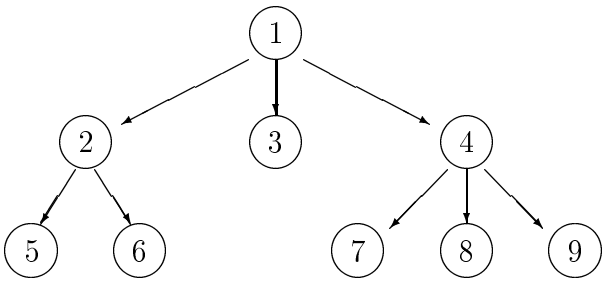
Array-Einbettung

Die Knoten werden ebenenweise von links nach rechts durchnumeriert (*breadth first*) und unter diesen Indizes in einem Array abgelegt. Geschwisterknoten stehen dann an aufeinanderfolgenden Positionen. Zum Auffinden der Vorgänger und Nachfolger dienen folgende Arrays:

```
VAR Parent,
    LeftMostChild,
    RightMostChild : ARRAY [1..N] OF CARDINAL ;
```

Der Index 0 wird für nicht vorhandene Knoten verwendet.

Beispiel



i	`Parent`	`LeftMostChild`	`RightMostChild`
1	0	2	4
2	1	5	6
3	1	0	0
4	1	7	9
5	2	0	0
6	2	0	0
7	4	0	0
8	4	0	0
9	4	0	0

Durchlauf-Funktionen sind in dieser Darstellung sehr einfach:

- Durchlaufen aller Knoten: $i = 1, \ldots, n$
- Durchlaufen der Nachfolger eines Knotens i:
  $k =$ `LeftMostChild[`i`]`, ..., `RightMostChild[`i`]`

Durchlaufen eines Baums

Für viele Anwendungen ist ein kompletter Durchlauf aller Knoten eines Baumes notwendig (Durchmusterung, Traversierung, tree traversal).

Tiefendurchlauf (*depth first*): Zu einem Knoten n werden rekursiv seine Teilbäume $n_1, \ldots, n_k$ durchlaufen. Je nach Reihenfolge, in der der Knoten und seine Teilbäume betrachtet werden, ergeben sich:

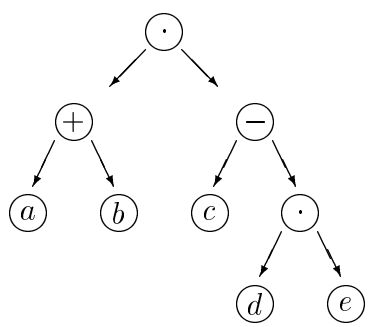
1. Preorder (Prefixordnung, Hauptreihenfolge):
   - betrachte n
   - durchlaufe $n_1 ... n_k$
2. Postorder (Postfixordnung, Nebenreihenfolge):
   - durchlaufe $n_1 ... n_k$
   - betrachte n
3. Inorder (Infixordnung, symmetrische Reihenfolge):
   - durchlaufe n_1
   - betrachte n
   - durchlaufe $n_2 ... n_k$

   Diese Ordnung ist nur für Binärbäume gebräuchlich.

Breitendurchlauf (*breadth first*, *level order*): Knoten werden ihrer Tiefe nach gelistet, und zwar bei gleicher Tiefe von links nach rechts.

Beispiel: Operatorbaum

Bei Anwendung auf einen Operatorbaum realisieren die unterschiedlichen Traversierungsarten die Darstellung des entsprechenden algebraischen Ausdrucks in verschiedenen Notationen. Beispiel: Der Ausdruck $(a+b)\cdot(c-d\cdot e)$ wird durch folgenden Operatorbaum repräsentiert:



Ausgabe aller Knoten mittels

- Preorder-Traversierung: $\cdot + ab - c \cdot de$
  (Prefixnotation, „polnische Notation“)
- Postorder-Traversierung: $ab + cde \cdot - \cdot$
  (Postfixnotation, „umgekehrte polnische Notation“)
- Inorder-Traversierung: $a + b \cdot c - d \cdot e$.
  (Infixnotation)

Rekursiver Durchlauf für Binärbäume
Wir wollen die genannten Tiefendurchläufe für Binärbäume in Pointer-Darstellung implementieren.

Preorder-Durchlauf

```
PROCEDURE Traverse(t : node) =
BEGIN
    IF t <> z THEN
        visit(t) ;
        traverse(t^.left) ;
        traverse(t^.right) ;
    END ;
END Traverse ;
```

Postorder-Durchlauf

```
PROCEDURE Traverse(t : node) =
BEGIN
    IF t <> z THEN
        traverse(t^.left) ;
        traverse(t^.right) ;
        visit(t) ;
    END ;
END Traverse ;
```

Inorder-Durchlauf

```
PROCEDURE Traverse(t : node) =
BEGIN
    IF t <> z THEN
        traverse(t^.left) ;
        visit(t) ;
        traverse(t^.right) ;
    END ;
END Traverse ;
```

Nicht-rekursiver Durchlauf für Binärbäume

Preorder-Durchlauf (benötigt einen Stack)

```
PROCEDURE Traverse(t : node) =
VAR n : node ;
BEGIN
    Push(t) ;
    REPEAT
        n := Pop() ;
        IF n <> z THEN
            visit(n) ;
            Push(n^.right) ;
            Push(n^.left) ;
        END ;
    UNTIL StackEmpty() ;
END Traverse ;
```

Man beachte, daß zuerst der *rechte* Teilbaum auf den Stack geschoben wird.

Level-Order-Durchlauf (benötigt eine Queue)

```
PROCEDURE Traverse(t : node) =
VAR n : node ;
BEGIN
    Put(t) ;
    REPEAT
        n := Get() ;
        IF n <> z THEN
            visit(n) ;
            Put(n^.left) ;
            Put(n^.right) ;
        END ;
    UNTIL QueueEmpty() ;
END Traverse ;
```

1.3.6 Abstrakte Datentypen (ADT)

Wenn man eine Datenstruktur nur durch die auf ihr zugelassenen Operationen definiert, und die spezielle Implementierung ignoriert, dann gelangt man zu einem sogenannten *abstrakten* Datentyp. Ein abstrakter Datentyp (ADT) wird definiert durch:

1. eine Menge von Objekten (Elementen)
2. Operationen auf diesen Elementen (legen die Syntax eines Datentyps fest)
3. Axiome (Regeln) (definieren die Semantik des Datentyps)

Die Abstraktion von der konkreten Implementierung erleichtert die Analyse von Algorithmen erheblich.

Die Idee des abstrakten Datentyps wird in Modula-3 durch die Trennung in sichtbare Schnittstellen (`INTERFACE`) und verborgene Implementierung (`MODULE`) unterstützt. Dies erhöht auch die Übersichtlichkeit von umfangreichen Programmpaketen.

Weiterführende Literatur: [Güting, Seiten 21–23], [Sedgewick 88, Seite 31]

Beispiel: Algebraische Spezifikation des Stacks

- Sorten (Datentypen)
  Stack (hier zu definieren)
  Element („ElementTyp“)

- Operationen

StackInit:			$\rightarrow$	Stack	(Konstante)
StackEmpty:		Stack	$\rightarrow$	Boolean	
Push:	Element $\times$	Stack	$\rightarrow$	Stack	
Pop:		Stack	$\rightarrow$	Element $\times$ Stack	

- Axiome
  x: Element („ElementTyp“)
  s: Stack

$$\begin{aligned}
\text{Pop}(\text{Push}(x,s)) &= (x,s) \\
\text{Push}(\text{Pop}(s)) &= s \qquad \text{für StackEmpty}(s) = \text{FALSE} \\
\text{StackEmpty}(\text{StackInit}) &= \text{TRUE} \\
\text{StackEmpty}(\text{Push}(x,s)) &= \text{FALSE}
\end{aligned}$$

Realisierung [Sedgewick 88, Seite 27].

Anmerkung: Situationen, in denen undefinierte Operationen, wie z.B. Pop(StackInit), aufgerufen werden, erfordern eine gesonderte Fehlerbehandlung.

Potentielle Probleme von ADTs

- Bei komplexen Fällen wird die Anzahl der Axiome sehr groß.
- Die Spezifikation ist nicht immer einfach zu verstehen.
- Es ist schwierig zu prüfen, ob die Gesetze vollständig und widerspruchsfrei sind.

Wir werden im weiteren aus folgenden Gründen nicht näher auf die abstrakte Spezifikation von Datentypen eingehen:

- Wir betrachten hier nur relativ kleine und kompakte Programme.
- Uns interessieren gerade die verschiedenen möglichen Implementierungen, die durch die abstrakte Spezifikation verborgen werden sollen.
- Wir wollen die Effizienz der konkreten Implementierung untersuchen, wobei ADTs gerade nicht helfen.

1.4 Entwurfsmethoden

Für den guten Entwurf kann man kaum strenge Regeln angeben, aber es gibt einige Prinzipien, die man je nach Zielsetzung, einsetzen kann, und die unterschiedliche Vorzüge haben. Wir werden folgende typische Methoden vorstellen:

1. Divide-and-Conquer
2. Dynamische Programmierung

1.4.1 Divide-and-Conquer-Strategie

Die Divide-and-Conquer-Strategie wird häufig benutzt, weil sie vielseitig ist und auf fast jedes Problem angewendet werden kann, das sich in kleinere Teilprobleme zerlegen läßt. Die Strategie besagt:

- **Divide**: Zerlege das Problem in Teilprobleme, solange bis es „elementar“ bearbeitet werden kann.
- **Conquer**: Setze die Teillösungen zur Gesamtlösung zusammen (merge).

Nach dieser Strategie entwickelte Algorithmen legen oft eine rekursive Implementierung nahe (gutes Beispiel: Baum-Durchlauf; schlechtes Beispiel: Fibonacci-Zahlen). Bei der Laufzeitanalyse solcher Algorithmen gelangt man meist zu Rekursionsgleichungen. Diese werden in Abschnitt 1.5 näher behandelt.

Beispiel: Gleichzeitige Bestimmung von MAX und MIN einer Folge

```
0    procedure MAXMIN (S);     [Aho et al.  74, Seite 61]
1    if ||S||=2 then
2     let S={a,b};
3     return (MAX (a, b), MIN (a, b));
4   else
5       divide S into two subsets S1 and S2, each with half the elements;
6       (max1,min1) ← MAXMIN(S1);
7       (max2,min2) ← MAXMIN(S2);
8       return (MAX (max1, max2), MIN (min1, min2));
9     end;
10  end MAXMIN;
```

Sei $T(n)$ die Zahl der Vergleiche, dann gilt

$$T(n) = \begin{cases} 1 & n = 2 \\ 2 \cdot T(\frac{n}{2}) + 2 & n > 2 \end{cases}$$

Erklärung:

$n = 2$	:	ein Vergleich (Zeile 3)
$n > 2$	:	zwei Aufrufe von MAXMIN mit $\frac{n}{2}$-elementigen Mengen (Zeilen 6 und 7) und zwei zusätzliche Vergleiche (Zeile 8)

Behauptung: $T(n) = \dfrac{3}{2}n - 2$

Verifikation:

$$\begin{aligned} T(2) &= \frac{3}{2} \cdot 2 - 2 = 1 \qquad \surd \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 &= 2 \cdot \left(\frac{3}{2} \cdot \frac{n}{2} - 2\right) + 2 \\ &= \frac{3}{2}n - 2 = T(n) \qquad \surd \end{aligned}$$

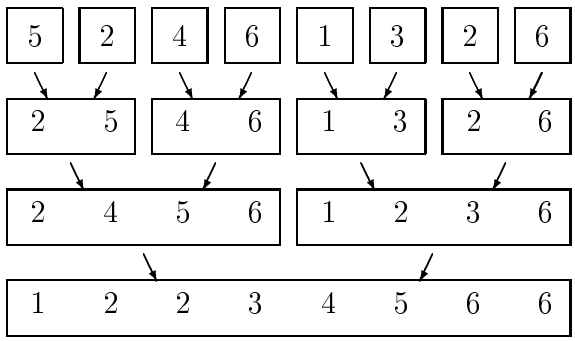
Zum Vergleich: Ein Algorithmus, der Minimum und Maximum separat bestimmt, benötigt $T(n) = 2n - 2$ Vegleiche.

Beispiel: MergeSort

MergeSort ist ein einfaches Beispiel für ein nach der Divide-and-Conquer-Strategie entworfenes Sortierverfahren. Sei $F = a_1, \ldots, a_n$ eine Folge von Zahlen, dann funktioniert MergeSort so:

1. Teile F in zwei etwa gleichgroße Folgen $F_1 = a_1, \ldots, a_{\lceil n/2 \rceil}$ und $F_2 = a_{\lceil n/2 \rceil + 1}, \ldots, a_n$. (divide)
2. Sortiere F_1 und F_2 mittels MergeSort, falls $|F_1| > 1$ bzw. $|F_2| > 1$. (conquer)
3. Verschmelze F_1 und F_2 zu einer sortierten Folge. (merge)

Wir veranschaulichen das MergeSort-Verfahren (genauer „bottom-up“- oder auch „2-way-straight“-MergeSort) an einem Beispiel:



Sei $T(n)$ die Zahl der Operationen (im wesentlichen Vergleiche), dann gilt:

$$T(n) = \begin{cases} c_1 & \text{, für } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n & \text{, für } n > 1 \end{cases}$$

für geeignete Werte $c_1, c_2 > 0$. Dabei charakterisiert c_1 den Aufwand für die Lösung des trivialen Problems ($n = 1$), und $n \cdot c_2$ denjenigen für das Verschmelzen zweier Listen.

Behauptung: $T(n) \leq (c_1 + c_2) \cdot n \operatorname{ld} n + c_1$

Beweis: Verifizieren der Rekursion mittels vollständiger Induktion von $\frac{n}{2}$ nach n.

- Induktionsanfang: $n = 1$ offensichtlich korrekt
- Induktionsschritt von $\frac{n}{2}$ nach n:

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n \\ &\leq 2\left[(c_1 + c_2)\frac{n}{2} \operatorname{ld} \frac{n}{2} + c_1\right] + c_2 n \\ &= (c_1 + c_2)n(\operatorname{ld} n - 1) + 2c_1 + c_2 n \\ &= (c_1 + c_2)n \operatorname{ld} n + 2c_1 - c_1 n \\ &= (c_1 + c_2)n \operatorname{ld} n + c_1 - c_1(n-1) \\ &\leq (c_1 + c_2)n \operatorname{ld} n + c_1 \end{aligned}$$

Exakte Analyse von MergeSort

Sei $T(n)$ die Zahl der Vergleiche. Die Aufteilung der Folge geschieht in zwei Teilfolgen mit $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Elementen.

Zur Verschmelzung werden $\lceil n/2 \rceil + \lfloor n/2 \rfloor - 1 = n - 1$ Vergleiche benötigt.

Daher ergibt sich folgende Rekursionsgleichung:

$$T(n) = \begin{cases} 0 & \text{für } n = 1 \\ n - 1 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) & \text{für } n > 1 \end{cases}$$

Für diese gilt [Mehlhorn, Seite 57]:

$$T(n) = n \cdot \lceil \operatorname{ld} n \rceil - 2^{\lceil \operatorname{ld} n \rceil} + 1$$

1.4.2 Dynamische Programmierung

Die Bezeichnung „dynamische Programmierung“ geht auf Richard Bellman (1957) zurück:

- dynamisch = sequentiell
- Programmierung = Optimierung mit Nebenbedingungen (vgl. Lineare Programmierung)

Die dynamische Programmierung kann auf Optimierungsprobleme angewandt werden, deren Lösungen sich aus den optimalen Lösungen von Teilproblemen zusammensetzen (*Bellmansches Optimalitätsprinzip*). Das Grundprinzip ist, zunächst Lösungen für „kleine“ Elementarprobleme zu finden und daraus sukzessive Lösungen für immer „größere“ Probleme zu konstruieren, bis schließlich das eigentliche Gesamtproblem gelöst ist. Diese Vorgehensweise von kleinen zu immer größeren Teillösungen charakterisiert die dynamische Programmierung als „*bottom-up*“-Strategie und unterscheidet sie vom „*top-down*“-Ansatz der Divide-and-Conquer-Methode.

Charakteristisch für die dynamische Programmierung ist es, eine Hilfsgröße zu definieren, die die Lösungen der Teilprobleme beschreibt, und für diese Größe eine Rekursionsgleichung (recurrence relation, DP equation) zu formulieren, die angibt wie die Teillösungen zusammenzusetzen sind. Konkret umfaßt ein auf dynamischer Programmierung basierender Algorithmus folgende Schritte:

- Teilprobleme bearbeiten
- **Teilergebnisse in Tabellen eintragen**
- Zusammensetzen der Gesamtlösung

Beispiel: Matrix-Kettenprodukte (matrix chain problem)

Gegeben seien zwei (reelwertige) Matrizen $B \in \mathbb{R}^{l \times m}$ und $C \in \mathbb{R}^{m \times n}$. Für die Elemente der Produktmatrix $A = B \cdot C$ gilt:

$$a_{i,k} = \sum_{j=1}^{m} b_{i,j} c_{j,k}$$

Zur Berechnung des Produkts von B und C sind offenbar $l \cdot m \cdot n$ (skalare) Multiplikationen und Additionen erfolderlich.

Bei mehreren Matrizen hängt der Wert des (Ketten-)Produkts bekanntlich nicht von der Klammerung, also davon in welcher Reihenfolge die Matrixmultiplikationen ausgewertet werden, ab (Assoziativgesetz). Der Rechenaufwand ist aber sehr wohl von der Klammerung abhängig, wie das folgende Beispiel belegt:

Konkretes Beispiel: Seien M_i, $i = 1, \ldots, 4$ reelle Matrizen, der Dimensionen 10×20, 20×50, 50×1 und 1×100. Wir betrachten den Rechenaufwand zur Berechnung des Matrizenproduktes $M_1 \cdot M_2 \cdot M_3 \cdot M_4$ gemessen in der Zahl der dafür notwendigen (skalaren) Multiplikationen für zwei verschiedene Klammerungen:

1. $$\underbrace{M_1 \cdot \underbrace{(M_2 \cdot \underbrace{(M_3 \cdot M_4)}_{[50,1,100]})}_{[20,50,100]}}_{[10,20,100]}$$

$$\begin{array}{rrl} & 5000 & \text{Operationen} \\ + & 100000 & \text{Operationen} \\ + & 20000 & \text{Operationen} \\ \hline & 125000 & \text{Operationen} \end{array}$$

2. $$\underbrace{\underbrace{(M_1 \cdot \underbrace{(M_2 \cdot M_3)}_{[20,50,1]})}_{[10,20,1]} \cdot M_4}_{[10,1,100]}$$

$$\begin{array}{rrl} & 1000 & \text{Operationen} \\ + & 200 & \text{Operationen} \\ + & 1000 & \text{Operationen} \\ \hline & 2200 & \text{Operationen} \end{array}$$

Allgemeine Problemstellung

Gegeben sei eine Folge $r_0, r_1, \ldots r_N$ von natürlichen Zahlen, die die Dimensionen von N Matrizen $M_i \in \mathbb{R}^{r_{i-1} \times r_i}$ beschreibt. Es soll das Matrix-Kettenprodukt $M_1 \cdot M_2 \cdot \ldots \cdot M_N$ berechnet werden. Das Problem besteht nun darin, die Klammerung für diesen Ausdruck zu finden, die die Anzahl der (skalaren) Multiplikationen minimiert.

Eine vollständige Suche (exhaustive search), die alle möglichen Klammerungen durchprobiert, hätte eine Komplexität, die exponentiell mit der Zahl N der Matrizen wächst.

Ansatz mittels dynamischer Programmierung

Definiere zunächst eine Hilfsgröße m_{ij} als die minimale Zahl der Operationen zur Berechnung des Teilprodukts

$$M_i \cdot M_{i+1} \cdot \ldots \cdot M_j \quad \text{mit} \quad 1 \leq i \leq j \leq N$$

Um die Rekursionsgleichung für m aufzustellen zerlegen wir dieses Produkt an der Stelle k mit $i \leq k < j$ in zwei Teile:

$$\underbrace{(M_i \cdot M_{i+1} \cdot \ldots \cdot M_k)}_{m_{i,k}} \cdot \underbrace{(M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_j)}_{m_{k+1,j}} + r_{i-1} \cdot r_k \cdot r_j \quad \text{Operationen}$$

Aufgrund der Definition von m_{ij} erhalten wir die folgende Rekursionsgleichung durch Minimierung über die Split-Stelle k:

$$m_{i,j} = \begin{cases} 0 & \text{für} \quad j = i \\ \min\limits_{i \le k < j} \{ m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j \} & \quad\quad j > i \end{cases}$$

Implementierung

Um zu einem lauffähigen Programm zu gelangen muß noch festgelegt werden, in welcher Reihenfolge die $m_{i,j}$ ausgewertet werden. Zur Berechnung des Aufwands für ein Kettenprodukt der Länge l benötigt man gemäß obiger Rekursionsgleichung nur Ergebnisse für kürzere Produkte. Daher beginnt man mit einzelnen Matrizen, berechnet dann den Aufwand für alle Produkte von zwei Matrizen, dann den für Dreierprodukte, usw.:

Längenindex	**auszuwertende Terme**	
$l = 0$	$m_{i,i}$	$\forall\, i = 1, \ldots, N$
$l = 1$	$m_{i,i+1}$	$\forall\, i = 1, \ldots, N-1$
$l = 2$	$m_{i,i+2}$	$\forall\, i = 1, \ldots, N-2$
...		
l	$m_{i,i+l}$	$\forall\, i = 1, \ldots, N-l$
...		
$l = N-2$	$m_{i,i+N-2}$	$\forall\, i = 1, 2$
$l = N-1$	$m_{1,N}$	

Dies läßt sich dann als Programm (Pseudo-Code) folgendermaßen formulieren:

```
VAR r :  ARRAY [0..N] OF INTEGER ;
    m,s :  ARRAY [1..N],[1..N] OF INTEGER ;
BEGIN
  FOR i := 1 TO N DO
    m[i,i] = 0 ;
  END ;
  FOR l := 1 TO N - 1 DO          (* Laengenindex *)
    FOR i := 1 TO N - l DO      (* Positionsindex *)
      j := i + l ;
      m[i,j] := min_{i≤k<j}   { m[i,k] + m[k+1,j] + r[i-1] · r[k] · r[j] };
      s[i,j] := arg min_{i≤k<j} { m[i,k] + m[k+1,j] + r[i-1] · r[k] · r[j] };
    END ;
  END ;
END ;
```

Ergebnisse

- `m[1,N]`: Unter diesem Eintrag in der Tabelle `m` findet man die Anzahl der minimal nötigen Operationen.
- `s[1,N]`: Unter diesem Eintrag der Tabelle `s` ist die *Split-Stelle* für die äußerste Klammerung (Matrixprodukt der größten Länge) vermerkt. Für die weitere Klammerung muß man unter den Einträgen für die entsprechenden Indexgrenzen nachschauen.
- Die Laufzeitkomplexität des vorgestellten Algorithmus ist $O(N^3)$.

1.4.3 Memoization: Tabellierung von Zwischenwerten

Die mögliche Ineffizienz direkter rekursiver Implementierung kann man oft durch Memoization vermeiden, indem man Zwischenwerte in Tabellen speichert.

Beispiel: Fibonacci-Zahlen

```
CONST N_max = 100 ;
VAR   F := ARRAY [0..N_max] OF INTEGER {-1,..} ;

PROCEDURE Fibonacci(n : CARIDNAL) : CARDINAL =
BEGIN
    IF F[n] < 0 THEN
        IF n <= 1 THEN
            F[n] := n ;
        ELSE
            F[n] := Fibonacci(n-1) + Fibonacci(n-2) ;
        END ;
    END ;
    RETURN F[n] ;
END Fibonacci ;
```

Dieses Programm hat den Vorteil, daß es die Definition der Fibonacci-Zahlen aus Abschnitt 1.2.5 genau wiederspiegelt, aber dennoch (bis auf einen konstanten Faktor) genauso effizient ist, wie die dort vorgestellte iterative Lösung.

Auch das Problem der besten Klammerung von Matrixkettenprodukten kann mittels Memoization implementiert werden (Übungsaufgabe).

1.5 Rekursionsgleichungen

Dieses Kapitel beschäftigt sich mit Rekursionsgleichungen (Differenzengleichungen) wie sie bei der Laufzeitanalyse insbesondere rekursiver Algorithmen auftreten.

1.5.1 Sukzessives Einsetzen

Viele Rekursionsgleichungen lassen sich durch *sukzessives Einsetzen* lösen. (Bei den Beispielen 2 – 5 wird der Einfachheit halber angenommen, daß n eine Zweierpotenz ist $n = 2^k,\ k \in \mathbb{N}$.)

Beispiel 1:

$$\begin{aligned} T(n) &= T(n-1) + n \quad , \quad n > 1 \qquad (T(1) = 1) \\ &= T(n-2) + (n-1) + n \\ &= \ldots \\ &= T(1) + 2 + \ldots + (n-2) + (n-1) + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Beispiel 2:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \quad , \quad n > 1 \qquad (T(1) = 0) \\ &= T\left(\frac{n}{4}\right) + 1 + 1 \\ &= T\left(\frac{n}{8}\right) + 1 + 1 + 1 \\ &= \ldots \\ &= T(1) + \operatorname{ld} n \\ &= \operatorname{ld} n \end{aligned}$$

Beispiel 3:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + n \quad , \quad n > 1 \qquad (T(1) = 0) \\ &= T\left(\frac{n}{4}\right) + \frac{n}{2} + n \\ &= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n \\ &= \ldots \\ &= T(1) + \ldots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n \\ &= 2(n-1) \qquad \text{für } n > 1 \end{aligned}$$

Beispiel 4:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \quad , \quad n > 1 \qquad (T(1) = 0) \\
\frac{T(n)}{n} &= \frac{T\left(\frac{n}{2}\right)}{n/2} + 1 \\
&= \frac{T\left(\frac{n}{4}\right)}{n/4} + 1 + 1 \\
&= \dots \\
&= \operatorname{ld} n \\
T(n) &= n \operatorname{ld} n
\end{aligned}$$

Beispiel 5:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 1 \quad , \quad n > 1 \qquad (T(1) = 1) \\
&= 2[2T\left(\frac{n}{4}\right) + 1] + 1 \\
&= 4T\left(\frac{n}{4}\right) + 3 \\
&= 8T\left(\frac{n}{8}\right) + 7 \\
&= \dots \\
&= n \cdot T(1) + n - 1 \\
&= 2n - 1 \\
\text{Verifiziere:} &= 2 \cdot (n-1) + 1 \qquad \surd
\end{aligned}$$

1.5.2 Master-Theorem für Rekursionsgleichungen

Bei der Analyse von Algorithmen, die nach der Divide-and-Conquer-Strategie entworfen wurden, treten meist Rekursionsgleichungen der folgenden Form auf:

$$T(n) = \begin{cases} 1 & \text{für} \quad n = 1 \\ a \cdot T(\frac{n}{b}) + d(n) & \quad\;\; n > 1 \end{cases}$$

mit Konstanten $a \geq 1$ und $b > 1$ und einer positiven Funktion $n \mapsto d(n)$. Statt $T(\frac{n}{b})$ kann auch $T(\lfloor \frac{n}{b} \rfloor)$ oder $T(\lceil \frac{n}{b} \rceil)$ stehen.

Dann gilt für $d(n) \in O(n^\gamma)$ mit $\gamma > 0$:

$$T(n) \in \begin{cases} O(n^\gamma) & & a < b^\gamma \\ O(n^\gamma \log_b n) & \text{für} & a = b^\gamma \\ O(n^{\log_b a}) & & a > b^\gamma \end{cases}$$

Anmerkung: Es gibt eine allgemeinere Form des Master-Theorems, die allerdings auch etwas umständlicher ist [Cormen et al., S. 62 f.].

Beweis

in drei Schritten:

a) Beweis für $d(n) = n^\gamma$ und Potenzen von b ($n = b^k$, $k \in \mathbb{N}$)

b) Erweiterung von $d(n) = n^\gamma$ auf allgemeine Funktionen $n \mapsto d(n)$

c) Erweiterung von Potenzen von b auf beliebige n.

Teil a: Für $\frac{n}{b^i}$ statt n gilt speziell für $i = 1, 2, 3, \ldots$:

$$T(\frac{n}{b^i}) = aT(\frac{n}{b^{i+1}}) + d(\frac{n}{b^i})$$

Sukzessives Einsetzen liefert:

$$\begin{aligned}
T(n) &= a \cdot T(\frac{n}{b}) + d(n) \\
&= a \cdot \left[aT(\frac{n}{b^2}) + d(\frac{n}{b})\right] + d(n) \\
&= a^2 \cdot T(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n) \\
&= a^3 \cdot T(\frac{n}{b^3}) + a^2 d(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n) \\
&= \ldots \\
&= a^k T(\frac{n}{b^k}) + \sum_{j=0}^{k-1} a^j d(\frac{n}{b^j})
\end{aligned}$$

Wir wählen einen speziellen Wert für k:

$$k_n := \log_b n, \qquad \text{d.h. } n = b^{k_n}$$

und erhalten damit

$$T(n) = a^{k_n} \cdot T(1) + \sum_{j=1}^{k_n - 1} a^j d(b^{k_n - j})$$

Wir wählen speziell:

$$n \mapsto d(n) = n^\gamma \text{ mit } \gamma > 0$$

Dann gilt:

$$\begin{aligned}
\sum_{j=0}^{k_n-1} a^j d(b^{k_n-j}) &= \sum_{j=0}^{k_n-1} a^j b^{\gamma k_n} b^{-\gamma j} \\
&= b^{\gamma k_n} \cdot \sum_{j=0}^{k_n-1} \left(\frac{a}{b^\gamma}\right)^j \\
\text{falls } a \neq b^\gamma: &= b^{\gamma k_n} \cdot \frac{\left(\frac{a}{b^\gamma}\right)^{k_n} - 1}{\frac{a}{b^\gamma} - 1} \\
&= \frac{a^{k_n} - b^{\gamma k_n}}{\frac{a}{b^\gamma} - 1} \\
\text{falls } a = b^\gamma: &= b^{\gamma k_n} \cdot \sum_{j=0}^{k_n-1} 1 \\
&= k_n \cdot b^{\gamma k_n}
\end{aligned}$$

Beachte:

$$\begin{aligned}
a^{k_n} &= a^{\log_b n} \\
&= (b^{\log_b a})^{\log_b n} \\
&= (b^{\log_b n})^{\log_b a} \\
&= n^{\log_b a}
\end{aligned}$$

1. Fall $a < b^\gamma$**:** geometrische Reihe konvergiert für $k_n \to \infty$:

$$\begin{aligned}
T(n) &\leq a^{k_n} + b^{\gamma k_n} \cdot \frac{-1}{\frac{a}{b^\gamma} - 1} \\
&= n^{\log_b a} + n^\gamma \cdot \underbrace{\frac{1}{1 - \frac{a}{b^\gamma}}}_{>0 \text{ und unabh. von } n}
\end{aligned}$$

Wegen $\log_b a < \gamma$ folgt:

$$\boxed{T(n) \in O(n^\gamma)}$$

2. Fall $a = b^\gamma$**:** Wegen $\gamma = \log_b a$ ist

$$\begin{aligned}
T(n) &= a^{k_n} + b^{\gamma k_n} \cdot k_n \\
&= n^{\log_b a} + n^\gamma \cdot \log_b n \\
&= n^\gamma + n^\gamma \cdot \log_b n
\end{aligned}$$

Also:

$$\boxed{T(n) \in O(n^\gamma \log_b n)}$$

3. Fall $a > b^\gamma$**:**

$$T(n) \quad = \quad a^{k_n} + \frac{a^{k_n} - (b^\gamma)^{k_n}}{\frac{a}{b^\gamma} - 1}$$

Wegen $a^{k_n} > (b^\gamma)^{k_n}$ folgt:

$$\boxed{T(n) \in O(n^{\log_b a})}$$

Teil b: Erweiterung auf allgemeine Funktion $d(n)$:

$$T(n) = a^{k_n} \cdot T(1) + \sum_{j=0}^{k_n - 1} a^j d(b^{k_n - j})$$

$d(n) \in O(n^\gamma)$ heißt: Für genügend großes n gilt:

$$d(n) < c \cdot n^\gamma \text{ für geeignetes } c > 0$$

Damit kann man $T(n)$ nach oben abschätzen:

$$T(n) \leq a^{k_n} \cdot T(1) + c \cdot b^{\gamma k_n} \cdot \sum_{j=0}^{k_n - 1} \left(\frac{a}{b^\gamma}\right)^j$$

Auf diese Ungleichung kann man dieselben Überlegungen wie im Teil a anwenden und erhält dieselben Ergebnisse in Abhängigkeit von γ.

Teil c: Erweiterung auf allgemeine n [Cormen et al., S. 70]

$$T(n) = aT(\lceil \frac{n}{b} \rceil) + d(n)$$

Sukzessives Einsetzen liefert die Argumente $n, \lceil \frac{n}{b} \rceil, \lceil \lceil \frac{n}{b} \rceil / b \rceil, \ldots$

Definiere:

$$n_i := \begin{cases} n & \text{für} \quad i = 0 \\ \lceil \frac{n_{i-1}}{b} \rceil & \quad\quad\; i > 0 \end{cases}$$

Ungleichung: $\lceil x \rceil \leq x + 1$

$$\begin{aligned} n_0 &\leq n \\ n_1 &\leq \frac{n}{b} + 1 \\ n_2 &\leq \frac{n}{b^2} + \frac{n}{b} + 1 \\ \text{allgemein: } n_i &\leq \frac{n}{b_i} + \sum_{j=0}^{i-1} \frac{1}{b^j} \\ &\leq \frac{n}{b_i} + \frac{b}{b-1} \end{aligned}$$

Für T(n) kann man dann zeigen:

$$T(n) \leq a^{\lfloor \log_b n \rfloor - 1} \cdot T(1) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j d(n_j)$$

und schätzt dann weiter nach oben ab (mühsame Rechnung).

1.5.3 Rekursions<u>un</u>gleichungen

Oft ist es einfacher, statt einer Gleichung eine Ungleichung zu betrachten.

Beispiel: Mergesort – Beweis durch vollständige Induktion.

Konstruktive Induktion

Eine spezielle Variante der vollständigen Induktion, die sog. *konstruktive Induktion*, wird benutzt um zunächst noch unbekannte Konstanten zu bestimmen.

Beispiel: Fibonacci-Zahlen

$$f(n) = \begin{cases} n & \text{für} \quad n \leq 1 \\ f(n-1) + f(n-2) & \quad\quad n > 1 \end{cases}$$

Behauptung: Für $n \geq n_0$ gilt:

$$f(n) \geq a \cdot c^n$$

mit Konstanten $a > 0$ und $c > 0$, die noch zu bestimmen sind.

Induktionsschritt:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &\geq ac^{n-1} + ac^{n-2} \end{aligned}$$

Konstruktiver Teil: Wir bestimmen ein $c > 0$ mit der Forderung

$$ac^{n-1} + ac^{n-2} \overset{!}{\geq} a \cdot c^n$$

Umformen ergibt:

$$c + 1 \geq c^2$$

oder

$$c^2 - c - 1 \leq 0$$

Lösung:

$$c \leq \frac{1+\sqrt{5}}{2}$$

(Negative Lösung scheidet aus.)

Induktionsanfang: Wähle $a > 0$ und n_0 geeignet.

Beispiel zur Warnung

Der exakte Wert der Konstanten in der Ungleichung ist wichtig. Wir betrachten dazu folgendes Beispiel:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$

Falsch ist der folgende „Beweis“:

$$\begin{array}{rrcl} \text{Behauptung:} & T(n) & \leq & c \cdot n \in O(n) \qquad \boxed{\textbf{Falsch!}} \\ \text{„Beweis“:} & T(n) & \leq & 2 \cdot (c \cdot \lfloor \frac{n}{2} \rfloor) + n \\ & & \leq & cn + n \\ & & \in & O(n) \end{array}$$

Der Fehler:

- Behauptet wurde: $T(n) \leq c \cdot n$
- Gezeigt wurde: $T(n) \leq (c+1)n$

2 Sortieren

2.1 Einführung

Das Sortieren von Datensätzen ist ein wichtiger Bestandteil von vielen Anwendungen. Laut IBM werden in kommerziellen Rechenanlagen 25% der Rechenzeit für das Sortieren aufgewendet [Mehlhorn 88]. Die Effizienzansprüche an die Algorithmen sind dementsprechend hoch. Als Einstieg in die Problematik werden in diesem Kapitel zunächst einige grundlegende, sog. *elementare Sortierverfahren* vorgestellt und untersucht.

Elementare Sortierverfahren:

- SelectionSort (Sortieren durch Auswahl)
- InsertionSort (Sortieren durch Einfügen)
- BubbleSort
- BucketSort

Elementare Sortierverfahren sind dadurch charakterisiert, daß sie im Mittel eine (in der Größe der Eingabe) quadratische Laufzeit besitzen[1]. Im Unterschied hierzu weisen die *höheren Sortierverfahren* eine im Mittel überlineare Laufzeit $O(N \log N)$ auf:

Höhere Sortierverfahren:

- MergeSort
- QuickSort
- HeapSort

Konvention
Für die nachfolgenden Betrachtungen sei stets eine Folge $a[1], \ldots, a[N]$ von Datensätzen (Records) gegeben. Jeder Datensatz $a[i]$ besitzt eine Schlüsselkomponente $a[i].key$ ($i =$

[1] BucketSort nimmt hier einen Sonderstatus ein. Zwar vermag das Verfahren eine Schlüsselfolge in nur linearer Zeit zu sortieren, jedoch muß die Schlüsselmenge zusätzliche Eigenschaften besitzen, die bei einem allgemeinen Sortierverfahren nicht gefordert sind.

$1, \ldots, N$). Darüber hinaus können die Datensätze weitere Informationseinheiten (z.B. Name, Adresse, PLZ, etc.) enthalten. Die Sortierung erfolgt auschließlich nach der Schlüsselkomponente *key*. Hierzu muß auf der Menge aller Schlüssel eine *totale Ordnung* definiert sein.

Definition: Partielle Ordnung
Es sei $\mathcal{M}$ eine nicht leere Menge und $\leq \subseteq \mathcal{M} \times \mathcal{M}$ eine binäre Relation auf $\mathcal{M}$. Das Paar $< \mathcal{M}, \leq >$ heißt eine *partielle Ordnung auf* $\mathcal{M}$, genau dann wenn $\leq$ die folgenden Eigenschaften erfüllt:

- **Reflexivität:** $\forall x \in \mathcal{M}: \quad x \leq x$
- **Transitivität:** $\forall x, y, z \in \mathcal{M}: \quad x \leq y \wedge y \leq z \implies x \leq z$
- **Antisymmetrie:** $\forall x, y \in \mathcal{M}: \quad x \leq y \wedge y \leq x \implies x = y$

Definition: Strikter Anteil einer Ordnungsrelation
Für eine partielle Ordnung $\leq$ auf einer Menge $\mathcal{M}$ definieren wir die Relation $<$ durch:

$$x < y \;:=\; x \leq y \wedge x \neq y$$

Die Relation $<$ heißt auch der *strikte Anteil von* $\leq$.

Definition: Totale Ordnung
Es sei $\mathcal{M}$ eine nicht leere Menge und $\leq \subseteq \mathcal{M} \times \mathcal{M}$ eine binäre Relation über $\mathcal{M}$. $\leq$ heißt eine *totale Ordnung auf* $\mathcal{M}$, genau dann wenn gilt:

- $< \mathcal{M}, \leq >$ ist eine partielle Ordnung und
- **Trichotomie:** $\forall x, y \in \mathcal{M}: \quad x < y \;\dot{\vee}\; x = y \;\dot{\vee}\; y < x$

Eine totale Ordnung $< \mathcal{M}, \leq >$ ist also eine partielle Ordnung, bei der sämtliche Elemente aus der Grundmenge $\mathcal{M}$ miteinander vergleichbar, d.h. anordnenbar sind.

Daß eine partielle Ordnung nicht notwendigerweise auch eine totale Ordnung sein muß, läßt sich an folgendem Beispiel zeigen. So ist die Potenzmenge $\mathcal{M} := \mathcal{P}(\{1, 2, 3\})$ zusammen mit der Mengeninklusion $\subseteq$ eine partielle Ordnung. $< \mathcal{M}, \subseteq >$ ist jedoch keine totale Ordnung, denn die Elemente $\{1\}, \{2\} \in \mathcal{M}$ sind bzgl. $\subseteq$ nicht anordnenbar.

Definition: Sortierproblem
Gegeben sei eine Folge $a[1], \ldots, a[N]$ von Datensätzen (Records) mit einer Schlüsselkomponente $a[i].key$ $(i = 1, \ldots, N)$ und eine totale Ordnung $\leq$ auf der Menge aller Schlüssel. Das *Sortierproblem* besteht darin, eine Permutation π der ursprünglichen Folge zu bestimmen, so daß gilt:

$$a[\pi_1].key \;\leq\; a[\pi_2].key \;\leq\; \ldots \;\leq\; a[\pi_{N-1}].key \;\leq\; a[\pi_N].key$$

Beispiele:

Liste	Schlüsselelement	Ordnung
Telefonbuch	Nachname	lexikographische Ordnung
Klausurergebnisse	Punktezahl	$\leq$ auf $\mathbb{Q}$
Lexikon	Stichwort	lexikographische Ordnung
Studentenverzeichnis	Matrikelnummer	$\leq$ auf $\mathbb{N}$
Entfernungstabelle	Distanz	$\leq$ auf $\mathbb{R}$
Fahrplan	Abfahrtszeit	„früher als“

Beachte: Da wir es mit einer Folge (und keiner Menge) von Datensätzen zu tun haben, können Schlüssel oder ganze Datensätze mehrfach auftreten.

Unterscheidungskriterien
Sortieralgorithmen können nach verschiedenen Kriterien klassifiziert werden:

- Sortiermethode
- Effizienz: $O(N^2)$ für elementare Sortierverfahren, $O(N \log N)$ für höhere Sortierverfahren
- intern (alle Records im Arbeitsspeicher) oder extern (Platten)
- direkt/indirekt (d.h. mit Pointern oder Array-Indizes)
- im Array oder nicht
- in situ (d.h. in einem einzigen Array ohne zusätzliches Hilfsfeld) oder nicht
- allgemein/speziell
  (z.B. fordert BucketSort zusätzliche Eigenschaften der Schlüsselmenge)
- stabil (Reihenfolge von Records mit gleichem Schlüssel bleibt erhalten) oder nicht

Viele Aufgaben sind mit dem Sortieren verwandt und können auf das Sortierproblem zurückgeführt werden:

- Bestimmung des Median (der Median ist definiert als das Element an der mittleren Position der sortierten Folge)
- Bestimmung der k kleinsten/größten Elemente

Weitere (hier jedoch nicht behandelte) Sortierverfahren sind:

- BinaryInsertionSort
- ShellSort
- ShakerSort (Variante von BubbleSort)
- MergeSort (vgl. Abschnitt 1.4.1)

Deklarationsteil
Die folgende Typdeklaration ist exemplarisch für die zu sortierenden Datensätze:

```
TYPE KeyType  = REAL;
TYPE ItemType = RECORD
                  ...
                  key : KeyType;
                END;

VAR  a : ARRAY[1..N] OF ItemType;
```

Für `KeyType` können auch andere Datentypen auf denen eine Ordnung definiert ist verwendet werden (z.B. `INTEGER`). Da stets nur nach der Schlüsselkomponente `key` sortiert wird, werden wir zur Vereinfachung der Algorithmen nur noch Felder vom Typ `KeyType` verwenden:

```
VAR  a : ARRAY[1..N] OF KeyType;
```

2.2 Elementare Sortierverfahren

2.2.1 SelectionSort

Sortieren durch Auswahl

Gegeben sei eine Folge $a[1], \ldots, a[N]$ von Schlüsselelementen.

Prinzip: Wir betrachten den i-ten Durchgang der Schleife $i = 1, ..., N$:

- Bestimme den Datensatz mit dem kleinsten Schlüssel aus $a[i], ..., a[N]$
- vertausche dieses Minimum mit $a[i]$

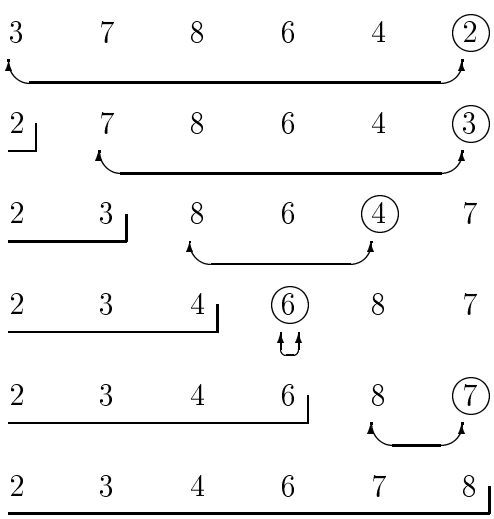
Vorteil von SelectionSort: Jeder Datensatz wird höchstens einmal bewegt.

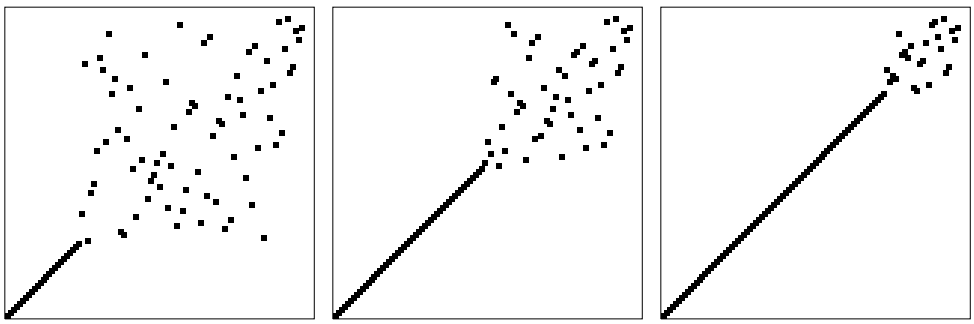
Achtung: Es existieren Varianten von SelectionSort mit anderer Anzahl von Vertauschungen.

Programm:

```
PROCEDURE SelectionSort (VAR a : ARRAY[1..N] OF KeyType) =
  VAR min : CARDINAL;
      t   : KeyType;
  BEGIN
    FOR i := 1 TO N-1 DO
      min := i;
      FOR j := i+1 TO N DO
        IF a[j] < a[min] THEN min := j; END;
      END;
      t := a[min]; a[min] := a[i]; a[i] := t;
    END;
  END SelectionSort;
```

Beispiel:
Markiert ist jeweils das kleinste Element `a[min]` der noch unsortierten Teilfolge.





(a) (b) (c)

Abbildung 2.1: SelectionSort einer zufälligen Permutation von N Schlüsselelementen, nachdem (a) $N/4$, (b) $N/2$ und (c) $3N/4$ der Schlüssel sortiert worden sind.

Komplexitätsanalyse
Zum Sortieren der gesamten Folge $a[1], \ldots, a[N]$ werden $N-1$ Durchläufe benötigt. Pro Schleifendurchgang i gibt es eine Vertauschung, die sich aus je drei Bewegungen und $N-i$ Vergleichen zusammensetzt, wobei $N-i$ die Anzahl der noch nicht sortierten Elemente ist. Insgesamt ergeben sich:

- $3 \cdot (N-1)$ Bewegungen und
- $(N-1) + (N-2) + \ldots + 2 + 1 \;=\; \dfrac{N \cdot (N-1)}{2}$ Vergleiche.

Da die Anzahl der Bewegungen nur linear in der Anzahl der Datensätze wächst, ist SelectionSort besonders für Sortieraufgaben geeignet, in denen die einzelnen Datensätze sehr groß sind.

2.2.2 InsertionSort

Sortieren durch Einfügen.

Gegeben sei eine Folge $a[1], \ldots, a[N]$ von Schlüsselelementen.

Prinzip: Wir betrachten den i-ten Durchgang der Schleife $i = 2, ..., N$. Dabei sei die $i-1$-elementige Teilfolge $a[1], \ldots, a[i-1]$ bereits sortiert:

- Füge den Datensatz $a[i]$ an der korrekten Position der bereits sortierten Teilfolge $a[1], \ldots, a[i-1]$ ein.

Programm:

```
PROCEDURE InsertionSort (VAR a : ARRAY[0..N] OF KeyType) =
  VAR j : CARDINAL;
      v : KeyType;
  BEGIN
    a[0] := −∞; (* in Modula-3: a[0]:=FIRST(KeyType); *)
    FOR i := 2 TO N DO
      v := a[i];
      j := i;
      WHILE a[j-1]>v DO
        a[j] := a[j-1];
        DEC(j);
      END;
      a[j] := v;
    END;
  END InsertionSort;
```

Beachte: Um in der `WHILE`-Schleife eine zusätzliche Abfrage auf die linke Feldgrenze zu vermeiden, wird ein sog. *Sentinel-Element* (=Wärter; Anfangs- oder Endmarkierung) $a[0] := -\infty$ verwendet.

Bemerkung: Bei *nicht-strikter* Übersetzung des Compilers kann die `WHILE`-Schleife auch ohne Verwendung eines Sentinel-Elementes formuliert werden:

```
WHILE 1<j AND a[j-1]>v DO
```

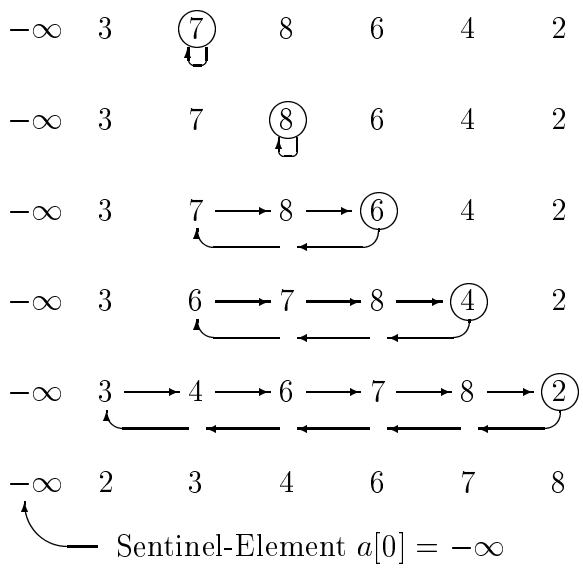
Dabei wird zunächst nur der linke Teil `1<j` der Abbruchbedingung ausgewertet. Liefert die Auswertung den Wert `FALSE`, so wird die Schleife verlassen, ohne daß der rechte Teilausdruck `a[j-1]>v` ausgewertet wird. (Sprechweise: „Der Operator `AND` ist *nicht-strikt*“ bzw. „`AND` ist ein *short-circuit operator*“) Die Auswertung des rechten Teilausdrucks erfolgt nur dann, wenn das linke Argument von `AND` zu `TRUE` ausgewertet werden kann.

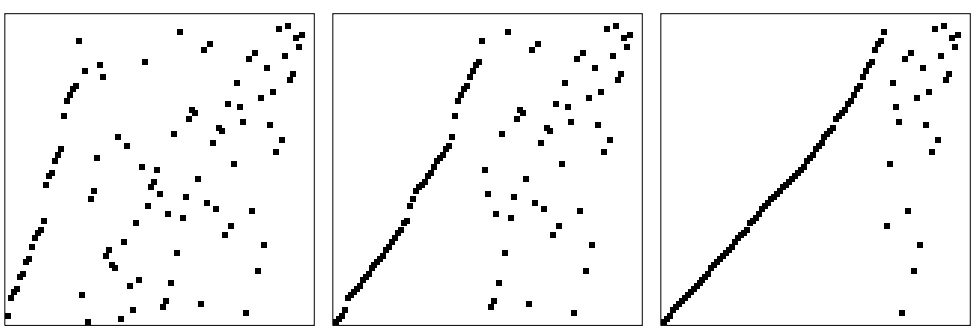
Bei *strikter* Übersetzung des Compilers kann die Abbruchbedingung der `WHILE`-Schleife auch innerhalb einer `LOOP`-Anweisung formuliert werden:

```
LOOP
  IF 1<j THEN
    IF a[j-1]>v THEN
      a[j] := a[j-1];
      DEC(j);
    ELSE EXIT; END;
  ELSE EXIT; END;
END;
```

Die strikte Auswertung eines Compilers heißt auch *leftmost-innermost* oder *call-by-value*. Andere Bezeichnungen für die nicht-strikte Auswertung sind *lazy evaluation* (verzögerte Auswertung), *leftmost-outermost* oder *call-by-need*.

Beispiel:
InsertionSort durchläuft die zu sortierende Folge von links nach rechts. Dabei ist die Anfangsfolge des Feldes zwar in sich sortiert, die Elemente befinden sich jedoch noch nicht notwendigerweise an ihrer endgültigen Position.





(a) (b) (c)

Abbildung 2.2: InsertionSort einer zufälligen Permutation von N Schlüsselelementen, nachdem (a) $N/4$, (b) $N/2$ und (c) $3N/4$ der Schlüssel sortiert worden sind.

Komplexitätsanalyse
Vergleiche: Das Einfügen des Elements $a[i]$ in die bereits sortierte Anfangsfolge $a[1], \ldots, a[i-1]$ erfordert mindestens einen Vergleich, höchstens jedoch i Vergleiche. Im Mittel sind dies $i/2$ Vergleiche, da bei zufälliger Verteilung der Schlüssel die Hälfte der bereits eingefügten Elemente größer ist als das Element $a[i]$.

- *Best Case* – Bei vollständig vorsortierten Folgen ergeben sich $N-1$ Vergleiche.
- *Worst Case* – Bei umgekehrt sortierten Folgen gilt für die Anzahl der Vergleiche:

$$\begin{aligned}\sum_{i=2}^{N} i &= \left(\sum_{i=1}^{N} i\right) - 1 = \frac{N(N+1)}{2} - 1 \\ &= \frac{N^2}{2} + \frac{N}{2} - 1\end{aligned}$$

- *Average Case* – Die Anzahl der Vergleiche ist etwa $N^2/4$

Bewegungen: Im Schleifendurchgang i ($i = 2,\ldots,N$) wird bei bereits sortierter Anfangsfolge $a[1],\ldots,a[i-1]$ das einzufügende Element $a[i]$ zunächst in die Hilfsvariable v kopiert (eine Bewegung) und anschließend mit höchstens i Schlüsseln (beachte das Sentinel-Element), wenigstens jedoch mit einem Schlüssel und im Mittel mit $i/2$ Schlüsseln verglichen. Bis auf das letzte Vergleichselement werden die Datensätze um je eine Position nach rechts verschoben (jeweils eine Bewegung). Anschließend wird der in v zwischengespeicherte Datensatz an die gefundene Position eingefügt (eine Bewegung). Für den Schleifendurchgang i sind somit mindestens zwei Bewegungen, höchstens jedoch $i+1$ und im Mittel $i/2+2$ Bewegungen erforderlich. Damit ergibt sich insgesamt der folgende Aufwand:

- *Best Case* – Bei vollständig vorsortierten Folgen $2(N-1)$ Bewegungen
- *Worst Case* – Bei umgekehrt sortierten Folgen $N^2/2$ Bewegungen
- *Average Case* – $\sim N^2/4$ Bewegungen

Für „fast sortierte“ Folgen verhält sich InsertionSort nahezu linear. Im Unterschied zu SelectionSort vermag InsertionSort somit eine in der zu sortierenden Datei bereits vorhandene Ordnung besser auszunutzen.

2.2.3 BubbleSort

Sortieren durch wiederholtes Vertauschen unmittelbar benachbarter Array-Elemente.

Gegeben sei eine Folge $a[1], \ldots, a[N]$ von Schlüsselelementen.

Prinzip: Wir betrachten den i-ten Durchgang der Schleife $i = N, N-1, \ldots, 2$:

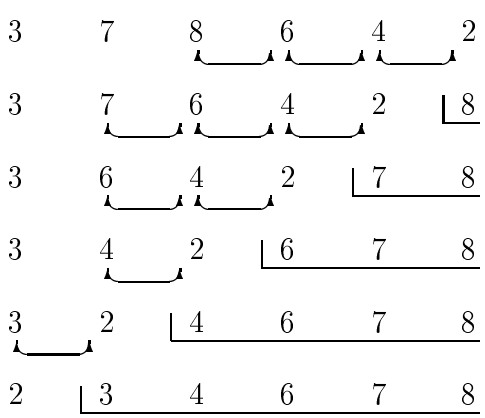
- Schleife $j = 2, 3, \ldots, i$: ordne $a[j-1]$ und $a[j]$

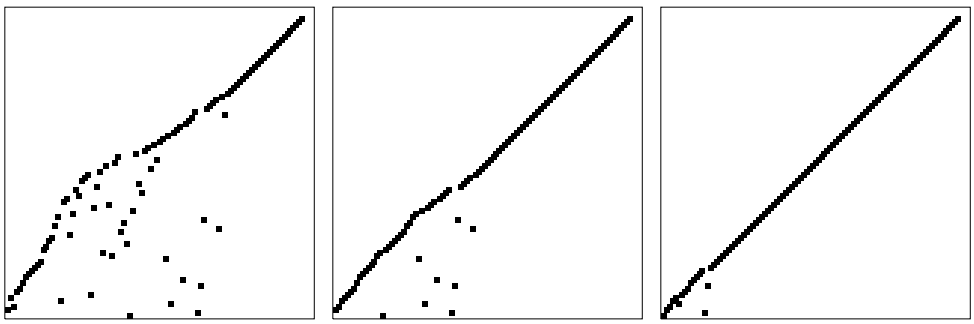
Programm:

```
PROCEDURE BubbleSort (VAR a : ARRAY[1..N] OF KeyType) =
  VAR t : KeyType;
BEGIN
  FOR i:= N TO 1 BY -1 DO
    FOR j:= 2 TO i DO
      IF a[j-1]>a[j] THEN
        t := a[j-1];
        a[j-1] := a[j];
        a[j] := t;
      END;
    END;
  END;
END BubbleSort;
```

Durch wiederholtes Vertauschen von unmittelbar benachbarten Schlüsselelementen wandern die größeren Schlüssel nach und nach an das rechte Ende des zu sortierenden Feldes. Nach jedem Durchgang der äußersten Schleife nimmt das größte Element der noch unsortierten Teilfolge seine endgültige Position im Array ein. Dabei wird zwar im Verlauf der Sortierung auch der Ordnungsgrad aller noch unsortierten Schlüsselelemente der Folge erhöht, jedoch ist BubbleSort nicht in der Lage, hieraus für die weitere Sortierung einen Nutzen zu ziehen.

Beispiel:





(a) (b) (c)

Abbildung 2.3: BubbleSort einer zufälligen Permutation von N Schlüsselelementen, nachdem (a) $N/4$, (b) $N/2$ und (c) $3N/4$ der Schlüssel sortiert worden sind.

Komplexitätsanalyse

Vergleiche: Die Anzahl der Vergleiche ist unabhängig vom Vorsortierungsgrad der Folge. Daher sind der *worst case*, *average case* und *best case* identisch, denn es werden stets alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen. Im i-ten Schleifendurchgang ($i = N, N-1, \ldots, 2$) enthält die noch unsortierte Anfangsfolge $N - i + 1$ Elemente, für die $N - i$ Vergleiche benötigt werden.

Um die ganze Folge zu sortieren, sind $N - 1$ Schritte erforderlich. Die Gesamtzahl der Vergleiche wächst damit quadratisch in der Anzahl der Schlüsselelemente:

$$\begin{aligned}\sum_{i=1}^{N-1}(N-i) &= \sum_{i=1}^{N-1} i \\ &= \frac{N(N-1)}{2}\end{aligned}$$

Bewegungen: Aus der Analyse der Bewegungen für den gesamten Durchlauf ergeben sich:

- im *Best Case*: 0 Bewegungen
- im *Worst Case*: $\sim \dfrac{3N^2}{2}$ Bewegungen
- im *Average Case*: $\sim \dfrac{3N^2}{4}$ Bewegungen.

Vergleich elementarer Sortierverfahren

Anzahl der Vergleiche elementarer Sortierverfahren:

Verfahren	**Best Case**	**Average Case**	**Worst Case**
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	N	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

Anzahl der Bewegungen elementarer Sortierverfahren:

Verfahren	**Best Case**	**Average Case**	**Worst Case**
SelectionSort	$3(N-1)$	$3(N-1)$	$3(N-1)$
InsertionSort	$2(N-1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

Folgerungen:

BubbleSort: ineffizient, da immer $N^2/2$ Vergleiche

InsertionSort: gut für fast sortierte Folgen

SelectionSort: gut für große Datensätze aufgrund konstanter Zahl der Bewegungen, jedoch stets $N^2/2$ Vergleiche

Fazit: InsertionSort und SelectionSort sollten nur für $N \leq 50$ eingesetzt werden. Für größere N sind höhere Sortierverfahren wie z.B. QuickSort und HeapSort besser geeignet.

2.2.4 Indirektes Sortieren

Bei sehr großen Datensätzen kann das Vertauschen der Records den Rechenaufwand für das Sortieren dominieren. In solchen Fällen ist es sinnvoll, statt der Datensätze selbst nur Verweise auf diese zu sortieren.

Verfahren:

1. Verwende ein Index-Array $p[1..N]$, das mit $p[i] := i \quad (i = 1, \ldots, N)$ initialisiert wird
2. für Vergleiche erfolgt der Zugriff auf einen Record mit $a[p[i]]$
3. Vertauschen der Indizes $p[i]$ statt der Array-Elemente $a[p[i]]$
4. optional werden nach dem Sortieren die Records selbst umsortiert (Aufwand: $O(N)$)

Programm:

```
PROCEDURE InsertionSort_Indirect (VAR a : ARRAY[0..N] OF KeyType;
                                  VAR p : ARRAY[0..N] OF [0..N]) =
  VAR j, v : CARDINAL;
  BEGIN
    a[0] := −∞;
    FOR i := 0 TO N DO
      p[i] := i;
    END;

    FOR i := 2 TO N DO
      v := p[i];
      j := i;
      WHILE a[p[j-1]]>a[v] DO
        p[j] := p[j-1];
        DEC(j);
      END;
      p[j] := v;
    END;
  END InsertionSort_Indirect;
```

Mit Hilfe der indirekten Sortierung kann jedes Sortierverfahren so modifiziert werden, daß nicht mehr als N Record-Vertauschungen nötig sind.

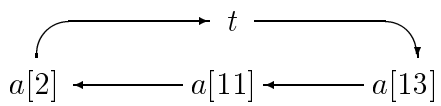
Umsortierung der Datensätze:
Sollen die Datensätze im Anschluß an die indirekte Sortierung selbst umsortiert werden, so gibt es hierzu zwei mögliche Varianten:

(a) Permutation mit zusätzlichem Array b: $b[i] := a[p[i]] \quad (i = 1, \ldots, N)$

(b) Permutation ohne zusätzliches Array: *in situ*, *in place* (lohnt nur bei großen Records):

Ziel: $p[i] = i \qquad (i = 1, \ldots, N)$

- falls $p[i] = i$: nichts zu tun;
- sonst: zyklische Vertauschung durchführen:
  1. kopiere Record: $t := a[i]$
     Ergebnis: Platz (Loch) an Position i;
  2. „iterieren“
     Beispiel: $t = a[2]$; $a[2] = a[11]$; $a[11] = a[13]$; $a[13] = t$;
     Ersetzungsreihenfolge des Ringtausches:



Beispiel: Indirektes Sortieren

Vor dem Sortieren:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$a[i]$	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
$p[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Nach dem indirekten Sortieren:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$a[i]$	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
$p[i]$	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

$a[p[i]] \leq a[p[i+1]]$

Durch Permutation mittels p[i] ergibt sich das sortierte Array:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$b[i]$	A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
$p[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

mit $b[i] := a[p[i]]$

Das folgende Programm führt eine in-situ-Permutation durch wiederholte verkettete Ringtauschoperationen aus:

Programm:

```
PROCEDURE InsituPermutation (VAR a : ARRAY[1..N] OF KeyType;
                             VAR p : ARRAY[1..N] OF CARDINAL) =
  VAR t : KeyType;
      j, k : CARDINAL;
  BEGIN
    FOR i := 1 TO N DO
      IF p[i] <> i THEN
        t := a[i];
        k := i;
        REPEAT
          j := k; a[j] := a[p[j]];
          k := p[j]; p[j] := j;
        UNTIL k=i;
        a[j] := t;
      END;
    END;
  END InsituPermutation;
```

2.2.5 BucketSort

Andere Namen: Bin Sorting, Distribution Counting, Sortieren durch Fachverteilen, Sortieren mittels Histogramm

Voraussetzung: Schlüssel können als ganzzahlige Werte im Bereich $0, \ldots, M-1$ dargestellt werden, so daß sie als Array-Index verwendet werden können.

$$a[i] \in \{0, ..., M-1\} \qquad \forall i = 1, \ldots, N$$

Prinzip:

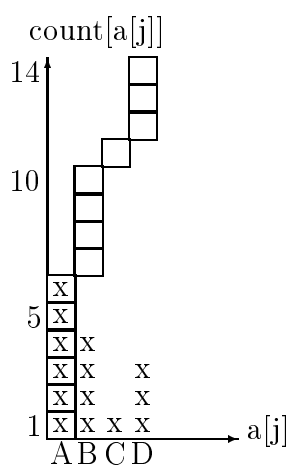
1. Erstelle ein Histogramm, d.h. zähle für jeden Schlüsselwert, wie häufig er vorkommt.
2. Berechne aus dem Histogramm die Position für jeden Record.
3. Bewege die Records (mit rückläufigem Index) an ihre errechnete Position.

Programm:

```
TYPE KeyType = [0..M-1];

PROCEDURE BucketSort (VAR a : ARRAY[1..N] OF KeyType) =
  VAR count : ARRAY KeyType OF CARDINAL;
      b     : ARRAY [1..N] OF KeyType;
  BEGIN
    FOR j := 0 TO M-1 DO     (* Initialisierung *)
      count[j] := 0;
    END;
    FOR i := 1 TO N DO       (* Erstelle Histogramm *)
      count[a[i]] := count[a[i]] + 1;
    END;
    FOR j := 1 TO M-1 DO     (* Berechne Position für jeden Schlüsselwert *)
      count[j] := count[j-1] + count[j];
    END;
    FOR i := N TO 1 BY -1 DO (* Bewege Record an errechnete Position *)
      b[count[a[i]]] := a[i];
      count[a[i]] := count[a[i]] - 1;
    END;
    FOR i := 1 TO N DO
      a[i] := b[i];
    END;
  END BucketSort;
```

Beispiel:

ABBAC|ADABB|ADDA



Eigenschaften:

- Wegen des rückläufigen Index ist BucketSort stabil.
- Für die Zeit- und Platzkomplexität gilt:

$$\begin{aligned} T(N) &= O(N+M) \\ &= O\big(\max\{N,M\}\big) \end{aligned}$$

- BucketSort arbeitet in dieser Form **nicht** in situ.
- Eine in-situ Variante ist möglich, jedoch verliert man dabei die Stabilität.

2.3 QuickSort

QuickSort wurde 1962 von C.A.R. Hoare entwickelt.

Prinzip: Das Prinzip folgt dem Divide-and-Conquer-Ansatz:

Gegeben sei eine Folge F von Schlüsselelementen.

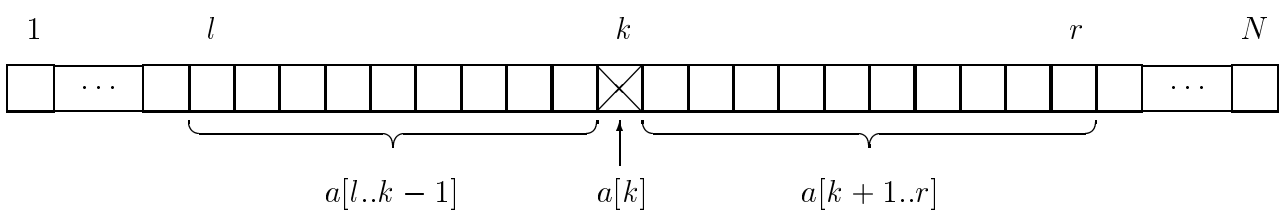
1. Zerlege F bzgl. eines partitionierenden Elementes (engl.: *pivot* = Drehpunkt) $p \in F$ in zwei Teilfolgen F_1 und F_2, so daß gilt:

$$\begin{aligned} x_1 \leq p \qquad & \forall\, x_1 \in F_1 \\ p \leq x_2 \qquad & \forall\, x_2 \in F_2 \end{aligned}$$

2. Wende dasselbe Schema auf jede der so erzeugten Teilfolgen F_1 und F_2 an, bis diese nur noch höchstens ein Element enthalten.

QuickSort realisiert diese Idee folgendermaßen:

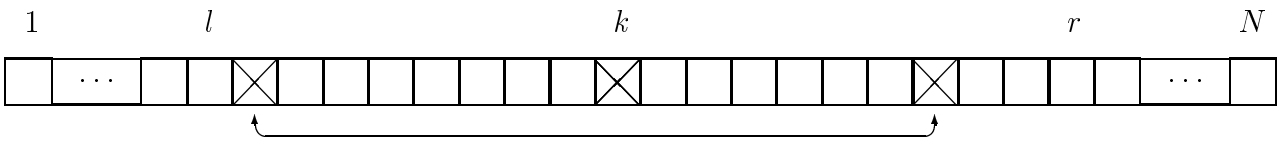
- **Ziel:** Zerlegung (Partitionierung) des Arrays $a[l..r]$ bzgl. eines Pivot-Elementes $a[k]$ in zwei Teilarrays $a[l..k-1]$ und $a[k+1..r]$



so daß gilt:

$$\begin{aligned} \forall\, i \in \{l, \ldots, k-1\} : \quad & a[i] \leq a[k] \\ \forall\, j \in \{k+1, \ldots, r\} : \quad & a[k] \leq a[j] \end{aligned}$$

- **Methode:** Austausch von Schlüsseln zwischen beiden Teilarrays



- **Rekursion:** linkes Teilarray $a[l..k-1]$ und rechtes Teilarray $a[k+1..r]$ bearbeiten

Komplexitätsabschätzung

Warum gewinnt man mit diesem Verfahren Rechenzeit? Hierzu betrachten wir die Abschätzung für den idealen Fall, in dem das partitionierende Element immer so gewählt wird, daß es die Folge F nach Umsortierung genau halbiert.

1. Schritt	N							
2. Schritt	$\frac{N}{2}$				$\frac{N}{2}$			
3. Schritt	$\frac{N}{4}$		$\frac{N}{4}$		$\frac{N}{4}$		$\frac{N}{4}$	
4. Schritt	$\frac{N}{8}$	$\frac{N}{8}$	$\frac{N}{8}$	$\frac{N}{8}$	$\frac{N}{8}$	$\frac{N}{8}$	$\frac{N}{8}$	$\frac{N}{8}$
$\vdots$					$\vdots$			
(ld N)-ter Schritt	1	1	1	1	1	1	$\cdots$	1

Um die Folge F in 1-elementige Teilfolgen zu zerlegen, werden ld N Schritte benötigt. Für das Vertauschen der Schlüsselelemente sind in jedem Schritt N Vergleiche notwendig, so daß sich insgesamt

$$T(N) = N \cdot \text{ld } N$$

Vergleiche ergeben.

Das folgende Programm bildet das Grundgerüst des Quicksort-Algorithmus. Die Parameter l und r definieren die linke bzw. rechte Feldgrenze der zu sortierenden Teilfolge.

Programm:

```
PROCEDURE QuickSort(l, r : CARDINAL) =
  VAR k : CARDINAL;
  BEGIN
    IF l<r THEN
      k := Partition(l, r);
      QuickSort(l, k-1);
      QuickSort(k+1, r);
    END;
  END QuickSort;
```

Die Prozedur `Partition(l, r)` muß die folgende Eigenschaft besitzen:

> Sei $a[k]$ das Pivot-Element, dann werden die Elemente im Array $a[l..r]$ so umsortiert, daß die folgende Bedingung erfüllt ist:
>
> $$\begin{array}{rll} & \forall\, i \in \{l, \ldots, k-1\} : & a[i] \leq a[k] \\ \text{und} & \forall\, j \in \{k+1, \ldots, r\} : & a[k] \leq a[j] \end{array}$$

Als Konsequenz ergibt sich hieraus, daß für $k :=$ `Partition(l, r)` das Pivot-Element $a[k]$ bereits seine endgültige Position im Array eingenommen hat.

Beachte, daß sich das Pivot-Element im allgemeinen erst **nach** der Umsortierung durch den Partitionierungsschritt an der Position k befindet.

Algorithmus:
Der folgende Algorithmus beschreibt informell die Funktionsweise der `Partition`-Prozedur.

```
PROCEDURE Partition(l, r : CARDINAL) : CARDINAL =
  BEGIN
    i := l-1;
    j := r;
    wähle Pivot-Element:  v:=a[r];

    REPEAT
      durchsuche Array von links (i:=i+1), solange bis a[i] ≥ v;
      durchsuche Array von rechts (j:=j-1), solange bis a[j] ≤ v;
      vertausche a[i] und a[j];
    UNTIL j ≤ i (* Zeiger kreuzen *)

    rückvertausche a[i] und a[j];
    vertausche a[i] und a[r]; (* positioniere Pivot-Element *)

    RETURN i; (* ≙ endgültige Position des Pivot-Elements *)
END Partition;
```

Programm:

```
PROCEDURE Partition(l, r : CARDINAL) : CARDINAL =
  VAR i, j : CARDINAL;
      v, t : KeyType;
  BEGIN
    i := l-1;
    j := r;
    v := a[r]; (* wähle Pivot-Element *)

    REPEAT
      REPEAT INC(i) UNTIL a[i]>=v;
      REPEAT DEC(j) UNTIL a[j]<=v;
      t := a[i]; a[i] := a[j]; a[j] := t;  ❶
    UNTIL j<=i; (* Zeiger kreuzen *)

    (* Rückvertauschung und Positionierung des Pivot-Elements *)
    a[j] := a[i];  ❷
    a[i] := a[r];  ❸
    a[r] := t;     ❹

    RETURN i;
  END Partition;
```

Anmerkungen zum QuickSort-Algorithmus:

(a) **Allgemeine Warnung:**
In der Literatur sind zahlreiche Varianten des QuickSort-Algorithmus zu finden, die sich in der Wahl des Pivot-Elementes und der Schleifenkontrolle unterscheiden.

(b) In der hier vorgestellten Implementierung wird in jeder der beiden Schleifen

```
REPEAT INC(i) UNTIL a[i]>=v;
REPEAT DEC(j) UNTIL a[j]<=v;
```

ein (explizites oder implizites) Sentinel-Element verwendet, d.h. für die Korrektheit der Schleifenkontrollen muß stets gewährleistet sein, daß es Elemente $a[l-1]$ und $a[r]$ gibt, so daß für das Pivot-Element v gilt:

$$a[l-1] \;\leq\; v \;\leq\; a[r]$$

Die folgende Fallunterscheidung zeigt, daß diese Forderung stets erfüllt ist.

Obere Grenze: Wegen $v := a[r]$ gilt auch $v \leq a[r]$

Untere Grenze: falls $l = 1$: $a[0] := -\infty$

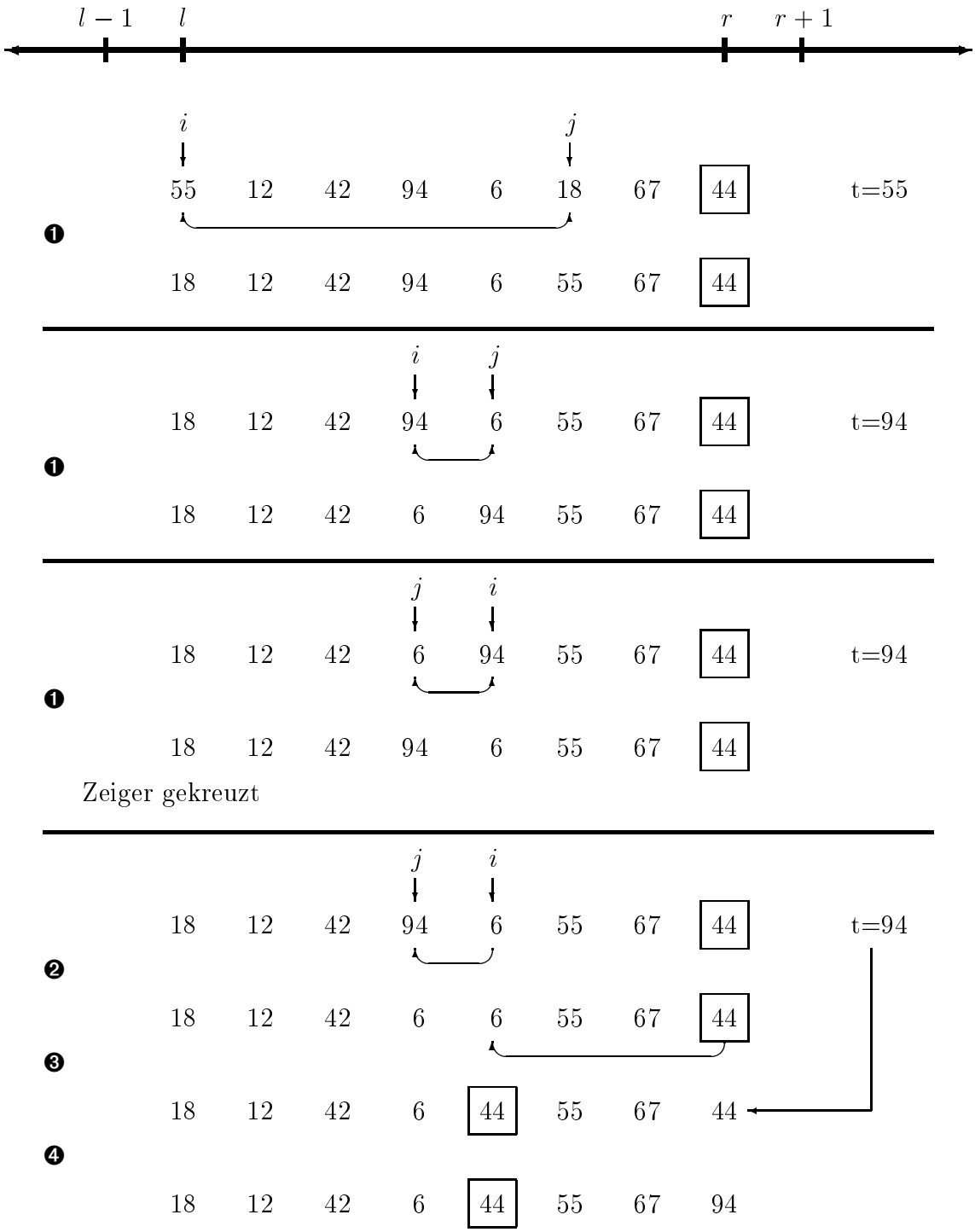
falls $l > 1$: Das Element $a[l-1]$ existiert aufgrund der Konstruktion des QuickSort-Algorithmus, da das Teilarray $a[1..l-1]$ vor dem Array $a[l..r]$ abgearbeitet wird. Dadurch befindet sich das Element $a[l-1]$ bereits an seiner endgültigen Position und es gilt $a[l-1] \leq a[i]$ für $i = l, \ldots, N$. Insbesondere ist dann auch $a[l-1] \leq v$.

(c) Beim Kreuzen der Zeiger i und j wird eine Vertauschung zuviel durchgeführt, die nach Abbruch der äußeren Schleife rückgängig zu machen ist:

- Mache die letzte Vertauschung wieder rückgängig:
  vertausche $a[i]$ und $a[j]$
- Positioniere das Pivot-Element:
  vertausche $a[i]$ und $a[r]$

Diese Anweisungen werden in den Zeilen ❷, ❸ und ❹ realisiert.

Beispiel: Die folgende Abbildung illustriert die Funktionsweise der Prozedur `Partition` für ein Array in den Grenzen $l, \ldots, r$. Der Schlüssel 44 ist das Pivot-Element:



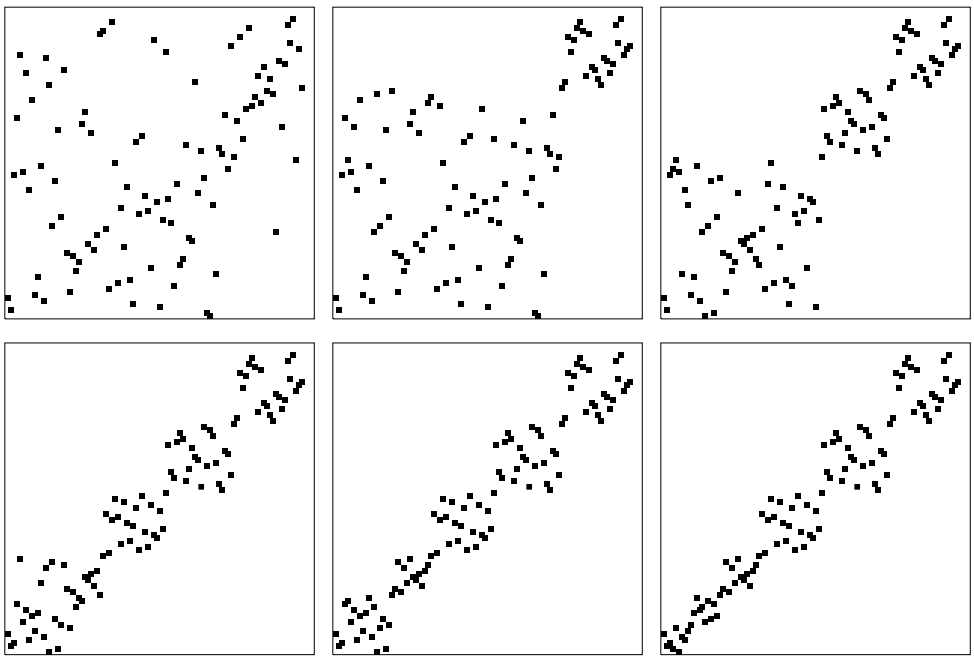


Abbildung 2.4: Anordnung der Schlüsselelemente bei der Sortierung durch QuickSort nach 0, 1, 2, 3, 4 bzw. 5 Partitionierungen. Jede Partition enthält mindestens 10 Elemente.

Zeitkomplexität:
Vergleiche: Wird mit $T(N)$ die Zahl der Vergleiche für Teilarrays der Größe N bezeichnet, dann gilt:

1. *Best Case* – Exakte Halbierung jedes Teilarrays:

$$T(N) = (N+1) + \underbrace{\min_{1 \le k \le N} \left\{ T(k-1) + T(N-k) \right\}}_{\text{günstigster Fall}}$$

$$= (N+1) + 2 \cdot T\Big(\frac{N+1}{2}\Big)$$

Lösung:

$$T(N) = (N+1) \cdot \mathrm{ld}\,(N+1)$$

2. *Worst Case* – Ungünstigste Aufteilung der Teilarrays:

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

$$T(N) = (N+1) + \underbrace{\max_{1 \le k \le N} \{ T(k-1) + T(N-k) \}}_{\text{ungünstigster Fall}}$$

Dann gilt:

$$T(N) \le \frac{(N+1)\cdot(N+2)}{2} - 3$$

Beweis durch Vollständige Induktion über N

Diese Schranke ist scharf, denn für ein aufsteigend sortiertes Array ohne Duplikate (d.h. ohne Mehrfachvorkommen gleicher Schlüssel) gilt:

$$\begin{aligned} T(N) &= (N+1) + N + (N-1) + \ldots + 3 \\ &= \frac{(N+1)\cdot(N+2)}{2} - 3 \end{aligned}$$

3. *Average Case* – Zur Bestimmung der mittleren Anzahl von Vergleichen muß über alle möglichen Pivot-Elemente $a[k]$ $(k = 1, \ldots, N)$ gemittelt werden:

$$\begin{aligned} N \ge 2: \qquad T(N) &= \frac{1}{N} \cdot \sum_{k=1}^{N} \Big[N + 1 + T(k-1) + T(N-k) \Big] \\ &= N + 1 + \underbrace{\frac{1}{N} \cdot \Big[\sum_{k=1}^{N} T(k-1) + \sum_{k=1}^{N} T(N-k) \Big]}_{= \frac{2}{N} \sum_{k=1}^{N} T(k-1)} \\ &= N + 1 + \frac{2}{N} \sum_{k=1}^{N} T(k-1) \end{aligned}$$

Elimination der Summe:

$$N \cdot T(N) = N \cdot (N+1) \; + \; 2 \cdot \sum_{k=1}^{N} T(k-1) \qquad ①$$

$$(N-1) \cdot T(N-1) = (N-1) \cdot N \; + \; 2 \cdot \sum_{k=1}^{N-1} T(k-1) \qquad ②$$

$$① - ② : \quad N \cdot T(N) \; - \; (N-1) \cdot T(N-1) = 2 \cdot N \; + \; 2 \cdot T(N-1)$$

$$N \cdot T(N) = 2 \cdot N \; + \; (N+1) \cdot T(N-1)$$

$$\frac{T(N)}{N+1} = \frac{2}{N+1} \; + \; \frac{T(N-1)}{N}$$

Sukzessives Substituieren:

$$\begin{aligned}
\frac{T(N)}{N+1} &= \frac{2}{N+1} \; + \; \frac{T(N-1)}{N} \\
&= \frac{2}{N+1} \; + \; \frac{2}{N} \; + \; \frac{T(N-2)}{N-1} \\
&\vdots \\
&= \sum_{k=2}^{N} \frac{2}{k+1} \; + \; \frac{T(1)}{2} \\
&= 2 \cdot \sum_{k=3}^{N+1} \frac{1}{k} \; + \; \frac{T(1)}{2}
\end{aligned}$$

Mit der Ungleichung

$$\ln \frac{N+1}{M} \; \leq \; \sum_{k=m}^{N} \frac{1}{k} \; \leq \; \ln \frac{N}{N-1} \qquad \text{(Beweis: siehe Abschnitt 2.3.1)}$$

ergeben sich folgende Schranken:

$$\frac{T(N)}{N+1} \; \leq \; 2 \cdot \ln \frac{N+1}{2} + \frac{T(1)}{2}$$

$$\text{und} \qquad 2 \cdot \ln \frac{N+2}{3} + \frac{T(1)}{2} \; \leq \frac{T(N)}{N+1}$$

Ergebnis:

$$\begin{aligned}
T(N) &= 2 \cdot (N+1) \cdot \ln(N+1) \; + \; \Theta(N) \\
&= 1.386 \cdot (N+1) \cdot \mathrm{ld}(N+1) \; + \; \Theta(N)
\end{aligned}$$

Zusammenfassung
Analyse von Quicksort für die hier vorgestellte Implementierung:

- *Best Case*: $T(N) = (N+1) \cdot \mathrm{ld}(N+1)$
- *Average Case*: $T(N) = 1.386 \cdot (N+1) \cdot \mathrm{ld}(N+1)$
- *Worst Case*: $T(N) = \dfrac{(N+1) \cdot (N+2)}{2} - 3$

Varianten und Verbesserungen
Die Effizienz von QuickSort beruht darauf, daß in der innersten und somit am häufigsten ausgeführten Schleife nur ein Schlüsselvergleich durchgeführt wird. Im worst-case (d.h. für aufsteigend sortierte Folgen) wächst die Zahl der Vergleiche jedoch quadratisch in der Größe der Eingabefolge. Zudem ist QuickSort aufgrund des Rekursionsoverheads für kleine Folgen nicht geeignet. In der Literatur finden sich daher einige Verbesserungsvorschläge für die hier vorgestellte Implementierung:

- andere Wahl des Pivot-Elements:
  - $v := \big(a[l] + a[r]\big)/2$
  - median-of-three: wähle das mittlere Element dreier zufällig ausgewählter Elemente

    Vorteile: + keine expliziten Sentinel-Elemente erforderlich
    + worst-case wird weniger wahrscheinlich
    + insgesamt: 5% kürzere Laufzeit
- Beobachtung: QuickSort ist ineffizient bei kleinen Arrays, z.B. $M = 12$ oder 22.

  Abhilfe:
  ```
  IF M<(r-l) THEN QuickSort(l, r);
                 ELSE InsertionSort(l, r);
                 END;
  ```
- Der Speicherplatzbedarf wird aufgrund der Rekursion indirekt über die Größe der Aktivierungsblöcke bestimmt. Wenn jedoch stets das kleinere der beiden Teilarrays zuerst bearbeitet wird, dann ist die Größe der Aktivierungsblöcke nach oben durch $2 \cdot \mathrm{ld}\ N$ beschränkt.
- Iterative Variante von Quicksort: Vermeidet die Rekursion (erfordert jedoch Stack!)

2.3.1 Beweis der Schranken von $\sum_{k=m}^{N} 1/k$ mittels Integral-Methode

Es sei

$$\begin{aligned} f : \mathbb{R}^+ &\to \mathbb{R}^+ \\ x &\mapsto f(x) \end{aligned}$$

eine monoton fallende Funktion, z.B. $f(x) = \dfrac{1}{x}$. Durch Bildung der Unter- und Obersumme erhält man die folgenden Schranken:

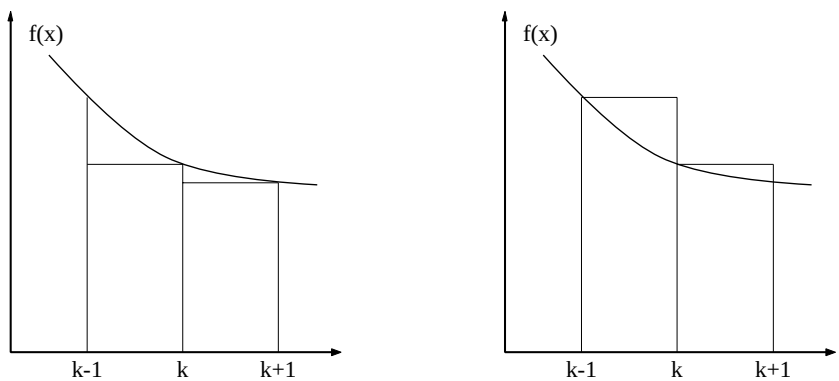


Abbildung 2.5: Unter- und Obersummen

$$\int_{k}^{k+1} f(x)dx \;\leq\; 1 \cdot f(k) \;\leq\; \int_{k-1}^{k} f(x)dx$$

Summation von $k = m, \ldots, n$:

$$\int_{m}^{n+1} f(x)dx \;\leq \sum_{k=m}^{n} f(k) \;\leq\; \int_{m-1}^{n} f(x)dx$$

Insbesondere gilt dann für $f(x) = \dfrac{1}{x}$ und $m \geq 2$

$$\ln \frac{n+1}{m} \;\leq\; \sum_{k=m}^{n} \frac{1}{k} \;\leq\; \ln \frac{n}{m-1}$$

Für $m = 1$ ergeben sich die sog. *Harmonischen Zahlen.*

Definition: Harmonische Zahlen
Als Harmonische Zahlen bezeichnet man die Reihe

$$H_n := \sum_{k=1}^{n} \frac{1}{k}\,, \qquad n \in \mathbb{N}$$

Mit $H_n \;=\; \sum_{k=1}^{n} \frac{1}{k} \;=\; 1 + \sum_{k=2}^{n} \frac{1}{k}$ erhält man die Ungleichung

$$\ln(n+1) \;\leq\; H_n \;\leq\; \ln n + 1$$

Es gilt (ohne Beweis):

$$\begin{aligned} \lim_{n\to\infty} (H_n - \ln n) &= \gamma \quad \text{(Eulersche Konstante)} \\ &\approx 0.5772 \end{aligned}$$

2.4 HeapSort

J.W.J. Williams 1964 und R.W. Floyd 1994

Erweiterung von SelectionSort mittels eines Heaps.

Definition: Heap, Heap-Eigenschaft
Ein *Heap* ist ein links-vollständiger Binärbaum, der in ein Array eingebettet ist (vgl. Abschnitt 1.3.5):

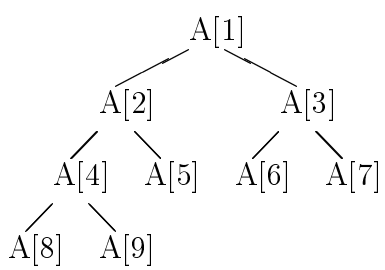
- Ein Array $a[1..N]$ erfüllt die *Heap-Eigenschaft*, falls gilt:

$$a\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \;\geq\; a[i] \qquad \text{für } i = 2, \ldots, N$$

- Ein Array $a[1..N]$ ist ein *Heap beginnend in Position* $l = 1, \ldots, N$, falls:

$$a\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \;\geq\; a[i] \qquad \text{für } i = 2l, \ldots, N$$

Beispiel:



Die Heap-Eigenschaft bedeutet demnach, daß der Schlüssel jedes inneren Knotens größer ist als die Schlüssel seiner Söhne.

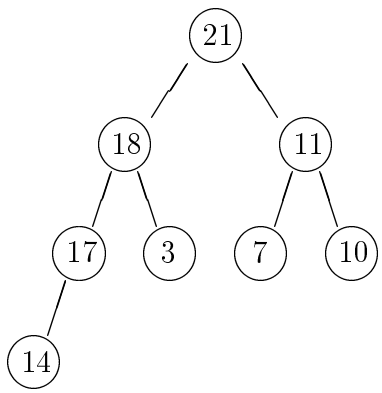
Insbesondere gilt damit für einen Heap im Array $a[1..N]$:

$$a[1] = \max \left\{ a[i] \;\middle|\; i = 1, \ldots, N \right\}$$

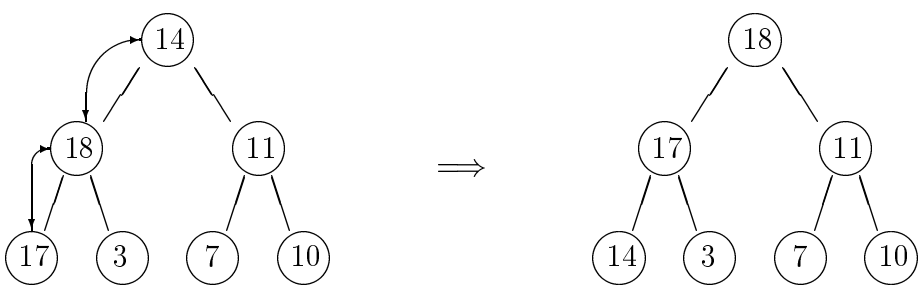
Korrolar: Jedes Array $a[1..N]$ ist ein Heap beginnend in Position $l = \left\lfloor \frac{N}{2} \right\rfloor + 1$

Um die Arbeitsweise von HeapSort zu illustrieren, betrachten wir das folgende Beispiel:

Beispiel
Das folgende Array erfüllt die Heap-Eigenschaft:



Wir entfernen die Wurzel (21) und setzen das letzte Element des Arrays (14) auf die Wurzelposition. Um die Heap-Eigenschaft wieder herzustellen, wird das Element 14 solange mit dem größeren seiner Söhne vertauscht, bis alle Sohnknoten nur noch kleinere Schlüssel enthalten oder aber keine weiteren Sohnknoten existieren. Dieser Schritt heißt auch *Versickern*, *DownHeap* oder *ReHeap*.



Durch das Platzieren des Schlüssels 14 auf die Wurzelposition gewinnen wir einen freien Speicherplatz im Array, in den wir das Element 21 ablegen können.

Algorithmus:

1. Wandle das Array $a[1..N]$ in einen Heap um.
2. `FOR` $i := 1$ `TO` $N - 1$ `DO`
   (a) Tausche $a[1]$ (=Wurzel) und $a[N - i + 1]$
   (b) Stelle für das Rest-Array $a[1..(N - i)]$ die Heap-Eigenschaft wieder her

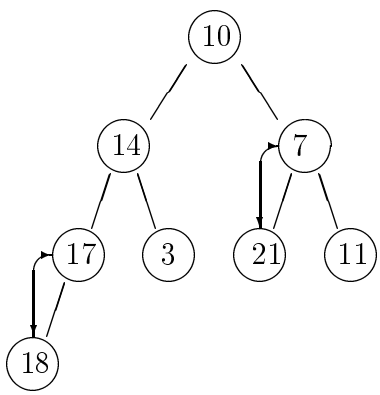
Der erste Schritt, d.h. der *Heap-Aufbau* bleibt zu klären. Wie erwähnt, ist die Heap-Eigenschaft für jedes Array ab der Position $l = \lfloor N/2 \rfloor + 1$ bereits erfüllt. Indem man nun sukzessive die Elemente $a[i]$ mit $i = l - 1, \ldots, 1$ versickern läßt, ergibt sich ein vollständiger Heap.

Beispiel

Ausgangssituation:

10	14	7	17	3	21	11	18

→ Heap



Vertausche 17 ↔ 18

10	14	7	18	3	21	11	17

→ Heap

Vertausche 7 ↔ 21

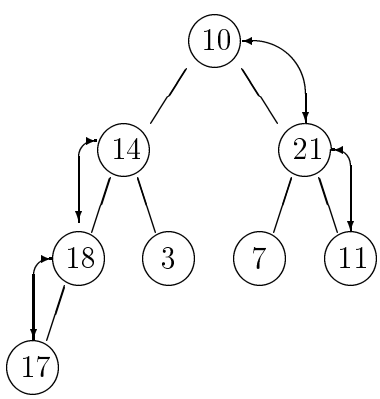
10	14	21	18	3	7	11	17

→ Heap

Versickere 14 ↔ 18 ↔ 17

10	18	21	17	3	7	11	14

→ Heap



Versickere zuletzt 10 ↔ 21 ↔ 11

21	18	11	17	3	7	10	14

Programm:

Gegeben sei ein Array $a[1..N]$. Die Prozedur `DownHeap(i, k, a)` läßt das Element $a[i]$ in dem Teilarray $a[i..k]$ versickern.

```
PROCEDURE HeapSort(VAR a : ARRAY[1..N] OF KeyType) =
  VAR k : CARDINAL;
      t : KeyType;
BEGIN

  (* Heap-Aufbau *)
  FOR i := (N DIV 2) TO 1 BY -1 DO
    DownHeap(i, N, a);
  END;

  (* Sortierung *)
  k := N;
  REPEAT
    t := a[1];
    a[1] := a[k];
    a[k] := t;
    DEC(k);
    DownHeap(1, k, a);
  UNTIL k<=1;

END HeapSort;

PROCEDURE DownHeap(i, k : CARDINAL; VAR a : ARRAY OF KeyType) =
  VAR j : CARDINAL;
      v : KeyType;
BEGIN
  v := a[i];
  LOOP
    IF i<=(k DIV 2) THEN
      j := 2*i;              (* Berechne linken Sohn *)
      IF j<k THEN            (* Existiert rechter Sohn? *)
        IF a[j]<a[j+1] THEN  (* Wähle größeren Sohn *)
          INC(j);
        END;
      END;
      IF a[j]<=v THEN EXIT;  (* Beide Söne kleiner? *)
      END;
      a[i] := a[j];
      i := j;
    ELSE EXIT;               (* Blatt erreicht! *)
    END;
  END;
  a[i] := v;
END DownHeap;
```

Erläuterungen zu der Prozedur `DownHeap(i, k, a)`:

- Start in Position i
- Falls nötig, wird $a[i]$ mit dem größeren der beiden Söhne $a[2i]$ bzw. $a[2i+1]$ vertauscht
  Abfragen:
  - existieren beide Söhne?
  - Blatt erreicht?
- ggf. mit dem Sohn fortfahren

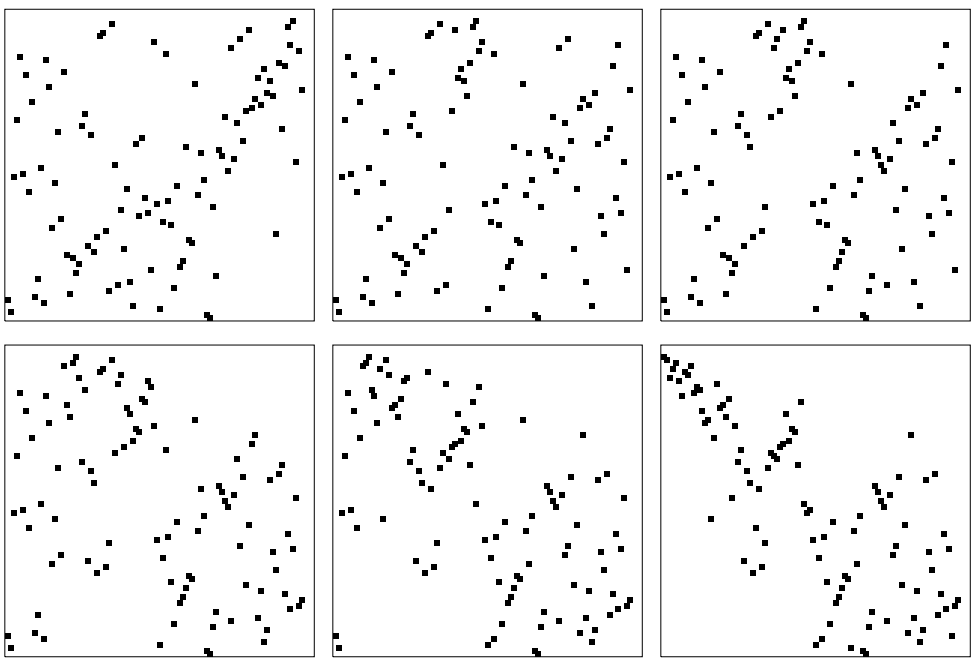


Abbildung 2.6: HeapSort einer zufälligen Permutation von Schlüsselelementen: Aufbau des Heaps

2.4.1 Komplexitätsanalyse von HeapSort

[Mehlhorn]
Wir betrachten die Anzahl der Vergleiche, um ein Array der Größe $N = 2^k - 1$, $k \in \mathbb{N}$ zu sortieren.

Zur Veranschaulichung betrachten wir die Analyse exemplarisch für ein Array der Größe $N = 2^5 - 1 = 31$ (also $k = 5$).

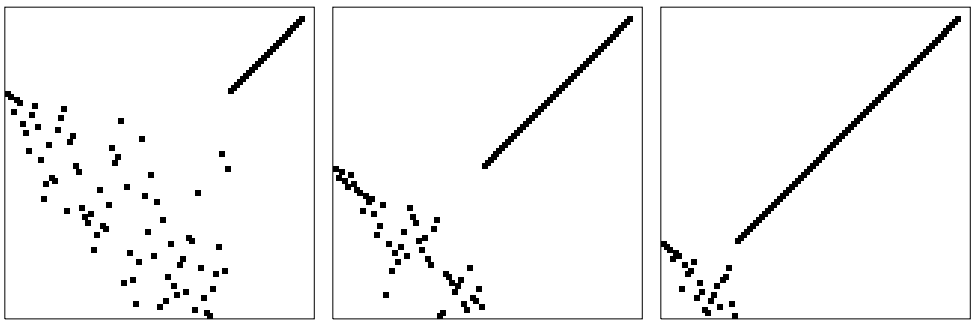
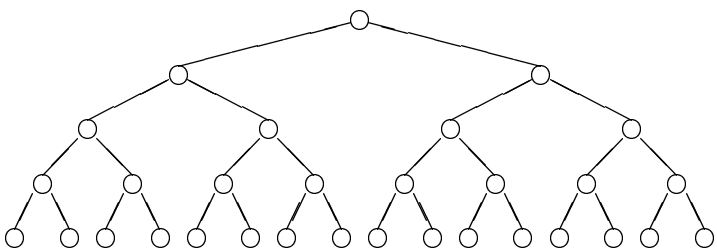


Abbildung 2.7: HeapSort einer zufälligen Permutation von Schlüsselelementen: Sortierphase

Beispiel:



$i = 0$:	$2^0 = 1$	Knoten
$i = 1$:	$2^1 = 2$	Knoten
$i = 2$:	$2^2 = 4$	Knoten
$i = 3$:	$2^3 = 8$	Knoten
$i = 4$:	$2^4 = 16$	Knoten

Heap-Aufbau für Array mit $N = 2^k - 1$ Knoten

- Auf der Ebene i ($i = 0, \ldots, k-1$) gibt es 2^i Knoten.
- Wir fügen ein Element auf dem Niveau $i = (k-2), (k-3), \ldots, 0$ hinzu
- Dieses Element kann maximal auf Niveau $k-1$ sinken.
- Pro Niveau werden dazu höchstens zwei Vergleiche benötigt:
  1. Vergleich: `IF a[j]<=v THEN` ...
     Wird bei jedem Durchlauf der `WHILE`-Schleife ausgeführt, die ihrerseits bei jedem Prozeduraufruf `DownHeap(i, k, a)` mindestens einmal durchlaufen wird.
  2. Vergleich: `IF a[j]<a[j+1] THEN` ...
     Wird ausgeführt, falls 2. Sohn existiert.

  Für die Gesamtzahl der Vergleiche ergibt sich damit die obere Schranke:

$$\sum_{i=0}^{k-2} 2 \cdot (k-1-i) \cdot 2^i = 2^{k+1} - 2(k+1) \tag{2.1}$$

  Beweis durch vollständige Induktion über k.

Sortierphase
Nach dem Aufbau des Heaps muß noch die endgültige Ordnung auf dem Array hergestellt werden. Dazu wird ein Knoten der Tiefe i auf die Wurzel gesetzt. Dieser Knoten kann mit DownHeap maximal um i Niveaus sinken. Pro Niveau sind hierzu höchstens zwei Vergleiche erforderlich. Damit ergibt sich für die Anzahl der Vergleiche die obere Schranke:

$$\sum_{i=0}^{k-1} 2i \cdot 2^i = 2(k-2) \cdot 2^k + 4 \tag{2.2}$$

Beweis durch vollständige Induktion über k

Zusammen:
Sei $N = 2^k - 1$, dann gilt für die Anzahl $T(N)$ der Vergleiche:

$$\begin{aligned} T(N) &\leq 2^{k+1} - 2(k+1) + 2(k-2) \cdot 2^k + 4 \\ &= 2k \cdot (2^k - 1) - 2(2^k - 1) \\ &= 2N \operatorname{ld}(N+1) - 2N \end{aligned}$$

Für $N \neq 2^k - 1$ erhält man ein ähnliches Ergebnis. Die Rechnung gestaltet sich jedoch umständlicher.

Resultat: HeapSort sortiert jede Folge $a[1..N]$ mit höchstens

$$2N \operatorname{ld}(N+1) - 2N$$

Vergleichen.

Bemerkung: In [Güting, S. 196] wird eine Bottom-Up-Variante von HeapSort beschrieben, die die Zahl der erforderlichen Vergleiche auf nahezu $1 \cdot N \operatorname{ld}(N+1)$ senkt.

2.5 Untere und obere Schranken für das Sortierproblem

bisher: Komplexität eines Algorithmus

jetzt: Komplexität eines Problems (Aufgabenstellung)

Ziel: Sei $T_{\mathcal{A}}(N) :=$ Zahl der Schlüsselvergleiche um eine N-elementige Folge von Schlüsselelementen mit Algorithmus $\mathcal{A}$ zu sortieren.

$T_{\min}(N) :=$ Zahl der Vergleiche für den effizientesten Algorithmus

Suche nach einer unteren Schranke:
Gibt es ein $T_0(N)$, so daß

$$T_0(N) \leq T_{\mathcal{A}}(N) \quad \forall \mathcal{A}$$

gilt (d.h. jeder *denkbare* Algorithmus braucht in diesem Falle mindestens $T_0(N)$ Vergleiche) ?

Suche nach einer oberen Schranke:
Wir wählen einen (möglichst effizienten) Sortieralgorithmus $\mathcal{A}$ mit Komplexität $T_{\mathcal{A}}(N)$.

Sprechweise: $T_{\mathcal{A}}(N)$ Vergleiche reichen, um jedes Sortierproblem zu lösen.

Zusammen:

$$T_0(N) \leq T_{\min}(N) \leq T_{\mathcal{A}}(N)$$

Wunsch: $T_0(N)$ und $T_{\mathcal{A}}(N)$ sollen möglichst eng zusammen liegen

Konkret:
Im folgenden betrachten wir für das Sortierproblem nur *Vergleichsoperationen*, d.h. auf der Suche nach einer unteren und oberen Schranke für das Sortierproblem werden wir uns nur auf solche Algorithmen beschränken, die ihr Wissen über die Anordnung der Eingabefolge allein durch (binäre) Vergleichsoperationen erhalten. Dabei werden wir sehen, daß BucketSort die von uns ermittelte untere Schranke durchbricht. Dies hängt damit zusammen, daß BucketSort zusätzliche Bedingungen an die Schlüsselmenge knüpft und somit kein *allgemeines Sortierverfahren* ist.

Obere Schranke:
Wir wählen als „effizienten“ Algorithmus MergeSort.

$$\begin{aligned} T_{\mathcal{A}}(N) &= N \lceil \operatorname{ld} N \rceil - 2^{\lceil \operatorname{ld} N \rceil} + 1 \\ &\leq N \lceil \operatorname{ld} N \rceil - N + 1 \end{aligned}$$

Untere Schranke:
Gegeben sei eine N-elementige Folge.

$$\text{Sortieren: } \widehat{=} \text{ Auswahl einer Permutation dieser Folge}$$

Es gibt $N!$ Permutationen, aus denen die „richtige“ auszuwählen ist.

Beispiel:
Der binäre Entscheidungsbaum aus Abbildung 2.8 „sortiert“ ein 3-elementiges Array $a[1..3]$. Da $3! = 6$, muß der Entscheidungsbaum 6 Blätter besitzen. Wegen $\text{ld } 6 \approx 2.58$ existiert in dem Baum mindestens ein Pfad der Länge 3.

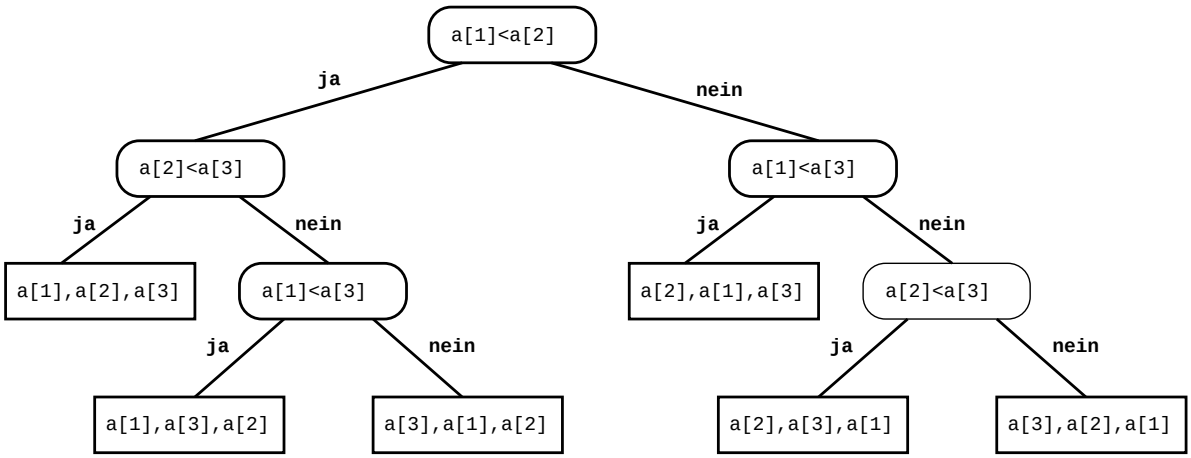


Abbildung 2.8: Binärer Entscheidungsbaum zur Sortierung eines 3-elementigen Arrays

Ein binärer Entscheidungsbaum für das Sortierproblem besitzt genau $N!$ Blätter. Damit ergibt sich als untere Schranke für das Sortierproblem:

$$\text{ld } N! \leq \lceil \text{ld } N! \rceil \leq T_0(N)$$

Mit der Ungleichung (Beweis: siehe Abschnitt 2.6)

$$N \text{ ld } N - N \text{ ld } e \leq \text{ld } N!$$

erhalten wir das Ergebnis:

$$N \text{ ld } N - N \text{ ld } e \leq T_{\min}(N) \leq N \lceil \text{ld } N \rceil - N + 1$$

2.6 Schranken für $N!$

Die Fakultät wird oft benötigt. Eine direkte Berechnung ist aber gleichzeitig umständlich und analytisch schwierig zu behandeln. Aus diesem Grund werden enge Schranken für $N!$ benötigt.

1. **Einfache Schranken:**
   - Obere Schranke:

$$\begin{aligned} N! &= \prod_{i=1}^{N} i \\ &\leq \prod_{i=1}^{N} N \\ &= N^N \end{aligned}$$

   - Untere Schranke:

$$\begin{aligned} N! &= \prod_{i=1}^{N} i \\ &\geq \prod_{i=\lceil N/2 \rceil}^{N} i \\ &\geq \prod_{i=\lceil N/2 \rceil}^{N} \lceil N/2 \rceil \\ &\geq (N/2)^{N/2} \end{aligned}$$

   - Zusammen:

$$(N/2)^{N/2} \leq N! \leq N^N$$

2. **Engere Schranken:**
   Engere Schranken für $N!$ können mittels der *Integral-Methode* berechnet werden. Bei der Integral-Methode wird das Flächenintegral monotoner und konvexer Funktionen von oben und unten durch Trapezsummen approximiert.

   Für die Fakultätsfunktion gilt:

$$\ln N! = \ln \prod_{i=1}^{N} i = \sum_{i=1}^{N} \ln i$$

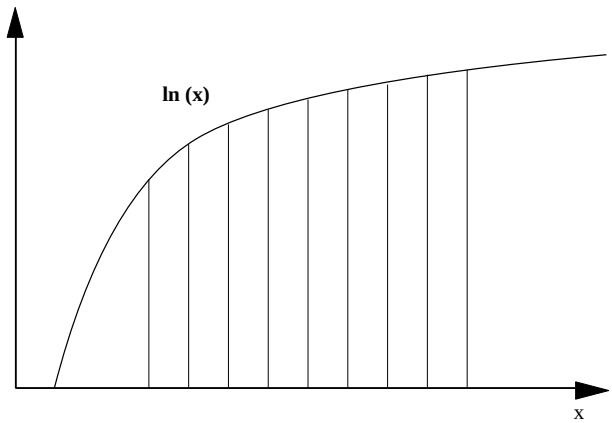


Abbildung 2.9: Graph zu $\ln x$

Untere Schranke:
Die Logarithmusfunktion $x \mapsto f(x) := \ln x$ ist monoton und konvex. Das Integral über $f(x)$ in den Grenzen $i - 1/2$ und $i + 1/2$ bildet daher eine untere Schranke für die Trapez-Obersumme, deren Flächenmaßzahl durch $\ln i$ gegeben ist (vgl. Abbildung 2.10)

$$\int_{i-1/2}^{i+1/2} \ln x \, dx \;\leq\; \ln i$$

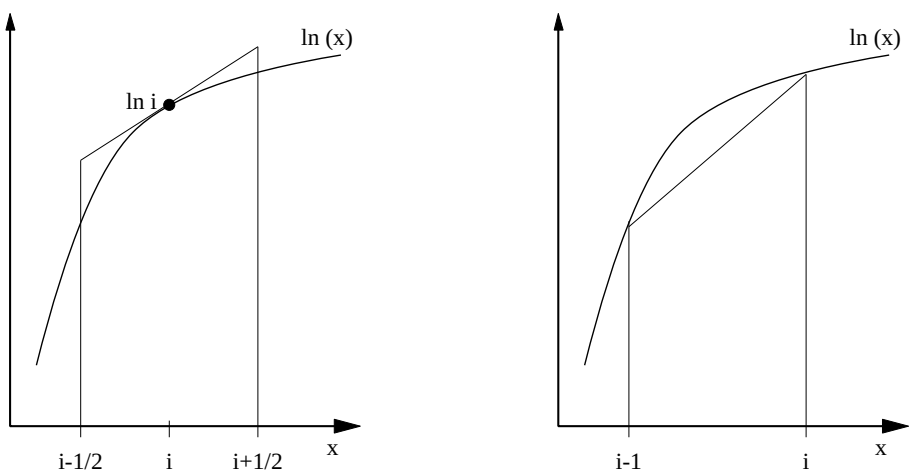


Abbildung 2.10: Trapezober- und Trapezuntersumme

Summation von $i = 1, \ldots, N$ ergibt:

$$\int_{1/2}^{N+1/2} \ln x \, dx \;\leq\; \sum_{i=1}^{N} \ln i \;=\; \ln N!$$

Mit

$$\int_{a}^{b} \ln x \, dx = x \cdot \ln x \;-\; x\Big|_a^b$$

$$= b \cdot \ln b \;-\; a \cdot \ln a$$

folgt die untere Schranke für $\ln N!$

$$N \cdot \ln \frac{N}{e} \;+\; \frac{1}{2} \cdot \ln 2 \;\leq\; \ln N!$$

Obere Schranke:
Das Integral über $\ln x$ in den Grenzen $i-1$ und i ist eine obere Schranke der zugehörigen Trapezuntersumme (vgl. Abbildung 2.10):

$$\frac{1}{2} \cdot \big[\ln(i-1) \;+\; \ln(i)\big] \;\leq\; \int_{i-1}^{i} \ln x \, dx$$

Summation von $i = 2, \ldots, N$ ergibt:

$$\frac{1}{2} \cdot \ln N! \;-\; \ln N \;\leq\; \int_{1}^{N} \ln x \, dx = N \cdot \ln \frac{N}{e} \;+\; 1$$

$$\ln N! \leq N \cdot \ln \frac{N}{e} \;+\; \frac{1}{2} \cdot \ln N \;+\; 1$$

Zusammen:

$$N \cdot \ln \frac{N}{e} \;+\; \frac{1}{2} \cdot \ln N \;+\; \frac{1}{2} \cdot \ln 2 \;\leq\; \ln N! \;\leq\; N \cdot \ln \frac{N}{e} \;+\; \frac{1}{2} \cdot \ln N \;+\; 1$$

Durch die Näherungen ergibt sich der folgende Fehler:

$$N \leq 2: \qquad \sqrt{2} \;\leq\; \frac{N!}{(N/e)^N \cdot \sqrt{N}} \;\leq\; e$$

3. **Engste Schranken und Stirlingsche Formel** (ohne Beweis)
   Es gilt:

$$N! = \sqrt{2\pi N}\left(\frac{N}{e}\right)^N e^{\Theta(1/N)}$$

   mit

$$\frac{1}{12N+1} < \Theta(1/N) < \frac{1}{12N}$$

   zum Beweis: siehe U. Krengel, *Einführung in die Wahrscheinlichkeitstheorie und Statistik*, 3. Auflage, Vieweg Studium, S.78 ff.

   Der Fehler $\Theta(1/N)$ strebt sehr schnell gegen 0:

$$\lim_{N\to\infty} \Theta(1/N) = 0$$

2.7 MergeSort

MergeSort wurde bereits in Abschnitt 1.4.1 behandelt. Die wichtigsten Ergebnisse waren:

- worst case = average case: $N \operatorname{ld} N$
- zusätzlicher Speicherplatz: $O(N)$, nicht in situ, aber sequentiell

2.8 Zusammenfassung

Die folgende Tabelle faßt die Rechenzeiten der in diesem Kapitel behandelten Sortierverfahren zusammen. Sämtliche Algorithmen wurden dabei in RAM-Code (vgl. Kapitel 1.2.1) umgesetzt. Als Kostenmaß wurde das Einheitskostenmaß verwendet.

Verfahren	**Komplexität**	**Referenz**
SelectionSort	$2.5N^2 + 3(N+1)\operatorname{ld} N + 4.5N - 4$	(2.2.1)
QuickSort	$9(N+1)\operatorname{ld}(N+1) + 29N - 33$	(2.3)
HeapSort	$20N \operatorname{ld} N - N - 7$	(2.4)
MergeSort	$12N \operatorname{ld} N + 40N + 97 \operatorname{ld} N + 29$	(1.4.1)

3 Suchen in Mengen

3.1 Problemstellung

Gegeben:
Eine Menge von Records (Elementen), von denen jeder aus einer Schlüssel-Komponente und weiteren Komponenten besteht. In der Regel werden Duplikate ausgeschlossen, wobei sich der Begriff Duplikat beziehen kann:

- auf den Schlüssel;
- auf den vollen Record (exakt: keine Menge mehr).

Typische Aufgabe:

- Finde zu einem vorgegebenen Schlüsselwert den Record und führe gegebenenfalls eine Operation aus.

Damit sind (u.U.) alle Operationen auf Mengen wünschenswert.
Die Darstellung als Menge und deren Verarbeitung kommt oft vor (Datenbanken!).
Im folgenden betrachten wir primär das *Dictionary-Problem* (Wörterbuch-Problem).

Notation: etwa wie [Mehlhorn]

Universum:	$U :=$ Menge aller möglichen Schlüssel
Menge:	$S \subseteq U$

Wörterbuch-Operationen:

Search(x, S):	Falls $x \in S$, liefere den vollen zu x gehörigen Record (oder Information oder Adresse), sonst Meldung: „$x \notin S$“.
Insert(x, S):	Füge Element x zur Menge S hinzu: $S := S \cup \{x\}$ (Fehlermeldung falls $x \in S$).

Delete(x, S): Entferne Element x aus der Menge S: $S := S \backslash \{x\}$ (Fehlermeldung falls $x \notin S$).

Hinweis: Die Terminologie ist nicht standardisiert:
z.B.: Search = Member = Contains = Access

Weitere Operationen:

Order(k, S): Finde das k-te Element in der geordneten Menge S.

ListOrder(S): Produziere eine geordnete Liste der Elemente der Menge S.

Enumerate(S): Zähle alle Elemente der Menge S auf.

FindMin(S): := Order$(1, S)$

Initialize(S): Initialisiere die Darstellung, d.h. $S := \emptyset$.

Weitere Operationen auf Mengen, die prinzipiell in Frage kommen:

Seien S, A Teilmengen von U:

Union (Join): $S := S \cup A$

Intersection: $S := S \cap A$

Difference: $S := S \backslash A$

Seien A_k Teilmengen von U:

Find(x,A_k): Finde die Menge A_k, zu der das Element x gehört.

Bemerkung: Bisher wird nicht berücksichtigt, ob das Universum groß oder klein ist.

Beispiele [Mehlhorn, Seite 97]:

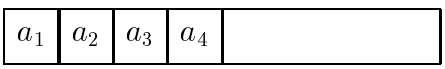
- Symboltabelle (Compiler): 6 Character (z.B. Buchstaben + Ziffern):
  $|U| = (26 + 10)^6 = 2.2 * 10^9$
- Autorenverzeichnis einer Bibliothek in lexikograph./alphabetischer Ordnung
  $|U|$ = ähnliche Größenordnung
- Konten einer Bank (6-stellige Konto-Nummer, 50% davon tatsächlich genutzt)
  $|U| = 10^6$. Hier sind die Größe des Universums und die Menge der benutzten Schlüssel gleich groß.

3.2 Einfache Implementierungen

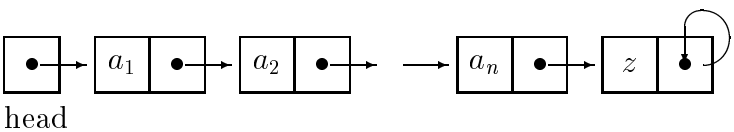
3.2.1 Ungeordnete Arrays und Listen

Darstellung: vgl. Abschnitt 1.3.2

- Liste im Array



- verkettete Liste



3.2.2 Vergleichsbasierte Methoden

Voraussetzung: Existenz einer Vergleichsoperation. Die Annahme einer Ordnung ist keine wirkliche Einschränkung: Es gibt immer eine Ordnung auf der internen Darstellung der Elemente von U [Mehlhorn, Seite 139].

Wir betrachten Suchverfahren für Daten im geordneten Array S mit $S[i] < S[i+1]$ für $1 \leq i \leq n-1$.

Wir unterscheiden:

- sequentielle oder lineare Suche (auch ohne Ordnung möglich);
- Binärsuche;
- Interpolationssuche.

Allgemeines Programmschema:

```
S : array [0..n+1] of element;
S[0] = -∞;
S[n+1] = +∞;

procedure Search(a,S);
  var low, high:  element;
begin
  low := 1; high := n;
```

```
  next := an integer ∈ [low..high]
  while (a ≠ S[next]) and (low < high) do
    begin
      if a < S[next]
      then high := next - 1
      else low := next + 1;
      next := an integer ∈ [low..high]
    end
  if a = S[next]
  then output "a wurde an Position " next " gefunden.";
  else output "a wurde nicht gefunden!";
  return
end;
```

Varianten der Anweisung `next := an integer ∈ [low..high]` :

1. **Lineare Suche** (sequentiell): $next := low$
2. **Binärsuche** (sukzessives Halbieren):

$$next := \left\lceil \frac{high + low}{2} \right\rceil$$

3. **Interpolationssuche** (lineare Interpolation):

$$next := (low - 1) + \left\lceil (high - low + 1) \cdot \frac{a - S[low - 1]}{S[high + 1] - S[low - 1]} \right\rceil$$

Komplexität (Zahl der Schlüsselvergleiche $T(n)$):

1. **Lineare Suche**: $O(n)$
   zwischen 1 und n Operationen:
   a) ungeordneter Array/Liste:
      - $n/2$: erfolgreich
      - n: erfolglos (jedes Element muß geprüft werden)
   b) geordnete Liste:
      - $n/2$: erfolgreich
      - $n/2$: erfolglos

2. **Binärsuche**: $O(\operatorname{ld} n)$
   Rekursionsgleichung:
$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ \Rightarrow T(n) &< \operatorname{ld} n + 1 \end{aligned}$$
   immer weniger als $\operatorname{ld} n + 1$ Vergleiche

3. **Interpolationssuche** (ohne Beweis und Erläuterung):
   - average-case: $\operatorname{ld}(\operatorname{ld} n) + 1$
   - worst-case: $O(n)$ (Entartung in lineare Suche)

Korrektheit des Programms

Das folgende Prädikat P ist eine Invariante der **while** -Schleife:

$$P \equiv (a \in S \Rightarrow a \in S[low..high]) \ \wedge \ (low \leq high \Rightarrow low \leq next \leq high)$$

Beweis:

- vor der Schleife: P ist erfüllt
- in der Schleife:
  es gilt $a \neq S[next]$ und also
  1. entweder $a < S[next]$
     $\Rightarrow a \notin S[next..high]$ (wegen Ordnung!)
     und somit: $a \in S \Rightarrow a \in S[low..next - 1]$
  2. oder $a > S[next]$: analoge Behandlung

Falls $low \leq high$ gilt, folgt: $low \leq next \leq high$ (wegen der ± 1 Änderung)
- Falls also die Schleife terminiert, gilt P und
  1. entweder $a = S[next]$: Suche erfolgreich
  2. oder $low \geq high$
     Sei nun $a \neq S[next]$. Weil P gilt, folgt aus $a \in S[1..n]$, daß $a \in S[low..high]$.
     Nun $low \geq high$:
     a) Falls $high < low$: dann ist $a \notin S[1..n]$
     b) Falls $high = low$: dann ist $next = high$ wegen P
        (insbesondere wegen $a \neq S[next]$) und somit $a \notin S[1..n]$

     in beiden Fällen: Suche erfolglos
- Schleife terminiert:
  In jedem Durchlauf wird $high - low$ mindestens um 1 verringert.

3.2.3 Bitvektordarstellung (Kleines Universum)

Annahme:

$N = |U|$ = vorgegebene maximale Anzahl von Elementen
$S \subset U = \{0, 1, ..., N-1\}$

Methode: Schlüssel = Index in Array
Bitvektor (auch: charakteristische Funktion; Array of Bits):

- $Bit[i]$ = false : $i \notin S$
- $Bit[i]$ = true : $i \in S$

Vergleich der Komplexitäten (O-Komplexität):

(falls möglich: Binärsuche)

$N = |U|$ = die Kardinalität des Universums U, und

$n = |S|$ die Kardinalität der Teilmenge $S \subseteq U$ der tatsächlich vorhandenen Elemente.

Methode	**Zeitkomplexität**			**Platzkomplexität**
	Search	Insert	Delete	
ungeordnete Liste	n	1^*	n	n
ungeordnetes Array	n	1^*	n	N
geordnete Liste	n	n	n	n
geordnetes Array	$\mathrm{ld}\, n^*$	n	n	N
Bitvektor	1	1	1	N

Anmerkungen:

- 1^* heißt: Erfordert mit Duplikateneliminierung $O(n)$.
  $\mathrm{ld}\, n^*$ heißt: Search ist mit Binärsuche implementiert.
- Bitvektor:
  - Operationen $O(1)$ gut, aber:
  - Initialize = $O(N)$
  - Platz = $O(N)$
- Ideal wäre: 3 Operationen mit Zeit O(1) und Platz $O(n)$

3.2.4 Spezielle Array-Implementierung

[Mehlhorn, Seite 270]
Prinzip: Bitvektor-Darstellung mit zwei Hilfsarrays ohne $O(N)$-Initialisierung

andere Namen:

- lazy initialization
- invertierte Liste
- array/stack-Methode

Deklarationen:

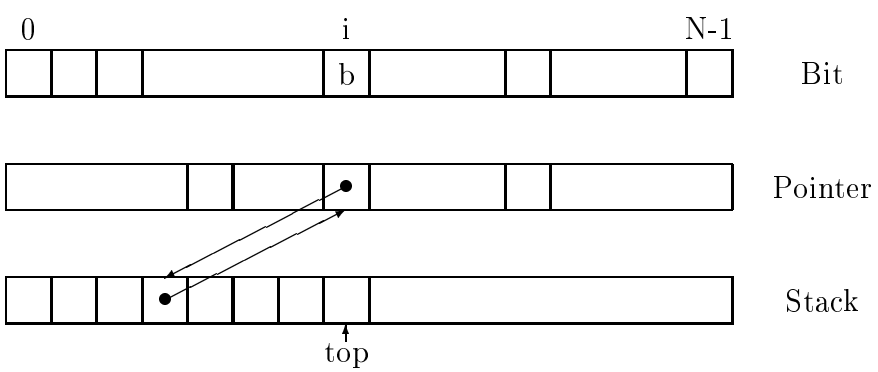
Bit[0..N-1] : **array of boolean**	Bitvektor
Ptr[0..N-1] : **array of integer**	Pointer-Array
Stk[0..N-1] : **array of integer**	Stack-Array

Das Konzept beruht auf der Invarianten: $i \in S$ genau dann, wenn

1. $Bit[i] = \text{true}$ und
2. $0 \leq Ptr[i] \leq top$ und
3. $Stk[Ptr[i]] = i$.

wobei anfangs: $top = -1$

Durch den Zähler *top* und den Backpointer in $Stk[.]$ wird die Initialisierung ersetzt.



Implementierung der Operationen

```
procedure Initialize(S);
  var top :  integer;
begin
  top := -1
end;

function Search(i, S): integer;
begin
  if (0 ≤ Ptr[i] ≤ top) and (Stk[Ptr[i]] = i)
  then return Bit[i]
  else return 'i ∉ S and S not initialized.'
end;
```

Anmerkung: Beim logischen **and** in der **if**-Anweisung sei nochmals auf die Problematik der Auswertung solcher Ausdrücke hingewiesen („Lazy Evaluation“, vgl. Absch. 2.2.2).

Insert(i, S) und Delete(i, S) werden realisiert durch:

setze Bit[i] = b : b = 0: Delete(i,S)
b = 1: Insert(i,S)

```
procedure Insert(i,S) / Delete(i,S);
begin
  if (0 ≤ Ptr[i] ≤ top) and (Stk[Ptr[i]] = i)
    then Bit[i] := b
    else begin
      top := top + 1;
      Ptr[i] := top;
      Stk[top] := i;
      Bit[i] := b
    end;
  return;
end;
```

Anmerkungen:

- Die Variable *top* gibt die maximale Anzahl der jemals angesprochenen Elemente von $S \subseteq U$ an. Bei der Delete-Operation wird nichts zurückgesetzt außer $Bit[i]$.
- Die Information des Bitvektors $Bit[0..N-1]$ könnte auch direkt als zusätzliche Information von jeweils einem Bit im Pointer-Array $Ptr[0..N-1]$ gespeichert werden.
- Der Index des Arrays *Stack* kann statt eines Bits auch einen Pointer auf einen Speicherbereich enthalten, in dem der gesamte Record gespeichert ist.
- Die Methode funktioniert auch für das Initialisieren großer Arrays in numerischen Rechnungen.

Komplexität

- Platzkomplexität: Drei Arrays der Länge N, also $O(N)$.
- Zeitkomplexität für die Standard-Operationen:
  - Initialize(S): $O(1)$
  - Search(i, S): $O(1)$
  - Insert(i, S): $O(1)$
  - Delete(i, S): $O(1)$
  - Enumerate(S): $O(1)$

3.3 Hashing

3.3.1 Begriffe und Unterscheidung

- offenes/geschlossenes Hashing
- Kollisionsstrategien:
  - lineares Sondieren
  - quadratisches Sondieren
  - Doppelhashing
- Hash-Funktionen
- erweiterbares Hashing in Verbindung mit Hintergrundspeicher

Ausgangspunkt:

Bei BucketSort wurde aus dem Schlüssel direkt die Speicher-Adresse berechnet ebenso wie bei der Bitvektor-Darstellung einer Menge.
Hashing kann man als Erweiterung dieser Methode interpretieren, indem die Speicher-Adresse nicht mehr eindeutig umkehrbar (auf den Schlüssel) sein muß, und somit Mehrfach-Belegungen der Speicher-Adresse zulässig sind ('Kollisionen').

Prinzip

Zur Verfügung stehen m Speicherplätze in einer *Hashtabelle* T:

var T: **array** $[0..m-1]$ **of** element .

Dann wird ein Schlüssel $x \in$ Universum $U = \{0, \ldots, N-1\}$ auf einen Speicherplatz in der Hashtabelle abgebildet. Dazu dient die *Hashfunktion* $h(x)$ (deutsch: Schlüssel-Transformation, Streuspeicherung):

$$\begin{aligned} h : U &\rightarrow \{0, 1, ..., m-1\} \\ x &\rightarrow h(x) \end{aligned}$$

Nach dieser Adress-Berechnung kann das Element mit dem Schlüssel x an der Stelle $T[h(x)]$ gespeichert werden, falls dieser Platz noch frei ist.

Da in der Regel $m \ll N$ ist, kann es zu *Kollisionen* kommen, d.h.

$$h(x) = h(y) \text{ , für } x \neq y \text{ .}$$

x wird also nicht notwendigerweise in $T[h(x)]$ selbst gespeichert. Entweder enthält $T[h(x)]$ dann einen Verweis auf eine andere Adresse (offene Hashverfahren), oder mittels einer *Sondierungsfunktion* muß ein anderer Speicherplatz berechnet werden (geschlossene Hashverfahren).

Dementsprechend hat die Operation *Search (x,S)* dann zwei Teile. Zunächst muß $h(x)$ berechnet werden, dann ist x in $T[h(x)]$ zu suchen.

Beispiel: Symboltabelle für Compiler

Ein Anwendungsgebiet für Hashing sind Symboltabellen für Compiler. Das Universum U, nämlich die Menge aller Zeichenketten mit der maximalen Länge 20 (z.B. Namen) ist hier sehr groß: Selbst wenn man nur Buchstaben und Ziffern zuläßt, erhält man

$$|U| = (26 + 10)^{20} = 1.3 \cdot 10^{31}.$$

Somit ist keine umkehrbare Speicherfunktion realistisch.

3.3.2 Hashfunktionen

An die Hashfunktion $h(x)$ werden folgende Anforderungen gestellt:

- Die ganze Hashtabelle sollte abgedeckt werden, d.h. $h(x)$ ist surjektiv.
- $h(x)$ soll die Schlüssel x möglichst gleichmäßig über die Hashtabelle verteilen.
- Die Berechnung soll effizient, also nicht zu rechenaufwendig sein.

Nun wollen wir einige typische Hashfunktionen vorstellen [Güting, S. 109]. Es wird hierbei davon ausgegangen, daß für die Schlüssel gilt: $x \in \mathbb{N}_0$. Fast alle in der Praxis vorkommenden Schlüssel lassen sich entsprechend umwandeln, z.B. durch Umsetzung von Buchstaben in Zahlen.

Divisions-Rest-Methode:

Sei m die Größe der Hashtabelle. Dann definiert man $h(x)$ wie folgt:

$$h(x) = x \mod m.$$

Bewährt hat sich folgende Wahl für m:

- m prim.
- m teilt nicht $2^i \pm j$, wobei i, j kleine Zahlen $\in \mathbb{N}_0$ sind.

Diese Wahl gewährleistet eine surjektive und gleichmäßige Verteilung über die ganze Hashtabelle [Ottmann].

Die Divisions-Rest-Methode hat den Vorteil der einfachen Berechenbarkeit. Der Nachteil ist, daß aufeinanderfolgende Schlüssel auf aufeinanderfolgende Speicherplätze abgebildet werden. Das führt zu unerwünschtem Clustering, welches die Effizienz des Sondierens reduziert (s. Abschnitt 3.3.5).

Beispiel: Verteilen von Namen über m Behälter [Güting, Seite 97].

Die Zeichenketten $c = c_1 \ldots c_k$ werden auf $\mathbb{N}_0$ abgebildet und auf die Hashtabelle verteilt:

$$h(c) := \sum_{i=1}^{k} N(c_i) \mod m$$

mit $N(A) = 1, N(B) = 2, \ldots, N(Z) = 26$.

Zur Vereinfachung werden nur die ersten drei Buchstaben betrachtet:

$$h(c) := [N(c_1) + N(c_2) + N(c_3)] \mod m.$$

Wir wählen $m = 17$; $S =$ deutsche Monatsnamen (ohne Umlaute).

0:	November
1:	April, Dezember
2:	Maerz
3:	
4:	
5:	
6:	Mai, September
7:	
8:	Januar
9:	Juli
10:	
11:	Juni
12:	August, Oktober
13:	Februar
14:	
15:	
16:	

Beachte: Es gibt drei Kollisionen.

Mittel-Quadrat-Methode:

Die Mittel-Quadrat-Methode zielt darauf ab, auch nahe beieinanderliegende Schlüssel auf die ganze Hashtabelle zu verteilen, um ein Clustering (siehe Abschnitt 3.3.5) aufeinanderfolgender Zahlen zu verhindern.

$$h(x) = \text{mittlerer Block von Ziffern von } x^2.$$

Der mittlere Block hängt von allen Ziffern von x ab, und deswegen wird eine bessere Streuung erreicht.

Für $m = 100$ ergibt sich:

x	$x \mod 100$	x^2	$h(x)$
127	27	16**12**9	12
128	28	16**38**4	38
129	29	16**64**1	64

Es gibt noch weitere gebräuchliche Methoden, wie z.B. die *Multiplikative Methode*, auf die wir hier nicht näher eingehen wollen.

3.3.3 Wahrscheinlichkeit von Kollisionen

Das Hauptproblem beim Hashing besteht in der Behandlung von Kollisionen. Bevor wir verschiedene Methoden der Kollisionsbehandlung erörtern, wollen wir berechnen wie häufig Kollisionen auftreten.

In der Mathematik gibt es ein analoges, unter dem Namen *Geburtstags-Paradoxon* bekanntes Problem:

Wie groß ist die Wahrscheinlichkeit, daß mindestens 2 von n Personen am gleichen Tag Geburtstag haben ($m = 365$) ?

Analogie zum Hashing:

$m = 365$ Tage $\equiv$ Größe der Hash-Tabelle;
n Personen $\equiv$ Zahl der Schlüssel.

Annahme: Die Hash-Funktion sei ideal, d.h. die Verteilung über die Hash-Tabelle sei gleichmäßig.

Für die Wahrscheinlichkeit mindestens einer Kollision bei n Schlüsseln und einer m-elementigen Hashtabelle $Pr(Kol|n,m)$ gilt:

$$Pr(Kol|n,m) = 1 - Pr(NoKol|n,m).$$

Sei $p(i;m)$ die Wahrscheinlichkeit, daß der i-te Schlüssel ($i = 1,\ldots,n$) auf einen freien Platz abgebildet wird, wie alle Schlüssel zuvor auch. Es gilt

$$\begin{aligned}
p(1;m) &= \frac{m-0}{m} = 1 - \frac{0}{m} && \text{, weil 0 Plätze belegt und } m-0 \text{ Plätze frei sind} \\
p(2;m) &= \frac{m-1}{m} = 1 - \frac{1}{m} && \text{, weil 1 Platz belegt und } m-1 \text{ Plätze frei sind} \\
&\vdots \\
p(i;m) &= \frac{m-i+1}{m} = 1 - \frac{i-1}{m} && \text{, weil } i-1 \text{ Plätze belegt und } m-i+1 \text{ Plätze frei sind}
\end{aligned}$$

$Pr(NoKol|n,m)$ ist dann das Produkt der Wahrscheinlichkeiten $p(1;m)$ bis $p(n;m)$

$$\begin{aligned}
Pr(NoKol|n,m) &= \prod_{i=1}^{n} p(i;m) \\
&= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right)
\end{aligned}$$

Tabelle zum Geburtstagsproblem ($m = 365$):

n	$Pr(Kol\|n,m)$
10	0,11695
20	0,41144
$\vdots$	$\vdots$
22	0,47570
23	0,50730
24	0,53835
$\vdots$	$\vdots$
30	0,70632
40	0,89123
50	0,97037

Die Wahrscheinlichkeit für einen gleichzeitigen Geburtstag zweier Personen aus einer Gruppe von 23 Personen beträgt also 50,7%.

Approximation

Frage: Wie muß m mit n wachsen, damit $Pr(Kol|n,m)$ und also auch $Pr(NoKol|n,m)$ konstant bleibt. Die obige Formel gibt auf diese Frage nur eine indirekte Antwort. Darum die folgende Vereinfachung:

$$\begin{aligned} Pr(NoKol|n,m) &= \prod_{i=0}^{n-1}\left(1-\frac{i}{m}\right) \\ &= \exp\left[\sum_{i=0}^{n-1}\ln\left(1-\frac{i}{m}\right)\right] \end{aligned}$$

Nun verwendet man: $\ln(1+\varepsilon)\approx\varepsilon$ für $\varepsilon \ll 1$, d.h. $\frac{i}{m} < \frac{n}{m} \ll 1$ (siehe Abbildung 3.1).

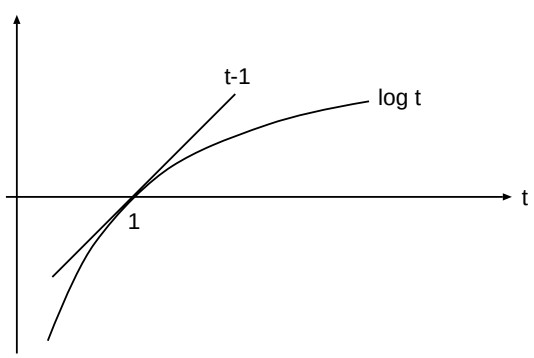


Abbildung 3.1: Eigenschaft des Logarithmus

$$\begin{aligned} Pr(NoKol|n,m) &\approx \exp\left[-\sum_{i=0}^{n-1}\frac{i}{m}\right] \\ &= \exp\left[-\frac{n(n-1)}{2m}\right] \end{aligned}$$

Also folgt für die Wahrscheinlichkeit keiner Kollision

$$Pr(NoKol|n,m) \simeq \exp\left[-\frac{n^2}{2m}\right]$$

Ergebnis: $Pr(NoKol|n,m)$ bleibt etwa konstant, wenn die Größe m der Hashtabelle quadratisch mit der Zahl der Elemente n wächst.

Übungsaufgaben:

1. Die obige Approximation ist eine obere Schranke, finden sie eine enge untere Schranke.
2. Geben sie eine gute Approximation an, in der nur „paarweise“ Kollisionen betrachtet werden, d.h. das *genau* zwei Elemente auf eine Zelle abgebildet werden.

Terminologie

In den folgenden Abschnitten wollen wir uns mit den Vorgehensweisen bei Kollisionen beschäftigen. Da die Terminologie auf diesem Gebiet in der Literatur aber sehr uneinheitlich ist, wollen wir kurz definieren, welche wir benutzen.

Als *Offene Hashverfahren* bezeichnen wir Verfahren, die mit dynamischem Speicher (verketteten Listen) arbeiten, und daher beliebig viele Schlüssel unter einem Hashtabellen-Eintrag unterbringen können.

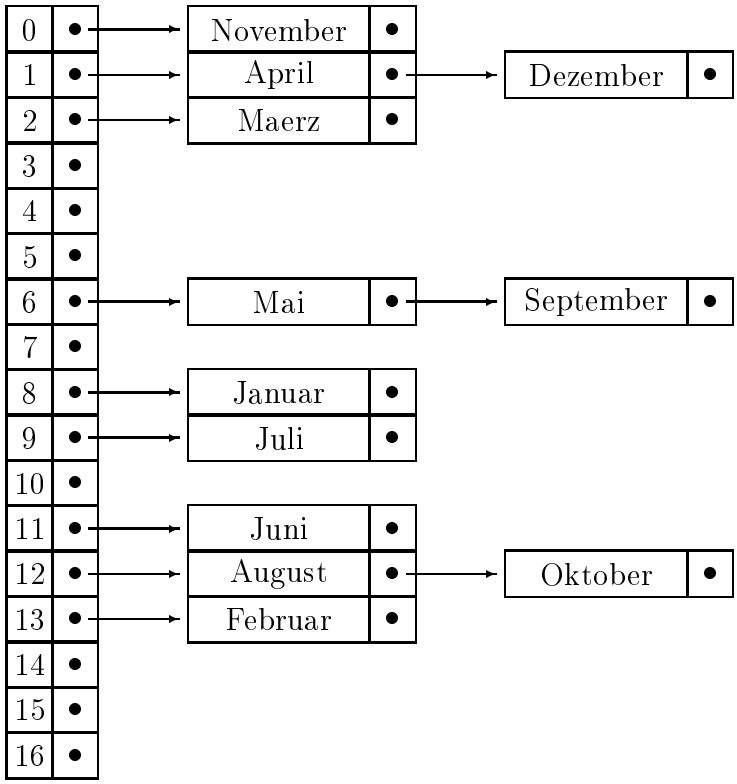
Geschlossene Hashverfahren hingegen können nur eine begrenzte Zahl von Schlüsseln (meistens nur einen) unter einem Eintrag unterbringen. Sie arbeiten mit einem Array und werden wegen des abgeschlossenen Speicherangebots als geschlossen bezeichnet. Sie müssen bei Kollisionen mittels Sondierungsfunktionen neue Adressen suchen.

3.3.4 Offenes Hashing (Hashing mit Verkettung)

Jeder Behälter wird durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt. Ein Array von Zeigern verwaltet die Behälter.

var HashTable : **array** [0..m-1] **of** ↑ListElem.

Für obiges Beispiel mit den Monatsnamen ergibt sich:



Kostenabschätzung: Die drei Operationen Insert(x, S), Delete(x, S), Search(x, S) setzen sich dann aus zwei Schritten zusammen:

1. Berechnung der Adresse und Aufsuchen des Behälters.
2. Durchlaufen der unter der Adresse gespeicherten Einträge.

Der *Belegungsfaktor* $\alpha = \frac{n}{m}$ gibt auch gleichzeitig die durchschnittliche Listenlänge an. Daraus ergibt sich für die *Zeit* folgende Kostenabschätzung:

- Adresse berechnen: $O(1)$
- Behälter aufsuchen: $O(1)$
- Liste durchsuchen:
  - Im Average Case, also bei erfolgreicher Suche: $O(1 + \frac{\alpha}{2})$
  - Im Worst Case, also bei erfolgloser Suche: $O(\alpha) = O(n)$

Die *Platzkomplexität* beträgt $O(m + n)$.

Verhalten:

- $\alpha \ll 1$: wie Array-Addressierung
- $\alpha \approx 1$: Übergangsbereich
- $\alpha \gg 1$: wie verkettete Liste

3.3.5 Geschlossene Hashverfahren (Hashing mit offener Adressierung)

[Güting, S.100] [Mehlhorn, S.117]
Diese Verfahren arbeiten mit einem statischen Array

var HashTable : **array** [0..m-1] **of** Element;

und müssen daher Kollisionen mit einer neuerlichen Adressberechnung behandeln („Offene Adressierung"). Dazu dient eine *Sondierungsfunktion* oder *Rehashing-Funktion*, die eine Permutation aller Hashtabellen-Plätze und damit die Reihenfolge der zu sondierenden Plätze angibt:

- jedes $x \in U$ definiert eine Folge $h(x, j)$, $j = 0, 1, 2, ...$ von Positionen in der Hash-Tabelle.
- diese Folge muß bei jeder Operation durchsucht werden.
- Definition der zusätzlichen Hash-Funktionen $h(x, j)$, $j = 0, 1, 2, ...$ erforderlich

Achtung: Die Operation „Delete(x, S)" erfordert eine Sonderbehandlung, weil die Elemente, die an x mittels Rehashing vorbeigeleitet wurden, auch nach Löschen von x noch gefunden werden müssen. Dazu führt man zusätzlich zu den Werten aus U die Werte „empty" und „deleted" ein, wobei „deleted" kennzeichnet, daß nach diesem Feld noch weitergesucht werden muß.

Lineares Sondieren

Die (Re)Hash-Funktionen $h(x, i)$, $i = 0, 1, 2, ...$ sollen so gewählt werden, daß für jedes x der Reihe nach sämtliche m Zellen der Hash-Tabelle inspiziert werden.

Einfachster Ansatz: Lineares Sondieren

$$h(x, j) = (h(x) + j) \mod m \qquad 0 \leq j \leq m - 1$$

Lineares Sondieren tendiert zur primären Häufung (primary clustering). Das bedeutet, daß lange kontinuierliche Teilstücke wesentlich schneller wachsen als kurze, da ein Schlüssel, sobald er einmal auf ein solches Stück trifft, bis an dessen Ende rutscht, und es so noch weiter verlängert. Diese langen Stücke machen dann das Sondieren ineffizient.

Beispiel: Insert(x, S) mit Monatsnamen in natürlicher Reihenfolge

0:	November	
1:	April	(Dezember)
2:	Maerz	
3:	Dezember	
4:		
5:		
6:	Mai	(September)
7:	September	
8:	Januar	
9:	Juli	
10:		
11:	Juni	
12:	August	(Oktober)
13:	Februar	
14:	Oktober	
15:		
16:		

Verallgemeinerung:

$$h(x,i) = (h(x) + c \cdot i) \mod m \qquad 1 \leq i \leq m-1$$

c ist eine Konstante, wobei c und m teilerfremd sein sollten.
Keine Verbesserung: Esb bilden sich Ketten mit Abstand c.

Quadratisches Sondieren

Um die oben angesprochene Häufung zu vermeiden, kann man mit folgender (Re-)Hashfunktion quadratisch Sondieren

$$h(x,j) = (h(x) + j^2) \mod m \quad \text{für j=0, ..., m-1}$$

Warum i^2 ? Wenn m eine Primzahl ist, dann sind die Zahlen $i^2 \mod m$ alle verschieden für $i = 0, 1, \ldots, \left\lfloor \frac{m}{2} \right\rfloor$.

Verfeinerung: $0 \leq j \leq \frac{m-1}{2}$

$$\begin{aligned} h(x, 2j-1) &= (h(x)+j^2) \mod m \\ h(x, 2j) &= (h(x)\text{-}j^2) \mod m \end{aligned}$$

Wähle m prim mit $m \mod 4 = 3$ (Basis: Zahlentheorie).

Quadratisches Sondieren kann zwar auch keine Primärkollisionen verhindern, weil für $h(x) = h(y)$ auch $h(x,0) = h(y,0)$ ist, jedoch wird *Primäres Clustering* verhindert, es entstehen also keine langen Ketten. Sekundäres Clustering tritt bei dieser Sondierungsart jedoch auch auf. D.h., daß Schlüssel mit gleichem Hashwert ($h(x) = h(y)$) auch auf die gleiche Sondierungsbahn gebracht werden.

Doppel-Hashing

Beim Doppel-Hashing wählt man zwei Hash-Funktionen $h(x)$ und $h'(x)$ und nehme an:

$$\begin{aligned} Pr(h(x) = h(y)) &= \frac{1}{m} \\ Pr(h'(x) = h'(y)) &= \frac{1}{m} \ . \end{aligned}$$

Falls die beiden Funktionen h und h' unabhängig sind, gilt

$$Pr(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2} \ .$$

Wir definieren

$$h(x,i) = (h(x) + h'(x) \cdot i^2) \mod m \ .$$

Eigenschaften:

- mit Abstand die beste der hier vorgestellten Kollisionsstrategien,
- kaum unterscheidbar von idealem Hashing.

Komplexität des geschlossenen Hashings

Wir wollen nun die Kosten für geschlossene Hashverfahren analysieren [Mehlhorn, Seiten 118f.] [Aho et al. 83]. Dabei interessieren uns die Kosten der drei Standard-Operationen *Search, Insert* und *Delete*. Um jedoch Aussagen darüber machen zu können, benötigen wir Aussagen über die Kollisionswahrscheinlichkeit.

Gegeben sei eine *ideale Hashfunktion*, d.h. alle Plätze sind gleichwahrscheinlich und es gibt keine Kettenbildung.

$q(i;n,m) :=$ Wahrscheinlichkeit, daß mindestens i Kollisionen auftreten, wenn m die Größe der Hash-Tabelle ist und bereits n Elemente eingetragen sind.

$=$ Wahrscheinlichkeit, daß die ersten $i-1$ Positionen $h(x,0)$, $h(x,1)$,..., $h(x,i-1)$ der Rehashing-Strategie besetzt sind (für n und m).

Dann gilt für

i=1:

$$q(1;n,m) = \frac{n}{m}$$

i=2: Nach der ersten Kollision werden beim Rehashing $m-1$ Zellen geprüft, von denen $n-1$ besetzt sind

$$q(2;n,m) = \frac{n}{m} \cdot \frac{n-1}{m-1}$$

i: Nach $i-1$ Kollisionen testet die Sondierungsfunktion noch $m-(i-1)$ Zellen, von denen $n-(i-1)$ besetzt sind. Mittels Rekursion ergibt sich so

$$\begin{aligned} q(i;n,m) &= q(i-1;n,m) \cdot \frac{n-i+1}{m-i+1} \\ &\vdots \\ &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ \ldots \ \cdot \frac{n-i+1}{m-i+1} \\ &= \prod_{j=0}^{i-1} \frac{n-j}{m-j} \end{aligned}$$

- $C_{Ins}(n,m)$ seien die mittleren Kosten von Insert(x,S), also der Eintragung des $(n+1)$-ten Elementes in eine m-elementige Hashtabelle. Dann gilt

$$\begin{aligned} C_{Ins}(n,m) &= \sum_{i=0}^{n} q(i;n,m) \\ &= \frac{m+1}{m+1-n} \\ &\approx \frac{1}{1-\alpha} \quad \text{mit } \alpha := \frac{n}{m} \end{aligned}$$

  Der Beweis kann mittels vollständiger Induktion über m und n geführt werden.

- $C^{-}_{Sea}(n,m)$ seien die mittleren Kosten von Search(x,S) bei erfolgloser Suche.

  Dazu werden die Positionen $h(x,0), h(x,1), \ldots$ getestet. Bei der ersten freien Zelle wird abgebrochen. Also

$$C^{-}_{Sea}(n,m) = C_{Ins}(n,m)$$

- $C^{+}_{Sea}(n,m)$ seien die mittleren Kosten von Search(x,S) bei erfolgreicher Suche.

Dabei werden alle Positionen $h(x,0), h(x,1), \ldots$ durchlaufen, bis wir ein i finden mit $T[h(x,i)] = x$. Dieses i hat auch die Kosten für die Operation Insert(x,S) bestimmt, und so erhält man durch Mittelung über alle Elemente $j = 0, \ldots, n-1$ aus $C_{Ins}(j,m)$ folgende Kosten

$$\begin{aligned}
C^+_{Sea}(n,m) &= \frac{1}{n} \cdot \sum_{j=0}^{n-1} C_{Ins}(j,m) \\
&= \frac{m+1}{n} \cdot \sum_{j=0}^{n-1} \frac{1}{m+1-j} \\
&= \frac{m+1}{n} \cdot \left[\sum_{j=1}^{m+1} \frac{1}{j} - \sum_{j=1}^{m+1-n} \frac{1}{j} \right] \\
&\vdots \qquad \text{(harmonische Zahlen)} \\
&\approx \frac{m+1}{n} \cdot \ln \frac{m+1}{m+1-n} \\
&\approx \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}
\end{aligned}$$

- $C_{Del}(n,m)$ seien die mittleren Kosten von Delete(x,S).

  Dann werden alle Zellen $h(x,0), h(x,1), \ldots$ durchlaufen bis $T[h(x,i)] = x$. Also entsprechen die Kosten denen der erfolgreichen Suche:

$$C_{Del}(n,m) := C^+_{Sea}(n,m)$$

Näherung für die Kostenanalyse: Falls $n, m \gg i$:

$$q(i;n,m) = \left(\frac{n}{m}\right)^i$$

Damit können wir die Formeln der geometrischen Reihe verwenden und einfacher rechnen. Es genügt, die beiden Arten von Kosten zu betrachten:

$$\begin{aligned}
C^-_{Sea}(n,m) = C_{Ins}(n,m) &\approx \sum_{i=0}^{\infty} \left(\frac{n}{m}\right)^i \\
&= \frac{1}{1-\alpha}
\end{aligned}$$

$$\begin{aligned}
C^+_{Sea}(n,m) = C_{Del}(n,m) &\approx \frac{1}{n} \int_0^{n-1} \frac{m}{m-x}\, dx \\
&= \frac{m}{n} \ln \frac{m}{m+1-n} \\
&\approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
\end{aligned}$$

Nichtideales Hashing: Clustering (Kettenbildung) mit linearem Sondieren:

Formeln von Knuth[1973]
Zwei Größen genügen:

$$C^{+}_{Sea}(n,m) = \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

$$C^{-}_{Sea}(n,m) = \frac{1}{2}\left[1+\frac{1}{1-\alpha^2}\right]$$

3.3.6 Zusammenfassung der Hashverfahren

Im Average Case zeigen die Hashverfahren allgemein ein effizientes Verhalten ($O(1)$), erst im Worst Case liegen die Operationen bei $O(n)$.

Ein Nachteil ist, daß alle Funktionen die auf einer Ordnung basieren, z.B. *ListOrder(S)* nicht unterstützt werden.

Anwendungen liegen dort, wo $|U| \gg |S|$ und dennoch ein effizienter Zugriff auf die Elemente wünschenswert ist. Als Beispiel haben wir die Symboltabellen von Compilern kennengelernt.

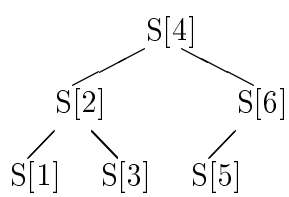
3.4 Binäre Suchbäume

3.4.1 Allgemeine binäre Suchbäume

[Mehlhorn, Seite 140]
Ausgangspunkt: Binäre Suche

$$next := \left\lceil \frac{high + low}{2} \right\rceil$$

Veranschaulichung durch binären Baum für Array S[1..6]:



Zeitkomplexität für die Array-Darstellung:

- Search:

  - Der Wert von $[high - low + 1]$ wird bei jedem Durchgang der While-Schleife halbiert.
  - höchstens: $\operatorname{ld} n$ Zeitkomplexität

- Insert/Delete-Operationen: problematisch
  z.B. erfordert Insert das Verschieben eines Teils des Arrays. Daher: Zeitkomplexität $O(n)$.
  Ausweg: Wird der obige Suchbaum durch **Zeiger** statt durch ein Array realisiert, laufen auch die Operationen *Delete* und *Insert* effizient ab mit

$$T(n) \in O(\operatorname{ld} n) \ .$$

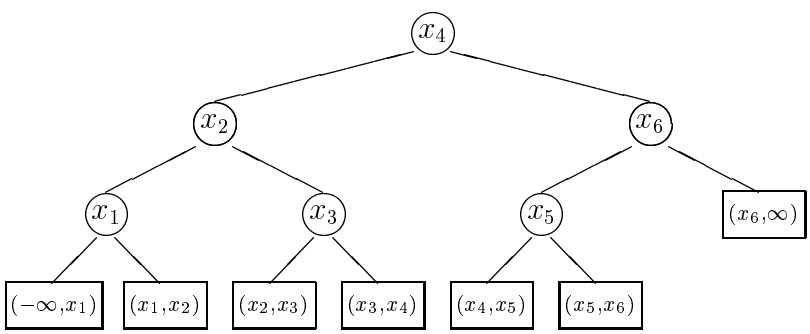
Definition 3.4.1 (binärer Suchbaum) Ein binärer Suchbaum für die n-elementige Menge $S = \{x_1 < x_2 < ... < x_n\}$ ist ein binärer Baum mit n Knoten $\{v_1 ... v_n\}$.
Die Knoten sind mit den Elementen von S beschriftet, d.h. es gibt eine injektive Abbildung $Inhalt : \{v_1 ... v_n\} \to S$. Die Beschriftung bewahrt die Ordnung, d.h. wenn v_i im linken Unterbaum ist, v_j im rechten Unterbaum, v_k Wurzel des Baumes, so gilt:

$$\text{Inhalt } [v_i] < \text{Inhalt } [v_k] < \text{Inhalt } [v_j] \ .$$

Äquivalente Definition: Ein binärer Suchbaum ist ein Binärbaum, dessen *Inorder-Durchlauf* die Ordnung auf S ergibt.

In der Regel identifizieren wir Knoten mit ihrer Beschriftung: Den Knoten v mit Beschriftung x bezeichnen wir also auch mit „Knoten x“.

Beispiel



Die $n + 1$ *Blätter* des Baumes stellen die Intervalle von U dar, die kein Element von S enthalten, also enden erfolglose Suchoperationen entsprechend immer in Blättern.

Somit steht

$$(x_1, x_2) \quad \text{für} \quad \{y \mid y \in U : x_1 < y < x_2\}$$

Da die Blätter sich aus S ergeben, müssen sie nicht explizit abgespeichert werden.

Beispiel: Deutsche Monatsnamen
Dargestellt ist ein binärer Suchbaum in lexikographischer Ordnung entstanden durch Einfügen der Monatsnamen in kalendarischer Reihenfolge.

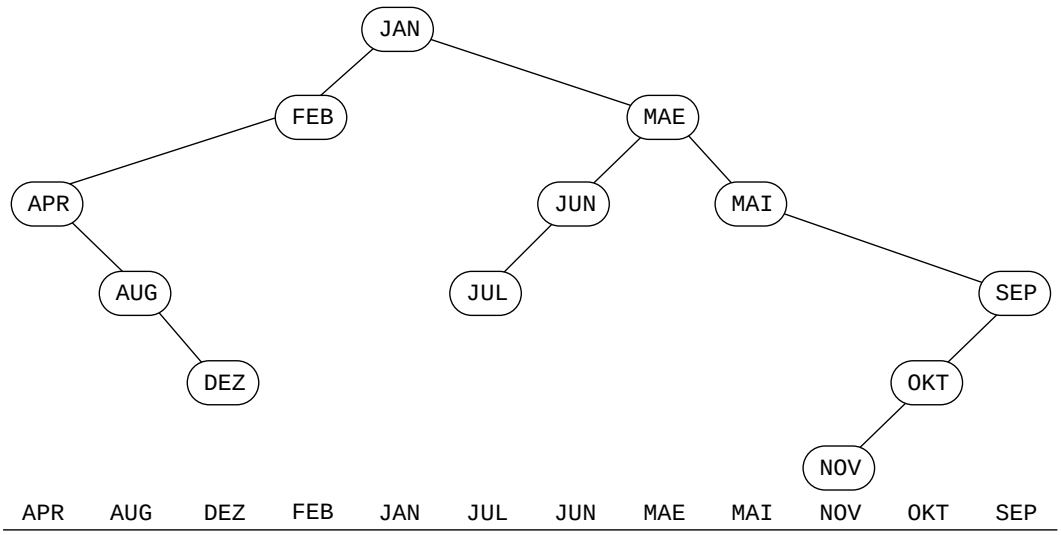


Abbildung 3.2: Binärer Suchbaum für deutsche Monatsnamen

Die Inorder-Traversierung bzw. die Projektion auf die x-Achse erzeugt die alphabetische Reihenfolge.

Übung: Erstellen Sie den Baum für die umgekehrte Reihenfolge.

Programm: Binäre Suchbäume [Mehlhorn, Seite 141]

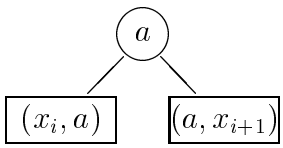
```
procedure Search(a,S);     (* mit expliziten Blaettern *)
begin
  v := Root of T;
  while (v is node) and (a ≠ Inhalt[v]) do
    if (a < Inhalt[v])
      then v = LeftSon[v]
      else v = RightSon[v]
end;
```

Das obige Programm Search endet

- in einem Knoten v, falls $a = Inhalt[v]$;
- in einem Blatt (x_i, x_{i+1}) mit $x_i < a < x_{i+1}$, falls $a \notin S$.

Dann haben wir:

- **Insert**(a, S): ersetze das Blatt (x_i, x_{i+1}) mit dem Intervall, in dem a liegt durch den folgenden Teilbaum



- **Delete**(a, S): Diese Operation gestaltet sich etwas komplizierter, denn falls der Knoten a einen oder mehrere Söhne hat, kann er nicht einfach entfernt werden, sondern einer seiner Söhne muß dann an seine Stelle treten. Blätter, die ja nicht explizit gespeichert werden, wollen wir nicht als Söhne betrachten. Man geht dann folgendermaßen vor:

  Die Suche endet in einem Knoten v mit $Inhalt[v] = a$. Endet sie in einem Blatt, erhält man eine Fehlermeldung.
  Fallunterscheidung:

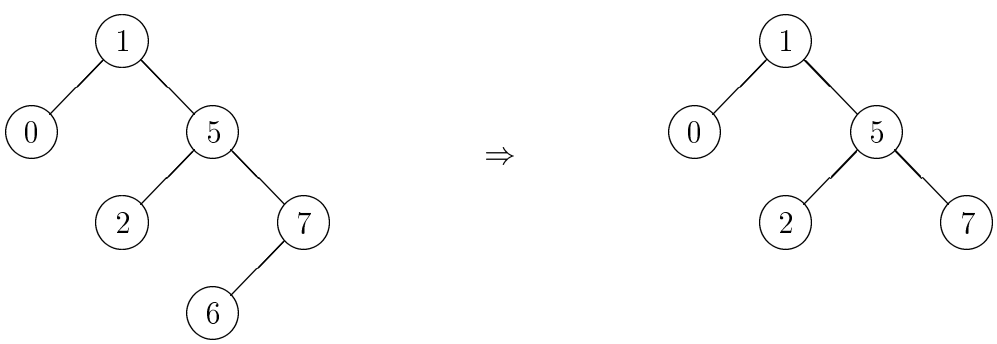
  a) Der Knoten v hat mindestens ein Blatt als Sohn:
     - ersetze v durch den anderen Sohn und
     - streiche v und das Blatt aus dem Baum.

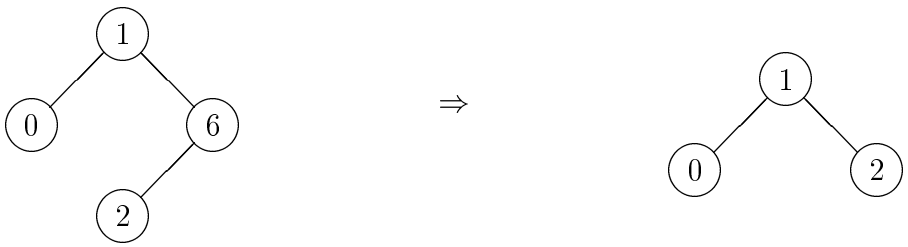
  b) Der Knoten v hat zwei Söhne die innere Knoten sind:
     - sei w der rechteste Unterknoten im linken Unterbaum von v
       Dieser wird gefunden durch:
       * $LeftSon[v]$
       * rekursiv $RightSon[v]$, bis wir auf ein Blatt treffen.
     - $Inhalt[v] = Inhalt[w]$
     - um w zu entfernen, fahre fort wie in Fall a).

  Dabei bleibt die Suchbaum-Eigenschaft erhalten!

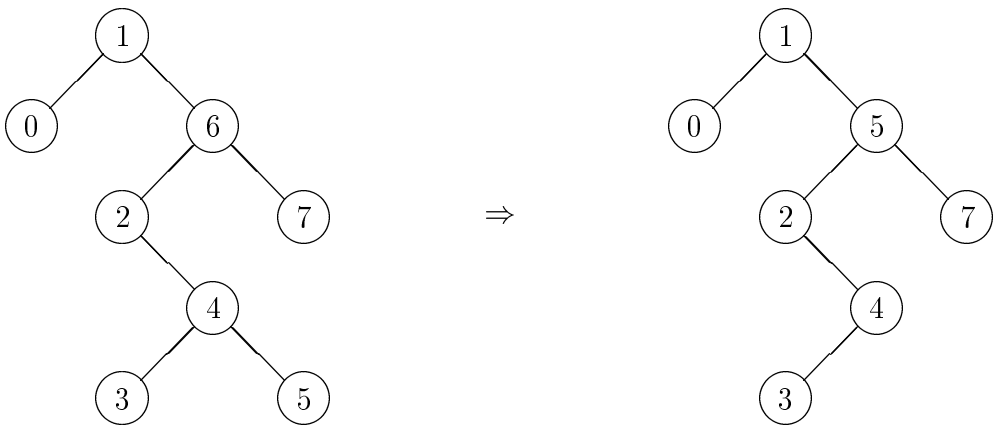
Beispiel: Delete(6,S) (ähnlich [Mehlhorn, Seite 143])

Fall a): Der Knoten 6 hat mindestens ein Blatt als Sohn.





Fall b): Der Knoten 6 hat zwei Söhne.



Zeitkomplexitäten

Standard-Operationen *Search*, *Insert* und *Delete*:

- im wesentlichen: Baum-Traversieren u. einige lokale Änderungen, die $O(1)$ Zeitkomplexität haben.
- deswegen: insgesamt $O(h(T))$, wobei $h(T)$ die Höhe des Suchbaums T ist.

Operation *ListOrder(S)*:

- Traversieren in symmetrischer Ordnung
- $O(|S|) = O(n)$

Operation *Order(k, S)* (Ausgabe der k kleinsten oder größten Elemente):

- In jedem Knoten wird zusätzlich die Anzahl der Knoten seines linken Unterbaumes gespeichert. Mit Aufwand $O(1)$ kann dieser Wert bei Delete- und Insert-Operationen aktualisiert werden.
- Damit kann man die Funktion *Order* mit einer Komplexität $O(h(T))$ implementieren (Übungsaufgabe).

Implementierung eines Binären Suchbaums

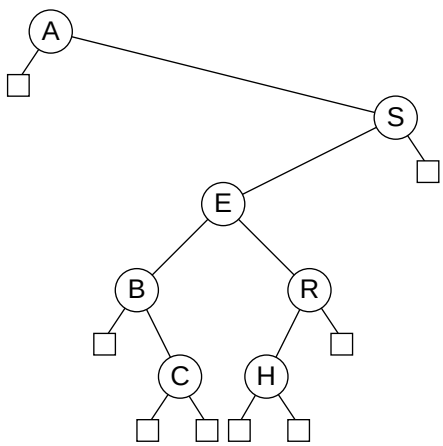
[Sedgewick 88, Seite 208].

Wie üblich wollen wir die drei Standard-Operationen implementieren. Zusätzlich entwerfen wir noch Routinen *Initialize(S)* und *ListOrder(S)* zum Initialisieren bzw. zur geordneten Ausgabe eines binären Suchbaums.

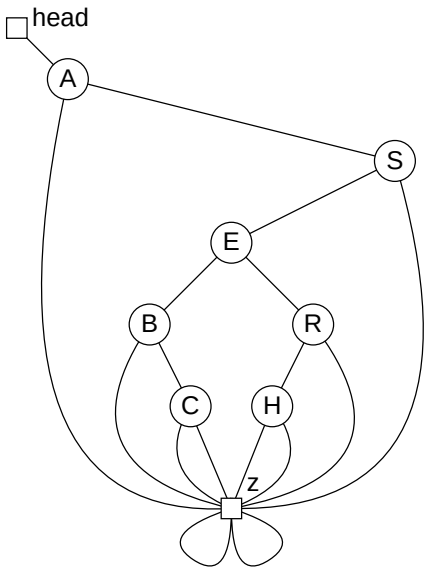
Beachte die Eigenheiten der Implementierung:

- das *Head-Element* `head` (Tree Header Node, Anker, Anchor), also der Listenanfang (nicht zu verwechseln mit der Wurzel), und
- die *Darstellung der Blätter*, z.B. als Dummy Node `z`.

Der folgenden grafischen Darstellung kann man die Verwendung von `head` und `z` ablesen.



Binärer Suchbaum



Implementation mit Head-Element und Dummy-Knoten

Implementierung:

```
type link = ↑node;
     node = record
              key, info :  integer;
              l, r :link
            end;
var head,z:  link;

procedure TreeInitialize;
begin
  new(z); z↑.l := z; z↑.r := z;
  new(head); head↑.key := 0; head↑.r := z;
end;
```

Anmerkung: Falls auch negative Elemente zulässig sind, muß `head ↑.key = -∞` sein.

```
function TreeSearch (v :integer; x:  link) :  link;
begin
  z↑.key := v;
  repeat
    if v < x↑.key
      then x := x↑.l
      else x := x↑.r
  until v = x↑.key;
  TreeSearch := x
end;
```

```
function TreeInsert (v :  integer; x:  link) :  link;
  var p:  link;
begin
  repeat
    p := x;      (* p ist Vorgaenger von x *)
    if v < x↑.key
      then x := x↑.l
      else x := x↑.r
  until x = z;
  new (x); x↑.key := v; x↑.l := z; x↑.r := z; (* x wird ueberschrieben *)
  if v < p↑.key
    then p↑.l := x (* x wird linker Nachfolger von p *)
    else p↑.r := x; (* x wird rechter Nachfolger von p *)
  TreeInsert := x
end;

procedure TreePrint (x:  link); (* inorder *)
begin
  if x <> z then begin
    TreePrint(x↑.l);
    printnode(x);
    TreePrint(x↑.r)
  end
end;
```

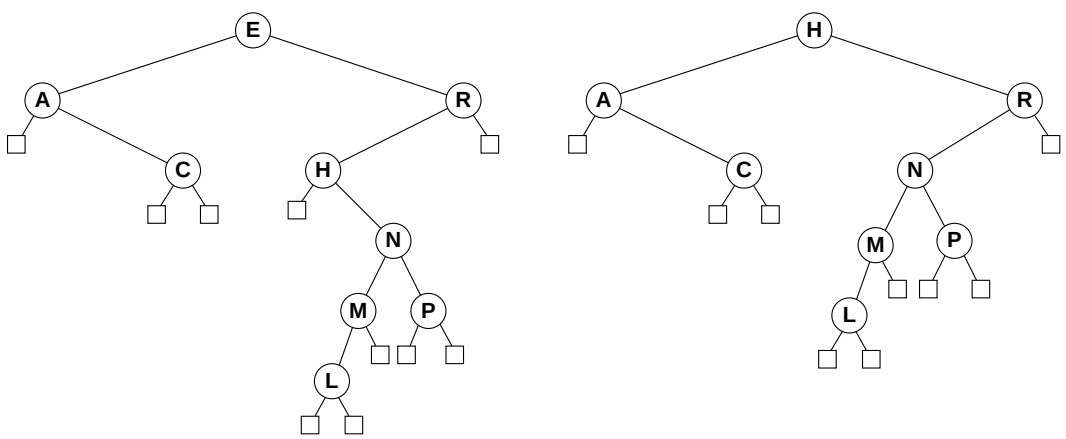


Abbildung 3.3: Beispiel eines Baumes T vor und nach **Delete**(T, E)

Um das im Bild 3.3 dargestellte **Delete**(T, E) zu erreichen geht man wie folgt vor:

1. Suche den kleinsten Schlüssel im rechten Teilbaum von E. Dazu geht man einmal nach rechts und danach immer nach links, bis man auf einen Knoten trifft, der keinen linken Sohn mehr hat, das ist H.

2. Dann macht man, falls vorhanden den rechten Sohn dieses Knotens (also N) zum linken Sohn des Vaterknotens (also R). Dazu wird einfach die Adresse von H im l-Zeiger von R überschrieben mit der von N. Dabei wird *ein Zeiger* geändert.

3. Jetzt muß H noch die Position von E einnehmen, dazu gehört:

   a) Die Nachfolger-Zeiger von H müssen auf die Söhne von E zeigen; also Link l auf A und Link r auf R. Dabei werden *zwei Zeiger* umdirigiert.

   b) Falls notwendig muß das entsprechende Link (l,r) des Vaters von E überschrieben werden durch einen Verweis auf H. Dazu wird *ein Zeiger* umdirigiert.

Insgesamt werden also nur vier Zeiger umgesetzt.

```
procedure TreeDelete (t, x :  link);
  var p, c :  link;
  (* p=parent, c=child *)
begin

  (* Suchen des Knotens t im von x induzierten Teilbaum *)
  repeat
    p := x; (* p ist Vorgänger von x *)
    if t↑.key < x↑.key
      then x := x↑.l
      else x := x↑.r
  until x = t; (* Knoten t ist gefunden *)

  (* t hat keinen rechten Sohn *)
  if t↑.r = z
    then x := t↑.l (* t wird durch seinen linken Sohn ersetzt *)

  (* t hat rechten Sohn, der keinen linken hat *)
  else if t↑.r↑.l = z;
    then begin
      x := t↑.r (* t wird durch seinen rechten Sohn ersetzt *)
      x↑.l := t↑.l (* der rechte Sohn übernimmt den linken Sohn von t*)
    end

  (* sonst *)
  else begin
    c := t↑.r; (* c speichert den Nachfolger von t *)
    while c↑.l↑.l <> z do c := c↑.l;
      (* c↑.l ist nun der linkeste Sohn und c dessen Vater *)
    x := c↑.l; (* t wird durch linkesten Sohn ersetzt *)
    c↑.l := x↑.r; (* c uebernimmt den rechten Sohn von x *)
    x↑.l := t↑.l;
    x↑.r := t↑.r
  end;

  (* Je nachdem ob t linker oder rechter Sohn von p war, *)
  (* muß der entsprechende Zeiger auf x verweisen *)
  if t↑.key < p↑.key
    then p↑.l := x
    else p↑.r := x;
end;
```

Anmerkungen zu TreeDelete(t,x):

- unsymmetrisch: hier Analyse des rechten Teilbaums
- Variable (Zielwerte):
  - p = parent of x (tree to be searched)
  - c = child of t (to be deleted)
- Ergebnis: x = child of p
- Ablauf:
  1. Suchen des Knotens t im Teilbaum x
  2. Fallunterscheidung für t
     a) Kein rechter Sohn (C, L, M, P, R) $\surd$
     b) Rechter Sohn ohne linken Sohn (A, N) $\surd$
     c) sonst: (E, H)
        - suche kleinsten Schlüssel im rechten Teilbaum von t (Name: c) (beachte: c hat keinen linken Sohn)
        - setze Zeiger um.
  3. setze Zeiger für Knoten x

Mit diesen Prozeduren und Funktionen lassen sich die fünf Operationen folgendermaßen realisieren:

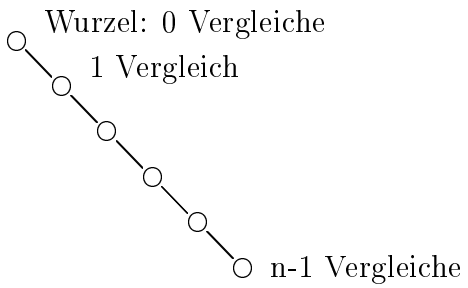
```
Initialize(S):  TreeInitialize()
Search(v,S):    y := TreeSearch(v,head)
Insert(v,S):    y := TreeInsert(v,head)
Delete(v,S):    TreeDelete(TreeSearch(v,head),head)
ListOrder(S):   TreePrint(head↑.r)
```

Komplexitätsanalyse

Annahme: Konstruktion des binären Suchbaums durch n Insert-Operationen.

$C(n)$ = Zahl der Vergleiche.

1. Im *Worst Case* handelt es sich um einen Suchbaum, der zu einer linearen Liste entartetet, da die Elemente in aufsteigender Reihenfolge eingetragen werden.



Also gilt für die Zahl der Vergleiche

$$C(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

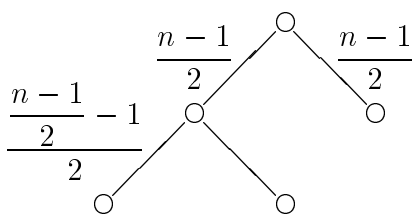
und für den *Aufwand*

$$\frac{C(n)}{n} = \frac{n-1}{2}.$$

2. Im *Best Case* handelt es sich um einen Binärbaum mit minimaler Höhe $h = \mathrm{ld}(n+1)$, in dem man maximal $n = 2^h - 1$ Knoten unterbringen kann (s. Absch. 1.3.5).

   Auf der Wurzelebene (0. Ebene) kann man ohne Vergleich einen Knoten plazieren, auf der ersten Ebene mit jeweils einem Vergleich zwei Knoten, auf der zweiten mit jeweils zwei Vergleichen vier Knoten usw.

   Folgende Darstellung veranschaulicht die Verteilung der n Datensätze auf die Zweige des Binärbaumes.

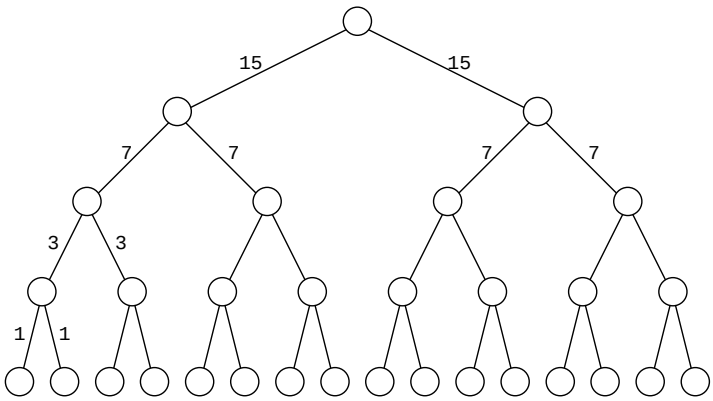


   Über die Anzahl der Vergleiche mit der Wurzel $(2 \cdot \frac{n-1}{2})$ kommt man zu folgender Rekursionsgleichung:

$$C(n) = n - 1 + 2 \cdot C\left(\frac{n-1}{2}\right), \quad C(1) = 0$$

   Beispiel: Die folgende Darstellung veranschaulicht die Herleitung der Formel über Gesamtzahl der Vergleiche, die direkt abhängt von der Knotenanzahl im Baum.

   Sei $h = 5$ und somit $n = 2^h - 1 = 31$.



	Zahl der Vergleiche
i=1	2 * 15 = 30
i=2	4 * 7 = 28
i=3	8 * 3 = 24
i=4	16 * 1 = 16
i=5=h	Summe : 98

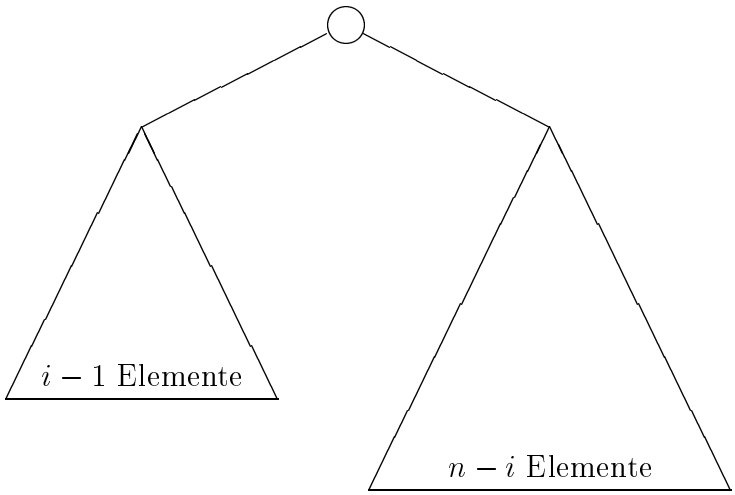
Summiert man nun über die Anzahl *aller* Vergleiche, so erhält man

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{h-1} \underbrace{2^i}_{\text{Äste}} \cdot \underbrace{(2^{h-i}-1)}_{\text{Vergleiche pro Ast}} \\
&= \sum_{i=1}^{h-1} (2^h - 2^i) \\
&= (h-1)\cdot 2^h - \left[\sum_{i=0}^{h-1} 2^i - 1\right] \\
&= (h-1)\cdot 2^h - \frac{2^h-1}{2-1} - 1 \\
&= 2^h \cdot (h-2) + 2 \\
&= (n+1)\cdot \mathrm{ld}(n+1) - 2n
\end{aligned}
$$

und für den Aufwand

$$\frac{C(n)}{n} \approx \mathrm{ld}\, n \ .$$

3. Im *Average Case* werden von n Elementen $i-1$ im linken Teilbaum eingetragen und der Rest abgesehen von der Wurzel im rechten.



Dabei ist jede Aufteilung $i = 1, ..., n$ gleichwahrscheinlich. Für die Anzahl der benötigten Vergleich gilt dann:

$C(n) = n - 1$ Vergleiche, da alle Elemente an der Wurzel vorbei müssen
$+C(i-1)$ Vergleiche im linken Teilbaum
$+C(n-i)$ Vergleiche im rechten Teilbaum

Und somit ergibt sich folgende Anzahl von Vergleichen:

$$\begin{aligned} C(n) &= \frac{1}{n}\sum_{i=1}^{n}[n-1+C(i-1)+C(n-i)] \\ &= n-1+\frac{2}{n}\sum_{i=0}^{n-1}C(i) \qquad \text{(wie Quicksort)} \\ &= 2n\cdot\ln n+\ldots \\ &= 2n\cdot(\ln 2)\,\mathrm{ld}\,n+\ldots \\ &= 1.386n\,\mathrm{ld}\,n+\ldots \end{aligned}$$

und entsprechend für den Aufwand:

$$\frac{C(n)}{n} \cong 1.386\,\mathrm{ld}\,n\ .$$

3.4.2 Der optimale binäre Suchbaum

[Mehlhorn, Seite 144]

Diese Bäume sind gewichtet mit der Zugriffsverteilung, die die Häufigkeit (oder Wichtigkeit) der Elemente von S widerspiegelt.

Wir beschränken uns auf die Betrachtung von *Search(a, S)*.

Definition *Zugriffsverteilung*: Gegeben sei eine Menge $S = \{x_1 < x_2 < \ldots < x_n\}$ und zwei Sentinels x_0 und x_{n+1} mit $x_0 < x_i < x_{n+1}$ für $i = 1, \ldots, n$.

Die Zugriffsverteilung ist ein $(2n+1)$-Tupel $(\alpha_0, \beta_1, \alpha_1, \beta_2, \alpha_2, \ldots, \alpha_n, \beta_n)$ von Wahrscheinlichkeiten, für das gilt

- $\beta_i \geq 0$ ist die Wahrscheinlichkeit, daß eine Search(a, S)-Operation *erfolgreich* im Knoten $x_i = a$ endet, und
- $\alpha_j \geq 0$ ist die Wahrscheinlichkeit, daß eine Search(a, S)-Operation *erfolglos* mit $a \in (x_j, x_{j+1})$ endet.

Gilt für die Werte α_j und β_i

$$\sum_{i=1}^{n} \beta_i + \sum_{j=0}^{n} \alpha_j = 1$$

so spricht man von einer *normierten Verteilung*. Die Normierung ist zwar sinnvoll, wird aber im folgenden nicht benötigt.

Beispiel [Aigner, Seite 187]

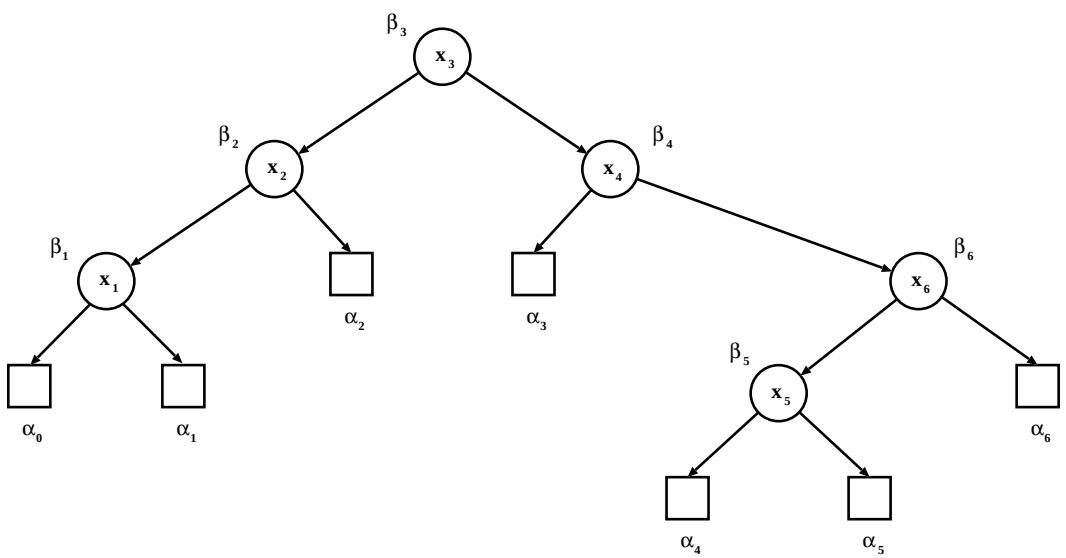


Abbildung 3.4: Baumdarstellung der Zugriffsverteilung

Komplexität von Search(a, S)

Wir betrachten nur den *Average Case*, also die mittlere Zahl von Vergleichen. Die Suche endet in einem inneren Knoten oder einem Blatt. Die Höhe eines Knotens oder eines

Blatts gibt direkt die Zahl der Vergleiche.
Sei ein Suchbaum gegeben mit:

b_i: Tiefe des Knotens x_i

a_j: Tiefe des Blattes (x_j, x_{j+1})

Sei P („Pfadlänge“) die mittlere Anzahl von Vergleichen, (auch gewichtete Pfadlänge):

$$P = \sum_{i=1}^{n} (\beta_i \cdot (1 + b_i)) + \sum_{j=0}^{n} (\alpha_j \cdot a_j)$$

Beachte: Ein Blatt erfordert einen Vergleich weniger als ein Knoten.

Bellman-Knuth-Algorithmus

[Ottmann, Seite 397]
Ziel: Konstruktion des optimalen Suchbaums, der bei gegebener Zugriffsverteilung die Pfadlänge minimiert.

Lösung: Dynamische Programmierung. Der Bellman-Knuth-Algorithmus fußt auf folgender Überlegung. Angenommen $S = \{x_i, \ldots, x_j\}$ sei ein optimaler Suchbaum und $x_k \in S$. Dann definiert der Teilbaum mit Wurzel x_k eine Zerlegung von S in zwei Teilmengen

$$\{x_i, \ldots, x_{k-1}\} \text{ und } \{x_k, \ldots, x_j\}$$

Aufgrund der Additivität der Pfadlänge muß auch jeder der beiden Teilbäume optimal bezüglich der (Teil-)Pfadlänge sein.

Um die dynamische Programmierung anzuwenden, definieren wir die Hilfsgröße $c(i, j)$ mit $i < j$:

$$c(i, j) := \text{ optimale Pfadlänge für den Teilbaum } \{x_i, \ldots, x_j\}$$

Dann kann man $\{x_i, \ldots, x_j\}$ aufgrund der Definition von $c(i, j)$ in zwei Teilbäume $\{x_i, \ldots, x_{k-1}\}$ und $\{x_k, \ldots, x_j\}$ zerlegen, die ihrerseits wiederum optimal sein müssen (Siehe Abbildung 3.5).

Daher gilt

$$\begin{aligned} c(i, i) &= 0 \qquad \text{für alle } i \\ c(i, j) &= w(i, j) + \min_{i<k\leq j}[c(i, k-1) + c(k, j)] \quad \text{für } i < j \end{aligned}$$

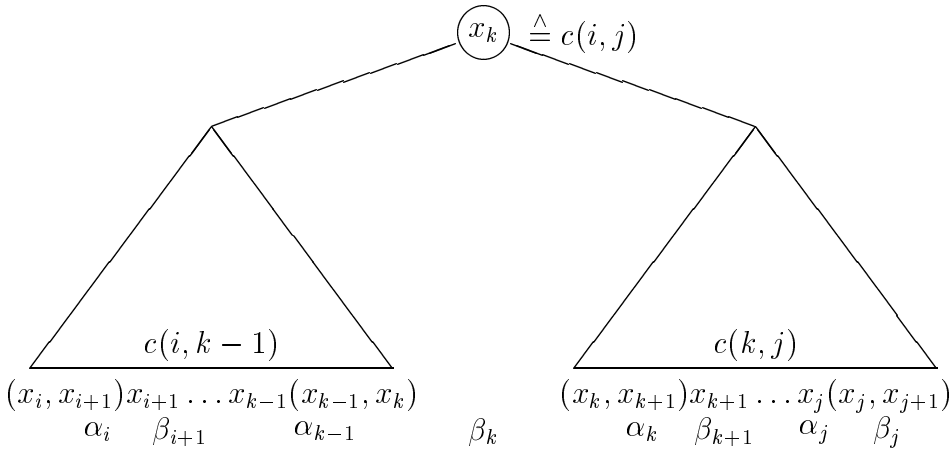


Abbildung 3.5: Schematische Darstellung der Teilbäume.

mit $w(i,j) = \sum_{l=i}^{j} \alpha_l + \sum_{l=i+1}^{j} \beta_l$.
Diese Gleichung ist iterativ, nicht rekursiv zu lösen. (DP-Gleichung; Gleichung der dynamischen Programmierung; Bellmansche Optimalitätsgleichung):
gesuchtes Ergebnis:

$$P = c(0,n)\ .$$

Programm-Entwurf zur iterativen Lösung der DP-Gleichung

Die Lösung ist ähnlich der für das Matrix-Produkt (s. Abschnitt 1.4.2). Wir benötigen zwei zweidimensionale Arrays.

c: **array** [0..n][0..n] **of real** speichert die Pfadlängen, und

r: **array** [0..n][0..n] **of** [0..n] dient der Rekonstruktion des optimalen Suchbaums.

Dann geht man wie folgt vor:

1. berechne $w(i,j)$ vorab;
   initialisiere $c(i,i) = 0$.

2. Berechnung eines optimalen Suchbaumes mit zunehmend längeren Pfaden der Länge l.

   for l=1 **to** n **do**
   for i=0 **to** $n-l$ **do**
   $j := i + l$
   $c(i,j) = w(i,j) + \min_{i<k\leq j}[c(i,k-1) + c(k,j)]$
   $r(i,j) = \arg\min_{i<k\leq j}[c(i,k-1) + c(k,j)]$

3. Rekonstruktion des optimalen Suchbaums mittels des Index-Arrays $r(i,j)$

Dem Entwurf entnehmen wir die Komplexität der Konstruktion eines optimalen Suchbaumes:

- die *Zeitkomplexität* $O(n^3)$ und
- die *Platzkomplexität* $O(n^2)$.

Hinweise:

- Vergleiche:
  - Die naive Implementierung mittels Rekursion hat *exponentielle* Komplexität.
  - Dynamischen Programmierung:
    - Speichern der Zwischenwerte in einer Tabelle;
    - Geschickter Aufbau der Tabelle.
- Verbesserung der Komplexität:
  - Matrix $r(i,j)$ (optimaler Index) ist monoton in jeder Spalte und Zeile für $0 \leq i < j \leq n$; d.h.:

$$r(i,j-1) \leq r(i,j) \leq r(i+1,j) \;\; \forall i,j \;\; \in 0,\ldots,n$$

  - Reduktion der Komplexität von $O(n^3)$ auf $O(n^2)$ (Beweis [Mehlhorn, Seite 151] mittels „Vierecksungleichung“ für $w(i,j)$).

An folgendem Beispiel wollen wir den Algorithmus veranschaulichen.

Beispiel [Aigner, Seite 187]

α_0	β_1	α_1	β_2	α_2	β_3	α_3	β_4	α_4
12	5	6	15	16	8	16	12	10

D.h. bei hundert Suchoperationen wird 12mal erfolglos nach einem Schlüssel kleiner x_1 gesucht, 5mal erfolgreich nach x_1, 6mal erfolglos nach einem Schlüssel zwischen x_1 und x_2 usw.

Gesucht ist jetzt nach einer Anordnung, bei der diese hundert Suchoperationen mit minimalen Vergleichskosten c durchgeführt werden können, bzw. die minimalen Kosten selbst für die hundert Suchoperationen bei optimaler Anordnung.

$l = 0$	$c(i,i) = 0$
$l = 1$	$c(0,1) = w(0,1) + c(0,0) + c(1,1) = 23$ $c(1,2) = w(1,2) + c(1,1) + c(2,2) = 37$ $c(2,3) = w(2,3) + c(2,2) + c(3,3) = 40$ $c(3,4) = w(3,4) + c(3,3) + c(4,4) = 38$
$l = 2$	$c(0,2) = w(0,2) + \min[c(0,0) + c(1,2), c(0,1) + c(2,2)]$ $= 54 + \min[37, 23] = 77$ $c(1,3) = w(1,3) + \min[c(1,1) + c(2,3), c(1,2) + c(3,3)]$ $= 61 + \min[40, 37] = 98$ $c(2,4) = w(2,4) + \min[c(2,2) + c(3,4), c(2,3) + c(4,4)]$ $= 62 + \min[38, 40] = 100$
$l = 3$	$c(0,3) = w(0,3) + \min[c(0,0) + c(1,3), c(0,1) + c(2,3), c(0,2) + c(3,3)]$ $= 78 + \min[98, 63, 77] = 141$ $c(1,4) = w(1,4) + \min[c(1,1) + c(2,4), c(1,2) + c(3,4), c(1,3) + c(4,4)]$ $= 83 + \min[100, 75, 98] = 158$
$l = 4$	$c(0,4) = w(0,4) + \min[c(0,0) + c(1,4), c(0,1) + c(2,4), c(0,2) + c(3,4), c(0,3) + c(4,4)]$ $= 100 + \min[158, 123, 115, 141] = 215$

r(i,j)

i \ j	1	2	3	4
0	1	2	2	3
1		2	3	3
2			3	3
3				4

w(i,j)

i \ j	1	2	3	4
0	23	54	78	100
1		37	61	83
2			40	62
3				38

c(i,j)

i \ j	0	1	2	3	4
0	0	23	77	141	**215**
1		0	37	98	158
2			0	40	100
3				0	38
4					0

3.5 Balancierte Bäume

Die natürlichen binären Suchbäume haben einen gravierenden Nachteil. Im Worst Case haben sie eine Komplexität $O(n)$. Dieser Fall tritt ein, wenn sie aus der *Balance* geraten, d.h. in Strukturen entarten, die einer linearen Liste ähnlich sind. Dies passiert entweder aufgrund der Reihenfolge der Elemente bei Insert(x, S)-Operationen oder auch wegen Delete(x, S)-Operationen.

Abhilfe: Balancierte Bäume, die *garantierte* Komplexitäten von $O(\log n)$ haben.

Man unterscheidet (mindestens) zwei Balance-Kriterien:

1. *Gewichtsbalancierte Bäume* auch BB(a)-Bäume (**B**ounded **B**alance) genannt: Die Anzahl der Blätter in den Unterbäumen wird möglichst gleich gehalten, wobei a die beschränkte Balance heißt (beschränkt den max. relativen Unterschied in der Zahl der Blätter zwischen den beiden Teilbäumen jedes Knotens) [Mehlhorn, Seite 176].

2. *Höhenbalancierte Bäume*: Bei diesen Bäumen ist das Balance-Kriterium die Höhe der Unterbäume, deren Unterschied möglichst gering gehalten wird. Je nach Ausprägung kann man hier viele Untertypen unterscheiden; die wichtigsten sind die

   - *AVL-Bäume* und die
   - *(a, b)-Bäume*, z.B. ein (2,4)-Baum.
     Wenn sie das Kriterium $b = 2a-1$ erfüllen, werden sie als *B-Baum* bezeichnet. Das 'B' steht hier für „balanced". Ein bekannter B-Baum ist der (2,3)-Baum [Mehlhorn, Seite 186].

3.5.1 AVL-Bäume

[Güting, Seite 120] Adelson-Velskij und Landis 1962:

- historisch erste Variante eines balancierten Baums;
- vielleicht nicht die effizienteste Methode, aber gut zur Darstellung des Prinzips geeignet.

Definition: Ein *AVL-Baum* ist ein binärer Suchbaum mit einer Struktur-Invarianten: Für jeden Knoten gilt, daß sich die Höhen seiner beiden Teilbäume höchstens um eins unterscheiden.

Um diese Invariante auch nach Update-Operationen (Delete, Insert) noch zu gewährleisten benötigt man *Rebalancierungs-Operationen*. Je nach vorangegangener Operation und Position des veränderten Elements können diese sehr aufwendig sein.

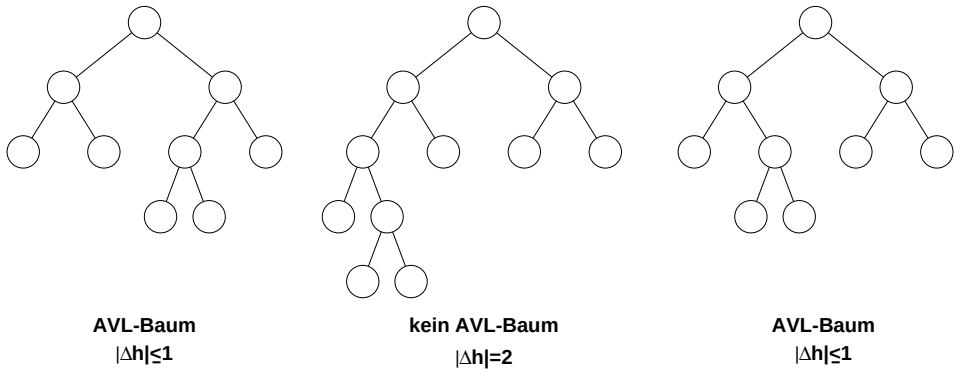


Abbildung 3.6: Beispiele für AVL-Bäume

Höhe eines AVL-Baumes

Wie hoch kann ein AVL-Baum bei vorgegebener Knotenzahl maximal, also im Worst Case sein? Oder: Wie stark kann ein AVL-Baum entarten?

Sei $N(h)$ die minimale Anzahl der Knoten in einem AVL-Baum der Höhe h.

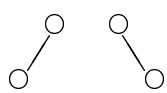
Ein AVL-Baum läßt sich folgendermaßen rekursiv definieren, und wir werden sehen, wie sehr diese Definition der der Fibonacci-Zahlen (siehe Abschnitt 1.2.5) ähnelt.

- $h = 0$: Der Baum besteht nur aus der Wurzel und es gilt

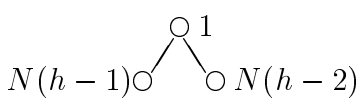
$$N(0) = 1$$

- $h = 1$: Da wir minimal gefüllte Bäume betrachten enthält die nächste Ebene nur einen Knoten, und es gilt

$$N(1) = 2$$



- $h \geq 2$: Methode: kombiniere 2 Bäume der Höhe $h-1$ und $h-2$ zu einem neuen, minimal gefüllten Baum mit neuer Wurzel:



Rekursionsgleichung für $N(h)$:

$$N(h) = 1 + N(h-1) + N(h-2)$$

wie Fibonacci (exakt nachrechnen), wobei $Fib(n)$ die Fibonacci-Zahlen sind:

$$N(h) = Fib(h+3) - 1$$

In Worten: Ein AVL-Baum der Höhe h hat mindestens $Fib(h+3)-1$ Knoten. Also gilt für die Höhe h eines AVL-Baums mit n Knoten:

$$N(h) \leq n < N(h+1)$$

Beachte: Damit ist h exakt festgelegt.

Um von der Anzahl der Knoten n auf die *maximale Höhe* des AVL-Baumes zu schließen, formen wir die Gleichung noch nach h um.

Mit $\Phi = \dfrac{1+\sqrt{5}}{2}$ und $\phi = \dfrac{1-\sqrt{5}}{2}$ gilt:

$$Fib(i) = \frac{1}{\sqrt{5}}\left(\Phi^i - \phi^i\right) \ .$$

Einsetzen:

$$\begin{aligned} Fib(h+3) &= \frac{1}{\sqrt{5}}\left[\Phi^{h+3} - \phi^{h+3}\right] \\ &\geq \frac{1}{\sqrt{5}}\Phi^{h+3} - \frac{1}{2} \end{aligned}$$

Und mit $Fib(h+3) \leq n+1$ folgt dann

$$\begin{aligned} \frac{1}{\sqrt{5}}\Phi^{h+3} &\leq n + \frac{3}{2} \\ \log_\Phi\left(\frac{1}{\sqrt{5}}\right) + h + 3 &\leq \log_\Phi\left[n + \frac{3}{2}\right] \\ h &\leq \log_\Phi n + \text{const} \\ &= (\ln 2/\ln\Phi)\,\mathrm{ld}\, n + \text{const} \\ &= 1.4404\,\mathrm{ld}\, n + \text{const} \end{aligned}$$

Also ist ein AVL-Baum maximal um 44% höher als ein vollständig ausgeglichener binärer Suchbaum.

Operationen auf AVL-Bäumen und ihre Komplexität

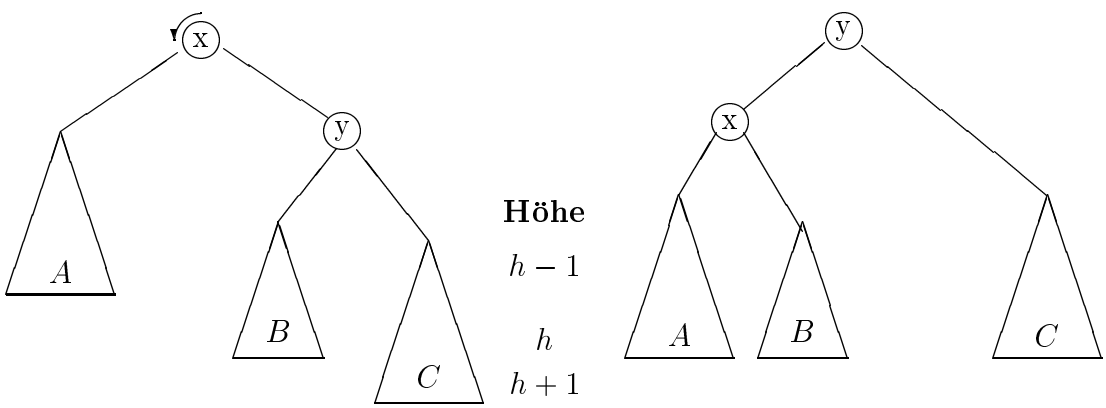
Nun wollen wir uns den Operationen auf AVL-Bäumen zuwenden. Dabei können zunächst die normalen Operationen wie gehabt ausgeführt werden, jedoch muß im Anschluß die Balance überprüft, und gegebenenfalls wiederhergestellt werden.

Die Balance-Überprüfung muß *rückwärts* auf dem ganzen Pfad vom veränderten Knoten bis zur Wurzel durchgeführt werden. Wir haben aber gesehen, daß dieser Pfad maximal eine Länge $\cong 1.44 \cdot \mathrm{ld}\, n$ haben kann. Die Überprüfung hat also eine Komplexität $O(\mathrm{ld}\, n)$.

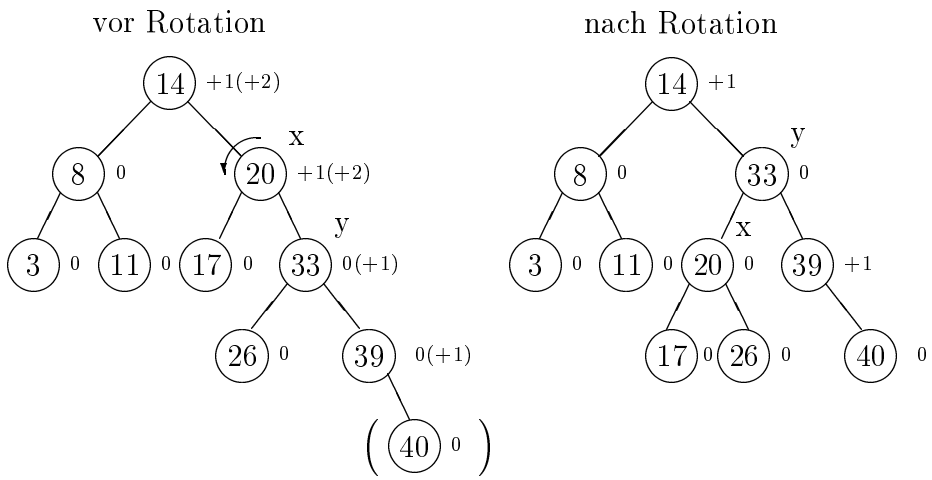
Ist die Balance an einer Stelle gestört, so kann sie in konstanter Zeit ($O(1)$) mittels einer *Rotation* oder *Doppel-Rotation* wieder hergestellt werden.

Ein AVL-Baum ist in einem Knoten aus der Balance, falls die beiden Teilbäume dieses Knotens einen Höhenunterschied größer eins aufweisen.

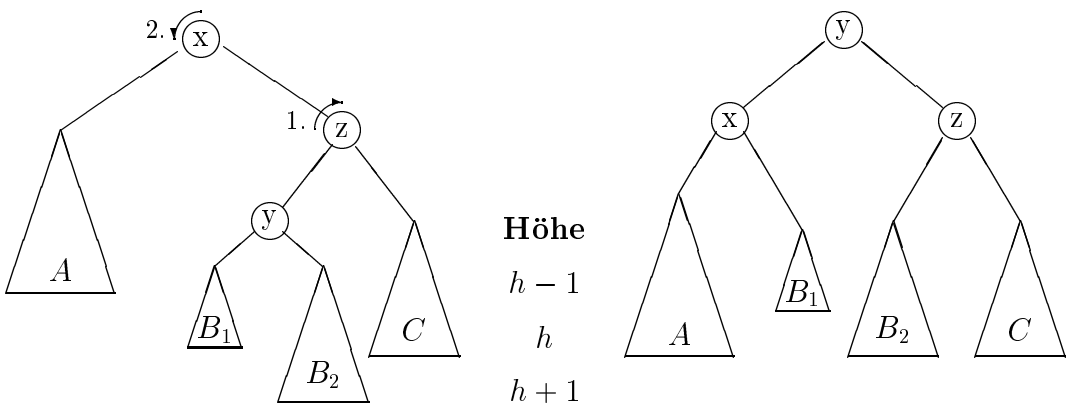
- **Rotation**: Ist ein Knoten betroffen (x), so betrachtet man nur den von ihm induzierten Teilbaum. Eine einfache Rotation muß durchgeführt werden, wenn der betroffene (also höhere) Teilbaum (C) *außen* liegt. Dann rotiert der betroffene Knoten zum kürzeren Teilbaum hinunter und übernimmt den inneren Sohn (B) des heraufrotierenden Knotens (y) als inneren Sohn.



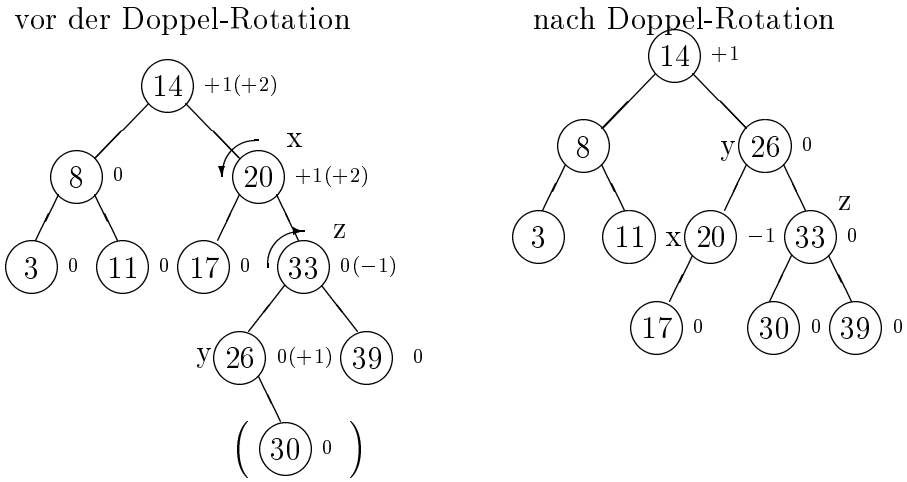
Zur Veranschaulichung ein Beispiel: In den folgenden AVL-Baum sei das Element **40** eingefügt worden, wodurch die AVL-Bedingung im Element **20** verletzt wurde.



- **Doppel-Rotation**: Eine Doppel-Rotation muß durchgeführt werden, falls der betroffene Teilbaum (mit Wurzel y) *innen* liegt, denn dann kann eine einfache Rotation das Ungleichgewicht auch nicht auflösen. Es wird zunächst eine Außen-Rotation im Vaterknoten der Wurzel des betroffenen Teilbaumes (z) durchgeführt, und anschließend eine Rotation in entgegengesetzter Richtung im Vaterknoten dieses Knotens (x).



Zur Veranschaulichung wieder ein Beispiel: In den obigen AVL-Baum sei diesmal das Element **30** (in den Teilbaum B_2) eingefügt worden, wodurch die AVL-Bedingung wieder im Element **20** (x) verletzt wurde, jedoch ist diesmal der innere Teilbaum betroffen.



Anmerkung: Es müssen insgesamt nur vier Pointer umdirigiert werden, um eine Rotation bzw. Doppel-Rotation durchzuführen. Dbaei bleibt die Suchbaum-Ordnung erhalten.

Für die Komplexitäten haben wir also folgendes gesehen.

- Zeitkomplexität $O(\operatorname{ld} n)$ für die drei Standard-Operationen, und
- Platzkomplexität $O(n)$.

3.5.2 (a,b)-Bäume

Prinzip eines (a,b)-Baums: „Bei einem (a,b)-Baum haben alle Blätter gleiche Tiefe; die Zahl der Söhne eines Knotens liegt zwischen a und b." Achtung: Definitionen variieren leicht!

Definition: [Mehlhorn]: Seien $a, b \in \mathbb{N}$ mit $a \leq 2$ und $2a - 1 \leq b$. $\rho(v)$ sei die Anzahl der Söhne des Knotens v. Ein Baum heißt (a,b)-*Baum*, wenn

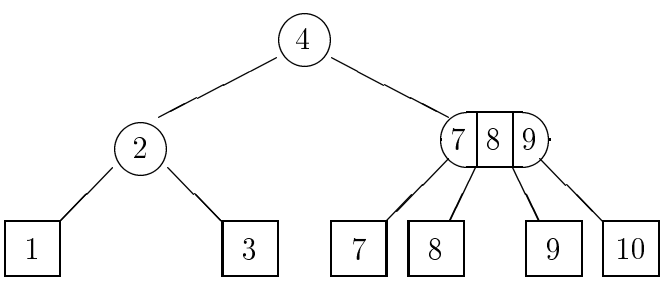
1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten v gilt $\rho(v) \leq b$,
3. für alle Knoten v *außer der Wurzel* gilt $a \leq \rho(v)$, und
4. die Wurzel mindestens zwei Söhne hat.

Gilt $b = 2a - 1$, dann spricht man von einem *B-Baum* der Ordnung b.

Bei der Umsetzung gibt es noch zwei verschiedene Spielarten, die *blattorientierte Speicherung*, bei der alle Elemente nur in Blättern eingetragen werden und die inneren Knoten nur als Wegweiser dienen, und die herkömmliche Speicherung, bei der auch in den Knoten selbst Elemente abgelegt werden.

Wir wollen hier nur die blattorientierte Speicherung betrachten.

Beispiel: (2,4)-Baum für $S = \{1, 3, 7, 8, 9, 10\} \subseteq \mathbb{N}$



Speicherung einer Menge als (a,b)-Baum

[Mehlhorn, Seiten 189ff.]

Sei $S = \{x_1 < \ldots < x_n\} \subseteq U$ eine geordnete Menge und T ein vorgegebener leerer (a,b)-Baum mit n Blättern. Dann wird S in T wie folgt gespeichert:

1. Die Elemente von S werden in den Blättern w von T entsprechend ihrer Ordnung von links nach rechts gespeichert

2. Jedem Knoten v werden $\rho(v) - 1$ Elemente $k_1(v) < k_2(v) < \ldots < k_{\rho(v)-1}(v) \in U$ so zugeordnet, daß für alle Blätter w im i-ten Unterbaum von v mit $1 < i < \rho(v)$ gilt
$$k_{i-1}(v) < Inhalt[w] \leq k_i(v)$$
Entsprechend gilt für die Randblätter w (also $i = 1$ bzw. $i = \rho(v)$) eines jeden Knotens:

   - Befindet sich w im ersten Unterbaum eines Knoten, d.h. $i = 1$, so gilt
   $$Inhalt[w] \leq k_1(v)$$
   - befindet sich w im letzten Unterbaum eines Knoten, d.h. $i = \rho(v)$, gilt
   $$k_{\rho(v)-1}(v) < Inhalt[w]$$

Für einen (a, b)-Baum mit n Blättern und Höhe h läßt sich folgendes ableiten:

$$\begin{aligned} 2a^{h-1} &\leq n \leq b^h \\ \log_b n &\leq h \leq 1 + \log_a \frac{n}{2} \end{aligned}$$

Speicherausnutzung: Wie man schon dem Beispiel entnehmen kann, muß jeder Knoten $(2b - 1)$ Speicherzellen besitzen, die sich unterteilen in b Zeiger zu den Söhnen und $b - 1$ Schlüssel (Elemente).

Die Speicherausnutzung beträgt bei einem (a,b)-Baum entsprechend mindestens $(2a - 1)/(2b - 1)$, z.B. für einen (2,4)-Baum 3/7.

Standard-Operationen:

- Search(y, S): Pfad von der Wurzel zum Blatt auswählen über die Elemente $k_1(v), \ldots, k_{\rho(v)-1}(v)$ an jedem Knoten v („Wegweiser“).
- Nach einer Insert(y, S)-Operation ist eventuell wiederholtes Spalten von Knoten notwendig, falls diese zu voll geworden sind.
- Nach einer Delete(y, S)-Operation muß man u.U. Knoten *verschmelzen* oder *stehlen* um den Baum zu rebalancieren.

Insgesamt liegt die Zeitkomplexität für alle drei Operationen sowohl für den Worst case als auch für den Average case bei $O(\log n)$ mit $n = |S|$.

Typische (a,b)-Bäume

Aus der Optimierung der Zugriffszeit ergeben sich grob zwei Varianten, die oft anzutreffen sind, und in Abhängigkeit der Speicherung der Daten auf Hintergrundmedien (Festplatte, Band, CD-ROM) oder im Hauptspeicher ihre Vorzüge haben.

- Bei der *internen Speicherung*: *(2,4)-Bäume* und *(2,3)-Bäume* .
- Bei der *externen Speicherung*, bei der man von einer langen Zugriffszeit ausgeht, verwendet man vorwiegend *B-Bäume* hoher Ordnung (zwischen 100 und 1000), die man mit der Zugriffszeit steigen läßt (lange Zugriffszeit $\rightarrow$ hohe Ordnung). Denn es gilt, daß die Anzahl der Zugriffe proportional zur Höhe des Baumes wächst.

  Beispiel: Sei $n = 10^6$ die Anzahl der Blätter, dann gilt für die Höhe und entsprechend die Anzahl der Zugriffe auf einen binären Suchbaum

  $$h = \operatorname{ld} n \approx 20$$

  und für einen B-Baum mit $a = 100$

  $$h \leq 1 + \lceil \log_a(n/2) \rceil = 1 + 3 = 4$$

  Das entspricht also einer Verringerung der Plattenzugriffe von 20 auf 4.

3.6 Priority Queue und Heap

Priority Queues sind Warteschlangen, bei denen die Elemente nach Priorität geordnet sind. Prinzipiell funktionieren sie zunächst wie einfache Queues: Elemente können nur am Kopf entfernt und am Schwanz eingefügt werden. Jedoch verlangt die Ordnung, daß am Kopf immer die Elemente mit höchster Priorität stehen. Also muß es eine Funktion geben, die es neu eingefügten Elementen hoher Priorität erlaubt, in der Warteschlange weiter nach vorne zu wandern.

Priority Queues findet man in Betriebssystemen beim *process scheduling*. Verschiedene Prozesse haben für das Betriebssystem unterschiedliche Prioritäten und sollen deshalb auch in der entsprechenden Reihenfolge abgearbeitet werden.

Beschreibung:

- nach Priorität geordnete Menge S
- *primäre Operationen:*
  - Insert(x, S),
  - DeleteMin(x, S): Entfernen des Elements mit der höchsten Priorität,

- weitere Operationen:
  - Initialize(S)
  - ReplaceMin(x, S)
  - und unter Umständen Delete(x, S), Search(x, S)

Datenstruktur:

- AVL-Baum oder
- Heap (wie bei HeapSort): „partiell geordneter, links-vollständiger Baum“: Partiell geordneter Binärbaum, bei dem in jedem Knoten das Minimum (Maximum) des jeweiligen Teilbaumes steht.

Zwei Standardoperationen (vgl. HeapSort):

- **Insert**(y, S): Fügt Element y an der letzten Stelle im Heap-Array ein und bewegt es aufwärts (überholen), bis es die seiner Priorität entsprechende Position erreicht hat (Prozedur *Upheap*).
- **DeleteMin**(S): Entfernt das Element in der Wurzel und setzt dort das letzte Element des Arrays ein, um es dann mittels *DownHeap* (wie bei HeapSort) an die richtige Position zurückzuführen.

Von HeapSort wissen wir, daß die **Komplexität** für beide Operationen bei maximal $2 \cdot \operatorname{ld} n$ Vergleichen liegt.

Übung: Warum wird die Heap-Datenstruktur nicht zur Mengendarstelllung mit den drei Standard-Operationen *Search*, *Insert* und *Delete* verwendet?

Antwort: *Search* ist problematisch.

4 Graphen

4.1 Motivation: Wozu braucht man Graphen?

Viele reale Fragestellungen lassen sich durch Graphen darstellen:

- Beziehungen zwischen Personen:
  - Person A kennt Person B
  - Person A spielt gegen Person B
- Verbindungen zwischen Punkten:
  - Straßen
  - Eisenbahn
  - ...
- Telefonnetz
- Darstellung elektronischer Schaltungen

Bezogen auf einen Graphen ergeben sich spezielle Aufgaben und Fragen:

- Existiert eine Verbindung zwischen A und B? Existiert eine zweite Verbindung, falls die erste blockiert ist?
- Wie lautet die kürzeste Verbindung von A nach B?
- Wie sieht ein minimaler Spannbaum zu einem Graphen aus?
- Wie plane ich eine optimale Rundreise? (*Traveling Salesman Problem*)

Begriffe:

- Knoten ("Objekte")
- Kanten
- gerichtet/ ungerichtet
- gewichtetet/ ungewichtet

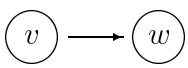
4.2 Definitionen und Graph-Darstellungen

Definition: Ein *gerichteter Graph* (engl. digraph = „directed graph“) ist ein Paar $G = (V, E)$ mit einer endlichen, nichtleeren Menge V, deren Elemente Knoten (nodes, vertices) heißen, und einer Menge $E \subseteq V \times V$, deren Elemente Kanten (edges, arcs) heißen.

Bemerkungen:

- $|V|$ = Knotenanzahl
- $|E| \leq |V|^2$ = Kantenanzahl
  (wenn man alle Knoten-Paare zuläßt, machmal auch: $|E| = \frac{|V|(|V|-1)}{2}$)
- Meist werden die Knoten durchnumeriert: $i = 1, 2, \ldots, |V|$

Graphische Darstellung einer Kante e von v nach w:



Begriffe:

- v ist *Vorgänger* von w
- w ist *Nachfolger* von v
- v und w sind *Nachbarn* bzw. *adjazent*

Weitere Definitionen:

- *Grad* eines Knotens := Anzahl der ein- und ausgehenden Kanten
- Ein *Pfad* ist eine Folge von Knoten $v_1, \ldots, v_n$ mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$, also eine Folge von “zusammenhängenden” Kanten.

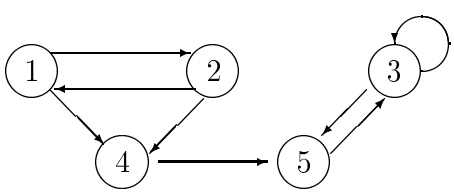


Abbildung 4.1: Beispiel-Graph G_1 (aus Güting, S.145-146). Das Beispiel wird später für den Graph-Durchlauf benötigt werden.

- *Länge eines Pfades* := Anzahl der Kanten auf dem Pfad
- Ein Pfad heißt *einfach*, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein *Zyklus* ist ein Pfad mit $v_1 = v_n$ und Länge= $n - 1 \geq 2$.
- Ein *Teilgraph* eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$.

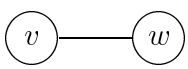
Man kann Markierungen oder Beschriftungen für Kanten und Knoten einführen. Wichtige Kostenfunktionen oder -werte für Kanten sind:

- $c[v, w]$ oder $cost(v, w)$
- Bedeutung: Reisezeit oder -kosten (etwa die Entfernung zwischen v und w)

Definition: Ein *ungerichteter Graph* ist ein gerichteter Graph, in dem die Relation E symmetrisch ist:

$$(v, w) \in E \Rightarrow (w, v) \in E$$

Graphische Darstellung (ohne Pfeil):



Bemerkung: Die eingeführten Begriffe (Grad eines Knoten, Pfad, ...) sind analog zu denen für gerichtete Graphen. Bisweilen sind Modifikationen erforderlich, z.B. muß ein Zyklus hier mindestens drei Knoten haben.

4.2.1 Graph-Darstellungen

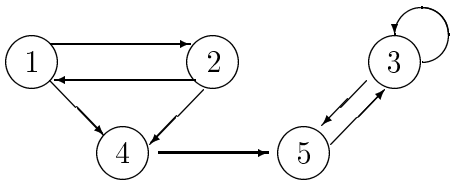
Man kann je nach Zielsetzung den Graphen knoten- oder kanten-orientiert abspeichern, wobei erstere Darstellungsform weitaus gebräuchlicher ist und in vielen verschiedenen Variationen existiert. Man identifiziert zur Vereinfachung der Adressierung die Knoten mit natürlichen Zahlen.

Sei daher $V = \{1, 2, \ldots, |V|\}$.

- Bei der *knoten-orientierten Darstellung* existieren folgende Darstellungsformen:
  - Die **Adjazenzmatrix** A eine boolesche Matrix mit:

    $$A_{ij} = \begin{cases} true & \text{falls } (i, j) \in E \\ false & \text{andernfalls} \end{cases}$$

    Eine solche Matrix A_{ij} läßt sich als Array $A[i, j]$ darstellen. Damit folgt für den Beispiel-Graph G_1 mit der Konvention $true = 1$, $false = 0$:



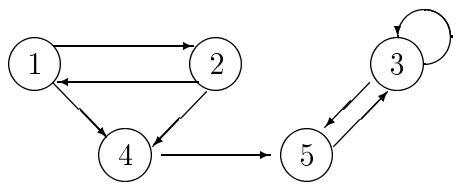
von \ nach	1	2	3	4	5
1		1		1	
2	1			1	
3			1		1
4					1
5			1		

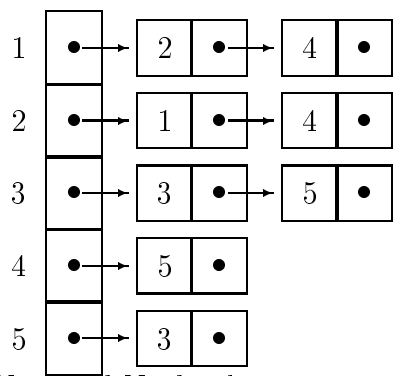
Vor- und Nachteile:

+ Entscheidung, ob $(i, j) \in E$ in Zeit $O(1)$
+ erweiterbar für Kantenmarkierungen und Kosten(-Funktionen)
– Platzbedarf stets $O(|V|^2)$, ineffizient falls $|E| \ll |V|^2$
– Initialisierung benötigt Zeit $O(|V|^2)$

– Im Fall der **Adjazenzliste** wird für jeden Knoten eine Liste der Nachbarknoten angelegt (Vorgänger: "invertierte" Adjazenzliste).

Es ergibt sich für den Graphen G_1:





Vor- und Nachteile:

+ geringer Platzbedarf von $O(|V| + |E|)$
– Entscheidung, ob $(i, j) \in E$ in Zeit $O(|E|/|V|)$ im Average Case

- *kanten-orientierte Darstellung*: eigener Index e für Kanten erforderlich.
  Prinzip: Adressierung jeder Kante mittels $e \in E$.
  Dabei ist für jede Kante gespeichert:

  – Vorgänger-Knoten
  – Nachfolger-Knoten
  – ggfs.: Markierung/ Kosten

  Methode: meist statische Darstellung, seltener dynamische Listen.

Man unterscheidet verschiedene Typen von Graphen:

- *vollständiger* (complete) Graph: $|E| = |V|^2$ oder $|E| = \frac{|V|(|V|-1)}{2}$
- *dichter* (dense) Graph: $|E| \simeq |V|^2$
- *dünner* (sparse) Graph: $|E| \ll |V|^2$

Die Dichte eines Graphen ist das Hauptkriterium bei der Auswahl einer geeigneten Darstellungsform (mit dem Ziel einer optimalen Speicherplatz- und Zugriffs-Effizienz).

4.2.2 Programme zu den Darstellungen

Zunächst widmen wir uns der Implementation der Adjazenzmatrix-Darstellung für einen ungewichteten, gerichteten Graphen. Dabei sei V die Anzahl der Knoten und E die der Kanten.

```
PROGRAM adjmatrix (input, output);
  CONST maxV = 50;
  VAR j, x, y, V, E : INTEGER;
                  a :  ARRAY[1..maxV,1..maxV] OF BOOLEAN;
BEGIN
  readln (V, E);
  FOR x:= 1 TO V DO
    FOR y := 1 TO V DO a[x,y] := FALSE;
  FOR x:= 1 TO V DO a[x,x] := TRUE; (* spezielle Konvention *)
  FOR j := 1 TO E DO
    BEGIN
      readln(v1, v2); (* Kante wird eingelesen *)
      x := index(v1); y := index(v2); (* berechnet Index fuer jeden Knoten *)
      a[x,y] := TRUE;
      a[y,x] := TRUE; (* bei ungerichtetem Graphen *)
    END
END;
```

Anmerkung: Die Funktion `index(vertex)` ist immer dann nötig, wenn die Knoten nicht einfach durchnumeriert werden (andernfalls gilt `index(`v`)=`v).

Implementierung der Adjazenzliste:

```
PROGRAM adjlist (input, output);
  CONST maxV = 1000;
  TYPE  link = REF node;
        node = RECORD
                 v:  INTEGER;
                 next:  link
               END;
  VAR   j, x, y, V, E : INTEGER;
        t, z :  link;
        adj :  ARRAY[1..maxV] OF link; (* Listenbeginn fuer jeden Knoten *)
BEGIN
  readln (V,E);
  z := NEW(link); z↑.next := z;
  FOR j := 1 TO V DO adj[j] := z;
  FOR j:= 1 TO E DO
    BEGIN
      readln(v1,v2);
      x := index(v1); y:= index(v2);
      t := NEW(link); t↑.v := x; t↑.next := adj[y]; adj[y] := t; (* Symmetrie *)
      t := NEW(link); t↑.v := y; t↑.next := adj[x]; adj[x] := t; (* Start *)
    END
END;
```

4.3 Graph-Durchlauf

Definition: Die *Expansion* $X_G(v)$ eines Graphen G in einem Knoten v ist ein Baum, der wie folgt definiert ist:

1. Falls v keine Nachfolger hat, ist $X_G(v)$ nur der Knoten v.
2. Falls $v_1, ..., v_k$ die Nachfolger von v sind, ist $X_G(v)$ der Baum mit der Wurzel v und den Teilbäumen $X_G(v_1), ..., X_G(v_k)$.

Beachte:

- Die Knoten des Graphen kommen in der Regel mehrfach im Baum vor.

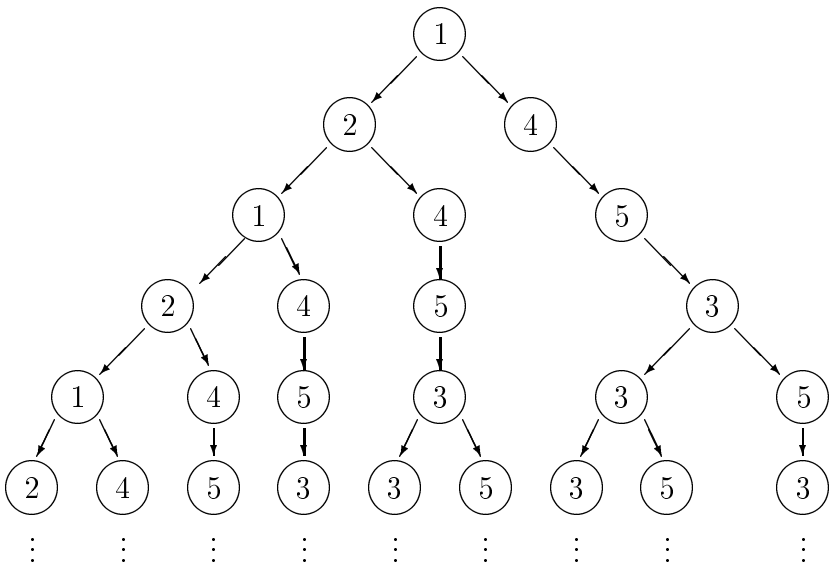


Abbildung 4.2: Beispiel Graph G_1q (s. Absch. 4.1) in seiner Expansion $X_{G_1}(v)$ mit $v = 1$.

- Ein Baum ist unendlich, falls der Graph Zyklen hat.
- Der Baum $X_G(v)$ stellt die Menge aller Pfade dar, die von v ausgehen.

4.3.1 Programm für den Graph-Durchlauf

Gleiches Konzept wie Baum-Durchlauf (vgl. Abschn. /refsec:tree-traversal)

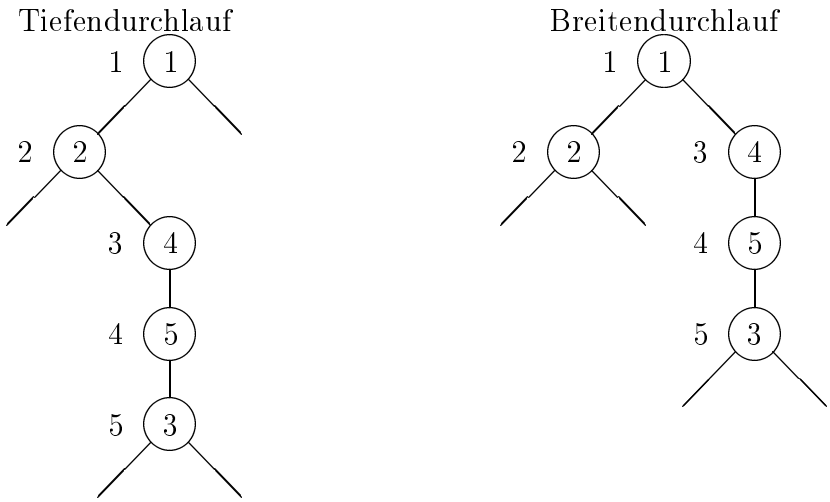
- Breitendurchlauf: preorder traversal (breadth first) mittels stack (oder rekursiv)
- Tiefendurchlauf: level order traversal (depth first) mittels queue

Wichtige Modifikation:

a) Das Markieren der schon besuchten Knoten ist notwendig, weil Graph-Knoten im Baum mehrfach vorkommen können.

b) Abbruch des Durchlaufs bei schon besuchten Knoten.

Beispiel G_1 mit Startknoten: $v = 1$



Programm-Ansatz:

1. Initialisierung alle Knoten als „not visited“
2. Abarbeiten der Knoten
   if („node not visited”) **then**
   - bearbeite
   - markiere: „visited“

Interpretation: Graph-Durchlauf Während des Graph-Durchlaufs werden die Graph-Knoten in 3 Klassen eingeteilt:

- Ungesehene Knoten (unseen vertices):
  Knoten, die noch nicht erreicht worden sind:
  $$val[v] = 0$$
- Baum-Knoten (tree vertices):
  Knoten, die schon besucht und abgearbeitet sind. Diese Knoten ergeben den „Suchbaum”:
  $$val[v] = id > 0$$
- Rand-Knoten (fringe vertices):
  Knoten, die über eine Kante mit einem Baum-Knoten verbunden sind:
  $$val[v] = -1$$

Die Verallgemeinerung für ein beliebiges Auswahlkriterium führt zu folgendem Programm-Schema:

- Start: Markiere den Startknoten als Rand-Knoten und alle anderen Knoten als ungesehene Knoten.
- Schleife: **repeat**
  - Wähle einen Rand-Knoten x mittels eines Auswahlkriteriums (depth first, breadth first, priority first).
  - Markiere x als Baum-Knoten und bearbeite x.
  - Markiere alle ungesehenen Nachbar-Knoten von x als Rand-Knoten.

  ... **until** (alle Knoten abgearbeitet)

Setzt man dieses Vorgehen in ein Programm um, so erhält man für den Graph-Durchlauf folgendes Programm-Gerüst (angelehnt an Sedgewick 88).

```
PROCEDURE Graph-Traversal;
  VAR id, k :  INTEGER;
      val :  ARRAY[1..maxV] OF INTEGER;
BEGIN
  id := 0;
  stackinit/ queueinit; (* select one *)
  FOR k := 1 TO V DO val[k] := 0; (* not yet visited *)
  FOR k := 1 TO V DO
    IF val[k] = 0 THEN visit(k)
END;
```

Die Unterschiede der verschiedenen Durchlauf- und Abspeicherungs-Arten offenbaren sich in der Prozedur *visit(k)*.

Rekursiver Tiefendurchlauf mit Adjazenzliste

```
PROCEDURE visit (k :  INTEGER);
  VAR t :  link;
BEGIN
  id := id + 1; val[k] := id;
  t := adj[k];
  WHILE t ≠ z DO BEGIN
    IF val[t↑.v] = 0 THEN visit(t↑.v); (* excluded > 0 *)
    t := t↑.next
  END
END;
```

Rekursiver Tiefendurchlauf mit Adjazenzmatrix

```
PROCEDURE visit (k :  INTEGER);
  VAR t :  INTEGER;
BEGIN
  id := id + 1; val[k] := id;
  FOR t := 1 TO V DO
    IF a[k,t]
      THEN IF val[t] = 0 THEN visit(t);
END;
```

Iterativer Tiefendurchlauf mit Stack und Adjazenzliste

```
PROCEDURE visit (k :  INTEGER);
  VAR t :  link;
BEGIN
  push (k);
  repeat
    k := pop; id := id + 1; val[k] := id;
    t := adj[k];
    WHILE t ≠ z DO
      BEGIN
        IF val[t↑.v] = 0 (* excluded -1 and >0 *)
          THEN BEGIN
            push(t↑.v); val[t↑.v] := -1;
          END;
        t := t↑.next
      END
  UNTIL stackempty
END;
```

Komplexität der Tiefensuche:

- Zeit bei Verwendung einer Adjazenzliste: $O(|E| + |V|)$.
- Zeit bei Verwendung einer Adjazenzmatrix: $O(|V|^2)$.

Iterativer Breitendurchlauf mit Queue und Adjazenzliste

```
PROCEDURE visit (k :  INTEGER);
  VAR t :  link;
BEGIN
  put(k);
  REPEAT
    k := get; id := id + 1; val[k] := id;
    t := adj[k];
    WHILE t ≠ z DO
      BEGIN
        IF val[t↑.v] = 0
          THEN BEGIN
            put(t↑.v); val[t↑.v] := -1;
          END;
        t := t↑.next
      END
  UNTIL queueempty
END;
```

Hinweise zur nicht-rekursiven Implementation:
Knoten, die schon erreicht („touched") aber noch nicht abgearbeitet sind, werden auf dem Stack gehalten. [Rekursiv: lokale Variable t]

Implementierung mit Hilfe des Arrays *val* (vgl. Absch. 4.3.1)

$$val[v] = \begin{cases} 0 & \text{Knoten } v \text{ noch nicht erreicht} \\ -1 & \text{Knoten } v \text{ auf dem Stack/ Queue} \\ id > 0 & \text{Knoten } v \text{ abgearbeitet} \end{cases}$$

Beachte: Die Reihenfolge ist nicht exakt dieselbe wie im rekursiven Programm.

Analog: Queue für die Breitensuche

4.4 Kürzeste Wege

Zu den wichtigen Verfahren auf Graphen gehören diejenigen zum Auffinden kürzester Wege (von A nach B, von einem Knoten zu allen anderen oder auch von allen Knoten zu allen anderen).

4.4.1 Dijkstra-Algorithmus (Single-Source Best Path)

Der Dijkstra-Algorithmus berechnet den kürzesten Weg von einem Startknoten zu jedem anderen Knoten. Dijkstra publizierte die 3-seitige (!) Originalarbeit im Jahre 1959.

Gegeben sei ein gerichteter Graph G mit der Bewertungsfunktion

$$c[v,w] \quad \begin{cases} \geq 0 & \text{falls eine Kante von } v \text{ nach } w \text{ existiert,} \\ = \infty & \text{falls keine Kante von } v \text{ nach } w \text{ existiert,} \\ = 0 & \text{für } w = v \text{ (spezielle Konvention, falls nötig.)} \end{cases}$$

Aufgabe: Der Start-Knoten v_0 und Endknoten w seien vorgegeben. Finde den Pfad mit minimalen Gesamtkosten von v_0 nach w (wobei sich die Gesamtkosten aus den Kosten der Kanten ergeben), d.h. den Pfad $\{v_0, v_1, ..., v_n\}$ mit Startknoten v_0 und Endknoten v_n mit:

$$\min_{n;v_0,v_1,...,v_n} \sum_{i=1}^{n} c[v_{i-1}, v_i] \text{ mit } v_n = w$$

Es wird sich zeigen, daß die folgende Verallgemeinerung nicht mehr Aufwand erfordert:

Sei der Start-Knoten v vorgegeben: Finde zu jedem Knoten w den Pfad von v_0 mit minimalen Gesamtkosten.

Erläuterungen: Die besten Pfade werden systematisch aufgebaut, indem wir bereits bekannte beste Pfade Kante um neue Kanten wachsen lassen:

- Durch Hinzunahme einer Kante können die Gesamtkosten nur wachsen.
- Falls die besten Pfade von v_0 zu allen anderen Knoten ungleich w höhere Kosten haben, ist der beste Pfad von v_0 nach w gefunden.
- Der beste Pfad kann keinen Zyklus haben, falls die Kosten des Zyklus größer Null sind. Sind diese Kosten gleich Null, so gibt es einen besten Pfad ohne Zyklus mit denselben Kosten.
- Der beste Pfad hat maximal $(|V| - 1)$ Kanten.

Arbeitsweise des Algorithmus (Aho 83, S.205)

Der Dijkstra-Algorithmus baut die gesuchten kürzeste Wege sukzessive aus schon bekannten kürzesten Pfaden auf. Das entspricht etwa einer äquidistanten Welle um den Startpunkt, für die während der Ausbreitung jeder Weg zu einem gerade erreichten Knoten vermerkt wird (vgl. Abb. 4.3).

Sei V die Menge der Knoten, und es sei der Startknoten v_0 gegeben.

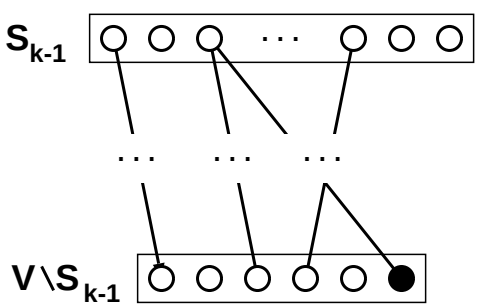
Definition:

S_k	:=	Die Menge der Knoten v mit den k besten Pfaden $v_0 \to v$
$D(S_k, v)$	:=	Die Kosten des besten Pfades von einem Knoten in S_k nach v ("der Pfad liegt in S_k").

Schema des Dijkstra-Algorithmus:

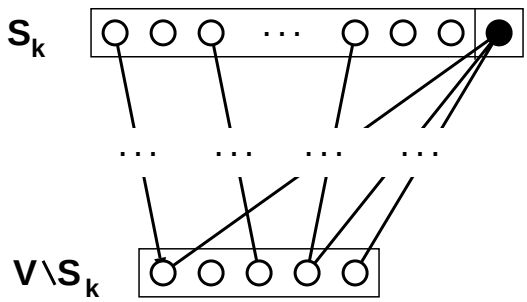
```
(1) INITIALIZATION
(2)    S_0 := {v_0};
(3)    D(S_0, v) := c[v_0, v]  ∀ v ∈ V \ S_0
(4) FOR each rank k = 1 TO |V| − 1 DO BEGIN
(5)    w_k := arg min{D(S_{k−1}, w) : w ∈ V \ S_{k−1}};
(6)    S_k := S_{k−1} ∪ {w_k};
(7)    FOR each veretx v ∈ V \ S_k DO
(8)       D(S_k, v) := min{D(S_{k−1}, v), D(S_{k−1}, w_k) + c[w_k, v]}
(9) END;
```

Veranschaulichung des Dijkstra-Algorithmus:



$$w_k = \arg\min_{w \in V \setminus S_{k-1}} D(S_{k-1}, w)$$

$$S_k := S_{k-1} \cup \{w_k\}$$



$$D(S_k, v) = \min\{D(S_{k-1}, v), D(S_{k-1}, w_k) + c[w_k, v]\}$$

Durch den Dijkstra-Algorithmus wird für jede Pfadlänge $k = 1, ..., |V|$ die Knotenmenge V in drei disjunkte Teilmengen zerlegt:

a) S_k: abgearbeitete Knoten, d.h. der global beste Pfad zu diesen Knoten ist schon gefunden.

b) $\{v \in V : D(S_k, v) < \infty\} \setminus S_k$: Randknoten, die von S_k aus erreicht worden sind.

c) $\{v \in V : D(S_k, v) = \infty\}$: Unerreichte Knoten, zu denen noch kein Pfad existiert.

Eine Verallgemeinerung dieses Konzeptes führt zu dem sogenannten A^*-Algorithmus und verwandten heuristischen Suchverfahren, wie sie in der Künstlichen Intelligenz verwendet werden.

Der Index k wird letztlich nicht benötigt, so daß die abgearbeiteten Knoten w in *einer* Menge S (ohne Index) verwaltet werden können:

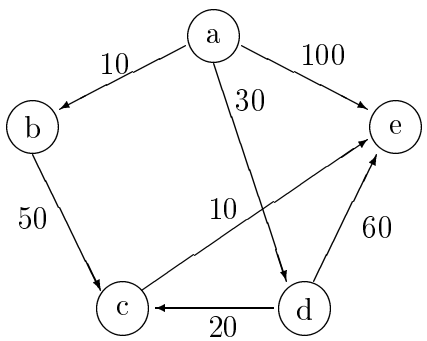
Seien $V := \{1, ..., n\}$ und $S :=$ { Knoten, die bereits abgearbeitet sind.}

```
    PROCEDURE Dijkstra;
        { Dijkstra computes the cost of the shortest paths from vertex 1
        to every vertex of a directed graph}
    BEGIN
(1)     S := {1};
(2)     FOR i := 2 TO n DO
(3)         D[i] := c[1,i]; { initialize D}
(4)     FOR i := 1 TO n-1 DO BEGIN
(5)         choose a vertex w in V \ S such that D[w] is a minimum;
(6)         add w to S;
(7)         FOR each vertex v in V \ S DO
(8)             D[v] := min(D[v], D[w] + c[w,v])
        END
    END; { Dijkstra }
```

Anmerkungen zum Dijkstra-Algorithmus:

- Der Dijkstra-Algorithmus liefert keine Näherung, sondern garantiert Optimalität.
- Bei negativen Bewertungen ist er zunächst nicht anwendbar.
- Existieren Zyklen mit negativer Bewertung, so gibt es keinen eindeutigen Pfad mit minimalen Kosten mehr.
- Frage bei einem vollständigem Graphen:
  Ist eine niedrigere Komplexität vorstellbar, wobei jede Kante weniger als einmal betrachtet wird?

Beispiel: [Aho 83, S.205]



k	w_k	S_k	$D(S_k, b)$	$D(S_k, c)$	$D(S_k, d)$	$D(S_k, e)$
0	-	$\{a\}$	10	∞	30	100
1	b	$\{a, b\}$	—	60	30	100
2	d	$\{a, b, d\}$		50	—	90
3	c	$\{a, b, d, c\}$		—		60
4	e	$\{a, b, d, c, e\}$				—

Bester Pfad von 'a' nach 'e': $a \to d \to c \to e$ mit Kosten 60.

Übung: Erstelle das vollständige Programm und rekonstruiere den optimalen Pfad (Vorgängerknoten 'merken')

Basis des Dijkstra-Algorithmus

Optimalitätsprinzip ([Ottmann S.620])
Für jeden besten Pfad $p = (v_0, v_1, ..., v_k)$ von v_0 nach v_k ist jeder Teilpfad $p' = (v_i, ..., v_j)$, $0 \leq i < j \leq k$, ein bester Pfad von v_i nach v_j.

Beachte: Die Eindeutigkeit muß nicht gegeben sein und wird auch nicht behauptet.

Beweis des Optimalitätsprinzips (Indirekter Beweis):
Wäre dies nicht so, d.h. gäbe es einen kürzeren Weg p'' von v_i nach v_j, so könnte in p der Teilpfad p' durch p'' ersetzt werden, und der so konstruierte Pfad wäre besser als p. Dies ist aber ein Widerspruch zur Annahme, daß p bester Pfad von v_0 nach v_k ist.

Beachte: Die Additivität der Kostenfunktion ist hier wie immer wesentlich.

Komplexitätsanalyse für Varianten des Dijkstra-Algorithmus:

1. mit Adjazenz-Matrix (wie beschrieben): Zeitkomplexität $O(|V|^2)$
2. mit Priority Queue (Heap):
   $|E| \ll |V| \Rightarrow$ Adjazenz-Liste (+ Kosten) und Priority Queue für $V \setminus S$

   Zeilen (7) und (8):
   Durchlaufen der Adjazenz-Liste und Updaten der Abstände in der Priority Queue
   Insgesamt: $|E|$ updates in $O(\log |V|)$
   Zeilen (1-3) (4):
   Jeweils $O(|V|)$
   Zeilen (5) und (6):

   - implementieren „DeleteMin $(y, V \setminus S)$"
   - $(|V| - 1)$ Iterationen mit je $O(\log |V|)$ Komplexität

   $\Rightarrow O([|E| + |V|] \cdot \log |V|)$
   da meist $|V| \leq |E|$ folgt: $O(|E| \log |V|)$
3. Dijkstra mit „Fibonacci-Heap" (ohne Erläuterung): $O(|E| + |V| \cdot \log |V|)$

4.4.2 Floyd-Algorithmus (All Pairs Best Path)

Im Jahre 1962 stellte Floyd einen Algorithmus vor, der alle kürzesten Pfade zwischen zwei beliebigen Knoten eines Graphen berechnet.

Prinzipiell machbar:
Dijkstra-Algorithmus für jeden Knoten anwenden, ergibt Zeitkomplexität $O(|V| \cdot |V|^2) = O(|V|^3)$.

Andere Möglichkeit: Floyd-Algorithmus (basierend auf *dynamischer Programmierung*).

Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{1, 2, ..., n\}$, $n = |V|$ und nicht-negativer Bewertung

$$c[v, w] \quad \begin{cases} \geq 0 & \text{, falls Kante } v \to w \text{ existiert} \\ = \infty & \text{, sonst} \end{cases}$$

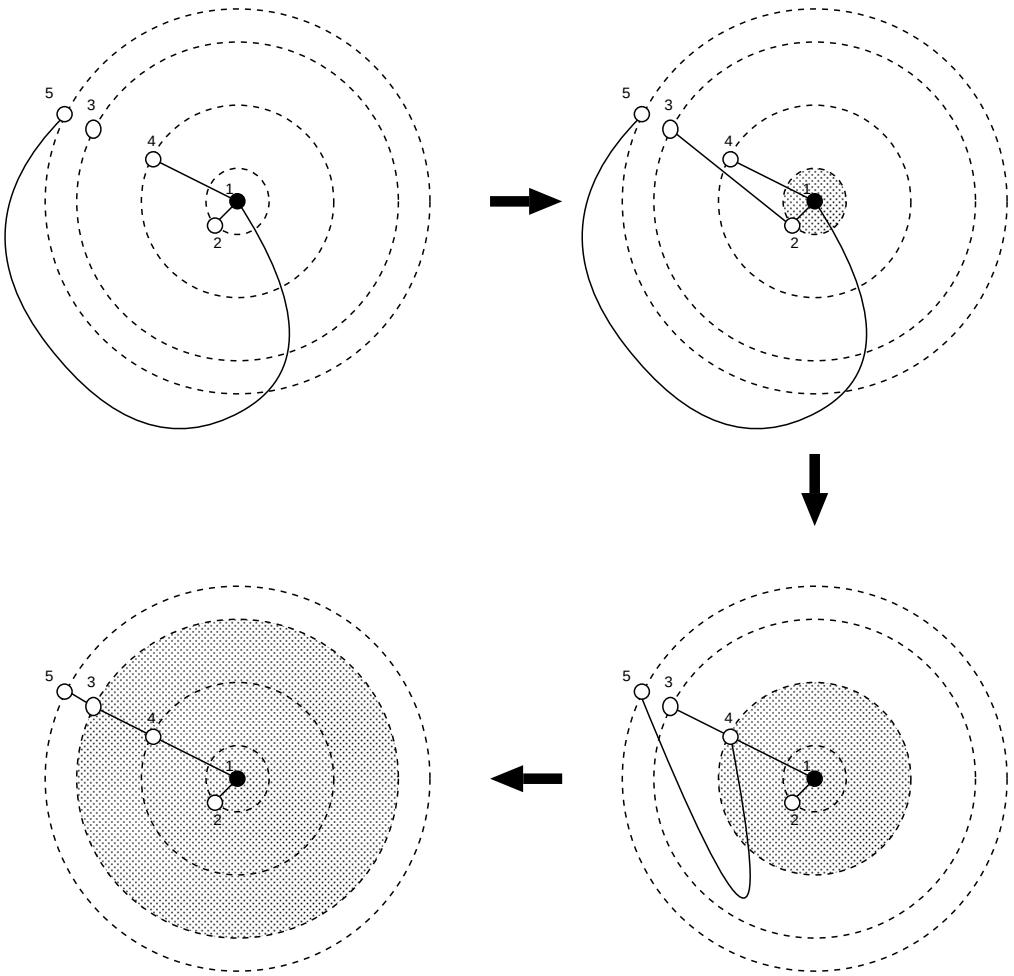


Abbildung 4.3: Visualisierung zum Dijksta-Algorithmus: Die Ausbreitung der „aquidistanten Welle“ um den Startpunkt.

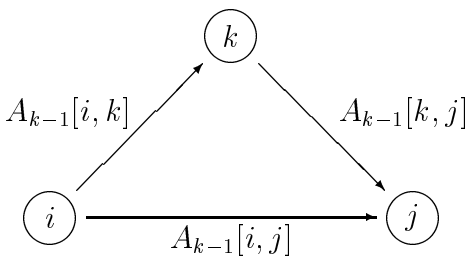
Wir definieren zunächst eine $|V| \times |V|$-Matrix A_k mit den Elementen:

$A_k[i,j] :=$ minimale Kosten, um über Knoten aus {1,...,k} vom Knoten i zum Knoten j zu gelangen

Aufgrund der Definition gilt folgende Rekursionsgleichung (Gleichung der DP):

$$A_k[i,j] := \min\{A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]\}$$

Visualisierung der DP-Gleichung:



Vereinfachung:
Es ändert sich kein Element $A_k[i,j]$, wenn i oder j gleich k sind:

- $A_k[i,k] = A_{k-1}[i,k]$
- $A_k[k,j] = A_{k-1}[k,j]$

Man benötigt also keine Matrizen $A_k[i,j]$, $k = 1, ..., n$, sondern nur eine einzige Matrix $A[i,j]$.

Initialisierung: $A[i,j] = c[i,j] \quad \forall i,j \in \{1,\ldots,|V|\}$

Programm zum Floyd-Algorithmus

Die Matrix C repräsentiert den Graphen in der Adjazenz-Matrix-Darstellung. Die Matrix A ist die oben definierte, und die Matrix P speichert die jeweiligen Vorgänger (**P**redecessor) eines jeden Knotens auf dem besten Pfad, sie dient also der Rekonstruktion dieses Pfades. [Aigner]

```
PROCEDURE Floyd (VAR A: ARRAY [1..n, 1..n] OF REAL;
                     c:  ARRAY [1..n,1..n] OF REAL;
                     P: ARRAY [1..n,1..n] OF INTEGER);
```

```
  VAR i, j, k :  INTEGER;
BEGIN
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO BEGIN
      A[i,j] := c[i,j];
      P[i,j] := 0;
    END;
  FOR i := 1 TO n DO A[i,i] := 0;
  FOR k := 1 TO n DO
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        IF A[i,k] + A[k,j] < A[i,j]
          THEN BEGIN
            A[i,j] := A[i,k] + A[k,j];
            P[i,j] := k
          END
END;
```

Komplexität:

- Programm-Struktur: Schleifen über i, j und k
- Zeit $O(|V|^3)$
- Platz $O(|V|^2)$

4.4.3 Warshall-Algorithmus (1962)

Spezialfall des Floyd-Algorithmus für 'unbewertete' Graphen:

$$c[v,w] = \begin{cases} \textbf{true} & \text{falls Kante } v \to w \text{ existiert} \\ \textbf{false} & \text{sonst.} \end{cases}$$

Es wird also die lediglich die Existenz einer Verbindung zwischen allen Knotenpaaren geprüft. Die Matrix A beschreibt hier die *transitive Hülle* des Graphen.

Definition: Zwei Knoten v und w eines ungerichteten Graphen heißen *verbunden*, wenn es einen Pfad von v nach w gibt.

Definition: Zwei Knoten v und w eines gerichteten Graphen heißen *stark verbunden*, wenn es einen Pfad von v nach w und einen Pfad von w nach v gibt.

Definition: Gegeben sei ein gerichteter Graph $G = (V, E)$. Die *transitive Hülle* (transitive closure) von G ist der Graph $\bar{G} = (V, \bar{E})$, wobei $\bar{E}$ definiert ist durch:

$$(v, w) \in \bar{E} \iff \text{es gibt in } G \text{ einen Pfad von } v \text{ nach } w.$$

Die entsprechende Modifkation des Floyd-Algorithmus ergibt den Algorithmus von Warshall, wobei statt der Real/Integer-Kosten-Matrix eine boolesche Matrix verwendet wird. Das Element $A[i, j]$ der booleschen Matrix gibt dabei an, ob ein Pfad von i nach j existiert.

```
PROCEDURE Warshall (VAR A: ARRAY [1..n,1..n] OF BOOLEAN;
                        c:  ARRAY [1..n,1..n] OF BOOLEAN);
  VAR i, j, k :  INTEGER;
BEGIN
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO A[i,j] := c[i,j];
  FOR k := 1 TO n DO
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        IF NOT A[i,j]
          THEN A[i,j] := A[i,k] AND A[k,j];
END;
```

4.5 Minimaler Spannbaum

Alternative Bezeichnungen: Minimum Spanning Tree (MST), Spannbaum mit minimalen Kosten,

ähnlich: Kruskal-Algorithmus 1957 (und auch Dijsktra nach Uminterpretation)

4.5.1 Definitionen

Gegeben sei ein ungerichteter Graph $G = (V, E)$:

- Ein Knotenpaar (v, w) heißt *verbunden*, wenn es in G einen Pfad von v nach w gibt. Der Graph G heißt verbunden (oder zusammenhängend), falls alle Knotenpaare verbunden sind.
- Ein verbundener Graph ohne Zyklen heißt *freier Baum*. Wählt man einen Knoten daraus als Wurzel, so erhält man einen (normalen) Baum.
- Ein *Spannbaum* ist ein freier Baum, der alle Knoten von G enthält und dessen Kanten eine Teilmenge von E bilden. Ist G bewertet, dann ergeben sich die Kosten eines Spannbaums von G als die Summe über die Kosten seiner Kanten.

Definition: Sei G ein ungerichteter, bewerteter Graph. Dann heißt ein Spannbaum, für den die Summe über die Kosten aller seiner Kanten minimal ist, *minimaler Spannbaum* (Minimum Spanning Tree) von G.

Anwendungen: Minimale Spannbäume haben relevante Anwendungen bei der Optimierung von Telefonnetzen, Straßennetzen und anderen Netzwerken.

Um einen minimalen Spannbaum zu einem gegebenen Graph zu berechnen, macht sich der **Prim-Algorithmus** (1957) eine MST-Eigenschaft (MST Property) zunutze, die im wesentlichen auf der additven Zerlegbarkeit der Kostenfunktion beruht:

4.5.2 MST Property

Sei $G = (V, E)$ ein verbundener, ungerichteter Graph mit der Bewertungsfunktion $c[v, w] \geq 0$ für eine Kante (v, w) und außerdem $U \subset V$.

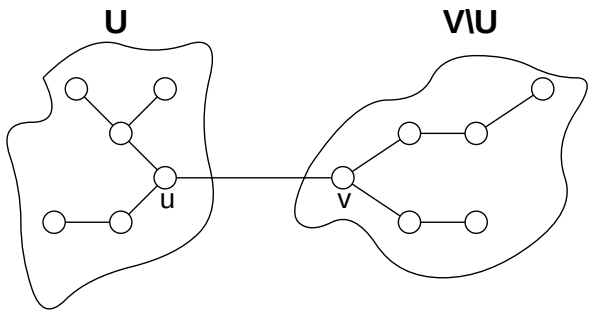
Gilt nun für die Kante (u, v)

$$(u, v) = \arg\min\{c[u', v'] : u' \in U, v' \in V \setminus U\}$$

dann gibt es einen minimalen Spannbaum, der die Kante (u, v) enthält.

Beweisidee:
durch Widerspruch (ähnlich Optimimalitätsprinzip; wiederum ist die Additivität der Kostenfunktion wesentlich):

Gälte die Aussage nicht, dann enthielte der minimale Spannbaum eine Kante (u', v'), die die beiden disjunkten Mengen verbindet und die „schlechter“ ist als (u, v). Ersetzt man diese Kante entsprechend, so erhält man einen „besseren“ Spannbaum. Das führt zu einem Widerspruch zu der Aussage, daß der erste Spannbaum schon minimal war.



Eigenschaft eines Minimalen Spannbaums

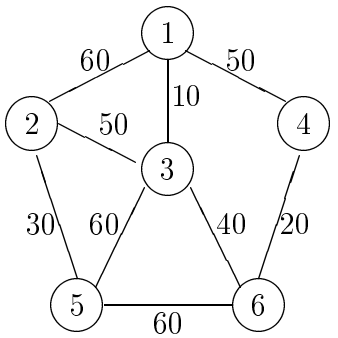
Die *MST property* besagt also, daß alle paarweise disjunkten Teilmengen eines minimalen Spannbaumes über eine „minimale“ Kante verbunden sind.

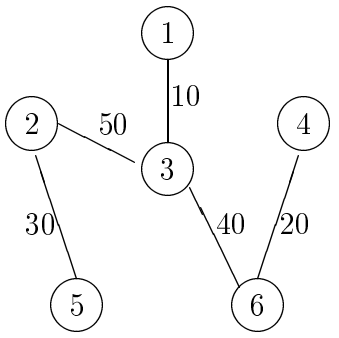
4.5.3 Prim-Algorithmus (1957)

Sei $V = \{1, ..., n\}$, T der zu konstruierende Minimalbaum und U die Menge seiner Knoten. Dann kann man den Prim-Algorithmus folgendermaßen formulieren:

```
Initialisierung von T := {} und U := {v_0};
WHILE (U ≠ V) do DO BEGIN
  Wähle beste Kante (u,v) (mit u ∈ U und v ∈ V \ U), die U und V \ U verbindet;
  U := U ∪ {v}; T := T ∪ {(u,v)}
END:
```

Beispiel





Um die beste Kante zwischen U und $V \setminus U$ zu finden, werden zwei Arrays definiert, die bei jedem Durchgang aktualisiert. Für $i \in V \setminus U$ gilt dann:

- CLOSEST[i] $:= \arg\min\{c[u,i] : u \in U\}$
- LOWCOST[i] $:= c[i, CLOSEST[i]]$

Der vorgestellte Code basiert auf dem Prim-Algorithmus aus [Aho et al. 83].

```
PROCEDURE Prim (C:ARRAY [1..n,1..n] OF REAL);
  VAR LOWCOST : ARRAY [1..n] OF REAL;
      CLOSEST : ARRAY [1..n] OF INTEGER;
      REACHED : ARRAY [1..n] OF BOOLEAN;
      i, j, k:  INTEGER;
      min :  REAL;
      { i and j are indices.  During a scan of the LOWCOST array, k is the
      index of the closest vertex found so far, and min = LOWCOST[k] }
BEGIN
  FOR i:= 2 TO n DO BEGIN
    { initialize with only vertex 1 in the set U }
    LOWCOST[i] := C[1,i];
    CLOSEST[i] := 1;
    REACHED[i] := FALSE;
  END;
  FOR i:= 2 TO n DO BEGIN
    { find the closest vertex k outside of U to some vertex in U }
    min := ∞;
    FOR j:= 2 TO n DO         {*}
      IF (REACHED[j]=FALSE) AND (LOWCOST[j] < min) THEN BEGIN
        min := LOWCOST[j];
```

```
        k := j;
      END;
    writeln (k, CLOSEST[k])       { print edge }
    REACHED[k] := TRUE;           { k is added to U }
    FOR j := 2 TO n DO
      IF (REACHED[j]=FALSE) AND (C[k,j] < LOWCOST[j]) THEN BEGIN
        LOWCOST[j] := C[k,j];
        CLOSEST[j] := k;
      END
  END
END; { Prim }
```

Komplexität

Der Code besteht im wesentlichen aus zwei verschachtelten Schleifen der Komplexität $O(|V|)$. Damit ergibt sich eine Zeitkomplexität von

$$O(|V|^2)$$

für den Prim-Algorithmus.

4.6 Zusammenfassung

Wir haben gesehen wie und in welcher Komplexität grundlegende Graph-Algorithmen arbeiten:

Single Source Best Path : Dijkstra-Algorithmus $O(|V|^2)$

All Pairs Best Path : Floyd/Warshall-Algorithmus $O(|V|^3)$

Minimaler Spannbaum (MST) : Prim-Algorithmus $O(|V|^2)$

Nicht untersucht haben wir das **Traveling Salesman Problem**. Hier sei nur erwähnt, daß man die Rundreise-Probleme mit einer Komplexität $O(n!)$ (bzw. $O((n-1)!)$ lösen kann. Mittels dynamischer Programmierung läßt sich diese Komplexität auf $O(n^2 \cdot 2^n)$ reduzieren. [Aigner]

Generell ist die Frage noch offen, ob das Traveling Salesman Problem auch in polynomieller Zeit (also in $O(n^k)$) gelöst werden kann.